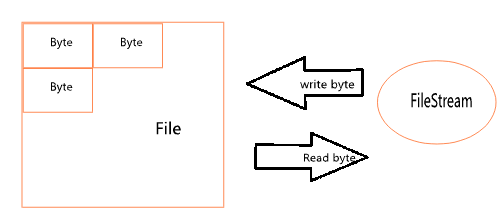
C# 温故而知新：Stream篇（四）FileStream

如何去理解FileStream?

通过前3章的学习相信大家对于Stream已经有一定的了解，但是又如何去理解FileStream呢？请看下图



我们磁盘的中任何文件都是通过2进制组成，最为直观的便是记事本了，当我们新建一个记事本时，它的大小是0KB, 我们每次输入一个数字或字母时文件便会自动增大4kb,可见随着我们输入的内容越来越多，文件也会相应增大，同理当我们删除文件内容时，文件也会相应减小，对了，聪明的你肯定会问：谁将内容以怎么样的形式放到文件中去了？好问题，还记得第一篇流的概念么？对了，真实世界的一群鱼可以通过河流来往于各个地方，FileStream也是一样，byte可以通过FileStream进行传输，这样我们便能在计算机上对任何文件进行一系列的操作了。

FileStream 的重要性

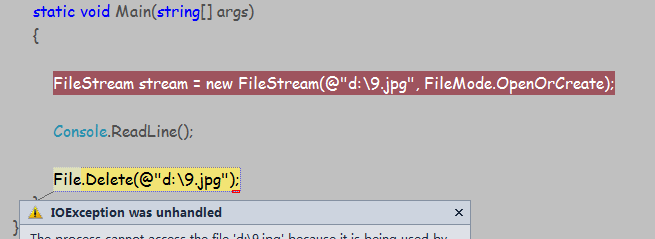
FileStream 顾名思义文件流，我们电脑上的文件都可以通过文件流进行操作，例如文件的复制，剪切，粘贴，删除, 本地文件上传，下载,等许多重要的功能都离不开文件流,所以文件流不仅在本机上非常重要，在如今的网络世界也是万万不能缺少的，想象一下我们开启虚机后，直接从本地复制一个文件到虚机上，是多么方便，如果没有文件流，这个将难以想象。（大家别误解，文件流无法直接通过网络进行传输，而是通过网络流将客户端上传的文件传到服务器端接收，然后通过文件流进行处理，下载正好相反）

FileStream 常用构造函数介绍（可能理解上有点复杂，请大家务必深刻理解）

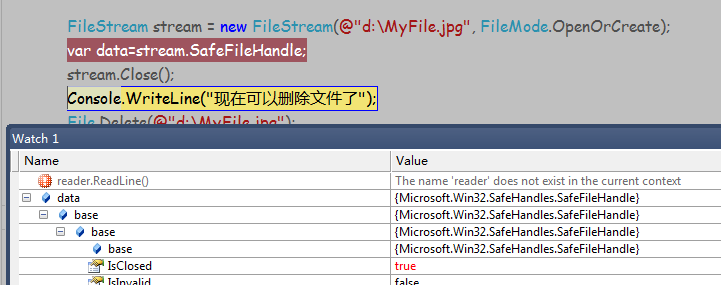
 \*1: FileStream(SafeFileHandle, FileAccess)

非托管参数SafeFileHandle简单介绍

SafeFileHandle :是一个文件安全句柄，这样的解释可能大家一头雾水，别急，大家先不要去理睬这深邃的含义，只要知道这个类型是c#非托管资源，  也就是说它能够调用非托管资源的方法，而且不属于c#回收机制，所以我们必须使用GC手动或其他方式（Finalize 或Dispose方法）进行非托管资源的回收，所以SafeFileHandle是个默默无闻的保镖 ，一直暗中保护FileStream和文件的安全  为了能让大家更好的理解这个保镖，请看第一段代码：



会什么会报错呢？其实程序被卡在 Console.ReadLine()这里，FileStream并没有  被释放，系统不知道这个文件是否还有用﹐所以帮我们保护这个文件  (那个非托管资源SafeFileHandle所使用的内存还被程序占用着)  所以SafeFileHandled 在内部保护了这个文件从而报出了这个异常 如果我们将流关闭后，这个问题也就不存在了



