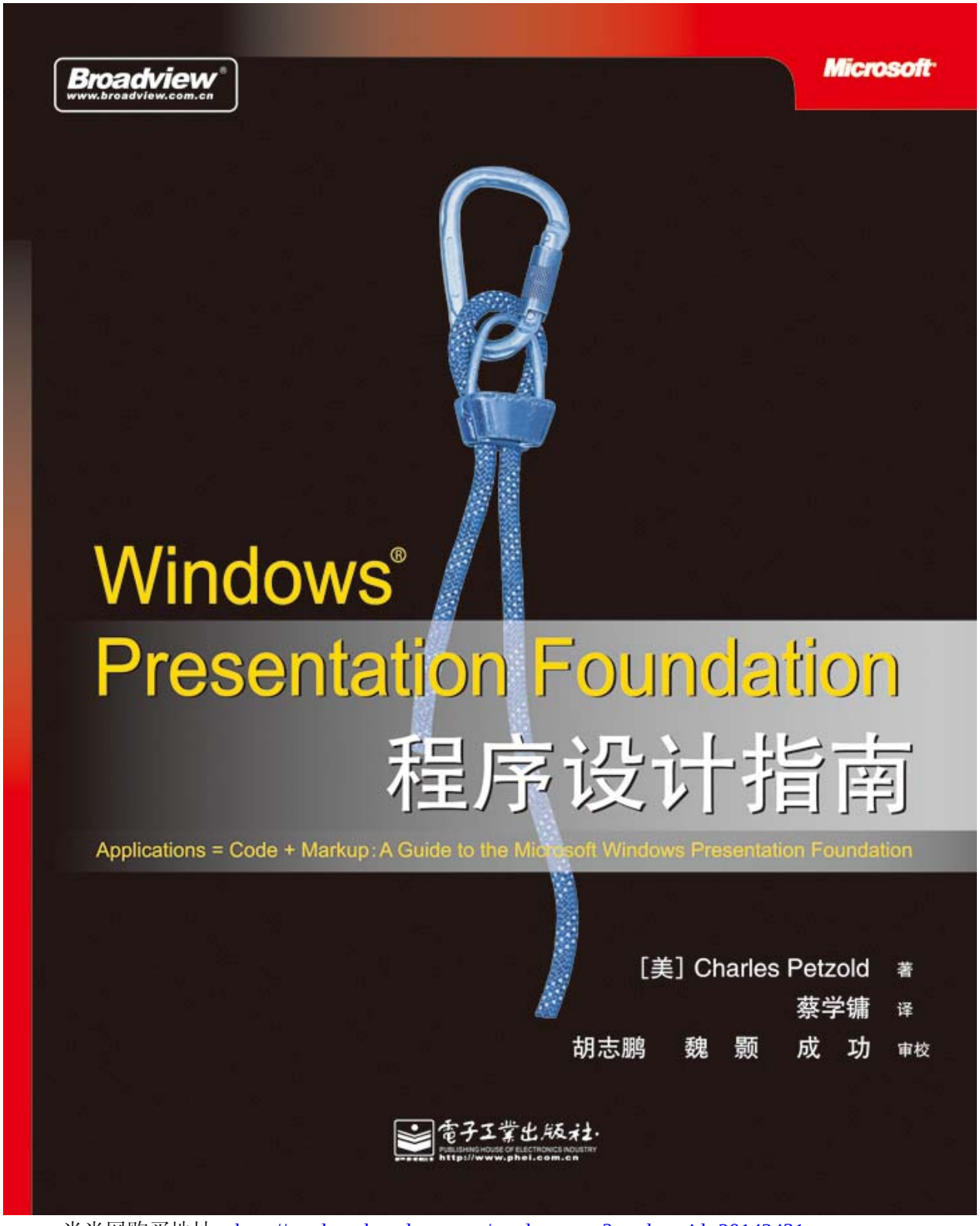

当当网购买地址：http://product.dangdang.com/product.aspx?product_id=20142431

互动网购买地址：http://www.china-pub.com/computers/common/info.asp?id=37794

新一代界面编程体验

在 GUI 编程的发展历程中，伴随着 Windows 视窗操作系统出现的 GDI 技术，把 GUI 编程引入了一个新的阶段。GDI 技术统治了近 20 年的 GUI 应用。随着应用要求的提高，Misrosoft 提供了增强的 GDI 技术：GID+。她在支持渐变功能、象素透明度等等方面进行显著的改进来弥补 GDI 技术在这种方面的不足。但是，运行 GDI 和 GDI+时所使用的界面元素仍然由操作系统 API 提供。另一方面，Microsoft 推出了新的开发技术平台.Net。在这个新的框架体系中，置入了新的 Windows Forms 和 Web Forms 两个 GUI 编程应用，分别拥有传统的桌面应用程序和网页开发。而且，.NET 框架内置了大量常用的基本元素来提高界面编程的效率。所有这些变化都短短的几年时间完成，这一切都预示着 GUI 编程正在经历一个快速发展的时期。

虽然 Windows Forms 和 GDI+等已能满足几乎所有的应用，但随着硬件技术的飞速发展，用户对视觉体验的要求也更高。因此，开始出现一些用 DirectX 技术封装实现的高效 GUI 程序库。毫无疑问，应用 DirectX 技术实现的 GUI 将会更绚、更快，但它们的开发成本也是不可小视的。

正是在这样的环境下，Microsoft 的 Windows Presentation Foundation 技术应运而生。WPF 作为.NET 3.0 的核心组件之一，她运用 Direct3D 技术提供了一套全新的 GUI 编程框架，并在此基础上提供了更加丰富的窗口元素。WPF 打破了传统的 GUI 实现方式，分离了界面元素的外观和行为。例如，按钮的点击行为和她的外观无关，技术按钮的内容是一张图片，该按钮也能完美的响应点击事件。同时，WPT 利用时下最广泛的 XML 技术，引入了一种新的语言 Xaml。Xaml 是一种声明式的界面描述语言，通过它我们可以直接用简单的标记代码来实现界面。最为重要的是，Xaml 语言帮助我们实现了程序界面和应用逻辑的分离。我们可以用 Xaml 实现界面，而使用 C#或者 VB.NET 实现应用逻辑，两者可以实现真正的无缝合成。

本书《Application=Code+Markup》的组织形式正是基于 WPF 的这种思想，分为代码和 Xaml 标记两部分来陈述 WPF 技术。通过代码部分的学习，我们可以掌握 WPF 框架体系的结构和界面元素的功能细节。而在 Xaml 标记部分，我们可以感受到 Xaml 的便捷、强大。并巧妙结合二者来实现高效、快速的应用开发。

本书的作者是 Charles Petzold 先生，他已是 Windows 编程领域的泰斗。Charles Petzold 用非常详细的的叙述和大量简单、精干的示例展示了 WPF 技术的每个细节。作为译者的蔡学镛老师，拥有多年的开发、写作经历。很好地向我们展示了该书所表达的含义。我们有理由相信（事实也确实如此）该书是学习 WPF 不可或缺的参考书籍。

赖仪灵

《深入解析ATL（第 2 版）》译者

2008 年 1 月 1 日 上海

步入.NET3.0 开发殿堂

WPF(Windows Presentation Foundation) 在框架上彻底改变了它的前辈 WinForm 架构设计。如果说 WinForm 多少还保留着 Win32 或者 MFC 的结构，那么在 WPF 中我们看到的是完全不同的新的形式。

以往的平台中，每个可视化控件只负责自己在屏幕上所占据的范围，这使得要实现特殊的效果变得很困难。

WPF 把整个窗口看成一个整体，窗口上的每个控件可以在屏幕的任何位置进行绘画，不再受控件本身范围的限制。

WPF 这个翻天覆地的变化，对我们技术人员来说，是一个非常大的门槛。很多之前的经验将不再适用。如何跨过这个门槛，是我们都要面对的一个问题。

Charles Petzold 的 《Applications = Code + Markup》这本书就很适合帮助我们跨过这个门槛。

细节决定成败，如果你想看上几十页就上手编程，这本书肯定不适合你。《Applications = Code + Markup》是循序渐进的讲解很多语言内部的东西和细节上的东西。而这些细节的积累才能帮助我们跨过 WPF 编程这道门槛。

这本书首先是一本：“学习用书”。书中每一个例子都很精短，但是不少例子都可以实现非常非常酷的功能。通过例子讲解知识，这对我们的学习非常有帮助。几乎每一个知识点都有一个配套的例子。在学习一个知识点后，可以动手做个酷的程序出来，学习的成就感就在这里体现出来了。

这本书又可以做一本非常有价值的“参考用书”。壹千页的容量让它可以包含非常多的知识点。据说：微软内部的不少 WPF 开发人员就使用本书作为必备参考书。

最后一点，WPF 中引入了 XAML 的标记语言，微软的 WPF 设计软件 Microsoft Expression Design 就可以让美工设计出非常酷的 WPF 界面，但是这些设计的产生的结果文件是 XAML 文件。

我有一个感觉，也许几年后，大多数的程序员，设计界面的时候，都是使用这些设计工具，而不是手写 XAML，或者手写 Code。要想写出性能高，精简的 WPF 程序，就必须懂 WPF 的各种细节。

这本书命名为《Applications = Code + Markup》，就是告诉我们 WPF 程序的各种效果，既可以用 Code 写出来，也可以用标志语言 XAML 写出来。本书章节安排上，上半部书写 Code 实现 WPF 效果，后半部书写 XAML，就是要让书友避免陷入不明根源，只知道用软件设计 WPF 的问题。

蝈蝈俊

我们准备用什么创造未来的应用

.NET 3.0 的推出又把我们中的很多人推倒一个十字路口：我们是否应该重新去学习一轮全新的开发技术还是经营自己已经坚守多年的技术呢？不论我们作何选择，但 Charles Petzold 告诉我们世界变了，未来的应用是 Code + Markup。和很多同行一样，10 多年前开始编写第一行 Windows 代码之前看的就是 Charles Petzold 的书，但就在不久之前随着 Vista 的推出，Windows API 的概念从浩繁的 C/C++函数过渡到.NET 语言和一组 Markup 了，正当我准备再次用也许真的很笨办法——遍历 Windows Vista SDK 的 WPF、WCF 和 WF 部分的时候看到了 Charles Petzold 的新书，虽然他着力于 WPF，但给我们一个全新的启示——Markup 在未来应用中的作用。

以前，代码是程序员间无间的纽带，但随着应用越来越复杂，单单几行代码似乎能描述的内容越来越片面和乏力了，不过 XML 给了我们一个新的选择，让我们可以在不同抽象层次定义应用的描述，其中 XAML 作为一个代表，通过它程序员可以告诉 Windows“我就要这样的界面效果”了。Markup 不仅仅是一点点技术的技巧，它打开了隔离不同开发语言、不同 Windows 平台甚至是异构平台间屏障，让我们用一个方式、一次编写描绘计算无处不在的世界中的应用。但这并不是说代码成为历史，恰恰相反代码依然程序员赋予应用的灵魂，如果说 Markup 更多的是我们与 Windows 对话的桥梁，那么代码则是实实在在描述“我们自己的”而不是“其他什么的”应用，也就是我们应该更吝惜我们的代码到实际描述自己应用的解决方案之中。

那么该怎么做呢？Charles Petzold 的书中提到：

- 如何创建并进一步增强那些天天要面对的应用界面元素，包括菜单、工具栏、树和列表；
- 如何以更加动感的方式安排控件和图形的布局；
- 如何通过 XAML 的资源和模版来整体替换应用界面的呈现风格；
- 如何通过在 XAML 中采用数据绑定技术简化应用开发的同时令操作体验更为流畅；
- 如何创建并发布面向浏览器客户端的 XAML 应用；
- 如何通过图形、多媒体和动画改善用户交互体验；

此外，我个人非常喜欢 Charles Petzold 稍有些“啰嗦”的风格，因为之前每每实际置身项目之后，才发现当时一些看似“啰嗦”的内容其实却非常必要，感觉就像之前很多技术先锋评价凡尔纳的那样——“生活的导演”，不过不同的是 Charles Petzold 不是导演而是实实在在走在我们前面的向导。

感谢蔡先生和博文视点，他们的的努力让这本书呈现到国内读者的面前，从样章的书稿看，译文忠实而清晰的反映出原文的内质，相信它将启迪正在经历全新一轮开发技术浪潮的众多同行，助力我们用 Code 和 Markup 的双勾拳打拼未来的应用。

王翔（Vision Wang）

2008 开年大礼：《Application = Code + Markup》中文版面世

Charles Petzold 的又一部经典力作《Application = Code + Markup》中文版终于要面世了。成为 2008 开年大礼。相信有很多对 WPF 有兴趣，但又苦于没有经典书籍来支撑的朋友都一直在期待着这本书的中文版上市，博文视点让这一期待成为现实。 与大家一样都很兴奋。因此在这里也大家一起分享之。

其实早在 2007 年 5 月份就在圣殿祭司那拿到了英文原版的书，初看了几章节，因工作太忙，加上英文啃起来比较吃力而没有读完，虽然后来也读了不少，但看中文版还是轻松多一些。（自勉之：不要让英文成为学习障碍）不过目前我也没有拿到中文版的书，只有几章的样稿，这本书由台湾牛人蔡学镛翻译，相信更加忠于原著了。估计会在年前面世，那就可以在春节这有限的时间里去啃啃了。

《Application = Code + Markup》原版 2006 年出版，作者 Charles Petzold 自不必多说了。书名：WPF 程序设计指南 Application = Code + Markup ，初见书名的时候还真的愣了一下。现在想想，过不了几年，这世界又要变了。

他是要告诉我们，今后的应用程序就是 Code,比如：C#、VB… 与 XAML 的天下了？

还是来看看它的章节吧，全书有千页之多，但一共仅分为两个部份（我们知道要实现 WPF 有两种方式，一种是 XAML，一种是 C#等，这不正是 Code + Markup）：

第 1 部分代码 共十八个章节。

第 2 部分 Markup 共三十一个章节。

主要介绍了微软新一代操作系统平台上的 Microsoft Windows Presentation Foundation 核心技术的原理、概念、技术、技巧与开发实践。全书全面细致、深入浅出，主要内容包括 Windows Presentation Foundation 概述、基本 Bushes、Content 概念、Button 及其他控件、Stack、Wrap、Dock、Grid、Canvas（画布）、依赖性属性、Routed Input Event、定制元素等诸多内容。

这是一本关于界面编程的书，但书中没有一处示例截图，如果要想看书中的例程，你必须得下载或手工输入代码，编译后运行它，动手始终是学习一门语言的必经之路。但正因为如此，它对核心技术与原理的讲解是非常地道的。

最后还想说一点，Win32 与 MFC 是无法模拟 WPF 的。但 Win32 与 MFC 能做的 WPF 都能做，并且可以更加容易、轻松的去实现，WPF 给我们很大的想象空间。相信这本千页大作将会是另一个经典。

林焰峰

微软 MVP

目　录

The Application and the Window

第 1 章　应用程序与窗口

为 Windows Presentation Foundation（WPF）开发应用程序，一般来说，一开始需要花一点点时间创建 Application 对象与 Window 对象。下面是一个很简单的 WPF 程序：

