第 17 章　几何图形、图像和可视化层

在本书前面的内容中介绍了 WPF 中形状的运用、画刷或变换等一些基本元素的使用。尽管这些元素的功能非常简洁实用，但是如果为了绘制复杂的图形，会显得有些心有余而力不足。WPF 提供了几何图形绘制方法，使开发人员可以使用弧线或曲线进行绘图，也可以使用在绘图领域中应用广泛的贝塞尔曲线进行复杂的图形绘制。

17.1　路径和几何图形

在第 16 章中，介绍了派生自 Shape 的几个图形类。除此之外，还有一个特别重要的而且也是功能最强大的图形类 Path，也称为路径。使用 Path 类为绘图带来了无限的潜力。使用 Path 可以绘制本书前面章节介绍的任何图形或组合多个图形，而且也可以使用更复杂的元素，如曲线。

Path 类只有一个属性，名为 Data，该属性接收单个 Geometry 几何对象。也就是说，Path 使用几何对象来作为其结果呈现。因此要使用 Path，需要创建一个 Geometry 的派生对象，并将其赋值给 Path 对象的 Data 属性。Geometry 是一个抽象的基类，该类定义了如下所示的子类用于绘制图形。

- LineGeometry：用于绘制几何直线线条，等同于 Line 图形。
- RecgangleGeometry：用于绘制直角或圆角矩形，等同于 Rectangle 图形。
- EllipseGeometry：绘制一个椭圆形，等同于 Ellipse 图形。
- GeometryGroup：用于组合多个几何图形，可以使用 FillRule 来决定如何填充图形。
- CombinedGeometry：在一个形状内部结合两个几何图形，可以使用 CombineMode 属性选择结合方式，比如，并集、差集或交集。
- PathGeometry：可能由弧、曲线、椭圆、直线和矩形组成的复杂形状。
- StreamGeometry：PathGeometry 的轻量替代图形：它不支持数据绑定、动画或修改。

Path 和几何图形的区别在于，几何图形定义了形状，而 Path 对象则允许绘制这些形状，因而几何对象定义了形状的细节，比如坐标和尺寸的信息，Path 对象提供了 Stroke 和 Fill 画刷来进行绘制。此外，Path 类派生自 UIElement，因此提供了对于键盘和鼠标的响应支持。Geometry 类派生自 Freezable 基类，因此具有变更通知的支持。如果使用几何形状创建了一个路径，然后修改该几何形状，则 Path 将会自动的重绘以更新图形。

17.1.1　线型、矩形和椭圆几何图形

线、矩形和椭圆是 3 个非常基本的几何图形，这 3 个对象 LineGeometry、

RectangleGeometry 和 EllipseGeometry 类分别对应到派生自 Shape 的 Line、Rectangle 和 Ellipse。由于它们的基类并不相同，因此也具有如下的一些不同点。

- 由于 Shape 派生自 FrameworkElement，而 Geometry 派生自 Freezable，因此 Geometry 不能参与到布局系统，不会自行调整自身以适合布局。
- Geometry 对象比 Shape 对象更通用，例如 Geometry 对象可用于为二维图形定义几何区域、定义裁剪区域或者定义命中测试区域等。下面的代码演示了如何绘制这 3 个几何图形。

示例代码 17-1

```
  <Path Stroke="Black" StrokeThickness="1" Grid.Row="0" >
   <Path.Data>
       <!--绘制直线-->
       <LineGeometry StartPoint="10,20" EndPoint="100,20" />
   </Path.Data>
</Path>
<Path Stroke="Black" StrokeThickness="1" Grid.Row="1" >
   <Path.Data>
       <!--绘制矩形-->
       <RectangleGeometry Rect="10,0 100,50"></RectangleGeometry>
   </Path.Data>
</Path>
<Path Stroke="Black" StrokeThickness="1" Grid.Row="2" >
   <Path.Data>
       <!--绘制椭圆-->
      <EllipseGeometry RadiusX="50" RadiusY="25" Center="50,25">
      </EllipseGeometry>
   </Path.Data>
</Path>
```

与 Line 对象指定 X1、Y1、X2 和 Y2 不同，LineGeometry 需要指定 StartPoint 起始点和 EndPoint 终止点这两个属性。RectangleGeometry 需要获取用于描述尺寸和位置的 Rect 对象，前两个数字描述 X 和 Y 坐标点，后两个数字指定矩形的宽度和高度，RectangleGeometry 也具有 RadiusX 和 RadiusY 属性来指定圆角，EllipseGeometry 需要指定圆的水平和垂直半径以及中心点。

17.1.2 使用 GeometryGroup 组合形状

GeometryGroup 可以同时组合多个几何图形，这些图形具有同样的填充色和边框设置。下面的代码将 17.1.1 节中绘制的三个几何图形组合到一个 Path 对象中。

```
<Canvas>
   <Path StrokeThickness="1" Fill="Yellow" Stroke="Blue" Margin="5"
   Canvas.Top="10" Canvas.Left="10">
       <Path.Data>
           <GeometryGroup>
           <!--绘制直线-->
           <LineGeometry StartPoint="10,20" EndPoint="100,20" />
           <!--绘制矩形-->
           <RectangleGeometry Rect="120,0 100,50"></RectangleGeometry>
           <!--绘制椭圆-->
```

```
            <EllipseGeometry RadiusX="50" RadiusY="25" Center="50,80">
            </EllipseGeometry>
            </GeometryGroup>
        </Path.Data>
    </Path>
</Canvas>
```

使用 GeometryGroup，开发人员可以在一个元素中放置两个或多个图形，好处是可以减轻用户界面的负担。通常，一个使用了较少元素的界面要比使用了更多元素的界面具有更好的性能和更快的响应速度。此外还可以将 GeometryGroup 定义为资源，以便应用程序进行重用。例如下面的示例代码。

```
<Window.Resources>
    <!--定义窗口资源-->
    <GeometryGroup x:Key="GeometryGroupResource">
        <RectangleGeometry Rect="0 ,0 100 ,100"></RectangleGeometry>
        <EllipseGeometry Center="150, 50" RadiusX="35" RadiusY="25">
        </EllipseGeometry>
    </GeometryGroup>
</Window.Resources>
<Canvas>
    <Path Fill="Blue" Stroke="Red" Margin="5" Canvas.Top="140" Canvas.Left=
    "10" Data="{StaticResource GeometryGroupResource}">
    </Path>
</Canvas>
```

