C# 温故而知新：Stream篇（五）MemoryStream

简单介绍一下MemoryStream

MemoryStream是内存流,为系统内存提供读写操作，由于MemoryStream是通过无符号字节数组组成的，可以说MemoryStream的性能可以

算比较出色，所以它担当起了一些其他流进行数据交换时的中间工作，同时可降低应用程序中对临时缓冲区和临时文件的需要,其实MemoryStream的重要性不亚于FileStream，在很多场合我们必须使用它来提高性能

MemoryStream和FileStream的区别

前文中也提到了，FileStream主要对文件的一系列操作，属于比较高层的操作，但是MemoryStream却很不一样，它更趋向于底层内存的操作，这样能够达到更快的速度和性能，也是他们的根本区别，很多时候，操作文件都需要MemoryStream来实际进行读写，最后放入到相应的FileStream中,不仅如此，在诸如XmlWriter的操作中也需要使用到MemoryStream提高读写速度

通过部分源码深入了解下MemoryStream

 由于篇幅关系，本篇无法详细说明其源码，还请大家海涵，这里我就简单介绍下Write()方法的源码

public override void Write(byte[] buffer, int offset, int count) {

if (!\_isOpen) \_\_Error.StreamIsClosed();

if (!\_writable) \_\_Error.WriteNotSupported();

if (buffer==null)

throw new ArgumentNullException("buffer", Environment.GetResourceString("ArgumentNull\_Buffer"));

if (offset < 0)

throw new ArgumentOutOfRangeException("offset", Environment.GetResourceString("ArgumentOutOfRange\_NeedNonNegNum"));

if (count < 0)

throw new ArgumentOutOfRangeException("count", Environment.GetResourceString("ArgumentOutOfRange\_NeedNonNegNum"));

if (buffer.Length - offset < count)

throw new ArgumentException(Environment.GetResourceString("Argument\_InvalidOffLen"));

int i = \_position + count;

// Check for overflow

if (i < 0) throw new IOException(Environment.GetResourceString("IO.IO\_StreamTooLong"));

if (i > \_length) {

bool mustZero = \_position > \_length;

if (i > \_capacity) {

bool allocatedNewArray = EnsureCapacity(i);

if (allocatedNewArray)

mustZero = false;

}

if (mustZero) Array.Clear(\_buffer, \_length, i - \_length);

\_length = i;

}

if (count <= 8) {

int byteCount = count;

while (--byteCount >= 0)

\_buffer[\_position + byteCount] = buffer[offset + byteCount];

}

else

Buffer.InternalBlockCopy(buffer, offset, \_buffer, \_position, count);

\_position = i;

return;

}

关于MemoryStream的源码大家可以自己学习，这里主要分析下MemoryStream最关键的Write()方法，自上而下，最开始的一系列判断大家很容易看明白，以后对有可能发生的异常应该了如指掌了吧，判断后会取得这段数据的长度 int i=\_position+count ，接下来会去判断该数据的长度是否超过了该流的长度，如果超过再去检查是否在流的可支配容量(字节)之内，(注意下EnsureCapacity方法，该方法会自动扩容stream的容量，但是前提条件是你使用了memoryStream的第二个构造函数，也就是带有参数是Capaciy)如果超过了流的可支配容量则将尾巴删除（将超过部分的数据清除）,接下来大家肯定会问，为什么要判断count<=8,其实8这个数字在流中很关键，个人认为微软为了性能需要而这样写：当字节小于8时则一个个读，当字节大于八时则用block拷贝的方式，在这个范围内递减循环将数据写入流中的缓冲\_buffer中，这个缓冲\_buffe是memoryStream的一个私有byte数组类型，流通过读取外部byte数据放入内部那个缓冲buffer中，如果流的长度超过了8，则用Buffer.InternalBloackCopy方法进行数组复制,不同于Array.Copy 前者是采用内存位移而非索引位移所以性能上有很大的提升。其实

