C# 温故而知新：Stream篇（—）

**什么是Stream?**

MSDN 中的解释太简洁了: 提供字节序列的一般视图

（我可不想这么理解，这必定让我抓狂，我理解的流是向自然界的河流那样清澈而又美丽，c#中的流也是一样，许多技术或者说核心技术都需要流的帮忙）

**那什么是字节序列呢？**

其实简单的来理解的话字节序列指的是：字节对象都被存储为连续的字节序列，字节按照一定的顺序进行排序组成了字节序列

那什么关于流的解释可以抽象为下列情况：打个比方：一条河中有一条鱼游过，这个鱼就是一个字节，这个字节包括鱼的眼睛，嘴巴，等组成8个二进制，显然这条河就是我们的核心对象：流

**马上进入正题，让我们来解释下c#的 Stream 是如何使用的**

让我们直接温故或学习下Stream类的结构，属性和相关方法

首先是构造函数，Stream 类有一个protected 类型的构造函数, 但是它是个抽象类，无法直接如下使用

Stream stream = new Stream();

所以我们自定义一个流继承自Stream 看看哪些属性必须重写或自定义:

public class MyStreamExample : Stream {  
 public override bool CanRead{get { throw new NotImplementedException(); }}//判断该流是否能够读取：  
 public override bool CanSeek{get { throw new NotImplementedException(); }}//判断该流是否支持跟踪查找  
 public override bool CanWrite{get { throw new NotImplementedException(); }}//判断当前流是否可写  
  
 public override void Flush(){  
 throw new NotImplementedException();  
 }  
 public override long Length{get { throw new NotImplementedException(); }}  
 public override long Position{  
 get{throw new NotImplementedException();}  
 set{throw new NotImplementedException();}  
 }  
 public override int Read(byte[] buffer, int offset, int count){  
 throw new NotImplementedException();  
 }  
 public override long Seek(long offset, SeekOrigin origin){  
 throw new NotImplementedException();  
 }  
 public override void SetLength(long value){  
 throw new NotImplementedException();  
 }  
 public override void Write(byte[] buffer, int offset, int count){  
 throw new NotImplementedException();  
 }  
 }

可以看出系统自动帮我们实现了Stream 的抽象属性和属性方法

1：CanRead: 只读属性，判断该流是否能够读取

2：CanSeek: 只读属性，判断该流是否支持跟踪查找

3：CanWrite: 只读属性，判断当前流是否可写

4：void Flush(): 当我们使用流写文件时, 数据流会先进入到缓冲区中，而不会立刻写入文件，当执行这个方法后，缓冲区的数据流会立即注入基础流

     MSDN中的描述：使用此方法将所有信息从基础缓冲区移动到其目标或清除缓冲区，或者同时执行这两种操作。根据对象的状态，可能需要修

     改流内的当前位置（例如，在基础流支持查找的情况下即如此）当使用 [StreamWriter](http://msdn.microsoft.com/zh-cn/library/system.io.streamwriter.aspx) 或 [BinaryWriter](http://msdn.microsoft.com/zh-cn/library/system.io.binarywriter.aspx) 类时，不要刷新 [Stream](http://msdn.microsoft.com/zh-cn/library/system.io.stream.aspx) 基对象。

     而应使用该类的 **Flush** 或 [Close](http://msdn.microsoft.com/zh-cn/library/system.io.stream.close.aspx) 方法，此方法确保首先将该数据刷新至基础流，然后再将其写入文件。

5：Length:表示流的长度

6：Position属性：（非常重要）

虽然从字面中可以看出这个Position属性只是标示了流中的一个位置而已，可是我们在实际开发中会发现这个想法会非常的幼稚，

很多asp.net项目中文件或图片上传中很多朋友会经历过这样一个痛苦：Stream对象被缓存了，导致了Position属性在流中无法

找到正确的位置，这点会让人抓狂，其实解决这个问题很简单，聪明的你肯定想到了，其实我们每次使用流前必须将Stream.Position

设置成0就行了，但是这还不能根本上解决问题，最好的方法就是用Using语句将流对象包裹起来，用完后关闭回收即可。

7：abstract int Read(byte[] buffer, int offset, int count)

这个方法包含了3个关键的参数：缓冲字节数组，位移偏量和读取字节个数，每次读取一个字节后会返回一个缓冲区中的总字节数

第一个参数：这个数组相当于一个空盒子，Read（）方法每次读取流中的一个字节将其放进这个空盒子中。（全部读完后便可使用buffer字节数组了）

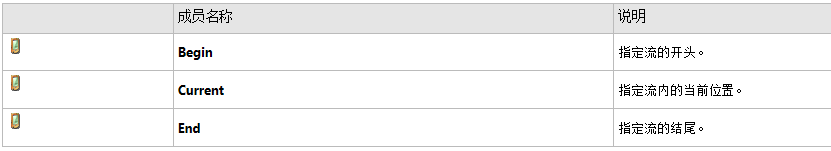
第二个参数：表示位移偏量，告诉我们从流中哪个位置（偏移量）开始读取。

最后一个参数：就是读取多少字节数。

返回值便是总共读取了多少字节数.

8：abstract long Seek(long offset, SeekOrigin origin)

    大家还记得Position属性么？其实Seek方法就是重新设定流中的一个位置，在说明offset参数作用之前大家先来了解下SeekOrigin这个枚举：



如果 offset 为负，则要求新位置位于 origin 指定的位置之前，其间隔相差 offset 指定的字节数。如果 offset 为零 (0)，则要求新位置位于由 origin 指定的位置处。

如果 offset 为正，则要求新位置位于 origin 指定的位置之后，其间隔相差 offset 指定的字节数.