当多个图形交叠时，还可以使用 FillRule 属性来指定填充方式。在第 16 章中已经介绍过该属性，可选值有 EvenOdd 和 Nonzero，默认值是 EvenOdd。下面的代码将本小节开头的示例的填充规则更改为 Nonzero。

```
<Path StrokeThickness="1" Fill="Yellow" Stroke="Blue" Margin="5"
Canvas.Top="10" Canvas.Left="10">
    <Path.Data>
        <!--更改多个几何图形的填充规则为 Nonzero-->
        <GeometryGroup FillRule="Nonzero">
        <!--绘制直线-->
        <LineGeometry StartPoint="10,20" EndPoint="100,20" />
        <!--绘制矩形-->
        <RectangleGeometry Rect="120,40 100,50"></RectangleGeometry>
        <!--绘制椭圆-->
        <EllipseGeometry RadiusX="50" RadiusY="25" Center="180,60">
        </EllipseGeometry>
        </GeometryGroup>
    </Path.Data>
</Path>
```

运行效果如图 17.1 所示。

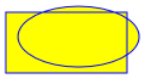

图 17.1　在 GeometryGroup 中指定 FillRule 效果

17.1.3 使用 CombinedGeometry 结合形状

尽管使用 GeometryGroup 可以组合多个图形，但是无法为组合的图形应用一些布尔运算的效果，比如并集、交集或差集，CombinedGeometry 类则提供了这样的功能。该对象具有一个 GeometryCombineMode 的枚举类型属性，具有如下所示的可选值。

- Union：通过采用两个区域的并集合并两个区域。所生成的几何图形为几何图形 A 和几何图形 B 的。
- Intersect：通过采用两个区域的交集合并两个区域。新的区域由两个几何图形之间的重叠区域组成。
- Xor：将在第一个区域中但不在第二个区域中的区域与在第二个区域中但不在第一个区域中的区域进行合并。新的区域由（A–B）+（B–A）组成，其中 A 和 B 为几何图形。
- Exclude：从第一个区域中除去第二个区域。如果给出两个几何图形 A 和 B，则从几何图形 A 的区域中除去几何图形 B 的区域，所产生的区域为 A–B。

注意：CombinedGeometry 中只能包含两个几何图形对象。

下面的示例代码演示了如何使用 CombinedGeometry 对象。

示例代码 17-2

```
<Path Stroke="Black" StrokeThickness="1" Fill="#CCCCFF">
   <Path.Data>
      <!-- 使用并集组合两个图形 -->
      <CombinedGeometry GeometryCombineMode="Union">
         <CombinedGeometry.Geometry1>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="75,75" />
         </CombinedGeometry.Geometry1>
         <CombinedGeometry.Geometry2>
            <EllipseGeometry RadiusX="50" RadiusY="50" Center="125,75"
            />
         </CombinedGeometry.Geometry2>
      </CombinedGeometry>
   </Path.Data>
</Path>
```

示例以及其他的几种 GeometryCombineMode 的效果如图 17.2 所示。尽管一个 CombinedGeometry 只能组合两个形状，但是开发人员也可以在一个 CombinedGeometry 对象中再嵌套多个 CombinedGeometry 对象。下面的代码使用嵌套的 CombinedGeometry 创建了一个禁止样式的的图形。

示例代码 17-3

```
<Path Fill="Yellow" Stroke="Blue">
   <Path.Data>
      <CombinedGeometry GeometryCombineMode="Union">
         <CombinedGeometry.Geometry1>
            <!--嵌套 CombinedGeometry-->
            <CombinedGeometry GeometryCombineMode="Exclude">
               <CombinedGeometry.Geometry1>
          <EllipseGeometry Center="50,50" RadiusX="50" RadiusY="50">
```

```
            </EllipseGeometry>
                </CombinedGeometry.Geometry1>
              <!--嵌套 CombinedGeometry-->
                <CombinedGeometry.Geometry2>
              <EllipseGeometry Center="50,50" RadiusX="40" RadiusY="40">
              </EllipseGeometry>
                </CombinedGeometry.Geometry2>
              </CombinedGeometry>
          </CombinedGeometry.Geometry1>
              <!--嵌套 CombinedGeometry-->
          <CombinedGeometry.Geometry2>
              <RectangleGeometry Rect="44,5 10,90">
                <RectangleGeometry.Transform>
              <RotateTransform Angle="45" CenterX="50" CenterY="50">
              </RotateTransform>
                </RectangleGeometry.Transform>
              </RectangleGeometry>
          </CombinedGeometry.Geometry2>
      </CombinedGeometry>
   </Path.Data>
</Path>
```

运行效果如图 17.3 所示。

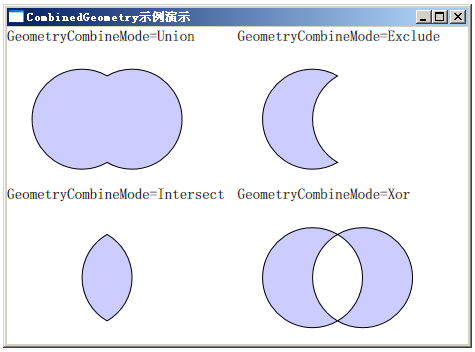

图 17.2　各种 GeometryCombineMode 的示例效果

图 17.3　嵌套的 CombinedGeometry 示例

17.1.4　认识 PathGeometry 对象

PathGeometry 是几何图形中功能最强大的几何元素，使用该对象可以绘制弧形、曲线、椭圆、直线和矩形组成的复杂形状等，当然其语法也是这几个几何元素中最复杂的。每个 PathGeomery 对象都使用一个和多个 PathFigure 对象，该对象存储在 PathGeometry.Figures 集合中。每个 PathFigure 对象都可由一个或多个 PathSegment 对象组成，例如 ArcSegment 和 LineSegment，这些对象定义了自己的形状。

