1 WPF及Silverlight版FlexGrid概述

WPF及Silverlight版FlexGrid是一个具有轻量级，灵活对象模型的DataGrid控件。基于流行的WinForms版本进行建模，WPF及Silverlight版FlexGrid提供了许多统一的功能，比如说非绑定模式，灵活的单元格合并，以及多单元格的行列标题。

1.1 WPF及Silverlight版ComponentOne Studio 帮助

入门

关于安装WPF及Silverlight版ComponentOne Studio，许可，信息，技术支持，命名空间以及通过控件创建一个工程，请访问WPF版本入门或Getting Started Silverlight版本入门。

2 XAML快速参考

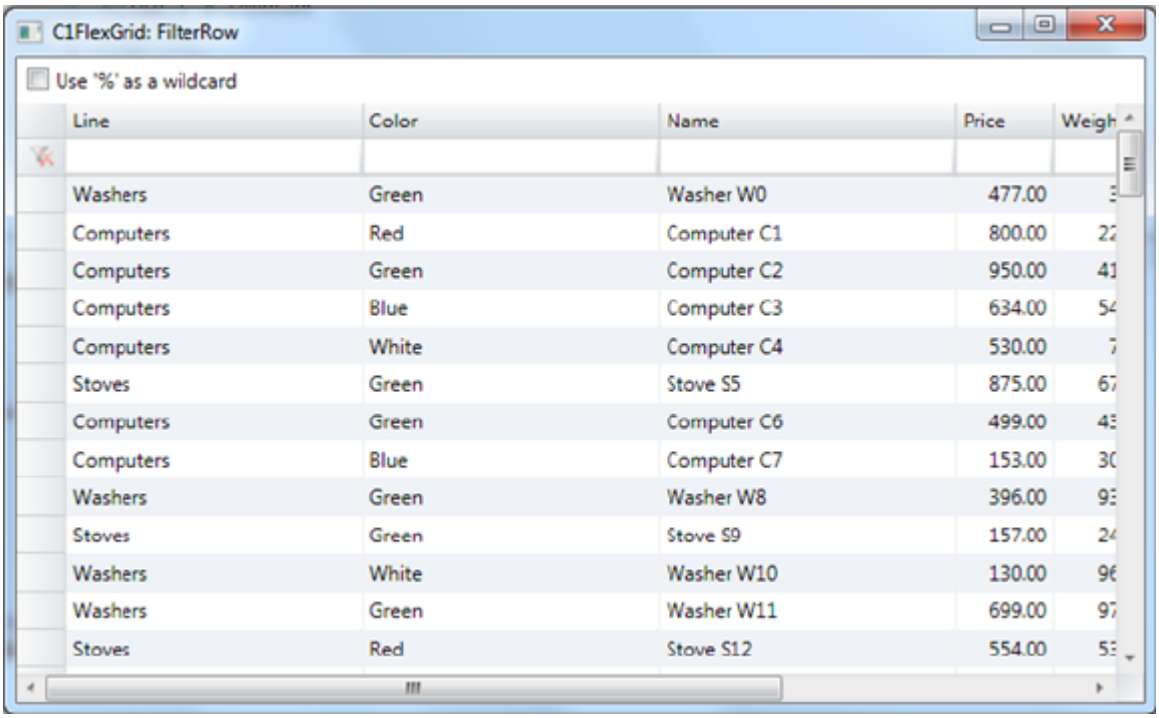
本主题将提供一个用于创建一个C1FlexGrid控件的XAML的简要概述。

在开始开发之前，须在根元素标签上添加一个c1的命名空间声明。这一点在WPF以及Silverlight下是相同的。但有所不同的是放置代码的位置。在WPF版本中，目标位置是在Window类中，而在Silverlight中，则是位于UserControl类。

XAML

```
xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
```

这是从FilterRow示例中抽取的C1FlexGrid的示例：

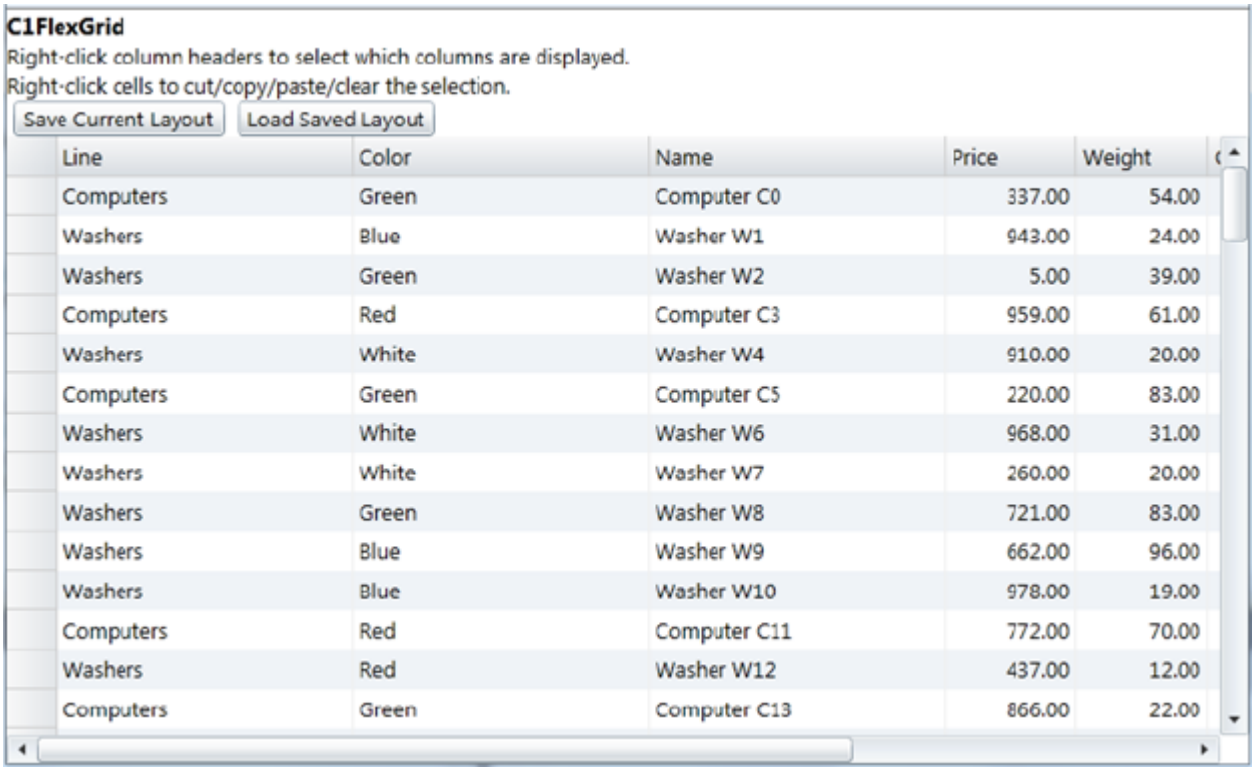


以下是该示例可以在WPF下使用的XAML：

WPF XAML

```
<Window x:Class="FilterRow.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        Title="C1FlexGrid: FilterRow" Height="350" Width="700"
WindowStartupLocation="CenterScreen" >
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <CheckBox Content="Use '%' as a wildcard" Margin="6" Click="CheckBox_Click"
/>
        <c1:C1FlexGrid Name="_flex" KeyActionTab="MoveAcross" Grid.Row="1"/>
    </Grid>
</Window>
```

这是从ColumnPicker 示例中抽取的C1FlexGrid的示例：



Line	Color	Name	Price	Weight
Computers	Green	Computer C0	337.00	54.00
Washers	Blue	Washer W1	943.00	24.00
Washers	Green	Washer W2	5.00	39.00
Computers	Red	Computer C3	959.00	61.00
Washers	White	Washer W4	910.00	20.00
Computers	Green	Computer C5	220.00	83.00
Washers	White	Washer W6	968.00	31.00
Washers	White	Washer W7	260.00	20.00
Washers	Green	Washer W8	721.00	83.00
Washers	Blue	Washer W9	662.00	96.00
Washers	Blue	Washer W10	978.00	19.00
Computers	Red	Computer C11	772.00	70.00
Washers	Red	Washer W12	437.00	12.00
Computers	Green	Computer C13	866.00	22.00

以下是该示例可以在Silverlight下使用的XAML

Silverlight XAML

```
<UserControl x:Class="ColumnPicker.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="Segoe UI" FontSize="13"
    d:DesignHeight="300" d:DesignWidth="600">
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <StackPanel Orientation="Vertical">
            <TextBlock Text="C1FlexGrid" FontSize="14" FontWeight="Bold"/>
            <TextBlock Text="Right-click column headers to select which columns are
displayed." />
            <TextBlock Text="Right-click cells to cut/copy/paste/clear the
selection." />
            <StackPanel Orientation="Horizontal">
                <Button Content="Save Current Layout" Click="SaveLayout_Click"
Margin="4 0" Padding="6 2"/>
                <Button Content="Load Saved Layout" Click="LoadLayout_Click"
Margin="4 0" Padding="6 2"/>
            </StackPanel>
        </StackPanel>
        <c1:C1FlexGrid Name="_flex" Grid.Row="1" />
    </Grid>
</UserControl>
```

3 创建C1FlexGrid

添加一个C1FlexGrid控件至您的应用程序，所需步骤和添加任何的自定义控件严格一致。在此方面C1FlexGrid没有任何特别。您将从向工程添加一个到C1FlexGrid所在程序集的引用开始，接下来就是通过XAML添加该控件：

WPF XAML

```
<Window x:Class="FilterRow.MainWindow"
        ...
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        ... >
    <Grid x:Name="LayoutRoot">
        <c1:C1FlexGrid/>
    </Grid>
</Window>
```

Silverlight XAML

```
<UserControl x:Class="MainTestApplication.MainPage"
             ...
             xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
             ... >
  <Grid x:Name="LayoutRoot">
    <c1:C1FlexGrid/>
  </Grid>
</UserControl>
```

您也可以在代码中创建C1FlexGrid：

C# code

```
var fg = new C1.Silverlight.FlexGrid.C1FlexGrid();
LayoutRoot.Children.Add(fg);
```

4 填充该Grid

在向应用程序添加了C1FlexGrid之后，像大多数的其他Grid产品一样，您可以通过ItemsSource属性填充该Grid。

ItemsSource属性需要传入一个实现了IEnumerable 接口的对象，但是在大多数情况下，您将使用更高层次的，实现了ICollectionView 接口的对象。ICollectoinView接口是Silverlight以及WPF主要的数据绑定接口（在WinForms平台下，该角色由IBindingList接口扮演）。
ICollectionView是一个丰富的接口。除了枚举数据项，它还同时提供了排序、过滤、分页、分组以及Currency管理的服务。

Silverlight提供了一个实现ICollectionView接口PagedCollectionView 类。PagedCollectionView的构造函数接受一个IEnumerable对象作为参数，并自动提供所有的ICollectionView服务。WPF提供类似的类型，比方说ListCollectionView。

例如，为了在一个C1FlexGrid Silverlight应用程序中显示一个客户对象的列表，请使用如下所示代码：

C#

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
_flexGrid.ItemsSource = view;
```

您也可以直接将grid绑定到客户列表。但是绑定到一个ICollectionView通常是一个较好的主意，因为它为应用程序保留了大量的数据配置相关信息，这些信息可以跨控件共享。

如果多个控件绑定到同一个ICollectionView对象，则它们将展示同样的视图。在一个控件中选择一个项目会自动更新所有其他绑定控件中的选择。过滤，分组，或排序，也被所有的控件绑定到相同的视图中。

上面的代码会导致Grid扫描数据源，并为每一个数据源中的项的公共属性生成列。您可以通过代码自定义这些自动生成的列。或者您可以干脆禁用自动生成列的功能，通过代码或者XAML自己生成这些列。

例如，以下XAML禁用了自动列生成并手动通过XAML指定列：

C1FlexGrid XAML

```
<!-- create columns on a C1FlexGrid -->
<fg:C1FlexGrid x:Name="_flexiTunes"
     AutoGenerateColumns="False" >
   <fg:C1FlexGrid.Columns>
     <fg:Column Binding="{Binding Name}" Header="Title"
         AllowDragging="False" Width="300"/>
     <fg:Column Binding="{Binding Duration}"
         HorizontalAlignment="Right" />
     <fg:Column Binding="{Binding Size}"
         HorizontalAlignment="Right" />
     <fg:Column Binding="{Binding Rating}" Width="200"
         HorizontalAlignment ="Center" />
   </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

这和您在Microsoft DataGrid 或者最初的ComponentOne DataGrid控件中完成相同的任务所需要的XAML类似：

DataGrid XAML

```
<!-- create columns on an MSDataGrid (or C1DataGrid) -->
<ms:DataGrid Name="_msSongs"
     AutoGenerateColumns="False" >
   <ms:DataGrid.Columns>
     <ms:DataGridTextColumn Binding="{Binding Name}" Header="Title"
         CanUserReorder="False" Width="300" />
     <ms:DataGridTextColumn Binding="{Binding Duration}" />
     <ms:DataGridTextColumn Binding="{Binding Size}" />
     <ms:DataGridTextColumn Binding="{Binding Rating}" Width="200" />
   </ms:DataGrid.Columns>
</ms:DataGrid>
```

正如你所看到的，语法是几乎完全相同的。注意，您可以将此绑定做为访问Grid的列集合的索引。例如，如果您希望通过代码将“Rating”列的宽度变为300像素，则可以这样添加代码：