这个方法的原形是属于c++中的。

分析MemorySteam最常见的OutOfMemory异常

先看下下面一段很简单的测试代码

//测试byte数组 假设该数组容量是256M

byte[] testBytes=new byte[256\*1024\*1024];

MemoryStream ms = new MemoryStream();

using (ms){

for (int i = 0; i < 1000; i++){

try{

ms.Write(testBytes, 0, testBytes.Length);

}

catch

{

Console.WriteLine("该内存流已经使用了{0}M容量的内存,该内存流最大容量为{1}M,溢出时容量为{2}M",

GC.GetTotalMemory(false) / (1024 \* 1024),//MemoryStream已经消耗内存量

ms.Capacity / (1024 \* 1024), //MemoryStream最大的可用容量

ms.Length / (1024 \* 1024));//MemoryStream当前流的长度（容量）

break;

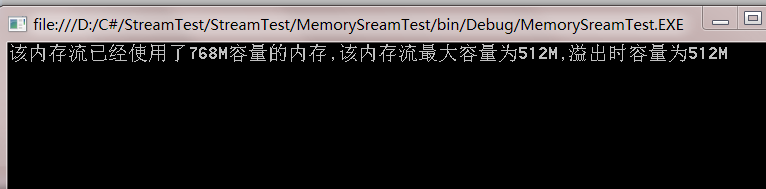
}

}

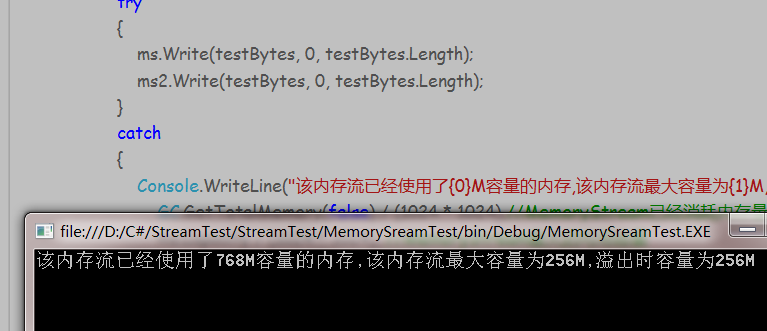
}

Console.ReadLine();

由于我们设定了一个256M的byte（有点恐怖），看下溢出时的状态



从输出结果看，MemoryStream默认可用最大容量是512M  发生异常时正好是其最大容量，聪明的你肯定会问：如果同时使用2个MemoryStream甚至于多个内存是怎么分配的？很好，还是用代码来看下输出结果，可以明显看出内存平均分给了2个MemoryStream但是最大容量还是512M



但是问题来了，假设我们需要操作比较大的文件，该怎么办呢?其实有2种方法能够搞定，一种是前文所说的分段处理，我们将byte数组分成等份进行处理，还有一个方法便是尽量增加MemoryStream的最大可用容量（字节），我们可以在声明MemoryStream构造函数时利用它的重载版本：MemoryStream(int capacity)到底怎么使用哪种方法比较好呢？其实笔者认为具体项目具体分析，前者分段处理的确能够解决大数据量操作的问题，但是牺牲了性能和时间（多线程暂时不考虑），后者可以得到性能上的优势但是其允许的最大容量是 int.MAX，所以无法给出一个明确的答案，大家在做项目按照需求自己定制即可，最关键的还是要取到性能和开销的最佳点位。

还有一种更恶心的溢出方式，往往会让大家抓狂，就是不定时溢出，就是MemoryStream处理的文件可能只有40M或更小时也会发生OutOfMemory的异常，关于这个问题，终于在老外的一篇文章中得到了解释，运气不错，[陈彦铭大哥在他的博客中正好翻译了下](http://www.cnblogs.com/ymind/archive/2012/03/25/2416162.html)，免去我翻译的工作^^,由于这个牵涉到windows的内存机制，包括 内存页，进程的虚拟地址空间等，比较复杂，所以大家看他的这篇文章前，我先和大家简单介绍下页和进程的虚拟地址