PathFigure 具有如下所示的几个重要属性。

- ❑ StartPoint：指定线段的起始点。
- ❑ Segments：一个 PathSegment 对象的集合，用于绘制图形。
- ❑ IsClosed：如果设为 True，WPF 添加一个直线连接起始点和终止点。
- ❑ IsFilled：如果设为 True，图形的内部区域将使用 Path.Fill 画刷填充。

PathFigure 由一系列的线段组成，WPF 中提供了几个派生自 PathSegment 的类。一些比较简单的例如 LineSegment 用于绘制直线，复杂的同时也是功能最强大的是贝塞尔曲线

BezierSegment。可用的 PathSegment 派生类如下所示。

- LineSegment：在两个点之间创建直线。
- ArcSegment：在两个点之间创建圆弧。
- BezierSegment：在两个点之间创建贝塞尔曲线。
- QuadraticBezierSegment：在 PathFigure 的两点之间创建一条二次贝塞尔曲线。
- PolyLineSegment：创建一系列直线，可以使用多个 LineSegment 对象获得同样的效果，但是单个 PolyLineSegment 更简洁。
- PolyBezierSegment：创建一条或多条三次方贝塞尔曲线。
- PolyQuadraticBezierSegment：创建一系列二次贝塞尔线段。

下面分别举例说明如何绘制这些不同的形状。

17.1.5 用 PathGeometry 对象绘制直线

下面的代码使用 LineSegment 绘制了两条交叉的直线。

```
<Path Stroke="Blue">
   <Path.Data>
      <PathGeometry>
         <PathFigure IsClosed="True" StartPoint="10,10">
            <!--使用 LineSegment 绘制直线-->
            <LineSegment Point="10,100" />
            <LineSegment Point="100,50" />
         </PathFigure>
      </PathGeometry>
   </Path.Data>
</Path>
```

代码中 LineSegment 具有一个 Point 属性用于指定直线的终点，直线的起点已经由 PathFigure 的 StartPoint 进行指定，因此只需要为直线指定终点即可。由于 PathFigure.IsClosed 属性设置为 True，因而会有一条连接线连接起始点和终止点，最终形成一个三角形形状，如图 17.4 所示。

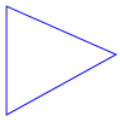

图 17.4 直线绘制示例

17.1.6 用 PathGeometry 对象绘制弧线

ArcSegment 可以绘制弧形，弧线形由下列元素定义：线的起点和终点、X 轴的半径和 Y 轴的半径、X 轴的旋转角度、一个指示弧是否应大于 180 度的值和一个描述弧的绘制方向的值。ArcSegment 类不包含弧的起点属性，它仅定义它所要表示的弧的目标点。弧的起点是 PathFigure（其中添加了 ArcSegment）的当前点，因此 ArcSegment 提供了如下所示的属性。

- Point：指定弧线的终止点。
- Size：弧线的 X 轴和 Y 轴的半径的 Size 结构。
- RotationAngle：弧线的旋转角度。
- IsLargeArc：弧的方向的角度值，如果弧应大于 180 度，则为 true；否则为 false。默认值为 false。

- SweepDirection：指定弧是以正向还是逆向绘制，可选值有两个。其中，Clockwise 表示正向，Counterclockwise 表示逆向。默认值为 Counterclockwise。

下面的代码创建了一条简单的弧线。

示例代码 17-4

```
<Path Stroke="Black" StrokeThickness="1" Grid.Row="1">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigureCollection>
                    <PathFigure StartPoint="10,10">
                        <PathFigure.Segments>
                            <PathSegmentCollection>
                                <!--绘制弧线-->
                                <ArcSegment Size="100,50" RotationAngle="45"
                                IsLargeArc="True" SweepDirection=
                                "CounterClockwise" Point="200,100" />
                            </PathSegmentCollection>
                        </PathFigure.Segments>
                    </PathFigure>
                </PathFigureCollection>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>
```

运行效果如图 17.5 所示。如果将弧线的 SweepDirection 属性设置为 Clockwise，则按相反的方向进行绘制，如图 17.6 所示。

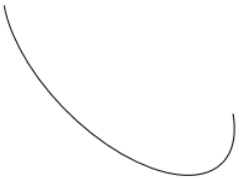

图 17.5　逆时针弧线示例

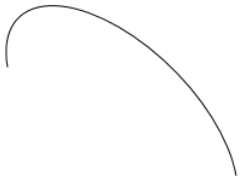

图 17.6　顺时针弧线

17.1.7　用 PathGeometry 对象绘制贝塞尔曲线

BezierSegment 对象用于绘制一条三次贝塞尔曲线。三次贝塞尔曲线由 4 个点定义：一个起点，一个终点（Point3）和两个控制点（Point1 和 Point2）。由于 PathFigure 已经定义了贝塞尔曲线的起点，因此在使用 BezierSegment 时，只需要定义终点和两个控制点。