C#

```
_flexiTunes.Columns["Rating"].Width = new GridLength(300);
```

这是C1FlexGrid 用户最熟悉的语法。当您使用控件的WinForms版本时，这些命令也完全相同。

5 非绑定模式

C1FlexGrid 被设计用来和ICollectionView数据源配合使用，充分利用其提供的功能。

但是它也可以工作在非绑定模式。如果您仅向Grid添加行和列，则您可可以通过下面所示熟悉的索引表示法获取或设置单元格的值：

C#

```
// 为非绑定模式的Grid添加行和列
for (int i = 0; i < 20; i++)
{
  fg.Columns.Add(new Column());
}
for (int i = 0; i < 500; i++)
{
  fg.Rows.Add(new Row());
}

// 填充非绑定Grid
for (int r = 0; r < fg.Rows.Count; r++)
{
  for (int c = 0; c < fg.Columns.Count; c++)
  {
    fg[r, c] = string.Format("cell [{0},{1}]", r, c);
  }
}
```

索引表示法应当也是C1FlexGrid用户所熟悉的。这是和该控件的WinForms版本所实现的同样的表示方法。可以指定行和列索引、行索引和列名称、或行索引和列对象指定单元格。

索引表示法针对绑定和非绑定模式同样有效。在绑定模式中，将获取或者应用位于数据源中各个项目的数据。在绑定模式下，数据是由Grid内部存储。

WinForms 以及Silverlight及WPF版本C1FlexGrid控件之间一个最重要的区别在于，该控件的WinForms版本，索引包含固定的行和列。在Silverlight及WPF版，固定的行和列不包括在内。

下图显示了在Grid的WinForms版本中，单元格索引的方案：

0,0	**0,1**	**0,2**	**0,3**
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

下图显示了在Silverlight及WPF版Grid中使用的新的单元格索引方案：

	0,0	0,1	0,2
	1,0	1,1	1,2
	2,0	2,1	2,2

新的标注方式使得索引更加容易使用，因为索引将匹配数据项的索引值（第零行包含索引值为零的项），同时列数匹配显示的属性个数。

唯一的缺点是，您需要一种新的方法来访问固定的单元格的内容，而不是使用标准的索引方案。这种新的方法由额外的叫做RowHeaders以及ColumnHeaders的属性组成。

这些属性返回一个GridPanel类型的对象，该对象可以被看作具有其行和列的集合的“子-Grid”。
例如，您可以使用以下代码来自定义行标题：

C#

```
// 获得Grid的行标题
```

```
GridPanel rh = fg.RowHeaders;

// 向Grid添加一个新的固定列
rh.Columns.Add(new Column());

// 设置宽度和行标题内容
for (int c = 0; c < rh.Columns.Count; c++)
{
  //该列的宽度
  rh.Columns[c].Width = 60;
  for (int r = 0; r < rh.Rows.Count; r++)
  {
    // 该单元格的内容
    rh[r, c] = string.Format("hdr {0},{1}", r, c);
  }
}
```

注意这里GridPanel类提供了和主Grid一样的Rows和Columns集合，并支持一样的索引表示法。您可以使用和操作Grid内容区域单元格（可滚动区域）相同的技术自定义以及填充行头和列头。

6 使用FlexGrid

Silverlight及WPF版C1FlexGrid控件在相关的ComponentOne StudioV2 2010发布版中引入，自那时以来，该控件已经得到大幅改善。

6.1 选择

6.1.1 选择和选择模式

除了以表格形式显示数据外，大多数grid控件允许用户使用鼠标和键盘选择数据的一部分。

C1FlexGrid具有丰富的选择模型，由SelectionMode属性进行控制。以下选择模式可用：

- Cell: 选择对应单个单元格。
- CellRange: 选择对应于一个单元格区域（相邻单元格的一块区域）。
- Row: 选择对应于一个整行。
- RowRange: 选择对应于一段连续的整行。
- ListBox: 选择对应于一组行（不一定是连续的）。

默认的SelectionMode是CellRange，提供了一个用户熟悉的，类似于Excel操作风格的选择行为。基于行进行选择的选项在需要选择整个数据行而不是单个Grid单元格的场景下同样有用。

无论当前的选择模式是什么，Grid都通过Selection属性暴露了当前的选择。此属性获取或设置当前的选择作为一个CellRange的对象。

6.1.1.1 监控选择

一旦选择发生变化，可以是由于用户操作或者代码修改所引起，Grid将触发SelectionChanged事件，允许您处理新的选择结果。

例如，下面的代码可以监视选择并在选择更改时发送信息到控制台：

C#

```
void _flex_SelectionChanged(object sender, CellRangeEventArgs e)
{
  CellRange sel = _flex.Selection;
  Console.WriteLine("selection: {0},{1} - {2},{3}",
      sel.Row, sel.Column, sel.Row2, sel.Column2);
  Console.WriteLine("selection content: {0}",
      GetClipString(_flex, sel));
}
static string GetClipString(C1FlexGrid fg, CellRange sel)
{
  var sb = new System.Text.StringBuilder();
  for (int r = sel.TopRow; r <= sel.BottomRow; r++)
  {
    for (int c = sel.LeftColumn; c <= sel.RightColumn; c++)
    {
      sb.AppendFormat("{0}\t", fg[r, c].ToString());
    }
    sb.AppendLine();
  }
  return sb.ToString();
}
```

一旦选择发生变化，代码将列举表示当前选择的CellRange的坐标。

同时它还使用GetClipString方法输出选择范围的内容，该方法遍历选择的项目并使用Grid的索引获取选择范围中每一

个单元格的内容。关于索引，在本文档之前有所介绍。

请注意，GetClipString方法中的for循环使用单元格的TopRow，ButtomRow，LeftColumn以及RightColumn苏醒，而不是Row，Row2，Column以及Column2属性。

这是必须的，因为Row可能大于或者小于Row2，取决于用户如何执行该选择（在选择时向上或者向下拖拽鼠标）。

您可以很容易地通过Rows.GetDataItems方法从Selection中抽取大量有用的信息。该方法返回一个关联到一个CellRange的数据项的集合。一旦你有了这个集合，你可以使用LINQ进行提炼和总结有关选定项目的信息。

例如，可以考虑为一个绑定到Customer对象集合的Grid使用以下SelectionChanged事件处理的替代方案：

C#

```
void _flex_SelectionChanged(object sender, CellRangeEventArgs e)
{
  // 获取选定范围内的客户对象
  var customers =
    _flex.Rows.GetDataItems(_flex.Selection).OfType<Customer>();

  // 使用LINQ从选中的客户对象集合中抽取信息
  _lblSelState.Text = string.Format(
    "{0} items selected, {1} active, total weight: {2:n2}",
    customers.Count(),
    (from c in customers where c.Active select c).Count(),
    (from c in customers select c.Weight).Sum());
}
```

注意，该代码使用OfType运算符将选中的数据项转换为Customer类型。一旦该步骤完成，代码将使用LINQ获取一个总数，一个“active”状态的客户数，以及选中客户的总权重。LINQ是完成这类工作的完美工具。它是灵活，富有表现力的，紧凑而且高效。

6.1.1.2 通过代码选择单元格和对象

Selection属性是可读写的，所以你可以使用代码选择单元格范围。您还可以使用Select方法进行选择，就像在WinForms版本的C1FlexGrid控件那样。Select方法允许您选择单元格或范围，并可选地滚动新的选择范围到可视视图，以便用户可以看到它。

例如，为了选中Grid的第一个单元格并确保它位于用户可见范围，应使用以下代码：

C#

```
// 选择第零行，第零列，并确保该单元格可见
fg.Select(0, 0, true);
```

选择相关的方法均基于行和列的索引进行工作。但是您也可以用来基于单元格内容进行选择。例如，以下代码选择在Grid的“Name”列中，第一个包含指定字符串的行。

C#

```
bool SelectName(string name)
{
  // 在“Name”列查找包含指定字符串的行
  int col = _flexGroup.Columns["Name"].Index;
  int row = FindRow(_flex, name, _flex.Selection.Row, col, true);
  if (row > -1)
  {
    _flex.Select(row, col);
    return true;
  }
  // 未查询到……
  return false;
}
```

该代码使用的FindRow辅助方法定义如下：

C#

```
// 在一个指定的列中查找包含某个文本的行
int FindRow(C1FlexGrid flex, string text,
            int startRow, int col, bool wrap)
{
  int count = flex.Rows.Count;
  for (int off = 0; off <= count; off++)
  {
    // 搜索到底部且不允许循环查找？立即退出
    if (!wrap && startRow + off >= count)
    {
      break;
    }

    // 从行中获取文本
    int row = (startRow + off) % count;
    var content = flex[row, col];

    // 如果查找到匹配项则返回匹配行的索引
    if (content != null &&
        content.ToString().IndexOf(text,
        StringComparison.OrdinalIgnoreCase) > -1)
    {
      return row;
    }
  }

  // 未查询到……
  return -1;
}
```

该FindRow方法实现了在C1FlexGrid的WinForms版本中提供的FindRow方法的功能。此方法没有内置在Silverlight或WPF版本的Grid产品中，以使得控件所占用的尺寸尽可能的小。该方法在给定的列中搜索指定的字符串，从指定的行开始搜索，并可选地在搜索到末尾仍未查找到时返回到行首开始搜索。这在许多场景下已经具有足够的使用灵活性。

另一个常见的选择场景是在数据源中选择一个特定对象的情况。你的第一反应可能会查找并使用PagedCollectionView.IndexOf 方法从数据源集合中查找对象的索引，然后使用索引选择行。这种方法的问题是，它只有在数据没有进行分组时能够正常工作。如果已经将数据进行了分组，则分组行也将进行计数，这将导致数据源中的项目的索引和Grid中的行索引不匹配。

解决这种问题的最简单的方式是枚举每一行，并将您搜索的项目和每一行的DataItem属性进行比较。下面的代码演示了如何做到这一点：

C#

```
var customer = GetSomeCustomer;

#if false // ** 千万不要使用这种方式，因为它将在存在分组的情况下无法正常工作

  int index = view.IndexOf(customer);
  if (index > -1)
  {
    _flex.Select(index, 0);
  }

#else // 这是在Grid中搜索对象的安全方式

  for (int row = 0; row <= _flex.Rows.Count; row++)
  {
    if (row.DataItem == customer)
    {
      _flex.Select(row, 0);
```

```
            break;
        }
    }
#endif
```

6.1.1.3 自定义选择显示

C1FlexGrid提供两种功能，允许您自定义选择高亮显示给最终用户的方式。

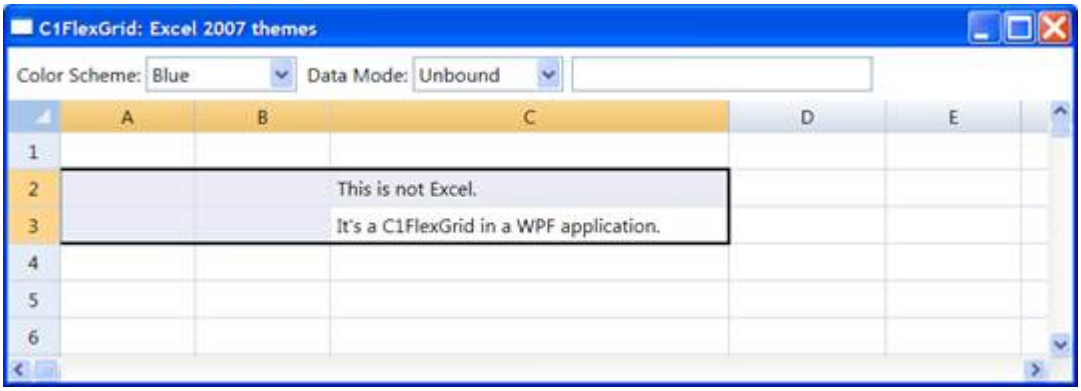
Excel-风格的Marquee

如果您设置了ShowMarquee属性的值为True，Grid将在选择区域自动绘制一个矩形框，使得非常容易的可以发现选择区域。默认情况下，Marquee是一个两个像素宽的黑色矩形框，但是您可以通过Marquee属性对其进行定制。

选中的单元格Header

如果您指定了自定义的画刷对象至Grid的ColumnHeaderSelectedBackground以及RowHeaderSelectedBackground属性，则Grid将高亮显示关联到选中单元格的Header，使得最终用户可以非常容易地确认包含选择区域的行和列的位置。

将这些属性综合在一起使用，使得您可以实现一个具有熟悉的Excel外观和使用习惯的Grid。下面的图像显示了这样的一个例子：



C1FlexGrid设计器具有一个允许您选择类似于Excel的预定义配色方案（Blue，Silver，Black）的上下文菜单。您可以容易地复制由设计器生成的XAML至可重复使用的样式资源。

6.2 编辑

6.2.1 自动完成并映射列

自动完成和映射列由一个内置的叫做ColumnValueConverter的类实现。这个类涉及三种常见的绑定场景：

自动完成独占模式（列表框风格的编辑）

列只能取有限的特定值。例如，你有一个“Country”列，为字符串类型，同时具有一个国家名称的列表。用户应该从列表中选择一个国家，而不被允许输入的任何国家的名称不在此列表中。

您仅需两行代码即可处理这个场景：

C#

```
var c = _flexEdit.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), true);
```

ColumnValueConverter构造器的第一个参数提供了一个合法值的列表。第二个参数确定是否最终用户允许输入在此列表中不存在的值（在此示例中，用户不允许这么做）。