SayHello.cs

```
//---------------------------------------
// SayHello.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.Windows;

namespace Petzold.SayHello
{
    class SayHello
    {
        [STAThread]
        public static void Main()
        {
            Window win = new Window();
            win.Title = "Say Hello";
            win.Show();

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

我假设你已经熟悉 System 命名空间（namespace）了，如果你还不熟悉此命名空间，那你或许应该到我的网站 *www.charlespetzold.com* 读我写的另一本书*.NET Book Zero*。上面这个 SayHello 程序利用 using 编译指示符(directive)，将 System.Windows 命名空间加入工程。这个命名空间包含了所有的基本 WPF 类别、结构（struct）、接口（interface）、委托(delegate)、以及枚举类型（enum），其中包括 Application 和 Window 两个类。其他的 WPF 命名空间均以 System.Window 开头，例如 System.Windows.Controls、System.Windows.Input、System.Windows.Media。只有 System.Windows.Forms 是个例外，它主要是 Windows.Forms 的命名空间。除了 System.Windows.Forms.Integration 这个命名空间里的类是用来集成 Windows.Forms 和 WPF 程序的，其他所有以 System.Windows.Forms 开头的命名空间，都属于传统的 Windows.Forms 命名空间。

本书的范例具有一致的命名方式。每个程序都属于一个特定的 Microsoft Visual Studio 工程。工程中的所有代码都被包含在一个统一的命名空间中。我所使用的命名空间，开头是我的姓氏，然后是工程名称。以第一个范例来说，工程名是 SayHello，所以命名空间就是 `Petzold.SayHello`。工程中的每个类都有一个独立的文件，而且文件名一般都会与类名一致。如果工程内只包含一个类（第一个范例正是如此），那么这个类的名字通常会和工程名一样。

任何一个WPF程序，Main的前面都必须有个`[STAThread]`属性(attribute)，否则编译会失败[1]。这个attribute是用来申明该应用程序的初始线程模型为“single threaded apartment”，以便和Component Object Model(COM)兼容。 “single threaded apartment”是旧的COM年代的词汇，是.NET之前的词汇，但基于我们此刻的目的，你可以把它理解成：STAThread表示我们的应用程序不会使用“源自运行环境”的多线程。

在 SayHello 程序中，`Main` 一开始创建一个 Window 类的对象，这个类用来创建标准应用程序窗口。`Title` property 是显示在窗口标题栏里的文字，而 `Show` 方法会将窗口显示在屏幕上。

这里最重要的步骤是，调用 `Application` 对象的 `Run` 方法。在传统 Windows 编程的思维中，这么做的目的是要建立一个消息循环，让应用程序可以接收用户键盘或鼠标输入。如果此程序是在 Tablet PC 上执行，此应用也会接收到来自手写笔（stylus）的输入。

你很可能是使用 Visual Studio 2005 来新建、编译并执行 WPF 应用程序。果真如此，你可以采取下面的步骤来重新建立 SayHello 工程：

1. 从“File”菜单，选取“New Project”。
2. 在“New Project”对话框中，选取“Visual C#”、“Windows Presentation Foundation”和“Empty Project”。指定一个存放该工程的目录，将工程命名为“SayHello”并取消“Create Directory For Solution”，然后按下“OK”[2]。
3. 在右边的“Solution Explorer”中，“References”栏必须包含“PresentationCore”、“PresentationFramework”、“System”以及“WindowsBase”。如果在“References”栏里没有出现这些 DLL，请手动加上它们，做法很简单，只要用鼠标右键选择“Reference”，然后选取“Add Reference” 就可以了 （另一个做法是， 从“Project”菜单中，选取“Add Reference”）。在这个“Add Reference”对话框中，选择“.NET”项，然后选取必要的 DLLS，然后按下“OK”。
4. 在右边的“Solution Explorer”中，用鼠标右键选择 SayHello 工程名，然后从“Add”菜单中选“New Item”（另一个做法是，从“Project”菜单中，选取“Add New Item”）。在“Add New Item”对话框中，选择“Code File”，键入代码文件名“SayHello.cs”， 最后按下“OK”。

[1] 实际上编译可以通过，只是在运行时会抛出System.InvalidOperationException异常。——审校者注

[2] 很不幸，直到本书翻译完毕，Visual Studio 2005 还没有提供新建空WPF工程的模板向导，读者可以到作者的主页http://www.charlespetzold.com/wpf/index.html 上去下载WPF空白工程模板，copy 到 \My Documents\Visual Studio 2005\Templates\ProjectTemplates\Visual,C# 则可以在New Project -> C# 用户自定模板中找到空白WPF模板。——审校者注

5. 将本章前面的代码输入到 SayHello.cs 文件中。
6. 从“Debug”菜单中，选取“Start Without Debugging”（另一种做法是，直接按下 Ctrl+F5），就可以编译并且执行该程序。

本书 Part I 所示的大部分程序，都采用和上面相同的步骤来建立工程，只有某些涉及多个源代码文件的工程做法不太一样（本章就有一个这样的例子）。

在关闭 SayHello 创建的窗口时，你会发现一个 console 窗口也在运行。这是源自编译选项的设定， 你可以在工程的 property 中，修改此编译选项。用鼠标右击工程名，并从弹出菜单中选择“Properties”（另一个做法是， 从“Project”菜单中选择“Properties”）。现在你可以查看工程的各种设定，也可以改变设定。特别注意，“Output Type”被设定为“Console Application”， 显然，这样的设定并不会阻碍用 console 程序来建立 GUI 窗口。 将“Output Type”设为“Windows Application”，程序同样会顺利执行，而这次就不会再出现 console 窗口了。我认为在开发阶段，console 窗口其实是相当有用的。程序运行时我利用它来显示一些文本信息，以便调试。如果程序的 bug 太多，甚至无法将 GUI 窗口显示出来，或者进入无限循环，这个时候，只要在 console 窗口中键入 Ctrl+C, 就可以轻易地关闭程序。这些都是 console 窗口的附带好处。

SayHello 使用到了 `Window` 和 `Application` 类，这两个类都是继承自 `DispatcherObject`，但 `Window` 在继承树中离 `DispatcherObject` 较远，请看下面的类继承树：

Object

DispatcherObject (abstract)

Application

DependencyObject

Visual (abstract)

UIElemcnt

FrameworkElement

Control

ContentControl

Window

当然，你目前可能尚未熟悉此类继承树，但随着你一路走向 WPF 技术，你会一次又一次地遇到这些类。

在一个程序中，只能创建一个 Application 对象，对程序的其他地方来说，此 Application 对象的作用就如同固定的锚一般。你在屏幕上是看不见 Application 对象的，但可以见到 Window 对象。Window 对象出现在屏幕上，这就是正常的 Windows 系统窗口，具有的标题属性（Title property）的值会变成标题栏上的文字。系统菜单在标题栏的左边，最大化、最小化和关闭窗口图标则在它右边。此窗口有一个可以调整窗口大小的边框， 窗口中很大的面积被一个客户区（client area）所占据。

在某些限制下，你可以调整 SayHello 程序中 Main 方法里的语句顺序，程序依然可以执行。比方说，你可以在调用完 Show 之后，才改变 Title property。理论上，这样的改变使得窗口一开始显示时标题栏上没有名字，后来才加上名字，但这个时间差距太短，你可能根本看不出来。

你可以在创建 Window 对象之前，先创建 Application 对象，但对于 Run 的调用必须保留在最后。Run 方法一旦被调用，就不会返回，直到窗口被关闭为止。Run 返回后，Main 方法结束，Windows 操作系统会做一些清除工作。如果你将源代码中调用 Run 的那行删除，Window 对象依然会被创建并显示在屏幕上，但是会在 Main 结束之后，立刻被销毁。

想要将窗口显示出来，你也可以不调用 Window 对象的 Show 方法，而是直接将 Window 对象当作参数传递给 Run 方法：

```
app.Run(win) ;
```

在这种做法中，Run 方法会去调用 Window 对象的 Show 方法。

程序调用 Run 方法之后才真正开始运行。只有在调用 Run 之后，Window 对象才能响应用户的输入。当用户关闭窗口时，Run 方法就会返回，程序也就准备结束。因此，程序运行时几乎所有的时间都是花在 Run 内。那么程序把所有时间都花在 Run 内，究竟为了做些什么呢？

在初始化之后，几乎程序所做的一切事情，都是在响应各种事件。这些事件通常是关于键盘、鼠标、或手写笔的输入。UIElement 类（顾名思义，是“用户界面元素”的意思）定义了一些和键盘、鼠标、手写笔相关的事件；Window 类继承了所有的事件。其中一个事件名为 MouseDown。只要用户用鼠标点击（click）窗口的客户区，MouseDown 事件就会发生。

下面是一个范例，稍微将 Main 里的语句次序做了改变，同时也安装了一个事件处理器（event handler），专门处理 MouseDown 事件：

HandleAnEvent.cs

```
//---------------------------------------------
// HandleAnEvent.cs (c) 2006 by Charles Petzold
//---------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
```

```
namespace Petzold.HandleAnEvent
{
    class HandleAnEvent
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();

            Window win = new Window();
            win.Title = "Handle An Event";
            win.MouseDown += WindowOnMouseDown;

            app.Run(win);
        }
        static void WindowOnMouseDown(object sender, MouseButtonEventArgs args)
        {
            Window win = sender as Window;
            string strMessage =
                string.Format("Window clicked with {0} button at point ({1})",
                              args.ChangedButton, args.GetPosition(win));
            MessageBox.Show(strMessage, win.Title);
        }
    }
}
```

我习惯的事件处理器命名方式是：用负责响应事件的类或对象名作为开头（比如 Window），后面接着“On”，最后接事件名(所以这里我将事件处理器命名为 WindowOnMouseDown)。但是，这只是我个人的习惯，你可以有自己的命名方式。

MouseDown 事件所需要的事件处理器，必须符合 MouseButtonEventHandler 委托，也就是说，第一个参数的类型是 object，第二参数的类型是 MouseButtonEventArgs。这个类定义在 System.Windows.Input 命名空间中，所以代码中使用 using 编译指令来调用此命名空间。因为“使用此事件处理器的 Main 方法”是 static 的，所以我们必须把此事件处理器也申明为 static。

本书大多数后继的程序，都会利用 using 指令将 Sysgtem.Windows.Input 命名空间包含进来，即使没有用到还是这么做。

当用户在窗口的客户区中按下鼠标，MouseDown 事件就会发生。其事件处理器的第一个参数是“此事件的来源”，也就是 Windows 对象。事件处理器可以轻易地将此对象转换成为 Window 类的对象，然后才加以利用。

在 HandleAnEvent 类中的事件处理器，之所以需要 Window 对象，有两个目的：首先，将 Window 对象作为 GetPosition 方法的参数(GetPosition 方法在 MouseButtonEventArgs 类中)，此方法返回一个 Point 类型（这是定义在 System.Windows 命名空间内的结构体类型）

的对象，表示鼠标坐标（相对于 GetPosition 参数的左上角）。其次，事件处理器读取 Window 对象的 Title property，使用此 property 作为消息框（Message Box）的名称。

MessageBox 类也是定义在 System.Windows 命名空间中的，它具有 12 个静态的 Show 重载方法，让你可以选择显示按钮或是图片。默认情况下，只显示 OK 按钮。

在 HandleAnEvent 程序中，Message Box 显示出鼠标光标的位置，这个位置是相对于客户区的。你可能会很自然地认定这些坐标是以像素(Pixel)为单位，但事实并非如此，它们是和设备无关的，以 1/96 英寸为一个单位。 本章稍后，我会再度解释这个奇特的坐标系统(coordinate system)。

HandleAnEvent 的事件处理器将 sender 参数转型为 Window 对象。对于此事件处理器来说，想得到这个 Windows 对象，做法不只一种。在 Main 中所建立的 Window 对象可以被储存在一个静态字段（static field）中，以便此事件处理器稍后使用。另一种做法，是利用 Application 类的某些 property。Application 具有一个 static property，名为 Current，此 property 会存放程序所创建的 Application 对象（我之前说过，一个程序只能创建一个 Appliation 对象）。Application 也包含一个名为 MainWindow 的 instance property，利用此 property，就可以得到 Window 对象。所以此事件处理器可以设定一个 Window 类型的局部变量，做法如下：

```
Window win = Application.Current.MainWindow;
```

如果获取 Window 对象仅仅是为了取得 Title 标题文字以提供消息框使用，那么 MessageBox.Show 方法可以直接调用 Application.Current.MainWindow.Title，而不用设置局部变量。

Application 类定义了很多有用的事件。在.NET 中的习惯是，大多数的事件都有对应的 protected 方法，可以用来发出事件。Application 所定义的 Startup 事件是利用 protected OnStartup 方法来产生的。一旦调用 Application 对象的 Run 方法，OnStartup 方法就会被立刻调用。当 Run 即将返回时，会调用 OnExit 方法（并发出对应的 Exit 事件）。你可以利用接收到这两个事件的时机来进行整个应用程序的初始化和清理工作。

OnSessionEnding 方法和 SessionEndin 事件表示用户已经选择要注销 Windows 操作系统，或者要关闭电脑。此事件附带一个 SessionEndingCancelEventArgs 类型的参数，此类型继承了 CancelEventArgs 的 Cancel property，你可以将此 property 设为“true”，就可以防止 Windows 操作系统被关闭. 你必须将程序编译成“Windows Application”而非“Console Application”，才有可能会收到此事件。

如果你的程序需要 Application 类的某些事件，你可以为这些事件安装事件处理器，另一个更方便的做法，就是定义一个类继承 Application，例如下一个范例 InheritTheApp 正是如此。

这个类继承 Application，并将“负责发出事件的方法” 予以覆盖 （override）。

InheritTheApp.cs

```
//-----------------------------------------------
// InheritTheApp.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.InheritTheApp
{
   class InheritTheApp : Application
   {
      [STAThread]
      public static void Main()
      {
         InheritTheApp app = new InheritTheApp();
         app.Run();
      }
      protected override void OnStartup(StartupEventArgs args)
      {
         base.OnStartup(args);

         Window win = new Window();
         win.Title = "Inherit the App";
         win.Show();
      }
      protected override void OnSessionEnding(SessionEndingCancelEventArgs args)
      {
         base.OnSessionEnding(args);

         MessageBoxResult result =
            MessageBox.Show("Do you want to save your data?",
                            MainWindow.Title, MessageBoxButton.YesNoCancel,
                            MessageBoxImage.Question, MessageBoxResult.Yes);

         args.Cancel = (result == MessageBoxResult.Cancel);
      }
   }
}
```

InheritTheApp 类继承自 Application，且覆盖（override）两个 Application 类定义的方法：OnStartup 与 OnSessionEnding。在此程序中，Main 方法并没有创建 Application 类的对象，而是创建了 InheritTheApp 类对象，而且 Main 方法本身也是定义在 InheritTheApp 类中。让 Main 建立一个“自己所属的类”对象，这看起来可能有点奇怪，不过却是完全合法的，因为 Main 是一个静态方法，所以即使在 InheritTheApp 对象尚未创建之前，Main 就已经存在了。

InheritTheApp 同时覆盖（override）两个方法: OnStartup（在 Run 被调用后，这个方法就立即被调用）与 OnsessionEnding。程序可以利用 OnStartup 的机会创建一个 Window 对

象，并把它显示出来。此 InheritTheApp 类还可以在构造函数内进行此工作。

在 override 版的 OnSessionEnding 中，弹出一个“Yes、No、Cancel”对话框。请注意此对话框的标题被设定为 MainWindow.Title。因为 OnSessionEnding 是继承自 Application 的实例方法，所以只要直接使用 MainWindow，就可以得到此 Application 实例的 property。你可以在 MainWindow 的前面加上 this 关键字，来更清楚地表示 MainWindow 是 Application 对象的 property。

当然，此程序在结束前没有文档可以被存储，所以它忽略了 Yes 和 No 的响应，直接让此应用程序关闭，让 Windows 操作系统终结当前用户的进程。如果用户的回应是 Cancel，会造成 SessionEndingCancelEventArgs 对象的 Cancel 标志被设为 true，从而阻止 Windows 操作系统被关闭，或者注销（log off）。通过 SessionEndingCancelEventArgs 的 ReasonSessionEnding property，你可以分辨到底是关闭还是注销。ReasonSessionEnding 的值可以是 ReasonSessionEnding.Logoff 或 ReasonSessionEnding.Shutdown。

不管是 OnStartup 还是 OnSessionEnding，一开始都是先调用基类（base class）的方法。此调用并非绝对必要，但也不会有坏处。一般来说，我们会去调用基类的方法，除非你有特别的理由不这么做。

你可以从命令提示窗口（Command Prompt Window）执行程序，这样就可以指定命令行参数，Windows 程序也不例外。想取得命令行参数（command-line argument），定义 Main 的方法稍微有点不同：

```
public static void Main(string[] args)
```

命令行参数会以字符串数组的形式传入 Main 中。在 OnStartup 方法中，也可以利用 StartupEventArgs 参数的 Args property，取得此字符串数组。

Application 具有 MainWindow property，这表示一个程序可以有多个窗口。一般来说，许多窗口都只是短暂出现的对话框，但对话框其实就是 Window 对象，只是有些小差异（对话框的显示方式，以及和用户的交互方式）。

下面的程序将几个窗口一起放到桌面上显示：

ThrowWindowParty.cs

```
//-----------------------------------------------
// ThrowWindowParty.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
```

