

.NET 基础拾遗 (3) : 字符串、集合和流 - 文章 - 伯乐在线



一、字符串处理1.1 StringBuilder类型有什么作用？

众所周知，在.NET中String是引用类型，具有不可变性，当一个String对象被修改、插入、连接、截断时，新的String对象就将被分配，这会直接影响到性能。但在实际开发中经常碰到的情况是，一个String对象的最终生成需要经过一个组装的过程，而在这个组装过程中必将会产生很多临时的String对象，而这些String对象将会在堆上分配，需要GC来回收，这些动作都会对程序性能产生巨大的影响。事实上，在String的组装过程中，其临时产生的String对象实例都不是最终需要的，因此可以说是没有必要分配的。

鉴于此，在.NET中提供了StringBuilder，其设计思想源于构造器（Builder）设计模式，致力于解决复杂对象的构造问题。对于String对象，正需要这样的构造器来进行组装。StringBuilder类型在最终生成String对象之前，将不会产生任何String对象，这很好地解决了字符串操作的性能问题。

以下代码展示了使用StringBuilder和不适用StringBuilder的性能差异：（这里的性能检测工具使用了老赵的CodeTimer类）

C#

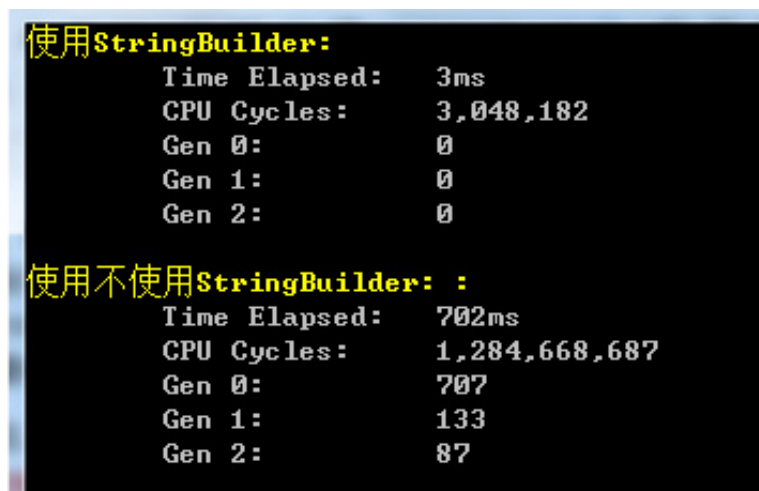
```
public class Program
{
    private const String item = "一个项目";
    private const String split = ";";
    static void Main(string[] args)
    {
        int number = 10000;
        // 使用StringBuilder
        CodeTimer.Time("使用StringBuilder: ", 1, () =>
        {
            UseStringBuilder(number);
        });
        // 不使用StringBuilder
        CodeTimer.Time("使用不使用StringBuilder: : ", 1, () =>
        {
            NotUseStringBuilder(number);
        });
        Console.ReadKey();
    }
}
```

```
}

static String UseStringBuilder(int number)
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    for (int i = 0; i < number; i++)
    {
        sb.Append(item);
        sb.Append(split);
    }
    sb.Remove(sb.Length - 1, 1);
    return sb.ToString();
}

static String NotUseStringBuilder(int number)
{
    String result = "";
    for (int i = 0; i < number; i++)
    {
        result += item;
        result += split;
    }
    return result;
}
}
```

上述代码的运行结果如下图所示，可以看出由于StringBuilder不会产生任何的中间字符串变量，因此效率上优秀不少！



```
使用StringBuilder:
Time Elapsed:    3ms
CPU Cycles:     3,048,182
Gen 0:          0
Gen 1:          0
Gen 2:          0

使用不使用StringBuilder: :
Time Elapsed:   702ms
CPU Cycles:    1,284,668,687
Gen 0:         707
Gen 1:         133
Gen 2:          87
```

