//C#文件流写文件,默认追加FileMode.Append   
            string msg = "okffffffffffffffff";  
            byte[] myByte = System.Text.Encoding.UTF8.GetBytes(msg);  
            using (FileStream fsWrite = new FileStream(@"D:\1.txt", FileMode.Append))  
            {  
                fsWrite.Write(myByte, 0, myByte.Length);  
            };   
            //c#文件流读文件   
            using (FileStream fsRead = new FileStream(@"D:\1.txt", FileMode.Open))  
            {  
                int fsLen = (int)fsRead.Length;  
                byte[] heByte = new byte[fsLen];  
                int r = fsRead.Read(heByte, 0, heByte.Length);  
                string myStr = System.Text.Encoding.UTF8.GetString(heByte);  
                Console.WriteLine(myStr);  
                Console.ReadKey();  
            }

FileStream对象表示在磁盘或网络路径上指向文件的流。这个类提供了在文件中读写字节的方法，但经常使用StreamReader或StreamWriter执行这些功能。这是因为FileStream类操作的是字节和字节数组，而Stream类操作的是字符数据。字符数据易于使用，但是有些操作，比如随机文件访问(访问文件中间某点的数据)，就必须由FileStream对象执行，稍后对此进行介绍。

还有几种方法可以创建FileStream对象。构造函数具有许多不同的重载版本，最简单的构造函数仅仅带有两个参数，即文件名和FileMode枚举值。

|  |
| --- |
| FileStream aFile = new FileStream(filename, FileMode.Member); |

FileMode枚举有几个成员，规定了如何打开或创建文件。稍后介绍这些枚举成员。另一个常用的构造函数如下：

|  |
| --- |
| FileStream aFile = new FileStream(filename, FileMode.Member, FileAccess. Member); |

第三个参数是FileAccess枚举的一个成员，它指定了流的作用。FileAccess枚举的成员如表22-6所示。

表  22-6

|  |  |
| --- | --- |
| 成   员 | 说    明 |
| Read | 打开文件，用于只读 |
| Write | 打开文件，用于只写 |
| ReadWrite | 打开文件，用于读写 |

对文件进行不是FileAccess枚举成员指定的操作会导致抛出异常。此属性的作用是，基于用户的身份验证级别改变用户对文件的访问权限。

在FileStream构造函数不使用FileAccess枚举参数的版本中，使用默认值FileAccess. ReadWrite。

FileMode枚举成员如表22-7所示。使用每个值会发生什么，取决于指定的文件名是否表示已有的文件。注意这个表中的项表示创建流时该流指向文件中的位置，下一节将详细讨论这个主题。除非特别说明，否则流就指向文件的开头。

表  22-7

|  |  |  |
| --- | --- | --- |
| 成    员 | 文 件 存 在 | 文件不存在 |
| Append | 打开文件，流指向文件的末尾，只能与枚举FileAccess.Write联合使用 | 创建一个新文件。只能与枚举FileAccess.Write联合使用 |
| Create | 删除该文件，然后创建新文件 | 创建新文件 |
| CreateNew | 抛出异常 | 创建新文件 |
| Open | 打开现有的文件，流指向文件的开头 | 抛出异常 |
| OpenOrCreate | 打开文件，流指向文件的开头 | 创建新文件 |
| Truncate | 打开现有文件，清除其内容。流指向文件的开头，保留文件的初始创建日期 | 抛出异常 |

File和FileInfo类都提供了OpenRead()和OpenWrite()方法，更易于创建FileStream对象。前者打开了只读访问的文件，后者只允许写入文件。这些都提供了快捷方式，因此不必以FileStream构造函数的参数形式提供前面所有的信息。例如，下面的代码行打开了用于只读访问的Data.txt文件：

|  |
| --- |
| FileStream aFile = File.OpenRead("Data.txt"); |

注意下面的代码执行同样的功能：

|  |
| --- |
| FileInfo aFileInfo = new FileInfo("Data.txt");  FileStream aFile = aFile.OpenRead(); |

**1. 文件位置**

FileStream类维护内部文件指针，该指针指向文件中进行下一次读写操作的位置。在大多数情况下，当打开文件时，它就指向文件的开始位置，但是此指针可以修改。这允许应用程序在文件的任何位置读写，随机访问文件，或直接跳到文件的特定位置上。当处理大型文件时，这非常省时，因为马上可以定位到正确的位置。

实现此功能的方法是Seek()方法，它有两个参数：第一个参数规定文件指针以字节为单位的移动距离。第二个参数规定开始计算的起始位置，用SeekOrigin枚举的一个值表示。Seek Origin枚举包含3个值：Begin、Current和End。

