

# 第 48 章

## 使用 GDI+ 绘图

本章主要内容：

- 绘图规则
- 颜色和安全调色板
- 钢笔和笔刷
- 线条和简单图形
- BMP 图像和其他图像文件
- 绘制文本
- 字体和字体系列
- 处理打印

本书有许多章节介绍用户交互和 .NET Framework，第 39 章主要介绍了如何显示对话框或 SDI、MDI 窗口，以及如何把各种控件放在这些窗口上，如按钮、文本框和列表框。它还讨论了在 Windows 窗体中使用许多 Windows 窗体控件处理各种数据源中的数据。

这些标准控件的功能非常强大，使用它们就可以获得许多应用程序的完整用户界面。但是，有时还需要在用户界面上有更大的灵活性。例如，要在窗口的精确位置以给定的字体绘制文本，或者不使用图像框控件显示图像，或者绘制简单的形状和其他图形。这些都不能使用第 39 章介绍的控件来完成。要显示这种类型的输出，应用程序必须直接告诉操作系统需要在其窗口的什么地方显示什么内容。

在这个过程中，还需要使用各种帮助对象，包括钢笔(用于定义线条的特征)、画笔(用于定义区域的填充方式)和字体(用于定义文本字符的形状)。我们还将介绍设备如何识别和显示不同的颜色。

下面首先讨论 GDI+(Graphics Device Interface)技术。GDI+ 由 .NET 基类集组成，这些基类可用于在屏幕上完成自定义绘图，能把合适的指令发送到图形设备的驱动程序上，确保在屏幕上显示正确的输出(或打印到打印件中)。

### 48.1 理解绘图规则

本节讨论一些基本规则，只有理解了它们，才能开始在屏幕上绘图。首先概述 GDI，GDI+ 技术就建立在 GDI 的基础上，然后说明它与 GDI+ 的关系。接着介绍几个简单的例子。

48.1.1 GDI 和 GDI+

一般来说，Windows 的一个优点 (实际上通常是现代操作系统的优点) 是它可以让开发人员不考虑特定设备的细节。例如，不需要理解硬盘设备驱动程序，只需在相关的.NET 类中调用合适的方法 (在.NET 推出之前，使用等价的 Windows API 函数)，就可以编程读写磁盘上的文件。这条规则也适用于绘图。计算机在屏幕上绘图时，它把指令发送给视频卡。问题是市面上有几百种不同的视频卡，大多数有不同的指令集和功能。如果把这个考虑在内，在应用程序中为每个视频卡驱动程序编写在屏幕上绘图的特定代码，这样的应用程序就根本不可能编写出来。这就是为什么在 Windows 最早期的版本中就有 Windows 图形设备界面(Graphical Device Interface, GDI)的原因。

GDI 提供了一个抽象层，隐藏了不同视频卡之间的区别，这样就可以调用 Windows API 函数完成指定的任务了，GDI 会在内部指出在客户运行特定的代码时，如何让客户端的视频卡完成要绘制的图形。GDI 还可以完成其他任务。大多数计算机都有多个显示设备——例如，显示器和打印机。GDI 成功地使应用程序所使用的打印机看起来与屏幕一样。如果要打印某些东西，而不是显示它，只需告诉系统输出设备是打印机，再用相同的方式调用相同的 Windows API 函数即可。

可以看出，DC(设备环境)是一个功能非常强大的对象，在 GDI 下，所有的绘图工作都必须通过设备环境来完成。DC 甚至可用于不涉及在屏幕或某些硬件设备上绘图的其他操作，例如，在内存中修改图像。

GDI 给开发人员提供了一个相当高级的 API，但它仍是一个基于旧的 Windows API 并且有 C 语言风格函数的 API，所以使用起来不是很方便。GDI+在很大程度上是 GDI 和应用程序之间的一层，提供了更直观且基于继承性的对象模型。尽管 GDI+基本上是 GDI 的一个包装器，但 Microsoft 已经能通过 GDI+提供新功能了，它还有一些性能方面的改进。

.NET 基类库的 GDI+部分非常庞大，本章不解释其功能。这是因为只要解释库中的一小部分内容，就会把本章变成一个仅列出类和方法的参考指南。而理解绘图的基本规则更重要；这样您就可以自己研究这些类。当然，关于 GDI+中类和方法的完整列表，可以参阅 SDK 文档。



有 VB 6 背景的开发人员会发现，自己并不熟悉绘图过程涉及的概念，因为 VB 6 的重点是处理绘图的控件。有 C++/MFC 背景的开发人员则比较熟悉这个领域，因为 MFC 要求开发人员使用 GDI 更多地控制绘图过程。但是，即使您具备很好的 GDI 背景知识，也会发现本章有许多新内容。

1. GDI+名称空间

表 48-1 列出了 GDI+基类的主要名称空间。  
本章使用的几乎所有的类、结构等都包含在 System.Drawing 名称空间中。

表 48-1

名 称 空 间	说 明
System.Drawing	包含与基本绘图功能有关的大多数类、结构、枚举和委托
System.Drawing.Drawing2D	为大多数高级 2D 和矢量绘图操作提供了支持，包括消除锯齿、几何变形和图形路径

(续表)

名 称 空 间	说 明
System.Drawing.Imaging	包括有助于处理图像(位图、GIF 文件等)的各种类
System.Drawing.Printing	包含把打印机或打印预览窗口作为“输出设备”时使用的类
System.Drawing.Design	包含一些预定义的对话框、属性表和其他用户界面元素，与在设计期间扩展用户界面相关
System.Drawing.Text	包含对字体和字体系列执行更高级操作的类

2. 设备上下文和 Graphics 对象

在 GDI 中，识别输出设备的方式是使用设备上下文(DC)对象。该对象存储特定设备的信息，并能把 GDI API 函数调用转换为要发送给该设备的任何指令。还可以查询设备上下文对象，确定对应的设备有什么功能(例如，打印机是彩色的，还是黑白的)，以便据此调整输出结果。如果要求设备完成它不能完成的任务，设备上下文对象就会检测到，并采取相应的措施(这取决于具体的情形，可能抛出一个异常，或修改请求从而获得与该设备的功能最相近的匹配)。

DC 对象不仅可以处理硬件设备，还可以用作 Windows 的一个桥梁，因此能考虑到 Windows 绘图的要求或限制。例如，如果 Windows 知道只有一小部分应用程序的窗口需要重新绘制，DC 就可以捕获和取消在该区域外绘图的工作。因为 DC 与 Windows 的关系非常密切，所以通过设备上下文来工作就可以在其他方面简化代码。

例如，硬件设备需要知道在什么地方绘制对象，通常它们需要相对于屏幕(或输出设备)左上角的坐标。但应用程序可能使用自己的坐标系统，在自己的窗口的工作区(用于绘图的窗口)的特定位置上绘图。而因为窗口可以放在屏幕上的任何位置，用户可以随时移动它，所以在两个坐标系统之间转换就是一个比较困难的任务。然而，DC 总是知道窗口在什么地方，并能自动进行这种转换。

在 GDI+中，DC 包装在.NET 基类 System.Drawing.Graphics 中。大多数绘图工作都是调用关于 Graphics 实例的方法完成的。实际上，因为正是 Graphics 类负责处理大多数绘图操作，所以 GDI+中几乎没有操作不涉及 Graphics 实例。于是，理解如何处理这个对象是理解如何使用 GDI+在显示设备上绘图的关键。

48.1.2 绘制图形

下面用一个小示例 DisplayAtStartup 来说明如何在应用程序的主窗口中绘图。本章的示例都在 Visual Studio 2010 中创建为 C# Windows 应用程序。对于这种类型的项目，代码向导会提供一个类 Form1，它派生自 System.Windows.Forms，窗体表示应用程序的主窗口。还会生成一个类 Program(在 Program.cs 文件中)，它表示应用程序的主起点。除非特别声明，否则在所有的代码示例中，新代码或修改过的代码都添加到向导生成的代码中(可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 或随书附赠光盘中找到示例代码)。



在.NET 的用法中，当说到显示各种控件的应用程序时，“窗口”大都用术语“窗体”来代替，窗体表示一个矩形对象，它代表应用程序占据了屏幕上的一块区域。本章使用术语“窗口”，因为在手工绘图时，它更有意义。当谈到用于实例化窗体/窗口的.NET 类时，使用术语“窗体”。最后，“绘图”或“绘制”可以互换使用，以描述在屏幕或其他显示设备上显示一些项的过程。

第一个示例只创建一个窗体，并在启动窗体时在构造函数中绘制它。注意，这并不是在屏幕上绘图的最佳方式或正确方式，因为这个示例并不能在启动后按照需要重新绘制窗体。但利用这个示例，我们不必做太多的工作，就可以说明绘图的许多问题。

对于这个示例，启动 Visual Studio 2010，创建一个 Windows From Application 项目。首先把窗体的背景色设置为白色。把这行代码放在 `InitializeComponent()` 方法的后面，这样 Visual Studio 2010 就会识别该行命令，并能够改变窗体的设计视图的外观。单击 `Form1.cs` 文件旁边的箭头图标(这会展开与 `Form1.cs` 文件相关的文件层次结构)，就可以看到 `Form1.Designer.cs` 文件。正是在这个文件中发现了 `InitializeComponent()` 方法。本来也可以使用设计视图设置背景色，但这会导致自动添加相同的代码：



可从  
wrox.com  
下载源代码

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
}
```

代码段 DrawingShapes.sln

接着，给 `Form1` 构造函数添加代码。使用窗体的 `CreateGraphics()` 方法创建一个 `Graphics` 对象，这个对象包含绘图时需要使用的 Windows 设备上下文。创建的设备上下文与显示设备相关，也与这个窗口相关。



可从  
wrox.com  
下载源代码

```
public Form1()
{
    InitializeComponent();

    Graphics dc = CreateGraphics();
    Show();
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

代码段 DrawingShapes.sln

然后调用 `Show()` 方法显示窗口。之所以让该窗口立即显示，是因为在该窗口显示出来之前，我们不能做任何工作——没有可供绘图的地方。

最后，显示一个矩形，其坐标是(0,0)，宽度和高度是 50，再绘制一个椭圆，其坐标是(0, 50)，

宽度是 80，高度是 50。注意坐标(x,y)表示从窗口的工作区左上角开始向右的 x 个像素，向下的 y 个像素——这些坐标从要显示的图形的左上角开始计算。

我们使用的 `DrawRectangle()`和 `DrawEllipse()`重载方法分别带 5 个参数。第一个参数是 `System.Drawing.Pen` 类的实例。`Pen` 是许多帮助绘图的辅助对象中的一个，它包含如何绘制线条的信息。第一个 `Pen` 表示线条应是蓝色的，其宽度为 3 个像素；第二个 `Pen` 表示线条应是红色的，其宽度为两个像素。后面的 4 个参数是坐标和大小。对于矩形，它们分别表示矩形的左上角坐标(x,y)、其宽度和高度。对于椭圆，这些数值的含义相同，但它们是指椭圆假想的外接矩形，而不是椭圆本身。运行代码，会得到如图 48-1 所示的图形。当然，本书不是彩书，所以看不到颜色。

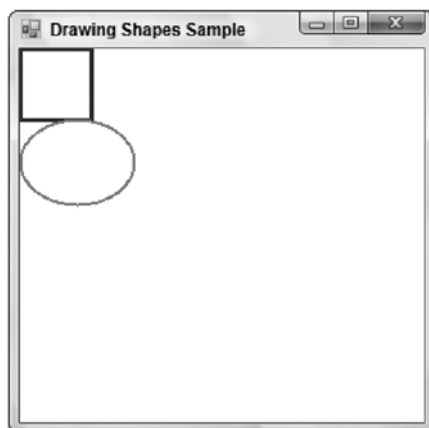


图 48-1

图 48-1 说明了两个问题。首先，用户可以很清楚地看到在窗口中工作区的位置。它是一块白色的区域——该区域受到 `BackColor` 属性设置的影响。还要注意，矩形放在该区域的一角，因为它指定了坐标(0,0)。其次，注意椭圆的顶部与矩形有轻度的重叠，这与代码中给出的坐标有点不同。这是因为 Windows 在重叠的区域上放置了矩形和椭圆的边框。在默认情况下，Windows 会试图把图形边框所在的线条放在中心位置——但这并不是总能做到的，因为线条是以像素为单位来绘制的，但每个图形的边框理论上通常位于两个像素之间。结果，1 个像素宽的线条就会正好位于图形的顶边和左边的里面，而在右边和底边的外面。这样，从严格意义上讲，相邻图形的边框就会有一个像素的重叠。我们指定的线条宽度比较大，因此重叠区域也会比较大。设置 `Pen.Alignment` 属性(详见 SDK 文档说明)，就可以改变默认的操作方式，但这里使用默认的操作方式就足够了。

但如果运行这个示例，就会注意到窗体的行为有点奇怪。如果把它放在那里，或者用鼠标在屏幕移动该窗体，它就工作正常。但如果最小化该窗体，再还原它，绘制好的图形就不见了。如果在该窗体上拖动另一个窗口，使之只遮挡一部分图形，再拖动该窗口远离这个窗体，临时被挡住的部分就消失了，只剩下一半椭圆或矩形了！

这是怎么回事？其原因是，如果窗口的一部分被隐藏了，Windows 通常会立即丢弃与其中显示的内容相关的所有信息。这是必需的，否则存储屏幕数据的内存量就会是个天文数字。一般的计算机在运行时，视频卡设置为显示 1024×768 像素、24 位彩色模式，这表示屏幕上的每个像素占据 3 个字节，于是显示整个屏幕就需要 2.25MB(本章后面会说明 24 位颜色的含义)。但是，用户常常让任务栏上有 10 个或 20 个最小化的窗口。下面考虑一种最糟糕的情况：20 个窗口，每个窗口如果没

有最小化，它就占据整个屏幕。如果 Windows 存储了这些窗口包含的可视化信息，当用户还原它们时，它们就会有 45MB。目前，比较好的图形卡有 512MB 的内存，可以应付这种情况，但在几年前图形卡有 256MB 的内存就不错了。多余的部分需要存储在计算机的主内存中。许多人仍在使用旧计算机，一些人甚至还在使用 256MB 的图形卡。很显然，Windows 不可能这样管理用户界面。

在窗口的任何某一部分消失时，隐藏的那些像素也就丢失了。因为 Windows 释放了保存这些像素的内存。但要注意，窗口的一部分被隐藏了，当它检测到窗口不再被隐藏时，就请求拥有该窗口的应用程序重新绘制其内容。这条规则有一些例外——通常窗口的一小部分被挡住的时间比较短(例如，从主菜单中选择一个菜单项，该菜单项向下拉出，临时挡住了下面的部分)。但一般情况下，如果窗口的一部分被挡住，应用程序就需要在以后重新绘制它。

这就是示例应用程序的问题的根源。我们把绘图代码放在 Form1 的构造函数中，这些代码仅在应用程序启动时调用一次，不能在以后需要时再次调用该构造函数，重新绘制图形。

在使用 Windows 窗体的服务器控件时，不需要知道如何完成上述任务，这是因为标准控件非常专业，能在 Windows 需要时重新绘制它们自己。这是编写控件时不需要担心实际绘图过程的原因之一。如果要应用程序在屏幕上绘图，还需要在 Windows 要求重新绘制窗口的全部或部分时，确保应用程序会正确响应。下一节将修改这个示例，确保应用程序的正确响应。

### 48.1.3 使用 OnPaint()方法绘制图形

上面的解释让您觉得绘制自己的用户界面是比较复杂的，实际上并非如此。让应用程序在需要时绘制自身是非常简单的。

Windows 会触发 Paint 事件通知应用程序完成一些重新绘制的要求。有趣的是，Form 类已经实现了这个事件的处理程序，因此不需要再添加处理程序了。Paint 事件的 Form1 处理程序处理虚方法 OnPaint()的调用，并给它传递一个参数 PaintEventArgs，这表示，我们只需重写 OnPaint()方法执行绘图操作。

我们选择重写 OnPaint()方法，也可以为 Paint 事件添加自己的事件处理程序(如 Form1\_Paint()方法)来得到相同的结果，其方式与为任何其他 Windows 窗体事件添加处理程序一样。后一种方法更方便一些，因为可以通过 Visual Studio 2010 的属性窗口添加新的事件处理程序，不必输入某些代码。但是我们采用重写 OnPaint()方法的方式要略为灵活一些，因为这样可以控制何时调用基类窗口进行处理，并允许避免把控件的处理程序关联到它自己的事件上。