可以看见stream.SafeFileHandle的IsClose属性变成true了，也就是说这时候可以安全的删除文件了所以又回到了一个老问题上面，我们每次使用完FileStream后都必须将他关闭并释放资源

\*2: FileStream(String, FileMode)

String 参数表示文件所在的地址,FIleMode是个枚举，表示确定如何打开或创建文件。

FileMode枚举参数包含以下内容:

|  |  |
| --- | --- |
| 成员名称 | 说明 |
| Append | 打开现有文件并查找到文件尾，或创建新文件。FileMode.Append 只能同 FileAccess.Write 一起使用。 |
| Create | 指定操作系统应创建新文件。如果文件已存在，它将被改写。这要求 FileIOPermissionAccess.Write。System.IO.FileMode.Create 等效于这样的请求：如果文件不存在，则使用 CreateNew；否则使用 Truncate。 |
| CreateNew | 指定操作系统应创建新文件。此操作需要 FileIOPermissionAccess.Write。如果文件已存在，则将引发 IOException。 |
| Open | 指定操作系统应打开现有文件。打开文件的能力取决于 FileAccess   所指定的值。如果该文件不存在，则引发 System.IO.FileNotFoundException。 |
| OpenOrCreate | 指定操作系统应打开文件（如果文件存在）；否则，应创建新文件。如果用 FileAccess.Read   打开文件，则需要 FileIOPermissionAccess.Read。如果文件访问为 FileAccess.Write 或 FileAccess.ReadWrite，则需要 FileIOPermissionAccess.Write。如果文件访问为 FileAccess.Append，则需要 FileIOPermissionAccess.Append。 |
| Truncate | 指定操作系统应打开现有文件。文件一旦打开，就将被截断为零字节大小。此操作需要 FileIOPermissionAccess.Write。试图从使用 Truncate 打开的文件中进行读取将导致异常。 |

\*3: FileStream(IntPtr, FileAccess, Boolean ownsHandle)

       FileAccess 参数也是一个枚举, 表示对于该文件的操作权限

|  |  |
| --- | --- |
| ReadWrite | 对文件的读访问和写访问。可从文件读取数据和将数据写入文件 |
| Write | 文件的写访问。可将数据写入文件。同 Read组合即构成读/写访问权 |
| Read | 对文件的读访问。可从文件中读取数据。同 Write组合即构成读写访问权 |

参数ownsHandle：也就是类似于前面和大家介绍的SafeFileHandler,有2点必须注意：

      1对于指定的文件句柄，操作系统不允许所请求的 access，例如，当 access 为 Write 或 ReadWrite 而文件句柄设置为只读访问时，会报出异常。所以 ownsHandle才是老大，FileAccess的权限应该在ownsHandle的范围之内

            2. FileStream 假定它对句柄有独占控制权。当 FileStream 也持有句柄时，读取、写入或查找可能会导致数据破坏。为了数据的安全，请使用句柄前调用 [Flush](http://msdn.microsoft.com/zh-cn/library/2bw4h516(v=vs.80).aspx)，并避免在使用完句柄后调用 Close 以外的任何方法。

\*4: FileStream(String, FileMode, FileAccess, FileShare)

FileShare：同样是个枚举类型：确定文件如何由进程共享。

|  |  |
| --- | --- |
| Delete | 允许随后删除文件。 |
| Inheritable | 使文件句柄可由子进程继承。Win32 不直接支持此功能。 |
| None | 谢绝共享当前文件。文件关闭前，打开该文件的任何请求（由此进程或另一进程发出的请求）都将失败。 |
| Read | 允许随后打开文件读取。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取的请求（由此进程或另一进程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 |
| ReadWrite | 允许随后打开文件读取或写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取或写入的请求（由此进程或另一进程发出）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 |
| Write | 允许随后打开文件写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行写入的请求（由此进程或另一进过程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。 |

\*5: FileStream(String, FileMode, FileAccess, FileShare, Int32, Boolean async )