例如，下面的代码行将文件指针移动到文件的第8个字节，其起始位置就是文件的第1个字节：

|  |
| --- |
| aFile.Seek(8,SeekOrigin.Begin); |

下面的代码行将指针从当前位置开始向前移动2个字节。如果在上面的代码行之后执行下面的代码，文件指针就指向文件的第10个字节：

|  |
| --- |
| aFile.Seek(2,SeekOrigin.Current); |

注意读写文件时，文件指针也会改变。在读取了10个字节之后，文件指针就指向被读取的第10个字节之后的字节。

也可以规定负查找位置，这可以与SeekOrigin.End枚举值一起使用，查找靠近文件末端的位置。下面的代码会查找文件中倒数第5个字节：

|  |
| --- |
| aFile.Seek(–5, SeekOrigin.End); |

以这种方式访问的文件有时称为随机访问文件，因为应用程序可以访问文件中的任何位置。稍后介绍的Stream类可以连续地访问文件，不允许以这种方式操作文件指针。

**2. 读取数据**

使用FileStream类读取数据不像使用本章后面介绍的StreamReader类读取数据那样容易。这是因为FileStream类只能处理原始字节(raw byte)。处理原始字节的功能使FileStream类可以用于任何数据文件，而不仅仅是文本文件。通过读取字节数据，FileStream对象可以用于读取图像和声音的文件。这种灵活性的代价是，不能使用FileStream类将数据直接读入字符串，而使用StreamReader类却可以这样处理。但是有几种转换类可以很容易地将字节数组转换为字符数组，或者进行相反的操作。

FileStream.Read()方法是从FileStream对象所指向的文件中访问数据的主要手段。这个方法从文件中读取数据，再把数据写入一个字节数组。它有三个参数：第一个参数是传输进来的字节数组，用以接受FileStream对象中的数据。第二个参数是字节数组中开始写入数据的位置。它通常是0，表示从数组开端向文件中写入数据。最后一个参数指定从文件中读出多少字节。

下面的示例演示了从随机访问文件中读取数据。要读取的文件实际是为此示例创建的类文件。

**试试看：从随机访问文件中读取数据**

(1) 在目录C:\BegVCSharp\Chapter22下创建一个新的控制台应用程序ReadFile。

(2) 在Program.cs文件的顶部添加下面的using指令：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Text;  using System.IO; |

(3) 在Main()方法中添加下面的代码：

|  |
| --- |
| static void Main(string[] args)  {  byte[] byData = new byte[200];  char[] charData = new Char[200];  try  {  FileStream aFile = new FileStream("http://www.cnblogs.com/Program.cs",FileMode.Open);  aFile.Seek(135,SeekOrigin.Begin);  aFile.Read(byData,0,200);  }  catch(IOException e)  {  Console.WriteLine("An IO exception has been thrown!");  Console.WriteLine(e.ToString());  Console.ReadKey();  return;  }  Decoder d = Encoding.UTF8.GetDecoder();  d.GetChars(byData, 0, byData.Length, charData, 0);  Console.WriteLine(charData);  Console.ReadKey();  } |

(4) 运行应用程序。结果如图22-2所示。

|  |
| --- |
|  |
| 图 22-2 |

示例的说明  
此应用程序打开自己的.cs文件，用于读取。它在下面的代码行中使用..字符串向上逐级导航两个目录，找到该文件：

|  |
| --- |
| FileStream aFile = new FileStream("http://www.cnblogs.com/Program.cs",FileMode.Open); |

下面两行代码实现查找工作，并从文件的具体位置读取字节：

|  |
| --- |
| aFile.Seek(135,SeekOrigin.Begin);  aFile.Read(byData,0,200); |

第一行代码将文件指针移动到文件的第135个字节。在Program.cs中，这是namespace的 “n”；其前面的135个字符是using指令和相关的#region。第二行将接下来的200个字节读入到byData字节数组中。

注意这两行代码封装在try…catch块中，以处理可能抛出的异常。

|  |
| --- |
| try  {  aFile.Seek(135,SeekOrigin.Begin);  aFile.Read(byData,0,100);  }  catch(IOException e)  {  Console.WriteLine("An IO exception has been thrown!");  Console.WriteLine(e.ToString());  Console.ReadKey();  return;  } |

文件IO涉及到的所有操作都可以抛出类型为IOException的异常。所有产品代码都必须包含错误处理，尤其是处理文件系统时更是如此。本章的所有示例都具有错误处理的基本形式。