看到StringBuilder这么优秀，不禁想发出一句：卧槽，牛逼！

于是，我们拿起我们的锤子（Reflector）撕碎StringBuilder的外套，看看里面到底装了什么？我们发现，在StringBuilder中定义了一个字符数组m_ChunkChars，它保存StringBuilder所管理着的字符串中的字符。

```
[Serializable, ComVisible(true), __DynamicallyInvokable]
public sealed class StringBuilder : ISerializable
{
    // Fields
    private const string CapacityField = "Capacity";
    internal const int DefaultCapacity = 0x10;
    internal char[] m_ChunkChars; -> 保存StringBuilder所管理着的字符串中的字符
    internal int m_ChunkLength;
    internal int m_ChunkOffset;
    internal StringBuilder m_ChunkPrevious;
    internal int m_MaxCapacity;
    private const string MaxCapacityField = "m_MaxCapacity";
    internal const int MaxChunkSize = 0x1f40;
    private const string StringValueField = "m_StringValue";
    private const string ThreadIDField = "m_currentThread";

    // Methods
    [TargetedPatchingOptOut("Performance critical to inline across NGen image boundaries"), __DynamicallyInvokable]
    public StringBuilder();
    [TargetedPatchingOptOut("Performance critical to inline across NGen image boundaries"), __DynamicallyInvokable]
    public StringBuilder(int capacity);
    [__DynamicallyInvokable, TargetedPatchingOptOut("Performance critical to inline this type of method across NGen im
```

经过对StringBuilder默认构造方法的分析，系统默认初始化m_ChunkChars的长度为16（0x10），当新追加进来的字符串长度与旧有字符串长度之和大于该字符数组容量时，新创建字符数组的容量会增加到2n+1（假如当前字符数组容量为2n）。

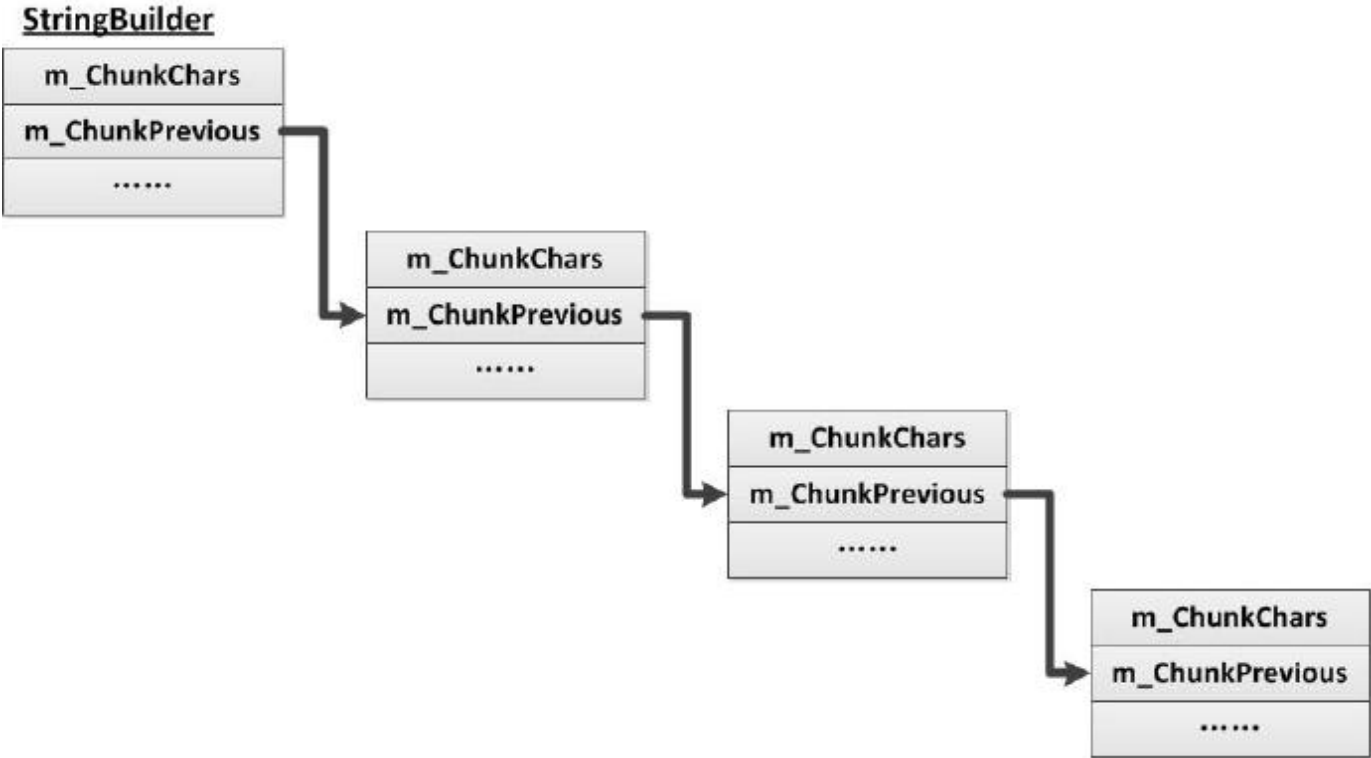
此外，StringBuilder内部还有一个同为StringBuilder类型的m_ChunkPrevious，它是内部的一个StringBuilder对象，前面提到当追加的字符串长度和旧字符串长度之合大于字符数组m_ChunkChars的最大容量时，会根据当前的（this）StringBuilder创建一个新的StringBuilder对象，将m_ChunkPrevious指向新创建的StringBuilder对象。