三次贝塞尔曲线在很多绘图软件中都可以看到，它包含两个控制点。线条将沿着控制点的方向进行扭曲，但不一定会经过控制点，如图 17.7 所示。

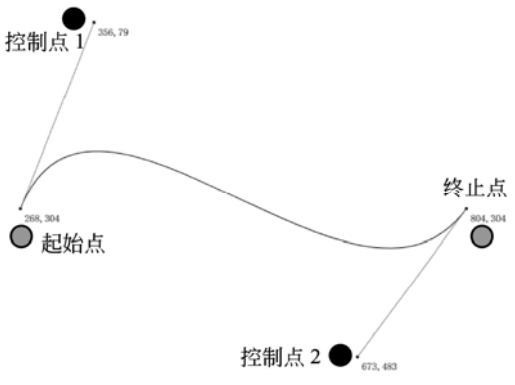

图 17.7　三次贝塞尔曲线

要使用 BezierSegment 创建贝塞尔曲线，需要指定如下所示的 3 个点。

- Point1 和 Point2：指定第一和第二个控制点。
- Point3：指定曲线的终点。

下面的示例代码演示了如何创建一条三次贝塞尔曲线，并同时绘制了到控制点的连接线和二个控制点。

示例代码 17-5

```
  <Path Stroke="Blue" StrokeThickness="5" Canvas.Top="20">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="10,10">
                <!--绘制贝塞尔曲经-->
                <BezierSegment Point1="130,30" Point2="40,140"
                               Point3="150,150"></BezierSegment>
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
<Path Stroke="Green" StrokeThickness="2" StrokeDashArray="5 2" Canvas.
Top="20">
    <Path.Data>
        <GeometryGroup>
            <!--添加控制点 1 和控制点 2 的连线-->
            <LineGeometry StartPoint="10,10" EndPoint="130,30">
            </LineGeometry>
            <LineGeometry StartPoint="40,140" EndPoint="150,150">
            </LineGeometry>
        </GeometryGroup>
    </Path.Data>
</Path>
<Path Fill="Red" Stroke="Red" StrokeThickness="8" Canvas.Top="20">
    <Path.Data>
        <GeometryGroup>
            <!--绘制两个控制点-->
            <EllipseGeometry Center="130,30"></EllipseGeometry>
            <EllipseGeometry Center="40,140"></EllipseGeometry>
        </GeometryGroup>
    </Path.Data>
</Path>
```

运行效果如图 17.8 所示。

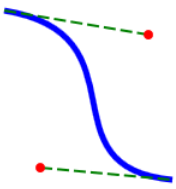

图 17.8　贝塞尔曲线示例

17.1.8　使用几何迷你语言

到目前为止，本书所介绍的示例都相对简单，图形的定义只需要指定几个点即可。大多数复杂的图形具有相同的思路，但是可能需要成百上千个线段。如果分别地定义这些直线、曲线和弧线，将非常繁琐而且有些没必要。通常使用 Expression Blend 等设计工具来实现这些复杂图形而很少自己手工编写代码。为了简化复杂图形的标记语法，WPF 提供了一种简洁的语法用于描述几何图形的定义。这种语法常称为几何迷你语言，有时也称为路径迷你语言。

迷你几何语言实际上是使用一个字符串来存放一系列的命令。WPF 的类型转换器读取这个字符串，然后创建相应的几何图形。每个命令是单个字符，有可能跟随一些用空格分

开的数字信息。命令之间使用空格分隔。

例如，为了创建在上一小节中创建的三角形，使用迷你几何语言，则代码如下所示。

```
<Path Stroke="Blue" Data="M 10,10 L 10,100 L 100,50 Z"/>
```

这个路径使用了 4 个命令，M 创建了一个 PathFigure 并设置其起始点为 10，10。接下来的两个 L 创建了 LineSegment 对象，指定各自的终止点，最后的 Z 设置 IsClosed 属性为 True。当使用几何迷你语言创建了几何对象后，实际上是创建了一个 StreamGeometry 对象而非 PathGeometry。因而，以后不可以使用代码进行修改。开发人员也可以选择使用下面的语法显示地创建一个 PathGeometry 对象。

不难发现，几何迷你语言实际上是由一些字符命令的组合来简化代码的编写。如表 17.1 所示列出了 WPF 中的几何迷你语言中的命令。

表 17.1　几何迷你语言命令

命　　令	描　　述
M x,y	创建一个新的 PathFigure 并在 x 和 y 中指定其开始点。除了 F 之外，该命令也必须放在字符串的开头。在绘图时，也可以使用该命令转移到绘制的开始点
L x,y	创建一个 LineSegment，x 和 y 用于指定终止点
H x	创建一个水平的 LineSegment，使用特定的 x 值，但 y 值保持不变
V y	创建一个垂直的 LineSegment，使用特定的 y 值，但 x 值保持不变
A radiusX,radiusY degrees isLargeArc,isClockwise x,y	创建一个 ArcSegment，指定 Arc 的椭圆半径，度数和 IsLargeArc 以及 IsClockwise 属性，并指定终止点
C x1,y1,x2,y2,x,y	创建贝塞尔曲线，x1，y1 和 x2，y2 用于指定控制点，x 和 y 指定终止点
Q x1,y1,x,y	创建二次贝塞尔曲线，指定一个控制点和一个终止点
S x2,y2,x,y	创建平滑的贝塞尔曲线，使用上一个命令的终止点做为起始点，x2，y2 指定控制点
Z	终止当前的 PathFigure，并设置 IsClosed 为 True。如果不想将 IsClosed 设为 True，则不用指定此命令。如果需要开始一个新的 PathFigure，可以使用 M 或者是终止字符串

17.1.9　几何图形的裁切

裁切是指将特定的元素强制性的放在一个特定的几何图形中，使用几何图形，可以为元素应用裁切特性。在 WPF 中，所有的元素都提供了一个 Clip 属性。该属性接收一个 Geometry 类型的几何图形作为其裁剪边界。下面的代码演示了两种裁剪定义的 XAML 实现。

示例代码 17-6

```
<!--定义几何图形资源-->
<Window.Resources>
   <GeometryGroup x:Key="clipGeometry" FillRule="Nonzero">
      <EllipseGeometry RadiusX="75" RadiusY="50" Center="100,150">
      </EllipseGeometry>
      <EllipseGeometry RadiusX="100" RadiusY="25" Center="200,150">
      </EllipseGeometry>
```

```
        <EllipseGeometry RadiusX="75" RadiusY="130" Center="140,140">
        </EllipseGeometry>
    </GeometryGroup>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
    </Grid.RowDefinitions>
    <!--使用属性元素语法设置裁剪效果-->
    <Image Source="SL381417.jpg"
           Width="200" Height="150" HorizontalAlignment="Center">
        <Image.Clip>
            <EllipseGeometry
               RadiusX="100"
               RadiusY="75"
               Center="100,75"/>
        </Image.Clip>
    </Image>
    <!--获取资源应用裁剪效果-->
    <Image Source="SL381106.jpg" Grid.Row="1"
    Stretch="Fill"
           Clip="{StaticResource clipGeometry}"
           HorizontalAlignment="Center"/>
</Grid>
```

