Metro Revealed_Building Window 8 apps with XAML and C#

中文翻译合集

版本 1.0

翻译时间：2012-09-03

DevDiv 翻译：
Hsany330 痛饮狂歌 fengyun1989 lanseshuibei BeyondVincent(破船)
DevDiv 校对： BeyondVincent(破船)
DevDiv 编辑： BeyondVincent(破船)

写在前面

目前，移动开发被广大的开发者们看好，并大量的加入移动领域的开发。

鉴于以下原因：

- 国内的相关中文资料缺乏
- 许多开发者对 E 文很是感冒
- 电子版的文档利于技术传播和交流

DevDiv.com 移动开发论坛特此成立了翻译组，翻译组成员具有丰富的移动开发经验和英语翻译水平。组员们利用业余时间，把一些好的相关英文资料翻译成中文，为广大移动开发者尽一点绵薄之力，希望能对读者有些许作用，在此也感谢组员们的辛勤付出。

关于 DevDiv

DevDiv 已成长为国内最具人气的综合性移动开发社区。

更多相关信息请访问 DevDiv 移动开发论坛。

技术支持

首先 DevDiv 翻译组对您能够阅读本文以及关注 DevDiv 表示由衷的感谢。

在您学习和开发过程中，或多或少会遇到一些问题。DevDiv 论坛集结了一流的移动专家，我们很乐意与您一起探讨移动开发。如果您有什么问题和技术需要支持的话，请访问网站 www.devdiv.com 或者发送邮件到 BeyondVincent@DevDiv.com，我们将尽力所能及的帮助你。

关于本文的翻译

感谢 Hsany330、痛饮狂歌、fengyun1989、lanseshuibei 和 BeyondVincent(破船)对本文的翻译，同时非常感谢 BeyondVincent(破船)在百忙中抽出时间对翻译初稿的认真校验，指出了文章中的错误。才使本文与读者尽快见面。由于书稿内容多，我们的知识有限，尽管我们进行了细心的检查，但是还是会存在错误，这里恳请广大读者批评指正，并发送邮件至 BeyondVincent@devdiv.com，在此我们表示衷心的感谢。

本书翻译汇总贴请看下面链接

Metro Revealed Building Windows 8 apps with XAML and C#中文翻译汇总

其他学习资料推荐

Windows 8 Metro App 开发 Step by Step

更多相关资料请关注

Windows 8 开发论坛

目录

第 1 章 入门

在微软的 Windows 8 中，Metro 应用是非常重要的一部分，针对台式机、平板电脑和智能手机，Metro 应用提供了一对一、一致性的编程和互动模式。Metro 应用的用户体验与之前传统的 Windows 应用有非常大的区别：Metro 风格的应用是全屏的，并且从娱乐的角度来看，它支持的风格是简单、直接和易用。

Metro 应用与之前的 Windows 版本的应用完全不同。Metro 应用提供了新的 API、新的交互控件和使用完全不同的方法来管理应用的生命周期。开发 Metro 应用可以使用的语言非常多，包括 javaScript，Visual Basic，C++和这本书使用的语言 C#。Windows 8 立足于大家都熟悉的开发，开发者可以使用目前对 C#和 XAML 的经验来开发丰富的 Metro 应用，并融入到广阔的 Windows 平台。本书提供基本的知识，来帮助开发者进入到 Metro 世界中。最后，通过此书，你将会学到如何使用控件和 Metro 核心体验特性。

注意：微软的说法是 Metro 风格(Metro style)和 Metro 风格应用(Metro-style app)。我不会使用这复杂的说法，我用 Metro 和 Metro 应用来表示，如果需要，你可以精神上插入 style。

1.1. 关于此书

本书可以让你初尝 Windows 8 Metro App 的开发。

1.1.1. 阅读本书之前，需要知道什么？

你必须熟练的知道 C#，最好 XAML 也懂。如果你做过 WPF 或 Silverlight 工程，那么你将有足够的 XAML 知识来创建 Metro 应用。如果没有做过 XAML 开发，也不用担心，你去学，也很容易。在本章中，我会给出一个简单的介绍，帮助你开始了解 XAML。我会列出我们用到 XAML 的主要内容。

1.1.2. 学习这本书，需要什么软件？

你需要 Windows 8 发行预览版和 Windows Visual Studio 2012 RC。除了上述软件外，你不需要任何其它的工具来开发 Metro 应用和学习本书给出的例子。【译者注：译者在这里进行更新：目前最新版是 Windows 8 RP】

Windows 8 发行预览版是一个未完成的产品，它还有些不稳定的问题。如果在指定 PC 机型上安装 Windows 8，将会获得最好的体验，但是如果你还没有准备好，可以在虚拟机上安装。

1.1.3. 本书的结构是什么？

本书主要集中在开发一个 Metro 应用涉及到的关键技术和特性。你已经知道如何写 C#，我不会把时间浪费在你已经具有的知识上。本书帮助你，并把你具有的 C#和 XAML 开发经验转换到 Metro 世界中，也就是说焦点集中在开发 Metro 应用细节上。

我将用一种轻松的方式结合各主题。除了每章的主干（main theme），你将会看到一些必要的内容去解释为什么（那些）特性很重要，为什么你应该实现他们。在这本书的最后，你将会明白如何去构建一个集成到 win8 里面的应用 Metro 应用,以及如何展现用户体验，你可用用 c++或 javascript 等其他语言来编写 Metro 应用。这只是你编写 win8Metro 应用的开始，本书并不是一个全面的教程，因此，我关注 Metro 应用的主要构建部分。由于 win8 的内容很多，本书没有完全覆盖，如果你想了解得更多，请参考 Jesse Liberty 的《Pro Windows 8 Development with XAML and C#》，这本书是针对 win8 的最终发布版。他们还会发布《Pro Windows 8 Development withHTML5 and JavaScript》，如果你想用面向 web 方面的技术去构建你的 Metro 应用，请参考这本书。本书的章节主要是：

第 1 章：入门

这一章，除了介绍本书，我还将介绍如何用 VS 创建 Metro 工程，这可贯穿整书。我给出 XAML 的简要概述，解说在 Metro 开发工程里面重要的文件，告诉你如何运行模拟器，以及如何调试。

第 2 章：数据（Data）

绑定（Bingding),和页面（Page)数据是 Metro 应用的核心，在这一章节里面展示如何定义 view model，以及如何用 Metro data bingding 给你的应用展现层带来数据。这些技术是构建 Metro 应用的必要部分，并且他们很容易扩展和维护。然后我将介绍如何用 pages 把你的应用分成 XAML 和 C#代码。

第 3 章：AppBars ,Flyouts,和导航栏（NavBars）

这些是通用的用户操作控件，与用什么语言去创建他们无关。在这一章节，我将演示如何创建配置 AppBar，Flyouts 和 NavBars，这些是通用控件里面最重要的。

第 4 章：Layouts 和 Tiles

Metro 应用的功能扩展到 win8 的开始画面，就是提供一些展示额外信息的方式。在这一章，我将演示如何去创建和动态更新开始瓷片（tiles),以及如何运用瓷片的徽章（badges）。

我也将演示如何图处理 Metro 风格的 snapped 和 filled 布局。你只要用一点代码和 XAML 就能构建出这些布局。两种方法我都会演示。

第 5 章：应用生命周期和 Contracts

win8 的 Metro 应用拥有一种与众不同的生命周期模型。在这一章节，我会解析这种模型如何工作的，演示如何接受和响应生命周期中的大部分事件。并且解析，以及在挂起和运行转换中的管理。我也会演示如何创建异步任务，以及当你的应用切换到后台后，如何控制这些异步任务。最后，我演示如何支持 Metro 的 contracts.contracts 让你的应用无缝集成更广泛的功能。

1.1.4. 关于 Metro 应用的例子

这本书的例子是购物店清单，这种方式称为 MetroGrocer，每个应用都是独立的。MetroGrocer 很单调，但这足以演示大部分 Metro 特点的完美平台。在图 1-1，你能看到这个应用长啥样。

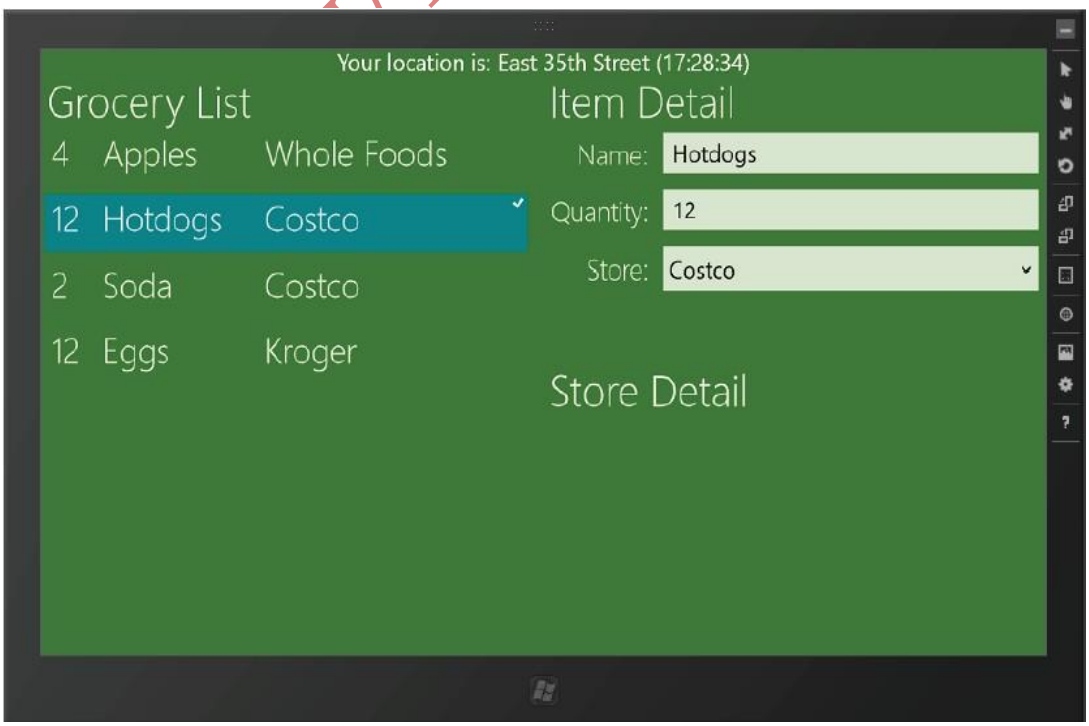

Figure 1-1. 示例程序

这是一本关于如何编写而不是设计的书，MetroGrocer 不是一个很漂亮的应用，甚至我都没有实现它的全部功能。它的就是一个演示编写代码技术的方式，简洁，纯净。如果你想学设计，那你选错书了，如果你是一个码农，好吧，你找对地方了。

1.1.5. 这本书有很多代码吗？

是的，事实上代码太多了，要不是有编辑，我都不搞不过来。当我介绍一个新主题或者做很多更改的时候，我将演示完整的 C#代码或者 XAML 文件。当我只做一小点改动或者强调几个关键点，我将展示一些代码段并用高亮标记重点更改的地方。你能在 Listing1-1 里面看到这些长啥样，这是从第 5 章截取的。

```
protected override void OnLaunched(LaunchActivatedEventArgsargs) {
    if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) {
        //TODO: Load state from previously suspended application
    }
    // Create a Frame to act navigation context and navigate to the first page
    varrootFrame = new Frame();
rootFrame.Navigate(typeof(Pages.MainPage));

    // Place the frame in the current Window and ensure that it is active
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
}
```

这种方式可以让我很容易加入更多代码到书里，可是这样的话你在 visual studio 里面操作这些例子就麻烦了。如果你想操作这些例子，最好的方法是从 Apress.com 下载源码。书里面的例子都是免费的，甚至连工程文件都包含在里面了。

1.2. 开始和运行

在这一章里，我会创建一个工程，并且展示 Visual Studio 工程的每一个元素。我会把整个过程分解成一步步的操作。如果你喜欢，你可以从 Apress.com 下载已经做好的工程。

1.2.1. 创建工程

运行 Visual Studio，然后选择“File->New Project”。在这个新的工程对话框里，从模板里面选择”Visual C#”，然后从可选的模板中选择”Blank Application”。如图图 1-2.所示：

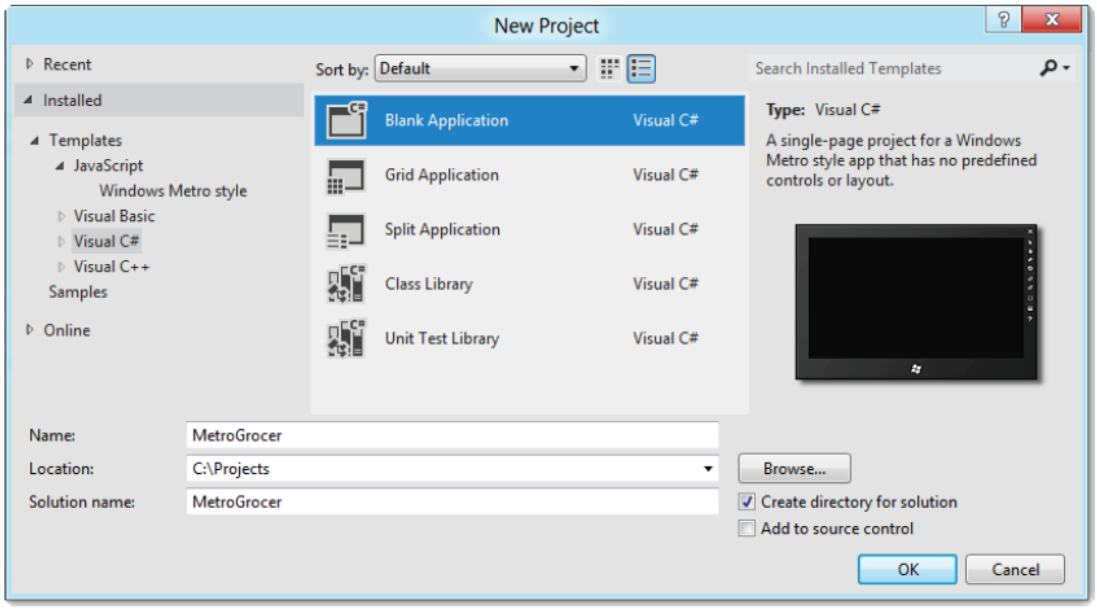

图 1-2. 创建示例工程

设置工程的名字，然后点击 OK 按钮，Visual Studio 就会创建和部署工程。

图 1-3.通过 Visual Studio Solution Explorer 展示新工程的内容。在余下的章节，我将描叙工程里面最重要的文件。

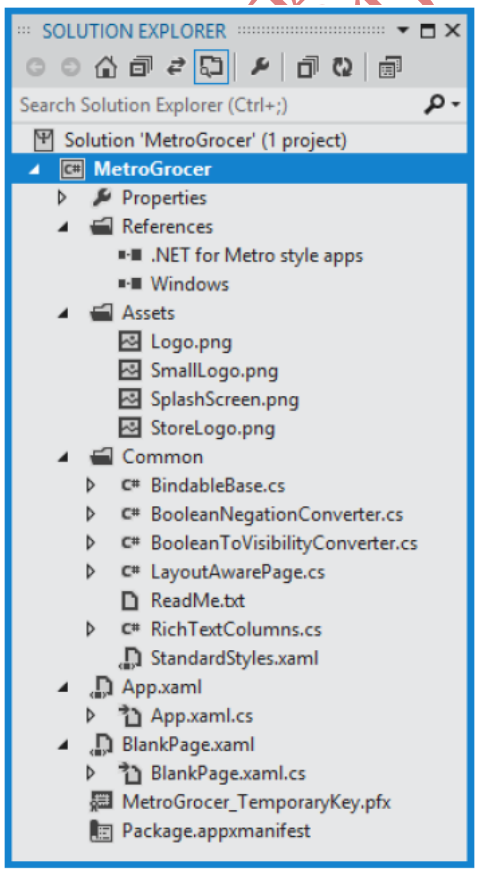

图 1-3.使用 Blank Application template 创建的工程内容

提示：Metro 应用使用简装版的.NET 框架库。你可以在解决方案管理器里面通过双击 References 里面的项看到他们对应的命名空间。

1.2.2. 浏览 App.xaml 文件

App.xaml 文件和它对应的代码文件 App.xaml.cs，是用于启动 Metro app。这个 XAML 文件的主要用途是从 Common 文件夹关联 StandardStyles.xaml。如列表 1-2 所示：

列表 1-2. The App.xaml File

```
<Application
    x:Class="MetroGrocer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer">

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

我简短地说明下 Standardstyles.xaml 文件，在本章的后面，我将更新 App.xaml 去引用我自己的资源字典。代码文件更有趣。具体如列表 1-3 所示：

列表 1-3. The App.xaml.cs File

```
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace MetroGrocer {
    sealed partial class App : Application {

        public App() {
            this.InitializeComponent();
                this.Suspending += OnSuspending;
        }
        protected override void OnLaunched(LaunchActivatedEventArgs args) {
            if (args.PreviousExecutionState ==
ApplicationExecutionState.Terminated) {
                //TODO: Load state from previously suspended application
            }
            // Create a Frame to act navigation context and navigate to the
first page
    var rootFrame = new Frame();
    rootFrame.Navigate(typeof(BlankPage));

    // Place the frame in the current Window and ensure that it is active
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
    }
    void OnSuspending(object sender, SuspendingEventArgs e) {
        //TODO: Save application state and stop any background activity
        }
    }
}
```

Metro 应用有很特别的生命周期。它是通过 App.xaml.cs 文件来交流的。关键是要理解和接受这种模式，具体内容我这第五章节解释。这时，你只需要知道 OnLanunched 方法是在应用开始的时候被调用的。这开始

的时候，Blankpage 类会在被加载的时候创建一个新的实例,这个实例将作为应用的主接口。

1.2.3. 浏览 BlankPage.xaml 文件

页面是 Metro 应用基本的编译块。当你创建一个空白的应用模板，Visual Stuido 就创建一个空白的页面。列表 1-4 展示 BlankPage.xaml 文件的内容。【译者注：在 Windows 8 RP 版里面叫 MainPage.xaml】

列表 1-4. The Contents of the BlankPage.xaml File

```
<Page
    x:Class="MetroGrocer.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

  <Grid Background="{StaticResourceApplicationPageBackgroundBrush}">
  </Grid>
</Page>
```

如果你以前用过 XAML，你一定认识 Grid 控件。Metro UI 控件大体上和 WPF 或 Silverlight 的工作机制一样，但控件数相对少了些，并且一些高级的布局和数据绑定的特性不可用。我会在第二章创建更多的有用的 Page 布局。如果你有 XAML 的经验就感觉这个文件对应的代码文件有点熟悉。具体展示如列表 1-5 所示:

列表 1-5. The Contents of the BlankPage.xaml.cs File

```
usingWindows.UI.Xaml.Controls;
usingWindows.UI.Xaml.Navigation;

namespaceMetroGrocer {
    public sealed partial class BlankPage : Page {
        publicBlankPage() {
            this.InitializeComponent();
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
    }
}
```

1.2.4. 浏览 StandardStyles.xaml 文件

在 Visual Studio 工程模板的“Common”文件夹里，我唯一关心的就是 StandardStyles.xaml，它是一个资源文件，这些浏览资源文件关系到 App.xaml(Listing1-2)。StandardStyles.xaml 文件包含一些风格和模板，这让创建应有更容易。我不打算完全列举这个文件，因为它包含了很多内容，当列表 1-6 展示一个文本相关风格的例子。

列表 1-6. StandardStyles.xaml 文件中的一个 Style

```
...
<Style x:Key="HeaderTextStyle" TargetType="TextBlock"
        BasedOn="{StaticResourceBaselineTextStyle}">
    <Setter Property="FontSize" Value="56"/>
    <Setter Property="FontWeight" Value="Light"/>
    <Setter Property="LineHeight" Value="40"/>
    <Setter Property="RenderTransform">
```

```
            <Setter.Value>
                <TranslateTransform X="-2" Y="8"/>
                </Setter.Value>
    </Setter>
</Style
```

1.2.5. 浏览 Package.appxmanifest 文件

最后值得一提的便是这个应用设置文件了，“Package.appxmanifest”。这是一个提供你的 Metro 应用信息给 Windows runtime 的 XML 文件。你可以编辑原始的 XML，不过 Visual Studio 提供了很好的属性编辑器，我会在后面的章节里回来讲如果配置应用设置。

1.3. XAML 的概况

别担心如果你以前没有接触过 XAML。如果你在别的 XAML 应用里使用过一些高级特性，有可能不被 Metro 应用支持。

它的核心是，XAML 创建使用接口的声明，而不是代码，所以，如果我想加一堆按钮到我的工程，我就是加一堆标记到我的 XAML 文件。如列表 1-7 所示。

列表 1-7. Adding Controls to the XAML Document

```
<Page
    x:Class="MetroGrocer.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResourceApplicationPageBackgroundBrush}">
    <StackPanelHorizontalAlignment="Center" VerticalAlignment="Center">
      <Button x:Name="FirstButton" HorizontalAlignment="Center"
            Click="ButtonClick">Click Me!</Button>
      <Button Style="{StaticResourceTextButtonStyle}"
            HorizontalAlignment="Center"
            Click="ButtonClick">Or Click Me!</Button>