下面是StringBuilder中实现扩容的核心代码：

C#

```
private void ExpandByABlock(int minBlockCharCount)
{
    .....
    int num = Math.Max(minBlockCharCount, Math.Min(this.Length, 0x1f40));
    this.m_ChunkPrevious = new StringBuilder(this);
    this.m_ChunkOffset += this.m_ChunkLength;
    this.m_ChunkLength = 0;
    .....
    this.m_ChunkChars = new char[num];
}
```

可以看出，初始化m_ChunkPrevious在前，创建新的字符数组m_ChunkChars在后，最后才是复制字符到数组m_ChunkChars中（更新当前的m_ChunkChars）。归根结底，StringBuilder是在内部以字符数组m_ChunkChars为基础维护一个链表m_ChunkPrevious，该链表如下图所示：



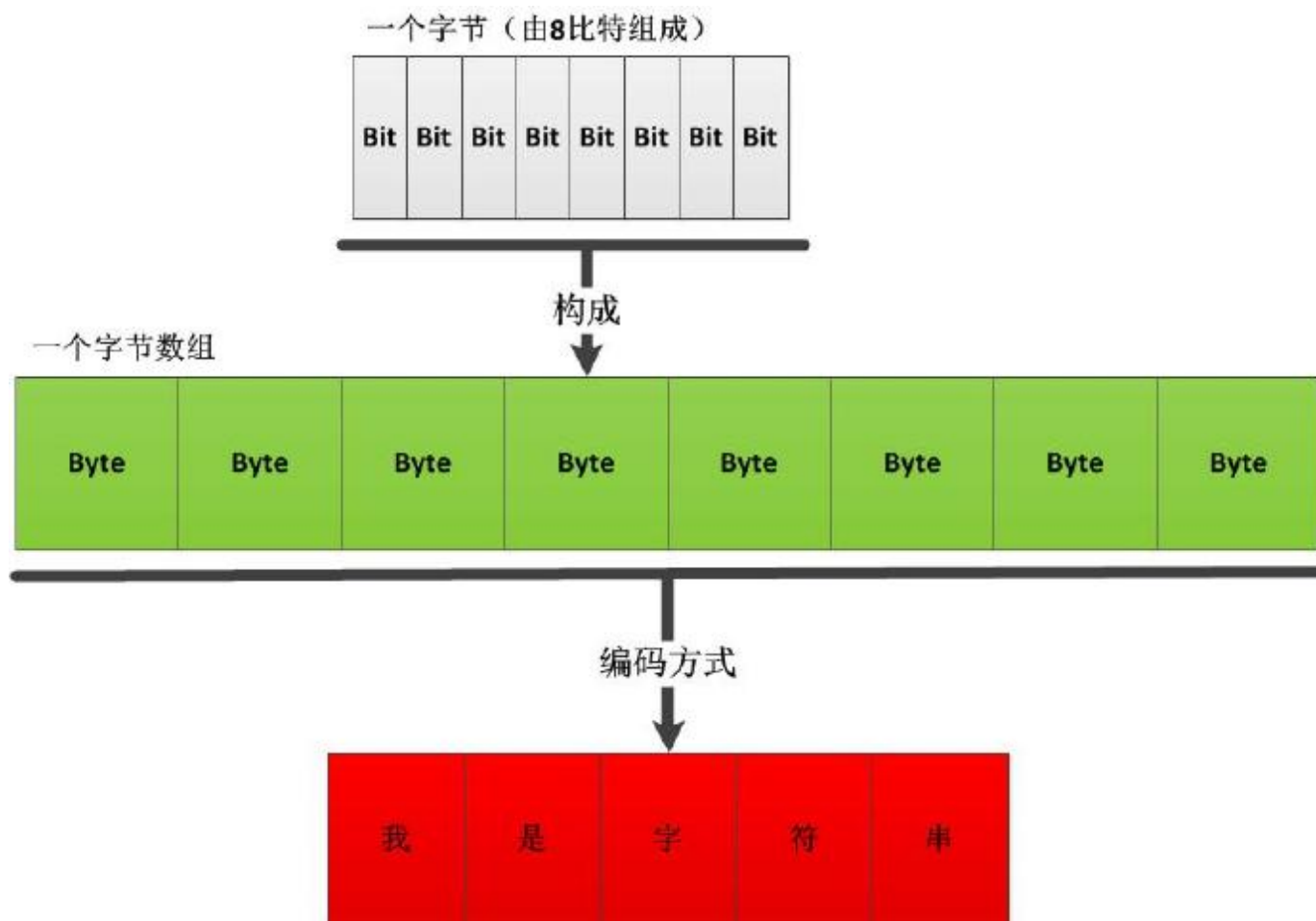
在最终的ToString方法中，当前的StringBuilder对象会根据这个链表以及记录的长度和偏移变量去生成最终的一个String对象实例，StringBuilder的内部实现中使用了一些指针操作，其内部原理有兴趣的园友可以自己通过反编译工具查看源代码。