内存页：内存页分为：文件页和计算页；内存中的文件页是文件缓存区，即文件型的内存页，用于存放文件数据的内存页（也称永久页），作用在于读写文件时可以减少对磁盘的访问，如果它的大小设置得太小，会引起系统频繁地访问磁盘，增加磁盘I/O；设置太大，会浪费内存资源。内存中的计算页也称为计算型的内存页，主要用于存放程序代码和临时使用的数据

进程的虚拟地址：每一个进程被给予它的非常私有的虚拟地址空间。对于32位的进程，地址空间是4G因为一个32位指针能够有从0x00000000到0xffffffff之间的任意值。这个范围允许指针有从4294967296个值的一个，覆盖了一个进程的4G范围。对于64位进程，地址空间是16eb因为一个64位指针能够指向18,446,744,073,709,551,616个值中的一个，覆盖一个进程的16eb范围。这是十分宽广的范围。

上述概念都来自windows核心编程 这本书，其实这本书对我们程序员来说很重要，对于内存的操作，本人也是小白，看来这本书非买不可了。。。。

MemoryStream 的构造

MemoryStream()：MemoryStream 允许不带参数的构造

 MemoryStream(byte[] byte)：Byte数组是包含了一定的数据的byte数组，这个构造很重要，初学者或者用的不是很多的程序员会忽略这个构造导致后面读取或写入数据时发现memoryStream中没有byte数据，会导致很郁闷的感觉,大家注意下就行，有时也可能无需这样，因为很多方法返回值已经是MemoryStream了

 MemoryStream(int capacity)：这个是重中之重，为什么这么说呢？我在本文探讨关于OutOfMemory异常中也提到了，如果你想额外提高MemoryStream的吞吐量(字节)，也只能靠这个方法提升一定的吞吐量，最多也只能到int.Max,这个方法也是解决OutOfMemory的一个可行方案

 MemoryStream(byte[] byte, bool writeable)：Writeable参数定义该流是否可写

 MemoryStream(byte[] byte, int index, int count)：Index 参数定义从byte数组中的索引index，Count  参数是获取的数据量的个数

MemoryStream(byte[] byte,int index, int count, bool writeable, bool publiclyVisible)：publiclyVisible 参数表示true 可以启用 GetBuffer方法，它返回无符号字节数组，流从该数组创建；否则为 false，（大家一定觉得这很难理解，别急下面的方法中我会详细讲下这个东东）

 MemoryStream 的属性

Memory 的属性大致都是和其父类很相似，这些功能在我的这篇中已经详细讨论过，所以我简单列举一下其属性：



其独有的属性:Capacity:这个前文其实已经提及，它表示该流的可支配容量（字节），非常重要的一个属性

MemoryStream 的方法

[对于重写的方法这里不再重复说明，大家可以参考我写的第一篇](http://www.cnblogs.com/JimmyZheng/archive/2012/03/17/2402814.html" \l "no4)

以下是memoryStream独有的方法

virtual byte[] GetBuffer()

这个方法使用时需要小心，因为这个方法返回无符号字节数组，也就是说，即使我只输入几个字符例如”HellowWorld”我们只希望返回11个数据就行，

可是这个方法会把整个缓冲区的数据，包括那些已经分配但是实际上没有用到的字节数据都返回出来，如果想启用这个方法那必须使用上面最后一个构

造函数，将publiclyVisible属性设置成true就行，这也是上面那个构造函数的作用所在

virtual void WriteTo(Stream stream)

这个方法的目的其实在本文开始时讨论性能问题时已经指出，memoryStream常用起中间流的作用，

所以读写在处理完后将内存流写入其他流中

 简单示例 XmlWriter中使用MemoryStream

[IMG_263](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

/// <summary>

/// 演示在xmlWriter中使用MemoryStream

/// </summary>

public static void UseMemoryStreamInXMLWriter()

{

MemoryStream ms = new MemoryStream();

using (ms)

{

//定义一个XMLWriter

using (XmlWriter writer = XmlWriter.Create(ms))