自动完成非独占模式（组合框式编辑）

列具有一些共通的值选项，但是同样也可以接受其它值。例如，你有一个“Country”列，为字符串类型，并希望提供一个常见的国家名称列表供最终用户容易地进行选择。但在这种情况下，用户也应该允许键入在列表中不存在的值。

你仍然仅需要两行行代码即可处理这个场景：

C#

```
var c = _flexEdit.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), false);
```

和上一个示例一样，ColumnValueConverter构造器的第一个参数提供一组合法值的列表。在这个场景下，第二个参数决定了该列表是非排他性的，因此用户可以输入该列表中不存在的值。

使用一个键值字典自动完成

列将包含键值而不是实际的值。例如，该列可能包含一个整数，表示一个国家标识，但用户应该看到并编辑相应的国家名称。下面的代码演示了如何处理这个场景：

C#

```
// 建立键-值对应关系字典
var dct = new Dictionary<int, string>();
foreach (var country in GetCountryNames())
{
  dct[dct.Count] = country;
}

// 获取列
var c = _flexEdit.Columns["CountryID"];

// 创建并为值字典分配Converter
c.ValueConverter = new ColumnValueConverter(dct);

// 将列靠左对齐
c.HorizontalAlignment = HorizontalAlignment.Left;
```

6.2.2 数据映射列

数据映射列包含键而不是实际值。例如，该列可能包含一个整数，表示一个国家标识，但用户应该看到并编辑相应的国家名称。

这种情况需要一点点的代码行：

C#

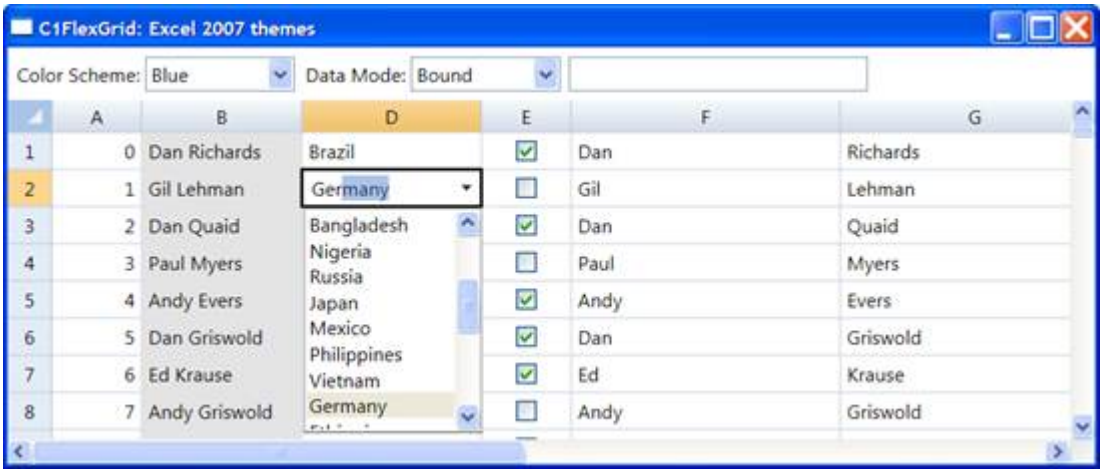
```
// 建立键-值字典
var dct = new Dictionary<int, string>();
foreach (var country in GetCountryNames())
{
  dct[dct.Count] = country;
}

// 指定字典至列
var c = _flexEdit.Columns["CountryID"];
c.ValueConverter = new ColumnValueConverter(dct);
c.HorizontalAlignment = HorizontalAlignment.Left;
```

代码首先是构建一个映射国家标识值（整数）到国家名称（字符串）的字典。

然后使用词典构建ColumnValueConverter 并指定转换器至列的ValueConverter属性，就像之前的示例那样。
用户可以选择在字典里现有的国家，且不允许输入任何未经映射的值。

最后，该代码将该列的对齐方式设置为左对齐。由于该列实际上包含整数值，它将默认为右对齐。但是，因为我们现在需要显示国家的名字，在这里左对齐是一个更好的选择。

下图显示编辑器的外观，同时从列表中选择一个值。注意这里编辑器是如何支持自动完成的，当用户键入“Ger”下拉框将自动选择有效的选项“Germany”（而不是“Guatemala”，或者“Eritrea”，再或者“Romania”）。



6.2.3 使用自定义编辑器

C1FlexGrid提供两种内置的编辑器：一个复选框用于编辑布尔类型的值，以及一个C1FlexComboBox，用于使用上面所述的自动完成功能扩展常规的TextBox。

您可以使用和该文档中之前描述的创建自定义单元格相同的机制创建并使用您自己的编辑器。

- 实现一个自定义的CellFactory 类并重写CreateCellEditor 方法创建并绑定您的编辑器至底层数据的值。
- 使用XAML为需要自定义编辑器的列指定CellEditingTemplate。

6.2.4 配置编辑器

无论您使用的是内置的或自定义的编辑器，你可以利用PrepareCellForEdit 事件在编辑器激活之前配置该编辑器。例如，下面的代码将更改编辑器的选择状态为蓝色底色，以黄色显示：

C#

```
// 挂载事件处理函数
_grid.PrepareCellForEdit += _grid_PrepareCellForEdit;

// 通过修改选择的外观自定义编辑器
void _grid_PrepareCellForEdit(object sender, CellEditEventArgs e)
{
  var b = e.Editor as Border;
  var tb = b.Child as TextBox;
  tb.SelectionBackground = new SolidColorBrush(Colors.Blue);
  tb.SelectionForeground = new SolidColorBrush(Colors.Yellow);
}
```

6.3 分组

6.3.1 分组数据（使用ICollectionView）

ICollectionView接口包含对分组的支持，这将允许您创建分层的数据视图。例如，在上面的示例中，如果要对客户按照不同的国家和城市进行分组，您只需要简单地将生成Grid数据的部分修改为以下：

C#

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
using (view.DeferRefresh())
{
  view.GroupDescriptions.Clear();
  view.GroupDescriptions.Add(new PropertyGroupDescription("Country"));
  view.GroupDescriptions.Add(new PropertyGroupDescription("Active"));
}
_flexGrid.ItemsSource = view;
```

表达式 using(view.DeferRefresh()) 为可选。它通过挂起来自于数据源的通知直至全部的分组设置完毕，以提高性能。

下图显示执行的结果：

数据项按照不同的国家以及其活动的区域进行分组。用户可以单击分组Header上的图标以折叠或者展开分组，就像他们在操作一个TreeView控件一样。

如果您希望在Grid层面禁用分组，请设置Grid的GroupRowPosition属性的值为GroupRowPosition.None（其他可用的选项为AboveData以及BelowData）。

由ICollectionView类提供的分组机制简单但功能强大。每一个级别的分组由PropertyGroupDescription对象定义。该对象允许您选择用于进行分组的属性，以及一个ValueConverter用来决定如何在分组时使用这个属性。

例如，如果我们想按国家的首字母来分组，则可以简单地修改代码如下：

C#

```
List<Customer> list = GetCustomerList();
PagedCollectionView view = new PagedCollectionView(list);
using (view.DeferRefresh())
{
  view.GroupDescriptions.Clear();
  view.GroupDescriptions.Add(new PropertyGroupDescription("Country"));
  view.GroupDescriptions.Add(new PropertyGroupDescription("Active"));
  var gd = view.GroupDescriptions[0] as PropertyGroupDescription;
  gd.Converter = new CountryInitialConverter();
}
_flexGrid.ItemsSource = view;
```

CountryInitialConverter类实现了IValueConverter接口。它返回国家名称的第一个英文字母，使用该字母进行分组而不再是使用完整的国家名称。

C#

```
// 按照首字母对国家进行分组的转换器
class CountryInitialConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    return ((string)value)[0].ToString().ToUpper();
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

在完成这个小小的变化之后，客户将按照所在国家的第一个字母而不再是按照国家名称进行分组：

注意，该分组行将显示该分组的信息（依照分组的属性以及其值，以及其中包含的项目个数）。

为定制此信息，可以创建一个新的IValueConverter 类并将其指定给Grid的GroupHeaderConverter 属性。

例如，默认的分组Header转换器（显示如图片所示信息的那个默认转换器）的实现如下所示：

C#

```
// 用作格式化显示分组标题的类
public class GroupHeaderConverter : IValueConverter
{
  public object Convert(object value,
    Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    var gr = parameter as GroupRow;
    var group = gr.Group;
    if (group != null && gr != null && targetType == typeof(string))
    {
      var desc = gr.Grid.View.GroupDescriptions[gr.Level] as
                 PropertyGroupDescription;
      return desc != null
        ? string.Format("{0}: {1} ({2:n0} items)",
                 desc.PropertyName, group.Name, group.ItemCount)
        : string.Format("{0} ({1:n0} items)",
```

```
                group.Name, group.ItemCount);
    }
    return value;
  }
  public object ConvertBack(object value, Type targetType,
    object parameter,
    System.Globalization.CultureInfo culture)
  {
    return value;
  }
}
```

6.3.2 通过C1FlexGridGroupPanel管理分组

C1FlexGridGroupPanel是一个新控件，提供了在C1FlexGrid控件中管理分组的UI。

C1FlexGridGroupPanel控件目前在Silverlight以及WPF版本中可用。它实现在一个单独的程序集中，这取决于平台。

- C1.WPF.FlexGrid.GroupPanel.4.dll
- C1.Silverlight.FlexGrid.GroupPanel.5.dll

6.3.3 使用C1FlexGridGroupPanel控件

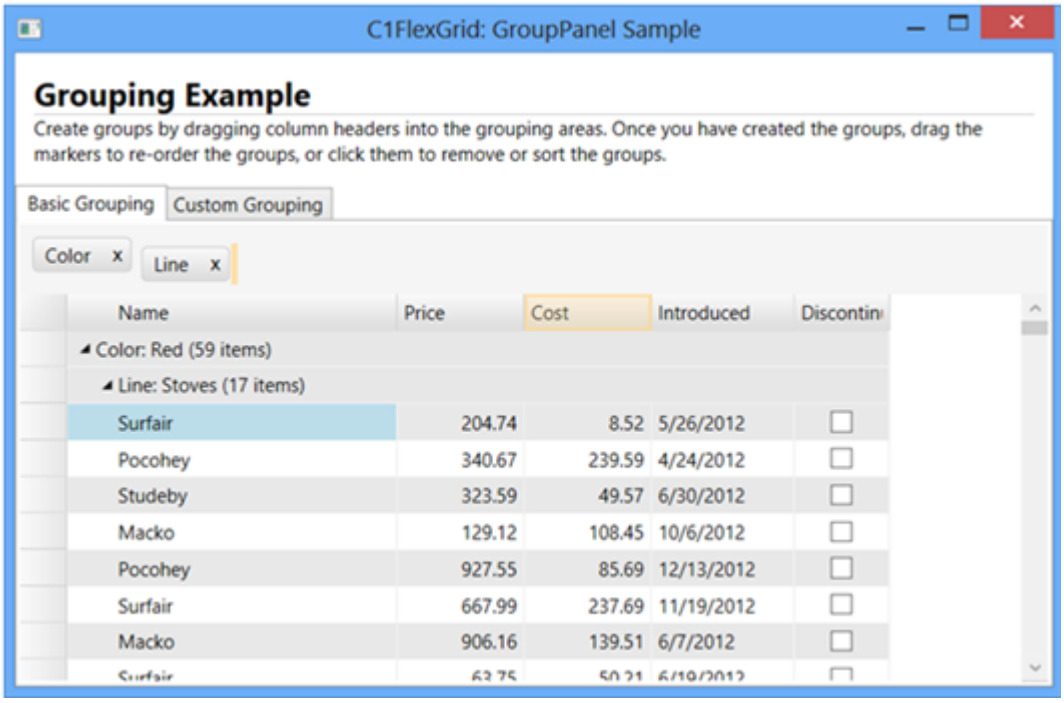
为使用C1FlexGridGroupPanel控件，添加该控件在窗体上的某个C1FlexGrid控件上方，并设置其FlexGrid属性，使得其引用该C1FlexGrid用来显示数据。例如：

