**使用C#进行图像处理的几种方法**

**文章目录**

1. [1. Bitmap类](http://kuangqi.me/image-processing/image-processing-using-c-sharp/#bitmap类)
2. [2. BitmapData类](http://kuangqi.me/image-processing/image-processing-using-c-sharp/#bitmapdata类)
3. [3. unsafe代码](http://kuangqi.me/image-processing/image-processing-using-c-sharp/#unsafe代码)
4. [4. 字节对齐问题](http://kuangqi.me/image-processing/image-processing-using-c-sharp/#字节对齐问题)

本文讨论了C#图像处理中Bitmap类、BitmapData类和unsafe代码的使用以及字节对齐问题。

**Bitmap类**

命名空间：System.Drawing

封装GDI+位图，此位图由图形图像及其属性的像素数据组成。Bitmap是用于处理由像素数据定义的图像的对象。

利用C#类进行图像处理,最方便的是使用Bitmap类，使用该类的GetPixel()与SetPixel()来访问图像的每个像素点。下面是MSDN中的示例代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public void GetPixel\_Example(PaintEventArgs e)  {  // Create a Bitmap object from an image file.  Bitmap myBitmap = new Bitmap("Grapes.jpg");  // Get the color of a pixel within myBitmap.  Color pixelColor = myBitmap.GetPixel(50, 50);  // Fill a rectangle with pixelColor.  SolidBrush pixelBrush = new SolidBrush(pixelColor);  e.Graphics.FillRectangle(pixelBrush, 0, 0, 100, 100);  } |

可见，Bitmap类使用一种优雅的方式来操作图像，但是带来的性能的降低却是不可忽略的。比如对一个800\*600的彩色图像灰度化，其耗费的时间都要以秒为单位来计算。在实际项目中进行图像处理，这种速度是决对不可忍受的。

**BitmapData类**

命名空间：System.Drawing.Imaging

指定位图图像的属性。BitmapData类由Bitmap类的LockBits和 UnlockBits方法使用。不可继承。

好在我们还有BitmapData类，通过BitmapData LockBits()可将 Bitmap 锁定到系统内存中。该类的公共属性有：

* Width：获取或设置Bitmap对象的像素宽度。这也可以看作是一个扫描行中的像素数。
* Height：获取或设置Bitmap对象的像素高度。有时也称作扫描行数。
* PixelFormat：获取或设置返回此 BitmapData对象的Bitmap对象中像素信息的格式。
* Scan0：获取或设置位图中第一个像素数据的地址。它也可以看成是位图中的第一个扫描行。
* Stride：获取或设置Bitmap对象的跨距宽度（也称为扫描宽度）。

下面的MSDN中的示例代码演示了如何使用 PixelFormat、Height、Width 和Scan0属性；LockBits和 UnlockBits方法；以及ImageLockMode枚举。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | private void LockUnlockBitsExample(PaintEventArgs e)  {  // Create a new bitmap.  Bitmap bmp = new Bitmap("c:\\fakePhoto.jpg");  // Lock the bitmap's bits.  Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);  System.Drawing.Imaging.BitmapData bmpData =  bmp.LockBits(rect, System.Drawing.Imaging.ImageLockMode.ReadWrite,  bmp.PixelFormat);  // Get the address of the first line.  IntPtr ptr = bmpData.Scan0;  // Declare an array to hold the bytes of the bitmap.  // This code is specific to a bitmap with 24 bits per pixels.  int bytes = bmp.Width \* bmp.Height \* 3;  byte[] rgbValues = new byte[bytes];  // Copy the RGB values into the array.  System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);  // Set every red value to 255.  for (int counter = 0; counter < rgbValues.Length; counter+=3)  rgbValues[counter] = 255;  // Copy the RGB values back to the bitmap  System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, ptr, bytes);  // Unlock the bits.  bmp.UnlockBits(bmpData);  // Draw the modified image.  e.Graphics.DrawImage(bmp, 0, 150);  } |

上面的代码演示了如何用数组的方式来访问一幅图像，而不在使用低效的GetPixel()和SetPixel()。

**unsafe代码**

而在实际中上面的做法仍然不能满足我们的要求，图像处理是一种运算量比较大的操作，不同于我们写的一般的应用程序。我们需要的是一种性能可以同C++程序相媲美的图像处理程序。C++是怎么提高效率的呢，答曰：指针。幸运的是.Net也允许我们使用指针，只能在非安全代码块中使用指针。何谓非安全代码？