下面新建一个 Windows 应用程序 DrawShapes 来完成这个操作。与以前一样，使用属性窗口把背景色设置为白色，再把窗体的文本改为 DrawShapes Sample，接着在 Form1 类的自动生成代码中添加如下代码：



可从  
wrox.com  
下载源代码

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0,0,50,50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

代码段 DrawingShapes.sln

注意，OnPaint()方法声明为 protected。因为 OnPaint()方法一般在类的内部使用，所以类外部的其他代码不必知道存在 OnPaint()方法。

PaintEventArgs 是派生自 EventArgs 类的一个类，一般用于传递有关事件的信息。PaintEventArgs 类有另外两个属性，其中比较重要的是 Graphics 实例，它们主要用于优化窗口中需要绘制的部分。这样就不必调用 CreateGraphics()方法在 OnPaint()方法中获取设备上下文了——用户总是可以得到设备上下文。后面将介绍其他属性。这个属性包含窗口的哪些部分需要重新绘制的详细信息。

在 OnPaint()方法的实现代码中，首先从 PaintEventArgs 类中引用 Graphics 对象，再像以前那样绘制图形。最后调用基类的 OnPaint()方法，这一步非常重要。我们重写了 OnPaint()方法，完成了绘图工作，但 Windows 在绘图过程中可能会执行一些它自己的辅助工作。这些工作都在.NET 基类的 OnPaint()方法中完成。



对于这个示例，删除 base.OnPaint()的调用似乎并没有任何影响，但不要试图删除这个调用。这样有可能阻止 Windows 正确执行任务，结果是无法预料的。

在应用程序第一次启动和窗口第一次显示时，也调用了 OnPaint()方法，所以不需要在构造函数中复制绘图代码。

运行这段代码，得到的结果将与前面的示例的结果相同，但现在，当最小化窗口或隐藏它的一部分时，应用程序会正确执行。

48.1.4 使用剪切区域

上一节的 DrawShapes 示例说明了在窗口中绘图的主要规则，但它并不是很高效。原因是它试图绘制窗口中的所有内容，而没有考虑需要绘制多少内容。如图 48-2 所示，运行 DrawShapes 示例，当该示例在屏幕上绘图时，打开另一个窗口，把它移动到 DrawShapes 窗体上，使之遮挡一部分窗体。

但移动重叠的窗口时，DrawShapes 窗口会再次全部显示出来，Windows 通常会给窗体发送一个 Paint 事件，要求它重新绘制本身。因为矩形和椭圆都位于工作区的左上角，所以在任何时候它们都是可见的，在本例中不需要重新绘制这部分，而只需要重新绘制白色背景区域。但是，Windows 并不知道这一点，它认为应触发 Paint 事件，调用 OnPaint()方法的实现代码。OnPaint()方法不必重新绘制矩形和椭圆。

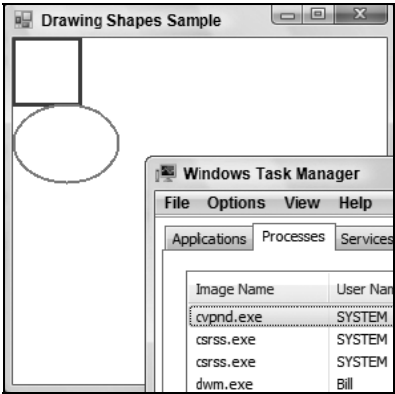


图 48-2

在本例中，没有重新绘制图形。原因是我们使用了设备上下文。Windows 将利用重新绘制哪些区域所需要的信息预先初始化设备上下文。在 GDI 中，被标记出来的重绘区域称为无效区域，但在 GDI+ 中，该术语通常改为剪切区域。设备上下文能识别这个区域。于是，它截取在这个区域外部的绘图操作，且不把相关的绘图命令传送给图形卡。这听起来不错，但仍有一个潜在的性能损失。在确定是在无效区域外部绘图前，我们不知道必须进行多少设备环境处理。在某些情况下，要处理的任务比较多，因为计算哪些像素需要改变为什么颜色，将会频繁占用处理器(好的图形卡会提供硬件加速，对此有一定的帮助)。

其底线是让 Graphics 实例完成在无效区域外部的绘图工作，这肯定会浪费处理器的时间，减慢应用程序的运行。在设计优良的应用程序中，代码将执行一些检查，以查看需要进行哪些绘图工作，然后调用相关的 Graphics 实例的方法。本节将编写一个新示例 DrawShapesWithClipping，方法是修改 DisplayShapes 示例，只完成需要的重新绘制工作。在 OnPaint() 方法的代码中，进行一个简单的测试，看看无效区域是否与需要绘制的区域重叠，如果是，就调用绘图方法。

首先，需要获得剪切区域的信息。这需要使用 PaintEventArgs 类的另一个属性——ClipRectangle，它包含要重绘区域的坐标，并封装在一个结构的实例 System.Drawing.Rectangle 中。Rectangle 是一个相当简单的结构，它包含 4 个属性：Top、Bottom、Left 和 Right。它们分别包含矩形的上下边的垂直坐标、左右边的水平坐标。

接着，需要确定进行什么测试，以决定是否进行绘制。这里进行一个简单的测试。注意，在绘图过程中，矩形和椭圆完全包含在从(0,0)点到(80,130)点的矩形工作区中，实际上，(82,132)点就已经在安全区域中了，因为线条大约偏离这个区域的外侧一个像素。所以我们要看看剪切区域的左上角是否在这个矩形区域内。如果是，就重新绘制；如果不是，就不必麻烦了。

下面是代码：



```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 132 && e.ClipRectangle.Left < 82)
    {
        Pen bluePen = new Pen(Color.Blue, 3);
        dc.DrawRectangle(bluePen, 0,0,50,50);
        Pen redPen = new Pen(Color.Red, 2);
        dc.DrawEllipse(redPen, 0, 50, 80, 60);
    }
}
```

代码段 DrawingShapes.sln

注意，显示的结果与以前显示的结果完全相同——但这次进行了早期测试，确定了哪些区域不需要绘制，提高了性能。还要注意这个示例关于是否进行绘图的测试是非常粗略的。还可以进行更精细的测试，分别确定矩形或者椭圆是否要重新绘制。这里有一个平衡。可以在 OnPaint() 方法进行更复杂的测试，以提高性能，但也可以使 OnPaint() 方法的代码更复杂一些。进行一些测试总是值得的，因为编写一些代码，可以更多地解除 Graphics 实例之外的绘制内容，Graphics 实例只是盲目地执行绘图命令。



## 48.2 测量坐标和区域

在上一个示例中，我们遇到了基本结构 `Rectangle`，它用于表示矩形的坐标。GDI+实际上使用几个类似的结构来表示坐标或区域。表 48-1 列出了几个结构，它们都是在 `System.Drawing` 名称空间中定义的，如表 48-2 所示。

表 48-2

结 构	主要的公共属性
<code>Point</code> 和 <code>PointF</code>	<code>X</code> 、 <code>Y</code>
<code>Size</code> 和 <code>SizeF</code>	<code>Width</code> 、 <code>Height</code>
<code>Rectangle</code> 和 <code>RectangleF</code>	<code>Left</code> 、 <code>Right</code> 、 <code>Top</code> 、 <code>Bottom</code> 、 <code>Width</code> 、 <code>Height</code> 、 <code>X</code> 、 <code>Y</code> 、 <code>Location</code> 、 <code>Size</code>

注意，其中的许多对象都有许多其他属性、方法或运算符重载，这里没有列出来。本节只讨论某些最重要的成员。

### 48.2.1 `Point` 和 `PointF` 结构

从概念上讲，`Point` 在这些结构中最简单的。在数学上，它等价于一个二维矢量。它包含两个公共整型属性，它表示与某个特定位置的水平和垂直距离(在屏幕上)，如图 48-3 所示。

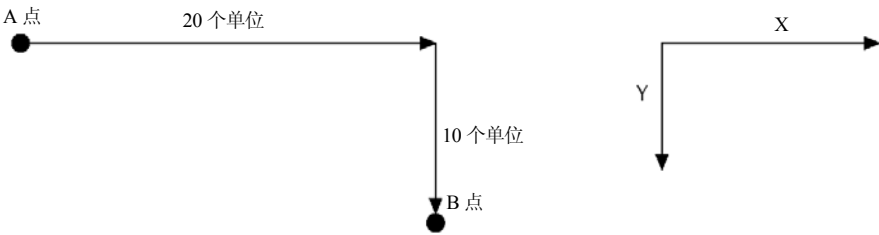


图 48-3

为了从 A 点到 B 点，需要水平移动 20 个单位，并向下垂直移动 10 个单位，在图 48-3 中标为 `x` 和 `y`，这就是它们的一般含义。下面的 `Point` 结构表示该线条：

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

`X` 和 `Y` 都是读写属性，因此可以在 `Point` 中设置这些值，例如：

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

注意，按照惯例，水平坐标和垂直坐标表示为 `x` 和 `y`(小写)，但相应的 `Point` 属性是 `X` 和 `Y`(大写)，因为在 C# 中，公共属性的一般约定是名称以一个大写字母开头。

`PointF` 与 `Point` 完全相同，但 `X` 和 `Y` 属性的类型是 `float`，而不是 `int`。`PointF` 用于坐标不是整数

值的情况。已经为这些结构定义了数据类型强制转换,这样就可以把 `Point` 隐式转换为 `PointF`(注意,因为 `Point` 和 `PointF` 是结构,这种强制转换实际上涉及数据的复制)。但没有相应的逆过程,要把 `PointF` 转换为 `Point`,必须复制对应的值,或使用下面的 3 个转换方法 `Round()`、`Truncate()`和 `Ceiling()`中的一个:

```
PointF abFloat = new PointF(20.5F, 10.9F);
// converting to Point
Point ab = new Point();
ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);
// but conversion back to PointF is implicit
PointF abFloat2 = ab;
```

下面看看测量单位。在默认情况下, GDI+把单位看作是屏幕(或打印机,无论图形设备是什么,都可以这样认为)上的像素。这就是 `Graphics` 对象的方法把它们接收到的坐标看作其参数的方式。例如, `new Point(20,10)`点表示在屏幕上水平向右移动 20 个像素,垂直向下移动 10 个像素。通常这些像素从窗口客户区域的左上角开始测量,如上面的示例所示。但是,情况并不总是如此。例如,在某些情况下,需要以整个窗口的左上角(包括其边框)为原点来绘图,甚至以屏幕的左上角为原点。但在大多数情况下,除非文档特别说明,否则都可以假定像素值是相对于客户区域的左上角。

在分析了滚动后,本章将在讨论 3 种坐标系统(世界、页面和设备坐标)时介绍这个主题。

## 48.2.2 Size 和 SizeF 结构

与 `Point` 和 `PointF` 一样, `Size` 也有两个变体。`Size` 结构用于 `int` 类型, `SizeF` 用于 `float` 类型。除此之外, `Size` 和 `SizeF` 是完全相同的。下面主要讨论 `Size` 结构。

在许多情况下, `Size` 结构与 `Point` 结构是相同的,它也有两个整型属性,分别表示水平距离和垂直距离。主要区别是这两个属性的名称不是 `X` 和 `Y`,而是 `Width` 和 `Height`。图 48-3 可以表示为:

```
Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width, ab.Height);
```

严格地讲, `Size` 在数学上与 `Point` 表示的含义相同;但在概念上它的使用方式略有不同。`Point` 用于说明实体在什么地方,而 `Size` 用于说明实体有多大。但是, `Size` 和 `Point` 是紧密相关的,目前甚至支持它们之间的相互转换:

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

例如,考虑前面绘制的矩形,其左上角的坐标是(0,0),大小是(50,50)。这个矩形的大小是(50,50),可以用一个 `Size` 实例来表示。其右下角的坐标也是(50,50),但它由一个 `Point` 实例来表示。要理解这个区别,假定在另一个位置绘制该矩形,其左上角的坐标是(10,10):

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

现在其右下角的坐标是(60,60)，但大小不变，仍是(50,50)。

因为 Point 和 Size 结构的加法运算符都已经重载了，所以可以把一个 Size 加到 Point 结构上，得到另一个 Point 结构：



可从  
wrox.com  
下载源代码

```
static void Main(string[] args)
{
    Point topLeft = new Point(10,10);
    Size rectangleSize = new Size(50,50);
    Point bottomRight = topLeft + rectangleSize;
    Console.WriteLine("topLeft = " + topLeft);
    Console.WriteLine("bottomRight = " + bottomRight);
    Console.WriteLine("Size = " + rectangleSize);
}
```

代码段 PointsAndSizes.sln

把这段代码作为控制台应用程序 PointAndSizes 来运行，会得到如图 48-4 所示的结果。

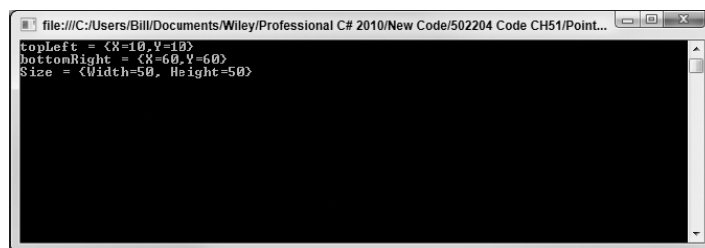


图 48-4

注意，这个结果也说明了 Point 和 Size 的 ToString()方法已被重写，并以{X,Y}格式显示该值。

也可以从一个 Point 结构中减去某个 Size 结构，得到另一个 Point 结构，还可以把两个 Size 加在一起，得到另一个 Size。但不能把一个 Point 结构加到另一个 Point 结构上。Microsoft 认为 Point 结构相加在概念上没有意义，所以不支持加法(+)运算符的任何重载版本进行这样的操作。

还可以在 Point 和 Size 结构之间进行显式的数据类型强制转换：

```
Point topLeft = new Point(10,10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;
```

在进行这样的数据类型强制转换时，s1.Width 被赋予 topLeft.X 的值，s1.Height 被赋予 topLeft.Y 的值。因此 s1 包含(10,10)。p1 最终的值与 topLeft 的值相同。

### 48.2.3 Rectangle 和 RectangleF 结构

这两个结构表示一个矩形区域(通常在屏幕上)。与 Point 和 Size 一样，这里只介绍 Rectangle 结构，RectangleF 与 Rectangle 基本相同，但它的属性类型是 float，而 Rectangle 的属性类型是 int。

Rectangle 可以看作由一个 Point 和一个 Size 组成，其中 Point 表示矩形的左上角，Size 表示其大小。它的一个构造函数把 Point 和 Size 结构作为其参数。下面重写前面的 DrawShapes 示例的代码，绘制一个矩形：

```
Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
```

```
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);
```

这段代码也使用 Graphics.DrawRectangle()方法的另一个重载版本，它的参数是 Pen 和 Rectangle 结构。

通过按顺序提供矩形的左上角水平和左上角垂直坐标，宽度和高度(它们都是数字)，可以构造一个 Rectangle:

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50)
```

Rectangle 包含几个读写属性，如表 48-3 所示，可以用不同的属性组合来设置或提取它的维度。

表 48-3

属 性	说 明
int Left	左边界的 x 坐标
int Right	右边界的 x 坐标
int Top	顶边的 y 坐标
int Bottom	底边的 y 坐标
int X	与 Left 相同
int Y	与 Top 相同
int Width	矩形的宽度
int Height	矩形的高度
Point Location	左上角
Size Size	矩形的大小

注意，这些属性并非都是独立的。例如，设置 Width 会影响 Right 的值。

48.2.4 Region

Region 表示屏幕上一个包含复杂图形的区域。例如，图 48-5 中的阴影区域就可以用 Region 表示。

可以想象，初始化 Region 实例的过程相当复杂。从广义上看，可以指定哪些简单的图形组成这个区域，或指定绘制这个区域边界的路径。如果需要处理这样的区域，就应掌握 SDK 文档中的 Region 类。



图 48-5

## 48.3 调试须知

下面准备进行一些更高级的绘图工作。但首先介绍几个调试问题。如果在本章的示例中设置了断点，就会注意到调试图形例程不像调试程序的其他部分那样简单。这是因为进入和退出调试程序常常会把 **Paint** 消息传递给应用程序。结果是在 **OnPaint()**重载方法上设置的断点会让应用程序一遍又一遍地绘制它本身，这样应用程序基本上就不能完成任何工作。

这是很典型的一种情形。要明白为什么应用程序没有正确显示，可以在 **OnPaint()**事件中设置断点。应用程序会像期望的那样，遇到断点后，就会进入调试程序，此时在前台会显示开发环境 MDI 窗口。如果把开发环境设置为全屏显示，以便更易于观察所有的调试信息，就会完全隐藏目前正在调试的应用程序。

接着，检查某些变量的值，希望找出某些有用的信息。然后按 F5 键，告诉应用程序继续执行，完成某些处理后，看看应用程序在显示其他内容时会发生什么。但首先发生的是应用程序显示在前台中，Windows 快速检测到窗体再次可见，并提示给它发送了一个 **Paint** 事件。当然这表示程序遇到了断点。如果这就是我们想要的结果，那就很好。但更常见的是，我们希望以后在应用程序绘制了某些有趣的内容后再遇到断点，例如，在选择某些菜单项以读取一个文件或者以其他方式改变显示的内容之后，再遇到断点。这听起来就是我们需要的结果。我们或者根本没有在 **OnPaint()**中设置断点，或者应用程序不会显示它在最初的启动窗口中显示的边界点之外的其他内容。

有一种方式可以解决这个问题。

如果有足够大的屏幕，最简单的方式就是平铺开发环境窗口，而不是把它设置为最大化，使之远离应用程序窗口，这样首先应用程序就不会被挡住了。但在大多数情况下，这并不是一个有效的解决方案，因为这样会使开发环境窗口过小也可以使用第二个监视器。另一个解决方案使用相同的规则，即在调试时把应用程序声明为最上面的应用程序。方法是在 **Form** 类中设置属性 **TopMost**，这很容易在 **InitializeComponent()**方法中完成：