XAML

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>

  <c1:C1FlexGridGroupPanel
    Background="WhiteSmoke"
    FlexGrid="{Binding ElementName=_flex}"/>
  <c1:C1FlexGrid x:Name="_flex" Grid.Row="1" />
</Grid>
```

该段XAML是非常简单的。C1FlexGridGroupPanel不需要设置样式，因为它将从控制的C1FlexGrid获取所需的属性设置。如果您改变了列Header的背景，前景或者字体，则C1FlexGridGroupPanel 将自动地使用这些元素来渲染分组标记，使得其看起来和列Header一样。

该段XAML将产生下图所示结果：



该图像显示了按照“Color”以及“Line”进行分组的产品的C1FlexGrid。该分组通过拖拽Grid列至分组区操作进行创建。

在本图像中，“Cost”列高亮显示，因为它正在被拖到分组区域。位于“Line”分组标记右侧的橙色光标指示新分组插入的位置。

您可以通过设置DragMarkerColor属性改变拖拽标记的颜色（图中所示为橙色）。
一旦一个分组被创建，则关联的列默认将被隐藏。这是一个可选的功能，您可以通过设置HideGroupedColumns 属性的值为false关闭该设置。

您可以单击分组标记以便按照升序或者降序的方式对分组中的数据进行排序。按下Control键并单击分组Header以移除排序状态。这是和在其他平台下的C1FlexGrid相同的行为。

您也可以在分组区域拖拽分组标记以便重新排布分组，或者将其拖拽回Grid区域已撤销分组，并将该列恢复到任意位置。分组标记也具有关闭按钮（“×”），您可以单击以删除该分组。

C1FlexGridGroupPanel通过用作Grid数据源的ICollectionView接口管理分组。对分组所做的全部改变对绑定到相同的ICollectionView的控件可见。注意，这意味着分组特性不能用于非绑定使用场景。

当C1FlexGridGroupPanel创建了一个新的分组时，它将触发一个PropertyGroupCreated事件，允许应用程序自定义该新建的分组。该事件通常用来为新创建的分组指定自定义转换器。例如：

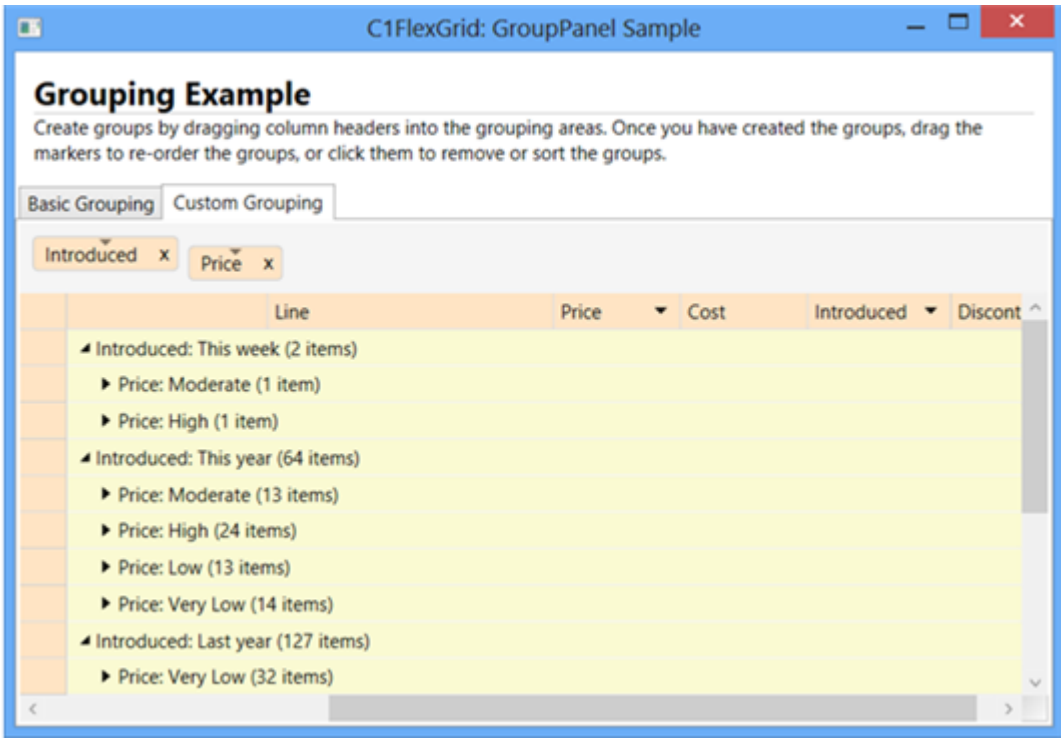
C#

```
/// <summary>
/// 自定义由C1FlexGridGroupPanel创建的分组描述。
/// </summary>
void _groupPanel_PropertyGroupCreated(object sender, PropertyGroupCreatedEventArgs
e)
{
  var pgd = e.PropertyGroupDescription;
  switch (pgd.PropertyName)
  {
    case "Introduced":
      pgd.Converter = new DateTimeGroupConverter();
      break;
    case "Price":
      pgd.Converter = new AmountGroupConverter(1000);
      break;
    case "Cost":
      pgd.Converter = new AmountGroupConverter(300);
      break;
```

```
    }
}
```

该代码处理了PropertyGroupCreated事件，并指定自定义转换器至数据源的不同列。在本示例中，DateTimeGroupConverter以及AmountGroupConverter类是用来对DateTime以及双精度类型的值转换为不同范围的简单转换器。

下图显示了自定义分组的效果：



请注意项目可能出现在多个不同的分组中。例如，DateTimeGroupConverter将日期分组为“本周”，“本年度”，“去年”，以及“去年以前”。“本周”分组的项目也包括在“本年度”分组中。

这是ICollectionView接口本身提供的功能，并不是C1FlexGrid或者C1FlexGridGroupPanel提供的特定功能。

6.3.4 C1FlexGridGroupPanel实现注意事项

我们决定将分组UI的功能提供在一个独立的程序集而不是将其直接添加在C1FlexGrid控件上，原因如下：

- 尽可能地保持C1FlexGrid轻巧，快速，可移植。Silverlight版本的C1FlexGrid仍旧小于250K，这使得其始终保持目前市售版本的data grid类产品中最小的尺寸。它也是最广泛使用的data grid，包括WinForms，Silverlight，WPF，Windows Phone，WinRT，ActiveX，以及Compact Framework版本。
- 允许轻松定制的分组界面。C1FlexGridGroupPanel是一个非常简单的控件，我们将源代码提供给用户，使得用户可以创建其自定义的版本。
- 为演示C1FlexGrid控件的可扩展性和灵活性。C1FlexGridGroupPanel使用由C1FlexGrid暴露的简单强大的对象模型进行开发，不使用任何的自定义扩展，修改，或者使用任何内部的后门方法。（与此相同的还有C1FlexGridFilter组件，该组件提供了自组织的过滤，并且同样地包含在另一个独立的程序集中）。

C1FlexGridGroupPanel实际上是一个Grid元素，包含一个用来显示水印消息的TextBlock，以及一个用来显示源ICollectionView中所有可用分组的水平放置的StackPanel。

分组由GroupMarker元素表示，该元素可以被单击对分组进行排序或者收起整个分组，再或者可以进行拖拽以重新排布分组的顺序。
C1FlexGridGroupPanel处理四种类型的拖拽行为：

- 在分组区域内拖拽GroupMarkers以便重新排布这些分组，
- 拖拽GroupMarkers至Grid区域，以便移除该分组并在特定位置恢复该列，
- 从Grid中拖拽ColumnHeader元素至分组区域以创建新的分组，以及
- 在Grid中间拖拽ColumnHeader元素以便重新排布Grid的列。

- 所有的拖拽行为由一个叫做DragDropManager的辅助类处理。

前两种拖拽行为由GroupMarker类发起，将检测鼠标拖拽动作并调用DragDropManager上的DoDragDrop，同时传入Marker做为一个参数。后两种拖拽行为将在响应C1FlexGrid的DraggingColumn事件的事件处理函数中发起。

当DoDragDrop方法被调用时，DragDropManager将在整个页面显示一个透明的元素，捕获鼠标消息，并触发Dragging事件，使得调用方可以更新拖放的目标位置。当用户释放鼠标时，DragDropManager将触发Dropped事件，调用方可以结束整个拖放动作。

该实现非常简短并且具有可移植性。程序集仅有25K左右，并且代码可以在Silverlight或者WPF下进行编译，完全不需要任何的条件编译代码块。

6.3.5 聚合数据

一旦您对数据进行了分组，则通常下一步有用的操作时计算这些分组的聚合值。例如，如果您将销售数据按国家或产品类别分组，您可以显示每个国家和产品类别的总销售额。为了使用C1FlexGrid实现这一点，可以设置列上的GroupAggregate属性为聚合，则Grid将自动地计算并显示聚合结果。当数据发生变化时，聚合结果将自动地重新计算。

请注意，聚合将出现在该分组的Header行中。设置Grid的AreGroupHeadersFrozen属性的值为False以使得其可见。

例如，考虑以下的Grid定义：

XAML

```
<fg:C1FlexGrid x:Name="_flex" AutoGenerateColumns="False">
  <fg:C1FlexGrid.Columns>
    <fg:Column Header="Line" Binding="{Binding Line}" />
    <fg:Column Header="Color" Binding="{Binding Color}" />
    <fg:Column Header="Name" Binding="{Binding Name}" />
    <fg:Column Header="Price" Binding="{Binding Price}"
      Format="n2" HorizontalAlignment="Right" Width="*"/>
    <fg:Column Header="Cost" Binding="{Binding Cost}"
      Format="n2" HorizontalAlignment="Right" Width="*"/>
    <fg:Column Header="Weight" Binding="{Binding Weight}"
      Format="n2" HorizontalAlignment="Right" Width="*"/>
    <fg:Column Header="Volume" Binding="{Binding Volume}"
      Format="n2" HorizontalAlignment="Right" Width="*"/>
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

如果您设置了Grid的ItemsSource为一个ICollectionView对象，该对象设置为按照“Line”，“Color”以及“Name”进行分组，则您将获得一个类似于下图的Grid：

Line	Color	Name	Price	Cost	Weight	Volume
Total: (200 items)						
Line: Washers (55 items)						
Color: Green (16 items)						
Price: Medium (1 items)						
Washers	Green	P 0	68.00	233.00	3.00	1,171.00
Price: Very High (8 items)						
Washers	Green	P 2	678.00	201.00	12.00	2,748.00
Washers	Green	P 23	840.00	283.00	60.00	2,077.00
Washers	Green	P 42	687.00	107.00	59.00	1,856.00
Washers	Green	P 88	747.00	408.00	88.00	899.00
Washers	Green	P 91	630.00	249.00	82.00	3,505.00
Washers	Green	P 132	658.00	15.00	13.00	3,857.00
Washers	Green	P 187	889.00	349.00	68.00	2,952.00
Washers	Green	P 194	865.00	185.00	20.00	582.00
Price: High (6 items)						
Washers	Green	P 67	487.00	521.00	65.00	2,196.00
Washers	Green	P 75	257.00	350.00	19.00	526.00

该Grid以一个可折叠的大纲试图格式显示这些分组，并自动地显示在每一个分组中元素的个数。这一点和Microsoft DataGrid控件显示的视图非常相似。

C1FlexGrid让您更进一步，以显示列的聚合值。例如，为了显示“Price”，“Cost”，“Weight”以及“Volume”列的合计，请按照如下方式修改XAML：

XAML

```
<fg:C1FlexGrid x:Name="_flex" AutoGenerateColumns="False"
  AreRowGroupHeadersFrozen="False">
  <fg:C1FlexGrid.Columns>
    <fg:Column Header="Line" Binding="{Binding Line}" />
    <fg:Column Header="Color" Binding="{Binding Color}" />
    <fg:Column Header="Name" Binding="{Binding Name}" />
    <fg:Column Header="Price" Binding="{Binding Price}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Cost" Binding="{Binding Cost}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Weight" Binding="{Binding Weight}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
    <fg:Column Header="Volume" Binding="{Binding Volume}"
      Format="n2" HorizontalAlignment="Right" Width="*"
      GroupAggregate="Sum"/>
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

该段XAML代码包含两点变化：

- 设置Grid的AreGroupHeadersFrozen属性的值为False。这个设置是必要的，因为分组的聚合信息将出现在分组的Header行上，如果Header被冻结，则聚合信息将不可见。
- 同时还设置了若干列的GroupAggregate属性的值为“Sum”。这将使得Grid计算并在分组的Header行上显示聚合信息。有这么几个聚合选项可用，包括“Sum”，“Average”，“Count”，“Minimum”，“Maximum”等等。

做了这个改变后，Grid将看起来像这样：

Line	Color	Name	Price	Cost	Weight	Volume
Total: (200 items)			103,610.00	61,744.00	10,089.00	573,086.00
Line: Washers (55 items)			27,656.00	16,865.00	2,697.00	144,544.00
Color: Green (16 items)			8,150.00	4,658.00	669.00	33,853.00
Price: Medium (1 items)			68.00	233.00	3.00	1,171.00
Washers	Green	P 0	68.00	233.00	3.00	1,171.00
Price: Very High (8 items)			5,994.00	1,797.00	402.00	18,476.00
Washers	Green	P 2	678.00	201.00	12.00	2,748.00
Washers	Green	P 23	840.00	283.00	60.00	2,077.00
Washers	Green	P 42	687.00	107.00	59.00	1,856.00
Washers	Green	P 88	747.00	408.00	88.00	899.00
Washers	Green	P 91	630.00	249.00	82.00	3,505.00
Washers	Green	P 132	658.00	15.00	13.00	3,857.00
Washers	Green	P 187	889.00	349.00	68.00	2,952.00
Washers	Green	P 194	865.00	185.00	20.00	582.00
Price: High (6 items)			2,084.00	2,476.00	193.00	12,436.00
Washers	Green	P 67	487.00	521.00	65.00	2,196.00
Washers	Green	P 75	257.00	350.00	19.00	526.00

请注意，分组Header是如何显示分组的聚合值的。当数据发生变化时，聚合值将自动地重新计算。

6.4 合并

6.4.1 合并单元格

Grid上的AllowMerging属性将在Grid级别启用单元格合并。一旦您在Grid级别启用了合并，则可以使用Row.AllowMerging以及Column.AllowMerging属性以选择特定的行和列进行合并。

例如，下面的代码合并包含相同国家的单元格：

C#

```
// 在可滚动区域启用合并
fg.AllowMerging = AllowMerging.Cells;

// 在列"Country" 以及"FirstName"上启用合并
fg.Columns["Country"].AllowMerging = true;
fg.Columns["FirstName"].AllowMerging = true;
```

此代码将创建如下图所示grid：

6.5 过滤

6.5.1

C1FlexGrid和“扩展程序集”一起打包发布，这些扩展程序集包括C1.Silverlight.FlexGridFilter 和 C1.WPF.FlexGridFilter（在线文档'C1.WPF.FlexGridFilter.4 程序集'），这些程序集提供了Excel样式的过滤功能。为使用这些组件，需要将它们添加到您的工程引用，并创建一个C1FlexGridFilter 对象关联到一个现有的Grid。例如：

C#