为了保持类型安全，默认情况下，C#不支持指针运算。不过，通过使用unsafe关键字，可以定义可使用指针的不安全上下文。在公共语言运行库（CLR）中，不安全代码是指无法验证的代码。C#中的不安全代码不一定是危险的，只是其安全性无法由CLR进行验证的代码。因此，CLR只对在完全受信任的程序集中的不安全代码执行操作。如果使用不安全代码，由您负责确保您的代码不会引起安全风险或指针错误。不安全代码具有下列属性：

方法、类型和可被定义为不安全的代码块。 在某些情况下，通过移除数组界限检查，不安全代码可提高应用程序的性能。当调用需要指针的本机函数时，需要使用不安全代码。使用不安全代码将引起安全风险和稳定性风险。在C#中，为了编译不安全代码，必须用/unsafe编译应用程序。

正如《C#语言规范》中所说无论从开发人员还是从用户角度来看，不安全代码事实上都是一种“安全”功能。不安全代码必须用修饰符unsafe明确地标记，这样开发人员就不会误用不安全功能，而执行引擎将确保不会在不受信任的环境中执行不安全代码。

以下代码演示如何借助BitmapData类采用指针的方式来遍历一幅图像，这里的unsafe代码块中的代码就是非安全代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //创建图像  Bitmap image = new Bitmap("C:\\images\\image.gif");  //获取图像的BitmapData对像  BitmapData data = image.LockBits(new Rectangle(0, 0, image.Width, image.Height), ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);  //循环处理  unsafe  {  byte\* ptr = (byte\*)(data.Scan0);  for(int i = 0; i < data.Height ; i++)  {  for(int j = 0; j < data.Width; j++)  {  // write the logic implementation here  ptr += 3;  }  ptr += data.Stride - data.Width \* 3;  }  } |

毫无疑问，采用这种方式是最快的，所以在实际工程中都是采用指针的方式来访问图像像素的。

**字节对齐问题**

上例中ptr += data.Stride - data.Width \* 3，表示跨过无用的区域，其原因是图像数据在内存中存储时是按4字节对齐的，具体解释如下：

假设有一张图片宽度为6，假设是Format24bppRgb格式的(每像素3字节，在以下的讨论中，除非特别说明，否则Bitmap都被认为是24位RGB)。显然，每一行需要6\*3=18个字节存储。对于Bitmap就是如此。但对于BitmapData，虽然data.Width还是等于image.Width，但大概是出于显示性能的考虑，每行的实际的字节数将变成大于等于它的那个离它最近的4的整倍数，此时的实际字节数就是Stride。就此例而言，18不是4的整倍数，而比18大的离18最近的4的倍数是20，所以这个data.Stride = 20。显然，当宽度本身就是4的倍数时，data.Stride = image.Width \* 3。

画个图可能更好理解。R、G、B分别代表3个原色分量字节，BGR就表示一个像素。为了看起来方便我在们每个像素之间插了个空格，实际上是没有的。X表示补足4的倍数而自动插入的字节。为了符合人类的阅读习惯我分行了，其实在计算机内存中应该看成连续的一大段。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | |---------Stride----------|  |---------Width--------| |  Scan0：  BGR BGR BGR BGR BGR BGR XX  BGR BGR BGR BGR BGR BGR XX  BGR BGR BGR BGR BGR BGR XX  ... |

首先用data.Scan0找到第0个像素的第0个分量的地址，这个地址指向的是个byte类型，所以当时定义为byte\* ptr。行扫描时，在当前指针位置（不妨看成当前像素的第0个颜色分量）连续取出三个值（3个原色分量。注意，0、1、2代表的次序是B、G、R。在取指针指向的值时，貌似p[n]和p += n再取p[0]是等价的），然后下移3个位置（ptr += 3，看成指到下一个像素的第0个颜色分量）。做过Bitmap.Width次操作后，就到达了Bitmap.Width \* 3的位置，应该要跳过图中标记为X的字节了（共有Stride - Width \* 3个字节），代码中就是ptr += dataIn.Stride - dataIn.Width \* 3。

通过阅读本文，相信你已经对使用C#进行图像处理可能用到的几种方法有了一个了解。至于采用哪种方式，取决于你的性能要求。其中第一种方式最优雅；第三种方式最快，但不是安全代码；第二种方式取了个折中，保证是安全代码的同时又提高了效率。熟悉C/C++编程的人可能会比较偏向于第三种方式，我个人也比较喜欢第三种方式。

来源： <<http://kuangqi.me/image-processing/image-processing-using-c-sharp/>>