1.2 String和Byte[]对象之间如何相互转换？

在实际开发中，经常会对数据进行处理，不可避免地会遇到字符串和字节数组相互转换的需求。字符串和字节数组的转换，事实上是代表了现实世界信息和数字世界信息之间的转换，要了解其中的机制，需要先对比特、直接以及编码这三个概念有所了解。

- （1）比特：bit是一个位，计算机内物理保存的最基本单元，一个bit就是一个二进制位；
- （2）字节：byte由8个bit构成，其值可以由一个0~255的整数表示；
- （3）编码：编码是数字信息和现实信息的转换机制，一种编码通常就定义了一种字符集和转换的原则，常用的编码方式包括UTF8、GB2312、Unicode等。

下图直观地展示了比特、字节、编码和字符串的关系：



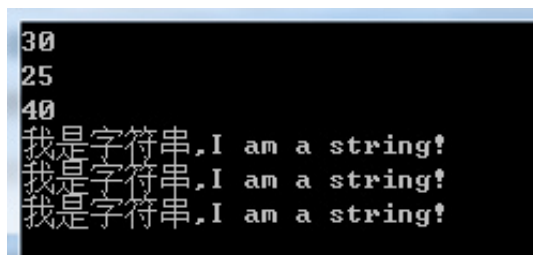
从上图可以看出，字节数组和字符串的转换必然涉及到某种编码方式，不同的编码方式由不同的转换结果。在C#中，可以使用System.Text.Encoding来管理常用的编码。

下面的代码展示了如何在字节数组和字符串之间进行转换（分别使用UTF8、GB2312以及Unicode三种编码方式）：

```
C#
class Program
{
    static void Main(string[] args)
    {
        string s = "我是字符串,I am a string!";
        // 字节数组 -> 字符串
        Byte[] utf8 = StringToByte(s, Encoding.UTF8);
        Byte[] gb2312 = StringToByte(s, Encoding.GetEncoding("GB2312"));
        Byte[] unicode = StringToByte(s, Encoding.Unicode);
        Console.WriteLine(utf8.Length);
        Console.WriteLine(gb2312.Length);
        Console.WriteLine(unicode.Length);
        // 字符串 -> 字符数组
        Console.WriteLine(ByteToString(utf8, Encoding.UTF8));
        Console.WriteLine(ByteToString(gb2312,
            Encoding.GetEncoding("GB2312")));
    }
}
```

```
Console.WriteLine(ByteToString(unicode, Encoding.Unicode));
Console.ReadKey();
}
// 字符串 -> 字节数组
static Byte[] StringToByte(string str, Encoding encoding)
{
    if (string.IsNullOrEmpty(str))
    {
        return null;
    }
    return encoding.GetBytes(str);
}
// 字节数组 -> 字符串
static string ByteToString(Byte[] bytes, Encoding encoding)
{
    if (bytes == null || bytes.Length <= 0)
    {
        return string.Empty;
    }
    return encoding.GetString(bytes);
}
}
```