运行效果如图 17.9 所示。

图 17.9　Clip 功能示例效果

Clip 有一个限制，几何图形不会随着窗口或图像的尺寸变化而变化。如果将上面示例中的窗口大小进行调整，会看到图像的大小自动进行变化，而用于裁切的几何图形则纹丝不动。

17.2　绘　　图

在 17.1 节学习了使用 Path 和 Geometroy 几何图形进行绘图。在 WPF 中，还存在一个 Drawing 抽象基类。可以视该类为一个轻量级的绘图对象，允许在 WPF 中将几何形状、图像、文本和多媒体内容绘制到应用程序中。之所以称之为轻量级对象是因为 Drawing 的派生类不提供对布局和输入事件以及焦点的支持。Drawing 对象具有较好的性能优势，因而通常作为背景和裁切图形的理想选择。WPF 中具有如下所示的几种类型的 Drawing 对象用于绘制不同类型的内容。

- GeometryDrawing：使用几何对象绘制形状。
- ImageDrawing：绘制图像。
- GlyphRunDrawing：绘制文本。
- VideoDrawing：播放音频或视频文件。
- DrawingGroup：绘制其他绘图。使用绘图组可以将其他绘图组合为一个复合绘图。

17.2.1　绘制形状

GeometryDrawing 可用于绘制几何图形，其功能与在 17.1 节中学习过的几何图形相似，

但是语法却有较大差别。如果要将图 17.4 中的三角图形使用 GeometryDrawing 绘制出来，则需要使用如下代码。

示例代码 17-7

```
<Button Name="btn1" Content="演示 GeometryDrawing">
   <Button.Background>
      <!--在按钮的背景上绘制图形-->
      <DrawingBrush>
         <!--使用 DrawingBrush 显示图形-->
         <DrawingBrush.Drawing>
            <!--指定 GeoetryDrawing 的画刷和笔形-->
            <GeometryDrawing Brush="Yellow">
               <GeometryDrawing.Pen>
                  <Pen Brush="Blue" Thickness="3"></Pen>
               </GeometryDrawing.Pen>
               <GeometryDrawing.Geometry>
                  <!--使用 PathGeometry 绘制三角形-->
                  <PathGeometry>
                     <PathFigure IsClosed="True" StartPoint="10,100">
                        <LineSegment Point="100,100" />
                        <LineSegment Point="100,50" />
                     </PathFigure>
                  </PathGeometry>
               </GeometryDrawing.Geometry>
            </GeometryDrawing>
         </DrawingBrush.Drawing>
      </DrawingBrush>
   </Button.Background>
</Button>
```

示例代码在为按钮的背景绘制了一个蓝色边框、黄色填充的三角形。派生自 Drawing 的类并不是元素，因而如果直接放在用户界面中的画，VS 2008 会提示错误，需要使用如表 17.2 所示的三个类来显示绘图。

表 17.2　显示绘图的类

类　名　称	基　　类	描　　述
DrawingImage	ImageSource	允许在一个 Image 元素内部宿主绘图
DrawingBrush	Brush	允许在一个画刷中包含一个绘图，使用该对象，可以在任何元素表面进行绘图
DrawingVisual	Visual	允许在一个低级别的可视对象上放置一个绘图，Visual 对象没有元素对象的性能开销，但是也能显示图形

上面的示例使用 DrawingBrush 作为图形的宿主，可以看到 GeometryDrawing 定义了一个 Pen 属性，用于绘制形状的轮廓。Pen 对象定义了 Thickness 和 Brush，用于定义轮廓的粗线和画刷。也可以通过指定 DashStyle 、 DashCap 、 LineJoin 、 StartLineCap 和 EndLineCap，创建更复杂的 Pen。GeometryDrawing 的 Geometry 用于指定一个几何图形，可以使用 17.1 节介绍的几何图形对象来绘制。上面示例的运行效果如图 17.10 所示。

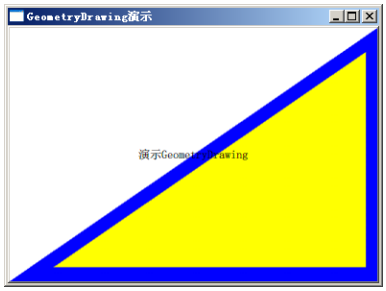

图 17.10　GeometryDrawing 示例效果

17.2.2 绘制图像

ImageDrawing 提供了绘制图像的功能，可以使用 ImageSource 属性来指定要绘制的图像，使用 Rect 属性定义绘制图像的区域。例如下面的代码在矩形的 50，50 位置处绘制一个大小为 100，100 的图像。

```
<Rectangle Width="250" Height="100" Stroke="Black" StrokeThickness="2">
   <Rectangle.Fill>
      <DrawingBrush>
         <DrawingBrush.Drawing>
            <!--使用 ImageDrawing 绘制图像-->
         <ImageDrawing Rect="50,50,100,100" ImageSource="SL381106.jpg"/>
          </DrawingBrush.Drawing>
      </DrawingBrush>
   </Rectangle.Fill>
</Rectangle>
```

