第 15 章　图形图像编程

在 Web 应用中，良好的图形图像的运用能够提升网站的友好度和易用性。.NET Framework 提供了图形图像编程的方法，开发人员可以运用图形图像编程技术进行良好的 Web 应用中图形图像布局和编程，也可以通过 GDI+实现类似 Photoshop 的功能。

15.1　图形图像基础

使用图形图像可以进行良好的页面布局，在现有的很多 Web 应用中，其应用程序的页面布局经常需要使用图像，这样能够让页面整体效果更加友好。用户会对界面友好的应用程序印象深刻从而会进行回访。ASP.NET 不仅能够进行图形图像显示，还能够使用 GDI+进行图形图像的绘制。

15.1.1　图像布局

在页面布局中，很多设计人员喜欢使用 CSS 设计，这样能够简化页面代码，将页面布局代码和页面代码相分离，从而提高了维护性。虽然随着技术的发展，越来越多的动态生成页面布局，以及动态生成图像的方法也越来越多的被开发人员和设计人员所认知，但是开发人员和设计人员还是比较喜欢使用 CSS 和 IMG 标签进行页面布局，这是因为 CSS 和 IMG 标签都比较简单，可以说是“轻量级”的，即不需要页面进行逻辑处理也不需要动态生成。

1．IMG 标签

IMG 标签是图像标签，IMG 标签属于 HTML 控件，在 Web 应用中可以看到在页面中包含大量的 IMG 标签用于图形图像显示，示例代码如下所示。

```
<body>
    <img src="autom.jpg" />
</body>
```

上述代码显式了一个名为 autom 的 JPG 图像到页面中，如图 15-1 所示。

图 15-1　插入图片

使用 IMG 标签能够轻松的为网页添加图片，IMG 标签包括以下常用属性：

- Src：图片的地址，可以是图片的相对地址也可以是绝对地址。
- Width：设定图片的宽度。
- Height：设定图片的高度。
- Alt：当图片显示不了时提示的字符。
- Border：图片的边框的宽度。
- Align：图片的周片文字的对齐方式。
- Title：当鼠标放在图片上出现的提示字符。

开发人员能够通过编写 Width 和 Height 属性进行图像的大小控制，也可以编写 Alt 属性当图片显示不了时进行提示，如图 15-2 和图 15-3 所示。

图 15-2　Width 属性

图 15-3　Alt 属性

2. CSS

通过 CSS 能够使用图像进行页面布局和样式控制。当需要使背景呈现渐变效果时，无需使用 JavaScript 进行控制，可以直接使用 CSS 和图像进行搭配使用即可。CSS 背景属性包括：

- 背景颜色属性（background-color）：该属性为 HTML 元素设定背景颜色。
- 背景图片属性（background-image）：该属性为 HTML 元素设定背景图片。
- 背景重复属性（background-repeat）：该属性和 background-image 属性连在一起使用，决定背景图片是否重复。如果只设置 background-image 属性，没设置 background-repeat 属性，在缺省状态下，图片既 x 轴重复，又 y 轴重复。
- 背景附着属性（background-attachment）：该属性和 background-image 属性连在一起使用，决定

图片是跟随内容滚动，还是固定不动。

- 背景位置属性（background-position）：该属性和 background-image 属性连在一起使用，决定了背景图片的最初位置。
- 背景属性（background）：该属性是设置背景相关属性的一种快捷的综合写法。

为了方便进行页面背景布局，可以使用 CSS 背景属性，示例代码如下所示。

```
<body style="background:#6094d7 url('bg.jpg') repeat-x;">
    <div style="width:800px; border:1px solid #333333; margin:0px auto; background:white">
        编写内容
    </div>
</body>
```

上述代码将 BODY 标签样式编写为背景颜色为#6094d7、背景图片编写为 bg.jpg 并且背景图片按照 x 轴重复。在 body 标签下方包含一个 DIV 标签，该标签作为主样式进行内容编写，编写内容后如图 15-4 所示。

图 15-4　使用图形进行布局

3．JavaScript 进行图像编程

HTML 图像控件支持 JavaScript 进行图像操作，可以为图像控件进行事件处理，JavaScript 代码如下所示。

```
<script type="text/javascript">
function cut()
{
    var pic=document.getElementById("pic1")                    //获取 ID 为 pic1 的图片的属性
    pic.width=100;                                             //设置图片的宽度
    pic.height=100;                                            //设置图片的高度
}
</script>
```

上述代码获取图片 ID 为 pic1 的图片属性，当触发该事件后，ID 为 pic1 的图片的宽度和高度将变为 100。为了让图片被单击时触发该事件，则应该在 IMG 标签中声明该事件，图片相应的代码如下所示。

```
<img alt="" height="323" src="autom.jpg" width="500" onclick="cut()" id="pic1"/>
```

上述代码定义了事件 onclick 并定义了图片 id 为 pic1，运行后如图 15-5 和图 15-6 所示。

图 15-5　原始图片

图 15-6　触发 JavaScript 事件后

使用 IMG 标签进行图形图像编程是非常简单的方法。IMG 标签和 CSS 配合能够进行页面布局，IMG 标签和 JavaScript 进行配合能够进行图形图像的处理，所以 IMG 标签是非常容易被学习和使用的。在 Web 应用开发中，大量的 IMG 被使用也说明 IMG 标签是一种高效率、门槛低的图形图像编程方法。

15.1.2　GDI+简介

虽然通过 IMG 标签和 CSS、JavaScript 相配合能够进行图形图像开发，但是其功能有限，并不能够进行高级的图形图像开发。高级的图形图像开发有点类似与 Photoshop 中图形处理，在 ASP.NET 中，可以使用强大的 GDI+进行图形图像开发，实现类似 Photoshop 中图形处理的功能。

GDI+是 Windows XP 中的一个子系统，它主要负责在显示屏幕和打印设备输出有关信息，它是一组通过 C++类实现的应用程序编程接口。GDI+的前身是 GDI，在 C++应用程序开发中，C++开发人员经常需要使用 GDI 进行窗口的绘制与重绘，在 Vista 操作系统之后的操作系统中，微软对图形图像编程进行了更新，在 Vista 等系统中，大量的使用了半透明、渐变、边缘模糊化等效果，这就要求在编程中强化图形图像渲染，如图 15-7 所示。