上述代码的运行结果如下图所示:



我们也可以从上图中看出, 不同的编码方式产生的字节数组的长度各不相同。

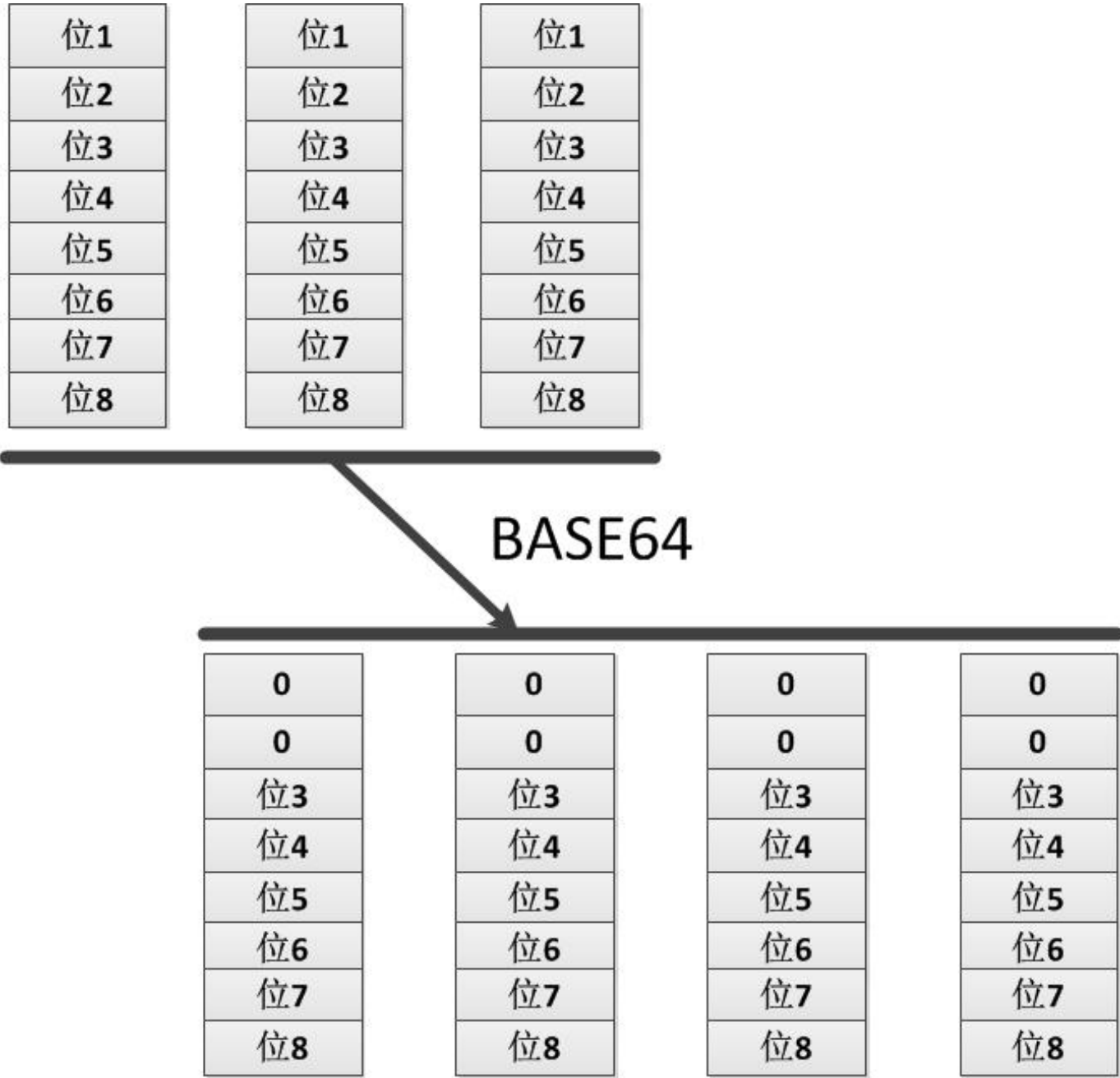
1.3 BASE64编码的作用以及C#中对其的支持

和传统的编码不同, BASE64编码的设计致力于混淆那些8位字节的数据流(解决网络传输中的明码问题), 在网络传输、邮件等系统中被广泛应用。需要明确的是: BASE64不属于加密机制, 但它却是把明码变成了一种很难识别的形式。