也可以在 Image 对象内部使用 ImageDrawing 进行绘图。下面的代码在 Image 控件内部同时绘制了 4 幅图像。

示例代码 17-8

```
<Border BorderBrush="Gray" BorderThickness="1"
HorizontalAlignment="Left" VerticalAlignment="Top"
Margin="20">
   <Image Stretch="None">
      <Image.Source>
         <DrawingImage>
            <DrawingImage.Drawing>
               <DrawingGroup>
                  <!-- 在 75，75 位置处绘制 100，100 的图像 -->
                  <ImageDrawing Rect="75,75,100,100" ImageSource=
                  "SL381106.jpg"/>
                  <!-- 在 0，150 位置处绘制 25，25 的图像 -->
                  <ImageDrawing Rect="0,150,25,25" ImageSource=
                  "InsertPictureHH.bmp"/>
                  <!-- 在 150，0 位置处绘制 25，25 的图像 -->
                  <ImageDrawing Rect="150,0,25,25" ImageSource=
                  "MoveToFolderHH.bmp"/>
                  <!-- 在位置 0，0 处绘置 75，75 的图形 -->
                  <ImageDrawing Rect="0,0,75,75" ImageSource=
                  "MultiplePagesHH.bmp"/>
               </DrawingGroup>
            </DrawingImage.Drawing>
         </DrawingImage>
      </Image.Source>
   </Image>
</Border>
```

运行效果如图 17.11 所示。

图 17.11 在 Image 中绘制图像

17.2.3 组合绘制

DrawingGroup 允许开发人员将多个绘图组合为一个复合绘图，使用该对象可以将形状、图像和文本组合到一个绘

图对象中。下面的代码在一个矩形中使用 DrawingGroup 绘制了两个椭圆和一个图像。

```
<Rectangle Width="300" Height="300">
   <Rectangle.Fill>
      <DrawingBrush>
         <DrawingBrush.Drawing>
            <!--使用 DrawingGroup 组合多个绘图-->
            <DrawingGroup>
               <GeometryDrawing Brush="#66B5F314">
                  <GeometryDrawing.Geometry>
                     <!--绘制一个椭圆形-->
                     <EllipseGeometry Center="50,50" RadiusX="50"
                     RadiusY="50"/>
                  </GeometryDrawing.Geometry>
                  <GeometryDrawing.Pen>
                     <Pen Brush="Black" Thickness="4" />
                  </GeometryDrawing.Pen>
               </GeometryDrawing>
               <!--绘制一幅图像-->
               <ImageDrawing ImageSource="SL381106.jpg" Rect="50,50,
               100,100"/>
               <!--绘制几何图形-->
               <GeometryDrawing Brush="#66B5F314">
                  <GeometryDrawing.Geometry>
                     <!--绘制椭圆形-->
                     <EllipseGeometry Center="150,150" RadiusX="50"
                     RadiusY="50"/>
                  </GeometryDrawing.Geometry>
                  <GeometryDrawing.Pen>
                     <Pen Brush="Black"
                     Thickness="4" />
                  </GeometryDrawing.Pen>
               </GeometryDrawing>
            </DrawingGroup>
         </DrawingBrush.Drawing>
      </DrawingBrush>
   </Rectangle.Fill>
</Rectangle>
```

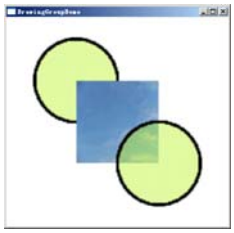

图 17.12　DrawingGroup 示例效果

运行效果如图 17.12 所示。

17.3　可 视 化 层

使用本章前面介绍的知识，可以实现非常复杂的图形绘制。但是对于一些需要大量使用图形元素的应用程序来说，比如一些图形软件需要根据数据库中的大量数据来生成分析动态图或者是模拟物理学中的粒子碰撞效果，这些程序往往需要大量的图形。即便是使用前面介绍的几何图形、路径等仍然会占用非常多的系统资源，造成应用程序较大的性能开销。

为了解决这个问题，WPF 提供了可视化层 API 作为一个低层次的编程方式。事实上，所有的图形元素都被实现在可视化层次上。可视化层 API 允许开发人员编写代码按需输入内容。Visual 对象是所有其他可视元素的基类，用于呈现可视化对象，提供对象执行转换、可视对象裁剪、命中测试等功能。Visual 对象不具有事件处理、布局、样式、数据绑定和全球化的功能。

17.3.1 在 Visual 上绘图

Visual 本身是一个抽象基类。为了在其上进行绘图，需要使用从其派生的几个类，比如 UIElement、Viewport3DVisual 用于呈现 3D 内容，ContainerVisual 放置其他可视化对象的基本容器，最有用的派生类是 DrawingVisual。该类派生自 ContainerVisual，并且添加了绘制图形内容的功能。

在表 17.2 中，列出了使用 DrawingVisual 可以在 Visual 对象上直接绘图，提供了较好的性能开销。这是开发一些具有复杂图形呈现又不需要事件处理、布局、样式和数据绑定以及全球化功能的应用程序的最好的绘图位置。为了使用 DrawingVisual 进行绘图，首先需要调用 DrawingVisual.RenderOpen 方法返回一个 DrawingContent 类型的对象，然后调用该对象提供的绘图方法进行图形绘制，最后调用 DrawingContext.Close 方法关闭绘图。如下代码所示。

```
DrawingVisual visual = new DrawingVisual();
DrawingContext dc = visual.RenderOpen();
dc.Close();//在这里实现图形的绘制
```

DrawingContext 类提供了大量的与绘图相关的方法，如表 17.3 所示。

表 17.3 DrawingContext方法

方 法 名	说 明
DrawDrawing	绘制指定的 Drawing 对象
DrawEllipse	绘制椭圆
DrawGeometry	使用指定的 Brush 和 Pe 绘制指定的 Geometry
DrawGlyphRun	绘制指定的文本
DrawImage	将图像绘制到由指定的 Rect 定义的区域中
DrawLine	使用指定的 Pen 绘制一条线
DrawRectangle	绘制一个矩形
DrawRoundedRectangle	绘制圆角矩形
DrawText	在指定位置绘制格式化文本
DrawVideo	将视频绘制到指定区域内
Pop	弹出推送到绘制上下文上的最后一个不透明蒙板、不透明度、剪辑、效果或转换操作
PushClip	将指定的剪辑区域推送到绘图上下文上
PushEffect	将指定的 BitmapEffect 推送到绘图上下文上
PushGuidelineSet	将指定的 GuidelineSet 推送到绘图上下文上
PushOpacity	将指定的不透明度设置推送到绘图上下文上
PushOpacityMask	将指定的不透明蒙板推送到绘图上下文上