```
private void InitializeComponent()
{
    this.TopMost = true;
```

也可以在 Visual Studio 2010 的属性窗口中设置这个属性。

窗口设置为 **TopMost** 表示应用程序不会被其他窗口挡住(除了其他放在最上面的窗口)。它仍然总是放在其他窗口的上面，甚至在另一个应用程序得到焦点时，也是这样。这是任务管理器的执行方式。

即使利用这个技巧也必须小心，因为我们不能确定 Windows 何时会决定因为某种原因触发 **Paint** 事件。如果在某些特殊的情况下，在 **OnPaint()**方法出了问题(例如，应用程序在选择某个特定菜单项后绘图，但此时出了问题)，那么最好的方式是在 **OnPaint()**方法中添加一些虚拟代码，该方法测试某些条件，这些条件只在特殊的情况下才为 **true**。然后在 **if** 块中设置断点，如下所示：

```
protected override void OnPaint( PaintEventArgs e )
{
    // Condition() evaluates to true when we want to break
    if (Condition())
    {
```

```
int ii = 0;    // <-SET BREAKPOINT HERE!!!
}
```

这是设置条件断点的一种简捷方式。

## 48.4 绘制可滚动的窗口

前面的 `DrawShapes` 示例运行良好，因为需要绘制的内容正好适合最初的窗口大小。本节介绍如果绘制的内容不适合窗口的大小，需要做哪些工作。

对于本示例，下面扩展 `DrawShapes` 示例，以解释滚动的概念。为了使该示例更符合实际，首先创建一个 `BigShapes` 示例，其中将矩形和椭圆画大一些。此时将使用 `Point`、`Size` 和 `Rectangle` 结构定义绘图区域，说明如何使用它们。进行了这样的修改后，`Form1` 类的相关部分如下所示：

```
// member fields
private readonly Point rectangleTopLeft = new Point(0, 0);
private readonly Size rectangleSize = new Size(200,200);
private readonly Point ellipseTopLeft = new Point(50, 200);
private readonly Size ellipseSize = new Size(200, 150);
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
    {
        Rectangle rectangleArea =
            new Rectangle (rectangleTopLeft, rectangleSize);
        Rectangle ellipseArea =
            new Rectangle (ellipseTopLeft, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

注意，这里还把 `Pen`、`Size` 和 `Point` 对象变成成员字段——这比每次需要绘图时都新建一个 `Pen` 的效率更高。

运行这个示例，得到如图 48-6 所示的结果。

很快这里有一个问题，图形在 300×300 像素的绘图区域中放不下。

一般情况下，如果文档太大，不能完全显示，应用程序就会添加滚动条，以便用户滚动窗口，查看其中选中的部分。这是另一个区域，在该区域中如果使用标准控件构建 `Windows` 窗体，就让 .NET 运行库和基类来处理所有操作。如果在窗体中有各种控件，那么 `Form` 实例一般知道这些控件在哪里，并且如果其窗口比较小，`Form` 实例就知道需要添加滚动条。`Form` 实例还会自动添加滚动条，不仅如此，它还可以正确地绘制用户滚动到的部分屏幕。此时，用户不需要在代码中做什么工作。但在本章中，我们要在屏幕上绘制图形，所以要帮助 `Form` 实例确定何时能滚动。

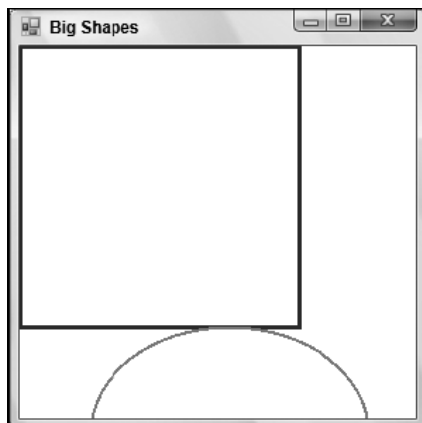


图 48-6

添加滚动条实际上很简单。Form 仍会处理所有的操作——因为它不知道绘图区域有多大。在上面的 BigShapes 示例中没有滚动条的原因是，Windows 不知道它们需要滚动条。我们需要确定的是，矩形的大小从文档的左上角(或者是在进行任何滚动前的客户区域的左上角)开始向右下角延伸，其大小应足以包含整个文档。本章把这个区域称为文档区域，如图 48-7 所示，本例的文档区域应是 250×350 个像素。

使用相关的属性 Form.AutoScrollMinSize 即可轻松确定文档的大小。因此给 InitializeComponent() 方法或 Form1 的构造函数添加下述代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
    this.AutoScrollMinSize = new Size(250, 350);
}
```

另外，AutoScrollMinSize 属性还可以用 Visual Studio 2010 的属性窗口设置。注意要访问 Size 类，需要添加下面的 using 语句：

```
using System.Drawing;
```

在应用程序启动时设置最小尺寸，并保持不变，在这个特定的应用程序中是必要的，因为我们知道屏幕区域一般有多大。在运行该应用程序时，这个“文档”不会改变大小。但要记住，如果应用程序执行显示文件的内容这样的操作，或者执行改变屏幕的哪个区域的操作，就需要在其他时间设置这个属性(此时，必须手动调整代码，Visual Studio 2010 的属性窗口只能在构建窗体时设置属性的初始值)。

设置 MinScrollSize 属性只是一个开始，仅有它是不够的。图 48-8 显示了示例应用程序目前的外观——开始时，屏幕会正确地显示图形。

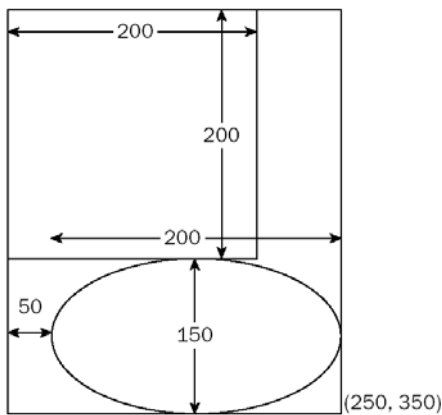


图 48-7

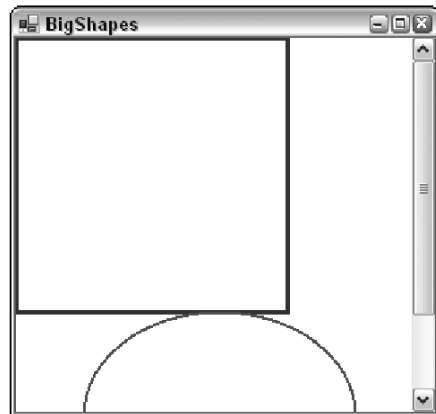


图 48-8

注意，不仅窗体正确地设置了滚动条，而且它的大小也正确设置了，以指定文档当前显示的比例。可以试着在运行示例时重新设置窗口的大小，这样就会发现滚动条会正确地响应，甚至如果使窗口变得足够大，不再需要滚动条时，滚动条就会消失。

但是，如果使用一个滚动条，并向下滚动它，会发生什么情况呢？结果如图 48-9 所示。显然，出现了错误！

出错的原因是我们没有在 `OnPaint()` 重写方法的代码中考虑滚动条的位置。如果最小化窗口，再还原它，从而重新绘制窗口自身，就可以很清楚地看出这一点。结果如图 48-10 所示。

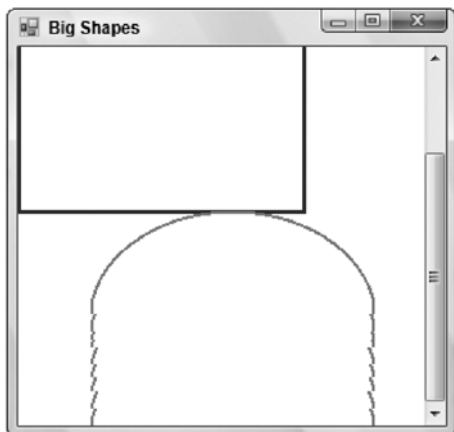


图 48-9

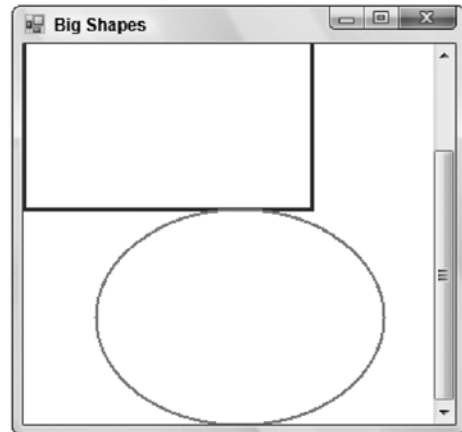


图 48-10

图形像以前一样进行了绘制，矩形的左上角嵌套在客户区域的左上角，就好像根本没有移动过滚动条一样。

在介绍如何更正这个问题前，先介绍一下在这些屏幕截图上发生了什么。

首先从 `BigShapes` 示例开始，如图 48-8 所示。在这个例子中，整个窗口刚刚重新进行了绘制。看看前面的代码，该代码的作用是使图形实例用左上角坐标(0,0)(相对于窗口的客户区域的左上角)绘制一个矩形——它是已经绘制过的。问题是，图形实例在默认情况下把坐标解释为是相对于客户



窗口的，它不能识别滚动条。代码还没有尝试为滚动条的位置调整坐标。椭圆也是这样。

下面处理图 48-9 中的屏幕截图。在向下滚动后，注意窗口上半部分显示正确，这是因为它是在应用程序第一次启动时绘制的。在滚动窗口时，Windows 没有要求应用程序重新绘制已经显示在屏幕中的内容。Windows 足够智能可以判断屏幕上目前显示的哪些内容可以平滑地移动，以匹配滚动条所处的位置。这是一个非常高效的过程，因为它也能使用某些硬件来加速完成。在这个屏幕截图中，有错误的是窗口下部的 1/3 部分。在应用程序第一次显示时，没有绘制这部分窗口，因为在滚动窗口前，这部分在客户区域的外部。这表示 Windows 要求 BigShapes 应用程序绘制这个区域。它触发 Paint 事件，把这个区域作为剪切的矩形。这也是 OnPaint()重载方法完成的任务。

该问题的另一种表达方式是我们将坐标表示为相对于文档开头的左上角——需要转换它们，用相对于客户区域的左上角的坐标表示它们。图 48-11 说明了这一点。

为了使该图更清晰，我们向下向右扩展了该文档，超出了屏幕的边界，但这不会改变我们的推理。我们还假定其上还有一个水平滚动条和一个垂直滚动条。

在图 48-11 中，细线条的矩形标记了屏幕区域的边框和整个文档的边框。粗线条标记了要绘制的矩形和椭圆。P 标记要绘制的某个任意点，这个点在后面会作为一个示例。在调用绘图方法时，提供图形实例和从 B 点到 P 点的矢量，这个矢量表示为一个 Point 实例。我们实际上需要给出从 A 点到 P 点的矢量。

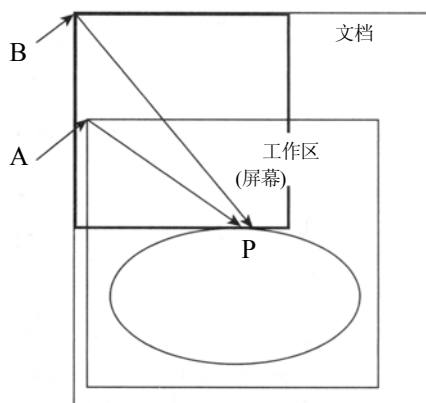


图 48-11

问题是，我们不知道从 A 点到 P 点的矢量。而知道从 B 点到 P 点的矢量，这就是 P 点相对于文档左上角的坐标——我们要在文档的 P 点绘图的位置。还知道从 B 点到 A 点的矢量就是滚动的距离，它存储在 Form 类的一个属性 AutoScrollPosition 中。但是不知道从 A 点到 P 点的矢量。

要解决这个问题，只需进行矢量相减即可。例如，要从 B 点到 P 点，可以水平向右移动 150 个像素，再垂直向下移动 200 个像素。而要从 B 点到 A 点，就需要水平向右移动 10 个像素，再垂直向下移动 57 个像素。这表示，要从 A 点到 P 点，需要水平向右移动 140 个像素( $=150-10$ )，再垂直向下移动 143 个像素( $=200-57$ )。为了使之更简单，Graphics 类实际上实现了一个方法来进行这些计算，这个方法是 TranslateTransform()，我们给它传递水平坐标和垂直坐标，表示工作区的左上角相对于文档的左上角(AutoScrollPosition 属性，它是图 48-11 中从 B 点到 A 点的矢量)。然后 Graphics 设备考虑客户区域相对于文档区域的位置，计算出它所有的坐标。

如果把上述解释转化为代码，通常所需做的就是下面这行代码添加到绘图代码中：

```
dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

但在本示例中，它有点复杂，因为我们还要查看剪切区域，看看是否需要进行绘制工作。这个测试需要调整，把滚动的位置也考虑在内。完成后，该示例的整个绘图代码如下所示：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Size scrollOffset = new Size(this.AutoScrollPosition);
    if (e.ClipRectangle.Top+scrollOffset.Width < 350 ||
        e.ClipRectangle.Left+scrollOffset.Height < 250)
    {
        Rectangle rectangleArea = new Rectangle
            (rectangleTopLeft+scrollOffset, rectangleSize);
        Rectangle ellipseArea = new Rectangle
            (ellipseTopLeft+scrollOffset, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

现在，滚动代码工作正常，最后得到正确地滚动的屏幕截图，如图 48-12 所示。

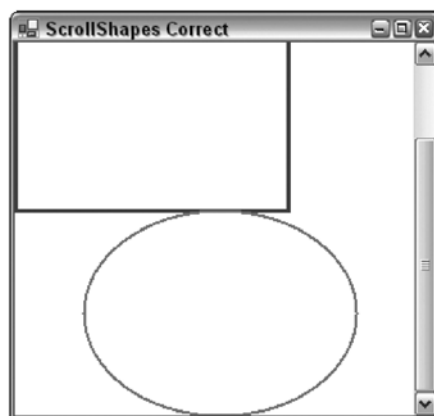


图 48-12

## 48.5 世界、页面和设备坐标

测量相对于文档左上角的位置和测量相对于屏幕(桌面)左上角的位置之间的区别如此重要，因此 GDI+ 对于这些坐标系统有专门的名称：

- **世界坐标(World Coordinate)**——要测量的点距离文档左上角的位置(以像素为单位)。
- **页面坐标(Page Coordinate)**——要测量的点距离客户区域左上角的位置(以像素为单位)。



熟悉 GDI 的开发人员要注意，世界坐标对应于 GDI 中的逻辑坐标。页面坐标对应于设备坐标。还要注意，编写逻辑坐标和设备坐标之间的转换代码在 GDI+中有了变化。在 GDI 中，使用 Windows API 函数 LPtoDP()和 DPtoLP()通过设备上下文进行转换，而在 GDI+中，正是 Control 类来维护转换过程中所需要的信息，Form 和各种 Windows 窗体控件设备派生自 Control 类。