    </StackPanel>
    </Grid>
</Page>
```

在 XAML 元素里面的标签指示的控件都会被加到布局里面。我添加了一个 StackPanel 和两个 Button 控件到我的工程。StackPanel 是个简单的容器，它用于帮助添加结构到布局中；它的子元素不是水平布局就是垂直布局。Button 控件正如你预想的：当用户点击的时候这个按钮触发一个事件。

XML 的层次性对应于 UI 控件的层次。在 StackPanel 部署 Button 元素，我指定了 StackPanel 为 Button 元素负责。

1.3.1. 使用 Visual Studio 设计界面

你可以用 C#代码在 Metro 工程里实现所有事情,可以不用 XAML。但是也有引入注目的事情让你去使用 XAML. 主要的优势是：在 Visual Studio 支持用 XAML 来设计。在大部分的时候，它能及时反馈效果。如图 1-4 所示，Visual Studio 显示 StackPanel 和 Button 元素在它的 XAML 设计界面上。虽然这和真实运行的程序不一样，但它能真实展现具体情况，如果用 C#代码来创建，这些交互界面是看不到的。

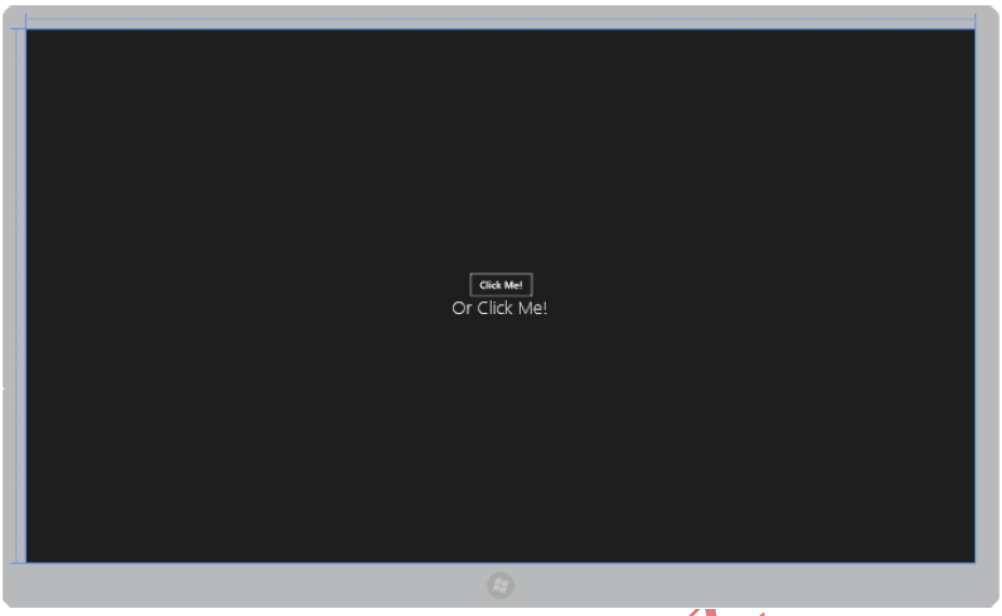

图 1-4. Visual Studio 将 XAML 文件内容反映到设计界面中

尽管 XAML 很冗长，但 Visual Studio 做了很多事情让创建和编辑更容易。他还有一些完美的自动完成的特性，它为标签名称，属性和值提供建议。你可以从工具箱中拖动控件来设计界面。如果这些都不适合你，你可以通过 Blend 为 Metro 应用创建 XAML。Blend 是安装在你机器上的 Visual Studio 的一部分。我喜欢在代码编辑器里编辑 XAML。作为程序老油条，我一直没有真正的可视化设计工具。尽管这些工具这些年变得相当好。你也许不一样，你应该为 UI 开发尝试不同的适合你的风格。为了这个目的在这本书里，我将展现直接更改 XAML。

1.3.2. 在 XAML 里配置控件

通过在 XAML 里进行属性设置，可以配置 Metro UI 控件。所以，比如：我想 StackPanel 的子元素都居中对齐。为到达这个，我为 HorizontalAlignment 和 VerticalAlignment 属性设值。像这样：

```
..
<StackPanelHorizontalAlignment="Center" VerticalAlignment="Center">
...
```

应用样式

属性也被用于应用样式在 UI 控件。举个例子，我提供 TextButtonStyle，这是在微软的 StandardStyles.xaml 文件里面定义的。

```
...
<Button Style="{StaticResourceTextButtonStyle}" HorizontalAlignment="Center"
Click="ButtonClick">Or Click Me!</Button>
...
```

这有很多种不同的方式在 XAML 里定义和引用样式，我用 StaticResourc 去指定我想要的样式，但这也有选项从各种文件里面去获取样式的信息。我打算在这本书里让事情简单，并且尽可能集中在 Metro 特定的功能。

指定事件处理(Handlers)

为一个事件指定处理方法，你简单地使用元素属性，它对应于你请求的相应事件。像这样：

```
...
```

```
<Button x:Name="FirstButton" HorizontalAlignment="Center"
Click="ButtonClick">Click Me!</Button>
...
```

我已经指定了 click 事件，它对应 ButtonClick 方法。当你提供一个事件属性时，Visual Studio 将提供一个事件处理的方法。我会在下一章展示这个关系的另一面。

1.3.3. 在代码里配置控件

这依赖于一些聪明技巧和 C#特性，被称为局部函数。标记 XAML 文件转换和代码隐藏文件相结合以建立单一的.NET 类.在开始时，这看起了有点老，但是他允许混合模式，你可以在 XAML 和对应的代码文件里定义和配置控件。

列表 1-8 展示 BlankPage.xaml.cs 文件，它是 BlankPage.xaml 文件对应的代码文件。

列表 1-8. BlankPage.xaml.cs 文件

```
usingWindows.UI.Xaml;
usingWindows.UI.Xaml.Controls;
usingWindows.UI.Xaml.Navigation;

namespaceMetroGrocer {
    public sealed partial class BlankPage : Page {
        publicBlankPage() {
            this.InitializeComponent();
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
    private void ButtonClick(object sender, RoutedEventArgs e) {
        System.Diagnostics.Debug.WriteLine("Button Clicked: "
            + ((Button)e.OriginalSource).Content);
        }
    }
}
```

我可以在没有参照 XMAL 文件的情况下调用 ButtonClick 方法，因为从 XAMl 文件里面产生的代码和 c#代码在背后被合并成一个类。结果是当 app 的 Button 元素被点击，我的 C# ButtonClick 方法被调用。

注意一些在 XAML 文件里面的元素有 x:Name 属性。比如：

```
...
<Button x:Name="FirstButton" HorizontalAlignment="Center"
Click="ButtonClick">Click Me!</Button>
...
```

当你为这个属性指定一个值时，编译器就创建一个变量，这个变量的值就是 ui 控件，这个 ui 控件就是从 XAML 元素里面创建出来的。这就意味着你可以通过你的程序来支持 XAML 的配置。在列表 1-9，我更改按钮的配置，这个按钮的名字是 FirstButton，它在 ButtonClick 方法里面.

Listing 1-9. 在代码中的响应事件中配置控件

```
...
private void ButtonClick(object sender, RoutedEventArgs e) {
FirstButton.Content = "Pressed";
FirstButton.FontSize = 50;
    System.Diagnostics.Debug.WriteLine("Button Clicked: "
        + ((Button)e.OriginalSource).Content);
}
```

...

我没有用任何方式指定控件的名字。在这个例子里面，我更改了 button 的内容和大小以及字体。一旦 click 事件得到处理，FirstButton 的配置就会发生变化。这就是你目前需要知道关于 XAML 的内容。总结：

1、XAML 被转化成代码，然后和对应的代码文件混合创建一个单独的.NET 类。
2、你可以从 XAML 或者代码里面配置 UI 控件。
3、使用 Visual studio 设计工具处理 XAML，它很好用。XAML 一开始读起来很难，但你很快就能习惯它。我发现用 XAML 比用 c#代码要简单。

1.4. 运行和调试 Metro 应用

现在我们有一个很简单的 Metro 应用，是时候关注如何运行和调试 Metro 应用了。Visual Studio 提供了三种方式来运行 Metro 应用：在本地机器上，在模拟器上，或者在远程机器上。

目前最好的方式是用 Visual Studio 模拟器，它提供可信赖的 Metro 体验并且让你更改设备的能力。包括更改屏幕大小，模拟触摸事件，以及产生合成地理数据。从弹出菜单选择模拟器，如图 1-5 所示。

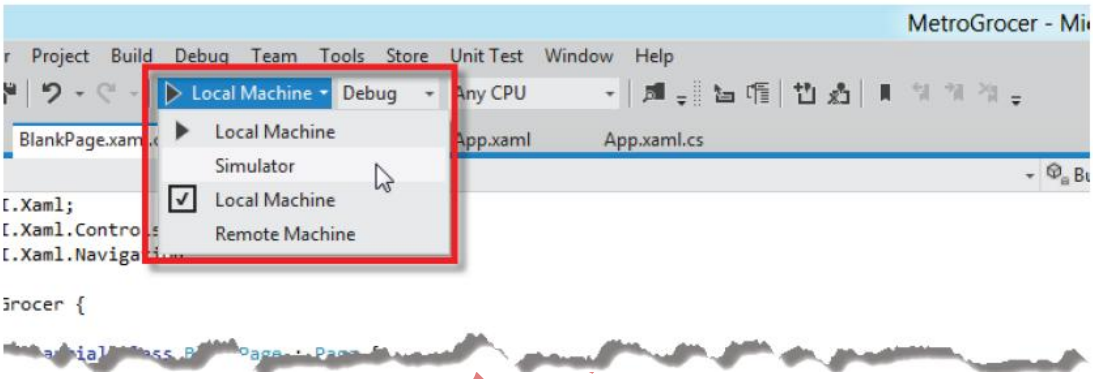

图 1-5. 选择 Visual Studio simulator 来测试 Metro apps

1.4.1. 在模拟器里运行 Metro App 模拟器

运行例子应用，点击工具栏按钮,或者 Debug 菜单选择“Start with Debug”。Visual Studio 将打开模拟器，然后编译和部署 Metro 应用。如下所示图 1-6.

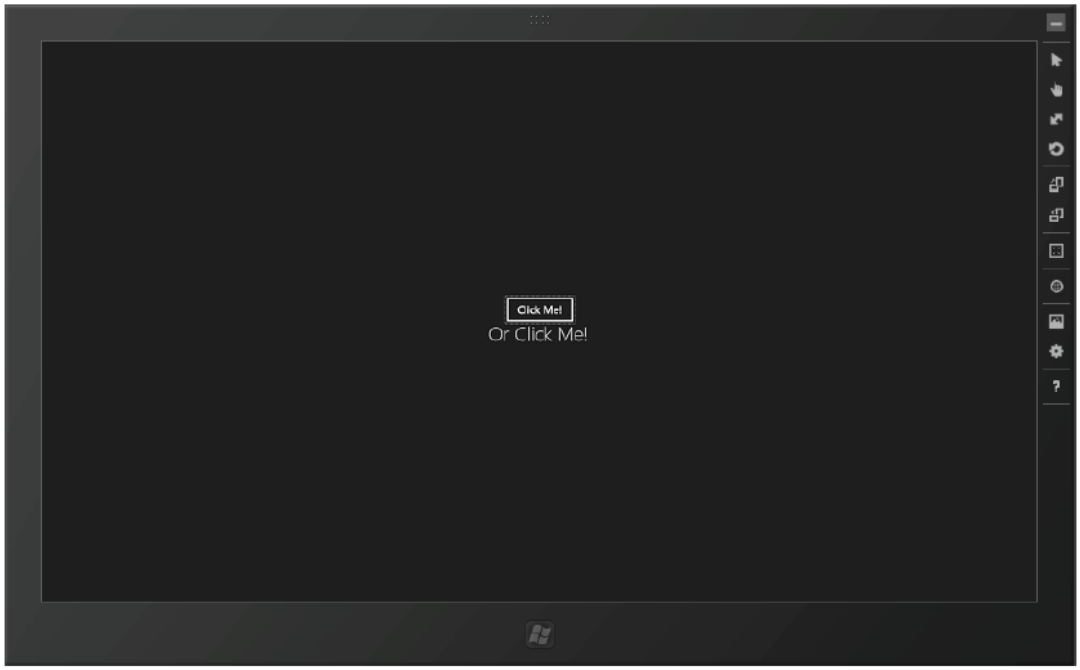

图 1-6. 使用 Visual Studio simulator

你可以用右边的按钮去更改屏幕大小和横竖屏。切换鼠标和触摸输入，以及地理信息数据。这里没有更多的可看了，因为只是个简单的例子。

点击应用里面的 button 去触发事件处理方法，然后更改按钮的配置。如图 1-7 所示：

图 1-7.点击按钮的结果

你会在输出窗口看到一条消息，类似如下：

```
Button Clicked: Pressed
```

Metro 应用在 debugger 的状态下，任何异常都会导致 debugger 中断。除了运行和调试用 Visual Studio 设备，你可以通过模拟器设置断点来强制调试。

1.5. 总结

在这一章，我提供了本书的预览和介绍用 XAML 和 C#编写 Metro 应用。我提供很基础的 XAML 的预览以及展示创建简单应用。在下一章，我将开始视图模型，并添加主要的结构组件来构建应用。

第 2 章 数据、绑定与页面

在本章中，我将向你展示如何定义与使用构成 Metro 应用程序核心的数据。为此，我将松散地遵循视图模型模式，这样可以帮助我清楚地将数据与应用程序中负责显示数据及处理用户交互的部分相分离开。

从诸如 Model-View-Controller (MVC，模型-视图-控制器)和 Model-View-View Controller (MVVC，模型-视图-视图控制器)等设计模式中，你可能已熟悉了视图模型。在这本书中，我并不会深入探讨这些模式的细节，在维基百科上有许多有关 MVC 与 MVVC 的有益信息可用，其中不乏公正并具深刻见解的描述。

我发现使用视图模型的好处是巨大的，对于所有项目都是值得考虑的，除了最简单的 Metro 项目，并且我建议你参照本书同样的思路来认真考虑视图模型的使用。我不是一个模式狂热者，我只采用能解决实际问题的那部分模式与技术，并改写它们以便用在特定的项目中。为此，你会发现我花费了一些篇幅来论述应该如何使用一个视图模型。

本章聚焦于一个 Metro 应用程序的幕后事物，打下一个我可以构建来演示不同 Metro 特性的基础。我将逐步地讲解，先定义一个简单的视图模型，并展示使用数据绑定将来自视图模型的数据显示在应用程序 UI 中的与众不同的技术。然后我将教你如何分解应用程序为多个页面，并将这些页面带入主布局，以及改变页面用来反映 Metro 应用程序的状态。表 2 - 1 提供了本章所有知识点的总结。

注意

如果你是使用下载自 Apress.com 的源代码来学习，你需要卸载来自前一章的 Metro 示例应用程序——在模拟器中用鼠标右键单击开始屏幕上的 MetroGrocer 磁贴并选择 Uninstall（卸载）。如果你从不同的项目路径运行一个已安装的应用程序，Visual Studio 将报告一个错误。因此当你从一章转至另一章时，你必须先将原示例应用程序卸载。

表 2-1. 本章知识点小结

问题	解决方案	代码清单
创建一个可观测（observable） 类	实现 INotifyPropertyChanged 接口。	1
创建一个可观测集合	使用 ObservableCollection 类。	2
改变 Metro 应用程序启动时载入的页面	改变在 App.xaml.cs 的 OnLaunched 方法中指定的页面类型。	3
设置数据绑定值的来源	使用 DataContext 属性。	4
创建可重用的样式与模板	创建一个资源字典。	5, 6
绑定 UI 控件至视图模型	使用 Binding 关键字。	7
加入另一页面至应用程序布局	添加一个 Frame 控件到主布局并使用 Navigate 方法指定要显示的页面。	8, 10
动态插入页面至应用程序布局	使用 Frame.Navigate 方法，可选择性地传递一个与上下文场景相关的视图模型对象给内嵌的页面。	11

2.1. 添加视图模型

好的 Metro 应用程序的核心是一个视图模型，使用它可以让我将应用程序数据与呈现数据给用户的方法相分离。对于创建易持续增强与维护的应用程序，视图模型是一个必要的基础。在特定的 UI 控件中直接处理数据犹如权威般迷惑人，但如果那样做的话，你很快就会发现要找出你的数据在何处以及它将如何更新会变得越来越难以管理。

首先，我在 Visual Studio 项目中创建了一个新的 Data 文件夹以及两个新的类文件。第一个定义了 GroceryItem 类，其表示购物清单上的单个采购项。你可以在代码清单 2-1 中看到 GroceryItem.cs 的内容。

代码清单 2-1 GroceryItem 类：

```
using System.ComponentModel;
namespace MetroGrocer.Data {
    public class GroceryItem : INotifyPropertyChanged {
        private string name, store;
```

```
        private int quantity;
        public string Name {
            get { return name; }
            set { name = value; NotifyPropertyChanged("Name"); }
        }
        public int Quantity {
            get { return quantity; }
            set { quantity = value; NotifyPropertyChanged("Quantity"); }
        }
        public string Store {
            get { return store; }
            set { store = value; NotifyPropertyChanged("Store"); }
        }
        public event PropertyChangedEventHandler PropertyChanged;
        private void NotifyPropertyChanged(string propName) {
            if (PropertyChanged != null) {
                PropertyChanged(this, new PropertyChangedEventArgs(propName));
            }
        }
    }
}
```

这个类定义了要采购项的名称、数量及哪里能买到它的商店属性。这个类唯一值得关注的地方在于它是可观测的。Metro UI 控件一个很好的特性是它们支持数据绑定，这意味着当他们所显示的可观测数据发生变化时控件将会自动更新。

实现 System.ComponentModel.INotifyPropertyChanged 接口以使得类是可观测的，并在任一可观测属性被修改时触发由这个接口指定的 PropertyChangedEventHandler 事件处理器。

我加入到 Data 命名空间中的另一个类是 ViewModel，其包含在 ViewModel.cs 中。这个类包含用户数据与应用程序状态，你可以在代码清单 2-2 中看到该类的定义。

代码清单 2-2 ViewModel 类

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
namespace MetroGrocer.Data {
    public class ViewModel : INotifyPropertyChanged {
        private ObservableCollection<GroceryItem> groceryList;
        private List<string> storeList;
        private int selectedItemIndex;
        private string homeZipCode;
        public ViewModel() {
            groceryList = new ObservableCollection<GroceryItem>();
            storeList = new List<string>();
            selectedItemIndex = -1;
            homeZipCode = "NY 10118";
        }
        public string HomeZipCode {
            get { return homeZipCode; }
            set { homeZipCode = value; NotifyPropertyChanged("HomeZipCode"); }
        }
        public int SelectedItemIndex {
            get { return selectedItemIndex; }
            set {
                selectedItemIndex = value; NotifyPropertyChanged("SelectedItemIndex");
            }
        }
```

```
        public ObservableCollection<GroceryItem> GroceryList {
            get {
                return groceryList;
            }
        }
        public List<string> StoreList {
            get {
                return storeList;
            }
        }
        public event PropertyChangedEventHandler PropertyChanged;
        private void NotifyPropertyChanged(string propName) {
            if (PropertyChanged != null) {
                PropertyChanged(this, new PropertyChangedEventArgs(propName));
            }
        }
    }
}
```

这个简单的视图模型中最重要的组成部分是代表购物清单的 GroceryItem 对象集合。我想要这个清单是可观测的，那样对清单的改变将自动更新应用程序的 UI 控件。为此，我使用了一个来自 System.Collections.ObjectModel 命名空间的 ObservableCollection 类。这个类实现了一个集合的基本特性，并在向清单中的项有被添加、删除或替换时发送事件通知。注意 ObservableCollection 类本身并不会发送一个事件通知，当它包含的某个对象的数据值被修改时，只有通过创建一个可观测的 GroceryList 对象的可观测集合，才能确保对购物清单所做的任何改变都将导致 UI 控件的更新。

ViewModel 类也同样实现 INotifyPropertyChanged，因为视图模型中有两个可观测属性。第一个属性 HomeZipCode 是用户数据，我将在第三章中用来演示如何创建弹出框。第二个可观测属性 SelectedItemIndex 是应用程序状态的组成部分，跟踪采购清单中的哪一项是被用户选中的（如果有的话）。

这是一个非常简单的视图模型，且如我在前面提及的，我对如何在项目中构造视图模型作了充分的论述。也就是说，它包含了所有的要素——我要用来演示如何使用数据绑定以使我的 Metro UI 控件自动更新。

2.2. 添加主页

现在我已经完成了视图模型的定义，我可以开始设计用户界面了。第一步是加入应用程序主页面。我能理解为什么 Visual Studio 会生成一个名为 BlankPage 的页面，但它不是一个有用的名字，所以我首先是添加一个具有更明了名称的页面。

提示

我将不再使用 BlankPage.xaml，所以你可以从项目中删除它。

我喜欢使用更具结构层次的项目，因此我先添加了一个新的 Pages 文件夹到项目中，再用鼠标右键单击 Pages 文件夹，选择弹出菜单中的 Add ä New Item 菜单项，然后选取 Blank Page 模板来添加了一个新的 ListPage.xaml 页面。Visual Studio 将创建该 XAML 文件及其代码隐藏文件 ListPage.xaml.cs。

提示

如果你对使用 XAML 构建应用程序还不熟悉，那么理解你通常不会按我在讲述示例应用程序所采取的顺序工作是重要的。在你使用 XAML 声明控件，添加代码以支持他们，以及定义样式来减少标记中的重复内容时，XAML 方法支持更加迭代的风格，它是一种比我展现在这里的更加自然的方法，但它难以用在一本书中说明基于 XAML 开发应用程序的本质。

我想让 ListPage.xaml 这个页面在我的 Metro 应用程序启动后加载，这需要更改 App.xaml.cs，如代码清单 2-3 所示。

代码清单 2-3 更新 App.xaml.cs 以使用 ListPage

```
using Windows.ApplicationModel;
```

```
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer {
    sealed partial class App : Application {

public App() {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }
        protected override void OnLaunched(LaunchActivatedEventArgs args) {
            if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) {
                //TODO: Load state from previously suspended application
            }
            // Create a Frame to act navigation context and navigate to the first page
            var rootFrame = new Frame();
            rootFrame.Navigate(typeof(Pages.ListPage));
            // Place the frame in the current Window and ensure that it is active
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
        void OnSuspending(object sender, SuspendingEventArgs e) {
            //TODO: Save application state and stop any background activity
        }
    }
}
```