从表 17.3 中可以看到，DrawingContext 的绘图具有 Pop、Push 和 Drawing 3 大类方法。尽管看起来与在本章前面内容介绍的绘图方式一致，但是 DrawingContext 的绘图功能大不相同，前面介绍的图形绘制是用于及时模式图形系统。当下达绘图命令时，图形及时地绘制在宿主上。而 DrawingContext 对象的命令在绘图时，实际是在存储一系列的呈现指令而

不是及时地绘制到屏幕上，以供图形系统在以后使用。

注意：DrawingContext 是抽象类，不要直接进行实例化，可以通过 DrawingGroup.Open 和 DrawingVisual.RenderOpen 等方法获取绘图上下文。

下面的代码在绘图上下文对象中添加了一个矩形。

```
// 创建一个包含一个矩形的 DrawingVisual
private DrawingVisual CreateDrawingVisualRectangle()
{
    DrawingVisual drawingVisual = new DrawingVisual();
    //接收 DrawingContext 对象用于创建绘图上下文
    DrawingContext drawingContext = drawingVisual.RenderOpen();
    Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
    //在绘图上下文中创建一个矩形
    drawingContext.DrawRectangle(Brushes.LightBlue, (Pen)null, rect);
    drawingContext.Close();     // 保存绘图上下文
    return drawingVisual;
}
```

该方法返回一个 DrawingVisual 对象，绘图指令保存在 DrawingVisual.Drawing 属性中，以便于在窗口需要时进行重绘。在 DrawingContext 中绘图的顺序十分重要，后面的绘图将总是输出在已存绘图的前面，使用 Pushxxx 之类的方法应用的设置将应用到后来的绘制操作。比如。如果使用 PuashOpacity 改变了透明度级别，则后续的绘图操作将总会受到影响。开发人员可以调用 Pop()方法来取消最近一次的 Push()调用。因此可以使用后续的 Pop()方法调用来取消前次对 Push()方法的调用。下面一小节将讨论如何在宿主容器中使用 DrawingVisual 对象。

17.3.2　DrawingVisual 宿主容器

为了呈现 DrawingVisual 对象，需要创建一个宿主容器，该容器必须派生自 FrameworkElement 使之提供 DrawingVisual 所缺乏的布局和事件支持。通常需要实现如下几个步骤。

（1）调用 AddVisualChild()和 AddLogicalChild()方法来注册 Visual，使之呈现在视觉树和逻辑树中。

（2）重载 VisualChildrenCount 属性并返回添加的 Visual 的数量。

（3）重载 GetVisualChild()方法，当需要指定索引的 Visual 时返回该 Visual。

创建 DrawingVisual 最好的办法是创建一个自定义的类来包装 Visual 对象的显示。下面的代码演示了如何创建一个自定义的 Visual 宿主类。该类派生自 FrameElement，并在该类中同时创建了 3 个 DrawVisual 对象依序添加到 VisualCollection 集合中，代码如下所示。

示例代码 17-9

```
// 创建一个派生自 FrameworkElement 的 Visual 宿主
// 该类为 Visual 对象提供了布局、事件处理和容器支持
public class MyVisualHost : FrameworkElement
{
    // 创建子元素的集合私有变量
    private List<Visual> visuals = new List<Visual>();
```

```
//构造函数中添加 DrawingVisual 对象
public MyVisualHost()
{
    visuals.Add(CreateDrawingVisualRectangle());
    visuals.Add(CreateDrawingVisualText());
    visuals.Add(CreateDrawingVisualEllipses());
}
// 创建一个包含矩形的 DrawingVisual
private DrawingVisual CreateDrawingVisualRectangle()
{
    DrawingVisual drawingVisual = new DrawingVisual();
    // 获取 DrawingContext 以便于创建一个绘图上下文
    DrawingContext drawingContext = drawingVisual.RenderOpen();
    Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
    // 绘制一个矩形
    drawingContext.DrawRectangle(Brushes.LightBlue, (Pen)null, rect);
    drawingContext.Close();     // 保存绘图上下
    return drawingVisual;
}
private DrawingVisual CreateDrawingVisualText()
// 创建一个包含文本的 DrawingVisual
{
    // 创建 DrawingVisual 实例
    DrawingVisual drawingVisual = new DrawingVisual();
    DrawingContext drawingContext = drawingVisual.RenderOpen();
    //在 DrawingContext 上下文中绘制格式化文本字符串
    drawingContext.DrawText(
       new FormattedText("单击",
          CultureInfo.GetCultureInfo("zh-cn"),
          FlowDirection.LeftToRight,
          new Typeface("Verdana"),
          36, Brushes.Black),
          new Point(200, 116));
    drawingContext.Close();
    // 关闭 DrawingContext 以保存 DrawingVisual 的变更
    return drawingVisual;
}
//创建一个包含矩形的 DrawingVisual
private DrawingVisual CreateDrawingVisualEllipses()
{
    DrawingVisual drawingVisual = new DrawingVisual();
    DrawingContext drawingContext = drawingVisual.RenderOpen();
    //绘制矩形
    drawingContext.DrawEllipse(Brushes.Maroon, null, new Point(430,
    136), 20, 20);
    drawingContext.Close();
    return drawingVisual;
}
// 重载 VisualChildrenCount()方法，反回当前的 DrawingVisual 数量
protected override int VisualChildrenCount
{
    get { return visuals.Count; }
}
// 提供所需要的 GetVisualChild()重载方法
protected override Visual GetVisualChild(int index)
{
    if (index < 0 || index >= visuals.Count)
        throw new ArgumentOutOfRangeException();
    return visuals[index];
```

```
    }
}
```