```
// 创建C1FlexGrid
var flex = new C1FlexGrid();

// 在Grid上启用过滤
var gridFilter = new C1FlexGridFilter(flex);
```

此外，C1.Silverlight.FlexGrid.GroupPanel以及C1.WPF.FlexGrid.GroupPanel （在线文档'C1.WPF.FlexGrid.GroupPanel.4 程序集'）程序集同样也做为扩展程序集提供对C1FlexGrid控件的分组管理功能。我们决定使用扩展组件而不是将功能直接添加到控件上的原因有两个：

- 能够使我们保持Grid所在的程序集为一个较小的尺寸，并允许您选择在每一个工程中使用哪些扩展。
- 允许通过自定义代码扩展C1FlexGrid的功能。该代码示例演示了由WPF及Silverlight版提供的ExcelBook以及ExcelGrid的功能。

您同样也可以在Grid声明的XAML文件中启用过滤。这里是实现该功能的语法：

XAML

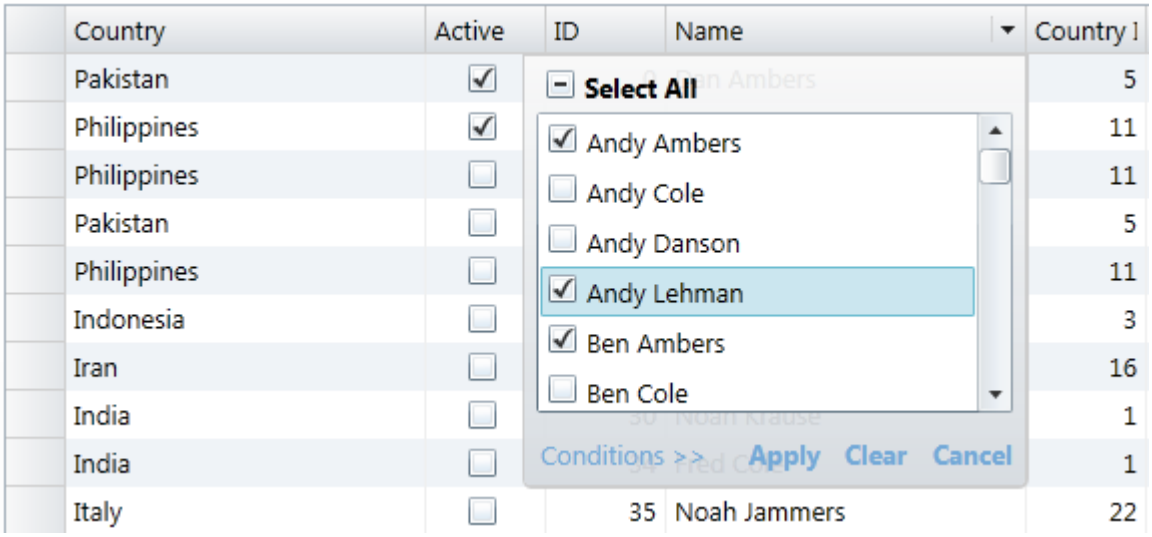
```
<c1:C1FlexGrid Name="_flex" >
  <!-- 向控件添加过滤支持： -->
  <c1:C1FlexGridFilterService.FlexGridFilter>
    <c1:C1FlexGridFilter />
  </c1:C1FlexGridFilterService.FlexGridFilter>
</c1:C1FlexGrid>
```

一旦启用了过滤功能，当鼠标悬停在列头上时，Grid会显示一个下拉图标。下拉菜单中显示一个编辑器，允许用户指定如何筛选该列中的数据。用户可选择两种类型的过滤器：

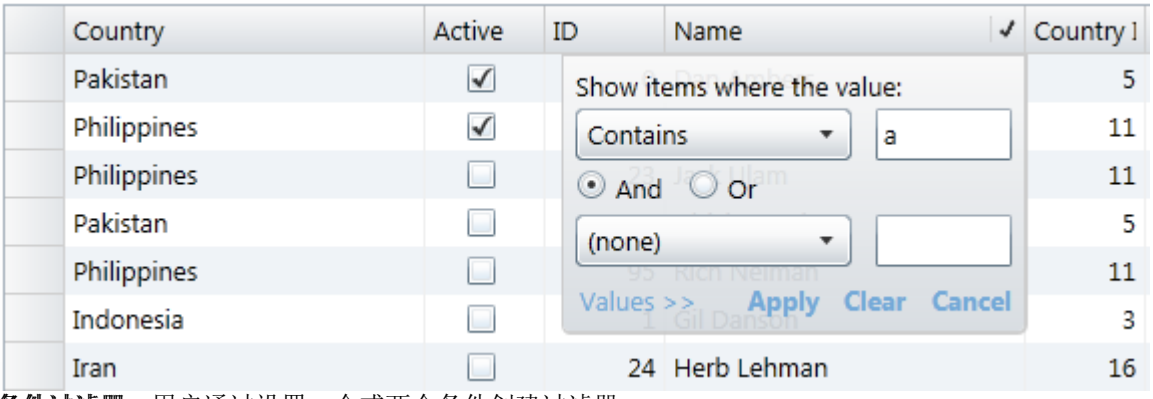
- **值过滤器**：该过滤器允许用户从列表中显示的值中进行选择。
- **条件过滤器**：此过滤器允许用户指定由一个操作符（大于，小于等等）和一个参数。条件本身可以通过一个

AND或者一个OR运算符进行组合。

下面的图像显示，当正在进行编辑时，过滤器的外观：



值过滤器：用户通过从列表中选择值创建筛选器。



条件过滤器：用户通过设置一个或两个条件创建过滤器。

对于大多数应用程序，默认的过滤器设置是足够的，但您可以通过几种不同的方法自定义筛选器。

选择筛选模式

过滤器工作在两种不同的模式下，取决于UseCollectionView属性的设置。

如果您设置UseCollectionView属性的值为false，则不满足过滤条件的行将隐藏（过滤器将设置它们的Visible属性为false）。在这种模式下，过滤器对行计数没有影响。您可以在绑定以及非绑定模式的Grid中使用该模式。

如果您设置了过滤器的UseCollectionView属性值为true，则过滤将应用到数据源（通过ICollectionView.Filter属性）。在这种模式下，对过滤器所做的改变将直接影响数据源暴露给Grid的数据项的个数，与此同时，任何绑定到相同数据源的其他控件将受到影响。您只能在绑定模式使用该模式的过滤器。

例如：

C#

```
// 创建C1FlexGrid
var flex = new C1FlexGrid();

// 在Grid上启用过滤
var gridFilter = new C1FlexGridFilter(flex);

// 在数据源级别进行过滤
gridFilter.UseCollectionView = true;
```

或者，通过 XAML:

XAML

```
<c1:C1FlexGrid Name="_flex" >
  <!-- 向控件添加过滤支持： -->
  <c1:C1FlexGridFilterService.FlexGridFilter>
    <c1:C1FlexGridFilter UseCollectionView="True"/>
  </c1:C1FlexGridFilterService.FlexGridFilter>
</c1:C1FlexGrid>
```

为每一列自定义过滤器类型

默认情况下，每个列将启用过滤器。包含布尔型或者枚举值的数据列将使用一个值过滤器，而包含其他数据值类型的列将可以使用值过滤器或者条件过滤器。

您可以通过FilterType属性改变这一行为并指定在每一列上启用的过滤器类型。

在当列举有大量的相同值或者当列包含无法进行过滤的绑定关系时，指定过滤器类型将非常重要。

例如，包含图像的列不能用值或条件筛选器来过滤。在这种情况下，你可以通过设置FilterType 属性为None禁用过滤器。
一个包含数千个项目的Grid可能具有一个唯一标识符的列，该列添加了过多的项目至值过滤器，这将使得其性能低下，实际上这种过滤通常没有什么实际用途。在这种情况下，通过设置FilterType属性为Condition以禁用值过滤器。

下面的代码演示了如何实现这一点：

C#

```
// 创建C1FlexGrid
var flex = new C1FlexGrid();

// 在Grid上启用过滤
var gridFilter = new C1FlexGridFilter(flex);

// 在图像类型的列上禁用过滤
var columnFilter = gridFilter.GetColumnFilter(flex.Columns["Image"]);
columnFilter.FilterType = FilterType.None;

// 在标识符列上禁用值过滤器
columnFilter = gridFilter.GetColumnFilter(flex.Columns["ID"]);
columnFilter.FilterType = FilterType.Condition;
```

在代码中指定过滤器

在大多数情况下，用户仅需设置过滤器。但是ColumnFilter类同样也提供了完整的对象模型，允许开发人员通过代码检查以及修改过滤条件。
例如，下面的代码将过滤器应用到第二列。该过滤器将使的Grid显示第二列中的值包含字母Z的项目：

C#

```
// 创建C1FlexGrid
var flex = new C1FlexGrid();

// 在Grid上启用过滤
var gridFilter = new C1FlexGridFilter(flex);

// 获取第一列的过滤器
var columnFilter = gridFilter.GetColumnFilter(flex.Columns[0]);

// 创建过滤条件（包含字母'Z'）
var condition = columnFilter.ConditionFilter.Condition1;
condition.Operator = ConditionOperator.Contains;
```

```
condition.Parameter = "Z";

// 应用该过滤器
gridFilter.Apply();
```

保存过滤器

C1FlexGridFilter类包含一个FilterDefinition属性，该属性可以以XML字符串形式获取或设置当前的过滤器状态。当用户退出应用程序时，可以使用该字符串来保持该过滤器状态，这样您可以稍后还原。

您也可以保存几个过滤器的定义，允许用户选择并自定义这些预设过滤器。您还可以通过SaveFilterDefinition 以及LoadFilterDefinition方法保存和恢复过滤器的定义。

6.5.2 过滤数据（使用ICollectionView）

ICollectionView接口通过其Filter属性支持数据过滤。Filter属性指定一个方法，该方法将为集合中的每一个数据项进行调用。如果方法返回true，则该项目包含在视图中。如果方法返回false，则该项目将被过滤掉。（这种方法被称为断言）。

该文档包含的MainTestApplication示例包括一个SearchBox控件，该控件包含一个文本框，在此用户可以键入一个待搜索的值，除此之外，还包含了一个Timer。定时器提供了一个小的延迟，允许用户在输入待搜索值时，不会再每一个字符键入时立刻重复应用搜索。

当用户停止输入，定时器经过一段时间后将通过以下代码应用过滤器：

C#

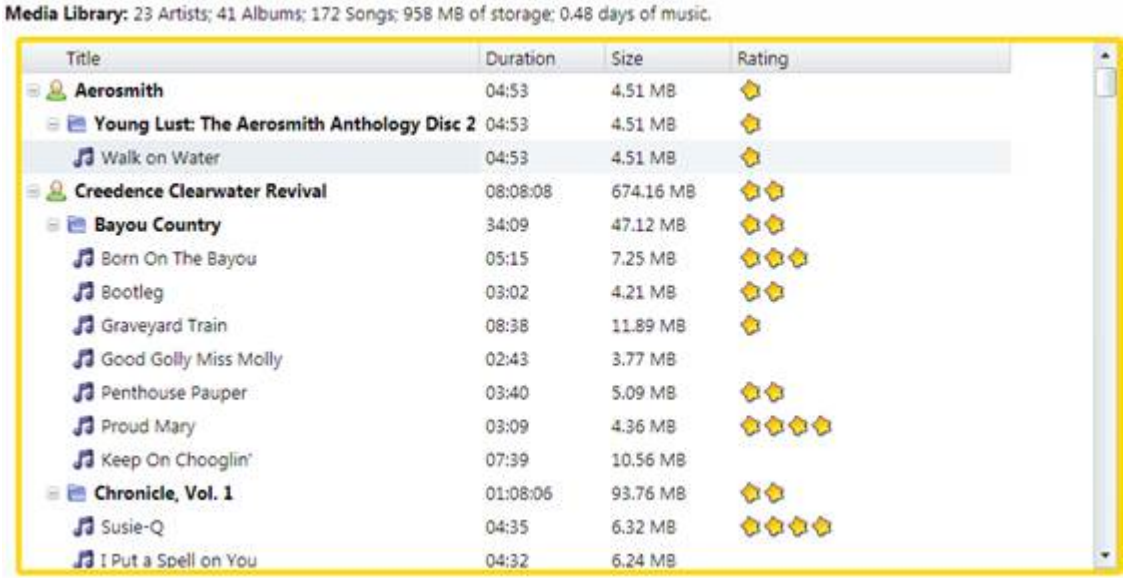
```
_view.Filter = null;
_view.Filter = (object item) =>
{
  var srch = _txtSearch.Text;
  if (string.IsNullOrEmpty(srch))
  {
    return true;
  }
  foreach (PropertyInfo pi in _propertyInfo)
  {
    var value = pi.GetValue(item, null) as string;
    if (value != null &&
        value.IndexOf(srch, StringComparison.OrdinalIgnoreCase) > -1)
    {
      return true;
    }
  }
  return false;
};
```

注意这里代码是如何使用一个lambda函数设置Filter属性的。我们可以提供一个单独的方法，但这种符号通常更方便，因为它是简洁的，并且允许我们使用本地变量，如果我们需要它们。

lambda函数接受一个项目作为一个参数，获取该对象的指定属性的值，如果任何对象的属性包含待搜索的字符串则返回true。

例如，如果对象是“歌曲”类型，并且指定的属性是“标题”，“专辑”，“艺术家”，如果在歌曲的标题，专辑或艺术家中包含待搜索的字符串则返回true。这是一个功能强大且易于使用的搜索机制，类似于一个用于在苹果的iTunes应用程序。

一旦应用了过滤器，grid（以及其他绑定到相同的ICollectionView对象的控件）都将仅显示由该过滤器筛选出来的项目，以反映该变化。
注意过滤和分组可以一起工作。下图（来自于MainTestApplication 示例）显示一个非常大的歌曲列表，并应用了一个过滤器：



The image shows the filter set to the word “Water.” The filter looks for matches in all fields (song, album, artist), so all “Creedence Clearwater Revival” songs are automatically included.

Notice the status label above the grid. It automatically updates whenever the list changes, so when the filter is applied the status updates to reflect the new filter. The routine that updates the status uses LINQ to calculate the number of artists, albums, and songs selected, as well as the total storage and play time. The song status update routine is implemented as follows:

C#

