

葵花宝典——
WPF
自学手册

李响 著



■名家推荐■

文字新颖流畅，讲解清晰准确，内容深入浅出，读者渐入佳境！

孙展波

微软总部WPF产品组成员，在微软技术大会多年主讲WPF及相关技术

构思独辟蹊径，对武林异类WPF的介绍徐徐道来，丝丝入扣。对初涉WPF者来说，可谓是“英雄莫问出处，编程即为利器”！

Y.John Chen

IBM高级企业架构师

一口气读完《葵花宝典——WPF自学手册》，还沉浸在绝情谷、桃花岛无限风光，黄蓉、小龙女娇羞美貌，乔峰、令狐冲的武功盖世之中，那些亦真亦幻的奇思妙想不由得让人热血沸腾，无穷回味！

我不得不佩服这幕编程大戏的导演，他创造性地将武侠与编程融会贯通，轻松地为编程学习者打通了任督二脉，学习编程居然可以如此的简单，奇才啊！

黎涛

北京超图软件股份有限公司专用软件事业部总经理

学习程序就像修炼上乘武功一样，得此宝典，你一定能练成WPF神功，早日登上华山之巅。

陈荣国

中国科学院地理科学与资源研究所研究员

之前一直用WPF做项目，零零星星地看了一些WPF编程方面的内容。机缘巧合有幸遇到了这本武侠奇书^_^，刚开始只是觉得很有意思，后来发现书里总是可以找到熟悉的武侠类比依赖属性、路由事件等复杂的概念。我服了，挺好，这本书！

刘佳升

中国科学院软件研究所博士生

上架建议：WPF



ISBN 978-7-121-11405-2



9 787121 114052 >

定价：79.00元（含DVD光盘1张）



责任编辑：孙学瑛
责任美编：李玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

葵花宝典——

WPF

自学手册

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

内 容 简 介

这本书最大的作用是让从未接触过 Microsoft Windows Presentation Foundation 的读者能够从初学到精通掌握，运用 WPF 来进行桌面开发，而且这本书的叙事风格和手法使得读者在经历掌握 Microsoft WPF 开发的整个过程是如此轻松快乐，在作者风趣调侃的语言当中不知不觉地学会 WPF 开发。本书从 WPF 的相关工具开始讲起，从 WPF 的体系结构、XAML、依赖属性、路由事件、命令等方面为读者奠定了一个坚实的学习基础。之后就切入了应用程序窗口、页面导航、布局等起步应用，能让读者及时地体会到学习的成就感和乐趣。接下来的控件、样式、数据绑定、二维图形、动画等相关内容则能够为读者的 WPF 技术提升到一个比较高的层次，如同插上翅膀，自由翱翔。

这本书对于 WPF 核心技术的原理、概念、技术、技巧与开发实践的讲述，是基于一位完全不懂 WPF 的菜鸟学习经历的，非常符合国内程序员 WPF 技术的初学路线，如果您想学习 Microsoft WPF 技术的话，那么这本书将是您的不二选择。

图书在版编目（CIP）数据

葵花宝典：WPF 自学手册 / 李响著. — 北京：电子工业出版社，2010.9

ISBN 978-7-121-11405-2

I. ①葵… II. ①李… III. ①窗口软件，Windows Vista—用户界面—程序设计—技术手册 IV. ①TP316.7-62

中国版本图书馆 CIP 数据核字（2010）第 138130 号

责任编辑：孙学瑛

印 刷：北京天宇星印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：860×1092 1/16 印张：39.25 字数：897 千字

印 次：2010 年 9 月第 1 次印刷

印 数：4000 册 定价：79.00 元（含 DVD 光盘 1 张）

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

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

服务热线：(010) 88258888。

楔 子

江湖中盛传一部奇书，谁要依此书修炼武功就能“号令群雄，莫敢不从”，此书正是天下第一奇书——《葵花宝典》。

天下武功都是循序渐进，越到后来越难。但此书却违背常理，最难的是第一关。只要打通这一关，以后就遇山开山，遇水架桥，佛来斩佛，魔来斩魔。而这第一关则是“欲练神功，引刀自宫”，据说有人挥泪自宫之后成为了传说中的“东方不败”。但是也有不幸的人是在地摊上花上五毛钱买的盗版《葵花宝典》，翻开第1页是“欲练神功，必先自宫”，于是依言挥刀自宫。血还未止，忍疼翻到第2页，又是8个大字“即使自宫，未必成功”映入眼帘。此时后悔已经来不及了，只好继续翻到第3页，还是8个大字“如不自宫，也能成功”，偶的神啊！

废话不能再多说了，本书的头号主角就要闪亮登场。此人姓木，单名一个木，全名木木，呆若木鸡的木。大学二年级，好金庸，喜武侠；除此之外，对计算机编程有几分兴趣，也有几分傻气。学了一点C++和C#语言，有过一些MFC及WinForm编程的经验。闲时没事会在旧书市场闲逛，妄想淘得一本程序上的武功秘籍，一举成为Bill Gates那样的首富。“到那时候，嘿，嘿……”木木干笑了几声：“一定要开宝马车去校园门口的鸡蛋饼摊买鸡蛋饼，或者在街头坐着一个小马扎吃热干面，不过是百元大钞摞起来的小马扎而已……”

这一次还是在旧书市场闲逛，木木在众多的“贪官和他的情妇们”、“二十位都市男女情感口述”等此类杂志中发现了一本泛黄的书，封面上写着“葵花”两字。抽出来一看，才发现下面还有“宝典”二字，完整起来就是传说中的“葵花宝典”。木木不禁心扑腾扑腾乱跳，再一细看，下面还有一行小字“WPF自学手册”。以前隐约听说过WPF，应该是.NET里面的新技术，看来这本书就是传说中程序的“武功秘笈”了。旧书都用不了多少钱的，木木丢给了老板五元钱后，就像做贼一样将书揣在怀里回去了。

回到家里，木木备好了剪刀（自宫而用），又放了一张女友靓照（看看也许会后悔）。闭上眼，颤抖地翻开了第1页。慢慢睁开眼，上面赫然写着几个大字“定价：3.9元”。不禁大骂老板黑心，新书才3.9元，旧书居然卖给我5元。气愤之余不忘翻开书的第2页，果然有8个大字“欲练神功，必先自功”。木木早就听说过盗版葵花宝典的故事，于是继续往下翻，直到最后一页也没有“未必

成功、无需自宫”这样的字眼。宫还是不宫？木木左手拿剪刀，右手拿女友靓照，还真是难以抉择啊。

等等，木木又翻回第 2 页，揉揉眼再看了看，是成功的功，而不是自宫的宫。“唉，又是盗版书，‘宫’字还印错了。依法修炼，肯定不能成功了？”木木心想。于是将剪刀和靓照都抛去，拿着书躺在床上权当小杂志读读。谁知翻到第 3 页，该书却写到：“你宫了么？若挥刀自宫，请立刻拨打 120，兴许有救。此功非彼宫，学习 WPF 需要用功、用功再用功。因此欲练神功，必先自功。”木木腾地一下从床上跳起来，拿着书一屁股坐到书桌边，看来这的确是一本程序的武功秘笈，而且自功不自宫，相当划算喔。

诸位看到这儿，一定也获得这本武功秘籍了吧。那么请你们和木木一起走进 WPF 的世界，一定要“自功”才能学到真正的 WPF 哟^_^。

目 录

第一卷 程序江湖

第1章 上路吧，WPF	
1.1 江湖前传	2
1.1.1 微软的四重门	2
1.1.2 DirectX——无心插柳柳成荫	4
1.2 WPF来了	4
1.2.1 七十二变	5
1.2.2 WPF 的与众不同之处	8
1.3 接下来做什么	9
参考文献	10
第2章 WPF相关工具——十八般兵器	
2.1 Microsoft Visual Studio 2010	12
2.1.1 13年间	12
2.1.2 认识Visual Studio 2010	13
2.2 命令行和记事本——小米加步枪	17
2.2.1 编译简单的C#程序	18
2.2.2 引用外部程序集	19
2.2.3 编译WPF应用程序	20
2.3 Microsoft Expression Blend	23
2.3.1 优势	23
2.3.2 组成	23
2.4 XamlPad	24
2.5 Reflector	26
2.6 接下来做什么	27
参考文献	27
第3章 WPF体系结构——藏宝图	
3.1 Windows体系结构	28
3.2 WPF内部结构	30
3.2.1 切入点之一：托管和非托管的界限	30
3.2.2 切入点之二：WPF如何实现绘制	30
3.2.3 切入点之三：WPF类层次结构	33
参考文献	36

第二卷 心 法

第4章 XAML——反两仪刀法和正两仪剑法	
4.1 从C#到XAML	39
4.2 命名空间及其映射	43
4.2.1 WPF的命名空间	43
4.2.2 XAML的命名空间	45
4.2.3 其他命名空间	46
4.3 简单属性和附加属性	49
4.3.1 简单属性	49
4.3.2 附加属性	50
4.4 Content属性	51
4.5 类型转换器	53
4.5.1 功能	53
4.5.2 自定义类型转换器	54
4.6 标记扩展	56
4.7 分别使用XAML和C#构建应用程序——刀还是刀，剑还是剑	57
4.7.1 XAML——反两仪刀法	57
4.7.2 C#——正两仪剑法	59

4.8 使用 XAML 和 C# 构建应用程序——	
刀剑合璧	60
4.8.1 第 1 次刀剑合璧	61
4.8.2 完美的刀剑合璧	63
4.8.3 还有一种方法——在 XAML 中 嵌入代码	67
4.9 接下来做什么	68
参考文献	68
6.2 路由事件的定义	104
6.3 路由事件的作用	106
6.4 路由事件	108
6.4.1 识别路由事件	108
6.4.2 路由事件的旅行	109
6.5 路由事件示例	113
6.6 接下来做什么	116
参考文献	116

第 5 章 依赖属性——木木的“汗血宝马”

5.1 属性与依赖	69
5.2 认识依赖属性	72
5.2.1 分辨依赖属性	72
5.2.2 引入依赖属性的原因	73
5.2.3 依赖属性的组成部分	82
5.3 自定义依赖属性	83
5.3.1 何时需要自定义一个依赖 属性	83
5.3.2 自定义依赖属性示例	84
5.4 所有规则大排队	90
5.4.1 按钮到底是什么颜色	90
5.4.2 依赖属性设置优先级列表	91
5.4.3 验证优先级的示例	92
5.5 附加属性和“等餐号”	95
5.5.1 如果没有附加属性	96
5.5.2 附加属性的本质	96
5.6 接下来做什么	97
参考文献	98

第 6 章 路由事件——绝情谷底玉蜂飞

6.1 从玉蜂说起，回顾.NET 事件模型	99
-----------------------------	----

6.2 路由事件的定义	104
6.3 路由事件的作用	106
6.4 路由事件	108
6.4.1 识别路由事件	108
6.4.2 路由事件的旅行	109
6.5 路由事件示例	113
6.6 接下来做什么	116
参考文献	116

第 7 章 WPF 的命令（Command）—— 明教的圣火令

7.1 木木的写字板（无 Command）	117
7.1.1 简单的写字板原型	118
7.1.2 右键菜单和快捷键	120
7.1.3 控制功能状态	121
7.1.4 小徐的写字板为何如此 简单	124
7.2 小徐的写字板（有 Command）	126
7.3 Command 的作用	128
7.4 WPF 的 Command 模型	129
7.4.1 Command——圣火令	130
7.4.2 Command Sources—— 明教教主	132
7.4.3 Command Binding—— 波斯三使	132
7.4.4 Command Target—— 金毛狮王	133
7.5 接下来做什么	133
参考文献	134

第三卷 小有所成

第 8 章 应用程序窗口——大侠的成长路线

8.1 新建一个应用程序	136
8.1.1 手动创建	136
8.1.2 使用向导创建	139
8.2 应用程序及其生命周期	139
8.2.1 小强的成长路线图	139

8.2.2 应用程序的生命周期	140
8.3 窗口	145
8.3.1 窗口组成	146
8.3.2 窗口的生命周期	146
8.3.3 窗口属性	149
8.3.4 非规则窗口	155
8.4 接下来做什么	158

参考文献	158
第 9 章 页面和导航——天罡北斗阵演绎	
9.1 导航应用程序演绎	159
9.1.1 第 3 类应用程序	159
9.1.2 两种形式	160
9.1.3 4 个核心	160
9.2 页面	161
9.2.1 Page	161
9.2.2 Page 的宿主窗口	163
9.3 导航连接	164
9.3.1 超链接	164
9.3.2 通过编程导航	166
9.3.3 其他方式导航	168
9.4 历史管理	169
9.5 导航和 Page 的生命周期	171
9.5.1 这一“点击”的背后	171
9.5.2 Page 的生命周期	177
9.6 页面状态保留和数据传递	177
9.6.1 构建登录应用程序	179
9.6.2 由前向后传递数据	181
9.6.3 WPF 固有的页面状态保留机制	183
9.6.4 使用依赖属性保留简单的页面状态信息	183
9.6.5 由后向前传递数据方法的 PageFunction	185
9.6.6 使用 IProvideCustomContentState 接口保留复杂的页面状态信息	188
9.7 XAML 浏览器应用程序	192
9.7.1 将一个基于窗口的导航程序 变换成 XBAP 程序——乾坤大挪移	193
9.7.2 XAML 浏览器应用程序小结	194
9.8 接下来做什么	196
参考文献	196
第 10 章 布局——药师的桃花岛	
10.1 憨木木误闯桃花宝岛	197
10.2 老顽童试解桃花玄机	198
10.2.1 Canvas	199
10.2.2 StackPanel	200
10.2.3 WrapPanel	202
10.2.4 DockPanel	203
10.2.5 Grid	205
10.3 黄岛主演绎布局精妙	210
10.3.1 桃树林的属性	210
10.3.2 自定义布局	213
10.4 接下来做什么	216
参考文献	216
第 11 章 控件与 Content——北冥神功	
11.1 缘起	218
11.2 Content 模型及其家族	219
11.2.1 Content 模型	219
11.2.2 Content 家族	220
11.3 经典控件	222
11.3.1 Content 控件	222
11.3.2 HeaderedContent 控件	226
11.3.3 Items 控件	230
11.3.4 Range 控件	238
11.4 接下来做什么	242
参考文献	243

第四卷 峰回路转 夯实基础

第 12 章 资源——雪山宝藏	
12.1 程序集资源	245
12.1.1 资源文件	246
12.1.2 内容文件	248
12.1.3 Site of Origin 文件	250
12.2 URI 语法	250
12.2.1 WPF 中的 URI	251
12.2.2 一个全面的 URI 用法示例	251
12.2.3 WPF 中的 URI 处理顺序	253
12.3 逻辑资源	254
12.3.1 静态资源和动态资源	255
12.3.2 系统资源	257

12.3.3 共享资源	259	
12.3.4 通过代码定义和访问资源 ..	259	
12.3.5 使用 ResourceDictionary 组织资源	260	
12.3.6 在程序集之间共享资源	262	
12.4 接下来做什么	264	
参考文献	265	
参考文献		289

第 13 章 样式和控件模板——听香水榭，千变阿朱

13.1 样式那一点事儿	267
13.1.1 何来样式	267
13.1.2 基本用法	269
13.1.3 触发器	270
13.2 模板示例——听香水榭边，须发如银人	273
13.3 模板工作原理——淡淡少女香，侃侃孙三谈	276
13.3.1 模板绑定和模板触发器	279
13.3.2 其他修改	279
13.4 控件模板的浏览器程序——龙钟老太太，妙龄俏阿朱	280
13.5 样式、模板和换肤——阿朱技高超，木木向来痴	285
13.5.1 混合使用	285
13.5.2 组织模板资源和更换皮肤 ..	286
13.6 接下来做什么	289

参考文献	289
------------	-----

第 14 章 数据绑定——桃花岛软件公司人员管理系统之始末

缘起	290
14.1 人员管理系统	290
14.1.1 浏览和修改人员信息（无数据绑定）	290
14.1.2 数据绑定（木木，老婆喊你回家吃饭）	294
14.1.3 使用数据绑定	294
14.2 数据绑定基础	296
14.2.1 数据绑定模型	296
14.2.2 数据绑定的方向	297
14.2.3 数据绑定的触发条件	299
14.2.4 绑定数据源的 4 种方式 ..	301
14.2.5 值转换	302
14.2.6 数据验证	303
14.3 高级主题——与数据集合绑定	307
14.3.1 实现一个数据源集合	307
14.3.2 绑定目标和集合	308
14.3.3 数据模板	309
14.3.4 集合视图	311
14.4 后记	315
14.5 接下来做什么	315
参考文献	315

第五卷 紫杉红烛

第 15 章 奇妙的二维图形世界——面壁

15.1 面壁	317
15.2 二维图形的数学基础（第一块石壁）	319
15.2.1 分辨率无关	319
15.2.2 坐标系	324
15.2.3 点和向量	326
15.2.4 几何变换	330
15.2.5 齐次坐标	333
15.2.6 WPF 中的对象变换	341
15.3 WPF 的二维图形架构（第二块石壁）	342

15.3.1 立即模式和保留模式	343
15.3.2 WPF 二维图形体系结构	350
15.3.3 WPF 二维图形的重要元素 ..	352
15.3.4 书架上到底放什么书	355
15.4 颜色和画刷（第一本书）	356
15.4.1 颜色	356
15.4.2 画刷	359
15.4.3 使用画刷制作特效	369
15.5 Shape（第二本书）	372
15.5.1 简单的 Shape 元素	373
15.5.2 线型、线帽、线的连接和填充规则	376
15.5.3 放置并调整 Shape 大小 ..	380

15.5.4 Path	382
15.6 Geometry (第三本书)	383
15.6.1 理解 Geometry	383
15.6.2 简单的 Geometry 类型	384
15.6.3 GeometryGroup 和 CombineGeometry	386
15.6.4 PathGeometry 和 StreamGeometry	387
15.6.5 路径描述语言	394
15.7 Drawing 和 Visual	395
15.7.1 Drawing 及其派生类	395
15.7.2 Drawing 类型	396
15.7.3 Visual	400
15.8 接下来做什么 (面壁之后)	406
参考文献	406

第 16 章 动画——降龙的最后一掌

16.1 七公和他的降龙十八掌	407
16.2 WPF 实现动画的方式	408
16.2.1 基于计时器的动画	408
16.2.2 基于帧的动画	410
16.2.3 基于属性的动画	411
16.3 WPF 动画的基本知识	411
16.3.1 前提条件	411
16.3.2 动画类的类层次结构	412
16.3.3 时间线的基本行为	414
16.4 3 种基本类型动画	422
16.4.1 From/To/By 类型动画	422
16.4.2 KeyFrame 类型动画	423
16.4.3 Path 类型动画	428
16.5 动画的交互控制	431
16.6 后记：降龙的最后一掌	432
16.7 接下来做什么	434
参考文献	434

第 17 章 WPF3D 图形

17.1 WPF3D 引言	435
17.1.1 WPF3D 图形的作用	435
17.1.2 用 2D 图形产生立体感	437
17.1.3 WPF3D 类概览	440
17.2 WPF3D 数学基础	444
17.2.1 坐标系	444

17.2.2 空间点	445
17.2.3 向量	446
17.2.4 矩阵和几何变换	451
17.3 从 3D 物体到 2D 图形	454
17.3.1 3 个坐标系	455
17.3.2 Camera 对象	457
17.3.3 坐标变换	462
17.4 基本几何体	471
17.4.1 使用直线 ScreenSpaceLines3D	471
17.4.2 构建立方体	473
17.4.3 构建球面	475
17.5 光源和材质	479
17.5.1 光源	479
17.5.2 着色和法线	483
17.5.3 计算 DiffuseMaterial 和表面颜色	487
17.5.4 其他材质	488
17.5.5 纹理	490
17.6 动画和交互	493
17.6.1 动画	493
17.6.2 交互	496
17.7 接下来做什么	500
参考文献	500

第 18 章 文本和文档——从黑风双煞的“练门”说起

18.1 从 TextElement 说起	502
18.1.1 文本	502
18.1.2 TextElement	503
18.1.3 TextElement 的属性	504
18.2 TextBlock 控件	509
18.2.1 与文本相关的属性	509
18.2.2 文本属性	513
18.2.3 其他简单的文本控件	515
18.3 理解 WPF 的文档	515
18.3.1 ContentElement	515
18.3.2 流文档模型	516
18.3.3 固定文档	523
18.4 文档控件	524
18.4.1 固定文档的浏览控件	524
18.4.2 流文档的浏览控件	526
18.4.3 注释功能	528

18.5 实现一个简单的文档浏览器.....	531	18.5.4 实现缩略图功能	543
18.5.1 应用程序组成	531	18.5.5 实现书签和标注功能	545
18.5.2 打开一个流文档	532	18.6 接下来做什么	550
18.5.3 另存为不同格式的文件	535	参考文献	550

第六卷 华山之巅

第 19 章 互操作——“小无相功”

19.1 为什么需要互操作?	553
19.2 互操作的几种类型	553
19.3 Windows Forms 和 WPF.....	554
19.3.1 对话框	554
19.3.2 在同一个窗口中混合 WPF 和 WinForm 内容	558
19.4 在 Win32 中嵌入 WPF 内容	564
19.4.1 现有的 Win32 程序	564
19.4.2 使用 WPF 制作钟表	568
19.4.3 将 WPF 内容嵌入在 Win32 程序中	569
19.5 在 WPF 中嵌入 Win32 内容	574
19.5.1 一个 Win32 的 DLL 工程.....	574
19.5.2 使用 HwndHost	577
19.5.3 支持键盘导航	580
19.6 接下来做什么	585
参考文献	585

第 20 章 自定义控件——出手无招，何招可破

20.1 风老前辈登场	586
20.2 用 RadioButton 实现红绿灯	588
20.3 何时自定义控件?	590
20.3.1 不要被控件的外观所欺骗， 要考虑其内在本质	590
20.3.2 Content 模型、模板和 附加属性	591
20.3.3 使用附加属性扩展现有 控件	592
20.4 自定义控件	598
20.4.1 自定义控件的 3 个层次	599
20.4.2 从 UserControl 开始	600
20.5 无外观控件	603
20.5.1 无形才是有形	603
20.5.2 定义命令	605
20.5.3 在主题中定义控件外观	606
20.6 接下来做什么	609
参考文献	609

第七卷 志向无限大

第 21 章 木木能行，我也行

葵花宝典的真正秘密	611
写给大学生	612
致谢	613

附录 A WPF 类图

程序江湖

第一卷



第1章

上路吧，WPF

沧海一声笑，滔滔两岸潮，浮沉随浪只记今朝。苍天笑，纷纷世上潮，谁负谁胜出天知晓。
江山笑，烟雨遥，涛浪淘尽红尘俗事几多骄。清风笑，竟惹寂寥，豪情还剩了，一襟晚照……

——《笑傲江湖，词曲：黄霑编》

程序也有江湖，回首且看有多少公司的兴衰，多少语言的纷争，多少平台的没落。

木木即将要接触的 Windows Presentation Foundation (WPF) 是微软新一代的桌面平台技术，它是程序江湖的“锦带吴钩，把酒临风，潇湘之水，盈盈红烛的三生泪”。

本章内容如下。

- (1) 江湖前传。
- (2) WPF 来了。
- (3) 接下来做什么。

1.1 江湖前传

1.1.1 微软的四重门

了解 WPF，先要了解其之前的江湖。了解江湖，先要了解在软件领域里横行 20 余年的江湖盟主——微软。在这 20 多年里，微软力主推行的编程语言和平台 API 大致经历了四重门^[1]，如图 1-1 所示。

(1) 第一重门 (1985~1991)：C 语言和 Windows API。新生代的 .Net 开发人员可能从未接触过注册一个窗口类，创建一个窗口或者在窗口过程函数中不断地 switch...case...case...。如果有时间的话，还是有必要了解 Windows 编程模型原理。为此笔者慎重推荐 Charles Peztold 所著的《Windows 程序设计》，无人能出其右¹。

¹ 也可以参见笔者的博文《永远的窗口》一文，放在脚注位置，实在是不敢和 Charles Peztold 大师比肩。

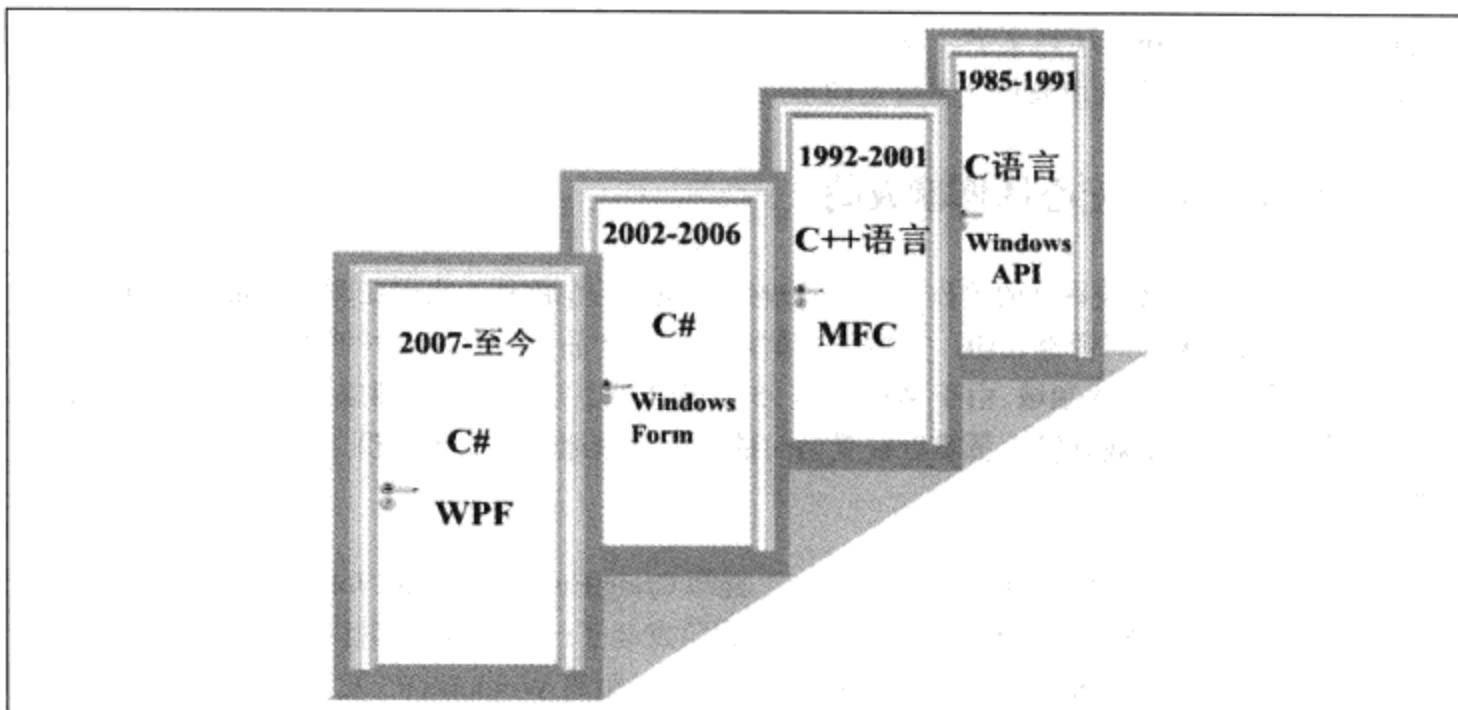


图 1-1 微软的四重门

(2) 第二重门（1992~2001）：C++语言和 Microsoft Foundation Class（MFC）。尽管 MFC 一直以来被人诟病，但其至今却是历史上使用人数最多的 Windows 编程方法。如果想要掌握 MFC 的编程方法，那么一本大部头的《MFC 程序设计》可能是必需的；如果想要深入了解 MFC 的原理，那么 20 世纪风靡一时的侯大师的《深入浅出 MFC》可能作为案头书十分合适。当然还有一部《MFC Internals:Inside the Microsoft Foundation Class Architecture》（中文名是《深入解析 MFC》）是探求 MFC 源码的最佳书籍。

(3) 第三重门（2002~2006）：C#和 Windows Form。C#是伴随微软的.NET 大战略产生新的语言，也是微软对抗 Java 的最有力武器。这个时期 IT 的技术焦点从桌面转向了网络，桌面开发显得有些冷清。Charles Peztold 此时推出了一本新书，即《C# Windows 程序设计》。但是这时已经不是“一家独大”，而是“城头变换大王旗，你方唱罢我登场”的百家争鸣时期。

(4) 第四重门（2007 至今）：C#和 WPF。WPF 伴随着 Windows Vista 推出，虽然 Windows Vista 已经被逐渐证实只是一个过渡产品，但是 WPF 作为一个桌面开发平台并没有被摒弃。不过 Windows Vista 的失利，还是在一定程度上影响了 WPF 的推广。WPF 在 2006 年年底发布，但时至今日，仍未成燎原之势。如果希望在 Windows 平台上快速地打造够酷够炫的应用程序，则非 WPF 莫属。

无论是前三重门的哪一重门，Windows 开发人员从本质上都是在使用两种技术——User 和 GDI 子系统。一个 Windows 窗口实际上只是做两件事情，一是等待用户的输入，可能是鼠标、键盘、手写笔，以及单击菜单等，然后做出恰当的反应；二是在窗口上渲染图形。前者使用 User 子系统，后者使用 GDI 子系统，MFC 使用的正是这两个子系统。到了 Windows Form 阶段，微软在 GDI 的基础上封装成 GDI+，但是 GDI+ 绘制归根结底还是 GDI，因此 Windows Form 也是使用的这两个子系统。

第四重门的 WPF 几乎改变了原有的 Windows 技术，之所以是“几乎”，因为其仍然使用 User 子系统，但是绘制图形则交给了新的图形平台 DirectX，准确地说是 DirectX 的一部分 Direct3D。这意味着

着无论是绘制一个按钮，还是文字，或者大型的三维场景，它们都会转换为 3D 三角形、材质和其他 Direct3D 对象，并直接由硬件（主要是图形处理卡）渲染。

1.1.2 DirectX——无心插柳柳成荫

DirectX 曾经一度是并不被看好的项目^[3]，早在 1994 年，微软的全部精力集中在即将发布的 Windows 95 上。从各个领域传来的声音都表明，它将是一款划时代的产品。但是在一片叫好声中也存在一些不和谐的音符，主要来自游戏领域。由于 Windows 95 在硬件之上建立的层层抽象限制了对设备的访问，所以对于性能敏感的游戏开发人员难以接受当时的游戏玩家通常会保留一份 DOS 系统来玩高质量的视频游戏。

事情的转变发生在 1994 年下半年，微软的 3 名工程师 Craig Eisler、Alex St. John 和 Eric Engstrom 不甘心让 Windows 95 沦落为二流的游戏平台。他们冒险发起了一个项目，其目标是让 Windows 95 平台上的开发人员也能全速访问硬件设备。这个项目的发起距离 Windows 95 发布的时间只有几个月，因此被视为一个可有可无的项目。就在这样的窘况之下，DirectX 在 Windows 95 发布仅一个月后就以“Windows Games SDK”名义发布了。

最初的 DirectX 并没有和当时大名鼎鼎的 OpenGL 构成明显的竞争关系，二者分工明确。DirectX 对硬件要求低，对应普通家用 PC 配置，长于开发 2D 游戏；OpenGL 对硬件要求高，对应高端图形工作站，长于 3D 图形。从技术上来说，当时的微软正处在 COM 的热潮中，所有的组件都要 COM 化，DirectX 也不例外。DirectX 以 COM 组件的方式暴露 API，与 OpenGL 这种平民化的 C 风格 API 相比显得格外烦琐。

DirectX 3.0 发布不久，微软发布了一个更新的版本，内部版本号从 4.04.00.0068 增加到 4.04.00.0069。这样微小的更新一般人是无法察觉到的，但是非常具有讽刺意义。这次更新实际上是一次对整个产业具有决定影响的重大事件，因为在更新版本中出现了一个称为“Direct3D”的组件，正是这个组件揭开了 DirectX 与 OpenGL 正式对抗的序幕。在随后不到 5 年的时间里，Direct3D 风卷残云，成为主宰整个 PC 游戏图形工业的标准，彻底打破了 OpenGL 一家独大的局面。

DirectX 升级到 9 时，COM 渐渐落幕，时代已经进入到 .Net。微软用托管代码包装了 DirectX（称为托管“DirectX”，与之对应的则是非托管 DirectX），以使其可以应用到基于 .Net 架构的程序开发中。这时微软已经认识到可以将 DirectX 作为底层实现传统的界面设计，从而跳出过去 GDI/GDI+ 的限制。于是，WPF 来了……

1.2 WPF 来了

2001 年微软成立了一个新的团队，它有一个听起来简单，却很宏伟的使命。即创建一个统一的界面呈现平台，最终替代微软现有的 UI 平台，并能提供新的开发方案以满足用户日益增长的用户体验需求。起初这个团队将团队和项目都命名为“Aralon”，此即后来的 WPF。

1.2.1 七十二变

“嘘……可是我还是什么都没看到。”木木已经有些不耐烦了，随手把书扔在一边。谁知叮咣一声，从书的中间掉下来了一张光盘。“葵花宝典还配光盘啊。”木木知道，按照以往的常识，这个光盘应该是随书附带的源码。但是毕竟这是一本葵花宝典，说不定会有一些蹊跷。忍不住好奇，将光盘插入到电脑的光驱中。

“咔，暂停！”由于全书的示例均是在木木的笔记本电脑上编写的，因此不得不介绍这个头号道具，木木的笔记本电脑为 IBM T400，配置如下。

- (1) CPU 为双核，主频 2.26 GHz。
- (2) 内存为 2 GB。
- (3) 显卡为 ATI Mobility Radeon HD 3400 Series。
- (4) 操作系统为 Windows XP SP3。
- (5) 编程工具为 VS2010、Reflector 和 XAMLPad。

木木满怀期待地以为有什么奇迹发生，光驱在“咔、咔、咔”地读盘，窗口弹了出来。唉，还是随书附带的源码，葵花宝典不过如此。既然打开了，那么还是看看吧。

源码第 1 章中的其他名称都很普通，唯有一个“看我七十二变”的文件夹听名字稍稍吸引人一些。于是木木点开这个文件夹，这是一个在 VS 2010 下运行的一个项目。运行后的界面非常简单，右侧有 6 个按钮，左侧则为空空荡荡，如图 1-2 所示。

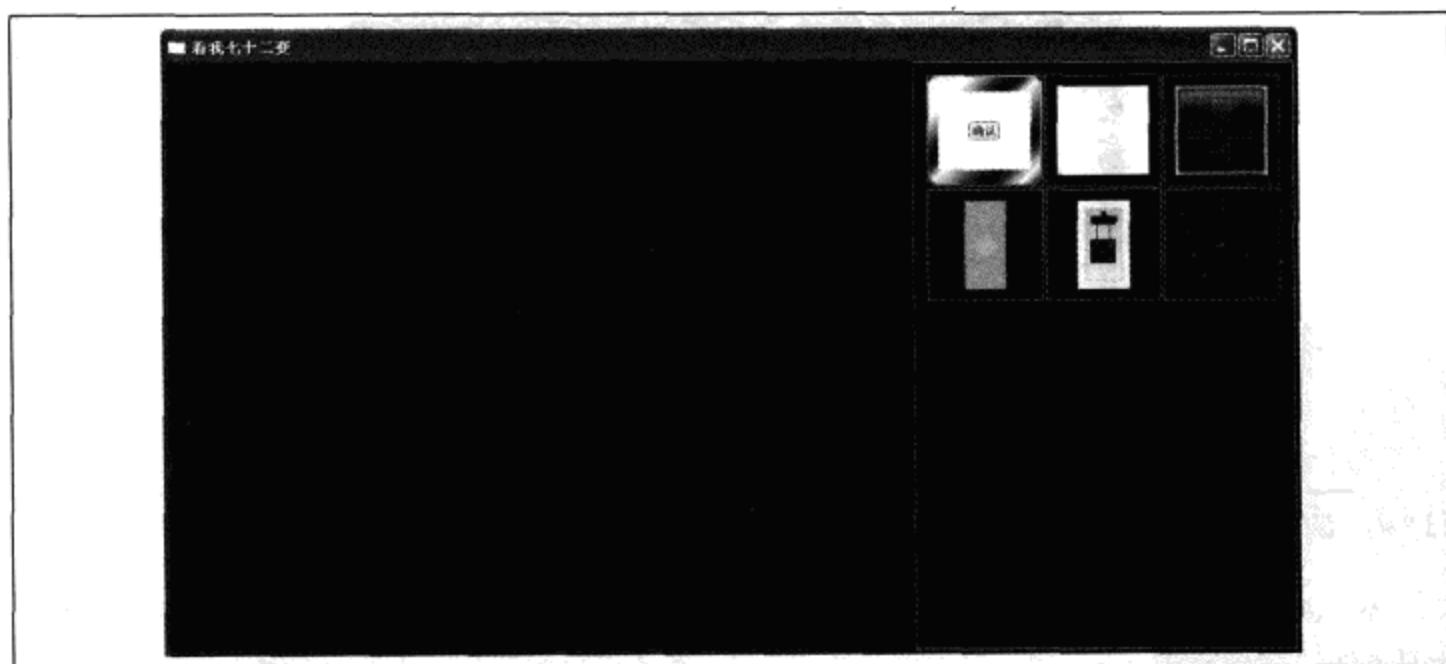


图 1-2 “看我七十二变”运行界面

木木单击右侧的第一个按钮，按钮旋转一下后弹出其形状。这是一个普通的按钮，不过分为两个页

面。即一个按钮及其代码实现，代码语言类似 XML，它的名称叫做 XAML，是微软为设计 UI 界面所提供的一套新语言，如图 1-3 所示。

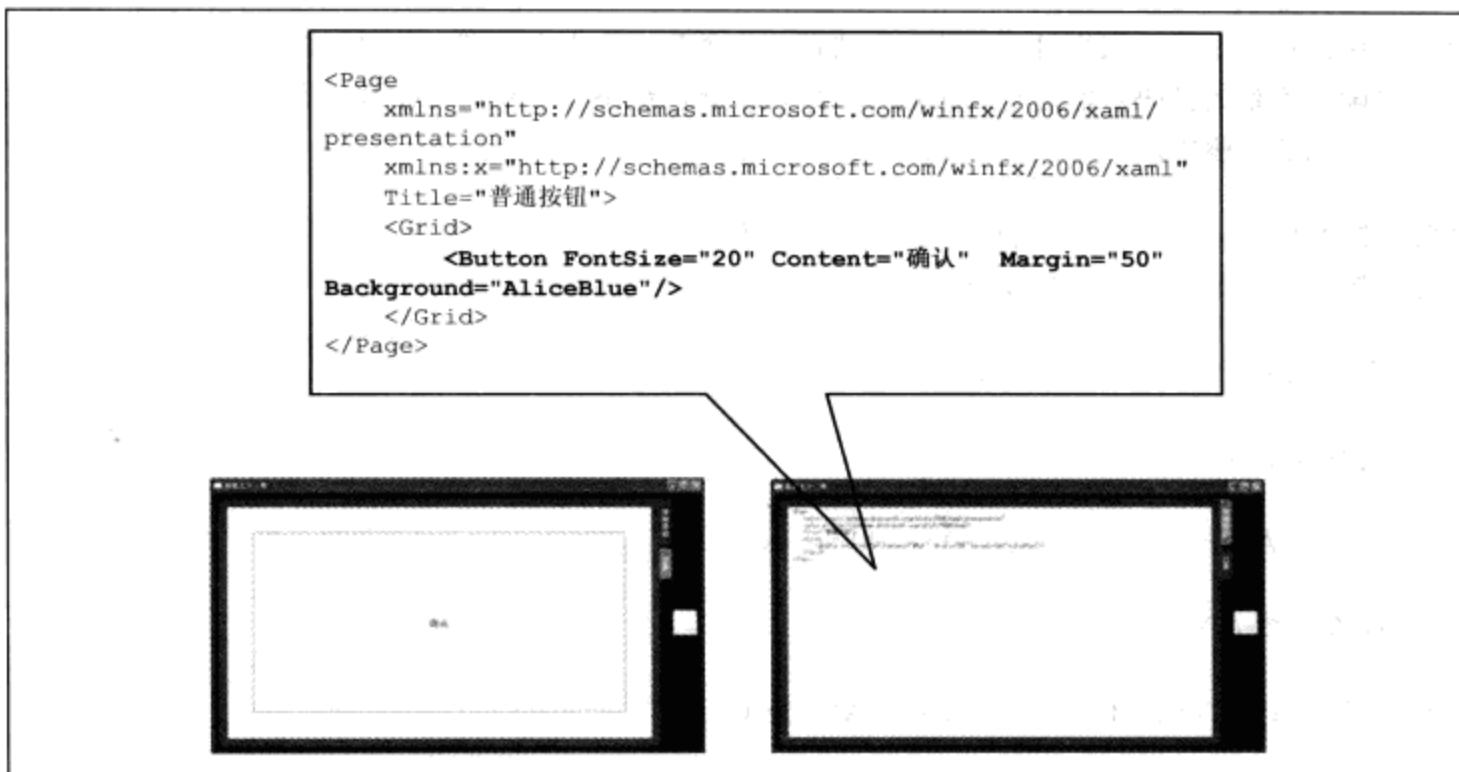


图 1-3 第 1 个按钮界面及其代码语言

“这个也太普通了吧”，木木顿时感到非常扫兴。原来听说 WPF 如何炫目，也不过如此。“反正既然点开了索性看完”木木心想，于是点开第 2 个按钮。没想到这是一个 3D 按钮，暗灰色调，如图 1-4 所示。单击后可以旋转，但是代码太长，木木也无心看下去。

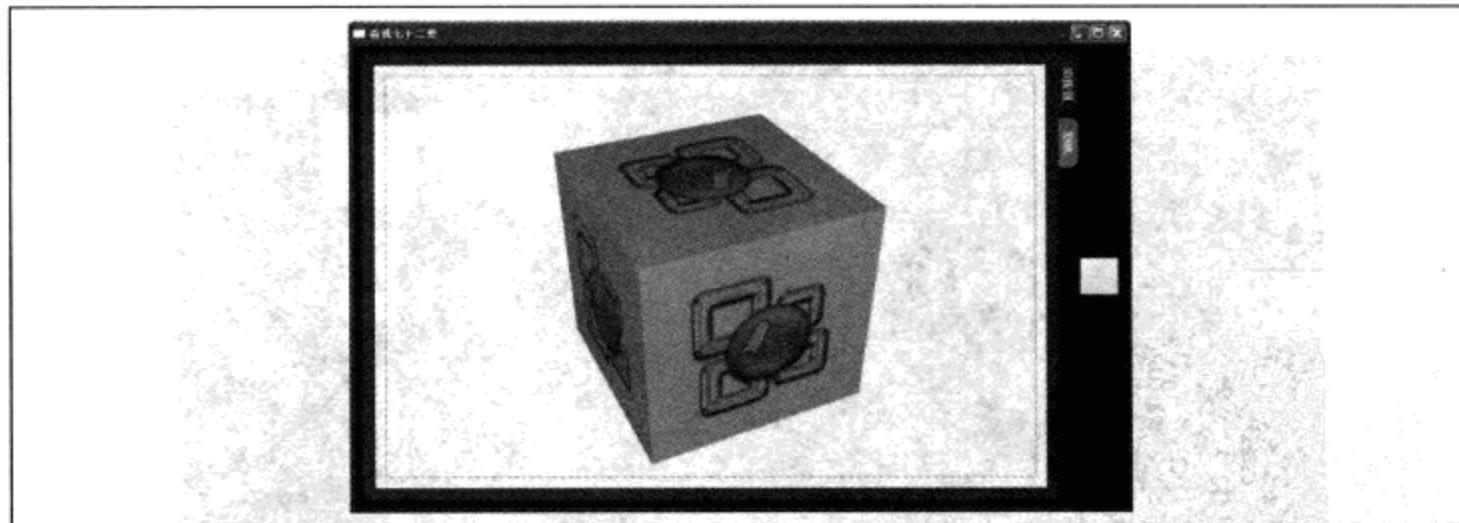


图 1-4 3D 按钮

第 3 个，木木开始有些期待了，是一个简单的十字形。但是鼠标放上后会突出放大，这样用户体验倒是不错。试着单击一下，也可以发出声音和变亮，最难能可贵的是它还有一种倒影的效果。木木以前接触过一些控件编程，知道在过去的 VC++ 程序中设计一个这样的按钮颇费工夫，心里开始隐隐觉得 WPF 还真是个好东东。

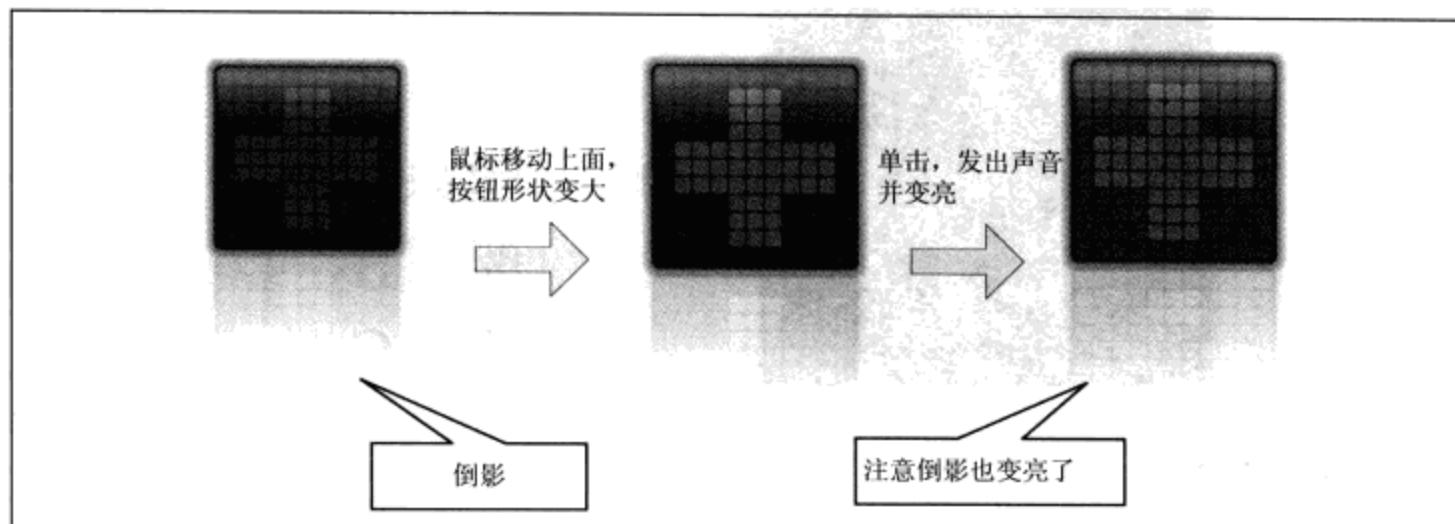


图 1-5 第 3 个按钮

接下来的就开始显得诡异了。第 4 个按钮看起来是一个红绿灯。单击某个灯，该灯就会变亮。木木以为这是 WPF 当中提供的一个控件，但是在 XAML 页面中发现它实际上是一个 RadioButton。

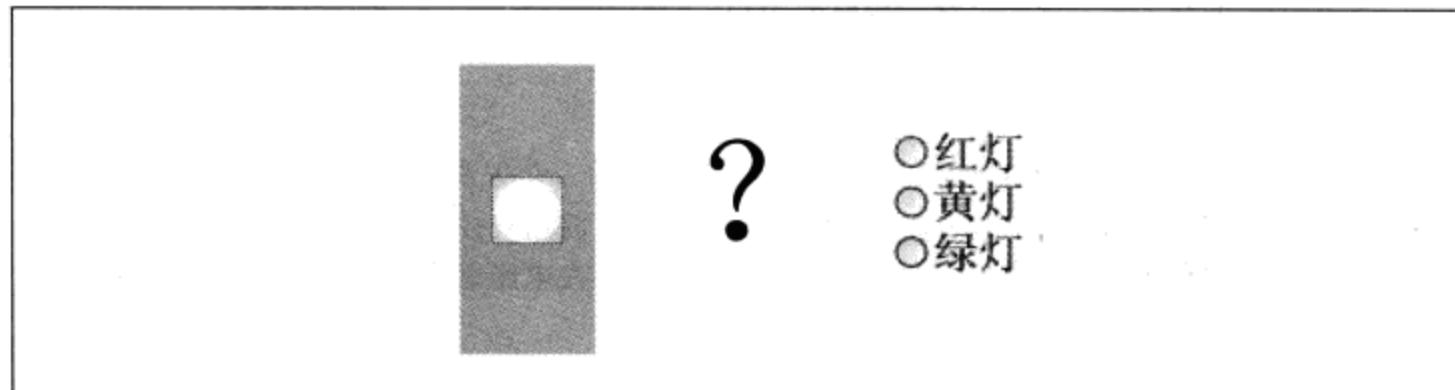


图 1-6 第 4 个按钮

第 5 个按钮如图 1-7 所示，它看起来是一个金属色与蓝色搭配的闸刀。单击后合上，再单击则推上。查看其 XAML 代码，明确无误地写的是 CheckBox。

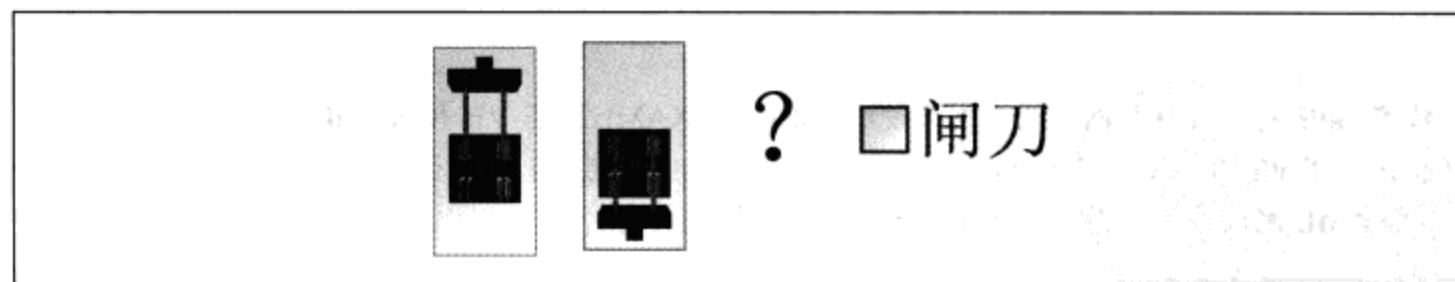


图 1-7 第 5 个按钮

木木点开了如图 1-8 所示的第 6 个按钮，一个潜艇在绿色网的中央不断地发出扫描线，周围是不停移动的飞机和船只。在其代码中他看到了一个令人难以相信的结果——ListBox。

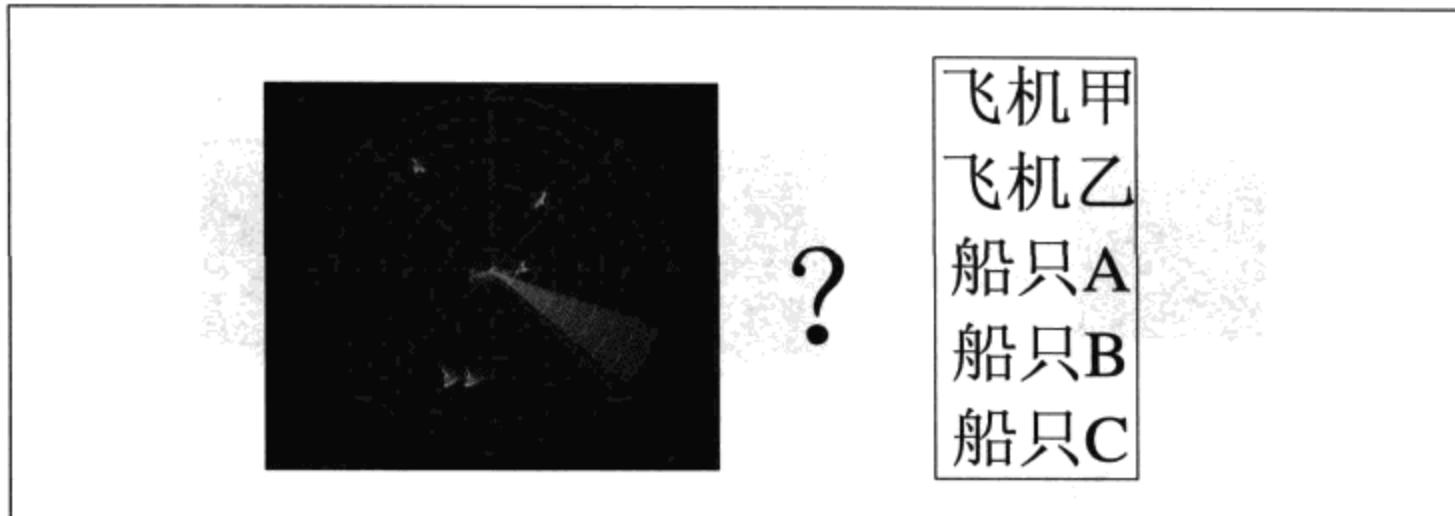


图 1-8 第 6 个按钮

虽然没有七十二变，但是这 6 种变化已经彻底将木木征服了。

1.2.2 WPF 的与众不同之处

体验了前面的“看我七十二变”，我们可以对 WPF 与众不同之处做一个小小的总结。

(1) 打造一个“富表现力”的平台

WPF 在整合所有过去表现技术的基础上建立了一个超级集合，从矢量图形及渐变效果等基本功能，到诸如 3D、动画和多媒体等高级功能一应俱全。而且 WPF 将用户控件、文档和媒体无缝地集成到一起，可以形成“你中有我，我中有你”的局面。过去只有在电影里看到的酷炫场景，现在都可以通过 WPF 实现。

(2) 引入 XAML 语言并将开发和设计分开

WPF 以前的桌面平台，无论是 Win32、MFC，还是 Windows Form，用户界面的设计往往占据了开发人员的大量时间，如在对话框中摆放合适的按钮并为其添加一个合适的图片等。即使有专业设计人员介入，他们也只能做出应用程序的设计图或者是图标等，开发人员必须将其转换为真正可用的程序。

一般的 WPF 程序往往是两种语言配合使用，如 C# 和 XAML 或者 VB 和 XAML。微软的最初想法是开发人员在 VS 2008 或者 VS 2010 中实现应用程序的业务逻辑，而设计人员在 Expression Blend 中使用 XAML 来设计用户界面，从而将开发和设计彻底分开。

虽然到目前为止开发和设计并没有如微软所希望的那样彻底分开，主要原因是通过 ExpressionBlend 生成的 XAML 文件往往比较低效，一个项目团队中会有一个开发人员负责简化设计人员的 XAML 文件使之更为高效。但是 WPF 在分离开发和设计上迈了很大一步，相信随着 Expression Blend 不断成熟，设计和开发会最终分开。

(3) Web 和桌面的统一模糊了二者程序的界限

在 WPF 程序中有 3 种不同的形式，即标准的桌面程序、导航模式的程序和完全寄宿于 IE 的程序（XAML Browser Application, XBAP），如图 1-9 所示。事实上有些 XAML 文件也可以单独地在 IE 中运行，我们将其称为“松散 XAML 文件”。

WPF 在很大程度上统一了 Web 程序和桌面程序，模糊了二者之间的区别，因此一个 WPF 程序可以较为轻松地移植到 Silverlight 程序上面。

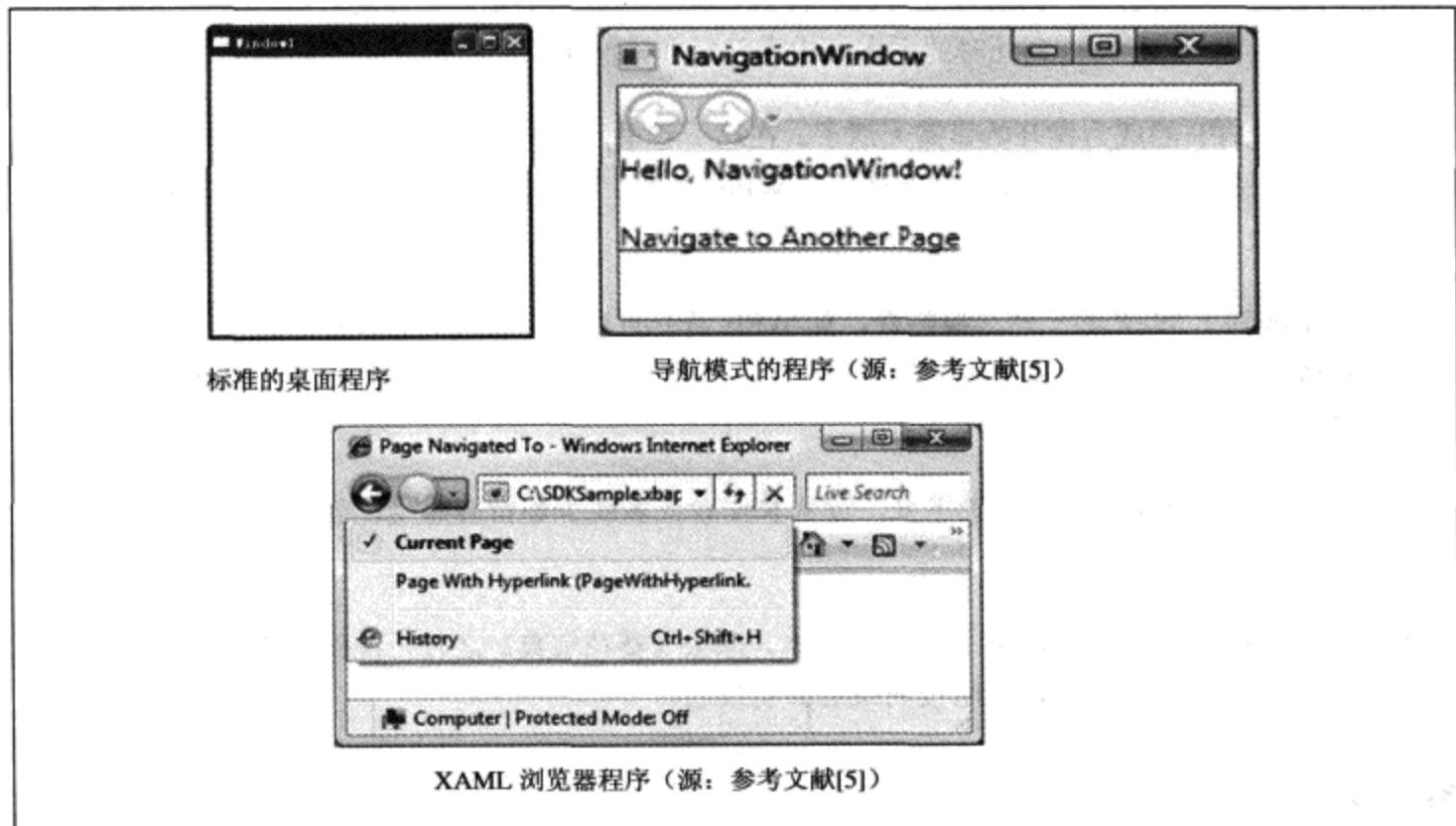


图 1-9 3 种不同形式的应用程序

(4) 硬件加速

WPF 的渲染是依赖于 DirectX，因此 WPF 应用程序可以获得更平滑的图像和更好的性能。

(5) 分辨率无关

WPF 使用的默认单位是 1/96 英寸，从而保证无论在显示器上，还是打印输出大小保持一致（还需要特定的条件，参见第 15 章）。此外 WPF 基本上都是使用矢量图形，因此放大后不会出现锯齿现象。

1.3 接下来做什么

接下来，我们首先会熟悉使用什么工具去编写 WPF 的应用程序（第 2 章 WPF 相关工具——十八般兵器）。然后对 WPF 的整体架构有所了解（第 3 章 WPF 体系结构——藏宝图）。自此，我们第一卷程序江湖就结束了。

进入第二卷。该卷是本书的心法，所谓心法就是基本功。这一卷的难度是最大的，内容涉及了 XAML 语言以及 WPF 的主要特性，依赖属性，路由事件和 WPF 的命令模型。阅读这一卷适当的走马观花是可以的。第一遍不懂，实践多了自然也就懂了。

第三卷就开始构建一个 WPF 的小小应用。相对第二卷来说，这一卷就简单许多。相信绝大多数人都和木木一样阅读这一卷是兵来将挡，水来土掩，遇魔降魔，遇佛杀佛。

第四卷我们又开始回归心法，这一卷里多是 WPF 与以往 UI 平台的不同的特性，而且这些特性和 XAML 语言结合非常紧密。内容涉及资源、样式、模板和数据绑定。不过这一卷相比第二卷来说，内容上相对简单，而且阅读到此时的木木，内力已经非同小可了。相信一定能够顺利过关。

到了第五卷，WPF 的威力就开始显露无遗了。在这一卷里，示例开始变得炫目了。这一节里涉及到二维图形、三维图形、动画、文本和文档，是 WPF 的“富”内容的具体展现，所谓紫杉红烛，打造视觉盛宴。

第六卷里牵涉到了 WPF 的一些高级特性，如 WPF 当中的互操作和自定义控件。这两卷涵盖的知识杂，内容较深，和心法一样算是本书的一个难点。到了这一步，真的是达到了“重剑无锋，大巧不工”的境界。

第七卷里只是一个展望，一本书读下来，读者和作者或多或少都留有感受，就全在“木木能行，我也行”七个大字当中了。

当然《葵花宝典》只是众多 WPF 书当中的一本，除去《葵花宝典》之外，在作者的博客上会有其他 WPF 的若干书评。读者也可以弃暗投明或者弃明投暗。

参考文献

- [1] 蔡学庸，“WPF 精粹（一）”，《程序员》杂志，2007 年 3 月。
- [2] Adam Nathan 著，瞿杰 单佐一 夏寒译，《WPF 揭秘》，2~9 页。
- [3] 黄晓春，“微软和开放——一段不得不说的往事（二）”《程序员》杂志，2007 年 12 月。
- [4] Chris Anderson 著 朱永光译，《WPF 核心技术》，2~6 页。
- [5] MSDN Library for Visual Studio 2008 SP1 Navigation Overview。

WPF 相关工具——十八般兵器

韦小宝心想：“这老和尚笨得要命。”笑道：“那又何必都学全了？只消知道小姑娘会什么招式。有道是兵来将挡，水来土掩。小姑娘这一招打来，老和尚这一招破去，管教杀得她们落荒而逃，片甲不回。”

——《鹿鼎记》，“第二十二回 老衲山中移漏处 佳人世外改妆时”^[1]

这一部分讲的是韦小宝向少林寺般若堂首座澄观讨教速成的少林功夫，用来对付他如花似玉的老婆阿珂，防止她“谋杀亲夫”。但是澄观老和尚非常拘泥不化，做事定要顺着次序。学习一指禅，必须先得学什么少林长拳、罗汉拳、伏虎拳、韦陀拳、散花手、波罗蜜手、金刚神掌和拈花擒拿手等一大套。等到学会，少则也要三四年，韦小宝当然没有这份耐心。韦小宝有他的一套速成理论，只需要兵来将挡、水来土掩就行。小姑娘一招打来，我韦小宝一招破解即可，哪还去在那里潜心数十年练那些啰里八嗦的少林功夫。

韦小宝的一套理论看似荒谬，其实也有他的道理。本章称为“十八般兵器”，主要介绍 WPF 的相关工具，其中 VS 2010 是最为复杂的。对于一个初学者来说，要完全掌握它就好像韦小宝学一指禅一样。如果按照澄观老和尚的学法，虽不至于数十年，十天半个月，甚至更长时间还是有可能的。因此掌握这些工具还是按照小宝的速成理论，先选一些最常用、最关键的掌握即可。随着不断地使用，逐步加深理解这些工具。

本章内容如下。

- (1) Microsoft Visual Studio 2010。
- (2) 命令行和记事本——小米加步枪。
- (3) Microsoft Expression Blend。
- (4) Xaml Pad。
- (5) Reflector。
- (6) 接下来做什么。

2.1 Microsoft Visual Studio 2010

2.1.1 13 年间

如果要追溯 Microsoft Visual Studio 系列，我们就要回到 13 年前，即 20 世纪的 1997 年。这一年的 3 月 19 日，微软公司发布了一套让人耳目一新的软件开发工具 Visual Studio 97^[2]。尽管以现在的角度来看，它只是多个迥然不同的工具的机械拼合，然而在当时已经是一个非常了不起的成就。

现在的开发人员绝大多数都没有见过这个套装的第一个版本，但是对 Visual Studio 6.0 并不陌生。Visual Studio 6.0 在 1998 年秋季推出，之后 Visual Studio 开始广为人知。

到 21 世纪，微软吹响了.NET 的号角。Visual Studio 7.0 最终还是贴上了.NET 的标签，Visual Studio (.NET) 2002 揭开了 Visual Studio 新的一页。从此 Visual Studio 工具就深深地打上了.NET 的烙印。这个版本的变化非常巨大，它是一个基于.NET 框架全新打造的开发环境。然而这个工具好景不长，刚刚推出不到一年的时间，就被 Visual Studio (.NET) 2003 所替代。

Visual Studio 2003 所使用的.NET 框架是 1.0 版本，到了 Visual Studio 2005 时，该框架从架构上做了一些根本的调整。即 2.0 版本，被认为是一个非常成功的版本。尽管 Visual Studio 2005 还在被广泛地使用，但是.NET 框架 3.0 很快推出了。3.0 并不像 2.0 那样是一个大的改进，而是在原来基础上的扩充。注意！WPF 就是在这时出现的，.Net 框架 3.5 和 Visual Studio 2008 也开始登场。2007 年 11 月，微软正式发布了 Visual Studio 2008。紧接着在 2008 年 8 月发布了 Visual Studio 2008 + Pack1（补丁），与之对应的.NET 框架是 3.5Pack1，即补丁。如果使用 Visual Studio 2008，本书推荐使用带有 Pack1 的 VS 2008。因为它与没有打补丁的 VS 2008 差别相当大。

目前.Net 框架最新版本是 4.0，截至本书出版之前，微软已于 2010 年 4 月 12 日正式推出了 Visual Studio 2010。值得一提的是，Visual Studio 2010 本身部分界面就是由 WPF 开发实现的。以前常常有人对 WPF 提出质疑，认为 WPF 无法胜任复杂的应用，VS 2010 将 WPF 作为其部分界面实现的技术本身就是对这种质疑最有力的回复。

由于在本书写作期间正式的 VS 2010 还没有发布，因此我们所采用的工具是 VS 2010RC 版。通过和微软的相关技术人员沟通，这一版和正式版仅有微小的区别。RC（Release Candidate，发布的候选版本）即这个版本发布之后，下一个版本就是正式版。当然在 RC 版本期间也可能经历 RC1、RC2 和 RC3，然后到正式版本。一般来说，微软内部发布一个产品会经历内部测试版、Alpha 版、Beta 版、RC 版和 RTM 版（正式版）。

VS 2010RC 版的官方下载地址为 <http://msdn.microsoft.com/en-us/vstudio/dd582936.aspx>。VS2010 支持的操作系统如下。

- (1) Windows XP Service Pack3。
- (2) Windows Server 2003 Service Pack2。
- (3) Windows Vista Service Pack2。
- (4) Windows Server 2008 Service Pack2。

- (5) Windows Server 2008 R2。
- (6) Windows 7。

目前一般用户使用的操作系统大多为 Windows XP、Vista 或者 Windows 7。如果是前两者，则需要打上相应的补丁；否则可以直接安装 VS 2010。

2.1.2 认识 Visual Studio 2010

Visual Studio 经历了 13 年的演绎，构成了一个庞大的家族。大部分和木木一样的初学者需要的只是“临敌应变的招术”，而不是全部的“少林武功”。

VS2010 速成只需要 3 招，木木，你准备好了没有？

1. 第 1 招：利用向导新建一个 WPF 项目

第 1 次启动 VS2010。启动时选择 Visual C# 语言开发环境，如图 2-1 所示。

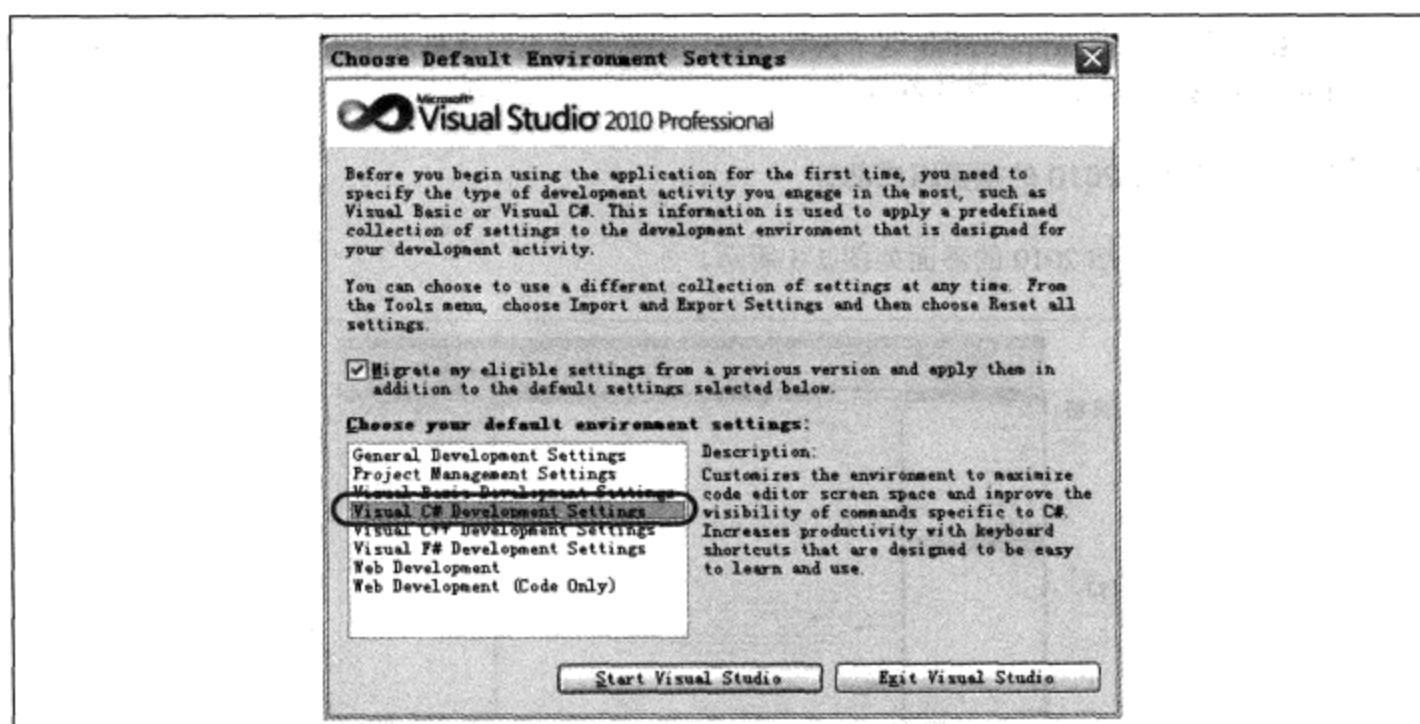


图 2-1 选择 Visual C# 语言开发环境

如果第 1 次选择错误，还可以在 VS 2010 的 Tools 菜单的 Import and Export Setting 选项中选择默认的语言环境。

选择 File>New\Project 选项，弹出 New Project 对话框，如图 2-2 所示。在 Project Type 中选择 Visual C# 的 Windows，在 Templates 中选择一个标准的 WPF Application 模板。然后指定一个存放该工程的目录，命名为“mumu_wpfapplication”，单击“OK”按钮。

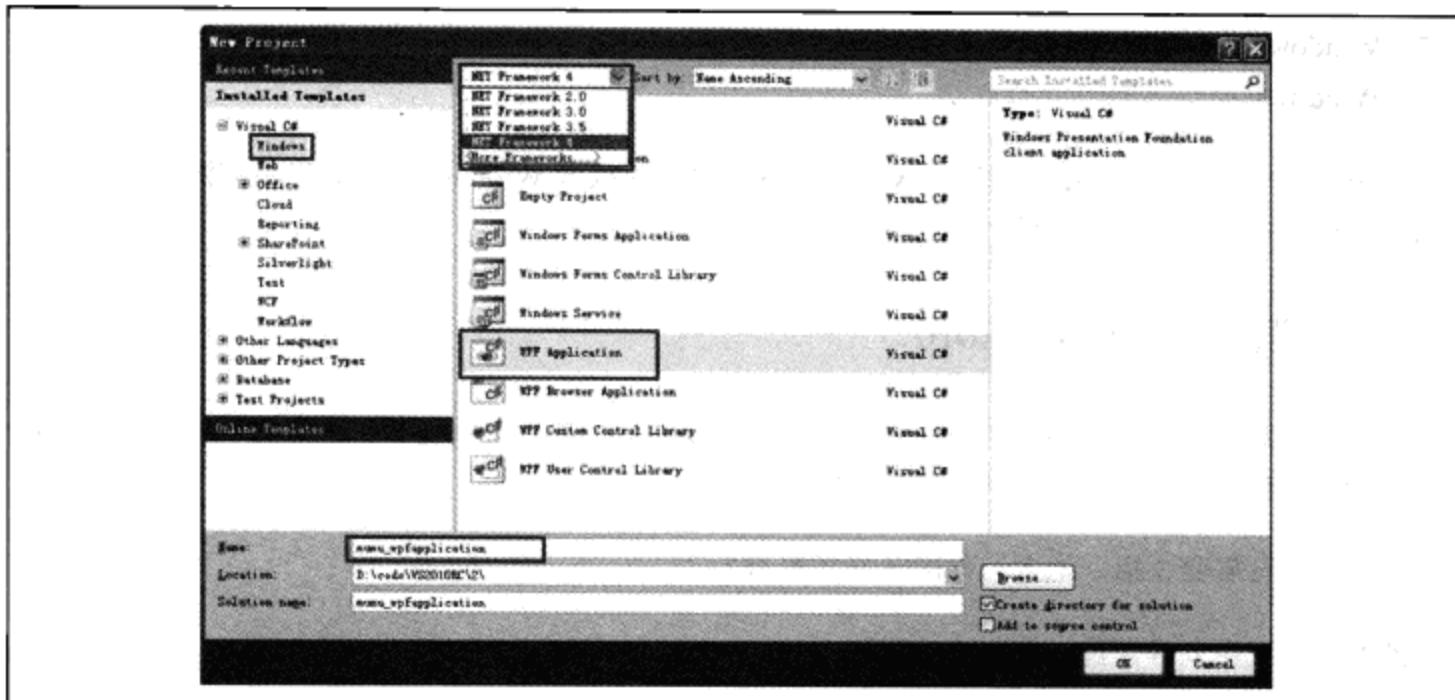


图 2-2 .Net Project 对话框

注意，如果找不到 WPF Application 这个模板选项，则需要检查对话框左上方的 ComboBox 中是否为 .Net Framework 3.0 以上，该复选框默认设置为 .Net Framework 4.0。

2. 第 2 招：认识 VS 2010 的重要组成部分

新建应用程序后通常 VS 2010 的界面如图 2-3 所示。



图 2-3 VS 2010 的界面

左侧上方是工具箱，其中放置了一些常用控件，如按钮及文本等，界面设计时可以将所需控件直接拖动到窗口中；左侧下方的 Document Outline 列表框用来显示用户界面的组成结构。

中间是代码编辑器，非常类似某些编写网页的环境。它提供了两种方式来设计界面：一是直接使用 XAML 语言（下半部分）；二是在可视化环境中拖动控件（上半部分）。可以上下调换二者位置，也可以按照左右排列或只显示某一个页面，如图 2-4 所示。

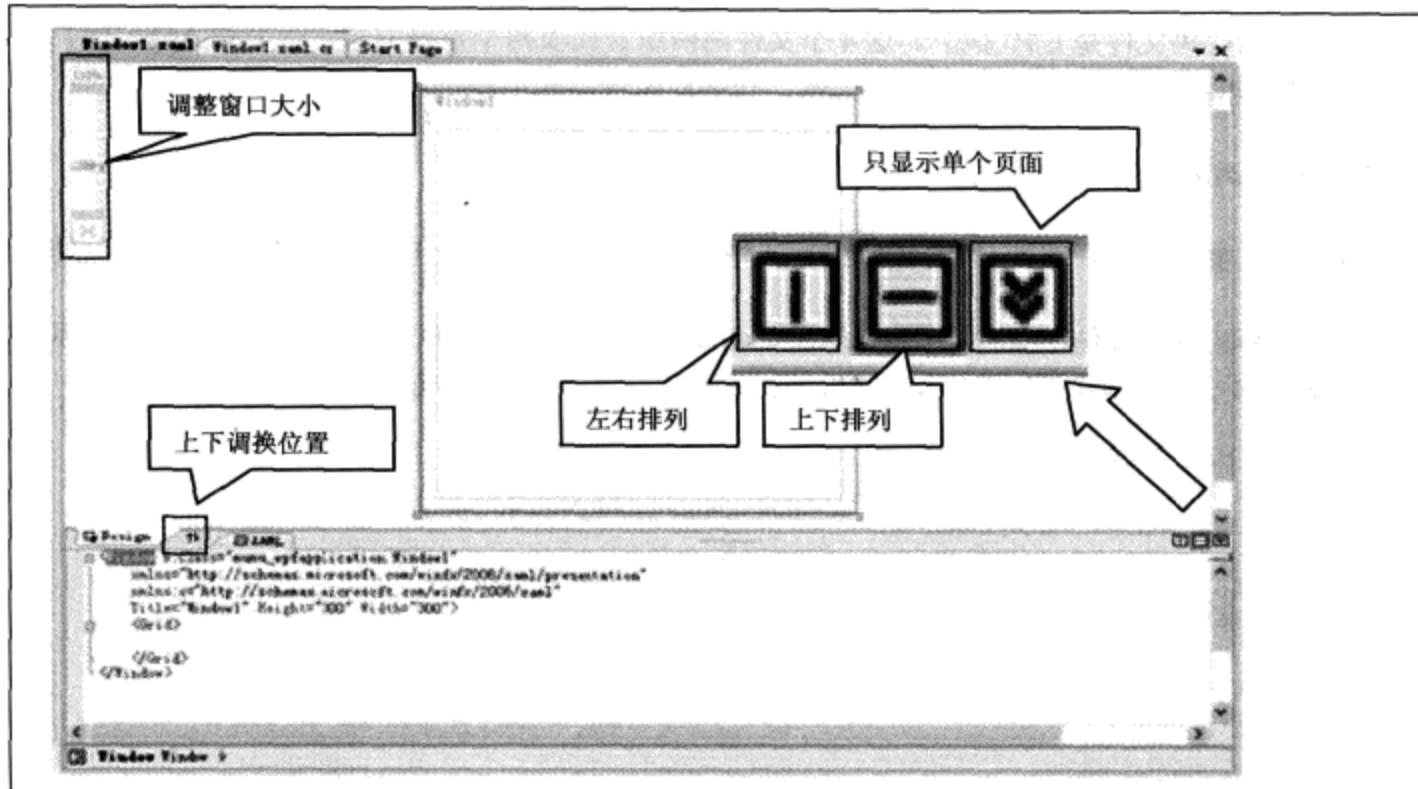


图 2-4 代码编辑器

最右侧是解决方案资源管理器（Solution Explorer），如图 2-5 所示。



图 2-5 解决方案资源管理器

其中一个解决方案是一个树状结构，根节点是 Solution，可以包含多个项目（Project）。在 Project 下面有一个 Reference 节点，它是该项目需要引用的外部程序集。一个 WPF 程序最少需要引用 5 个程序集，即 System、WindowsBase、PresentationCore、PresentationFramework 和 System.Xaml。再下面是一个 WPF 应用程序的 4 个典型文件，即 App.xaml、App.xaml.cs、MainWindow.xaml 和 MainWindow.xaml.cs¹。在 WPF 中往往是一个 XAML 文件和一个代码文件配合使用（VB 或者 C# 代

¹ 在 VS2008 中，至少需要引用 4 个程序集，不包括 System.Xaml，其默认窗口文件为 Window1.xaml 和 Window1.xaml.cs。

码，本书所指代码文件是 C# 代码）。这 4 个文件两两组合代表两个重要的类，一个是 App，代表一个 WPF 的应用程序；另一个是 MainWindow，代表应用程序的主窗口。

3. 第 3 招：调试 WPF 应用程序

调试程序首先要让这个新建的 WPF 程序执行某种操作，如单击按钮弹出一个“Hello WPF”消息框。

在 MainWindow.xaml 文件中添加一个按钮的标签<Button>，设置其宽度和高度，并添加一个 Click 事件处理函数。VS 会自动地在代码文件 MainWindow.xaml.cs 中添加上一个空的实现，如代码 2-1 所示。

```
MainWindow.xaml
<Window x:Class="mumu_wpfapplication.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
    <Grid>
        <Button Content="Hello WPF" Margin="5" Click="Button_Click"/>
    </Grid>
</Window>

MainWindow.xaml.cs
.....
namespace mumu_wpfapplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private void Button_Click(object sender, RoutedEventArgs e)
        {
        }
    }
}
```

代码 2-1 添加按钮的 Click 事件处理函数

在 Button_Click 函数中写下代码 2-2 所示的代码，表示如果单击按钮，则弹出一个“Hello WPF”消息框。

```
string s = "Hello WPF";
MessageBox.Show(s);
```

代码 2-2 Click 事件处理函数的实现代码

调试程序首先要在执行的代码中设置断点。当程序运行到该行代码时就会中断。这样开发人员就可以查看代码中的变量，或者再继续一步一步地执行。设置断点的简单方法是单击代码编辑器的左侧，会出现一个红色圆圈表示已经设置断点，如图 2-6 所示。

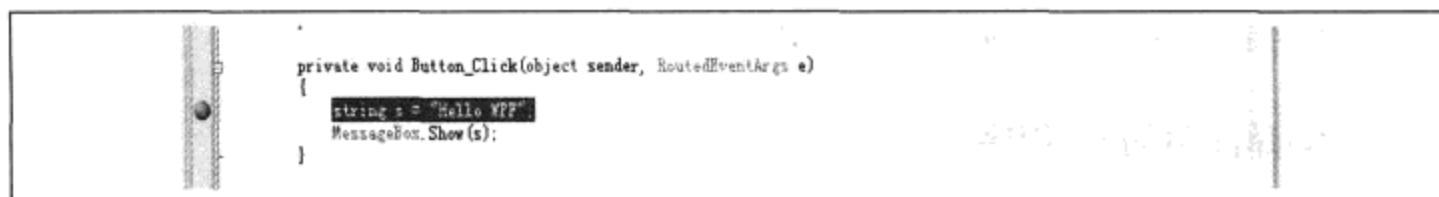


图 2-6 设置断点

再次单击这个红色圆圈会删除断点，设置和删除断点也可以通过按快捷键 F9 实现。在 VS 环境中可以单击工具栏中如图 2-7 所示的一个绿色箭头按钮，或者按快捷键 F5 启动调试程序。

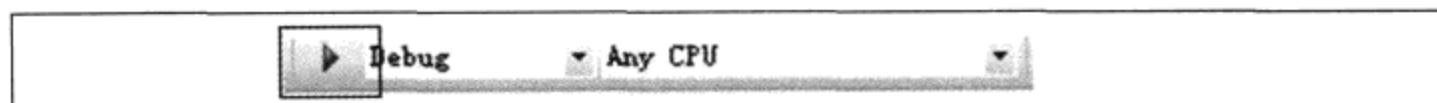


图 2-7 启动调试的按钮

单击该按钮应用程序停止在设置的断点处，此时查看变量值。简单的方法是在代码编辑器中把鼠标指针放在该变量名上，此时会显示一个小方框，其中给出该变量的值，如图 2-8 所示。

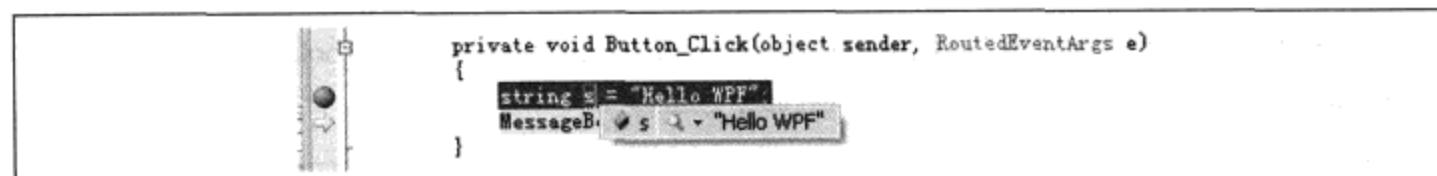


图 2-8 查看变量值

在调试时 VS 环境下方会出现 Autos、Locals 和 Watch 窗口，其中 Autos 用于监视程序运行时最后访问的变量；Locals 监视当前执行方法中的变量；Watch 监视用户希望看到的变量，如可以将变量 s 拖放到 Watch 窗口中来查看其值，如图 2-9 所示。

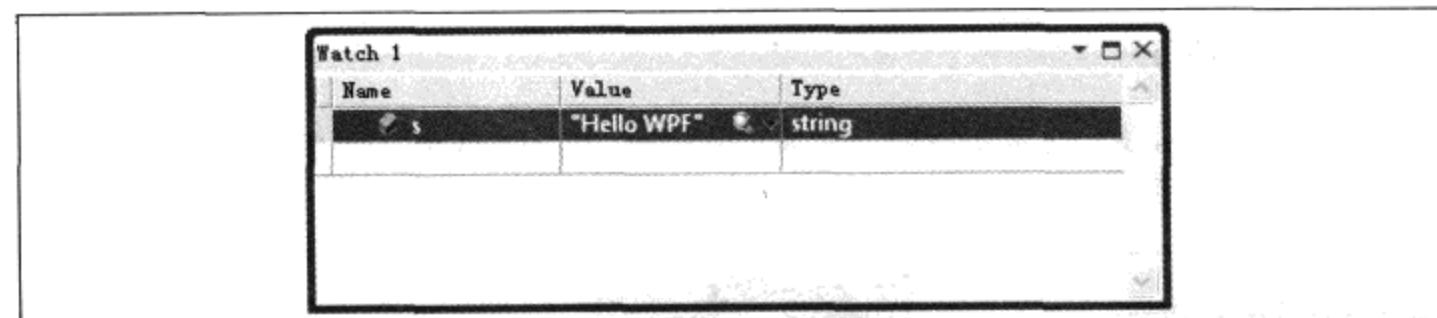


图 2-9 Watch 窗口

可以让程序分步执行（快捷键 F10），也可以让它进入到某一个函数（快捷键 F11），或者跳出某一个函数（组合键 Shift+F11），甚至是结束调试程序（组合键 Shift+F5）。

2.2 命令行和记事本——小米加步枪

本书用基本的方式来构建一个简单的 C# 程序，进而构建一个 WPF 应用程序，这样的做法好处如下。

- (1) 未必有 Visual Studio 2010 这样大型的开发工具，有 .Net Framework 4.0 SDK 即可实现。

(2) 可以更好地理解 Visual Studio 2010 的原理。

2.2.1 编译简单的 C# 程序

我们按照下面的步骤来编译一个简单的 C# 程序。

(1) 用记事本写一个简单的 `mumu_hellocsharp.cs` 文件，如代码 2-3 所示，并且假定保存在 `D:\code\VS2010RC\2\simple C#文件夹下`。

```
using System;
class mumu_hellocsharp
{
    static void Main()
    {
        Console.WriteLine("Hello,mumu!");
    }
}
```

代码 2-3 `mumu_hellocsharp.cs` 文件

(2) 选择“开始”|“所有程序”|`Microsoft Visual Studio 2010\Visual Studio Tools\Visual Studio 2010 Command Prompt` 选项，弹出如图 2-10 所示的命令行窗口²。

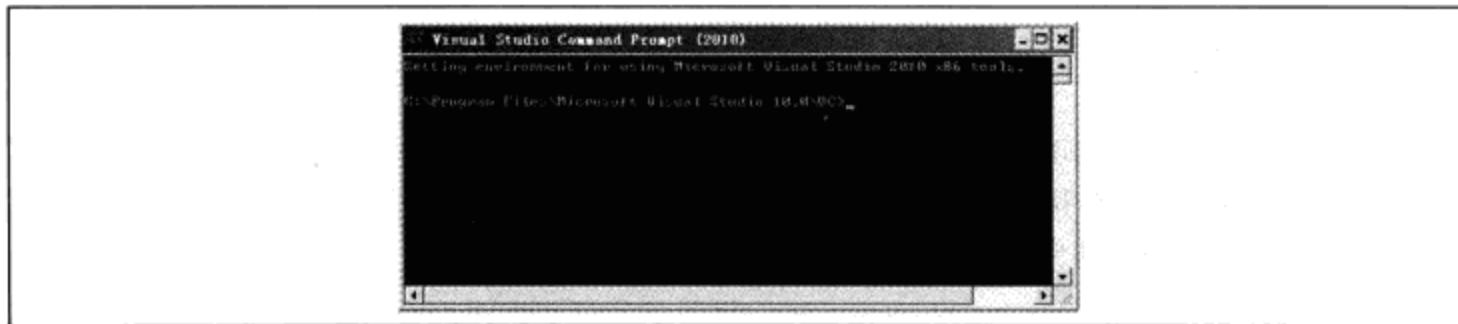


图 2-10 命令行窗口

输入 `csc -?`（注意 `csc` 和 `-?` 之间为空格），如果一切正常，可以看到 C# 命令行编译器的命令行参数列表，如图 2-11 所示。



图 2-11 命令行参数列表

² 如果使用 Microsoft Visual Studio 2008，则选择“开始”|“所有程序”|`Microsoft Visual Studio 2008\Visual Studio Tools\Visual Studio 2008 Command Prompt` 选项。

(3) 输入“D:”切换到 D 盘，然后输入 cd \code\VS2010RC2\simple C#，进入到 D:\code\VS2010RC2\simple C#目录。输入 csc mumu_hellocsharp.cs，编译 mumu_hellocsharp.cs 文件。在目录下生成一个 mumu_hellocsharp.exe 文件，双击该文件即可启动运行。也可以直接在命令行中输入 mumu_hellocsharp 运行，运行结果如图 2-12 所示。

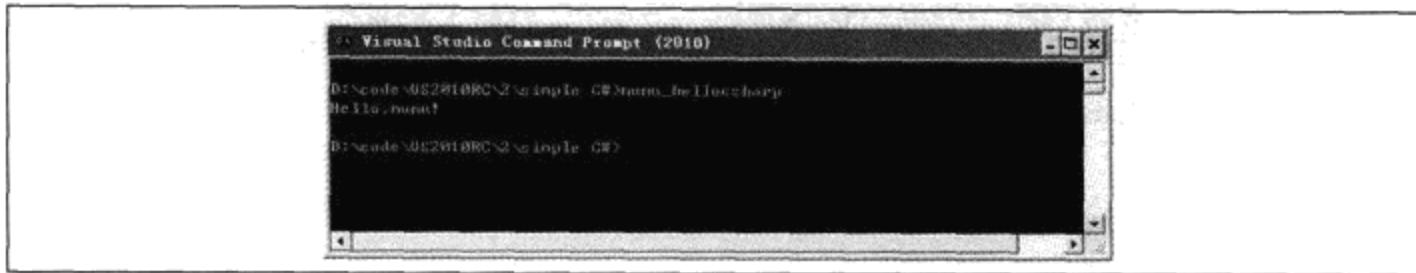


图 2-12 运行结果

2.2.2 引用外部程序集

如果希望通过一个消息框弹出“Hello, mumu! ”，则需要引用一些外部的程序集。

将 mumu_hellocsharp.cs 文件稍加修改，另存为 mumu_hellocsharp2.cs 文件。并且保存在 D:\code\VS2010RC2\simple C#目录下，如代码 2-4 所示。

```
using System.Windows;
class mumu_hellocsharp
{
    static void Main()
    {
        MessageBox.Show("Hello,mumu!");
    }
}
```

代码 2-4 mumu_hellocsharp2.cs 文件

MessageBox 属于 PresentationFramework 程序集，因此在编译这个文件时需要引用该程序集。在命令行中引用程序集需要用到/reference 标识。整个编译命令³（为了显示得清楚一些，我们在编译命令中使用了回车键换行。实际上中间不能使用回车键换行，否则命令行会以为是一条完整的编译命令），如代码 2-5 所示。

```
csc mumu_hellocsharp2.cs
/reference:"C:\Program Files\Reference
Assemblies\Microsoft\
Framework\.NETFramework\v4.0\Profile\Client\presentationframework.dll"
```

代码 2-5 在命令行窗口中输入的命令

编译后在同一个目录下生成 mumu_hellocsharp2.exe 文件，直接在命令行中运行，结果如图 2-13 所示。

³ 如果在 VS 2008 下，则 presentationframework.dll 在路径 C:\Program Files\Reference Assemblies\Microsoft\ Framework\v3.0\下面。

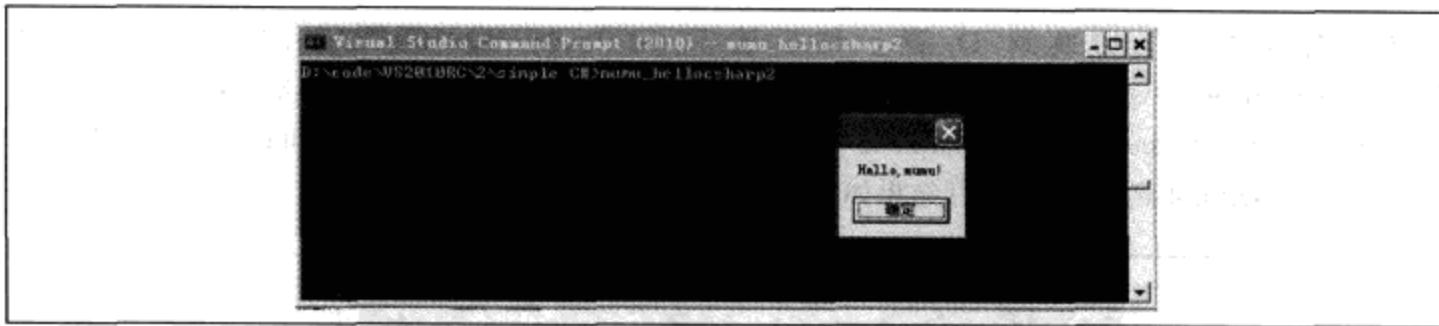


图 2-13 mumu_hellocsharp2.exe 的运行结果

2.2.3 编译 WPF 应用程序

1. 使用 csc 命令

新建一个 `mumu_hellowpf.cs` 文件，该程序简单地弹出一个 WPF 窗口。任何一个 WPF 程序的 Main 前面都必须有一个[STA Thread]属性；否则运行时会抛出异常。这个属性用来声明该应用程序的线程模型为套间线程模型（Single Thread Apartment），如代码 2-6 所示。

```
using System;
using System.Windows;

class mumu_hellowpf
{
    [STAThread]
    public static void Main()
    {
        Window win = new Window();
        win.Title = "Hello mumu!";
        win.Width = 300;
        win.Height = 200;
        win.Show();

        Application app = new Application();
        app.Run();
    }
}
```

代码 2-6 一个简单的 WPF 应用程序实现

`System.dll` 是默认引用的程序集，因此这个程序的编译命令如代码 2-7 所示。

```
csc mumu_hellowpf.cs
/reference:"C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\Profile\Client\presentationframework.dll"
/reference:"C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\Profile\Client\windowsbase.dll"
/reference:"C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\Profile\Client\presentationcore.dll"
/reference:"C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\Profile\Client\system.xaml.dll"
```

代码 2-7 编译命令

可以将这个编译命令保存为一个批处理文件，如命名为“`build.bat`”，这样在命令行中输入 `build.bat`

可编译该文件。然后输入 `mumu_hellowpf` 即可运行该程序，如图 2-14 所示。

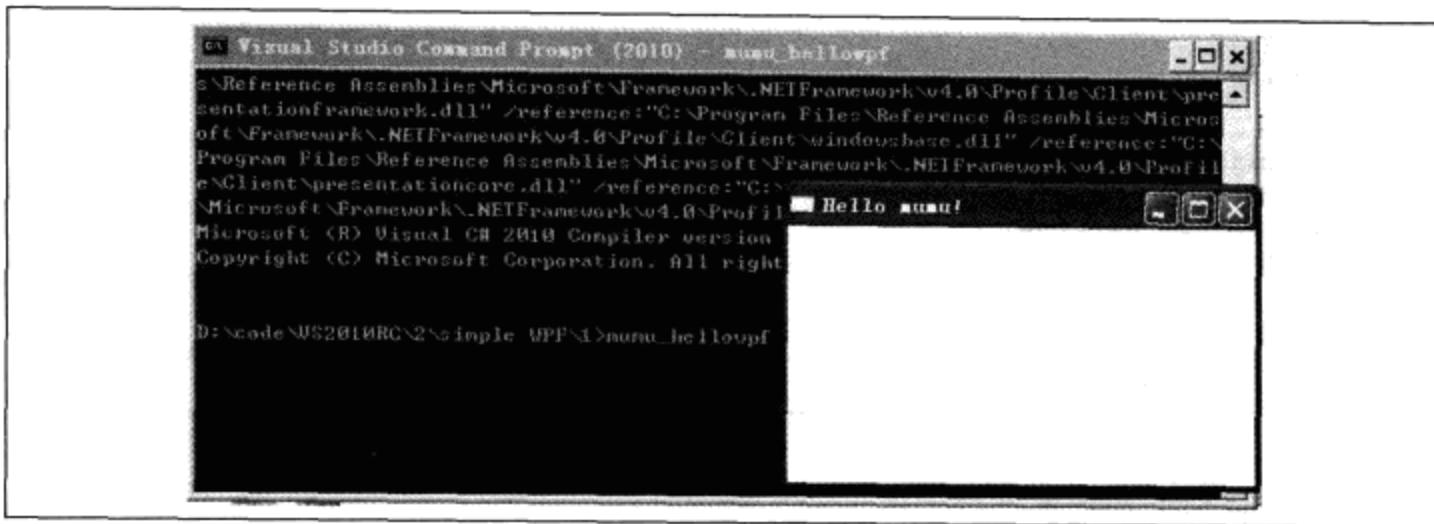


图 2-14 运行 `mumu_hellowpf` 程序

2. 使用 msbuild 命令

(1) 典型 WPF 应用程序的文件组成

VS2010 创建的一个典型 WPF 程序由 `App.xaml`、`App.xaml.cs`、`MainWindow.xaml` 和 `MainWindow.xaml.cs` 共 4 个文件组成的。其中 `App.xaml` 和 `App.xaml.cs` 两个文件描述一个 `App` 类型的对象，表示 WPF 的应用程序对象，如代码 2-8 所示。

```
App.xaml
<Application x:Class="mumu_hellowpf2.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>

App.xaml.cs
using System;
using System.Windows;
namespace mumu_hellowpf2
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}
```

代码 2-8 `App.xaml` 和 `App.xaml.cs` 文件

`MainWindow.xaml` 和 `MainWindow.xaml.cs` 两个文件描述 WPF 程序的主窗口对象，如代码 2-9 所示。

```
MainWindow.xaml
<Window x:Class="mumu_hellowpf2.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="300" Width="300">
<Grid>
</Grid>
</Window>

MainWindow.xaml.cs
using System;
using System.Windows;
namespace mumu_hellowpf2
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}

```

代码 2-9 MainWindow.xaml 和 MainWindow.xaml.cs 文件

(2) MSBuild 项目文件

因为 csc 命令无法编译 XAML 文件，所以使用 MSBuild 命令来编译 WPF 应用程序。该命令编译的对象是 MSBuild 项目文件，它使用 XML 格式组织需要编译的源码。针对前面的 4 个文件，建立一个 project.csproj 项目文件。实际上 VS2010 的项目文件本身就是一个 MSBuild 项目文件，通过 MSBuild 命令进行编译，如代码 2-10 所示。

```

①  <Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/
developer/msbuild/2003">
②    <PropertyGroup>
        <AssemblyName>mumu_hellowpf2</AssemblyName>
        <OutputType>winexe</OutputType>
    </PropertyGroup>
③    <ItemGroup>
        <Reference Include="System" />
        <Reference Include="WindowsBase" />
        <Reference Include="PresentationCore" />
        <Reference Include="PresentationFramework" />
        <Reference Include="System.Xaml" />
    </ItemGroup>
④    <ItemGroup>
        <ApplicationDefinition Include="App.xaml" />
        <Compile Include="App.xaml.cs" />
        <Page Include="MainWindow.xaml" />
        <Compile Include="MainWindow.xaml.cs" />
    </ItemGroup>
⑤    <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>

```

代码 2-10 MSBuild 项目文件

project.csproj 项目文件其实是一个 XML 文件，后缀名为 csproj 表示这是一个 C# 项目；如果是一个 VB 项目，则后缀名为 vbproj。MSBuild 项目文件的根标签元素是 Project（代码①），在 Project 元

素中声明了 XML 的命名空间和编译工具的版本等。在第 1 个 PropertyGroup 的标签中（代码②）主要指定程序集输出的名字和类型。这里输出的程序集名称为“mumu_hellowpf2”，是一个类型为 EXE 的可执行程序。如果 OutputType 标签中是 library，则会输出一个后缀名为“dll”的动态链接库文件。ItemGroup 的标签中的内容相当于 MSBuild 的输入，MSBuild 在编译时会处理这些内容。第 1 个 ItemGroup 标签中（代码③）是需要引用的程序集；第 2 个（代码④）则是需要编译的源代码文件。后缀名为 cs 的文件的标签都是 Compile，如 App.xaml.cs 和 MainWindow.xaml.cs。App.xaml 文件的标签必须是 ApplicationDefinition，而 MainWindow.xaml 这种窗口文件的标签则是 Page。最后的标签称为“目标”（Targets）文件（代码⑤），用于通知 MSBuild 如何 Build 这个项目^[5]。

（3）编译 MSBuild 项目文件

为编译 MSBuild 项目文件，在命令行中键入命令 msbuild project.csproj。在当前目录的 bin\Debug 目录下生成 mumu_hellowpf2.exe，双击该程序即可运行。

2.3 Microsoft Expression Blend

Microsoft Visual Studio 2010 更适合开发人员使用，而微软希望有一款产品使得图形设计师能够自由地设计理想的用户界面，然后自动生成相应的 XAML 文件，这就是 Microsoft Expression Blend（以下简称“Expression Blend”）出现的原因。

2.3.1 优势

Expression Blend 实际上属于 Microsoft Expression Studio 系列工具的一种，其中包括 Microsoft Expression Design、Microsoft Expression Encoder 和 Microsoft Expression Web。自本书出版前，目前的最新版本为 4.0，可以在 <http://www.microsoft.com/expression/try-it/#PageTop> 下载。

Expression Blend 是一款功能齐全的全新专业设计工具，它的一个主要优点是图形设计师可以和使用 Adobe Photoshop 或者 Macromedia Director 一样来设计用户界面，同时可以输出生成后的 XAML 文件。其目标是图形设计师在不编写 C# 代码的情况下构建丰富的 UI，这是一个美好的远景，但是目前还存在差距。ExpressionBlend 甚至能够将 Photoshop 或者 Illustrator 设计的文件直接转换为 XAML 文件，但是转换后的文件过于冗长，而且效率低下。因此真正的实践中往往会有一个人配合图形设计师，将其设计的 XAML 文件优化后使用。

Expression Blend 的另外一个优点是和 Microsoft Visual Studio 共用相同的工作空间，这样图形设计师完成用户界面设计之后，C# 开发人员就可以直接打开项目编码，中间没有任何转换过程。

2.3.2 组成

Expression Blend 中的工作区由多种可视界面元素组成，包括中央的设计面板、项目面板、交互面板、XAML 代码编辑区、工具箱和属性面板等，如图 2-15 所示。



图 2-15 Expression Blend 的工作区

- (1) 设计面板：在其中调整各个元素，以便可视化预览所设计界面。
- (2) 项目面板：所有的文件均组织在其中，类似 Microsoft Visual Studio 2010 的项目窗口。
- (3) 交互面板：查看所有对象的层次结构，选择对象以便修改。并且为控件对象创建和修改模板、动画时间线及触发器等。
- (4) XAML 代码编辑区：可以使用 XAML 代码编辑界面。
- (5) 工具箱：包括选择、视图、画笔、对象和资源工具等。
- (6) 属性面板：设置各个元素的属性，如设置一个 Button 按钮的前景色、背景色、字号、字体、透明度、Margin、Padding、长度和宽等。

虽然 Expression Blend 已经开始流行，但是本书主要还是面对程序员，因此使用的主要工具仍旧是 VS 2010。

2.4 XamlPad

XamlPad 是学习 XAML 语言的一个实用小工具，你安装了 Windows SDK 之后就会包含这个小工具。

XamlPad 是一个记事本风格的文本编辑器，如图 2-16 所示。与 VS 2010 中 WPF 设计器类似，在窗口下方的 XAML 代码窗格添加 XAML 代码，正上方是代码的可视化效果。

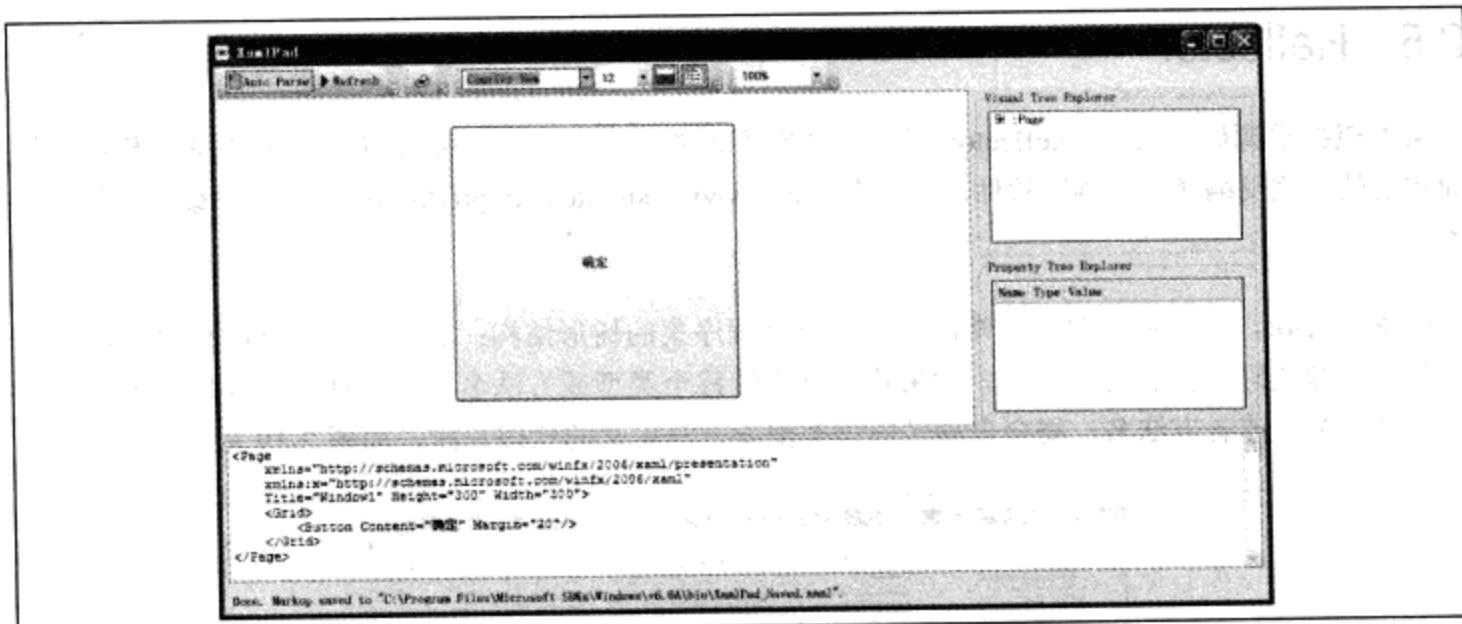


图 2-16 XamlPad 工具

当代码发生错误时，整个文本会变成红色。同时最下方的状态栏报告错误信息。如图 2-17 所示。

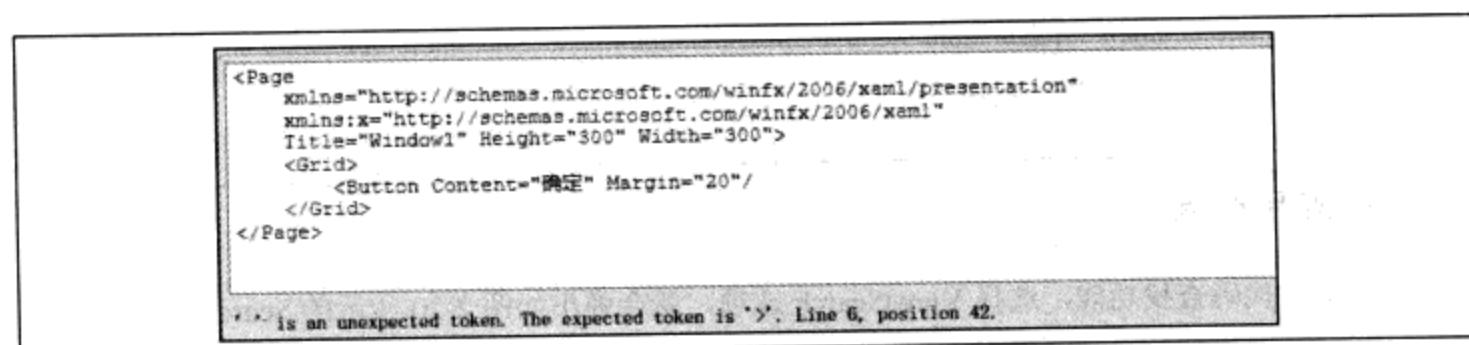


图 2-17 报告错误信息

Xaml Pad 最为难能可贵的是提供了一个 WPF 可视化树窗口，用于探索 WPF 的可视化树（虽然在 VS 2010 中可以通过 Document outline 窗口查看 WPF 的逻辑树，但是无法查看其可视化树），如图 2-18 所示。

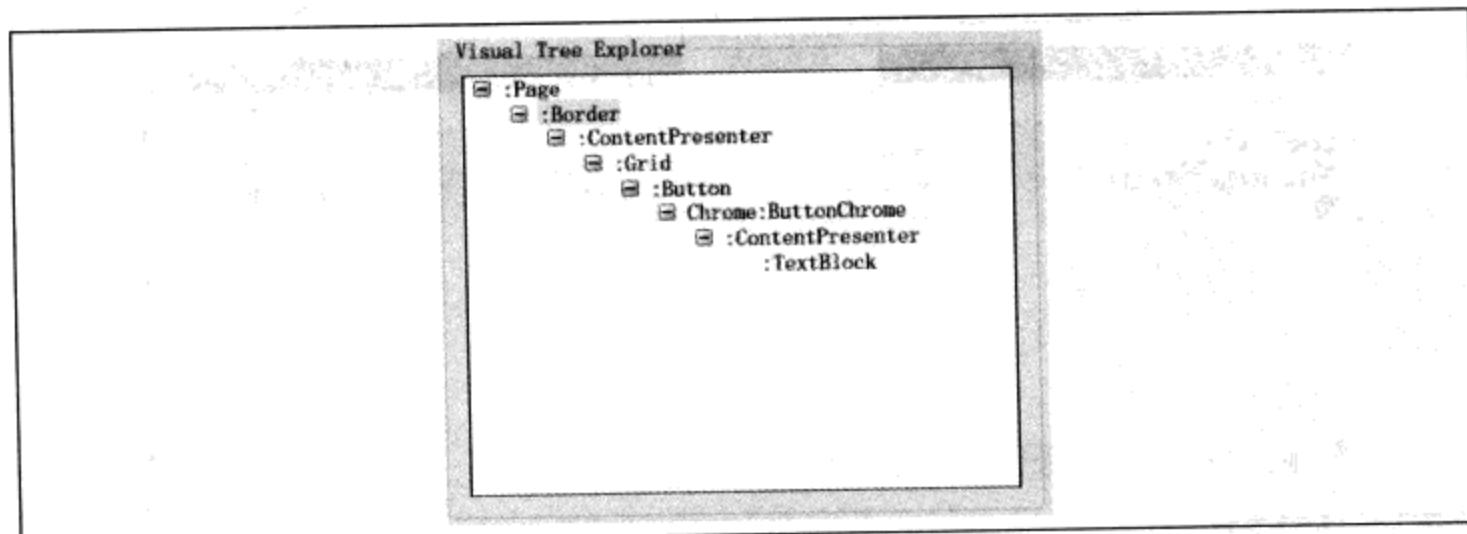


图 2-18 可视化树窗口

2.5 Reflector

如果希望探求源码⁴，那么 Reflector 是一个非常不错的工具。它是利用.NET 反射机制来分析程序集而实现的一款反编译及类浏览软件，可以在 <http://www.red-gate.com/products/reflector/> 下载到最新的版本。

运行 Reflector，应用程序窗口非常简洁。左侧是程序集的树形结构；右侧是所选择的程序节点的源代码，其源代码支持超链接行为。如果将鼠标指针移至类型或方法名上面，就会出现属性提示框。单击这个类型或者方法名，将会自动转向该类型或方法定义的代码片段，如图 2-19 所示。

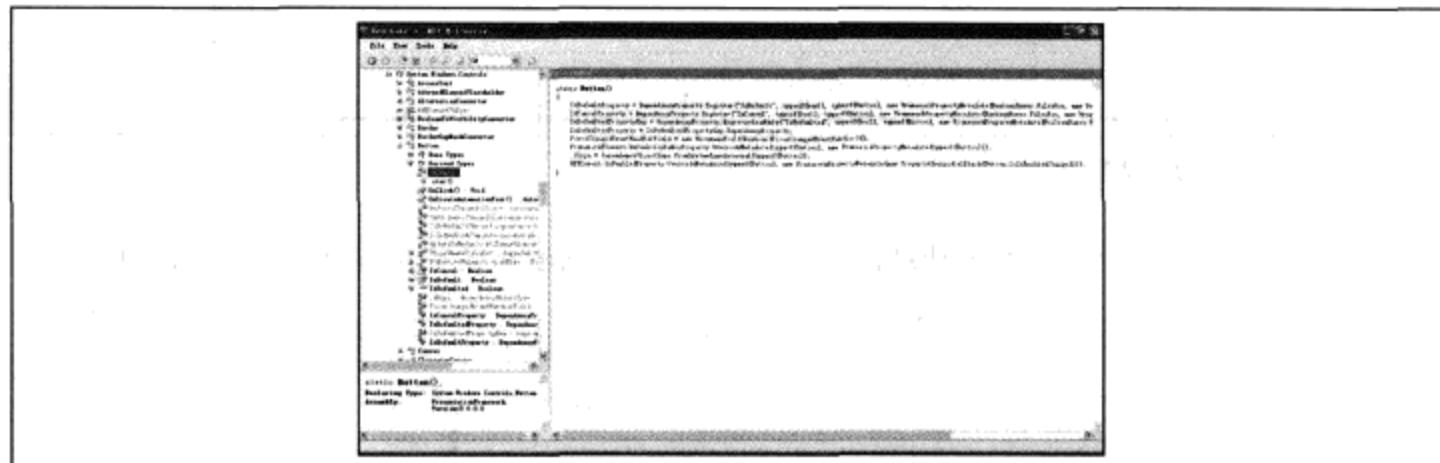


图 2-19 应用程序窗口

Reflector 提供代码查找功能，选择 View|Search 选项，就会弹出如图 2-20 所示的 Search 对话框。

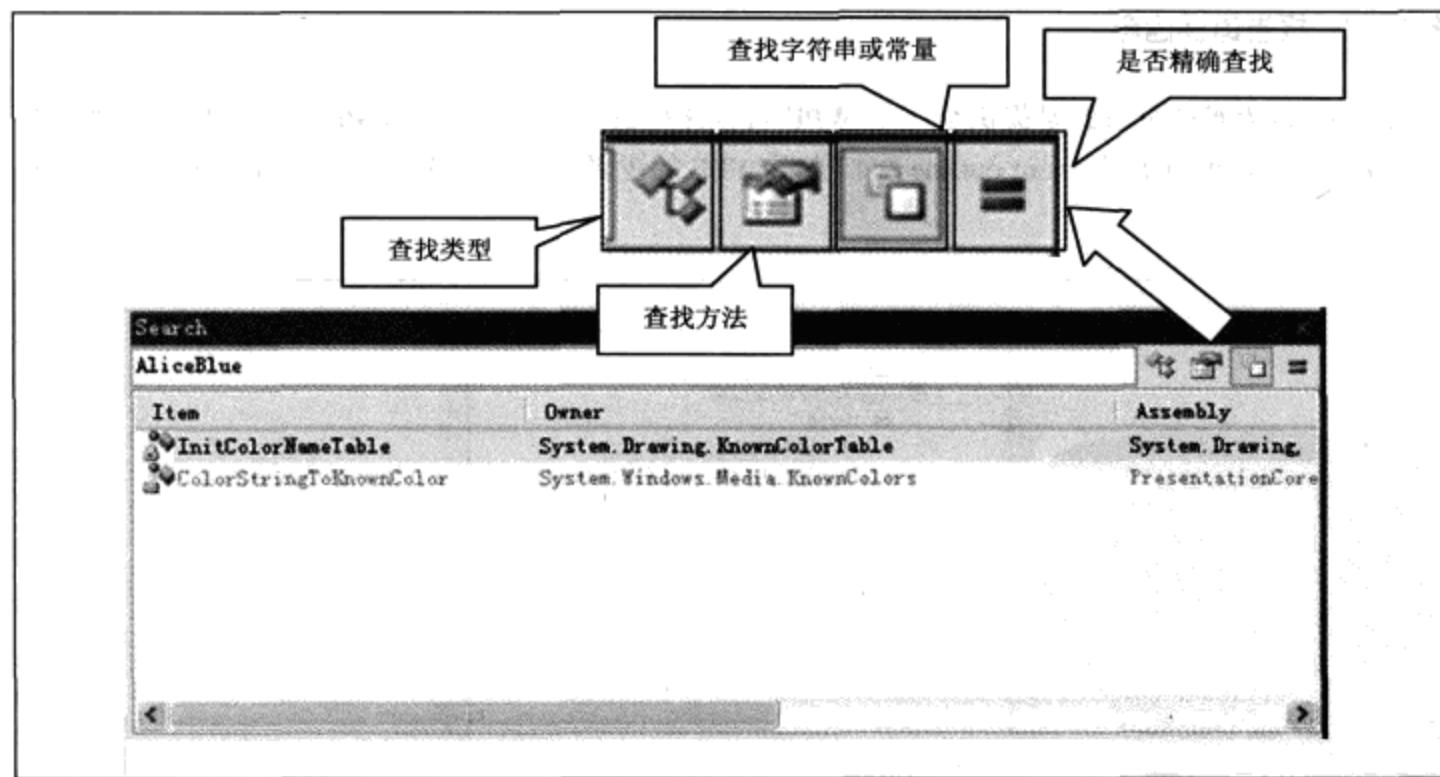


图 2-20 Search 对话框

⁴ 探索.NET 源码可以参见作者的一篇博文“我探索.NET 源码的三个阶段”。

在其中可以搜索类型、方法、字符串和常量，并提供精确和模糊两种方式。

2.6 接下来做什么

认识了 WPF 的相关工具，木木相当于选好了兵器。剩下的事情是需要拿到一张藏宝图，其中蕴含的正是 WPF 体系架构的秘密。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作，《金庸全集典藏版 鹿鼎记》，“第二十二回 老衲山中移漏处 佳人世外改妆时”。
- [2] 冉林仓，“Visual Studio 十年磨一剑”《开发人员》，2007 年 5 月。
- [3] Christian Nagel、Bill Evjen 和 Jay Glynn 等著，李铭译，《C#高级编程（第 6 版）》，360~396 页。
- [4] 成心文，“.NET 编程利器：Relector for .NET”《开发人员》，2007 年 4 月。
- [5] MSDN Library for Visual Studio 2008 SP1 Building a WPF Application (WPF)。
- [6] Adam Nathan 著，瞿杰、单佐一和夏寒译，《WPF 揭秘》，484~486 页。

WPF 体系结构——藏宝图

韦小宝跟着她走到桌边，只见桌上大白布上钉满了几千枚绣花针，几千块碎片已拼成一幅完整无缺的大地图。难得的是几千片碎皮拼在一起，既没多出一片，也没少了一片……

——《鹿鼎记》，“第三十四回 一纸兴亡看复鹿 千年灰劫付冥鸿”^[1]

上面这一段讲的是好双儿为小宝将几千张碎羊皮拼成了一幅藏宝图，其中隐藏了大清龙脉的秘密。WPF 中也有一幅藏宝图，其中包含了 WPF 的所有秘密，此即 WPF 的类图^[1]。木木要搞清楚 WPF 的类图，首先必须对 WPF 的体系结构有个大致的了解。

本章内容如下。

- (1) Windows 体系结构。
- (2) WPF 内部结构。
- (3) 接下来该做什么。

3.1 Windows 体系结构

我们可以把整个 Windows 的体系架构比做一个恢弘的大厦，各种不同的技术，包括 WPF 都在某一层的某一个房间之内，如图 3-1^[2]所示。

这座大厦的最上层是用户构建的应用程序，如 Office 程序、Maya 软件或者是木木写的“Hello WPF”。其下是系统应用程序，如桌面窗口管理器（Desktop Window Manager）或者 IE 浏览器。其下是应用程序框架，是开发人员比较熟知的，如 MFC/ATL、WinForms 和 WPF。其下是高一中一低层的抽象，如应用程序框架提供的控件、按钮、组合框、托管 DirectX、OpenGL 和非托管的 DirectX。再其下是用户模式的驱动，即用户模式的底部。下面是内核模式^[2]，这一部分是操作系统的内核，如内存及设备管理。其下是硬件的驱动，最终到达 Windows 体系架构的底部。

把这个大厦分为 3 层，即用户应用程序、系统图形框架，以及操作系统核心和硬件驱动，如图 3-2 所示。WPF 处在系统图形框架这个中间层。

¹ WPF 的类图见本书附录 A。

² 用户模式和内核模式是操作系统的两种模式，请参见操作系统的相关书籍。

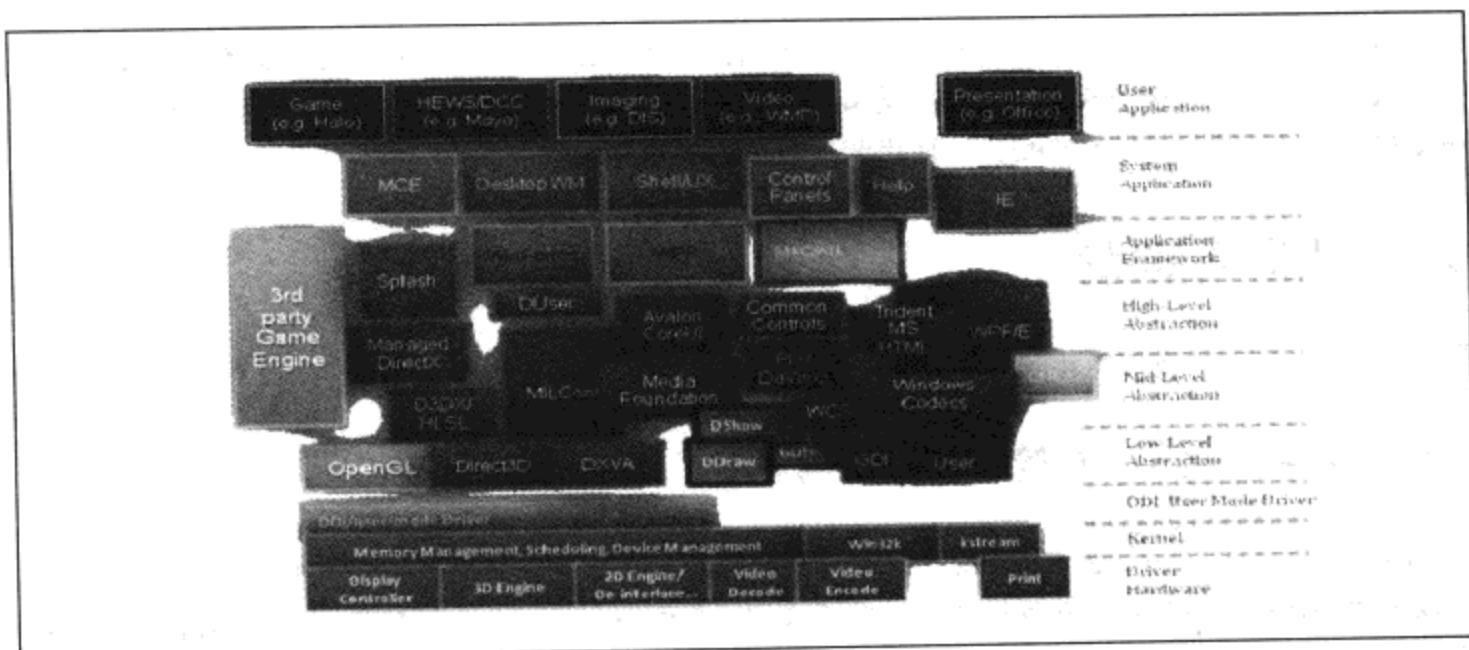


图 3-1 Windows 体系结构图

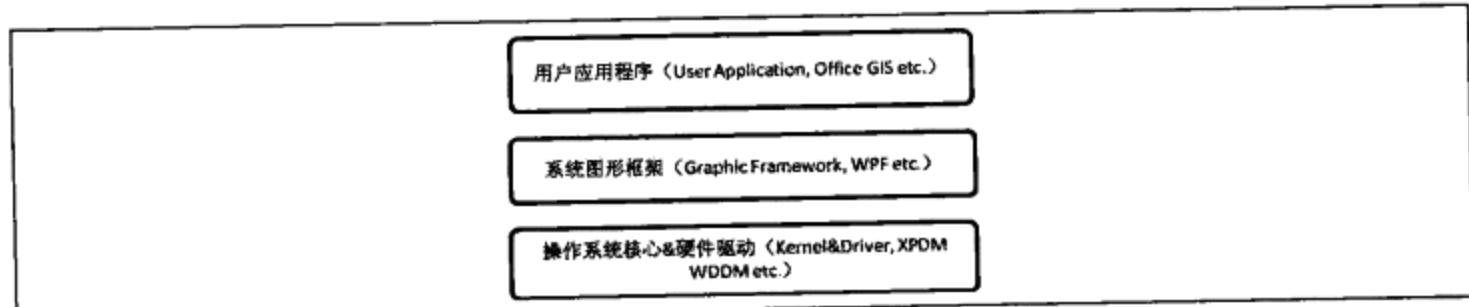


图 3-2 Windows 大厦的 3 层体系结构

第 2 层实际上是一个“复式结构”，如图 3-3 所示。

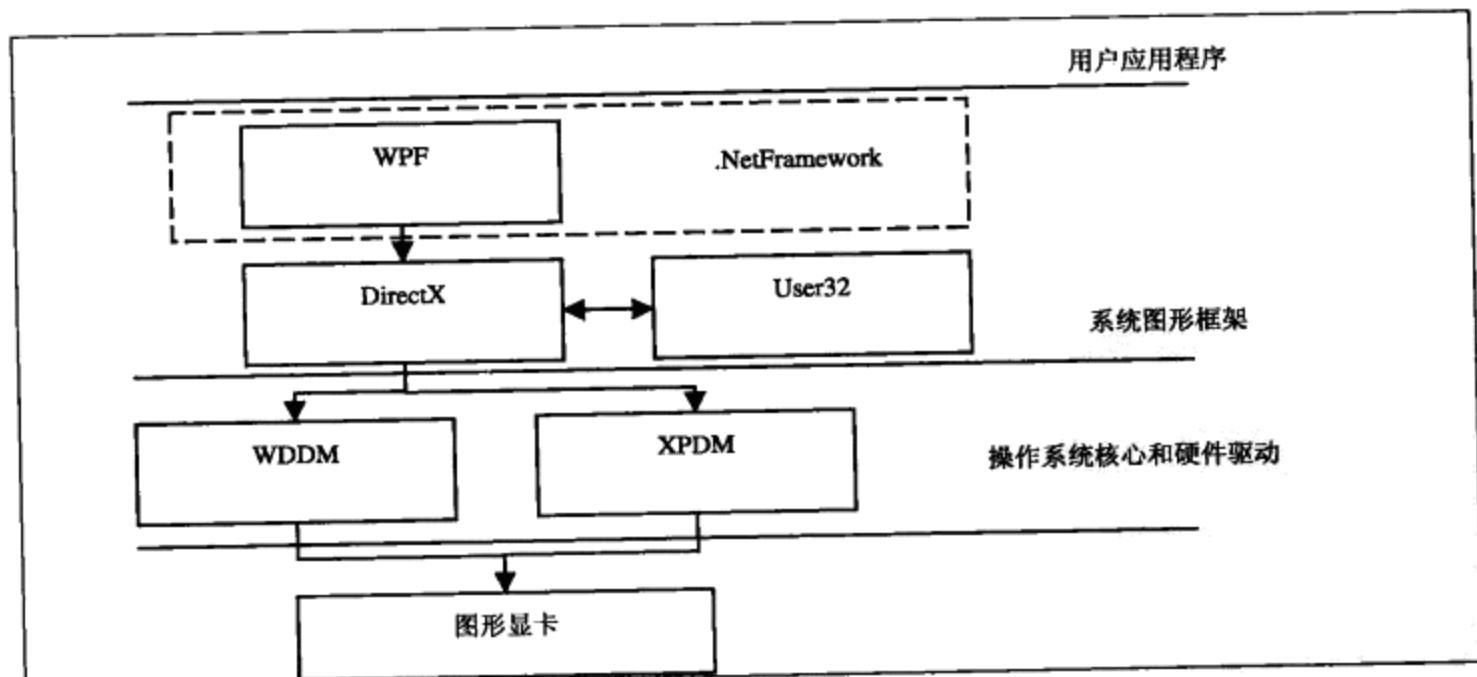


图 3-3 第 2 层为“复式结构”

在系统图形框架层中可以看到 WPF 的底层基于 DirectX 和 User32，前者负责渲染，后者负责响应窗

口消息。从中可以看出 WPF 是.NET Framework 的一个组成部分。窗口中的按钮或者文本最终都要变化成 DirectX 可以接受的三角形、材质和其他 Direct3D 对象时，然后，DirectX 会调用相应的渲染驱动模型（WDDM 或者 XPDPM），使用图形显卡渲染这些数据。

3.2 WPF 内部结构

揭开 WPF 内部结构的真相并非易事，需要层层剥离才能山高月小，水落石出。

3.2.1 切入点之一：托管和非托管的界限

在 CLR (Common Language Runtime, 公共语言运行时) 之上的托管模块稳定性好，并且开发简便，可以迅速构建应用程序；非托管模块则更接近底层，会更为高效一些，二者分别位于 WPF 的上层和底层。WPF 暴露给开发人员的都是托管的 API 接口，而在和 DirectX 交互时则需要用到非托管的代码，WPF 的内部结构如图 3-4 所示。

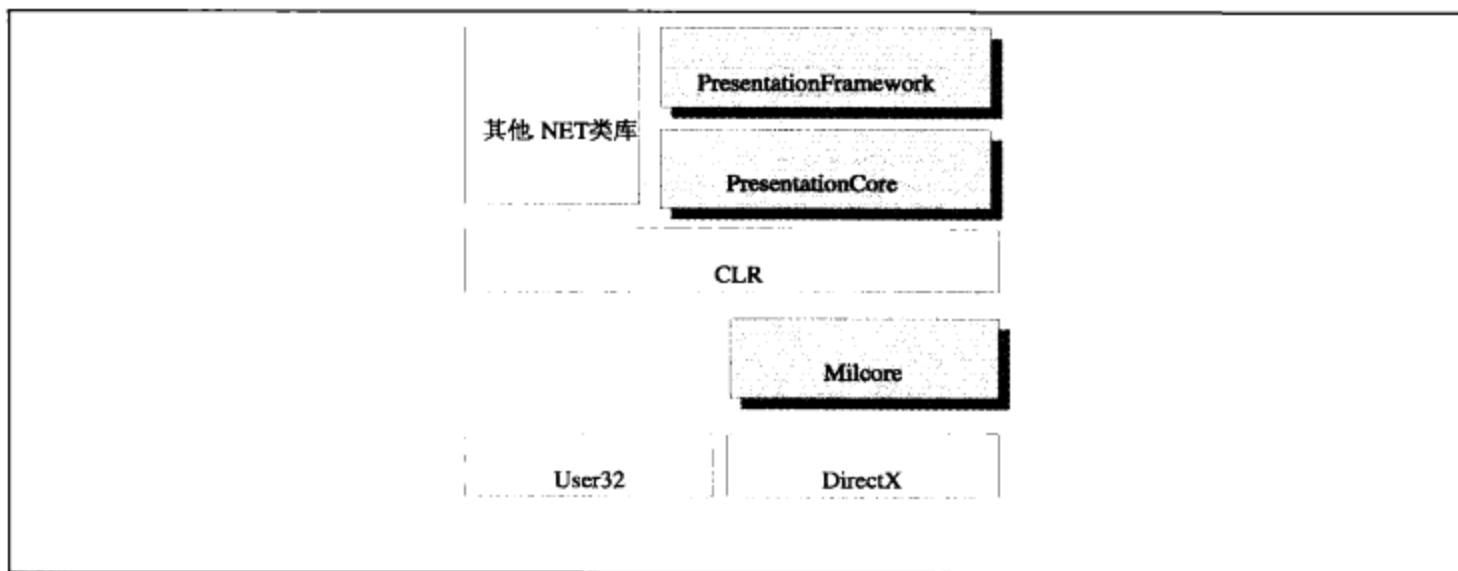


图 3-4 WPF 内部结构

图中加阴影的模块是 WPF 的主要组件，CLR 之上是托管代码，之下是非托管代码。由此可见，WPF 中只有组件 Milcore 是非托管的，是一个和 DirectX 交互的非托管组件。之所以使用非托管的方式实现，一方面是为了和 DirectX 结合更为紧密；另一方面则是不惜牺牲 CLR 的诸多优点来换取性能上的优势。CLR 之上的 PresentationCore 组件提供了一组基本服务，如事件处理、输入及布局等。在其提供的基础功能之上，组件 PresentationFramework 实现了 WPF 中的各种外观，如按钮及图形效果的实现。

3.2.2 切入点之二：WPF 如何实现绘制

要了解 WPF 如何实现绘制，先要清楚 WPF 绘制所用的数据结构。

1. WPF 绘制所用的数据结构

使用 XAML 文件描述 WPF 应用程序界面时，其组织结构从根节点自顶向下看起来就像一棵树，称为“逻辑树”（Logic Tree），图 3-5 所示为页面中一个按钮的逻辑树结构。

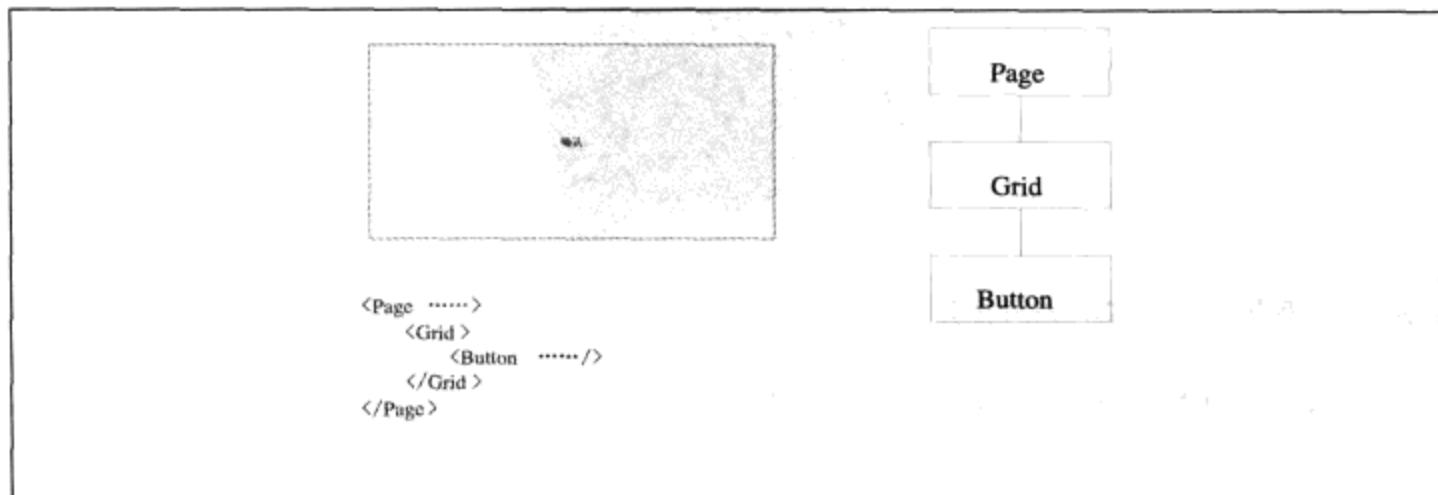


图 3-5 页面中一个按钮的逻辑树结构

逻辑树在描述用户界面的组成、依赖属性的继承（参见第 6 章）及路由事件（参见第 7 章）的传递时有相当大的作用，但是在界面可视化方面用其来描述界面元素的组成远远不够。实际上 WPF 将每一个小的可视化单元都看做是一个 Visual，这个 Visual 类似早期程序世界中的窗口。每个小的 Visual 同样也是自顶向下组成了一棵树，称为“可视化树”（Visual Tree），其中的每个节点都保存绘制该节点所需要的命令。如图 3-6 所示，图中灰底的节点既是逻辑树节点，也是可视化树节点。不难发现，可视化树比逻辑树在描述用户界面时更为详尽。

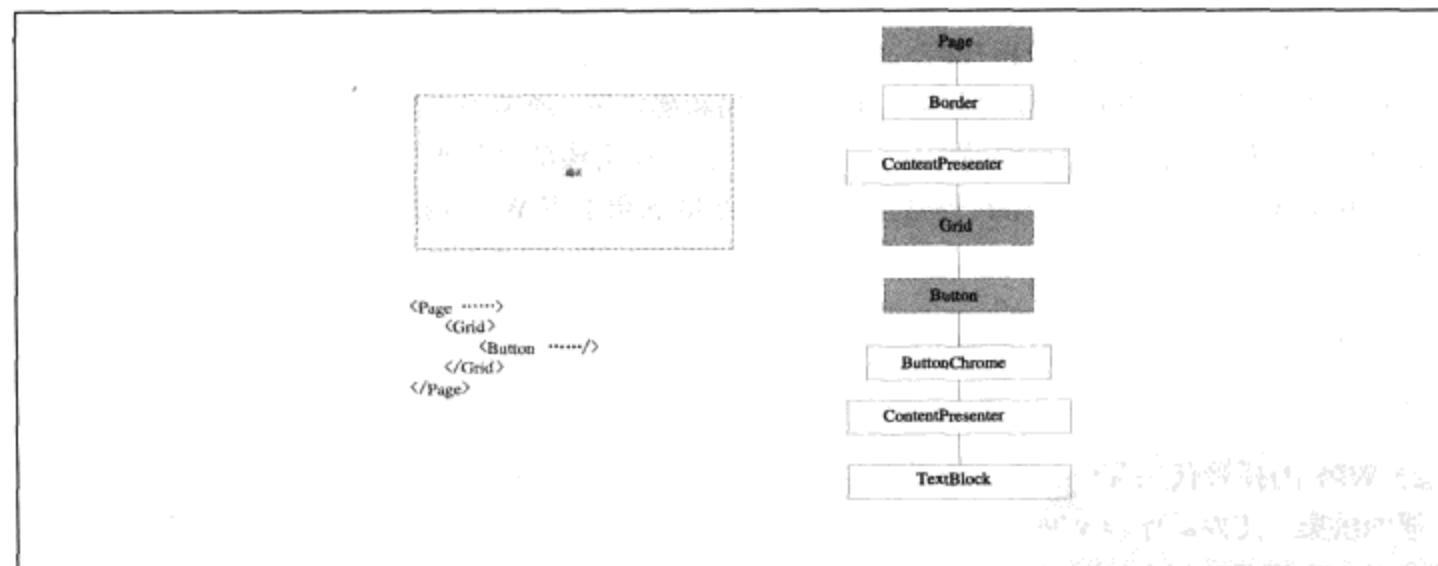


图 3-6 可视化树结构

WPF 所认识的树实际上是一种称为“Composition Tree”的树，其中的每个节点称为“Compositon Node”。

构成 3D 图形的基本图元是一个一个三角形，由三角形及其材质或者纹理构成复杂的几何图形，图 3-7 所示为三角形面片构成的地形表面。

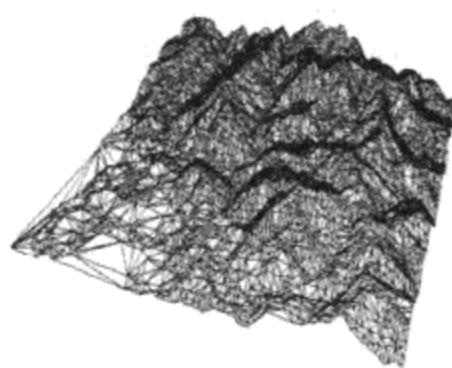


图 3-7 三角形面片构成的地形表面

图 3-8 所示为不同数量的三角形构成的兔子模型。

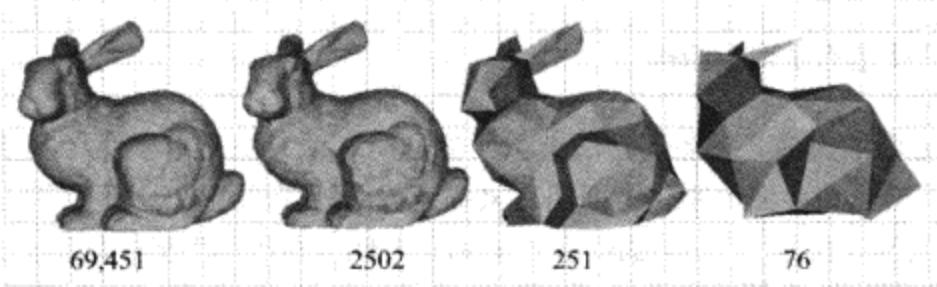


图 3-8 不同数量的三角形构成的兔子模型

2. 从绘制的角度分析 WPF 体系架构

一个 WPF 应用程序从两个线程开始，分别负责用户界面（UI）的管理和渲染。如前所述，托管代码为用户提供构建 WPF 所需要的各种功能，如布局和绑定等，通常用户界面管理线程使用该部分代码；非托管代码主要负责图形的叠加显示和渲染，那么使用该部分代码的是不能被用户接触的渲染线程，两个线程之间通过消息来传递数据。我们从绘制的角度将 WPF 内部结构图进一步细化，如图 3-9 所示，说明如下。

(1) 在最上层是逻辑树，如在页面中添加一个按钮或者一个矩形，其实质都是在逻辑树上添加一个节点。

(2) WPF 的托管代码部分包含一个重要的子系统，即 Visual System，它是托管代码与非托管代码沟通的桥梁，其内部保存 WPF 程序需要显示的可视化树结构。从可视化树的角度来说，其实质是添加多个 Visual 到可视化树。

(3) 前面两步均在用户界面线程中，添加 Visual 之后用户界面线程和渲染线程会通过消息传递可视化数据。

(4) 非托管代码部分（Milcore）又称为“媒体整合层”（Media Integration Layer）。其中包含了合成子系统（Compositon System）和渲染引擎（Rendering Engine）。合成子系统接受 Visual 之后会将其转换为 Composition 节点，并添加入到 Composition Tree 中。

(5) 渲染引擎将 Composition Tree 转换为 DirectX 可以识别的三角形，这个过程称为“Tessellate”（三角剖分）。

(6) DirectX 经过驱动（WDDM 或者 XPDM）通知显卡开始绘制像素到屏幕。

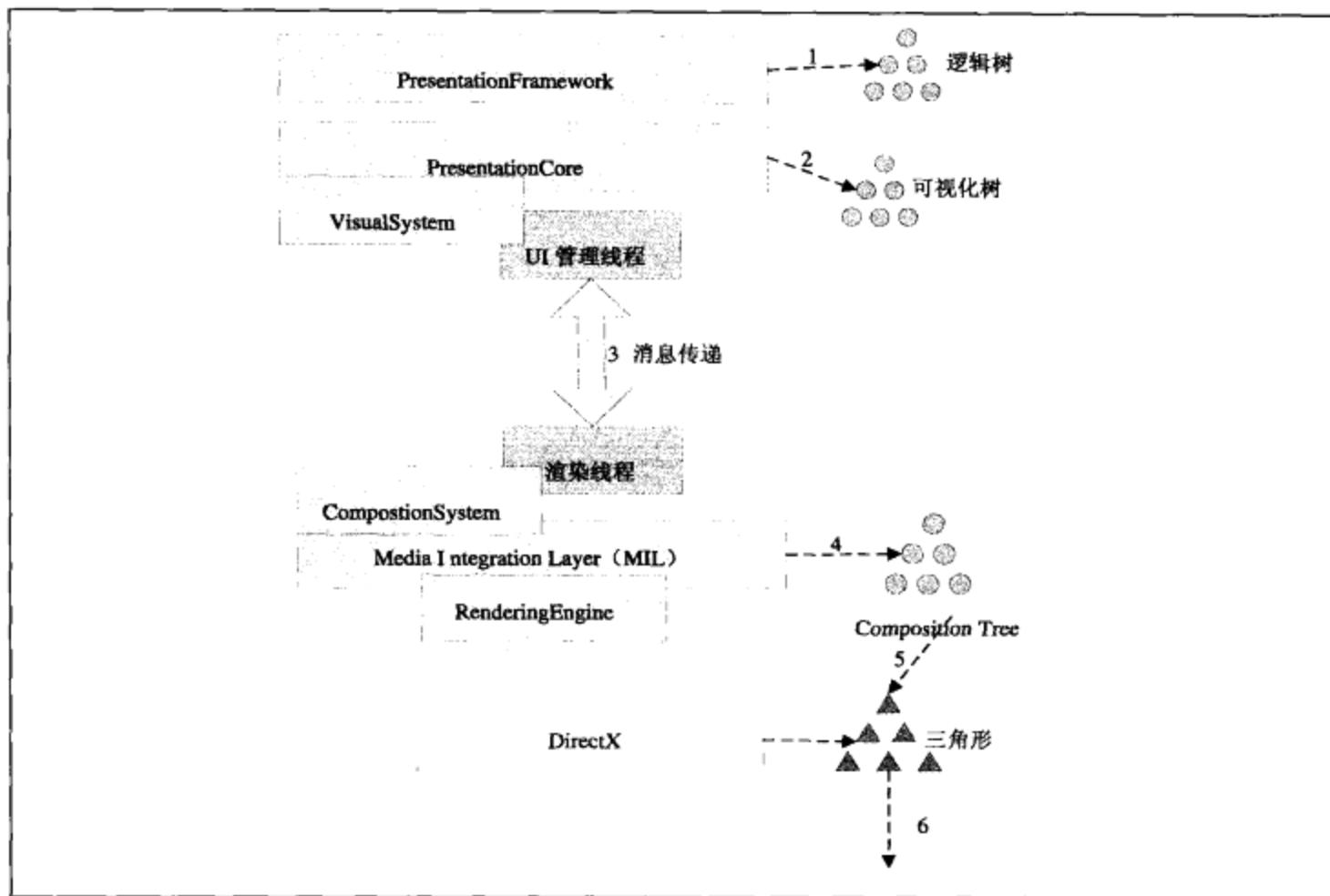


图 3-9 WPF 内部结构图

需要 Composition Tree 的原因如下。

(1) 可视化树只存在于用户界面线程中，如果渲染线程需要绘制 WPF 的用户界面，则也需要一个数据结构。这个数据结构就是 Composition Tree，它存在于渲染线程中；

(2) 渲染引擎并不知道可视化树，它只知道 Composition Tree，并将其转换为 DirectX 可以识别的三角形。

3.2.3 切入点之三：WPF 类层次结构

WPF 的类主要集中在 WindowsBase.dll、PresentationCore.dll 和 PresentationFramework.dll 程序集中。

WPF 的命名空间均以 System.Windows 开头，如 System.Windows.Controls 是关于 WPF 控件的类；System.Windows.Media 则是 WPF 中关于绘图、画刷、视频和音频相关的类。唯一例外的是 System.Windows.Forms，它主要是 Windows Form 编程的相关类。但是命名空间下的类并不都属于

Windows Form，也有一个例外的，就是 System.Windows.Forms.Integration。这个命名空间中的类用来集成 Windows Form 和 WPF 程序。某一个命名空间也不是专属于某个程序集，它们之间是一种“多对多”的关系。如图 3-10 所示，System.Windows 命名空间出现在 WindowBase.dll，PresentationCore.dll 和 PresentationFramework.dll 程序集中，WindowBase.dll 则可以包含 System.Windows 及 System.Windows.Data 等若干命名空间。

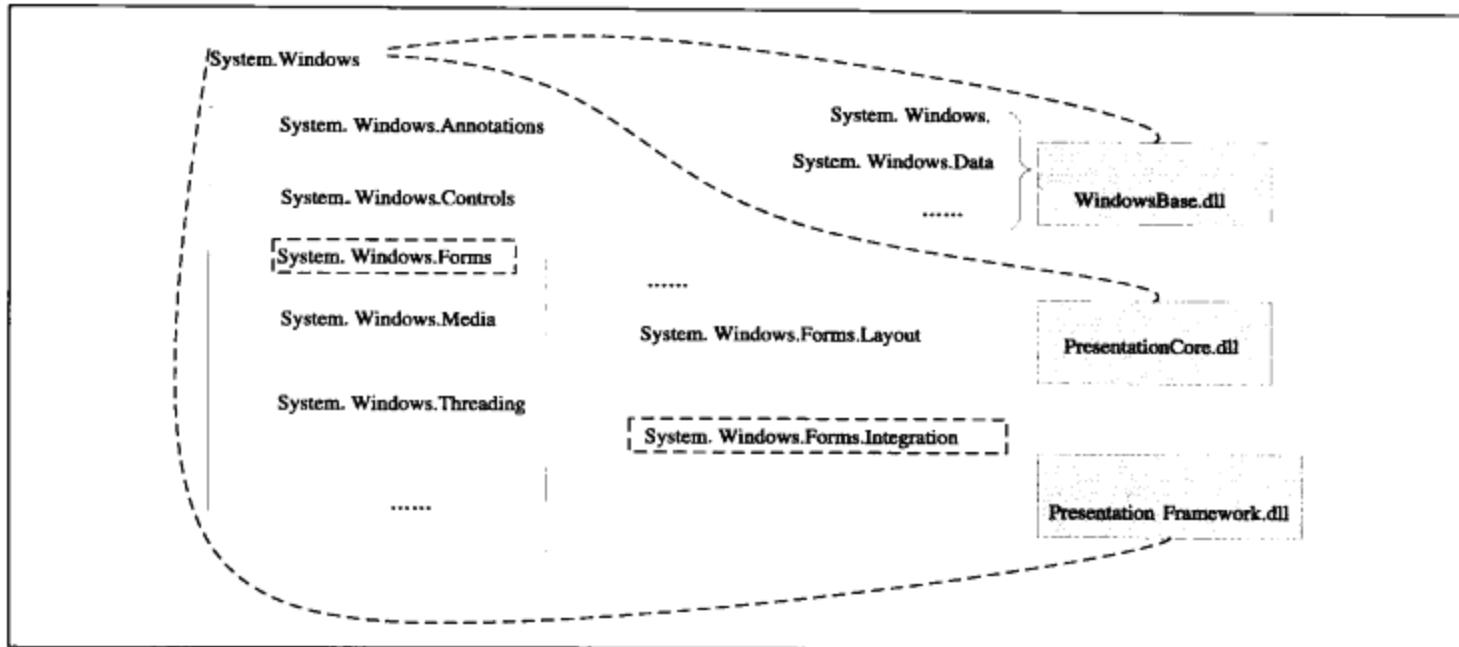


图 3-10 命名空间和程序集对应的关系

WPF 的类图虽然复杂，但主要的类结构如图 3-11 所示。

(1) Object (System.Object)

该类型是所有.NET 对象的父类。

(2) DispatcherObject (System.Threading.DispatcherObject)

大多数的 WPF 类型都派生自 DispatcherObject，如果不涉及多线程，也许并不用与其打交道。

(3) DependencyObject (System.Windows.DependencyObject)

这是所有能够支持依赖属性的基类，即如果希望一个类能够支持依赖属性（参见第 6 章），则必须从该类派生。

(4) Freezable (System.Windows.Freezable)

从该类型派生的类非常多，包括其中的画刷(Brush)、时间线(Timeline，用于动画)及几何(Geometry)等类型。Freezable 有一个 Changed 事件，可以通知 WPF 其值发生改变。此外它的 Freeze 方法可以将自身“冻结”起来，变成一个只读状态的对象。通常在两个线程共享或者提高效率时，需要将对象“冻结”。

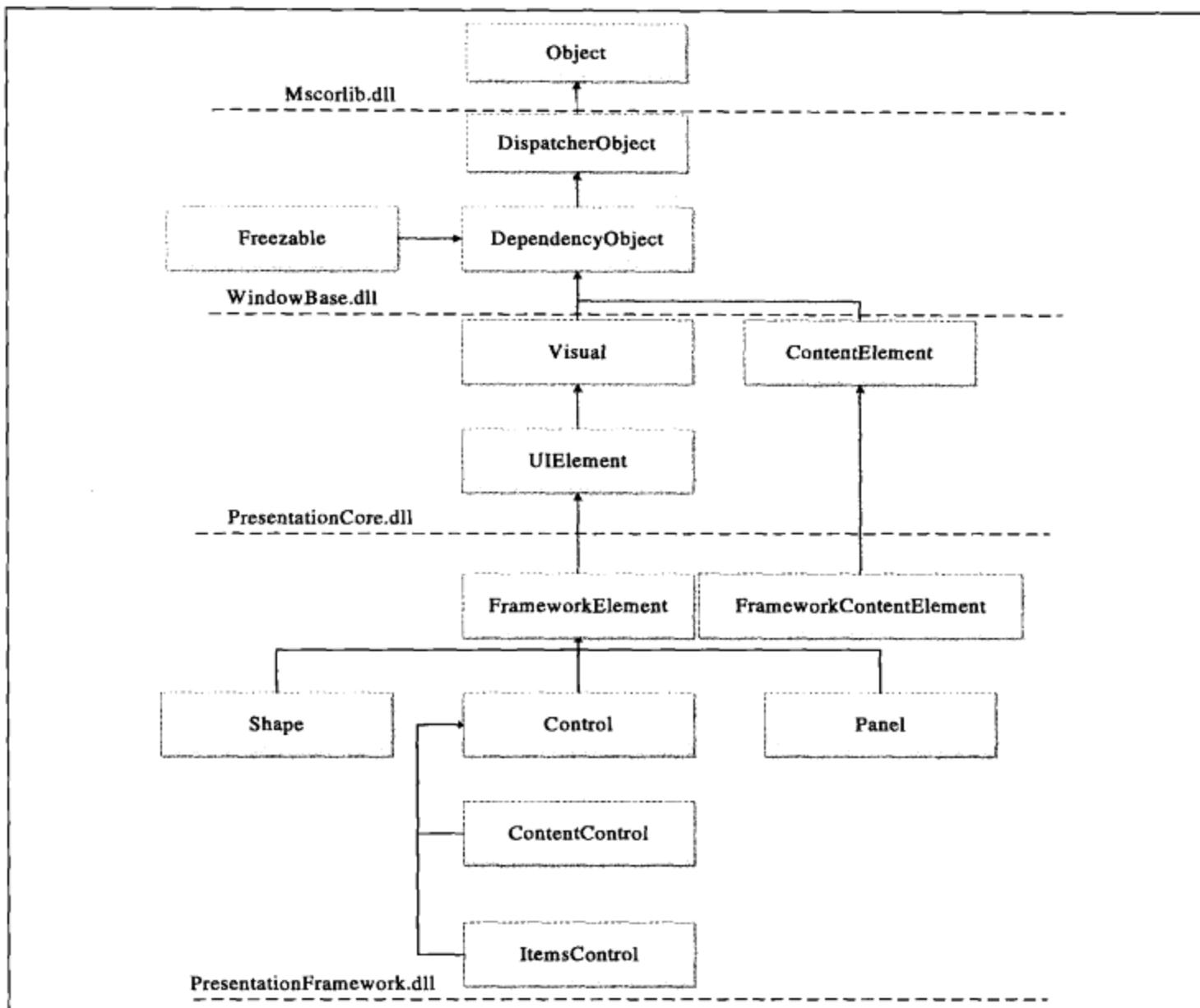


图 3-11 WPF 主要的类结构

(5) Visual (System.Windows.Media.Visual)

`Visual` 是可视化树的一个节点，包含绘制自身的所有命令。它是一个抽象类，主要作用是为 WPF 提供可视化支持，包括输出显示、透明度、坐标转换及区域剪切等。

(6) UIElement (System.Windows.UIElement)

`UIElement` 派生自 `Visual`，在 `Visual` 的基础上增强了 3 个方面的内容，即布局、输入和事件。从 `UIElement` 开始，可视化树慢慢淡去，逻辑树慢慢显现。

(7) FrameworkElement (System.Windows.FrameworkElement)

`FrameworkElement` 类派生自 `UIElement`，在其基础上增强了数据绑定、样式及资源等 WPF 的核心功能。

(8) ContentElement (System.Windows.ContentElement)

`ContentElement` 类似 `UIElement`，不同的是该类型无法将自己可视化出来；必须借助于一个派生自 `Visual` 的类才能显示在屏幕上，它和 `FrameworkContentElement` 主要用在流文档模型中（参见第 18 章）。

(9) `FrameworkContentElement` (`System.Windows.FrameworkContentElement`)

`FrameworkContentElement` 类似 `FrameworkElement`，也支持动画及数据绑定等特性。不同的是该类型无法将自己可视化出来，需要借助一个派生自 `Visual` 的类才能显示在屏幕上。

(10) `Shape` (`System.Windows.Shapes.Shape`)

这个类是基本形状的基类，用于创建基本的图形，如长方形、多边形、椭圆、线和路径等。由于它派生自 `FrameworkElement`，因此也具备一些元素的特性。如可以响应鼠标和键盘消息等，这一点 WPF 和以往的平台框架差别较大。

(11) `Control` (`System.Windows.Controls.Control`)

这个类是控件的基类，控件就是可以与用户交互的元素，如文本框、按钮、列表框等。WPF 的控件可以支持控件模板，使得控件的外观可以随心所欲地变化（请参见第 13 章）。需要注意在以往的 Windows 编程中窗口中所有可视化内容都称为“控件”。但在 WPF 中不再如此。可视化内容被称为“元素”（Element），对控件模板的支持是判别控件和元素的一个重要标准。

(12) `ContentControl` (`System.Windows.Controls.ContentControl`)

`ContentControl` 派生自 `Control`，是具有单一内容的控件，内容指可以在其放置文字、图片及视频等任何类型数据的控件（请参见第 11 章）。

(13) `ItemsControl` (`System.Windows.Controls.ItemsControl`)

这是所有能够支持多个条目显示的控件基类，如列表框和树形视图。与 `ContentControl` 不同的是，其内容是所有显示选项的集合，因此具有很好的灵活特性。实际上，WPF 菜单、工具栏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。

(14) `System.Windows.Controls.Panel`

面板（Panel）可以用做所有布局的容器，可以包含一个或多个子控件并且可以将其按照布局单位排列。这些容器是 WPF 布局系统的基础，并且合理使用容器是灵活布局界面内容的关键（请参见第 10 章）。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作，《金庸全集典藏版 鹿鼎记》，“第三十四回 一纸兴亡看复鹿 千年灰劫付冥鸿”。

- [2] Leonardo Blanco, Development Lead Desktop and Graphics Team Microsoft Corporation, Windows Graphics Architecture WinHEC 2007.
- [3] MSDN Library for Visual Studio 2008 SP1 WPF Architecture.
- [4] 张晗雨编著,《WPF 全视角分析》,机械工业出版社,5~7页。
- [5] Matthew MacDonald 著,王德才译,《WPF 编程宝典——使用 C# 2008 和.NET 3.5 (第 2 版)》,13~17页。
- [6] 周永恒,《一站式 WPF——Window (一)》博客园。



心法



XAML——反两仪刀法和正两仪剑法

两人刀剑相交，各自退开一步。不禁一怔，心中均十分佩服对方这一招的精妙。两人派别不同，武功大异，生平从未见过面。但一招之下，发觉自己这套武功和对方若合符节，配合得天衣无缝。犹似一个人一生寂寞，突然间遇到了知己般的喜欢。

——《倚天屠龙记》，“第二十一章 排难解纷当六强”

这一段讲的是张无忌在明教光明顶迎战六大门派一幕，华山派耆宿高矮老者使的是反两仪刀法，而昆仑派掌门何太冲和他的妻子班淑娴使的是正两仪剑法。两仪剑法和两仪刀法虽然正反有别，但均是太极化为阴阳两仪之理，因此配合得天衣无缝。即使像张无忌这样身负九阳神功和乾坤大挪移两大绝学，也险些败下阵来。

在 WPF 之前设计图形用户界面往往使用一种语言。但在 WPF 当中，由于引入了 XAML 语言。因此在界面设计方面，一般使用 XAML 语言，而在业务逻辑上使用 C# 或者 VB 这样的后台代码。XAML 和后台代码既可以配合得丝丝入扣，又可以将界面设计和业务逻辑分离。这就好比两仪剑法和两仪刀法，一正一反。但均是从八卦中化出，再回归八卦，可说是殊途而同归。

本章内容如下：

- (1) 从 C# 到 XAML。
- (2) 命名空间及其映射。
- (3) 简单属性和附加属性。
- (4) 类型转换器。
- (5) 标记扩展。
- (6) 分别使用 XAML 和 C# 构建应用程序——刀还是刀，剑依旧是剑。
- (7) 使用 XAML 和 C# 构建应用程序——刀剑合璧。

4.1 从 C# 到 XAML

XAML 与 C# 不同，但与 HTML 和 XML 一样是一种声明式语言。代码 4-1 所示为一段完整的 XAML 代码 (mumu_button.xaml)：

```
<!--开始标签(Start Tag)-->
```

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Width = "300" Height="200"><!--属性-->
    <!--两者中间的内容-->
    <Button.Content>
        Hello XAML
    </Button.Content>
</Button>
<!--结束标签(End Tag)-->
```

代码 4-1 一段完整的 XAML 代码

它由开始标签、结束标签及二者中间的内容组成，开始标签中包括一个命名空间（xmlns）和两个属性说明。将这段代码复制到 XamlPad，可以看到它描述的是一个普通的 WPF 按钮，其可视化结果如图 4-1 所示。

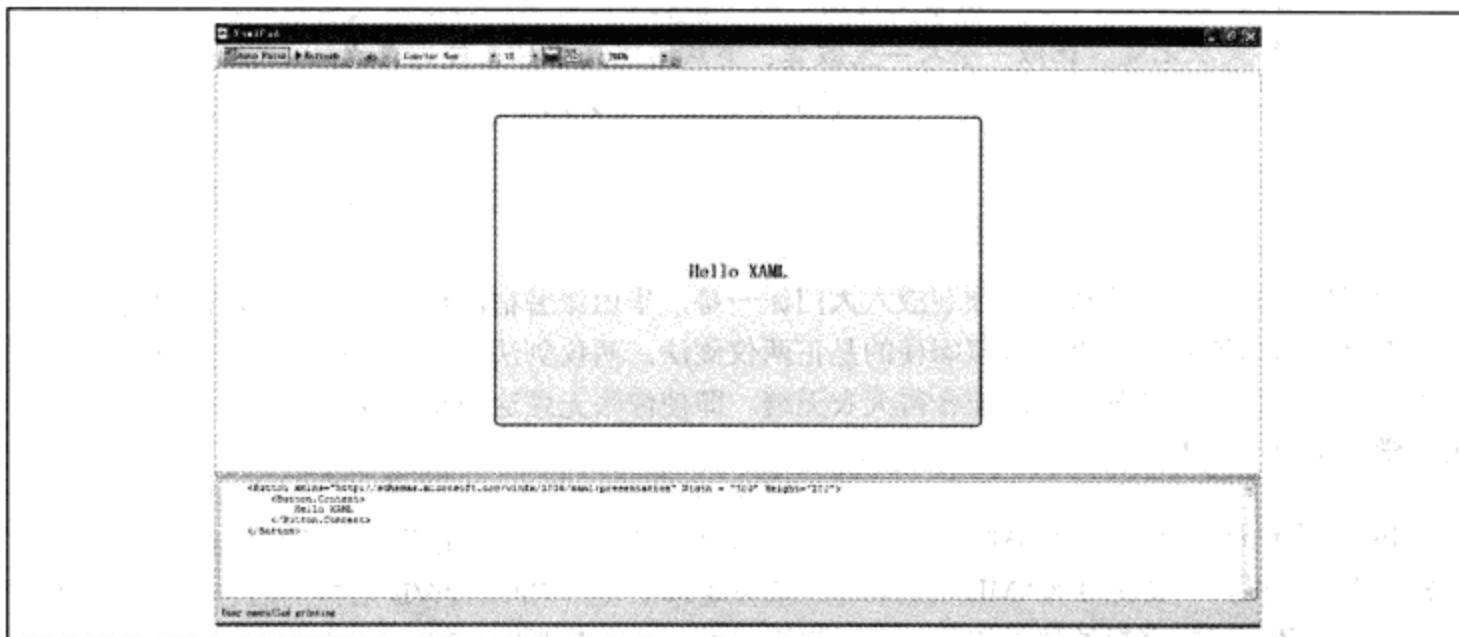


图 4-1 mumu_button.xaml 的可视化结果

代码 4-2 是一个更为复杂的 XAML 文件的例子（mumu_window.xaml）：

```
<Window Title="XAML 窗口" xmlns="http://schemas.microsoft.com/winfx/ 2006/
xaml/presentation" Width = "300" Height="200" >
    <DockPanel>
        <Button DockPanel.Dock="Left"
Background="AliceBlue" Margin="0 5 0 10" Content="Hello XAML"/>
        <Button DockPanel.Dock="Right">
            <Button.Background>
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                    <GradientStop Color="Yellow" Offset="0.0" />
                    <GradientStop Color="Red" Offset="0.25" />
                    <GradientStop Color="Blue" Offset="0.75" />
                    <GradientStop Color="LimeGreen" Offset="1.0" />
                </LinearGradientBrush>
            </Button.Background>
            Hello XAML
        </Button>
    </DockPanel>
</Window>
```

代码 4-2 mumu_window.xaml 文件

其中描述包含一个面板（DockPanel）的窗口（Window），在面板的左右侧各有一个按钮。需要注意的是，窗口不会像前面的按钮一样直接在 XamlPad 里显示，需要 F5 键运行才能弹出窗口来。如图 4-2 所示。

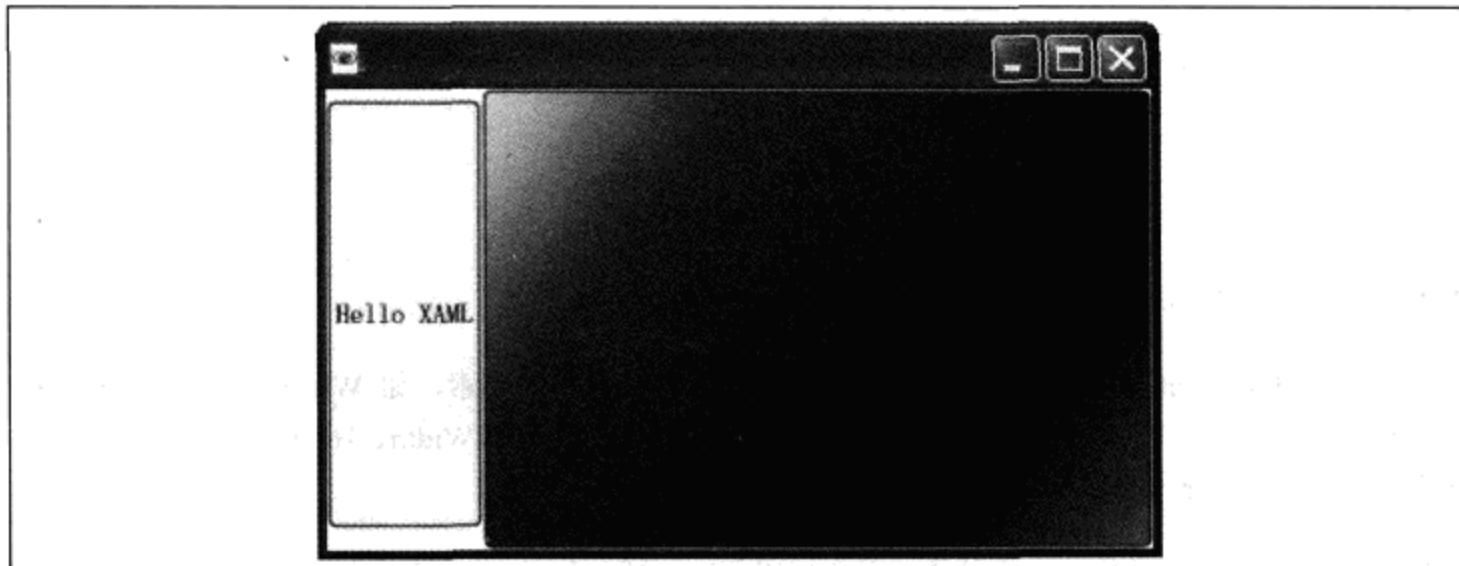


图 4-2 mumu_window.xaml 运行结果

这段 XAML 代码相当于如代码 4-3 所示的 C# 代码：

```
public class MyWindow : Window
{
    [STAThread]
    public static void Main()
    {
        MyWindow win = new MyWindow();
        win.Show();
        Application app = new Application();
        app.Run();
    }
    public MyWindow()
    {
        InitializeComponent();
    }
    private void InitializeComponent()
    {
        this.Title = "XAML 窗口";
        this.Width = 300;
        this.Height = 200;
        DockPanel panel = new DockPanel();

        Button btn = new Button();
        btn.Content = "Hello XAML";
        btn.Background = new SolidColorBrush(Colors.AliceBlue);
        btn.Margin = new Thickness(0, 5, 0, 10);
        DockPanel.SetDock(btn, Dock.Left);
        panel.Children.Add(btn);

        Button btn2 = new Button();
        btn2.Content = "Hello XAML";
        LinearGradientBrush brush = new LinearGradientBrush();
        brush.StartPoint = new Point(0, 0);
        brush.EndPoint = new Point(1, 1);
```

```

        brush.GradientStops.Add(new GradientStop(Colors.Yellow, 0));
        brush.GradientStops.Add(new GradientStop(Colors.Red, 0.25));
        brush.GradientStops.Add(new GradientStop(Colors.Blue, 0.75));
        brush.GradientStops.Add(new GradientStop(Colors.LimeGreen, 1));
        btn2.Background = brush;
        DockPanel.SetDock(btn2, Dock.Right);
        panel.Children.Add(btn2);
        this.Content = panel;
    }
}

```

代码 4-3 C#代码

不难发现，XAML 代码在描述界面上会比 C# 简洁得多。

XAML 文件有两个重要组成部分：一是有完整开始和结束标签的要素，如 Window、DockPanel 和 Button 等，称为“元素”（Element）；二是依附于元素的要素，如 Width、Height 和 Background，称为“属性”（Attribute）。

在 C# 代码中创建一个对象并为其赋值等都会使用 using 关键字声明其命名空间，在 XAML 中也如此。在 Window 元素中的 xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation 就是声明的命名空间，该命名空间貌似一个网站。但是不要企图在 IE 地址栏中输入了该地址，因为根本就不会有这样一个网站，它仅仅是一个命名空间的标示而已。

在 C# 中通过 new 关键字创建一个 Button 对象，而在 XAML 中是通过一个开始标签和结束标签来实现的。还有一种更简洁的写法是用一个反斜线 “/” 取代结束标签，称为“empty-element 语法”，如代码 4-4 所示。

```
<Button DockPanel.Dock="Left"
       Background="AliceBlue" Margin="0 5 0 10" Content="Hello XAML"/>
```

代码 4-4 第 1 个按钮的代码

新建完对象后，接下来的事情就是要为该对象设置属性。在 C# 里设置属性是一件非常简单的事情，double 类型的属性就给它赋予 double 类型的值，string 类型的属性就给它赋上 string 类型的值等。但是在 XAML 里，无论你的属性是 double 还是 string，你能给属性赋的值统统都是字符串。表 4-1 我们列举了一些典型的属性赋值的例子。

其中仅有 Window 的 Title 属性类型是字符串，在 C# 和在 XAML 中均直接将值设置为字符串。此外 Window 的 Width 类型也是一种基本型 double，在 XAML 中给该类型的属性赋值也非常简单，它们均属于“简单属性”。DockPanel.Dock 不是 Button 的属性，在 XAML 中将其值设置为 Left，表示该按钮在 DockPanel 的左侧。在 C# 中将其转换为一种方法，这种属性称为“附加属性”。

表 4-1 在 C# 和 XAML 中设置属性的方法

类	属性	C#	XAML
Window	Title	this.Title = "XAML 窗口";	Title="XAML 窗口"
Window	Width	this.Width = 300;	Width = "300"

续表

类	属性	C#	XAML
Button	DockPanel.Dock	DockPanel.SetDock(btn, Dock.Left);	DockPanel.Dock="Left"
Button	Background	<pre> btn.Background = new SolidColorBrush(Colors.AliceBlue); LinearGradientBrush brush = new LinearGradientBrush(); brush.StartPoint = new Point(0, 0); brush.EndPoint = new Point(1, 1); brush.GradientStops.Add(new GradientStop(Colors.Yellow, 0)); btn2.Background = brush; </pre>	Background="AliceBlue" <Button.Background> <LinearGradientBrush StartPoint="0,0" EndPoint="1,1"> </LinearGradientBrush> </Button.Background>
Button	Content	btn.Content = "Hello XAML";	Content="Hello XAML"

Button 的 Background 属性是一个画刷类型，第 1 个按钮的该属性在 C# 中创建了一个画刷对象，然后将该对象赋给属性。在 XAML 中直接用一个“AliceBlue”字符串为其赋值，使得更为简洁，这正是类型转换器起到的作用，我们将在第 4.5 节类型转换器里一探究竟；第 2 个按钮的背景色是一个渐变画刷。Background 属性没有写在元素的标签中，而是如同一个子元素写在 Button 的开始和结束标签之间。前面设置 Background 属性的方法称为“Attribute 语法”，后一种称为“Property-Element 语法”。“Property-Element 语法”得名是因为这个时候的 Background 属性设定更像是一个元素设定的方法，但是由于本质上还是一个属性，因此是 Property-Element。

Button 的 Content 属性是一个特殊的属性，其类型为 Object。它不仅仅是一个属性，更确切地说是一个嵌套的元素。从理论上来说它可以设置为任何类型的对象。本章将在第 4.4 节 Content 属性来讨论这个神奇的属性。

此外在 C# 里还有很多情况是一个属性会去引用一个对象，或者将一个属性赋值成空。在 XAML 里该如何实现呢？我们会在第 4.6 节标记扩展里讨论这个问题。还有事件的处理函数该如何写，如何使用 XAML 构建一个应用程序，或者混合使用 XAML 和 C# 构建应用程序？我们在第 4.7 节分别使用 XAML 和 C# 构建应用程序——刀还是刀，剑依旧是剑和第 4.8 节使用 XAML 和 C# 构建应用程序——刀剑合璧里寻求答案。

4.2 命名空间及其映射

4.2.1 WPF 的命名空间

前面我们已经看到在 XAML 中声明的一个命名空间是一个 URL，即 <http://schemas.microsoft.com/winfx/2006/xaml/presentation>。

这是 WPF 命名空间的标识，XAML 文件规定必须至少指定一个 XML 的命名空间。为了使该命名空间作用于整个文件，通常会将其放置在根元素中，如代码 4-5 所示。

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button Width = "300" Height="200">
        <Button.Content>
            Hello XAML
        </Button.Content>
    </Button>
</Page>
```

代码 4-5 mumu_page.xaml 文件

在 WPF 中只有以下 4 种元素可以作为根元素。

- (1) Window：代表一个窗口。
- (2) Page：类似一个网站的页面。
- (3) Application：代表一个应用程序（参见第 8 章）。
- (4) ResourceDictionary：代表一个逻辑资源的集合（参见第 12 章）。



木木不禁有些奇怪，在前面 XAML 代码中的根元素不是 Button 吗？该文件在 IE 浏览器和 XamlPad 都能够正常显示，这是为什么？

实际上这种松散 XAML 文件会和一个 PresentationHost.exe 程序关联，只要双击该文件，就会执行 PresentationHost。该程序发现没有根元素，则建立一个 Page 类型的对象，然后将其设置为 Page 的 Content 属性。看上去根元素是 Button，但实际上还是 Page。任何继承自 FrameworkElement 的元素都可以看上去作为根元素，但只有 Window 对象不可，因其不能作为某个元素的子元素。

我们知道 WPF 中不同的控件分属于不同的命名空间，如 Window 类型属于 System.Windows 命名空间，而 Button 属于 System.Windows.Controls 命名空间。如果在 C# 代码中用到这些控件，则必须连续使用 using 语句，直到包括所有用到的命名空间，如代码 4-6 所示。

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

代码 4-6 在 C# 中引用命名空间

在 WPF 中一个 URL 标识就可全部涵盖所有用到控件的命名空间，原因是其在 PresentationFramework 和 PresentationCore 这样的程序集中使用 XmlNsDefinition 将 XML 和 CLR 命名空间关联起来。下面是通过 Reflector 查看到的程序集属性，可以看到已经将 http://schemas.microsoft.com/winfx/2006/xaml/presentation 与和 System.Windows 及 System.Windows.Controls 相关联。XAML 解析器会检查应用程序所加载的所有组件的 XmlNsDefinition 属性，以此确定对应的 CLR 命名空间：

```
[assembly: XmlNsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation", "System.Windows.Controls")]
[assembly: XmlNsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation", "System.Windows.Documents")]
```

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation", "System.Windows.Shapes")]
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation", "System.Windows.Data")]
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation", "System.Windows")].
....
```

如果想确切地知道这个 XML 的命名空间对应的 CLR 命名空间，可以使用在本章示例代码中提供的工具（mumu_dumpCLRNamespace 工程）查看。图 4-3 所示为 PresentationFramework 程序集中与 `http://schemas.microsoft.com/winfx/2006/xaml/presentation` 对应的 CLR 命名空间。



图 4-3 PresentationFramework 对应的 CLR 命名空间

4.2.2 XAML 的命名空间

第 2 个常用的命名空间是 XAML 的命名空间，如果需要使用 XAML 专用的元素和属性（事实上一定会使用），那么必须声明该声明空间。习惯上 XAML 的声明空间被声明成 x 前缀 `xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`，一般使用 VS 2010 生成的默认的 XAML 文件都会包含 WPF 和 XAML 命名空间的声明，如代码 4-7 所示。

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

代码 4-7 WPF 和 XAML 命名空间的声明（mumu_customnamespace 工程中的 Window1.xaml 文件）

其中 Window、Grid 和 Button 属于 WPF 命名空间中的元素，`x:Class` 属于 XAML 命名空间中的属性。由于 WPF 命名空间是默认空间，因此 Window、Grid 和 Button 没有声明前缀。



木木还是产生了一些疑问，既然是 XAML 文件，为什么不将 XAML 的命名空间而将 WPF 命名空间作为默认空间？这是由于用到的 WPF 的元素和属性过多，而 XAML 命名空间的元素和属性相对较少。所以为了避免在 XAML 文件中全部添加类似 x:这样的前缀，而将 WPF 命名空间作为默认空间。

木木不禁还想问，XAML 命名空间里的元素和属性比较少，那有多少？

木木 XAML 中的元素和属性如下表所示。

x:Class	x>TypeArguments
x:ClassModifier	x:Uid
x:Code	x:xData
x:FileModifier	x:Array
x:Name	x:Null
x:Shared	x:Static
x:Subclass	x>Type

木木又问，这些元素和属性的含义是什么？如何用？有例子吗？

木木 其中部分元素和属性很常用，如 x:Class 和 x:Name 等；部分偶尔会用到，如 x:Null、x:Shared、x>Type、x:Array 和 x:Type 等。这些只要稍稍留心都会在本书中找到例子。

木木最后问，WPF 命名空间里的元素和属性很多，有多少？

木木 拖出去砍了！不予回答。

4.2.3 其他命名空间

1. 使用系统类

除去前面两个命名空间，如果 XAML 文件要使用其他.NET 对象或者应用程序或者其他程序集中的自定义对象，也要声明命名空间，如代码 4-1 所示的 XAML 文件。

如果希望明确地将 Button 的 Content 属性设置为 String 类型（尽管这样做是画蛇添足），由于 String 类型属于程序集 mscorelib.dll 且其命名空间为 System，所以在声明后才能使用，如代码 4-8 所示。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:s="clr-namespace:System;assembly=mscorelib"
        Width="300" Height="200">
    <Button.Content>
        <s:String>
            Hello XAML
        </s:String>
    </Button.Content>
</Button>
```

代码 4-8 mumu_button2.xaml 文件

注意通过声明 xmlns:s="clr-namespace:System;assembly=mscorelib" 关联 System 命名空间和前缀 s，引号内的字符串开始是 clr-namespace，用来指定命名空间。assembly 用来指定程序集名称。

2. 使用自定义类

有时需要自定义类，然后在 XAML 文件中使用，其中有如下两种情况。

(1) 使用本地程序的自定义类

例如，在本地程序中定义一个 Book 类，如代码 4-9 所示。

```
namespace mumu_customnamespace
{
    public class Book
    {
        public Book() { }
        public string Name { get; set; }
        public double Price { get; set; }
    }
}
```

代码 4-9 Book 类的实现

如果需要在 XAML 文件中使用该类，必须做如下声明：

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

由于 Book 类属于本地应用程序，因此声明中省略了设置 assembly，如代码 4-10 所示。

```
<Window x:Class="mumu_customnamespace.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:mumu_customnamespace"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <Button FontSize="14" >
            <local:Book Name="葵花宝典" Price="0.1"/>
        </Button>
    </Grid>
</Window>
```

代码 4-10 在 XAML 文件中使用 Book 对象

但是运行程序，按钮上面只会显示该类的类型，如图 4-4 所示。



图 4-4 在按钮上显示类的类型

显然这不是我们期待的效果，解决的办法是重载 `ToString` 函数，如代码 4-11 所示。

```
public override string ToString()
{
    string str = Name + "售价为：" + Price + "元";
    return str;
}
```

代码 4-11 重载 Book 类的 `ToString` 函数

再次运行程序，结果如图 4-5 所示。



图 4-5 运行结果

自定义类必须有一个默认的无参数的构造函数，比如下面的 `Book` 类，程序可能运行时会因无法实例化该类而报异常：

```
public class Book
{
    public Book(string name,double price) { }
    .....
}
```

但是编译器有时也可能会自动地添加一个无参数的构造函数，因此这样的问题算是一类不容易发现的 Bug。我们在自定义类时最好显式地添加无参数的构造函数以避免此类 Bug。

(2) 使用外部程序的自定义类

使用外部程序的自定义类需要设置 `assembly`，如新建一个类型为 `Class Library`，名为“`mumu_customlib`”的项目。然后将刚才的 `Book` 类添加到其中。

现在一个解决方案中有两个项目，其中 `mumu_customnamespace` 是一个本地应用程序；`mumu_customlib` 是一个外部程序集。首先在本地应用程序中添加对 `mumu_customlib` 的引用，然后在 `XAML` 文件中声明并使用 `Book` 类，如代码 4-12 所示。

```
<Window x:Class="mumu_customnamespace.Window1"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:customlib="clr-namespace:mumu_customlib;assembly=mumu_customlib"
Title="Window1" Height="300" Width="300">
<Grid>
    <Button FontSize="14">
        <customlib:Book Name="葵花宝典" Price="0.1"/>
    </Button>
</Grid>
</Window>
```

代码 4-12 引用外部程序集的 Book 对象

该示例的完整代码详见 `mumu_customnamespace` 工程。

4.3 简单属性和附加属性

4.3.1 简单属性

前面讨论的 `Title` 和 `Width` 属性均为简单属性，为其赋值时一定要将值放在双引号之内。XAML 解析器会根据属性的类型执行隐式转换，将字符串转换为相应的 CLR 对象。

其中枚举类型的属性设置会略有不同，在 C# 中设置一个枚举类型的值通常是“`类型.值`”。而在 XAML 中则省略前面的“`类型`”。在代码 4-13 中为 `SolidColorBrush` 的 `Color` 属性直接赋值，没有前面的类型 `Colors`。

```
C#
SolidColorBrush solidbrush = new SolidColorBrush();
solidbrush.Color = Colors.AliceBlue;

XAML
<Button Color="AliceBlue" />
```

代码 4-13 在 C# 和 XAML 中为 Color 赋值的不同方法

在 C# 中还有一种枚举类型的值可以通过或运算符 (`|`) 组合而成，如代码 4-14 所示。

```
public enum CarOptions
{
    SunRoof = 0x01,
    Spoiler = 0x02,
    FogLights = 0x04,
    TintedWindows = 0x08,
}

class FlagTest
{
    static void Main()
    {
        CarOptions options = CarOptions.SunRoof | CarOptions.FogLights;
    }
}
```

代码 4-14 C# 中为 flagwise 枚举类型的属性赋值

这类枚举值称为“flagwise 枚举值”，不能组合的枚举值称为“nonflag 枚举值”。在 WPF 中前者非常少见，代码 4-15 可能会用到该枚举值。Glyphs 对象可以用来描述文字，其 StyleSimulations 属性是一个 flagwise 枚举类型。在这里我们将字体设置为加粗和斜体，值中间用“，”取代了 C# 中的“|”。

```
<Glyphs  
    FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"  
    FontRenderingEmSize = "100"  
    StyleSimulations   = "BoldSimulation,ItalicSimulation"  
    UnicodeString     = "Hello XAML!"  
    Fill              = "Black"  
    OriginX           = "100"  
    OriginY           = "200"  
/>
```

代码 4-15 mumu_glyphs.xaml 文件

在 XamlPad 中的运行结果如图 4-6 所示。



图 4-6 运行结果

即使是该属性，WPF 中也提供了一个 BoldItalicSimulation 枚举值，从而省略了枚举值之间的组合。

4.3.2 附加属性

附加属性是可以用于多个控件，但是在另外一个类中定义的属性。在 WPF 中该类属性常用在布局中，如前面例子中的 DockPanel.Dock="Left"。附加属性的命名方式是“**定义类型.属性**”，这样可以让 XAML 解析器将其与普通属性区分开。附加属性的设置可以使用 Attribute 和 Property-Element 语法，使用后者时类型必须是包含该属性的类型，如代码 4-16 所示。

```
<Button >  
    <!--这里是 DockPanel，而不是 Button-->  
    <DockPanel.Dock>  
        Right  
    </DockPanel.Dock>  
        Hello XAML  
    </Button>
```

代码 4-16 附加属性的 Property-Element 语法

4.4 Content 属性

在前面的例子中可以看到 XAML 文件好像是一棵大的嵌套树，Window 包含 StackPanel、StackPanel 和 Button，其中 Content 属性起到了重要的作用。很多 WPF 类中都有这样一个特殊属性，即 Content（内容）属性，我们在前面已经看到 Button 的 Content 属性放置的是一个字符串。

代码 4-17 所示为 Content 的 Property-Element 语法。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
Width = "300" Height="200">  
    <Button.Content>  
        Hello XAML  
    </Button.Content>  
</Button>
```

代码 4-17 Content 的 Property-Element 语法

代码 4-18 所示为 Content 的 Attribute 语法。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
Width = "300" Height="200" Content="Hello XAML">  
</Button>
```

代码 4-18 Content 的 Attribute 语法

也可以省略掉 Content 标签，这样 XAML 会更加简洁，如代码 4-19 所示。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
Width = "300" Height="200" >  
    Hello XAML  
</Button>
```

代码 4-19 Content 的常见写法

Content 属性值可以放在 Button 元素中间的任何位置，如代码 4-20 中的“Hello XAML”放置在 Button 元素的末尾处，同样也可以放置在标记 1 和 2 处。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" >  
    <!-1 Hello XAML-->  
    <Button.Width>  
        300  
    </Button.Width>  
    <!-2 Hello XAML-->  
    <Button.Height>  
        200  
    </Button.Height>  
    Hello XAML  
</Button>
```

代码 4-20 Content 属性的值可以放置在 Button 元素中间的任何位置上

但是 Content 不能分开放置，代码 4-21 所示的写法是错误的，不允许在 Content 的值中插入其他标签。

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" >  
    <Button.Width>
```

```
    300
  </Button.Width>
XAML
  <Button.Height>
    200
  </Button.Height>
  Hello
</Button>
```

代码 4-21 Content 属性设值的错误写法

例外情况是 TextBlock 的 Content 属性中可以放置一些加粗或者斜体标签，如代码 4-22 所示。

```
<TextBlock xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" >
  <Italic>Hi everyOne,<Italic>This is <Bold>XAML</Bold>
</TextBlock >
```

代码 4-22 mumu_textblock.xaml 文件

Content 属性不需要使用 Property-Element 语法即可直接成为 Button 元素的子类，这是因为当用户定义一个类时使用 ContentProperty 来说明某个属性是 Content 属性，这样 WPF 就知道该属性可以不需要使用 Property-Element 语法。如 Button 类，通过 Reflector 可以查看到其基类 ContentControl 的标识，如代码 4-23 所示。

```
[DefaultProperty("Content"), ContentProperty("Content"),
Localizability(LocalizationCategory.None, Readability=Readability.Unreadable)]
public class ContentControl : Control, IAddChild
....
```

代码 4-23 ContentControl 的标识

同样可以为前面的 Book 类添加一个 Text 属性，并且将其指定为 Content 属性。然后在 ToString 函数中添加 Text 的内容，如代码 4-24 所示。

```
[ContentProperty("Text")]
public class Book
{
  public string Text { get; set; }
  public override string ToString()
  {
    string str = Name + "售价为：" + Price + "元" + "\n" + Text;
    return str;
  }
  ...
}
```

代码 4-24 在 ToString 函数中添加 Text 的内容

在 XAML 文件中也可以按照内容属性语法来设置 Book 的 Text 属性值，如：

```
<Button FontSize="14">
  <customlib:Book Name="葵花宝典" Price="0.1">
    欲练此功 必先自宫
  </customlib:Book>
</Button>
```

WPF 当中到底有哪些类有 Content 属性，它们的 Content 属性的名字又是什么？本章同样提供了一

个小工具可以查看不同类的 Content 属性（详见示例代码中 `mumu_dumpcontentproperty` 工程），如图 4-7 所示。

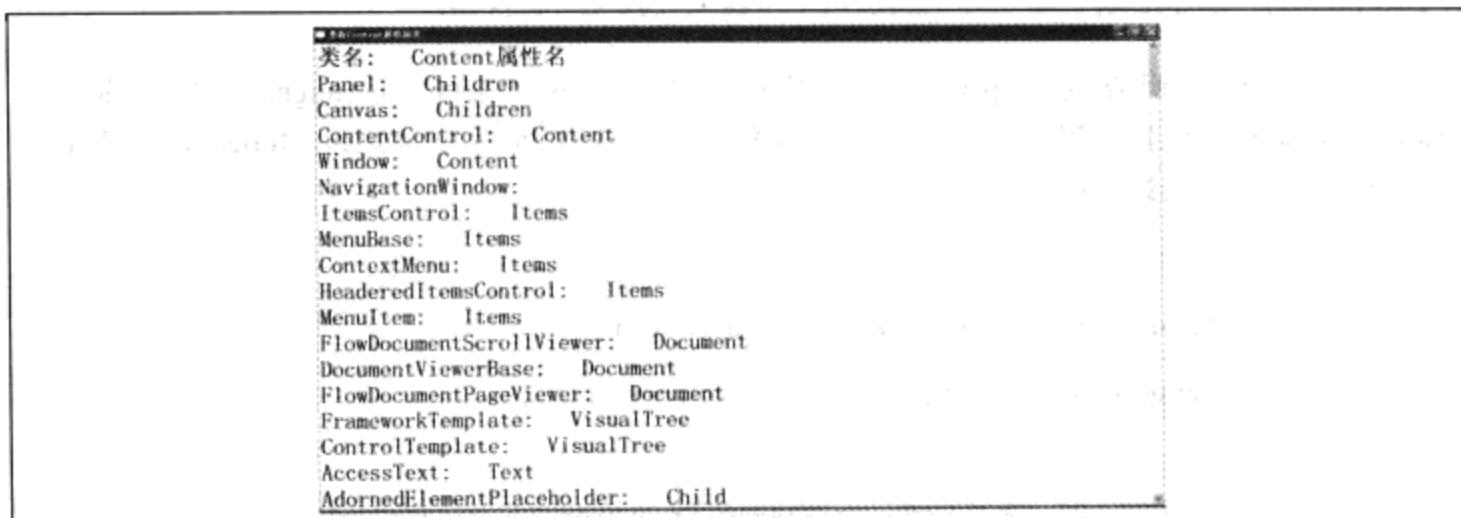


图 4-7 查看不同类的 Content 属性

上图中可以看出不仅仅是名称为 Content 属性才是真正 Content 属性，而 Panel 的 Children 及 MenuBase 的 Items 属性也是 Content 属性。这两种属性和 Content 属性不同的是它们是一个集合属性，可以放置一个或多个对象。

4.5 类型转换器

4.5.1 功能

XAML 解析器通过类型转换器跨越字符串值和非字符串值的鸿沟，在 XAML 中输入的字符串通过类型转换器将这些字符串转换为相应的 CLR 对象。这就是类型转化器所起到的作用，如图 4-8 所示。

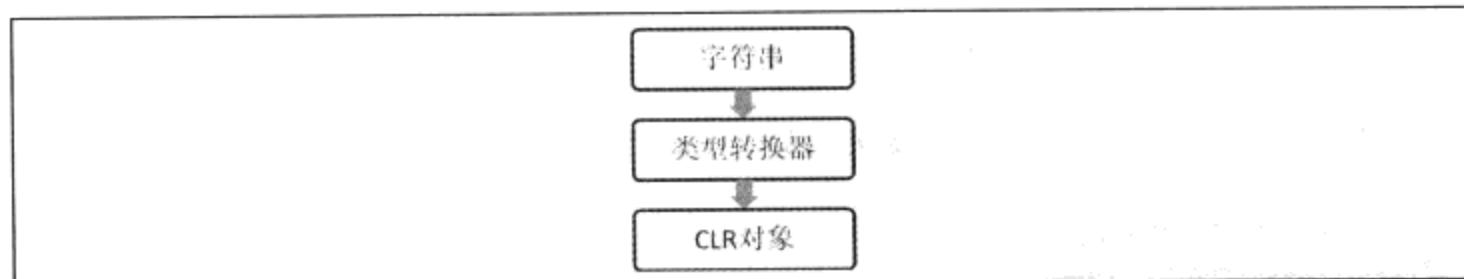


图 4-8 类型转换器

所有的类型转换器都派生自 `TypeConverter`。`TypeConverter` 提供的 4 个重要的方法是 `CanConvertTo`、`CanConvertFrom`、`ConvertTo` 和 `ConvertFrom`。

`ConvertFrom` 方法将 XAML 中的字符串转换为相应的 CLR 对象；`ConvertTo` 方法将 CLR 对象转换为相应的字符串；`CanConvertFrom` 用来检查能否从字符串转换为相应的 CLR 对象；`CanConvertTo` 检查 CLR 对象能否转换为相应的字符串，如果可以，则返回 `true`，否则返回 `false`。

从 TypeConverter 派生的转换器类有 100 多个，我们不可能在这儿一一列举。比较常用的比如 BrushConverter 就可以实现从字符串转换成相应的画刷类型。但是 XAML 是如何知道何种属性使用什么转换器呢？XAML 解析器实际上通过两个步骤来查找类型转换器：

(1) 检查属性声明查找 TypeConverter 特性，如窗口的 Width 和 Height 属性（派生自 FrameworkElement）前面都会声明一个类型转换器。代码 4-25 所示为通过 Reflector 查看到的 FrameworkElement 的源码片段。

```
public class FrameworkElement.....  
{  
    [.....TypeConverter(typeof(LengthConverter))]  
    public double Height { get; set; }  
    [.....TypeConverter(typeof(LengthConverter))]  
    public double Width { get; set; }  
}
```

代码 4-25 通过 Reflector 查找到的 FrameworkElement 源码片段

(2) 如果在属性声明中没有 TypeConverter 特性，XAML 解析器会检查对应数据类型的类的声明。如按钮的 Background 属性类型是 Brush，并在 Brush 类头部就声明了一个 BrushConverter 转换器，这样在设置 Background 属性时 XAML 解析器会自动应用该转换器。代码 4-26 所示为通过 Reflector 查看到的 Brush 的源码片段。

```
[.....TypeConverter(typeof(BrushConverter)).....]  
public abstract class Brush : Animatable, .....
```

代码 4-26 Reflector 查看到的 Brush 的源码片段

4.5.2 自定义类型转换器

我们还是以自定义的类 Book，说明如何使 Book 的 Price 属性稍作改进能够支持人民币，还能支持美元。如：

```
<local:Book Name="葵花宝典" Price="0.1"/>
```

我们认为葵花宝典的价格还是 0.1 元，如果写成：

```
<local:Book Name="葵花宝典" Price="$0.1"/>
```

则认为葵花宝典的价格是 0.8 元（假定 1 美元=8 元）。

为 Price 定义一个新的类型为 MoneyType，它只是简单地封装了一个 double 类型的变量。同时提供了一个静态的 Parse 方法将一个字符串正确转换为 MoneyType 类型的对象，如代码 4-27 所示。

```
[TypeConverter(typeof(MoneyConverter))]  
public class MoneyType  
{  
    private double _value;  
    public MoneyType() { _value=0; }  
    public MoneyType(double value) { _value = value; }
```

```

public override string ToString()
{
    return _value.ToString();
}
public static MoneyType Parse(string value)
{
    string str = (value as string).Trim();
    if (str[0] == '$')
    {
        string newprice = str.Remove(0, 1);
        double price = double.Parse(newprice);
        return new MoneyType(price * 8);
    }
    else
    {
        double price = double.Parse(str);
        return new MoneyType(price);
    }
}

```

代码 4-27 mumu_customlib 项目的 CustomClass.cs 文件

在第 1 行中为这个类提供了一个 MoneyConverter 类型的转换器。下面是其实现。为了使这个类看起来比较完整，将实现其 4 个方法。事实上关键的是 ConvertFrom 方法，通过 MoneyType 的静态方法 Parse 将字符串转换为正确的 MoneyType 对象，如代码 4-28 所示。

```

public class MoneyConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context,
                                         Type sourceType)
    {
        if (sourceType == typeof(string))
            return true;
        return base.CanConvertFrom(context, sourceType);
    }

    public override bool CanConvertTo(ITypeDescriptorContext context,
                                     Type destinationType)
    {
        if (destinationType == typeof(string))
            return true;
        return base.CanConvertTo(context, destinationType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
                                       CultureInfo culture,
                                       object value)
    {
        if (value.GetType() != typeof(string))
            return base.ConvertFrom(context, culture, value);
        return MoneyType.Parse((string)value);
    }

    public override object ConvertTo(ITypeDescriptorContext context,
                                    CultureInfo culture,
                                    object value,
                                    Type destinationType)
    {

```

```
        if (destinationType == typeof(string))
            return base.ConvertTo(context, culture, value, destinationType);
        return value.ToString();
    }
}
```

代码 4-28 MoneyConverter 的实现

我们在原来的 XAML 文件中 Price 的值前面加上一个“\$”符号，再次运行程序，结果如图 4-9 所示。



图 4-9 运行结果

4.6 标记扩展

现在绝大多数的属性 XAML 都可以工作得很好了，但是依旧会有几种情况 XAML 难以胜任。

- (1) 将一个属性赋值为 null。
- (2) 将一个属性赋值给一个静态变量，如将按钮的 Background 赋值为一个预定义画刷，参看代码 4-29 所示。

```
Button btn = new Button();
btn.Content = "Hello XAML";
btn.Background = SystemColors.ActiveCaptionBrush;
.....
```

代码 4-29 在 C# 中将 Background 属性设置为一个静态变量

以上这几种情况，我们就需要使用标记扩展了。类型转换器悄无声息地实现了类型转换，而标记扩展则是通过 XAML 的显式的、一致的语法调用实现。标记扩展比类型转换器更为强大，和类型转换器一样，标记扩展也可以通过自定义来实现 XAML 的语义扩展。

在 XAML 中只要属性值被一对花括号 {} 括起，XAML 解析器就会认为这是一个标记扩展，而不是一个普通的字符串。前面的两种情况都可以用标记扩展的方法来解决，将一个按钮的 Background 属性赋值为空，表示这个按钮的背景颜色为透明色。{x:Null} 是 XAML 中提供的一种标记扩展，表示一个空值，如代码 4-30 所示。

```
<Button Name="btn" Content="MyButton" Click="btn_Click" Background="{x:Null}">
```

代码 4-30 在 XAML 里将 Background 属性设置为空值

XAML 中提供了一个{x:Static}的标记扩展，可以引用一个类的静态变量，如代码 4-31 所示。

```
<Button Name="btn" Content="MyButton" Click="btn_Click">
    <Button.Background>
        <x:Static Member="SystemColors.ActiveCaptionBrush"/>
    </Button.Background>
</Button>
```

代码 4-31 {x:Static}标记扩展

有时候，我们只是想显示一个字符串，但是不巧字符串里就有一对花括号（{}），如代码 4-32 所示，这个时候 XAML 解析器会固执地认为这是一个标记扩展。由于它又无法找到 HelloXAML 这样的标记扩展，所以无法编译通过。

```
<TextBlock Text="{HelloXAML}" />
```

代码 4-32 无法编译通过的例子

解决这个问题的方法是在该字符串前面添加一个空的花括号，以指示 XAML 它只是一个普通的字符串，而不是一个标记扩展，如代码 4-33 所示。

```
<TextBlock Text="{}{HelloXAML}" />
```

代码 4-33 {}语法

4.7 分别使用 XAML 和 C# 构建应用程序——刀还是刀，剑还是剑

4.7.1 XAML——反两仪刀法

XAML 语言，我们称之为标记式（Markup）语言。它的这种层次结构天生地适合描述 WPF 界面。如果有过桌面用户界面编程的经验，无论是 MFC，还是 WinForm。Visual Studio 都会提供一个设计工具，程序员通过设计工具设计界面，然后生成程序代码，这些程序代码为了设计工具能够管理和编辑通常要严格地遵循某种格式。因此程序员最好不要轻易地动这些程序代码。而 XAML 却不一样了，如果说以前的描述界面的代码适合机器阅读，而不适合人类阅读。XAML 则是既适合机器阅读，也适合人类阅读。

XAML 还有一个其他代码语言（C#或者 VB）难以比拟的优点，即甚至不需要编译，可以在 IE 或者在 XamlPad 中直接浏览。这种文件称为“松散 XAML 文件”，可以容易地将其从桌面应用移植到 Web 应用。

尽管我们一直在 WPF 的范畴里讨论 XAML，但是不要误以为 XAML 是 WPF 的专用品。事实上，XAML 和 WPF 并不相互依赖，WPF 完全可以只用代码语言来实现应用程序。XAML 虽然最初为 WPF 设计，但也可以应用于如下方面。

- (1) WPF XAML：本书所讨论的 XAML。

(2) XPS XAML: WPF XAML 的一部分, 它定义了一个表示格式化的电子文档的 XML。

(3) Silverlight XAML: Silverlight 是一个跨平台的浏览器插件, 它是 WPF 的一个子集。可以使用 XAML 来描述 Silverlight 的二维图形、动画, 以及音频和视频以创建富互联网应用。

(4) WF XAML: 描述 Windows 工作流基础内容的元素, 可以参阅相关书籍, 也可以访问 <http://wf.netfx3.com> 了解更多内容。

XAML 毕竟是种相对简单的声明式语言, 不适用于应用程序的业务逻辑实现。代码 4-34 所示为一个仅仅依靠 XAML 实现的应用程序, 该项目由 App.xaml 和 Window1.xaml 文件组成。

```
App.xaml
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="Window1.xaml">
</Application>

Window1.xaml
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="AppByOnlyXAML" Height="200" Width="600" >
    <StackPanel Width="500">
        <Button Content="MyButton"/>
        <Ellipse Stroke="Red" Height="60" StrokeThickness="3" />
    </StackPanel>
</Window>
```

代码 4-34 只用 XAML 构建应用程序 (完整示例见 mumu_appbyonlyxaml 工程)

应用程序的入口是 App.xaml 文件, XAML 文件的根元素之一即 Application。该文件的 BuildAction 必须设置为 ApplicationDefinition (右击 App.xaml 文件, 选择快捷菜单中的 Properties 选项, 然后在打开的 Properties 对话框中设置)。Application 元素的 StartupUri 属性设置需要启动的窗口文件, 在这里是 Window1.xaml 文件, 如图 4-10 所示。

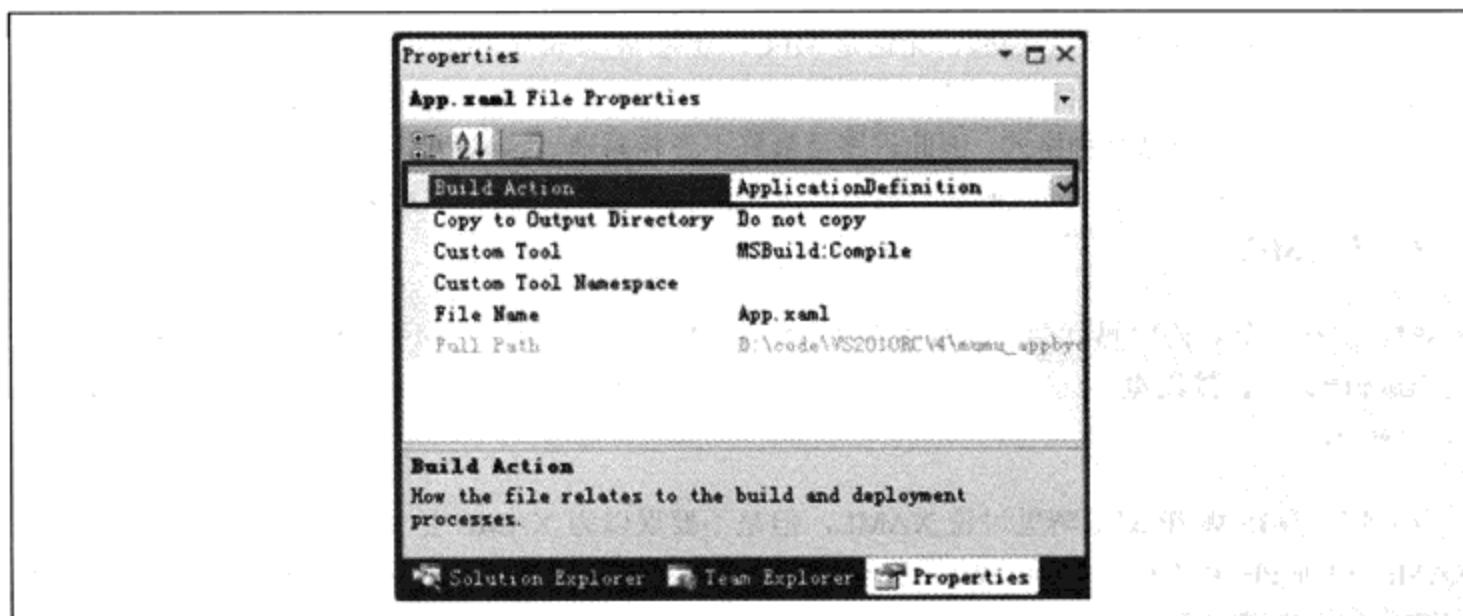


图 4-10 设置 App.xaml 文件的 Build Action 属性

Window1.xaml 文件的 Build Action 选项设置为 Page，该窗口中包含一个面板 StackPanel。其中放置一个按钮和一个椭圆形，运行结果如图 4-11 所示。

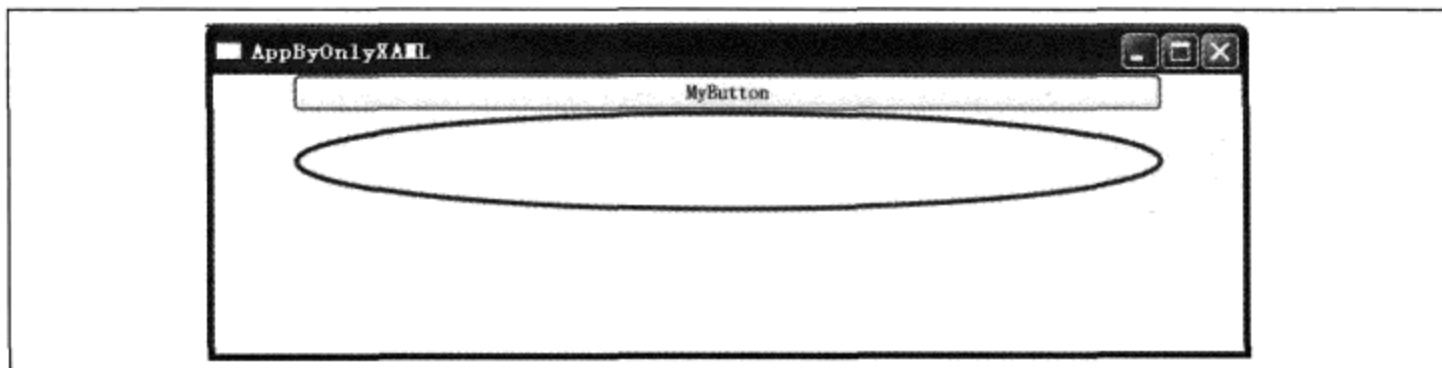


图 4-11 运行结果

如果单击按钮令椭圆的边界色由红色变成蓝色，则难以处理。XAML 毕竟是种简单的声明式的语言，对于这样的应用程序业务逻辑显得力不从心，因此需要更为强大 C# 语言——正两仪剑法。

4.7.2 C#——正两仪剑法

C# 强大到任何 XAML 写的应用程序都可以轻易地通过 C# 实现，我们通过代码方式将这个 StackPanel 作为一个窗口的 Content 属性显示。如代码 4-35 所示，函数 InitializeComponent 主要用来描述界面设置。虽然有的时候我们会抱怨 XAML 过于冗长，但是在描述界面方面，它确实比用代码的方式要简洁和清晰很多。

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

namespace mumu_appbyonlycode
{
    public class WindowByOnlyCode : Window
    {
        public WindowByOnlyCode()
        {
            InitializeComponent();
        }
        private Ellipse elip;
        public void InitializeComponent()
        {
            this.Width = 600;
            this.Height = 200;
            this.Title = "AppByOnlyCode";

            StackPanel panel = new StackPanel();
            panel.Width = 500;

            Button btn = new Button();
            btn.Content = "MyButton";
            panel.Children.Add(btn);

            elip = new Ellipse();
        }
    }
}
```

```

        elip.Stroke = new SolidColorBrush(Colors.Red);
        elip.Height = 60;
        elip.StrokeThickness = 3;
        panel.Children.Add(elip);

        this.Content = panel;
    }

    [STAThread]
    public static void Main()
    {
        Application app = new Application();
        app.Run(new WindowByOnlyCode());
    }
}

```

代码 4-35 完整示例见 `mumu_appbyonlycode` 工程

程序运行结果同上节，只是修改了窗口的标题，如图 4-12 所示。

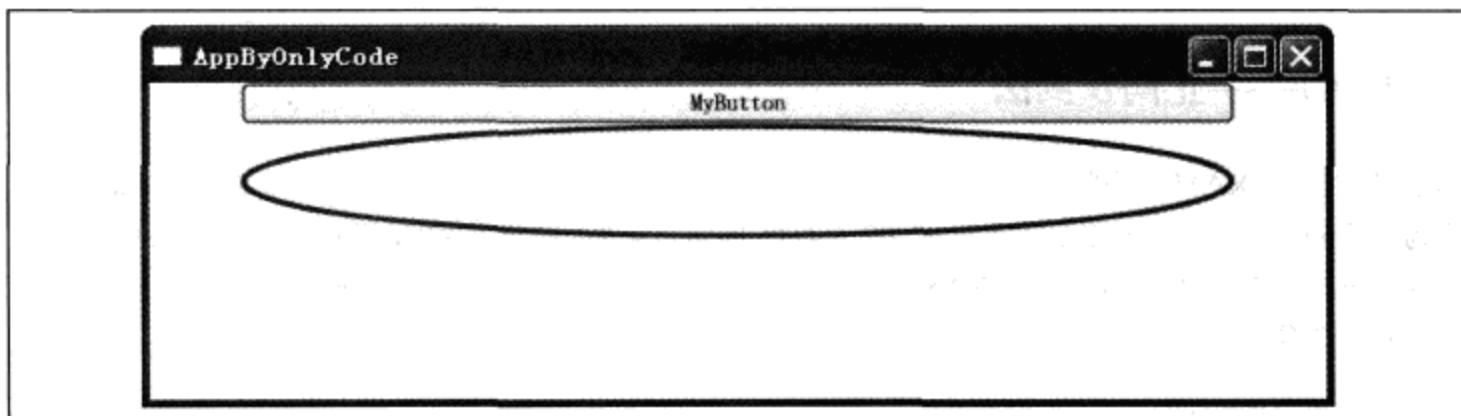


图 4-12 运行结果

现在可以方便地实现上一节的业务逻辑，在 `InitializeComponent` 函数中为 `Button` 的 `Click` 事件注册事件处理函数 `btn_Click`，然后在其中将椭圆的颜色修改成蓝色，如代码 4-36 所示。

```

public void InitializeComponent()
{
    .....
    btn.Click += new RoutedEventHandler(btn_Click);
}

void btn_Click(object sender, RoutedEventArgs e)
{
    elip.Stroke = new SolidColorBrush(Colors.Blue);
}

```

代码 4-36 在 C# 中添加按钮 Click 事件处理函数

4.8 使用 XAML 和 C# 构建应用程序——刀剑合璧

既然 XAML 在描述界面上比 C# 更为简洁，那么很自然地可以想到将描述界面的任务交给 XAML，而将实现业务逻辑的交给 C#。

4.8.1 第1次刀剑合璧

一个最为直接的想法是编写一个程序，解析前例中 XAML 的字符串，通过反射机制最终将这些字符串变成一个对象。这样的想法虽然直接，但是会是一种失败的“合璧”，会呈几何倍数地增大工作量。如果 WPF 提供了这样的解析类，则情况将不同。

WPF 中提供了这样的解析类，在 System.Windows.Markup 命名空间中包含一个 XamlReader 类，它通过一个静态的 Load 方法将 XAML 字符串解析为一个对应的对象。但是该方法不能直接接受字符串作为参数，而且需要一个 XmlReader 对象作为输入参数，可以通过 StringReader 作为中介将字符串转换为一个 XmlTextReader 对象（XmlTextReader 派生自 XmlReader）。我们按照这样的思路，在前面的例子中第 1 次混合使用 XAML 和 C# 来构建应用程序，如代码 4-37 所示。

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Shapes;
using System.Windows.Media;
using System.Windows.Markup;
using System.Xml;
using System.IO;
namespace mumu_appbyxamlcode1
{
    public class WindowByXAMLAndCode : Window
    {
        public WindowByXAMLAndCode()
        {
            InitializeComponent();
        }
        public void InitializeComponent()
        {
            this.Width = 600;
            this.Height = 200;
            this.Title = "AppByXAMLAndCode1";

            string strXaml = "<StackPanel xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation' Width='500'>" +
                "<Button Content='MyButton' />" +
                "<Ellipse Stroke='Red' Height='60' StrokeThickness='3' />" +
                "</StackPanel>";
            StringReader strReader = new StringReader(strXaml);
            XmlTextReader xmlreader = new XmlTextReader(strReader);
            StackPanel obj = (StackPanel)XamlReader.Load(xmlreader);
            Content = obj;
        }
    }
}
```

代码 4-37 完整示例见 mumu_appbyxamlcode1 工程

由于涉及 `StringReader`、`XmlTextReader` 和 `XamlReader` 几个新类，因此需要增加一个 `System.Xml.dll` 的引用，同时需要增加 `System.Windows.Markup`、`System.Xml` 和 `System.IO` 命名空间。除了改变窗口的标题，程序运行结果和前面一节相同。

当然也可以不从字符串，而直接从一个松散 XAML 文件解析对象，如将前面的字符串写成一个松散 XAML 文件（`mumu_stackpanel.xaml`）后添加到项目中。注意 `mumu_stackpanel.xaml` 文件的 Build Action 选项一定要设置为 Resource，如代码 4-38 所示。

```
public void InitializeComponent()
{
    this.Width = 600;
    this.Height = 200;
    this.Title = "AppByXAMLAndCode1";
    Uri uri = new Uri("pack://application:,,,/mumu_stackpanel.xaml");
    Stream stream = Application.GetResourceStream(uri).Stream;
    StackPanel obj = (StackPanel)XamlReader.Load(stream);
    Content = obj;
}
```

代码 4-38 使用 `XamlReader` 加载文件

如果为 `Button` 添加事件，则需要为 `mumu_stackpanel.xaml` 添加 `Name` 属性，如代码 4-39 所示。

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            Width="500">
    <Button Name="btn" Content="MyButton"/>
    <Ellipse Name="elip" Stroke="Red" Height="60" StrokeThickness="3" />
</StackPanel>
```

代码 4-39 `mumu_stackpanel.xaml` 文件

在 `InitializeComponent` 函数中通过 `FindName` 方法找到按钮和椭圆对象，将椭圆对象保存在事先定义的成员变量中，然后为按钮对象注册 `Click` 事件处理函数。在该函数中通过保存的椭圆对象来改变边界颜色，如代码 4-40 所示。

```
public class WindowByXAMLAndCode :Window
{
    private Ellipse elip;
    public void InitializeComponent()
    {
        .....
        StackPanel obj = (StackPanel)XamlReader.Load(stream);
        Content = obj;
        elip = obj.FindName("elip") as Ellipse;
        Button btn = obj.FindName("btn") as Button;
        btn.Click+=new RoutedEventHandler(btn_Click);
    }
    void btn_Click(object sender, RoutedEventArgs e)
    {
        if(elip != null)
            elip.Stroke = new SolidColorBrush(Colors.Blue);
    }
    .....
}
```

代码 4-40 为按钮对象注册 `Click` 事件处理函数

4.8.2 完美的刀剑合璧

1. Markup+Code-Behind

WPF 提供了完美的刀剑合璧，看同样的例子（完整示例见 `mumu_appbyxamlcode2` 工程）。WPF 的解决方案提供了 4 个文件（`App.xaml` 和 `App.xaml.cs`, `MainWindow.xaml` 和 `MainWindow.xaml.cs`），两两成对，如图 4-13 所示。

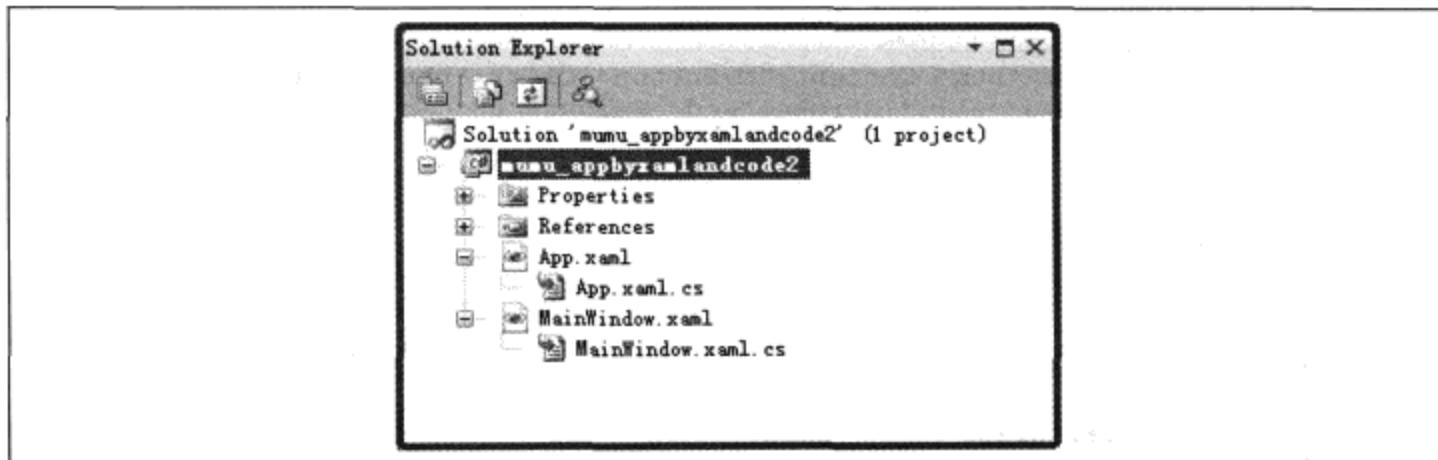


图 4-13 `mumu_appbyxamlcode2` 工程文件组成

`App.xaml` 为应用程序的入口；`App.xaml.cs` 是其后台代码，用于实现业务逻辑，二者通过 `x:Class` 关联。在 XAML 文件中 `x:Class` 指定的类名必须和对应的.cs 文件中的类名一致，在.cs 文件中类的声明必须加上关键字 `partial`。

`MainWindow.xaml` 为应用程序的主窗口，其中描述该窗口的界面组成。而对应的.cs 文件实现的是事件响应处理函数，二者之间的关联和前面类似，但是注意必须在 `MainWindow` 的构造函数中调用 `InitializeComponent` 方法。

同样需要为按钮添加 `Click` 事件处理函数，在 `MainWindow.xaml` 文件的按钮标签中加上 `Click="btn_Click"`。该函数的实现放在 `MainWindow.xaml.cs` 文件中，这就是 WPF 中典型的 `Markup+Code-behind` 的做法。

4 个文件的内容分别如代码 4-41 所示。

```
App.xaml
<Application x:Class="mumu_appbyxamlcode2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>

App.xaml.cs
namespace mumu_appbyxamlcode2
{
    /// <summary>
```

```

/// Interaction logic for App.xaml
/// </summary>
public partial class App : Application
{
}

MainWindow.xaml
<Window x:Class="mumu_appbyxamlcode2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="AppByXAMLAndCode2" Height="200" Width="600">
    <StackPanel Width="500">
        <Button Name="btn" Content="MyButton" Click="btn_Click" />
        <Ellipse Name="elip" Stroke="Red" Height="60" StrokeThickness="3" />
    </StackPanel>
</Window>

MainWindow.xaml.cs
namespace mumu_appbyxamlcode2
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void btn_Click(object sender, RoutedEventArgs e)
        {
            if (elip != null)
                elip.Stroke = new SolidColorBrush(Colors.Blue);
        }
    }
}

```

代码 4-41 mumu_appbyxamlcode2 工程的完整代码

2. 工作原理

我们看看 WPF 是如何将 XAML 文件和代码文件关联起来的。工程目录下 obj 子目录中的 Release 或者 Debug 子目录（取决于编译状态是 Debug 还是 Release）中后缀名为.g.cs 的两个文件是 App 和 MainWindow 类的另外一部分。g(generated) 指这些文件是自动产生的，两个文件的内容如代码 4-42 所示。

```

App.g.cs
namespace mumu_appbyxamlcode2 {

    /// <summary>
    /// App
    /// </summary>
    public partial class App : System.Windows.Application {

        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCode()]

```

```

        public void InitializeComponent() {
            #line 4 "..\..\App.xaml"
            this.StartupUri = new System.Uri("MainWindow.xaml",
                System.UriKind.Relative);
            #line default
            #line hidden
        }

        /// <summary>
        /// Application Entry Point.
        /// </summary>
        [System.STAThreadAttribute()]
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public static void Main() {
            mumu_appbyxamlcode2.App app = new
            mumu_appbyxamlcode2.App();
            app.InitializeComponent();
            app.Run();
        }
    }
}

```

代码 4-42 mumu_appbyxamlcode2 工程中的 App.g.cs 文件

我们可以看到代码①处才是真正的函数入口，App 类仍从 Main 函数开始，如代码 4-43 所示。

```

MainWindow.g.cs
namespace mumu_appbyxamlcode2 {

    /// <summary>
    /// MainWindow
    /// </summary>
    public partial class MainWindow : System.Windows.Window,
        System.Windows.Markup.IComponentConnector {

        #line 6 "..\..\MainWindow.xaml"
        ② internal System.Windows.Controls.Button btn;

        #line default
        #line hidden

        #line 7 "..\..\MainWindow.xaml"
        ③ internal System.Windows.Shapes.Ellipse elip;

        #line default
        #line hidden

        ④ private bool _contentLoaded;

        /// <summary>
        /// InitializeComponent
        /// </summary>
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        ⑤ public void InitializeComponent() {
            if (_contentLoaded) {
                return;
            }
            _contentLoaded = true;
            System.Uri resourceLocater = new System.Uri("/mumu_

```

```

    appbyxamlcode2;component/mainwindow.xaml",
    System.UriKind.Relative);

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

        #line default
        #line hidden
    }

    [System.Diagnostics.DebuggerNonUserCode()]

```

⑥ [System.ComponentModel.EditorBrowsableAttribute(System.ComponentModel.EditorBrowsableState.Never)]

```

[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Design", "CA1033:InterfaceMethodsShouldBeCallableByChildTypes")]
    void System.Windows.Markup.IComponentConnector.Connect(int
connectionId, object target) {
        switch (connectionId)
        {
        case 1:
            this.btn = ((System.Windows.Controls.Button)(target));

            #line 6 "..\..\MainWindow.xaml"
            this.btn.Click += new System.Windows.
RoutedEventHandler(this.btn_Click);

            #line default
            #line hidden
            return;
        case 2:
            this.elip = ((System.Windows.Shapes.Ellipse)(target));
            return;
        }
        this._contentLoaded = true;
    }
}

```

代码 4-43 MainWindow.g.cs 文件

在代码②和代码③处会发现 `btn` 和 `elip` 的字段声明，XAML 的名称和 XAML 文件中按钮和椭圆的 `Name` 属性一致。`btn` 和 `elip` 会在 `Connect` 方法中（⑥）获得该窗口的按钮和椭圆对象的实例，并实现事件处理函数的注册。`Connect` 方法是 `IComponentConnector` 必须实现的一个接口方法，在 `MainWindow` 的 `InitializeComponent` 函数中（⑤）会根据初始的 xaml 文件（这里是 `MainWindow.xaml`），将其转换为当前的应用对象。该函数调用 `LoadComponent` 后将一个 `bool` 类型的私有变量设置为 `true`，以保证在程序的生命周期内请求的资源只加载一次。加载的资源是一个 BAML 文件（这里是 `MainWindow.baml`），它和生成的 `.g.cs` 文件在同一个目录下。

BAML 文件是一个二进制形式的 XAML 文件，它会作为一个二进制资源嵌入到程序集中，可以通过 Reflector 工具查到，如图 4-14 所示。该文件比原来的 XAML 文件小得多，这样会显著提高程序运行时的性能。

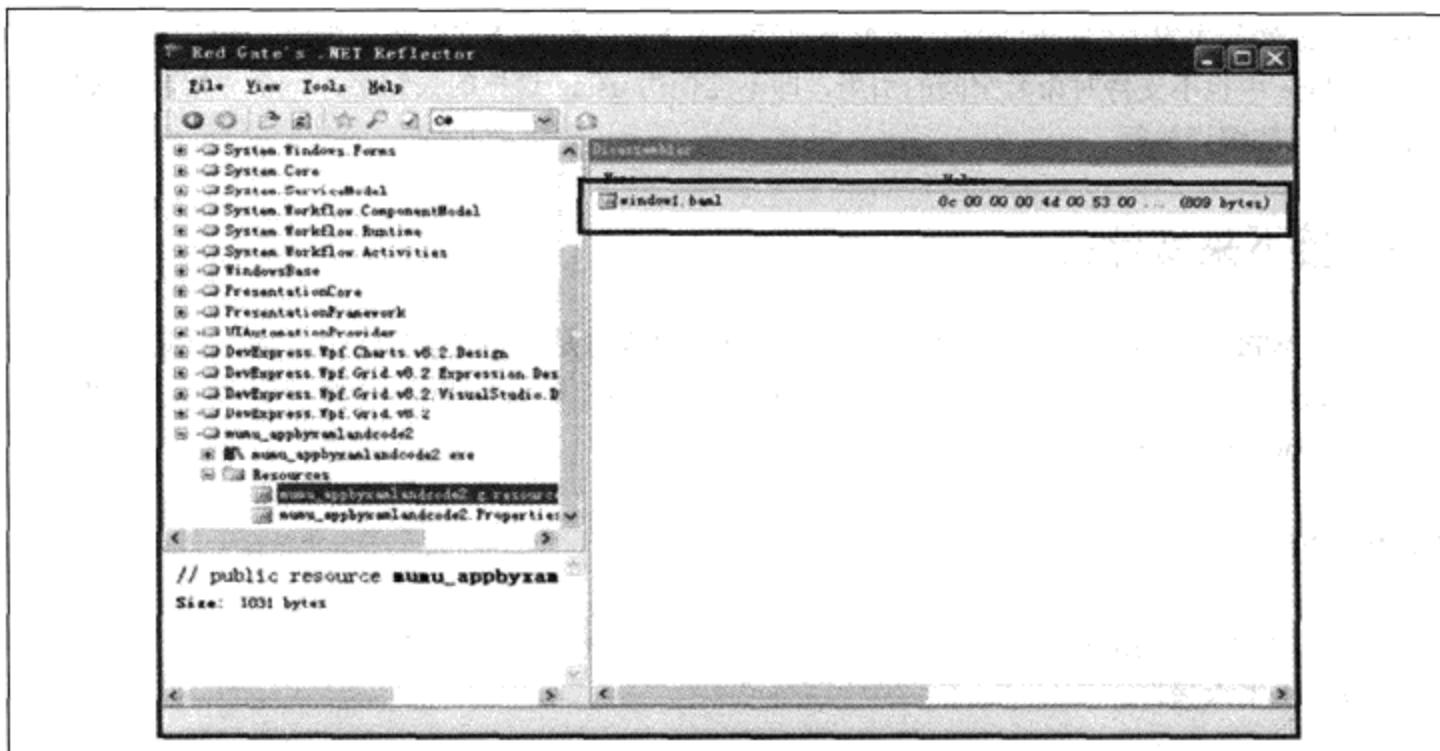


图 4-14 BAML 文件嵌入在程序集中

4.8.3 还有一种方法——在 XAML 中嵌入代码

其实 XAML 和 C#还有一种混合使用的方法，就是在 XAML 中嵌入 C#代码，如代码 4-44 所示。

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="MainWindow"
    Title="AppByOnlyXAML" Height="200" Width="600" >
    <StackPanel Width="500">
        <Button Name="btn" Content="MyButton" Click="btn_Click" >
        </Button>
        <Ellipse Name="elip" Stroke="Red" Height="60" StrokeThickness="3" />
        <x:Code>
            <![CDATA[
                private void btn_Click(object sender, RoutedEventArgs e)
                {
                    if (elip != null)
                        elip.Stroke = new SolidColorBrush(Colors.Blue);
                }
            ]]>
        </x:Code>
    </StackPanel>
</Window>
```

代码 4-44 在 XAML 中嵌入 C#代码

嵌入的程序代码需要使用 x:Code 元素及 x:Code 中的 CDATA (Character data, 字符数据) 节，它实际是 XML 中的规定。即必须以“<![CDATA[”开头，以“]]>”结尾。如果中间出现“]]>”这样的字符，则会出问题，如代码 array1[array2[i]]>5。这样的情况虽然极为罕见，但是一定要小心。如果遇到，则需要在“>”号中多加一个空格。

这样的方式看起来很方便，但是实际上很不灵活。为了嵌入代码该文件必须要使用 `x:Class` 关键字。此外 `x:Code` 中也不支持对命名空间的引用，即不能使用 `using` 这样的关键字，一个“松散” XAML 文件变得只能编译执行。

4.9 接下来做什么

XAML 在 WPF 中主要是一种描述界面的语言。学习语言总是需要先从理论上了解，再到实践中体会。心法这一卷整个来说都是理论多于实践。接下来将介绍 WPF 中的一个重要概念——依赖属性（Dependency property）。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 倚天屠龙记》，“第二十一章 排难解纷当六强”。
- [2] MSDN Library for Visual Studio 2008 SP1 XAML Overview。
- [3] MSDN Library for Visual Studio 2008 SP1 XAML Syntax Terminology。

依赖属性——木木的“汗血宝马”

郭靖也是一股子的倔强脾气，被那小红马累得满身大汗。忽地右臂伸入马颈底下，双臂环抱，运起劲来。他内力一到臂上，越收越紧。小红马翻腾跳跃，摆脱不开。到后来呼气不得，窒息难当，这才知道了真主，忽地立定不动。韩宝驹喜道：“成啦，成啦！”郭靖怕那马逃去，还不敢跳下马背。韩宝驹道：“下来吧。这马跟定了你，你赶也赶不走啦。”郭靖依言跃下，那小红马伸出舌头来舐他的手背，神态十分亲热，众人看得都笑了起来。

——《射雕英雄传》，“第五回 弯弓射雕”^[1]

这是郭靖驯服“小红马”的一段，小红马是神马，着实很难驯化，需要郭靖那样的倔强脾气。“依赖属性”倒真是有几分像“小红马”，如果不理解，确实让它往东（希望属性值为 a），它偏偏往西（最后获得值为 b），最后还闹得个“人仰马不翻”的结局。

依赖属性是 WPF 平台的基础概念，也是一个全新的概念，整个 WPF 平台上都会用到它。

本章内容如下：

- (1) 先说“属性”，再说“依赖”。
- (2) 认识依赖属性。
- (3) 自定义一个依赖属性。
- (4) 所有规则大排队。
- (5) 附加属性和“等餐号”。
- (6) 接下来做什么。

5.1 属性与依赖

依赖属性是 WPF 引入的一个新的属性类型。

过去的属性很简单，如一个文件名、一个用户名和一个窗口的标题等都是属性，并且作为一个类的成员变量存储在类中。如代码 5-1 (mumu_CLRProperty 工程) 所示，其中_filename 是 mumu_File 的属性。

```
mumu_File.cs  
class mumu_File
```

```
    {
        private string _filename;
        public string Filename
        {
            get
            {
                return _filename;
            }
            set
            {
                if (_filename != value)
                    _filename = value;
            }
        }
    }
```

代码 5-1 mumu_File 类的_filename 属性

在 Main 函数中访问_filename 属性如代码 5-2 所示。

```
Program.cs
class Program
{
    static void Main(string[] args)
    {
        mumu_File file = new mumu_File();
        file.Filename = "mumu_File.txt";
        string s = file.Filename;
    }
}
```

代码 5-2 在 Main 函数中访问_filename 属性

代码 5-3 所示为 Button 中的一个 IsDefault 属性，注意它是一个依赖属性。

```
public Button() : ButtonBase
{
    // 依赖属性
    public static readonly DependencyProperty IsDefaultProperty;
    static Button()
    {
        // 注册属性
        Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
typeof(bool), typeof(Button),
new FrameworkPropertyMetadata(false, new
PropertyChangedCallback(OnIsDefaultChanged)));
    }

    // .NET 属性包装器
    public bool IsDefault
    {
        get { return (bool)GetValue(Button.IsDefaultProperty); }
        set { SetValue(Button.IsDefaultProperty); }
    }
    private static void OnIsDefaultChanged(DependencyObject o,
DependencyPropertyChangedEventArgs e) {}
}
```

代码 5-3 Button 中的一个 IsDefault 依赖属性

类型为 `DependencyProperty` 的 `IsDefaultProperty` 是依赖属性，不过用户使用的是暴露后的 `IsDefault` 普通的.NET 属性，从中可以看出依赖属性是 WPF 在原来的属性基础上提供给用户功能更加丰富的一种类型的属性。

依赖属性是一种类型为 `DependencyProperty` 的属性，其依赖属性标识(Dependency property identifier)则是依赖属性的实例。如 `IsDefaultProperty` 是依赖属性标识，其本身是一种依赖属性类型。但是大多数情况下不必区分是依赖属性还是其标识，从上下文即可得知。

与依赖属性相关的术语如下。

(1) `DependencyObject`: WPF 中的一种类型，继承该类后才可以注册和拥有依赖属性。



木木看到此处有些奇怪，上例中的 `Button` 不就有依赖属性，难不成它是继承了 `DependencyObject`?于是查阅 MSDN，果不其然。`Button` 的继承层次如下所示：

Button 的继承层次

```
System.Object
  System.Windows.Threading.DispatcherObject
  System.Windows.DependencyObject
  System.Windows.Media.Visual
  System.Windows.UIElement
  System.Windows.FrameworkElement
  System.Windows.Controls.Control
  System.Windows.Controls.ContentControl
  System.Windows.Controls.Primitives.ButtonBase
  System.Windows.Controls.Button
```

从上面的继承层次也可以看出 `DependencyObject` 在整个 WPF 的继承层次上处在相当高的位置，因此 WPF 绝大多数类都继承自 `DependencyObject`。

(2) **WPF 属性系统**: WPF 通过提供一系列的服务扩展了普通的.NET 属性，这些服务总称为“**WPF 属性系统**”。

(3) **.NET 属性包装器**: 指属性的 `get` 和 `set` 的实现，如代码 5-3 中的 `IsDefault` 的 `get` 和 `set` 实现。.Net 属性包装器可以把依赖属性包装成普通的.NET 属性暴露给用户使用，在这个实现中均调用 `DependencyObject` 的 `GetValue` 和 `SetValue` 方法。

依赖属性已经不像简单的面向对象语言里的属性，它更像一个计算过程，根据所输入的值经过一些计算最终得到另外一个值。整个计算过程“依赖”其他属性与内在和外在的多种因素，“依赖”正是由此而来，如图 5-1 所示。

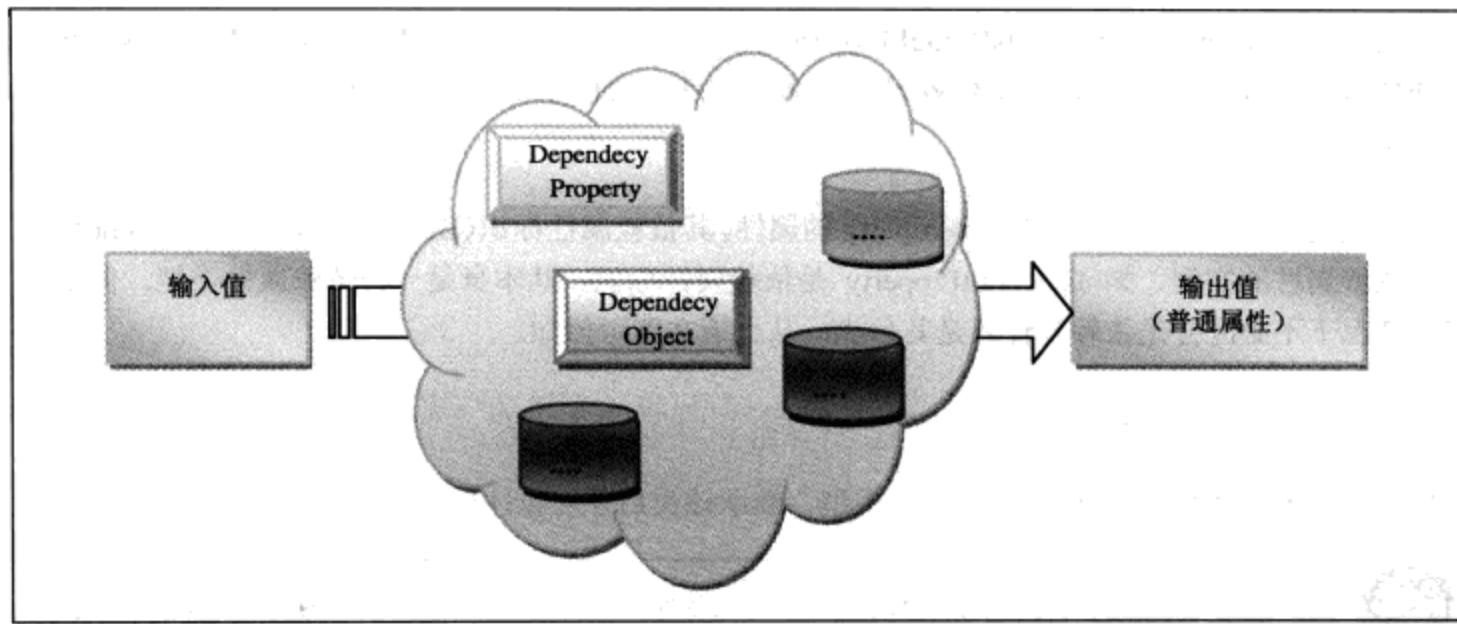


图 5-1 依赖属性的值取决于多个因素

5.2 认识依赖属性

5.2.1 分辨依赖属性

WPF 中相当多的元素的属性是依赖属性，以按钮(Button)为例，宽度(Width)、背景色(Background)和字体大小(FontSize)等都是依赖属性。为分辨一个元素的属性，可以查阅 MSDN 文档。如果该属性是依赖属性，则该文档中必然会有依赖属性信息(Dependency Property Information)这一节。按钮的宽度属性如图 5-2 所示，在其属性页面中清楚地显示了依赖属性这一节。

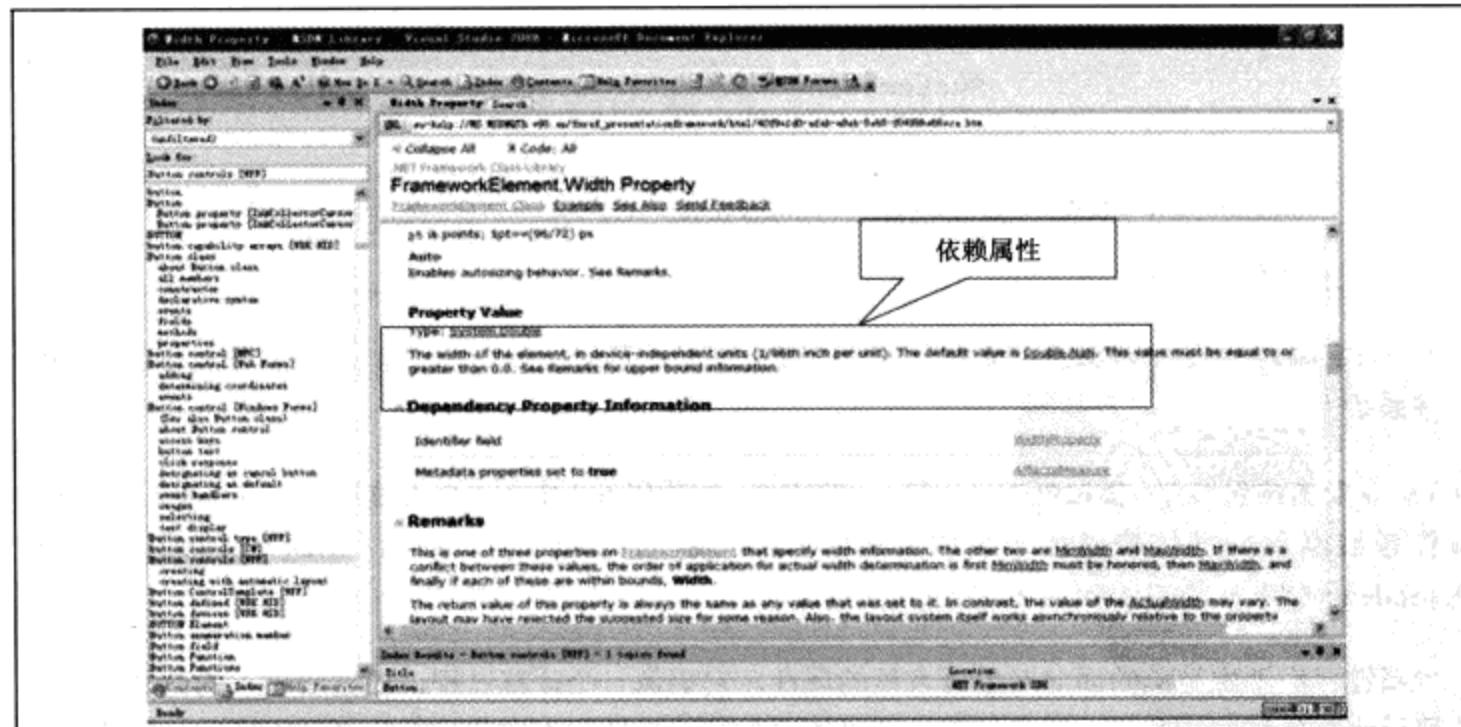


图 5-2 在 MSDN 中显示依赖属性这一节

5.2.2 引入依赖属性的原因

WPF 为什么引入依赖属性呢？我们不妨先看看 MSDN 里的一句话：

One of the primary architectural philosophies used in building WPF was a preference for properties over methods or events^[2].

这句话的意思是 WPF 主要的设计思想之一是侧重属性胜于方法和事件，即如果属性能解决问题，则坚决不使用方法和事件。

在过去不用方法和事件单单用属性是很难想象的。因为属性的功能太单一了，仅仅就是提供一个类型的值。因此 WPF 需要提供一个新的属性类型即依赖属性和与之配套的服务，让它能够做方法和事件所能做的事情。

具体来说依赖属性与以前的属性相比，提供了对资源引用、样式、动画、数据绑定、属性值继承、元数据重载及 WPF 设计器的集成支持功能的支持。这些功能都是 WPF 的重要特性，由此依赖属性在 WPF 平台中的重要地位可见一斑。

下例主要展示依赖属性（主要是按钮的背景色和字体大小属性）对上述功能的支持，程序运行结果如图 5-3 所示。图中“金色按钮”的背景色属性引用了一个画刷资源；“绿色按钮”通过样式使背景变成绿色；“动画按钮”通过动画使按钮的背景由红变绿，再由绿变红反复循环；“我被绑定成红色”按钮是通过和一个 .Net 对象的属性绑定变成红色。最下面一行是属性继承支持。从左至右依次有 3 个按钮，单击“设置窗口字体：16”按钮，则所有标签和按钮的字体大小都会变成 16；单击“设置按钮字体：8”按钮，则该按钮字体大小变成 8；单击最后一个“重置字体：12”按钮，则所有的按钮字体大小恢复为初始状态大小¹。

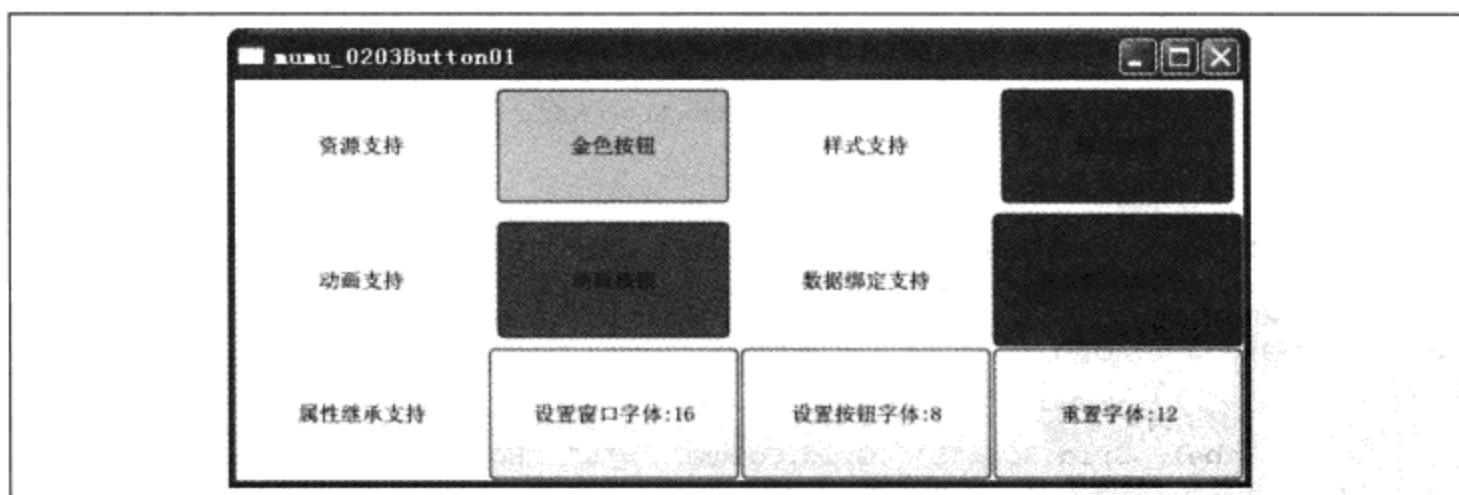


图 5-3 程序的运行结果

建立一个典型的 WPF 应用程序的工程（在 mumu_Button01\step1 文件夹下），并调整其布局，界面效果如图 5-4 所示。

¹ 在木木的机器上，字体默认大小是 12，因此所有字体大小恢复为 12。

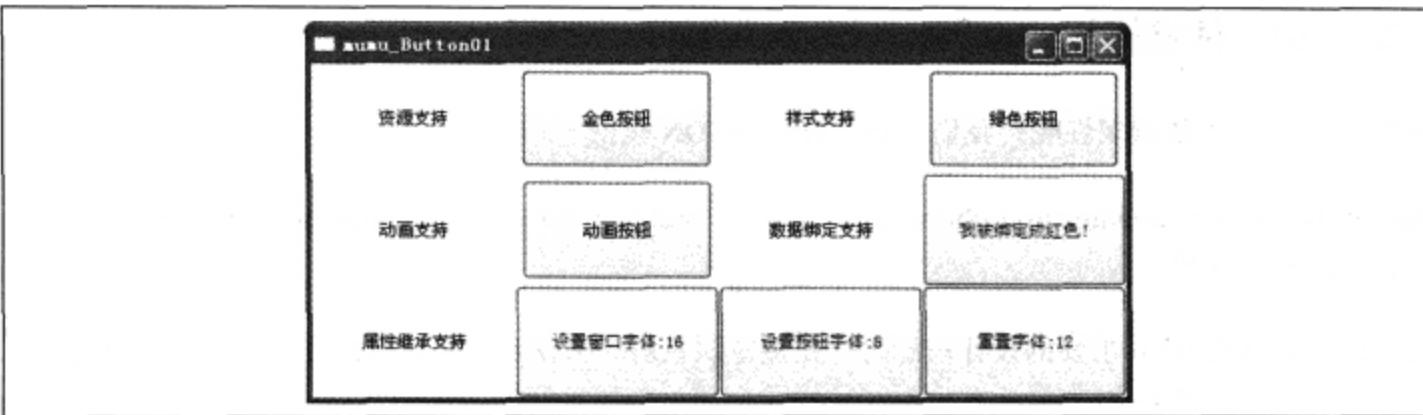


图 5-4 界面效果

MainWindow.xaml 文件内容如代码 5-4 所示。

```
MainWindow.xaml
<Window x:Class="mumu_Button01.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    Title="mumu_Button01" Height="269" Width="572">
    <Grid Name="Grid1">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <!--资源支持-->
        <Label HorizontalAlignment="Center" VerticalAlignment="Center">资源支持
    </Label>
        <Button Grid.Row="0" Grid.Column="1" Name="resouceBtn" Margin="5">金色按钮</Button>

        <!--样式支持-->
        <Label Grid.Row="0" Grid.Column="2" HorizontalAlignment="Center" VerticalAlignment="Center">样式支持</Label>
        <Button Grid.Row="0" Grid.Column="3" Name="styleBtn" Padding="0" Margin="5">绿色按钮</Button>

        <!--动画支持-->
        <Label Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center" VerticalAlignment="Center">动画支持</Label>
        <Button Grid.Row="1" Grid.Column="1" Name="animationBtn" Margin="5">动画按钮</Button>

        <!--数据绑定支持-->
        <Label Grid.Row="1" Grid.Column="2" HorizontalAlignment="Center" VerticalAlignment="Center">数据绑定支持</Label>
        <Button Grid.Row="1" Grid.Column="3" Name="BindingBtn" >我被绑定成红色!</Button>
```

```

<!--属性值继承-->
<Label Grid.Row="2" Grid.Column="0" HorizontalAlignment="Center"
VerticalAlignment="Center">属性继承支持</Label>
    <Button Grid.Row="2" Grid.Column="1" Name="FontSizeWinBtn">设置窗口字体:16</Button>
    <Button Grid.Row="2" Grid.Column="2" Name="FontSizeBtn">设置按钮字体:8</Button>
    <Button Grid.Row="2" Grid.Column="3" Name="ResetFontSizeBtn">重置字体:12</Button>

</Grid>
</Window>

```

代码 5-4 MainWindow.xaml 文件

1. 依赖属性对资源引用的支持

将金色按钮的背景色设置为金黄色，为了说明依赖属性对资源引用的支持，采用下面的做法（mumu_Button01\step2 文件夹）。

(1) 在 App.xaml 中定义一个 Key 值为 MyBrush 的画刷资源，x:Key 属性用来唯一标识该资源，如代码 5-5 所示。

```

App.xaml
<Application x:Class="mumu_Button01.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
    </Application.Resources>
</Application>

```

代码 5-5 App.xaml 文件

(2) 在 MainWindow.xaml 文件中将金色按钮的背景色属性设置为该资源的引用，如代码 5-6 所示。

```

<Button Grid.Row="0" Grid.Column="1" Name="resouceBtn" Margin="5"
Background="{DynamicResource MyBrush}">金色按钮</Button>

```

代码 5-6 MainWindow.xaml 文件

程序运行结果如图 5-5 所示。

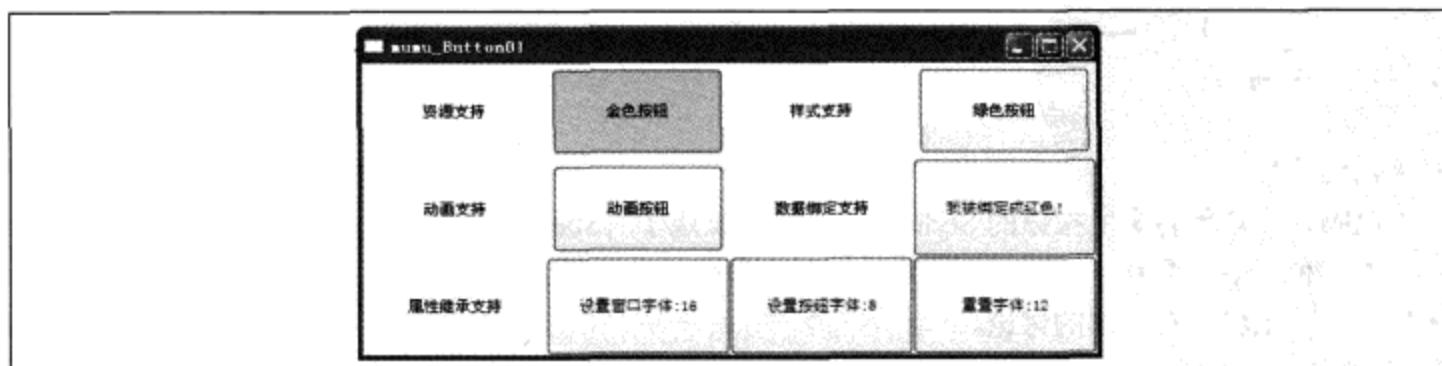


图 5-5 运行结果

2. 依赖属性对样式的支持

样式也是 WPF 当中的一个重要概念，见第 13 章。在这里我们先简单地认为样式是 WPF 为我们提供的能够方便的对多个控件应用同一个属性值的一种机制。

通过样式使绿色按钮背景为绿色的步骤如下（参见 `mumu_Button01\step3` 文件夹）。

(1) 在 `App.xaml` 文件中添加一个新的样式“`GreenButtonStyle`”，作为应用程序的一个资源。其作用是将按钮的背景色属性值设置为绿色，如代码 5-7 所示。

```
<Application.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
    <!--GreenButtonStyle 将按钮的背景色设置为绿色-->
    <Style x:Key="GreenButtonStyle">
        <Setter Property="Control.Background" Value="Green"/>
    </Style>
</Application.Resources>
```

代码 5-7 在 `App.xaml` 文件中添加一个新的样式

(2) 在 `MainWindow.xaml` 文件中绿色按钮同样通过关键字来引用该样式，将其 `Style` 属性值设置为“`{StaticResource GreenButtonStyle}`”，如代码 5-8 所示。

```
<Button Grid.Row="0" Grid.Column="3" Name="styleBtn" Padding="0" Margin="5" Style="{StaticResource GreenButtonStyle}">绿色按钮</Button>
```

代码 5-8 引用该样式

程序运行结果如图 5-6 所示。

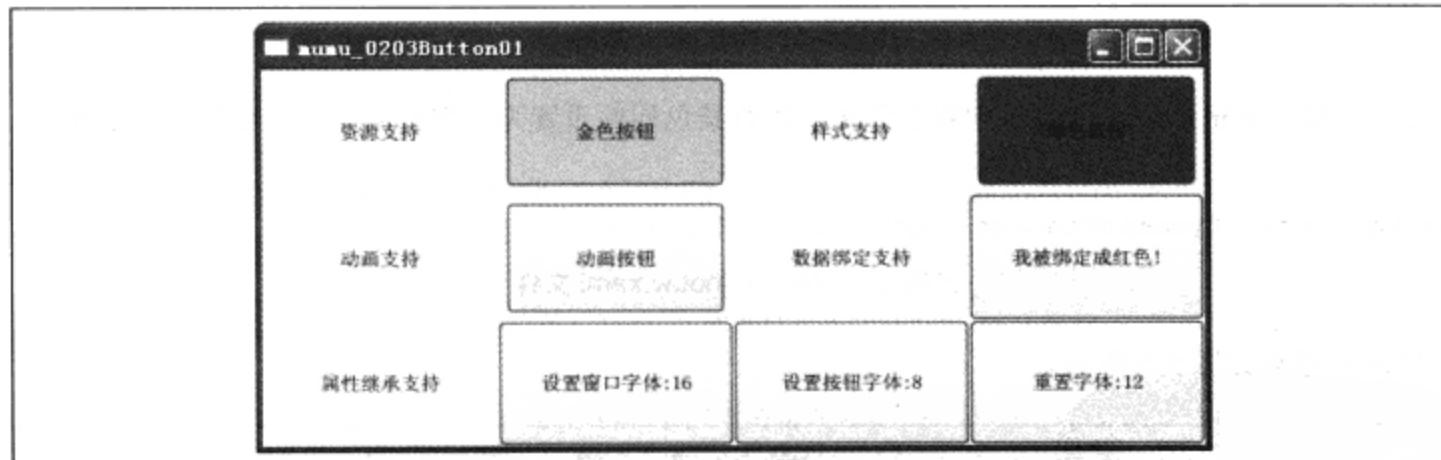


图 5-6 运行结果

木木做到这里，禁不住有这样的好奇：如果对前一个“金色按钮”也使用样式，那么金色按钮会变成绿色吗？于是他将金色按钮的 `Style` 属性值同样赋成了 `{StaticResource GreenButtonStyle}`，可是结果让木木很失望，因为金色按钮的背景颜色没有丝毫的变化。这其中的原因，我们在 5.4 所有规则大排队一节里将会有详细的讨论。

3. 依赖属性对动画的支持

为动画按钮添加如代码 5-9 所示代码（参见 `mumu_Button01\step4`），使动画按钮的背景色从红变绿并反复循环。

```
<Button Grid.Row="1" Grid.Column="1" Name="animationBtn" Margin="5">
    <Button.Background>
        <SolidColorBrush x:Name="AnimBrush"/>
    </Button.Background>
    <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <ColorAnimation
                        Storyboard.TargetName="AnimBrush"
                        Storyboard.TargetProperty="(SolidColorBrush.Color)"
                        From="Red" To="Green" Duration="0:0:5"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
    动画按钮</Button>
    ...

```

代码 5-9 为按钮添加动画效果

4. 依赖属性对数据绑定的支持

数据绑定略为复杂一些（参见 `mumu_Button01\step5` 示例）。

(1) 新建一个名为“BindingData”的类作为绑定的数据源，并绑定按钮的背景色和 BindingData 的 `ColorName` 属性，如代码 5-10 所示。

```
BindingData.cs
using System;

namespace mumu_Button01
{
    class BindingData
    {
        public BindingData()
        { ColorName = "Red"; }

        private string name = "Red";

        public string ColorName
        {
            get { return name; }
            set
            {
                name = value;
            }
        }
    }
}
```

代码 5-10 自定义一个 BindingData 类作为绑定数据源

(2) 在 App.xaml 文件中添加 BindingData 对象的资源，如代码 5-11 所示。

```
<Application.Resources>
    <!--引入一个.NET 对象的资源-->
    < local:BindingData x:Key="myDataSource"/>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
    <!--GreenButtonStyle 将按钮的背景色设置为绿色-->
    <Style x:Key="GreenButtonStyle">
        <Setter Property="Control.Background" Value="Green"/>
    </Style>
</Application.Resources>
```

代码 5-11 添加 BindingData 对象资源

如第 4 章中所述，如果需要在 XAML 文件中引用一个自定义的类，则需要像上例中那样，声明 xmlns:c="clr-namespace:mumu_Button01" 后才能使用该自定义类。

(3) 在 MainWindow.xaml 文件中将该.NET 对象绑定到“我被绑定成红色”按钮的背景色上，如代码 5-12 所示。

```
<Button Grid.Row="1" Grid.Column="3" Name="BindingBtn" Background= "{Binding Source={StaticResource myDataSource},Path=ColorName}">我被绑定成红色!</Button>
```

代码 5-12 将背景色属性和.NET 对象绑定

程序运行结果如图 5-7 所示。

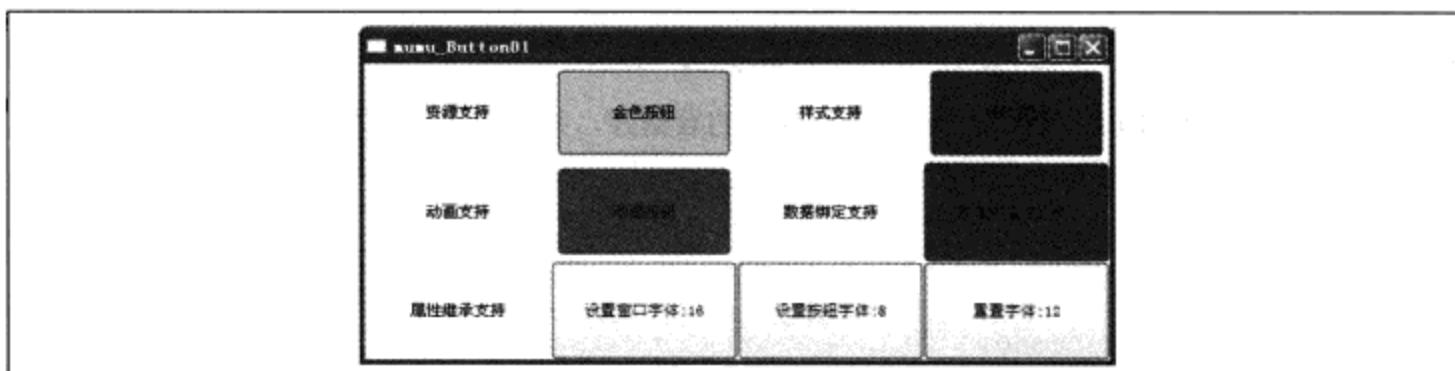


图 5-7 运行结果

5. 依赖属性对属性值继承的支持

“属性值继承”也是一个新的概念，仍以按钮的字体大小属性（FontSize）为例。

(1) 在 MainWindow.xaml.cs 文件（参见 mumu_Button01\end 文件夹）中为 MainWindow 类添加一个成员变量 _oldFontSize = 0，并在构造函数中保存当前窗口默认的字体大小值，其作用是将字体大小恢复为初始状态值。

```
MainWindow.xaml.cs
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
```

```
using System.Windows.Shapes;

namespace mumu_Button01
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            _oldFontSize = FontSize;
        }
        private double _oldFontSize = 0;
    }
}
```

(2) 为“设置窗口字体:16”按钮添加一个单击事件处理函数，如代码 5-13 所示。

```
MainWindow.xaml
...
<Button Grid.Row="0" Grid.Column="0" Name="FontSizeWinBtn" Margin="5"
Click="FontSizeWinBtn_Click">设置窗口字体:16</Button>
...
```

代码 5-13 为按钮添加单击事件处理函数

(3) 在 MainWindow.xaml.cs 文件中添加该事件的行为，注意 FontSize 属性是整个窗口的属性。即单击该按钮会将窗口的字体大小属性值赋为 16，如代码 5-14 所示。

```
MainWindow.xaml.cs
...
private void FontSizeWinBtn_Click(object sender, RoutedEventArgs e)
{
    FontSize = 16;
}
...
```

代码 5-14 为窗口的 FontSize 属性赋值

(4) 为“设置按钮字体:8”的按钮添加事件处理函数，注意这里是设置该按钮的字体大小，如代码 5-15 和代码 5-16 所示。

```
MainWindow.xaml
...
<Button Grid.Row="1" Grid.Column="0" Name="FontSizeBtn" Margin="5"
Click="FontSizeBtn_Click">设置按钮字体:8</Button>
...
```

代码 5-15 为按钮添加单击事件处理函数

```
MainWindow.xaml.cs
...
private void FontSizeBtn_Click (object sender, RoutedEventArgs e)
{
    this.FontSizeBtn.FontSize = 8;
}
```

代码 5-16 为 FontSizeBtn 按钮的 FontSize 属性赋值

(5) 为“重置字体:12”的按钮添加事件处理函数，这里将所有的字体大小都还原成初始的默认值，如代码 5-17 和代码 5-18 所示。