暂时不去理会这个类的其余部分，我将在第五章讲到怎样响应 Metro 应用程序的生命周期时，再返回来讲这个类。

2.2.1. 编写代码

对我来说，以与你通常在项目中所用方式相反的顺序来讲解我所创建的应用程序如何呈现内容是最简单方式。为此，我将从 ListPage.xaml.cs 代码隐藏文件开始。你可以在代码清单 2 - 4 中看到我在这个文件中对 Visual Studio 默认内容所做的添加。

代码清单 2-4 ListPage.xaml.cs 文件

```
using MetroGrocer.Data;

using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
namespace MetroGrocer.Pages {
    public sealed partial class ListPage : Page {
        ViewModel viewModel;
        public ListPage() {
            viewModel = new ViewModel();
            viewModel.StoreList.Add("Whole Foods");
            viewModel.StoreList.Add("Kroger");
            viewModel.StoreList.Add("Costco");
            viewModel.StoreList.Add("Walmart");
            viewModel.GroceryList.Add(new GroceryItem { Name = "Apples",
                Quantity = 4, Store = "Whole Foods" });
            viewModel.GroceryList.Add(new GroceryItem { Name = "Hotdogs",
                Quantity = 12, Store = "Costco" });
            viewModel.GroceryList.Add(new GroceryItem { Name = "Soda",
                Quantity = 2, Store = "Costco" });
            viewModel.GroceryList.Add(new GroceryItem { Name = "Eggs",
```

```
                Quantity = 12, Store = "Kroger" });
            this.InitializeComponent();
            this.DataContext = viewModel;
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
        private void ListSelectionChanged(object sender, SelectionChangedEventArgs e) {
            viewModel.SelectedItemIndex = groceryList.SelectedIndex;
        }
    }
}
```

小心：
如果你此时编译程序将得到一个错误，因为我所引用的 groceryList 控件我还没有添加。你应该等到“运行应用程序”章节，那是一切就位之时。

ListPage 类的构造函数创建一个新的 ViewModel 对象并以一些示例数据填充它。这个类中最值得关注的语句是这条：

```
this.DataContext = viewModel;
```

Metro UI 控件的核心特性是支持数据绑定，通过数据绑定我可以在 UI 控件中显示来自视图模型的内容。要做到这一点，我必须指定我的数据来源。DataContext 属性指定绑定到一个 UI 控件及其所有子控件的数据的来源。我可以使用 this 关键字来为整个布局设置 DataContext，因为 ListPage 类是由代码隐藏文件的内容合并 XAML 文件的内容而成的，意味着 this 引用的页面对象包含所有以 XAML 方式声明的控件。

我最后添加的内容定义了一个方法，处理来自 ListView 控件的 SelectionChanged 事件。这是我将用来显示购物清单中所有项的控件。在我定义 XAML 时，我会做出安排，以便在用户选中某一采购项时调用该方法。该方法基于 ListView 控件的 SelectedIndex 属性来设置视图模型中的 SelectedItemIndex 属性。因为 SelectedItemIndex 属性是可观测的，当用户做出选择时，我的应用程序的其他部分能够收到通知。

2.2.2. 添加资源字典

在第一章中，我解释了由 Visual Studio 创建的 StandardStyles.xaml 文件所定义的若干用于 Metro 应用程序的 XAML 样式和模板。像这样定义样式是一个好的主意，因为这意味着你只需在一处地方做出改变，而不用去找出你对 UI 控件直接应用颜色或字体设置的所有地方进行更改。我的示例项目需要用到一些标准样式和模板，为此，我创建了一个新的 Resources 文件夹，并用 Resource Dictionary（资源字典）项模板创建了一个新的 GrocerResourceDictionary.xaml 文件。你可以在代码清单 2-5 中看到这个文件的内容。

提示：
如我在第一章中解释的，微软禁止添加样式到 StandardStyles.xaml 中。你必须创建你自己的资源字典。

代码清单 2-5 定义资源字典

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Resources">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/Common/StandardStyles.xaml" />
    </ResourceDictionary.MergedDictionaries>

<SolidColorBrush x:Key="AppBackgroundColor" Color="#3E790A"/>
    <Style x:Key="GroceryListItem" TargetType="TextBlock"
            BasedOn="{StaticResource BasicTextStyle}" >
        <Setter Property="FontSize" Value="45"/>
        <Setter Property="FontWeight" Value="Light"/>
```

```
        <Setter Property="Margin" Value="10, 0"/>
        <Setter Property="VerticalAlignment" Value="Center"/>
    </Style>

<DataTemplate x:Key="GroceryListItemTemplate">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Quantity}"
                       Style="{StaticResource GroceryListItem}" Width="50"/>
            <TextBlock Text="{Binding Name}"
                       Style="{StaticResource GroceryListItem}"    Width="200"/>
            <TextBlock Text="{Binding Store}"
                       Style="{StaticResource GroceryListItem}"    Width="300"/>
        </StackPanel>
    </DataTemplate>
</ResourceDictionary>
```

在本书中，我不会深入讲述样式与模板的细节，但我将解释我在这个文件中做了什么，因为它将为后面的代码清单提供一些背景说明。最简单的声明是这一个：

```
...
<SolidColorBrush x:Key="AppBackgroundColor" Color="#3E790A"/>
...
```

Metro 应用程序的默认配色方案是黑底白字，我不喜欢。做出改变的第一步是定义一个不同的颜色，这个声明所做的正是把一个代表暗绿色的值与键名 AppBackgroundColor 关联在一起。稍后你将看到我在创建 XAML 布局时通过颜色键名来使用这种颜色。

接着是声明一个样式，其由多个属性值组成：

```
...
<Style x:Key="GroceryListItem" TargetType="TextBlock"
    BasedOn="{StaticResource BasicTextStyle}" >
    <Setter Property="FontSize" Value="45"/>
    <Setter Property="FontWeight" Value="Light"/>
    <Setter Property="Margin" Value="10, 0"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
</Style>
...
```

这个被称为 GroceryListItem 的样式，定义了数个属性的值：FontSize、FontWeight 等等。但请注意我在声明该样式时所使用的 BasedOn 特性。这允许我继承在另一样式中定义的所有的值。在本例中，我继承微软定义在 StandardStyles.xaml 文件中 BasicTextStyle 样式。在我可以派生新的样式之前，我必须把其他资源字典文件带入作用域，我使用这条声明来达到这个目的：\

```
<ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="/Common/StandardStyles.xaml" />
</ResourceDictionary.MergedDictionaries>
```

你可以按这样的方式随喜好导入多个文件，但导入必须出现在你依据该文件的内容派生样式之前。

最后声明的是一个数据模板，我可以用它定义一组层次化的元素，用来表示数据源中的每个数据项。你可能已经猜到了，我的数据源将是视图模型中采购项的集合。集合中的每个采购项将由一个 StackPanel 控件呈现，其包含三个 TextBlock 控件。注意以粗体标注的两个特性：

```
...
<DataTemplate x:Key="GroceryListItemTemplate">
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Quantity}"
            Style="{StaticResource GroceryListItem}" Width="50"/>
        <TextBlock Text="{Binding Name}"
```

```
                Style="{StaticResource GroceryListItem}"    Width="200"/>
            <TextBlock Text="{Binding Store}"
                Style="{StaticResource GroceryListItem}"    Width="300"/>
        </StackPanel>
</DataTemplate>
...
```

Text 特性的值是重要的。Binding 关键字告诉运行时，我希望 Text 特性的值是从控件的 DataContext 属性中获得。在前一节中，我指定视图模型作为数据源。而指定 Quantity 是告诉运行时我要用数据源对象的 Quantity 属性来显示模板。通过在代码隐藏文件中设置 DataContext 属性，我先指定大局（“使用我的视图模型作为绑定的数据源”），Binding 关键字则让我指定细节（“显示这个特定属性的值”）。当我回到主 XAML 文件时，我就能够连接这两者，那样运行时就知道采用视图模型的哪一部分作为对应的属性值。

我标注的另一个特性不是那么有趣但仍是有用的。对于我指定的 Style 特性，我想要一个名为 GroceryListItem 的 StaticResource（静态资源）。StaticResource 关键字告诉运行时我寻找的资源是已经定义好的。我使用的这个 GroceryListItem 样式就是我在刚才指定的。此处的好处是我可以在一个地方改变我的三个 TextBlock 控件的外观，我可以很容易地为我想要有相似外观的控件派生出新的样式。最后一步是把自定义的资源字典加入到 App.xaml 文件中，使其在应用程序中变得可用，看看我在代码清单 2 - 6 中是怎样做的。

代码清单 2-6 加入自定义资源字典到 App.xaml 文件

```
<Application
    x:Class="MetroGrocer.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Common/StandardStyles.xaml"/>
                <ResourceDictionary Source="Resources/GrocerResourceDictionary.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

说明：
在后面的章节中我并不会列出我在我的资源字典中创建所有的样式，除非样式是与我讲解的特性相关的。你可以从 Apress.com 下载本书的源代码以获得完整的样式列表。

2.2.3. 编写 XAML

现在，我的代码隐藏文件和资源字典都已经就位，我可以转向创建 XAML 文件来声明构成应用程序布局的那些控件。代码清单 2 - 7 显示 ListView.xaml 文件的内容。

代码清单 2-7 ListView.xaml 文件：

```
<Page
    x:Class="MetroGrocer.Pages.ListPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Background="{StaticResource AppBackgroundColor}">
        <Grid.RowDefinitions>
            <RowDefinition/>
```

```
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <StackPanel Grid.RowSpan="2">
            <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10"
                       Text="Grocery List"/>
            <ListView x:Name="groceryList" Grid.RowSpan="2"
                ItemsSource="{Binding GroceryList}"
                ItemTemplate="{StaticResource GroceryListItemTemplate}"
                SelectionChanged="ListSelectionChanged" />
        </StackPanel>
        <StackPanel Orientation="Vertical" Grid.Column="1">
            <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10"
                       Text="Item Detail"/>
        </StackPanel>
        <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
            <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10"
                       Text="Store Detail"/>
        </StackPanel>
    </Grid>
</Page>
```

你可以看到，我设置 Grid（网格）控件的 Background 特性为我在资源字典中指定的颜色，并用 Grid.RowDefinitions 和 Grid.RowDefinitions 元素把 Grid 控件分成了同等大小的两行两列。

对于布局的左侧，我添加了一个设置好的 StackPanel 控件，它将跨两行并填充布局的左半部分：

```
...
<StackPanel Grid.RowSpan="2">
...
```

这个 StackPanel 控件包含一个用来显示标题的 TextBlock 控件以及一个我将很快用到的 ListView 控件。对于屏幕的右边，我加入了一对 StackPanel 控件，每行一个。我已经用 Grid.Row 和 Grid.Column 特性指定了每个 StackPanel 控件属于哪一行哪一列。这些特性使用一个基于零的索引值，未设置这些特性的控件将放置在第一行第一列。您可以在图 2-1 中看到出现在 Visual Studio 设计图面中的布局是怎样的。

图 2-1 显示在 Visual Studio 设计图面上的 XAML 布局的静态部分

设计图面不能够显示动态生成的内容，这就是为什么你在图 2-1 中看不到的视图模型数据的原因。动态内容将显示在一个 ListView 控件中。顾名思义，ListView 控件在一个列表中显示一组数据项，有三个 XAML 特性用于设置来完成这件事：

```
...
<ListView x:Name="groceryList" Grid.RowSpan="2"
      ItemsSource="{Binding GroceryList}"
      ItemTemplate="{StaticResource GroceryListItemTemplate}"
      SelectionChanged="ListSelectionChanged" />
...
```

我使用这些特性来作用大局层面的 DataContext 属性与模板中的细节层面属性间的桥梁。ItemSource 特性告诉 ListView 控件它应该从哪里获得它要显示的数据项。而 Binding 关键字带一个 GroceryList 值则告诉 ListView 控件它应该显示我在代码隐藏文件中设置的 DataContext 对象的 GroceryList 属性的内容。

ItemTemplate 特性告诉 ListView 控件应该如何显示 ItemSource 中的每一数据项。StaticResource 关键字和 GroceryListItemTemplate 值表示将使用我在资源字典中指定的数据模板，意味着对 ViewModel.GroceryList 集合中的每一项都会新生成一个包含有三个 TextBlock 控件的 StackPanel 控件。

最后的 SelectionChanged 特性将我在代码隐藏文件中定义的事件处理器方法与 ListView 控件发送的 SelectionChanged 事件相关联。

提示：

你可以使用 Visual Studio 属性窗口或通过查阅 Microsoft API 文档来获得控件定义的事件列表。创建具有正确参数的事件处理器方法的最简单方式是让 XAML 编辑器来为你创建它们，作为自动完成过程的一部分。

2.3. 运行应用程序

Visual Studio 设计图面只能显示布局的静态内容部分。查看动态内容的唯一方式是运行应用程序。所以，要看到 XAML 和 C#聚合结果的方法就是从 Visual Studio 调试（Debug）菜单中选择开始调试（Start Debugging），Visual Studio 将生成项目并把应用程序推送至模拟器。你可以在图 2 - 2 中看到结果。

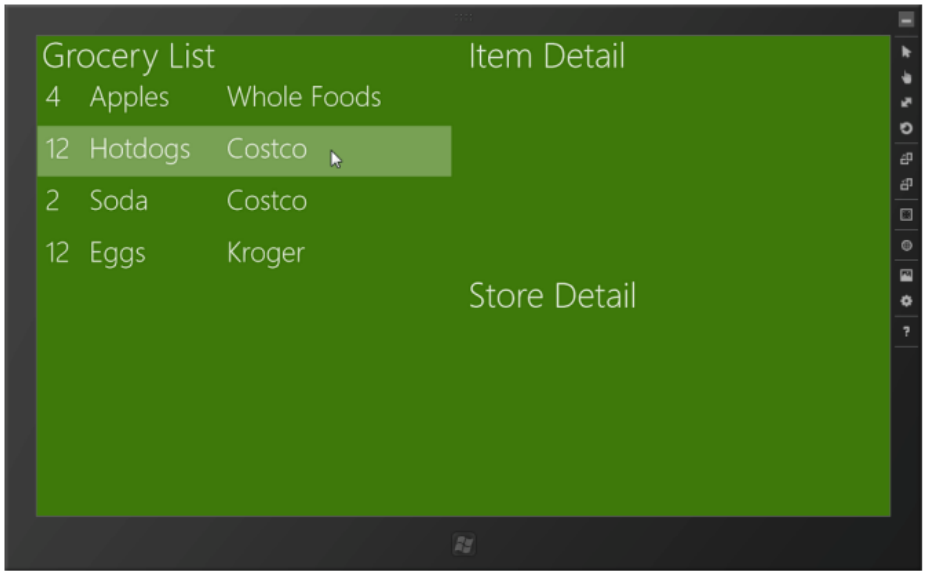

图 2-2 在模拟器中运行示例程序

您可以看到数据绑定是如何将视图模型中的数据项添加到 ListView（列表视图）控件中的，以及我定义在资源字典中的模板中是如何被用于格式化它们的。对于 Metro UI 控件有很多内置的行为。例如，在鼠标移到它们上时会以一个较淡的背景颜色显示（你可以在图 2 - 2 看到），当用户点击以选择某一项时又以不同的背景颜色显示。一个控件使用的所有样式都是可以改变的，但我为简单起见将继续使用默认设置。

2.4. 插入其它页面至布局中

你不必把所有的控件和代码放在单个 XAML 文件及其代码隐藏文件中。要使一个项目易于管理，你可以创建多个页面，然后在应用程序中把它们聚集起来。作为一个简单的演示，我在 Pages 项目文件夹中创建了一个名为 NoItemSelected.xaml 的新页面。代码清单 2-8 显示这个文件的内容。

代码清单 2-8 NoItemSelected.xaml 文件

```
<Page
    x:Class="MetroGrocer.Pages.NoItemSelected"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Background="{StaticResource AppBackgroundColor}" Margin="10">
        <TextBlock Style="{StaticResource HeaderTextStyle}"
                   FontSize="30" Text="No Item Selected"/>
    </Grid>
</Page>
```

这是一个非常简单的页面，你通常不会需要创建这样一个如此简单的页面，因为它仅仅显示一些静态文本。但它对于演示一个重要的 Metro 应用程序特性——允许我将我的应用程序分解成可管理的块是有帮助的。添加页面至我的主应用程序布局的关键是 Frame 控件，我已经将它加入到 ListView.xaml 布局文件中，如代码清单 2-9 所示。

代码清单 2-9 添加一个 Frame 控件至主应用程序布局

```
...
<StackPanel Orientation="Vertical" Grid.Column="1">
    <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10"
               Text="Item Detail"/>
    <Frame x:Name="ItemDetailFrame"/>
</StackPanel>
...
```

Frame（框架）控件是一个 Page 占位符，我在代码隐藏文件中使用 Navigate 方法来告诉 Frame 控件我要它显示哪个页面，如代码清单 2-10 所示。

代码清单 2-10 指定 Frame 控件显示的页面

```
using MetroGrocer.Data;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
namespace MetroGrocer.Pages {
    public sealed partial class ListPage : Page {
        ViewModel viewModel;
        public ListPage() {
            viewModel = new ViewModel();
            // ... test data removed for brevity
            this.InitializeComponent();
            this.DataContext = viewModel;
            ItemDetailFrame.Navigate(typeof(NoItemSelected));

        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
```

```
        private void ListSelectionChanged(object sender, SelectionChangedEventArgs e) {
            viewModel.SelectedItemIndex = groceryList.SelectedIndex;
        }
    }
}
```

Navigate 方法的参数是一个 System.Type 的值，表示你想要加载的 Page（页面）类，获得一个 System.Type 值的最简单方式是用 typeof 关键字。Navigate 方法实例化 Page 对象（记住 XAML 文件与代码隐藏文件组合起来创建一个 Page 的子类），其结果被插入到布局中的，如图 2-3 所示。

图 2-3 插入另一页面至布局

2.5. 动态插入页面至布局

您也可以基于你的应用程序的状态，使用 Frame（框架）控件来动态地插入不同的页面到你的布局之中。为了演示这一点，我在 Pages 文件夹中创建了一个新的 ItemDetail.xaml 页面，其内容显示在代码清单 2 - 11 中。(这个文件依赖我的自定义资源字典中的样式，你可以在本章下载的源代码中看到我是如何定义它们的）。

代码清单 2-11 ItemDetail.xaml 页面：

```
<Page
    x:Class="MetroGrocer.Pages.ItemDetail"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Background="{StaticResource AppBackgroundColor}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Name:" Style="{StaticResource ItemDetailText}" />
        <TextBlock Text="Quantity:" Style="{StaticResource ItemDetailText}"
                   Grid.Row="1"/>
        <TextBlock Text="Store:" Style="{StaticResource ItemDetailText}"
                   Grid.Row="2"/>
        <TextBox x:Name="ItemDetailName"
                 Style="{StaticResource ItemDetailTextBox}"
                 TextChanged="HandleItemChange"
                 Grid.Column="1"/>

<TextBox x:Name="ItemDetailQuantity"
                 Style="{StaticResource ItemDetailTextBox}"
                 TextChanged="HandleItemChange"
                 Grid.Row="1" Grid.Column="1"/>
```

```
<ComboBox x:Name="ItemDetailStore"
                        Style="{StaticResource ItemDetailStore}"
                        Grid.Column="1" Grid.Row="2"
                        ItemsSource="{Binding StoreList}"
                        SelectionChanged="HandleItemChange"
                        DisplayMemberPath="" />
    </Grid>
</Page>
```

这个页面的布局基于一个网格控件，具有文本标签与输入控件（允许用户编辑购物清单中某一采购项的详细信息），其细节我将很快在代码隐藏文件中设置。

提示：

我使用一个 ComboBox 控件来呈现商店列表给用户，该列表由我直接在视图模型中创建。我设置了 ItemSource 特性以使控件绑定到视图模型的 StoreList 属性，但 ComboBox 控件还期望被告知它应该显示所使用对象集合中的哪个属性。因为我的商店列表只是一个字符串数组，我通过设置 DisplayMemberPath 特性为空字符串来解决这个问题。

2.5.1. 切换页面

现在我有了两个页面，我可以在应用程序状态发生变化时切换它们。当用户未从 ListView 选中某项时，我将显示 NoItemSelected 页面；而在用户做出了选择时则显示 ItemDetail 页面。代码清单 2 - 12 显示了为实现此任务而添加到 ListPage.xaml.cs 文件的代码。

代码清单 2-12 基于应用程序状态显示页面

```
...
public ListPage() {
    viewModel = new ViewModel();
// ...test data removed for brevity
    this.InitializeComponent();
    this.DataContext = viewModel;
    ItemDetailFrame.Navigate(typeof(NoItemSelected));
    viewModel.PropertyChanged += (sender, args) => {
        if (args.PropertyName == "SelectedItemIndex") {
            if (viewModel.SelectedItemIndex == -1) {
                ItemDetailFrame.Navigate(typeof(NoItemSelected));
            } else {
                ItemDetailFrame.Navigate(typeof(ItemDetail), viewModel);
            }
        }
    };
}
...
```