```
// update song status
 void UpdateSongStatus()
 {
   var view = _flexiTunes.ItemsSource as ICollectionView;
   var songs = view.OfType<Song>();
   _txtSongs.Text = string.Format(
      "{0:n0} Artists; {1:n0} Albums; {2:n0} Songs; " +
      "{3:n0} MB of storage; {4:n2} days of music.",
     (from s in songs select s.Artist).Distinct().Count(),
     (from s in songs select s.Album).Distinct().Count(),
     (from s in songs select s.Name).Count(),
     (double)(from s in songs select s.Size/1024.0/1024.0).Sum(),
     (double)(from s in songs select s.Duration/3600000.0/24.0).Sum());
 }
```

This routine is not directly related to the grid, but is listed here because it shows how you can leverage the power of LINQ to summarize status information that is often necessary when showing grids bound to large data sources.

The LINQ statement above uses the **Distinct** and **Count** commands to calculate the number of artists, albums, and songs currently exposed by the data source. It also uses the **Sum** command to calculate the total storage and play time for the current selection.

7 自定义单元格

如果您之前使用过任意一种Microsoft data grid控件（WinForms，Silverlight，或者WPF平台），您可能知道您需要创建自定义的Column对象，覆盖若干方法并且通过代码添加自定义列至grid以完成任何的重大自定义过程。这是一个不坏的方法（Silverlight以及WPF版本的DataGrid控件均遵循这个模型，主要目的是和Microsoft grid保持兼容性）。但C1FlexGrid控件使用了一种非常不同的方法。

7.1 通过代码自定义单元格：CellFactory 类

grid具有一个CellFactory类，用来创建显示在grid上的每一个单元格。为创建自定义单元格，则可以创建一个类继承自ICellFactory接口，并指定该类型至Grid的CellFactory属性。

和自定义列相似，自定义ICellFactory类可以是高度专业化以及面向应用程序的，也可以是通用的，可重复使用，可自定义的类。通常，自定义ICellFactory类比自定义列简单的多，因为它将直接处理单元格（而自定义列，与之相反，需要处理列本身以及单元格和其他一些包含在列中的对象）。

ICellFactory 接口看起来像这样：

C#

```
public interface ICellFactory
{
  FrameworkElement CreateCell(
    C1FlexGrid grid,
    CellType cellType,
    CellRange range);

  FrameworkElement CreateCellEditor(
    C1FlexGrid grid,
    CellType cellType,
    CellRange range)

  void DisposeCell(
    C1FlexGrid grid,
    CellType cellType,
    FrameworkElement cell);
}
```

第一个方法，CreateCell，将创建一个表示单元格的FrameworkElement对象。这些参数包括：拥有该单元格的grid，待创建的单元格类型，以及表示的CellRange。CellType参数指定是否即将创建的单元格是一个普通的数据单元格还是一个行头或者列头，再或者是位于左上角或者右下角的固定单元格。

第二个方法，CreateCellEditor，和第一个方法的功能类似，不过它将用来创建位于编辑态的单元格。

最后一个方法，DisposeCell，将在单元格从grid中移除时调用。用作为调用者提供一个时机释放关联到该单元格对象的资源。

当使用自定义单元格时，最重要的一点是要理解grid的单元格是临时性存在的。当用户滚动，排序或者在单元格上选择一个范围时，单元格将持续不断地创建 及销毁。这个过程被称为虚拟化，在Silverlight和WPF应用中相当普遍。如果没有虚拟化，grid通常需要同时创建几千个可视元素，这将破坏性能。

实现自定义的ICellFactory类相当容易，因为您可以从C1FlexGrid包含的默认的CellFactory类派生。默认的CellFacgtory类被设计为可扩展的，因此您可以让其处理单元格创建的具体步骤，仅需要自定义您需要的逻辑部分。

iTunes的示例（在线文档）和金融应用程序示例是Grid使用自定义ICellFactory类实现功能的示例。这些示例是该文档中包含的MainTestApplication的一部分，每一个示例都包含了Silverlight以及WPF版本。本文档中的描述将集中在关键的实现点。关于详情，请参阅示例应用程序源代码。

7.2 在XAML中自定义单元格：CellTemplate 以及CellEditingTemplate

如果您希望通过XAML，而不是代码，创建自定义单元格，您也可以实现。C1FlexGrid Column对象具有CellTemplate以及CellEditingTemplate属性，您可以用来指定列上的显示模式/编辑模式的单元格使用的不同的可视

化元素。

例如，下面的XAML代码定义自定义的视觉元素，用于显示和编辑列中的值。在该列中的单元格显示为一个绿色，粗体，文本居中对齐，并通过一个具有一个编辑图标的文本框进行编辑：

XAML

```
<c1:C1FlexGrid x:Name="_fgTemplated">
  <c1:C1FlexGrid.Columns>
    <!-- 添加模版列-->
    <c1:Column ColumnName="_colTemplated" Header="Template" Width="200">
      <!-- 显示模式下单元格的模版 -->
      <c1:Column.CellTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Name}"
           Foreground="Green" FontWeight="Bold"
           VerticalAlignment="Center"/>
        </DataTemplate>
      </c1:Column.CellTemplate>
      <!-- 编辑模式下的单元格模版-->
      <c1:Column.CellEditingTemplate>
        <DataTemplate>
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="Auto" />
              <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Image Source="edit_icon.png" Grid.Column="0" />
            <TextBox Text="{Binding Name, Mode=TwoWay}" Grid.Column="1" />
          </Grid>
        </DataTemplate>
      </c1:Column.CellEditingTemplate>
    </c1:Column>
  </c1:C1FlexGrid.Columns>
</c1:C1FlexGrid>
```

8 打印支持

8.1 基本打印

Pint方法是打印一个C1FlexGrid的简单方法。

Pint方法允许您指定文档名称，页边距，缩放和最大打印页数。输出是一个对grid的原样呈现，包括全部的样式元素，字体，渐变，图片，等等。行头和列头包含在每一页中。

注意：C1.Silverlight.FlexGrid.5 库提供了对Pint方法的额外重载，以支持打印机Fallback，并使用Silverlight5 提供的DefaultPrinter能力。这些重载只能在Silverlight 5版本的控件中可用。

8.2 高级打印

如果您希望在打印过程中获得更多的控制能力，请使用GetPageImages方法以自动将grid拆分为多张图像，每一张图像将呈现为一个独立的Page。每一张图像都是对grid的某个部分的100%精确呈现，包括央视，自定义元素，在每一页上重复的行头和列头，等等。

GetPageImages方法同样允许调用方缩放该图片，因此整个Grid可以以原始尺寸呈现，缩放以适应单个页面，或者缩放以适应单个页面的宽度。

一旦你得到的页面图像，你可以使用WPF或Silverlight的打印支持将其呈现至文档，您将具有完全的控制灵活性。例如，您可以创建包含多个grid、图表和其他类型内容的文档。您还可以自定义页眉和页脚，添加信笺抬头，等等。

在WPF和Silverlight的打印框架是不同的。一下章节演示应用程序如何在各个平台（WPF和Silverlight）使用GetPageImages方法将C1FlexGrid打印至文档。

在WPF中打印C1FlexGrid

和Silverlight相比，在WPF中打印文档需要一个稍微不同的步骤：

1. 创建一个PrintDialog 对象。
2. 如果对话框的ShowDialog方法返回true，则：
3. 创建一个Paginator 对象，该对象将提供文档的内容。
4. 调用对话框的Print方法。

下面的代码显示了该机制的示例实现。

C#

```
// 打印 grid
void _btnPrint_Click(object sender, RoutedEventArgs e)
{
  var pd = new PrintDialog();
  if (pd.ShowDialog().Value)
  {
    // 获取页边距，缩放模式
    var margin = 96.0;
    var scaleMode =;

    // 获取页面尺寸
    var pageSize = new Size(pd.PrintableAreaWidth,
                            pd.PrintableAreaHeight);

    //  创建 paginator
    var paginator = new FlexPaginator(
      _flex, ScaleMode.PageWidth,
      pageSize,
      new Thickness(margin), 100);
```

```
    // 打印文档
    pd.PrintDocument(paginator, "C1FlexGrid printing example");
  }
}
```

FlexPaginator类提供了页面图像，从概念上和Silverlight中使用的PrintPage事件处理相似。实现如下：

C#

```
/// <summary>
/// 用来呈现C1FlexGrid控件的DocumentPaginator类。
/// </summary>
public class FlexPaginator : DocumentPaginator
{
  Thickness _margin;
  Size _pageSize;
  ScaleMode _scaleMode;
  List<FrameworkElement> _pages;

  public FlexPaginator(C1FlexGrid flex,
    ScaleMode scaleMode,
    Size pageSize,
    Thickness margin, int maxPages)
  {
    // 保存参数
    _margin = margin;
    _scaleMode = scaleMode;
    _pageSize = pageSize;

    // 在构建grid图像之前为页边距调整页面尺寸
    pageSize.Width -= (margin.Left + margin.Right);
    pageSize.Height -= (margin.Top + margin.Bottom);

    // 获取grid每一页的图像
    _pages = flex.GetPageImages(scaleMode, pageSize, maxPages);
  }
```

该构造函数创建页面图像。它们将在稍后再打印framework调用pageinator的GetPage方法时呈现到页面上：

C#

```
  public override DocumentPage GetPage(int pageNumber)
  {
    // 创建页面元素
    var pageTemplate = new PageTemplate();

    // 设置页边距
    pageTemplate.SetPageMargin(_margin);

    // 设置内容
    pageTemplate.PageContent.Child = _pages[pageNumber];
    pageTemplate.PageContent.Stretch =
      _scaleMode == ScaleMode.ActualSize
        ? System.Windows.Media.Stretch.None
        : System.Windows.Media.Stretch.Uniform;

    // 设置页脚文本
    pageTemplate.FooterRight.Text = string.Format("Page {0} of {1}",
        pageNumber + 1, _pages.Count);
```

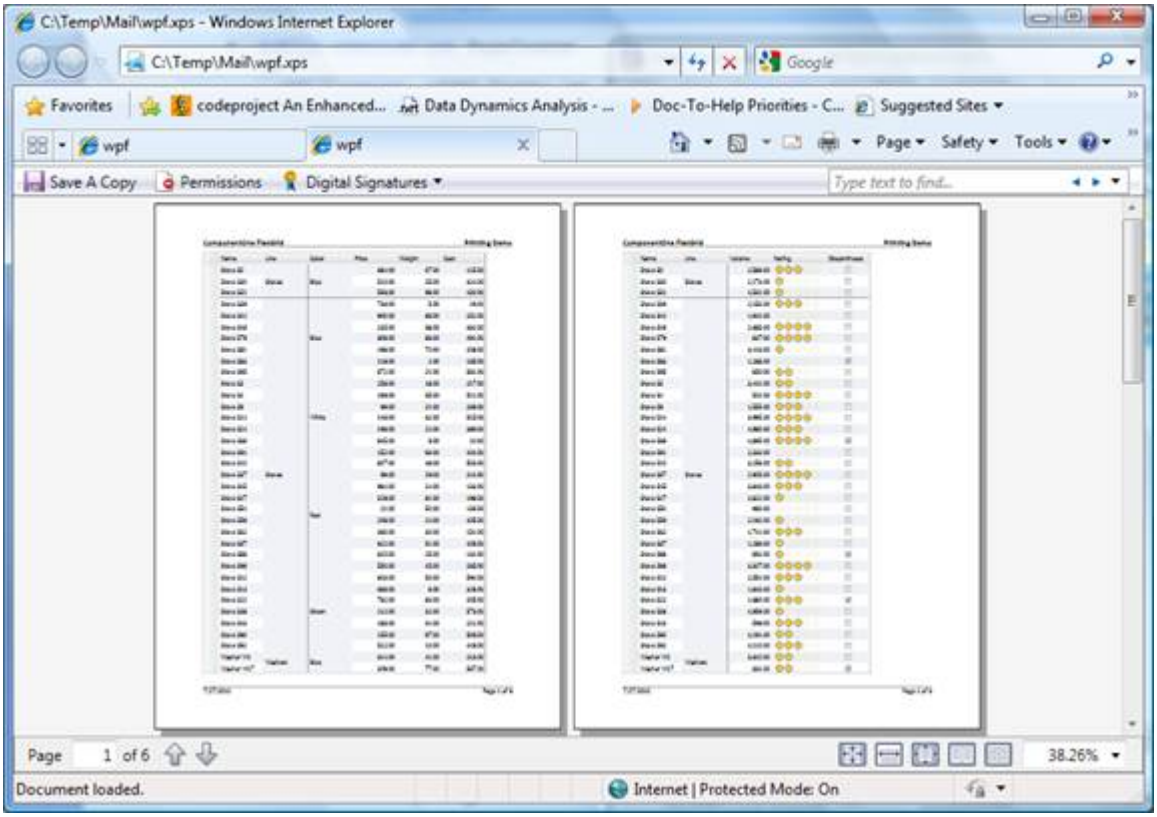
```
    // 排布页面上的元素
    pageTemplate.Arrange(
        new Rect(0, 0, _pageSize.Width, _pageSize.Height));