```
MainWindow.xaml
...
<Button Grid.Row="2" Grid.Column="0" Name="ResetFontSizeBtn" Margin="5"
Click="ResetFontSizeBtn_Click">重置字体:12</Button>
...
```

代码 5-17 为按钮添加单击事件处理函数

```
MainWindow.xaml.cs
...
private void ResetFontSizeBtn_Click (object sender, RoutedEventArgs e)
{
    FontSize = _oldFontSize;
    this.FontSizeBtn.FontSize = _oldFontSize;
}
...
```

代码 5-18 重置窗口和按钮的 FontSize 属性值

窗口的初始状态所有的字体大小均为 12（默认值），如图 5-8 所示。

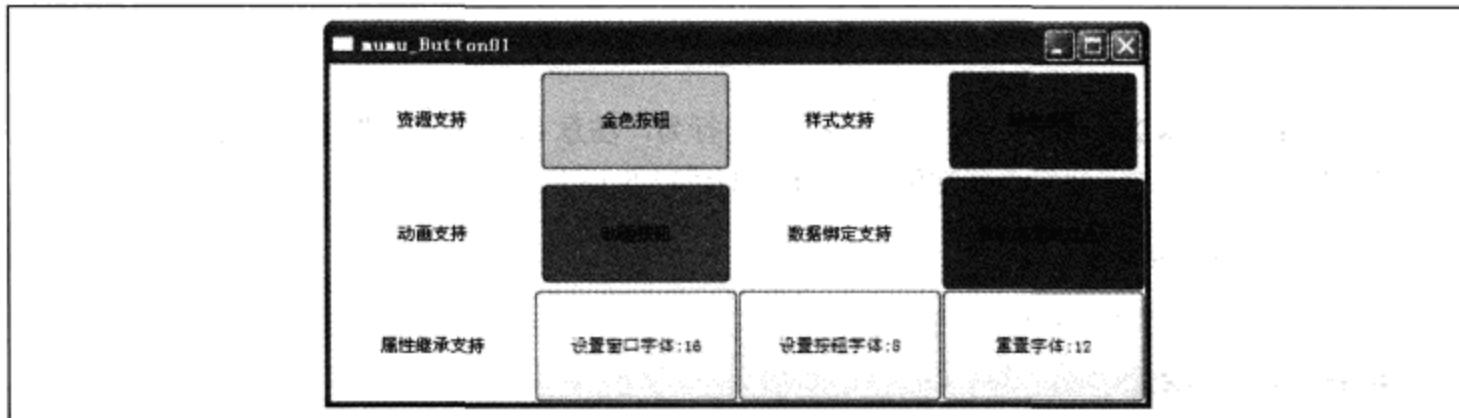


图 5-8 窗口的初始状态所有的字体大小均为 12（默认值）

当单击“设置窗口字体:16”按钮时会发现设置窗口字体大小会影响所有窗口下面有字体大小的元素，这里主要是标签（Label）和按钮（Button），此即为“属性值继承”。这种继承需要区别于面向对象编程的“继承”概念，它实质是一个树中的子对象沿袭了父对象的属性。如图 5-9 所示。

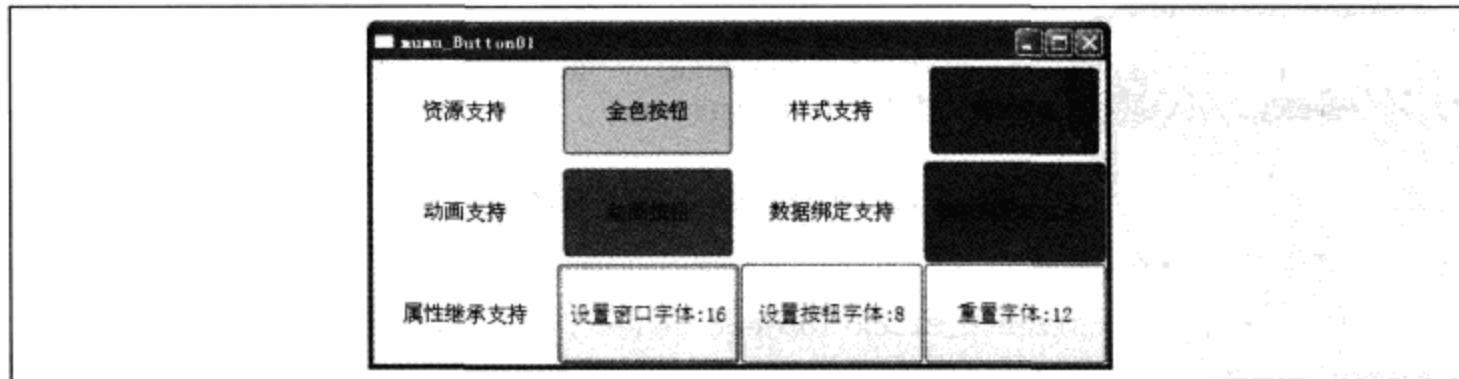


图 5-9 所有字体变为 16

当单击“设置按钮字体:8”的按钮时，只有该按钮字体大小变小。如图 5-10 所示。

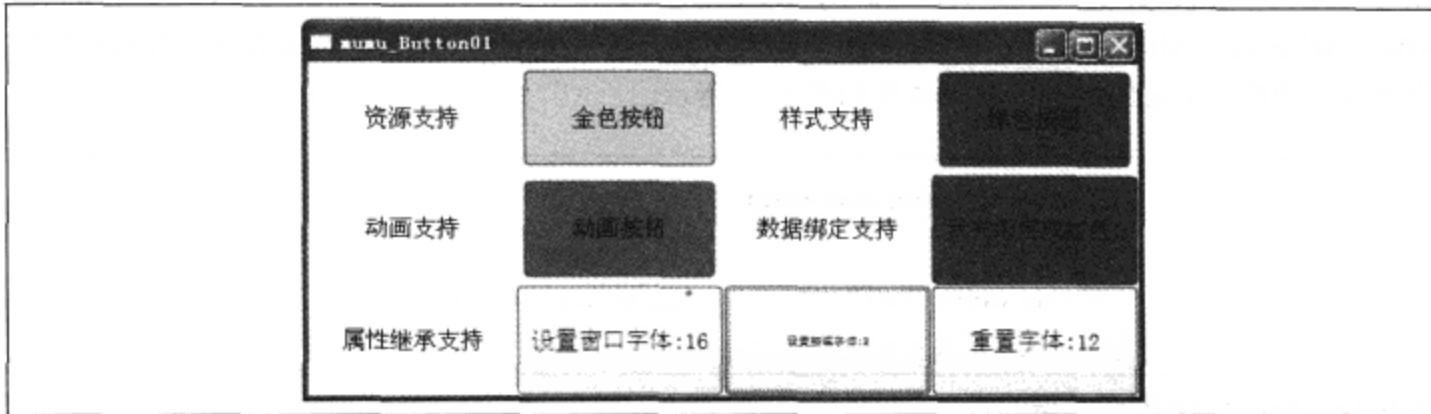


图 5-10 “设置按钮字体:8”的按钮字体大小变为 8

当单击“重置字体:12”的按钮，则全部标签和按钮恢复这原状。

6. 依赖属性对元数据重载的支持

依赖属性和普通.NET 属性的区别之一是有一个元数据对象，元数据和依赖属性是一对一的关系。通过设置元数据对象，可以改变依赖属性的状态和行为。如元数据会规定“你的默认值必须是红色”、“你这个整型值只能在 3~10 之间”或者“你发生了改变一定要通知某某”等。

一般用到的元数据类是 `PropertyMetaData` 和 `FrameworkPropertyMetaData`，前者是后者的基类，如图 5-11 所示。

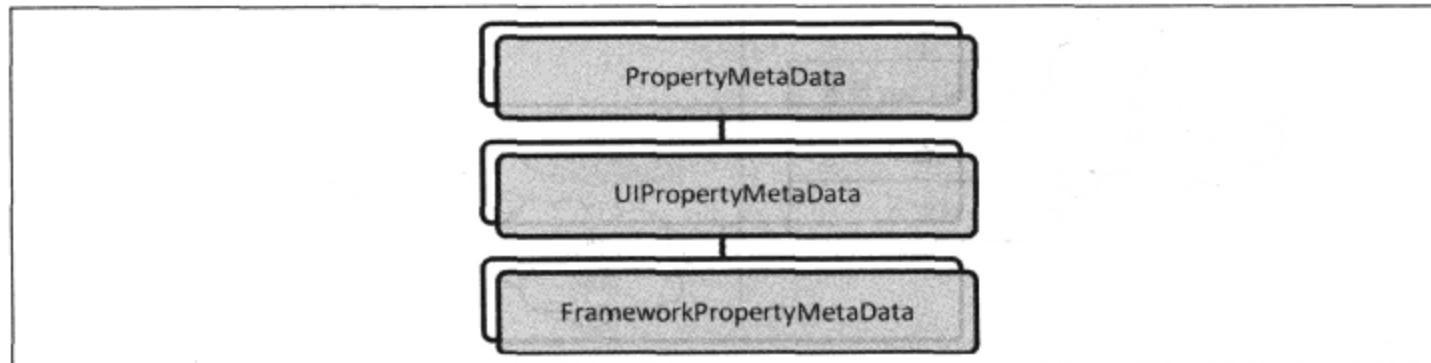


图 5-11 元数据类的继承结构

一般依赖属性的元数据的类型为 `PropertyMetaData`，而大部分控件的依赖属性，如按钮的 `Width` 及 `Background` 这样的依赖属性就会用到 `FrameworkPropertyMetaData` 元数据对象。

一般元数据对象包括如下类型的信息。

- (1) 默认值：如 `Background` 的默认值是红色，`Width` 的默认值是 30 等。
- (2) 引用回调函数：其中 `PropertyChangedCallback`，在属性值发生改变时调用；`CoerceValueCallback` 用于限制属性值，如一个进度条的当前值需要限制在最小值和最大值之间。如果该值小于最小值 `minimum`，则为 `minimum`；如果大于最大值 `maximum`，则为 `maximum`。

(3) 如果是框架级别的一些属性，如按钮的 Width, Background 等，则会有一些标识告知 WPF 该属性的某些状态信息，这是 FrameworkPropertyMetadata 类型的元数据的特点。如前例的 FontSize 属性，AffectsMeasure、AffectsRender 和 Inherits 即此类信息，其中 Inherits 为 true 意味着该属性具有属性值继承的特性²。FontSize 的元数据特性如图 5-12 所示。

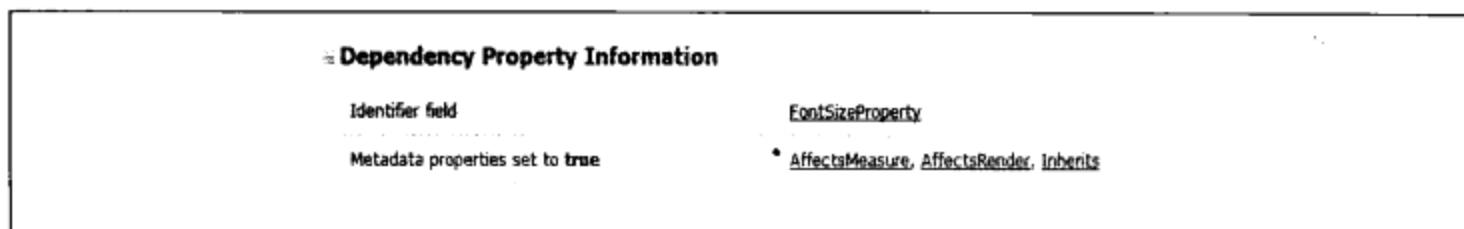


图 5-12 FontSize 的元数据特性

有了这个元数据对象，我们编写程序的思想也发生了一些变化。在过去，如果一个继承类需要一个属性 A，这个属性 A 和基类的一个属性 B “有点像又有点不像”的时候，左思右想，最后我们只好放弃代码优美，硬生生地在继承类里重新加上了一个属性 A。但是现在在 WPF 里遇到了这种情况，我们仍然保留基类的这个属性 B，但是将它的元数据对象换一个就可以满足要求，就被称为元数据重载。如图 5-13 所示。

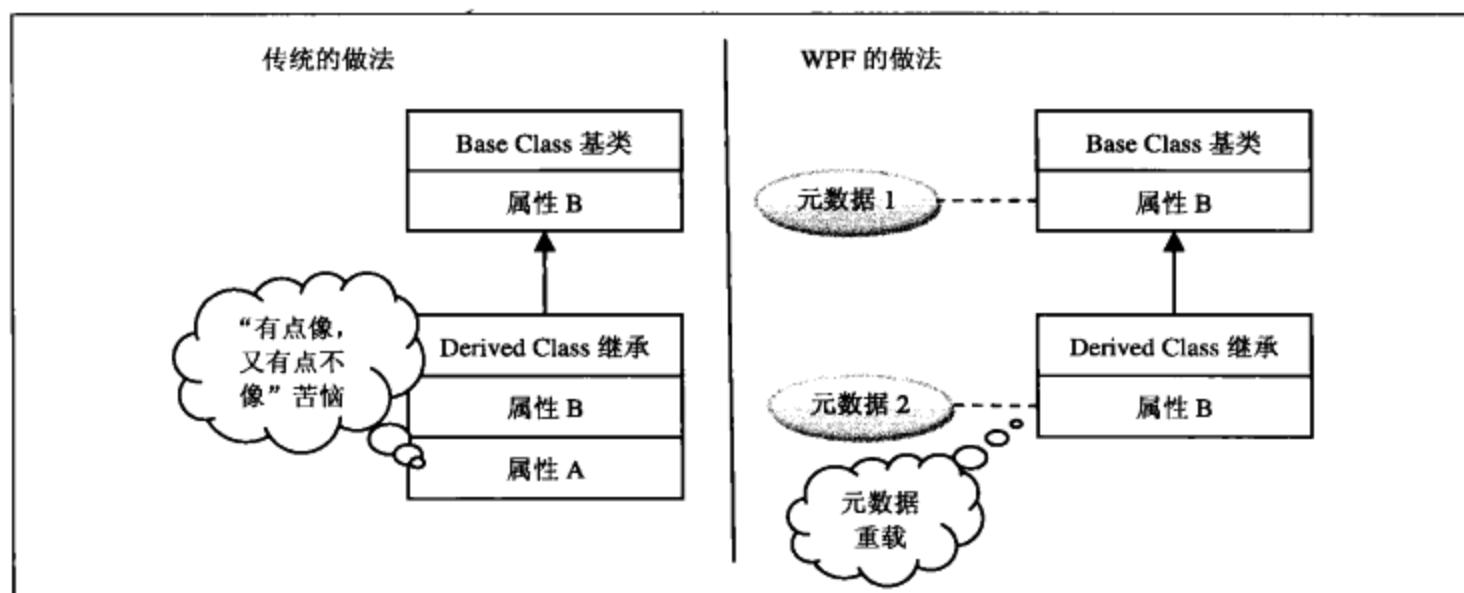


图 5-13 传统和 WPF 的做法

7. 依赖属性对 WPF 设计器的集成支持

对 WPF 设计器的集成支持主要体现在自定义一个控件，如自定义一个按钮并为其添加了一个依赖属性，那么在 WPF 的属性窗口中会显示该项，而不显示普通属性。

5.2.3 依赖属性的组成部分

还记得代码 5-3 吗？我们不妨再把它看一遍。

² 这些标识的详细意义请参见 MSDN 中的 FrameworkPropertyMetadataOptions，该页面的链接为 ms-help://MS.MSDNQTR.v90.en/fxref_system.windows/html/45301b3f-9749-bed7-8468-50804cefc5bc.htm。

依赖属性总是和普通的.NET 属性配合使用的，而且我们注意观察 Register 函数的参数，为了方便说明问题，我们列举了参数最全的一个 Register 函数声明，如代码 5-19 所示：

```
最全参数的 Register 函数声明
public static DependencyProperty Register(
    string name,
    Type propertyType,
    Type ownerType,
    PropertyMetadata typeMetadata,
    ValidateValueCallback validateValueCallback
)
```

代码 5-19 Register 函数声明

第一个参数是指的属性名，这个名字和依赖属性相关的.NET 属性名相同。第二个参数是指该.NET 属性的类型。第三个参数是该依赖属性属于什么类型，比如 IsDefaultProperty 注册的时候，这个参数就应该输入 `typeof(Button)`。第四个参数就是依赖属性的元数据。第五个参数是值验证的回调函数，该回调函数是负责检验值的最后一道关卡，该函数返回一个 `bool` 值，如果值检验正确就返回真，反之返回假。比如有一个依赖属性是月份 `month`，那么这个变量只可能是一月、二月、三月……十二月。这个值检验函数实现如代码 5-20 所示。

```
PRIVATE static bool MonthValidateCallback(object value)
{
    Month mon = (Month)value;
    return (mon == Months.Jan
        || mon == Months.Feb
        || mon == Months.Mar
        || mon == Months.Apr
        || mon == Months.May
        || mon == Months.Jun
        || mon == Months.Jul
        || mon == Months.Aug
        || mon == Months.Sep
        || mon == Months.Oct
        || mon == Months.Nov
        || mon == Months.Dec)
```

代码 5-20 月份验证回调函数

由上面的分析可知，依赖属性会涉及这几个部分：依赖属性变量（如 `IsDefaultProperty`）、普通的.NET 属性（`IsDefault`）、.NET 属性包装器、元数据和值验证函数。下一节基本上就是和这几个部分打交道了。

5.3 自定义依赖属性

5.3.1 何时需要自定义一个依赖属性

这个问题看似不好回答，实际非常容易解答。上一节讲了依赖属性支持某某功能。那么也就是说当我们自定义一个属性的时候，需要让该属性支持那些功能，那么我们就必须把它实现成依赖属性。假定我们现在设计一个属性，变量名叫做……叫做什么了？就叫它“萌萌”吧^_^！我们再比对上面

的功能说一遍：

- (1) 希望“萌萌”支持动态资源引用。
- (2) 希望在样式中使用“萌萌”。
- (3) 希望“萌萌”支持动画。
- (4) 希望“萌萌”支持数据绑定。
- (5) 希望“萌萌”支持属性值继承。
- (6) 希望“萌萌”发生改变时触发一系列行为。
- (7) 希望“萌萌”有自己的元数据。
- (8) 希望“萌萌”得到 WPF 设计器的支持，如在 WPF 属性窗口中直接修改其值。

那么我们就把“萌萌”实现成一个依赖属性。

实现一个依赖属性必须满足以下条件。

- (1) 该类必须继承自 DependencyObject 类，只有 DependencyObject 类才可以注册和拥有依赖属性。
- (2) 该类中必须定义一个 public static readonly 成员变量，类型为 DependencyProperty，如“public static readonly DependencyProperty IsDefaultProperty;”。
- (3) 该依赖属性名必须以“属性名+Property”命名，如 Button 的 IsDefault 属性，命名为“IsDefaultProperty”。
- (4) 必须调用 DependencyProperty 的注册方法（Register 及 RegisterReadOnly）在 WPF 属性系统中注册该依赖属性或者使用依赖属性的 AddOwner 方法。两种方法均返回一个 DependencyProperty 类型的标识并将其保存在定义的 DependencyProperty 成员变量中。
- (5) 为依赖属性实现一个.NET 属性包装器。

5.3.2 自定义依赖属性示例

该例继承一个按钮类，实现一个自定义按钮。为该按钮添加一个依赖属性 Space，用来控制按钮字符的间距。然后将该属性实现成依赖属性，使其具备属性值继承的特性，运行结果如图 5-14 所示。

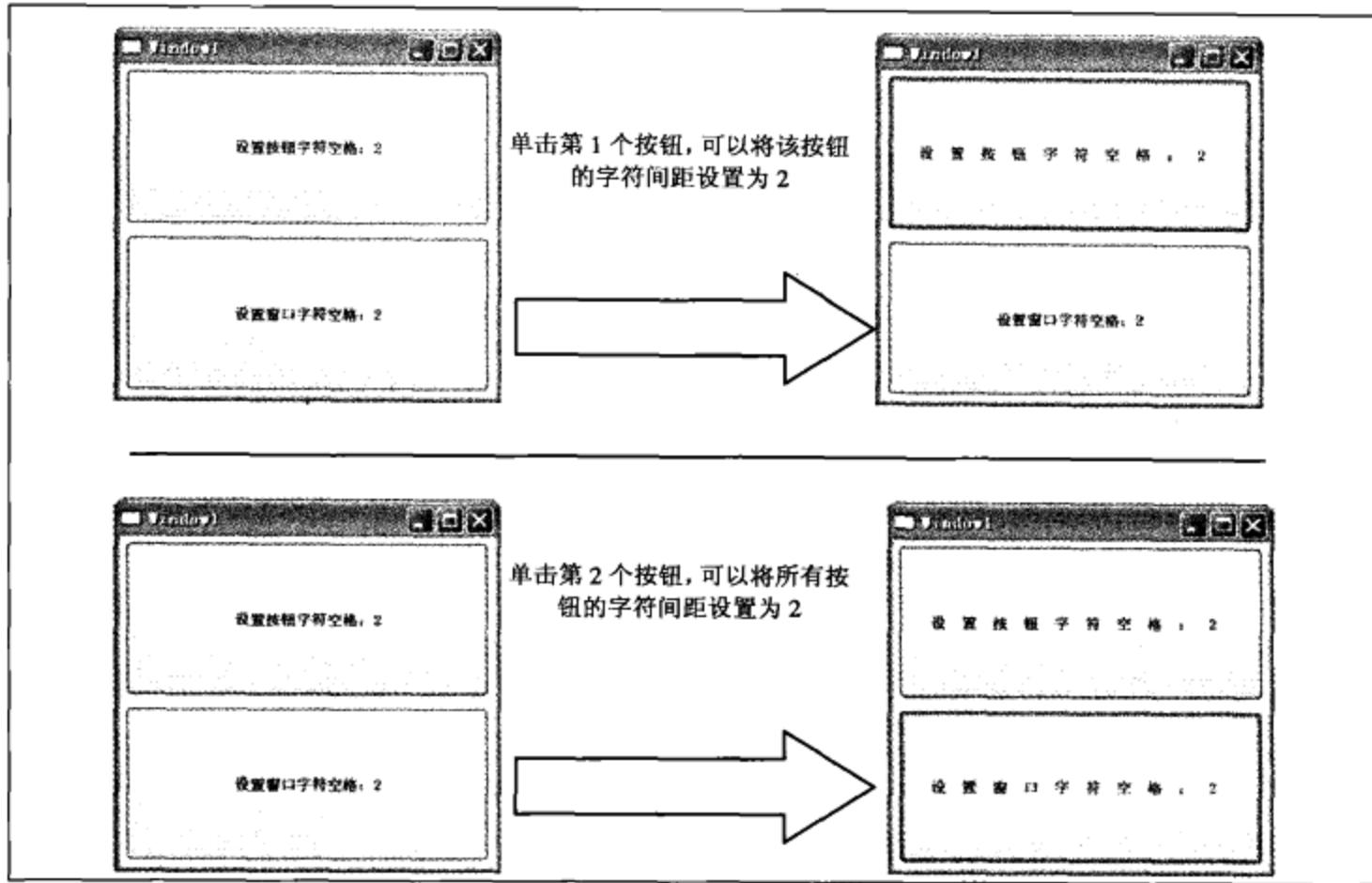


图 5-14 mumu_Button02 工程的运行结果

1. 自定义 Button 按钮

自定义 SpaceButton 按钮，如代码 5-21 所示。

```
SpaceButton.cs
using System;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace mumu_Button02
{
    public class SpaceButton : Button
    {
        ①         // 传统.NET 做法 私有字段搭配一个公开属性
        string txt;

        public string Text
        {
            set
            {
                txt = value;
                Content = SpaceOutText(txt);
            }
            get
            {
                return txt;
            }
        }
    }
}
```

```

    }

    // ② 依赖属性
    public static readonly DependencyProperty SpaceProperty;

    // ③ .NET 属性包装器
    public int Space
    {
        set
        {
            SetValue(SpaceProperty, value);
        }
        get
        {
            return (int)GetValue(SpaceProperty);
        }
    }

    // ④ 静态的构造函数
    static SpaceButton()
    {
        // 定义元数据
        FrameworkPropertyMetadata metadata = new
        FrameworkPropertyMetadata();
        metadata.DefaultValue = 0;
        metadata.PropertyChangedCallback += OnSpacePropertyChanged;

        // 注册依赖属性
        SpaceProperty = DependencyProperty.Register("Space",
        typeof(int), typeof(SpaceButton), metadata, ValidateSpaceValue);
    }

    // ⑤ 值验证的回调函数
    static bool ValidateSpaceValue(object obj)
    {
        int i = (int)obj;
        return i >= 0;
    }

    // ⑥ 属性值改变的回调函数
    static void OnSpacePropertyChanged(DependencyObject obj,
    DependencyPropertyChangedEventArgs args)
    {
        SpaceButton btn = obj as SpaceButton;
        string txt = btn.Content as string;
        if (txt == null) return;
        btn.Content = btn.SpaceOutText(txt);
    }

    // ⑦ 该方法为字符间距添加空格
    string SpaceOutText(string str)
    {
        if (str == null)
            return null;

        StringBuilder build = new StringBuilder();

        // 在其中添加 Space 个空格
        foreach (char ch in str)
            build.Append(ch + new string(' ', Space));
    }
}

```

```
        return build.ToString();
    }
}
}
```

代码 5-21 SpaceButton.cs 文件

SpaceButton 继承 Button 类，添加了一个普通属性 Text。这是传统的.NET 做法，即一个私有字段 txt 搭配一个 Text 属性（代码①）。一个依赖属性 SpaceProperty（代码②）表示字符的空格间距，如图 5-15 所示。

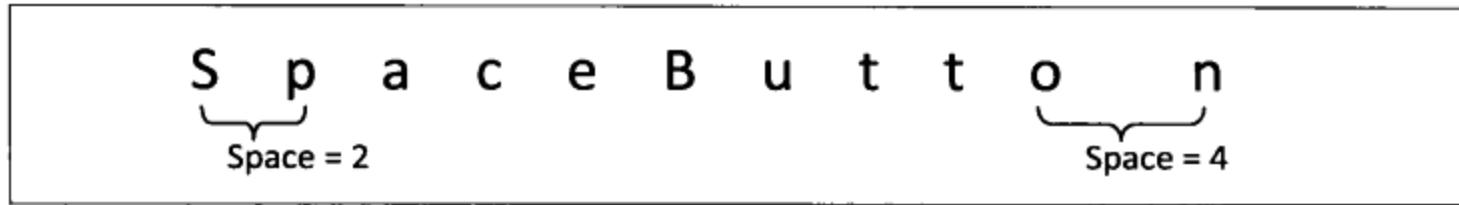


图 5-15 Space 的含义

为 SpaceProperty 实现一个.NET 属性包装器（代码③），因为依赖属性本身是一个静态变量，所以在类的静态构造函数中创建和设置元数据，并注册依赖属性（代码④）。这里元数据设置 Space 的默认值 0，并且添加了一个实现的 OnSpacePropertyChanged 回调函数（代码⑥）。只要 Space 属性值发生改变，则调用这个回调函数。

在注册时还设置了一个实现的 ValidateSpaceValue 值验证回调函数（代码⑤），用于检查 Space 值是否大于等于 0。如果该值在外部设置小于 0，则程序会报告异常。注意两个回调函数都是静态方法。

SpaceOutText 方法（代码⑦）主要为现有字符串添加空格，空格的个数由 Space 变量决定。

2. 外部调用自定义按钮

在 MainWindow.xaml 文件中调用 SpaceButton 按钮，如代码 5-22 所示。

```
MainWindow.xaml
<Window x:Class="mumu_Button02.SpaceWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local ="clr-namespace:mumu_Button02"
    Title="mumu_Button02" Width="300" Height="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <local:SpaceButton x:Name="btnSpace" Grid.Column="0" Grid.Row="0"
Margin="5" Click="btnSpace_Click" Text="设置按钮字符空格: 2">
        </local:SpaceButton>
        <local:SpaceButton x:Name= "winSpace" Grid.Column="0" Grid.Row="1"
Margin="5" Click="winSpace_Click" Text="设置窗口字符空格: 2">
        </local:SpaceButton>
    </Grid>
</Window>
```

代码 5-22 在 MainWindow.xaml 文件中调用 SpaceButton 按钮

注意一是在 XAML 中调用自定义类，需要声明 xmlns:local，然后调用时的标签是<local:SpaceButton>.....；二是自定义的普通.NET 属性 Text，也可在 XAML 文件中设置。

为第 1 个按钮添加方法，如代码 5-23 所示。

```
MainWindow.xaml.cs
...
private void BtnSpace_Click(object sender, RoutedEventArgs e)
{
    this.btnSpace.Space = 2;
}
...
```

代码 5-23 为 SpaceButton 实现 Click 事件处理函数

单击第 1 个按钮时，该按钮的字符间距就拉大了，如图 5-16 所示。



图 5-16 第 1 个按钮的字符间距变为 2

3. 为依赖属性增加属性值继承的特性

为 Space 增加属性值继承特性，首先为窗口增加 Space 依赖属性，如代码 5-24 所示。

```
MainWindow.xaml.cs
...
static SpaceWindow()
{
    FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
    metadata.Inherits = true;
    SpaceProperty = SpaceButton.SpaceProperty.AddOwner(typeof(SpaceWindow));
    SpaceProperty.OverrideMetadata(typeof(Window), metadata);
}

public static readonly DependencyProperty SpaceProperty;
public int Space
{
    set
    {
        SetValue(SpaceProperty, value);
    }
    get
    {
```

```
        return (int)GetValue(SpaceProperty);
    }
}

....
```

代码 5-24 为窗口增加 Space 依赖属性

注意依赖属性 Space 不是通过注册而来，而是从 SpaceButton 的 Space 属性的 AddOwner 方法得来的。即依赖属性可以选择把自身添加给其他属性，这是普通属性不可实现的。需要特别注意原始元数据不能再使用，必须新建一个。为了实现属性值继承，将 Inherit 标识设置为 true。

在 SpaceButton.cs 代码中将 SpaceButton 的 Space 的元数据 Inherit 标识也设置为 true，如代码 5-25 所示。

```
SpaceButton.cs
namespace mumu_Button02
{
    public class SpaceButton : Button
    {
        ....
        // 静态构造函数
        static SpaceButton()
        {
            // 定义元数据
            FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
            metadata.DefaultValue = 0;
            metadata.PropertyChangedCallback += OnSpacePropertyChanged;
            metadata.Inherits = true;
            // 注册依赖属性
            SpaceProperty = DependencyProperty.Register("Space", typeof(int),
                typeof(SpaceButton), metadata, ValidateSpaceValue);
        }
        ....
    }
}
```

代码 5-25 将 SpaceButton 的 Space 依赖属性

为第 2 个按钮添加行为代码，如代码 5-26 所示。

```
MainWindow.xaml.cs
.....
private void BtnSpace_Click(object sender, RoutedEventArgs e)
{
    this.btnSpace.Space = 2;
}

private void WinSpace_Click(object sender, RoutedEventArgs e)
{
    this.Space = 2;
}
....
```

代码 5-26 实现第 2 个按钮的 Click 事件处理函数

这里设置 SpaceWindow 的 Space 属性，但是由于属性值继承关系，所以单击第 2 个按钮时两个按钮的字符间距均变为两个空格，如图 5-17 所示。



图 5-17 由于 Space 属性继承，因此两个按钮的间距均为 2

5.4 所有规则大排队

5.4.1 按钮到底是什么颜色

首先查看代码 5-27：

```
按钮的背景色
<Button Background="Red">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Green"/>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="Blue" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
    Click
</Button>
```

代码 5-27 在 3 处设置了按钮的背景色

该代码中有 3 处（加粗处）设置了按钮的背景色，那么按钮的背景色到底是红色？绿色还是蓝色？这里面就牵涉到“优先级”的问题。不妨还是先运行一下程序，看看结果如何？

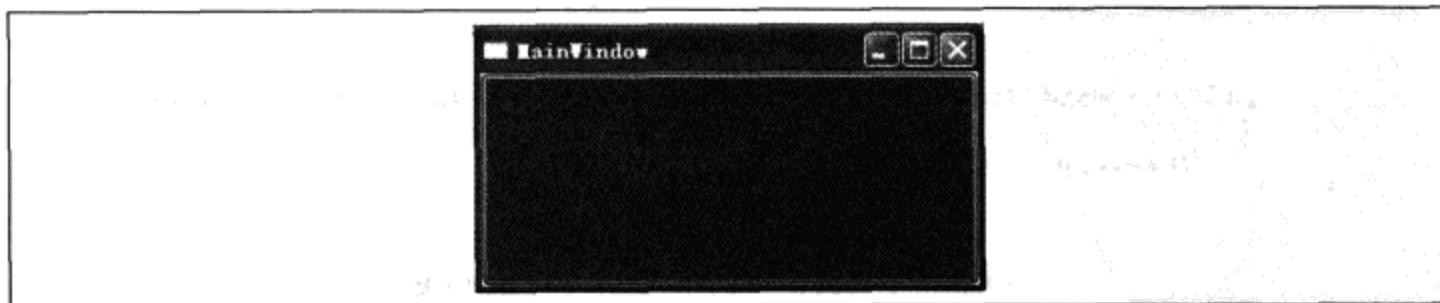


图 5-18 程序运行结果

结果是红色，鼠标移动到按钮上时按钮颜色未改变。说明直接设置的值具有最高优先级，甚至能够屏蔽触发器设置属性的行为。删除 `Background = "Red"`，如代码 5-28 所示。

```
按钮的背景色
<Button Background="Red">
    <Button.Style>
        ...
    </Button.Style>
    Click
    </Button>
```

代码 5-28 删除 Background="Red"

运行结果是如果鼠标没有移到按钮上，按钮的背景色为绿色；否则为蓝色。



如此说明优先级是“直接设置的值>样式中触发器设置的值>样式中 setter 设置的值”。

5.4.2 依赖属性设置优先级列表

我们这一章开头把依赖属性比做小红马，主要还是因为 WPF 提供了很多种设置依赖属性值的方法。如果不搞清楚这些方法的优先级，确实很难驾驭依赖属性。一个依赖属性从设置到最终结果的完整流程如图 5-19 所示。

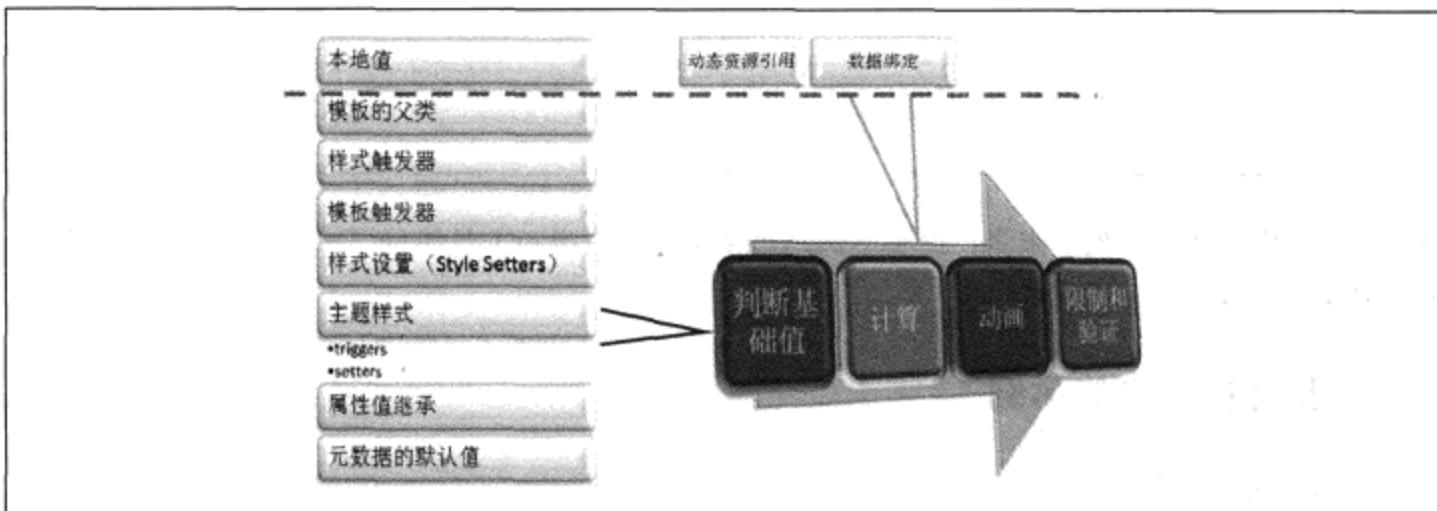


图 5-19 依赖属性从设置到最终结果的完整流程

还记得图 5-1 里面的“那团云”吧，这就是云开雾散的真风景……

依赖属性经历了 4 个步骤，即判断基础值、计算、动画，以及限制和验证。基础值是依赖属性在计算之前的值，主要相对计算和动画而言；本地值主要指在代码中直接通过.NET 属性包装器或者在 XAML 中设置的值。如果依赖属性的值是一个资源（动态和静态）或者一个数据绑定的引用，也称为“本地值”。

WPF 设置依赖属性值方法的优先级由横向和纵向两个方面决定，横向后面的步骤优先级高；纵向主要是第 1 步和第 2 步中 WPF 规定的各个方法的优先级顺序，优先级依次从高到低排列。

按照图 5-19，一个整体的优先级列表如表 5-1 所示。

表 5-1 优先级列表

序号	名称	描述
1	限制 (Coerce)	如果依赖属性已经注册 CoerceValueCallback，可以在该函数的实现中修改依赖属性的值进行修改，比较典型的是 WPF 中 ProgressBar 的 Value 依赖属性使用该回调函数将值限制在最小值和最大值之间
2	动画	如果动画在运行，则其可以改变依赖属性值
3	本地值	本地值包括在代码和 XAML 中直接设置的值，以及动态资源引用和数据绑定
4	模 板 的 父 类 (TemplatedParent)	大部分元素的模板父类 (TemplatedParent) 属性为空，只有在模板中创建的元素才有其模板父类，因此模板的父类对所有依赖属性无效。在模板的父类中 Triggers 设置依赖属性值的优先级高于在 Setters 中设置依赖属性值
5	样式触发器	主要指在 Application 或者 Page 中的样式，不包括主题样式
6	模板触发器	参见第 13 章
7	样式设置	不包括主题样式
8	主题样式	参见第 20 章
9	属性值继承	
10	元数据的默认值	

尽管把所有优先次序都列了出来，但木木仍然感觉是一团雾水。于是他自己又做了几个例子来验证了这个优先级列表。

5.4.3 验证优先级的示例

1. 设置本地值 > 模板父类 > 样式

什么是 TemplatdParent？比如一个按钮按照模板创建的，并且这个模板中有一个椭圆元素，那么这个椭圆的 TemplatdParent 是按钮。

下例验证设置本地值 > 模板父类 > 样式 (mumu_Button03 工程)，运行结果如图 5-20 所示。其中的按钮由模板 ButtonTemplate 生成，该模板由一个按钮和一个椭圆元素组成。

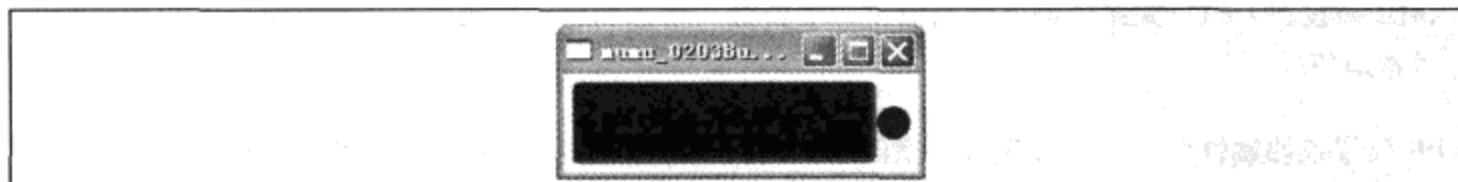


图 5-20 mumu_Button03 工程的运行结果

代码如代码 5-29 所示。

```
MainWindow.xaml
<Window x:Class="mumu_Button03.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="mumu_Button03" SizeToContent="WidthAndHeight">
    ①   <Window.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Red"/>
        </Style>
    ②   <ControlTemplate TargetType="{x:Type Button}">
        <x:Key="ButtonTemplate">
            <ControlTemplate.Resources>
                <Style TargetType="{x:Type Button}">
                    <Setter Property="Background" Value="Yellow"/>
                </Style>
            </ControlTemplate.Resources>
            <Grid Background="AliceBlue">
                ③       <Grid.ColumnDefinitions>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                </Grid.ColumnDefinitions>
                <Button Background="Blue" Grid.Column="0"
                    Width="180" Height="50" >
                    </Button>
                    <Ellipse Grid.Column="1" Fill="{Binding
                        RelativeSource={RelativeSource TemplatedParent}, Path = Background}"
                        Width="20" Height="20" />
                </Grid>
            </ControlTemplate>
        </Window.Resources>
        <Grid Name="Re">
            <Grid.RowDefinitions>
                <RowDefinition />
            </Grid.RowDefinitions>
            <Button Margin="5" Grid.Row="0" Template="{StaticResource
                ButtonTemplate}"></Button>
        </Grid>
    </Window>
```

代码 5-29 在 3 处设置了按钮的背景色

该代码中的三处按钮的背景色，一是在样式中将其设为红色（代码①）；二是在模板的资源样式中将其设为黄色（代码②）；三是在设置本地值中将其设为蓝色（代码③），第一次程序运行时按钮的背景色为蓝色。

如果删除本地值设置，那么按钮的背景色应该是什么颜色？这里需要遵循模板的父类优先级高于样式。模板中的按钮设置的样式优先级高于外部样式，因此按钮的背景色变为黄色，如图 5-21 所示。证明模板的父类优先级高于样式设置，如代码 5-30 所示。

```
MainWindow.xaml
<Window x:Class="mumu_Button03.MainWindow"
    .....
    <ControlTemplate TargetType="{x:Type Button}" x:Key="ButtonTemplate">
        .....
        <Button Background="Blue" Grid.Column="0" Width="180" Height="50" >
```

```
</ControlTemplate>
.....
</Window>
```

代码 5-30 将本地值设置删除

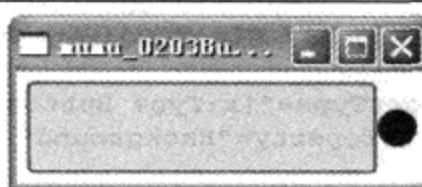


图 5-21 按钮的背景色变为黄色

删除 ControlTemplate 资源中的样式，按钮的背景色变为样式中设置的红色，如代码 5-31 所示。

```
MainWindow.xaml
<Window x:Class="mumu_Button03.MainWindow"
.....
    <ControlTemplate TargetType="{x:Type Button}" x:Key="ButtonTemplate">
        <ControlTemplate.Resources>
            <Style TargetType="{x:Type Button}">
                <Setter Property="Background" Value="Yellow"/>
            </Style>
        </ControlTemplate.Resources>
    </ControlTemplate>
</Window>
```

代码 5-31 删除模板里的样式设置

运行结果如图 5-22 所示。

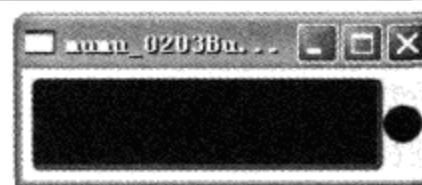


图 5-22 按钮的背景色变为红色

2. 样式触发器>模板触发器

代码 5-32 验证样式触发器的优先级高于模板触发器（参见 mumu_Button04 文件夹）：

```
MainWindow.xaml
<Window x:Class="mumu_Button04.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" SizeToContent="WidthAndHeight">
    <Window.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate>
                        <Ellipse Width="50" Height="50"
                            Fill="{TemplateBinding Background}">
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Button Grid.Column="1" Grid.Row="1" Content="按钮1" />
        <Button Grid.Column="2" Grid.Row="1" Content="按钮2" />
    </Grid>
</Window>
```

```

    </Ellipse>
    <ControlTemplate.Triggers>
        ①      <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="Background" Value="Green" />
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
    <Setter.Value>
    </Setter>
    <Style.Triggers>
        ②      <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="Background" Value="Blue" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<Grid>
    <Button/>
</Grid>
</Window>

```

代码 5-32 mumu_Button04 中的 MainWindow.xaml 文件

这个按钮仍由模板生成，该模板由一个椭圆元素组成。其中定义的两个触发器在鼠标移到按钮或者是椭圆上时改变背景色，在 ControlTemplate.Triggers 中定义的触发器将背景色变为绿色，在 Style.Triggers 中变为蓝色。

该程序的运行结果如图 5-23 所示。

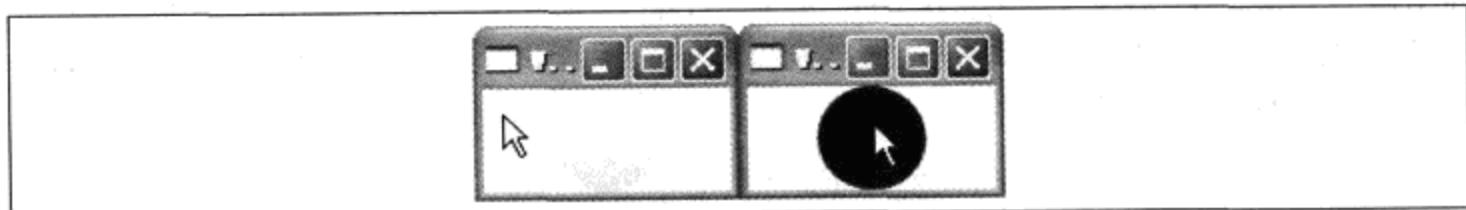


图 5-23 运行结果

当鼠标移动到椭圆上背景色变为蓝色，证明样式触发器优先级高于模板触发器。

同样删除样式触发器中的代码，运行结果如图 5-24 所示。

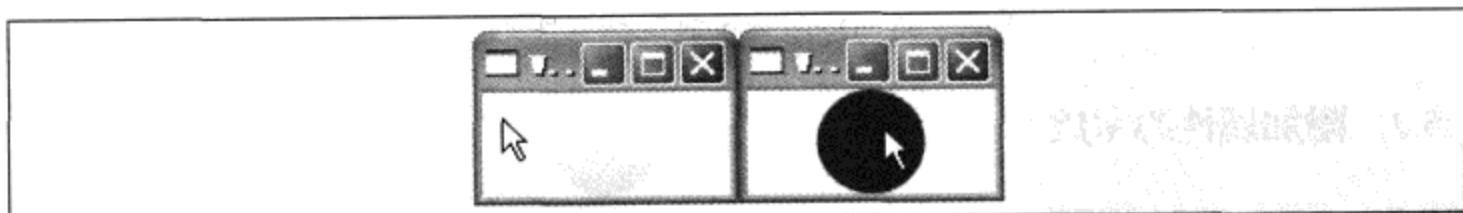


图 5-24 运行结果

5.5 附加属性和“等餐号”

学到此处，确实也算对依赖属性精通了。但是整个依赖属性里还有一样神秘的东西，那就是附加属性（Attached Property）。还是先打个比方来说说附加属性吧。

近来经济不景气，但是似乎丝毫没有影响国人的食欲。无论哪家餐馆都极其火爆，于是那些火爆得不行的餐馆，门口总会有一个小生给每个等待的顾客写上一个号码。然后依次叫号，叫到号的顾客就会被服务员带到不同的餐桌上。附加属性就好比这个小小的“等餐号”，明明是餐馆的属性，但是任何一个来餐馆的顾客都可以设置这个属性值（比如 1, 2, 3……），服务员再根据顾客的这个属性值安排合适的位置。

在 WPF 里，最典型的附加属性就是各种布局里面的属性，比如 Grid 的 Row, Column 或者是 DockPanel 里的 Dock 属性等。Row 和 Column 也像“等餐号”一样，让每个元素去设置其属性值，再由 Grid 按照属性值把元素安排在合适的位置。

5.5.1 如果没有附加属性

如果没有附加属性要处理布局的问题，以 DockPanel 为例，则可能为每个元素添加一个 Dock 属性，然后将其设置为 Left 或 Right 等，如：

```
Ctrl.Dock = Dock.Left; //Ctrl 是一个元素
```

但是如果这个 Ctrl 要加入到 Grid 里去，那么它又要添加 Row 和 Column 属性。这就好比顾客今天去“海底捞”，明天去“橘子洲头”，他必须得有所有他打算去的餐馆的“等餐号”。这样做显然是不合理的。

而 WPF 的做法和我们现实生活的做法一样，如果你打算在“海底捞”吃饭，那么这个时候顾客就拥有了“海底捞”的“等餐号”，不去吃饭则和这个属性完全没有关系。在 XAML 中，DockPanel 的 Dock 属性设置如代码 5-33 所示。

```
<Window x:Class="mumu_Button03.MainWindow"
.....
<DockPanel>
  <Button DockPanel.Dock ="Left">
    Left
  </Button>
</DockPanel>
.....
</Window>
```

代码 5-33 DockPanel 的 Dock 属性设置

5.5.2 附加属性的本质

附加属性实质上是一个依赖属性，与普通的依赖属性相比有以下不同。

- (1) 注册不再是通过 Register 方法注册，而是通过 RegisterAttached 方法注册。
- (2) 没有普通的.NET 属性包装器，而是通过 Get 和 Set 属性名来实现属性包装。
- (3) 没有普通的.NET 属性。

代码 5-34 是一个附加属性 IsBubbleSource 的示例：

```
public static readonly DependencyProperty IsBubbleSourceProperty =
DependencyProperty.RegisterAttached(
    "IsBubbleSource",
    typeof(Boolean),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(false, FrameworkPropertyMetadataOptions.
AffectsRender)
);
public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}
public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

代码 5-34 IsBubbleSource 附加属性

5.6 接下来做什么

关于依赖属性，在 MSDN WPF Fundamentals\Properties，如“依赖属性的安全性”及“集合类型的依赖属性”有较为详细的讲述。

依赖属性是 WPF 的重要特性，有关问题解答如下。

(1) 为什么在展示依赖属性属性值继承的示例中选择用按钮的字体大小属性，而不用背景色？

这是因为并不是所有依赖属性都支持属性值继承特性，背景色就不支持。而字号只有依赖属性的元数据 Inherits 属性为真，则该依赖属性才能支持属性值继承。MSDN 中关于按钮的背景色和字号依赖属性信息如图 5-25 和图 5-26 所示。FontSize 依赖属性一节中清楚地指出了 Inherits 为真。

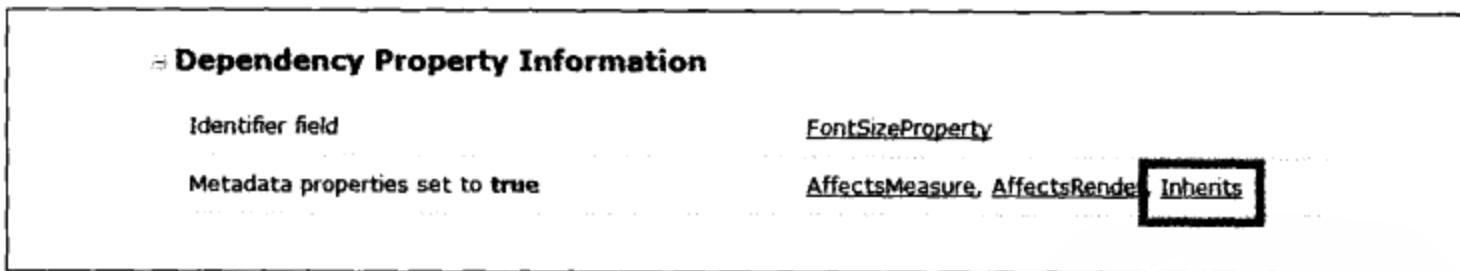


图 5-25 FontSize 元数据信息

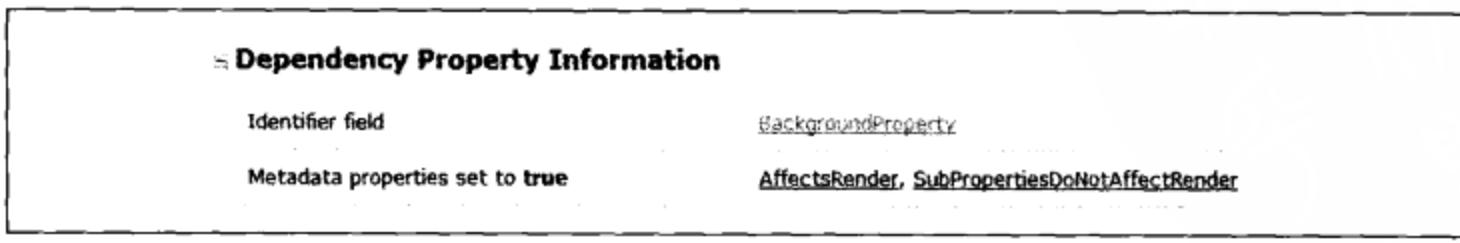


图 5-26 Background 元数据信息

(2) 在上例中元素树结构中 Grid 并没有字体大小属性，如图 5-27 所示，属性值继承是否可以从祖父传给孙子？

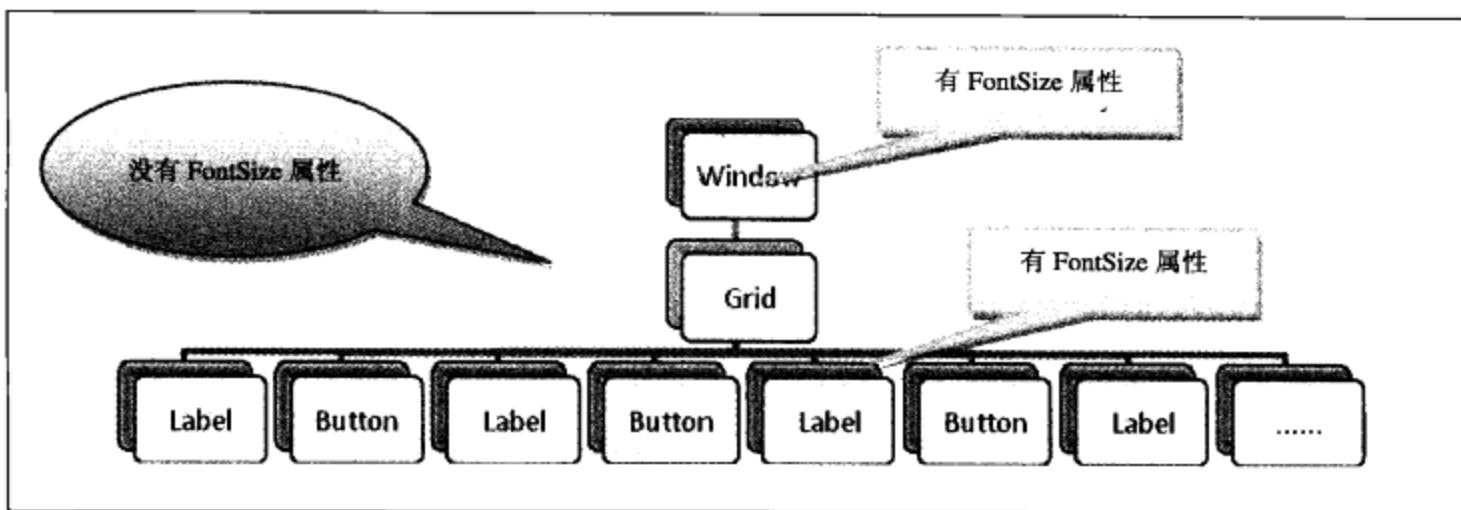


图 5-27 Grid 并没有字体大小属性

是的，显然 WPF 的继承机制并不是简单的树上的“父传子”，即这种继承可以让父类下面所有包含同一属性的子元素都得到继承。

接下来将是 WPF 的另外一个极其重要的特性，它足以和依赖属性并列，它就是路由事件。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 射雕英雄传》，“第五回 弯弓射雕”。
- [2] MSDN Library for Visual Studio 2008 SP1 WPF Architecture。
- [3] MSDN Library for Visual Studio 2008 SP1 Dependency Properties Overview。

路由事件——绝情谷底玉蜂飞

黄蓉凝目看去，只见那两只玉蜂双翅上也都有字。那六个字也是一模一样，右翅是“情谷底”；左翅是“我在绝”。黄蓉大奇，暗想：“造物虽奇，也绝造不出这样一批蜜蜂来之理。其中必有缘故。”……

黄蓉不答，只是轻轻念着：“情谷底，我在绝。情谷底，我在绝。”她念了几遍，随即省悟：“啊！那是‘我在绝情谷底’。是谁在绝情谷底啊？难道是襄儿？”心中怦怦乱跳……

——《神雕侠侣》，“第三十八回 生死茫茫”^[1]

这一段讲的是小龙女深陷绝情谷底，用花树上的细刺在玉蜂翅上刺下“我在绝情谷底”6个字，盼望玉蜂飞上之后能为人发现。结果蜂翅上的细字被周伯通发现，而被黄蓉隐约猜到了其中含义。

本章内容如下。

- (1) 从玉蜂说起，回顾.NET 事件模型。
- (2) 什么是路由事件？
- (3) CLR 事件足够完美，为什么还需要路由事件？
- (4) 言归正传，话路由事件。
- (5) 路由事件的实例。
- (6) 接下来做什么。

6.1 从玉蜂说起，回顾.NET 事件模型

木木熟悉神雕侠侣的故事，他根据“玉蜂传信”信手画了一幅有趣的图，如图 6-1 所示。

其实这幅“玉蜂传信图”暗合.NET 的事件模型，小龙女是事件的发布者，她发布了事件“我在绝情谷底”。老顽童和黄蓉是事件的订阅者，不过他并没有处理该事件，而黄蓉处理了事件，隐约能猜出其中含义。至于可怜的小杨过，则根本没有订阅事件，只是苦苦念叨“龙儿，龙儿，你在哪儿……”而玉蜂正是传递信息的事件，事件、事件的发布者和事件的订阅者构成了.NET 事件模型的 3 个角色。在.NET 中，一个事件用关键字 event 来表示。如代码 6-1 所示。

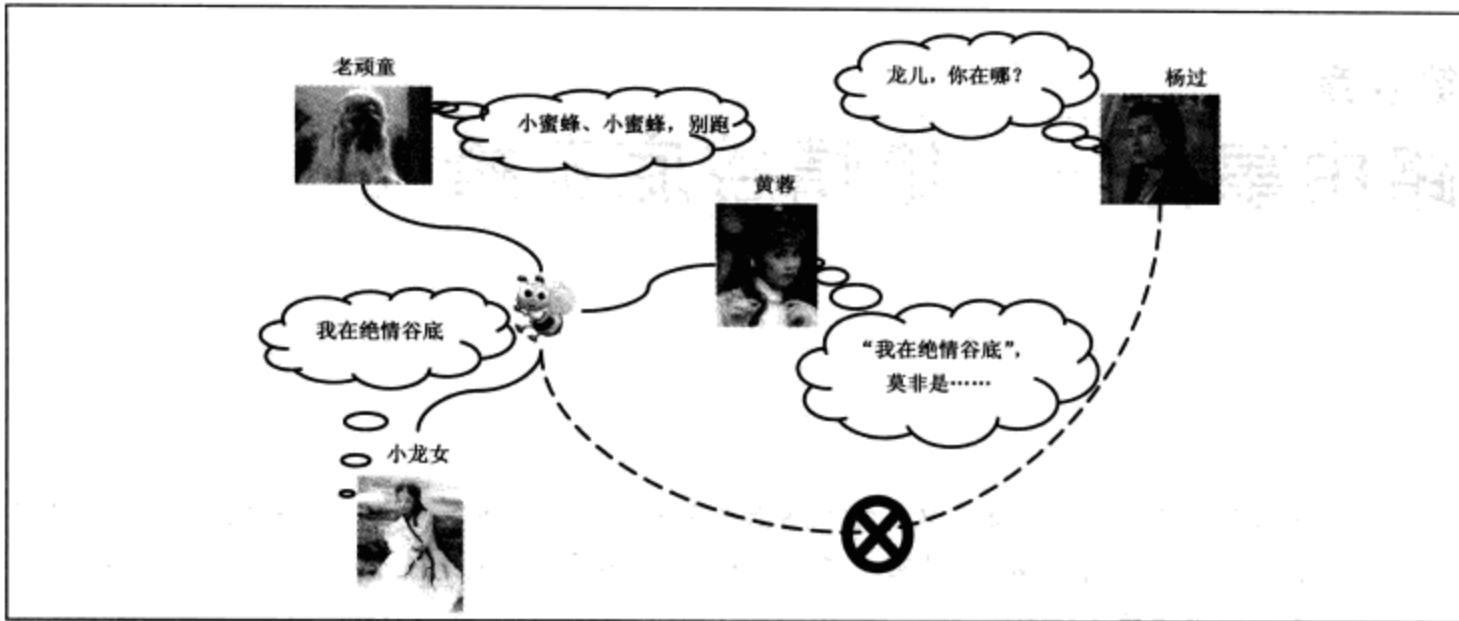


图 6-1 玉蜂传信图

```
public delegate void WhiteBee(string param); //声明了玉蜂的委托
// 小龙女类
class XiaoLongnv
{
    public event WhiteBee WhiteBeeEvent; //玉蜂事件
    public void OnFlyBee()
    {
        Console.WriteLine("小龙女在谷底日复一日地放着玉蜂，希望杨过有一天能看到.....");
        WhiteBeeEvent(msg);
    }
    private string msg = "我在绝情谷底";
}

// 老顽童类
class LaoWantong
{
    public void ProcessBeeLetter(string msg)
    {
        Console.WriteLine("老顽童：小蜜蜂、小蜜蜂，别跑");
    }
}

// 黄蓉类
class Huangrong
{
    public void ProcessBeeLetter(string msg)
    {
        Console.WriteLine("黄蓉：'{0}'，莫非.....",msg);
    }
}

// 杨过类
class YangGuo
{
    public void ProcessBeeLetter(string msg)
    {
        Console.WriteLine("杨过：'{0}'，我一定会找她！", msg);
    }
}
```

```

public void Sign()
{
    Console.WriteLine("杨过叹息：龙儿，你在哪儿……");
}
static void Main(string[] args)
{
    // 第1步 人物介绍
    XiaoLongnv longnv = new XiaoLongnv(); //小龙女
    LaoWantong wantong = new LaoWantong(); //老顽童
    Huangrong rong = new Huangrong(); //黄蓉
    YangGuo guo = new YangGuo(); //杨过

    // 第2步 订阅事件，唯独没有订阅杨过的 ProcessBeeLetter;
    longnv.WhiteBeeEvent += wantong.ProcessBeeLetter;
    longnv.WhiteBeeEvent += rong.ProcessBeeLetter;
    // longnv.WhiteBeeEvent += guo.ProcessBeeLetter; //杨过没有订阅小龙女的玉蜂事件

    // 第3步 小龙女玉蜂传信
    longnv.OnFlyBee();

    // 第4步 杨过叹息
    guo.Sign();
}

```

代码 6-1 详见 `mumu_whitebee` 工程

程序运行结果如图 6-2 所示。

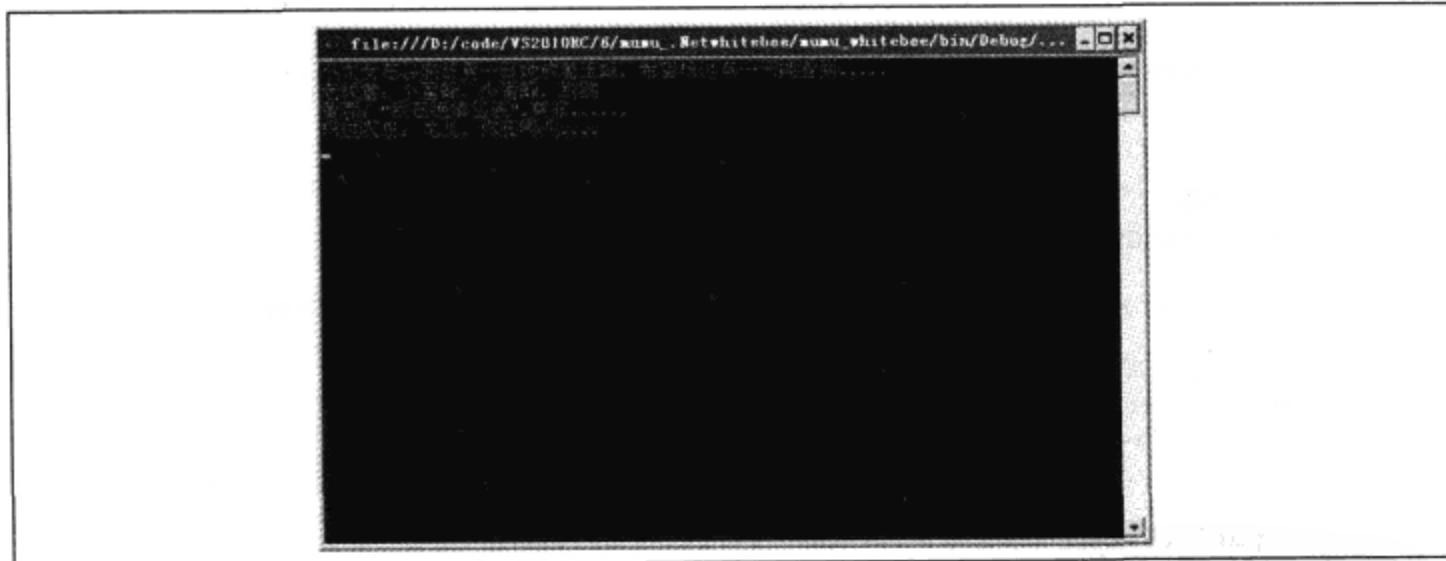


图 6-2 运行结果

不过这种事件看起来不像.NET 的事件，于是木木改写为如代码 6-2 所示的代码。

```

// 新增一个 WhiteBeeEventArgs 类
public class WhiteBeeEventArgs : EventArgs
{
    public readonly string _msg;
    public WhiteBeeEventArgs(string msg)
    {
        this._msg = msg;
    }
}

```

```

}

public delegate void WhiteBeeEventHandler(object sender, WhiteBeeEventArgs e); //声明了玉蜂的委托
// 小龙女类
class XiaoLongnv
{
    public event WhiteBeeEventHandler WhiteBeeEvent; //玉蜂事件
    public void OnFlyBee()
    {
        Console.WriteLine("小龙女在谷底日复一日地放着玉蜂，希望杨过有一天能看到……");
        WhiteBeeEventArgs args = new WhiteBeeEventArgs(msg);
        WhiteBeeEvent(this, args);
    }
    private string msg = "我在绝情谷底";
}

// 老顽童类
class LaoWantong
{
    public void ProcessBeeLetter(object sender, WhiteBeeEventArgs e)
    {
        Console.WriteLine("老顽童：小蜜蜂、小蜜蜂，别跑");
    }
}

// 黄蓉类
class Huangrong
{
    public void ProcessBeeLetter(object sender, WhiteBeeEventArgs e)
    {
        Console.WriteLine("黄蓉：" + "{0}" + "莫非.....", e._msg);
    }
}

// 杨过类
class YangGuo
{
    public void ProcessBeeLetter(object sender, WhiteBeeEventArgs e)
    {
        // 真的是龙儿吗？
        XiaoLongnv longnv = sender as XiaoLongnv;
        if (longnv != null)
            Console.WriteLine("杨过：" + "{0}" + "，我一定会找她！", e._msg);
    }
    public void Sign()
    {
        Console.WriteLine("杨过叹息：龙儿，你在哪儿……");
    }
}

static void Main(string[] args)
{
    // 第1步 人物介绍
    XiaoLongnv longnv = new XiaoLongnv(); //小龙女
    LaoWantong wantong = new LaoWantong(); //老顽童
    Huangrong rong = new Huangrong(); //黄蓉
    YangGuo guo = new YangGuo(); //杨过
}

```

```

// 第2步 订阅事件,唯独没有订阅杨过的 ProcessBeeLetter;
longnv.WhiteBeeEvent += wantong.ProcessBeeLetter;
longnv.WhiteBeeEvent += rong.ProcessBeeLetter;
// longnv.WhiteBeeEvent += guo.ProcessBeeLetter; //杨过没有订阅小龙女的玉蜂事件

// 第3步 小龙女玉蜂传信
longnv.OnFlyBee();

// 第4步 杨过叹息
guo.Sign();
}

```

代码 6-2 详见 `mumu_netwhitebee` 工程

上述代码中加粗部分是经过修改的，可以归纳为如下几点。

(1) 委托类型的名称修改为以 `EventHandler` 结束，原型有一个 `void` 返回值并接受两个输入参数，即一个 `Object` 类型和一个 `EventArgs` 类型（或继承自 `EventArgs`）。

修改前	<code>public delegate void WhiteBee(string param);</code>
修改后	<code>public delegate void WhiteBeeEventHandler(object sender,WhiteBeeEventArgs e);</code>

(2) 事件的命名为委托去掉 `EventHandler` 之后剩余的部分。

修改前	<code>public event WhiteBee WhiteBeeEvent;</code>
修改后	<code>public event WhiteBeeEventHandler WhiteBee;</code>

(3) 继承自 `EventArgs` 的类型应该以 `EventArgs` 结尾。

修改前	无事件参数
修改后	<pre>public class WhiteBeeEventArgs : EventArgs { public readonly string _msg; public WhiteBeeEventArgs(string msg) { this._msg = msg; } }</pre>

这样修改的目的不仅仅符合.NET 规范，而且带来了更大的灵活性。例如，如果杨过收到了小龙女的玉峰传信，则可以通过 `sender` 参数来判断是否真是小龙女，甚至还可以了解更多有关的细节：

```

public void ProcessBeeLetter(object sender, WhiteBeeEventArgs e)
{
    // 真的是龙儿吗?
    XiaoLongnv longnv = sender as XiaoLongnv;
    if(longnv != null)
        Console.WriteLine("杨过: \"{}\", 我一定会找她! ", e._msg);
}

```

6.2 路由事件的定义

什么是路由事件呢？MSDN 从功能和实现两种视角给出了路由事件的定义：

“Functional definition: A routed event is a type of event that can invoke handlers on multiple listeners in an element tree, rather than just on the object that raised the event.”

“Implementation definition: A routed event is a CLR event that is backed by an instance of the RoutedEvent class and is processed by the Windows Presentation Foundation (WPF) event system.”

以 Button 的 Click 事件为例，该事件是个路由事件。可以通过 Reflector 查看 ButtonBase 的源码，如代码 6-3 所示。

```
public abstract class ButtonBase : ContentControl, ICommandSource
{
    // 路由事件的定义
    public static readonly RoutedEvent ClickEvent;

    // 传统的事件包装器
    public event RoutedEventHandler Click
    {
        add
        {
            base.AddHandler(ClickEvent, value);
        }
        remove
        {
            base.RemoveHandler(ClickEvent, value);
        }
    }
    // 事件的注册
    static ButtonBase()
    {
        ClickEvent = EventManager.RegisterRoutedEvent("Click",
RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));
        ....
    }
    ....
}
```

代码 6-3 ButtonBase 的 Click 事件

同依赖属性一样，路由事件也需要注册，不同的是使用 EventManager.RegisterRoutedEvent 方法。

同依赖属性，用户不会直接使用路由事件，而是使用传统的 CLR 事件。有两种方式关联事件及其处理函数，在代码中，仍然按照原来的方法关联和解除关联事件处理函数（+= 和 -=），如代码 6-4 所示。

```
Button b2 = new Button();
// 关联事件及其处理函数
b2.Click += new RoutedEventHandler(Onb2Click);

//事件处理函数
void Onb2Click(object sender, RoutedEventArgs e)
```

```
{  
    //logic to handle the Click event  
}
```

代码 6-4 关联事件和事件处理函数

WPF 实现的机制已经发生变化，传统的 CLR 事件关联和解除关联处理函数均通过委托来实现，而路由事件则通过 AddHandler 和 RemoveHandler 方法实现。二者在 UIElement 中定义，多数 WPF 的类需要继承自该类，如表 6-1 所示。

表 6-1 CLR 事件和路由事件关联和解除关联事件处理函数的差别

传统的 CLR 事件+=的背后	Delegate.Combine(.....)
路由事件+=的背后	AddHandler(.....)
传统的 CLR 事件-=的背后	Delegate.Remove(.....)
路由事件-=的背后	RemoveHandler(.....)

上面的代码也可以改写为代码 6-5：

```
Button b2 = new Button();  
// 关联事件及其处理函数  
b2.AddHandler(Button.ClickEvent, new RoutedEventHandler(Onb2Click));  
  
// 事件处理函数  
void Onb2Click(object sender, RoutedEventArgs e)  
{  
    //logic to handle the Click event  
}
```

代码 6-5 关联事件和事件处理函数

在 XAML 中，事件和处理函数这样关联，如代码 6-6 所示。

```
<Button Click=" Onb2Click ">button</Button>
```

代码 6-6 在 XAML 中关联事件和事件处理函数

传统的事件触发往往直接调用其委托（因为事件的本质是委托），而路由事件则通过一个 RaiseEvent 方法触发，调用该方法后所有关联该事件的对象都会得到通知。在 ButtonBase 中即有代码 6-7：

```
RoutedEventArgs e = new RoutedEventArgs(ClickEvent, this);  
base.RaiseEvent(e);
```

代码 6-7 ButtonBase 里触发事件

路由事件通过 EventManager.RegisterRoutedEvent 方法注册；通过 AddHandler 和 RemoveHandler 来关联和解除关联的事件处理函数；通过 RaiseEvent 方法来触发事件；通过传统的 CLR 事件封装后供用户调用，使得用户如同使用传统的 CLR 事件一样使用路由事件。

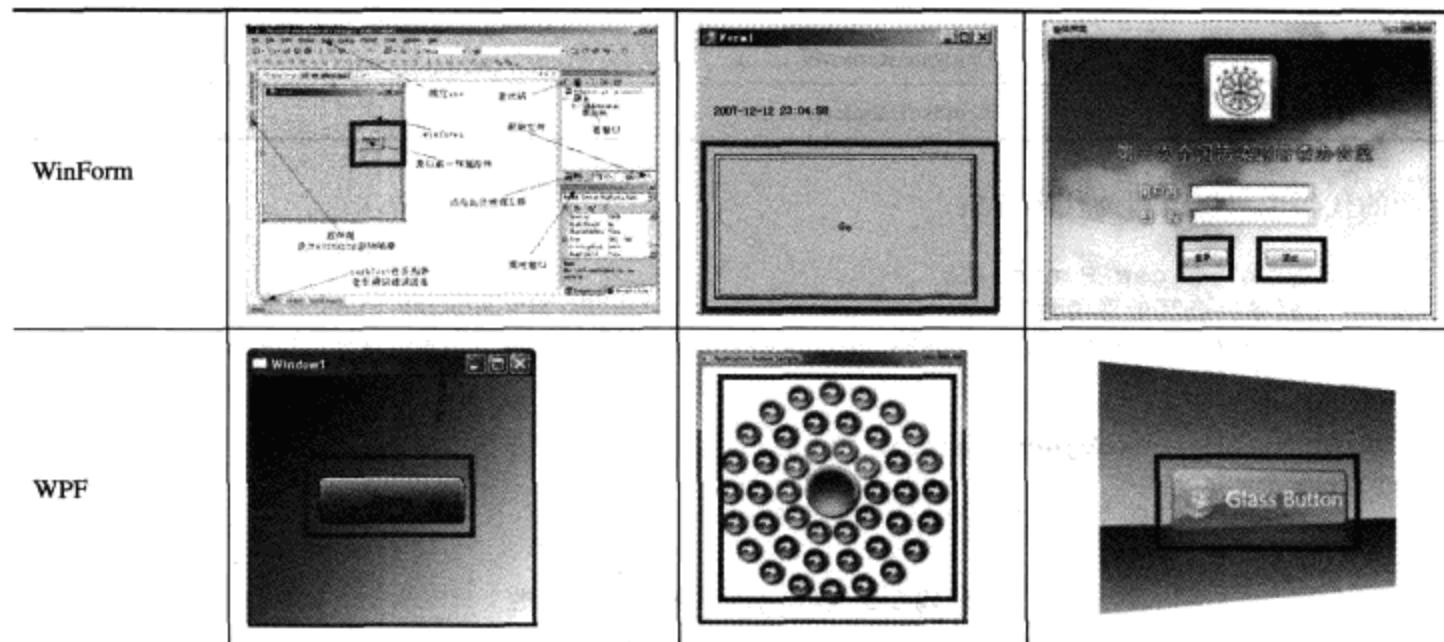
6.3 路由事件的作用

虽然木木已经明白了什么是路由事件，但是头脑中仍有一个很大的疑问：WPF 为什么引入路由事件，难道 CLR 事件就不能满足应用吗？这个还是得从 WPF 的界面元素的特点上说起。

以按钮为例，在 WinForm 时代一个按钮就是一个按钮；在 WPF 中一个按钮可以是其他部件。

表 6-2 所示为利用 Google 图片搜索功能分别用 WinForm Button 和 WPF Button 关键词搜索任意尺寸和任意类型的前几位图片（2009 年 7 月 4 日搜索结果，用框标识 Button）。

表 6-2 互联网上的 WinForm 和 WPF 按钮



最直观的反应是 WPF 的按钮要比 WinForm 来得炫目，但绝不仅于此。WPF 中的任何一个界面元素都可以任意嵌套，并且装配极其容易。如代码 6-8 所示。

```
<Button Margin="50" Background="AliceBlue" BorderBrush="Black"
BorderThickness="2">
    <StackPanel>
        <TextBlock Margin="3" >
            Image and picture Button</TextBlock>
        <Image Source="happyface.jpg" Stretch="None" />
        <TextBlock Margin="3" >
            Courtesy of the StackPanel</TextBlock>
    </StackPanel>
</Button>
```

代码 6-8 一个嵌套按钮的例子

程序运行结果和按钮的结构如图 6-3 所示。

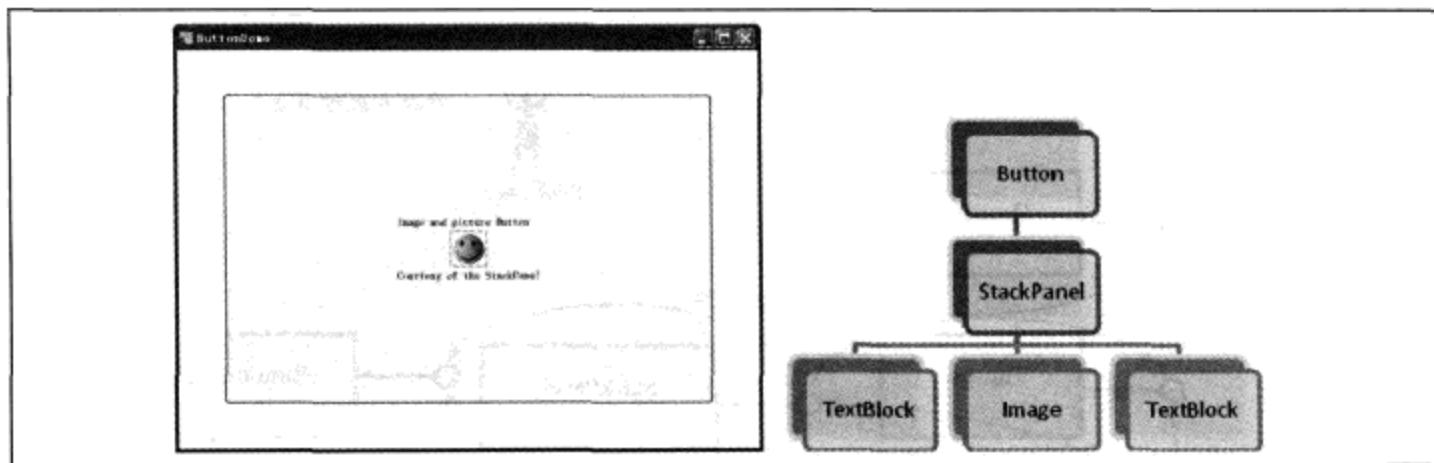


图 6-3 程序运行结果和按钮的结构

如果在图片（Image）上单击一下，应该哪个元素去响应这个点击事件呢，是图片？还是按钮呢？理想的情况当然是无论鼠标点击到图片，还是文本块（TextBlock），都应该由按钮来处理。使用传统的 CLR 事件，程序员势必要对所有元素的事件响应处理函数重写，而且每个事件处理函数最后都要调用按钮的事件处理函数，如图 6-4 所示。

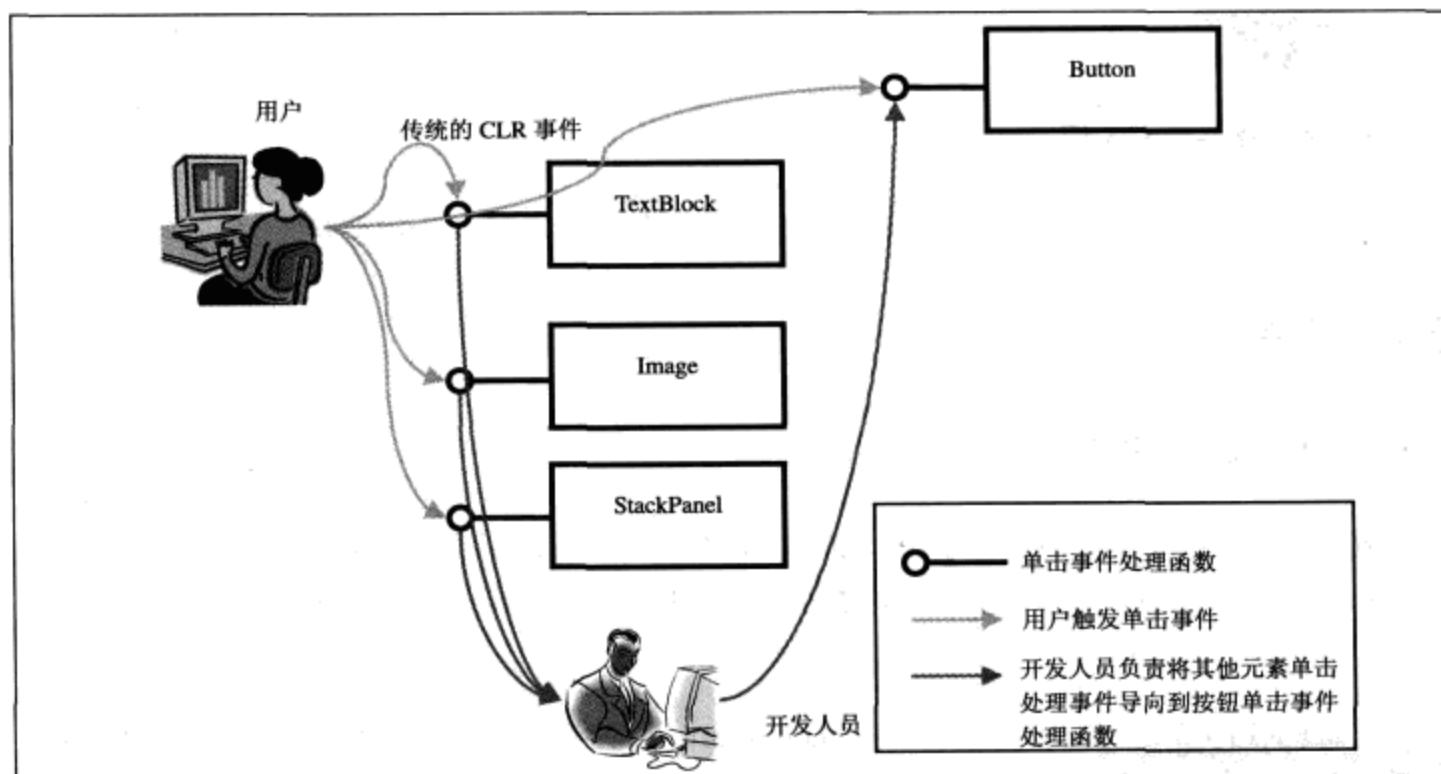


图 6-4 传统的 CLR 事件处理流程

路由事件则完全不同，它可以向上游走（Bubble）或向下沉底（Tunnel）。在这个例子中，用户单击到文本块（TextBlock）或者图片（Image）时事件可以经由 TextBlock/Image——StackPanel——Button 一路上浮，游走到按钮的事件处理函数中，而开发人员不必做任何工作，如图 6-5 所示。

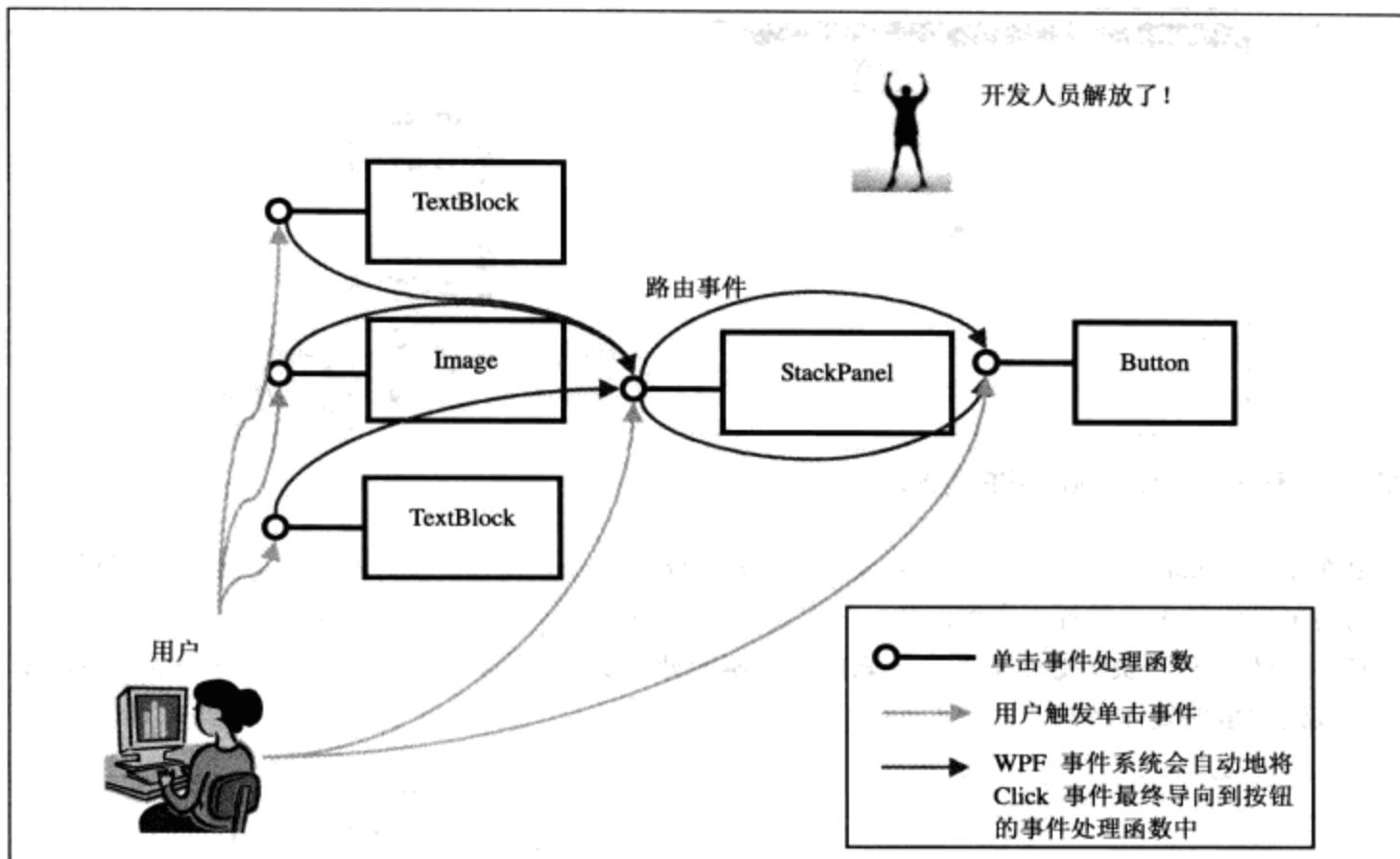


图 6-5 路由事件处理流程

路由事件的引入既能满足控件的任意组合，又能保证控件 Hit-test 的行为的完整性，何乐而不为呢？

6.4 路由事件

6.4.1 识别路由事件

和依赖属性一样，如果一个事件是路由事件，则在 MSDN 文档中会有路由事件（Routed Event Information）一节描述其路由信息。而普通的 CLR 事件（如 `UIElement.IsVisibleChanged`）则没有这一节信息，如图 6-6 所示。

图 6-6 描述的路由信息主要包括唯一标识 `ClickEvent`、事件委托形式 `RoutedEventHandler`，以及路由策略，`Click` 事件的路由策略是 `Bubbling`。

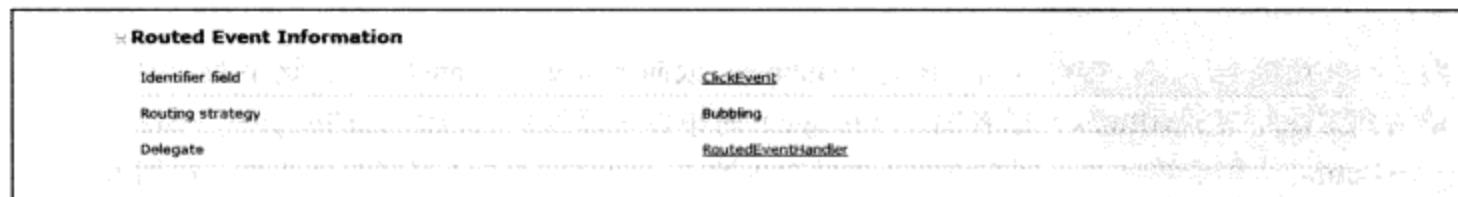


图 6-6 ButtonBase.Click 事件页面中包含 Routed Event Information

6.4.2 路由事件的旅行

1. 路由事件的旅行策略

一个人在外旅行，沿途会休息，会遇到各种各样好玩的事情。路由事件的旅行相对要简单很多。归纳起来在路由事件的旅行当中，一般只出现两种角色：一是事件源，由其触发事件，是路由事件的起点；二是事件监听者，通常针对监听的事件有一个相应的事件处理函数。当路由事件经过事件监听者，就好比经过一个客栈，要做短暂的停留，由事件处理函数来处理该事件。

路由事件的策略有如下 3 种。

- (1) Bubbling：事件从事件源出发一路上溯直到根节点，很多路由事件使用该策略。
- (2) Direct：事件从事件源出发，围绕事件源转一圈结束。
- (3) Tunneling：事件源触发事件后，事件从根节点出发下沉直到事件源。

上述三种策略都只能看做是路由事件的一个旅行计划，实际上当路由事件开始旅行的时候，由于事件监听者的干预，它的旅行计划会有所改变。这一点，实际上也很好理解，我们每次出游的时候都会有个预定的路线，但是事实上什么飞机晚点、黑店勒索、山洪暴发等诸如此类的事情会影响我们的预定路线，最终导致实际路线和预定路线有所偏差。

2. 改变旅行策略因素——事件处理函数

一个最基本的路由事件处理函数的原型如代码 6-9 所示。

```
public delegate void RoutedEventHandler( Object sender, RoutedEventArgs e)
```

代码 6-9 路由事件处理函数原型

事件处理函数之间有微小差异，如鼠标事件的处理函数原型如代码 6-10 所示。

```
public delegate void MouseEventHandler(Object sender, MouseEventArgs e)
```

代码 6-10 鼠标事件的处理函数原型

这种事件处理函数有如下两个特点。

- (1) 返回原型为 void。
- (2) 有两个参数，第 1 个是一个 Object 类型的对象，表示拥有该事件处理函数的对象；第 2 个是 RoutedEventArgs 或者是 RoutedEventArgs 的派生类，带有其路由事件的信息。

RoutedEventArgs 结构包括 4 个成员变量，如表 6-3 所示。

表 6-3 RoutedEventArgs 结构

名称	描述
Source	表明触发事件的源，如当键盘事件发生时，触发事件的源是当前获得焦点的对象；当鼠标事件发生时，触发事件的源是鼠标所在的最上层对象
OriginalSource	表明触发事件的源，一般来说 OriginalSource 和 Source 相同，区别在于 Source 表示逻辑树上的元素；OriginalSource 是可视树中的元素。如单击窗口的边框，Source 为 Window；OriginalSource 为 Border
RoutedEventArgs	路由事件对象
Handled	布尔值，为 true，表示该事件已处理，这样可以停止路由事件

Handled 属性是改变路由事件旅行的“元凶”。一旦在某个事件处理函数中将 Handled 的值设置为 true，路由事件就停止传递。如：

```
private void ***_Click(object sender, RoutedEventArgs e)
{
    ...
    e.Handled = true;
}
```

一个事件被标记为处理，事件处理函数则不可处理该事件。但是也有例外，WPF 还提供了一种机制，即使事件被标记为处理，事件处理函数仍然可以处理，但是关联事件及其处理函数需要稍做处理。AddHandler 重载了两个方法，其中之一如下所示，需要将第 3 个参数设置为 true：

```
public void AddHandler(RoutedEvent routedEvent, Delegate handler, bool handledEventsToo)
```

换句话说，因事件标识为处理而终止只是一种假象，路由事件的旅行仍然在继续，只不过普通的事件处理函数无法处理它。

3. 改变旅行策略因素之二——类和实例事件处理函数

事件处理函数有两种类型：一是前面所说的普通事件处理函数，称为“实例事件处理函数”（Instance Handlers）；二是通过 EventManager.RegisterClassHandler 方法将一个事件处理函数和一个类关联起来，这种事件处理函数，称为“类事件处理函数”（Class Handlers），其优先权高于前者。也就是说事件在旅行时，会先光临类事件处理函数，然后再光临实例事件处理函数。

4. 路由事件的旅行图

木木做了一个奇怪的梦，醒后记录了梦里的内容。

（1）主角是个路由事件，渴望旅行。

（2）路由事件有 3 种旅行计划：一是北上观北国风光千里冰封（Bubbling）；二是南下赏江南秀色，小桥流水（Tunneling）；三是同城一日游（Direct）。

（3）路由事件旅行线路中有 4 种不同的客栈，即普通客栈（响应事件，但是不会把事件标记为“已

处理”的实例事件处理函数)、黑心客栈(响应事件,而且会把事件身上所带的钱和衣物全部抢走。标记事件为“处理”,以至几乎所有的客栈都不会再让事件住宿)、政府指定客栈(类事件处理函数,事件必须优先在该函数响应)和慈善客栈(即使事件已被标记为处理,一无所有,该客栈仍然会欢迎事件住宿)。

(4) 事件在旅行时会有两种状态:未处理和已处理状态。事件默认时都是未处理状态,这个时候的事件就好比一个大款,每个客栈看到事件,都会热情地请事件留宿,而“大款”事件也从来不拒绝客栈。但是事件如果一不小心留宿黑心客栈,黑心客栈会抢走它所有的钱和衣物(标记为“处理”状态),这个时候再一路旅行,由于衣衫不整,事件只能走隐蔽的路线,所有的客栈也都会很冷漠地对待它,只有慈善客栈仍然欢迎事件留宿。

图 6-7 所示为路由事件的旅行北上图(Bubbling 策略)。

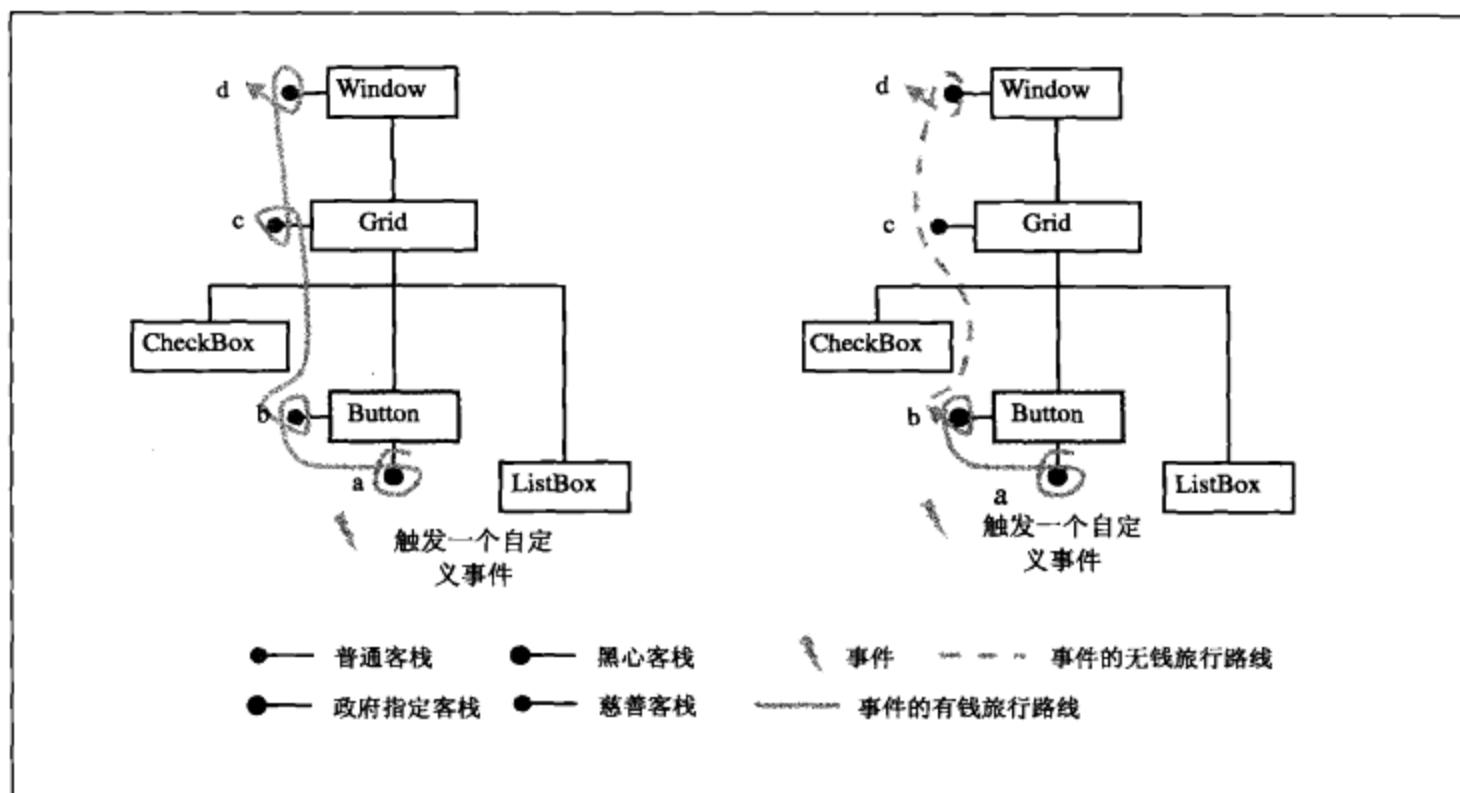


图 6-7 路由事件的旅行北上图

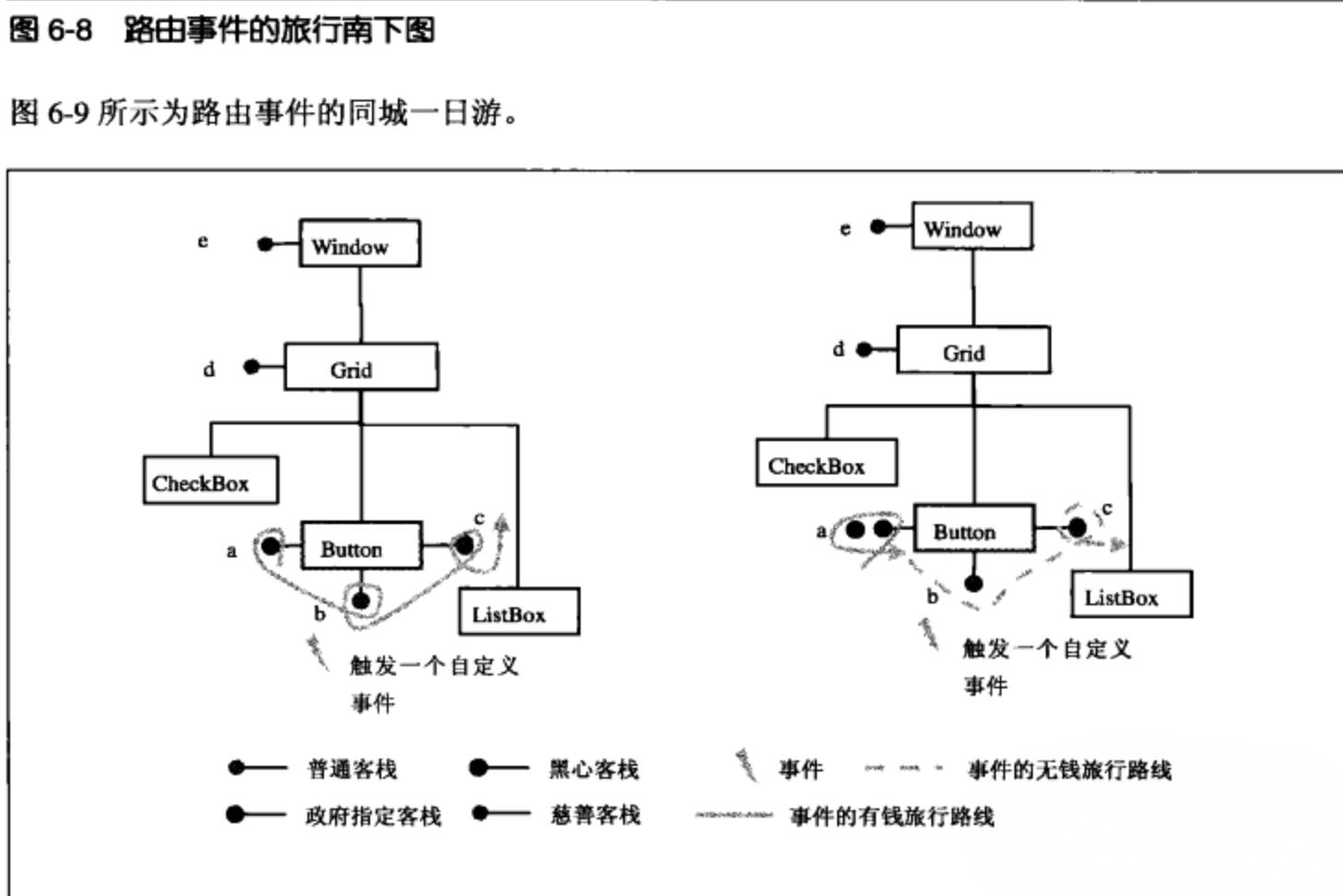
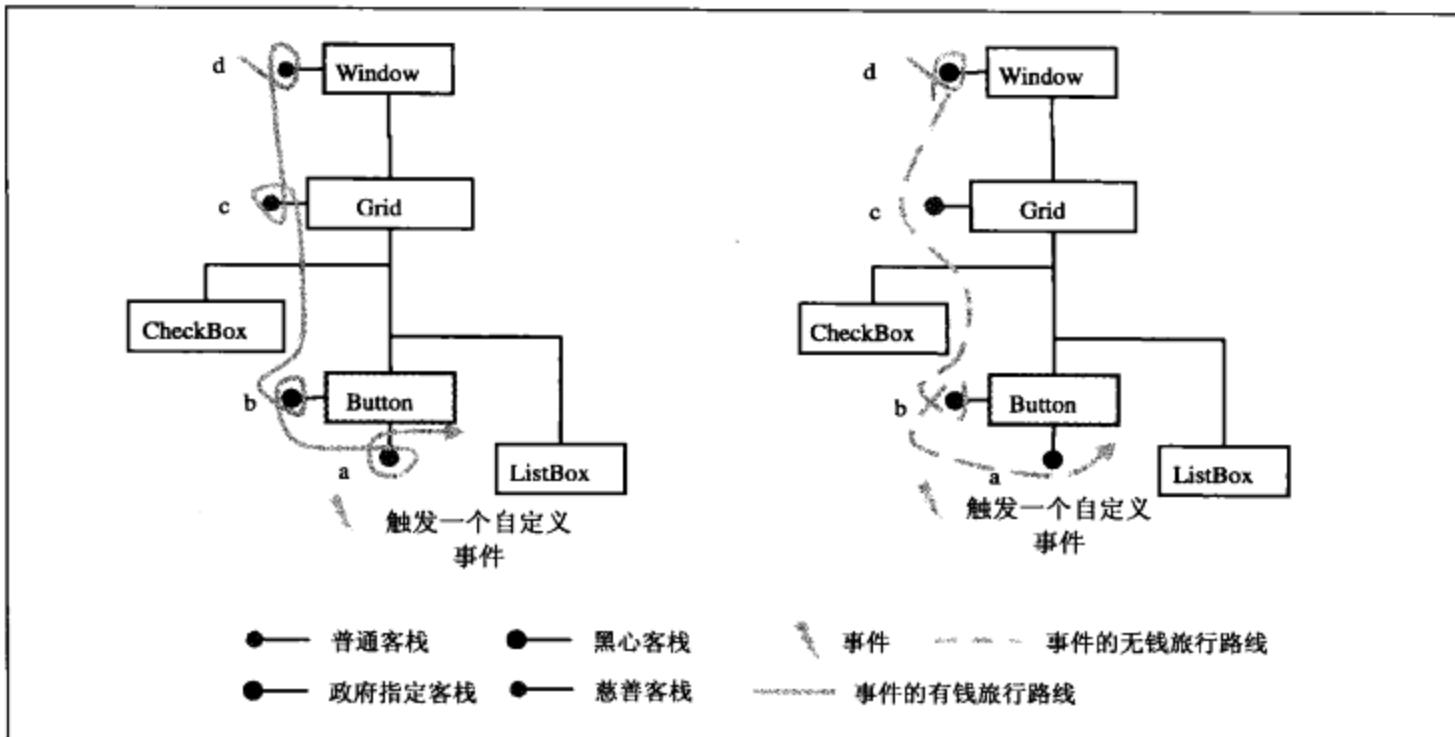
路由事件北上旅行之一从 Button 出发,首先在政府指定客栈停留(a),然后在 Button 的普通客栈(b)停留。沿途从 c 到 d 都是普通客栈,平安到达 Window。

路由事件北上旅行之二从 Button 出发,仍然在政府指定客栈停留(a),然后在 b 客栈停留。结果 b 是一个黑店,于是路由事件只好落荒而逃。客栈 c 不接待它,最后只有 d 这个慈善客栈接待了它。

图 6-8 所示为路由事件的旅行南下图。

路由事件旅行南下之一非常顺利,从 Window 的普通客栈(d)一路南下到 Button。

路由事件旅行南下之二出门不顺利,遇上黑店(d)。于是经过普通客栈(c)不能留宿,终于经过慈善客栈(b)留宿结束了旅行。



6.5 路由事件示例

自定义一个路由事件，名为“CustomClickEvent”。单击按钮时，这个事件就会触发为 Window, Grid 和 Button 装配不同的事件处理函数。然后单击按钮，观察路由事件的路由，如图 6-10 所示。

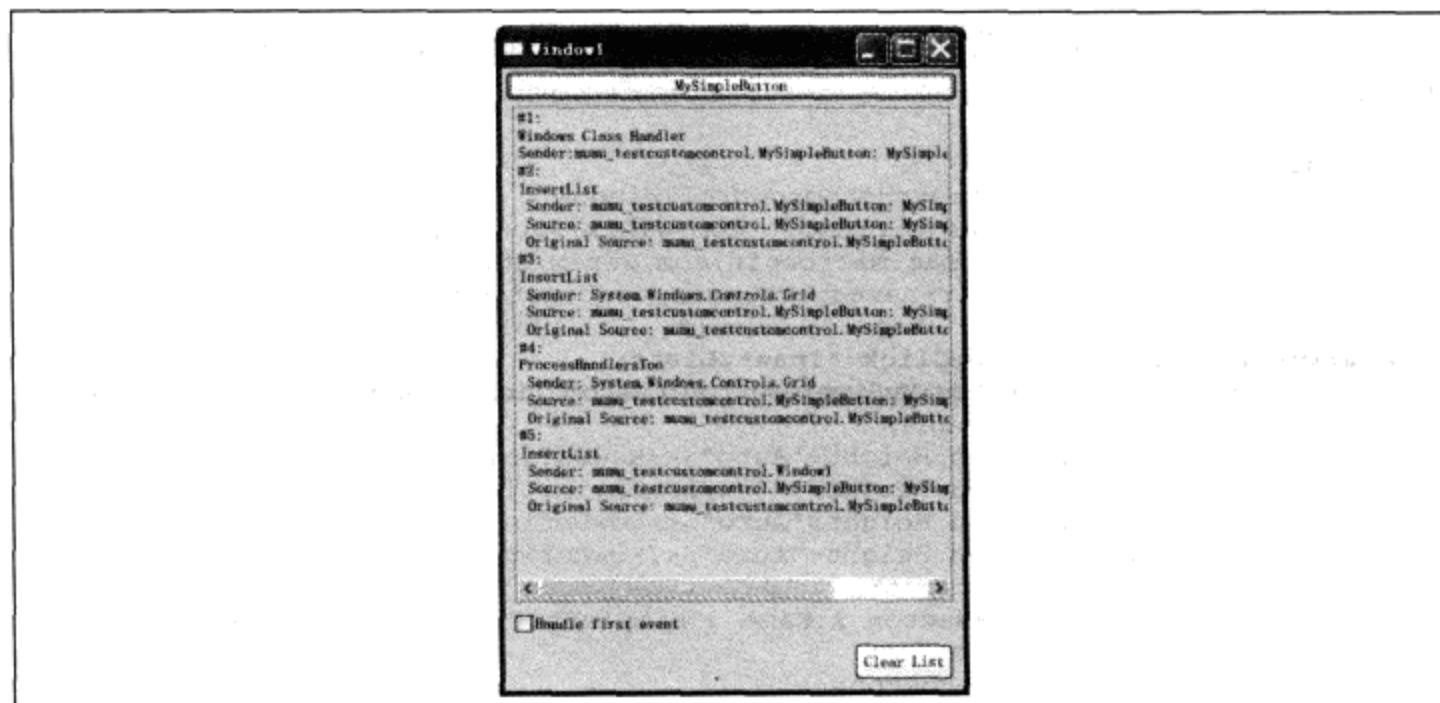


图 6-10 自定义路由事件

(1) 需要继承一个按钮类，然后自定义 CustomClickEvent 的路由事件。其路由策略为 Bubble，如代码 6-11 所示。

```
MySimpleButton.cs
public class MySimpleButton : Button
{
    static MySimpleButton()
    {
    }
    // 创建和注册该事件，该事件路由策略为 Bubble
    public static readonly RoutedEvent CustomClickEvent =
EventManager.RegisterRoutedEvent(
    "CustomClick", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
    typeof(MySimpleButton));
    // CLR 事件的包装器
    public event RoutedEventHandler CustomClick
    {
        add { AddHandler(CustomClickEvent, value); }
        remove { RemoveHandler(CustomClickEvent, value); }
    }
    // 触发 CustomClickEvent
    void RaiseCustomClickEvent()
    {
        RoutedEventArgs newEventArgs = new RoutedEventArgs(MySimpleButton.
CustomClickEvent);
        RaiseEvent(newEventArgs);
    }
    // OnClick 触发 CustomClickEvent
```

```

protected override void OnClick()
{
    RaiseCustomClickEvent();
}
}

```

代码 6-11 自定义一个路由事件

(2) 设计个应用程序的界面, 为 Window、Grid 和 MySimpleButton 关联相应的事件处理函数, 如代码 6-12 所示。

```

Window1.xaml
<Window x:Class="mumu_testcustomcontrol.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom ="clr-namespace:mumu_testcustomcontrol"
    Title="Window1"           Name="window1"           Height="300"           Width="300"
custom:MySimpleButton.CustomButton="InsertList">
    <Grid Margin="3" custom:MySimpleButton.CustomButton="InsertList" Name="grid1" >
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <custom:MySimpleButton x:Name = "simpleBtn" CustomClick="InsertList">
            MySimpleButton
        </custom:MySimpleButton>
        <ListBox Margin="5" Name="lstMessages" Grid.Row="1"></ListBox>
        <CheckBox Margin="5" Grid.Row="2" Name="chkHandle">Handle first
event</CheckBox>
        <Button Grid.Row="3" HorizontalAlignment="Right" Margin="5"
Padding="3" Click="cmdClear_Click">Clear List</Button>
    </Grid>
</Window>

Window1.xaml.cs
public partial class Window1 : Window
{
    public Window1()
    {
        InitializeComponent();
    }

    protected int eventCounter = 0;

    private void InsertList(object sender, RoutedEventArgs e)
    {
        eventCounter++;
        string message = "#" + eventCounter.ToString() + ":\r\n" +
            "InsertList\r\n" +
            " Sender: " + sender.ToString() + "\r\n" +
            " Source: " + e.Source + "\r\n" +
            " Original Source: " + e.OriginalSource;
        lstMessages.Items.Add(message);
        e.Handled = (bool)chkHandle.IsChecked;
    }

    private void cmdClear_Click(object sender, RoutedEventArgs e)
    {
        eventCounter = 0;
    }
}

```

```
        lstMessages.Items.Clear();
    }
}
```

代码 6-12 Window1.xaml 和 Window1.xaml.cs 文件

(3) 添加特殊的事件处理函数，为 MySimpleButton 添加一个指定客栈——类事件处理函数 CustomClickClassHandler（代码①）。为了通知外部窗口，把路由事件旅游的信息输出在 ListBox 列表中，因此需要添加一个普通的 CLR 事件 ClassHandlerProcessed（代码②）。在 Windows 的 Load 事件中为 Grid 添加一个慈善客栈 ProcessHandlersToo（代码③），如代码 6-13 所示。

```
MySimpleButton.cs
public class MySimpleButton : Button
{
    static MySimpleButton()
    {
        ①          // 将 CustomClickEvent 和一个 Class Handler 关联起来
        EventManager.RegisterClassHandler(typeof(MySimpleButton),
CustomClickEvent, new RoutedEventHandler(CustomClickClassHandler), false);
    }

    ②          // 普通 CLR 事件
    public event EventHandler ClassHandlerProcessed;
    public static void CustomClickClassHandler(object sender,
RoutedEventArgs e)
    {
        MySimpleButton simpleBtn = sender as MySimpleButton;
        EventArgs args = new EventArgs();
        simpleBtn.ClassHandlerProcessed(simpleBtn, args);
    }
}

Window1.xaml.cs
public partial class Window1 : Window
{
    public Window1()
    {
        InitializeComponent();
        // MySimpleButton 的类事件处理函数处理过 Window 能够得到通知
        this.simpleBtn.ClassHandlerProcessed += new
        EventHandler(simpleBtn_RaisedClass);
    }

    private void simpleBtn_RaisedClass(object sender, EventArgs e)
    {
        eventCounter++;
        string message = "#" + eventCounter.ToString() + ":\r\n" +
        "Windows Class Handler\r\n" + "Sender:" + sender.ToString();
        lstMessages.Items.Add(message);
    }

    private void ProcessHandlersToo(Object sender, RoutedEventArgs e)
    {
        eventCounter++;
        string message = "#" + eventCounter.ToString() + ":\r\n" +
        "ProcessHandlersToo\r\n" +
        " Sender: " + sender.ToString() + "\r\n" +
        " Source: " + e.Source + "\r\n" +
```

```
        " Original Source: " + e.OriginalSource;
        lstMessages.Items.Add(message);
    }
③     private void Window_Loaded(object sender, RoutedEventArgs e)
{
    grid1.AddHandler(MySimpleButton.CustomButtonEvent,      new
RoutedEventHandler(ProcessHandlersToo), true);
}
}
```

代码 6-13 添加特殊的事件处理函数

通过查看列表中的信息观察，就可以印证路由事件的旅行北上图。

6.6 接下来做什么

事件一直是个挥之不去的话题。如果从纵向的角度去了解路由事件的话，那么它的历史发展轨迹如下：

- (1) 第 1 阶段：“上古” Win32 时期，回调函数和函数指针。
- (2) 第 2 阶段：“上古” MFC 时期，MFC 的消息映射和类的函数指针。
- (3) 第 3 阶段：.NET 时期或者说 WinForm 时期，事件和委托。
- (4) 第 4 阶段：当代，WPF 时期及路由事件。

如果希望了解路由事件的演变，请参考笔者的博文《路由事件的演变史》；如果希望了解委托和事件，请参考如下两篇关于委托和事件的博文。

- (1) “2007 C#中的委托和事件” <http://www.cnblogs.com/jimmyzhang/archive/2007/09/23/903360.html>。
- (2) “2008 C#中的委托和事件” <http://www.cnblogs.com/JimmyZhang/archive/2008/08/22/1274342.html>。

如果希望深入了解委托，请参见 Jeffery Richter 所著的《.Net 框架程序设计 CLR Via C#》（第 2 版）第 15 章。接下来，我们将要讲述的是 WPF 的 Command 模型，此为心法一卷最后一章。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 神雕侠侣》，“第三十八回 生死茫茫”。
- [2] MSDN Library for Visual Studio 2008 SP1 Routed Events Overview。

WPF 的命令 (Command) —— 明教的圣火令

谢逊仰天长笑，声动山谷，大声道：“金毛狮王光明磊落，别说不杀同伙朋友，此人即令是谢某的深仇大怨，既被你们擒住，已然无力抗拒，谢某岂能再以白刃相加？”

……

只听妙风使道：“明教教徒，见圣火令如见教主，你胆敢叛教吗？”谢逊昂然道：“谢某双目已盲了二十余年，你便将圣火令放在我眼前，我也瞧它不见，说什么‘见圣火令如见教主’？”

——《倚天屠龙记》“第二十九章 四女同舟何所望”

这一段讲的是波斯明教护法以圣火令命金毛狮王杀死金花婆婆，谁知金毛狮王豪迈爽朗以“眼盲看不到圣火令”为由拒绝了波斯明教三使。

WPF 也有这样的圣火令——Command，本章首先从木木学做写字板程序开始讨论，然后讨论 WPF 中的 Command 模型。

本章内容如下。

- (1) 木木的写字板（无 Command）。
- (2) 小徐的写字板（有 Command）。
- (3) 为什么需要 Command。
- (4) WPF 的 Command 模型。
- (5) 接下来做什么。

7.1 木木的写字板（无 Command）

了解完依赖属性和路由事件，木木确实感觉自己内功见长。但是前面的学习理论多，实践少。于是木木想从一个最常见的写字板程序做起。由于木木是初学者，好多 WPF 基础知识还不具备（如布局等）。因此每一步的进行都很艰难，好在木木边翻书，边上网，本着“内事问百度，外事问谷歌”的原则，一番折腾倒也搭建出了写字板的程序界面。

7.1.1 简单的写字板原型

木木搭建写字板程序界面的布局容器为 Grid，它将界面划分为 3 行。由上至下分别是菜单（Menu）、工具栏（ToolBar）和文本块（TextBlock），如图 7-1 所示。

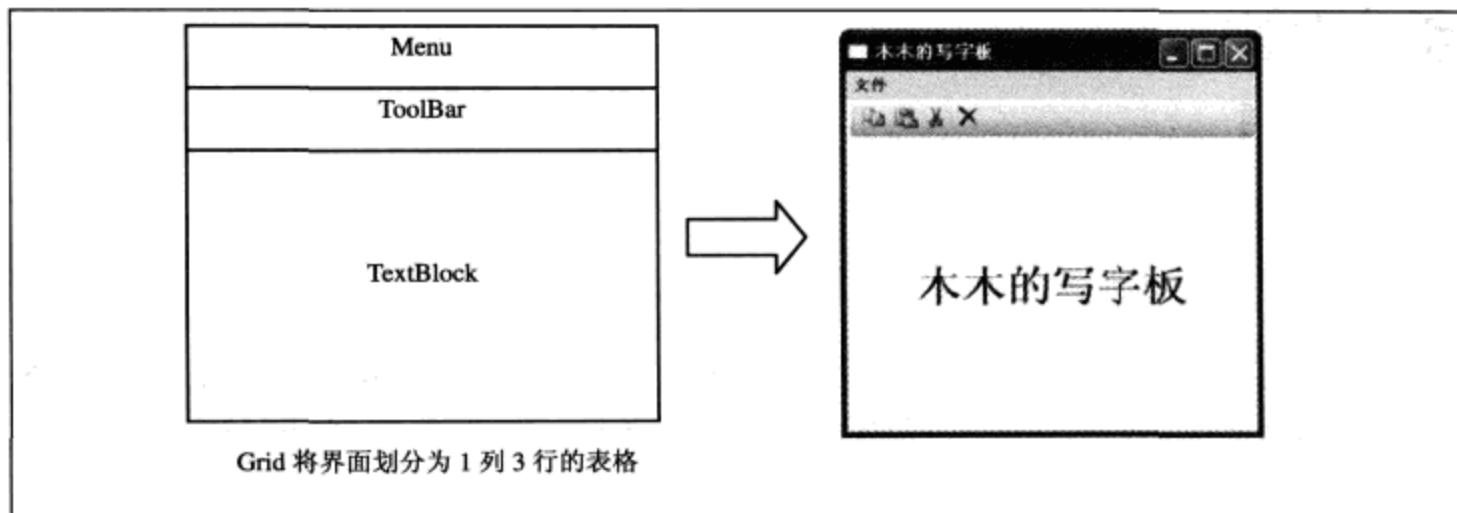


图 7-1 木木的写字板程序布局

代码如 7-1 所示。

```
<Window x:Class="mumu_notpadwithoutcommand.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="木木的写字板" Height="300" Width="300">
<Grid>
    <!--将 Grid 面板划分为 3 行，Auto 表示前两行的高度适应控件（菜单和工具栏）的高度，* 表示第 3 行的高度占据其剩下的空间-->
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <!--Grid.Row=0 表示 Menu 在 Grid 面板的第 1 行-->
    <Menu Grid.Row="0">
        <MenuItem Header="文件">
            <MenuItem Header="复制" />
            <MenuItem Header="粘贴" />
            <MenuItem Header="剪切" />
            <MenuItem Header="删除" />
        </MenuItem>
    </Menu>
    <ToolBar Grid.Row="1">
        <!--ToolBar 中摆放的按钮，而按钮中摆放的是 Image，Image 的 source 属性指定图片的来源-->
        <Button >
            <Image Source="/images/CopyHS.png"/>
        </Button>
        <Button >
            <Image Source="/images/PasteHS.png"/>
        </Button>
        <Button >
            <Image Source="/images/CutHS.png"/>
        </Button>
```

```

        </Button>
        <Button >
            <Image Source="images/DeleteHS.png" />
        </Button>
    </ToolBar>
    <TextBlock x:Name="text" Grid.Row="2" Text="木木的写字板" FontSize="32"
HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Grid>
</Window>

```

代码 7-1 Window1.xaml 文件（详见 mumu_notpadwithoutcommand 工程）

复制和粘贴功能主要借助剪贴板（Clipboard）类来实现，删除功能则将文本数据直接赋成空值，剪切的实质是先复制后删除两项操作的组合。

首先在 XAML 文件中添加事件处理函数，由于工具栏按钮和菜单选项的功能一一对应，因此二者关联同一个事件处理函数，如代码 7-2 所示。

```

window1.xaml
.....
<!--为菜单添加事件处理函数 -->
<MenuItem Header="文件">
    <MenuItem Header="复制" Click="CopyOnClick" />
    <MenuItem Header="粘贴" Click="PasteOnClick" />
    <MenuItem Header="剪切" Click="CutOnClick" />
    <MenuItem Header="删除" Click="DeleteOnClick" />
</MenuItem>
<!--为工具按钮添加事件处理函数 -->
<ToolBar Grid.Row="1">
    <Button Click="CopyOnClick">
        <Image Source="/images/CopyHS.png" />
    </Button>
    <Button Click="PasteOnClick">
        <Image Source="/images/PasteHS.png" />
    </Button>
    <Button Click="CutOnClick">
        <Image Source="images/CutHS.png" />
    </Button>
    <Button Click="DeleteOnClick">
        <Image Source="images/DeleteHS.png" />
    </Button>
</ToolBar>
.....

```

代码 7-2 为工具栏和菜单添加事件处理函数

在.cs 文件（code-behind）中剪切、复制、粘贴和删除功能的具体实现如代码 7-3 所示。

```

Window1.xaml.cs
.....
protected void CutOnClick(object sender, RoutedEventArgs args)
{
    CopyOnClick(sender, args);
    DeleteOnClick(sender, args);
}
protected void CopyOnClick(object sender, RoutedEventArgs args)
{
    if (text.Text != null && text.Text.Length > 0)

```

```

        Clipboard.SetText(text.Text);
    }
    protected void PasteOnClick(object sender, RoutedEventArgs args)
    {
        if (Clipboard.ContainsText())
            text.Text = Clipboard.GetText();
    }
    protected void DeleteOnClick(object sender, RoutedEventArgs args)
    {
        text.Text = null;
    }

```

代码 7-3 剪切、复制、粘贴和删除的具体实现

做完这些，木木还是很有成就感的。于是把正在午睡的室友小徐叫起来，让他看看自己的成果。小徐睡意正浓，被木木吵醒颇不高兴。将木木的程序看了看，很不以为然，说到：“这是个什么破程序。只能通过菜单和工具栏来实现剪切、复制和粘贴的功能。没有右键菜单，也没有 Ctrl+C 这样的快捷键。你这个写字板实在是太‘山寨’了”。小徐说完继续倒头就睡。

7.1.2 右键菜单和快捷键

木木被小徐的一番话郁闷了，但是想想确实说得也有道理。木木一股“牛”劲上来。没有右键菜单，我加上！没有快捷键，我也加上！

首先为 TextBlock 添加右键菜单，只需为 TextBlock 添加 ContextMenu 属性即可。在其中仍然复用过去的事件处理函数，如代码 7-4 所示。

```

Window1.xaml
<TextBlock x:Name="text" Grid.Row="2" Text="木木的写字板" FontSize="32"
HorizontalAlignment="Center" VerticalAlignment="Center" MinWidth="50"
MinHeight="50" Background="AliceBlue">
    <TextBlock.ContextMenu>
        <ContextMenu>
            <MenuItem Header="复制" Click="CopyOnClick"/>
            <MenuItem Header="粘贴" Click="PasteOnClick"/>
            <MenuItem Header="剪切" Click="CutOnClick"/>
            <MenuItem Header="删除" Click="DeleteOnClick" />
        </ContextMenu>
    </TextBlock.ContextMenu>
</TextBlock>

```

代码 7-4 添加右键菜单

添加快捷键的确难倒了木木，好在木木找到了这样一个类 KeyGesture，KeyGesture 最主要的是一个静态方法 Matches，该方法可以比较出用户的输入是否是类似 Ctrl+C 这样的组合键。木木重载了窗口的 OnPreviewKeyDown 函数，用于比较其中键盘的输入参数是否为剪切、复制、粘贴和删除快捷键。注意如果发现用户输入为以上几种快捷键，则要将 e.Handled 设置为 true，防止键盘输入事件进一步传递，如代码 7-5 所示。

```

Window1.xaml.cs
public partial class Window1 : Window
{

```

```

// 声明 4 个 KeyGesture，分别是 Ctrl+X、Ctrl+C、Ctrl+V 及 delete 共 4 种组合方式
private KeyGesture gestCut = new KeyGesture(Key.X, ModifierKeys.Control);
private KeyGesture gestCopy = new KeyGesture(Key.C, ModifierKeys.Control);
private KeyGesture gestPaste = new KeyGesture(Key.V, ModifierKeys.Control);
private KeyGesture gestDelete = new KeyGesture(Key.Delete);

// 重载窗口的 OnPreviewKeyDown 函数，以比较输入参数是否是预定义的几种快捷键方式
protected override void OnPreviewKeyDown(KeyEventArgs e)
{
    base.OnPreviewKeyDown(e);
    e.Handled = true;
    if (gestCut.Matches(null, e))
        CutOnClick(this, e);
    else if (gestCopy.Matches(null, e))
        CopyOnClick(this, e);
    else if (gestPaste.Matches(null, e))
        PasteOnClick(this, e);
    else if (gestDelete.Matches(null, e))
        DeleteOnClick(this, e);
    else
        e.Handled = false;
}

}

```

代码 7-5 添加快捷键

7.1.3 控制功能状态

这个时候，小徐已经睡醒了，见木木还在埋头编程，不由凑过去看了看。没想到刚才自己无心的说话，木木居然当了真，已经把右键菜单和快捷键都添加了上去，心里不由暗暗佩服这小子的一股认真劲。小徐这回非常认真地看了看木木的程序，说到：“还是有一个问题，木木。比如当我使用了剪切功能之后，除了粘贴功能可以启用，其他功能都是禁用。你这里面应该对功能是否启用有所控制。”

木木觉得小徐确实说的有理，于是仔细想了想，有如下两种条件：

- (1) 剪切、复制和删除功能取决于 text 的文本是否为空，如果为空，则不能实现；否则可以。
- (2) 粘贴功能取决于剪贴板中是否包含文本，如果包含文本，则可以；否则不可。

木木想清楚了这两个问题，觉得还是比较简单的。于是开始编写代码了。谁知道这回远没有前一次那么顺利。最后算是改完了，可是木木已经满头大汗。而且木木觉得代码写得格外别扭。我们一起来看看是怎么做的。

1. 为菜单和右键菜单添加状态控制

木木的思路是在每次菜单打开之前，检查各个功能的状态，然后决定是否可用。因此在文件菜单项里添加了 SubmenuOpened 事件的处理函数。为了对每个菜单进行控制，需要给每个菜单对象加上了 x:Name 属性，以便在 C# 文件里能够直接访问到这些对象，如代码 7-6 所示。

```

Window1.xaml
<MenuItem Header="文件" SubmenuOpened="MenuItem_SubmenuOpened">
```

```

<MenuItem x:Name="copyitem" Header="复制" Click="CopyOnClick"/>
<MenuItem x:Name="pasteitem" Header="粘贴" Click="PasteOnClick"/>
<MenuItem x:Name="cutitem" Header="剪切" Click="CutOnClick"/>
<MenuItem x:Name="deleteitem" Header="删除" Click="DeleteOnClick" />
</MenuItem>

Window1.xaml.cs
private void MenuItem_SubmenuOpened(object sender, RoutedEventArgs e)
{
    cutitem.IsEnabled = copyitem.IsEnabled = deleteitem.IsEnabled =
text.Text != null && text.Text.Length > 0;
    pasteitem.IsEnabled = Clipboard.ContainsText();
}

```

代码 7-6 为菜单功能项添加控制状态

右键菜单也如此，不同的是需要在 TextBlock 的 ContextMenuOpening 事件中检查各个功能的状态。

```

<TextBlock x:Name="text" Grid.Row="2" Text="木木的写字板" FontSize="32"
HorizontalAlignment="Center" VerticalAlignment="Center" MinWidth="50" MinHeight="50"
Background="AliceBlue" ContextMenuOpening="text_ContextMenuOpening">
```

2. 为快捷键添加状态控制

快捷键状态控制在 OnPreviewKeyDown 函数中，如代码 7-7 所示。

```

protected override void OnPreviewKeyDown(KeyboardEventArgs e)
{
    base.OnPreviewKeyDown(e);
    // 添加状态控制代码
    bool iscut, isdelete, iscopy, ispaste;
    iscut = isdelete = iscopy = text.Text != null && text.Text.Length > 0;
    ispaste = Clipboard.ContainsText();

    if (iscut && gestCut.Matches(null, e))
    {
        CutOnClick(this, e);
        e.Handled = true;
    }
    else if (iscopy && gestCopy.Matches(null, e))
    {
        CopyOnClick(this, e);
        e.Handled = true;
    }

    else if (ispaste && gestPaste.Matches(null, e))
    {
        PasteOnClick(this, e);
        e.Handled = true;
    }
    else if (isdelete && gestDelete.Matches(null, e))
    {
        DeleteOnClick(this, e);
        e.Handled = true;
    }

    else
    {

```

```
        e.Handled = false;
    }
}
```

代码 7-7 为快捷键添加状态控制

3. 为工具栏添加状态控制

同样也要为工具栏添加状态控制代码，为此在每次剪切、复制、粘贴和删除操作之后判断哪个工具可用：

```
protected void CutOnClick(object sender, RoutedEventArgs args)
{
    .....
    ProcessToolBar();
}
protected void CopyOnClick(object sender, RoutedEventArgs args)
{
    .....
    ProcessToolBar();
}
protected void PasteOnClick(object sender, RoutedEventArgs args)
{
    .....
    ProcessToolBar();
}
protected void DeleteOnClick(object sender, RoutedEventArgs args)
{
    .....
    ProcessToolBar();
}
```

开始 ProcessToolBar 和前面的处理相同，即经过判断发现功能不可用，则将工具栏的按钮 IsEnable 属性设置为 false；否则为 true。但是按钮在不可用时仅仅是单击没有响应，而不是图像变灰来表示不可用，如图 7-2 所示。

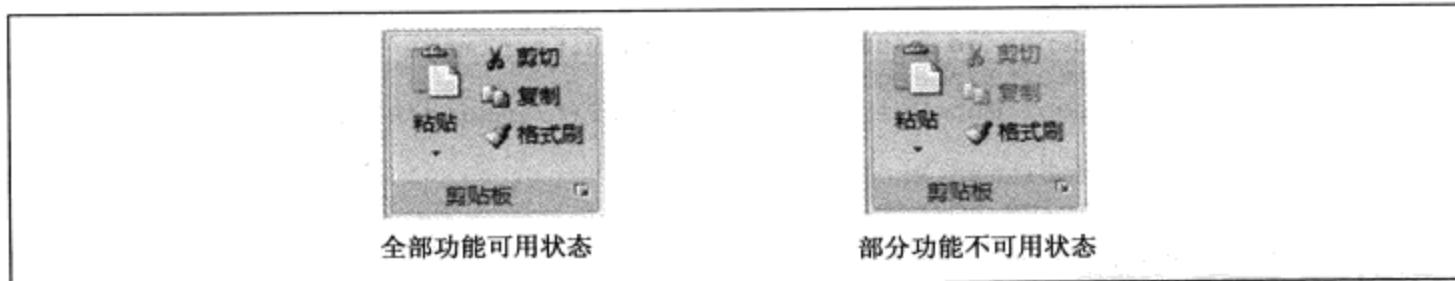


图 7-2 在 Word 中不可用的按钮图像会变灰

在 ProcessToolBar 中根据功能是否可用更换按钮的不同图片，如代码 7-8 所示。

```
private void ProcessToolBar()
{
    bool isNoPaste = text.Text != null && text.Text.Length > 0;
    if (!isNoPaste)
    {
        // 复制
        BitmapImage copybitmap = new BitmapImage();
        copybitmap.BeginInit();
```

```

        copybitmap.UriSource = new Uri("/images/grayCopyHS.png",
UriKind.Relative);
        copybitmap.EndInit();

        copyimg.Source = copybitmap;
        // 删除
        ....
        // 剪切
        ....
    }
else
{
    // 复制
    BitmapImage copybitmap = new BitmapImage();
    copybitmap.BeginInit();
    copybitmap.UriSource = new Uri("/images/CopyHS.png",
UriKind.Relative);
    copybitmap.EndInit();
    copyimg.Source = copybitmap;
    // 删除
    ....
    // 剪切
    ....
}

bool ispaste = Clipboard.ContainsText();
if (!ispaste)
{
    // 粘贴
    BitmapImage pastebitmap = new BitmapImage();
    pastebitmap.BeginInit();
    pastebitmap.UriSource = new Uri("/images/grayPasteHS.png",
UriKind.Relative);
    pastebitmap.EndInit();
    pasteimg.Source = pastebitmap;
}
else
{
    // 粘贴
    BitmapImage pastebitmap = new BitmapImage();
    pastebitmap.BeginInit();
    pastebitmap.UriSource = new Uri("/images/PasteHS.png",
UriKind.Relative);
    pastebitmap.EndInit();
    pasteimg.Source = pastebitmap;
}
}

```

代码 7-8 不可用时使按钮变灰

7.1.4 小徐的写字板为何如此简单

自此，木木的写字板算是大功告成了。小徐体验了一下。唔，还算是不错。突然小徐想到了什么，打开自己的笔记本，说到：“木木，我这儿也有一个和你非常类似的写字板程序，不过实现起来要比你的简单多了。”

```

Window1.xaml
<Window x:Class="mumu_notpadwithcommand.Window1"

```

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="写字板" Height="300" Width="300">
<Window.CommandBindings>
    <CommandBinding Command="Close" Executed="CloseCommand"/>
    <CommandBinding Command="Save" Executed="SaveExecuted"
CanExecute="SaveCanExecute"/>
</Window.CommandBindings>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Menu Grid.Row="0">
        <MenuItem Header="文件">
            <MenuItem Header="剪切" Command="Cut"/></MenuItem>
            <MenuItem Header="复制" Command="Copy"/></MenuItem>
            <MenuItem Header="粘贴" Command="Paste"/></MenuItem>
            <Separator></Separator>
            <MenuItem Header="保存" Command="Save"/>
            <MenuItem Header="关闭" Command="Close"/></MenuItem>
        </MenuItem>
    </Menu>
    <ToolBar Grid.Row="1">
        <Button Command="Cut">
            剪切
        </Button>
        <Button Command="Copy">
            复制
        </Button>
        <Button Command="Paste">
            粘贴
        </Button>
        <Button Command="Save">
            保存
        </Button>
        <Button Command="Close">
            关闭
        </Button>
    </ToolBar>
    <TextBox Grid.Row="2" AcceptsReturn="True" TextChanged="TextBox_TextChanged">
    </TextBox>
</Grid>
</Window>

```

代码 7-9 mumu_notpadwithcommand 工程中的 Window1.xaml 文件

代码 7-10 更为简单，只是多了几个响应保存和关闭命令的函数。

```

Window1.xaml.cs
public partial class Window1 : Window
{
    private bool isdirty = false;
    public Window1()
    {
        InitializeComponent();
    }

    private void CloseCommand(object sender, ExecutedRoutedEventArgs e)

```

```

    {
        MessageBox.Show("CloseCommand triggered with " + e.Source.ToString());
        App.Current.Shutdown();
    }

    private void SaveExecuted(object sender, ExecutedRoutedEventArgs e)
    {
        MessageBox.Show("Save command triggered with " + e.Source.ToString());
        isdirty = false;
    }

    private void SaveCanExecute(object sender, CanExecuteRoutedEventArgs e)
    {
        e.CanExecute = isdirty;
    }

    private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
    {
        isdirty = true;
    }
}

```

代码 7-10 mumu_notpadwithcommand 工程中的 Window1.xaml.cs 文件

没有右键菜单、快捷键、控制状态代码，甚至没有剪切、复制、粘贴和删除的功能代码，但是这些功能居然全部实现了。

7.2 小徐的写字板（有 Command）

上一节的写字板（简称“徐版”）的所有功能均通过 Command 来驱动，如表 7-1 所示。

表 7-1 徐版写字板所用到的 Command

名称	描述
ApplicationCommands.Cut	剪切
ApplicationCommands.Copy	复制
ApplicationCommands.Paste	粘贴
ApplicationCommands.Save	保存
ApplicationCommands.Close	关闭

此外使用 TextBox 替代了 TextBlock，从继承的层次关系来看 TextBox 派生自 Control。而 TextBlock 直接派生自 FrameworkElement，说明 TextBox 本身比 TextBlock 的功能更为复杂，如图 7-3 所示。

徐版支持的 5 个命令中的前 3 个命令由于 TextBox 本身内置了处理函数，因此不需要编写任何多余的代码。

以剪切功能为例，只需将 MenuItem 和工具栏的 Button 的 Command 属性设为 ApplicationCommands.Cut 即可。由于 TextBox 默认包含剪切、复制和粘贴右键菜单，因此也不需要为其添加右键菜单，如代码 7-11 所示。

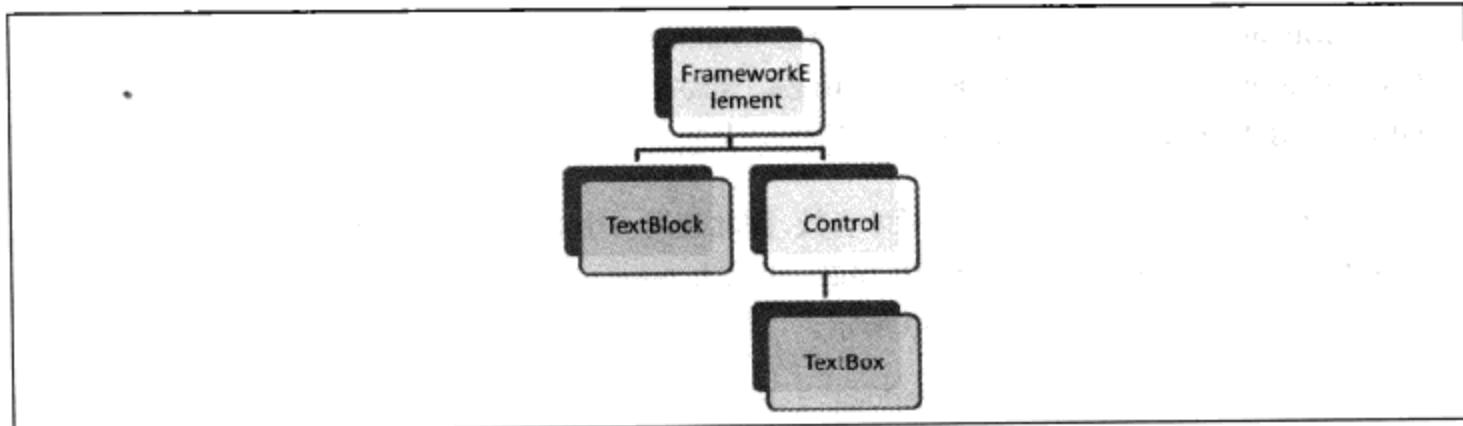


图 7-3 TextBox 的类层次结构

```

<MenuItem Header="文件" Grid.Row="0" >
    <MenuItem Header="剪切" Command="ApplicationCommands.Cut"></MenuItem>
    ...
</MenuItem>

<Button Command="ApplicationCommands.Cut">
    剪切
</Button>
...
  
```

代码 7-11 通过 Command 添加剪切功能

当没有选中文本时菜单、工具栏、右键菜单和快捷键（Ctrl+X）的剪切功能都会自动禁用（菜单、工具栏和右键菜单会自动变灰，快捷键无效），选中文本时则自动启用，如图 7-4 所示。

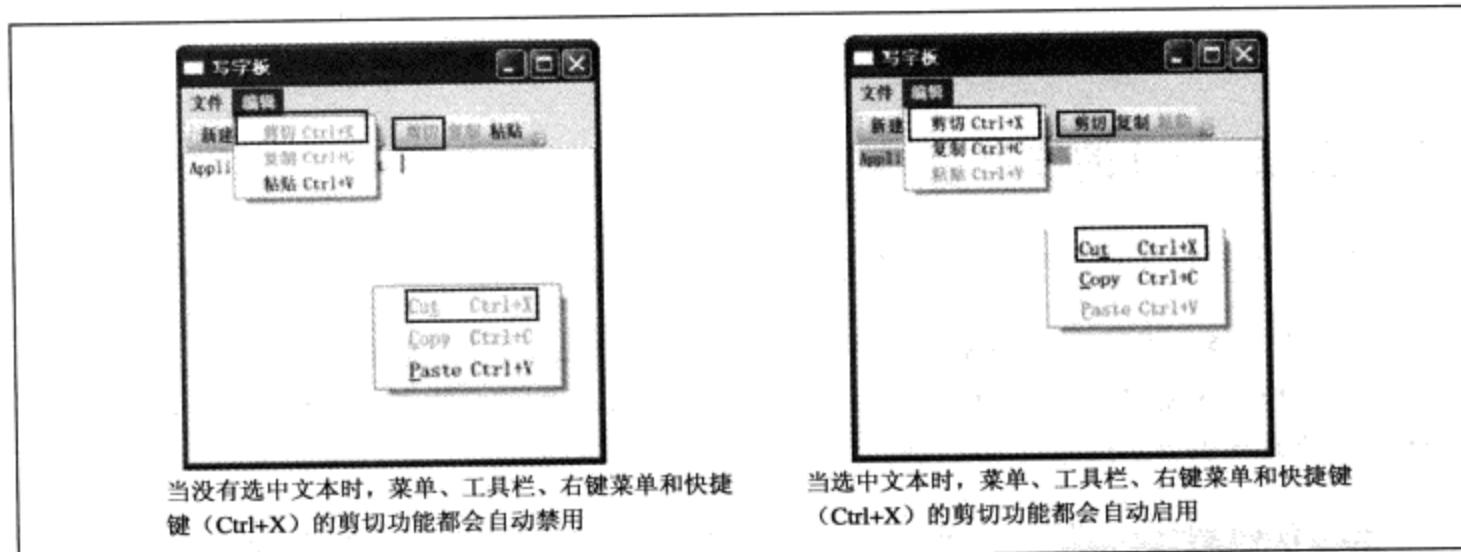


图 7-4 使用 Command 自动控制各功能的状态

关闭和保存命令需要编写相应的命令处理函数，以保存命令为例，首先要设置菜单和工具栏的 Command 属性。然后通过 CommandBinding 关联 Command 和 Command 处理函数，如代码 7-12 所示。

```

Window1.xaml
<Window.CommandBindings>
    <CommandBinding CanExecute="SaveCanExecute" />
    <CommandBinding Command="Save" Executed="SaveExecuted" />
</Window.CommandBindings>
  
```

代码 7-12 通过 CommandBinding 将 Command 和 Command 处理函数联系起来

CommandBinding 有两个重要事件，执行关联的 Command 时，会触发该 Executed 事件的处理函数；触发 CanExecute 事件时，相应的事件处理函数需要将传递来的参数 e 的 CanExecute 属性为 true 或者 false，告知 WPF 命令系统该命令是否可用。

如保存命令只有在文本内容发生改变时才有效，因此使用一个 bool 变量 isDirty 来表示 TextBox 的文本是否改变。当发生改变时，isDirty 为 true；否则为 false。在 SaveCanExecute 函数中就将 isDirty 直接赋值给 e.CanExecute，如代码 7-13 所示。

```
public partial class Window1 : Window
{
    private bool isDirty = false;
    .....

    private void SaveExecuted(object sender, ExecutedRoutedEventArgs e)
    {
        MessageBox.Show("Save command triggered with " + e.Source.ToString());
        isDirty = false;
    }

    private void SaveCanExecute(object sender, CanExecuteRoutedEventArgs e)
    {
        e.CanExecute = isDirty;
    }

    private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
    {
        isDirty = true;
    }
}
```

代码 7-13 保存命令的状态控制

7.3 Command 的作用

从前面木木的写字板（简称木版）和徐版的对比，我们已经能够感受到 Command 的强大威力了吧。

木版中同一个功能由不同 UI 事件触发（单击菜单为菜单的 Click 事件，单击工具栏为工具栏按钮的 Click 事件，按下快捷键则是 PreviewKeydown 事件），然后转到相应的事件处理函数中处理。而各个 UI 的状态还需要分别控制，因此代码会非常凌乱，如图 7-5 所示。

徐版则把所有 UI 触发（单击菜单、单击工具栏按钮和按下快捷键等）统一转到 CommandBinding 关联将特定的命令和命令处理函数，它通过 Executed 和 CanExecute 事件集中处理事件和控制 UI 状态，这正是徐版简洁且功能齐备的原因，如图 7-6 所示。

Command 本身是比事件更为高级的概念，它把应用程序的功能划分为任务（如保存及剪切等），而这些任务可以由多种途径（菜单、工具栏和快捷键等）来触发。使用 Command 的主要原因如下。

- (1) 使代码更为简洁，容易支持菜单、工具栏和快捷键等多种途径触发。
- (2) 无须担心各个 UI 的状态（启用或者禁用）。

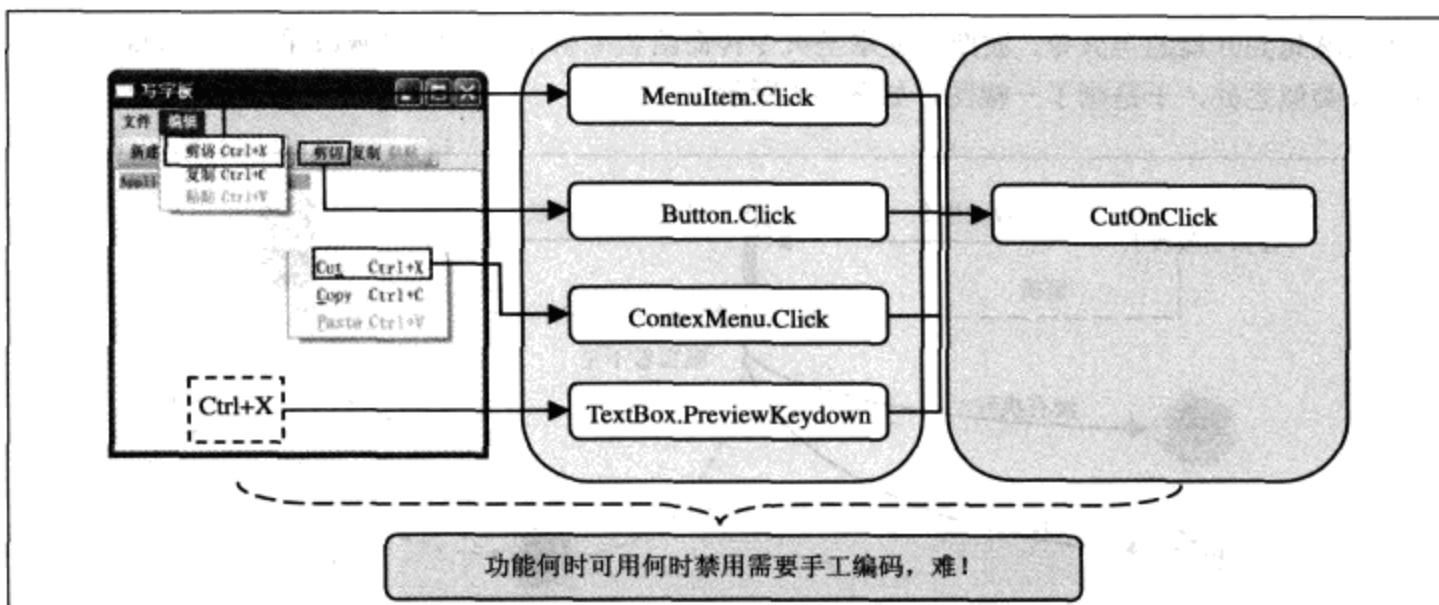


图 7-5 木版不同 UI 事件触发响应模型

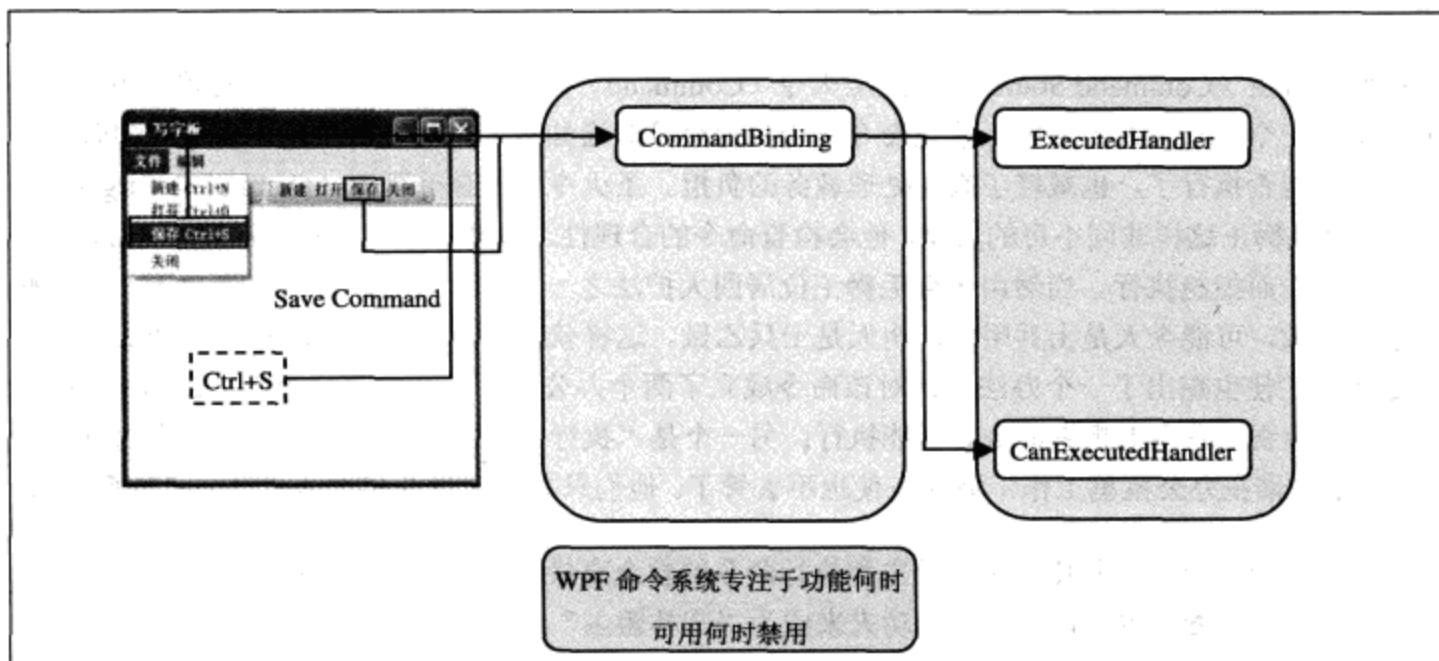


图 7-6 徐版 UI 事件触发响应模型

7.4 WPF 的 Command 模型

WPF 的 Command 模型包含如下 4 个部分。

- (1) Command: 应用程序需要执行的任务，比如前面的关闭等任务。
- (2) CommandBinding: 连接 Command 和特定的应用程序逻辑，如前面 CommandBinding 连接 Save 命令及其处理函数。
- (3) Command Source: 触发 Command 的对象，如前述的菜单和工具栏，单击可以触发绑定的 Command。
- (4) Command target: Command 应用在上面的对象，如前述的 TextBox。

木木突然想到开篇的圣火令，波斯三使拿圣火令传命给金毛狮王的过程和 WPF 的 Command 模型颇有几分类似之处，于是画了一幅图，如图 7-7 所示。

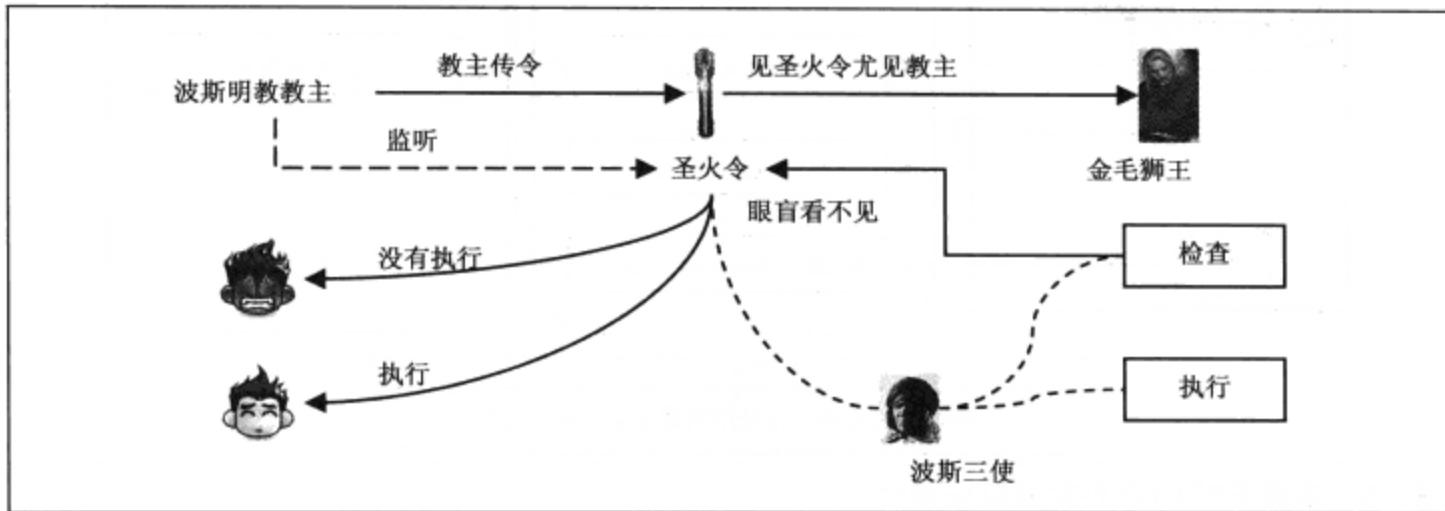


图 7-7 WPF 的 Command 模型

波斯明教教主（Command Source）通过圣火令（Command）向教众发布命令。当然明教圣物——圣火令也有一个非常神奇的功能，就是命令能否执行会及时通知教主。这样教主就不必天天问张三，李四命令能否执行了，也减轻了教主处理教务的负担。圣火令到了金毛狮王（Command Target）这儿，像金毛狮王这样非同小可的人物，他会检查命令的合理性，如果命令不合理，金毛狮王就以“眼盲看不见”而拒绝执行。当然由于金毛狮王位居四大护法之一，非常大牌。这些事情也不一定是狮王亲自去做，可能今天是士兵甲做，明天是士兵乙做。这样就苦了波斯三使（CommandBinding），于是波斯三使也想出了一个办法，针对该命令成立了两个办公室：一个是“检查”（CanExecute）办公室，负责检查金毛狮王命令是否能执行；另一个是“执行”（Execute）办公室，负责执行该命令。至于是谁在办公室里工作¹，波斯三使也不去管了。他们只关心把命令和这两个办公室联系起来。

这样传圣火令的方式，避免了明教教主和执行命令的教众直接联系。明教教主和教众形成了一种松耦合关系，这样教主也可以腾出更多功夫来成为“武林盟主”。

7.4.1 Command——圣火令

1. ICommand 接口

一个 Command 代表应用程序的任务，如前面的保存、关闭和剪切等任务。在 WPF 中的所有 Command 都要实现 ICommand 接口，该接口有两个方法（Execute 和 CanExecute）和一个事件（CanExecuteChanged）。

(1) Execute 方法：当 Command 被触发时调用该方法，执行与命令相对应的操作。

(2) CanExecute 方法：用来判断该命令是否可应用到当前 Command target 上，如果该方法返回为 true，则可以；否则不行。

¹ 这个工作多半既不是金毛狮王，也不是士兵甲或者士兵乙去做，而是“苦命”的程序员。

(3) CanExecuteChanged 事件：Command 有执行或者不执行两种状态，状态改变时触发该事件。一般监听该事件的是 CommandSource，它监听到该事件后会调用绑定的 Command 的 CanExecute 方法检查当前 Command 的状态，然后决定 CommandSource 是启用还是禁用。

尽管上述的细节听起来有些复杂，但是好在开发人员很少直接从 ICommand 继承实现一个 Command，WPF 提供了一个 RoutedCommand 类实现了 ICommand 接口。

2. 类 RoutedCommand

按照一般思路，可能会认为 RoutedCommand 直接实现 Execute 函数，如代码 7-14 所示。

```
public class NewCommand : ICommand
{
    void Execute(object parameter)
    {
        NewFile();
    }
    private void NewFile()
    {
        .....
    }
}
```

代码 7-14 从 ICommand 派生而来的 NewCommand

这种方式存在的问题是命令本身仅描述应用程序任务的类型，并不负责实现这个任务。实现该任务是开发人员负责的业务逻辑，因此“描述任务”和“实现任务”需要分开（解耦）。

RoutedCommand 通过路由事件使其解耦，Execute 方法实际只是从 CommandTarget 开始触发 PreviewExecuted 和 Executed 事件；CanExecute 方法负责从 CommandTarget 开始触发 CanExecute 和 PreviewCanExecute 事件。这些事件有的从逻辑树上下沉到底——Tunnel（PreviewExecuted 和 PreviewCanExecute 事件），有的从逻辑树上一直上升——Bubble（Executed 和 CanExecute），总之最终到达 CommandBinding 绑定的事件处理函数中处理。

RoutedCommand 的 InputGestures 属性用来指定鼠标和键盘动作，从而可以关联命令和对应的鼠标键盘动作。

RoutedCommand 的一个派生类为 RoutedUICommand，它多了一个 Text 属性，用来在 UI 上显示该命令的描述信息。WPF 的设计者最初的想法也许是希望通过 Text 属性来解决本地化的问题，不希望在本地化时修改各个控件的标题。而只是修改 Text 属性，从而使所有的控件标题信息都发生改变。

3. WPF 内置的 Command 库

WPF 提供了 100 多个通用的命令，这些命令分为 5 种类型。它们通过静态属性提供各种命令，如表 7-2 所示。

表 7-2 WPF 内置的 Command 库

库	说明
ApplicationCommands	提供一些常用的功能，如剪切（Cut）、复制（Copy）、粘贴（Paste），以及一些文本的命令，如新建（New）、打开（Open）、打印（Print）和打印预览（PrintPreview）等
NavigationCommands	提供导航一类的命令，如前进（BrowserForward）和后退（BrowseBack）等
EditingCommands	提供标准的编辑命令，如左对齐（AlignLeft）、右对齐（AlignRight）和加粗（ToggleBold）等
ComponentCommands	处理用户界面的命令，如焦点前移（MoveFocusForward）和焦点后移（MoveFocusBackward）等
MediaCommands	提供处理多媒体的命令，如播放（Play）及暂停（Pause）等

这些内置的 Command 都可以简写，如 ApplicationCommand.New 简写为“New”，例如“<MenuItem Command="New"/>”。

7.4.2 Command Sources——明教教主

Command Source 是能够触发 Command 的对象按钮和菜单，包括键盘和鼠标操作都可以是 Command Source。

Command Source 必须实现 ICommandSource 接口，该接口定义了如下 3 个属性。

- (1) Command: Command Source 会触发的 Command。
- (2) CommandParameter: Command 执行时需要的参数，大多数情况下命令的执行并不需要参数。
- (3) CommandTarget: Command 的应用对象。

WPF 中实现 ICommandSource 接口的有 ButtonBase、MenuItem、Hyperlink 和 InputBinding，其中前 3 个被单击时会触发 Command 命令；而 InputBinding 则通过鼠标和键盘操作的执行触发 Command。

7.4.3 Command Binding——波斯三使

CommandBinding 关联 Command 和实现 Command 的事件处理函数，如前面徐版当中 CommandBinding 就将 Save 命令与 SaveExecuted 和 SaveCanExecute 函数联系起来。

CommandBinding 类包括 Command 属性，以及 PreviewExecuted、Executed、PreviewCanExecute 和 CanExecute 事件。如果一个元素派生自 UIElement、ContentElement 和 UIElement3D 之一，那么 CommandBinding 对象可以添加到该元素中。在 XAML 中的添加方式如代码 7-15 所示。

```
<Window.CommandBindings>
    <CommandBinding Command="Close" Executed="CloseCommand"/>
    <CommandBinding Command="Save" Executed="SaveExecuted"
CanExecute="SaveCanExecute"/>
</Window.CommandBindings>
```

代码 7-15 在 XAML 中添加 CommandBinding

CommandBinding 也可以在代码文件中添加，如代码 7-16 所示：

```
public Window1()
{
    CommandBinding SaveCommandBinding = new
    CommandBinding(ApplicationCommands.Save, SaveExecuted, SaveCanExecute);
    this.CommandBindings.Add(SaveCommandBinding);
}
```

代码 7-16 在 C# 中添加 CommandBinding

注意 CommandBinding 的位置。当 Command 执行时会触发路由事件，路由事件会顺着元素树寻找相应的 CommandBinding 对象。如果该对象不在事件的路由上，则无法回应关联的命令处理函数。图 7-8 所示的 CommandBinding 放置在不同位置的情况，在图(a)和图(b)中 CommandBinding 对象分别在窗口元素(Window)和 Grid 面板上，均在事件的路由路线上，因此当按钮触发 Command 命令时会响应该命令对应的命令处理函数。而图(c)中 CommandBind 对象在按钮的兄弟节点上，并不在事件的路由线路上，因此无法响应对应的命令处理函数。

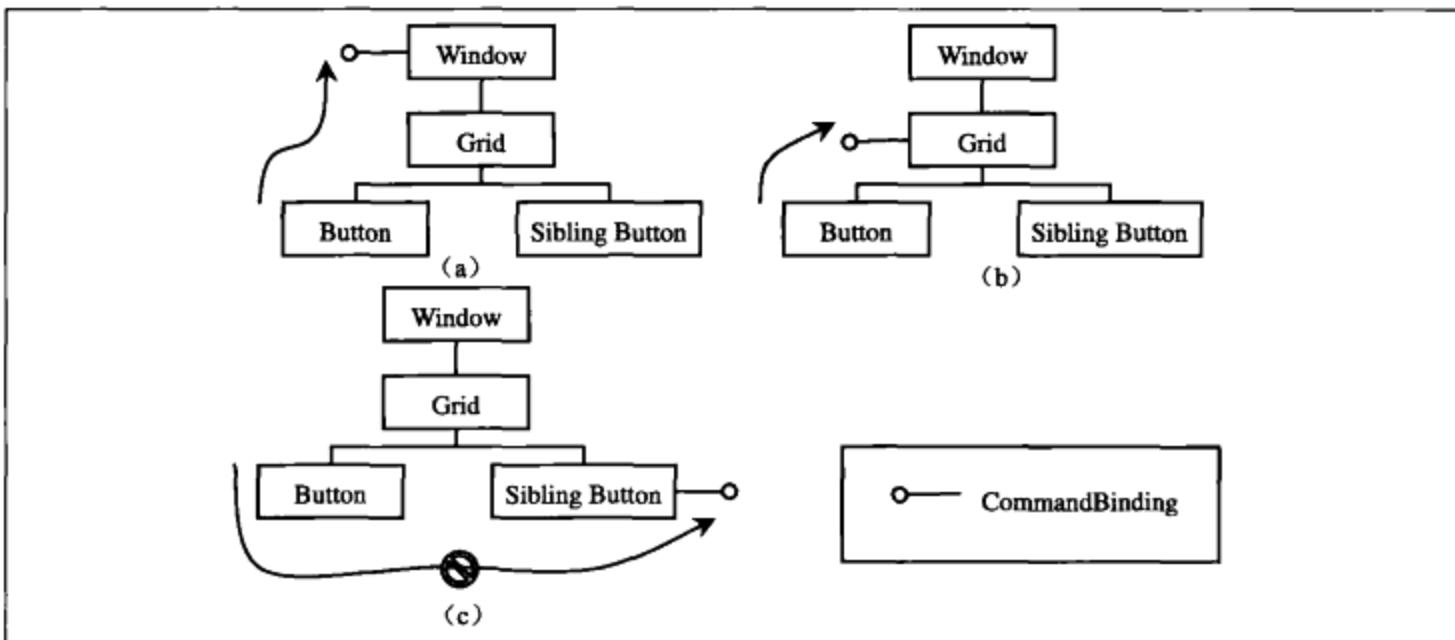


图 7-8 CommandBinding 在逻辑树上的不同位置

7.4.4 Command Target——金毛狮王

Command Target 是 Command 会应用在上面的对象，如徐版的 TextBox。前面已经说过 Command Source 都需要实现一个 ICommandSource 接口，其中就有一个属性是 Command Target。当 Command Source 不明确指定 Command Target 属性时，会把当前获得焦点的元素认为是 Command Target。

此外 WPF 中很多 Command Target 均内置对某些命令的支持，最典型的是 TextBox 元素内置了剪切、复制和粘贴等多项文本命令的支持。

7.5 接下来做什么

在这一章当中，自定义 Command 模型，我们并没有讲述。这一部分的内容将在第 20 章里讲述。心

法一卷到此就告一段落，实际上第四卷峰回路转，夯实基础也属于心法。不过光理论不实践是空洞的理论，下一卷我们将开始打造一个完整的 WPF 应用。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 倚天屠龙记》，“第二十九章 四女同舟何所望”。
- [2] MSDN Library for Visual Studio 2008 SP1 Commanding Overview。

小有所成

第三卷



第8章

应用程序窗口——大侠的成长路线

应用程序和窗口这样的概念对木木来说并不算陌生，无论过去 Windows 编程还是 WinForm 都有过这样类似的概念，因此这一章对木木来说也许是“最熟悉的陌生人”^_^。

本章内容如下。

- (1) 新建一个应用程序。
- (2) 应用程序和它的生命周期。
- (3) 窗口。
- (4) 接下来做什么。

8.1 新建一个应用程序

8.1.1 手动创建

我们平常是按照向导的方式建立起一个标准的 WPF Application。这种方式只需要设置好名称和路径，按下 OK 按钮即可。但是无疑这种过于自动化的方式掩盖了很多的细节，那么我们就按照手动的方式来新建一个自己的工程，步骤如下：

- (1) 在 VS 2010 环境下，单击 File>New Project 选项，弹出如图 8-1 所示的 New Project 对话框。在 Project Type 列表框中选择 Visual C# 的 Windows 选项，在 Templates 列表框中选择 Empty Project 选项，指定一个存放该工程的目录。然后将工程命名为“mumu_manualapplication”，单击 OK 按钮。
- (2) 新建的工程中为空，在 Solution Explorer 窗口中右击 Reference 选项添加引用。WPF 所需要的引用主要是 System、WindowsBase、PresentationCore、PresentationFramework 和 System.Xaml¹，右键添加这 5 个程序集引用，如图 8-2 所示。
- (3) 右击 mumu_manualapplication 选项，选择快捷菜单中的 Add>New Item 选项，弹出如图 8-3 所示的 Add New Item 对话框。选择 Code File 选项，键入代码文件名“ManualApplication.cs”，单击 OK 按钮。

¹ 在 WPF 4.0 以前，只需要引用 4 个程序集，即 System、WindowsBase、PresentationCore 和 PresentationFramework。

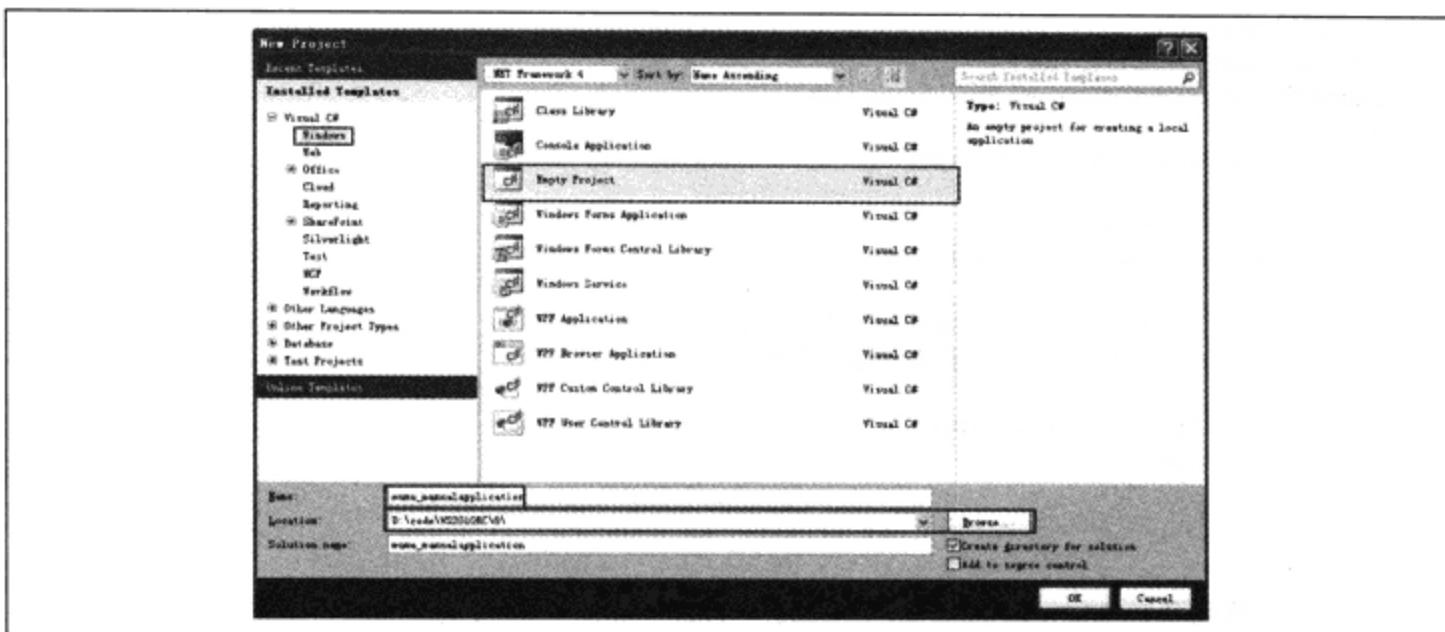


图 8-1 New Project 对话框

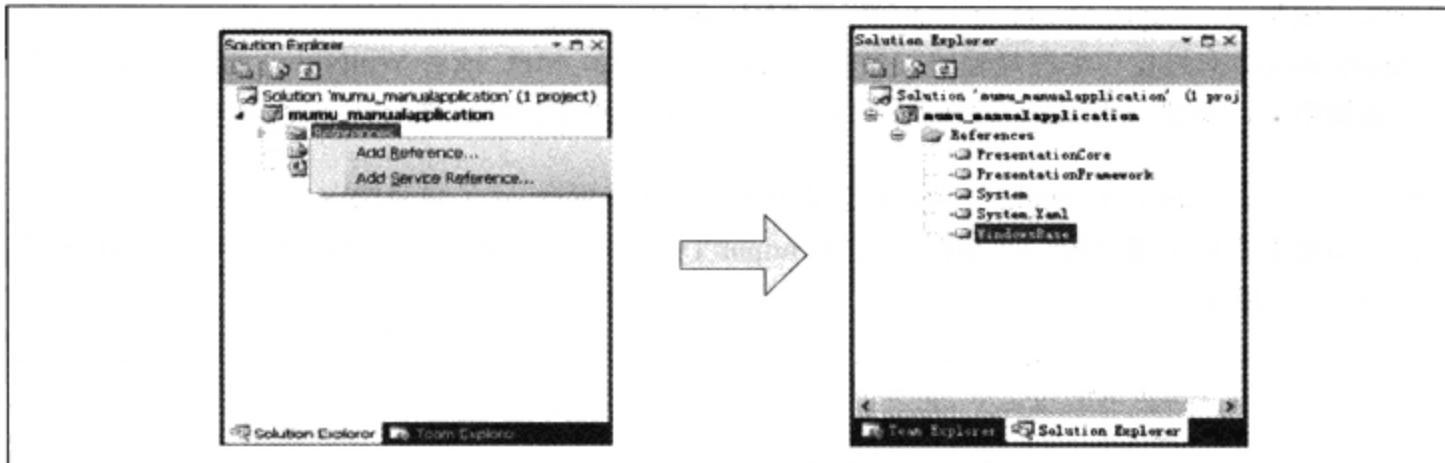


图 8-2 添加 WPF 引用

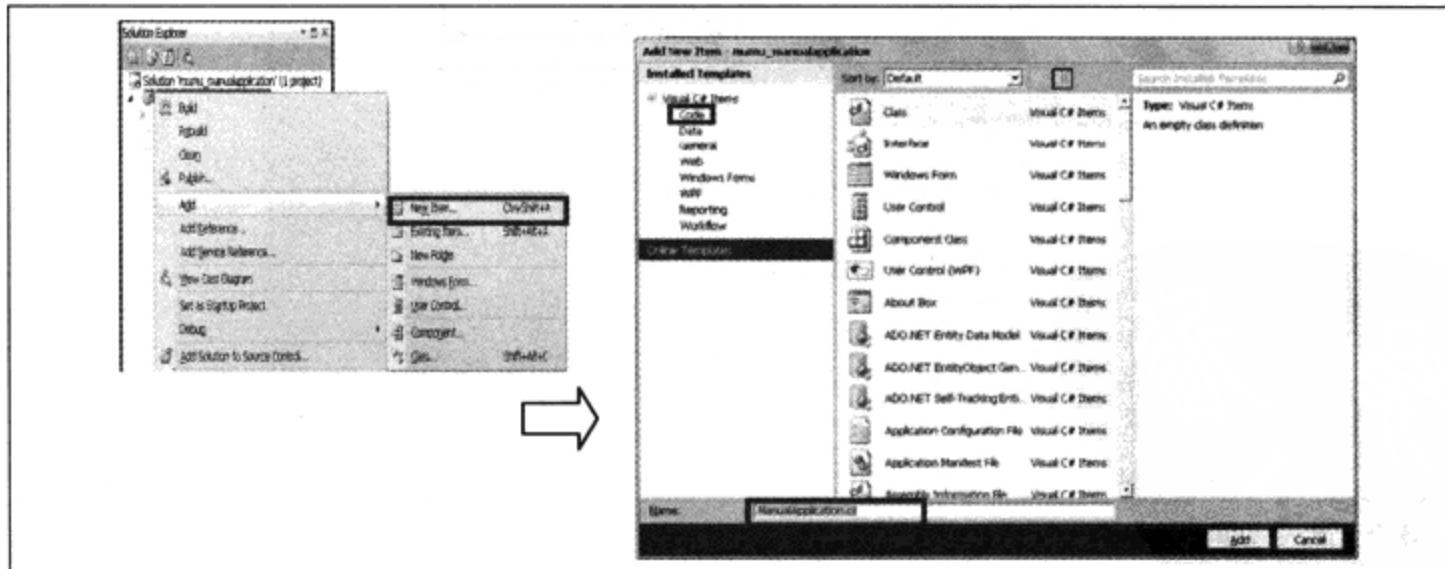


图 8-3 新建代码文件

(4) 在 ManualApplication.cs 文件中键入代码 8-1 所示的代码。

```

using System;
using System.Windows;
namespace mumu_manualapplication
{
    class ManualApplication
    {
        [STAThread]
        public static void Main()
        {
            Window win = new Window();
            win.Title = "手动创建一个应用程序";
            win.Show();
            Application app = new Application();
            app.Run();
        }
    }
}

```

代码 8-1 ManualApplication.cs 文件（详见 mumu_manualapplication 工程）

Main 的前面必须有一个[STAThread]属性；否则在运行时会抛出异常，这个属性用来声明该应用程序的线程模型为“single thread apartment”。Main 函数里面两个重要的类是窗口 Window 和应用程序 Application。有过窗口编程经验的程序员（无论是 Win32，MFC 或者 WinForm）都不会对这几句代码感到陌生，新建一个窗口，然后通过 Run 方法建立起消息循环，接受外部用户的输入。

(5)选择 Debug|Start Debugging 选项，或者按 F5 键，运行该程序。除了自身的窗口，还有一个 Console 窗口。如果不希望出现该窗口，则在工程的 Output Type 下拉列表框中选择 Windows Application 选项，如图 8-4 所示。

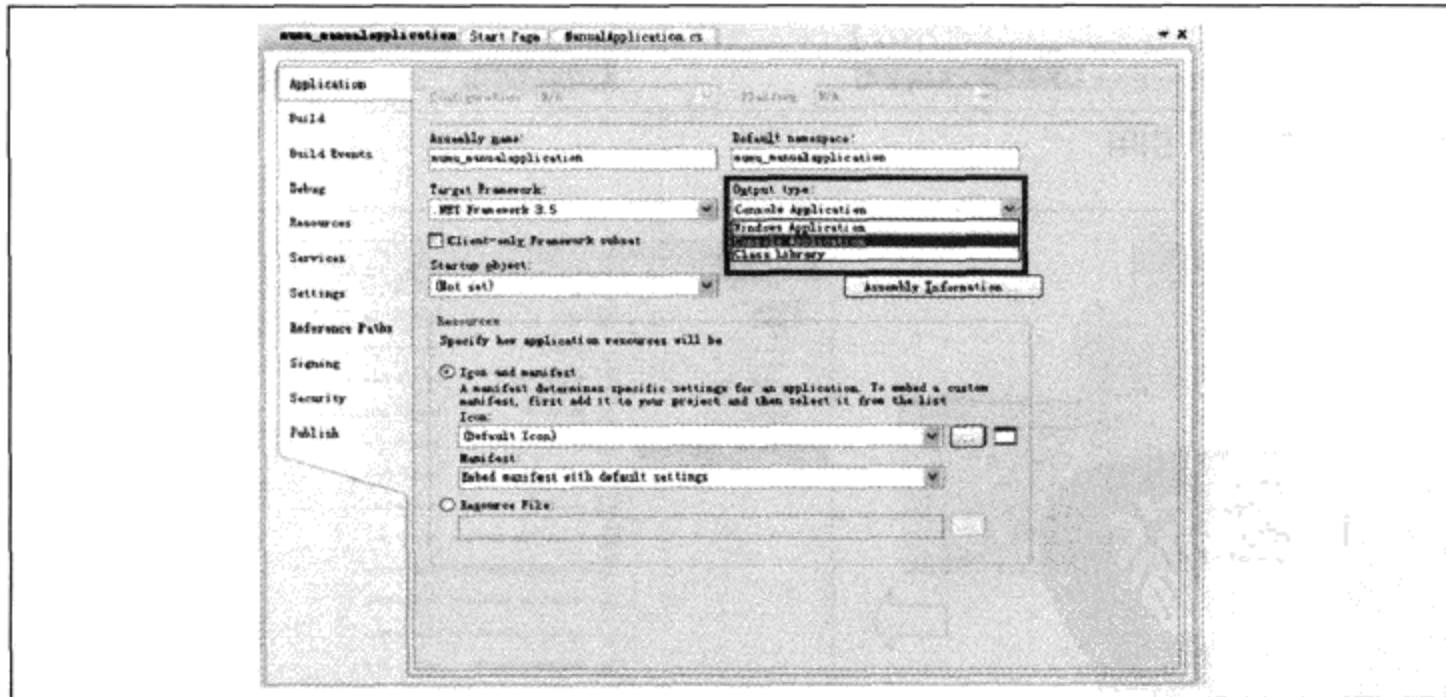


图 8-4 选择 Windows Application 选项

在开发阶段 Console 窗口相当有用，当程序出现问题导致 GUI 窗口无法显示时可以利用该窗口输出一些调试信息，也可以通过关闭该窗口来关闭整个应用程序。

8.1.2 使用向导创建

使用向导创建 WPF 应用程序会自动添加所需的引用，同时生成两个成对的文件，即 App.xaml 和 App.xaml.cs，以及 MainWindow.xaml 和 MainWindow.xaml.cs。它们分别代表前面的 Application 和 Window 类型，如图 8-5 所示。

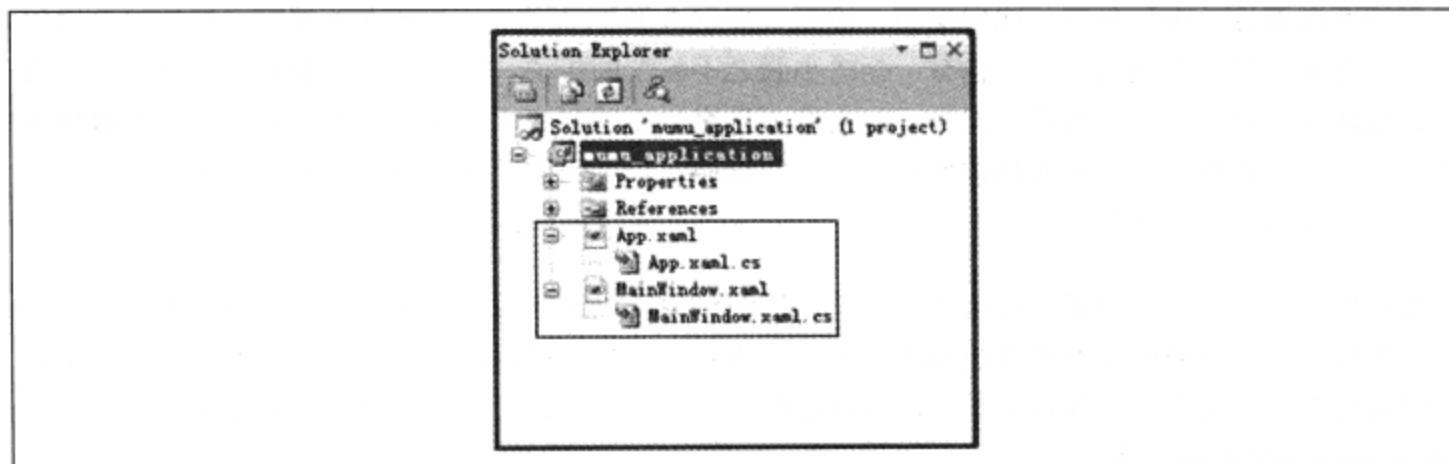


图 8-5 WPF 应用程序下的两对文件

WPF 自动生成了一些代码，这些代码在工程的\obj\Debug 目录下且后缀名为“.g.cs”的文件中。Main 函数的实现在 App.g.cs 文件中，如代码 8-2 所示：

```
public partial class App : System.Windows.Application {  
    ....  
    [System.STAThreadAttribute()]  
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]  
    public static void Main() {  
        mumu_application.App app = new mumu_application.App();  
        app.InitializeComponent();  
        app.Run();  
    }  
}
```

代码 8-2 App.g.cs 文件（详见 mumu_application 工程）

8.2 应用程序及其生命周期

WPF 应用程序的重要两个类型是 Application 和 Window，前者在一个应用程序中是全局唯一的，代表一个应用程序。它可以提供很多基础的应用程序级的服务，应用程序也有其生命周期。

8.2.1 小强的成长路线图

我们在考察应用程序的生命周期之前，不妨先看看武侠小说中一个默默无闻的愣头小子是如何成长为大侠的。

一个命中注定要成为拯救世界的大侠客，我们不妨称他为小强吧。刚一出场往往会有个特殊事件来揭开序幕，比如从小父母被坏人杀害，不幸得了不治之症，或者是出身皇族却厌倦了奢华的生活。

再不济也得出身奇异，比如从小生活在妓院，突然有一天碰到了一个身受重伤之人等。

经历了这个特殊事件之后，小强开始闯荡江湖了。在江湖上只有两种状态，一种是打斗；另外一种是准备打斗，即修炼的过程。在这个循环过程中，小强打斗——失败——修炼——再打斗……自己的功力不断提升。

当然闯荡江湖也是很危险的，比如受伤、中毒、被别人推下悬崖，或者被女人骗等，往往这个时候小强会奄奄一息。当然武侠之中也有一个“小强之不死”的传说，男主角无论怎么受尽折磨，都会神奇地恢复，同时功力得到更大提升。被人推下悬崖，要感到庆幸，因为很有可能不是一个猿猴掏出一本经书让你修炼，就是遇到神仙姐姐这样的奇遇。当小强恢复了之后，又重新进入了打斗——失败——修炼——再打斗的循环。

当然如果主角不是小强的话，那么“小强不死”的传说就无法应验。小强很有可能面临的是退出江湖这个舞台，这种退出是不正常的退出。而与之相对应的是一种正常的退出，这种退出往往是小强已经成长为了一代大侠，拯救了世界，并且抱得了美人归。从此厌倦了江湖，归隐于田园之中，图 8-6 所示为小强的成长路线。

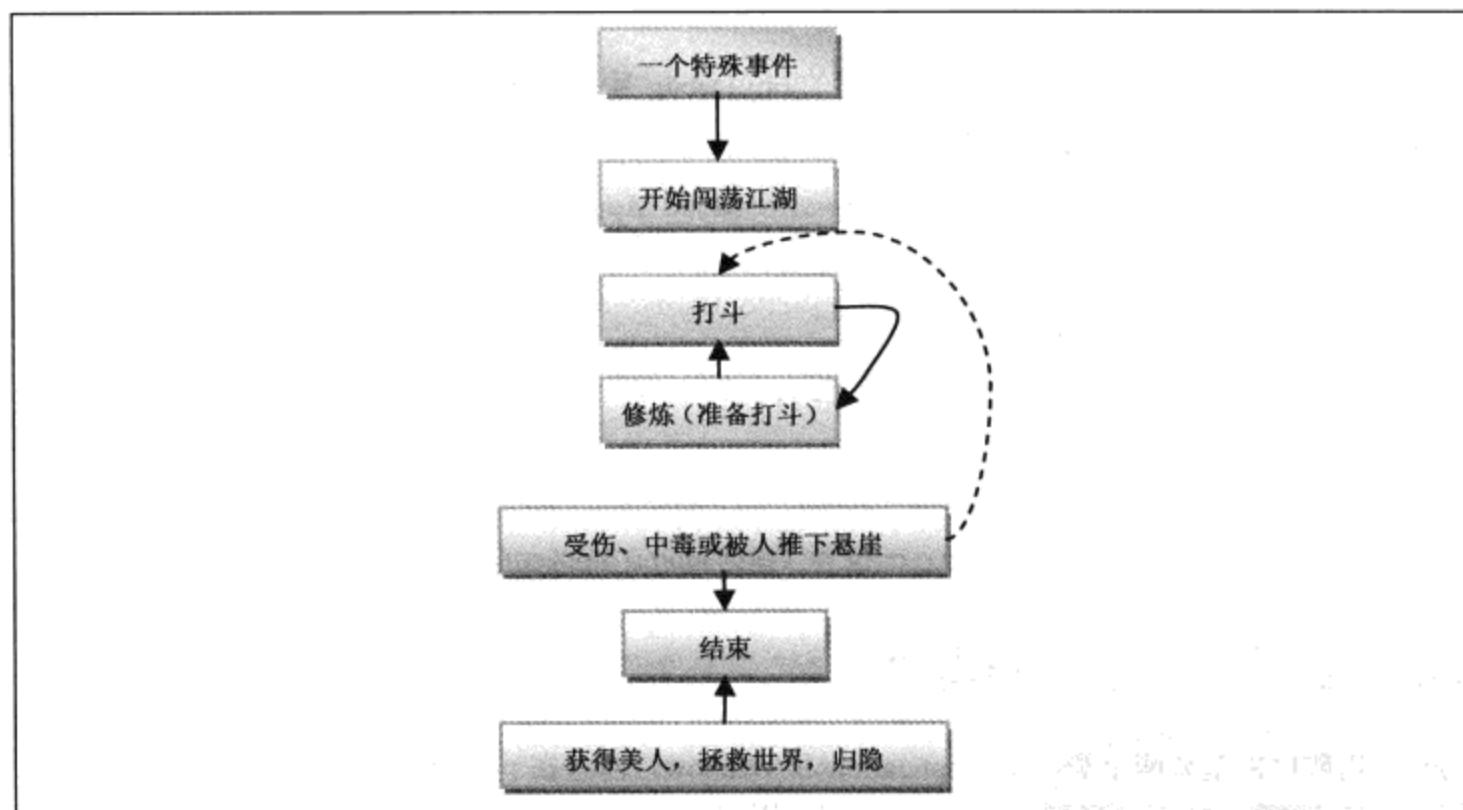


图 8-6 小强的成长路线

8.2.2 应用程序的生命周期

应用程序的生命周期和小强的成长路线图非常类似。应用程序的启动一般情况下是通过用户双击应用程序，由操作系统启动应用程序，当运行了 Run 函数之后，应用程序会触发一个 Startup 事件，就进入了“江湖”。它会一直监听鼠标，键盘或者手写笔的收入。应用程序也有两种状态，一种是激活（Activated）状态，另一种是非激活（Deactivated）状态。正常情况下，应用程序是通过用户关

闭而退出，但有的时候会出现各种各样的异常情况，导致应用程序异常退出。但应用程序不是小强，因此它没有“小强不死”的传说，但是 WPF 提供了一种捕捉异常的机制，因此当出现异常的时候，可以选择是继续“混迹江湖”还是“归隐田园”。图 8-7 所示为应用程序的生命周期。

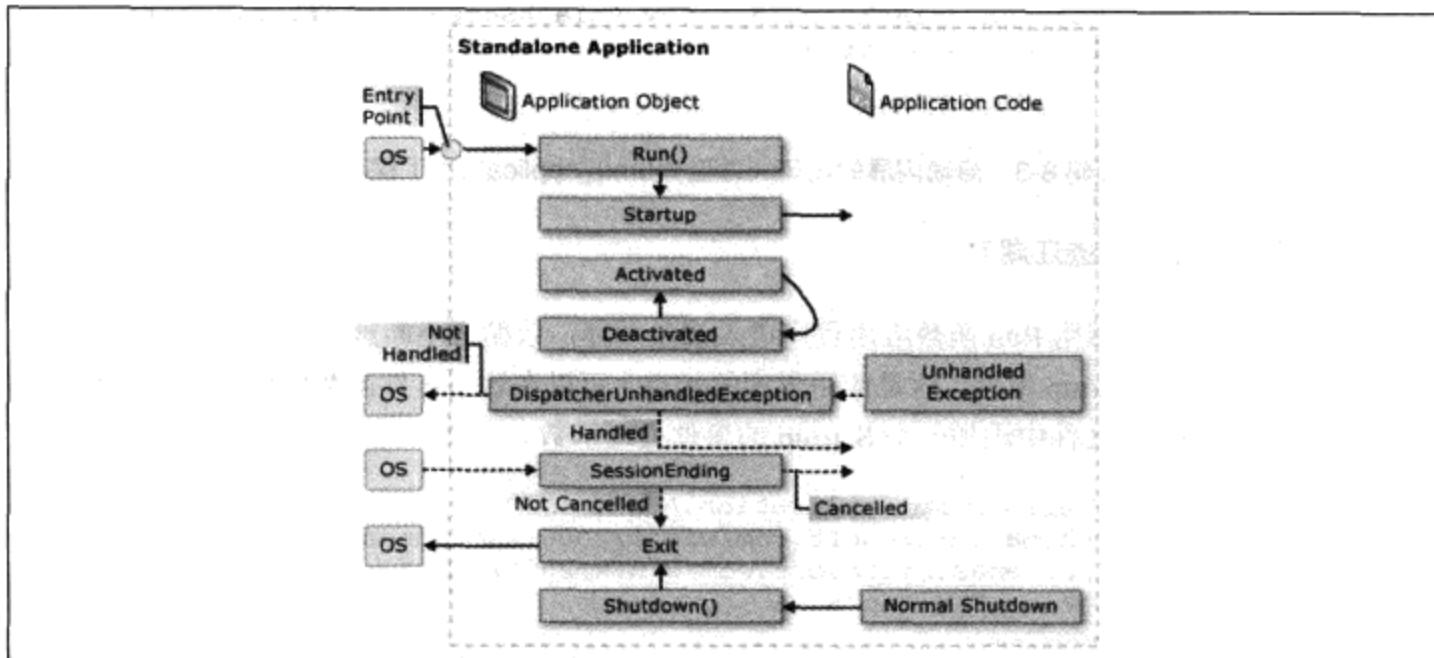


图 8-7 应用程序的生命周期^[1]

1. 应用程序入口（江湖前传）

应用程序的入口都是 Main 函数，其中两种主要类型的对象是 Application 和 Window。前者在一个程序中是全局唯一的，可以通过 Application.Current 来获得当前的 Application 对象，第 1 个创建的窗口对象被认为是主窗口。

在 Application 调用 Run 之前，首先创建窗口对象。而且设置了一些必要的属性，如前例中的 Title 属性。也可以在应用程序启动之初做好一个闪屏，为此在.NET 3.5 之后，只需要添加一个图片。选中图片右键选择属性，会在属性窗口中罗列出该图片的所有属性。将其 Build Action 属性设置为 SplashScreen，如图 8-8 所示。

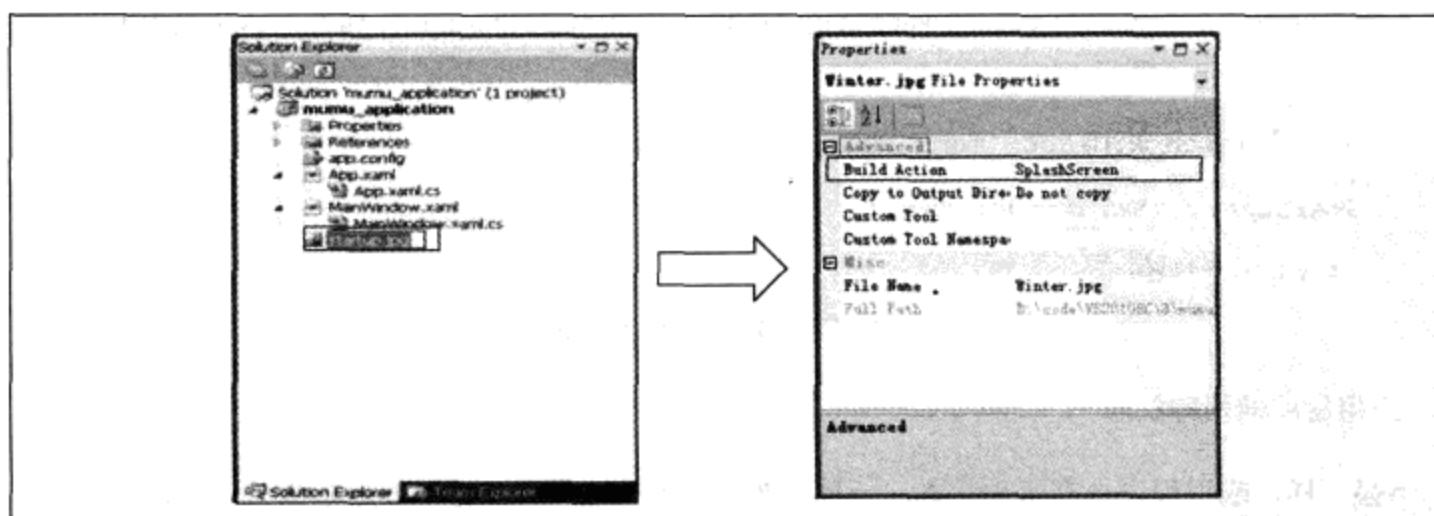


图 8-8 将图片的 Build Action 属性设置为 SplashScreen

实际上如果我们查看 WPF 自动生成的 App.g.cs 会发现多了两行出现闪屏的代码(如代码 8-3 所示):

```
[System.STAThreadAttribute()]
[System.Diagnostics.DebuggerNonUserCodeAttribute()]
public static void Main()
{
    SplashScreen splashScreen = new SplashScreen("startup.jpg");
    splashScreen.Show(true);
    ...
}
```

代码 8-3 启动闪屏的代码 (详见 mumu_application 工程)

2. 应用程序的起点 (混迹江湖)

如前所述, Application 调用 Run 函数应用程序进入消息循环, 该循环不断地响应并处理事件。应用程序的第一个事件是 Startup 事件, 这是应用程序的起点, 可以在这个事件函数中创建窗口。如代码 8-4 所示, 在 App.xaml 文件中添加一个 Startup 的事件处理函数。

```
<Application x:Class="mumu_application.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Startup="Application_Startup">
<Application.Resources>
</Application.Resources>
</Application>
```

代码 8-4 在 App.xaml 文件中添加 Startup 的事件处理函数

在相应的 App.xaml.cs 文件中实现该事件处理函数, 在其中创建主窗口并显示, 如代码 8-5 所示。

```
public partial class App : Application
{
    private void Application_Startup(object sender, StartupEventArgs e)
    {
        MainWindow win = new MainWindow();
        win.Show();
    }
}
```

代码 8-5 在 Startup 事件处理函数中创建主窗口

更为简单的方法是将窗口的 xaml 文件赋值给 Application 对象的 StartupUri 属性, 这也是 WPF 的默认做法, 如代码 8-6 所示。

```
<Application x:Class="mumu_application.App"
  ...
  StartupUri="MainWindow.xaml">
</Application>
```

代码 8-6 通过 StartupUri 启动主窗口

3. 应用程序的两种状态

和小强一样, 应用程序也有两种状态: 一种是激活 (Activated); 另一种是非激活 (Deactivated) 状态。一般来说, 在桌面上只有一个窗口处在激活状态, 其他窗口都处在非激活状态。激活窗口后

可以接受用户的输入，拥有激活窗口的应用程序即激活状态，而其他应用程序则处在非激活状态。

用户可以通过 Alt+Tab 键来切换应用程序的状态，也可以通过选中应用程序的窗口来激活应用程序。WPF 里可以通过检查 Activated 和 Deactivated 事件来判断应用程序处于何种状态，如代码 8-7 所示，在 App.xaml 文件中添加这两个事件的处理函数。

```
<Application x:Class="mumu_application.App"
    ...
    StartupUri="MainWindow.xaml" Activated="Application_Activated"
    Deactivated="Application_Deactivated">
    ...
</Application>
```

代码 8-7 Activated 和 Deactivated 事件

在相应的 App.xaml.cs 文件中分别实现这两个事件的处理函数，如代码 8-8 所示。

```
public partial class App : Application
{
    private void Application_Activated(object sender, EventArgs e)
    {
        MessageBox.Show("activated");
    }

    private void Application_Deactivated(object sender, EventArgs e)
    {
        MessageBox.Show("deactivated");
    }
}
```

代码 8-8 Activated 和 Deactivated 事件处理函数

4. 异常情况

应用程序不可能没有 bug，过去异常情况的出现往往会导致程序死机，甚至伴随刺耳的声音和显示红色感叹号的对话框。

WPF 在程序出错时会触发一个 DispatcherUnhandledException 事件，使程序响应该事件以做出相应的处理。如在窗口中添加两个按钮的 Click 事件处理函数，在其中有意制造两个错误分别用于抛出异常和显示一个值为 null 的窗口。如代码 8-9 所示。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    throw new DivideByZeroException("Recoverable Exception");
}
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Window win = null;
    win.Show();
}
```

代码 8-9 有意制造出来的两个错误

在 App.xaml 文件中添加处理异常的事件响应函数，如代码 8-10 所示。

```
<Application x:Class="mumu_application.App"
...
    DispatcherUnhandledException="Application_DispatcherUnhandledException">
</Application>
```

代码 8-10 在 App.xaml 文件中添加 DispatcherUnhandledException 事件

同理，仍然在 App.xaml.cs 文件中实现该函数，在其中将 DivideByZeroException 视为可以恢复的异常，而其他异常需要应用程序立刻退出。开始将传递过来类型为 DispatcherUnhandledEventArgs 参数 e 的 Handled 属性设置为 true；否则 WPF 会认为没有处理任何异常，而仍然会按照默认的行为来处理这些异常。即让应用程序退出，甚至在退出时还弹出一个极不友好的对话框，如代码 8-11 所示。

```
private void Application_DispatcherUnhandledException(object sender,
System.Windows.Threading.DispatcherUnhandledEventArgs e)
{
    e.Handled = true;
    if (e.Exception is DivideByZeroException)
    {
        MessageBox.Show("嗨，兄弟你遇到麻烦了！不过还好，程序仍可以恢复运行！");
    }
    else
    {
        MessageBox.Show("嗨，兄弟你遇到大麻烦了！程序马上退出。啊~~~");
        this.Shutdown(-1);
    }
}
```

代码 8-11 DispatcherUnhandledException 事件处理函数的实现

运行该程序，单击“可以恢复的异常”按钮后弹出一个对话框，应用程序仍然正常运行；单击“不可以恢复的异常”按钮后弹出一个对话框，提示“嗨，兄弟你遇到大麻烦了！程序马上退出。啊~~~”。然后应用程序立刻退出，如图 8-9 所示。

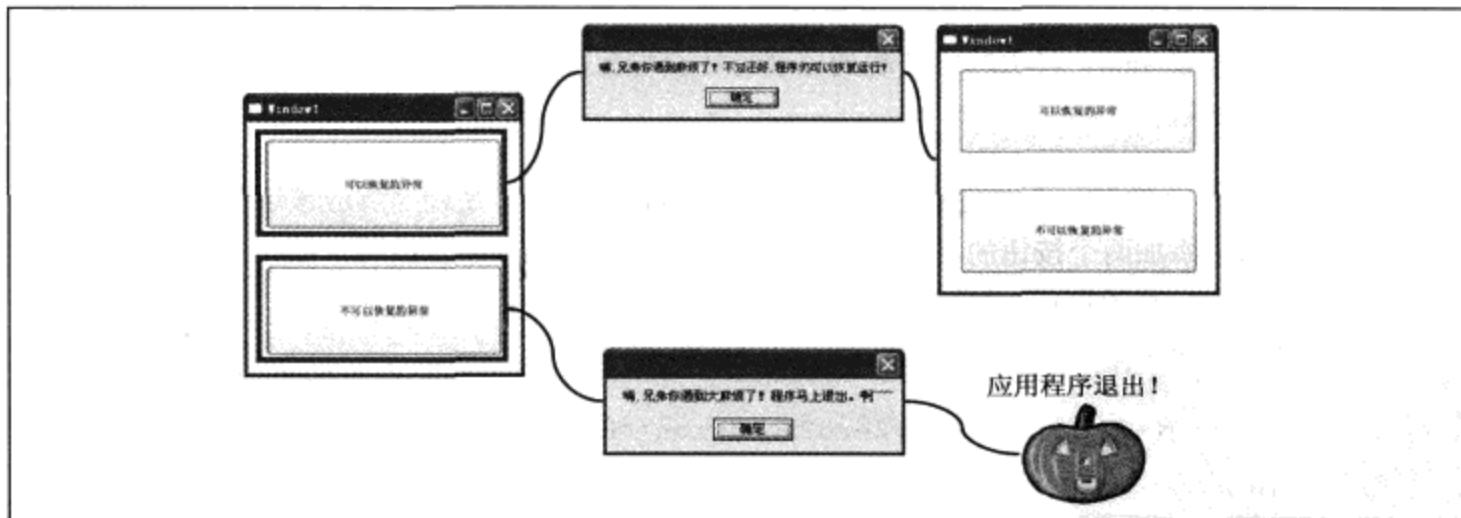


图 8-9 可以恢复和不可以恢复的异常

还有一种非正常的情况，即关机、重启或者注销当前用户时应用程序也会退出。这种情况发生时应用程序会接受一个 SessionEnding 事件，该事件的处理函数中类型为 SessionEndingCancelEventArgs 的参数 e 包含 Session 结束的原因信息（ReasonSessionEnding 属性）。如果用户不希望操作系统关机，重启或者注销的话，还可以将该参数 e 的 Cancel 属性设置为 true。在代码 8-12 所示的事件处理

函数中，弹出了一个提示对话框。如果用户选择 No，则将 Cancel 属性设置为 true，阻止操作系统关机、重启或者注销。

```
<Application x:Class="mumu_application.App"
...
SessionEnding="Application_SessionEnding">

    private void Application_SessionEnding(object sender,
SessionEndingCancelEventArgs e)
    {
        string msg = string.Format("{0}. End session?", e.ReasonSessionEnding);
        MessageBoxResult result = MessageBox.Show(msg, "Session Ending",
MessageBoxButton.YesNo);
        if (result == MessageBoxResult.No)
        {
            e.Cancel = true;
        }
    }
}
```

代码 8-12 SessionEnding 事件的处理函数实现

5. 应用程序正常退出（归隐田园）

应用程序可以在以下 3 种情况下正常退出，即关闭主窗口、关闭应用程序的所有窗口及显式调用 Shutdown 函数。这 3 种情况由 Application 的 ShutdownMode 属性决定。该属性的如下 3 种值分别对应上述 3 种情况。

- (1) OnMainWindowClose：表示只有关闭主窗口时应用程序才会退出。
- (2) OnLastWindowClose：表示关闭应用程序的所有窗口时应用程序才会退出。
- (3) OnExplicitShutdown：表示显式调用 Shutdown 函数应用程序才会退出。

Shutdown 函数有两种形式：一是不用传入任何参数；二是需要传入一个 int 类型的值，这个值作为一个退出代码（Exit Code）。

为什么应用程序在退出的时候需要一个 Exit Code 呢？

一般情况下应用程序都是通过用户交互，然后由操作系统启动。但是也有一种情况，是一个应用程序（launching application）启动另一个应用程序（launched application）。用户通过眼睛观察就可以知道应用程序是因为什么原因退出的。但是对于 launching application 来说，它只能通过返回的代码才能知道退出的原因，以便它做出正确的反应。为此需要 Exit Code。通常正常退出其默认值为 0。

8.3 窗口

窗口几乎是一个永恒的话题，用户和应用程序交互实际上通过与一个个窗口交互完成。

8.3.1 窗口组成

首先来分析一下窗口的组成，如图 8-10 所示。

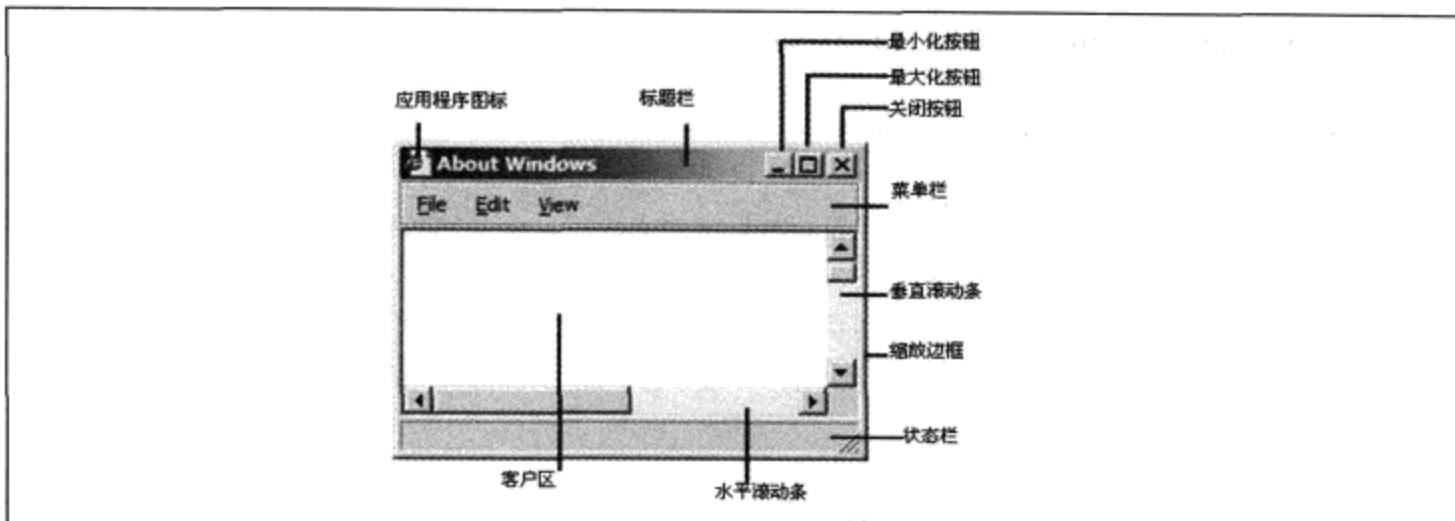


图 8-10 窗口组成

窗口中有多个界面元素，如标题栏、菜单栏和状态栏等。但是可以把这些元素归为两类：一是客户区，即窗口的中间部分，客户区由开发人员负责绘制，可以在其中显示交互文本或图形；二是非客户区，包括标题栏和水平滚动条，这一部分由操作系统负责绘制。

8.3.2 窗口的生命周期

WPF 中的一个 Window 类代表一个窗口，窗口也有其生命周期。

与窗口有关的事件如下。

- (1) **SourceInitiated:** 窗口的第一个事件。
- (2) **Activated:** 一般说来，在默认情况下创建的窗口是激活状态，因此该事件会紧接其后。
- (3) **Loaded:** 表示已经初始化窗口。
- (4) **ContentRended:** 如果一个窗口的 Content 属性为空或者客户区中没有任何内容，不会触发该事件。但是这种情况在实际应用中很少见，因此往往都会触发该事件。
- (5) **Deactivated:** 当窗口处在非激活状态时触发该事件。
- (6) **Closing:** 该事件发生在窗口关闭之前，也可以通过处理该事件阻止关闭窗口。
- (7) **Closed:** 该事件标识一个窗口生命周期结束。

窗口的生命周期如图 8-11 所示。

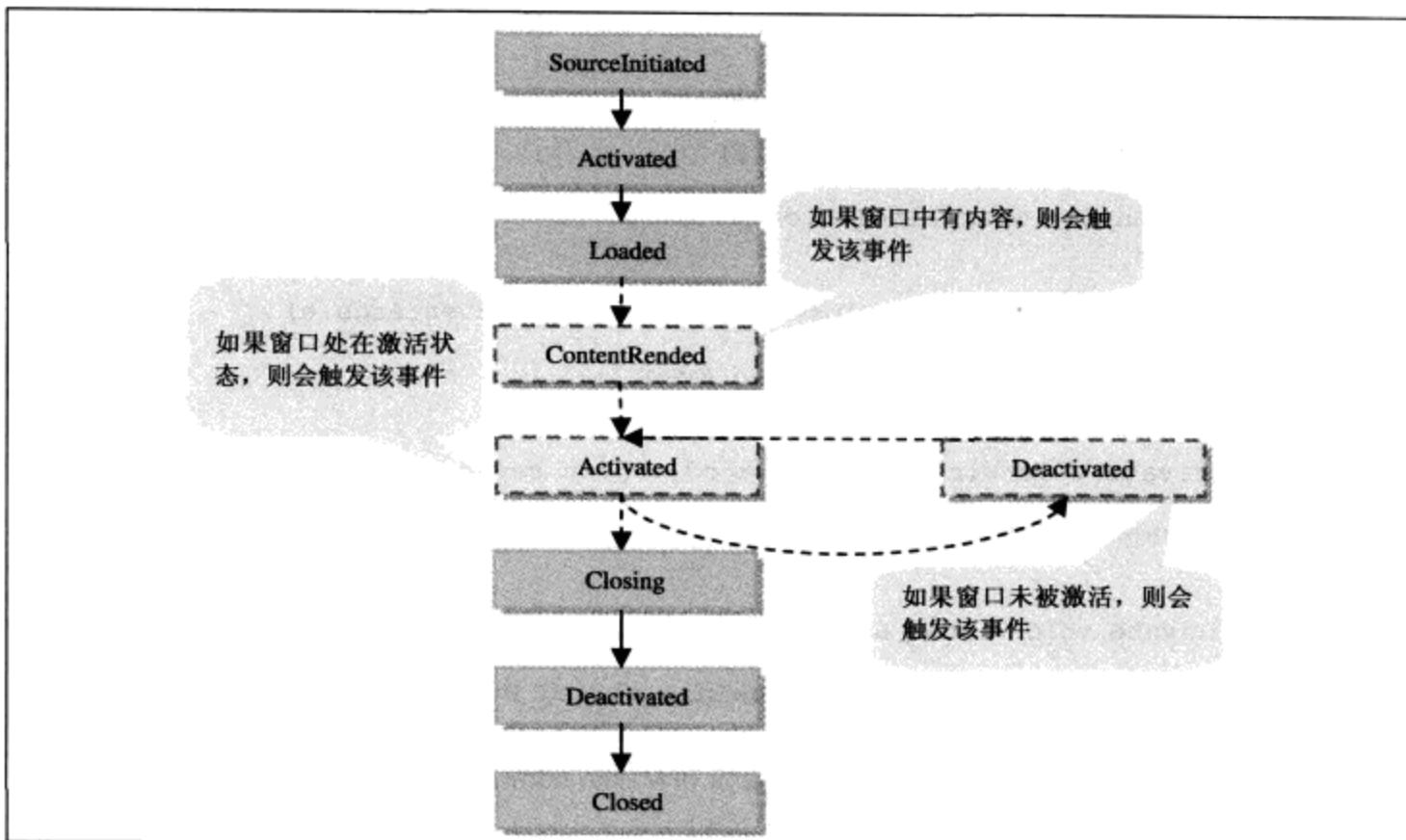


图 8-11 窗口的生命周期^[2]

观察上述一系列事件的最好方法是为这些事件添加事件处理函数，然后在命令窗口中输出字符串观察其调用的顺序，如代码 8-13 所示。

```

MainWindow.xaml
<Window x:Class="mumu_application.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Window1" Height="300" Width="300"
SourceInitialized="Window_SourceInitialized"
Activated="Window_Activated"
Deactivated="Window_Deactivated"
Loaded="Window_Loaded"
ContentRendered="Window_ContentRendered"
Closing="Window_Closing"
Closed="Window_Closed"
>
.....

```

```

MainWindow.xaml.cs
.....
private void Window_Activated(object sender, EventArgs e)
{
    Console.WriteLine("Activated Event");
}

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    Console.WriteLine("Loaded Event");
}

private void Window_ContentRendered(object sender, EventArgs e)
{
}

```

```

        Console.WriteLine("ContentRendered Event");
    }

    private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    Console.WriteLine("Closing Event");
}

private void Window_Closed(object sender, EventArgs e)
{
    Console.WriteLine("Closed Event");
}

private void Window_Deactivated(object sender, EventArgs e)
{
    Console.WriteLine("Deactivated Event");
}

private void Window_SourceInitialized(object sender, EventArgs e)
{
    Console.WriteLine("SourceInitialized Event");
}

```

代码 8-13 MainWindow.xaml 和 MainWindow.xaml.cs 文件

注意需要将应用程序的输出类型设置为 Console Application，这样才会显示命令窗口。我们可以在该窗口中观察事件调用的顺序，也可以在 Closing 的事件处理函数中通过设置 e.Cancel 的属性为 true 来阻止窗口关闭，如代码 8-14 所示。

```

private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
    Console.WriteLine("Closing Event");
}

```

代码 8-14 在 Closing 事件处理函数中阻止窗口关闭

图 8-12 所示为从命令窗口中显示的事件调用顺序。

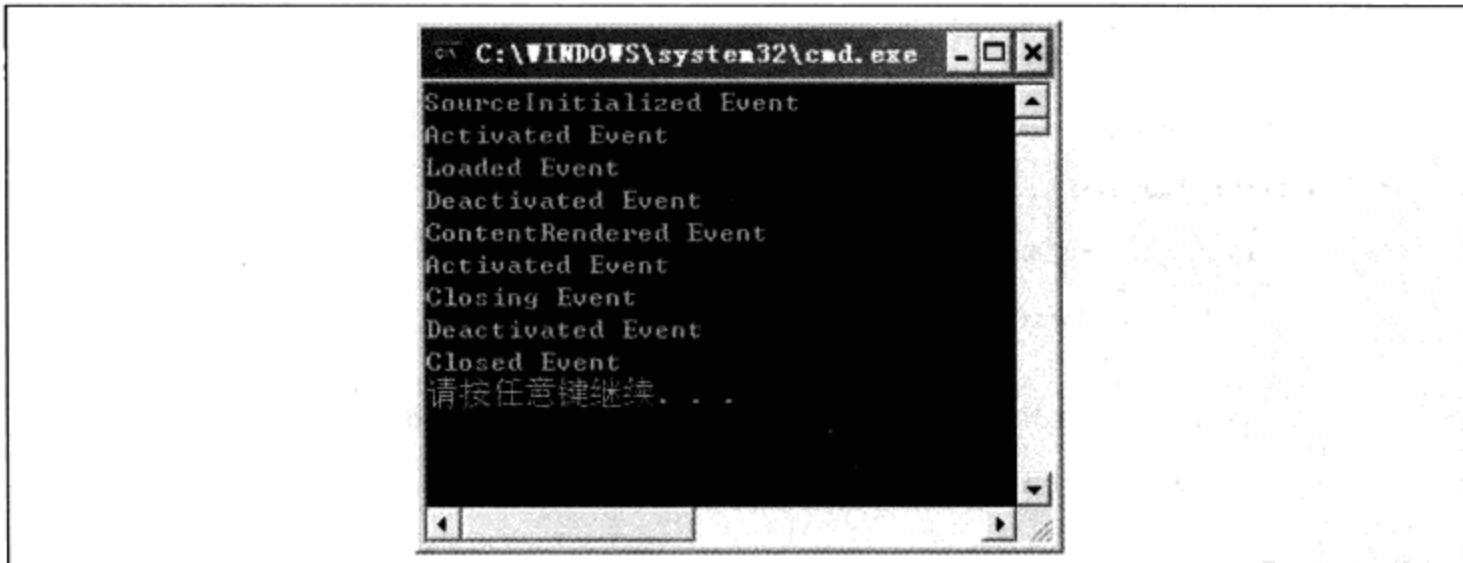


图 8-12 事件调用顺序

8.3.3 窗口属性

窗口的属性可以分为位置和尺寸，以及外观和样式两大类。为了了解窗口属性对窗口的影响，编写一个名为“窗口工厂”的程序（详见 `mumu_windowfactory` 工程）。如图 8-13 所示，主窗口中罗列了窗口的各种属性。所有的属性都有默认值，因此可以选择设置或不设置。当设置后单击“创建窗口”按钮，即可按照所设置的属性创建窗口，创建的子窗口的客户区中显示该窗口的各种属性值。

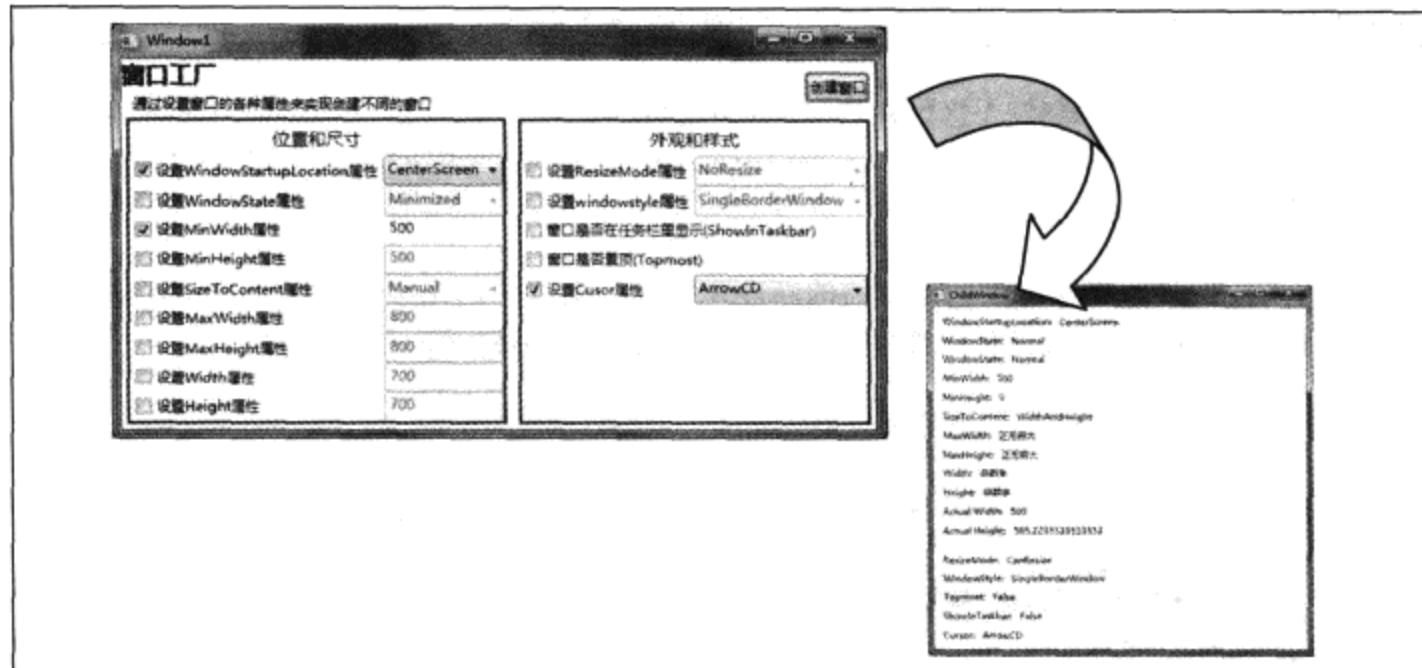


图 8-13 窗口的属性

1. 与位置和尺寸相关的属性

表 8-1 所示为与窗口位置和尺寸的相关属性。

表 8-1 与窗口位置和尺寸的相关属性

WindowStartupLocation	描述窗口弹出的位置，它的 3 个枚举值是 Manual（默认）、CenterScreen 和 CenterOwner。为 Manual，表示窗口位置由其 Left 和 Top 属性决定。如果未设置这两个属性，WPF 会提供一个默认的值：为 CenterScreen，窗口显示在屏幕中心。为 CenterOwner，窗口显示在父窗口的中心。注意如果未设置窗口的 Owner 属性，WPF 会默认认为所有窗口的父窗口为桌面窗口
WindowState	描述窗口的状态，3 个枚举值是 Normal（默认）、Minimized 和 Maximized，为 Normal，窗口处于正常状态；为 Minimized，窗口弹出时处于最小化状态，即在任务栏中显示；为 Maximized，窗口弹出时处于最大化状态，即占据整个桌面大小
MinWidth 和 MinHeight	指定窗口最小的尺寸
MaxWidth 和 MaxHeight	指定窗口最大的尺寸

SizeToContent	描述窗口大小是否根据其中内容调整，它的4个枚举值是Manual（默认）、Width、Height和WidthAndHeight。为Manual，窗口不会自动根据内容来调整；为Width，根据内容来调整宽度，但是高度不变；为Height，根据内容来调整高度，但是宽度不变；为WidthAndHeight，根据内容调整高度和宽度
Width 和 Height	窗口的宽度和高度
ActualWidth 和 ActualHeight	两个只读属性，表示窗口的实际宽度和高度

与窗口宽度和高度有关的属性不仅仅有Width和Height，还有ActualWidth和ActualHeight。这是因为如果设置Width和Height为700，但是窗口的宽度和高度不一定为700，只有通过后两个只读属性才能获得窗口的实际宽度和高度。

有多个不同优先级的属性可以确定窗口的高度和宽度，宽度的优先级为MinWidth>MaxWidth>SizeToContent>Width；高度的优先级为MinHeight>MaxHeight>SizeToContent>Height。

利用“窗口工厂”程序来观察这几个属性的优先级，以窗口宽度为例。

(1) 将窗口的Width属性设置为700，其他所有属性均保持默认值，如图8-14所示。

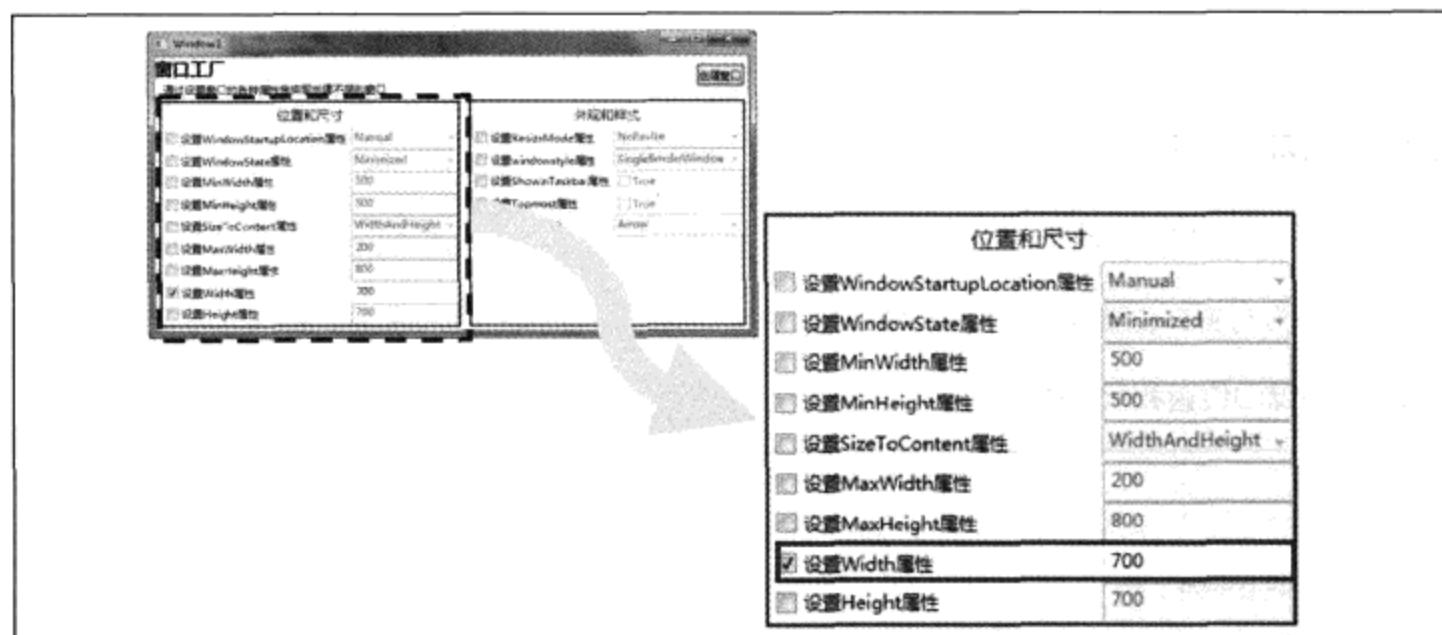


图8-14 窗口的Width属性设置700

可以看到弹出窗口的Width和ActualWidth属性均为700。如图8-15所示。

(2) 保持窗口的Width属性为700，设置SizeToContent属性为Width或者是WidthAndHeight。如图8-16所示。

这时弹出窗口的实际宽度是根据内容调整后的值，这里为249.8。可以看到Width的值仍为700，但是ActualWidth已经为249.8，说明SizeToContent的优先级高于Width (SizeToContent>Width)。如果将SizeToContent值设置为Height或者Manual，则窗口的宽度仍然由Width的值来决定，如图8-17所示。

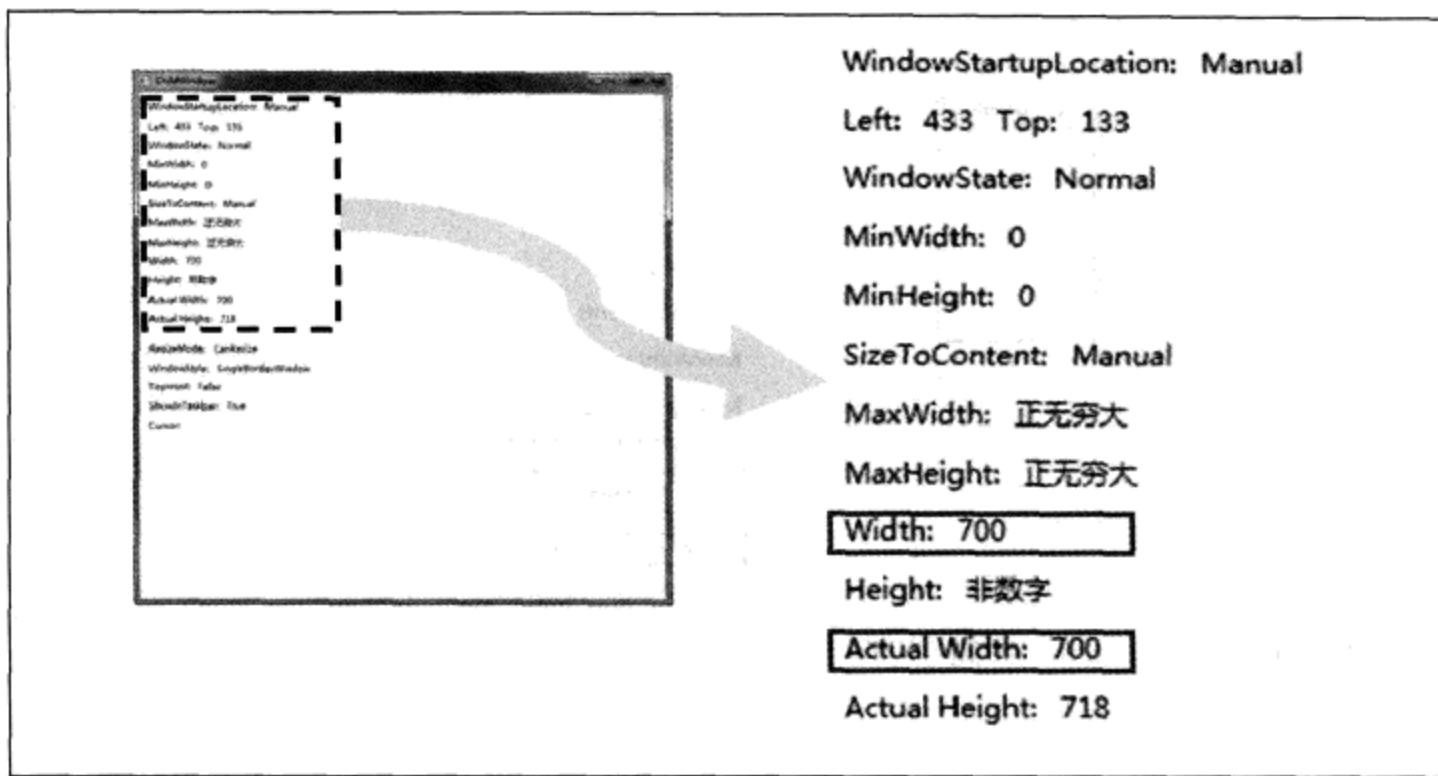


图 8-15 窗口的 Width 和 Actual Width 属性均为 700

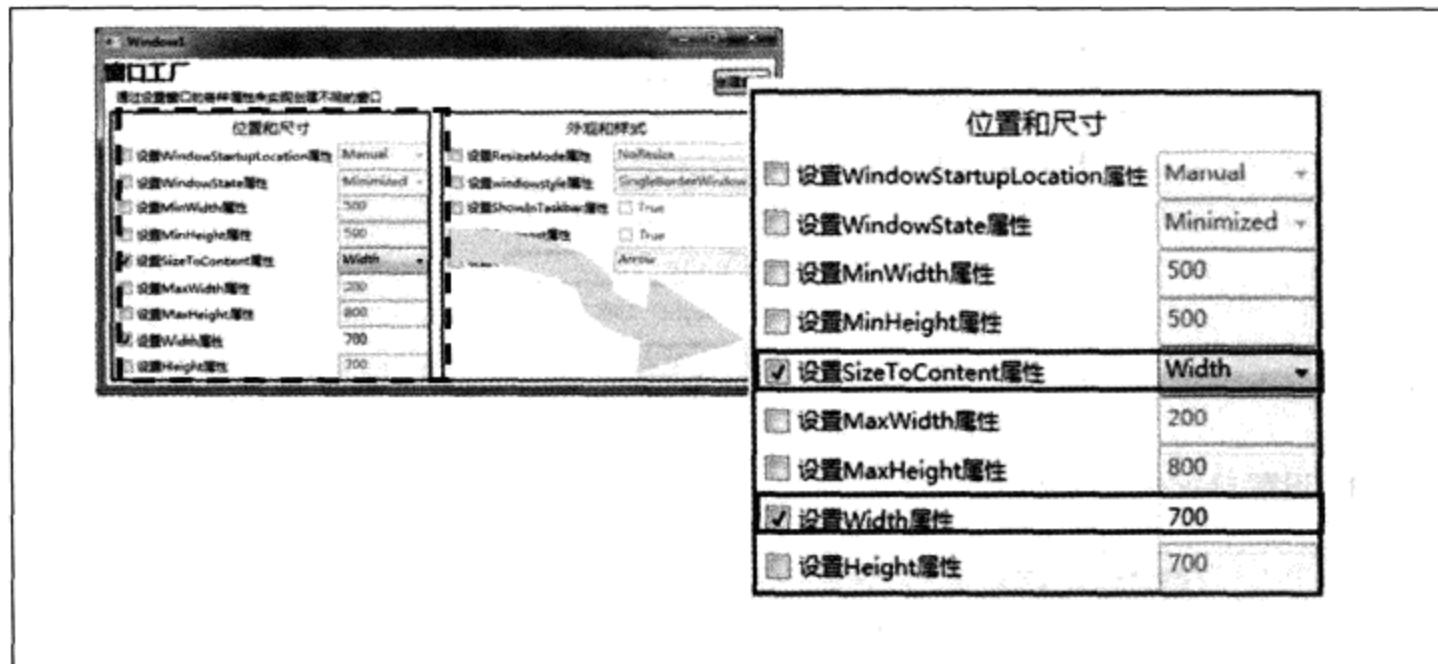


图 8-16 设置 Width 和 SizeToContent 属性

(3) 保持前面设置的属性，将 MaxWidth 属性设置为 200。尽管窗口需要至少 249.8 的宽度才能显示全部内容，但是其实际宽度只能为 200。这是因为 MaxWidth 的优先级高于 SizeToContent，如图 8-18 所示。

(4) 保持前面设置的属性，将 MinWidth 属性设置为 500。尽管这样的设置不合理，但是可以观察到弹出窗口的宽度变为 500。说明 MinWidth 的优先级最高，如图 8-19 所示。

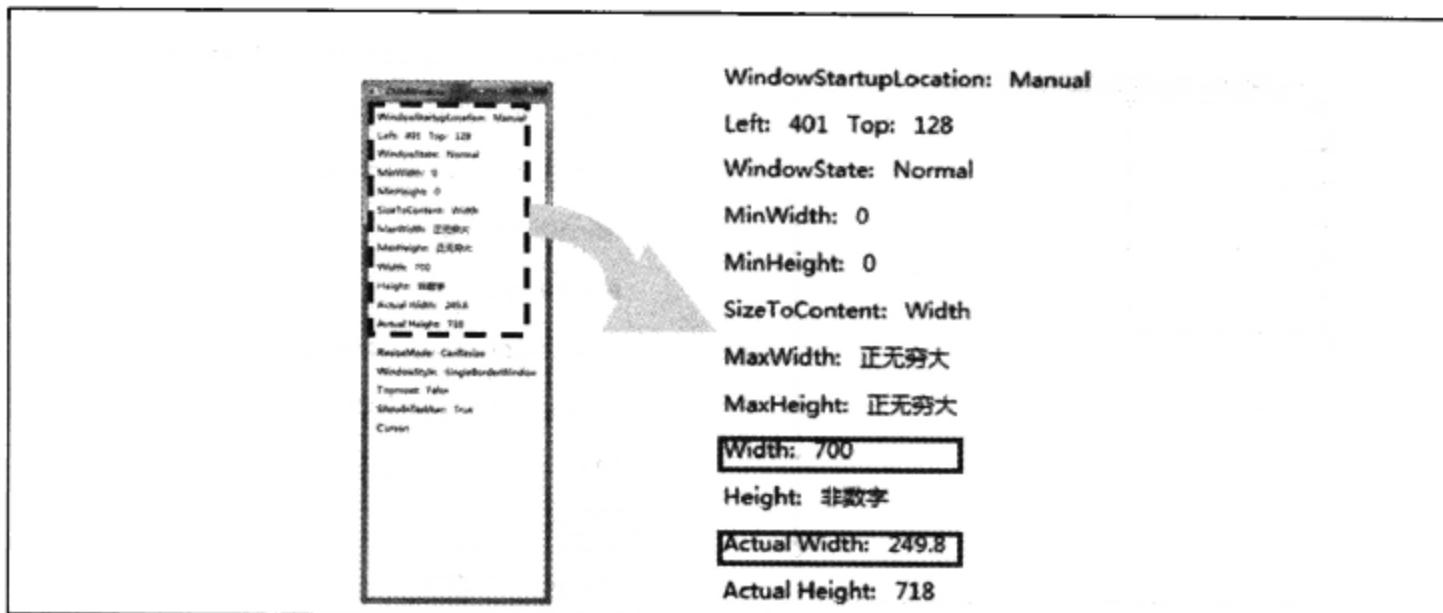


图 8-17 Width 和 Actual Width 属性已经不一致

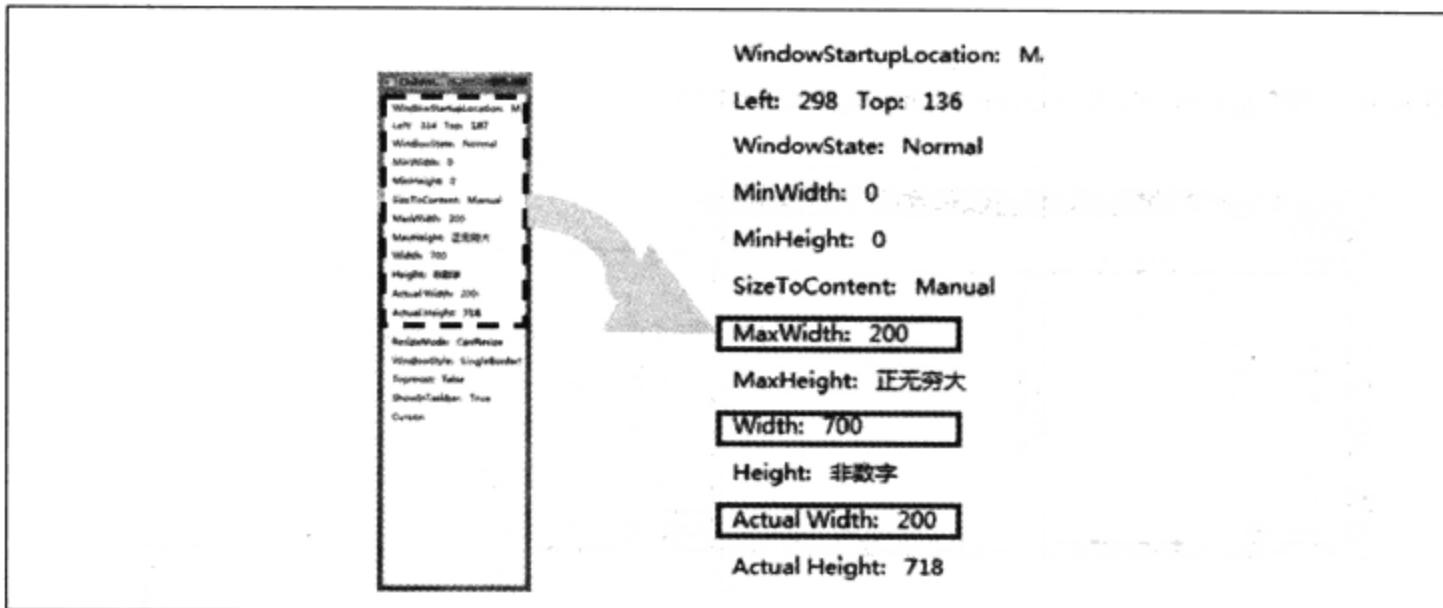


图 8-18 设置 MaxWidth 的结果

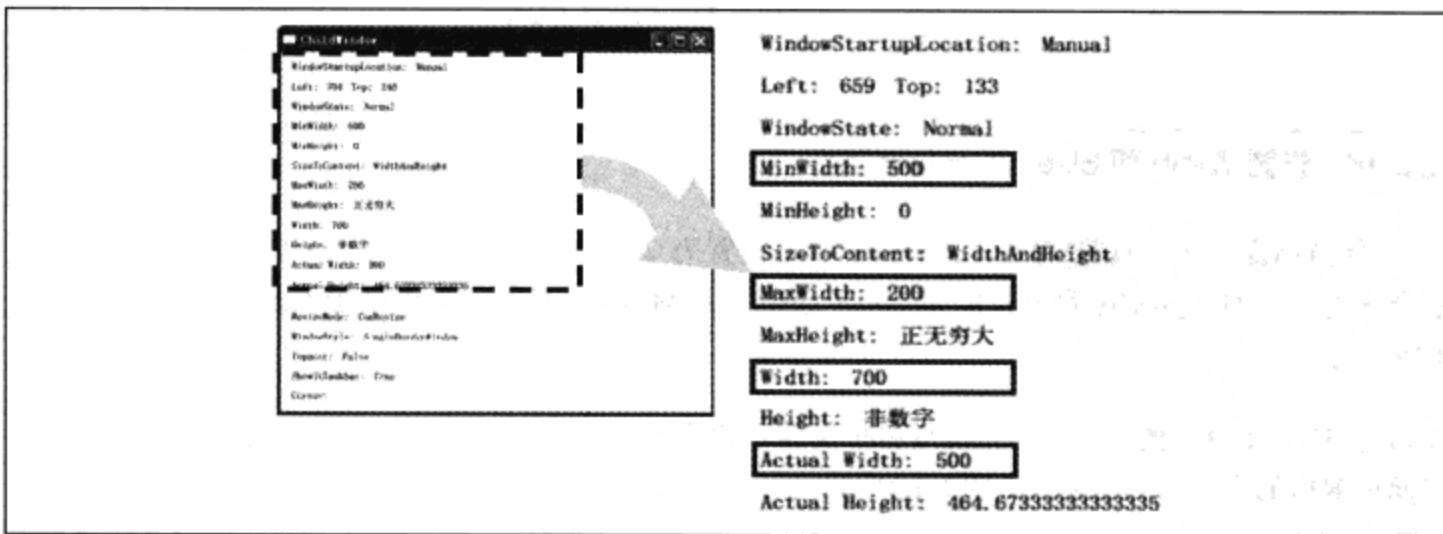


图 8-19 设置 MinWidth 的结果

2. 与外观和样式相关的属性

与窗口外观和样式的相关属性如下。

(1) `ResizeMode`

`ResizeMode` 有 4 种枚举变量，其中 `NoResize` 表示窗口不能调整其大小，同时没有最小化和最大化按钮；`CanMinimize` 表示窗口可以最小化，但是不能最大化，并且不能调整其大小；`CanResize` 表示窗口可以调整大小，包括最小化和最大化按钮；`CanResizeWithGrip` 类似 `CanResize`，不同是在窗口的右下角会增加一个可调整其大小的控制把手（Grip）。

`NoResize` 窗口如图 8-20 所示。

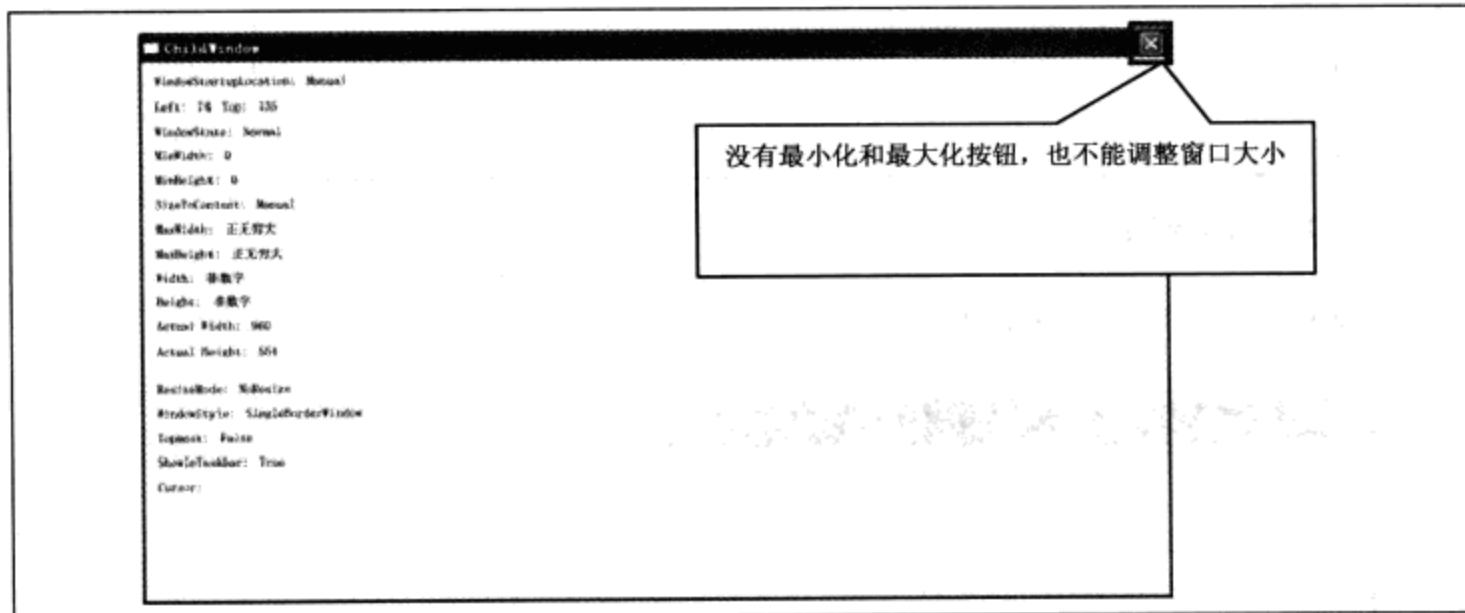


图 8-20 `NoResize` 窗口

`CanMinimize` 窗口如图 8-21 所示。

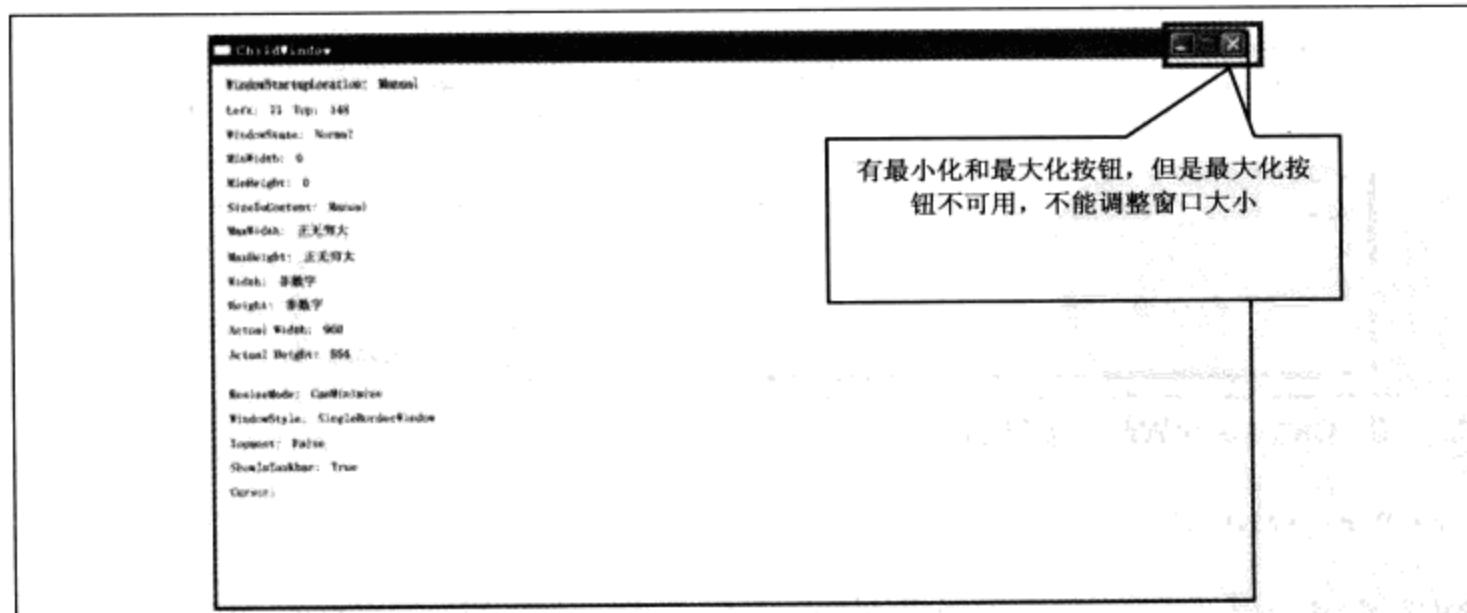


图 8-21 `CanMinimize` 窗口

CanResize 窗口如图 8-22 所示。

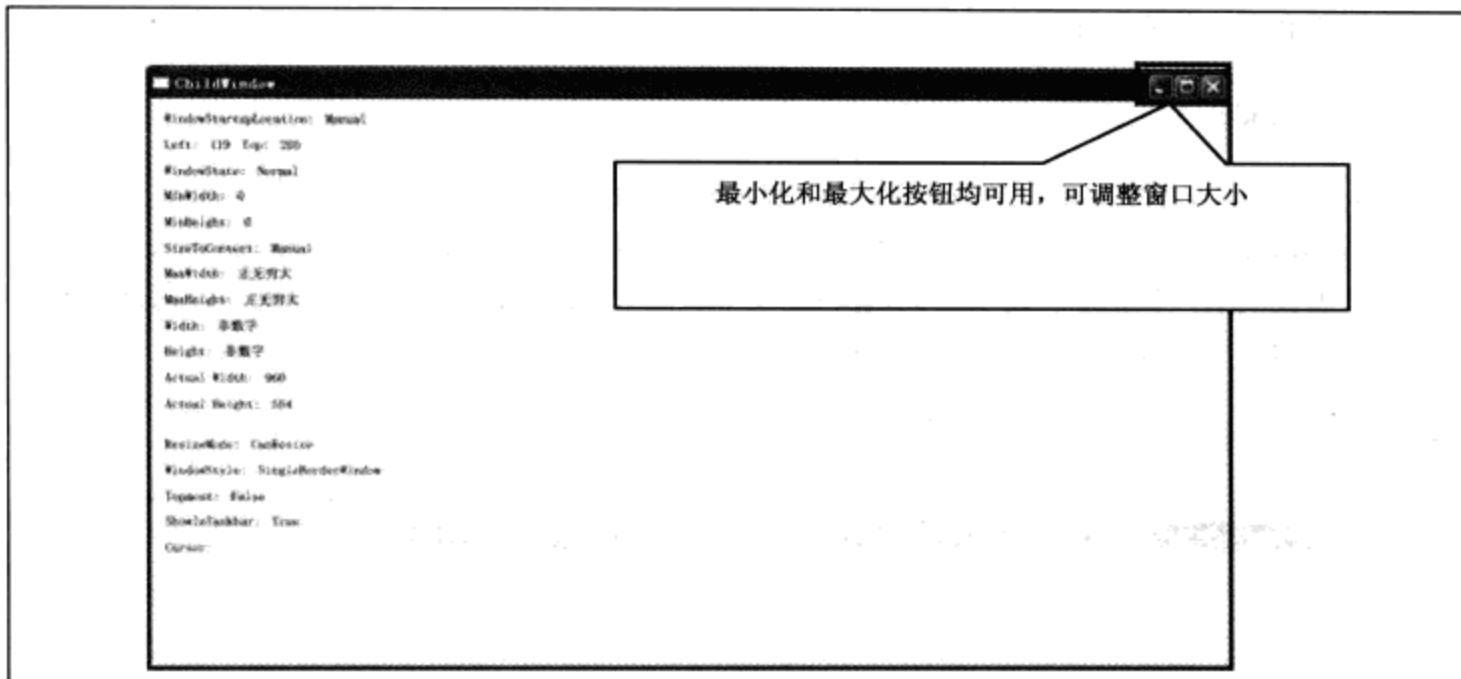


图 8-22 CanResize 窗口

CanResizeWithGrip 窗口如图 8-23 所示。

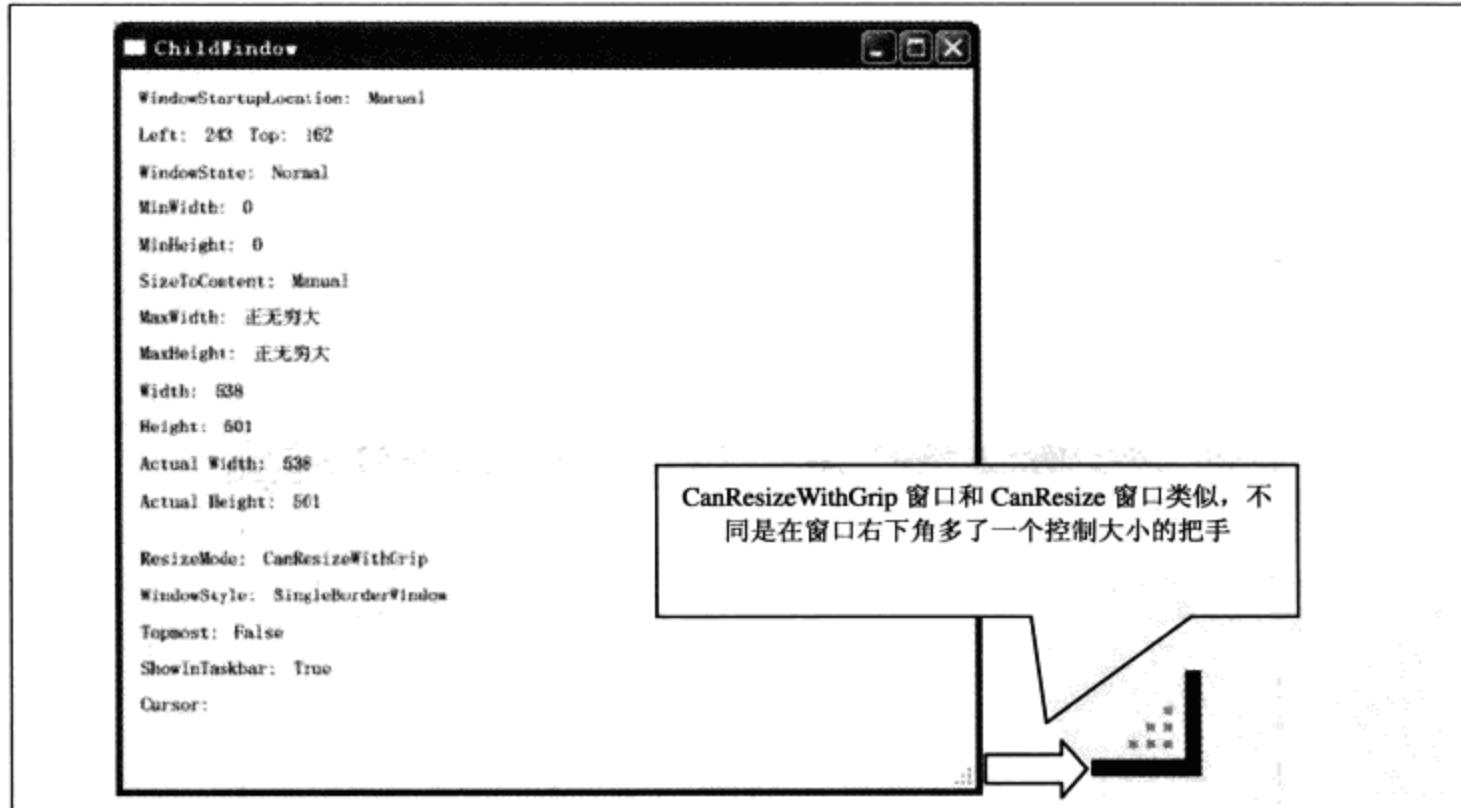


图 8-23 CanResizeWithGrip 窗口

(2) WindowStyle

WindowState 的 4 种枚举变量是 SingleBorderWindow、None、ThreeDBorderWindow 和 ToolWindow，分别如图 8-24、图 8-25、图 8-26 和图 8-27 所示。

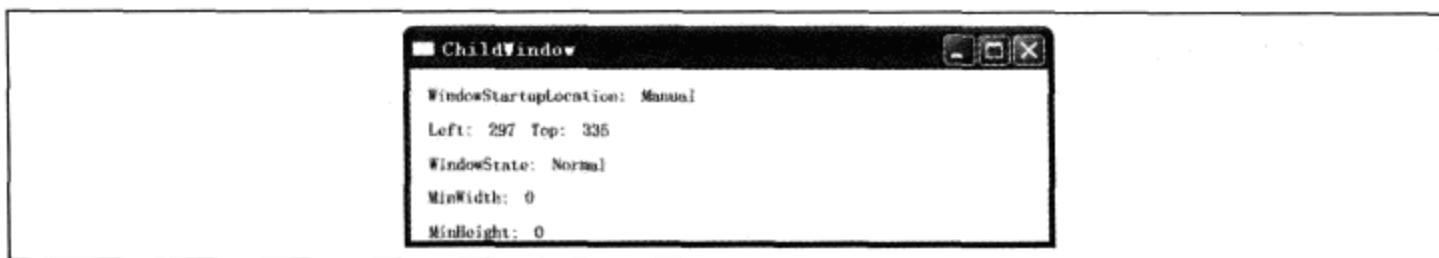


图 8-24 SingleBorderWindow 窗口样式

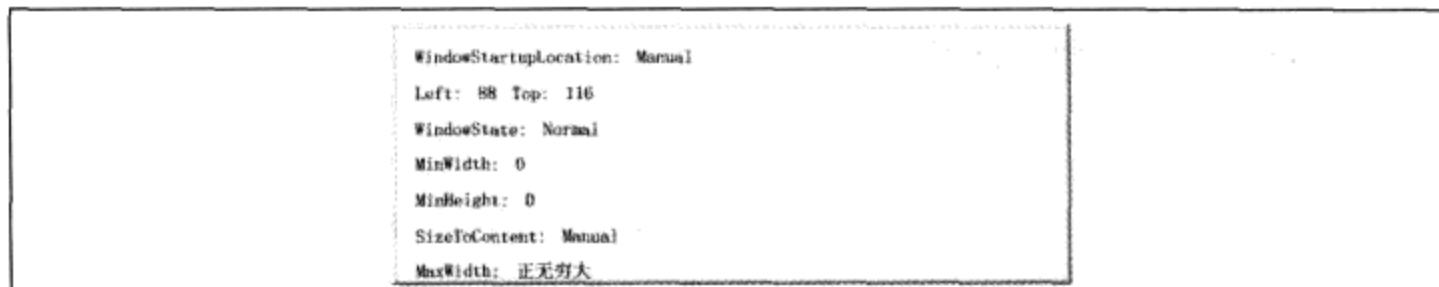


图 8-25 None 窗口样式

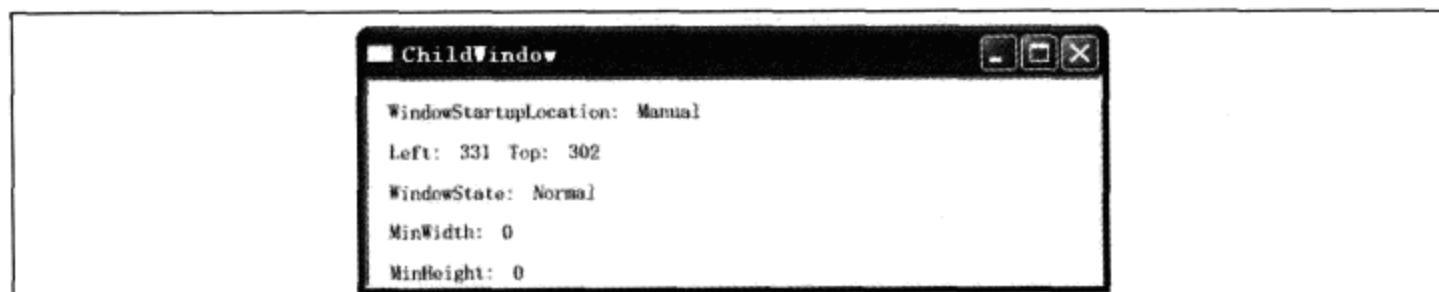


图 8-26 ThreeDBorderWindow 窗口样式

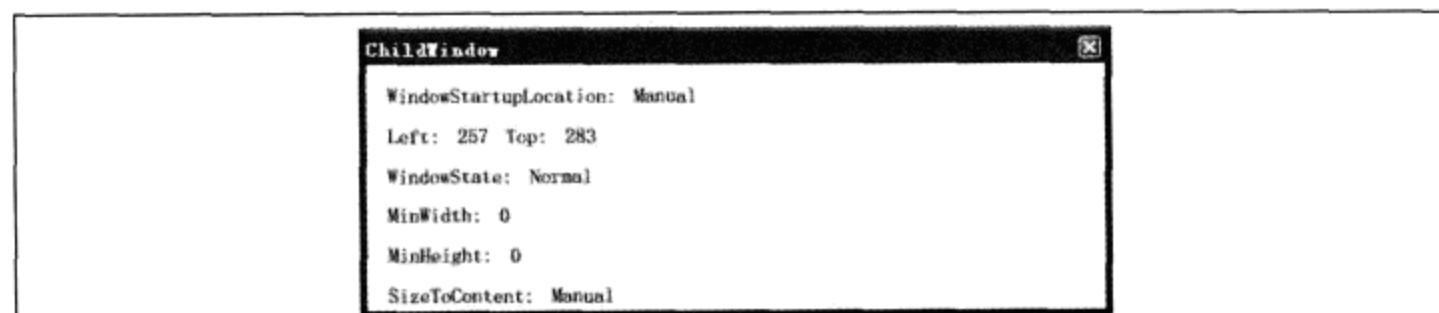


图 8-27 ToolWindow 样式

(3) ShowInTaskbar 和 Topmost

ShowInTaskbar 属性如果为 true，则窗口最小化会显示在任务栏中；否则不显示。Topmost 属性表示窗口是否为所有窗口的最上层，如果为 true，则在最上层；否则不在。

8.3.4 非规则窗口

有的时候，我们需要的窗口不一定都是中规中矩的矩形。在 WPF 中制作这种非规则窗口很简单，如做一个类似“标注”的对话框。可以拖动这个窗口并且关闭（详见 [mumu_nonRectangularwindow](#) 工

程)，如图 8-28 所示。



图 8-28 类似“标注”对话框的窗口

实际上非规则的窗口还是一个矩形窗口，只不过其背景设置为透明色。

(1) 在一个规则的窗口中绘制上面的图形，图形全部绘制在 Canvas 面板中。由 3 个部分组成。一是通过一个 Path 对象绘制的整个轮廓；二是“拖曳我！”的 Label；三是一个自定义控件外观的关闭按钮，如代码 8-15 所示。

```
<Window x:Class="mumu_nonRectangularwindow.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" SizeToContent="WidthAndHeight">
    <!--窗口的形状都绘制在 Canvas 面板中-->
    <Canvas Width="200" Height="200" >
        <!--窗口的轮廓-->
        <Path Stroke="DarkGray" StrokeThickness="2">
            <Path.Fill>
                <LinearGradientBrush StartPoint="0.2,0" EndPoint="0.8,1" >
                    <GradientStop Color="White" Offset="0" /></GradientStop>
                    <GradientStop Color="White" Offset="0.45" /></GradientStop>
                    <GradientStop Color="LightBlue" Offset="0.9" /></GradientStop>
                    <GradientStop Color="Gray" Offset="1" /></GradientStop>
                </LinearGradientBrush>
            </Path.Fill>
            <Path.Data>
                <PathGeometry>
                    <PathFigure StartPoint="40,20" IsClosed="True">
                        <LineSegment Point="160,20" /></LineSegment>
                        <ArcSegment Point="180,40" Size="20,20" SweepDirection="Clockwise" /></ArcSegment>
                        <LineSegment Point="180,80" /></LineSegment>
                        <ArcSegment Point="160,100" Size="20,20" SweepDirection="Clockwise" /></ArcSegment>
                    <LineSegment Point="90,100" /></LineSegment>
                    <LineSegment Point="90,150" /></LineSegment>
                    <LineSegment Point="60,100" /></LineSegment>
                    <LineSegment Point="40,100" /></LineSegment>
                    <ArcSegment Point="20,80" Size="20,20" SweepDirection="Clockwise" /></ArcSegment>
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>
    <!--“拖曳我！” 的标签-->
    <Label Content="Drag Me" />
</Canvas>
</Window>
```

```

<Label Width="200" Height="120" FontSize="15"
HorizontalContentAlignment="Center" VerticalContentAlignment="Center">Drag Me</Label>
<!--关闭按钮-->
<Button Canvas.Left="155" Canvas.Top="30">
    <Button.Template>
        <ControlTemplate>
            <Canvas>
                <Rectangle Width="15" Height="15" Stroke="Black"
RadiusX="3" RadiusY="3">
                    <Rectangle.Fill>
                        <SolidColorBrush x:Name="myAnimatedBrush" Color="Red" />
                    </Rectangle.Fill>
                </Rectangle>
                <Line X1="3" Y1="3" X2="12" Y2="12" Stroke="White"
StrokeThickness="2"></Line>
                <Line X1="12" Y1="3" X2="3" Y2="12" Stroke="White"
StrokeThickness="2"></Line>
            </Canvas>
        </ControlTemplate>
    </Button.Template>
</Button>
</Canvas>
</Window>

```

代码 8-15 MainWindow.xaml 文件

运行程序，只是在一个常规窗口的客户区中绘制了一个图形而已，如图 8-29 所示。



图 8-29 在常规窗口中绘制图形

(2) 需要设置窗口的部分属性，设置窗口的背景色为透明色，设置 AllowsTransparency 为 true，表示窗口的客户区支持透明色；另外为了删除窗口的标题栏和最小化按钮等，设置窗口的样式为 None，如代码 8-16 所示。

```

<Window .....
Background="Transparent" AllowsTransparency="True" WindowStyle="None">
.....