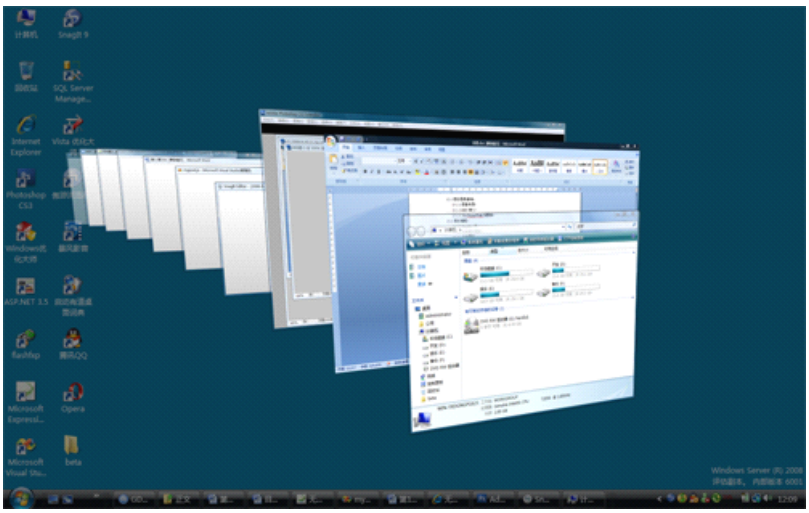

图 15-7　Vista 特效

为了适应更高的用户要求和用户体验的要求，微软对 GDI 进行了升级，就成为 GDI+。GDI+不仅能够在 C++开发当中使用，也能够在.NET 中使用 GDI+强大的绘图效果。GDI+相比与 GDI，进行了一些加强，这些加强功能如下所示。

- 渐变的画刷（Gradient Brushes）：GDI+允许开发人员使用渐变的画刷来绘制线条、图形以及外观。
- 基数样条函数（Cardinal Splines）：GDI+支持基数样条函数而 GDI 不支持，基数样条能够防止锯齿的出现，使得窗口以及图形的绘制能够平滑过渡。
- 持久路径对象（Persistent Path Objects）：在 GDI 中，绘制路径在窗口更改需要通过重绘来保持图形的持久化，而在 GDI+中，可以通过创建对个对象来持久化。
- 变形和矩阵对象（Transformations ＆ Matrix Object）：GDI+提供了强大的矩阵对象，开发人员可以通过矩阵对象进行图形的翻转、平移和缩放。
- 可伸缩区域（Scalable Regions）：GDI+允许在一定的范围内进行任何图形变换。

GDI+不仅包括这些新特性，还包括混合以及等多种图像类型支持等特性。ASP.NET 相对于 ASP 的强大之处就在于 ASP.NET 可以使用 GDI+进行图形图像编程，实现不同的 Web 应用功能。

15.1.3　绘制线条示例

通过 GDI+能够在 Web 应用中绘制线条，如果需要使用 GDI+，则首先需要引用命名空间 System.Drawing，示例代码如下所示。

```
using System.Drawing;                                                    //使用绘图命名空间
```

使用了命名空间后，就能够使用 System.Drawing 中的方法进行线条绘制，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap MyImage = new Bitmap(400, 400);                               //创建 Bitmap 对象
    Graphics gr = Graphics.FromImage(MyImage);                           //创建绘图对象
    Pen pen = new Pen(Color.Green, 10);                                  //创建画笔对象
    gr.Clear(Color.WhiteSmoke);                                          //格式化画布
    gr.DrawLine(pen, 50, 200, 400,20);                                   //绘制直线
    MyImage.Save(Response.OutputStream,System.Drawing.Imaging.ImageFormat.Jpeg);     //输出
    MyImage.Dispose();                                                   //释放对象
    gr.Dispose();
}
```

上述代码绘制了使用 GDI+的 Graphice 类的对象进行线条的绘制，当页面被载入时则会开始绘制线条。当线条开始绘制时，首先需要创建一个 Bitmap 对象，Bitmap 对象处理由像素定义的图像对象，示例代码如下所示。

```
Bitmap MyImage = new Bitmap(400, 400);                                   //创建 Bitmap 对象
```

上述代码创建了一个 Bitmap 对象并定义该对象区域为 400*400，Bitmap 就像画布。Graphice 对象能够在该区域内绘制图形图像，示例代码如下所示。

```
Graphics gr = Graphics.FromImage(MyImage);                               //创建绘图对象
```

定义了 Graphice 对象后就需要创建 Pen 对象进行绘制，Pen 对象就像画笔，能够在画布上绘制图形，示例代码如下所示。

```
Pen pen = new Pen(Color.Green, 10);                                      //创建画笔对象
```

上述代码定义了画笔的颜色和宽度，定义画笔后就需要清除整个绘图面并进行背景颜色填充，示例

代码如下所示。

```
gr.Clear(Color.WhiteSmoke);                                    //格式化画布
```

填充背景后，就能够使用 Graphice 的 DrawLine 方法进行线条绘制，示例代码如下所示。

```
gr.DrawLine(pen, 50, 200, 400,20);                             //绘制直线
```

上述代码则会在“画布”上绘制线条，运行后如图 15-8 所示。

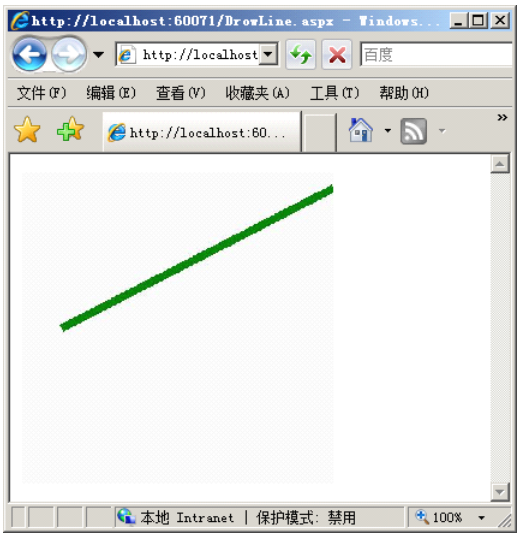

图 15-8　绘制线条

通过使用 GDI+的 Graphice 类能够在页面中绘制图形，Graphice 类还能够处理图片，实现锐化、底片等效果，Graphice 类将在后面的章节详细讲到。

15.1.4　.NET Framework 绘图类

GDI+包括很多的类，结构和枚举用于为开发人员提供快速进行图形图像开发提供保障和指导。

1．命名空间

GDI+包括很多的类、结构和枚举用于为开发人员提供图形编程，这些类、结构和枚举都定义在命名空间中，这些命名控件如下所示。

- System.Drawing：提供对 GDI+基本图形图像功能的访问，Graphice 包含在此命名空间中。
- System.Drawing.Drawing2D：提供高级的二维和矢量图形功能。
- System.Drawing.Imaging：提供高级的图像处理功能。
- System.Drawing.Text：提供高级的文字处理及排版功能。
- System.Drawing.Printing：提供图形打印所需要的类。
- System.Drawing.Design：提供开发 UI 设计时所需要的类。

这些命名空间为开发人员提供了图形图像编程的基本保障，其中最常用的是 System.Drawing，该命名空间提供了 Graphice 类进行图形图像处理。System.Drawing.Drawing2D 提供了高级的二维图形和矢量图形的处理功能，使用 System.Drawing.Drawing2D 能够进行二维图形和二维游戏的开发和编写。

System.Drawing.Imaging 命名空间主要提供了图像处理的功能，例如将图像进行锐化处理，或者将图像变成黑白色或底片都可以通过使用 System.Drawing.Imaging 命名空间的方法。System.Drawing.Text 命名空间提供了文字处理能力，通过 System.Drawing.Text 类能够实现 Word 中艺术字的效果。

2．类和方法

在.NET Framework 中，包括诸多的命名空间以保证开发人员能够快速的进行图形图像开发，在这些命名空间中，最常用的是 System.Drawing 命名空间，该命名空间提供的类如下所示。

- Bitmap：在 Bitmap 上使用图形工具，并在其中存储图形图像的绘图面板。
- Graphics：提供直线、曲线、多边形等绘画方法，也提供对一些位图的处理，例如平移、缩放等。
- Pen：提供直线、曲线等功能需要的画笔属性。
- Brush：提供文本填充和图形绘画，可以填充图形如圆形、椭圆形和多边形。
- Color：提供颜色的枚举，用于定义 Pen 和 Brush 的颜色。
- Font：提供文本的字体属性，定义文本的字体类型、样式和大小等。
- Point：用于定义有序的坐标对，这些坐标能够定义二维平面上的点。
- Size：定义区域的大小。
- Image：用于支持位图、指针和图标等文件类型。
- Rectangle：用于定义矩形区域。
- StringFormat：用于定义文本在位图上的对齐方式等属性。

简而言之，Bitmap 就相当于绘画时需要的纸，图形能够绘画到纸上面。而 Graphice 相当于绘画的人，因为人能够提供只写、曲线、多边形等绘画方法，而 Pen 和 Brush 相当于绘画工具，如铅笔、笔刷等，Color 就相当于是绘画所需要的颜料。

在绘画过程中，首先需要使用一张纸，固定到绘画板上，然后有一个人能够进行绘画，这个人能够进行素描、水彩等绘画。但是在绘画前，需要给这个人基本的工具，包括铅笔、笔刷和颜料盘等。在这些基本物质准备完毕后，就能够开始绘制了。GDI+的绘制过程与之非常的相似，首先需要定义一个画布，并通过构造函数定义画布的大小，示例代码如下所示。