GDI+还有第 3 种坐标，即设备坐标(Device Coordinate)。设备坐标类似于页面坐标，但其测量单位不是像素，而使用用户调用 Graphics.PageUnit 属性指定的某些其他单位。它可以使用的单位除了默认的像素外，还包括英寸和毫米。本章虽然没有使用 PageUnit 属性，但它可用作获取设备的不同像素密度的方式。例如，在大多数显示器上，100 个像素大约是 1 英寸。但是，激光打印机可以达到 1 200 dpi(点/英寸)——这表示 100 个像素宽的图形在该激光打印机上打印时会比较小。把单位设置为英寸，指定图形为 1 英寸宽，就可以确保图形在不同的设备上有相同的大小。这可通过如下代码来说明：

```
Graphics dc = this.CreateGraphics();
dc.PageUnit = GraphicsUnit.Inch;
```

通过 GraphicsUnit 枚举可用的值如下：

- **Display**——指定显示器的测量单位
- **Document**——把文档单位(1/300 英寸)定义为测量单位
- **Inch**——把英寸定义为测量单位
- **Millimeter**——把毫米定义为测量单位
- **Pixel**——把像素定义为测量单位
- **Point**——把打印机的点数(1/72 英寸)定义为测量单位
- **World**——把世界坐标系定义为测量单位

## 48.6 颜色

本节介绍如何在绘制图形时指定使用的颜色。

在 GDI+中，颜色用 System.Drawing.Color 结构的实例来表示。一般情况下，初始化这个结构后，就不能使用对应的 Color 实例对该结构进行操作了——只能把它传递给其他需要 Color 的任何其他主调方法。前面我们遇到过这种结构，在前面的每个示例中都设置了窗口的客户区域的背景色，还设置了要显示的各种图形的颜色。Form.BackColor 属性返回一个 Color 实例。本节将详细介绍这个结构，特别是要介绍构建 Color 的几种不同方式。

### 48.6.1 RGB 值

显示器可以显示的颜色总数非常大——超过 1600 万。其确切的数字是 2 的 24 次方，即 16 777 216。显然，我们需要对这些颜色进行索引，才能指定在任何给定的像素上要显示什么颜色。

对颜色进行索引的最常见方式是把它们分为红、绿、蓝分量，这种思想基于下述原理：人眼可

以分辨的任何颜色都由一定量的红色光、绿色光和蓝色光组成。这些光称为分量(component)。实际上,如果每种分量的光分为 256 种不同的强度,它们提供了足够平滑的过渡,可以把人眼能观察到的图像显示为高质量的照片。因此,指定颜色时,可以给出这些分量的量,其值在 0~255 之间,其中 0 表示没有这种分量,255 表示这种分量的光达到最大的强度。

这给出了向 GDI+说明颜色的第一种方式。可以调用静态函数 `Color.FromArgb()`指定该颜色的红、绿、蓝值。Microsoft 没有为此提供构造函数,原因是除了一般的 RGB 分量外,还有其他方式来表示颜色。因此,Microsoft 认为给定义的任何构造函数传递参数会引起误解:

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor = Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

3 个参数分别是红、绿、蓝值。这个函数有许多重载方法,其中一些也允许指定 Alpha 混合值(这是方法名 `FromArgb()`中的 A)。Alpha 混合超出了本章的范围,但把它与屏幕上已有的任何颜色混合起来,可以绘出半透明的颜色。这可以得到一些很漂亮的效果,常常用于游戏中。

## 48.6.2 命名颜色

使用 `FromArgb()`方法构造颜色是一种非常灵活的技巧,因为它表示我们可以指定人眼能辨识出的任何颜色。但是,如果要得到一种简单、标准、众所周知的纯色,如红色或蓝色,命名想要的颜色就比较简单。因此 Microsoft 还在 `Color` 中提供了许多静态属性,每个属性都返回一种命名颜色。在下面的示例中,把窗口的背景色设置为白色时,就使用了其中一个属性:

```
this.BackColor = Color.White;
// has the same effect as:
// this.BackColor = Color.FromArgb(255, 255, 255);
```

有几百种这样的颜色。完整的列表参见 SDK 文档。它们包括所有的纯色:红、白、蓝、绿和黑等,还包括 `MediumAquaMarine`、`LightCoral` 和 `DarkOrchid` 等颜色。还有一个 `KnownColor` 枚举,它列出了命名颜色。



每种命名颜色都表示一组精确的 RGB 值,它们最初是多年前选择出来用在 Internet 上的。这种思想提供了色谱上一组有用的颜色,Web 浏览器可以辨识出这些颜色的名称——因此不必在 HTML 代码中显式写出它们的 RGB 值。几年前,这些颜色仍很重要,因为早期的浏览器不必准确地显示非常多的颜色,命名的颜色应该提供了一组在大多数浏览器中准确地显示出来的颜色。目前它们已经不那么重要了,因为现代的 Web 浏览器能准确地显示任何 RGB 值。还有一些 Web 安全的调色板可以为开发人员提供可用于大多数浏览器的颜色的完整列表。

## 48.6.3 图形显示模式和安全的调色板

尽管原则上显示器可以显示超过 1600 万种 RGB 颜色,但实际上这取决于如何在计算机上设置

显示属性。在 Windows 中，传统上有 3 个主要的颜色选项(尽管有些计算机还依据硬件提供其他选项)：真彩色(24 位)、增强色(16 位)和 256 色(在目前的一些图形卡上，真彩色实际上标记为 32 位，这必须对硬件进行优化，尽管此时 32 位中只有 24 位用于该颜色)。

只有真彩色模式允许同时显示所有的 RGB 颜色。这听起来是最佳选择，但它是有代价的：完整的 RGB 值需要用 3 个字节来保存，这表示要显示的每个像素都需要用图形卡内存中的 3 个字节来保存。如果图形卡内存费用较高(这种限制现在已经不像以前那样普遍了)，就可以选择一种其他模式。增强色模式用两个字节表示一个像素。每个 RGB 分量用 5 位就足够了。所以红色只有 32 种不同的强度，而不是 256 种。蓝色和绿色也是这样，总共有 65 536 种颜色。这对于需要偶尔查看照片质量的图像足够了，但比较微妙的阴影区域会被破坏。

256 色模式给出的颜色更少。但是在这种模式下，可以选择任何颜色，系统会建立一个调色板，这是一个从 1600 万种 RGB 颜色中选择出来的 256 种颜色的一个列表。在调色板中指定了颜色后，图形设备就能够只显示所指定的这些颜色。调色板在任何时候都可以改变——但图形设备每次只能在屏幕上显示 256 种不同的颜色。只有当获得高性能和视频内存费用较高时，才使用 256 色模式。大多数计算机游戏都使用这种模式——它们仍能得到相当好的图形，因为调色板经过了非常仔细的选择。

一般情况下，如果显示设备使用增强色或 256 色模式，并要显示某种 RGB 颜色，它就会从能显示的颜色池中选择一种在数学上最接近的匹配颜色。因此知道颜色模式非常重要。如果要绘制某些涉及微妙阴影或照片质量的图像，而用户没有选择 24 位颜色模式，就看不到期望的效果。如果要使用 GDI+ 进行绘制，就应该用不同的颜色模式测试应用程序(应用程序也可以通过编程设置给定的颜色模式，尽管本章不讨论这个问题)。

#### 48.6.4 安全调色板

下面简要介绍安全调色板，这是一种非常常见的默认调色板。它工作的方式是为每种颜色分量设置 6 个间隔相等的值，这些值分别是 0、51、102、153、204、255。换言之，红色分量可以是这些值中的任一个。绿色分量和蓝色分量也一样。所以安全调色板中的颜色就包括(0,0,0) (黑色)、(153,0,0) (暗红色)、(0, 255, 102) (蓝绿色)等，这样就得到了  $6^3=216$  种颜色。这是一种让调色板包含色谱中间隔相等的颜色和所有亮度的简单方式，但实际上这是不可行的，因为数学上等间隔的颜色分量并不表示这些颜色的区别在人眼看来也是相等的。

如果把 Windows 设置为 256 色模式，默认的调色板就是安全调色板，其中添加了 20 种标准的 Windows 颜色和 20 种备用颜色。

### 48.7 画笔和钢笔

本节介绍两个辅助类，在绘制图形时需要使用它们。前面已经用到过 Pen 类，它用于告诉 graphics 实例如何绘制线条。相关的类是 System.Drawing.Brush，它告诉图形实例如何填充线条。例如，Pen 用于绘制前面示例中矩形和椭圆的边框。如果需要把这些图形绘制为实心的，就要使用画笔指定如何填充它们。这两个类有一个共同点：很难对它们调用任何方法。用需要的颜色和其他属性构造一个 Pen 或 Brush 实例，再把它传递给需要 Pen 或 Brush 的绘图方法即可。



如果读者以前使用 GDI 编程,就可能会注意到在前两个示例中,在 GDI+中使用 Pen 的方式是不同的。在 GDI 中,一般是调用一个 Windows API 函数 SelectObject(),它实际上把钢笔关联到设备上下文上。这支钢笔用于所有需要钢笔的绘图操作中,直到再次调用 SelectObject()方法通知设备上下文停止使用它时为止。这条规则也适用于画笔和其他对象,如字体和位图。而使用 GDI+, Microsoft 选择一种无状态的模式,其中没有默认的钢笔或其他辅助对象。只需给特定方法调用指定合适的辅助对象即可。

### 48.7.1 画笔

GDI+有几种不同类型的画笔,本章不准备详细介绍它们,这里仅解释几种比较简单的画笔,读者掌握其要领即可。每种画笔都由一个派生自抽象类 System.Drawing.Brush 的类的实例来表示。最简单的画笔 System.Drawing.SolidBrush 仅指定了区域用纯色来填充:

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
Brush solidFunnyOrangyBrownBrush = new SolidBrush(Color.FromArgb(255,155,100));
```

另外,如果画笔是一种 Web 安全色,就可以用另一个类 System.Drawing.Brushes 构造画笔。Brushes 是永远不能真正地实例化的一个类(它有一个私有构造函数,禁止实例化它)。它只有许多静态属性,每个属性都返回指定颜色的画笔。可以像下面这样使用画笔:

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

比较复杂的一种画笔是影线画笔(hatch brush),它通过绘制一种图案来填充区域。因为这种类型的画笔比较高级,所以在 Drawing2D 名称空间中,用 System.Drawing.Drawing2D.HatchBrush 类表示。Brushes 类不能帮助我们使用影线画笔,而需通过提供一个影线型式和两种颜色(前景色和背景色,背景色可以忽略,此时将使用默认的黑色),来显式构造一支影线画笔。影线样式可以取自于枚举 System.Drawing.Drawing2D.HatchStyle, 其中有许多 HatchStyle 值,其完整列表参阅 SDK 文档。大体上,一般的样式包括 ForwardDiagonal、Cross、DiagonalCross、SmallConfetti 和 ZigZag。构造影线画笔的示例如下所示。

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross, Color.Azure);
// background color of CrossBrush is black
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
```

GDI 只能使用实线画笔和影线画笔, GDI+添加了两种新画笔:

- System.Drawing.Drawing2D.LinearGradientBrush 用一种在屏幕上变化的颜色填充区域。
- System.Drawing.Drawing2D.PathGradientBrush 与上述画笔类似,但其颜色沿着要填充的区域的路径变化。

如果细心使用,这些画笔就可以显现一些惊人的效果。

48.7.2 钢笔

与画笔不同，钢笔只用一个类 `System.Drawing.Pen` 来表示。但钢笔比画笔复杂一些，因为它需要指定线条应有多宽(像素)，对于比较宽的线条，还要确定如何填充该线条中的区域。钢笔还可以指定其他许多属性，本章不讨论它们，但其中包括前面提到的 `Alignment` 属性，该属性表示相对于图形的边框，线条该如何绘制，以及在线条的末尾绘制什么图形(是否使图形光滑过渡)。

粗线条中的区域可以用纯色填充，或者使用画笔来填充。因此 `Pen` 实例可以包含 `Brush` 实例的引用。这非常强大，因为这表示我们可以绘制用影线阴影或线性阴影着色的线条。构造自己设计的 `Pen` 实例有 4 种不同的方式：可以传递一种颜色，或者传递一支画笔。这两个构造函数都会生成一个像素宽的钢笔。另外，还可以传递一种颜色或一支画笔，以及一个表示钢笔宽度的 `float` 类型的值。(该宽度必须是一个 `float` 类型的值，以允许执行绘图操作的 `Graphics` 对象使用非默认的单位，如毫米或英寸，例如可以指定宽度是小数形式的英寸)。例如，可以构造类似如下的钢笔：

```
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
Pen solidBluePen = new Pen(Color.FromArgb(0,0,255));
Pen solidWideBluePen = new Pen(Color.Blue, 4);
Pen brickPen = new Pen(brickBrush);
Pen brickWidePen = new Pen(brickBrush, 10);
```

另外，为了快速构造钢笔，还可以使用 `System.Drawing.Pens` 类，它与 `Brushes` 类一样，包含许多备用钢笔。这些钢笔的宽度都是一个像素，使用通常的 Web 安全色，这样就可以用下述方式构建钢笔：

```
Pen solidYellowPen = Pens.Yellow;
```

48.8 绘制图形和线条

前面介绍了在屏幕上绘制规定的图形所需要的所有基类和对象。下面复习 `Graphics` 类可以使用的一些绘图方法，并用一个小示例来介绍几种画笔和钢笔的用法。

`System.Drawing.Graphics` 类有很多方法，利用这些方法可以绘制各种线条、空心图形和实心图形。表 48-4 所示的列表并不完整，但给出了主要的方法，您应能据此掌握绘制各种图形的要领。

表 48-4

方 法	常 见 参 数	绘制的图形
<code>DrawLine()</code>	钢笔、起点和终点	一条直线
<code>DrawRectangle()</code>	钢笔、位置和大小	空心矩形
<code>DrawEllipse()</code>	钢笔、位置和大小	空心椭圆
<code>FillRectangle()</code>	画笔、位置和大小	实心矩形
<code>FillEllipse()</code>	画笔、位置和大小	实心椭圆
<code>DrawLines()</code>	钢笔、点数组	一组线条，把数组中的每个点按顺序连接起来
<code>DrawBezier()</code>	钢笔、4 个点	经过两个端点的一条光滑曲线，剩余的两个点用于控制曲线的形状

(续表)

方 法	常 见 参 数	绘制的图形
DrawCurve()	钢笔、点数组	经过这些点的一条光滑曲线
DrawArc()	钢笔、矩形、两个角	由角度定义的矩形中圆的一部分
DrawClosedCurve()	钢笔、点数组	与 DrawCurve 一样，但还要绘制一条用于闭合曲线的直线
DrawPie()	钢笔、矩形、两个角	矩形中的空心楔形
FillPie()	画笔、矩形、两个角	矩形中的实心楔形
DrawPolygon()	钢笔、点数组	与 DrawLines 一样，但还要连接第一点和最后一点，以闭合绘制的图形

在结束绘制简单对象的主题前，本节用一个简单示例来说明使用画笔可以得到的各种可视效果。该示例是 ScrollMoreShapes，它基本上是 ScrollShapes 的修正版本。除了矩形和椭圆外，我们还添加了一条粗线条，用各种自定义画笔填充图形。前面解释了绘图规则，所以这里只给出代码，而不进行过多的注释。首先，因为添加了新画笔，所以需指定使用 System.Drawing.Drawing2D 名称空间：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

接着是 Form1 类中的一些额外字段，其中包含了要绘制图形详细的位置信息，以及要使用的各种钢笔和画笔：



```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
                                                    new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
                                                new Size(200,150));
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
private readonly Brush solidAzureBrush = Brushes.Azure;
private readonly Brush solidYellowBrush = new SolidBrush(Color.Yellow);
private static readonly Brush brickBrush = new
    HatchBrush(HatchStyle.DiagonalBrick, Color.DarkGoldenrod, Color.Cyan);
private readonly Pen brickWidePen = new Pen(brickBrush, 10);
```

代码段 ScrollMoreShapes.sln

把 BrickBrush 字段声明为静态，就可以使用该字段的值初始化 brickWidePen 字段。C#不允许使用一个实例字段初始化另一个实例字段，因为还没有定义要先初始化哪个实例字段。然而，如果把字段声明为静态字段就可以解决这个问题，因为只实例化了 Form1 类的一个实例，字段是静态字段还是实例字段就不重要了。