```
namespace Petzold.ThrowWindowParty
{
   class ThrowWindowParty: Application
   {
      [STAThread]
      public static void Main()
      {
         ThrowWindowParty app = new ThrowWindowParty();
         app.Run();
      }
      protected override void OnStartup(StartupEventArgs args)
      {
         Window winMain = new Window();
         winMain.Title = "Main Window";
         winMain.MouseDown += WindowOnMouseDown;
         winMain.Show();

         for (int i = 0; i < 2; i++)
         {
            Window win = new Window();
            win.Title = "Extra Window No. " + (i + 1);
            win.Show();
         }
      }
      void WindowOnMouseDown(object sender, MouseButtonEventArgs args)
      {
         Window win = new Window();
         win.Title = "Modal Dialog Box";
         win.ShowDialog();
      }
   }
}
```

与 InheritTheApp 类一样，ThrowWindowParty 类继承自 Application，且在 override 版本的 OnStartup 方法中，创建一个 Window 对象。然后再创建两个 Window 对象，也将它们显示出来。（我很快就会讨论 MouseDown 事件处理器内做了什么事。）

你会注意到的第一件事，就是 OnStartup 所创建的三个窗口，在此应用程序中具有同等的地位。你可以点击任何窗口，然后该窗口就会出现在最上面。你可以用任何次序关闭这些窗口，且只有在最后一个窗口被关闭后，程序才会结束。如果窗口标题上没有注明“Main Window”，你根本无从识别主窗口。

然而，如果你在程序里查看Application对象的MainWindow property， 会发现第一个调用Show的窗口会被当作此程序的主窗口[3]（至少一开始是这样）。

Application 类也包含了一个名为 Windows（注意字尾有 s）的 property，其类型是 WindowCollection。WindowCollection 是常用的.NET collection 类型，它实现了 ICollection

[3] 质疑：应似是第一个被创建的窗口会被作为主窗口（Application 的MainWindow 属性）。读者可以自己做实验证明这一点。——审校者注

和 IEnumerable 接口，顾名思义，WindowCollection 用来存储多个 Window 对象。此类型包含一个名为 Count 的 property，且具有一个 indexer。只要窗口调用过 Show，且还存在，你可以用此 indexer 轻易地取得特定窗口。OnStartup 覆盖后，Windows.Count property 会变成 3，且 Windows[0]会得到标题为“Main Window”的窗口。

此程序有一个古怪的地方，三个窗口都出现在 Windows taskbar 上（大部分的用户都将 Windows taskbar 放在屏幕下方）。一个程序在 Windows taskbar 中占用不止一项，大家通常认为这不太理想。想要让这些多余的窗口不要占用 taskbar 的空间，你必须在 for 循环里加入下面的语句：

```
win.ShowInTaskbar = false;
```

但另一件古怪的事接踵而至：如果你先关闭标题为“Main Window”的窗口，会看到 taskbar 中的对应项消失了，但是还有两个窗口在桌面，程序依然在执行！

一般来说，在 Run 方法返回之后，程序就会结束，而且在用户关闭最后一个窗口后，Run 方法就会返回。这样的行为受到 Application 的 ShutdownMode property 控制，其值为 ShutdownMode 枚举值，默认为 ShutdownMode.OnLastWindowClose，你还可以将其设定为 ShutdownMode.OnMainWindowClose。在调用 Run 方法之前，可先执行下面的语句：

```
app.ShutdownMode = ShutdownMode.OnMainWindowClose;
```

或者，你可以试着将下面的语句插入 OnStartup 内的任何位置。（在 Main 中你必须在此 property 的前面冠以此 Application 对象的名称；在 OnStartup 方法中，你可以直接使用此 property，或者在前面加上“this”关键字也可以。）

```
ShutdownMode = ShutdownMode.OnMainWindowClose;
```

现在，当主窗口关闭，Run 方法返回之后，程序就会结束。

不要删除对 Shutdown property 所作的改变，试着将下面的语句加入 for 循环中：

```
MainWindow = win;
```

你应该还记得，MainWindow 是 Application 类的 property。你的程序可以利用这个 property，将你所选择的窗口指定为主窗口（main window）。在 for 循环最后，标题为“Extra Window No.2”的窗口成为主窗口，也就是关闭此窗口就会结束程序。

ShutdownMode 还有第三个选项：你可以将此 property 设定为 ShutdownMode.OnExplicitShutdown，这么一来，只有当程序呼叫 Application 的 Shutdown 方法时，Run 方法才会返回。

现在，把你所加入和 Application 类的 ShutdownMode 与 MainWindow property 相关的语句都删除。还有另一个方法可以在多个窗口之间建立层次（hierarchy）关系，这是通过 Window 类的 Owner property 来实现的。默认情况下，此 property 的值是 null，表示该窗口没有主人（owner）。你可以设定 Owner property 为此程序中其他的 Window 对象。（但是要注意一点：层层往上追查 owner，不可以最后追查回自己。也就是说，“拥有”的关系不可以形成回路。）比方说，试着在 for 循环中，插入以下代码:

```
win.Owner = winMain;
```

现在，主窗口拥有这两个子窗口，这三个窗口之间还可以在屏幕上相互切换，但是不管怎么切换，你会发现“被拥有的”窗口一定会出现在“拥有者”窗口的前面。当你将“拥有者”窗口最小化时，“被拥有的”窗口也会从屏幕上消失不见，而当你将“拥有者”窗口关闭时，“被拥有的”窗口也会被自动关闭。这两个子窗口实际上变成了 modeless 类型的对话框。

对话框可以大致分为两大类：modeless 对话框是其中比较不常见的那一种，另一种 modal 对话框则比较常见。只要你用鼠标在 ThrowWindowParty 客户区域点一下，就可以看见 modal 对话框的实际范例。WindowOnMouseDown 方法会建立另一个 Window 对象，并设定好其 Title property，但是并不是调用 Show，而是调用 ShowDialog 来将它显示出来。ShowDialog 和 Show 不一样，ShowDialog 不会立刻返回，且利用此方法显示的 modal 对话框，不会让你切换到同一个程序的其他窗口（但是可以切换到别的程序窗口）。只有在你关闭此对话框之后，ShowDialog 的调用才会返回。

另一方面，modeless 对话框允许你适当地同时使用主窗口和对话框。Visual Studio 的 Quick Find 对话框就是 modeless 对话框，用来找寻源代码内的字符串，即使 Quick Find 对话框尚未关闭，依然可以继续在主窗口中编辑源代码。Modal 对话框则会将用户输入的一切事件都捕捉起来，只有在此对话框关闭之后，才能处理程序中的其他窗口。Modeless 对话框则没有这样的限制。

试着这么做：回到第一个范例程序 SayHello，将源代码中的 Show 改成 ShowDialog，并且将所有引用到此 Application 对象的地方，都先注释掉。这个程序依然可以执行，因为 ShowDialog 实现了自己的消息循环，以处理输入事件。正因为 modal 对话框没有参与应用程序的消息循环（而是有自己的消息循环），且 modal 对话框不让应用程序得到输入消息，所以 modal 对话框具有模式（modal）的作用。

前面的两个程序都定义有一个 Application 的子类。一般来说，程序经常定义自己的 Window 子类。下面的程序包含三个类和三个源码文件。要想在 Visual Studio 2005 的现有工程中，多加入一个空的源代码文件，做法是在“Solution Explorer”的工程名上按鼠标右键，然后从菜单中选取“Add New Item”。或者从“Project”菜单中选择“Add New Item”。不管是哪一种做法，你要加入的项目都是“Code File”，这一开始就是空白的。

下面的工程名称是 InheritAppAndWindow， 这也是一个类的名称，此类只包含 Main：

InheritAppAndWindow.cs

```
//-----------------------------------------------------
// InheritAppAndWindow.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.InheritAppAndWindow
{
   class InheritAppAndWindow
   {

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

Main 创建一个类型为 MyApplication 的对象，且调用此对象的 Run 方法。MyApplication 类继承自 Application，它的定义如下：

MyApplication.cs

```
//-----------------------------------------------
// MyApplication.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.InheritAppAndWindow
{
   class MyApplication : Application
   {
      protected override void OnStartup(StartupEventArgs args)
      {
         base.OnStartup(args);

         MyWindow win = new MyWindow();
         win.Show();
      }
   }
}
```

在 OnStartup 方法的 override 版本中，此类创建一个类型为 MyWindow 的对象，这是该工程中的第三个类，它继承自 Window：

MyWindow.cs

```
//---------------------------------------
// MyWindow.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.InheritAppAndWindow
{
    public class MyWindow : Window
    {
        public MyWindow()
        {
            Title = "Inherit App & Window";
        }
        protected override void OnMouseDown(MouseButtonEventArgs args)
        {
            base.OnMouseDown(args);

            string strMessage =
                string.Format("Window clicked with {0} button at point ({1})",
                              args.ChangedButton, args.GetPosition(this));
            MessageBox.Show(strMessage, Title);
        }
    }
}
```

继承自 Window 的类，通常使用 Window 类的构造函数来初始化窗口本身，唯一需要自己做的初始化工作，就是设定 Title property。请注意，此 property 前面不需要加上任何对象名称，因为 MyWindow 已经从 Window 类继承了这个 property。前面可以加上关键字 this，也可以不加：

```
this.Title = "Inherit App & Window";
```

不给 MouseDown 事件安装事件处理器，而是改将 OnMouseDown 方法覆盖（override）。因为 OnMouseDown 是一个实例方法，所以 this 代表的是 Window 对象，可以将 this 关键字当作参数传进 GetPosition 方法中，且可以直接存取 Title property。

虽然刚刚所显示的程序没有什么不对的地方，但其实在只有代码（而没有 XAML markup）的 WPF 程序中，更常见（且更容易）的做法就是定义一个 Window 的子类，而非定义 Application 的子类。下面是一个很典型的单一代码文件程序：

InheritTheWin.cs

```
//-----------------------------------------------
// InheritTheWin.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.InheritTheWin
{
    class InheritTheWin : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new InheritTheWin());
        }
        public InheritTheWin()
        {
            Title = "Inherit the Win";
        }
    }
}
```

本书的 Part I 中，我的许多范例程序都是使用上面的结构。这样的代码比较短，而且如果你真的想要将 Main 方法尽可能精简，还可以将所有的调用都放在同一条语句中：

```
new Application().Run(new InheritTheWin());
```

让我们开始把玩这个程序，我会建议如何改造这个程序，你可以跟着做，或者你愿意照自己的想法做，那样更好。

此窗口摆放的位置和尺寸，是由 Windows 操作系统所决定的，但是你也可以改变它。Window 类型从 FrameworkElement 继承了 Width 和 Height property，你可以在构造函数中设定这些 property：

```
Width = 288;
Height = 192;
```

设定这两个 property 时，不限定只能用整型，也可以用双精度浮点型，所以下面的设定方式也是可行的：

```
Width = 100 * Math.PI;
Height = 100 * Math.E;
```

Width 和 Height property 一开始都是没有定义的，如果你在程序中没有设定这两个 property，那么它们就会一直没有定义，也就是说，它们的值是 NaN。NaN 是 IEEE 浮点数的缩写，它的意思是“非数字”（Not a Number）。

因此，如果你需要取得窗口的实际尺寸，不要使用 Width 和 Height property，而是要改用 ActualWidth 和 ActualHeight 这两个只读 property。然而，在创建窗口的过程中，ActualWidth 和 ActualHeight 可能为 0；只有在窗口已经出现在屏幕上时，这两个 property 才会生效。

你可能会以为，我之前选择设定宽和高的两个数字，应该是随便输入的：

```
Width = 288;
Height = 192;
```

其实，这两个数字不是像素（pixel）。如果 Width 和 Height property 是以像素为单位，就不可能被指定为 double 型浮点数值。在 WPF 中，关于长宽和位置，你所制定的单位有时候被称为“设备无关像素”(device-independent pixel) 或称“逻辑像素”（logical pixel），但是或许最好连提都不要提到“像素”这两个字，免得引起误解，所以我称其为“与设备无关的单位”（device-independent unit）。每个单位是 1/96 英寸，所以 288 和 192 实际上是用来指示此窗口的宽为 3 英寸，高为 2 英寸。

如果你真的拿尺子测量你的屏幕，或许会发现实际显示的尺寸是有误差的。像素和英寸之间的关系，是由 Windows 操作系统所建立，用户也可以主动改变。在 Windows 桌面点鼠标右键，从菜单中选取“Properties”，选择“Settings”页，然后按下“Advanced”（高级）按钮，再按下“General”(一般)页，就可以看到设定值。

Windows 操作系统默认的显示分辨率（display resolution）是每英寸 96 像素，如果你的电脑也是这样设定的，Width 和 Height 的值分别为 288 和 192，就会精确地对应到 288 和 192 个像素。

然而，如果你让显示分辨率率变为 120DPI，那么 WPF 程序如果设定 Width 和 Height property 为 288，192，就等同于 360 像素和 240 像素，也是 3 英寸和 2 英寸。

科技持续进步，以后的屏幕分辨率会变得越来越高，即使分辨率改变了，WPF 程序还是可以不用改变，执行起来没有误差。比方说，假设你有一个屏幕，每英寸大概有 200 像素。为了避免屏幕上一切都变得很小，用户需要把“Display Properties”设定成对应的分辨率（或许是 192DPI）。当 WPF 程序设定 Width 和 Height 为 288 和 192 个单位时，其实就是等同于 596 像素和 384 像素，依然是 3 英寸和 2 英寸。

在 WPF 中，普遍地使用这些与设备无关的单位。比方说，本章稍早的某个程序使用对话框来显示鼠标相对于客户区左上角的位置，也不是以像素为单位，而是以“设备无关单位”为单位，这里是以 1/96 英寸为单位。

如果你做个实验，把 Width 和 Height 设定成非常小，你将会发现窗口一定至少显示出标题栏的部分。你可以利用 SystemParameters.MinimumWindowWidth 和 SystemParameters.Minimum WindowHeight 这两个只读 property，来得知窗口的最小长宽是多少（单位与设备无关）。SystemParameters 类有很多这样的静态 property。

如果想要将窗口放在屏幕上的特定位置，你可以利用 Window 类的 Left 和 Top property：

```
Left = 500;
Top = 250;
```

这两个 property 用来指定窗口左上角的位置（相对于屏幕左上角），也是使用设备无关单位（类型为 double），如果没有设定这两个 property，它们的值也会是 NaN。Window 类并不具有 Right 和 Bottom property。想知道窗口的右下角位置，可以利用 Left 与 Top 以及窗口的长宽推算出来。

假设你的显卡和屏幕可以支持 1600 * 1200 像素，而且你在“Display Properties”对话框中的分辨率也是如此设定。如果你查看静态的 SystemParameters.PrimaryScreenWidth 和 SystemParameters.PrimaryScreenHeight，这两个值也会是 1600 和 1200 吗？当且仅当你的屏幕 DPI 为 96 时。这表示你的屏幕宽为 16-2/3 英寸，高为 12-1/2 英寸。

然而，如果你设定的 DPI 为 120，SystemParameters.PrimaryScreenWidth 和 SystemParameters.PrimaryScreenHeight 会是 1280 和 960（与设备无关）单位，这意味着你的屏幕宽高分别为 13-1/3 英寸和 10 英寸。

因为 SystemParmaeters 一律使用设备无关的单位，只有 SmallIconWidth 和 SmallIconHeight 这两个 property 是例外，它们使用像素做单位。因为 SystemParameters 的 property 普遍不受分辨率的影响，所以你可以安全地使用大部分的值，不需要转换。比方说，下面的代码把窗口放在屏幕的右下角：

```
Left = SystemParameters.PrimaryScreenWidth - Width;
Top = SystemParameters.PrimaryScreenHeight - Height;
```

这段代码运行之前，必须设定好 Width 和 Height property。否则这段代码运行的结果，你可能不大喜欢。如果你的屏幕底部有 taskbar，你的窗口会被遮盖住一部分。你可能想要将

窗口放在工作区（work area）的右下角（而非屏幕的右下角）。工作区就是桌面 toolbar 以外的区域。（最常见的桌面 toolbar 就是 Windows 操作系统的 taskbar）。

SystemParameters.WorkArea property 返回 Rect 类型的对象，此 Rect 结构定义一个矩形（内容包括了左上角位置与长度高度）。此 WorkArea property 必须是 Rect 类型才完整，不可只有宽和高，因为用户可能会将 taskbar 放在屏幕左边。如果真的在左边，Rect 结构的 Left property 就是非零值，而 Width property 就等于屏幕宽度减去 Left 值。

下面的代码，可以将窗口放在工作区的右下角：

```
Left = SystemParameters.WorkArea.Width - Width;
Top = Systemparameters.WorkArea.Height - Height;
```

下面的代码，可以将窗口放在工作区的中间：

```
Left = (SystemParameters.WorkArea.Width - Width) / 2 +
               SystemParameters.WorkArea.Left;
Top = (SystemParameters.WorkArea.Height - Height) / 2 +
               SystemParameters.WorkArea.Top;
```

如果不这么做，另一个做法就是使用 Window 类的 WindowStartupLocation property。这个 property 的值是属于 WindowStartupLocation 枚举类型的，默认为 WindowStartup-Location.Manual，这表示如果程序没有设定窗口位置，就由 Windows 操作系统负责摆放窗口。你只要将此 property 设定为 WindowStartupLocation.CenterScreen，就可以将窗口放在中间。尽管名称为 CenterScreen，但其实是放在工作区的中间，而非整个屏幕的中间。（另外还有一个值为 WindowStartupLocation.CenterOwner，这是利用 modal 对话框用的值，将对话框放在“拥有者”的中央。）