为了在窗口上呈现这个自定义的宿主对象，下面在 Window 类的 Loaded 事件处理器中添加了如下的代码，使 MyVisualHost 作为窗口中画布的子元素。

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    MyVisualHost visualHost = new MyVisualHost();
    //实例化 MyVisualHost 对象
    MyCanvas.Children.Add(visualHost);    //添加到 MyCanvas 中
}
```

在 Window1.xaml 中，添加了一些描述性的文本和一个名为 MyCanvas 的 Canvas 控件。代码如下所示。

示例代码 17-10

```
<Window x:Class="DrawingVisualDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DrawingVisual 示例" Height="480" Width="640" Loaded="Window_
    Loaded">
    <StackPanel>
        <TextBlock TextWrapping="Wrap" Margin="20" FontSize="12">
            本示例演示了使用 DrawingVisual 对象，在程序中，一个矩形
            、文本和圆形对象被绘制到 DrawingVisual 上，DrawingVisual
            被宿主到一个自定义的宿主对象中。
        </TextBlock>
        <Canvas Name="MyCanvas" Width="640" Height="480"
      Background="Cornsilk">
        </Canvas>
    </StackPanel>
</Window>
```

示例运行效果如图 17.13 所示。

图 17.13　DrawingVisual 示例演示

17.3.3　使用命中测试

命中测试是指确定某个几何图形或点值是否位于 Visual 的呈现内容内，从而可以实现用户界面行为。例如，可以获取鼠标当前的位置点是否在某个 Visual 呈现的图形上面，然后执行一些不同的行为。UIElement 类提供了 InputHitTest()方法，可用于针对使用给定坐标值的元素执行命中测试。对于可视化层来说，需要使用 VisualTreeHelper 类中的 HitTest()方法。该方法确定几何图形或点坐标值是否位于给定对象（如控件或图形元素）的呈现内容内。

VisualTreeHelper.HitTest 方法具有三个重载，最基本的 HitTest 方法需要获取一个要进行命中测试的点和一个要进行命中测试的顶层的 Visual 对象，例如下面的代码将使用最基本的 HitTest 方法检测 Canvas 中的鼠标点：

```
// 鼠标按下时，对当前的鼠标位置进行命中测试
public void OnMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
```

```
    Point pt = e.GetPosition((UIElement)sender);   //获取当前的鼠标坐标.
    HitTestResult result = VisualTreeHelper.HitTest(myCanvas, pt);
                                                   //实现命中测试
    if (result != null) //检测返回值
        //如果测试结果不为空，则执行相应的行为
}
```

HitTest 返回一个 HitTestResult 抽象类，用于提供命中测试返回值的基类。因此，如果要判断特定的命中元素时，需要进行强制类型的转换。

HitTest 另外的一个重载方法允许指定一个回调函数。在该函数中枚举命中检测的结果，然后返回一个 HitTestResultBehavior 的枚举值，用来指定命中检测的行为。可选值有：Stop，停止任何进一步的命中测试并从回调中返回；Continue，对可视化树层次结构中的下一个可视对象继续进行命中测试。下面演示使用回调函数的 HitTest 函数为 17.3.2 节的示例添加命中测试。当鼠标指针位于 DrawingVisual 上方时，会变化其透明度。首先在自定义的宿主容器的构造函数中添加一个 MouseLeftButtonUp 事件处理器，代码如下：

```
//构造函数中添加 DrawingVisual 对象
public MyVisualHost()
{
    visuals.Add(CreateDrawingVisualRectangle());
    visuals.Add(CreateDrawingVisualText());
    visuals.Add(CreateDrawingVisualEllipses());
    //添加 MouseLeftButtonUp 事件处理器
    this.MouseLeftButtonUp += new MouseButtonEventHandler(MyVisualHost_
    MouseLeftButtonUp);
}
```

在 MouseLeftButtonUp 中，首先获取鼠标所在的位置，然后调用 HitTest()方法进行命中测试，并指定了一个回调函数。代码如下所示。

```
void MyVisualHost_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    Point pt = e.GetPosition((UIElement)sender);
    // 获取当前鼠标所在的坐标位置
    // 调用 HitTest 初始化命中测试，并指定 myCallback 回调函数
    VisualTreeHelper.HitTest(this, null, new
    HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
 }
```

回调函数将获取 HitTest 的返回值做为参数，通过判断该参数，判断鼠标是否位于 DrawingVisual 对象上。如果是，则进行透明操作的处理，myCallback 代码如下所示。

```
//如果 Visual 对象被命中，设置其透明度指定其被命中
public HitTestResultBehavior myCallback(HitTestResult result)
{
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (((DrawingVisual)result.VisualHit).Opacity == 1.0)
            ((DrawingVisual)result.VisualHit).Opacity = 0.4;
        else
            ((DrawingVisual)result.VisualHit).Opacity = 1.0;
    }
    return HitTestResultBehavior.Stop; //停止命中测试
}
```

代码中使用 HitTestResult 的 VisualHit 属性来获取已被点击的可视对象，调用 GetType() 方法获取其类型与 DrawingVisual 相比较。如果相等，则先强制类型转换 VisualHit 属性值，并设置其 Opacity 为 0.4。如果 Opaticy 已经是 0.4，则设置其值为 1.0 表示不透明。运行效果如图 17.14 所示。

图 17.14　命中测试示例效果

17.4　小　　结

本章讨论了 WPF 中的 2D 绘图工具，首先介绍了路径和几何图形，介绍了几何线型、几何矩形和几何椭圆的使用方法。讨论了 GeometryGroup 和 CombinedGeometry 各自如何组织多个几何图形，并详细地介绍了 PathGeometry 对象，在其中绘制贝塞尔曲线等。同时对几何迷你语言和几何图形的裁切技术进行了示例讲解。接下来介绍了如何在 WPF 的元素上绘制图形，最后介绍了在 WPF 中出于性能优化考虑的可视化层，介绍了如何在上面绘图，如何编写自定义的可视化绘制宿主容器，并简要介绍了命中测试。