```
Bitmap MyImage = new Bitmap(500, 500);                    //创建 Bitmap 对象
```

上诉代码定义了一个大小为 500*500 的画布，定义了画布之后，就需要有一个人进行绘画操作，示例代码如下所示。

```
Graphics gr = Graphics.FromImage(MyImage);                //创建 Graphics 对象
```

上述代码定义了一个 Graphics 对象并指定该对象在 MyImage 画布上进行绘画，定义了 Graphics 对象后，需要给该对象指定工具，示例代码如下所示。

```
Pen pen = new Pen(Color.Green, 10);                       //创建画笔
```

上述代码为对象指定了一个绿色的宽度为 10 的画笔，指定画笔后，这个“人”就能够进行绘画了。示例代码如下所示。

```
gr.DrawLine(pen, 50, 200, 400,20);                        //绘制直线
MyImage.Save(Response.OutputStream,System.Drawing.Imaging.ImageFormat.Jpeg);    //输出
```

通过 DrawLine 方法进行直线的绘画，绘画完毕需要执行 Save 方法把图片保存到文件中，或者使用 Response.OutputStream 将图片输出到 HTTP 流中在客户端浏览器中呈现。

15.2　图形编程

通过上面的章节了解了什么是 GDI+，以及如何使用 GDI+进行图形图像编程。在介绍了.NET Framework 绘图类所需要的命名空间和方法后，就需要了解如何使用相应的命名空间和方法进行图形图像的绘制和处理。

15.2.1 Graphics 类

Graphics 类在 GDI+的开发过程中非常重要，Graphics 类封装了 GDI+界面画图方法，以及图形显示设备，极大的简化了开发人员的编程过程。

1．Graphics 类的属性

Graphics 类的属性如下所示。

- DpiX：获取对象的水平分辨率。
- DpiY：获取对象的垂直分辨率。
- IsClipEmpty：为对象指定裁剪区域。
- IsVisibleClipEmpty；判断裁剪区域是否为空。
- TextGammaValue：返回一个提供文本灰度值的信息的整数值。
- TextRenderingHint：获取或设置与该图形相关联的文本着色模式。

通过 Graphics 类的属性能够获取 Graphics 对象的水平分辨率和垂直分辨率，并能够为 Graphics 对象进行裁剪区域的选择和判断。

2．Graphics 类的方法

Graphics 类提供的属性通常用于 Graphics 对象的信息获取，如果需要使用 Graphics 对象进行图形图像的绘制，则需要使用 Graphics 类提供的方法，Graphics 类的部分方法如下所示。

- Dispose：删除图形并释放已分配的内存。
- DrawArc：绘制弧线。
- DrawBezier：绘制后三次贝塞尔曲线。
- DrawClosedCurve：绘制封闭曲线。
- DrawCurve：绘制曲线。
- DrawEllipse：绘制椭圆。
- DrawIcon：绘制图标图像。
- DrawIconUnstretched：绘制图标图像，并可将图像缩放到指定大小。
- DrawImage：绘制图像。
- DrawImageUnscaled：绘制图像，并可将图像缩放到指定大小。
- DrawImageUnscaledAndClipped：在不进行缩放的情况下进行图像绘制。
- DrawLine：绘制线条。
- DrawLines：绘制一系列线条组。
- DrawPath：绘制 GraphicsPath。
- DrawPie：绘制扇形。
- DrawPolygon：绘制多边形。
- DrawRectangle：绘制矩形。
- DrawString：绘制字符串。
- Equals：判断两个 Object 类型是否相同。
- FillClosedCurve：填充封闭曲线的内部区域。
- FillEllipse：填充椭圆内部。
- FillPath：填充 GraphicsPath 内部。
- FillPie：填充扇形内部。

- GetHdc：获取图形上下文设备句柄。
- Restore：恢复图形状态。
- Save：保存图形。
- SetClip：为对象设置剪辑区域。

Graphics 类还包括其他方法提供对图形图像的绘制和处理进行编程，开发人员能够通过.NET Framework 提供的 Graphics 类进行图形任何图形图像的编程。开发人员不仅能够可以绘制现有图形，还可以对图形进行渲染处理，这些渲染处理的效果能够应用到图形、文字和图表上。

15.2.2 绘制基本图形

通过使用 Graphics 类，开发人员能够绘制基本图形，这些基本图形包括线条、矩形、椭圆和多边形 。

1．绘制线条

在上面的章节中讲解了如何进行线条的绘制，这里再进行简单的介绍，能够加深对 Graphics 类的理解，示例代码如下所示。