下面的一个小程序是将窗口放在工作区的中间，每次按下“向上键”或者“向下键”，就会将窗口的尺寸增减百分之十：

GrowAndShrink.cs

```
//--------------------------------------------
// GrowAndShrink.cs (c) 2006 by Charles Petzold
//--------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.GrowAndShrink
{
    public class GrowAndShrink : Window
    {
        [STAThread]
```

```
        public static void Main()
        {
            Application app = new Application();
            app.Run(new GrowAndShrink());
        }
        public GrowAndShrink()
        {
            Title = "Grow & Shrink";
            WindowStartupLocation = WindowStartupLocation.CenterScreen;
            Width = 192;
            Height = 192;
        }
        protected override void OnKeyDown(KeyEventArgs args)
        {
            base.OnKeyDown(args);

            if (args.Key == Key.Up)
            {
                Left -= 0.05 * Width;
                Top -= 0.05 * Height;
                Width *= 1.1;
                Height *= 1.1;
            }
            else if (args.Key == Key.Down)
            {
                Left += 0.05 * (Width /= 1.1);
                Top += 0.05 * (Height /= 1.1);
            }
        }
    }
}
```

任意键被按下，就会造成 OnKeyDown 方法（以及和它相关的 KeyDown 事件）被执行。每次你按下并放开键盘上的某个键，OnKeyDown 和 OnKeyUp 方法就会被调用。你可以覆盖（override）这些方法，进行按键的事件处理。利用 KeyEventArgs 对象的 Key property（其值是 Key 枚举型），就可以知道牵涉其中的按键是哪一个。因为 Left、top、Width 与 Height property 都是浮点型，所以增减窗口尺寸时不会丢失精度。你会达到 Windows 操作系统所允许的最大和最小值的限制，但是这些 property 的值依然不会受到影响（依然是计算的结果值）。

想要获取光标移动键（例如：上、下）和功能键（例如：F1、F2）的按键事件时，OnKeyDown 和 OnKeyUp 方法相当有用。但是如果你想要从键盘获得实际的 Unicode 字符，你应该重载的方法是 OnTextInput。TextCompositionEventArgs 参数的 Text property，是一个 Unicode 字符串。一般来说，这个字符串里只有一个字符，但是语音和手写输入法也可能会调用 OnTextInput，这种情况下，字符串有可能会稍微长一些。

下面的程序没有设定 Title property，而是让用户自行输入：

TypeYourTitle.cs

```
//---------------------------------------------
// TypeYourTitle.cs (c) 2006 by Charles Petzold
//---------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.TypeYourTitle
{
    public class TypeYourTitle : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new TypeYourTitle());
        }
        protected override void OnTextInput(TextCompositionEventArgs args)
        {
            base.OnTextInput(args);
            if (args.Text == "\b" && Title.Length > 0)
                Title = Title.Substring(0, Title.Length - 1);
            else if (args.Text.Length > 0 && !Char.IsControl(args.Text[0]))
                Title += args.Text;
        }
    }
}
```

此方法允许的唯一控制字符是倒退键（backspace，“\b”），只有当此 Title 至少有一个字符长时，倒退键才会生效。除了倒退键之外，此方法就只是将从键盘输入的文字加入到 Title 属性的后面。

Window 类定义了其他的 property，可以影响窗口的外观和行为。你可以将 WindowStyle property 设定为 WindowStyle 枚举类型值。默认的值是 WindowStyle.SingleBorderWindow。WindowStyle.ThreeDBorderWindow 比较炫，但是会使得客户区的面积缩小一点点。通常将对话框设定为 WindowStyle.ToolWindow。这样标题栏比较短，而且只有开关钮，没有缩小放大钮。然而，你依然可以最小和最大化窗口，只要按“Atl + Space”就可以调出系统菜单。你也可以调整 tool 窗口的大小。WindowStyle.None 也具有可调整大小的边框，但没有标题栏。你依然可以利用“Atl + Space”来调用系统菜单。没有标题栏，当然无法在窗口上显示 Title property，但是 Title property 在 taskbar 中还是有用的。

一个窗口有没有“缩放边框”（sizing border），是受到 ResizeMode property 的影响，此 property 的值是 RezieMode 枚举类型，缺省是 ResizeMode.CanResize，这表示使用者可以调整窗口大小、最小化（minimize）、或最大化（maxmize）。只要设定成 ResizeMode.CanResizeWithGrip，在客户区的右下角就会显示一个小把手（grip）ResizeMode.CanMinimize 会将缩放边框消除掉，且让最大化按钮被禁用（disable），但是窗口依然可以被最小化。这个选项可以让窗口的大小固定。最后，ResizeMode.NoResize 去掉了最小化和最大化的按钮，也去掉了缩放边框。.

WindowState property 的类型为 WindowState 枚举类型，用来控制你的窗口一开始如何显示。可能的值有 WindowState.Normal、WindowState.Minimized 和 WindowState.Maximized。

将 Topmost property 设定为 true，表示此窗口在其他窗口的上面（使用这个 property 的时候，要特别注意，只有对真的有必要的窗口才做这样的设定。当然，也要让用户可以关闭这样的设定）。

Window 类还有一个重要的 property，就是 Background。这是 Window 从 Control 继承过来的，负责掌管客户区的颜色。不过对于 Background property 来说“颜色”不具有足够的表现力。所以，Backgroup property 是一个 Brush（画刷）类型的对象，“彩绘窗口的画刷”可以有很多变化，包括渐变（gradient）画刷与位图（bitmap）画刷。画刷对于 WPF 相当重要，所以本书特别用了两章的篇幅来介绍。这两章的第一章，就是下一章。现在就让我们开始讨论画刷吧！

Basic Brushes

第 2 章　基本画刷

标准窗口的内部，被称为客户区（client area）。正是在窗口的这一区域，显示文字、图形、控件，并在此接收用户的输入。

之前范例所创建的窗口，其客户区可能被填上白色，这是因为白色是客户区默认的颜色。你可以使用微软 windows 操作系统的控制面板（Control Panel）来设定你自己的系统颜色，以彰显你的美学品味或独特风格。更严肃点的原因，有人可能觉得黑底白字会看得更清楚。如果你是这样的人，你大概会希望更多开发者注意到你的需要，且重视你对颜色的自主选择权。

WPF 的颜色被封装成 `Color` 结构（structure），定义在 `System.Window.Media` 命名空间中。和一般的图形环境一样，Color 结构使用红、绿、蓝三原色来表达颜色。这三种原色（primary）分别被简称为 R、G、B，用这三原色定义出来的三维空间，就是 RGB 颜色空间（color space）。

`Color` 结构包含名为 R、G、B 的三个可读写 property，它们的类型都是 byte。这三种 property 的值范围从 0 到 255。当这三个 property 皆为 0 时，就是黑色；当这三个 property 皆为 255 时，就是白色。

除了这三原色之外，`Color` 结构还包括一个“alpha channel”，其 property 名称为 `A`。所谓“alpha channel”是用来控制颜色的“不透明”（opacity）程度，值越小表示越透明（transparent），0 表示完全透明，而 255 表示不透明。

和所有的结构体一样，`Color` 具有一个无参数（parameterless）的构造函数，但是此构造函数产生的颜色，其 `A`、`R`、`G`、`B` property 都是设为 0，这样的颜色不但是黑色，也是完全透明的。想要让它变成看得见的颜色，你的程序可以手动地设定这四个 property，如同下面的范例所示：

```
Color cir = new Color();
clr.A = 255;
clr.R = 255;
clr.G = 0;
clr.B = 255;
```

最后得到的颜色是不透明的洋红色（magenta）。

此 Color 结构也包含了几个静态的方法，让你可以用一行代码创建 Color 对象。此方法需要三个 byte 类型的参数：

```
Color clr = Color.FromRgb(r, g, b)
```

最后得到的颜色，其中 A 的值是 255。你也可以自己指定 alpha 值：

```
Color clr = Color.FromArgb(a, r, g, b)
```

由各一个 byte 的红绿蓝所组成的 RGB 颜色空间，也被称为 sRGB 颜色空间，这里的“s”是 standard（标准）的意思。sRGB 颜色空间将显示点阵图像的惯用做法予以正规化，从扫描仪到电脑显示器，再到数码相机都适用。当用来在屏幕上显示颜色时，sRGB 原色的值一般会和“从显卡送到显示器”的电流电压信号成正比。

然而在其他的设备上，颜色相当不适合用 sRGB 来表现。比方说，如果一个特定的打印机有能力比一般电脑屏幕更绿，这样的绿怎么能用屏幕绿色最大值的 255 来表达呢？

正是从这方面考虑，有一些其他的 RGB 颜色空间也被定义出来。WPF 的 Color 结构也支持这些另类颜色空间的一种，那就是 scRGB 颜色空间，这种颜色空间以前常被称为 sRGB64，因为原色是用 64 位的值来表达的。在 Color 结构中，scRGB 原色其实是被储存成单精度(single-precision)的浮点数。想要容纳 scRGB 颜色空间，Color 结构包含四个主要的 property，类型都是 float，分别为 ScA、ScR、ScG、ScB。这些 property 和 A、R、G、B property 会相互影响，改变 G property 也会造成 ScG property 的改变，反之亦然。

当 G property 为 0，ScG property 也会为 0；当 G property 为 255，ScG property 就会为 1。在这个范围之内，关系并非是线性的，如表 2-1 所示。

表 2-1

scG	G
<= 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

ScR 与 R 之间的关系，ScB 与 B 之间的关系，以及 ScG 与 G 之间的关系，也都是一样的。ScG 的值可以小于 0 或者大于 1，以容纳超出显示器和 sRGB 数字范围的颜色。

现在常用的阴极射线管(Cathode Ray Tube)并没有以线性的方式显示光线。光线的强度(intensity, I)和送到显示器的电压(voltage, V)，两者之间是幂次关系，如下面的式子所示：

$$I=V^{\gamma}$$

这里的伽玛（gamma）的值，受到显示器和环境光的影响。但是以一般常用的显示器和视觉状况来说，gamma 的值通常介于 2.2 和 2.5 之间（sRGB 标准假定这个值为 2.2）。

对于光线的强度，人类视觉感知能力也是非线性的，感知能力大致和光强度的 1/3 次方成正比。幸好，人类感知的非线性和 CRT 的非线性彼此相互抵消一部分，所以 sRGB 原色（这和显示器的电压成正比），在感知上差不多可以算是线性。也就是说，RGB 的值如果为 80-80-80（十六进制表示法），大致对应到人类认为的“中度灰阶”。这就是为什么 sRGB 会如此广泛的作为标准使用的原因之一。

scRGB 原色却是故意设计成“和光强度成线性关系”的，所以 scG 和 G 的关系如下：

$$scG \cong \left(\frac{G}{255}\right)^{2.2}$$

其中，指数 2.2 是 sRGB 标准所假定的 gamma 值。请注意，上面的式子是逼近式，不是等式。当值不高时，会比较不精确。至于 alpha channel 的关系，则比较简单：

$$scA = \frac{A}{255}$$

你可以利用下面的静态方法，以 scRGB 原色来创建 Color 对象：

```
Color clr = Color.FromScRgb(a, r, g, b);
```

这些参数是浮点数，可以小于 0 或者大于 1。

System.Windows.Media，也包含了一个名为 Colors（注意是复数）的类，此类型包含了 141 个静态的只读 property，这些 property 的名称都是好记的颜色引用方式，从 AliceBlue 和 AntiqueWhite 到 Yellow 和 YellowGreen。比方说:

```
Color clr = Colors.PapayaWhip;
```

这些颜色的名称和 Web 浏览器常用的颜色名称是一样的，但 Transparent property 是个例外，此 property 返回一个 alpha 值为 0 的 Color 对象。Colors 类中其他的 140 个 property

所返回的 Color 对象，其 alpha 值都是 255。

程序可以设定 Background（背景）property，来改变客户区的背景颜色。这个 property 是 Window 从 Control 继承来的。然而，你不可以把 Background 设定成一个 Color 对象，而应将 Background 设定为更多元化的 Brush（画刷）对象。

画刷在 WPF 中使用得相当广泛，所以我们必须早一点注意到它。Brush 本身是一个抽象类，如下面的类层次图所示：

Object

DispatcherObject (abstract)

DependencyObject

Freezable (abstract)

Animatable (abstract)

Brush (abstract)

GradientBrush (abstract)

LinearGradientBrush

RadialGradientBrush

SolidColorBrush

TileBrush (abstract)

DrawingBrush

ImageBrush

VisualBrush

只有 Brush 的子类实例才能用来设定 Window 对象的 Background property。所有和 Brush 相关的类都放在 System.Window.Media 命名空间。本章稍后将讨论 SolidColorBrush 类（单色画刷）和两个继承自 GradientBrush 的类（渐变画刷）。

顾名思义，SolidColorBrush 正是最简单的画刷，只使用单一颜色。在第 1 章后面的程序中，你可以改变客户区的颜色，做法是用 using 编译指示符包含 System.Windows.Media，然后在此 Window 类的构造函数中，加入下面的代码片段：

```
Color clr = Color.FromRgb(0, 255, 255);
SolidColorBrush brush = new SolidColorBrush(clr);
Background = brush;
```

它造成背景变成青色（cyan）。当然，这三行代码可以缩写成一行：

```
Background = new So1idCo1orBrush(Co1or. FromRgb(0, 255, 255));
```

SolidColorBrush 也具有一个无参数的构造函数，和一个名为 Color 的 property。让你在创建画刷对象之后可以设定或者改变画刷的颜色。比方说：

```
SolidColorBrush brush = new SolidColorBrush();
brush.Color = Color.FromRgb(128, 0, 128);
```

下面的程序执行时，会依据“鼠标指针靠近窗口中心的程度”，来改变客户区的背景颜色。此程序利用 using 指示符将 System.Window.Media 命名空间加进来，本书后面大部分的程序也都会用到这个命名空间。

VaryTheBackground.cs

```
//-----------------------------------------------
// VaryTheBackground.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.VaryTheBackground
{
    public class VaryTheBackground : Window
    {
        SolidColorBrush brush = new SolidColorBrush(Colors.Black);

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new VaryTheBackground());
        }
        public VaryTheBackground()
        {
            Title = "Vary the Background";
            Width = 384;
            Height = 384;
            Background = brush;
        }
        protected override void OnMouseMove(MouseEventArgs args)
        {
            double width = ActualWidth
                - 2 * SystemParameters.ResizeFrameVerticalBorderWidth;
            double height = ActualHeight
                - 2 * SystemParameters.ResizeFrameHorizontalBorderHeight
                - SystemParameters.CaptionHeight;

            Point ptMouse = args.GetPosition(this);
            Point ptCenter = new Point(width / 2, height / 2);
```

```
            Vector vectMouse = ptMouse - ptCenter;
            double angle = Math.Atan2(vectMouse.Y, vectMouse.X);
            Vector vectEllipse = new Vector(width / 2 * Math.Cos(angle),
                                            height / 2 * Math.Sin(angle));
            Byte byLevel = (byte) (255 * (1 - Math.Min(1, vectMouse.Length /
                                                       vectEllipse.Length)));
            Color clr = brush.Color;
            clr.R = clr.G = clr.B = byLevel;
            brush.Color = clr;
        }
    }
}
```

当你向客户区移动鼠标时，背景变成较亮的灰色。当鼠标超出“客户区为边界的假想椭圆”时，背景会变成黑色。

这一切都是发生在 override 版本的 OnMouseMove 方法中，而且只要鼠标在程序的客户区内移动，OnMouseMove 方法就会被持续调用。因为一系列问题，导致这个方法有些复杂。首先，这个方法要计算客户区尺寸，但除非客户区里有实际的东西，否则我们无法得知客户区的尺寸。除了自己算，没有别的好的办法来解决这个问题。这个方法一开始使用窗口的 ActualWidth 和 ActualHeight property，然后减去边框和标题栏的尺寸（利用 SystemParameters 类的静态 property，可以取得这些尺寸）。

这里调用 MouseEventArgs 的 GetPosition 方法，来取得鼠标的位置，然后将这个 Point 对象存在 ptMouse 中，这个位置和客户区的中央点之间是有一点距离的。中央点也是一个 Point 对象，名为 ptCenter。然后将 ptMouse 减 ptCenter。如果你查看一下 Point 结构的文档，你将会发现两个 Point 相减会得到一个 Vector 对象，我们这里将此 Vector 对象命名为 vectMouse。就数学上来说，向量（vector）包括“量”（magnitude）和“方向”（direction）。vectMouse 的量就是 ptCenter 和 ptMouse 之间的距离。你可以从 Vector 结构的 Length property 来取得这个“量”；你可以利用 Vector 对象的 X 和 Y property，得知“方向”，代表从原点（0，0）指向（X，Y）的方向。在这个例子中，vectMouse.X 等于是 ptMouse.X 减去 ptCenter.X，Y 也是类似的式子。

Vector 对象的方向也可以表示为角度。Vector 结构包含名为 AngleBetween 的静态方法，用来计算两个 Vector 对象的夹角。VaryTheBackground 程序的 OnMouseMove 方法根据 vectMouse 的 Y 和 X property 比值，以反正切（inverse tangent）函数计算出角度。此角度是从水平轴开始顺时针方向转动，所得到的夹角，以弧度为单位（一周的弧度是 2PI）。接着使用刚刚得到的角度，计算另一个代表“从客户端中心点到客户区内切椭圆的点”之间距离的 Vecotr 对象。灰度和“这两个向量的比率”成正比。