BASE64的算法如下:

BASE64把所有的位分开, 并且重新组合成字节, 新的字节只包含6位, 最后在每个字节前添加两个0, 组成了新的字节数组。例如: 一个字节数组只包含三个字节(每个字节又有8位比特), 对其进行BASE64编码时会将其分配到4个新的字节中(为什么是4个呢? 计算 $3 \times 8 / 6 = 4$), 其中每个字节只填充低6位, 最后把高2位置为零。

下图清晰地展示了上面所讲到的BASE64的算法示例：



在.NET中，BASE64编码的应用也很多，例如在ASP.NET WebForm中，默认为我们生成了一个ViewState来保持状态，如下图所示：


```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
</title></head>
<body>
  <form method="post" action="HttpNoStatus.aspx" id="form1">
    <div class="aspNetHidden">
      <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwUENTlZNTY4ODI3MWEkPm0lddUgcxPMeExp5WbHICw8Ja+u2yvns6V/PbXMEHdYI=" />
    </div>
    <div class="aspNetHidden">
      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION" value="/vEdAAKIOSAM1fQgruEkGhyUqq7bqeU3bDnfCS7nk18QqTDSkXz4DI" />
    </div>
    <div align="center">
      <span id="lblNumber">0</span>
      <input type="submit" name="btnGetNumber" value="Get Next" id="btnGetNumber" />
    </div>
  </form>
</body>
</html>
```

大隐隐于页的ViewState!

藏得可够深啊!

这里的ViewState其实就是服务器在返回给浏览器前进行了一次BASE64编码，我们可以通过一些解码工具进行反BASE64编码查看其中的奥秘：

The screenshot shows the ViewStateDecoder 2 application. On the left, the 'ViewState string:' field contains a Base64-encoded string. A red box highlights this string, with an arrow pointing to the 'Decode' button. On the right, the 'Tree Display' tab shows the decoded structure. A red box highlights a portion of the tree, showing a 'Pair' object with a 'String' property 'age' and an 'Int32' property '1'. Red arrows point from the text '刚刚在服务器端进行了 ViewState ["age"]=1' to these properties.

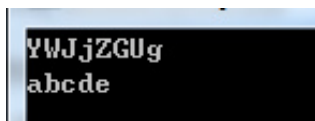
那么，问题来了？在.NET中开发中，怎样来进行BASE64的编码和解码呢，.NET基类库中提供了一个Convert类，其中有两个静态方法提供了BASE64的编码和解码，但要注意的是：Convert类型在转换失败时会直接抛出异常，我们需要在开发中注意对潜在异常的处理（比如使用is或as来进行高效的类型转换）。下面的代码展示了其用法：

```
C#
class Program
{
    static void Main(string[] args)
    {
        string test = "abcde ";
```



```
// 生成UTF8字节数组
byte[] bytes = Encoding.UTF8.GetBytes(test);
// 转换成Base64字符串
string base64 = BytesToBase64(bytes);
Console.WriteLine(base64);
// 转换回UTF8字节数组
bytes = Base64ToBytes(base64);
Console.WriteLine(Encoding.UTF8.GetString(bytes));
Console.ReadKey();
}
// Bytes to Base64
static string BytesToBase64(byte[] bytes)
{
    try
    {
        return Convert.ToBase64String(bytes);
    }
    catch
    {
        return null;
    }
}
// Base64 to Bytes
static Byte[] Base64ToBytes(string base64)
{
    try
    {
        return Convert.FromBase64String(base64);
    }
    catch
    {
        return null;
    }
}
```

上面代码的执行结果如下图所示:



1.4 简述SecureString安全字符串的特点和用法

也许很多人都是第一次知道还有SecureString这样一个类型,我也不例外。SecureString并不是一个常用的类型,但在一些拥有特殊需求的场合,它就会有很大的作用。顾名思义,SecureString意为安全

的字符串，它被设计用来保存一些机密的字符串，完成传统字符串所不能做到的工作。

(1) 传统字符串以明码的形式被分配在内存中，一个简单的内存读写软件就可以轻易地捕获这些字符串，而在这某些机密系统中是不被允许的。也许我们会觉得对字符串加密就可以解决类似问题，But，事实总是残酷的，对字符串加密时字符串已经以明码方式驻留在内存中很久了！对于该问题唯一的解决办法就是在字符串的获得过程中直接进行加密，SecureString的设计初衷就是解决该类问题。