```
Bitmap images = new Bitmap(200, 200);                                    //创建画纸
Graphics gr = Graphics.FromImage(images);                                //创建 Graphics 对象
Pen pen = new Pen(Color.Red, 5);                                         //创建画笔
gr.Clear(Color.White);                                                   //设置画笔的颜色
gr.DrawLine(pen, 0, 0, 200, 200);                                        //开始绘画
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
gr.Dispose();                                                            //释放绘图对象
images.Dispose();                                                        //释放图形对象
```

上述代码使用了 DrawLine 方法进行直线的绘制，其中 DrawLine 包括五个需要传递的参数，这五个此参数分别为起点的坐标和终点的坐标以及画笔。在上述代码中，图像大小为 200*200，并在画布中从（0，0）坐标开始绘画到终点坐标为（200，200）的直线，绘制后的效果如图 15-9 所示。

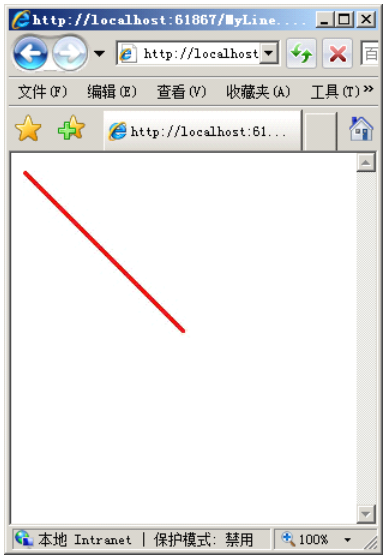

图 15-9　绘制线条形

2．绘制矩形

绘制矩形的方法同绘制线条基本相同，但是绘制矩形不仅要指定矩形的坐标，还需要指定矩形的高

度和宽度，示例代码如下所示。

```
Bitmap images = new Bitmap(400, 400);                                    //创建画纸
Graphics gr = Graphics.FromImage(images);                                //创建 Graphics 对象
Pen pen = new Pen(Color.Red, 5);                                         //创建画笔
gr.Clear(Color.White);                                                   //设置画笔颜色
gr.DrawLine(pen, 0, 0, 200, 200);                                        //绘制线条
gr.DrawRectangle(pen, 200, 200, 50, 50);                                 //绘制矩形
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
gr.Dispose();
images.Dispose();
```

从上面的代码不难看出，绘制矩形的方法同绘制线条的方法基本相同，绘制矩形使用了 DrawRectangle 方法进行绘制，DrawRectangle 方法同样包括五个参数，这五个参数同绘制线条有一点不同。绘制线条需要指定绘制过程中开始坐标和终点坐标，而绘制矩形时需要指定开始坐标，矩形的高度和宽度以及所需的画笔，示例代码如下所示。

```
gr.DrawRectangle(pen, 200, 200, 50, 50);                                 //绘制矩形
```

上述代码绘制了一个矩形，其开始坐标为（200，200），并且高度和宽度都为 50 的矩形，运行后如图 15-10 所示。

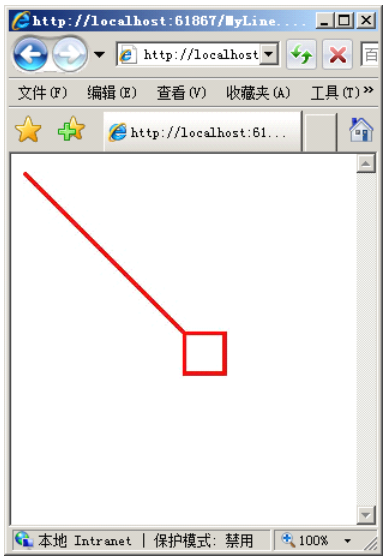

图 15-10　绘制矩形

3．绘制圆形和椭圆

绘制椭圆的方法只需要使用 DrawEllipse 方法即可，示例代码如下所示。

```
gr.DrawEllipse(pen, 0, 0, 300, 200);                                     //绘制椭圆
```

上述代码绘制了一个椭圆形，该椭圆形绘制的起点为（0，0），宽度为 300，高度为 200。DrawEllipse 方法同 DrawRectangle 方法基本相同，因为这两个方法都包括五个参数，这 5 个参数都需要指定绘制起点、宽度和高度。当需要绘制圆形时，只需要将宽度和高度设置相等即可，示例代码如下所示。

```
gr.DrawEllipse(pen, 0, 0, 200, 200);                                     //绘制圆
```

当设置宽度和高度相等时，该椭圆就会以圆形呈现，上述代码就实现了圆形的绘制。

4．绘制多边形

绘制多边形的方法只需要使用 DrawPolygon 方法即可，与绘制规则图形不同的是，绘制多边形需要指定多边形的各个节点，DrawPolygon 方法通过获取这些节点即可组成一个多边形，示例代码如下所示。

```
Point pt1 = new Point(50, 50);                                  //设置节点
Point pt2 = new Point(150, 150);                                //设置节点
Point pt3 = new Point(200, 200);                                //设置节点
Point pt4 = new Point(350, 170);                                //设置节点
Point pt5 = new Point(90, 30);                                  //设置节点
Point pt6 = new Point(180, 90);                                 //设置节点
Point[] pts = { pt1, pt2, pt3, pt4, pt5, pt6 };                 //设置节点组
gr.DrawPolygon(pen, pts);                                       //绘制多边形
```

Point 对象表示一个二维坐标中的一个点，通过 Point 的默认构造函数能够在二维坐标中指定一个点，示例代码如下所示。

```
Point pt1 = new Point(50, 50);                                  //设置节点
```

定义了平面上的一个点之后，需要为多边形定义多个点，多个点能够组成一个多边形，这些点组成一个二维坐标的集合，示例代码如下所示。

```
Point[] pts = { pt1, pt2, pt3, pt4, pt5, pt6 };                 //设置节点组
```

上述代码定义了一个 Point 类型的数组声明一个二维坐标的集合，通过 DrawPolygon 方法即可定义绘制多边形，示例代码如下所示。

```
gr.DrawPolygon(pen, pts);                                       //绘制多边形
```

5．绘制文字

通过使用 DrawString 方法能够绘制文字并呈现在图像中，示例代码如下所示。

```
Font font = new Font("宋体", 20);                               //创建文字对象
Brush brush=new SolidBrush(Color.Red);                          //创建笔刷对象
gr.DrawString("我的字符串", font, brush, 200,200);              //绘制文字
```

使用 DrawString 方法，需要对 DrawString 方法进行参数传递，DrawString 方法需要五个参数，其中包括需要输出的字符串、文本格式对象、笔刷对象以及文字开始绘制的坐标。上述代码中，输出字串为“我的字符串”。文本格式通过 Font 默认构造函数构造，并在坐标为(200,200)位置开始绘制。

15.2.3　图形绘制实例

上面一节讲解了如何分别绘制图形，使用相应的方法能够绘制相应的图形。.NET Framework 为图形图像开发进行了封装，使用.NET Framework 进行图形图像开发非常简单，下面绘制一些基本图形，并填充基本图形，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(400, 400);                       //创建画纸
    Graphics gr = Graphics.FromImage(images);                   //创建绘图类
    Pen pen = new Pen(Color.Red, 5);                            //创建画笔
    gr.Clear(Color.White);                                      //绘制直线
    gr.DrawLine(pen, 0, 0, 200, 200);                           //绘制矩形
    gr.DrawRectangle(pen, 200, 200, 50, 50);                    //绘制椭圆
    gr.DrawEllipse(pen, 0, 0, 300, 200);                        //绘制多边形
    Point pt1 = new Point(50, 50);                              //设置节点
    Point pt2 = new Point(150, 150);                            //设置节点
    Point pt3 = new Point(200, 200);                            //设置节点
    Point pt4 = new Point(350, 170);                            //设置节点
    Point pt5 = new Point(90, 30);                              //设置节点
    Point pt6 = new Point(180, 90);                             //设置节点
```

```
            gr.DrawPolygon(pen, pts);                                          //绘制文字
            Font font = new Font("宋体", 20);                                  //设置字体大小
            Brush brush=new SolidBrush(Color.Red);                             //创建红色笔刷
            gr.DrawString("我的字符串", font, brush, 200,200);                 //填充矩形
            SolidBrush brush2 = new SolidBrush(Color.YellowGreen);
            gr.FillRectangle(brush2,new Rectangle(100,100,100,100));           //填充矩形
            images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
            gr.Dispose();
            images.Dispose();
        }
```