```

代码 8-16 进一步调整主窗口

再次运行程序，已经变成一个不规则的窗口，但是不能拖动。

(3) 为窗口添加一个单击事件来支持拖动，并为按钮添加一个 Click 事件处理函数支持关闭窗口，

如代码 8-17 所示。

```
MainWindow.xaml
<Window x:Class="mumu_nonRectangularwindow.MainWindow".....
MouseLeftButtonDown="Window_MouseLeftButtonDown".....>
.....
    <!--关闭按钮-->
<ButtonCanvas.Left="155"Canvas.Top="30"Click="Button_Click">
.....
```

代码 8-17 添加鼠标左键和按钮的 Click 事件

两个事件处理函数的实现非常简单，窗口本身有一个 DragMove 方法支持拖动，而窗口关闭则使用 Close 方法，如代码 8-18 所示。

```
MainWindow.xaml.cs
private void Window_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    this.DragMove();
}
private void Button_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

代码 8-18 实现鼠标左键和按钮的 Click 事件处理函数

WPF 还有一个非常好的特性，即背景色为透明处不接受鼠标的检测。即只能在这个标注轮廓中接收鼠标消息，而其他透明处则不可，这样用户使用时感觉不到这是一个背景为透明色的矩形窗口。

8.4 接下来做什么

窗口是每种框架都不可避免的话题，一切都从窗口开始。在 Windows 操作系统上，万物皆窗口。如果希望进一步了解窗口，可以参考作者的博文《永远的窗口》。如果希望了解 Win32 的窗口和 WPF 的窗口之间的联系，可以参考周永恒的一篇博文《一站式 WPF——Window（一）》。

接下来，我们谈论的仍是窗口，不过这种窗口看起来更像一种网络程序，它就是基于导航的窗口。

参考文献

[1] MSDN Library for Visual Studio 2008 SP1 Application Management Overview。

[2] MSDN Library for Visual Studio 2008 SP1 WPF Windows Overview。

页面和导航——天罡北斗阵演绎

全真七子马钰位当天枢，谭处端位当天璇，刘处玄位当天玑，丘处机位当天权，四人组成斗魁；王处一位当玉衡，郝大通位当开阳，孙不二位当摇光，三人组成斗柄。北斗七星中以天权光度最暗，却是居魁柄相接之处，最是冲要，因此由七子中武功最强的丘处机承当。斗柄中以玉衡为主，由武功次强的王处一承当……

——《射雕英雄传》，“第二十五回 荒村野店”^[1]

上面这一部分讲的是全真七子第一次用天罡北斗阵迎敌，对手是铁尸梅超风。天罡北斗阵是全真教中最上乘的玄门功夫，当年重阳仙师为此阵花过无数心血。这阵法依照北斗七星的位置布置，全真七子除迎敌时出一掌外，另一掌总是搭在身旁之人肩上。环环相扣，七人之力合而为一，实在是威不可当。别说一个小小的梅超风，就是东邪西毒南帝北丐四大宗师任何一个人都未必挡得住天罡北斗阵的威力。

天罡北斗阵中这七个道士一个连着一个的样子和 WPF 导航应用程序有几分相似，本章通过剖析武学宗师王重阳演绎天罡北斗阵来一窥 WPF 导航应用程序的奥秘。

本章内容如下。

- (1) 导航应用程序演绎。
- (2) 导航的内容——页面。
- (3) 导航的连接。
- (4) 历史管理——Journal。
- (5) 导航和 Page 的生命周期。
- (6) 页面状态保留和数据传递。
- (7) XAML 浏览器应用程序。
- (8) 接下来做什么。

9.1 导航应用程序演绎

9.1.1 第3类应用程序

在传统的江湖里，存在两种格斗模式，一种是“单挑”；另一种则是“群殴”。天罡北斗阵开创了第3种格斗模式，它介于“单挑”和“群殴”之间。小则可以联手搏击，化而为大，可用于两军对阵。

在传统的程序中，同样是两类应用程序模式：一是桌面程序，由窗口和对话框组成；二是 Web 程序，由多个页面组成。页面之间用超链接连接，并在 IE 浏览器中运行。WPF 的导航应用程序属于第 3 类应用程序模式，它模糊了前两类应用程序的界限。因为其外观可能是窗口，也可能是浏览器。

WPF 导航应用程序从行为上来看，更像一个 Web 程序。通过超链接来改变窗口或者窗口的部分内容，但其本质仍然是一个桌面程序。

贾宝玉说女儿是水做的，男儿是泥做的。套用这么一句话就是“Web 程序是‘水’做的，桌面程序是‘泥’做的，WPF 导航应用程序则是‘水泥’做的，它是第三类应用程序。”^_^

9.1.2 两种形式

天罡北斗阵在实战中有两种形式，一种是七人之阵，用于联手搏击；另一种则可以每七人一组。每七个北斗阵又组成一个大北斗阵，如此反复循环，可用于两军对垒^[2]。

WPF 导航应用程序也表现为两种形式：一是将导航的内容寄宿于窗口中，实际上这样一类导航应用程序。平常所使用的词典、MSDN 或者文件资源管理器均属此类，如图 9-1 所示。

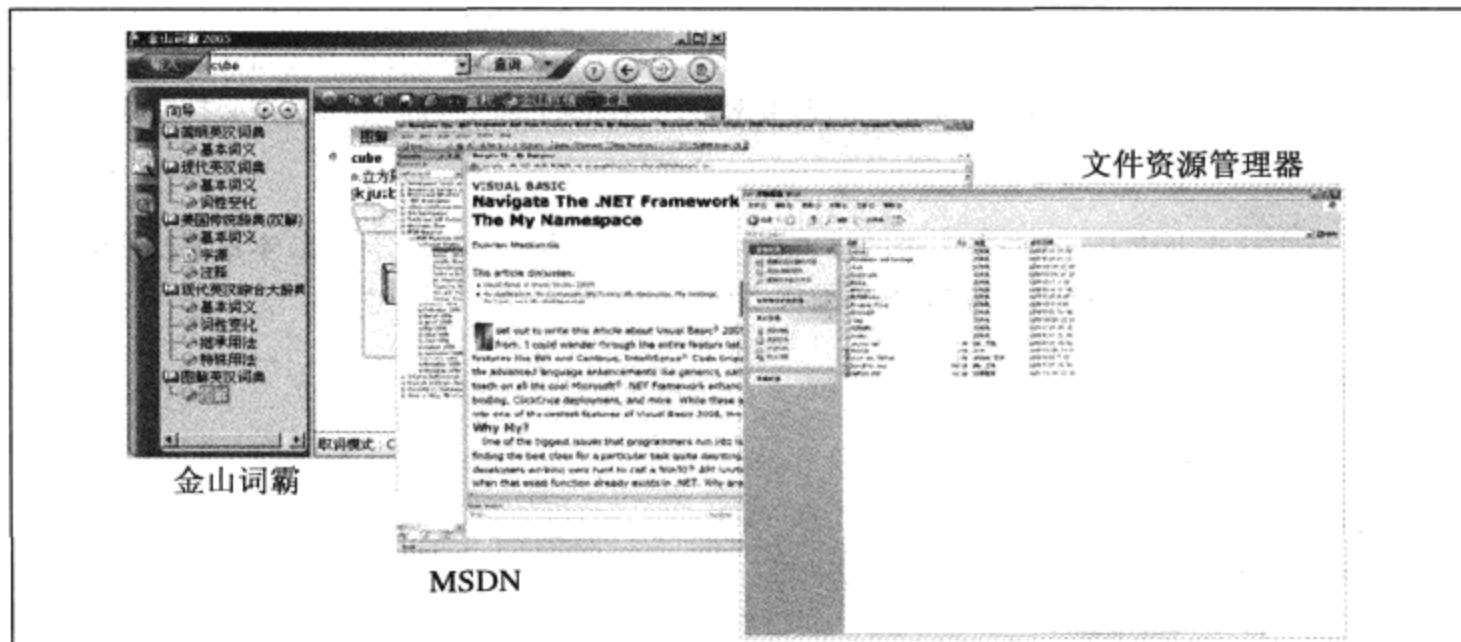


图 9-1 导航内容寄宿于窗口中的导航应用程序

二是 XAML 浏览器应用程序（XAML Browser Applications, XBAP），扩展名为“.xbap”，可以直接在浏览器中运行。

9.1.3 4 个核心

天罡北斗阵中最重要的是天枢、天璇等七星点位及其之间的联系，全真七子占据北斗阵中天枢、天璇等七个位置。他们之间通过内力相连，敌人击首则尾应，击尾则首应，击腰则首尾皆应。

WPF 导航应用程序中重要的导航内容及其之间的联系，其 4 个核心要素如下。

- (1) 页面 (Page)：WPF 将导航的内容封装为多个页面。
- (2) 超链接 (Hyperlink)：页面之间通过超链接连接。
- (3) NavigationServices：页面之间连接的另一种方式，通过编程实现，实际上超链接连接时底层也调用 NavigationServices 的方法。
- (4) Journal：每次连接通过 Journal 记录作为历史记录。

总结起来即页面为内容、超链接相连、NavigationServices 编程连接和 Journal 记录。

9.2 页面

9.2.1 Page

页面在 WPF 中是一个 Page 类型，它继承自 FrameworkElement，比 Window 类更加精简。该类未提供 Window 类中的 Show 和 Hide 方法来显示和隐藏一个窗口，而是通过导航来实现页面的切换。

此外，Page 一般不设置自身尺寸，因为页面尺寸由包含它的宿主窗口决定。如果设置了页面的 Width 和 Height 属性，则页面具有相应大小。如果宿主窗口比页面小，那么页面中的一些内容就会被裁减；如果宿主窗口比页面大，那么页面就会在宿主窗口中居中显示。页面可以通过设置 Page 的 WindowWidth、WindowHeight 和 WindowTitle 属性来改变宿主窗口的宽度、高度和标题属性，这个宿主窗口可以是普通的窗口或浏览器，如代码 9-1 所示。

```
App.xaml
<Application x:Class="mumu_simplepage.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Startup="Application_Startup">
    <Application.Resources>
        </Application.Resources>
</Application>

App.xaml.cs
public partial class App : Application
{
    private void Application_Startup(object sender, StartupEventArgs e)
    {
        NavigationWindow win = new NavigationWindow();
        // 1 没有配置的情况
        win.Content = new Page1();
        // 2 宿主窗口大于 Page 尺寸
        // win.Content = new Page1(300,300,500,500);
        // 2 宿主窗口小于 Page 尺寸
        // win.Content = new Page1(500,500,450, 450);
        win.Show();
    }
}
```

```

Page1.xaml
<Page x:Class="mumu_simplepage.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1" >
    <Border Margin="5" BorderThickness="3" BorderBrush="Red"
Background="AntiqueWhite">
    <TextBlock FontSize="22" x:Name="text" VerticalAlignment="Center"
HorizontalAlignment="Center"/>
  </Border>
</Page>

Page1.xaml.cs
public partial class Page1 : Page
{
    public Page1(double width, double height, double hostwinWidth, double
hostwinHeight) : this()
    {
        this.Width = width;
        this.Height = height;
        this.WindowHeight = hostwinHeight;
        this.WindowWidth = hostwinWidth;

        this.WindowTitle = "对窗口尺寸进行了配置";
        this.text.Text = "Width = " + width + "\n\n" + "Height = " + height
+ "\n\n" + "WindowWidth = " + hostwinWidth + "\n\nWindowHeight = " + hostwinHeight;
    }
    public Page1()
    {
        InitializeComponent();
        this.WindowTitle = "没有对窗口尺寸进行了配置";
    }
}

```

代码 9-1 mumu_simplepage 工程的部分代码文件

图 9-2 分别对应该程序的 3 种情况，即没有配置宿主窗口大小、宿主窗口大于页面尺寸和宿主窗口小于页面尺寸。

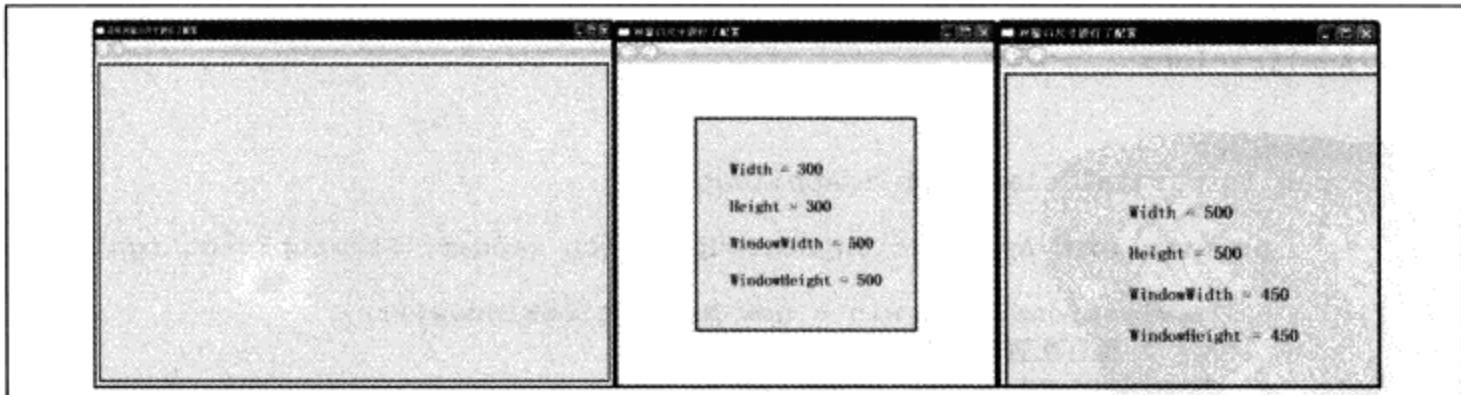


图 9-2 程序的 3 种情况

9.2.2 Page 的宿主窗口

Page 的宿主窗口包括浏览器、导航窗口（NavigationWindow）和 Frame。

后两种是 WPF 提供的 Page 宿主窗口。这种宿主窗口提供了从一个 Page 导航到另一个 Page 的功能，并且能记录这些导航的历史，提供一系列的导航事件。NavigationWindow 派生自 Window，在外观上与普通窗口最大的区别是多了一个导航工具栏，不过它可以通过设置 ShowsNavigationUI 属性来控制该工具栏是否可视。

NavigationWindow 是一个顶层窗口，不允许嵌入到其他控件中。而 Frame 则为轻量级，可以嵌入到其他控件，如 NavigationWindow 或者 Page，甚至其他 Frame 中。Frame 默认没有导航工具栏，可以设置其 NavigationUIVisibility 属性为 Visible 使其导航工具栏可见。代码 9-2 的功能是在一个 NavigationWindow 中放置一个页面，然后在该页面中嵌套一个 Frame。

```
App.xaml
<Application x:Class="mumu_navigationwindowandframe.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Page1.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>

Page1.xaml
<Page x:Class="mumu_navigationwindowandframe.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page1">
    <Border BorderBrush="Red" BorderThickness="2" Margin="2">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <TextBlock Grid.Row="0" Text="该页面的宿主窗口是一个 NavigationWindow"
HorizontalAlignment="Center" VerticalAlignment="Center" />
                <Frame Source="Page2.xaml" NavigationUIVisibility="Visible" />
            </Grid>
        </Border>
    </Page>
Page2.xaml
<Page x:Class="mumu_navigationwindowandframe.Page2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page2">
    <Grid>
        <Border Grid.Row="2" BorderBrush="Blue" BorderThickness="2" Margin="2">
            <TextBlock Grid.Row="0" Text="该页面的宿主窗口是一个 Frame"
HorizontalAlignment="Center" VerticalAlignment="Center" />
        </Border>
    </Grid>
</Page>
```

代码 9-2 mumu_navigationwindowandframe 工程中的部分文件

该程序并没有新建一个 NavigationWindow，只是在 App.xaml 文件中将 Application 的 StartupUri 指向 Page1.xaml 文件。WPF 发现所指内容不是一个窗口，而是一个 Page 对象时会自动为该对象创建一个 NavigationWindow。为了清晰地辨别 Page 和 Frame，我们为页面添加了一个边框，并将 Frame 的导航工具栏设置为可见。该程序的运行结果如图 9-3 所示。

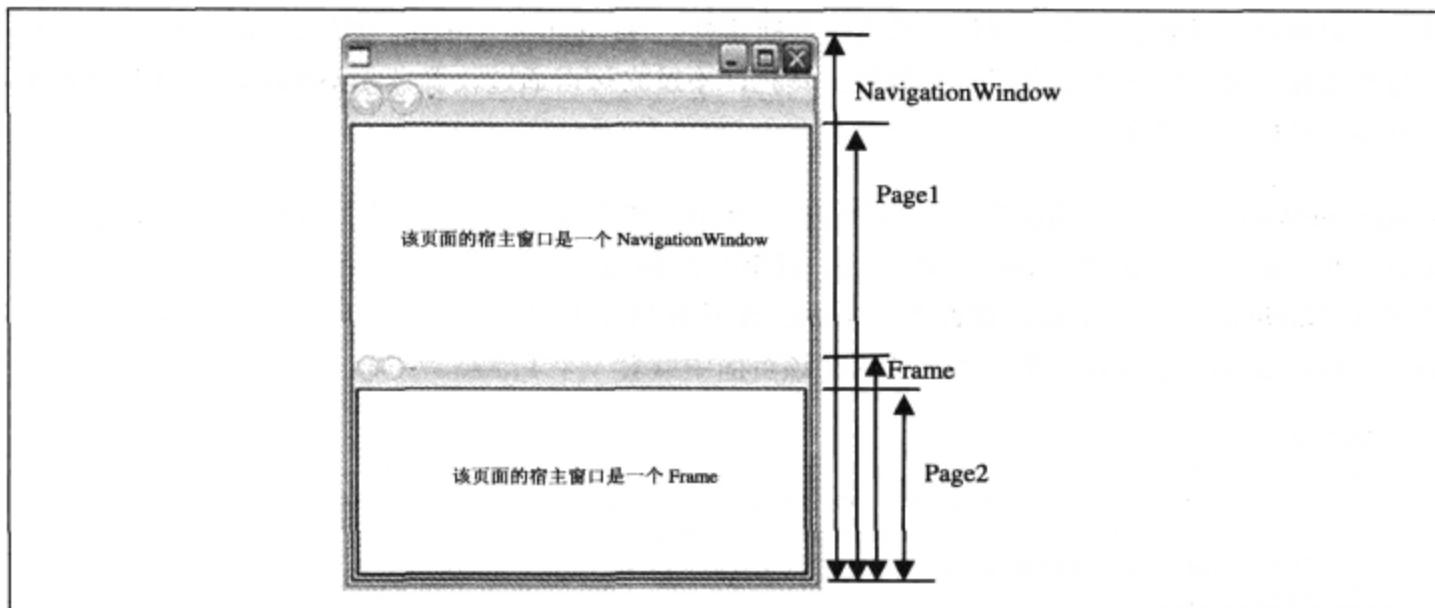


图 9-3 运行结果

9.3 导航连接

9.3.1 超链接

超链接的典型式样是一种带有蓝色下画线的文本，单击该文本就会被导航到指定位置，如图 9-4 所示。

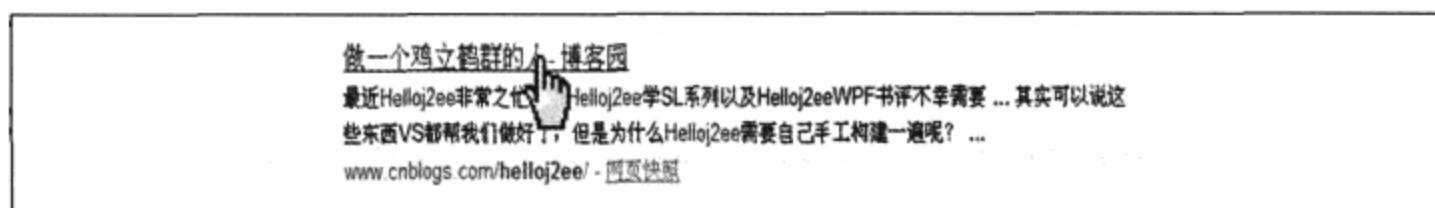


图 9-4 超链接（图片来源为 <http://news.google.cn/nwshp?hl=zh-CN&tab=wn>）

WPF 中超链接类型为 Hyperlink，它派生自 Span，属于流文档模型中的一个要素。利用 Hyperlink 导航的简单方法是设置其 NavigateUri 属性，如代码 9-3 所示。

```
<Hyperlink NavigateUri="Page2.xaml">  
    开始阅读路由事件  
</Hyperlink>
```

代码 9-3 设置 NavigateUri 属性

Hyperlink 除了能够在页面之间导航，还可以在同一个页面之间进行段落导航（fragment navigation）。如代码 9-4 所示，这个 Page 页面是第 6 章中的节选。前面的列表可以通过 Hyperlink 链接后面的内

容，这时 NavigateUri 的设置方法是“页面名#元素名”。

```
<Page x:Class="mumu_Hyperlink.Page2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page2">
    <Grid>
        <FlowDocumentReader>
            <FlowDocument>
                .....
                <Section LineHeight="25" FontSize="15">
                    .....
                    <List >
                        <ListItem>
                            <Paragraph>
                                <Hyperlink NavigateUri="Page2.xaml#first">
                                    从玉蜂说起，回顾.NET 事件模型
                                </Hyperlink>
                            </Paragraph>
                        </ListItem>
                        <ListItem>
                            <Paragraph>
                                <Hyperlink NavigateUri="Page2.xaml#second">
                                    什么是路由事件？
                                </Hyperlink>
                            </Paragraph>
                        </ListItem>
                    </List>
                </Section>
                <Paragraph x:Name="first" FontSize="20" Background="AliceBlue">
                    1. 从玉蜂说起，回顾.NET 事件模型
                </Paragraph>
                .....
                <Paragraph x:Name="second" FontSize="20" Background="AliceBlue">
                    2. 什么是路由事件？
                </Paragraph>
                <Paragraph>
                    什么是路由事件呢？木木很快查看了一下 MSDN，MSDN 从功能和实现两种视角给出了路由事件的定义。
                </Paragraph>
            </FlowDocument>
        </FlowDocumentReader>
    </Grid>
</Page>
```

代码 9-4 页面中的段落导航

Hyperlink 也有 Click 事件，也可以在其处理函数中通过 NavigationService 的 Navigate 方法导航。如代码 9-5 所示。

```
private void Hyperlink_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("pack://application:,,,/Page2.xaml"));
}
```

代码 9-5 使用 NavigationService 导航

本小节的所有代码都来自 mumu_Hyperlink 示例，该程序运行后的第 1 个页面中只有一个超链接“开始阅读路由事件”。单击后会切换到第 2 个页面，这是一个页面之间的导航。单击第 2 个页面之间

的链接，会切换到相应的段落，这是一个段落导航，如图 9-5 所示。

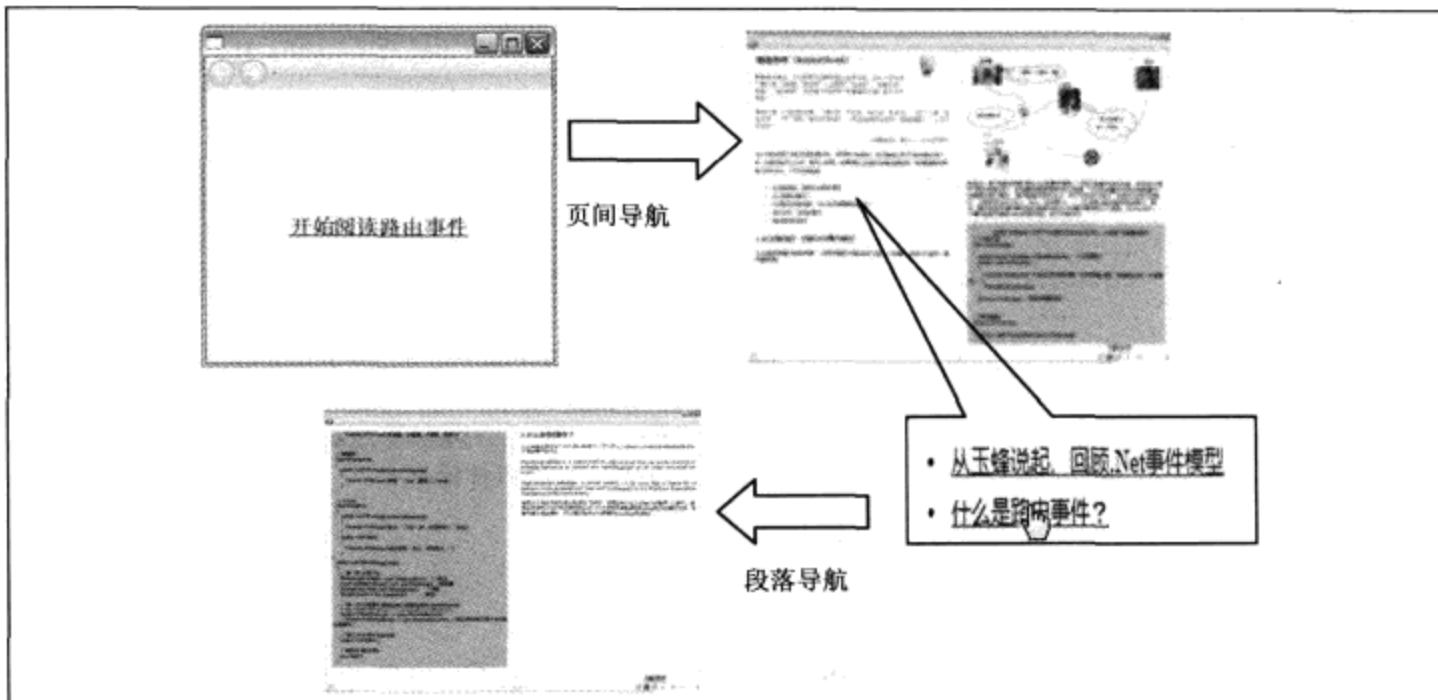


图 9-5 页间导航和段落导航

注意超链接有时也会无效，如页面寄宿的窗口不是 `NavigationWindow`，而是一个普通的窗口，则超链接是无法导航。如果将 9.2.1 节中 `mumu_simplepage` 的 `App.xaml.cs` 文件的 `Application_Startup` 事件处理函数修改为代码 9-6，那么在 `Page1` 页面中添加任何 `Hyperlink` 都无法实现导航的：

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    // NavigationWindow win = new NavigationWindow();
    Window win = new Window();
    win.Content = new Page1();
    win.Show();
}
```

代码 9-6 `NavigationWindow` 类替换成 `Window` 类

9.3.2 通过编程导航

虽然超链接方便实用，但是遇到下面几种情况的时候，超链接就会显得束手无策了。

- (1) 在导航到页面之前传递一个数值给该页面对象。
- (2) 在导航到页面之前设置该页面的某些属性。
- (3) 事先不知道需要导航的目标页面，而在程序运行时才决定导航到哪个页面^[3]。

以上 3 种情况需要编程来实现导航，为此主要依靠 `NavigationService` 类。事实上 `Hyperlink` 被单击时，WPF 也调用 `NavigationService` 的 `Navigate` 方法来导航页面^[3]。

`NavigationService` 的 `Navigate` 方法可以接受一个对象或一个 URI。虽然大多数情况看到的是从一个

页面导航到另一个页面，但事实上通过 `NavigationService` 不仅可以导航到页面，还可以导航到 HTML 文件，甚至是普通的.NET 对象，如代码 9-7 所示。

```
Page1.xaml
<Page x:Class="mumu_navigateanyone.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1">
    <Page.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Margin" Value="5"/>
        </Style>
    </Page.Resources>
    <StackPanel>
        <TextBlock>
            导航到
            <Hyperlink x:Name="link1" Click="Hyperlink_Click">
                Page2.xaml
            </Hyperlink>
        </TextBlock>
        <TextBlock>
            导航到
            <Hyperlink x:Name="link2" Click="Hyperlink_Click">
                Page2.xaml
            </Hyperlink>
            (且调用带有参数的构造函数)
        </TextBlock>
        <TextBlock>
            导航到一个松散 XAML 文件
            <Hyperlink x:Name="link3" Click="Hyperlink_Click">
                LooseXaml.xaml
            </Hyperlink>
        </TextBlock>
        <TextBlock>
            导航到
            <Hyperlink x:Name="link4" Click="Hyperlink_Click">
                普通.NET 对象
            </Hyperlink>
        </TextBlock>
        <TextBlock>
            导航到网页
            <Hyperlink x:Name="link5" Click="Hyperlink_Click">
                http://www.cnblogs.com/helloj2ee/
            </Hyperlink>
        </TextBlock>
    </StackPanel>
</Page>

Page1.xaml.cs
public partial class Page1 : Page
{
    public Page1()
    {
        InitializeComponent();
    }
    private void Hyperlink_Click(object sender, RoutedEventArgs e)
    {
        Hyperlink link = sender as Hyperlink;
```

```

        if (link == link1)
            NavigationService.Navigate(new
Uri("pack://application:,,,/Page2.xaml"));
        else if (link == link2)
            NavigationService.Navigate(new Page2("使用 NavigationService 导航
到 Page2.xaml"));
        else if (link == link3)
            NavigationService.Navigate(new
Uri("pack://application:,,,/LooseXaml.xaml"));
        else if (link == link4)
            NavigationService.Navigate(new Person("木木", "男", 20));
        else if (link == link5)
            NavigationService.Navigate(new
Uri("http://www.cnblogs.com/helloj2ee/"));
    }
}

```

代码 9-7 导航到任意对象（详见 `mumu_navigateanyone` 工程）

9.3.3 其他方式导航

1. 导航工具栏

导航工具栏会记录每次页面的跳转，然后用户通过单击其的前进和后退按钮即可实现导航。导航工具栏上由 3 个部分组成，即前进和后退按钮，以及最近列表，如图 9-6 所示。

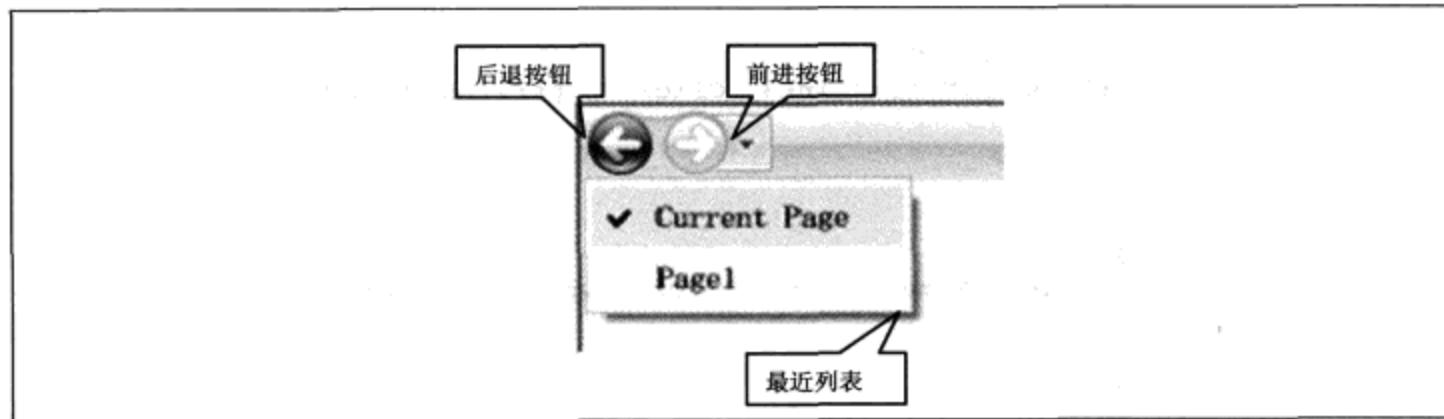


图 9-6 导航工具栏

最近列表中的列表项默认显示页面的 URI 希望修改的名称，如将图中的“Page1”修改为“木木的首页”，则 WPF 中提供如下 3 种方法。

- (1) 设置一个附加属性“`JournalEntry.Name`”。
- (2) 设置 `Page` 的 `Title` 属性。
- (3) 设置 `Page` 的 `WindowTitle` 属性。

3 种方法的优先级为 `JournalEntry.Name > Page.Title > Page.WindowTitle`，即如果设置了 `JournalEntry.Name`，则忽略其他属性。代码 9-8 在列表项中显示“松散 XAML 文件”。

```

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

```

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="LooseXaml" JournalEntry.Name="松散 XAML 文件" WindowTitle ="窗口">
.....
</Page>
```

代码 9-8 列表项设置的不同优先级

2. 导航命令

除了导航工具栏以外，也可以通过添加命令的方式使前进和后退功能可用，代码 9-9 为两个按钮添加了导航中的前进和后退功能。

```
<Button Height="50" Width="60" Content="后退" Command=
"NavigationCommands.BrowseBack "/>
<Button Height="50" Width="60" Content="前进" Command=
"NavigationCommands.BrowseForward "/>
```

代码 9-9 为按钮添加导航中的前进和后退功能

9.4 历史管理

Journal 记录每一次的导航，这样在导航工具栏中才能实现前进和后退功能。

我们不妨再戏说一下“天罡北斗阵”，其中最重要的是需要 7 个人内力可以任意传递，如从天枢到天璇再到天玑。但是又要依照该顺序收回，即从天玑到天璇再到天枢。如果只是简单地这样顺序地传递并收回，那么重阳老先生的 7 个弟子还是可以记住的。但是如果内力任意跳转，中间内力传递不正确再需要重来几次的话，恐怕只有像黄蓉那样记忆惊人的人才能记住。全真七子资质平常，显得很为难。于是重阳老先生在旁边看他们练习时，需要不停地问：“丘处机错了，马钰你应该传递给孙不二……”有时老先生一不留神，自己也被搞糊涂了。

但是老先生何许人也？中神通，当世武学泰斗，武学界的“巨鳄”，这样的小问题岂能难得住他老人家。老人家准备了两个盒子，分别写有“后退”和“前进”，此外还有写有“天枢”及“天权”等七星位置的小纸片。

现在来看老先生如何指挥全真七子训练，如图 9-7 所示天罡北斗阵，刚开始七子向老师行礼，然后摆开架势练习。

- (1) 老先生的两个盒子都是空的，即没有任何历史记录，如图 9-8 所示。
- (2) 天枢位将内力传递给天璇位，老先生将一个写有“天枢”的小纸片放入后退盒中，如图 9-9 所示。
- (3) 天璇位将内力传递给天权位，老先生又将一个写有“天权”的小纸条放入后退盒中，如图 9-10 所示。

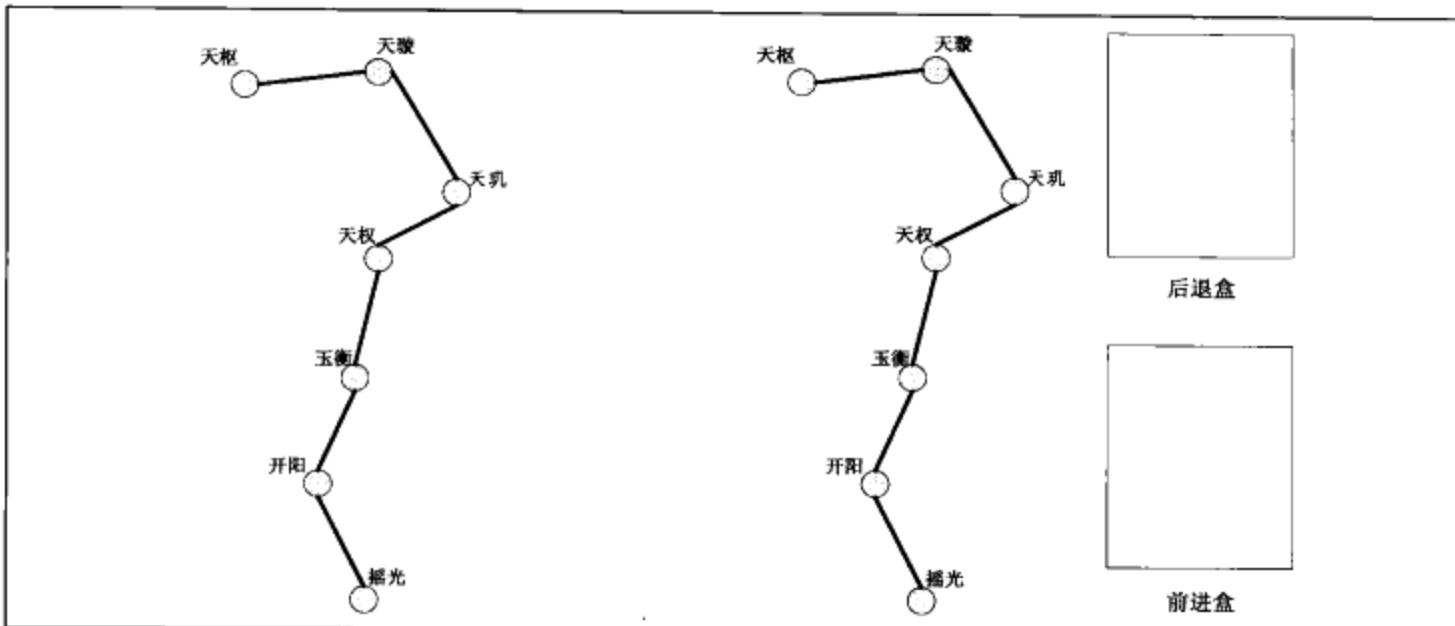


图 9-7 天罡北斗阵

图 9-8 前进盒和后退盒的状态一

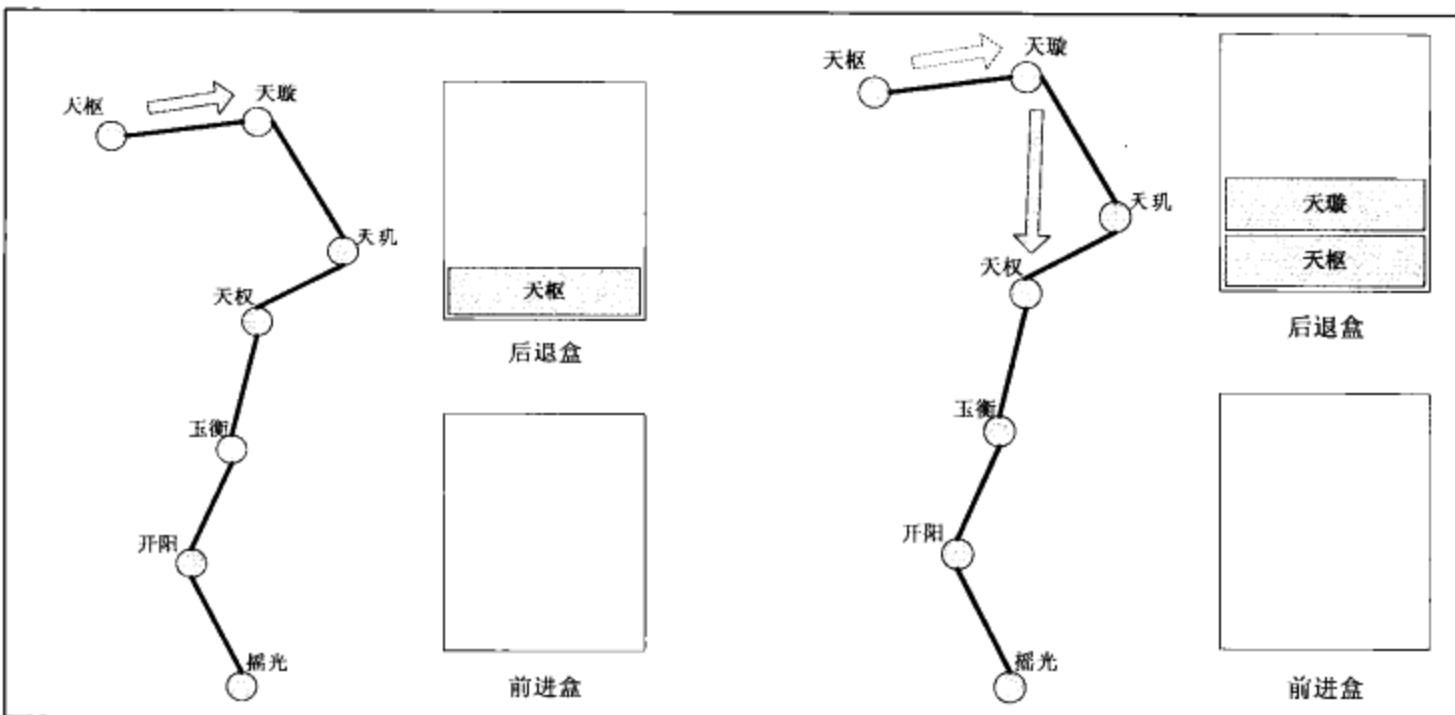


图 9-9 前进盒和后退盒的状态二

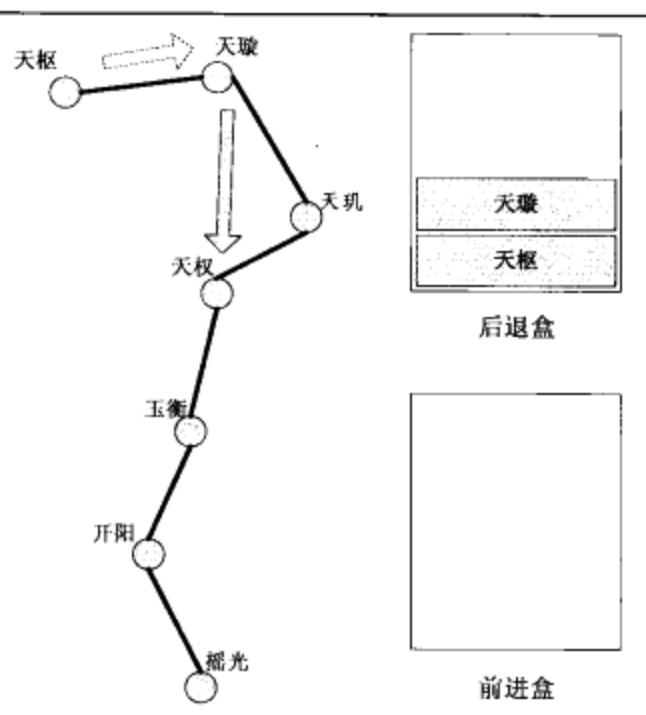


图 9-10 前进盒和后退盒的状态三

(4) 这时老先生发现丘处机(处于天权位)练功分心了, 接受内力的姿势已经不对, 于是说道: “重来, 丘处机你将内力回传。”但是丘处机因为分神, 所以早就不知道内力从何而来。老先生看了一下回退盒上面的第一个纸片, 纸片上写有“天璇”, 于是说到: “你将内力回传给谭处端(处于天璇位)。”说完将“天璇”的纸片从后退盒里拿出来, 将“天权”的纸片放入前进盒, 如图 9-11 所示。

(5) 处于天璇位的谭处端将内力重新传给处于天权的丘处机, 这次丘处机总算没有大意, 非常完美。于是重阳老先生不需要保留前进盒中的天权纸片, 将其拿了出来。又将写有“天璇”的纸片放入了后退盒中, 如图 9-12 所示。

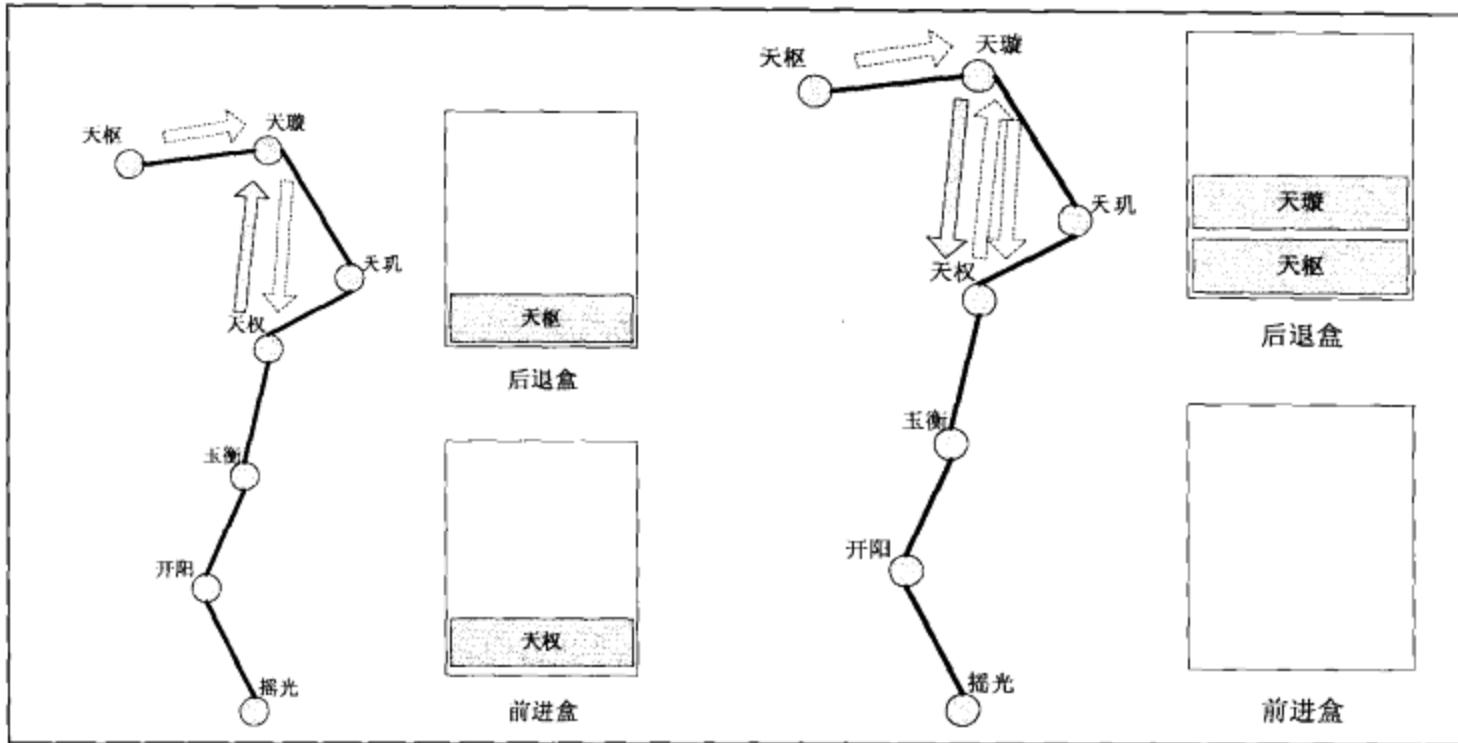


图 9-11 前进盒和后退盒的状态四

图 9-12 前进盒和后退盒的状态五

如此下来，全真七子才最终训练成震古铄金的“天罡北斗阵”。

重阳先生记录内力的方法和 Journal 记录导航历史的方法如出一辙，重阳先生的后退盒和前进盒分别是 Journal 的两个栈，即 Back Stack 和 Forward Stack。而每个小纸片则是栈中的每一项，在 Journal 中为 Journal Entry。全真七子即每个页面，而其之间的内力传递可以看做是一次导航。由于丘处机的分心使得需要重来一次，我们可以看做是一次回退，而谭处端将内力重新传递给丘处机则是一次前进操作。

9.5 导航和 Page 的生命周期

9.5.1 这一“点击”的背后

高手过招的时候，动作往往快如闪电，看似只是一招，对手应声倒地，实际上这里面已经隐含了七拳八脚。导航也是一样，看似只是在页面上点击了一下链接，实际上这里面隐藏了若干个重要的事件。假定在 Page1 的页面中有一个超链接是指向 Page2 页面的，那么当我们在 Page1 页面上点击了一下超链接之后，到底发生了什么呢？我们来仔细剖析一下这一“点击”的背后。

如果 Page1 能够成功导航到 Page2，首先会触发 NavigationService 的 Navigating 事件，标志导航开始。随后创建 Page2 对象，并且触发 NavigationProgress 事件。该事件用于提供导航进度信息，每次返回 1 KB 数据就会引发该事件。随后触发 Navigated 事件，LoadCompleted 紧随其后，这时表明页面已经下载完毕。Page1 触发 UnLoaded 事件，宣告其结束。Page2 触发 Loaded 事件，表明其开始。

如图 9-13 所示。

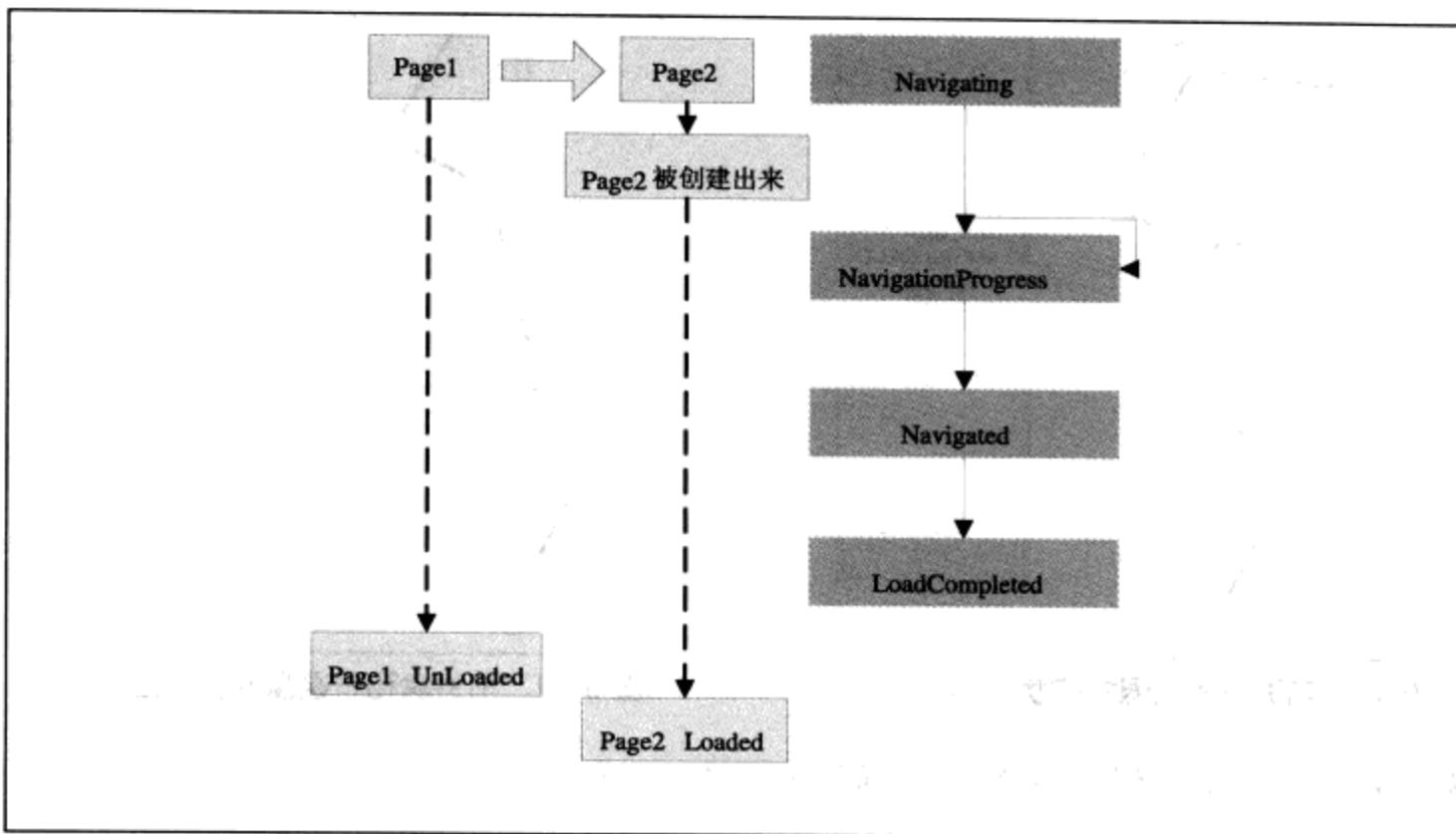


图 9-13 点击之后发生的情况

如果 Page2 可从一个段落导航到另一个段落，则如图 9-14 所示。

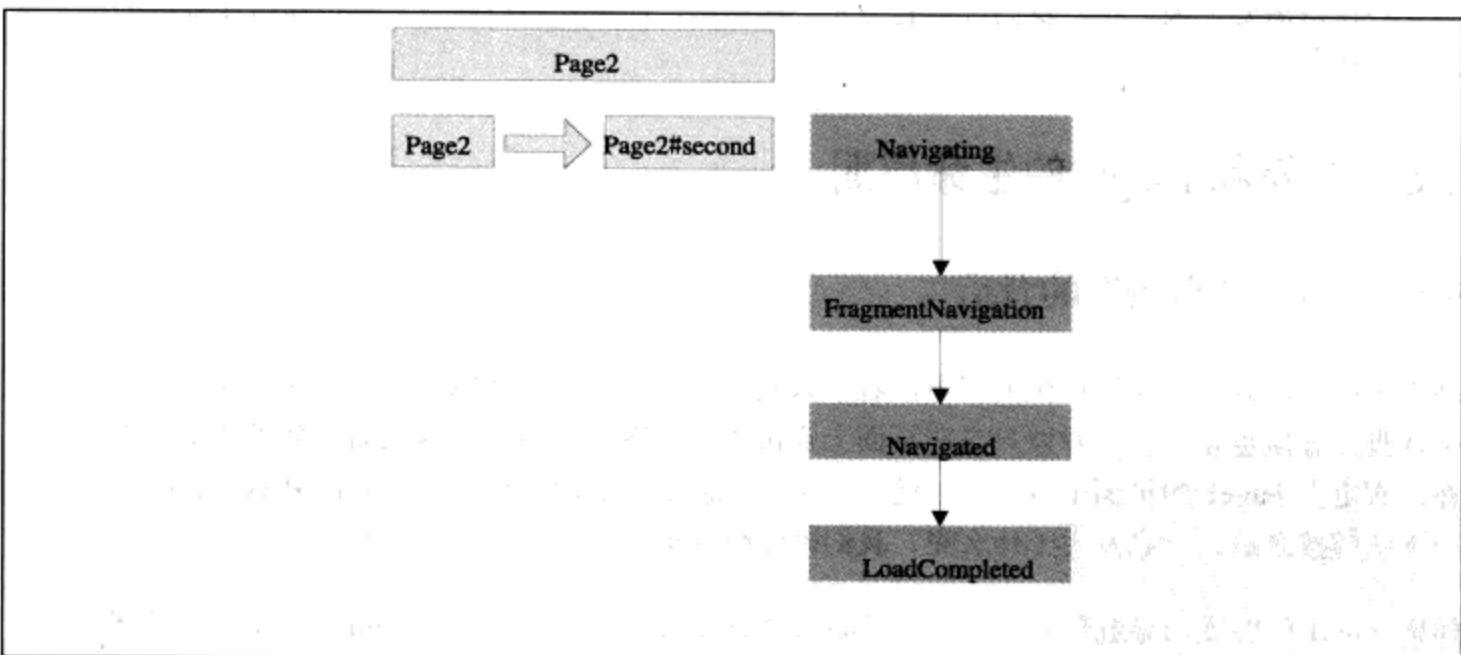


图 9-14 从一个段落导航到另一个段落

除去触发 **Navigating**、**Navigated** 和 **LoadCompleted** 事件以外，还触发 **FragmentNavigation** 事件。

如果导航失败，如从 Page1 导航到一个并不存在的页面 Page3，则触发 **NavigationFailed** 表示失败。可以在该事件处理函数中将传递过来的 **NavigationFailedEventArgs** 的 **Handled** 属性设置为 **true**，从而

防止异常继续上传转变为一个未处理的应用程序异常。随后还会触发 NavigationStopped 事件表示导航已经停止，如图 9-15 所示。

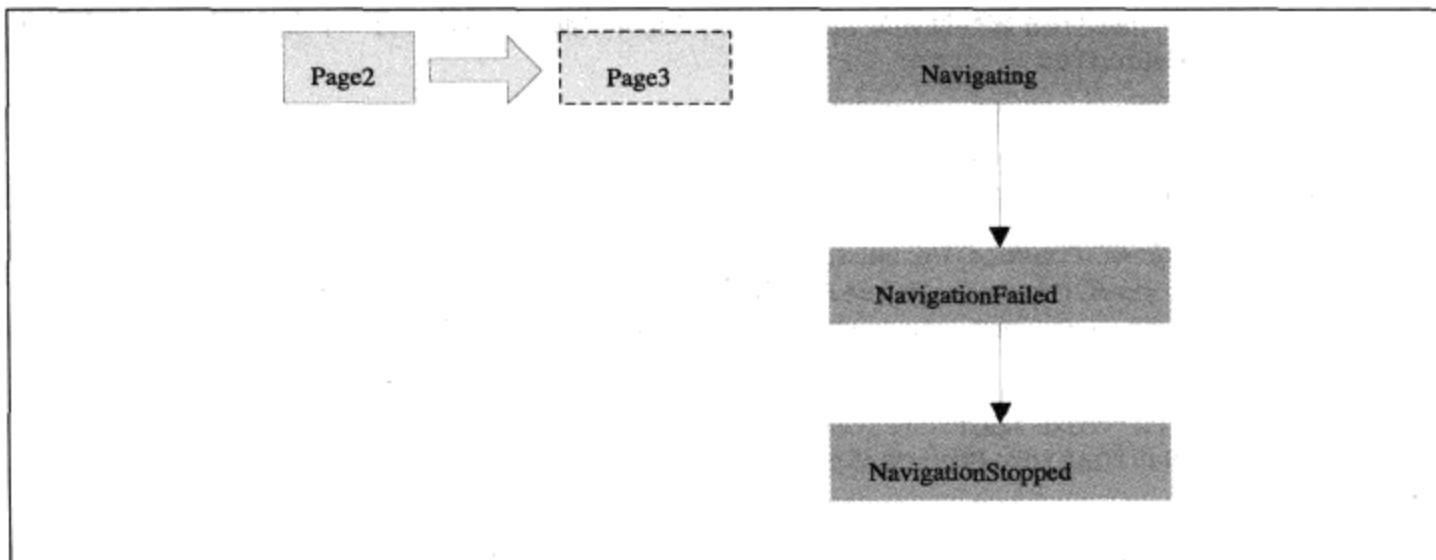


图 9-15 从 Page1 导航到了一个并不存在的页面 Page3

我们可以建立一个工程（mumu_navigationevents）来观察上面几种不同情况触发的导航事件。在实践中更多地使用 Application 类的导航事件，而不是 NavigationService 类。如代码 9-10 所示。

```
App.xaml
<Application x:Class="mumu_navigationevents.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri=" MainPage.xaml " Navigating="Application_Navigating"
    NavigationFailed="Application_NavigationFailed"
    Navigated="Application_Navigated"
    NavigationProgress="Application_NavigationProgress"
    NavigationStopped="Application_NavigationStopped"
    LoadCompleted="Application_LoadCompleted"
    FragmentNavigation="Application_FragmentNavigation"
    >
    <Application.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="18" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Application.Resources>
</Application>

App.xaml.cs
public partial class App : Application
{
    private void Application_Navigating(object sender,
System.Windows.Navigation.NavigatingCancelEventArgs e)
    {
        System.Console.WriteLine("-----");
        System.Console.Write("触发的事件为: Application_Navigating\n");
        System.Console.WriteLine("导航页面的 Uri:" + e.Uri.ToString());
    }

    private void Application_NavigationFailed(object sender,
System.Windows.Navigation.NavigationFailedEventArgs e)
```

```

    {
        System.Console.WriteLine("-----");
        System.Console.Write("触发的事件为: Application_NavigationFailed\n");
        System.Console.WriteLine("失败的异常是: " + e.Exception.ToString());
        // Handled 属性设置为 true, 从而防止异常继续上传转变为一个未处理的应用程序异常
        e.Handled = true;
    }

    private void Application_Navigated(object sender,
System.Windows.Navigation.NavigationEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.Write("触发的事件为: Application_Navigated\n");
    System.Console.WriteLine("导航页面的 Uri:" + e.Uri.ToString());
}

private void Application_NavigationProgress(object sender,
System.Windows.Navigation.NavigationProgressEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.Write("触发的事件为: Application_NavigationProgress\n");
    System.Console.WriteLine("导航页面的 Uri:" + e.Uri.ToString());
    System.Console.WriteLine("已经得到的字节数为{0}", e.BytesRead);
}

private void Application_NavigationStopped(object sender,
System.Windows.Navigation.NavigationEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.Write("触发的事件为: Application_NavigationStopped\n");
    System.Console.WriteLine("导航页面的 Uri:" + e.Uri.ToString());
}

private void Application_LoadCompleted(object sender,
System.Windows.Navigation.NavigationEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.Write("触发的事件为: Application_LoadCompleted\n");
    System.Console.WriteLine("导航页面的 Uri:" + e.Uri.ToString());
}

private void Application_FragmentNavigation(object sender,
System.Windows.Navigation.FragmentNavigationEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.Write("触发的事件为: Application_FragmentNavigation\n");
    System.Console.WriteLine("导航的段落为:" + e.Fragment);
}
}

```

代码 9-10 mumu_navigationevents 工程的部分文件

同样需要在应用程序项目属性中的 Output Type 下拉列表框中选择 Console Application 选项，以观察触发的事件¹。该程序由 3 个页面组成，即 MainPage、Page1 和 Page2，其中 MainPage 中嵌入了一个 Frame，它是 Page1 和 Page2 的容器。Page1 的页面中包括如下 4 种导航情况。

- (1) 正常导航。
- (2) 导航到一个不存在的页面。
- (3) 导航到一个页面，但是随即停止。
- (4) 刷新当前页面。

Page1 的代码如代码 9-11 所示。

```
Page1.xaml
<Page x:Class="mumu_navigationevents.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page1" Loaded="Page_Loaded" Unloaded="Page_Unloaded">
    <StackPanel>
        <TextBlock>
            从 Page1 导航到
            <Hyperlink NavigateUri="Page2.xaml">
                Page2.xaml
            </Hyperlink>
        </TextBlock>
        <TextBlock>
            从 Page1 导航到
            <Hyperlink NavigateUri="Page3.xaml">
                一个不存在的网页
            </Hyperlink>
        </TextBlock>
        <TextBlock>
            从 Page1 导航到
            <Hyperlink Click="Hyperlink_Click">
                Page2.xaml
            </Hyperlink>
            但是随即会取消
        </TextBlock>
        <TextBlock>
            <Hyperlink Click="Hyperlink_Click_1">
                刷新
            </Hyperlink>
            当前页面
        </TextBlock>
    </StackPanel>
</Page>

Page1.xaml.cs
public partial class Page1 : Page
{
    public Page1()
    {
        System.Console.WriteLine("-----");
    }
}
```

¹参见第 8 章。

```

        System.Console.WriteLine("Page1 被创建出来");
        InitializeComponent();
    }

private void Hyperlink_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("pack://application:,,,/Page2.xaml"));
    Thread.Sleep(500);
    NavigationService.StopLoading();
}

private void Hyperlink_Click_1(object sender, RoutedEventArgs e)
{
    NavigationService.Refresh();
}

private void Page_Loaded(object sender, RoutedEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.WriteLine("Page1 Loaded");
}

private void Page_Unloaded(object sender, RoutedEventArgs e)
{
    System.Console.WriteLine("-----");
    System.Console.WriteLine("Page1 UnLoaded");
}
}

```

代码 9-11 Page1.xaml 和 Page1.xaml.cs 文件

可以在 Console 窗口中看到一次正常导航触发的事件、Page 的装载和卸载过程，如图 9-16 所示。

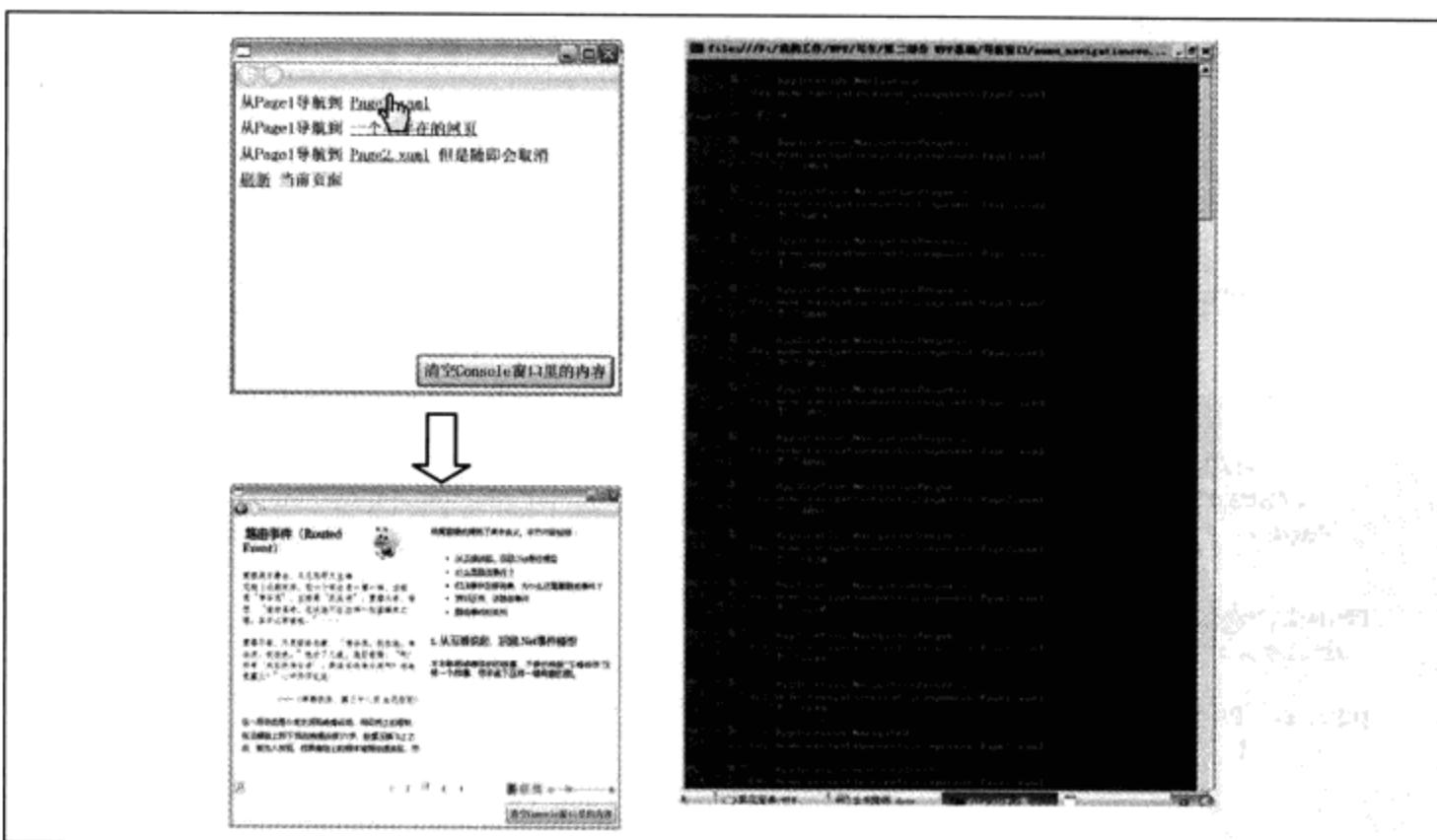


图 9-16 一次正常的导航所触发的事件

9.5.2 Page 的生命周期

在运行 `mumu_navigationevents` 程序时，可发现“Page1 被创建出来”及“Page1 Loaded”等字样，这实际上是 Page 创建、加载和卸载的生命周期。正常情况下每次导航都伴随 Page 的生和死，图 9-17 所示为 Page 的生命周期。当导航到该页面时创建页面，同时装载并触发 Loaded 事件；在离开该页面时则卸载该页面，同时触发 UnLoaded 事件。

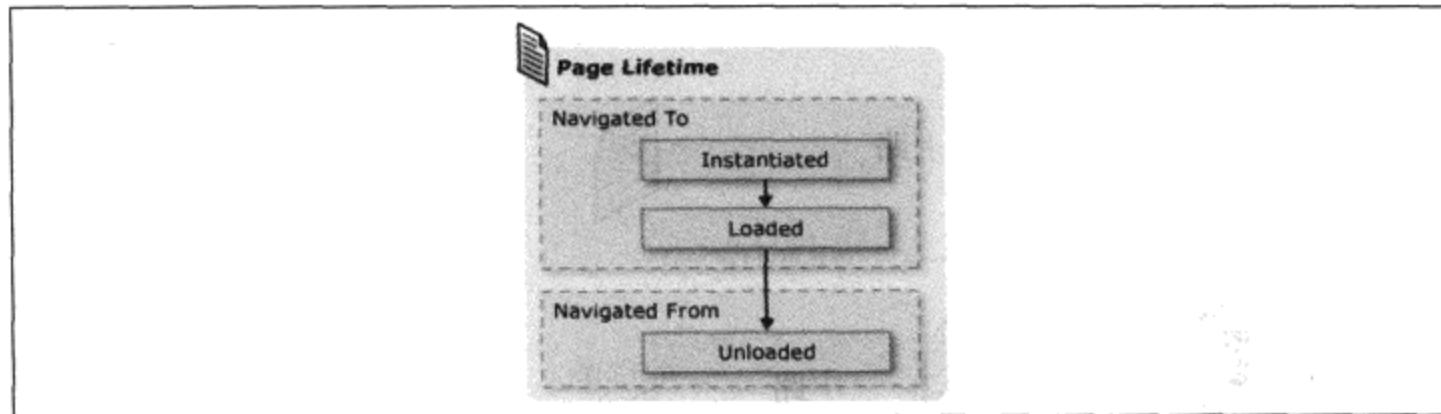


图 9-17 Page 的生死图之一^[3]

上面是正常情况，但有些问题。如一个页面中包括图片及视频等信息。每次导航都会重新创建页面，从而导致运行效率比较低。有一种解决办法是将 Page 的 `KeepAlive` 属性设置为 `true`，那么该页面就驻留在内存中。这样导航时不会重新创建该页面，而仅触发 `Loaded` 和 `UnLoaded` 事件，如图 9-18 所示。

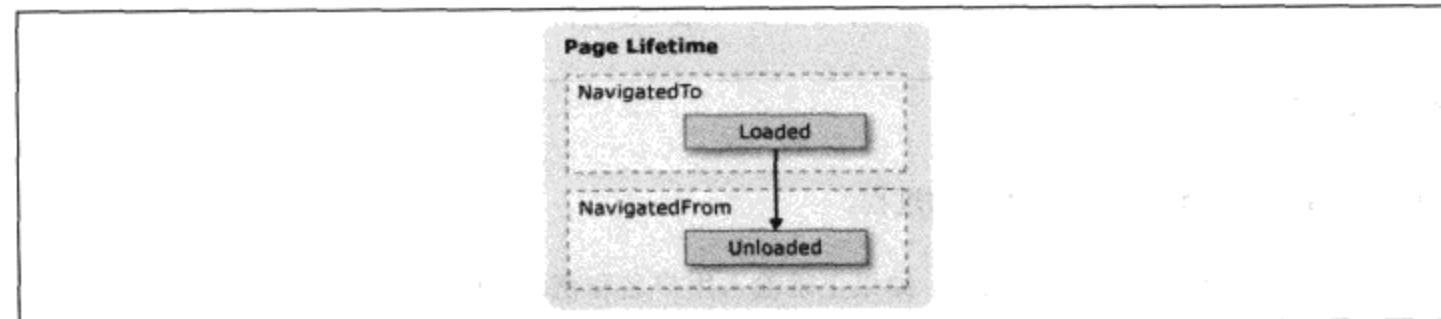


图 9-18 Page 的生死图之二^[3]

9.6 页面状态保留和数据传递

“页面状态保留和数据传递”听起来有些复杂，但是实际上很多基于导航的应用程序都会涉及到这两个问题。相信大家都有在网上注册用户的不愉快经历吧。比如在一个“注册页面”上填写用户名、密码、再次确认密码和验证码等一大堆东西之后，点击确认后，却弹出一个“错误页面”提示说“您的用户名已经注册”。这个时候再返回到“注册页面”时，让人崩溃的是所有填写的东西都为空，无奈只好再次重新填写一次。但是如果我们将返回的时候，页面除去密码外，其他信息都还保留在网页上。你一定会觉得这个网站还颇为人性化。这就是导航时的**页面状态保留问题**。

当输入正确的用户名和密码时出现“欢迎页面”，提示“欢迎***（如木木）”。“欢迎页面”知道

用户名是因为“登录页面”将用户名传递给它。这是一种导航时的**数据传递问题**，是由前面一个页面向后面一个页面传递数据（**由前向后的数据传递**）；另一种情况是在“注册页面”中正确地填写注册信息并单击“确定”按钮后返回到“登录页面”，这时该页面中已经完全填写用户名和密码信息，这是因为“注册页面”向“登录页面”传递了用户名和密码信息。与前面的数据传递不同，这是一种**由后向前的数据传递**。

图 9-19 所示的登录流程中出现多个页面状态保留和数据传递的环节。

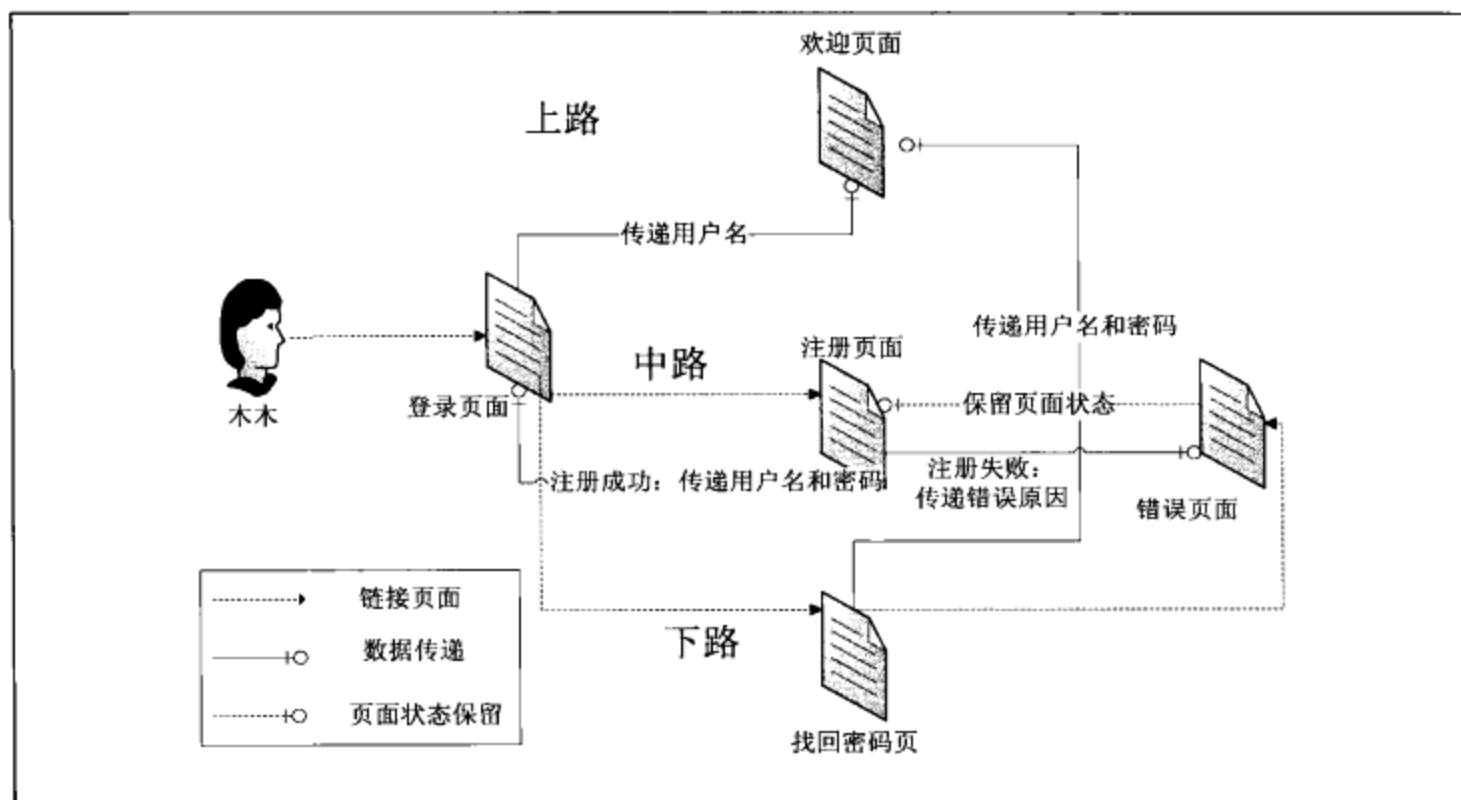


图 9-19 登录流程

该图分为上中下 3 路，即 3 个不同的过程。

(1) 上路：用户进入“登录页面”，如果正确输入用户名和密码，则可进入“欢迎页面”。两个页面之间存在一次数据传递，即“欢迎页面”会根据“登录页面”传递的用户名（如木木）显示“欢迎你木木”。

(2) 中路：用户进入“登录页面”，如果未注册过，则可单击“注册”按钮进入“注册页面”。其中会出现如下两种情况。

- 注册失败：“注册页面”会传递注册失败的原因给“错误页面”，显示错误原因提示用户。当用户单击“返回”链接时会返回“注册页面”，此时“注册页面”保留除密码信息以外的其他信息。
- 注册成功：“注册页面”传递用户名和密码给“登录页面”，其中直接填写用户名和密码。

(3) 下路：如果用户忘记了密码并希望找回，则单击“忘记密码了……”链接进入“找回密码页”，也会出现如下两种情况。

- 回答密码保护问题出错：进入“错误页面”。
- 回答密码保护问题正确：“找回密码页”将用户名和密码传递给“欢迎页面”，由其负责显示正确的用户名和密码。

9.6.1 构建登录应用程序

首先构建一个简单的用户数据结构 User 类，除了包含姓名 Name 和密码 Password 属性，还有一个用户颜色的集合 FavColors，该属性主要作为用户密码保护问题的答案。当用户忘记密码时，如果能够正确地回答喜欢的颜色，即可要回原来的密码，如代码 9-12 所示。

```
User.cs
public class User
{
    private string _name;
    private string _password;
    private List<string> _favColors;

    public User() { }
    public User(string name, string password)
    {
        this._name = name;
        this._password = password;
        _favColors = new List<string>();
    }
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public string Password
    {
        get { return _password; }
        set { _password = value; }
    }
    public List<string> FavColors
    {
        get { return _favColors; }
        set { _favColors = value; }
    }
    public override string ToString()
    {
        return this._name;
    }
}
```

代码 9-12 mumu_dataretainandtransfer 工程中的 User.cs 文件

保存用户集合信息的最佳处为 App 类，这是因为一个应用程序对象 App 是唯一的，并且所有的页面都可以通过 App.Current 访问应用程序的对象。App.xaml 和 App.xaml.cs 文件的内容如代码 9-13 所示。

```
App.xaml
<Application x:Class="mumu_dataretainandtransfer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Startup="Application_Startup">
    <Application.Resources>
```

```

    </Application.Resources>
</Application>

App.xaml.cs
public partial class App : Application
{
    public List<User> users;
    private void Application_Startup(object sender, StartupEventArgs e)
    {
        users = new List<User>();
        User user = new User("木木", "mumu");
        user.FavColors.Add("红色");
        user.FavColors.Add("绿色");
        users.Add(user);
        NavigationWindow win = new NavigationWindow();

        win.Width = 480;
        win.Height = 400;
        win.Content = new LoginPage();
        win.Show();
    }
}

```

代码 9-13 App.xaml 和 App.xaml.cs 文件

在 App 的 Startup 事件处理函数中首先初始化用户列表，并且默认新建一个用户“木木”。随后新建一个 NavigationWindow 对象，并将“登录页面”（LoginPage）赋值给 NavigationWindow 对象的 Content 属性。

“登录页面”如图 9-20 所示。

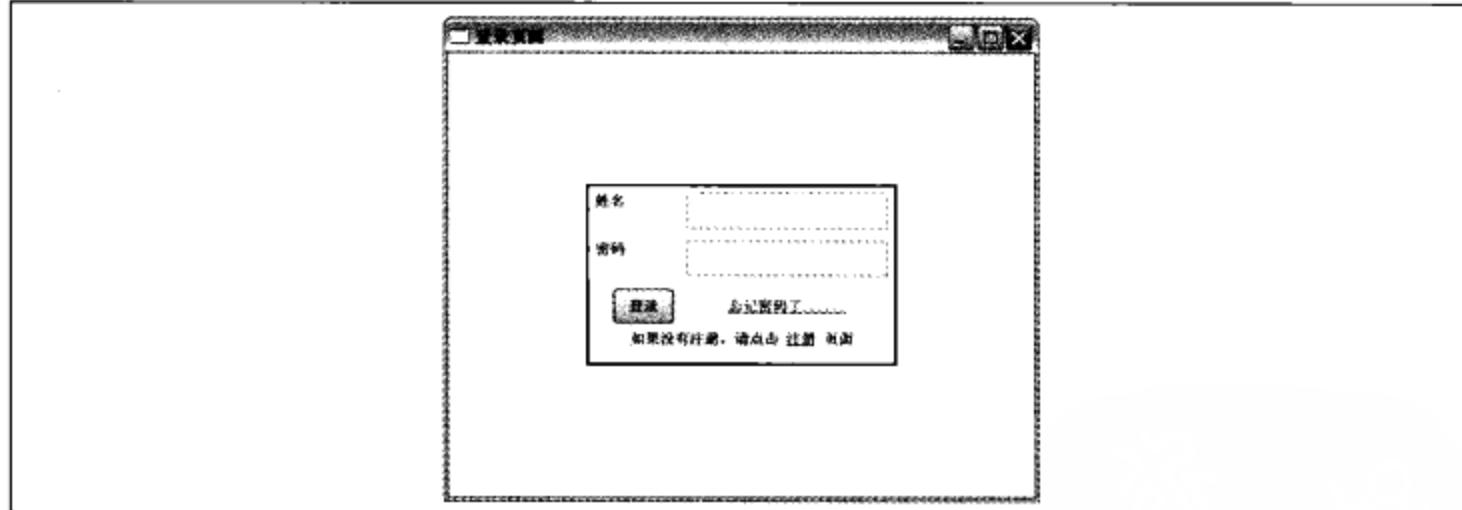


图 9-20 “登录页面”

代码 9-14 是 LoginPage 的 XAML 实现。

```

LoginPage.xaml
<Page x:Class="mumu_dataretainandtransfer.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
①     ShowsNavigationUI="False"
      Title="登录页面" WindowTitle="登录页面" >

```

```

        <Border BorderBrush="Black" BorderThickness="2" Height="150"
Width="250">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="3*"/>
            <ColumnDefinition Width="7*"/>
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="0" Text="用户名"
Margin="5"/>
        <TextBox Grid.Row="0" Grid.Column="1" Margin="5"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Text="密码"
Margin="5"/>
        <PasswordBox Grid.Row="1" Grid.Column="1" Margin="5"/>
        <Button x:Name="btn" Grid.Row="2"
HorizontalAlignment="Right" Width="50" Margin="5" Grid.Column="0"
② Click="Button_Click" >
            登录
        </Button>
        <TextBlock Grid.Row="2" Grid.Column="1" VerticalAlignment=
"Center" HorizontalAlignment="Center">
            <Hyperlink NavigateUri="GetPasswordPage.xaml">
                忘记密码了……
            </Hyperlink>
        </TextBlock>
        <TextBlock Margin="0" Grid.Row="3" Grid.ColumnSpan="2"
x:Name="hyperlinktext" HorizontalAlignment="Center" VerticalAlignment=
"Center">
            如果没有注册, 请单击
            <Hyperlink>
                注册
            </Hyperlink>
            页面
            <LineBreak />
        </TextBlock>
    </Grid>
</Border>
</Page>

```

代码 9-14 LoginPage 的 XAML 实现

我们不希望显示导航工具栏, 因此将该页面的 ShowsNavigationUI 属性设置为 False (代码①), 并且为登录按钮添加 Click 事件处理函数 (代码②)。

9.6.2 由前向后传递数据

参见代码 9-15, 在登录按钮 Click 事件处理函数中需要判断是否正确输入用户名和密码。如果正确, 则导航到“欢迎页面”(WelcomePage, 代码①); 否则导航到“错误页面”(ErrorPage, 代码③)。注意一定要添加 return (代码②); 否则均导航到“错误页面”。

```

LoginPage.xaml.cs
private void Button_Click(object sender, RoutedEventArgs e)

```

```

    {
        List<User> users = ((App)(App.Current)).users;
        int usercount = users.Count;
        User user = new User(name.Text, password.ToString());
        for (int i = 0; i < usercount; i++)
        {
            if (name.Text == users[i].Name && password.Password ==
users[i].Password)
            {
                ①     WelcomePage page = new WelcomePage(user, false);
                NavigationService.Navigate(page);
                ②     return;
            }
        ③     NavigationService.Navigate(new
Uri("pack://application:,,,/ErrorPage.xaml"));
        }
    }

```

代码 9-15 登录按钮 Click 事件处理函数实现

从“登录页面”到“欢迎页面”属于由前向后传递数据，这种传递通常通过如下两种方法实现。

- (1) “欢迎页面”实现一个带参数的构造函数，在创建页面时将数据传递到其中，然后导航到“欢迎页面”。
- (2) “欢迎页面”暴露一个公共属性，创建“欢迎页面”。然后将数据赋值给其暴露的公共属性，并导航到该页面。

这里采用的是第 1 种方法，“欢迎页面”实现了一个带参数的构造函数。它的第 1 个参数是用户信息；第 2 个是一个 bool 类型的变量，用来控制密码是否可见²，如代码 9-16 所示。

```

WelcomePage.xaml.cs
public partial class WelcomePage : Page
{
    public WelcomePage()
    {
        InitializeComponent();
    }
    public WelcomePage(User user, bool isPasswordVisible) :this()
    {
        welcome.Text = "欢迎你" + user.Name;
        if (isPasswordVisible)
            welcome.Text += "\n你的密码为：" + user.Password;
    }
}

```

代码 9-16 WelcomePage.xaml.cs 文件

从“登录页面”链接到“欢迎页面”如图 9-21 所示。

²这里不需要“欢迎页面”显示密码，所以第 2 个参数值为 false。但是从“找回密码页”到“欢迎页面”需要“欢迎页面”告诉用户忘记的密码，因此第 2 个参数值为 true。

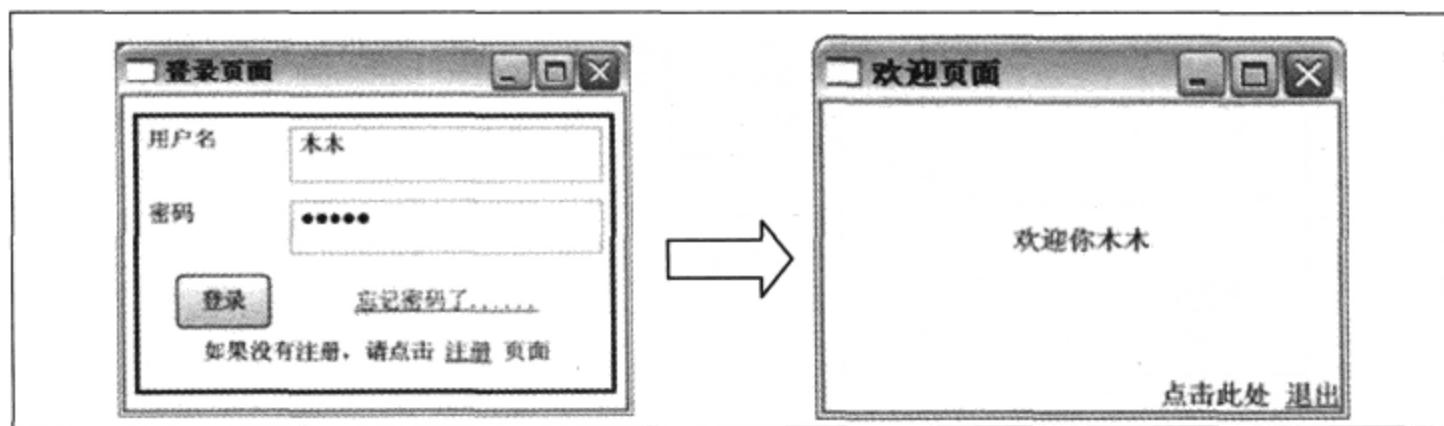


图 9-21 从“登录页面”链接到“欢迎页面”

9.6.3 WPF 固有的页面状态保留机制

如果登录失败，则会从“登录页面”链接到“错误页面”，如图 9-22 所示。

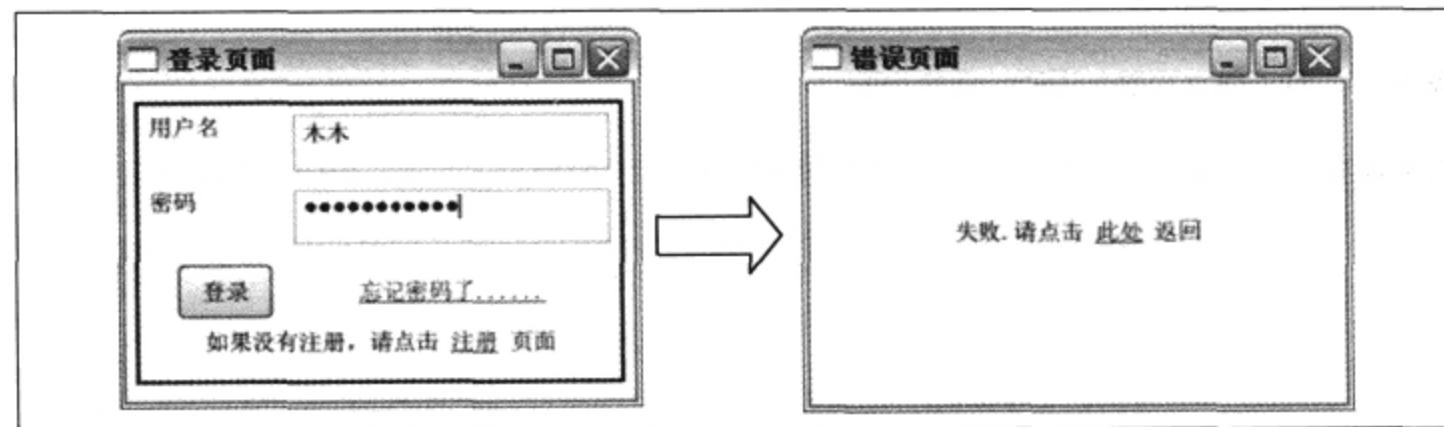


图 9-22 从“登录页面”链接到“错误页面”

如果从“错误页面”返回到“登录页面”时，你会很意外地发现，任何工作都没有做，用户名的信息保留了，密码信息并没有保留。这样的结果恰恰是我们希望的。当离开了某一个页面，再返回该页面时，WPF 有这样几类控件会自动地保留自己的信息^[3]：CheckBox、ComboBox、Expander、Frame、ListBox、ListBoxItem、MenuItem、ProgressBar、RadioButton、Slider、TabControl、TabItem 和 TextBox。

9.6.4 使用依赖属性保留简单的页面状态信息

从“错误页面”返回“登录页面”，“用户名”和“密码”文本框均没有焦点，需要重新将鼠标移到其中一个文本框使之获得输入焦点，如图 9-23 所示。

如果希望记录焦点最后所在的文本框，下次返回时该文本框自动获得焦点。这是一种非常细微的改进，但是它更大的意义在于说明如何用依赖属性保留简单的状态信息。

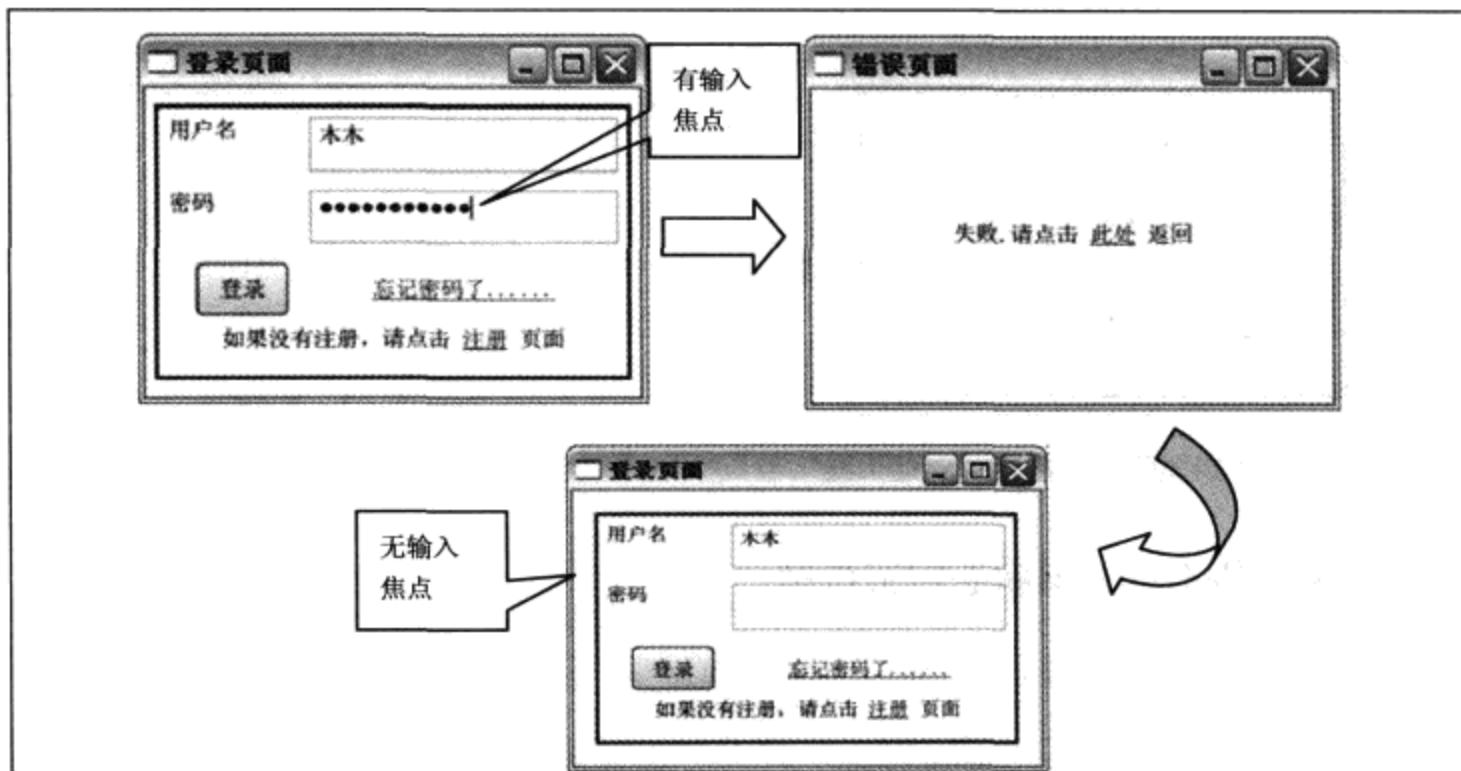


图 9-23 从“错误页面”返回“登录页面”

首先需要在 LoginPage 中定义一个依赖属性用来保存最后焦点获得的元素，如代码 9-17 所示。

```
public static readonly DependencyProperty FocusElementProperty;
public string FocusElement
{
    get
    {
        return (string)base.GetValue(LoginPage.FocusElementProperty);
    }
    set
    {
        base.SetValue(LoginPage.FocusElementProperty, value);
    }
}
```

代码 9-17 定义 FocusElement 依赖属性

定义一个静态函数注册该依赖属性，如代码 9-18 所示。

```
LoginPage.FocusElementProperty = DependencyProperty.Register("FocusElement",
typeof(string), typeof(LoginPage), newFrameworkPropertyMetadata(null,
FrameworkPropertyMetadataOptions.Journal));
```

代码 9-18 注册依赖属性

注意注册时需要将其元数据的第 2 个参数设置为 FrameworkPropertyMetadataOptions.Journal。这样设置的原因何在呢？我们还是从 WPF 的页面状态保留基本机制说起。

我们不要以为 WPF 会变魔术，什么都不做就能够将页面状态保留起来。实际上当导航离开页面时 WPF 选择一个位置保留控件的当时状态，然后再次导航到页面时将保存的数据重新赋值给该页面。现在关键的问题是 WPF 找的地方在哪儿？实际上有两个地方：一个是 Cookie；另一个则是 Journal。

Journal 维护两个历史列表，其中放置的是一个一个 JournalEntry。JournalEntry 中保留页面的状态信息。FrameworkPropertyMetadataOptions.Journal 选项就是要告诉 WPF 这个依赖属性将存在 JournalEntry 中，并纳入导航的历史管理，如图 9-24 所示。

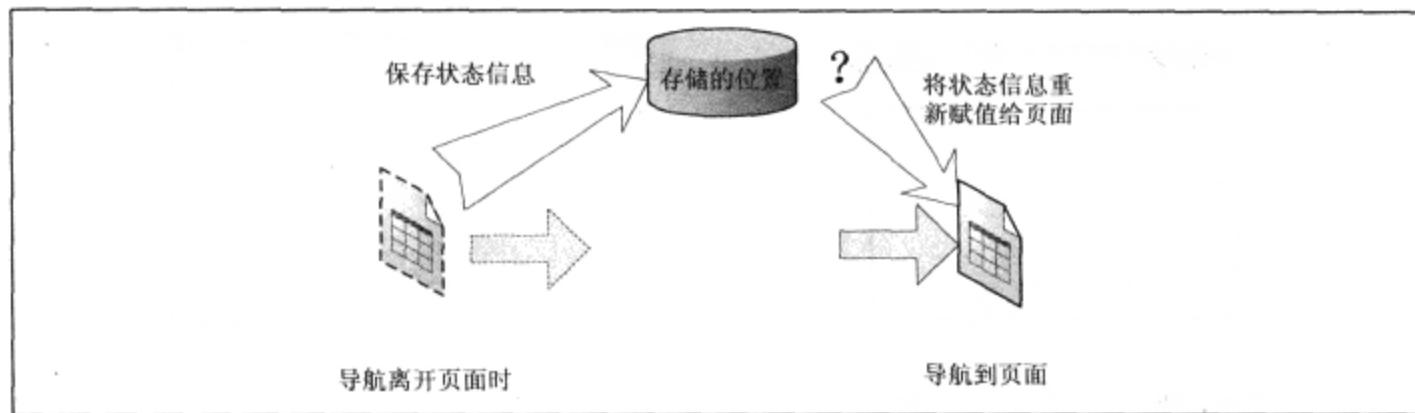


图 9-24 WPF 页面状态保留原理图

在 Page 的 PreviewLostKeyboardFocus 事件处理函数中记录焦点所在的文本框信息，如代码 9-19 所示。

```
private void Page_PreviewLostKeyboardFocus(object sender,
KeyboardFocusChangedEventArgs e)
{
    if (e.NewFocus == this.name || e.NewFocus == this.password)
    {
        this.FocusElement =
(string)((DependencyObject)e.NewFocus).GetValue(FrameworkElement.NameProperty));
    }
}
```

代码 9-19 Page 的 PreviewLostKeyboardFocus 事件处理函数

在 Page 的 Loaded 事件处理函数中根据记录的信息重新将焦点设置到相应的文本框中，如代码 9-20 所示。

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    if (this.FocusElement != null)
    {
        IInputElement element =
(IInputElement)LogicalTreeHelper.FindLogicalNode(this, this.FocusElement);
        Keyboard.Focus(element);
    }
}
```

代码 9-20 Page 的 Loaded 事件处理函数

9.6.5 由后向前传递数据方法的 PageFunction

再来看看中间一路，即从“登录页面”到“注册页面”，如图 9-25 所示。

在“登录页面”中单击“注册”，链接到“注册页面”。当“注册页面”注册成功需要向“登录页面”返回用户名和密码时是一种由后向前的数据传递，如果在普通的窗口程序里，这种由后向前的

数据传递通常由一个模态对话框实现。但是在导航程序中总是从一个页面链接到另一个页面，模态对话框是难以实现的。WPF 提供了一个 `PageFunction<T>` 模板类来取代模态对话框，它从 `Page` 派生而来，如图 9-26 所示。

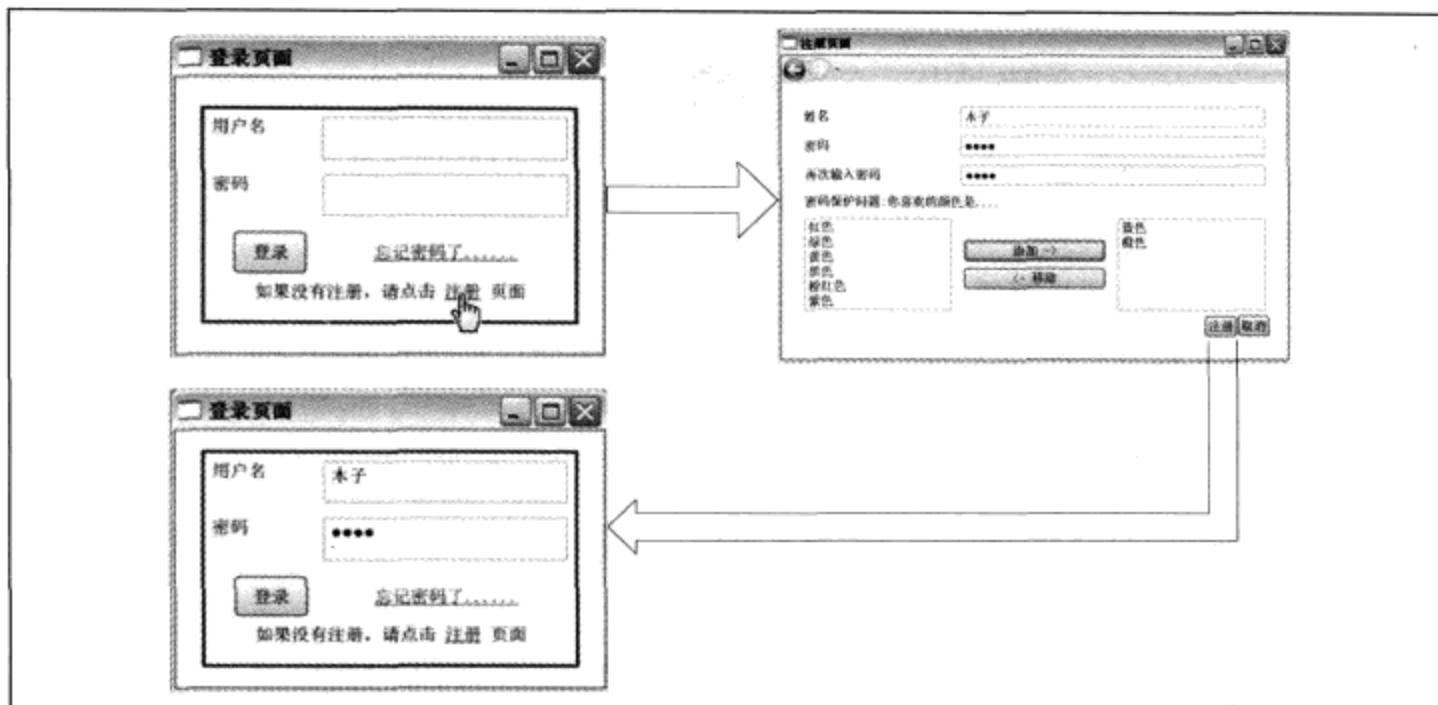


图 9-25 从“登录页面”到“注册页面”

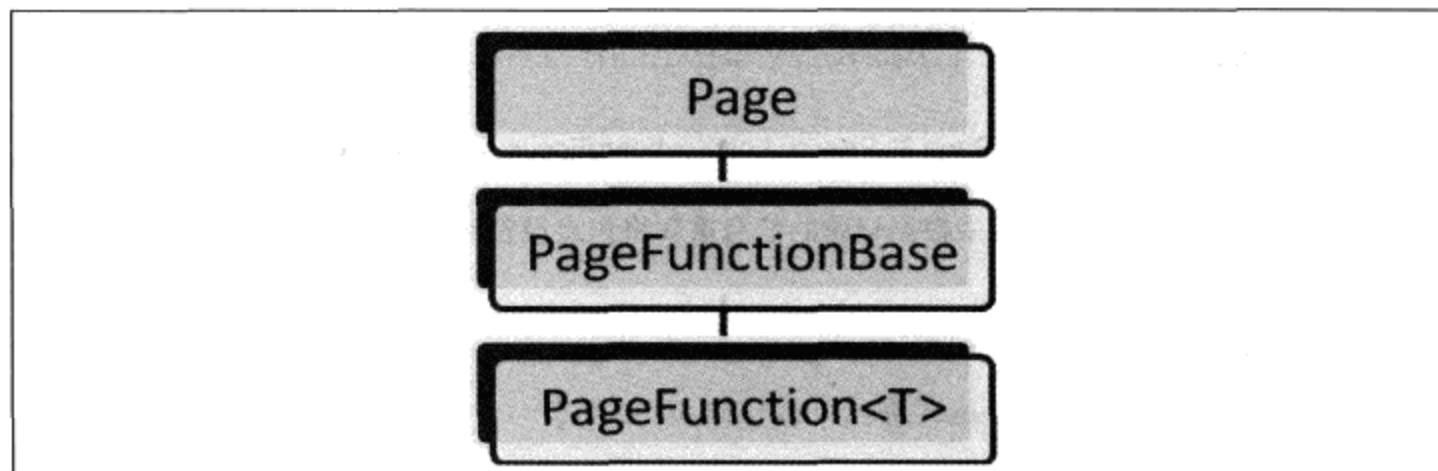


图 9-26 PageFunction<T>模板类的派生关系

`PageFunction<T>` 中的 `T` 代表的是需要传回来的数据类型，这种类似模态对话框的行为在导航程序中称为“结构化导航”（Structured Navigation）。“注册页面”不能派生自 `Page`，而派生自 `PageFunction<User>`。表示传递回来的数据是一个用户信息，如代码 9-21 所示。

```
public partial class RegisterPage : PageFunction<User>
```

代码 9-21 RegisterPage 从 PageFunction<User>派生而来

由于在 XAML 文件中没有模板这个概念，所以声明会稍稍复杂一些，如代码 9-22 所示：

```
① <PageFunction x:Class="mumu_dataretainandtransfer.RegisterPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local ="clr-namespace:mumu_dataretainandtransfer"
②  x:TypeArguments="local:User"
    WindowTitle="注册页面"
    Title="Page1" Margin="20" Loaded="Page_Loaded" ShowsNavigationUI="True">
    .....
        <WrapPanel Grid.Row="5" Grid.Column="2"
    HorizontalAlignment="Right">
    ③      <Button Content="注册" Click="cmdOK_Click"/>
    ④      <Button Content="取消" Click="cmdCancel_Click"/>
    </WrapPanel>
    .....
</PageFunction>

```

代码 9-22 RegisterPage.xaml 文件

XAML 文件中不再是 Page 标签，而是 PageFunction 标签（代码①），它使用 TypeArguments 属性取代了模板的概念。由于需要使用自定义类 User，因此需要声明工程的命名空间，以便于引用（代码②）。注册按钮和取消按钮的事件处理函数实际上类似模态对话框中的确认和取消按钮（代码③和④），代码 9-23 是这两个按钮的事件处理函数的实现。

```

RegisterPage.xaml.cs
public partial class RegisterPage : PageFunction<User>
{
    private User user;

    public RegisterPage()
    {
        InitializeComponent();
        user = new User();
    }
    .....
    private void cmdOK_Click(object sender, RoutedEventArgs e)
    {
        User user = CreateUser();
        if (user == null) return;
        else
            OnReturn(new ReturnEventArgs<User>(user));
    }

    private void cmdCancel_Click(object sender, RoutedEventArgs e)
    {
        OnReturn(null);
    }
    .....

```

代码 9-23 两个按钮的事件处理函数的实现

在注册按钮的事件处理函数中成功创建一个用户，则调用 OnReturn 函数将 User 返回给“登录页面”；在取消按钮的事件处理函数中同样调用函数，只不过传递了一个 null 参数，表示没有数据返回给“登录页面”。“登录页面”接收返回的数据，如代码 9-24 所示。

```

private void Hyperlink_Click(object sender, RoutedEventArgs e)
{
    RegisterPage CalledPageFunction = new RegisterPage(true);
    CalledPageFunction.Return += pageFunction_Return;
    this.NavigationService.Navigate(CalledPageFunction);
}

```

```

void pageFunction_Return(object sender, ReturnEventArgs<User> e)
{
    if(e == null)
    {
        return;
    }
    User user = (User)e.Result;
    if (user != null)
    {
        this.name.Text = user.Name;
        this.password.Password = user.Password;
    }
    List<User> users = ((App)(App.Current)).users;
    users.Add(user);
}

```

代码 9-24 “登录页面”如何接受返回过来的数据

在单击“注册”链接时，“登录页面”为“注册页面”添加一个 Return 的事件处理函数。当“登录页面”调用 OnReturn 时，触发该事件处理函数。该函数的第 2 个参数是 OnReturn 传递的参数，pageFunction_Return 根据该参数在用户名和密码文本框中填写返回的用户信息，并把用户添加到用户列表中。

9.6.6 使用 IProvideCustomContentState 接口保留复杂的页面状态信息

如果在“注册页面”中注册信息失败，如前后两次输入的密码不同，则链接到“错误页面”后返回。虽然姓名文本框仍然保留原来的信息，但是用户密码已经不存在（虚框），如图 9-27 所示。

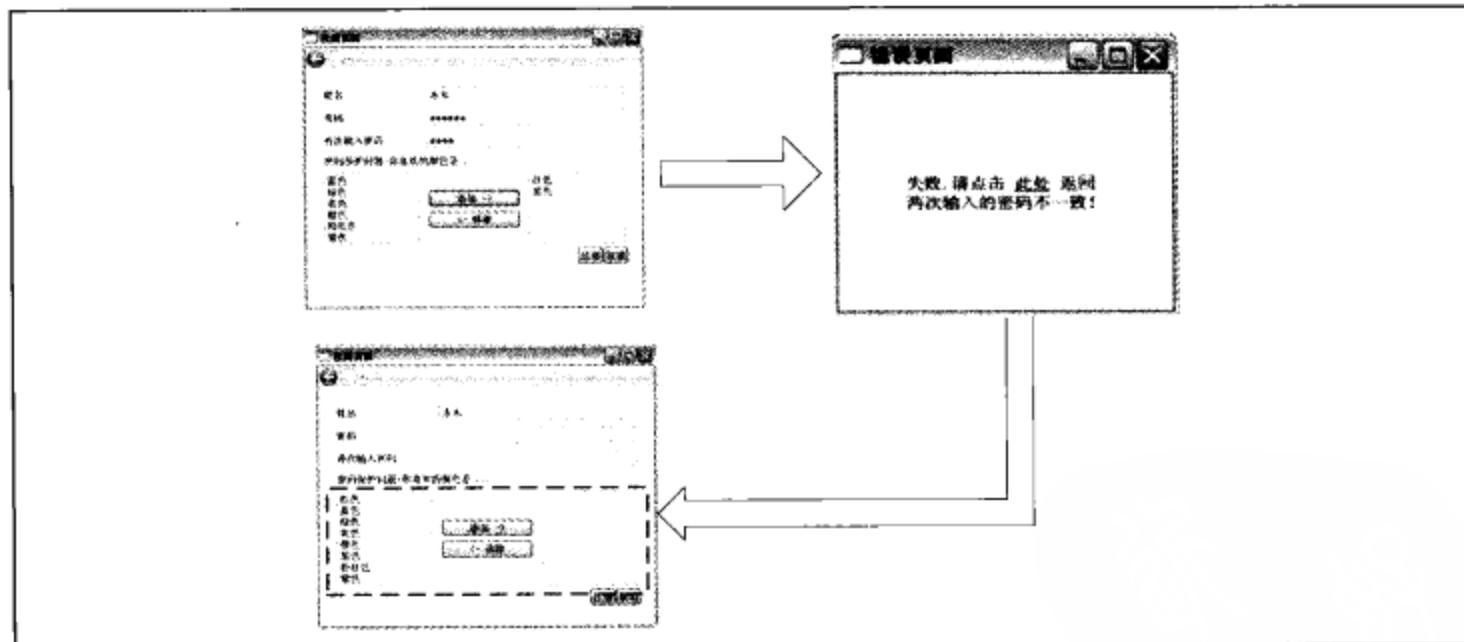


图 9-27 链接到“错误页面”

同样也可以使用依赖属性来保存两个列表框的状态信息，但是其中的数据结构更为复杂。因此采用另一种方法 IProvideCustomContentState 接口来保留页面状态信息更为合适，其基本原理和使用依赖属性保存页面状态信息相同。只不过 JournalEntry 提供了一个 CustomContentState 类来保存页面状态信息，该类有一个需要重载的属性和方法。重载的属性为 JournalEntryName，重载该属性之后本次

操作的名称会显示在导航栏中。重载的方法为 Replay，该方法负责将保存的页面信息恢复到页面中。需要保留页面状态信息的页面应实现 IProvideCustomContentState 接口，该接口有一个 GetContentState 方法需要重载。当导航离开页面时 WPF 会调用该方法，将页面状态的信息保存在 JournalEntry 中。

当导航离开页面时，由于该页面实现了 IProvideCustomContentState 接口，所以 WPF 会调用 GetContentState 方法将页面信息保存在相应 JournalEntry 的 CustomContentState 属性中；重新导航到页面时，WPF 会调用 CustomContentState 的 Replay 方法将原来的状态信息恢复到页面中，如图 9-28 所示。

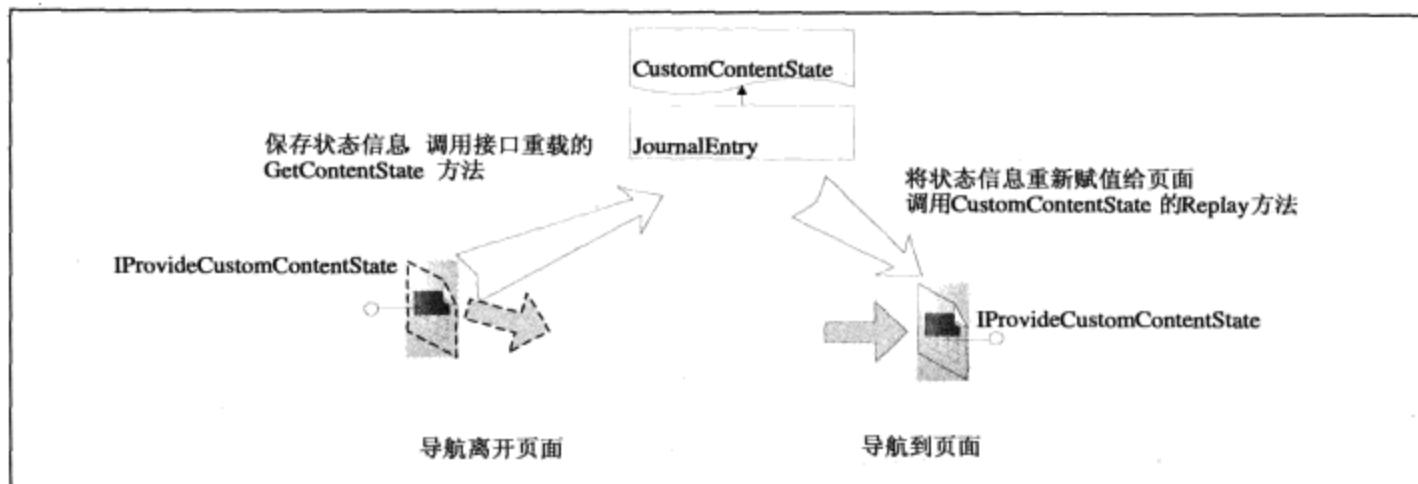


图 9-28 使用 IProvideCustomContentState 接口来保留页面状态信息

导航离开“注册页面”时需要保存的是两个 ListBox 中的颜色，因此设计如下一个派生自 CustomContentState 的类，如代码 9-25 所示。

```
① [Serializable()]
public class ListSelectionJournalEntry : CustomContentState
{
    ②     private List<String> _sourceItems;
    public List<String> SourceItems
    {
        ③         get { return _sourceItems; }
    }
    private List<String> _targetItems;
    public List<String> TargetItems
    {
        ④         get { return _targetItems; }
    }

    public override string JournalEntryName
    {
        ⑤         get
        {
            return journalName;
        }
    }
    ⑥     private string journalName;

    public ListSelectionJournalEntry(
        List<String> sourceItems, List<String> targetItems,
```

```

        string journalName)
    {
        this._sourceItems = sourceItems;
        this._targetItems = targetItems;
        this.journalName = journalName;
    }

⑤     public override void Replay(NavigationService
navigationService, NavigationMode mode)
{
    RegisterPage page = navigationService.Content as RegisterPage;
    if (page == null) return;

    page.lstSource.Items.Clear();
    foreach (string item in SourceItems)
    { page.lstSource.Items.Add(item); }

    page.lstTarget.Items.Clear();
    foreach (string item in TargetItems)
    { page.lstTarget.Items.Add(item); }

    page.restoredStateName = JournalEntryName;
}
}

```

代码 9-25 ListSelectionJournalEntry 的实现

ListSelectionJournalEntry 的属性 SourceItems 和 TargetItems（代码②和③）用来保存两个列表中的字符串，在重载的 Replay 方法中（代码⑤）将这两个属性保存的内容重新放在 Page 页面中。需要注意的是该类必须声明是可序列化的（代码①）。

在“注册页面”中需要实现 IProvideCustomContentState 接口，如代码 9-26 所示。

```

①     public partial class RegisterPage : PageFunction<User>,
IProvideCustomContentState
{
    private User user;
    public RegisterPage()
    {
        InitializeComponent();
        user = new User();
    }

②     public RegisterPage(bool isload)
        : this()
    {
        this.isload = isload;
    }
    bool isload = false;
    ③     private void Page_Loaded(object sender, RoutedEventArgs e)
    {
        if (isload)
        {
            lstSource.Items.Add("红色");
            lstSource.Items.Add("蓝色");
            lstSource.Items.Add("绿色");
            lstSource.Items.Add("黄色");
            lstSource.Items.Add("橙色");
        }
    }
}

```

```

        lstSource.Items.Add("黑色");
        lstSource.Items.Add("粉红色");
        lstSource.Items.Add("紫色");
    }
}

④ public CustomContentState GetContentState()
{
    string journalName;
    if (restoredStateName != "")
        journalName = restoredStateName;
    else
        journalName = "RegisterPage";
    return GetJournalEntry(journalName);
}

public string restoredStateName;
private ListSelectionJournalEntry GetJournalEntry(string
journalName)
{
    List<String> source = GetListState(lstSource);
    List<String> target = GetListState(lstTarget);

    return new ListSelectionJournalEntry(
        source, target, journalName);
}

private List<String> GetListState(ListBox list)
{
    List<string> items = new List<string>();
    foreach (string item in list.Items)
    {
        items.Add(item);
    }
    return items;
}

```

代码 9-26 RegisterPage 类需要实现 IProvideCustomContentState 接口

在 GetContentState(代码④)中返回一个保存当前页面两个 ListBox 状态的 ListSelectionJournalEntry，这里也存在一个小问题。由于从“登录页面”到“注册页面”时需要初始化两个 ListBox，但是从“错误页面”返回到“登录页面”时则不需要。因此为“注册页面”增加一个带参数的构造函数，表示是否需要初始化 ListBox(代码②)。这样从“登录页面”链接到“注册页面”时需要明确调用参数值为 true 的构造函数，使“注册页面”初始化 ListBox；从“错误页面”返回“登录页面”时默认调用无参数的构造函数，不会初始化 ListBox。

如果希望每次添加或者移除一个喜欢的颜色均记录在历史导航中，则只需要在添加和移除按钮的事件处理函数中调用 NavigationService 的 AddBackEntry 方法，如代码 9-27 所示。

```

private void cmdAdd_Click(object sender, RoutedEventArgs e)
{
    if (lstSource.SelectedIndex != -1)
    {
        NavigationService nav = NavigationService.GetNavigationService(this);
        string itemText = lstSource.SelectedItem.ToString();
        string journalName = "Added " + itemText;

```

```

        nav.AddBackEntry(GetJournalEntry(journalName));

        lstTarget.Items.Add(itemText);
        lstSource.Items.Remove(itemText);
    }
}
private void cmdRemove_Click(object sender, RoutedEventArgs e)
{
    if (lstTarget.SelectedIndex != -1)
    {
        NavigationService nav = NavigationService.GetNavigationService(this);
        string itemText = lstTarget.SelectedItem.ToString();
        string journalName = "Removed " + itemText;

        nav.AddBackEntry(GetJournalEntry(journalName));

        lstSource.Items.Add(itemText);
        lstTarget.Items.Remove(itemText);
    }
}

```

代码 9-27 调用 NavigationService 的 AddBackEntry 方法

这样相当于在一个页面中实现了前进和后退功能，如图 9-29 所示，其中历史中已经记录添加和删除某种颜色的记录。



图 9-29 在页面中实现前进和后退功能

9.7 XAML 浏览器应用程序

如前所述，Page 的宿主窗口有 3 种，即浏览器、导航窗口和 Frame。我们将运行在浏览器中的导航程序称为“XAML 浏览器应用程序”。

运行 XBAP 程序的计算机必须满足如下条件。

- (1) 安装.NET Framework 3.0 或者以上版本（3.5 及 4.0）。
- (2) 操作系统中有 IE（6 或以上版本）或者火狐（FireFox）浏览器。

如果已经安装了 VS 2008，即意味着已经安装了 .Net Framework 3.5；如果安装了 Windows Vista 或者 Windows 7 操作系统，则已经内置了 .NetFramework 3.0。而 IE6 及其以上版本是 Windows 操作系统必备的组件。

好了，一切准备妥当。现在我们就使用乾坤大挪移的手法将登陆应用程序从导航窗口移植到浏览器当中。速度可能比你想象的还要快喔。

9.7.1 将一个基于窗口的导航程序变换成 XBAP 程序——乾坤大挪移

新建一个与原来登录应用程序名称相同的 XBAP 的应用程序，在新建项目时会有 WPF Browser Application 选项，如图 9-30 所示。

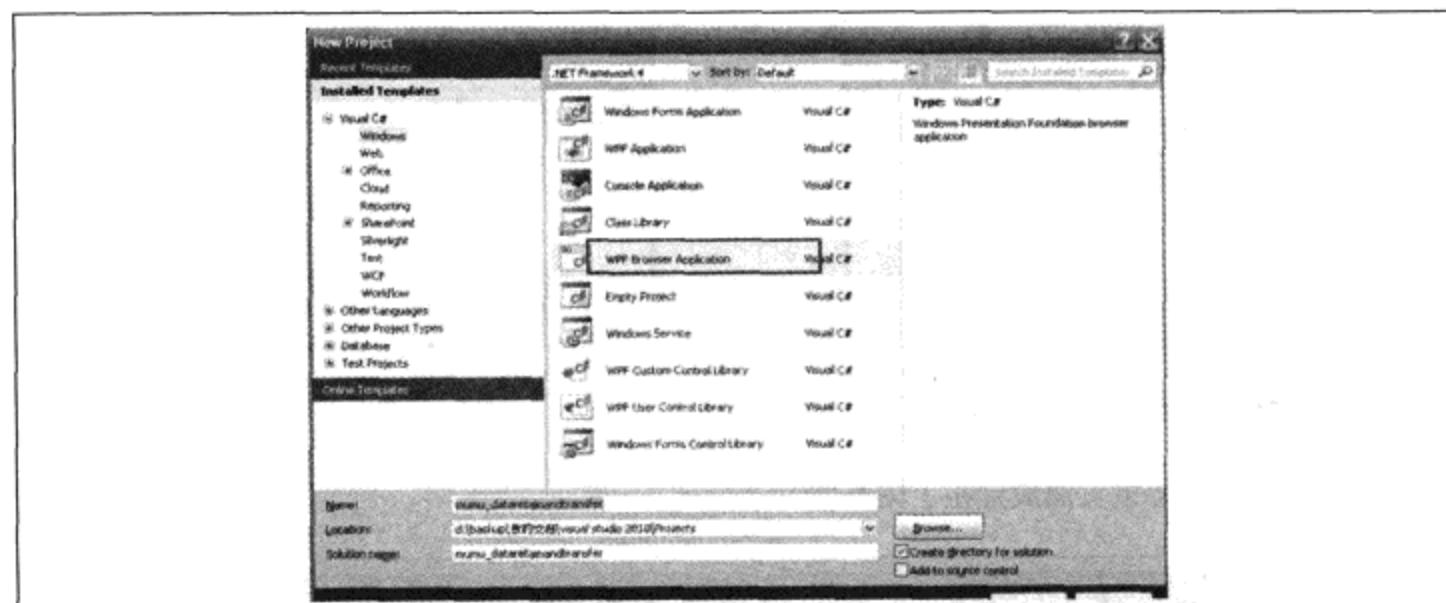


图 9-30 新建同名的 XBAP 应用程序

XBAP 程序和普通的 WPF 应用程序一样提供了一个默认的 App 类和一个 Page1 类（在普通的 WPF 应用程序中是 MainWindow 类）。除此以外，Visual Studio 还提供了一个自动生成的认证文件，后缀名为 “.pfx” 。我们可以删除 App 类和 Page1 类，只保留认证文件。

将原来的登录应用程序的页面文件和相应的代码文件全部添加到该工程中，所有文件无须修改，仅需略微修改 App 类。由于 Page 的宿主窗口已经不再是导航窗口，而是浏览器。因此在 App 的 Startup 事件处理函数中需要隐藏创建和显示代码导航窗口，只保留用户列表的初始化，如代码 9-28 所示。

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    users = new List<User>();
    User user = new User("木木", "mumu");
    user.FavColors.Add("红色");
    user.FavColors.Add("绿色");
    users.Add(user);
    //NavigationWindow win = new NavigationWindow();
    //win.Width = 480;
    //win.Height = 400;
    //win.Content = new LoginPage();
```

```
//win.Show();
}
```

代码 9-28 仅仅保留用户列表的初始化

此外需要将 App.xaml 文件中初始的页面设置为 LoginPage.xaml 文件，如代码 9-29 所示。

```
<Application x:Class="mumu_dataretainandtransfer.App"
.....StartupUri="LoginPage.xaml" Startup="Application_Startup">
```

代码 9-29 将 StartupUri 设置为 LoginPage.xaml 文件

注意在添加 App.xaml 文件时 VS 2010 有可能将该文件的 Build Action 选项修改为默认的 Page，需要重新将该选项修改为 ApplicationDefinition，如图 9-31 所示。

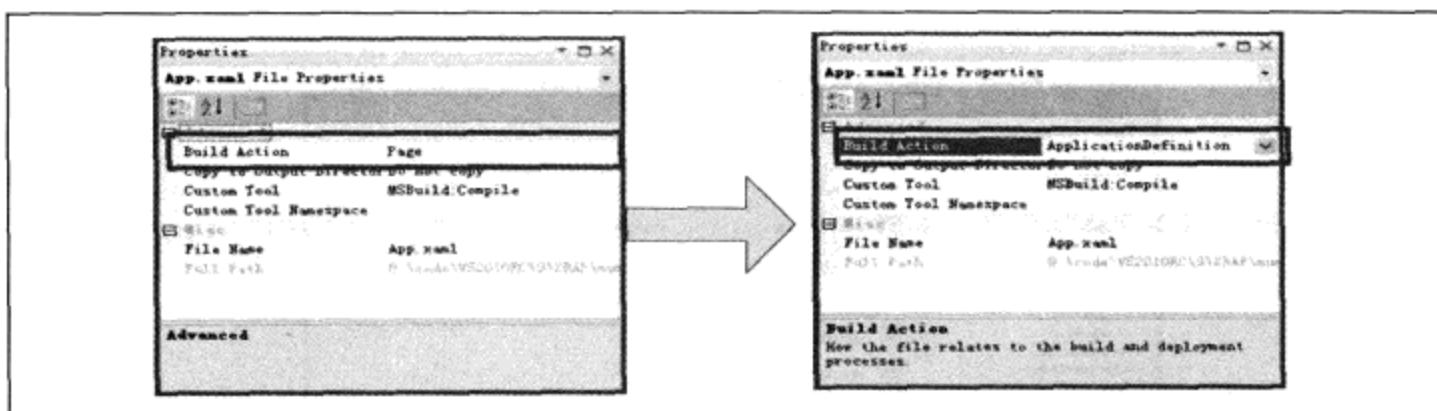


图 9-31 修改为 ApplicationDefinition

按 F5 键运行程序，登录页面放置在了浏览器中。



图 9-32 在浏览器中运行的导航程序

9.7.2 XAML 浏览器应用程序小结

不要因为 XBAP 程序运行在浏览器中，就将其和现在流行的 Silverlight 程序混淆，它是完全的 WPF 应用程序。从技术来看，所有类型的 WPF 应用程序也包括 XBAP 应用程序，都作为一个单独并由 CLR 管理的进程运行，XBAP 应用程序只不过是披上了浏览器的外壳而已，而 Silverlight 程序则直

接加载到浏览器进程中。不过 XBAP 程序与其他 WPF 程序有所不同，区别如下^[4]。

(1) 运行在浏览器窗口中并且有两种方式，一是上例中页面占据整个浏览器；二是使用<iframe>元素将其放置在普通的 HTML 文档的某处，如代码 9-30 所示。

```
一个普通的 HTML 页面
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
    <iframe src=mumu_dataretainandtransfer.xbap></iframe>
  </body>
</html>
```

代码 9-30 通过<iframe>标签将 xbap 文件放置在网页中

(2) 出于安全考虑，XBAP 程序的权限分为两种，即部分信任和完全信任。默认为前者，这意味着 XBAP 程序在一般情况下不能写文件并与其他计算机资源交互，如修改和访问注册表，以及连接数据库等。如果希望构建完全信任的 XBAP 程序，可以在项目属性页中的 Security 选项卡中修改权限，如图 9-33 所示。

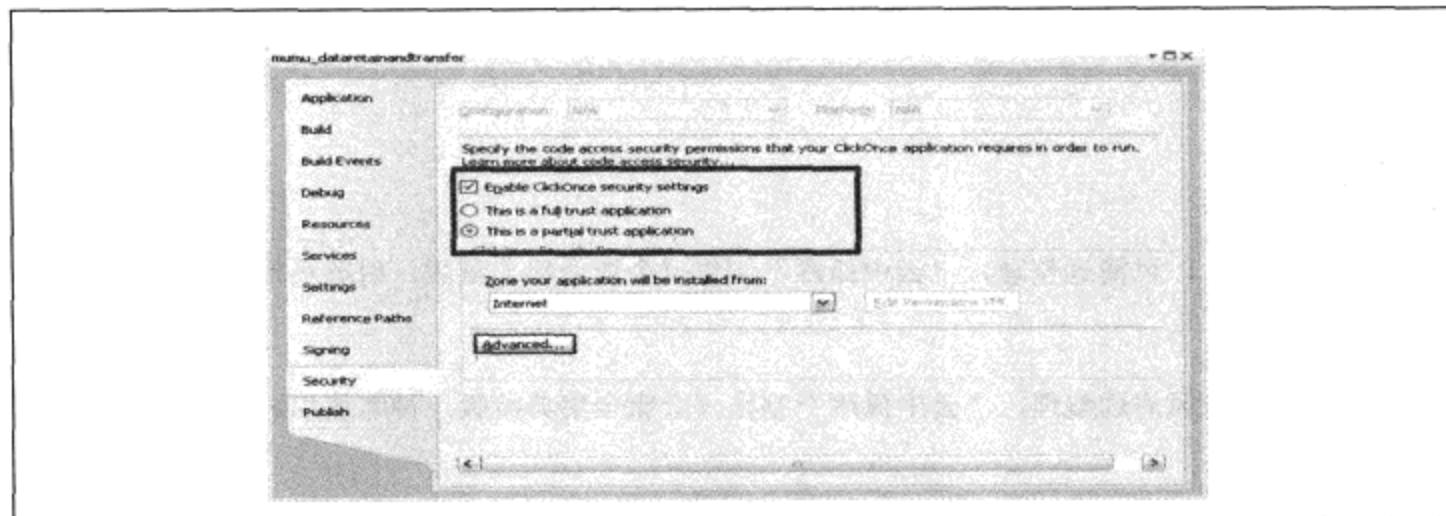


图 9-33 修改权限

(3) 不需要安装，运行 XBAP 应用程序时它会下载并缓存到浏览器中。如果浏览器发现有新的版本，会将其下载并缓存在浏览器中。

不过在调试 XBAP 程序时经常会出现奇怪的问题，即修改页面后无论 Rebuild 多少次，XBAP 总是停留在上一个尚未修改的版本，这是因为没有清空缓存所致。这时可以在 VS 2010 提供的命令窗口中键入命令 Mage.exe -cc 清空缓存，再次运行 XBAP 程序则是修改的版本，如图 9-34 所示。

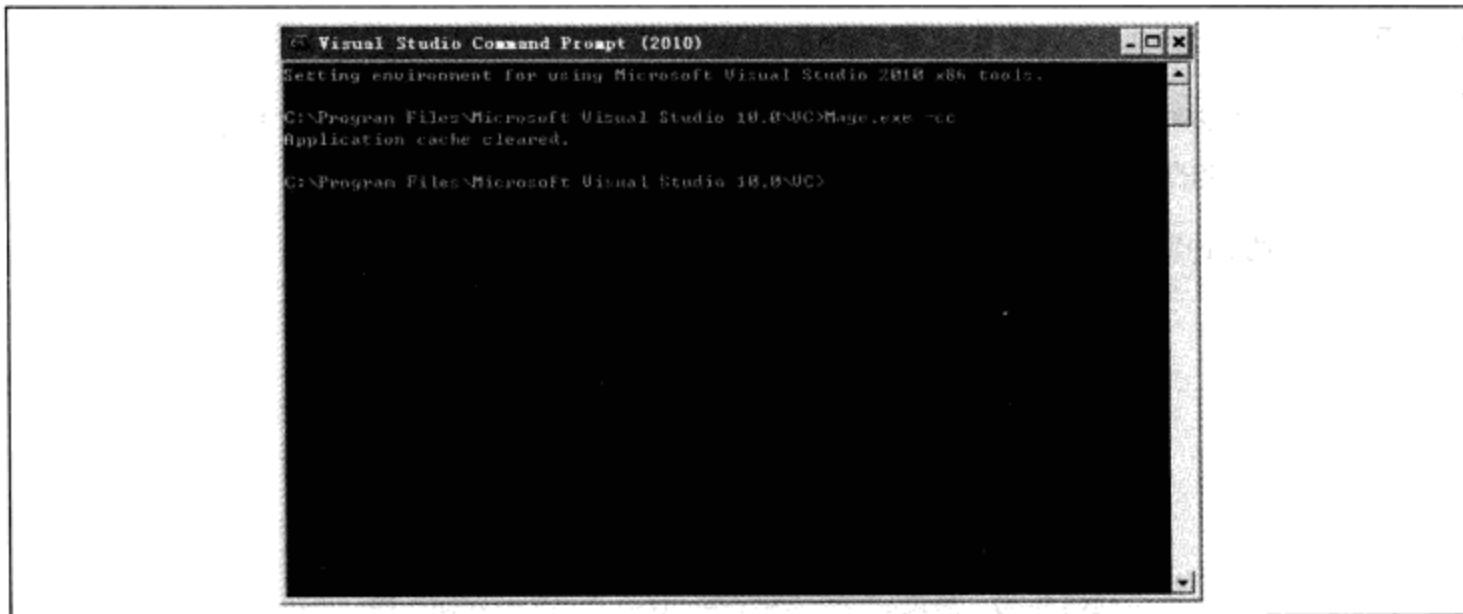


图 9-34 通过 Mage.exe 命令清空缓存

9.8 接下来做什么

讨论完了两类不同的窗口程序：标准窗口程序和基于导航的程序。接下来需要将各类不同的控件摆放在窗口当中，而这其中涉及到了 WPF 的一个重要特性就是布局。接下来，我们就进入下一章布局——药师的桃花岛。

参考文献

- [1] “北风之神”，风清远整理，“云中孤雁”制作《金庸全集典藏版，射雕英雄传》，“第二十五回 荒村野店”。
- [2] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版，神雕侠侣》，“第三回 求师终南”。
- [3] MSDN Library for Visual Studio 2008 SP1Navigation Overview.
- [4] Matthew MacDonald 著王德才译，《WPF 编程宝典——使用 C# 2008 和.NET 3.5》（第 2 版），清华大学出版社。

布局——药师的桃花岛

郭靖见她（黄蓉）在花丛中东一转西一晃，霎时不见了影踪。急忙追去，只奔出十余丈远，立时就迷失了方向。只见东南西北都有小径，却不知走向哪一处好。他走了一阵，似觉又回到了原地。想起在归云庄之时，黄蓉曾说那庄子布置虽奇，却哪及桃花岛阴阳开阖、乾坤倒置之妙。这一迷路，若是乱闯，定然具有越走越糟。于是坐在一株桃树之下，只待黄蓉来接。

——《射雕英雄传》：“第十六回 九阴真经”^[1]

这是郭靖初上桃花岛的一段，黄岛主按照五行生克和阴阳八卦的变化来布置桃花岛，因此常人无法近得桃花岛半步。

WPF 的布局也有这样的类似巧妙之处，不懂得其中奥秘是寸步难行。一旦悟透其间相生相克的道理，则如履平地。为了对布局有一个感性认识，我们为木木同学量身定做了一个现代版黄药师出题招婿故事，故事的名字叫做“布局巧设桃花岛，木木憨娶俏黄蓉”。

- (1) 憨木木误闯桃花宝岛。
- (2) 老顽童试解桃花玄机。
- (3) 黄岛主演绎布局精妙。
- (4) 接下来做什么。

10.1 憨木木误闯桃花宝岛

话说桃花岛软件公司药师有两件烦心事：一是公司技术集体转型，从 WinForm 集体转型到 WPF 急需 WPF 专业型人才；二是药师丧妻之后与女儿蓉儿相依为命，对她宠爱无比。眼看已经出落成十八九的大姑娘，但甚是娇纵，毫无规矩，希望找个青年才俊将其许配。

药师是个聪明之人，一摸胡须想何不将招聘技术人员和招婿合为一起，这样将来也好继承我桃花岛软件公司的大好基业。于是药师大笔一挥，写下了如此这般的招良才贤婿的广告：

小女蒲柳弱质，性又顽劣，原难侍奉君子。有道男大当婚，女大当嫁。老夫愿得一良才贤婿，继承家业，共享天伦之乐。

药师的招婿广告贴出之后，出现了一个奇特的景象。一时间桃花岛外熙熙攘攘，恍若闹市；桃花岛上寂然无声，门可罗雀。还是因为岛上奇门八卦之阵，青年才俊们都近不得岛半步。他们知道桃花

岛凶险，都不愿意打头阵，害怕错失良机。于是他们的目光集中在倚在墙角的那个人身上——木木。所有人都认为论长相和才智，再没有一个人比木木差了。因此不妨让木木一试，既不用担心木木会娶到黄蓉，也可以多了解桃花岛的情况。于是欧阳克公子不由木木分说，抓住他的领脖，就将他甩向了桃花岛。

木木醒来已是夜深，忽听到一阵箫声。似浅笑，似低诉，柔靡万端。木木不由痴了，打从娘胎出来，第一次听到如此这般的天籁之音。正自沉吟，忽听得前面发出一阵急促喘气之声，正是一人盘膝而坐。这时那洞箫声情致飘忽，缠绵宛转。便似一个女子一会儿叹息，一会儿呻吟，一会儿又软语温存，柔声叫唤。木木年纪尚小，对男女之事不甚了了。听到箫声时感应甚淡，听了也不以为意。但对面那人却是气喘愈急，听他呼吸声真是痛苦难当，正拼了全力来抵御箫声的诱惑。

这时木木有些害怕，为了给自己壮胆，他不由跟着箫声哼了一首周杰伦的“七里香”。木木那五音不全，还走调的歌曲似乎是箫声的劲敌，立刻打破了箫声的神秘气氛。眼看那人作势便待跃起，听到木木的《七里香》，心中一静，重新盘膝而坐，闭目运功。过了良久，月光从花树中照射下来。木木才看清那人面容，须发苍然，并未全白，原来此人正是传说中的老顽童——周伯通。

10.2 老顽童试解桃花玄机

周伯通微微笑了笑，说道：“你上岛是为招婿而来？”木木有些不好意思了，说到：“前辈，我是过来看看热闹的，但是没想到被人给丢了上来。”老顽童又仔细打量了木木一下，掩口而笑：“黄老邪怎么可能看上你，阔鼻大耳，身高不足1.7米。走吧，我带你去看看一些好玩意。”不由分说，又抓住木木的领脖，往桃花岛的最高峰奔去。

约摸一柱香的工夫，周伯通和木木登上了桃花岛的最高峰——首阳。周伯通在桃花岛独居已久，无聊之极。忽有一个人与他说话解闷，大感愉悦。他拍着木木的肩说：“你往下看，桃花岛的所有奥秘都在此？”木木往下看，只见桃花开得正艳，东一片，西一片。一阵晕眩，也看不出所以然。老顽童呵呵一笑：“在桃花岛上，实在是无聊，于是我天天坐在这儿看着一片一片桃花发呆。结果有天老天爷发脾气了，下了好大的暴雨。把我的桃花打得七零八落，我着实有些惆怅。这个时候有道彩虹徐徐而起，将那桃花林连成一片，突然间我终于明白了……”老顽童故作停顿，眼中发光：“一切皆因布局！”

老顽童从上衣口袋，拿出一支笔，又从裤袋里拿出了一张皱巴巴的纸铺开说：“你看那些一片一片桃花杂乱无章，而我看它们则非常规律。”木木大感兴趣，于是一老一小就蹲在那儿开始研究起来。

大多数 GUI 程序都有许多控件，这些控件如何放置，放置控件的容器大小改变时又要如何调整控件，这样的主题称为“布局”（layout）。黄老邪是做软件的，因此他的桃花岛布置完全合乎布局的这种思路。他将桃花岛分为 6 个区域，每个区域是一种布局，而每个区域的桃树林则可以看成是一个个控件。

木木听着老顽童的话，向第 1 个区域看去，只见几片红、浅红和白等不同颜色的桃树林东一片西一

片，没有丝毫异常。接着他又观察第 2 个区域，这里的桃树林是同样宽度，整整齐齐地沿纵向摆放。他的视线又转到了第 3 个区域，只见这个区域又划分成了若干个小区域。每个小区域还不一样，有的平行摆放了若干桃林，有的不行摆放不下，又换做第 2 行。木木头已经大了，不过还是接着看第 4 个区域。这个区域更是奇特，所有的桃林一层层由外到内不停嵌套。木木一阵晕眩，有点站立不稳，幸好老顽童站在旁边搀扶住了他。木木稍加休息，接着看第 5 个区域。这个区域倒并不是特别奇特，不同颜色的桃树林排成规则的格网。接着第 6 个区域，这个区域的桃树排成一个圈状，如图 10-1 所示。

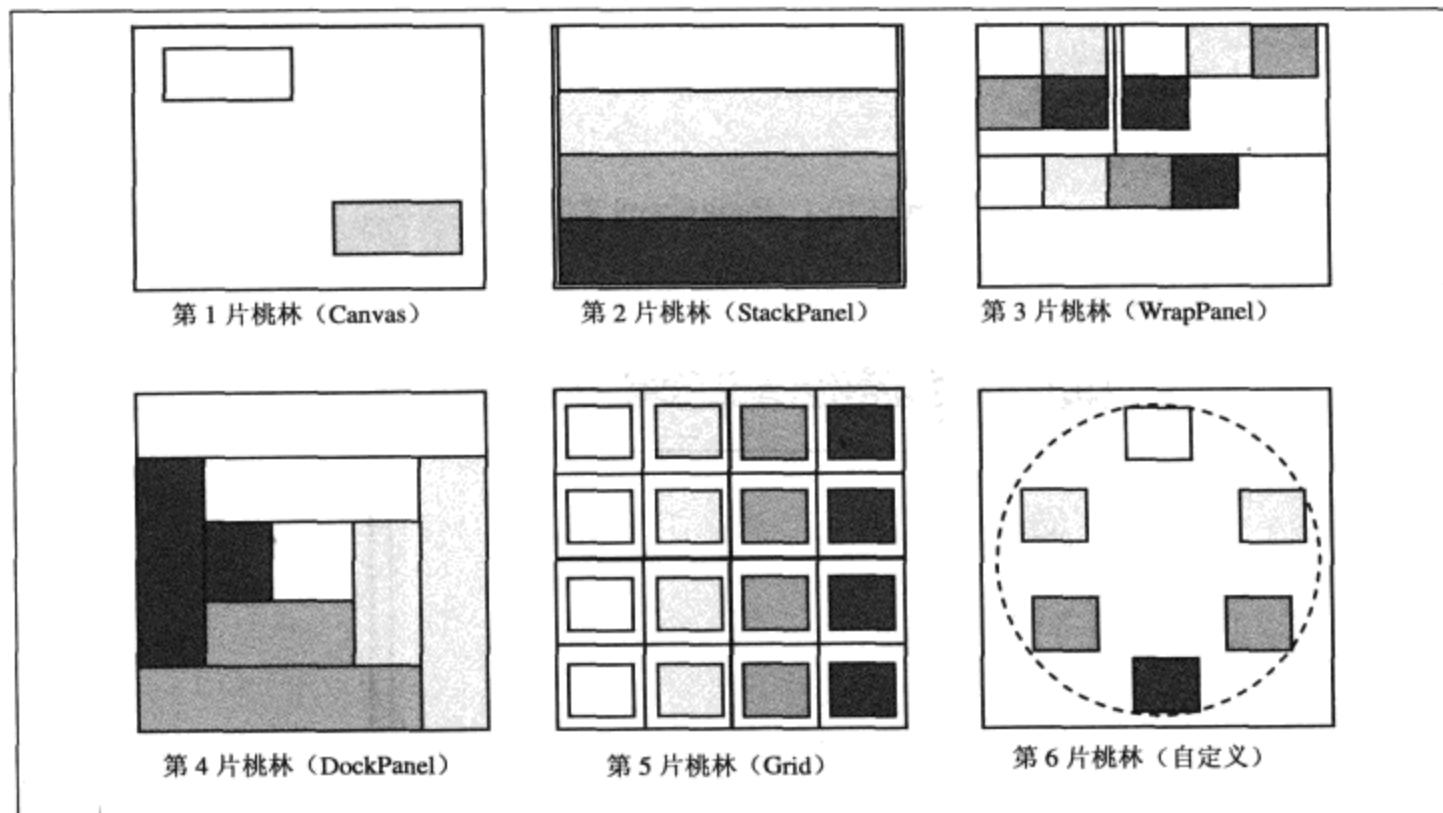


图 10-1 桃花岛 6 个区域的布局

老顽童嘻嘻一笑说到：“我初看这些桃林，也和你一般，看得时间久了也是非常晕眩。但是后来我看穿了黄老邪的秘密，再看这些桃林，发现特别规整。第 1 个是 Canvas 布局；第 2 个是 StackPanel 布局；第 3 个是 WrapPanel 布局；第 4 个你看起来比较晕眩的是 DockPanel 布局，实际上这种布局，一旦明白就会觉得非常简单；第 5 个和第 6 个布局看似平常，实际上未必简单。第 5 个是 Grid 布局，而第 6 个则是黄老邪自定义的一种布局。”

木木听老顽童这么一说，学习布局的兴趣骤然大增，于是打开笔记本开始逐个学习布局。

10.2.1 Canvas

Canvas 是基本面板，仅仅支持用与设备无关的坐标来定位元素。这是一种传统的布置用户界面的方式，在 Win32、MFC，甚至 WinForm 时期都是这样做的。

Canvas 用 4 个附加属性 Left、Top、Right 和 Bottom 来定位子元素，用代码 10-1 所示的代码可以模仿桃花岛上的第 1 个桃林区域（完整示例详见 `mumu_layout` 工程）。

```
<Page x:Class="mumu_layout.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="308" d:DesignWidth="294"
    Title="第一片桃林仿真 by 木木">
    <Border BorderThickness = "2" BorderBrush = "Black" Margin = "5">
        <Canvas>
            <Button Canvas.Left = "24" Canvas.Top = "50" Background="#00000000"
                Content = "Left=24,Top=50"/>
            <Button Canvas.Right="24" Canvas.Bottom="50" Background ="#FFFFCCFF"
                Content = "Right=24,Bottom=50"/>
        </Canvas>
    </Border>
</Page>
```

代码 10-1 Page1.xaml 文件

程序运行结果如图 10-2 所示。

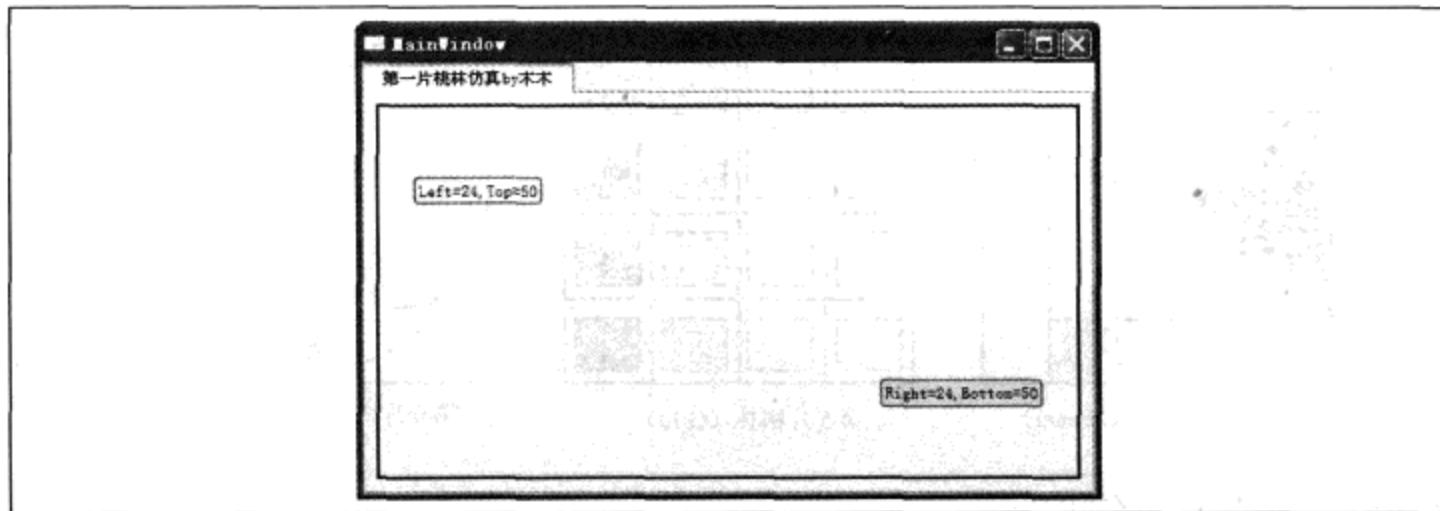


图 10-2 第一片桃林仿真

Canvas 只允许设置一个横向和纵向成对的坐标，如 Left 和 Top，以及 Right 和 Bottom 等。如果设置了 Left 和 Right 或者 Top 和 Bottom，则忽略一个值。

木木做完了这个仿真还是有些疑问，问到：“周大哥，这么原始的面板会有什么用呢？”老顽童看木木这么快做完一个例子，倒还真有些惊喜，说到：“你比我以前的一个结拜师弟要聪明得多。这种面板由于简单，自然效率就高，用在矢量绘图上是再合适不过了。”木木听了若有所思，紧接着开始学习第二种面板……

10.2.2 StackPanel

StackPanel 是一种非常受欢迎的面板，用于顺序垂直或者水平的排列子元素。它通过 Orientation 属性来控制水平 (Horizontal) 和垂直 (Vertical) 排列，默认值是纵向。模仿桃花岛的第 2 个区域的代码如代码 10-2 所示。

```
<Page x:Class="mumu_layout.Page2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    mc:Ignorable="d"
    d:DesignHeight="308" d:DesignWidth="251"
    Title="第二片桃林仿真 by 木木">
    <Border BorderThickness = "2" BorderBrush = "Black" Margin = "5">
        <StackPanel>
            <Button Background="#00000000" MinHeight="50" MinWidth="50"
Content = "1"/>
            <Button Background ="#FFFFCCFF" MinHeight="50" MinWidth="50"
Content = "2"/>
            <Button Background ="#FFFF9BFF" MinHeight="50" MinWidth="50"
Content = "3"/>
            <Button Background ="#FFFF00FF" MinHeight="50" MinWidth="50"
Content = "4"/>
        </StackPanel>
    </Border>
</Page>
```

代码 10-2 Page2.xaml 文件

程序运行结果如图 10-3 所示。

修改 Orientation 属性，桃林就可以由垂直排列变为水平排列，如代码 10-3 所示。

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

代码 10-3 修改 Orientation 属性为 Horizontal

程序运行结果如图 10-4 所示。

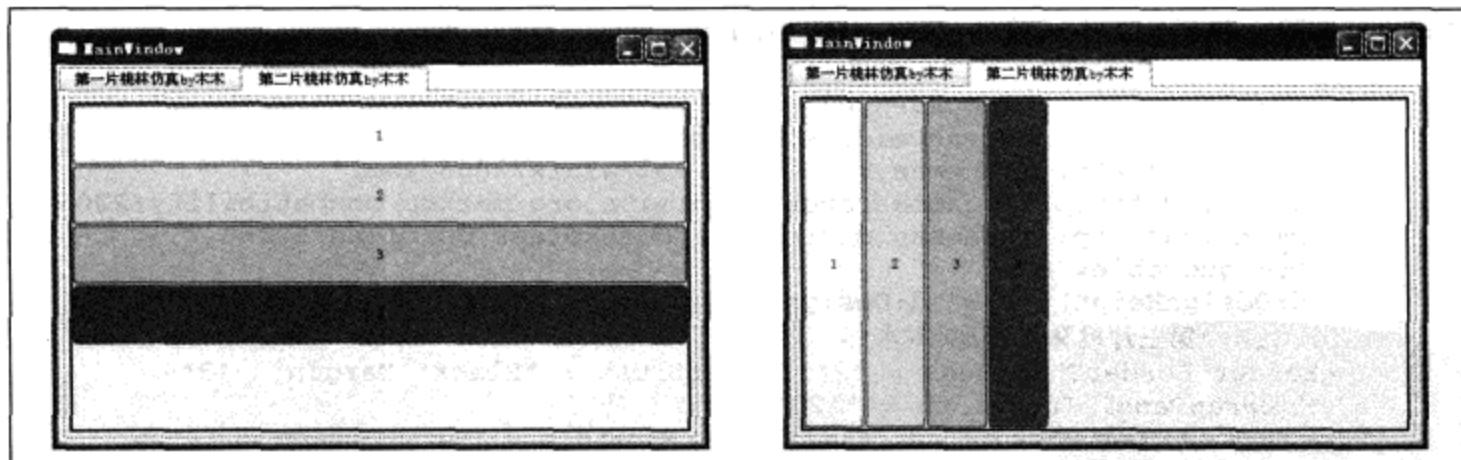


图 10-3 第 2 片桃林仿真

图 10-4 第 2 片桃林变换之一：Orientation = "Horizontal"

修改 StackPanel 的一处属性（FlowDirection）如下：

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

程序运行结果如图 10-5 所示。

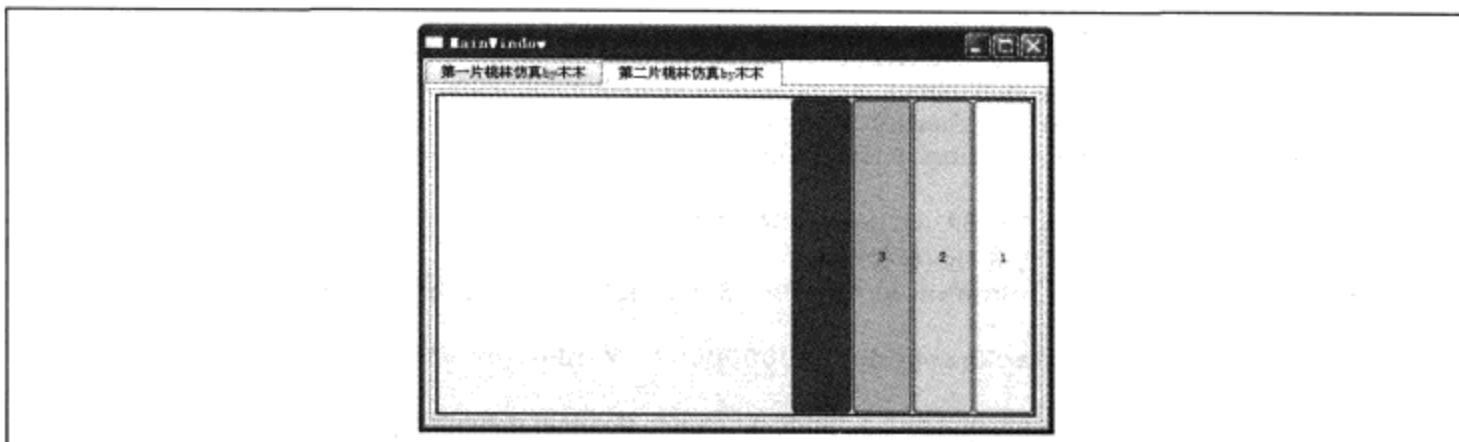


图 10-5 第2片桃林变换之二：Orientation = "Horizontal" FlowDirection = "RightToLeft"

10.2.3 WrapPanel

WrapPanel 与 StackPanel 类似，与 StackPanel 不同的是除了会自动垂直和水平排列子元素以外，当没有空间放置子元素时会自动将其放置在下一行或者下一列中，它特别适用于元素个数不确定的情况。

WrapPanel 有 3 个控制其行为的属性，如表 10-1 所示。

表 10-1 WrapPanel 的 3 个属性

属性	描述
Orientation	类似 StackPanel 的 Orientation，默认为水平排列
ItemHeight	允许子元素的最大高度，任何比 ItemHeight 高的子元素都将被截断
ItemWidth	允许子元素的最大宽度，任何比 ItemWidth 宽的子元素都将被截断

使用 WrapPanel 模拟第 3 个桃林的代码如代码 10-4 所示。

```
<Page x:Class="mumu_layout.Page3"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="233" d:DesignWidth="251"
    Title="第三片桃林仿真 by 木木">
    <Border BorderThickness = "2" BorderBrush = "Black" Margin = "5">
        <WrapPanel ItemWidth = "120">
            <Button Background="#00000000" MinWidth = "50" Content = "1"/>
            <Button Background ="#FFFFCCFF" MinWidth = "50" Content = "2"/>
            <Button Background ="#FFFF9BFF" MinWidth = "50" Content = "3"/>
            <Button Background ="#FFFF00FF" MinWidth = "150" Content = "4 MinWidth
= 150"/>
        </WrapPanel>
    </Border>
</Page>
```

代码 10-4 Page3.xaml 文件

程序运行结果如图 10-6 所示，第 4 个按钮由于长度超出 WrapPanel 指定的范围，因此被截断。而且随着窗口的大小改变，子元素会自动地变换其位置。

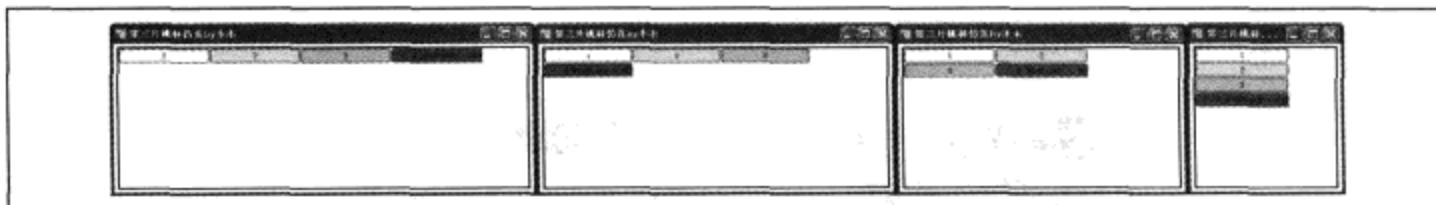


图 10-6 水平排列的按钮随着窗口的宽度变小重新排列

将其改为垂直排列，运行结果如图 10-7 所示。

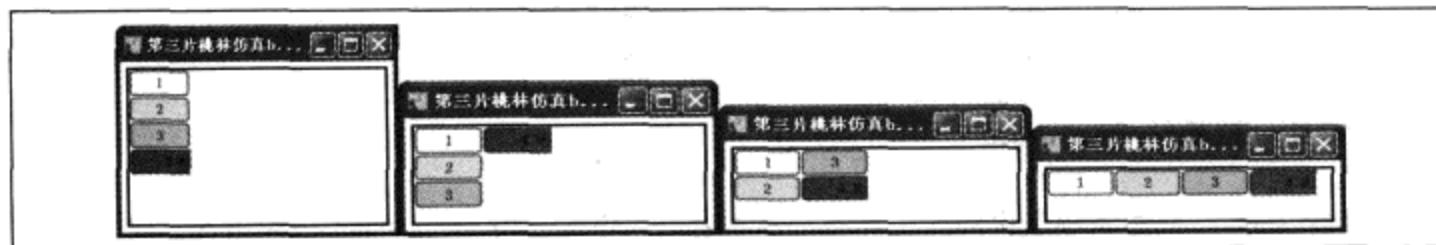


图 10-7 垂直排列的按钮随着窗口的高度变小重新排列

10.2.4 DockPanel

DockPanel 可以使子元素停靠在面板的某一条边上，然后拉伸元素以填满全部宽度或高度。它有一个 Dock 附加属性，子元素用 4 个值来控制其停靠，即 Left（默认）、Top、Right 和 Bottom。注意 Dock 并没有 Fill 值，默认情况下最后一个添加到 DockPanel 的子元素将填满所有剩余的空间；除非 DockPanel 的 LastChildFill 属性设置为 false。

模拟第 4 片桃林的代码如代码 10-5 所示。

```
<Page x:Class="mumu_layout.Page4"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300"
    Title="第四片桃林仿真 by 木木">
    <Border BorderThickness = "2" BorderBrush = "Black" Margin = "5">
        <!--两种不同属性值的设置，LastChildFill 默认为 true-->
        <!--<DockPanelLastChildFill = "false">-->
        <DockPanel>
            <Button Background="#00000000" DockPanel.Dock = "Left" Content = "1"/>
            <Button Background ="#FFFFCCFF" DockPanel.Dock = "Top" Content = "2"/>
            <Button Background ="#FFFF9BFF" DockPanel.Dock = "Right" Content = "3"/>
            <Button Background ="#FFFF00FF" DockPanel.Dock = "Bottom" Content = "4 "/>
            <Button Background="#00000000" DockPanel.Dock = "Left" Content = "5"/>
            <Button Background ="#FFFFCCFF" DockPanel.Dock = "Top" Content = "6"/>
            <Button Background ="#FFFF9BFF" DockPanel.Dock = "Right" Content = "7"/>
            <Button Background ="#FFFF00FF" DockPanel.Dock = "Bottom" Content = "8 "/>
        </DockPanel>
    </Border>
</Page>
```

代码 10-5 Page4.xaml 文件

程序运行结果如图 10-8 所示。

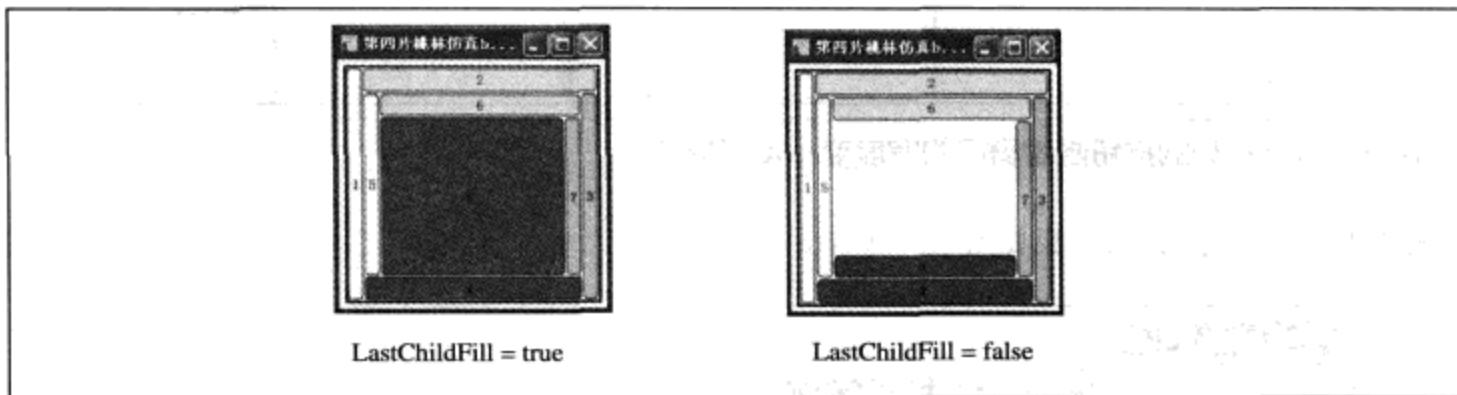


图 10-8 第 4 片桃林 DockPanel

写完这些代码后，突然老顽童说到：“且慢，木木老弟，你可以用 DockPanel 模拟出 StackPanel 的效果么？”

木木想了想说：“这有何难！”于是写下了如代码 10-6 所示。

```
<Page x:Class="mumu_layout.Page5"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="281" d:DesignWidth="267"
    Title="第四片桃树林 by 木木 DockPanel 模拟 StackPanel">
    <Border BorderThickness = "2" BorderBrush = "Black" Margin = "5">
        <DockPanel>
            <Button Background="#00000000" DockPanel.Dock = "Top" Content = "1"/>
            <Button Background ="#FFFFCCFF" DockPanel.Dock = "Top" Content = "2"/>
            <Button Background ="#FFFF9BFF" DockPanel.Dock = "Top" Content = "3"/>
            <Button Background ="#FFFF00FF" DockPanel.Dock = "Top" Content = "4 "/>
        </DockPanel>
    </Border>
</Page>
```

代码 10-6 Page5.xaml 文件

木木运行程序看了之后，虽然大致和自己想的差不多，但是最后一个按钮的面积过大。正在木木思考如何修改时，老顽童抢过笔记本，在 DockPanel 标签中加上了“LastChildFill = false”。一切变得完美了，如图 10-8 所示。两人相视一笑。

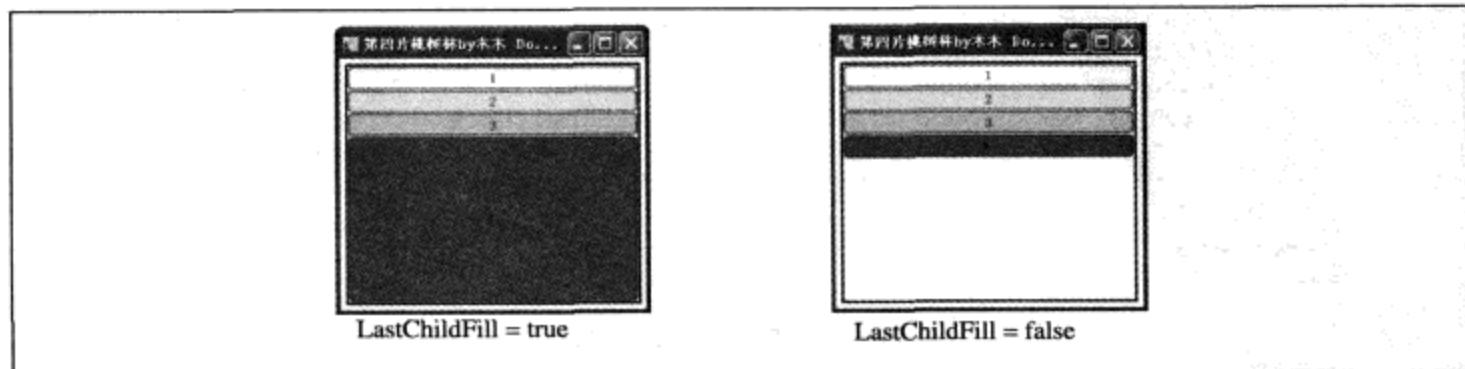


图 10-9 LastChildFill 的不同设置的效果

10.2.5 Grid

Grid 是 WPF 中常用且功能最为强大的布局，允许在一个多行多列的表中排列子元素。它有 4 个常用附加属性，如表 10-2 所示。

表 10-2 Grid 的 4 个常用附加属性

属性	说明
Column	标识子元素所在的列（以 0 为第 1 列）
Row	标识子元素所在的行（以 0 为第 1 行）
ColumnSpan	标识子元素占据的列数
RowSpan	标识子元素占据的行数

Grid 的行和列尺寸有如下 3 种单位。

(1) 绝对尺寸 (Absolute Sizing)：设置 Height 或 Width 为一个设备无关的值，当 Grid 的尺寸改变时绝对尺寸的行和列不会改变。

(2) 自动尺寸 (Autosizing)：设置 Height 或 Width 为 Auto，对于一行，这是最高元素的高度；对于一列，这是最宽元素的宽度。

(3) 比例尺寸 (Proportional sizing) 或者称为“星号尺寸” (Star sizing)：设置 Height 或 Width 为一种*号的特殊语法，可以使行和列按比例来分配可用的区域，一个采用比例尺寸的行和列会随着 Grid 的尺寸的改变而改变。

木木不太理解这种星号语法 (Star Syntax)，老顽童顺手拿过笔记本，写下了如代码 10-7 所示的例子：

```
<Page x:Class="mumu_layout.Page6"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    mc:Ignorable="d"
    d:DesignHeight="564" d:DesignWidth="406"
    Title="星号尺寸 Demoby 老顽童">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid Grid.Row = "0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width = "100"/>
            <ColumnDefinition Width = "*" />
        </Grid.ColumnDefinitions>
        <Button Background="#00000000" Grid.Column = "0" Content = "Width = 100"/>
        <Button Background ="#FFFFCCFF" Grid.Column = "1" Content = "Width = *"/>
    </Grid>
```

```

<Grid Grid.Row = "1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width = "100"/>
        <ColumnDefinition Width = "*" />
        <ColumnDefinition Width = "*" />
        <ColumnDefinition Width = "*" />
    </Grid.ColumnDefinitions>
    <Button Background="#00000000" Grid.Column = "0" Content = "Width = 100"/>
    <Button Background ="#FFFFCCFF" Grid.Column = "1" Content = "Width = *"/>
    <Button Background ="#FFFF9BFF" Grid.Column = "2" Content = "Width = *"/>
    <Button Background ="#FFFF00FF" Grid.Column = "3" Content = "Width = *"/>
</Grid>
<Grid Grid.Row = "2">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width = "100"/>
        <ColumnDefinition Width = "*" />
        <ColumnDefinition Width = "2*" />
        <ColumnDefinition Width = "*" />
    </Grid.ColumnDefinitions>
    <Button Background="#00000000" Grid.Column = "0" Content = "Width = 100"/>
    <Button Background ="#FFFFCCFF" Grid.Column = "1" Content = "Width = *"/>
    <Button Background ="#FFFF9BFF" Grid.Column = "2" Content = "Width = 2*"/>
    <Button Background ="#FFFF00FF" Grid.Column = "3" Content = "Width = *"/>
</Grid>
<Grid Grid.Row = "3">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width = "100"/>
        <ColumnDefinition Width = "*" />
        <ColumnDefinition Width = "2*" />
        <ColumnDefinition Width = "*" />
    </Grid.ColumnDefinitions>
    <Button Background="#00000000" Grid.Column = "0" Content = "Width = 100"/>
    <Button Background ="#FFFFCCFF" Grid.Column = "1" Content = "Width = 3*"/>
    <Button Background ="#FFFF9BFF" Grid.Column = "2" Content = "Width = 6*"/>
    <Button Background ="#FFFF00FF" Grid.Column = "3" Content = "Width = 3*"/>
</Grid>
</Grid>
</Page>

```

代码 10-7 Page6.xaml 文件

这是一个嵌套 panel 的示例，一个 4 行的 Grid 中每一行嵌套一个 Grid，如图 10-10 所示。

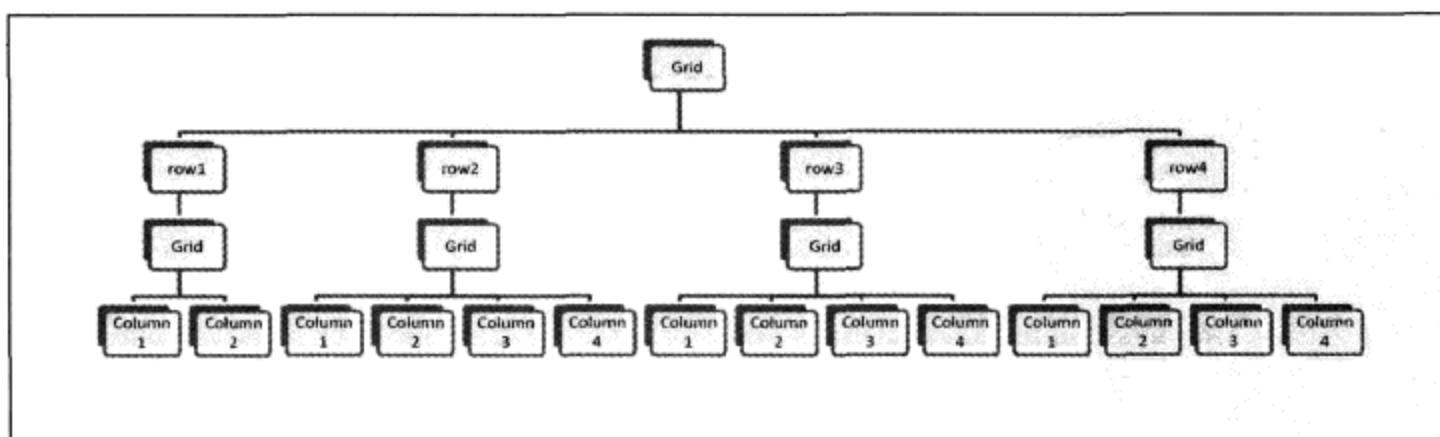


图 10-10 一个嵌套 panel 的例子

程序运行结果如图 10-11 所示。

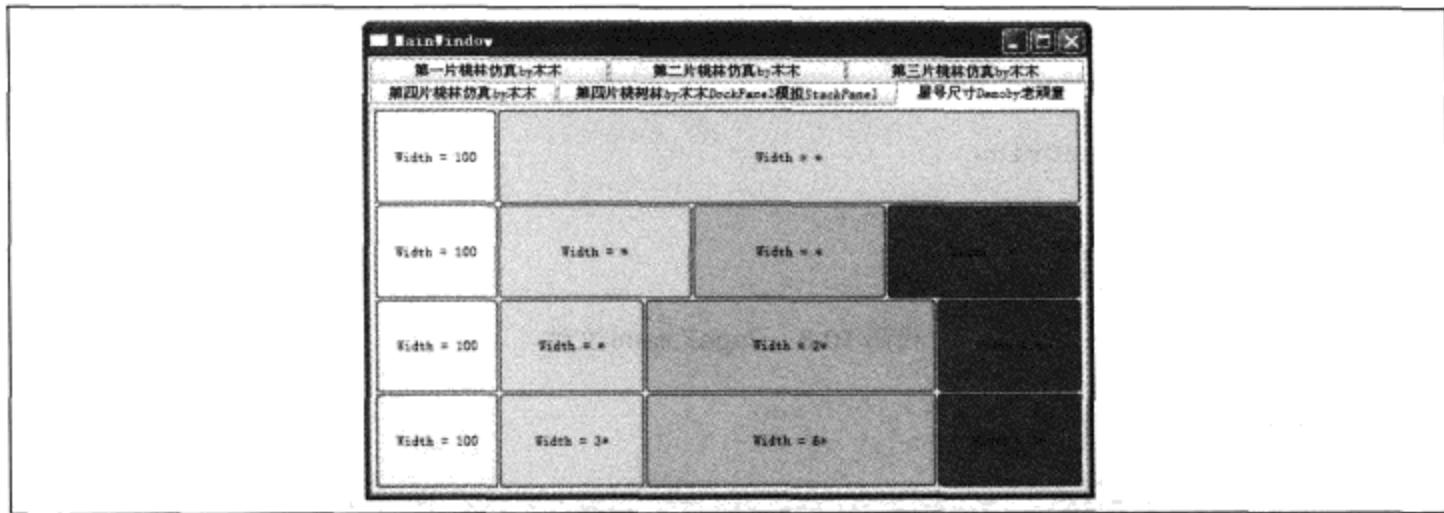


图 10-11 程序运行结果

这种语法基本上有这样的特点，老顽童在纸上写到：

- (1) 当一个行的高度或列的宽度设置为*时会占据所有的剩余空间。
- (2) 当多行或者多列使用*时剩余的空间会被等量地分配给这些行或列。
- (3) 行或列可以在*之前放一个系数，如 2*和 6*，表示按比例比其他行或列占据更多的空间。在同一个 Grid 中一个宽度为 2*的列的宽度是一个宽度为*（1*的缩写）列的两倍。

“而且还有一个好玩的东西”，老顽童说着，又十指如飞，把程序代码做了如下修改“通过将 Grid 的属性 `IsSharedSizeScope` 设置成 `true`（代码①），然后将行和列的属性 `SharedSizeGroup` 设置为一个大小写敏感的字符串，表示这个组的名称（代码②和③）。不仅可以将一个 Grid 里面的行和列设置成同样的高度或宽度，甚至是不同 Grid 的行和列也可以设置成同样的高度或者宽度。”老顽童说着说着，都手舞足蹈起来，不过木木确实没觉得这个属性有什么好玩的，……如下为代码 10-8。

```
<Page
    .....
    Title="共享尺寸 Demoby 老顽童">
①   <Grid Grid.IsSharedSizeScope = "true">

    .....
    <Grid Grid.Row = "0">
        <Grid.ColumnDefinitions>
            ②           <ColumnDefinition Width = "Auto" SharedSizeGroup = "a"/>
            <ColumnDefinition Width = "Auto" SharedSizeGroup = "a"/>
        </Grid.ColumnDefinitions>
    .....
    </Grid>
    .....
    <Grid Grid.Row = "2">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width = "100"/>
            <ColumnDefinition Width = "*" />
            ③           <ColumnDefinition Width = "2*" SharedSizeGroup = "a"/>
```

```

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

```

代码 10-8 Page7.xaml 文件

程序运行结果如图 10-12 所示。

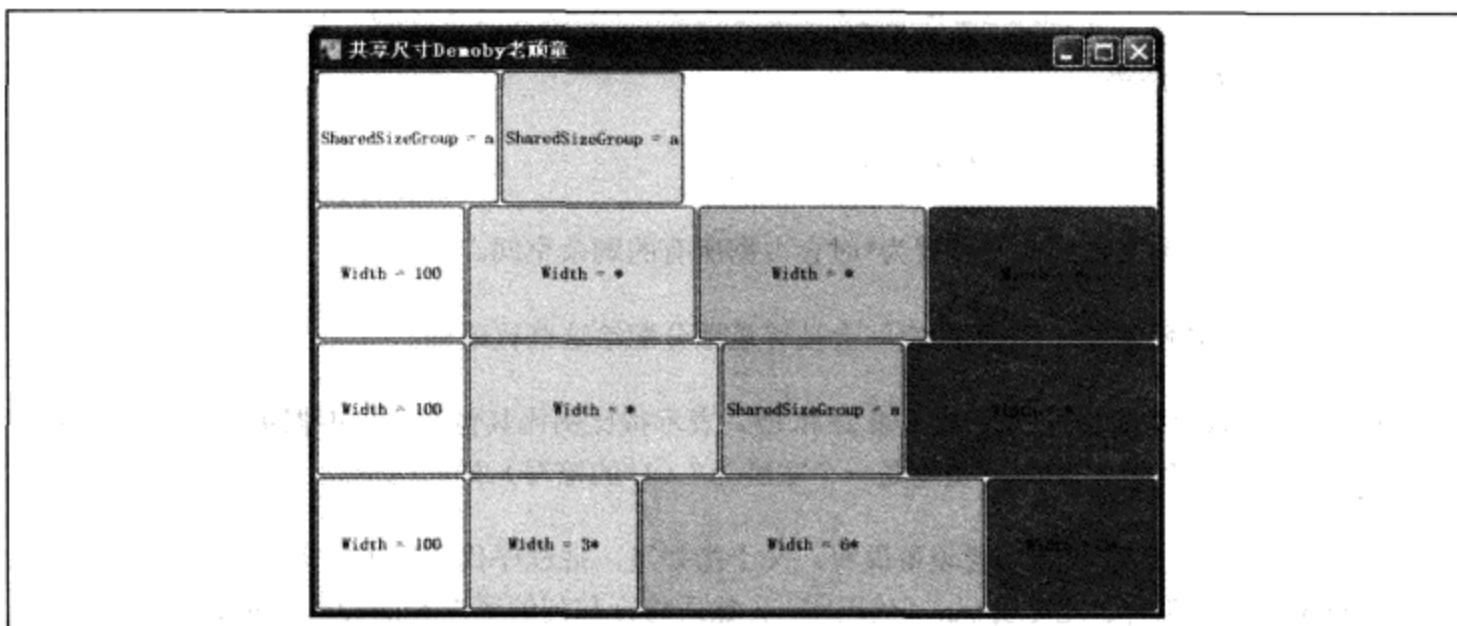


图 10-12 共享尺寸属性使 SharedSizeGroup 名称一样的行和列保持同一高度或者宽度

“还有一个特别好玩的称之为 GridSplitter，你有了它就可以用鼠标和键盘来改变尺寸。比如……”老顽童说着，又噼哩叭啦敲了一通，“成了，就是这样……”如代码 10-9 所示。

```

<Page x:Class="mumu_layout.Page8"
      .....
      Title="GridSplitterDemoby 老顽童">
    <Grid Grid.IsSharedSizeScope = "true">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        ① <GridSplitter Grid.Row="2" HorizontalAlignment="Stretch"
                      VerticalAlignment="Top" Background="Black"
                      ShowsPreview="true" ResizeDirection="Rows"
                      Height="5"/>
        <Grid Grid.Row = "0">
            .....
        </Grid>
        <Grid Grid.Row = "1">
            .....
        </Grid>
        <Grid Grid.Row = "2">

```

```

<Grid.ColumnDefinitions>
    <ColumnDefinition Width = "100"/>
    <ColumnDefinition Width = "*" />
    <ColumnDefinition Width = "2*" SharedSizeGroup = "a"/>
    <ColumnDefinition Width = "*" />
</Grid.ColumnDefinitions>
②   <GridSplitter Grid.Column="2" HorizontalAlignment="Left"
                VerticalAlignment="Stretch" Background="Black"
                ShowsPreview="true" Width="10"/>
③   <Button Background="#00000000" Grid.Column = "0" Content
        = "Width = 100" Margin="5"/>
        <Button Background="#FFFFCCFF" Grid.Column = "1" Content
        = "Width = *" Margin="5"/>
        <Button Background="#FFFF9BFF" Grid.Column = "2" Content
        = "SharedSizeGroup = a" Margin="5"/>
        <Button Background="#FFFF00FF" Grid.Column = "3" Content
        = "Width = ** Margin=5"/>
    </Grid>
    <Grid Grid.Row = "3">
        .....
    </Grid>
</Grid>
</Page>

```

代码 10-9 GridSplitter 的实现

老顽童添加了两个 GridSplitter，其中一个是第 3 行（代码①处）。它位于第 3 行的顶部，这是由 VerticalAlignment 属性所决定的；另外该 GridSplitter 的 HorizontalAlignment 属性为 Stretch，因此该 GridSplitter 会横跨所有的列。ResizeDirection 用来设置该 GridSplitter 用来调整行的还是列的一个属性，在这里用来调整行的。另外一个 GridSplitter 是在第 2 行的 Grid 中添加的（代码②）。老顽童为了能够将两个 GridSplitter 都显现出来，有意将几个按钮的 Margin 属性设置为 5，以便为 GridSplitter 留出显示的空间（代码③）。

老顽童写完之后，很快运行程序，然后用鼠标不断地拖拉着两个 GridSplitter，表情很沉醉。“真不知道怎么能乐成这样”木木小声嘀咕着。

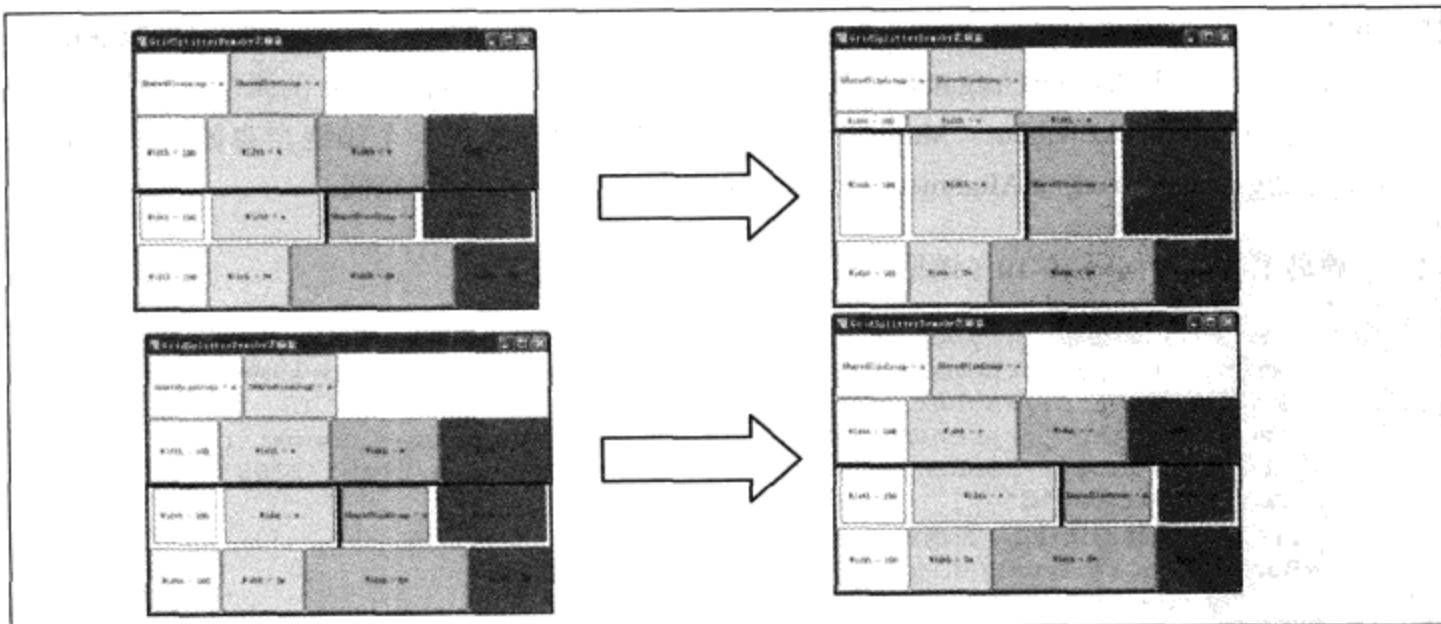


图 10-13 利用 GridSplitter 调整列宽和行高

“其实改变 GridSplitter 的属性 ResizedBehavior 之后,还可以这样玩”老顽童正欲往下说,突然一阵箫声又响起。箫声由远及近,不到半柱香的功夫,一个青衣怪客就近得前来……

10.3 黄岛主演绎布局精妙

那青衣怪客左手拿箫,右手慢慢从脸上揭下一层人皮面具。但见他形相清瘦,丰姿隽爽,萧疏轩举,湛然若神,正是桃花岛主黄药师。老顽童嘻嘻一笑,说道:“黄老邪,你用碧海潮生曲降我不住,倒被这傻小子轻松破解。”黄药师不由把木木仔细打量了一番,只见此人阔鼻大耳。还冒着三分傻气,实在不像身怀绝技之人。

黄药师袍袖一翻,木木的笔记本顷刻间就到他手上。药师一瞥,不由大惊,内心暗想:“好小子,原来你们正在研究我桃花岛布局的秘密。”心中不禁又有了几分欢喜之意,但是仍不动声色,问到:“傻小子,你上我桃花岛来干什么?”木木顿时满脸通红,老顽童说:“黄老邪,你是真傻还是假傻,当然是来上门娶你的宝贝丫头啊!”

说话间,黄药师已经看过老顽童和木木这几天游戏玩乐的代码。黄药师叹了一口气说:“伯通,你果然是个编程奇才,居然悟到了我桃花岛布置最精妙之处。”老顽童乐得手舞足蹈起来:“哈哈,老邪服输了吧。”药师眉头微微一皱,淡淡笑道:“不过你忽视了研究桃林本身。”老顽童一惊,随即又喜笑颜开,说:“老邪,赶紧讲讲,要不我拜你为师了。”话没说完咚咚 3 个响头……

10.3.1 桃树林的属性

药师其实也是喜好编程之人,看周伯通发自真心地佩服,说:“伯通不敢当,我愿意共同讨论讨论。说的不对,请你多指教。”3 人走到首阳峰一亭上,岛上下起小雨,飘飘洒洒,甚是舒服,悉听药师演绎布局。

桃花岛上的布局实际上是一个槽(slot)模型,其中每个区域(父对象)分配给桃林(子对象)一个槽。桃林能自由占用这个槽中空间的任何部分,该功能通过桃树林的 3 个属性,即 Margin、HorizontalAlign 和 VerticalAlign 来实现。它们都是 FrameworkElement 的属性,而大多数控件都要继承 FrameworkElement。Margin 允许子控件在槽内部获得一个围绕自身的缓冲空间,HorizontalAlign 和 VerticalAlign 决定子控件如何占用槽中的保留空间。

说着,黄岛主写下了代码 10-10(详见工程 mumu_layout2)。

```
<Page x:Class="mumu_layout2.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      Title="桃林的属性 by 黄药师" >
<Border Background="LightBlue"
        BorderBrush="Black"
        BorderThickness="2"
        CornerRadius="45"
```

```

        Padding="25">
    <StackPanel Name="StackPanel1" Background = "White">
        <TextBlock FontSize="18" HorizontalAlignment="Center"
Margin="0,0,0,15" Text = "StackPanel1">
        </TextBlock>
        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button Margin="5,10,15,20" >Normal</Button>
        </Border>
        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button Margin="5,10,15,20" HorizontalAlignment = "Left">Left</Button>
        </Border>
        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button HorizontalAlignment = "Right">
                Margin="5,10,15,20">Right</Button>
            </Border>
            <Border BorderThickness = "1" BorderBrush = "black" >
                <Button Margin="5,10,15,20" HorizontalAlignment =
"Center">Center</Button>
            </Border>
            <TextBlock>Button.Margin="5,10,15,20"</TextBlock>
        </StackPanel>
    </Border>
</Page>

```

代码 10-10 Page1.xaml 文件

运行该程序，结果如图 10-14 所示。

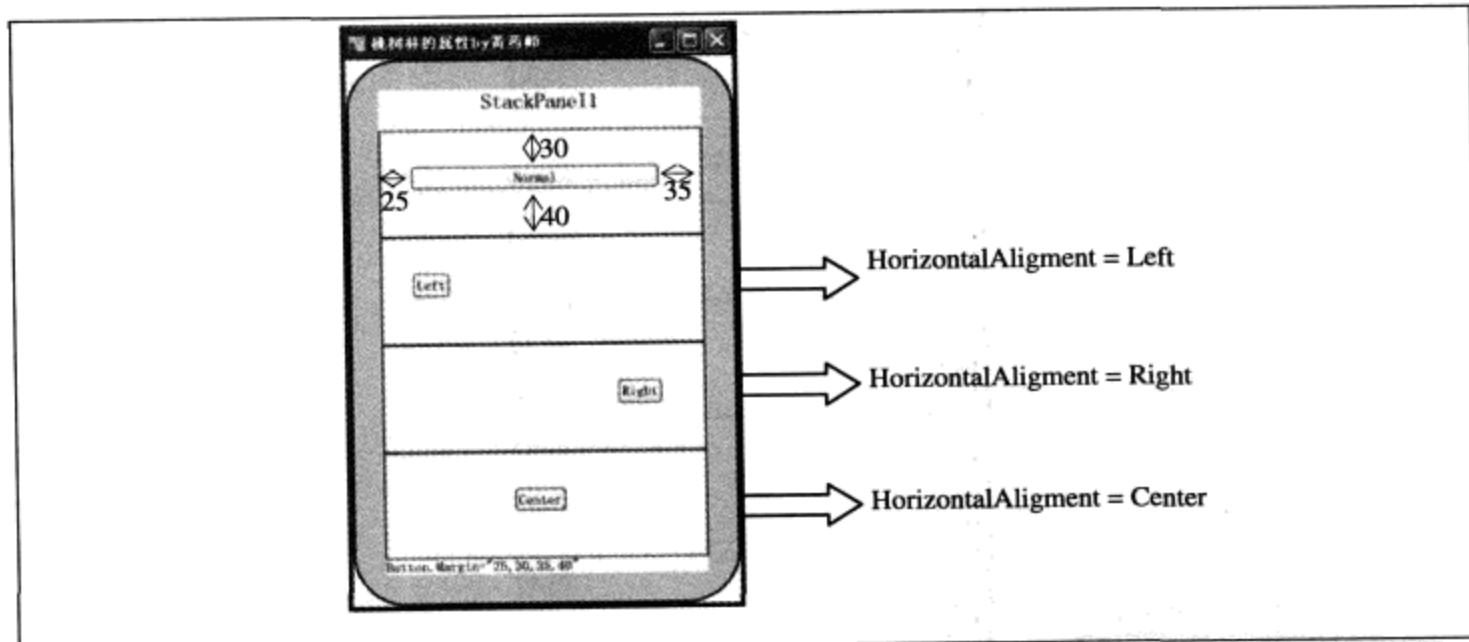


图 10-14 运行结果

老顽童笑了笑说：“老邪啊，只是我的注意力都集中在各种布局上了，忽视了控件本身的属性，这个算不上精妙啊。”黄药师没有理会，又添加了几行代码。如代码 10-11 所示（代码①、②和③处）。

```

        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button Margin="5,10,15,20" HorizontalAlignment =
"Left" Content="Left">
                ①           <Button.LayoutTransform>
                    <RotateTransform Angle = "15" />
                </Button.LayoutTransform>
            </Button>

```

```

        </Border>
        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button HorizontalAlignment = "Right"
Margin="5,10,15,20" Content="Right">
                <Button.LayoutTransform>
                    <RotateTransform Angle = "45" />
                </Button.LayoutTransform>
            </Button>
        </Border>
        <Border BorderThickness = "1" BorderBrush = "black" >
            <Button Margin="5,10,15,20" HorizontalAlignment =
"Center" Content="Center">
                <Button.LayoutTransform>
                    <RotateTransform Angle = "75" />
                </Button.LayoutTransform>
            </Button>

```

② ③

代码 10-11 给按钮添加上 LayoutTransform 属性 (Page2.xaml 文件)

“伯通，要不你来运行这个程序。”药师微微笑道。周伯通按了一下“F5”键，只见后面 3 个按钮都按照不同角度进行旋转，周伯通又惊又喜。如图 10-15 所示。药师说道：“这是桃树林的另一种变换。”

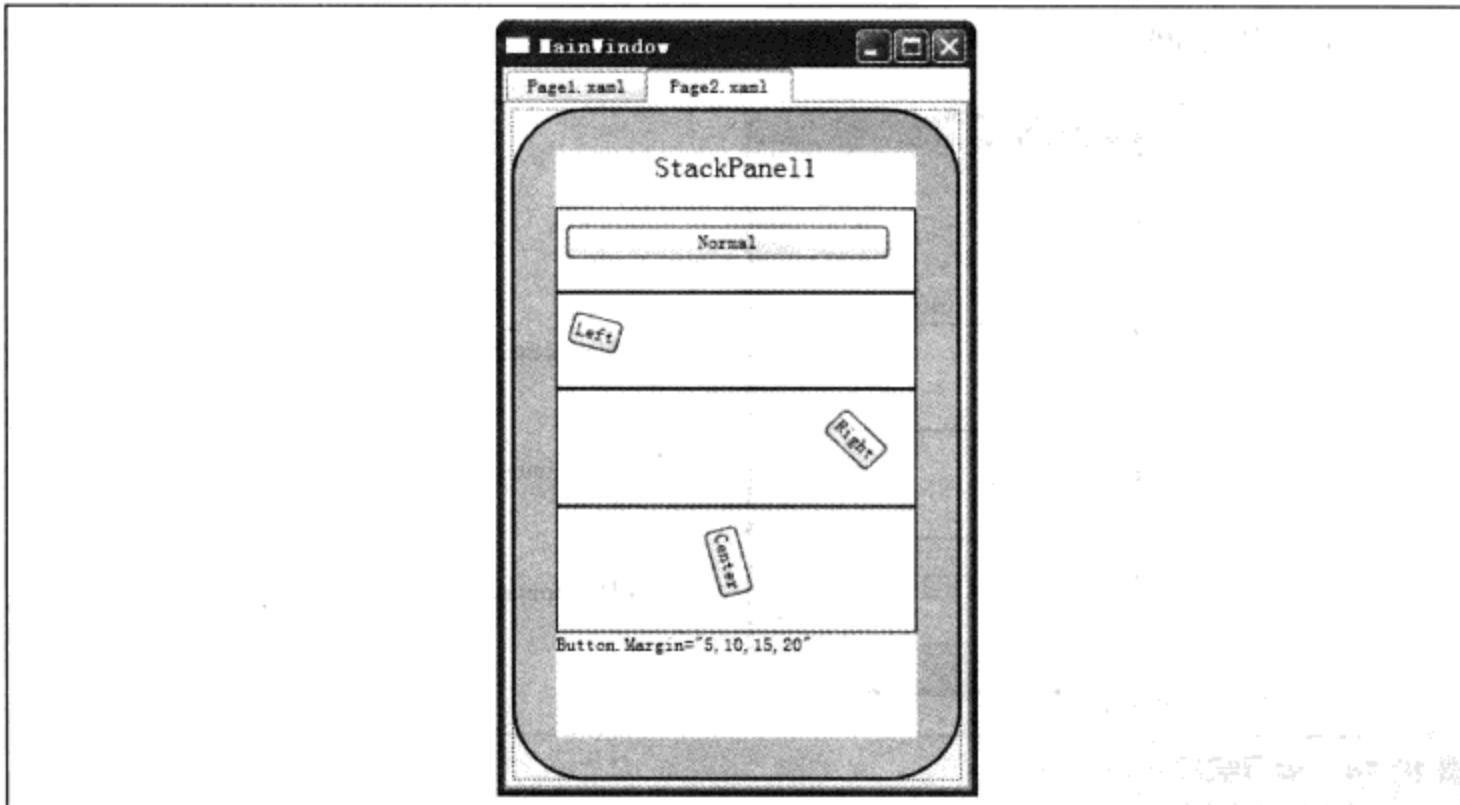


图 10-15 3 个按钮按照不同角度旋转变换

木木倒是陷入了沉思良久，拱手问到：“黄前辈……”药师眉头微皱喝道：“什么前辈不前辈，我黄老邪最烦人喊我前辈前辈个不停。”木木没有在乎这几句话，继续问到：“我刚才观察了一下 Button 除了有一个 LayoutTransform 属性，还有一个 RenderTransform 属性，为什么你不用后者呢？”黄老邪转怒为喜：“好小子，这其间的区别在于此。”说着又写下代码 10-12。

```

<Button Content="LayoutTransform">
    <Button.LayoutTransform>

```

```
<RotateTransform Angle = "15" />
</Button.LayoutTransform>
</Button>
<Button Content="RenderTransform">
    <Button.RenderTransform>
        <RotateTransform Angle = "15" />
    </Button.RenderTransform>
</Button>
```

代码 10-12 LayoutTransform 和 RenderTransform (Page3.xaml 文件)

运行结果如图 10-16 所示。

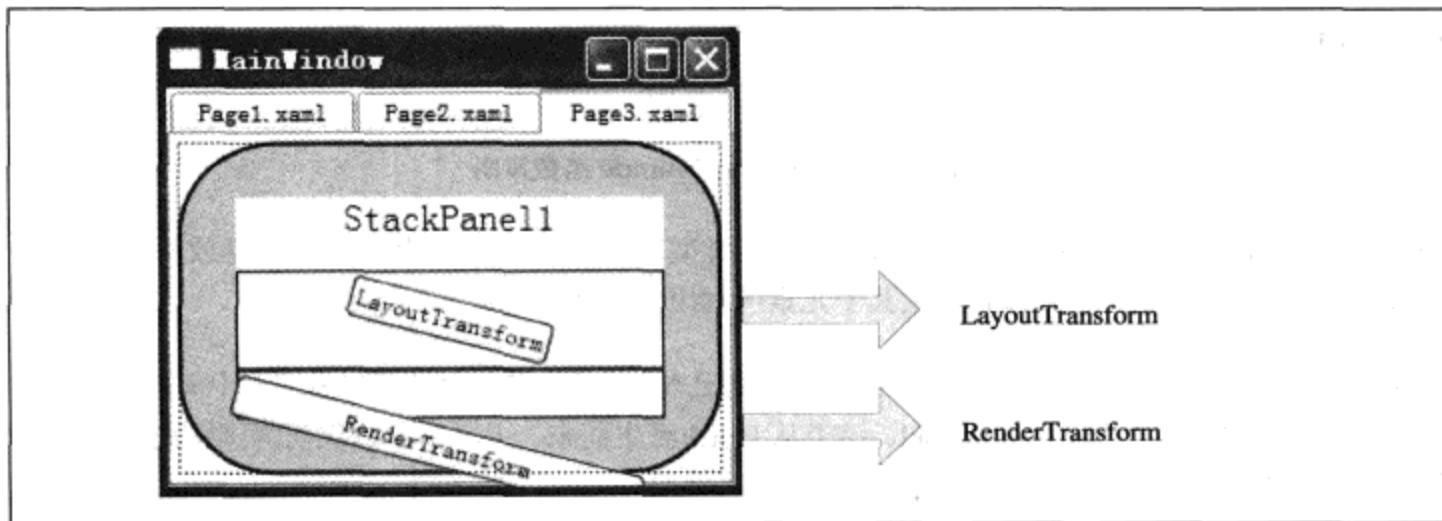


图 10-16 运行结果

“看出区别了没有，LayoutTransform 参与布局，而 RenderTransform 并不参与，因此应用 RenderTransform 的按钮会扩展到面板边界的外面。”药师说到。

“黄岛主，”木木已经不敢喊前辈二字了：“您前面的五片桃林，我大致能看个明白，唯独这第六片桃林……”殊不知这第六片桃林是黄药师集大成之作，黄药师虽然城府极深，但木木这一问也是挠到他的痒处，不禁问到：“怎样？”木木略一沉吟，说到：“我初看第六片桃树林，没有觉得什么稀奇的。相反现在越学越觉得第六片桃树林不简单，以前听周大哥说过，这个是岛主自定义的一个布局。现在想想的确任何一个布局都无法做到，因此觉得十分困惑，不明白黄岛主您是如何做到的？”

黄药师久居岛上，自己经常沉浸于这其中的精彩之处，但谁有人知，谁有人晓，听到木木这一说，虽没有任何赞誉之词，但却有遇到知音的感觉。黄药师脸上不动声色，但是心里大感快慰。“黄老邪，快讲讲你的自定义布局，我也好奇得紧。”老顽童嚷到，“不过老邪，这回老顽童是真佩服你了。”

药师见两人真诚，于是也不再有所保留，开始讲起了他的自定义布局……

10.3.2 自定义布局

要掌握自定义布局，必须了解布局的两个阶段。实际上确定控件最佳尺寸的经历了两个阶段，第 1

个阶段为测量（Measure）阶段，即父元素询问子元素所期望的尺寸，从而确定自身的尺寸；第 2 阶段为布置（Arrange）阶段，在这个期间每个父元素会告知子元素的尺寸和位置。

从编程模型来看，具体到两个重载函数，即 MeasureOverride 和 ArrangeOverride，如代码 10-13 所示。

```
protected override Size MeasureOverride(Size constraint)
```

代码 10-13 MeasureOverride 函数声明

MeasureOverride 传递的参数为 Size 类型，实际是上一级父元素告知当前元素可分配的空间（availableSize）；返回的参数 Size 类型，是该元素所期望的空间（desiredSize），理想的情况是如代码 10-14 所示。

```
protected override Size ArrangeOverride(Size finalSize)
```

代码 10-14 ArrangeOverride 函数声明

ArrangeOverride 传递和返回的参数同样是 Size 类型，传递的参数指定是该元素摆放所用的尺寸（finalSize）；返回参数同为该元素及其子元素所占用的尺寸。

黄药师见他们还是有很多疑惑，于是拿过木木的笔记本，把第六片桃林实现的原理用代码写了下来。首先黄药师自定义了一个 CustomPanel。该类从 Panel 派生而来，主要用于重载药师刚刚说的两个函数。如代码 10-15 所示。

```
CustomPanel.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace mumu_layout2
{
    public class CustomPanel : Panel
    {
        public CustomPanel()
            : base()
        {

        }

        protected override Size MeasureOverride(Size availableSize)
        {
            double maxChildWidth = 0.0;
            double maxChildHeight = 0.0;
            foreach (UIElement child in InternalChildren)
            {
                child.Measure(availableSize);
                maxChildWidth = Math.Max(child.DesiredSize.Width, maxChildWidth);
                maxChildHeight = Math.Max(child.DesiredSize.Height, maxChildHeight);
            }
            double idealCircumference = maxChildWidth * InternalChildren.Count;
            double idealRadius = (idealCircumference / (Math.PI * 2) + maxChildHeight);

            Size ideal = new Size(idealRadius * 2, idealRadius * 2);
            Size desired = ideal;
            if (!double.IsInfinity(availableSize.Width))
```

```

    {
        if (availableSize.Width < desired.Width)
            desired.Width = availableSize.Width;
    }
    if (!double.IsInfinity(availableSize.Height))
    {
        if (availableSize.Height < desired.Height)
            desired.Height = availableSize.Height;
    }
    return desired;
}

protected override Size ArrangeOverride(Size finalSize)
{
    Rect layoutRect;
    if (finalSize.Width > finalSize.Height)
    {
        layoutRect = new Rect((finalSize.Width - finalSize.Height) / 2,
0, finalSize.Height, finalSize.Height);
    }
    else
    {
        layoutRect = new Rect(0, (finalSize.Height - finalSize.Width) / 2,
finalSize.Width, finalSize.Width);
    }
    double angleInc = 360 / InternalChildren.Count;
    double angle = 0;
    foreach (UIElement child in InternalChildren)
    {
        Point childLocation = new Point(layoutRect.Left +
(layoutRect.Width - child.DesiredSize.Width) / 2, layoutRect.Top);

        child.RenderTransform = new RotateTransform(angle,
child.DesiredSize.Width / 2, finalSize.Height / 2 - layoutRect.Top);
        angle += angleInc;
        child.Arrange(new Rect(childLocation, child.DesiredSize));
    }
    return finalSize;
}
}
}

```

代码 10-15 自定义一个面板

黄药师随后新建一个页面，在其中调用 CutstomPanel，如代码 10-16 所示。

```

<Page .....
    xmlns:local="clr-namespace:mumu_layout2">
    <local:CustomPanel>
        <Button Background="#00000000" MinWidth="100">1</Button>
        <Button Background ="#FFFFCCFF" MinWidth="100">2</Button>
        <Button Background ="#FFFF9BFF" MinWidth="100">3</Button>
        <Button Background ="#FFFF00FF" MinWidth="100">4</Button>
        <Button Background="#FFFFCCFF" MinWidth="100">5</Button>
    </local:CustomPanel>
</Page>

```

代码 10-16 在页面中放置 CustomPanel

药师轻点运行，果然桃花岛的第六片桃林样子展现出来。这时老顽童是喜得手舞足蹈，木木却一反常态，细心体会黄岛主的代码。药师暗暗赞许，觉得此人确是可造之材。

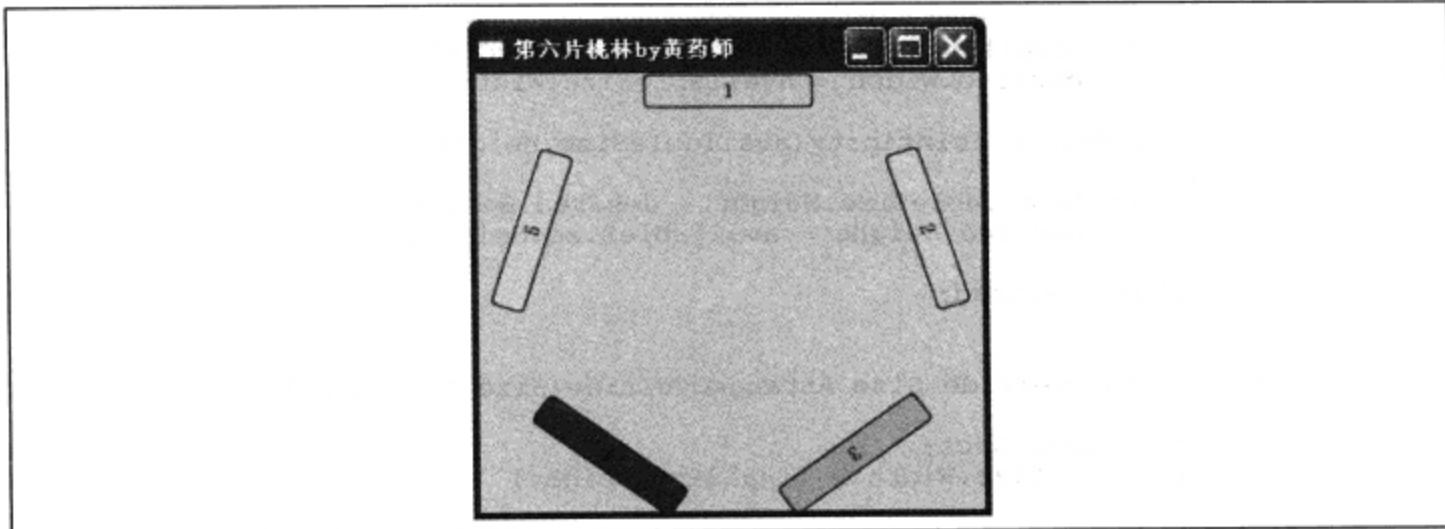


图 10-17 第 6 片桃林的布局

突然身后有人轻轻一笑，竹林里已闪身出一轻盈女子，长发披肩，全身白衣，头发上束了条金带，更是灿然生光。木木不由抬头，见这少女一身装束犹如仙女一般，不禁看得呆了。这少女正是黄蓉，“来，蓉儿”，不等药师招手，黄蓉早已飞一般地扑到爹爹怀里。药师笑到：“木木，你已通过我的测验，可以成为我的女婿了。”

木木还在发愣，被老顽童推了一个踉跄，“还不拜见岳父大人。”有道是布局巧设桃花岛，木木憨娶俏黄蓉。

10.4 接下来做什么

这是全书当中出现的第一个故事，后面还会出现类似这样的故事。布局解决的是窗口如何摆放控件的问题。接下来，我们需要知道的就是窗口摆放的内容——控件。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 射雕英雄传》，“第十六回 九阴真经”。

控件与 Content——北冥神功

段誉当下将帛卷又展开少许，见下面的字是：北冥神功系引世人之内力而为我有。北冥大水，非由自生。语云：百川江海，大海之水以容百川而得。汪洋巨浸，端在积聚……

—《天龙八部》，“第二章 玉壁月华明”^[1]

这一章讲的是段誉误入无量山剑湖底，在神仙姐姐玉像前得到北冥神功的武林秘笈。北冥神功是引人之内力而为己有，练习过北冥神功的段誉就好像是一个万能的容器。无论对手是谁，有多强，一旦被段誉抓住，内力尽数被吸走。

WPF 中也有这样万能的容器，无论什么部件，都可以包容其中。不妨先看图 11-1^[2]，这是一个普通的按钮。放大 7.5 倍后，其中包含一个完整的文档。放大 30 倍，还有一个按钮。放大 375 倍，有一张完整的图片。

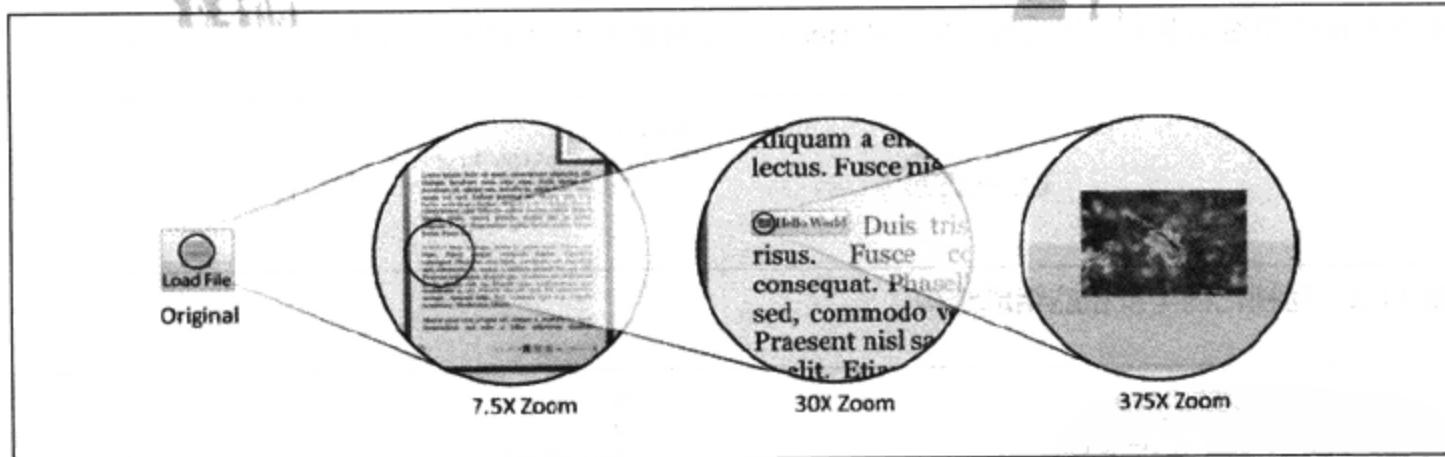


图 11-1 按钮下面的秘密^[2]

这样奇妙的效果来自 WPF 的 Content 模型（Content Model），可以说是整个 WPF 的精髓之一。一个 Content 犹如北冥神功一般，无论对象、控件、文档和媒体，或是其他，都是海纳百川，无所不包的。

本章内容如下。

- (1) 缘起。
- (2) Content 模型及其家族。

(3) 经典控件。

(4) 接下来做什么。

11.1 缘起

控件的出现由来已久。WPF 的控件出现之前已经有多种控件框架，这些控件框架的一个关键问题是缺乏一致的灵活性。

为了理解这个概念，不妨先看 Win32 的控件。图 11-2 所示为一个标准按钮、滚动条，以及一个标题栏中的最小化、还原和关闭按钮，其中均有按钮，但是其行为完全不同，并且有不同的实现方式。即尽管都是按钮，但是完全不能复用。WPF 团队希望提供一个按钮，并且可以用于任何地方，这就是一致的灵活性。

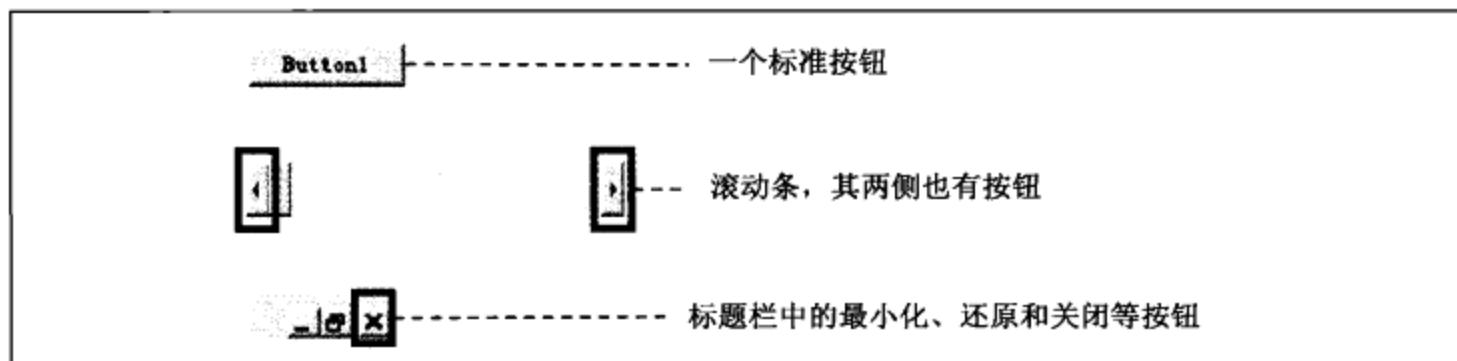


图 11-2 Win32 下的传统控件

WPF 采用了比 Control 更小的粒度单元，即按钮和滚动条由更小的单元组成。我们可以使用 XamlPad 来查看按钮和滚动条的可视化树（Visual Tree），分别如图 11-3 和图 11-4 所示。

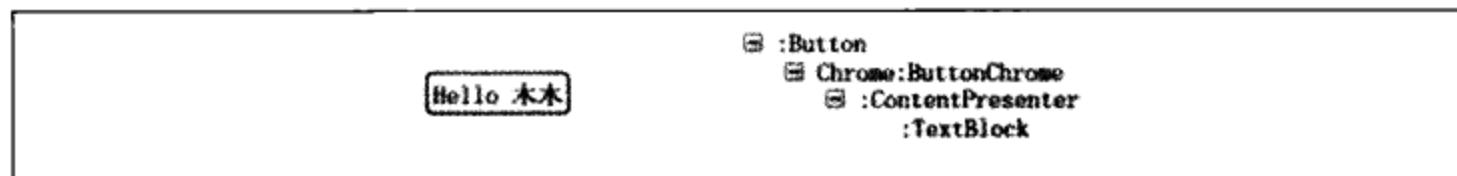


图 11-3 可视化树中按钮的组成结构

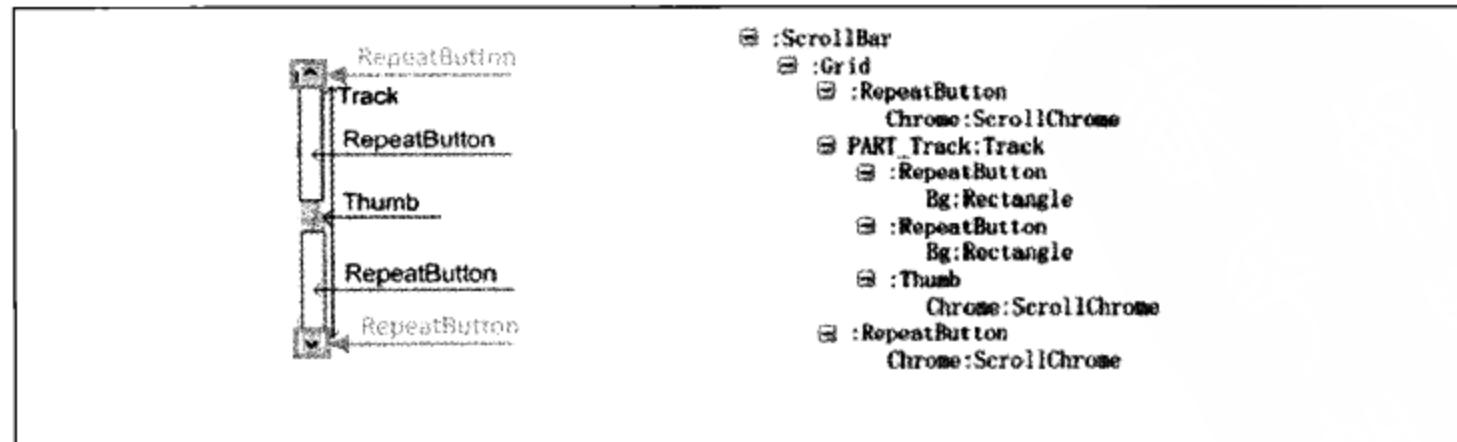


图 11-4 可视化树中滚动条的组成结构

另外在 WPF 设计初期，其团队的一个宏伟目标是为 UI、文档和媒体提供一个集成的平台。以按钮为例，它有一个类型为 Object 的 Content 属性。图 11-15 所示的从上至下的 4 个按钮依次为标准按钮，以及包含 UI 元素、文档元素和视频元素的按钮。

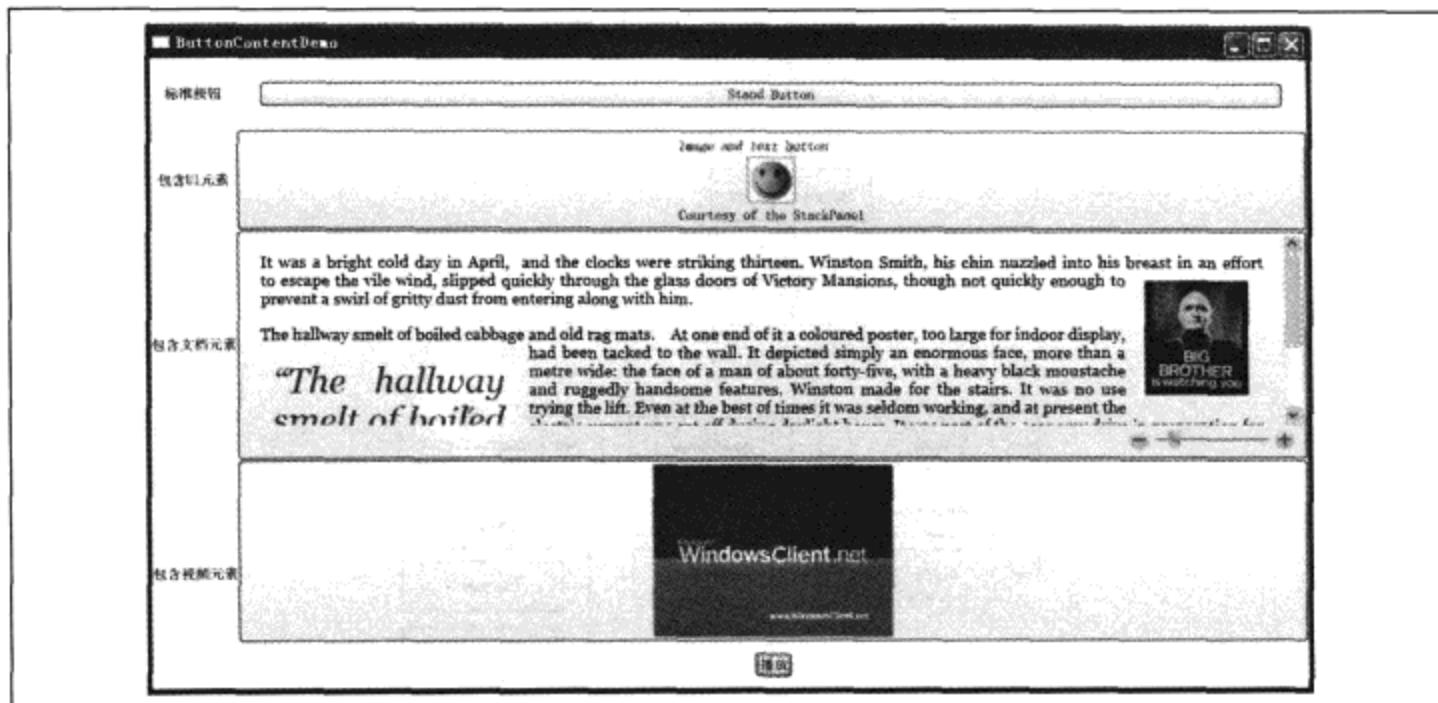


图 11-5 包含各种元素的按钮例子（源码参见 `mumu_ButtonContentDemo`）

“天下武功无不为我所用，犹之北冥，大舟小舟无不载，大鱼小鱼无不容。”木木不由自言自语，内心已经无比向往 Content 模型。

11.2 Content 模型及其家族

11.2.1 Content 模型

从 ContentControl 继承的类均包含一个 Content 属性，图 11-6 所示为其类继承结构。

Content 模型是如何达到“大舟小舟无不载，大鱼小鱼无不容”的能力？从图 11-3 中，不难发现 Button 的组成当中有一个相当重要的类 ContentPresenter，该类根据需要将 Content 显示为一个字符串、日期、图片或者一个视频。

Content 将所有元素分为两类，即 UIElement 和非 UIElement 类，前者 WPF 会利用 OnRender 方法来显示前者所需要的内容；后者则利用一般类都会有的方法 ToString 来显示其内容。

Content 也不是万能的，它也有若干个限制。其中最大的限制是一个 Content 只允许设置一种元素，解决方案之一是将一个面板（StackPanel 和 Grid 等）设置给 Content，然后在面板中放置其他元素；方案之二是完全抛弃 ContentControl 的派生类，而使用 ItemsControl 及其派生类。Content 的另外两个限制是不可以将 Window 这样的“树根”类型的值赋值给它。

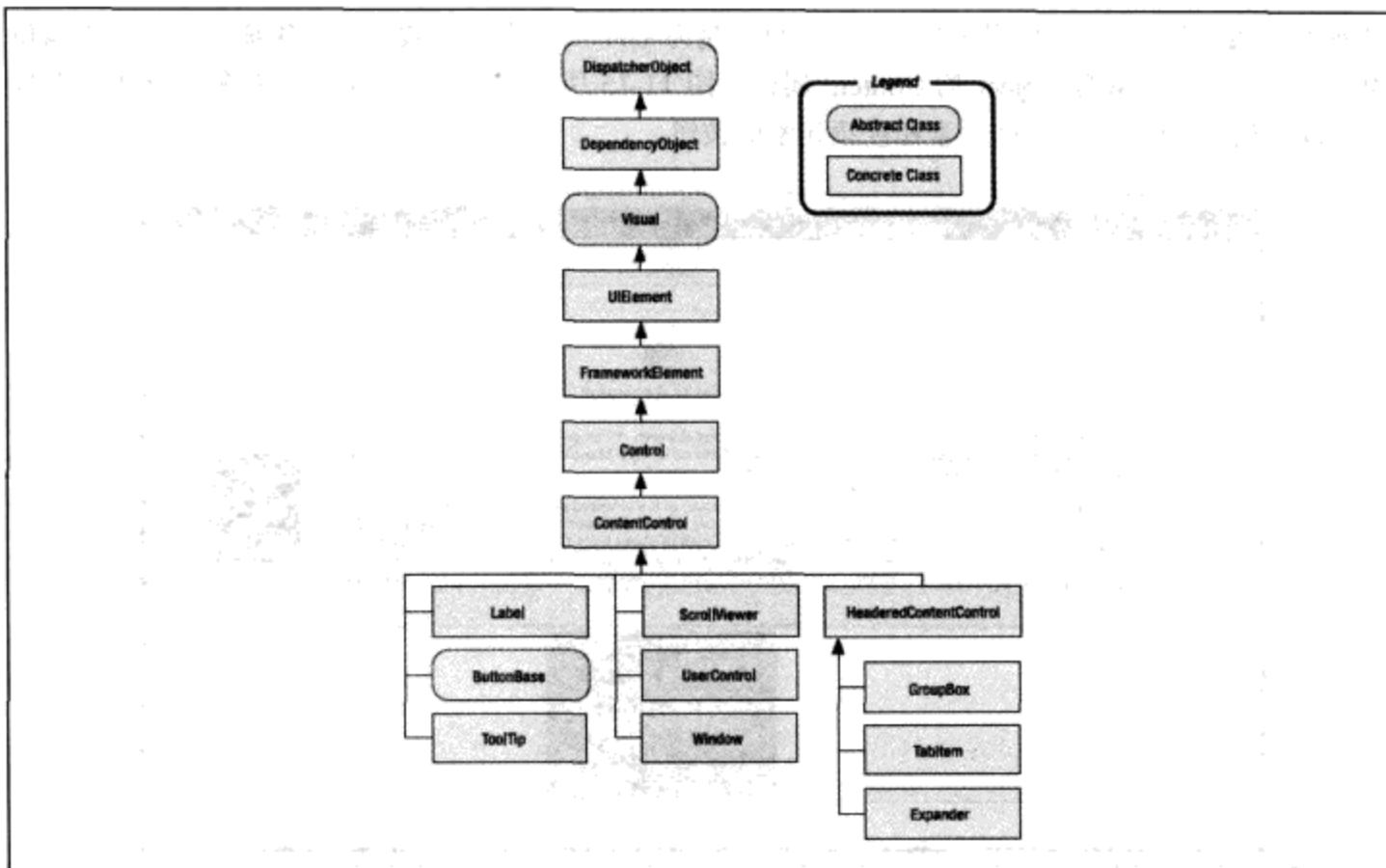


图 11-6 ContentControl 的类继承结构

11.2.2 Content 家族

Content 家族有它的直系和远房亲属。

1. 4 大直系

Content 的 4 大直系都是从 Control 派生的，分别是 ContentControl、HeaderedContentControl、ItemsControl 和 HeaderedItemsControl，如图 11-7 所示。

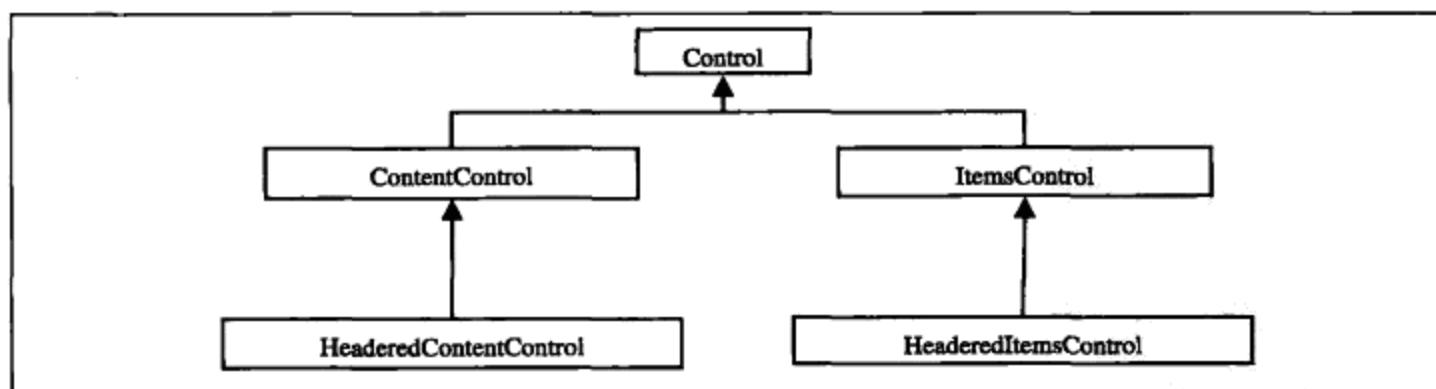


图 11-7 Content 模型的 4 大直系

从 ContentControl 派生的类均包含一个 Content 属性，Button、Window、CheckBox 和 ToolTip 这样的控件均派生自 ContentControl。

HeaderContentControl 继承自 ContentControl，与其不同的是除了 Content 属性，还包含一个 Header 属性。该属性为 Object 类型，可以包容文本、UI 和媒体等多种元素，GroupBox 控件派生自该类。

ItemsControl 和 ContentControl 不同，它包含一个 Content 集合。这个集合在 ItemsControl 中用 Items 或者 ItemsSource 属性来表示，Menu、ListBox、ListView、TreeView 和 TabControl 等控件均派生自该类。

HeaderItemsControl 派生自 ItemsControl，二者的关系和 HeaderedContentControl 与 ContentControl 相同。HeaderItemsControl 除了包含一个集合外，还包含一个 Header 属性。其中可以包容文本、UI 和媒体等多种元素，MenuItem 派生自该类。

2. 远亲

虽然有一些类型并没有包含 Content 属性，但其组织方式类似 Content 模型。因此将其归类为该模型的远亲，如图 11-8 所示。

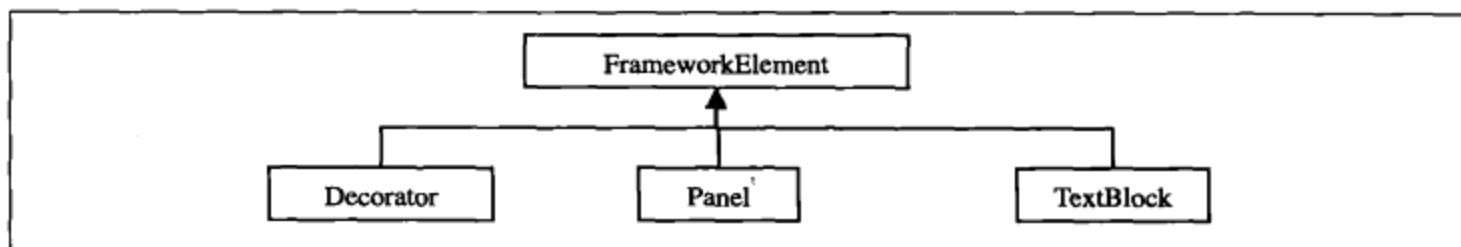


图 11-8 Content 模型的远亲

Decorator（装饰者）会围绕其子控件添加一些效果，如边框等。它有一个 Child 属性，其类型为 UIElement。即 Decorator 可以为任何 UIElement 提供装饰，Border 类派生自 Decorator。

Panel 和 Decorator 不同，后者只有一个孩子（Child）；而前者有一群孩子（Children）。在自定义面板时使用该属性，StackPanel 和 Grid 等都派生自 Panel。

TextBlock 是一个轻量级的文本控件，比 Label 更轻量，后者使用 Content 属性来设置文本；后者通过 inlines 属性来设置文本。通过 inlines 属性可以为一段话设置多种不同的字体，如代码 11-1 所示（详见 mumu_textblock 工程）。

```
<Window x:Class="mumu_textblock.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TextBlockDemo" Height="300" Width="300">
    <StackPanel>
        <TextBlock>
            <Run FontSize="30" TextDecorations="underline">木木,</Run>
            <Run FontSize="50" Foreground="Purple"> 北冥神功</Run>
        </TextBlock>
    </StackPanel>
</Window>
```

代码 11-1 为一段文本设置多种不同字体

程序运行结果如图 11-9 所示。



图 11-9 程序运行结果

11.3 经典控件

理解了 Content 模型及其家族，可以发现尽管控件的个数很多，但是其开始分门别类且条理更为清晰。

11.3.1 Content 控件

Content 控件均继承自 ContentControl，比较常用的是 Label（标签）、Button（按钮）和 ToolTip 控件。

1. Label 控件

Label 是一种最简单的控件，传统上该控件在表单（Form）或者对话框中显示一段文本。WPF 中的 TextBlock 功能似乎比 Label 更为强大。但是 Label 自然有它存在的道理，和 TextBlock 相比，WPF 中的 Label 控件支持以键盘快捷键的方式获得焦点，可以使得与其有密切关系的控件获得焦点，如代码 11-2 所示（详见 mumu_label 工程）。

```
<Window x:Class="mumu_label.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
<StackPanel Margin="5">
    <Label Target="{Binding ElementName=txtA}">Choose _A</Label>
    <TextBox Name="txtA"></TextBox>
    <Label Target="{Binding ElementName=txtB}">Choose _B</Label>
    <TextBox Name="txtB"></TextBox>
</StackPanel>
</Window>
```

代码 11-2 Label 的用法

程序运行结果如图 11-10 所示。

当按下 Alt+A 快捷键，光标会切换到第 1 个文本框；按下 Alt+B 快捷键会切换到第 2 个文本框。在 WPF 中通过下画线_ 设置快捷键的字母，Label 使用与 Target 属性关联的快捷键和相关控件。

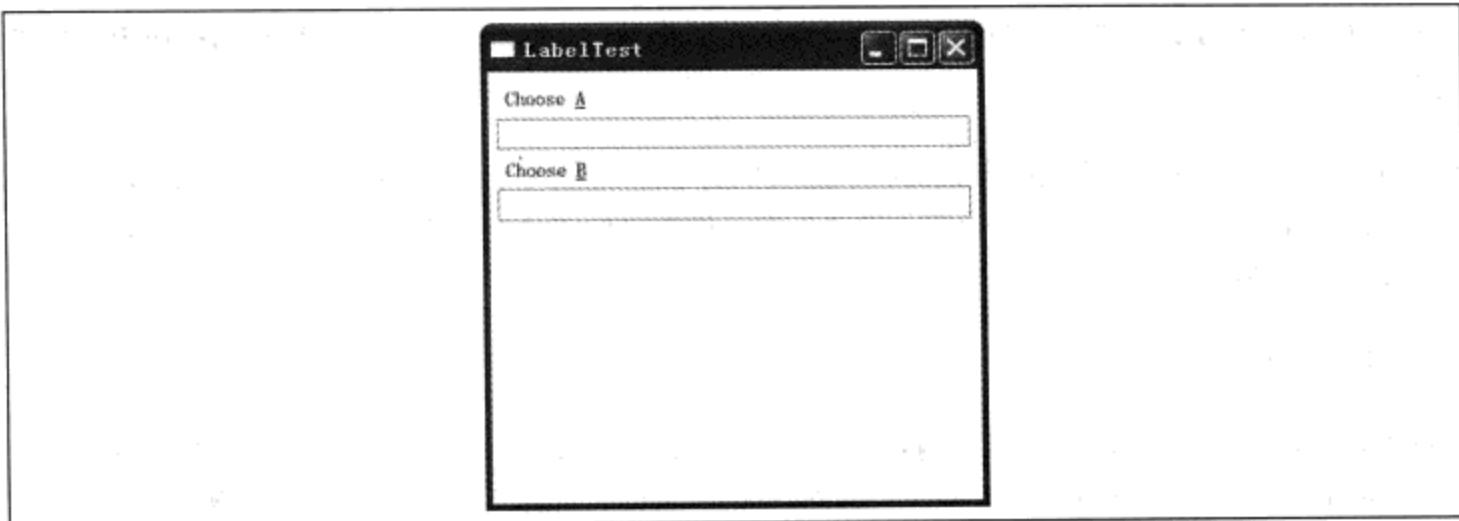


图 11-10 程序运行结果

2. Button 控件

在 WPF 中 Button 控件是一个可以被单击，但不能被双击的 Content 控件。ButtonBase 类将这种行为抽象出来，图 11-11 所示为 ButtonBase 及其派生类。

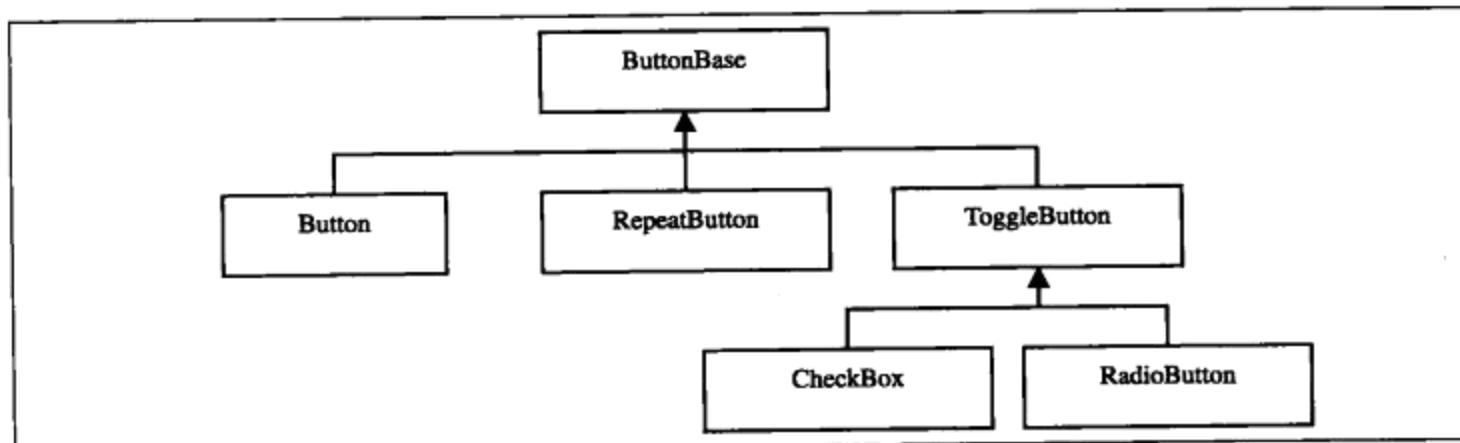


图 11-11 ButtonBase 及其派生类

(1) ButtonBase

ButtonBase 类定义了 Click 事件和 ClickMode 属性，通过该属性使用枚举值可以准确控制何时触发 Click 事件。默认值为 Release，即鼠标释放时触发 Click 事件；此外还有 Press（鼠标按下时触发）和 Hover（鼠标滑过按钮时触发）等。

(2) Button

Button 在 ButtonBase 的基础上增加了 IsDefault 和 IsCancel 属性，如果 IsDefault 属性设置为 true，则即使焦点不在这个按钮上，按下回车键也会触发 Click 事件。如果对话框中的一个按钮的 IsCancel 属性设置为 true，那么按下 ESC 键就会触发该按钮的 Click 事件。

(3) RepeatButton 控件

滚动条中包含 RepeatButton 控件，它属于 System.Windows.Controls.Primitives 命名空间，该命名空间

提供控件的基类或者组成更复杂控件的基本控件。因此 RepeatButton 一般不单独使用，而用于组成更复杂的控件。

RepeatButton 会在按钮一直被按着的情况下触发单击事件，产生单击事件的频率由其 Delay 及 Interval 属性指定。RepeatButton 主要是应用于滚动条中，如当鼠标在其上面长时间未释放时，滚动条下部的按钮可以重复按下行为。

(4) ToggleButton

ToggleButton 是一种在单击时可以保持状态的按钮，第 1 次被单击时其 IsChecked 属性会被设置为 true；再单击一次则变为 false。该控件的 IsThreeState 属性为 true，IsChecked 会有 3 种值。即 true、false 和 null，其中 True 对应 Checked 事件；false 对应 UnChecked 事件；null 对应 Indeterminate 事件。

ToggleButton 也属于 System.Windows.Controls.Primitives 命名空间，说明 WPF 并不期望用户直接使用它，但是可以使用 CheckBox 和 RadioButton 两个派生于 ToggleButton 的类。

(5) CheckBox 控件

CheckBox CheckBox 除了外观和 ToggleButton 外，其他均与 ToggleButton 相同。

(6) RadioButton 控件

RadioButton 是一种从 ToggleButton 继承的控件，它支持互斥性，当多个 RadioButton 放在一个组中时每次只有一个被选中。一般情况下，组是直观的概念，如把所有的 RadioButtons 放在一个 StackPanel 中就称为一个组。如果需要自定义方法分组 RadioButton，则需要使用 GroupName 属性，如代码 11-3 所示（详见 mumu_RadioButtonGroups 工程）。

```
<Window x:Class="mumu_RadioButtonGroups.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
    <StackPanel>
        <GroupBox Margin="5">
            <StackPanel>
                <RadioButton>Group 1</RadioButton>
                <RadioButton>Group 1</RadioButton>
                <RadioButton>Group 1</RadioButton>
                <RadioButton Margin="0,10,0,0" GroupName="Group2">Group 2</RadioButton>
            </StackPanel>
        </GroupBox>
        <GroupBox Margin="5">
            <StackPanel>
                <RadioButton>Group 3</RadioButton>
                <RadioButton>Group 3</RadioButton>
                <RadioButton>Group 3</RadioButton>
                <RadioButton Margin="0,10,0,0" GroupName="Group2">Group 2</RadioButton>
            </StackPanel>
        </GroupBox>
    </StackPanel>
</Window>
```

代码 11-3 使用 GroupName

程序运行结果如图 11-12 所示，其中分为 3 个组，第 2 组并不在一个 GroupBox 中。

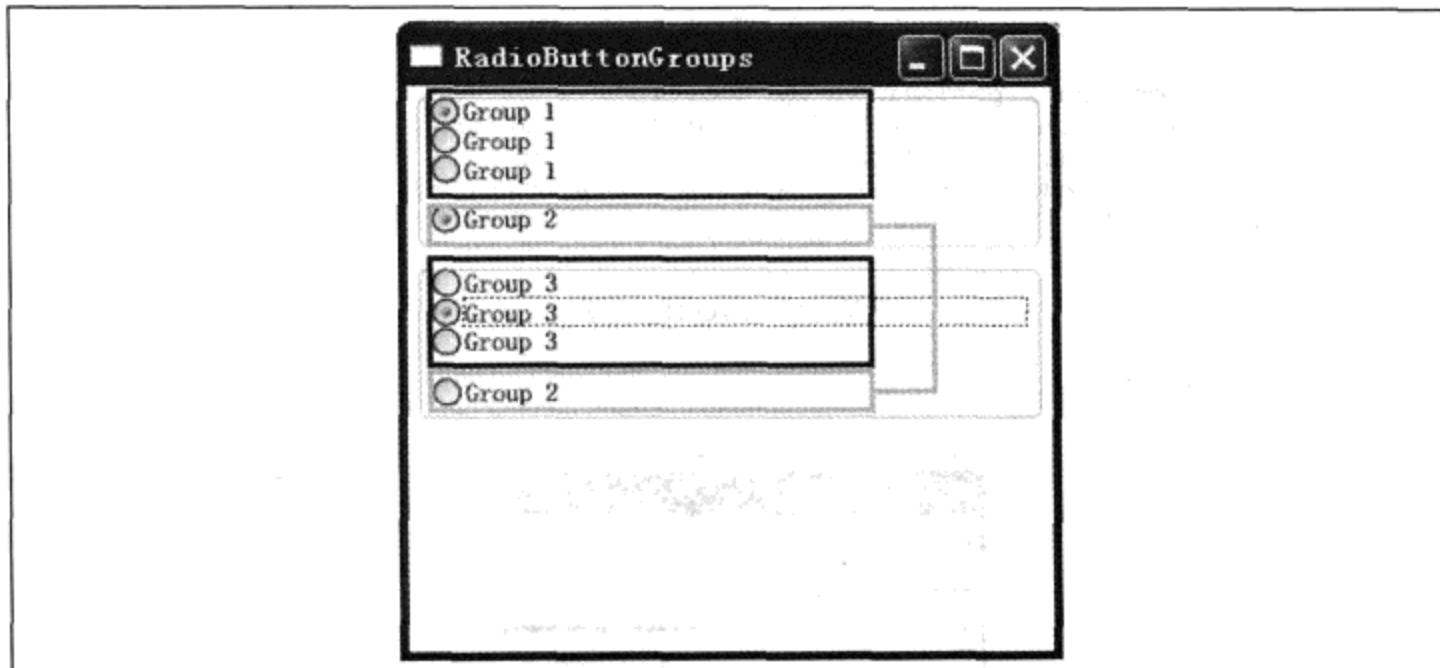


图 11-12 程序运行结果

3. ToolTip

ToolTip 控件将其内容放在一个浮动框中，当鼠标移过与之关联的控件时会显示该控件的内容；移开以后则消失。ToolTip 控件不能直接放在 UI 元素树中，必须被赋给另一个元素的 ToolTip 属性（FrameworkElement 和 FrameContentElement 中均定义了该属性）。

ToolTip 不会获得焦点，而且不能单击或者与其交互。有时可能我们需要在 ToolTip 出现或者消失时执行某种操作，ToolTip 定义了 Open 和 Closed 事件及其属性来调节其行为。ToolTipService 定义了一些附加属性，能够设置在任何一个使用 ToolTip 的元素上并且有多个与 ToolTip 相同的属性，但是优先级更高。它还有一些 ToolTip 不具备的属性，如 ShowDuration 属性控制鼠标悬停在一个元素后多长时间显示 ToolTip，InitialShowDelay 属性控制 ToolTip 第 1 次显示的时间长度等。

代码 11-4 所示是一个 ToolTip 的例子（详见 mumu_ToolTips 工程）。

```
<Window x:Class="mumu_ToolTips.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="321" Width="301" >
    <StackPanel Margin="5" ToolTip="StackPanel tooltip">
        <Button ToolTip="This is my tooltip">
            <ToolTip Service.InitialShowDelay="5000">I have a tooltip</ToolTip>
            <Button ToolTipService.InitialShowDelay="0" ToolTipService.BetweenShowDelay="5000">
                <Button.ToolTip>
                    <ToolTip Background="#60AA4030" Foreground="White" HasDropShadow="False" >
                        <StackPanel>
                            <TextBlock Margin="3" >Image and text</TextBlock>
                            <Image Source="happyface.jpg" Stretch="None" />
                            <TextBlock Margin="3" >Image and text</TextBlock>
                
```

```

        </StackPanel>
    </ToolTip>
    </Button.ToolTip>
    <Button.Content>I have a fancy tooltip</Button.Content>
</Button>
<Button ToolTip="This is my tooltip"
    ToolTipService.Placement="Bottom">Placement test</Button>
<Button Padding="50">Does Nothing</Button>
<TextBox TextWrapping="Wrap" MinLines="2" AutoWordSelection="True"></TextBox>
</StackPanel>
</Window>

```

代码 11-4 ToolTip 示例

程序运行结果如图 11-13 所示。



图 11-13 程序运行结果

11.3.2 HeaderedContent 控件

HeaderedContent 控件是继承 HeaderedContentControl 而来。

1. GroupBox

GroupBox 是一种用来组织多种控件的常见控件，通常用来包含多个项。但是由于它是内容控件，所以只能直接包含一项。如果需要包含多项，则必须使用一个可以包含多个子内容的中间控件，如 StackPanel 等。GroupBox 的 Header 和 Content 属性可以被设置为任意类型的对象，代码 11-5 所示是一个GroupBox 的示例（详见 mumu_groupboxdemo 工程）。

```

<Window x:Class="mumu_groupboxdemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300">
<Grid>
    <GroupBox>
        <GroupBox.Header>
            <WrapPanel>
                <Image Source ="duanyuicon.jpg"/>

```

```

        <TextBlock Text="段誉吸人内力列表:" />
    </WrapPanel>
</GroupBox.Header>
<GroupBox.Content>
    <StackPanel TextBlock.FontSize="20" >
        <TextBlock Text="追杀木婉清的一帮人"/>
        <TextBlock Text="云中鹤"/>
        <TextBlock Text="段正淳等"/>
        <TextBlock Text="燕子坞老太婆"/>
        <TextBlock Text="鸠摩智"/>
    </StackPanel>
</GroupBox.Content>
</GroupBox>
</Grid>
</Window>

```

代码 11-5 GroupBox 示例

程序运行结果如图 11-14 所示。



图 11-14 程序运行结果

2. Expander

Expander 类似 GroupBox，但是包含一个按钮，可以展开或者折叠其中包含的内容（默认处于折叠状态）。该控件定义了 IsExpanded 属性及 Expanded/Collapsed 事件，并且可以用 ExpandDirection 属性控制其方向。将上例改为 Expander 形式。如代码 11-6 所示（详见 mumu_expanderDemo 工程）。

```

<Window x:Class="mumu_expanderDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ExpanderDemo" Height="300" Width="300">
    <StackPanel>
        <Expander>
            <Expander.Header>
                <WrapPanel>
                    <Image Source ="duanyuicon.jpg" Height="36" Width="29" />
                    <TextBlock Text="段誉吸人内力列表:" />
                </WrapPanel>
            </Expander.Header>
            <Expander.Content>
                <StackPanel TextBlock.FontSize="20" >

```

```
<TextBlock Text="追杀木婉清的一帮人"/>
<TextBlock Text="云中鹤"/>
<TextBlock Text="段正淳等"/>
<TextBlock Text="燕子坞老太婆"/>
<TextBlock Text="鸠摩智"/>
</StackPanel>
</Expander.Content>
</Expander>
<Expander ExpandDirection="Left">
<Expander.Header>
<WrapPanel>
<Image Source ="duanyuicon.jpg" Height="36" Width="29" />
<TextBlock Text="段誉吸人内力列表:"/>
</WrapPanel>
</Expander.Header>
<Expander.Content>
<StackPanel TextBlock.FontSize="20" >
<TextBlock Text="追杀木婉清的一帮人"/>
<TextBlock Text="云中鹤"/>
<TextBlock Text="段正淳等"/>
<TextBlock Text="燕子坞老太婆"/>
<TextBlock Text="鸠摩智"/>
</StackPanel>
</Expander.Content>
</Expander>
<Expander ExpandDirection="Right">
<Expander.Header>
<WrapPanel>
<Image Source ="duanyuicon.jpg" Height="36" Width="29" />
<TextBlock Text="段誉吸人内力列表:"/>
</WrapPanel>
</Expander.Header>
<Expander.Content>
<StackPanel TextBlock.FontSize="20" >
<TextBlock Text="追杀木婉清的一帮人"/>
<TextBlock Text="云中鹤"/>
<TextBlock Text="段正淳等"/>
<TextBlock Text="燕子坞老太婆"/>
<TextBlock Text="鸠摩智"/>
</StackPanel>
</Expander.Content>
</Expander>
<Expander ExpandDirection="Up">
<Expander.Header>
<WrapPanel>
<Image Source ="duanyuicon.jpg" Height="36" Width="29" />
<TextBlock Text="段誉吸人内力列表:"/>
</WrapPanel>
</Expander.Header>
<Expander.Content>
<StackPanel TextBlock.FontSize="20" >
<TextBlock Text="追杀木婉清的一帮人"/>
<TextBlock Text="云中鹤"/>
<TextBlock Text="段正淳等"/>
<TextBlock Text="燕子坞老太婆"/>
<TextBlock Text="鸠摩智"/>
</StackPanel>
</Expander.Content>
```

```
</Expander>
</StackPanel>
</Window>
```

代码 11-6 Expander 代码

程序运行结果如图 11-15 所示。

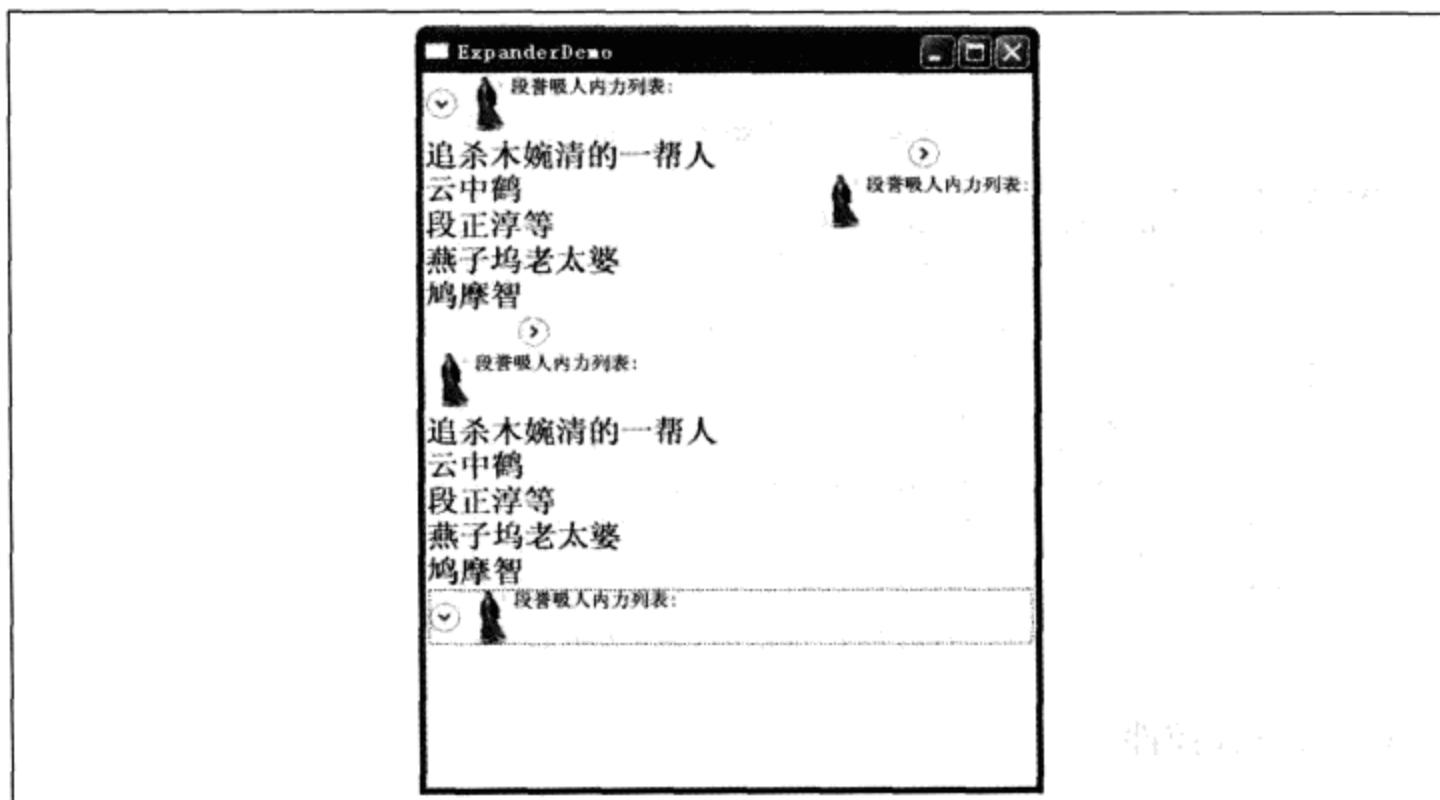


图 11-15 程序运行结果

为简化代码，可以新建一个名为“ExpanderEx”的类，如代码 11-7~代码 11-9 所示。

```
Expander.xaml
<Expander x:Class="mumu_expanderDemo.ExpanderEx"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Expander.Header>
        <WrapPanel>
            <Image Source ="duanyuicon.jpg" Height="36" Width="29" />
            <TextBlock Text="段誉吸人内力列表:"/>
        </WrapPanel>
    </Expander.Header>
    <Expander.Content>
        <StackPanel TextBlock.FontSize="20" >
            <TextBlock Text="追杀木婉清的一帮人"/>
            <TextBlock Text="云中鹤"/>
            <TextBlock Text="段正淳等"/>
            <TextBlock Text="燕子坞老太婆"/>
            <TextBlock Text="鸠摩智"/>
        </StackPanel>
    </Expander.Content>
</Expander>
```

代码 11-7 Expander.xaml

```
Expander.xaml.cs
namespace mumu_expanderDemo
{
    public partial class ExpanderEx : Expander
    {
        public ExpanderEx()
        {
            InitializeComponent();
        }
    }
}
```

代码 11-8 Expander.xaml.cs

```
MainWindow.xaml.cs
<Window x:Class="mumu_expanderDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:mumu_expanderDemo"
    Title="ExpanderDemo" Height="300" Width="300">
    <StackPanel>
        <local:ExpanderEx />
        <local:ExpanderEx ExpandDirection="Left" />
        <local:ExpanderEx ExpandDirection="Right" />
        <local:ExpanderEx ExpandDirection="Up" />
    </StackPanel>
</Window>
```

代码 11-9 MainWindow.xaml.cs

11.3.3 Items 控件

Items 控件继承自 ItemsControl，包含一个拥有多个 Item 集合的 Content。每个 Item 可以是任意对象，ItemsControl 将其内容保存在 Items 属性中。

Items 控件除了 Items 属性之外，还有一个 ItemsSource 属性。该属性可以将一个任意类型的集合赋给 Items 集合，一般多用于数据绑定。

Items 控件中有多个复杂控件，本节仅简单介绍 ListBox、ComboBox 和 Menu。

1. ListBox

ListBox 为用户提供了一个选项列表，该列表可以是固定的或动态绑定，它与 ComboBox 的一个显著不同是其中的所有选项可以对用户可见。

ListBox 提供了 SelectionMode 属性，其取值及其说明如表 11-1 所示。

表 11-1 ListBox 的 SelectionMode 属性

值	说明
Single	用户一次只能选择一项
Multiple	用户无需按下 Ctrl 键或者 Shift 键就可以选择多项
Extended	用户可以按住 Shift 键选择多个连续项，或按住 Ctrl 键并选择多个非连续项

代码 11-10 是一个 `ListBox` 示例（详见 `mumu_ListBoxDemo` 工程）。

```
MainWindow.xaml
<Window x:Class="mumu_ListBoxDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <GroupBox Margin="5">
            <GroupBox.Header>
                <TextBlock Text="选择 SelectionMode 属性"></TextBlock>
            </GroupBox.Header>
            <GroupBox.Content>
                <StackPanel Margin="3" Background="AliceBlue" RadioButton.Checked=
"Radio_Checked">
                    <RadioButton Name="radioSingle">Single</RadioButton>
                    <RadioButton Name="radioMultiple">Multiple</RadioButton>
                    <RadioButton Name="radioExtended">Extended</RadioButton>
                </StackPanel>
            </GroupBox.Content>
        </GroupBox>
        <GroupBox Margin="5">
            <GroupBox.Header>
                <TextBlock>效果</TextBlock>
            </GroupBox.Header>
            <GroupBox.Content>
                <ListBox Name="lb">
                    <ListBoxItem>Item 1</ListBoxItem>
                    <ListBoxItem>Item 2</ListBoxItem>
                    <ListBoxItem>Item 3</ListBoxItem>
                    <ListBoxItem>Item 4</ListBoxItem>
                    <ListBoxItem>Item 5</ListBoxItem>
                </ListBox>
            </GroupBox.Content>
        </GroupBox>
    </StackPanel>
</Window>
```

后台代码如下：

```
MainWindow.xaml.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace mumu_ListBoxDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
```

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void Radio_Checked(object sender, RoutedEventArgs e)
    {
        RadioButton currentRadioBtn = (RadioButton)e.OriginalSource;
        string selectMode = currentRadioBtn.Name;
        switch (selectMode)
        {
            case "radioSingle":
                lb.SelectionMode = SelectionMode.Single;
                break;
            case "radioMultiple":
                lb.SelectionMode = SelectionMode.Multiple;
                break;
            case "radioExtended":
                lb.SelectionMode = SelectionMode.Extended;
                break;
        }
    }
}

```

代码 11-10 MainWindow.xaml 和 MainWindow.xaml.cs 文件

程序运行结果如图 11-16 所示。



图 11-16 程序运行结果

2. ComboBox

ComboBox 是一种经典控件，允许用户在一个列表中选择一个 Item，其中的下拉列表框可以通过单击或者按 Alt+↑ 组合键、Alt+↓ 组合键和 F4 键打开并关闭。ComboBox 定义了 DropDownOpened 和 DropDownClosed 事件，允许打开或者关闭下拉列表框时执行相应的操作。代码 11-11 中的 Item 保存段誉等人的照片及其简历。

```

MainWindow.xaml
<Window x:Class="mumu_comboBoxDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="天龙八部人物谱" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="0.3*"/>
            <RowDefinition Height="0.7*"/>

```

```

</Grid.RowDefinitions>
<TextBlock Grid.Row ="0" Text="天龙八部" HorizontalAlignment="Center"
VerticalAlignment="Center" FontSize="25"/>
<ComboBox Grid.Row="1" HorizontalAlignment="Stretch" VerticalAlignment=
"Center" >
    <StackPanel Orientation="Horizontal" Margin="5">
        <Image Source="duanyu.jpg"/>
        <StackPanel Width="200" >
            <TextBlock Margin="5.0" FontSize="24" FontWeight="Bold"
VerticalAlignment="Center" Text="段誉"/>
            <TextBlock Margin="5.0" FontSize="14" TextWrapping="Wrap">
                看他一袭青衫，容仪如玉，明净柔和，有着说不出的与生俱来的优雅气质。他毫无
世俗间的心机，纯为一派天真，爽朗而通达，隽秀的脸上永远也不会显露出尘世间经常可以见到的冷酷傲慢的
表情。他是理想中的书生形象，即使其迂腐的一面也让人觉得可喜可爱。
            </TextBlock>
        </StackPanel>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5">
        <Image Source="quanfeng.jpg"/>
        <StackPanel Width="200" >
            <TextBlock Margin="5.0" FontSize="24" FontWeight="Bold"
VerticalAlignment="Center" Text="萧峰"/>
            <TextBlock Margin="5.0" FontSize ="14" TextWrapping="Wrap">
                在萧峰的面前，既往的一切陈述都变得苍白和空洞，无可阻挡地进行价值的消解和
缺失。萧峰的出现是空谷来风，是平地的一声春雷，是我们所有凡人琐屑生活中梦寐以求渴望的高贵气息，是
英雄有力、骄傲、坚定的自白。段誉喝彩道：“好一条大汉！这定是燕赵北国的悲歌慷慨之士。”仅此一句话，
就足可表现出萧峰天人般大气磅礴、神威凛凛之气势。
            </TextBlock>
        </StackPanel>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5">
        <Image Source="xuzhu1.jpg"/>
        <StackPanel Width="200" >
            <TextBlock Margin="5.0" FontSize="24" FontWeight="Bold"
VerticalAlignment="Center" Text="虚竹"/>
            <TextBlock Margin="5.0" FontSize ="14" TextWrapping="Wrap">
                虚竹对佛的信仰不可置疑，但为何他却屡犯死规，更破了佛家杀戒、淫戒、荤戒三
大戒。这其实是人性的力量，佛家处处限制人欲、人性，而虚竹出寺之后，内心深处的人性人欲被激发，对佛
的信仰终究敌不过人性人欲的力量，最终破戒、还俗。
            </TextBlock>
        </StackPanel>
    </StackPanel>
</ComboBox>
</Grid>
</Window>