OnMouseMove 方法从 SolidColorBrush 对象的 Color property 获得一个 Color 对象，然后设定好适当的灰度之后，再将 Color 对象设定回 SolidColorBrush 的 Color property，让画刷具有新的值。

当看到这个程序运行时，你可能会吃一惊。显然，每次只要 brush 一有改变，客户区就会被重绘，但这一切都是幕后进行的。这里之所以会有动态的反应，是因为 Brush 继承自 Freezable 类，而 Freezable 类实现了一个名为 Changed 的事件（event），Brush 对象只要一有改变，这个事件就会被触发。所以只要一改变 brush，背景就会被重画。

WPF 底层大量使用 Changed 事件和类似机制，以实现动画（animation）和其他特性。

正如 Colors 类提供了 141 个静态只读的 property 一样，Brushes（也是复数）类也提供了 141 个静态只读的 property，名称和 Color 的 property 都一样，但是 Brushes 的 property 返回的是 SolidColorBrush 对象。你可以用下面的方式设定 Background：

```
Background = new SolidColorBrush(Colors.PaleGoldenrod);
```

也可以改用下面的方式：

```
Background = Brushes.PaleGoldenrod;
```

虽然这两行代码都可以把窗口背景填上特定的颜色，但是这两种做法其实有一点儿差异，像 VaryTheBackground 这样的程序就可以感受到此差异。下面的语句中，将字段（field）定义：

```
SolidColorBrush brush = new SolidColorBrush(Colors.Black);
```

改成如下语句：

```
SolidColorBrush brush = Brushes.Black;
```

重新编译并执行，现在会弹出一个“Invalid Operation Exception”（无效操作异常）消息框，详细信息是“无法设定‘#FF000000’对象的 property，因为此对象为只读状态”。问题发生在 OnMouseMove 方法的最后一条语句，这里试图设定画刷的 Color property。（异常消息的单引号中出现的十六进制数，是目前 Color property 的值。）

利用 Brushes 所取得的 SolidColorBrush 对象是处于冻结（frozen）状态，也就是说，不能再被改变。就像 Changed 事件一样，Freezable 实现了冻结，而 Brush 的冻结正是从这里继承而来的。如果 Freezable 对象的 CanFreeze property 是 true，可以调用 Freeze 方法来实现对象的冻结和不可变动。IsFrozen property 如果变成 true，就表示（对象）已经被冻结。将对象冻结，可以提高效率，因为被冻结的对象不会被改变，所以不需要监控。冻结的 Freezable 对象还可以在不同的线程之间共享，没有被冻结的 Freezable 对象则不行。虽然

无法将冻结的对象解冻（unfreeze），但是你可以做出一个没冻结的复制版本。下面的代码可以定义 VaryTheBackground 中的 brush 字段（field）：

```
SolidColorBrush brush = Brushes.Black.Clone() ;
```

如果你想看到这 141 个画刷出现在同一个窗口的客户区，FlipThroughTheBrushes 程序可以达成你的愿望，你可以用上下箭头来改变画刷。

FlipThroughTheBrushes.cs

```
//-----------------------------------------------------
// FlipThroughTheBrushes.cs (c) 2006 by Charles Petzold
//-----------------------------------------------------
using System;
using System.Reflection;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.FlipThroughTheBrushes
{
    public class FlipThroughTheBrushes : Window
    {
        int index = 0;
        PropertyInfo[] props;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new FlipThroughTheBrushes());
        }
        public FlipThroughTheBrushes()
        {
            props = typeof(Brushes).GetProperties(BindingFlags.Public |
                                                  BindingFlags.Static);
            SetTitleAndBackground();
        }
        protected override void OnKeyDown(KeyEventArgs args)
        {
            if (args.Key == Key.Down || args.Key == Key.Up)
            {
                index += args.Key == Key.Up ? 1 : props.Length - 1;
                                          index %= props.Length;
                SetTitleAndBackground();
            }
            base.OnKeyDown(args);
        }
        void SetTitleAndBackground()
        {
            Title = "Flip Through the Brushes - " + props[index].Name;
            Background = (Brush) props[index].GetValue(null, null);
        }
    }
}
```

此程序使用 reflection(反射)来取得 Brushes 类的成员。构造函数第一行使用“typeof(Brushes)”来得到 Type 类的对象。Type 类定义了一个方法，名为 GetProperties，并返回 PropertyInfo 对象的数组，数组内的每个元素都对应到 Brushes 类里的一个 property。调用 GetProperties 时，以 BinddingFlags 为参数，程序就可以清楚地把自己限制在 Brushes 公开和静态的 property 上。因为 Brushes 的 property 本来就全部都是 public 和 static，这个例子不需要这样的限制，但是这么做也不会有什么不好的影响。

在构造函数和 override 版本的 OnKeyDown 方法中，程序都调用了 SetTitleAndBackground，以便将 Title property 和 Backgound property 设定为 Brushes 类的某个成员。“props[0].Name”会返回一个字符串，这是类第一个 property 的名称，也就是“AliceBlue”。“props[0].GetValue(null, null)”返回实际的 SolidColorBrush 对象。这里的 GetValue 方法需要两个 null 参数：通常第一个参数是 property 所在的对象，因为 Brushes 是一个静态 property，所以没有对应的对象，因此传入 null；第二个参数只有在 property 是 indexer 时才有必要。

System.Windows 命名空间具有 SystemColors 类，其作用类似于 Colors 和 Brushes，只具有静态的只读 property，返回 Color 值和 SolidColorBrush 对象。这些设定存储在 Windows 注册表（registry）中。利用此类，可以得知目前用户的颜色喜好（preferences）。比方说，SystemColors.WindowColor 用来表示用户对于客户区的颜色喜好，而 SystemColors.WindowTextColor 是用户对于客户区文字的颜色喜好。SystonColors.WindowBrush 和 SystemColors.WindowTextBrush 则是返回对应颜色的 SolidColorBrush 对象。对于大多数的真实应用程序来说，应该使用这些颜色，可以达到统一、协调的视觉效果。

从 SystemColors 返回的画刷对象都是冻结的。你的程序可以改变下面的画刷：

```
Brush brush = new SystemColorBrush(SystemColors.WindowColor) ;
```

但是不能改变下面的画刷：

```
Brush brush = SystemColors.WindowBrush;
```

只有继承自 Freezable 类的对象，才可以被冻结。至于 Color 对象，则没有冻结不冻结的问题，因为 Color 是个结构体（structure）。

如果不用单色画刷，可以改用渐变（gradient）画刷，将两种（或多种）颜色混合，逐渐改变。一般说来，渐变画刷是高级的程序主题，但是对于 WPF 来说，创建渐变画刷是非常容易的，且渐变画刷在现代的色彩设计中也很受欢迎。

渐变画刷最简单的形式是 LinearGradientBrush，只需要两个 Color（我们不妨称这两种颜色为 clr1 和 clr2）对象，和两个 Point（pt1 和 pt2）对象。pt1 的位置的颜色是 clr1，且 pt2 的位置的颜色是 clr2。在 pt1 和 pt2 之间的连线上，则是混合了 crl1 和 clr2 的颜色，连线中心点是 clr1 和 clr2 的平均值。垂直于连线的位置，和连线上的点使用相同的颜色。至于超过 pt1 和 pt2 的两边会是什么颜色，稍后再讨论。

有一个好消息：通常你指定这两个点时，使用的单位是像素（pixel）或者（在 WPF 中）是“设备无关单位”，如果你又想要窗口背景采用渐变，那么，只要窗口尺寸发生改变，你就必须重新指定这两个点。

而 WPF 渐变画刷有一个特性，让你不用基于窗口尺寸而调整画刷的点。默认情况下，你指定的点是“相对于表面”的，这里的表面被视为一个单位宽，一个单位高。表面的左上角是（0，0），右下角是（1，1）。

比方说，如果你想要让客户区的左上角为红色，右下角为蓝色，中间则是渐变色，你可以使用下面的构造函数，这里需要指定两种颜色和两个点：

```
LinearGradientBrush brush = new LinearGradientBrush(Colors.Red, Colors.Blue,
                                new Point(0, 0), new Point(1, 1));
```

下面是完整的程序：

GradiateTheBrush.cs

```
//-----------------------------------------------
// GradiateTheBrush.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.GradiateTheBrush
{
    public class GradiateTheBrush : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new GradiateTheBrush());
        }
        public GradiateTheBrush()
        {
            Title = "Gradiate the Brush";

            LinearGradientBrush brush =
            new LinearGradientBrush(Colors.Red, Colors.Blue,
                new Point(0, 0), new Point(1, 1));
            Background = brush;
        }
    }
}
```

当你改变客户区的尺寸时，渐变画刷会随之改变。这要归功于 Freezable 所实现的 Changed 事件。

虽然使用相对坐标系统来设定点很方便，但这不是唯一的做法。GradientBrush 类有一个 MappingMode property，类型为 BrushMappingMode 枚举。此枚举只有两种值，分别为 RelativeToBoundingBox（使用相对坐标是默认值）和 Absolute（使用“设备无关单位”）。

在 GradiateTheBrush 中，用 16 进制数表示 RGB 颜色值，客户区左上角为 FF-00-00，且右下角是 00-00-FF。你可能认为这个中间的颜色如果不是 7F-00-7F 就是 80-00-80，至于是何者，完全由四舍五入的结果而定。确实如此，因为默认的 ColorInterpolationMode property 值是 ColorInterpolationMode.SRgbLinearInterpolation（线性插值）。此 property 的值也可以是 ColorInterpolationMode.ScRgbLinearInterpolation，这表示中间的颜色 scRGB 值 0.5-0-0.5，等于 sRGB 的 BC-00-BC。

如果你需要建立水平或者垂直的渐变，还可以使用 LinearGradientBrush 的构造函数：

```
new LinearGradientBrush(clr1, clr2, angle);
```

指定角度（以度为单位，也就是一周为 360 度。）0 度是水平渐变，clr1 在左边，等同于：

```
new LinearGradientBrush(clr1, c1r2, new Point(0, 0), new Point(1, 0));
```

90 度是垂直渐变，clr1 在上面，等同于：

```
new LinearGradientBrush(clr1, c1r2, new Point(0, 0), new Point(0, 1));
```

其他的角度用起来可能需要一点技巧，就一般的例子来说，第一个点一定是原点，第二个点的计算方式如下：

```
new Point(cos(angle), sin(angle))
```

以 45 度为例，第二个点逼近（0.707, 0.707）。别忘了这是“相对于”客户区的点，所以，如果客户区不是正方形（正方形的概率其实比较低），这两个点之间的连线不会是真正的 45 度。另外，窗口右下角有一大块超出这点之外，这里又会如何？默认状况下，这里会着上第二种颜色。这受 SpreadMethod property 的控制。此 property 类型是 GradientSpreadMethod 枚举，默认是 Pad，表示超出的部分就延续之前的颜色，不用变化；除了 Pad，还有 Reflect 和 Repeat。你可以试着把 GradiateTheBrush 程序修改成下面这样：

```
LinearGradientBrush brush =
   new LinearGradientBrush(Colors.Red, Colors.Blue,
new Point(0, 0), new Point(0.25, 0.25));
brush.SpreadMethod = GradientSpreadMethod.Reflect;
```

在（0，0）和（0.25，0.25）之间，画刷从红到蓝渐变；然后在（0.25，0.25）和（0.5，0.5）之间，从蓝到红渐变；接着在（0.5，0.5）到（0.75，0.75）之间，从红到蓝渐变；最后在（0.75，0.75）和（1，1）之间，从蓝到红渐变。

如果扩大水平和垂直的差异，让窗口变得相当窄，或者相当扁，那么等色线(uniformly colored lines)就会变得几乎垂直或者几乎水平。你或许反而会希望“这两个对角之间的渐变”是以“另两个对角的连线”为等色线。上面这句话有点儿难懂，用图来说明应该可以帮助理解！图 2-1 是被拉长的 GradiateTheBrush 的客户区。

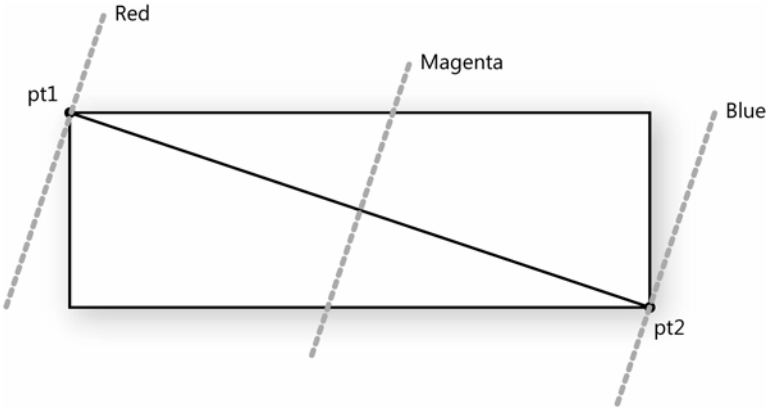

图2-1

虚线代表的是等色线（该线上都是同一个颜色），这一定垂直于 pt1 和 pt2 的连线。你可能比较希望渐变的方式如图 2-2。

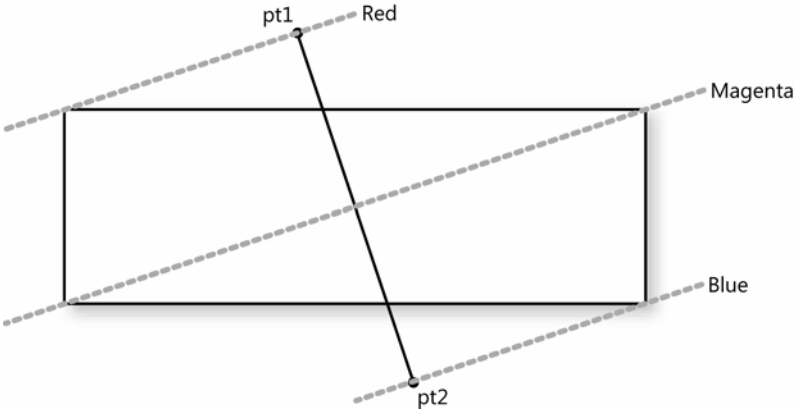

图2-2

现在，洋红色（magenta）的等色线是在对角线上。这么做会有一点麻烦，我们必须计算出 pt1 和 pt2，好让连接 pt1 和 pt2 的线垂直于左下到右上的对角线。

矩形中心点到 pt1（或 pt2）的距离（我称它为 L），可以这样计算：

$$L = \frac{W \cdot H}{\sqrt{W^2 + H^2}}$$

其中 W 是窗口的宽，H 是高。为了要让你更清楚，图 2-3 多了一些线段和文字标识。

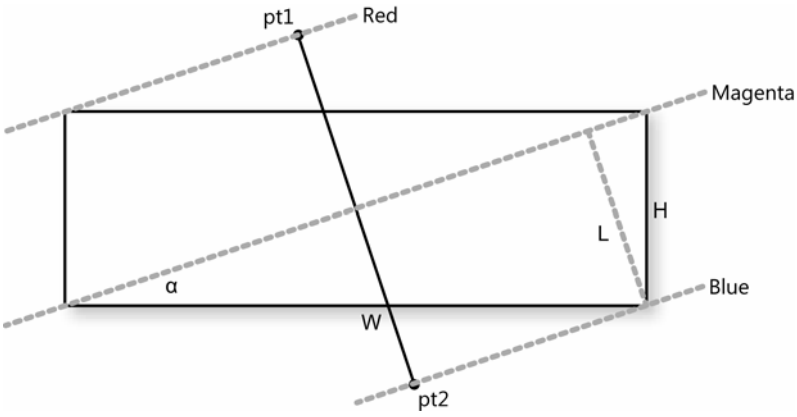

图2-3

请注意，标识为 L 的线段平行于“pt1 和 pt2 的连线”。角度 α 的正弦值可以用两种方式计算出来，第一种方式，用 H 除以对角线长度：

$$\sin(\alpha) = \frac{H}{\sqrt{W^2 + H^2}}$$

另一种方式，如果 L 是对边（opposite side），W 是直角三角形的斜边（hypotenuse）：

$$\sin(\alpha) = \frac{L}{W}$$

这两个式子，解方程组，就可以求出 L。

下面的程序在构造函数中建立一个“可以被修改的”LinearGradientBrush 对象，其 MappingMode 是 Absolute。构造函数也安装了 SizeChanged 事件的处理器，只要窗口尺寸改变，就会跟着发生 SizeChanged 事件。

AdjustTheGradient.cs

```
//------------------------------------------------
// AdjustTheGradient.cs (c) 2006 by Charles Petzold
//------------------------------------------------
using System;
```