默认情况下，我仍显示 NoItemSelected 页面，因为那反映了我的应用程序的初始状态，在应用程序首次启动时用户不可能做出选择。

附加的代码添加了一个 PropertyChanged 事件处理器，该事件由视图模型定义。该事件也被 XAML 控件的数据绑定特性所使用。通过直接注册事件处理器，我能够在我的代码隐藏文件中对视图模型的属性值发生改变时做出反应。在我处理这个事件时，接收到的事件参数通过 PropertyName 属性告诉我哪个属性改变了。如果 SelectedItemIndex 属性的值有改变，那么我使用 Frame.Navigate 方法去显示 NoItemSelected 或 ItemDetail 页面。请注意，当我显示 ItemDetail 页面时，传递视图模型对象作为 Navigate 方法的一个参数：

```
ItemDetailFrame.Navigate(typeof(ItemDetail), viewModel);
```

通过这个参数你可以传递上下文件相关的视图模型对象给要显示的页面。我将在下一节中向你展示如何使用这个对象。

2.5.2. 实现嵌入式页面

当然，仅为我的 ItemDetail 页面定义 XAML 是不够的，我还必须编写代码来显示所选项的详细信息并允许用户进行更改。代码清单 2-13 显示 ItemDetail.xaml.cs 代码隐藏文件，其完成上述任务。

代码清单 2-13 ItemDetail.xaml.cs 代码隐藏文件

```
using System;
using MetroGrocer.Data;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
namespace MetroGrocer.Pages {
    public sealed partial class ItemDetail : Page {
        private ViewModel viewModel;
        public ItemDetail() {
            this.InitializeComponent();
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
            viewModel = e.Parameter as ViewModel;
            this.DataContext = viewModel;
            viewModel.PropertyChanged += (sender, eventArgs) => {
                if (eventArgs.PropertyName == "SelectedItemIndex") {
                    if (viewModel.SelectedItemIndex == -1) {
                        SetItemDetail(null);
                    } else {
                        SetItemDetail(viewModel.GroceryList
                            [viewModel.SelectedItemIndex]);
                    }
                }
            };
            SetItemDetail(viewModel.GroceryList
                [viewModel.SelectedItemIndex]);
        }
        private void SetItemDetail(GroceryItem item) {
            ItemDetailName.Text = (item == null) ? "" : item.Name;
            ItemDetailQuantity.Text = (item == null) ? ""
                : item.Quantity.ToString();
            if (item != null) {
                ItemDetailStore.SelectedItem = item.Store;
            } else {
                ItemDetailStore.SelectedIndex = -1;
            }
        }
        private void HandleItemChange(object sender, RoutedEventArgs e) {
            if (viewModel.SelectedItemIndex > -1) {
                GroceryItem selectedItem = viewModel.GroceryList
                    [viewModel.SelectedItemIndex];
                if (sender == ItemDetailName) {
                    selectedItem.Name = ItemDetailName.Text;
                } else if (sender == ItemDetailQuantity) {
                    int intVal;
                    bool parsed = Int32.TryParse(ItemDetailQuantity.Text,
                        out intVal);
                    if (parsed) {
                        selectedItem.Quantity = intVal;
```

```
                    }
                } else if (sender == ItemDetailStore) {
                    string store = (String)((ComboBox)sender).SelectedItem;
                    if (store != null) {
                        viewModel.GroceryList
                            [viewModel.SelectedItemIndex].Store = store;
                    }
                }
            }
        }
    }
}
```

SetItemDetail 方法设置 UI 控件的内容去显示选中采购项的详细信息，HandleItemChange 方法在用户改变了某一个控件的内容时更新采购项。

这个代码清单中最有意思的部分是 OnNavigatedTo 方法，其用粗体标注。

这个方法在 ItemDetail 页面显示时调用，而我传递给 Frame.Navigate 方法的对象可通过事件参数的 Parameter 属性得到。你可以在图 2 - 4 中看到 ItemDetail 页面在布局中的样子。

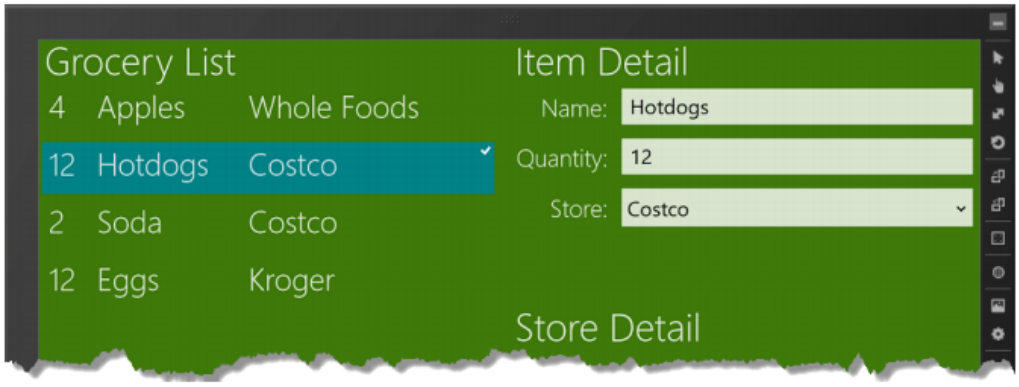

图 2-4 显示 ItemDetail 页面

通过以这样的方式传递视图模型对象，我能确保我的所有页面使用的是同一数据，从而能够将应用程序分解成易管理的若干块。

2.6. 小结

在本章中，我向你展示了如何在一个 Metro 应用中创建视图模型并使其是可观测的，这样你可以使用数据绑定来保持你的应用程序布局与视图模型数据间的同步。这是一项创建健壮的、更易维护的 Metro 应用程序的重要技术。

我还向你展示了如何通过创建页面来把应用程序分解为可管控的块，以及如何使用 Frame （框架）在主布局中显示这些页面。在下一章中，我将向你展示如何创建那些最重要的 Metro 应用程序控件：应用栏（AppBar）、导航栏（NavBar）与弹出框（flyouts）。

第 3 章 应用程序工具栏、弹出画面和导航栏

本章将介绍如何创建和使用一些用户交互的东西，它们是 Metro 用户体验最基础的一部分。应用程序工具栏和导航栏提供了用户可以在应用程序中进行内容、特征和导航的交互功能。在本章，同时也会介绍如何创建弹出画面，并从用户采集信息，该弹出画面通常用于响应应用程序工具栏的按钮事件。表 3-1 提供了本章概要。

表 3-1 章节概要

问题	方案	列表
添加应用程序工具栏	在 XAML 的 Page.BottomAppBar 属性下声明一个应用程序工具栏	1
在应用程序工具栏添加按钮	使用按钮控件，用预定义或者自定义的风格进行格式化	2至5
添加弹出画面	声明一个 Popup 控件，并把属性 IsLightDismissEnabled 设置为 True	6,7
显示弹出画面	定位与Popup相关的应用程序工具栏按钮	8至10
在弹出画面简单的获取视图模型	使用DataContext属性	11至14
添加导航工具栏	创建一个wrapper页面，在XAML的Page.TopAppBar属性声明一个AppBar元素	15,17
在应用程序中导航	在wrapper Page中添加一个适当的控件，当点击NavBar按钮时，使用Navigate方法来显示另一个Page 画面。	16,18

3.1. 添加应用程序工具栏

当用户做一个向上滑动的手势或者用点击鼠标右键，应用程序工具栏会出现在屏幕的底部，Metro UI 强调在主布局上尽量少出现控件，主要依靠应用程序工具栏，并且任何交互方式并不是立即就可以使用，但是在当前布局中交互是存在的。在本节中，将介绍如何定义和使用应用程序工具栏。

提示： 在屏幕的顶部，有相似的控件，叫导航工具栏，用于 Metro 应用中不同画面之间的导航。在本章中，也将会介绍如何创建和使用导航工具栏。

3.1.1. 声明应用程序工具栏

创建应用程序工具栏最简单的方法是在 XAML 文件中声明。列表 3-1 显示了在例子工程里面添加的 ListPage.xaml 文件。

列表 3-1. 定义一个应用程序工具栏

```
<Page
x:Class="MetroGrocer.Pages.ListPage" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:local="using:MetroGrocer.Pages"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006" mc:Ignorable="d">
  - <Grid Background="{StaticResource AppBackgroundColor}">
    - <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
```

```
    - <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
    - <StackPanel Grid.RowSpan="2">
        <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10" Text="Grocery List" />
        <ListView x:Name="groceryList" Grid.RowSpan="2" ItemsSource="{Binding GroceryList}"
ItemTemplate="{StaticResource GroceryListItemTemplate}" SelectionChanged="ListSelectionChanged" />
      </StackPanel>
    - <StackPanel Orientation="Vertical" Grid.Column="1">
        <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10" Text="Item Detail" />
        <Frame x:Name="ItemDetailFrame" />
      </StackPanel>
    - <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
        <TextBlock Style="{StaticResource HeaderTextStyle}" Margin="10" Text="Store Detail" />
      </StackPanel>
  </Grid>
  - <Page.BottomAppBar>
    - <AppBar>
      - <Grid>
        - <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
          </Grid.ColumnDefinitions>
        - <StackPanel Orientation="Horizontal" Grid.Column="0" HorizontalAlignment="Left">
            <Button x:Name="AppBarDoneButton" Style="{StaticResource DoneAppBarButtonStyle}" IsEnabled="false"
Click="AppBarButtonClick" />
          </StackPanel>
        - <StackPanel Orientation="Horizontal" Grid.Column="1" HorizontalAlignment="Right">
            <Button x:Name="AppBarAddButton" Style="{StaticResource AddAppBarButtonStyle}"
AutomationProperties.Name="New Item" Click="AppBarButtonClick" />
            <Button x:Name="AppBarStoresButton" Style="{StaticResource StoresAppBarButton}" Click="AppBarButtonClick" />
            <Button x:Name="AppBarZipButton" Style="{StaticResource HomeAppBarButtonStyle}"
AutomationProperties.Name="Zip Code" Click="AppBarButtonClick" />
          </StackPanel>
        </Grid>
      </AppBar>
  </Page.BottomAppBar>
</Page>
```

为了创建应用程序工具栏，上面的列表中，我在 Page.BottomAppBar 属性中声明了 AppBar 控件。上述代码效果是创建 AppBar、AppBar 的内容和把他们赋给页面的 BottomAppBar 属性。

提示：在 Page.TopAppBar 属性里可以通过声明 AppBar 控件来创建 NavBar。

应用程序工具栏包含按钮，规定选中的项显示在工具栏的左边，而 app-wide 按钮显示在右边。（Windows 8 消费者预览版的用户体验准则还没有全部完成，因此可能在发行版中改变这一规定。）

为了遵守这个规定，我在 AppBar 控件里添加了一个一行两列的网格(Grid)，每列包含一个栈面板(StackPanel)，这里有两种方法可以在 AppBar 中添加按钮：在已经声明好的 StandardStyles.xaml 文件里面，可以选择适合的一个，或者自己创建一个。下面的小节中，两种方法都被使用到。

3.1.2. 使用预定义的应用程序栏按钮

大部分 StandardStyles.xaml 文件都为应用程序工具栏提供了按钮控件，比如列表 3-2 显示的内容。

列表 3-2. 添加应用程序工具栏按钮的风格

```
...
<Style x:Key="AddAppBarButtonStyle" TargetType="Button"
```

```
BasedOn="{StaticResource AppBarButtonStyle}">
   <Setter Property="AutomationProperties.AutomationId" Value="AddAppBarButton"/>
   <Setter Property="AutomationProperties.Name" Value="Add"/>
   <Setter Property="Content" Value="&#xE109;"/>
</Style>
...
```

所有预定义的按钮风格都是继承自 AppBarButtonStyle，AppBarButtonStyle 定义了应用程序工具栏按钮的基本特征。在下一节中创建自定义按钮也会继承自这个风格。

提示：在文件 StandardStyles.xaml 中已经定义了 29 中应用程序按钮风格，但是来自 XAML 小组的微软项目经理提供了非正式的替代文件，该文件定义了 150 种不同的风格。该文件可以在下面的连接中看到：
http://timheuer.com/blog/archive/2012/03/05/visualizing-appbar-command-styles-windows-8.aspx

区分不同按钮的两个属性是 AutomationProperties.Name 和 Content。AutomationProperties.Name 属性指定了按钮显示的文本，Content 属性指定会被用到的 icon。这个属性(Content)的值来自 Segoe UI Symbol 字体的编码。可以通过 Windows 8 提供的字符表(Character Map)工具来查看这种字体定义的 icon。值 E109 对应的是一个加号。

在列表中显示的风格并不是我想要的，我喜欢 icon，但是我想改用文本。针对我的需求，我简单的使用了预定义风格，并重写了我想改变的部分，如列表 3-3.

列表 3-3.使用预定义的应用程序工具栏按钮

```
...
   <Button x:Name="AppBarAddButton"
   Style="{StaticResource AddAppBarButtonStyle}"
   AutomationProperties.Name="New Item"
   Click="AppBarButtonClick"/>
```

3.1.3. 创建自定义的应用程序工具栏按钮风格

另外一种可选的方法是为你的AppBar按钮定义自己的风格。列表3-4显示了我使用的一个风格。

列表3-4.定制一个AppBar按钮风格

```
...
<Style x:Key="StoresAppBarButton" TargetType="Button"
BasedOn="{StaticResource AppBarButtonStyle}">
   <Setter Property="AutomationProperties.Name" Value="Stores"/>
   <Setter Property="Content" Value="&#xE14D;"/>
</Style>
...
```

我的这个风格是继承自AppBarButtonStyle的,所以它具有AppBar按钮基本的外观，并且我为属性AutomationProperties.Name和Content设置了值。你可以进一步的重定义一些内部风格的核心特征，但是应用的用户体验可能会远离标准的Metro外观和体验。图3-1中可以看到添加的应用程序工具栏和按钮控件。如果你想在设备上看到应用程序工具栏，请启动示例程序，并从屏幕的顶部或底部滑动，或单击鼠标右键。

图3-1.在Metro示例程序中添加应用程序工具栏

3.1.4. 实现应用程序工具栏按钮的事件

目前，应用程序工具栏的按钮控件还不能做任何动作。为了响应事件，接下来将会实现 Done 按钮的动作，在图 3-1 中 Done 按钮是无效的。

当用户在 grocery 列表项中做了选择，Done 按钮将被激活。当用户点击 Done 按钮，将会从列表中移除选中的项，可以表示用户已经购买了一项。列表 3-5 显示了代码隐藏文件 ListView.xaml.cs 的改变。

列表 3-5. 实现 Done 按钮

```
using MetroGrocer.Data;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
namespace MetroGrocer.Pages
{
    public sealed partial class ListPage : Page
    {
        ViewModel viewModel;
        public ListPage()
        {
            viewModel = new ViewModel();
            // ...test data removed for brevity
            this.InitializeComponent();
            this.DataContext = viewModel;
            ItemDetailFrame.Navigate(typeof(NoItemSelected));
            viewModel.PropertyChanged += (sender, args) =>
            {
                if (args.PropertyName == "SelectedItemIndex")
                {
                    if (viewModel.SelectedItemIndex == -1)
                    {
                        ItemDetailFrame.Navigate(typeof(NoItemSelected));
                        AppBarDoneButton.IsEnabled = false;
                    }
                    else
                    {
                        ItemDetailFrame.Navigate(typeof(ItemDetail), viewModel);
                        AppBarDoneButton.IsEnabled = true;
                    }
                }
            };
        }
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }
        private void ListSelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            viewModel.SelectedItemIndex = groceryList.SelectedIndex;
        }
        private void AppBarButtonClick(object sender, RoutedEventArgs e)
        {
            if (e.OriginalSource == AppBarDoneButton
            && viewModel.SelectedItemIndex > -1)
            {
                viewModel.GroceryList.RemoveAt(viewModel.SelectedItemIndex);
                viewModel.SelectedItemIndex = -1;
            }
```

```
        }
    }
}
```

上面的代码中，有两个点需要注意。第一点，就是在应用程序工具栏里实现按钮的动作，也就是响应点击事件。

第二点，你可以在代码里面看到视图模型的优势。在代码里面的 AppBarButtonClick 方法不需要把视图内容切换到 NoItemSelected 页面，当选择项项完成后，也不需要把 Done 按钮失效。只需要更新一下视图模型即可，and the rest of the app adapts to those changes to present the userwith the right layout and overall experience.（不知所云）

3.2. 创建弹出画面

Done 按钮已经有一个动作，这个动作可以可以在代码里直接执行点击事件。然而大多数的工具栏按钮需要别的一些用户交互，在这里使用了弹出画面。

弹出画面是一个 pop-up 窗口，显示在被点击按钮附近，当用户点击或者触摸屏幕的非弹出画面区域，弹出画面会自动的消失。在 JavaScript Metro 应用中是具有 Flyout(弹出画面)控件的，但用 XAML 和 C#想要得到相同的效果，需要使用 Popup 控件，以及需要自己编写弹出画面位置的代码。在 Windows 8 发行版中，我希望微软在 C#和 Javascript 之间提供的内容是对等的，包括一个 XAML Flyout 控件，因为用代码准确的定位 Popup 的位置是非常麻烦的，下面你马上会感受到。

3.2.1. 创建用户控件

XAML 文件会很快的变得很长并且难以管理。我喜欢把弹出画面定义为用户控件，就像一段 XAML 元素，以及一个后端代码文件。（在这里我跳过了 XAML 的一些细节，但是当你在读本节会名表我的意思）。在示例工程里面，我创建了一个叫 Flyouts 的文件夹，UserControl 模板被用来创建一个叫 HomeZipCodeFlyout.xaml 的新项，在列表 3-6 中会看到里面的内容。

列表3-6. HomeZipCodeFlyout.xaml文件

```
<UserControl
x:Class="MetroGrocer.Flyouts.HomeZipCodeFlyout"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MetroGrocer.Flyouts"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d"
d:DesignHeight="130"
d:DesignWidth="350">
  <Popup x:Name="HomeZipCodePopup"
  IsLightDismissEnabled="True" Width="350" Height="130" >
    <StackPanel Background="Black">
      <Border Background="#85C54C" BorderThickness="4">
        <StackPanel>
          <StackPanel Orientation="Horizontal" Margin="10">
            <TextBlock Style="{StaticResource PopupTextStyle}"
            Text="Home Zip Code:" VerticalAlignment="Center"
            Margin="0,0,10,0" />
            <TextBox Height="40" Width="150" FontSize="20"
                    Text="{Binding Path=HomeZipCode, Mode=TwoWay}" />
          </StackPanel>
          <Button Click="OKButtonClick" HorizontalAlignment="Center"
          Margin="10">OK</Button>
        </StackPanel>
      </Border>
    </StackPanel>
```

```
    </Popup>
</UserControl>
```

就如文件名一样，这个 flyout 允许用户改变在视图模型中 HomeZipCode 属性的值。在例子中，这个属性除了提供一些有用的例子外，不做任何事情。

用户控件的作用跟模板控件一样。在 UserControl 元素里面，定义的 XAML 元素表示你想创建的控件。在创建弹出画面时，必须使用 Popup 控件，在 Popup 里面放置的内容取决于需求。我定义的控件布局由下面内容构成：一个用于从用户那里收集新数据的 TextBox，一个按钮，当用户输入了新的值，可以点击这个按钮，以及周围的一些元素，用来当做提示信息。

在 Popup 元素里有 3 个重要的属性需要设置，每一个我都在列表中加粗。IsLightDismissEnabled 属性表示当用户点击或者触摸屏幕的任意位置，不包括 Popup 的位置，pop-up 画面是否消失。当把 Popup 用作弹出画面时，这个属性必须设置为 True，因为这是基本的弹出画面(flyout）用户体验。

Width 和 Height 属性也必须设置，才能适合布局里面包含的内容，当在定位 Popup 时，我需要确定这些属性的值，下面即将做演示。

警告：如果你使用我的定位代码(下面我即将描述的)来管理你的 flyouts，必须明确的提供 Width 和 Height 值。如果这些值被你忽略或者提供的是错误的，那么 flyout 的位置将会是错误的。

3.2.2. 编写用户控件代码

尽管这里的用户控件只是 XAML 的一部分，用户控件仍然具有隐藏的代码文件。

列表 3-7 显示了 HomeZipCodeFlyout.xaml.cs 文件的内容。

列表 3-7 HomeZipCodeFlyout.xaml.cs 文件：

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer.Flyouts
{
    public sealed partial class HomeZipCodeFlyout : UserControl
    {
        public HomeZipCodeFlyout()
        {
            this.InitializeComponent();
        }
        public void Show(Page page, AppBar appbar, Button button)
        {
            HomeZipCodePopup.IsOpen = true;
            FlyoutHelper.ShowRelativeToAppBar(HomeZipCodePopup, page, appbar, button);
        }
        private void OKButtonClick(object sender, RoutedEventArgs e)
        {
            HomeZipCodePopup.IsOpen = false;
        }
    }
}
```

我写的这个弹出画面主要的问题是必须定位 Popup 控件的位置。约定响应按钮的弹出画面显示在被点击按钮上方。

3.2.3. 定位弹出的控件

Metro 控件没有提供简单的方法来计算出布局中元素的相关位置，所以需要使用一些间接的方法。列表 3-8 显示了 FlyoutHelper 类的内容，这个类被添加在 Flyouts 文件夹中，它定义了一个静态方法

ShowRelativeToAppBar。这个方法计算出相关按钮上显示的 Popup 的正确位置，这样做，需要传入 Popup 控件、包含 AppBar 的 Page、AppBar 控件和被点击的 button 按钮。这个方法并不好，但这是我发现的可以获得 flyout 准确位置的唯一方法。

列表3-8. 定位与工具栏按钮相关的Pop-up的位置