下面是 OnPaint()方法的重写版本：





可从  
wrox.com  
下载源代码

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Point scrollOffset = AutoScrollPosition;
    dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);
    if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
        e.ClipRectangle.Left+scrollOffset.Y < 250)
    {
        dc.DrawRectangle(bluePen, rectangleBounds);
        dc.FillRectangle(solidYellowBrush, rectangleBounds);
        dc.DrawEllipse(redPen, ellipseBounds);
        dc.FillEllipse(solidAzureBrush, ellipseBounds);
        dc.DrawLine(brickWidePen, rectangleBounds.Location,
            ellipseBounds.Location+ellipseBounds.Size);
    }
}
```

代码段 ScrollMoreShapes.sln

与以前一样，也将 AutoScrollMinSize 设置为(250,350)，结果如图 48-13 所示。

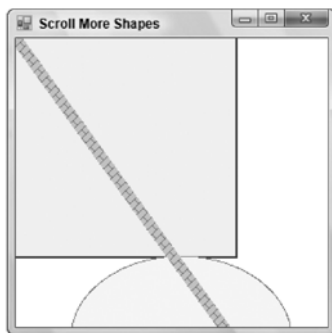


图 48-13

注意，粗对角线已在矩形和椭圆上绘制出来，它需要最后一个绘制。

## 48.9 显示图像

使用 GDI+ 最常见的操作是希望显示文件中已有的图像。这确实要比绘制自己的用户界面简单多了，因为图像已经预先绘制好了。实际上，我们只需要加载文件，让 GDI+ 显示它即可。图像可以只包含一个绘制的简单线条或一个图标，也可以比较复杂，如一张照片。对图像也可以执行某些操作，如拉伸或旋转图像，也可以选择只显示图像的一部分。

略有变化，本节将先给出一个示例，再讨论显示图像时需要注意的一些问题。可以这么做的原因是显示图像的代码非常简单。

我们需要 .NET 的一个基类 `System.Drawing.Image`。`Image` 类的一个实例表示一幅图像。读取一幅图像仅需使用一行代码：

```
Image myImage = Image.FromFile("FileName");
```

FromFile()方法是 Image 类的一个静态成员,是实例化图像的常用方式。文件可以是任何支持的图形文件格式,包括.bmp、.jpg、.gif 和.png。

显示图像也很简单,假定手头有一个合适的 Graphics 实例,那么调用 Graphics.DrawImage Unscaled()方法或 Graphics.DrawImage()方法就足够了。这些方法都有许多重载方法,可以根据图像的位置和要绘制的大小非常灵活地处理用户提供的信息。然后本示例使用 DrawImage()方法:

```
dc.DrawImage(myImage, points);
```

在这行代码中,假定 dc 是一个 Graphics 实例,MyImage 是要显示的图像,points 是一个 Point 结构数组,其中 points[0]、points[1]和 points[2]是图像左上角、右上角和左下角的坐标。



熟悉 GDI 的开发人员可以从图像中看出 GDI 与 GDI+的最大区别。在 GDI 中,显示图像涉及几个重要的步骤。如果图像是一个位图,加载它就很简单,但如果它是任何其他类型的文件,加载它就会涉及 OLE 对象的一系列调用。实际上,把加载的图像显示到屏幕上要获得它的一个句柄,把它放在一个内存设备上下文中,然后在设备环境之间执行一个块传输。尽管设备上下文和句柄仍在后台上,但如果要开始从代码中对图像进行复杂的编辑,就需要它们。简单的任务完美地封装在 GDI+对象模型上。

下面用一个示例 DisplayImage 来说明显示图像的过程。这个示例在应用程序的主窗口中显示一个.jpg 文件。要使操作过程简单一些,.jpg 文件的路径硬编码到应用程序中(如果运行该示例,就需要改变该路径,以反映在系统中文件的位置)。要显示的.jpg 文件是圣彼得堡的日落图片。

与其他例子一样,DisplayImage 项目是 C# Visual Studio 2010 生成的一个标准的 Windows 应用程序。在 Form1 类中添加下述字段:

```
readonly Image piccy;  
private readonly Point [] piccyBounds;
```

然后在 Form1()构造函数中加载文件:



可从  
wrox.com  
下载源代码

```
public Form1()  
{  
    InitializeComponent();  
    piccy =  
        Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.jpg");  
    AutoScrollMinSize = piccy.Size;  
    piccyBounds = new Point[3];  
    piccyBounds[0] = new Point(0,0); // top left  
    piccyBounds[1] = new Point(piccy.Width,0); // top right  
    piccyBounds[2] = new Point(0,piccy.Height); // bottom left  
}
```

代码段 DisplayPicture.sln

注意,图像的大小(以像素为单位)通过其 Size 属性来获得,我们使用该属性设置文档区域。再建立一个 piccyBounds 数组,它用于标识图像在屏幕上的位置。选择 3 个角的坐标,按实际大小和图形来绘制图像,但如果要重新设置图像的大小、拉伸图像,或甚至把图像变形为非矩形的平行四

边形，则可以仅改变 `piccyBounds` 数组中 `Point` 的值来实现上述操作。

通过 `OnPaint()` 重写方法显示该图像：



可从  
wrox.com  
下载源代码

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.ScaleTransform(1.0f, 1.0f);
    dc.TranslateTransform(AutoScrollPosition.X, AutoScrollPosition.Y);
    dc.DrawImage(piccy, piccyBounds);
}
```

代码段 DisplayPicture.sln

最后，特别注意对 IDE 生成的 `Form1.Dispose()` 方法代码进行修改：



可从  
wrox.com  
下载源代码

```
protected override void Dispose(bool disposing)
{
    piccy.Dispose();
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
```

代码段 DisplayPicture.sln

只要不再需要该图像了，就应立即删除它，这一点很重要因为图像在使用时一般会占用许多内存。在调用 `Image.Dispose()` 方法后，因为 `Image` 实例不再引用任何实际图像，所以不能再显示该图像(除非加载了一幅新图像)。

运行代码，会得到如图 48-14 所示的结果。

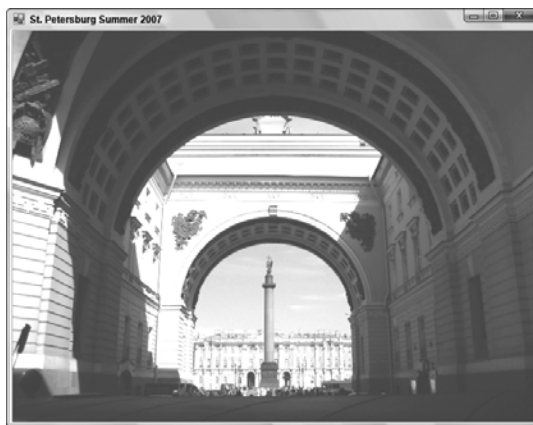


图 48-14

## 48.10 处理图像时的问题

尽管显示图像很简单，但需要理解一些在后台执行的什么操作。

理解图像最重要的一点是，图像的形状总是矩形。这不只是出于方便原因，其原因是底层的技术。所有现代图形卡都内置了硬件，从而可以非常高效地从内存的一个地方把像素块复制到内存的另一个地方。假定像素块表示一个矩形区域，这个硬件加速操作可以虚拟为一个操作，而且执行速度非常快。实际上，这是现代高性能图形的关键。这个操作称为位图块传输(或者 BitBlt)。Graphics.DrawImageUnscaled()方法在内部使用 BitBlt，这就是为什么能够看见一幅大图像的原因，该图像也许包含上百万个像素，但几乎是立即就显示出来。如果计算机必须把图像逐个像素地复制到屏幕上，则该图像最多在几秒钟内逐渐画出来。

因为 BitBlt 的效率非常高，所以图像的所有绘制和处理操作都使用 BitBlt 完成。甚至图像的某些编辑操作也使用表示内存区域的设备环境之间部分图像的 BitBlt 完成。在 GDI 时代中，Windows 32 API 函数 BitBlt()是最重要、使用最广泛的图像处理函数，而在 GDI+中，BitBlt 操作一般隐藏在 GDI+对象模型中。

尽管很容易模拟类似的效果，但图像的 BitBlt 区域不可能是矩形。一种方式是为了 BitBlt，把某种颜色标记为透明的。这样源图像中该颜色的区域不会覆盖目标设备中对应像素的已有颜色。在 BitBlt 过程中还可以指定，结果图像的每个像素会在进行 BitBlt 前，对源图像上和目标设备上该像素的颜色进行某些逻辑操作来形成(如按位 AND)。这样的操作由硬件加速来支持，用于产生各种微妙的效果。注意 Graphics 对象实现另一个方法 DrawImage()，该方法类似于 DrawImageUnscaled()方法，但它有许多重载方法，可以指定 BitBlt 更复杂的形式，以便在绘图过程中使用。DrawImage()方法还可以只绘制(使用 BitBlt)图像的某个特定部分，或者对它执行其他特定操作，如在绘图时缩放(缩放其大小)它。

## 48.11 绘制文本

到目前为止，本章还有一个非常重要的问题要讨论——显示文本。因为在屏幕上绘制文本通常比绘制简单图形更复杂。在不考虑外观的情况下，只显示一两行文本非常简单——它只需调用 Graphics 实例的一个方法 Graphics.DrawString()。但如果要显示一个文档，其中有许多文本，则事情很快变得复杂多了，这有两个原因：

- 如果只考虑外观，则需要理解字体。图形的绘制需要使用画笔和钢笔作为帮助对象，绘制文本的过程则需要把字体作为帮助对象。而且，理解字体并不容易。
- 在窗口中需要仔细布局文本。用户通常期望文字一个跟一个地排列——排成一行，其间有一定的间隔。这是一个比想象中更困难的任务。对于初学者，一般事先不知道屏幕上文字之间的间隔有多大。这需要计算(使用 Graphics.MeasureString()方法)。另外，屏幕上文字占用的间隔会影响文档中后续的文字在屏幕上放置的位置。如果应用程序自动换行，就需要在确定应在何处放置换行符前仔细斟酌文字的大小。下次运行 Microsoft 的 Word 时，仔细看看 Word 是如何重新定位用户输入的文本的。这里有许多复杂的处理操作。有可能任何 GDI+ 应用程序都不像 Word 那样复杂，但如果需要显示任何文本，仍需要考虑同样的问题。

总之，好的文本处理需要一定的技巧。假定知道字体和放置它的位置，那么把一行文本显示在屏幕上实际上非常简单。因此，下一节介绍一个小示例，说明如何显示一些文本。此后，探讨字体和字体系列的一些规则，并介绍一个更真实(和相关)的文本处理示例 CapsEditor。

## 48.12 简单的文本示例

这个示例——DisplayText 是常见的 Windows 窗体效果。这次重写了 OnPaint()方法，添加了成员字段，如下所示：



```
private readonly Brush blackBrush = Brushes.Black;
private readonly Brush blueBrush = Brushes.Blue;
private readonly Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private readonly Font boldTimesFont = new Font("Times New Roman", 10,
    FontStyle.Bold);
private readonly Font italicCourierFont = new Font("Courier", 11,
    FontStyle.Italic | FontStyle.Underline);

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.DrawString("This is a groovy string", haettenschweilerFont, blackBrush,
        10, 10);
    dc.DrawString("This is a groovy string " +
        "with some very long text that will never fit in the box",
        boldTimesFont, blueBrush,
        new Rectangle(new Point(10, 40), new Size(100, 40)));
    dc.DrawString("This is a groovy string", italicCourierFont, blackBrush,
        new Point(10, 100));
}
```

代码段 DisplayText.sln

运行这个示例，会得到如图 48-15 所示的结果。

这个示例说明了如何使用 Graphics.DrawString()方法绘制文本项，DrawString()方法有许多重载版本，这里介绍其中的 3 个。但这些不同的重载方法都需要用参数指定要显示的文本、绘制字符串所使用的字体，以及用于构造各种线条和曲线以组成每个文本字符的画笔。其余的参数有另外两种指定方式。但一般情况下，可以指定一个 Point(或两个等价的数字)或一个 Rectangle。

如果指定 Point，文本就从该 Point 的左上角开始，并仅向右延伸。如果指定 Rectangle，Graphics 实例就把字符串放在该矩形的内部。如果文本在矩形内部容纳不下，它就会被剪切，如图 48-15 中的第 4 行文本所示。把矩形传递给 DrawString()方法，表示绘图过程将持续较长时间，因为 DrawString()方法需要指定在什么地方放置换行符，但其结果应看起来好一些(如果字符串可以放在矩形中)。



图 48-15

这个示例还说明了构造字体的两种方法，一般需要包含字体的名称及其大小(高度)。也可以选择性地传递不同的样式以修改文本的绘制方式(黑体、下划线等)。

## 48.13 字体和字体系列

字体准确地描述每个字母的显示方式。在一个文档中选择合适的字体和提供适当数量的不同字体，对于提高该文档的可读性非常重要。

大多数人在提到字体时，会指定 Arial、Times New Roman(Windows 用户)或 Times、Helvetica(Mac OS 用户)等。实际上，这些都不是字体，它们是字体系列(font families)，字体系列以通俗的术语来说，是文本的可视化风格。字体系列是应用程序整体外观中的一个关键因素，大多数人都能识别常用字体系列的风格，即使我们意识不到这一点。

而实际字体应是像 Arial 9-point italic 这样的东西。换言之，字体添加了更多的信息，如指定文本的大小，以及是否对该文本进行其他修改等。这些修改包括文本是否为黑体、斜体、带下划线，或者显示为小型大写字母或下标，这种修改在技术上称为样式，尽管在某些情况下该术语有误导作用，因为可视化外观主要取决于字体系列。

通过指定文本的高度，就可以测量其大小。高度以点为单位来测量，这是一个传统单位，表示 1/72 英寸(0.351 毫米)。例如，10 点的字体高度大约为 1/7 英寸或 3.5 毫米。但字体大小为 10 点的 7 行文本，在屏幕或纸上的高度不等于 1 英寸，因为还需要考虑行与行之间的间隔。



严格来讲，测量高度并不像这样简单，因为还需要考虑几个不同的高度。例如，较高字母，如 A 或 F 的高度(这是指我们讨论高度时字母的真实高度)，重音字母如 Å 或 Ñ 中的额外高度(内部前导)，以及字母的尾部超出底线的附加高度如 y 和 g(下行高度)。但是本章不讨论这些高度，一旦指定了字体系列和主要高度，这些辅助高度就会自动确定。

在处理字体时，还会遇到其他常用于描述某种字体系列的术语：

- **Serif** 字体系列在组成字符的许多线条尾部有一个小标记(这些标记称为衬线)。Times New Roman 就是这种字体的一个典型例子。
- 相反，**Sans Serif** 字体系列没有这些小标记。例如，Arial 和 Verdana。没有小标记常会使文本看起来比较生硬，所以 Sans Serif 字体常用于重要的文本。
- **TrueType** 字体系列以一种精确的数学方式定义组成字符的曲线形状。这表示可以使用相同的定义来计算如何在系列内部绘制任意大小的字体。目前，我们实际上使用的所有字体都是 TrueType 字体。Windows 3.1 时代的一些旧字体系列通过指定每种字体大小的每个字符的位图来定义，但现在最好不要使用这些字体。

Microsoft 提供了两个主要类来处理何时选择或处理字体：

- `System.Drawing.Font`
- `System.Drawing.FontFamily`

前面已经介绍了 `Font` 类的主要用法。在绘制文本时，实例化 `Font` 的一个实例，并把它传递给 `DrawString()` 方法，以确定应该如何绘制文本。`FontFamily` 实例用于表示一个字体系列。

FontFamily 类的一个用法是：如果知道需要某种类型的字体(Serif、Sans Serif 或 TrueType)，但不介意使用哪种字体，就可以使用该类。静态属性 GenericSerif、GenericSansSerif 和 GenericMonospace 返回满足这些条件的默认字体：

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

然而，如果编写一个高级应用程序，就应以更先进的方法选择字体。可以实现绘图代码，从而使它检查哪些字体系列可用，并选择合适的字体，例如从偏爱的字体列表中选择第一种可用的字体。进一步地，如果要让应用程序的用户友好性更高，则列表中的第一个选项可能是用户上次运行软件时选择的字体。通常情况下，安全起见，最好使用最常用的字体系列，如 Arial 和 Times New Roman。但如果要使用某种不存在的字体显示文本，则结果通常总是不可预料的。很容易发现，Windows 仅是替代了标准系统字体，从而系统很容易绘制文本，但这看起来并不是非常友好，如果文档出现这种情况，就会使人觉得软件的质量非常糟糕。

使用 InstalledFontCollection 类可以查看系统上的可用字体，该类在 System.Drawing.Text 名称空间中。这个类实现一个属性 Families，该属性是一个包含系统上所有可用字体的数组：

```
InstalledFontCollection insFont = new InstalledFontCollection();
FontFamily [] families = insFont.Families;

foreach (FontFamily family in families)
{
    // do processing with this font family
}
```