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

namespace Petzold.AdjustTheGradient
{
    class AdjustTheGradient: Window
    {
        LinearGradientBrush brush;

        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new AdjustTheGradient());
        }
        public AdjustTheGradient()
        {
            Title = "Adjust the Gradient";
            SizeChanged += WindowOnSizeChanged;

            brush = new LinearGradientBrush(Colors.Red, Colors.Blue, 0);
            brush.MappingMode = BrushMappingMode.Absolute;
            Background = brush;
        }
        void WindowOnSizeChanged(object sender, SizeChangedEventArgs args)
        {
            double width = ActualWidth
                - 2 * SystemParameters.ResizeFrameVerticalBorderWidth;
            double height = ActualHeight
                - 2 * SystemParameters.ResizeFrameHorizontalBorderHeight
                - SystemParameters.CaptionHeight;

            Point ptCenter = new Point(width / 2, height / 2);
            Vector vectDiag = new Vector(width, -height);
            Vector vectPerp = new Vector(vectDiag.Y, -vectDiag.X);

            vectPerp.Normalize();
            vectPerp *= width * height / vectDiag.Length;

            brush.StartPoint = ptCenter + vectPerp;
            brush.EndPoint = ptCenter - vectPerp;
        }
    }
}
```

事件处理器一开始是计算客户区的宽度和高度，如同本章稍早的 VaryTheBackground 程序做法一样。用 Vector 对象 vectDiag 来表示对角线的向量（从左下到右上）。也可以利用右上角坐标减左下角坐标，来计算得到：

```
vectDiag = new point(width, 0) - new Point(0, height);
```

vectPerp 向量垂直于对角线。建立相互垂直的向量很容易，只要把 X 和 Y property 的值对调，并把其中一个数的正负号反向就可以了。Normalize 方法将 X 和 Y 的值除以 Length property 值，使 vectPerp 向量的 Length 变成 1。事件处理器再将 vectPerp 乘以长度 I（就是我们前面计算出的 L）。

最后的步骤是设定 StartPoint 和 EndPoint property。这些 property 一般是通过画刷的构造函数来设定的，而且除去继承的 property，它们是 LinearGradientBrush 仅有的两个 property。（当然，LinearGradientBrush 也从抽象的 GradientBrush 类继承一些 property）。

再次提醒你，程序只需要改变 LinearGradientBrush，窗口就会自动地将结果反映出来。这是 Freezable 类（以及 WPF 的其他类）的 Changed 事件奇妙的地方。

LinearGradientBrush 其实变化多端，远比这两个例子所能展示的更多。渐变的颜色不限定两个，可以有更多。想运用这样的特性，需要使用到 GradientBrush 定义的 GradientStops property。

GradientStops property 是 GradientStopCollection 类型，这是 GradientStop 对象的集合（Collection）。GradientStop 具有 Color 和 Offset property，而包含这两个 property 的构造函数为：

```
new GradientStop(clr, offset)
```

Offset property 的值正常是在 0 和 1 之间，其意义是 StartPoint 和 EndPoint 的相对距离。比方说，如果 StartPoint 是（70，50）且 EndPoint 是（150，90），Offset property 的值为 0.25，就表示此点的位置是在 StartPoint 往 EndPoint 的方向四分之一的地方，也就是（90，60）。当然，如果你的 StartPoint 是（0，0），且你的 EndPoint 是（0，1）或（1，0）或（1，1），那么 Offset 对应的点在哪里，会比较容易推算。

下面的程序创建一个水平 LinearGradientBrush，且针对彩虹的 7 种颜色设定数个 GradientStop 对象。它们依次从左到右，每个 GradientStop 是 1/6 个窗口宽。

FollowTheRainbow.cs

```
//-----------------------------------------------
// FollowTheRainbow.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.FollowTheRainbow
{
    class FollowTheRainbow: Window
    {
        [STAThread]
```

```
        public static void Main()
        {
            Application app = new Application();
            app.Run(new FollowTheRainbow());
        }
        public FollowTheRainbow()
        {
            Title = "Follow the Rainbow";

            LinearGradientBrush brush = new LinearGradientBrush();
            brush.StartPoint = new Point(0, 0);
            brush.EndPoint = new Point(1, 0);
            Background = brush;

            // 采用 Rey G.Biv 的彩虹记住系统
            brush.GradientStops.Add(new GradientStop(Colors.Red, 0));
            brush.GradientStops.Add(new GradientStop(Colors.Orange, .17));
            brush.GradientStops.Add(new GradientStop(Colors.Yellow, .33));
            brush.GradientStops.Add(new GradientStop(Colors.Green, .5));
            brush.GradientStops.Add(new GradientStop(Colors.Blue, .67));
            brush.GradientStops.Add(new GradientStop(Colors.Indigo, .84));
            brush.GradientStops.Add(new GradientStop(Colors.Violet, 1));
        }
    }
}
```

从这里，我们从 LinearGradientBrush 变到 RadialGradientBrush，要做的改变不多，只需要改变画刷的类别名称，且删除指定 StartPoint 和 EndPoint 的语句即可：

CircleTheRainbow.cs

```
//-------------------------------------------------
// CircleTheRainbow.cs (c) 2006 by Charles Petzold
//-------------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.CircleTheRainbow
{
    public class CircleTheRainbow : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new CircleTheRainbow());
        }
        public CircleTheRainbow()
        {
```

```
            Title = "Circle the Rainbow";

            RadialGradientBrush brush = new RadialGradientBrush();
            Background = brush;

            //  采用 Rey G.Biv 的彩虹记住系统
            brush.GradientStops.Add(new GradientStop(Colors.Red, 0));
            brush.GradientStops.Add(new GradientStop(Colors.Orange, .17));
            brush.GradientStops.Add(new GradientStop(Colors.Yellow, .33));
            brush.GradientStops.Add(new GradientStop(Colors.Green, .5));
            brush.GradientStops.Add(new GradientStop(Colors.Blue, .67));
            brush.GradientStops.Add(new GradientStop(Colors.Indigo, .84));
            brush.GradientStops.Add(new GradientStop(Colors.Violet, 1));
        }
    }
}
```

现在画刷从客户区的中心点以红色开始，然后遍历这些颜色，一直到紫色（Violet）定义客户区的“内切椭圆”。在椭圆以外的区域（客户区的四个角落），继续使用紫色，因为 SpreadMethod 的默认值是 Fill。

很明显，RadialGradientBrush 的许多 property 都已经具备实用的默认值。其中三个 property 用来定义一个椭圆：Center 是 Point 类型，定义为（0.5, 0.5），也就是画刷涵盖区域的中心点。RadiusX 以及 RadiusY 的 property 皆为 double 类型，分别代表椭圆的水平和垂直轴半径，默认值是 0.5，所以，无论是横向还是纵向，椭圆都达到了当前画刷作用区域的边界。

椭圆的圆周受到 Center、RadiusX、RadiusY property 的影响，圆周的颜色，正是 Offset property 值为 1 时的颜色。（在 CircleTheRainbow 范例中，此颜色是 Violet 紫色。）

还有个名为 GradientOrigin 的 property，和 Center 一样，是 Point 对象，默认值是（0.5，0.5）。顾名思义，GradientOrigin 是渐变开始的原点。在这个点，你会看到 Offset 为 0 时的颜色。（在 CircleTheRainBow 范例中，此颜色是红色。）

在 GradientOrigin 和椭圆圆周之间，就是发生渐变的地方。如果 GradientOrigin 等于 Center（默认是如此），那么渐变会从椭圆圆心到椭圆圆周之间扩散。如果 GradientOrigin 和 Center 之间有一段距离，那么在“GradientOrigin”和“最接近的椭圆圆周”之间，渐变的颜色变化会比较剧烈，相反方向的颜色变化就比较缓和。想看看这样的影响，请在 CircleTheRainBow 中，插入下面的代码：

```
brush.GradientOrigin = new Point(0.75, 0.75);
```

你可能想要自己体验一下 Center 和 GradientOrigin property 的变化所造成的视觉效果，那么 ClickTheGradientCenter 程序来做实验。此程序使用 RadialGradientBrush 带两个参数的构造函数，定义 GradientOrigin 和椭圆圆周的颜色，然而，设定 RadiusX 和 RadiuxY 值

为 0.1，且 SpreadMethod 为 Repeat，所以画刷显示的是一系列的同心渐变圆圈。

ClickTheGradientCenter.cs

```
//---------------------------------------------------------
// ClickTheGradientCenter.cs (c) 2006 by Charles Petzold
//---------------------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.ClickTheGradientCenter
{
   class ClickTheRadientCenter : Window
   {
      RadialGradientBrush brush;

      [STAThread]
      public static void Main()
      {
         Application app = new Application();
         app.Run(new ClickTheRadientCenter());
      }
      public ClickTheRadientCenter()
      {
         Title = "Click the Gradient Center";
         brush = new RadialGradientBrush(Colors.White, Colors.Red);
         brush.RadiusX = brush.RadiusY = 0.10;
         brush.SpreadMethod = GradientSpreadMethod.Repeat;
         Background = brush;
      }
      protected override void OnMouseDown(MouseButtonEventArgs args)
      {
         double width = ActualWidth
            - 2 * SystemParameters.ResizeFrameVerticalBorderWidth;
         double height = ActualHeight
            - 2 * SystemParameters.ResizeFrameHorizontalBorderHeight
            - SystemParameters.CaptionHeight;

         Point ptMouse = args.GetPosition(this);
         ptMouse.X /= width;
         ptMouse.Y /= height;
         if (args.ChangedButton == MouseButton.Left)
         {
            brush.Center = ptMouse;
            brush.GradientOrigin = ptMouse;
         }
         else if (args.ChangedButton == MouseButton.Right)
            brush.GradientOrigin = ptMouse;
      }
   }
}
```

此程序覆盖 (override)了 OnMouseDown 方法，所以点击客户区，也会有反应。鼠标左键将 Center 和 GradientOrigin 设定成相同的值，你会看到整个画刷从客户区中央移动；鼠标右键只会改变 GradientOrigin。让 GradientOrigin 尽量接近 Center 点，至少停留在圆的内部。现在你可以看到这个渐变如何在一边被挤压，在另一边却比较宽松。

这个效果相当有趣，所以我决定把它做成动画。下面的 RotateTheGradientOrigin 程序没有使用 WPF 的任何动画功能，只用计时器（timer）改变 GradientOrigin property。

在.NET 中，至少有 4 个计时器类型，其中 3 个都叫做 Timer。System.Threading 和 System.Timers 内的 Timer 类，在我们这个例子中并不能被使用，因为这些 timer 事件发生在不同的线程中，而 Freezable 对象只能被（创建它的）同一个线程所改变，而不能被其他线程改变。System.Windows.Forms 内的 Timer 类型封装了标准的 Windows 操作系统计时器，但是如果使用它的话，还需要在工程中加入对 System.Windows.Forms.dll 组件（assembly）的引用。

如果你需要让事件发生在 WPF 程序的主线程中，那么 System.Windows.Threading 命名空间的 DispatcherTimer 类是最适合的。你可以利用 TimeSpan 来设定 Interval property，但是这有精度限制，最高频率是每 10 millisecond 发出一次到时消息。

下面的程序创建一个 4 英寸的正方形窗口，窗口不大，以免占用太多系统时间。

RotateTheGradientOrigin.cs

```
//-------------------------------------------------------
// RotateTheGradientOrigin.cs (c) 2006 by Charles Petzold
//-------------------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Threading;

namespace Petzold.RotateTheGradientOrigin
{
    public class RotateTheGradientOrigin : Window
    {
        RadialGradientBrush brush;
        double angle;

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

```
        public RotateTheGradientOrigin()
        {
            Title = "Rotate the Gradient Origin";
            WindowStartupLocation = WindowStartupLocation.CenterScreen;
            Width = 384; //  相当于 4 英寸
            Height = 384;

            brush = new RadialGradientBrush(Colors.White, Colors.Blue);
            brush.Center = brush.GradientOrigin = new Point(0.5, 0.5);
            brush.RadiusX = brush.RadiusY = 0.10;
            brush.SpreadMethod = GradientSpreadMethod.Repeat;
            Background = brush;

            DispatcherTimer tmr = new DispatcherTimer();
            tmr.Interval = TimeSpan.FromMilliseconds(100);
            tmr.Tick += TimerOnTick;
            tmr.Start();
        }
        void TimerOnTick(object sender, EventArgs args)
        {
            Point pt = new Point(0.5 + 0.05 * Math.Cos(angle),
                                 0.5 + 0.05 * Math.Sin(angle));
            brush.GradientOrigin = pt;
            angle += Math.PI / 6; // ie, 30 degrees
        }
    }
}
```

在本章，我将焦点放在 Window 的 Background property，但是 Window 还有另外三个 property 也是 Brush 类型的，其中一个是 OpacityMask，这个 property 是从 UIElement 继承而来的，在第 31 章讨论到位图（bitmap）时，会对此详细讨论。

Window 的另外两个 Brush property 都是从 Control 继承而来的。一个是 BorderBrush，可以在客户区的周边绘制一个边框。把下面的代码插入到程序中，看看执行结果会如何：

```
BorderBrush = Brushes.SaddleBrown;
BorderThickness = new Thickness(25, 50, 75, 100);
```

Thickness 结构有四个 property，名为 Left、Top、Right、Bottom，也具有一个四参数的构造函数，用来设定这些 property。这里用和设备无关的单位来指示客户区四边的边界宽度。如果你想要让四边具有相同的边框，可以使用单一参数版本的构造函数：

```
BorderThickness = new Thickness(50) ;
```

当然，你也可以在边框使用渐变画刷。

```
BorderBrush = new GradientBrush(Colors.Red, Colors.Blue,
                                new Point(0, 0), new Point(1, 1));
```

这段程序看起来很像是在客户区使用渐变画刷，红色在左上角，蓝色在右下角，不过却不是在客户区，而是在客户区的周边。BorderBrush 会让客户区的面积变小。如果你使用 BorderBrush 且设定 Background property 为渐变画刷，就很容易看出来。即使 BorderBrush 和 Background 都具有相同的渐变画刷，这两个画刷也不会彼此交融。

```
Background = new GradientBrush(Co1ors.Red, Colors.Blue,
                               new Point(0, 0), new point(l, 1));
```

此背景画刷会在“不包含边框”的客户区中，完成全部的渐变。

Window 类中，另外一个类型为 Brush 的 property 是 Foreground，为了要让此 property 可以发挥效用，我们需要在窗口上放置一些内容。内容的形式有很多种，可以是文字、图、控件(control)等，这正是下一章要开始讨论的主题。

The Concept of Content

第 3 章　内容的概念

Window 类有超过 100 个公开的（public）property，其中有一些（像是对应到标题的 Title property）相当重要。但是，迄今为止，Window 最重要的 property 是 Content。你想要在窗口的客户区显示什么东西，将它设定给 Content property 即可。

你可以将 Content 设定成一个字符串，也可以将它设定成位图（bitmap），还可以将它设定成一张矢量图（drawing），设定成按钮、滚动条（scrollbar），或者是 WPF 所支持的 50 多个控件。你几乎可以将 Content property 设定成任何东西，但是有一个小问题：

你只能设定“一个”东西给 Content property。

刚开始使用 Content 时，这个限制会让你觉得沮丧。当然，后来你会学会如何让 Content property 的对象当宿主（host），以容纳多个其他的对象。至于现在，光是单一内容对象就够我们研究好一阵子。

Window 类的 Content property 是从 ContentControl 类继承来的。CotentControl 继承自 Control，而 Window 直接继承自 ContentControl。ContentControl 类存在的目的，几乎就只是为了要定义 Content property 以及几个相关的 property 和方法。

Content property 被定义为 Object 类型，这似乎暗示着它可以是任何对象，事实也相去不远。我说“相去不远”，是因为你不可以把 Content property 设定成另一个 Window 类型的对象。这么做会使运行时期出现异常，异常信息提醒开发者：Window 必须是“树根”，而不可以是另一个 Window 对象的分支。

你可以将 Content property 设为字符串，如下面的例子：

DisplaySomeText.cs

```
//-----------------------------------------------
// DisplaySomeText.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.DisplaySomeText
{
   public class DisplaySomeText : Window
   {
      [STAThread]
```

```
        public static void Main()
        {
            Application app = new Application();
            app.Run(new DisplaySomeText());
        }
        public DisplaySomeText()
        {
            Title = "Display Some Text";
            Content = "Content can be simple text!";
        }
    }
}
```

此程序在客户区左上角显示文字“Content can be simple text！”。如果此窗口太窄而无法容纳所有的文字，你会看到文字尾端被截掉，而不是被自动放到下一行，但是你可以在文字中主动插入换行符，可以是 CR 字符（“\r”）或者是 LF 字符(“\n”)，或者两者一起出现(“\r\n”)。