    // 返回新的文档页面
    return new DocumentPage(pageTemplate);
  }
```

和Silverlight示例中所展示的一样，一个辅助的PageTemplate类被用作保存grid的图像并提供页边距，页眉和页脚。paginator上的其它方法提供了简单的实现：

C#

```
  public override int PageCount
  {
    get { return _pages.Count; }
  }
  public override IDocumentPaginatorSource Source
  {
    get { return null; }
  }
  public override Size PageSize
  {
    get { return _pageSize; }
    set { throw new NotImplementedException(); }
  }
  public override bool IsPageCountValid
  {
    get { return true; }
  }
}
```

下图展示了当grid呈现到一个XPS文件时，所创建的文档。该图像是非常准确的，包括在样本中使用的自定义rating单元格。行头和列头会自动包含在每一个页面中，同时包含一个简单的页面标题和标准的“Page n of m”页脚。



在Silverlight 4中打印C1FlexGrid

在Silverlight中打印文档，须遵循以下步骤：

1. 创建PrintDocument对象。
2. 添加处理函数至BeginPrint，PrintPage，以及EndPrint事件。
3. 调用文档的Print方法。

Print方法显示打印对话框。如果用户单击OK，则文档将触发一次BeginPrint事件，接下来为打印每一页触发一次PrintPage事件，最终将在最后一页呈现完毕时触发一个EndPrint事件。下面的代码显示了该机制的示例实现。

我们使用两个变量来保存页面图像并保持跟踪当前打印的页面：

C#

```
List<FrameworkElement> _pages;
int _currentPage;
```

这里是调用打印文档的处理程序：

C#

```
// 打印 grid
void _btnPrint_Click(object sender, RoutedEventArgs e)
{
  // 创建PrintDocument变量
  var pd = new System.Windows.Printing.PrintDocument();

  // prepare to print
  _pages = null;
  pd.PrintPage += pd_PrintPage;

  // 打印文档
  pd.Print("C1FlexGrid");
}
```

PrintPage方法做所有的工作。当第一次被调用时，它将生成全部的页面图像，接下来呈现这些图形至页面。

C#

```
void pd_PrintPage(object sender, PrintPageEventArgs e)
{
  if (_pages == null)
  {
    // 计算页面尺寸，扣除页边距
    var sz = e.PrintableArea;
    sz.Width -= 2 * 96; // one inch left/right margins
    sz.Height -= 2 * 96; // one inch top/bottom margins

    // 上下页边距为1英寸
    _currentPage = 0;
    _pages = _flex.GetPageImages(ScaleMode.ActualWidth, sz, 100);
  }

  // 创建表示该页面的视觉元素
  var pageTemplate = new PageTemplate ();

  // 应用页边距至页面模版
  pageTemplate.SetPageMargin(new Thickness(_margin));

  // 添加内容至页面模板
  pageTemplate.PageContent.Child = _pages[_currentPage];

  // 应用页脚文本
  pageTemplate.FooterRight.Text = string.Format("Page {0} of {1}",
    _currentPage + 1, _pages.Count);

  // 呈现页面
  e.PageVisual = pageTemplate;

  // 移动到下一页
  _currentPage++;
  e.HasMorePages = _currentPage < _pages.Count;
}
```

该示例使用一个自定义的辅助类，叫做PageTemplate，而不是将grid图像直接呈现在页面上。

该类提供了页边距，页眉，页脚以及一个host实际grid图像的ViewBox控件。将grid图像直接呈现到页面也能达到同样效果，但是模版增加了很多灵活性。（注意PageTemplate类在示例中实现，其并非属于C1FlexGrid程序集的一部分）。
这里是定义PageTemplate类的XAML代码：

XAML

```
<Grid x:Name="LayoutRoot" Background="White">  <Grid.RowDefinitions>
<RowDefinition Height="96" />    <RowDefinition Height="*"/>    <RowDefinition
Height="96" />  </Grid.RowDefinitions>  <Grid.ColumnDefinitions>
<ColumnDefinition Width="96"/>    <ColumnDefinition Width="*" />
<ColumnDefinition Width="96"/>  </Grid.ColumnDefinitions>

  <!-- header -->  <Border Grid.Column="1" HorizontalAlignment="Stretch"
VerticalAlignment="Bottom" Margin="0 12"        BorderBrush="Black"
BorderThickness="0 0 0 1" >    <Grid>      <TextBlock Text="ComponentOne FlexGrid"
FontWeight="Bold" FontSize="14"        VerticalAlignment="Bottom"
HorizontalAlignment="Left" />      <TextBlock Text="Printing Demo"
FontWeight="Bold" FontSize="14"        VerticalAlignment="Bottom"
HorizontalAlignment="Right" />    </Grid>  </Border>

  <!-- footer -->  <Border Grid.Column="1" Grid.Row="2"
```

```
HorizontalAlignment="Stretch"      VerticalAlignment="Top"  Margin="0 12"
BorderBrush="Black" BorderThickness="0 1 0 0" >    <Grid>      <TextBlock
x:Name="FooterLeft" Text="Today"         VerticalAlignment="Bottom"
HorizontalAlignment="Left" />      <TextBlock x:Name="FooterRight" Text="Page {0} of
{1}"         VerticalAlignment="Bottom" HorizontalAlignment="Right" />    </Grid>
</Border>

  <! -- page content -->  <Viewbox x:Name="PageContent" Grid.Row="1" Grid.Column="1"
VerticalAlignment="Top" HorizontalAlignment="Left" /></Grid>
```

9 布局与外观

9.1 ComponentOne ClearStyle 技术

ClearStyle 技术是一种新的，提供Silverlight以及WPF控件样式的方法。ClearStyle 允许您为控件创建自定义样式，而不用处理麻烦的XAML模板和样式资源。

目前，为全部的WPF控件添加一个主题支持，您必须创建一个样式资源模版。在Microsoft Visual Studio中，处理这个过程很难；这也是为什么微软引入了Expression Blend使得该项任务变得稍微简单一些。对于那些不熟悉Blend或者没有足够的时间学习的开发人员来说，必须在两个开发环境之间切换是一个挑战。您也可以考虑聘请一个设计师，但是当您的设计师和开发人员共享同一个XAML文件时，事情会变得有些复杂。

这就是为什么引入ClearStyle的原因。通过ClearStyle的修改样式的能力，您可以在Visual Studio中尽可能的直观的修改控件样式。在大多数情况下，您可能只想对您的应用程序中的控件做一些简单的样式改变，因此这一过程应当非常简单容易。例如，如果您只想改变您的data grid中间行的颜色，这个操作应当简单的就像设置一个属性一样。您不应该创建一个完整的和复杂的模板，只是为了简单地改变一些颜色。

9.1.1 ClearStyle的结构

控件的每一个关键部分都是以简单的颜色属性出现的。这导致了为每个控件准备的一组独特的样式属性。例如，一个Gauge具有PointFill以及PointStroke属性，而一个DataGrid的行具有SelectedBrush以及MouseOverBrush属性。

假定在您的Form上具有一个不支持ClearStyle的控件。您可以获取由ClearStyle创建的XAML，并用其做为匹配界面上其他元素外观的模具（比方说抓取精确的颜色）。或者假定您希望通过ClearStyle重写部分的生成样式（比如说您的自定义滚动条）。这同样也是可行的，因为ClearStyle可以被扩展，您可以覆盖期望的样式。

ClearStyle旨在提供一个快速容易地修改样式的方案，但是您仍旧可以自由地按照老式的方式操作ComponentOne控件，获取需要的精确样式。ClearStyle的接口将不支持那些需要全面自定制的较少见的场景。

9.1.2 ClearStyle 属性

WPF及Silverlight版FlexGrid支持ComponentOne的这一新的ClearStyle技术，让您轻松的在不需要改变控件模版的情况下，改变控件的颜色。通过设置几个颜色属性，您可以快速改变控件的样式。

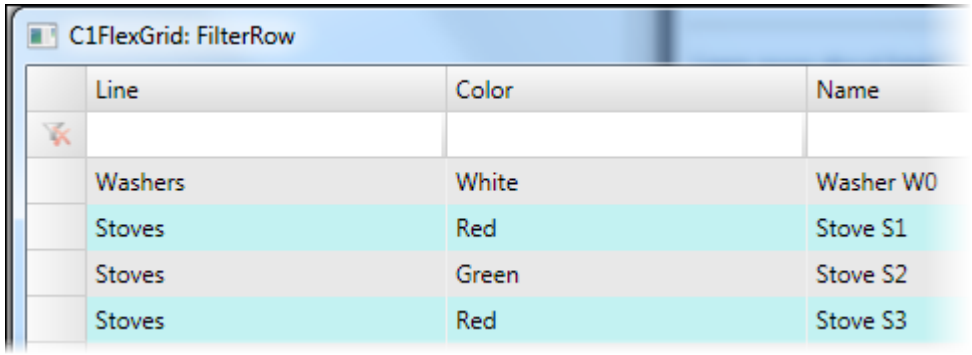
下表概述了C1FlexGrid 控件的画刷属性：

画刷	描述
Background	获取或设置控件背景的画刷。
AlternatingRowBackground	获取或设置用来绘制奇数行背景的System.Windows.Media.Brush画刷。
BottomRightCellBackground	获取或设置用来绘制右下角单元格背景的System.Windows.Media.Brush画刷。
ColumnHeaderBackground	获取或设置用来绘制列头背景的System.Windows.Media.Brush画刷。
ColumnHeaderForeground	获取或设置用来绘制列头内容的System.Windows.Media.Brush画刷。
ColumnHeaderSelectedBackground	获取或设置用来绘制选中单元格所在的列头背景的System.Windows.Media.Brush画刷。
CursorBackground	获取或设置用来绘制光标单元格背景的System.Windows.Media.Brush画刷。
CursorForeground	获取或设置用来绘制光标单元格前景的System.Windows.Media.Brush画刷。
EditorBackground	获取或设置在编辑模式下，用来绘制单元格背景的System.Windows.Media.Brush画刷。
EditorForeground	获取或设置在编辑模式下，用来绘制单元格前景的System.Windows.Media.Brush画刷。
FrozenLinesBrush	获取或设置用作绘制grid中，冻结区和滚动区域之间的分隔线的

	System.Windows.Media.Brush画刷。
GridLinesBrush	获取或设置用来绘制单元格之间的线形的System.Windows.Media.Brush画刷。
GroupRowBackground	获取或设置用来绘制分组行背景的 System.Windows.Media.Brush画刷。
GroupRowForeground	获取或设置用来绘制分组行前景的 System.Windows.Media.Brush画刷。
HeaderGridLinesBrush	获取或设置用来绘制行和列头单元格之间线形的 System.Windows.Media.Brush画刷。
RowBackground	获取或设置用来绘制行背景的System.Windows.Media.Brush画刷。
RowHeaderBackground	获取或设置用来绘制行头背景的System.Windows.Media.Brush画刷。
RowHeaderForeground	获取或设置用来绘制行头前景的System.Windows.Media.Brush画刷。
RowHeaderSelectedBackground	获取或设置用来绘制选中单元格所在行的行头背景的 System.Windows.Media.Brush画刷。
SelectionBackground	获取或设置用来绘制出了光标单元格之外选中的单元格背景的 System.Windows.Media.Brush画刷。
SelectionForeground	获取或设置用来绘制出了光标单元格之外选中的单元格前景的 System.Windows.Media.Brush画刷。
TopLeftCellBackground	获取或设置用来绘制grid左上角单元格背景的System.Windows.Media.Brush画刷。

请注意以上描述中，到属性的引用链接指向WPF版本；对于Silverlight版本，请参见Silverlight命名空间下具有相同名称的成员。

你可以通过设置一个或多个属性，彻底地改变C1FlexGrid控件的外观，例如，如果你设置AlternatingRowBackground属性的值为“#FFC3F2F2”，则C1FlexGrid外观类似下图所示：



10 金融类应用程序示例

本节介绍了一个在金融行业中应用的示例应用程序。财务应用程序通常依赖于大量的数据，通常是从强大的、专用的服务器中获得的。

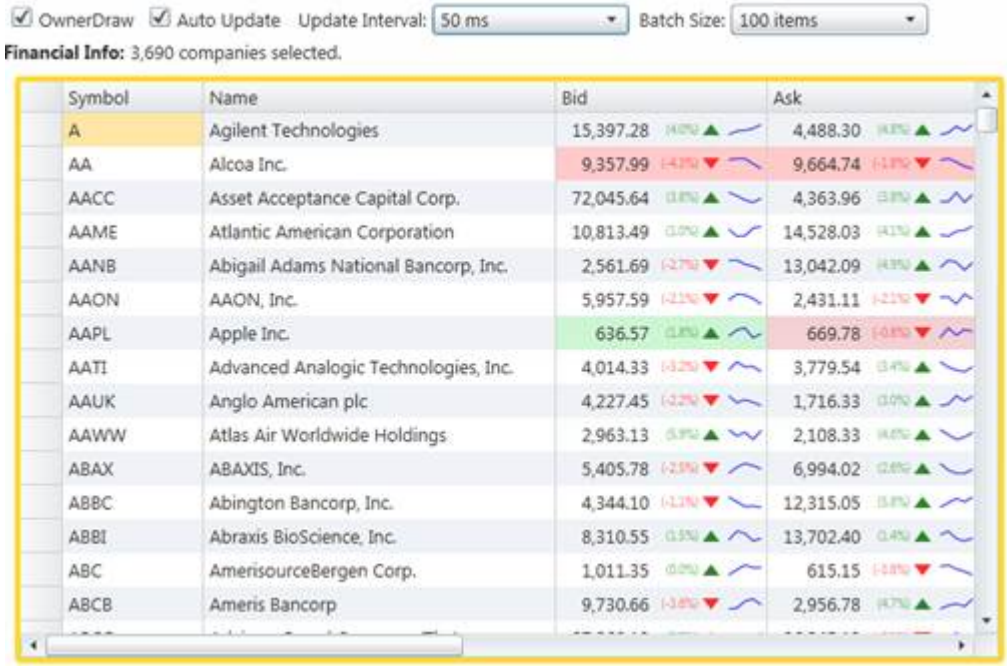
在这些应用中的信息的实时性要求快速更新（应用程序必须跟上来自服务器的数据流）和机制，以一个清晰，有效的方式向用户传达变化。

一种常用的方法来实时显示实时的变化是使用闪烁的元素。例如，当它的值改变时，grid单元可能会闪烁不同的颜色。闪光只持续短短的时间，就足以把用户的注意力吸引到该变化来。

另一个日益流行的机制，用于在一个快速和紧凑的形式传达丰富的信息，称作sparkline。一个Sparkline是一个小型的图表，用来显示趋势以及总结信息，相对于一个长长的数据列表，这将更加清晰有效。

本节描述了一个具有数据实时更新，闪烁单元格以及Sparkline的示例金融类应用程序。在Silverlight及WPF中提供的实力突出展示了C1FlexGrid的性能。

下面的图片显示我们的示例金融应用程序的作用。该图像无法展示应用程序动态变化以展示不断变化的数据的特性，因此我们建议您有机会尝试运行该示例。



10.1 产生数据

我们的金融类应用程序使用一个模拟实际的提供不断更新的动态数据的实际服务器。

由于我们并非金融业专家，我们就从维基百科上获得了一些启示（http://en.wikipedia.org/wiki/Market_data）。

我们的数据源由FinancialData对象组成，用来表示典型的来自于NYSE，TSX，Nasdaq股票市场的数据信息。每个数据对象都包含这样的信息：

股票代码	**IBM**
竞标价	89.02
询问价	89.08
竞标规模	300
询问大小	1000
最终销售价	89.06
最终size	200

股票代码	IBM
报价时间	14:32:45
交易时间	14.32.44
成交量	7808

事实上，这些信息通常是一个不同数据源的聚合，如报价数据（bid, ask, bid size, ask size），以及交易数据（last sale, last size, volume）均产生自不同的数据源。

为捕捉到的数据的动态性，我们的数据源对象提供了一个具有大约4000个FinancialData对象的FinancialDataList，以及一个计时器，按照预定的计划修改对象。调用方可以决定数值更新的频率以及单次更新发生时数值修改的范围大小。

将FinancialDataList绑定到grid，并在程序运行期间修改数据源更新参数，以查看grid如何做到紧跟数据源变化。

为检查数据源实现细节，请参见示例源代码中的FinancialData.cs文件。

将grid绑定至金融类数据源非常简单。我们创建了一个PagedCollectionView 做为一个中介的角色，提供常规的Currency，排序，分组，以及筛选服务，而不是直接将grid绑定到FinancialDataList。这里是代码：

C#