上述代码分别绘制了直线、矩形、椭圆、文字和多边形，并填充了矩形，运行后如图 15-11 所示。

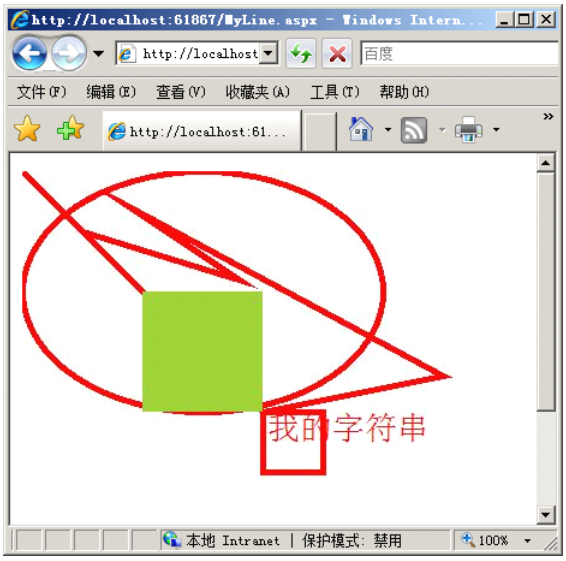

图 15-11　图形绘制实例

正如图 15-20 所示，开发人员能够使用.NET Framework 提供的绘图类进行基本图形的绘制，不仅如此，.NET Framework 绘图类还提供了文字和图片的特效绘制方法。

15.3　绘制文字特效

在 Word 中，文本编辑人员经常使用艺术字进行 Word 编辑和排版，艺术字在很大程度上丰富了排版功能和艺术效果，通过使用.NET Framework 绘图类能够实现文字的艺术化效果从而丰富页面中的文本显示效果。

15.3.1　投影特效

使用 System.Drawing.Drawing2D 和 System.Drawing.Text 能够进行文字投影特效。在制作文字投影特效前，首先需要使用命名空间 System.Drawing.Drawing2D 和 System.Drawing.Text。在实现投影效果前，首先需要了解如何制作投影。

投影特效的难度在于如何描述本体的影子。其实在画面上，影子是不可能像平常的描述一样呈现在图片上的，这也就是说，影子其实也是本体对象的另一种表现形式。首先，影子可以看作是本体的压缩

和平移，在对本体进行压缩和平移后，从一定的角度上看就好像是本地的影子。其次，影子是没有颜色的，通常用灰色输出即可实现影子的效果。在制作投影特效时，需要使用到 Matrix 类，该类需要使用 System.Drawing.Drawing2D 和 System.Drawing.Text 命名空间，示例代码如下所示。

```
using System.Drawing;                                    //使用绘图类
using System.Drawing.Drawing2D;                          //使用绘图类
using System.Drawing.Text;
```

引用了命名空间后，就可以使用相应的方法进行特效的制作，示例代码如下所示。

```
Bitmap images = new Bitmap(600, 150);                    //创建 Bitmap 对象
Graphics gr = Graphics.FromImage(images);                //创建 Graphics 对象
gr.Clear(Color.WhiteSmoke);                              //填充背景颜色
gr.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;  //设置文本输出质量
gr.SmoothingMode = SmoothingMode.AntiAlias;
Font newFont = new Font("宋体", 32);
Matrix matrix = new Matrix();                            //执行投射
matrix.Shear(-1.5f, 0.0f);                               //执行缩放
matrix.Scale(1, 0.5f);                                   //执行平移
matrix.Translate(130, 88);                               //执行坐标转换
gr.Transform = matrix;
SolidBrush grayBrush = new SolidBrush(Color.Gray);
SolidBrush colorBrush = new SolidBrush(Color.Red);
string text = "ASP.NET 3.5 开发大全";                     //设置文字
gr.DrawString(text, newFont, grayBrush, new PointF(0, 30));  //绘制阴影
gr.ResetTransform();                                     //图形变形
gr.DrawString(text, newFont, colorBrush, new PointF(0, 40));  //绘制前景
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

上述代码使用了 Matrix 类进行了倒影效果的实现，Matrix 类封装表示几何变换的 3 x 3 仿射矩阵。Matrix 类中常用的方法有：

- Shear：通过预先计算比例向量，将指定的比例向量应用到此矩阵。
- Scale：通过预先计算切变向量将指定的切变向量应用到此矩阵。
- Translate：通过预先计算转换向量，将指定的转换向量应用到此矩阵。

使用 Matrix 类能够进行对象的投射、缩放以及平移，并通过执行坐标转换呈现在图片中。作为投影特效，Matrix 类通过将现有的对象进行转换、压缩、平移，并通过 Graphics 对象的 DrawString 方法进行输出，使之看上去向文字的投影效果一样，如图 15-12 所示。

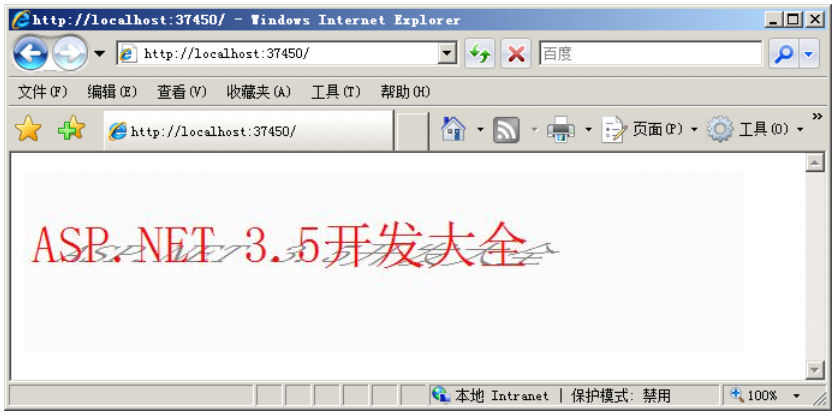

图 15-12　实现文字的投影效果

15.3.2 倒影特效

在 Photoshop 中，可以很容易的实现倒影效果，通过 GDI+同样也能够快捷的实现倒影效果。在实现倒影效果时，首先需要理清倒影是如何实现的，倒影同影子一样。在编程中，没有倒影的概念，其实倒影就是本体的另一种表现形式，与投影不同的是，投影是将本体进行压缩或拉伸，从一个角度来看成为影子的效果，而倒影其实就是本地的垂直旋转。

在进行垂直旋转后，倒影通常情况下颜色要比本体要淡，这样才能真实的模拟倒影的效果，实现倒影效果示例代码如下所示。