此程序包含一个 using 指示符来加入 `System.Windows.Media` 命名空间，以方便使用那些可以影响文字颜色和字体的 property。这些 property 都是 `Window` 从 `Control` 类所继承来的。

除了你品味的高低，没什么东西可以阻碍你设定用何种字体来显示客户区文字，作法如下：

```
FontFamily = new FontFamily("Comic Sans MS");
FontSize = 48;
```

这段代码可以放在此类的构造函数内的任何地方。

在 WPF 中，没有所谓的 `Font` 类。这里第一行使用到一个 `FontFamily` 对象。Font family（也称为 type family）是相关字体的 collection。在 Windows 操作系统中，font family 有一些大家相当熟悉的名字，像是 Courier New、Times New Roman、Arial、Palatino Linotype、Verdana 和 Comic Sans MS 等。

Typeface（也称为 face name）是“font family”和“变化”的结合，例如 Time New Roman Bold（粗）、Times New Roman Italic（斜）和 Times New Roman Bold Italic（粗斜）。并非所有的 font family 都支持每一种变化；某些 font family 具有“影响字符宽度”的变化，例如 Arial Narrow（窄）。

Font 这个词一般来说表示特定的 typeface 和尺寸的结合。常用的字型测量法是“em size”（这个名称是有典故的，在早期活字印刷时代，大写的 M 是正方形金属，用 M 的尺寸来作代表。M 的发音正是 em）。通常用 `em size` 来代表拉丁字母（拉丁字母是大写的和小写从 A 到 Z 的字母，不包含那些使用“区别符号”的字符）的字符高度，从高部（ascender）的顶端，到低部（descender）的底端。（译者注：高部指的是比小写 x 高的部分，低部指的是比小写 x 低的部分。）然而，em size 并非公制（metrical）的概念，只是一种设计的概念。在特定的 font

中，字符的实际尺寸会比 em size 的值略大或略小。

一般来说，em size 的指定，是以“点”为单位的。在传统排版中一个点是 0.01384 英寸，以电脑排版来说，点被精确地认定为 1/72 英寸。因此，36 点的 em size（常常缩略的写成“36 点 font”）代表的是“字符的高度约为 1/2 英寸”。

在 WPF 中，利用 FontSize property 指定 em size，但是不用点作单位。和所有的 WPF 度量方式一样，FontSize 使用“与设备无关的单位”，也就是 1/96 英寸。将 FontSize property 设定成 48，会导致 em size 为 1/2 英寸，等同于 36 点。

如果你习惯用点（point）当单位来指定 em size，在指定 FontSize property 时，只要将点的尺寸乘以 4/3（或者除以 0.75）就可以了；如果你还不习惯用点来指定 em size，那么你应该要习惯这么做，只要在指定 FontSize property 时，将点的尺寸乘以 4/3 即可。

默认的 FontSize property 是 11，也就是 8.25 点。纽约时报大量使用 8 点当作印刷字的大小；新闻周刊则是使用 9 点的字。本书的原文使用 10 点的字。

你应该在 FontFamily 的构造函数中使用完整的 type face 名称：

```
FontFamily = new FontFamily("Times New Roman Bold Italic");
FontSize = 32;
```

这是 24 点的 Times New Roman Bold Italic 字体。然而，比较惯用的方式，是在 FontFamily 构造函数中指定 font family 名称，然后在 FontStyle 和 FontWeight property 中指定粗体和斜体：

```
FontFamily = new FontFamily("Times New Roman");
FontSize = 32;
FontStyle = FontStyles.Italic;
FontWeight = FontWeights.Bold;
```

请注意，FontStyle 和 FontWeight property 值是设定成 FontStyles（复数）类和 FontWeights（复数）类的静态只读 property。这些静态 property 都会返回 FontStyle 和 FontWeight 类型的对象，这些对象都是“限定自己使用”的结构体。

下面是一个有趣的小改变：

```
FontStyle = FontStyles.Oblique;
```

斜体（Italic）的 typeface 常常会在文体上和非斜体（或称正体 roman）的 typeface 有差异。看看小写的“a”你就会知道怎么回事了。但是倾斜（oblique）的 typeface 却是直接拿正体字母来作向右倾斜处理。某些 font family 可以让你用 FontStretches 类的静态 property，来设定 FontStretch property。

你已经熟悉了用来将客户区着色的 Background property，而 Foregound property 的着色对象则是文字本身。试试下面的代码：

```
Brush brush = new LinearGradientBrush(Colors.Black, Colors.White,
                            new Point(0, 0), new Point(1, 1));
Background = brush;
Foreground = brush;
```

前景和背景都使用相同的画刷，你可能会担心看不见文字，然而，不会有这样的事发生。在第 2 章我们就看到过，渐变画刷用来进行背景着色时，会默认地根据客户区的尺寸作调整。类似的，前景画刷会自动地根据内容（实际的字符串）的尺寸作调整。改变窗口的尺寸不会影响前景画刷，但是当你让客户区和文字的尺寸完全一样时，这两个画刷会完全重叠，造成文字和背景之间完全融合，看不见文字。

现在试试这个：

```
SizeToContent = SizeToContent.WidthAndHeight;
```

Window 类的 SizeToContent property 造成窗口大小会根据内容的尺寸作调整。如果还是让背景和前景使用相同的 LinearGradientBrush，你会看不见文字。你可以将 SizeToContent property 设定为某个 SizeToContent 枚举值：Manual（默认）、Width、Height 或 WidthAndHeight。后面 3 个会让窗口根据内容的宽和高来作调整。这是一个相当简单的 property，常常用在对话框或者表单（form）中。当设定窗口大小符合内容的尺寸时，你也常常会想要用下面的方式除去“调整尺寸的边框”：

```
ResizeMode = ResizeMode.CanMinimize;
```

或

```
ResizeMode = ResizeMode.NoResize;
```

你可以为第 2 章最后的代码加上客户区内的边框：

```
BorderBrush = Brushes.SaddleBrown;
BorderThickness = new Thickness(25, 50, 75, 100);
```

你会看到前景画刷和 SizeToContent 都会考虑到边框。内容一定会出现在此边框以内。

DisplaySomeText 程序所显示出来的问题，其实比乍看之下更一般化。你知道，所有的对象都具备 ToString 方法，此方法会返回一个“代表对象”的字符串。这里的窗口正是使用 ToString 方法来展示对象。试着把 Content property 设定成别的对象，你就会相信了：

```
Content = Math.PI;
```

或者试试下面的：

```
Content = DateTime.Now;
```

不管是上面哪一个例子，窗口所显示出来的内容都会和 ToString 传出来的字符串一模一样。如果这里的对象没有覆盖（override）ToString 方法，那么默认的 ToString 方法会显示出完整的类型名称（包含命名空间）。例如：

```
Content = EventArgs.Empty;
```

这会显示出“System.EventArgs”字符串。我只发现了一个例外，即如果你用下面的做法：

```
Content = new int[57];
```

窗口会显示出字符串“Int32[ ]Array”，而 ToString 方法返回的却是“System.Int32[]”。

下面的程序设定 Content property 为空字符串，然后将键盘输入的字符加入其中。这个程序类似第 1 章的 TypeYourTitle 程序，不过这个版本还可以让你输入 CR（换行）和 Tab（跳格）。

RecordKeystrokes.cs

```
//-----------------------------------------------
// RecordKeystrokes.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.RecordKeystrokes
{
    public class RecordKeystrokes : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new RecordKeystrokes());
        }
        public RecordKeystrokes()
        {
            Title = "Record Keystrokes";
            Content = "";
        }
        protected override void OnTextInput(TextCompositionEventArgs args)
        {
            base.OnTextInput(args);
            string str = Content as string;

            if (args.Text == "\b")
            {
```

```
            if (str.Length > 0)
                str = str.Substring(0, str.Length - 1);
        }
        else
        {
            str += args.Text;
        }
        Content = str;
    }
  }
}
```

RecordKeystrokes 程序之所以能有这样的效果，是因为每当有按键被按下，Content property 就会被改变，且 Window 类会负责把新的内容绘制到客户区。稍微做一个小改动，可能就会让这个程序无法如我们所愿地正常运行。比方说，定义空字符串字段（field）：

```
string str = "";
```

在构造函数中，将 Content property 设定成这个变量：

```
Content = str;
```

从 OnTextInput 中删除相同的语句，并且也删除（原来的）str 的定义。现在，这个程序就不正常了。看看 OnTextInput 内的一条复合赋值语句：

```
str += args.Text;
```

它等同于下面的语句：

```
str = str + args.Text;
```

这两条语句的问题在于：字符串加法（字符串连接）的结果是一个新的字符串对象，而不是原来的字符串对象。别忘了，字符串是固定不变的（immutable）。字符串连接之后产生新字符串，但是 Content property 依然指向原来的字符串。

现在，试着做一些改变。你需要将 System.Text 命名空间加进来，定义一个 StringBuilder 对象类型的字段：

```
StringBuilder build = new StringBuilder("text");
```

在构造函数中，将 Content 设定为该对象：

```
Content = build;
```

你应该知道窗口会显示出“text”，因为 StringBuilder 对象的 ToString 方法会返回它

内部的字符串，将 OnTextInput 方法内的代码块替换成：

```
if (args.Text == "\b")
{
   if (build.Length > 0)
       build.Remove(build.Length - 1, 1);
}
else
{
   build.Append(args.Text);
}
```

又一次，程序不能正常执行了。虽然此程序只有一个 StringBuilder 对象，但窗口无法知道 StringBuilder 对象内的字符串何时发生了改变，所以无法用新的字符串更新窗口。

我之所以指出这些例子，是因为有时候 WPF 的对象似乎会很神奇地自动更新自己。其实，里面运作的原理一点都不神奇，一定是某事件通知某个方法，画面才会被更新。知道将会采用的运作方式，可以帮助你更了解环境。使用 StringBuilder 对象的改版 RecordKeystrokes 程序根本不能运行，如果在 OnTextInput 方法的底部插入下面的代码：

```
Content = build;
```

窗口够聪明，它知道你指定相同的对象给 Content（多此一举），所以它不会进行更新，但如果你这么做：

```
Content = null;
Content = build;
```

程序就恢复正常了。

我们看到窗口内容可以是纯文字，但是 Content property 存在的目的只是显示简单的无格式字符串吗？不，不是，当然不是！Content property 真正需要的是本质上更图形化的东西，由 UIElement 继承而来的类的实例。

在 WPF 中，UIElement 是一个极为重要的类。它实现键盘、鼠标以及手写笔（stylus）事件的处理。UIElement 类也包含一个很重要的方法，名为 OnRender。OnRender 方法被调用，以显示出对象的外观。（本章结束时会有这样的例子）

在 Content property 的世界中，分成两组对象：一组继承自 UIElement，另一组则不是。后面这一组的显示结果，就是 ToString 方法的返回值；前面这一组，则是利用 OnRender 来显示。继承自 UIElement 的类（以及其对象）被称为 element。

唯一直接继承 UIElement 的类是 FrameworkElement，在 WPF 中所有的 element 都是继承自 FrameworkElement。理论上，UIElement 提供关于用户界面和屏幕显示的必要结构，可以支持各种各样的编程框架（programming framework）。WPF 正是这样的框架，它包含继承自 FrameworkElement 的所有类。实际上，你不太可能会去区分到底哪些 property、方法、事件是由 UIElement 所定义，哪些又是由 FrameworkElement 所定义的。

Image 是继承自 FrameworkElement 的一个很常见的类型。下面是 Image 的继承层次图：

Object

DispatcherObject (abstract)

DependencyObject

Visual (abstract)

UIElement

FrameworkElement

Image

Image（图像）类让你可以轻易地在文件（document）或者应用内包含影像。下面的程序从网站上获得一幅位图，然后在窗口上显示出来：

ShowMyFace.cs

```
//---------------------------------------
// ShowMyFace.cs (c) 2006 by Charles Petzold
//---------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Petzold.ShowMyFace
{
   class ShowMyFace : Window
   {
      [STAThread]
      public static void Main()
      {
         Application app = new Application();
         app.Run(new ShowMyFace());
      }
```

```
        public ShowMyFace()
        {
            Title = "Show My Face";

            Uri uri = new Uri("http://www.charlespetzold.com/PetzoldTattoo.jpg");
            BitmapImage bitmap = new BitmapImage(uri);
            Image img = new Image();
            img.Source = bitmap;
            Content = img;
        }
    }
}
```

想要显示图像，需要几个步骤。此程序先建立一个 Uri 对象，用来表示位图的位置。然后将 Uri 对象传进 BitmapImage 构造函数当参数，构造函数会将此图像载入内存（WPF 支持许多格式，包括 GIF、TIFF、JPEG 与 PNG）。想要利用 Image 类在窗口上显示出图像，你只要将 Image 类的实例（instance）指定给窗口的 Content property 即可。

你可以在 System.Windows.Controls 命名空间找到 Image 类。严格来说，Image 不算是控件（control），它并非继承自 Control 类。但是 System.Windows.Controls 命名空间是如此的重要，所以从这里开始，我几乎把它加到后面所有的程序中。BitmapImage 类位于 System.Windows.Media.Image 命名空间。如果你会用到位图，这就是很重要的命名空间。然而，除非真的需要，否则我不会在程序中把此命名空间加进来。

另外，你也可以从本地硬盘获得图像。这时候需要用文件的绝对路径，作为Uri构造函数的参数，也可以用相对文件名，但前面必须写上“file://”。 下面是替换的代码，用来动态地获得渔夫处理鳟鱼的图像：

```
Uri uri = new uri (
   System.IO.Path.Combine(
      Environment.GetEnvironmentVariable("windir"), "Gone Fishing.bmp")) ;
```

Environment.GetEvironmentVariable 方法取出“windir”环境变量，其值看起来像是“C:\WINDOWS”这样的字符串。Path Combine 方法结合图片文件的路径名称和文件名，这样我（懒惰的程序员）就不用费心正确地插入斜杠了。你可以在 Path 类型名称的前面冠上 System.IO（我就是这么做的）或者利用 using 指示符将此命名空间加进来。

你可以不用将 Uri 对象传给 BitmapImage 构造函数，而是将 Uri 对象设定给 BitmapImage 的 UriSource property。不过，建议你在设定此 property 的前后，要分别调用 BitmapImage 的 BeginInit 和 EndInit 方法：

```
bitmap.BeginInit();
bitmap.UriSource = uri;
bitmap.EndInit();
```

你已经知道，位图具有以像素（pixel）为单位的宽和高，这样的信息被编码在文件中。BitmapImage 类从 BitmapSource 类继承到整数类型，只读的 PixelWidth 和 PixelHeight property。Bitmap 常常（但并非绝对）具有内置的分辨率消息。有时候此分辨率消息非常重要，有时候则不重要。你可以从 DpiX 和 DpiY property 得到以 DPI（dot per inch，一英寸的点数）为单位的分辨率信息，这是只读的，且类型为 double。BitmapImage 类也包含只读的 Width 与 Height。这些值是用下面的式子来计算的：

$$Width = \frac{96 \cdot PixelWidth}{DpiX}$$

$$Height = \frac{96 \cdot PixelWidth}{DpiY}$$

分子如果没有 96，这些式子计算出的宽和高以英寸为单位。乘以 96 就可以将英寸转成设备无关单位。Height 和 Width property 描述的位图尺寸使用和设备无关单位。BitmapImage 也从 BitmapSource 继承一个名为 Format 的 property，让我们可以得知位图的格式；对于那些使用“调色板”的位图，Palette property 让我们可以取得调色板。

不管 ShowMyFace 程序是显示我的脸，还是鳟鱼和渔夫，或者是你选择的任何图像，你会注意到在窗口的边界限制之下，图像会尽量用最大的方式显示，而且不会失真（也就是会维持一样的长宽比）。除非客户区的长宽比（aspect ratio）和图像的长宽比一样，否则你会看到客户区的一部分背景（不是在图像的上下，就是在左右）。

窗口内的图像的尺寸受到 Image 自己的几个 property 的控制，其中一个是 Stretch（拉伸）。默认情况下，Stretch property 等于 Stretch.Uniform，这表示图像会均匀的（uniformly）增大或者缩小（也就是说，水平和垂直方向会一样），以填满客户区。

你也可以将 Stretch property 设定成 Stretch.Fill：

```
img.Stretch = Stretch.Fill;
```

这样的设定方式，会造成图像填满整个窗口，一般来说，图像会失真，因为水平和垂直增加的比例不同。Stretch.UniformToFill 则是将图像均匀地拉伸，但是会把整个客户区都完全覆盖，超出的部分会被截掉。

Stretch.None 选项造成图像以原始尺寸显示，也就是依照 BitmapSource 的 Width 和 Height property。

只要 Stretch property 不是 Stretch.None，你也可以设定 Image 的 StretchDirection property。此 property 默认值是 StretchDirection.Both，这表示图像可以大于或者小于其原始尺寸。