 Int32:这是一个缓冲区的大小，大家可以按照自己的需要定制，

 Boolean async:是否异步读写，告诉FileStream示例，是否采用异步读写

\*6: FileStream(String, FileMode, FileAccess, FileShare, Int32, FileOptions)

      FileOptions：这是类似于FileStream对于文件操作的高级选项

FileStream 常用属性介绍

 \*1：CanRead ：指示FileStream是否可以读操作

 \*2：CanSeek：指示FileStream是否可以跟踪查找流操作

 \*3：IsAsync：FileStream是否同步工作还是异步工作

 \*4：Name：FileStream的名字 只读属性

 \*5：ReadTimeout ：设置读取超时时间

 \*6：SafeFileHandle : 文件安全句柄 只读属性

 \*7：position：当前FileStream所在的流位置

FileStream 常用方法介绍

以下方法重写了Stream的一些虚方法（\*\*[这里大家点击这里可以参考第一篇来温故下](http://www.cnblogs.com/JimmyZheng/archive/2012/03/17/2402814.html" \l "no4)，这里不再叙述）

1：IAsyncResult BeginRead  异步读取

2：IAsyncResult BeginWrite  异步写

3：void  Close  关闭当前FileStream

4：void EndRead 异步读结束

5：void  EndWrite 异步写结束

6：void Flush 立刻释放缓冲区，将数据全部导出到基础流（文件中）

7：int Read 一般读取

8：int ReadByte 读取单个字节

9：long Seek 跟踪查找流所在的位置

10：void SetLength 设置FileStream的长度

11：void Write 一般写

12：void  WriteByte写入单个字节

属于FileStream独有的方法

\*1：FileSecurity  GetAccessControl()： 这个不是很常用，FileSecurity 是文件安全类，直接表达当前文件的访问控制列表（ACL）的符合当前文件权限的项目，ACL大家有个了解就行，以后会单独和大家讨论下ACL方面的知识

\*2:  void Lock(long position,long length)： 这个Lock方法和线程中的Look关键字很不一样，它能够锁住文件中的某一部分，非常的强悍！用了这个方法我们能够精确锁定住我们需要锁住的文件的部分内容

\*3:  void SetAccessControl(FileSecurity fileSecurity)：和GetAccessControl很相似，ACL技术会在以后单独介绍

\*4:  void Unlock (long position,long length)：正好和lock方法相反，对于文件部分的解锁

文件的新建和拷贝（主要演示文件同步和异步操作）

            首先我们尝试DIY一个IFileConfig

/// <summary>文件配置接口</summary>  
 public interface IFileConfig{  
 string FileName { get; set; }  
 bool IsAsync { get; set; }  
 }

创建文件配置类CreateFileConfig，用于添加文件一些配置设置，实现添加文件的操作

/// <summary>创建文件配置类</summary>  
 public class CreateFileConfig : IFileConfig{  
 // 文件名  
 public string FileName { get; set; }  
 //是否异步操作  
 public bool IsAsync { get; set; }  
 //创建文件所在url  
 public string CreateUrl { get; set; }  
 }