```

代码 11-11 ComboBox 示例

程序运行结果如图 11-17 所示。

但这个例子存在了两个问题：

问题 1 是 ComboBox 的滚动条不能拖动，而是依靠其上下两个 RepeatButton 移动。

ComboBox 滚动条按钮按照内容滚动的，即 thumb 控件不能在滚动条上平滑地拖动，而是逐个 Item 跳动。解决的方法是将一个附加属性 ScrollViewer.CanContentScroll 设置为 false，如代码 11-12 所示。

```
<ComboBox Grid.Row="1" HorizontalAlignment="Stretch" VerticalAlignment="Top"
    ScrollViewer.CanContentScroll ="false">
```

代码 11-12 使 ComboBox 滚动条平滑滚动



图 11-17 程序运行结果

问题 2 是直接在输入 ComboBox 中如“段誉”这样的字段，然后是否可以自动选中下面的选项。

为解决这个问题，首先将 ComboBox 的 IsEditable 属性设置为 True，表示 ComboBox 允许输入字符串。但是这样又带来了另外一个问题，当选中一项后 ComboBox 中显示 Item 类型 System.Windows.Controls.StackPanel，未达到预期的效果，如图 11-18 所示。



图 11-18 将 IsEditable 属性设置为 true

问题的解决必须借助于 TextSearch 的两个附加属性 TextPath 和 Text。解决的方法有两种：一种是使用附加属性 TextPath。如代码 11-13 所示，仍然将 ComboBox 的 IsEditable 属性设置为 True。然后将 ComboBox 的每一项加上名字属性，如第 1 个 StackPanel 的 Name 属性为“段誉”，然后将 TextPath 设置为 Name 即可。

```
<ComboBox ..... IsEditable="True"
    TextSearch.TextPath="Name">
    <StackPanel Orientation="Horizontal" Margin="5" Name="段誉">
        ....
```

```
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="5" Name="萧峰">
    .....
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="5" Name="虚竹">
    .....
</StackPanel>
</ComboBox>
```

代码 11-13 使用附加属性 TextPath

这样选择每一项时，ComboBox 会显示 Name 属性的值。不仅如此，当用户在 ComboBox 中键入如“虚竹”一词时，也会自动选中名为“虚竹”的 StackPanel，如图 11-19 所示。



图 11-19 使用附加属性 TextPath

第 2 种方法更为简单，直接使用 Text 属性为每一个 StackPanel 添加一个 Text 属性。这种方法和第 1 种方法效果相同，如代码 11-14 所示。

```
<ComboBox IsEditable="True">
    <StackPanel Orientation="Horizontal" Margin="5" TextSearch.Text="段誉">
        .....
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5" TextSearch.Text="萧峰">
        .....
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5" TextSearch.Text="虚竹" >
        .....
    </StackPanel>
</ComboBox>
```

代码 11-14 使用附加属性 Text

3. Menu

菜单是一种非常常见的控件，如 VS 2010 中有一个“文件”（File）菜单项，其中的“打开”（Open）选项用来打开文件，如图 11-20 所示。

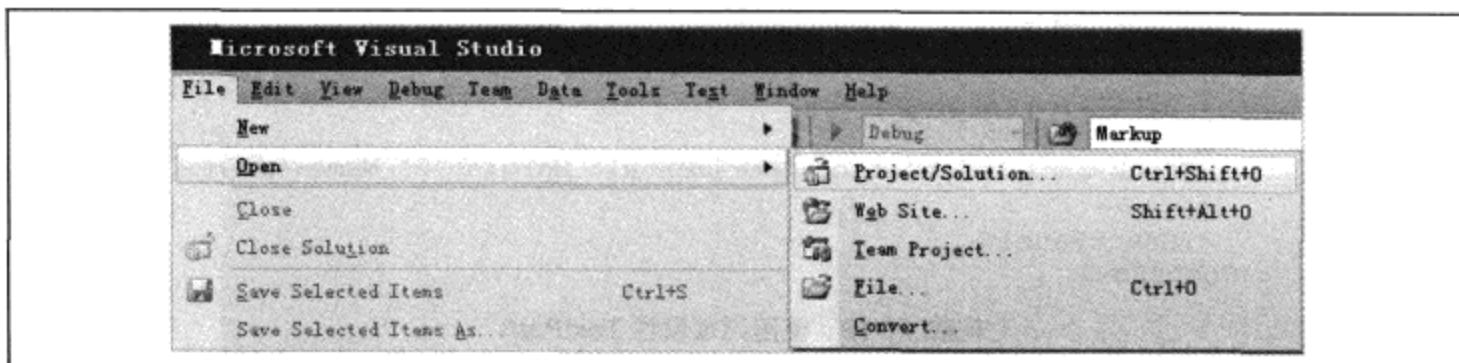


图 11-20 VS2010 中的菜单

工具栏中有一个打开按钮，如图 11-21 所示。

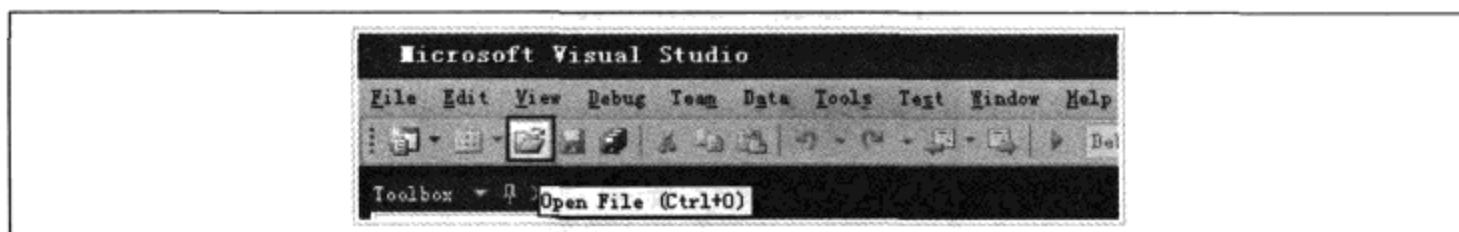


图 11-21 打开按钮

一个简单的写字板程序可以通过菜单中的功能项对文本框中的文本执行剪切、复制、字体放大和缩小，以及设置粗体和斜体等操作，也可以通过右键菜单执行剪切、复制和粘贴等功能（详见 `mumu_menudemo` 工程），如图 11-22 所示。

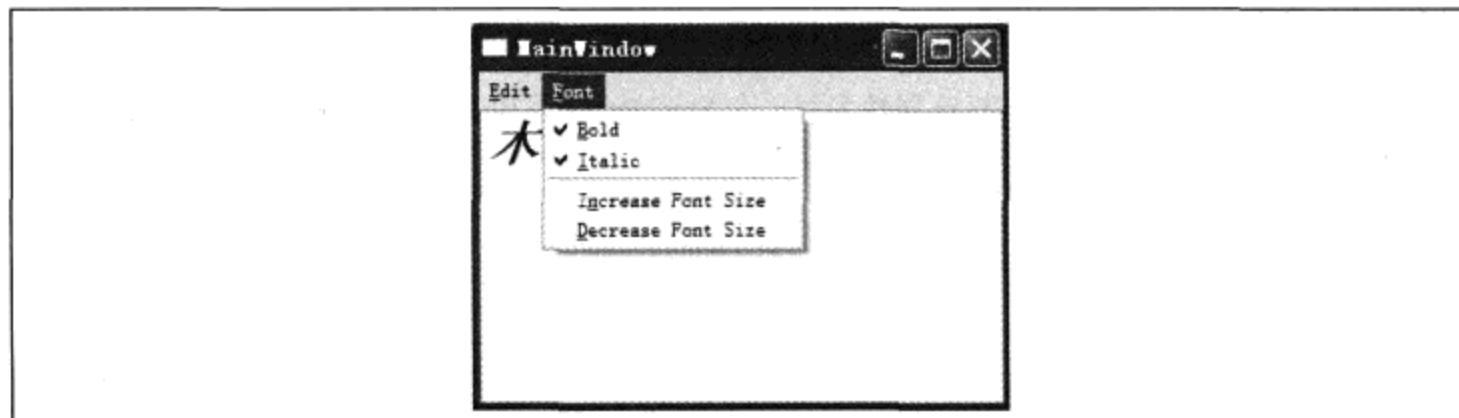


图 11-22 简单的写字板程序

其中 `MainWindow.xaml` 和 `MainWindow.xaml.cs` 文件如代码 11-15 所示，该例展示了菜单的几种典型用法。第 1 个“编辑”（Edit）菜单使用的 Command 命令，由于 `TextBox` 控件本身内置对 Command 的支持，因此子菜单项只需要设置相应的 Command 命令即可（代码①）。字体加粗（Bold）和斜体（Italic）两个子菜单设置 `IsCheckable` 属性为 `True`，这样具备了 `CheckBox` 控件的特性。当选中时在其左侧会有一个“√”符号，而且可以多项选择，如选择字体为粗斜体。菜单会相应提供两个事件，一个是 `Checked`；另一个是 `UnChecked` 事件（代码②），在代码文件中有相应的实现（代码⑤ 处）。字体放大和缩小菜单使用的是典型的 Click 事件（代码③），此外右键菜单添加的方式可以通过 `ContextMenu` 实现（代码④）。

MainWindow.xaml

```
<Window x:Class="mumu_menudemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Menu Grid.Row="0">
            ①             <MenuItem Header="_Edit">
                <MenuItem Command="ApplicationCommands.Copy"/>
                <MenuItem Command="ApplicationCommands.Cut"/>
                <MenuItem Command="ApplicationCommands.Paste"/>
            </MenuItem>
            ②             <MenuItem Header="_Font">
                <MenuItem Header="_Bold" IsCheckable="True"
                    Checked="Bold_Checked"
                    Unchecked="Bold_Unchecked"/>
                <MenuItem Header="_Italic" IsCheckable="True"
                    Checked="Italic_Checked"
                    Unchecked="Italic_Unchecked"/>
                <Separator/>
            ③             <MenuItem Header="I_ncrease Font Size"
                    Click="IncreaseFont_Click"/>
                <MenuItem Header="_Decrease Font Size"
                    Click="DecreaseFont_Click"/>
            </MenuItem>
        </Menu>
        <TextBox Name="textBox1" Grid.Row="1">
            <TextBox.ContextMenu>
                ④             <ContextMenu>
                    <MenuItem Command="ApplicationCommands.Copy"/>
                    <MenuItem Command="ApplicationCommands.Cut"/>
                    <MenuItem Command="ApplicationCommands.Paste"/>
                </ContextMenu>
            </TextBox.ContextMenu>
        </TextBox>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
namespace mumu_menudemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            textBox1.Text = "木木的菜单";
        }

    ⑤        private void Bold_Checked(object sender, RoutedEventArgs e)
        {
            textBox1.FontWeight = FontWeights.Bold;
        }
    }
```

```

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

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

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

private void IncreaseFont_Click(object sender, RoutedEventArgs e)
{
    if (textBox1.FontSize < 28)
    {
        textBox1.FontSize += 2;
    }
}

private void DecreaseFont_Click(object sender, RoutedEventArgs e)
{
    if (textBox1.FontSize > 10)
    {
        textBox1.FontSize -= 2;
    }
}
}

```

代码 11-15 MainWindow.xaml 和 MainWindow.xaml.cs 文件

11.3.4 Range 控件

本节介绍两种 Range 控件，即 ProgressBar 和 Slider。

1. ProgressBar

ProgressBar 主要用来表示操作进度，其重要属性有 Minimum、Maximum 和 Value。Minimum 表示该进度条的最小值；Maximum 表示该进度条的最大值；Value 表示进度条的当前值。当设置其 IsIndeterminate 属性为 true 时，进度条会以一定周期不停运转，这时不必设置 Minimum、Maximum 和 Value 属性（详见 mumu_ProgressBar 工程）。

代码 11-16 实现了两个进度条，其中第 1 个进度条的 IsIndeterminate 属性为 true，会不停运转（代码①）；第 2 个进度条在单击“开始”按钮时才会一步一步增加其进度（代码②和③），其中牵涉到动画的相关知识（参见第 16 章）。

MainWindow.xaml

```

<Window x:Class="mumu_ProgressBarDemo.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Window1" Height="200" Width="300"

```

```

        Background="LightBlue">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <StackPanel Grid.Row="0" VerticalAlignment="Center"
HorizontalAlignment="Center">
    ①           <ProgressBar Name="progressbar1" Height="20" Width="200"
Foreground="Red" IsIndeterminate="True"></ProgressBar>
        </StackPanel>
        <StackPanel Grid.Row="1" Orientation="Vertical" VerticalAlignment="Center"
HorizontalAlignment="Center">
            <ProgressBar Name="progressbar2" Height="20" Width="200"
Foreground="Blue"></ProgressBar>
    ②           <Button Name="btnBegin" MaxWidth="50" Margin="10"
Click="btnBegin_Click">开始</Button>
        </StackPanel>
    </Grid>
</Window>

MainWindow.xaml.cs
namespace mumu_ProgressBarDemo
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

    ③        private void btnBegin_Click(object sender, RoutedEventArgs e)
        {
            Duration duration = new Duration(TimeSpan.FromSeconds(10));
            DoubleAnimation doubleanimation = new DoubleAnimation(100.0,
duration);
            progressbar2.BeginAnimation(ProgressBar.ValueProperty,
doubleanimation);
        }
    }
}

```

代码 11-16 MainWindow.xaml 和 MainWindow.xaml.cs 文件

程序运行结果如图 11-23 所示。

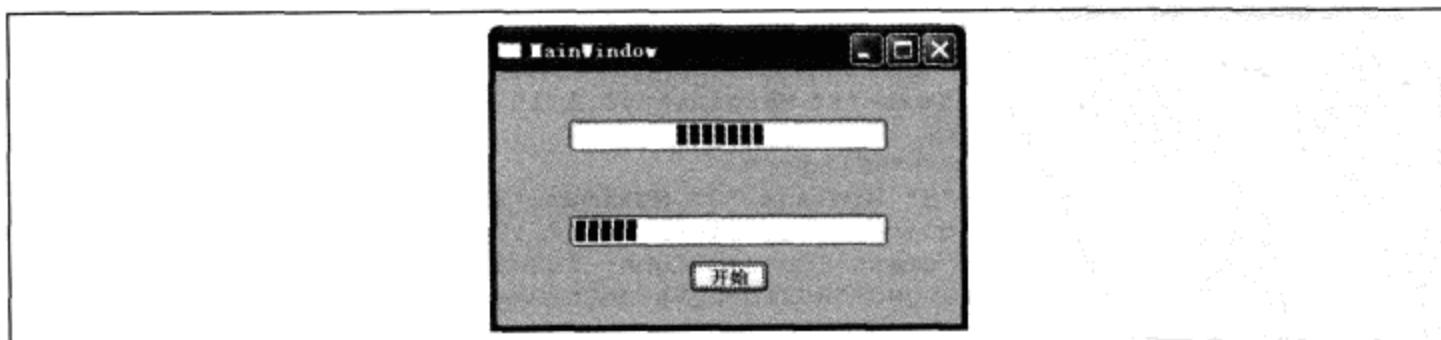


图 11-23 程序运行结果

2. Slider

Slider 是一种常见控件，用于调节音量等，其部分重要属性如表 11-2 所示。

表 11-2 Slider 控件的部分重要属性

属性	描述
Orientation	方向，包括水平和垂直
TickPlacement	获取或设置与 Slider 的 Track 相关刻度线的位置
TickFrequency	获取或设置刻度线之间的间隔
Ticks	获取或设置为 Slider 显示的刻度线的位置
IsSnapToTickEnabled	获取或设置一个值，该值指示 Slider 是否自动将 Thumb 移动到最近的刻度线
IsSelectionRangeEnabled	获取或设置一个值，该值指示 Slider 是否沿 Slider 显示选择范围，这是一个依赖属性

下例通过 Slider 实现一个颜色调整对话框（详见 `mumu_SliderDemo` 工程），如图 11-24 所示。

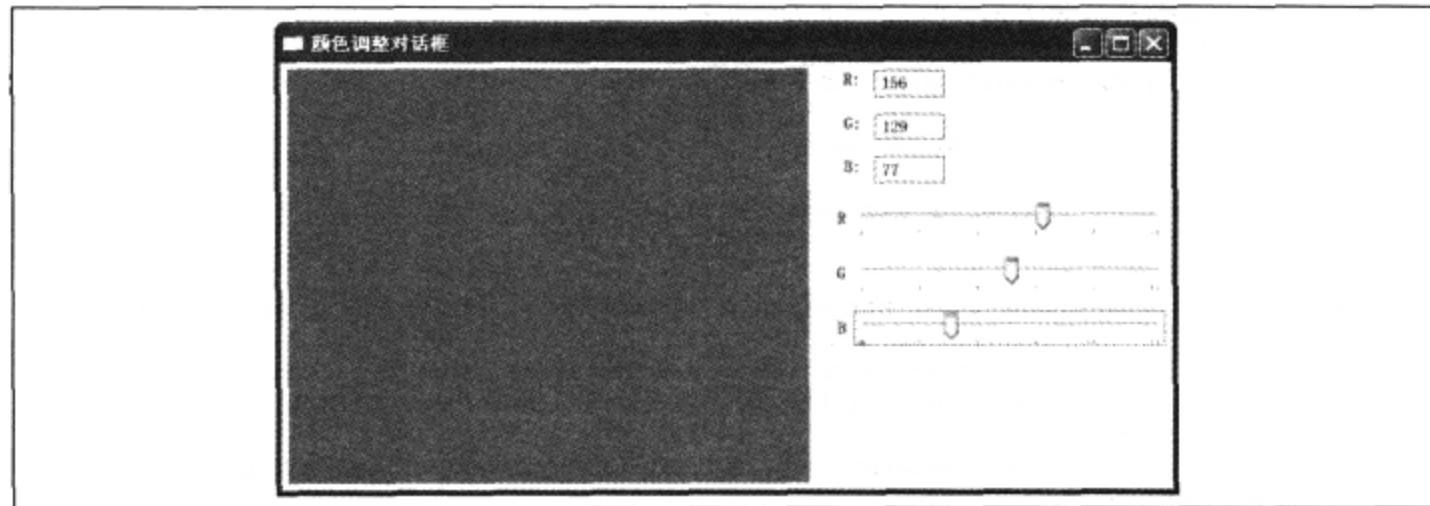


图 11-24 通过 Slider 实现的颜色调整对话框

Slider 提供了一个 `ValueChanged` 事件，该程序主要通过其处理函数改变矩形框的颜色，如代码 11-17 所示。

```
<Rectangle x:Name="rec" Fill="Black">
</Rectangle>
.....
<StackPanel Orientation="Horizontal" Margin="10,2,5,2">
<TextBlock Text="R" Margin="5,1,1,1" VerticalAlignment="Center">
</TextBlock>
<Slider Name="RSlider"
Margin="5" Minimum="0" Maximum="255"
TickFrequency="10" Ticks="0,50,100,150,200,250"
TickPlacement="BottomRight" IsSnapToTickEnabled="False"
ValueChanged="RSlider_ValueChanged" MinWidth="220">
</Slider>
</StackPanel>

<StackPanel Orientation="Horizontal" Margin="10,2,5,2">
```

```

</TextBlock>
    <TextBlock Text="G" Margin="5,1,1,1" VerticalAlignment="Center">
        <Slider Name="GSlider"
            Margin="5" Minimum="0" Maximum="255"
            TickFrequency="10" Ticks="0,50,100,150,200,250"
            TickPlacement="BottomRight" IsSnapToTickEnabled="False"
            ValueChanged="GSlider_ValueChanged" MinWidth="220">
        </Slider>
    </StackPanel>

    <StackPanel Orientation="Horizontal" Margin="10,2,5,2">
        <TextBlock Text="B" Margin="5,1,1,1" VerticalAlignment="Center">
        </TextBlock>
        <Slider Name="BSlider"
            Margin="5" Minimum="0" Maximum="255"
            TickFrequency="10" Ticks="0,50,100,150,200,250"
            TickPlacement="BottomRight" IsSnapToTickEnabled="False"
            IsSelectionRangeEnabled="True"
            ValueChanged="BSlider_ValueChanged" MinWidth="220">
        </Slider>
    </StackPanel>

    private void BSlider_ValueChanged(object sender,
RoutedEventArgs e)
{
    Update();
}

private void GSlider_ValueChanged(object sender,
RoutedEventArgs e)
{
    Update();
}

private void RSlider_ValueChanged(object sender,
RoutedEventArgs e)
{
    Update();
}

private void Update()
{
    color = Color.FromRgb(Convert.ToByte(RSlider.Value),
Convert.ToByte(GSlider.Value), Convert.ToByte(BSlider.Value));
    rec.Fill = new SolidColorBrush(color);
}
}
}

```

代码 11-17 通过 ValueChanged 事件改变矩形框的背景色

为了使 Slider 的值能够在文本框中连续显示，需要通过绑定来实现。同时需要自定义一个类型转换器，负责将颜色的 RGB 值转换为字符串（参见第 14 章），如代码 11-18 和代码 11-19 所示。

```

<Window.Resources>
    <local:DoubleConverter x:Key="Converter"/>
</Window.Resources>
<StackPanel Orientation="Horizontal" Margin="10,5,5,5">
    <TextBlock Text="R:" Margin="10,1,5,1"></TextBlock>
    <TextBox Name="txtR" Margin="5,1,5,1" MinWidth="50"

```

```

        Text="{Binding ElementName=RSlider,Path=Value,
                    Converter={StaticResource Converter},Mode=TwoWay}">
    </TextBox>
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="10,5,5,5">
    <TextBlock Text="G:" Margin="10,1,5,1"></TextBlock>
    <TextBox Name="txtG" Margin="5,1,5,1" MinWidth="50"
        Text="{Binding ElementName=GSlider,Path=Value,
                    Converter={StaticResource Converter},Mode=TwoWay}"></TextBox>
</StackPanel>
<StackPanel Orientation="Horizontal" Margin="10,5,5,5">
    <TextBlock Text="B:" Margin="10,1,5,1"></TextBlock>
    <TextBox Name="txtB" Margin="5,1,5,1" MinWidth="50"
        Text="{Binding ElementName=BSlider,Path=Value,
                    Converter={StaticResource Converter},Mode=TwoWay}"></TextBox>
</StackPanel>

```

代码 11-18 MainWindow.xaml 文件

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Data;

namespace mumu_SliderDemo
{
    public class DoubleConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            Byte rgbValue = System.Convert.ToByte(value) ;
            string txtRGBValue = rgbValue.ToString();
            return txtRGBValue;
        }
        public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            string str = (string)value;
            double dbValue = double.Parse(str);
            return dbValue;
        }
    }
}

```

代码 11-19 DoubleConverter.cs 文件

11.4 接下来做什么

本章的关键是 Content 模型，对于控件的理解还需要迈过几道坎才算得上修成正果。第一道坎即是 Content 模型；第二道坎则是样式和模板（第 13 章）；第三道坎是数据绑定（第 14 章）；而到最后能够信手拈来自定义一个控件，才是真正的对控件的理解（第 20 章）。

目前已经有一本名为《WPF Control Development UNLEASHED》的书专门讨论 WPF 中的控件，目前尚未见到中译本。有兴趣的读者可以阅读电子版，其中的示例相当不错。

参考文献

- [1] “北风之神”，风清远整理，“云中孤雁”制作《金庸全集典藏版，天龙八部》，“第二章 玉壁月华明”。
- [2] Chris Anderson 著，《Essential Windows Presentation Foundation》。
- [3] MSDN Library for Visual Studio 2008 SP1 Menu with Commands and Events Sample。
- [4] Matthew MacDonald 著，王德才译，《WPF 编程宝典——使用 C# 2008 和.NET 3.5（第 2 版）》，英文名为“Pro WPF in C#2008 Windows Presentation Foundation with .NET 3.5 Second Edition”。

第四卷

峰回路转
夯实基础



资源——雪山宝藏

霍青桐赠送短剑之时，曾说故老相传，剑中蕴藏着一个极大秘密，一向无人参透得出。今日若非机缘巧合，巨狼死命咬住，两下用力拉扯，才拔出了第二层剑鞘，否则有谁想得到这柄锋利的短剑之中，竟是剑内有剑？

……陈家洛手指微一用劲，蜡丸破裂。里面是个小纸团，摊开纸团，却是一张薄如蝉翼的纱纸。纸上写着许多字，都是古文回字，旁边是一张地图，画得密如蛛网。

——《书剑恩仇录》，“第十六回 我见犹怜二老意 谁能遣此双姝情”^[1]

看过《书剑恩仇录》的读者都知道，这一段讲的是陈家洛、霍青桐和香香公主同张召重等其他坏人一块深陷狼群，无意中发现了短剑里居然隐藏着雪山宝藏的藏宝图。

资源给木木的第一感觉就好像那个雪山宝藏，它可以是数据文件、用户界面，或者是一个简单的画刷等。类型多样，取之不竭。

WPF 的资源有两种，一种称为“程序集资源”（assembly resources）或者“二进制资源”（binary resources），在 MSDN 中将其称为“应用程序数据文件”（application data files）；另外一种 WPF 中称为“资源”或者“对象资源”（object resources）、“逻辑资源”（logical resources），甚至“声明式资源”（declarative resources）。本书统一将前者称为“程序集资源”，将后者称为“逻辑资源”。

在 WPF 中要成功地获取资源，必须掌握一套统一资源标识（URI，Uniform Resource Identifiers）。URI 便是我们打开雪山宝藏的金钥匙。

本章包括如下几节：

- (1) 程序集资源。
- (2) URI 语法。
- (3) 逻辑资源。
- (4) 接下来做什么。

12.1 程序集资源

应用程序中常常依赖一些 XAML、图片、音频和视频等文件，可以将其作为程序集资源组织起来。

程序集资源可以以如下3种方式打包。

- (1) 资源文件 (Resource File)：直接嵌入到程序集中。
- (2) 内容文件 (Content File)：该文件的相关信息会编译到程序集中，如文件的相对位置等。
- (3) Site of Origin 文件：不参加编译，应用程序不知道该文件是否存在。

这三个文件倒是可以打个不错的比喻。这资源文件就好比老婆，那是要随时在高官身边的，出入各种公开场合。内容文件就好比高官的情人，高官掌握了情人的相关信息，不能随时在身边，但是总能随时找到。这 Site of Origin 文件，就是各种偶然了，各种的不期相遇，各种的萍水相逢了……

12.1.1 资源文件

资源文件是直接打包到程序集当中的。我们不妨和木木一块看个例子。新建一个“mumu_resouceDemo”工程。在其中新建一个目录，名为“image”，将系统的图片复制到该目录下。选择该目录中的文件，可以查看到文件的Build Action选项均为Resource，如图 12-1 所示。

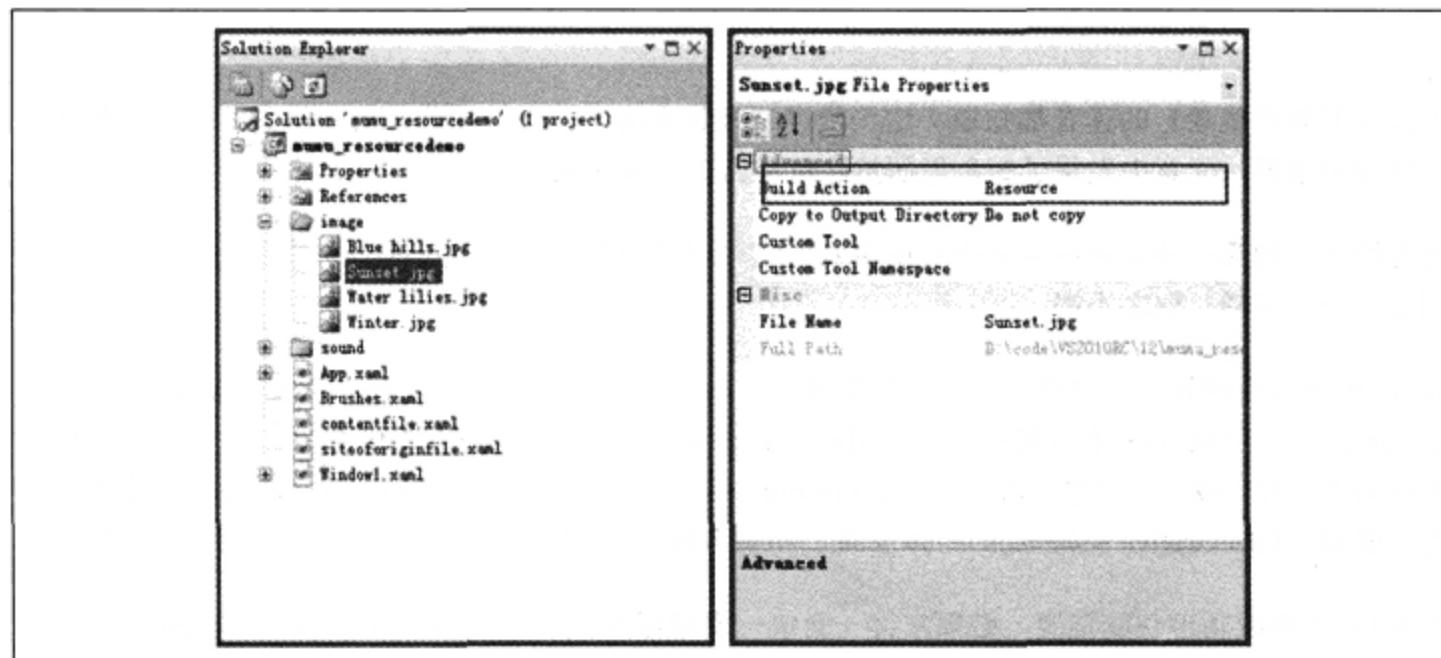


图 12-1 工程的文件结构及其文件属性

现在4张图片均为该工程的资源文件，MainWindow.xaml文件的内容如代码 12-1 所示。

```
<Window x:Class="mumu_resourcedemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ResourceDemo" Height="300" Width="300" Loaded="Window_Loaded">
<StackPanel>
    <GroupBox Header="Pack URIs">
        <StackPanel>
            <Image Source="pack://application:,,,/image/Sunset.jpg"/>
        </StackPanel>
    </GroupBox>
</StackPanel>
```

```
</Window>
```

代码 12-1 MainWindow.xaml 文件

加粗部分为引用该资源文件的 XAML 语句，运行该程序，结果如图 12-2 所示。



图 12-2 运行结果

木木看完这个例子，大致明白了什么是资源文件，但是存在如下疑问。



疑问之一：属性页中 Build Action 中有一个 Embedded Resource 和 Resource 有什么区别？

解答：Embedded Resource 和 Resource 很容易混淆，二者都会在程序集中嵌入一个程序集资源。前者用于在 WinForm 项目中嵌入程序集资源，在 WPF 中则选择 Resource。

疑问之二：引用资源的“pack://application:,,,/image/Sunset.jpg”语句是什么含义？

解答：这种字符串是按照 URI 的规范来写的，在下节将要系统介绍。我们也可以用代码来访问这个图片资源，如代码 12-2 所示。

```
Uri uri = new Uri("/image/Sunset.jpg", UriKind.Relative);
StreamResourceInfo info =
Application.GetResourceStream(uri);
BitmapImage bitmapimg = new BitmapImage();
// 注意为 BitmapImage 赋值需要调用 BeginInit 和 EndInit 两个函数
bitmapimg.BeginInit();
bitmapimg.StreamSource = info.Stream;
bitmapimg.EndInit();
img.Source = bitmapimg;
```

代码 12-2 用代码引用资源

也可以直接指定 BitmapImage 的 Uri，如代码 12-3 所示。

```
img.Source = new BitmapImage(new Uri("/image/Sunset.jpg",
UriKind.Relative));
```

代码 12-3 用代码引用资源



疑问之三：我怎么知道这些资源文件被编译到程序集中？

解答：这些资源文件均会被编译到一个名为“工程名.g.resources”文件中，本例是 mumu_resourcedemo.g.resources 文件（\obj\Debug 目录下）。

如果换作别人，这一节就到此结束了。但是谁要木木是本书的主角呢？他看到疑问之三的解答，于是尝试用记事本程序打开 `mumu_resourcedemo.g.resources` 文件，结果失败！于是他产生了第四个疑问。



疑问之四：我怎么能看到这个资源文件的内容，从而证明这些资源文件都被编译到该程序集中？

解答：要查看这个资源文件必须借助 Reflector 工具，可以看到在 Resources 文件夹下 `mumu_resourcedemo.g.resources` 文件中包含的图片，包括 `MainWindow.xaml` 文件也同样在资源集合中。

图 12-3 所示为通过 Reflector 查看到的 `mumu_resourcedemo.g.resources` 文件中包含的内容。

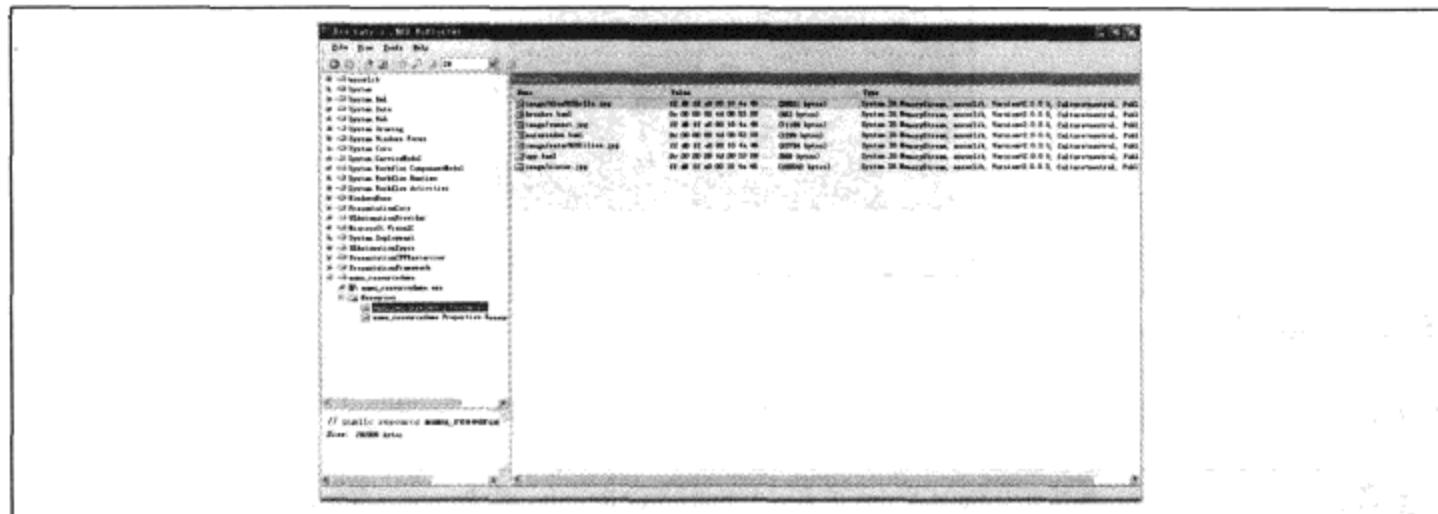


图 12-3 通过 Reflector 工具查看到的内容

12.1.2 内容文件

内容文件并不打包到程序集当中，但是内容文件还算是在应用程序的掌控之中。因为内容文件的相关信息会被编译到应用程序当中。在上面的工程中新建一个名为“sound”的文件夹，在其中添加一个系统的音频文件 `chimes.wav`。选中该文件，可以看到 VS 将该文件的 Build Action 选项设置为 Content 属性。将文件的另一个属性 Copy to Output Directory 设置为 Copy always 或者 Copy If newer，方可将其复制到 `bin\debug` 目录下。只不过前者每次编译均复制；后者则在文件更新时复制，如图 12-4 所示。

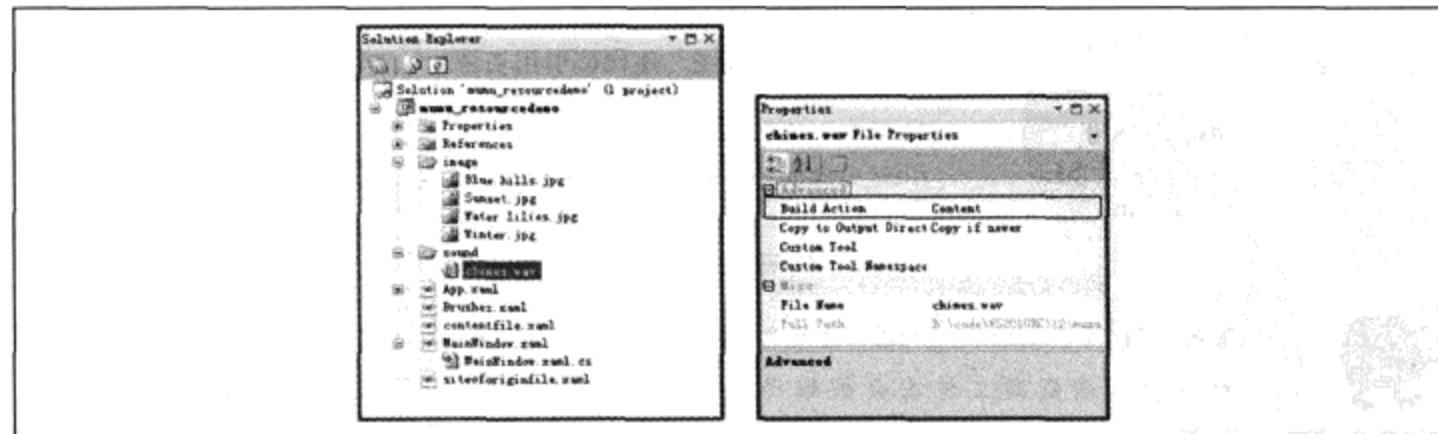


图 12-4 `chimes.wav` 文件及其属性

这个声音文件主要用来当触发按钮单击事件时播发声音，如代码 12-4 所示。

```
<Button Content="Play chimes.wav">
    <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
            <EventTrigger.Actions>
                <SoundPlayerAction Source ="\\sound\\chimes.wav"/>
            </EventTrigger.Actions>
        </EventTrigger>
    </Button.Triggers>
</Button>
```

代码 12-4 MainWindow.xaml 文件部分代码

也可以用代码来访问 Content 文件，在这个工程中添加一个名为“ContentFile.xaml”的文件。将其 Build Action 属性设置为 Content，Copy to Output Directory 设置为 Copy always，如代码 12-5 所示。

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <TextBlock>在应用程序中的内容文件</TextBlock>
</Page>
```

代码 12-5 contentFile.xaml 文件

访问内容文件使用 GetContentStream 方法，如代码 12-6 所示。

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // 1 引用资源文件 Sunset.jpg
    .....
    // 2 引用内容文件 contentfile.xaml
    Uri uri = new Uri("/contentFile.xaml", UriKind.Relative);
    StreamResourceInfo info = Application.GetContentStream(uri);
    System.Windows.Markup.XamlReader reader = new
System.Windows.Markup.XamlReader();
    Page page = (Page)reader.LoadAsync(info.Stream);
    this.contentfileframe.Content = page;
}
```

代码 12-6 在代码里访问 Content 文件

代码 12-7 更为简练。

```
this.contentfileframe.Source = new Uri("/contentFile.xaml", UriKind.Relative);
```

代码 12-7 在代码里访问 Content 文件

这个 contentFile 也可以用 XAML 引用，如代码 12-8 所示。

```
<Frame Source="pack://application:,,,/contentfile.xaml"/>
```

代码 12-8 在 XAML 里访问 Content 文件

木木基本上理解了内容文件，但是也存在一个疑问。



疑问：前面说了多次内容文件对应用程序是已知的，即内容文件的相关信息被编译到应用程序中，但是从何处可以看出来？

解答：实际上当一个工程中包含了内容文件，系统编译时会产生一个“工程名_Content.g.cs”文件（在\obj\Debug 目录下，对应该工程的文件为 mumu_resourceDemo_Content.g.cs），该文件用 AssemblyAssociatedContentFileAttribute 将内容文件的元数据信息编译到程序集中。

代码 12-9 所示为 mumu_resourceDemo_Content.g.cs 文件的代码。

```
//-----
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:4.0.30128.1
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//-----
[assembly: System.Windows.Resources.AssemblyAssociatedContent
FileAttribute("contentfile.xaml")]
[assembly: System.Windows.Resources.AssemblyAssociatedContent
FileAttribute("sound/chimes.wav")]
```

代码 12-9 用 AssemblyAssociatedContentFileAttribute 编译内容文件

12.1.3 Site of Origin 文件

Site Of Origin 文件根本就不参加编译。换句话说，应用程序在编译时根本不知道 Site of Origin 文件的存在，只有运行时才知道，如代码 12-10 所示。

```
<Frame Source="http://www.cnblogs.com/helloj2ee/" />
<Image Source="file:///C:/DataFile.bmp" />
```

代码 12-10 在 XAML 里访问 Site Of Origin 文件

配置 Site of Origin 文件需要把 Build Action 属性设置为 None，Copy to Output Directory 设置为 Copy always 或者是 Copy If newer。通过代码访问则需要使用 GetRemoteStream 方法，如代码 12-11 所示。

```
Uri uri = new Uri("/siteoforiginfile.xaml", UriKind.Relative);
StreamResourceInfo info = Application.GetRemoteStream(uri);
System.Windows.Markup.XamlReader reader = new System.Windows.Markup.XamlReader();
Page page = (Page)reader.LoadAsync(info.Stream);
this.siteoforiginframe.Content = page;
```

代码 12-11 代码访问 Site of Origin 文件

12.2 URI 语法

早在 WPF 之前，Web 上可用的每种资源——HTML 文档、图像、视频片段、程序等——都是由 URI 来进行定位的。一个 URI 一般由如下 3 个部分组成。

(1) 协议，也称为“服务方式”。

(2) 该资源的主机 IP 地址，有时也包括端口号。

(3) 主机资源的地址，如目录和文件名等。

如 <http://www.cnblogs.com/helloj2ee/> 表示一个可通过 HTTP 协议访问的资源，位于主机 www.cnblogs.com，通过路径/helloj2ee/访问。

12.2.1 WPF 中的 URI

在 WPF 中通过 URI 来标识和访问资源，包括如下情况。

(1) 当应用程序启动时设置需要显示的用户界面。

(2) 装载图像。

(3) 页面之间的导航。

(4) 装载资源、内容和 Site of Origin 文件。

URI 可以通过当前程序集、被引用的程序集、相对程序集的一个位置和任意位置标识访问资源。

WPF 中的 URI 同样由 3 个部分组成，第 1 个部分的协议是 pack；第 2 个部分称为“authority”，有两种值：一是 application:///，表示编译时知道的文件，主要指资源和内容文件；二是 siteoforigin:///，表示 Site of Origin 文件；第 3 个部分是路径，如果是引用程序集中的资源，情况会稍微复杂一些。路径必须包含引用的程序集名称和一个 Component 标识，表示引用的不是本地程序集的资源，有时还需要加上版本信息。

WPF 中的路径分为相对和绝对路径，如代码 12-12 所示，前者是一个绝对路径；后者则是一个相对路径。WPF 中的默认 URI 设置是 pack://application:,,,，因此引用 site of origin 文件必须要用绝对路径。

```
pack://application:,,,/ResourceFile.xaml  
/ResourceFile.xaml
```

代码 12-12 WPF 中的路径

12.2.2 一个全面的 URI 用法示例

MSDN 中一个示例基本上涵盖了所有 URI 的用法，该示例包括两个项目：一是 WPFPackUriSample，相当于本地程序集；二是 ReferencedAssembly，相当于引用程序集。本地程序集中包含如前所述的 3 种不同类型的文件，而引用程序集中仅仅包含资源文件，这些文件分别在工程的根目录和一个 subfolder 目录下各放置一个副本，如图 12-5 所示。

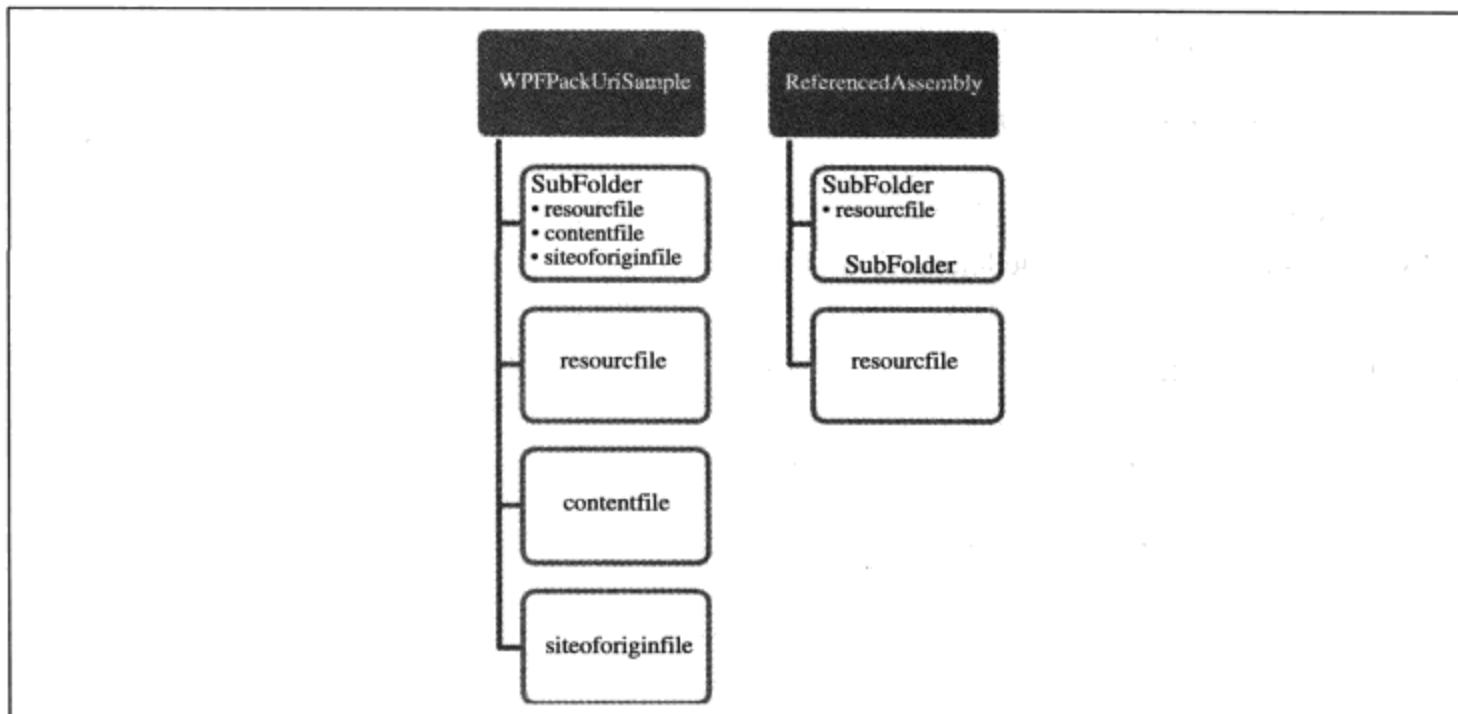


图 12-5 示例的组成结构

在工程的 MainWindow.xaml 文件中分别以相对和绝对路径方式访问这些资源文件，如代码 12-13 所示。

```

MainWindow.xaml
<Window x:Class="WPFPackUriSample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Pack Uri Sample" Height="450" Width="350"
    >
    <DockPanel>
        <StackPanel DockPanel.Dock="Top">

            <GroupBox Header="Absolute Pack URIs">
                <StackPanel>
                    <Frame Source="pack://application:,,,/ResourceFile.xaml" />
                    <Frame Source="pack://application:,,,/Subfolder/ResourceFile.xaml" />
                    <Frame Source="pack://application:,,,/ReferencedAssembly;component/
ResourceFile.xaml" />
                    <Frame Source="pack://application:,,,/ReferencedAssembly;component/
Subfolder/ResourceFile.xaml" />
                    <Frame Source="pack://application:,,,/ReferencedAssembly;v1.0.0.1;component/
ResourceFile.xaml" />
                    <Frame Source="pack://application:,,,/ContentFile.xaml" />
                    <Frame Source="pack://application:,,,/Subfolder/ContentFile.xaml" />
                    <Frame Source="pack://siteoforigin:,,,/SiteOfOriginFile.xaml" />
                    <Frame Source="pack://siteoforigin:,,,/Subfolder/SiteOfOriginFile.xaml" />
                </StackPanel>
            </GroupBox>

            <GroupBox Header="Relative Pack URIs">
                <StackPanel>
                    <Frame Source="/ResourceFile.xaml" />
                    <Frame Source="/Subfolder/ResourceFile.xaml" />
                    <Frame Source="/ReferencedAssembly;component/ResourceFile.xaml" />
                    <Frame Source="/ReferencedAssembly;component/Subfolder/ResourceFile.xaml" />
                    <Frame Source="/ReferencedAssembly;v1.0.0.1;component/ResourceFile.xaml" />
                </StackPanel>
            </GroupBox>
        </StackPanel>
    </DockPanel>

```

```

<Frame Source="/ContentFile.xaml" />
<Frame Source="/Subfolder/ContentFile.xaml" />
<Frame Source="pack://siteoforigin:,,,/SiteOfFile.xaml" />
<Frame Source="pack://siteoforigin:,,,/Subfolder/SiteOfFile.xaml" />
</StackPanel>
</GroupBox>

</StackPanel>
<GroupBox Header="Version-Specified Pack URIs">
    <DockPanel>
        <Button DockPanel.Dock="Top" Click="click0">Get Resource File in Reference
Assembly v1.0.0.0</Button>
        <Button DockPanel.Dock="Top" Click="click1">Get Resource File in
Reference Assembly v1.0.0.1</Button>
        <Frame Name="frame" NavigationUIVisibility="Hidden" />
    </DockPanel>
</GroupBox>
</DockPanel>
</Window>

```

代码 12-13 MainWindow.xaml

针对同一个程序集的不同版本的 URI 设置方式如代码 12-14 所示。

```

MainWindow.xaml.cs
void click0(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri("/VersionedReferencedAssembly;v1.0.0.0;component/
ResourceFile.xaml", UriKind.RelativeOrAbsolute);
    this.frame.Source = uri;
}
void click1(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri("/VersionedReferencedAssembly;v1.0.0.1;component/
ResourceFile.xaml", UriKind.RelativeOrAbsolute);
    this.frame.Source = uri;
}

```

代码 12-14 MainWindow.xaml.cs



疑问： WPF 中的 URI 中的 3 个逗号从何而来？

WPF 程序中规定 application:/// 和 siteoforigin:/// 的反斜杠号都要替换为逗号，因此 URI 中的 3 个反斜杠号替换而来。

12.2.3 WPF 中的 URI 处理顺序

在 WPF 程序中只有两种 URI 处理系统，即 application:/// 和 siteoforigin:///。对于前者，WPF 程序会按照提供的路径查找 SiteofOrigin 文件；对于后者，WPF 遵循如下处理顺序。

- (1) 在程序集的 AssemblyAssociatedContentFileAttribute 属性的元数据中查找，如果 URI 的路径匹配正确，则该资源是一个内容文件。
- (2) 如果未找到，则继续在程序集的资源中查找；如果找到，则该资源是一个资源文件。
- (3) 如果未找到，则该 URI 无效。

12.3 逻辑资源

逻辑资源是 WPF 程序中的特有概念，它是一些保存在元素 Resources 属性中的.NET 对象，通常需要共享给多个子元素。

还是通过例子来了解逻辑资源。新建一个名为“mumu_logicresourcedemo”的 WPF 应用程序，添加一个 happyface.jpg 的图像资源文件。在窗口的 Resources 属性中新建一个画刷资源，并将其应用到按钮的背景色中，如代码 12-15 所示。

```
<Window x:Class="mumu_logicresourcedemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="逻辑资源 Demo" Height="300" Width="300">
    <Window.Resources>
        <ImageBrush x:Key="TileBrush" TileMode="Tile"
            ViewportUnits="Absolute" Viewport="0 0 32 32"
            ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
    </Window.Resources>
    <Grid>
        <Button Background="{StaticResource TileBrush}" Padding="5"
            FontWeight="Bold" FontSize="14" Margin="5">A Tiled Button</Button>
    </Grid>
</Window>
```

代码 12-15 MainWindow.xaml

在 WPF 中 Application、FrameworkElement 和 FrameworkContentElement 等基类均包含 Resources 属性，因此大部分 WPF 类都有这个属性。该属性实际上是一个资源集合的容器，每个资源需要用 x:key 关键字来唯一标识。定义在窗口中的资源可以在整个窗口内使用，定义在应用程序内的资源可以在整个应用程序中使用。我们仍通过标记扩展的方式来引用资源，如上例中的 Background="{StaticResource TileBrush}"。运行程序，结果如图 12-6 所示。

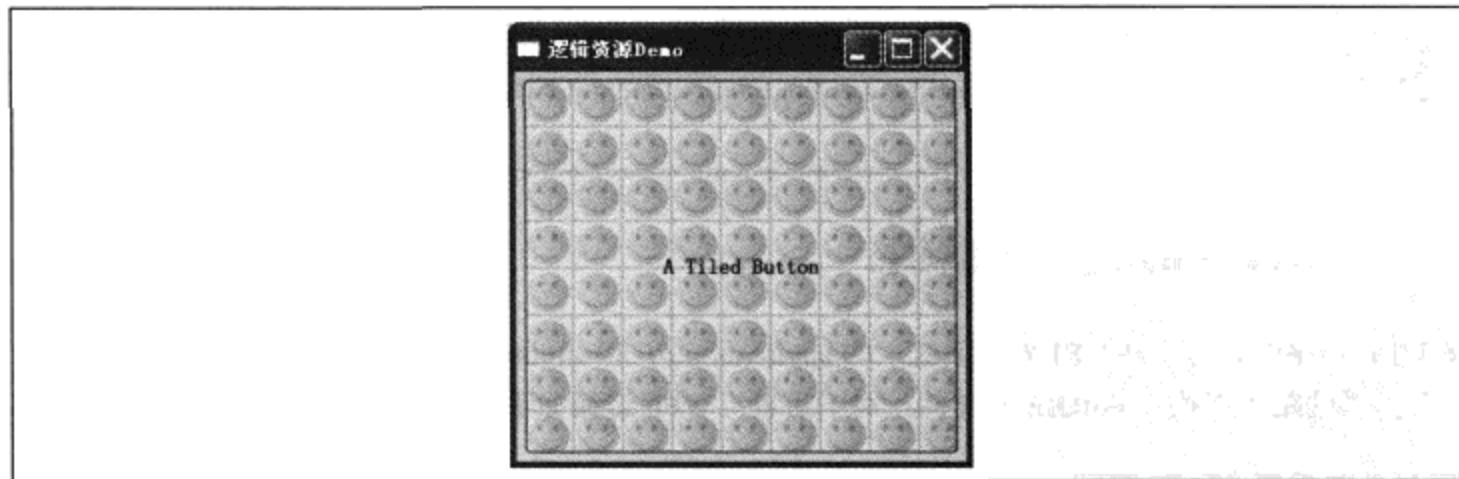


图 12-6 程序运行结果

不仅仅是画刷这样典型的 WPF 对象，一般普通的.NET 对象均可作为资源。如果需要添加一个字符串对象作为资源，则首先声明 system 命名空间。在原 XAML 文件中添加如下加粗的一行，如代码 12-16 所示。

```
<Window x:Class="mumu_logicresourcedemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:s = "clr-namespace:System;assembly=mscorlib"
    ....>
```

代码 12-16 在 XAML 文件中声明 system 命名空间

在 Window 的资源标签页中添加一个字符串资源，如代码 12-17 所示。

```
<Window.Resources>
    <ImageBrush x:Key="TileBrush" TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
    <s:String x:Key="ButtonContent">
        字符串作为逻辑资源
    </s:String>
</Window.Resources>
```

代码 12-17 添加一个字符串资源

使用标记扩展来引用该资源，如代码 12-18 所示。

```
<Button Grid.Row="1" Margin="5" Content="{StaticResource ButtonContent}" />
```

代码 12-18 标记扩展引用字符串资源

12.3.1 静态资源和动态资源

WPF 提供两种访问逻辑资源的方式：一是静态资源，通过 StaticResource 标记扩展来实现，前面引用资源的方式均为这种方式；二是动态资源，通过 DynamicResource 标记扩展来实现。

静态资源和动态资源标记扩展的主要区别在于前者只从资源字典中查找一次资源，后者在应用程序需要时查找资源。

在前面例子中 MainWindow.xaml 文件里添加两个按钮，一个按钮采用静态资源的方式引用 TileBrush 资源；另一个采用动态资源的方式引用，如代码 12-19 所示。

```
<Button Background="{StaticResource TileBrush}" Padding="5"
    FontWeight="Bold" FontSize="14" Margin="5" Click="Button_Click">A
Tiled Button 静态资源引用</Button>
<Button Grid.Row="1" Background="{DynamicResource TileBrush}" Padding="5"
    FontWeight="Bold" FontSize="14" Margin="5" Click="Button_Click">A
Tiled Button 动态资源引用</Button>
```

代码 12-19 采用两种方式引用资源

在按钮 Click 事件中编写如代码 12-20 所示。按照木木的设想，静态资源引用的按钮不会发生改变，而动态资源引用的按钮会改变。

```
ImageBrush brush = (ImageBrush)this.Resources["TileBrush"];
brush.Viewport = new Rect(0, 0, 5, 5);
```

代码 12-20 第一次尝试

但是运行程序，静态和动态资源引用按钮全部发生变化，如图 12-7 所示。

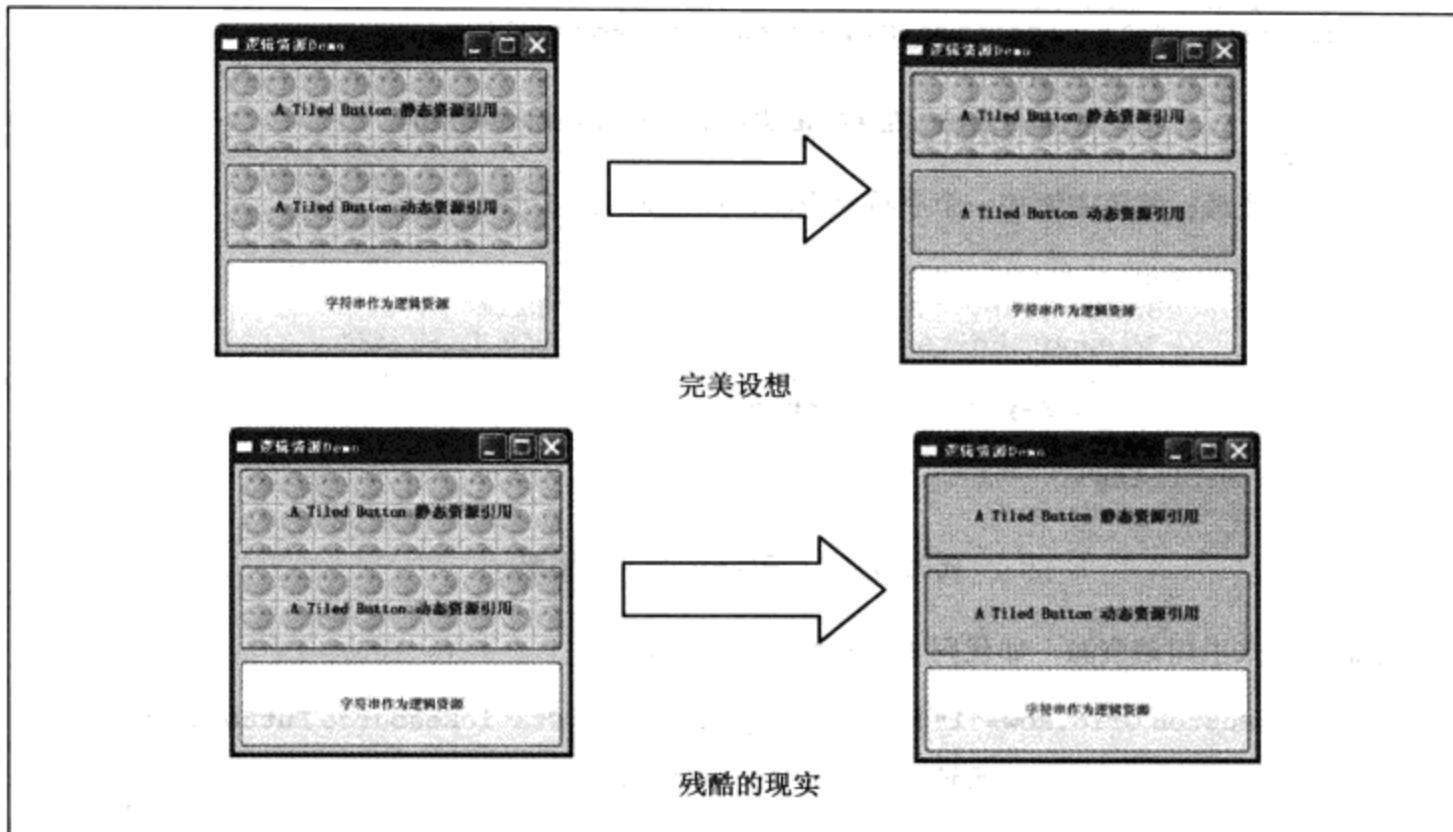


图 12-7 预期和现实的结果

前面是画刷资源，我们再看看如果是字符串资源，结果会如何？第 3 个按钮的 Content 属性引用一个字符串的资源，修改后添加一个按钮，如代码 12-21 和代码 12-22 所示。

```
MainWindow.xaml
<Button Grid.Row="2" Margin="5" Content="{StaticResource
ButtonContent}" Click="Button_Click_1" />
<Button Grid.Row="3" Margin="5" Content="{DynamicResource
ButtonContent}" Click="Button_Click_1" />
```

代码 12-21 添加一个按钮

```
MainWindow.xaml.cs
String str = (String)this.Resources["ButtonContent"];
str = "木木的再一次尝试";
```

代码 12-22 再一次尝试

运行结果是引用字符串资源的按钮内容均不变化。说明这种变化与静态和动态资源无关。真正的原因为在于画刷资源继承自 `Freezable` 类。该类本身提供了一种跟踪变化的基本机制，因此两个按钮均发生变化。而字符串资源是一种基本的.NET 类型，不具备这种跟踪机制，因此两个按钮不发生变化。

应该如何修改才能达到原来预期的结果呢？木木思考了很久写下了如下代码，这一次他不是鲁莽地改变画刷的属性，而是替换资源字典里的资源，如代码 12-23 所示。

```
// 改变画刷资源的代码
ImageBrush brush = (ImageBrush)this.Resources["TileBrush"];
ImageBrush newbrush = brush.Clone();
```

```
newbrush.Viewport = new Rect(0, 0, 5, 5);
this.Resources["TileBrush"] = newbrush;
// 改变字符串资源的代码
this.Resources["ButtonContent"] = "木木的再一次尝试";
```

代码 12-23 成功尝试

运行程序，结果如图 12-8 所示。

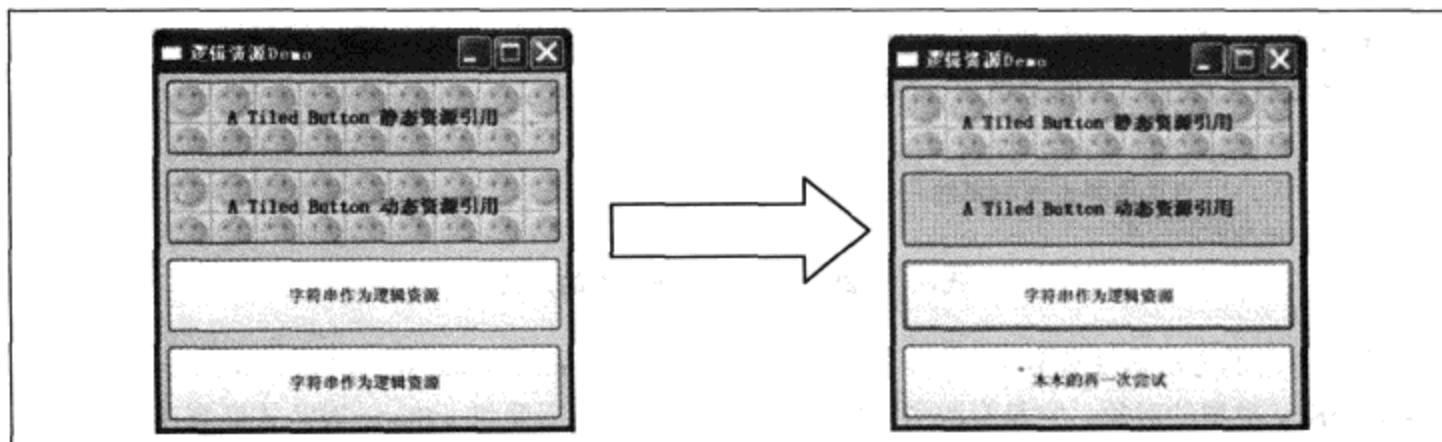


图 12-8 运行结果

静态和动态资源的区别总结如下。

- (1) 静态资源只有一次机会从资源字典中查找资源，一般是加载窗口或者页面时；动态资源标记扩展在应用程序每次需要时查找资源。
- (2) 由于需要跟踪变化，动态资源需要占用更多的资源，因此过多地使用会影响程序的性能；另一方面，使用动态资源可以改善加载时间，因其仅在需要时加载。
- (3) 静态资源不支持向前引用（forward reference），即任何资源都必须声明后使用；动态资源则没有这种限制。代码 12-24 由于是向前引用，所以 Window 的 Title 属性只能使用动态资源。

```
<Window ....
Title="{DynamicResource Title}" Height="300" Width="300">
<Window.Resources>
<s:String x:Key="Title">
    逻辑资源 Demo
</s:String>
</Window.Resources>
....
```

代码 12-24 Window 的 Title 属性引用动态资源

大多数情况下使用静态资源的方法，一般只有使用系统资源时才使用动态资源。

12.3.2 系统资源

动态资源主要用在系统资源，这是因为当系统环境设置发生变化时应用程序也需要随之变化，那么只有动态资源才能满足这样的需求。系统资源主要指 SystemColors、SystemFonts 和 SystemParameters 类。

在 System.Drawing 和 System.Windows 命名空间里均有这 3 个类。这里用到的都是 System.Windows 命名空间里的。而 System.Drawing 命名空间里的三个类主要是用在 WinFrom 程序和 .NetFramework2.0 程序当中。微软内部还是在不断地重复制造轮子，木木不由感慨了一下。

SystemColor 成员的所有属性均成对出现，如果有一个资源叫做“木木”，则一定有一个叫做“木木 Key”。所有以 Key 为后缀的属性都是 ResourceKey 类型，相当于前面的资源集合中的 x:Key。

因此用到 SystemColor 有两种方式：一种方式是不通过资源引用，而直接使用该属性。由于这些属性大部分是静态属性，因此要使用 x:Static，如代码 12-25 所示。

```
<Button Background="{x:Static SystemColors.ActiveCaptionBrush}">  
    直接用静态画刷属性  
</Button>
```

代码 12-25 不通过资源引用的方式

通过资源引用的方法语法要复杂一些，首先通过 {x:Static SystemColors.ActiveCaptionBrushKey} 返回 ResourceKey 类型的对象。它是资源引用所需要的类型对象，需要将 x:Static 表达式嵌套在资源引用表达式的内部，如代码 12-26 所示。

```
<Button MinHeight="70" Background="{DynamicResource {x:Static  
SystemColors.ActiveCaptionBrushKey}}">  
    使用动态资源引用  
</Button>
```

代码 12-26 通过资源引用的方式

代码 12-27 使用了 3 种表达式。

```
<Button MinHeight="70" Background="{x:Static SystemColors.ActiveCaptionBrush}">  
    直接用静态画刷属性  
</Button>  
<Button MinHeight="70" Background="{DynamicResource {x:Static  
SystemColors.ActiveCaptionBrushKey}}">  
    使用动态资源引用  
</Button>  
<Button MinHeight="70" Background="{StaticResource {x:Static  
SystemColors.ActiveCaptionBrushKey}}">  
    使用静态资源引用  
</Button>
```

代码 12-27 使用 3 种表达式

运行程序，结果如图 12-9 所示。

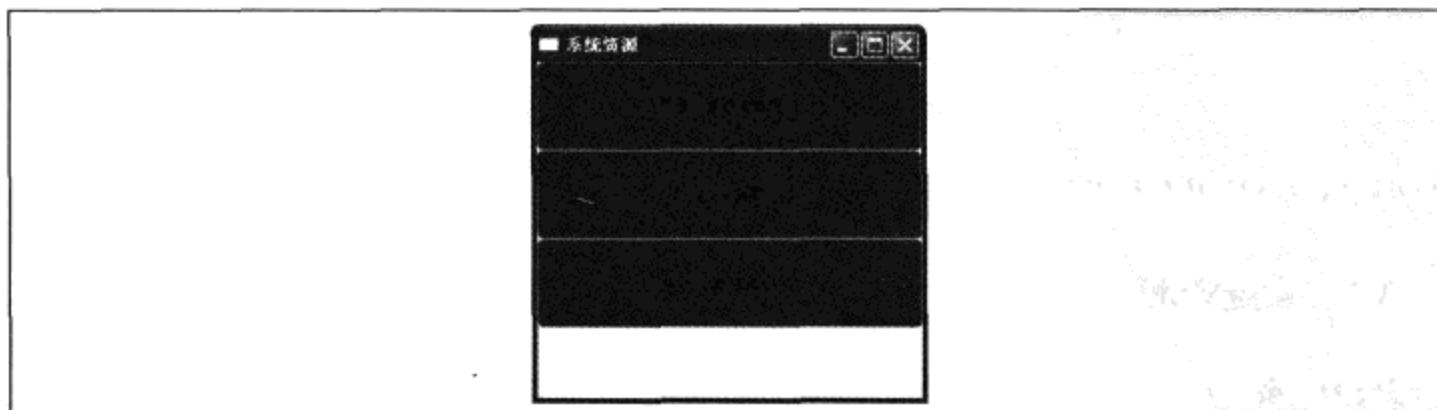


图 12-9 程序运行结果

将显示属性中的色彩方案由默认的蓝色改为银色，这时应用程序的 3 个按钮中只有动态资源按钮的背景色改变为银色，如图 12-10 所示。

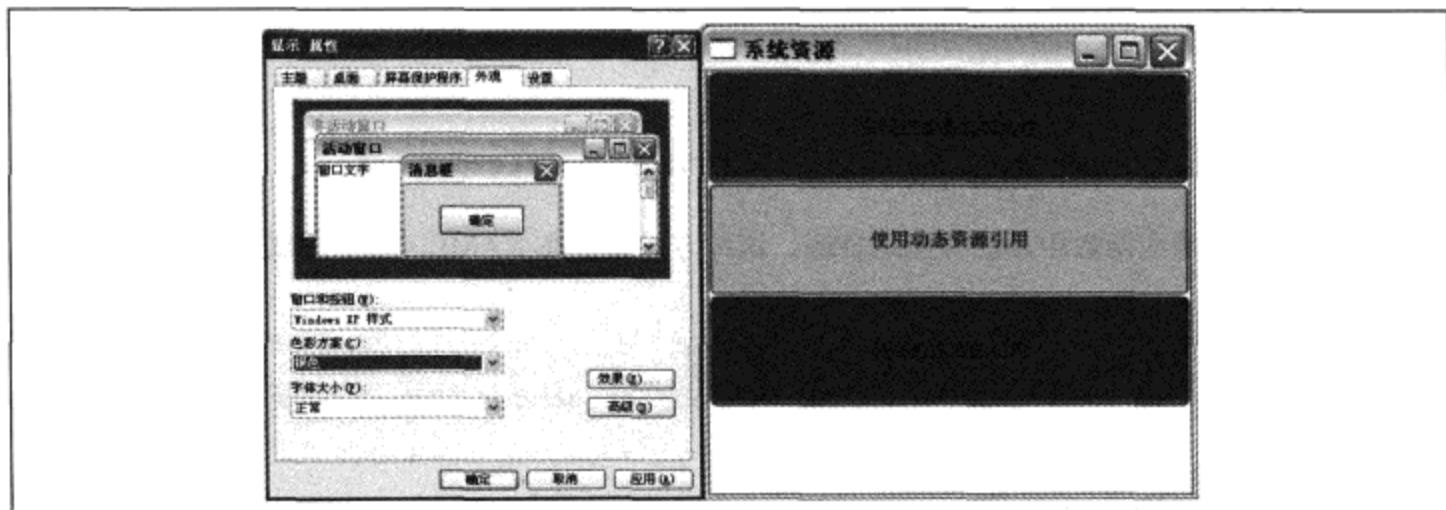


图 12-10 色彩方案修改后程序运行结果

12.3.3 共享资源

默认情况下，当有一个资源被应用到多处时使用的都是一个对象实例，这是理想情况。如果希望应用程序在应用资源的每处都有一个不同的对象实例，可以在资源中标记 `x:Shared="False"`。

如将 `Image` 作为资源，由于其派生自 `Visual` 类，会被添加到逻辑树和可视化树中，因此不能将 `Image` 的对象作为资源多次使用；否则会抛出异常。但是可以将 `Image` 资源标记成 `x:Shared="False"`，这样在应用这个资源时实际创建了不同的对象实例。

更好的做法应该是将 `BitmapImage` 作为资源，如代码 12-28 所示。

```
<Window.Resources>
    <BitmapImage x:Key="img" UriSource="avatar.jpg" />
</Window.Resources>
<StackPanel>
    <Button Width="64" Height="64" >
        <Image Source="{StaticResource img}" />
    </Button>
    <Button Width="64" Height="64" >
        <Image Source="{StaticResource img}" />
    </Button>
    ...

```

代码 12-28 将 `BitmapImage` 作为资源

12.3.4 通过代码定义和访问资源

在代码中定义和访问资源如代码 12-29 所示，首先在 XAML 文件中指定 `Window` 类的对象名 `MainWindow` 并添加一个 `Loaded` 事件。

```
MainWindow.xaml
<Window x:Class="mumu_accessresourcebycode.MainWindow"
    ...

```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Name="MainWindow" Loaded="MainWindow_Loaded"
Title="MainWindow" Height="300" Width="300">
<Grid>
    <Button Name="button1" />
</Grid>
</Window>
```

代码 12-29 指定 Window 类的对象名和添加 Loaded 事件

在 Loaded 事件的响应函数中添加并应用资源，这种方式类似在 XAML 中使用静态资源的方式，如代码 12-30 所示。

```
MainWindow.xaml.cs
private void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    // 添加新的画刷资源
    MainWindow.Resources.Add("yellowbrush", new
SolidColorBrush(Colors.Yellow));

    // 静态资源的引用方式
    button1.Background = (Brush)button1.FindResource("yellowbrush");
}
```

代码 12-30 在代码中使用静态资源

FindResource 会逐级向上查找资源，最终在应用程序中查找。如果未找到，则抛出一个异常 TryFindResource，如果查找失败，则返回一个 null 值。FindResource 也不支持前向引用，即如果上述代码 FindResource 在添加资源之前，则会抛出异常。当然访问资源也可以用一种更为直接的方式，如代码 12-31 所示。

```
button1.Background = (Brush)MainWindow.Resources["yellowbrush"];
```

代码 12-31 访问资源更为直接的方式

在代码中类似动态资源引用的方法是调用 SetResourceReference 方法，并且支持向前引用，如代码 12-32 所示。

```
private void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    // 动态资源的引用方式
    button1.SetResourceReference(Button.BackgroundProperty, "yellowbrush");
    // 添加新的画刷资源
    MainWindow.Resources.Add("yellowbrush", new
SolidColorBrush(Colors.Yellow));
}
```

代码 12-32 调用 SetResourceReference 方法

12.3.5 使用 ResourceDictionary 组织资源

当资源越来越多的时候，自然会引发一个问题，就是文件会越来越长。WPF 中提供一种 ResourceDictionary（资源字典）类型的 XAML 文件用于组织资源，在 VS2010 的解决方案窗口中右

击项目。选择快捷菜单中的 Add Resource Dictionary 选项可新建一个 ResourceDictionary 类型的文件，如图 12-11 所示。

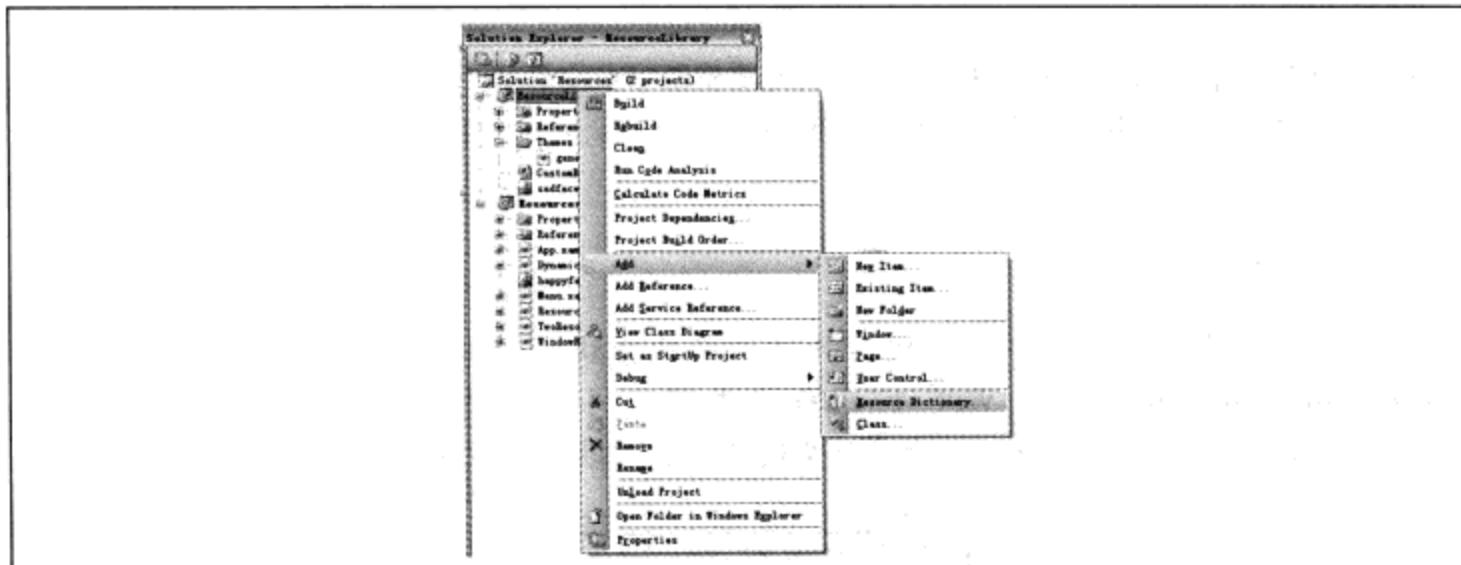


图 12-11 新建一个 ResourceDictionary 类型的文件

资源可以分门别类地放在该文件中，如所有的画刷资源均放在一个 Brushes.xaml 文件中，如代码 12-33 所示。

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:ResourceLibrary"
    >
    <ImageBrush
        x:Key="SadTileBrush"
        TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="/sadface.jpg" Opacity="0.3">
    </ImageBrush>
    ...
</ResourceDictionary>
```

代码 12-33 Brushes.xaml 文件

在应用程序 App.xaml 文件中通过 MergedDictionaries 包含资源字典文件，这样应用程序可以访问 Brushes.xaml 文件中的资源，如代码 12-34 所示。

```
<Application ....>
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Brushes.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

代码 12-34 包含资源字典文件

12.3.6 在程序集之间共享资源

1. 复制方法

在程序集之间共享资源有多种方法，最简单的是将资源字典文件从一个项目复制到另一个项目中。但是这种方法的问题是各项目中均可修改该资源字典文件，随着时间的推移，容易造成版本的不一致。

2. 程序集方法

将资源集中编译成一个程序集，然后共享给其他模块。为此新建两个工程，一个名为“mumu_sharerresources”，是 WPF 应用程序类型；另一个是 WPF Library 类型，名为“mumu_resourcesLib”。该工程为其他程序集提供资源，如图 12-12 所示。



图 12-12 mumu_sharerresources 工程的组织结构

右击 mumu_sharerresources 工程，选择快捷菜单中的 Project Dependencies（工程的依赖关系）选项，弹出 Project Dependencies 如图 12-13 所示的对话框。选择 mumu_resourcesLib 复选框，表明 mumu_sharerresources 工程需要依赖 mumu_resourcesLib 工程，然后单击 OK 按钮。

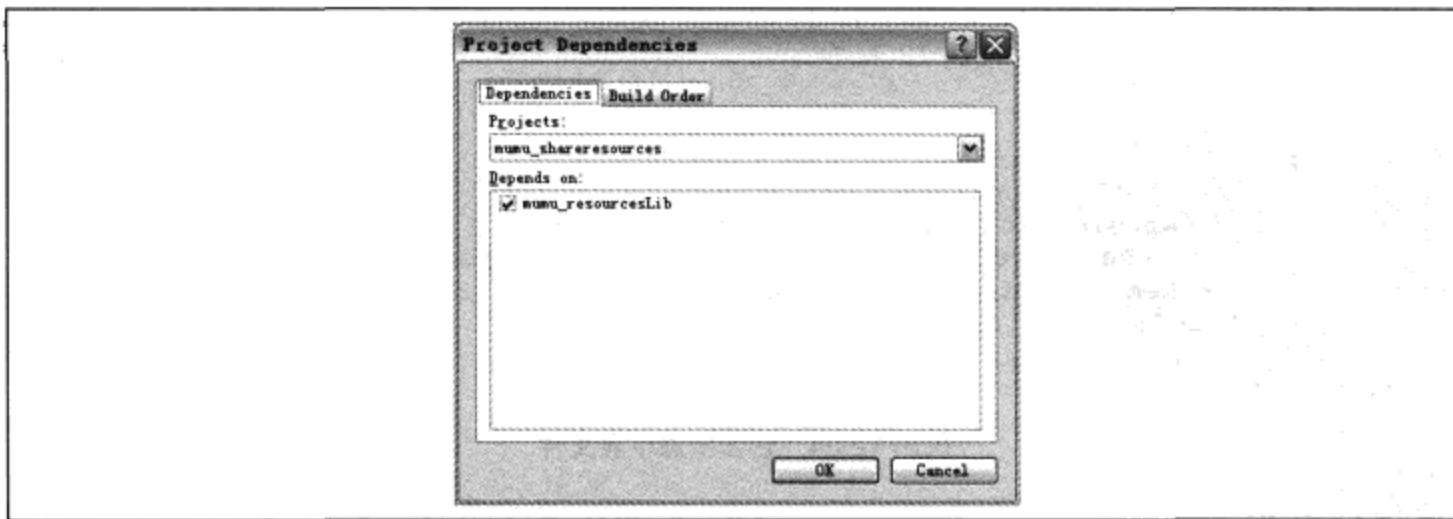


图 12-13 Project Dependencies 对话框

同时还需要为 `mumu_sharerources` 工程添加对 `mumu_resourcesLib` 工程的引用。

在 `Brushes.xaml` 文件中添加一个名为“`SadFaceBrush`”的资源，如代码 12-35 所示。

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ImageBrush
        x:Key="SadFaceBrush"
        TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="sadface.jpg" Opacity="0.3">
    </ImageBrush>
</ResourceDictionary>
```

代码 12-35 添加一个 `SadFaceBrush` 的资源

在 `mumu_sharerources` 工程的 `App.xaml` 文件中通过 `MergedDictionaries` 指定该文件，如代码 12-36 所示。

```
<Application x:Class="mumu_sharerources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary
Source="/mumu_resourcesLib;component/Brushes.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

代码 12-36 通过 `MergedDictionaries` 指定好 `App.xaml` 文件

这样即可直接使用资源字典中的任意一个资源，如代码 12-37 所示。

```
<Button Background="{StaticResource SadFaceBrush}" />
```

代码 12-37 直接使用资源字典中资源

3. 直接使用资源方法

还有一种方法甚至不用在 `App.xaml` 文件里通过 `MergedDictionaries` 将资源字典文件包含进来。它就像我们使用类似于 `SystemColor` 这种系统资源一样，直接取用。为此需要使用一种新的标记扩展 `ComponentResourceKey`。

要使用 `ComponentResourceKey`，每个资源的键名不再是一个简单的字符串，而是一个表达式，这个表达式包含两部分信息：`ComponentResourceKey` 所在的类型名称和资源 ID 的名称。

如在 `mumu_resourcesLib` 新建一个类，名为“`mumuBrushes`”，如代码 12-38 所示。

```
using System.Windows;
namespace mumu_resourcesLib
{
```

```
public class mumuBrushes
{
    public static ComponentResourceKey happyfaceBrushKey
    {
        get
        {
            return new ComponentResourceKey(typeof(mumuBrushes), "happyfaceBrush");
        }
    }
}
```

代码 12-38 在 mumu_resourcesLib 新建一个 mumuBrushes 类

需要在资源字典中声明该资源，注意这里的 `x:Key` 是一个 `ComponentResourceKey` 的表达式，`ImageSource` 属性使用了一个完整的 URI。为了很清楚地指定资源的位置，这种写法会更好一些，如代码 12-39 所示。

```
<ResourceDictionary .....>
.....
<ImageBrush
    x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type
local:mumuBrushes},ResourceId=happyfaceBrush}"
    TileMode="Tile"
    ViewportUnits="Absolute" Viewport="0 0 32 32"
    ImageSource="/mumu_resourcesLib;component/happyface.jpg" Opacity="0.3">
</ImageBrush>
</ResourceDictionary>
```

代码 12-39 在资源字典中声明该资源

在 `mumu_sharerесources` 工程中引用该资源，需要首先声明 `mumu_resourcesLib` 程序集。引用的语法也稍有些特殊，如代码 12-40 所示。

```
<Window x:Class="mumu_sharerесources.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:res ="clr-namespace:mumu_resourcesLib;assembly=mumu_resourcesLib"
    Title="MainWindow" Height="300" Width="300">
.....
    <Button Grid.Row="1" Background="{StaticResource {ComponentResourceKey
TypeInTargetAssembly= res:mumuBrushes,ResourceId=happyfaceBrush}}" />
</Window>
```

代码 12-40 在 `mumu_sharerесources` 工程中引用该资源

12.4 接下来做什么

在过去的程序里，如果有一个固定的常量，天真的做法是硬编码在代码文件里，随处可见。一旦需要修改，不得不使用文本编辑器的“查找\替换”功能。这种方法对于小项目还凑合能用，但是对于中大型项目无疑是灾难性的。于是我们常常会作这样的定义 `static const PI =3.14`。一旦需要修改只需要修改 `PI` 这一处即可。而 WPF 里使用了逻辑资源大大扩展了常量这种概念，确实使程序的可扩展性和可维护性更好，开发应用效率更高。

有了资源的基础，下面是 XAML 当中非常重要的概念——样式和模板。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 书剑恩仇录》，“第十六回 我见犹怜二老意 谁能遣此双姝情”。
- [2] MSDN Library for Visual Studio 2008 SP1 Resources Overview。
- [3] MSDN Library for Visual Studio 2008 SP1 Pack URIs in Windows Presentation Foundation。

第13章

样式和控件模板——听香水榭，千变阿朱

阿碧话未说完，后堂转出一个须发如银的老人，手中撑着一根拐杖，说道：“阿碧，是谁在这里大呼小叫的？”说的却是官话，语音甚是纯正……那老人双手乱摇，说道：“这个我可做不起主，我也不是什么管家。”鸠摩智道：“那么尊府的管家是谁？请出来一见。”……过了半晌，只听得脚步声响，内堂走出一个五十来岁的瘦子。脸色焦黄，下额留一丛山羊短须，一副精明能干的模样。身上衣着颇为讲究，左手小指戴一枚汉玉扳指，看来便是慕容府中的管家了……

过了好一会，只听得佩环叮当，内堂走出一位老夫人来。只见她身穿古铜缎子袄裙，腕戴玉镯，珠翠满头，打扮得雍容华贵。脸上皱纹甚多，眼睛迷迷蒙蒙的，似乎已瞧不见东西……

——《天龙八部》，“第十一章 向来痴”^[1]

“向来痴”一章是阿朱初次登场，她一出场就表现不俗，乔装改扮之术神乎其微。不但形状极似，而言语举止，无不毕肖，可说没半点破绽。因此以鸠摩智之聪明机智，崔百泉之老于江湖，都没有丝毫疑心。但是阿朱的易容术还是有其破绽，在于她那淡淡的少女体香，段誉就从她身上无法掩饰的一些淡淡幽香之中发现了真相。

控件模板也是如此，开发人员利用模板只要有足够的想象力，即可使按钮变成任何一个部件。千变万化，但是万变不离其宗，改变不了的是按钮单击仍然会触发 Click 事件这样固有的特性。

本章分为两个部分：一是介绍改变控件的基础方法，即样式；二是介绍一种更能灵活定制控件外观的方法，即控件模板。介绍控件模板的时候，我们仍然为木木量身定做了一个故事，故事的名字就叫做“木木，阿朱和她的易容术”（貌似一个很古老的故事名字“泥巴，女人和狗”^_^）。

本章内容如下。

- (1) 样式那一点事儿。
- (2) 听香水榭边，须发如银人。
- (3) 淡淡少女香，侃侃孙三谈。
- (4) 龙钟老太太，妙龄俏阿朱。
- (5) 接下来做什么。

不过在这之前，我们还是先从样式开始……

13.1 样式那点事儿

13.1.1 何来样式

代码 13-1 在一个 WrapPanel 面板中依次放置 3 个按钮（详见 `mumu_simplebuttonsbycode` 工程）。

```
<Window x:Class="mumu_simplebuttonsbycode.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="样式" Height="300" Width="300">
    <WrapPanel x:Name="wrappanel">
        <Button Content="Button1"/>
        <Button Content="Button2"/>
        <Button Content="Button3"/>
    </WrapPanel>
</Window>
```

代码 13-1 在 WrapPanel 面板中依次放置 3 个按钮

如果希望稍稍改变按钮外观，如图 13-1 所示。

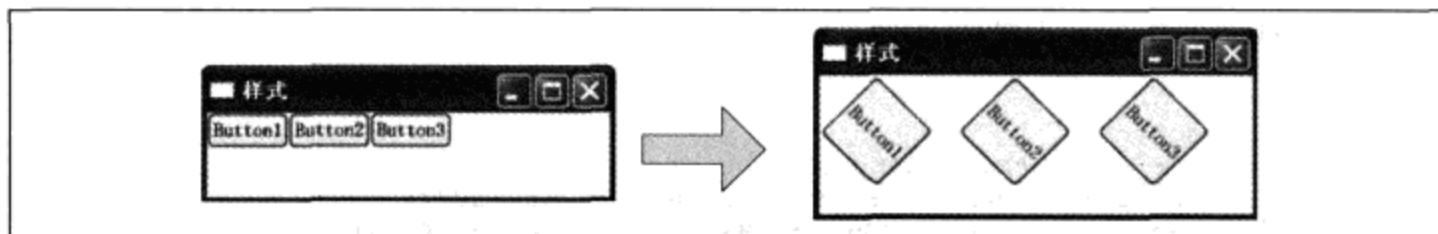


图 13-1 改变按钮外观

在窗口的 Loaded 事件中通过循环设置按钮的属性，如代码 13-2 所示。

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    int count = wrappanel.Children.Count;
    for (int i = 0; i < count; i++)
    {
        Button btn = wrappanel.Children[i] as Button;
        RotateTransform rotatetransform = new RotateTransform();
        rotatetransform.Angle = 45;
        btn.RenderTransform = rotatetransform;
        btn.Background = new SolidColorBrush(Colors.Beige);
        btn.Height = 50;
        btn.Margin = new Thickness(35, 0, 0, 0);
    }
}
```

代码 13-2 Loaded 事件处理函数

如果在 XAML 文件中设置，由于其不支持循环。因此必须多次重复操作，代码 13-3 通过 XAML 语言来设置的按钮外观。

```

<Window x:Class="mumu_simplebuttonsbyxaml.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="样式" Height="300" Width="300" >
    <WrapPanel x:Name="wrappanel">
        <Button Content="Button1" Height="50" Margin="35,0,0,0" Background="Beige">
            <Button.RenderTransform>
                <RotateTransform Angle="45"/>
            </Button.RenderTransform>
        </Button>
        <Button Content="Button2" Height="50" Margin="35,0,0,0" Background="Beige">
            <Button.RenderTransform>
                <RotateTransform Angle="45"/>
            </Button.RenderTransform>
        </Button>
        <Button Content="Button3" Height="50" Margin="35,0,0,0" Background="Beige">
            <Button.RenderTransform>
                <RotateTransform Angle="45"/>
            </Button.RenderTransform>
        </Button>
    </WrapPanel>
</Window>

```

代码 13-3 通过 XAML 语言设置按钮外观

好在这个例子只有 3 个按钮，如果不是 3 个，是 30 个，300 个。使用 XAML 语言这样不停地复制粘贴，是会“死人”的。而样式弥补了 XAML 语言中不能使用循环的缺憾，可以为一个或者多个元素设置属性。通过样式实现上例，首先定义一个按钮样式的资源，如代码 13-4 所示（详见 `mumu_simplebuttonsbyxaml` 工程）。

```

.....
<Window.Resources>
    <Style x:Key="ButtonStyle">
        <Setter Property="Button.Height" Value="50"/>
        <Setter Property="Button.Margin" Value="35,0,0,0"/>
        <Setter Property="Button.Background" Value="Beige"/>
        <Setter Property="Button.RenderTransform">
            <Setter.Value>
                <RotateTransform Angle="45"/>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
</Window>

```

代码 13-4 定义一个按钮样式的资源

只要从 `FrameworkElement` 或 `FrameworkContentElement` 派生的对象均有一个 `Style` 属性，按钮当然也不例外。即可将定义的样式赋值给按钮的该属性，如代码 13-5 所示。

```

<WrapPanel x:Name="wrappanel">
    <Button Content="Button1" Style="{StaticResource ButtonStyle}" />
    <Button Content="Button2" Style="{StaticResource ButtonStyle}" />
    <Button Content="Button3" Style="{StaticResource ButtonStyle}" />
</WrapPanel>

```

代码 13-5 将定义的样式赋值给按钮的 `Style` 属性

13.1.2 基本用法

我们可以将 Style 的 TargetType 属性设置为 Button 类型，这样所有 Property 的值都可以省略前面的 Button，如代码 13-6 所示。

```
<Window.Resources>
    <Style x:Key="ButtonStyle" TargetType="{x:Type Button}">
        <!--<Setter Property="Button.Height" Value="50"/><!--&gt;
        &lt;Setter Property="Height" Value="50"/&gt;
        &lt;Setter Property="Margin" Value="35,0,0,0"/&gt;
        &lt;Setter Property="Background" Value="Beige"/&gt;
        &lt;Setter Property="RenderTransform"&gt;
            &lt;Setter.Value&gt;
                &lt;RotateTransform Angle="45"/&gt;
            &lt;/Setter.Value&gt;
        &lt;/Setter&gt;
    &lt;/Style&gt;
&lt;/Window.Resources&gt;</pre>
```

代码 13-6 将 Style 的 TargetType 属性设置成 Button 类型

甚至只需要设置 TargetType 属性，而不需要设置 x:Key 关键字。这种 Style 称为“隐式 Style”，它会应用到符合 TargetType 指定的类型元素上。如果这个 Style 资源在窗口中定义，则其应用范围为该窗口；如果是在应用程序中定义的，则其应用范围就是整个应用程序。如代码 13-7 所示，所有的按钮并没有设置 Style 属性，但默认应用窗口资源中定义的 Style：

```
<Page .....>
    <Page.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Height" Value="50"/>
            <Setter Property="Margin" Value="35,0,0,0"/>
            <Setter Property="Background" Value="Beige"/>
            <Setter Property="RenderTransform">
                <Setter.Value>
                    <RotateTransform Angle="45"/>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>
    <WrapPanel x:Name="wrappanel">
        <Button Content="Button1" />
        <Button Content="Button2" />
        <Button Content="Button3" />
    </WrapPanel>
</Page>
```

代码 13-7 按钮默认应用窗口资源中定义的 Style

在该面板中添加另外一种类型的控件，如 RadioButton。该控件的一些属性的设置同 Button，如高度为 50，并且旋转 45 度角等。但是还希望有一些属性的设置与 Button 不同，如果样式有一种类似面向对象的继承机制，则可保证其复用，如图 13-2 所示。

样式的继承机制通过 BasedOn 属性来实现，我们将 Button 和 RadioButton 需要共同设置的属性抽象为一个基类样式。由于二者均派生自 Control 类，因此可以将这个样式的 TargetType 设置为 Control。然后通过 BasedOn 从这个样式派生控制 Button 和 RadioButton 其他属性的样式，如代码 13-8 所示。

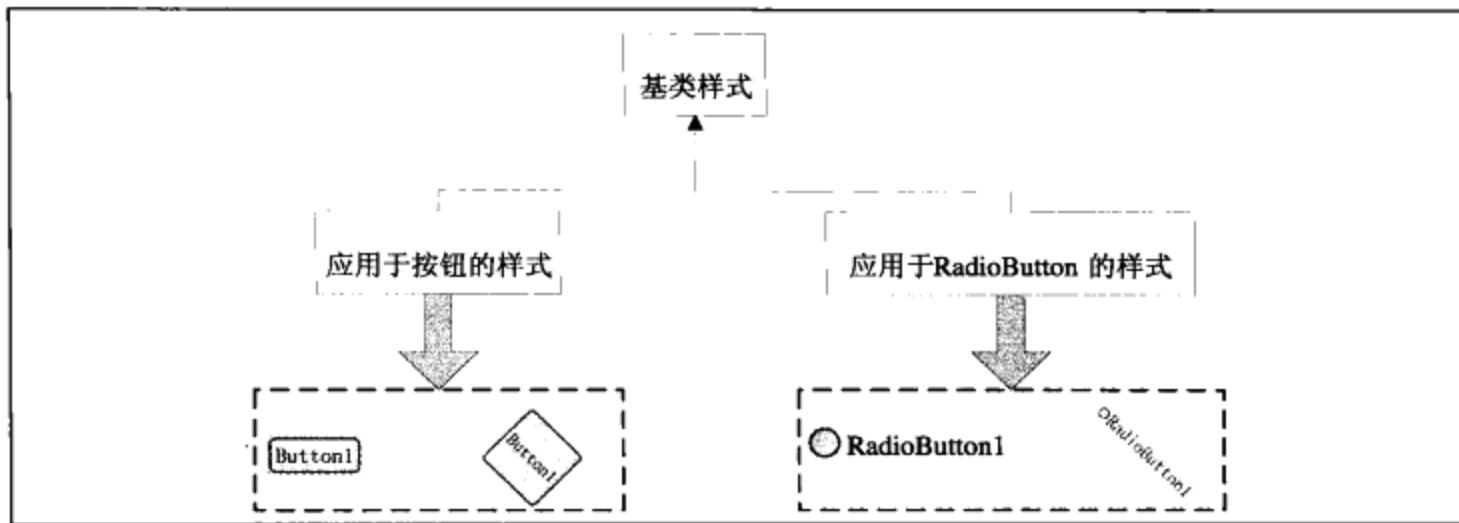


图 13-2 样式的继承机制

```

<!--基类样式-->
<Style TargetType="{x:Type Control}">
    <Setter Property="Height" Value="50"/>
    <Setter Property="Margin" Value="35,0,0,0"/>
    <Setter Property="Background" Value="Beige"/>
    <Setter Property="RenderTransform">
        <Setter.Value>
            <RotateTransform Angle="45"/>
        </Setter.Value>
    </Setter>
</Style>
<!--从基类样式派生下来的 Button 样式-->
<Style BasedOn="{StaticResource {x:Type Control}}"
    TargetType="{x:Type Button}">
    <Setter Property="Foreground" Value="Blue"/>
</Style>
<!--从基类样式派生下来的 RadioButton 样式-->
<Style BasedOn="{StaticResource {x:Type Control}}"
    TargetType="{x:Type RadioButton}">
    <Setter Property="FontSize" Value="22"/>
    <Setter Property="Foreground" Value="Red"/>
</Style>

```

代码 13-8 样式的继承

在上面的代码中 `BasedOn` 设置的值使用了一个比较奇怪的语句，即 `{StaticResource {x:Type Control}}`。如果基类样式有一个键名为“`basestyle`”，即 `x:Key="basestyle"`，那么 `BasedOn` 的设置应该写为“`BasedOn="{StaticResource basestyle}"`”。但是基类样式没有键名，因此其键名就被认为是 `TargetType` 值。注意该值是一个 `Type` 类型，而不是一个字符串。如果显式访问无键名的 `Style`，则变为“`{StaticResource {x:Type Control}}`”。

13.1.3 触发器

样式还有一个 `Trigger` 集合，这是专门存储触发器的集合。我们首先查看如何通过触发器实现鼠标滑过按钮时改变其背景色的功能，如代码 13-9 所示。

```

<Style BasedOn="{StaticResource {x:Type Control}}"
    TargetType="{x:Type Button}">

```

```
<Setter Property="Foreground" Value="Blue"/>
<Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="Background" Value="BlanchedAlmond">
        </Setter>
    </Trigger>
</Style.Triggers>
</Style>
```

代码 13-9 通过触发器实现鼠标滑过按钮时改变按钮背景色的功能

上面代码中使用的触发器称为“属性触发器”，在属性值改变时执行该触发器中的 Setter 集合（注意不是样式中的集合）。WPF 有如下 3 种类型的触发器。

- (1) 属性触发器 (Trigger)：如上所述。
- (2) 数据触发器 (DataTrigger)：普通.NET 属性，而不仅仅依赖属性值改变时触发。
- (3) 事件触发器 (EventTrigger)：触发路由事件时会被调用。

Trigger 集合保存在样式和模板中，FrameworkElement 中也包含该集合，但是 FrameworkElement 仅仅支持一种类型的事件触发器。

代码 13-10 为一个本书部分章节的清单。

```
<Page.Resources>
    <XmlDataProvider x:Key="InventoryData" XPath="Inventory/Book">
        <x:XData>
            <Inventory xmlns="">
                <Book>
                    <Chapter Number="2">
                        <Title>依赖属性—木木的“汗血宝马”</Title>
                    </Chapter>
                    <Chapter Number="2">
                        <Title>路由事件—绝情谷底玉蜂飞</Title>
                    </Chapter>
                    <Chapter Number="3">
                        <Title>应用程序和窗口—大侠的成长路线</Title>
                    </Chapter>
                    <Chapter Number="3">
                        <Title>导航—天罡北斗阵演绎</Title>
                    </Chapter>
                    <Chapter Number="4">
                        <Title>样式和控件模板—听香水榭，千变阿朱</Title>
                    </Chapter>
                </Book>
            </Inventory>
        </x:XData>
    </XmlDataProvider>
....
```

代码 13-10 本书部分章节的清单

代码 13-11 通过一个 ListBox 显示这个列表清单。

```
<StackPanel Margin="20">
```

```

<ListBox HorizontalAlignment="Center"
Padding="2">
    <ListBox.ItemsSource>
        <Binding Source="{StaticResource InventoryData}"
XPath="*"/>
    </ListBox.ItemsSource>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock FontSize="20" Margin="0,0,10,0" FontFamily="华文行楷">
                <TextBlock.Text>
                    <Binding XPath="Title"/>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
</StackPanel>

```

代码 13-11 通过一个 ListBox 显示这个列表清单

程序运行结果如图 13-3 所示。

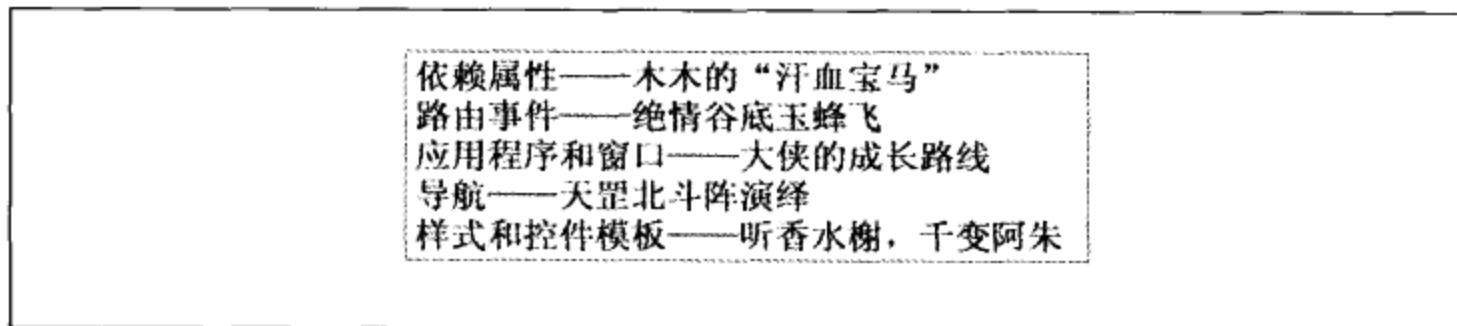


图 13-3 程序运行结果

现在首先为 ListBox 添加一个从透明，慢慢清晰到最终显示的一个动画，这个效果可以通过事件触发器来实现。该触发器甚至不需要写在样式，而直接放置在 ListBox 中，如代码 13-12 所示。

```

<ListBox.Triggers>
    <EventTrigger RoutedEvent="ListBox.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetProperty="Opacity" From="0"
To="1" Duration="0:0:5"/>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</ListBox.Triggers>

```

代码 13-12 为 ListBox 添加一个动画

EventTrigger 的 RoutedEvent 属性表示发生的事件，上面的代码表示 ListBox 的 Loaded 事件。与属性触发器不同，EventTrigger 中是 TriggerAction 的集合，即发生的事件和执行的操作。BeginStoryboard 是一种操作，即开始动画效果，程序运行结果如图 13-4 所示。

通过每一章不同的卷号来设置不同的颜色（Column），Column 不是一个依赖属性，因此只能用 DataTrigger 来实现。DataTrigger 之所以能够支持普通.NET 属性，是因为它比属性触发器多了一个属

性 Binding。可以通过绑定来指定相关属性，如代码 13-13 所示。

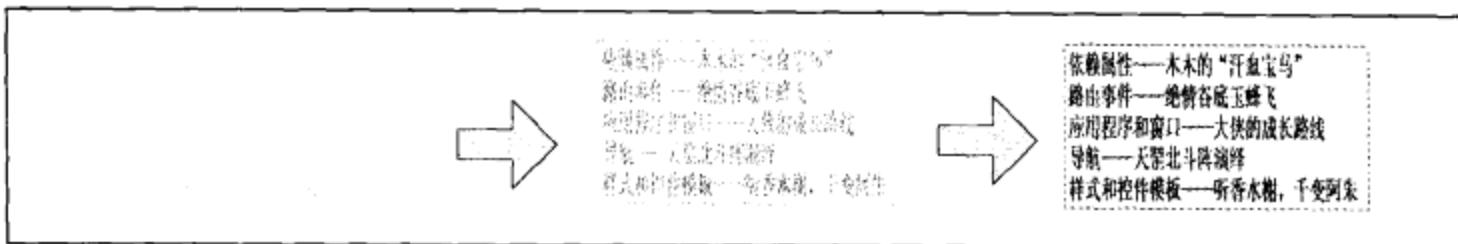


图 13-4 程序运行结果

```
<Style TargetType="{x:Type ListBoxItem}">
    <Setter Property="Margin" Value="0,2,0,2" />
    <Setter Property="Padding" Value="0,2,0,2" />
    <Setter Property="FontFamily" Value="华文行楷" />
    <Style.Triggers>
        <DataTrigger Binding="{Binding XPath=@Column}" Value="3">
            <Setter Property="TextBlock.Foreground" Value="CadetBlue"/>
        </DataTrigger>
        <DataTrigger Binding="{Binding XPath=@Column}" Value="4">
            <Setter Property="TextBlock.Foreground" Value="SlateGray"/>
        </DataTrigger>
    </Style.Triggers>
</Style>
```

代码 13-13 DataTrigger 通过绑定来指定相关属性

这样最终的程序运行结果如图 13-5 所示。

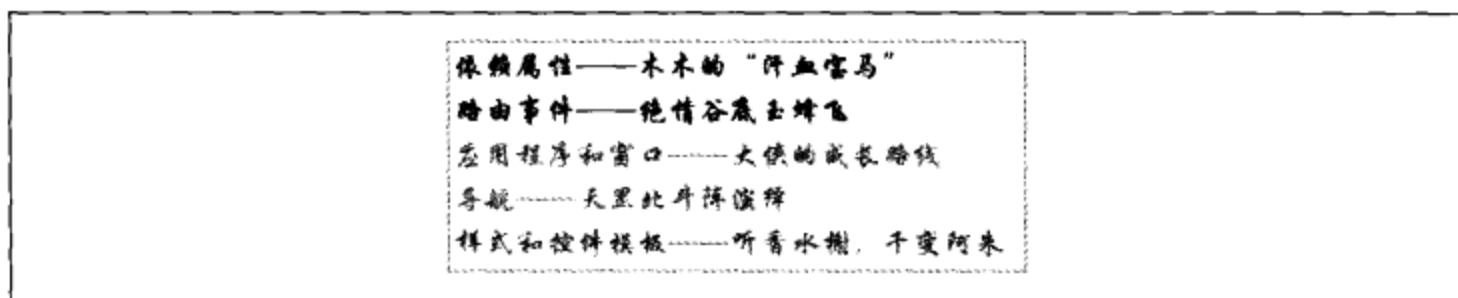


图 13-5 最终的程序运行结果

样式的那一点儿事我们就说完了，接下来就进入了“木木，阿朱和她的易容术”的故事。

13.2 模板示例——听香水榭边，须发如银人

阿朱姑娘在过去以易容术见长，为人修改容貌，现在阿朱姑娘是一名软件设计师，为软件修改容貌。自从黄岛主的软件公司集体转型到 WPF（具体参见布局——药师的桃花岛），她加盟到了桃花岛软件公司。这个时候的木木已经是黄岛主的乘龙快婿，过了一段新婚燕尔的生活，不免有些腻了。由于喜好 WPF，于是经常在桃花岛软件公司里向很多技术人员讨教学习，好不快活。

一日，木木仍然像往常一样在桃花岛软件公司闲逛，走到一处，抬头一看“听香水榭”。好雅的名字，于是推门进去。轻轻一推，门“吱”的一声开了。里面的景象让木木大吃一惊，一汪白水，一座横桥斜搭在一个不知是小岛还是半岛之上，疏疏落落四五座房舍，最前面的一个房舍小巧玲珑，

颇为精雅，小舍匾额上写着“听香水榭”四字，笔迹颇为潇洒。

木木踏入小舍厅堂，大声问到：“请问有人么？”话音刚落，就听到一阵咳嗽声，顺着声音望去，右手旁一个须发如银的老人正襟危坐在电脑旁，知道有人进来，颤颤巍巍，柱着拐杖站起来。扭过身来，只见这老人弓腰曲背，满脸都是皱纹，没有九十也有八十岁，他嘶哑着嗓子说道：“欢迎，欢迎啊，听香水榭好久都没有客人了。”木木不由倒吸一口寒气，自己见过对技术痴迷的程序员最大年龄的也莫过于周伯通大哥，但是眼前的这个老人好像比周大哥还要大上二三十岁，这么老的程序员，即使微软、Google 这样的软件公司恐怕也找不出来一个。

木木连忙快走几步上前说到：“老人家，您快坐。”说完扶着老人坐下，木木侧目一看，老人电脑上开着 VS2010，于是好奇心大起。老人咳嗽了两声说到：“我黄老头，是这儿的设计师，我正在用模板设计一个按钮，主要是两个圆形，一个圆形是外圆，填充从蓝到红的渐变色；另一个圆形是内圆，填充从白色到透明的渐变画刷。”只见老黄的代码（如代码 13-14 所示，详见 `mumu_fancybutton` 工程）：

```
<Window x:Class="mumu_fancybuttontrigger.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="老黄的按钮" Height="300" Width="300">
<Grid>
    <Grid.Resources>
        <ControlTemplate x:Key="buttonTemplate">
            <Grid Width="100" Height="100">
                <!--第 1 个外圆-->
                <Ellipse Width="100" Height="100">
                    <Ellipse.Fill>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                            <GradientStop Offset="0" Color="Blue"/>
                            <GradientStop Offset="1" Color="Red"/>
                        </LinearGradientBrush>
                    </Ellipse.Fill>
                </Ellipse>
                <!--第 2 个内圆-->
                <Ellipse Width="80" Height="80">
                    <Ellipse.Fill>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                            <GradientStop Offset="0" Color="White"/>
                            <GradientStop Offset="1" Color="Transparent"/>
                        </LinearGradientBrush>
                    </Ellipse.Fill>
                </Ellipse>
            </Grid>
        </ControlTemplate>
    </Grid.Resources>
    <Button Template="{StaticResource buttonTemplate}" Click="Button_Click"/>
</Grid>
</Window>
```

代码 13-14 按钮之一

程序的运行结果如图 13-6 所示。

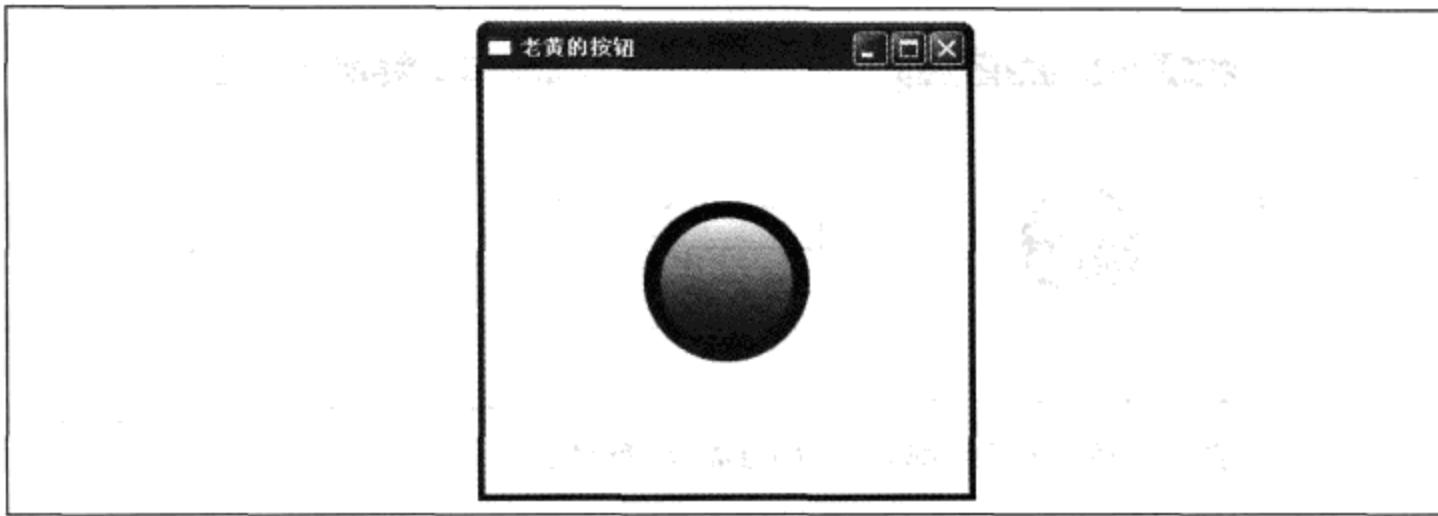


图 13-6 程序的运行结果

老黄用鼠标滑动到按钮上，点了一下，不由皱皱眉头，自言自语到这个还不行。于是又做了如下改动：

- (1) 为第 1 个外圆添加名为“outerCircle”，显然要在后面处理这个外圆。
- (2) 为控件模板添加一些 Trigger，使得鼠标在按钮上滑过和按下时有所变化。

代码 13-15 中加粗部分为修改的代码（详见 `mumu_fancybuttontrigger` 工程）。

```
<Grid.Resources>
    <ControlTemplate x:Key="buttonTemplate">
        <Grid Width="100" Height="100">
            <Ellipse x:Name="outerCircle" Width="100" Height="100">
                .....
            </Ellipse>
            .....
        </Grid>
        <ControlTemplate.Triggers>
            <Trigger Property="Button.IsMouseOver" Value="True">
                <Setter TargetName="outerCircle" Property="Fill"
Value="Orange"/>
            </Trigger>
            <Trigger Property="Button.IsChecked" Value="True">
                <Setter Property="RenderTransform">
                    <Setter.Value>
                        <ScaleTransform ScaleX="0.9" ScaleY="0.9"/>
                    </Setter.Value>
                </Setter>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
</Grid.Resources>
```

代码 13-15 按钮之二

运行程序，单击按钮时按钮不仅缩小，而且整体位置偏移，如图 13-7 所示。

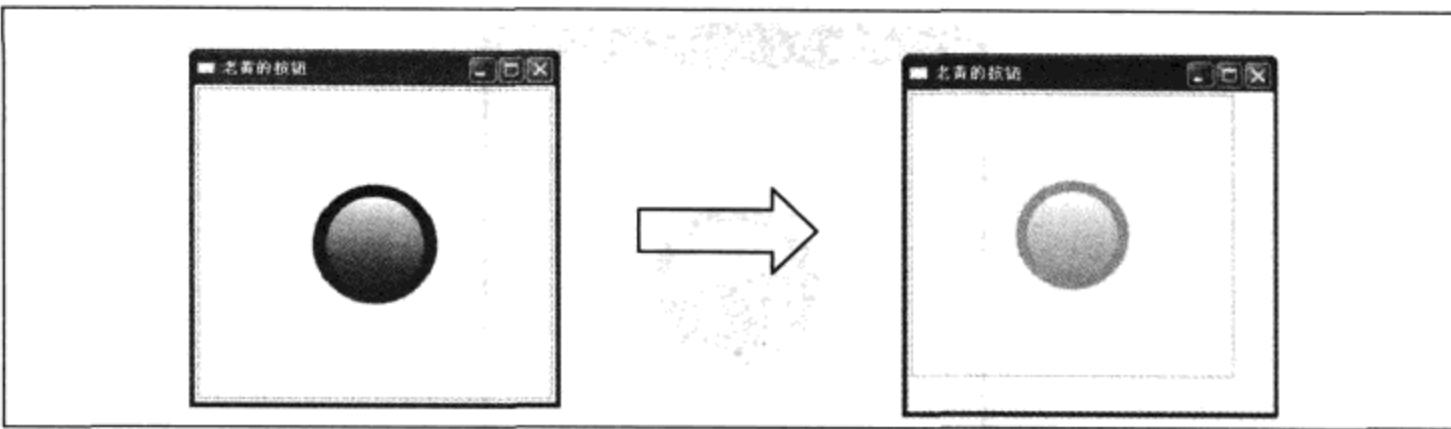


图 13-7 单击按钮时按钮的外部轮廓变小，而且整体位置偏移

木木见状说到：“老人家，您看看是不是让变换的原点设置在中心，鼠标按下就不会偏移了。”说完，木木加了一行代码：

```
<Setter Property ="RenderTransformOrigin" Value=".5,.5"/>
```

代码 13-16 添加的代码

运行程序，单击按钮时按钮的外部轮廓变小且整体位置不再偏移，如图 13-8 所示。

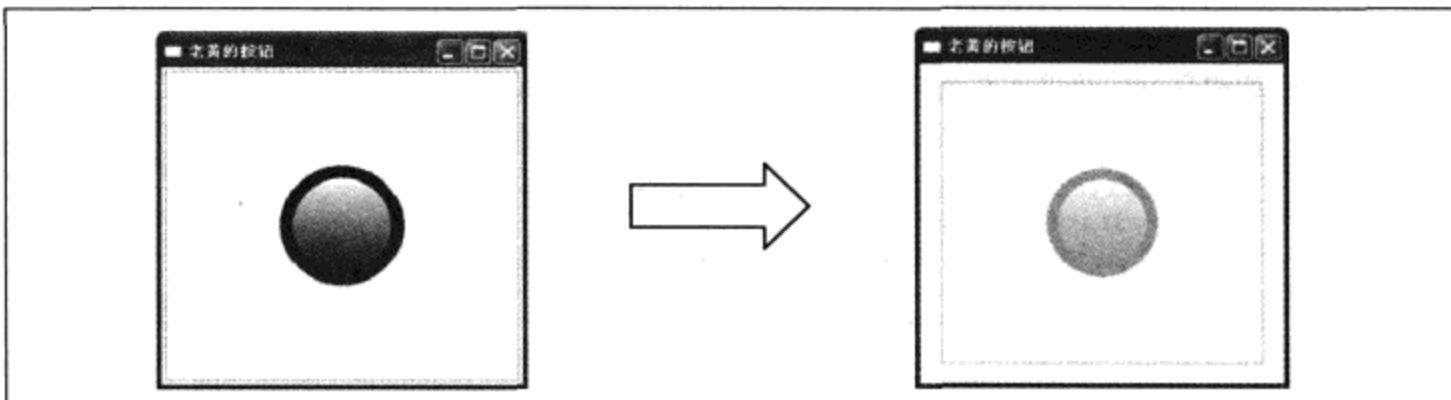


图 13-8 单击按钮时按钮的外部轮廓变小且整体位置不再偏移

黄老头不敢小看眼前的年轻人了，说到：“看不出来，你对 WPF 也有点研究啊。”木木憨憨一笑，说到：“哪有，我只不过是知道有这个属性而已，这控件模板确实是头一次看到，以前学过样式，但是样式还只是停留在改改属性，可是模板将整个控件的外观完全改变，威力实在是太强大，令人咂舌。老人家，您能不能教教我这控件模板？”

黄老头双手乱摇，说道：“这个我可担当不起，我也着实不懂，要不还是请我们这儿的孙工给你讲讲？”木木大喜说道：“那么孙工在哪儿？请出来一见。”黄老头连连点头，说道：“很好，很好！我去请孙工来。”转过身子，摇摇摆摆地走了出去。

13.3 模板工作原理——淡淡少女香，侃侃孙三谈

过了半晌，只听得脚步声响，内堂中走出一个五十来岁的瘦子，看来便是孙工了。孙工向木木行了一礼说道：“黄老头说听香水榭来了一个年轻客人，对 WPF 很感兴趣。我是这儿的工程师孙三，还

请你多多指教。”他说到这里，木木忽然闻到一股香气，心中一惊：“奇怪，奇怪。”先前黄老头在的时候，木木便闻到一股香气，但是黄老头出去，这股香气就此消失，现在孙工出来，这股香气又随即出现。于是木木寻思：“莫非后堂种植了什么奇花异卉，有谁从后堂出来，身上便带有幽香？要不然那黄老头和孙公都是女子所扮。”

木木虽然疑心孙工是女子所扮，但瞧来瞧去，确实无半点破绽，忽然想起：“女子要扮男人，这喉结须假装不来。”凝目向孙工喉间瞧去，只见他将衬衣领子竖起，刚好挡住了喉头。木木又见他胸间饱满，虽不能就此说是女子，但这样精瘦的一个男人，胸间决不会如此肌肉丰隆。木木发觉了这个秘密，甚觉有趣，赶紧还了一个礼，说道：“孙工，我确实不太懂模板，还请您从原理上讲讲。”于是孙工开始系统地讲解模板原理……

WPF 有 4 种模板分别为 ControlTemplate（控件模板）、DataTemplate（数据模板）、HierarchicalDataTemplate 和 ItemsPanelTemplate，它们均继承自 FrameworkTemplate，如图 13-9 所示。

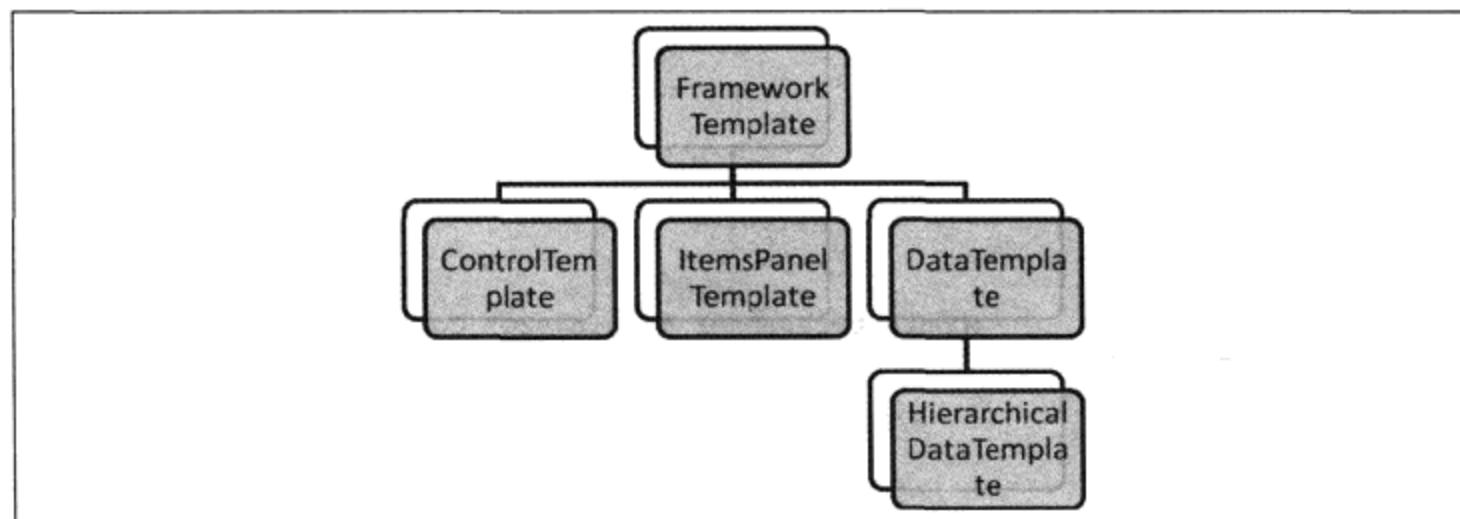


图 13-9 4 种不同类型模板均继承自 FrameworkTemplate

模板通过改变控件的可视化树（visual Tree）来彻底改变其外观，如前例将按钮的可视化树变为两个嵌套的圆。

控件均派生自 Control 的类，只有 Control 及其派生类才有 template 这样的属性。孙工抿了一口茶，说到：“前面黄老头的例子是一个很不完善的控件模板例子，主要的不足之处在于两点。”

（1）不能设置按钮的多个属性，如设置最常用的 Content 属性没有任何效果。

（2）按钮以前的默认行为消失，如按钮不可用时字体变灰等这些行为。

说完，孙工将老黄的例子修改了一下（详见 mumu_fancybuttontrigger2 工程）：

```
<Window x:Class="mumu_fancybuttontrigger.MainWindow"
.....
    Title="老黄的按钮" Height="300" Width="300">
        <Grid>
            <Grid.Resources>
                <ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type
                    Button}">
```

```

    // 已经将 Grid 的 Width 和 Height 值去掉
    <Grid <!-- Width="100" Height="100"--> >
        <Ellipse x:Name="outerCircle">
            <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0"
                    EndPoint="0,1">
                    // 一种常规的数据绑定方法
                    <GradientStop Offset="0" Color="{Binding
                        RelativeSource={RelativeSource TemplatedParent}, Path=Background.Color}"/>
                    <GradientStop Offset="1" Color="Red"/>
                </LinearGradientBrush>
            </Ellipse.Fill>
        </Ellipse>
        <Ellipse RenderTransformOrigin=".5,.5">
            // 第 2 个椭圆的大小是第 1 个椭圆的 0.8，与绝对数值无关
            <Ellipse.RenderTransform>
                <ScaleTransform ScaleX=".8" ScaleY=".8"/>
            </Ellipse.RenderTransform>
            <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0"
                    EndPoint="0,1">
                    <GradientStop Offset="0" Color="White"/>
                    <GradientStop Offset="1"
                        Color="Transparent"/>
                </LinearGradientBrush>
            </Ellipse.Fill>
        </Ellipse>
        <Viewbox>
            // 模板绑定
            <ContentPresenter Margin="20" Content="{TemplateBinding
                Content}"/>
        </Viewbox>
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="Button.IsMouseOver" Value="True">
            // 一种常规的数据绑定方法
            <Setter TargetName="outerCircle" Property="Fill"
                Value="{Binding RelativeSource={RelativeSource TemplatedParent},
                    Path=BorderBrush}"/>
        </Trigger>
        <Trigger Property="Button.IsChecked" Value="True">
            .....
        </Trigger>
        // 设置当按钮不能用时，将圆形的填充色设置为灰色
        <Trigger Property="IsEnabled" Value="False">
            <Setter TargetName="outerCircle" Property="Fill"
                Value="Gray">
            </Setter>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
</Grid.Resources>
<StackPanel>
    <Button Width="100" Height="100"
        Template="{StaticResource buttonTemplate}" Click="Button_Click"
        Content="OK" Background="CadetBlue" BorderBrush="BurlyWood"/>
    <TextBlock Text="IsEnable = True"
        HorizontalAlignment="Center"/>
    <Button Width="100" Height="100"
        Template="{StaticResource buttonTemplate}" Click="Button_Click"
        Content="Disabled" IsEnabled="False"/></Button>

```

```
<TextBlock Text="IsEnable = False"  
HorizontalAlignment="Center"/>  
</StackPanel>  
</Grid>  
</Window>
```

代码 13-17 完善后的代码

13.3.1 模板绑定和模板触发器

完善后的代码使用 TemplateBinding（模板绑定）绑定 Button 的 Content 和 ContentPresenter 属性，为 Button 的 Content 属性赋值实际上是在 ContentPresenter 的 Content 属性赋值（代码 3-17③处）。

模板绑定（TemplateBinding）类似一般的数据绑定，其功能并不强大，但是方便使用。与一般的绑定相比，它有如下限制。

- (1) 仅在模板的可视化树内部有效，在模板外部，甚至模板的 triggers 中也无效。
- (2) 不能应用在 Freezable 派生对象的属性上，如果尝试绑定 Brush 的 Color 属性，则会失败。

上例中有两处使用常规的数据绑定，一是在 Ellipse 的 Fill 属性中。这是因为 Color 属于 Freezable 派生对象的属性，不能使用模板绑定（代码 3-17 处①处）。

二是在 Triggers 中对 IsMouseOver 的处理，这是因为 Trigger 不属于控件模板的可视化树内部，因此使用模板绑定无效（代码 3-17④处）。

模板和样式触发器比较类似，但是仍然有区别：

- (1) 样式的触发器无法应用于模板的某个子元素，而模板的触发器可以。如上例中可以在 IsMouseOver 为 True 时设置第 1 个圆的 Fill 属性，而在样式中只能设置整个控件的某个属性。Setter 的 TargetName 和 Trigger 的 SourceName 属性均用来指定模板中的某个子元素，该子元素必须有一个名字。
- (2) 样式的触发器的优先级高于模板的触发器（第 5 章 5.4 节）。

13.3.2 其他修改

“除去模板绑定和模板的触发器以外，其他的改进都是小的技巧。”孙工嘿嘿一笑：“比如不再具体设置圆形的长度和宽度，只是指定内圆的大小是外圆的 0.8 倍这样的相对数值，便于控件模板的复用（代码 13-17②处）。IsEnable 为 False 时，将按钮变成灰色来标示不可用（代码 13-17⑤处）。”运行程序，结果如图 13-10 所示。

“其实创建一个成熟可用的模板并不是那么容易。”孙工感慨到：“当你改变了控件的外观时，你会发现新创建的模板又会丧失很多的特性。”“那有没有办法，能够查看到现有的模板的代码呢？以此作为参考岂不是很好？”木木寻思到。“这种想法确实很好，我们也想到了。我们庄子里就有

人写了个程序，可以查看各种控件模板的 XAML 文件。”“喔！能否请那位高人出来一见。”虽然木木已经知道是一个人所办，但是并不道破。孙工脸上颇显为难之色，说道：“这也容易，不过……写该程序的人乃是我们庄子里的老夫人，老夫人今年 70，你要向她老人家请教，得磕上好几个响头才行。”木木起先看到孙工脸上难色，心中颇为紧张。听到孙工说完，提上的心才放下来，舒了一口气说道：“这个不碍事，这本来是我们年轻晚辈的本分。”“那好，我这就请老夫人出来。”孙工说着走进内堂。



图 13-10 完善后的程序运行结果

13.4 控件模板的浏览器程序——龙钟老太太，妙龄俏阿朱

过了好一会，只听得佩环叮当，内堂走出一位老夫人来，人未到，那淡淡的幽香已先传来。老夫人走到前面，木木鞠了一躬，说道：“老夫人，有句话想和您说。”老夫人问道：“你说什么？”木木道：“我有一个侄女儿，最是聪明伶俐不过，可是却也顽皮透顶。她最爱扮小猴儿玩，今天扮公的，明儿扮母的，还会变把戏呢。老太太见了她一定欢喜。可惜这次没带她来向你老人家磕头。”

这老夫人正是以易容术见长的阿朱姑娘，先前的黄老头和孙工皆为她所扮。阿朱听木木这么一说，吃了一惊，但丝毫不动声色，仍是一副老态龙钟、耳聋眼花的模样，说道：“乖孩子，孙工和你说过见我老婆婆的规矩了么？”

木木一拍脑袋，想到还差点把这回事给忘了，于是咚咚磕了十几个响头，震得山响。阿朱十分欢喜，心道：“他明知我是个小丫头，居然还肯向我磕头，当真十分难得。”连忙扶起木木。仍然颤颤巍巍坐到电脑旁，说道：“那老婆子，就给这位公子爷看看如何写一个程序，可以查看各种控件模板的 XAML 文件。”说完老夫人边说边开始写控件模板的浏览器程序（详见 `mumu_controltemplatebrowser` 工程）：

```
<Window x:Class="mumu_controltemplatebrowser.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:cm="clr-namespace:System.ComponentModel;assembly=System"
    Title="ControlTemplate 浏览器" Height="544" Width="713" Loaded="Window_Loaded">
    <Grid>
        <Grid.RowDefinitions>
```

```

<RowDefinition Height="*"/>
<RowDefinition Height="20"/>
</Grid.RowDefinitions>
<Grid Grid.Row="0" Name="grid" >
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/></ColumnDefinition>
        <ColumnDefinition Width="2"/></ColumnDefinition>
        <ColumnDefinition Width="3*"/></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <GridSplitter Grid.Column="1" ResizeDirection="Columns"
VerticalAlignment="Stretch" Width="2" Background="Black" HorizontalAlignment="Center"
ShowsPreview="True"/>
        <TreeView DisplayMemberPath="Name" Name="lstTypes"
SelectedItemChanged="lstTypes_SelectedItemChanged"></TreeView>
        <TextBox Grid.Column="2" Name="txtTemplate" TextWrapping="Wrap"
VerticalScrollBarVisibility="Visible" FontFamily="Consolas"></TextBox>
    </Grid>
    <TextBlock x:Name="txtbar" Grid.Row="1" Height="18"
HorizontalAlignment="Left" Margin="10,0,0,0" Text="Wait"></TextBlock>
</Grid>
</Window>

```

代码 13-18 代码之一： MainWindow.xaml

程序布局如图 13-11 所示，主要由左侧的一个树、右侧的一个 TextBox 和下方的一个 TextBlock 组成。树用来展示 Control 及其派生类的层次结构；右侧的 TextBox 则显示控件的默认控件模板；下方则是一个传统的状态栏。老夫人还特意在树控件和 TextBox 之间加上了一个 GridSplitter 便于改变左右侧控件的大小尺寸。如图 13-11 所示。

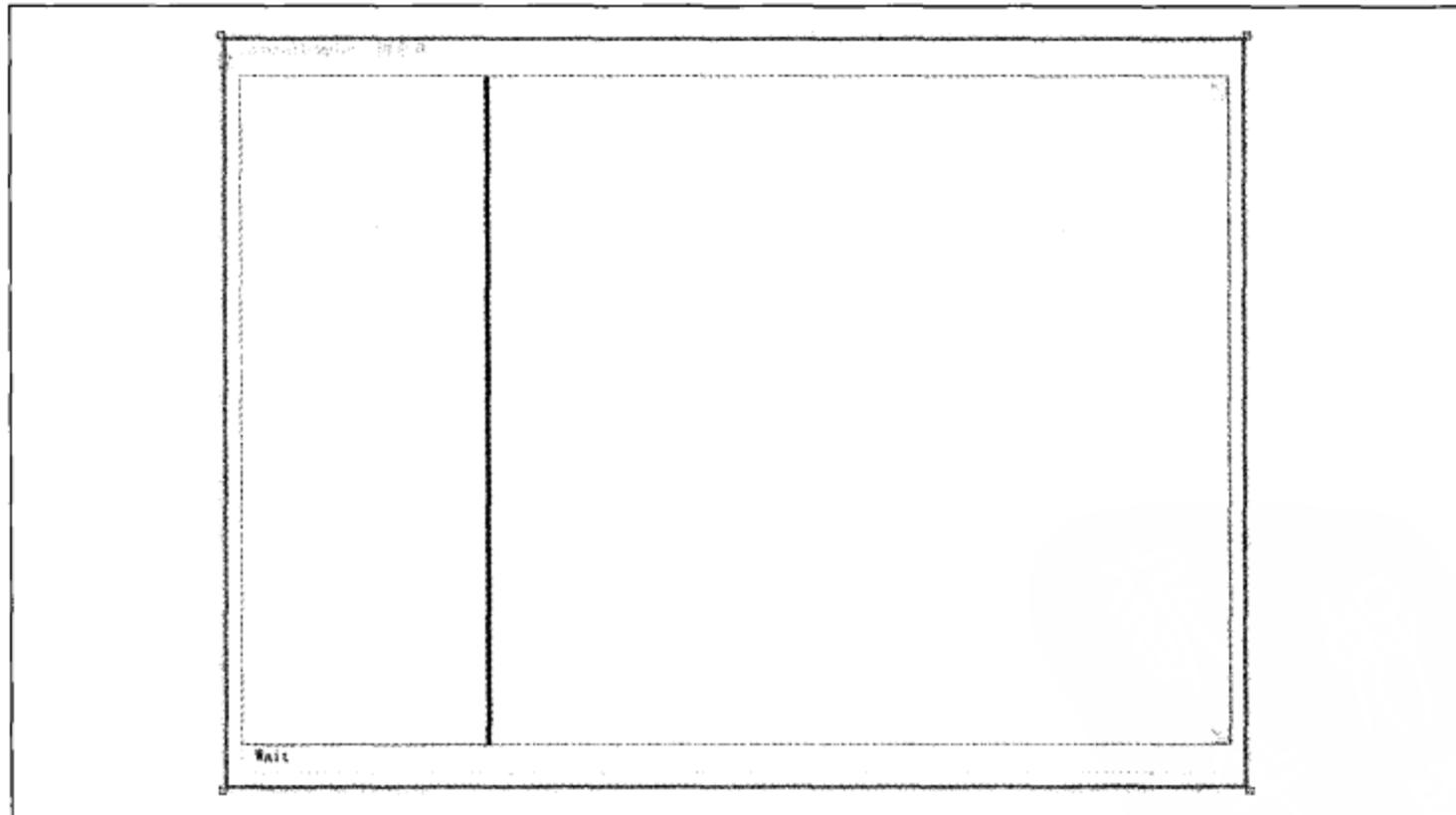


图 13-11 程序布局

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Reflection;
using System.Windows.Markup;
using System.Xml;

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

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // 获得 Control 的程序集
            Assembly asbly = Assembly.GetAssembly(typeof(Control));

            // 获得该程序集中的所有类型
            Type[] atype = asbly.GetTypes();

            // 使用该列表存储
            SortedList<string, TreeViewItem> sortlst =
                new SortedList<string, TreeViewItem>();

            TreeViewItem item = new TreeViewItem();
            item.Header = "Control";
            item.Tag = typeof(Control);
            sortlst.Add("Control", item);
            lstTypes.Items.Add(item);

            // 遍历所有的类型，然后将派生自 control 的类型添加到列表中
            foreach (Type typ in atype)
            {
                if (typ.IsPublic && (typ.IsSubclassOf(typeof(Control))))
                {
                    item = new TreeViewItem();
                    item.Header = typ.Name;
                    item.Tag = typ;
                    sortlst.Add(typ.Name, item);
                }
            }

            // 构建树
            foreach (KeyValuePair<string, TreeViewItem> kvp in sortlst)
            {
                if (kvp.Key != "Control")
                {
                    string strParent = ((Type)kvp.Value.Tag).BaseType.Name;
                    TreeViewItem itemParent = sortlst[strParent];
                    itemParent.Items.Add(kvp.Value);
                }
            }
        }
    }
}
```

```
        }

    }

    private void lstTypes_SelectedItemChanged(object sender,
RoutedPropertyChangedEventArgs<object> e)
{
    try
    {
        Cursor oldcur = this.Cursor;
        this.Cursor = Cursors.Wait;

        // 获得选中的类型
        TreeViewItem selectedItem = (TreeViewItem)lstTypes.SelectedItem;
        if (selectedItem.HasItems)
        {
            this.Cursor = oldcur;
            return;
        }
        Type type = (Type)selectedItem.Tag;

        // 实例化该 type
        ConstructorInfo info =
type.GetConstructor(System.Type.EmptyTypes);
        Control control = (Control)info.Invoke(null);

        // 添加该控件 但是将属性状态设置为 Collapsed.
        control.Visibility = Visibility.Collapsed;
        grid.Children.Add(control);

        // 获得模板
        ControlTemplate template = control.Template;

        // 获得模板的 XAML 文件
        XmlWriterSettings settings = new XmlWriterSettings();
        settings.Indent = true;
        StringBuilder sb = new StringBuilder();
        XmlWriter writer = XmlWriter.Create(sb, settings);
        XamlWriter.Save(template, writer);

        // 显示模板
        txtTemplate.Text = sb.ToString();

        txtbar.Text = type.Name + "Control Template";
        // 移出该控件
        grid.Children.Remove(control);

        this.Cursor = oldcur;
    }
    catch (Exception err)
    {
        .
        txtTemplate.Text = "<< Error generating template: " + err.Message + ">>";
    }
}
}
```

代码 13-19 代码之二： MainWindow.xaml.cs

在窗口的 Loaded 事件中构造 Control 及其 Control 的派生类的层次树，在树上选中某一项，右边则显示其控件模板。lstTypes_SelectedIndexChanged 函数即选中树上某一项的响应函数，其中的一个小技

巧是刚刚创建控件时其模板属性值为 null。因此将控件添加到 Grid 中，并将其 Visibility 属性设置为 Collapsed。使其既不可见，又不占用 Grid 的位置。然后获取 Template 属性，之后便将控件移除。

木木本来就对 grid.Children.Add(control) 和 grid.Children.Remove(control) 这两句话就有些疑问，听老夫人这么一说才明白。木木还注意到在处理写控件模板的 XAML 文件时将缩进设置为 true (settings.Indent = true;) 便于阅读。此外在处理数据之前，将光标设置为等待状，处理完之后再恢复箭头状的鼠标。不由暗暗佩服老夫人的这种细微之处见长的代码风格。

老夫人笑笑运行了这个控件模板浏览器程序，说到：“以后自定义一个控件模板就可以参考默认的控件模板了。”



图 13-12 运行结果

木木也笑了说到：“姑娘的易容术好生高明，也需要一个浏览器来参考参考呢。”老夫人一惊随即露出了少女般的微笑：“公子稍等，我卸下妆来再陪公子讨论这控件模板。”

过了一会儿，从内堂走出来个身穿淡绛纱衫的女郎，盈盈二十四五年纪，一脸精灵顽皮的神气。这正是阿朱本来面目。阿朱出来便问：“公子怎么能看出破绽？”

木木笑到：“阿朱姑娘，你说按钮我们可以通过控件模板定义不同的外观，为啥无论怎么变，我们都能看出它是个按钮。”

阿朱到：“它变的是外观，但是有不变的特性，比如点击都会响应 Click 事件。”

木木笑到：“姑娘千变万化，但是姑娘身上不变的是一股淡淡好闻的香味。”说完阿朱姑娘满脸羞红……

13.5 样式、模板和换肤——阿朱技高超，木木向来痴

13.5.1 混合使用

模板可以作为资源放在窗口或者应用程序的资源标签（Resources）中，如代码 13-20 所示（详见 `mumu_stylecontroltemplate` 工程）。

```
<Application.Resources>
    <ControlTemplate x:Key="buttonTemplate" TargetType="{x:Type Button}">
        .....
    </ControlTemplate>
    <Style x:Key="GreenRedGradientButtonStyle" TargetType="{x:Type Button}">
        <Setter Property="Template" Value="{StaticResource buttonTemplate}"/>
        <Setter Property="Background" Value="Green"/>
    </Style>
</Application.Resources>
```

代码 13-20 修改后的程序

“为什么好端端的需要将模板放在样式里呢？”木木问到。

阿朱心里暗暗赞叹，从木木刚一进屋，阿朱就不禁对这年轻人充满好感，倒不是因为相貌，而是憨态可掬，并且喜欢思考。过去阿朱也曾和别人讲过这样的问题，但是很多人只是知道如何用，却未曾像木木这样多问几个为什么，而这种为什么时常会触动到程序设计者的思维的本源，而这又恰恰是精髓之所在。

“你说呢？”阿朱反问到。

木木思考了一下，说到：“姑娘，我确实天资愚钝，只想到在这个例子里如果将模板用在样式里，则所有按钮都会自动应用了该模板，但是不放在样式里，则没有这样的效果。”阿朱听了略微点头，说到：“公子，你悟性其实已经颇高了，这确实是一个原因。不过还有一个更为重要的原因。这种做法可以让模板能够得到更好的复用。”

假如我们希望按钮的背景色变成绿色，一种办法是在模板里进行硬编码。但这样的做法，会使得模板的复用性不高，下次我们需要一个蓝色的背景色按钮，这样小小的一点变动也需要重新做一个模板。另外一种办法是模板不修改，而在每个控件里设置。两种做法复用性都不强。但是这种模板和样式的混用，模板不需要修改，只需要增加一个样式即可。如代码 13-21 所示。

```
<Style x:Key="GreenRedGradientButtonStyle" TargetType="{x:Type Button}">
    <Setter Property="Template" Value="{StaticResource buttonTemplate}"/>
    <Setter Property="Background" Value="Green"/>
</Style>
<Style x:Key="Blue RedGradientButtonStyle" TargetType="{x:Type Button}">
    <Setter Property="Template" Value="{StaticResource buttonTemplate}"/>
    <Setter Property="Background" Value="Blue"/>
</Style>
```

代码 13-21 不修改模板，只是增加一个样式

13.5.2 组织模板资源和更换皮肤

这些天来木木一直和阿朱姐学习模板，自己也做了很多自定义的模板，但是做的模板资源老是摆放杂乱，而阿朱姐却是井然有序。于是木木又带着这个问题来到了听香水榭。

阿朱听了木木的问题，笑到：“过去我几乎为每个控件都做过模板，当然也遇到和你一样的问题。好的组织方式怎么说呢？”阿朱顿了顿：“Keeping these together in one place, but separate from other controls, is good organization.”

“意思是好的组织方式是把所有的模板资源都放在同一个地方，但是按照不同控件进行分类存放。”阿朱继续往下说：“MSDN 当中的一个例子 simplestyles^[2]就体现了这样的思想。”

该示例将所有的控件模板放在一个 Resources 子目录下，如图 13-13 所示。

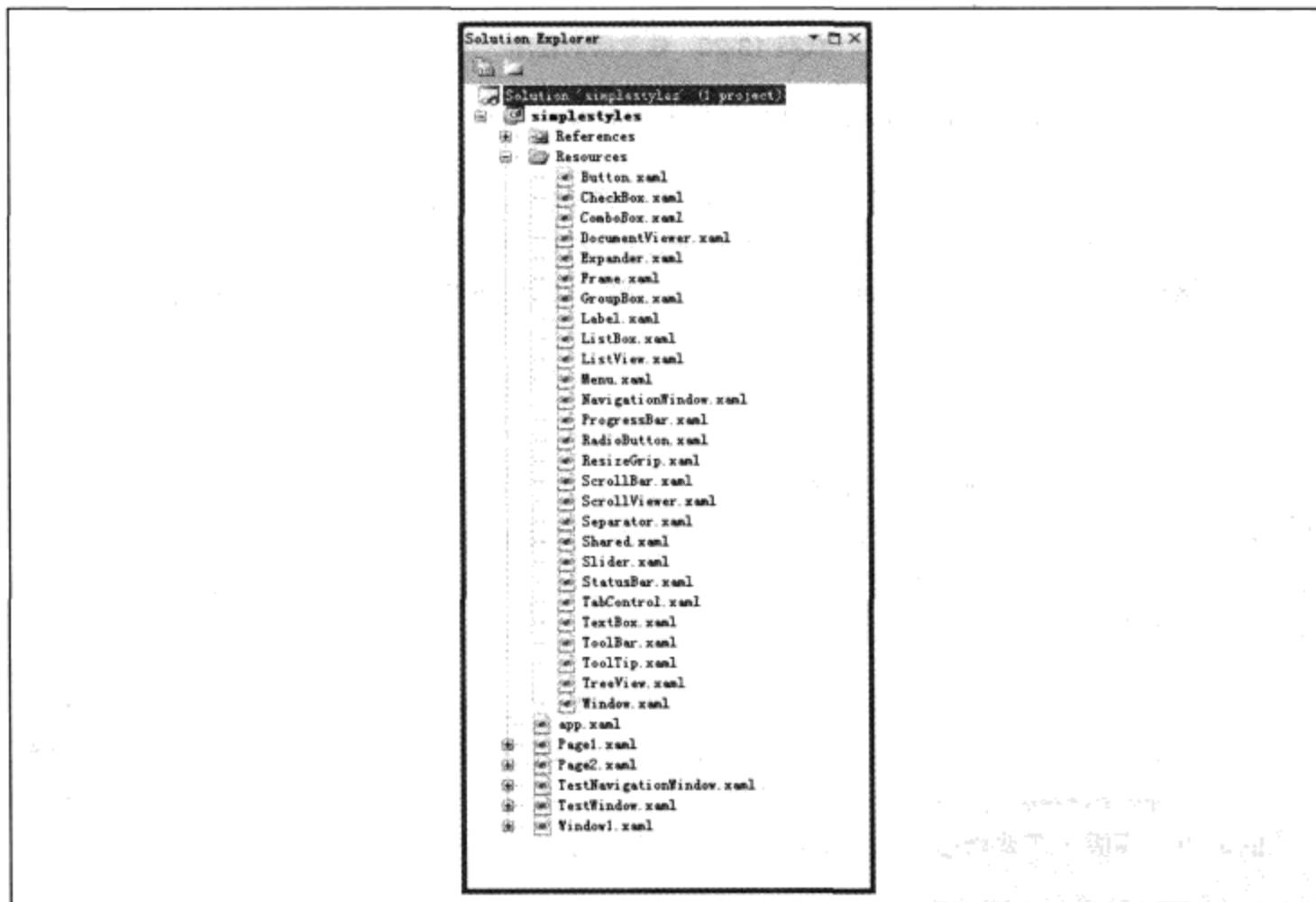


图 13-13 资源组织方式

其中的 Shared.xaml 文件保存一些共用资源（如画刷或者一些小的共用控件），其他文件为按照各个控件分类组织的模板资源。比如 Button.xaml 文件就是按钮的控件模板资源文件。

在 App.xaml 文件当中通过 ResourceDictionary.MergedDictionaries 标签页来引用这些文件，如代码 13-22 所示。

```
<Application x:Class="SimpleStyles.app"
```

```

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml"
>
<Application.Resources>
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources\Shared.xaml" />
        <ResourceDictionary Source="Resources\Button.xaml" />
        <ResourceDictionary Source="Resources\CheckBox.xaml" />
        <ResourceDictionary Source="Resources\RadioButton.xaml" />
        <ResourceDictionary Source="Resources\ComboBox.xaml" />
        <ResourceDictionary Source="Resources\ListBox.xaml" />
        <ResourceDictionary Source="Resources\Menu.xaml" />
        <ResourceDictionary Source="Resources\ScrollBar.xaml" />
        <ResourceDictionary Source="Resources\Expander.xaml" />
        <ResourceDictionary Source="Resources\GroupBox.xaml" />
        <ResourceDictionary Source="Resources\Label.xaml" />
        <ResourceDictionary Source="Resources\TabControl.xaml" />
        <ResourceDictionary Source="Resources\ProgressBar.xaml" />
        <ResourceDictionary Source="Resources\Slider.xaml" />
        <ResourceDictionary Source="Resources\TextBox.xaml" />
        <ResourceDictionary Source="Resources\ToolTip.xaml" />
        <ResourceDictionary Source="Resources\TreeView.xaml" />
        <ResourceDictionary Source="Resources\Frame.xaml" />
        <ResourceDictionary Source="Resources\ListView.xaml" />
        <ResourceDictionary Source="Resources\ScrollViewer.xaml" />
        <ResourceDictionary Source="Resources\ToolBar.xaml" />
        <ResourceDictionary Source="Resources\StatusBar.xaml" />
        <ResourceDictionary Source="Resources\DocumentViewer.xaml" />
        <ResourceDictionary Source="Resources\Window.xaml" />
        <ResourceDictionary Source="Resources\NavigationWindow.xaml" />
    </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
</Application>

```

代码 13-22 app.xaml

程序运行结果如图 13-14 所示。

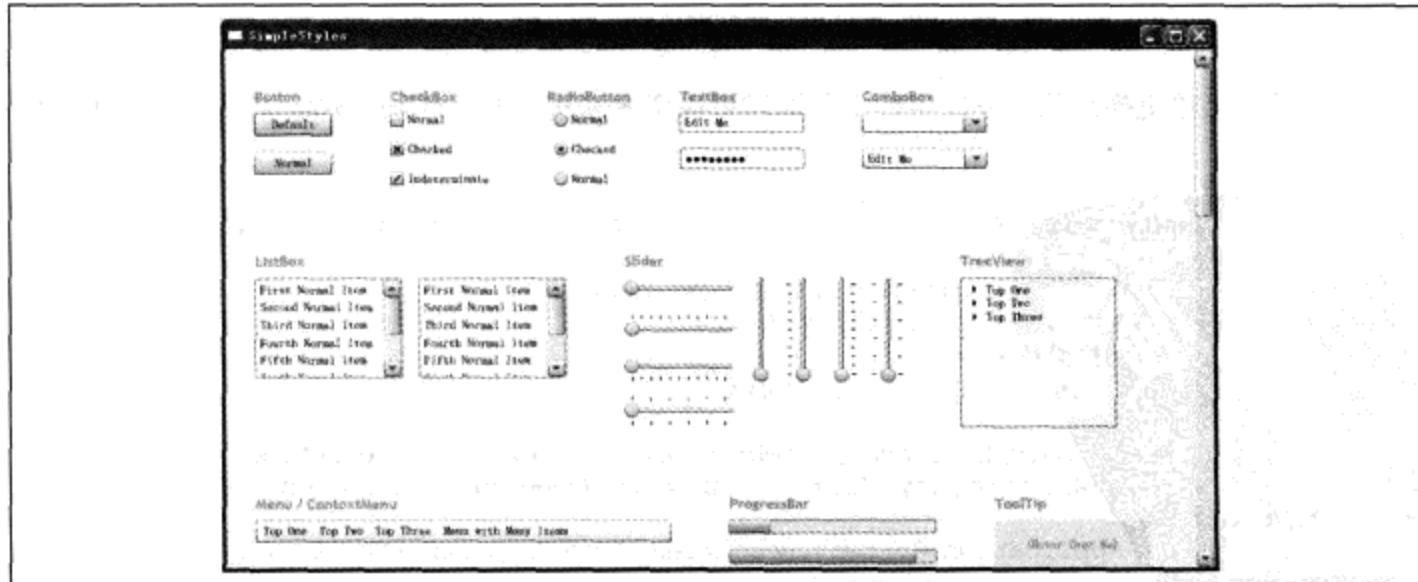


图 13-14 程序运行结果

换肤即改变应用程序的外观（或皮肤），在 WPF 中没有皮肤这样的概念，但是可以通过样式和模板来实现。细心的阿朱已经将前面按钮的示例综合整理为一个换肤的示例（mumu_userselectskin 工程），在该例的 Resources 文件夹下有两个皮肤文件，其中 Button.xaml 和 Shared.xaml 文件构成了一个简单按钮风格；fancyButton.xaml 文件构成了个性化按钮风格。程序刚开始运行时，默认使用简单按钮风格。因此在 App.xaml 文件当中引用的资源文件是 button.xaml 文件，如代码 13-23 和代码 13-24 所示。

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Resources/button.xaml"></ResourceDictionary>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

代码 13-23 App.xaml

```
<Window x:Class="mumu_userselectskin.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="换肤程序" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Menu Grid.Row="0" VerticalAlignment="Top">
            <MenuItem Header="皮肤" Click="MenuItem_Click" IsChecked="True">
                <MenuItem Header="简单按钮风格" Name="simpleSkin" Click="MenuItem_Click" IsChecked="True"/>
                <MenuItem Header="个性化按钮风格" Name="fancySkin" Click="MenuItem_Click" IsChecked="False"/>
            </MenuItem>
        </Menu>
        <Button Grid.Row="1" Content="OK" VerticalAlignment="Center" HorizontalAlignment="Center"/>
    </Grid>
</Window>
```

代码 13-24 MainWindow.xaml

程序可以通过菜单更换皮肤。皮肤实际上是通过更换资源来实现的，代码 13-25 是菜单的点击事件的处理函数。

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    MenuItem menuItem = (MenuItem)sender;
    if(menuItem == null) return;
    // 切换简单风格
    if (menuItem == simpleSkin)
    {
        ResourceDictionary newDictionary = new ResourceDictionary();
        newDictionary.Source = new Uri("Resources/button.xaml", UriKind.Relative);
        Application.Current.Resources.MergedDictionaries[0] = newDictionary;
        menuItem.IsChecked = true;
        fancySkin.IsChecked = false;
    }
}
```

```
// 切换个性风格
else if (menuItem == fancySkin)
{
    ResourceDictionary newDictionary = new ResourceDictionary();
    newDictionary.Source = new Uri("Resources/fancyButton.xaml",
UriKind.Relative);
    Application.Current.Resources.MergedDictionaries[0] = newDictionary;
    menuItem.IsChecked = true;
    simpleSkin.IsChecked = false;
}
}
```

代码 13-25 MainWindow.xaml.cs

程序运行结果如图 13-15 所示。

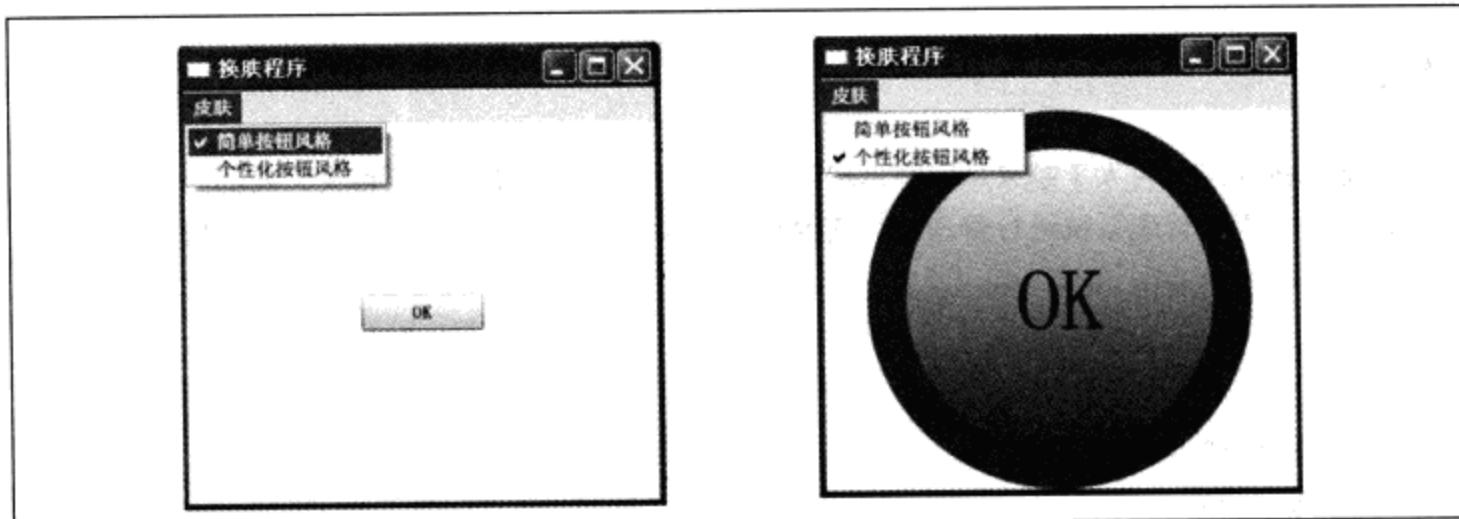


图 13-15 程序运行结果

13.6 接下来做什么

样式和控件模板都是改变控件外观的有力手段。在第 20 章里我们还会借助控件模板来改变自定义控件的外观。接下来是 WPF 关于数据的一个重要特性，名为数据绑定。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作，《金庸全集典藏版 天龙八部》，“第 11 章向来痴”。
- [2] MSDN Library for Visual Studio 2008 SP1 Styling with ControlTemplates Sample。

数据绑定——桃花岛软件公司人员管理系统之始末

缘起

话说药师的桃花岛公司人丁越来越兴旺，已经由原来的几十号人变成了上百人。药师一琢磨，公司还是需要一套人员管理系统进行管理的。交给谁做呢？药师一想，不妨让木木尝试着做做，也好试试他几斤几两……

本章内容如下。

- (1) 人员管理系统。
- (2) 数据绑定基础。
- (3) 高级主题——与数据集合绑定。
- (4) 后记。
- (5) 接下来做什么。

14.1 人员管理系统

14.1.1 浏览和修改人员信息（无数据绑定）

木木接到这个任务，顿时傻眼，不知从何下手。走一步看一步吧。于是木木首先定义了一个最简单的人员类（Person），如代码 14-1 所示（详见 `mumu_withoutdatabinding` 工程）。

```
public class Person
{
    // 姓名属性
    string name;
    public string Name
    {
        get { return this.name; }
        set
        {
            if (this.name == value) { return; }
            this.name = value;
        }
    }
}
```

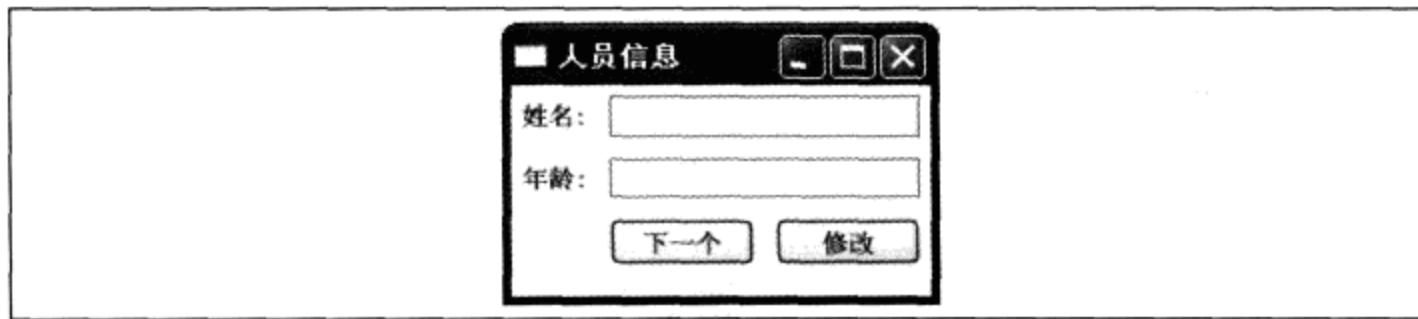
```

// 年龄属性
int age;
public int Age
{
    get { return this.age; }
    set
    {
        if (this.age == value) { return; }
        this.age = value;
    }
}
// 构造函数
public Person() { }
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
}

```

代码 14-1 定义的一个人员类

紧接着，设计了一个“人员信息”窗口，如图 14-1 所示。



■ 14-1 “人员信息”窗口

在其中可以通过“下一个”按钮来依次浏览人员信息；通过“修改”按钮修改当前人员信息，XAML 文件的内容如代码 14-2 所示。

```

<Window x:Class="mumu_withoutdatabinding.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="人员信息" Height="135"
    Width="200" >
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Row="0" Grid.Column="0" Margin="5" VerticalAlignment="Center">
姓名:</TextBlock>
        <TextBox Name="nameTextBox" Grid.Row="0" Grid.Column="1" Margin="5" />

```

```

<TextBlock Grid.Row="1" Grid.Column="0" Margin="5"
VerticalAlignment="Center">年龄:</TextBlock>
<TextBox Name="ageTextBox" Grid.Row="1" Grid.Column="1" Margin="5" />
<Grid Grid.Row="2" Grid.Column="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button Name="nextButton" Grid.Row="0" Grid.Column="0" Margin="5" >
下一个</Button>
    <Button Name="modifyButton" Grid.Row="0" Grid.Column="1" Margin="5" >
修改</Button>
    </Grid>
</Grid>
</Window>

```

代码 14-2 窗口代码

人员信息保存在类型为 Person 的数组中，在窗口的构造函数中添加 3 条样本数据。curIndex 为人员信息记录的索引号，初始值为 0。“人员信息”窗口中的初始值为数组中的第 1 条数据，如代码 14-3 所示。

```

public partial class MainWindow : Window
{
    Person[] persons = null;
    int curIndex = 0;
    public MainWindow()
    {
        InitializeComponent();

        // 数据集
        persons = new Person[3];
        persons[0] = new Person("木木", 22);
        persons[1] = new Person("黄药师", 48);
        persons[2] = new Person("黄蓉", 20);
        curIndex = 0;
        this.nameTextBox.Text = persons[curIndex].Name;
        this.ageTextBox.Text = persons[curIndex].Age.ToString();
    }
}

```

代码 14-3 添加初始记录

为“下一个”按钮添加事件处理函数，如代码 14-4 所示。

```

MainWindow.xaml
<Button Name="nextButton" Grid.Row="0" Grid.Column="0" Margin="5"
Click="nextButton_Click">下一个</Button>

```

代码 14-4 为“下一个”按钮添加事件处理函数

每次单击“下一个”按钮，curIndex 自动加 1。如果超过数组大小，则重新从 0 开始，如代码 14-5 所示。

```

MainWindow.xaml.cs
private void nextButton_Click(object sender, RoutedEventArgs e)
{
    curIndex++;
}

```

```
    if (curIndex >= 3) curIndex = 0;
    this.nameTextBox.Text = persons[curIndex].Name;
    this.ageTextBox.Text = persons[curIndex].Age.ToString();
}
```

代码 14-5 curIndex 自动计数

同样要在 XAML 文件中为“修改”按钮添加 Click 事件处理函数，其中可以将 TextBox 中的文本直接赋给当前 Person 的 Name 属性作为姓名。但是年龄必须由字符串转换为整数，通过 int 的一个静态方法 TryParse 实现。如果转换成功，则将转换的结果赋给第 2 个输出参数，否则返回 false，如代码 14-6 所示。

```
private void modifyButton_Click(object sender, RoutedEventArgs e)
{
    persons[curIndex].Name = nameTextBox.Text;
    int age = 0;
    if (int.TryParse(ageTextBox.Text, out age))
    {
        persons[curIndex].Age = age;
    }
}
```

代码 14-6 为修改按钮添加 Click 事件处理函数

“修改”按钮默认情况设置为禁用，只有文本框内容发生改变时才可以启用。因此在构造函数和“修改”按钮的 Click 事件处理函数中将该按钮的 IsEnable 属性设置为 false。在 TextBox 的 TextChanged 事件处理函数和“下一个”按钮的 Click 事件处理函数中将 IsEnable 属性设置为 true，如代码 14-7 所示。

```
public MainWindow()
{
    .....
    modifyButton.IsEnabled = false;
}

private void nameTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    modifyButton.IsEnabled = true;
}

private void ageTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    modifyButton.IsEnabled = true;
}

private void nextButton_Click(object sender, RoutedEventArgs e)
{
    .....
    modifyButton.IsEnabled = false;
}
private void modifyButton_Click(object sender, RoutedEventArgs e)
{
    .....
    modifyButton.IsEnabled = false;
}
```

代码 14-7 设置“修改”按钮属性

14.1.2 数据绑定（木木，老婆喊你回家吃饭）

木木还要接着往下做，这时“老婆，老婆我爱你”的手机铃声响起。不用说是老婆蓉儿的电话。

“喂，木木怎么还没下班啊？”木木一看表已经7点，自己居然做着忘记掉了时间。说到：“喔，爸爸给我安排了一个任务，我做着做着忘记时间了。”

“唉~”黄蓉小声地叹了一口气，她知道木木的这股痴劲。问到：“做一个什么东西？”

木木知道老婆才智远甚于他，于是就悉数告诉黄蓉。

黄蓉听后，沉默片刻，说到：“木木，你现在做的工作概括起来就是将数据显示在用户界面上，然后通过用户界面进行修改，从而再存储到数据结构里面。对吧？”

木木想想的确如此，于是说到：“确实是这样的。”

黄蓉接着说：“其实程序员平常大部分的工作都是这样，尤其是这种人员管理系统。因此WPF提供了一种数据绑定技术，来简化这类工作。简单地说，数据绑定就是将控件连接到数据的技术。它可以很简单，比如将一个 CheckBox 连接到一个 Boolean 变量，也可以很复杂，将一个数据库绑定到一个数据面板上。”

木木说到：“老婆……”

还未等木木话说完，黄蓉轻叹一口气说：“我知道你在想什么，笔记本我的文件夹下有一个数据绑定的简单例子，看明白了，赶紧回家吃饭！”

木木大喜：“好的，蓉儿。”

14.1.3 使用数据绑定

黄蓉的例子和木木所做非常相似（详见mumu_rongerwithdatabinding工程），但是在两点上考虑得更加周全。

(1) 检查年龄，如果年龄小于0岁或者大于128岁，或者是字母，则将文本框的边框变为红色，以示警示。同时提示错误的原因，如图14-2所示。

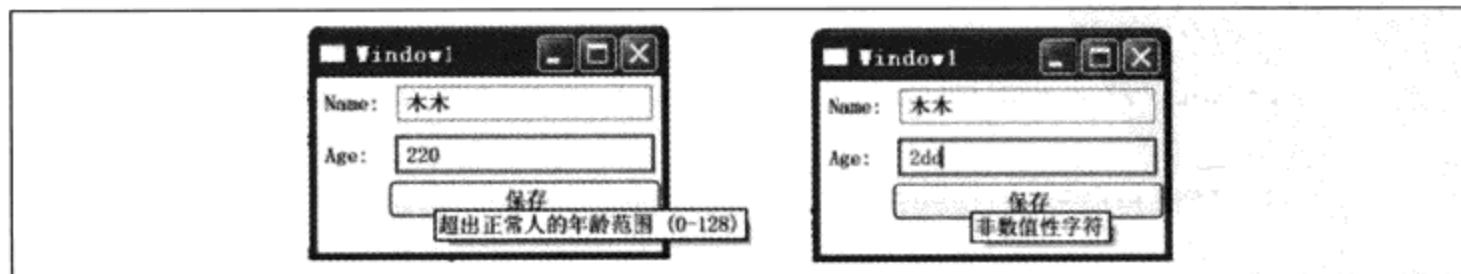


图 14-2 年龄的合法性检查

(2) 如果录入的年龄数值小于 18 岁, 文本的字体颜色会变为红色。提示该年龄属于未成年人, 如图 14-3 所示。



图 14-3 年龄是否达到成年检查

该例采用数据绑定将数据源和两个文本框控件关联起来, 数据源就是一个 Person 对象。初始的 Name 属性为“木木”, 年龄为“22”。并采用资源方式保存, 如代码 14-8 所示。

```
xmlns:local = "clr-namespace:mumu_rongerwithdatabinding"
.....
<Window.Resources>
    <local:Person x:Key="mumu" Name="木木" Age="22" />
.....
</Window.Resources>
```

代码 14-8 采用资源方式保存

采用代码 14-9 语法关联两个文本框的 Text 属性和数据源的 Name 和 Age 属性。

```
.....
<TextBox x:Name="nameTextBox" Grid.Row="0" Grid.Column="1" Margin="5"
Text="{Binding Source={StaticResource mumu}, Path=Name}" />
.....
<TextBox Grid.Row="1" Grid.Column="1" Margin="5" >
    <TextBox.Text>
        <Binding Path="Age" UpdateSourceTrigger="PropertyChanged"
Source="{StaticResource mumu}">
            </Binding>
        </TextBox.Text>
    </TextBox>
```

代码 14-9 关联属性

数据绑定同样属于一种标记扩展 (mark up extension), 在第 1 个文本框的 Text 属性中指定需要绑定的数据源是资源 mumu, 绑定的属性为 Name 属性; 第 2 个文本框采用了 Property 语法, 指定 Text 属性绑定资源 mumu 的 Age 属性。UpdateSourceTrigger 指定文本框更新绑定数据源的条件, PropertyChanged 表示只要文本框中的文本发生变化即更新数据源。

此外该例还新建了两个类, 即 AgeToForegroundConverter 和 NumberRangeRule 分别用于检查年龄值的合法性和提示是否为未成年人。木木一时也不明白这两个类的原理, 天色已晚, 只好悻悻回家吃饭。

14.2 数据绑定基础

黄蓉的饭菜做得甚是可口。饭后，木木忍不住又想起了两个类。不禁问到：“蓉儿，有两个类，我大概能猜到它们的作用，但是我不明白其中的原理。”黄蓉莞尔，说到：“木木，不单单是那两个类，我们还得从原理上去了解数据绑定吧。”

14.2.1 数据绑定模型

数据绑定实际上是关联数据源和目标的一种方式，其中目标一般是应用程序的用户界面。数据源则可能是一个集合对象、一个 XML 文件、一个 Web 服务、一个数据表、一个自定义对象，甚至一个 WPF 元素，如 Button。当数据发生改变时，用户界面会自动反映该变化。

数据绑定的模型由 5 个部分组成，即目标对象、目标属性、数据源对象、数据源属性和绑定对象，如图 14-4 所示。

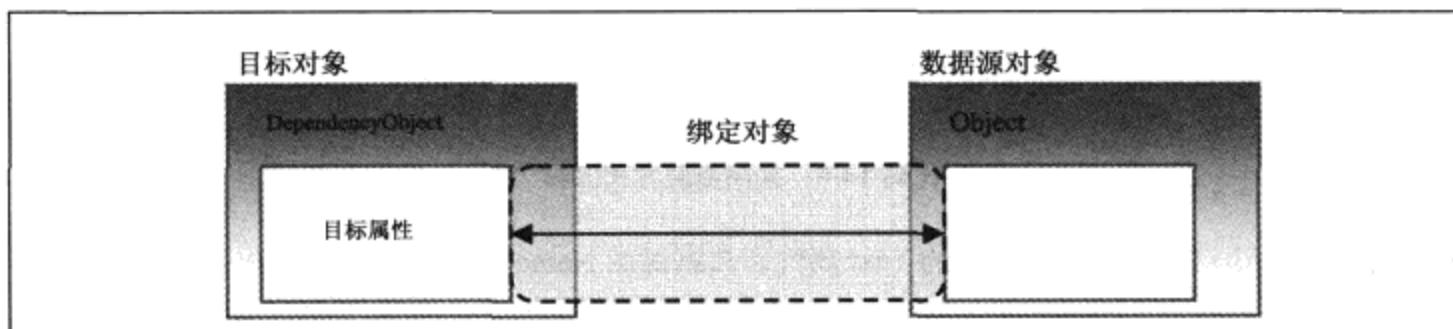


图 14-4 数据绑定模型

在上例中，TextBox 是目标对象，Text 属性是目标属性，而 Person 对象则是数据源对象。Age 和 Name 属性为数据源属性，它们通过 Binding 对象关联。

需要注意的是，目标属性必须是依赖属性，而数据源属性可以是普通或者依赖属性。

以上例中的第 1 个 TextBox 为例，设置绑定在 XAML 文件中实现，如代码 14-10 所示。

```
<TextBox Grid.Row="0" Grid.Column="1" Margin="5" Text="{Binding Source={StaticResource mumu}, Path=Name}" />
```

代码 14-10 在 XAML 文件中设置绑定

也可以在代码文件中设置数据绑定，如代码 14-11 所示。

```
Binding binding = new Binding("Name");
Person person = (Person)this.FindResource("mumu");
binding.Source = person;
nameTextBox.SetBinding(TextBox.TextProperty, binding);
```

代码 14-11 通过代码设置数据绑定

Binding 构造函数中指定了 Path 的值，通过 TextBox 的 SetBinding 方法关联目标属性和绑定对象。这里目标属性不能指定 Text 属性，而是直接指向与 Text 相关的依赖属性 TextProperty。也可以用代

码 14-12 方式关联目标属性和 Binding 对象。

```
BindingOperations.SetBinding(nameTextBox, TextBox.TextProperty, binding);
```

代码 14-12 关联目标属性和 Binding 对象

14.2.2 数据绑定的方向

TextBox 的 Text 属性根据 Person 对象的属性值（Age 和 Name）初始化并显示在文本框中，数据绑定的方向为从数据源对象到目标对象；另外一种情况是当用户在文本框中的修改结果保存在数据源对象中，数据绑定的方向为从目标对象到数据源对象。

数据绑定的方向基本上包括 OneWay、TwoWay 和 OneWayToSource 共 3 种情况，其中 OneWay 从数据源到目标对象，即目标对象随数据源对象改变而改变；OneWayToSource 则是从目标对象到数据源，即通过目标对象来更新数据源对象；TwoWay 则是双向，如图 14-5 所示。

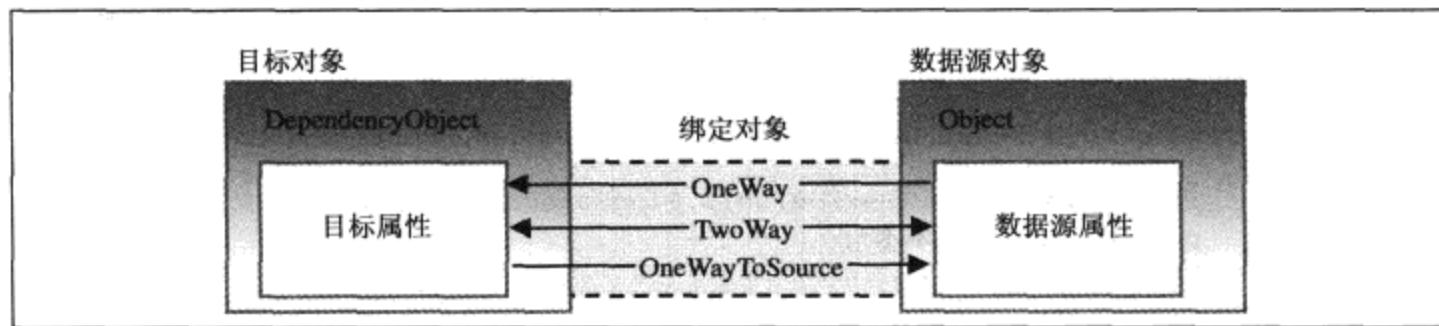


图 14-5 数据绑定的方向

通过 Binding 的 Mode 属性设置数据绑定的方向，除了可以设置上述 3 个值，该属性还可以设置为 OneTime 和 Default。其中 OneTime 可以理解成一次性的 OneWay，即目标根据数据源的值初始化后不会持续跟踪数据源的变化；Default 选项指 Binding 的 Mode 将和目标属性一致。一般说来，除了 TextBox 的 Text 和 CheckBox 的 IsChecked 属性是 TwoWay，而其他属性大多是 OneWay。Binding 的 Mode 属性默认为 Default。

要知道 Text 属性是 OneWay，还是 TwoWay，可以查阅 MSDN。如果该依赖属性支持 TwoWay，那么其依赖属性节中的元数据中会明确地说明 BindsTwoWayByDefault 值为 true；否则为 OneWay”，如图 14-6 所示。

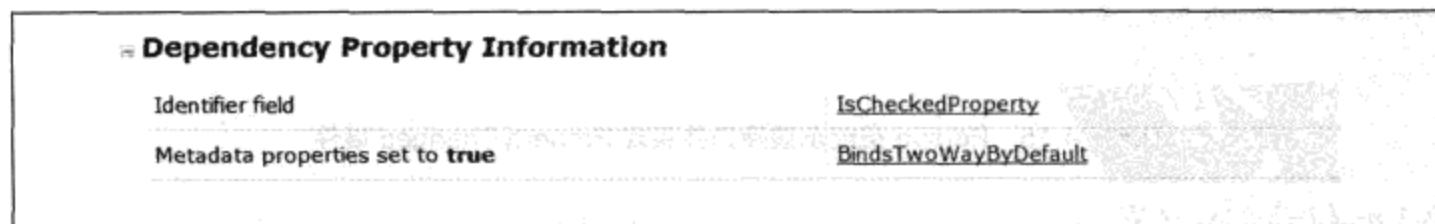


图 14-6 MSDN 中依赖属性的元数据信息

也可通过 GetMetaData 方法检查 BindsTwoWayByDefault 属性是否为 true。检查 TextBox 的 Text 属性是否为 TwoWay，如代码 14-13 所示。

```
FrameworkPropertyMetadata metadata = TextBox.TextProperty.GetMetadata(nameTextBox) as FrameworkPropertyMetadata;
bool res = metadata.BindsTwoWayByDefault;
```

代码 14-13 检查 TextBox 的 Text 属性是否为 TwoWay

1. INotifyPropertyChanged 接口

值得注意的是，虽然数据绑定的方向规定为 OneWay 或者 TwoWay，但是如果数据源没有一种机制通知目标，属性发生改变，那么目标也得不到相应的更新。通常解决这个问题有如下两种方法。

(1) 将数据源属性实现为依赖属性，由于大多数 WPF 元素的属性为依赖属性。因此这种方式多用于元素之间的绑定，如将一个文本块 (TextBlock) 的字体大小和滑块 (Slider) 的值绑定在一起。但是对于前面的 Person 这种类型，该方法不合适。

(2) 数据源实现 INotifyPropertyChanged 接口。

在前例中添加一个“重设”按钮，将 Person 对象重新设置为初始值，该按钮的单击事件处理函数如代码 14-14 所示。

```
private void ResetButton_Click(object sender, RoutedEventArgs e)
{
    Person person = (Person)this.FindResource("mumu");
    person.Name = "木木";
    person.Age = 22;
}
```

代码 14-14 “重设”按钮的单击事件处理函数

但是单击“重设”按钮后两个文本框无法恢复原默认值，原因在于文本框并不知道数据源发生了改变。Person 类需要实现 INotifyPropertyChanged 接口，第 1 步是让该类继承该接口并实现 PropertyChanged 事件，如代码 14-15 所示。

```
public class Person : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void Notify(string propName)
    {
        if (this.PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propName));
        }
    }
    .....
}
```

代码 14-15 Person 类继承该接口并实现 PropertyChanged 事件

第 2 步是在每次设置某一属性时调用 Notify 函数触发 PropertyChanged 事件，如代码 14-16 所示。

```
// 姓名属性
string name;
public string Name
{
    get { return this.name; }
```

```

set
{
    if (this.name == value) { return; }
    this.name = value;
    Notify("Name");
}
}

// 年龄属性
int age;
public int Age
{
    get { return this.age; }
    set
    {
        if (this.age == value) { return; }
        this.age = value;
        Notify("Age");
    }
}

```

代码 14-16 每次设置某一属性时调用 Notify 函数触发 PropertyChanged 事件

运行程序，当姓名修改为黄蓉且年龄修改为 20 后单击“重设”按钮，则恢复原状，如图 14-7 所示。

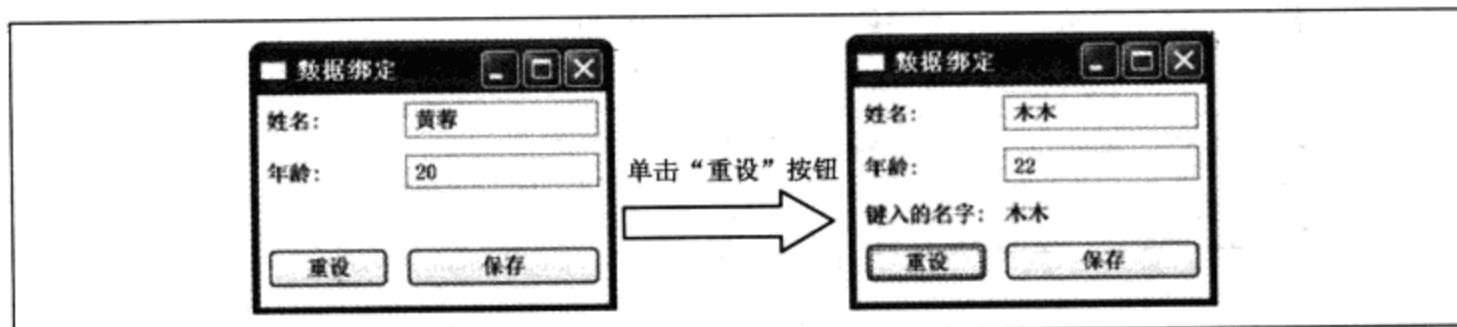


图 14-7 Person 对象重新设置为初始值

14.2.3 数据绑定的触发条件

接下来的问题是当用户在修改文本框中的文本后何时更新数据源？即在文本每次发生变化还是在文本框失去焦点时更新？Binding 的 UpdateSourceTrigger 属性用于指定目标更新数据源的条件。

如图 14-8 所示。

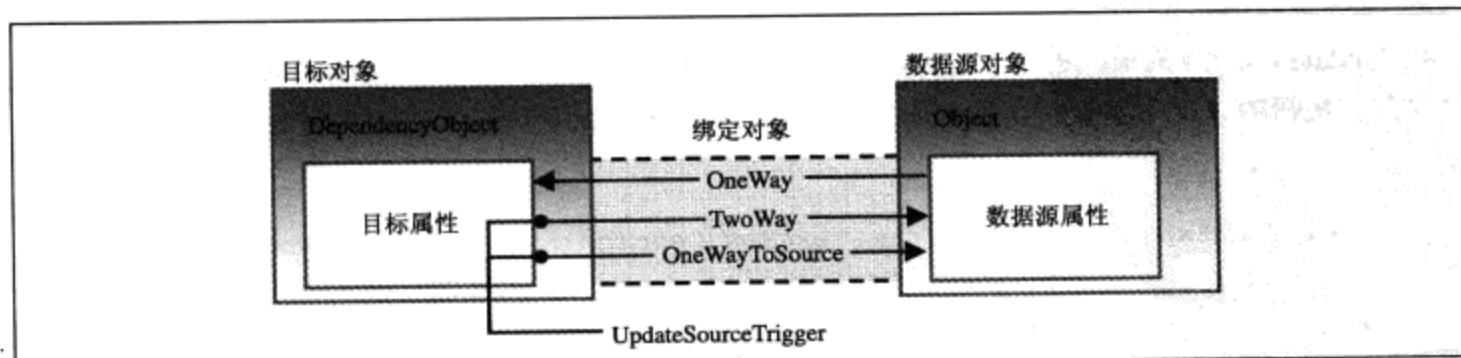


图 14-8 文本框中的文本修改数据源的时间

`UpdateSourceTrigger` 和 `Mode` 一样，也是一个枚举类型，取值如下。

- (1) `LostFocus`: 目标对象失去焦点时更新数据源。
- (2) `PropertyChanged`: 表示目标属性发生变化即更新数据源。
- (3) `Explicit`: 只有显式通知才会更新数据源。
- (4) `Default`: `Binding` 的 `UpdateSourceTrigger` 属性和目标属性一致，大部分依赖属性行为为 `PropertyChanged`。而 `Text` 属性为了避免效率太低，其行为为 `LostFocus`。

举例说明，为了方便观察文本框中的文本何时修改数据源，添加另一个 `TextBlock` 并且 `Text` 属性绑定资源 `mumu` 的 `Name` 属性。然后原来输入姓名的 `TextBox` 绑定语句的 `UpdateSourceTrigger` 分别设置为 `LostFocus`、`PropertyChanged`、`Explicit` 和 `Default`，程序运行的结果如图 14-9 所示。

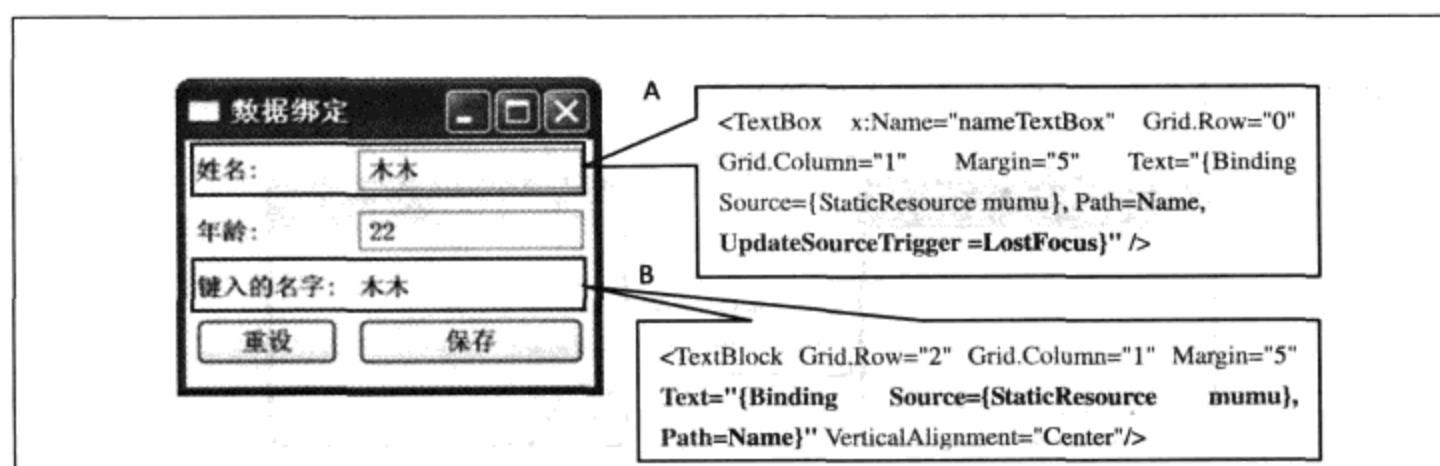


图 14-9 程序运行结果

- (1) `UpdateSourceTrigger` 设置为 `LostFocus`: 文本框 A 在修改时文本块 B 不变，只有当焦点离开文本框 A 时才会随之更新。
- (2) `UpdateSourceTrigger` 设置为 `PropertyChanged`: 文本框 A 在修改时文本块 B 会随之变化。
- (3) `UpdateSourceTrigger` 设置为 `Default`: 效果同 `LostFocus`，这也证明了 `Text` 的默认 `UpdateSourceTrigger` 行为为 `LostFocus`。
- (4) `UpdateSourceTrigger` 设置为 `Explicit`: 文本框 A 在修改和失去焦点时文本块 B 都不变。只有在“保存”按钮的单击事件处理函数中添加如代码 14-17 所示的代码，单击该按钮时文本块 B 才会变化。

```
BindingExpression be = nameTextBox.GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();
```

代码 14-17 文本块 B 发生改变

14.2.4 绑定数据源的 4 种方式

绑定数据源有 4 种方式，即 Source、ElementName、RelativeSource 和 DataContext。

1. Source

前例中的数据源均通过 Source 属性指定，该属性适用数据源为普通的.NET 对象的情况。

2. Element

Element 多用于元素之间的绑定，代码 14-18 绑定文本框的 Text 属性和滑块值（详见 mumu_elementtoelementbinding 工程）。

```
<StackPanel Margin="5">
    <Slider Name="sliderFontSize" Margin="3" Minimum="1" Maximum="40"
Value="10" TickFrequency="1" IsSnapToTickEnabled="True" TickPlacement="TopLeft">
    </Slider>
    <TextBlock Margin="10" Name="lblSampleText"
        FontSize="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}"
        Text="木木">
    </TextBlock>
</StackPanel>
```

代码 14-18 绑定文本框的 Text 属性和滑块的值

拖动滑块条可以控制字号，如图 14-10 所示。

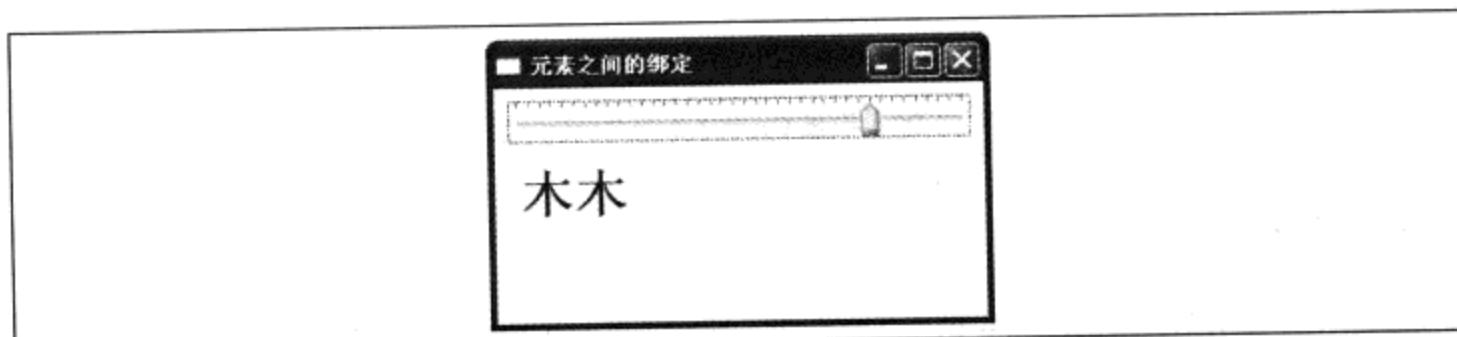


图 14-10 控制字号

3. RelativeSource

如果指定数据源时需要设置相对位置，则使用 RelativeSource 属性，代码 14-19 绑定一个文本框的 Text 属性和窗口的标题。

```
<TextBlock Text="{Binding Path=Title, RelativeSource= {RelativeSource
FindAncestor, AncestorType={x:Type Window}}}">
</TextBlock>
```

代码 14-19 绑定一个文本框的 Text 属性和窗口的标题

其中 FindAncestor 表示将从目标对象沿着元素树向上查找，AncestorType 表示需要查找的对象类型。即将从目标对象为起点，沿着元素树往上查找，直至找到第 1 个类型为 Window 的对象作为数据源。

除了 FindAncestor 选项，还有如下 3 个选项。

- (1) Self：数据源即为目标对象，用来绑定目标对象的 A 属性和 B 属性。
- (2) PreviousData：多用在列表中，表示前一个列表项。
- (3) TemplatedParent：表示拥有该模板的父类。

4. DataContext

在某些情况下会出现多个元素绑定同一个对象，如前例中两个 TextBox 均绑定 Person 对象。在每个绑定表达式中均指定了数据源（如代码 14-9）。

如果设置 Grid 的 DataContext 属性为 "{StaticResource mumu}"，则无需为每个 TextBox 指定 Source 值，如代码 14-20 所示。

```
<Grid DataContext="{StaticResource mumu}">
    .....
    <TextBox x:Name="nameTextBox" Grid.Row="0" Grid.Column="1" Margin="5"
Text="{Binding Path=Name, UpdateSourceTrigger = Explicit}" />
    <TextBox Name="ageTextBox" Grid.Row="1" Grid.Column="1" Margin="5" ....>
        <TextBox.Text>
            <Binding Path="Age" UpdateSourceTrigger="PropertyChanged">
                .....
            </Binding>
        </TextBox.Text>
    </TextBox>
    .....
</Grid>
```

代码 14-20 设置 Grid 的 DataContext 属性

14.2.5 值转换

有时希望保存在数据源中的数据和在用户界面中表现的数据有所不同，如在数据源中数值 1 表示男，数值 2 表示女，在用户界面中表现的是男和女。再如数据源中保存的时间可能是以毫秒为单位的整型数值，但是表现在用户界面中可能是“2009-10-28 16:50:56”等多种形式。前面黄蓉的例子中的 AgeToForegroundConverter 就是一个自定义的值转换类，如图 14-11 所示。

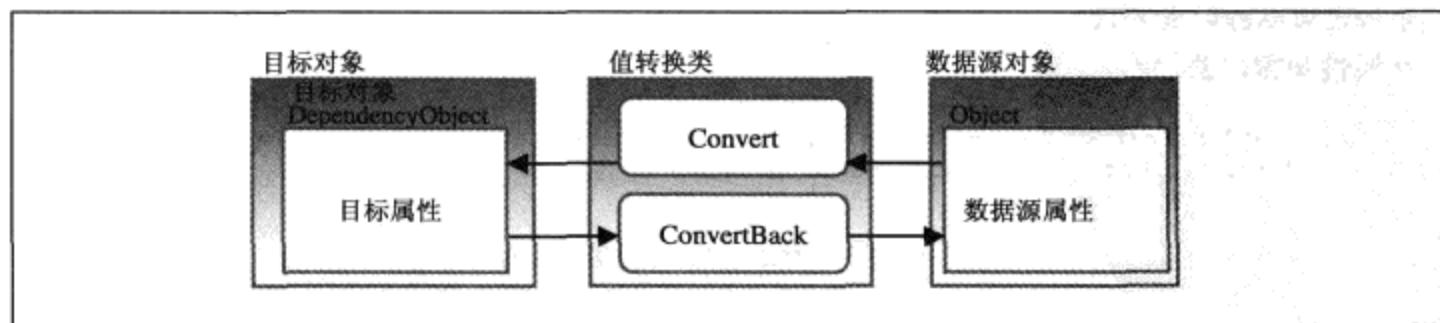


图 14-11 自定义的值转换类

值转换类实现 IValueConverter 接口，该接口需要实现 Convert 和 ConverBack 函数。数据源属性经过

Convert 函数转换为目标对象需要显示的目标属性，而逆过程由 ConvertBack 函数完成。

在前面的例子中 AgeToForegroundConverter 将不同年龄段转换为不同的前景色显示，如代码 14-21 所示。

```
public class AgeToForegroundConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        if (targetType != typeof(Brush)) { return null; }
        int age = int.Parse(value.ToString());
        return (age < 18 ? Brushes.Red : Brushes.Black);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

代码 14-21 将不同年龄段转换为不同的前景色显示

将转换类设置给目标属性最简单的方法是将该类声明为一个资源，然后将该资源赋值给 Binding 的 Converter 属性，如代码 14-22 所示。

```
<Window x:Class="mumu_rongerwithdatabinding.MainWindow"
.....
    xmlns:local ="clr-namespace:mumu_rongerwithdatabinding"
    Title="数据绑定" Height="155"
    Width="200" >
<Window.Resources>
    .....
    <local:AgeToForegroundConverter x:Key="ageConverter" />
</Window.Resources>
    .....
    <TextBox Name="ageTextBox" Grid.Row="1" Grid.Column="1" Margin="5"
Foreground="(Binding Path=Age, Mode=OneWay, Converter={StaticResource ageConverter})" ....>
    </TextBox>
```

代码 14-22 将转换类设置给目标属性

14.2.6 数据验证

除了值验证以外，数据绑定模型也提供了一种数据验证机制，以确保用户的输入符合预期的结果。如果不符，用户界面会以一种反馈方式提示错误的原因。

1. 自定义数据验证类

Binding 的 ValidationRules 属性是数据验证规则的集合，前例中的另外一个类 NumberRangeRule 是一个自定义的数据验证规则类，它派生自一个抽象类 ValidationRule，其中必须实现 Validate 函数。该函数用于实现自定义验证逻辑，返回一个 ValidationResult 对象，其声明如下：

```
public ValidationResult(bool isValid, Object errorContent);
```

第1个参数是一个 bool 值变量，表示数据验证是否合法。如果合法，则为 true，否则为 false；第2个参数表示数据验证不合法的相关信息。NumberRangeRule 的实现代码如代码 14-23 所示。

```
public class NumberRangeRule : ValidationRule
{
    int _min;
    public int Min
    {
        get { return _min; }
        set { _min = value; }
    }

    int _max;
    public int Max
    {
        get { return _max; }
        set { _max = value; }
    }

    public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
    {
        int number;
        if (!int.TryParse((string)value, out number))
        {
            return new ValidationResult(false, "非数值性字符");
        }

        if (number < _min || number > _max)
        {
            string s = string.Format("超出正常人的年龄范围 ({0}-{1})", _min, _max);
            return new ValidationResult(false, s);
        }

        return ValidationResult.ValidResult;
    }
}
```

代码 14-23 NumberRangeRule 的实现代码

2. 不合法信息的提示反馈

当信息验证不合法时用户需要相应提示反馈，如前例中通过 ToolTip 来提示不合法的原因，并且文本框的边框变为红色。

(1) ToolTip 的提示反馈

代码 14-24 绑定 ToolTip 和出错信息，其中使用了 RelativeSource 的方法绑定，Self 表示绑定的数据源是 TextBox 自身。Validation.Errors 是一个附加属性，它是 ValidationError 类型的集合。当数据验证不合法时，系统会自动创建一个 ValidationError 并添加到该集合中。由于每次验证时系统会自动清空 Errors 集合，因此 ToolTip 绑定的都是集合中的第 1 个 ValidationError 对象。ErrorContent 是 ValidationError 的属性，该属性值来自于 Validate 函数中返回的 ValidationResult 的第 2 个参数值。

```

<TextBox Name="ageTextBox" Grid.Row="1" Grid.Column="1" Margin="5"
Foreground="{Binding Path=Age, Mode=OneWay, Converter={StaticResource
ageConverter}}" ToolTip="{Binding RelativeSource={RelativeSource Self},
Path=(Validation.Errors)[0].ErrorContent}" >
.....

```

代码 14-24 ToolTip 和出错信息绑定

(2) ErrorTemplate

如果验证不合法，文本框的边框以红色显示，这是文本框的默认行为。如果希望以另外一种可视化方式来提示反馈，则需要通过如下两个步骤。

(1) 创建一个控件模板，其中`<AdornedElementPlaceholder>`标签表示需要装饰的控件。这里指`TextBox`控件，如代码 14-25 所示。

```

<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>

```

代码 14-25 创建一个控件模板

(2) 将控件模板赋值给`Validation.ErrorTemplate`附加属性，如代码 14-26 所示。

```

<TextBox Name="ageTextBox" Grid.Row="1" Grid.Column="1" Margin="5"
Foreground="{Binding Path=Age, Mode=OneWay, Converter={StaticResource
ageConverter}}" ToolTip="{Binding RelativeSource={RelativeSource Self},
Path=(Validation.Errors)[0].ErrorContent}"
Validation.ErrorTemplate="{StaticResource validationTemplate}">
.....

```

代码 14-26 将控件模板赋值给`Validation.ErrorTemplate`属性

如图 14-12 所示，在“年龄”文本框中输入不合法的字符。文本框边框颜色不变，而在文本框之前显示一个红色惊叹号。

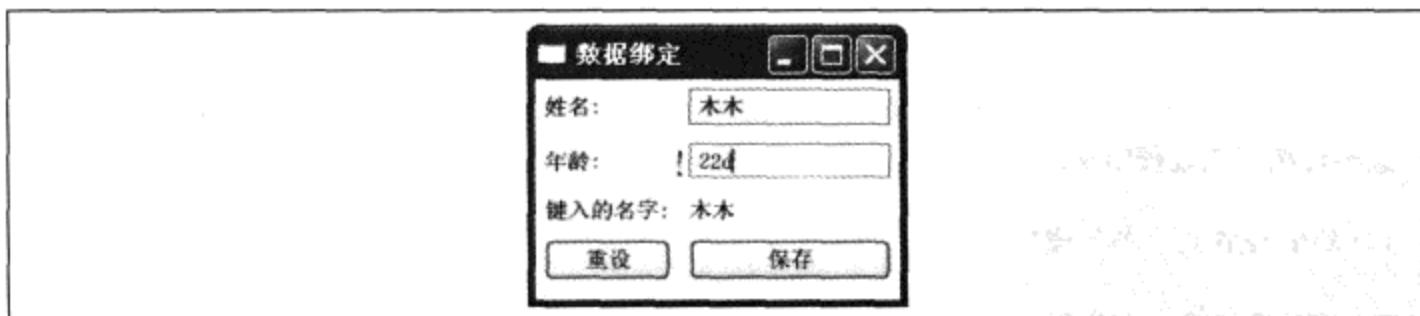


图 14-12 为不合法信息的提示反馈

3. 数据验证过程

数据验证和值转换不同，它只发生在从目标到数据源转换的过程中，即数据验证只能用在模式为`TwoWay` 或者`OneWayToSource` 的两种绑定。`ValidationRule` 的`ValidationStep` 属性用来标识

ValidationRule 的 Validate 函数的调用顺序，它有 4 种不同的枚举值，即 RawProposedValue、ConvertedProposedValue、UpdatedValue 和 CommittedValue。

数据验证过程如图 14-13 所示。

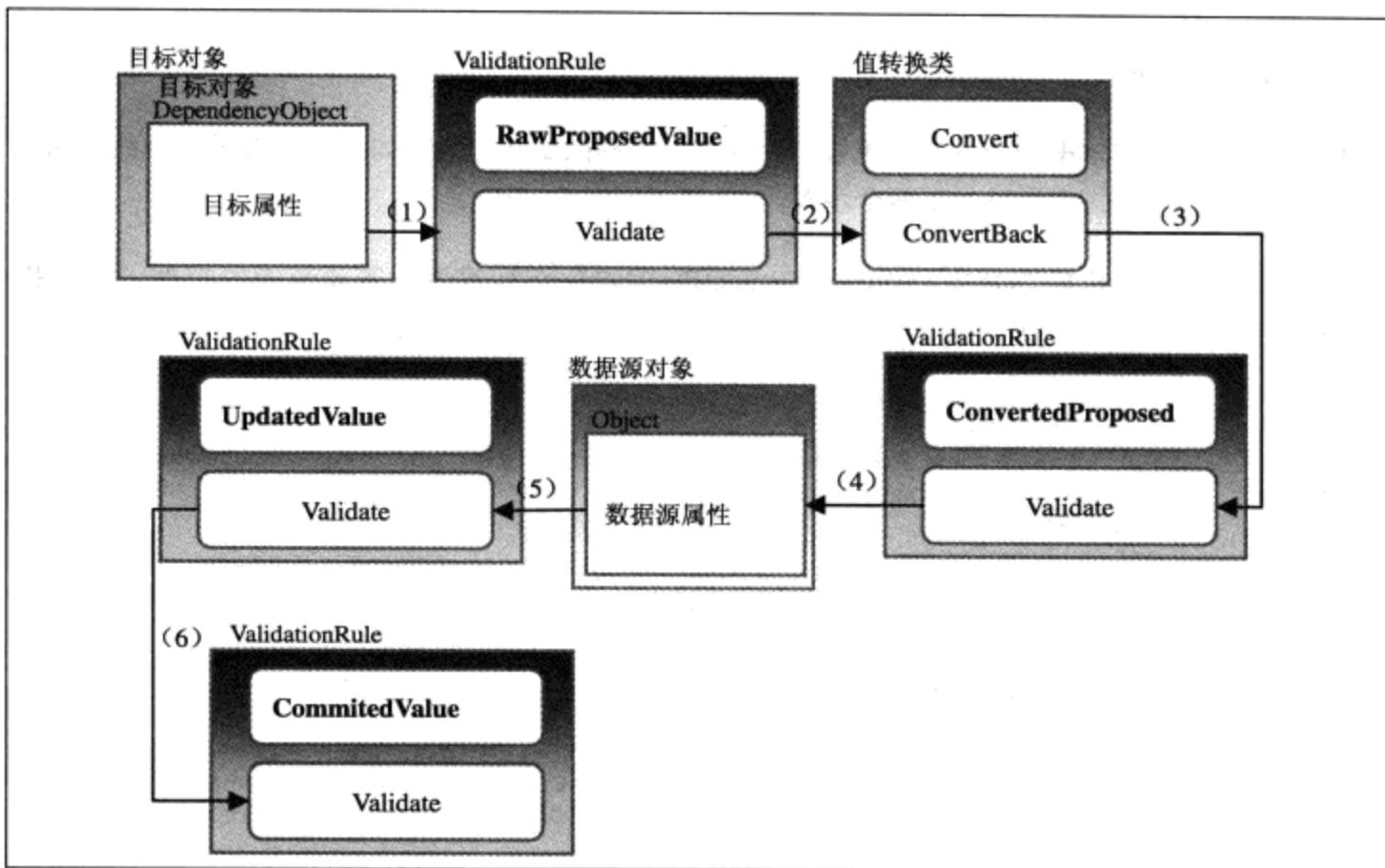


图 14-13 数据验证过程

- (1) 如果有自定义的 ValidationRule，WPF 会首先调用值为 `RawProposedValue`（`ValidationStep`）的 `Validate` 函数。如果当验证不合法，该过程结束；否则继续。
- (2) 如果有值转换类，WPF 会调用值转换类的 `ConvertBack` 函数。如果转换不成功，该过程结束；否则继续。
- (3) WPF 会继续检查值为 `ConvertedProposed` 的自定义 ValidationRule，调用其 `Validate` 函数。如果验证不合法，该过程结束；否则继续。
- (4) WPF 设置数据源的属性值。
- (5) WPF 继续检查值为 `UpdatedValue` 的自定义 ValidationRule，调用其 `Validate` 函数。如果验证不合法，该过程结束；否则继续。
- (6) WPF 检查值为 `CommittedValue` 的自定义 ValidationRule，调用其 `Validate` 函数，整个验证过程结束。

木木再重新看黄蓉的例子，感觉已经如履平地了。但是转念一想，又有些犯愁，说到：“蓉儿，人

员管理系统肯定不只一条记录。有没有绑定多条记录的方法啊？”黄蓉说到：“当然有，这是数据绑定里面一个高级的主题——与数据集合绑定。”黄蓉正要说下去。木木说到：“蓉儿，你已经给我讲了很多了，我自己来研究研究吧。”黄蓉微微一笑：“好的，木木，那别睡太晚啊。”

14.3 高级主题——与数据集合绑定

与数据集合绑定的模型如图 4-14 所示。

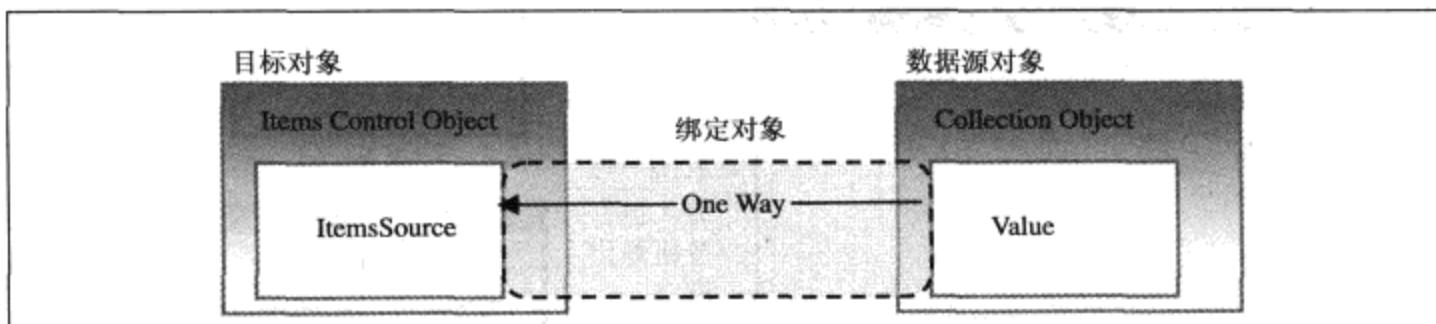


图 14-14 与数据集合绑定的模型

其中目标对象要求是派生自 ItemsControl 的控件，如 ListBox，TreeView 或者 ListView 等。绑定的目标属性是 ItemsSource，它默认支持 OneWay 绑定模式；数据源对象则是一个集合。

14.3.1 实现一个数据源集合

与数据集合绑定同样会出现与单个对象绑定的问题，即目标对象无法知道数据源对象何时改变。但是集合和单个目标对象不同，发生改变有如下两种情况。

- (1) 其中的一个对象的属性变化，该集合中的类型都要实现 INotifyPropertyChanged 接口。
- (2) 集合插入或者删除一个对象，需要该集合实现 INotifyCollectionChanged 接口。

好在 WPF 提供了一个集合类 ObservableCollection(T) 实现 INotifyCollectionChanged 接口，因此可以在前面的例子中重新构建一个 person 的集合类 People，它继承自 ObservableCollection(Person)，如代码 14-27 所示。

```
public class People : ObservableCollection<Person>
{
    public People()
        : base()
    {
        Add(new Person("木木", 22));
        Add(new Person("黄蓉", 20));
        Add(new Person("黄药师", 40));
        Add(new Person("士兵甲", 26));
        Add(new Person("士兵乙", 32));
        Add(new Person("士兵丙", 28));
    }
}
```

代码 14-27 重新构建一个 person 的集合类 People

14.3.2 绑定目标和集合

重新规划程序，主要由两个窗口组成，一个是前面例子中的窗口（AddPerson），用于添加人员；一个则是浏览所有人员的窗口（MainWindow），该窗口可以浏览分类、排序和过滤人员信息，如图 14-15 所示。

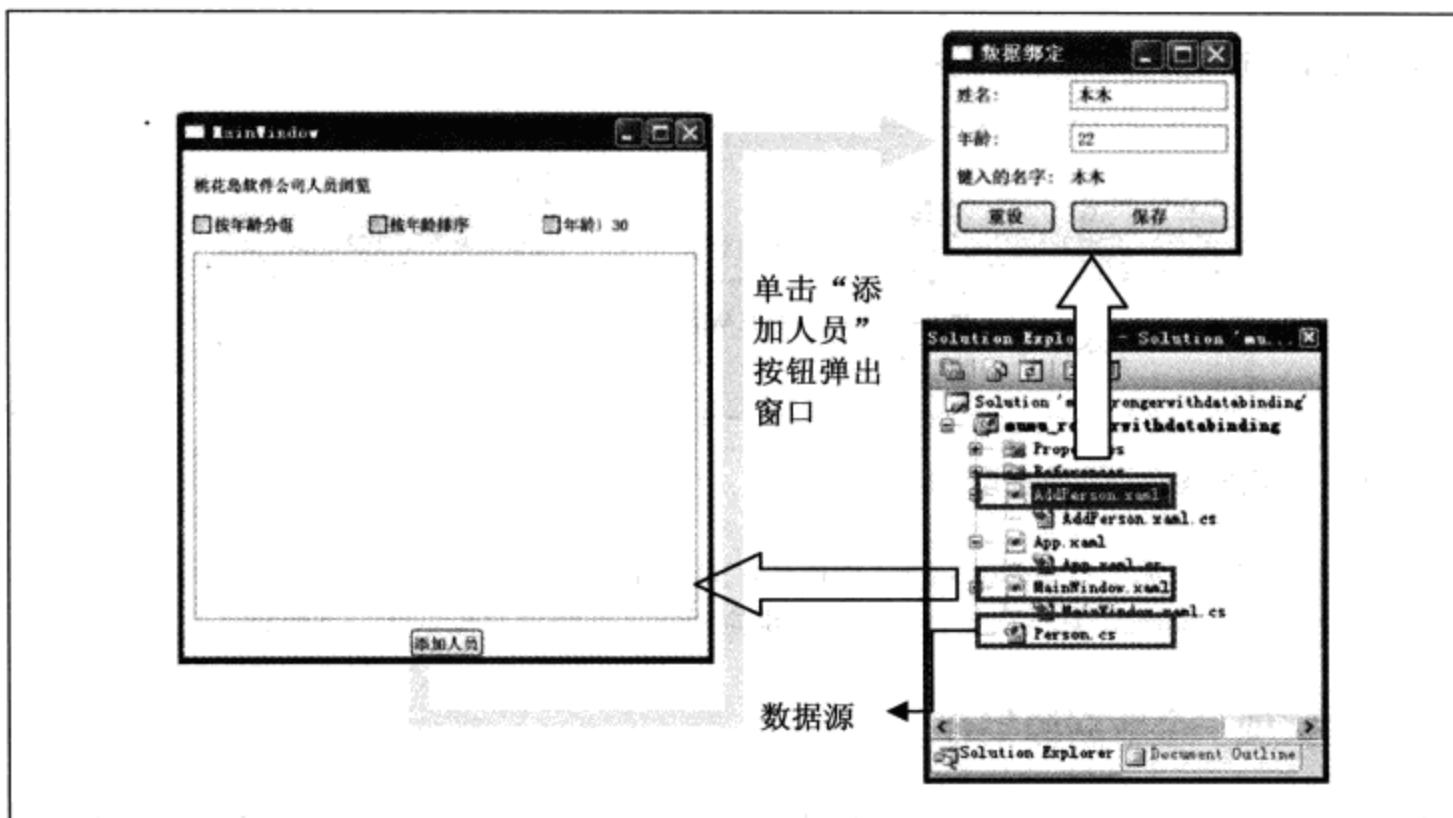


图 14-15 新规划

将数据源作为窗口的一个资源，随后将 ListBox 的 ItemSource 与之绑定，如代码 14-28 所示。

```
....  
<Window.Resources>  
    <local:People x:Key="personsource"/>  
</Window.Resources>  
  
....  
    <ListBox Grid.Row="2" Grid.ColumnSpan="3" Margin="8" ItemsSource="{Binding  
Source={StaticResource personsource}}}></ListBox>
```

代码 14-28 将 ListBox 的 ItemSource 与之绑定

运行结果是 ListBox 中显示 Person 的类型，如图 14-16 所示。

木木想了想，又重载了 Person 的 ToString 函数，如代码 14-29 所示。

```
public override string ToString()  
{  
    return name.ToString();  
}
```

代码 14-29 重载 Person 的 ToString 函数

运行程序，结果如图 14-17 所示。



图 14-16 运行结果

图 14-17 运行结果

虽然如期显示了重载的 `ToString` 返回的值，但是木木仍然不满意，这毕竟是一种蹩脚的方法，而且非常的有限和不灵活。WPF 的解决方案是数据模板。

14.3.3 数据模板

数据模板和控件模板类似，只不过用来定义数据的可视化外观。为上例 `ListBox` 的 `ItemTemplate` 定制数据模板，该数据模板是一个 `Grid` 面板。其中放置了 4 个 `TextBlock`，以及两个绑定 `Person` 的 `Name` 和 `Age` 属性，如代码 14-30 所示。

```

<ListBox.ItemTemplate>
    <DataTemplate>
        <Border BorderThickness="1" BorderBrush="Gray"
            Padding="7" Name="border" Margin="3" MinWidth="400">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition/>
                    <RowDefinition/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="40" />
                    <ColumnDefinition/>
                </Grid.ColumnDefinitions>
                <TextBlock Grid.Row="0" Grid.Column="0" Text="姓名:"/>
                <TextBlock Grid.Row="1" Grid.Column="0" Text="年龄:"/>
                <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding
                    Path=Name}" />
                <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding
                    Path=Age}" />
            </Grid>
        </Border>
    </DataTemplate>
</ListBox.ItemTemplate>

```

代码 14-30 为 `ListBox` 的 `ItemTemplate` 定制的数据模板

运行程序，浏览所有人员的窗口如图 14-18 所示。



图 14-18 浏览所有人员的窗口

数据模板也可以作为资源复用，修改代码如代码 14-31 所示。

```
<Window.Resources>
    <local:People x:Key="personsource"/>
    <DataTemplate x:Key="persondatatemplate">
        <Border BorderThickness="1" BorderBrush="Gray"
            Padding="7" Name="border" Margin="3" MinWidth="400">
            <Grid>
                .....
            </Grid>
        </Border>
    </DataTemplate>
</Window.Resources>
.....
<ListBox Grid.Row="2" Grid.ColumnSpan="3" Margin="8" ItemsSource="{Binding
Source={StaticResource personsource}}" ItemTemplate="{StaticResource
persondatatemplate}">
</ListBox>
```

代码 14-31 数据模板作为资源

此外可以利用数据模板的 `DataType` 属性，如代码 14-32 所示。

```
<DataTemplate DataType="{x:Type local:Person}">
    .....
</DataTemplate>
.....
<ListBox Grid.Row="2" Grid.ColumnSpan="3" Margin="8" ItemsSource="{Binding
Source={StaticResource personsource}}">
</ListBox>
```

代码 14-32 利用数据模板的 `DataType` 属性

这里数据模板没有添加 `x:Key` 关键字，而且 `ListBox` 也未指定数据模板。这是因为 `DataType` 属性类似样式的 `TargetType` 属性，可以自动应用到所有类型为 `Person` 的对象。

14.3.4 集合视图

对于一个人员管理系统，过滤（filter）、排序（sort）和分组（group）都是一些必要的功能。WPF 中借助集合视图来完成。

集合视图从实现的角度来说是实现 ICollectionView 接口的类，它可以在不改变数据集的情况下过滤、分组和排序数据。数据集和数据视图是一对多的关系，并且当目标直接绑定数据源时，WPF 也会为数据源创建一个默认的数据视图。实际上绑定的还是数据视图，如图 14-19 所示。

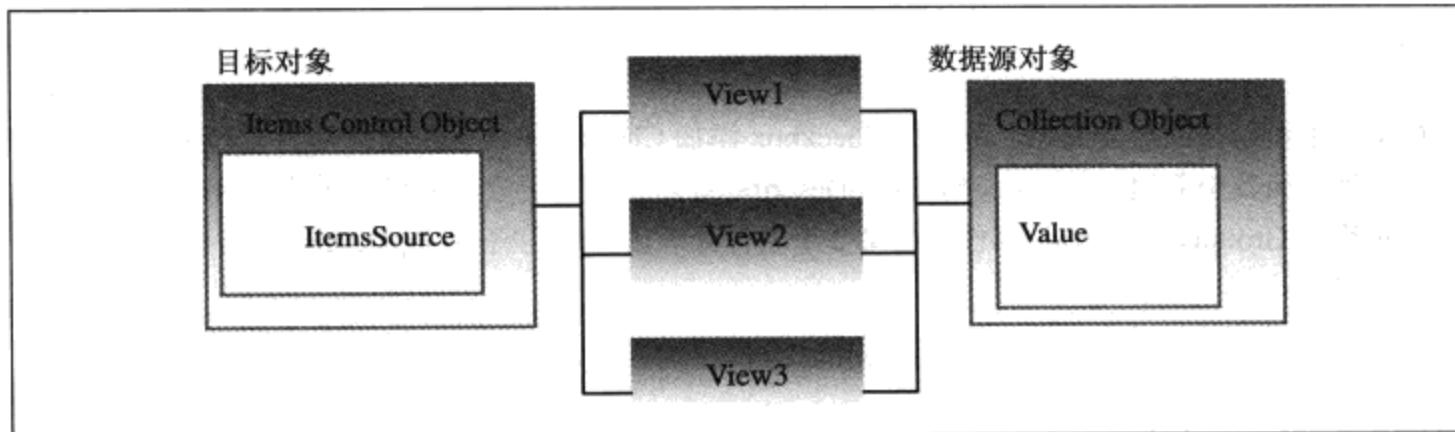


图 14-19 数据视图

1. 创建一个集合视图

创建一个集合视图非常简单，因为在 XAML 中已经提供了一个 CollectionViewSource 类，在窗口资源中声明，如代码 14-33 所示。

```
....  
<Window.Resources>  
....  
    <CollectionViewSource  
        Source="{Binding Source={StaticResource personsource}}"  
        x:Key="listingDataView" />  
</Window.Resources>
```

代码 14-33 在窗口资源中声明

Source 属性绑定了数据源，然后 ListBox 的 ItemsSource 属性绑定该视图，如代码 14-34 所示。

```
<ListBox x:Name="listbox" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"  
ItemsSource="{Binding Source={StaticResource listingDataView}}">  
</ListBox>
```

代码 14-34 ListBox 的 ItemsSource 属性绑定该视图

在 code 文件中添加一个类型为 CollectionViewSource 成员变量 listing DataView，在构造函数中从资源中获得集合视图对象，如代码 14-35 所示。

```
CollectionViewSource listingDataView;  
public MainWindow()  
{  
    InitializeComponent();
```

```
        listingDataView  
(CollectionViewSource)(this.Resources["listingDataView"]);  
    }  
}
```

代码 14-35 获得集合视图对象

获取集合视图对象后即可分组、排序和过滤数据。

2. 分组、排序和过滤数据

本节说明如何分组、排序和过滤数据。

(1) 分组

人员浏览中可以按照年龄分组，为分组 Checkbox 添加 Checked 和 Unchecked 事件处理函数。在第 1 个事件处理函数中新建一个 PropertyGroupDescription 对象，指定分组依据的属性，然后添加到集合视图对象的 GroupDescriptions 属性；在第 2 个事件处理函数中则将 GroupDescriptions 属性清空，如代码 14-37 所示。

```
private void Grouping_Checked(object sender, RoutedEventArgs e)  
{  
    PropertyGroupDescription groupDescription = new  
PropertyGroupDescription();  
    groupDescription.PropertyName = "Age";  
    listingDataView.GroupDescriptions.Add(groupDescription);  
}  
  
private void Grouping_Unchecked(object sender, RoutedEventArgs e)  
{  
    listingDataView.GroupDescriptions.Clear();  
}
```

代码 14-36 按照年龄分组

分组结果如图 14-20 所示。



图 14-20 分组结果

为 GroupStyle 的 HeaderTemplate 添加一个数据模板，这样做的目的是使分组效果更明显一些。如代

码 14-37 所示。

```
<Window.Resources>
    <DataTemplate x:Key="groupingHeaderTemplate">
        <TextBlock Text="{Binding Path=Name}"
            Foreground="Navy" FontWeight="Bold" FontSize="12"/>
    </DataTemplate>

</Window.Resources>

<ListBox x:Name="listbox" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
ItemsSource="{Binding Source={StaticResource listingDataView}}" >
    <ListBox.GroupStyle>
        <GroupStyle
            HeaderTemplate="{StaticResource groupingHeaderTemplate}"/>
    </ListBox.GroupStyle>
</ListBox>
```

代码 14-37 为 GroupStyle 的 HeaderTemplate 添加一个数据模板

分组结果如图 14-21 所示。



图 14-21 分组结果

(2) 排序

为排序 CheckBox 添加 Checked 和 Unchecked 事件处理函数，如代码 14-38 所示。

```
private void Sorting_Checked(object sender, RoutedEventArgs e)
{
    listingDataView.SortDescriptions.Add(
        new SortDescription("Age", ListSortDirection.Ascending));
}
private void Sorting_Unchecked(object sender, RoutedEventArgs e)
{
    listingDataView.SortDescriptions.Clear();
}
```

代码 14-38 按年龄排序

升序排序结果如图 14-22 所示。



图 14-22 升序排序结果

(3) 过滤

添加一个事件处理函数实现过滤条件，以过滤年龄大于 30 岁的人员为例，如代码 14-39 所示。

```
private void ShowOnlyGreater30Filter(object sender, FilterEventArgs e)
{
    Person person = e.Item as Person;
    if (person != null)
    {
        if (person.Age > 30)
        {
            e.Accepted = true;
        }
        else
        {
            e.Accepted = false;
        }
    }
}
private void Filtering_Checked(object sender, RoutedEventArgs e)
{
    listingDataView.Filter += new FilterEventHandler
(ShowOnlyGreater30Filter);
}

private void Filtering_Unchecked(object sender, RoutedEventArgs e)
{
    listingDataView.Filter -= new FilterEventHandler
(ShowOnlyGreater30Filter);
}
```

代码 14-39 过滤年龄大于 30 岁的人员

过滤结果如图 14-23 所示。

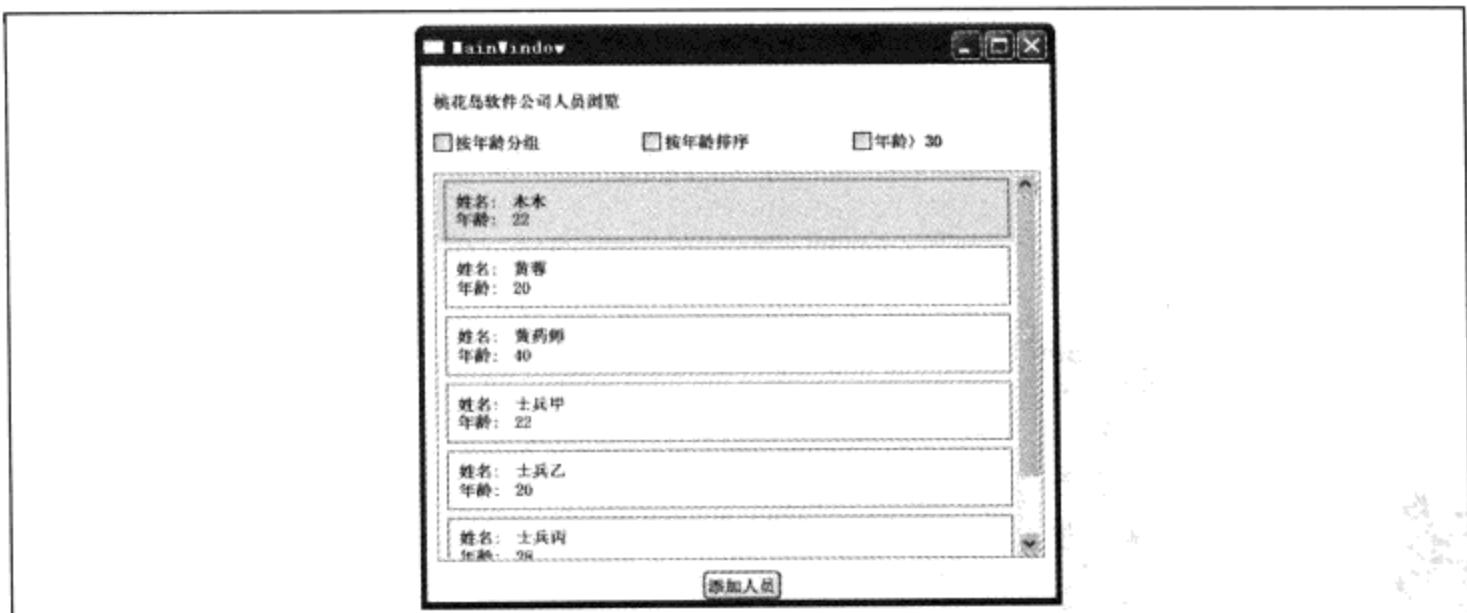


图 14-23 过滤结果

14.4 后记

第二天，木木把这个实验成果拿给了黄药师看。黄药师看后，心中不免暗喜，想到这小子居然能够一晚上达到这种程度，实在是很不简单。不过仍然板着脸说到：“木木，你这个离实际的系统还差太远了。数据项太简单；为什么不记出生年月，年龄是可以根据出生年月计算出来的；用户交互太粗糙……”

后来，后来……木木就真的做了一个桃花岛软件公司人员管理系统，据说用得还不错，也得到了岳父大人的表扬。

14.5 接下来做什么

关于数据绑定还有和多种数据源的绑定，比如 XML 文件，ADO 对象等等。当然也无法避开另外一个话题——LINQ。这些就靠木木自学了。第四卷自此就算全部结束了。第四卷其实相当于心法 2.0 版本，是在实践之后，又一次的理论学习。下面第五卷将是绚丽多姿的一卷。

参考文献

- [1] MSDN Library for Visual Studio 2008 SP1 Data Binding Overview.
- [2] MSDN Library for Visual Studio 2008 SP1 Data Templating Overview.