StretchDirection.DownOnly 表示图像不可以比原始尺寸大，StretchDirection.UpOnly 则表示不可以比原始尺寸小。

不管你的图像尺寸如何，图像总是放在窗口中央（除非 Stretch.UniformToFill）。你可以改变这一点——设定 Image 的 HorizontalAlignment 和 VerticalAlignment property（这是从 FrameworkElement 继承来的 property）。比方说，下面的代码将图像移动到客户区的右上角：

```
img.HorizontalAlignment = HorizontalAlignment.Right;
img.VerticalAlignment = VerticalAlignment.Top;
```

HorizontalAlignment 与 VertialAlignment property 在 WPF 的 layout（版面布局）中扮演相当重要的角色。你将会一再遇到这些 property。如果你希望 Image 对象出现在右上角，但是不要紧贴着边界，你可以在 Image 对象的周围设定边界（margin）：

```
img.Margin = new Thickness(10);
```

Margin 是 FrameworkElement 所定义的 property，常常用来在 element 之间插入隔离空间。你可以使用 Thickness 结构体来定义边界，每边都一样（在本例中，10/96 英寸或约 0.1 英寸），或者每边都不一样。你应该记得，Thickness 构造函数有 4 个参数，依次用来设定上、下、左、右：

```
img.Margin = new Thickness(192, 96, 48, 0);
```

现在边界是左边 2 英寸，上面 1 英寸，右边 1/2 英寸，下面没有边界。如果你将窗口调整到很小，图像会比边界先消失，显然边界比较受重视。

Image 对象也具有 Width 和 Height property（从 FrameworkElement 继承来的），它们是可读写的 double 值。如果你查看它们的值，你会发现他们没有定义，值为 NaN（这和 Window 对象的这两个同名 property 一样）。你也可以为 Image 对象设定精确的 Width 和 Height，不过有可能会和某些 Stretch 的设定不一致。

你也可以将窗口的尺寸调整到图像的原始尺寸：

```
SizeToContent = SizeToContent.WidthAndHeight;
```

设定 Window 对象的 Foreground property，对于图像的显示不会有影响。Foreground property 只会对于文字内容有影响，或者对于会显示文字的 element 有影响（稍后有例子）。

正常的情况下，为 Window 设定 Background property，只会影响没有被图像覆盖的区域。但是，试试下面的代码：

```
img.Opacity = 0.5;
Background = new LinearGradientBrush(Colors.Red, Colors.Blue,
                                     new Point (0, 0), new Point (1, 1));
```

现在通过图像，还是会看到画刷。Opacity property（这是 Image 从 UIElement 继承而来的 property）默认是 1，但是你可以将它设定成 0 到 1 之间的任何值，让 element 变得透明。（然而，如果是 Window 对象，这个 property 是无效的。）

第 29 章对于图形变换（transform）有完整的讨论，不过目前你可以看到转动一个位图相当容易：

```
img.LayoutTransform = new RotateTransform(45);
```

Image 类不具有自己的 Background 和 Foreground property，因为这两个 property 是由 Control 类所定义的，而 Image 并非继承自 Control。一开始可能不太容易区别。在早期的 Windows API（application programming interface）中，几乎屏幕上所有的东西，都被认为是控件（control），现在却不是如此。控件是视觉对象（visual object），其特征是对用户的输入有反应。像 Image 这样的 element，当然可以得到用户的输入，因为所有的键盘、鼠标、手写笔的输入事件都是由 UIElement 所定义的。

看一下 System.Windows.Shapes 这个命名空间，它包含了名为 Shape 的抽象类，以及 6 个子类。这些类也继承自 UIElement、FrameworkElement：

Object

DispatcherObject (abstract)

DependencyObject

Visual (abstract)

UIElement

FrameworkElement

Shape (abstract)

Ellipse

Line

Path

Polygon

Polyline

Rectangle

虽然 Image 是显示点阵图像（raster 图像）的标准做法，这些 Shape 类实现了简单的二维（two-dimensional）矢量图（vector graphics）。下面的程序创建一个椭圆（Ellipse）类的对象：

ShapeAnEllipse.cs

```
//---------------------------------------------
// ShapeAnEllipse.cs (c) 2006 by Charles Petzold
//---------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Petzold.ShapeAnEllipse
{
    class ShapeAnEllipse : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new ShapeAnEllipse());
        }
        public ShapeAnEllipse()
        {
            Title = "Shape an Ellipse";

            Ellipse elips = new Ellipse();
            elips.Fill = Brushes.AliceBlue;
            elips.StrokeThickness = 24; // 1/4 inch
            elips.Stroke =
                new LinearGradientBrush(Colors.CadetBlue, Colors.Chocolate,
                                        new Point(1, 0), new Point(0, 1));
            Content = elips;
        }
    }
}
```

该椭圆会填满客户区。周长线是 1/4 英寸粗，且使用渐变画刷。椭圆内部（interior）使用 AliceBlue 画刷来着色。这个颜色以是美国总统罗斯福（Teddy Roosevelt）的女儿来命名。

不管是 Shape 类还是 Ellipse 类，都没有定义任何 property 可以让我们用来设定椭圆的尺寸，但是 Ellipse 类从 FrameworkElement 类继承到 Width 和 Height property，这两个 property 正是作此用途的：

```
elips.Width = 300;
elips.Height = 300;
```

和 Image 一样，你现在可以使用 HorizontalAlignment 和 VerticalAlignment property 来设定椭圆的位置，是在中间，水平对齐于左、右，或者垂直对齐于上、下：

```
elips.HorizontalAlignment = HorizontalAlignment.Left;
elips.VerticalAlignment = VerticalAlignment.Bottom;
```

HorizontalAlignment 与 VerticalAlignment 枚举类型都有名为 Center 的成员，也有名为 Stretch 的成员。对于许多 element 来说，默认使用 Stretch。Ellipse 的默认就是 Stretch，这也就是为什么 Ellipse 一开始会填满整个客户区，Element 会拉伸到容器的边界。事实上，如果你设定 HorizontalAlignment 和 VerticalAlignment 为 Stretch 以外的值，且没有同时设定 Width 和 Height property，此椭圆将会缩小（collapse）成 1/4 英寸的小球，只看得到圆周。

然而，如果你没有设定 Width 和 Height property，你可以设定 MinWidth、MaxWidth、MinHeight 和 MaxHeight property（全都是从 FrameworkElement 继承而来的），其中任何一个，或者全部，以限制椭圆尺寸在特定的范围内。默认状况下，这些 property 都没有被定义。任何时候（除了窗口构造函数内不可以），程序可以利用只读的 ActualWidth 和 ActualHeight property 来得知椭圆的尺寸。

如果你认为我对于“作为窗口内容的 element”的尺寸大小表现得过度关心，那是因为它是一个重要议题。你或许习惯为控件和图形对象指定特定的尺寸，但是 WPF 不需要如此死板，所以你需要好好地了解视觉元件是如何被调整尺寸的。

在 Ellipse 类内，你找不到可以让你将椭圆指定到客户区特定位置的 property，HorizontalAlignment 和 VerticalAlignment 算是最接近这种目的的 property 了。

之前我向你演示过如何将 Window 的 Content property 设定为字符串，以及如何设定此文字的 font。然而，你直接设定到 Content property 的文字，总具有相同的格式，比方说，你无法将其中几个字设定为粗体或斜体。

如果你需要这样，就不要将 Content property 设定成字符串，而是设定成一个 TextBlock 类型的对象：

FormatTheText.cs

```
//-------------------------------------------
// FormatTheText.cs (c) 2006 by Charles Petzold
//-------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Documents;
```

```
namespace Petzold.FormatTheText
{
   class FormatTheText : Window
   {
      [STAThread]
      public static void Main()
      {
         Application app = new Application();
         app.Run(new FormatTheText());
      }
      public FormatTheText()
      {
         Title = "Format the Text";

         TextBlock txt = new TextBlock();
         txt.FontSize = 32; // 24 points
         txt.Inlines.Add("This is some ");
         txt.Inlines.Add(new Italic(new Run("italic")));
         txt.Inlines.Add(" text, and this is some ");
         txt.Inlines.Add(new Bold(new Run("bold")));
         txt.Inlines.Add(" text, and let' s cap it off with some ");
         txt.Inlines.Add(new Bold(new Italic(new Run("bold italic"))));
         txt.Inlines.Add(" text.");
         txt.TextWrapping = TextWrapping.Wrap;

         Content = txt;
      }
   }
}
```

虽然，这是本书第一个主动创建 TextBlock 对象的程序，但其实你以前就见过 TextBlock 了。如果你将 Content property 设定成字符串，那么 ContentControl（这是 Window 的祖先类）会先创建一个 Text Block 类型的对象，以实际显示出此字符串。Text Block 类直接继承自 FrameworkElement，它定义了 Inlines property（类型为 InlineCollection，这是 Inline 对象的 collection）。

TextBlock 本身是属于 System.Windows.Controls 命名空间。但是 Inline 是属于 System.Windows.Documents 命名空间，而且甚至不是继承自 UIElement。下面是部分的类集成图，显示出 Inline 和它的某些后代：

Object

DispatcherObject (abstract)

DependencyObject

ContentElement

FrameworkContentElement

TextElement (abstract)

Inline (abstract)
Run
Span
Bold
Hyperlink
Italic
Underline

你可能注意到这个类层次跟之前的似乎有类似的结构——ContentElement 和 FrameworkContentElement 类对比于 UIElement 和 FrameworkElement 类。然而，ContentElement 类不包含 OnRender 方法。继承自 ContentElement 的类所产生的对象，不会在屏幕上把自己画出来。它们却是要通过“继承自 UIElement 的类” 才能在屏幕上达到视觉的演示，由“继承自 UIElement 的类”提供它们所欠缺的 OnRender 方法。

更明确地说，Bold 和 Italic 对象不会自己绘制自己。在 FormatTheText 程序中，这些 Bold 和 Italic 对象都是通过 TextBlock 对象才能显示出来的。

不要把 ContentElement 类和 ContentControl 混为一谈。ContentControl 是控件，像 Window 一样，可以具有 Content property。就算 Content property 是空的，ContentControl 对象还是会在屏幕上显示自己。而 ContentElement 对象一定要是其他可以显示的对象的一部分（也就是说，必须是控件的内容），才能得以显示。

FormatTheText 程序用来组合 TextBlock 的 Inlines collection。InlineCollection 类实现了 Add 方法，可以加入字符串对象、Inline 对象以及 UIElement 对象（最后一个让你在 TextBlock 中嵌入其他 element）。然而，Bold 和 Italic 构造函数只接受 Inline 对象，而不接受字符串对象，所以程序先为每个 Bold 或 Italic 对象使用 Run 构造函数。

下面的代码设定 TextBlock 的 FontSize property：

```
txt.FontSize = 32;
```

然而，如果改变窗口的 FontSize property，效果也会一样：

```
FontSize = 32;
```

类似的，下面的代码可以设定窗口的 Foreground property，然后 TextBlock 的文字也会变成这个颜色：

```
Foreground = Brushes.CornflowerBlue;
```

屏幕上的 element，呈现树状的组织方式。窗口是 TextBlock 的父亲（parent），而 TextBlock 又是数个 Inline element 的父亲。除非孩子明确地设定某些 property，否则直接从此树状结构中，沿袭父亲的 Foreground property 和所有文字相关的 property。在第 8 章，你会看到实际的作用。

类似 UIElement 类，ContentElement 类定义了许多用户输入事件，可以把事件处理器（event handler）安装到“由 TextBlock 显示出来的”Inline elements 上。下面的程序展示此技巧。

ToggleBoldAndItalic.cs

```
//-----------------------------------------------
// ToggleBoldAndItalic.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;

namespace Petzold.ToggleBoldAndItalic
{
   public class ToggleBoldAndItalic : Window
   {
      [STAThread]
      public static void Main()
      {
         Application app = new Application();
         app.Run(new ToggleBoldAndItalic());
      }
      public ToggleBoldAndItalic()
      {
         Title = "Toggle Bold & Italic";

         TextBlock text = new TextBlock();
         text.FontSize = 32;
         text.HorizontalAlignment = HorizontalAlignment.Center;
         text.VerticalAlignment = VerticalAlignment.Center;
         Content = text;

         string strQuote = "To be, or not to be, that is the question";
         string[] strWords = strQuote.Split();

         foreach (string str in strWords)
         {
            Run run = new Run(str);
            run.MouseDown += RunOnMouseDown;
            text.Inlines.Add(run);
            text.Inlines.Add(" ");
         }
      }
```

```
        void RunOnMouseDown(object sender, MouseButtonEventArgs args)
        {
            Run run = sender as Run;

            if (args.ChangedButton == MouseButton.Left)
                run.FontStyle = run.FontStyle == FontStyles.Italic ?
                    FontStyles.Normal : FontStyles.Italic;

            if (args.ChangedButton == MouseButton.Right)
                run.FontWeight = run.FontWeight == FontWeights.Bold ?
                    FontWeights.Normal : FontWeights.Bold;
        }
    }
}
```

此构造函数将哈姆雷特（Hamlet）中的一句话分割成为多个单词（word），然后根据每个单词创建一个 Run 对象，再将这些单词放在 TextBlock 对象的 Inlines collection 中。在此过程中，程序也会将 RunOnMouseDown 事件处理器连接到每个 Run 对象的 MouseDown 事件中。

Run 类从 TextElement 类继承了 FontStyle 和 FontWeight property，且事件处理器会根据被按下的鼠标按钮类型，来改变这些 property。如果是是鼠标左键，并且 FontStyle property 目前是 FontStyles.Italic，那么事件处理就会将此 property 设定成 FontStyle.Normal；如果此 property 目前是 FontStyle.Normal， 那么事件处理器会将它改变成 FontStyle.Italic。类似地，FontWeight property 也会在 FontWeights.Normal 和 FontWeights.Bold 之间切换。

我在前面曾经提到，窗口的 Content property 其实想要的是继承自 UIElement 类的实例，因为此类定义一个名为 OnRender 的方法，负责在屏幕上显示此对象。本章最后一个程序名为 RenderTheGraphic，需要两个源代码文件：第一个文件是一个自定义（custom）的 element 类，第二个文件将该类的实例设置为窗口的 content。下面的类继承自 FrameworkElement（这是直接继承自 UIElement 的唯一类别），它将相当重要的 OnRender 方法予以 override，以获得一个 DrawingContext 对象。通过此对象和 DrawEllipse 方法就可以绘制椭圆。此类单纯地模仿 System.Windows.Shapes 命名空间的 Ellipse 类。

SimpleEllipse.cs

```
//---------------------------------------------
// SimpleEllipse.cs (c) 2006 by Charles Petzold
//---------------------------------------------
using System;
using System.Windows;
using System.Windows.Media;

namespace Petzold.RenderTheGraphic
{
    class SimpleEllipse : FrameworkElement
    {
```

```
        protected override void OnRender(DrawingContext dc)
        {
            dc.DrawEllipse(Brushes.Blue, new Pen(Brushes.Red, 24),
                new Point(RenderSize.Width / 2, RenderSize.Height / 2),
                RenderSize.Width / 2, RenderSize.Height / 2);
        }
    }
}
```

在 OnRender 还没有被调用之前，根据可能的 Width 和 Height 设定值以及该类和它的容器之间的协商，RenderSize property 可以被确定下来。

如果你之前就有 Windows 编程经验（或者即便你没有什么经验），你可能会认为此方法直接在屏幕上绘制椭圆，其实不是这样的。DrawEllipse 的参数会被保留，以供以后在屏幕上显示椭圆。所谓“以后”，也有可能是马上，但是会将不同来源的图形集中在一起，然后在屏幕上进行组合。WPF 可以变出更多这样子的图形把戏。

下面的程序，创建一个 SimpleEllipse 对象，然后将它设定给 Content property：

RenderTheGraphic.cs

```
//-----------------------------------------------
// RenderTheGraphic.cs (c) 2006 by Charles Petzold
//-----------------------------------------------
using System;
using System.Windows;

namespace Petzold.RenderTheGraphic
{
    class RenderTheGraphic : Window
    {
        [STAThread]
        public static void Main()
        {
            Application app = new Application();
            app.Run(new ReplaceMainWindow[1]());
        }
        public ReplaceMainWindow[2]()
        {
            Title = "Render the Graphic";

            SimpleEllipse elips = new SimpleEllipse();
            Content = elips;
        }
    }
}
```

[1] 质疑：似应是RenderTheGraphic。——审校者注
[2] 质疑：似应是RenderTheGraphic。——审校者注

椭圆会填满客户区。当然，你也会想要实验一下 SimpleEllipse 的 Width 与 Height property，以及 HorizontalAlignment 与 VerticalAlignment property，做不同设定时的影响会如何。

虽然本章使用到了 System.Windows.Controls 命名空间的 element，但是没有使用任何继承自 Control 的类（当然 Window 类本身是个例外）。控件是设计来取得用户的输入，并作出反应的。在下一章，你会明白这是怎么一回事。