让我们定义一个文件流测试类：FileStreamTest 来实现文件的操作

/// <summary>/// FileStreamTest 类/// </summary>  
 public class FileStreamTest

在该类中实现一个简单的Create方法用来同步或异步的实现添加文件，FileStream会根据配置类去选择相应的构造函数，实现异步或同步的添加方式

/// <summary>/// 添加文件方法/// </summary>  
 /// <param name="config"> 创建文件配置类</param>  
 public void Create(IFileConfig config){  
 lock (\_lockObject){  
 //得到创建文件配置类对象  
 var createFileConfig = config as CreateFileConfig;  
 //检查创建文件配置类是否为空  
 if (this.CheckConfigIsError(config)) return;  
 //假设创建完文件后写入一段话，实际项目中无需这么做，这里只是一个演示  
 char[] insertContent = "HellowWorld".ToCharArray();  
 //转化成 byte[]  
 byte[] byteArrayContent = Encoding.Default.GetBytes(insertContent, 0, insertContent.Length);  
 //根据传入的配置文件中来决定是否同步或异步实例化stream对象  
 FileStream stream = createFileConfig.IsAsync ?  
 new FileStream(createFileConfig.CreateUrl, FileMode.Create, FileAccess.ReadWrite, FileShare.None, 4096, true)  
 : new FileStream(createFileConfig.CreateUrl, FileMode.Create);  
 using (stream) {  
 // 如果不注释下面代码会抛出异常，google上提示是WriteTimeout只支持网络流  
 // stream.WriteTimeout = READ\_OR\_WRITE\_TIMEOUT;  
 //如果该流是同步流并且可写  
 if (!stream.IsAsync && stream.CanWrite)  
 stream.Write(byteArrayContent, 0, byteArrayContent.Length);  
 else if (stream.CanWrite)//异步流并且可写  
 stream.BeginWrite(byteArrayContent, 0, byteArrayContent.Length, this.End\_CreateFileCallBack, stream);  
 stream.Close();  
 }  
 }  
 }

如果采用异步的方式则最后会进入End\_CreateFileCallBack回调方法，result.AsyncState对象就是上图stream.BeginWrite()方法的最后一个参数还有一点必须注意的是每一次使用BeginWrite()方法事都要带上EndWrite()方法，Read方法也一样

/// <summary>/// 异步写文件callBack方法/// </summary>  
 /// <param name="result">IAsyncResult</param>  
 private void End\_CreateFileCallBack(IAsyncResult result){  
 //从IAsyncResult对象中得到原来的FileStream  
 var stream = result.AsyncState as FileStream;  
 //结束异步写   
 Console.WriteLine("异步创建文件地址：{0}", stream.Name);  
 stream.EndWrite(result);  
 Console.ReadLine();  
 }

文件复制的方式思路比较相似，首先定义复制文件配置类，由于在异步回调中用到该配置类的属性，所以新增了文件流对象和相应的字节数组

/// <summary>/// 文件复制/// </summary>  
 public class CopyFileConfig : IFileConfig{  
 // 文件名  
 public string FileName { get; set; }  
 //是否异步操作  
 public bool IsAsync { get; set; }  
 //原文件地址  
 public string OrginalFileUrl { get; set; }  
 //拷贝目的地址  
 public string DestinationFileUrl { get; set; }  
 //文件流，异步读取后在回调方法内使用  
 public FileStream OriginalFileStream { get; set; }  
 //原文件字节数组，异步读取后在回调方法内使用  
 public byte[] OriginalFileBytes { get; set; }  
 }

然后在FileStreamTest 类中新增一个Copy方法实现文件的复制功能

/// <summary>/// 复制方法/// </summary>  
 /// <param name="config">拷贝文件复制</param>  
 public void Copy(IFileConfig config){  
 lock (\_lockObject){  
 //得到CopyFileConfig对象  
 var copyFileConfig = config as CopyFileConfig;  
 // 检查CopyFileConfig类对象是否为空或者OrginalFileUrl是否为空  
 if (CheckConfigIsError(copyFileConfig) || !File.Exists(copyFileConfig.OrginalFileUrl)) return;  
 //创建同步或异步流  
 FileStream stream = copyFileConfig.IsAsync ?  
 new FileStream(copyFileConfig.OrginalFileUrl, FileMode.Open, FileAccess.Read, FileShare.Read, 4096, true)  
 : new FileStream(copyFileConfig.OrginalFileUrl, FileMode.Open);  
 //定义一个byte数组接受从原文件读出的byte数据  
 byte[] orignalFileBytes = new byte[stream.Length];  
 using (stream){  
 // stream.ReadTimeout = READ\_OR\_WRITE\_TIMEOUT;  
 //如果异步流  
 if (stream.IsAsync){  
 //将该流和读出的byte[]数据放入配置类，在callBack中可以使用  
 copyFileConfig.OriginalFileStream = stream;  
 copyFileConfig.OriginalFileBytes = orignalFileBytes;  
 if (stream.CanRead)  
 //异步开始读取，读完后进入End\_ReadFileCallBack方法，该方法接受copyFileConfig参数  
 stream.BeginRead(orignalFileBytes, 0, orignalFileBytes.Length, End\_ReadFileCallBack, copyFileConfig);  
 }else{//否则同步读取  
 if (stream.CanRead){  
 //一般读取原文件  
 stream.Read(orignalFileBytes, 0, orignalFileBytes.Length);  
 }  
 //定义一个写流，在新位置中创建一个文件  
 FileStream copyStream = new FileStream(copyFileConfig.DestinationFileUrl, FileMode.CreateNew);  
 using (copyStream){  
 // copyStream.WriteTimeout = READ\_OR\_WRITE\_TIMEOUT;  
 //将源文件的内容写进新文件  
 copyStream.Write(orignalFileBytes, 0, orignalFileBytes.Length);  
 copyStream.Close();  
 }  
 }  
 stream.Close();  
 Console.ReadLine();  
 }  
 }  
 }