```
using System;
using Windows.Foundation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
namespace MetroGrocer.Flyouts
{
    public class FlyoutHelper
    {
        public static void ShowRelativeToAppBar(Popup popup, Page page,
            AppBar appbar, Button button)
        {
            Func<UIElement, UIElement, Point> getOffset =
            delegate(UIElement control1, UIElement control2)
            {
                return control1.TransformToVisual(control2)
                .TransformPoint(new Point(0, 0));
            };
            Point popupOffset = getOffset(popup, page);
            Point buttonOffset = getOffset(button, page);
            popup.HorizontalOffset = buttonOffset.X - popupOffset.X
            - (popup.ActualWidth / 2) + (button.ActualWidth / 2);
            popup.VerticalOffset = getOffset(appbar, page).Y
            - popupOffset.Y - popup.ActualHeight;
            if (popupOffset.X + popup.HorizontalOffset
            + popup.ActualWidth > page.ActualWidth)
            {
                popup.HorizontalOffset = page.ActualWidth
                - popupOffset.X - popup.ActualWidth;
            }
            else if (popup.HorizontalOffset + popupOffset.X < 0)
            {
                popup.HorizontalOffset = -popupOffset.X;
            }
        }
    }
}
```

代码会将 Popup 定位在与其相关的 AppBar 按钮的上方，如果 Popup 从屏幕的左侧或右侧消失那么 Popup 将被重新定位，我不打算介绍这段代码的细节，因为它太绕了，我希望在 Windows 8 最终版发布的时候，不需要这样做。如果你遇到相关问题，大多数原因是没有设置 Popup 的 Width 和 Height 属性。

3. 2. 4. 显示和隐藏 Popup 控件

HomeZipCodeFlyout 类的另外一个功能是负责显示和隐藏 Popup 控件。在这个弹出画面，我用了简洁的代码。如果你回头看列表 3-6 中的 XAML，会看见我已经制定了数据绑定(dinding)的方式，如下：

```
...
<TextBox Height="40" Width="150" Text="{Binding Path=HomeZipCode, Mode=TwoWay}" />
...
```

数据绑定的默认模式是 one-way，意思是利用视图模型的改变来更新控件的显示。而我指定的模式是 two-

way，在 one-way 作用之上，增加一种方法：用户在 TextBox 控件输入的值会被用来更新视图模型中对应的属性。

提示：注意，在绑定过程中，没有必要设置 DataContext。因为用户控件会被添加到主 XAML 布局中，这就意味着，用户控件来自顶级（top-level）页面对象中 DataContext 的值。

通过上面的数据绑定，我在处理 OK 按钮的 click 事件时，只需要简单的隐藏 Popup 控件即可；没必要担心从 TextBox 取得值和明确的更新视图模型。

这种方法的负面影响是在弹出画面消失之前，视图模型可能会被多次更新，如果在程序在别处监听了相应的属性的值改变事件，会出问题。在此，对于 HomeZipCode 属性是没有问题的，我只是想展示如何用非常简单的方法处理用户输入的技术。

3.2.5. 在应用程序中添加弹出画面

我之所以如此麻烦的创建用户控件，是因为想让主布局中的 XAML 尽量集中。我需要将用户控件添加到 XAML 中，因此，下面的列表 3-9 你将看到我是如何做到的。

Listing 3-9. 在主布局的 XAML 中添加一个弹出画面

```
<Page
x:Class="MetroGrocer.Pages.ListPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MetroGrocer.Pages"
xmlns:flyouts="using:MetroGrocer.Flyouts"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
  <Grid Background="{StaticResource AppBackgroundColor}">
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.RowSpan="2">
      // ...contents removed for brevity
    </StackPanel>
    <StackPanel Orientation="Vertical" Grid.Column="1">
      // ...contents removed for brevity
    </StackPanel>
    <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
      // ...contents removed for brevity
    </StackPanel>
    <flyouts:HomeZipCodeFlyout x:Name="HomeZipFlyout"/>
  </Grid>
  <Page.BottomAppBar>
    // ...contents removed for brevity
  </Page.BottomAppBar>
</Page>
```

我必须定义一个新的 XAML 名称空间，才能够使用 Flayouts 文件夹中的用户控件，所以，我把下面的一行代码添加到了 XAML 中：

xmlns:flyouts="using:MetroGrocer.Flyouts"

此行代码重要的部分是名为 flayouts，即赋值给 xmlns 后面的那部分。在我声明用户控件的时候，我也必须使用同样的名字，如下：

<flyouts:HomeZipCodeFlyout x:Name="HomeZipFlyout"/>

注意，flyout 声明在 Grid 里面；即使 flyout 不是立即显示，弹出用户控件也必须声明为主程序布局的一部分，Page 控件可以只具有合法的子元素(这也是为什么 AppBar 控件必须声明在 Page.BottomAppBar 属性里面)。

3.2.6. 显示弹出画面

所有的工作，只剩下当用户点击应用程序工具栏按钮时，触发我的弹出画面显示了。列表 3-10 显示了添加到 ListPage.xaml.cs 文件的代码，这段代码会完成触发任务。

Listing 3-10. 当应用程序工具栏按钮被点击时，响应并显示弹出画面

```
...
private void AppBarButtonClick(object sender, RoutedEventArgs e)
{
    if (e.OriginalSource == AppBarDoneButton
    && viewModel.SelectedItemIndex > -1)
    {
        viewModel.GroceryList.RemoveAt(viewModel.SelectedItemIndex);
        viewModel.SelectedItemIndex = -1;
    }
    else if (e.OriginalSource == AppBarZipButton)
    {
        HomeZipFlyout.Show(this, this.BottomAppBar, (Button)e.OriginalSource);
    }
}
...
```

我调用了定义在用户控件里面的 Show 方法，并传了一组控件进去，在准确定位 Popup 的时候用到。你看到的结果如图 3-2.

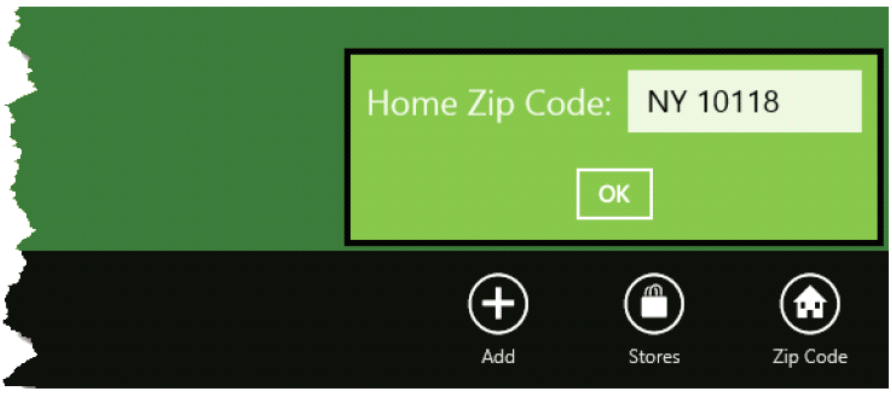

图 3-2.在与应用程序工具栏按钮相关的附近显示一个弹出画面

3.2.7. 创建一个更复杂的弹出画面

目前，我已经演示了基本的弹出画面，可以弹出一个弹出画面，允许用户添加新项到 grocery 列表中。然而上面的演示并没有使用 two-way 绑定方法来处理视图模型。这并不是一项特别复杂的技术；我想在这里把两种方法都介绍一下，你可以在你的项目中选择一个。我将介绍更多的弹出画面例子，当你亲自写的时候，会发现很简单。

我使用 UserControl 模板，在 Flyouts 工程文件中创建 AddItemFlyout.xaml 文件。I then followed the same basic approach of laying out my content in a Popup, as shown in Listing 3-11.

Listing 3-11. 在 XAML 中添加 Additem 弹出画面

```
<UserControl
x:Class="MetroGrocer.Flyouts.AddItemFlyout"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MetroGrocer.Flyouts"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d"
d:DesignHeight="265"
d:DesignWidth="435">
   <Popup x:Name="AddItemPopup" IsLightDismissEnabled="True" Width="435" Height="265" >
      <StackPanel Background="Black">
         <Border Background="#85C54C" BorderThickness="4">
            <Grid Margin="10">
               <Grid.RowDefinitions>
                  <RowDefinition/>
                  <RowDefinition/>
                  <RowDefinition/>
                  <RowDefinition/>
               </Grid.RowDefinitions>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto"/>
                  <ColumnDefinition Width="300"/>
               </Grid.ColumnDefinitions>
               <TextBlock Text="Name:" Style="{StaticResource AddItemText}" />
               <TextBlock Text="Quantity:" Grid.Row="1"
               Style="{StaticResource AddItemText}" />
               <TextBlock Text="Store:" Grid.Row="2"
               Style="{StaticResource AddItemText}" />
               <TextBox x:Name="ItemName" Grid.Column="1"
               Style="{StaticResource AddItemTextBox}" />
               <TextBox x:Name="ItemQuantity" Grid.Row="1" Grid.Column="1"
               Style="{StaticResource AddItemTextBox}" />
               <ComboBox x:Name="ItemStore" Grid.Column="1" Grid.Row="2"
               Style="{StaticResource AddItemStore}"
               ItemsSource="{Binding StoreList}"
               DisplayMemberPath="" />
               <StackPanel Orientation="Horizontal" Grid.Row="3"
               HorizontalAlignment="Center"
               Grid.ColumnSpan="2">
                  <Button Click="AddButtonClick">Add Item</Button>
               </StackPanel>
            </Grid>
         </Border>
      </StackPanel>
   </Popup>
</UserControl>
```

Popup 的布局与我在第二章中创建的 ItemDetail page 非常相似。It is possible to embed Frame controls (and therefore Pages) into Popups for flyouts,

but the effort required to adjust the styling and change the code-behind behavior often makes it more attractive to simply duplicate the elements. I am happy to do this for simple projects, even though I have a nagging feeling that I will be revisiting the project at some point to remove the duplication and do it properly.

3.2.8. 编写代码

我想让你通过看列表 3-12 的代码来认识这部分的弹出画面。

Listing 3-12. AddItemFlyout.xaml.cs 文件

```
using System;
using MetroGrocer.Data;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer.Flyouts
{
    public sealed partial class AddItemFlyout : UserControl
    {
        public AddItemFlyout()
        {
            this.InitializeComponent();
        }
        public void Show(Page page, AppBar appbar, Button button)
        {
            AddItemPopup.IsOpen = true;
            FlyoutHelper.ShowRelativeToAppBar(AddItemPopup, page, appbar, button);
        }
        private void AddButtonClick(object sender, RoutedEventArgs e)
        {
            ((ViewModel)DataContext).GroceryList.Add(new GroceryItem
            {
                Name = ItemName.Text,
                Quantity = Int32.Parse(ItemQuantity.Text),
                Store = ItemStore.SelectedItem.ToString()
            });
            AddItemPopup.IsOpen = false;
        }
    }
}
```

我需要获取到视图模型对象，才能给它添加新项。有多种方法可以做到，最简单的方法就是列表 3-12 可以看到的，读取 DataContext 属性的值。有些地方需要注意，在此，我假定属性的值是视图模型对象，这个视图模型对象我在 ListPage 类（在 ListPage.xaml.cs 文件中）中构建。正如我前面提到的，DataContext 属性是被继承的，但是需要确保是的在布局层次中，中间控件没有赋值给 DataContext 属性。

一旦得到了 ViewModel 对象，往 GroceryList 集合添加一个新的 GroceryItem 对象就是非常简单了。因为 GroceryList 集合是一个观察者，添加的新 GroceryItem 项会被自动的反映到应用程序的其他地方。

3.2.9. 在应用程序中添加弹出画面

剩下的工作就是把心的弹出画面添加到 ListPage 布局和代码中，与之前的弹出画面做类似的事情即可。列表 3-13 展示了布局中 XAML 的声明。

Listing 3-13. 在 XAML 中声明 Add Item Flyout

```
...
<flyouts:HomeZipCodeFlyout x:Name="HomeZipFlyout"/>
<flyouts:AddItemFlyout x:Name="AddItemFlyout"/>
...
```

列表 3-14 显示了在 ListPage 类中方法 AppBarButtonClick 增加的部分，即当 Add Item 应用程序工具栏按钮被点击时，显示出弹出画面。

Listing 3-14. 显示弹出画面以此响应应用程序工具栏按钮的点击

```
...
private void AppBarButtonClick(object sender, RoutedEventArgs e)
{
    if (e.OriginalSource == AppBarDoneButton
    && viewModel.SelectedItemIndex > -1)
    {
        viewModel.GroceryList.RemoveAt(viewModel.SelectedItemIndex);
        viewModel.SelectedItemIndex = -1;
    }
    else if (e.OriginalSource == AppBarZipButton)
    {
        HomeZipFlyout.Show(this, this.BottomAppBar, (Button)e.OriginalSource);
    }
    else if (e.OriginalSource == AppBarAddButton)
    {
        AddItemFlyout.Show(this, this.BottomAppBar, (Button)e.OriginalSource);
    }
}
```

在图 3-3 中你可以看到弹出画面是如何出现的。与应用程序工具栏相关，Popup 控件的 light dismiss 风格表示，同时只显示一个弹出画面。

图 3-3 添加项的弹出画面

3. 3. 在 Metro 应用程序内进行导航

如果你在你的程序中包含不同的功能点，那么你需要提供一个导航栏，让用户可以方便的在不同功能点之间切换。最简单的方法就是重构 Metro 应用，达到提供一致的导航，在功能区域显示一个 Frame 控件，并将该控件放到一个经封装的 Page 页面中。

3. 3. 1. 进行封装

为了进行封装，我在 Pages 文件夹中使用 Blank Page 模板创建了一个 MainPage.xaml 文件。在列表 3-15 中，你可以看到这个文件的内容。

列表 3-15. MainPage.xaml 文件

```
<Page
x:Class="MetroGrocer.Pages.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
xmlns:local="using:MetroGrocer.Pages"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
<Page.TopAppBar>
<AppBar>
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
<ToggleButton x:Name="ListViewButton"
Style="{StaticResource ToggleAppBarButtonStyle}"
AutomationProperties.Name="List View" IsChecked="True"
Content="&#xE14C;" Click="NavBarButtonPress"/>
<ToggleButton x:Name="DetailViewButton"
Style="{StaticResource ToggleAppBarButtonStyle}"
AutomationProperties.Name="Detail View"
Content="&#xE1A3;" Click="NavBarButtonPress"/>
</StackPanel>
</AppBar>
</Page.TopAppBar>
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
<Frame x:Name="MainFrame" />
</Grid>
</Page>
```

为了添加一个 NavBar，我在 Page.TopAppBar 属性里面声明了一个 AppBar 控件。NavBar 的机制与 AppBar（底部）类似，但是在这里我使用了 ToggleButton 控件，因为在这里我想向你展示在程序里面支持的两个视图。

除了 NavBar，MainPage 的布局中还包含一个 Frame，我将用它来显示不同的视图。

提示：在 Windows 8 Consumer Preview 中，没有把 ToggleButton 控件放到 AppBar 上面的内置风格。这里为了演示效果，我使用了 Tim Heuer 定义的 ToggleAppBarButtonStyle，将其添加到了 grocerResourceDictionary.xaml 文件中了。风格内容比较长，你可以通过这个链接看到 Tim 对其的描述 http://timheuer.com/blog/archive/2012/03/19/creating-a-metrostyle-toggle-button-for-appbar.aspx，当然你也可以从本书的合作伙伴处下载相应的源码进行查看。

关于此布局部分，代码方面的编写相对比较简单，如列表 3-16 所示。我通过导航到相应的 page 页面，以响应 ToggleButton 控件的单击事件，并且还会改变按钮的 IsChecked 属性。同时，我也在该类中创建了 ViewModel 对象，这样可以确保整个程序中只有一个 ViewModel 实例对象，通过 Frame.Navigate 方法，将该对象传递给不同的页面。

列表 3-16 Mainpage.xaml.cs

```
using System;
using MetroGrocer.Data;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Navigation;

namespace MetroGrocer.Pages {
public sealed partial class MainPage : Page {
private ViewModel viewModel;
public MainPage() {
this.InitializeComponent();
viewModel = new ViewModel();
// ...test data removed for brevity
this.DataContext = viewModel;
MainFrame.Navigate(typeof(ListPage), viewModel);
}
protected override void OnNavigatedTo(NavigationEventArgs e) {
}
private void NavBarButtonPress(object sender, RoutedEventArgs e) {
```

```
Boolean isListView = (ToggleButton)sender == ListViewButton;
MainFrame.Navigate(isListView ? typeof(ListPage)
: typeof(DetailPage), viewModel);
ListViewButton.IsChecked = isListView;
DetailViewButton.IsChecked = !isListView;
}
}
}
```

在程序的构造函数中，我首先导航到缺省页面：Listpage，在之前的例子中，我有使用过。

提示 为了获取从 OnNavigatedTo 方法中的参数获取视图模型，我已经把 ListPage 进行了重构。由于改动比较小，这里就不把代码列举出来了，你可以从 Apress.com 下载本书的源码，来查看我修改的地方。

我需要更新一下 App.xaml.cs 文件，把我封装的 view 放进去。如列表 3-17 所示。

列表 3-17 将 MainPage 设置为默认页面

```
...
protected override void OnLaunched(LaunchActivatedEventArgs args) {
if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) {
//TODO: Load state from previously suspended application
}
// Create a Frame to act navigation context and navigate to the first page
var rootFrame = new Frame();
rootFrame.Navigate(typeof(Pages.MainPage));
// Place the frame in the current Window and ensure that it is active
Window.Current.Content = rootFrame;
Window.Current.Activate();
}
...
```

3.3.2. 创建其他视图

为了完成这个示例，我需要添加另外的一个 Page 到程序中。我在 Pages 文件夹中创建了 DetailPage.xaml；它的布局如 3-18 列表所示：

列表 3-18. DetailPage 布局

```
<Page
x:Class="MetroGrocer.Pages.DetailPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MetroGrocer.Pages"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
<StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
<TextBlock Style="{StaticResource HeaderTextStyle}" Text="Detail View"/>
</StackPanel>
</Grid>
</Page>
```

这个页面不包含任何功能；在这里只是为了展示 Metro App 中导航的处理。

3.3.3. 测试导航

剩下的任务就是测试导航了。如果你启动程序，并显示 AppBar，你会看到 NavBar 会自动的显示出来。如

图 3-4 所示。

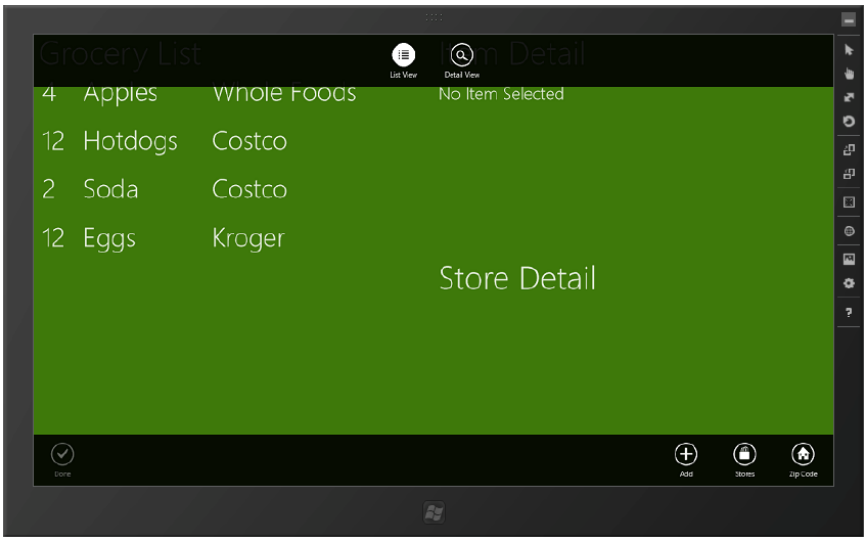

图 3-4 显示（全局）NavBar，以及相关的页面的 AppBar

3.4. 总结

本章中，我展示了如何创建 AppBar，NavBar 和 flyout，它们提供了 Metro 用户体验的必要内容。实现这些交互，是非常重要的，可以使应用程序具有一致的、广泛的用户体验，在这里我建议你花一些时间确保显示的控件总是与内容相关。在下一章中，我会介绍一些特性：瓷贴和徽章。

第 4 章 布局和磁贴

在这一章,我描述由 Windows 8 经验提出的两个特性,可以使 Metro 应用适应广大的用户。第一个特性是 Metro 应用可以被设置为 snapped 或者 filled，以便使这样两个应用程序可以并排浏览。当你的应用程序需要放置在这样的布局我会向您展示如何修改使你的应用程序适应这样的布局和当你的交互不适应布局约束时如何改变布局。

第二个功能是 Metro 磁贴模型。磁贴在 Windows 8 的核心替代是开始菜单。最简单的,它们是静态的按钮,可以用来启动您的应用程序,但只要做一点工作就可以呈现你的应用程序的宝贵快照给用户，允许用户无需运行应用程序本身获得概览。在这一章,我将向您展示如何创建动态磁贴应用更新并利用相关特性、徽章。表 4 - 1 提供对于这一章的总结。

警告：你需要卸载前一章的示例 Metro 应用如果你使用和从 Appress.com 下载的代码例子一样的代码。在模拟器的开始菜单右键点击 MetroGrocer 磁贴并且选择卸载。如果你运行一个从不同项目路径的应用，Visual Studio 会报错。当你从一章移动到另外一章时你必须卸载这个应用。

表 4-1 本章总结

问题	解决方案	Listing
调整一个应用程序的布局当其被放置在一个 snapped 或 filled 布局	通过修改你的控件的布局处理 ViewStateChanged 事件	1 , 2
使用 XAML 声明布局改变所需	使用 VisualStateManage.	3 , 4
打破 snapped 视图	使用 TryUnsnap 方法	5 , 6
为应用创建动态磁贴	修改一个 XML 模板的内容,并使用 Windows.UI.Notification 命名空间中的类。	7 , 8
更新方磁贴和宽磁贴	准备更新两个模板并将它们合并一起	9
在磁贴上使用 badge(徽章)	填充并应用一个 XML 徽章模板	10 , 11

4.1. 支持 Metro 布局

到目前为止,我的应用程序假定它有使用屏幕的完全专用权。然而 Metro 应用通过安排编号,以便他们是 snapped 或 filled 视图。一个 snapped 模式的应用程序在屏幕左边或右边的边缘占用了一个 320 像素的地带。filled 模式的应用程序靠在 snapped 模式的应用程序的旁边并且占据了除了 320 像素地带外剩下的屏幕。演示不同的布局,我添加了一些内容到 DetailPage.xaml 文件,如 Listing4 - 1 所示。

Listing 4-1. Adding Content to the DetailPage.xaml File