## 48.14 示例：枚举字体系列

本节介绍一个小示例 EnumFontFamilies，该示例列出系统上所有可用的字体系列，使用合适的字体(该字体系列的 12 点常规版本)显示它们的名称。运行这个示例，会得到如图 48-16 所示的结果。

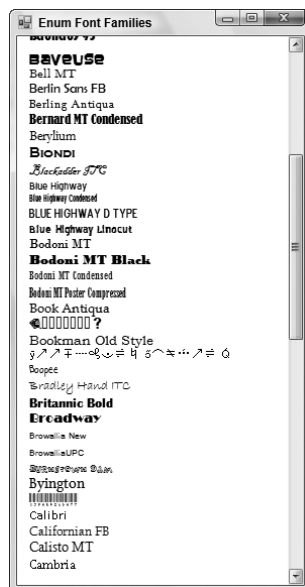


图 48-16

当然，根据计算机上安装的字体，用户在运行这个示例时，可能会得到不同的结果。

对于这个示例，创建一个标准的 C# Windows 应用程序 ListFonts，首先添加要搜索的另一个名称空间。我们要使用 `InstalledFontCollection` 类，它在 `System.Drawing.Text` 名称空间中定义：

```
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;
```

然后在 `Form1` 类中添加如下常量：

```
private const int margin = 10;
```

`margin` 是文本与文档边缘之间的左边距和上边距的大小——它防止文本显示在工作区边缘的右边。

由于该示例以快捷方式显示字体系列，因此，代码比较粗糙，在许多情况下并不是以实际应用程序中的方式执行任务。例如，我们把文档的大小设置为一个估计值(200,1500)，使用 Visual Studio 2010 的 Properties 窗口把 `AutoScrollMinSize` 属性设置为这个值。一般情况下，必须查看要显示的文本，计算出文档的大小。下一节介绍这个过程。

下面是 `OnPaint()` 方法：



```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    int verticalCoordinate = margin;
    InstalledFontCollection insFont = new InstalledFontCollection();
    FontFamily [] families = insFont.Families;
    e.Graphics.TranslateTransform(AutoScrollPosition.X,
                                AutoScrollPosition.Y);

    foreach (FontFamily family in families)
    {
        if (family.IsStyleAvailable(FontStyle.Regular))
        {
            Font f = new Font(family.Name, 10);
            Point topLeftCorner = new Point(margin, verticalCoordinate);
            verticalCoordinate += f.Height;
            e.Graphics.DrawString (family.Name, f,
                                  Brushes.Black, topLeftCorner);

            f.Dispose();
        }
    }
}
```

代码段 ListFonts.sln

在这段代码中，首先使用 `InstalledFontCollection` 对象获得一个数组，其中包含所有可用的字体系列的详细信息。对于每个系列，我们实例化大小为 10 点的一个 `Font`。为 `Font` 使用了一个简单的构造函数——有许多构造函数可以指定更多的选项。使用的构造函数带有两个参数：系列的名称和字体的大小：

```
Font f = new Font(family.Name, 10);
```

这个构造函数构造了一个一般风格的字体。但是为了安全起见，在使用该字体显示任何文本前，



先检查一下这种风格是否可用于每个字体系列。这利用方法 `FontFamily.IsStyleAvailable()` 实现。这种检查非常重要，因为并不是所有的字体都可以使用所有的风格：

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

`FontFamily.IsStyleAvailable()` 方法带一个参数，即 `FontStyle` 枚举。该枚举包含许多标记，它们可以用按位 OR 运算符来组合。可能的标记有 `Bold`、`Italic`、`Regular`、`Strikeout` 和 `Underline`。

最后，使用 `Font` 类的一个属性 `Height`，该属性返回显示该字体对应的文本需要的高度，以便算出行间距：

```
Font f = new Font(family.Name, 10);
Point topLeftCorner = new Point(margin, verticalCoordinate);
verticalCoordinate += f.Height;
```

为了使事情简单化，`OnPaint()` 方法的这个版本暴露出一些不太好的编程问题。首先，没有检查哪些文档区域需要绘制——而是显示所有内容。另外，如前所述，因为实例化 `Font` 是一个计算量较大的过程，所以应保存字体，而不是每次调用 `OnPaint()` 方法时都实例化新副本。因为这样设计出来的代码对于本示例将需要花费相当长的时间来绘图。为了节省内存，帮助垃圾回收器，在完成操作后对每个 `Font` 实例调用 `Dispose()` 方法。如果不这样，在 10 或 20 次绘图操作后，就会浪费许多内存存储不再需要的字体。

## 48.15 编辑文本文档：CapsEditor 示例

下面是本章中经过扩展的一个示例。`CapsEditor` 示例用于说明如何把前面介绍的绘图规则应用到实际环境中。这个示例除了响应用户通过鼠标输入的信息外，不需要任何新资料，但它将说明如何管理文本的绘制，让应用程序仍具有高性能，并确保主窗口的工作区的内容总是最新的。

`CapsEditor` 程序允许用户读取文本文件，该文本逐行显示在工作区中。如果用户双击任何一行文本，该行就会全部变为大写。这就是该示例的功能。即使只有这组有限的功能，也涉及许多复杂的工作：确保把所有的信息都显示在正确的位置上，并考虑性能问题。特别是这里有一个新元素，文档的内容可以修改——用户选择菜单项，以读取一个新文件时；或用户双击一行，使之全部变为大写字母时，都要修改文档的内容。在第一种情况下，需要更新文档的大小，使滚动条仍能正确工作，并重新显示所有内容。在第二种情况下，需要仔细检查文档的大小是否发生了变化，哪些文本需要重新显示。

本节首先看看 `CapsEditor` 的外观。第一次运行该应用程序时，它没有已加载的文档，显示结果如图 48-17 所示。

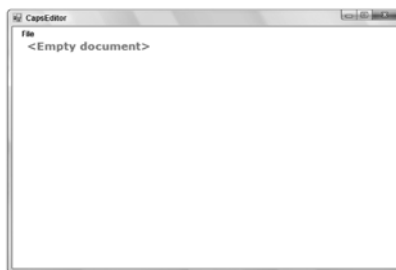


图 48-17

File 菜单有两个菜单项: Open 和 Exit。Open 调用 OpenFileDialog()方法, 读取用户选择的任何文件 Exit 退出应用程序。图 48-18 是 CapsEditor 显示其源文件 Form1.cs 的情形(我们已经在该图片中双击几行, 把它们全部转换为大写)。

水平和垂直滚动条的大小也是正确的。滚动工作区, 使之正好显示整个文档。CapsEditor 不会给文本换行——即使没有这个功能, 该示例也比较复杂了。它只显示文件被读取的各行文本。对文件的大小没有限制, 但这里假定这是一个文本文件, 且不包含任何非打印字符。

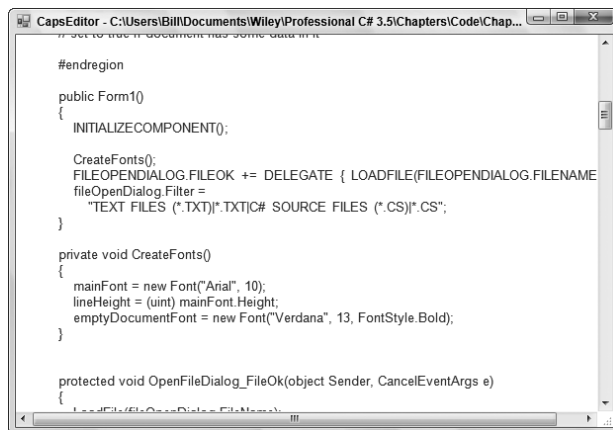


图 48-18

首先添加一些 using 命令:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
```

这是因为我们使用了 StreamReader 类, 它在 System.IO 名称空间中。接着, 在 Form1 类中添加一些字段:



可从  
wrox.com  
下载源代码

```
#region Constant fields
private const string standardTitle = "CapsEditor";
// default text in titlebar
private const uint margin = 10;
// horizontal and vertical margin in client area
#endregion
#region Member fields
// The 'document'
private readonly List<TextLineInformation> documentLines =
    new List<TextLineInformation>();
private uint lineHeight; // height in pixels of one line
private Size documentSize; // how big a client area is needed to
// display document
private uint nLines; // number of lines in document
private Font mainFont; // font used to display all lines
private Font emptyDocumentFont; // font used to display empty message
```

```

private readonly Brush mainBrush = Brushes.Blue;
                                // brush used to display document text
private readonly Brush emptyDocumentBrush = Brushes.Red;
                                // brush used to display empty document message
private Point mouseDoubleClickPosition;
    // location mouse is pointing to when double-clicked
private readonly OpenFileDialog fileOpenDialog = new OpenFileDialog();
    // standard open file dialog
private bool documentHasData = false;
    // set to true if document has some data in it
#endregion

```

---

代码段 CapsEditor.sln

---

这些字段中的大多数都很容易理解。`documentLines` 字段是一个 `List<TextLineInformation>`，它包含文件中已经读入的实际文本。实际上，这个字段包含了文档中的数据。`documentLines` 的每个元素都包含一行已读入的文本的信息。因为它是一个 `List<TextLineInformation>`，而不是常见的数组，所以在读取文件时，可以给它动态地添加元素。

如前所述，每个 `documentLines` 元素都包含一行文本的相关信息。这些信息实际上是另一个类 `TextLineInformation` 的一个实例：

```

class TextLineInformation
{
    public string Text;
    public uint Width;
}

```

`TextLineInformation` 类看起来像一个典型的示例，其中宁愿使用结构，而不是类，因为它只是把几个字段组合在一起。但它的实例总是作为 `List<TextLineInformation>` 的元素来访问，这样其元素就会存储为引用类型。

每个 `TextLineInformation` 实例都存储了一行文本——它可以看作是显示为一项的最小项，一般情况下，在 GDI+应用程序中，对于每个这样的项，都要存储该项的文本，以及显示它的世界坐标和其大小(只要用户进行了滚动操作，页面坐标通常就会改变，而世界坐标通常只有在文档的其他部分以某种方式进行了修改时才会改变)。本例只需存储该项的 `Width` 即可。其原因是此时的高度是选中字体的高度，它对于所有的文本行都一样，所以不必为每行文本单独地存储高度。它只需要在字段 `Form1.lineHeight` 字段中存储一次即可，位置也是这样。在这里，`x` 坐标等于页边距，`y` 坐标很容易计算出来：

```
margin + lineHeight*(本行上面显示的行数)
```

如果要显示和操作单个单词，而不是整行文本，则每个单词的 `x` 坐标必须使用该文本行上在该单词前面的所有单词的宽度来计算，但为了使这个计算简单一些，应把每行文本当作单一的项来看待。

下面处理主菜单。这部分应用程序涉及更多的是 Windows 窗体的内容(参见第 39 章)，而不是 GDI+ 的内容。在 Visual Studio 2010 中使用设计视图添加菜单项，把它们重命名为 `menuFile`、`menuFileOpen` 和 `menuFileExit`。接着使用 Visual Studio 2010 的 Properties 窗口添加 File Open 和 File Exit 菜单项的事件处理程序。这些事件处理程序的名称是 Visual Studio 2010 生成的，即 `menuFileOpen_Click()` 和 `menuFileExit_Click()`。

还需要在 Form1()构造函数中添加一些初始化代码:



```
public Form1()
{
    InitializeComponent();
    CreateFonts();
    fileOpenDialog.FileOk += delegate { LoadFile(fileOpenDialog.FileName); };
    fileOpenDialog.Filter =
        "Text files ( * .txt)| * .txt|C# source files ( * .cs)| * .cs";
}
```

代码段 CapsEditor.sln

这里为实例添加的事件处理程序是在用户单击 **File Open** 对话框中的 **OK** 按钮时执行的。我们还给 **Open File** 对话框设置了筛选器, 只能加载文本文件。因为本示例选择了 .txt 文件和 C# 源文件, 所以可以使用应用程序查看示例的源代码。

CreateFonts()是一个辅助方法, 它挑选出要使用的字体:

```
private void CreateFonts()
{
    mainFont = new Font("Arial", 10);
    lineHeight = (uint)mainFont.Height;
    emptyDocumentFont = new Font("Verdana", 13, FontStyle.Bold);
}
```

处理程序的实际定义非常标准:

```
protected void menuFileOpen_Click(object sender, EventArgs e)
{
    fileOpenDialog.ShowDialog();
}
protected void menuFileExit_Click(object sender, EventArgs e)
{
    Close();
}
```

下面介绍 LoadFile()方法。这个方法处理文件的打开和读取操作, 并确保引发 **Paint** 事件, 以使用新文件强制重新绘图:



```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
}
```

```

        documentHasData = (nLines > 0) ? true: false;
        CalculateLineWidths();
        CalculateDocumentSize();
        Text = standardTitle + "-" + FileName;
        Invalidate();
    }

```

---

代码段 CapsEditor.sln

这个函数的大部分代码都是标准的文件读取操作(参见第 29 章)。注意在读取文件时,逐渐地给 documentLines ArrayList 添加文本行,使这个数组最终按顺序包含每行文本的信息。在读取文件后,设置 documentHasData 标记,它指定是否有要显示的信息。下一个任务是确定要显示内容的位置,之后确定显示文件的工作区有多大,以及用于设置滚动条的文档大小。最后,设置标题栏文本,并调用 Invalidate()方法。Invalidate()是 Microsoft 提供的一个非常重要的方法,所以下一节首先介绍这个方法的应用,之后介绍 CalculateLineWidths()和 CalculateDocument Size()方法的代码。

### 48.15.1 Invalidate()方法

Invalidate()方法是 System.Windows.Forms.Form 的一个成员,它把窗口的工作区标记为无效,因此在需要重新绘制时,它可以确保引发 Paint 事件。Invalidate()方法有两个重载版本:可以给它传递一个矩形,该矩形指定(使用页面坐标)需要重新绘制窗口的哪个区域。如果不提供任何参数,它就把整个工作区标记为无效。

如果知道需要绘制某些内容,为什么不调用 OnPaint()方法或直接完成绘制任务的其他方法?原因是,一般情况下,最好不要直接调用绘图例程——如果代码要完成某些绘图任务,一般就应调用 Invalidate()方法。其原因如下所示:

- 因为绘图总是 GDI+应用程序执行的一种处理器最密集型的任务,所以在其他工作的中间进行绘图会妨碍其他工作的进行。在前面的示例中,如果从 LoadFile()方法中直接调用一个方法来完成绘图,LoadFile()方法就将在绘图工作完成后才能返回。在这段时间里,应用程序不会响应任何其他事件。另一方面,通过调用 Invalidate()方法,在从 LoadFile()方法快速返回之前,仅可以让 Windows 引发一个 Paint 事件。接着 Windows 就可以检查等待处理的事件。其内部的工作方式是事件被当作消息队列中的一条消息。Windows 会定期检查该队列,如果其中有事件,Windows 就选择它,并调用相应的事件处理程序。现在 Paint 事件是队列中的唯一事件,所以 OnPaint()方法会被立即调用。但是,在一个比较复杂的应用程序中,可能会有其他事件,其中一些优先级比 Paint 事件高。特别是如果用户已决定退出应用程序,该事件就会用消息 WM\_QUIT 来标记。
- 如果有一个比较复杂的多线程应用程序,就会希望用一个线程处理所有的绘图操作。使用 Invalidate()方法可以把所有的绘图操作传递到消息队列中,这有助于确保无论其他线程请求什么绘图操作,都由同一个线程完成所有的绘图操作(无论什么线程负责消息队列,都是由线程 Application.Run()方法处理绘图操作)。
- 还有一个与性能有关的原因。假定在同一时刻有几个不同的屏幕绘制请求,也许代码刚刚修改文档,以确保显示更新后的文档,而同时用户刚刚移开另一个覆盖部分工作区的窗口。调用 Invalidate()方法,可以让 Windows 有机会注意到发生的事件。Windows 就会在需要时合并 Paint 事件,合并无效的区域,这样绘图操作就只执行一次。

- 最后，执行绘图的代码可能是应用程序中最复杂的代码部分，特别是当有一个比较复杂的用户界面时，就更是如此。需要长时间维护该代码的人员希望我们把所有的绘图代码都放在一个地方，且尽可能简单——如果程序的其他部分没有过多的路径进入该部分代码，维护就更容易。

其底线是最好把所有的绘图代码都放在 `OnPaint()` 例程中，或者从该方法中调用的其他方法中。但是要维持一个平衡。如果要在屏幕上替换一个字符，最好不要影响到已经绘制好的其他内容，那么此时可能不需要使用 `Invalidate()` 方法(因为系统开销过大)，而只需编写一个独立的绘图例程。



在非常复杂的应用程序中，甚至可以编写一个完整的类，它专门负责在屏幕上绘图。几年前当 MFC 仍是 GDI 密集型应用程序的标准技术时，MFC 就遵循这种模型，使用一个 C++ 类 `C<ApplicationName>View` 完成绘图操作。但即使是这样，这个类也有一个成员函数 `OnDraw()`，它用作大多数绘图请求的入口点。

## 48.15.2 计算项的大小和文档的大小

下面返回 `CapsEditor` 示例，介绍从 `LoadFile()` 方法中调用的 `CalculateLineWidths()` 和 `CalculateDocumentSize()` 方法：



可从  
wrox.com  
下载源代码