```
Brush shadowBrush = Brushes.LightBlue;                                    //创建倒影笔刷
Brush foreBrush = Brushes.Blue;                                           //创建本体笔刷
Font font = new Font("微软雅黑", Convert.ToInt16(40), FontStyle.Italic);   //配置字体
Bitmap images = new Bitmap(600, 150);
Graphics gr = Graphics.FromImage(images);                                 //创建 Graphice
gr.Clear(Color.WhiteSmoke);
string text = "ASP.NET 3.5 开发大全";                                      //设置文字
SizeF size = gr.MeasureString(text, font);                                //设置矩形大小
int posX = (600 - Convert.ToInt16(size.Width)) / 2;                       //设置平移坐标
int posY = (150 - Convert.ToInt16(size.Height)) / 2;                      //设置平移坐标
gr.TranslateTransform(posX, posY);                                        //执行转换
GraphicsState state = gr.Save();                                          //图形保存
gr.ScaleTransform(1, -1.0F);                                              //图形变换
gr.DrawString(text, font, shadowBrush, 0, -120);                          //输出倒影
gr.Restore(state);                                                        //图形重置
gr.DrawString(text, font, foreBrush, 0, 0);                               //输出本体
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

倒影效果相对与投影效果来说比较简单，倒影效果只是将 Graphics 对象的矩形大小进行转换，然后通过平移操作进行翻转操作，翻转完成后直接输出倒影，而又因为倒影输出的文字比本体输出的文字要淡，看上去很像倒影的效果，如图 15-13 所示。

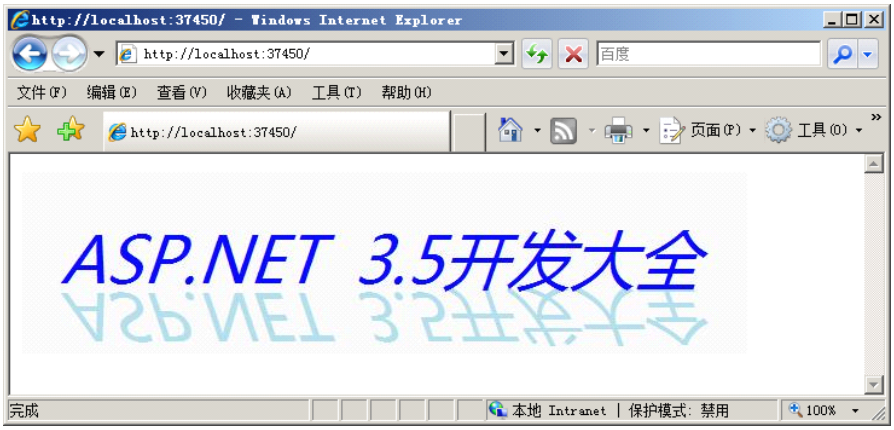

图 15-13 倒影效果

15.3.3 旋转特效

通过 GDI+能够通过非常简单的代码实现非常酷的效果，例如旋转特效就是一个非常酷的效果，在

进行旋转特效编写前，首先需要了解如何实现旋转特效的。如果要实现旋转特效，首先需要获取一段文字，该文字进行通过平移坐标原点进行变换，当需要实现旋转时，则通过循环不停的实现旋转平移，示例代码如下所示。

```
Bitmap images = new Bitmap(400, 400);                                    //创建Bitmap对象
Graphics gr = Graphics.FromImage(images);                                //创建绘图对象
gr.Clear(Color.WhiteSmoke);                                              //格式化画布
gr.SmoothingMode = SmoothingMode.AntiAlias;                              //设置边缘
for (int i = 0; i <= 360; i += 20)                                       //循环旋转
{
    gr.TranslateTransform(200, 200);                                     //变形
    gr.RotateTransform(i);                                               //按角度变形
    Brush brush = Brushes.Red;                                           //创建画笔
    Font font = new Font("微软雅黑", 12);                                //创建文字
    gr.DrawString("ASP.NET 3.5 开发大全 ", font, brush, 0, 0);           //绘制文字
    gr.ResetTransform();                                                 //重置变形
}
images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
```

上述代码运行后如图 15-14 和图 15-15 所示。

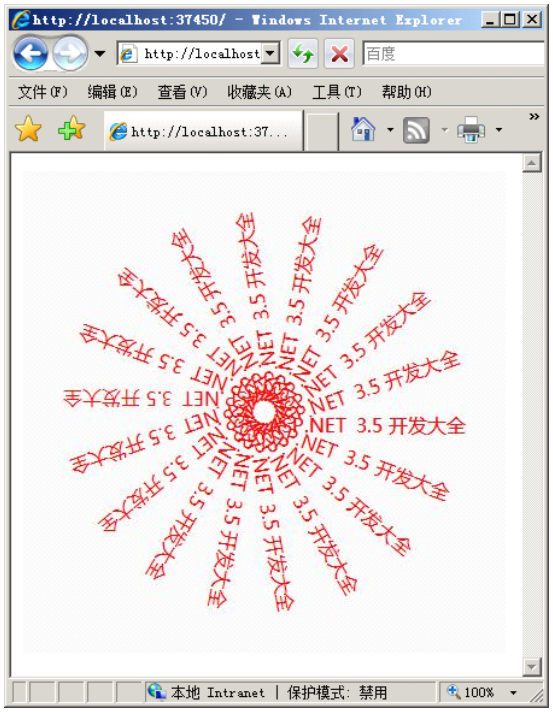

图 15-14　旋转特效

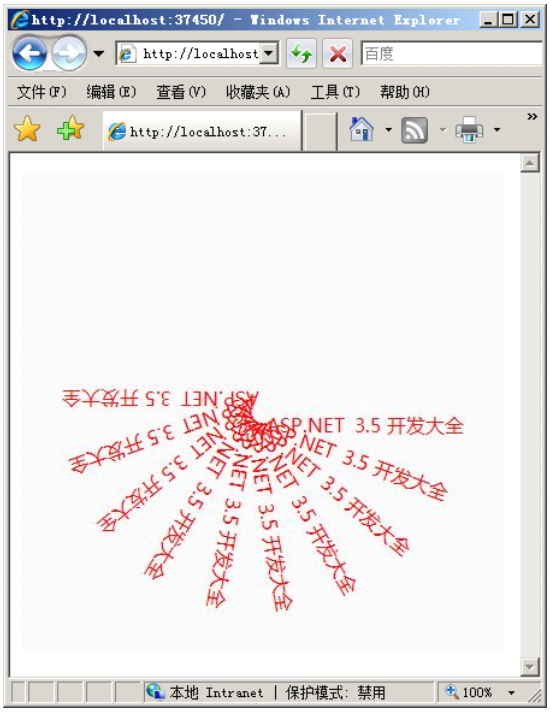

图 15-15　旋转半周

旋转特效就是按照循环不停的将本体进行循环并执行输出，在执行循环的过程中，因为圆的角度是 360 度的，所以循环时也需要循环 360 度，如果循环不被等于 360，则会出现半周的情况，如图 15-15 所示。