```
<Page
    x:Class="MetroGrocer.Pages.DetailPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid x:Name="GridLayout" Background="#71C524">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
```

```
        <StackPanel x:Name="TopLeft" Background="#3E790A">
            <TextBlock x:Name="TopLeftText"
                       Style="{StaticResource DetailViewLabelStyle}"
                       Text="Top-Left"/>
        </StackPanel>
        <StackPanel x:Name="TopRight" Background="#70a524" Grid.Column="1" Grid.Row="0">
            <TextBlock x:Name="TopRightText"
                       Style="{StaticResource DetailViewLabelStyle}"
                       Text="Top-Right"/>
        </StackPanel>
        <StackPanel x:Name="BottomLeft" Background="#1E3905" Grid.Row="1">
            <TextBlock x:Name="BottomLeftText"
                       Style="{StaticResource DetailViewLabelStyle}" Text="Bottom-Left"/>
        </StackPanel>
        <StackPanel x:Name="BottomRight" Background="#45860B" Grid.Column="1"
                    Grid.Row="1">
            <TextBlock x:Name="BottomRightText"
                       Style="{StaticResource DetailViewLabelStyle}"
                       Text="Bottom-Right"/>
        </StackPanel>
    </Grid>
</Page>
```

此布局创建了一个简单的彩色网格。你可以在图 4-1 看到在布局展示的 Filled、snapped 模式。在布局展示的另外的应用程序是一个简单的占位符,这些你可以在 Apress.com 下载的源代码看到。

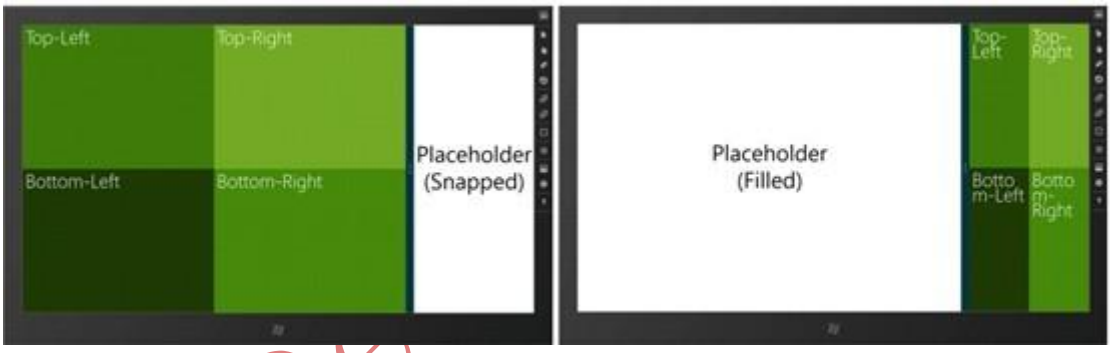

图 4-1.fill 和 snapped 模式下的示例程序

Note 应用程序可以被截断只有在横屏视图,先前的 Windows 8 消费者版本只有在水平分辨率为 1366 像素或更高支持 snapped。如果你想要尝试 snapped 模式你必须确保你的模拟器设置了正确的方向和分辨率。

对于大多数应用来说，为了腾出空间的 320 像素的损失不会造成很大的破坏。问题开始在你的应用从 filled 转换为 snapped 视图时出现,您可以在图的右边看到。显然,这个应用程序需要适应新的布局,在接下来的章节里,我将向您展示不同的机制来处理这样的情况。

Tip 你可以移动应用程序的不同的布局通过按 Win+。(Windows 和“.”键)。每一次你按这些键,该应用程序将周期性地变成新的布局。

4.1.1. 在代码中应对布局变更

在布局系统的核心是 Windows.UI.ViewManagement.ApplicationView 类发出的 ViewStateChanged 事件。通过处理这个事件,你可以通过重新配置您的应用程序来应对布局变化。Listing4-2 展示了 DetailView 页面的代码，里面有该事件的处理。

Listing 4-2. Controlling the Layout in the DetailView Code-Behind Class

```
using Windows.UI.ViewManagement;
```

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer.Pages {
    public sealed partial class DetailPage : Page {
        public DetailPage() {
            this.InitializeComponent();
            ApplicationView.GetForCurrentView().ViewStateChanged
                += (sender, args) => {
                        HandleViewStateChange(args.ViewState);
                };
        }
        private void HandleViewStateChange(ApplicationViewState viewState) {
            if (viewState == ApplicationViewState.Snapped) {
                GridLayout.ColumnDefinitions[0].Width
                    = GridLengthHelper.FromPixels(0);
            } else {
                GridLayout.ColumnDefinitions[0].Width
                    = GridLengthHelper.FromValueAndType(1, GridUnitType.Star);
            }
        }
    }
}
```

在此处理事件传递了一项 ApplicationViewStateChangedEventArgs 对象,其 ViewState 属性的返回值是描述了目前的布局的 ApplicationViewState 枚举。 在这个枚举里的是 snapped, Filled,FullScreenPortrait,FullScreenLandscape;最后两个允许你在应用程序显示全屏时区分 landscape 和 portrait 模式。

HandleViewStateChange 的方法查看布局的类型和调整。如果应用程序以 snapped 视图展示,然后我把我的网格第一列的宽度设为零。当你的应用程序恢复全屏模式的时候你的应用状态不会自动重置,因此您还必须定义代码来处理其他的事件状态;在这种情况下,对除了 snapped 外的任何其他布局,我重置了列的宽度。(使用列的语法很笨拙,但还是达到了目的)。在图 4-2 你可以看到的变化。

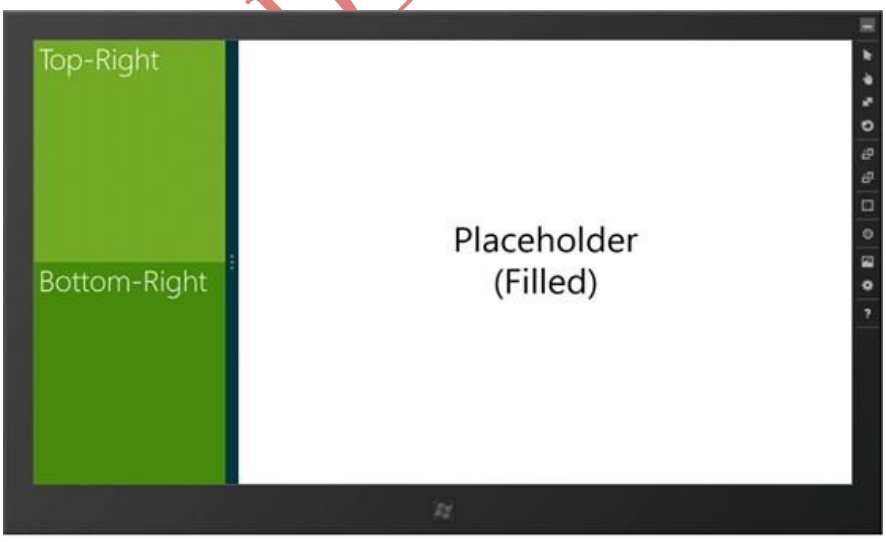

图 4-2.适配过的 snapped 视图

不是所有东西都压扁成了一扇小窗,我可以显示我的应用程序的一部分。相对于 snapped 应用能够访问的较小的屏幕空间显示简化功能是最明智的方式。你会惊奇地发现多少您可以装入这个空间,但是它和访问整个屏幕不一样。

Tip 作为一种替代方法来适应当前的布局,您可以显示一个完全不同的页面。示例见在第二章和第三章中如何使用框架控制来做到这一点。

4.1.2. 在 XAML 中应对布局变化

你可以使用 XAML 在你的应用程序设置变化。XAML 语法这很冗长,难以阅读,更难以执行---以至于我建议你坚持代码的方法。但为了完整起见,Listing4-3 展示了我如何使用 XAML 指定修改。

Note 在 XAML 中使用的 VisualStateManager 的特性，是一个标准的 WPF 和 Silverlight 特性。 它有很多特性,我无法在这本书给它充分关注。我的建议是使用基于代码的方式,但如果你是一个真正的 XAML 迷,那么您可以看到 WPF 或 Silverlight 文档元素的更多细节你可以使用。

Listing 4-3. Defining Layout Changes in XAML

```
<Page
    x:Class="MetroGrocer.Pages.DetailPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroGrocer.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
<Grid x:Name="GridLayout" Background="#71C524">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="OrientationStates">
            <VisualState x:Name="Snapped">
                <Storyboard>
                    <ObjectAnimationUsingKeyFrames
                            Storyboard.TargetProperty="Grid.ColumnDefinitions[0].Width"
                            Storyboard.TargetName="GridLayout">
                        <DiscreteObjectKeyFrame KeyTime="0">
                            <DiscreteObjectKeyFrame.Value>
                                <GridLength>0</GridLength>
                            </DiscreteObjectKeyFrame.Value>
                        </DiscreteObjectKeyFrame>
                    </ObjectAnimationUsingKeyFrames>
                </Storyboard>
            </VisualState>
            <VisualState x:Name="Others">
                <Storyboard>
                    <ObjectAnimationUsingKeyFrames
                            Storyboard.TargetProperty="Grid.ColumnDefinitions[0].Width"
                            Storyboard.TargetName="GridLayout">
                        <DiscreteObjectKeyFrame KeyTime="0">
                            <DiscreteObjectKeyFrame.Value>
                                <GridLength>*</GridLength>
                            </DiscreteObjectKeyFrame.Value>
                        </DiscreteObjectKeyFrame>
                    </ObjectAnimationUsingKeyFrames>
                </Storyboard>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
```

```
    </Grid.ColumnDefinitions>
    // ...StackPanel elements removed for brevity...

</Grid>
</Page>
```

在这个清单我声明了两个 VisualState 元素。第一,snapped,设置第一列的宽度为零像素;当改应用是 snapped 模式时这是我要进入的状态。第二个命名为 Others,作用是恢复宽宽度;当用用不是 snapped 模式时这是我要进入状态。你可以体会到我所说的冗长;我花 31 行 XAML 代码来取代 8 行代码。

我仍然要处理的 ViewStateChanged 事件,以便我可以进入我定义的 XAML 模式。Listing4 - 4 显示了在后台代码文件需要的改变。

Listing 4-4. Invoking the VisualStateManager in Response to the ViewStateChanged Event

```
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer.Pages {
    public sealed partial class DetailPage : Page {
        public DetailPage() {
            this.InitializeComponent();
            ApplicationView.GetForCurrentView().ViewStateChanged
                += (sender, args) => {
                    string stateName = args.ViewState ==
                        ApplicationViewState.Snapped ? "Snapped" : "Others";
                    VisualStateManager.GoToState(this, stateName, false);
                };
        }
    }
}
```

我调用了静态的 VisualStateManager.GoToState 方法在我在 XAML 定义的状态之间移动。该方法的参数是当前页面对象,要进入的状态的名字,以及是否应该显示的中间态。这最后的参数为 true 时,Windows 为布局过渡提供了动画。

4.1.3. 打破 snapped 视图

如果你在 snapped 视图给用户展示简化功能,当用户与应用程序交互时你可以以某种方式回到一个更广泛的布局。为了证明这一点,我添加了一个按钮到布局 DetailView 页面,如 Listing4 - 5 所示。

Listing 4-5. Adding a Button to the Layout

```
...
<StackPanel x:Name="TopRight" Background="#70a524" Grid.Column="1" Grid.Row="0">
    <TextBlock x:Name="TopRightText"
                Style="{StaticResource DetailViewLabelStyle}"
                Text="Top-Right"/>
    <Button Click="HandleButtonClick">Unsnap</Button>
</StackPanel>
...
```

Listing4 - 6 显示了点击事件处理程序，使用 TryUnsnap 方法 unsnap 应用程序。

Listing 4-6. Unsnapping an App

```
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer.Pages {
    public sealed partial class DetailPage : Page {
```

```
        public DetailPage() {
            this.InitializeComponent();
            ApplicationView.GetForCurrentView().ViewStateChanged
                += (sender, args) => {
                    string stateName = args.ViewState ==
                        ApplicationViewState.Snapped ? "Snapped" : "Others";
                    VisualStateManager.GoToState(this, stateName, false);
                };
        }
        private void HandleButtonClick(object sender, RoutedEventArgs e) {
            Windows.UI.ViewManagement.ApplicationView.TryUnsnap();
        }
    }
}
```

TryUnsnap 的方法将改变布局,但是,只有在应用程序有情景;也就是说,只有在响应用户交互时一些后台活动可以使布局自动改变。

Tip 我得到一些不一致的结果,当我 unsnap 这样一个应用程序在 Windows 8 消费者预览版。有时,应用程序转变为了 filled 视图,其他时候它会变成全屏布局。

4.2. 使用磁贴和徽章

磁贴是将应用程序放置在开始菜单中。最简单的,磁贴都是用来打开应用的静态的图标。然而,只要努力一下,你就可以用你的磁贴为用户显示关于您的应用程序的状态相关的摘要。

在接下来的小节中,我将通过 Metro 示例应用演示如何使用磁贴。当你创建一个动态的磁贴时，有两种可能。你要么试图鼓励用户运行您的应用程序要么是劝阻他们运行它。如果你正在努力吸引用户,然后你的磁贴成为你提供的一个体验，见解或内容的广告。这是适合娱乐应用或者那些如新闻一样呈现外面内容的应用。

劝阻用户运行一个应用程序的目标可能看起来很奇怪,但它可以显著改善用户体验。考虑提高工作效率的应用程序作为示例。当我需要检查我的下一个约会是什么，以及我最紧急的行动，我不想让应用加载已经失效的日程或者待办事项。当使用你的应用时你可以使你的用户体验减少摩擦和挫折,创造一个更直接令人愉悦的体验通过显示用户需要的信息在自己的应用程序磁贴中。

这两个目标都需要仔细考虑。整个 Metro 体验是单调的,简单的,柔和的。如果您使用磁贴作为一个广告,那么 Metro 低调的性质使应用更容易通过创建磁贴脱颖而出。但是如果你做得太过火,您将创建一些不和谐和不吸引人的东西。

如果你的目标是减少用户需要运行您的应用程序的次数,那么您需要在正确的时间提供正确的信息。这需要很好地理解是什么在驱动你的用户使用您的应用程序并且可以提供定制什么样的数据。适应性是非常必要的;例如，每次你为用户提供错误的信息,你迫使他们运行您的应用程序来获取他们所需要的。

Tip 只有一个应用程序运行的时候，才可以更新它的磁贴。在第五章中,我详细讲述 Metro 应用程序的生命周期,你会发现当用户切换到另一个应用程序 Metro 应用被放入一个暂停状态。这意味着你不能在后台提供更新。Windows 8 支持推动的模型,您可以从云端发送 XML 更新,但这项服务在消费者预览版还不能使用。

4.2.1. 改善静态磁贴

最简单的方法来改善应用程序在开始菜单的外观是改变用于应用程序的磁贴图片。你应该为您的应用程序定制图片,即使你不使用磁贴的其它特性。

要做到这一点,你需要一组三张图片，具体尺寸为:30×30 像素,150 x 150 像素和 310 x 150 像素。这些图像应该包含你想要显示的徽标或文本并且是不透明的。我在我的示例应用程序使用一个条形码图案,并且创建的图片命名为 tile30.png,tile150.png,tile310.png,把他们放置在我的 Visual Studio 项目的 Assets 文件夹。

应用新的图像,在解决方案管理器打开 package.appxmanifest 文件。 应用程序 UI 选项卡上有一个磁贴部分,有一些选项来设置徽标,宽徽标,以及小徽标。对每个选项需要的大小会有提示说明。你还需要设置磁贴的背景颜色;我设置的是和我的应用背景相同的颜色(十六进制 RGB 值#3E790A)。

Tip 在 manifest 中设置背景颜色是很重要的,并不仅仅是图片的背景。在接下来的部分，我将演示当更新一个磁贴替换的是动态信息,

并且在 manifest 指定了背景颜色。

你可能要在开始界面卸载你的 Metro 应用以使磁贴图片生效。接着你在 Visual Studio 运行你的应用程序,您应看到新的静态磁贴;你可以之间切换的标准和广泛的视图通过选择磁贴和在 AppBar 点击更大或更小的按钮。你可以在图 4 - 3 看到示例应用程序的方磁贴和宽磁贴格式。

图 4-3.更新静态宽瓷贴

注意在磁贴底部显示的 Grocer 这个词，在应用程序 UI 中我指定了这个文本作为了短名称的值和在显示名称选项选择了所有徽章选项以便它能应用于普通磁贴和宽磁贴。

Tip 你也可以替换在程序启动的时候展示给用户的启动画面。在应用程序 UI 选项卡的底部有一个启动画面部分,您可以在其中指定需要展示的图像和背景颜色。用于启动画面的图像必须是 630×300 像素。

4.2.2. 创建动态磁贴

动态磁贴可以提供有关你的应用的信息给用户，对于我的示例应用程序，我将展示购物清单前面的那几项,它并不是最有益的特性,但它确实让我展示动态磁贴特性。

磁贴的更新都是基于预配置的模板,它包含一个图形和文本的组合,适用于标准或宽的磁贴。首先你必须做的是选择你想要的模板。最简单的方法是看为 Windows.UI.Notifications API 文档。 TileTemplateType 枚举,在 http://googl/kBL7O 可以看到(我在这一章用短的 url,因为微软的 url 既长又难以阅读)。模板系统是基于 XML 片段,您可以看到你在文档中选择了的模板的 XML 结构。我选择了 TileSquareText03 模板。 这是一个方磁贴和有四行 nonwrapping 文本,并且不带任何图片。你可以看到表现磁贴的这个 XML 片段在 Listing4 – 7

Listing 4-7. The XML Fragment for the tileSquareText03 Tile Template

```
<tile>
  <visual lang="en-US">
    <binding template="TileSquareText03">
      <text id="1">Text Field 1</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

想法是从应用程序填充文本元素信息并传递结果给 Metro 磁贴通知系统。我想要在 MainPage 类建立我的磁贴的更新,但是这样做意味着重构我的应用程序,以便 ViewModel 对象在 MainPage 类中被创建,而不是在 ListPage 类。Listing4-8 展示了 MainPage 类的修改来支持视图模型和更新磁贴。

Listing 4-8. Refactoring the MainPage Class

```
using System;
using MetroGrocer.Data;
using Windows.Data.Xml.Dom;
```

```
using Windows.UI.Notifications;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Navigation;
namespace MetroGrocer.Pages {

    public sealed partial class MainPage : Page {
        private ViewModel viewModel;
        public MainPage() {
            this.InitializeComponent();
            viewModel = new ViewModel();
            // ...test data removed for brevity
            this.DataContext = viewModel;
            MainFrame.Navigate(typeof(ListPage), viewModel);
            viewModel.GroceryList.CollectionChanged += (sender, args) => {
                UpdateTile();
            };
            UpdateTile();
        }
        private void UpdateTile() {
            XmlDocument tileXml = TileUpdateManager.
                GetTemplateContent(TileTemplateType.TileSquareText03);
            XmlNodeList textNodes =
                tileXml.GetElementsByTagName("text");
            for (int i = 0; i < textNodes.Length &&
                    i < viewModel.GroceryList.Count; i++) {
                textNodes[i].InnerText = viewModel.GroceryList[i].Name;
            }
            for (int i = 0; i < 5; i++) {
                TileUpdateManager.CreateTileUpdaterForApplication()
                    .Update(new TileNotification(tileXml));
            }
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
        private void NavBarButtonPress(object sender, RoutedEventArgs e) {
            Boolean isListView = (ToggleButton)sender == ListViewButton;
            MainFrame.Navigate(isListView ? typeof(ListPage)
                : typeof(DetailPage), viewModel);
            ListViewButton.IsChecked = isListView;
            DetailViewButton.IsChecked = !isListView;

        }
    }
}
```

在这么短短几行代码中做了很多事，所以我在接下来的几部分分开解释。

填充 XML 模板

为了获得模板的 XML 片段,我调用 TileUpdateManager。 GetTemplateContent 方法要指定一个模板我从 TileTemplateType 枚举选择了一个值。 Windows.Data.Xml.Dom.XmlDocument 对象让我可以在模板汇总使用标准的 DOM 方法来设置文本元素的值。对了，在 XmlDocument 对象中没有实现 GetElementById 方法，所以我不得不使用 GetElementsByTagName 方法来获取一个数组包含在 XML 中所有的 text 元素。

```
...
XmlNodeList textNodes = tileXml.GetElementsByTagName("text");
...
```

返回的文本节点的顺序与它们被定义在 XML 片段的顺序相同,这意味着我可以迭代和设置每个元素的 innerText 属性来对应我的购物清单的每一项：

```
...
for (int i = 0; i < textNodes.Length && i < viewModel.GroceryList.Count; i++) {
    textNodes[i].InnerText = viewModel.GroceryList[i].Name;
}
...
```

Tip 定义在 XML 模板中的四个 text 元素只有 3 个 text 元素可以在开始菜单可见，最后一个元素被应用的名字或者图标遮挡了，这就是大多数磁贴模板的真相。

应用磁贴更新

一旦我创建好了 XML 文档的内容,我用它来为应用程序磁贴创建一个更新。我需要从 XML 创建一个 TileNotification 对象,然后传递这个对象到 TileUpdater 对象的 update 方法中。TileUpdater 对象是由 TileUpdateManager.CreateTileUpdaterForApplication 这个静态方法返回的：