```
// 创建数据源
var list = FinancialData.GetFinancialData();
var view = new PagedCollectionView(list);

// 绑定数据源到grid
_flexFinancial.ItemsSource = view;
```

和前面的示例一样，我们设置AutoGenerateColumns属性为false并使用XAML创建grid的列：

XAML

```
<fg:C1FlexGrid x:Name="_flexFinancial"
  MinColumnWidth="10"
  MaxColumnWidth="300"
  AutoGenerateColumns="False" >
  <fg:C1FlexGrid.Columns>
    <fg:Column Binding="{Binding Symbol}"    Width="100" />
    <fg:Column Binding="{Binding Name}"      Width="250" />
    <fg:Column Binding="{Binding Bid}"       Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding Ask}"       Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding LastSale}"  Width="150"
        Format="n2" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding BidSize}"   Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding AskSize}"   Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding LastSize}"  Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding Volume}"    Width="100"
        Format="n0" HorizontalAlignment="Right" />
    <fg:Column Binding="{Binding QuoteTime}" Width="100"
        Format="hh:mm:ss" HorizontalAlignment="Center" />
    <fg:Column Binding="{Binding TradeTime}" Width="100"
        Format="hh:mm:ss" HorizontalAlignment="Center" />
  </fg:C1FlexGrid.Columns>
</fg:C1FlexGrid>
```

10.2 搜索及过滤

就像iTunes用户，金融分析师通常不会对一次查看全部的数据感兴趣，因此我们需要一些过滤或者搜索机制。

一个真实的应用程序允许分析师选择查看特定的页，并有可能将这些视图保存下来，并在它们中间切换查看。我们的示例采取了一些简单的方法，简单地重用了在iTuns示例（在线文档）中描述的SearchBox控件。我们的用户可以在搜索框中输入"bank"或者"electric"以筛选数据，以替代直接选择特定的数据项。

把搜索框关联到金融数据源的代码如下：

C#

```
// 创建数据源
FinancialDataList list = FinancialData.GetFinancialData();
var view = new PagedCollectionView(list);

// 绑定数据源到grid
_flexFinancial.ItemsSource = view;

// 关联搜索框（用户可以通过公司名称或代码搜索）
_srchBox.View = view;
var props = _srchCompanies.FilterProperties;
props.Add(typeof(FinancialData).GetProperty("Name"));
props.Add(typeof(FinancialData).GetProperty("Symbol"));
```

10.3 自定义单元格

如果你现在运行这个示例，grid已经工作并按照期望更新数据。您可以更改更新参数以使其更或更少的频繁，在更新过程中滚动grid，等等。
然而，虽然更新正在发生，他们很难理解。有太多的数字在屏幕上随机位置刷新。

自定义单元格通过闪烁以及sparkline提供了一个更好的用户体验。

当单元格包含的值发生改变时，闪烁动画将临时地改变单元格的背景。如果一个数值增加，该单元格将在瞬间变成绿色，然后逐渐的变淡，直至恢复成白色。

Sparkline是显示在每一个单元格中的微型图表。该微型图表展示了该单元格的最后五个值，因此用户可以立即识别出数值变化的趋势（该值在上升，下降，或者持平）。

要使用自定义单元格，我们继续在上一示例中进行修改。我们首先从创建一个FinancialCellFactory类并指定该类的一个实例至grid的CellFactory属性开始。

C#

```
// 使用自定义单元格工厂
_flexFinancial.CellFactory = new FinancialCellFactory();
// 自定义单元格工厂定义
public class FinancialCellFactory : CellFactory{
  public override void CreateCellContent(C1FlexGrid grid, Border bdr, CellRange
range)
{
    // 获取单元格信息
    var r = grid.Rows[range.Row];
    var c = grid.Columns[range.Column];
    var pi = c.PropertyInfo;
    // 检查这是一个我们期望的单元格
    if (r.DataItem is FinancialData && (pi.Name == "LastSale" || pi.Name == "Bid" ||
pi.Name == "Ask"))
   {
      // 创建 StockTicker 元素并添加至单元格
      var ticker = new StockTicker();
      bdr.Child = ticker;
     // 绑定StockTicker 至该行的FinancialData 对象
      var binding = new Binding(pi.Name);
      binding.Source = r.DataItem;
      binding.Mode = BindingMode.OneWay;
```

```
      ticker.SetBinding(StockTicker.ValueProperty, binding);
     // 添加一些信息至 StockTicker 元素
      ticker.Tag = r.DataItem;
      ticker.BindingSource = pi.Name;    }
    else
    {
      // 使用默认实现
      base.CreateCellContent(grid, bdr, range);    }  }
}
```

我们的自定义单元格工厂首先检查该行的数据是否是FinancialData，并且该列是否绑定到数据对象的LastSale, Bid, 或者 Ask属性。如果所有这些条件都满足，单元格工厂将创建一个新的StockTicker元素并绑定到数据。

StockTicker元素将数据显示给用户。它由一个包含以下子元素的四列的Grid元素组成：

元素描述	类型	名称
当前值	TextBlock	_txtValue
最后变化的百分比	TextBlock	_txtChange
上/下图标	Polygon	_arrow
Sparkline	Polyline	_sparkLine

这些元素在StockTicker.xaml文件中定义，我们这里不列举完整的清单。

StockTicker.xaml文件中最有趣的部分是用来实现闪烁行为脚本的定义。Storyboard逐渐地改变控件的背景色从当前值至透明：

XAML

```
<UserControl.Resources>  <Storyboard x:Key="_sbFlash" >    <ColorAnimation
Storyboard.TargetName="_root"      Storyboard.TargetProperty=                "
(Grid.Background).(SolidColorBrush.Color)"      To="Transparent"
Duration="0:0:1"    />  </Storyboard></UserControl.Resources>
```

StockTicker的实现代码包含在StockTicker.cs 文件中。有趣的部分注释如下：

C#

```
/// <summary>
/// Interaction logic for StockTicker.xaml
/// </summary>
public partial class StockTicker : UserControl
{
  public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register(
      "Value",
      typeof(double),
      typeof(StockTicker),
      new PropertyMetadata(0.0, ValueChanged));
```

我们从定义一个叫做ValueProperty的DependencyProperty开始，该属性将被用作绑定底层的数据值。Value属性实现如下：

C#

```
public double Value
{
  get { return (double)GetValue(ValueProperty); }
  set { SetValue(ValueProperty, value); }
}
private static void ValueChanged(DependencyObject d,
            DependencyPropertyChangedEventArgs e)
{
```

```
var ticker = d as StockTicker;
var value = (double)e.NewValue;
var oldValue = (double)e.OldValue;
```

为了实现sparkline，控件需要访问除了当前值和上一个值之外更多的值。这将通过在控件的Tag属性保存FinancialData对象完成，接下来调用FinancialData.GetHistory方法。

实际上，由事件的OldValue属性提供的前一个数值在这种情况下无论如何是靠不住的。这是由于grid将虚拟化显示单元格，StockTicker.Value可能会发生变化，因为控件由于滚动到可见区域刚刚被创建。在这种情况下，控件没有以前的值。从FinancialData对象获取之前的值也需要考虑这个问题。

C#

```
// 获取历史数据
var data = ticker.Tag as FinancialData;
var list = data.GetHistory(ticker.BindingSource);
if (list != null && list.Count > 1)
{
  oldValue = (double)list[list.Count - 2];
}
```

一旦这些值可用，控件将计算以百分比表示的上一次改变的变化率，并更新控件文本：

C#

```
// 计算以百分比表示的变化率
var change = oldValue == 0 || double.IsNaN(oldValue)
  ? 0
  : (value - oldValue) / oldValue;

// 更新文本
ticker._txtValue.Text = value.ToString(ticker._format);
```

百分比表示的变化率也用来更新up/down符号以及文本和闪烁的颜色。如果没有变化，则该up/down符号将被隐藏，并且文本被设置为默认颜色。
如果变化是负向的，则代码将通过设置其ScaleY变换为-1，使得up/down符号向下显示，同时设置符号的颜色，以及闪烁动画的颜色为红色。
如果变化是正向的，则代码将通过设置其ScaleY变换为+1，使得up/down符号向上显示，同时设置符号的颜色，以及闪烁动画的颜色为绿色。

C#

```
// 更新符号和闪烁动画的颜色
var ca = ticker._flash.Children[0] as ColorAnimation;
if (change == 0)
{
  ticker._arrow.Fill = null;
  ticker._txtChange.Foreground = ticker._txtValue.Foreground;
}
else if (change < 0)
{
  ticker._stArrow.ScaleY = -1;
  ticker._txtChange.Foreground = ticker._arrow.Fill = _brNegative;
  ca.From = _clrNegative;
}
else
{
  ticker._stArrow.ScaleY = +1;
  ticker._txtChange.Foreground = ticker._arrow.Fill = _brPositive;
  ca.From = _clrPositive;
}
```

下一步，代码通过使用由早先调用FinancialData.GetHistory方法提供的历史值数组更新sparkline多边形的Points属性，以更新sparkline。sparkline的Stretch属性被设置为Fill，因此线形将自动缩放以适应可用的空间大小。

C#

```
  // 更新sparkline
  if (list != null)
  {
    var points = ticker._sparkLine.Points;
    points.Clear();
    for (int x = 0; x < list.Count; x++)
    {
      points.Add(new Point(x, (double)list[x]));
    }
  }
```

最后，如果值真的发生了变化，而且控件不是刚刚被创建，则代码通过调用StoryBoard.Begin方法闪烁单元格

C#

```
  // 闪烁新值（但是不能是控件创建之后立刻就刷新）
  if (!ticker._firstTime)
  {
    ticker._flash.Begin();
  }
  ticker._firstTime = false;
}
```

总结一下StockTicker控件。如果您运行该示例应用程序，并选择“自定义单元格”复选框，您将立即看到一个具有更多信息显示的界面，单元格将在数值发生变化时闪烁显示，同时sparkline提供一个关于数值变化趋势的快速指示。