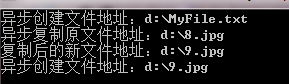
最后，如果采用异步的方式，则会进入End\_ReadFileCallBack回调函数进行异步读取和异步写操作

/// <summary>异步读写文件方法</summary>  
 /// <param name="result"></param>  
 private void End\_ReadFileCallBack(IAsyncResult result){  
 //得到先前的配置文件  
 var config = result.AsyncState as CopyFileConfig;  
 //结束异步读  
 config.OriginalFileStream.EndRead(result);  
 //异步读后立即写入新文件地址  
 if (File.Exists(config.DestinationFileUrl)) File.Delete(config.DestinationFileUrl);  
 FileStream copyStream = new FileStream(config.DestinationFileUrl, FileMode.CreateNew);  
 using (copyStream){  
 Console.WriteLine("异步复制原文件地址：{0}", config.OriginalFileStream.Name);  
 Console.WriteLine("复制后的新文件地址：{0}", config.DestinationFileUrl);  
 //调用异步写方法CallBack方法为End\_CreateFileCallBack，参数是copyStream  
 copyStream.BeginWrite(config.OriginalFileBytes, 0, config.OriginalFileBytes.Length, this.End\_CreateFileCallBack,copyStream);  
 copyStream.Close();  
 }  
 }

 最后让我们在main函数调用下：

static void Main(string[] args){  
 FileStreamTest test = new FileStreamTest();  
 //创建文件配置类  
 CreateFileConfig createFileConfig = new CreateFileConfig { CreateUrl = @"d:\MyFile.txt", IsAsync = true };  
 //复制文件配置类  
 CopyFileConfig copyFileConfig = new CopyFileConfig{  
 OrginalFileUrl = @"d:\8.jpg",  
 DestinationFileUrl = @"d:\9.jpg",  
 IsAsync = true  
 };  
 test.Create(createFileConfig);  
 test.Copy(copyFileConfig);  
 }

输出结果：



实现文件本地分段上传

            上面的例子是将一个文件作为整体进行操作，这样会带来一个问题，当文件很大或者网络不是很稳定的时候会发生意想不到的错误，那我们该怎么解决这一问题呢？其实有种思路还是不错的，那就是分段传输：



那就DIY一个简单的分段传输的例子，我们先将处理每一段的逻辑先整理好

/// <summary>/// 分段上传例子/// </summary>  
 public class UpFileSingleTest{  
 //我们定义Buffer为1000  
 public const int BUFFER\_COUNT = 1000;  
  