```
private void CalculateLineWidths()
{
    Graphics dc = this.CreateGraphics();
    foreach (TextLineInformation nextLine in documentLines)
    {
        nextLine.Width = (uint)dc.MeasureString(nextLine.Text,
            mainFont).Width;
    }
}
```

代码段 `CapsEditor.sln`

这个方法仅遍历已经读取的每行文本，并使用 `Graphics.MeasureString()` 方法计算和存储字符串在屏幕上需要的水平间距。存储这个值是因为 `MeasureString()` 方法要进行大量的计算。如果 `CapsEditor` 示例不够简单，不能很容易地计算出每一行的高度和位置，那么这个方法也肯定需要按计算所有量的方式来实现。

知道屏幕上每一项的大小后，就可以计算每一项的位置，并计算出文档的实际大小。高度是每行文本的高度乘以行数。宽度则需要计算，方法是遍历每行文本，确定哪一行最长。对于高度和宽度，也可以在显示的文档周围设置一个较小的页边距，从而使应用程序看起来更吸引人。

下面是计算文档大小的方法：



可从  
wrox.com  
下载源代码

```
private void CalculateDocumentSize()
{
    if (!documentHasData)
    {
        documentSize = new Size(100, 200);
    }
    else
```

```

    {
        documentSize.Height = (int)(nLines*lineHeight) + 2*(int)margin;
        uint maxLineLength = 0;
        foreach (TextLineInformation nextWord in documentLines)
        {
            uint tempLineLength = nextWord.Width;
            if (tempLineLength > maxLineLength)
            {
                maxLineLength = tempLineLength;
            }
        }
        maxLineLength += 2*margin;
        documentSize.Width = (int)maxLineLength;
    }
    AutoScrollMinSize = documentSize;
}

```

代码段 CapsEditor.sln

这个方法首先检查是否有数据要显示。如果没有，就使用硬编码的文档大小，该大小足以显示很大的红色<Empty Document>警告。如果要正确显示数据，就应使用 MeasureString()方法确定警告信息到底有多大。

计算出文档大小后，就设置 Form.AutoScrollMinSize 属性，以告诉 Form 实例文档有多大。完成后，后台就会发生一些有趣的事。在设置这个属性的过程中，工作区会标记为无效，并引发 Paint 事件。出于非常合理的原因，改变文档的大小就意味着需要添加或修改滚动条，需要重新绘制整个工作区。为什么说这很有趣？如果回过头来看看 LoadFile()方法的代码，就会发现在该方法中调用 Invalidate()方法实际上是多余的。在设置文档大小时，肯定要使工作区无效。在 LoadFile()方法中显式调用 Invalidate()方法，说明了在一般情况下应如何完成任务。实际上，在这个示例中，再次调用 Invalidate()方法将重复请求 Paint 事件，这是不必要的。但是，这依次说明了 Invalidate()方法如何给 Windows 一个优化性能的机会。第二个 Paint 事件实际上并不会被引发：Windows 发现队列中已经有一个 Paint 事件，就会比较请求的无效区域，看看是否需要合并它们。在本例中，因为两个 Paint 事件都指定了整个工作区，所以不需要合并，Windows 会撤销第二个 Paint 请求。当然，这个过程会占用处理器一定的时间，但与某些绘图操作所占用的时间相比，它可以忽略不计。

### 48.15.3 OnPaint()方法

前面介绍了 CapsEditor 如何加载文件，下面看看如何绘图：



可从  
wrox.com  
下载源代码

```

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    int scrollPositionX = AutoScrollPosition.X;
    int scrollPositionY = AutoScrollPosition.Y;
    dc.TranslateTransform(scrollPositionX, scrollPositionY);
    if (!documentHasData)
    {
        dc.DrawString("<Empty document>", emptyDocumentFont,
            emptyDocumentBrush, new Point(20,20));
        base.OnPaint(e);
    }
}

```

```

        return;
    }
    // work out which lines are in clipping rectangle
    int minLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
                                    scrollPositionY);

    if (minLineInClipRegion == -1)
    {
        minLineInClipRegion = 0;
    }
    int maxLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom -
                                    scrollPositionY);

    if (maxLineInClipRegion >= documentLines.Count ||
        maxLineInClipRegion == -1)
    {
        maxLineInClipRegion = documentLines.Count-1;
    }
    TextLineInformation nextLine;
    for (int i=minLineInClipRegion; i<=maxLineInClipRegion; i++)
    {
        nextLine = (TextLineInformation)documentLines[i];
        dc.DrawString(nextLine.Text, mainFont, mainBrush,
                      LineIndexToWorldCoordinates(i));
    }
}

```

代码段 CapsEditor.sln

这个OnPaint()重载方法的核心是一个循环,它迭代文档的每行文本,同时调用Graphics.DrawString()方法绘制每行文本。其余的代码主要是关于优化绘图操作——通常是精确地确定哪些部分需要绘制,而不是告诉图形实例绘制所有内容。

首先检查一下文档中是否有数据。如果没有,就显示一条消息,说明调用基类的 OnPaint()方法并退出。如果有数据,就开始查看剪切的矩形,方法是调用另一个方法 WorldYCoordinateToLineIndex()。后面再介绍这个方法,但本质上该方法带一个相对于文档顶部的 y 坐标,计算此时应显示文档中的哪些文本。

第一次调用 WorldYCoordinateToLineIndex()方法时,给它传递坐标值 e.ClipRectangle.Top - scrollPositionY,这是剪切区域的顶部(把它转换为世界坐标后)。如果返回值为-1,就是安全的,并假定需要从文档的开头开始(如果剪切区域位于顶部边距中,就会出现这种情况)。

完成后,必须对剪切矩形的底部重复相同的过程,找出文档在剪切区域中的最后一行文本。第一行和最后一行的索引分别存储在 minLineInClipRegion 和 maxLineInClipRegion 中。这样就可以在这些值之间运行 for 循环,以执行绘图操作。在绘图循环中,实际上需要通过 WorldYCoordinateToLineIndex()方法粗略地进行逆转换:给出一行文本的索引,要确定该行文本应绘制在什么位置。这个计算实际上相当简单,我们把它封装到另一个方法 LineIndexToWorldCoordinates()中,它返回该行文本左上角的坐标。尽管返回的坐标是世界坐标,但这很好,因为我们已在 Graphics 对象上调用了 TranslateTransform()方法,所以在要求显示文本时,需要给它传递世界坐标,而不是页面坐标。



#### 48.15.4 坐标转换

本节介绍在 CapsEditor 示例中编写的几个辅助方法，以帮助进行坐标转换。这些方法是上一节提到的 WorldYCoordinateToLineIndex()和 LineIndexToWorldCoordinates()方法，还有几个其他的方法。

首先，LineIndexToWorldCoordinates()方法的参数是一个给定的行索引，它使用已知的页边距和行高度，计算出该行左上角的世界坐标：



```
private Point LineIndexToWorldCoordinates(int index)
{
    Point TopLeftCorner = new Point(
        (int)margin, (int)(lineHeight*index + margin));
    return TopLeftCorner;
}
```

代码段 CapsEditor.sln

我们还使用一个方法在 OnPaint()方法中粗略地进行逆转换。WorldYCoordinateToLineIndex()方法可计算出行索引，但它只考虑了一个垂直的世界坐标。这是因为它相对于剪切区域的顶部和底部来计算行索引。

```
private int WorldYCoordinateToLineIndex(int y)
{
    if (y < margin)
    {
        return -1;
    }
    return (int)((y-margin)/lineHeight);
}
```

还有 3 个方法，它们从处理程序例程中调用，这些例程响应用户的双击鼠标操作。首先，用一个方法计算出要在给定世界坐标处显示的文本行的索引。与 WorldYCoordinateToLineIndex()方法不同，这个方法考虑了 x 坐标和 y 坐标，如果在该坐标上没有文本行，它就返回-1：

```
private int WorldCoordinatesToLineIndex(Point position)
{
    if (!documentHasData)
    {
        return -1;
    }
    if (position.Y < margin || position.X < margin)
    {
        return -1;
    }
    int index = (int)(position.Y-margin)/(int)this.lineHeight;
    // check position is not below document
    if (index >= documentLines.Count)
    {
        return -1;
    }
    // now check that horizontal position is within this line
    TextLineInformation theLine =
        (TextLineInformation)documentLines[index];
    if (position.X > margin + theLine.Width)
```

```
        {
            return -1;
        }
        // all is OK. We can return answer
        return index;
    }
```

最后，需要在行索引和页面坐标(而不是世界坐标)之间转换，下面的方法完成这个任务：

```
private Point LineIndexToPageCoordinates(int index)
{
    return LineIndexToWorldCoordinates(index) +
        new Size(AutoScrollPosition);
}
private int PageCoordinatesToLineIndex(Point position)
{
    return WorldCoordinatesToLineIndex(position-new
        Size(AutoScrollPosition));
}
```

注意在转换到页面坐标时，添加了 `AutoScrollPosition`，这是一个负值。

虽然这些方法看起来并不是很有趣，但它们说明了需要经常使用的一个技巧。在 `GDI+`中，系统常常会给出一些特定坐标(如用户在此处单击鼠标的坐标)，而我们需要确定在该坐标处要显示什么内容。或者相反——给出要显示的内容，确定它在什么地方显示。因此，如果编写一个 `GDI+`应用程序，就会发现编写完成等价的坐标转换的方法很有效。

48.15.5 响应用户输入

到目前为止，除了 `CapsEditor` 示例中的 `File` 菜单外，本章介绍的所有内容都是单向的：应用程序告诉用户一些信息，方法是在屏幕上显示它们。当然，几乎所有的软件都是双向的：用户也可以与应用程序进行交互。下面就把这个功能添加到 `CapsEditor` 示例中。

让 `GDI+`应用程序响应用户输入，实际上比编写代码在屏幕上绘图更简单。实际上第 39 章已经介绍了如何处理用户输入。即重写 `Form` 类的方法，这些方法从相关的事件处理程序中调用。在引发 `Paint` 事件时，就是以这种方式调用 `OnPaint()`方法。

当用户单击或移动鼠标时，可以重写的方法如表 48-5 所示。

表 48-5	
方 法	何 时 调 用
<code>OnClick(EventArgs e)</code>	单击鼠标
<code>OnDoubleClick(EventArgs e)</code>	双击鼠标
<code>OnMouseDown(MouseEventArgs e)</code>	按鼠标左键
<code>OnMouseHover(MouseEventArgs e)</code>	鼠标在移动后仍停留在某处
<code>OnMouseMove(MouseEventArgs e)</code>	移动鼠标
<code>OnMouseUp(MouseEventArgs e)</code>	释放鼠标左键

如果要检测用户什么时候输入文本，就可以重写表 48-6 中的方法。

表 48-6

方 法	何 时 调 用
OnKeyDown(KeyEventArgs e)	按下一个键
OnKeyPress(KeyPressEventArgs e)	按下并释放一个键
OnKeyUp(KeyEventArgs e)	释放被按下的键

注意，其中的一些事件是重叠的。例如，如果用户按下鼠标按钮，就会引发 `MouseDown` 事件。如果立即释放按钮，就会引发 `MouseUp` 事件和 `Click` 事件。另外，一些方法接受一个派生自 `EventArgs` 的参数，而不是 `EventArgs` 本身的实例。派生类的这些实例可用于给出特定事件的更多信息。`MouseEventArgs` 有两个属性(X 和 Y)，它们给出鼠标按下时设备的坐标。`KeyEventArgs` 和 `KeyPressEventArgs` 的属性则指定与事件相关的键。


这就是用户响应方法。用户应考虑自己要完成的工作的逻辑。唯一要注意的是，编写 GDI+应用程序可能要比编写 `Windows.Forms` 应用程序做更多的逻辑工作，这是因为在 `Windows.Forms` 应用程序中，一般响应的是比较高级的事件(例如，文本框的 `TextChanged` 事件)。GDI+则相反，其中的事件都比较基本，用户单击鼠标，或按 H 键。应用程序采取的操作取决于一系列事件，而不是单个事件。例如，假定应用程序的工作方式类似于 `Microsoft Word for Windows`：为了选择一些文本，用户通常首先要单击鼠标左键，再移动鼠标，最后释放鼠标左键。虽然应用程序接收到 `MouseDown` 事件，但仅有这个事件是不能完成更多任务，它只是记录下该鼠标单击时光标的位置。那么，当接收到 `MouseDown` 事件时，就需要检查刚才的记录，确定鼠标左键是否被按下，如果按下，突出显示的文本就是用户选择的文本。当用户释放鼠标左键时，对应操作(在 `OnMouseUp()`方法中)就需要检查：按下的鼠标按钮是否有拖动操作，在方法中是否有相应的操作。只有这样，这个序列才算完成。

另外，因为某些事件有重叠，常常要选择希望代码响应哪个事件。

重要规则是仔细考虑用户可能启动的鼠标移动或单击和键盘事件的每个组合的逻辑，确保应用程序以直观的方式响应，使应用程序在各种情况下都按照期望的方式执行。我们的工作大多数是思考，而不是编码，尽管通过编码工作很复杂，因为我们需要考虑用户输入的许多组合。例如，如果用户开始输入文本，而鼠标按钮处于按下状态，应用程序该如何响应？这听起来是不可能的，但最终用户会尝试这么做的！

在 `CapsEditor` 示例中，为了使工作尽可能简单，并没有真正考虑用户输入的任何组合。在本例中只响应用户的双击，此时把光标悬停在那行文本变为大写字母。

这应是一个相当简单的任务，但有一个障碍，需要捕获 `DoubleClick` 事件。但表 48-6 说明，这个事件接受一个 `EventArgs` 参数，不是一个 `MouseEventArgs` 参数。问题是如果要把一行文本正确地标识为大写，我们需要知道用户双击时光标在什么地方，为此又需要一个 `MouseEventArgs` 参数。有两个变通方法，一个方法是使用由 `Form1` 对象 `Control.MousePosition` 实现的一个静态方法，来确定光标的位置：



可从  
wrox.com  
下载源代码

```
protected override void OnDoubleClick(EventArgs e)
{
    Point MouseLocation = Control.MousePosition;
    // handle double click
}
```

代码段 CapsEditor.sln

在大多数情况下,这个方法可以正常工作。但如果应用程序(甚至是其他一些有较高优先级的应用程序)在用户双击时进行某些计算密集型的工作,该方法就会有問題。此时要在用户双击之后大约半秒钟内,才调用 `OnDoubleClick()` 事件处理程序。我们不期望有这样的延迟,因为这会让用户感到很厌烦,但有时因为应用程序不能控制的原因(如计算机较慢),会偶尔发生这种情况。问题是半秒的时间足够光标在屏幕上移动到其他位置了——此时调用 `Control.MousePosition` 会返回完全错误的位置!

比较好的方法是依赖于鼠标事件之间的其中一个重叠。双击鼠标的第一部分涉及按下鼠标左键。这表示如果调用 `OnDoubleClick()` 方法,我们就知道刚调用 `OnMouseDown()` 方法,此时鼠标的位置不变。可以使用 `OnMouseDown()` 方法的重写版本记录光标的位置,为 `OnDoubleClick()` 方法做准备。这就是在 `CapsEditor` 中采用的方法:

```
protected override void OnMouseDown(MouseEventArgs e)
{
    base.OnMouseDown(e);
    mouseDoubleClickPosition = new Point(e.X, e.Y);
}
```

下面查看 `OnDoubleClick()` 方法的重写版本,这里该方法要完成许多工作:



```
protected override void OnDoubleClick(EventArgs e)
{
    int i = PageCoordinatesToLineIndex(mouseDoubleClickPosition);
    if (i >= 0)
    {
        TextLineInformation lineToBeChanged =
            (TextLineInformation)documentLines[i];
        lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
        Graphics dc = this.CreateGraphics();
        uint newWidth = (uint)dc.MeasureString(lineToBeChanged.Text,
            mainFont).Width;

        if (newWidth > lineToBeChanged.Width)
            lineToBeChanged.Width = newWidth;
        if (newWidth + 2 * margin > this.documentSize.Width)
        {
            documentSize.Width = (int)newWidth;
            AutoScrollMinSize = this.documentSize;
        }
        Rectangle changedRectangle = new Rectangle(
            LineIndexToPageCoordinates(i),
            new Size((int)newWidth,
                (int)this.lineHeight));

        Invalidate(changedRectangle);
    }
    base.OnDoubleClick(e);
}
```

代码下载 `CapsEditor.sln`

首先调用 `PageCoordinatesToLineIndex()` 方法计算在用户双击时光标悬停在哪个文本行上。如果该调用返回-1,则光标没有悬停在任何文本行上,因此什么也不做。当然,调用 `OnDoubleClick()` 方法的基础版本,可以让 Windows 执行一些默认操作。

假定已经标识一行文本，那么可以使用 `string.ToUpper()` 方法把它转换为大写。这很容易实现。比较难的部分是确定要在什么地方重绘什么内容。幸运的是，因为这个示例非常简单，所以没有太多的组合。可以假定把文本转换为大写通常要么保持这行文本的宽度不变，要么增大其宽度。因为大写字母比小写字母大，所以宽度不会减小。因为本例没有换行功能，所以文本行不会溢出到下一行上，并使其他文本向下移动。把文本行转换为大写的操作不会改变正在显示的任何其他项的位置，这是一个非常大的简化！

接着代码使用 `Graphics.MeasureString()` 方法计算文本的新宽度。这有两种可能性：

- 新宽度会使该行最长，导致整个文档的宽度增加。如果情况是这样，就需要把 `AutoScrollMinSize` 设置为新值，以便正确地放置滚动条。
- 文档的大小没有变化。

在这两种情况下，都需要调用 `Invalidate()` 方法重绘屏幕。因为只有一行文本改变了，所以不希望重绘整个文档。而需要计算出仅包含被修改的文本行的矩形边界，并把这个矩形传递给 `Invalidate()` 方法，确保只重绘该行文本。这就是以前代码的作用。`Invalidate()` 方法会在鼠标事件处理程序最终返回时调用 `OnPaint()` 方法。本章前面曾提到在 `OnPaint()` 方法中设置断点的困难，如果运行该示例，在 `OnPaint()` 方法中设置断点，捕获绘图操作，就会发现传递给 `OnPaint()` 方法的 `PaintEventArgs` 参数包含一个与指定矩形匹配的剪切区域。因为重载了 `OnPaint()` 方法来仔细考虑剪切区域，所以只重绘要求的文本行。

## 48.16 打印

本章前面主要介绍如何在屏幕上绘图。但有时还需要应用程序生成数据的打印件。这就是本节的主题。我们将扩展 `CapsEditor` 示例，以便它能进行打印预览，并打印正在编辑的文档。

但是，因为本书没有详细讲述这个过程，所以这里实现的打印功能非常基本。通常，如果要实现应用程序的打印数据功能，就要在主 `File` 菜单中为应用程序添加 3 个菜单项：

- **Page Setup**，允许用户选择各种选项，例如，要打印哪一页，要使用什么打印机等。
- **Print Preview**，打开一个新窗体，新窗体显示打印副本的预览外观。
- **Print**，实际打印文档。

在本例中，为了简单起见，不实现 `Page Setup` 菜单项。打印也只使用默认的设置。但要注意，如果要实现 `Page Setup`，Microsoft 已经编写了一个页面设置对话框类，以供使用。这个类是 `System.Windows.Forms.PrintDialog`。一般应编写一个事件处理程序，该事件处理程序显示这个窗体，并保存用户选择的设置。

在许多情况下，打印与在屏幕上显示完全相同：提供一个设备环境(`Graphics` 实例)，对该实例调用所有常见的显示命令。Microsoft 编写了许多类以帮助完成这个工作，我们需要的两个主要的类是 `System.Drawing.Printing.PrintDocument` 和 `System.Drawing.Printing.PrintPreviewDialog`。这两个类可以确保传递给设备环境的绘图命令能得到正确地处理，我们不必担心什么内容应打印到何处的问题。

打印/打印预览和显示在屏幕上有一些重要的区别。打印机不能滚动，但它们输出页面。所以需要确保找到一种合适的方式给文档分页，按要求绘制每一页。其他工作还有计算文档有多少内容可以放在一个页面上，因此计算出需要多少页，文档的每个部分应写到哪个页面上。

尽管上面描述得相当复杂，但打印过程相当简单。从编程的角度来看，需要采取的步骤大致如下：

- **打印**——实例化一个 `PrintDocument` 对象，并调用其 `Print()` 方法。这个方法向 `PrintPage` 事件发出打印第一个页面的信号。`PrintPage` 事件接受一个 `PrintPageEventArgs` 参数，该参数提供页面大小和页面设置的信息，以及用于绘图命令的 `Graphics` 对象。因此应为这个事件编写一个事件处理程序，并实现该处理程序，以打印页面。这个事件处理程序还应把 `PrintPageEventArgs` 参数的布尔属性 `HasMorePages` 设置为 `true` 或 `false`，表示是否还有要打印的页面。`PrintDocument.Print()` 方法将重复引发 `PrintPage` 事件，直到把 `HasMorePages` 属性设置为 `false` 为止。
- **打印预览**——在这种情况下，实例化 `PrintDocument` 对象和 `PrintPreviewDialog` 对象。使用 `PrintPreviewDialog.Document` 属性把 `PrintDocument` 关联到 `PrintPreviewDialog` 上，然后调用对话框的 `ShowDialog()` 方法。这个方法会模拟地显示对话框，使之呈现为 Windows 标准打印预览窗体，打印预览窗体显示文档的页面。在内部，则重复引发 `PrintPage` 事件，多次显示页面，直到把 `HasMorePages` 属性为 `false` 为止。不需为此编写一个事件处理程序；而可以使用打印每个页面时使用的同一个事件处理程序，因为绘图代码在这两种情况下应完全相同(毕竟，打印预览的内容应与打印出来的版本看起来完全相同)。

## 实现打印和打印预览

既然简要概述了该过程，本节介绍如何在代码中完成这些步骤。可以从 [www.wrox.com](http://www.wrox.com) 或随书附赠光盘上找到 `PrintingCapsEdit` 项目，它包含 `CapsEditor` 项目，以及下述修改。

首先使用 Visual Studio 2010 设计视图在 `File` 菜单中添加两个新菜单项：`Print` 和 `Print Preview`。再使用 `Properties` 窗口把这两个项命名为 `menuFilePrint` 和 `menuFilePrintPreview`，把它们设置为应用程序启动时是禁用的(只有打开文档，才能进行打印)。在主窗体的 `LoadFile()` 方法中添加如下代码，启用这两个菜单项，`LoadFile()` 方法负责把文件加载到 `CapsEditor` 应用程序中：



```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
    if (nLines > 0)
    {
        documentHasData = true;
        menuFilePrint.Enabled = true;
        menuFilePrintPreview.Enabled = true;
    }
    else
    {
        documentHasData = false;
        menuFilePrint.Enabled = false;
        menuFilePrintPreview.Enabled = false;
    }
}
```

```

    }
    CalculateLineWidths();
    CalculateDocumentSize();
    Text = standardTitle + "-" + FileName;
    Invalidate();
}

```

---

代码下载 [Printing.sln](#)

上面突出显示的代码是添加到这个方法中的新代码。接着，给 `Form1` 类添加一个成员字段：

```

public partial class Form1: Form
{
    private int pagesPrinted = 0;
}

```

这个字段用于指定目前正在打印的页面。使它成为一个成员字段，因为需要在 `PrintPage` 事件处理程序的调用之间记忆上述信息。

接着是用户选择 **Print** 或 **Print Preview** 菜单项时执行的事件处理程序：



```

private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintPreviewDialog ppd = new PrintPreviewDialog();
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += this.pd_PrintPage;
    ppd.Document = pd;
    ppd.ShowDialog();
}
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += new PrintPageEventHandler
        (this.pd_PrintPage);
    pd.Print();
}

```

---

代码下载 [Printing.sln](#)

前面解释了涉及打印的主要过程，可以看出，这些事件处理程序仅实现了这个过程。在打印和打印预览中，都实例化一个 `PrintDocument` 对象，并把一个事件处理程序附加到该对象的 `PrintPage` 事件中。对于打印，应调用 `PrintDocument.Print()` 方法；而对于打印预览，则把 `PrintDocument` 对象附加到 `PrintPreviewDialog` 中，并调用打印预览对话框对象的 `ShowDialog()` 方法。实际工作将在 `PrintPage` 事件的处理程序中完成。下面是该处理程序的代码：



```

private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
    float yPos = 0;
    float leftMargin = e.MarginBounds.Left;
    float topMargin = e.MarginBounds.Top;
    string line = null;
    // Calculate the number of lines per page.
    int linesPerPage = (int)(e.MarginBounds.Height /
        mainFont.GetHeight(e.Graphics));
}

```

```

int lineNo = pagesPrinted * linesPerPage;
// Print each line of the file.
int count = 0;
while(count < linesPerPage && lineNo < this.nLines)
{
    line = ((TextLineInformation)this.documentLines[lineNo]).Text;
    yPos = topMargin + (count * mainFont.GetHeight(e.Graphics));
    e.Graphics.DrawString(line, mainFont, Brushes.Blue,
        leftMargin, yPos, new StringFormat());
    lineNo++;
    count++;
}
// If more lines exist, print another page.
if(this.nLines > lineNo)
    e.HasMorePages = true;
else
    e.HasMorePages = false;
pagesPrinted++;
}

```

代码下载 [Printing.sln](#)

在声明了两个局部变量后，应先计算出一个页面上可以显示多少行文本——即页面的高度除以一文本行的高度，并向下舍入。页面的高度可以从 `PrintPageEventArgs.MarginBounds` 属性获得，这个属性是一个 `RectangleF` 结构，初始化该结构来给出页面的边界。一行文本的高度可以从 `Form1.mainFont` 字段中获得，该字段是显示文本所使用的字体。这里也必须使用相同的字体进行打印。注意，对于 `PrintingCapsEditor` 示例，因为每页的行数总相同，所以可以在第一次计算出该行数后，把它缓存起来。但是，该计算并不困难，在比较复杂的应用程序中，因为这个值可能会有变化，所以每次打印页面时重新计算它也是可以的。

我们还初始化一个 `lineNo` 变量。这给出页面第 1 行对应的文档的文本行索引(该索引从 0 开始)。这条信息非常重要，因为在原则上，本可以调用 `pd_PrintPage()` 方法打印任何页面，而不仅仅是打印第一页。`lineNo` 的值是每个页面的行数和已打印的页数的乘积。

接着运行一个循环，打印每一行文本。当打印完文档中的所有文本行，或者打印完本页面上的所有文本行时，这个循环就终止(只要满足这两个条件之一即可)。最后，检查是否还有要打印的文档，并据此设置 `PrintPageEventArgs` 参数的 `HasMorePages` 属性。同时递增 `pagesPrinted` 字段，这样下一次调用 `PrintPage` 事件处理程序时，就会打印正确的页面。

这个事件处理程序要注意的一点是不必担心绘图命令的发送目的地。我们使用所提供的 `Graphics` 对象和 `PrintPageEventArgs` 参数。Microsoft 编写的 `PrintDocument` 类将在内部确保，如果正在打印，`Graphics` 对象就与打印机关联起来；如果正在打印预览，`Graphics` 对象就与屏幕上的打印预览窗体关联起来。

最后，需要确保为类型定义搜索 `System.Drawing.Printing` 名称空间：

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;

```



```
using System.Windows.Forms;
using System.IO;
```

剩下的内容全部是编译项目，并检查代码的工作情况。如果运行 CapsEdit，加载一个文本文档(与以前一样，为该项目选择 C#源文件)，并选择 Print Preview，就可以显示出该文档的打印预览版本，如图 48-19 所示。

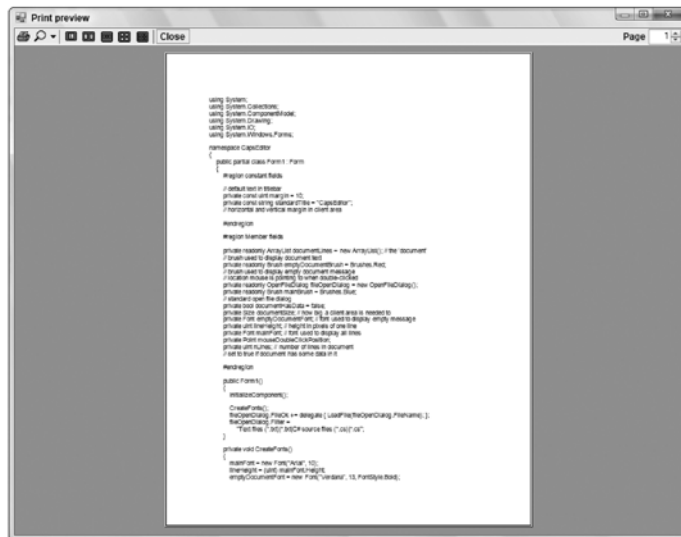


图 48-19

在图 48-19 上，滚动到文档的第 5 页上，把预览设置为显示正常的尺寸。PrintPreviewDialog 提供了许多功能，这可以从窗体顶部的工具栏中看出来。可用的选项包括打印文档、缩小和放大文档、一次显示 2、3、4 或 6 页。这些选项的功能都很完整，不需要我们做任何工作。例如，如果把缩放改为自动，单击显示 4 页的选项(右数第 3 个工具栏)，就会得到如图 48-20 所示的结果。

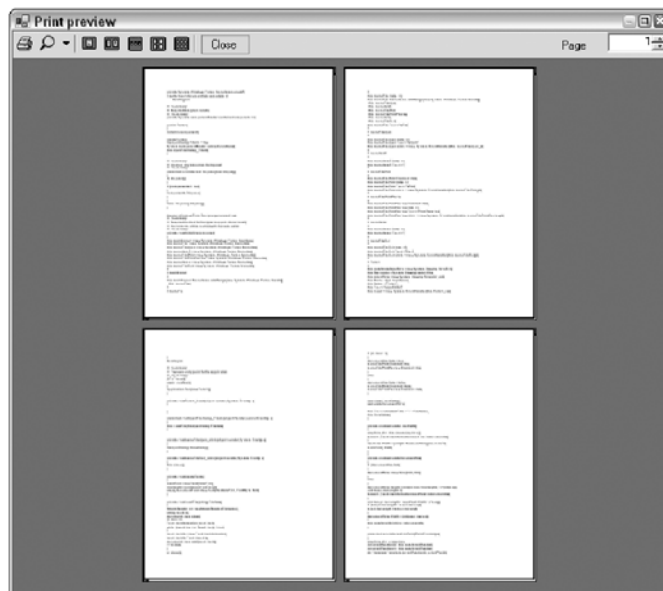


图 48-20

## 48.17 小结

本章介绍了如何在显示设备上绘图操作，其中绘图操作通过代码实现，而不是由一些预定义的控件或对话框实现，这就是 GDI+ 的实质。GDI+ 是一个功能强大的工具，有许多 .NET 基类都可用于在设备上绘图。绘图过程实际上相对简单。在大多数情况下，只使用几条 C# 语句，就可以绘制文本和专业的图形或显示图像。但是，管理绘图操作——后台的工作涉及计算出要绘制的内容、确定在什么地方绘制它，以及在任何给定情况下，哪些内容需要重绘，哪些内容不需要重绘。这些工作都非常复杂，需要经过仔细的算法设计。因此，很好地理解 GDI+ 的工作方式，以及 Windows 采取什么操作来完成工作也非常重要。特别是，由于 Windows 的体系结构，在绘图过程中，应使窗口的区域失效，并依赖 Windows 通过引发 Paint 事件来响应。

本章并没有介绍绘图操作所涉及的其他 .NET 类。然而，如果您理解了绘图操作的规则，就可以通过查看 SDK 文档中这些类的方法列表，实例化它们的实例，看看它们能做什么，以掌握它们。最后，如果要超越标准控件的功能，那么绘图与编程的任何其他方面一样，也需要逻辑、仔细的思考和清晰的算法。如果软件设计得好，它就有助于改善用户友好性和视觉外观。许多应用程序完全依赖于控件来设计其用户界面。这很有效，但这种应用程序看起来非常类似。通过添加一些 GDI+ 代码完成某些自定义绘图操作，可以使软件与众不同，显得比较新颖——这只会有助于软件的销售！