紫杉红烛



奇妙的二维图形世界——面壁

华山本来草木清华，景色极幽。这危崖却是例外，自古相传是玉女发钗上的一颗珍珠。当年华山派的祖师以此危崖为惩罚弟子之所，主要便因此处无草无木，无虫无鸟。受罚的弟子在面壁思过之时，不致为外物所扰，心有旁骛。

——《笑傲江湖》，“第八章 面壁”^[1]

这一部分讲的是令狐冲犯了华山戒条被岳不群罚在玉女峰面壁。这个期间是令狐冲习武生涯当中最为重要的一个阶段。任何一个图形框架，图形毋庸置疑都是极其重要的一部分，也是非常庞大繁杂的一部分。本章同样以木木受罚面壁开始，系统地学习 WPF 的二维图形，相信对木木来说将是 WPF 学习进阶最为重要的一个阶段。

本章内容如下。

- (1) 面壁。
- (2) 第一块石壁——二维图形的数学基础。
- (3) 第二块石壁——WPF 的二维图形架构。
- (4) 第一本书——颜色和画刷。
- (5) 第二本书——Shape。
- (6) 第三本书——Geometry。
- (7) 第四本书——Drawing 和 Visual。
- (8) 面壁之后——接下来做什么。

15.1 面壁

近段时间，木木总感觉学习 WPF 提高不快，而且时间长了也失去了原先的学习热情。于是每天闲逛无所事事。时间一晃过了半年之久。终于事情还是找上了门，一天药师把木木叫到了办公室。药师道：“木木我们需要做一个图形编辑器，能够现实点，线和面图元的编辑。”说着药师顺手打开了 Microsoft Office Visio 2007，又道：“和它的功能差不多也就行了。”

木木见如此复杂，自然不敢答应。

药师眉头一皱，说到：“就实现简单的图形移动和缩放吧。”

木木感到难度已经降低了不少，但是仍旧毫无头绪。虽然自己学习 WPF 的时间不短，但是和真正的项目实践还差得不少。于是木木只得继续沉默。

药师见他始终不答，早已没有了耐心，说到：“你哪像我药师的女婿，罚你在公司玉女峰悬崖面壁，好好想想为什么你学了这么久啥都做不出来？什么时候想好了再下山吧。”

于是木木开始了惨淡的面壁生涯……

木木算是一个宅男，每天依旧学习学习 WPF，累的时候听听歌，看看电影，加上蓉儿每天给他送饭，也会陪他说说话，在山上走走，日子也算过得逍遥。但是时间久了，木木学习仍不得其法，图形编辑器依旧没有任何思路，不由烦闷，于是对着石壁胡乱猛捶一番，忽然在一处听得空空的回声，似乎石壁后面有很大的空旷之处。于是木木好奇心起，到石洞外拾起一块斗大石头，运力向石壁上砸去，石头相击，石壁后隐隐有回声传来。他运力再砸，突然间砰的一声响，石头穿过石壁，落在彼端地下。木木发现石壁后别有洞天，霎时间便将什么依赖属性、模板、样式统统抛在九霄云外，也没有了烦恼，又去拾了石头再砸，砸不到几下，石壁上破了一个洞孔，脑袋已可从洞中伸入。他将石壁上的洞孔再砸得大些，钻将进去，只见里面是一个更大的石洞，壁上刻满了图和字。最为显眼的是左方七个大字“药师的图形世界”。

突然间木木明白了泰山大人的用意，原来药师早就料到木木会发现这块石壁，才罚他在此处面壁。木木细细打量了一下，从“药师的图形世界”开始由左至右，石壁被天然地划分为二块。于是木木从第一块开始看起，只见上面是……

超风为何斗不过全真老道？

超风自幼随我习武，和玄风盗得九阴真经勤加练习，武功大进，但与一流的全真老道差距甚大，为何？

木木实在好奇，继续往下看。

全真教的小道士入门不学招式，只学呼吸、坐下、行路、睡觉的心法。超风不同，修炼九阴真经，仅凭“九阴白骨爪”横行江湖，但武功始终处在二流，不能有所精进。绝顶高手的修炼还是要依照全真教这样的玄门正宗先打基础后学招式，重阳和伯通即为明证。

如图 15-1 所示，超风的学习曲线先快而后慢，而全真教学习曲线先慢而后快。WPF 的学习当效仿全真教的学习曲线。拐点以前，效仿全真教者上升缓慢，同时见效仿梅超风者上升迅速，与之对比更为郁闷和痛苦，需耐得住寂寞，切记！

黄药师的这番话似乎点到了木木心里，仔细想想前段时间的学习也的确是走得全真教的路线，自己尚处在拐点之前，进步缓慢也是自然现象，再加上周围确实有人看了一些速成的教材，转眼间就比自己高出了很多，于是更觉得自己是不是缺少天赋。木木的郁闷已经一扫而光，学习的劲头又足了起来，继续往下看去。

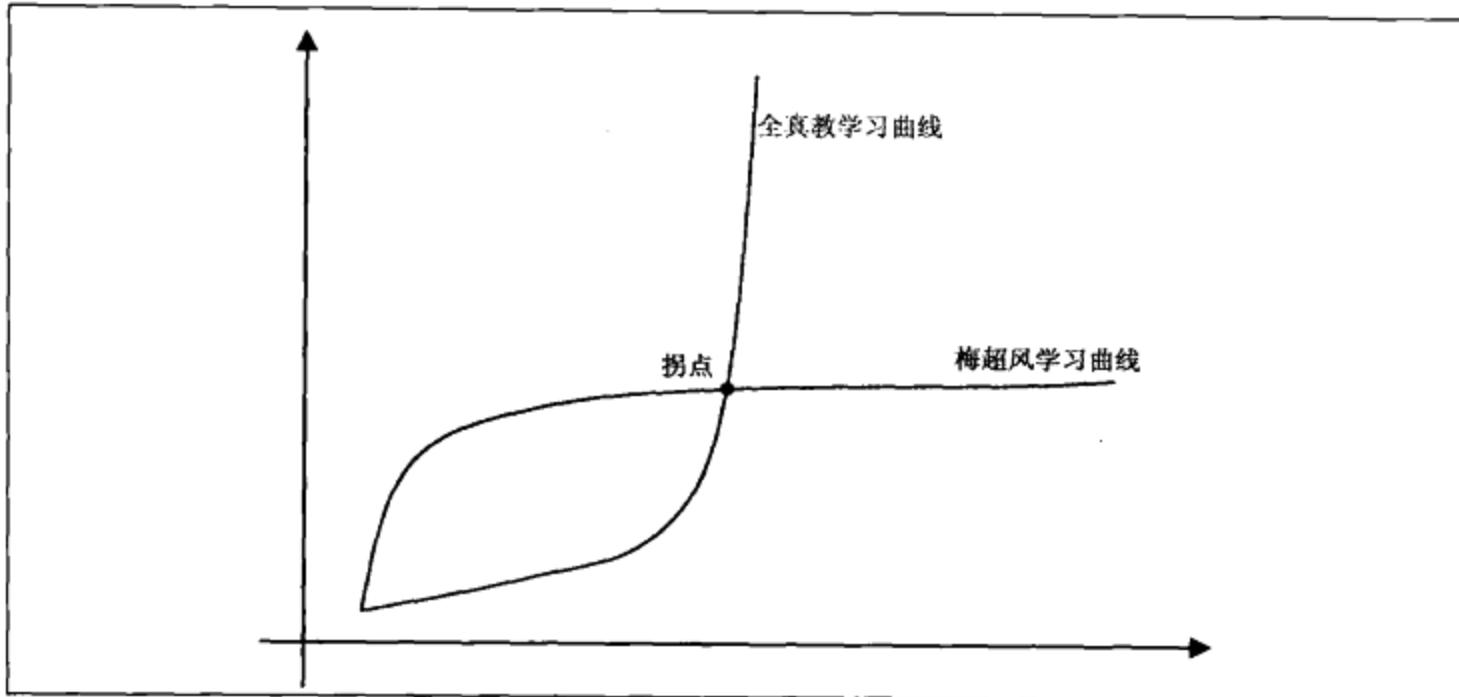


图 15-1 全真教和梅超风学习曲线图

WPF 的图形编程，图形的数学基础既是心法！

木木还想往下看去，后面已是大块的石灰白，显然是有人有意为之，于是有些沮丧，转念一想，石壁上没有，难道我还不能自己学？想到此处，于是返回悬崖，将洞孔用一副大大的老婆（蓉儿）画像遮住，开始潜心学习 WPF 的图形数学基础……

15.2 二维图形的数学基础（第一块石壁）

二维图形的数学基础适用于任何一种图形框架，我们首先讨论 WPF 一直宣称的“分辨率无关”的含义。然后介绍 WPF 中的坐标系，以及几何变换中的面、点、向量和矩阵的相关知识。

15.2.1 分辨率无关

WPF 宣称分辨率无关的主要原因是规定其坐标单位为 1/96 英寸，称为“DIU”¹。

1. 3 种误解

对分辨率无关的 3 种误解如下。

(1) 改变显示器的分辨率设置后，同一个 WPF 的用户界面的图形尺寸不会变化，即无论屏幕分辨率是 1 024 像素×768 像素还是 800 像素×600 像素，WPF 的窗口、按钮和组合框等的尺寸不变。

为验证，新建一个 WPF 应用程序窗口。其中高度为 400 DIU，宽度为 600 DIU。让这个窗口分别在分辨率设置为 1 280 像素×1 024 像素和 800 像素×600 像素的环境下运行，两个窗口的尺寸明显不

¹ DIU：Device independent unit，设备无关单位

同，如图 15-2 所示。



图 15-2 左图为 1280 像素×1024 像素分辨率，右图为 800 像素×600 像素分辨率

(2) 改变显示器的 DPI 设置后同一个 WPF 的用户界面和绘制的图形尺寸不会变化。

在 Windows XP 操作系统下，右击桌面，选择快捷菜单中的“属性”选项，弹出显示器的属性对话框。然后打开“常规”选项卡，可以查看显示器的 DPI 设置，如图 15-3 所示。



图 15-3 显示器的 DPI 设置

仍然以高度为 400 DIU，宽度为 600 DIU 的窗口分别运行在 96 DPI 和 192 DPI 两种显示环境下。从下图可以明显看出窗口的尺寸不同。如果比较精确地测量尺寸，会发现右图窗口尺寸是左图的两倍。

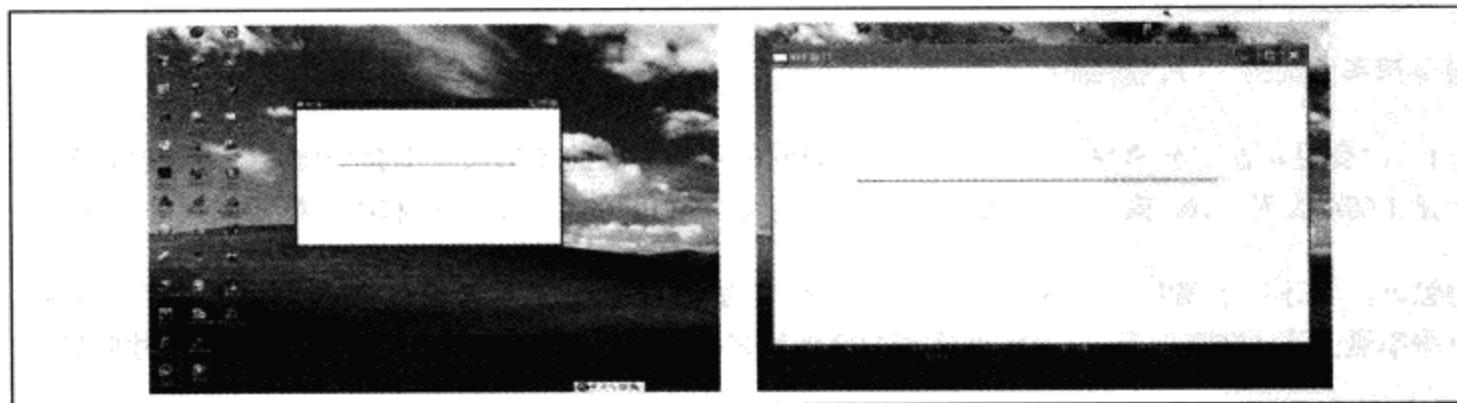


图 15-4 左图为 96 DPI，右图为 192 DPI

(3) 不同屏幕的系统 DPI 设置相同条件下同一个 WPF 的用户界面和绘制的图形尺寸不会变化，做这个实验至少需要两台显示器，实验环境如表 15-1 所示。

表 15-1 两个不同系统的实验环境各元素对照图

实验环境	系统 1	系统 2
显示器类型	桌面 LCD 显示器	笔记本 LCD 显示器
屏幕宽度和高度（像素）	1600×1200	1400×1050
屏幕宽度和高度（英寸）	17.0×12.75	12.0×9.0
实际的物理 DPI	纵向：1600/17.0=94 DPI 横向：1200/12.75=94 DPI	纵向：1400/12=117 DPI 横向：1050/9=117 DPI
操作系统的 DPI 设置	96 DPI	96 DPI

如桌面 LCD 显示器的实际屏幕宽度和高度为 1600 像素×1200 像素，这是显示设备的最大分辨率或物理分辨率。由于两个屏幕物理尺寸不同，所以实际的物理 DPI 可以通过上表中的计算公式得到。

在两个不同的系统中运行同一个 WPF 应用程序，该程序绘制一条长为 384 DIU 的直线，换算后为 $384/96=4$ 英寸，在两个系统中运行结果的实际尺寸如图 15-5 所示。

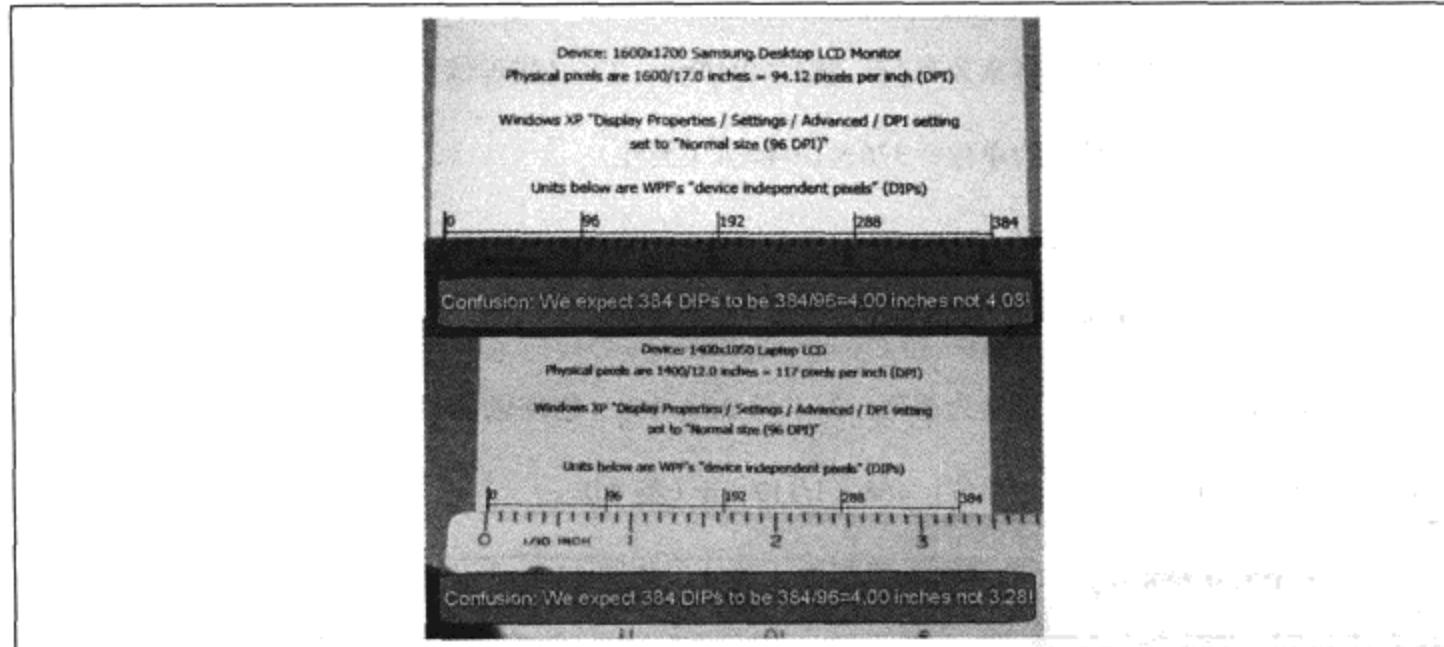


图 15-5 上图实际尺寸为 4.08 英寸，下图实际尺寸为 3.28 英寸^[2]

分别为 4.08 英寸和 3.28 英寸，均与 4 英寸无关。

2. 问题分析

原因在于实际的物理 DPI 和操作系统设置的 DPI 不一致，WPF 并不知道当前所用设备的实际物理 DPI，而只是通过操作系统获得并认为就是实际的物理 DPI 值。如操作系统设置为 96 DPI，那么 WPF 认为坐标单位 1 (DIU)=1 (物理像素)；如果操作系统设置为 120 DPI，那么 WPF 认为 1DIU=120/96

(物理像素) = 1.25 (物理像素)。

由此即可知道为什么测量的值是 4.08 和 3.28 英寸，第 1 个系统环境设置为 96 DPI。

(1) WPF 的 1DIU = 96/96(物理像素) = 1 (物理像素)。

(2) 绘制的线段长度以物理像素为单位：384。

(3) 绘制的线段长度以英寸为单位： $384 \times 1/94 = 4.08$ 。

第 2 个系统环境设置为 96 DPI。

(1) WPF 的 1DIU = 96/96(物理像素) = 1 (物理像素)。

(2) 绘制的线段长度以物理像素为单位：384。

(3) 绘制的线段长度以英寸为单位： $384 \times 1/117 = 3.28$ 。

如果将操作系统的系统 DPI 设置为实际的物理 DPI 值，则可做到真正的“分辨率无关”。如第 1 个系统环境设置为 94 DPI。

(1) WPF 的 1DIU = 94/96 (物理像素) = 0.98 (物理像素)。

(2) 绘制的线段长度以物理像素为单位： $384 \times (94/96) = 376$ (物理像素)。

(3) 绘制的线段长度以英寸为单位： $376 \times 1/94 = 4$ (英寸)。

第 2 个系统环境设置为 117 DPI。

(1) WPF 的 1DIU = 117/96 (物理像素) = 1.22 (物理像素)。

(2) 绘制的线段长度以物理像素为单位： $384 \times 1.22 = 468$ (物理像素)。

(3) 绘制的线段长度以英寸为单位： $468 \times 1/117 = 4$ (英寸)。

这样在两个不同显示器上显示的线段长度都为 4 英寸，如图 15-6 所示。

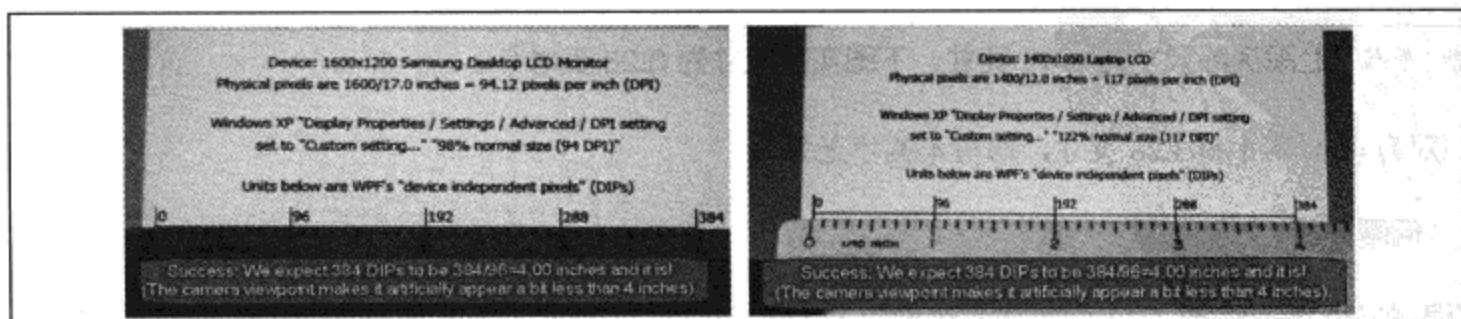


图 15-6 左图为桌面 LCD 显示器，将 DPI 设置为 94DPI；右图为笔记本显示器，将 DPI 设置为 117DPI^[3]

3. 真正含义

“分辨率无关”的真正含义包括如下 3 个方面。

(1) 屏幕显示

在屏幕显示时只有物理 DPI 值等于系统 DPI 值，WPF 才真正做到了“分辨率无关”或者说“设备无关”。但是它的优点并不体现在屏幕显示完全精准。因为很少有人会去找个尺子在屏幕上量一量一条线的长度究竟是多少。它的优点体现在无论是按钮这样的 UI 控件，还是图形，WPF 均使用设备无关的坐标单位。

谈起这个优点，必须与过去的图形系统做比较。比如绘制图形的时候我们往往采用的坐标单位是厘米或者英寸。但是窗口和按钮这样控件的尺寸大小仍然是物理像素。鼠标消息传递过来的坐标单位也是物理像素。在 VC 6.0 的对话框编辑里所使用的坐标单位又不是物理像素的，而是和系统 DPI 设置相关的某种单位。于是程序员需要考虑各种各样的坐标转换。而且一旦调整了系统设置的 DPI 时，会发现应用程序有的控件大小变了，有的没有变，相对的位置也发生了变化，该出来的控件没有出来，该绘制的图形没有出来等等一系列问题。

到 WPF 好了，控件图形包括鼠标消息传递过来的坐标等所有的东西都是采用了设备无关的单位。一切都统一了。（实际上在 WPF 中仍然有极少数使用与设备相关的单位，如 SystemParameters 的 SmallIconWidth 属性以像素为单位。）如果您没有早期的 Windows 用户界面编程的经验，当然很难体会到这样一个优点，但它的的确确是一个优点。

(2) 打印

在打印机上不会出现上述问题，4 英寸的直线打印后仍为 4 英寸。图 15-7 所示为一条 4 英寸的直线，分别在 DPI 设置为 96 DPI 和 120 DPI 下打印。结果尺寸相同，如图 15-7 所示。

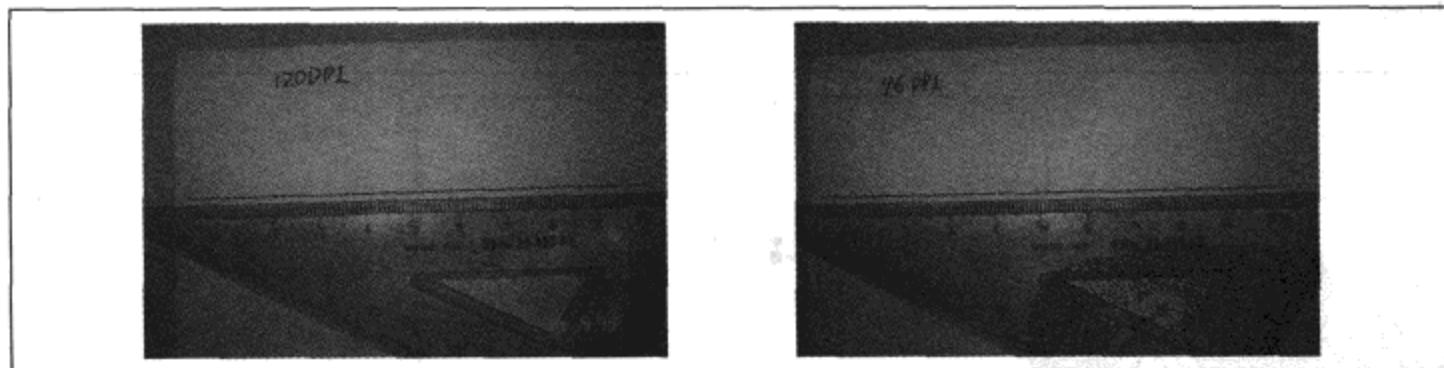


图 15-7 左图为系统设置 120DPI 下打印结果，右图为系统设置 96DPI 下打印结果

(3) 矢量图形

在传统的用户界面中我们经常会采用一个小的位图贴在工具栏中，但是在 WPF 中不这样处理。即使是一个小的图标，WPF 也采用矢量图形来实现。当调整显示器的分辨率或者 DPI 设置后位图由于分辨率不够，可能变得模糊，而矢量图形却不会。如下图所示，在 Vista 的放大镜程序中 WPF 的图形

仍然可以显示得非常完美，而与之对比的非 WPF 程序则出现模糊现象，如图 15-8 所示。

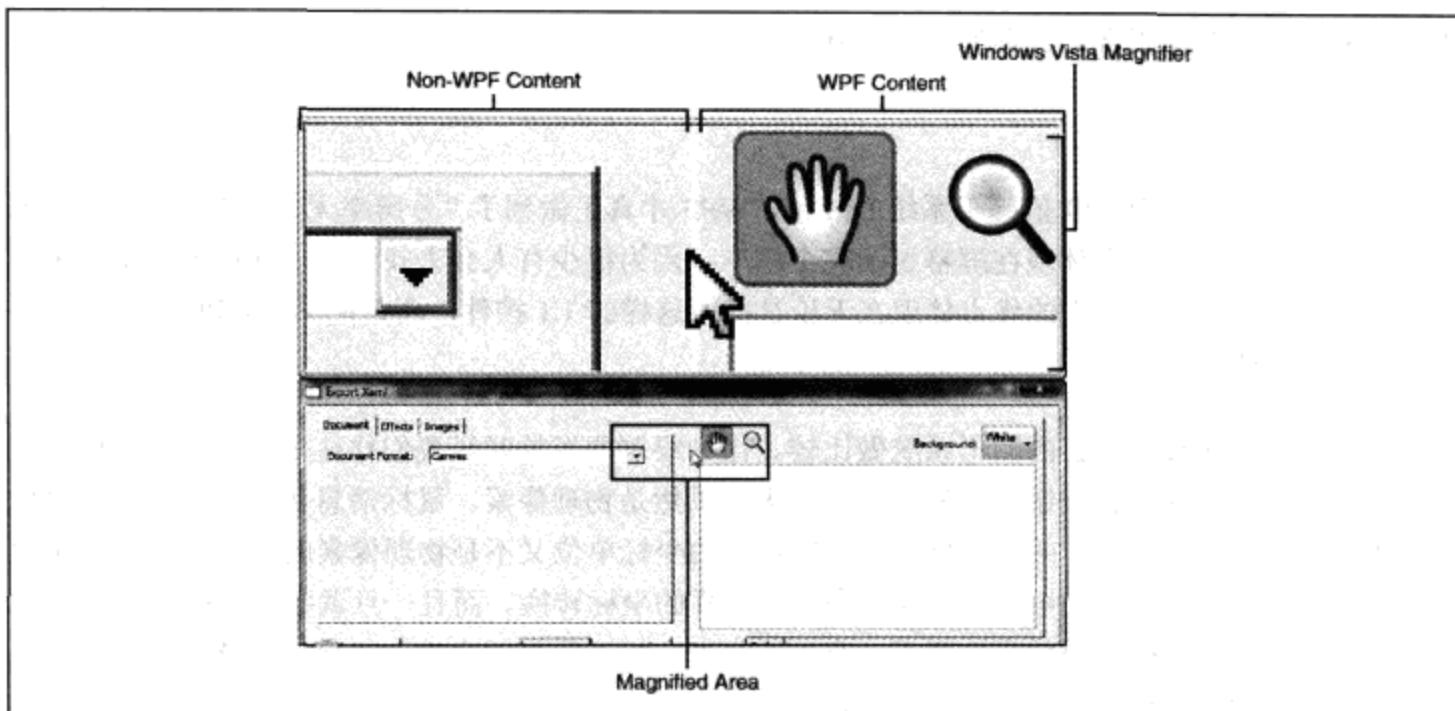


图 15-8 左图为非 WPF 程序，右图为 WPF 程序

15.2.2 坐标系

WPF 主要有两种坐标系，即默认坐标和自定义坐标系。

1. 默认坐标系

一个标准的平面直角坐标系主要包括原点位置、 X 和 Y 轴方向，以及坐标单位。WPF 的默认坐标系原点位置在绘制区域的左上角， X 轴向右， Y 轴向下。坐标单位即设备无关单位 1/96 英寸，如图 15-9 所示。

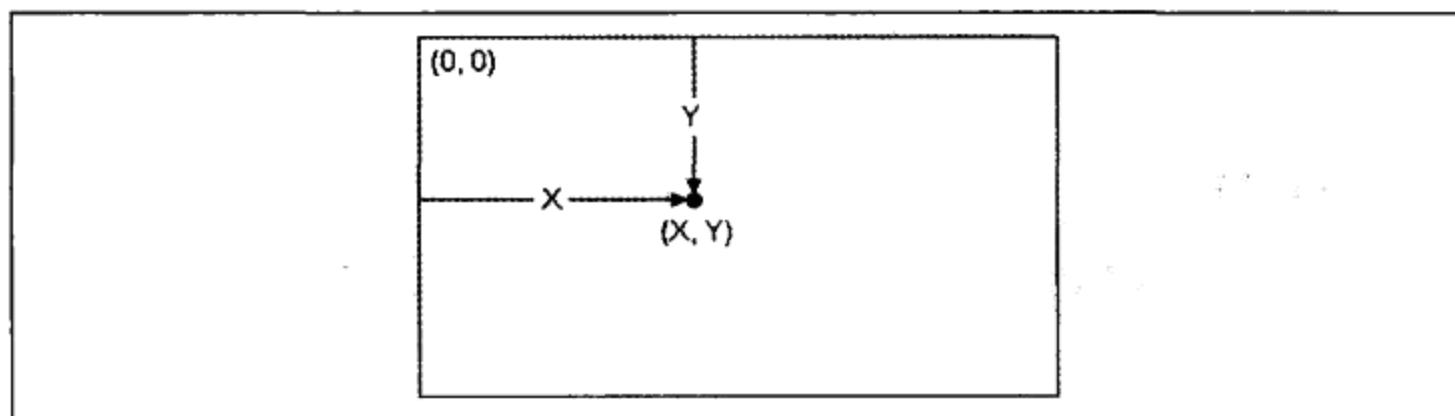


图 15-9 默认坐标系

WPF 中提供的 4 种不同的坐标单位，为防止精度损失，WPF 规定坐标值类型为双精度类型，如表 15-2 所示。

表 15-2 WPF 中提供的 4 种不同的坐标单位

名称	符号	描述
设备独立的像素	px	长度为 1/96 英寸（默认坐标单位）
英寸	in	英寸，1 in = 96 px
厘米	cm	厘米，1 cm = (96/2.54) px，1 英寸 = 2.54 厘米
磅 ²	pt	磅，1 pt = (96/72) px，1 英寸 = 72 磅

2. 自定义坐标系

自定义坐标系主要通过 `Transform` 类来实现，一般图形绘制程序的坐标系往往是原点在左下角，Y 轴正方向向上，X 轴正方向向右。查看代码 15-1 所示的示例（详见 `mumu_customCoordinate` 工程）。

```
<Window x:Class="mumu_customCoordinate.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CustomCoordinate" Height="240" Width="220">
    <Canvas Height="200" Width="200">
        <Canvas.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleY="-1"/>
                <TranslateTransform Y="200"/>
            </TransformGroup>
        </Canvas.RenderTransform>
        <Line X1="0" Y1="0" X2="100" Y2="100" Stroke="Black"
        StrokeThickness="2"/>
        <Button Canvas.Top="93" Canvas.Left="116" FontSize="15"
        Foreground="Red" Name="btn" Content="My Button" Height="25" Width="76" />
    </Canvas>
</Window>
```

代码 15-1 自定义坐标系的实现

一般二维图形绘制的程序会使用 `Canvas` 布局，其 `RenderTransform` 属性用来定义 `Canvas` 坐标系。`ScaleTransform` 变换是将 Y 轴的正方向设置为向上，而 `TranslateTransform` 则是将原点向下移了 200 个 DIU。现在的 `Canvas` 坐标系如图 15-10 所示。

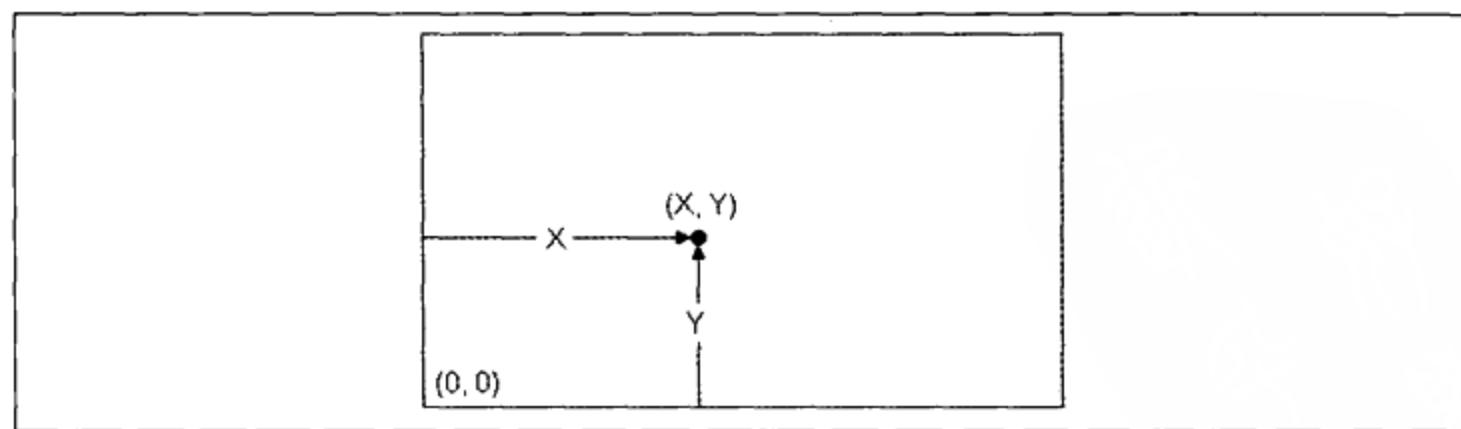


图 15-10 Canvas 坐标系

² 磅：印刷字体大小的一种单位，等于 0.01384 英寸，约为 1/72 英寸。

程序运行结果如图 15-11 所示。



图 15-11 使用 Canvas 布局运行结果

其中的一个明显问题是按钮的文字是倒的，因此必须做一个转换，如代码 15-2 所示。

```
<Button Canvas.Top="93" Canvas.Left="116" FontSize="15" Foreground="Red"
Name="btn" Content="My Button" Height="25" Width="76" >
    <Button.RenderTransform>
        <ScaleTransform ScaleY="-1"/>
    </Button.RenderTransform>
</Button>
```

代码 15-2 转换按钮

运行程序，一切正常，如图 15-12 所示。



图 15-12 转换后的 Canvas 布局运行结果

15.2.3 点和向量

1. 点

在二维空间中某个精确的位置均可用一个坐标点来表示，习惯记为(x,y)，WPF2D 中提供了一个名为“Point”的结构体来保存这些坐标点。

代码 15-3 用来创建一个点的坐标：

```
Point point = new Point(2, 1);
```

代码 15-3 创建一个点的坐标

在 XAML 文件中，Point 对象用文本字符串表示，并用空格或单个逗号分隔其中的数字。如“2 1”或者“2,1”，如代码 15-4 所示。

```
<LineSegment Point="2 1" />
```

代码 15-4 Point 对象的表示

2. 向量

一个向量封装幅度和方向信息，在二维空间中的一般表示如图 15-13 所示。

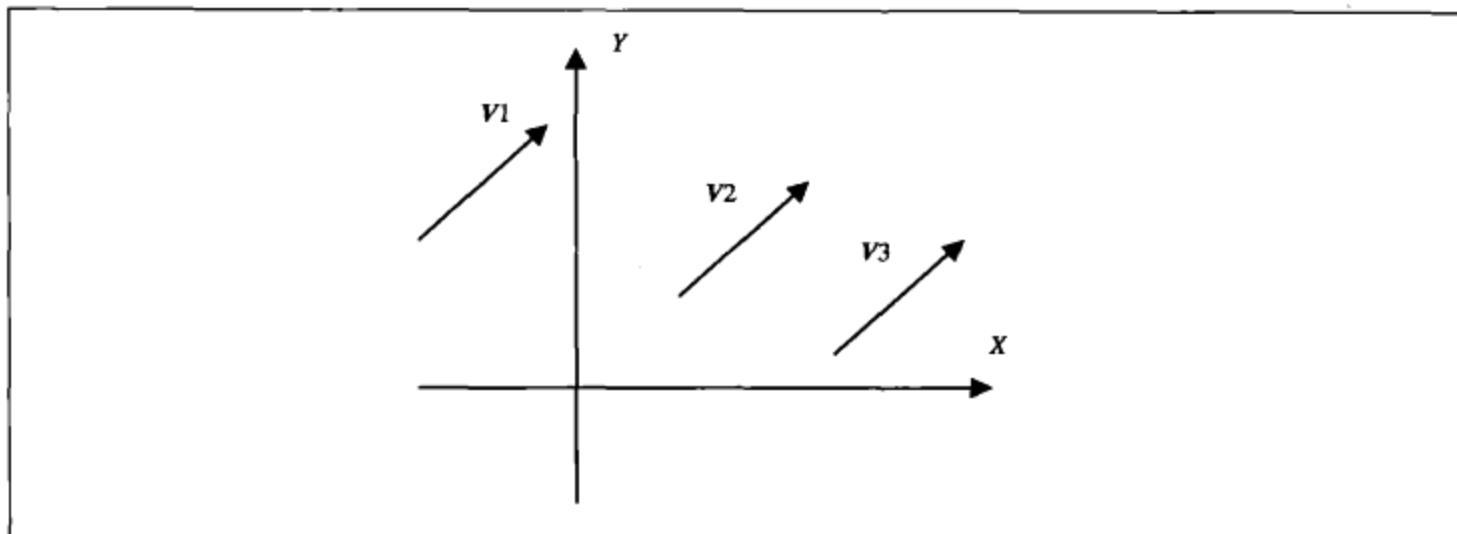


图 15-13 向量在二维空间中的一般表示

向量的幅度用其长度表示，方向用箭头来表示。它通常被展示在某个具体位置，从而容易给人一种错觉。实际上向量没有任何物理位置，与重量或者长度一样。上图中的向量 **V1**、**V2** 和 **V3** 相同，它们具有同样的幅度和方向。

虽然向量没有固定的位置，但是有时我们将其尾部想象为具体的某个点，如点 (x_0, y_0) 。这时向量的头部交于某个唯一的点，记为点 (x_1, y_1) ，那么这个向量表示为 $(x_1 - x_0, y_1 - y_0)$ ；如果向量尾部设在原点，则该向量表示为 (x_1, y_1) 。向量也用 X、Y 坐标值来表示，但是和点有本质区别，本书用粗体来表示向量以与点区分。

在二维坐标系中 WPF 用 Vector 结构体表示向量，其常用的公共属性如下。

- (1) **X**: X 坐标值。
- (2) **Y**: Y 坐标值。
- (3) **Length**: 向量长度，计算公式为 $\sqrt{X^2 + Y^2}$ 。
- (4) **LengthSquared**: 向量长度的平方，计算公式为 $X^2 + Y^2$ 。

LengthSquared 是 X、Y 两个分量的平方和，与 **Length** 相比没有平方根计算。因此如果只比较不同向量之间的幅度，则使用 **LengthSquared** 比 **Length** 稍快。

代码 15-5 用来创建一个向量，向量不能在 XAML 文件中创建。

```
Vector vec = new Vector(1, 2);
```

代码 15-5 创建一个向量

3. 点和向量的运算

通常向量可以通过两个点相减得到，一个点加上一个向量也可以得到另一个点：

$$(x,y) = (x_1,y_1) - (x_0,y_0)$$

$$(x_1,y_1) = (x,y) + (x_0,y_0)$$

如代码 15-6 所示。

```
Point point1 = new Point(2, 1);
Point point2 = new Point(1, 1);
// 向量的值为(1,0)
Vector vecres = point1 - point2;

Vector vec = new Vector(1, 2);
// 点的值为(3,3)
Point pointres = vec + point1;
```

代码 15-6 点和向量之间的运算

在 WPF 中向量和点之间也可以强制转换，如代码 15-7 所示。

```
Vector vec = (Vector)new Point(10, 20);
Point point = (Point)new Vector(10, 20);
```

代码 15-7 向量和点之间的强制转换

通常我们可能更关心向量的方向，而不是共幅度。如果向量的长度为 1，则称为“标准向量”。Vector 结构中提供了一个 Normalize 方法，用于将向量表格规格化后长度为 1。

(1) 向量加法

两个向量之间可以相加，并具有明确的几何意义。在图 15-14 中， V_1 和 V_2 都表示向量。两个向量相加等效于第 1 个向量的头部和第 2 个向量的尾部相连，而向量的和等于第 1 个向量尾部指向第 2 个向量头部所得到的向量。

向量的加法满足加法交换率，交换两个向量的位置，运算结果不变。Vector 中提供的 Add 方法相加两个向量，返回值仍然是一个向量；如果一个向量加一个点，则返回一个点的对象，如代码 15-8 所示。

```
public static Point Add(Vector vector, Point point);
public static Vector Add(Vector vector1, Vector vector2);
```

代码 15-8 用 Add 方法实现向量的加法

Vector 还重载了+运算符，因此向量之间，以及向量和点相加均可使用该运算符。

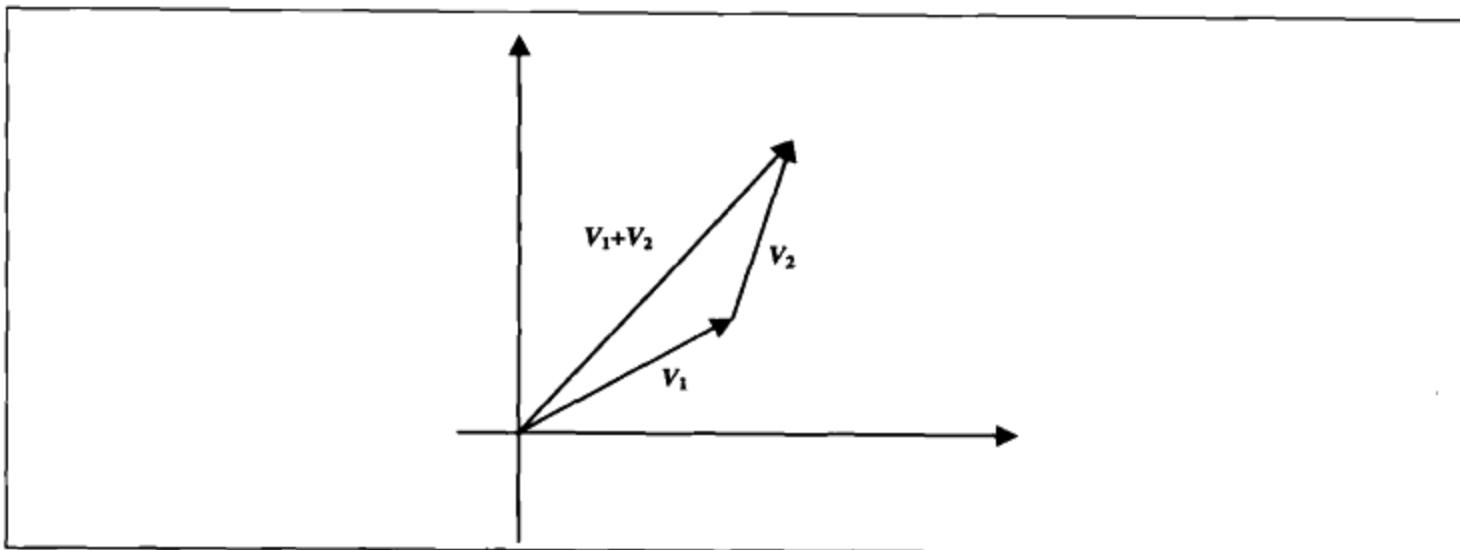


图 15-14 两个向量相加

(2) 向量的减法

向量的减法也有明确的几何意义，两个向量的相减等效于第 2 个向量的头部指向第 1 个向量的头部，如图 15-15 所示。

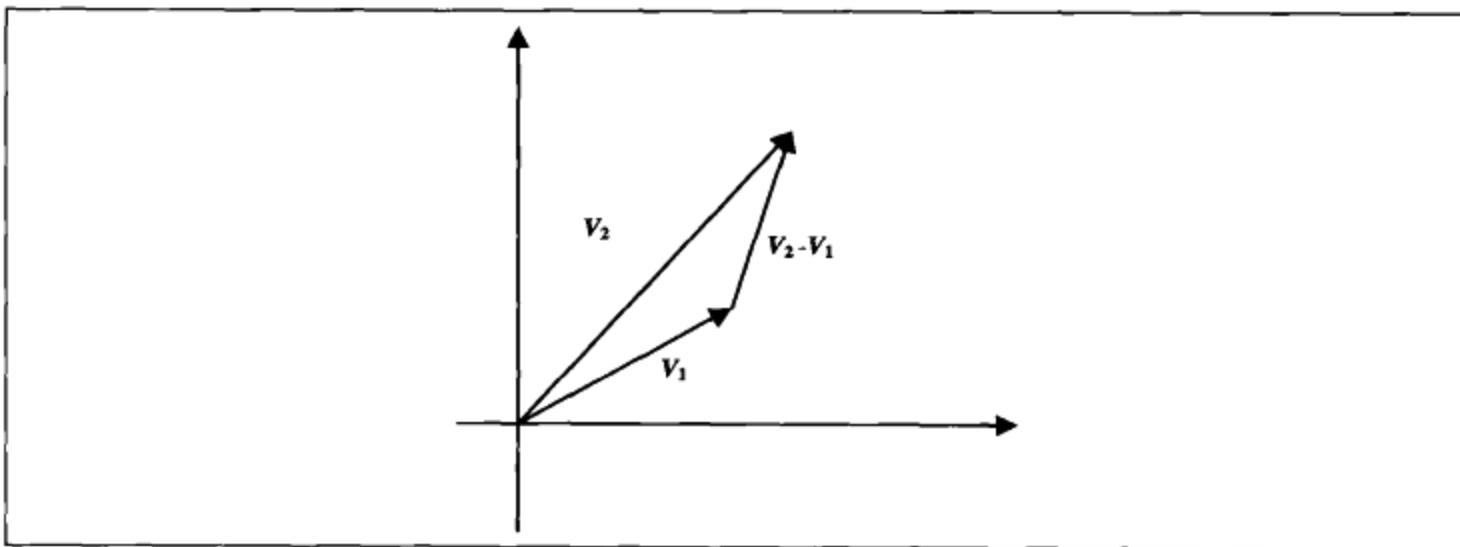


图 15-15 两个向量相减

Vector 中提供了 Subtract 方法，并且重载了一号和负号运算符，结果得到一个新的向量。其幅度与原向量相同，但方向相反。

(3) 向量和标量的乘除法

向量和标量的乘除法即向量的每个分量与标量乘除，在代码 15-9 中的 D 是一个 double 型的标量。

```
Vector vec = (Vector)new Vector(10, 20);
Double D = 10;
Vector vecres = vec * D;
vecres = D * vec;
vecres = vec / D;
```

代码 15-9 向量和标量的乘除法

向量还有两种乘法，分别称为“点乘”(dot product)和“叉乘”(cross product)，我们将在 WPF3D 图形学中讨论。

15.2.4 几何变换

基本的向量几何变换包括缩放、旋转、错切和平移。

1. 缩放

一个点 $P_0(x_0, y_0)$ 在 X 方向上缩放，只需要乘上一个比例因子 s_x 。同理，在 Y 方向上乘上一个比例因子 s_y ，经过缩放变换后会得到一个新的点 $P_1(x_1, y_1)$ 。变换公式如下：

$$\begin{cases} x_1 = x_0 \times s_x \\ y_1 = y_0 \times s_y \end{cases}$$

将该公式写为如下矩阵形式：

$$(x_1, y_1) = (x_0, y_0) \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} = (s_x x_0, s_y y_0)$$

$\begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$ 为缩放矩阵，如 $\begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$ 和 $\begin{pmatrix} 1.5 & 0 \\ 0 & 0.5 \end{pmatrix}$ 是两个缩放矩阵，图 15-16 所示为图形经过缩放矩阵变换后的效果。

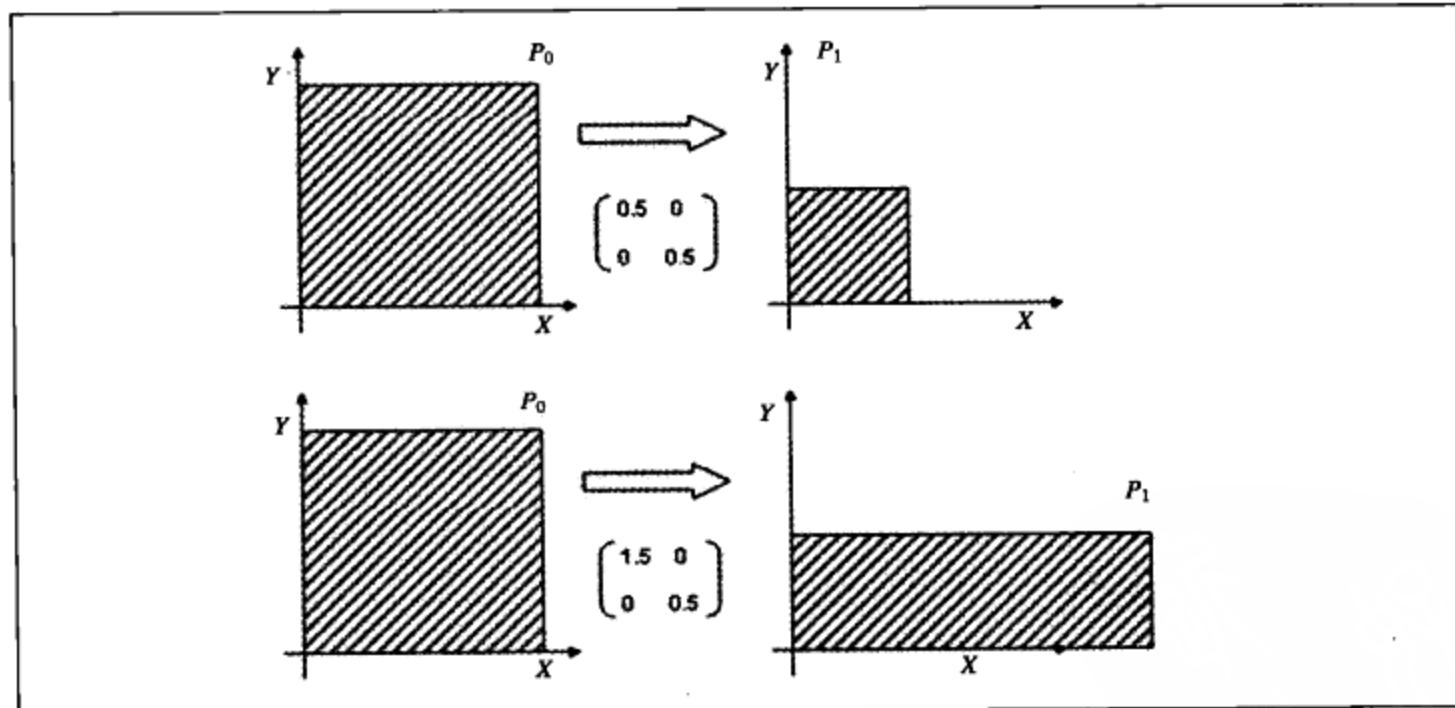


图 15-16 图形经过缩放矩阵变换的效果

2. 旋转

旋转是以某个参考点为圆心，将对象上的各点 $P_0(x_0, y_0)$ 围绕圆心转动一个逆时针角度 θ 变为新的坐标 $P_1(x_1, y_1)$ 的变换，如图 15-17 所示。

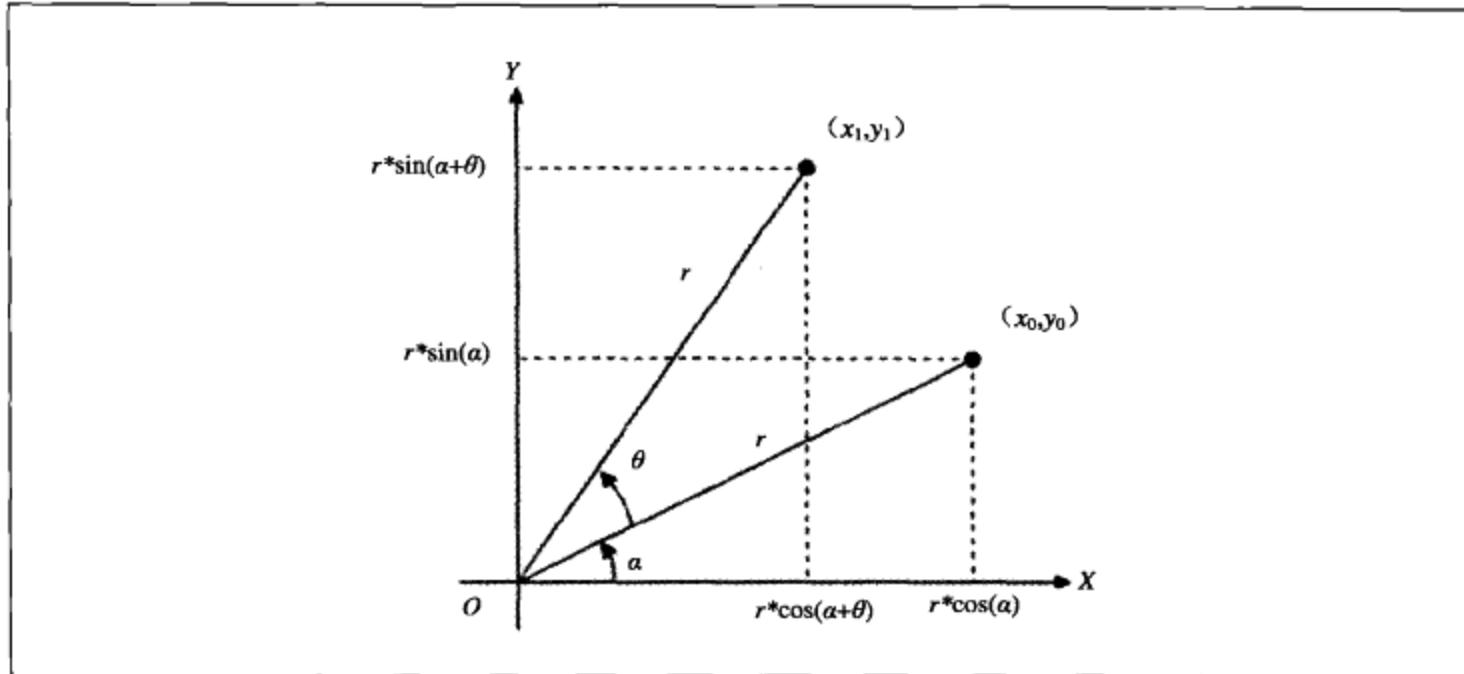


图 15-17 旋转示意图

图中所示的 (x_0, y_0) 可以用如下公式表示：

$$\begin{cases} x_0 = r \cos \alpha \\ y_0 = r \sin \alpha \end{cases}$$

(x_1, y_1) 可以用如下公式表示：

$$\begin{cases} x_1 = r \cos(\alpha + \theta) = r \cos \alpha \cos \theta - r \sin \alpha \sin \theta = x_0 \cos \theta - y_0 \sin \theta \\ y_1 = r \sin(\alpha + \theta) = r \sin \alpha \cos \theta + r \cos \alpha \sin \theta = x_0 \sin \theta + y_0 \cos \theta \end{cases}$$

用矩阵的形式来表示则是：

$$(x_1, y_1) = (x_0, y_0) \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = (x_0 \cos \theta - y_0 \sin \theta, x_0 \sin \theta + y_0 \cos \theta)$$

3. 错切

错切即使图形沿某轴方向的坐标发生变化，而与之垂直方向的轴坐标值不变，如图 15-18 所示。

下面两个图均将点 $P_0(x_0, y_0)$ 沿 X 轴错切 45° 角变成新的 $P_1(x_1, y_1)$ ，这个 45° 角的几何意义容易误解为左侧图中的角度。正确的几何意义是如果 $\theta_x=45$ ，则在 X 轴方向上错切的长度为 $\tan 45 \times y$ ；如果 $\theta_y=45$ ，则在 Y 轴方向上错切的长度为 $\tan 45 \times x$ 。错切变换可以用如下公式表示：

$$\begin{cases} x_1 = x_0 + y_0 \times \tan \theta_x \\ y_1 = y_0 + x_0 \times \tan \theta_y \end{cases}$$

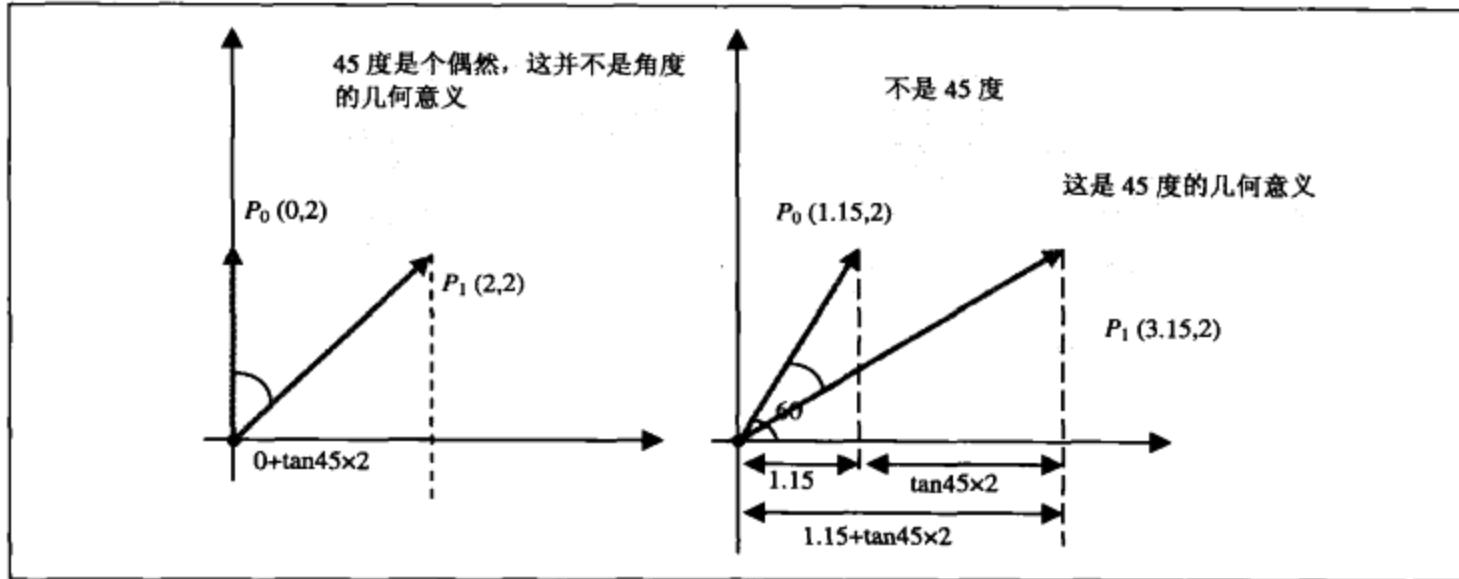


图 15-18 错切示意

当 $\tan\theta_y=0$ 时, $x_1=x_0+\tan\theta_x \times y_0, y_1=y_0$ 。此时 y 坐标不变, x_1 坐标随 (x_0, y_0) 及变换系数 $\tan\theta_x$ 执行线性变换, 在 X 轴方向执行错切变换; $\tan\theta_x>0$, 沿 X 轴正方向执行错切变换, $\tan\theta_x<0$, 沿 X 轴负方向执行错切变换。

同理 $\tan\theta_x=0$ 时, $x_1=x_0, y_1=y_0+x_0 \times \tan\theta_y$ 。此时 x 坐标不变, y_1 坐标随 (x_0, y_0) 及变换系数 $\tan\theta_y$ 执行线性变换, 在 Y 轴方向执行错切变换; $\tan\theta_y>0$, 沿 Y 轴正方向执行错切变换, $\tan\theta_y<0$, 沿 Y 轴负方向执行错切变换;

用矩阵的形式来表示则是:

$$(x_1, y_1) = (x_0, y_0) \begin{pmatrix} 1 & \tan\theta_y \\ \tan\theta_x & 1 \end{pmatrix} = x + \tan\theta_x y, y + \tan\theta_y x$$

4. 平移

平移是将对象从一个位置 $p_0(x_0, y_0)$ 移到另一个位置 $p_1(x_1, y_1)$ 的变换, 其公式为:

$$\begin{cases} x_1=x+dx \\ y_1=y+dy \end{cases}$$

然而无论如何均无法用矩阵形式来表示平移变换。因为平移无法表示成一个矩阵。因此用一种统一的形式来进行几何变换的想法就此破灭了。比如将一个点先缩放, 再旋转。实际上是缩放矩阵和旋转矩阵的乘法运算。缩放、错切和旋转的组合都可以用矩阵相乘的方式来实现。但是一遇到平移变换, 建立起来的规则就此打破, 这确实是一件让人遗憾的事情。对于平移来说, 怎样才能将其纳入像其他变换所采用的矩阵简便形式呢? 马上我们将找到答案。

15.2.5 齐次坐标

1. 用三维错切变换实现二维平移变换

前面的缩放、旋转和错切均可用下面的变换矩阵来表示：

$$(x_1, y_1) = (x_0, y_0) \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$$

只不过缩放时， $M_{11}=s_x$ ， $M_{12}=0$ ， $M_{21}=0$ 且 $M_{22}=s_y$ 。同理旋转和错切的 M_{11} 、 M_{12} 、 M_{21} 和 M_{22} 会取不同值。

比如，如图 15-19 所示为位于原点的 2D 小房屋的几种几何变换。

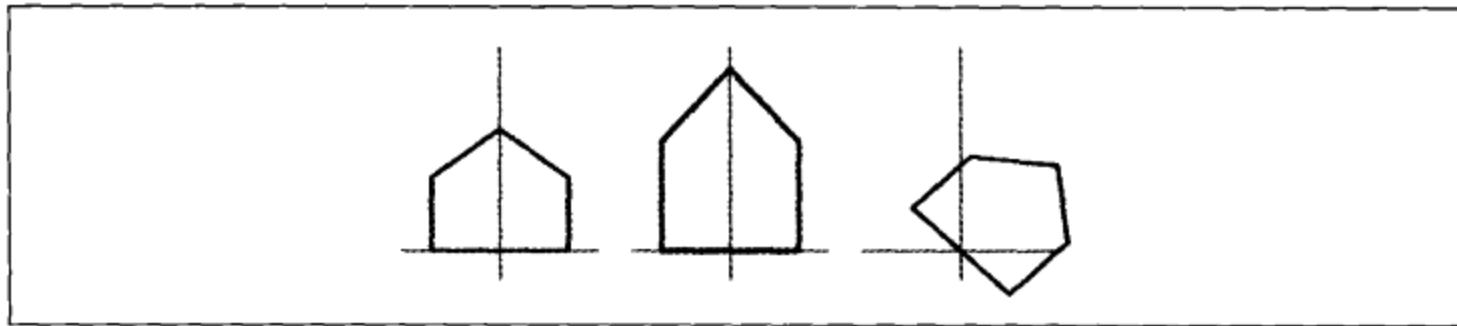


图 15-19 位于原点的 2D 小房屋的几何变换

图中从左至右第 1 个是房屋的原图，第 2 个是房屋在 Y 方向上放大 50%，第 3 个是房屋绕原点顺时针旋转 45° 。这些变换都可以用矩阵 $\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$ 来实现，然而平移无法用 2×2 矩阵来实现。

如果我们用一个 3×3 矩阵，如 $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1.25 & 0.25 & 1 \end{pmatrix}$ ，那么该矩阵隐含了下面的变换公式：

$$\begin{cases} x_1 = x_0 + 1.25z \\ y_1 = y_0 + 0.5z \\ z_1 = z \end{cases}$$

实际上这是一个三维错切变换， X 和 Y 会沿 Z 轴方向执行错切变换，如图 15-20 所示。房屋的面 $H1$ 会沿 $Z=1$ 的平面上错切成了一个新的房屋面 $H2$ ，这时实际已经在 X 和 Y 轴方向分别平移了 1.25 和 0.5。

这时的变换公式已经变为 x 和 y 的平移公式：

$$\begin{cases} x_1 = x_0 + 1.25 \\ y_1 = y_0 + 0.5 \end{cases}$$

上面的推导说明，三维错切变换可以有效地实现二维对象的平移变换，这正是将平移纳入到变换矩阵的关键。无论何时，如果在二维空间中绘图，可以想象为在三维空间下 z 为 1 的平面上绘制所有

二维图形。

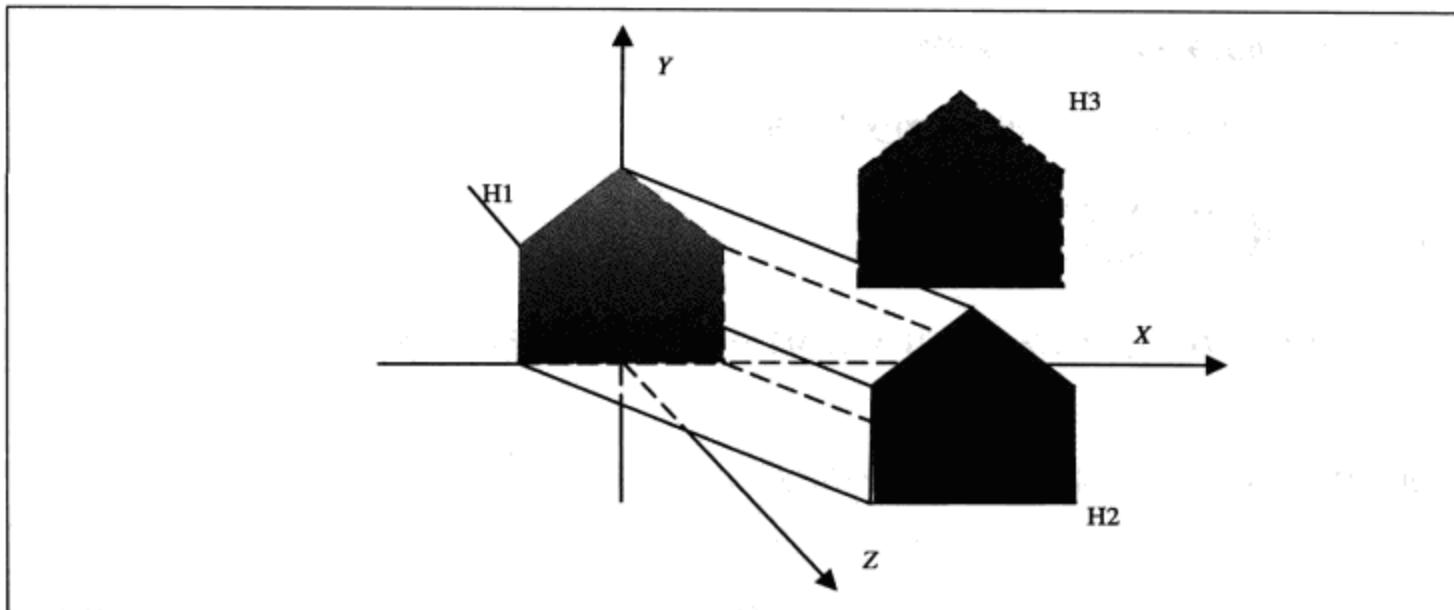


图 15-20 三维的错切变换

由于现在是在三维空间绘图，所以实际处理的三维点坐标就是 $(x,y,1)$ ，而不是 (x,y) 。原来的 2×2 的变换矩阵变成 3×3 的变换矩阵，即：

$$\begin{pmatrix} M_{11} & M_{12} & 0 \\ M_{21} & M_{22} & 0 \\ \text{OffsetX} & \text{OffsetY} & 1 \end{pmatrix}$$

变换公式如下：

$$\begin{cases} x_1 = M_{11} \times x_0 + M_{21} \times y_0 + \text{OffsetX} \\ y_1 = M_{12} \times x_0 + M_{22} \times y_0 + \text{OffsetY} \end{cases}$$

如何理解这个三维矩阵的最后一列呢？如果在一个三维空间中，则矩阵的最后一列是可以用于 Z 坐标的缩放旋转等。但是在二维空间中则没有任何作用，实际上这个 3×3 的变换矩阵规定了最后一列必须为 $0,0,1$ 。让我们暂时假设允许设置矩阵第 3 列中的元素，如下所示：

$$(x_1 \ y_1 \ 1) = (x_0 \ y_0 \ 1) \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ \text{OffsetX} & \text{OffsetY} & M_{33} \end{pmatrix}$$

此时的变换公式为：

$$\begin{cases} x_1 = M_{11} \times x_0 + M_{21} \times y_0 + \text{OffsetX} \\ y_1 = M_{12} \times x_0 + M_{22} \times y_0 + \text{OffsetY} \\ z_1 = M_{13} \times x_0 + M_{23} \times y_0 + M_{33} \end{cases}$$

显然变换后的点实际上跳出了 $z=1$ 的平面，由于处理的仍然是二维图形，所以需要将变换后的点还原在 $z=1$ 的平面上。最简单的方法就是将所有 3 个坐标值除以 z_1 ，得到：

$$(x_1, y_1, z_1) = \begin{pmatrix} x_1 & y_1 & z_1 \\ z_1 & z_1 & z_1 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ z_1 & z_1 \end{pmatrix}, 1$$

这样又回到 z 为 1 的平面，但是必须要小心 z_1 不能为 0；否则变换后的坐标值为无穷大。在这里我们为二维点添加了一个额外的坐标，即齐次坐标。

2. 齐次坐标下的点、向量和变换矩阵

齐次坐标的概念是德国数学家 August Ferdinand Möbius (1790 年~1868 年) 提出的，非常著名的莫比乌斯带（如图 15-21 所示）³就是以其名字命名的。

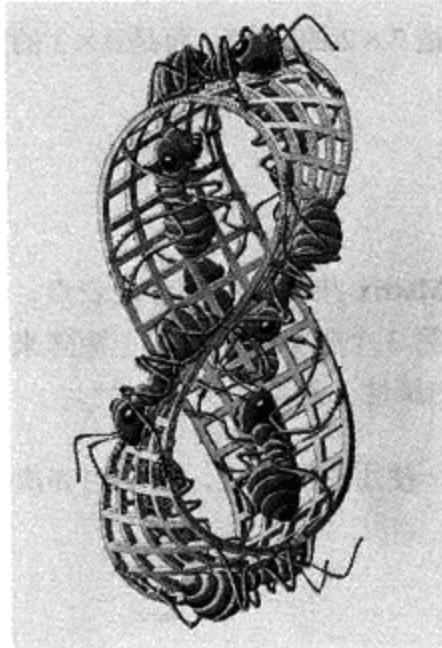


图 15-21 莫比乌斯带

齐次坐标指用 $n+1$ 维的量来表示 n 维的量，在这里不再用 z 来表示额外的坐标，而用 w 来表示。

二维的空间点可以用 $(x, y, 1)$ 来表示 (x, y) 坐标，实际上二维坐标及其相对应的齐次坐标关系的公式如下：

$$(x/w, y/w) = (x, y, w) = (x/w, y/w, 1)$$

从上述公式可以看出一个二维坐标及其相对应的齐次坐标是一对多的关系，同时 w 也有非常明确的几何意义。

当 $w=1$ 时， (x, y, w) 表示二维点坐标值为 (x, y) 。

当 $w=0$ 时，表示无穷远；

当 $w=\infty$ 时，表示为原点。

³ 一个普通的纸带在经过 180° 弯曲后把两头粘连起来，便具有魔术般的性质。普通纸带具有两个面，一个正面；一个反面。而这样的纸带只有一个面，一只小虫可以爬遍整个曲面，而不必跨过其边缘。

二维向量的齐次坐标并不是 1，而是 0，这是向量区别于点的一个关键。即使用 $(x,y,0)$ 来表示 (x,y) 向量。这一点应该易于理解，因为向量本身可以通过两个点相减得到。通过运算也可以发现其第 3 个坐标为 0，如下所示：

$$(x_1, y_1, 1) - (x_0, y_0, 1) = (x, y, 0)$$

在几何变换中，齐次坐标为 1 或 0 对缩放、旋转和错切变换没有影响，只有在平移变换中才会有显著的影响。当齐次坐标为 1 时，平移变换后 x 和 y 值会分别增加 `OffsetX` 和 `OffsetY`；为 0 时，平移变换后 x 和 y 值前后并没有变化。这样的结果也符合点和向量的特性，因为向量无论如何平移，其值不会发生变化。

具体到矩阵，前面已经说过它不再是 2×2 的矩阵，而是 3×3 的矩阵：

$$\begin{pmatrix} M_{11} & M_{12} & 0 \\ M_{21} & M_{22} & 0 \\ \text{Offset}_X & \text{Offset}_Y & 1 \end{pmatrix}$$

在 WPF 中分别用 `Point`、`Vector` 和 `Matrix` 共 3 个结构体代表点、向量和矩阵。为了提高性能和节省存储空间，WPF 省略了点和向量的第 3 个坐标值。而对于矩阵来说，最后一列也是固定为 0, 0, 1。因此 WPF 的 `Matrix` 只有前面两列的属性，省略了最后一列。

齐次坐标的引入可以使几何变化统一使用矩阵方式，我们用齐次坐标重新考察基本的几何变换，首先考察平移变换。

3. 齐次坐标下的平移变换

引入了齐次坐标后平移变换可以写为如下公式：

$$(x_1 \ y_1 \ 1) = (x_0 \ y_0 \ 1) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{pmatrix}$$

下面的代码是分别将点和向量在 X 轴方向上平移 5 个单位并在 Y 轴上平移 10 个单位，从运算的结果也可以看出对于向量来说任何的平移变换不会改变其值，如代码 15-10 所示。

```
// 矩阵为 1 0 0
//          0 1 0
//          5 10 1
Matrix translateMatrix = new Matrix(1, 0, 0, 1, 5, 10);
//
// 对点执行变换
//
Point P0 = new Point(10, 20);

// P1 为 (15, 30).
Point P1 = translateMatrix.Transform(P0);
//
// 对向量执行变换
//
```

```

Vector V0 = new Vector(10, 20);

// 向量为(10,20),没有任何变换
Vector V1 = translateMatrix.Transform(V0);

```

代码 15-10 分别将点和向量在 X 轴方向上平移 5 个单位并在 Y 轴上平移 10 个单位

平移矩阵有一些比较有意思的特性。例如，如果移动 $T(d_{x1} \ d_{y1})$ 后移动 $T(d_{x2} \ d_{y2})$ ，那么平移矩阵是否为 $T(d_{x1}+d_{x2} \ d_{y1}+d_{y2})$ ？推导如下：

$$P_1 = P \cdot T(d_{x1} \ d_{y1})$$

$$P_2 = P_1 \cdot T(d_{x2} \ d_{y2})$$

$$P_2 = P \cdot T(d_{x1} \ d_{y1}) \cdot T(d_{x2} \ d_{y2})$$

$$T(d_{x1} \ d_{y1}) \cdot T(d_{x2} \ d_{y2}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_{x1} & d_{y1} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_{x2} & d_{y2} & 1 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_{x1}+d_{x2} & d_{y1}+d_{y2} & 1 \end{pmatrix} = T(d_{x1}+d_{x2} \ d_{y1}+d_{y2})$$

推论证明 $T(d_{x1} \ d_{y1}) \times T(d_{x2} \ d_{y2}) = T(d_{x1}+d_{x2} \ d_{y1}+d_{y2})$ 成立。

4. 齐次坐标下的缩放

缩放变换用如下公式表示：

$$(x_1 \ y_1 \ 1) = (x \ y \ 1) \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$P_1 = P_0 \cdot S(s_x \ s_y)$$

代码 15-11 是分别将点和向量在 X 轴方向缩小为 0.5 倍并在 Y 轴上扩大为 1.5 倍，如图 15-22 所示。这一次矩阵不是直接通过构造函数来设置 $M11$ 、 $M12$ 、 $M21$ 、 $M22$ 、 $OffsetX$ 和 $OffsetY$ 值，而是通过 Matrix 的 Scale 方法来生成缩放矩阵。Scale 的第 1 个参数表示 X 轴上的缩放系数，第 2 个参数表示 Y 轴上的缩放系数。

```

Matrix scaleMatrix = Matrix.Identity;

// 矩阵为 0.5 0 0
//          0 1.5 0
//          0 0 1
scaleMatrix.Scale(0.5, 1.5);
//
// 对点进行转换.
//
Point P0 = new Point(10, 20);

```

```

// P1 为(5,30).
Point P1 = scaleMatrix.Transform(P0);
//
// 对向量进行转换..
//
Vector V0 = new Vector(10, 20);

// V1 为(5,30)
Vector V1 = scaleMatrix.Transform(V0);

```

代码 15-11 分别将点和向量在 X 轴方向缩小成 0.5 倍并在 Y 轴上扩大为 1.5 倍

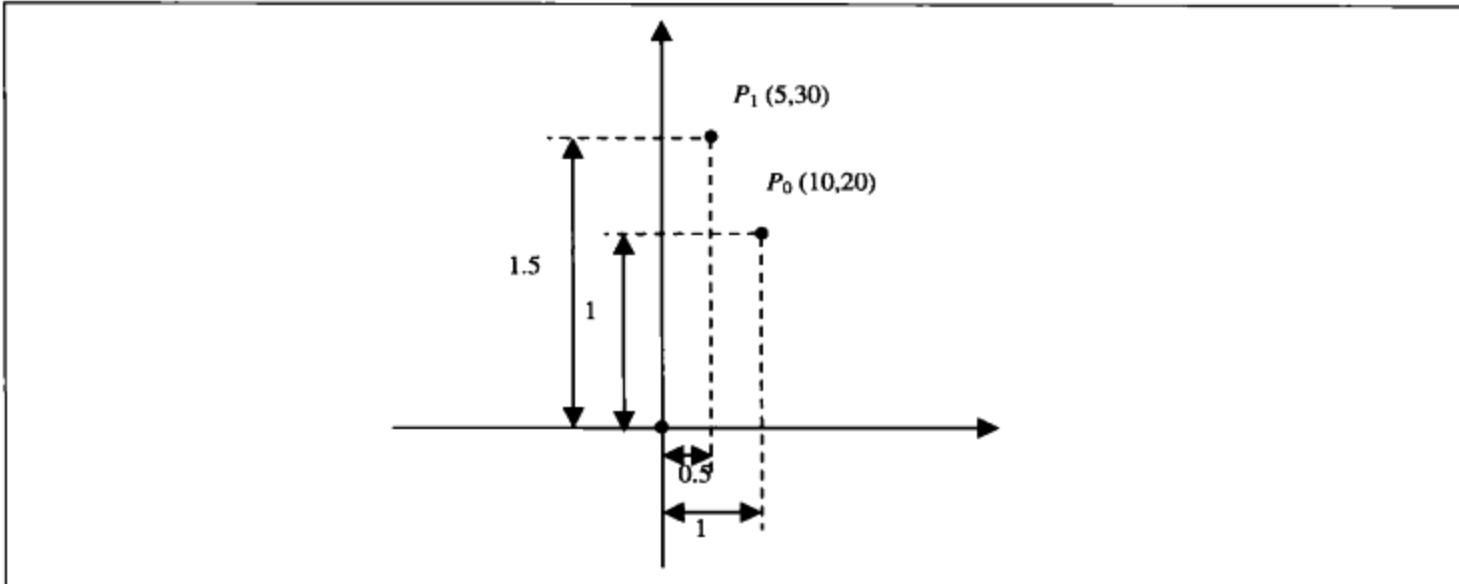


图 15-22 齐次坐标下的缩放扩大

除了 Scale 方法以外，Matrix 还提供了一个 ScaleAt 方法。Scale 是从原点缩放，而 ScaleAt 以某一点缩放。代码 15-12 对点和向量执行缩放变换，缩放系数同前例，不同的是以 (1,1) 点执行缩放变换。

```

Matrix scaleMatrix = Matrix.Identity;
// 矩阵为 0.5 0 0
// 0 1.5 0
// 0.5 -0.5 1
scaleMatrix.ScaleAt(0.5, 1.5, 1, 1);

//
// 对点进行转换.
//
Point P0 = new Point(10, 20);
// P1 为(5.5,29.5).
Point P1 = scaleMatrix.Transform(P0);
//
// 对向量进行转换..
//
Vector V0 = new Vector(10, 20);

// V1 为(5,30).
Vector V1 = scaleMatrix.Transform(V0);

```

代码 15-12 对点和向量执行缩放变换

经过缩放变换后，向量为(5,30)，这个值和从原点缩放变换的值一样。而点的坐标为(5.5,29.5)，该值

与从原点缩放变换不同。实际从(1,1)点进行缩放变换隐含了如下3步。

- (1) 将点和向量执行平移变换 $T0(-1,-1)$, 使(1,1)点变成原点。
- (2) 对点和向量执行缩放变换 $S1(0.5,1.5)$ 。
- (3) 将点和向量再执行一次平移变换 $T2(1,1)$, 恢复原来的原点。

因此变换矩阵为:

$$\text{scaleMatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix} \times \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 4.5 & 0 \\ 0.5 & -0.5 & 1 \end{pmatrix}$$

对于向量来说, 平移变换不会改变向量的值。因此向量无论以原点, 还是以(1,1)点执行缩放变换, 值不会改变。

如果缩放 $S(s_{x1} s_{y1})$ 后缩放 $S(s_{x2} s_{y2})$, 会不会是 $S(s_{x1}s_{x2} s_{y1}s_{y2})$? 推导证明如下:

$$P_1 = P \cdot S(s_{x1} s_{y1})$$

$$P_2 = P_1 \cdot S(s_{x2} s_{y2})$$

$$P_2 = P \cdot S(s_{x1} s_{y1}) \cdot S(s_{x2} s_{y2})$$

$$(s_{x1} s_{y1}) \cdot S(s_{x2} s_{y2}) = \begin{pmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} s_{x1} + s_{x2} & 0 & 0 \\ 0 & s_{y1} + s_{y2} & 0 \\ 0 & 0 & 1 \end{pmatrix} = S(s_{x1} + s_{x2} s_{y1} + s_{y2})$$

证明 $S(s_{x1} s_{y1}) \cdot S(s_{x2} s_{y2}) = S(s_{x1}s_{x2} s_{y1}s_{y2})$ 成立。

5. 齐次坐标下的旋转

旋转变换用如下公式表示:

$$(x_1 \ y_1 \ 1) = (x_0 \ y_0 \ 1) \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$P_1 = P_0 R(\theta)$$

代码 15-13 分别逆时针旋转点和向量 90° 。

```
// 旋转矩阵为 (cos(90), sin(90), -sin(90), cos(90), 0, 0), 逆时针旋转 90°
```

```

// 矩阵为 cos90 sin90 0
//          -sin90 cos90 0
//              0      0 1
Matrix rotateMatrix = new Matrix(0, 1, -1, 0, 0, 0);
//
// 对点进行转换.
//
Point P0 = new Point(10, 20);
// P1 为(-20,10).
Point P1 = rotateMatrix.Transform(P0);
//
// 对向量进行转换
//
Vector V0 = new Vector(10, 20);

// 向量为(-20,10).
Vector V1 = rotateMatrix.Transform(V0);

```

代码 15-13 分别对点和向量逆时针旋转 90 度

当使用 Rotate 方法时，点和向量以原点旋转，旋转变换后值仍然相同。Matrix 提供了 RotateAt 方法，用来围绕一个指定的点执行旋转变换。

如果执行一次旋转变换 $R(\theta_1)$ 后执行一次旋转变换 $R(\theta_2)$ ，其旋转矩阵是否为 $R(\theta_1+\theta_2)$ ？推导证明如下：

$$P_1 = P \cdot R(\theta_1)$$

$$P_2 = P_1 \cdot R(\theta_2)$$

$$P_2 = P \cdot R(\theta_1) \cdot R(\theta_2)$$

$$\begin{aligned}
R(\theta_1) \cdot R(\theta_2) &= \begin{pmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & \sin\theta_2 & 0 \\ -\sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \\
&= \begin{pmatrix} \cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2 & \cos\theta_1\sin\theta_2 - \sin\theta_1\cos\theta_2 & 0 \\ -\sin\theta_1\cos\theta_2 - \cos\theta_1\sin\theta_2 & \cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \\
&= \begin{pmatrix} \cos(\theta_1+\theta_2) & \sin(\theta_1+\theta_2) & 0 \\ -\sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 \\ 0 & 0 & 1 \end{pmatrix} = R(\theta_1+\theta_2)
\end{aligned}$$

推论证明 $R(\theta_1) \cdot R(\theta_2) = R(\theta_1+\theta_2)$ 成立。

6. 组合变换

Matrix 为错切变换提供了 Skew 方法。

由于齐次坐标的引入，所有的几何变换都可以用矩阵形式来实现，因此才可以执行组合变换。前面所说的 ScaleAt 和 RotateAt 方法隐含了多次变换，因此均为组合变换。

代码 15-14 是分别对点和向量执行错切、平移和旋转的一个组合变换。

```
Point P0 = new Point(10, 20);
Vector V0 = new Vector(10, 20);
// 转换后的矩阵为 0 1 0
//           -1 1 0
//           -10 5 1
Matrix compositionMatrix = Matrix.Identity;
compositionMatrix.Skew(45, 0);
compositionMatrix.Translate(5, 10);
compositionMatrix.Rotate(90);

// 转换后的 P1 为 -30 35
Point P1 = compositionMatrix.Transform(P0);
// 转换后的 V1 为 -20 30
Vector V1 = compositionMatrix.Transform(V0);
```

代码 15-14 分别对点和向量做了错切、平移和旋转的一个组合变换

该矩阵计算公式如下：

$$\text{compositionMatrix} = \begin{pmatrix} 1 & \tan 0 & 0 \\ \tan 45 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 10 & 1 \end{pmatrix} \times \begin{pmatrix} \cos 90 & 0 & 0 \\ -\sin 90 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 1 & 0 \\ -10 & 5 & 1 \end{pmatrix}$$

在执行组合变换时，矩阵相乘的顺序非常重要，不能前后颠倒；否则会得出错误的结果。Matrix 的 Skew、Translate 和 Rotate 实际是矩阵的左乘方法，它也提供了一种右乘的方法，以 Prepend 为后缀。该矩阵变换也可以按照如下方式实现，如代码 15-15 所示。

```
Matrix compositionMatrix = Matrix.Identity;
compositionMatrix.RotatePrepend(90);
compositionMatrix.TranslatePrepend(5, 10);
compositionMatrix.SkewPrepend(45, 0);
```

代码 15-15 矩阵的变换

15.2.6 WPF 中的对象变换

前面讨论的矩阵还只是对点和向量执行几何变换，WPF 中的图形和元素需要借助 Transform 及其派生类。

负责 WPF 的图形和元素几何变换的类称为“Transform”，它是一个抽象类。WPF 提供了 6 个类派生自 Transform 的类，如表 15-3 所示。

表 15-3 WPF 提供的 6 个派生于 Transform 类的一览表

名称	描述	重要属性
TranslateTransform	平移变换	X,Y 表示是在 X 和 Y 轴上的平移量
RotateTransform	旋转变换	Angle 表示旋转的角度 CenterX 和 CenterY 表示旋转的中心点坐标

名称	描述	重要属性
ScaleTransform	缩放变换	ScaleX 和 ScaleY 表示在 X 和 Y 轴上的缩放比例 CenterX 和 CenterY 表示缩放的原点坐标
SkewTransform	错切变换	AngleX 和 AngleY 表示错切变换的角度 CenterX 和 CenterY 表示错切变换的原点
MatrixTransform	矩阵变换	Matrix 表示变换矩阵
TransformGroup	当对一个图形或者元素需要执行一组变换时，可以通过 TransformGroup 组合若干个变换，实现组合变换	

Transform 可以应用到 WPF 类型中，如表 15-4 所示。

表 15-4 应用 Transform 类的类型

类型	Transformation 属性
Brush	Transform 及 RelativeTransform
ContainerVisual	Transform
DrawingGroup	Transform
FrameworkElement	RenderTransform 及 LayoutTransform
Geometry	Transform
TextEffect	Transform
UIElement	RenderTransform

随书代码中的 `mumu_matrixtransform` 工程对一个矩形进行了平移、缩放、旋转和组合变换，并且使用 MatrixTransform 来模拟这些变换。

15.3 WPF 的二维图形架构（第二块石壁）

第一块石壁——二维图形的数学基础算是学完了。对于木木来说已经是头昏脑胀，两眼昏花了，于是稍作洗漱，就上床睡觉了。第二天一大早，木木心里惦记着第二块石壁，胡乱吃了一点东西，就翻开蓉儿的画像，沿老路去看那第二块石壁了。第二块石壁原来大块的石灰白上也写上了东西，不过只有一个字“推”。木木推了推石壁，“咣当”一声响之后，竟然是一个月洞门，门旁壁上凿着四字：“琅擐福地”，这四个字用笔纵逸，清刚峭拔，但是又透着缠绵婉约，宛若一个持剑的少女，七分英姿中透着三分温柔。

走进月洞门内。一踏进门，洞中列满一排排的木制书架，可是架上却空洞洞地连一本书册也无。木木走近，只见书架上贴满了签条，尽是“大理段氏”、“昆仑派”、“少林派”、“四川青城派”、“山东蓬莱派”、“丐帮”等等名称。但在“少林派”的签条下注“缺易筋经”，在“丐帮”的签条下注“缺降龙十八掌”，在“大理段氏”的签条下注“缺一阳指法、六脉神剑剑法，憾甚”的字

样。想像当年架上所列，应该是各门各派武功的图谱经籍。

木木一拍大脑，想起这是什么地方了。这正是当年神仙姊姊练功的石室（具体请参见《天龙八部：第二章 玉壁月华明》）。神仙姊姊当年一定是想学什么武功了，就曼妙地飘到了“琅擐福地”，从书架上抽取一本武功秘籍，然后借着月光起舞弄清影…木木想得痴了，神仙姊姊果真是集美貌与聪明一体的女人，真是偶的神啊~。随即转念一想：罪过，罪过，我已经是老婆的人了，蓉儿待我这般好，怎么还能有这般心思。于是不再胡思乱想，仍是一排一排书架看过去，看看是否有什么玄机。走到最后一排，木木已经失望了，全部空空如也。只好失望地离开了“琅擐福地”，返回住处。

回来已快中午，不久蓉儿就到山顶给木木送饭。蓉儿见木木闷闷不乐，于是问起。木木也不再隐瞒，一五一十地告诉了老婆蓉儿，只是略去了对神仙姊姊的胡思乱想。蓉儿听完，稍加思索，哈哈笑到：“木木，这个空空的书架就是奥秘所在啊。你想，倘若没有这个书架，神仙姊姊如何去学这各门各派的武学？这个书架正是她的武学体系框架，她想学什么功夫，只需要了解该功夫在书架的哪一部分，然后取下来练习即可。即使缺了‘降龙十八掌’、‘一阳指’这样的高深功夫，以后有机缘能够得到，也可以顺利地纳入到她的体系框架当中。”黄蓉又顿了顿，说到：“我想 WPF 的图形框架就是你的书架，如果想学好二维图形，势必需要先把你的书架建立起来。”木木恍然大悟。

WPF 的图形框架与以往的图形框架（MFC，WinForm）其中一个很大的区别是，WPF 采用的是保留模式绘图，而以往的图形框架采用的是立即模式绘图。

15.3.1 立即模式和保留模式

图 15-23 是第 2 章中的一个 Windows 体系架构的简化图，WPF 和其他图形框架均处在中间层。开发人员一般关注的是上面两层，即用户应用程序和系统图形框架。绘制图形是用户应用程序和系统图形框架合力完成的。

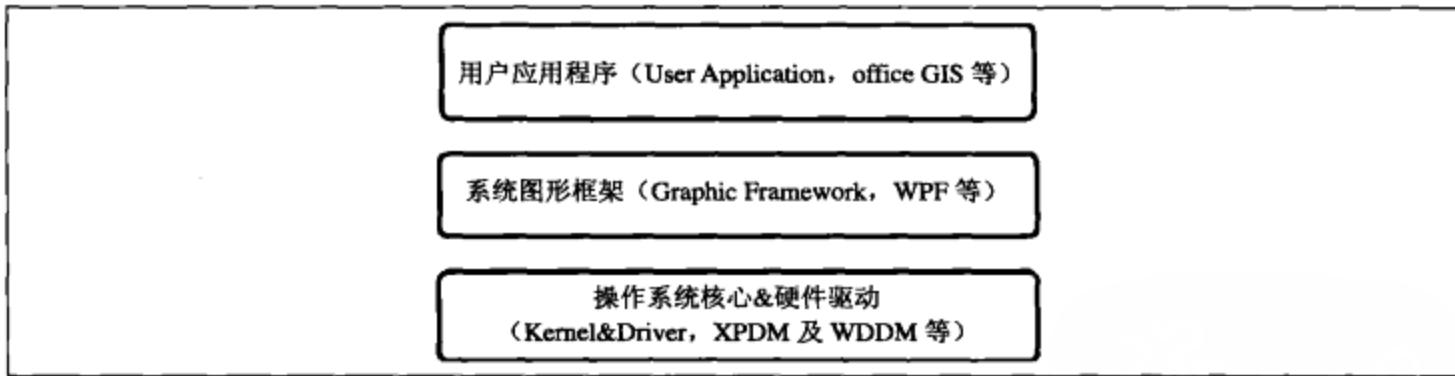


图 15-23 简化的 Windows 体系架构

绘制的方式有两种。为了方便叙述，我们现在将用户应用程序简称为小 A（Application），将系统图形框架简称为小 G（Graphic）。第一种方式我们称之为“说一件，做一件型”。

小 A 说：“画个矩形。”于是把矩形的大小，颜色等信息告诉了小 G。

小 G 说：“喔”，然后他就很快的将矩形画在屏幕上。

小 A 又说：“画个椭圆。”把椭圆的大小，颜色等信息也告诉了小 G。

小 G 又说：“喔”，然后他就又很快地将椭圆画在屏幕上了。

小 A 又说：“以后你就自己画哈。”

小 G 说：“不行，你每次都要告诉我怎么画。”

于是小 A 只能年复一年，日复一日地说：“画个矩形，画个椭圆，再画个矩形，再画个椭圆……”小 G 虽然需要小 A 每次重复，但小 A 只要一说，他就立马去做。

第二种方式是“一次搞定型”。

小 A 说：“画个矩形，矩形的大小是……颜色是……，然后画个椭圆，椭圆的大小是……颜色是……。”

……

小 G 说：“喔，画好了。”

小 A 说：“那我不管了啊。”

小 G 说：“喔。”

于是小 A 就开始了幸福的生活……

“说一件，做一件型”即立即模式绘图，图形系统框架总是能够按照用户应用程序的命令将图形绘制在屏幕上，但是他从不保留图形的信息。于是用户用应用程序需要每次都告诉他如何绘制。

“一次搞定型”即是保留模式绘图，用户应用程序规定好绘图的数据，然后一次告诉图形系统框架，至于窗口大小改变，位置发生变化等，他就交给图形系统框架全然不管了。

在过去的 Win32 和 WinForm 编程中使用的绘图底层分别是 GDI 和 GDI+³，它们属于立即模式。

1. GDI/GDI+和 WPF

GDI/GDI+属于立即模式，其绘制机制是利用 WM_PAINT 消息来重绘。当窗口位置及大小变化时，操作系统就会向应用程序发出 WM_PAINT 消息。应用程序也可以通过 API 函数（UpdateWindow 或者 RedrawWindow）强制操作系统发送该消息，如图 15-24 所示。

³ GDI+实际上在 GDI 的基础上封装而成。

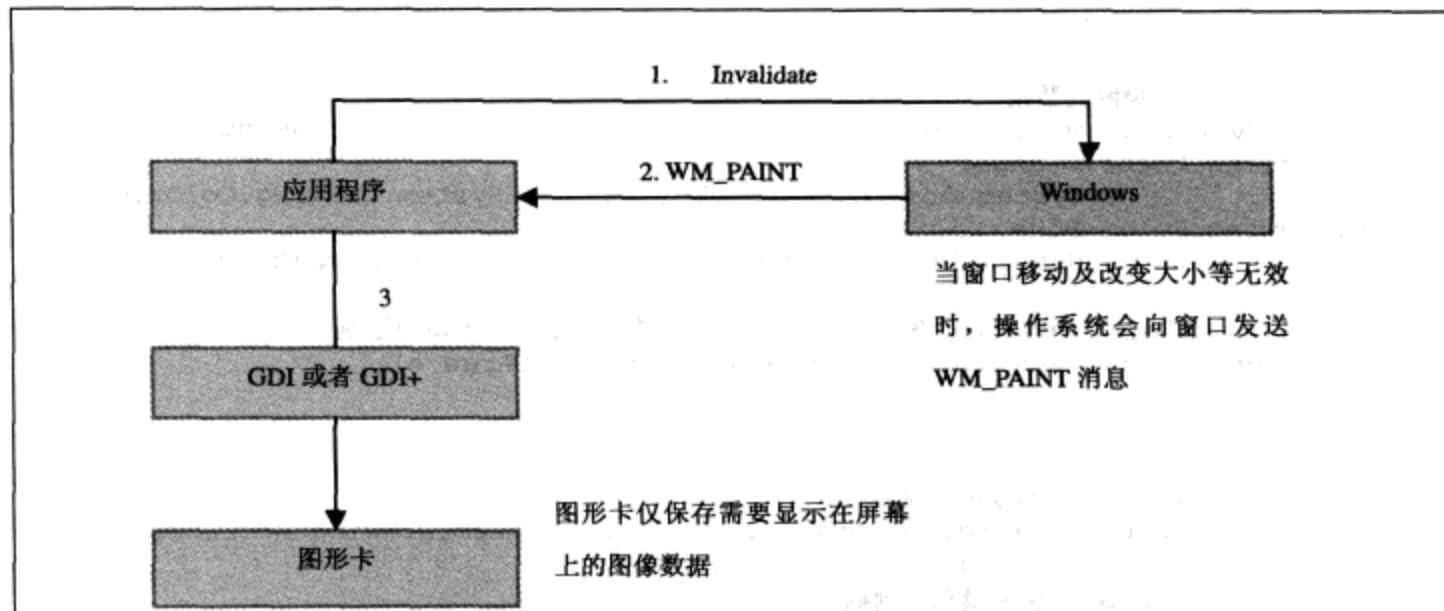


图 15-24 GDI/GDI+绘制的机制图^[4]

WPF 使用的是保留模式，其应用程序定义绘制数据，余下的工作由操作系统完成。

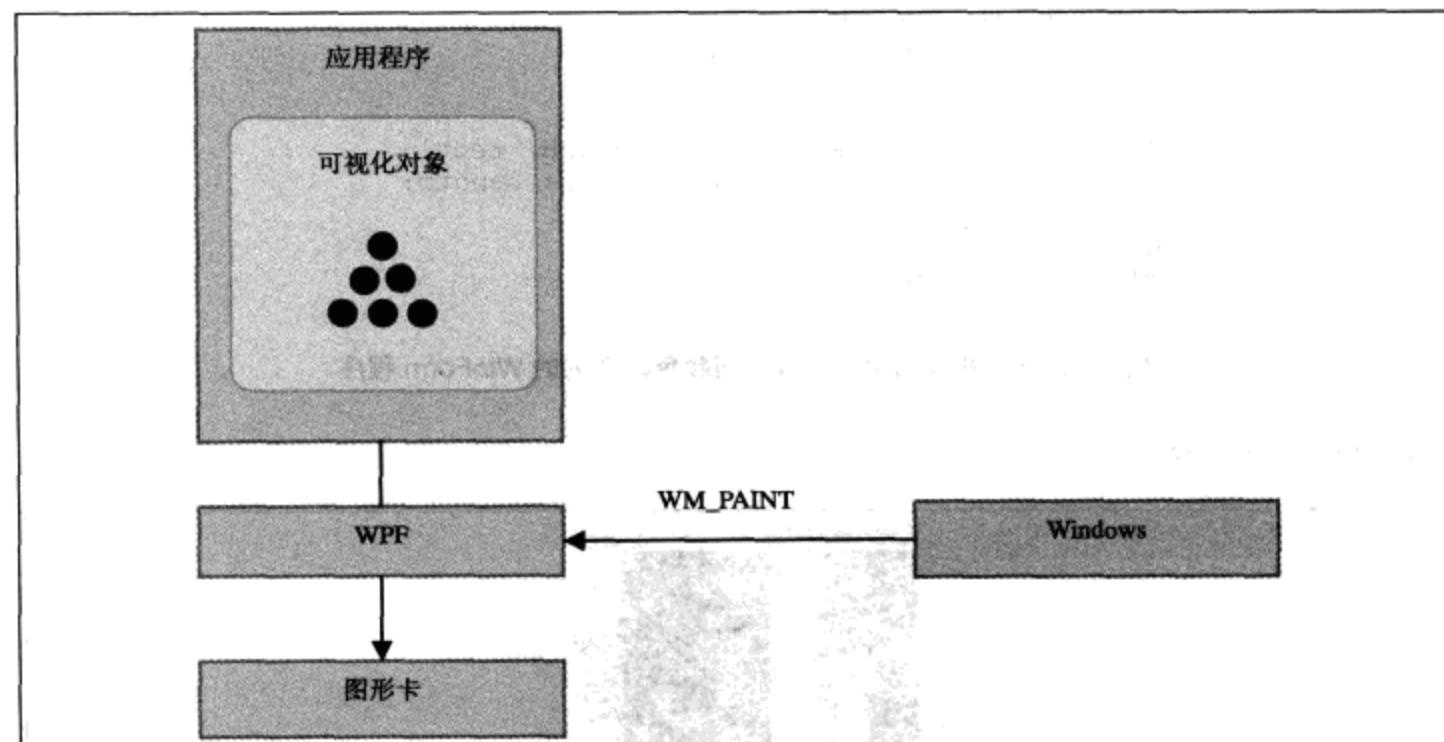


图 15-25 WPF 采用保留模式绘制数据^[4]

保留模式大大简化了开发人员的工作，并且能够很好地支持图形的半透明和动画。不妨看看下面一个例子来深入理解一下保留模式和立即模式之间的区别。

代码 15-16 是一个 WinForm 程序示例，它将窗口平均分为 7 列。每一列一种颜色，从而构成了一道彩虹（详见 `mumu_TryRainbowAttempt` 工程）。

```
public partial class Form1 : Form
{
    public Form1()
    {
```

```

        InitializeComponent();
    }
    // 一个颜色的数组
    System.Drawing.Color[] colors = { System.Drawing.Color.Red,
System.Drawing.Color.Orange,
                                    System.Drawing.Color.Yellow,     System.Drawing.Color.Green,
System.Drawing.Color.Blue,
                                    System.Drawing.Color.Indigo, System.Drawing.Color.Violet };

    // OnPaint 函数负责处理 WM_PAINT 消息，在 WinForm 窗口中绘制
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    // rect 为所绘制的区域
    System.Drawing.Rectangle rect = new System.Drawing.Rectangle(0, 0,
this.Size.Width / colors.Length, this.Size.Height);
    // myBrush 为填充的画刷
    System.Drawing.SolidBrush myBrush = new System.Drawing.SolidBrush
(System.Drawing.Color.Red);
    // GDI+绘制的图形句柄，类似于 GDI 中的 HDC
    System.Drawing.Graphics formGraphics;
    formGraphics = this.CreateGraphics();
    // 平均分成 7 列，循环绘制
    foreach (Color color in colors)
    {
        myBrush.Color = color;
        formGraphics.FillRectangle(myBrush, rect);
        rect.X += this.Size.Width / colors.Length;
    }
    myBrush.Dispose();
    formGraphics.Dispose();
}

```

代码 15-16 把窗口平均分成不同颜色的 7 列的 WinForm 程序

程序的运行结果如图 15-26 所示。

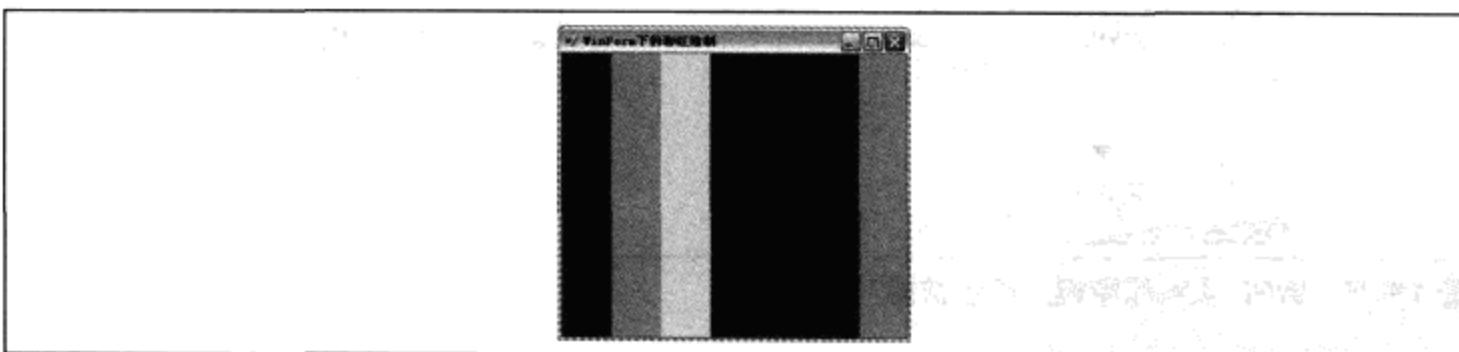


图 15-26 程序运行结果

首先自定义一个 RainbowFrameworkElement 类，派生自 FrameworkElement。并且在这个类中覆盖一个 OnRender 的函数，它类似 Form 的 OnPaint，如代码 15-17 所示（详见 mumu_wpfrainbow 工程）。

```

public class RainbowFrameworkElement : FrameworkElement
{
    // 颜色数组 在 WinForm 中使用 System.Drawing.Color
    // 在 WPF 中使用的是 System.Windows.Media.Color
    Color[] colors = { Colors.Red, Colors.Orange,

```

```

        Colors.Yellow, Colors.Green, Colors.Blue,
        Colors.Indigo, Colors.Violet );
protected override void OnRender(DrawingContext dc)
{
    // myBrush 为填充的画刷 在 WinForm 中使用的是 System.Drawing.SolidBrush
    // 在 WPF 中使用的是 System.Windows.Media.SolidColorBrush
    SolidColorBrush myBrush = new SolidColorBrush();
    // rect 为所绘制的区域 在 WinForm 中使用的是 System.Drawing.Rectangle
    // 在 WPF 中使用的是 System.Windows.Rect
    Rect rect = new Rect(0, 0, RenderSize.Width / colors.Length,
        RenderSize.Height);
    // 平均分成 7 列循环绘制
    foreach (Color color in colors)
    {
        myBrush.Color = color;
        dc.DrawRectangle(myBrush, null, rect);
        rect.X += RenderSize.Width / colors.Length;
    }
}
}

```

代码 15-17 用 WPF 实现 WinForm 程序的效果

在 MainWindow.xaml 文件中添加元素 RainbowFrameworkElement，如代码 15-18 所示。

```

<Window x:Class="mumu_wpfrainbow.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local ="clr-namespace:mumu_wpfrainbow"
    Title="WPF 下的彩虹绘制" Height="300" Width="300">
    <local:RainbowFrameworkElement>
    </local:RainbowFrameworkElement>
</Window>

```

代码 15-18 在 MainWindow.xaml 文件中添加元素 RainbowFrameworkElement

运行程序，彩虹消失，留下的只是 7 道紫罗兰色，如图 15-27 所示。

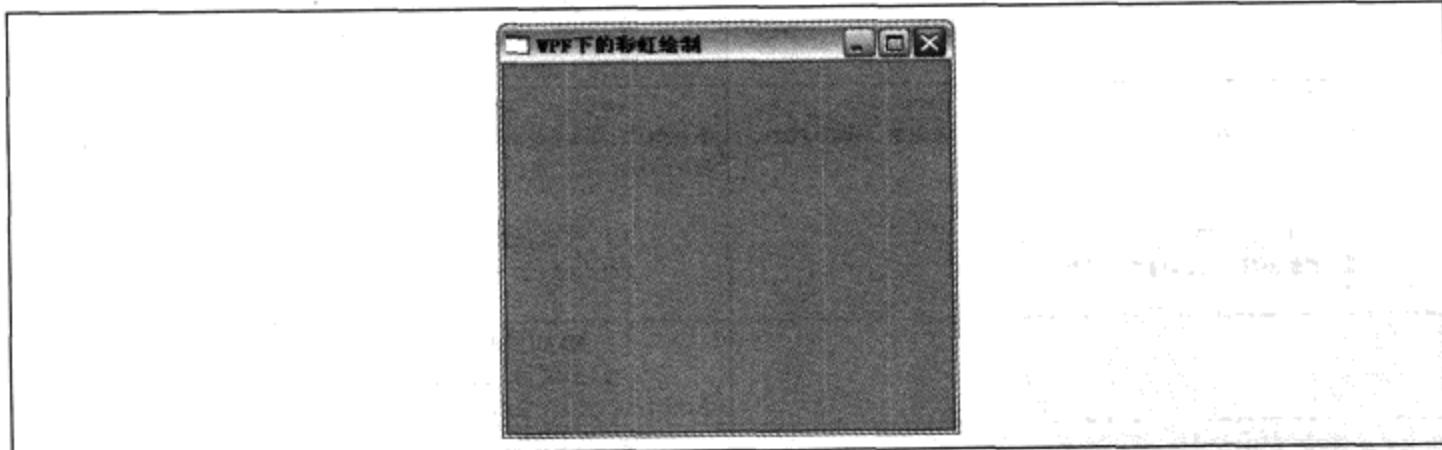


图 15-27 WPF 程序运行结果

2. 体味两种模式

下面是 WinForm 和 WPF 部分代码的一个比较，如代码 15-19 所示。

WinForm	WPF
<pre> protected override void OnPaint(PaintEventArgs e) { ... // myBrush 为填充的画刷 SolidBrush myBrush = new SolidBrush (Color.Red); Rectangle rect = new Rectangle(0, 0, this.Size.Width / colors.Length, this.Size.Height); // GDI+绘制的图形句柄，类似 GDI 中的 HDC Graphics formGraphics; formGraphics = this.CreateGraphics(); // 平均分成 7 列循环绘制 foreach (Color color in colors) { myBrush.Color = color; formGraphics.FillRectangle(myBrush, rect); rect.X += Size.Width / colors.Length; } ... } </pre>	<pre> protected override void OnRender(..) { ... SolidColorBrush myBrush = new SolidColorBrush(); Rect rect = new Rect(0, 0, RenderSize.Width / colors.Length, RenderSize.Height); // 平均分成 7 列循环绘制 foreach (Color color in colors) { myBrush.Color = color; dc.DrawRectangle(myBrush, null, rect); rect.X += RenderSize.Width / colors.Length; } ... } </pre>

代码 15-19 简化的 WinForm 和 WPF 代码

二者的程序逻辑是完全相同的，WPF 的程序只有做以下的修改，才能达到 WinForm 程序的效果。下面是修改之前和修改之后的代码对比。

WPF（修改之前）	WPF（修改后）
<pre> protected override void OnRender(..) { ... SolidColorBrush myBrush = new SolidColorBrush(); Rect rect = new Rect(0, 0, RenderSize.Width / colors.Length, RenderSize.Height); // 平均分成 7 列循环绘制 foreach (Color color in colors) { myBrush.Color = color; dc.DrawRectangle(myBrush, null, rect); rect.X += RenderSize.Width / colors.Length; } ... } </pre>	<pre> protected override void OnRender(..) { ... // 此处隐掉 // SolidColorBrush myBrush = new SolidColorBrush(); Rect rect = new Rect(0, 0, RenderSize.Width / colors.Length, RenderSize.Height); // 平均分成 7 列循环绘制 foreach (Color color in colors) { // 添加到此处 SolidColorBrush myBrush = new SolidColorBrush(); myBrush.Color = color; dc.DrawRectangle(myBrush, null, rect); rect.X += RenderSize.Width / colors.Length; } ... } </pre>

代码 15-20 修改之前与修改后的代码对比

前面的代码用一个画刷，每次循环时修改其颜色属性；后面的代码一次循环就新建一次画刷。同样的逻辑在 WinForm 中可以，而在 WPF 中则不可。这正是因为两者绘制模式的不同造成的。

在立即模式下，用户应用程序每次为系统图形框架传递一次画刷和矩形位置信息。然后图形框架将其绘制在屏幕上，随后丢弃这些信息。尽管绘制信息被丢弃了，但是图形已经被绘制在屏幕上。如图 15-28 所示。

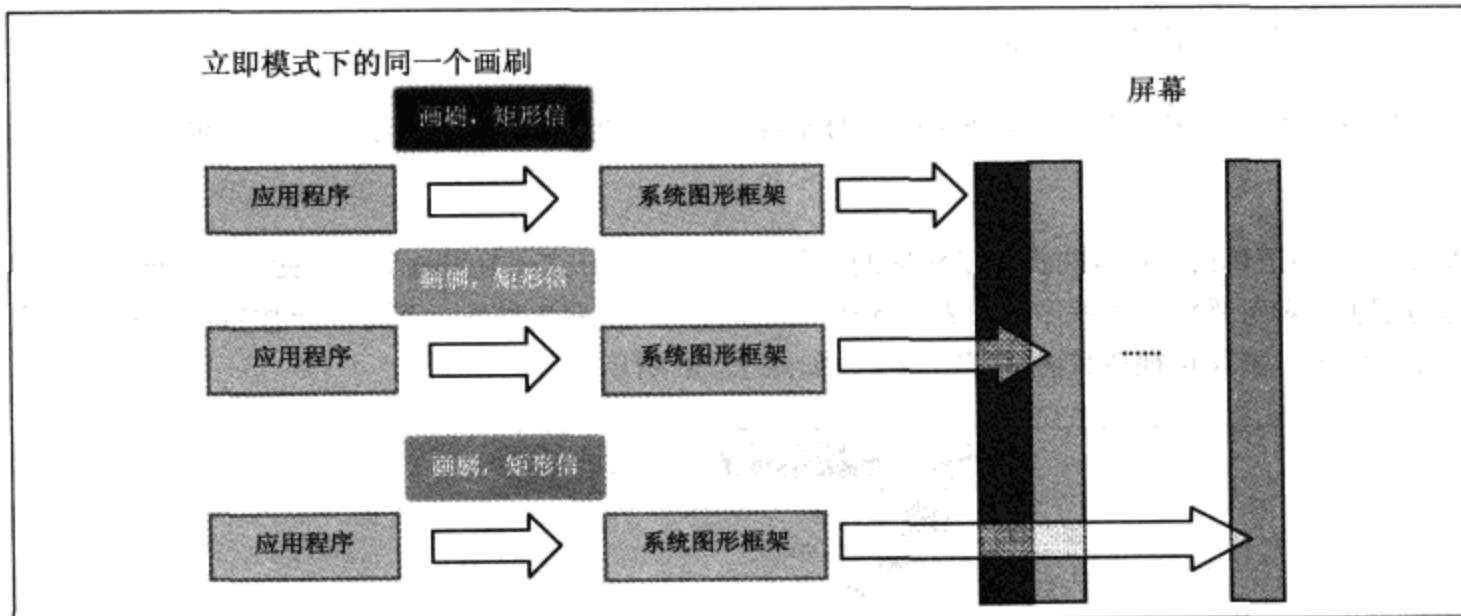


图 15-28 立即模式下的系统图形框架

在保留模式下，系统图形框架保留该画刷的实例。应用程序每次为系统图形框架传递的绘制命令，保存在图像卡的帧缓存中。直到所有绘制命令传达完毕，系统图形框架才会一次性绘制在屏幕上，但是这时画刷的颜色属性已经变为紫罗兰色。因此 WPF 窗口无法绘制彩虹，而是 7 道紫罗兰色，如图 15-29 所示。

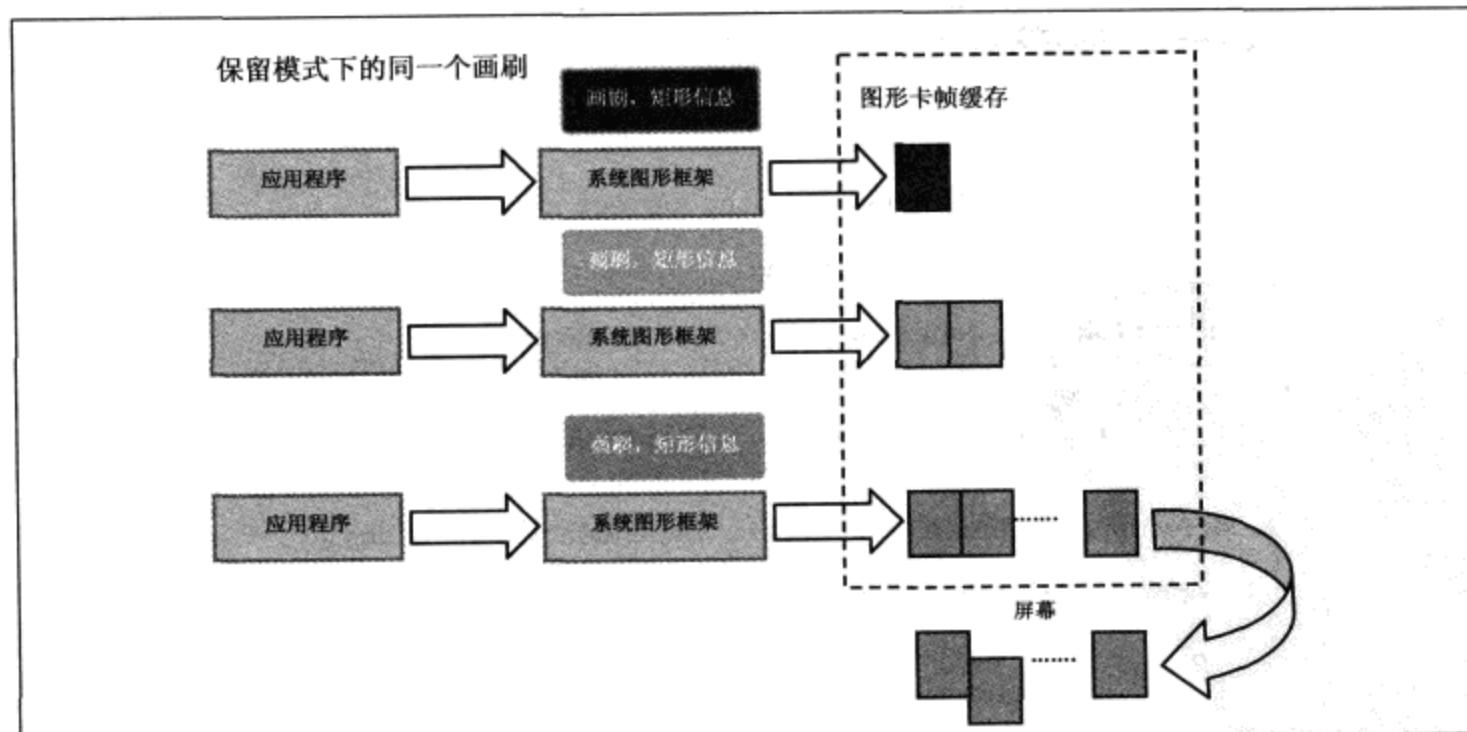


图 15-29 保留模式下的系统图形框架

15.3.2 WPF 二维图形体系结构

WPF 的二维图形体系结构首先从逻辑树 (logic tree) 和可视树 (visual tree) 开始。

1. 逻辑树和可视树

XAML 文件的结构就是一个树型结构，从一个根节点层层展开。这棵树称为“逻辑树”，树上的每一个节点都是一个完整而独立的元素。

如果用 XAMLPad 来查看，很多逻辑树上的元素又进一步细分构成另外一棵树，这棵树称为“可视树”。

WPF 为遍历逻辑树和可视树提供了相应的类和方法，即 LogicTreeHelper 和 VisualTreeHelper。一个遍历逻辑树和可视树的小程序运行界面描述“琅琊福地”的层次结构，下方 3 个按钮的功能分别是打印逻辑树、打印可视树和清空打印结果，如图 15-30 所示。



图 15-30 “琅琊福地”的层次结构

程序代码 15-21 如下：

```
void PrintLogicalTree(int depth, object obj)
{
    Console.WriteLine(new string(' ', depth) + obj.GetType().Name);

    if (!(obj is DependencyObject)) return;
    foreach (object child in LogicalTreeHelper.GetChildren(
        obj as DependencyObject))
        PrintLogicalTree(depth + 1, child);
}

void PrintVisualTree(int depth, DependencyObject obj)
{
    Console.WriteLine(new string(' ', depth) + obj.GetType().Name);

    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
        PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
}
```

代码 15-21 遍历逻辑树和可视树的代码

将项目的输出由 Windows Application 改为 Console Application（参见第 8 章）。这样程序运行时，除了弹出主窗口外，还会弹出控制台窗口，逻辑树和可视树的层次结构显示在该窗口中，如图 15-31 所示。



```
file:///F:/我的工作/WPF/写作/第四部分 WPF图形/4.2 WPF的图形总览/src/main... -> X
逻辑树:
Window1
  Grid
    Border
      ContentPresenter
        Grid
          TreeView
            Border
              ScrollViewer
                Grid
                  Rectangle
                  ScrollContentPresenter
                  ScrollBar
                  ScrollBar
                    TextBlock
          Grid
            Button
              ButtonChrome
                ContentPresenter
                  TextBlock
            Button
              ButtonChrome
                ContentPresenter
                  TextBlock
            Button
              ButtonChrome
                ContentPresenter
                  TextBlock
                    TextBlock
                    TextBlock
                    TextBlock
```

图 15-31 函数在控制台中输出逻辑树和可视树的层次结构

我们可以将逻辑树和可视树整理成下图，其中橘红色表示既是逻辑树节点又是可视树节点。

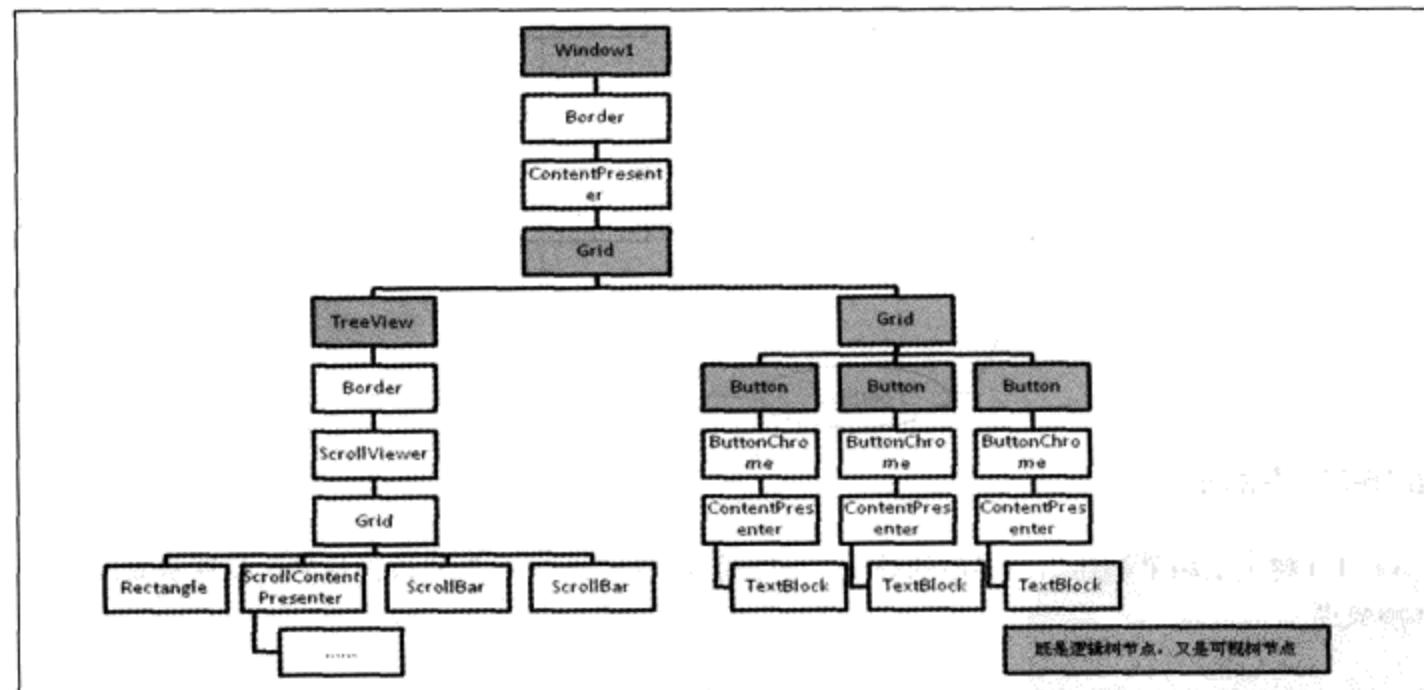


图 15-32 整理后的逻辑树和可视树

2. 逻辑树和可视树的区别

逻辑树和可视树的区别如下。

- (1) 逻辑树的层次结构和 XAML 文件的组织结构类似，而可视树则与其他节点必须是继承自 Visual

和 Visual3D。

(2) 逻辑树主要是用在如下方面。

- 资源查找：如一个按钮的背景引用了某一个画刷资源，则会按照逻辑树的节点层层向上查找，直至逻辑树的根节点。
- 属性值继承（参见第 5 章）：以一个例子说明，如某一个窗口设置了字体大小，则将按照逻辑树的层次结构依次影响到下面的节点。如果某一个节点，如 Grid 没有字体大小属性，那么会穿越该节点继续影响下面包含该属性的逻辑树节点，如代码 15-22 所示（详见 `mumu_propertyinheritdemo` 工程）。

```
<Window x:Class="mumu_propertyinheritdemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300" FontSize="15">
<Grid>
    <Button>
        属性值继承
    </Button>
</Grid>
</Window>
```

代码 15-22 属性值继承示例

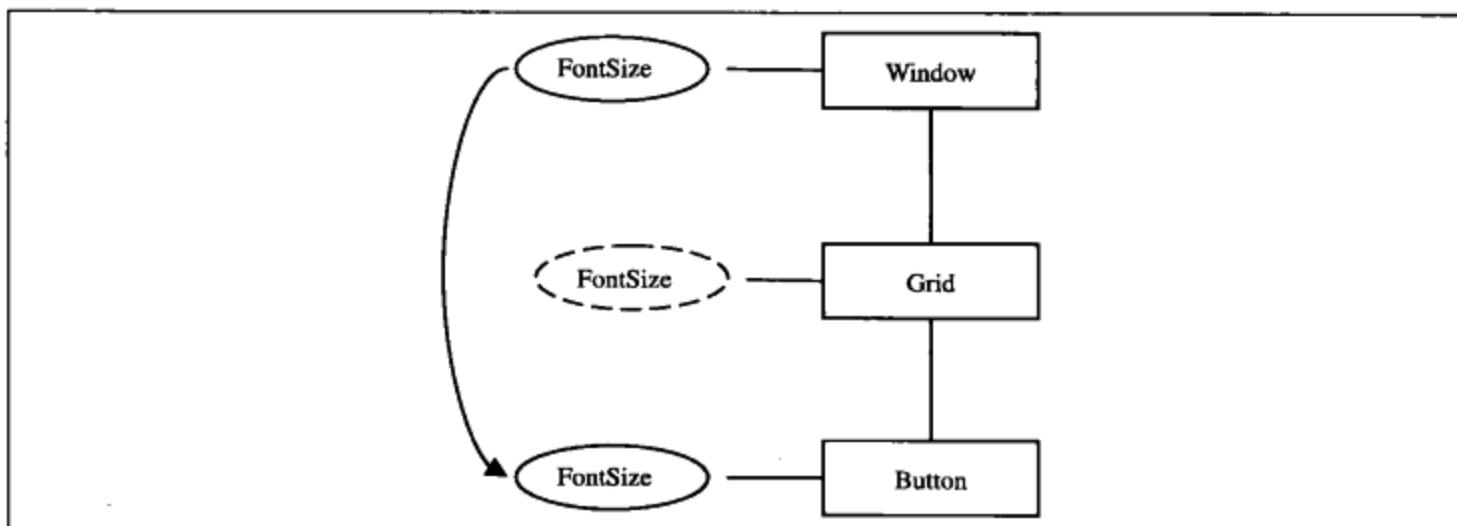


图 15-33 属性值继承

(3) 可视树主要用来描述用户界面的外观，可以通过自定义控件模板修改控件的可视树从而改变控件的外观。

注意 ContentElement 和 FrameContentElement 虽然可以作为逻辑树的节点，但是由于其本身并不继承 Visual 或者 Visual3D，因此不能作为可视树的节点。

15.3.3 WPF 二维图形的重要元素

WPF 为图形绘制提供了很多类，而且类的名字非常容易混淆，例如 DrawingImage 和 ImageDrawing。但是理解了 WPF 中 4 个重要的图形元素类，Visual、Drawing、Geometry 和 Shape 之后，整个 WPF

的绘图相关类会条理清晰很多。

1. Visual

可视树是 WPF 渲染的基础，暂且抛开三维要素（Visual3D），整个图形用户界面都是由 Visual。即 Visual 的派生类构成的可视化树，WPF 从根节点开始渲染这棵可视树。我们可以将 Visual 想像成 Win32 当中的窗口句柄，它的主要作用就是为 WPF 绘制提供支持。

2. Drawing

Visual 可以想像成一个窗口，那么窗口里绘制的内容该如何描述呢？答案是用 Drawing 来进行描述，Drawing 针对矢量、图像、文字甚至是视频派生出下面几种不同的 Drawing。

- (1) GeometryDrawing：绘制几何图形。
- (2) ImageDrawing：绘制图像。
- (3) GlyphRunDrawing：绘制文本。
- (4) VideoDrawing：播放音频和视频文件。
- (5) DrawingGroup：Drawing 的集合。

VisualTreeHelper 中的静态方法 GetDrawing 用来从 Visual 中获取 Drawing 的集合，其输入参数是一个 Visual，返回值则是一个 DrawingGroup，如代码 15-23 所示。

```
public static DrawingGroup GetDrawing( Visual reference)
```

代码 15-23 静态方法 GetDrawing

3. Geometry

绘制图像时 ImageDrawing 需要图像文件，然后将其绘制在指定的矩形中；播放多媒体时 VideoDrawing 需要多媒体文件，然后将其摆放在合适的位置；绘制几何图形时 GeometryDrawing 需要几何数据，然后设置画刷和画笔来绘制。这个几何数据使用 Geometry 及其派生类来描述，Geometry 只是用来描述二维图形的几何数据，而并不负责显示，其派生类中包括椭圆、矩形及直线等几何图形。

4. Shape

Shape 是一种高级的图形元素，由于它派生自 FrameworkElement，因此可以任意地嵌套在面板或者其他控件中，同时支持 WPF 的事件和布局。

5. 4 个重要图形元素之间的关系

4 个重要的图形元素之间联系如下。

(1) WPF 用户界面可视化树的每一个节点都派生自 Visual，如图 15-34 所示。

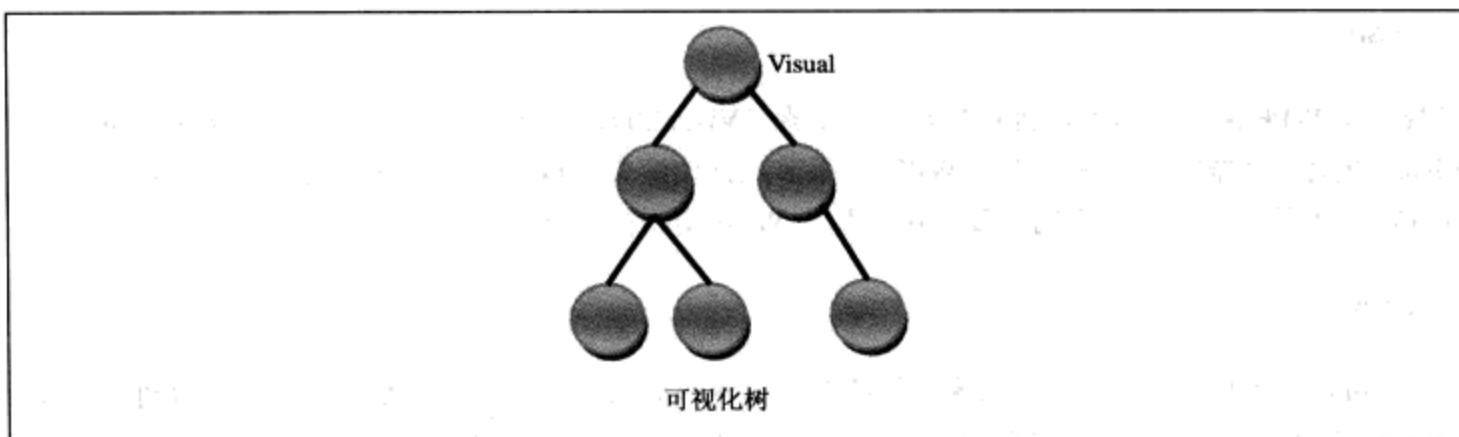


图 15-34 可视化树的每一个节点都派生自 Visual

(2) Shape 和 Visual 是继承关系，如图 15-35 所示。

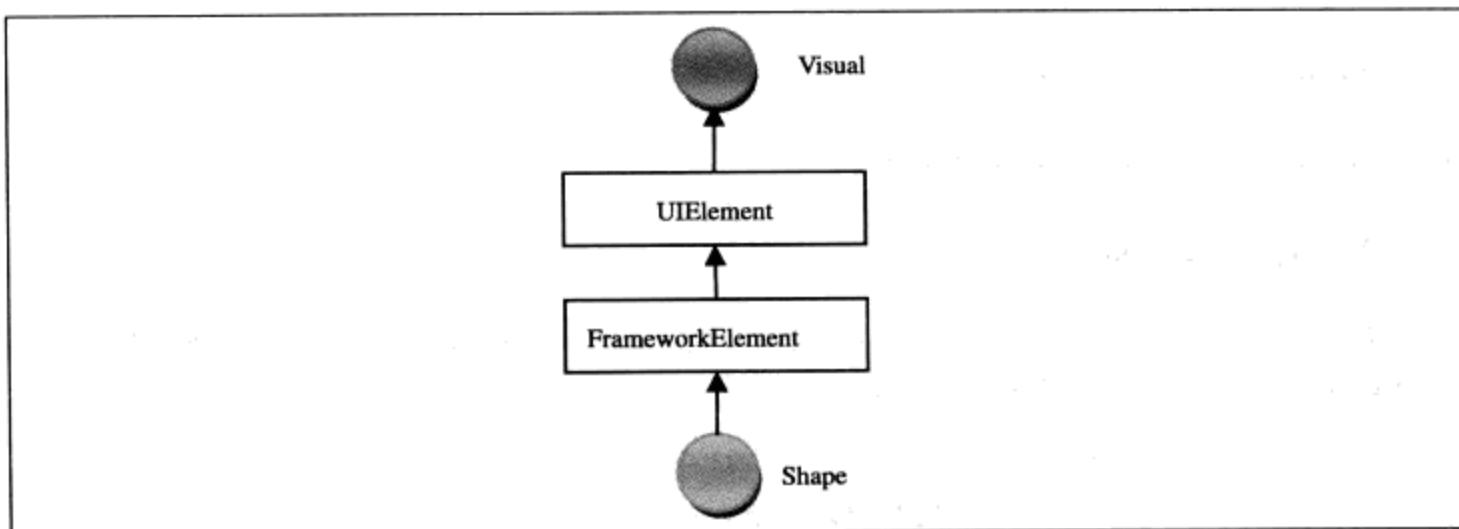


图 15-35 Shape 和 Visual 的关系

(3) Visual 的绘制内容可以通过 Drawing 来描述，通过 VisualTreeHelper 的静态方法 GetDrawing 可以从一个 Visual 中获得一个 DrawingGroup。从而可以遍历 Visual 中的所有 Drawing 对象，如图 15-36 所示。

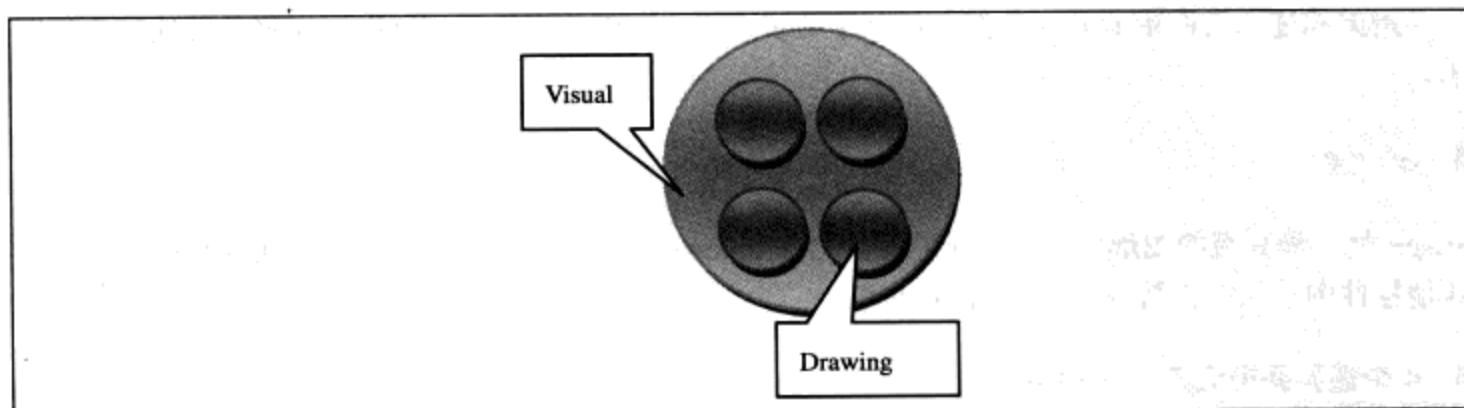


图 15-36 Visual 和 Drawing 的关系

(4) Drawing 的一个派生类 GeometryDrawing 需要用 Geometry 来描述其几何数据, 如图 15-37 所示。

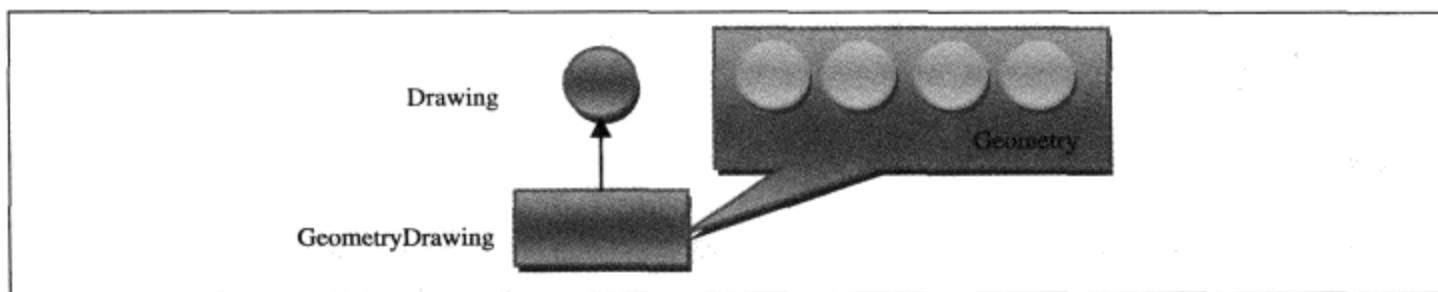


图 15-37 Drawing 的派生类 GeometryDrawing 需要用 Geometry 来描述其几何数据

下面的 XAML 代码 15-24 说明 GeometryDrawing 和 Geometry 之间的关系:

```
<GeometryDrawing>
<GeometryDrawing.Geometry>
    <GeometryGroup>
        <EllipseGeometry Center="50,50" RadiusX="45" RadiusY="20" />
        <EllipseGeometry Center="50,50" RadiusX="20" RadiusY="45" />
    </GeometryGroup>
</GeometryDrawing.Geometry>
</GeometryDrawing>
```

代码 15-24 GeometryDrawing 和 Geometry 之间的关系

(5) Shape 派生自 Visual, 因此具有 Visual 和 Drawing 之间的关系。并且 Shape 具有 Fill 属性, 可以通过 DrawingBrush 和 Drawing 关联。

(6) Path 通过一个 Data 属性与 Geometry 关联。

这四个图形元素是 WPF 绘图类最为重要的元素, 它们之间的联系千丝万缕, 绝不止于上面几条。此外在二维图形中还有一类极其重要的要素, 即画刷, 它是可以填充图形的表面的一种类型。几乎所有可视化的图形元素均有画刷类型的属性, 如前景色及背景色等, 它们通过该属性可以与 Visual (通过 VisualBrush) 或者 Drawing (DrawingBrush) 关联。

15.3.4 书架上到底放什么书

从本节开篇, 木木经蓉儿指点, WPF 的图形架构即是书架。那么这个书架上到底应该放什么样的书呢? 木木想了想, 觉得应该在这个书架上至少放这么四本书:

- (1) 颜色和画刷 (Brush)。
- (2) Shape。
- (3) Geometry。
- (4) Drawing 和 Visual。

下一节将是第一本书——颜色和画刷。

15.4 颜色和画刷（第一本书）

15.4.1 颜色

1. 概述

我们能够感受到五彩缤纷的世界，是因为我们的眼睛能够感知从红到紫的一系列颜色。颜色的产生是人眼对光的反应，光是一种人眼可以看见的电磁波，一般人的眼睛能够感受到波长为400nm~700nm之间的电磁波。当然也有少数人能够感受到更大范围的电磁波，但是波长范围也只是380nm到780nm之间。

这个范围虽然不大，但是已经有千千万万种不同的颜色了。我们该如何记录和重现这种颜色呢？幸运的是人们发现只要用3种基本的原色（红、绿和蓝）按照不同比例混合即可模拟出千变万化的各种颜色。图15-38所示为国际照明委员会（CIE）色度图，从中可以容易地看出3种原色和人眼所能感知的所有颜色之间的关系。

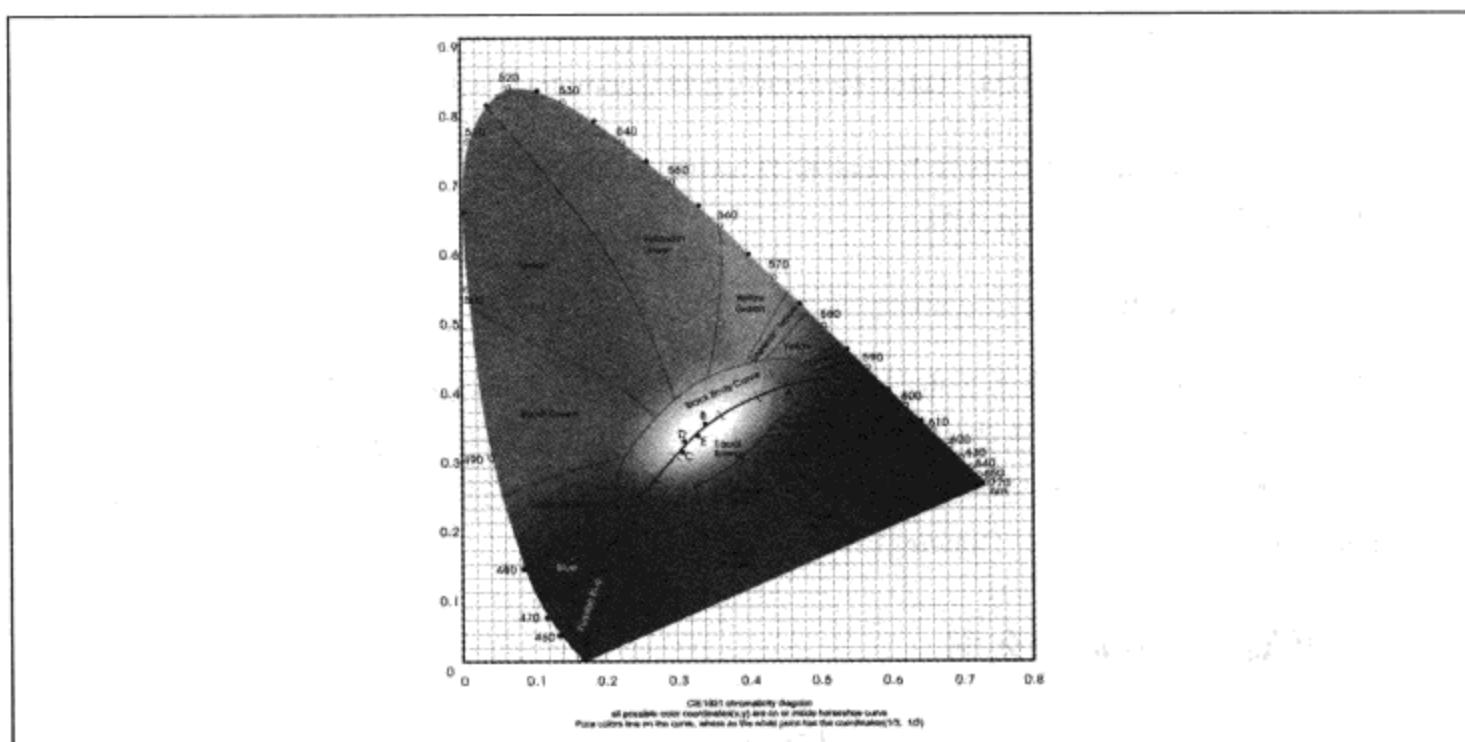


图15-38 国际照明委员会（CIE）色度图

图中的一个舌形区域代表人眼能感知的所有颜色，区域的轮廓线上所标注的数字代表光的波长。横轴(x)表示红色的分量；纵轴(y)表示绿色的分量；而蓝色的分量可以通过公式 $1-x-y$ 得到，如在(0.2,0.3)处的蓝色分量就为0.5。

1996年，微软和惠普公司推出一个标准红绿蓝的颜色空间标准(sRGB)，它得到了广泛的支持。sRGB也是用红、绿和蓝3种原色来表示，一般每种颜色分量的值范围为0~255。红色以红色分量为255，其他两个分量为0来表示(255,0,0)；蓝色和绿色也是同理。当3个分量为0时，表示黑色；为255时，则表示白色。sRGB所表示的颜色范围（色域）远远小于人眼所能感受到的颜色范围，如图15-39

所示，在 CIE 色度图中 sRGB 的红色位于 $(0.6400, 0.3300)$ 、绿色位于 $(0.3000, 0.6000)$ 、蓝色位于 $(0.1500, 0.0600)$ ，且白色位于 $(0.3127, 0.3290)$ 。

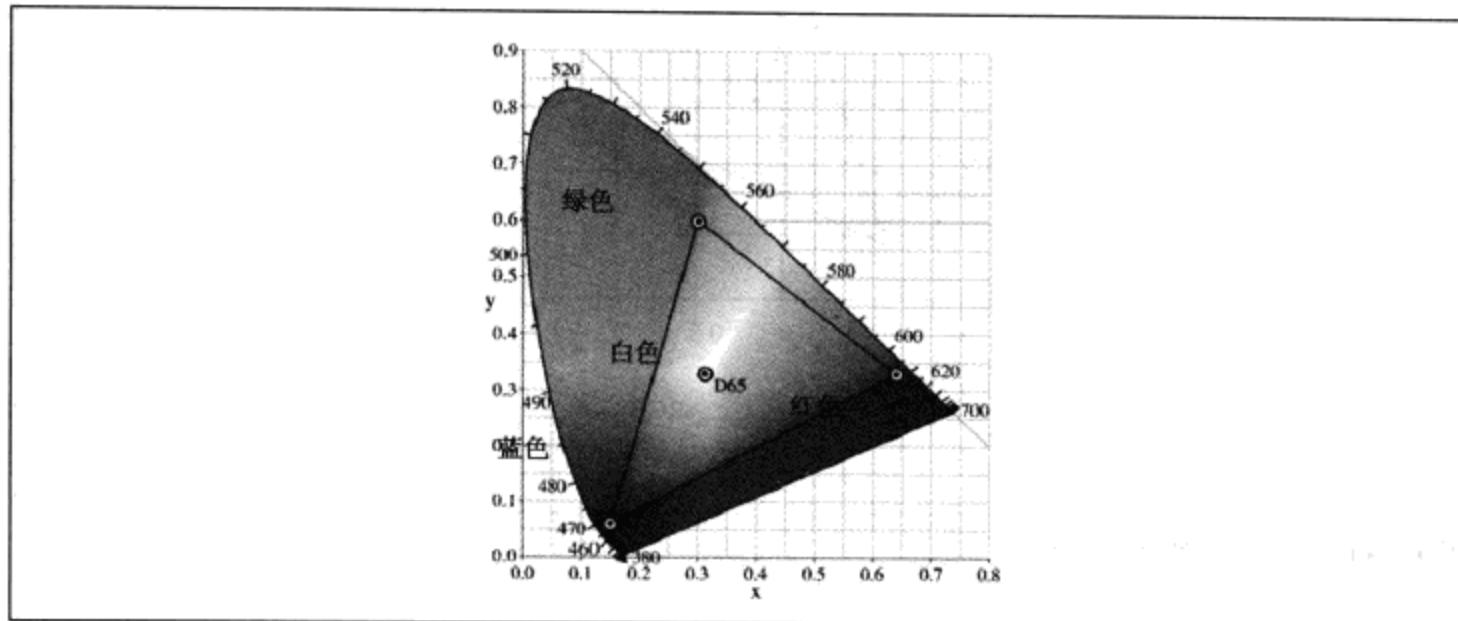


图 15-39 sRGB 色域示意图

由于 sRGB 的色域不够大，特别是蓝绿的色域，如某些特定的打印机能够打印出比 sRGB 的绿色更绿的颜色。于是在印刷行业中采用更多的是 AdobeRGB 颜色空间，如图 15-40 所示，AdobeRGB 的红色位于 $(0.6400, 0.3300)$ 、绿色位于 $(0.2100, 0.7100)$ 、蓝色位于 $(0.1500, 0.0600)$ ，且白色位于 $(0.3127, 0.3290)$ 。尽管 AdobeRGB 的色域范围比 sRGB 色域范围大，但是还是不能覆盖人眼能感受到的所有颜色范围。

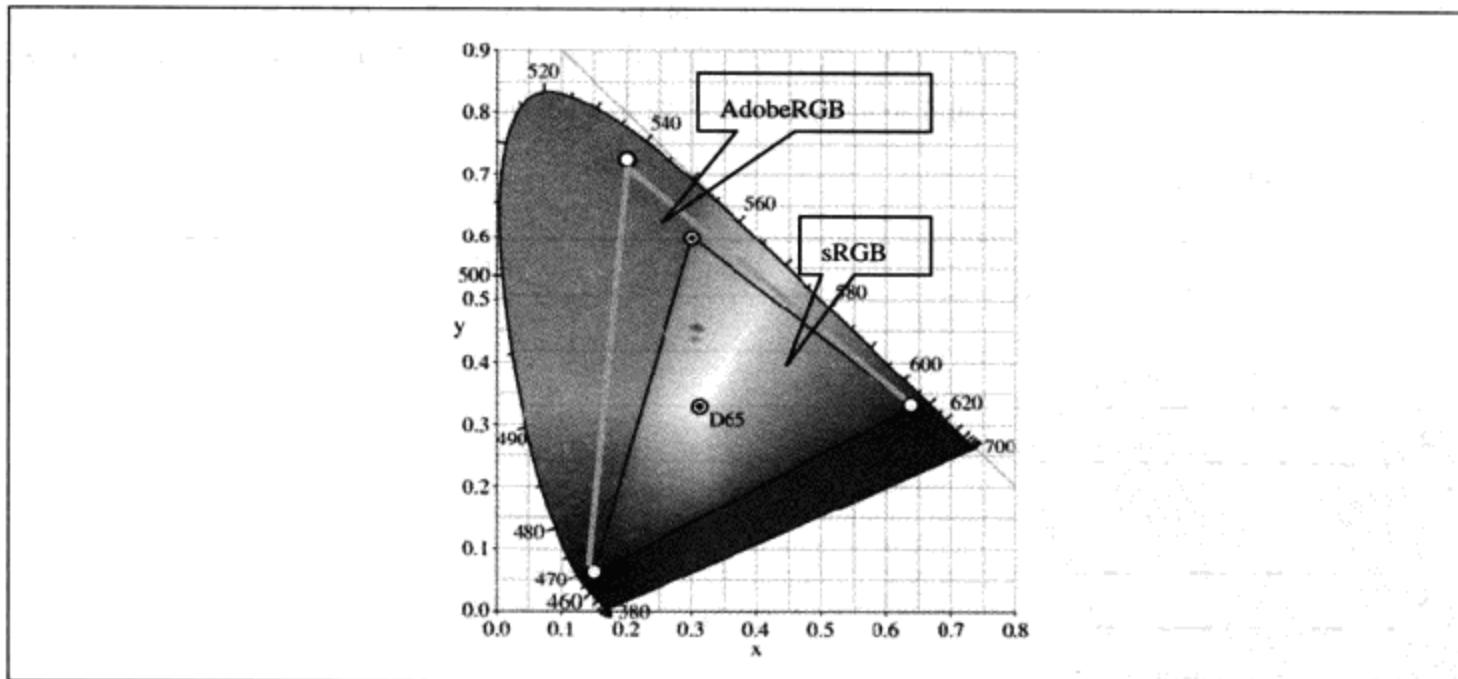


图 15-40 AdobeRGB 颜色空间

2003 年微软又推出了 scRGB 颜色空间，它大大拓展了色域范围。不仅全部覆盖人眼可见的色域范围，还把颜色空间扩展了很多，如图 15-41 所示。

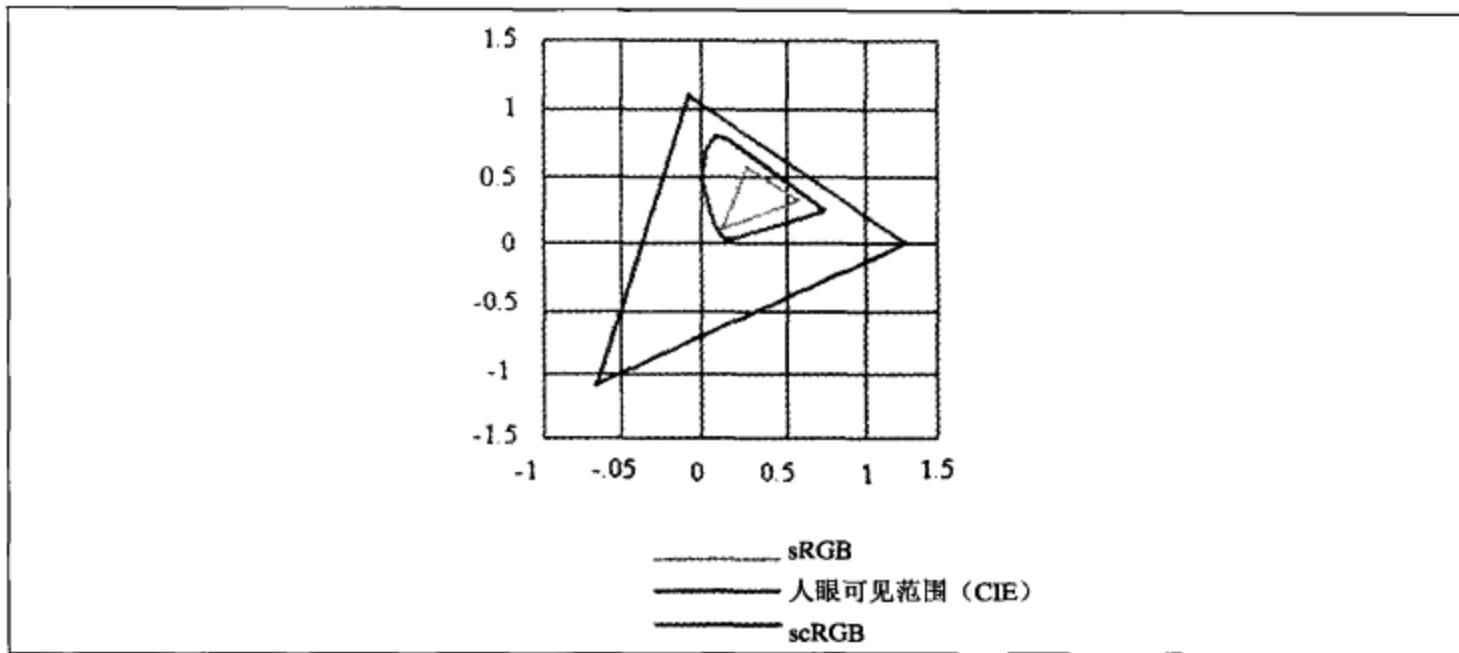


图 15-41 scRGB 颜色域空间

2. WPF 的颜色

WPF 的颜色被封装为 Color 结构体，定义在 System.Windows.Media 命名空间中。之所以强调其命名空间，是因为在 WinForm 中也有一个 Color 结构体，不过它定义在 System.Drawing 命名空间中。Color 结构支持 sRGB 颜色空间，使用了 R、G 和 B 共 3 个可读写的属性来表示 sRGB 颜色空间。3 个属性的类型均为 Byte，取值范围均为 0~255。除了三原色以外，Color 结构还包括一个透明属性 A。它也是一个 Byte 类型，0 表示完全透明；255 表示完全不透明。

Color 结构为了支持 scRGB 颜色空间，还有另外 4 个属性，即 scA、scR、scG 和 scB，与 A、R、G 及 B 相对应。不同是它们的类型是 float，表示的值范围也有所不同。如当 R 为 0 时，scR 为 0；R 为 255 时，scR 为 1。但是它们之间并不是线性关系，如表 15-5 所示。

表 15-5 R 值与 scR 值对照^[5]

scR/scG/scB	R/G/B
<= 0	0 -
0.1	89
0.2	124
0.3	149
0.4	170
0.5	188
0.6	203
0.7	218
0.8	231
0.9	243
>= 1.0	255

实际上二者有近似如下的函数关系：

$$scX \cong (\frac{X}{255})^{2.2} \quad X \text{ 表示 R、G、B}$$

只有 scA 和 A 保持的是一种线性关系：

$$scA = \frac{A}{255}$$

我们可以通过 A、R、G 及 B 或者 scA、scR、scG 和 scB 属性来指定任意颜色，如代码 15-25 所示。

```
Color color = new Color();
// 设置颜色为红色
color = Color.FromRgb(255, 0, 0);
// 设置颜色为半透明的红色(sRGB)
color = Color.FromArgb(100,255, 0, 0);
// 设置颜色为半透明的红色(ScRGB)
color = Color.FromScRgb(0.5f, 1.0f, 0.0f, 0.0f);
```

代码 15-25 通过 A、R、G 和 B 属性或者 scA、scR、scG 和 scB 来指定任意的颜色

WPF 通过 Colors 类型提供了多种系统颜色，因此也可以通过它来获得需要的系统颜色，如代码 15-26 所示。

```
Color color = new Color();
// 设置颜色为红色
color = Colors.Red;
```

代码 15-26 通过 Colors 类型来获得需要的系统颜色

WPF 还可以通过字符串的方式为颜色赋值，字符串的格式为“#aarrggbb”。其中前两位是透明度 A 的值，后面依次是 R、G 和 B 的值。并且需要用十六进制数值来表示，如代码 15-27 表示红色。

```
color = (Color)ColorConverter.ConvertFromString("#FFFF0000");
fillbrush.Color = color;
```

代码 15-27 通过字符串的方式为颜色赋值

15.4.2 画刷

虽然颜色在 WPF 中无处不在，但是 WPF 的元素几乎从不直接和颜色交互，而是通过一个画刷(Brush)对象来使用颜色。画刷是一种可以将颜色填充到目标区域的对象，WPF 元素通过多个类型为画刷的属性来填充其区域和边界线，如表 15-6 所示。

表 15-6 通过多个类型为画刷的属性填充区域和边界线的 WPF 元素

Class	Brush properties
Border	BorderBrush 及 Background
Control	Background 及 Foreground

Panel	Background
Pen	Brush
Shape	Fill 及 Stroke
TextBlock	Background

WPF 中包含如下 6 种画刷。

- (1) **SolidColorBrush**: 传统画刷，负责把某种颜色填充到目标区域。
- (2) **LinearGradientBrush**: 渐变颜色的画刷，它假定两个指定的点连成一条线段。在两点之间的颜色进行线性插值，然后使用这种渐变色来填充目标区域。
- (3) **RadialGradientBrush**: 渐变颜色的画刷，与 **LinearGradientBrush** 不同的是以一个起始点呈椭圆状向外散发。
- (4) **ImageBrush**: 一种位图画刷，负责将位图填充到目标区域。
- (5) **DrawingBrush**: 使用一种 **Drawing** 对象来填充目标区域，通过 **Drawing** 可以绘制几何图形、图像、文字甚至是视频，而且这些元素可以任意组合再形成更复杂的图形，这就意味着可以填充任意复杂的图形到目标区域中。
- (6) **VisualBrush**: 使用 **Visual** 对象来填充目标区域。所有的控件都派生自 **Visual**。那么意味着无论是一个按钮还是 **RadioButton** 都完全可以填充到目标区域当中。但是需要注意的是 **VisualBrush** 仅仅绘制了 **Visual** 的外观，填充到目标区域的按钮是不能点击的，**RadioButton** 也是不能选中的。

6 种画刷的填充效果如图 15-42 所示。

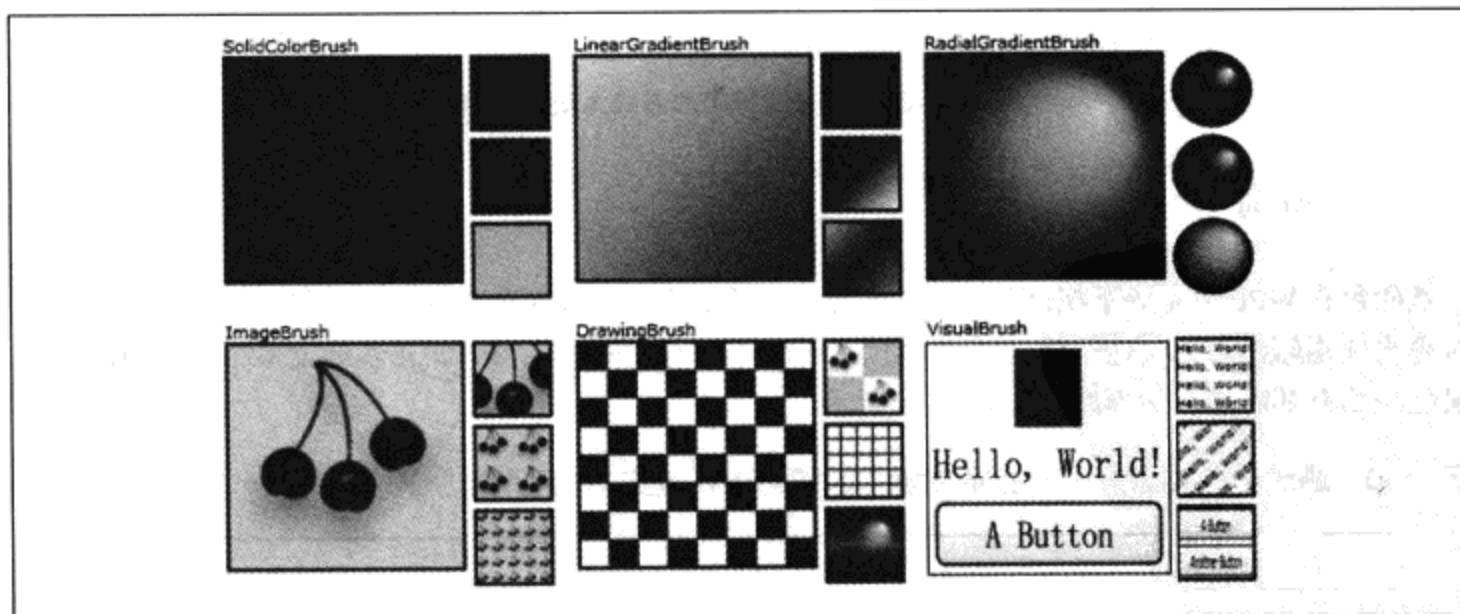


图 15-42 画刷的填充效果

这 6 种画刷均派生自 Brush，其类层次关系如图 15-43 所示。

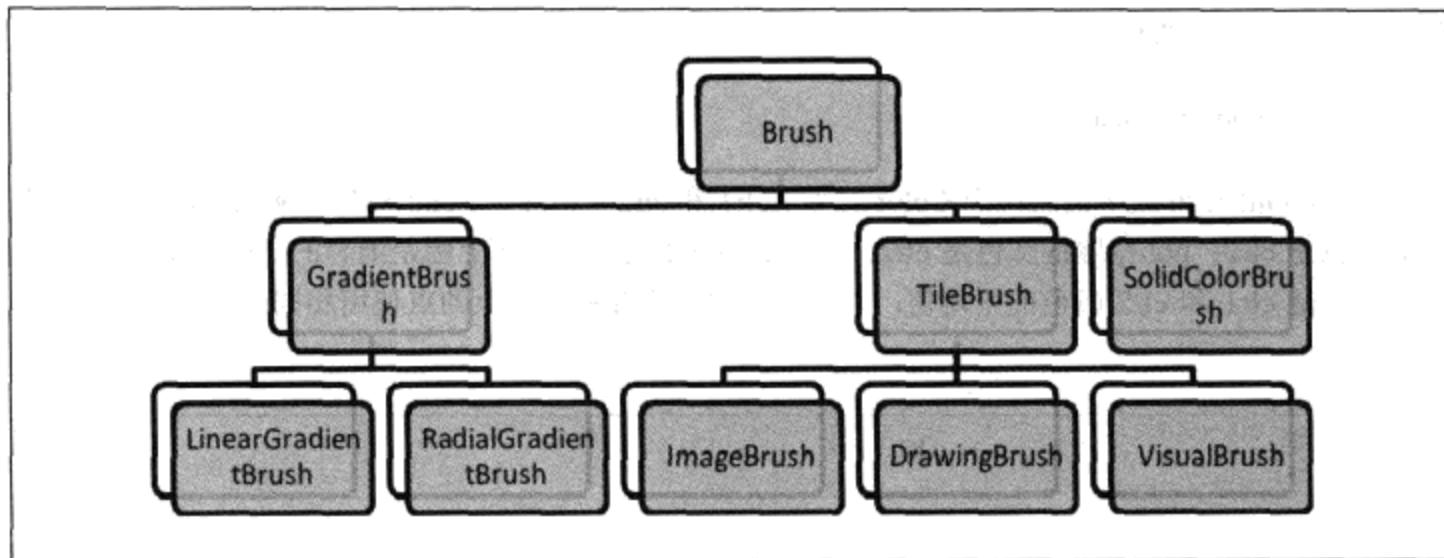


图 15-43 画刷的类层次

图中 SolidColorBrush 直接派生自 Brush，LinearGradientBrush 和 RadialGradientBrush 派生自 GradientBrush。它们均属于渐变画刷，只不过渐变方式不同。而 ImageBrush、DrawingBrush 和 VisualBrush 都派生自一个称为“TileBrush”的类型，TileBrush 可以使用重复的图案来填充目标区域。根据 TileBrush 派生的画刷不同，图案源可以是任意的 Drawing、Image 或者 Visual。

如果 WPF 直接使用颜色，那么元素的表面就仅仅只能使用固定的颜色，最多还有渐变色。但是使用画刷这种间接方式，可以填充元素表面的不仅仅是颜色，还可以是 Image、Drawing 或者 Visual。这就是画刷的威力。

1. SolidColorBrush

SolidColorBrush 的重要属性是 Color，通过设置该属性可以以不同颜色填充目标区域。实际上很多时候都在隐式地使用 SolidColorBrush，代码 15-28 用蓝色填充矩形。

```
<Rectangle Width="50" Height="50" Fill="Blue" />
```

代码 15-28 填充蓝色矩形代码

它实际上是代码 15-29 的简写。

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

代码 15-29 填充蓝色矩形的复杂代码

如果希望全面掌握 SolidColorBrush 的用法，可以参见随附光盘中的示例 `mumu_brushes`。在 `solidcolorbrush.xaml` 文件中共使用了 10 种不同的语法来设置一个矩形填充色为蓝色。

2. 漐变画刷

渐变画刷分为如下两种。

(1) LinearGradientBrush

LinearGradientBrush 最少需要两个 Color 对象 (clr1 和 clr2) 和两个 Point 对象 (pt1 和 pt2)，pt1 位置的颜色是 clr1；pt2 位置的颜色是 clr2；在 pt1 和 pt2 之间的连线上则是混合的 clr1 和 clr2 颜色。连线中心是 clr1 和 clr2 颜色的平均值；垂直于连线的位置和连线上的点使用相同颜色，如图 15-44 所示。

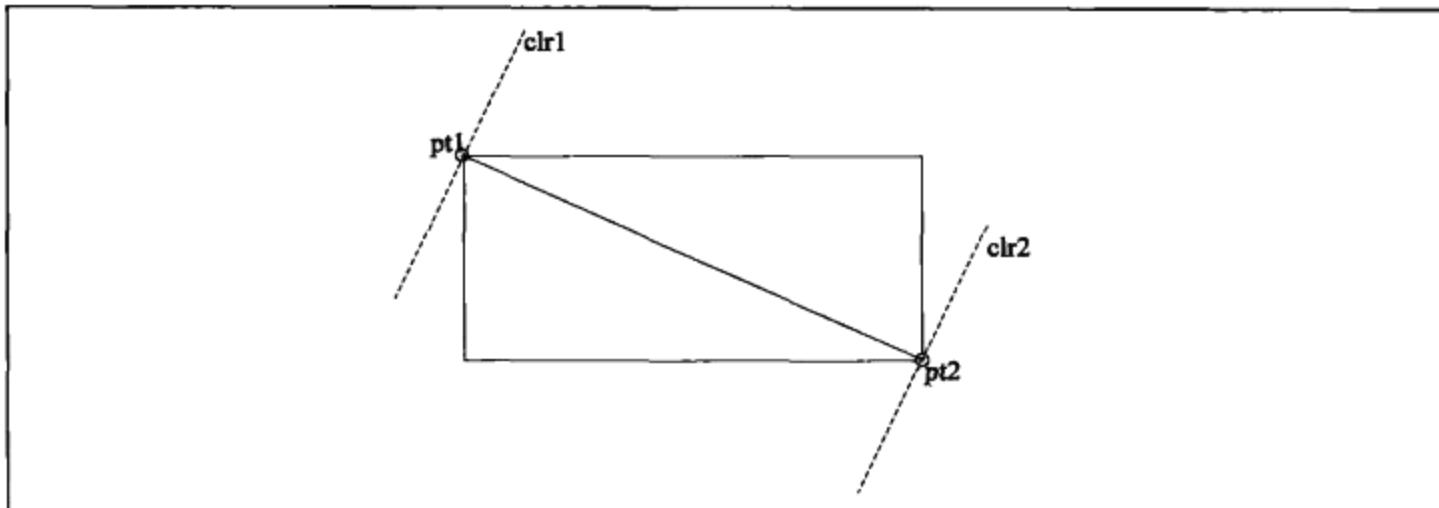


图 15-44 LinearGradientBrush

两个点的位置可以使用绝对坐标，单位为 DPI，但是最大的问题在于元素尺寸发生改变时必须重新指定这两个点。也可以使用相对坐标，这种方式规定需要填充的区域总是一个单位高，一个单位长，左上角点是 (0,0)，而右下角点是 (1,1)。图中 pt1 点的坐标即为 (0,0)，pt2 点的坐标为 (1,1)。在两点之间还可以任意插入更多不同颜色的点，图 15-45 中左上角点为黄色，右下角点颜色为 LimeGreen。中间插入了两个点，分别是红色 (Red) 和蓝色 (Blue)。中间插入的颜色点总是用一个相对量 Offset 来表示，如红色点的 Offset 为 0.25，表示此点的位置是在开始位置 (StartPoint) 向终点位置 (EndPoint) 的方向 1/4 处；蓝色点是在开始位置 (StartPoint) 向终点位置 (EndPoint) 的方向 3/4 处。

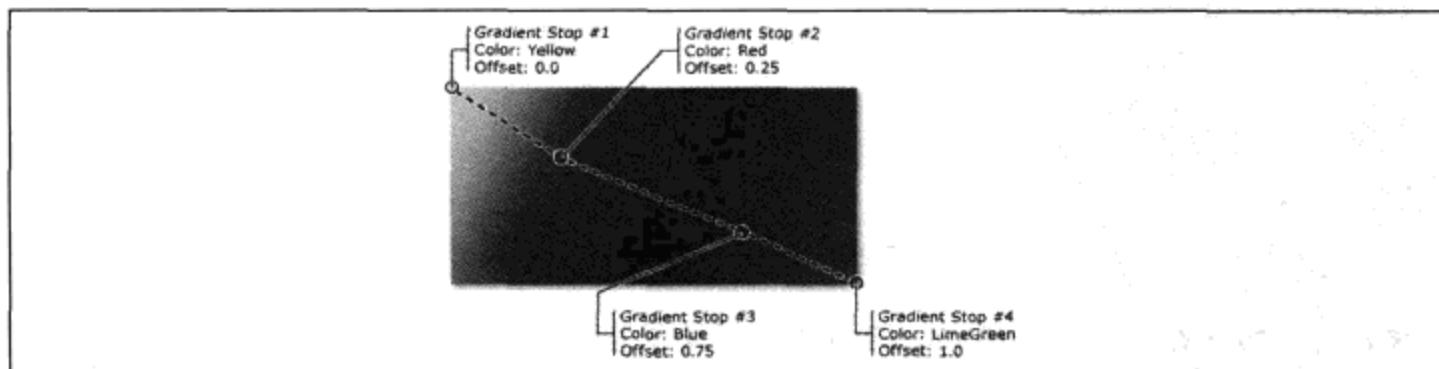


图 15-45 用相对坐标表示两点位置，在两点之间插入颜色

如果起始位置不在左上角点，终点位置也不在右下角点，则意味着有一部分超出了起始点和终点的范围。渐变画刷提供了一个 SpreadMethod 属性，适用于 LinearGradientBrush 和 RadialGradientBrush。它的 3 个枚举值是 Pad、Reflect 和 Repeat。图 15-46 所示的起始点和终点的相对坐标均为（0.5,0.5）和（1.0,0.5），不同的是其 SpreadMethod 属性。

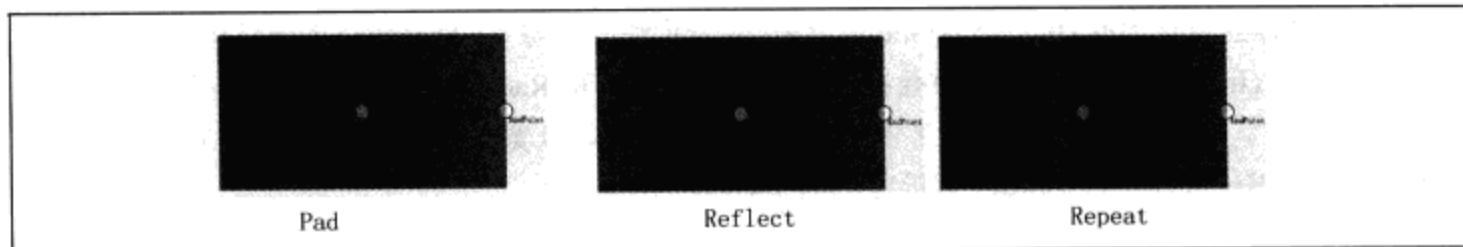


图 15-46 不同 SpreadMethod 属性效果

代码 15-30 是一个线性画刷应用在 Border 背景色的示例。

```
<Border.Background>
    <LinearGradientBrush>
        <GradientStop Offset="0" Color="Blue" />
        <GradientStop Offset="0.5" Color="Purple" />
        <GradientStop Offset="1" Color="Red" />
    </LinearGradientBrush>
</Border.Background>
```

代码 15-30 线性画刷应用在 Border 背景色的示例

如果需要进一步探究线性画刷的各种属性，可以参见随附光盘中的示例 `mumu_brushes`。在其中可以拖动矩形上的起始点和终点，也可以通过设置上面 `LinearGradientBrush` 的参数来观察其效果，如图 15-47 所示。

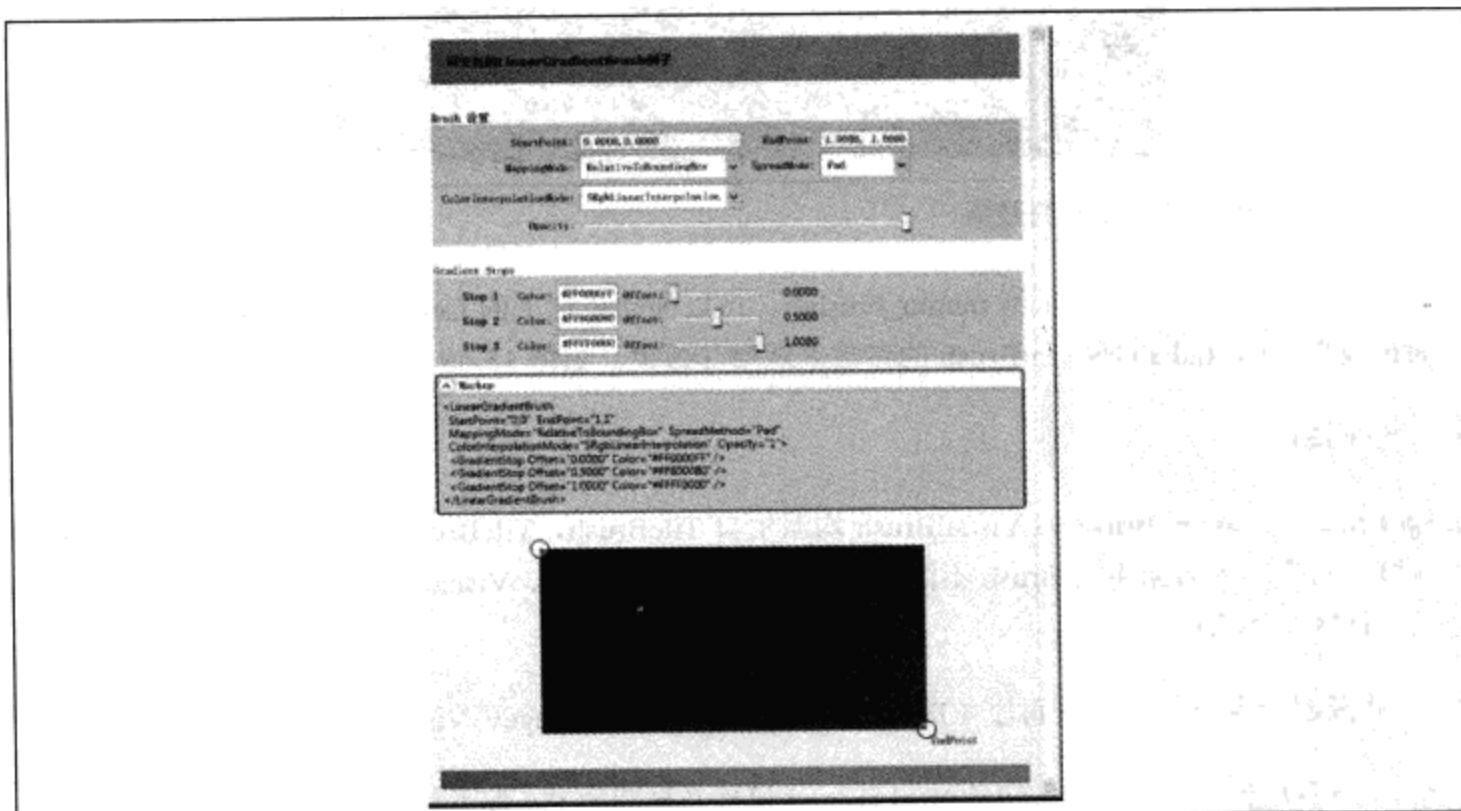


图 15-47 配套示例 `mumu_brushes` 效果

(2) RadialGradientBrush

RadialGradientBrush 以一个起始点呈椭圆状向外散发渐变，与 LinearGradientBrush 共享 GradientBrush 的属性，如 MappingMode（指定相对坐标或绝对坐标）和 SpreadMethod 等。RadialGradientBrush 由 Center、GradientOrigin、RadiusX 和 RadiusY 属性决定渐变方式，其中 Center 表示填充椭圆的中心位置，相对坐标的默认值是(0.5,0.5)；GradientOrigin 表示渐变的源点，相对坐标的默认值是(0.5,0.5)，即 GradientOrigin 和 Center 默认是重叠在一起的；RadiusX 和 RadiusY 表示椭圆的半径，其值均为 0.5，表示默认时该椭圆的长短半径是填充区域的长和宽的一半。代码 15-31 是一个 RadialGradientBrush 应用在 Border 背景色的例子。

```
<Border.Background>
    <RadialGradientBrush>
        <GradientStop x:Name="GradientStop1" Offset="0" Color="Blue" />
        <GradientStop x:Name="GradientStop2" Offset="0.5" Color="Purple" />
        <GradientStop x:Name="GradientStop3" Offset="1" Color="Red" />
    </RadialGradientBrush>
</Border.Background>
```

代码 15-31 RadialGradientBrush 应用在 Border 背景色

该 Border 对象填充的效果如图 15-48 所示。

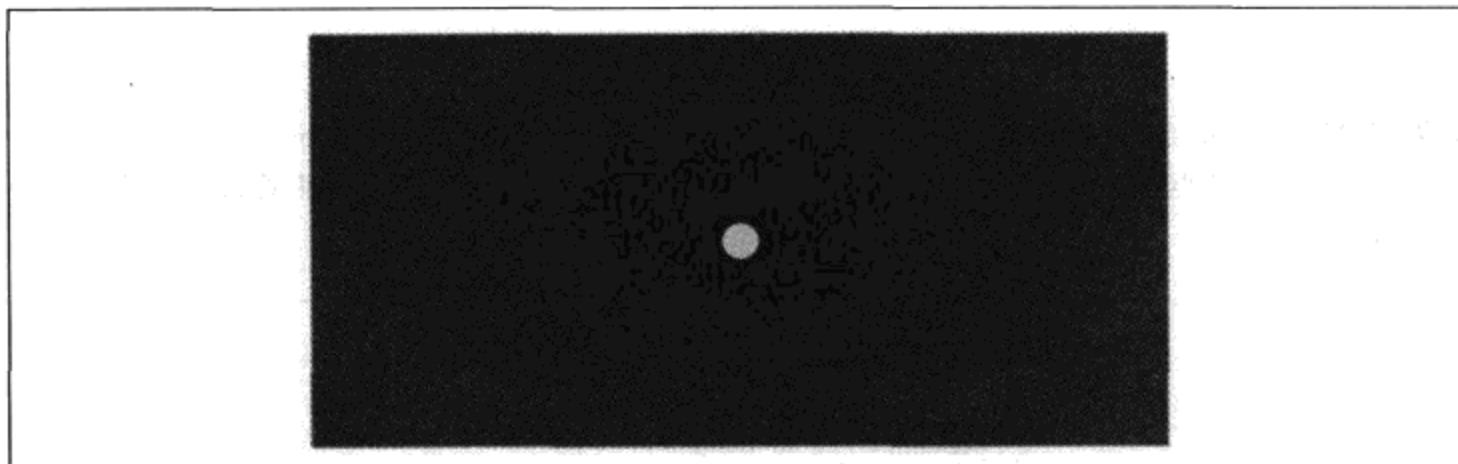


图 15-48 Border 对象填充的效果

同样可以参见随附光盘中的示例 `mumu_brushes`，可以拖动矩形上的 Center 和 GradientOrigin，也可以通过设置上面 RadialGradientBrush 的参数来观察其效果，如图 15-49 所示。

3. Tile 画刷

ImageBrush、DrawingBrush 和 VisualBrush 均派生自 TileBrush，TileBrush 可以用重复图案来填充目标区域。只不过因为派生的 Brush 不同，所以图案可以是 Image、Visual 或者 Drawing。一幅图案填充到目标区域经历如下两步。

(1) 从画刷的内容到一个小面片 (Tile)，画刷内容可以是 Image、Visual 或者 Drawing。

(2) 从小面片到目标区域，按钮的 Background 属性及椭圆的 Fill 属性均属于目标区域，如图 15-50 所示。

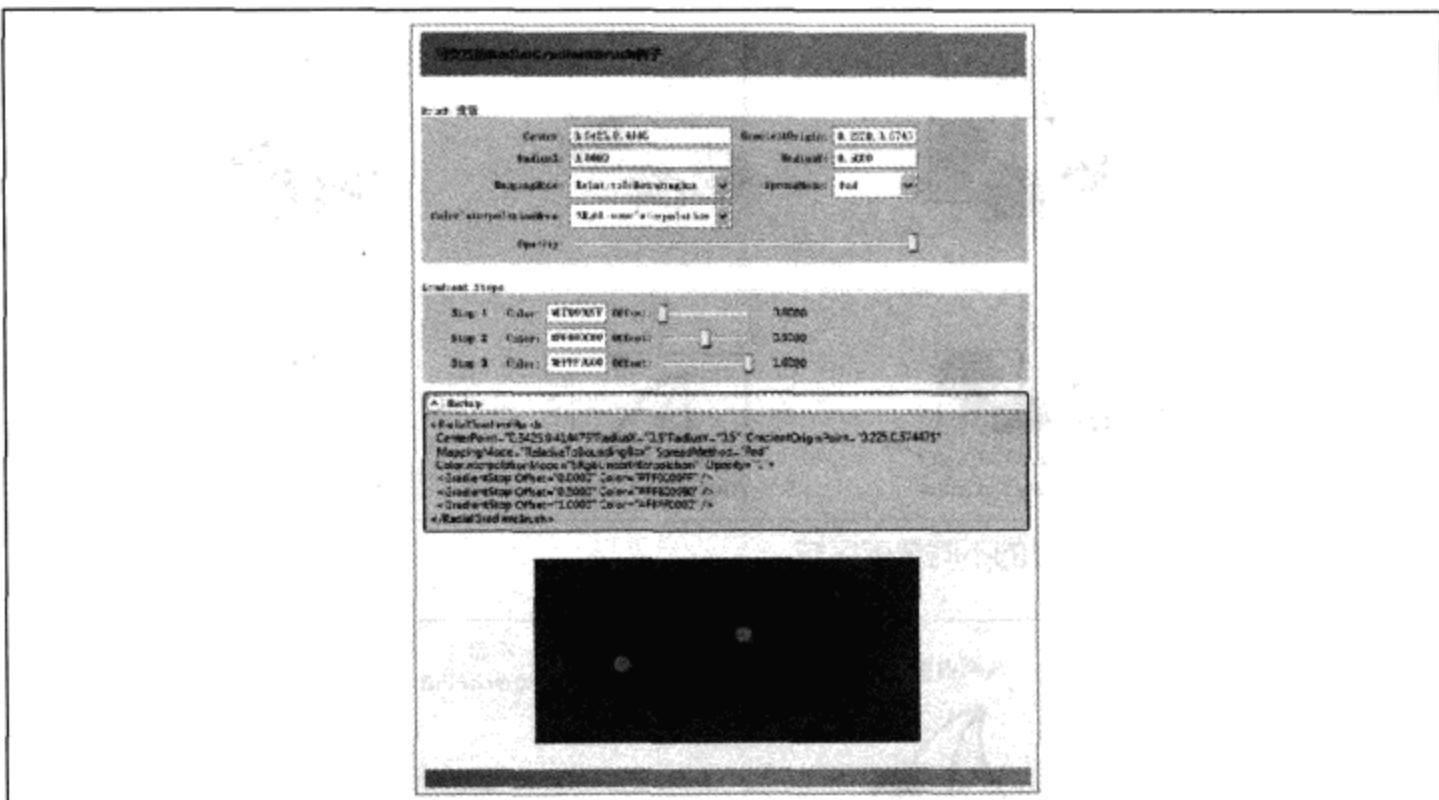


图 15-49 配套示例 `mumu_brushes` 运行效果示意图

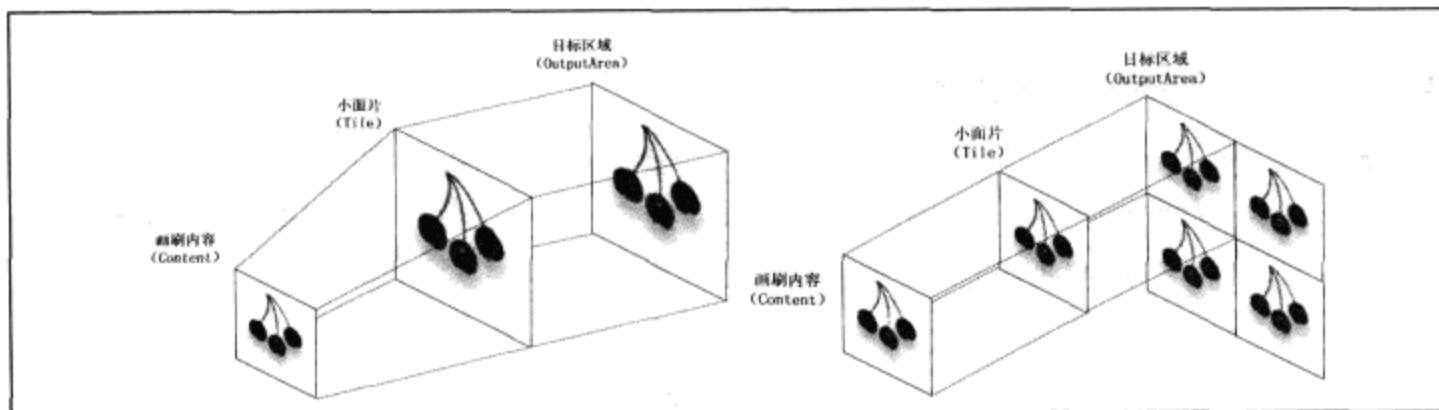


图 15-50 从图案填充到目标区域

画刷的内容分别由不同类型的画刷决定, `ImageBrush` 通过 `ImageSource` 指定画刷内容; `DrawingBrush` 通过 `Drawing` 属性指定; `VisualBrush` 通过 `Visual` 属性指定。

`TileBrush` 中有如下属性用来指定画刷内容如何映射到小面片中。

(1) `Stretch` 属性: 其 4 种枚举值为 `None`、`Fill`、`Uniform` 和 `UniformToFill`。为了说明其区别, 我们以 `ImageBrush` 画刷为例。假定该画刷的内容为一幅草莓的图片, 其长度和宽度均为 150 DPI, 而需要映射的小面片长度为 100 DPI, 宽度为 50 DPI。

- 枚举值为 `None`, 图片会被小面片所裁减。裁减的区域可以通过 `AlignmentX` 和 `AlignmentY` 确定, 如图 15-51 所示。
- 枚举值为 `Fill`, 图片会填充整个小面片。由于图片的长宽比为 1:1, 而小面片的长宽比为 2:1, 所以会造成图片的变形。这时 `AlignmentX` 和 `AlignmentY` 属性无任何作用, 如图 15-52 所示。

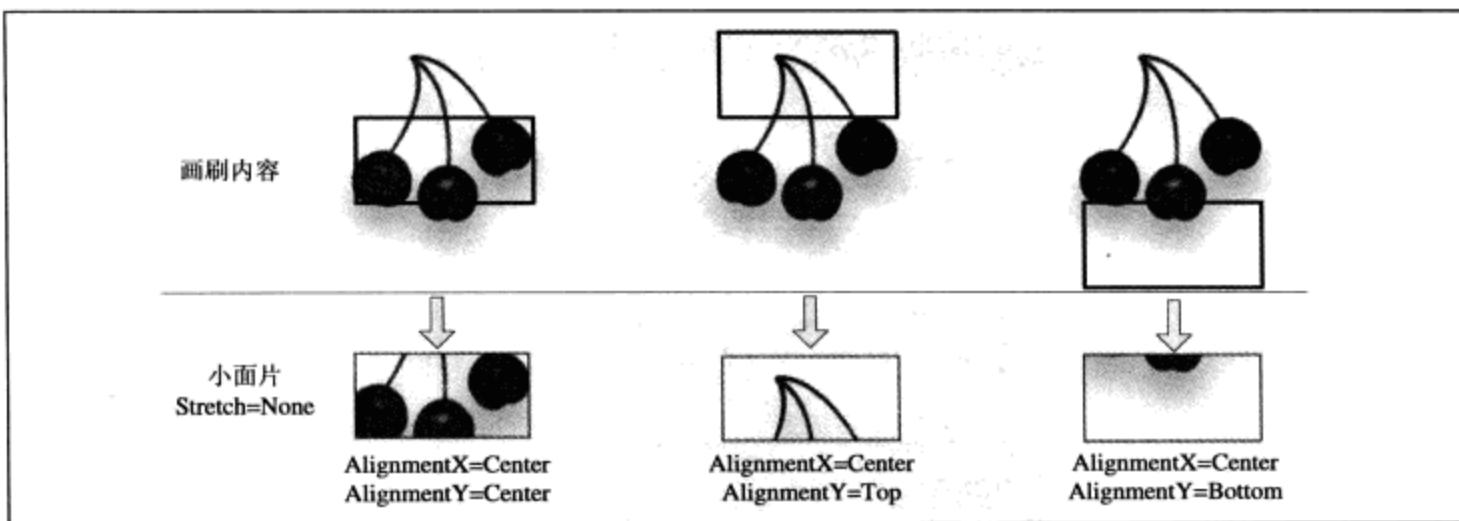


图 15-51 枚举值为 None 时的不同裁剪区域

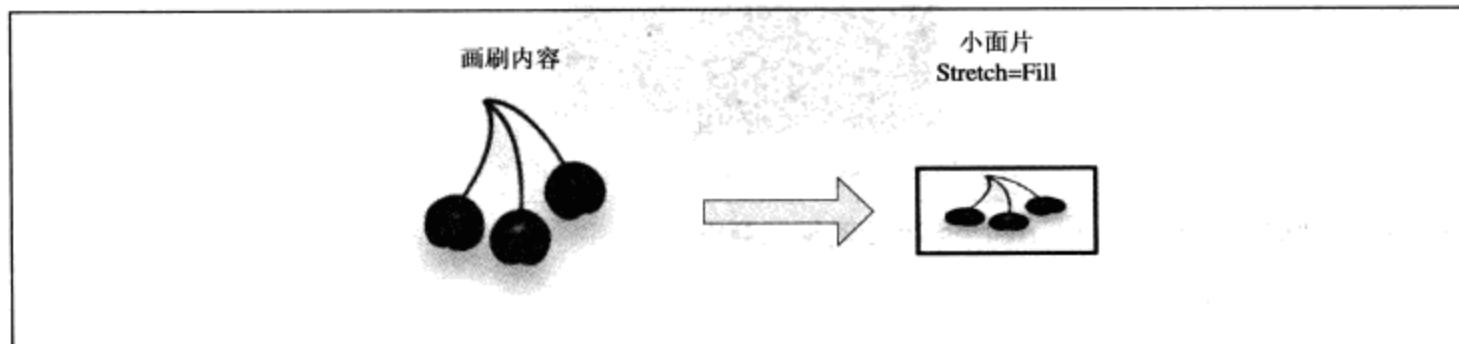


图 15-52 枚举值为 Fill 时的图片填充

- 枚举值为 Uniform，需要保持图片不变形，然后成比例地缩小填入到小面片中，图片的长度和宽度取小面片的长宽的最小值 50 DPI。由于小面片的长度为 100 DPI，宽度为 50 DPI，因此只有 AlignmentX 有效，而 AlignmentY 无效，如图 15-53 所示。

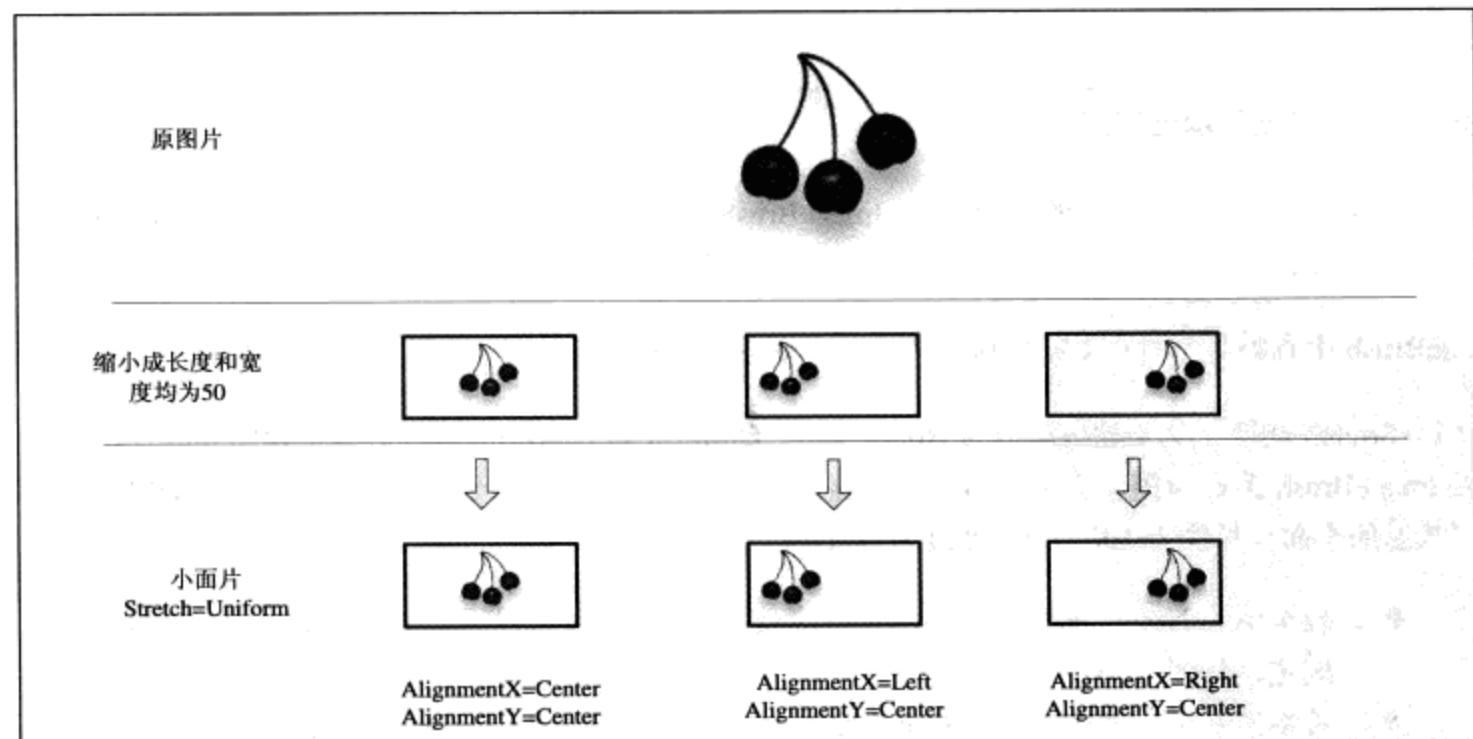


图 15-53 枚举值为 Uniform 时的图形填充

- 枚举值为 UniformToFill，首先要考虑的问题是需要完全填充小面片。如果图片超出小面片的区域，则被裁减。因此这里图片的长度和宽度变为 100 DPI，由于小面片的长度为 100 DPI，宽度为 50 DPI，因此 AlignmentX 无效，而 AlignmentY 有效，如图 15-54 所示。

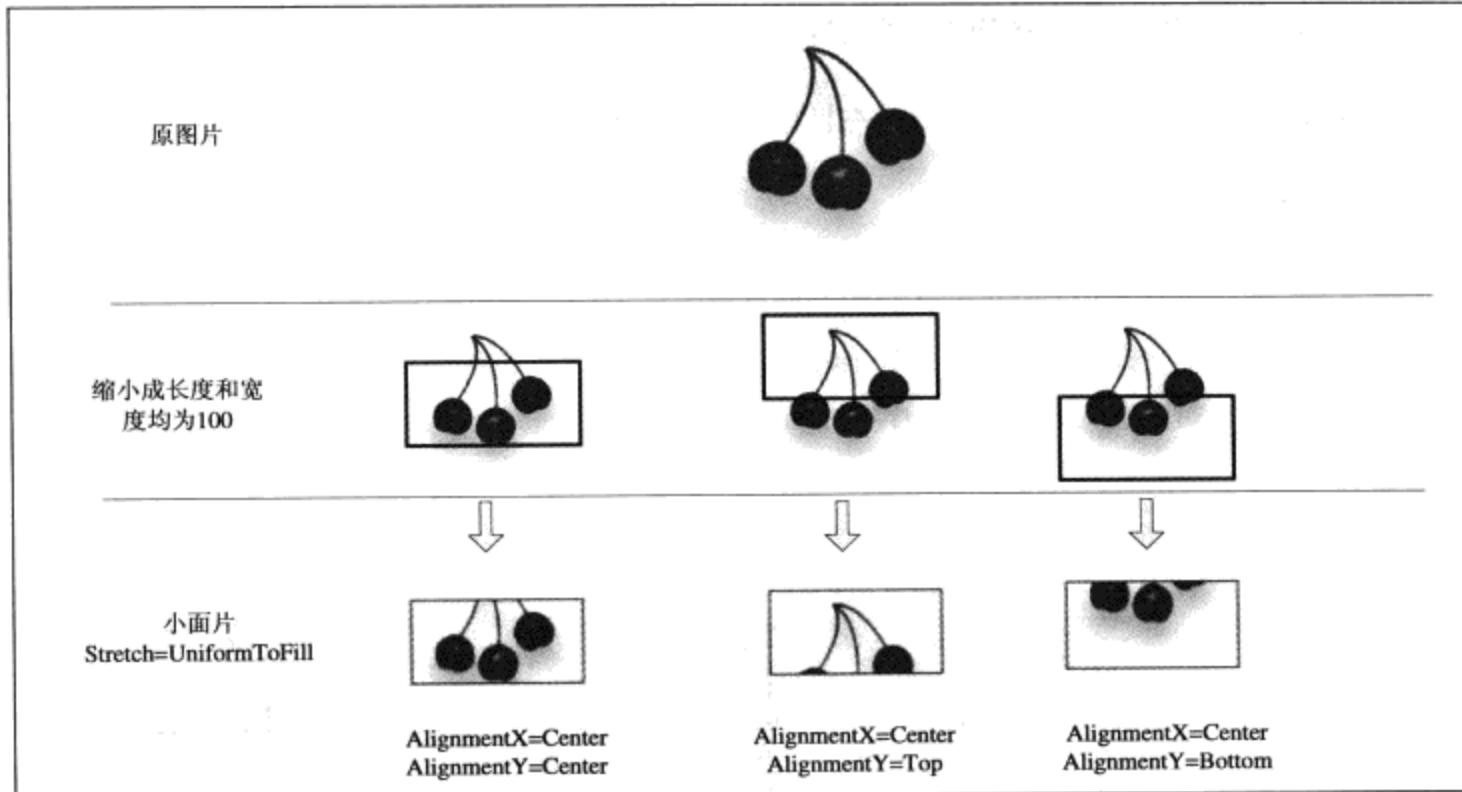


图 15-54 枚举值为 UniformToFill 时的图形填充

(2) Viewbox 属性：指定映射到小面片中的画刷内容，它是一个 Rect 类型，可以使用相对坐标和绝对坐标，由 ViewboxUnits 属性指定。相对坐标仍然是图片的左上角点为(0,0)，右下角点为(1,1)。ViewBox 的默认值是(0,0,1,1)，即画刷的全部内容均映射到小面片上。如果 Viewbox 为(0,0,0.5,0.5)，则表示画刷内容左上角的 1/4 部分映射到小面片上，如图 15-55 所示。

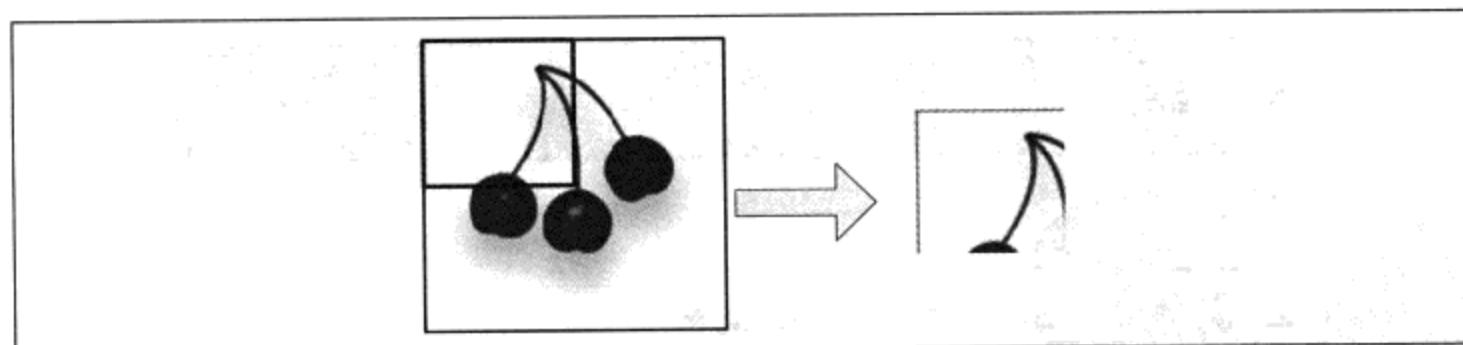


图 15-55 画刷内容映射到小面片上

当画刷的内容映射到小面片后，第 2 步则是考虑小面片如何映射到目标区域，TileBrush 的如下属性用来指定小面片如何映射到目标区域中。

(1) Viewport 属性，它是 Rect 类型，指定小面片映射到目标区域的哪个部分。Viewport 采用相对坐标还是绝对坐标由 ViewportUnits 属性决定。如果 Viewport 值为 (0,0,0.5,0.5) 且 TileMode 属性为 None，那么目标区域只有左上角的 1/4 被图片填充，如图 15-56 所示。

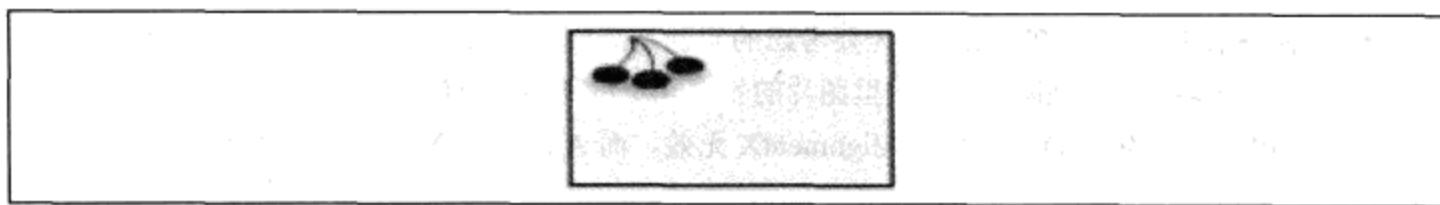


图 15-56 目标区域只有左上角的 1/4 被图片填充

(2) **TileMode** 属性：描述小面片如何填充目标区域，其枚举值如下。

- **None**: 不重复填充。
- **Tile**: 重复填充。
- **FlipX**: 在水平轴上对小面片隔列翻转填充。
- **FlipY**: 在垂直轴上对小面片隔行翻转填充。
- **FlipXY**: 在两个方向对小面片隔行隔列翻转填充。

如图 15-57 所示。

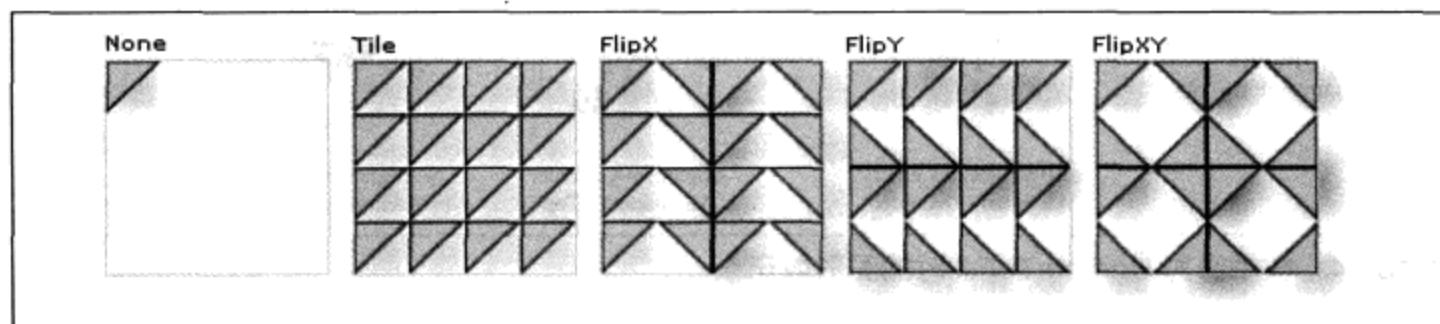


图 15-57 TileMode 属性的不同枚举值填充

请参见随附光盘中的示例 **mumu_brushes**，可以通过设置 **ImageBrush** 的参数来观察其效果，如图 15-58 所示。

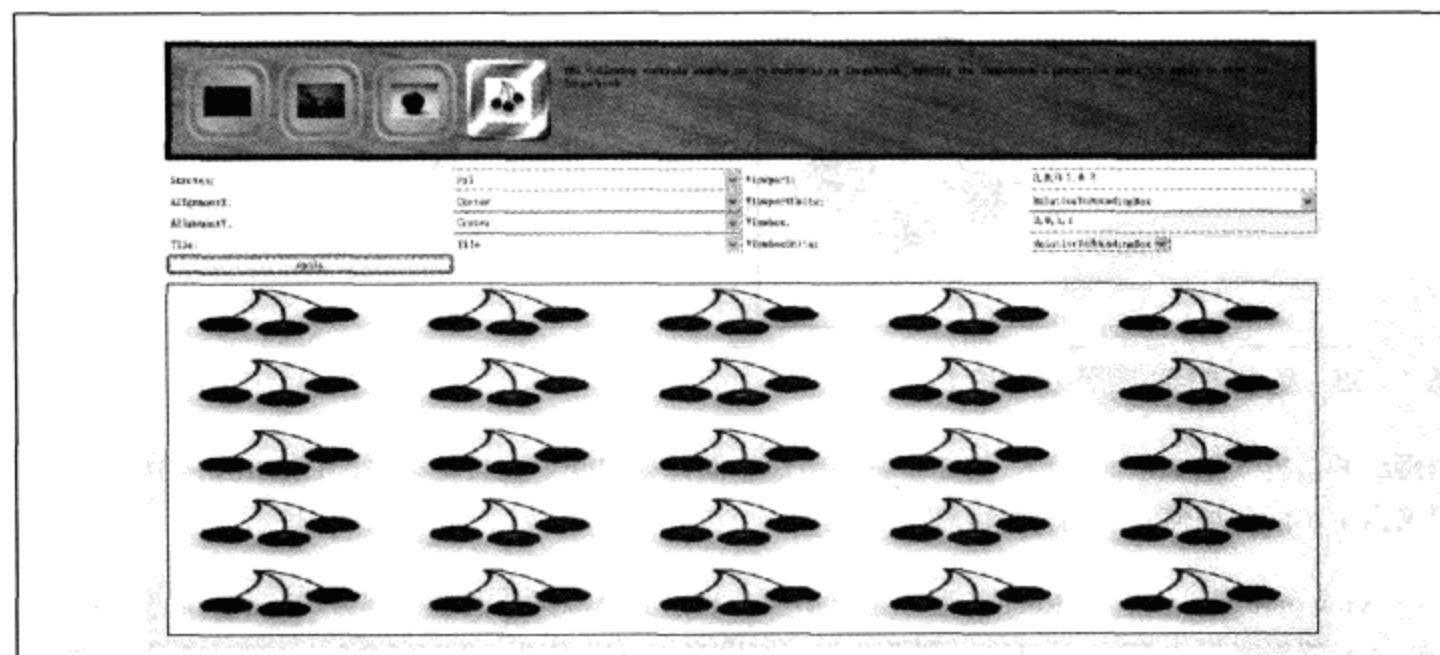


图 15-58 配套示例 **mumu_brushes** 的效果

15.4.3 使用画刷制作特效

WPF 画刷的功能异常强大，我们在本节中以两个示例说明。

1. 放大镜效果

实现放大镜效果需要使用 VisualBrush 画刷，我们可以将一个椭圆（为了模拟放大镜）的 Fill 属性设置为 VisualBrush 画刷，而其 Visual 属性为需要放大的元素。当鼠标移动时，通过改变画刷的 Viewbox 属性即可实时显示需要放大的内容，如图 15-59 所示。



图 15-59 放大镜效果

首先在一个 Grid 面板中放置一个文档和一个 Canvas，在 Canvas 中绘制一个椭圆作为放大镜。椭圆的 Fill 属性实际上是一个 DrawingBrush，其中包括两个矩形，一个使用固定颜色（白色）的画刷填充，作为放大镜的背景；另一个使用 VisualBrush 的 Visual 属性指定为容纳文档的 StackPanel 面板，如代码 15-32 所示。

```
<StackPanel Name="magnifiedPanel"
    VerticalAlignment="Stretch"
    MouseMove="updateMagnifyingGlass" >
    <FlowDocumentReader>
        <FlowDocument >
            ...
        </FlowDocument >
    </FlowDocumentReader>
</StackPanel>
<Canvas Name="magnifyingGlassCanvas">
    <Ellipse Name="magnifyingGlassEllipse" Width="100" Height="100"
    Stroke="Black">
        <Ellipse.Fill>
            <DrawingBrush>
                <DrawingBrush.Drawing>
                    <DrawingGroup>
                        <DrawingGroup.Children>
                            <GeometryDrawing Brush="White">
                                <GeometryDrawing.Geometry>
                                    <RectangleGeometry Rect="0,0,1,1" />
                                </GeometryDrawing.Geometry>
                            </GeometryDrawing>
                        </DrawingGroup.Children>
                    </DrawingGroup>
                </DrawingBrush.Drawing>
            </DrawingBrush>
        </Ellipse.Fill>
    </Ellipse>
</Canvas>
```

```

        </GeometryDrawing>
        <GeometryDrawing>
            <GeometryDrawing.Brush>
                <VisualBrush x:Name="myVisualBrush"
ViewboxUnits="Absolute"
                    Visual="{Binding ElementName=magnifiedPanel}"/>
                </GeometryDrawing.Brush>
                <GeometryDrawing.Geometry>
                    <RectangleGeometry Rect="0,0,1,1" />
                </GeometryDrawing.Geometry>
            </GeometryDrawing>
        </DrawingGroup.Children>
    </DrawingGroup>
</DrawingBrush.Drawing>
</DrawingBrush>
</Ellipse.Fill>
</Ellipse>
</Canvas>

```

代码 15-32 放大镜效果

当鼠标在 StackPanel 面板上移动时,根据其当前的点实时地修改 VisualBrush 画刷的 Viewbox 属性来改变其放大的内容,如代码 15-33 所示。

```

private void updateMagnifyingGlass(object sender, MouseEventArgs args)
{
    Mouse.SetCursor(Cursors.Cross);
    Point currentMousePosition = args.GetPosition(this);

    if (this.ActualWidth - currentMousePosition.X > magnifyingGlassEllipse.Width +
distanceFromMouse)
    {
        Canvas.SetLeft(magnifyingGlassEllipse, currentMousePosition.X +
distanceFromMouse);
    }
    else
    {
        Canvas.SetLeft(magnifyingGlassEllipse, currentMousePosition.X -
distanceFromMouse - magnifyingGlassEllipse.Width);
    }

    if (this.ActualHeight - currentMousePosition.Y > magnifyingGlassEllipse.Height +
distanceFromMouse)
    {
        Canvas.SetTop(magnifyingGlassEllipse, currentMousePosition.Y +
distanceFromMouse);
    }
    else
    {
        Canvas.SetTop(magnifyingGlassEllipse, currentMousePosition.Y -
distanceFromMouse - magnifyingGlassEllipse.Height);
    }

    myVisualBrush.Viewbox =
        new Rect(currentMousePosition.X - 10, currentMousePosition.Y - 10,
20, 20);
}

```

代码 15-33 根据鼠标的当前点实时地修改 VisualBrush 画刷的 Viewbox 属性来改变放大内容

2. 倒影效果

如一个位图 (img1) 需要实现倒影效果，可以将一个与该位图同样大小的矩形放置在其正下方。并且用 VisualBrush 画刷填充，该画刷的 Visual 属性指向 img1。不过该画刷需要沿 X 轴执行一个对称变换，如代码 15-34 所示。

```
<Image x:Name="ElementVisual"
       Source="Images/battery.png"
       Stretch="Fill"
       Grid.Row="0" />
<Rectangle
    Grid.Row="1"
    Width="{Binding ActualWidth, ElementName=ElementVisual}"
    Height="{Binding ActualHeight, ElementName=ElementVisual}">
    <Rectangle.Fill>
        <VisualBrush Visual="{Binding ElementName=ElementVisual}">
            <VisualBrush.RelativeTransform>
                <ScaleTransform ScaleX="1"
                               ScaleY="-1"
                               CenterX="0.5"
                               CenterY="0.5" />
            </VisualBrush.RelativeTransform>
        </VisualBrush>
    </Rectangle.Fill>
</Rectangle>
```

代码 15-34 位图实现倒影效果示例

运行结果如图 15-60 所示。

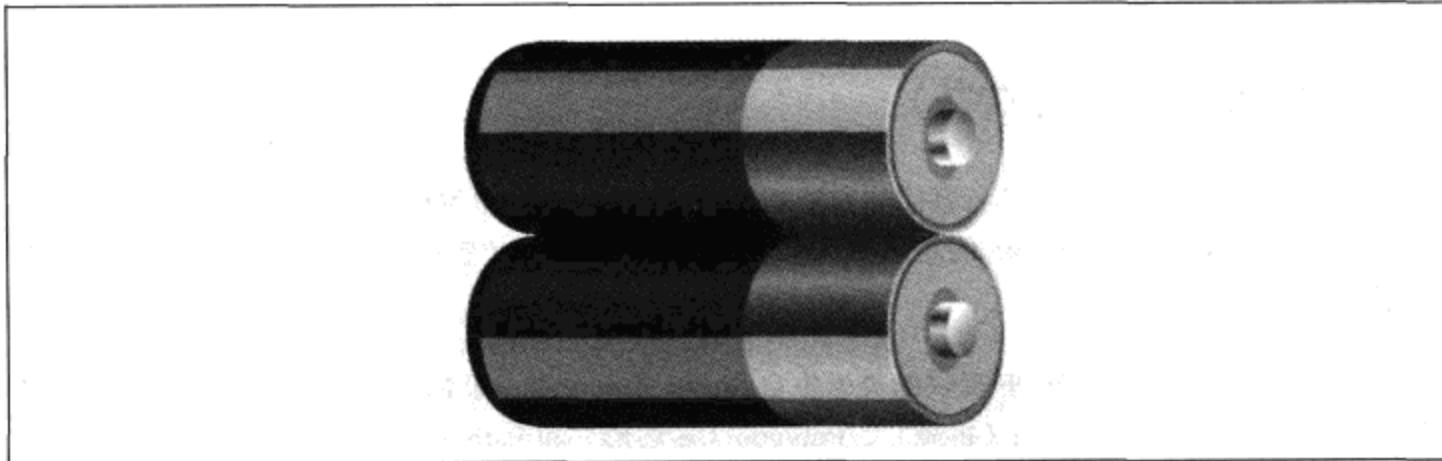


图 15-60 运行结果

但是这并不是理想的倒影效果。可以使用另外一个渐变画刷作为透明掩码 (OpacityMask)，凡派生自 UIElement 的类均有 OpacityMask 属性。该属性是一个画刷类型，以往使用 Opacity 属性来控制要素的透明程度，OpacityMask 比其更能精准地控制要素的透明部分。在这里我们使用一个由红渐变到透明的渐变画刷作为透明掩码，事实上使用由蓝渐变或者由绿渐变均可。关键是需要一个完全不透明的颜色，渐变画刷与 VisualBrush 合成效果如图 15-61 所示。

整个倒影效果如图 15-62 所示。

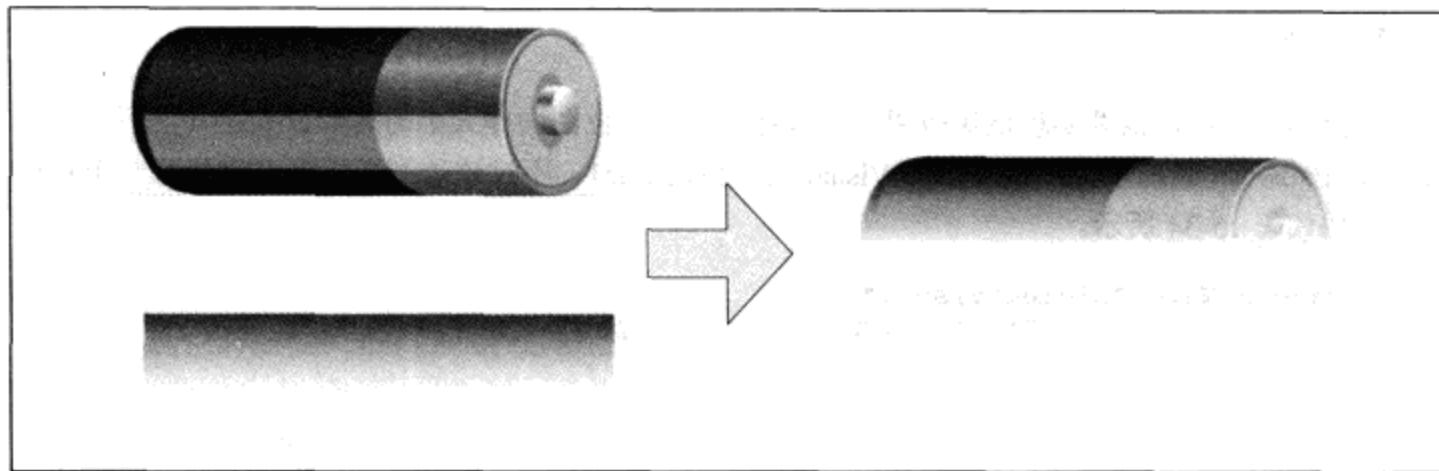


图 15-61 渐变画刷和 VisualBrush 合成效果

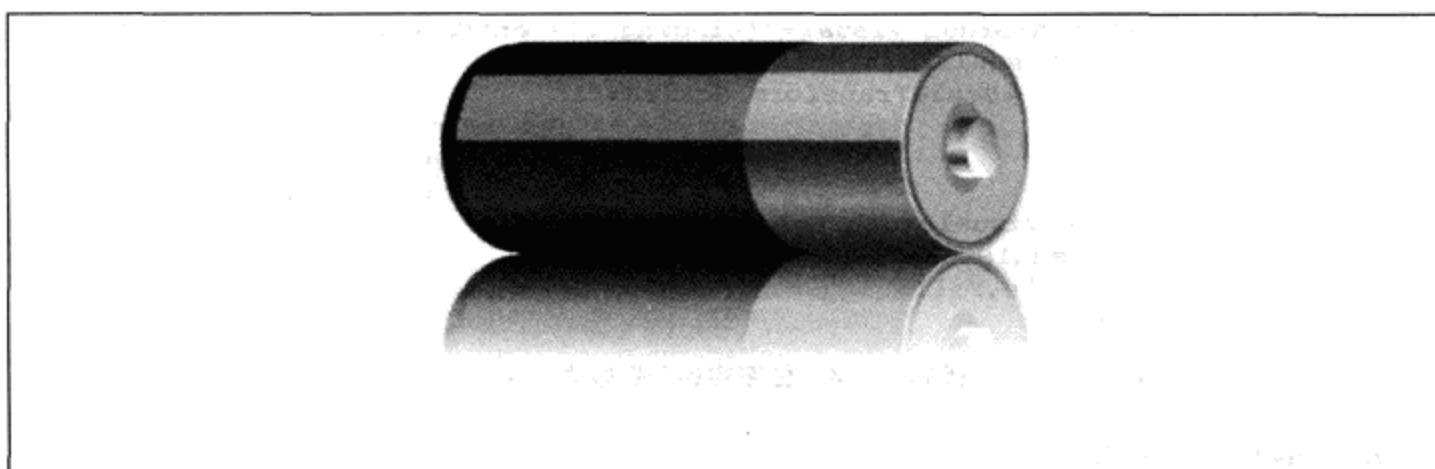


图 15-62 整个倒影效果

15.5 Shape (第二本书)

在 WPF 中绘制二维图形最简单和直接的方法是通过 **Shape**，由于派生自 **FrameworkElement**，因此其具备很多高级特性。如 **Shape** 的绘制由自身负责，并可以直接放置在布局容器中，也可以支持键盘及鼠标等事件。

Shape 本身是一个抽象类，WPF 提供的类派生自 **Shape** 的 6 个类，即 **Rectangle**（矩形）、**Ellipse**（椭圆）、**Line**（直线）、**Polyline**（折线）、**Polygon**（多边形）和 **Path**（路径）。

Shape 中有如下 3 个常用属性。

- (1) **Stroke** 属性：描述如何绘制 **Shape** 的轮廓线，WPF 将 **Stroke** 定义为一个画刷类型。
- (2) **StrokeThickness** 属性：描述 **Shape** 轮廓线的宽度。
- (3) **Fill** 属性：描述 **Shape** 内部如何填充，也为一个画刷类型。

Shape 必须显式设置 **Stroke** 或者 **Fill** 才能显示。

15.5.1 简单的 Shape 元素

1. Rectangle

Rectangle 通过继承的 FrameworkElement 的 Width 和 Height 属性来控制矩形的宽和高，并且定义了 double 类型的 RadiusX 和 RadiusY 属性来使矩形具有圆角功能。代码 15-35 展示不同 RadiusX 和 RadiusY 值的矩形。

```
<Canvas Width="400" Height="500" >
    <Rectangle Width="200" Height="100" Fill="Blue"
    Stroke="Black" Canvas.Left="100" Canvas.Top="30" />
        <TextBlock Canvas.Left="123" Canvas.Top="134" Height="15"
    Width="178">
            RadiusX=0 RadiusY=0
        </TextBlock>
        <Rectangle Width="200" Height="100" Fill="Blue" Stroke="Black"
RadiusX="20" RadiusY="20" Canvas.Left="100" Canvas.Top="170"/>
            <TextBlock Canvas.Left="122" Canvas.Top="284" Height="15"
Width="178">
                RadiusX=20 RadiusY=20
            </TextBlock>
            <Rectangle Width="200" Height="100" Fill="Blue" Stroke="Black"
RadiusX="100" RadiusY="50" Canvas.Left="100" Canvas.Top="320"/>
                <TextBlock Canvas.Left="122" Canvas.Top="426" Height="15"
Width="178">
                    RadiusX=100 RadiusY=50
                </TextBlock>
            </Canvas>
```

代码 15-35 展示不同 RadiusX 和 RadiusY 值的矩形

如图 15-63 所示，矩形的宽高分别为 200 和 100。RadiusX 最大为 Width 的一半 100，RadiusY 最大为 Height 的一半 50，这时矩形已经变为一个椭圆。即使 RadiusX 和 RadiusY 超过上述值也不会影响结果。

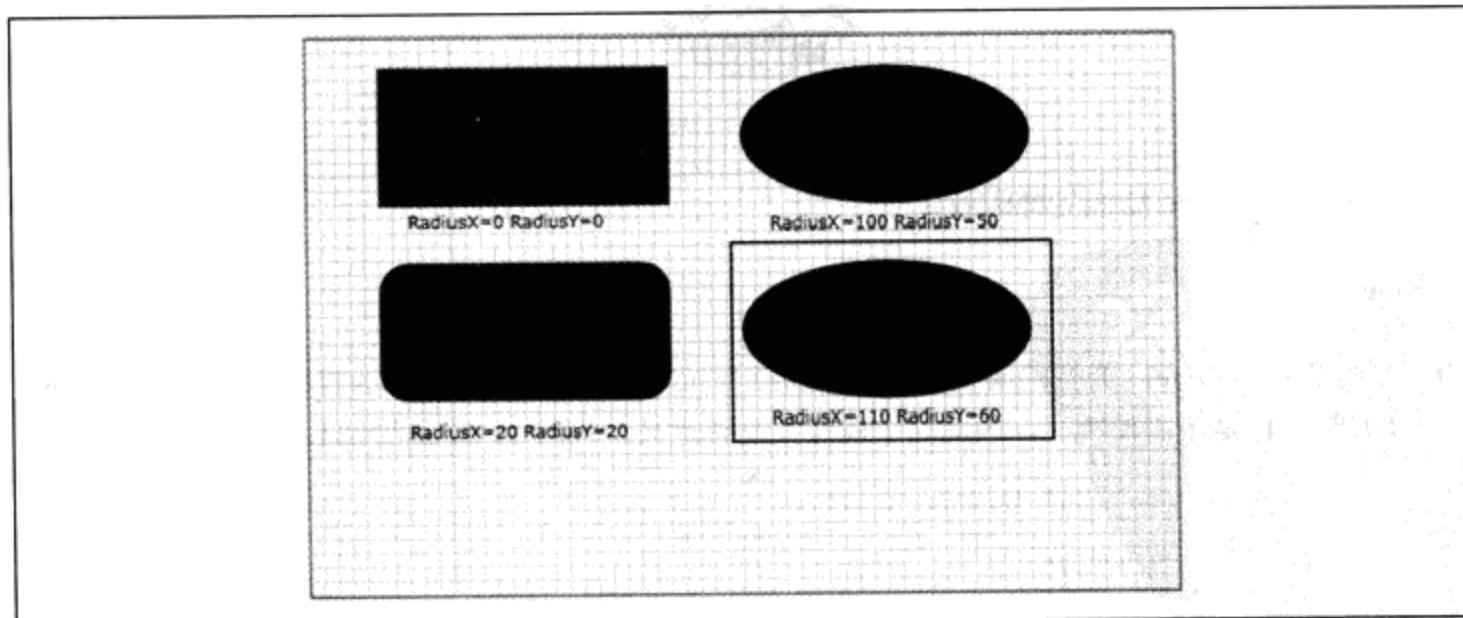


图 15-63 不同 RadiusX 和 RadiusY 值的矩形

2. Ellipse

Rectangle 可以通过设置 RadiusX 和 RadiusY 绘制椭圆，而 Ellipse 则可以更容易得到一个椭圆形，如代码 15-36 所示。

```
<Canvas Height="200" Width="130">
    <Ellipse
        Width="100"
        Height="50"
        Fill="Blue"
        Canvas.Left="10"
        Canvas.Top="25" />
    <Ellipse
        Width="100"
        Height="100"
        Fill="Blue"
        Stroke="Black"
        StrokeThickness="4"
        Canvas.Left="10"
        Canvas.Top="81"/>
</Canvas>
```

代码 15-36 绘制 Ellipse

绘制的椭圆效果如图 15-64 所示。

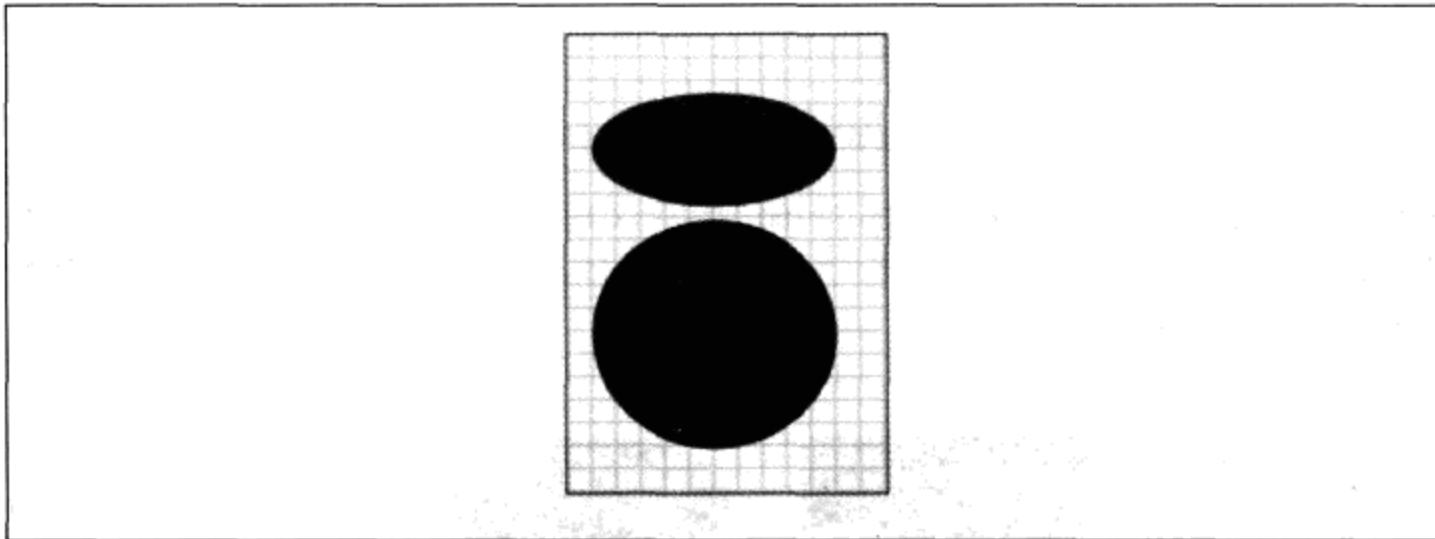


图 15-64 Rectangle 绘制的椭圆效果

3. Line

Line 定义了 4 个 double 类型的属性用来表示端点为 (X1,Y1) 和 (X2,Y2) 的直线，它继承了 Shape 的 Fill 属性，但是不需要填充，因此这个属性并没有意义。代码 15-37 是绘制 Line 的示例。

```
<Canvas Height="300" Width="300">
    <Line
        X1="10" Y1="10"
        X2="50" Y2="50"
        Stroke="Black"
        StrokeThickness="4" />
    <Line
        X1="10" Y1="60"
```

```
X2="150" Y2="60"
Stroke="Black"
StrokeThickness="4"/>
</Canvas>
```

代码 15-37 绘制 Line 的示例

绘制 Line 的示例效果如图 15-65 所示。

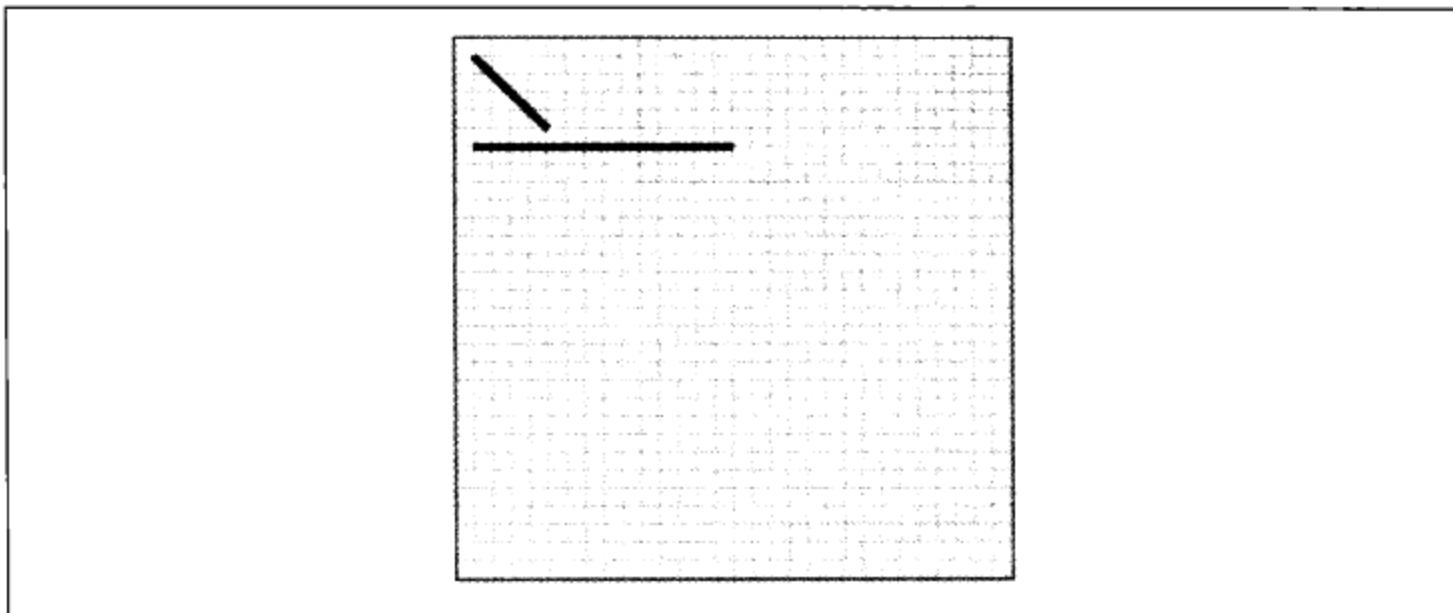


图 15-65 绘制 Line 的示例效果

4. Polyline 和 Polygon

Polyline 和 Polygon 均使用 Points 属性（Point 对象的集合）来表示一组线段，其区别在于后者自动添加一条从第 1 个点到最后一个点的连线，从而形成一个多边形；前者也可以被填充。代码 15-38 是 Polyline 和 Polygon 的示例。

```
<Canvas Height="400" Width="400">
    <Polyline
        Points="10,110 60,10 110,110"
        Stroke="Black"
        StrokeThickness="4" Fill="Red"/>
    <Polyline
        Points="10,110 110,110 110,10"
        Stroke="Black"
        StrokeThickness="4"
        Canvas.Left="150" />
    <Polygon
        Points="10,110 60,10 110,110"
        Stroke="Black"
        StrokeThickness="4" Fill="Red" Canvas.Top="120"/>
    <Polygon
        Points="10,110 110,110 110,10"
        Stroke="Black"
        StrokeThickness="4" Canvas.Top="120"
        Canvas.Left="150" />
</Canvas>
```

代码 15-38 Polyline 和 Polygon 的示例

效果如图 15-66 所示。

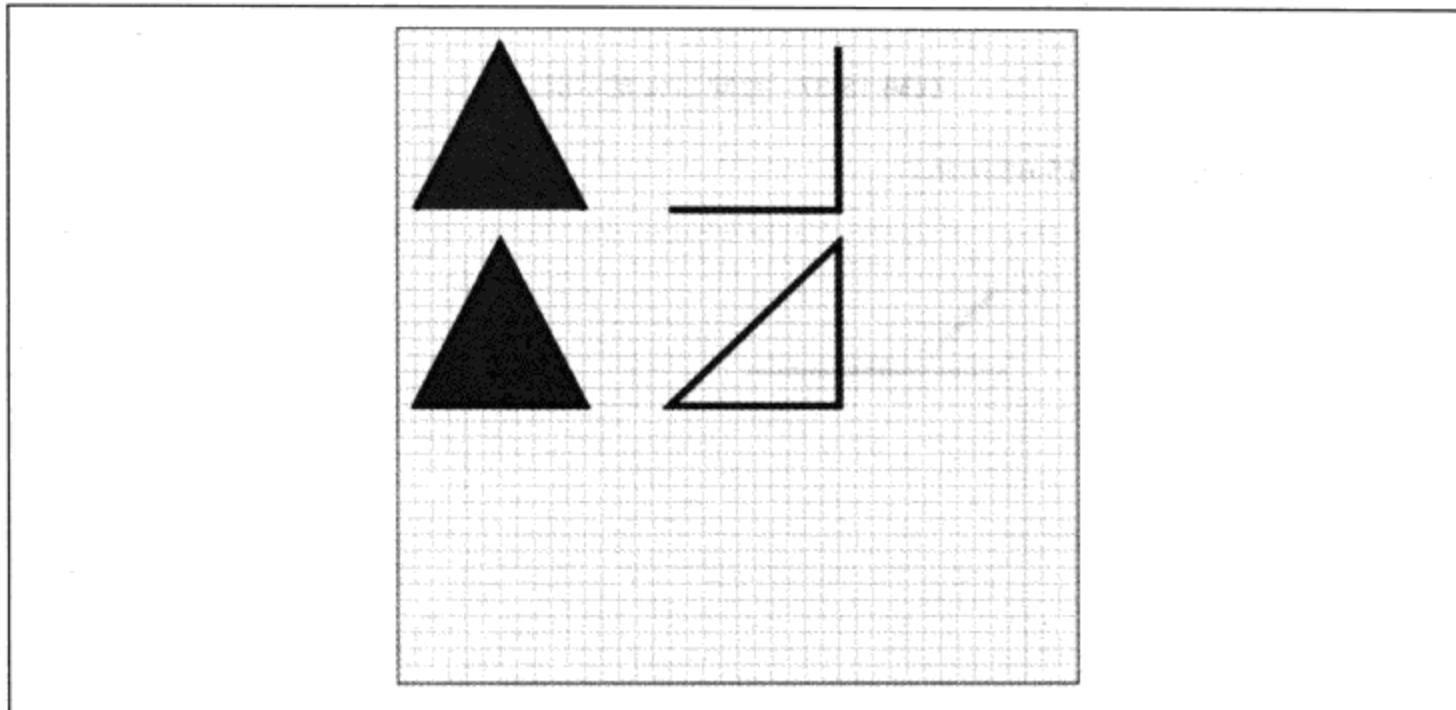


图 15-66 Polyline 和 Polygon 的示例效果

15.5.2 线型、线帽、线的连接和填充规则

一个图形元素基本上由外部轮廓线和内部的填充组成。

Shape 没有提供一个直接的画笔 (Pen) 属性来描述轮廓线，而是提供了一个画刷类型的 Stroke 来填充轮廓线。同时提供了 8 个 Stroke 为开头的属性用来调整内部 Pen 的设置，WPF 的设计者认为使用 Pen 的某一个属性比用一个单独的 Pen 对象要容易得多。

Shape 为填充提供了一个画刷类型的 Fill 属性，本节我们主要讨论两种填充规则 (EvenOdd 和 NonZero)。

1. 线型

线型主要由 StrokeDashArray 和 StrokeDashOffset 两个属性控制，前者类型为 Double 型的集合，通过一串 double 数值来描述线型。如将其设置为“5, 3”，那么表示该线段填充 5 个单元，然后空 3 个单元并这样反复循环；如设置为“5, 1, 3”，则表示线段填充 5 个单元，空 1 个单元，再填充 3 个单元，空 5 个单元，再填充 1 个单元，这样反复循环。这里所指的单元为当前线的宽度，由 StrokeThickness 来控制，如图 15-67 所示。

后者用来描述这种线型的开始位置，如一种线型的 StrokeDashArray 为 5, 3 的直线。设置其 StrokeDashOffset 属性值为 2，则表示该线型的第 1 段只填充 3 个单元 (5-2)，空 3 个单元。然后填充 5 个单元，空 3 个单元。如此反复循环，如图 15-68 所示。

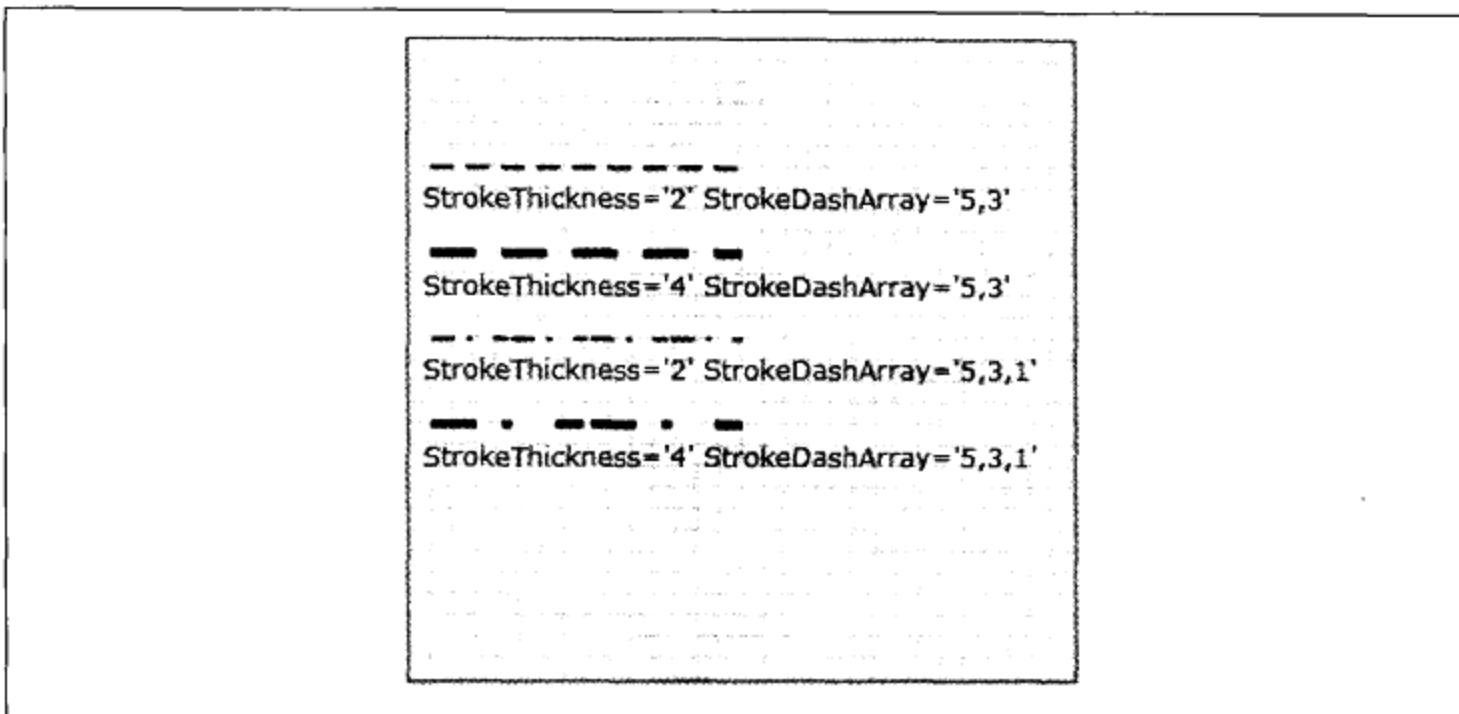


图 15-67 不同属性线型的控制效果

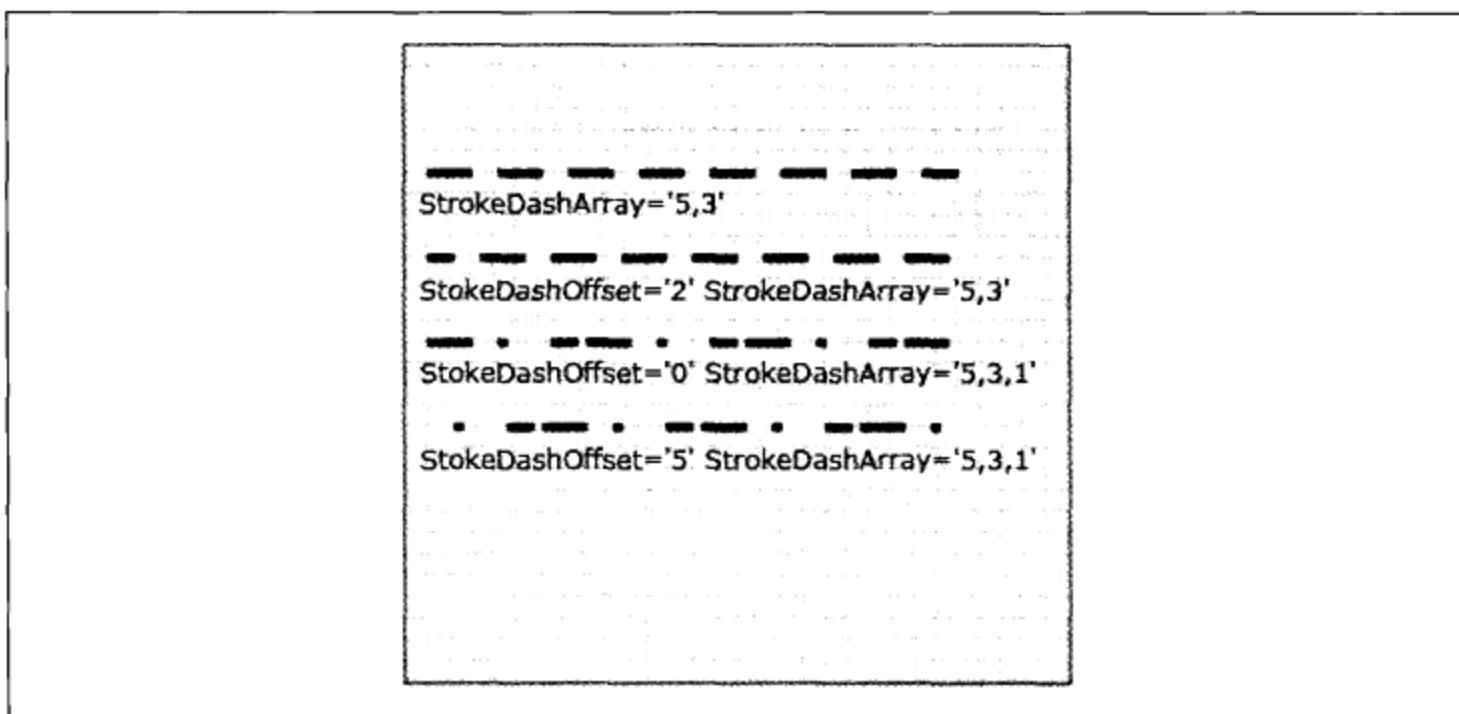


图 15-68 用 StrokeDashOffset 描述线型的示例效果

2. 线帽

线两端线帽可使用 `StrokeStartLineCap` 和 `StrokeEndLineCap` 两个属性值指定，WPF 提供了 4 种线帽，分别是 `Flat`（平线帽，默认）、`Square`（方帽）、`Round`（圆帽）和 `Triangle`（尖帽），如图 15-69 所示。

`Flat` 和 `Square` 类似，区别在于后者会将线段向两头延伸一半线宽作为方帽。`StrokeDashCap` 主要是控制线段内部每一小段的两端线帽，对线段的两端无效。只有非连续的线段才能看到其效果，如图 15-70 所示。

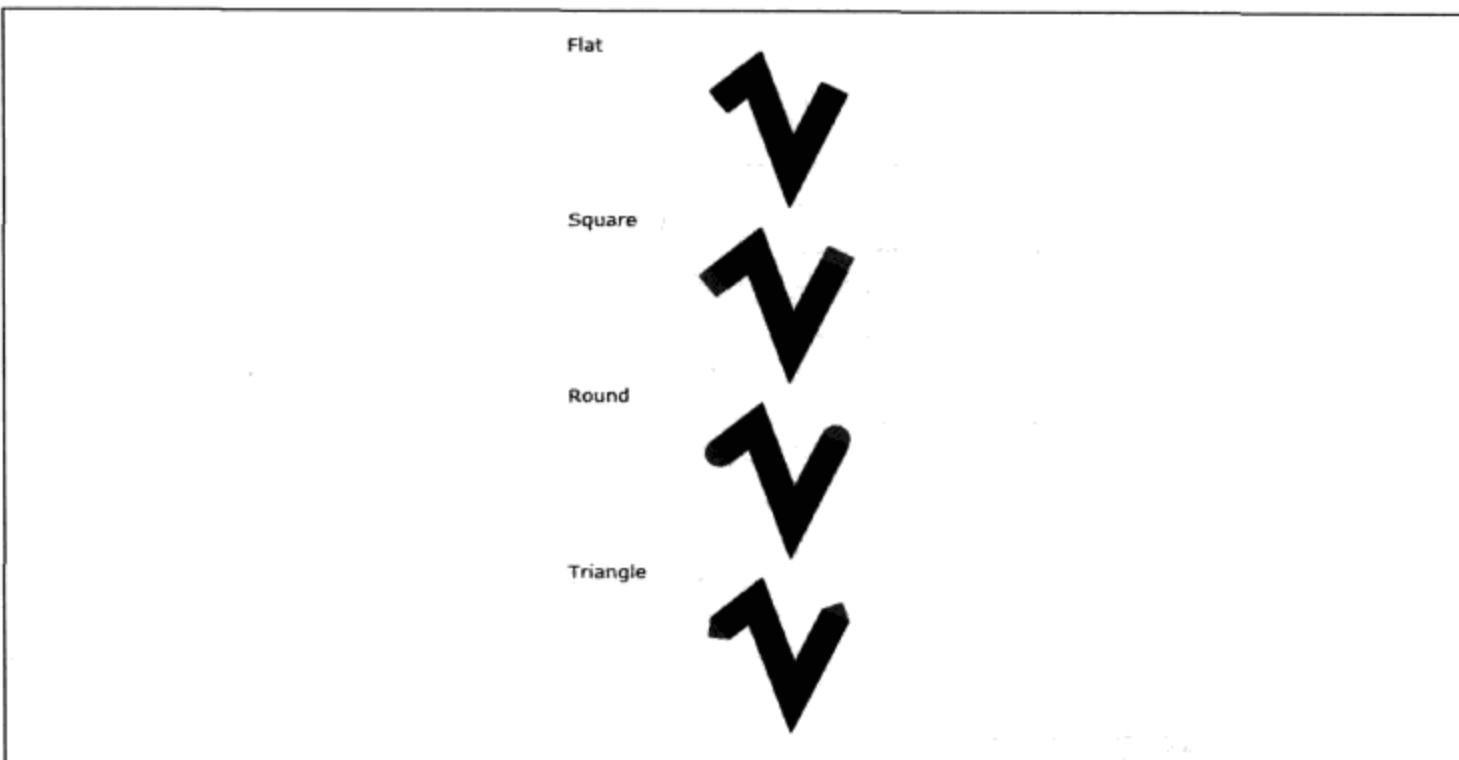


图 15-69 WPF 的 4 种线帽

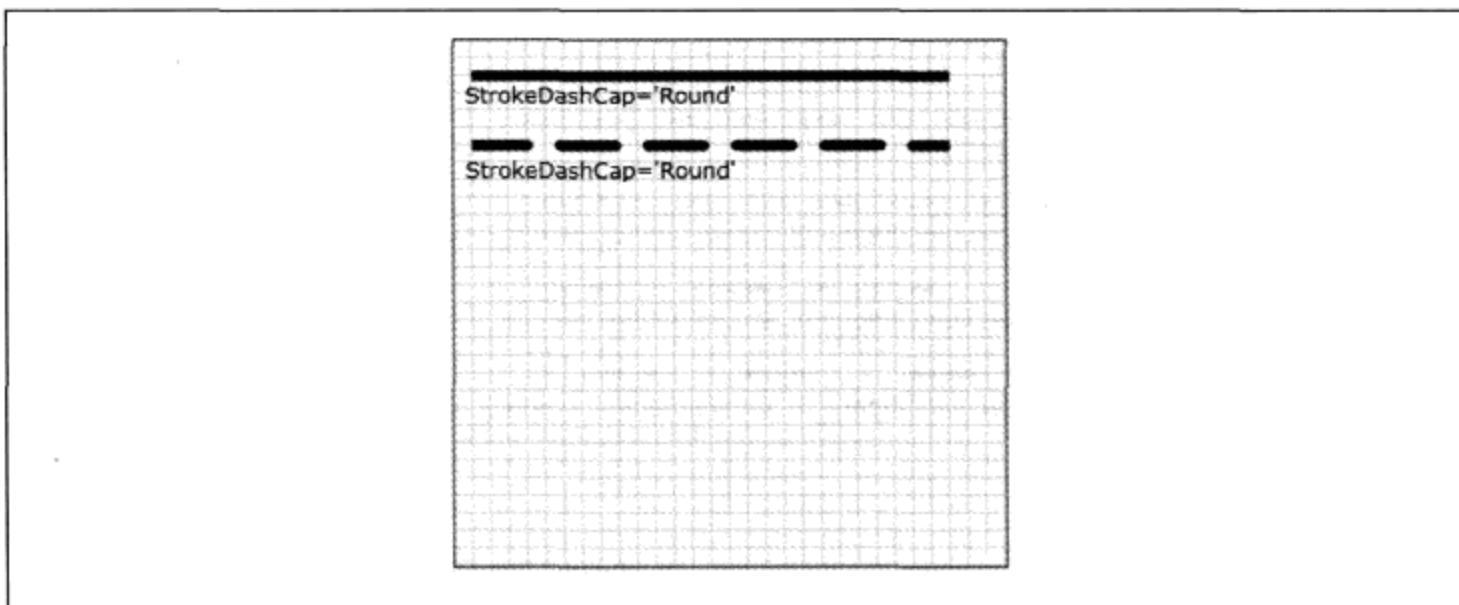


图 15-70 Flat 和 Square 线段处理效果

3. 线的连接

线的连接由 `StrokeLineJoin` 属性来控制，WPF 提供了 3 种连接方式，即 `Miter`（尖角连接，默认）、`Bevel`（平角连接）和 `Round`（圆角连接），如图 15-71 所示。

有时折线的夹角过小导致线的连接处会形成一个很长的尖角，为了防止这种情况，可以将 `StrokeLineJoin` 属性设置为 `Bevel` 或者 `Round`。也可以通过设置 `StrokeMiterLimit` 属性来控制尖角的长度，该值始终是大于或者等于 1 的正数。它描述的是尖角长度和线宽一半的最大比，即尖角长度（`MiterLength`）最长不能超过 $0.5 \times \text{StrokeThickness} \times \text{StrokeMiterLimit}$ 。`StrokeMiterLimit` 的默认值为 10（通过 Reflector 可以看到），如图 15-72 所示。

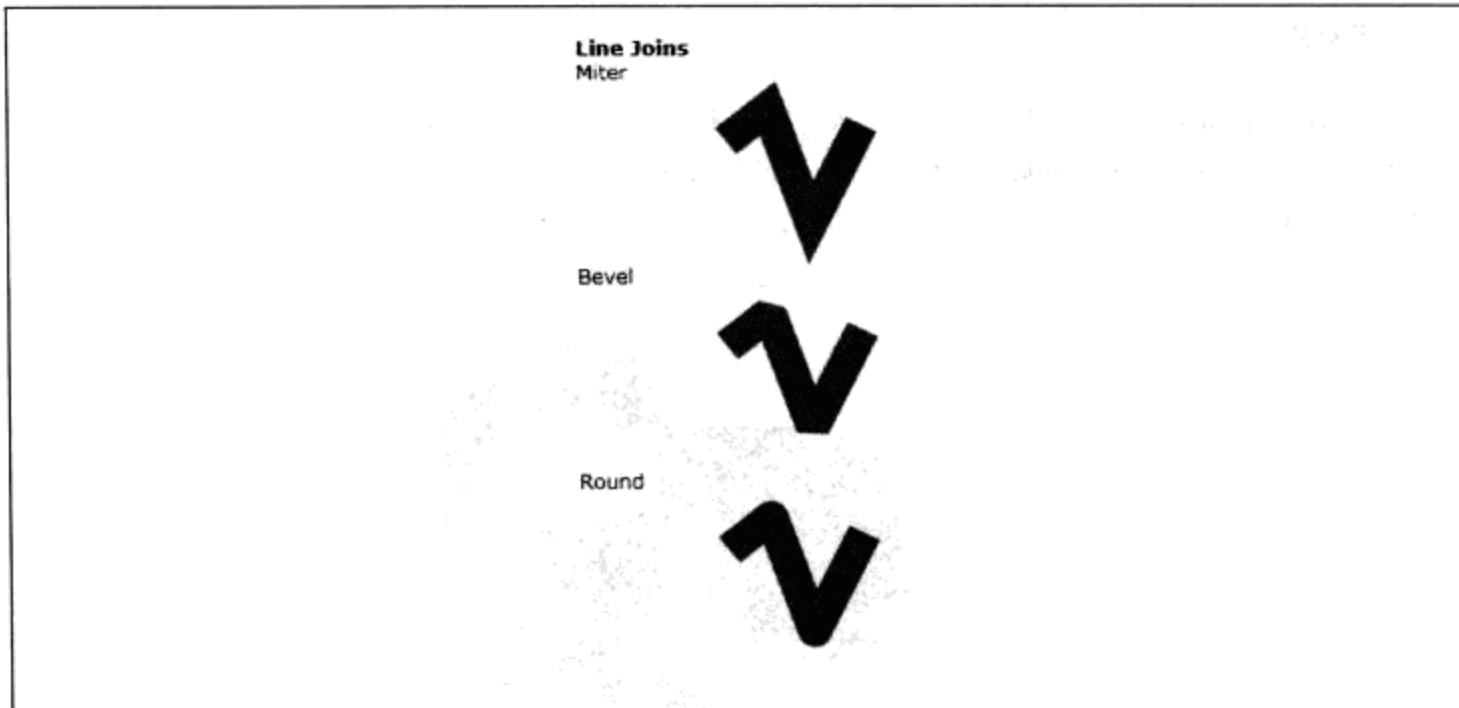


图 15-71 3 种线的连接方式

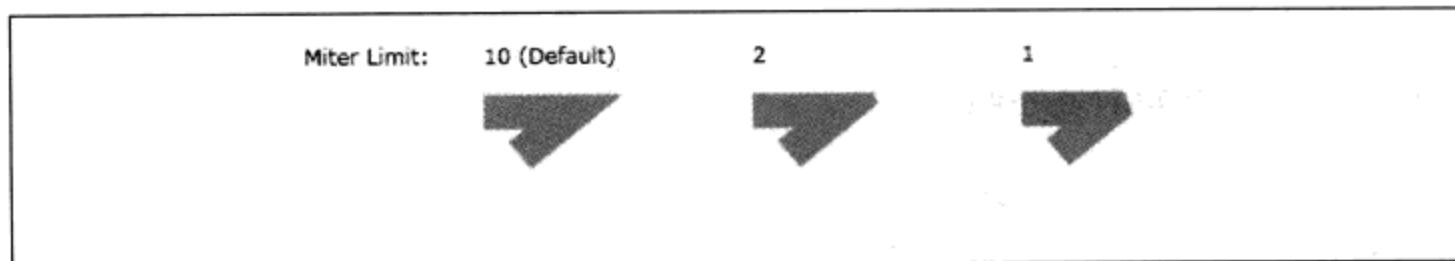


图 15-72 通过设置 `StrokeMiterLimit` 属性来控制尖角的长度效果

图 15-73 所示的正下方是 `StrokeMiterLimit` 设置为 2 的一个放大 5 倍的图, 可以看出 $\text{MiterLength} = 0.5 \times \text{StrokeThickness} \times 2 = \text{StrokeThickness}$ 。

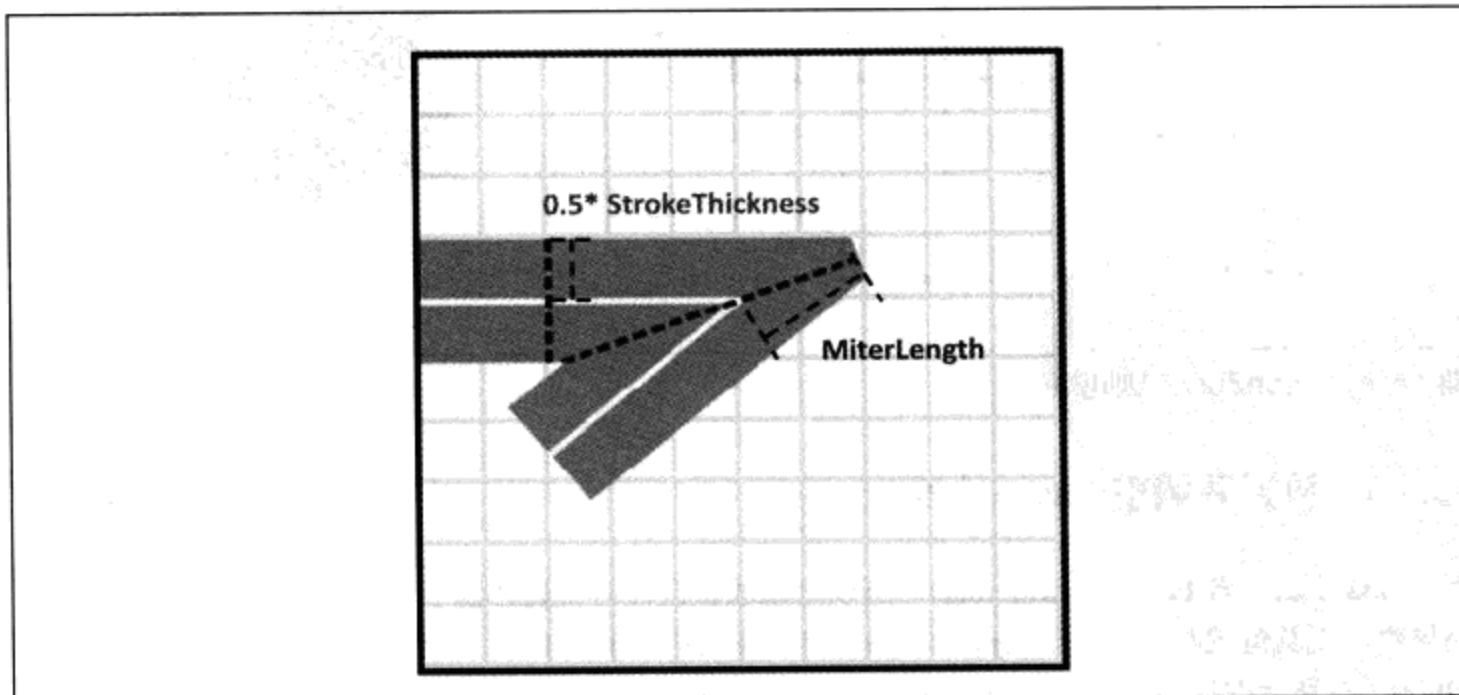


图 15-73 `StrokeMiterLimit` 设置为 2 的一个放大 5 倍的图

4. 填充规则

填充规则通过 FillRule 属性来控制，只有 PolyLine 和 Polygon 两种 Shape 元素才有该属性。WPF 提供了两种填充规则，即 EvenOdd 和 NonZero。前者指从一个点引出任意一条直线，如果该直线穿过的边数为奇数，则会填充该点；反之则不填充，如图 15-74 所示。

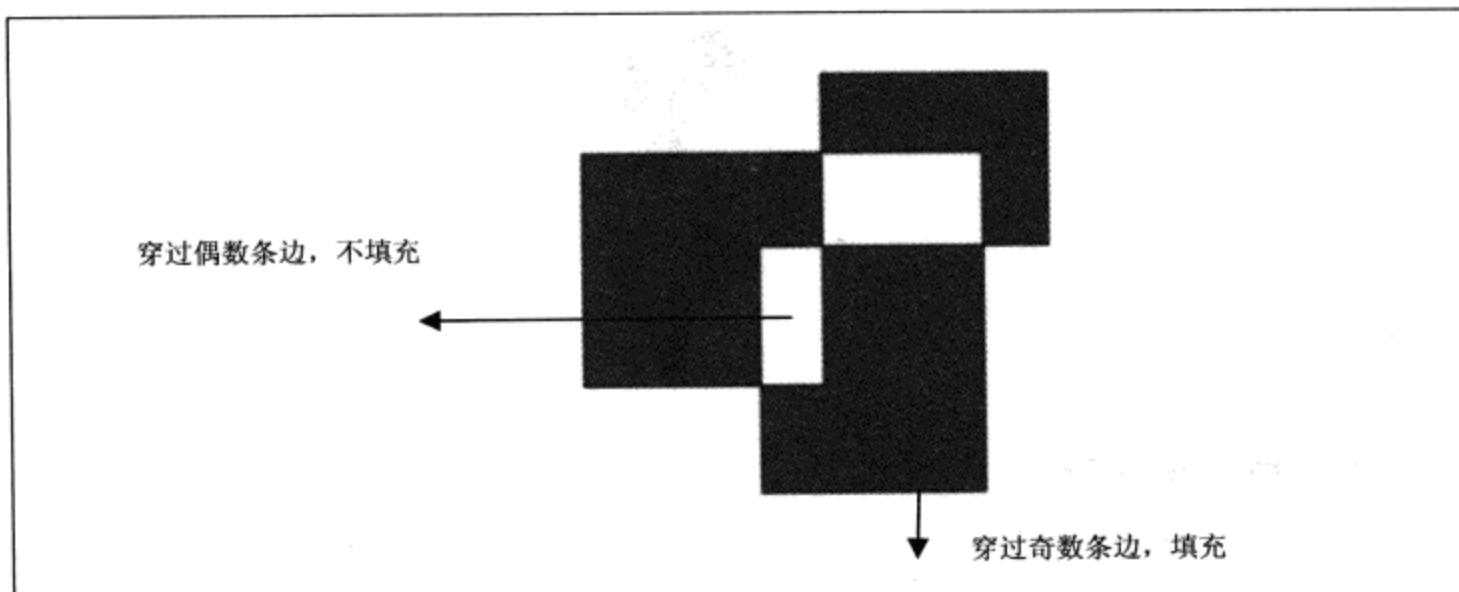


图 15-74 WPF 中的两种填充规则

后者遵循穿越奇数条边填充的规则，但是如果穿越的边数为偶数条，则要考虑多边形绘制的线的方向。我们假定多边形绘制的方向如图 15-75 所示。

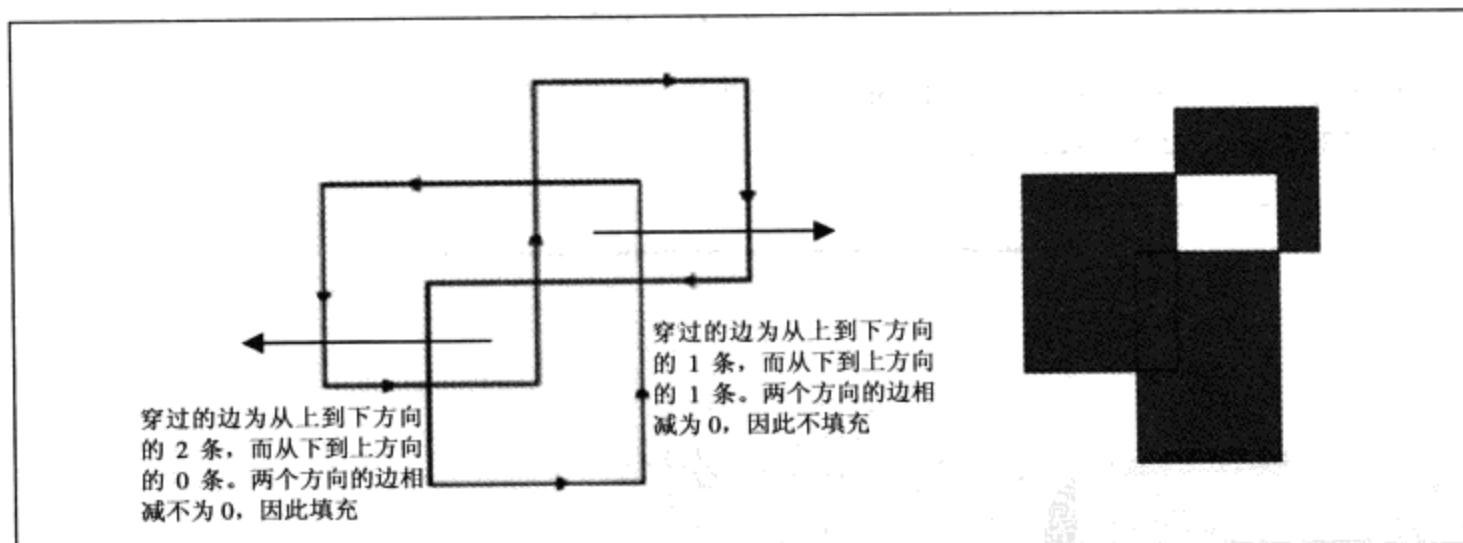


图 15-75 NonZero 规则填充

15.5.3 放置并调整 Shape 大小

由于 Shape 派生自 FrameworkElement，所以可以放置在任何布局容器中。如果 Shape 未显式设置宽高属性，则其位置和大小主要依赖于布局容器。我们将一个 Ellipse 放置在 Grid 的一个单元格中，Ellipse 将完全占据该单元格；如果放置在 StackPanel 中，需要显式设置其高度，其宽度默认为

StackPanel 的宽度。

Shape 提供一个 Stretch 属性来描述形状填充区域，其取值如下。

(1) None：表示图形不会自动伸展。如果未显式设置图形的长宽等属性（Width、MinWidth、Height 和 MinHeight），则不会显示图形。

(2) Fill：表示图形会自动伸展直到完全填充该区域，如果显式设置了图形的长宽属性，则这个值无效。

(3) Uniform：使图像自动伸展直到完全填充该区域，与 Fill 不同的是其长宽会成比例地伸展。如将一个 Ellipse 放置在 Grid 的一个长为 100，宽为 50 的单元中，则其会自动伸展成 50×50 的圆形。

(4) UniformToFill：与 UniformToFill 和 Uniform 不同，仍然是前面一个例子。这个 Ellipse 会变成 100×100 的圆形，超出的部分会被裁减掉。

图 15-76 所示为 Fill、Uniform 和 UniformToFill 的区别。

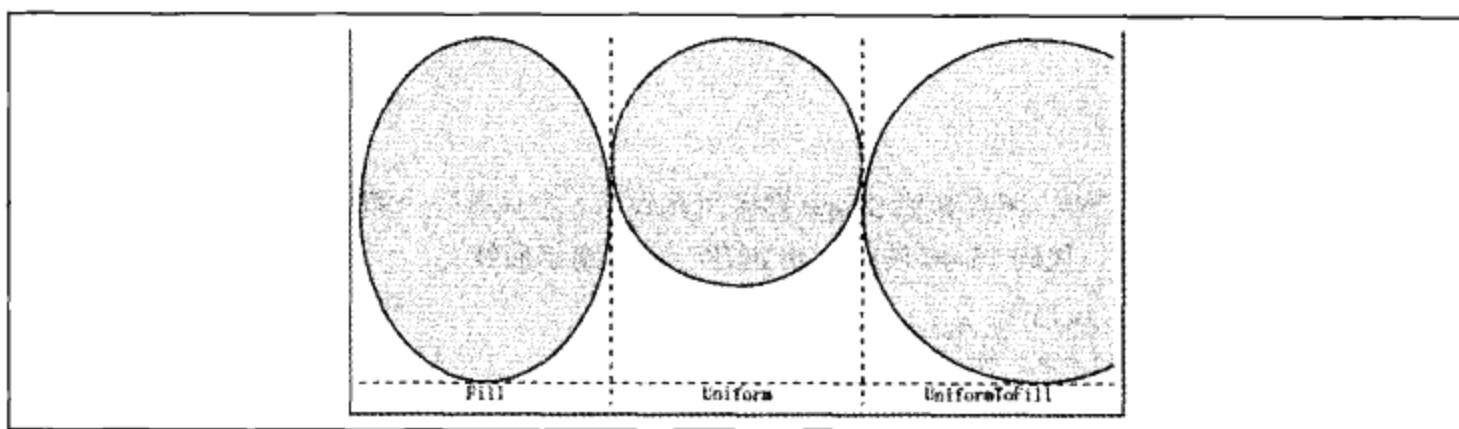


图 15-76 Fill、Uniform 和 UniformToFill 值的区别

一般绘制图形时常用的布局容器为 Canvas，如果需要图形随窗口大小变化而变化，则 Canvas 无法做到。WPF 为此提供了一个 ViewBox 类，它派生自 Decorator 类。我们将 Canvas 放置在 ViewBox 中，这样当窗口大小改变时 Canvas 中的图形会随之变化，如代码 15-39 所示。

```
<Viewbox Grid.Row="1" HorizontalAlignment="Left" MaxHeight="500">
    <Canvas Width="200" Height="150">
        <Ellipse Fill="Yellow" Stroke="Blue" Canvas.Left="10" Canvas.Top="50"
            Width="100" Height="50" HorizontalAlignment="Left"></Ellipse>
        <Rectangle Fill="Yellow" Stroke="Blue" Canvas.Left="30" Canvas.Top="40"
            Width="100" Height="50" HorizontalAlignment="Left"></Rectangle>
    </Canvas>
</Viewbox>
```

代码 15-39 Canvas 里的图形大小线宽随窗口大小的改变而变化

程序运行结果如图 15-77 所示。

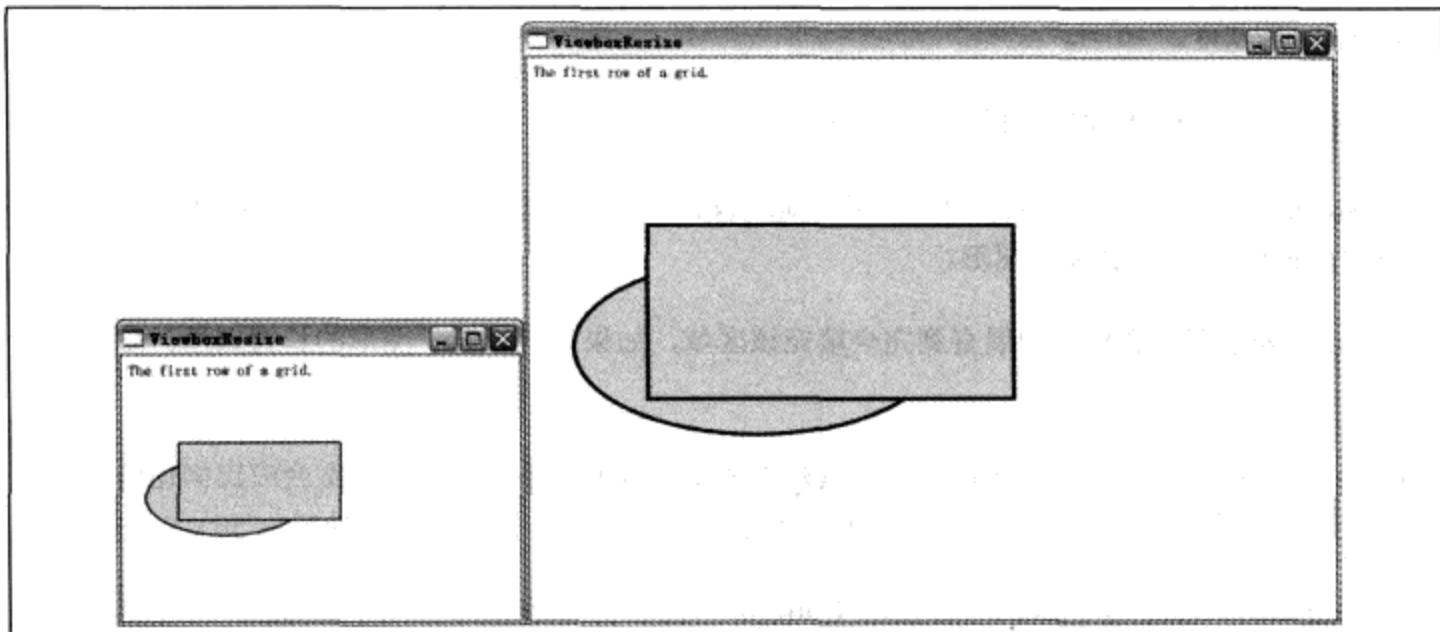


图 15-77 程序运行结果

Shape 也能实现几何变换。由于其派生自 FrameworkElement，因此可以通过 RenderTransform 及 LayoutTransform 属性实现几何变换。

15.5.4 Path

Path 是一种通用的图形元素，所有其他 Shape 均可用其表示。它只有一个简单的 Data 属性，可以设置为任何 Geometry 实例。代码 15-40 通过 Path 创建一条贝塞尔曲线。

```
<Canvas Height="400" Width="400">
  <Path Stroke="Black" StrokeThickness="1">
    <Path.Data>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigureCollection>
            <PathFigure StartPoint="10,100">
              <PathFigure.Segments>
                <PathSegmentCollection>
                  <BezierSegment Point1="100,0" Point2="200,200" Point3="300,100" />
                </PathSegmentCollection>
              </PathFigure.Segments>
            </PathFigure>
          </PathFigureCollection>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

代码 15-40 通过 Path 创建一条贝塞尔曲线

程序运行结果如图 15-78 所示。

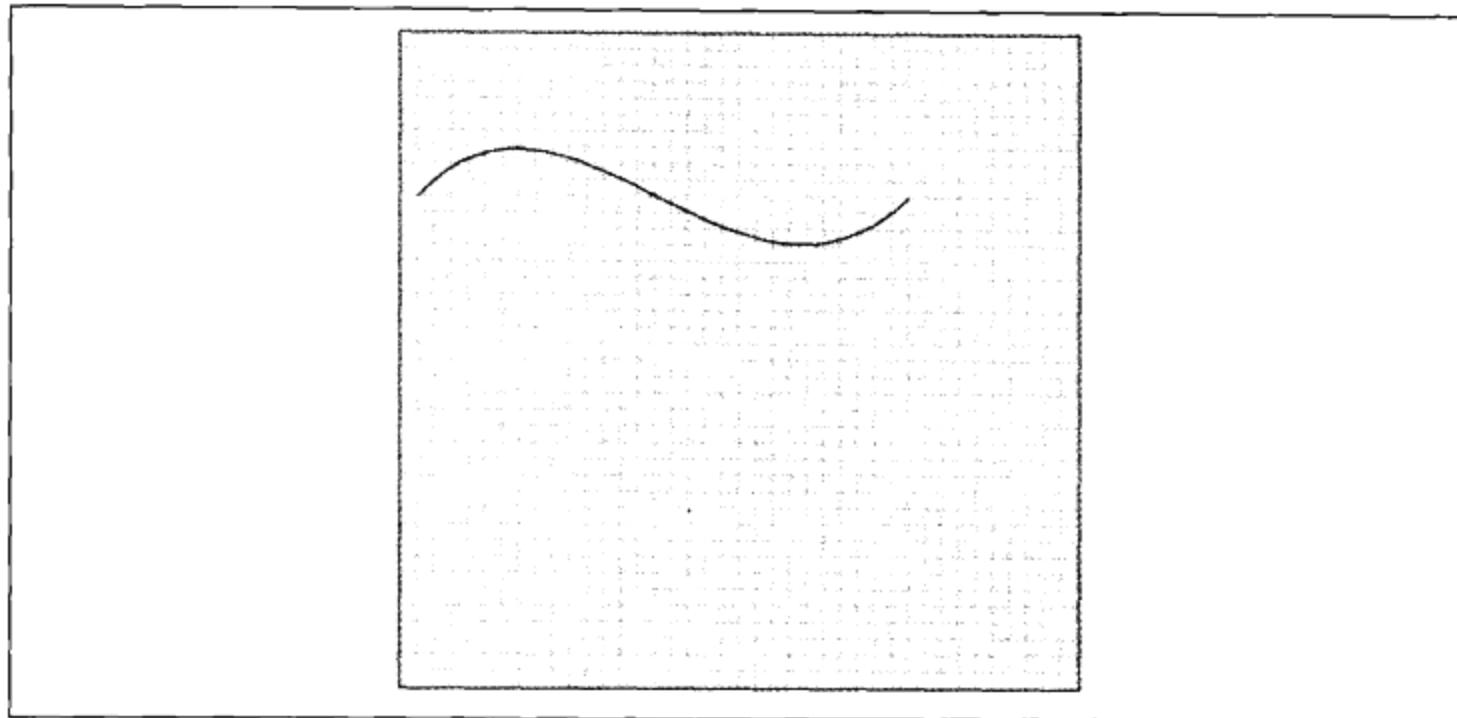


图 15-78 程序运行结果

15.6 Geometry (第三本书)

15.6.1 理解 Geometry

Path 中一个重要属性是 Data，为 Geometry 类型的对象。

在 WPF 二维图形中 Geometry 和 Shape 均描述二维图形，而且有非常类似的派生类，如表 15-7 所示。

表 15-7 Geometry 和 Shape 对比

图形	Geometry 派生类	Shape 派生类
矩形	RectangleGeometry	Rectangle
椭圆	EllipseGeometry	Ellipse
直线	LineGeometry	Line
路径	PathGeometry	Path

Shape 是高级图形元素，可以支持布局、鼠标键盘事件等；Geometry 只是对纯解析几何（如直线、矩形和椭圆等）的封装。我们可以用点和长度指定一个几何对象，但是 Geometry 无法绘制自身，必须使用其他类型（通常是 Path）设置填充画刷和画笔的相关属性来显示几何对象的外观。

Geometry 继承自 Freezable，有 6 个派生子类，其层次结构如图 15-79 所示。

在 WPF 中为 Geometry 类型的属性，如表 15-8 所示。

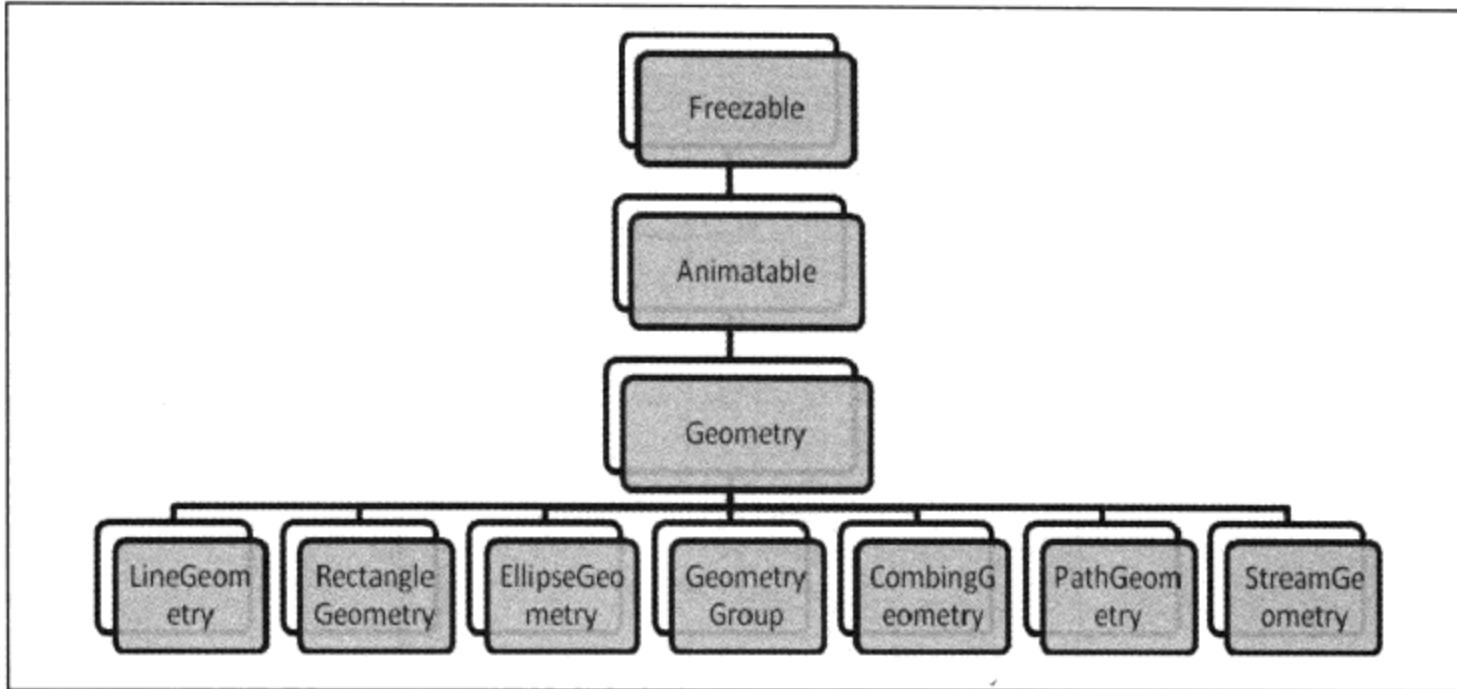


图 15-79 Geometry 类层次结构

表 15-8 为 Geometry 类型的属性

类型	属性
DoubleAnimationUsingPath	PathGeometry
DrawingGroup	ClipGeometry
UIElement	Clip
GeometryDrawing	Geometry
Path	Data

从表中我们可以归纳出 Geometry 主要用在以下 3 个方面。

- (1) 绘制图形：如在 Path 中设置 Data 属性绘制图形或者在 GeometryDrawing 中设置 Geometry 属性。
- (2) 定义裁减区域：如设置 UIElement 的 Clip 属性或者 DrawingGroup 的 ClipGeometry 属性。
- (3) 在动画中设置运动路径：如在 DoubleAnimationUsingPath 类中设置 PathGeometry 属性。

15.6.2 简单的 Geometry 类型

简单的 Geometry 类型如下。

- (1) LineGeometry：通过 StartPoint 和 EndPoint 属性来定义一条直线。
- (2) RectangleGeometry：通过 Rect 属性来定义矩形大小，如果矩形需要圆角，则设置 RadiusX 和 RadiusY 属性。

(3) EllipseGeometry：通过 RadiusX 和 RadiusY 设置 X 和 Y 轴的半径，通过 Center 属性设置椭圆的中心点。

通过 Path 绘制 LineGeometry、RectangleGeometry 和 EllipseGeometry 的示例为代码 15-41。

```
<Canvas Height="400" Width="400">
    <Path Stroke="Black" StrokeThickness="1">
        <Path.Data>
            <LineGeometry StartPoint="20,20" EndPoint="60,90"/>
        </Path.Data>
    </Path>
    <TextBlock Text="通过 Path 绘制 LineGeometry" Height="23"
Canvas.Left="125" Canvas.Top="37" Width="174" />
    <Path Stroke="Black" StrokeThickness="1" Fill="Yellow" >
        <Path.Data>
            <EllipseGeometry RadiusX="20" RadiusY="30" Center="45,145"/>
        </Path.Data>
    </Path>
    <TextBlock Text="通过 Path 绘制 EllipseGeometry" Height="23"
Canvas.Left="125" Canvas.Top="137" Width="174" />
    <Path Stroke="Black" StrokeThickness="1" Fill="Yellow" >
        <Path.Data>
            <RectangleGeometry RadiusX="5" RadiusY="5"
Rect="25,220,50,50"/>
        </Path.Data>
    </Path>
    <TextBlock Text="通过 Path 绘制 RectangleGeometry" Height="26"
Canvas.Left="125" Canvas.Top="237" Width="206" />
</Canvas>
```

代码 15-41 通过 Path 绘制 LineGeometry、RectangleGeometry 和 EllipseGeometry

程序运行结果如图 15-80 所示。

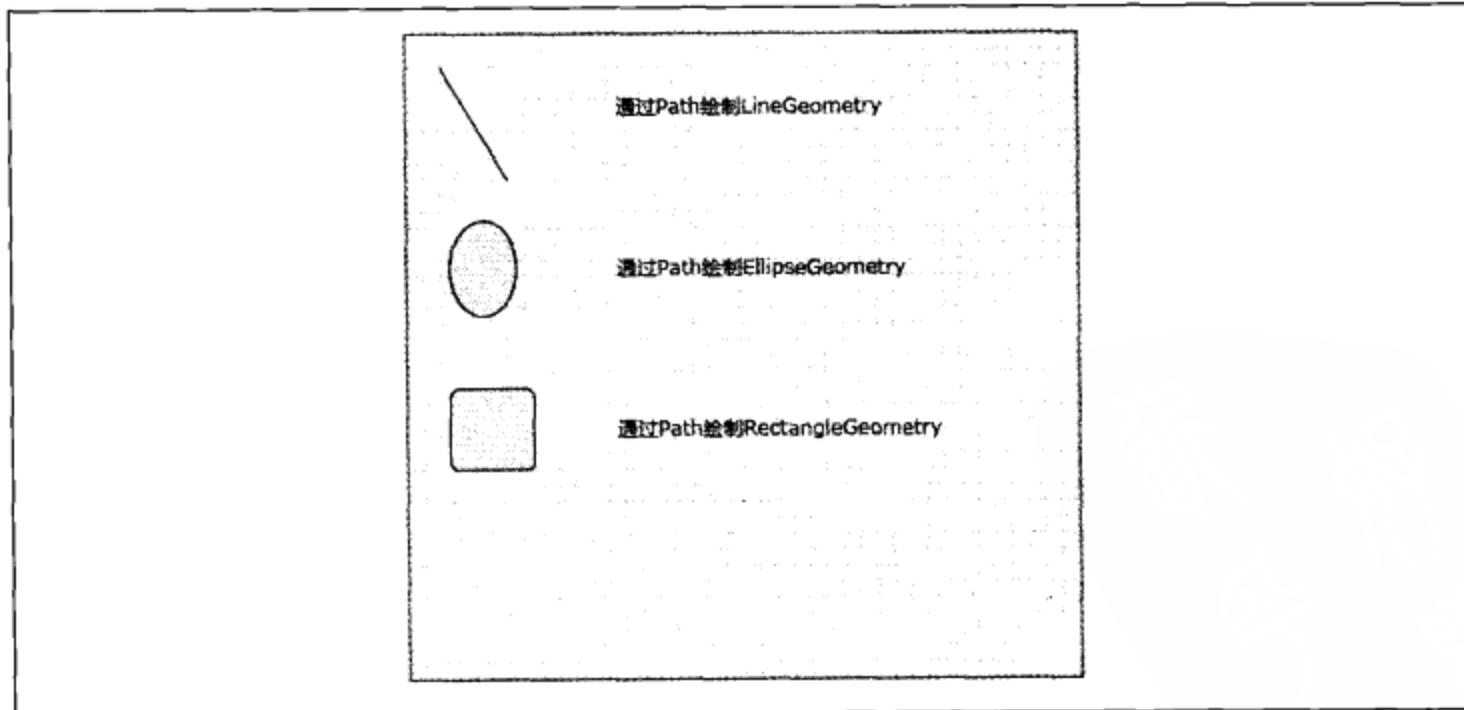


图 15-80 程序运行结果

15.6.3 GeometryGroup 和 CombineGeometry

Path 的一个缺陷是 Data 属性只能包含一个 Geometry 类型的对象，解决这个问题的方法是使用 GeometryGroup。GeometryGroup 继承自 Geometry，但是有一个 children 属性，该属性是 Geometry 类型的集合。因此 GeometryGroup 可以包含多个 Geometry，如代码 15-42 所示。