 /// <summary>将文件上传至服务器（本地），由于采取分段传输所以，每段必须有一个起始位置和相对应该数据段的数据</summary>  
 /// <param name="filePath">服务器上文件地址</param>  
 /// <param name="startPositon">分段起始位置</param>  
 /// <param name="btArray">每段的数据</param>  
 private void WriteToServer(string filePath,int startPositon,byte[] btArray) {  
 FileStream fileStream = new FileStream(filePath, FileMode.OpenOrCreate);  
 using (fileStream) {  
 //将流的位置设置在该段起始位置  
 fileStream.Position = startPositon;  
 //将该段数据通过FileStream写入文件中，每次写一段的数据，就好比是个水池，分段蓄水一样，直到蓄满为止  
 fileStream.Write(btArray, 0, btArray.Length);  
 }  
 }  
 /// <summary>处理单独一段本地数据上传至服务器的逻辑，根据客户端传入的startPostion和totalCount来处理相应段的数据上传至服务器（本地）</summary>  
 /// <param name="localFilePath">本地需要上传的文件地址</param>  
 /// <param name="uploadFilePath">服务器（本地）目标地址</param>  
 /// <param name="startPostion">该段起始位置</param>  
 /// <param name="totalCount">该段最大数据量</param>  
 public void UpLoadFileFromLocal(string localFilePath,string uploadFilePath,int startPostion,int totalCount) {  
 //if(!File.Exists(localFilePath)){return;}  
 //每次临时读取数据数  
 int tempReadCount = 0;  
 int tempBuffer = 0;  
 //定义一个缓冲区数组  
 byte[] bufferByteArray = new byte[BUFFER\_COUNT];  
 //定义一个FileStream对象  
 FileStream fileStream = new FileStream(localFilePath,FileMode.Open);  
 //将流的位置设置在每段数据的初始位置  
 fileStream.Position = startPostion;  
 using (fileStream){  
 //循环将该段数据读出在写入服务器中  
 while (tempReadCount < totalCount){  
   
 tempBuffer = BUFFER\_COUNT;  
 //每段起始位置+每次循环读取数据的长度  
 var writeStartPosition = startPostion + tempReadCount;  
 //当缓冲区的数据加上临时读取数大于该段数据量时，  
 //则设置缓冲区的数据为totalCount-tempReadCount 这一段的数据  
 if (tempBuffer + tempReadCount > totalCount) {  
 //缓冲区的数据为totalCount-tempReadCount   
 tempBuffer = totalCount-tempReadCount;  
 //读取该段数据放入bufferByteArray数组中  
 fileStream.Read(bufferByteArray, 0, tempBuffer);  
 if (tempBuffer > 0) {  
 byte[] newTempBtArray = new byte[tempBuffer];  
 Array.Copy(bufferByteArray, 0, newTempBtArray, 0, tempBuffer);  
 //将缓冲区的数据上传至服务器  
 this.WriteToServer(uploadFilePath, writeStartPosition, newTempBtArray);  
 }  
 } else if (tempBuffer == BUFFER\_COUNT) {  
 //如果缓冲区的数据量小于该段数据量，并且tempBuffer=设定BUFFER\_COUNT时，通过  
 //while 循环每次读取一样的buffer值的数据写入服务器中，直到将该段数据全部处理完毕   
 fileStream.Read(bufferByteArray, 0, tempBuffer);  
 this.WriteToServer(uploadFilePath, writeStartPosition, bufferByteArray);  
 }  
 //通过每次的缓冲区数据，累计增加临时读取数  
 tempReadCount += tempBuffer;  
 }  
 }  
 }  
 }

一切准备就绪，我们剩下的就是将文件切成几段进行上传了

static void Main(string[] args){  
 UpFileSingleTest test=new UpFileSingleTest();  
 FileInfo info = new FileInfo(@"G:\\Skyrim\20080204173728108.torrent");  
 //取得文件总长度  
 var fileLegth = info.Length;  
 //假设将文件切成5段  
 var divide = 5;  
 //取到每个文件段的长度  
 var perFileLengh = (int)fileLegth / divide;  
 //表示最后剩下的文件段长度比perFileLengh小  
 var restCount = (int)fileLegth % divide;  
 //循环上传数据  
 for (int i = 0; i < divide+1; i++){  
 //每次定义不同的数据段,假设数据长度是500，那么每段的开始位置都是i\*perFileLength  
 var startPosition = i \* perFileLengh;  
 //取得每次数据段的数据量  
 var totalCount = fileLegth - perFileLengh \* i > perFileLengh ? perFileLengh : (int)(fileLegth - perFileLengh \* i);  
 //上传该段数据  
 test.UpLoadFileFromLocal(@"G:\\Skyrim\\20080204173728108.torrent", @"G:\\Skyrim\\20080204173728109.torrent", startPosition, i == divide ? divide : totalCount);  
 }  
 }

上传结果：

IMG_281

总的来说，分段传输比直接传输复杂许多，我会在今后的例子中加入多线程，这样的话每段数据的传输都能通过一个线程单独处理，能够提升上传性能和速度