{

//写入xml头

writer.WriteStartDocument(true);

//写入一个元素

writer.WriteStartElement("Content");

//为这个元素新增一个test属性

writer.WriteStartAttribute("test");

//设置test属性的值

writer.WriteValue("逆时针的风");

//释放缓冲，这里可以不用释放，但是在实际项目中可能要考虑部分释放对性能带来的提升 writer.Flush();

Console.WriteLine("此时内存使用量为:{2}KB,该MemoryStream的已经使用的容量为{0}byte,默认容量为{1}byte",

Math.Round((double)ms.Length, 4), ms.Capacity,GC.GetTotalMemory(false)/1024);

Console.WriteLine("重新定位前MemoryStream所在的位置是{0}",ms.Position);

//将流中所在的当前位置往后移动7位，相当于空格

ms.Seek(7, SeekOrigin.Current);

Console.WriteLine("重新定位后MemoryStream所在的位置是{0}", ms.Position);

//如果将流所在的位置设置为如下所示的位置则xml文件会被打乱

//ms.Position = 0;

writer.WriteStartElement("Content2");

writer.WriteStartAttribute("testInner");

writer.WriteValue("逆时针的风Inner");

writer.WriteEndElement();

writer.WriteEndElement();

//再次释放 writer.Flush();

Console.WriteLine("此时内存使用量为:{2}KB,该MemoryStream的已经使用的容量为{0}byte,默认容量为{1}byte",

Math.Round((double)ms.Length, 4), ms.Capacity, GC.GetTotalMemory(false)/1024);

//建立一个FileStream 文件创建目的地是d:\test.xml

FileStream fs = new FileStream(@"d:\test.xml",FileMode.OpenOrCreate);

using (fs)

{

//将内存流注入FileStream ms.WriteTo(fs);

if(ms.CanWrite)

//释放缓冲区 fs.Flush();

}

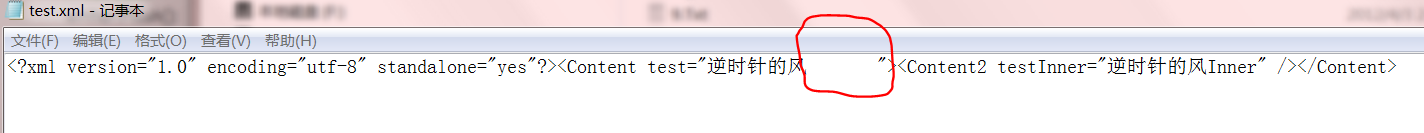
}

}

}

[IMG_264](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

      输出结果：



简单示例：自定义一个处理图片的HttpHandler

 有时项目里我们必须将图片进行一定的操作，例如水印，下载等，为了方便和管理我们可以自定义一个HttpHander 来负责这些工作

后台：

[IMG_266](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

public class ImageHandler : IHttpHandler

{

#region IHttpHandler Members

public bool IsReusable

{

get { return true; }

}

/// <summary>

/// 实现IHTTPHandler后必须实现的方法

/// </summary>

/// <param name="context">HttpContext上下文</param>

public void ProcessRequest(HttpContext context)

{

context.Response.Clear();

//得到图片名

var imageName = context.Request["ImageName"] == null ? "逆时针的风"

: context.Request["ImageName"].ToString();

//得到图片ID,这里只是演示，实际项目中不是这么做的

var id = context.Request["Id"] == null ? "01"

: context.Request["Id"].ToString();

//得到图片地址

var stringFilePath = context.Server.MapPath(string.Format("~/Image/{0}{1}.jpg", imageName, id));

//声明一个FileStream用来将图片暂时放入流中

FileStream stream = new FileStream(stringFilePath, FileMode.Open);

using (stream)

{

//透过GetImageFromStream方法将图片放入byte数组中

byte[] imageBytes = this.GetImageFromStream(stream,context);

//上下文确定写到客户短时的文件类型

context.Response.ContentType = "image/jpeg";