```
...
for (int i = 0; i < 5; i++) {
    TileUpdateManager.CreateTileUpdaterForApplication()
        .Update(new TileNotification(tileXml));
}
...
```

在消费者预览版并不是所有的磁贴更新都正确处理,这就是为什么我使用 for 循环重复这个通知。5 次似乎是最小的重复次数来保证一个更新就会显示在开始菜单。

调用磁贴更新方法

我在两种情况下调用 UpdateTile 方法。第一个是在构造函数直接调用,它保证了当应用程序启动的时候磁贴可以反映出视图模型的当前数据。第二个情况是,当集合的内容改变的时候调用：

```
...
viewModel.GroceryList.CollectionChanged += (sender, args) => {
    UpdateTile();
};
...
```

CollectionChanged 事件会触发当在购物清单项目集合添加、替换或删除一个项目。它不会被触发当个人的 GroceryList 对象的属性被修改,这样的安排,我必须添加每个对象集合的处理程序。没有 Metro-specific 技术显示你要这样做,所以我在这一章只想关注集合变更问题。

测试磁贴更新

在我测试磁贴更新之前需要两个预备步骤。首先,Visual Studio 模拟器不支持更新磁贴,这意味着我要直接在我的开发机器上测试。为此,我需要改变 Visual Studio 部署目标到本地计算机上,如图 4 - 4 所示。

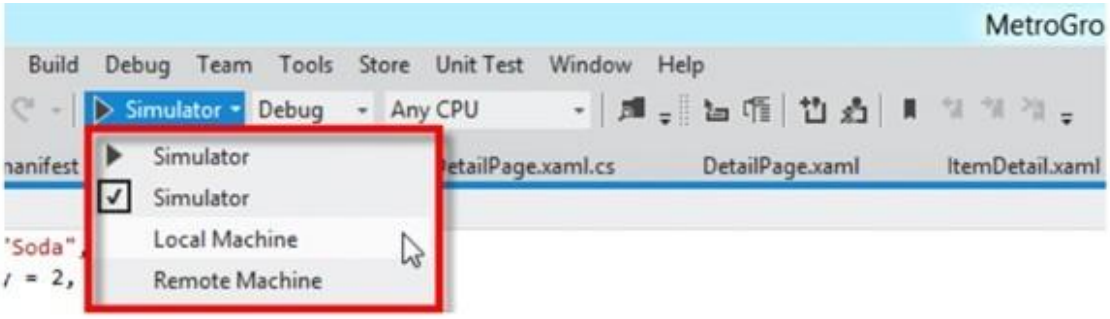

图 4-4.选择本地计算机调试

第二步是从开始菜单中卸载我的示例应用程序(这可以从 AppBar 选择卸载)。在消费者预览版,似乎有一些

“粘性”,应用程序之前依靠静态磁贴不能正确处理更新。

完成这些步骤后,我现在可以开始调试我的应用程序通过在 Visual Studio 的调试菜单选择开始调试。当应用程序启动后,我可以更改购物清单,和一个前三个条目的简洁的汇总将会显示在磁贴,见图 4 - 5。

图 4-5.更新程序的瓷贴

可能很难立即获得更新后磁贴。你可以从以下获得帮助:

关闭模拟器

1、重启 visual studio

2、在开始菜单卸载另一个（没有关系）应用

3、寻找使用开始开始菜单的 MetroGrocer 应用。

4、在开始菜单移动一些其他的磁贴

5、重启

要让磁贴一开始就出现可能会令人沮丧，但是一旦它就在那里,一切都将按预期工作,并且进一步更新应用程序很少使磁贴消失。

更新宽磁贴

如果你想让你的应用程序更新方的或宽的磁贴，我在前一节中向你展示了该技术是有效的。但是,除非你的数据需要非常特定的展示,你应该提供方磁贴和宽磁贴的更新因为你不知道你的用户会怎样选择。

要更新两个磁贴尺寸,你需要将两个 XML 模板来创建一个单一的片段,其中包含两个更新。在本节中,我将结合 TileSquareText03 和 TileWideBlockAndText01 模板。 宽模板有几个额外的字段,我将使用显示用户为获得购物清单上所有条目必须访问的商店的数量。你可以看到我做了什么在 Listing4-9 ——— 遵循相同的格式的单个模板的片段，但它结合了两种绑定元素。

Listing 4-9. Composing a Single XML Fragment

```
<tile>
    <visual lang="en-US">
        <binding template="TileSquareText03">
            <text id="1">Apples</text>
            <text id="2">Hotdogs</text>
            <text id="3">Soda</text>
            <text id="4"></text>
        </binding>
        <binding template="TileWideBlockAndText01">
            <text id="1">Apples (Whole Foods)</text>
            <text id="2">Hotdogs (Costco)</text>
            <text id="3">Soda (Costco)</text>
            <text id="4"></text>
            <text id="5">2</text>
            <text id="6">Stores</text>
        </binding>
    </visual>
</tile>
```

对于组合没有方便的 API。我采用的方法是使用 XML 处理来支持分开填充模板,然后在过程的最后将它们

组合,您在 Listing4-10 可以看到。

Listing 4-10. Producing a Single Update for Square and Wide Tiles

```
...
private void UpdateTile() {
    int storeCount = 0;
    List<string> storeNames = new List<string>();
    for (int i = 0; i < viewModel.GroceryList.Count; i++) {
        if (!storeNames.Contains(viewModel.GroceryList[i].Store)) {
            storeCount++;
            storeNames.Add(viewModel.GroceryList[i].Store);
        }
    }
    XmlDocument narrowTileXml = TileUpdateManager
        .GetTemplateContent(TileTemplateType.TileSquareText03);
    XmlDocument wideTileXml = TileUpdateManager
        .GetTemplateContent(TileTemplateType.TileWideBlockAndText01);
    XmlNodeList narrowTextNodes = narrowTileXml.GetElementsByTagName("text");
    XmlNodeList wideTextNodes = wideTileXml.GetElementsByTagName("text");
    for (int i = 0; i < narrowTextNodes.Length
        && i < viewModel.GroceryList.Count; i++) {
        GroceryItem item = viewModel.GroceryList[i];
        narrowTextNodes[i].InnerText = item.Name;
        wideTextNodes[i].InnerText = String.Format("{0} ({1})", item.Name, item.Store);
    }
    wideTextNodes[4].InnerText = storeCount.ToString();
    wideTextNodes[5].InnerText = "Stores";
    var wideBindingElement = wideTileXml.GetElementsByTagName("binding")[0];
    var importedNode = narrowTileXml.ImportNode(wideBindingElement, true);
    narrowTileXml.GetElementsByTagName("visual")[0].AppendChild(importedNode);

    for (int i = 0; i < 5; i++) {
        TileUpdateManager.CreateTileUpdaterForApplication()
            .Update(new TileNotification(narrowTileXml));
    }
}
...
```

宽磁贴给我提供了一个机会在每一行提供更多的信息给用户，在这种情况下，我把除了总体数量存储访问需要外存储需要购买的条目的信息。

组合模板并不是一个难以控制的过程,但你必须小心当试图合并这两个 XML 片段。我已经使用这个模板在方磁贴并且作为我组合更新的基础。当我从宽模板添加绑定元素,我首先必须将其导入到方的 XML 文档,像这样：

```
var importedNode = narrowTileXml.ImportNode(wideBindingElement, true);
```

ImportNode 的方法创建一个我的宽绑定元素的一个新的拷贝在我的方的文档的环境中。ImportNode 方法的参数是我想要导入的元素和一个 bool 值,代表是否我想要导入子节点(当然,我做的)。一旦我已经创建了这个新元素,我使用 AppendChild 元素将它插入到方的 XML：

```
narrowTileXml.GetElementsByTagName("visual")[0].AppendChild(importedNode);
```

结果就如在 Listing 4-9 我展示给你看的组合的文档，你可以在图 4-6 看到两种尺寸的磁贴。(你可以切换方磁贴和宽磁贴通过选择磁贴和使用开始菜单 AppBar。)

图 4-6.更新宽瓷贴

4.2.3. 应用徽章

Metro 设法使磁贴包含很多特性,包括支持可以是小图标或数字叠加的徽章添加到磁贴。这个覆盖物分进了 tile-as-an-ad 类别,因为很少有哪些情况一个数字表示邀请用户启动应用之外的任何事情。

Tip 虽然我展示磁贴和徽章一起使用，你也可以直接应用徽章在静态磁贴。
演示徽章,我将展示一个简单的指标基于在购物清单的项目数量。

Listing 4 - 11 显示了 MainPage 类的添加。

Listing 4-11. Adding Support for Tile Badges

```
using System;
using MetroGrocer.Data;
using Windows.Data.Xml.Dom;
using Windows.UI.Notifications;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Navigation;
using System.Collections.Generic;
namespace MetroGrocer.Pages {
    public sealed partial class MainPage : Page {
        private ViewModel viewModel;
This book was purchased by wangxinglongcn@163.com
        public MainPage() {
            this.InitializeComponent();
            viewModel = new ViewModel();
            // ...test data removed for brevity...
            this.DataContext = viewModel;
            MainFrame.Navigate(typeof(ListPage), viewModel);
            viewModel.GroceryList.CollectionChanged += (sender, args) => {
                UpdateTile();
                UpdateBadge();
            };
            UpdateTile();
            UpdateBadge();
        }
        private void UpdateBadge() {
            int itemCount = viewModel.GroceryList.Count;
            BadgeTemplateType templateType = itemCount > 3
                ? BadgeTemplateType.BadgeGlyph : BadgeTemplateType.BadgeNumber;
            XmlDocument badgeXml = BadgeUpdateManager.GetTemplateContent(templateType);
            ((XmlElement)badgeXml.GetElementsByTagName("badge")[0]).SetAttribute("value",
                (itemCount > 3) ? "alert" : itemCount.ToString());
```

```
            for (int i = 0; i < 5; i++) {
                BadgeUpdateManager.CreateBadgeUpdaterForApplication()
                    .Update(new BadgeNotification(badgeXml));
            }
        }
        private void UpdateTile() {
            // ...code removed for brevity...
        }
        protected override void OnNavigatedTo(NavigationEventArgs e) {
        }
        private void NavBarButtonPress(object sender, RoutedEventArgs e) {
            Boolean isListView = (ToggleButton)sender == ListViewButton;
            MainFrame.Navigate(isListView ? typeof(ListPage)
                : typeof(DetailPage), viewModel);
            ListViewButton.IsChecked = isListView;
            DetailViewButton.IsChecked = !isListView;
        }
    }
}
```

徽章的工作类似于磁贴通知。你获得 XML 模板,填充内容,并使用它通过开始菜单来展示一些信息提供给用户。有两种可用的徽章模板。第一,数字模板,将显示一个 1 到 99 之间的数值,第二,字形模板,将在窗口定义的有限的范围内显示一个小图像。

在消费者预览版数字和字形模板是相同的。如 Listing4-12 所示，比我在磁贴使用更简单。

Listing 4-12. The Template for Numeric and Image Badges

```
<badge value=""/>
```

目标是设置 value 属性关联一个数字或者一个字形的名字。如果在购物清单有三种或者更少的条目我就显示一个数字徽章，如果超过三个条目，我就使用一个图标来标明用户应该关心购物的债务程度。

创建一个徽章的过程来始于选择一个模板。这两个模板类型是 Windows.UI.Notifications。BadgeTemplateType:数字徽章使用 BadgeNumber 模板,而对于图标要使用 BadgeGlyph 模板。 在这两种情况下你可以使用相同的模板,因为他们返回相同的 XML,至少在消费预览版上。在以后的版本中这可能会改变,因此明智的做法是选择正确的模板,即使其内容是相同的。

下一步是在 XML 找到 value 属性并将给它分配一个数值或者图标的名字。数值范围在徽章中是指定的;它是 1 到 99。如果您设置的值小于 1,徽章不会被显示。在徽章中任何值大于 99 的结果就展示 99。

图标的列表也同样具有规定性。你不能用你自己的图标和必须从 windows 支持的十个中选择。你可以看到这个图标列表在 http://goo.gl/YoYee。在这个例子中,我选择了看上去就像一个星号的警报图标,。一旦 XML 被填充,您创建一个新的 BadgeNotification 对象,并使用它来发布更新。如同磁贴,我发现并非所有的徽章更新被处理,所以我重复更新五次,确保它能通过:

```
...
for (int i = 0; i < 5; i++) {
    BadgeUpdateManager.CreateBadgeUpdaterForApplication()
        .Update(new BadgeNotification(badgeXml));
}
...
```

剩下要做的就是确保我的徽章更新被创建。要做到这一点,我已经改变了购物清单的处理事件,从而使磁贴和徽章一起更新。在图 4- 7 你可以看到四个不同的徽章/磁贴配置：带有数值和图标徽章的宽、方磁贴。

4. 3. 总结

在这一章中,我向您展示了如何调整 Metro 的 snapped、filled 布局和如何使用磁贴诱惑您的用户来运行你的应用程序,或提供数据使他们避免这样做。这些特性在发表一个应用是至关重要的并且集成到了更广泛的 Metro 经验中。

你可能认为在 snapped 布局中其大量的可用空间过于局限于提供一些严肃的功能,但是经过仔细思考，它可

能是关注你提供本质的服务,并忽略其他一切。如果所有这些都不奏效,你可以给你的应用一个 information-only 总结和显式地打破布局。

仔细考虑需要得到最佳的磁贴和徽章。深思熟虑的徽章可以大大提高你的应用程序的吸引力或者实用性,但考虑不周的磁贴是令人讨厌的或仅仅是无用的。

第 5 章 应用程序生命周期与合约

在这里，本书的最后一章，我会告诉你如何通过响应关键的 Windows 事件来控制 Metro 应用程序的生命周期。如何在 Visual Studio 项目中修改代码，如何妥善处理暂停和恢复您的应用程序，以及如何执行合约，配合您的应用程序在 Windows 8 中提供的更广泛的用户体验。在这里，我将演示使用的地理定位功能，并告诉你如何设置和管理经常性的异步任务。表 5-1 提供了本章的总结。

注意您将需要卸载 Metro 应用程序的例子，如果你前一章使用从 Apress.com 上下载过源代码的例子。右键单击 MetroGrocer 开始菜单选择卸载。如果您运行了一个从不同的项目路径安装的程序，Visual Studio 会报告一个错误。您必须先卸载这一个和其他的。

表 5-1 本章小结

问题	解决方案	Listing
确保你的应用程序接收到生命周期	在应用程序类中处理暂停核恢复事件	1
当程序暂停时，确保终止后台清理任务	请求延迟 5 秒为暂停做准备	2-4
实施合约	在添加功能代码重载应用方法，所以在接受生命周期事件时合约被执行	5-7

5.1. 处理 Metro 应用程序生命周期

在第一章，我在 Visual Studio 中的 App.xaml.cs 文件向您展示的骨感简单的代码是我的项目例子的一个开始。这些代码就是 Metro 应用程序生命周期事件的句柄，确保操作系统发送信号时我能及时响应处理。在 Metro 应用程序生命周期中有三个关键阶段。

第一个阶段，激活，发生在你的应用程序被启动，当所有事情都准备好时，Metro runtime将会加载和处理你的内容和信号。在激活的过程中引起我的应用例子动态的内容，例如.

用户通常不会关闭Metro应用程序，他们只是离开窗口处理事件和转移到其他应用。这是为什么在Metro UI上没有后关闭按钮或菜单栏。这个Metro应用不会被改变为第二个暂停阶段。而暂停，是没有与用户交互，没有执行程序代码。

如果用户切换回暂停应用程序，第三阶段：程序恢复。应用程序是显示给用户，和执行应用程序。暂停并不是总能够恢复；如果设备内存不足，例如，windows可以简单的终止暂停的程序。

5.1.1. 纠正 Visual Studio 事件代码

遗憾的是添加到 Visual Studio 项目中的生命周期事件句柄不能正确工作。它能够良好的处理激活和暂停，但是当它恢复时是防止被通知的。幸运的是，这个解决方案很简单，你可以看需求的改变在 App.xaml.cs 中。如表 5-1 所示。

Listing 5-1. Handling the Life-Cycle Notification Events

```
using MetroGrocer.Data;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
namespace MetroGrocer {
    sealed partial class App : Application {
        private ViewModel viewModel;
        public App() {
```

```
            this.InitializeComponent();
            viewModel = new ViewModel();
            // ...test data removed for brevity...
                        this.Suspending += OnSuspending;
            this.Resuming += OnResuming;
        }
    protected override void OnLaunched(LaunchActivatedEventArgs args) {
            // Create a Frame to act navigation context and navigate to the first page
            var rootFrame = new Frame();
            rootFrame.Navigate(typeof(Pages.MainPage), viewModel);
            // Place the frame in the current Window and ensure that it is active
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
        private void OnResuming(object sender, object e) {
            viewModel.GroceryList[1].Name = "Resume";
        }
        void OnSuspending(object sender, SuspendingEventArgs e) {
            viewModel.GroceryList[0].Name = "Suspend";
        }
    }
}
```

你会发现第一个变化是，我已经把视图模型从 MainPage 类移到了这个里面。一直移动视图模型逐渐展示出其在 Metro 应用程序中的核心作用。在本节中，将使用它运转一个调试器限制，在后面的章节中，将讲述 Metro 合同。你会发现很多任务如果没有试图模型就不能执行，并且把视图模型对象放在应用程序的核心地位是非常准确的。

提示：这里已经对 MainPage 类作出了相应的更新，并为这些变化下载了源代码。在前面的章节中，它们遵循相同的模式。本节对应用程序文件的更新也做了注释，这样就可以在模拟器中测试生命周期事件。

第二个变化是，本节已经直接记录下了暂停和恢复事件；当它创建类时，Visual Studio 包含一个暂停事件的处理程序，但是必须添加一个恢复处理程序，以获得事件通知。当应用程序恢复时，我已通过 OnLaunched 删除代码试图工作，还是以失败告终。

本节中的 Metro 应用程序目前不执行因应用程序暂停和恢复而受影响的任何任务，但主要介绍了如何测试事件。为此，针对暂停和恢复事件，当事件收到时，通过改变在视图模型中 GroceryList 集合的前两个项目的名称属性做出信号反应。

5.1.2. 模拟生命周期事件

模拟生命周期事件的最简单的方法是使用 Visual Studio。当你用调试器启动一个应用程序时，Visual Studio 中就会在工具栏中添加了一些按钮，允许你往应用程序中发送生命周期事件。这些按钮没有对应的菜单项。由于它们是很难在单色视觉 Studio 界面中识别出来，我在图 5-1 进行了标注。

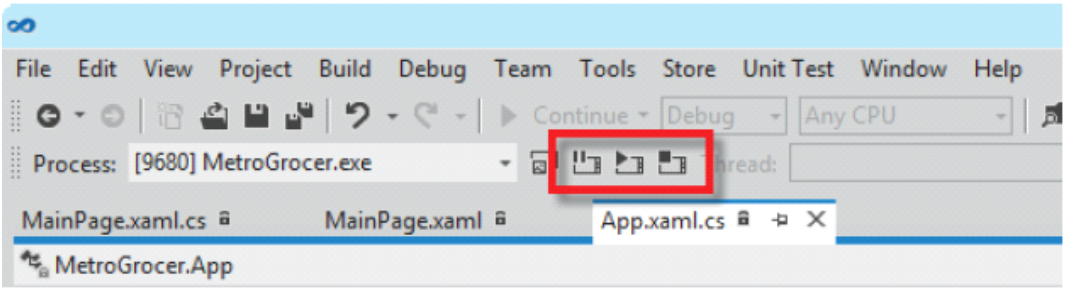

图 5-1 Visual Studio 按钮发送生命周期事件到应用程序

如果你想按按钮暂停和恢复应用程序，你将会在可视模版数据中看到这些变化，如图 5-2 所示。（你恢复

应用程序后才会看到这些变化，一旦应用程序被暂停，就不会有 UI 更新被处理。）

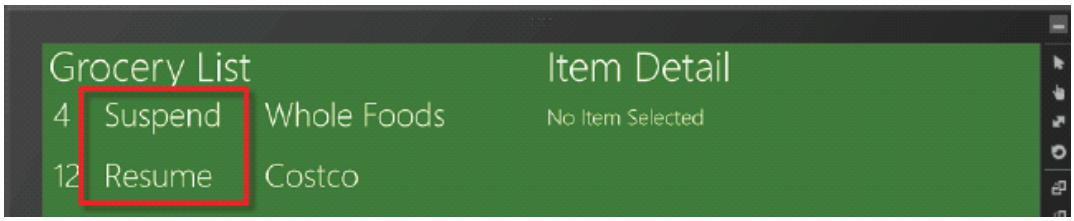

图 5-2 表明生命周期事件已收到

5.1.3. 测试生命周期事件

模拟生命周期事件的问题是，当它们很自然的出现时，你会得到不同的结果。为解决这个问题，你启动的应用程序不需要调试，这就是事件之所以通过可视化模板显示非常有用的原因。创建事件发送的循环需要一些特殊的指令。我将会在以下章节中逐步展开讲解。

启动应用程序

触发激活的事件，启动应用程序（从 Visual Studio 调试菜单中不经过调试选择开始）。应用程序没有调试从选择开始启动在。无论是在模拟器或在本地机器上，你也可以从开始菜单启动应用程序。重要的是，不要通过调试器启动应用程序。

暂停应用程序

暂停应用程序的最简单的方法是切换到桌面上，按 WIN + D 。打开任务管理器，右键单击 Metro 应用程序项目，并选择从细节弹出式菜单。任务管理器将切换到“详细信息”选项卡，并选择 **WWAHost.exe**，它是负责应用程序运行的。几秒钟后，在状态列显示的值将从运行到暂停态，这表示 Windows 已经暂停了应用程序。这个应用程序将会发送暂停事件（在应用程序恢复前，你不会看到任何迹象）。

恢复应用程序

切换到应用程序将它恢复。你会看到视图模型项目显示该事件已收到。

一个应用程序恢复时的状态和它在暂停时刻的状态是一样的。你的布局、数据、事件处理程序，以及其他一切都没有变化。

你的应用程序可能已经被暂停了很长时间，特别是设备处于低电状态时，比如休眠。网络连接将被你正在说话时的任何一个服务器断开（所以当你启用暂停事件时，最好把它们全都关闭），当你的应用程序恢复时必须重新打开。你也不得不刷新数据，当然这很可能会造成数据的丢失，包括位置数据，因为在应用程序被暂停期间设备很可能被移动了。

提示：Windows 允许用户通过 ALT + F4 终止 Metro 应用程序。但不能确定这个功能在 Windows 8 的最终版本中能否使用，但您可能需要考虑您的应用程序。Windows 没有有用的警告事件给你机会去整理数据和操作，相反，它只会终止你应用程序的进程。

5.2. 添加一个后台任务

当已确认应用程序可以获取和回应恢复和暂停事件时，就可以添加一些需要经常性的后台任务的功能。比如说，我要使用地理定位服务来报告对当前设备的位置。首先，我要在数据项目文件夹中创建一个新的类文件，命名为 Location.cs。这个文件的内容如表 5-2 所示。

表 5-2 Location.cs 文件

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
```

```
using Windows.Data.Json;
using Windows.Devices.Geolocation;

namespace MetroGrocer.Data {
   class Location {

      public static async Task<string> TrackLocation() {
         Geolocator geoloc = new Geolocator();
         Geoposition position = await geoloc.GetGeopositionAsync();

         HttpClient httpClient = new HttpClient();
         httpClient.BaseAddress = new Uri("http://nominatim.openstreetmap.org");
         HttpResponseMessage httpResult = await httpClient.GetAsync(
            String.Format("reverse?format=json&lat={0}&lon={1}",
            position.Coordinate.Latitude, position.Coordinate.Longitude));

         JsonObject jsonObject = JsonObject
            .Parse(await httpResult.Content.ReadAsStringAsync());

         return jsonObject.GetNamedObject("address")
            .GetNamedString("road") + DateTime.Now.ToString('" ('HH:mm:ss')"');
      }
   }
}
```