```
<Canvas>
    <!--每一个 Path 是一个矩形-->
    <Path Fill="Gold" Stroke="Red" StrokeThickness="3">
        <Path.Data>
            <RectangleGeometry Rect="96 96 192 192" />
        </Path.Data>
    </Path>
    <Path Fill="Gold" Stroke="Red" StrokeThickness="3">
        <Path.Data>
            <RectangleGeometry Rect="192 192 192 192" />
        </Path.Data>
    </Path>
    <!--一个 Path 包含了两个矩形-->
    <Path Fill="Gold" Stroke="Red" StrokeThickness="3">
        <Path.Data>
            <GeometryGroup>
                <RectangleGeometry Rect="480 96 192 192" />
                <RectangleGeometry Rect="576 192 192 192" />
            </GeometryGroup>
        </Path.Data>
    </Path>
</Canvas>
```

代码 15-42 GeometryGroup 和 CombineGeometry

程序运行结果如图 15-81 所示。

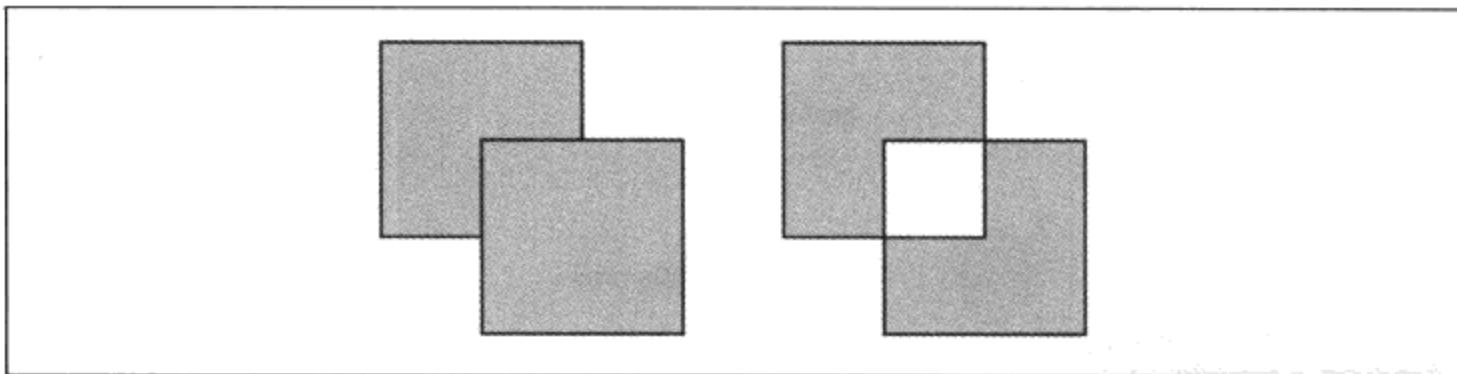


图 15-81 程序运行结果

左侧是两个 Path，每个 Path 包含的一个 RectangleGeometry 重叠在一起；右侧是一个 Path，包含的两个 RectangleGeometry 重叠在一起。两个矩形重叠的部分没有填充，这是因为 GeometryGroup 定义了自己的 FillRule 属性，默认值是 EvenOdd。

与 GeometryGroup 容易混淆的一个 Geometry 类型是 CombineGeometry，二者的区别如下。

(1) CombinedGeometry 没有 Children 属性，只有 Geometry1 和 Geometry2 属性。即与 CombinedGeometry 不同，它是两个且只有两个 Geometry 的组合。

(2) CombinedGeometry 没有 FillRule 属性，而多了一个 GeometryCombineMode 属性，用于指定几何体组合模式。WPF 提供了 4 种组合模式，即联合（Union）、相交（Intersect）、异或（Xor）和排除（Exclude）。

图 15-82 所示为使用 GeometryGroup 创建的两个椭圆重叠的形状，填充规则为 NonZero。

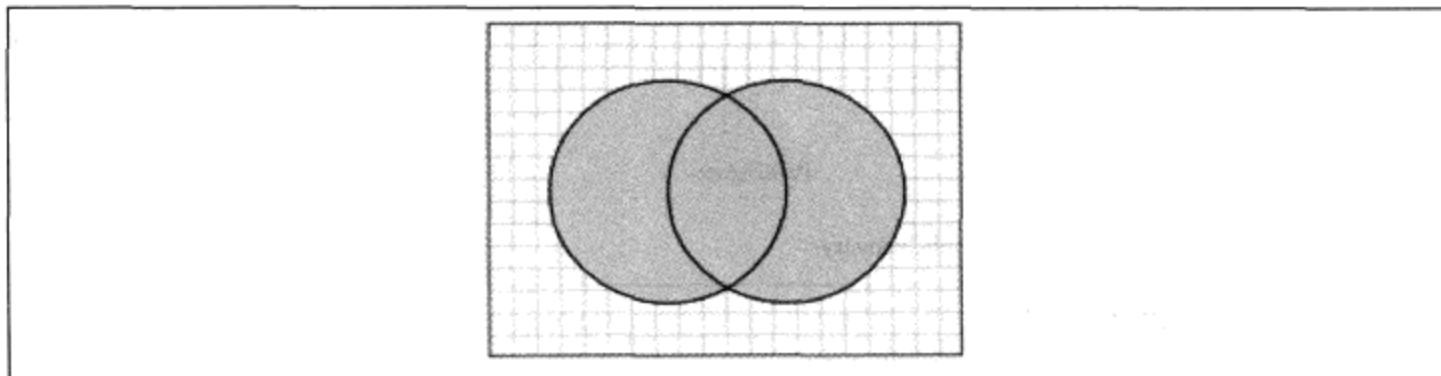


图 15-82 使用 GeometryGroup 创建两个重叠椭圆

使用 CombinedGeometry 组合这两个椭圆，并使用不同的组合模式，如图 15-83 所示。

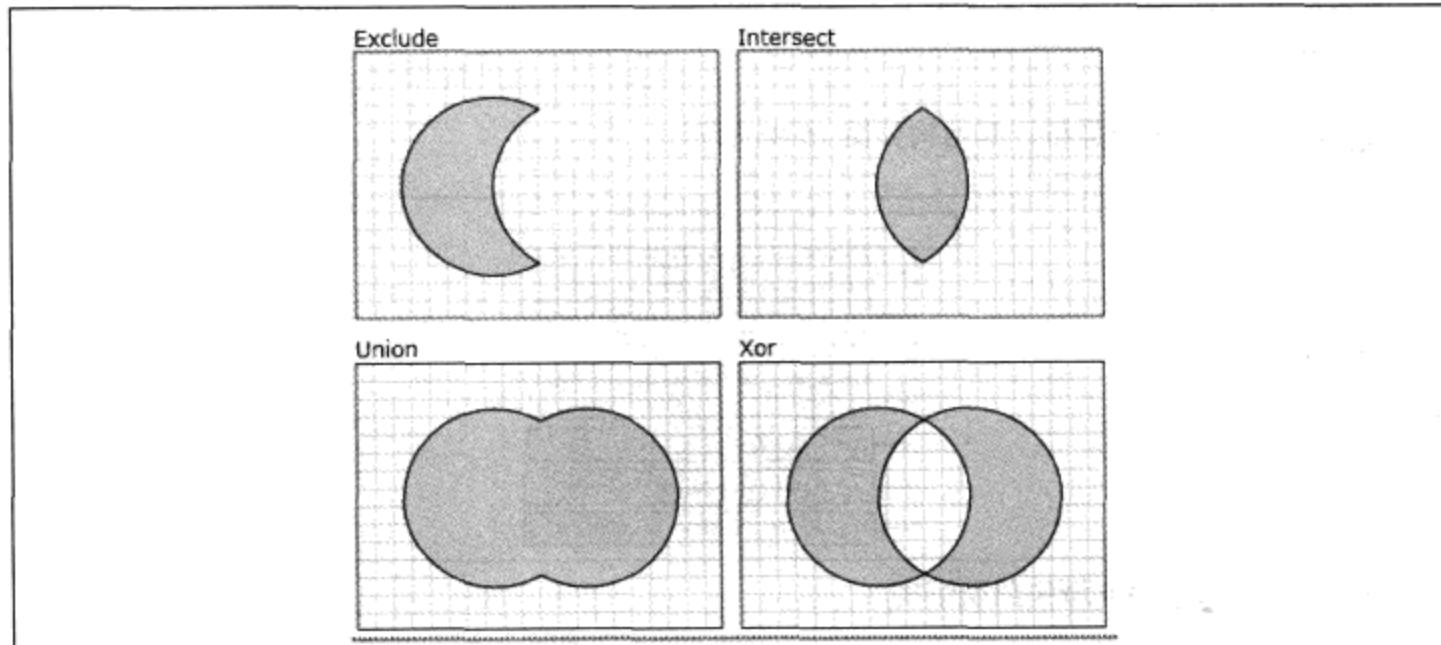


图 15-83 使用 CombinedGeometry 组合两个椭圆

15.6.4 PathGeometry 和 StreamGeometry

1. PathGeometry

一个 PathGeometry 是一系列 PathFigure 对象的集合，每个 PathFigure 表示一组连接的线段。这些线段可以是直线或曲线，即每个 PathFigure 是一系列的 PathSegment 对象的集合。PathSegment 表示一个线段，这个线段可以是直线或曲线。

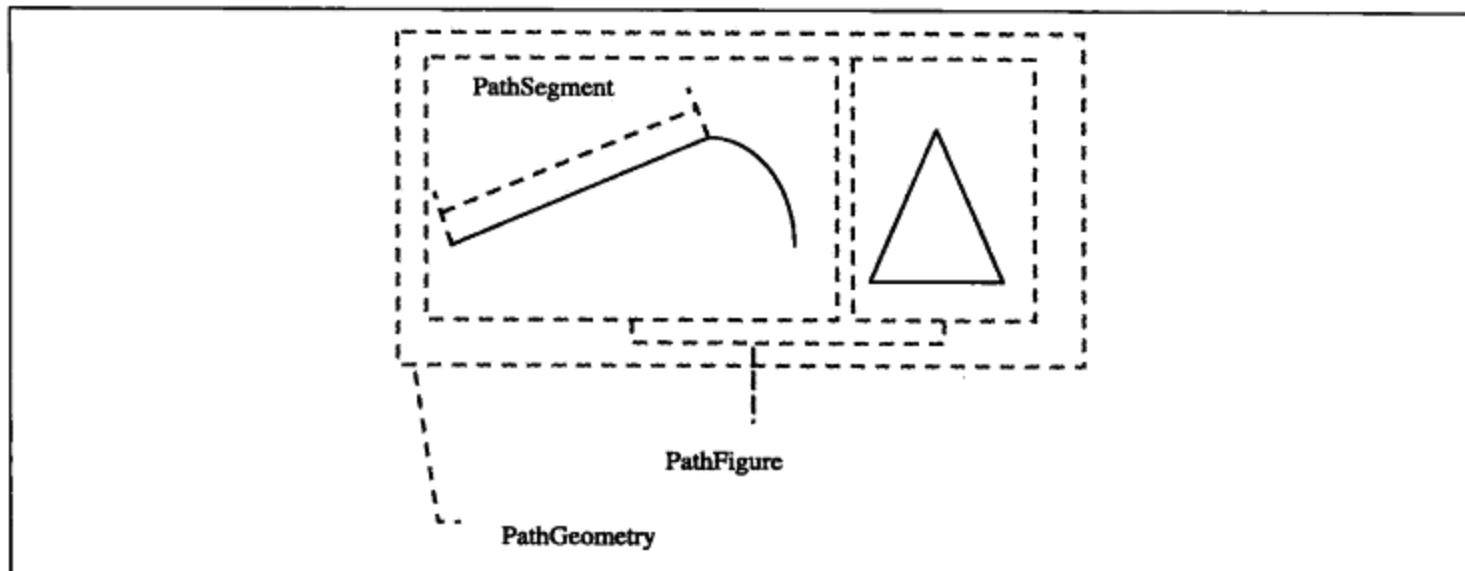


图 15-84 PathGeometry 示意

PathGeometry 的 FillRule 属性用来控制如何填充路径上的图形, PathFigure 定义了两个 Boolean 属性, 即 IsClosed 和 IsFilled 属性。前者的默认值为 false, 如果设置为 true, 则图形闭合, 反之则不闭合; 后者用来控制是否用画刷填充内部区域, 默认值为 True。PathFigure 是一系列相互连接的直线和曲线, 必须从某个点开始, 因此定义了一个 StartPoint 属性。另外定义了一个 Segment 的集合, 名为“Segments”。

PathSegment 派生了多个类型, 其结构如图 15-85 所示。

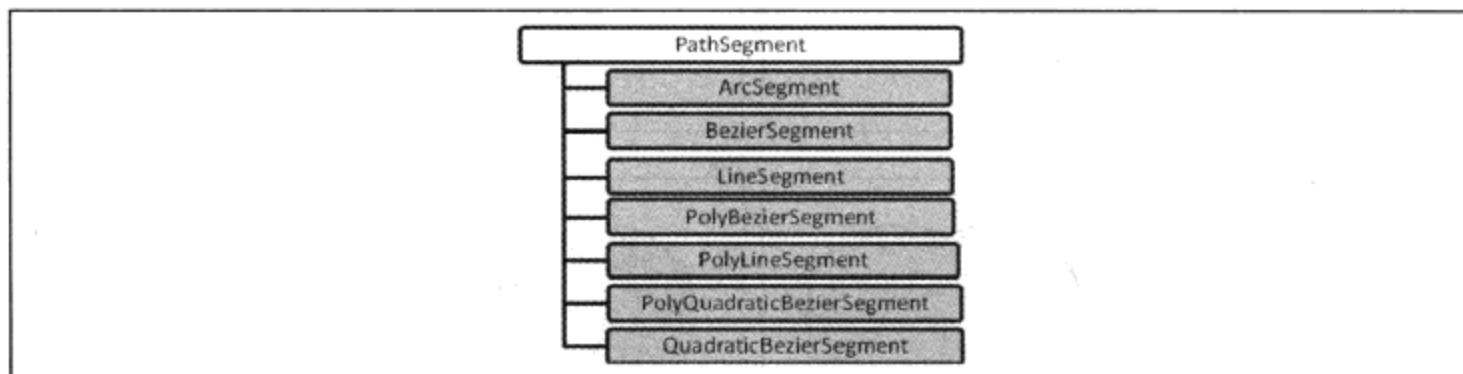


图 15-85 PathSegment 派生类型结构

其中最为简单的 LineSegment 对象表示一段直线, 由于 PathFigure 包含了一个起点属性(StartPoint), 因此该对象只有一个 Point 属性用于设置线的结束点。代码 15-43 用 4 个 LineSegment 对象绘制一个五角星。

```

<Canvas>
    <Path Fill="Aqua" Stroke="Maroon" StrokeThickness="3">
        <Path.Data>
            <PathGeometry>
                <PathFigure StartPoint="144 72">
                    <LineSegment Point="200 246" />
                    <LineSegment Point="53 138" />
                    <LineSegment Point="235 138" />
                    <LineSegment Point="88 246" />
                </PathFigure>
            </PathGeometry>
        </Path>
    </Canvas>

```

```
</Path.Data>
</Path>
</Canvas>
```

代码 15-43 用 4 个 LineSegment 对象绘制一个五角星

程序运行结果如图 15-86 所示。

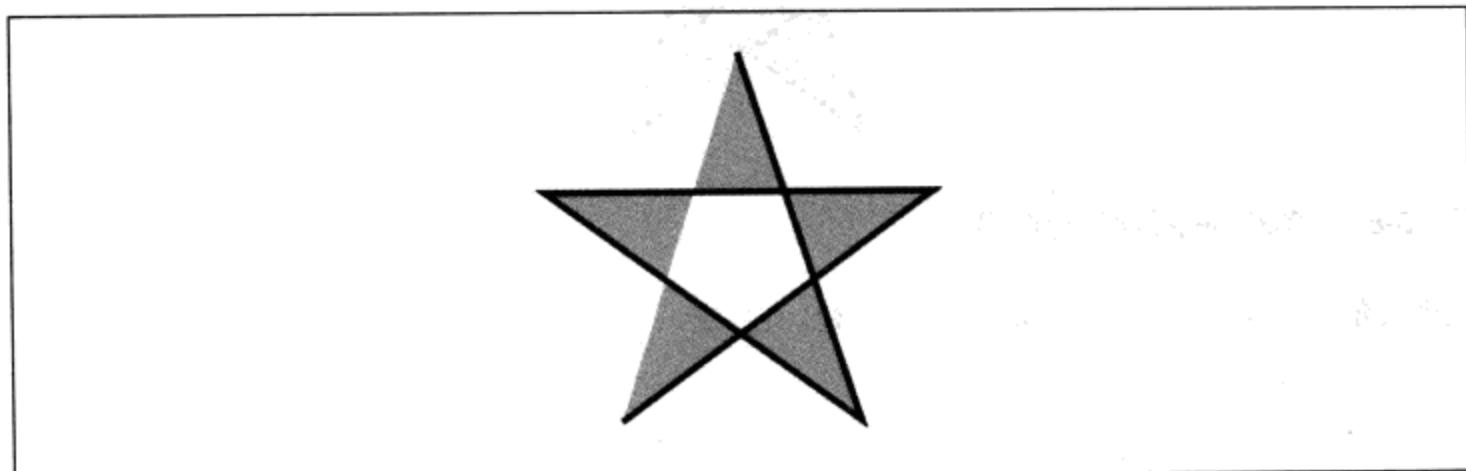


图 15-86 程序运行结果

可以看到该 PathFigure 没有闭合，解决的方法是添加一条 LineSegment，也可以将 PathFigure 的 IsClosed 属性设置为 true：

```
<PathFigure StartPoint="144 72" IsClosed="True">
```

修改后的程序运行结果如图 15-87 所示。

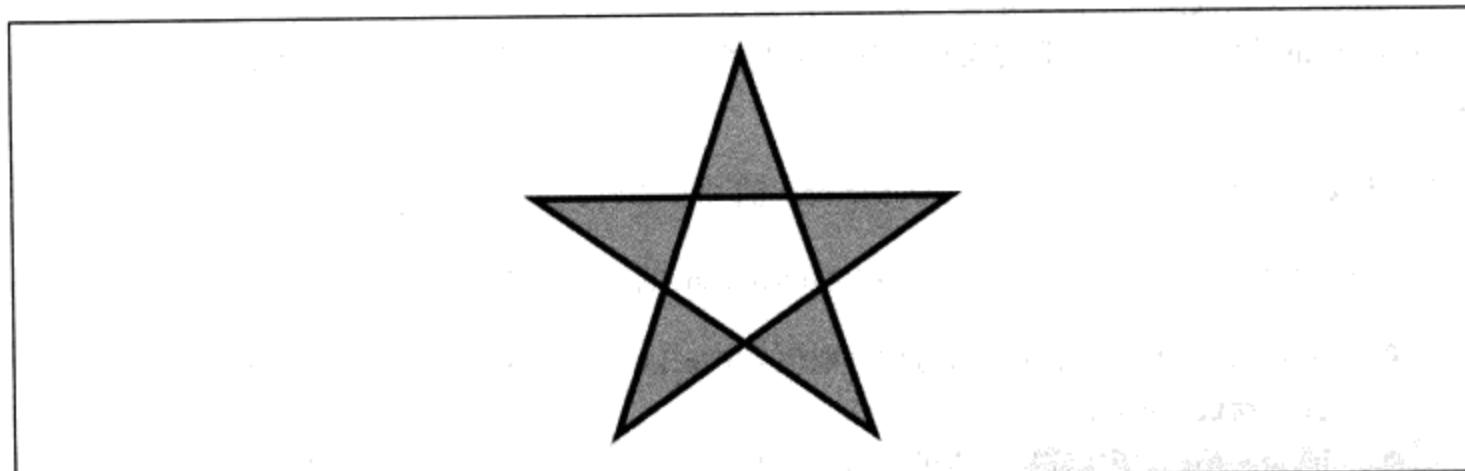


图 15-87 修改后的程序运行结果

此外 PathGeometry 的填充规则默认是 EvenOdd，也可以将其修改为 NonZero，这样也会填充五角星的中间：

```
<PathGeometry FillRule="Nonzero">
```

修改后的程序运行结果如图 15-88 所示。

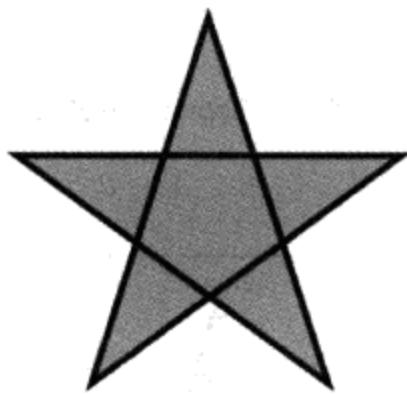


图 15-88 修改后的程序运行结果

这个五角星还可以用 PolyLineSegment 来创建，它表示一个多点折线，如代码 15-44 所示。

```
<Canvas>
    <Path Fill="Aqua" Stroke="Maroon" StrokeThickness="3">
        <Path.Data>
            <PathGeometry>
                <PathFigure StartPoint="144 72" >
                    <PolyLineSegment Points="200 246, 53 138, 235 138, 88 246" />
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>
</Canvas>
```

代码 15-44 用 PolyLineSegment 来创建五角星

ArcSegment 类型根据一个圆周定义一条曲线，类似 LineSegment。但其也只定义一个点，并且从上一个 Segment 的最后一个点到定义点绘制一条弧线。由于要绘制弧线，因此 ArcSegment 需要如下数据。

- (1) 弧线所在椭圆的两个半径，由 ArcSegment 的 Size 属性指定。
- (2) 连接两个点的弧线有 4 种可能性，由 ArcSegment 的 SweepDirection 和 IsLargeArc 属性指定。
 - SweepDirection：指定弧线方向，取值为 Clockwise（顺时针）和 Counterclockwise（逆时针，默认）。
 - IsLargeArc：指定连接两点的是否是大圆弧，为 true，则大圆弧；否则不是（默认）。

如图 15-89 所示的 4 条弧线的起始点和结束点，以及所在的圆弧大小相同，不同的是 SweepDirection 和 IsLargeArc 属性。

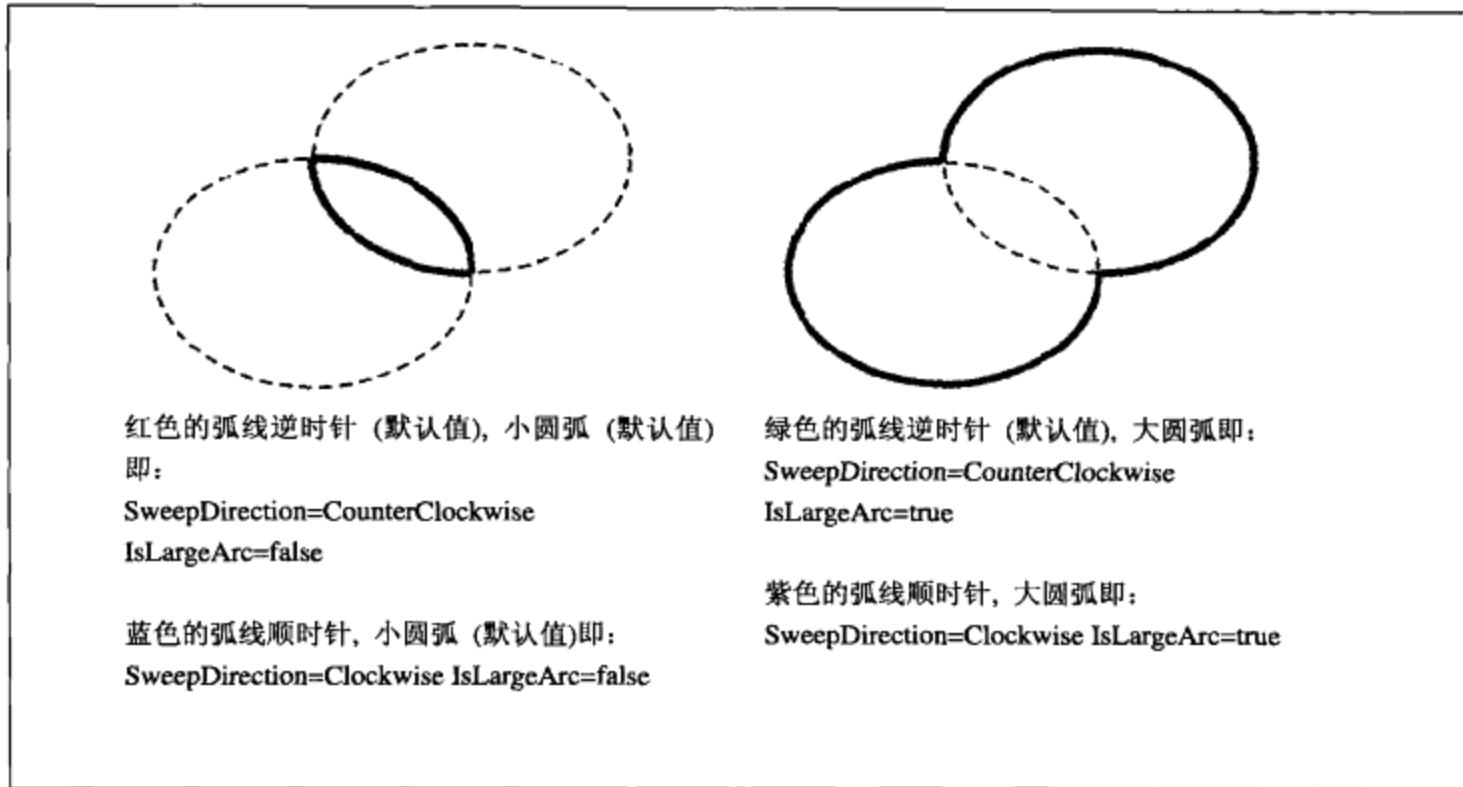


图 15-89 不同 SweepDirection 和 IsLargeArc 属性的圆弧效果

ArcSegment 定义了一个 RotationAngle 属性用来指示椭圆顺时针旋转的角度, 如将上例中的绿色圆弧 RotationAngle 设置为 45 度后的效果如图 15-90 所示。

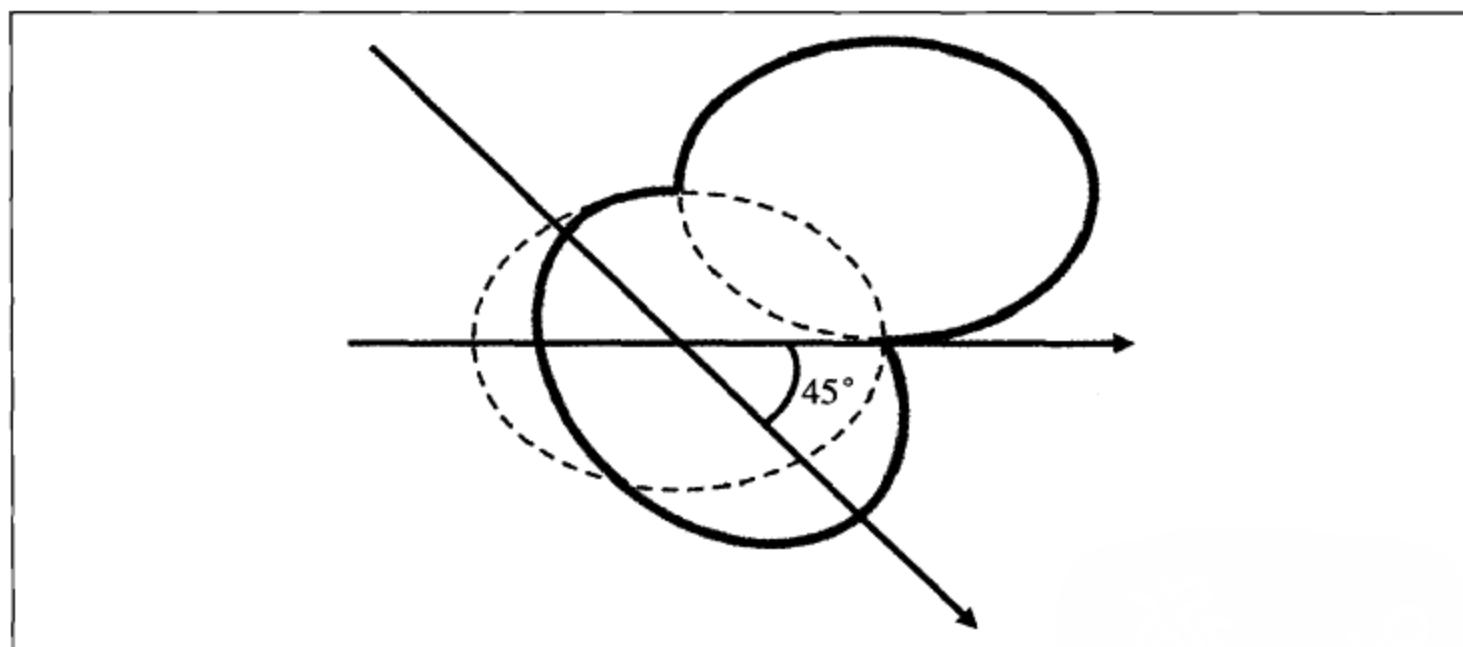


图 15-90 将上例的绿色圆弧 RotationAngle 设置为 45 度后的效果

在工程绘图中为常用的是贝塞尔曲线, WPF 提供了常规的三次贝塞尔曲线 (BezierSegment) 和一个计算速度更快的二次贝塞尔曲线 (QuadraticBezierSegment)。BezierSegment 由 4 个点定义 (P_0, P_1, P_2, P_3) , 曲线从 P_0 开始, 结束于 P_3 点, P_1 和 P_2 称为“控制点”。在 P_0 点的曲线正切于 P_0 和 P_1 的连线, 并在同侧; 在 P_3 点的曲线正切于 P_2 和 P_3 的连线, 并在同侧, 如图 15-91 所示。

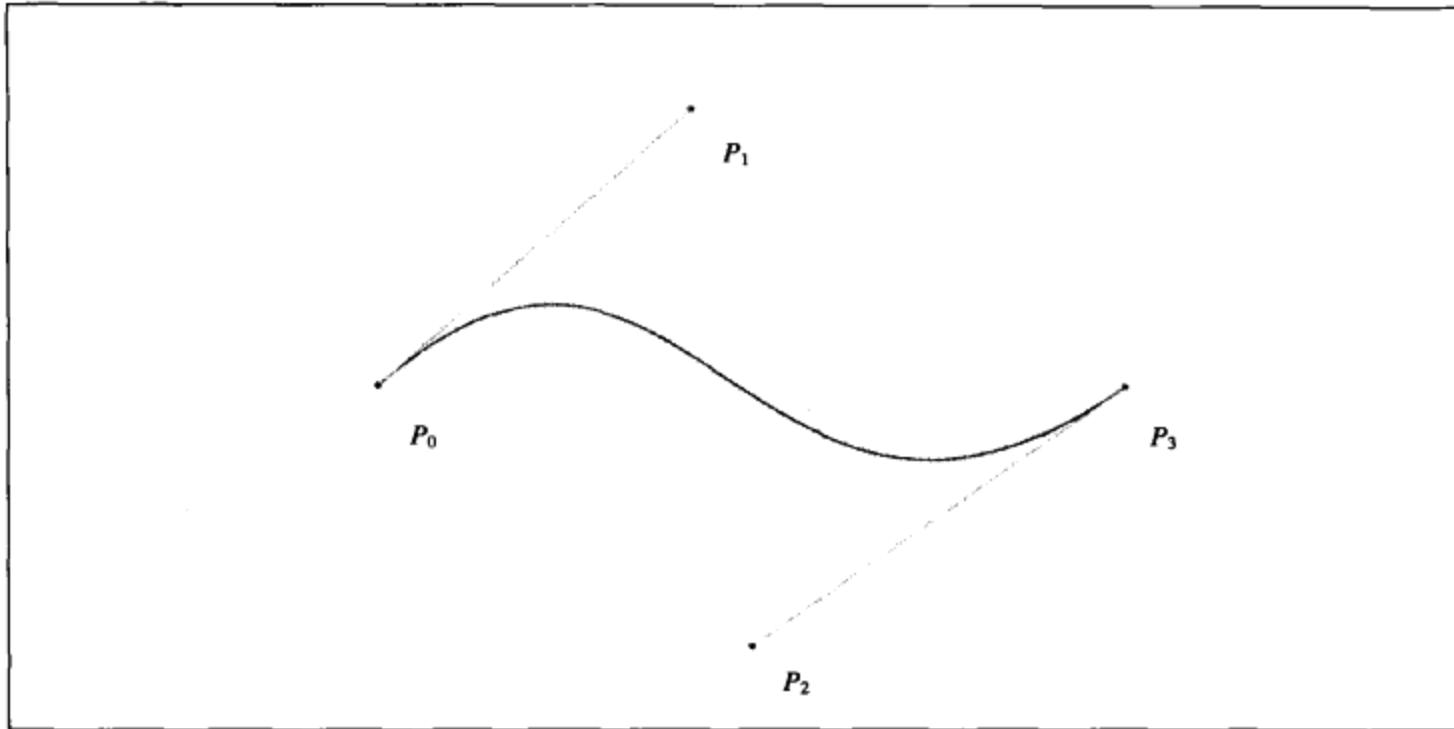


图 15-91 三次贝塞尔曲线

QuadraticBezierSegment 只有一个控制点，如图 15-92 所示。

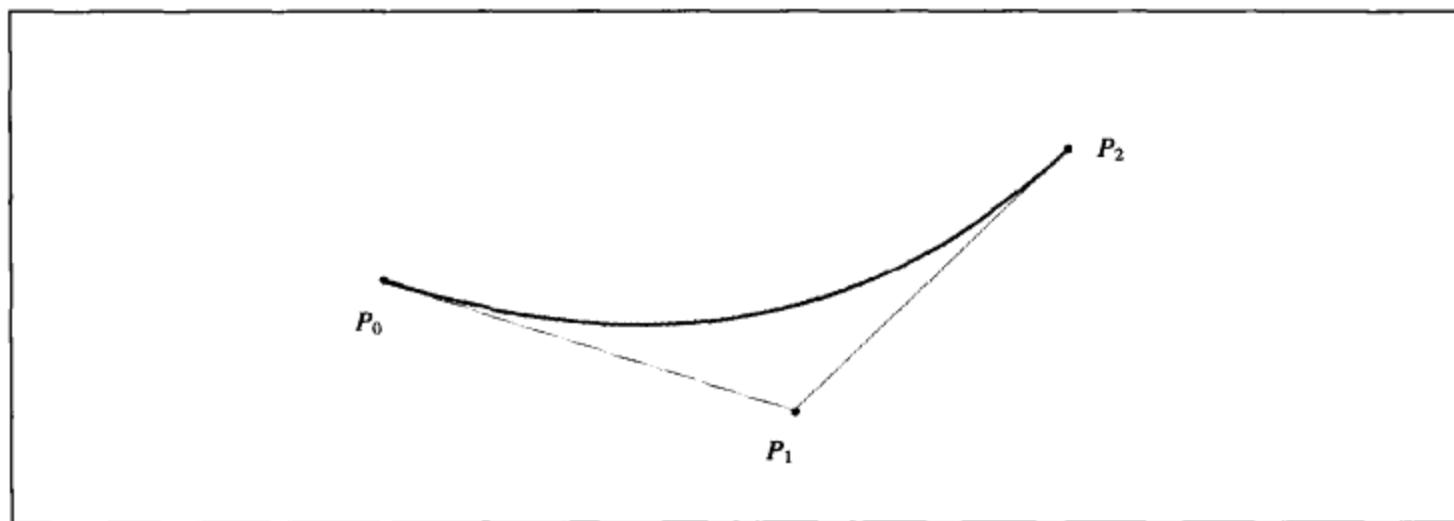


图 15-92 二次贝塞尔曲线

可以定义多个相互连接的三次贝塞尔曲线（PolyBezierSegment）和二次贝塞尔曲线（PolyQuadraticBezierSegment）。PolyBezierSegment 和 PolyQuadraticBezierSegment 都有一个 Points 属性，虽然它们对点的个数没有限制，但是 PolyBezierSegment 点的个数应该为 3 的倍数；PolyQuadraticBezierSegment 应该为 2 的倍数。以 PolyBezierSegment 为例，PathFigure 的 StartPoint 为第 1 个端点，PolyBezierSegment 的第 1 个点和第 2 个点是控制点；第 3 个点是结束点，也是第 2 段贝塞尔曲线的起始点；第 4 个点和第 5 个点是第 2 段贝塞尔曲线的控制点，如此反复循环，如代码 15-45 所示。

```
<Canvas>
    <Path Stroke="Black" StrokeThickness="1" Fill="Black" >
        <Path.Data>
            <GeometryGroup>
```

```
<PathGeometry>
    <PathFigure x:Name="fig"
        StartPoint="50 150" IsFilled="False" >
        <PolyBezierSegment Points="0,0 200,0 300,100 300,0 400,0
600,100" />
    </PathFigure>
</PathGeometry>
</GeometryGroup>
</Path.Data>
</Path>
</Canvas>
```

代码 15-45 绘制两端贝塞尔曲线

绘制的两端贝塞尔曲线如图 15-93 所示。



图 15-93 两端贝塞尔曲线

当绘制多条贝塞尔曲线时，如果希望两段贝塞尔曲线的连接处光滑，需要第 1 段贝塞尔曲线的第 2 个控制点，并且结束点和第 2 段的贝塞尔曲线的第 1 个控制点共线。例如，修改上面代码的点位坐标使两端贝塞尔曲线连接处光滑：

```
<PolyBezierSegment Points="0,0 200,0 300,100 400,200 400,0 600,100" />
```

结果如图 15-94 所示。

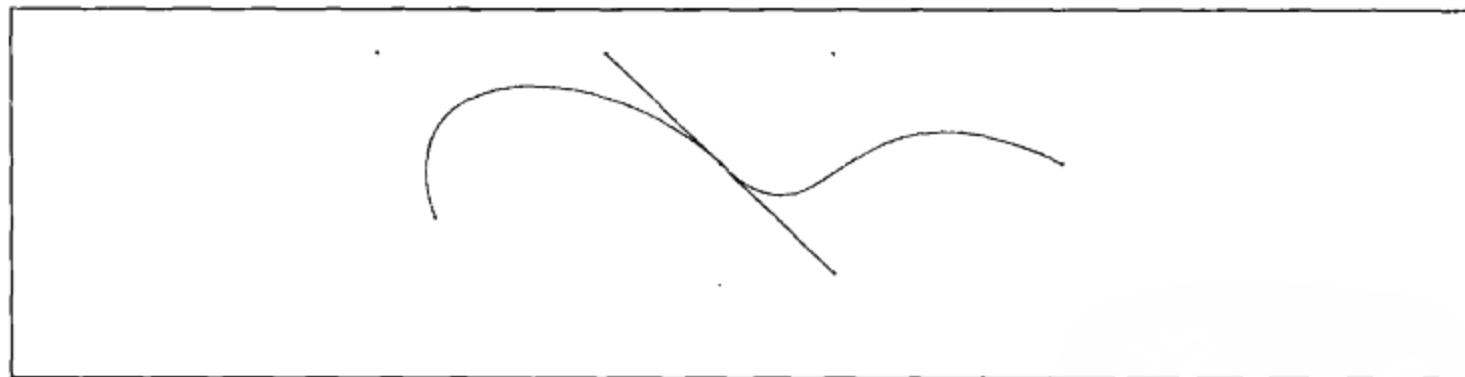


图 15-94 修改两端贝塞尔曲线连接处使之光滑

2. StreamGeometry

StreamGeometry 是一种轻量级的 PathGeometry，可以描述复杂的几何图形。为了提高效率，StreamGeometry 一旦被指定，则不允许修改，而且它不支持数据绑定和动画这样的特性。

StreamGeometry 通过一个 Open 方法获得 StreamGeometryContext 类型来绘制几何图形，代码 15-46 使用 StreamGeometry 绘制一个三角形。

```

public MainWindow()
{
    InitializeComponent();
    // myPath 为一个 Path 对象，该对象在 XAML 文件中通过 x:Name 属性设置
    myPath.Stroke = Brushes.Black;
    myPath.StrokeThickness = 1;
    // 创建一个 StreamGeometry 对象
    StreamGeometry geometry = new StreamGeometry();
    geometry.FillRule = FillRule.EvenOdd;
    // 通过 Open 方法获得 StreamGeometryContext
    using (StreamGeometryContext ctx = geometry.Open())
    {
        // 绘制第 1 个 PathFigure
        ctx.BeginFigure(new Point(10, 100), true, true);
        ctx.LineTo(new Point(100, 100), true, false);
        ctx.LineTo(new Point(100, 50), true, false);
    }
    geometry.Freeze();
    myPath.Data = geometry;
}

```

代码 15-46 使用 StreamGeometry 绘制一个三角形

15.6.5 路径描述语言

WPF 为描述路径专门提供了一套语法 (PathGeometry Markup Syntax)，可以使路径的描述更为简洁。例如，前面使用 4 个 LineSegment 来绘制的五角星（代码 15-43）可以使用代码 15-47 所示的字符串来描述。

```
<Path Fill="Aqua" Stroke="Maroon" StrokeThickness="3" Data ="M 144 72 L200 246,
53 138, 235 138,88 246"/>
```

代码 15-47 用字符串来简单描述五角星

“M” 表示移动 (Move)，相当于 PathFigure 的 StartPoint，“L” 是画线 (Line)，后面的 4 个点形成 3 条线。表 15-9 所示为所有的路径描述语言命令。

表 15-9 路径描述语言的所有命令

命令	含义
PathGeometry 和 PathFigure 属性	
F n	设置填充规则 (FillRule)，其中 0 表示 EvenOdd；1 表示 NonZero，该命令在所有字符串之前
M x, y	开始一个新的 PathFigure 并且设置 StartPoint 为 (x,y)
Z	结束该 PathFigure，并且设置 IsClosed 为 true。如果需该 PathFigure 闭合，则省略该命令。该命令用在一个 PathFigure 结束的末尾
PathSegment	
L x,y	绘制线到(x,y)点
A rx,ry d f1 f2 x,y	建立一个 ArcSegment，rx,ry 为长短半径，d 表示旋转角度，(x,y) 为圆弧的结束点。f1 表示 IsLargeArc 属性，0 表示 false；1 表示 true。f2 表示 SweepDirection 属性，0 表示逆时针；1 表示顺时针
Cx1,y1 x2,y2 x,y	建立一个 BezierSegment，(x1,y1) 和 (x2,y2) 表示两个控制点；(x,y) 表示结束点

命令	含义
Qx1,y1 x,y	建立一个 QuadraticBezierSegment, (x1,y1)为控制点; (x,y)为结束点
其他快捷方式	
H x	创建一条到(x,y)的直线, 其中 y 取当前点的 y 值; H 表示水平线
V y	创建一条到(x,y)的直线, 其中 x 取当前点的 x 值; V 表示垂直线
S x2,y2 x,y	用控制点(x1,y1)和(x2,y2)创建一条到(x,y)的平滑三次贝塞尔曲线, 其中(x1,y1)会自动计算保证曲线的平滑性

此外所有的命令都有一个对应的小写字母命令用来表示当前坐标值不是绝对值, 而是相对值。如 m x0,y0 表示移动到($x+x0, y+y0$)位置, (x,y)表示当前点的坐标。F、M 和 Z 命令大小写含义相同, 即可以用大小写字母表示。

在几何字符串中空格和逗号并没有严格要求, 但是命令和参数之间必须有空格, 参数相互之间必须用空格或者逗号分离。

15.7 Drawing 和 Visual

Drawing 比 Geometry 更完备一些, 包括所有需要绘制的信息, 如几何形状、画笔和画刷等。它不仅仅能够绘制几何形状, 还能够绘制图像、文本, 甚至视频。

15.7.1 Drawing 及其派生类

Drawing 是一个抽象类, 其派生类如表 15-10 所示。

表 15-10 Drawing 的派生类

类名	描述	属性
GeometryDrawing	绘制几何图形	Geometry: 描述几何图形 Brush: 描述如何填充 Pen: 描述如何填充外部轮廓线
ImageDrawing	绘制图像	ImageSource: 指定图像文件的路径 Rect: 指定图像文件的所在位置
VideoDrawing	播放视频	Player: 媒体播放器 Rect: 指定媒体播放器的所在位置
GlyphRunDrawing	绘制文本	GlyphRun: 描述绘制文本的信息 ForegroundBrush: 指定前景色
DrawingGroup	Drawing 的集合, 类似 GeometryGroup	Children: 指定该集合下的子对象

由于 Drawing 不是一个元素, 所以不能直接摆放在用户界面中, 显示其内容有如下方法。

(1) 将 Drawing 放在 DrawingImage 类中，该类一般作为 Image 的一个属性，如代码 15-48 所示。

```
<Image>
    <Image.Source>
        <DrawingImage>
            <DrawingImage.Drawing>
                <DrawingGroup>
                    <GeometryDrawing ...../>
                    <GeometryDrawing ...../>
                    <GeometryDrawing ...../>
                </DrawingGroup>
            </DrawingImage.Drawing>
        </DrawingImage>
    </Image.Source>
<Image>
```

代码 15-48 将 Drawing 放在 DrawingImage 类中

(2) 将 Drawing 放在 DrawingBrush 类中，DrawingBrush 派生自 Brush。因此可以用在多个元素的背景色（Background）和前景色（Foreground）等画刷类型的属性中，如代码 15-49 所示。

```
<Button>
    <Button.Background>
        <DrawingBrush>
            <DrawingBrush.Drawing>
                <DrawingGroup>
                    <GeometryDrawing ...../>
                    <GeometryDrawing ...../>
                    <GeometryDrawing ...../>
                </DrawingGroup>
            </DrawingBrush.Drawing>
        </DrawingBrush>
    </Button.Background>
</Button>
```

代码 15-49 将 Drawing 放在 DrawingBrush 类中

(3) 将 Drawing 放在 DrawingVisual 类中，DrawingVisual 派生自 Visual。虽然它有一个只读的 Drawing 属性，但是需要通过代码方式将 Drawing 放置在 DrawingVisual 中。

15.7.2 Drawing 类型

WPF 引入 Drawing 类型的重要原因是提高了效率。如果使用 Shape 绘制一只兔子，那么需要使用 5 个 Path、1 个 Line 和两个 Ellipse，如图 15-95 所示。

代码如代码 15-50 所示。

```
<Page.Resources>
    <SolidColorBrush x:Key="graybrush" Color="#FFC4C4C4"/>
</Page.Resources>
<Canvas>
    <TextBlock Height="27" Canvas.Left="91" Canvas.Top="224" Width="198">
        用 Shape 画成的兔子
        <LineBreak/>
        使用了 5 个 Path, 1 个 Line 和 2 个 Ellipse
    </TextBlock>
```

```

<Path Fill="{StaticResource graybrush}" Canvas.Left="-100"
Canvas.Top="-50" Data="... "/>
<Path Fill="{StaticResource graybrush}" Canvas.Left="-100"
Canvas.Top="-50" Data="... "/>
<Path Fill="{StaticResource graybrush}" Canvas.Left="-100"
Canvas.Top="-50" Data="... "/>
<Path Fill="White" Canvas.Left="-100" Canvas.Top="-50" Data="... "/>
<Path Fill="White" Canvas.Left="-100" Canvas.Top="-50" Data="... "/>
<Line Stroke="Black" Canvas.Left="-100" Canvas.Top="-50"
StrokeThickness="3" X1="267" Y1="246" X2="290.5" Y2="245.5" />
<Ellipse Fill="Black" Width="5" Height="5" Canvas.Left="149"
Canvas.Top="112"/>
<Ellipse Fill="Black" Width="5" Height="5" Canvas.Left="205"
Canvas.Top="116"/>
</Canvas>

```

代码 15-50 使用 Shape 绘制兔子

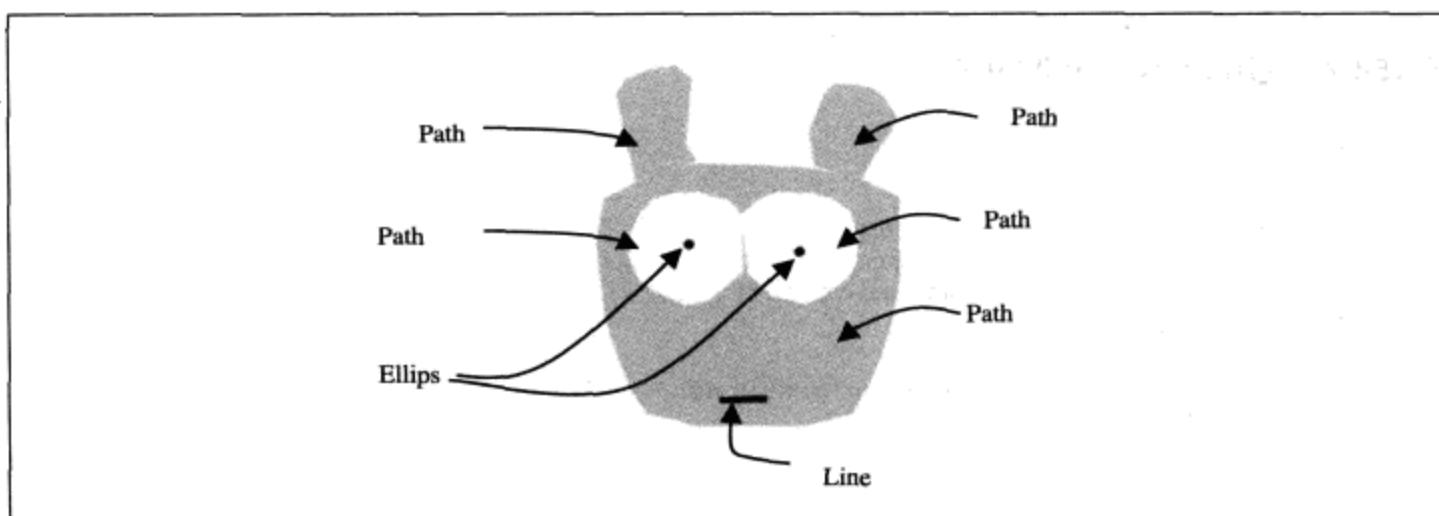


图 15-95 使用 Shape 绘制的兔子

Shape 继承自 FrameworkElement，不仅能够绘制图形，而且具有响应事件和参与布局的特性，因此带来的系统开销很大。一个好的方法是用一个 Path 来代替图形，但是由于兔子的眼睛和其他部位的颜色不同，所以只能做部分简化，如图 15-96 所示，由两个 Path、1 个 Line 和两个 Ellipse 组合而成。

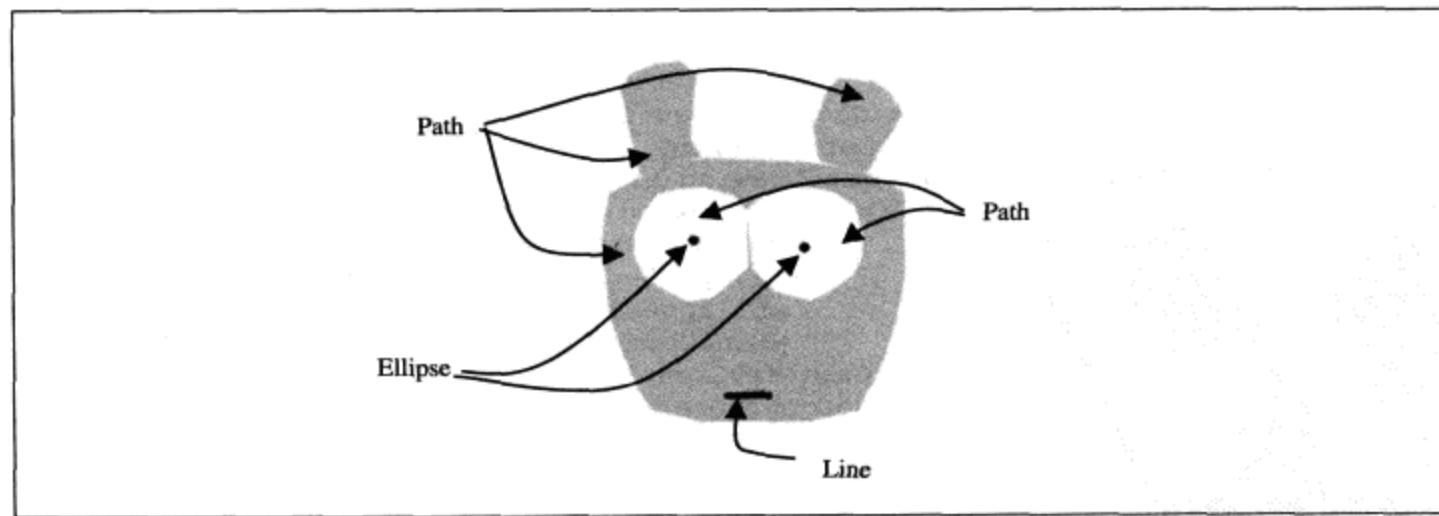


图 15-96 兔子的部分简化

这种简化只是在某种程度上减少了元素的个数，但是使用 Drawing 则产生了根本性变化。整只兔子

只使用了一个元素（Image），该元素由 3 个 GeometryDrawing 组成，如图 15-97 所示。

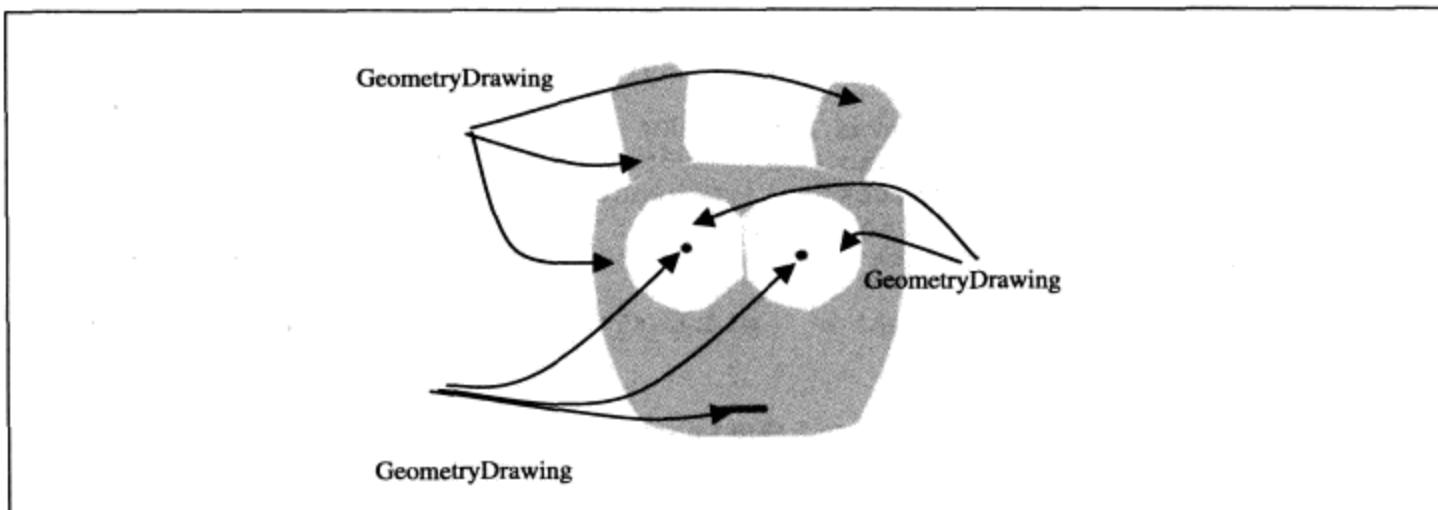


图 15-97 使用 Drawing 绘制兔子

代码如代码 15-51 所示。

```
<Page x:Class="mumu_rabbit.Page3"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page3">
    <Page.Resources>
        <SolidColorBrush x:Key="graybrush" Color="#FFC4C4C4"/>
    </Page.Resources>
    <Grid>
        <Image Stretch="None" HorizontalAlignment="Center" VerticalAlignment="Center">
            <Image.Source>
                <DrawingImage>
                    <DrawingImage.Drawing>
                        <DrawingGroup>
                            <GeometryDrawing Brush="{StaticResource graybrush}">
                                <GeometryDrawing.Geometry>
                                    <PathGeometry Figures="..." />
                                </GeometryDrawing.Geometry>
                            </GeometryDrawing>
                            <GeometryDrawing Brush="White">
                                <GeometryDrawing.Geometry>
                                    <PathGeometry Figures="... " />
                                </GeometryDrawing.Geometry>
                            </GeometryDrawing>
                            <GeometryDrawing Brush="Black">
                                <GeometryDrawing.Geometry>
                                    <GeometryGroup>
                                        <EllipseGeometry ...../>
                                        <EllipseGeometry ...../>
                                        <LineGeometry ...../>
                                    </GeometryGroup>
                                    </GeometryDrawing.Geometry>
                                </GeometryDrawing>
                            </GeometryDrawing>
                        </DrawingGroup>
                    </DrawingImage.Drawing>
                </DrawingImage>
            </Image.Source>
        </Image>
    </Grid>
```

```
</Page>
```

代码 15-51 GeometryDrawing 画法

虽然代码的长度没有缩短，但是 GeometryDrawing 和 Shape 相比已经是轻量级类型，图形绘制的效率会显著提高。关于兔子的示例请参见 `mumu_rabit` 工程。

Drawing 还带来了另外一个好处，即可以将图形、图像、文本和视频统一起来。图 15-98 所示为通过 DrawingBrush 画刷将一个 Rectangle 的背景设置为图片和视频的组合，如代码 15-52 所示（详见 `mumu_drawingbrushdemo` 工程）。

```
MainWindow.xaml
<Window x:Class="demo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="300" Width="300" Loaded="Window_Loaded">
    <Grid>
        <Rectangle x:Name="btn">
            <Rectangle.Fill>
                <DrawingBrush>
                    <DrawingBrush.Drawing>
                        <DrawingGroup>
                            <VideoDrawing x:Name="aVideoDrawing"/>
                            <ImageDrawing x:Name="img" ImageSource="XBox_logo.png"/>
                        </DrawingGroup>
                    </DrawingBrush.Drawing>
                </DrawingBrush>
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Window>
```

代码 15-52 通过 DrawingBrush 画刷组合图片和视频

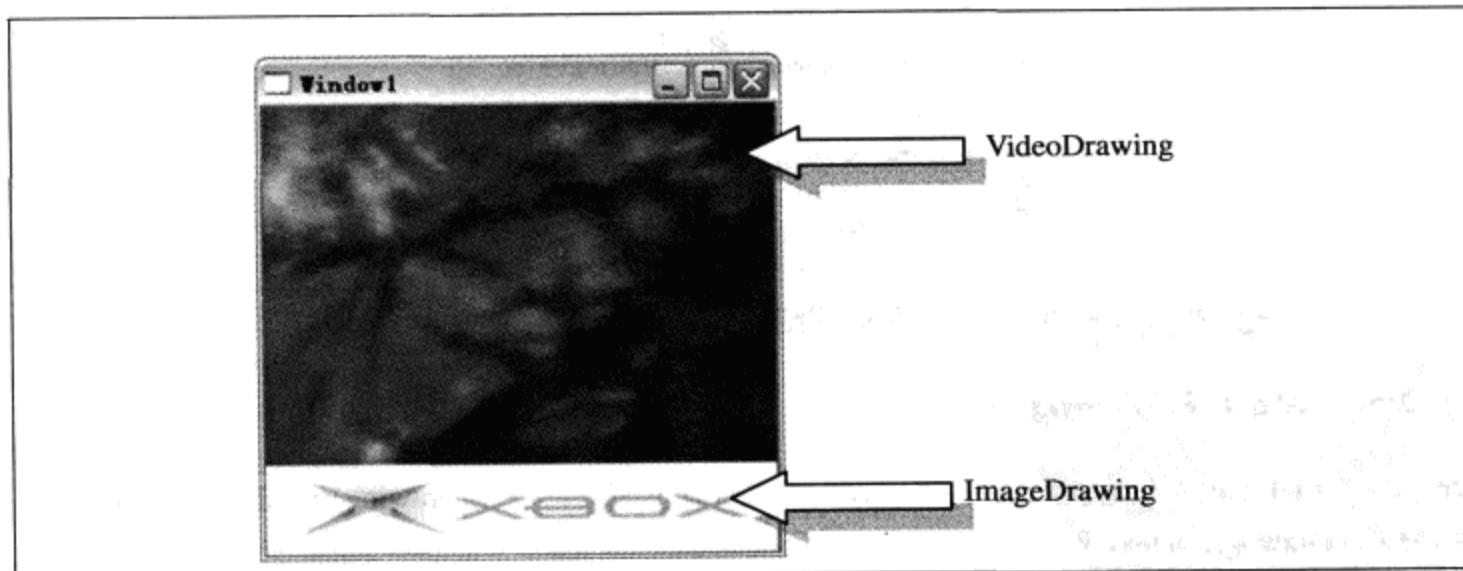


图 15-98 通过 DrawingBrush 画刷组合图片和视频

在窗口的 Loaded 事件中设置 VideoDrawing 和 ImageDrawing 的位置，并播放视频，如代码 15-53 所示。

```

MainWindow.xaml.cs

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        MediaPlayer player = new MediaPlayer();
        player.Open(new Uri(@"xbox.wmv", UriKind.Relative));
        aVideoDrawing.Rect = new Rect(0, 0, btn.ActualWidth, btn.ActualHeight * 0.8);
        aVideoDrawing.Player = player;
        player.Play();
        img.Rect = new Rect(0, btn.ActualHeight * 0.8, btn.ActualWidth,
        btn.ActualHeight * 0.2);
    }

```

代码 15-53 播放视频的代码实现

15.7.3 Visual

WPF 引入了一个 DrawingVisual 类，它与 FrameworkElement 派生的类型相比是一个轻量级的类。它没有很多高级特性，只保留了渲染所必需的特性，如透明和剪切等。同时由于 DrawingVisual 派生于 Visual，因此也保留了 Hit-test 行为，该行为是用户和 DrawingVisual 交互的必要条件。

Image 和 DrawingVisual 类的层次结构如图 15-99 所示。

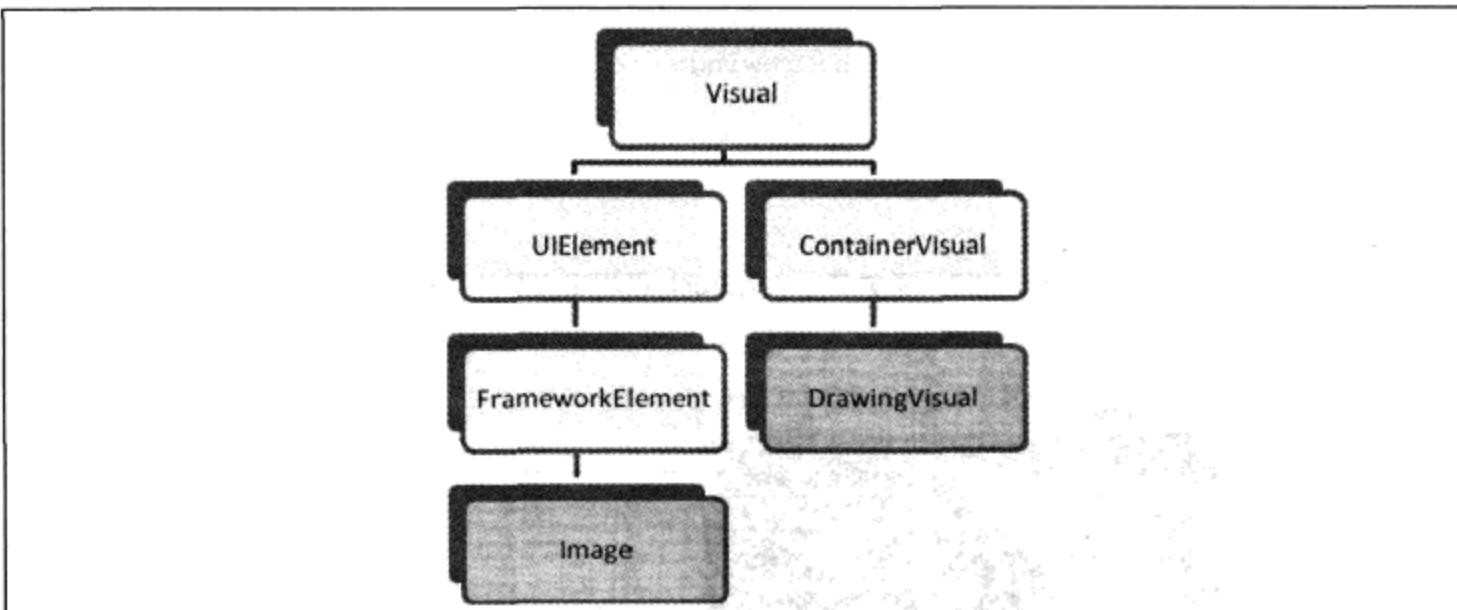


图 15-99 Image 和 DrawingVisual 的类层次结构

1. DrawingVisual 和 DrawingContext

DrawingVisual 不能直接在 XAML 文件中设置其 Drawing 属性，要在 DrawingVisual 上绘制 Drawing 必须借助 DrawingContext 类。

DrawingContext 类似 GDI 中的 HDC，表 15-11 所示为其主要方法。

表 15-11 DrawingContext 的主要方法

函数名称	描述
DrawLine, DrawRectangle, DrawingRoundedRectangle 和 DrawEllipse	绘制各种简单的几何图形
DrawGeometry	直接绘制 Geometry 对象
DrawText	绘制文本
DrawDrawing	直接绘制 Drawing 对象
DrawImage	绘制图像
DrawVideo	绘制视频
PushClip	将裁减区域应用到随后的绘图命令
PushEffect	将特效应用到随后的绘图命令
PushOpacity	将透明度设置应用到随后的绘图命令
PushTransform	将几何变换应用到随后的绘图命令
Pop	取消当前的设置，返回到上一次状态

Push***和 Pop 是一个成对命令，配合使用可以对所绘制的图形应用透明或者旋转等效果。

DrawingContext 通过 DrawingVisual 的 RenderOpen 方法获得，使用后还需要调用自身的 Close 方法将其关闭，代码 15-54 通过 DrawingVisual 在一个 Image 控件（名为“img”）上绘制文本。

```
FormattedText text = new FormattedText("ABC",
    new CultureInfo("en-us"),
    FlowDirection.LeftToRight,
    new Typeface(this.FontFamily,           FontStyles.Normal,
FontWeights.Normal, new FontStretch()),
    this.FontSize,
    this.Foreground);

DrawingVisual drawingVisual = new DrawingVisual();
DrawingContext drawingContext = drawingVisual.RenderOpen();
drawingContext.DrawText(text, new Point(2, 2));
drawingContext.Close();

RenderTargetBitmap bmp = new RenderTargetBitmap(180, 180, 120, 96,
PixelFormats.Pbgra32);
bmp.Render(drawingVisual);
img.Source = bmp;
```

代码 15-54 通过 DrawingVisual 在一个 Image 控件上绘制文本

2. 使用 Visual 绘制兔子

现在使用 Visual 绘制兔子，将其按照头部、耳朵、眼睛和嘴巴分为 4 个部分，由 4 个 DrawingVisual 组成。使用 Visual 绘制图形首先需要实现一个能够容纳 Visual 的容器。该容器需要重载一个属性 VisualChildCount 和一个方法 GetVisualChild，前者告诉 WPF 该容器 Visual 的数量，后者能够按照索引号获取任意一个 Visual。如代码 15-55 所示，其中 headdrawingvisual、eardrawingvisual、eyedrawingvisual 和 mousedrawingvisual 分别代表兔子的头部、耳朵、眼睛和嘴巴 4 个部分（详见

mumu_rabit 工程)。

```
public class DrawingVisualHost : FrameworkElement
{
    private VisualCollection _children;
    private DrawingVisual headdirrawingvisual;
    private DrawingVisual eardrawingvisual;
    private DrawingVisual eyedrawingvisual;
    private DrawingVisual mousedrawingvisual;

    public DrawingVisualHost()
    {
        _children = new VisualCollection(this);
        headdirrawingvisual = CreateHeadDrawingVisual(true);
        eardrawingvisual = CreateEarDrawingVisual(true);
        eyedrawingvisual = CreateEyeDrawingVisual(true);
        mousedrawingvisual = CreateMouseDrawingVisual(true);

        _children.Add(headdirrawingvisual);
        _children.Add(eardrawingvisual);
        _children.Add(eyedrawingvisual);
        _children.Add(mousedrawingvisual);
    }

    .....

    protected override int VisualChildrenCount
    {
        get { return _children.Count; }
    }
    protected override Visual GetVisualChild(int index)
    {
        if (index < 0 || index >= _children.Count)
        {
            throw new ArgumentOutOfRangeException();
        }
        return _children[index];
    }
}
```

代码 15-55 使用 Visual 绘制兔子的头部、耳朵、眼睛和嘴巴 4 个部分

代码 15-56 中的 CreateHeadDrawingVisual 函数负责绘制兔子的头部，它会根据传递过的参数值绘制不同形状的头部。

```
private DrawingVisual CreateHeadDrawingVisual(bool change)
{
    DrawingVisual drawingVisual = new DrawingVisual();
    DrawingContext drawingContext = drawingVisual.RenderOpen();
    GeometryDrawing headdiriving = new GeometryDrawing();
    SolidColorBrush solidcolorbrush = this.FindResource("headcolor") as
SolidColorBrush;
    if (solidcolorbrush == null)
    {
        drawingContext.Close();
        return null;
    }

    if (change)
    {
        Geometry headgeometry = this.FindResource("geometryhead1") as Geometry;
        if (headgeometry == null)
        {
```

```

        drawingContext.Close();
        return null;
    }
    headdrawing.Brush = solidcolorbrush;
    headdrawing.Geometry = headgeometry;
}
else
{
    Geometry headgeometry = this.FindResource("geometryhead2") as Geometry;
    if (headgeometry == null)
    {
        drawingContext.Close();
        return null;
    }
    headdrawing.Brush = solidcolorbrush;
    headdrawing.Geometry = headgeometry;
}

drawingContext.DrawDrawing(headdrawing);
drawingContext.Close();
return drawingVisual;
}

```

代码 15-56 CreateHeadDrawingVisual 函数负责绘制兔子的头部

绘制兔子的眼睛、耳朵和嘴巴均与上类似。如果需要为兔子添加一个黑眼圈，可以使用 `BitmapEffect` 添加特效。如代码 15-57 所示，在绘制兔子眼睛之前添加一个 `OuterGlowBitmapEffect` 效果，然后在绘制眼睛后调用 `DrawingContext` 的 `Pop` 方法。

```

OuterGlowBitmapEffect bitmapeffect = new OuterGlowBitmapEffect();
bitmapeffect.GlowColor = Colors.Black;
bitmapeffect.GlowSize = 10;
drawingContext.PushEffect(bitmapeffect, null);

// 兔眼眶
.....
// 兔眼珠
.....
drawingContext.Pop();

```

代码 15-57 绘制兔子眼睛之前添加一个 `OuterGlowBitmapEffect` 效果

运行结果如图 15-100 所示。

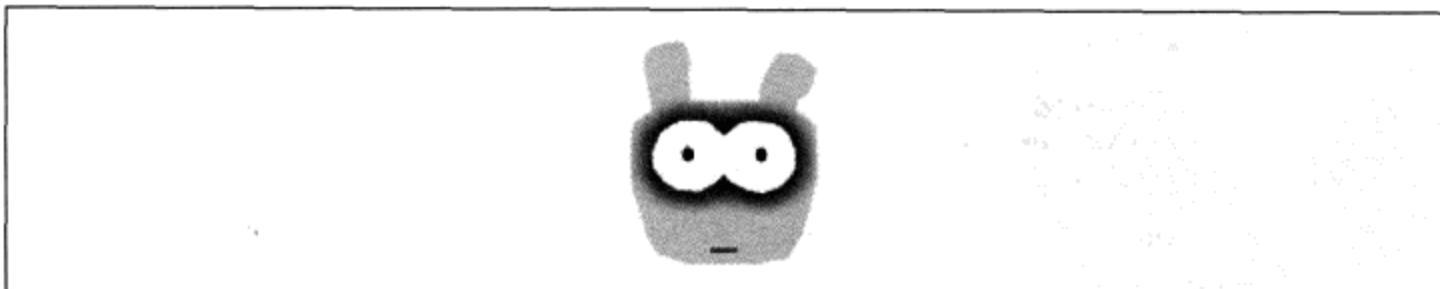


图 15-100 运行结果

3. HitTest 行为

我们之所以使用了 4 个 Visual 绘制一只兔子，是希望鼠标在单击每一个部分时能够检测到是否命中；

否则一个 Visual 足以胜任绘制工作。单击兔子的不同部位时，该部位检测到鼠标命中，则绘制另外一种形状；右击时兔子恢复原状，如图 15-101 所示。

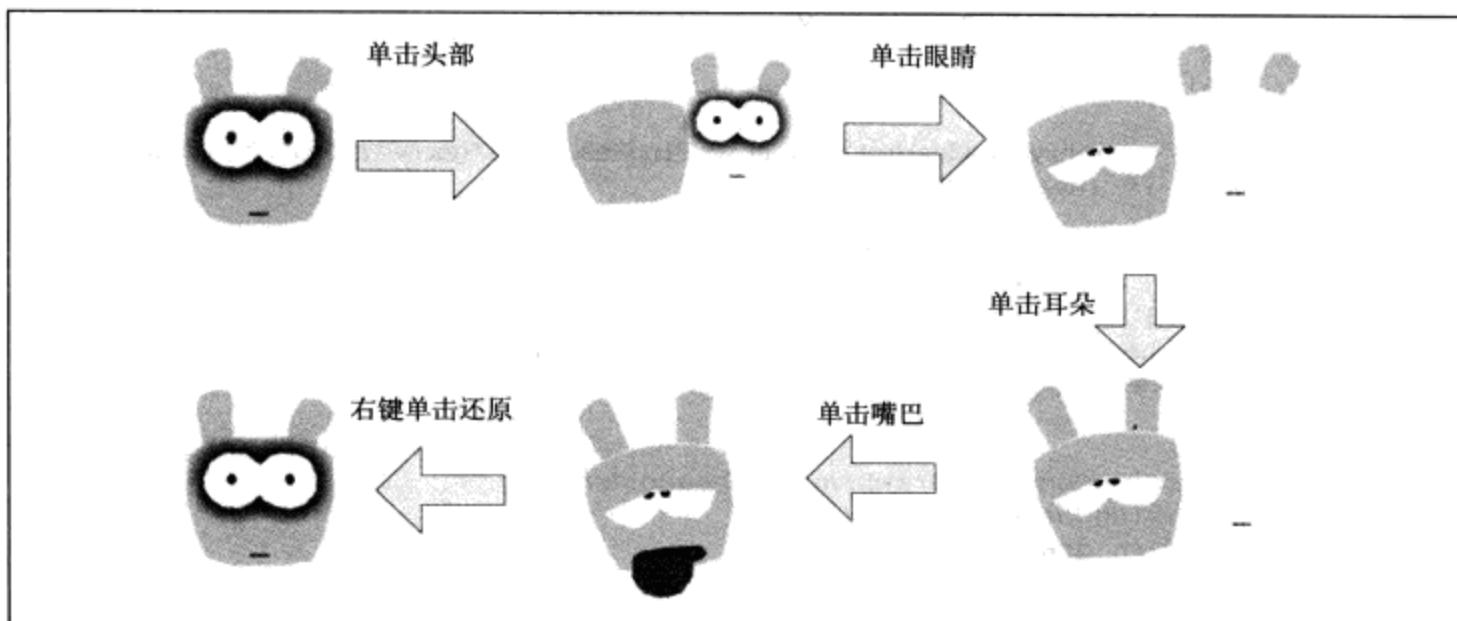


图 15-101 HitTest 行为效果

要检测鼠标主要通过 VisualTree 的一个静态方法 HitTest，可以直接将当前的鼠标点位作为参数传递给它。该方法返回一个 HitTestResult 类型的对象，其中有一个 VisualHit 属性。如果 Visual 检测到鼠标命中，则为该 Visual；否则为 null。代码 15-58 为鼠标左键抬起的事件中使用 VisualTree 的 HitTest 方法来检测鼠标是否命中。

```
void MyVisualHost_MouseLeftButtonUp(object sender,
System.Windows.Input.MouseEventArgs e)
{
    System.Windows.Point pt = e.GetPosition((UIElement)sender);

    HitTestResult hittestresult = VisualTreeHelper.HitTest(this, pt);

    if (headdrawingvisual == hittestresult.VisualHit)
    {
        _children.Remove(headdrawingvisual);
        headdrawingvisual = CreateHeadDrawingVisual(false);
        _children.Insert(0, headdrawingvisual);

    }
    else if (eardrawingvisual == hittestresult.VisualHit)
    {
        _children.Remove(eardrawingvisual);
        eardrawingvisual = CreateEarDrawingVisual(false);
        _children.Insert(1, eardrawingvisual);
    }
    else if (eyedrawingvisual == hittestresult.VisualHit)
    {
        _children.Remove(eyedrawingvisual);
        eyedrawingvisual = CreateEyeDrawingVisual(false);
        _children.Insert(2, eyedrawingvisual);
    }
    else if (mousedrawingvisual == hittestresult.VisualHit)
    {
        _children.Remove(mousedrawingvisual);
```

```
        mousedyngvisual = CreateMouseDrawingVisual(false);
        _children.Insert(2, mousedyngvisual);
    }
}
```

代码 15-58 鼠标左键抬起的事件中使用 VisualTree 的 HitTest 方法来检测鼠标是否命中

如果希望一次单击检测多个 Visual 对象（有重叠情况，如兔子的头部和眼睛重叠），那么可以使用回调函数的方法来检测鼠标命中，如代码 15-59 所示。

```
void MyVisualHost_MouseLeftButtonUp(object sender,
System.Windows.Input.MouseEventArgs e)
{
    VisualTreeHelper.HitTest(this, null, new
HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
}
public HitTestResultBehavior myCallback(HitTestResult result)
{
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (headdrawingvisual == result.VisualHit)
        {
            _children.Remove(headdrawingvisual);
            headdrawingvisual = CreateHeadDrawingVisual(false);
            _children.Insert(0, headdrawingvisual);

        }
        else if (eardrawingvisual == result.VisualHit)
        {
            _children.Remove(eardrawingvisual);
            eardrawingvisual = CreateEarDrawingVisual(false);
            _children.Insert(1, eardrawingvisual);
        }
        else if (eyedrawingvisual == result.VisualHit)
        {
            _children.Remove(eyedrawingvisual);
            eyedrawingvisual = CreateEyeDrawingVisual(false);
            _children.Insert(2, eyedrawingvisual);
        }

        else if (mousedyngvisual == result.VisualHit)
        {

            _children.Remove(mousedyngvisual);
            mousedyngvisual = CreateMouseDrawingVisual(false);
            _children.Insert(3, mousedyngvisual);
        }
    }
    else
    {
        _children.RemoveRange(0, 4);
        headdrawingvisual = CreateHeadDrawingVisual(true);
        eardrawingvisual = CreateEarDrawingVisual(true);
        eyedrawingvisual = CreateEyeDrawingVisual(true);
        mousedyngvisual = CreateMouseDrawingVisual(true);

        _children.Add(headdrawingvisual);
        _children.Add(eardrawingvisual);
        _children.Add(eyedrawingvisual);
        _children.Add(mousedyngvisual);
    }
}
```

```
        }
        return HitTestResultBehavior.Stop;
    }
}
```

代码 15-59 使用回调函数的方法来检测鼠标命中

myCallback 回调函数返回的枚举值如果是 HitTestResultBehavior 的 Continue 值，WPF 会继续查找下一个被鼠标命中的 Visual；如果是 Stop 值，则找到了第 1 个被鼠标命中的 Visual 后不再继续寻找。

15.8 接下来做什么（面壁之后）

两块石壁，四本书，这就是黄药师的图形世界。在这个图形世界里，唯有位图效果（Bitmap Effects）没有介绍，这项技术虽然能够展现一些特殊的效果，但是其本身并没有利用硬件加速，而是利用软件实现的。因此效率比较低。WPF 为了解决这一问题，在.NET Framework 3.5 SP1 当中增加了一个 Effect 类，用于取代 BitmapEffect。对于一个即将被取代的类，因此本书并没有覆盖。目前国外有一本《Practical WPF Graphics Programming》，作者是 Jack Xu（听名字貌似中国人^_^）详细讨论了 WPF 的二维图形。当然这本书也讨论了 WPF 的三维图形，但是有另外一本书近乎完美地讨论了 WPF 的三维图形。至于书的名字，我们还是留待 WPF3D 图形一章再来揭晓。

四本书看完，木木真是有一种恍若隔世的感觉，舒展舒展筋骨，数数面壁的日子已经过去了十天。十天来陪伴他的除去蓉儿，就是这无尽的图形。还要继续修炼下去，前面等待的将是更为精彩的画面，在哪里图形可以动起来，图形也可以变成立体。至于药师的图形编辑器，后来木木真的一个人独自完成了。不过那也是很久以后的事了。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作 《金庸全集典藏版 笑傲江湖》，“第八章 面壁”。
- [2] Helloj2ee 2009，“WPF 单位真的与分辨率无关吗？”，<http://www.cnblogs.helloj2ee/archive/2009/04/21/1440709.html>。
- [3] CalvinP. Schrottenboer 2006，“Is WPF Really Resolution Independent? ”，<http://www.wpflearningexperience.com/?p=41>。
- [4] MSDN Library for Visual Studio 2008 SP1 Windows Presentation Foundation Graphics Renering Overview.
- [5] Charles Peztold 著 蔡学庸译 Windows Presentation Foundation 程序设计指南：第二章 基本画刷。

动画——降龙的最后一掌

黄蓉哭了一会儿，抽抽噎噎地道：“我听爹爹说过，洪老前辈有一套武功，当真是天下无双、古今独步，甚至全真教的王重阳也忌惮三分。叫做……叫做……咦，我怎么想不起来啦，明明刚才我还记得的。我想求他教你，这套拳法叫做……叫做……”其实她哪里知道全是信口胡吹，洪七公在树顶上见她苦苦思索，实在忍不住了，喝道：“叫做‘降龙十八掌’！”

——《射雕英雄传》，“第十二回 兖龙有悔”^[1]

如果看过周星星的《武状元苏乞儿》的话，一定还记得降龙十八掌的最后一掌，它实际上是合前 17 掌连续而成。当苏乞儿被打倒在地，一阵妖风吹起，翻弄了降龙十八掌的秘籍，让前 17 掌形成了一幅连续的动画。苏乞儿终于明白了最后一掌的奥秘，这就是本章所要介绍的动画。

本章的内容如下。

- (1) 七公和他的降龙十八掌。
- (2) WPF 实现动画的方式。
- (3) WPF 动画的基本知识。
- (4) 3 种基本类型的动画。
- (5) 动画的交互控制。
- (6) 后记：降龙的最后一掌。
- (7) 接下来做什么。

16.1 七公和他的降龙十八掌

这天桃花岛公司迎来了一位非常尊贵的客人，丐帮帮主洪七公。药师和七公是多年的至交，一大早药师就带了众人在桃花岛公司门口迎接，木木自然也跟在药师左右。

等了良久，突然间远处隐隐传来一阵长啸之声。药师拿起了自己的玉箫，箫声即起。远处那啸声又呼地拔高，箫声跟着拔高，长啸和箫声此起彼伏。两般声音忽高忽低，长啸声时而如龙吟狮吼、狼嗥枭鸣，箫声则若长风振林、微雨湿花，极尽千变万化之致。这时发啸之人已经近在眼前，只见此人背负大红葫芦，右手拿着竹杖，正是丐帮帮主——洪七公是也。“老邪，老邪啊……”，七公笑

吟吟地走近。“七兄”，药师抱拳，虽不动声色，但也难掩笑意。俩人见过了礼，药师将手下高管一一介绍给七公。寒喧数语，便进了公司。

进得药师办公室之后，只剩下药师，七公和木木三人。药师指着木木说到：“这是我的小婿，久闻七公的降龙十八掌，七公可否不吝赐教。”说着深深一揖。“老叫化不敢当，不敢当。”七公连连摆手，从怀中掏出了一本册子，册子面上用篆文书写了《降龙十八掌》5个大字。七公说到：“我也年迈了，这几年我将每一掌都绘在了上面并配有文字说明，木木你自己拿去参详吧。”

木木谢过了七公，就将册子一页一页地读了起来。册子很薄，很快木木就翻完了。翻完之后，木木很是不解，上前鞠了一躬说到：“七公老前辈，为什么这上面只有17掌的图解，而没有最后一掌。”七公笑到：“你将书迅速从头翻到尾，便是第17掌了。”木木依言照做，插图随即形成了一幅连续的动画。木木顿时醒悟，原来这第17掌就是前十七掌的合而为一。谁知木木仍在发呆，过了半晌，还是一语不发。药师了解木木，知道这傻小子可能又在思考什么问题。七公不解，以为木木和郭靖一样也是个呆头呆脑的傻小子，没有悟到这第18掌的奥秘，不忍便说了一句：“木木，这第18掌就是连续的17掌啊！”

木木被七公的话拉回了思绪，连忙恭恭敬敬地又鞠一躬，说道：“老前辈，刚才翻书时，我已经悟到了这个道理。不过我想了想，快速翻书的方式也许陈旧了一些，也不便观察一招一式。我在想如果将您这17掌做成功动画，岂不是方便很多。”

药师和七公听完木木这番话大喜，没想到这个傻小子居然能想到这一层，真是有青出于蓝胜于蓝之势。药师说道：“木木，你这就将七公的书拿去，做成功动画吧。”“好的”木木拿着书退了出去……

经历了这么多次的学习，木木已经很有经验了。他并不着急将这个程序做出来，而系统地开始学习WPF的动画。

16.2 WPF 实现动画的方式

16.2.1 基于计时器的动画

过去实现动画的经典方法是建立一个定时器，然后根据其频率循环调用回调函数或者一个事件处理函数。在这个函数中可以手工更新目标属性，直到达到最终值，这时可以停止定时器。

WPF 中也提供了 DispatcherTimer 类型的定时器，可以通过该类实现这样的方案。下例通过 DispatcherTimer 实现一个矩形变宽的动画（详见 `mumu_animationwithtimer` 工程），如图 16-1 所示。

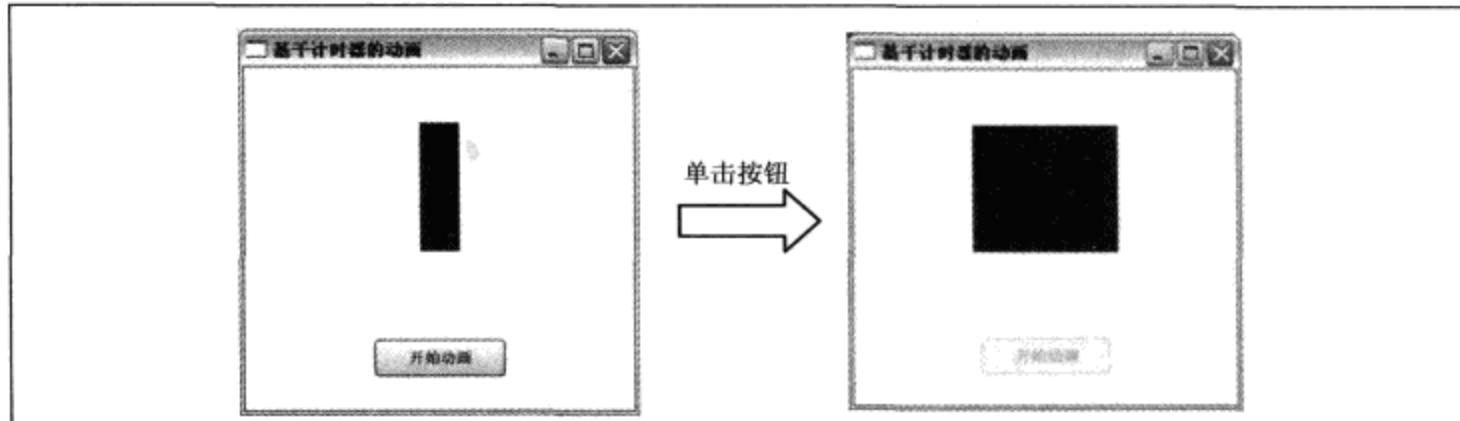


图 16-1 矩形变宽的动画



DispatcherTimer 和其他 .Net 计时器的区别?

DispatcherTimer 与 System.Threading.Timer 或者 System.Timers.Timer 等定时器的主要区别在于其 Tick 事件处理程序在 UI 线程中调用, 这一点对于 WPF 应用程序非常重要。因为这样 Tick 事件处理函数中可以直接操作 UI 元素, 修改其属性。如果使用其他定时器, 则需要将更新 UI 的逻辑放在不同的函数中, 然后用 Dispatcher 在 UI 线程中调用。

在 XAML 文件中定义了一个矩形和一个按钮, 单击该按钮即可开始动画, 如代码 16-1 所示。

```
<Window x:Class="mumu_animationwithtimer.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用定时器的动画" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="7*"/>
            <RowDefinition Height="3*"/>
        </Grid.RowDefinitions>
        <Rectangle Grid.Row="0" x:Name="rectangle" Width="30" Height="100" Fill="Blue">
        </Rectangle>
        <Button Grid.Row="1" Width="100" Height="30" Click="Button_Click">
            开始动画
        </Button>
    </Grid>
</Window>
```

代码 16-1 MainWindow.xaml 文件

在按钮的 Click 事件处理函数中设置了 DispatcherTimer 对象的时间间隔为 0.1 秒, 然后添加了 Tick 的事件处理函数(TimerOnTick), 最后通过 Start 方法启动。这样系统会每隔 0.1 秒¹ 调用 TimerOnTick。该函数每次将矩形的宽度增加 10 个 DPI, 直到最大的宽度恢复最初的宽度。然后通过 Stop 方法结束这个计时器。如代码 16-2 所示。

```
public partial class MainWindow : Window
{
    private double maxWidth = 100;
    private double startWidth = 0;
    public MainWindow()
    {
```

¹ 实际上计时器并不能保证精确的 0.1 秒, 但是能保证不小于 0.1 秒。

```

        InitializeComponent();
        startWidth = rectangle.Width;
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        DispatcherTimer tmr = new DispatcherTimer();
        tmr.Interval = TimeSpan.FromSeconds(0.1);
        tmr.Tick += TimerOnTick;
        tmr.Start();
    }

    void TimerOnTick(object sender, EventArgs args)
    {
        rectangle.Width += 10;
        if (rectangle.Width >= maxWidth)
        {
            rectangle.Width = startWidth;
            (sender as DispatcherTimer).Stop();
        }
    }
}

```

代码 16-2 基于计时器的动画

16.2.2 基于帧的动画

WPF 提供了一种基于帧的动画实现方式，由 `CompositionTarget` 类来完成。它提供了一个回调函数（`Rendering` 的事件处理函数），WPF 会在每次界面刷新时调用该回调函数。`CompositionTarget` 的刷新率与窗体保持一致，因此很难人工控制动画的快慢。与前例的 XAML 文件相同，如代码 16-3 所示（详见 `mumu_animationbasedonframe` 工程）。

```

public partial class MainWindow : Window
{
    private double maxWidth = 300;
    private double startWidth = 0;
    public MainWindow()
    {
        InitializeComponent();
    }
    void CompositionTarget_Rendering(object sender, EventArgs e)
    {
        rectangle.Width += 10;
        if (rectangle.Width >= maxWidth)
        {
            rectangle.Width = startWidth;
            CompositionTarget.Rendering -= new
                EventHandler(CompositionTarget_Rendering);
            btn.IsEnabled = true;
        }
    }
    private void btn_Click(object sender, RoutedEventArgs e)
    {
        CompositionTarget.Rendering += new
            EventHandler(CompositionTarget_Rendering);
    }
}

```

代码 16-3 基于帧的动画

动画的实现均在 `Rendering` 的事件处理函数中完成，如果矩形框达到指定的宽度，则移除该事件处理函数，动画结束。动画的快慢依赖于计算机硬件的性能。

16.2.3 基于属性的动画

代码 16-4 和上例实现同样的动画效果（详见 `mumu_animationbasedonproperty` 工程），其 XAML 文件相同。不同的是没有了定时器，有一个 `DoubleAnimation` 类指定起始值（`From="30"`）、终点值（`To="300"`）、时间（`Duration="0:0:2.7"`），以及动画结束应该如何（`FillBehavior="Stop"`）。设置好后该矩形调用 `BeginAnimation` 方法开始实现动画，`BeginAnimation` 指定需要应用动画的属性（注意这里传入的必须是依赖属性）和创建的 `DoubleAnimation`。

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    DoubleAnimation doubleanimation = new DoubleAnimation();
    doubleanimation.From = 30;
    doubleanimation.To = 300;
    doubleanimation.Duration = TimeSpan.FromSeconds(2.7);
    doubleanimation.FillBehavior = FillBehavior.Stop;
    rectangle.BeginAnimation(Rectangle.WidthProperty, doubleanimation);
}
```

代码 16-4 使用 `BeginAnimation` 方法触发动画

除了使用 `BeginAnimation` 方法以外，还可以使用 `ApplyAnimationClock` 方法。不过这里传入的第 2 个参数是一个 `Clock` 类型的对象，如代码 16-5 所示。

```
rectangle.ApplyAnimationClock(Rectangle.WidthProperty, doubleanimation.CreateClock());
```

代码 16-5 使用 `ApplyAnimationClock` 方法触发动画

实现动画的 3 种方式不能说孰优孰劣，应根据不同的问题和场景来选择。

基于属性的动画实现方式最为直观和方便实用，在 WPF 中用得最多的也是这种方式。本章将详细介绍这种方式，如果没有特别说明，所指动画的实现方式均基于属性。

16.3 WPF 动画的基本知识

16.3.1 前提条件

如果想要顺利使用 WPF 的动画，则满足以下条件。

- (1) 必须是依赖属性。
- (2) 应用动画的属性所属的类必须派生自 `DependencyObject`，而且实现了 `IAnimatable` 接口。由于 WPF 绝大多数类都会派生自 `DependencyObject`，所以实现 `IAnimatable` 接口主要有 3 个类，即 `UIElement`、`ContentElement` 和 `Animatable`，应用动画属性所属于的类必须派生自这 3 个类。

(3) 该属性的类型必须是可以应用动画的类型，如 Double、Int 或者 Point。Window 类型就不是一个可以应用动画的类型，WPF 针对 22 种基本类型提供了相应的动画类。如果希望一些类型可以应用动画，则需要扩展动画类。

由于 WPF 绝大多数类型，特别是界面元素派生自 UIElement 和 ContentElement 中的一个，而且 WPF 中的绝大多数属性都是依赖属性，因此动画可以应用到几乎任何地方。还有一点要注意，尽管有些属性不是可见的，只要满足上述条件，也可以应用动画。

前面的例子中使用 BeginAnimation 和 ApplyAnimationClock 方法来触发动画开始，更常用的做法是使用 Trigger 和故事板（Storyboard），如前面的例子还可以改写为代码 16-6。

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="7*"/>
        <RowDefinition Height="3*"/>
    </Grid.RowDefinitions>
    <Rectangle Grid.Row="0" x:Name="rectangle" Width="30" Height="100" Fill="Blue">
    </Rectangle>
    <Button Grid.Row="1" Width="100" Height="30" >
        <Button.Triggers>
            <EventTrigger RoutedEvent="Button.Click">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Storyboard.TargetName="rectangle"
                            Storyboard.TargetProperty="Width"
                            From="30" To="300" Duration="0:0:2.7" FillBehavior="Stop"/>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger >
        </Button.Triggers>
        开始动画
    </Button>
</Grid>
```

代码 16-6 使用 Trigger 和故事板触发动画

所有动画的设置在 XAML 文件中完成，动画通过 Button 的 Trigger 来启动。Button 的 Trigger 集合中有一个 EventTrigger 类型的 Trigger，这种类型大多与动画一起使用（也在播放声音时使用）。也可以用其他类型的 Trigger 来启动动画，EventTrigger 的 RoutedEvent 属性用来指定触发执行相关动作的事件，例中为按钮的 Click 事件。

在 XAML 中动画总是涉及 Storyboard，其作用主要是提供两个附加属性用于指定应用动画的对象和属性。只有 Storyboard 动画还是无法启动，一个重要的动作即 BeginStoryboard，通过它整个动画才得以启动。

16.3.2 动画类的类层次结构

在基于属性的动画实现方式中一个很重要的类是 DoubleAnimaiton，它用来描述一个动画。这些类都属于动画类，其类层次结构如图 16-2 所示。

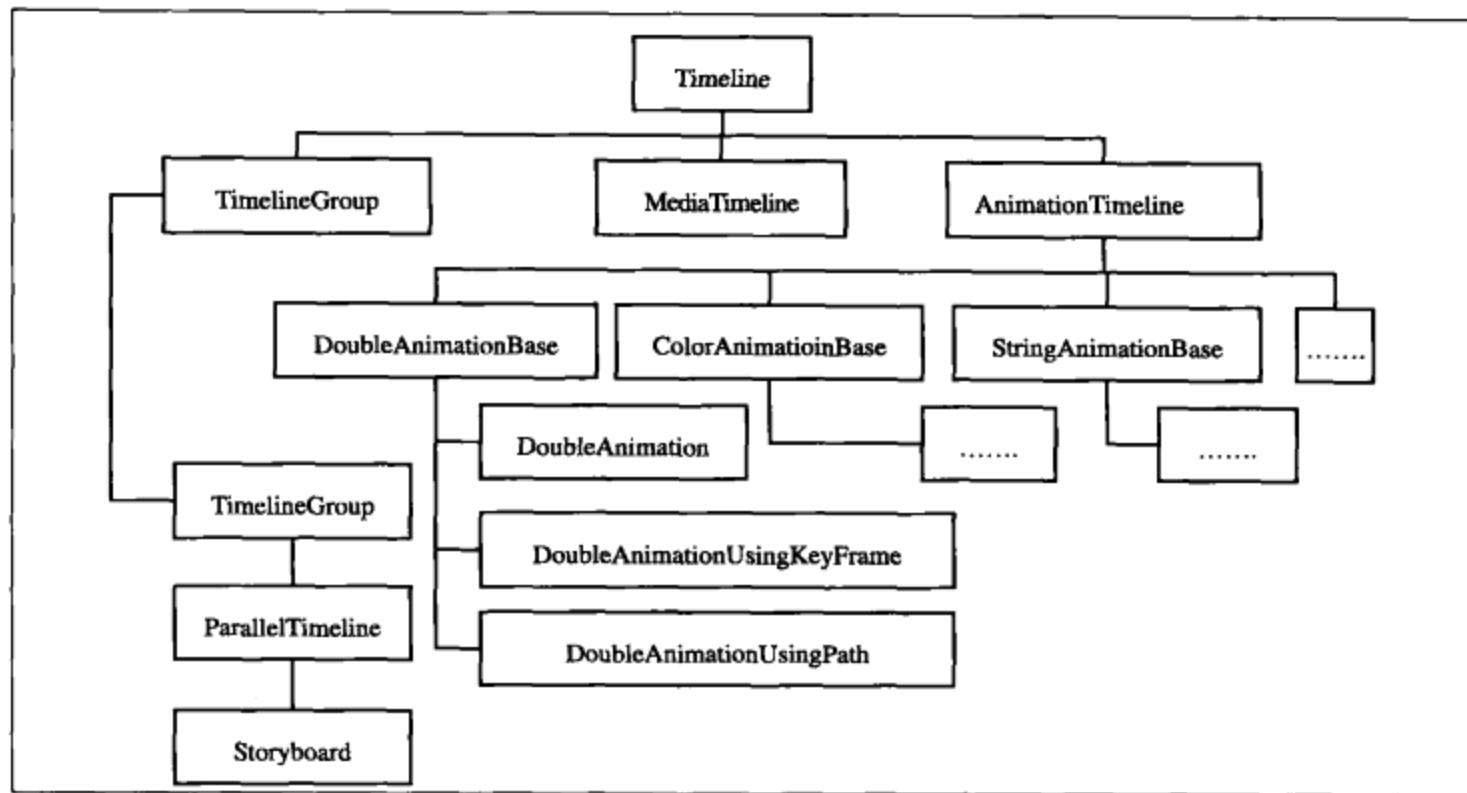


图 16-2 类层次结构

从图中可以看出，所有的动画类都继承自 Timeline，它代表的是一个时间段。其派生了 3 种不同的类型，其中 AnimationTimeline 是各种动画类的基类； MediaTimeline 代表媒体的时间段； TimelineGroup 类型可以集合多个 Timeline，一个很重要的类型 Storyboard 即派生自它。

AnimationTimeline 下面派生了很多种动画类型，WPF 针对一些基本的类型都有相应的动画类，共 22 个。基本上是如果有一个类型为 Type，就会有一个 TypeAnimationBase 类，如表 16-1 所示。

表 16-1 AnimationTimeline 下派生动画类型一览表

核心.NET 数据类型	相应的动画类	WPF 数据类型	相应的动画类
Boolean	BooleanAnimationBase	Thickness	ThicknessAnimationBase
Byte	ByteAnimationBase	Color	ColorAnimationBase
Char	CharAnimationBase	Size	SizeAnimationBase
Decimal	DecimalAnimationBase	Rect	RectAnimationBase
Int16	Int16AnimationBase	Point	PointAnimationBase
Int32	Int32AnimationBase	Point3D	Point3DAnimationBase
Int64	Int64AnimationBase	Vector	VectorAnimationBase
Single	SingleAnimationBase	Vector3D	Vector3DAnimationBase
Double	DoubleAnimationBase	Rotation3D	Rotation3DAnimationBase
String	StringAnimationBase	Matrix	MatrixAnimationBase
Object	ObjectAnimationBase	Quaternion	QuaternionAnimationBase

而每个***AnimationBase 一般来说又会派生如下 3 个不同的类型。

(1) From/To/By 类型的动画：如前面用到的 DoubleAnimation 属于这种类型，它描述的动画一般通过指定起始值 (From) 和终点值 (To) 或者终点值和起点值之间的差值 (By)。这种动画类型默认的命名方式是类型+Animation，如 DoubleAnimation。

(2) KeyFrame 类型的动画通过关键帧来描述一个动画，这里的关键帧和前面所说的基于帧的动画不可混淆。这个动画类型命名方式是类型 +AnimationUsingKeyFrames，如 DoubleAnimationUsingKeyFrames 或者 PointAnimationUsingKeyFrames。

(3) Path 类型的动画通过指定一个路径，让动画沿着路径运动。这种动画类型命名方式是类型 +AnimationUsingPath，如 DoubleAnimationUsingPath 或者 PointAnimationUsingPath。

并不是每种类型都会有这 3 种不同的动画类，如 Color 类型不会有 ColorAnimationUsingPath。



木木学习到此处，不免又有些疑问，为什么要用如此多的动画类，而不是直接使用泛型？换句话说为什么没有一个 Animation<T> 来取代所有的动画类？

主要有如下 3 个原因导致不能采用泛型，而使用这样庞大的动画类。

(1) 在 XAML 中缺乏对泛型的完全支持，这是一个相当显然的理由。

(2) 各种类型的动画实现方式不同，每个类要根据具体数据类型执行计算，一个 Double 和一个 Color 类型的计算完全不同。

(3) Animation<T> 的存在意味着动画可以支持任何数据类型，但事实上动画只能支持有限的类型。如果需要支持更多的类型，则必须扩展使用泛型，但是无法放置某种约束来充分表示支持的类。

16.3.3 时间线的基本行为

要理解动画，首要和重要的是理解时间。动画中的时间和真实时间不同，它总是和时间线 (Timeline) 相联系。Timeline 是所有动画的基类，代表一段动画的时间。零时被定义为起始，如“30 秒”即其开始之后的 30 秒钟。而且具备层次关系，其起始和结束时间均相对于父时间线的起始点来计算。描述一个 Timeline 主要通过以下属性。

(1) Duration

Duration 属性用来描述时间的长短，如前例中表示动画将持续 2.7 秒结束，如代码 16-7 所示。

```
DoubleAnimation doubleanimation = new DoubleAnimation();
doubleanimation.From = 30;
doubleanimation.To = 300;
doubleanimation.Duration = TimeSpan.FromSeconds(2.7);
doubleanimation.FillBehavior = FillBehavior.Stop;
rectangle.BeginAnimation(Rectangle.WidthProperty, doubleanimation);
```

代码 16-7 Duration 属性

在 XAML 文件中可以用时:分:秒.小数格式的字符串来表示时间，如代码 16-8 所示。

```
<DoubleAnimation
    Storyboard.TargetName="rectangle"
```

```
Storyboard.TargetProperty="Width"
From="30" To="300" Duration="0:0:2.7" FillBehavior="Stop" />
```

代码 16-8 用时:分:秒.小数这样格式的字符串来表示时间

在代码 16-9 中通过使用静态的 TimeSpan.Parse 方法也可以使用这样格式的字符串：

```
double animation.Duration = TimeSpan.Parse("0:0:2.7");
```

代码 16-9 静态的 TimeSpan.Parse 方法

Duration 有两个 TimeSpan 无法表达的值，即 Duration.Forever 和 Duration.Automatic。其中 Forever 表示时间为无限期，对于 DoubleAnimation 这样简单的动画，意味着将无限地停留在初始值；Automatic 是任何动画类的 Duration 属性的默认值，一般情况下表示 1 秒。但是在 Timeline 嵌套的情况下含义有所不同，如代码 16-10 所示（详见 mumu_NestedTimeline 工程）。

```
<StackPanel Margin="20">
    <Rectangle Width="100" Height="100" Name="myRectangle">
        <Rectangle.Fill>
            <SolidColorBrush x:Name="MyAnimatedBrush" Color="Black" />
        </Rectangle.Fill>
        <Rectangle.Triggers>
            <EventTrigger RoutedEvent="Rectangle.Loaded">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Storyboard.TargetName="myRectangle"
                            Storyboard.TargetProperty="Height"
                            To="300" Duration="0:0:1" />
                        <DoubleAnimation
                            Storyboard.TargetName="myRectangle"
                            Storyboard.TargetProperty="Width"
                            To="300" Duration="0:0:4" />
                        <ColorAnimation
                            Storyboard.TargetName="MyAnimatedBrush"
                            Storyboard.TargetProperty="Color"
                            To="Yellow" Duration="0:0:2" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Rectangle.Triggers>
    </Rectangle>
</StackPanel>
```

代码 16-10 Duration 属性中 Timeline 嵌套的情况

我们为一个矩形赋 3 个动画，它们均未设置 From 属性，表示起始点都是当前的初始值。第 1 个动画是让矩形的高度从初始值到 300，持续时间为 1 秒；第 2 个动画是让矩形的宽度从初始值到 300，持续时间为 4 秒；第 3 个动画是让矩形的颜色从初始值到黄色，持续时间为 2 秒。3 个动画都作为 ParallelTimeline 的子集，而 ParallelTimeline 属于一个 Storyboard。Storyboard 没有设置 Duration 属性，其默认值为 Automatic。这里的 Automatic 均表示其动画时间会等到最后一个子动画结束而结束，即时间为 4 秒。当 Storyboard 的 Duration 设置小于 4 秒，比如 0.5 秒。它会在所有子动画还没有结束时粗暴地结束所有动画，如图 16-3 所示。

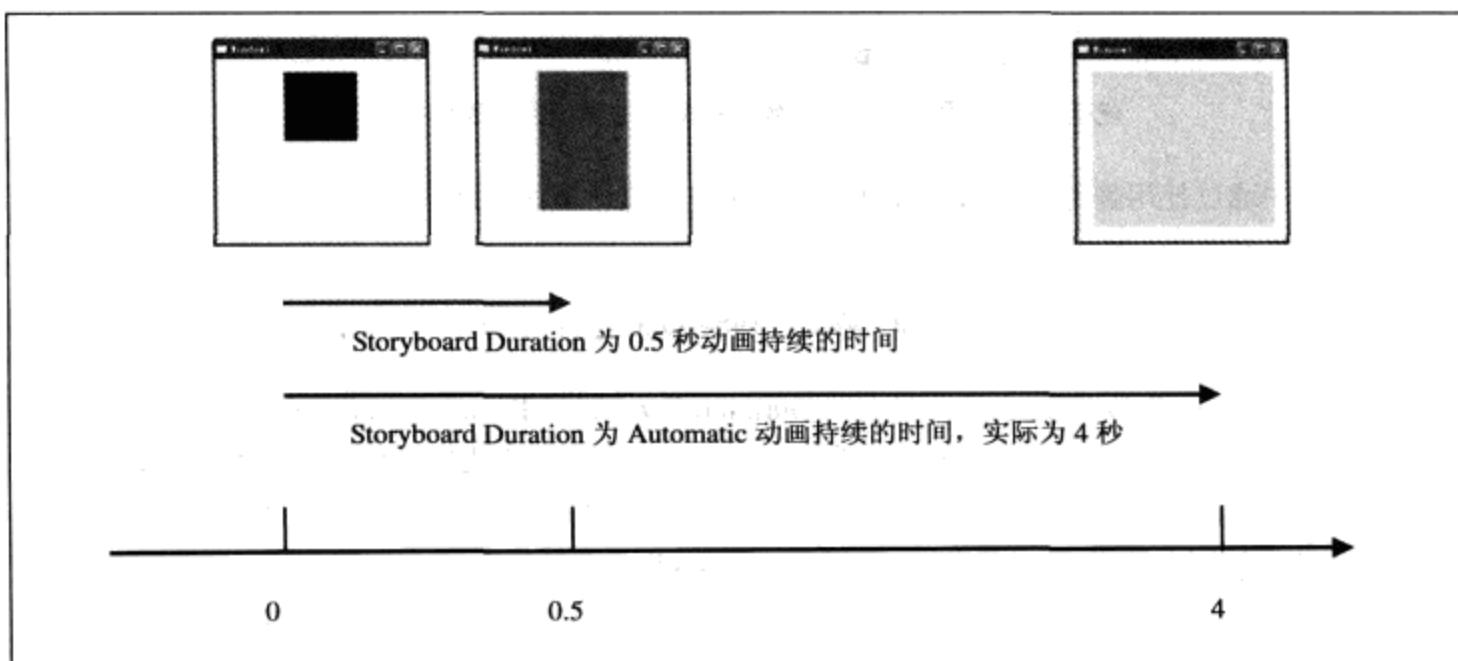


图 16-3 Duration 属性的嵌套示意

(2) BeginTime

如果不希望动画在调用 `BeginAnimation` 之后立即开始，可以设置 `BeginAnimation` 为一个 `TimeSpan` 来插入一段延迟。`BeginTime` 的值相对其父时间线，如设置为 0 秒，表示父时间线和子时间线开始；设置为 2 秒，则表示父时间线开始两秒之后子时间线开始。`BeginTime` 甚至可以设置负值，如 -2 秒，表示子时间线开始于父时间线两秒之前，如代码 16-11 所示（详见 `mumu_timebehaviorsample` 工程）。

```

<!-- defaultBeginTimeRectangle, delayedBeginTimeRectangle,
precededAnimationWithDelayedParentRectangle,
delayedAnimationWithDelayedParentRectangle 四个为应用动画的矩形 -->
<Button Margin="0,30,0,0" HorizontalAlignment="Left">重新开始动画
<Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation
                    Storyboard.TargetName="defaultBeginTimeRectangle"
                    Storyboard.TargetProperty="Width"
                    BeginTime="0:0:0"      From="20"      To="400"      Duration="0:0:2"
                    FillBehavior="HoldEnd" />
                <DoubleAnimation
                    Storyboard.TargetName="delayedBeginTimeRectangle"
                    Storyboard.TargetProperty="Width"
                    BeginTime="0:0:5"      From="20"      To="400"      Duration="0:0:2"
                    FillBehavior="HoldEnd" />
                <DoubleAnimation
                    Storyboard.TargetName="precededAnimationWithDelayedParentRectangle"
                    Storyboard.TargetProperty="Width"
                    BeginTime="-0:0:1"     From="20"      To="400"      Duration="0:0:2" />
                    <ParallelTimeline BeginTime="0:0:5">
                        <DoubleAnimation
                            Storyboard.TargetName="delayedAnimationWithDelayedParentRectangle"
                            Storyboard.TargetProperty="Width"

```

```
    BeginTime="0:0:5"      From="20"       To="400"       Duration="0:0:2"
FillBehavior="HoldEnd" />
        </ParallelTimeline>
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Button.Triggers>
</Button>
```

代码 16-11 BeginTimeExample.xaml 文件

有 4 个蓝色的矩形框，其动画效果都是在两秒钟将宽度从 20 变化到 400，不同的是其 BeginTime 属性。

第 1 个矩形框的 BeginTime 为 0:0:0，单击“重新开始动画”按钮时立即开始动画。

第 2 个矩形框的 BeginTime 为 0:0:5，单击“重新开始动画”按钮 5 秒钟后开始动画。

第 3 个矩形框的 BeginTime 为 -0:0:1，单击“重新开始动画”按钮时其起始长度已经不是 20，而是 $20 + \frac{400-20}{2} = 210$ 。以期给人造成了一种错觉，矩形的动画已经开始了 1 秒。

第 4 个矩形框将一个 DoubleAnimation 嵌套在一个 ParallelTimeline 中，ParallelTimeline 和 DoubleAnimation 的 BeginTime 为 0:0:5，整个动画延迟 10 秒钟后开始。

(3) AutoReverse 和 RepeatBehavior

AutoReverse 属性用来描述动画是否会在动画结束之后从后向前重复一遍，如代码 16-12 所示。

```
<DoubleAnimation Storyboard.TargetName="delayedBeginTimeRectangle"
    Storyboard.TargetProperty="Width"
    BeginTime="0:0:5" From="20" To="400" Duration="0:0:2" FillBehavior=
    "HoldEnd" AutoReverse="True" />
```

代码 16-12 AutoReverse 属性

AutoReverse 默认为 false，如果设置为 True，则这个矩形的宽度会从 20 变到 400，随后再从 400 变回 20。

RepeatBehavior 表示重复的时间，可以有两种设置方式，一是使动画重复一定的次数。如希望动画重复两次，则 RepeatBehavior="2x"，字母 x 表示次数；二是设置动画重复的时间，与设置 Duration 的方式相同。值得注意的是 BeginTime 只有在第 1 次播放动画时会产生延迟，在后面重复时不会产生延迟。

RepeatBehavior 也可以设置为 Forever，表示该动画将永远重复。也可以设置为比 Duration 的时间短，这时它会提前停止动画。

(4) SpeedRatio

SpeedRatio 表示动画快慢的程度，一个小于 1 的值会减慢动画速度；大于 1 的值会加速动画，但是其本身并不影响 BeginTime。例中分别有 3 个矩形框的动画，如代码 16-13 所示。

```

<!-- defaultSpeedRectangle, fasterRectangle, slowerRectangle
三个为应用动画的矩形 -->
    <Button Margin="0,30,0,0" HorizontalAlignment="Left">开始动画
        <Button.Triggers>
            <EventTrigger RoutedEvent="Button.Click">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Storyboard.TargetName="defaultSpeedRectangle"
                            Storyboard.TargetProperty="Width"
                            From="20" To="400" Duration="0:0:2" SpeedRatio="1" />
                        <DoubleAnimation
                            Storyboard.TargetName="fasterRectangle"
                            Storyboard.TargetProperty="Width"
                            From="20" To="400" Duration="0:0:2" SpeedRatio="2" BeginTime="0:0:1" />
                        <DoubleAnimation
                            Storyboard.TargetName="slowerRectangle"
                            Storyboard.TargetProperty="Width"
                            From="20" To="400" Duration="0:0:2" SpeedRatio="0.5" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Button.Triggers>
    </Button>

```

代码 16-13 SpeedExample.xaml 文件

第 1 个矩形框在两秒钟将宽度从 20 变化到 400，由于 SpeedRatio 设置为 1，所以动画的真实时间仍然为两秒钟。

第 2 个矩形框在两秒钟内将宽度从 20 变化到 400，由于 SpeedRatio 设置为 2，因此时间应该为 1 秒钟。再加上延迟时间（BeginTime=0:0:1），因此总时间仍为两秒钟。

第 3 个矩形框前面的设置同第 1 个矩形框，不同是 SpeedRatio 设置为 0.5。因此动画速度减慢，总的动画时间为 4 秒。

(5) 动画的时间线长度属性

动画的时间长度除了 Duration 属性，还与多个属性相关。我们可以用下面的公式来计算时间线的长度：

$$\text{时间线的长度} = \text{BeginTime} + (\frac{\text{Duration} * (\text{AutoReverse} ? 2 : 1)}{\text{SpeedRatio}} * \text{RepeatBehavior})$$

注意 RepeatBehavior 表示的次数，如果表示时间，则整个时间线的长度仅仅是 BeginTime 和 RepeatBehavior 之和。

时间线的长度 = BeginTime + RepeatBehavior

(6) AccelerationRatio 和 DecelerationRatio

默认情况下，动画以线性方式更新目标值。通过改变 AccelerationRatio 和 DecelerationRatio 的值可以使动画变成非线性，从而使动画更加生动。

AccelerationRatio 和 DecelerationRatio 两个属性可以设置为 0~1 范围内的 double 值，0 为默认值。AccelerationRatio 值表示目标值从静止开始加速的时间百分比，DecelerationRatio 值表示目标值减速到静止的时间百分比，因此这两个属性的总和必须小于或者等于 1。

图 16-4 所示为不同的 AccelerationRatio 和 DecelerationRatio 值的含义。

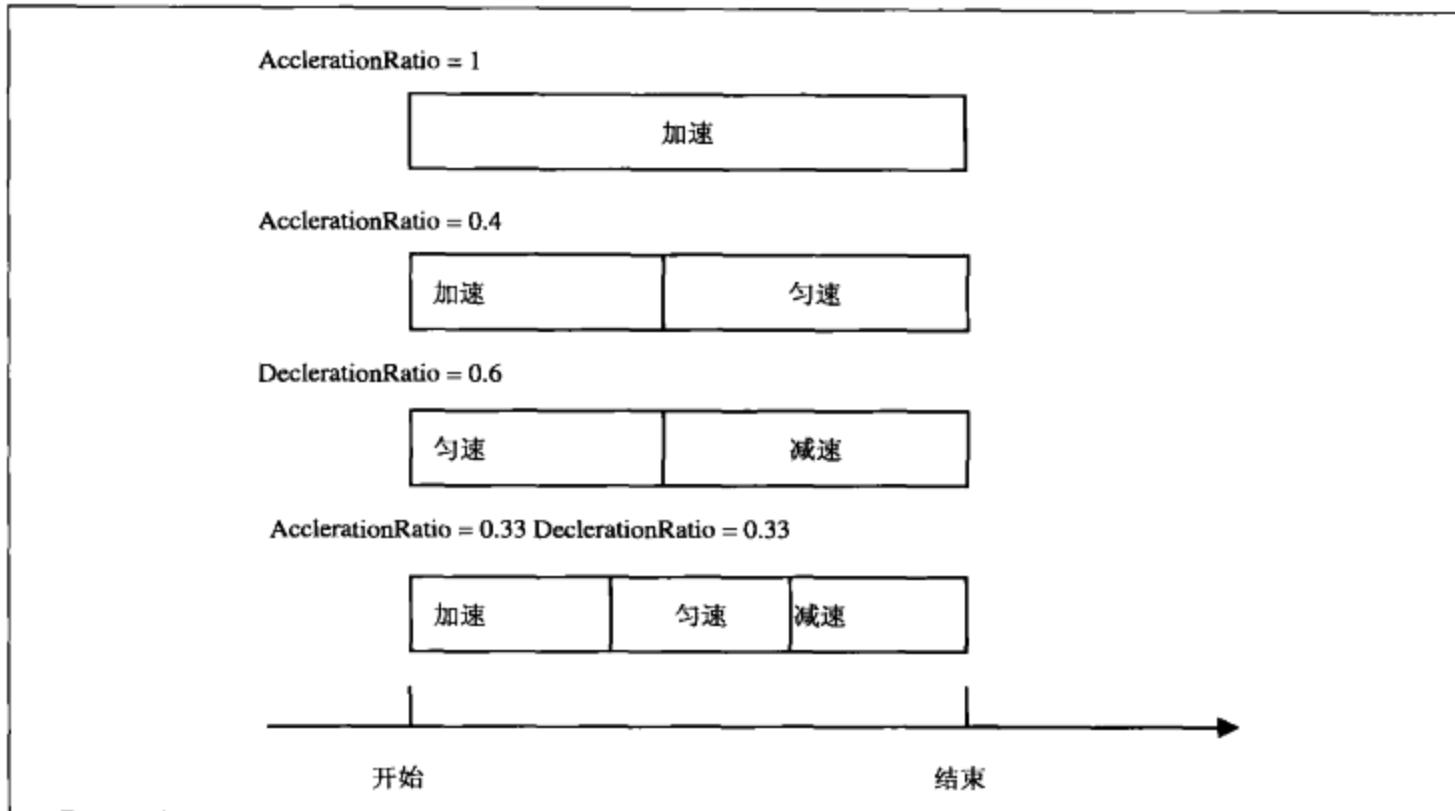


图 16-4 不同 AccelerationRatio 和 DecelerationRatio 值的含义

注意无论如何设置 AccelerationRatio 和 DecelerationRatio 都不会影响整个时间线的长度，如代码 16-14 中的 4 个不同动画的 AccelerationRatio 和 DecelerationRatio 不同，但是均在 10 秒钟到达终点。

```
<DoubleAnimation
    Storyboard.TargetName="nonAcceleratedOrDeceleratedRectangle"
    Storyboard.TargetProperty="Width"
    Duration="0:0:10" From="20" To="400" />
<DoubleAnimation
    Storyboard.TargetName="acceleratedRectangle"
    Storyboard.TargetProperty="Width"
    AccelerationRatio="0.4" Duration="0:0:10" From="20" To="400" />
<DoubleAnimation
    Storyboard.TargetName="deceleratedRectangle"
    Storyboard.TargetProperty="Width"
    DecelerationRatio="0.6" Duration="0:0:10" From="20" To="400" />
<DoubleAnimation
    Storyboard.TargetName="acceleratedAndDeceleratedRectangle"
    Storyboard.TargetProperty="Width"
    AccelerationRatio="0.33" DecelerationRatio="0.33" Duration="0:0:10"
    From="20" To="400" />
```

代码 16-14 4 个不同的动画设置

此外，AccelerationRatio 和 DecelerationRatio 值是时间的百分比，即加速和减速时间占整个时间的百分比。

(7) IsAdditive 和 IsCumulative

IsAdditive 和 IsCumulative 不是 Timeline 的属性，但是绝大多数动画类都有这两个属性。IsAdditive 默认为 false，如果设置为 true，会将目标属性的当前值添加到动画的 From 和 To 属性中。两个矩形框原来的宽度都为 100，动画的初始值都是 100，而终点值是 200。不同是上面的矩形框动画 IsAdditive 为 True，下为 false。

当第 1 次单击“开始动画”按钮时，上面的矩形框实际上是从 200 变化到 300，原因是把当前的宽度值加入到 From 和 To 属性中；第 2 次单击时 IsAdditive 为 false 的矩形框仍然将上次的动画运行一遍。但是 IsAdditive 为 true 的矩形框的宽度会从 300 变到 500，如图 16-5 所示。

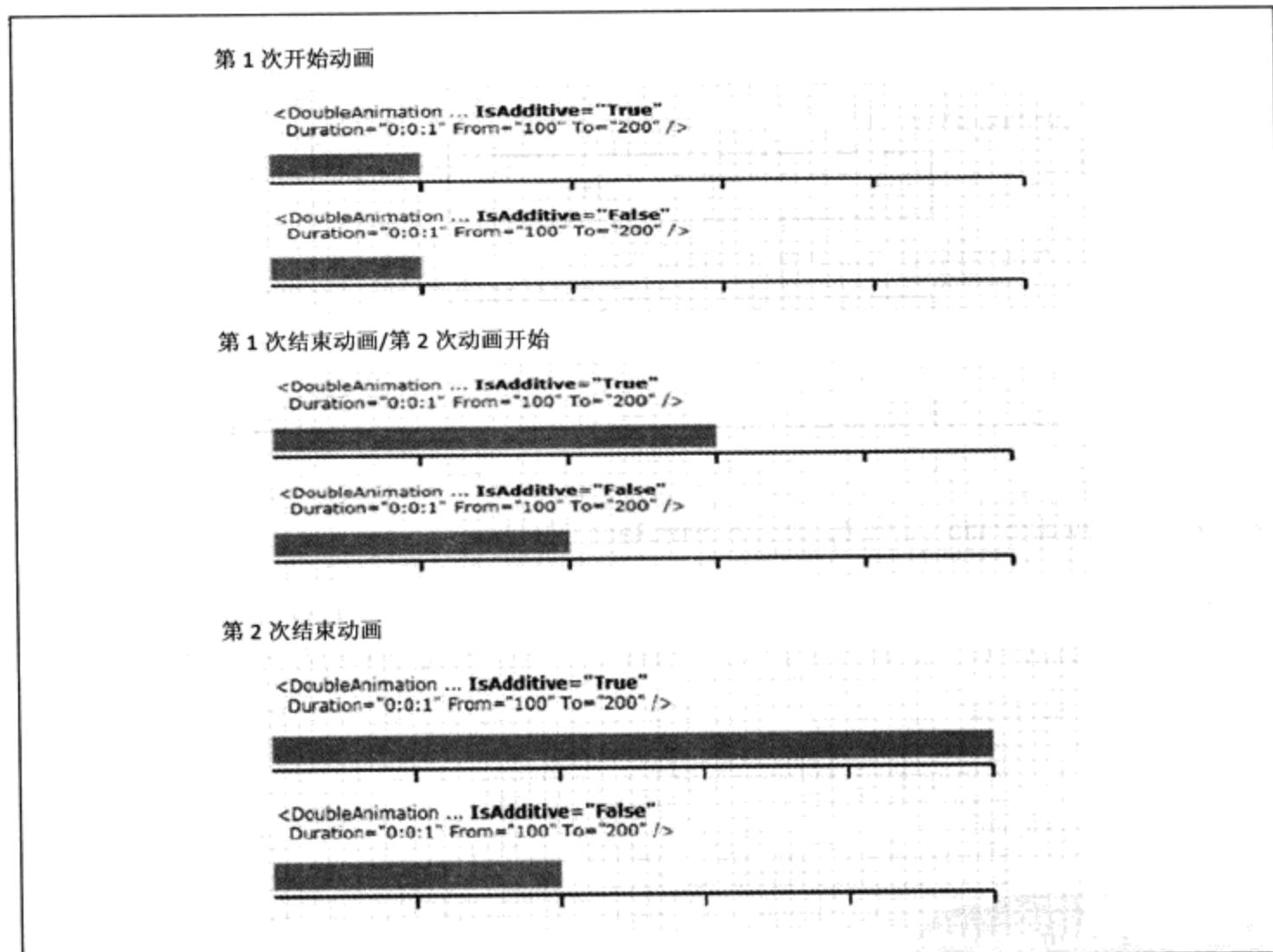


图 16-5 重复两次矩形框的动画效果

IsCumulative 和 IsAdditive 的不同在于与 RepeatBehavior 一起使用，而且仅能与 RepeatBehavior 一起使用。如两个矩形框的宽度值在 1 秒从 100 变到 200。重复两次且 AutoReverse 为 true，如图 16-6 所示。

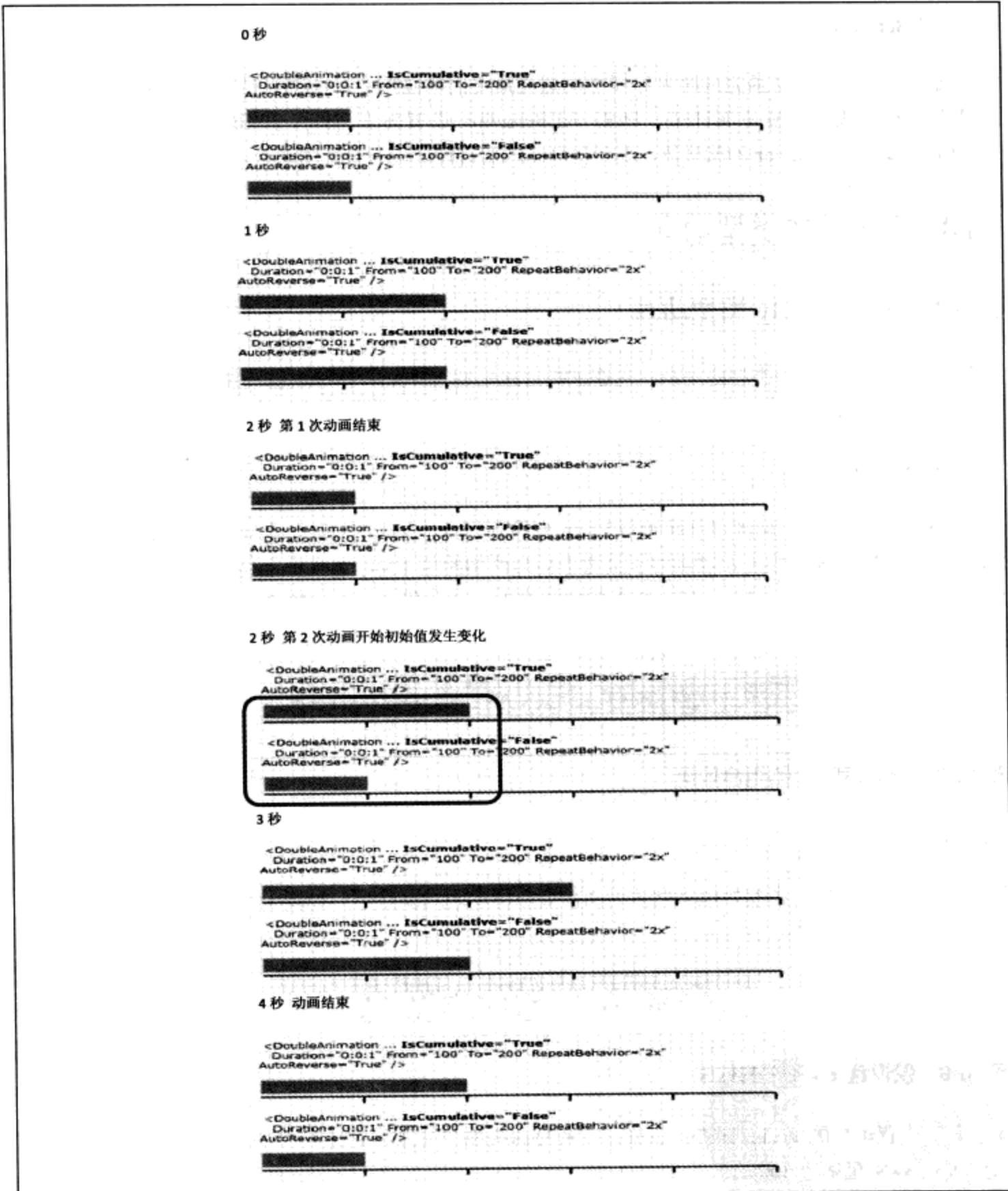


图 16-6 重复两次矩形框的动画效果

在第 1 次动画结束，第 2 次动画开始时应用 `IsCumulative` 为 `true` 的矩形框宽度的初始值由 100 变为 200，可以看到第 1 个矩形框的宽度突然跳动一下。再次单击“开始动画”按钮，则与 `IsAdditive` 不同。即在现有的矩形框的宽度上再开始动画，而是恢复了两个矩形框的最初值。如果未设置 `RepeatBehavior`，两个矩形框的最后宽度将相同。

(8) FillBehavior

默认情况下，当一个动画完成时目标属性仍然在最终的动画值上，通常也是期望的结果。如果希望属性在动画完成后恢复原属性值，则将 FillBehavior 设置为 Stop。FillBehavior 的另一个值为 HoldEnd，即 FillBehavior 的默认值，表示动画完成后目标属性为最终动画值。

16.4 3 种基本类型动画

16.4.1 From/To/By 类型动画

前面已经多次用到这种类型的动画，它通过指定起始值（From）和终点值（To）或者终点值和起点值之间的差值（By）来描述一个动画，From/To/By 有多种组合用法。

(1) From 和 To 组合

这是一种常见的组合，动画将从起始值 From 变换到终点值（To）。如一个矩形的宽度为 100，设置 From=50 且 To=300，则开始动画时不会考虑矩形的初始值 100，宽度将直接由 50 连续变为 300 后结束动画。图 16-7 中，橘红色代表矩形的原始宽度，蓝色代表其动画轨迹。

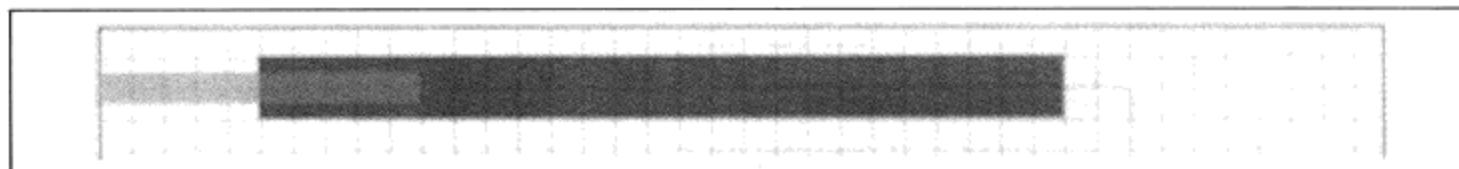


图 16-7 From 和 To 的组合效果

(2) To

如果仅设置 To 属性，则起始值将默认为当前要应用动画的属性值，如图 16-8 所示。

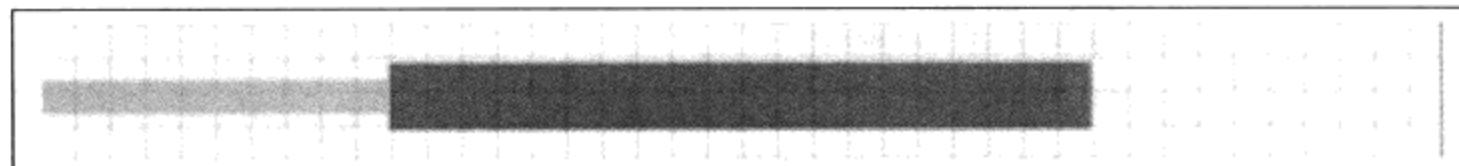


图 16-8 仅设置 To 属性的效果

注意必须设置矩形的 Width 属性；否则运行程序时会抛出异常，这是因为 Width 默认值为 NaN，动画无法从 NaN 变化为 300。

(3) By

By 代表一个差值，如果只设置该属性，那么起始值也将默认为当前要应用动画的属性值。仍然是宽度为 100 的矩形；设置 By=300，那么开始动画时宽度将从 100 连续变为 400，如图 16-9 所示。

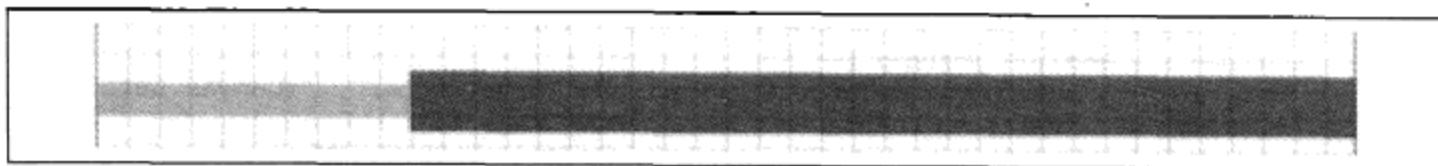


图 16-9 仅设置 By 属性的效果

(4) From 和 By 组合

通过该组合也可以指定起始值和终点值，如宽度为 100 的矩形设置 From=50 且 By=300。开始动画，宽度将从 50 连续变为 350，如图 16-10 所示。

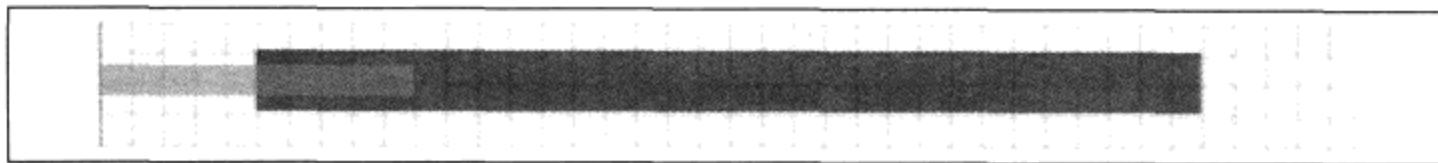


图 16-10 From 和 By 的组合效果

(5) From

如果仅设置 From 属性，那么终点值将默认为当前要应用动画的属性值。如将宽度为 100 的矩形的 From 设置为 50，开始动画，宽度将从 50 连续变到 100，如图 16-11 所示。



图 16-11 仅设置 From 属性的效果

(6) To 和 By 组合

如果同时设置 To 和 By，那么将忽略 By 属性。

16.4.2 KeyFrame 类型动画

1. 简单的 KeyFrame 动画

From/To/By 类型的动画仅支持从一个值到另一个值的线性内插，或者有限形式的非线性内插（AccelerationRatio 和 DecelerationRatio）。如果要描述一个更复杂的动画，如希望一个动画的属性从 A 值到 B 值再到 C 值，则 Key frame 动画。它在指定的时间提供指定的值，如代码 16-15 所示。

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"      Width="700"
      Height="400">
    <Canvas Background="{StaticResource MyGridBrushResource}">
        <Ellipse Name="elips"
                Width="48" Height="48" Fill="Red"
                Canvas.Left="480" Canvas.Top="96" />
    <Canvas.Triggers>
```

```

<EventTrigger RoutedEvent="Canvas.Loaded">
    <BeginStoryboard>
        <Storyboard TargetName="elips" TargetProperty="(Canvas.Left)">
            <DoubleAnimationUsingKeyFrames RepeatBehavior="Forever"
                Duration="0:0:10">
                <LinearDoubleKeyFrame KeyTime="0:0:5" Value="0" />
                <LinearDoubleKeyFrame KeyTime="0:0:5.5" Value="48" />
                <DiscreteDoubleKeyFrame KeyTime="0:0:6" Value="144" />
                <DiscreteDoubleKeyFrame KeyTime="0:0:7" Value="240" />
            </DoubleAnimationUsingKeyFrames>
        </Storyboard>
    </BeginStoryboard>
</EventTrigger>
</Canvas.Triggers>
</Canvas>
</Page>

```

代码 16-15 简单的 KeyFrame 动画示例

球的开始位置是(480,96)，这是通过 Canvas.Left 和 Canvas.Top 两个附加属性设置的。动画改变 Canvas.Left 属性值，为第 1 个 LinearDoubleKeyFrame 表示在动画的前 5 秒 Canvas.Left 属性值从 480 改变为 0，意味着球会匀速地向左运动直到 Canvas 的最左侧；第 2 个 LinearDoubleKeyFrame 表示在接下来的 0.5 秒 Canvas.Left 属性值从 0 改变为 48，意味着球会向右匀速运动；第 3 个 KeyFrame 是一个 DiscreteDoubleKeyFrame，表示在 0.5 秒后，Canvas.Left 属性值会从 48 直接变化为 144；第 4 个 DiscreteDoubleKeyFrame，表示 Canvas.Left 会在下秒直接变化为 240。即球在 6 秒时突然由横坐标 48 的位置突然跳到 144 的位置，然后在 7 秒时突然从 144 的位置跳到 240 的位置。



TargetProperty 如何设置附加属性？

如果 TargetProperty 的值是附加属性，需要添加小括号 TargetProperty="(Canvas.Left)"。实际上也不仅是附加属性，如前面设置 Width 属性时写成 TargetProperty="(Rectangle.Width)"，也需要使用括号，只不过普通属性大多没有必要这样写而已。

正如上看到的，关键帧中有两个重要属性，即 KeyTime 和 Value。关键帧通过这两个属性来共同描述属性值在某一个时间点（KeyTime）要达到某一个值（Value），从而实现关键帧动画。除了可以直接设置时间值以外，还可以以百分比的形式来设置 KeyTime。上例也可以改写为代码 16-16。

```

<DoubleAnimationUsingKeyFrames RepeatBehavior="Forever"
    Duration="0:0:10">
    <LinearDoubleKeyFrame KeyTime="50%" Value="0" />
    <LinearDoubleKeyFrame KeyTime="55%" Value="48" />
    <DiscreteDoubleKeyFrame KeyTime="60%" Value="144" />
    <DiscreteDoubleKeyFrame KeyTime="70%" Value="240" />
</DoubleAnimationUsingKeyFrames>

```

代码 16-16 关键帧的 KeyTime 属性

KeyTime 提供了两个特殊值，其中 Uniform 表示关键帧将会等分这些时间，如代码 16-17 所示。

```

<DoubleAnimationUsingKeyFrames Duration="0:0:10" RepeatBehavior=
"Forever">
    <LinearDoubleKeyFrame Value="100" KeyTime="Uniform" />
    <LinearDoubleKeyFrame Value="200" KeyTime="Uniform" />
    <LinearDoubleKeyFrame Value="500" KeyTime="Uniform" />

```

```
<LinearDoubleKeyFrame Value="600" KeyTime="Uniform" />
</DoubleAnimationUsingKeyFrames>
```

代码 16-17 KeyTime 中的特殊值 Uniform

实际上等价于代码 16-18。

```
<DoubleAnimationUsingKeyFrames Duration="0:0:10" RepeatBehavior=
"Forever">
    <LinearDoubleKeyFrame Value="100" KeyTime="0:0:2.5" />
    <LinearDoubleKeyFrame Value="200" KeyTime="0:0:5" />
    <LinearDoubleKeyFrame Value="500" KeyTime="0:0:7.5" />
    <LinearDoubleKeyFrame Value="600" KeyTime="0:0:10" />
</DoubleAnimationUsingKeyFrames>
```

代码 16-18 Uniform 所等价的 KeyTime 值

另一个值是 Paced，确保属性将以恒等的速率变化。即 KeyTime 会根据 Value 的插值来确定时间，如代码 16-19 所示。

```
<DoubleAnimationUsingKeyFrames Duration="0:0:10" RepeatBehavior=
"Forever">
    <LinearDoubleKeyFrame Value="100" KeyTime="Paced" />
    <LinearDoubleKeyFrame Value="200" KeyTime="Paced" />
    <LinearDoubleKeyFrame Value="300" KeyTime="Paced" />
    <LinearDoubleKeyFrame Value="500" KeyTime="Paced" />
</DoubleAnimationUsingKeyFrames>
```

代码 16-19 KeyTime 中的特殊值 Paced

实际上等价于代码 16-20。

```
<DoubleAnimationUsingKeyFrames Duration="0:0:10" RepeatBehavior=
"Forever">
    <LinearDoubleKeyFrame Value="100" KeyTime="0:0:2" />
    <LinearDoubleKeyFrame Value="200" KeyTime="0:0:4" />
    <LinearDoubleKeyFrame Value="300" KeyTime="0:0:6" />
    <LinearDoubleKeyFrame Value="500" KeyTime="0:0:10" />
</DoubleAnimationUsingKeyFrames>
```

代码 16-20 Paced 所等价的 KeyTime 值

2. 3 种不同类型的关键帧

在这个例子中我们已经看到了两种不同类型的关键帧，即线性关键帧（Linear***KeyFrame²）和离散关键帧（Discrete***KeyFrame）。还有一种是样条关键帧（Spline***KeyFrame），3 种不同类型的关键帧的主要差别在于差值的方法不同。

Linear***KeyFrame 采用线性插值的方法，如一个动画在 5 秒中从起始值 0 变化为终点值 10，如图 16-12 和表 16-2 所示。

² *** 表示类型，如 double 及 int 等。

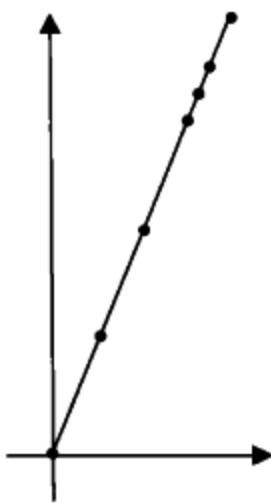


图 16-12 时间和值

表 16-2 时间和值

时间	值
0	0
1	2
2	4
3	6
4	8
4.25	8.5
4.5	9
5	10

Discrete***KeyFrame 是一种离散插值的方式，实际上可能不能称为“插值”，而只是在某一个时间时值到达指定位置。如果上述例子采用离散插值的方式，则如表 16-3 所示。

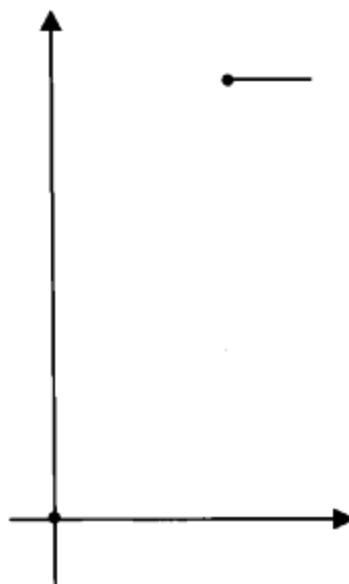


图 16-13 时间和值

表 16-3 离散插值法中时间和值

时间	值
0	0
1	0
2	0
3	0
4	0
4.25	0
4.5	0
5	10

Spline***KeyFrame 采用的是样条插值的方式，也是这里面最为复杂的方式。在奇妙的二维图形世界——面壁一章里，我们已经接触过贝塞尔曲线，该曲线是由起始点，终点和两个控制点描述的样条插值，可以使属性的内插值沿该曲线移动，Spline***KeyFrame 和前两种关键帧有所不同，除了提供 KeyTime 和 Value 属性以外，还提供了一个 KeySpline 属性。该属性是用来定义两个控制点参数的。

在 mumu_animationtype 工程中我们提供了一个可以交互修改这两个控制点参数以控制样条插值的示例，如图 16-14 所示的右侧面板中通过修改红色小球的横坐标（Canvas.Left）使其水平运动；左侧面板中通过 4 个 Slider 控件来修改控制点的值。在 (0~1) 左侧下方的坐标图中横轴表示时间轴，纵轴表示红色小球的横坐标值。

当 KeySpline=“0.00,0.00 0.00,0.00”，样条曲线是一条直线。红色球的横坐标是一个线性变化，因此从图中可以看出红色小球是匀速运动的。

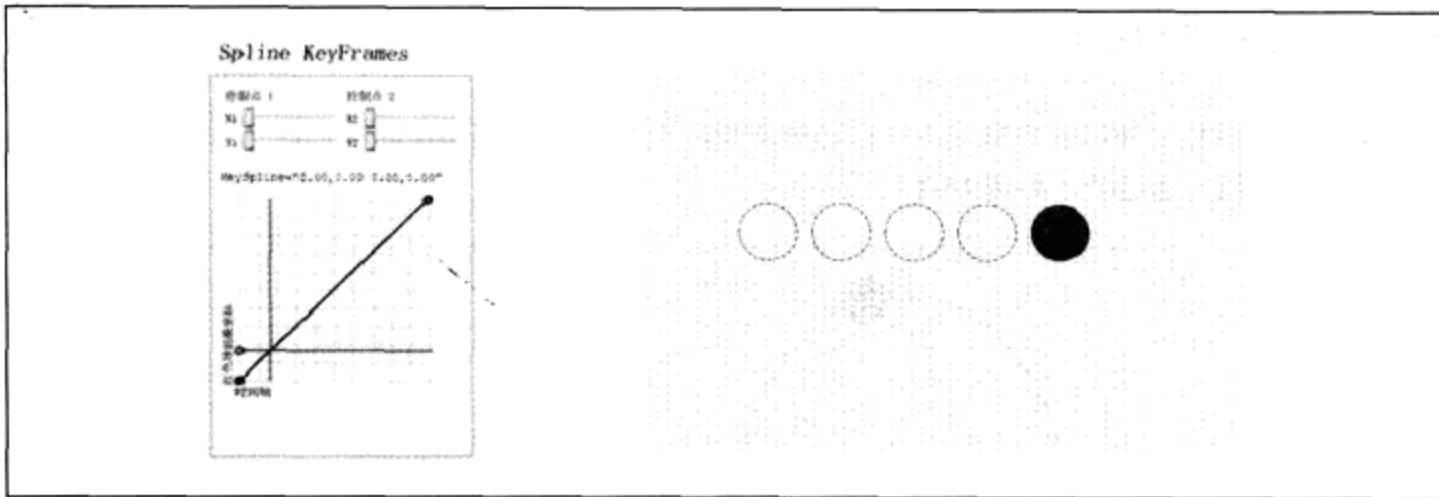


图 16-14 可以交互修改这两个控制点参数从而控制样条插值的示例

当 KeySpline=“0.00,0.00 1.00,0.00”，样条曲线如图 16-15 所示。红色小球最初运动会非常缓慢，随后会越来越快。

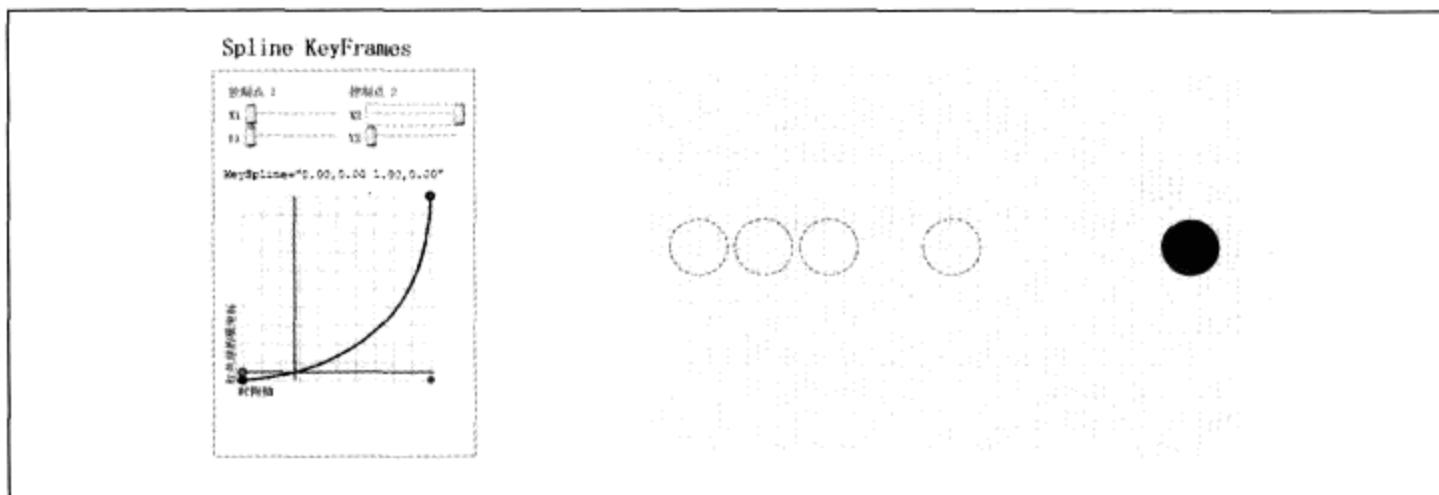


图 16-15 KeySpline=“0.00,0.00 1.00,0.00”时的样条曲线效果

如果 KeySpline=“0.00,1.00 1.00,0.00”，样条曲线如图 16-16 所示。红色小球在两端运动会非常迅速，而中间会非常缓慢。

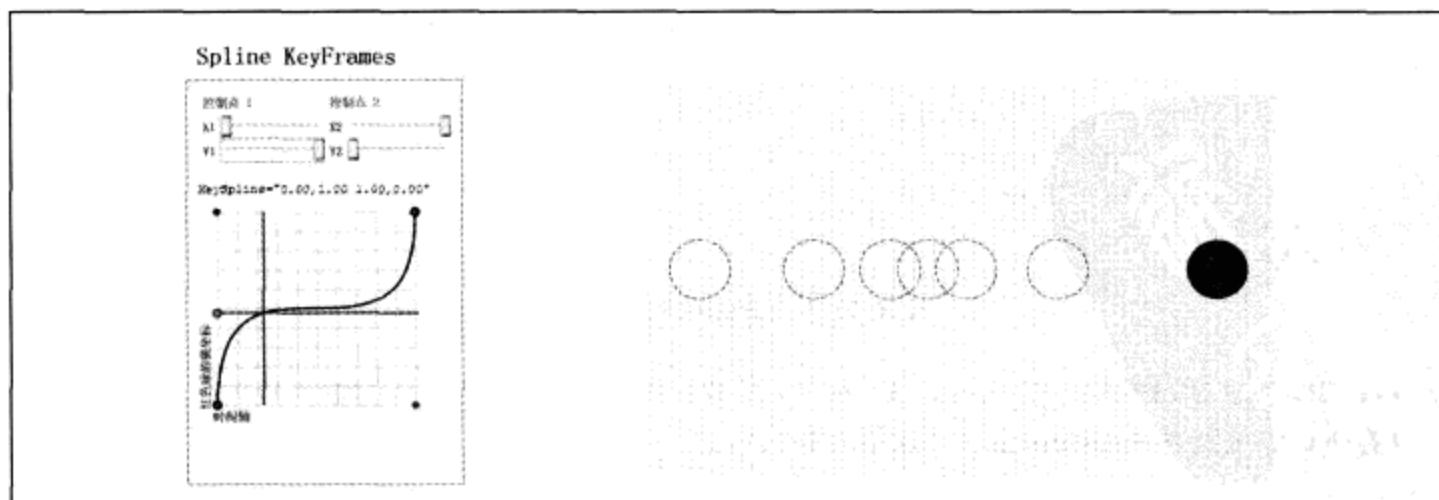


图 16-16 KeySpline=“0.00,1.00 1.00,0.00”时的样条曲线效果图

16.4.3 Path 类型动画

Path 类型的动画是一种可以沿指定路径运动的动画，先看一个简单的动画例子中一个蓝色的小球将会沿着一个路径匀速运动，如图 16-17 所示。

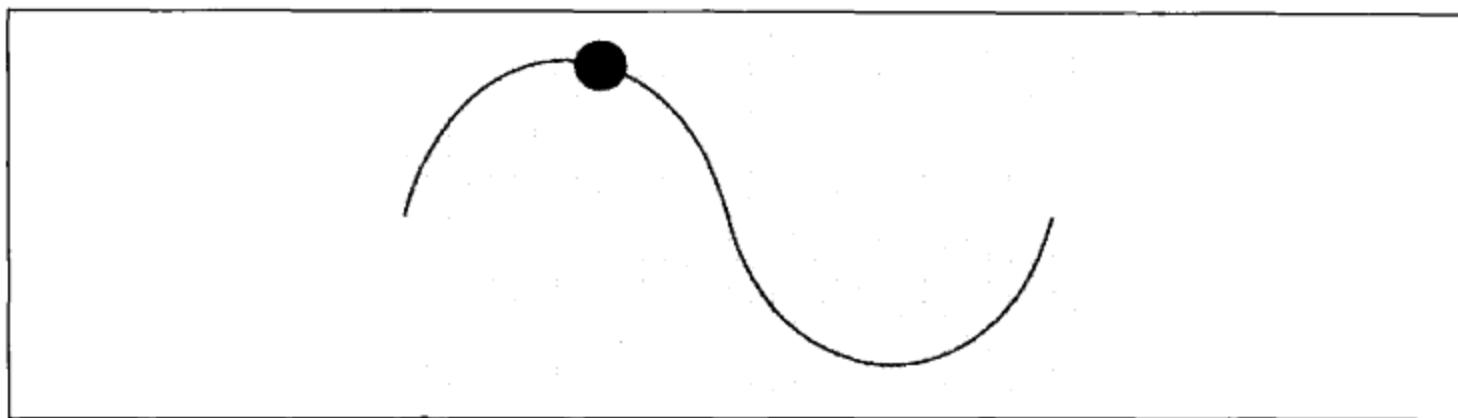


图 16-17 Path 类型的动画示例

如代码 16-21 所示。

```
<Canvas xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!--运动轨迹-->
    <Path Stroke="Black" StrokeThickness="1" Data="M 96 288 C 576 0, 0 0, 480 288" />
    <!--蓝色小球-->
    <Path Fill="Blue">
        <Path.Data>
            <EllipseGeometry x:Name="elips" RadiusX="12" RadiusY="12" />
        </Path.Data>
        <Path.Triggers>
            <EventTrigger RoutedEvent="Path.Loaded">
                <BeginStoryboard>
                    <Storyboard TargetName="elips" TargetProperty="Center">
                        <PointAnimationUsingPath Duration="0:0:2.5"
                            AutoReverse="True"
                            RepeatBehavior="Forever">
                            <PointAnimationUsingPath.PathGeometry>
                                <PathGeometry
                                    Figures="M 96 288 C 576 0, 0 0, 480 288" />
                            </PointAnimationUsingPath.PathGeometry>
                        </PointAnimationUsingPath>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Path.Triggers>
    </Path>
</Canvas>
```

代码 16-21 Path 类型的动画

From/To/By 类型动画的输入是 From、To 和 By 参数；KeyFrame 类型动画的输入是关键帧；而 Path 类型的动画输入则是 PathGeometry，这是其最大特点。

我们也可以将这个蓝色小圆换成任意一个图片，如“飞机”，如代码 16-22 所示。

```

<Image Source="飞机.png" Width="30" Height="30" x:Name="img"
Canvas.Left="0" Canvas.Top="0">
    <Image.Triggers>
        <EventTrigger RoutedEvent="Path.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimationUsingPath Storyboard.TargetName= "img"
Storyboard.TargetProperty="(Canvas.Top)" Duration="0:0:2.5"
                        AutoReverse="True"
                        RepeatBehavior="Forever"
                        Source="Y" >
                        <DoubleAnimationUsingPath.PathGeometry>
                            <PathGeometry
                                Figures="M 10,100 C 35,0 135,0 160,100 180,190
285,200 310,100" />
                        </DoubleAnimationUsingPath.PathGeometry>
                    </DoubleAnimationUsingPath>
                    <DoubleAnimationUsingPath Storyboard.TargetName= "img"
Storyboard.TargetProperty="(Canvas.Left)" Duration="0:0:2.5"
                        AutoReverse="True"
                        RepeatBehavior="Forever"
                        Source="X" >
                        <DoubleAnimationUsingPath.PathGeometry>
                            <PathGeometry
                                Figures="M 10,100 C 35,0 135,0 160,100 180,190
285,200 310,100" />
                        </DoubleAnimationUsingPath.PathGeometry>
                    </DoubleAnimationUsingPath>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Image.Triggers>
</Image>

```

代码 16-22 Path 类型的动画中蓝色小圆和图片的更替

与前例不同的是 Image 的坐标位置由 Canvas.Left 和 Canvas.Top 两个附加属性决定，因此采用的动画类型不是 PointAnimationUsingPath，而是 DoubleAnimationUsingPath。使用 DoubleAnimationUsingPath 时还需要多指定一个 Source 属性，表明这个动画是应用在 X 轴、Y 轴还是角度（Angle）上。运行这个示例，结果如图 16-18 所示。飞机沿着这个轨迹运动，但是美中不足的机头没有始终朝着轨迹方向。

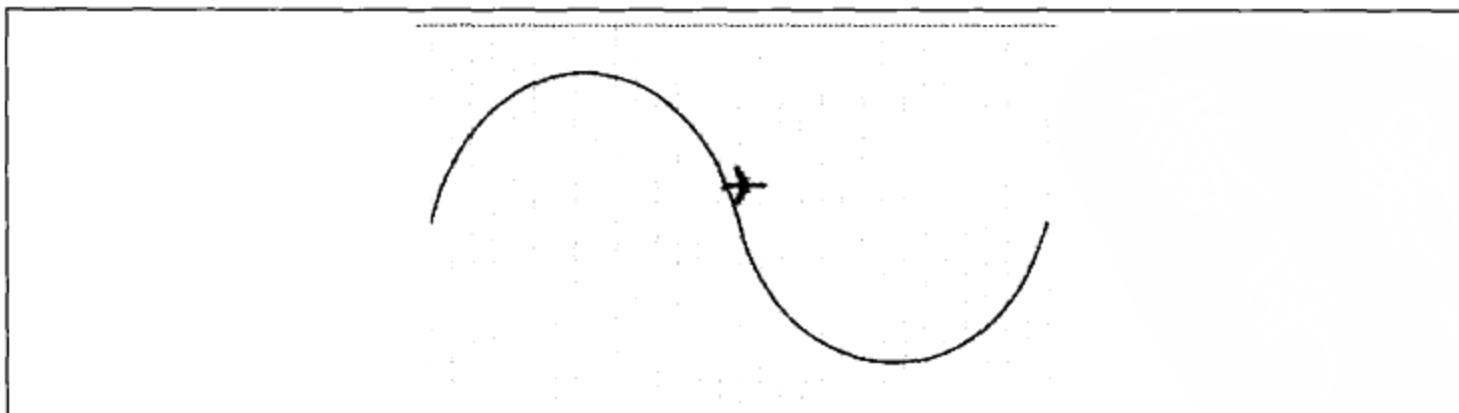


图 16-18 运行结果

`DoubleAnimationUsingPath` 的 `Source` 属性可以指定为 `X`、`Y` 或者 `Angle`，因此角度也可以随路径变化而变化，但是使用 `Canvas.Left` 和 `Canvas.Top` 两个附加属性无法指定角度。可以通过变换来指定图片的位置和角度，如代码 16-23 所示。

```
<Image Source="飞机.png" Width="30" Height="30" >
    <Image.RenderTransform>
        <TransformGroup>
            <RotateTransform x:Name="MyRotateTransform" Angle="0"
CenterX="15" CenterY="15"/>
            <TranslateTransform x:Name="MyTranslateTransform" X="10" Y="100" />
        </TransformGroup>
    </Image.RenderTransform>
</Image>
```

代码 16-23 通过变换来指定图片的位置和角度

这时不再对 `Canvas.Left` 和 `Canvas.Top` 两个附加属性应用动画，而是对两个变换的 `Angle`、`X` 和 `Y` 属性应用动画，如代码 16-24 所示。

```
<BeginStoryboard Name="MyBeginStoryboard">
    <Storyboard>
        <!-- 让角度沿路径运动-->
        <DoubleAnimationUsingPath
Storyboard.TargetName="MyRotateTransform"
Storyboard.TargetProperty="Angle"
Source="Angle"
Duration="0:0:5"
RepeatBehavior="Forever" AutoReverse="True" >
            <DoubleAnimationUsingPath.PathGeometry>
                <PathGeometry Figures="M 10,100 C 35,0 135,0 160,100
180,190 285,200 310,100" />
            </DoubleAnimationUsingPath.PathGeometry>
        </DoubleAnimationUsingPath>

        <!-- 让 X 坐标沿路径运动。-->
        <DoubleAnimationUsingPath
Storyboard.TargetName="MyTranslateTransform"
Storyboard.TargetProperty="X"
Source="X"
Duration="0:0:5"
RepeatBehavior="Forever" AutoReverse="True" >
            <DoubleAnimationUsingPath.PathGeometry>
                <PathGeometry Figures="M 10,100 C 35,0 135,0
160,100 180,190 285,200 310,100" />
            </DoubleAnimationUsingPath.PathGeometry>
        </DoubleAnimationUsingPath>

        <!-- 让 Y 坐标沿路径运动。-->
        <DoubleAnimationUsingPath
Storyboard.TargetName="MyTranslateTransform"
Storyboard.TargetProperty="Y"
Source="Y"
Duration="0:0:5"
RepeatBehavior="Forever" AutoReverse="True" >
            <DoubleAnimationUsingPath.PathGeometry>
                <PathGeometry Figures="M 10,100 C 35,0 135,0
160,100 180,190
285,200 310,100" />
            </DoubleAnimationUsingPath.PathGeometry>
        </DoubleAnimationUsingPath>
    </Storyboard>
</BeginStoryboard>
```

```
</DoubleAnimationUsingPath>
</Storyboard>
</BeginStoryboard>
```

代码 16-24 对两个变换的 Angle、X 和 Y 属性应用动画

运行程序，飞行轨迹如图 16-19 所示。

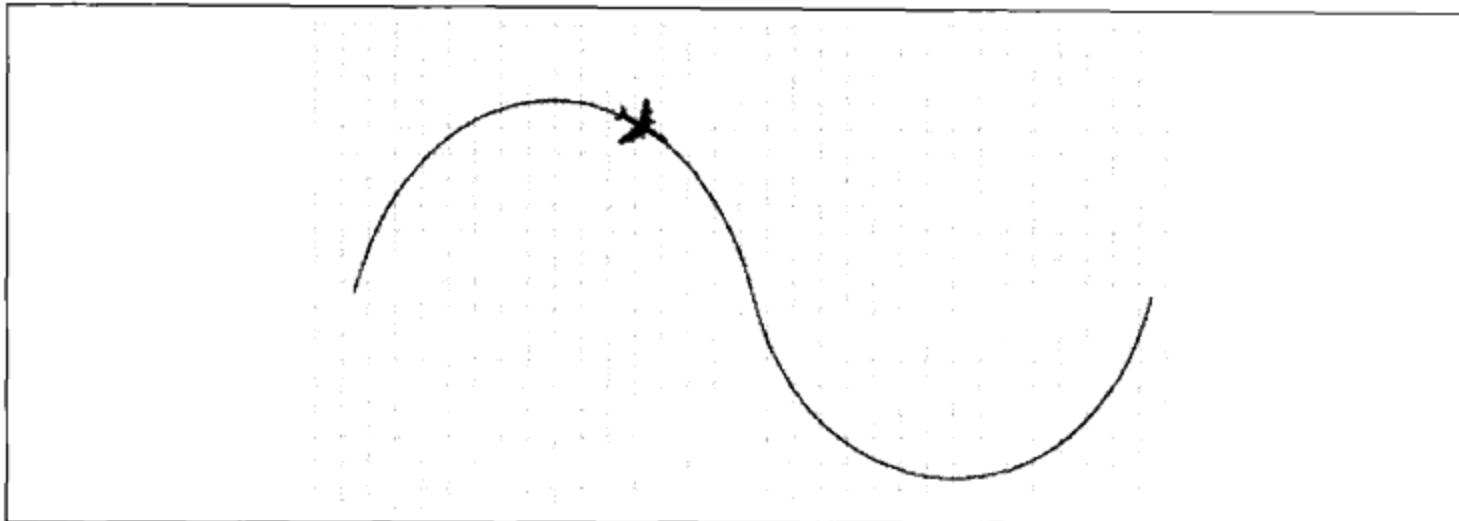


图 16-19 飞机飞行轨迹图

16.5 动画的交互控制

前面的所有动画都有一个重要的缺失，即仅仅能够触发动画开始，却不能暂停或者重新开始动画。前面已经接触过 `BeginStoryboard` 这样的行为（Action），为了实现动画的交互控制，WPF 提供了 `PauseStoryboard`, `ResumeStoryboard` 等控制动画的行为。

以前面的飞机动画为例，首先为其添加控制按钮。注意这些按钮需要为 `Name` 属性赋值，如代码 16-25 所示。

```
<Button Name="BeginButton">开始</Button>
<Button Name="PauseButton">暂停</Button>
<Button Name="ResumeButton">恢复</Button>
<Button Name="StopButton">停止</Button>
```

代码 16-25 为 Name 属性赋值

并且为原来 `BeginStoryboard` 添加一个 `Name` 属性，如代码 16-26 所示。

```
<BeginStoryboard Name="MyBeginStoryboard">
```

代码 16-26 为原来的 `BeginStoryboard` 添加一个 `Name` 属性

添加另外 3 个 `EventTrigger`，其 `SourceName` 属性分别对应相应的按钮名称。`PauseStoryboard`、`ResumeStoryboard` 和 `StopStoryboard` 都需要指定 `BeginStoryboardName` 属性，该属性值为刚刚添加的 `BeginStoryboard` 名。注意这 4 个 `EventTrigger` 需要添加到按钮的上一个节点，这里是 `StackPanel` 对

象。这样可以防止 PauseStoryboard 等找不到 BeginStoryboard 对象，如代码 16-27 所示。

```
<StackPanel Orientation="Horizontal" Margin="40">
    <StackPanel.Triggers>
        <EventTrigger RoutedEvent="Button.Click" SourceName="BeginButton">
            <BeginStoryboard Name="MyBeginStoryboard">
                ...
            </BeginStoryboard>
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="PauseButton">
            <PauseStoryboard BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="ResumeButton">
            <ResumeStoryboard BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="StopButton">
            <StopStoryboard BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
    ...

```

代码 16-27 添加另外 3 个 EventTrigger

如果是在代码动画中，相应地需要使用 Storyboard 的 Pause 及 Resume 等方法。

16.6 后记：降龙的最后一掌

为了使 17 掌的图片形成一个连续的动画，木木将 17 张图片按照 1~17 的顺序叠放在面板上，然后依次将图片的透明度由 1 变化到 0。当第 1 张图片的透明度变为 0 时显示第 2 张图片，这样形成了一幅连续的动画。第 18 张图片是前 17 张的一张合成图，当第 17 张图片慢慢淡去，最后的一掌是“山高月小，水落石出”，如图 16-20 所示。

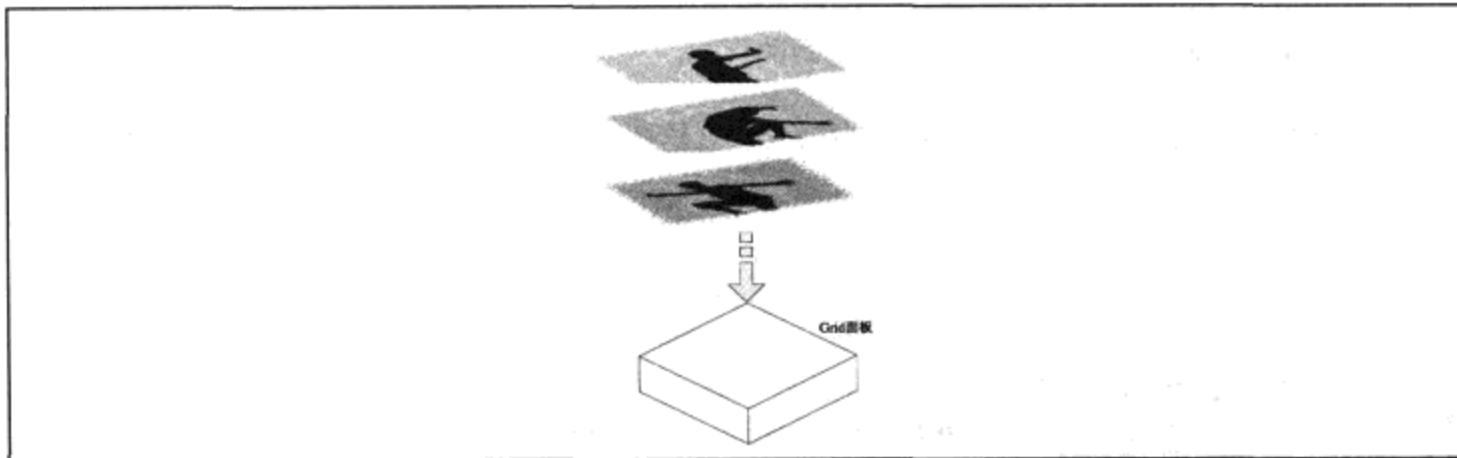


图 16-20 最后一掌

首先建立一个窗口，计划在 Grid 面板的第 0 行第 0 列放置降龙十八掌的每一掌的插图。然后在下方放置控制动画的按钮，以便反复查看，如代码 16-28 所示（详见 mumu_xl18 工程）。

```
<Window x:Class="mumu_xl18.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="降龙十八掌" Height="300" Width="300" Loaded="Window_Loaded">
    <Grid x:Name="grid">
        <Grid.RowDefinitions>
```

```

<RowDefinition />
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Button Grid.Column="0" Margin="5" Content="开始" Click="Button_Start"/>
    <Button Grid.Column="1" Margin="5" Content="暂停" Click="Button_Pause"/>
    <Button Grid.Column="2" Margin="5" Content="继续" Click="Button_Resume"/>
    <Button Grid.Column="3" Margin="5" Content="停止" Click="Button_Stop"/>
</Grid>
</Grid>
</Window>

```

代码 16-28 建立窗口在 Grid 面板中放置降龙十八掌的每一掌的插图

首先在窗口的 Loaded 事件中将 18 个 Image 控件顺序叠放在 Grid 面板中，然后为每个控件设置一个 DoubleAnimation 的动画来控制其透明度变化，如代码 16-29 所示。

```

Storyboard story;
private List<Image> imagelist;
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    story = new Storyboard();
    NameScope.SetNameScope(this, new NameScope());
    // 在 Grid 面板中放置 Image 控件，顺序应该由 18 到 1，这样才能保证第一掌在最上方
    ①
    for (int i = 0; i < 18; i++)
    {
        string name = "img" + (18 - i);
        string strUri;
        strUri = string.Format("pack://application:,,,/img/{0}.jpg",
        (18 - i));
        Image img = new Image();
        img.Name = name;
        BitmapImage myBitmapImage = new BitmapImage();
        myBitmapImage.BeginInit();
        myBitmapImage.UriSource = new Uri(strUri);
        myBitmapImage.EndInit();
        img.Source = myBitmapImage;
        this.RegisterName(name, img);
        grid.Children.Add(img);
        Grid.SetRow(img, 0);
        Grid.SetColumn(img, 0);
        imagelist.Add(img);
    }
    ②
    imagelist.Reverse();
    for (int i = 0; i < imagelist.Count - 1; i++)
    {
        ③
        DoubleAnimation anim = new DoubleAnimation();
        anim.BeginTime = TimeSpan.FromSeconds(0.5 * i);
        anim.From = 1;
        anim.To = 0;
        anim.Duration = TimeSpan.FromSeconds(0.5);
        Storyboard.SetTargetName(anim, imagelist[i].Name);
        Storyboard.SetTargetProperty(anim, new

```

```
PropertyPath(Image.OpacityProperty));
    story.Children.Add(anim);
}
```

代码 16-29 将 18 个 Image 控件按照顺序地叠放在 Grid 面板中并控制其透明度

注意 Image 控件在代码中创建，未通过 XAML 实现。如果需要对 Image 的属性实现动画，需要首先调用 NameScope 的一个静态方法 SetNameScope（代码①），然后注册该控件的名字（代码②），这样 Storyboard 在开始动画时才能找到该控件。此外添加控件时的顺序是从 18 到 1，设置动画时则恰恰相反，因此在开始为 Image 控件设置动画时反转 ImageList（代码③）。每张图片动画的持续时间为 0.5 秒，而后一张图片动画开始的时间总比前一张图片开始的时间晚 0.5 秒，从而保证图片动画的连续性。

设置后添加动画的开始、暂停、继续和停止控制代码，注意 Storyboard 的 Begin 函数需要将第 2 个参数设置为 true 表示该动画可控，如代码 16-30 所示。

```
private void Button_Start(object sender, RoutedEventArgs e)
{
    story.Begin(this,true);
}

private void Button_Pause(object sender, RoutedEventArgs e)
{
    story.Pause(this);
}

private void Button_Resume(object sender, RoutedEventArgs e)
{
    story.Resume(this);
}

private void Button_Stop(object sender, RoutedEventArgs e)
{
    story.Stop(this);
}
```

代码 16-30 添加动画的开始、暂停、继续和停止控制代码

一个简单的降龙十八掌掌法浏览器就完成了。从此降龙十八掌秘笈之后只有十八掌浏览器。

16.7 接下来做什么

本章是较为基本的动画知识，实际上动画大量用在了游戏中。在利用动画进行 WPF 或者 Silverlight 开发中，国内也有很优秀的网上教材，笔者慎重推荐深蓝色右手的博客——“C#开发 WPF-Silverlight 动画及游戏系列教程”。WPF 并不适合开发游戏，但是对于深入理解动画非常有益。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版 射雕英雄传》，“第十二回 亢龙有悔”。
- [2] MSDN Library for Visual Studio 2008 SP1 Animation and Timing.

WPF3D 图形

坐了二十余日大车，越是向北，越加寒冷。道上冰封雪积，大车已不能通行。两人改乘马匹，到后来连马也不能走了，便在密林雪原中徒步而行。好在韦小宝寻宝为名，避难是实。眼见穷山恶水，四野无人，心中越觉平安。双儿记性甚好，依循地图上所绘方位，慢慢向北寻去。遇到猎人参客，便打听地名，与图上所载印证。地图上有八个四色小圈，便是鹿鼎山的所在……

——《鹿鼎记》，第三十五回 “曾随东西南北路 独结冰霜雨雪缘”^[1]

这是韦小宝和双儿按照四十二章经中碎羊皮拼出的地图寻找宝藏的一幕。地图虽好，但终究是平面的。倘若是一幅立体地图，想必小宝和双儿不必找得如此辛苦。本章介绍 WPF 的 3D 图形。在最后一节中我们还会实现一个简单的放大、缩小和漫游的 3D 地球。

本章内容如下。

- (1) WPF3D 引言。
- (2) WPF3D 数学基础。
- (3) 从 3D 物体到 2D 图形。
- (4) 基本几何体。
- (5) 光源和材质。
- (6) 动画和交互。
- (7) 接下来做什么。

17.1 WPF3D 引言

17.1.1 WPF3D 图形的作用

WPF3D 的设计并不是为了游戏。微软承认如果需要画质更加精细，运行速度更快，那么应该使用

DirectX¹.

WPF3D 存在的重要的目的在于为开发人员提供一种能力，以便将 3D 功能集成到客户端应用程序中。使得 3D 和其他用户界面元素或者多媒体实现无缝集成，具体表现在如下方面。

(1) 为控件产生全新的 3D 外观, 如图 17-1 所示, 应用程序的下侧均是立方体式外观的按钮。

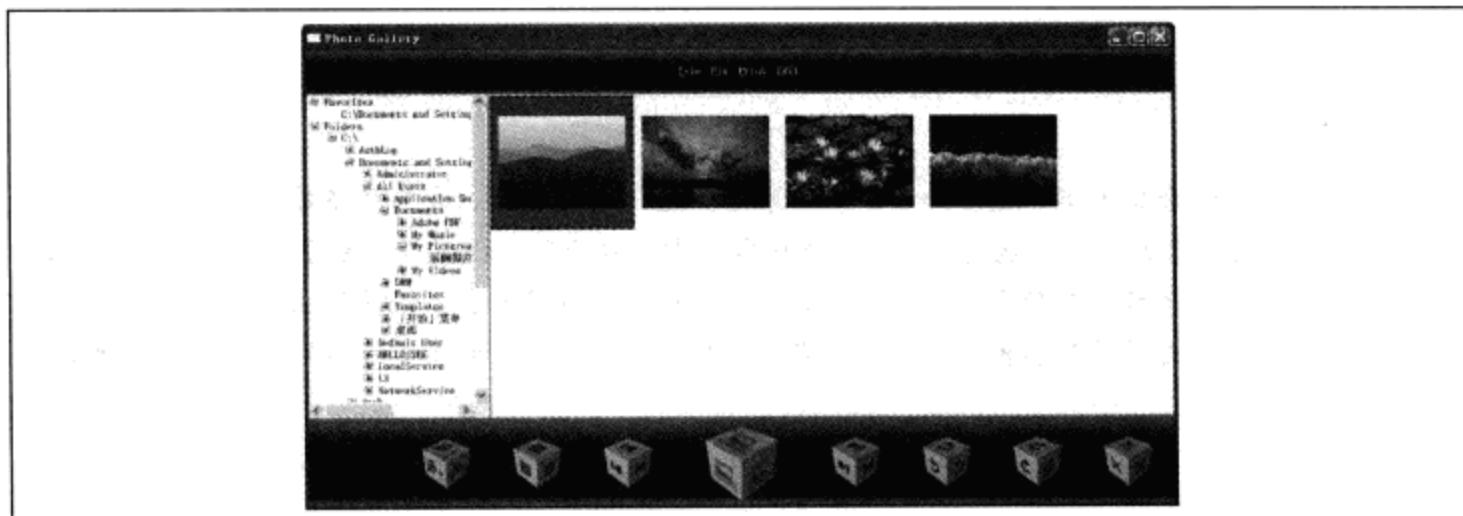


图 17-1 全新的立方体式外观的按钮

(2) WPF3D 内置动画功能，前面介绍的动画 API 均可应用在 WPF3D 中，以便实现复杂的 3D 图形功能。

(3) WPF3D 和数据绑定结合也非常紧密，开发人员可以将 3D 图形对象与一些控件（如滚动条）绑定在一起实现 3D 图形的移动或变换。

(4) 3D 控件和其他界面元素同样也可以放在模板、样式或者面板中。

(5) 2D 图形及视频等能够在 3D 模型的表面显示, 如图 17-2 所示为把各种控件放置在一个立方体的表面上。

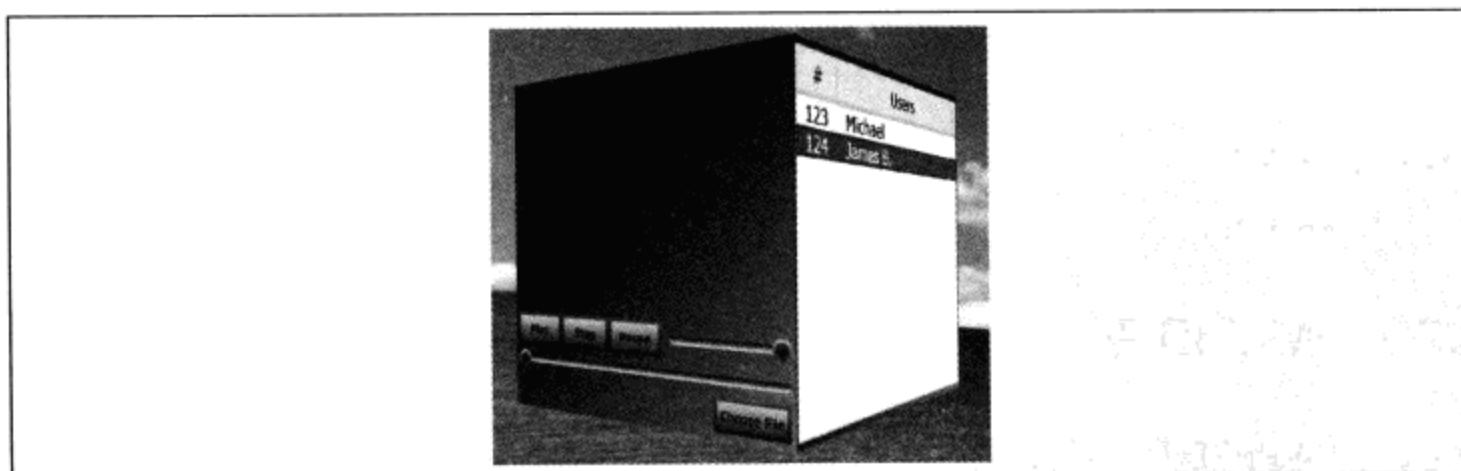


图 17-2 各种控件放置在一个立方体的表面^[2]

1 当然 OpenGL 是一个不错的选择，但是微软毕竟不会把用户推向自己的对手。

17.1.2 用 2D 图形产生立体感

2D 图形也能产生立体感，图 17-3 所示为一个 2D 圆形通过添加明暗效果和阴影产生立体感，变成一个球。

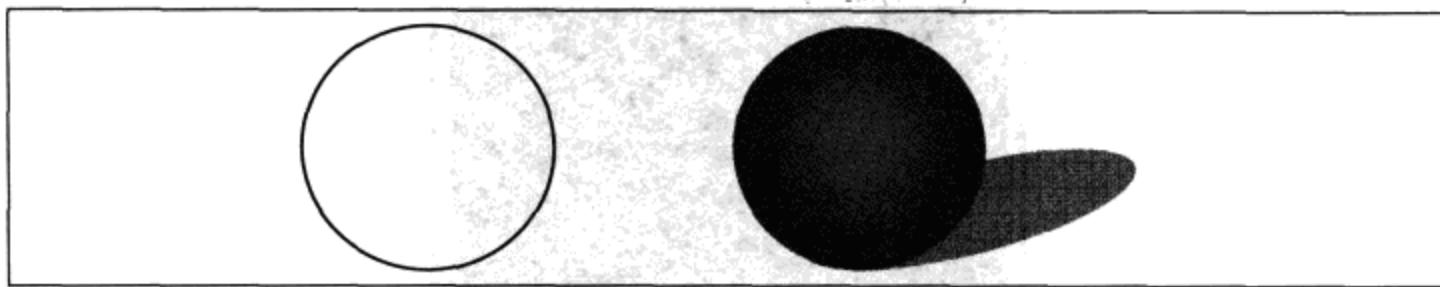


图 17-3 添加明暗效果和阴影产生立体感

图 17-4 所示为一个正方形，通过为其添加几条边形成一个立方体。

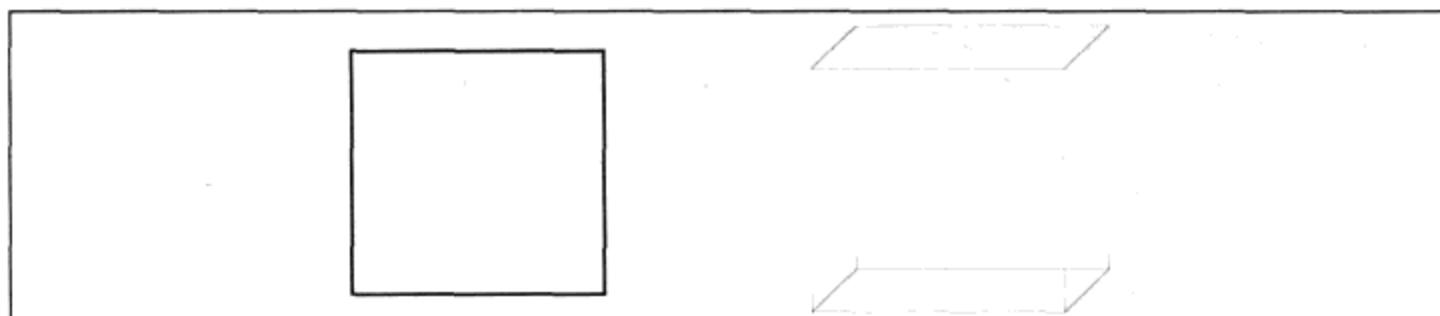


图 17-4 二维正方形变立方体

代码 17-1 用 2D Drawing 尝试绘制一个有立体感的房子（详见 `mumu_3DDemo` 工程）。

```
<Page Background="Black"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Image>
        <Image.Source>
            <DrawingImage>
                <DrawingImage.Drawing>
                    <DrawingGroup x:Name="House">
                        <GeometryDrawing x:Name="Front" Brush="Red"
                            Geometry="M0,260 L0,600 L110,670 L110,500 L190,550 L190,710 L300,775
                            L300,430 L150,175" />
                        <GeometryDrawing x:Name="Side" Brush="Green"
                            Geometry="M300,430 L300,775 L600,600 L600,260" />
                        <GeometryDrawing x:Name="Roof" Brush="Blue"
                            Geometry="M150,175 L300,430 L600,260 L450,0" />
                    </DrawingGroup>
                </DrawingImage.Drawing>
            </DrawingImage>
        </Image.Source>
    </Image>
</Page>
```

代码 17-1 绘制一个有立体感的房子（`House using 2D Drawings.xaml`）

运行结果如图 17-5 所示。

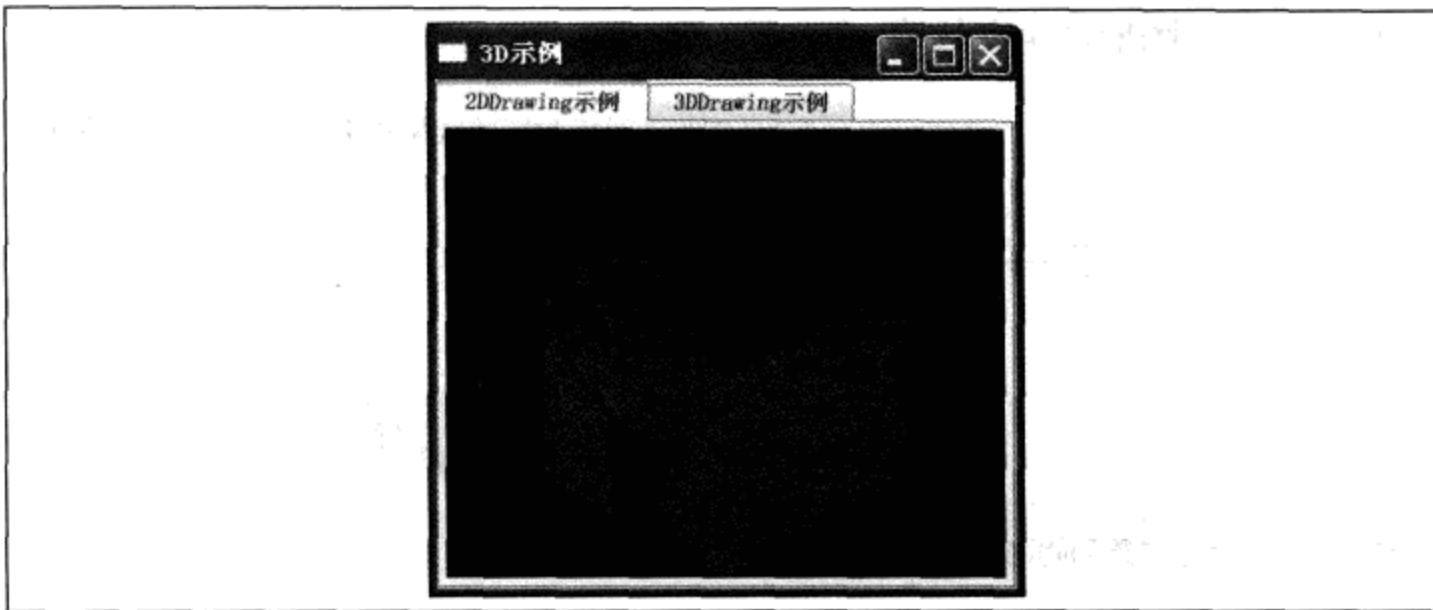


图 17-5 运行结果

虽然房子像 3D 的，但是产生图形的数据仍然是 2D 的，只不过是绘制了一些平面的多边形。我们无法看到房子的背面及内部，代码 17-2 使用 Model3D 来绘制相同的图形。

```
<Page Background="Black"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Viewport3D>
        <Viewport3D.Camera>
            <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
        </Viewport3D.Camera>
        <Viewport3D.Children>
            <!--第 1 个 ModelVisual3D 为光源-->
            <ModelVisual3D>
                <ModelVisual3D.Content>
                    <AmbientLight/>
                </ModelVisual3D.Content>
            </ModelVisual3D>
            <!--第 2 个 ModelVisual3D 为房屋模型的几何体-->
            <ModelVisual3D>
                <ModelVisual3D.Transform>
                    <RotateTransform3D>
                        <RotateTransform3D.Rotation>
                            <AxisAngleRotation3D x:Name="AboutY" Axis="0,1,0" />
                        </RotateTransform3D.Rotation>
                    </RotateTransform3D>
                </ModelVisual3D.Transform>
                <ModelVisual3D.Content>
                    <Model3DGroup x:Name="House">
                        <GeometryModel3D x:Name="Roof">
                            <GeometryModel3D.Material>
                                <DiffuseMaterial Brush="Blue" />
                            </GeometryModel3D.Material>
                            <GeometryModel3D.Geometry>
                                <MeshGeometry3D Positions="-1 1 1, 0 2 1, 0 2 -1, -1 1 -1, 0 2 1,
1 1 1, 1 1 -1, 0 2 -1" TriangleIndices="0 1 2, 0 2 3, 4 5 6, 4 6 7"/>
                            </GeometryModel3D.Geometry>
                        </GeometryModel3D>
                        <GeometryModel3D x:Name="Sides">
                            <GeometryModel3D.Material>
```

```

        <DiffuseMaterial Brush="Green" />
    </GeometryModel3D.Material>
    <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="-1,1,1 -1,1,-1 -1,-1,-1,-1,-1,1 1,1,-1
1,1,1 1,-1,1 1,-1,-1"
            TriangleIndices="0 1 2, 0 2 3, 4 5 6, 4 6 7"/>
    </GeometryModel3D.Geometry>
</GeometryModel3D>
<GeometryModel3D x:Name="FrontAndBack">
    <GeometryModel3D.Material>
        <DiffuseMaterial Brush="Red" />
    </GeometryModel3D.Material>
    <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="-0.25 -0 1, -1 1 1, -1 -1 1, -0.25 -1
1, -0.25 -0 1, -1 -1 1, 0.25 -0 1, 1 -1 1, 1 1 1, 0.25 -0 1, 0.25 -1 1, 1 -1 1, 1
1, 0 2 1, -1 1 1, -1 1 1, -0.25 -0 1, 0.25 -0 1, 1 1 1, 1 1 -1, 1 -1 -1, -1 -1 -1,
-1 1 -1, 1 1 -1, -1 1 -1, 0 2 -1"
            TriangleIndices="0 1 2, 3 4 5, 6 7 8, 9 10 11, 12 13 14, 15 16
17, 15 17 18, 19 20 21, 19 21 22, 23 24 25"/>
    </GeometryModel3D.Geometry>
</GeometryModel3D>
</Model3DGroup>
</ModelVisual3D.Content>
</ModelVisual3D>
</Viewport3D.Children>
</Viewport3D>
</Page>

```

代码 17-2 用 Model3D 来绘制立体房子

按 F5 键查看程序运行结果，几乎和前面的图形相同，但是这是一个真正的 3D 模型。为了说明这一点，我们添加一个动画使得房子可以旋转，如代码 17-3 所示。

```

.....
<Viewport3D.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
        <BeginStoryboard>
            <Storyboard Storyboard.TargetName="AboutY" Storyboard.TargetProperty="Angle">
                <DoubleAnimation From="0" To="360" Duration="0:0:2"
RepeatBehavior="Forever"/>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Viewport3D.Triggers>

```

代码 17-3 添加动画以不同视角查看房子

运行结果如图 17-6 所示。

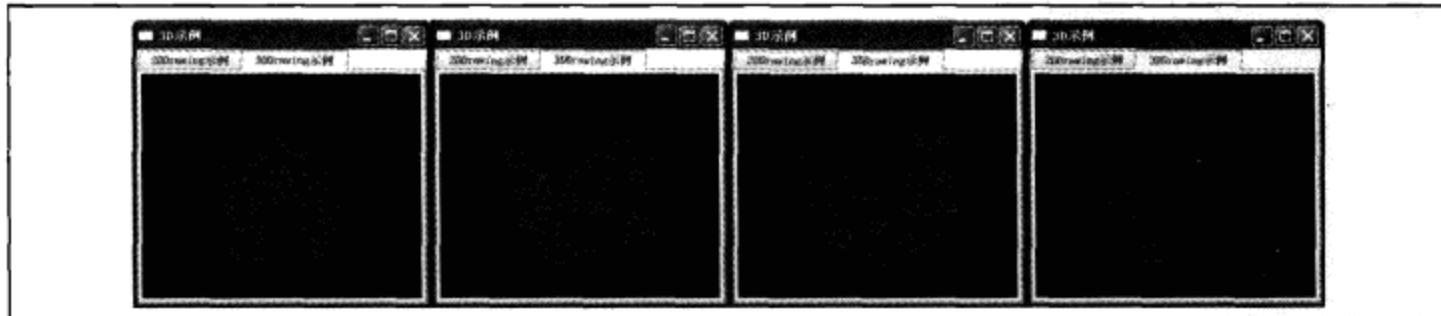


图 17-6 运行结果

17.1.3 WPF3D 类概览

前面绘制房子的 3D 代码，如代码 17-4 所示。

```
<Page Background="Black"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Viewport3D>
    ....
```

代码 17-4 Page

根部的元素是我们熟悉的 Page，其子元素是一个新的类型。名为“Viewport3D”，可以称得上是 2D 和 3D 之间的桥梁。一方面所有的 3D 场景都会绘制在该类型中；另外 Viewport3D 派生自 FrameworkElement，和其他 2D 元素没有任何差别，本身也是 WPF 可视化树上的一部分，能够接受鼠标、键盘和 stylus 的输入。

接着探索代码 17-5 所示。

```
.....
<Viewport3D>
    <Viewport3D.Camera>
        <OrthographicCamera Position="5,5,5" LookDirection="-1,-1,-1" Width="5"/>
    </Viewport3D.Camera>
    <Viewport3D.Children>
    ....
```

代码 17-5 Viewport3D 定义了两个关键属性

为了绘制 3D 场景，Viewport3D 定义了两个关键属性。其中 Camera 定义了观察 3D 场景的位置和方向，以及如何将 3D 物体绘制在 2D 平面上。它有两种重要投影方式，即透视投影和正射投影。

另一个 Children 是 Viewport3D 的内容属性，所有的 3D 图形（比如绘制的房子）均在其中。Children 实际上是一个 Visual3D 的集合，Visual3D 类似 2D 图形中的 Visual，它是一个独立的可视个体。Visual3D 也是一个抽象类，派生它的只有一个类 ModelVisual3D。在这个例子中有两个 ModelVisual3D，如代码 17-6 所示。

```
.....
<Viewport3D.Children>
    <ModelVisual3D>
        <!--第 1 个 ModelVisual3D 为光源-->
        <ModelVisual3D.Content>
            <AmbientLight/>
        </ModelVisual3D.Content>
    </ModelVisual3D>
        <!--第 2 个 ModelVisual3D 为房屋模型的几何体-->
    <ModelVisual3D>
        <ModelVisual3D.Transform>
            <RotateTransform3D>
            .....
            </RotateTransform3D>
        </ModelVisual3D.Transform>
        <ModelVisual3D.Content>
            <Model3DGroup x:Name="House">
```

```

<GeometryModel3D x:Name="Roof">
    .....
</GeometryModel3D>
<GeometryModel3D x:Name="Sides">
    .....
</GeometryModel3D>
<GeometryModel3D x:Name="FrontAndBack">
    .....
</GeometryModel3D>
</Model3DGroup>
</ModelVisual3D.Content>
</ModelVisual3D>

```

代码 17-6 ModelVisual3D

一个 ModelVisual3D 用来描述光源，另一个描述房屋模型。它有 Children 和 Content 属性，前者是一个 Visual3D 的集合，因此 ModelVisual3D 是一个可以自我嵌套的对象；后者是一个 Model3D 类型。Model3D 本身也是一个抽象类，它的 3 个派生类 GeometryModel3D、Model3DGroup 和 Light 分别代表一个几何模型、光源和一个 Model3D 的集合。

在 WPF 中用 Light 代表现实世界中的光源，如太阳或者电灯等。不过 Light 也是一个抽象类，该例中的 AmbientLight 派生自 Light。AmbientLight 称为“环境光”，它模拟一个多云天气的户外光照情况。所有的物体被照射的强度相同，并且没有任何阴影。Light 还派生了多种不同的光源，在后面陆续介绍。

第 2 个 ModelVisual3D 用来描述房屋，其 Content 属性被指定为一个 Model3DGroup。它由 3 个 GeometryModel3D 组成，分别是房屋的房顶、两个侧面和前后面，模型中也分别用蓝色、绿色和红色区分。以房屋的房顶为例探究 GeometryModel3D 的组成，如代码 17-7 所示。

```

<GeometryModel3D x:Name="Roof">
    <GeometryModel3D.Material>
        <DiffuseMaterial Brush="Blue" />
    </GeometryModel3D.Material>
    <GeometryModel3D.Geometry>
        <MeshGeometry3D Positions="-1 1 1, 0 2 1, 0 2 -1, -1 1 -1, 0 2 1,
1 1 1, 1 1 -1, 0 2 -1"
                           TriangleIndices="0 1 2, 0 2 3, 4 5 6, 4 6 7"/>
    </GeometryModel3D.Geometry>

```

代码 17-7 GeometryModel3D 的组成

GeometryModel3D 类有 3 个属性，其中 Geometry 用来描述几何构造；Material 用来描述物体正面的材质；BackMaterial 用来描述物体反面的材质。材质描述物体表面对光反射的性质。后两个属性的类型都为 Material，它也是一个抽象类。DiffuseMaterial 派生自它，主要用途是模拟漫反射效果，即将投射到表面的光线沿各个方向均匀反射。该表面的明暗度与摄像机（或观察者）的位置有关。在现实生活中，棉花或羊毛的编织物，塑料、金属或其他光滑的平坦表面都不是漫反射材质。

Geometry 属性的类型为 Geometry3D，Geometry3D 类似于 2D 图形中的 Geometry。不同的是 2D 的几何图形往往是用直线，矩形和椭圆等几何图形来表示，而传统的 3D 图形表面往往是用一系列的网格（mesh）来表达。在 WPF3D 中这种网格描述采用了最简单的多边形，即三角形。一般对于具有平坦表面的图形，只需要少量的三角形。但是复杂表面的图形则需要用到很多三角形，如图 17-7 所示。

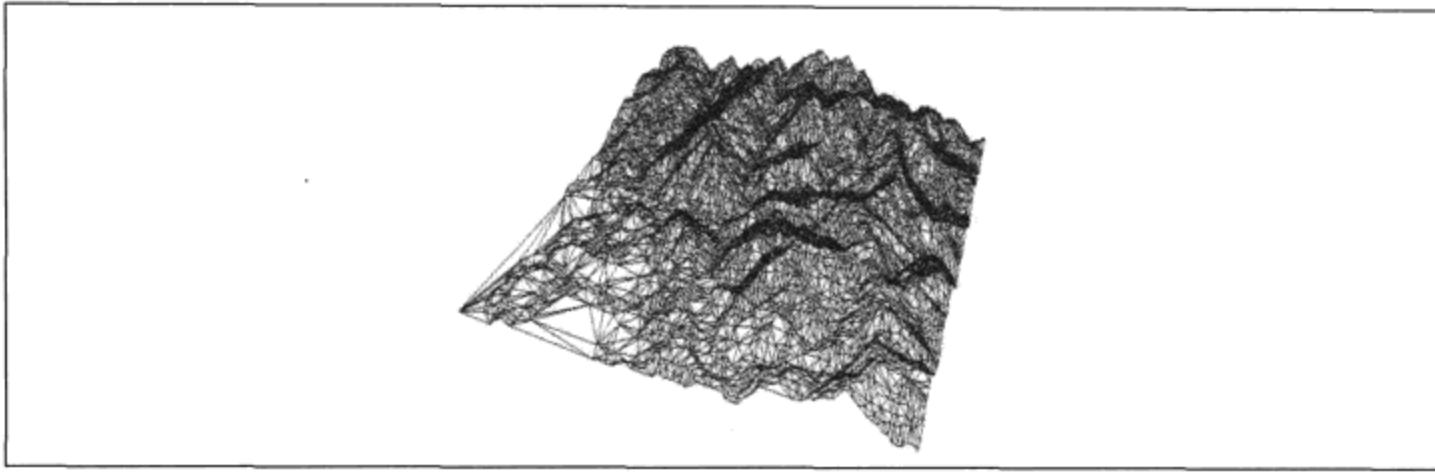


图 17-7 若干三角形模拟的地形表面

该图为一个由若干三角形模拟的地形表面，地形起伏比较平缓的地区三角形比较少；地形起伏剧烈的地区则比较多。

派生于 Geometry3D 的只有一个类，即 MeshGeometry3D，它有两个重要属性。其中 Positions 属性是一个坐标点串，用来描述三角形的坐标位置。每个坐标点由 (x,y,z) 3 个值来表示，坐标值可以用空格或者逗号分隔。推荐一个点的坐标值用空格隔开，点之间用逗号隔开，这样便于阅读。在 Positions 集合中每个 Point3D 对象都具备了一个从 0 开始的索引值，如上例的 Positions 中点的索引为 0~7。

另外一个重要属性是 TriangleIndices，它是一个整数的集合。这个集合根据点的索引定义了三角形，房屋屋顶的几何形状按照如代码 17-8 描述。

```
<MeshGeometry3D Positions="-1 1 1, 0 2 1, 0 2 -1, -1 1 -1, 0 2 1, 1 1  
1, 1 1 -1, 0 2 -1"  
TriangleIndices="0 1 2, 0 2 3, 4 5 6, 4 6 7"/>
```

代码 17-8 描述房屋屋顶的几何形状

表 17-1 所示为点的索引值和坐标值，表 17-2 所示为三角形和点的索引值的关系表。

表 17-1 点的索引值和坐标值

点的索引值	点的坐标值
0	-1 1 1
1	0 2 1
2	0 2 -1
3	-1 1 -1
4	0 2 1
5	1 1 1
6	1 1 -1
7	0 2 -1

表 17-2 三角形和点的索引值的关系

三角形	点的索引值
A	0 1 2
B	0 2 3
C	4 5 6
D	4 6 7

图 17-8 所示为按照上面两个表的数据构造的屋顶的 3D 表面图。

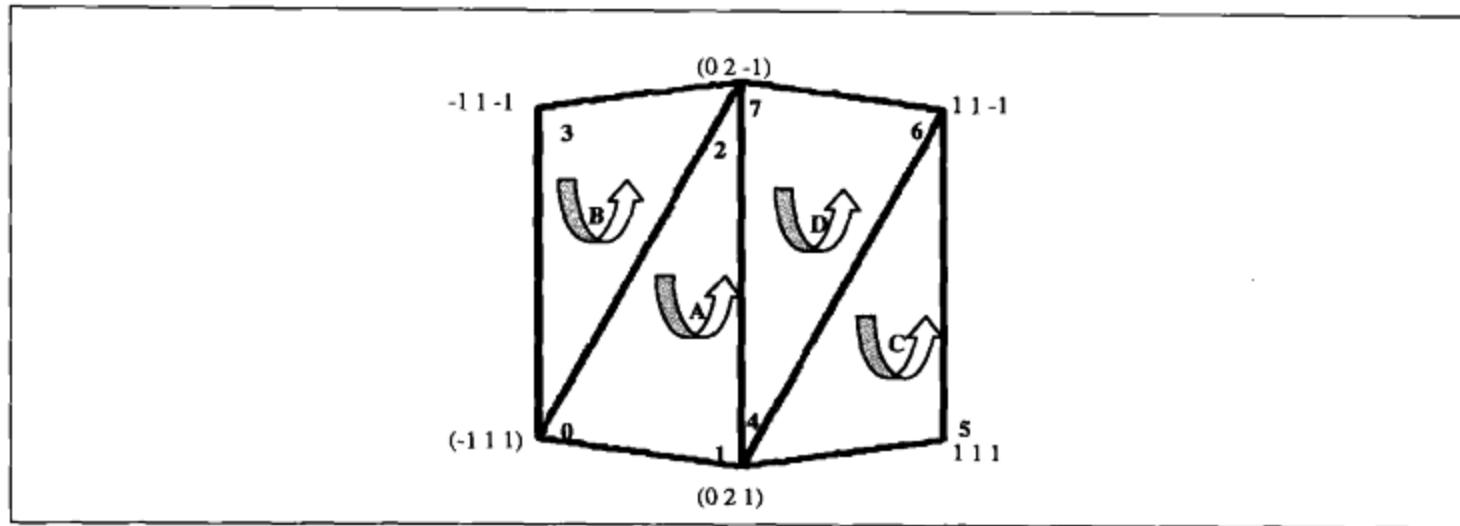


图 17-8 屋顶的 3D 表面

可以从图中看出 WPF 如何根据这两个属性构造 3D 表面，在 TriangleIndices 集合中点的索引值出现的次序不同，则图形的显示方式也不同。虽然每个三角形在同一个平面上，但是在三维空间中有前面和背面，按逆时针方向索引的三角形顶点的面是前面。由于从正上方观察房屋的屋顶，因此面对我们的面都是正面，可以看到 4 个三角形均按逆时针方向索引。

图 17-9 所示为 WPF3D 中重要类之间的关系，其中的 ModelUIElement3D 在上例中没有涉及，该类是在 .NetFramework3.5 SP1 中引入。比起 ModelVisual3D，它更方便接受用户的输入、焦点和事件。

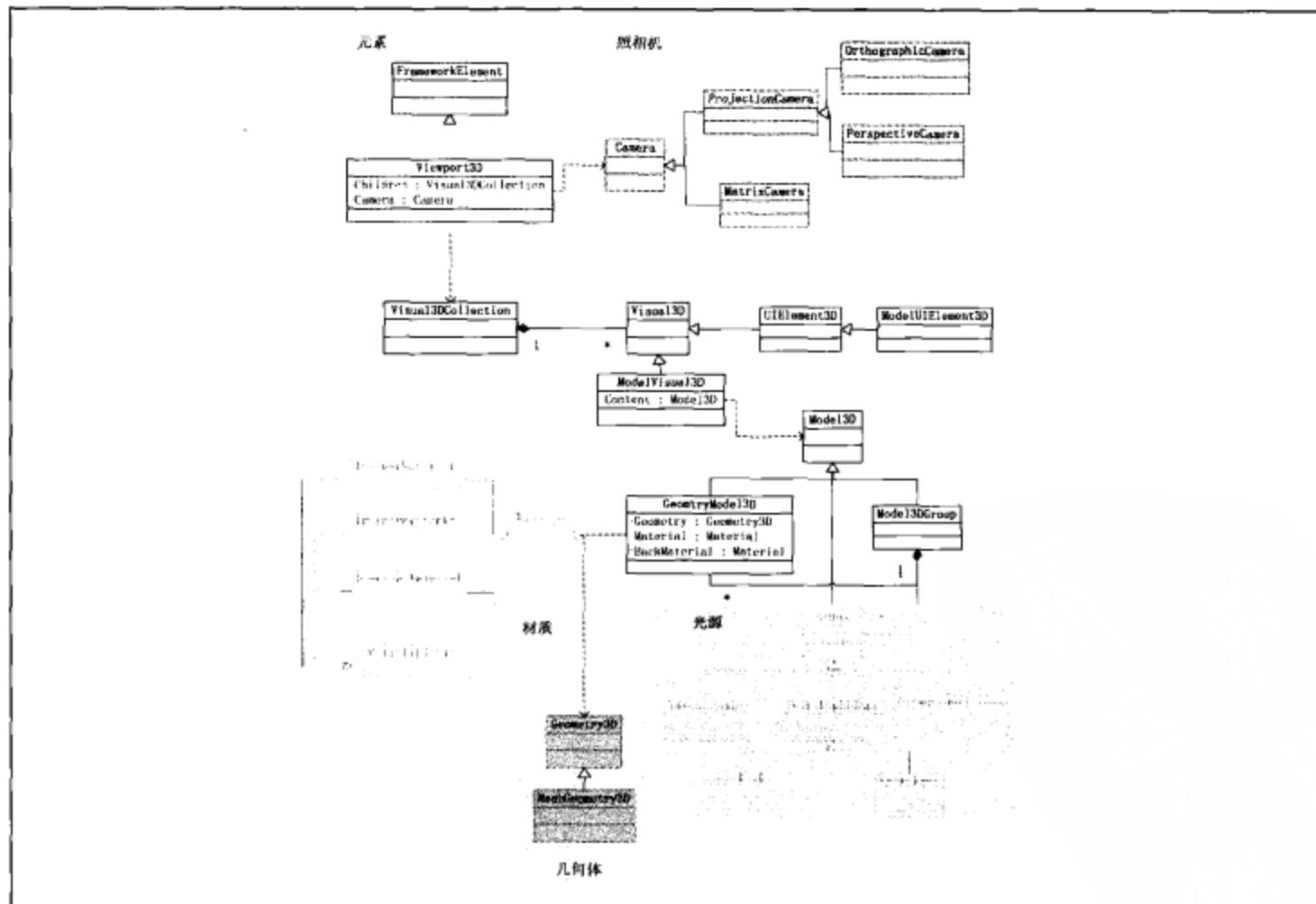


图 17-9 WPF3D 中重要类之间的关系

实际上 WPF3D 和 WPF2D 非常类似，表 17-3 所示为 3D 类型和与之相似的 2D 类型。

表 17-3 3D 类型和与之相似的 2D 类型

2D 类型	3D 类型	描述
Visual	Visual3D	Visual 是渲染 2D 元素的基类，Visual3D 是渲染 3D 元素的基类
Drawing	Model3D	Drawing 用来描述 Visual 中的 2D 图形内容，Model3D 用来描述 Visual3D 中的 3D 模型
Geometry	Geometry3D	Geometry 表示一个 2D 图形的几何类，而 Geometry3D 表示一个 3D 图形的几何类
Transform	Transform3D	Transform 为 2D 变换的基类，Transform3D 为 3D 变换的基类

17.2 WPF3D 数学基础

与学习 WPF2D 图形一样，首先需要掌握一些必备的数学知识。如果具有 2D 图形的数学基础，则会加快学习 3D 图形数学的进度。

17.2.1 坐标系

在 WPF 的 2D 坐标系中原点位置在绘制区域的左上角，X 轴向右，Y 轴向下；3D 坐标系通常会认为原点是在空间的中心，Y 轴向上，Z 轴的正方向是面向观察者的方向，如图 17-10 所示。

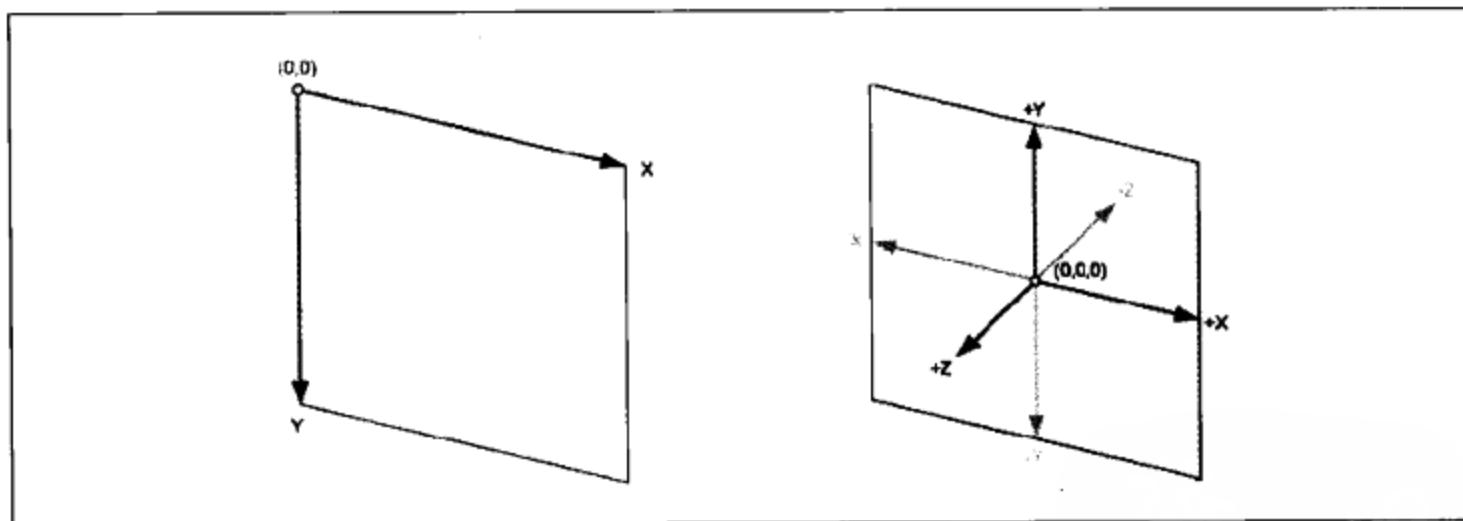


图 17-10 2D 与 3D 坐标系

这也是通常所说的右手坐标系，即右手的食指指向 X 值增加的方向；中指指向 Y 值增加的方向；大拇指所指方向为 Z 值增加的方向，如图 17-11 所示。

在 2D 坐标系中，两条轴线将平面分为 4 个象限；在 3D 坐标系中 3 个平面将空间分为 8 个象限。通常将这 3 个平面分别称为“YZ 平面”、“XZ 平面”，以及“XY 平面”。这 8 个象限也可以用“左下前”和“右下前”这样的短语来表示。如图 17-12 所示。

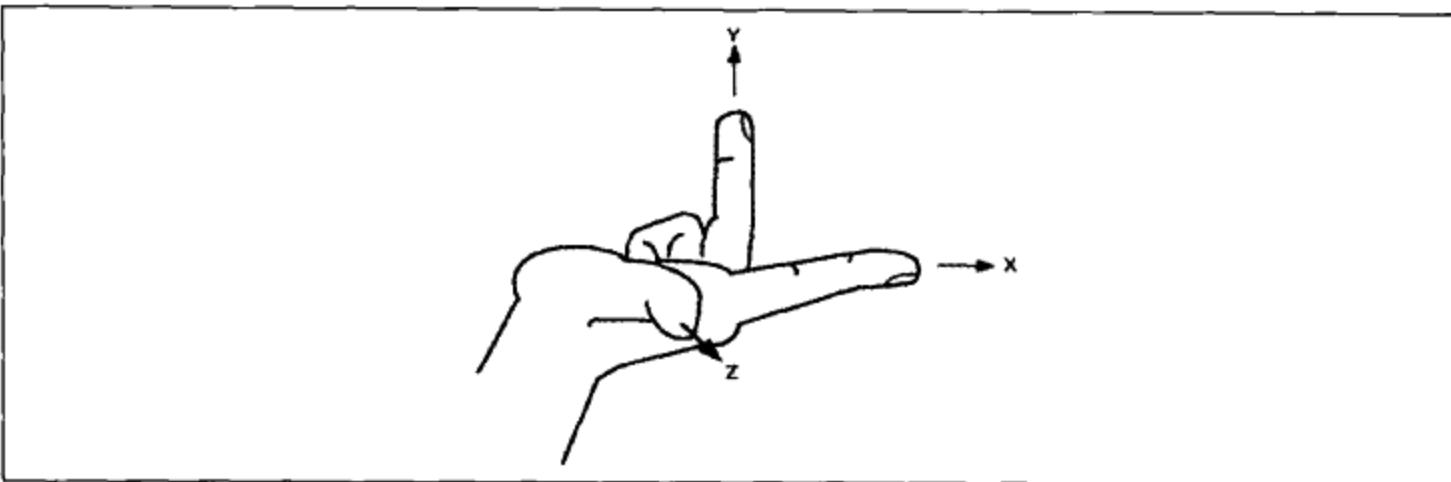


图 17-11 右手坐标系

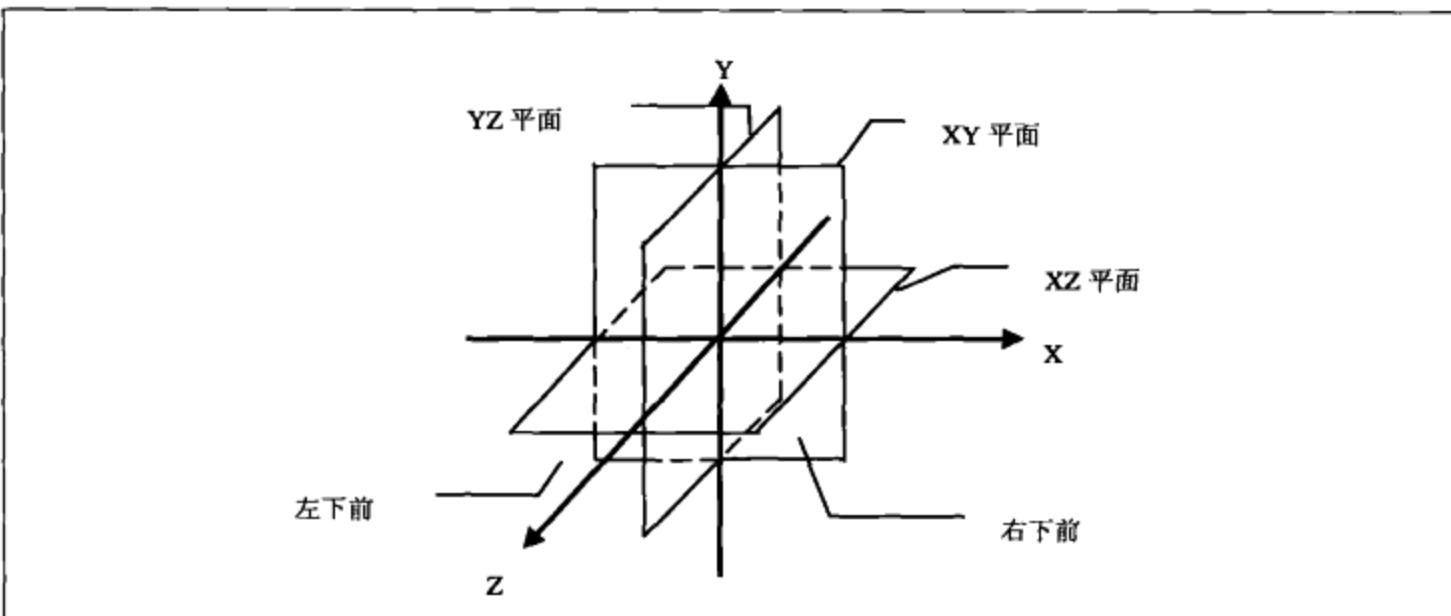


图 17-12 3D 坐标系的 8 个象限

在 2D 坐标系中，我们曾经讨论过分辨率无关的单位 DIU，而在 3D 坐标系中使用相对单位。通常会习惯性地将一个 3D 场景的长度单位设置的小一点，如 X、Y 和 Z 值的范围在 -1~1 之间。

17.2.2 空间点

在 3D 空间中某个精确的位置可以用一个坐标点来表示，习惯记为 (x,y,z) 。与二维空间点一样，3D 空间点也可以用齐次坐标表示，记为 $(x,y,z,1)$ 。WPF3D 定义了一个名为“Point3D”的结构体来存放这些坐标点。Point3D 只有 X、Y 和 Z 属性，实际上隐藏一个 4D 点。

Point3D 属于 System.Windows.Media.Media3D 命名空间，在使用该结构体时要注意引用该命名空间，代码 17-9 创建一个点的坐标。

```
Point3D point = new Point3D(2, 1, 1);
```

代码 17-9 创建一个点的坐标

在 XAML 文件中 Point3D 对象用文本字符串表示，并用空格或单个逗号隔开文本字符串中的数字，

如“2 1 1”或者“2,1,1”。在 XAML 中一个 3D 点的集合也可以用字符串来表示，为了表示清楚，建议用逗号分隔每个点。而每个点的坐标则用空格分隔，如“2 1 1, 1 1 1”。

17.2.3 向量

在 2D 坐标系中，WPF 用 Vector 结构体表示，在 3D 坐标系中用 Vector3D 结构体表示，也记为 (x,y,z) 。3D 空间向量的齐次坐标和 2D 空间一样，坐标的最后一个值记为 0，即 $(x,y,z,0)$ 。和 Point3D 一样，Vector3D 也属于 System.Windows.Media.Media3D 命名空间。

1. 3D 向量基础

和 2D 向量一样，3D 向量在 3D 空间中包含幅度和方向的信息。幅度用长度表示，有时称为“模”。向量 A 从原点 $(0,0,0)$ 指向 (x_a, y_a, z_a) ， A 的模用记号 $|A|$ 来表示，它可用勾股定理计算得出：

$$|A| = \sqrt{x_a^2 + y_a^2 + z_a^2}$$

如果用该向量除以其模，则得到规范化的向量，也称为“单位向量”，其模为 1。Vector3D 结构的 Normalize 方法即将规范化向量的方法。

在 3D 空间中有 3 个向量非常特殊，其方向与坐标轴方向一致。一般称为“基向量”，分别用记号 i ， j 和 k 来表示，定义如下：

$$i = (1, 0, 0)$$

$$j = (0, 1, 0)$$

$$k = (0, 0, 1)$$

一个坐标为 (x_a, y_a, z_a) 的向量 A 也可通过这 3 个基向量的投影表示为：

$$A = x_a i + y_a j + z_a k$$

2. 向量之间的夹角

Vector3D 本身提供了计算向量之间的夹角的方法 Vector3D.AngleBetween，该方法会返回两个 3D 向量之间的夹角，单位为度。代码 17-10 通过 AngleBetween 计算向量 $V_i(1, 0, 0)$ 和 $V_j(0, 1, 0)$ 之间的夹角。

```
Vector3D Vi = new Vector3D(1, 0, 0);
Vector3D Vj = new Vector3D(0, 1, 0);
// 返回的夹角为 90 度
double angle = Vector3D.AngleBetween(Vi, Vj);
```

代码 17-10 计算两个向量间夹角

要熟练掌握 3D 图形编程，应该了解其中基本的数学原理。实际上通过三角函数中的余弦定理来计

算两个向量所成的夹角，如图 17-13 所示。

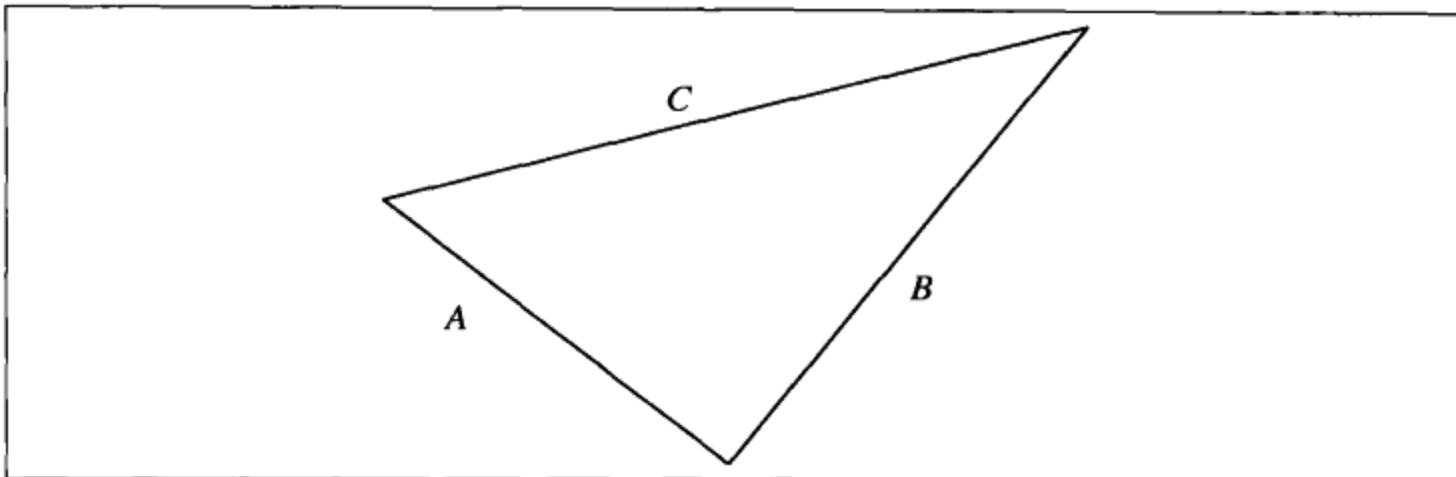


图 17-13 三角函数余弦定理

余弦定理的公式如下：

$$C^2 = A^2 + B^2 - 2AB\cos\alpha$$

现在我们把相同的概念运用在向量上，如图 17-14 所示。

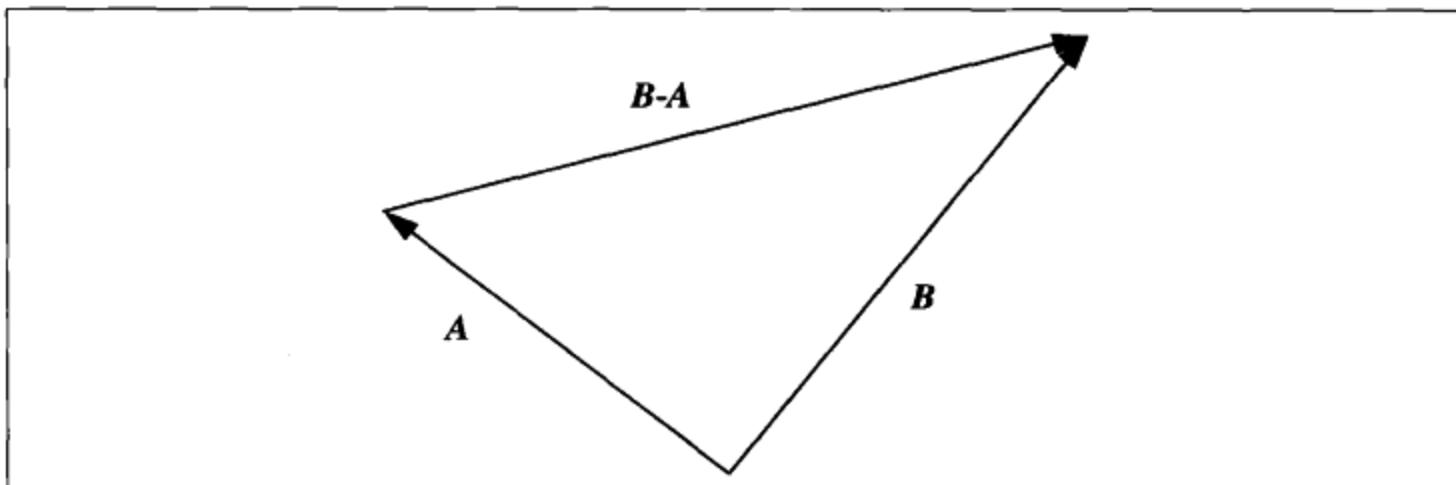


图 17-14 向量运算

有如下计算公式：

$$|B-A|^2 = |A|^2 + |B|^2 - 2|A||B|\cos\alpha$$

假设向量 B 坐标为 (x_b, y_b, z_b) ，向量 A 坐标为 (x_a, y_a, z_a) ，代入上式：

$$(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2 = x_a^2 + y_a^2 + z_a^2 + x_b^2 + y_b^2 + z_b^2 - |A||B|\cos\alpha$$

继续化简，可以得到：

$$\cos\alpha = \frac{x_a x_b + y_a y_b + z_a z_b}{|A||B|}$$

这样，向量 **A** 与 **B** 之间的夹角是上式右端项的反余弦，该夹角会在向量的乘法中大量出现。3D 向量的加减法，以及向量和标量的乘除法与 2D 向量一样。这里我们关注的乘法是 3D 中经常用到的向量的“点乘”（dot product）和“叉乘”（cross product）两种乘法。

3. 向量的点乘

向量的点乘也称为“内积”，用符号 \cdot 来表示。返回一个数值，即一个标量，其定义如下：

$$A \cdot B = |A| |B| \cos \alpha$$

将前面的公式（7）代入到公式（8）中可以得到：

$$A \cdot B = |A| |B| \cos \alpha = x_b x_a + y_b y_a + z_b z_a$$

由上面的公式可以得知，当两向量的分量已知时不必求二者之间夹角的余弦值。而是将对应分量分别相乘，然后求和即可。内积满足加法分配律和交换律，即：

$$A(B+C) = AB+AC$$

$$AB = BA$$

接着我们来查看向量内积的如下 3 种特殊情况。

- (1) 如果两个向量已经规范化，其模为 1，那么两个向量的内积为这两个向量夹角的余弦值。
- (2) 如果两个向量平行，那么余弦值为 1。两个向量的内积是两个向量模的乘积。一个向量与其自身的内积等于向量模的平方，基向量自身的内积为 1；

$$i \cdot j = j \cdot j = k \cdot k = 1$$

- (3) 如果两个向量的夹角为 90 度，这种情况称为“两个向量正交”。相应的余弦值为 0，因此向量的内积为 0。利用这个性质，我们可以判断两个向量是否正交。基向量两两正交，因此有如下公式：

$$i \cdot j = j \cdot k = k \cdot i = 0$$

由于向量内积也满足交换律，因此下面公式也成立：

$$j \cdot i = k \cdot j = i \cdot k = 0$$

Vector3D 提供了 DotProduct 方法来计算两个向量的内积。其实现如代码 17-11 所示。

```
internal static double DotProduct(ref Vector3D vector1, ref Vector3D vector2)
{
    return ((vector1._x * vector2._x) + (vector1._y * vector2._y)) +
(vector1._z * vector2._z));
}
```

代码 17-11 DotProduct 方法来计算两个向量的内积

当两个向量 A 和 B 中的 B 为单位向量时，有非常明显的几何意义。其内积是向量 A 在向量 B 方向上的投影。

$$A \cdot B = |A| |B| \cos\alpha = |A| \cos\alpha$$

如图 17-15 所示。

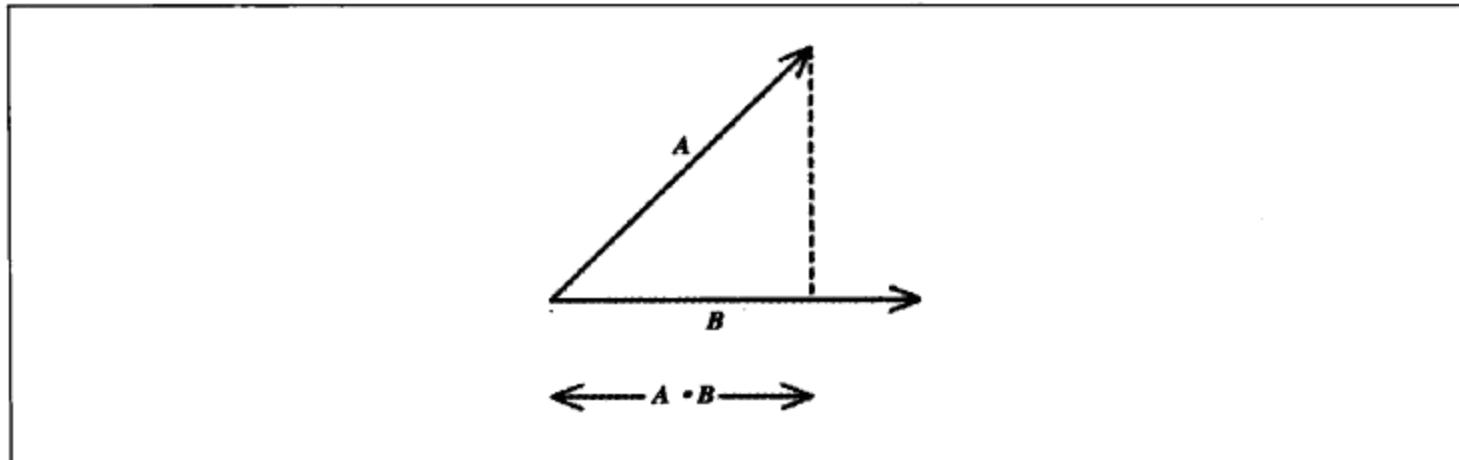


图 17-15 向量的内积

4. 向量的叉乘

叉乘又称为“叉积”，用 \times 符号表示。结果是一个向量，其模的计算公式如下：

$$|A \times B| = |A| |B| \sin\alpha$$

如果 A 和 B 平行，即二者之间的夹角为 0，那么向量的叉积为 0；如果 A 和 B 垂直，向量的叉积即两向量模的乘积。向量叉积的模有非常明显的几何意义，其值为这两个向量构成的平行四边形的面积，如图 17-16 所示。

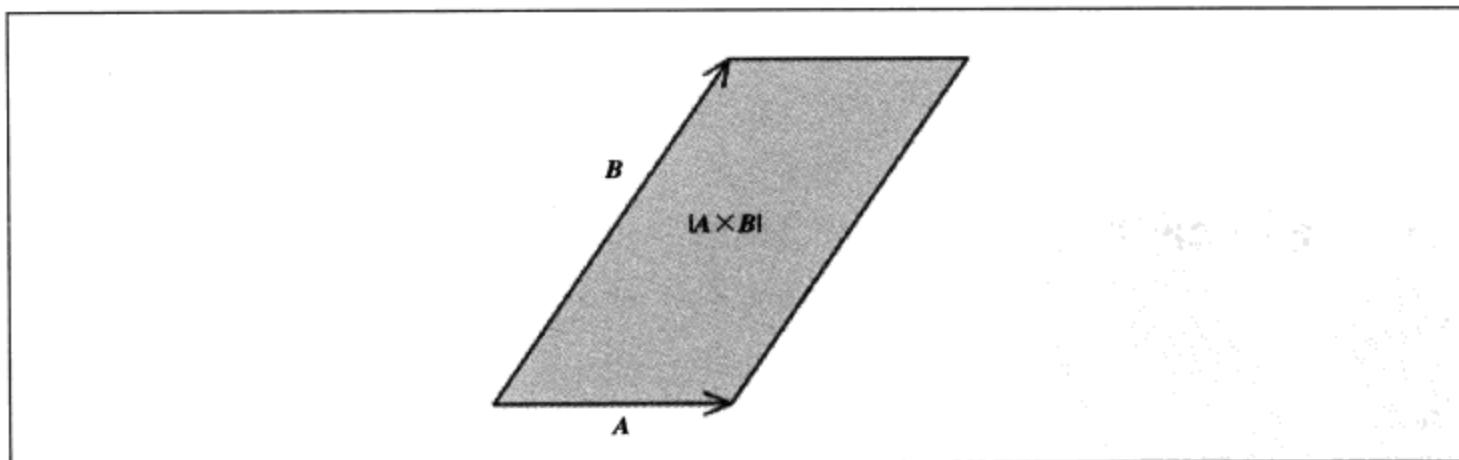


图 17-16 向量叉积的模

叉积的方向与向量 A 和 B 垂直，即叉积与 A 和 B 构成的平面垂直。叉积方向的确定根据右手法则，即用右手的 4 指表示向量 A 的方向，然后手指朝着手心的方向摆动到向量 B 的方向，大拇指所指的方向即叉积的方向。如图 17-17 所示。

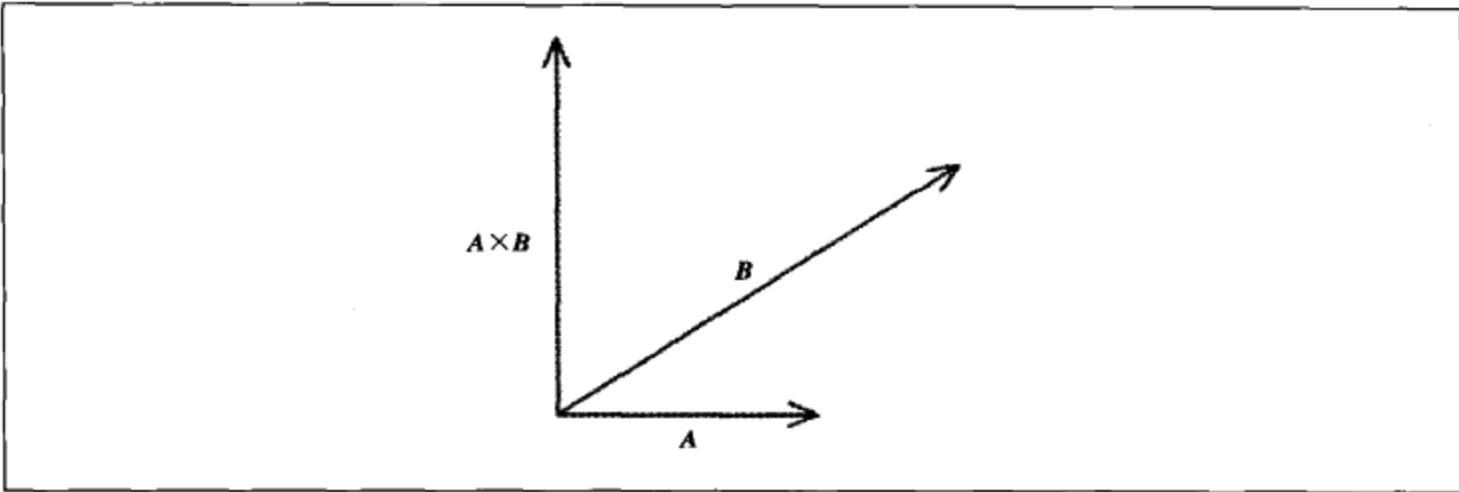


图 17-17 向量叉积的方向

虽然叉积不满足交换律，但是交换运算次序后模不变，且方向相反：

$$A \times B = -B \times A$$

基向量两两正交，且模均为 1，其叉积有如下特殊的性质。

(1) 基向量与其自身的叉积为 0：

$$i \times i = 0$$

$$j \times j = 0$$

$$k \times k = 0$$

(2) 每对基向量的叉积等于第 3 个基向量：

$$i \times j = k$$

$$j \times k = i$$

$$k \times i = j$$

(3) 如果交换运算次序，结果会方向相反：

$$j \times i = -k$$

$$k \times j = -i$$

$$i \times k = -j$$

假定义积满足加法分配律，则可以推导两向量的叉积公式如下：

$$A \times B = (x_a i + y_a j + z_a k) \times (x_b i + y_b j + z_b k)$$

$$=x_ax_bi \times i + x_ay_bi \times j + x_az_bi \times k + y_ax_bj \times i + y_ay_bj \times j + y_az_bj \times k + z_ax_bk \times i + z_ay_bk \times j + z_az_bk \times k$$

通过前面基向量的性质我们可以化简该公式如下：

$$\mathbf{A} \times \mathbf{B} = (y_az_b - z_ay_b)i + (z_ax_b - x_ay_b)j + (x_ay_b - y_ax_b)k$$

Vector3D 提供了一个 CrossProduct 方法来计算两个向量的叉积，其实现如代码 17-12 所示。

```
internal static void CrossProduct(ref Vector3D vector1, ref Vector3D vector2, out Vector3D result)
{
    result._x = (vector1._y * vector2._z) - (vector1._z * vector2._y);
    result._y = (vector1._z * vector2._x) - (vector1._x * vector2._z);
    result._z = (vector1._x * vector2._y) - (vector1._y * vector2._x);
}
```

代码 17-12 CrossProduct 方法计算两个向量的叉积

17.2.4 矩阵和几何变换

在 2D 图形中 WPF 提供了一个矩阵的结构体，名为“Matrix”。它实际上是一个 3×3 的矩阵，由于最后一列默认为 0, 0, 1。因此 Matrix 仅包含 M_{11} 、 M_{12} 、 M_{21} 、 M_{22} 、OffsetX 和 OffsetY 前两列，共计 6 个变量。

缩放、旋转和错切变换只需要一个 2×2 的矩阵即可实现。但是 2D 的平移变换需要借助 3D 的错切变换实现，因此为了统一几何变换，我们引入了齐次坐标，同时变换矩阵变为一个 3×3 的矩阵。由此可以合理地推测，3D 的平移变换需要借助 4D 的错切变换来实现。

建议不要想象一个 4D 空间中如何错切变换，因为这是非常困难的。只需要在数学上成立，则不必想象如何变换。3D 的几何变换需要一个更加高维的矩阵，即 4×4 的矩阵。从 2D 推广而来，该矩阵的最后一行表示在 X, Y 和 Z 方向上的 3 个平移系数，最后一列的默认值为 0, 0, 0, 1。WPF3D 中提供了一个矩阵的结构体 Matrix3D 代表这个 4×4 的矩阵，如下所示：

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{24} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ \text{OffsetX} & \text{OffsetY} & \text{OffsetZ} & M_{44} \end{pmatrix}$$

上面的 3 行和最左边的 3 列构成了 3D 线性变换矩阵，属性 OffsetX, OffsetY 和 OffsetZ 用于平移变换。和 2D 的 Matrix 不同，它保留了第 4 列的属性。即 Matrix3D 允许第 4 列的值不为 0, 0, 0, 1，实现非仿射变换。

1. 从 2D 几何变换矩阵类推 3D 的几何变换矩阵

3D 的几何变换矩阵可以通过二维的几何变换矩阵类推得到。

(1) 齐次坐标下的缩放矩阵

在 2D 几何变换中 X 轴缩放 s_x , Y 轴缩放 s_y , 缩放矩阵如下所示:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

3D 的几何变换中 X 轴缩放 s_x , Y 轴缩放 s_y , Z 轴缩放 s_z , 缩放矩阵如下所示:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(2) 齐次坐标下的平移矩阵

2D 的几何变换中 X 轴方向平移 d_x , Y 轴方向平移 d_y , 平移矩阵如下所示:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{pmatrix}$$

3D 的几何变换中 X 轴方向平移 d_x , Y 轴方向平移 d_y , Z 轴方向平移 d_z , 平移矩阵如下所示:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

(3) 齐次坐标下的旋转矩阵

2D 的几何变换中以原点为中心旋转 θ 角, 旋转矩阵为:

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

3D 的几何变换中旋转总是以某一个轴旋转。以 X 轴为旋转轴旋转 α 角, 旋转矩阵为:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

以 Y 轴为旋转轴旋转 β 角, 旋转矩阵为:

$$\begin{pmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

以 Z 轴为旋转轴旋转 γ 角，旋转矩阵为：

$$\begin{pmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

这 3 个角在数学上又称为“欧拉角”，欧拉定律表明在 3D 空间旋转中任意一个旋转都可以用这 3 个值来表示，即沿任意轴旋转变换都可以通过这 3 个基本的旋转变换组合而来。

(4) 齐次坐标下的复合变换

2D 几何的复合变换和 3D 几何的复合变换均通过矩阵相乘来实现，Matrix3D 和 Matrix 的用法几乎相同。

2. 沿任意轴旋转的几何变换矩阵

旋转沿任意一个轴旋转都可以用 3 个欧拉角来表示，在现实中我们只知道旋转前后的坐标系，并不知道这 3 个欧拉角。后面提到的需要推导从物方坐标系到照相机坐标系的几何变换。

我们假定图形的原坐标系 original 基于 3 个基向量 i 、 j 和 k ，现在新的坐标系 new 的 3 个单位向量为 u 、 v 及 w ，如图 17-18 所示。

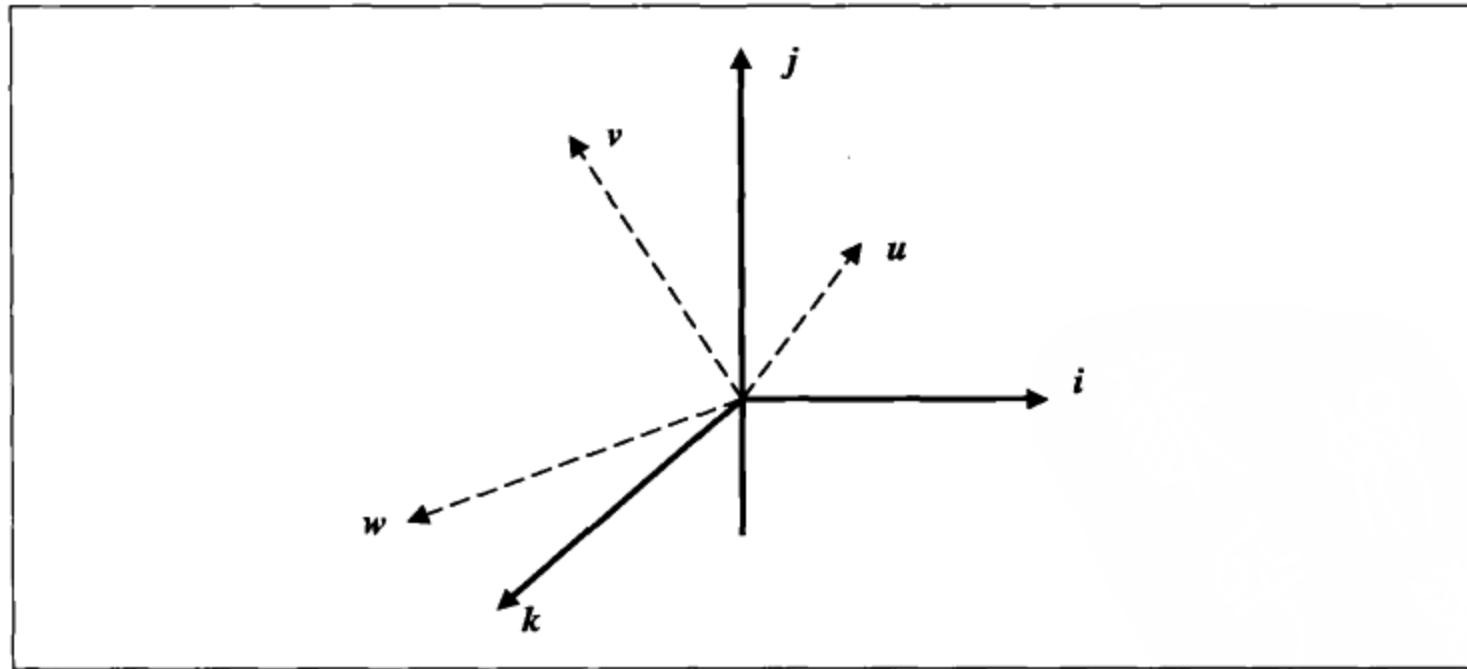


图 17-18 转换前和转换后的坐标系

在 original 坐标系下，如果一个单位向量沿 X 轴方向且其值为(1,0,0)，那么在 new 坐标系下是单位向

量 \mathbf{u} , 用 (u_x, u_y, u_z) 来表示。注意这个 (u_x, u_y, u_z) 也是 original 坐标系的值。由于是单位向量, 因此其模为 1。转换前后坐标系单位的向量值如表 17-4 所示。

表 17-4 转换前后坐标系单位的向量值

original 坐标系	original 坐标系
X 轴方向的单位向量 $(1, 0, 0)$	新坐标系下的 X 轴方向向量 (u_x, u_y, u_z)
Y 轴方向的单位向量 $(0, 1, 0)$	新坐标系下的 Y 轴方向向量 (v_x, v_y, v_z)
Z 轴方向的单位向量 $(0, 0, 1)$	新坐标系下的 Z 轴方向向量 (w_x, w_y, w_z)

表中 (u_x, u_y, u_z) 、 (v_x, v_y, v_z) 和 (w_x, w_y, w_z) 3 个向量幅度都为 1, 且两两正交。从 original 坐标系变换到 new 坐标系的变换矩阵为:

$$\begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix}$$

这个特殊的矩阵即旋转矩阵, 其作用是把一个坐标集合旋转后得到另一个坐标集合。任何由 3 个正交行向量构成的矩阵都是一个旋转矩阵, 任何旋转矩阵的 3 个行向量都是正交向量。我们也不难验证前面分别绕 X 轴、 Y 轴和 Z 轴的旋转矩阵的行向量两两正交, 幅度均为 1。而且该矩阵的逆非常特殊, 其逆矩阵即它的转置矩阵。

$$\begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix}$$

该矩阵可以实现坐标的反变换, 表示如下:

$$(u_x \ u_y \ u_z) \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} = (1 \ 0 \ 0)$$

同理, 该旋转矩阵也很容易扩展到齐次坐标下的 4×4 矩阵:

$$\begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

17.3 从 3D 物体到 2D 图形

现实世界的物体都是 3D 的, 但是最终显示在屏幕上则变为一个 2D 图形, 这个过程类似照相机成像的过程。我们照相时, 往往是照相者对准需要拍摄的物体。然后调整焦距, “咔嚓”一声按下快门完成整个过程。如图 17-19 所示, 其中蕴含至少 3 次坐标变换。

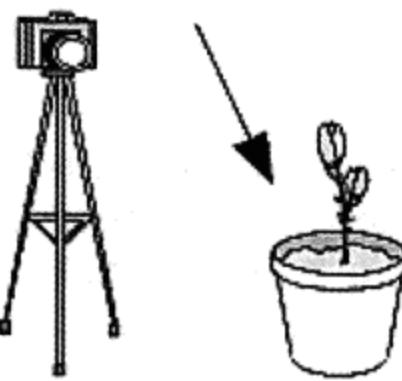


图 17-19 照相的过程^[3]

17.3.1 3 个坐标系

1. 第 1 个坐标系——物方坐标系

第 1 个坐标系我们称为“物方坐标系”或者“世界坐标系”，在这个坐标系下描述所拍摄物体和照相机位置的坐标。为了方便计算，我们往往会把物体放在坐标的原点。照相机的位置坐标指向方向及其正上方的方向，这两个方向均用向量表示。

如图 17-20 所示，物体摆放在坐标原点 $(0,0,0)$ ，而照相机的位置坐标为 $(0,0,5)$ 。其指向方向为 Z 轴的负方向，用一个向量 $(0,0,-1)$ 来表示；其正上方方向指向 Y 轴的正方向，用一个向量 $(0,1,0)$ 来表示。

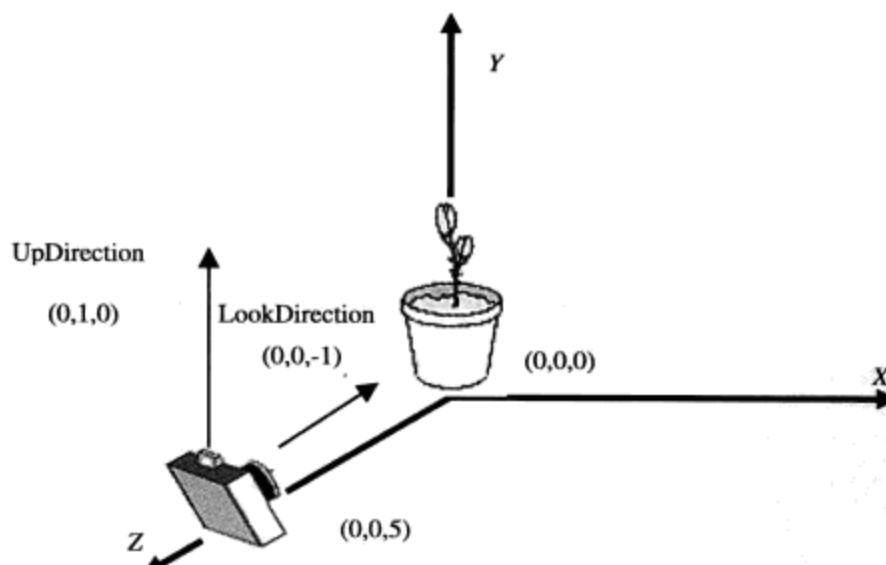


图 17-20 物方坐标系

WPF 通过 Camera 对象来描述照相机，其 Position 属性用于描述其位置，在上例中该坐标值为 $(0,0,5)$ ；UpDirection 属性用于描述照相机的正上方方向，由于该向量只关注方向信息，而不关注幅度信息，因此不妨用单位向量 $(0,1,0)$ 来表示；LookDirection 属性用来描述照相机的指向方向，该向量也只

关注方向信息，而不关注幅度信息，因此也可以用单位向量(0,0,-1)来表示。

2. 第2个坐标系——照相机坐标系

第2个坐标系我们称为“照相机坐标系”，其中照相机的所在位置永远是原点坐标点，而其X轴、Y轴和Z轴需要UpDirection和LookDirection两个向量来决定。在前例中照相机坐标系非常简单，其X、Y和Z轴均与世界坐标系的X、Y和Z轴平行，如图17-21所示。

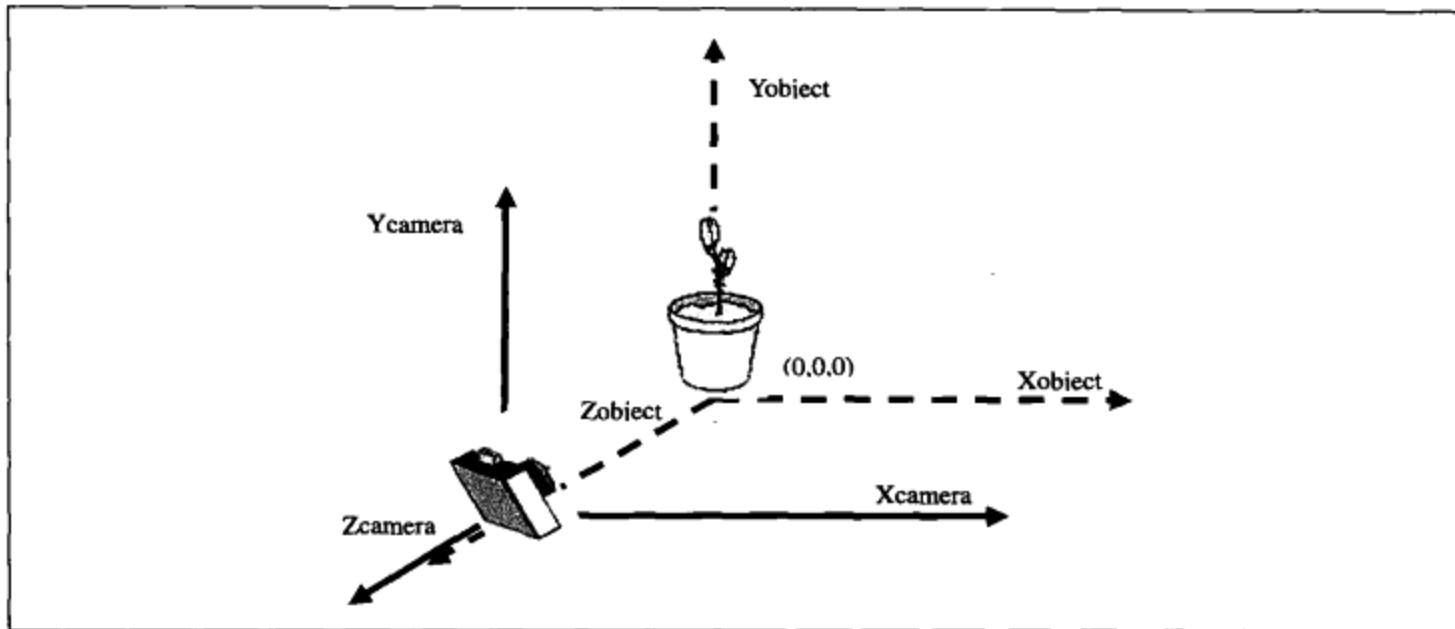


图 17-21 照相机坐标系

照相机坐标系的X、Y和Z轴可以通过下面的方法计算得出，首先最容易确定Z轴，它与LookDirection方向相反：

```
Vector3D Zcamera = -LookDirection;
```

并且经过归一化处理。

```
Zcamera.Normalize();
```

WPF3D中并不要求UpDirection和LookDirection正交，因此无法直接得到Y轴。但是我们知道，照相机的X轴垂直于UpDirection和LookDirection构成的平面，因此可以通过叉积运算得到照相机的X轴：

```
Vector3D Xcamera = Vector3D.CrossProduct(UpDirection, ZCamera);
Xcamera.Normalize();
```

确定照相机的Z轴和X轴后可以通过叉积运算得到Y轴：

```
Vector3D Ycamera = Vector3D.CrossProduct(ZCamera, XCamera);
```

由于Z轴和X轴向量已经是单位向量，因此不需要执行归一化处理。

3. 第3个坐标系——窗口坐标系

第3个坐标系即窗口坐标系，照相机将物体映射到照片上，而计算机将物体映射到窗口中。这个坐标系是一个2D坐标系，原点在窗口的左上角。而X轴方向向右，Z轴方向向下。

真实的3D物体成为窗口的2D图形，从数学的角度上来说就是将物体在世界坐标系下的坐标变换到窗口坐标系下的坐标，这样的变换需要多个步骤，其中扮演着一个至关重要的角色就是照相机 Camera。

17.3.2 Camera 对象

图17-22所示为Camera的类层次结构。

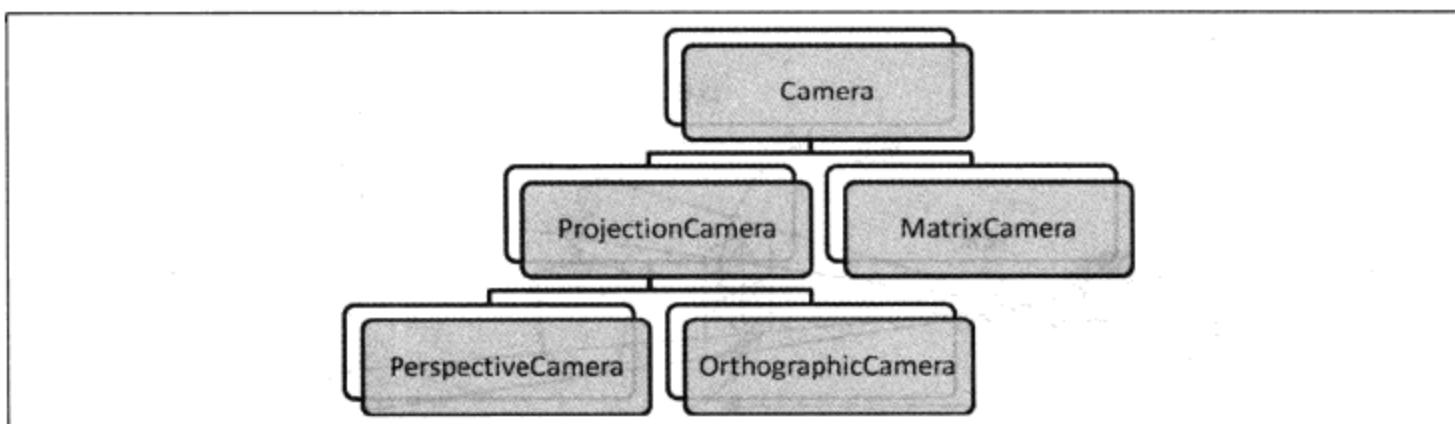


图17-22 Camera的类层次结构

从图中可以看出 Camera 派生了两个类，即 ProjectionCamera 和 MatrixCamera。从 ProjectionCamera 派生了两个类，即 OrthographicCamera 和 PerspectiveCamera。这两种照相机代表两种不同的投影方式，即透视投影和正射投影。

投影的目的是为了定义照相机的观察范围，称为“视景体”（a viewing volume）。视景体决定了一个物体如何映射到屏幕上（即通过正射还是透视投影），并且定义了会被裁减到最终图像之外的物体。

1. 透视投影

透视投影和照相机照相或者人眼观察事物的原理相同，其显著的特征是物体距离照相机越远，则越小；反之越大。透视投影的视景体可以看成是一个金字塔的平截头体，我们可以通过定义近侧平面的左侧、右侧、顶部、底端，以及距离照相机的最近和最远距离来确定该平截头体的范围，如图17-23所示。

因为我们比较难确定近侧平面的左侧、右侧、顶部和底端的值，因此可以换一种方式来指定视景体的范围。如图17-24所示，我们用视角和纵横比取代原来的左侧、右侧、顶部和底端4个值。

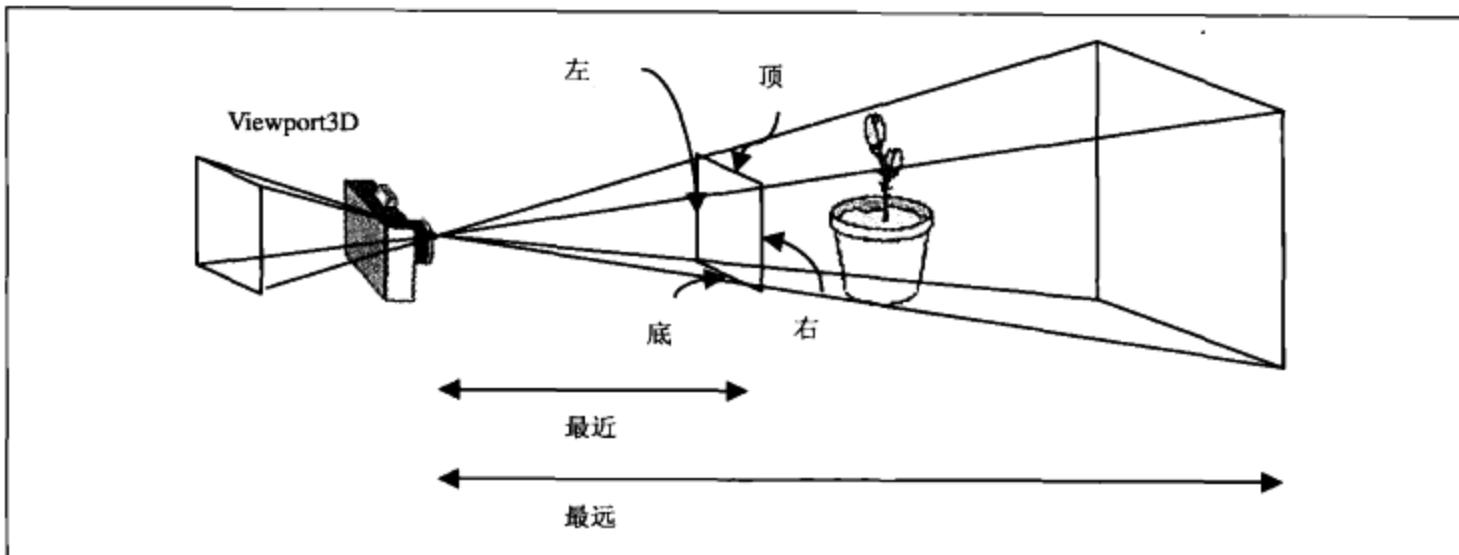


图 17-23 透视投影

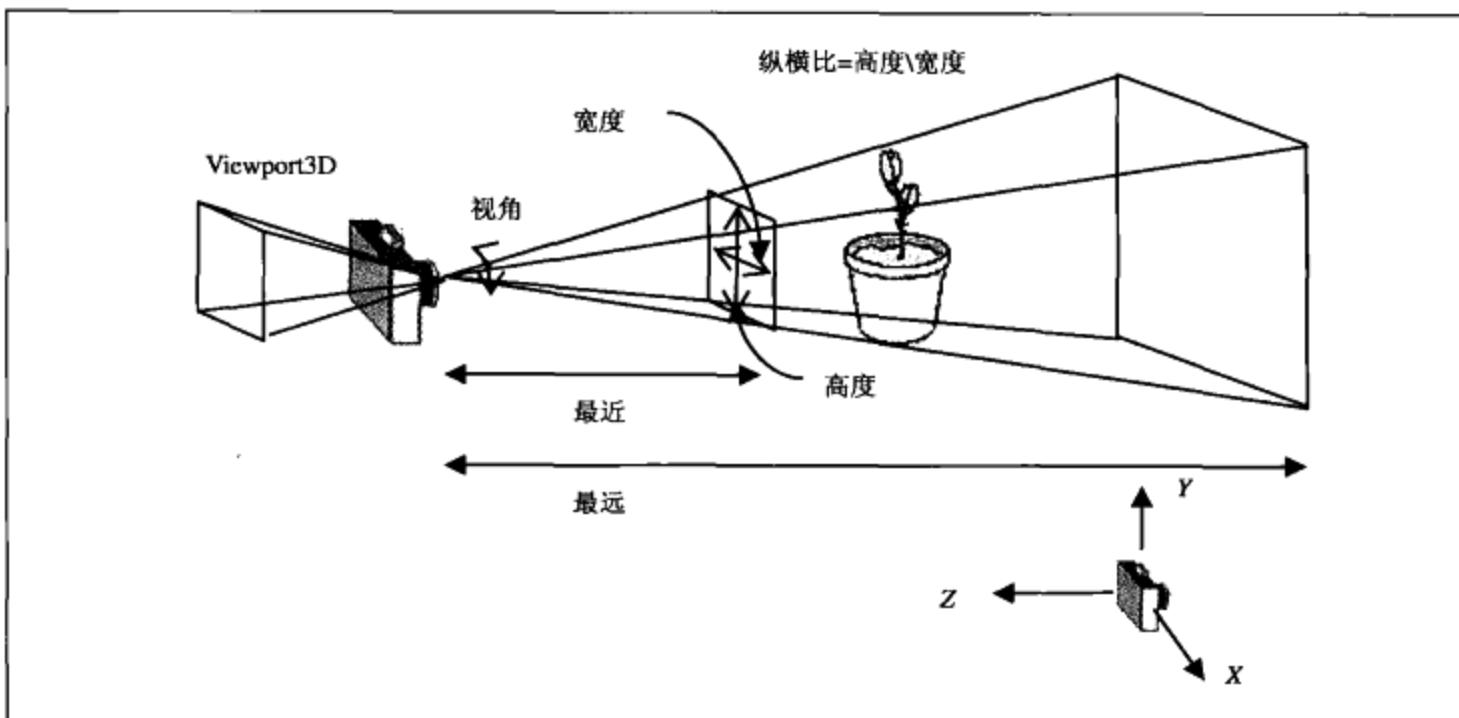


图 17-24 另一种方式来指定视景体的范围

当视角大时，相当一部广角镜头的照相机。视景体的范围越大，成像的物体越小；当视角小时，相当一部远摄镜头的照相机。视景体的范围越小，成像的物体就越大。实际上近侧平面和 Viewport3D 的纵横比相同，因此可以认为纵横比即 Viewport3D 的宽度和高度之比。

PerspectiveCamera 是代表透视投影的照相机，除了 Camera 的 Position、UpDirection 和 LookDirection 这 3 个属性以外，PerspectiveCamera 的父类 ProjectionCamera 定义了一个最近的平面距离 NearPlaneDistance 和一个最远的平面距离 FarPlaneDistance。属性 NearPlaneDistance 默认的值是 0.125，即照相机前面 0.125 个单位长度内所有图形是不可见的。属性 FarPlaneDistance 默认值是 Double.Infinity，即最远距离是无限大。如果想去除照相机一定长度之外的所有图形，那么可以设定一个具体的值。

假设照相机位于 Z 轴，即属性 Positon 为 (0,0,D)，其中 D 表示照相机到 XY 平面的距离。属性

LookDirection 为 $(0,0,-1)$ ，即直接指向 Z 轴负向。属性 UpDirection 是 $(0,1,0)$ ，即垂直向上。设属性 FieldOfView 的值为 F ，那么该照相机可观察的 X 轴的范围 W 是多少？图 17-25 所示为从上往下看的俯视图。

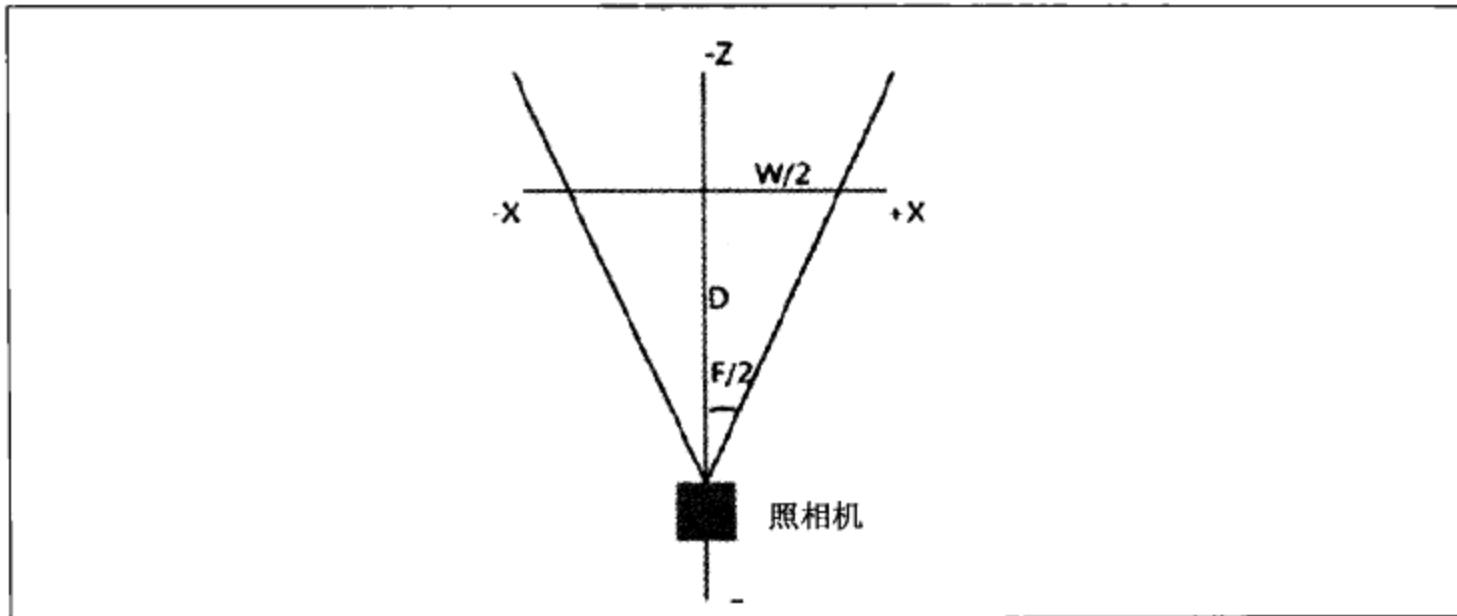


图 17-25 从上往下看的俯视图

如图所示，这是一个简单的三角函数关系， W 的计算公式如下：

$$\tan \frac{F}{2} = \frac{w/2}{D}$$

$$W = 2D \tan \frac{F}{2}$$

实际垂直可以见的范围 H 取决于 Viewport3D 的宽度和高度之比，如下所示：

$$H = W \times \frac{\text{Viewport3D的高度}}{\text{Viewport3D的宽度}}$$

图 17-26 所示是一个立方体，其正前方一面在 XY 平面上。左下角点坐标为 $(-0.5,-0.5,0)$ ，右上角点坐标为 $(0.5,0.5,0)$ ；其后方在 $Z=-4$ 的平面上，左下角点坐标为 $(-0.5,-0.5,4)$ ，右上角点坐标为 $(0.5,0.5,-4)$ 。透视投影的照相机的位置为 $(0,0,2)$ ，直接指向 Z 轴负方向。向上的方向为垂直向上，视角为 30 度。