15.4 绘制图片

通过 IMG 标签能够插入图像，IMG 标签小巧而灵活，但是在如果需要使用 GDI+实现图形图像的渲染，IMG 标签所呈现的图形显然是不行的，ASP.NET 提供了 Image 控件用来创建图片，并能够通过 Image 控件进行图片编程。

15.4.1 载入图像文件

使用 Image 控件能够载入图像文件，拖动一个 Image 控件到页面，页面会自动生成 HTML 代码，示例代码如下所示。

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Image ID="Image1" runat="server" />
    </div>
    </form>
</body>
```

在控件的章节中，讲到 Image 控件包括以下常用属性：

- AlternateText：在图像无法显示时显示的备用文本。
- ImageAlign：图像的对齐方式。
- ImageUrl：要显示图像的 URL。

通过配置以上三种属性能够呈现不同的图片效果，配置完成后的图像控件示例代码如下所示。

```
        <asp:Image ID="Image1" runat="server" AlternateText="图片不存在"
            ImageUrl="~/autom.jpg" />
```

上述代码配置了 Image 控件的基本属性，包括 Image 控件所需要呈现的图像，以及 Image 控件呈现的图像不存在时需要提示的信息，运行后如图 15-16 和图 15-17 所示。

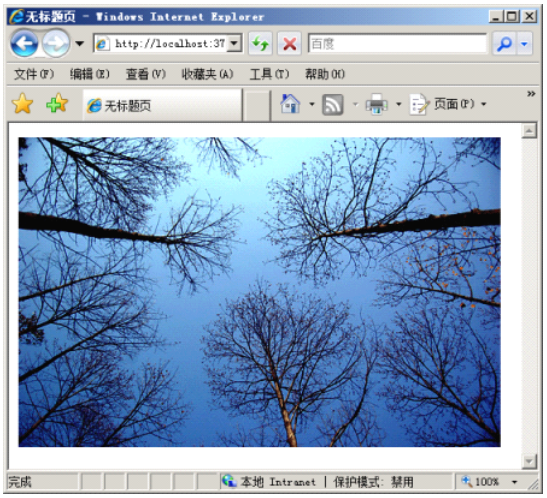

图 15-16 Image 控件

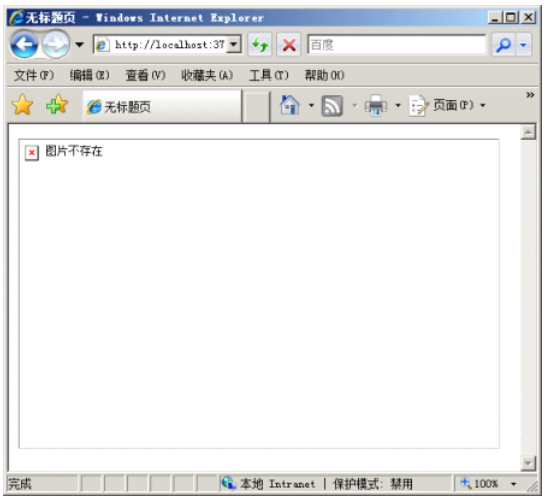

图 15-17 图片不存在

15.4.2 GDI+输出图像

使用 Image 控件能够快速的载入图形，但是 Image 控件并不支持 Click 事件，所以对 Image 控件是没有办法进行事件操作的，如果需要对图像进行事件操作并支持裁剪等高级方法时，可以采用 GDI+进行图形输出，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("autom.jpg"));          //读取现有图片
    images.Save(Response.OutputStream, images.RawFormat);             //格式化输出
    images.Dispose();                                                 //释放对象
}
```

上述代码使用了 Bitmap 类进行图形输出，Bitmap 类的默认构造方法能够载入现有的图片并执行输出。

注意：Bitmap 类的 RawFormat 属性能够直接返回现有文件的文件类型，在 Bitmap 的 Save 方法中可直接使用。

15.5 图像特效处理

相比与 IMG 标签而言，ASP.NET 能够通过 GDI+动态的创建图像并且进行图片特效处理。相对于文字处理而言，图片特效处理很像 Photoshop 中对图片的处理，开发人员能够实现不同的图片特效，如呈现底片效果、黑白效果等。

15.5.1 底片效果

通过 Photoshop 等软件能够快速的将图片制作成底片效果，但是在传统的图片处理领域中，只能通过软件进行图片效果的更改。在 ASP.NET 中，可以通过网页进行图片处理，包括底片、锐化等效果。

在进行底片效果制作前，首先需要了解底片效果是如何实现的，在图片显示中，其实是很多很多的点（像素）组成一个图片的，如果像素的数量很多，则图片显示的就清晰，如果像素数量较少，则图片看上去就不那么清晰。一个图片的组成是通过像素组成的，这也就是说，一个图片包括很多的小点进行组合，最后组合成图片，如图 15-18 所示。