//上下文将imageBytes中的数据写到前段 context.Response.BinaryWrite(imageBytes);

stream.Close();

}

}

/// <summary>

/// 将流中的图片信息放入byte数组后返回该数组

/// </summary>

/// <param name="stream">文件流</param>

/// <param name="context">上下文</param>

/// <returns></returns>

private byte[] GetImageFromStream(FileStream stream, HttpContext context)

{

//通过stream得到Image

Image image = Image.FromStream(stream);

//加上水印

image = SetWaterImage(image, context);

//得到一个ms对象

MemoryStream ms = new MemoryStream();

using (ms)

{

//将图片保存至内存流 image.Save(ms, ImageFormat.Jpeg);

byte[] imageBytes = new byte[ms.Length];

ms.Position = 0;

//通过内存流读取到imageBytes

ms.Read(imageBytes, 0, imageBytes.Length);

ms.Close();

//返回imageBytes

return imageBytes;

}

}

/// <summary>

/// 为图片加上水印，这个方法不用在意，只是演示，所以没加透明度

/// 下次再加上吧

/// </summary>

/// <param name="image">需要加水印的图片</param>

/// <param name="context">上下文</param>

/// <returns></returns>

private Image SetWaterImage(Image image,HttpContext context)

{

Graphics graphics = Graphics.FromImage(image);

Image waterImage = Image.FromFile(context.Server.MapPath("~/Image/逆时针的风01.jpg"));

//在大图右下角画上水印图就行 graphics.DrawImage(waterImage,

new Point {

X = image.Size.Width - waterImage.Size.Width,

Y = image.Size.Height - waterImage.Size.Height

});

return image;

}

#endregion

}

[IMG_267](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

别忘了还要在Web.Config中进行配置，别忘记verb和path属性，否则会报错

<httpHandlers>

<add type="ImageHandler.ImageHandler,ImageHandler" verb="\*" path="ImageHandler.apsx"/>

</httpHandlers>

这样前台便能使用了

[IMG_268](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">

<h2>

About

</h2>

<p>

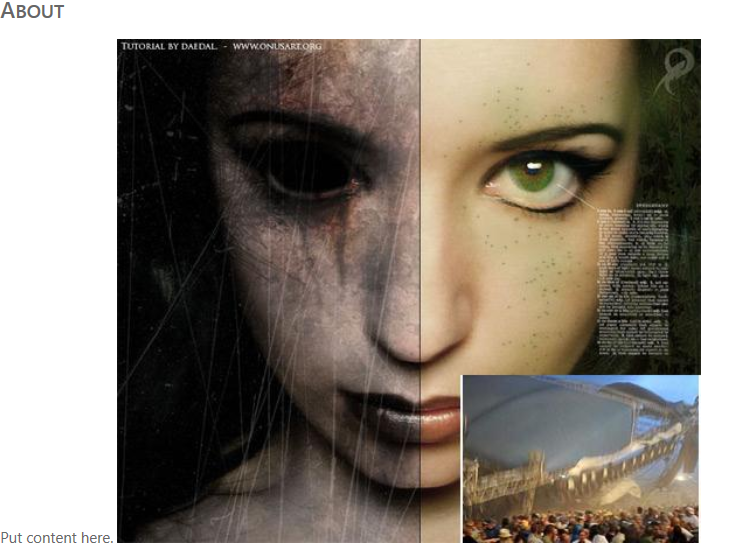
Put content here.

<asp:Image runat="server" ImageUrl="ImageHandler.apsx?ImageName=逆时针的风&Id=02" />

</p></asp:Content>

[IMG_269](http://www.cnblogs.com/JimmyZheng/archive/2012/04/14/javascript:void(0);)

输出结果



 本章总结

  本章主要介绍了MemoryStream 的一些概念，异常，结构，包括如何使用，如何解决一些异常等，感谢大家一直支持和鼓励，文中如出现错误还请大家海涵，深夜写文不容易，

  还请大家多多关注，下篇会介绍BufferedStream，尽请期待！