我们可以看到该立方体的正前方几乎占据整个 Viewport3D，这是因为当视角为 30 度时，照相机距离物体为两个单位长度。计算后在 X 轴上可见的宽度为 1.07 个单位长度，而立方体的正前方在 XY 平面上的宽度为 1 个单位长度。

当我们不断调整视角时，立方体的正前方的面会越来越小，如表 17-5 所示。

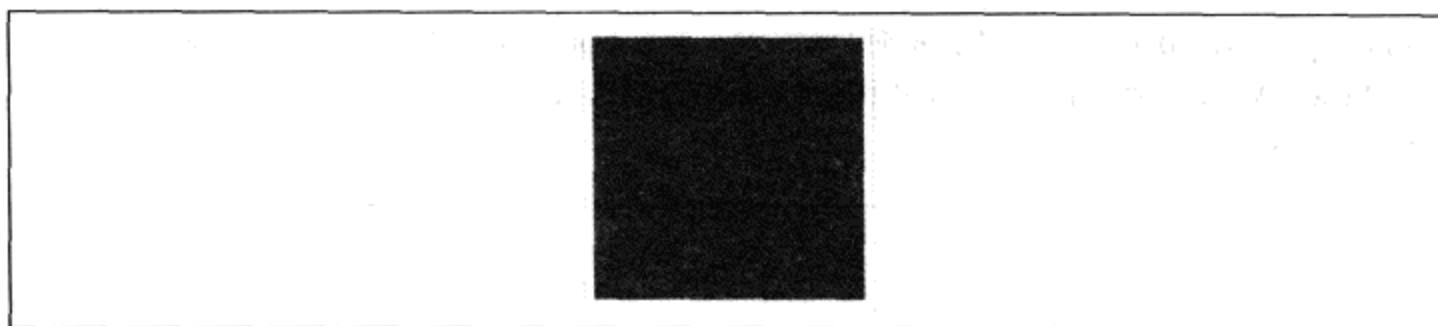


图 17-26 立方体

表 17-5 立方体的正前方的面随视角变小

照相机的距离均为 2		
视角为 30°，X 轴可见宽度约为 1.07	视角为 60°，X 轴可见宽度约为 2.3	视角为 90°，X 轴可见宽度约为 4.0

同样我们可以将照相机调整到合适的位置，使其能够显示立体效果，`PerspectiveCamera` 的属性如代码 17-13 所示。

```
<PerspectiveCamera x:Name="camera" Position="-2.5 1 6"  
LookDirection="0 0 -1" FieldOfView="{Binding  
ElementName=FieldOfViewAngle, Path=Value}"/>
```

代码 17-13 `PerspectiveCamera` 的属性

如图 17-27 所示是一个近大远小的透视图。

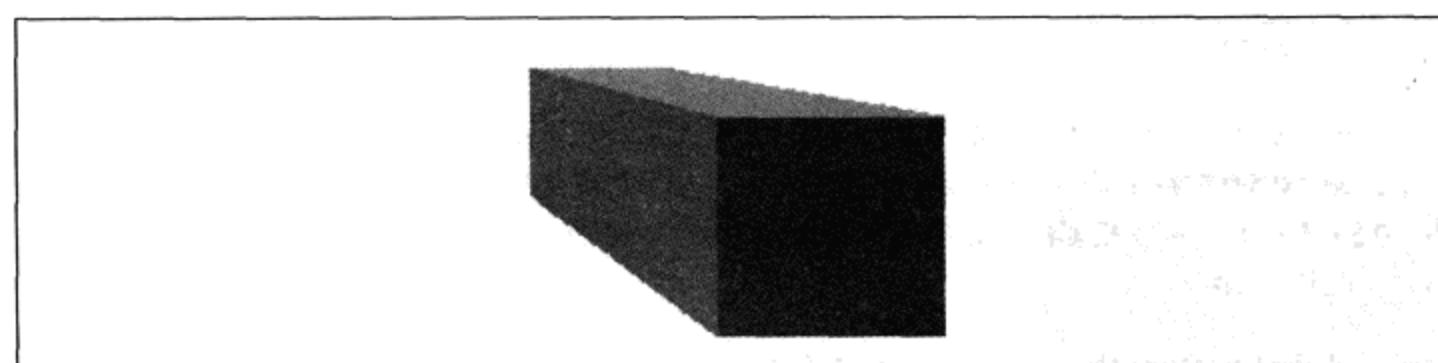


图 17-27 一个近大远小的立方体透视图

2. 正射投影

在正射投影下视景体是一个平行的长方体，物体和照相机之间的距离并不影响其看上去的大小，如图 17-28 所示。

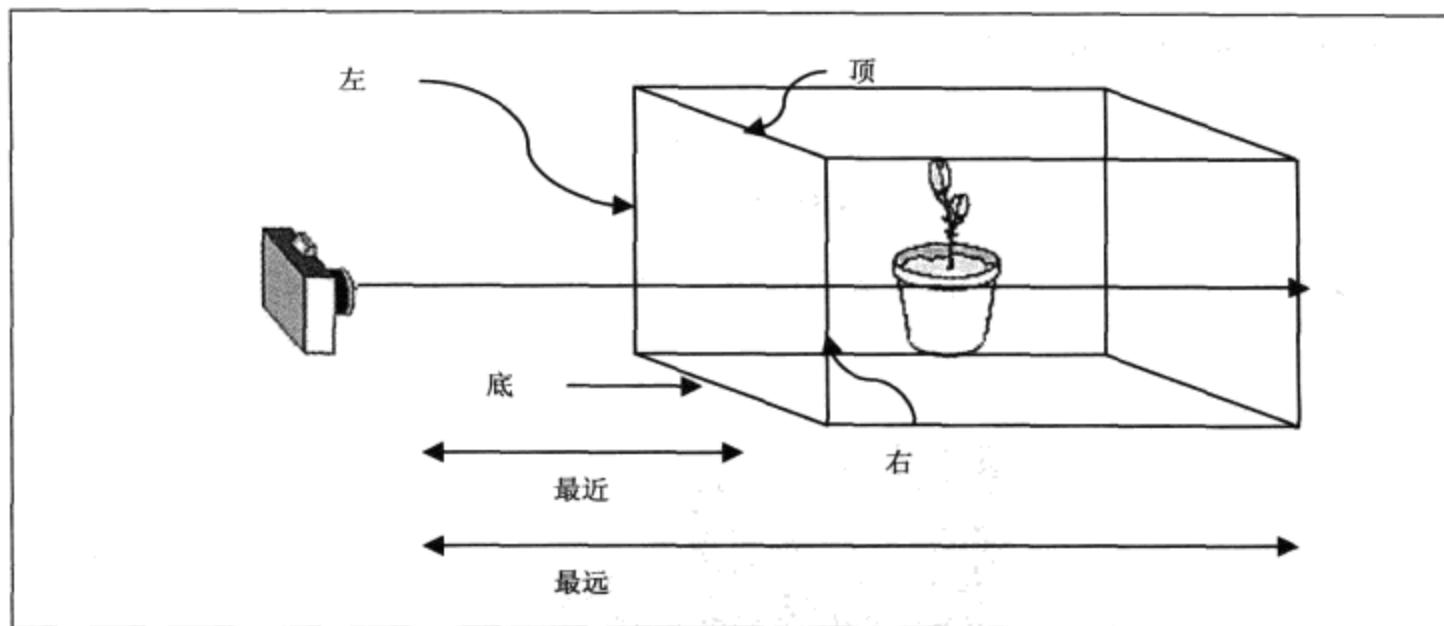


图 17-28 正射投影下的视景体

物体经过正射投影后，可以保持其实际大小及之间的角度，因此常用在建筑蓝图和计算机辅助设计中。正射投影的视景体也可以通过左侧、右侧、顶部、底端，以及距离照相机的最近和最远距离来确定其范围，但是和透视投影一样，为了使用直观，一般通过宽度和纵横比来决定。

在 WPF3D 中用 `OrthographicCamera` 类来表示正射投影的照相机，和 `PerspectiveCamera` 不同，它没有 `FieldOfView` 属性，而多了一个 `Width` 属性，默认值为 2。我们将视景体的正前方的一个面称为“投影面”，`OrthographicCamera` 通过 `Position` 确定该面的中心位置。该投影平面与向量 `LookDirection` 垂直，并且该平面的正上方方向与 `UpDirection` 一致。平面的宽度为 `Width` 个单位长度，平面的高度则由 `Width` 和 `Viewport3D` 的高度和宽度比确定。

下例仍然是前面的矩形，只不过将照相机换为 `OrthographicCamera`，其初始状态如代码 17-14 所示。

```
<OrthographicCamera x:Name="camera" Position="0 0 2"
    LookDirection="0 0 -1" UpDirection="0 1 0", Width="4">
```

代码 17-14 照相机的初始状态

运行结果如图 17-29 所示，同样我们也只能看到该立方体的正前方。但是与透视投影不同，当视线一直和立方体平行，即投影面和立方体的前后面平行，则无论如何调整照相机的位置，均只能看到立方体的前面或者后面，而看不到其他面。

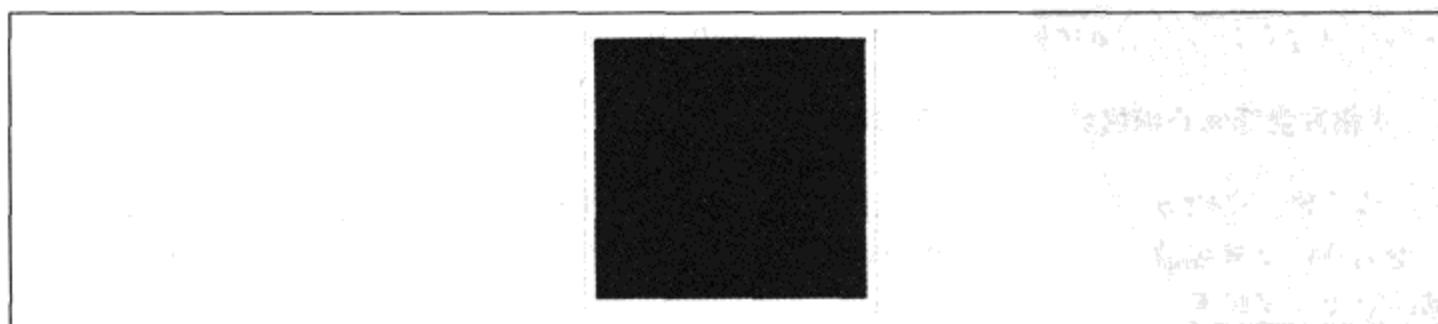


图 17-29 运行结果

当调整 OrthographicCamera 值时可以看出正射投影下的立方体，如代码 17-15 所示。

```
<OrthographicCamera x:Name="camera" Position="-2 1.5 4"  
LookDirection="2 -1 -4" UpDirection="0 1 0" Width="5"/>
```

代码 17-15 调整 OrthographicCamera 的值

运行结果如图 17-30 所示，图形的后端似乎比前端大，但这只是一种光学上的错觉，实际上前端和后端是一样大的。

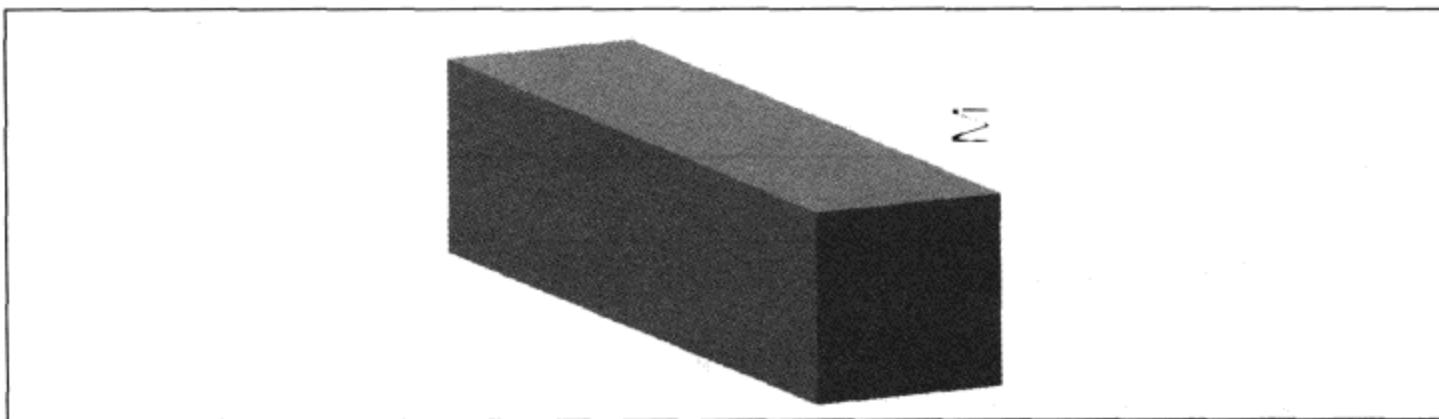


图 17-30 运行结果

3. 照相机中隐藏的矩阵变换

PerspectiveCamera 和 OrthographicCamera 均派生自 ProjectionCamera，从 Camera 派生的另外一个分支为 MatrixCamera。它没有 FieldOfView、Width、Position、UpDirection 和 LookDirection 这样的属性。而只有两个 4×4 的矩阵，分别为观察矩阵 ViewMatrix 和投影矩阵 ProjectionMatrix，这两个矩阵与之相对应的变换称为“观察变换”和“投影变换”。

从本质上说，PerspectiveCamera 和 OrthographicCamera 能够从 3D 物体变换为 2D 图形，也要计算得到这两个矩阵，这两个矩阵的值取决于 Position、UpDirection、LookDirection、FieldOfView 或者 Width 这样的属性。

17.3.3 坐标变换

从 3D 物体到 2D 图形，从数学的角度上来看经历了 3 次坐标变换。第 1 次是从物方坐标到照相机坐标（观察变换）；第 2 次是从照相机坐标下的视景体到一个单位立方体（投影变换）的变换；最后是这个单位立方体到屏幕的转换视口变换，对应 WPF3D，即 ViewPort3D 的客户区。

1. 从物方坐标到照相机坐标——观察变换

这一步将物体在物方坐标系下的坐标 (x_0, y_0, z_0) 变换为照相机坐标系下的坐标 (x_c, y_c, z_c) ，在 17.2.4 节中讨论的几何变换缩放、旋转或者平移的对象是点、向量或者一个形状现在则是在缩放、旋转或者平移的对象是坐标系，如图 17-31 所示。

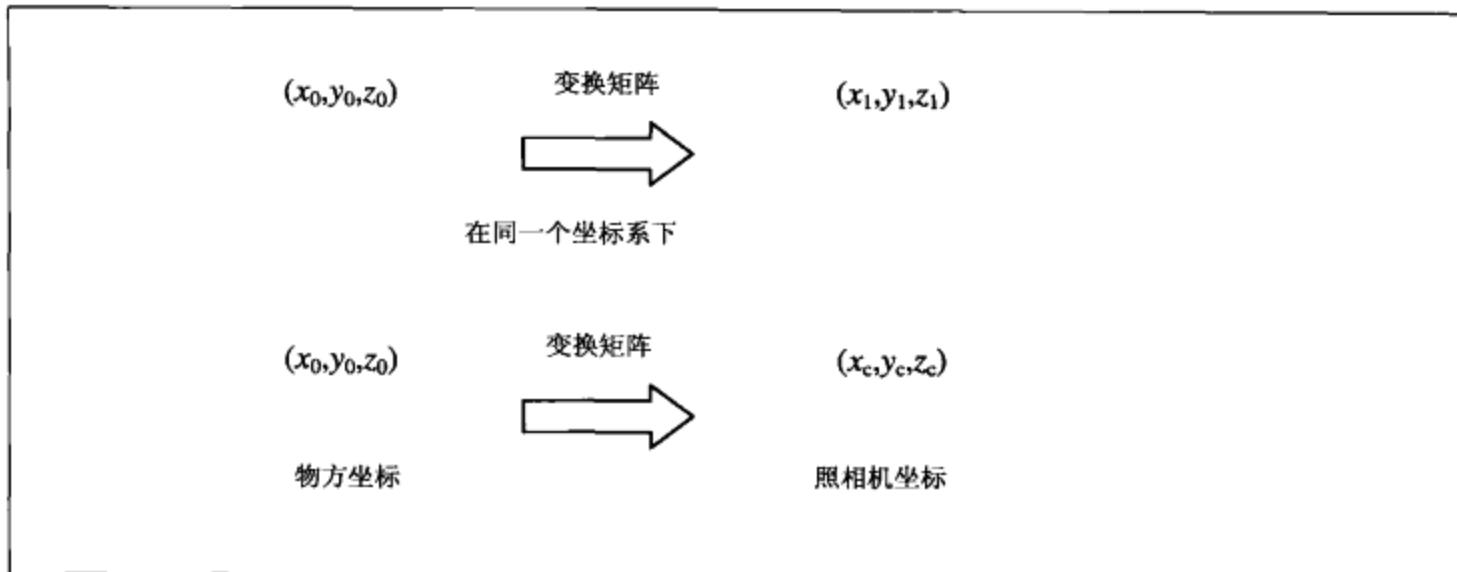


图 17-31 观察变换

如图 17-32 所示，从物方坐标 O 变换为照相机坐标 C 可以认为是一个复合变换。物方坐标系经过旋转变换变为一个中间坐标系 M ，然后将该坐标系执行平移变换变为照相机坐标。

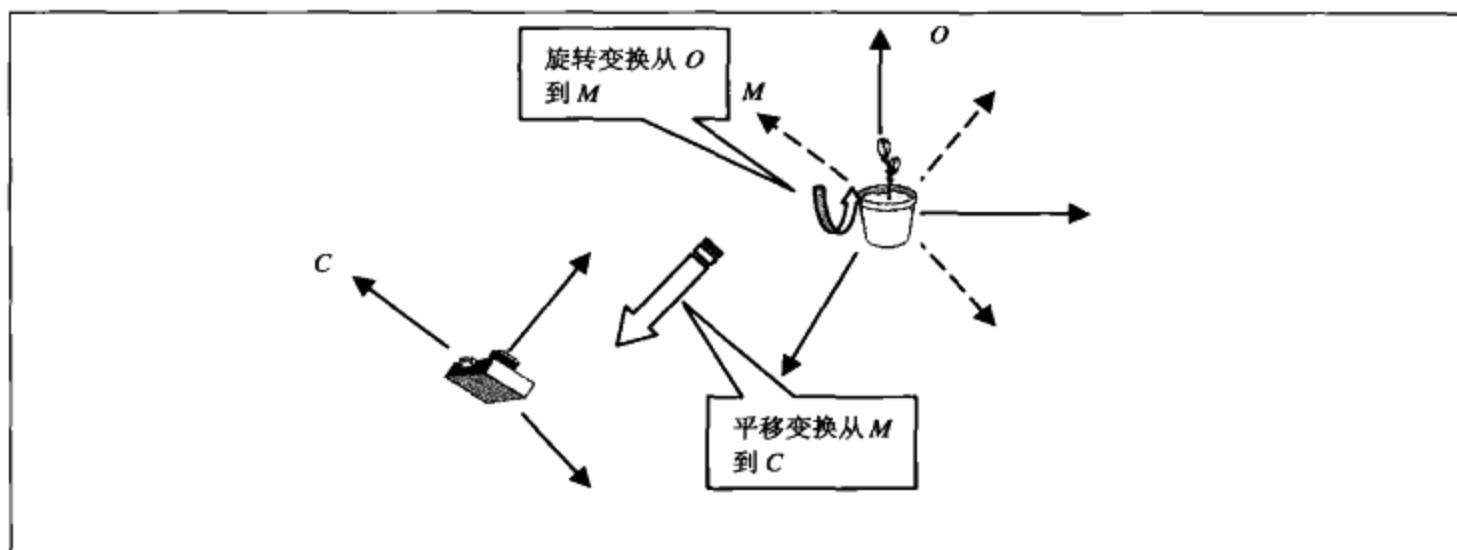


图 17-32 从物方坐标 O 变换到照相机坐标 C

(1) 旋转矩阵

我们首先考察旋转矩阵，在 M 坐标系下 3 个轴的单位向量分别为 $M_x(1,0,0)$ 、 $M_y(0,1,0)$ 和 $M_z(0,0,1)$ ，分别对应在 O 坐标系下向量为 $(M_x.X, M_x.Y, M_x.Z)$ 、 $(M_y.X, M_y.Y, M_y.Z)$ 和 $(M_z.X, M_z.Y, M_z.Z)$ ，如表 17-6 所示。

表 17-6 M 坐标系与 O 坐标系的对应关系

O 坐标系	M 坐标系
$(M_x.X, M_x.Y, M_x.Z)$	X 轴方向的单位向量 $(1,0,0)$
$(M_y.X, M_y.Y, M_y.Z)$	Y 轴方向的单位向量 $(0,1,0)$
$(M_z.X, M_z.Y, M_z.Z)$	Z 轴方向的单位向量 $(0,0,1)$

这个表和 17.2.4 节非常类似，不同在于前者是在一个坐标系下对点和向量执行几何变换；后者是坐标系之间的变换，因此旋转矩阵也有所不同：

$$\begin{pmatrix} Mx.X & My.X & Mz.X & 0 \\ Mx.Y & My.Y & Mz.Y & 0 \\ Mx.Z & My.Z & Mz.Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

向量 Mx 、 My 和 Mz 是 M 坐标系下的 3 个轴的单位向量，由于 M 坐标系的 3 个轴和 C 坐标系分别平行，因此 Mx 、 My 和 Mz 也可以认为是 C 坐标系下的 3 个轴的单位向量。我们在前面推导过照相机坐标系的 3 个轴向量的值，和照相机的 UpDirection 和 LookDirection 属性相关：

```
Vector3D Mz=-LookDirection;  
Mz.Normalize();  
Vector3D Mx= Vector3D.CrossProduct(UpDirection, Mz);  
Mx.Normalize();  
Vector3D My= Vector3D.CrossProduct(Mz, Mx);
```

注意这里计算得出的向量 Mx 、 My 和 Mz 的值均为在 O 坐标系下的值。

(2) 平移矩阵

平移变换的平移位置和照相机的 Position 属性相关，但是现在的 Position 还是 O 坐标系下的值，因此需要将其变换到 M 坐标系下：

(PositionM.X,PositionM.Y,PositionM.Z,1)

$$= (\text{PositionO.X}, \text{PositionO.Y}, \text{PositionO.Z}, 1) \begin{pmatrix} Mx.X & My.X & Mz.X & 0 \\ Mx.Y & My.Y & Mz.Y & 0 \\ Mx.Z & My.Z & Mz.Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

分别计算 PositionM 的 X 、 Y 和 Z 值：

$$\left\{ \begin{array}{l} \text{PositionM.X} = \text{PositionO.X} * Mx.X + \text{PositionO.Y} * Mx.Y + \text{PositionO.Z} * Mx.Z = \text{PositionO.Mx} \\ \text{PositionM.Y} = \text{PositionO.X} * My.X + \text{PositionO.Y} * My.Y + \text{PositionO.Z} * My.Z = \text{PositionO.My} \\ \text{PositionM.Z} = \text{PositionO.X} * Mz.X + \text{PositionO.Y} * Mz.Y + \text{PositionO.Z} * Mz.Z = \text{PositionO.Mz} \\ 1 = 1 \end{array} \right.$$

照相机在 M 坐标系下的坐标位置为 (PositionM.X,PositionM.Y,PositionM.Z)，在 C 坐标系下的坐标位置为 (0,0,0)，平移矩阵为：

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

$$\begin{cases} d_x = -PositionM.X \\ d_y = -PositionM.Y \\ d_z = -PositionM.Z \end{cases}$$

可以检验坐标(PositionM.X, PositionM.Y, PositionM.Z)经过该平移矩阵变换变为预期的(0,0,0)。

(3) 组合

物方坐标 O 经过旋转变成中间坐标系 M , 然后经过平移变成照相机坐标系, 因此其变换矩阵为:

$$\begin{pmatrix} Mx.X & My.X & Mz.X & 0 \\ Mx.Y & My.Y & Mz.Y & 0 \\ Mx.Z & My.Z & Mz.Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix} = \begin{pmatrix} Mx.X & My.X & Mz.X & 0 \\ Mx.Y & My.Y & Mz.Y & 0 \\ Mx.Z & My.Z & Mz.Z & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

该矩阵即是 MatrixCamera 的观察矩阵。

2. 投影变换

理想的情况是将照相机坐标系下的坐标(x_c, y_c, z_c)直接变换为窗口坐标系下(x_w, y_w), 我们知道透视投影的视景体是一个金字塔的平截头体, 而正射投影是一个与视线平行的长方体。在投影变换中都要将该视景体变成一个标准的单位立方体, 其坐标范围如下。

(1) X 坐标范围为 $-1 \sim +1$ 。

(2) Y 坐标范围为 $-1 \sim +1$ 。

(3) Z 坐标范围为 $0 \sim 1$ 。

实际上投影变换矩阵和下面的几个属性相关。

(1) NearPlaneDistance 和 FarPlaneDistance 属性: 在 ProjectionCamera 类中定义, 并被 PerspectiveCamera 和 OrthographicCamera 类继承。

(2) Width 或 FieldOfView 属性: Width 是 OrthographicCamera 的属性, FieldOfView 是 PerspectiveCamera 的属性。

(3) Viewport3D 的高度和宽度比。

对于正射和透视投影, 我们要分别推导其投影变换矩阵。

(1) 正射投影的变换矩阵

正射投影变换的第 1 步是将照相机坐标系 C 平移到视景体的前面, 变换为一个中间坐标系 M 。然后经过缩放变换到坐标系 N , 在该坐标系下视景体就是一个单位立方体, 如图 17-33 所示。

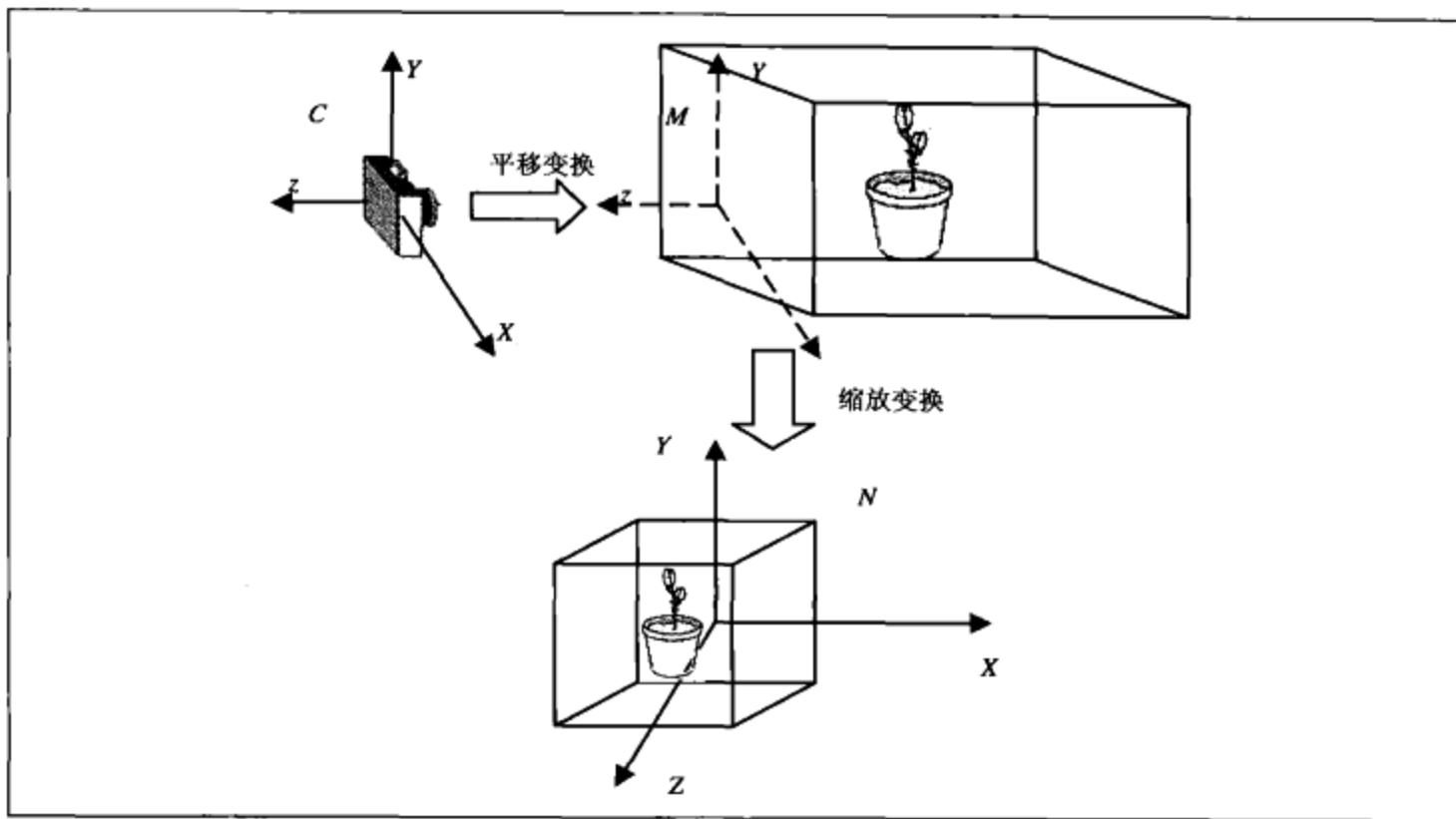


图 17-33 正射投影的两步变换

在 C 坐标系下坐标原点为 $(0,0,0)$ ，在 M 坐标系下为 $(0,0,\text{NearPlaneDistance})$ ，简写为“ $(0,0,z\text{Near})$ ”。由此可以推导出从 C 坐标系变换到 M 坐标系下的平移矩阵如下：

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & z\text{Near} & 1 \end{pmatrix}$$

从 M 坐标系到 N 坐标系经历一个缩放变换，其矩阵如下所示：

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

我们需要计算 s_x 、 s_y 和 s_z 的值，在 M 坐标系下视景体的正前方的宽度是 Width ，在 N 坐标系下为 2（ X 坐标在 $-1 \sim +1$ 范围内），因此可以计算：

$$s_x = 2 / \text{Width}。$$

在 M 坐标系下视景体的高度由 Width 和 Viewport3D 的高度和宽度比决定：

$$h = \text{Width} \times \frac{\text{Viewport3D 的高度}}{\text{Viewport3D 的宽度}}$$

在 N 坐标系下，其高度为 2，因此可以计算 s_y ：

$$s_y = \frac{2}{n} = 2 / (\text{Width} \times \frac{\text{Viewport3D的高度}}{\text{Viewport3D的宽度}})$$

假定 Viewport3D 的宽度为 800Dpi, 高度为 400Dpi, 则 $s_y = 4/\text{Width}$ 。

在 M 坐标系下, 视景体的长度为 $\text{FarPlaneDistance} - \text{NearPlaneDistance}$, 简写为“zFar-zNear”, 那么在 N 坐标系下长度为 1。而且 M 和 N 坐标系的 Z 轴方向相反, 因此可以计算 s_z :

$$s_z = -\frac{1}{\text{zFar} - \text{zNear}}$$

将 s_x 、 s_y 和 s_z 的值代入到旋转矩阵中, 得到:

$$\begin{pmatrix} 2/\text{Width} & 0 & 0 & 0 \\ 0 & 2 / (\text{Width} \times \frac{\text{Viewport3D的高度}}{\text{Viewport3D的宽度}}) & 0 & 0 \\ 0 & 0 & -\frac{1}{\text{zFar} - \text{zNear}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

从 C 坐标系变换到 N 坐标系的变换矩阵, 即投影矩阵为:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \text{zNear} & 1 \end{pmatrix} \begin{pmatrix} 2/\text{Width} & 0 & 0 & 0 \\ 0 & 2 / (\text{Width} \times \frac{\text{Viewport3D的高度}}{\text{Viewport3D的宽度}}) & 0 & 0 \\ 0 & 0 & -\frac{1}{\text{zFar} - \text{zNear}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 2/\text{Width} & 0 & 0 & 0 \\ 0 & 2 / (\text{Width} \times \frac{\text{Viewport3D的高度}}{\text{Viewport3D的宽度}}) & 0 & 0 \\ 0 & 0 & -\frac{1}{\text{zFar} - \text{zNear}} & 0 \\ 0 & 0 & -\frac{1}{\text{zFar} - \text{zNear}} & 1 \end{pmatrix}$$

(2) 透视投影的变换矩阵

透视投影比正射投影复杂, 因为其视景体并不是一个简单的长方体, 而是一个金字塔的平截头体。从平截头体到单位立方体已经不是前面所说的平移、缩放或者旋转这样基础的变换, 而是一个非仿

射变换²。由于不能用常规的几何变换方法推导，所以直接给出如下透视投影的变换矩阵公式：

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & -1 \\ 0 & 0 & dz & 1 \end{pmatrix}$$

$$s_x = \frac{1}{\tan(\text{FieldOfView}/2)}$$

$$s_y = s_x \times \frac{\text{Viewport3D 的宽度}}{\text{Viewport3D 的高度}}$$

$$s_z \begin{cases} -1 & zFar = \text{Double.PositiveInfinity} \\ \frac{zFar}{zNear - zFar} & zFar = \text{Double.PositiveInfinity} \end{cases}$$

$$d_z = s_z \times zNear$$

3. 从单位立方体到窗口坐标——视口变换

当经过观察变换和投影变换之后，物方坐标系下的坐标最终变换为一个单位立方体。Z 坐标变为一个深度坐标，WPF3D 会根据对象变换后的 Z 值判断哪些可见，哪些不可见。并根据 Viewport3D 的实际大小，变换到 Viewport3D 中。

4. 用 MatrixCamera 模拟 PerspectiveCamera 和 OrthographicCamera

通过计算 MatrixCamera 的观察矩阵和投影矩阵来模拟 PerspectiveCamera 的效果。

PerspectiveCamera 的设置如代码 17-16 所示，同时 Viewport3D 的宽度为 600，高度为 300。

```
<PerspectiveCamera x:Name="camera" Position="-2.5 1 6"
                    LookDirection="0 0 -1" FieldOfView="{Binding
ElementName=FieldOfViewAngle, Path=Value}"/>
```

代码 17-16 PerspectiveCamera 的设置

新建一个 XAML 文件，模型仍然是提到的立方体，只不过照相机改成了 MatrixCamera，如代码 17-17 所示（详见 mumu_3DDemo 工程）。

```
<Page x:Class="mumu_3DDemo.PerspectiveByMatrixCamera"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="PerspectiveByMatrixCamera" Loaded="Page_Loaded">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Viewport3D Grid.Row="0" Width="600" Height="300" x:Name="viewport">
```

² 平移、缩放、旋转和错切变换统称为“仿射变换”，在数学上通常指保持有限性。仿射变换不会将有限的 3D 空间点变换为一个坐标无限的点，在变换领域中它主要指带有附加因子的线性变换。

```

<ModelVisual3D>
    <ModelVisual3D.Content>
        <Model3DGroup>
            .....
            <AmbientLight Color="#404040" />
            <DirectionalLight Color="#C0C0C0" Direction="2 -3 -1" />

        </Model3DGroup>
    </ModelVisual3D.Content>
</ModelVisual3D>
<Viewport3D.Camera>
    <MatrixCamera x:Name="cam"/>
</Viewport3D.Camera>
</Viewport3D>

</Grid>
</Page>

```

代码 17-17 一个 XAML 文件

在该文件中明确地指定了 Viewport3D 的宽度和高度, 同时命名 Viewport3D 和 MatrixCamera。在 Page 的 Loaded 事件中为 MatrixCamera 指定观察矩阵和投影矩阵, 如代码 17-18 所示。

```

private void Page_Loaded(object sender, RoutedEventArgs e)
{
    // 透视投影的若干参数
    Vector3D LookDirection = new Vector3D(0,0,-1);
    Vector3D Position = new Vector3D(-2.5, 1, 6);
    Vector3D UpDirection = new Vector3D(0, 1, 0);
    double FieldOfView= 90;
    // 透视投影这两个值没有设置, 取默认值
    double zFar = Double.PositiveInfinity;
    double zNear = 0.125;

    // 计算 M 坐标系下的 3 个轴单位向量 Mx,My,Mz
    Vector3D Mz = -LookDirection;
    Mz.Normalize();
    Vector3D Mx = Vector3D.CrossProduct(UpDirection, Mz);
    Mx.Normalize();
    Vector3D My = Vector3D.CrossProduct(Mz, Mx);

    // 计算平移矩阵的 dx,dy,dz
    double dx = -Vector3D.DotProduct(Position, Mx);
    double dy = -Vector3D.DotProduct(Position, My);
    double dz = -Vector3D.DotProduct(Position, Mz);

    // 构建观察矩阵
    Matrix3D viewmatrix = new Matrix3D(Mx.X,My.X,Mz.X,0,
                                       Mx.Y,My.Y,Mz.Y,0,
                                       Mx.Z,My.Z,Mz.Z,0,
                                       dx, dy, dz,1);
    cam.ViewMatrix = viewmatrix;

    // 计算投影矩阵的 Sx,Sy,Sz,dz
    double FieldOfViewRadian = FieldOfView / 180 * Math.PI;
    double Sx = 1 / Math.Tan(0.5 * FieldOfViewRadian);
    double Sy = Sx * viewport.Width / viewport.Height;
    double Sz = -1;
    double Dz = Sz * zNear;
}

```

```

Matrix3D projectionmatrix = new Matrix3D(Sx, 0, 0, 0,
                                         0, Sy, 0, 0,
                                         0, 0, Sz, -1,
                                         0, 0, Dz, 1);
cam.ProjectionMatrix = projectionmatrix;
}

```

代码 17-18 设置 Viewport3D

同样也可以模拟 OrthographicCamera，最后 OrthographicCamera 的设置如代码 17-19 所示。

```
<OrthographicCamera x:Name="camera" Position="-2 1.5 4"
                    LookDirection="2 -1 -4" UpDirection="0 1 0" Width="5"/>
```

代码 17-19 模拟 OrthographicCamera

在 Page 的 Loaded 事件中为 MatrixCamera 指定观察矩阵和投影矩阵，如代码 17-20 所示。

```

// 正射投影的若干参数
Vector3D LookDirection = new Vector3D(2, -1, -4);
Vector3D Position = new Vector3D(-2, 1.5, 4);
Vector3D UpDirection = new Vector3D(0, 1, 0);
double Width = 5;
// 正射投影这两个值没有设置，取默认值
// 为了避免运算失败，没有将 zFar 取 Double.PositiveInfinity，而是取一个相对
// 大的数值
double zFar = 1e10;
double zNear = 0.125;

// 计算 M 坐标系下的 3 个轴单位向量 Mx、My 和 Mz
Vector3D Mz = -LookDirection;
Mz.Normalize();
Vector3D Mx = Vector3D.CrossProduct(UpDirection, Mz);
Mx.Normalize();
Vector3D My = Vector3D.CrossProduct(Mz, Mx);

// 计算平移矩阵的 dx、dy 和 dz
double dx = -Vector3D.DotProduct(Position, Mx);
double dy = -Vector3D.DotProduct(Position, My);
double dz = -Vector3D.DotProduct(Position, Mz);

// 构建观察矩阵
Matrix3D viewmatrix = new Matrix3D(Mx.X, My.X, Mz.X, 0,
                                    Mx.Y, My.Y, Mz.Y, 0,
                                    Mx.Z, My.Z, Mz.Z, 0,
                                    dx, dy, dz, 1);
cam.ViewMatrix = viewmatrix;

// 计算正射投影矩阵的 Sx、Sy、Sz 和 dz
double Sx = 2 / Width;
double Sy = Sx * viewport.Width / viewport.Height;
double Sz = 1/(zNear-zFar);
double Dz = zNear / (zNear - zFar);

Matrix3D projectionmatrix = new Matrix3D(Sx, 0, 0, 0,
                                         0, Sy, 0, 0,
                                         0, 0, Sz, 0,
                                         0, 0, Dz, 1);
cam.ProjectionMatrix = projectionmatrix;

```

代码 17-20 为 MatrixCamera 指定观察矩阵和投影矩阵

17.4 基本几何体

在 WPF2D 图形中提供了直线、矩形和椭圆等这样的一些基本图元，WPF3D 中所有的图元均由若干三角单元构成。本节我们会通过这些三角形来构成立方体和球这样的基本几何体。

WPF3D 最为遗憾的是没有提供直线这样基本的图元，因此绘制一个 3D 的坐标系也非常困难。好在 WPF3D 团队在开源社区 Codeplex 中上有一个称为“3DTools”的开源项目，为在 WPF3D 中开发更多新的特性提供了一系列有用的类，其中包括一个 ScreenSpaceLines3D 直线类。3DTools 可以在 <http://www.codeplex.com/3DTools> 下载，也可以在本书的随附光盘中找到。

17.4.1 使用直线 ScreenSpaceLines3D

如果使用直线 ScreenSpaceLines3D 类，需要将下载到的 3DTools.dll 添加到当前工程的引用中，如图 17-34 所示。



图 17-34 添加 3DTools.dll 到当前工程的引用中

如果在 XAML 文件中使用 ScreenSpaceLines3D 类，需要添加如代码 17-21 所示的声明。

```
<Page Background="Black"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:tool3d="clr-namespace:_3DTools;assembly=3DTools">
    .....
    <tool3d:ScreenSpaceLines3D Points="...."
```

代码 17-21 添加声明

在代码文件中使用 ScreenSpaceLines3D 类并引用其他程序集中的类，如代码 17-22 所示。

```
using _3DTools;
....
ScreenSpaceLines3D line3D = new ScreenSpaceLines3D();
....
```

代码 17-22 在代码文件中使用 ScreenSpaceLines3D 类

ScreenSpaceLines3D 类能在 3D 空间绘制直线，这些直线的宽度相同，一般对应于 2D 世界的一个设备独立单位。如果用一个极细的圆柱在 3D 空间里绘制直线，由于受到透视投影的影响，所以势必离照相机越远，其宽度就会变得越小，如图 17-35 所示。但是 ScreenSpaceLines3D 类通过得到照相机的变换，使 3D 直线保持在适当的宽度内，即这些直线在 3D 空间中看上去的宽度一致。

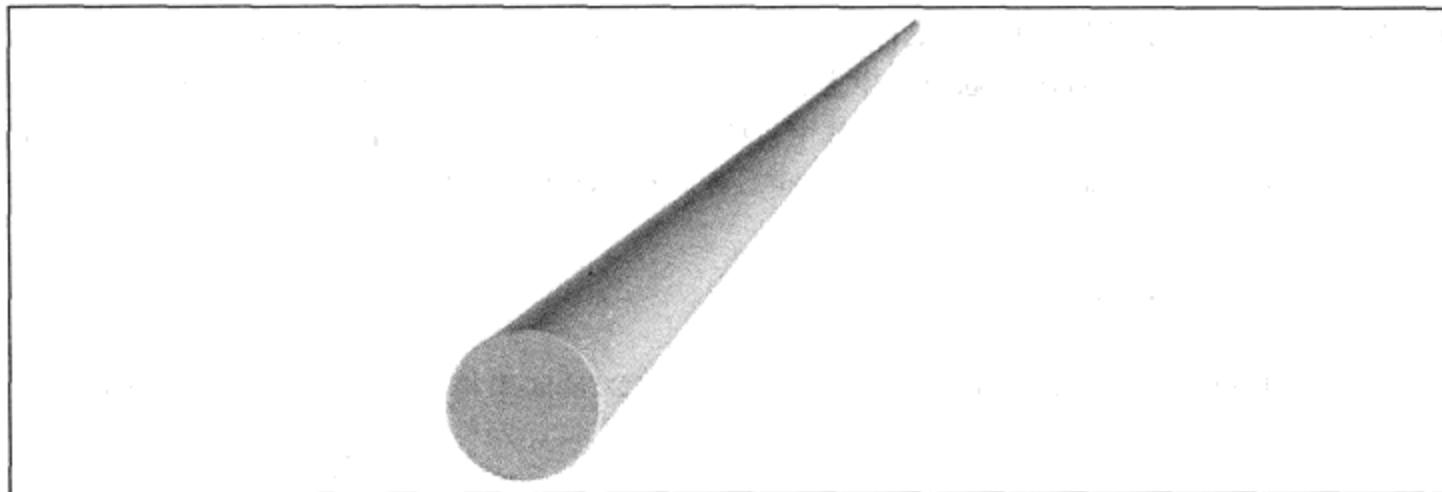


图 17-35 图形在 3D 空间中受透视投影的影响

ScreenSpaceLines3D 和普通直线一样有线宽（Thickness）、颜色（Color）和点的集合（Points）属性，它可以通过设置 Points 来绘制多条直线。注意它绘制直线的点都是成对的，如需要绘制一条从 P0 到 P1，再到 P2 的直线，不能写成{P0,P1,P2}集合，而需要写成{P0,P1,P1,P2}集合。

代码 17-23 通过 ScreenSpaceLines3D 绘制一个 3D 坐标系。

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:tool3d="clr-namespace:_3DTools;assembly=3DTools"
    Title="Cube" Height="300" Width="300">
    <Grid Margin="5">
        <Border BorderBrush="Gray" BorderThickness="1">
            <Viewport3D Name="myViewport">
                <Viewport3D.Camera>
                    <PerspectiveCamera Position="4,5,6" LookDirection="-4,-5,-6"
UpDirection="0,1,0"/>
                </Viewport3D.Camera>
                <ModelUIElement3D>
                    <DirectionalLight Color="White" Direction="-1,-1,-1" />
                </ModelUIElement3D>

                    <!--添加坐标系轴 -->
                    <tool3d:ScreenSpaceLines3D Points="-4,0,0 3,0,0" Color="Red"
Thickness="2"/>
                    <tool3d:ScreenSpaceLines3D Points="0,-5,0 0,3,0" Color="Green"
Thickness="3" />
                    <tool3d:ScreenSpaceLines3D Points="0,0,-10 0,0,3" Color="Blue"
Thickness="4"/>
                </Viewport3D>
            </Border>
        </Grid>
    </Page>
```

代码 17-23 通过 ScreenSpaceLines3D 绘制一个 3D 坐标系

运行结果如图 17-36 所示。

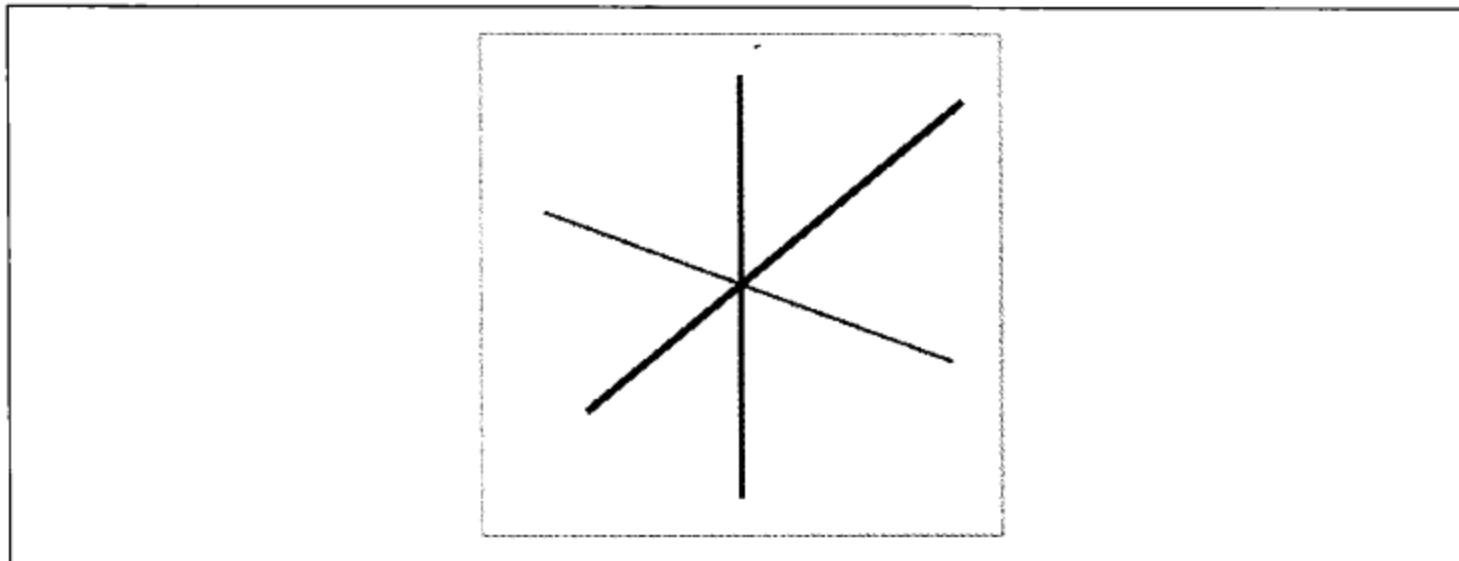


图 17-36 三维坐标系

17.4.2 构建立方体

构建一个立方体比较简单，假定一个中心点在原点且单位长度为 1 的立方体，则其 8 个顶点分别为 $(-0.5, 0.5, 0.5)$ 、 $(0.5, 0.5, 0.5)$ 、 $(0.5, 0.5, -0.5)$ 、 $(-0.5, 0.5, -0.5)$ 、 $(-0.5, -0.5, 0.5)$ 、 $(0.5, -0.5, 0.5)$ 、 $(0.5, -0.5, -0.5)$ 和 $(-0.5, -0.5, -0.5)$ 。

如果立方体的长度为 a ，那么这 8 个顶点的坐标值还要乘上相应的系数 a 。如果中心点不在原点，而在 $P_0(x_0, y_0, z_0)$ ，那么原来的点坐标也可以看成从原点指向该点的向量，因此新的点坐标为中心点和向量的和。一个立方体中心点在 (x_0, y_0, z_0) 且单位长度为 a ，则 8 个顶点分别为 $(-0.5a, 0.5a, 0.5a) + P_0$ 、 $(0.5a, 0.5a, 0.5a) + P_0$ 、 $(0.5a, -0.5a, -0.5a) + P_0$ 、 $(-0.5a, 0.5a, -0.5a) + P_0$ 、 $(-0.5a, -0.5a, 0.5a) + P_0$ 、 $(0.5a, -0.5a, 0.5a) + P_0$ 、 $(0.5a, -0.5a, -0.5a) + P_0$ 和 $(-0.5a, -0.5a, -0.5a) + P_0$ ，如图 17-37 所示。

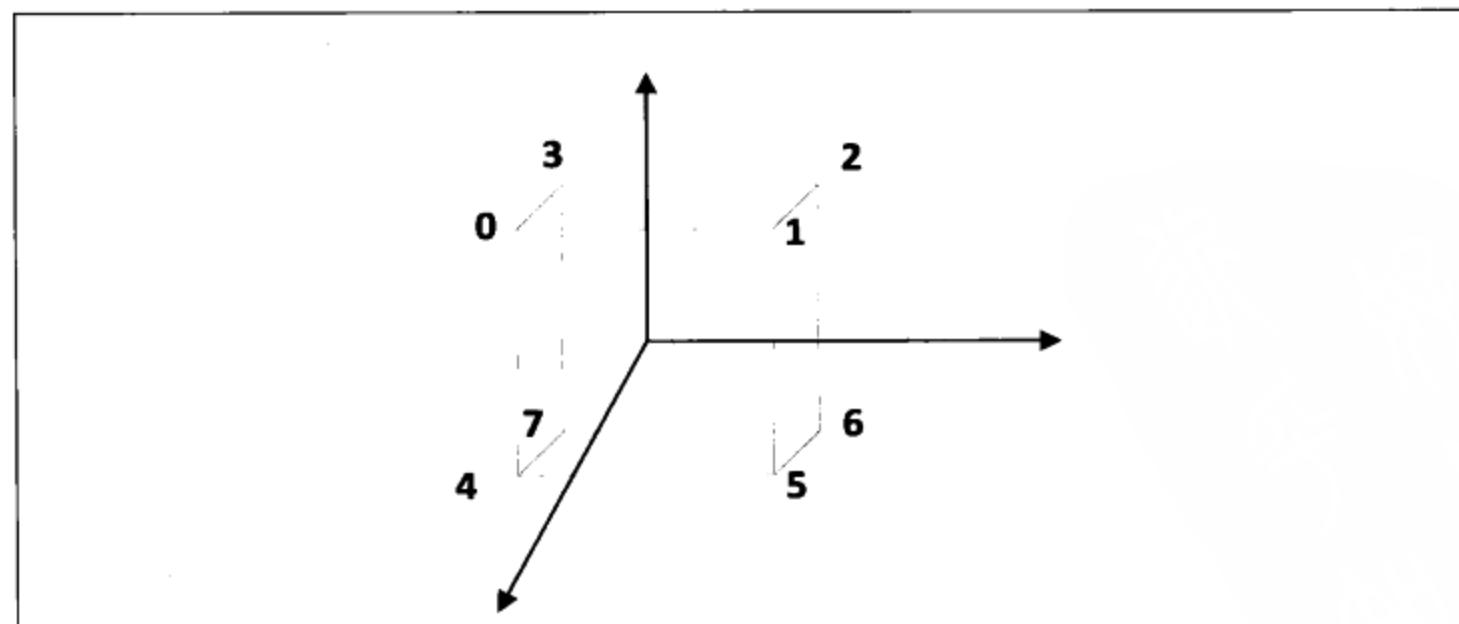


图 17-37 构建立方体

代码 17-24 构建一个立方体。

```
public MeshGeometry3D CreateCube(Point3D center, double side, ref ScreenSpaceLines3D line3D)
{
    // 如果 line3D 非空，则需要绘制构建的三角网
    bool isWire = false;
    if (line3D != null)
        isWire = true;

    MeshGeometry3D mesh = new MeshGeometry3D();
    double a = side / 2.0;
    Point3D[] p = new Point3D[8];
    p[0] = new Point3D(-a, a, a);
    p[1] = new Point3D(a, a, a);
    p[2] = new Point3D(a, a, -a);
    p[3] = new Point3D(-a, a, -a);
    p[4] = new Point3D(-a, -a, a);
    p[5] = new Point3D(a, -a, a);
    p[6] = new Point3D(a, -a, -a);
    p[7] = new Point3D(-a, -a, -a);

    // 添加顶点
    for (int i = 0; i < 8; i++)
    {
        p[i] += (Vector3D)center;
        mesh.Positions.Add(p[i]);
    }

    // 设置点的顺序
    CreateTriangleIndices(ref mesh, 0, 1, 3);
    CreateTriangleIndices(ref mesh, 1, 2, 3);

    CreateTriangleIndices(ref mesh, 4, 7, 0);
    CreateTriangleIndices(ref mesh, 3, 0, 7);

    CreateTriangleIndices(ref mesh, 4, 7, 5);
    CreateTriangleIndices(ref mesh, 5, 7, 6);

    CreateTriangleIndices(ref mesh, 1, 5, 6);
    CreateTriangleIndices(ref mesh, 2, 1, 6);

    CreateTriangleIndices(ref mesh, 0, 4, 1);
    CreateTriangleIndices(ref mesh, 4, 5, 1);

    CreateTriangleIndices(ref mesh, 3, 2, 7);
    CreateTriangleIndices(ref mesh, 2, 6, 7);

    // 如果需要构建三角网，使用 ScreenSpaceLines3D 构建三角网
    if (isWire)
    {
        CreateScreenlines(ref line3D, p[0], p[1], p[3]);
        CreateScreenlines(ref line3D, p[1], p[2], p[3]);

        CreateScreenlines(ref line3D, p[4], p[7], p[0]);
        CreateScreenlines(ref line3D, p[3], p[0], p[7]);

        CreateScreenlines(ref line3D, p[4], p[7], p[5]);
        CreateScreenlines(ref line3D, p[5], p[7], p[6]);
    }
}
```

```

        CreateScreenlines(ref line3D, p[1], p[5], p[6]);
        CreateScreenlines(ref line3D, p[2], p[1], p[6]);

        CreateScreenlines(ref line3D, p[0], p[4], p[1]);
        CreateScreenlines(ref line3D, p[4], p[5], p[1]);

        CreateScreenlines(ref line3D, p[3], p[2], p[7]);
        CreateScreenlines(ref line3D, p[2], p[6], p[7]);
    }
    return mesh;
}

```

代码 17-24 来构建一个立方体

`center` 表示立方体的中心原点, `side` 代表立方体的长度, `line3D` 是输出参数。如果 `line3D` 不为空, 则返回三角网, 我们可以方便地看出立方体如何通过三角网构建; 如果 `line3D` 为空, 则不返回。

程序运行结果如图 17-38 所示。

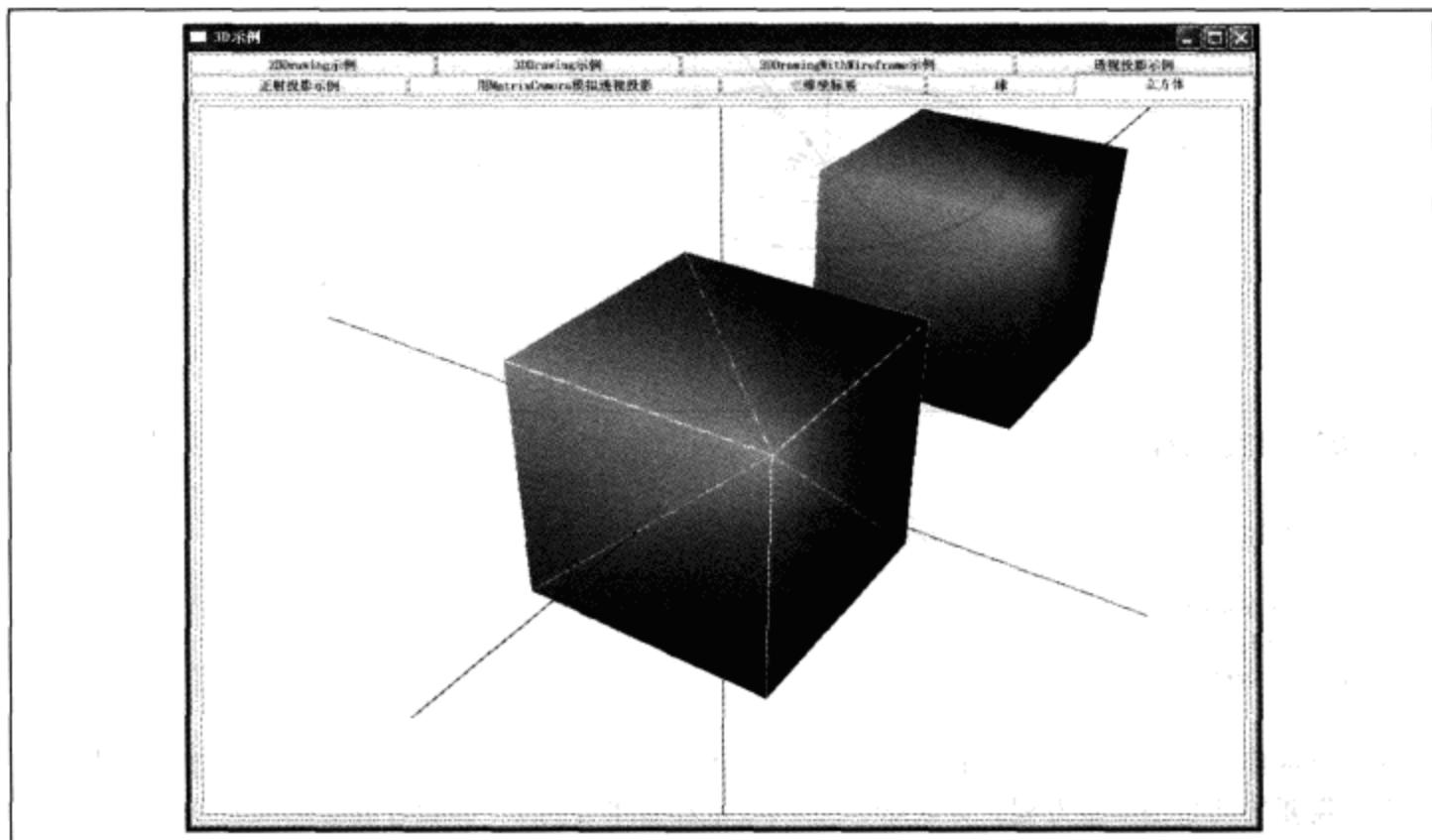


图 17-38 立方体

17.4.3 构建球面

1. 推导球面参数方程

如果一个球心位于坐标原点且半径为 1, 则球面方程为:

$$x^2 + y^2 + z^2 = 1$$

任意满足上述方程的点(x,y,z)都是该球面上的一点。如果球心不在原点，而在(x₀,y₀,z₀)且半径为 r，则球面方程为：

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

尽管通过这样的方程我们可以很容易判断某个点是否在球面上，但是无法生成球面上的一个点。我们想象这个球是地球，地球上的每一个点习惯用经度 θ 和纬度 φ 来表示。出于简化的目的，我们假定球心位于原点，球半径为 1 个单位。北极的坐标设为(0,1,0)，南极的坐标设为(0,-1,0)，赤道与 XZ 平面重合。 θ 的范围是 0~360，假定 $\theta=0$ 是球的正后方，即 Z 轴的负方向。经线的经度递增方向为自西向东， φ 的取值范围为 -90°（对应南极极点）~90°（对应北极极点），如图 17-39 所示。

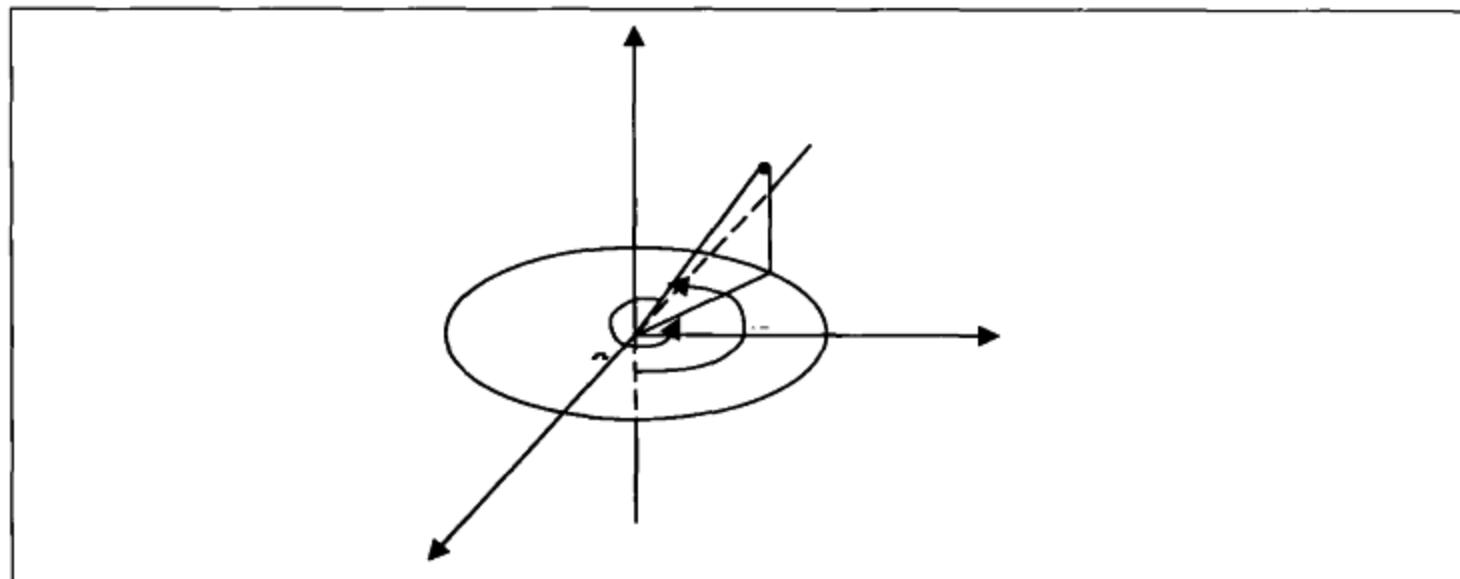


图 17-39 大地坐标系

任意一点的(x,y,z)可以用经度 θ 和纬度 φ 来表示：

$$\begin{cases} x(\theta, \varphi) = r \cos \varphi \sin \theta \\ y(\theta, \varphi) = r \sin \varphi \\ z(\theta, \varphi) = r \cos \theta \end{cases}$$

这样的方程称为“参数方程”，它通过一个或多个参数的变量表达一系列表示直线或曲面的点。我们真正需要的其实是参数方程。如果球心位于(x₀,y₀,z₀)且半径为 r，参数方程会略微复杂一些：

$$\begin{cases} x(\theta, \varphi) = x_0 + r \cos \varphi \sin \theta \\ y(\theta, \varphi) = y_0 + r \sin \varphi \\ z(\theta, \varphi) = z_0 + r \cos \theta \end{cases}$$

2. 剖分球面

剖分球面，也许我们第一个想法是用均匀的三角形来剖分。这的确是一个非常不错的想法，但是在数学上行不通。比较实际的做法是使用经纬线将球面划分为近似正方形的网格。然后将每个正方形划分为两个三角形，如图 17-40 所示。

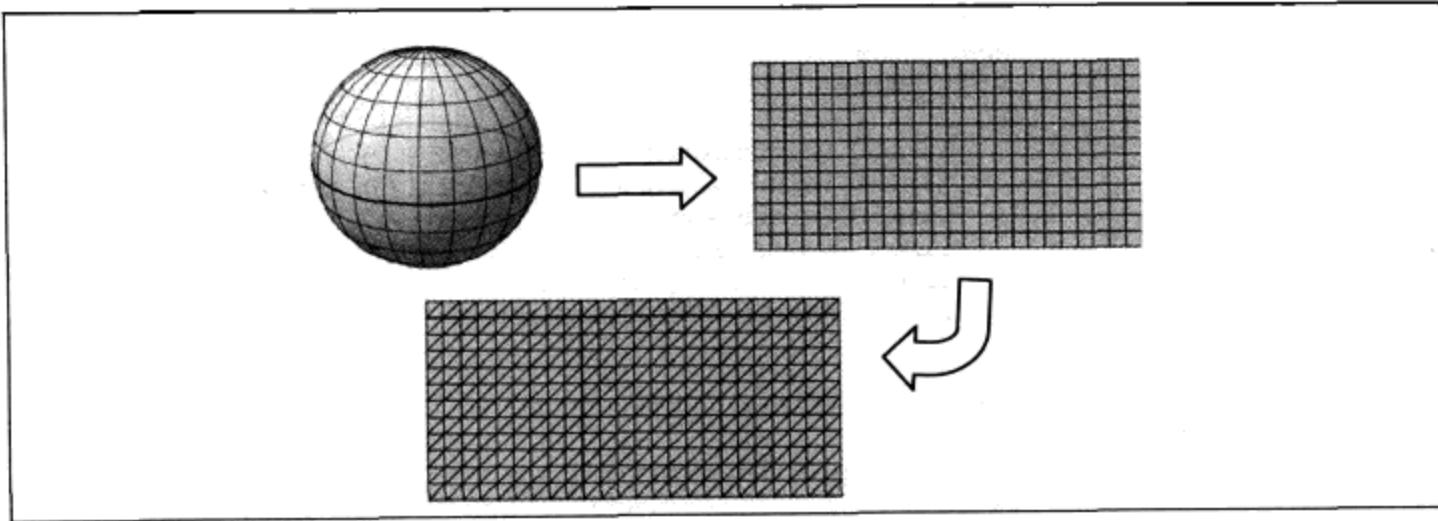


图 17-40 球面的剖分

我们可以为球体创建这样一个函数，如代码 17-25 所示。

```

MeshGeometry3D GenerateSphere(Point3D center, double radius,
                               int slices, int stacks, ref ScreenSpaceLines3D line3D)
{
    // 如果 line3D 非空，则需要绘制构建的三角网
    bool isWire = false;
    if (line3D != null)
        isWire = true;

    Point3DCollection points = new Point3DCollection();
    MeshGeometry3D mesh = new MeshGeometry3D();
    // 计算出来的点，然后添加到 MeshGeometry3D 的 Positions 属性中
    for (int stack = 0; stack <= stacks; stack++)
    {
        double phi = Math.PI / 2 - stack * Math.PI / stacks;
        double y = radius * Math.Sin(phi);
        double scale = -radius * Math.Cos(phi);

        for (int slice = 0; slice <= slices; slice++)
        {
            double theta = slice * 2 * Math.PI / slices;
            double x = scale * Math.Sin(theta);
            double z = scale * Math.Cos(theta);

            Vector3D normal = new Vector3D(x, y, z);
            mesh.Normals.Add(normal);
            mesh.Positions.Add(normal + center);

            Point3D point3D = normal + center;
            points.Add(point3D);
        }
    }

    // 将点的顺序加入到 MeshGeometry3D 的 TriangleIndices 属性
    for (int stack = 0; stack < stacks; stack++)
        for (int slice = 0; slice < slices; slice++)
    {
        int n = slices + 1;

        if (stack != 0)
        {
            mesh.TriangleIndices.Add((stack + 0) * n + slice);
        }
    }
}

```

```

        mesh.TriangleIndices.Add((stack + 1) * n + slice);
        mesh.TriangleIndices.Add((stack + 0) * n + slice + 1);
        if (isWire)
        {
            line3D.Points.Add(points[(stack + 0) * n + slice]);
            line3D.Points.Add(points[(stack + 1) * n + slice]);
            line3D.Points.Add(points[(stack + 1) * n + slice]);
            line3D.Points.Add(points[(stack + 0) * n + slice + 1]);
            line3D.Points.Add(points[(stack + 0) * n + slice + 1]);
            line3D.Points.Add(points[(stack + 0) * n + slice]);
        }
    }
    if (stack != stacks - 1)
    {
        mesh.TriangleIndices.Add((stack + 0) * n + slice + 1);
        mesh.TriangleIndices.Add((stack + 1) * n + slice);
        mesh.TriangleIndices.Add((stack + 1) * n + slice + 1);
        if (isWire)
        {
            line3D.Points.Add(points[(stack + 0) * n + slice + 1]);
            line3D.Points.Add(points[(stack + 1) * n + slice]);
            line3D.Points.Add(points[(stack + 1) * n + slice]);
            line3D.Points.Add(points[(stack + 1) * n + slice + 1]);
            line3D.Points.Add(points[(stack + 1) * n + slice + 1]);
            line3D.Points.Add(points[(stack + 0) * n + slice + 1]);
        }
    }
}
return mesh;
}

```

代码 17-25 为球体创建一个函数

center 表示球的中心原点，radius 代表球的半径，slices 表示经度的剖分数目，stacks 表示纬度剖分的数目。line3D 是个输出参数，如果不为空，则返回三角网，我们可以方便地看出球体是如何通过三角网构建出来的；否则不返回。

其中的关键是两个循环，第 1 个循环将前面参数方程计算的点，添加到 MeshGeometry3D 的 Positions 属性中。添加顺序是纬度 90（北极）~−90（南极），经度 0~360，如图 17-41 所示。

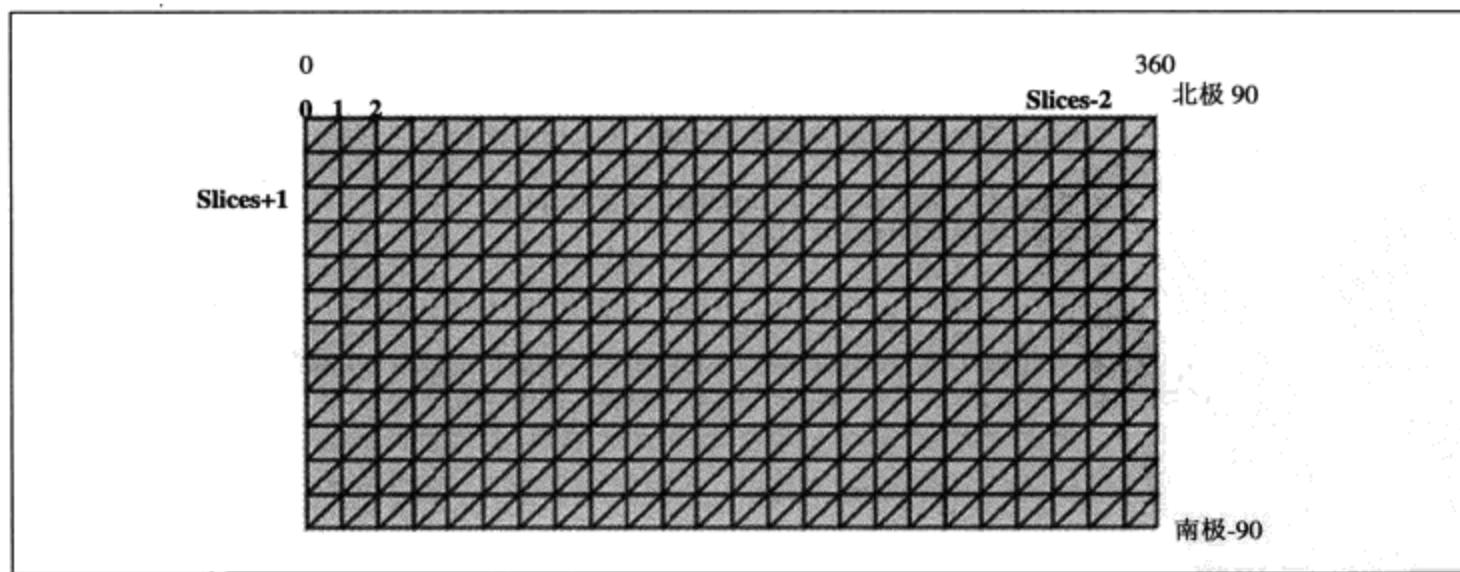


图 17-41 关键的两个循环

第 2 个循环是描述三角网顶点的顺序，在北极和南极的处理要稍稍特殊一些。有一半三角形实际上只是一条线，因此北极与南极和其他纬度不同。仅需要描述一半三角形的顶点，如图 17-42 所示。

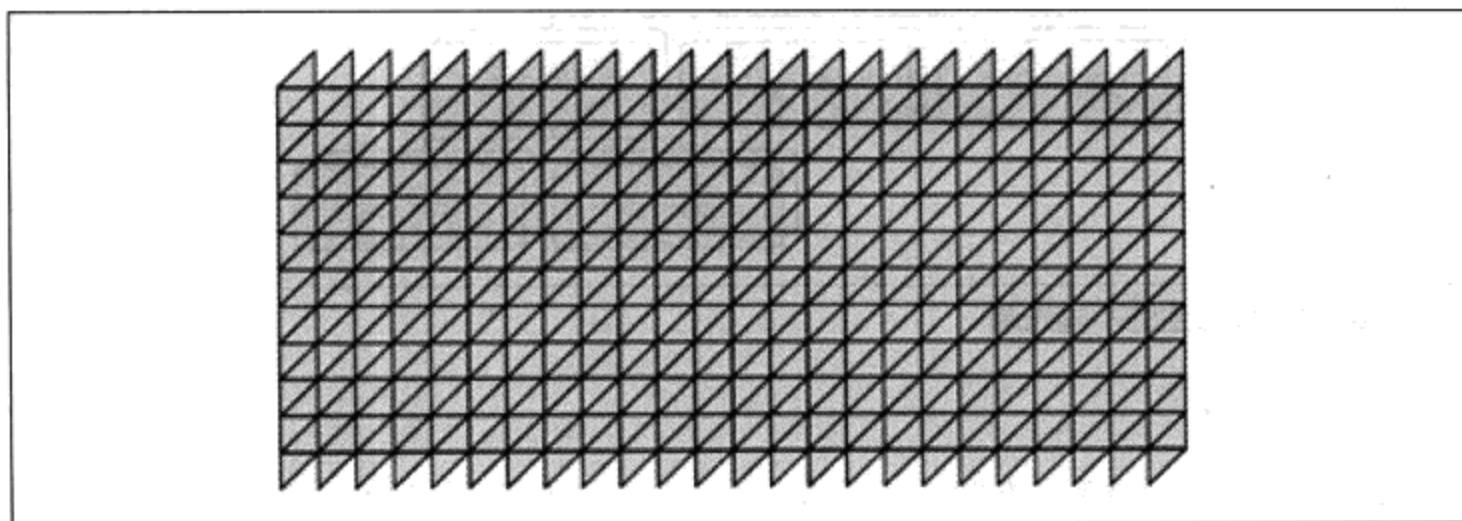


图 17-42 第 2 个循环南极与北极的特殊描述

程序运行结果如图 17-43 所示。

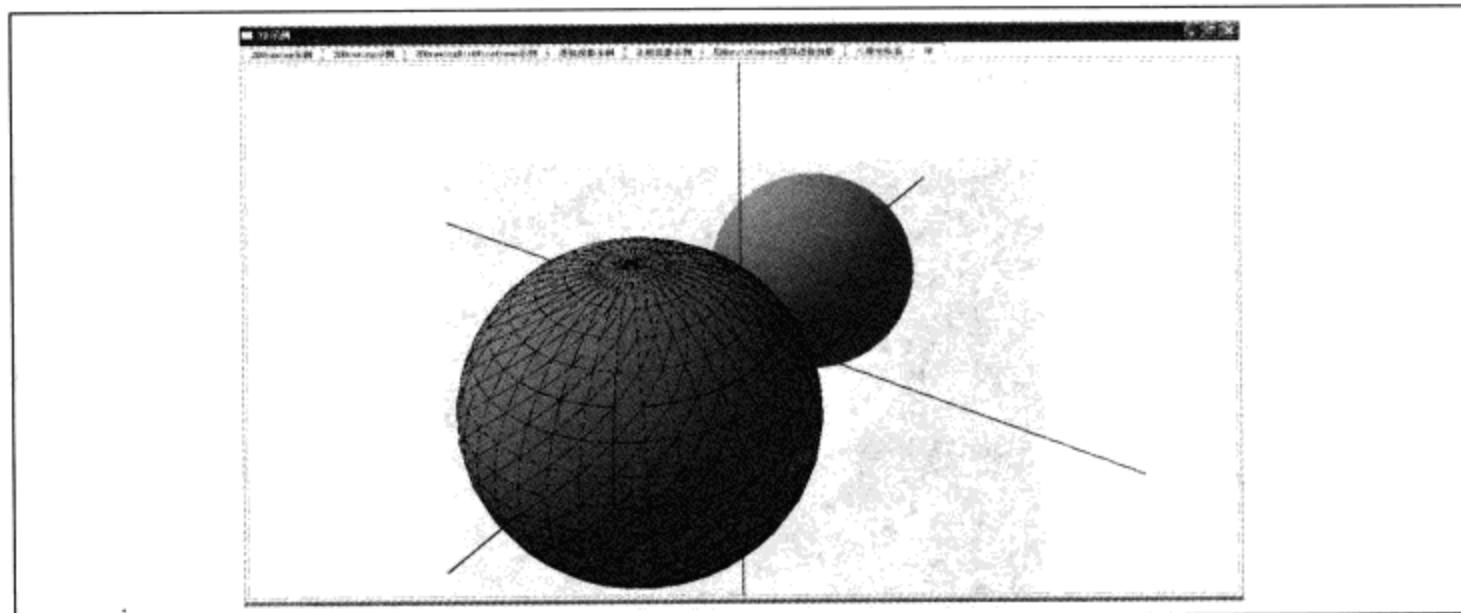


图 17-43 球面

17.5 光源和材质

前面绘制的基本几何体都忽视了 3D 中两个重要细节，即光源和材质。如果没有光，所有的物体黯淡无比。即使有了光，光照射到不同材质的物体上，吸收和反射的光不同，因此光源和材质的配合构成了色彩缤纷的世界。

17.5.1 光源

在 WPF 中用 Light 代表现实世界中的光源，Light 派生了多种不同的光源。如图 17-44 所示。

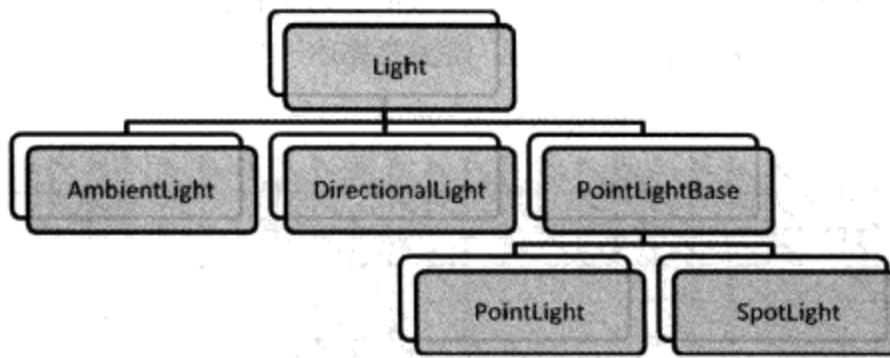


图 17-44 Light 派生的不同光源

(1) AmbientLight

AmbientLight 代表环境光源，模拟的是一个多云天气的户外光照情况，物体表面上任意一点被照射的强度相同。它的唯一属性是 Color，用于控制光源的颜色和光强。如#FFFFFF 是完全强度的白光，而#808080 是半强度的白光。而且光照效果可以叠加，两个半强度的光照产生的效果和一个完全强度的光照相同。

图 17-45 所示为 AmbientLight 照射一个模型——D3D 中的一个非常有名的壶，由于表面反射的光强相同，因此产生了相当平坦的图像，难以看出立体感。



图 17-45 用 AmbientLight 照射的壶

(2) DirectionalLight

DirectionalLight 代表有方向的光源，模拟远距离的光源，如太阳光从无限远的源点的平行光照射到物体上。它的 Color 属性控制光源的颜色和强度 Direction 控制光照方向。光源设置的示例如代码 17-26 所示。

```
<DirectionalLight Color="White" Direction="1,-1,-0.5"/>
```

代码 17-26 设置光源

光照从左上前方照射过来，效果如图 17-46 所示。



图 17-46 用 DirectionalLight 照射壺的效果

(3) PointLight

PointLight 代表一个点光源，模拟一个房间里的灯泡。PointLight 对象在 3D 空间中有具体的位置，从该点位置向各个方向发射均匀光线，光线的强度随着与发射点距离的增加而减弱。它的 Color 属性控制颜色和强度，Position 属性指定光源的位置，如代码 17-27 所示。

```
<PointLight Color="White" Position="4.5 4.5 4" />
```

代码 17-27 PointLight 的属性

为了使 PointLight 和 SpotLight 区别明显，直接使用一个在 XY 平面上的一个矩形。该矩形的左下角点为坐标原点，宽度为 9，如图 17-47 所示。

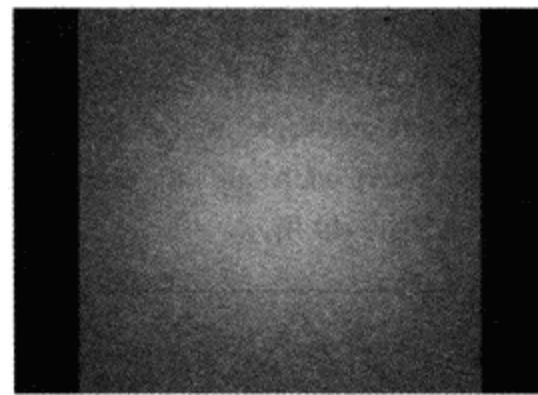


图 17-47 用 PointLight 照射的矩形

PointLight 的光强随着距离增强而衰减，衰减公式如下：

$$\text{衰减率} = \frac{1}{\max(1, C + Ld + Qd^2)}$$

d 是光源与被照射点的距离， C 、 L 和 D 分别由 ConstantAttenuation、LinearAttenuation 和 QuadraticAttenuation 属性决定，它们是从 PointLightBase 继承而来。如果 $C=1$ 、 $L=0$ 且 $Q=0$ ，则可以得到与距离无关且等强度的 PointLight。如果设定 PointLightBase 还有一个 Range 属性，则在该范围以外 PointLight 无效。

(4) SpotLight

SpotLight 代表一个圆锥光源，模拟一个手电筒。和 PointLight 一样在 3D 空间中有具体的位制，从该点发射锥形光线。

如图 17-48 所示。

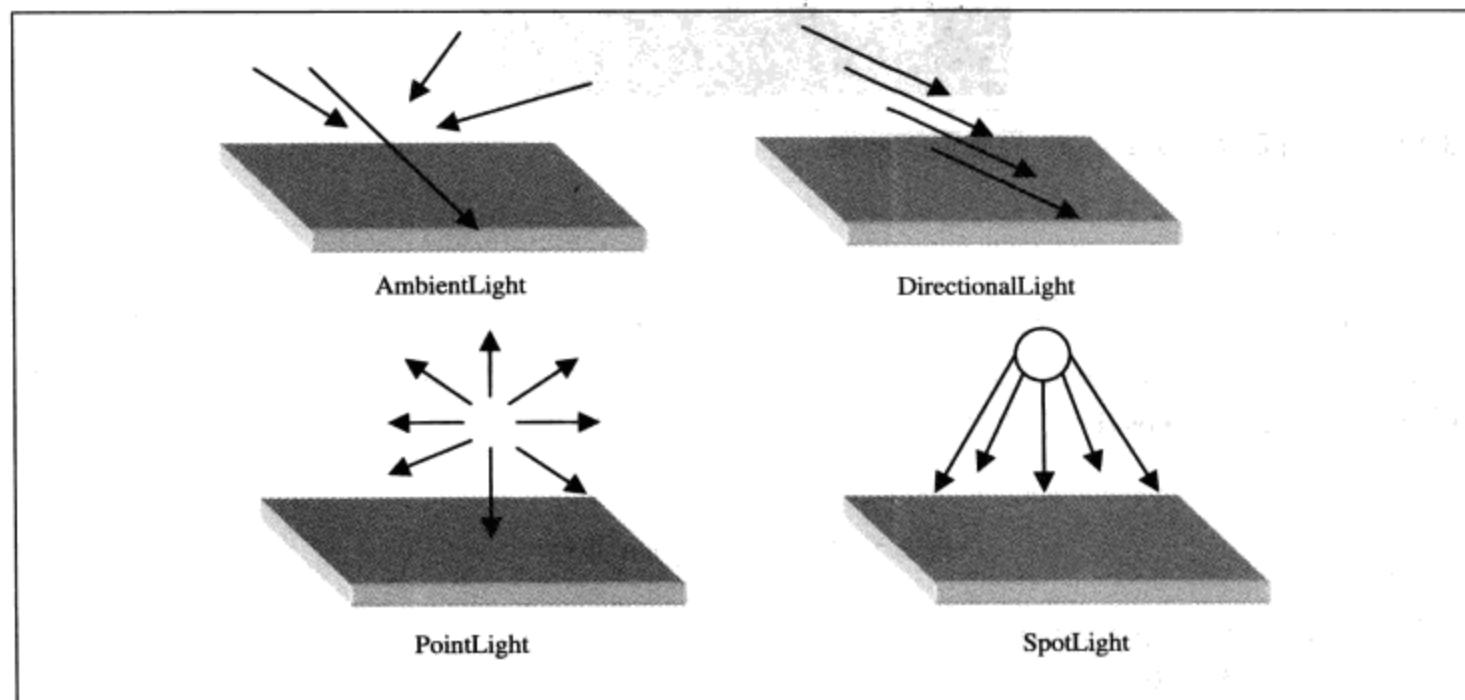


图 17-48 几种不同的光源

SpotLight 用 Color 表示光的颜色和光强，Position 表示光源的方向，Direction 表示光照的方向。除此之外，它还有 InnerConeAngle 和 OuterConeAngle 属性。在 InnerConeAngle 区域中接受由 Color 属性指定的颜色和强度的光照，在 InnerConeAngle 和 OuterConeAngle 之间会逐渐衰减，在 OuterConeAngle 之外则完全不可见，如图 17-49 所示。

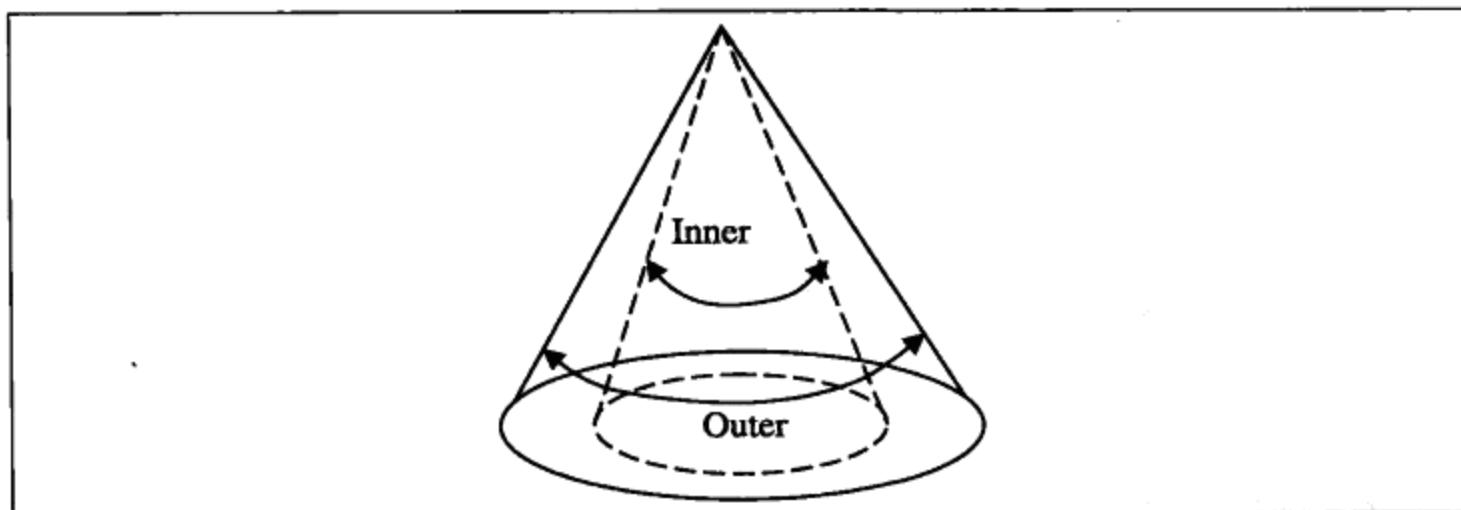


图 17-49 SpotLight 的 InnerConeAngle 和 OuterConeAngle 属性

SpotLight 的设置示例如代码 17-28 所示。

```
<SpotLight Position="4.5 4.5 10" Direction="0 0 -1" InnerConeAngle="30"  
OuterConeAngle="45" />
```

代码 17-28 设置 SpotLight

光照效果如图 17-50 所示。

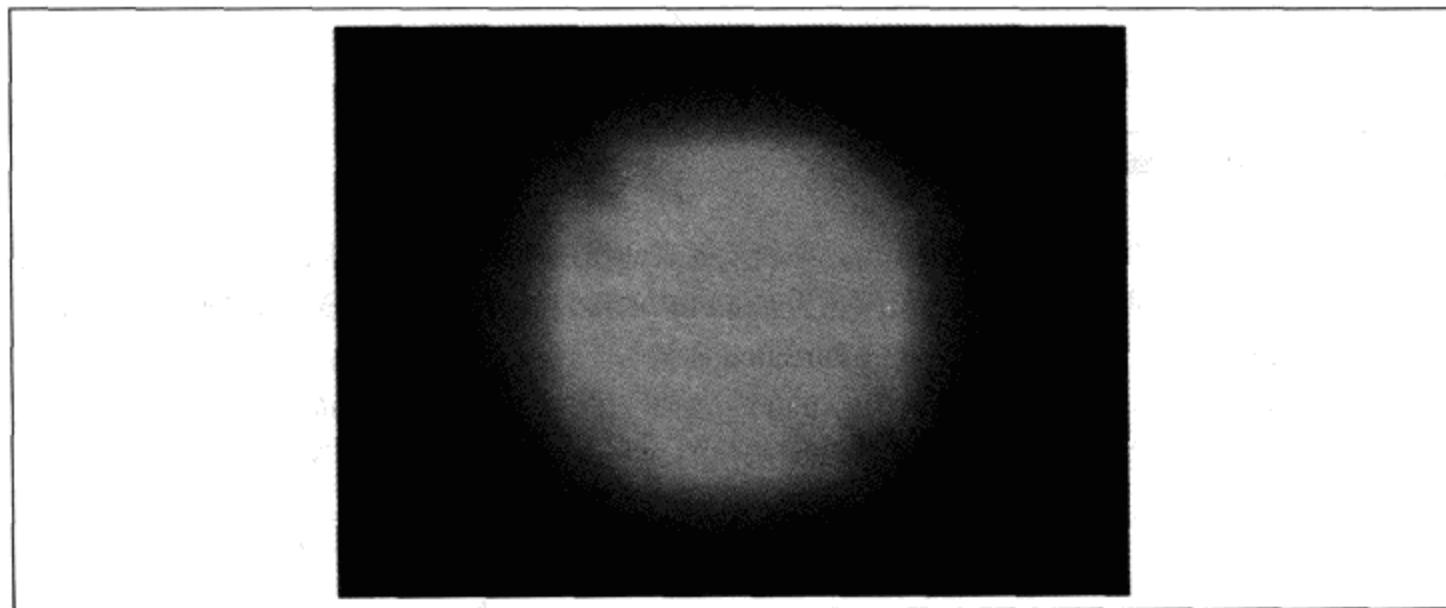


图 17-50 光照效果

(5) 光源组合

Light 对象可通过 Model3DGroup 组合，代码 17-29 用于产生由 25% 的环境光和 75% 的有向光组成的光源。

```
<ModelVisual3D.Content>  
  <Model3DGroup>  
    <AmbientLight Color="#404040"/>  
    <DirectionalLight Color="#C0C0C0" Direction="1,-1,-0.5"/>  
  </Model3DGroup>  
</ModelVisual3D.Content>
```

代码 17-29 Light 对象可通过 Model3DGroup 组合

17.5.2 着色和法线

1. 理论背景

在前面的 4 个例子中，除了环境光以外，其他有向光源照射在物体上都会有不同的明暗效果，表面的明暗由光线和表面之间的夹角决定。为了简化计算，本节中使用的光源都是 DirectionalLight，因为该光源的光线都是平行的。此外物体的材质均使用的 DiffuseMaterial，这种材质可以均匀地反射光线，模拟漫反射的效果，这样表面的明暗度与照相机的位置无关。

在 WPF3D 中为表面着色涉及法线 (Normal) 这样的向量，法向量是垂直于某个平面的向量，图 17-51 所示为一个三角形平面的法向量。

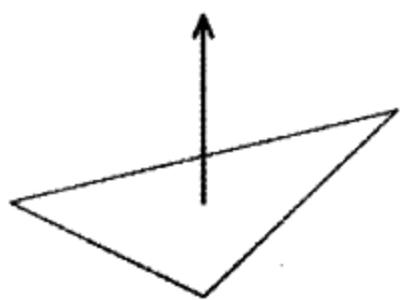


图 17-51 三角形平面的法向量

漫反射表面遵循朗伯余弦定律的明暗模型，即漫反射表面反射光的比例等于表面法向量与光源方向之间的夹角负余弦。假定某个表面使用颜色为#00FFFF 的漫反射材质，照射该表面的唯一光源是一个白色的 DirectionalLight。如果该光源的 Direction 向量与上述表面垂直，则法向量与光源方向之间的夹角的负余弦值为 1，该表面的颜色为#00FFFF；如果光源方向与垂直表面成 150° 角，那么负余弦值为 0.87 ($-\cos 150$)，则表面的颜色为原来颜色的 87%，即#00DEDE，如图 17-52 所示。

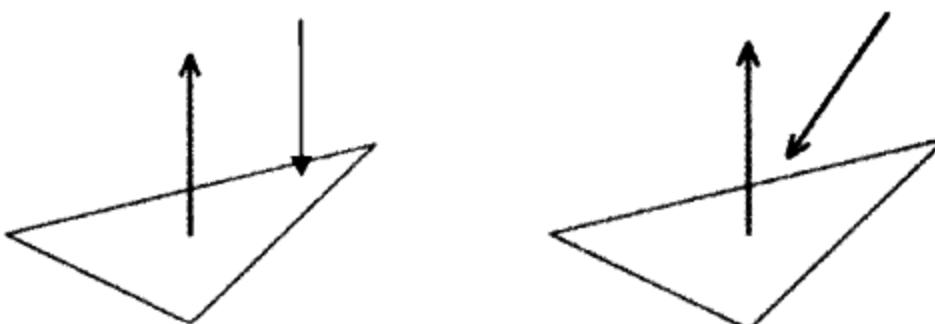


图 17-52 漫反射表面应遵循朗伯余弦定律的明暗模型

这种着色方法有明显缺点，由于三角形是一个平面，所以整个三角形区域都会反射等量的光照。如果两个三角单元成一定角度，之间会出现清晰的边缘。法国计算机科学家 Henri Gouraud 提出的高氏着色可以一定程度上解决这个问题，该算法通过为三角形的不同顶点指定法向量使一个三角形的不同区域具有不同的反射光照，如图 17-53 所示。

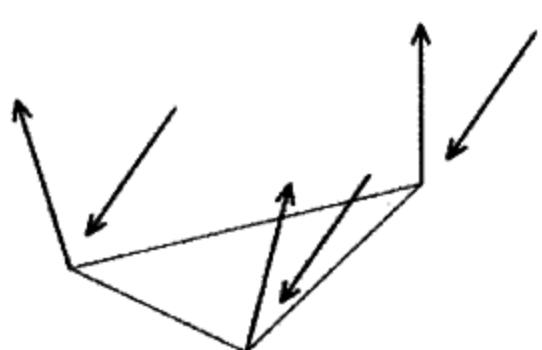


图 17-53 高氏着色算法

3 个顶点上的光照仍然由顶点的法向量和光源方向的负余弦来决定，其他点通过插值计算得到。边

界上的点通过边界上的两个顶点插值，三角形内部的点通过边界上的点来插值。

无论指定表面，还是顶点法向量，我们都需要计算光源方向和法向量之间的夹角。它很容易通过向量之间的运算得到，最简单的方法是通过 `Vector3D` 的 `AngleBetween` 方法获得两向量之间的夹角。但是我们要的并不是夹角，而是其余弦值，因此通过下面的公式直接得到余弦值更为高效：

$$\cos\alpha = \frac{A \cdot B}{|A||B|}$$

2. MeshGeometry3D 的 Normals 属性

在 WPF3D 中，`MeshGeometry3D` 的 `Normals` 属性是顶点法向量。

`Normals` 是一个向量的集合，对应是 `Positions` 属性中每个点的法向量。我们在前面构建球面时已经为球面的每个顶点指定了其法向量，球面的法向量的值实际上就是球心在原点时每个顶点的值，如图 17-54 所示。

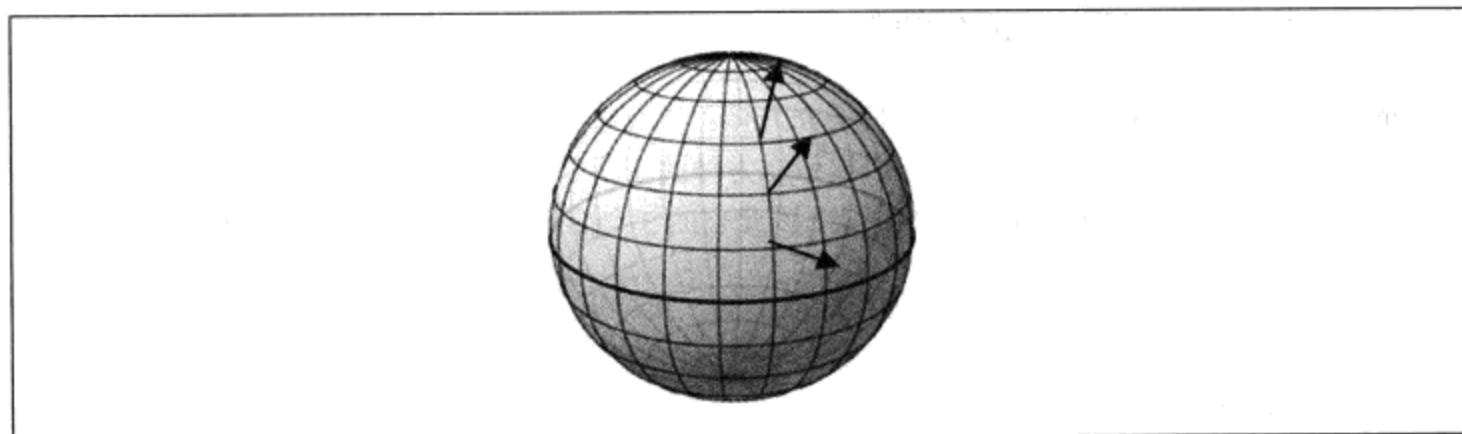


图 17-54 球面的法向量

如果未指定 `Normals` 属性，那么 WPF3D 会自动计算出一组顶点的法向量。它通过所有共享某一公共顶点的三角形单元的法向量进行平均而得出该顶点的法向量，图 17-55 所示矩形的 1 号点通过前面、右侧面和上面 3 个面 6 个三角形的表面法向量决定。

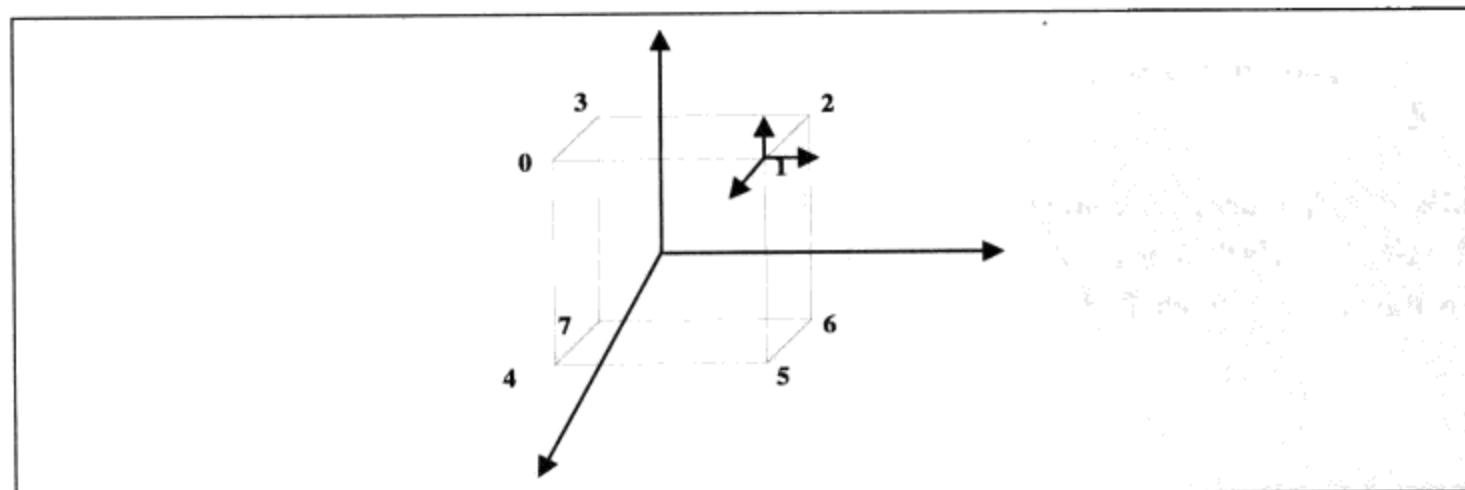


图 17-55 WPF3D 可自动计算出一组顶点的法向量

如果我们不是用 8 个顶点，而是用 36 个点来描述（一个面由两个三角形组成，一个三角形由 3 个顶点组成， $3 \times 2 \times 6 = 36$ 立方体）。尽管多个点的位置重复，但是不存在公共顶点，这时每个顶点的法向量由所在面的法向量决定。由 8 个顶点和 36 个顶点绘制的立方体由于法向量设置不一样，其着色效果明显不同，如图 17-56 所示。

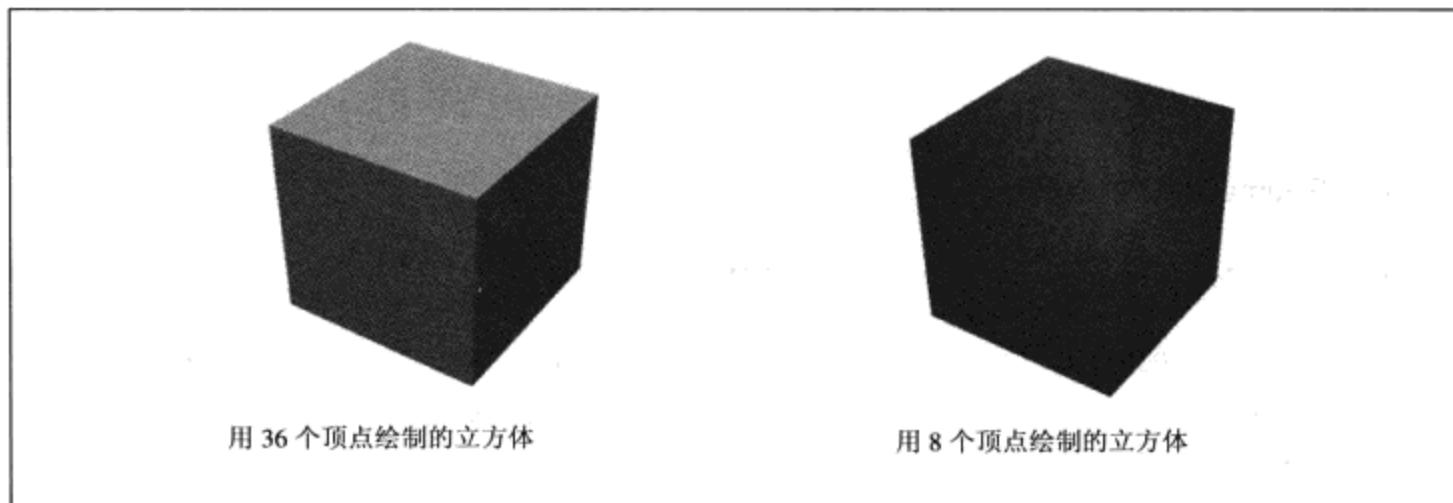


图 17-56 用 36 个顶点和 8 个顶点绘制的立方体

图中的法向量由 WPF3D 自动计算，左侧的立方体由于没有公共顶点且顶点和面的法向量相同，因此立方体每个面的颜色值相同，这样可以显示立方体的棱角；右侧的立方体有公共顶点，每个顶点的法向量由共用的 3 个面的法向量平均决定，因此它的 3 个面的颜色值有一个过渡，棱角并不清晰。

不要简单地认为 36 个顶点就会比 8 个顶点的效率低，实际上我们更多情况下不希望使用公用顶点。一个基本原则是如果希望边缘不被平滑，那么不应该使用共用顶点。为了提供绘制效率，应该显式地设定 Normals 集合。

3. SpotLight 和 PointLight

前面讨论光照强度时我们假定的光源为 DirectionalLight，实际上 SpotLight 和 PointLight 光照的效果和 DirectionalLight 一样由光源的方向和表面的法向量有关，不同的是它们还与三角网格的密集程度有关。如果定义一个由两个三角平面构成的矩形平面，然后在平面的正上方放置一个 PointLight 光源，则整个平面上的照明情况将由入射光与构成矩形的 4 个顶点的法向量之间的夹角决定。如果使用 SpotLight，所有的光源都处在一个圆锥之内，当 4 个顶点都在圆锥之外时则无法观察到任何照明情况。

如果要使 SpotLight 和 PointLight 的照明效果明显，即使平坦的表面也需要更多的小三角单元。三角单元越多，则效果越明显。图 17-57 所示为分别由 1 个矩形、4 个矩形和 81 个矩形组成的平面在 SpotLight 和 PointLight 的光照情况。

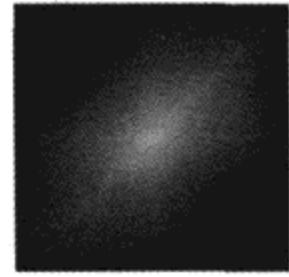
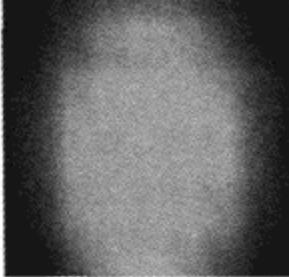
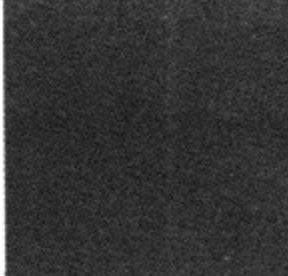
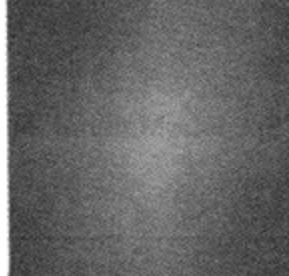
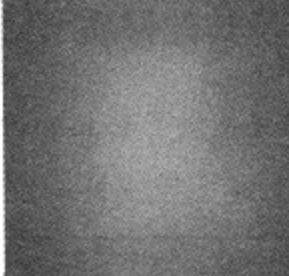
	1 个矩形	4 个矩形	81 个矩形
SpotLight			
PointLight			

图 17-57 平面组成单元个数影响 SpotLight 和 PointLight 的照明效果

17.5.3 计算 DiffuseMaterial 和表面颜色

DiffuseMaterial 材质还有 Color 和 AmbientColor 属性，前者表示从一个有向光的光源反射光的多少；后者表示从一个环境光的光源所反射光的多少。这两个属性类型均为 Color，默认值为 White，表示默认情况下材质会反射所有的光照。

假设有一个 Color 属性为 #4080C0 的 DirectionalLight 光源直射到一个三角形上，该三角形表面材质为 DiffuseMaterial。该对象的 Color 属性取值为 #808080（灰色），SolidColorBrush 属性取值为 #00C0FF。下面我们分别研究物体表面红、绿和蓝 3 种基色（注意这里的颜色值都是 16 进制）。

对于红基色，入射光为 0x40。由于 Color 的红基色为 0x80，即意味着只反射一半的红光（完整的红光是 0xFF）。由于 SolidColorBrush 的颜色值中没有红光，因此最终物体表面的红基色为 0。

对于绿基色，入射光为 0x80。Color 的绿基色也为 0x80，也是反射一半绿光。而 SolidColorBrush 的绿基色为 0xC0，它只允许 3/4 的绿色光通过，因此最终物体表面的绿基色为 0x30。

对于蓝基色，入射光为 0xC0。材质反射一半的蓝色光，而 SolidColorBrush 允许让蓝色光完全通过，因此最终物体表面的蓝基色为 0x60。

最终三角形表面的颜色值为 #003060，如果将 Color 设置为黑色，则表示物体不会反射任何光，这时忽略 Brush 属性。

AmbientColor 和 Color 表明反射的环境光，我们可以给出下面的公式来计算物体的红基色的值，也可以计算绿基色和蓝基色的值：

Color.R =

$$(\sum_{i=0}^n (L_i \cdot \text{Color.R} * \text{DiffuseMaterial.Color.R} * \text{NormalFactor}) + \text{AmbientLight.Color.R} * \text{DiffuseMaterial.Color.R}) * \text{DiffuseMaterial.Brush.Color.R}$$

L_i 代表有向光源，如 DirectionalLight。对于顶点，NormalFactor 代表光线与顶点法向量之间的负余弦；对于物体表面上的任意一点，则需要通过插值计算得到。NormalFactor 介于 0 和 1 之间。注意该公式的颜色值范围不是 0~255，而是 0~1，计算之后可以将该值变换为 0~255。

17.5.4 其他材质

WPF3D 还提供了 EmissiveMaterial、SpecularMaterial 和 MaterialGroup 材质类层次结构，如图 17-58 所示。

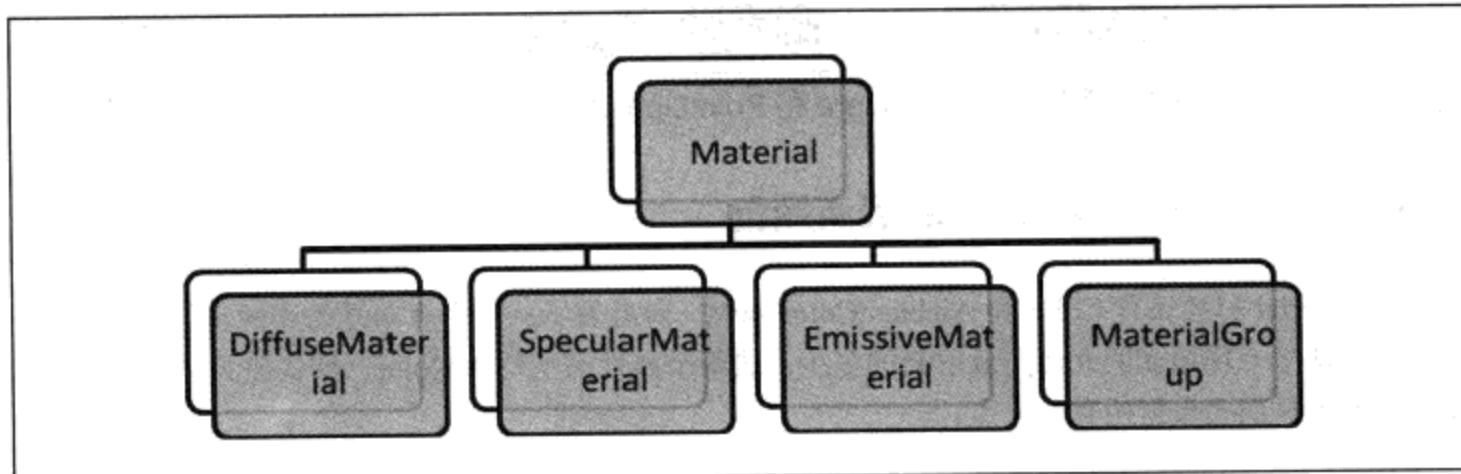


图 17-58 材质的类层次结构

SpecularMaterial 模拟的是一种反射光强的光滑材质，如平坦的金属或塑料等。使用 DiffuseMaterial，无论视角如何变换，物体总是呈现相同的颜色。但是 SpecularMaterial 不同，视角会影响物体表面颜色的变化。光线从 SpecularMaterial 上反弹回来，如被镜子反射一样。只有照相机在光线反射回来的线路上可以观察到最大亮度的图形，如图 17-59 所示。

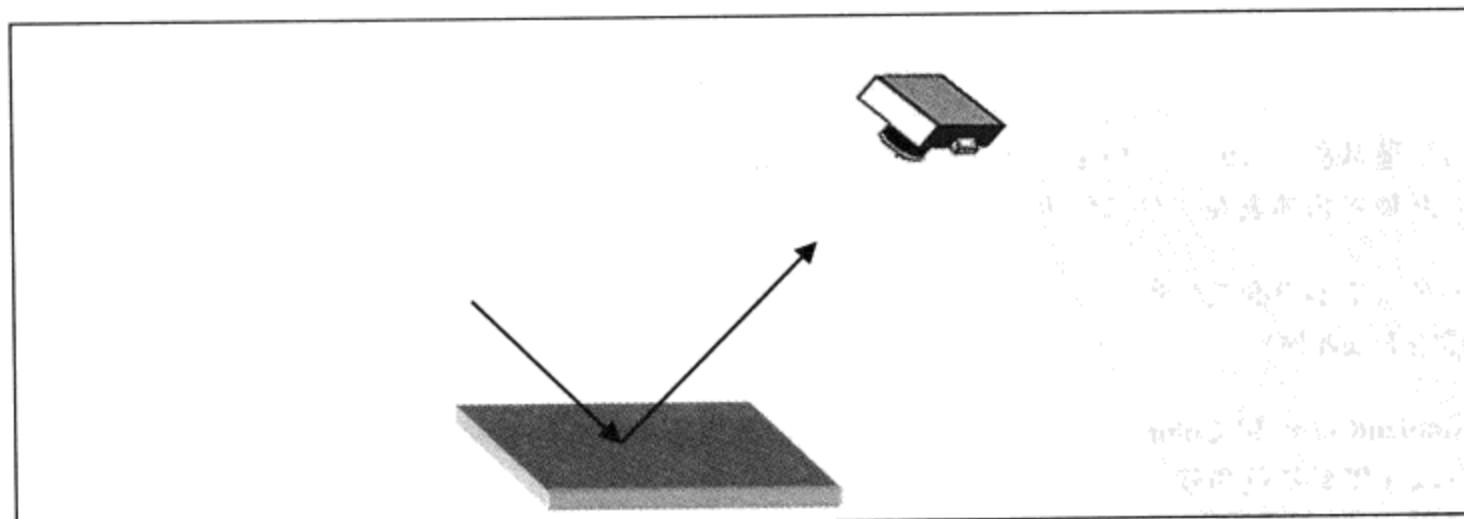


图 17-59 SpecularMaterial 的特性

代码 17-30 生成如图 17-60 所示只有 SpecularMaterial 材质的茶壶。

```
<GeometryModel3D.Material>
  <SpecularMaterial Brush="White" SpecularPower="10" />
</GeometryModel3D.Material>
```

代码 17-30 只有 SpecularMaterial 材质的茶壶

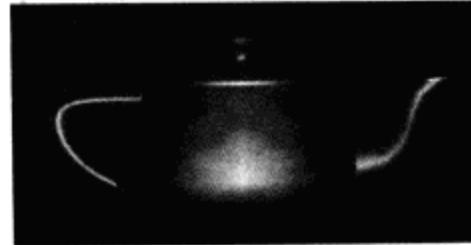


图 17-60 只有 SpecularMaterial 材质的茶壶

EmissiveMaterial 模拟能够发光的表面，如电灯泡或者高温金属，它会向照相机发出可见光，代码 17-31 生成如图 17-61 所示的材质为 EmissiveMaterial 的茶壶。

```
<GeometryModel3D x:Name="Teapot">
  <GeometryModel3D.Material>
    <EmissiveMaterial Brush="Green" />
  </GeometryModel3D.Material>
```

代码 17-31 材质为 EmissiveMaterial 的茶壶

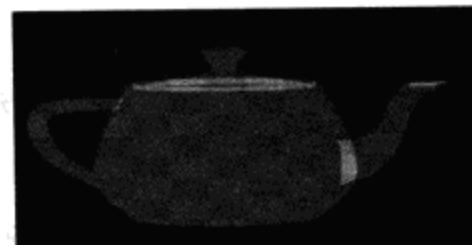


图 17-61 材质为 EmissiveMaterial 的茶壶

上述两种材质通常和 DiffuseMaterial 配合使用，起到增强图形亮度的效果。图 17-62 所示是红色 DiffuseMaterial 和 SpecularMaterial 组合的茶壶，这两种材质的组合可以通过 MaterialGroup 来实现。

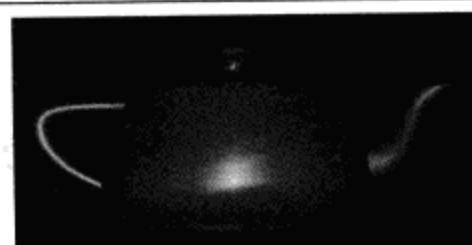


图 17-62 红色 DiffuseMaterial 和 SpecularMaterial 组合的茶壶

如代码 17-32 所示。

```
<GeometryModel3D x:Name="Teapot">
```

```
<GeometryModel3D.Material>
  <MaterialGroup>
    <DiffuseMaterial Brush="Red" />
    <SpecularMaterial Brush="White" SpecularPower="40" />
  </MaterialGroup>
```

代码 17-32 通过 MaterialGroup 来实现两种材质的组合

代码 17-33 生成如图 17-63 所示的白色 DiffuseMaterial 和 EmissiveMaterial 组合的茶壶。

```
<GeometryModel3D x:Name="Teapot">
  <GeometryModel3D.Material>
    <MaterialGroup>
      <DiffuseMaterial Brush="White" />
      <EmissiveMaterial Brush="Green" />
    </MaterialGroup>
```

代码 17-33 两种材质的组合



图 17-63 白色 DiffuseMaterial 和 EmissiveMaterial 组合的茶壶

17.5.5 纹理

前面我们一直用纯色 (SolidColorBrush) 来填充 3D 表面，实际上所有在 WPF2D 图形中的画刷都可以应用在 3D 表面上。即 3D 表面可以是渐变色、图片，甚至视频。

当利用 2D 画刷填充 3D 物体表面时必须建立图形的 3D 顶点与画刷的 2D 相对坐标之间的关系，为此需要 MeshGeometry3D 的第 4 个重要的属性 TextureCoordinates。在 2D 画刷中左上角点是原点坐标，而右下角点坐标为(1,1)，如图 17-64 所示。

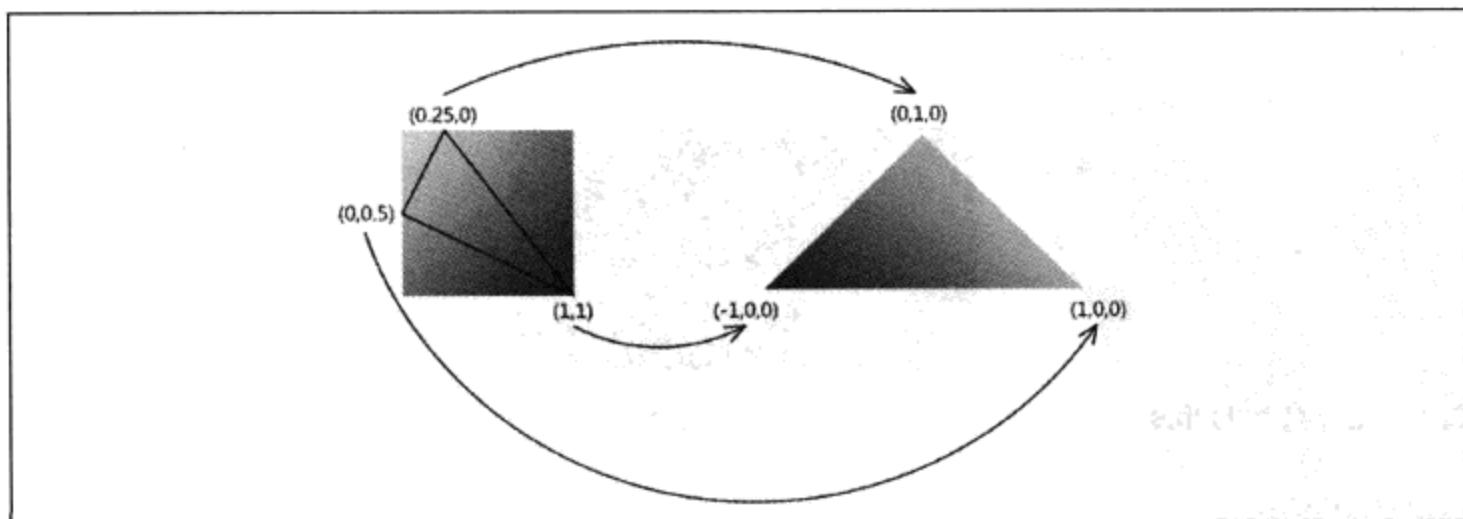


图 17-64 建立图形的 3D 顶点与画刷的 2D 相对坐标之间的关系

如果希望画刷的一部分映射到一个三角单元的表面上，那么该 MeshGeometry3D 应该写成如代码 17-34 形式。

```
<MeshGeometry3D TriangleIndices="0 1 2"  
    Positions="0 1 0, -1 0 0, 1 0 0"  
    TextureCoordinates="0.25 0, 1 1, 0 0.5" />
```

代码 17-34 MeshGeometry3D 代码

TextureCoordinates 是一个 2D 点的集合，用于存放画刷坐标系下的点坐标，点的顺序和 TriangleIndices 的顺序保持一致。

我们可以为前面的球加上一个影像纹理，使其看起来更像一个地球，如图 17-65 所示。如果使用 ImageBrush，那么图片的左上角为(0,0)，右下角为(1,1)。

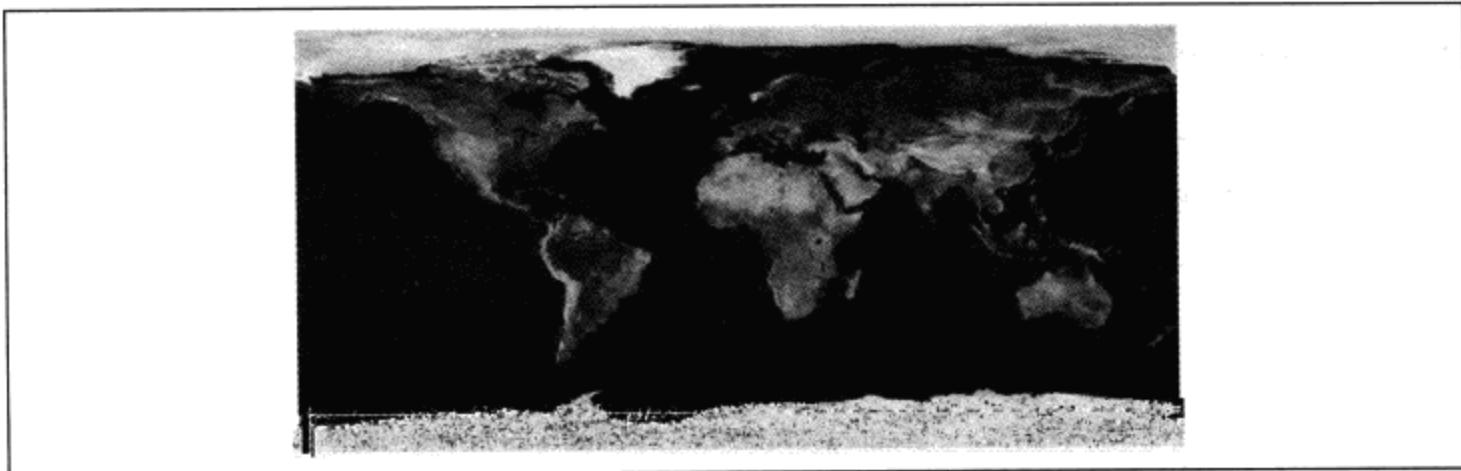


图 17-65 添加的影像纹理

如前所述，球面实际上是被剖分成多个小三角单元，它们按照图 17-66 组织。

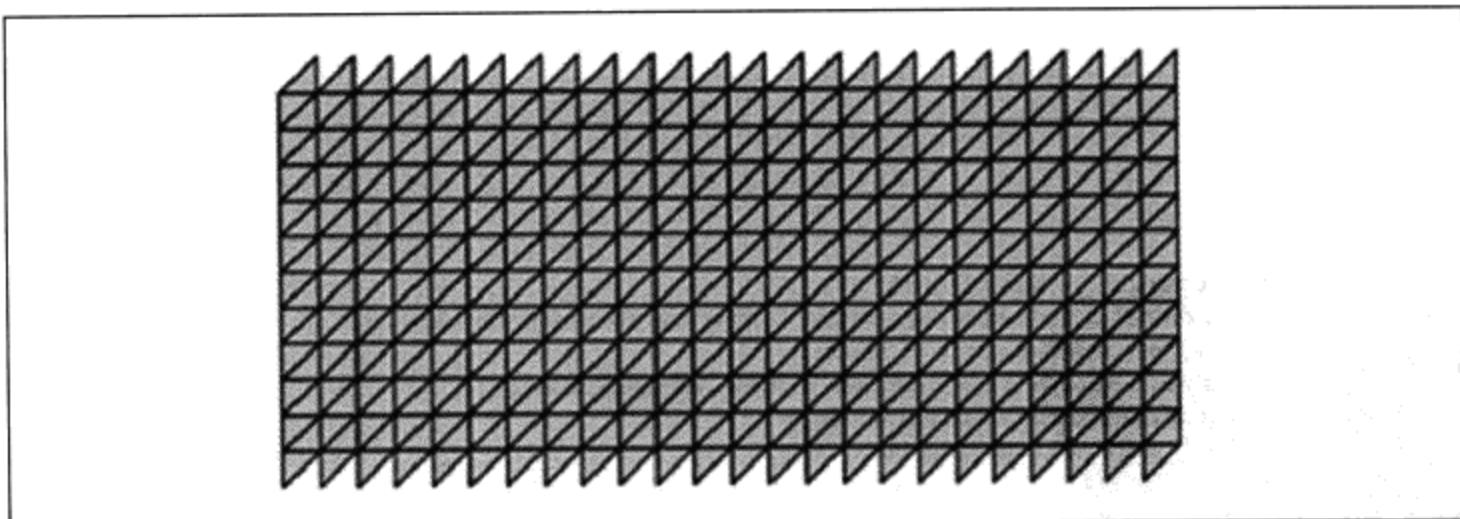


图 17-66 球面剖分单元的组织形式

它在图像中的纹理坐标可以用 (slice/slices 及 stack/stacks) 来表示，如图 17-67 所示。

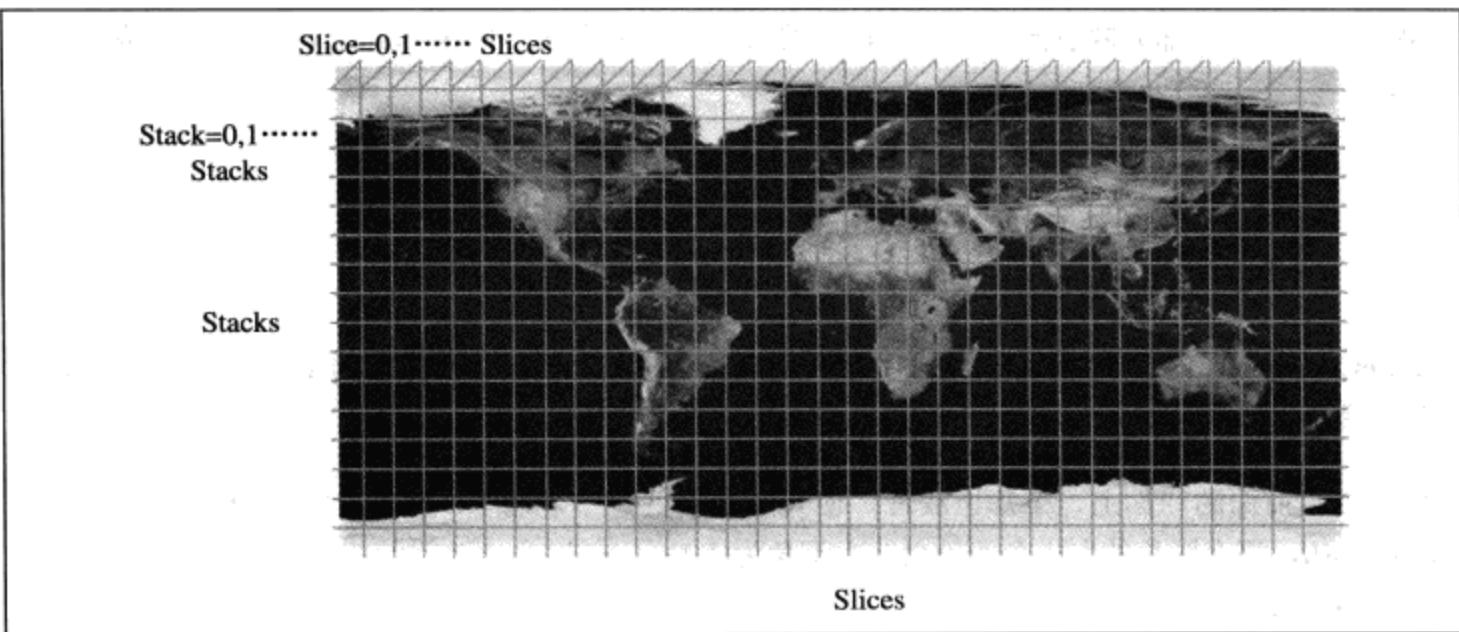


图 17-67 纹理坐标

我们在构建球面的 GenerateSphere 函数中添加纹理坐标的代码，然后新建一个 ImageBrush 画刷作为球的 Material 属性值，如代码 17-35 所示。

```
MeshGeometry3D GenerateSphere(Point3D center, double radius,
                               int slices, int stacks, ref ScreenSpaceLines3D
                               line3D)
{
    .....
    // 计算出来的点，然后加入到 MeshGeometry3D 的 Positions 属性
    for (int stack = 0; stack <= stacks; stack++)
    {
        mesh.TextureCoordinates.Add(
            new Point((double)slice / slices,
                      (double)stack / stacks));
    }
    .....
    return mesh;
}

public Sphere()
{
    InitializeComponent();
    GeometryModel3D geomod = new GeometryModel3D();
    geomod.Geometry = mesh;
    ImageBrush imagebrush = new ImageBrush();
    imagebrush.ImageSource =
        new BitmapImage(
            new Uri("earthmap.jpg", UriKind.Relative));
    geomod.Material = new DiffuseMaterial(imagebrush);
    .....
}
```

代码 17-35 新建一个 ImageBrush 画刷

程序运行效果如图 17-68 所示。

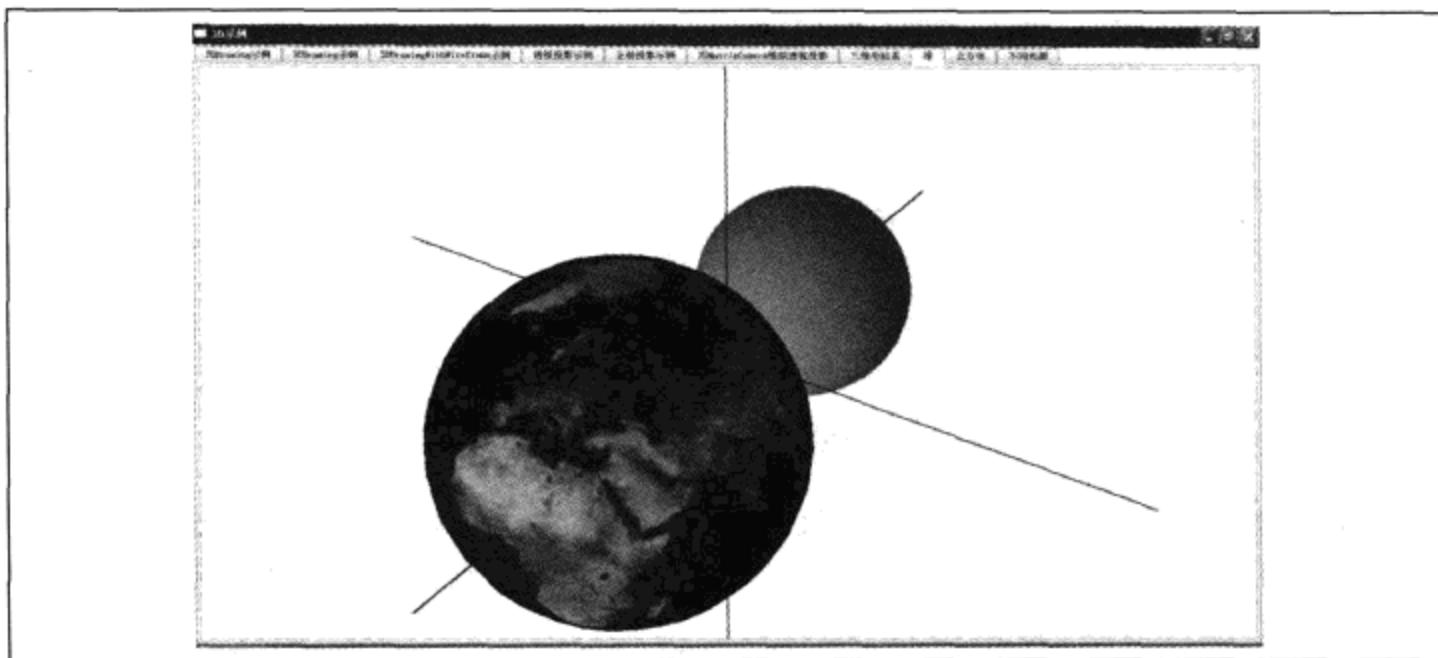


图 17-68 程序运行效果

17.6 动画和交互

动画和交互在 3D 编程中是一个高级主题，在这一节中我们将通过实现一个简单的 3D 地球程序（EarthDemo）来学习这个主题。

17.6.1 动画

让一个 3D 模型动起来的两种途径为让照相机和 3D 模型动。

让照相机动实际上是改变照相机的 Position、LookDirection 或者 UpDirection 等属性，这些属性都是依赖属性，因此动画可以应用到其上；而让 3D 模型动也有两种方法，常见的是对 WPF3D 变换类进行动画。WPF3D 和 2D 图形一样，也提供了平移、旋转和缩放等变换类，只不过这些变换类是在 3D 环境下应用的。通过对这些变换类的参数应用动画，从而使 3D 图形动起来；另一种方法是直接改变 3D 模型坐标点集合的值，即改变 MeshGeometry3D 的 Positions 属性，它是一个 Point3D 的集合（Point3DCollection）。这种对集合的操作本身比较烦琐，也需要一定的技巧。如果一不小心，可能会导致 3D 模型不动或者效率低下。

1. 照相机动

我们的目标实现类似 Google Earth 的效果，让地球自转着从太空中由远及近地出现在视野中。为实现地球由远及近，可以改变照相机的 Position 属性，如代码 17-36 所示。

```
<Window.Triggers>
    <EventTrigger RoutedEvent="Window.Loaded">
        <BeginStoryboard>
            <Storyboard x:Name="story" Completed="Storyboard_Completed">
                <Point3DAnimation Storyboard.TargetName="cam"
Storyboard.TargetProperty="Position" From="0 0 50" To="0 0 7" Duration="0:0:5">
```

```

        </Point3DAnimation>
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Window.Triggers>
<Grid>
    <tools:TrackballDecorator>
        <Viewport3D x:Name="viewport">
            <Viewport3D.Camera>
                <PerspectiveCamera x:Name="cam" Position="0,0,50"
LookDirection="0,0,-6" UpDirection="0,1,0">
.....

```

代码 17-36 改变照相机的 Position 属性使地球由远及近

我们将地球放置在了坐标原点(0,0,0)，照相机放置在(0,0,50)位置上，通过 Point3DAnimation 类将照相机的位置由(0,0,50)变换到(0,0,7)，看上去地球由远及近地出现在视野中。在 WPF3D 除了 Point3D 类型以外，还有如下对属性进行处理的动画类型。

(1) Vector3D：主要包括 ProjectionCamera 的 LookDirection 或者 UpDirection 属性，或者 DirectionLight 的 Direction 属性等应用动画。

(2) DoubleAnimation：主要包括 PerspectiveCamera 的 FieldOfView 属性或者 OrthographicCamera 类的 Width 属性应用动画。

(3) QuaternionAnimation：主要是对四元数进行动画，四元数通常用在 3D 模型的旋转变换中，是一种有力的数学工具³。

现在整个地球已经由远及近地出现，但是并没有自转，我们将通过 WPF3D 变换类来实现地球自转的动画。

2. 3D 模型动

WPF3D 提供的 3D 变换类，其类层次结构如图 17-69 所示。

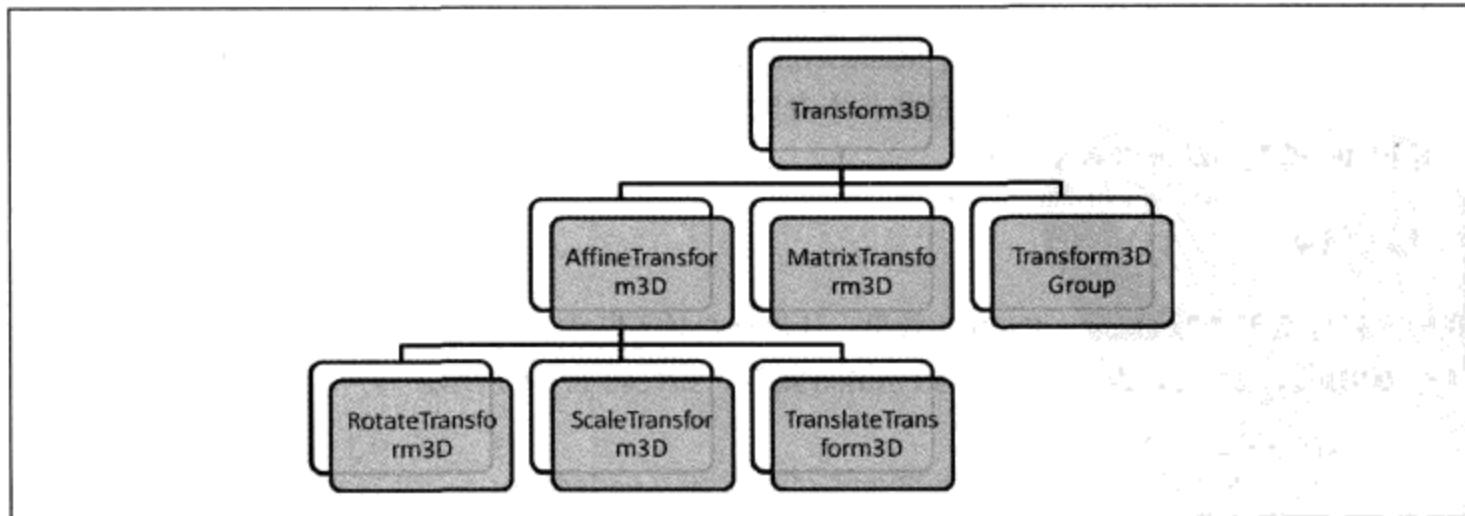


图 17-69 Transform3D 类层次结构

³ 在 Charles Peztold 所著的《精通 Windows3D 图形编程》一书中有详细介绍，见参考文献[4]。

由 Transform3D 派生了 3 个变换类，分别是 3D 仿射变换类（AffineTransform3D）、3D 矩阵变换类（MatrixTransform3D）和一个 Transform3D 的组合（Transform3DGroup）。平移变换、旋转变换和缩放变换都派生于仿射变换，均属于仿射变换的范畴。经过仿射变换直线仍然为直线，而平行线也总能保持平行。WPF3D 中也有非仿射变换，如透视投影变换。MatrixTransform3D 通过矩阵来变换，因此非常灵活；Transform3DGroup 可以将变换组合起来，形成组合变换。

同 2D 图形，这些变换通过为对象的 Transform 属性赋值来应用。在 WPF3D 中，主要有 3 个有 Transform 属性的类及其派生类，即 Camera、Model3D 和 ModelVisual3D。

地球模型是一个 GeometryModel3D，派生自 Model3D。因此可以为其添加一个旋转变换，如代码 17-37 所示。

```
<GeometryModel3D.Transform>
    <RotateTransform3D>
        <RotateTransform3D.Rotation>
            <AxisAngleRotation3D x:Name="YRotate" Angle="0" />
        </RotateTransform3D.Rotation>
    </RotateTransform3D>
</GeometryModel3D.Transform>
```

代码 17-37 添加一个旋转变换

RotateTransform3D 定义了属性 CenterX、CenterY 和 CenterZ，默认值为 0。它们确定了一个特殊的点，该点和 2D 图形旋转的中心点类似，在旋转时该点的位置保持不变；另外 RotateTransform3D 还定义了一个属性 Rotation，这是一个 Rotation3D 类型的对象。Rotation3D 是一个抽象类，它派生的两个类是 AxisAngleRotation3D 和 QuaternionRotation3D。前者通过指定旋转轴和旋转角度来确定旋转变换，后者通过四元数来确定。

AxisAngleRotation3D 有两个属性，其中 Axis 是 Vector3D 类型。默认值是(0,1,0)，即默认的旋转轴与 Y 轴平行的。如果 CenterX、CenterY 和 CenterZ 是默认值，则旋转轴为 Y 轴；如果 CenterX 和 CenterZ 不为 0，则旋转轴为(CenterX,y,CenterZ)；Angle 属性为确定旋转角度。

我们为地球添加围绕 Y 轴的旋转，且 Angle 初始值为 0。接下来为该变换添加动画，同样是在 Windows 的 Trigger 中添加，如代码 17-38 所示。

```
<Window.Triggers>
    <EventTrigger RoutedEvent="Window.Loaded">
        <BeginStoryboard>
            <Storyboard x:Name="story" >
                <Point3DAnimation Storyboard.TargetName="cam" Storyboard.TargetProperty="Position" From="0 0 50" To="0 0 7" Duration="0:0:5" FillBehavior="Stop">
                </Point3DAnimation>
                <DoubleAnimation Storyboard.TargetName="YRotate" Storyboard.TargetProperty="Angle" From="0" To="360" Duration="0:0:5" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Window.Triggers>
```

代码 17-38 为旋转变换添加动画

这样地球就可以自转着从太空中由远及近地出现在我们的视野中，如图 17-70 所示。

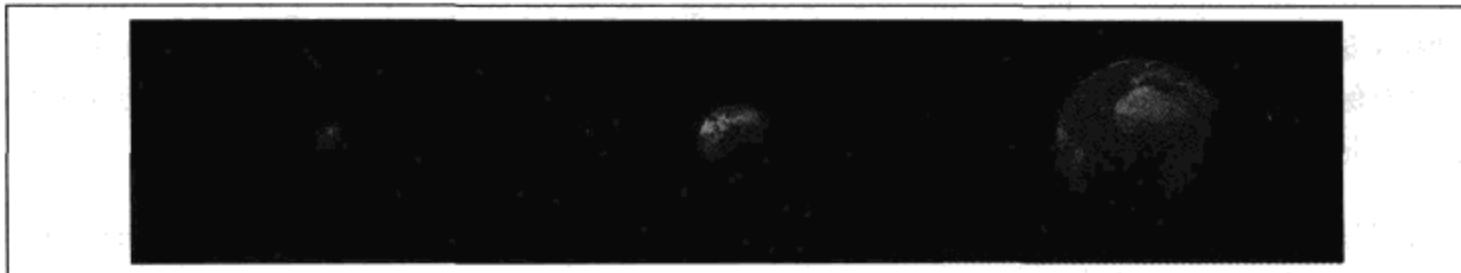


图 17-70 地球自转着从太空中由远及近地出现在我们的视野

如果希望地球出现在视野中仍然保持一定的角速度自转，则可以再添加一个 DoubleAnimation 动画，只不过要将其开始时间设置为前两个动画结束后的时间。然后将 RepeatBehavior 设置为 Forever，如代码 17-39 所示。

```
<DoubleAnimation BeginTime="0:0:5" Storyboard.TargetName="YRotate" Storyboard.TargetProperty="Angle" From="0" To="360" Duration="0:0:30" RepeatBehavior="Forever"/>
```

代码 17-39 地球继续保持角速自转

17.6.2 交互

和三维模型交互是 3D 编程当中的一个很重要的部分。在这一小节里我们继续完善 EarthDemo 这个例子，通过键盘实现自转的暂停和恢复，通过鼠标滚轮实现场景的拉近和拉远，通过鼠标拖动实现地球的漫游。

1. 地球自转的暂停和恢复

前面通过将旋转变换应用动画来实现地球的自转，也可以使用 CompositionTarget 的 Rendering 事件来实现。在窗口的构造函数中我们为 Rendering 事件添加事件处理函数 CompositionTarget_Rendering，在其中改变地球旋转变换（YRotate）的 Angle 值。当 isstop 为 true 时，将事件处理函数移除。窗口的 KeyDown 事件处理函数均判断按下的键是否为 S 键，如果是，则停止动画；再按一次则继续动画。停止和继续均由 isstop 变量控制，如代码 17-40 所示。

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        CompositionTarget.Rendering += new EventHandler(CompositionTarget_Rendering);
    }
    private bool isstop = false;
    void CompositionTarget_Rendering(object sender, EventArgs e)
    {
        YRotate.Angle++;
        if (YRotate.Angle > 360)
            YRotate.Angle = 0;
        if (isstop)
        {
```

```

        CompositionTarget.Rendering -= new EventHandler(CompositionTarget_
Rendering);
        isstop = true;
    }
}

private void Window_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.S)
    {
        if (isstop)
        {
            CompositionTarget.Rendering += new EventHandler(CompositionTarget_
Rendering);
            isstop = false;
        }
        else isstop = true;
    }
}
}

```

代码 17-40 动画的停止与继续

2. 拉近和拉远场景

拉近和拉远场景通过改变照相机的 Position 属性，主要是 Z 轴变量来实现，如代码 17-41 所示。

```

private void earthmodel_MouseWheel(object sender, MouseWheelEventArgs e)
{
    double z = cam.Position.Z;
    if (z > 100)
    {
        z = 99;
        cam.Position = new Point3D(0, 0, z);
        return;
    }
    if (z < 4)
    {
        z = 5;
        cam.Position = new Point3D(0, 0, z);
        return;
    }
    z = z - (double)(e.Delta/60);
    cam.Position = new Point3D(0, 0, z);
}

```

代码 17-41 场景的拉近和拉远

3. 鼠标的拖动实现地球的漫游

利用鼠标拖动实现地球的漫游一般来说需要通过 HitTest 检测判断可否选取 3D 模型，然后在相应的鼠标事件处理函数中实现拖动功能。在 WPF3D 中提供了不同层次的交互方法，这些方法由最底层的 HitTest 检测到最高级的跟踪球。

(1) HitTest 检测

在 3D 场景中同样可以通过 VisualTreeHelper 来实现 HitTest 检测。具体来说，在处理 3D 视图对象时

程序能够得到鼠标所指坐标下的 ModelVisual3D 对象。如果多个对象重叠出现在同一个点位置上，可以首先获取鼠标所指坐标下的前景 ModelVisual3D。然后获取更远背景的 ModelVisual3D 对象，同时还可以得到 GeometryModel3D、MeshGeometry3D、某个特定三角形的各点坐标，以及单击坐标在该三角形中的权重。

这个 HitTest 方法仍然是 2D 图形的方法，只不过第 1 个参数需要设置为 Viewport3D 对象；第 2 个参数是一个回调函数，这是一个可选的回调方法。可以为 null，用于过滤结果。第 3 个参数是一个对应于结果的回调函数，不能为 null；第 4 个参数是一个 HitTestParameters 对象，HitTestParameters 是一个抽象类，并含有两个派生类。其中 PointHitTestParameters 指用一个具体的二维点来 HitTest 检测，而 GeometryHitTestParameters 指用一个 2D 几何对象来实现 HitTest 检测，如代码 17-42 所示。

```
public static void HitTest(Visual reference, HitTestFilterCallback filterCallback, HitTestResultCallback resultCallback, HitTestParameters hitTestParameters);
```

代码 17-42 2D 图形的 HitTest 方法

VisualTreeHelper 提供了一个 3D 的 HitTest 方法，第 1 个参数是一个 Visual3D 对象；第 4 个参数是一个 HitTestParameters3D 对象，该对象只有一个派生类 RayHitTestParameters3D。它是由一个点和一个向量组成的一条射线，HitTest 检测沿着这条射线执行，如代码 17-43 所示。

```
public static void HitTest(Visual3D reference, HitTestFilterCallback filterCallback, HitTestResultCallback resultCallback, HitTestParameters3D hitTestParameters);
```

代码 17-43 一个 3D 的 HitTest 方法

第 3 个参数的回调函数是该方法的关键，它有一个 HitTestResult 类型的参数。在 Viewport3D 中执行单击测试时，实际上该参数 RayMeshGeometry3DHitTestResult，该类含有 VisualHit、ModelHit 和 MeshHit 等其他属性。这些属性提供了足够的信息，用于判断选中的图形对象，以及其选中的部分，如代码 17-44 所示。

```
public void HitTest(object sender,
System.Windows.Input.MouseEventArgs args)
{
    Point mouseposition = args.GetPosition(myViewport);
    Point3D testpoint3D = new Point3D(mouseposition.X, mouseposition.Y, 0);
    Vector3D testdirection = new Vector3D(mouseposition.X, mouseposition.Y, 10);
    PointHitTestParameters pointparams = new PointHitTestParameters(mouseposition);
    RayHitTestParameters rayparams = new RayHitTestParameters(testpoint3D, testdirection);

    //test for a result in the Viewport3D
    VisualTreeHelper.HitTest(myViewport, null, HTResult, pointparams);
}

public HitTestResultBehavior HTResult(System.Windows.Media.HitTestResult rawresult)
{
    //MessageBox.Show(rawresult.ToString());
    RayHitTestResult rayResult = rawresult as RayHitTestResult;
```

```

        if (rayResult != null)
        {
            RayMeshGeometry3DHitTestResult rayMeshResult = rayResult as
RayMeshGeometry3DHitTestResult;

            if (rayMeshResult != null)
            {
                GeometryModel3D hitgeo = rayMeshResult.ModelHit as
GeometryModel3D;

                UpdateResultInfo(rayMeshResult);
                UpdateMaterial(hitgeo, (side1GeometryModel3D.Material as
MaterialGroup));
            }
        }
        return HitTestResultBehavior.Continue;
    }
}

```

代码 17-44 第 3 个参数的回调函数

(2) UIElement3D

在.NET 3.0 中,从 Visual3D 派生的还只有 ModelVisual3D。在.NET 3.5 中多了一个新类 UIElement3D,它本身也是一个抽象类。由该类又派生了两个类,即 ModelUIElment3D 和 ContainerUIElement3D。后者是其他 Visual3D 的一个容器;前者可以如其他 2DElement 一样可以处理鼠标键盘事件,这样也省去了 HitTest 的检测。如将 EarthDemo 的地球模型封装成一个 ModelUIElment3D,同时为添加一个鼠标左键按下的事件处理函数。在其中只是简单地弹出一个消息框,这种方式和 UIElement 相同。ModelUIElement3D 的组织方式类似 ModelVisual3D,只不过在 Model 属性中放置 Model3D,如代码 17-45 和代码 17-46 所示。

```

<ModelUIElement3D x:Name="earthmodel" MouseLeftButtonDown=
"earthmodel_MouseLeftButtonDown">
    <ModelUIElement3D.Model>
        <GeometryModel3D Geometry="{Binding Source={StaticResource sphere},
Path=Geometry}">
            <GeometryModel3D.Material>
                <DiffuseMaterial>
                    <DiffuseMaterial.Brush>
                        <ImageBrush ImageSource="earthmap.jpg"/>
                    </DiffuseMaterial.Brush>
                </DiffuseMaterial>
            </GeometryModel3D.Material>
            <GeometryModel3D.Transform>
                <RotateTransform3D >
                    <RotateTransform3D.Rotation>
                        <AxisAngleRotation3D x:Name="YRotate" Angle="0"/>
                    </RotateTransform3D.Rotation>
                </RotateTransform3D>
            </GeometryModel3D.Transform>
        </GeometryModel3D>
    </ModelUIElement3D.Model>
</ModelUIElement3D>

```

代码 17-45 MainWindow.xaml

```

private void earthmodel_MouseLeftButtonDown(object sender,
MouseButtonEventArgs e)

```

```
{  
    MessageBox.Show("EarthDemo");  
}
```

代码 17-46 MainWindow.xaml.cs

(3) 跟踪球

跟踪球即用户在 3D 场景中的某处绕着一个假想的中心轴拖动时物体会相应旋转，旋转量依赖于拖动的距离。很多 3D 图形程序都使用跟踪球，WPF 并没有提供跟踪球的实现，在 3DTools 的开源项目提供了一个称为“TrackballDecorator”类的跟踪球实现。如果希望有拖动的功能，只需要在 Viewport3D 前面加上两个语句，如代码 17-47 所示。

```
<Window x:Class="mumu_EarthDemo.MainWindow"  
.....  
    xmlns:tools="clr-namespace:_3DTools;assembly=3DTools">  
<tools:TrackballDecorator>  
    <Viewport3D x:Name="viewport">  
        .....    </Viewport3D>  
</tools:TrackballDecorator>
```

代码 17-47 鼠标拖动的功能

WPF3D 跟踪球的本质是通过移动照相机来实现旋转，所以当对照相机应用动画时 WPF3D 的跟踪球无法实现漫游功能。因此最好的办法是避免照相机动画，而用 3D 模型动画取代。EarthDemo 地球的自转通过使地球模型沿轴旋转，而不是让照相机旋转实现的。

17.7 接下来做什么

WPF 并不擅长专业的 3D 图形，但是用于学习 3D 相当有益。如果希望深入研究 WPF3D 图形编程，作者推荐一本相当经典的书籍，即由 Charles Peztold 所著的《精通 Windows3D 图形编程》，英文名为《3D Programming for Windows: Three-Dimensional Graphics Programming for the Windows Presentation Foundation》。

参考文献

- [1] “北风之神”风清远整理，云中孤雁制作 《金庸全集典藏版 鹿鼎记》，“第三十五回 曾随东西南北路 独结冰霜雨雪缘”。
- [2] crashandburn5, 2009, Code Project, Making 3D Application with WPF http://www.codeproject.com/KB/WPF/Cube_App.aspx.
- [3] Dava Shreiner etc, OpenGL Programming Guide: The Official Guide to Learning OpenGL(2th Edition) Addison-Wesley Professional.
- [4] Charles Peztold 2009, 《精通 Windows3D 图形编程》，清华大学出版社。

第 18 章

文本和文档——从黑风双煞的“练门”说起

这铁尸（梅超风）浑号中有一个“铁”字，殊非偶然，周身真如铜铸铁打一般。她后心给全金发秤锤击中两下，却似并未受到重大损伤，才知她横练功夫亦已到了上乘境界。眼见她除了对张阿生的尖刀、韩小莹的长剑不敢以身子硬接之外，对其余兵刃竟是不大闪避，一味凌厉进攻……

陈玄风（铜尸）哼了一声，这时电光又是一闪。郭靖只见抓住自己的人面色焦黄，双目射出凶光，可怖之极。大骇之下，顺手拔出腰间的匕首，向他身上插去。这一下正插入陈玄风小腹的肚脐，八寸长的匕首直没至柄。陈玄风狂叫一声，向后便倒。他一身横练功夫，练门正是在肚脐之中。别说这柄匕首锋锐无比，就是寻常刀剑碰中了他练门，也是立时毙命。

——《射雕英雄传》，“第四回 黑风双煞”^[1]

这一段讲的是江南七怪恶斗黑风双煞，黑风双煞如鬼魅一般来去无踪并且两口子“铜尸”陈玄风和“铁尸”梅超风都是一身横练功夫，寻常兵刃都无法伤得了他们。但是殊不知“善泳溺水，平地覆车”，这个武功厉害之极的陈玄风，因为“练门”被匕首所刺，竟然丧生在一个全然不会武功的小儿郭靖之手。

应用程序中也有一样东西，它们出入也恍若幽灵。你总觉得背后凉飕飕，似乎它们时时刻刻都在。但猛一回头，只是斜阳、残砖和断瓦。等你继续埋头前行时，它们又冷不丁地从一个石缝里飘忽出来，这就是文本。我们开篇就要抓住文本的“练门”，让它无处遁形，这“练门”正是 `TextElement`。

本章内容如下。

- (1) 从 `TextElement` 说起。
- (2) `TextBlock` 控件。
- (3) 理解 WPF 的文档。
- (4) 文档控件。

(5) 实现一个文档的浏览器。

(6) 接下来做什么。

18.1 从 TextElement 说起

18.1.1 文本

文本和很多 WPF 中的要素不同，最常出现的地方是相关的文本控件，如 TextBlock、TextBox 或者 RichTextBox 等。

即使在一个按钮上设置一个文本的字体大小，也可能产生困惑，因为至少有如下 4 种设置方式。

- (1) 设置 Button 从 Control 派生的属性 FontSize。
- (2) 设置 TextBlock 的 FontSize 附加属性。
- (3) 设置 TextElement 的 FontSize 附加属性。
- (4) 设置父元素的 FontSize 属性。

如代码 18-1 所示（详见 `mumu_FontSizeInButton` 工程）。

```
<Window x:Class="mumu_FontSizeInButton.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="FontSizeDemo" Height="300" Width="300" FontSize="14"><!-- (4) -->
<StackPanel>
    <!--(1)-->
    <Button      HorizontalAlignment="Center"      VerticalAlignment="Center"
FontSize="24"TextBlock.FontSize="24"TextElement.FontSize="24"
```

代码 18-1 4 种方式设置文本的字体大小

从程序的运行结果来看，这 4 种方法等价，如图 18-1 所示。

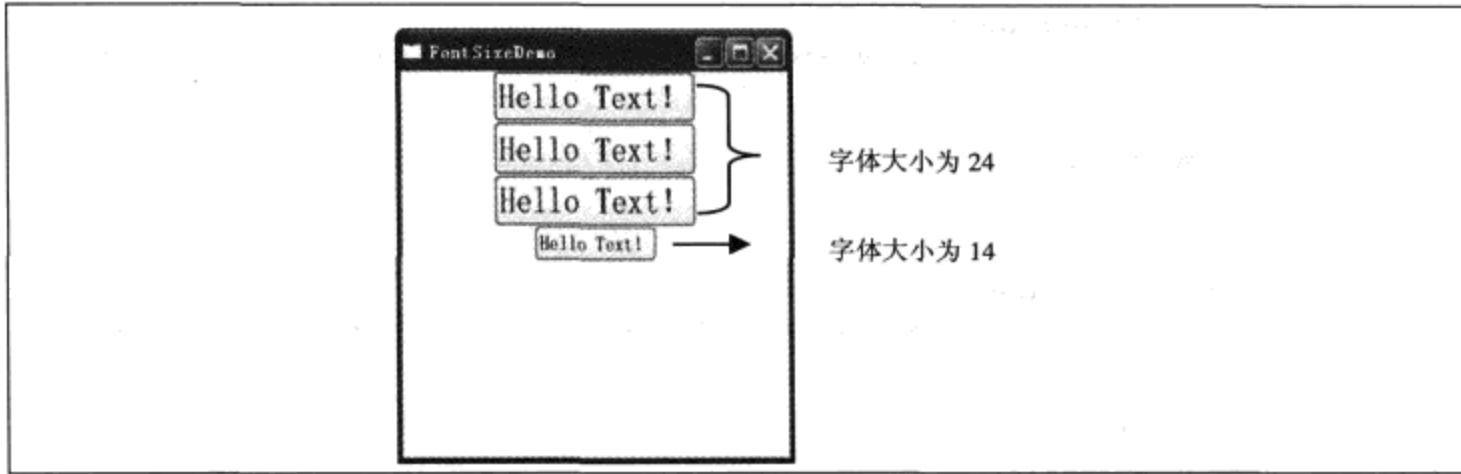


图 18-1 程序运行结果

18.1.2 TextElement

我们依旧以 `FontSize` 属性为例，在 WPF 中有 `FontSize` 属性的类为 `Control`、`AccessText`、`FlowDocument`、`Page`、`TextBlock` 和 `TextElement`。

其中仅 `Control` 类就包括了 `Button` 和 `Menu` 等众多子类。

`FontSize` 也是依赖属性，WPF 中通常会在类的静态构造函数中注册依赖属性。通过 Reflector 查看 `Control` 的构造函数，如代码 18-2 所示。

```
static Control()
{
    .....
    FontSizeProperty = TextElement.FontSizeProperty.AddOwner
(typeof(Control), new FrameworkPropertyMetadata(SystemFonts.MessageFontSize,
FrameworkPropertyMetadataOptions.Inherits));
}
```

代码 18-2 `Control` 的构造函数

在 `Control` 的静态构造函数中并没有为 `FontSize` 注册一个新的依赖属性，而是为 `TextElement` 已有的 `FontSize` 依赖属性添加一个新的拥有者。对下一个类 `AccessText` 继续 Reflector，如代码 18-3 所示。

```
static AccessText()
{
    FontSizeProperty = TextElement.FontSizeProperty.AddOwner
(typeof(AccessText));
    .....
}
```

代码 18-3 类 `AccessText` 的构造函数

`FlowDocument`、`Page`、`TextBlock` 及 `TextElement` 的静态构造函数如代码 18-4 所示。

```
// FlowDocument 的静态构造函数
static FlowDocument()
{
    .....
    _typeofThis = typeof(FlowDocument);
```

```

    FontSizeProperty =
        TextElement.FontSizeProperty.AddOwner(_typeofThis);
    ....
}
// Page 的静态构造函数
static Page()
{
    ....
    FontSizeProperty =
        TextElement.FontSizeProperty.AddOwner(typeof(Page));
    ....
}
// TextBlock 的静态构造函数
static TextBlock()
{
    ....
    FontSizeProperty =
        TextElement.FontSizeProperty.AddOwner(typeof(TextBlock));
    ....
}
// TextElement 的静态构造函数注册了 FontSize 属性
static TextElement()
{
    ....
    FontSizeProperty = DependencyProperty.RegisterAttached("FontSize",
typeof(double), typeof(TextElement), new FrameworkPropertyMetadata
(SystemFonts.MessageFontSize, FrameworkPropertyMetadataOptions.Inherits |
FrameworkPropertyMetadataOptions.AffectsRender |
FrameworkPropertyMetadataOptions.AffectsMeasure), new
ValidateValueCallback(TextElement.IsValidFontSize));
    ....
}

```

代码 18-4 FlowDocument、Page、TextBlock 以及 TextElement 类的静态构造函数

除了 TextElement 注册 FontSize 属性以外，其他类仅通过 AddOwner 方法重用 TextElement 的 FontSize 属性。从 TextElement 派生的类是通过继承重用 TextElement 的 FontSize 属性，即 WPF 中的所有类只要有 FontSize 属性，那么根源都来自 TextElement。只要掌握了 TextElement 的 FontSize 属性的使用方法，就掌握了 WPF 中所有类的 FontSize 属性的使用方法。

18.1.3 TextElement 的属性

TextElement 中关于文本设置的属性如下。

- (1) **FontFamily** 属性：设置字体类型，如将 **FontFamily** 设置为“华文仿宋”或者“宋体”等。
- (2) **FontSize** 属性：设置字号大小，单位可以用 WPF 中的设备无关单位，也可以用磅值、厘米和英寸。
- (3) **FontStretch** 属性：设置字体宽度缩放程度，可以将字体间隔设置为若干枚举值，如 **Condensed**（紧密）、**Normal**（正常）和 **Expanded**（扩展）等。
- (4) **FontStyle** 属性，设置字体样式，如设置字体为斜体（**Italic**）等。

(5) `FontWeight` 属性，设置字体粗细，如设置字体加粗（`Bold`）等。

(6) `Foreground` 属性，设置字体的前景色。

这些属性同 `FontSize` 属性，即其使用方式和本节介绍的内容完全一致。此外这些属性都是附加属性，因此可以在 XAML 中的任意元素中按照“`TextElement.Font**属性`”的格式设置。如果该元素或其子元素包含该属性，则可以通过继承应用到这些元素或者子元素的属性中。

1. `FontFamily`

`FontFamily` 通过一个字符串来标识某种字体，代码 18-5 为一个段落（`Paragraph`，该类继承自 `TextElement`）设置字体。

XAML

```
<Paragraph  
    FontFamily="Century Gothic" .....>
```

代码 18-5 `FontFamily` 设置多个字体

C#

```
Paragraph par = new Paragraph(run);  
par.FontFamily = new FontFamily("Century Gothic");
```

代码 18-6 查找“Century Gothic”字体

`FontFamily` 也可以设置多种字体，如“`Century Gothic, Courier New`”。应用程序会先查找“`Century Gothic`”字体，如果找不到，则使用“`Courier New`”字体。同时字体文件还可以作为一个资源放在应用程序中，该文件的路径前面需要加上一个“#”符号。以便与其他资源文件区分，如代码 18-7 所示。

```
<TextBlock FontFamily="Font\#Lindsey" FontSize="30">  
    Hello Text!  
</TextBlock>
```

代码 18-7 字体文件路径前需加上一个“#”符号

2. `FontSize`

`FontSize` 属性设置字号大小，4 种单位如代码 18-8 所示。

```
<StackPanel>  
    <TextBlock Text="30 pixels" FontSize="30"/>  
    <TextBlock Text="30 磅值" FontSize="30pt"/>  
    <TextBlock Text="1 厘米" FontSize="1cm"/>  
    <TextBlock Text="0.5 英寸" FontSize="0.5in"/>  
</StackPanel>
```

代码 18-8 4 种 `FontSize` 属性单位

字号大小如图 18-2 所示。



图 18-2 字号大小

我们平常所用的号数“一号字”、“二号字”和磅数，以及尺寸的对应关系如表 18-1 所示。

表 18-1 常用号数“一号字”、“二号字”和磅数，以及尺寸的对应关系

序号	号数	磅数	尺寸 (mm)
1	大特号	72	25.305
2	特号	63	22.142
3	初号	54	18.979
4	小初号	42	14.761
5	大一号	36	12.653
6	一号	31.5	11.071
7	二号	28	9.841
8	小二号	21	7.381
9	三号	18	6.326
10	小三号	16	5.623
11	四号	14	4.920
12	小四号	12	4.218
13	五号	10.5	3.690
14	小五号	9	3.163
15	六号	8	2.812
16	小六号	6.875	2.416
17	七号	5.25	1.845
18	八号	4.5	1.581

3. FontStretch

FontStretch 描述字体宽度缩放程度，用百分比来表示。正常字体的百分比是 100%，这时字体宽度没有任何变形。FontStretch 除提供 Normal 的枚举值以外，还提供了另外 9 种不同的枚举值分别代表不同的缩放尺度，如表 18-2 所示。

表 18-2 9 种不同的枚举值

Font stretch	和 Normal 相比
UltraCondensed	50.0%
ExtraCondensed	62.5%
Condensed	75.0%
SemiCondensed	87.5%
Medium	100.0%
SemiExpanded	112.5%
Expanded	125.0%
ExtraExpanded	150.0%
UltraExpanded	200.0%

绝大多数字体只提供了枚举值为“Normal”的字体或者少数几种。几乎没有字体完全支持这 9 种不同缩放程度。代码 18-9 分别为字体“华文仿宋”和“Gill Sans MT”设置不同的 FontStretch 值。

```
<Border Grid.Row="0" BorderThickness="1" BorderBrush="AliceBlue" Margin="5">
    <StackPanel TextBlock.FontSize="30" TextBlock.FontFamily="华文仿宋">
        <TextBlock Text="字体: 华文仿宋"/>
        <TextBlock Text="Hello 文本! Condensed" FontStretch="Condensed"/>
        <TextBlock Text="Hello 文本! Normal" FontStretch="Normal"/>
        <TextBlock Text="Hello 文本! ExtraExpanded" FontStretch="ExtraExpanded"/>
    </StackPanel>
</Border>
<Border Grid.Row="1" BorderThickness="1" BorderBrush="AliceBlue" Margin="5">
    <StackPanel Grid.Row="1" TextBlock.FontSize="30"
TextBlock.FontFamily="Gill Sans MT">
        <TextBlock Text="字体: Gill Sans MT"/>
        <TextBlock Text="Hello 文本! Condensed" FontStretch="Condensed"/>
        <TextBlock Text="Hello 文本! Normal" FontStretch="Normal"/>
        <TextBlock Text="Hello 文本! ExtraExpanded" FontStretch="ExtraExpanded"/>
    </StackPanel>
</Border>
```

代码 18-9 为字体设置不同的 FontStretch 值

从图 18-3 中可以看到“华文仿宋”字体的 FontStretch 的设置没有任何影响。而对于“Gill Sans MT”字体，FontStretch 设置为 Condensed 后字体的宽度明显变小。

4. FontStyle

FontStyle 有 3 种不同的枚举值，其中默认的 Normal 表示该字体是一个常规的字体；Italic 和 Oblique

表示是一个斜体的字体。图 18-4 所示是字体为“Gill Sans MT”，而 FontStyle 属性不同的文本。



图 18-3 FontStyle 的设置对字体的影响

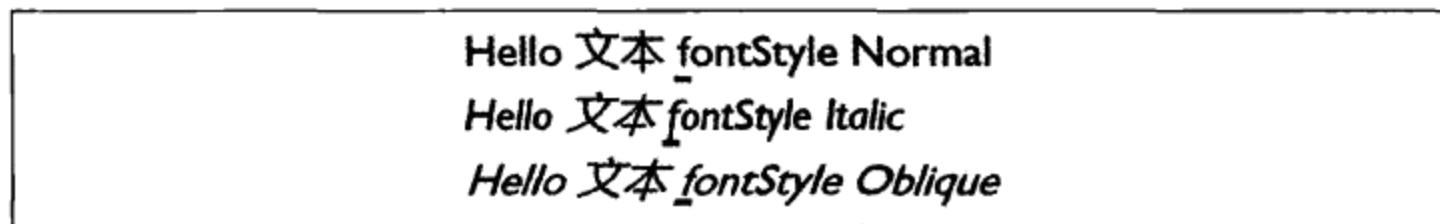


图 18-4 字体为“Gill Sans MT”而 FontStyle 属性不同的文本

Italic 和 Oblique 区别在于前者的字体是独立的，本身是为斜体设计的字体；Oblique 仅仅只是对字体做了一个错切变化。图中第 2 行字符与第 1 行及第 3 行略有区别，如字符“f”比较明显。

5. FontWeight

FontWeight 描述字体粗细，在 OpenType¹规范中将字体的粗细划分为 1~999。FontWeight 提供的枚举值代表的粗细程度如表 18-3 所示。

表 18-3 FontWeight 提供的枚举值代表的粗细程度

Font weight	粗细程度
Thin	100
ExtraLight	200
UltraLight	300
Light	400
Normal	500
Regular	
Medium	

¹ OpenType 是 TrueType 的扩展。

续表

Font weight	粗细程度
DemiBold	
SemiBold	600
Bold	700
ExtraBold	
UltraBold	800
Black	
Heavy	900
ExtraBlack	
UltraBlack	950

18.2 TextBlock 控件

承载文本的最简单控件是 TextBlock，代码 18-10 所示为其简单应用这一节我们会详细地讨论它的属性。

```
<TextBlock Text="Hello 文本! "/>
```

代码 18-10 TextBlock 的简单应用

18.2.1 与文本相关的属性

TextDecorations、TextTrimming、TextWrapping 和 TextAlignment 属性是 TextBlock 的相关文本属性，这些属性也广泛存在其他文本和文本控件中。

1. TextDecorations 属性

如果设置该属性，会有一条线从文本上面、中间或者下面穿过。该属性提供了 4 种不同的值来控制线条穿过的位罝，如图 18-5 所示。

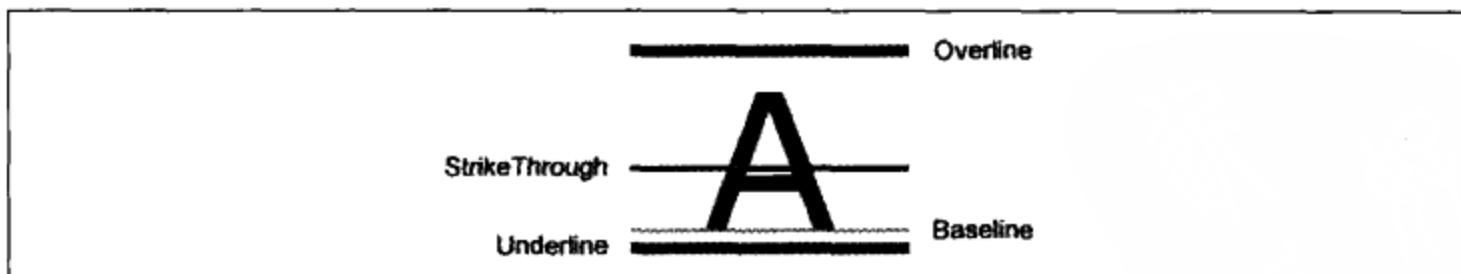


图 18-5 TextDecorations 属性的 4 种值控制线条位置

也可以任意组合使用这 4 个值，如设置 Overline 和 Baseline，则同时有上面和中间两条线从文本中穿过。代码 18-11 使用 XAML 来设置 TextDecorations 属性。

```
<TextBlock Name="underlineTextBlock" TextDecorations="Underline">The lazy dog</TextBlock>
<TextBlock Name="baseoverlineTextBlock"
TextDecorations="Baseline,Overline">The lazy dog</TextBlock>
```

代码 18-11 用 XAML 来设置 TextDecorations 属性

其效果如图 18-6 所示。



图 18-6 4 个值任意组合使用的效果

在 C# 代码中实现同样的效果则稍稍烦琐一些，如代码 18-12 所示。

```
// 为 underlineTextBlock 添加 Underline
private void SetUnderline()
{
    TextDecorationCollection myCollection = new
        TextDecorationCollection();
    TextDecoration myUnderline = new TextDecoration();
    myUnderline.Location = TextDecorationLocation.Underline;
    myCollection.Add(myUnderline);
    underlineTextBlock.TextDecorations = myCollection;
}

// 为 baseoverlineTextBlock 的文本添加 Underline 和 Baseline
private void SetBaseOverline()
{
    TextDecorationCollection myCollection = new
        TextDecorationCollection();
    TextDecoration mybaseline = new TextDecoration();
    mybaseline.Location = TextDecorationLocation.Baseline;
    TextDecoration myoverline = new TextDecoration();
    myoverline.Location = TextDecorationLocation.OverLine;
    myCollection.Add(mybaseline);
    myCollection.Add(myoverline);
    baseoverlineTextBlock.TextDecorations = myCollection;
}
```

代码 18-12 在 C# 代码中设置 TextDecorations 属性

因为 TextDecoration 有画笔属性，因此可以为这条穿过文本的线设置颜色、粗细、线型和渐变等画笔相关属性，代码 18-13 会实现图 18-7 所示的效果。

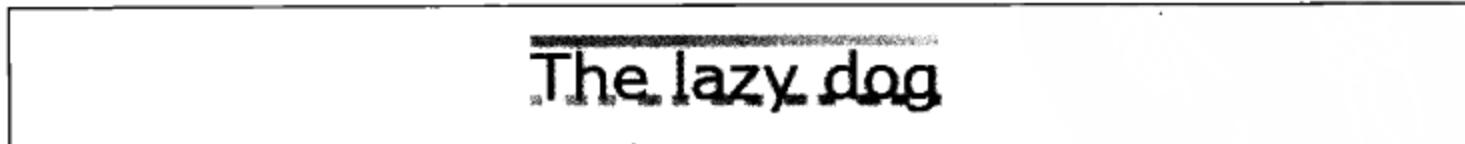


图 18-7 设置线的运行结果

```
<TextBlock FontSize="24" Width="180" VerticalAlignment="Center">The lazy dog
<TextBlock.TextDecorations>
    <TextDecoration Location="Baseline" PenThicknessUnit="FontRecommended">
        <TextDecoration.Pen>
            <Pen Thickness="3">
                <Pen.DashStyle>
```

```

        <DashStyle Dashes="0.5, 3, 1, 2" />
    </Pen.DashStyle>
    <Pen.Brush>
        <LinearGradientBrush Opacity="0.5"
            StartPoint="0,0.5" EndPoint="1,0.5">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Orange" Offset="0" />
                <GradientStop Color="Red" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Pen.Brush>
</Pen>
</TextDecoration.Pen>
</TextDecoration>

<TextDecoration Location="OverLine" PenThicknessUnit="FontRecommended">
    <TextDecoration.Pen>
        <Pen Thickness="3">
            <Pen.Brush>
                <LinearGradientBrush
                    StartPoint="0,0.5" EndPoint="1,0.5">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Color="LimeGreen" Offset="0" />
                        <GradientStop Color="Yellow" Offset="1" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Pen.Brush>
        </Pen>
    </TextDecoration.Pen>
</TextDecoration>
</TextBlock.TextDecorations>
</TextBlock>

```

代码 18-13 设置 TextDecoration 的画笔相关属性

AccessText、Inline 及其派生类、Paragraph、TextBlock、TextBox、TextParagraphProperties 和 TextRunProperties 类拥有 TextDecorations 属性。

2. Text Trimming 属性

Text Trimming 属性用来描述字符超出正常空间后的处理方式，提供的枚举值是 None、CharacterEllipsis 和 WordEllipsis。为 TextTrimming 属性设置不同枚举值的效果如图 18-8 所示。

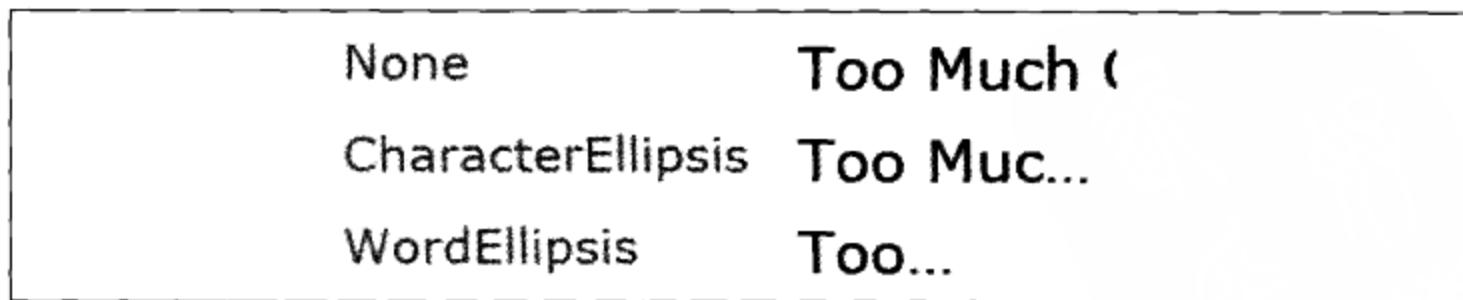


图 18-8 为 TextTrimming 属性设置不同的枚举值

当 TextTrimming 属性为 None 时，如果字符超出正常的显示空间，则将字符截断；为 CharacterEllipsis 和 WordEllipsis 时，则用一个省略号表示。不同的是 WordEllipsis 不会将一个完整的词截断，而 CharacterEllipsis 则不考虑词的完整性。如图中第 2 行的单词 Much 被截断；第 3 行则为了避免一个

完整的词截断，省略 Much 这个词。在中文中二者没有区别，如图 18-9 所示。

None	更详细的问题
CharacterEllipsis	更详细的
WordEllipsis	更详细的

图 18-9 中文中不同 TextTrimming 属性的效果

3. TextWrapping 属性

如果文本过长需要 TextBlock 换行显示，则使用 TextWrapping 属性。它提供的不同枚举值是 NoWrap、Wrap 和 WrapWithOverflow，如图 18-10 所示。

NoWrap	delicio
Wrap	delicio us food
WrapWithOverflow	delicio food

图 18-10 TextWrapping 提供的不同枚举值

当设置为 NoWrap 且文本超出了 TextBlock 的宽度时，不会自动换行；为 Wrap 和 WrapWithOverflow 时，文本会自动换行，二者的区别在于当一个词语的长度超过了 TextBlock 的宽度时 Wrap 会自动为这个词换行。WrapWithOverflow 会将这个词截断，而不换行。为了不破坏这个词的完整性，我们也可以为 TextBlock 的 IsHyphenationEnabled 属性设置为 true。这样会用一个“-”符号连接换行的一个词的两部分，如代码 18-14 所示。

```
<TextBlock      IsHyphenationEnabled="True"      Margin="10,5"      FontSize="24"  
TextWrapping="Wrap" Width="80" Text="delicious food" />
```

代码 18-14 IsHyphenationEnabled 属性为 True

运行结果如图 18-11 所示。

deli- cious food

图 18-11 运行结果

在中文中 Wrap 和 WrapWithOverflow 也没有区别。IsHyphenationEnabled 属性无效，如图 18-12 所示。

NoWrap IsHyphenationEnabled=true	美味的1
Wrap IsHyphenationEnabled=true	美味的 食物
WrapWithOverflow IsHyphenationEnabled=true	美味的 食物

图 18-12 中文中不同枚举值的影响

4. TextAlignment 属性

TextAlignment 属性表示如何对齐文本，它提供 4 种枚举值。其中 Left、Right 和 Center 分别表示文本左对齐、右对齐和居中。在一行的末尾，有些单词过长无法完整地显示，会另起一行显示。这样在行的末尾会留出一部分空间，而 Justify 会根据一行的空间合理调整文本的间距，如图 18-13 所示。

Left	Specifies whether the text in the object is left-aligned, right-aligned, centered, or justified.
Right	Specifies whether the text in the object is left-aligned, right-aligned, centered, or justified.
Center	Specifies whether the text in the object is left-aligned, right-aligned, centered, or justified.
Justify	Specifies whether the text in the object is left-aligned, right-aligned, centered, or justified.

行末尾留出了
一部分空间

合理调整了行
间距

图 18-13 TextAlignment 属性的 4 种不同枚举值

18.2.2 文本属性

在 TextBlock 中设置文本内容时最简单的方法使用 Text 属性中，该属性本身就是一个字符串。可以在其中显示一行简单的文本，也可以通过如下方式使文本自动换行，如代码 18-15 所示。

```
<TextBlock Margin="10,5" FontSize="20" Text="Hello!
Text!" />
```

代码 18-15 使文本自动换行

“`
`”表示一个换行符，以“`&`”开头，以“`;`”结尾，“`#`”后面可以是一个十进制或者十六进制的 ANSI 码值。如果是十六进制，则需要用“`x`”开头；十进制则可以直接跟在“`#`”之后。在 ANSI 码值中 10 表示换行符，因此也可以用“`
`”表示换行符，效果如图 18-14 所示。



Hello!
Text!

图 18-14 文本自动换行的效果

也可以通过添加一个 `LineBreak` 的标签实现换行，如代码 18-16 所示。

```
<TextBlock FontSize="20" >  
    Hello!<LineBreak/>Text!  
</TextBlock>
```

代码 18-16 添加一个 `LineBreak` 的标签实现换行

不要误认为“`Hello!<LineBreak/>Text!`”这段字符串赋给 `Text` 属性，这段代码做了部分简化，完整代码如代码 18-17 所示。

```
<TextBlock Margin="10,5" FontSize="20">  
    <TextBlock.Inlines>  
        <Run>Hello!</Run>  
        <LineBreak/>  
        <Run>Text!</Run>  
    </TextBlock.Inlines>  
</TextBlock>
```

代码 18-17 “`Hello!<LineBreak/>Text!`” 的完整代码

可以看出 `TextBlock` 的文本内容赋值给 `Inlines` 属性，它是一个 `Inline` 的集合。`Inline` 是流文档模型中的一个元素，可以支持复杂的文本表示。代码 18-18 将 `Hello` 更换为不同的字体，并且将其加粗和添加下画线。

```
<TextBlock Margin="10,5" FontSize="30">  
    <TextBlock.Inlines>  
        <Underline><Bold><Run FontFamily="Font\#Lindsey">  
            Hello!  
        </Run>  
        </Bold>  
        </Underline>  
        <LineBreak/>  
        <Run>Text!</Run>  
    </TextBlock.Inlines>  
</TextBlock>
```

代码 18-18 变换字体、加粗和添加下画线

效果如图 18-15 所示。



图 18-15 修改文本属性的效果

18.2.3 其他简单的文本控件

其他文本控件包括 Label、AccessText 和 TextBox，其中 Label 的作用是通过键盘的快捷键方式使与其关联的控件获得焦点；AccessText 可以为快捷键添加一条下画线；TextBox 则可以编辑文本。

18.3 理解 WPF 的文档

文档是一种复杂的文本形式，它也可以是文本、图片和表格的组合，如 Word 文档。WPF 中提供了两种类型的文档，即固定文档（fixed documents）和流文档（flow documents）。

固定文档的最大特点是“所见即所得”（WYSIWYG）²，文档中的所有内容位置都是精确和固定的。因此无论显示在 19 英寸的显示器，还是手机屏幕上或者通过打印机打印结果均相同，这种特性也称为“设备无关”。固定文档通常用在打印输出，它相当于 Adobe 公司的 PDF 文件。WPF 中也有类似 PDF 的文件，称为“XPS 文件”。

流文档与“设备相关”，其设计初衷是为了方便阅读，它类似于网页文件根据显示窗口尺寸、显示分辨率等动态地布局内容。

我们在应用程序中浏览文档多使用流文档，而打印时则使用固定文档。

18.3.1 ContentElement

理解流文档的关键是理解其模型，流文档模型中的要素如 Inline 及其派生类 Run、Bold 和 Underline。

流文档模型中的元素来自 ContentElement 类，从其派生的类有很多行为也与 UIElement 派生类类似，如支持鼠标和键盘事件等，但是 ContentElement 和 UIElement 有一个最大的不同之处是其无法绘制自身。

ContentElement 需要借助一个容器才能显示其内容，这正是我们前面提到的流文档和固定文档都需要一个控件来作为其容器的原因。从图 18-16 所示的类结构中可以看到 ContentElement 和 UIElement 从何处分开并自成体系。

² WYSIWYG 是“what you see is what you get”的缩写。

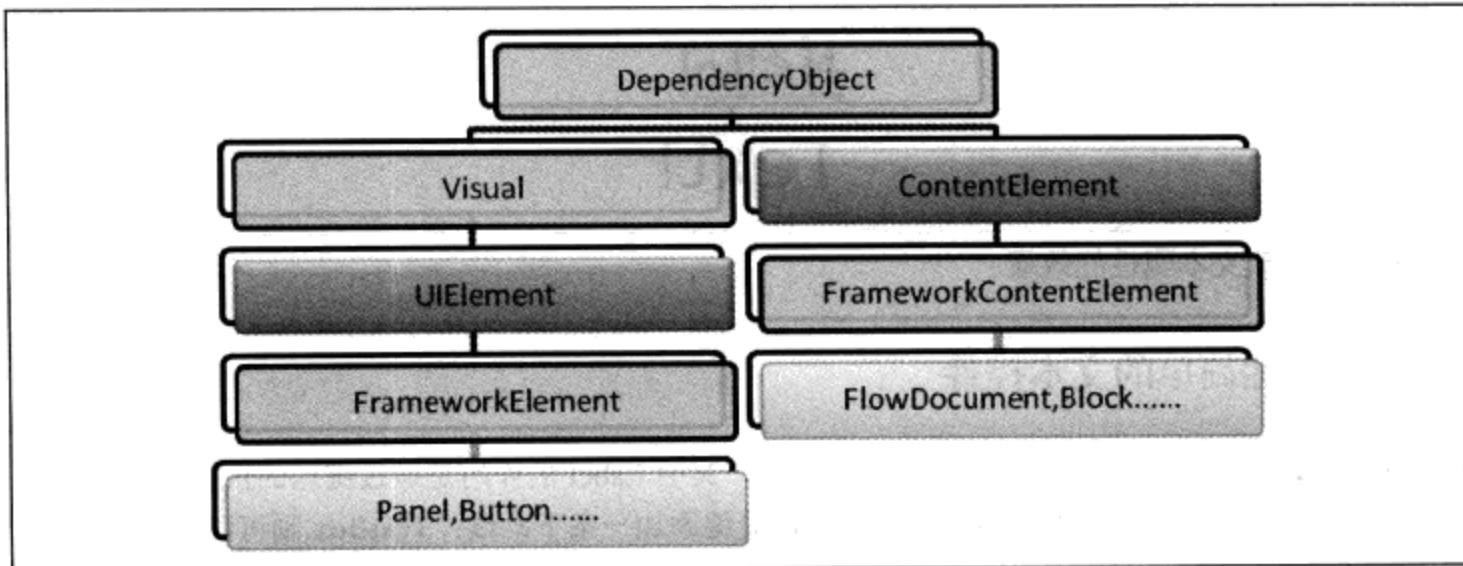


图 18-16 类结构

18.3.2 流文档模型

流文档模型中的所有元素从 ContentElement 派生，其类层次结构如图 18-17 所示。

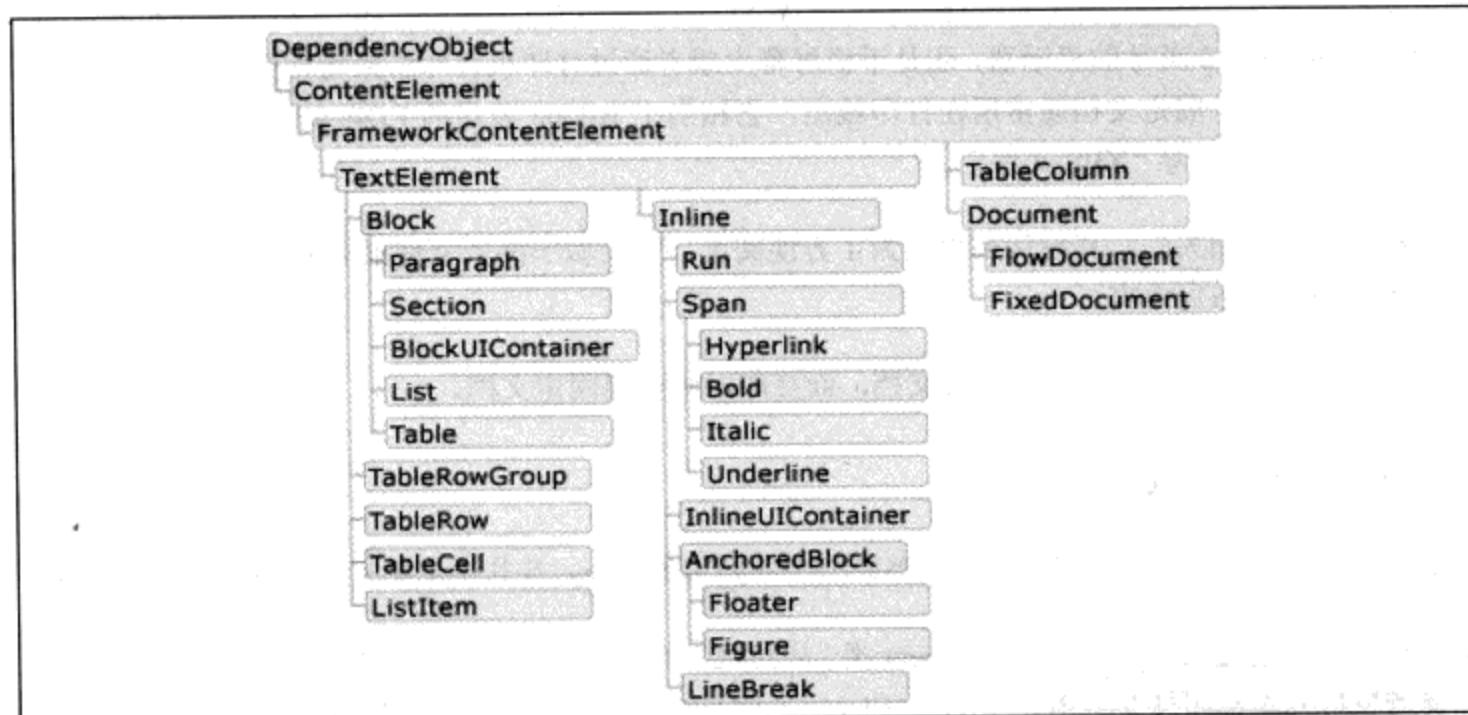


图 18-17 类层次结构

一个流文档中的内容均由 TextElement 的派生类组合而成，其中包括两大分支，即 Block 和 Inline。Block 代表的是一个矩形区域，如段落（Paragraph）和表格（Table）等；Inline 则是随着文本而变化的一个区域，通常是一个非矩形的区域。

1. Block 及其派生类

图 18-18 所示是一个简单的流文档，在从中可以看到从 Block 派生的典型流文档模型的元素。

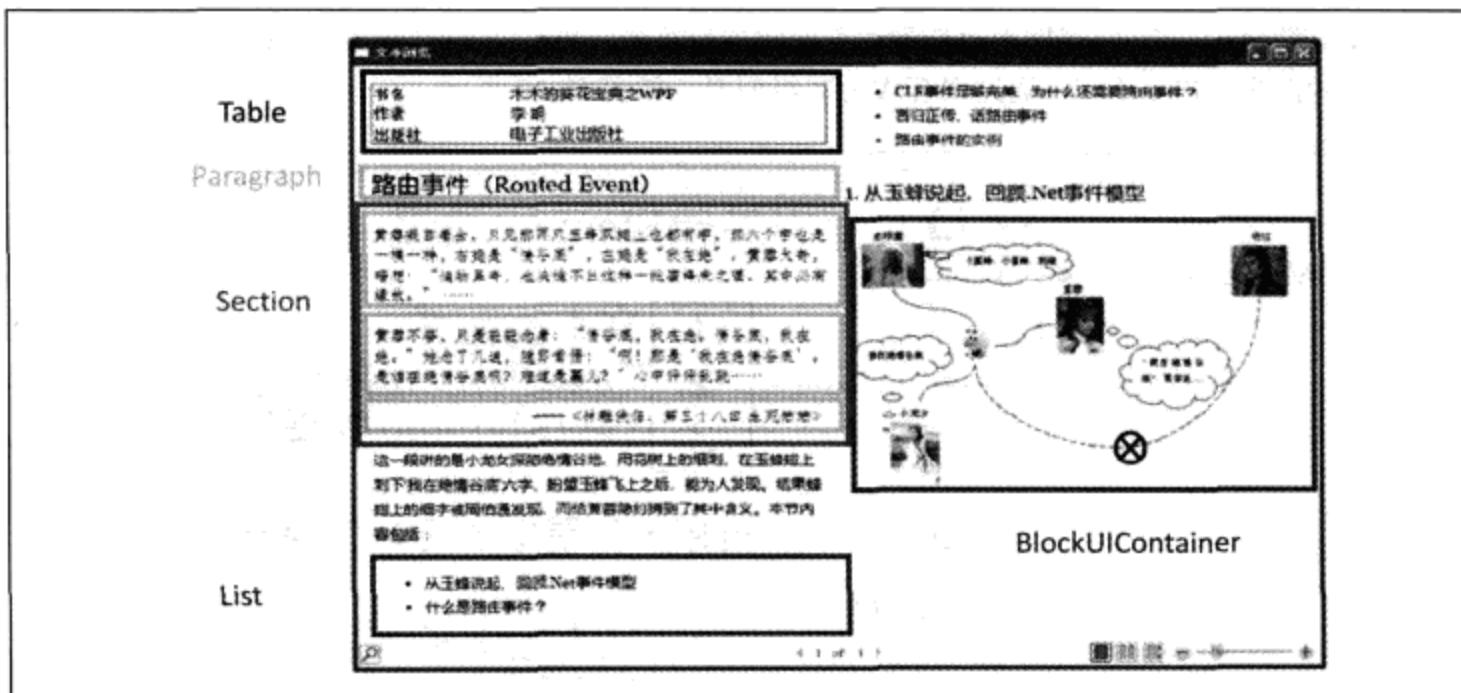


图 18-18 一个简单的流文档

从 Block 派生的主要是图中 5 种不同类型的要素。

(1) Paragraph (段落)

Paragraph 总是包含一个 Inline 的集合，即文档的内容。Paragraph 表示一个文本文档，如代码 18-19 所示。

```
<Paragraph FontSize="24" Background="AliceBlue">
    路由事件 (Routed Event)
</Paragraph>
```

代码 18-19 Paragraph

当直接在其内部放置文本时，它会隐式地创建一个 Run 元素，因此上面的一段代码的完整版如代码 18-20 所示。

```
<Paragraph Name="para" FontSize="24" Background="AliceBlue">
    <Run>
        路由事件 (Routed Event)
    </Run>
</Paragraph>
```

代码 18-20 Paragraph 隐式地创建一个 Run

理解这一点非常重要，当希望从代码中获得 Paragraph 的内容时由于它未提供类似 Text 这样的属性，所以必须从嵌套的 Run 对象里获得文本的内容，如代码 18-21 所示。

```
string txt = ((Run)para.Inlines.FirstInline).Text;
```

代码 18-21 从嵌套的 Run 对象里获得文本的内容

(2) List (列表)

List 通过设置 **MarkerStyle** 属性来改变列表前面的项目符号，并通过 **MarkerOffset** 来改变项目符号和文字之间的距离。List 还可以通过嵌套成为多级列表，如图 18-19 所示。

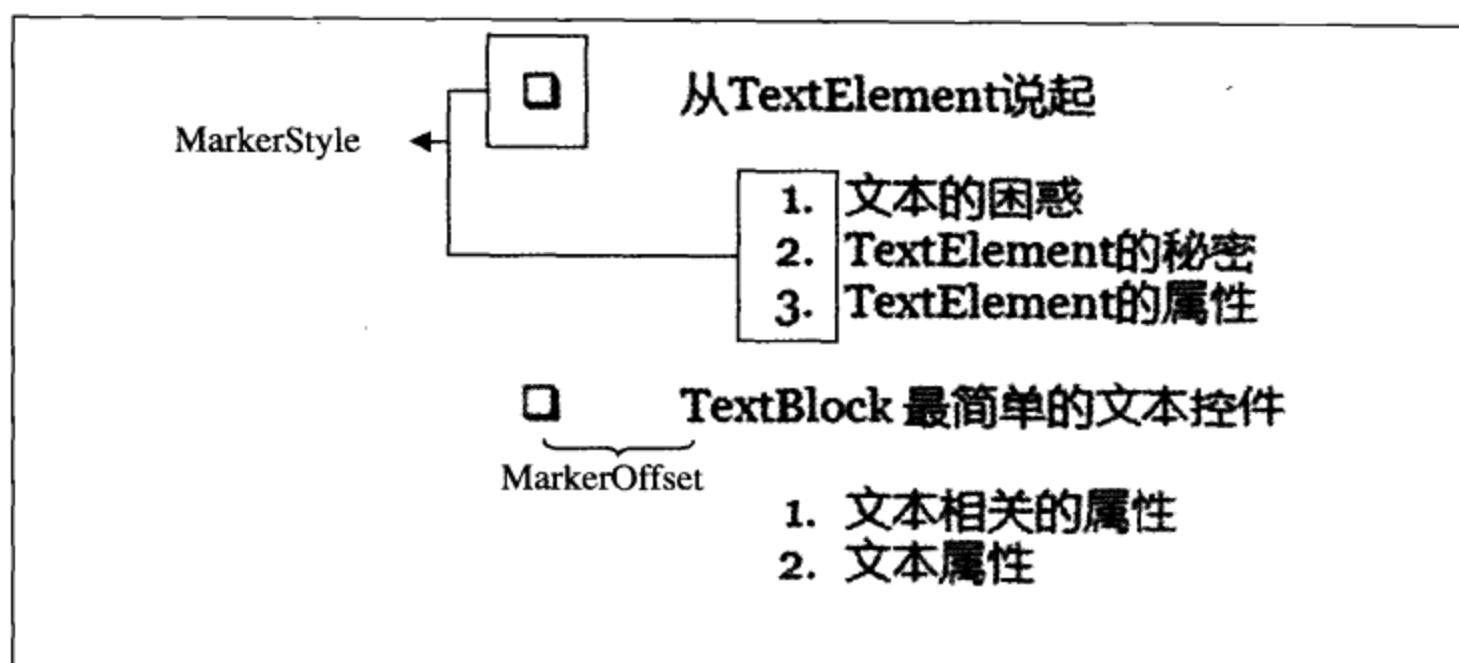


图 18-19 List 各属性

List 的实现如代码 18-22 所示：

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      HorizontalAlignment="Center" VerticalAlignment="Center">
  <FlowDocumentPageViewer>
    <FlowDocument>
      <List MarkerStyle="Square" MarkerOffset="40">
        <ListItem>
          <Paragraph>
            从 TextElement 说起
          </Paragraph>
        <List MarkerStyle="Decimal" MarkerOffset="10">
          <ListItem>
            <Paragraph>
              文本的困惑
            </Paragraph>
          </ListItem>
          <ListItem>
            <Paragraph>
              TextElement 的秘密
            </Paragraph>
          </ListItem>
          <ListItem>
            <Paragraph>
              TextElement 的属性
            </Paragraph>
          </ListItem>
        </List>
      </ListItem>
      <ListItem>
        <Paragraph>
          TextBlock 最简单的文本控件
        </Paragraph>
      </ListItem>
    </List>
  </FlowDocument>
</FlowDocumentPageViewer>
</Page>
```

```

<!--如果不希望编号从 0 开始，可以修改其 StartIndex 属性-->
<List MarkerStyle="Decimal" StartIndex="1">
    <ListItem>
        <Paragraph>
            文本相关的属性
        </Paragraph>
    </ListItem>
    <ListItem>
        <Paragraph>
            文本属性
        </Paragraph>
    </ListItem>
</List>
</List>
</FlowDocument>
</FlowDocumentPageViewer>
</Page>

```

代码 18-22 List 的实现

表 18-4 所示为 WPF 提供的不同项目符号样式。

表 18-4 WPF 提供的不同项目符号样式

MarkerStyle 属性值	项目符号
Box	■
Circle	○
Decimal	1, 2, 3.....
Disc	●
LowerLatin	a, b, c.....
UpperLatin	A, B, C.....
LowerRoman	i, ii, iii.....
UpperRoman	I, II, III.....
None	不显示任何内容
Square	□

(3) Table (表格)

Table 是针对流文档模型设计的表格，其中至少包含一个 TableRowGroup 元素。它是一组 TableRow 的集合，TableRow 代表一行。可以为 Table 添加多个表示行中一列的 TableCell 元素，每个 TableCell 元素包含一个 Block 元素。通常是 Paragraph，如图 18-20 所示。

如果希望为列指定明确的宽度，则必须设置 TableColumn；否则 WPF 会平均分配列宽。设置列宽时所有的列宽均使用相对单位（占总列宽的百分比）或者绝对单位，允许混合使用两种单位，每个 TableCell 可以通过 ColumnSpan 和 RowSpan 属性设置单元格占据的行列数。Table 有一个明显的局限，虽然它有 BorderThickness 和 BorderBrush 属性，但是设置的是表格的外边框。BorderThickness 和 BorderBrush 属性设置的是单元格的边框，因此在表格的每列中添加线非常困难。

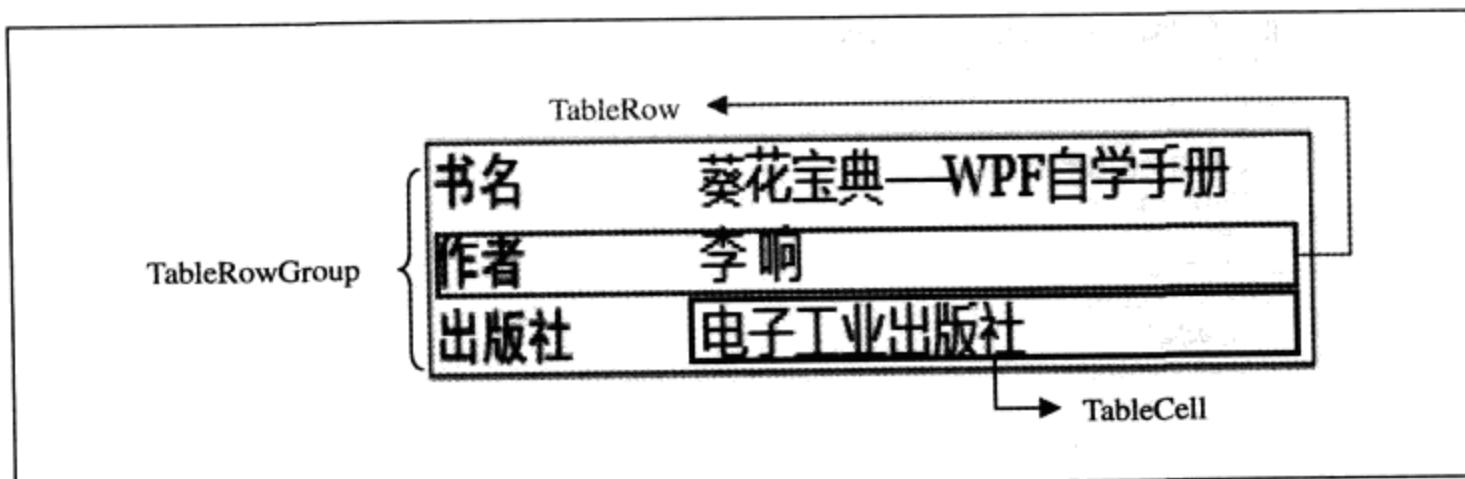


图 18-20 Table 各属性

Table 的实现如代码 18-23 所示：

```

<Table BorderThickness="1" BorderBrush="Blue">
    <Table.Columns>
        <TableColumn Width="3*" />
        <TableColumn Width="7*" />
    </Table.Columns>
    <TableRowGroup>
        <TableRow>
            <TableCell>
                <Paragraph FontFamily="黑体">
                    书名
                </Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph >
                    葵花宝典—WPF 自学手册
                </Paragraph>
            </TableCell>
        </TableRow>
        <TableRow>
            <TableCell>
                <Paragraph FontFamily="黑体">
                    作者
                </Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph >
                    李 响
                </Paragraph>
            </TableCell>
        </TableRow>
        <TableRow>
            <TableCell>
                <Paragraph FontFamily="黑体">
                    出版社
                </Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph >
                    电子工业出版社
                </Paragraph>
            </TableCell>
        </TableRow>
    </TableRowGroup>
</Table>

```

```
</TableRow>
</TableRowGroup>
</Table>
```

代码 18-23 Table 的实现代码

(4) BlockUIContainer

通过 BlockUIContainer 元素可以在文档中放置派生自 UIElement 的元素，如添加按钮、面板、图片和视频等。上例中的图片通过 BlockUIContainer 添加，如代码 18-24 所示。

```
<BlockUIContainer>
    <Image Source="路由事件.jpg"/>
</BlockUIContainer>
```

代码 18-24 通过 BlockUIContainer 添加图片

2. Inline 及其派生类

Inline 元素可以放置在 Block 级别的元素或者其他 Inline 元素中，WPF 中提供如下 Inline 元素。

- (1) Run: 其中包含普通文本，通常在 Paragraph 中添加文本时会隐式地创建一个该元素。
- (2) Span: 从该元素派生的有 Bold (加粗)、Italic (斜体)、Underline (下画线) 和 Hyperlink (超链接)。
- (3) LineBreak: 代表一个换行符。
- (4) InlineUIContainer: 可以嵌入到派生自 UIElement 的元素，并将 UIElement 放置在一个 Inline 级别的元素，而不是一个 Block 级别的元素中，如将一个按钮放在一行文本之间。
- (5) AnchoredBlock: 从该元素派生了两个特殊的 Inline 元素，即 Figure 和 Floater，专门为包含 Block 元素而设计。

Figure 类似一个小型的 FlowDocument，它嵌入到外部的 FlowDocument 中。其中的内容与外部隔离，外部的内容会把 Figure 包围起来。如代码 18-25 在前例中的“路由事件”标题旁添加一张小蜜蜂的图片：

```
<Paragraph x:Name="para" FontSize="24" Background="AliceBlue">
    <Figure Width="100" Height="100" HorizontalAnchor="ColumnRight"
    HorizontalOffset="-10" VerticalAnchor="ParagraphTop" VerticalOffset="-30">
        <BlockUIContainer>
            <Image Source="bee.png"/>
        </BlockUIContainer>
    </Figure>
    路由事件 (Routed Event)
</Paragraph>
```

代码 18-25 添加一个小蜜蜂的图片

程序运行结果如图 18-21 所示。

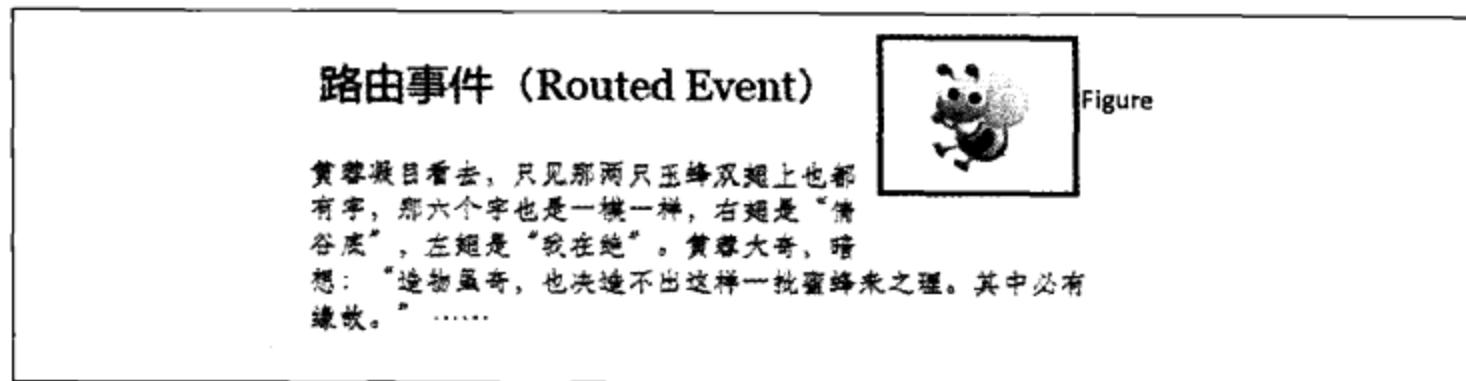


图 18-21 Figure 元素

其中 Figure 的位置通过 HorizontalAnchor、HorizontalOffset、VerticalAnchor 和 VerticalOffset 属性调整。Floater 元素是一种轻量级的 Figure，如果不需要更多控制位置，通常使用该元素即可。

3. 流文档中各个要素之间的关系

流文档中的各个要素之间的关系如图 18-22 所示。

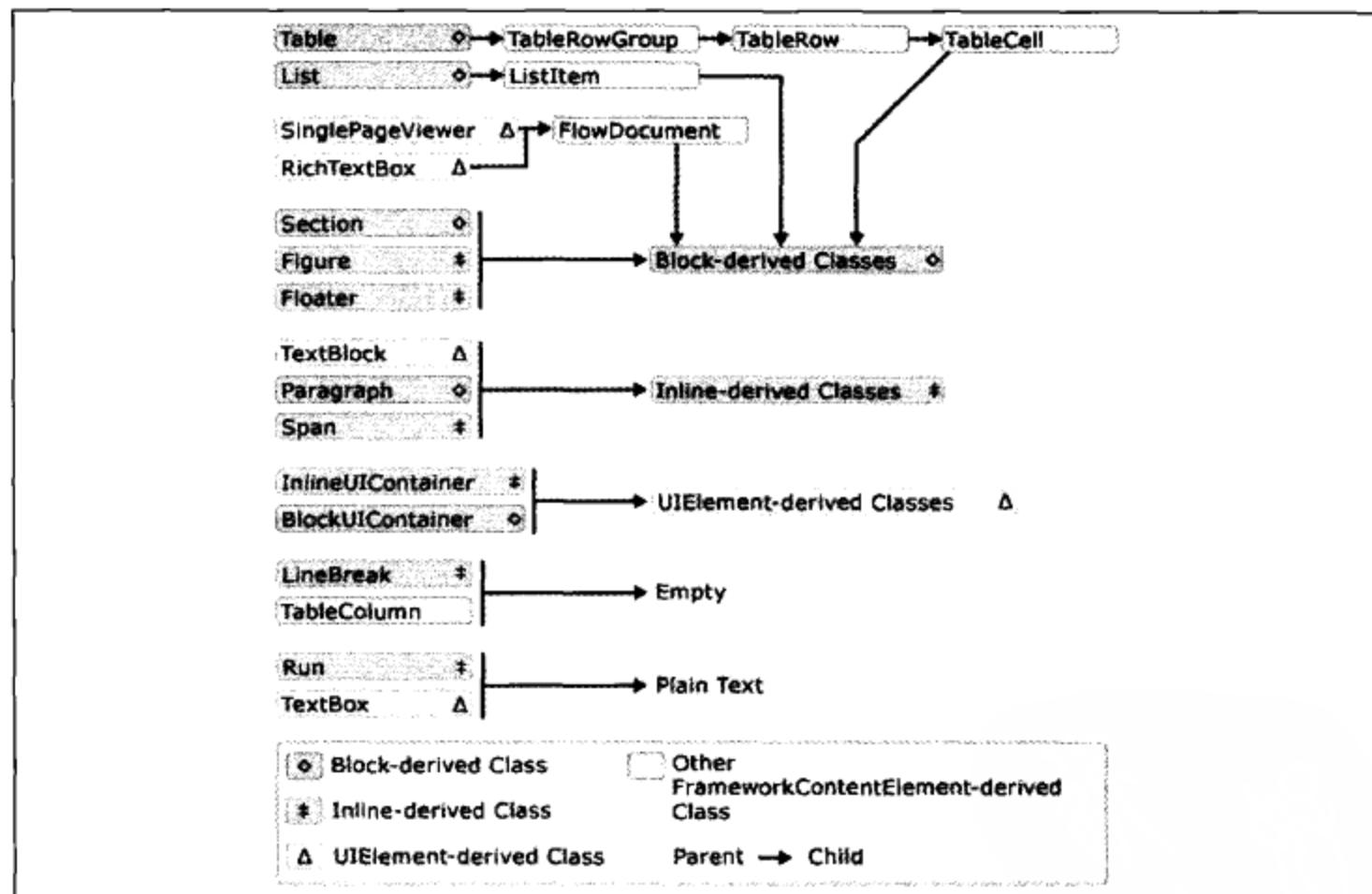


图 18-22 流文档的各个要素之间的关系图

4. 保留空格

XML 中多个空白字符会被压缩，因为 XAML 本身是基于 XML 的语言，所以也遵循这个原则，如下所示：

```
<Paragraph>
    <!--单词中间空了一格-->
    <Run>Hello Space!
    </Run>
    <LineBreak/>
    <!--单词中间空了多个空格-->
    <Run>Hello      Space!
    </Run>
    <LineBreak/>
    <!--单词开头空了多个空格-->
    <Run>          Hello Space!
    </Run>
</Paragraph>
```

从图 18-23 可以看出，无论开发人员在中间或者开头留多少空格，XAML 均会正确地压缩。



```
Hello Space!
Hello Space!
Hello Space!
```

图 18-23 没有设置 `xml:space` 属性

如果需要保留空格，则将 `xml:space` 属性设置为 `preserve`，如代码 18-26 所示：

```
<Paragraph>
    <!--单词中间空了一格-->
    <Run xml:space="preserve">Hello Space!</Run>
    <LineBreak/>
    <!--单词中间空了多个空格-->
    <Run xml:space="preserve">Hello      Space!</Run>
    <LineBreak/>
    <!--单词开头空了多个空格-->
    <Run xml:space="preserve">          Hello Space!</Run>
</Paragraph>
```

代码 18-26 XML 中空格的保留

程序运行结果如图 18-24 所示。



```
Hello Space!
Hello Space!
Hello Space!
```

图 18-24 设置 `xml:space` 属性等于 `preserve`

18.3.3 固定文档

流文档适合在屏幕上阅读，因其可以动态地布局复杂且大量的文本内容；固定文档使用精确的固定布局，支持字体嵌入，因此显示和打印的效果相同。WPF 中的固定文档主要指后缀名为“XPS”的文档，它不仅是 WPF 的一部分，还是一个紧密集成到 Windows Vista 的标准，在 Windows Vista 中

可以直接打印 XPS 文档。从本质上来说它也是一个压缩文件包，按照标准的 Zip 格式将图像和字体等文件打成包。图 18-25 所示为通过 WinRAR 软件打开的一个 XPS 文档，可以看到其内部组成。如果需要详细了解 XPS 文档的格式规范，可以访问 <http://msdn.microsoft.com/msdnmag/issues/06/01XMLPaper Specification>。

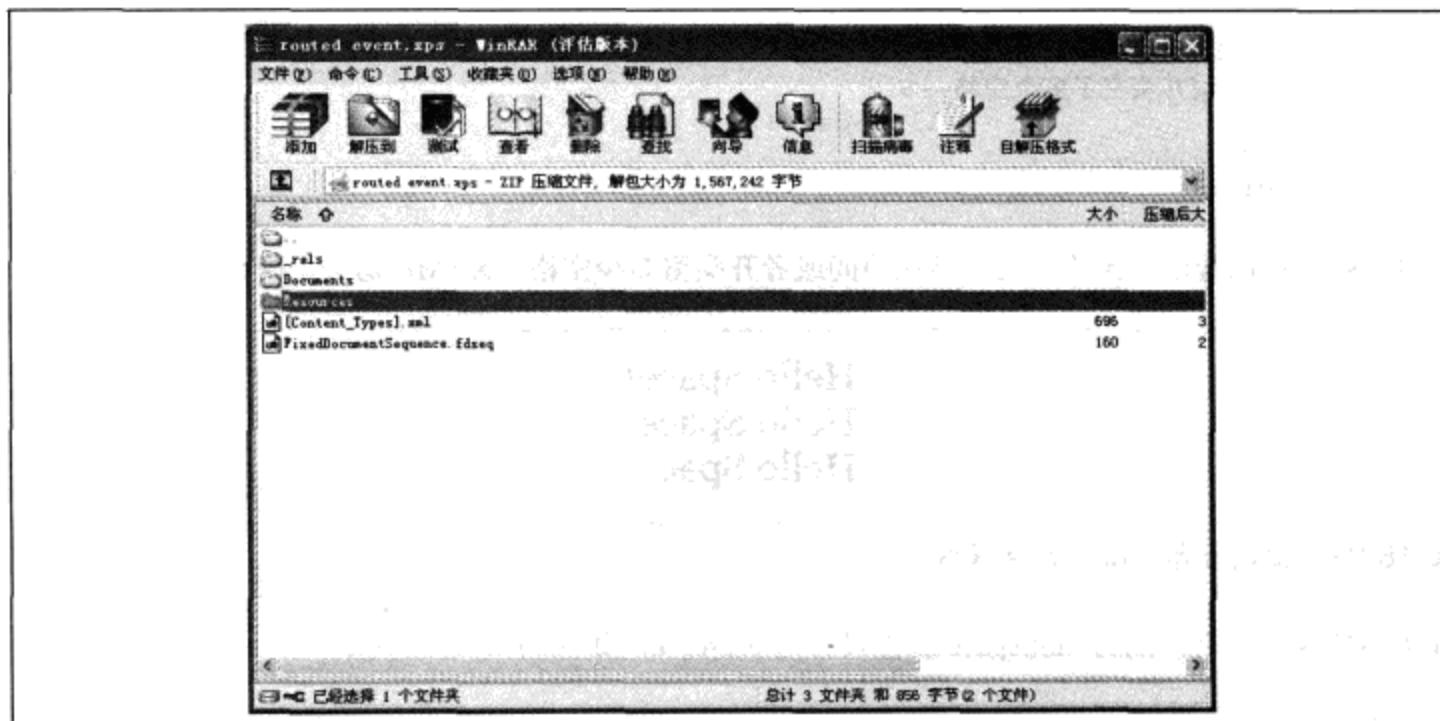


图 18-25 XPS 文件的内部组成

18.4 文档控件

WPF 为固定文档提供了一个浏览控件 DocumentViewer，为流文档提供了 3 种不同层次的浏览控件。如果希望修改流文档，则需要使用 RichTextBox 控件。

18.4.1 固定文档的浏览控件

DocumentViewer 控件不仅可以浏览 XPS 文档，还提供了查找和缩放的功能，下面是一个该控件的应用程序：

```
<Window x:Class="mumu_documentviewer.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="固定文档浏览器" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Menu Grid.Row="0">
            <MenuItem Header="文件">
                <MenuItem Header="打开..." Command="Open" />
                <MenuItem Header="关闭" Command="Close" />
```

```
</MenuItem>
</Menu>
<DocumentViewer Grid.Row="1" Name="docViewer" />
</Grid>
</Window>
```

代码 18-27 MainWindow.xaml 文件

```
public MainWindow()
{
    InitializeComponent();
    AddCommandBindings(ApplicationCommands.Open, OpenCommandHandler);
    AddCommandBindings(ApplicationCommands.Close, CloseCommandHandler);
}

private void OpenCommandHandler(
    object sender, ExecutedRoutedEventArgs e)
{
    // 弹出一个对话框
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.InitialDirectory = Directory.GetCurrentDirectory();
    dlg.Filter = "Xps Documents (*.xps)|*.xps";
    dlg.FilterIndex = 1;
    if (dlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        if (_xpsDocument != null)
            _xpsDocument.Close();
        try
        {
            _xpsDocument = new
                XpsDocument(dlg.FileName, System.IO.FileAccess.Read);
        }
        catch (UnauthorizedAccessException)
        {
            System.Windows.MessageBox.Show(
                String.Format("Unable to access {0}", dlg.FileName));
            return;
        }

        docViewer.Document = _xpsDocument.GetFixedDocumentSequence();
        _fileName = dlg.FileName;
    }
}

private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
    this.Close();
}

private void AddCommandBindings(ICommand command,
ExecutedRoutedEventHandler handler)
{
    CommandBinding cmdBindings = new CommandBinding(command);
    cmdBindings.Executed += handler;
    CommandBindings.Add(cmdBindings);
}

private XpsDocument _xpsDocument;
private string _fileName;
}
```

代码 18-28 MainWindow.xaml.cs 文件

在其中可以看到将一个固定文档加载到内存中包括如下两个步骤。

(1) 新建一个 XpsDocument 对象。

(2) 通过 GetFixedDocumentSequence 方法赋给 DocumentViewer 的 Document 属性。

程序运行结果如图 18-26 所示。

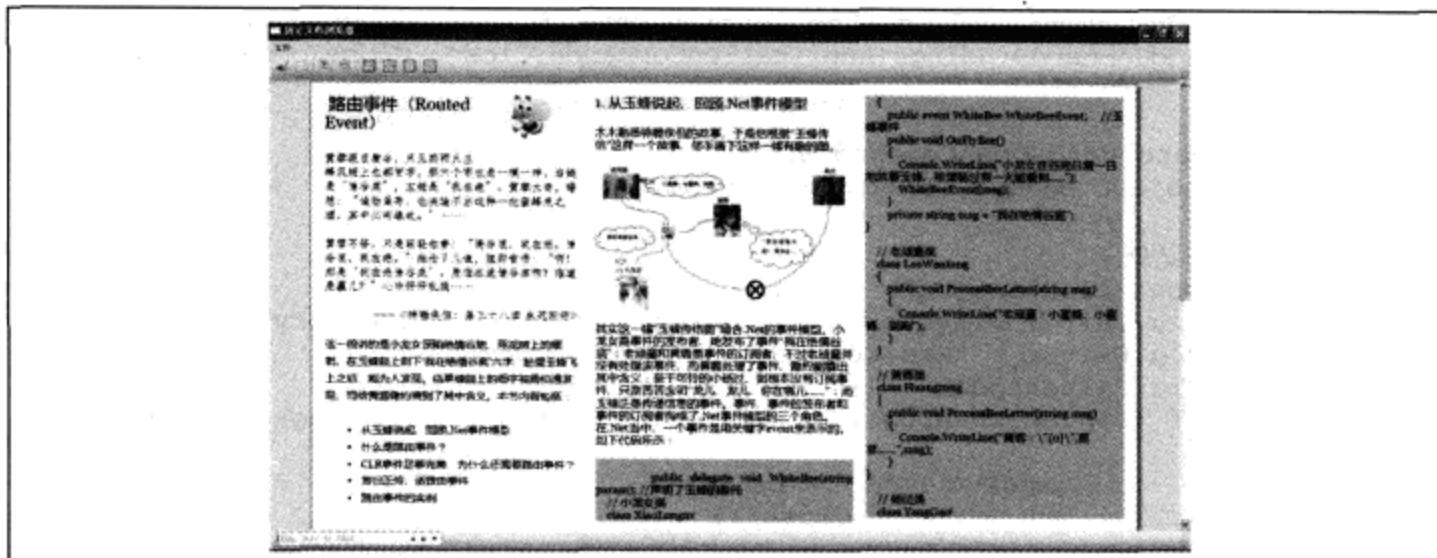


图 18-26 固定文档浏览器

18.4.2 流文档的浏览控件

由于流文档派生自 FrameContentElement，因此无法自我渲染。WPF 为流文档提供的 3 种不同的浏览控件如下。

(1) **FlowDocumentReader**: 在 3 种控件中功能最强，开销也最大。它提供 3 种不同的浏览方式，即单页模式 (Page Mode)、双页模式 (Two Page Mode) 和滚动模式 (Scroll Mode)，如图 18-27 所示。

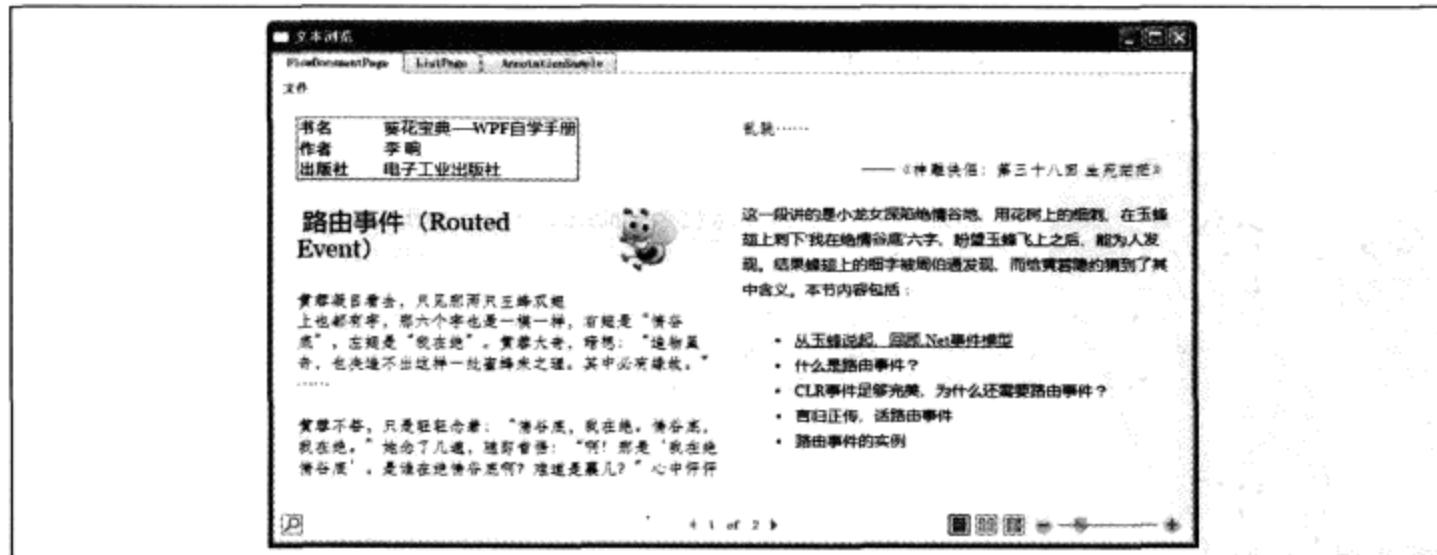


图 18-27 单页模式 (Page Mode)

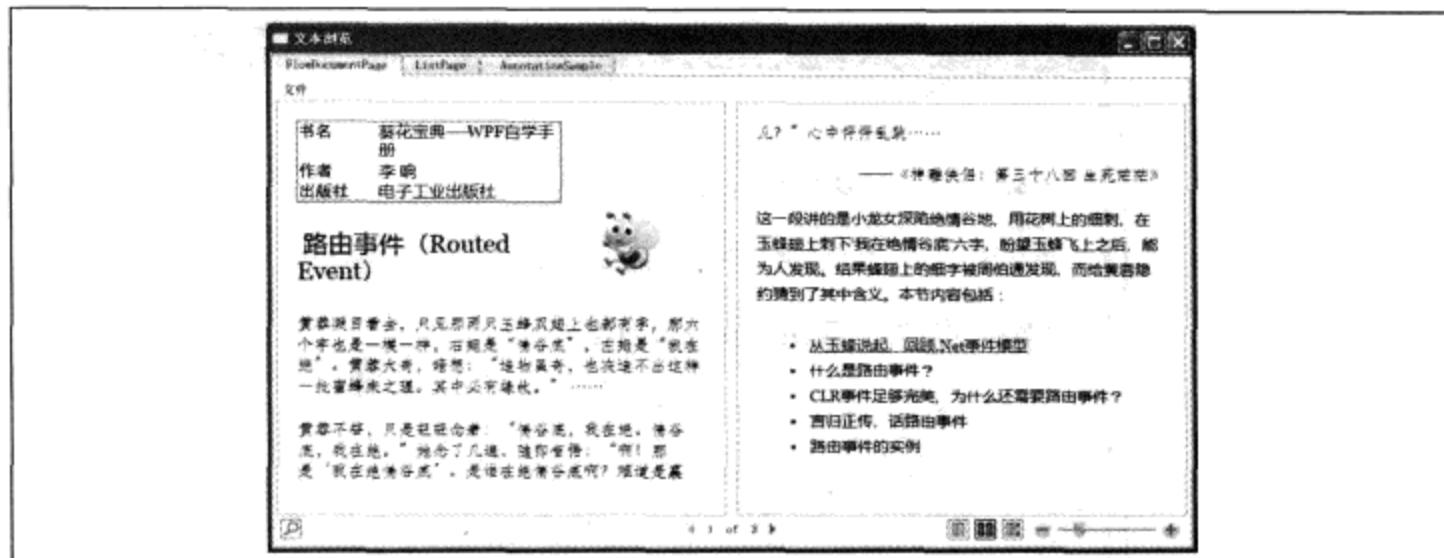


图 18-28 双页模式 (Two Page Mode)

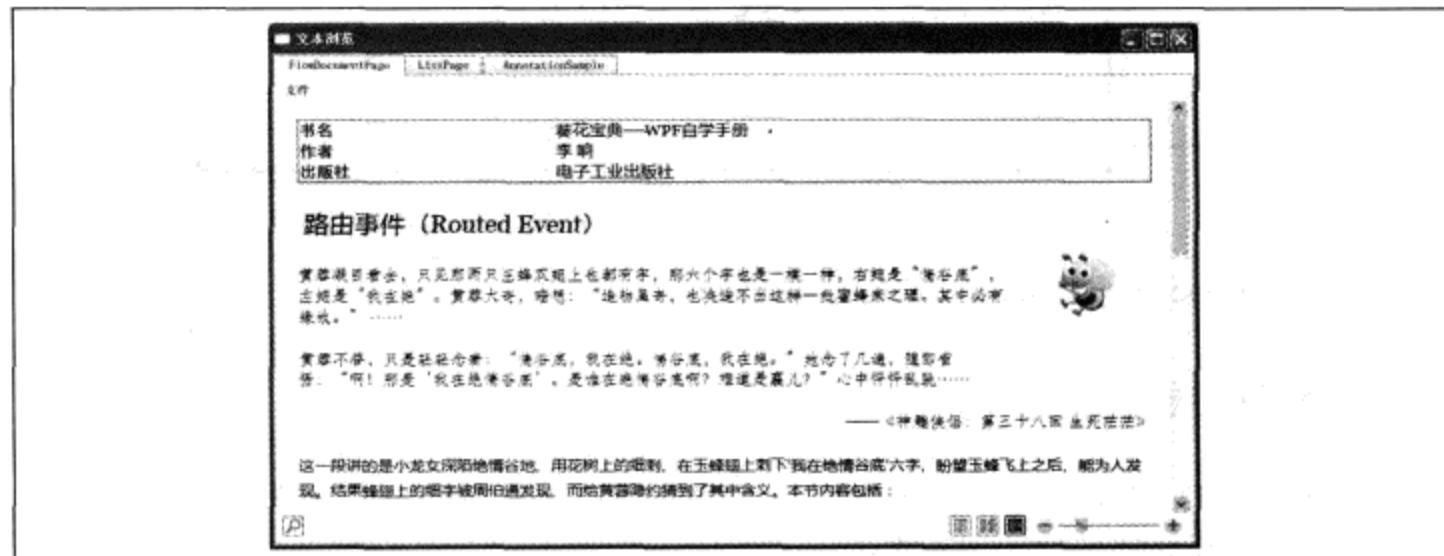


图 18-29 滚动模式 (Scroll Mode)

(2) **FlowDocumentScrollView**: 显示整个文档时采用滚动模式, 当文档尺寸超出自身的范围时可以使用滚动条移动文档。它是 3 种控件中开销最小的控件, 因为不必将流文档分为多页, 避免了将内容分配到页面中的额外计算。

(3) **FlowDocumentPageViewer**: 使用单页模式用户可以从一页进到下一页, 是 3 种控件中开销中等的控件。

一个标准的 FlowDocumentReader 浏览器主要由客户区和工具栏组成, 客户区用来显示文档的内容; 工具栏定义显示的工具, 包括缩放比例和浏览模式按钮, 以及导航和查找工具。FlowDocumentPageViewer 浏览器则少了浏览模式按钮(通过按 **Ctrl+F** 快捷键可以显示查找工具)。而 FlowDocumentScrollView 默认没有工具栏, 如果希望显示工具栏, 则将其 **IsToolBarVisible** 属性设置为 **true**。显示的工具栏中只有缩放比例按钮和查找工具。

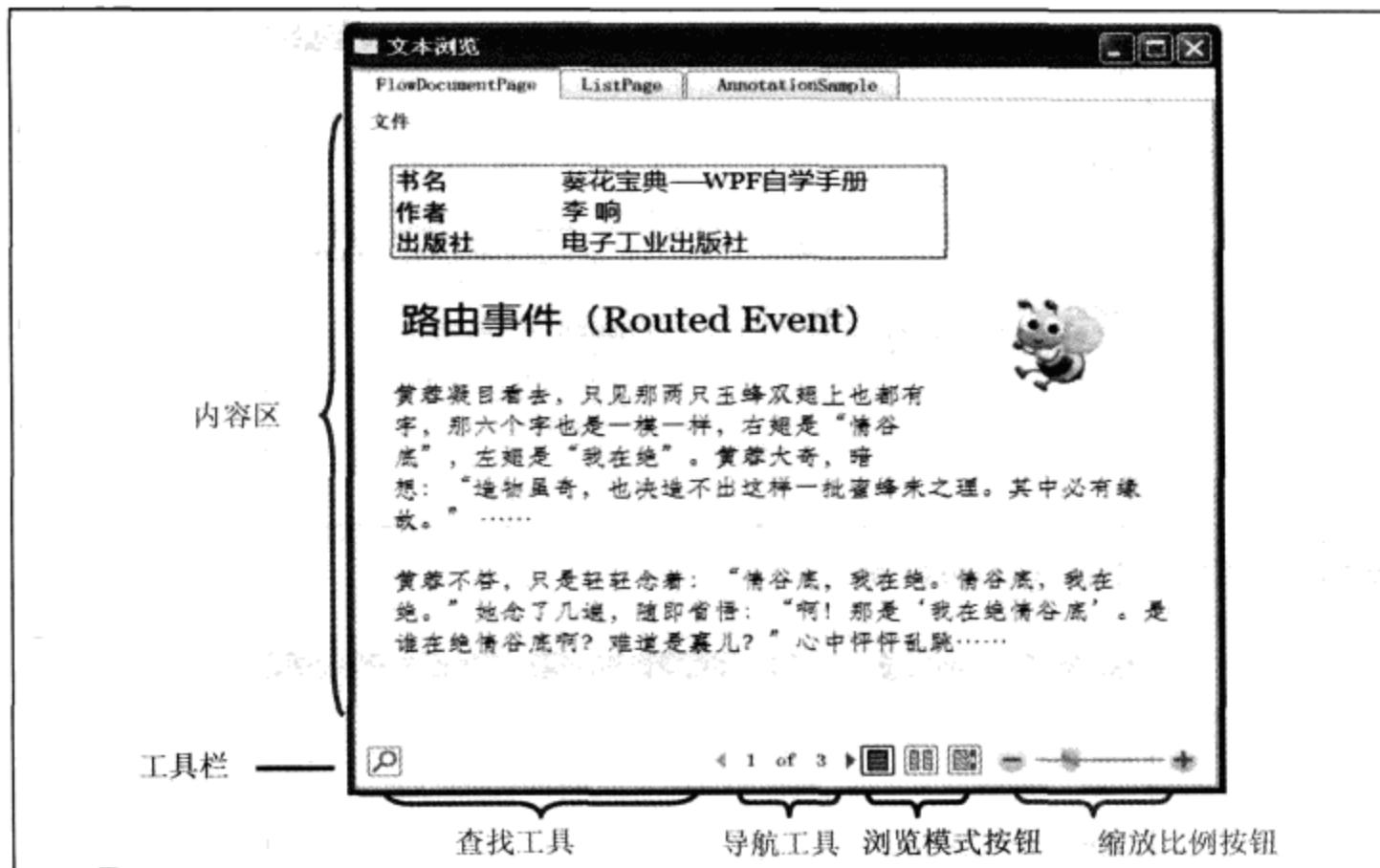


图 18-30 FlowDocumentReader 浏览器的组成部分

18.4.3 注释功能

FlowDocument 和 FixedDocument 的浏览器均支持注释 (annotation) 功能，即用户为了重点说明而添加的内容。WPF 提供如下 3 种注释形式。

- (1) 高亮显示。
- (2) 标签：在其中可以添加文本注释。
- (3) 标签：在其中通过手写形式添加注释。

图 18-31 所示为 3 种形式的注释。

通过命令形式为文本控件添加注释是最简单的方法，FlowDocument 和 FixedDocument 的浏览器均内置对这些命令的支持，这些命令在 System.Windows.Annotations 命名空间中。一般 XAML 的默认命名空间 <http://schemas.microsoft.com/winfx/2006/xaml/presentation> 并不包括该命名空间，因此使用注释时需要单独声明。详细的命令如表 18-5 所示。

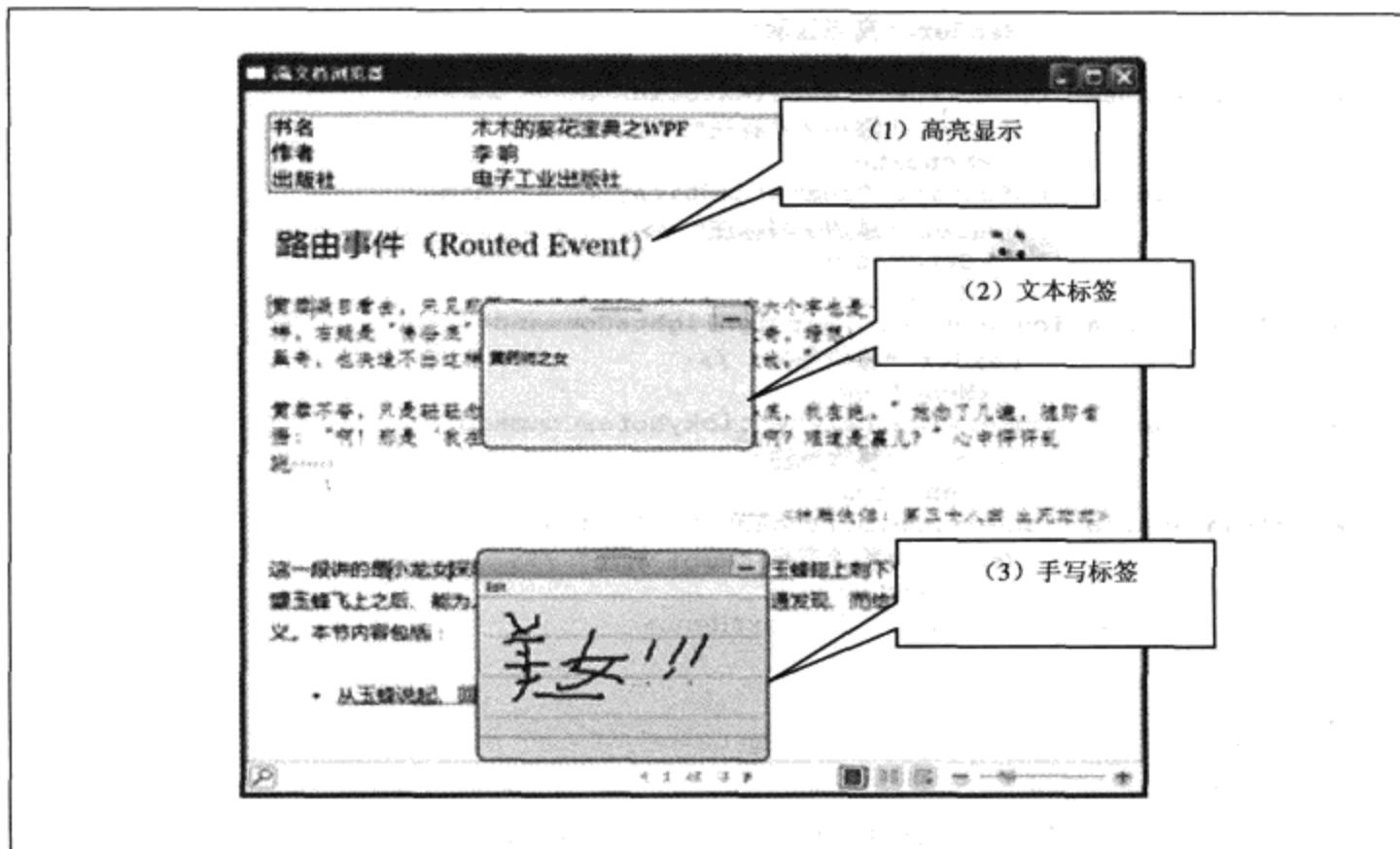


图 18-31 WPF 提供的三种形式的注释

表 18-5 文本标注的命令

命名	描述
CreateHighlightCommand	高亮选中的文本
ClearHighlightsCommand	清除选中的文本的高亮显示
CreateTextStickyNoteCommand	为选中的文本创建一个文本标签
CreateInkStickyNoteCommand	为选中的文本创建一个手写标签
DeleteStickyNotesCommand	删除当前选中的标签
DeleteAnnotationsCommand	移除当前选中的文本高亮和标签

为前面的流文档添加注释的代码如下，注意在 XAML 的根目录下声明 System.Windows.Annotations。

```

<Window x:Class="mumu_annotationsample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:ann="clr-namespace:System.Windows.Annotations;assembly=
PresentationFramework"
        Title="流文档浏览器" Height="300" Width="300"
        Initialized="Window_Initialized" Closed="Window_Closed">
    <Grid>
        <FlowDocumentReader x:Name="reader">
            <FlowDocument>
                <FlowDocument.ContextMenu>
                    <ContextMenu>
                        <MenuItem
Command="ann:AnnotationService.CreateHighlightCommand"

```

```

        Header="高亮显示" />
        <MenuItem
Command="ann:AnnotationService.CreateTextStickyNoteCommand"
        Header="添加文本标注" />
        <MenuItem
Command="ann:AnnotationService.CreateInkStickyNoteCommand"
        Header="添加手写标注" />
        <Separator />
        <MenuItem
Command="ann:AnnotationService.ClearHighlightsCommand"
        Header="移除高亮" />
        <MenuItem
Command="ann:AnnotationService.DeleteStickyNotesCommand"
        Header="移除标注" />
        <MenuItem
Command="ann:AnnotationService.DeleteAnnotationsCommand"
        Header="移除高亮 & 标注" />
        </ContextMenu>
    </FlowDocument.ContextMenu>
    .....
</FlowDocument>
</FlowDocumentReader>
</Grid>
</Window>

```

代码 18-29 为流文档添加注释功能

添加命令之后需要在 Windows 的 Initialized 和 Closed 事件处理函数中启用或禁用与 FlowDocumentReader 关联的注释服务（AnnotationService），代码如下：

```

private void Window_Initialized(object sender, EventArgs e)
{
    AnnotationService service = AnnotationService.GetService(reader);
    if (service == null)
    {
        stream = new FileStream("storage.xml", FileMode.OpenOrCreate);
        service = new AnnotationService(reader);
        AnnotationStore store = new XmlStreamStore(stream);
        service.Enable(store);
    }
}

private void Window_Closed (object sender, EventArgs e)
{
    AnnotationService service = AnnotationService.GetService(reader);
    if (service != null && service.IsEnabled)
    {
        service.Store.Flush();
    service.Disable();
    stream.Close();
    }
}

```

代码 18-30 启用注释功能

在 Window_Initialized 函数中启用 AnnotationService 之前必须指定一个保存注释的文件流，本例是一个 xml 文件。这样每次应用程序关闭时保存所有的注释，在下次运行该程序时将显示。

18.5 实现一个简单的文档浏览器

在本节中我们将实现一个简单的文档浏览器，其功能如下。

- (1) 支持 XAML 格式的文件浏览。
- (2) 可以另存为 XPS、XAML 和 HTML 格式的文件。
- (3) 支持缩略图。
- (4) 支持书签。
- (5) 支持标注。

18.5.1 应用程序组成

我们构建的应用程序的顶部是菜单和工具栏，中间部分是一个流文档的浏览器 FlowDocumentPageViewer。左侧面板由 3 个 Tab 页组成，分别是缩略图、标注和书签，如图 18-32 所示。

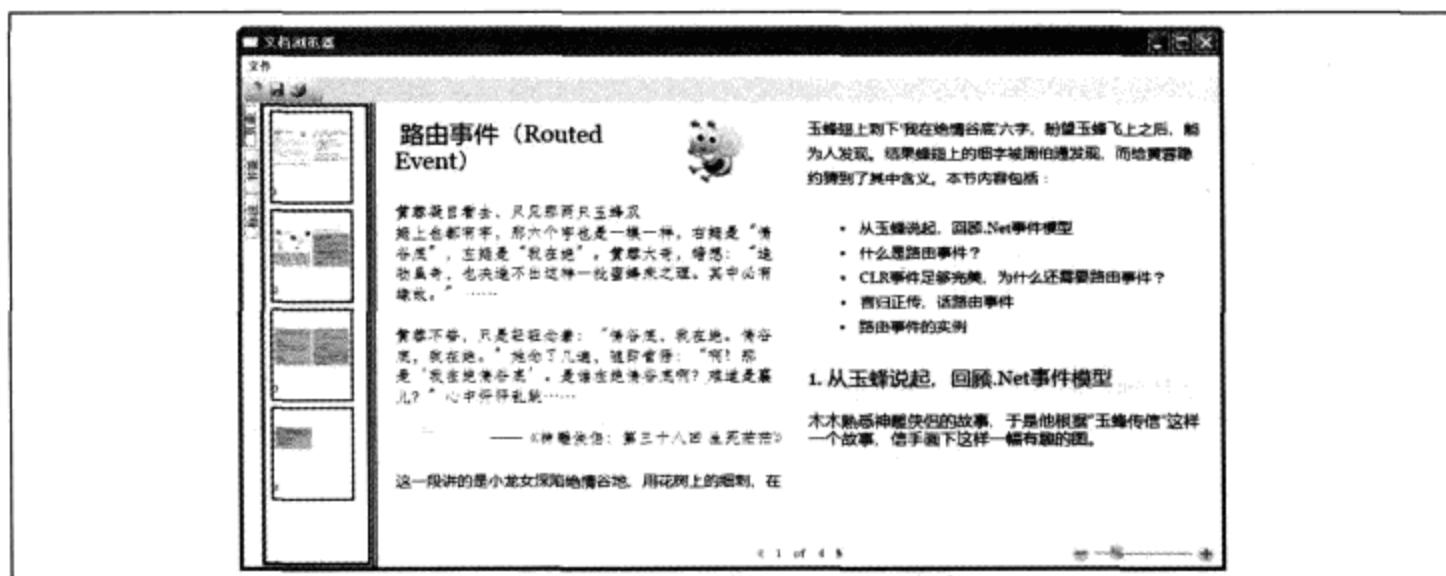


图 18-32 文档浏览器

代码如下：

```
<Window x:Class="mumu_documentreader.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:mumu_documentreader"
    Title="文档浏览器" >
    <Grid Name="MainGrid">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="125" />
            <ColumnDefinition Width="5" />
            <ColumnDefinition Width="*" />
```

```

</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
<!--菜单,占据 Grid 的第 1 行-->
<Menu Grid.Row="0" Grid.ColumnSpan="3" >
    .....
</Menu>
<!--工具栏,占据 Grid 的第 2 行-->
<ToolBarTray IsLocked="True" Grid.Row="1" Grid.ColumnSpan="3" >
    <ToolBar DockPanel.Dock="Top" >
        .....
    </ToolBar>
</ToolBarTray>
<!--Tab 控件,占据 Grid 的第 2 行, 左侧第 1 列-->
<TabControl Grid.Row="2" Grid.Column="0" TabStripPlacement="Left"
Margin="0" Padding="0" Name="LeftTabControl">
    <TabItem Name="Pages" VerticalAlignment="Top" HorizontalAlignment="Left"
Padding="0, 10">
        .....
    </TabItem>
    <TabItem Name="Bookmarks" VerticalAlignment="Top" HorizontalAlignment="Left"
Padding="0, 10">
        .....
    </TabItem>
    <TabItem Name="Comments" VerticalAlignment="Top" HorizontalAlignment="Left"
Padding="0, 10">
        .....
    </TabItem>
</TabControl>

<GridSplitter Grid.Row="2" Grid.Column="1" ResizeDirection="Columns"
DragCompleted="SplitterEndResize" BorderBrush="Blue" BorderThickness="1"
HorizontalAlignment="Stretch" Background="{DynamicResource {x:Static
SystemColors.ControlDarkBrushKey}}}">
</GridSplitter>
<!--文档浏览器控件,占据 Grid 的第 2 行, 左侧第 3 列-->
<FlowDocumentPageViewer x:Name="FDPV" Grid.Row="2" Grid.Column="2">
    </FlowDocumentPageViewer>
</Grid>
</Window>

```

代码 18-31 文档浏览器界面程序

首先需要能够打开和并显示一个流文档文件 (*.xaml)，一个流文档文件的根元素是 FlowDocument。可以将原来内嵌在控件的文档直接变为一个单独的文件，注意需要使用“`xml:space='preserve'`”保留文档中代码之间的空格。

18.5.2 打开一个流文档

使用 Command 实现打开流文档的功能，首先为 ApplicationCommands.Open 命令添加如下命令处理函数：

```
<Window.CommandBindings>
```

```

<CommandBinding Executed="OnOpen" Command="ApplicationCommands.Open"/>
</Window.CommandBindings>
.....
<Menu Grid.Row="0" Grid.ColumnSpan="3" >
    <MenuItem Header="文件">
        <MenuItem Header="打开" Command="ApplicationCommands.Open">
            <MenuItem.Icon>
                <Image Source="Images\FileOpen.png" />
            </MenuItem.Icon>
        </MenuItem>
    </MenuItem>
</Menu>
.....
<ToolBarTray IsLocked="True" Grid.Row="1" Grid.ColumnSpan="3" >
    <ToolBar >
        <Button ToolTip="打开文件" Command="ApplicationCommands.Open">
            <Image Source="Images\FileOpen.png" />
        </Button>
    </ToolBar>
</ToolBarTray>

```

代码 18-32 给 ApplicationCommands.Open 命令添加一个命令处理函数

OnOpen 函数会直接调用窗口的成员方法 OpenDocument，具体实现如下：

```

private void OnOpen(object sender, ExecutedRoutedEventArgs e)
{
    MainWindow win = (MainWindow)sender;

    if (win.OpenDocument())
    {
        // 如果打开文档成功，则有后续工作.....
        // .....
    }
}

public bool OpenDocument()
{
    // 如果有文档，则需要关闭之后打开
    ② if (FDPV.Document != null) CloseFile();

    // 弹出一个打开对话框
    Microsoft.Win32.OpenFileDialog dialog;
    dialog = new Microsoft.Win32.OpenFileDialog();
    dialog.CheckFileExists = true;
    dialog.InitialDirectory = Directory.GetCurrentDirectory();
    dialog.Filter = xamlfileFilter;
    bool result = (bool)dialog.ShowDialog(null);
    if (result == false) return false;
    string fileName = dialog.FileName;

    object newDocument = null;
    try
    {
        if (fileName.EndsWith(".xaml"))
        {
            using (FileStream inputStream = File.OpenRead(fileName))
            {
                ParserContext pc = new ParserContext();
                pc.BaseUri =
                    new Uri(System.Environment.CurrentDirectory

```

```

+ "/");
③
        // 如果此处未传递 ParserContext，则不显示外面的两张图片
        newDocument = XamlReader.Load(inputStream, pc)
        as object;

        if (newDocument == null)
        {
            MessageBox.Show(
                "无法解释该 XAML 文件" +
                fileName, "错误提示",
                MessageBoxButton.OK,
                MessageBoxIcon.Error);
            return false;
        }
    }
}
catch (Exception e)
{
    MessageBox.Show(
        "该文件可能不是正确的流文档格式：" +
        fileName + "\n", "错误提示",
        MessageBoxButton.OK, MessageBoxIcon.Error);
    return false;
}

if (newDocument is IDocumentPaginatorSource)
{
    FDPV.Document = (IDocumentPaginatorSource)newDocument;
    _fileName = fileName;
}
else
{
    throw new InvalidDataException(
        "Thumbnail viewer only supports
    IDocumentPaginatorSource");
}
return true;
}

private void CloseFile()
{
    FDPV.Document = null;
}

```

代码 18-33 OnOpen 及其相关函数的实现

在代码①处打开文档之后实际上还需要一些后续工作，主要是在左侧面板中生成文档的缩略图；在代码②处，如果当前的文档没有关闭，则需要首先关闭。关闭文档实际上只是将 Document 赋为空值；在代码③处，通过 XamlReader 的 Load 方法将文件加载到内存中。参数中除了需要传递文件流，还需要传递第 2 个参数 ParserContext，用于让 XamlReader 能够加载与流文档相关的资源。示例数据中的两张图片文件是流文档的一个外部资源，与流文档在同一个目录下，代码如下所示：

```

.....
<Figure Width="100" Height="100" HorizontalAnchor="ColumnRight" HorizontalOffset="-10"
VerticalAnchor="ParagraphTop" VerticalOffset="-30">
    <BlockUIContainer>

```

```

<Image Source="bee.png"/>
</BlockUIContainer>
</Figure>
.....
<BlockUIContainer>
<Image Source="routedevent.jpg"/>
</BlockUIContainer>

```

代码 18-34 示例数据

如果仅仅传递了文件流，而未传递上下文设备环境信息，则打开该文件时无法加载这两张图片。图 18-33 所示为两种不同加载方法的对比。

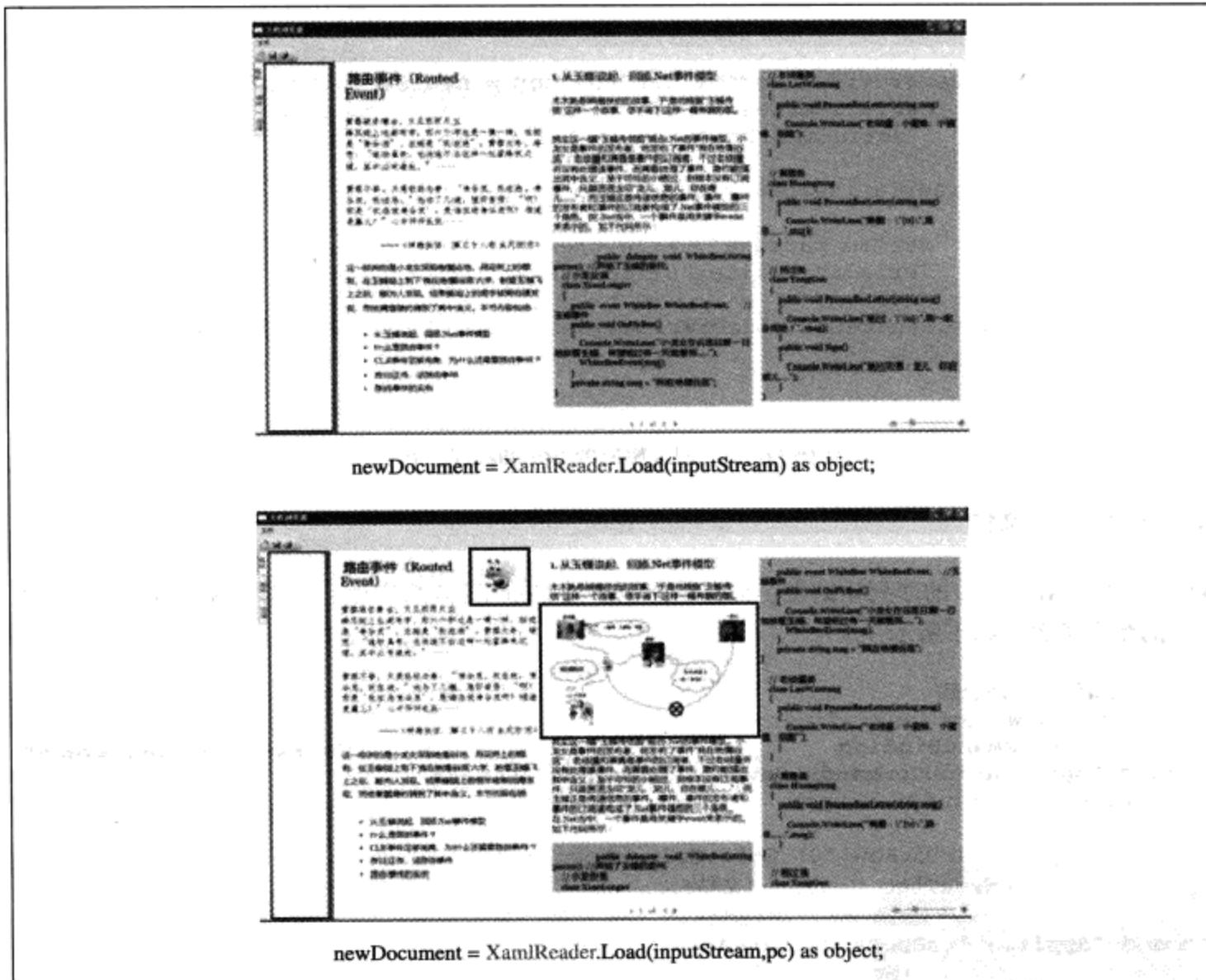


图 18-33 两种不同加载方法的对比

18.5.3 另存为不同格式的文件

如果希望流文件另存为不同格式的文件，需要使用 System.Windows.Documents.Serialization 命名空间中序列化的相关类。序列化即将内存中的数据作为文件存储在硬盘中，WPF 中的序列化通过 SerializerWriter 实现。SerializerWriter 可以由 WPF 提供，也可以由第三方来实现。WPF 内置的

SerializerWriter 可以将文件写为 XPS 格式，但是 SerializerWriter 不能直接获取，需要通过 SerializerProvider 发现。

1. 发现 SerializerWriter

通过 SerializerProvider 的发现机制查找系统中提供的 SerializerWriter 数量，其基本原理是 SerializerProvider 提供了一个 InstalledSerializers 属性，它是一个 SerializerDescriptor 集合，其中包括所有已安装的 SerializerWriter 的信息集合。该集合描述了不同的 SerializerWriter 的信息，如文件的扩展名等。SerializerProvider 根据 SerializerDescriptor 这些信息创建相应的 SerializerWriter，代码如下：

```
....  
    SerializerProvider serializerProvider = new SerializerProvider();  
    foreach (SerializerDescriptor serializerDescriptor in  
            serializerProvider.InstalledSerializers)  
    {  
        if (serializerDescriptor.IsLoadable)  
        {  
            Stream package = File.Create(fileName);  
            SerializerWriter serializerWriter =  
                serializerProvider.CreateSerializerWriter(serializerDescriptor, package);  
            ....  
        }  
    }  
}
```

代码 18-35 创建相应的 SerializerWriter

2. 另存为 XPS 文件

WPF 提供了 XPS 文件的序列化的类，另存功能也通过 Command 来实现。首先为 SaveAs 命令添加相应的命令函数，如下所示：

```
MainWindow.xaml  
    <Window.CommandBindings>  
        <CommandBinding  
            Command="ApplicationCommands.SaveAs" Executed="SaveAs"/>  
    </Window.CommandBindings>  
    ....  
        <Menu Grid.Row="0" Grid.ColumnSpan="3" >  
            <MenuItem Header="文件">  
                <MenuItem Header="另存为 ..... " x:Name="SaveAsMenu"  
                    Command="ApplicationCommands.SaveAs">  
                    <MenuItem.Icon>  
                        <Image Source="Images\filesave.png" />  
                    </MenuItem.Icon>  
                </MenuItem>  
            </Menu>  
            <ToolBarTray IsLocked="True" Grid.Row="1" Grid.ColumnSpan="3" >  
                <ToolBar DockPanel.Dock="Top" >  
                    <Button ToolTip="Save File As..... " x:Name="SaveAsToolbarButton"  
                        Command="ApplicationCommands.SaveAs">  
                        <Image Source="Images\FileSave.png" />  
                    </Button>
```

```
</ToolBar>  
</ToolBarTray>
```

代码 18-36 为菜单和工具栏关联 SaveAs 命名

其中 CanSaveAs 用于判断“另存为”功能何时可用，只有文档的 Document 属性不为空时才可用；否则不可用。SaveAs 函数会将当前文件另存为不同格式的文件，代码如下：

```
MainWindow.xaml.cs  
private void CanSaveAs(object sender, CanExecuteRoutedEventArgs e)  
{  
    MainWindow win = (MainWindow)sender;  
    if (win == null) return;  
    if (win.FDPV == null) return;  
    if (win.FDPV.Document != null)  
        e.CanExecute = true;  
}  
  