Stream. Seek(-3,Origin.End);  表示在流末端往前数第3个位置

Stream. Seek(0,Origin.Begin); 表示在流的开头位置

Stream. Seek(3,Orig`in.Current); 表示在流的当前位置往后数第三个位置

查找之后会返回一个流中的一个新位置。其实说道这大家就能理解Seek方法的精妙之处了吧

9：abstract void Write(byte[] buffer,int offset,int count)

这个方法包含了3个关键的参数：缓冲字节数组，位移偏量和读取字节个数和read方法不同的是 write方法中的第一个参数buffer已经有了许多byte类型的数据，我们只需通过设置 offset和count来将buffer中的数据写入流中

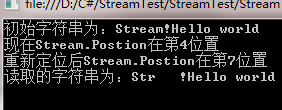
10：virtual void Close()

关闭流并释放资源，在实际操作中，如果不用using的话，别忘了使用完流之后将其关闭

这个方法特别重要，使用完当前流千万别忘记关闭！

为了让大家能够快速理解和消化上述属性和方法我会写个示例并且关键部分会详细说明

static void Main(string[] args){  
 byte[] buffer = null;  
  
 string testString = "Stream!Hello world";  
 char[] readCharArray = null;  
 byte[] readBuffer = null;  
 string readString = string.Empty;  
 //关于MemoryStream 我会在后续章节详细阐述  
 using (MemoryStream stream = new MemoryStream()) {  
 Console.WriteLine("初始字符串为：{0}", testString);  
   
 if (stream.CanWrite){//如果该流可写  
 //首先我们尝试将testString写入流中  
 //关于Encoding我会在另一篇文章中详细说明，暂且通过它实现string->byte[]的转换  
 buffer = Encoding.Default.GetBytes(testString);  
 //我们从该数组的第一个位置开始写，长度为3，写完之后 stream中便有了数据  
 //对于新手来说很难理解的就是数据是什么时候写入到流中，在冗长的项目代码面前，我碰见过很  
 //多新手都会有这种经历，我希望能够用如此简单的代码让新手或者老手们在温故下基础  
 stream.Write(buffer, 0,3);  
 Console.WriteLine("现在Stream.Postion在第{0}位置",stream.Position+1);  
  
 //从刚才结束的位置（当前位置）往后移3位，到第7位  
 long newPositionInStream =stream.CanSeek? stream.Seek(3, SeekOrigin.Current):0;  
  
 Console.WriteLine("重新定位后Stream.Postion在第{0}位置", newPositionInStream+1);  
 if (newPositionInStream < buffer.Length){  
 //将从新位置（第7位）一直写到buffer的末尾，注意下stream已经写入了3个数据“Str”  
 stream.Write(buffer, (int)newPositionInStream, buffer.Length - (int)newPositionInStream);  
 }  
   
 //写完后将stream的Position属性设置成0，开始读流中的数据  
 stream.Position = 0;  
  
 // 设置一个空的盒子来接收流中的数据，长度根据stream的长度来决定  
 readBuffer = new byte[stream.Length];  
  
 //设置stream总的读取数量 ，  
 //注意！这时候流已经把数据读到了readBuffer中  
 int count = stream.CanRead?stream.Read(readBuffer, 0, readBuffer.Length):0;  
   
 //由于刚开始时我们使用加密Encoding的方式,所以我们必须解密将readBuffer转化成Char数组，这样才能重新拼接成string  
 //首先通过流读出的readBuffer的数据求出从相应Char的数量  
 int charCount = Encoding.Default.GetCharCount(readBuffer, 0, count);  
 //通过该Char的数量 设定一个新的readCharArray数组  
 readCharArray = new char[charCount];  
 //Encoding 类的强悍之处就是不仅包含加密的方法，甚至将解密者都能创建出来（GetDecoder()），  
 //解密者便会将readCharArray填充（通过GetChars方法，把readBuffer 逐个转化将byte转化成char，并且按一致顺序填充到readCharArray中）  
 Encoding.Default.GetDecoder().GetChars(readBuffer, 0, count, readCharArray, 0);  
 for (int i = 0; i < readCharArray.Length; i++){  
 readString += readCharArray[i];  
 }  
 Console.WriteLine("读取的字符串为：{0}", readString);  
 }  
 stream.Close();  
 }  
 Console.ReadLine();  
 }  
显示结果：



大家需要特别注意的是stream.Positon这个很神奇的属性，在复杂的程序中，往往流对象操作也会很复杂，一定要切记将stream.Positon设置在你所需要的正确位置，还有就是 using语句的使用，它会自动销毁stream对象，当然Stream.Close()大家都懂的

 接着让我们来说下关于流中怎么实现异步操作

在Stream基类中还有几个关键方法,它们能够很好实现异步的读写，

异步读取

public virtual IAsyncResult BeginRead(byte[] buffer,int offset,int count,AsyncCallback callback,Object state)

异步写

public virtual IAsyncResult BeginWrite( byte[] buffer, int offset, int count, AsyncCallback callback, Object state )

结束异步读取

public virtual int EndRead( IAsyncResult asyncResult )

结束异步写

public virtual void EndWrite( IAsyncResult asyncResult )

大家很容易的就能发现前两个方法实现了IAsyncResult接口，后2个end方法也顺应带上了一个IAsyncResult参数，

其实并不复杂，(必须说明下 每次调用 Begin方法时都必须调用一次 相对应的end方法)

和一般同步read或write方法一致的是，他们可以当做同步方法使用，但是在复杂的情况下可能也难逃阻塞崩溃等等，但是一旦启用了

异步之后，这些类似于阻塞问题会不复存在，可见微软对于异步的支持正在加大。