从文件中获取了字节数组后，就需要将其转换为字符数组，以便在控制台显示它。为此，使用System.Text命名空间的Decoder类。此类用于将原始字节转换为更有用的项，比如字符：

|  |
| --- |
| Decoder d = Encoding.UTF8.GetDecoder();  d.GetChars(byData, 0, byData.Length, charData, 0); |

这些代码基于UTF8编码模式创建了Decoder对象。这就是Unicode编码模式。然后调用GetChars()方法，此方法提取字节数组，将它转换为字符数组。完成之后，就可以将字符数组输出到控制台。

**3. 写入数据**

向随机访问文件中写入数据的过程与从中读取数据非常类似。首先需要创建一个字节数组；最简单的办法是首先构建要写入文件的字符数组。然后使用Encoder对象将其转换为字节数组，其用法非常类似于Decoder。最后调用Write()方法，将字节数组传送到文件中。

下面构建一个简单的示例演示其过程。

**试试看：将数据写入随机访问文件**

(1) 在C:\BegVCSharp\Chapter22目录下创建一个新的控制台应用程序WriteFile。

(2) 如上所示，在Program.cs文件顶部添加下面的using指令：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Text;  using System.IO; |

(3) 在Main()方法中添加下面的代码：

|  |
| --- |
| static void Main(string[] args)  {  byte[] byData;  char[] charData;  try  {  FileStream aFile = new FileStream("Temp.txt", FileMode.Create);  charData = "My pink half of the drainpipe.".ToCharArray();  byData = new byte[charData.Length];  Encoder e = Encoding.UTF8.GetEncoder();  e.GetBytes(charData, 0, charData.Length, byData, 0, true);  // Move file pointer to beginning of file.  aFile.Seek(0, SeekOrigin.Begin);  aFile.Write(byData, 0, byData.Length);  }  catch (IOException ex)  {  Console.WriteLine("An IO exception has been thrown!");  Console.WriteLine(ex.ToString());  Console.ReadKey();  return;  }  } |

(4) 运行该应用程序。稍后就将其关闭。

(5) 导航到应用程序目录 —— 在目录中已经保存了文件，因为我们使用了相对路径。目录位于WriteFile\bin\Debug文件夹。打开Temp.txt文件。可以在文件中看到如图22-3所示的文本。

|  |
| --- |
|  |
| 图 22-3 |

示例的说明  
此应用程序在自己的目录中打开文件，并在文件中写入了一个简单的字符串。在结构上这个示例非常类似于前面的示例，只是用Write()代替了Read()，用Encoder代替了Decoder。

下面的代码行使用String类的ToCharArray()静态方法，创建了字符数组。因为C#中的所有事物都是对象，文本“My pink half of the drainpipe.”实际上是一个String对象，所以甚至可以在字符串上调用这些静态方法。

|  |
| --- |
| CharData = " My pink half of the drainpipe. ".ToCharArray(); |

下面的代码行显示了如何将字符数组转换为FileStream对象需要的正确字节数组。

|  |
| --- |
| Encoder e = Endoding.UTF8.GetEncoder();  e.GetBytes(charData,0,charData.Length, byData,0,true); |

这次，要基于UTF8编码方法来创建Encoder对象。也可以将Unicode用于解码。这里在写入流之前，需要将字符数据编码为正确的字节格式。在GetBytes()方法中可以完成这些工作，它可以将字符数组转换为字节数组，并将字符数组作为第一个参数(本例中的charData)，将该数组中起始位置的下标作为第二个参数(0表示数组的开头)。第三个参数是要转换的字符数量(charData.Length，charData数组中的元素个数)。第四个参数是在其中置入数据的字节数组(byData)，第五个参数是在字节数组中开始写入位置的下标(0表示byData数组的开头)。

最后一个参数决定在结束后Encoder对象是否应该更新其状态，即Encoder对象是否仍然保留它原来在字节数组中的内存位置。这有助于以后调用Encoder对象，但是当只进行单一调用时，这就没有什么意义。最后对Encoder的调用必须将此参数设置为true，以清空其内存，释放对象，用于垃圾回收。

之后，使用Write()方法向FileStream写入字节数组就非常简单：

|  |
| --- |
| aFile.Seek(0,SeekOrigin.Begin);  aFile.Write(byData,0,byData.Length); |

与Read()方法一样，Write()方法也有三个参数：要写入的数组，开始写入的数组下标和要写入的字节数。

https://www.cnblogs.com/vaevvaev/p/6804852.html