这个类使用 Windows8 地理定位功能来获取设备的位置。这个功能是通过在 Windows.Devices.Geolocation 命名空间中的 Geolocator 类的，GetGeopositionAsync 方法获得一个位置的单一快照（而不是像其他由 Geolocator 类支持的应用软件那样提供通过事件提供位置更新）。

注意：当新的 C＃await 关键字出现时表明已经进入了并行/异步领域编程。这是一个使用任务并行库（TPL）创建和管理后台任务的典型例子。本书不对 TPL 和.NET 并行编程进行详细的介绍。如果您想了解更多信息，推荐使用 C＃书中的 Pro.NET4 并行编程，它提供了很详细的内容。await 关键字是在 C＃4.5 中新增加的，表示“等待异步任务来完成”。

如果获得设备位置，就可以对反向地理编码服务做一个 HTTP 请求，这样就可以解析经纬度信息，从地理位置服务到街道地址。地理编码服务返回一个 JSON 字符串，把它解析成一个 C＃对象，就可以读取街道信息。从 TrackLocation 方法得到的结果是一个字符串，它列出街道名称、设备和一个表示位置更新的时间的时间戳。

提示：OpenStreetMap 地理编码服务并不需要一个独特的帐户令牌。这意味着您可以运行的例子，无需创建一个 Google 地图或 Bing 地图帐户。

5.2.1. 扩展视图模型

扩展视图模型，使其保持跟踪 TrackLocation 方法所产生的位置数据，这样可以将使用的数据绑定到用户的数据显示。ViewModel 类的增加如表 5-3 所示。

表 5-3 更新视图模型捕获位置数据

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;

namespace MetroGrocer.Data {
   public class ViewModel : INotifyPropertyChanged {
      private ObservableCollection<GroceryItem> groceryList;
      private List<string> storeList;
      private int selectedItemIndex;
```

```
    private string homeZipCode;
    private string location;

    public ViewModel() {
      groceryList = new ObservableCollection<GroceryItem>();
          storeList = new List<string>();
      selectedItemIndex = -1;
      homeZipCode = "NY 10118";
      location = "Unknown";
    }

    public string Location {
      get { return location; }
      set { location = value; NotifyPropertyChanged("Location"); }
    }
    // ...other properties removed for brevity...


    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged(string propName) {
      if (PropertyChanged != null) {
        PropertyChanged(this, new PropertyChangedEventArgs(propName));
       }
    }
  }
}
```

5.2.2. 位置数据的显示

更新 MainPage.xaml 文件添加控件将面向用户显示位置数据。数据绑定确保从视图模型中得到的最前数据不需要更改代码隐藏文件，能够直接使用。添加的控件如表 5-4 所示。

表5-4 将控件添加到XAML文件显示位置数据

```
<Page
  x:Class="MetroGrocer.Pages.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:MetroGrocer.Pages"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Page.TopAppBar>
    <AppBar>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <ToggleButton x:Name="ListViewButton"
          Style="{StaticResource ToggleAppBarButtonStyle}"
          AutomationProperties.Name="List View" IsChecked="True"
          Content="&#xE14C;" Click="NavBarButtonPress"/>
        <ToggleButton x:Name="DetailViewButton"
          Style="{StaticResource ToggleAppBarButtonStyle}"
          AutomationProperties.Name="Detail View"
          Content="&#xE1A3;" Click="NavBarButtonPress"/>
        </StackPanel>
  </AppBar>
</Page.TopAppBar>
```

```
<Grid Background="{StaticResource AppBackgroundColor}">
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
   </Grid.RowDefinitions>

   <StackPanel Orientation="Horizontal" HorizontalAlignment="Center" >
      <TextBlock FontSize="30" Text="Your location is:" Margin="0,0,10,0" />
      <TextBlock FontSize="30" Text="{Binding Path=Location}" />
   </StackPanel>

      <Frame x:Name="MainFrame" Grid.Row="1"/>
   </Grid>
</Page>
```

5.2.3. 声明应用能力

应用程序必须申报自己的需求来获得他们的清单中的位置服务。在运行更新应用程序之前，先打开package.appxmanifest，切换到“功能”选项卡，确保该地点功能已被检查，并保存该文件。应用程序也需要具有互联网（客户）性能，但这个声明时，默认情况下在 Visual Studio 中创建项目。

5.2.4. 控制后台任务

这个例子的重点是，对于所有的接口，可以用 Metor 生命周期事件实现对后台任务的管理与整合。表 5-5 列出了 App.xaml.cs 文件的变化。

再次注意，这是一个很好的例子。如果对 NET 模型并行编程不熟悉，可以阅读下一章节。下一章将介绍在 Metro 应用程序中如何实现 Windows 合同。

表5-5 更新后台任务App.xaml.cs文件

```
using System.Threading.Tasks;
using MetroGrocer.Data;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Core;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace MetroGrocer {
   sealed partial class App : Application {
      private ViewModel viewModel;
      private Task locationTask;
      private CancellationTokenSource locationTokenSource;
      private Frame rootFrame;

public App() {
   this.InitializeComponent();

   viewModel = new ViewModel();

   // ...test data removed for brevity...

   this.Suspending += OnSuspending;
   this.Resuming += OnResuming;
```

```
    StartLocationTracking();
    }
    protected override void OnLaunched(LaunchActivatedEventArgs args) {
        // Create a Frame to act navigation context and navigate to the first page
        rootFrame = new Frame();
        rootFrame.Navigate(typeof(Pages.MainPage), viewModel);

        // Place the frame in the current Window and ensure that it is active
        Window.Current.Content = rootFrame;
        Window.Current.Activate();
    }
    private void OnResuming(object sender, object e) {
        viewModel.Location = "Unknown";
        StartLocationTracking();
    }

    void OnSuspending(object sender, SuspendingEventArgs e) {
        StopLocationTracking();
        SuspendingDeferral deferral = e.SuspendingOperation.GetDeferral();
        locationTask.Wait();
        deferral.Complete();
    }

private void StartLocationTracking() {
    locationTokenSource = new CancellationTokenSource();
    CancellationToken token = locationTokenSource.Token;

    locationTask = new Task(async () => {
        while (!token.IsCancellationRequested) {
            string locationMsg = await Location.TrackLocation();

            rootFrame.Dispatcher.Invoke(CoreDispatcherPriority.Normal,
                (sender, context) => {
                    viewModel.Location = locationMsg;
                }, this, locationMsg);
            token.WaitHandle.WaitOne(5000);
            }
        });
        locationTask.Start();
    }

private void StopLocationTracking() {
    locationTokenSource.Cancel();
        }
    }
}
```

这一类的变化代表了两种不同的任务。首先是把追踪用户位置作为一个后台任务，这种变化是包含在 **StartLocationTracking** 和 **StopLocationTracking** 中的方法。因为没有精确的 TPL 的概念和特点，建议把这些方法作为黑盒子。重要的是，**StartLocationTracking** 启动一个后台任务（每 5 秒报告一次用户位置），**StopLocationTracking** 用来取消该任务。

那么如何用生命周期事件整合这个后台任务呢？应用程序启动或恢复事件是很容易的，只需采用简单的 StartLocationTracking 方法。

对于 Suspending 事件，必须确保应用程序被暂停前后台任务已完成。如果不这么做，就会产生两种风险。一种是应用程序恢复时造成数据的丢失；另一种是，在应用程序暂停使用期间，试图通过网络请求读取信息时，由于服务器关闭而导致错误。

为解决这个问题，SuspendingEventArgs.SuspendingOperation.GetDeferral 方法会告诉 Windows 在运行时，如果还没有准备好暂停应用程序，就要多预留一点时间。这其中会出现一个小窗口，等待任务完成。当准备好应用程序暂停时，GetDeferral 会返回一个 SuspendingDeferral 对象，点击完成即可。

要求延期授予一个额外的 5 秒暂停准备。这听起来可能并不多，但它却能使 window 在众多压力下，有足够的时间帮助应用程序使用系统资源。

注意：在客户预览时，Windows 将在允许的 5 秒内终止 Metro 应用程序。这不是对延时对象的完成操作。在最终版本完成前这个程序还会被改变，但是值得密切关注。

5.2.5. 调度 UI 更新

表5-5中其他方面值得注意的是

```
...
rootFrame.Dispatcher.Invoke(CoreDispatcherPriority.Normal,
   (sender, context) => {
viewModel.Location = locationMsg;
   }, this, locationMsg);
...
```

Metro 只允许从指定的线程更新 UI 控件，该路线是用来实例化应用程序的。假如想在后台任务中更新视图模型，这个事件是作为更新结果发生的，最终会因更新数据绑定和对用户显示位置而产生错误线程。这种异常的结果将会做详细介绍：

应用程序调用接口将为不同的线程编组

必须确保使用的是正确的线程分派更新。然而，从我们的 App 类中找不出可用的分派功能。为了解决这个问题，可以使用框架控件类创建的 Dispatcher 构造，这就是为什么把 rootFrame 从固有量变为一个实例变量。

5.2.6. 测试后台任务

接下来就是测试，测试后台任务是否能正确啮合生命周期事件。最简单的方法是使用模拟器，它支持模拟的位置数据。

开始是在模拟器中定义一个位置（点击模拟器窗口右侧的一个按钮打开设置位置对话框，可以在其中输入一个位置） 。

一旦指定一个位置，启动应用程序（记住这样做不能使用调试器）。几秒钟后，在顶部应用程序窗口会显示位置信息，如图 5-3 所示。

提示：此例使用的是帝国大厦的坐标。如果你想要做同样的，就在“设置位置”对话框中指定，纬度 40.748、经度-73.98。

图 5-3 显示位置信息

切换到桌面，并使用任务管理器来监视应用程序，直到它被暂停。应用程序暂停后，使用模拟器中的“设置位置”对话框改变位置。

提示：您可能需要授予模拟器访问您的位置数据和应用程序的权限。自动化过程会检查所需的设置，并提示您对您的系统配置做必要的变化。

5.3. 实施合同

暂停和恢复，不是唯一的生命周期事件。另外还有一些 Windows 使用的事件也作为系统合同的一部分，这可以让您的应用程序与操作系统的其余部分获得更紧密的集成。这一节将证明搜索合同的用途，它告诉 Windows ，Metro 应用程序非常乐意并能够使用平台范围内的搜索功能的。

5.3.1. 宣布支持合同

对合同实施的第一步是更新清单。打开 package.appxmanifest 文件并切换到“声明”选项卡。如果打开可用声明菜单，你可以在你看到的合同清单中声明支持名单。选择搜索并点击“添加”按钮。搜索合同上会出现支持的声明清单。最好不要忽视合同的性质，否则会让你根据另一个应用程序的搜索合同履行你的义务。

5.3.2. 实现搜索功能

搜索合同的目的是为了在应用程序里，通过搜索功能把操作系统和搜索系统连接上。对于示例应用程序，要对购物清单上的项目遍历处理搜索需求，并选择用户正在寻找的包含字符串的第一个请求。这不是最先进的搜索方式，但它却可以让你专注于合同，不会因创造许多新的代码处理搜索而陷入困境。如表 5-6 所示，对 ViewModel 类添加了一个 SearchAndSelect 方法

。

表 5-6 添加搜索支持视图模型

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;

namespace MetroGrocer.Data {
   public class ViewModel : INotifyPropertyChanged {
      private ObservableCollection<GroceryItem> groceryList;
      private List<string> storeList;
      private int selectedItemIndex;
      private string homeZipCode;
      private string location;

      public ViewModel() {
            groceryList = new ObservableCollection<GroceryItem>();
                storeList = new List<string>();
            selectedItemIndex = -1;
            homeZipCode = "NY 10118";
            location = "Unknown";
      }

      public void SearchAndSelect(string searchTerm) {
         int selIndex = -1;
         for (int i = 0; i < GroceryList.Count; i++) {
            if (GroceryList[i].Name.ToLower().Contains(searchTerm.ToLower())) {
            selIndex = i;
            break;
         }
      }
   SelectedItemIndex = selIndex;
   }

   // ...properties removed for brevity...

   public event PropertyChangedEventHandler PropertyChanged;
```

```
  private void NotifyPropertyChanged(string propName) {
    if (PropertyChanged != null) {
      PropertyChanged(this, new PropertyChangedEventArgs(propName));
      }
    }
  }
}
```

这种方法将通过用户正在寻找的字符串。它在购物清单中搜索条目，并会对它找到的第一个匹配项做出选择，如果没有匹配项它会输出-1。由于 SelectedItemIndex 权属是可见的，寻找项目将选择它，并在应用程序布局中显示其详情。

如果想用ListPage 中的ListView控件来选择匹配项目，就要对ListPage.xaml.cs代码隐藏类做一个微小的变化，如表5-7所示。

表5-7 确保正确显示推选

```
...
protected override void OnNavigatedTo(NavigationEventArgs e) {

  viewModel = (ViewModel)e.Parameter;

  ItemDetailFrame.Navigate(typeof(NoItemSelected));
  viewModel.PropertyChanged += (sender, args) => {
    if (args.PropertyName == "SelectedItemIndex") {
      groceryList.SelectedIndex = viewModel.SelectedItemIndex;
      if (viewModel.SelectedItemIndex == -1) {
          ItemDetailFrame.Navigate(typeof(NoItemSelected));
          AppBarDoneButton.IsEnabled = false;
        } else {
          ItemDetailFrame.Navigate(typeof(ItemDetail), viewModel);
          AppBarDoneButton.IsEnabled = true;
        }
    }
  };
}
...
```

5.3.3. 针对搜索生命周期事件

应用程序类，通过提供能覆盖全局的方法使合同实施非常容易。表 5-8 显示了 OnSearchActivated 的实施方法，当用户的目标搜索应用程序时调用。

表 5-8 回应搜索

```
using System.Threading;
using System.Threading.Tasks;
using MetroGrocer.Data;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Core;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace MetroGrocer {
  sealed partial class App : Application {
    private ViewModel viewModel;
    private Task locationTask;
    private CancellationTokenSource locationTokenSource;
```

```
    private Frame rootFrame;

    public App() {
t       his.InitializeComponent();

        viewModel = new ViewModel();

        // ...test data removed for brevity...

        this.Suspending += OnSuspending;
        this.Resuming += OnResuming;

        StartLocationTracking();
  }

  protected override void OnLaunched(LaunchActivatedEventArgs args) {
    // Create a Frame to act navigation context and navigate to the first page
    rootFrame = new Frame();
    rootFrame.Navigate(typeof(Pages.MainPage), viewModel);

    // Place the frame in the current Window and ensure that it is active
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
  }

  protected override void OnSearchActivated(SearchActivatedEventArgs args) {
    viewModel.SearchAndSelect(args.QueryText);
    }

    //...other methods removed for brevity
  }
}
```

这是所有需要的满足搜索合同的义务。为普及 OnSearchActivated 方法，我们针对用户对 Windows 增加了搜索应用程序功能。

5.3.4. 测试搜索合同

为了测试合同，启动示例应用程序。启动它用不用调试器都无所谓。找到符条并选择搜索图标。这个示例应用程序将被作为自动搜索的目标。开始搜索，并开始打字。

当您单击文本框右侧的“搜索”按钮，Windows 会调用搜索合同，并把查询的字符串反馈给示例应用程序。如果你想搜索相关的东西，输入"hot"（这样你的搜索将会在购物清单中匹配对热狗的项目），然后单击按钮即可。你会看到类似图 5-4 的内容。

这个合同的做法很让人喜欢。它能很轻易的实现搜索合同，并且仅用几行代码就能很轻易的集成到 Windows 中。你可以比我做的更多并且完全定制你的应用程序处理和回应搜索的方式。

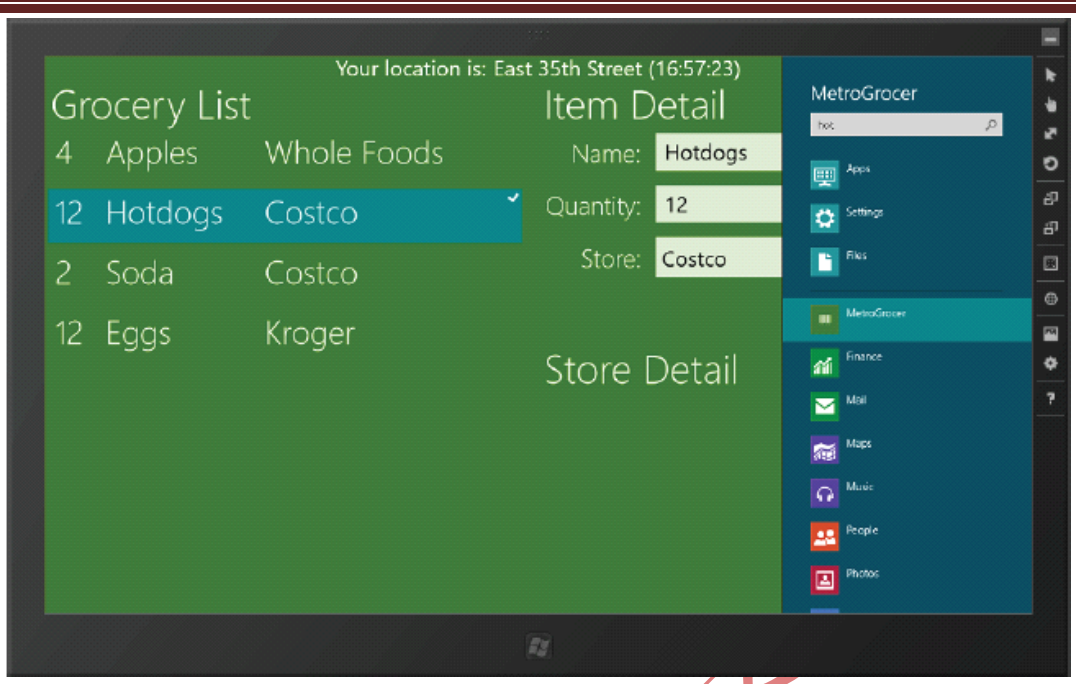

图 5-4 合同的搜索应用程序

5.4. 总结

本章介绍了如何使用生命周期事件应对 Windows 管理 Metro 应用程序的方法，讲述了关键事件并告诉你如何应对它们，以确保您的应用程序能够正确接收和处理它们。

特别注意，当一个应用程序暂停时必须把后台任务打包干净。本章向您介绍了通过要求暂停延期来控制这个过程，允许额外的几秒钟，以尽量减少潜在的错误或由于应用程序恢复时数据丢失造成的风险。

最后，本章向您介绍了生命周期事件如何让你履行合同，绑定Metro应用到更广泛的Windows平台，并且它很容易满足合同指定的义务。本章介绍的搜索合同不够全面，希望您有时间更深一步探讨。您实现的合同越多，您的应用程序就会与更多的Windows的终止操作和其它集成的Metro应用程序整合在一起。

本书介绍的核心系统功能将迅速打开Metro应用程序开发市场。本书介绍你俩如何使用数据绑定，如何使用主要结构控制，如何处理抢购和填补布局，如何定制你的应用程序，和这一章中介绍的如何控制应用程序的生命周期。有了这些作为您的技能基础，你将能够创造出丰富有价值的Metro应用，并先声夺人得到Windows 8的最终版本。

祝在Metro开发项目中获得成功！