① private string PlugInFileFilter  
{  
    get  
    { // 创建一个 SerializerProvider  
        SerializerProvider serializerProvider = new  
            SerializerProvider();  
        string filter = "*.*";  
        //  
        foreach (SerializerDescriptor serializerDescriptor in  
            serializerProvider.InstalledSerializers)  
        {  
            if (serializerDescriptor.IsLoadable)  
            {  
                if (filter.Length > 0) filter += "|";  
                filter += serializerDescriptor.DisplayName + "(*" +  
                    serializerDescriptor.DefaultFileExtension + ")|*" +  
                    serializerDescriptor.DefaultFileExtension;  
            }  
        }  
        return filter;  
    }  
}  
private void SaveAs(object sender, ExecutedRoutedEventArgs e)  
{  
    SaveDocumentAsFile(null);  
}  
  
public bool SaveDocumentAsFile(string fileName)  
{  
    // 如果文件名为空，则需要弹出一个 Save 对话框  
    if (fileName == null)  
    {  
        // 创建保存对话框  
        Microsoft.Win32.SaveFileDialog dialog;  
        dialog = new Microsoft.Win32.SaveFileDialog();  
        dialog.CheckFileExists = false;  
        dialog.Filter = this.PlugInFileFilter;  
  
        ② bool result = (bool)dialog.ShowDialog(null);  
        if (result == false) return false;  
        fileName = dialog.FileName;  
    }  
}
```

```

        return SaveToFile(fileName);
    }

    private bool SaveToFile(string fileName)
    {
        if (fileName == null) throw new ArgumentNullException("文
件名为空");
        if (File.Exists(fileName)) File.Delete(fileName);

        FlowDocument flowDocument = FDPV.Document as FlowDocument;
        try
        {
            SerializerProvider serializerProvider = new
                SerializerProvider();
            SerializerDescriptor selectedPlugIn = null;
            foreach (SerializerDescriptor serializerDescriptor in
                serializerProvider.Installed
            Serializers)
            {
                // 如果 IsLoadable 属性为真，且后缀名一致
                if (serializerDescriptor.IsLoadable &&
                    fileName.EndsWith(serializerDescriptor.Default
FileExtension))
                {
                    selectedPlugIn = serializerDescriptor;
                    break;
                }
            }

            if (selectedPlugIn != null)
            {
                Stream package = File.Create(fileName);
                SerializerWriter serializerWriter =
                    serializerProvider.CreateSerializer
Writer(selectedPlugIn, package);
                IDocumentPaginatorSource idoc =
                    flowDocument as IDocumentPaginatorSource;
                serializerWriter.Write(idoc.DocumentPaginator, null);
                package.Close();
                return true;
            }
        }
        catch (Exception e)
        {
            MessageBox.Show(
                "格式转换出错" +
                fileName + "\n", "错误提示",
                MessageBoxButton.OK, MessageBoxIcon.Error);
            return false;
        }
        return true;
    }
}

```

代码 18-37 SaveAs 和 CanSaveAs 函数实现

PlugInFileFilter 的实现在代码①处；在代码②处需要弹出一个保存对话框供用户选择要另存为文件的类型，该类型通过 SaveFileDialog 的 Filter 属性来过滤。Filter 属性被赋值为 PlugInFileFilter，这是一个字符串，通过遍历 SerializerProvider 提供的 SerializerDescriptor 来构建；在代码③处，

serializerProvider 根据相应的 SerializerDescriptor 创建 SerializerWriter，然后通过 Write 方法将内存中的数据序列化。

运行程序，可以看到目前只支持保存为 XPS 文件，如图 18-34 所示。

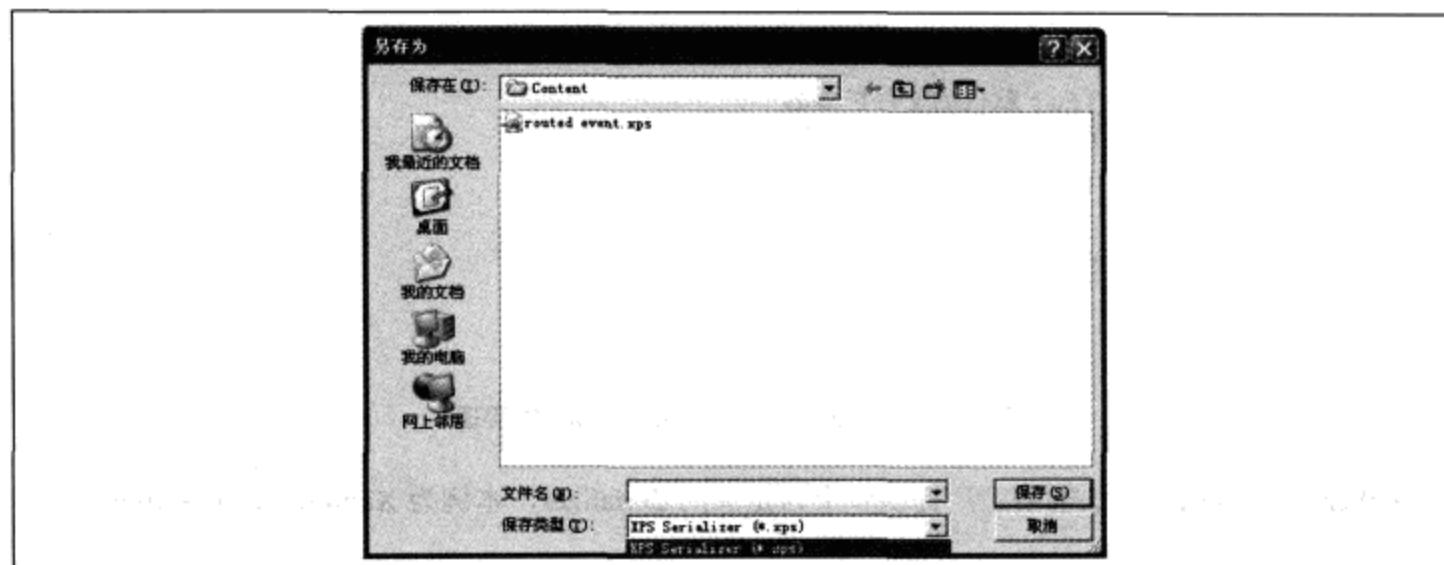


图 18-34 只支持 XPS 文件的存储

3. 自定义 SerializerWriter

如果要保存为不同格式的文件，则需要自定义一个 SerializerWriter。以保存为 XAML 文件格式为例自定义一个 SerializerWriter，步骤如下。

(1) 实现一个 ISerializerFactory 接口的类，该类需要提供文件的扩展名等属性，下面是 XamlSerializerFactory 的实现代码：

```
class XamlSerializerFactory : ISerializerFactory
{
    // 创建一个 serializer
    public SerializerWriter CreateSerializerWriter(Stream stream)
    {
        return new XamlSerializerWriter(stream);
    }

    // serializer 显示的名称
    public string DisplayName
    {
        get
        {
            return "Xaml Document Writer";
        }
    }

    // serializer 制造商的名称
    public string ManufacturerName
    {
        get
        {
            return "Microsoft";
        }
    }
}
```

```

    }
    // serializer 制造商的网站
    public Uri ManufacturerWebsite
    {
        get
        {
            return new Uri("http://www.microsoft.com");
        }
    }
    // serializer 默认的文件扩展名
    public string DefaultFileExtension
    {
        get
        {
            return ".xaml";
        }
    }
}

```

代码 18-38 XamlSerializerFactory 的实现

CreateSerializerWriter 方法需要工厂提供一个自定义的 Serializer，本例为 XamlSerializerWriter。

(2) 需要实现一个 Serializer，该类派生自 SerializerWriter，需要全部实现 SerializerWriter 中的接口和事件。该接口主要是 Write 方法，但是有同步和异步（方法名为“Async”），以及不同参数之分，因此导致 SerializerWriter 的 Write 方法很多，我们可以有选择地实现一部分。对于目前无法支持的部分方法，则在函数中抛出一个无法支持的异常（NotSupportedException）。下面是 XamlSerializerWriter 的实现代码：

```

class XamlSerializerWriter:SerializerWriter
{
    public XamlSerializerWriter(Stream stream)
    {
        _stream = stream;
    }
    public override void Write(Visual visual)
    {
        Write(visual, null);
    }
    public override void Write(Visual visual, PrintTicket printTicket)
    {
        SerializeObjectTree(visual);
    }
    public override void WriteAsync(Visual visual)
    {
        throw new NotSupportedException();
    }
    .....
    public override void Write(DocumentPaginator documentPaginator)
    {
        Write(documentPaginator, null);
    }
    public override void Write(DocumentPaginator documentPaginator,
PrintTicket printTicket)
    {
        SerializeObjectTree(documentPaginator.Source);
    }
    public override void Write(FixedPage fixedPage)

```

```

        {
            Write(fixedPage, null);
        }
        public override void Write(FixedPage fixedPage, PrintTicket printTicket)
        {
            SerializeObjectTree(fixedPage);
        }

        public override void WriteAsync(FixedPage fixedPage)
        {
            throw new NotSupportedException();
        }

        public override void Write(FixedDocument fixedDocument)
        {
            Write(fixedDocument, null);
        }
        public override void Write(FixedDocument fixedDocument, PrintTicket
printTicket)
        {
            SerializeObjectTree(fixedDocument);
        }

        public override void Write(FixedDocumentSequence fixedDocumentSequence)
        {
            Write(fixedDocumentSequence, null);
        }
        public override void Write(FixedDocumentSequence fixedDocumentSequence,
PrintTicket printTicket)
        {
            SerializeObjectTree(fixedDocumentSequence);
        }

        public override event WritingPrintTicketRequiredEventHandler
WritingPrintTicketRequired;
        public override event WritingProgressChangedEventHandler
WritingProgressChanged;
        public override event WritingCompletedEventHandler WritingCompleted;
        public override event WritingCancelledEventHandler WritingCancelled;

        private void SerializeObjectTree(object objectTree)
        {
            TextWriter writer = new StreamWriter(_stream);
            XmlTextWriter xmlWriter = null;
            try
            {
                // 创建一个 xmlWriter
                xmlWriter = new XmlTextWriter(writer);

                XamlDesignerSerializationManager manager = new
XamlDesignerSerializationManager(xmlWriter);

                System.Windows.Markup.XamlWriter.Save(objectTree, manager);
            }
            finally
            {
                if (xmlWriter != null)
                    xmlWriter.Close();
            }
        }
        private Stream _stream;
    }
}

```

}

代码 18-39 XamlSerializerWriter 的实现

(3) 通过 SerializerProvider 注册新的序列化类，注意在应用程序退出时取消该序列类，代码如下：

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    try
    {
        SerializerProvider.RegisterSerializer(SerializerDescriptor.CreateFromFactory
Instance(new XamlSerializerFactory()), false);
    }
    catch (ArgumentException)
    {
    }
}

private void Application_Exit(object sender, ExitEventArgs e)
{
    try
    {
        SerializerProvider.UnregisterSerializer(SerializerDescriptor.CreateFromFacto
ryInstance(new XamlSerializerFactory()));
    }
    catch (ArgumentException)
    {
    }
}
```

代码 18-40 应用程序退出时取消注册序列类

运行程序，可以看到在“另存为”对话框中不仅可以保存 XPS 文件，也可以保存 XAML 文件，如图 18-35 所示。

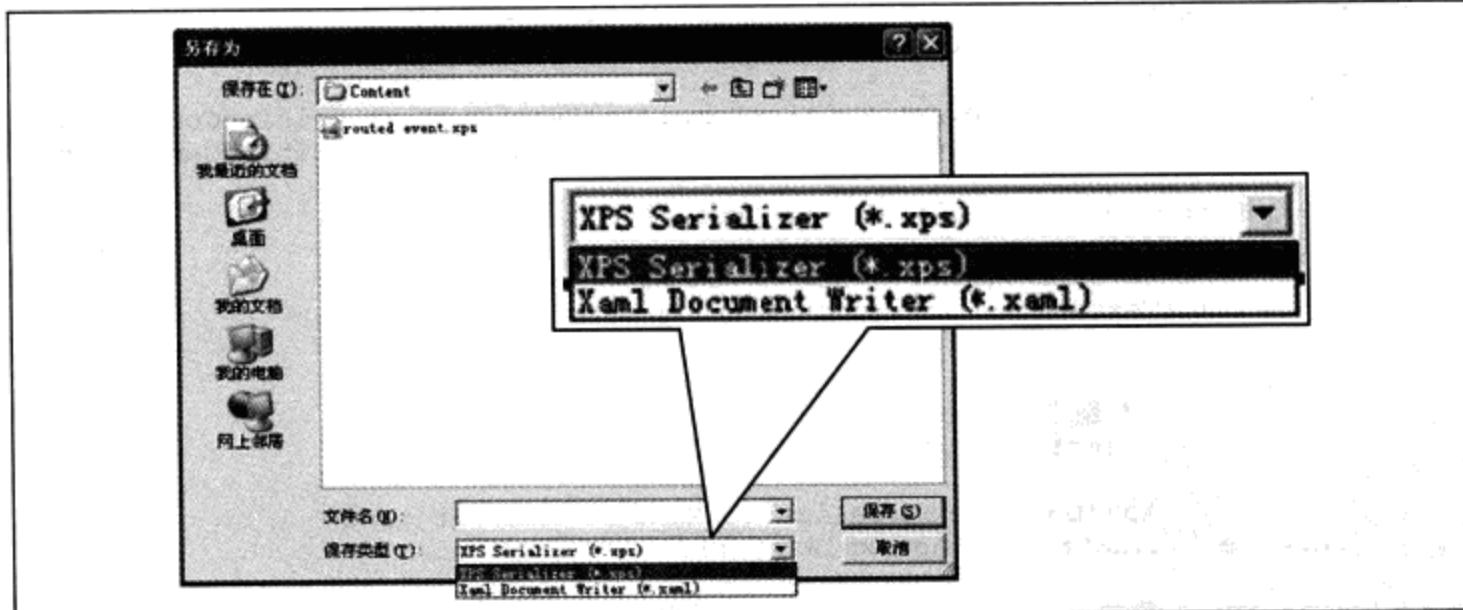


图 18-35 可存储不同格式的文件

以同样的方法可以实现另存为 HTML 格式文件的功能。

18.5.4 实现缩略图功能

1. 缩略图分页构成

我们希望在整个应用程序的左侧面板中生成文档的缩略图，以便于导航。当拖动 GridSplitter 时由于分页大小会发生变化，所以缩略图的排列也随之发生变化，如图 18-36 所示。

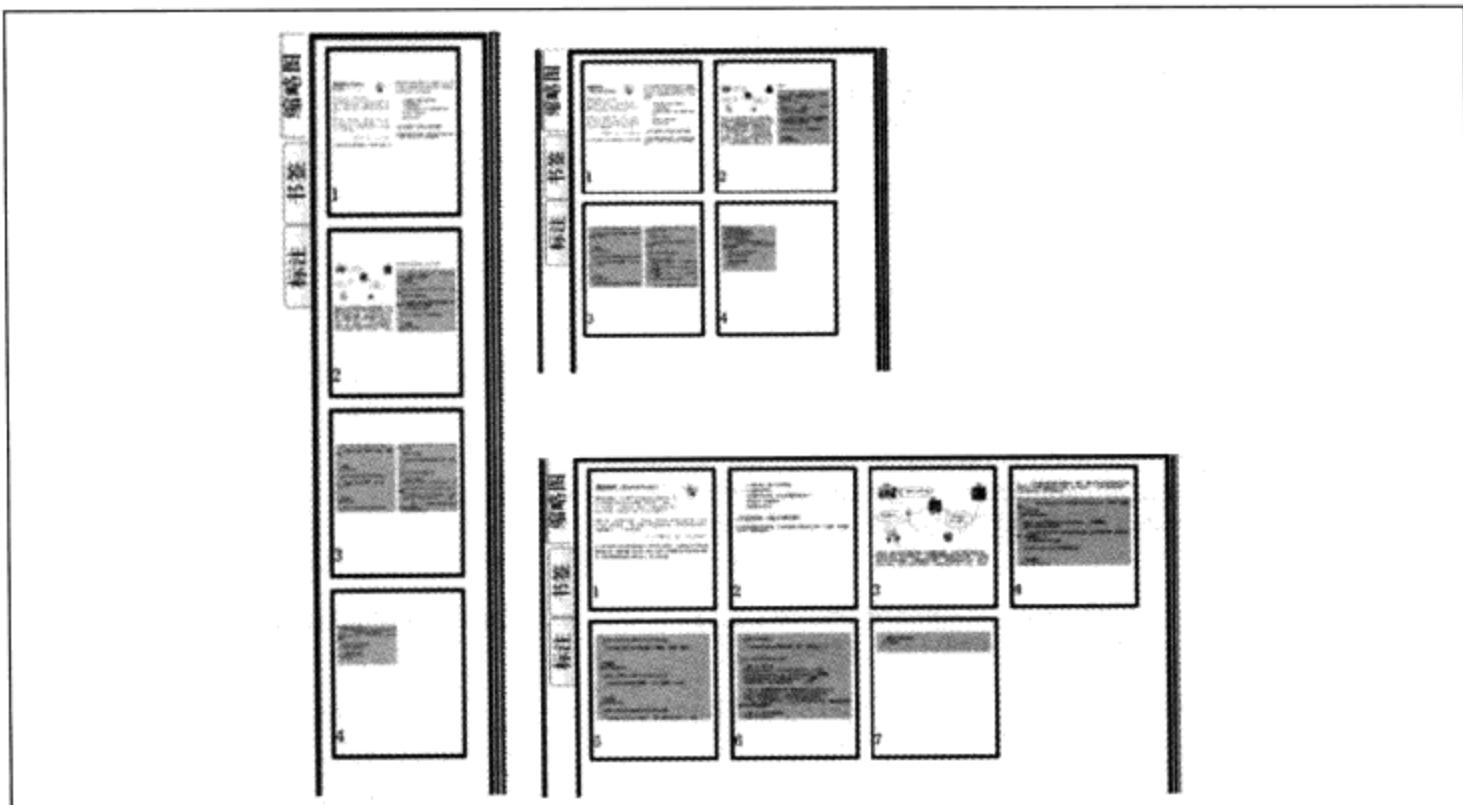


图 18-36 随分页大小变化的缩略图

缩略图集合需要通过 `ListBox` 来表现，缩略图随外部容器大小变化而均匀排列，这种特性使用 `WrapPanel` 实现最为简单。`ListBox` 的 `ItemsPanel` 属性为 `WrapPanel`，其中的每一个可视的 Item 由图 18-37 所示的部分组成。

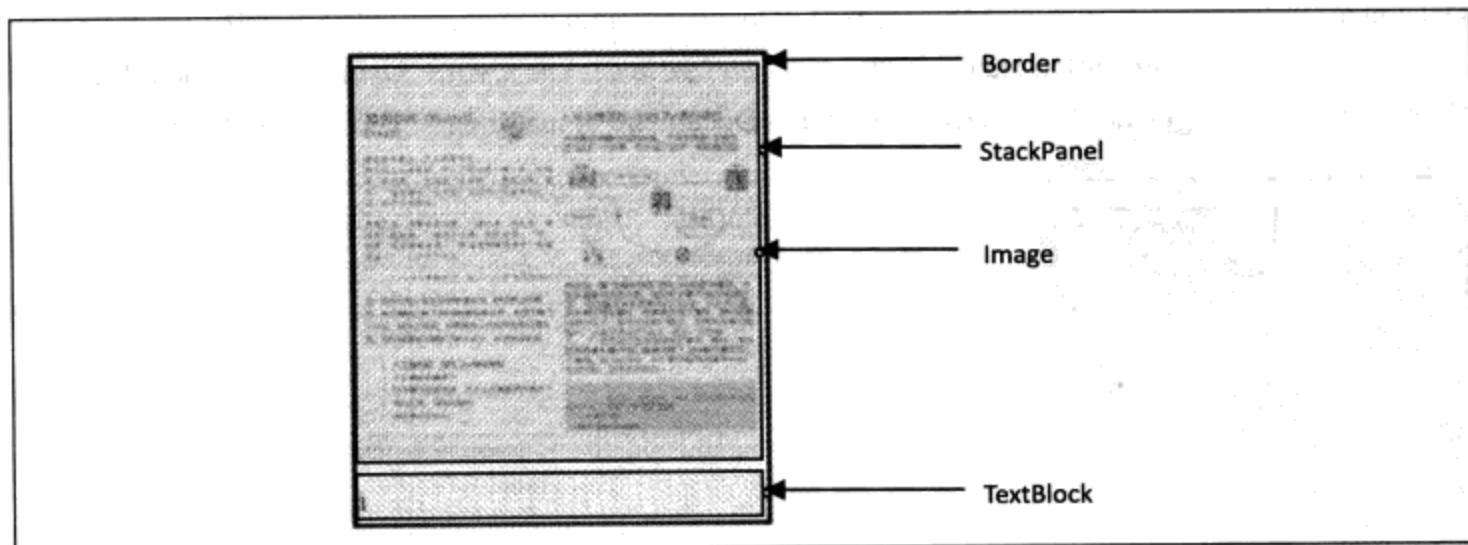


图 18-37 `ListBox` 当中可视的 Item 的组成

2. 创建和重排缩略图分页

在每次文档打开之后应该创建缩略图，因此在 OpenDocument 之后创建缩略图的分页。创建时会计算缩略图分页能够容纳的缩略图的个数（_maxThumbnails）和最大的显示个数（_numDisplayedThumbnails），代码如下：

```
private void OnOpen(object sender, ExecutedRoutedEventArgs e)
{
    MainWindow win = (MainWindow)sender;

    // 打开文档成功，则创建 Thumbs
    if (win.OpenDocument())
    {
        win.CreateThumbs();
    }
}

public void CreateThumbs()
{
    if ((ThumbList.Items != null) && (ThumbList.Items.Count > 0))
        ThumbList.Items.Clear();

    _currentThumbnail = 0;
    _maxThumbnails = CalculateMaxThumbnails();
    _numDisplayedThumbnails = (_maxThumbnails <= FDPV.PageCount) ?
        _maxThumbnails : FDPV.PageCount;

    for (int i = _currentThumbnail; i < _numDisplayedThumbnails; i++)
        AddPageThumb(i, true);

    for (int i = _numDisplayedThumbnails; i < FDPV.PageCount; i++)
        AddPageThumb(i, false);

    ThumbList.Items.MoveCurrentToPosition(_currentThumbnail);
    FDPV.Focus();
}
```

代码 18-41 计算_maxThumbnails 和_numDisplayedThumbnails

文档的页数发生变化时需要重排缩略图，这时会有一个名为“PaginationCompleted”的事件来通知应用程序。在我们应用程序中有一个类型为 FlowDocumentPageViewer 的控件 FDPV，它有一个 Document 属性，也是其父类的属性。类型为 IDocumentPaginatorSource，这是一个接口，其中只有一个需要实现的 DocumentPaginator 属性。它是一个抽象类，流文档的 DynamicDocumentPaginator 实现了该类。而 PaginationCompleted 是 DynamicDocumentPaginator 的一个事件，如图 18-38 所示。

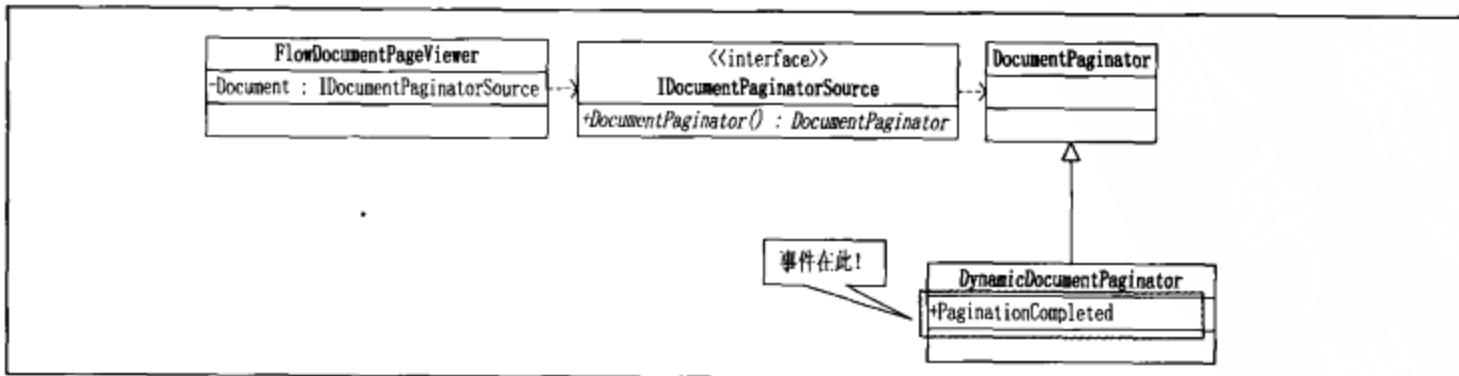


图 18-38 PaginationCompleted 事件的位置

打开文档之后可以为这个事件添加处理函数，在该函数中每次删除以前的所有分页。然后重新添加到缩略图列表中，代码如下：

```
private void OnOpen(object sender, ExecutedRoutedEventArgs e)
{
    MainWindow win = (MainWindow)sender;
    // 打开文档成功，则创建 Thumbs
    if (win.OpenDocument())
    {
        win.CreateThumbs();
        (FDPV.Document.DocumentPaginator as
        DynamicDocumentPaginator).PaginationCompleted +=
            new EventHandler(PaginationCompleted);
    }
}
void PaginationCompleted(object sender, EventArgs e)
{
    Dispatcher.BeginInvoke(DispatcherPriority.Normal,
        new DispatcherOperationCallback(
            delegate { RefreshThumbnails(); return null; }), null);
}
public void RefreshThumbnails()
{
    if ((ThumbList.Items != null) && (ThumbList.Items.Count > 0))
        ThumbList.Items.Clear();
    _currentThumbnail = 0;
    _maxthumbnails = FDPV.PageCount;
    for (int i = _currentThumbnail; i < _maxthumbnails; i++)
        AddPageThumb(i);
    ThumbList.Items.MoveCurrentToPosition(_currentThumbnail);
}
```

代码 18-42 PaginationCompleted 事件的处理函数实现

18.5.5 实现书签和标注功能

1. 书签和标注构成

如图 18-39 所示，书签和标注的外观相同，均由一个按钮和一个不规则多边形组成。不同是书签的不规则多边形填充色为黄色，而标注填充色为绿色。

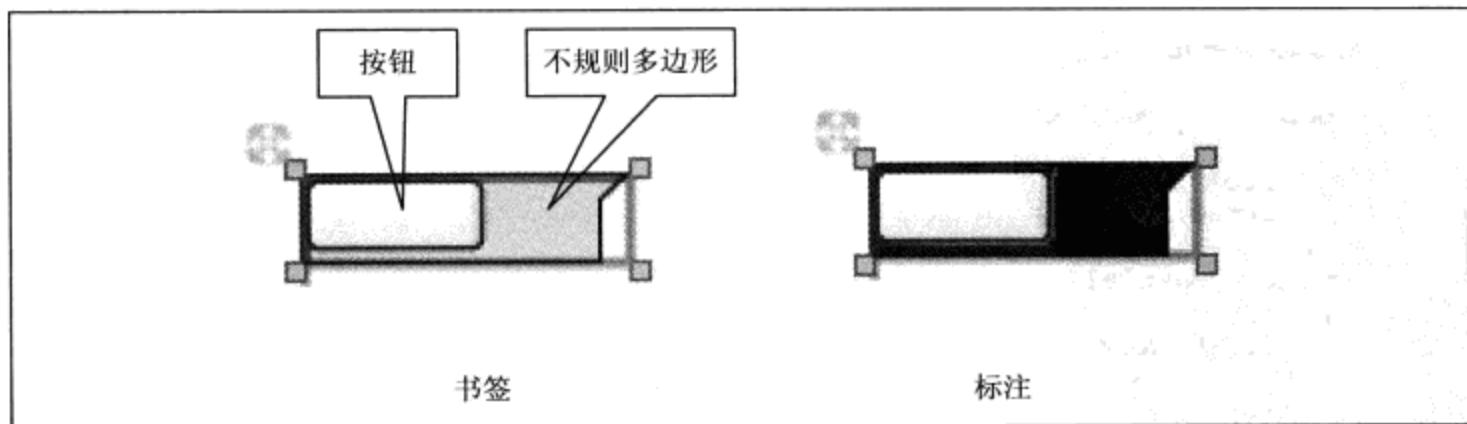


图 18-39 书签和标注的构成

我们将使用一个 StackPanel 来实现书签和标注，XAML 文件的内容如下：

```
<StackPanel x:Class="mumu_documentreader.BookOrCommandItem"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="StackPanel_Loaded"
    Orientation="Horizontal">
    <Canvas Height="28" VerticalAlignment="Center" Margin="1">
        <!--不规则多边形-->
        <Path StrokeThickness="1" x:Name="MarkPath" Fill="Green">
            <Path.Stroke>
                <SolidColorBrush Color="Black" />
            </Path.Stroke>
            <Path.Data>
                <PathGeometry>
                    <PathFigure IsClosed="True">
                        <LineSegment Point="0, 24" />
                        <LineSegment Point="78, 24" />
                        <LineSegment Point="78, 7" />
                        <LineSegment Point="85, 0" />
                    </PathFigure>
                </PathGeometry>
            </Path.Data>
        </Path>
        <!--按钮-->
        <Button Name="GoToMark">
            <Button.ToolTip>
                <ToolTip>到该位置</ToolTip>
            </Button.ToolTip>
            <TextBlock Name="TB" TextWrapping="NoWrap" TextTrimming="CharacterEllipsis"
Width="40"/>
            <Button.ContextMenu>
                <ContextMenu>
                    <MenuItem x:Name="GoToMenu" Header="到....."></MenuItem>
                    <MenuItem x:Name="DeleteMark" Header="删除"></MenuItem>
                </ContextMenu>
            </Button.ContextMenu>
        </Button>
    </Canvas>
</StackPanel>
```

代码 18-43 用 StackPanel 实现书签和标注

BookOrCommandItem 派生自 StackPanel，构造函数中需要传入一个枚举值以标识书签和标注，然后为不规则多边形填充不同的颜色，代码如下：

```
public enum BookOrCommandItemType
{
    Book,
    Command
};
public partial class BookOrCommandItem : StackPanel
{
    BookOrCommandItemType _type;
    public BookOrCommandItem(BookOrCommandItemType type)
    {
        InitializeComponent();
        _type = type;
    }
    private void StackPanel_Loaded(object sender, RoutedEventArgs e)
```

```

    {
        if (_type == BookOrCommandItemType.Book)
        {
            SolidColorBrush brush = new SolidColorBrush(Colors.Yellow);
            MarkPath.Fill = brush;
        }
        else
        {
            SolidColorBrush brush = new SolidColorBrush(Colors.Green);
            MarkPath.Fill = brush;
        }
    }
}

```

代码 18-44 BookOrCommandItem 的构造函数

2. 手工实现添加书签和标注功能

首先仍然需要在窗口的初始化函数中启用注释服务，然后在窗口关闭时禁用该服务。注意在初始化注释服务时会添加一个 StoreContentChanged 的事件处理函数，其主要作用是添加书签或者注释时会为书签列表或者注释列表添加一条记录，这些最终在 AddBookmarkOrComment 函数中完成。每添加一条记录，还为 3 个事件分别添加事件处理函数（代码③、④和⑤）。3 个事件分别是单击按钮事件、单击弹出菜单“到……”和“删除”两个事件，它们实际对应两个事件处理函数，即 GoToMark_Click 和 DeleteMark_Click，如图 18-40 所示。

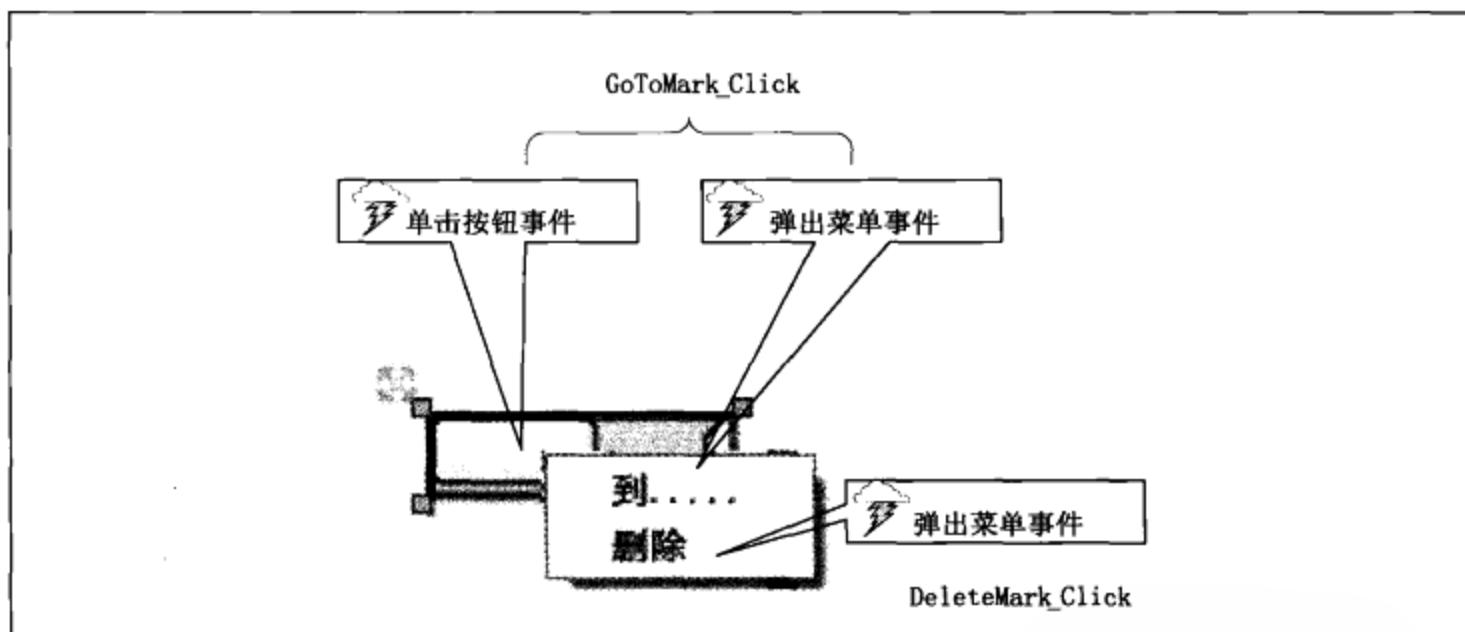


图 18-40 三个事件对应的两个事件处理函数

代码如下：

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    _annotationBuffer = new MemoryStream();
    _annStore = new XmlStreamStore(_annotationBuffer);
    _annServ = new AnnotationService(FDPV);
    ① _annStore.StoreContentChanged +=
        new
        StoreContentChangedEventHandler(_annStore_StoreContentChanged);
    _annServ.Enable(_annStore);
}

```

```

        }
        void _annStore_StoreContentChanged(object sender,
StoreContentChangedEventArgs e)
{
    if (e.Action == StoreContentAction.Deleted) return;
    Annotation ann = e.Annotation;
    ② if (ann.AnnotationType.Name == "Highlight")
    {
        AddBookmarkOrComment(BookmarkList, ann);
    }
    else
    {
        AddBookmarkOrComment(CommentsList, ann);
    }
}

private void AddBookmarkOrComment(ListBox collection, Annotation ann)
{
    .....
    StackPanel EntryInList ;
    if (collection == BookmarkList)
        EntryInList = new BookOrCommandItem(BookOrCommandItemType.Book);
    else
        EntryInList = new BookOrCommandItem
            (BookOrCommandItemType.Command);

    .....
    Button GoToMark = LogicalTreeHelper.FindLogicalNode(
        EntryInList, "GoToMark") as Button;
    if (GoToMark != null)
    {
        GoToMark.Tag = ann;
        GoToMark.Click += new RoutedEventHandler(GoToMark_Click);
    }

    MenuItem GoToMenu = LogicalTreeHelper.FindLogicalNode(
        GoToMark.ContextMenu, "GoToMenu") as
MenuItem;

    ④ GoToMenu.Click += new RoutedEventHandler(GoToMark_Click);
    GoToMenu.Tag = ann;

    MenuItem DeleteMark = LogicalTreeHelper.FindLogicalNode(
        GoToMark.ContextMenu, "DeleteMark") as
MenuItem;
    DeleteMark.Click += new
    ⑤ RoutedEventHandler(DeleteMark_Click);
    DeleteMark.Tag = ann;
    collection.Items.Add(EntryInList);
    .....
}

```

代码 18-45 手工方式添加书签和标注功能

3. 自定义命令和绑定命令

我们需要自定义命令并绑定命令及其处理函数，添加书签的命令代码如下：

```
public static RoutedCommand AddBookmark
```

```

    { get { return DeclareCommand(ref _AddBookmark, "AddBookmark"); } }
private static RoutedCommand DeclareCommand(ref RoutedCommand command,
                                            string commandDebugName, InputGesture gesture)
{
    if (command == null)
    {
        InputGestureCollection collection = null;
        if (gesture != null)
        {
            collection = new InputGestureCollection();
            collection.Add(gesture);
        }
        command = new RoutedCommand(commandDebugName,
                                     typeof(MainWindow), collection);
    }
    return command;
}

```

代码 18-46 添加书签

在窗口的 Loaded 事件中绑定该命令和 OnAddBookmark 函数，该函数通过 AnnotationHelper 的一个静态方法 CreateHighlightForSelection 来创建一个书签，代码如下：

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    this.CommandBindings.Add(
        new CommandBinding(MainWindow.AddBookmark,
                           new ExecutedRoutedEventHandler(OnAddBookmark),
                           new CanExecuteRoutedEventHandler(OnNewQuery)));
    ....
}

void OnAddBookmark(object sender, RoutedEventArgs args)
{
    try
    {
        System.Windows.Media.Color col = (System.Windows.Media.Color)System.
Windows.Media.ColorConverter.ConvertFromString("#A6FFFF00");
        System.Windows.Media.Brush myBrush = new SolidColorBrush(col);
        string userName = System.Windows.Forms.
SystemInformation.UserName;
        AnnotationHelper.CreateHighlightForSelection(
            _annServ, userName, myBrush);
    }
    catch (InvalidOperationException)
    {
        return;
    }
}
private static void OnNewQuery(
    object target, CanExecuteRoutedEventArgs args)
{
    args.CanExecute = true;
}

```

代码 18-47 创建一个书签

最后在 FlowDocumentPageViewer 的 ContextMenu 中添加该命令，代码如下：

```

<FlowDocumentPageViewer x:Name="FDPV" Grid.Row="2" Grid.Column="2">
    <FlowDocumentPageViewer.ContextMenu>

```

```
<ContextMenu>
    .....
    <Separator />
    <MenuItem Name="cm_Bookmark" Header="添加书签"
Command="local:MainWindow.AddBookmark" />
</ContextMenu>
</FlowDocumentPageViewer.ContextMenu>
</FlowDocumentPageViewer>
```

代码 18-48 在 ContextMenu 里添加“添加书签”命令

18.6 接下来做什么

自此第五卷就完全结束了。这一卷内容涵盖了二维、三维图形、动画、文本和文档。几乎涵盖了所有“富”媒体的内容。但是本卷并没有涉及 WPF 对音频和视频的讨论。实际上 WPF 对音频的支持是有限制的，不过好在能够足以完成一些常见的任务。WPF 对视频的支持也仅仅是 Windows Media Player API 所提供的功能的一小部分而已。因此本书不会对 WPF 音频和视频作专门的介绍。接下来我们进入的将是 WPF 的高级话题。

参考文献

- [1] “北风之神”凤清远整理，“云中孤雁”制作《金庸全集典藏版》射雕英雄传：第四回 黑风双煞。
- [2] MSDN Library for Visual Studio 2008 SP1 Documents in Windows Presentation Foundation。
- [3] MSDN Library for Visual Studio 2008 SP1 Document Serialization and Storage。

华山之巅

第六卷



第 19 章

互操作——“小无相功”

玄生见他这三下出手，无不远胜于己，霎时间心丧若死：“只怕这位神僧所言不错，我少林派七十二门绝技确是传自天竺，他从原地习得秘奥，以致比我中土高明得多。”当即合十躬身，说道：“国师神技，令小僧大开眼界，佩服，佩服！”……霎时之间，大殿上寂静无声，人人都为鸠摩智的绝世神功所镇慑……

——《天龙八部：第三十九章 解不了 名缰系嗔贪》^[1]

这是吐蕃国国师鸠摩智闯少林宝刹的一段，他用小无相功的内力模仿少林七十二绝技。连少林高僧都误以为他精通七十二绝技，不免心灰意冷。在程序世界里，开发人员都有两个梦想。第 1 个梦想是编译一次程序就可以在任何平台（Windows 及 Linux 等）上运行；第 2 个梦想就是掌握一种程序语言，就可以调用所有其他语言编写的程序，其实这两个梦想从某种程度上来说也是催生 .Net 平台的源动力。

尽管这两个梦想到目前并没有完全实现，但是 WPF 的互操作似乎就是我们的“小无相功”。无论是 WPF 调用 Windows Forms，或者调用 Win32，或者 Windows Form 嵌入 WPF 等。下面不断用内力催动的是“小无相功”，即 WPF 的互操作。

掌握了 WPF 的互操作，或许离我们的第 2 个梦想就更近一步。

本章内容如下。

- (1) 为什么需要互操作。
- (2) 互操作的几种类型。
- (3) Windows Forms 和 WPF。
- (4) 在 Win32 中嵌入 WPF 内容。
- (5) 在 WPF 中嵌入 Win32 内容。
- (6) 接下来做什么。

19.1 为什么需要互操作？

WPF 中之所以需要互操作，至少存在以下两个方面的原因。

(1) 过去可能已经用 Win32 或 Windows Forms（简称 Win Form）开发了大量的应用，如果现在用 WPF 完全重写一遍，则代价太大。一种更为明智的办法就是将 WPF 编写的新的功能和现有的应用结合起来，或者是在 WPF 的框架中复用现有的应用。这两种结合模式都需要用到 WPF 的互操作。

(2) WPF 与 Win32 及 WinForm 比起来毕竟是一个新生事物，过去很多成熟的控件在 Win32 和 Win Form 中已经声名大噪，耳熟能详。但是在 WPF 中可能没有，也可能不完善。使用 WPF 的互操作，可以直接使用 Win32 或者 Win Form 里的现有控件来弥补 WPF 缺失的特性。

WPF 之所以能够实现互操作也取决于两个方面的原因，一方面当然是 WPF 自身的设计能够支持互操作；另外一方面归功于 .Net 对托管和非托管代码的互操作支持。这一章牵涉到 Win32、Win Form 和 WPF，以及托管和非托管代码的部分知识。内容多而杂，因此我们大部分知识将会以向导的方式来介绍，在一步一步做的过程中体会 WPF 的互操作。

19.2 互操作的几种类型

我们本章讨论或者不讨论的互操作的几种类型如图 19-1 所示。Win32 是 Windows 平台的基础，无论是 MFC、Win Form 或者 WPF，如果能够深入挖掘源代码，则总能看到 Win32 的“魅影”。严格意义上来说 MFC、DirectX 和 Win32 本身不存在互操作的问题，甚至 ActiveX 和 Win32 之间也不存在这个问题。它们之间的调用本身是无障碍的。而 WPF 和 ActiveX 之间没有直接的通道可以互操作，因此如果希望 WPF 调用 ActiveX 控件，则必须借助 Win32 或者 Win Form 作为一个技术中转。Win32 和 Win Form 之间的互操作主要还是借助于标准的 .Net 互操作技术来实现。本章涉及的互操作部分图中以实线标出，而虚线部分则是本章不会涉及的内容。每种类型的互操作实际上涉及两个方面，如 A 与 B 的互操作实际上包括 A 如何调用 B 和 B 如何调用 A 两个方面。Win Form 和 WPF 之间的互操作相对来说会简单一些，因为它们毕竟是托管代码之间的调用，而 Win32 和 WPF 之间的互操作就稍稍显得复杂一些。

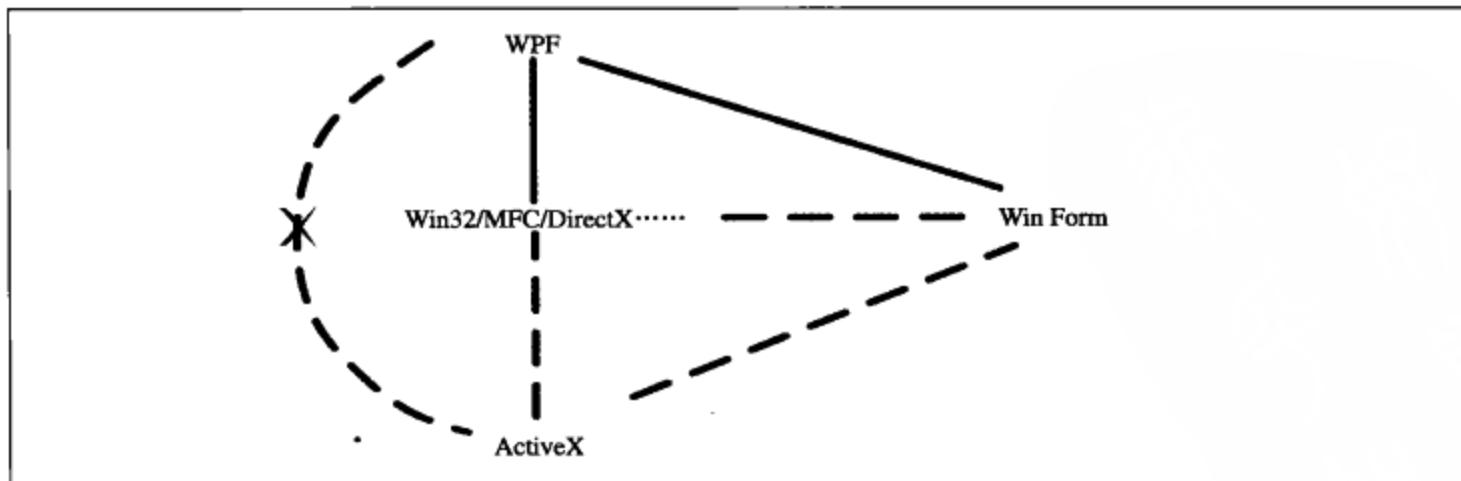


图 19-1 互操作的几种类型

19.3 Windows Forms 和 WPF

Win Form 和 WPF 互操作有如下两种形式。

(1) WPF 程序将 WinForm 作为对话框弹出，或者反之。

(2) 将 WinForm 内容嵌入在 WPF 窗口中，或者反之。

19.3.1 对话框

第 1 种形式还需要细分为两种，即作为模态对话框和非模态对话框弹出。二者的区别在于前者弹出之后且关闭之前，用户无法切换到同一程序的另一个窗口；后者弹出之后，用户可以自由地在该对话框与同一程序之间的对话框间自由交换。正是这一区别，导致 WPF 弹出模态 WinForm 窗口和非模态的 WinForm 窗口会有些细微的区别，同样在 WinForm 程序中也会有些细微的区别。

1. 在 WPF 项目中弹出 WinForm 窗口

(1) 建立一个 WPF 程序，窗口中放置了两个按钮。一个按钮弹出一个模态的对话框；另一个弹出非模态的对话框，如代码 19-1 所示（详见 `mumu_wpfwinformdlg` 工程）。

```
<Window x:Class="mumu_wpfwinformdlg.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="150" Width="300">
    <StackPanel>
        <Button x:Name="modelbtn" Margin="5" Content="模态对话框"/>
        <Button x:Name="modelessbtn" Margin="5" Content="非模态对话框" />
    </StackPanel>
</Window>
```

代码 19-1 模态对话框和非模态对话框

(2) 在该工程中添加一个 WinForm 窗口，右击工程。选择“Add”|“New Item”选项，弹出 Add New Item 对话框。选择左侧树节点中的“Windows Forms”及右侧列表框中的 Windows Forms 选项，新建一个 Form 表单，如图 19-2 所示。

在 VS2010 中添加 WinForm 窗口之后，它会自动添加相关程序集的引用，如 `System.Windows.Forms.dll` 及 `System.Drawing.dll`，而且可以在 WPF 的项目中设计 WinForm 窗口。当切换到 WinForm 窗口页面时会出现 WinForm 设计器，同时工具栏中也会出现 WinForm 的相关控件。我们可以添加 `Label`、`RichTextBox` 和 `Button` 这样简单的控件，如图 19-3 所示。

此外需要将“确定”按钮的 `DialogResult` 属性设置为 `OK`，表示按下该按钮窗返回的结果是 `OK`。同理为“取消”按钮的 `DialogResult` 属性设置为 `Cancel`，如图 19-4 所示。

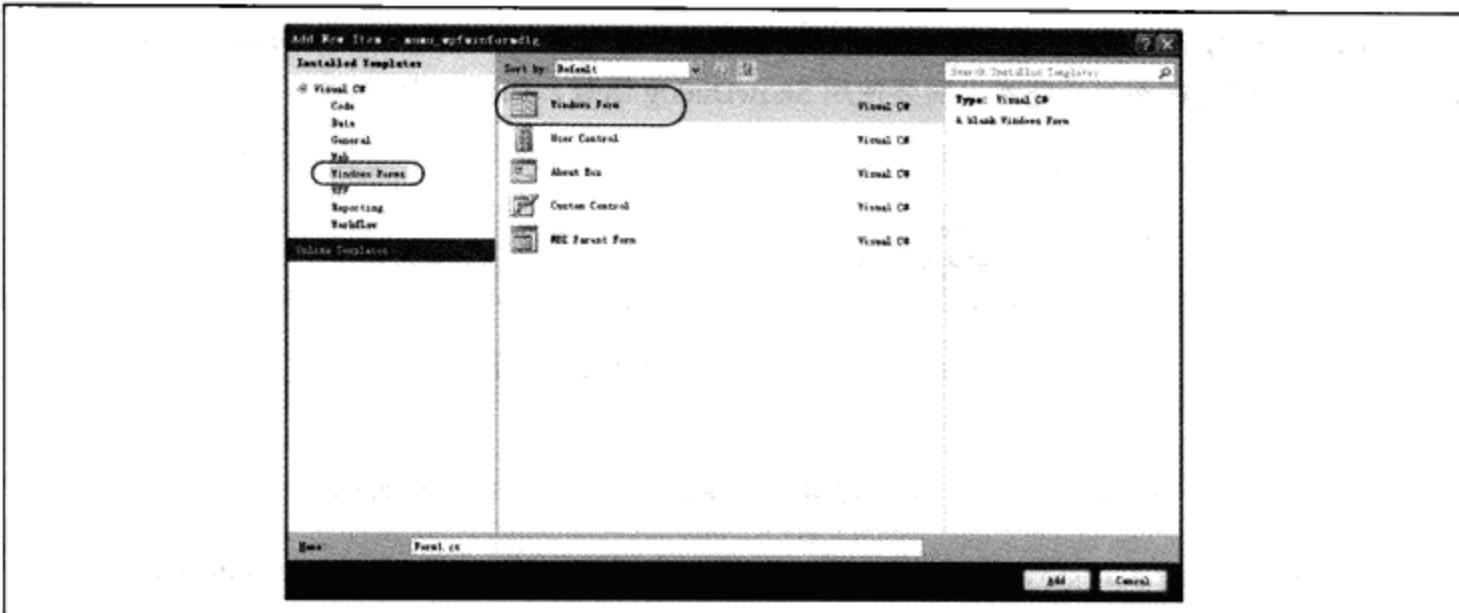


图 19-2 新建一个 Form 表单

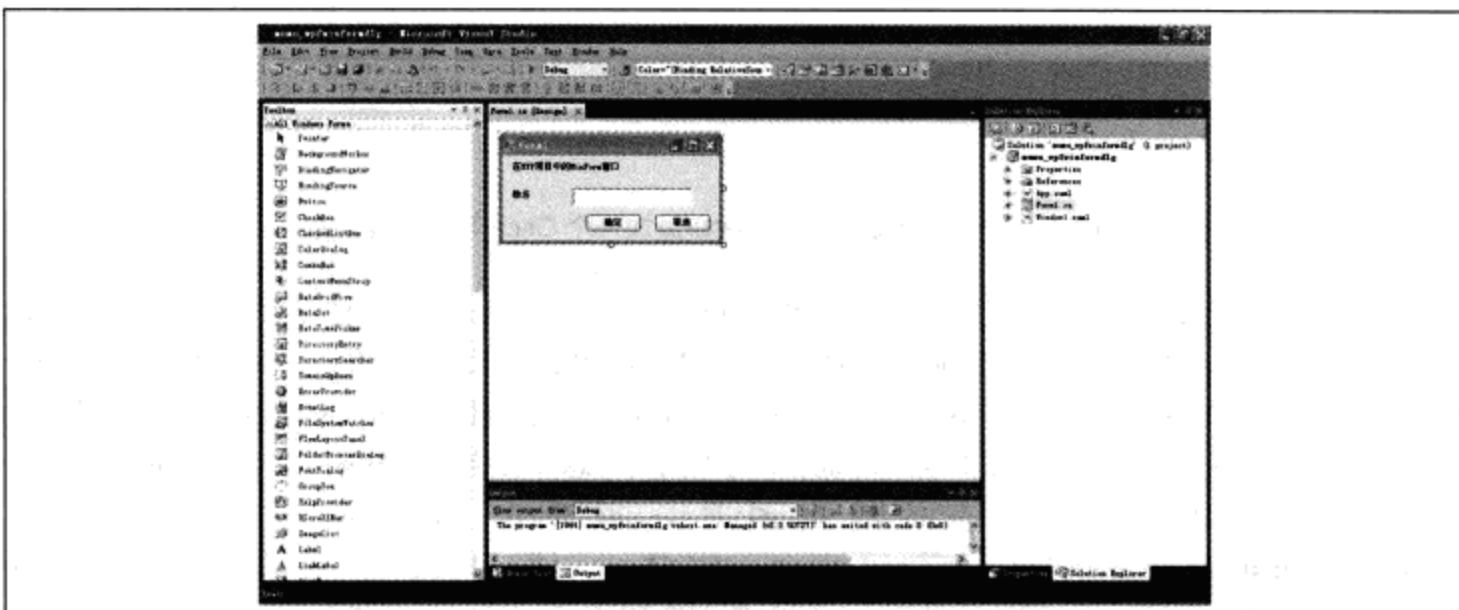


图 19-3 添加控件

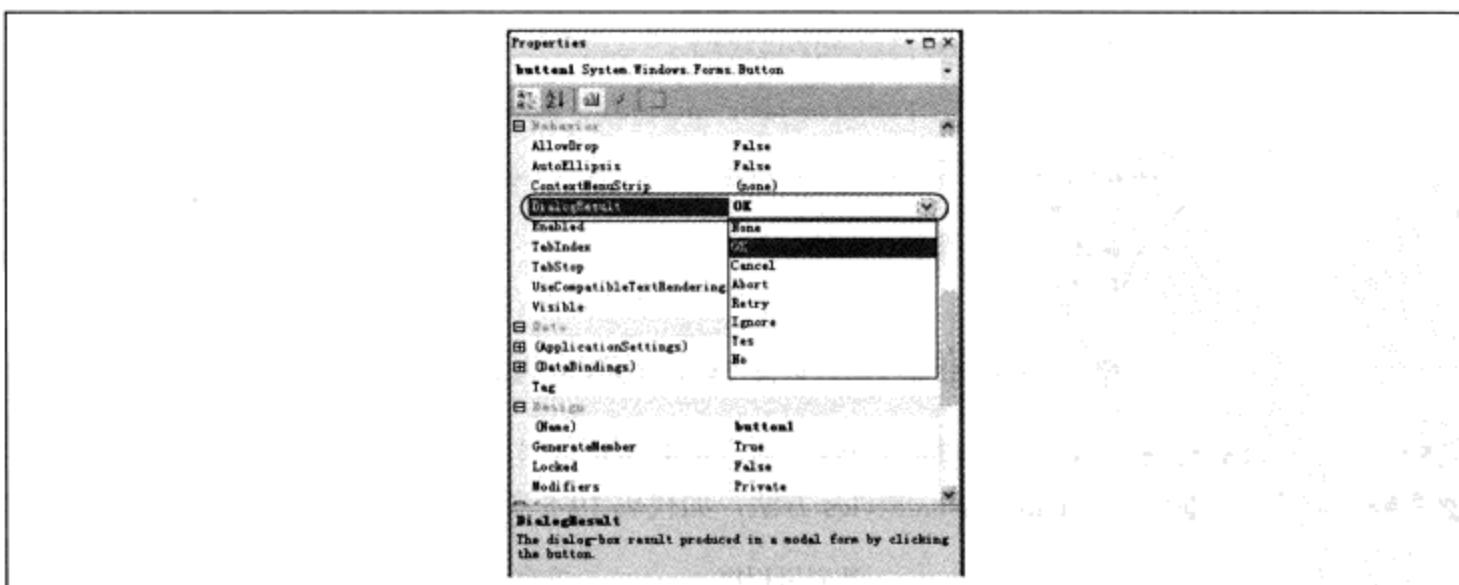


图 19-4 设置“确定”和“取消”按钮的 DialogResult 属性

(3) 为第 1 个“模态对话框”按钮添加一个 Click 事件处理函数，该函数弹出 WinForm 模态对话框。WinForm 窗口作为模态对话框弹出需要使用 ShowDialog 函数，如代码 19-2 所示。

```
private void modelbtn_Click(object sender, RoutedEventArgs e)
{
    Form1 frm = new Form1();
    frm.Text = "模态对话框";
    if (frm.ShowDialog() == System.Windows.Forms.DialogResult.OK)
    {
        MessageBox.Show("模态对话框单击确定按钮后关闭");
    }
}
```

代码 19-2 为“模态对话框”按钮添加一个 Click 事件处理函数

(4) 为第 2 个“非模态对话框”按钮添加一个 Click 事件处理函数，该函数弹出 WinForm 非模态对话框。WinForm 窗口作为非模态对话框弹出需要使用 Show 函数，如代码 19-3 所示。

```
private void modelessbtn_Click(object sender, RoutedEventArgs e)
{
    Form1 frm = new Form1();
    frm.Text = "非模态对话框";
    frm.Show();
}
```

代码 19-3 为“非模态对话框”按钮添加一个 Click 事件处理函数

这样弹出的对话框会有一些问题，如弹出的 WinForm 窗口无法完全识别所有的键盘消息，以及按下 Tab 键后无法在控件之间切换焦点。这些问题正是模态对话框与非模态对话框的区别造成的，模态对话框弹出时，它不会和应用程序其他窗口切换，因此所有的鼠标和键盘消息都会交由该对话框去处理。但是非模态对话框情况不同，应用程序需要经过判断之后将不同的消息分发给合适的窗口。为了能够让非模态对话框正确地接受到所有的键盘消息，需要在 WinForm 窗口显示调用 WindowsFormsHost.EnableWindowsFormsInterop 方法。调用该方法还需要添加 WindowsFormsIntegration.dll 程序集的引用。同时需要声明 System.Windows.Forms.Integration 命名空间，如代码 19-4 所示。

```
using System.Windows.Forms.Integration;
.....
private void modelessbtn_Click(object sender, RoutedEventArgs e)
{
    WindowsFormsHost.EnableWindowsFormsInterop();
    Form1 frm = new Form1();
    frm.Text = "非模态对话框";
    frm.Show();
}
```

代码 19-4 让非模态对话框正确地接受到所有的键盘消息

(5) 如果希望 WinForm 的对话框和整个 WPF 项目中的对话框风格尽可能的相近一些，那么可以在整个应用程序启动时调用 EnableVisualStyles 方法，如代码 19-5 所示。

```
// Application_Startup 是应用程序 Startup 的事件处理函数
private void Application_Startup(object sender, StartupEventArgs e)
```

```
        System.Windows.Forms.Application.EnableVisualStyles();
    }
```

代码 19-5 使 WinForm 的对话框和整个 WPF 项目中的对话框风格尽可能的相近

我们可以从图 19-5 中比较调用和不调用 EnableVisualStyles 方法窗口之间的区别。

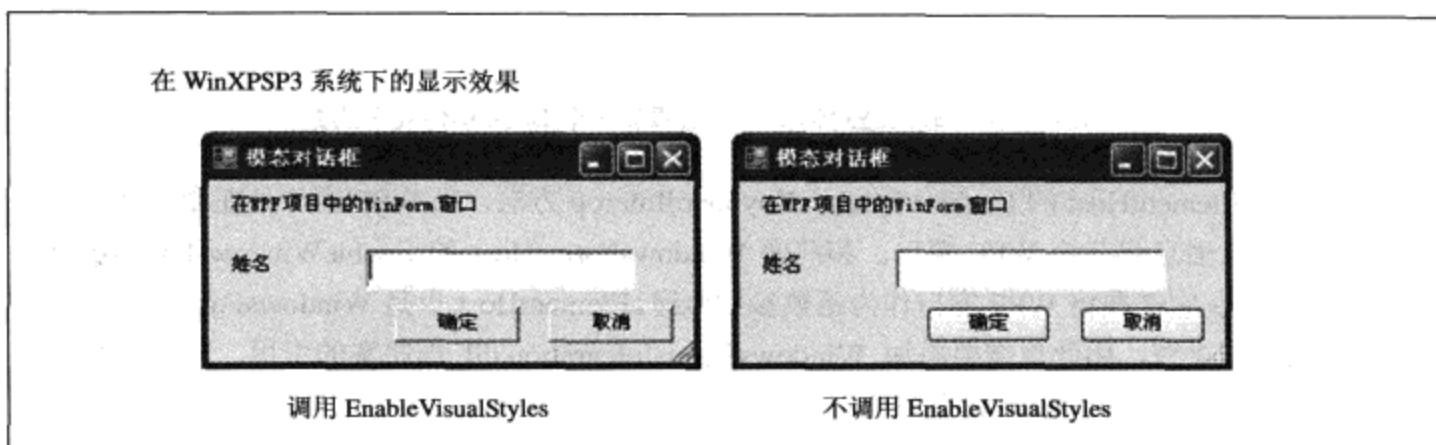


图 19-5 两种方法的不同窗口

2. 在 WinForm 项目中弹出 WPF 窗口

(1) 新建一个 WinForm 项目，在 Form 窗口中添加两个按钮，分别弹出一个模态和非模态的对话框，如图 19-6 所示（详见 `mumu_winformwpfdlg` 工程）。

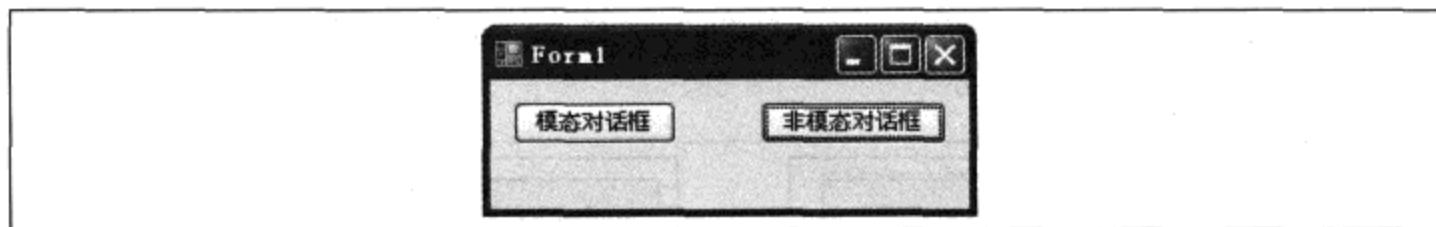


图 19-6 新建一个 WinForm 项目

(2) 在 WinForm 项目中不能直接新建一个 WPF 窗口，但是可以添加一个现有的 WPF 窗口。在 WinForm 项目中，如果当前的页面是 WPF 窗口，那么 VS2010 也会自动出现 WPF 的设计器，同时 ToolBox 也会自动加载 WPF 相关的控件。

(3) 为第 1 个“模态对话框”按钮添加一个 Click 事件处理函数，该函数弹出 WPF 窗口。它需要使用 ShowDialog 函数，如下代码 19-6 所示。

```
private void modelbtn_Click(object sender, EventArgs e)
{
    MainWindow win = new MainWindow();
    win.Title = "模态对话框";
    win.ShowDialog();
}
```

代码 19-6 为“模态对话框”按钮添加一个 Click 事件处理函数

(4) 为第 2 个“非模态对话框”按钮添加一个 Click 事件处理函数，该函数弹出 WPF 窗口，它需要使用 Show 函数。同样 WPF 窗口作为非模态对话框也会不能完全接受到键盘消息，甚至比 WinForm

窗口更为严重，无法接收到任何键盘消息。因此也需要特殊处理，如代码 19-7 所示。

```
private void modelessbtn_Click(object sender, EventArgs e)
{
    MainWindow win = new MainWindow();
    ElementHost.EnableModelessKeyboardInterop(win);
    win.Title = "非模态对话框";
    win.Show();
}
```

代码 19-7 为“非模态对话框”按钮添加一个 Click 事件处理函数

这次调用的是 ElementHost 的 EnableModelessKeyboardInterop 方法，当 WPF 窗口处在激活状态，该方法可以将键盘消息截获给 WPF 窗口。和前面 WindowsFormsHost 的 EnableWindowsFormsInterop 有些细微的差别，它需要将 WPF 窗口作为函数参数传递。ElementHost 也是 WindowsFormsIntegration 程序集中的一个类型，因此也需要添加 WindowsFormsIntegration.dll 程序集的引用，同时需要声明 System.Windows.Forms.Integration 命名空间。

19.3.2 在同一个窗口中混合 WPF 和 WinForm 内容

1. 空域（Airspace）规则

在同一个窗口中混合 WPF 和 WinForm 内容相互不容影响侵犯，这就是空域原则。空域原则容易违背的情况。一是 WPF 中的动画超出了其内容空域而侵占了 WinForm 或者 Win32 内容的空域；二是 WPF 内容以半透明的形式叠加在 WinForm 或者 Win32 内容上方。这两种情况由于违背“空域”原则，因此不能实现，如图 19-7 所示。

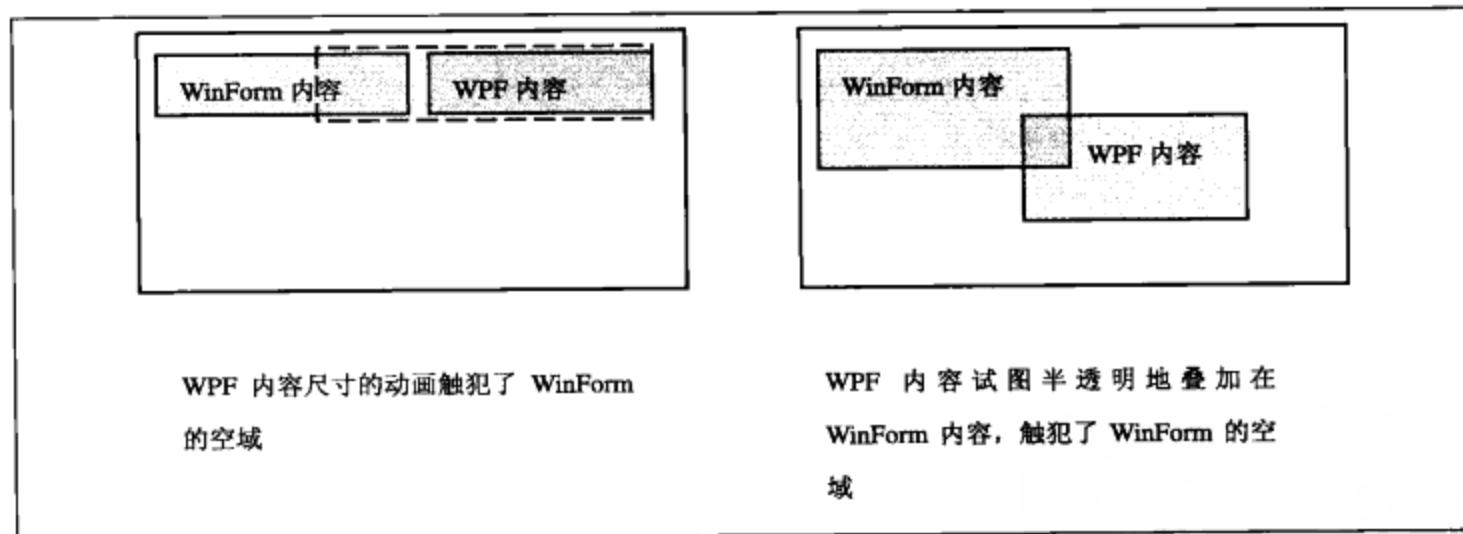


图 19-7 两种空域原则容易违背的情况

一个 Win32 窗口中嵌套 WPF 内容，然后 WPF 内容中再嵌套 WinForm 内容，这两种情况不违背空域规则。这时除去 WinForm 内容以外，其他两个“空域”是带“洞”的矩形。这种空域有些类似国中之国的梵蒂冈，如图 19-8 所示。

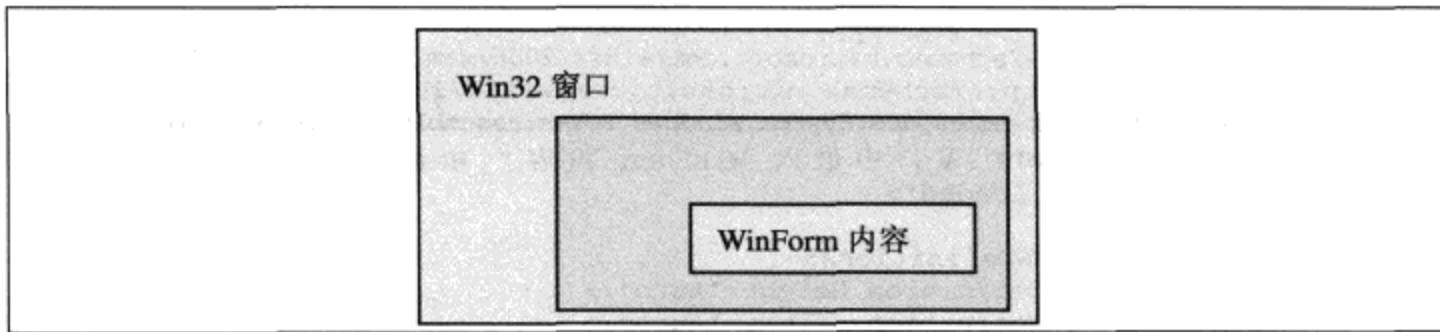


图 19-8 窗口嵌套

2. 在 WPF 窗口中嵌入 WinForm 内容

在 WPF 中嵌入 WinForm 内容非常简单，只需要借助于一个 WindowsFormsHost 类型。WindowsFormHost 本身是派生自 FrameworkElement，因此它本身是一个普通的 WPF 元素，可以自由地作为 WPF 的逻辑树的一个节点。同时它也可以包装 WinForm 内容，这样就实现了在 WPF 窗口中嵌入 WinForm 内容。

WinForm 中一个功能非常强大的控件是 PropertyGrid 控件，它通过反射机制将 .Net 的任何对象的属性都显示在表格中，同时还可以修改这些属性。我们这里将一个 PropertyGrid 嵌入到 WPF 窗口中，如图 19-9 所示。当用户在 ComboBox 中选择了 Button，右侧的 PropertyGrid 就会选择按钮的所有属性，可以修改按钮的字体属性，使其字体变大。

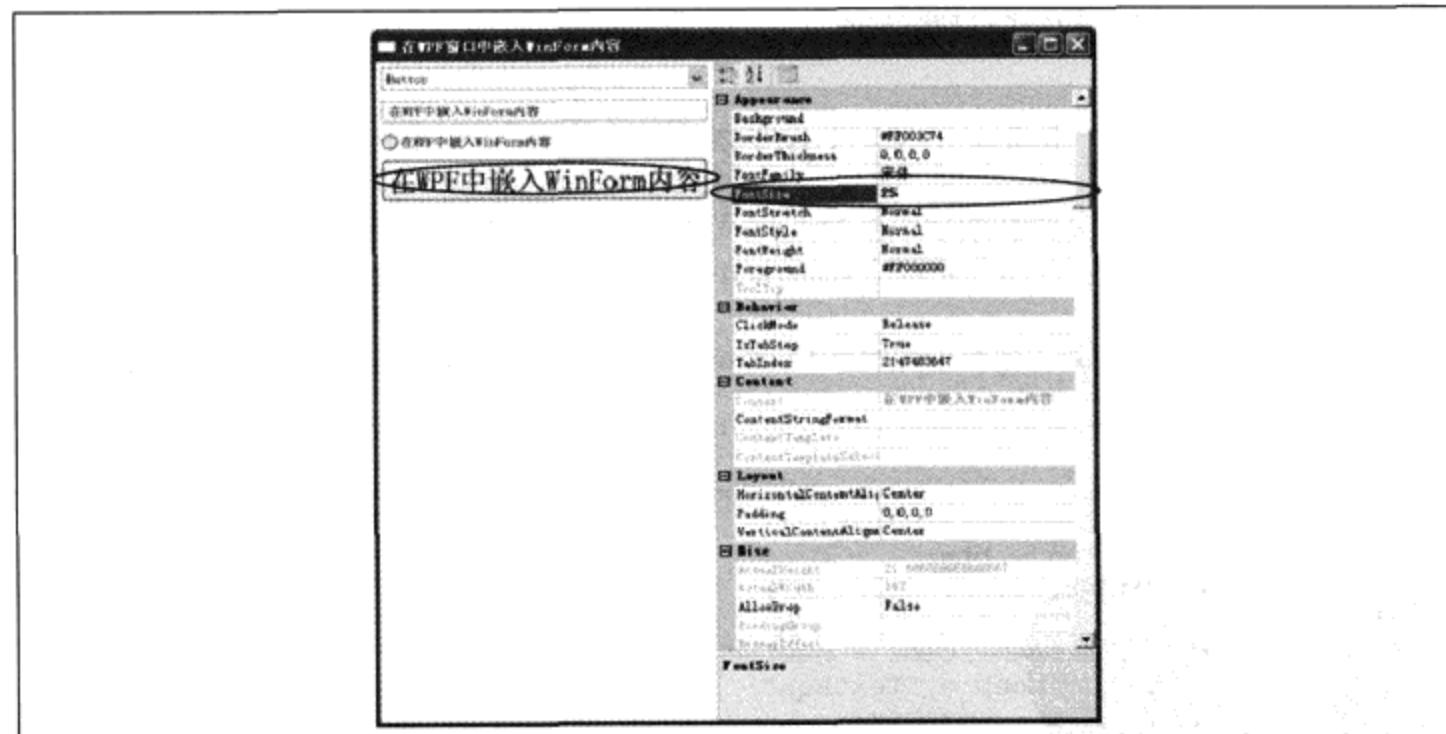


图 19-9 修改按钮的属性

(1) 新建一个 WPF 项目，在这个项目中要嵌入 PropertyGrid，因此需要添加 System.Windows.Forms.dll 程序集的引用。并且由于需要借助于 WindowsFormHost 类包装 PropertyGrid 包装，因此需要添加 WindowsFormsIntegration.dll 程序集的引用。窗口的代码如代码 19-8 所示（详见 mumu_wpfhostwinform 工程）。

```

<Window x:Class="mumu_wpfhostwinform.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
①      xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
        Title="在 WPF 窗口中嵌入 WinForm 内容" Height="631" Width="434"
        Loaded="Window_Loaded">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <ComboBox x:Name="combobox" Margin="5" Height="20"
② SelectionChanged="ComboBox_SelectionChanged">
        </ComboBox>
        <Button x:Name="btn" Grid.Row="3" Content="在 WPF 中嵌入 WinForm 内容"
Margin="5"/>
        <TextBox x:Name="textbox" Grid.Row="1" Text="在 WPF 中嵌入 WinForm
内容" Margin="5"/>
        <RadioButton x:Name="radiobt" Grid.Row="2" Content="在 WPF 中嵌入
WinForm 内容" Margin="5"/>
        <WindowsFormsHost Grid.Column="1" Grid.RowSpan="5" >
            <wf:PropertyGrid x:Name="propertyGrid"/>
        </WindowsFormsHost>
    </Grid>
</Window>

```

代码 19-8 在一个 WPF 项目中嵌入 PropertyGrid

上述代码的关键一是代码①处声明程序集 System.Windows.Forms.dll，如果未声明，则无法在 XAML 中使用 PropertyGrid 类型；代码②处通过 WindowsFormsHost 类型包装 PropertyGrid。

(2) 在窗口 Loaded 事件处理函数中初始化 ComboBox，如代码 19-9 所示。

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ComboBoxItem item1 = new ComboBoxItem();
    item1.Content = "Button";
    item1.Tag = btn;
    combobox.Items.Add(item1);

    ComboBoxItem item2 = new ComboBoxItem();
    item2.Content = "TextBox";
    item2.Tag = textbox;
    combobox.Items.Add(item2);

    ComboBoxItem item3 = new ComboBoxItem();
    item3.Content = "RadioButton";
    item3.Tag = radiobt;
    combobox.Items.Add(item3);

    ComboBoxItem item4 = new ComboBoxItem();
    item4.Content = "ComboBox";

```

```
        item4.Tag = combobox;
        combobox.Items.Add(item4);
    }
```

代码 19-9 在窗口 Loaded 事件处理函数中初始化 ComboBox

(3) 在 ComboBox 的 SelectionChanged 事件中判断用户选择的控件，然后将该控件的引用赋值给 PropertyGrid 的 SelectedObject。这样在 PropertyGrid 可以显示该控件的所有属性，如代码 19-10 所示。

```
private void ComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    ComboBoxItem lbi = ((sender as ComboBox).SelectedItem as ComboBoxItem);
    UIElement ui = lbi.Tag as UIElement;
    ui.Focus();
    propertyGrid.SelectedObject = lbi.Tag;
}
```

代码 19-10 显示用户选择的控件的所有属性

3. 在 WinForm 窗口中嵌入 WPF 内容

在 WinForm 窗口中嵌入 WPF 内容也非常简单，这里也有一个类似 WindowsFormsHost 的包装类 ElementHost 包装 WPF 内容起来。它派生自 System.Windows.Forms.Control，因此在 Form 窗口中也被当做一个和 Button 一样的控件。这次我们在 WinForm 窗口中嵌入一个 WPF 的 UserControl，该 UserControl 中只有一个按钮。外部的 WinForm 控件可以设置该按钮的背景色、前景色和字体，如图 19-10 所示。

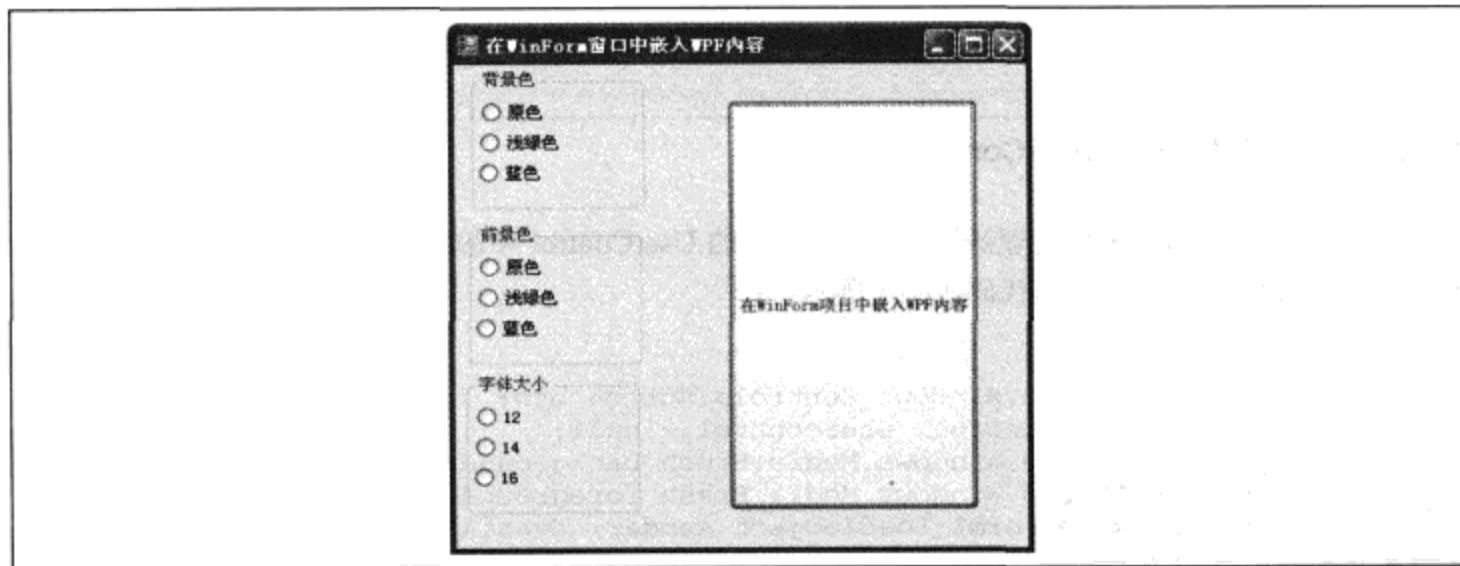


图 19-10 在 WinForm 窗口中嵌入一个 WPF 的 UserControl

(1) 新建一个 WinForm 工程，在其中添加一个新项，在弹出的 Add New Item 对话框中选择 WPF 的 UserControl 选项如图 19-11 所示（详见 mumu_winformhostwpf 工程）。

(2) 在 Form 的设计器中摆放所需控件，如果需要插入 WPF 的 UserControl，需要在工具箱中拖动一个 ElementHost 到窗口上。选中 ElementHost 后在其左上角会出现一个小三角符号，单击可以为 ElementHost 选择内容。如果其中为空，则需要重新编译工程，如图 19-12 所示。

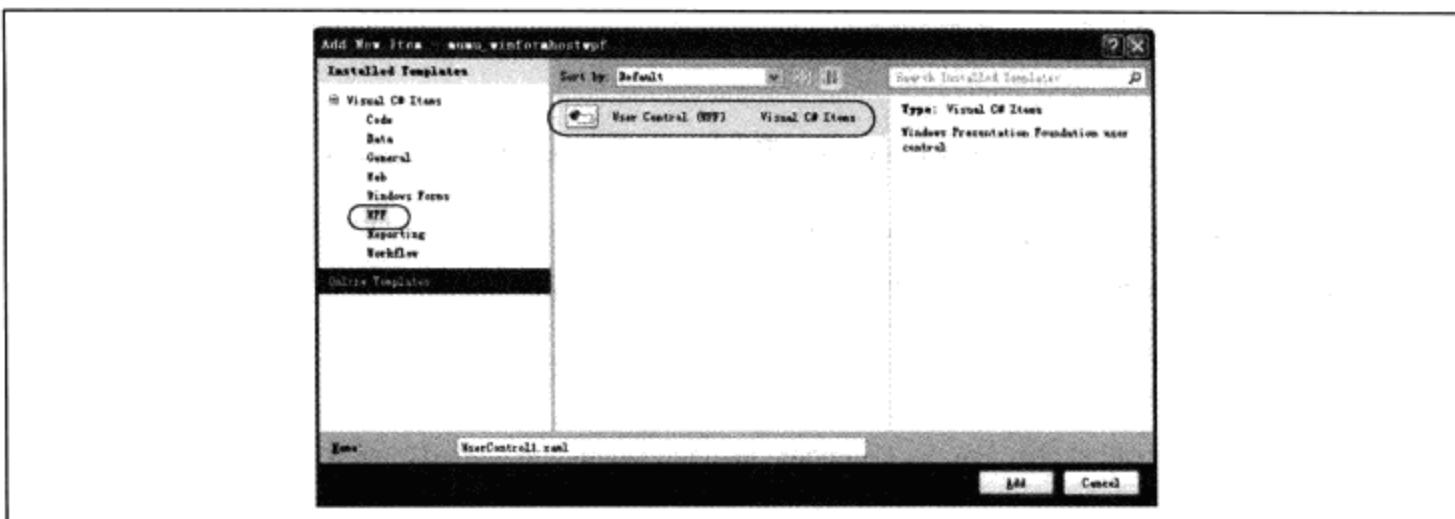


图 19-11 选择 WPF 的 UserControl 选项

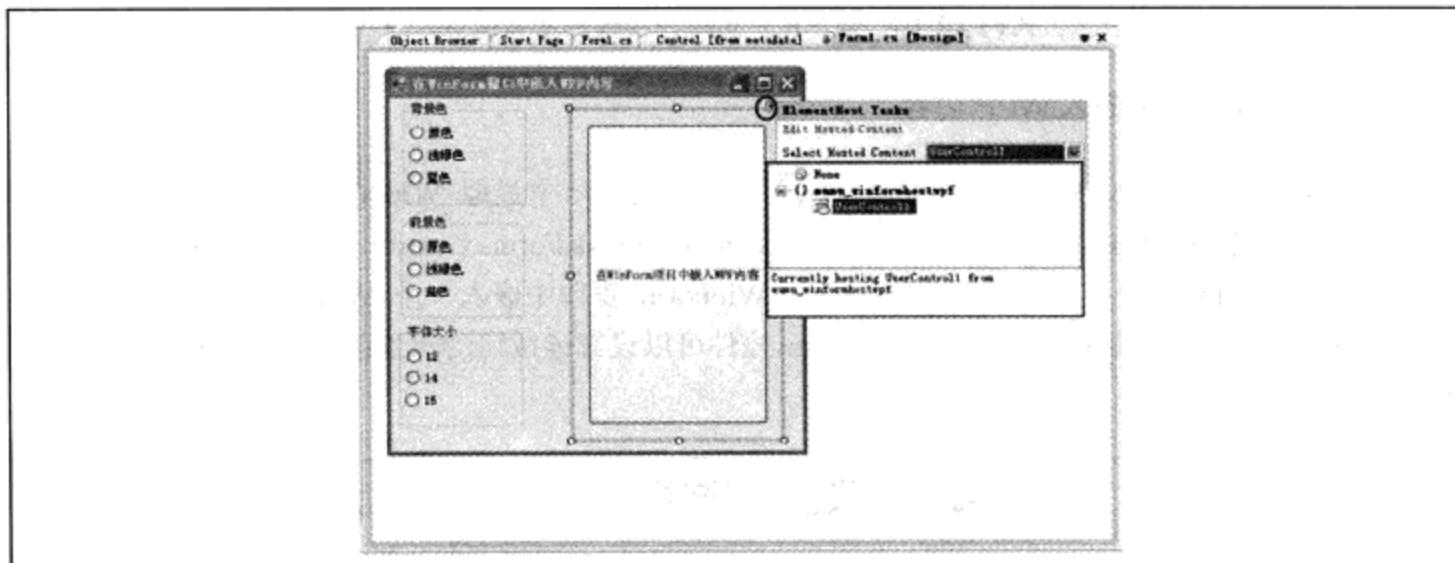


图 19-12 新建 WPF 的 UserControl

(3) 在窗口的 Loaded 事件处理函数中保存 WPF 的 UserControl 中的按钮实例，以便后面修改其属性。访问该按钮非常直接，如代码 19-11 所示。

```
.....
private System.Windows.Controls.Button btn;
private UserControl1 usercontrol = null;
private System.Windows.Media.Brush backgroundbrush;
private System.Windows.Media.Brush foregroundbrush;
private void Form1_Load(object sender, EventArgs e)
{
    usercontrol= elementHost1.Child as UserControl1;
    // 在 UserControl 里面 按钮的定义如下
    // <Button x:Name="btn" Content="在WinForm项目中嵌入WPF内容" Margin="15"/>
    btn = usercontrol.btn;
    backgroundbrush = btn.Background;
    foregroundbrush = btn.Foreground;
}
```

代码 19-11 保存 WPF 的 UserControl 中的按钮实例

(4) 在 RadioButton 的 CheckedChanged 事件处理函数中设置 Button 的属性，如代码 19-12 所示。

```

private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
    btn.Background = new System.Windows.Media.Solid
ColorBrush(System.Windows.Media.Colors.LightGreen);
}
private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
    btn.Background = backgroundbrush;
}
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
    btn.Background = new System.Windows.Media.Solid
ColorBrush(System.Windows.Media.Colors.Blue);
}
private void radioButton5_CheckedChanged(object sender, EventArgs e)
{
    btn.Foreground = foregroundbrush;
}
private void radioButton6_CheckedChanged(object sender, EventArgs e)
{
    btn.Foreground = new System.Windows.Media.Solid
ColorBrush( System.Windows.Media.Colors.LightGreen);
}
private void radioButton4_CheckedChanged(object sender, EventArgs e)
{
    btn.Foreground = new
System.Windows.Media.SolidColorBrush(System.Windows.Media.Colors.Blue);
}
private void radioButton8_CheckedChanged(object sender, EventArgs e)
{
    btn.FontSize = 12;
}
private void radioButton9_CheckedChanged(object sender, EventArgs e)
{
    btn.FontSize = 14;
}
private void radioButton7_CheckedChanged(object sender, EventArgs e)
{
    btn.FontSize = 16;
}

```

代码 19-12 设置 Button 的属性

4. Tab 键和快捷键

WPF 和 WinForm 的互操作几乎不用考虑 Tab 键和快捷键的特殊处理,所有这些细节 WPF 和 WinForm 都会考虑,如代码 19-13 所示。

```

<Window x:Class="mumu_tabandMnemonics.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
Title="Tab 键和快捷键" Height="300" Width="300">
<StackPanel>
    <Button Click="cmdClicked" Margin="5">Use Alt+_A</Button>
    <WindowsFormsHost Grid.Row="1" Margin="5">
        <wf:Button Text="Use Alt+&B" Click="cmdClicked"></wf:Button>
    </WindowsFormsHost>
    <Button Grid.Row="2" Click="cmdClicked" Margin="5">Use Alt+_C</Button>
</StackPanel>
</Window>

```

代码 19-13 使用 Tab 键和快捷键进行 WPF 和 WinForm 的互操作

该 WPF 窗口中由 3 个按钮，中间一个是 WinForm 的按钮；上下两个是 WPF 的按钮。它们之间完全可以通过 Tab 键和 Shift+Tab 键切换焦点，也可以使用 Alt+A、Alt+B 和 Alt+C 快捷键触发按钮的单击事件处理函数。

19.4 在 Win32 中嵌入 WPF 内容

19.4.1 现有的 Win32 程序

Win32 程序对于绝大多数现在.NET 开发人员来说是一个古老的命题，我们首先介绍如何在 VS 2010 中建立一个 Win32 程序。

(1) 新建一个 Win32 空的项目，如图 19-13 所示（详见 mumu_clock 工程）。

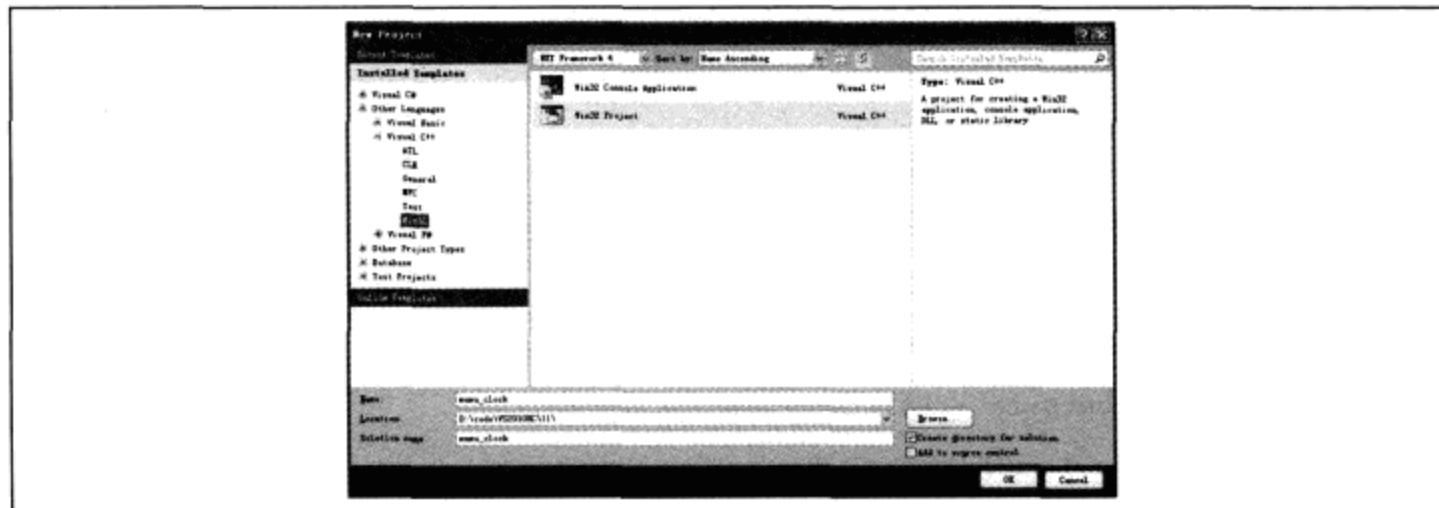


图 19-13 新建一个 Win32 空的项目

(2) 单击 OK 按钮，弹出向导对话框，如图 19-14 所示。

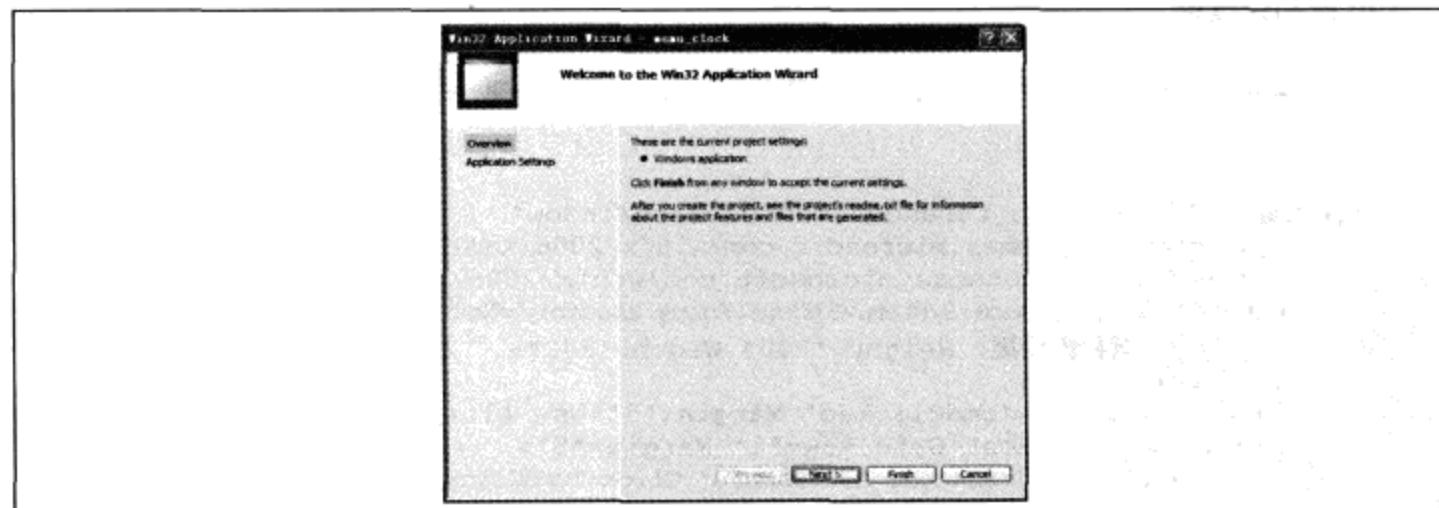


图 19-14 向导对话框

(3) 选择 Empty Project 复选框，如图 19-15 所示，然后单击 Finish 按钮。

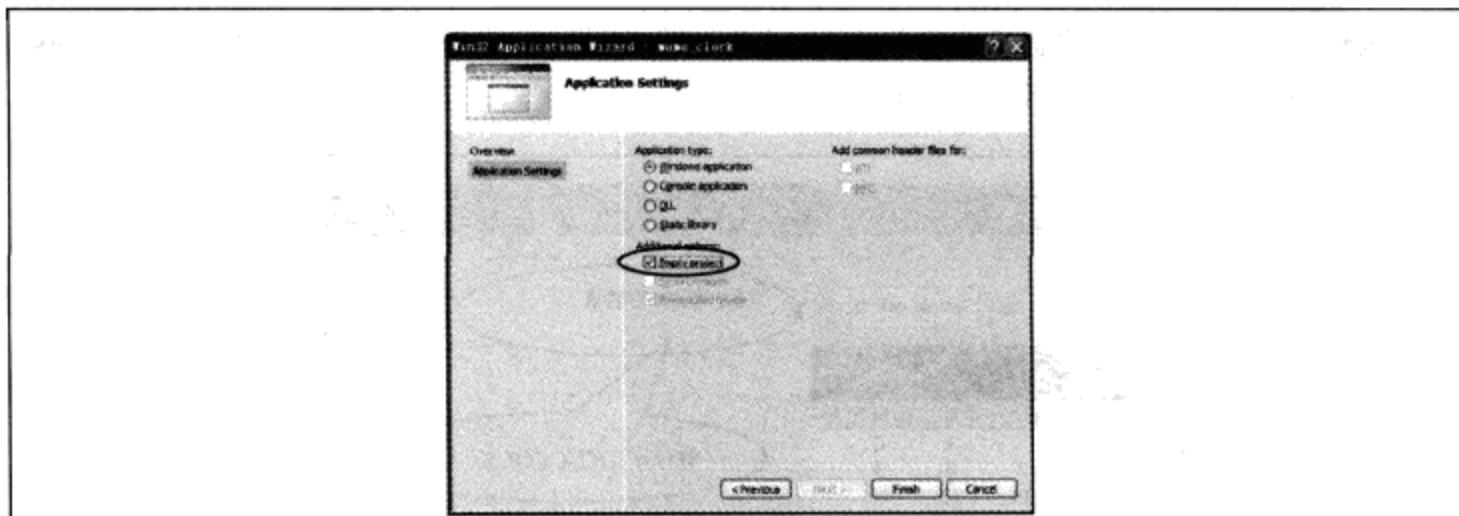


图 19-15 选择 Empty Project

(4) 这时只是新建一个空项目，为这个空项目添加一个对话框资源，如图 19-16 和图 19-17 所示。

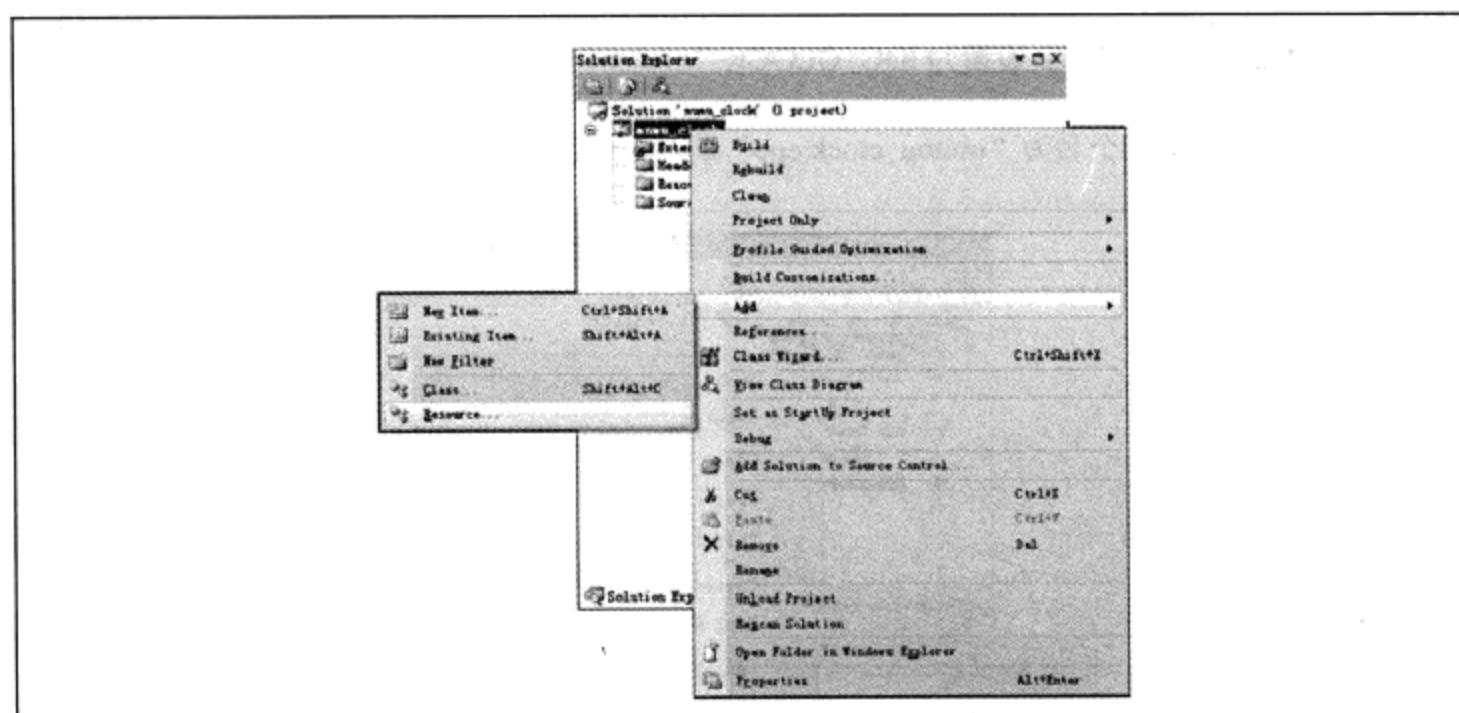


图 19-16 右键添加资源

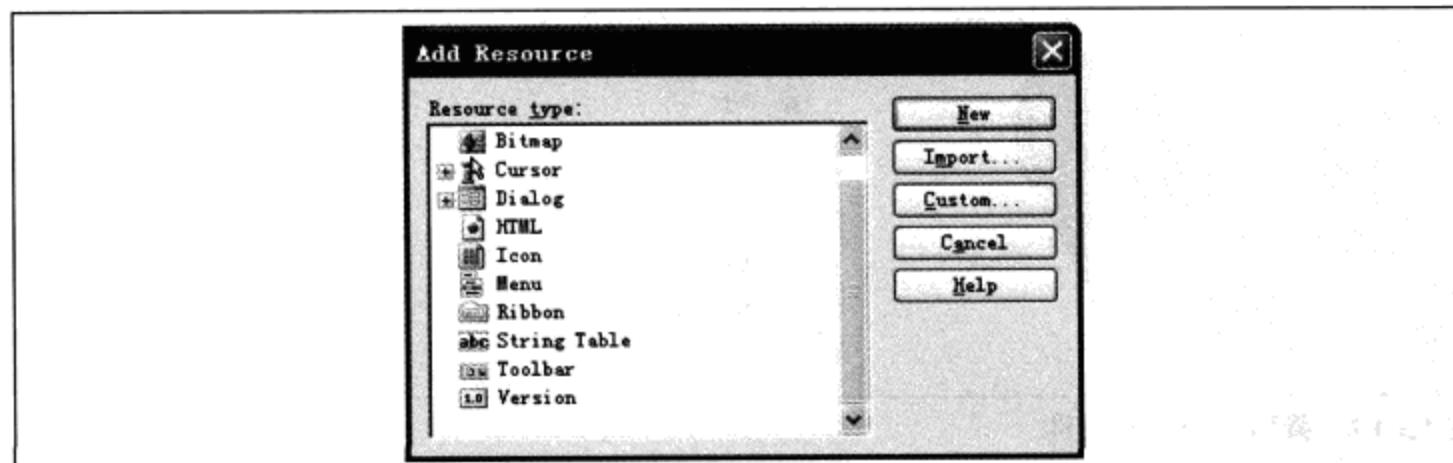


图 19-17 Add Resource 对话框

(5) 编辑新建的对话框，注意通过一个 Label 控件为后续插入 WPF 钟表留下一个位置。将该 Label 控件的 ID 设置为 IDC_CLOCK，如图 19-18 所示。

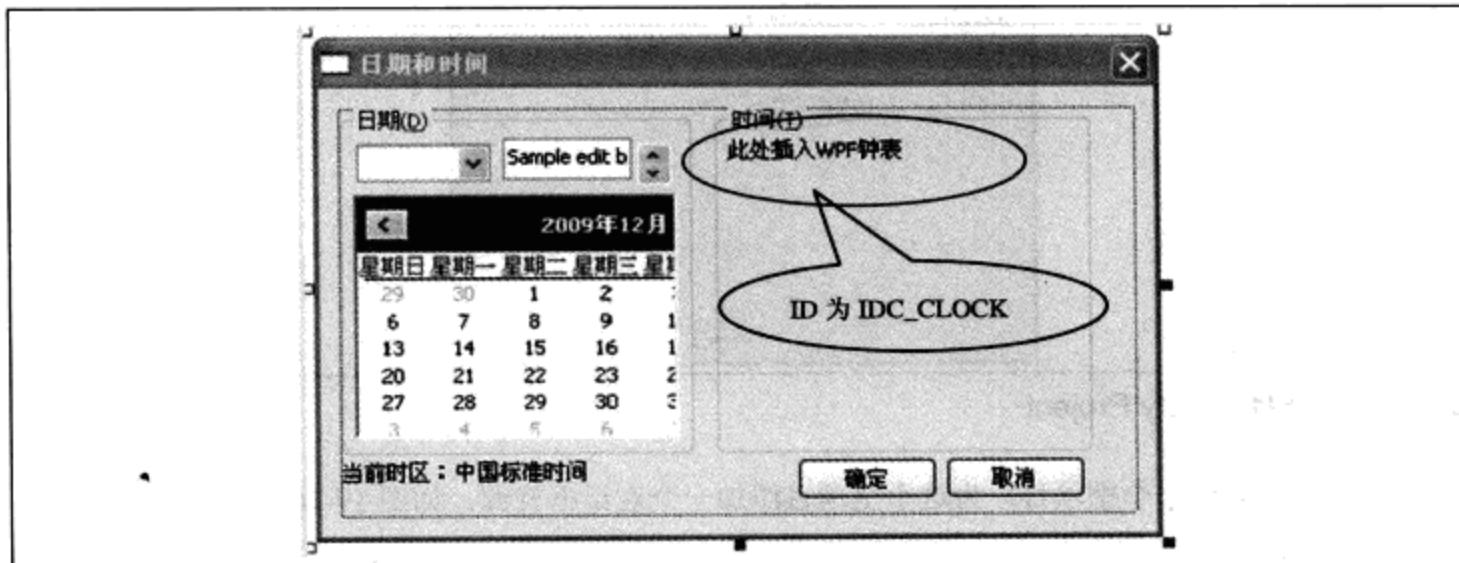


图 19-18 将 Label 控件的 ID 设置为 IDC_CLOCK

(6) 为这个项目添加一个名为“mumu_clock.cpp”的 cpp 文件，如图 19-19 和图 19-20 所示。

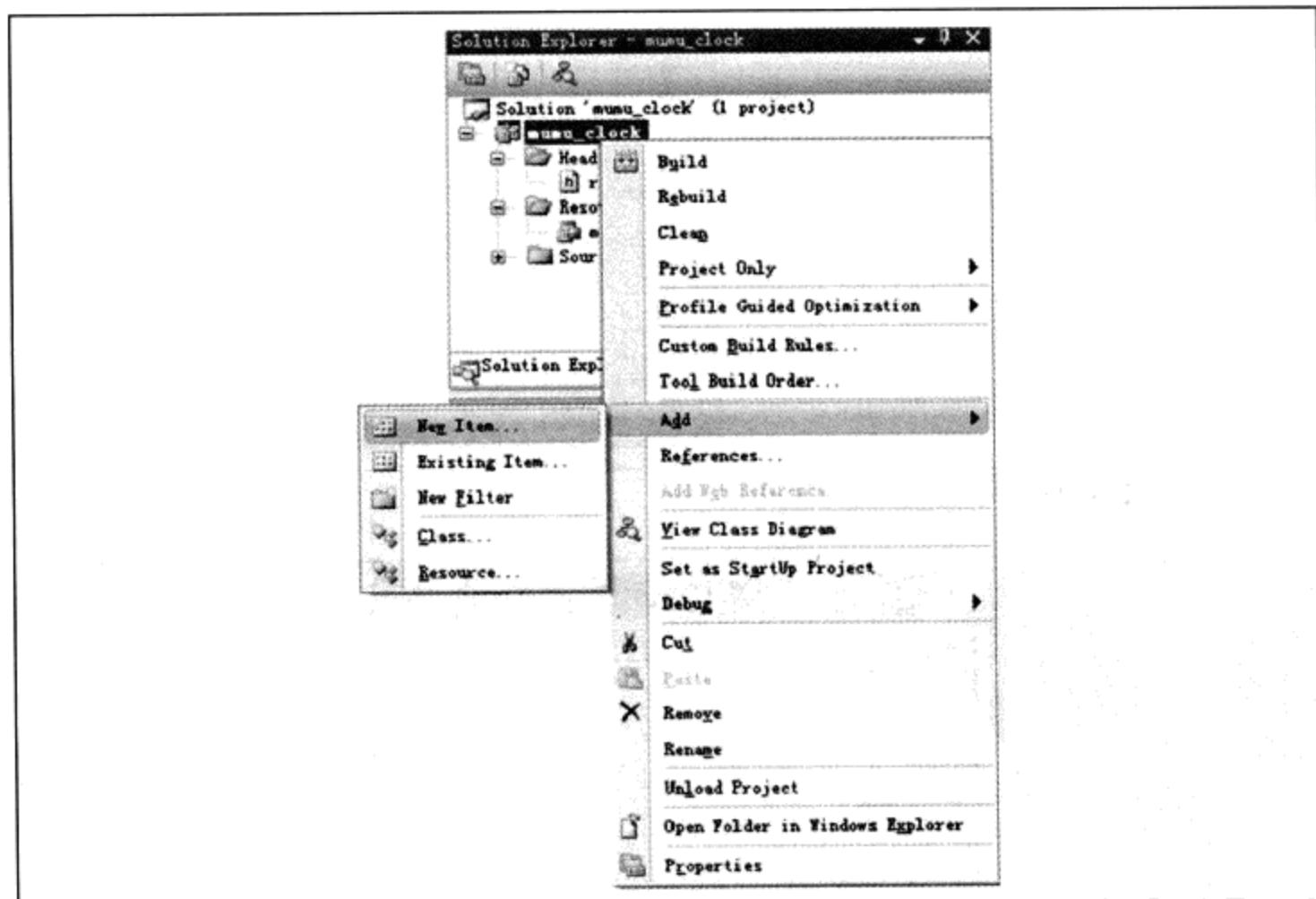


图 19-19 添加一个 cpp 文件



图 19-20 将 cpp 文件命名为“mumu_clock.cpp”

mumu_clock.cpp 文件需要包含两个头文件，一个是 windows.h，所有的 Windows 应用程序都需要包含这个文件；另外一个是 resource.h，该文件是对资源对话框的各种控件 ID 的定义。在 WinMain 函数中，只是简单地调用该对话框。About 是对话框的窗口过程函数，在其中处理 WM_COMMAND 和 WM_CLOSE 两个消息。在 WM_COMMAND 消息中，如果用户单击了“确定”或者“取消”按钮，则对话框会结束。如果对话框收到 WM_CLOSE 消息，也会关闭对话框，如代码 19-14 所示。

```
#include "resource.h"
#include <windows.h>
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
    DialogBox(hInstance, (LPCTSTR)IDD_DATETIMEDLG, NULL, (DLGPROC)About);
    return 0;
}

LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_INITDIALOG:
        {
            return TRUE;
        }
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
        break;
    case WM_CLOSE:
        {
            EndDialog(hDlg, LOWORD(wParam));
            return TRUE;
        }
    }
```

```
        break;  
    }  
    return FALSE;  
}
```

代码 19-14 mumu_clock.cpp 文件的源代码

运行程序结果如图 19-21 所示。

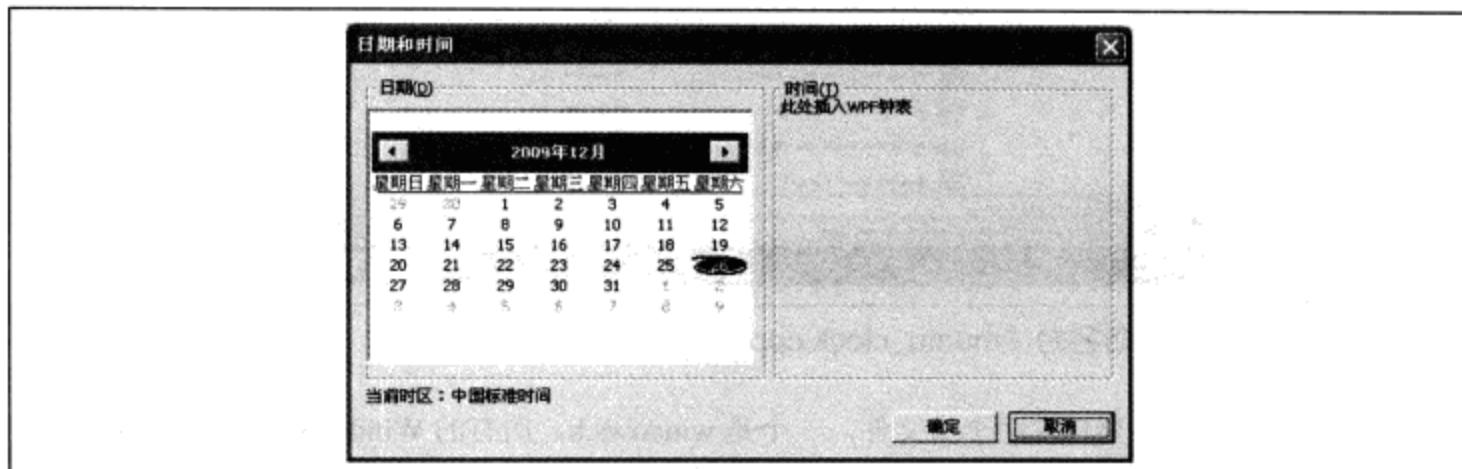


图 19-21 程序运行结果

下一步使用 WPF 制作钟表。

19.4.2 使用 WPF 制作钟表

为了便于调试和测试，一个小的技巧是可以新建一个普通的 WPF 窗口，程序名为“WPFClock”。然后在该程序中实现所需功能，如制作钟表控件。最后删除 App.xaml 和 App.xaml.cs 文件，并将应用程序输出类型从 Windows Application 改成 Class Library。这样应用程序可以变为一个 Dll，并嵌入在 Win32 程序中，如图 19-22 所示。

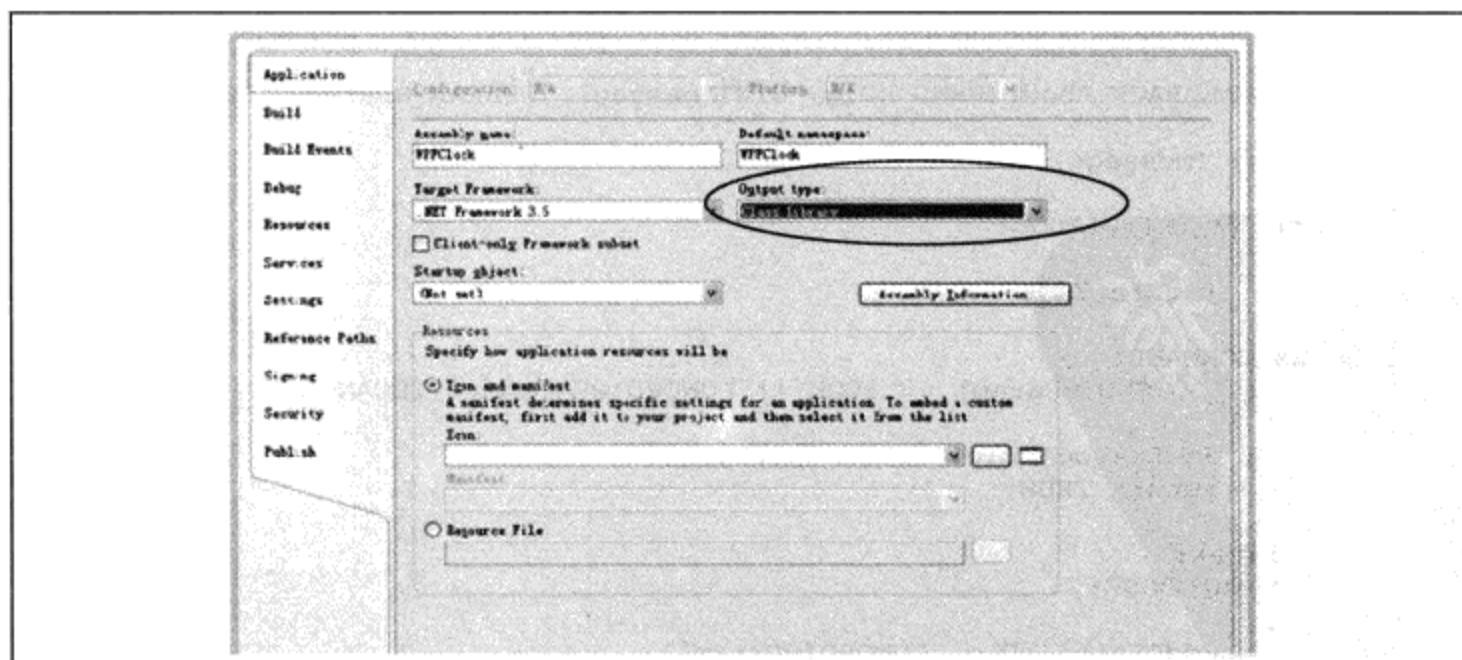


图 19-22 输出类型从 Windows Application 改成 Class Library

WPF 程序制作两种类型的钟表，一是传统的钟表；二是一个数字钟表。数字钟表使用 3D 数字，这样数字每一次变化均是一个 3D 几何变换。如图 19-23 所示，具体的实现代码将不再详述。

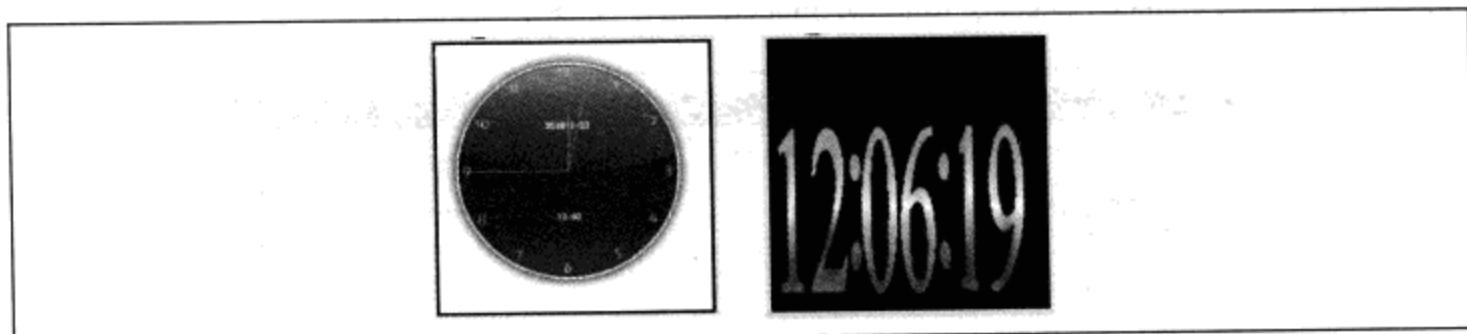


图 19-23 WPF 程序制作的两种类型的钟表

19.4.3 将 WPF 内容嵌入在 Win32 程序中

将 WPF 内容嵌入在 Win32 程序中，第 1 道鸿沟是托管与非托管代码之间的鸿沟。Win32 程序传统上是一个非托管工程，而 WPF 程序是一个托管工程，弥补这道鸿沟的方法是将 Win32 程序设置为托管工程。

1. 设置 Win32 工程

(1) 设置对 CLR 的支持

在解决方案窗口中选中 Win32 工程，右键选择 Properties 选项，如图 19-24 所示。

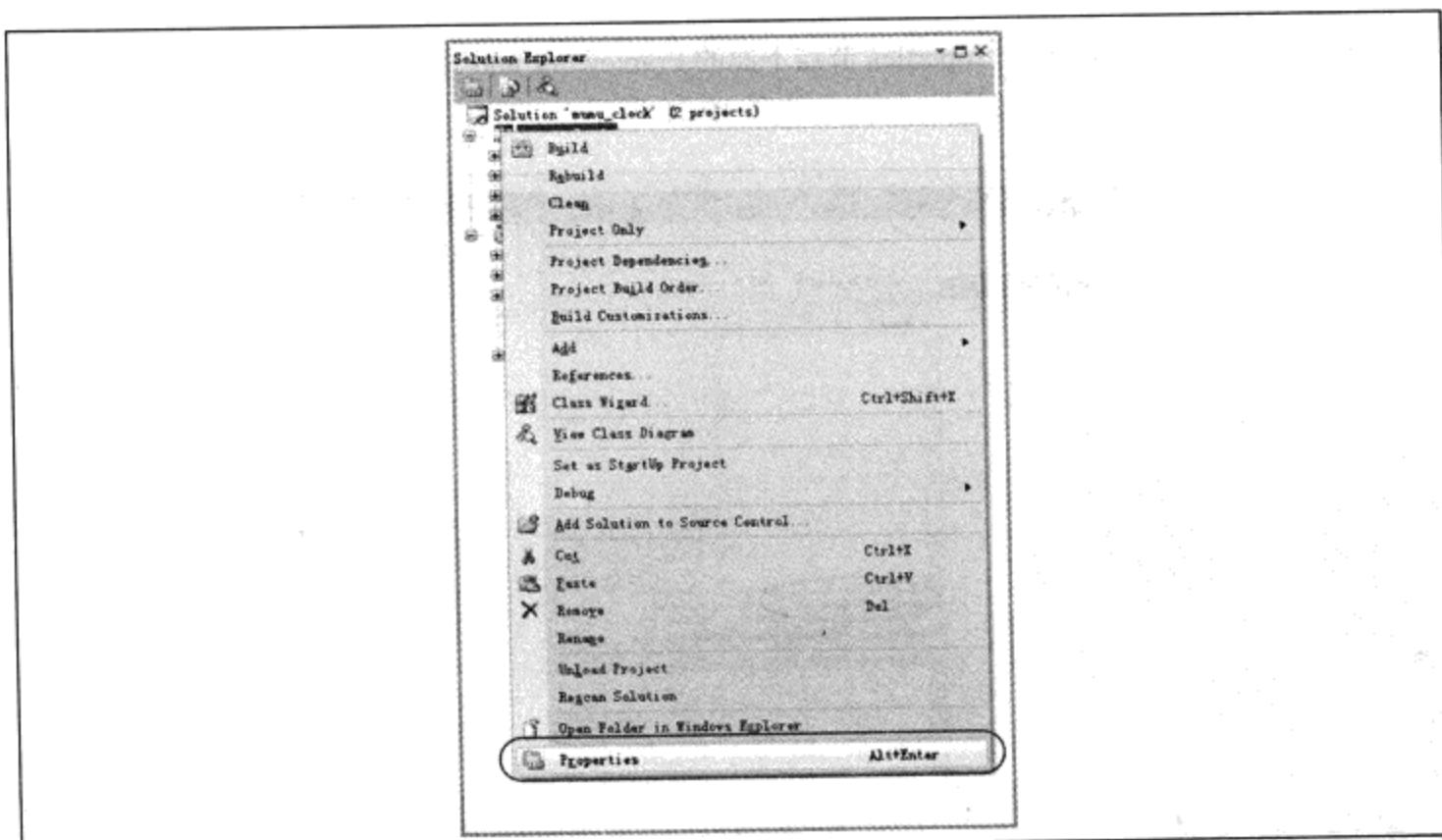


图 19-24 Win32 工程的属性

弹出该工程的属性对话框，选择左侧树控件中的 General 选项。右侧列表上会有一项 Common Language Runtime support(CLR 支持)，默认为 No Common Language Runtime support(不支持 CLR)。将该选项设置为 Common Language Runtime Support (/clr)，如图 19-25 所示。

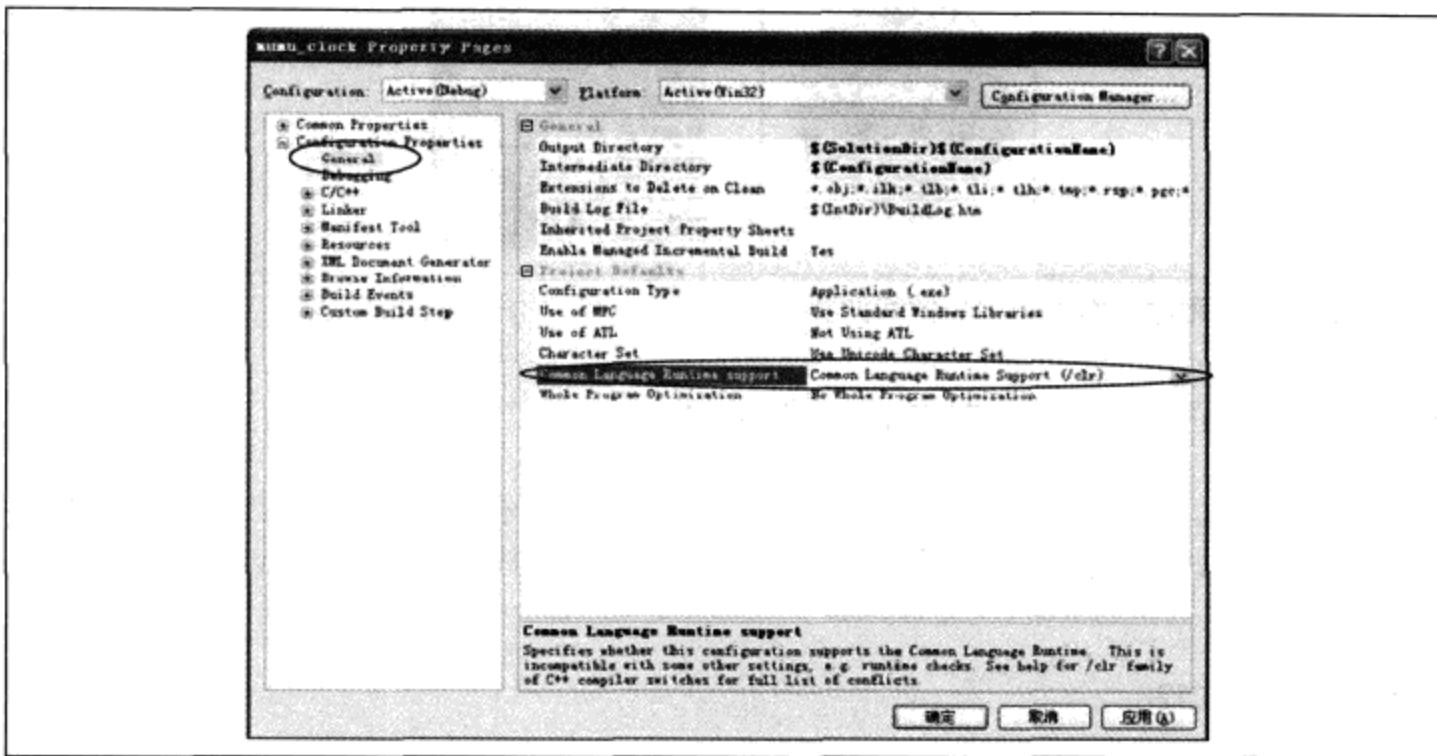


图 19-25 General 选项设置为 Common Language Runtime Support (/clr)

(2) 添加相关引用

在左侧树控件中选中 Common Properties 节点下面的 Framework and References，单击右侧面板中的 Add New Reference（添加新的引用）按钮，如图 19-26 所示。

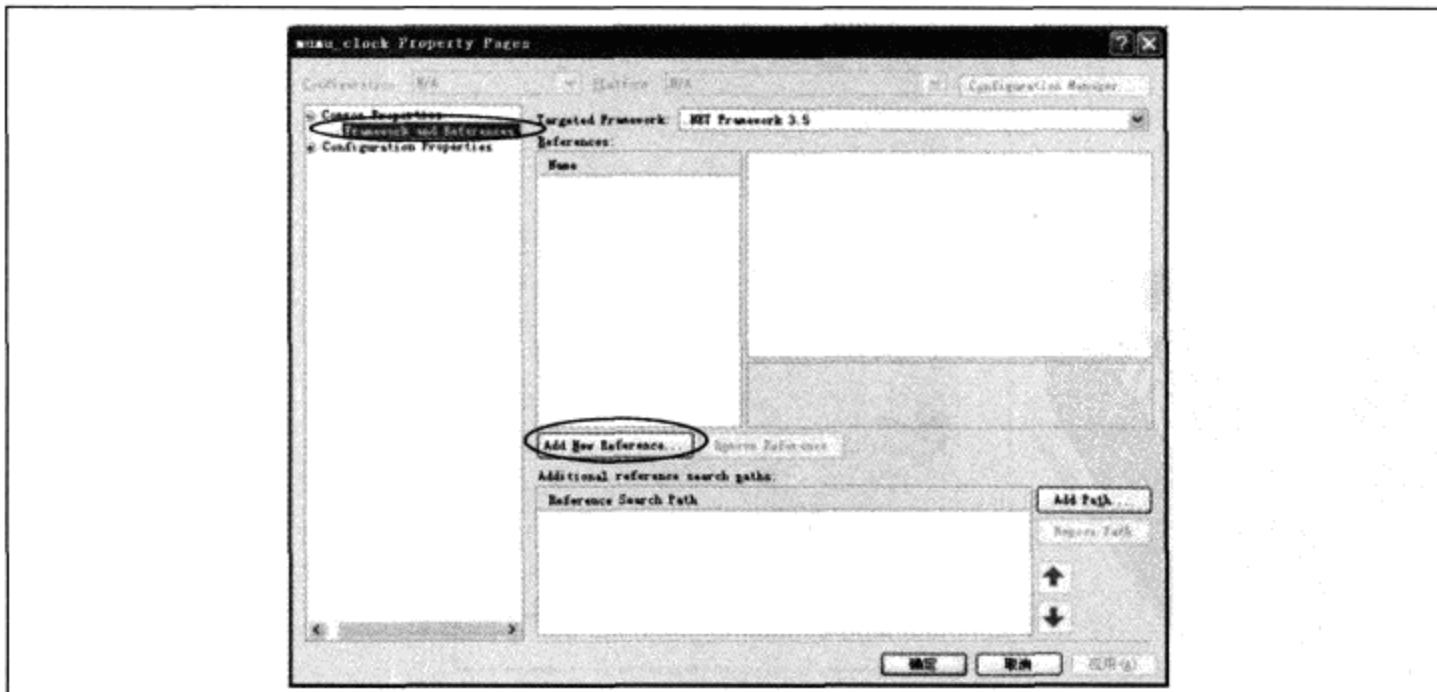


图 19-26 Add New Reference 按钮

需要在弹出的添加引用对话框中增加 System、PresentationCore、PresentationFramework、WindowsBase、UIAutomationTypes、UIAutomationProvider，以及 WPFClock 引用。

实际上只有设置对 CLR 的支持之后才能添加这些引用，也可以尝试重新恢复成不支持 CLR。然后单击 Add New Reference 按钮，则无法看到.NET 和 Com 等选项卡，如图 19-27 所示。

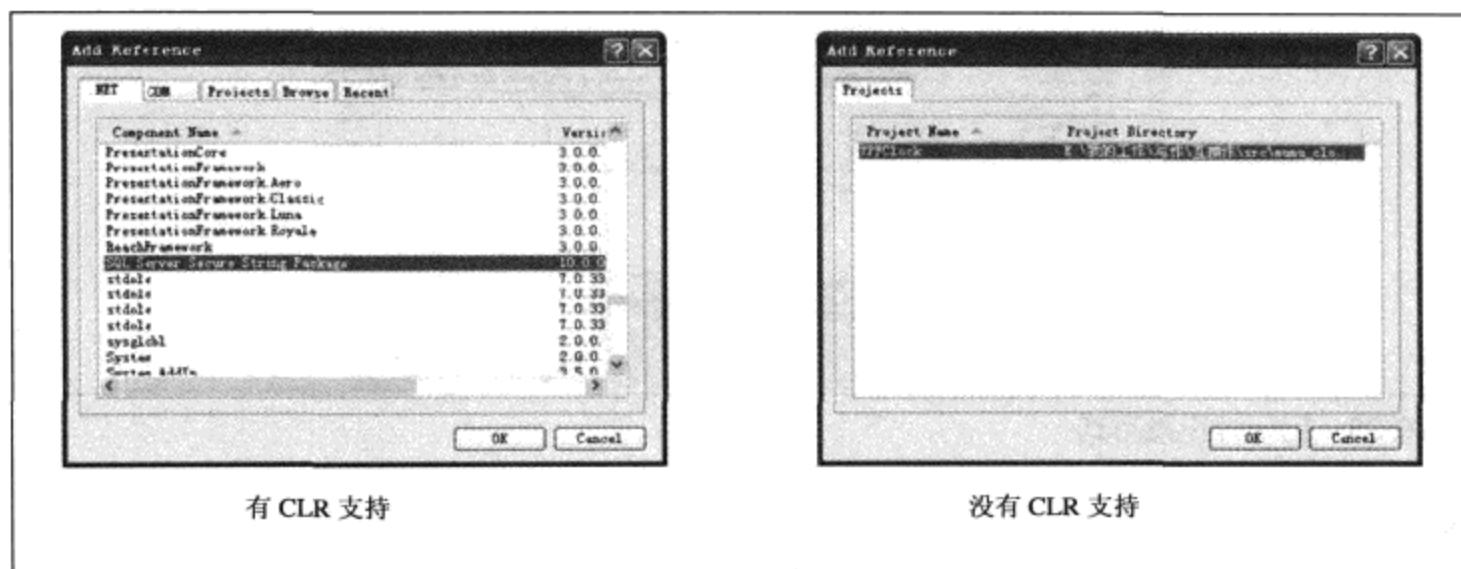


图 19-27 对比有 CLR 支持和没有 CLR 支持

(3) 设置线程模型

由于 WPF 程序需要在 STA 线程模型下运行，因此在 Win32 程序中要通过设置告诉 CLR 当初始化 WPF 的内容时需要使用 STA 线程模型。这种设置有两种方式，一种是直接在 WinMain 函数前面添加[System::STAThreadAttribute]语句，如代码 19-15 所示。

```
[System::STAThreadAttribute]
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
....
```

代码 19-15 第 1 种设置方式

另外一种方式是在工程的属性对话框中设置，选择 mumu_clock 工程。右键选择 Properties 选项，弹出该工程的属性对话框。选择左侧 Linker 节点下的 Advanced 选项，右侧下拉列表框中的 CLR Thread Attribute 选项的默认设置是 No threading attribute set。打开下拉列表框，选择 STA threading attribute 选项，这种方式比较适合在工程中没有 WinMain 函数的情况。如 MFC 的工程，如图 19-28 所示。

2. 使用 HWndSource

在 Win32 程序中所有的控件都是窗口，HWndSource 起的作用是将 WPF 的内容赋值在其根节点上，然后它为外部 Win32 程序提供一个 HWND 句柄。

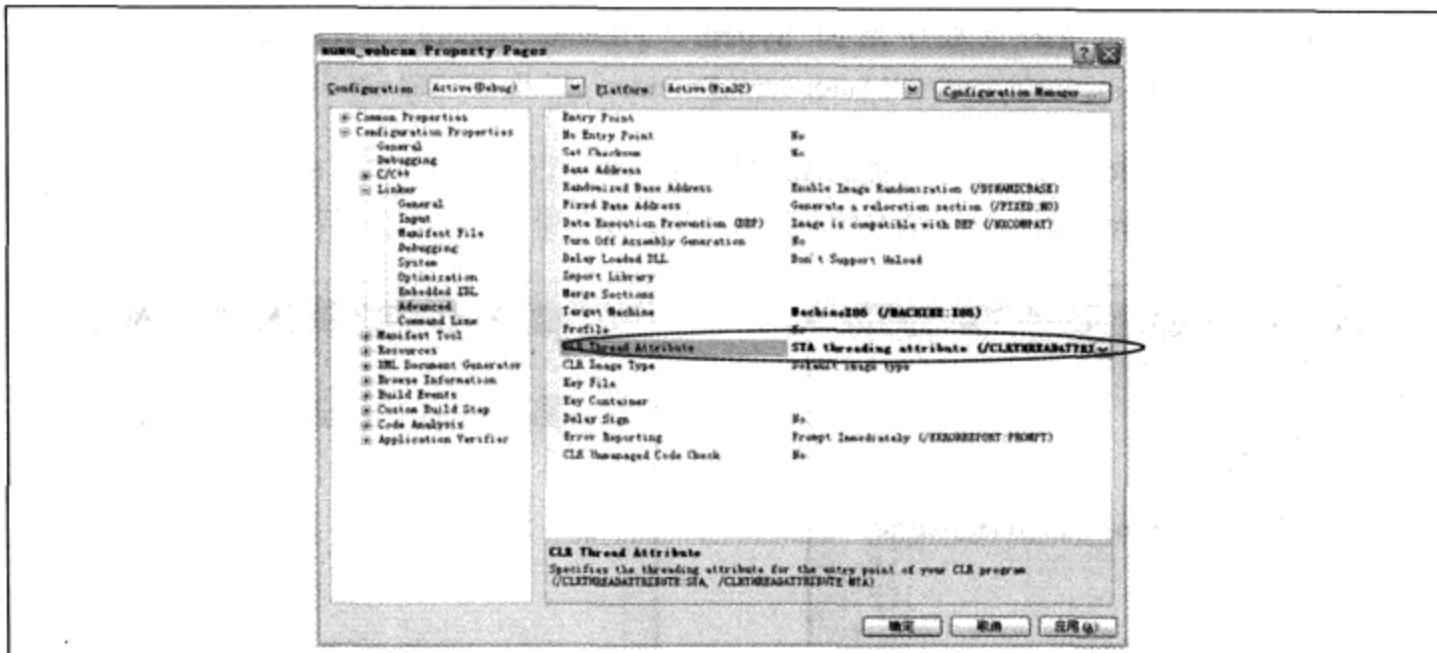


图 19-28 另一种设置方式

代码 19-16 中的①声明命名空间；第②行自定义一个函数 GetHwnd。该函数创建 WPF 的内容，最后返回一个 HWND 句柄；第③行创建 HWndSource 对象，其构造函数非常类似 Win32 中创建窗口的函数 CreateWindow；第④行是创建 WPF 的内容；第⑤行将 WPF 的内容赋值给 HWndSource 的根节点；第⑥行通过 HWndSource 的 Handle 返回窗口句柄。

```

namespace ManagedCode
{
    ①    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Media;
    ②    HWND GetHwnd(HWND parent, int x, int y, int width, int height)
    {
        ③        HwndSource^ source = gcnew HwndSource(
            0, // class style
            WS_VISIBLE | WS_CHILD, // style
            0, // exstyle
            x, y, width, height,
            "hi", // 
            IntPtr(parent) // parent window
        );
        ④        UIElement^ page = gcnew WPFClock::AnalogClock();
        ⑤        source->RootVisual = page;
        ⑥        return (HWND) source->Handle.ToPointer();
    }
}

```

代码 19-16 Win32 程序

3. 将 WPF 的内容摆放在合适的位置

WPF 内容将在 WM_INITDIALOG 消息中创建，我们在 Win32 工程中用一个标签控件标识 WPF 内容需要插入的位置。如代码 19-17 所示，第①行通过 GetDlgItem 找到该标签控件的 HWND 句柄；第②行计算该标签的外接矩形的位置和大小，以便将 WPF 内容放置在该矩形的位置上；第④行通过

前面我们写的 GetHwnd 方法创建 WPF 内容。

```
switch (message)
{
    case WM_INITDIALOG:
        {
            HWND placeholder = GetDlgItem(hDlg, IDC_CLOCK);
            int result;
            RECT rectangle;
            GetWindowRect(placeholder, &rectangle);
            int width = rectangle.right - rectangle.left;
            int height = rectangle.bottom - rectangle.top;
            POINT point;
            point.x = rectangle.left;
            point.y = rectangle.top;
            result = MapWindowPoints(NULL, hDlg, &point, 1);
            ShowWindow(placeholder, SW_HIDE);
            HWND clock = ManagedCode::GetHwnd(hDlg, point.x, point.y, width,
                height);
            return TRUE;
        }
        ....
```

代码 19-17 标识 WPF 内容需要插入的位置

由于 WPF 内容有固定的大小，所以有时放置在 Win32 程序里可能会出现大小不一致的情况，如图 19-29 所示。



图 19-29 WPF 内容在 Win32 程序中大小改变

为了能够让 WPF 内容能够根据外部空间安排的大小自动调整其大小，我们为 AnalogClock 添加一个 SetDesiredWidthAndHeight。该方法会根据外部空间的大小缩放 WPF 的内容，以适应外部空间，如代码 19-18 所示。

```
AnalogClock.xaml.cs
public void SetDesiredWidthAndHeight(int width, int height)
{
    scale.CenterX = 0.5;
    scale.CenterY = 0.5;
    scale.ScaleX = width / (this.Width + 30);
    scale.ScaleY = width / (this.Height + 30);
}
```

代码 19-18 让 WPF 内容根据外部空间安排的大小自动调整其大小

scale 是一个缩放变换，它在对应的 XAML 文件中定义，如代码 19-19 所示。

```
AnalogClock.xaml
<Grid>
    .....
    <Grid.RenderTransform>
        <ScaleTransform x:Name="scale" />
    </Grid.RenderTransform>
</Grid>
```

代码 19-19 在 XAML 中定义 ScaleTransform

这样在创建 WPF 内容之后需要调用该方法，如代码 19-20 所示。

```
mumu_clock.cpp
HWND GetHwnd(HWND parent, int x, int y, int width, int height)
{
    .....
    UIElement^ page = gcnew WPFClock::AnalogClock();
    WPFClock::AnalogClock^ analogclock = (WPFClock::AnalogClock^)page;
    analogclock->SetDesiredWidthAndHeight(width,height);
    source->RootVisual = page;
    return (HWND) source->Handle.ToPointer();
}
```

代码 19-20 调用 SetDesiredWidthAndHeight 方法

再次运行程序可以得到大小合适的日期和时间对话框，同样还可以为时钟添加一种数字钟表的形式。它们都在 WM_INITDIALOG 消息中创建，但是只显示其中之一。当需要切换时显示另一个，如图 19-30 所示。

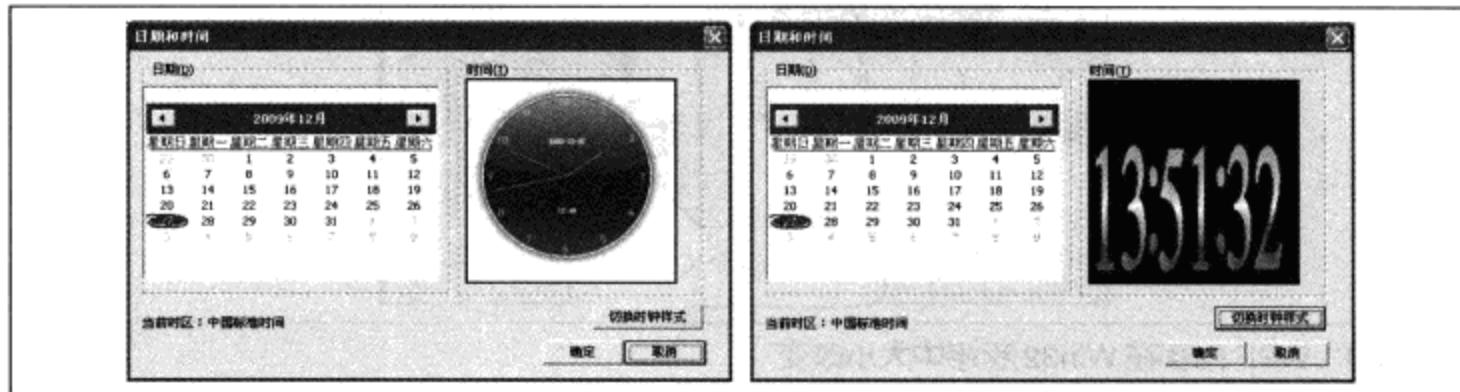


图 19-30 大小合适的日期和时间对话框

19.5 在 WPF 中嵌入 Win32 内容

19.5.1 一个 Win32 的 DLL 工程

1. 新建工程

(1) 新建一个空的 Win32 工程，将其选择为 DLL 工程，如图 19-31 所示。

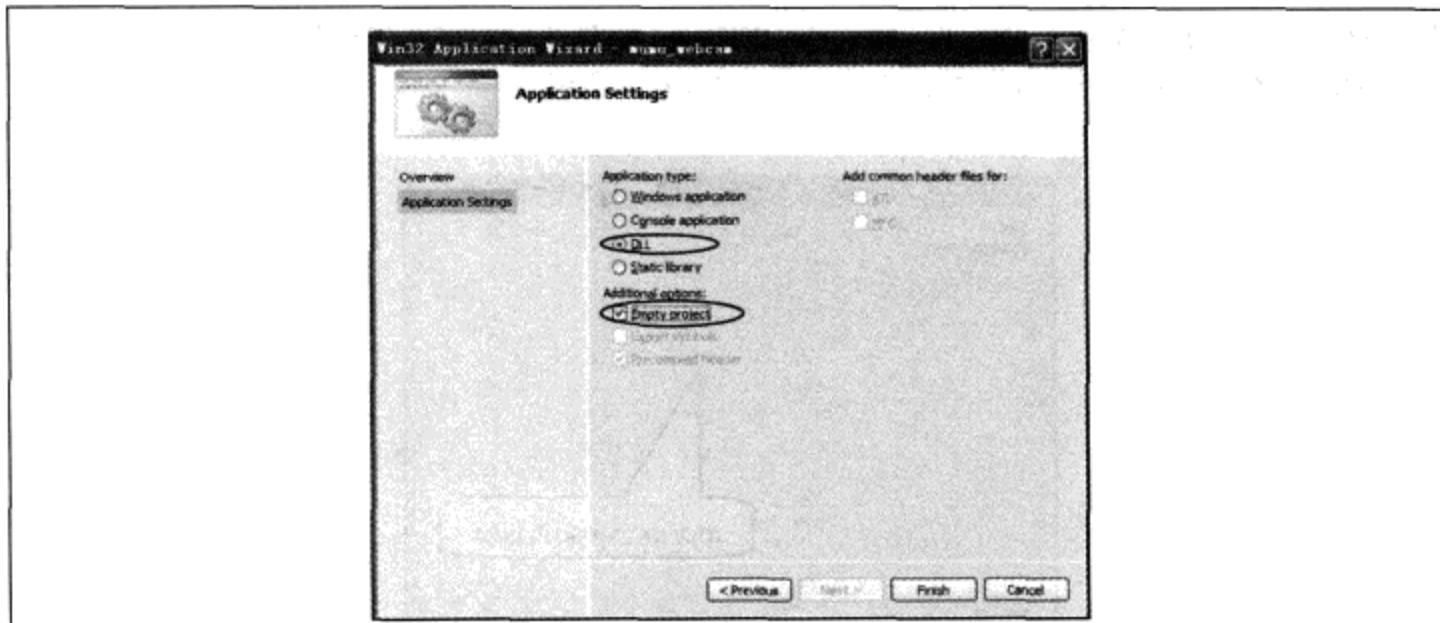


图 19-31 新建 DLL 工程

(2) 为该工程添加如表 19-1 所示的文件和资源。

表 19-1 添加的文件和资源

文件名	描述
mumu_webcam.cpp	DllMain 函数
resource.h	添加了一个对话框资源，则会自动添加这两个文件
mumu_webcam.rc	
stdafx.h	对 windows.h 和 resource.h 等这类公共文件的引用

代码 19-21 所示为 mumu_webcam.cpp 文件。由于该工程是个动态链接库工程，因此程序的入口不再是 WinMain，而是 DllMain。

```
#include "stdafx.h"
HINSTANCE hInstance;
BOOL APIENTRY DllMain( HINSTANCE hInst,
                       DWORD ul_reason_for_call,
                       LPVOID lpReserved
                      )
{
    hInstance = hInst;
    return TRUE;
}
```

代码 19-21 mumu_webcam.cpp 文件

代码 19-22 所示为 stdafx.h 文件，其中包含 windows.h 和 resource.h 文件。

```
#pragma once

#include <windows.h>
#include "resource.h"
```

代码 19-22 stdafx.h 文件

在资源文件中添加一个资源 ID 为 IDD_VIDEOODLG 的对话框，在其中央位置放置一个 ID 为 IDC_VIDEO 的 Label 作为摄像头渲染的窗口，如图 19-32 所示。

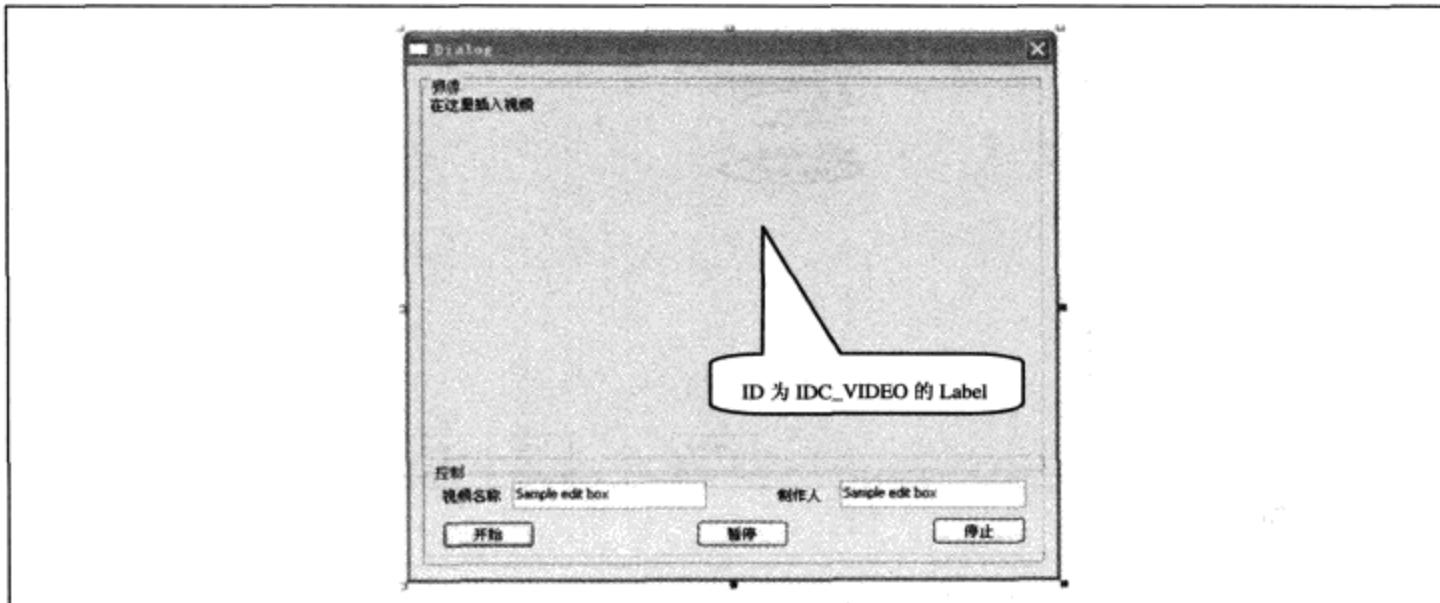


图 19-32 资源 ID 为 IDD_VIDEOODLG 的对话框

由于该对话框最终作为一个控件嵌入到 WPF 的应用程序中，因此需要修改其样式。为此选中 `mumu_webcam.rc` 文件，右键选择 View Code 选项，对话框的默认样式如代码 19-23 所示。

```
IDD_VIDEOODLG DIALOGEX 0, 0, 350, 266
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION |
WS_SYSMENU
```

代码 19-23 对话框的默认样式

将 `STYLE` 修改成下面的代码：

```
STYLE DS_SETFONT | WS_CHILD | WS_BORDER | DS_CONTROL
```

代码 19-24 修改 `STYLE`

2. 添加 WebCam 类

为该工程添加一个 `WebCam` 类，使用非托管的 C++ 封装 DirectShow 的 Com 对象，可以控制网络摄像头。它主要由多个静态方法组成，如代码 19-25 所示。

```
#if !defined(WEBCAM_H)
#define WEBCAM_H
#include <wtypes.h>
class Webcam
{
public:
    // 初始化网络摄像头
    static HRESULT Initialize(int width, int height);
    // 和某一个窗口连接起来
    static HRESULT AttachToWindow(HWND hwnd);
    // 摄像头开始工作
    static HRESULT Start();
```

```

// 摄像头暂停工作
static HRESULT Pause();
// 摄像头停止工作
static HRESULT Stop();
// 重绘画面
static HRESULT Repaint();
// 退出时释放所有 Com 对象
static HRESULT Terminate();
// 获得工作窗口的宽度
static int GetWidth();
// 获得工作窗口的高度
static int GetHeight();
};

#endif // !defined(WEBCAM_H)

```

代码 19-25 添加一个 WebCam 类

注意使用 Webcam 类需要引用到两个 Lib 库，即 quartz.lib 和 strmiids.lib，并且在工程的属性对话框中设置。如图 19-33 所示，选中左侧的 Linker 节点下的 Input 选项，在右侧 Additional Dependencies 列表框中添加这两个 Lib 库。

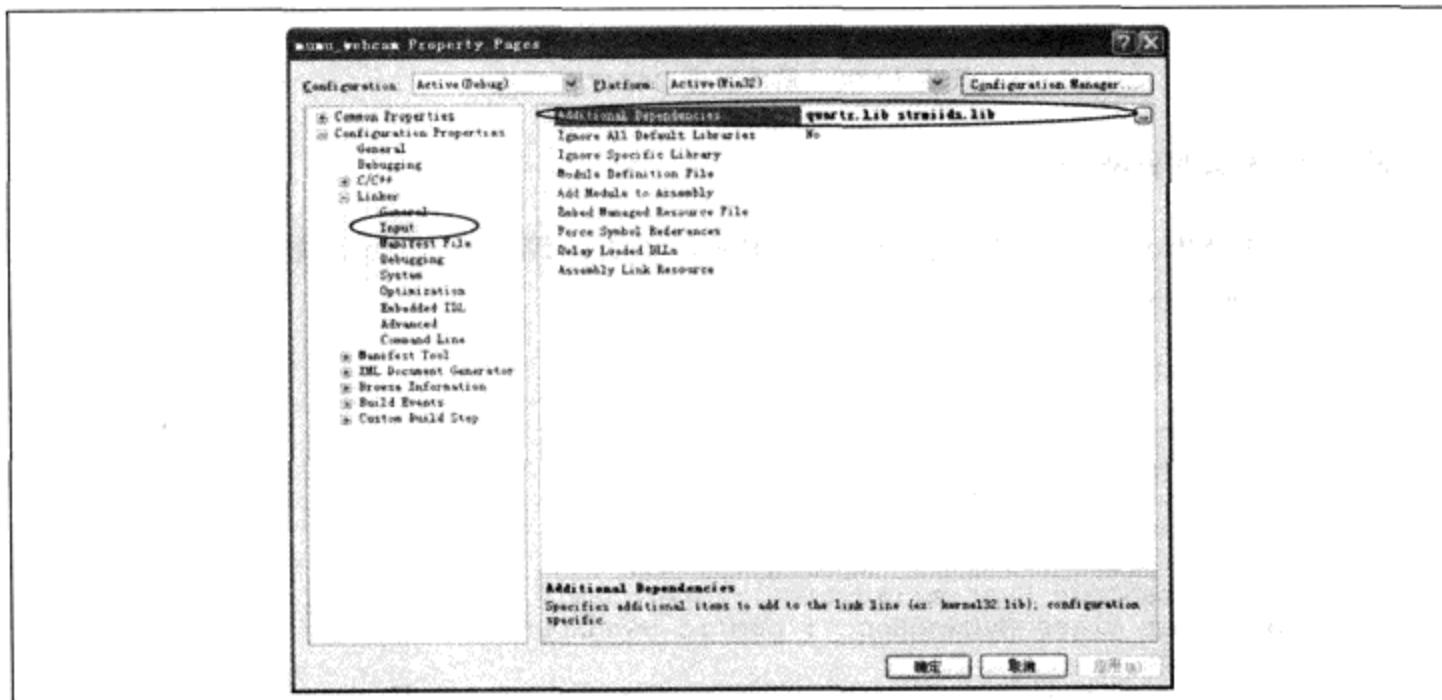


图 19-33 在 Additional Dependencies 列表框中添加两个 Lib 库

19.5.2 使用 HwndHost

在 Win32 中嵌入 WPF 内容，我们使用的是 HWndSource。与之相反，在 WPF 中嵌入 win32 内容需要借助另一个类型 Hwndhost。因为 HwndHost 派生自 FrameworkElement，因此它可以将 Win32 中的一个窗口封装成 WPF 中普通的元素。使用 HwndHost 非常简单，就是继承该类。然后实现两个关键方法，即 BuildWindowCore 和 DestroyWindowCore。不过在这之前需要设置已有的 DLL 工程，使其支持 CLR。

1. CLR 支持

和前面在 Win32 中嵌入 WPF 内容一样，该动态链接库工程也需要设置支持 CLR。一个问题是设置工程支持 CLR 之后，重新编译工程会报出警告 C4747，大致意思是 DLL 的入口函数不能运行在托管代码上。为解决这个问题，可以通过预编译命令让 DLL 的入口函数不运行在托管代码上，如代码 19-26 所示。

```
#ifdef _MANAGED
#pragma managed(push, off)
#endif
HINSTANCE hInstance;
BOOL APIENTRY DllMain( HINSTANCE hInst,
                       DWORD ul_reason_for_call,
                       LPVOID lpReserved
                      )
{
    hInstance = hInst;
    return TRUE;
}
#endif
#pragma managed(pop)
#endif
```

代码 19-26 解决警告 C4747 的问题

2. 实现一个新类 MyHwndHost

现在可以从 HwndHost 派生一个新类 MyHwndHost，在 BuildWindowCore 函数中创建对话框。在 DestroyWindowCore 销毁对话框，如代码 19-27 所示。

```
namespace ManagedCpp
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Input;
    using namespace System::Windows::Media;
    using namespace System::Runtime::InteropServices;
    public ref class MyHwndHost : public HwndHost{
protected:
    virtual HandleRef BuildWindowCore(HandleRef hwndParent) override {
        HWND dialog = CreateDialog(hInstance,
                                  MAKEINTRESOURCE(IDD_DIALOG1),
                                  (HWND) hwndParent.Handle.ToPointer(),
                                  (DLGPROC) About);
        return HandleRef(this, IntPtr(dialog));
    }
    virtual void DestroyWindowCore(HandleRef hwnd) override {
        ::DestroyWindow((HWND) hwnd.Handle.ToInt32());
    }
};
```

代码 19-27 从 HwndHost 派生一个新类 MyHwndHost

代码 19-28 是对话框的窗口过程函数 About 的实现，在 WM_INITDIALOG 消息中初始化摄像头，并指定渲染窗口；在 WM_COMMAND 消息里处理开始、暂停和停止按钮；在 WM_DESTROY 消息

中释放了所有 Com 对象。

```
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hwnd;
    RECT rect;
    switch (message)
    {
        .
        case WM_INITDIALOG:
            hwnd = ::GetDlgItem(hDlg, IDC_VIDEO);
            ::GetWindowRect(hwnd, &rect);
            if (FAILED(Webcam::Initialize((rect.right - rect.left), (rect.bottom - rect.top))))
                ::MessageBox(NULL, L"初始化摄像头设备失败", L"错误", 0);
            Webcam::AttachToWindow(hwnd);
            Webcam::Start();
            return (INT_PTR)TRUE;

        case WM_COMMAND:
            if (LOWORD(wParam) == ID_START)
            {
                Webcam::Start();
            }
            else if (LOWORD(wParam) == ID_STOP)
            {
                Webcam::Stop();
            }
            else if (LOWORD(wParam) == ID_PAUSE)
            {
                Webcam::Pause();
            }
            break;
        case WM_DESTROY:
            Webcam::Terminate();
            break;
    }
    return (INT_PTR)FALSE;
}
```

代码 19-28 对话框的窗口过程函数 About

3. 将 MyHwndHost 嵌入 WPF 程序中

使用 MyHwndHost 和使用其他 WPF 元素没有任何区别。我们只需要添加 Win32 工程的引用。然后在 XAML 文件中声明这个 Dll，就可以将 MyHwndHost 添加到 WPF 窗口中，如代码 19-29 所示。

```
<Window x:Class="WpfApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:ManagedCpp;assembly=mumu_webcam"
SizeToContent="WidthAndHeight"
    Title="摄像程序" >
    <Grid>
        <c:MyHwndHost />
    </Grid>
</Window>
```

代码 19-29 将 MyHwndHost 添加到 WPF 窗口中

运行程序，可以看到 Win32 窗口已经嵌入到 WPF 程序中，如图 19-34 所示。



图 19-34 摄像程序运行结果

19.5.3 支持键盘导航

目前我们只是将 Win32 控件显示在 WPF 的应用程序上，还应该支持一些常规特性，如键盘导航。为了说明这个问题，我们将 WPF 程序略加修改，如图 19-35 所示。

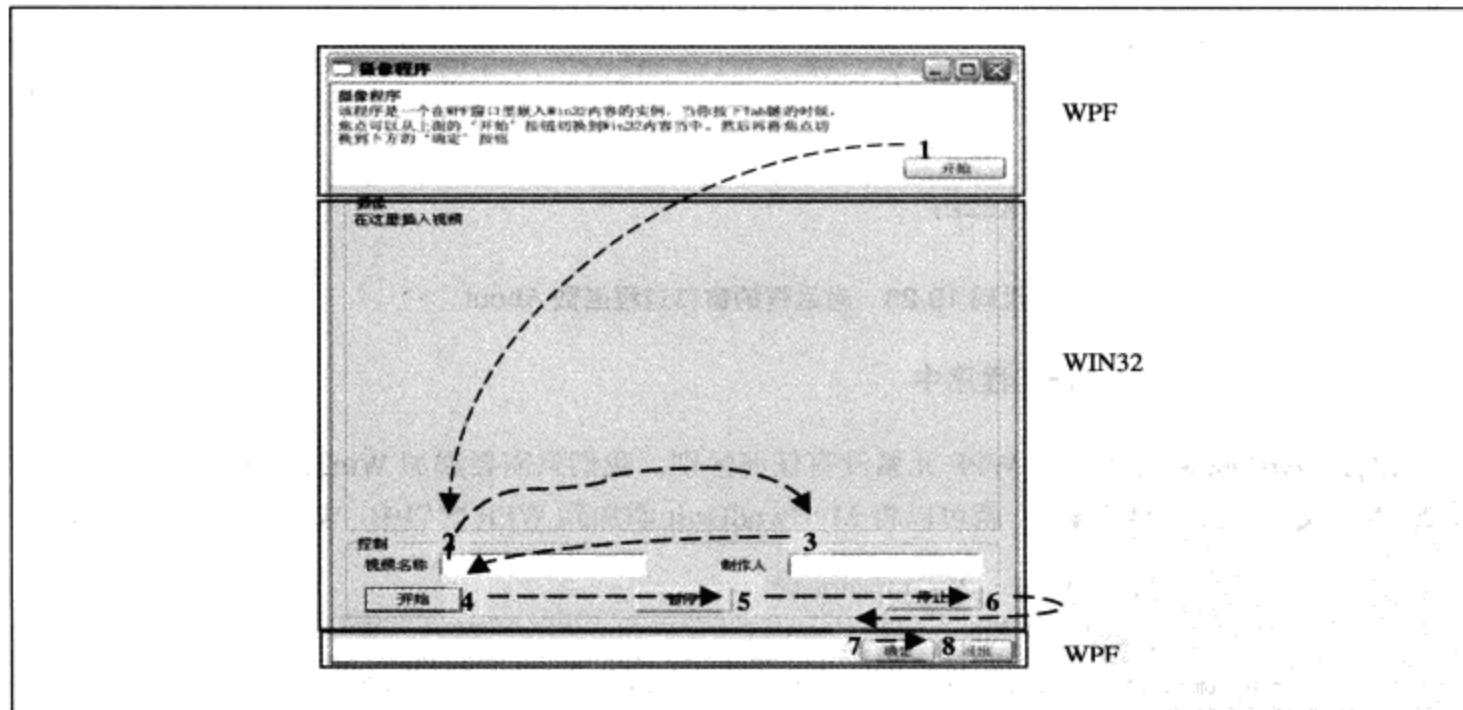


图 19-35 Tab 键的顺序

如图 19-35 所示，整个应用程序由 WPF 和 Win32 元素混合组成。我们希望按下 Tab 键，1 号按钮首先获得焦点。再次按下 Tab 键时，进入 Win32 窗口，其中的 2 号编辑框获得焦点。继续按 Tab 键，焦点依次从 3 到 4、到 5、到 6。再次按下焦点离开 Win32 窗口，进入下方的 WPF 元素中，7 号按

钮获得焦点。再按下 Tab 键，则最终 8 号按钮获得焦点，如此反复循环。当然如果按下 Shift+Tab 键，那么整个顺序与前面描述的顺序相反。

1. 设置 Tab 键的顺序

在 WPF 中 Tab 键的先后顺序取决于控件在 XAML 文件中出现的先后顺序，这样的规定相当合理，因此一般不干预 WPF 中控件的 Tab 键的先后顺序。在 Win32 程序中设置 Tab 键的顺序通过 FormatTab Order 选项实现¹，也可以按快捷键 Ctrl+D 显示每个控件的 Tab 键的顺序。通过鼠标选择每一个控件来重新设置其 Tab 键的顺序，如图 19-36 所示设置 Win32 窗口中的 Tab 键顺序。

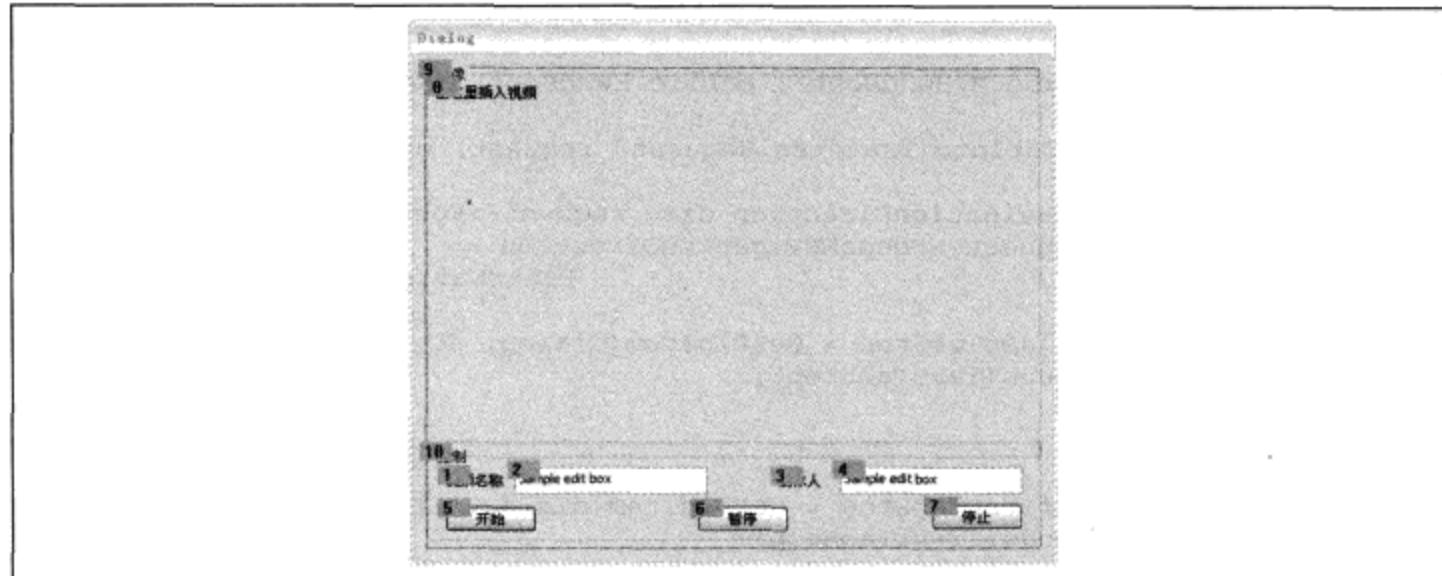


图 19-36 设置 Win32 窗口中的 Tab 键的顺序

再次运行程序，我们会发现焦点一直停留在 WPF 程序中的控件之上，无法进入 Win32 窗口中。即使使用鼠标将焦点放置在编辑框中，按下 Tab 键之后仍然是 WPF 的控件获得焦点，如图 19-37 所示。

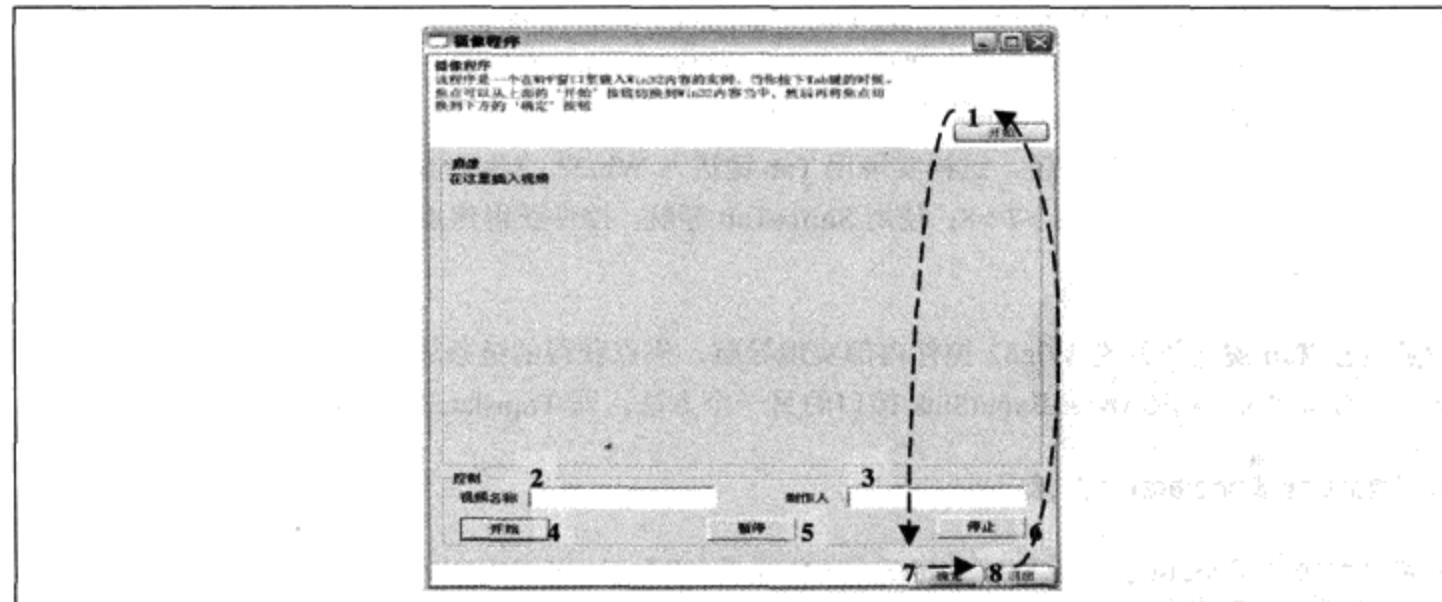


图 19-37 焦点无法进入到 Win32 窗口中

¹ View 下拉菜单中也有一个 Tab Order 选项，但是其用于设置 WinForm 窗口控件的 Tab 键顺序，与 Win32 窗口无关。

2. 用 Tab 键进入 Win32 窗口

按 Tab 键进入 Win32 窗口有如下两种情况。

- (1) 按 Tab 键从 1 号“开始”按钮进入 Win32 控件；
- (2) 按下 Shift+Tab 键从 7 号“确定”按钮返回到 Win32 控件。

这两种情况需要 HwndHost 重写 IKeyboardInputSink 接口中的 TabInfo 方法，当按下 Tab 键或者 Shift+Tab 键时，如果 HwndHost 获得焦点，这个方法就会被调用。重写的 TabInfo 方法如代码 19-30 所示。

```
public ref class MyHwndHost : public HwndHost, IKeyboardInputSink
{
    virtual bool TabInfo(TraversalRequest^ request) override
    {
        FocusNavigationDirection dir = request->FocusNavigationDirection;
        if (request->FocusNavigationDirection ==
            FocusNavigationDirection::Last)
        {
            HWND lastTabStop = GetDlgItem(dialog, ID_STOP);
            SetFocus(lastTabStop);
        }
        else
        {
            HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
            SetFocus(firstTabStop);
        }
        return true;
    }
}
```

代码 19-30 重写的 TabInfo 方法

当焦点在 1 号“开始”按钮时按下 Tab 键，HwndHost 可以获得焦点，此时 request->FocusNavigationDirection 的值为 FocusNavigationDirection::First，通过 SetFocus 方法将焦点设置到 2 号编辑框上。当焦点在 7 号“确定”按钮时按下 Shift+Tab 键，HwndHost 也可以获得焦点，request->FocusNavigationDirection 的值为 FocusNavigationDirection::Last。同样通过 SetFocus 方法将焦点设置到 6 号“停止”按钮，这样实现用 Tab 键进入 Win32 的两种情况。我们使用 Tab 键导航，控件获得焦点的顺序为 1->2->7->8；使用 Shift+Tab 导航，控件获得焦点的顺序为 8->7->6->1，如图 19-38 所示。

但是现在 Tab 键还无法在 Win32 控件内部实现导航，焦点获得的链条还缺少 2->3->4->5->6 这样一环。为此需要重写 IKeyboardInputSink 接口的另一个方法，即 TranslateAccelerator。

3. TranslateAccelerator 方法

当应用程序从 Windows 收到 WM_KEYDOWN 或者 WM_SYSKEYDOWN 等键盘消息时，该函数就会被调用。但是重写这个函数之前，注意由于标准的 Windows 头文件 winuser.h 将 TranslateAccelerator 定义为 Win32 的 TranslateAcceleratorW（Unicode）或者 TranslateAcceleratorA（非 Unicode）函数的别名。因此为了避免编译器将该函数误解为 win32 的函数。需要在该函数前加上 undefine 声明，如

代码 19-31 所示。

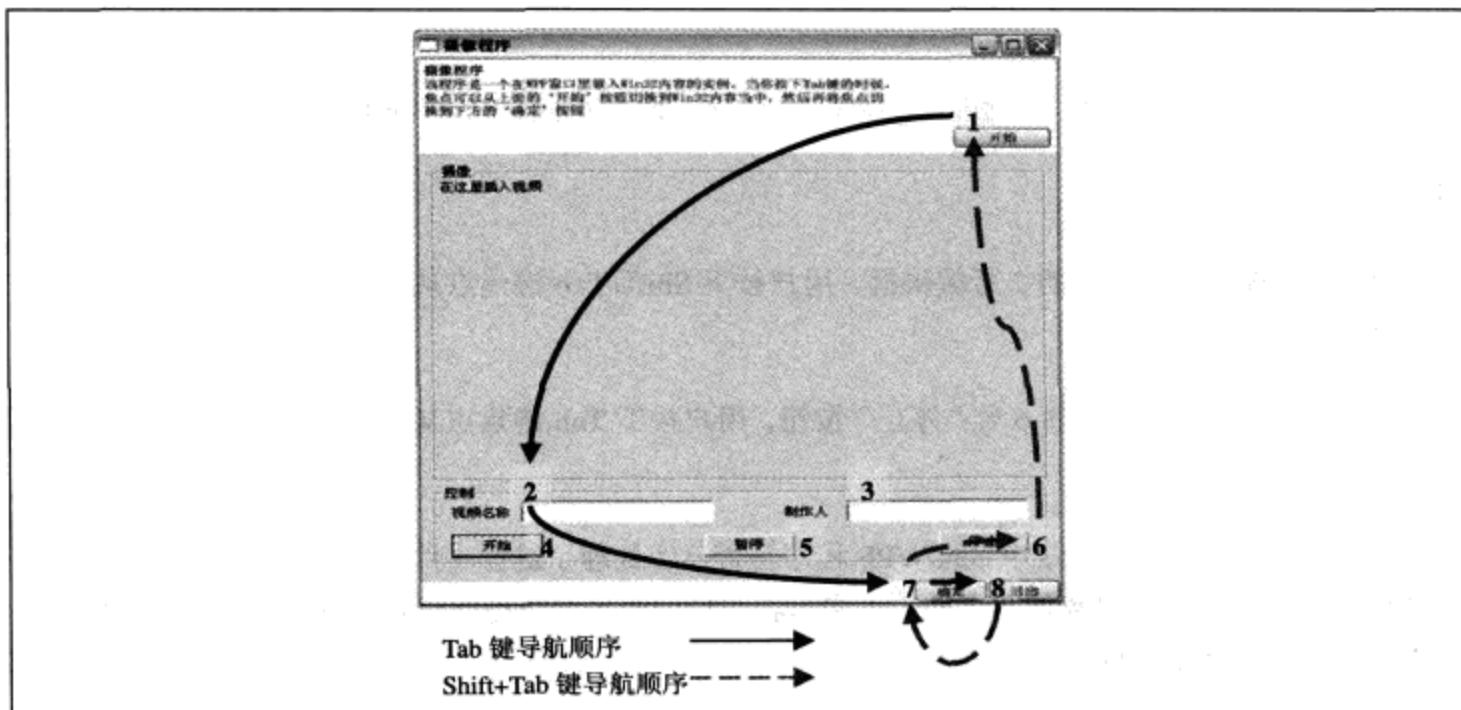


图 19-38 用 Tab 键实现进入 Win32 的两种情况

```
#undef TranslateAccelerator  
virtual bool TranslateAccelerator(System::Windows::Interop::MSG% msg,  
ModifierKeys modifiers) override  
....
```

代码 19-31 在该函数前加上 undefine 声明

TranslateAccelerator 函数中首先要处理焦点在 Win32 控件中如何离开 Win32 控件，将焦点返还给外部的 WPF 程序，如代码 19-32 所示。

```
if(msg.message == WM_KEYDOWN && msg.wParam == IntPtr(VK_TAB))  
{  
    HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);  
    HWND lastTabStop = GetDlgItem(dialog, ID_STOP);  
    TraversalRequest^ request = nullptr;  
    SHORT keystate = GetKeyState(VK_SHIFT);  
    BYTE downstate = HIBYTE(keystate);  
    BYTE togglestate = LOBYTE(keystate);  
    if(downstate)  
    {  
        if(GetFocus() == firstTabStop)  
        {  
            request = gcnew  
TraversalRequest(FocusNavigationDirection::Previous);  
            return  
((IKeyboardInputSink^)this)->KeyboardInputSite->  
OnNoMoreTabStops(request);  
        }  
        else  
        {  
            if(GetFocus() == lastTabStop)  
            {  
                request = gcnew  
TraversalRequest(FocusNavigationDirection::Next);  
                return  
            }  
        }  
    }  
}
```

```

        ((IKeyboardInputSink^)this)->KeyboardInputSite->
OnNoMoreTabStops(request);
}
}
}

```

代码 19-32 焦点离开 Win32 控件返还给外部的 WPF 程序

其中处理了如下两种情况。

- (1) 焦点在 Win32 控件中的 2 号编辑框，用户按下 Shift+Tab 键焦点从 2 号编辑框切换到 1 号“开始”按钮。
- (2) 焦点在 Win32 控件中的 6 号“停止”按钮，用户按下 Tab 键焦点从 6 号编辑框切换到 7 号“确定”按钮。

TranslateAccelerator 然后告诉外部的 WPF 程序是否已经处理了键盘事件，如果处理，则返回 true；否则返回 false。在处理键盘消息之前由于先前我们使用的消息结构类型是 Microsoft::Win32::MSG，而不是传统的::MSG 结构，所以需要编写一个函数实现两个类型的转换，如代码 19-33 所示。

```

::MSG ConvertMessage(System::Windows::Interop::MSG% msg)
{
    ::MSG m;
    m.hwnd = (HWND) msg.hwnd.ToPointer();
    m.lParam = (LPARAM) msg.lParam.ToPointer();
    m.message = msg.message;
    m.wParam = (WPARAM) msg.wParam.ToPointer();
    m.time = msg.time;
    POINT pt;
    pt.x = msg.pt_x;
    pt.y = msg.pt_y;
    m.pt = pt;
    return m;
}

```

代码 19-33 实现两个类型的转换

让对话框处理其需要处理的键盘消息函数，为此使用 Win32 的函数 IsDialogMessage。该函数会判断这个消息是否为对话框特定的消息，如果是，则处理；反之则不处理。

在这里我们使用 Win32 的 IsDialogMessage 函数尽可能地处理键盘消息，但是并不知道 IsDialogMessage 到底能够处理哪些键盘消息。从 MSDN 的相关文档来看，微软内部人员在做示例代码时也通过逆向工程来查看 IsDialogMessage 能够处理的键盘消息。我们的程序如代码 19-34 所示。

```

if (msg.message == WM_SYSKEYDOWN || msg.message == WM_KEYDOWN )
{
    ::MSG m = ConvertMessage(msg);
    switch (m.wParam)
    {
        case VK_TAB:
        case VK_LEFT:
        case VK_UP:
        case VK_RIGHT:
        case VK_DOWN:
        case VK_EXECUTE:

```

```

        case VK_RETURN:
        case VK_ESCAPE:
        case VK_CANCEL:
            IsDialogMessage(dialog, &m);
            return true;
        }
    }
    return false;
}

```

代码 19-34 用 IsDialogMessage 函数处理键盘消息

4. 支持快捷键

如果希望 Win32 控件中支持快捷键，那么还需要重写 IKeyboardInputSink 的 OnMnemonic 方法，如当按下 Alt+a 键将焦点切换到视频名称的编辑框中；按下 Alt+b 键焦点会切换到制作人编辑框中。OnMnemonic 方法同样要返回一个 bool 值以便标识是否处理相应的消息，如代码 19-35 所示。

```

virtual     bool     OnMnemonic(System::Windows::Interop::MSG%     msg,
ModifierKeys modifiers) override {
    ::MSG m = ConvertMessage(msg);
    if (msg.message == WM_SYSCHAR && GetKeyState(VK_MENU /*alt*/) ) {
        int dialogitem = 9999;
        switch (m.wParam) {
            case 'a': dialogitem = IDC_EDIT1; break;
            case 'b': dialogitem = IDC_EDIT2; break;
        }
        if (dialogitem != 9999) {
            HWND hwnd = GetDlgItem(dialog, dialogitem);
            SetFocus(hwnd);
            return true;
        }
    }
    return false; // key unhandled
};

```

代码 19-35 在 Win32 控件中支持快捷键

至此完成了一个在 WPF 中嵌入 Win32 内容的完整示例。

19.6 接下来做什么

WPF 的互操作的关键是掌握互操作的几种类型，然后按照该图研究在 WPF 中嵌入 Win32 和在 Win32 嵌入 WPF 等。

接下来，我们将进入全书的最后一个技术章节，即自定义控件。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作，《金庸全集典藏版 天龙八部》，“第三十九章解不了 名缰系嗔贪”。
- [2] MSDN Library for Visual Studio 2008 SP1Tutorial: Create a WPF Application Hosting Win32 Content。

自定义控件——出手无招，何招可破

那老者道：“唉，蠢才，蠢才！无怪你是岳不群的弟子，拘泥不化，不知变通。剑术之道，讲究如行云流水，任意所至。你使完那招‘白虹贯日’，剑尖向上，难道不会顺势拖下来吗？剑招中虽没这等姿式，难道你不会别出心裁，随手配合么？”这一言登时将令狐冲提醒，他长剑一勒，自然而然地便使出“有凤来仪”，不等剑招变老，已转“金雁横空”。长剑在头顶划过，一勾一挑，轻轻巧巧地变为“截手式”。转折之际，天衣无缝，心下甚是舒畅。

——《笑傲江湖：第十章 传剑》^[1]

这是风清扬老前辈传剑给令狐冲的一个片断，武学的一招一式即使精妙，也是固定的，拘泥于招式的人终究会被破招。倘若出手无招，又何招可破呢？

本章开篇会让风清扬老前辈出场点化木木，能否顿悟就看木木的造化了。本章的内容如下。

- (1) 风老前辈登场。
- (2) 用 RadioButton 实现红绿灯。
- (3) 何时自定义控件。
- (4) 自定义控件。
- (5) 无外观控件。
- (6) 接下来做什么。

20.1 风老前辈登场

木木下班离开桃花岛软件公司，阴沉沉的天下着小雨。木木的心情也和这天气一般，有些小小的郁闷。原来公司需要用 WPF 扩展和重新构建一部分控件。木木原以为自己已经算是精通 WPF 了，于是信心满满地参与了这项工作。但是工作了一段时间之后，发现困难重重。自己学习《葵花宝典》这么长时间，花费了不知多少精力。到头来，虽然依赖属性、路由事件、资源、样式和模板等，每一个 WPF 的特性都知道，但是就不知道该如何下手。

郁闷中，不知不觉居然又逛到了过去那个旧书市场¹。木木蹲下来，正在随意翻翻。突然听到耳边一个苍老的声音：“年轻人，我看你相貌不俗，天庭饱满，正是百年不世出的程序界奇才。这里有一本《降龙十八掌秘笈——Silverlight 自学手册》，原价 50 元。我和你有缘，仅仅 5 元就卖给你吧。”木木一听这话，觉得颇为熟悉，并不声张。悄然把书翻到了最后一页，扫了一眼。果不其然和《葵花宝典》一样，定价只是 3.9 元，知道不过是骗人买书的伎俩。本想起身就走，抬头一看，一个白须青袍的老者坐在一个破桌旁。桌上放了一台笔记本电脑，正笑吟吟地看着他。

不知有何种魔力，木木竟不由自主地和老者攀谈起来，这位老者正是风清扬老前辈。

“老师傅，读书没用啊。”

“喔，怎么没用？”

“我过去买过一本《葵花宝典——WPF 自学手册》。”不等木木把话说完。老者就接过话茬：“加上这一本，两书搭配，效果更佳啊。”

“唉……”木木长叹一声：“书倒是看了很久，WPF 的每个特性我也都清楚。但是现在公司要重新扩展和定义控件，我还是无从下手。”

老者笑笑，一手搭在木木的肩上，一手指了指路旁的红绿灯，说道：“小伙子，你看这红绿灯像不像 WPF 中的 RadioButton 啊？”

木木一听，本想脱口而出：“一个红绿灯，一个 RadioButton，这哪儿跟哪儿？怎么可能像？”但是转念一想这老头肯定不会无缘无故地这般说，还是要仔细琢磨琢磨，于是木木脑海里出现了，如图 20-1 所示的画面。

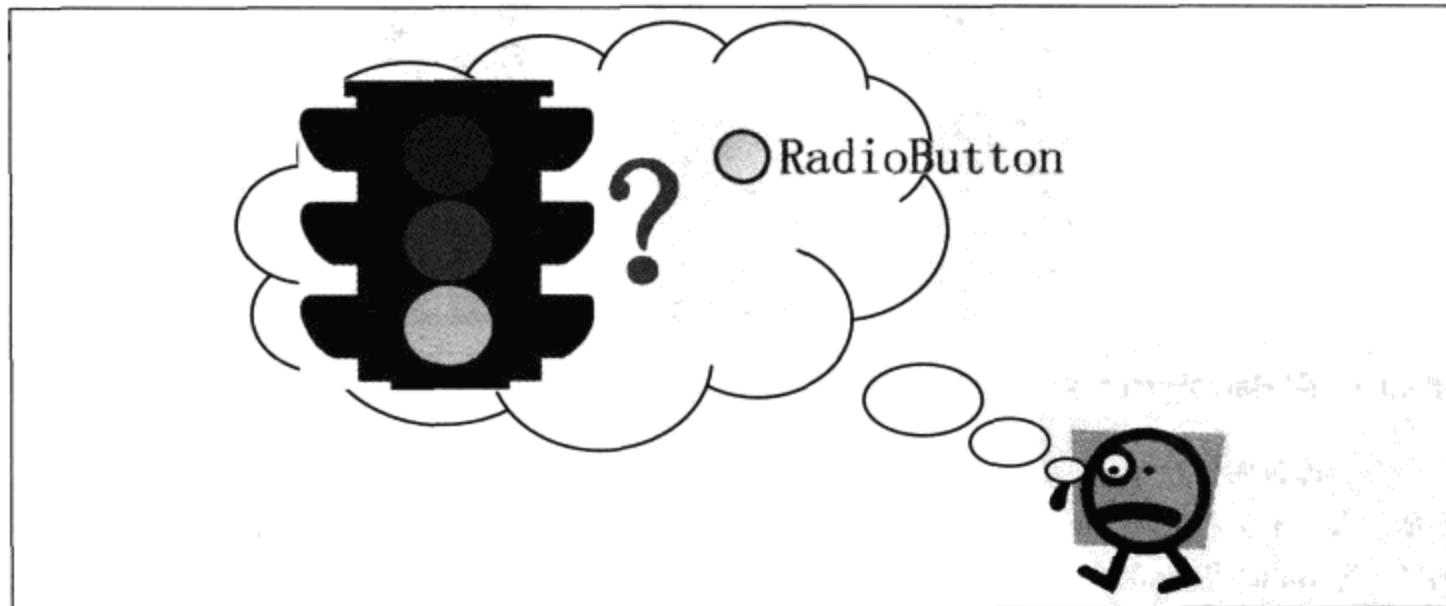


图 20-1 木木脑海里的画面

¹ 参见楔子，木木是在旧书市场发现的《葵花宝典》。

木木想了一会儿，吞吞吐吐地说：“有点像。”

老者的神情显得有些兴奋，心想这小伙子莫非还真悟出了一点道理。于是问道：“它们哪一点像？”

木木答：“它们都有一个圆圈。”

这个答案是他老人家始料未及的，风老前辈来回踱了几步才说：“年轻人，你太过拘泥于控件的外表了。我说它们两者像，并不是指形似，而是神似。”

“神似？”木木很是不解。

“一组 RadioButton 一次只能选中一个，而红绿灯一次也只能亮一盏，这就是它们的神似。”

木木知道遇上高人了，连忙作揖道：“老师傅，还请您多多指教。”

20.2 用 RadioButton 实现红绿灯

现在我们就来看一下风老前辈用 RadioButton 实现的红绿灯吧，如图 20-2 所示。当鼠标选中某一盏灯时该灯会自动变亮；选中另外一盏灯时，则会自动变暗，另外一个选中的灯又会自动变亮。

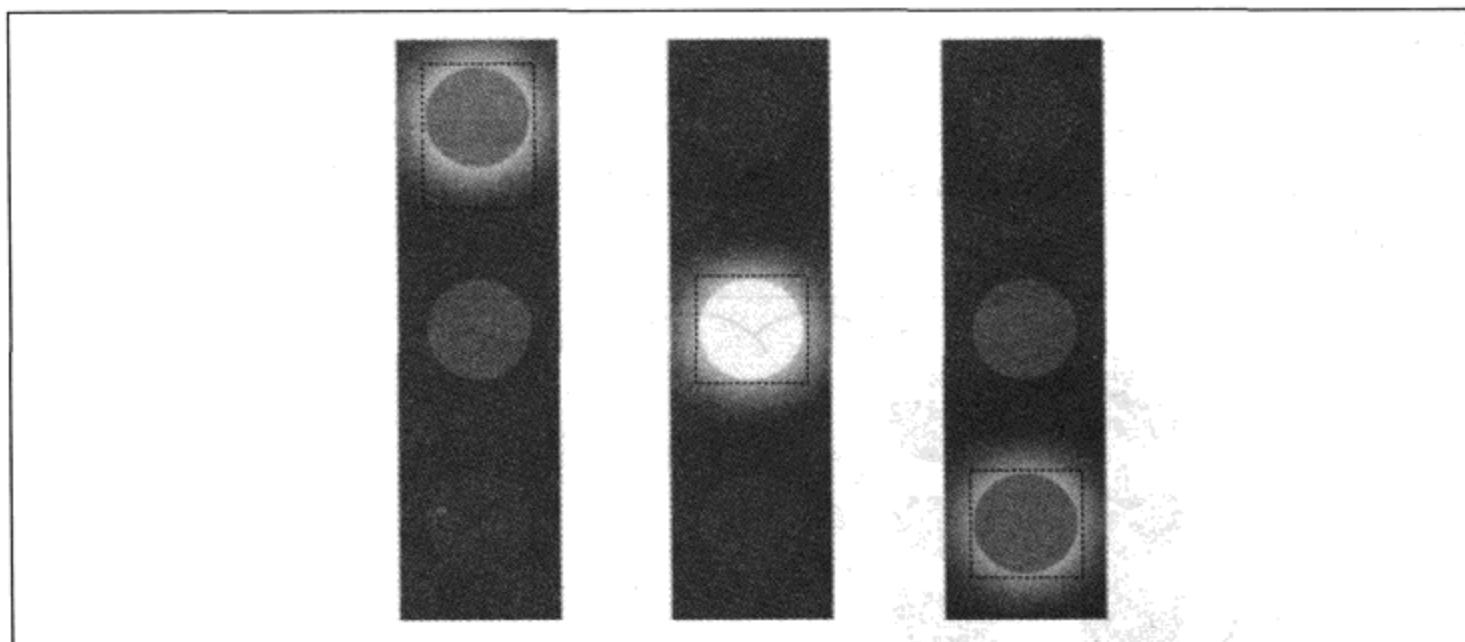


图 20-2 用 RadioButton 实现的红绿灯

“木木，你首先看看 RadioButton 传统的构成。”经过攀谈，风老前辈已经得知眼前的这个年轻人名如其人，木头木脑。“传统的 RadioButton 是通过 BulletDecorator 构成的。BulletDecorator 有两个属性，即 Bullet 和 Child。”如代码 20-1 所示。

通过《葵花宝典》中提供的控件模板浏览器查看到的部分 RadioButton 模板代码

```
<BulletDecorator  
    Background="#00FFFFFF">  
    <BulletDecorator.Bullet>  
        <mwt:BulletChrome  
            ...../>
```

```
</BulletDecorator.Bullet>
<ContentPresenter
    ....>
</BulletDecorator>
```

代码 20-1 部分 RadioButton 模板代码

从图 20-3 所示可以看出传统的 RadioButton 中 Bullet 和 Child 所处的位置。

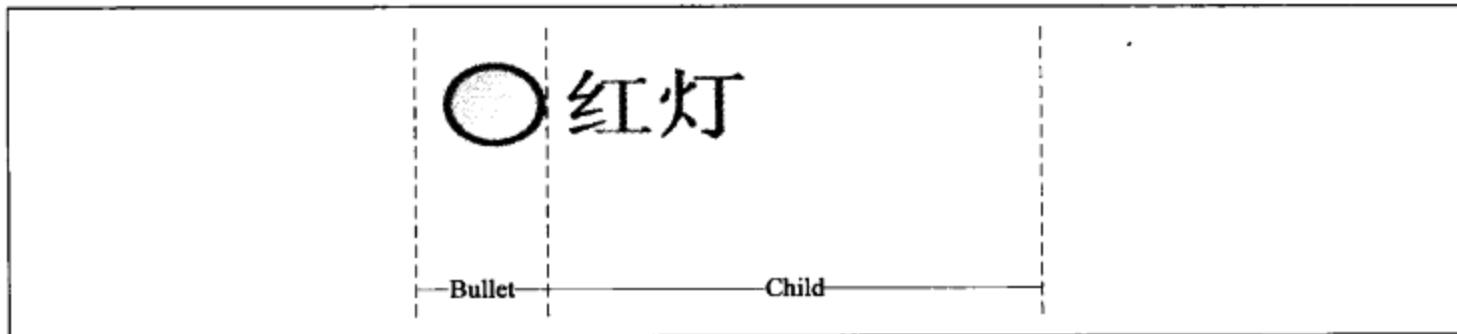


图 20-3 Bullet 和 Child 所处的位置

“我们把传统的 RadioButton 的模板略做调整，设置 Bullet 的地方用一个椭圆，而 Child 根本不用考虑。”说完风老前辈写下如代码 20-2 所示的代码。

```
<Style TargetType="{x:Type RadioButton}" >
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type RadioButton}">
                <BulletDecorator >
                    <BulletDecorator.Bullet>
                        <Grid Width="48" Height="48">
                            <Ellipse x:Name="background" Margin="2"
Fill="{TemplateBinding Background}" Width="Auto" Height="Auto" Opacity="0.2">
                                <Ellipse.BitmapEffect>
                                    <OuterGlowBitmapEffect
x:Name="myOuterGlowBitMapEffect" GlowColor="Gold" GlowSize="0" />
                                </Ellipse.BitmapEffect>
                            </Ellipse>
                        </Grid>
                    </BulletDecorator.Bullet>
                </BulletDecorator>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

代码 20-2 Bullet 部分用椭圆代替

风老前辈说：“让某个灯由暗变亮或者由亮变暗的过程需要使用动画，实际上这是由两个动画共同完成的。当灯由暗变亮时，是 OuterGlowBitmapEffect 的 Size 属性从 0 到 20，而另一个则是椭圆透明度属性从半透明（0.2）到完全不透明（1）；反之亦然。”如代码 20-3 所示。

```
<ControlTemplate.Triggers>
    <EventTrigger RoutedEvent="RadioButton.Checked">
        <BeginStoryboard>
            <Storyboard>
```

```
<DoubleAnimation  
Storyboard.TargetName="myOuterGlowBitmapEffect"  
Storyboard.TargetProperty="GlowSize" From="0" To="20" Duration="0:0:0.1" />  
    <DoubleAnimation Storyboard.TargetName="background"  
        Storyboard.TargetProperty="Opacity"  
        From="0.2" To="1" Duration="0:0:0.1" />  
    </Storyboard>  
</BeginStoryboard>  
</EventTrigger>  
<EventTrigger RoutedEvent="RadioButton.Unchecked">  
    <BeginStoryboard>  
        <Storyboard>  
            <DoubleAnimation  
                Storyboard.TargetName="myOuterGlowBitmapEffect"  
                Storyboard.TargetProperty="GlowSize"  
                From="20" To="0" Duration="0:0:0.1" />  
            <DoubleAnimation Storyboard.TargetName="background"  
                Storyboard.TargetProperty="Opacity"  
                From="1" To="0.2" Duration="0:0:0.1" />  
        </Storyboard>  
    </BeginStoryboard>  
</EventTrigger>  
</ControlTemplate.Triggers>
```

代码 20-3 控制某盏灯的明暗

木木越看越觉得佩服，自己也知道动画，也知道模板，更知道 RadioButton。但从来没有想过像风老前辈这样将它们随意组合，然后构建一个全新的控件。

20.3 何时自定义控件？

木木说：“老前辈，现在我们公司需要自定义一套控件。我不知道该如何下手，还请您赐教。”

风清扬摆了摆手，说：“你们公司的想法有点问题。”

木木大惊，这个自定义控件的项目可是黄岛主亲自过问的。凭黄药师的聪明才智，怎么可能会有问题呢？

“很多时候都不需要自定义控件，木木。”

“为何？老前辈。”

“原因有两点。”

20.3.1 不要被控件的外观所欺骗，要考虑其内在本质

很多时候首先要考虑的是控件的内在本质，好比这个红绿灯和 RadioButton。尽管它们在外观上差异很大，但是本质上是相似的。在要自定义一个控件之前，首先要想的是这个控件的行为是否和原有控件的行为类似。如果能找到，应该是修改原有的控件，而不是重新定义一个控件。

风老前辈又指了指墙上的一个闸刀，问到：“如果你们公司让你做这样一个闸刀控件，你能在

WPF 中找到类似的控件么？”

木木想了很久没有想出来。

木木说：“老前辈，WPF 的控件有那么多。我一个一个将它们和闸刀比照，真的是要到猴年马月了。”

“WPF 的控件虽然多，但是在我眼里，它们只有 5 种。”说完，风老前辈列出了表 20-1。

表 20-1 WPF 的控件

单一控件	Button、RadioButton、CheckBox……
一定范围的控件	Slider、ProgressBar……
列表控件	ItemsControl、ListBox、ListView、ComboBox……
层次结构的控件	TreeView……
文本控件	TextBox、TextBlock……

风老前辈说：“再不行，就要把这些控件进行组合，如 UserControl 也可以称为‘第 6 类组合控件’。那么多控件一个一个比照肯定不行，但是你首先判断闸刀和哪一类控件行为相似，缩小了范围再比对，就好找多了。”

一番话，木木豁然开朗。寻着风老前辈的思路想下去，居然没有多久心中有了答案。于是恭恭敬敬地说：“老前辈，这闸刀应该属于单一的控件。闸刀有打开和关闭两种状态，它只能处于其中一种状态。CheckBox 和 RadioButton 都有这样的特性，但是 RadioButton 的状态变换是通过将焦点切换到其他 RadioButton 上实现的。这一点和闸刀并不相像，我想应该是 CheckBox 吧。”

如图 20-4 所示。

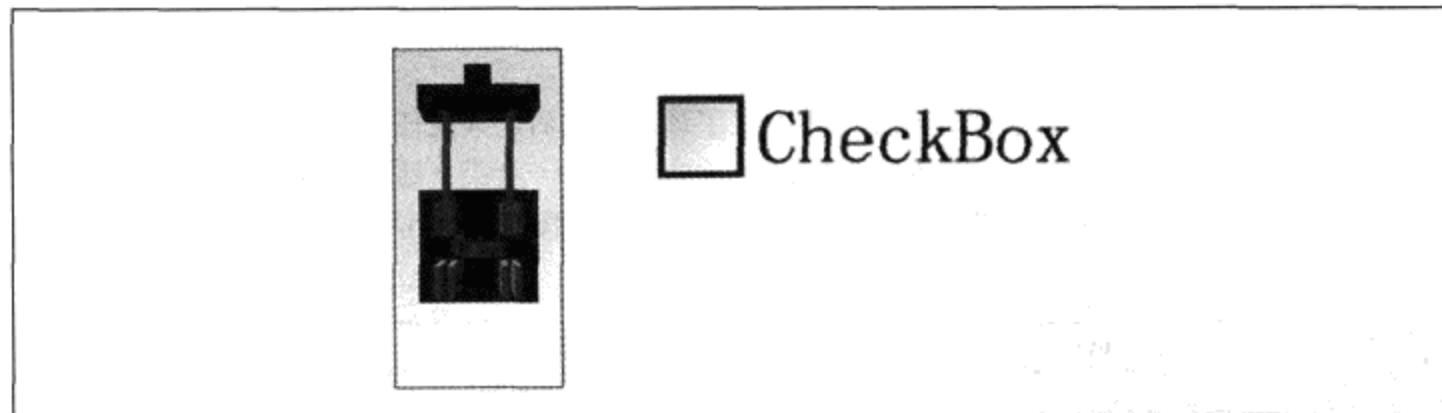


图 20-4 使用 CheckBox 实现的类似闸刀的控件

风老前辈面露喜色，心中暗暗赞道：“小伙子，上道了”。

20.3.2 Content 模型、模板和附加属性

在 WPF 中自定义控件所处的地位其实已经被大大弱化，这得益于 WPF 的如下特性。

(1) Content 模型

将 Content 比做北冥神功是因为无论什么控件，它总能将其包容进来，这样一个有 Content 属性的控件非常容易扩展。如将一个按钮中放置一个图片或者在按钮中绘制一些图形，这在 MFC 和 WinForm 时代必须考虑派生自按钮的一个自定义按钮，但是在 WPF 中则完全不用。

(2) 模板

模板更为强大，控件的外观已经可以像变形金刚一样任意变换，不变的只是控件的“神”。前面的红绿灯和闸刀都已经和传统的控件外观相去甚远，这就是模板的力量。这里的模板指控件和数据模板，前者定义控件的外观；后者定义数据的外观。

(3) 附加属性

附加属性有助于我们复用原来的控件，而不需要重新定义一个控件。如希望在原有的一个控件 oldctrl 基础上添加一个新的属性 newprop，原来的思路势必是从 oldctrl 派生一个新的控件 newctrl。然后为其添加属性 newprop，如图 20-5 所示。

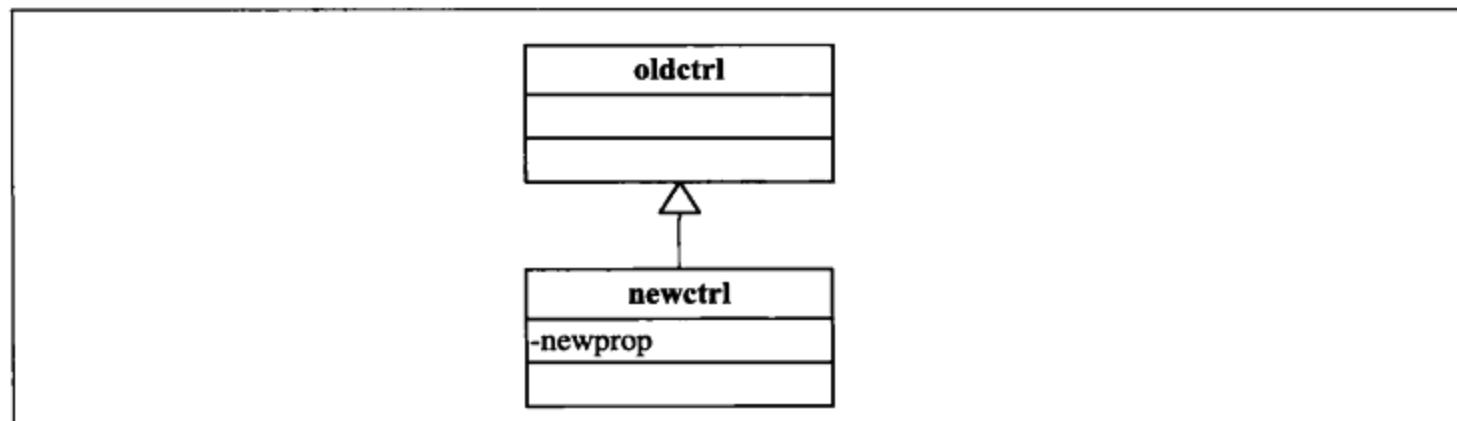


图 20-5 原来的思路

附加属性最为明显的一个特点就是在同一个类中定义了该属性之后，其他任何类都可以使用。这样在 oldctrl 上扩展一个属性不必派生一个新类 newctrl，而是在另外一个类（这个类以 Helper 作为后缀）上定义一个附加属性 newprop 供其他需要该属性的类使用，如图 20-6 所示。

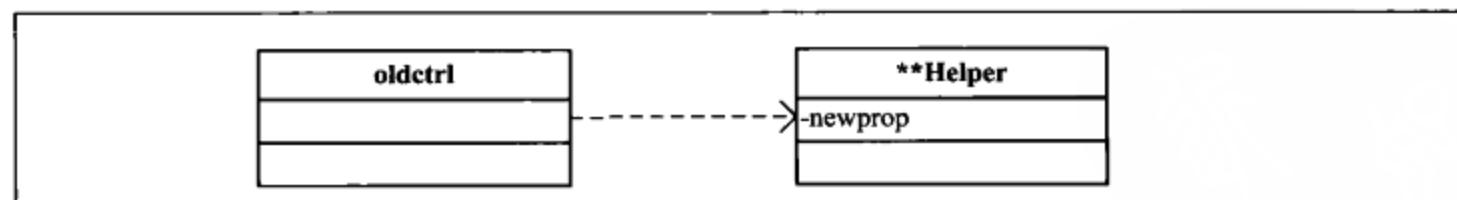


图 20-6 有了附加属性后的思路

20.3.3 使用附加属性扩展现有控件

我们还是以一个例子来说明附加属性如何扩展现有控件的功能，传统的进度条如图 20-7 所示，它显示一个连续的进度过程。

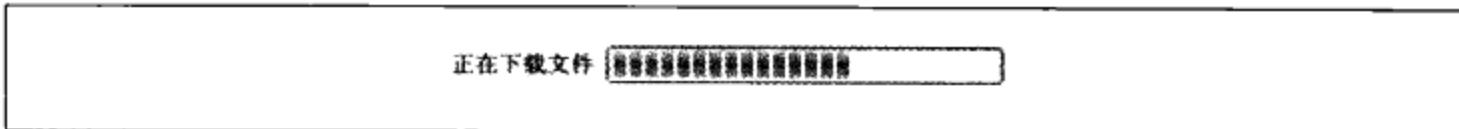


图 20-7 传统的进度条

如果希望显示每一个阶段的进度，如工作完成了 20% 亮起一盏红灯；完成了 40%，亮起两盏红灯^[3]，如图 20-8 所示。

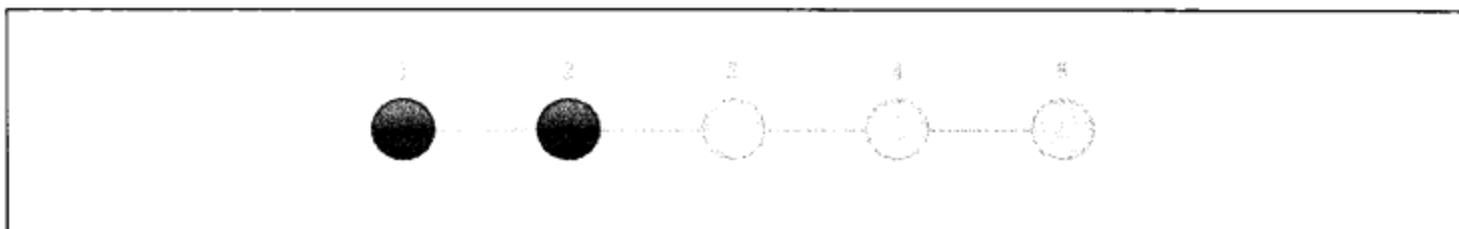


图 20-8 使用附加属性扩展后的进度条

首先定义一个进度条的帮助类，并命名为“ProcessStageHelper”。该类只是定义两个附加属性，因此只需要派生自 DependencyObject。两个附加属性，一个是 ProcessCompletion 属性，这个值和进度条的 Value 值均表示进度条的当前进度值；是一个 Double 数值型，另一个是 ProcessStage 属性，它表示不同阶段值。根据 ProcessCompletion 值分为 6 个阶段，如表 20-2 所示。

表 20-2 根据 ProcessCompletion 值分为 6 个阶段

20% > ProcessCompletion >= 0%	Stage0
40% > ProcessCompletion >= 20%	Stage1
60% > ProcessCompletion >= 40%	Stage2
80% > ProcessCompletion >= 60%	Stage3
100% > ProcessCompletion >= 80%	Stage4
ProcessCompletion = 100%	Stage5

在代码 20-4 中，当 ProcessCompletion 改变时 OnProcessCompletionChanged 函数中会根据 ProcessCompletion 当前的值来设置 ProcessStage 为哪个阶段。

```
public enum ProcessStage
{
    Stage0,
    Stage1,
    Stage2,
    Stage3,
    Stage4,
    Stage5
}
public class ProcessStageHelper : DependencyObject
{
    public static readonly DependencyProperty ProcessCompletionProperty =
        DependencyProperty.RegisterAttached(
            "ProcessCompletion", typeof(double), typeof(ProcessStageHelper), new
            PropertyMetadata(0.0, OnProcessCompletionChanged));
}
```

```

        private static readonly DependencyPropertyKey ProcessStagePropertyKey
= DependencyProperty.RegisterAttachedReadOnly(
    "ProcessStage", typeof(ProcessStage), typeof(ProcessStageHelper), new
PropertyMetadata(ProcessStage.Stage0));

        public static readonly DependencyProperty ProcessStageProperty =
ProcessStagePropertyKey.DependencyProperty;

        private static void OnProcessCompletionChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
{
    double progress = (double)e.NewValue;
    ProgressBar bar = d as ProgressBar;
    if (progress >= 0 && progress < 20)
bar.SetValue(ProcessStagePropertyKey, ProcessStage.Stage0);
    if (progress >= 20 && progress < 40)
bar.SetValue(ProcessStagePropertyKey, ProcessStage.Stage1);
    if (progress >= 40 && progress < 60)
bar.SetValue(ProcessStagePropertyKey, ProcessStage.Stage2);
    if (progress >= 60 && progress < 80)
bar.SetValue(ProcessStagePropertyKey, ProcessStage.Stage3);
    if (progress >= 80 && progress < 100)
bar.SetValue(ProcessStagePropertyKey, ProcessStage.Stage4);
    if (progress == 100) bar.SetValue(ProcessStagePropertyKey,
ProcessStage.Stage5);
}

        public static void SetProcessCompletion(ProgressBar bar, double
progress)
{
    bar.SetValue(ProcessCompletionProperty, progress);
}

        public static void SetProcessStage(ProgressBar bar, ProcessStage stage)
{
    bar.SetValue(ProcessStagePropertyKey, stage);
}
}

```

代码 20-4 根据 ProcessCompletion 当前的值设置 ProcessStage 的阶段

接下来需要重新定义控件的模板来改变进度条的外观，进度条由 5 个椭圆，4 个矩形和 5 个文本框共同构成。它们在一个 2 行 9 列的 Grid 面板中，如图 20-9 所示。

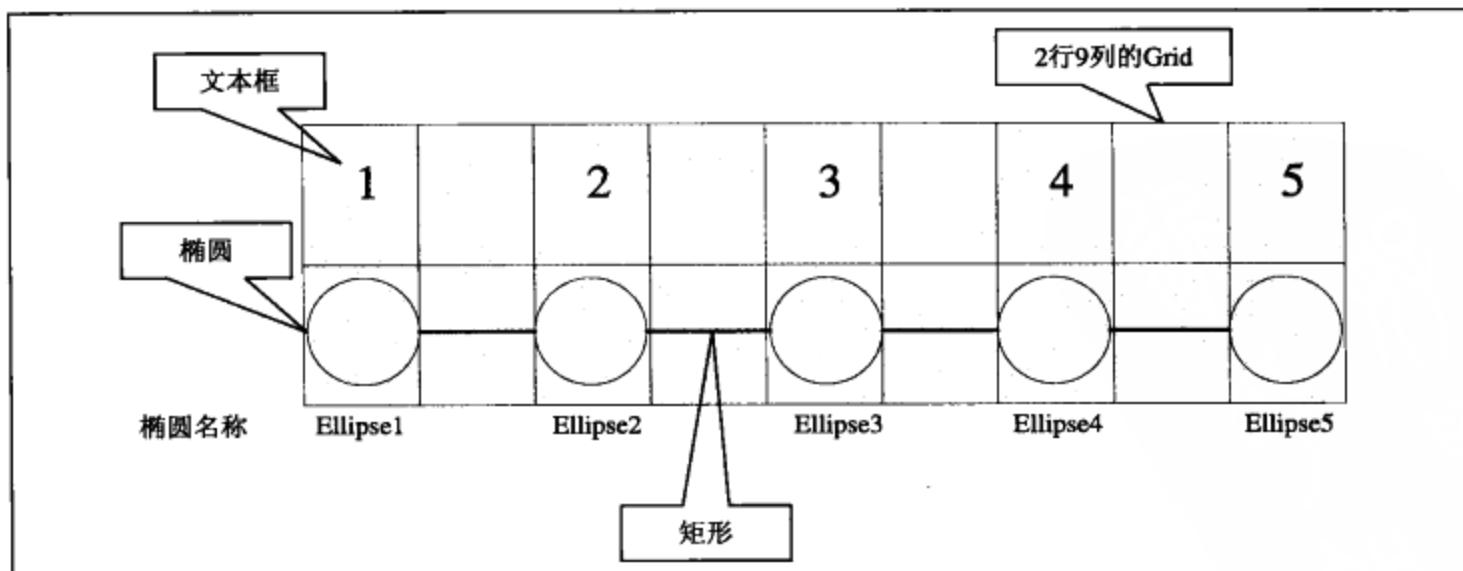


图 20-9 重新定义控件的模板

在该模板的触发器集合中必需定义以下 5 个触发器。

- (1) 当 ProcessStageHelper.ProcessStage 值为 Stage1, Ellipse1 的填充色发生改变。
- (2) 当 ProcessStageHelper.ProcessStage 值为 Stage2, Ellipse1 和 Ellipse2 的填充色发生改变。
- (3) 和 (4) 以此类推。
- (5) 当 ProcessStageHelper.ProcessStage 值为 Stage5、Ellipse1、Ellipse2、Ellipse3、Ellipse4 和 Ellipse5 的填充色都要发生改变。

代码 20-5 所示为 ProgressBar 控件模板的代码。

```
<UserControl.Resources>

    <SolidColorBrush x:Key="NormalBrush"
        Color="#eeeeee" />
    <LinearGradientBrush x:Key="SelectedStageBrush"
        StartPoint="0,1"
        EndPoint="0,0">
        <!--5个椭圆-->
        <Ellipse x:Name="Ellipse1"
            Fill="#eeeeee"
            Stroke="#c7c7c7"
            StrokeThickness="1"
            Grid.Column="0"
            Grid.Row="1" />
        <Ellipse x:Name="Ellipse2"
            Fill="#eeeeee"
```

```
        Stroke="#c7c7c7"
        StrokeThickness="1"
        Grid.Column="2"
        Grid.Row="1" />
<Ellipse x:Name="Ellipse3"
        Fill="#eeeeee"
        Stroke="#c7c7c7"
        StrokeThickness="1"
        Grid.Column="4"
        Grid.Row="1" />
<Ellipse x:Name="Ellipse4"
        Fill="#eeeeee"
        Stroke="#c7c7c7"
        StrokeThickness="1"
        Grid.Column="6"
        Grid.Row="1" />
<Ellipse x:Name="Ellipse5"
        Fill="#eeeeee"
        Stroke="#c7c7c7"
        StrokeThickness="1"
        Grid.Column="8"
        Grid.Row="1" />
<!--4个矩形-->
<Rectangle Fill="#c7c7c7"
        Grid.Column="1"
        Grid.Row="1"
        Height="1" />
<Rectangle Fill="#c7c7c7"
        Grid.Column="3"
        Grid.Row="1"
        Height="1" />
<Rectangle Fill="#c7c7c7"
        Grid.Column="5"
        Grid.Row="1"
        Height="1" />
<Rectangle Fill="#c7c7c7"
        Grid.Column="7"
        Grid.Row="1"
        Height="1" />
<!--5个文本框-->
<TextBlock Text="1"
        Grid.Column="0"
        Grid.Row="0"
        TextAlignment="Center" />
<TextBlock Text="2"
        Grid.Column="2"
        Grid.Row="0"
        TextAlignment="Center" />
<TextBlock Text="3"
        Grid.Column="4"
        Grid.Row="0"
        TextAlignment="Center" />
<TextBlock Text="4"
        Grid.Column="6"
        Grid.Row="0"
        TextAlignment="Center" />
<TextBlock Text="5"
        Grid.Column="8"
        Grid.Row="0"
        TextAlignment="Center" />
</Grid>
```

```

<ControlTemplate.Triggers>
    <!--ProcessStage 值为 Stage1 时，第 1 个椭圆改变填充色 -->
    <Trigger Property="local:ProcessStageHelper.ProcessStage"
        Value="Stage1">
        <Setter Property="Fill"
            Value="{StaticResource SelectedStageBrush}"
            TargetName="Ellipse1" />
        <Setter Property="Stroke"
            Value="#bb2d00"
            TargetName="Ellipse1" />
    </Trigger>
    <!--下面以此类推-->
    <Trigger Property="local:ProcessStageHelper.ProcessStage"
        Value="Stage2">
        <Setter Property="Fill"
            Value="{StaticResource SelectedStageBrush}"
            TargetName="Ellipse2" />
        <Setter Property="Stroke"
            Value="#bb2d00"
            TargetName="Ellipse2" />
        <Setter Property="Fill"
            Value="{StaticResource SelectedStageBrush}"
            TargetName="Ellipse1" />
        <Setter Property="Stroke"
            Value="#bb2d00"
            TargetName="Ellipse1" />
    </Trigger>
    <Trigger Property="local:ProcessStageHelper.ProcessStage"
        Value="Stage3">
        .....
    </Trigger>
    <Trigger Property="local:ProcessStageHelper.ProcessStage"
        Value="Stage4">
        .....
    </Trigger>
    <Trigger Property="local:ProcessStageHelper.ProcessStage"
        Value="Stage5">
        .....
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</UserControl.Resources>

```

代码 20-5 ProgressBar 控件模板的代码

最后一步是为 ProgressBar 设置模板，同时要将其 Value 值绑定到 ProcessCompletion 附加属性。这里数据绑定使用 RelativeSource Self 表示绑定自身，即 ProgressBar。同时 ProgressBar 通过一个 Slider 来控制进度，因此其 Value 值又和 Slider 的 Value 属性绑定，如代码 20-6 所示。

```

<Slider x:Name="_slider"
    Minimum="0"
    Maximum="100"
    Value="0"
    Orientation="Horizontal"
    DockPanel.Dock="Bottom" />
<ProgressBar Template="{StaticResource ProcessStageTemplate}"
    Margin="20"
    Height="50"
    Value="{Binding Value, ElementName=_slider}"
```

```
local:ProcessStageHelper.ProcessCompletion="{Binding Value,  
RelativeSource={RelativeSource Self}}" />
```

代码 20-6 为 ProgressBar 设置模板

木木开始有点糊涂了，我们不妨看看图 20-10 了解一下数据是如何一步一步推动的。

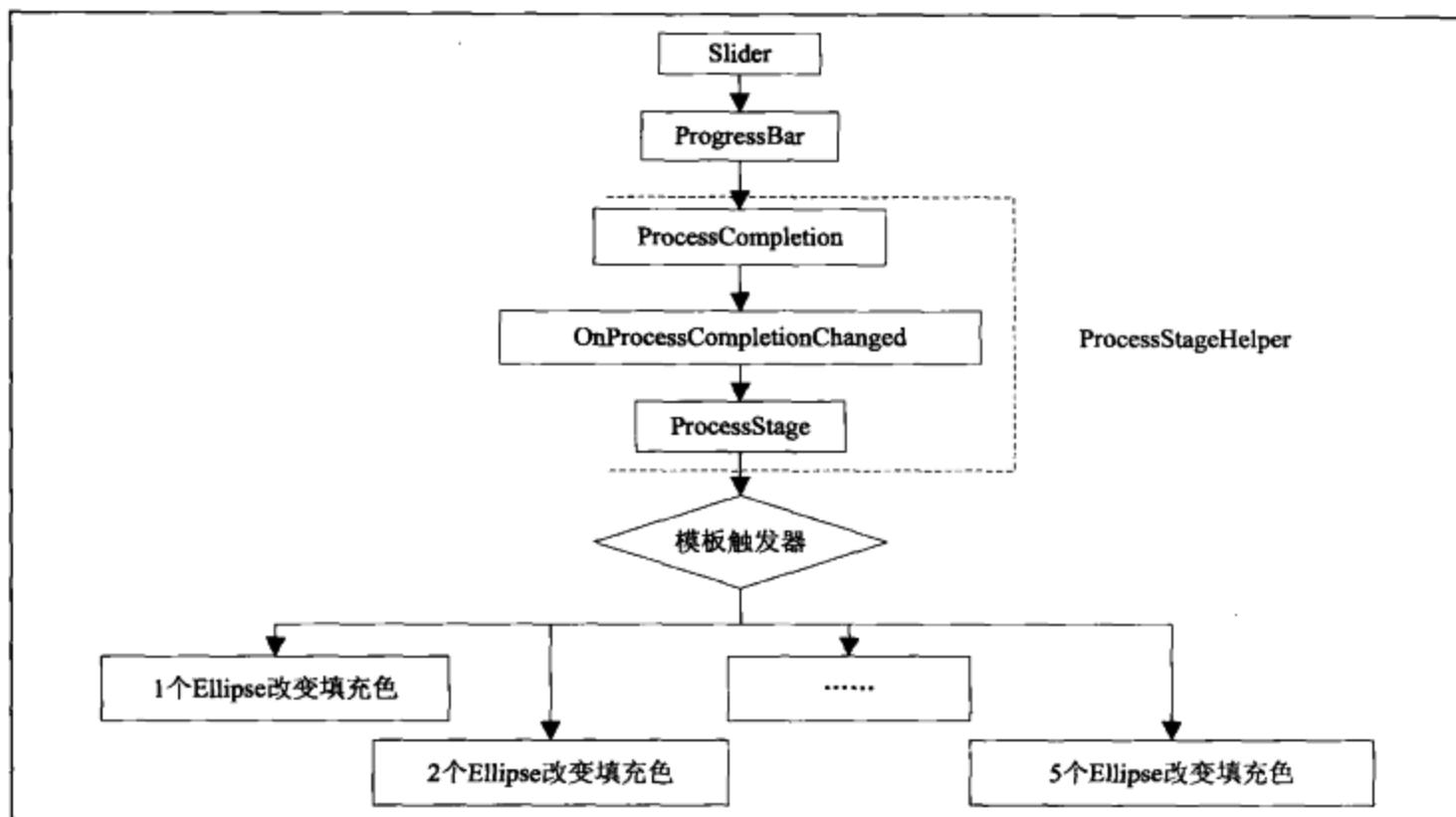


图 20-10 数据的推动过程

- (1) 首先用户通过操作改变 Slider 的 Value 值。
- (2) 由于 ProgressBar 的 Value 属性和 Slider 的 Value 属性绑定在一起，因此 ProgressBar 的 Value 值也发生了改变。
- (3) 同样 ProcessStageHelper 的附加属性 ProcessCompletion 也发生了改变。
- (4) OnProcessCompletionChanged 函数会被调用，在其中根据 ProcessCompletion 的值设置了 ProcessStage 的值。
- (5) 模板触发器根据不同的 ProcessStage 值来改变椭圆的填充色。

20.4 自定义控件

我们可以得出一个结论，即不到万不得已时才自定义一个控件。当需要改变控件外观或者扩展控件功能时，通常遵循如图 20-11 所示的思路。

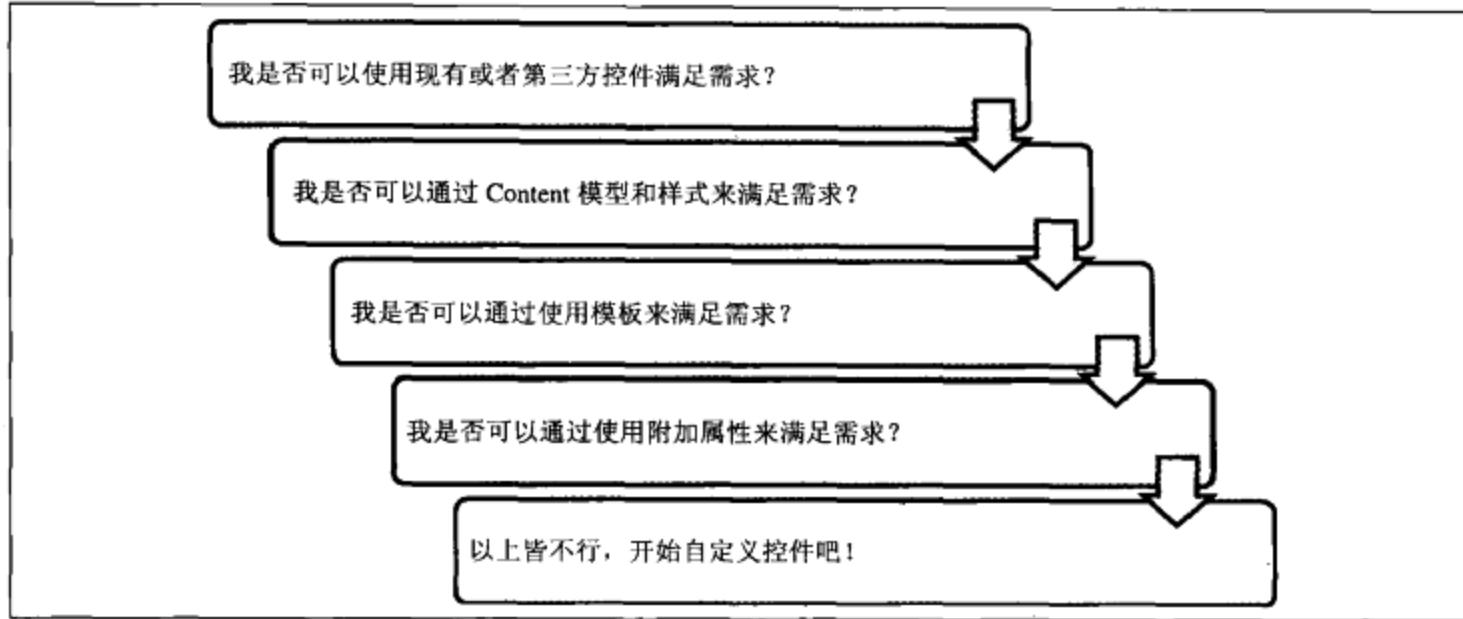


图 20-11 需改变控件外观或者扩展控件功能时通常遵循的思路

20.4.1 自定义控件的 3 个层次

自定义控件按照从不同的基类派生分为 3 个层次，如图 20-12 所示。

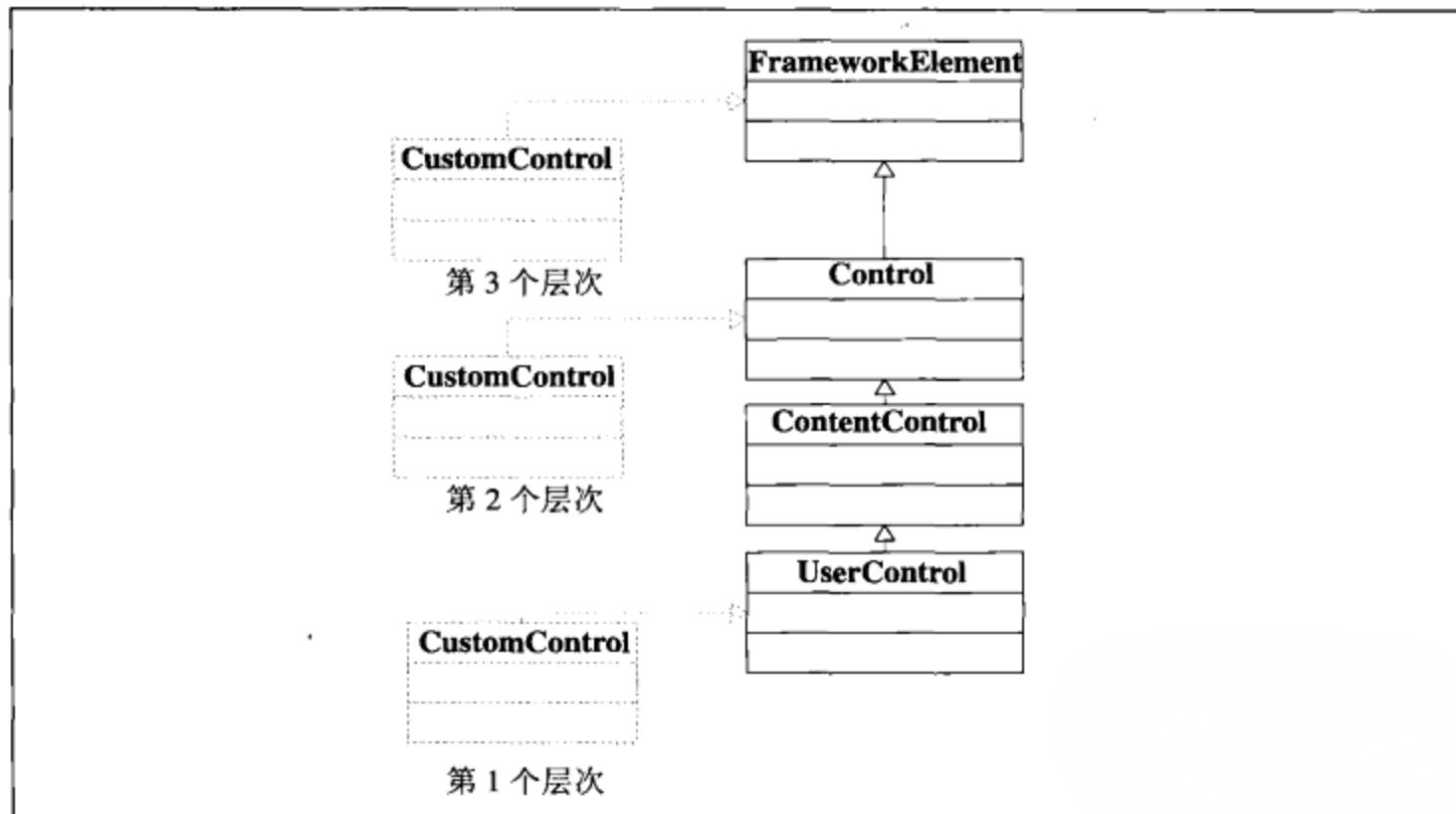


图 20-12 自定义控件的 3 个层次

第 1 个层次从 **UserControl** 派生而来，这是自定义控件最为简单的方法。优点是简单；问题是控件一旦设置好，外观就是固定的，不允许使用者改变。

第 2 个层次从 **Control** 派生而来，相对复杂一些。但是允许使用者通过模板自定义控件的外观，也可

以支持不同的主题。

第3个层次则从 FrameworkElement 派生而来，非常类似过去在 WinForm 和 MFC 下的自定义控件方法，如通过重载 OnRender 函数绘制控件。这种方法比较适用于更精准地控制控件外观，如重新定义一个不同的形状。

20.4.2 从 UserControl 开始

我们先从 UserControl 开始定义一个能够通过上下按钮调整数值的控件^[4]，该控件实际上是两个按钮和一个文本框的组合。每单击“增加”按钮一次，文本框的数值加 1；每单击“减少”按钮一次，文本框的数值减 1，如图 20-13 所示。

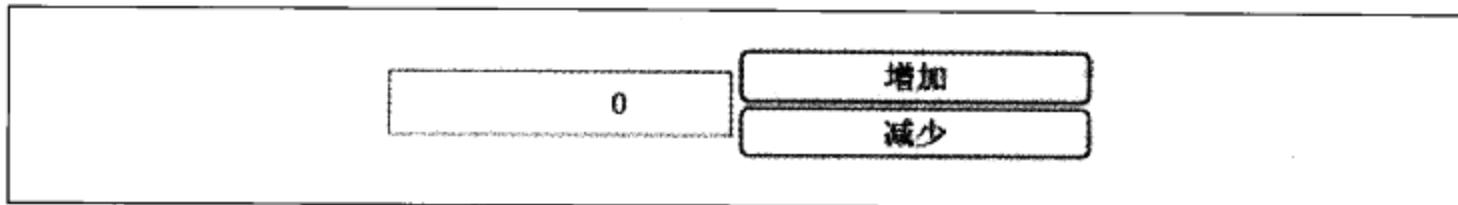


图 20-13 一个能够通过上下按钮调整数值的控件

这个控件的外观并不复杂，在一个工程中新建一个 UserControl，将其命名为“NumericUpDown”，其 XAML 文件的内容如代码 20-7 所示。

```
<UserControl x:Class="mumu_customcontrol.NumericUpDown"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:mumu_customcontrol">
    <Grid Margin="3">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Border BorderThickness="1" BorderBrush="Gray" Margin="2"
            Grid.RowSpan="2" VerticalAlignment="Center" HorizontalAlignment=
"Stretch">
            <TextBlock
                Width="60" TextAlignment="Right" Padding="5"
                Text="{Binding RelativeSource={RelativeSource FindAncestor,
                    AncestorType={x:Type local:NumericUpDown}},
                    Path=Value}"/>
            </Border>
            <RepeatButton Name="upButton" Click="upButton_Click"
                Grid.Column="1" Grid.Row="0">增加</RepeatButton>
            <RepeatButton Name="downButton" Click="downButton_Click"
                Grid.Column="1" Grid.Row="1">减少</RepeatButton>
        </Grid>
    </UserControl>
```

代码 20-7 NumericUpDown 的 XAML 文件

这个派生自 `UserControl` 的新的控件至少需要为用户提供两个基本的功能。

(1) 当前数值多少。

(2) 数值何时改变?

为此, 需要定义一个依赖属性 `Value` 和一个路由事件 `ValueChangedEvent`。属性 `Value` 表示文本框中的当前数值, 当 `Value` 值发生改变时会触发 `ValueChangedEvent` 事件。

1. 自定义依赖属性

自定义一个依赖属性的代码如代码 20-8 所示。

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
        "Value", typeof(decimal), typeof(NumericUpDown),
        new FrameworkPropertyMetadata(MinValue, new
    PropertyChangedCallback(OnValueChanged), new CoerceValueCallback(CoerceValue)));

    public decimal Value
    {
        get { return (decimal)GetValue(ValueProperty); }
        set { SetValue(ValueProperty, value); }
    }
    private static object CoerceValue(DependencyObject element, object value)
    {
        decimal newValue = (decimal)value;
        NumericUpDown control = (NumericUpDown)element;
        newValue = Math.Max(MinValue, Math.Min(MaxValue, newValue));
        return newValue;
    }
```

代码 20-8 自定义一个依赖属性

`ValueProperty` 在注册时定义了 `CoerceValue` 回调函数, 是用来将 `Value` 值限制在最小值和最大值之间; 此外还定义了一个 `OnValueChanged` 回调函数在 `Value` 值发生变化时被调用。这个回调函数最重要的是通过触发路由事件 `ValueChangedEvent` 通知用户 `Value` 值已经发生改变。

2. 自定义路由事件

自定义路由事件的代码如代码 20-9 所示。

```
①     public static readonly RoutedEvent ValueChangedEvent =
    EventManager.RegisterRoutedEvent("ValueChanged", RoutingStrategy.Bubble,
        typeof(RoutedPropertyChangedEventHandler<decimal>),
        typeof(NumericUpDown));

②     public event RoutedPropertyChangedEventHandler<decimal> ValueChanged
    {
        add { AddHandler(ValueChangedEventArgs, value); }
        remove { RemoveHandler(ValueChangedEventArgs, value); }
    }
    private static void OnValueChanged(DependencyObject obj,
    DependencyPropertyChangedEventArgs args)
    {
```

```

        NumericUpDown control = (NumericUpDown)obj;
        RoutedPropertyChangedEventArgs<decimal> e = new
        RoutedPropertyChangedEventArgs<decimal>(
            (decimal)args.OldValue, (decimal)args.NewValue, ValueChangedEvent);
        control.OnValueChanged(e);
    }
    protected virtual void
    OnValueChanged(RoutedPropertyChangedEventArgs<decimal> args)
    {
        ③     RaiseEvent(args);
    }

```

代码 20-9 自定义路由事件

路由事件和依赖属性一样也需要注册，通过 EventManager 的 RegisterRoutedEvent 方法指定了事件的名称、事件路由的策略、委托的类型和拥有该事件的类（代码①）。定义并注册了之后需要创建标准的.NET 事件包装路由事件（代码②），最后在属性改变的回调函数中通过 RaiseEvent 触发该事件（代码③）。

同时为“增加”按钮和“减少”按钮添加两个 Click 事件处理函数，在该函数中增加和减少 Value 值，如代码 20-10 所示。

```

private void upButton_Click(object sender, EventArgs e)
{
    Value++;
}
private void downButton_Click(object sender, EventArgs e)
{
    Value--;
}

```

代码 20-10 为两个按钮添加两个 Click 事件处理函数

3. 使用自定义控件

首先在 MainWindow.xaml 页面中声明命名空间，如代码 20-11 所示。

```
<Window .....xmlns:local ="clr-namespace:mumu_customcontrol".....>
```

代码 20-11 声明命名空间

将自定义控件摆放在面板中，如代码 20-12 所示。

```

<Grid>
    .....
    <local:NumericUpDown      x:Name="numericupdown"      Grid.ColumnSpan="2"
valueChanged="numericupdown_ValueChanged" Margin="10"/>
    .....

```

代码 20-12 将自定义控件摆放在面板中

我们可以为自定义的路由事件 ValueChanged 添加一个事件处理函数，在其中获得 Value 修改后的值并显示在文本框中，如代码 20-13 所示。

```

private      void      numericupdown_ValueChanged(object      sender,
RoutedPropertyChangedEventArgs<decimal> e)

```

```
{  
    txt.Text = e.NewValue.ToString();  
}
```

代码 20-13 为 ValueChanged 添加一个事件处理函数

由于控件的 Value 属性本身是一个依赖属性，因此可以直接通过数据绑定方式将控件的 Value 属性绑定到文本框的 Text 属性直接显示，如代码 20-14 所示。

```
<TextBlock FontSize="20" Text="{Binding ElementName=numericedropdown, Path=Value}"  
Grid.Row="1" HorizontalAlignment="Center" Grid.Column="1"/>
```

代码 20-14 通过数据绑定的方式显示控件的 Value 属性

20.5 无外观控件

20.5.1 无形才是有形

木木看到“无外观控件”5个字时感到非常奇怪，按钮的外观是一个矩形，鼠标点上去会有压下去的感觉。如果外观都没有，还能称上控件吗？由于这个命题和天下武术能够相互印证，因此我们还是引用《笑傲江湖》中风清扬传剑给令狐冲的一段话吧：

风清扬道：“活学活用，只是第一步。要做到出手无招，那才真是踏入了高手的境界。你说‘各招浑成，敌人便无法可破’，这句话还只说对了一小半。不是‘浑成’，而是根本无招。你的剑招使得再浑成，只要有迹可寻，敌人便有隙可乘。但如你根本并无招式，敌人如何来破你的招式？”令狐冲一颗心怦怦乱跳，手心发热，喃喃地道：“根本无招，如何可破？根本无招，如何可破？”斗然之间，眼前出现了一个生平从所未见，连做梦也想不到的新天地。

——《笑傲江湖》，“第十章 传剑”⁽¹⁾

上面的是风清扬传剑给令狐冲，以下就是木木和风老前辈的对话：

“老前辈，为什么还会有无外观控件？”

“刚才你使用 UserControl 自定义控件最大的缺点是什么呢？”

“其实我没太多体会，我觉得还是非常方便的。但是《葵花宝典》上说，最大的缺点就是用户不能自定义控件的外观。”

“为什么不能自定义控件的外观呢？”

木木陷入了沉思。

风老前辈见木木不语，顿了顿说：“最大的原因是它已经有了一个固定的外观，倘若这个控件没有外观，那么它不就可以千变万化有任意外观了吗？无即是有啊。”

木木不语，心想虽然老前辈讲得很有道理，但是也太玄了一点吧，问题是如何落实呢？

风老前辈似乎看穿了木木的心理，说：“你是想我在故弄玄虚吧。”

木木连忙作揖说：“不敢，老前辈。”

风老前辈哈哈一笑，说：“其实落实也非常简单，那就是通过 C# 实现控件的行为，注意它仅仅只是实现控件的行为。而控件的外观全部交给模板，这样才真正实现了控件的行为和外观的分离。这也是符合微软的设计初衷，将控件的行为交给开发人员实现，而将控件的外观交给设计人员实现。”

“你来看。”风老前辈打开了他破桌上的笔记本。仍然是刚才的自定义控件，但是这一次是从 Control 派生而来。说完风老前辈将 NumericUpDown 的父类由 UserControl 改成了 Control，如代码 20-15 所示。

```
public class NumericUpDown : Control
{
    ...
}
```

代码 20-15 将 NumericUpDown 的父类由 UserControl 改成了 Control

木木仔细看了看，非常不解。问：“老前辈，我觉得这个行为和外观实在是难以分离。”

他指了指代码 20-16 和代码 20-17 所示的代码说：“您看，这两个按钮的事件处理函数是以 C# 代码的方式编写的。但是我们无论怎么用 XAML 定义外观，必须要有两个 RepeatButton，而且按钮事件处理函数的名称必须是 upButton_Click 和 downButton_Click。因此虽然微软的理想是好的，但也只是徒有虚名的行为与外观分离。”

```
NumericUpDown.xaml.cs
private void upButton_Click(object sender, EventArgs e)
{
    Value++;
}
private void downButton_Click(object sender, EventArgs e)
{
    Value--;
}
```

代码 20-16 两个按钮的事件处理函数

```
NumericUpDown.xaml
<RepeatButton Name="upButton" Click="upButton_Click"
    Grid.Column="1" Grid.Row="0">增加</RepeatButton>

<RepeatButton Name="downButton" Click="downButton_Click"
    Grid.Column="1" Grid.Row="1">减少</RepeatButton>
```

代码 20-17 两个按钮的事件处理函数

风老前辈听了这番话，非但没有任何不快，反而大有喜色，说：“木木，你的话并不是没有道理。只有良好的设计者才能将行为和外观尽可能地分离，形成一种松耦合的关系。你刚才的问题其实有解决办法，那就是命令。”

20.5.2 定义命令

要实现行为和外观的分离，需要遵循的一个原则就是不要定义事件处理函数，而是定义命令。仍然是上例，我们通过命令将控件的行为和外观解耦，使其形成一种松耦合的关系。实际上，行为和外观的完全分离是不可能的，但是可以通过良好的设计使它们尽可能地松散。为此分别定义两个命令，一是 IncreaseCommand，表示增加值；二是 DecreaseCommand，表示减少值，如代码 20-18 所示。

```
private static RoutedCommand _increaseCommand;
private static RoutedCommand _decreaseCommand;
public static RoutedCommand IncreaseCommand
{
    get
    {
        return _increaseCommand;
    }
}
public static RoutedCommand DecreaseCommand
{
    get
    {
        return _decreaseCommand;
    }
}
```

代码 20-18 定义 IncreaseCommand 和 DecreaseCommand 两个命令

我们通过 CommandManager 的 RegisterClassCommandBinding 方法将这两个命令注册为与类关联的命令，这样做的好处是将两个命令固化到类中用户不能轻易地修改。通过注册绑定 IncreaseCommand 和 OnIncreaseCommand，以及 DecreaseCommand 和 OnDecreaseCommand 静态函数。不过最终还是会分别调用控件的 OnIncrease 和 OnDecrease 方法，如代码 20-19 所示。

```
private static RoutedCommand _increaseCommand;
private static RoutedCommand _decreaseCommand;
public static RoutedCommand IncreaseCommand
{
    get
    {
        return _increaseCommand;
    }
}
public static RoutedCommand DecreaseCommand
{
    get
    {
        return _decreaseCommand;
    }
}
private static void InitializeCommands()
{
    _increaseCommand = new RoutedCommand("IncreaseCommand",
typeof(NumericUpDown));
    CommandManager.RegisterClassCommandBinding(typeof(NumericUpDown),
new CommandBinding(_increaseCommand,
OnIncreaseCommand));
    CommandManager.RegisterClassInputBinding(typeof(NumericUpDown),
```

```

        new InputBinding(_increaseCommand, new
KeyGesture(Key.Up)));
    _decreaseCommand = new RoutedCommand("DecreaseCommand",
typeof(NumericUpDown));
    CommandManager.RegisterClassCommandBinding(typeof(NumericUpDown),
new CommandBinding(_decreaseCommand,
OnDecreaseCommand));
    CommandManager.RegisterClassInputBinding(typeof(NumericUpDown),
new InputBinding(_decreaseCommand, new
KeyGesture(Key.Down)));
}

private static void OnIncreaseCommand(object sender, ExecutedRoutedEventArgs e)
{
    NumericUpDown control = sender as NumericUpDown;
    if (control != null)
    {
        control.OnIncrease();
    }
}
private static void OnDecreaseCommand(object sender, ExecutedRoutedEventArgs e)
{
    NumericUpDown control = sender as NumericUpDown;
    if (control != null)
    {
        control.OnDecrease();
    }
}

protected virtual void OnIncrease()
{
    Value++;
}
protected virtual void OnDecrease()
{
    Value--;
}

```

代码 20-19 将两个命令注册为与类关联的命令

而整个命令的初始化实际上在静态构造函数中完成，如代码 20-20 所示。

```

static NumericUpDown()
{
    DefaultStyleKeyProperty.OverrideMetadata(typeof(NumericUpDown),
        new FrameworkPropertyMetadata(typeof(NumericUpDown)));
    InitializeCommands();
}

```

代码 20-20 在静态的构造函数中完成整个命令的初始化

代码 20-20 中有一行出现了一个奇怪的属性 DefaultStyleKeyProperty，该行代码的意思是让该控件支持主题，控件的外观正是在一个默认的主题文件 generic.xaml 文件中定义的。

20.5.3 在主题中定义控件外观

在 Windows 操作系统中实际上存在不同的主题，如果在桌面上右键选择属性，即可看到当前和备选

的主题；如果切换到外观选项卡，还能查看到当前和备选的色彩方案，如图 20-14 所示。



图 20-14 在 Windows 操作系统中变换主题外观

WPF 在设计时考虑到了同一个控件可以依据不同主题风格的操作系统来变换其外观，这是一个不错的想法。WPF 约定用于特定主题的资源字典文件的名称和存放的位置，它们被要求放在项目文件夹下的 Theme 子文件夹中。表 20-3 所示是一些针对不同主题资源字典的常用文件名。

表 20-3 不同主题资源字典的常用文件名

操作系统	基本主题名	主题颜色名	文件名
Windows Vista	Aero	NormalColor	Areo.NormalColor.xaml
Windows XP(蓝色， 默认)	Luna	NormalColor	Luna.NormalColor.xaml
Windows XP(橄榄绿)	Luna	Homestead	Luna.Homestead.xaml
Windows XP(银白色)	Luna	Metallic	Luna.Metallic.xaml
WindowsXP 或者 WindowsVista	Classic		Classic.xaml

如果还有新的主题风格不在上面的列表中，WPF 规定一个 generic.xaml 文件为默认的主题资源字典文件。即如果 WPF 找不到特定主题风格的资源字典文件，则使用 generic.xaml 文件中提供的控件外观。

我们在 generic.xaml 文件中定义这个控件的外观，如代码 20-21 所示。

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:mumu_customcontrolfromcontrol">
    <Style TargetType="{x:Type local:NumericUpDown}">
        <Setter Property="HorizontalAlignment" Value="Center"/>
        <Setter Property="VerticalAlignment" Value="Center"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:NumericUpDown}">
```

```

<Grid Margin="3">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Border BorderThickness="1" BorderBrush="Gray" Margin="2"
            Grid.RowSpan="2" VerticalAlignment="Center"
            HorizontalAlignment="Stretch">
        ① <TextBlock Text="{Binding
                    RelativeSource={RelativeSource TemplatedParent}, Path=Value}" Width="60"
                    TextAlignment="Right" Padding="5"/>
        </Border>
        <RepeatButton
            ② Command="{x:Static
                        local:NumericUpDown.IncreaseCommand}"
            Grid.Column="1" Grid.Row="0">Up</RepeatButton>
        <RepeatButton
            ③ Command="{x:Static
                        local:NumericUpDown.DecreaseCommand}"
            Grid.Column="1" Grid.Row="1">Down</RepeatButton>
    </Grid>
    </ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</ResourceDictionary>

```

代码 20-21 在 generic.xaml 文件中定义控件的外观

代码 20-21 有如下 3 处需要注意。

(1) 由于控件的外观已经不在 UserControl 中, 而是改在模板中定义, 因此 TextBlock 的 Text 属性需要绑定到 TemplatedParent, 即绑定到使用该模板的控件 NumericUpDown (代码①)。

(2) RepeatButton 不再需要声明事件处理函数, 而是将命令设置为规定的 IncreaseCommand 和 DecreaseCommand (代码②和③)。

同样可以为控件添加多个主题风格的资源文件, 图 20-15 所示为不同主题下 NumericUpDown 控件的外观。

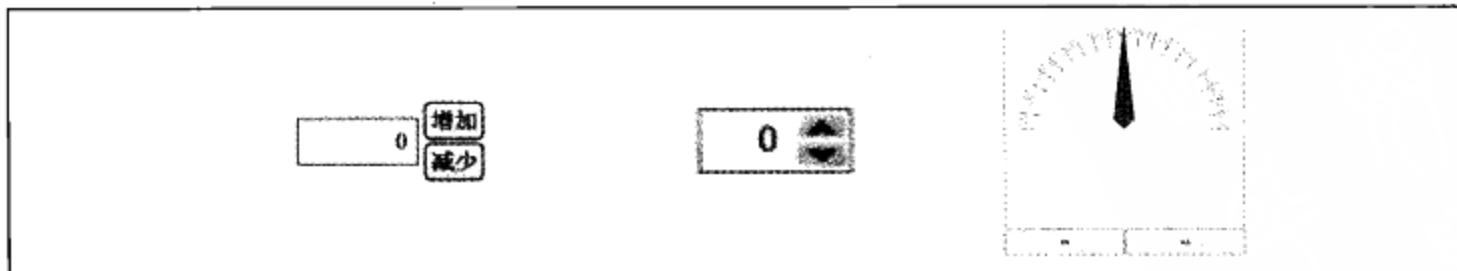


图 20-15 不同主题下的 NumericUpDown 控件的外观

20.6 接下来做什么

前面介绍的资源、样式、依赖属性、路由事件、命令及控件模板等都是死的，只有灵活应用才能融会贯通。如果希望对控件深入研究，笔者慎重推荐《WPF Control Development unleashed: building advanced user experiences》一书。

接下来，我们进入最后一章。

参考文献

- [1] “北风之神”风清远整理，“云中孤雁”制作《金庸全集典藏版面 笑傲江湖》，“第十章 传剑”。
- [2] Charles Peztold 著，段菲及陈正华译，《精通 Windows 3D 图形编程》。
- [3] Pavan Podila,Kevin Hoffman 著 WPF Control Development Unleashed ,Chapter3 Getting Started Writing Custom Control。
- [4] MSDN Library for Visual Studio 2008 SP1 NumericUpDown UserControl with DependencyProperty and RoutedEvent Sample。

第七卷

志向无限大

木木能行，我也行

2009 年的冬天格外漫长，一天寒风萧杀，我突然才发现竟然找不到一个人吃饭，有时孤独的感觉就这样顷刻而至。相信读者短则三五天，长则三五个月将本书看完。不知道看完后做何感想，难道真有一股内力由脚而发。升至头顶，左突右出，抑制不住吗？我想在这里有责任和义务告诉各位《葵花宝典》的秘密。

葵花宝典的真正秘密

我相信世间有天才，笑谈中就将所有的知识尽收囊中，当然他不必读此书。但是我们绝大多数人都和木木一样是一个平凡的人，上一个平庸的大学。恍恍惚惚度过 4 年，谈过一次或者几次荒唐的恋爱，为《我的青春谁做主》流过眼泪，看过《蜗居》之后不再相信男人或者女人……

如果读者是一个工科大学生，特别不幸地是工作又涉及了计算机编程，并恰巧是在 Windows 上进行桌面编程，那么了解 Windows 最新的桌面技术是必须的。如果读者的年龄还稍长，应该接触过少量的 Win32，操刀 MFC 大张旗鼓干项目。然后转到 WinForm，那么更应该看看 WPF。

如果读者从事网络编程，尤其从事前端编程，那么真心恭喜你。因为现在已经不是一个桌面流行的年代了，在网络大行其道的今天，WPF 有点生不逢时的感觉。但是相信我，它和 SilverLight 相通。就好像一个精通星际争霸的高手，很快就能转型为魔兽的顶尖高手。况且微软是凭着桌面起家的，在学习 Silverlight 之前，体味一下 WPF 又何如？笔者敢肯定一个精通 WPF 的人也一定会成为一个 Silverlight 高手。

如果读者一直从事设计，使用 PhotoShop 或者 Flash 等设计工具，并且能够抽出时间来学习 WPF，很有可能成为真正意义的复合型人才。如果读者是一个优秀的设计人才，又掌握了 WPF，那么你就真是一个真正意义的复合型人才。因为你可以把优秀的设计用程序变为现实。

但是我们还有一个关键的问题没有解决。

读完了这本书，又能怎样呢？

答案是不能怎样，你依旧还是做个平凡的人。掌握了一门新的工具也许能够承担一个小的项目，也许年终老板表扬你，但是终究还是会平平淡淡地生活。

生活毕竟如此，即使平淡，我们也要努力地生活，努力地提升自己，即使提升后我们还是平淡。这是笔者写此书的目的，也是读者阅读此书的目的。这也是葵花宝典的真正秘密，即使平淡，我们也要努力，好好地生活。

写给大学生

笔者是一个大学老师。因此本想写个结尾就结束，但是忍不住有些话想和正在犹豫不决买书的你——一个一个刚刚跨入校门或者一个即将毕业虔诚地幻想未来的大学生再唠叨上几句。

我们都是平凡的人，没有葵花宝典，唯有苦学。学习并不是一件轻松的事，希望你一章一章看过去，一段一段代码地运行，本着怀疑一切和批判一切的精神看书。可以和笔者交流（邮箱为 LoveHelloj2ee@126.com，虽然无法保证回复每个邮件，但是能够保证在笔者的博客上有集中的解答），也可以更多地阅读 MSDN，MSDNMagizine，Codeproject，Codeplex。

毫不夸张地说，在写本书时笔者已经阅读过市面上和国外的几乎所有 WPF 书籍。但自始至终笔者都认为 MSDN 算是学习微软技术最权威的资料。不过对于初学者来说，还是需要一本书。市面上有很多译著，但是技术书籍原汁原味的还是好很多。《葵花宝典》只是引路，从本书开始阅读到无尽的英文技术资料，笔者甚是快慰。笔者向来认为学习计算机技术算法为魂，英文次之。如果读者在一公司打工，英文确实躲也躲不过去。尤其一整天和技术打交道的人不会看英文资料，真的很难混。

学习英文，八仙过海，各显神通，笔者大致经历了如下 3 个阶段。

(1) 咖啡与金山词霸阶段，即拿一本英文技术书，一杯咖啡入肚。趁着一股暖流打开金山词霸，正襟危坐地逐字逐句地看。

(2) 背上中英文图书上路的阶段，由于经常出差，所以在火车上笔者经常会带一本英文技术书，然后相应地带一本中文译著。由于手边缺少金山词霸，因此只好阅读英文。不懂再掏出一本中文书，对比查之，这样读书的效果非常显著。

(3) “横着看，竖着看，躺着看，睡着看”阶段，笔者尚未达到这个阶段。毕竟英文不是母语，不能一目十行。

有了一定英文的基础，就要坚持多看一些 MSDN 或者技术性很强的博客。这样眼界就会越来越高，越来越宽。慢慢葵花宝典就在脚下，回过头来你就会笑笑，葵花宝典，何来葵花宝典？

本书的主人翁叫做“木木”实在是希望本书的主角能够持之以恒。相信读者也能够，耐得住寂寞坚持。不抛弃，不放弃。

木木能行，我也行！我们一定能行！

致谢

最后在这里我要感谢所有为这本书付出了辛勤劳动的各位同事、好友和亲人。如果没有你们，就很难有这本书的问世。下面我会循着我的记忆去寻找在这本书出生的路上扮演航灯的你们。

谢谢王双，如果没有你，我就不可能有源动力去写这本书；

谢谢王德才，如果不是你给我开这扇窗，我也不可能勇敢地踏上作者这条路；

谢谢孙学瑛编辑，如果没有您，当然是不会有这本书，谢谢您在我写作的路上给予的欣赏、肯定、建议和耐心；

谢谢徐彩虹，如果没有你，书中会少了很多故事，少了很多的优雅；

谢谢孙丰垒，如果没有你，我可能还在为数据绑定和控件两章而抓耳挠腮；

谢谢刘静、王红科、秦宝华、王志坚、王丽娜和李承武，你们为我承担了校对、制作视频、制作 PPT 和绘制类图等等琐碎又很重要的工作。如果没有你们，我何以堪；

谢谢腾讯的法拉利（网名）和你组建的 WPF Silverlight 群以及你的棒棒牛 WPF/Silverlight 开发技术中文社区，对你的仰慕就犹如滔滔江水……^_^；

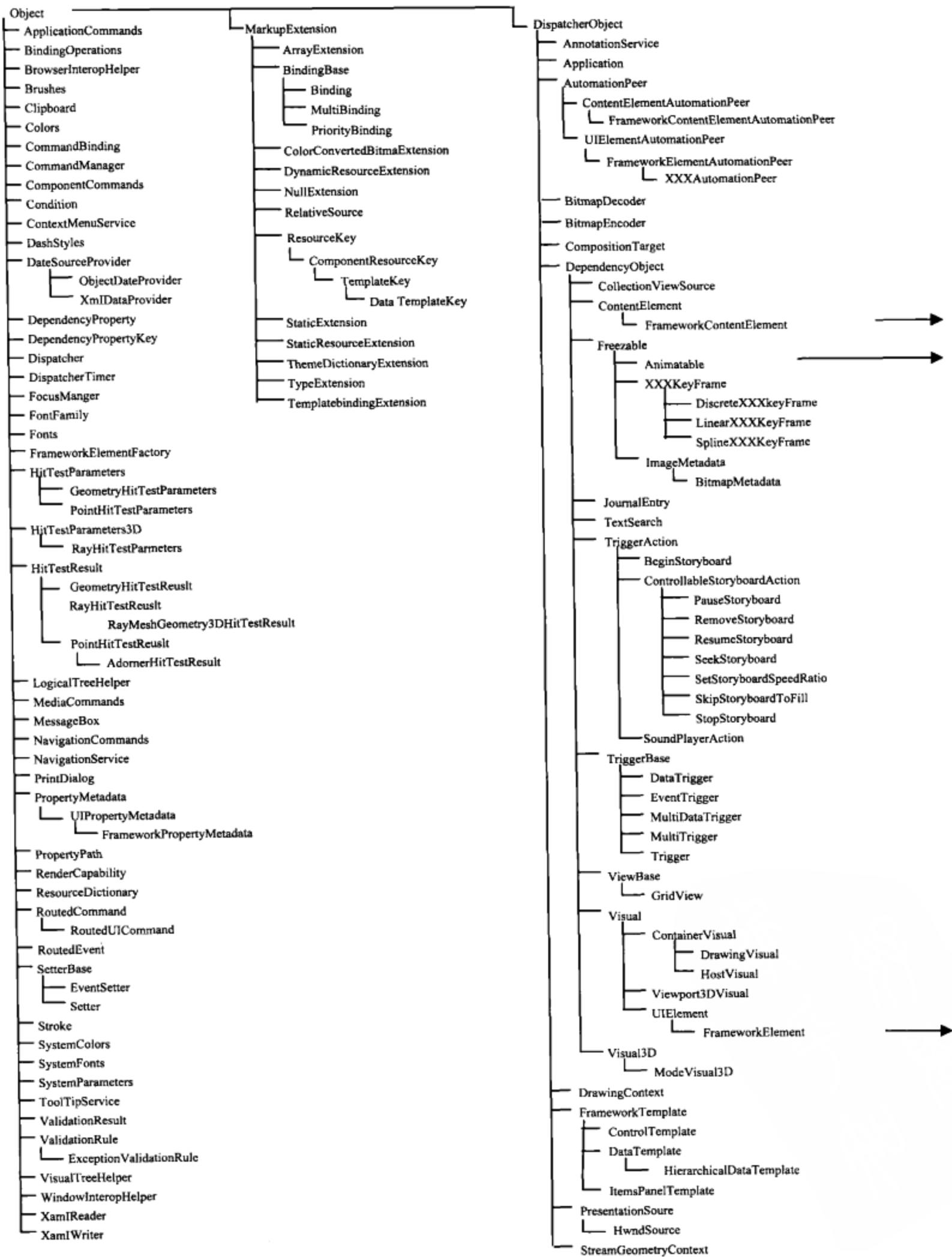
谢谢程序员的执行总编孟迎霞老师，每次遇到困难都会想起您；

在我写作和学习的过程中，还有很多微软的同仁，网上的博友无私地给予了我帮助，还有像 Charles Peztold 等等这样的大师著作给予了我不尽的启发，如果有一天见到你们，真要说一声谢谢。

最后要谢谢我的导师华一新教授，没有您的支持和耐心，学生又何能扬扬洒洒写下几十万字。

这本书是我的第一个作品，它恍若我的一个小孩，我经历了从他的萌芽到呱呱坠地。因此我最想把这本书献给我的父母，献给我的姐姐和姐夫，希望他们看到这儿，能够欣慰地一笑，永远快乐，永远健康。

附录A WPF类图





技术凝聚实力 专业创新出版

博文视点 (www.broadview.com.cn) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台，博文视点致力于——IT专业图书出版，为IT专业人士提供真正专业、经典的好书。

请访问 www.dearbook.com.cn (第二书店) 购买优惠价格的博文视点经典图书。

请访问 www.broadview.com.cn (博文视点的服务平台) 了解更多更全面的出版信息；您的投稿信息在这里将会得到迅速的反馈。

博文本版精品汇聚



加密与解密（第三版）

段钢 编著
ISBN 978-7-121-06644-3

定价：69.00元
畅销书升级版，出版一月销售10000册。
看雪软件安全学院众多高手，合力历时4年精心打造。



疯狂Java讲义

新东方IT培训广州中心
软件教学总监 李刚 编著
ISBN 978-7-121-06646-7

定价：99.00元（含光盘1张）
用案例驱动，将知识点融入实际项目的开发。
代码注释非常详细，几乎每两行代码就有一行注释。



Windows驱动开发技术详解

张帆 等编著
ISBN 978-7-121-06846-1
定价：65.00元（含光盘1张）

原创经典，威盛一线工程师倾力打造。
深入驱动核心，剖析操作系统底层运行机制。



Struts 2权威指南

李刚 编著
ISBN 978-7-121-04853-1
定价：79.00元（含光盘1张）

可以作为Struts 2框架的权威手册。
通过实例演示Struts 2框架的用法。



你必须知道的.NET

王涛 著
ISBN 978-7-121-05891-2
定价：69.80元

来自于微软MVP的最新技术心得和感悟。
将技术问题以生动易懂的语言展开，层层深入，以例说理。



Oracle数据库精讲与疑难解析

赵振平 编著
ISBN 978-7-121-06189-9
定价：128.00元

754个故障重现，件件源自工作的经验教训。
为专业人士提供的速查手册，遇到故障不求人。



SOA原理·方法·实践

IBM资深架构师毛新生 主编
ISBN 978-7-121-04264-5
定价：49.8元

SOA技术巅峰之作！
IBM中国开发中心技术经典呈现！



VC++深入详解

孙鑫 编著
ISBN 7-121-02530-2
定价：89.00元（含光盘1张）

IT培训专家孙鑫经典畅销力作！

博文视点资讯有限公司
电 话：(010) 51260888 传 真：(010) 51260888-802
E-mail：market@broadview.com.cn(市场)
editor@broadview.com.cn jj@phei.com.cn(投稿)
通信地址：北京市万寿路173信箱 北京博文视点资讯有限公司
邮 编：100036

电子工业出版社发行部
发 行 部：(010) 88254055
门 市 部：(010) 68279077 68211478
传 真：(010) 88254050 88254060
通 信 地 址：北京市万寿路173信箱
邮 编：100036

博文视点·IT出版旗舰品牌