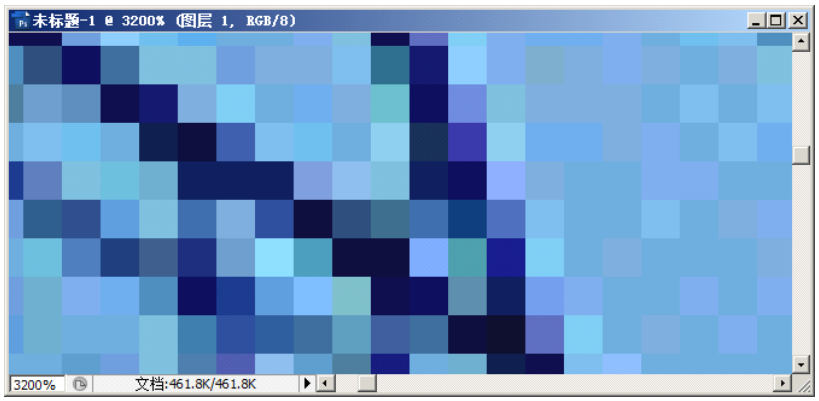

图 15-18 图片中的像素

在进行底片效果的制作时，只需要分别找到图片中的这些点，并获取这些点的像素的值，再取反保存即可，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("change.jpg"));                    //载入图片
    for (int i = 0; i < images.Width; i++)                                       //循环遍历宽
    {
        for (int j = 0; j < images.Height; j++)                                  //循环遍历高度
        {
            Color pix = images.GetPixel(i, j);                                   //获取图像像素值
            int r = 255 - pix.R;                                                 //像素值取反
            int g = 255 - pix.G;                                                 //转换颜色
            int b = 255 - pix.B;                                                 //转换颜色
            images.SetPixel(i, j, Color.FromArgb(r, g, b));                      //保存像素值
        }
    }
    images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
    images.Dispose();
}
```

上述代码通过循环遍历像素值并进行像素值取反则能够实现底片效果，运行效果前后如图 15-19 和图 15-20 所示。

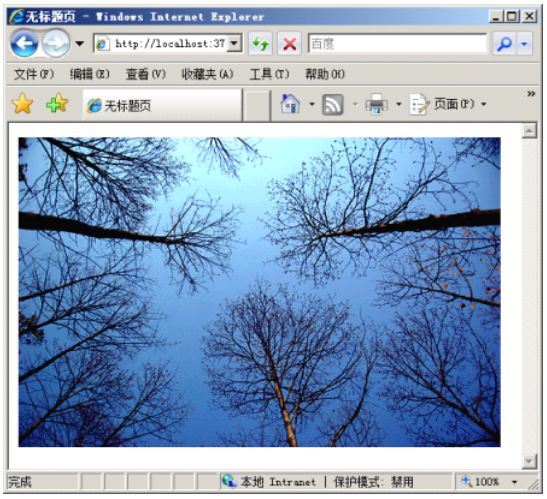

图 15-19　原图像

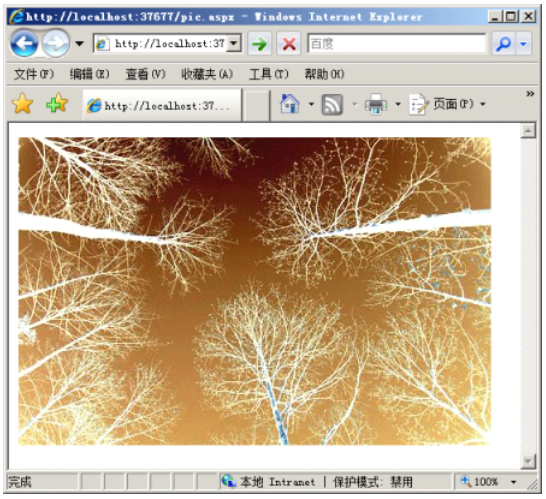

图 15-20　执行底片效果

在图像中，图像通常是使用 RGB 三原色进行图形渲染的，RGB 分别为红色、绿色和蓝色，图形通过协调这三原色进行图形渲染。RGB 在计算机中通常描述的范围是从 0-255 之间的某个数值的，所以当需要取反时，只需通过 255 减去现有的像素的色值即可。

15.5.2　浮雕效果

对于图片效果的更改和渲染都是通过修改像素的值来进行渲染的，当像素足够大时，人眼无法辨认其像素的多少，所以更改像素的大小对人眼来说是无法发觉的。

执行浮雕效果与底片效果实现手法非常类似，但是浮雕效果的实现与底片效果的实现中所需要使用

的算法又不尽相同。实现浮雕效果通常是将图像上每个像素点与其对角线的像素点形成差值，使相似颜色值淡化，不同颜色值之间保持突出，从而形成纵深感，达到浮雕的效果。在程序开发中，可以讲像素点的像素值与周边的像素值相减后加上 128，则可以呈现浮雕效果，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    Bitmap images = new Bitmap(Server.MapPath("change.jpg"));                    //载入图片
    for (int i = 0; i < images.Width-1; i++)                                     //循环遍历宽
    {
        for (int j = 0; j < images.Height-1; j++)                                //循环遍历高度
        {
            Color pix1 = images.GetPixel(i, j);                                  //获取图像像素值
            Color pix2 = images.GetPixel(i+1, j+1);                              //获取图像像素值
            int r = Math.Abs(pix1.R - pix2.R + 128);                             //实现浮雕效果
            int g = Math.Abs(pix1.G - pix2.G + 128);
            int b = Math.Abs(pix1.B - pix2.B + 128);
            r = check(r);                                                        //判断是否溢出
            g = check(g);
            b = check(b);
            images.SetPixel(i, j, Color.FromArgb(r, g, b));                      //设置像素值
        }
    }
    images.Save(Response.OutputStream, System.Drawing.Imaging.ImageFormat.Jpeg);
    images.Dispose();
}
```

在进行浮雕效果实现时，需要进行判断，判断转换后的值是否超过 255 或者小于 0，如果超过 255 则按照 255 进行处理，如果小于 0 则按照 0 进行处理。check 函数代码如下所示。

```
protected int check(int x)
{
    if (x > 255)                                                                 //如果像素值大于 255
    {
        return 255;                                                              //返回 255
    }
    else if (x < 0)                                                              //如果像素值小于 0
    {
        return 0;                                                                //返回 0
    }
    else
    {
        return x;                                                          //直接返回
    }
}
```

程序运行后，图片渲染前后效果如图 15-21 和图 15-22 所示。

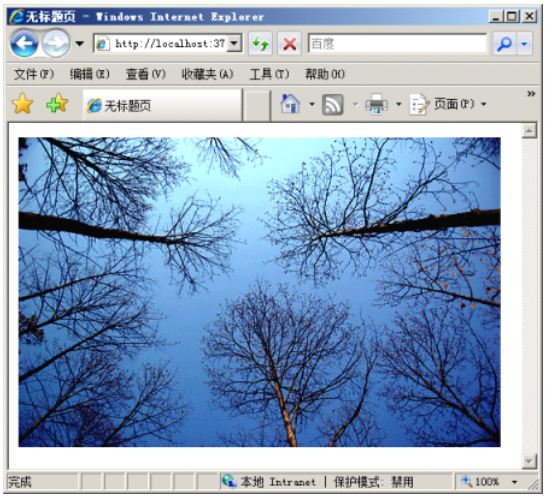

图 15-21　原图像

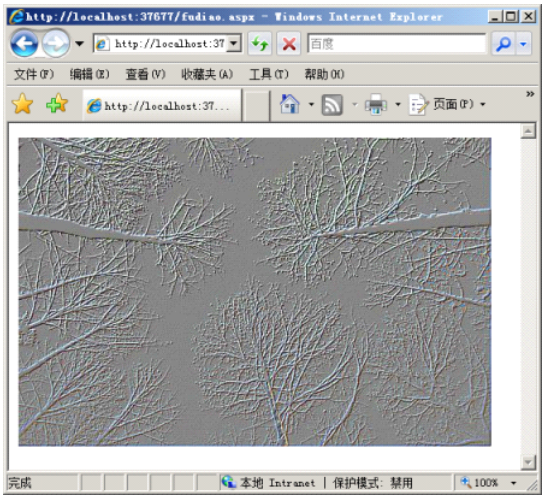

图 15-22　执行浮雕效果

15.6　小结

本章介绍了 ASP.NET 图形图像编程，通过 ASP.NET 图形图像编程能够在 Web 上执行图形图像的修改以及渲染。在页面中绘制图形图像包含很多方法，最简单的方法就是使用 Graphics 类中的方法进行图形的绘制，Graphics 类不仅提供了基本图形的绘制，还提供了图像、图标图像的绘制。GDI+看上去好像比较复杂，但是通过几个实例就能够了解其实 GDI+并不困难，在 ASP.NET 中，最常用的图形图像编程的方类属 Graphics 类了。本章还包括：

- 图形图像基础：介绍了图形图像编程基础。
- GDI+简介：介绍了 GDI+编程基础。
- .NET Framework 绘图类：介绍了.NET Framework 绘图类。
- 图形绘制实例：进行图形图像编程，通过使用现有类进行基本图形的创建。
- 绘制文字特效：使用 GDI+进行文字绘制和特效渲染。
- 图像特效处理：使用 GDI+进行图像特效的处理。

本章还介绍了图像显示的基本原理，以及图像特效的实现思路。