(2) 为了保证安全性，SecureString是被分配在非托管内存上的（而普通String是被分配在托管内存中的），并且SecureString的对象从分配的一开始就以加密的形式存在，我们所有对于SecureString的操作（无论是增删查改）都是逐字符进行的。

逐字符机制：在进行这些操作时，驻留在非托管内存中的字符串就会被解密，然后进行具体操作，最后再进行加密。不可否认的是，在具体操作的过程中有小段时间字符串是处于明码状态的，但逐字符的机制让这段时间维持在非常短的区间内，以保证破解程序很难有机会读取明码的字符串。

(3) 为了保证资源释放，SecureString实现了标准的Dispose模式（Finalize+Dispose双管齐下，因为上面提到它是被分配到非托管内存中的），保证每个对象在作用域退出后都可以被释放掉。

内存释放方式：将其对象内存全部置为0，而不是仅仅告诉CLR这一块内存可以分配，当然这样做仍然是为了确保安全。熟悉C/C++的朋友可能就会很熟悉，这不就是 `memset` 函数干的事情嘛！下面这段C代码便使用了`memset`函数将内存区域置为0：

C#// 下面申请的20个字节的内存有可能被别人用过

```
char chs[20];
```

```
// memset内存初始化:memset(void *,要填充的数据,要填充的字节个数)
```

```
memset(chs,0,sizeof(chs));
```

看完了SecureString的原理，现在我们通过下面的代码来熟悉一下在.NET中的基本用法：

C#

```
using System;
```

```
using System.Runtime.InteropServices;
```

```
using System.Security;
```

```
namespace UseSecureString
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            // 使用using语句保证Dispose方法被及时调用
```

```
            using (SecureString ss = new SecureString())
```

```
            {
```

```
                // 只能逐字符地操作SecureString对象
```

```
                ss.AppendChar('e');
```

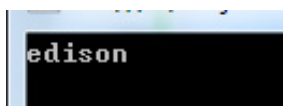
```
                ss.AppendChar('i');
```

```

        ss.AppendChar('s');
        ss.AppendChar('o');
        ss.AppendChar('n');
        ss.InsertAt(1, 'd');
        // 打印SecureString对象
        PrintSecureString(ss);
    }
    Console.ReadKey();
}
// 打印SecureString对象
public unsafe static void PrintSecureString(SecureString ss)
{
    char* buffer = null;
    try
    {
        // 只能逐字符地访问SecureString对象
        buffer = (char*)Marshal.SecureStringToCoTaskMemUnicode(ss);
        for (int i = 0; *(buffer + i) != '\0'; i++)
        {
            Console.Write(*(buffer + i));
        }
    }
    finally
    {
        // 释放内存对象
        if (buffer != null)
        {
            Marshal.ZeroFreeCoTaskMemUnicode((System.IntPtr)buffer);
        }
    }
}
}

```

其运行显示的结果很简单:



这里需要注意的是: 为了显示SecureString的内容, 程序需要访问非托管内存, 因此会用到指针, 而要在C#使用指针, 则需要使用unsafe关键字(前提是你项目属性中勾选了允许不安全代码, 对你没错, 指针在C#可以使用, 但是被认为是不安全的!). 此外, 程序中使用了Marshal.SecureStringToCoTaskMemUnicode方法来把安全字符串解密到非托管内存中, 最后就是就是

们不要忘记在使用非托管资源时需要确保及时被释放。

1.5 简述字符串驻留池机制

字符串具有不可变性，程序中对于同一个字符串的大量修改或者多个引用赋值同一字符串在理论上会产生大量的临时字符串对象，这会极大地降低系统的性能。对于前者，可以使用StringBuilder类型解决，而后者，.NET则提供了另一种不透明的机制来优化，这就是传说中的字符串驻留池机制。

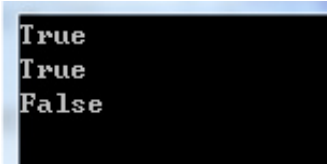
使用了字符串驻留池机制之后，当CLR启动时，会在内部创建一个容器，该容器内部维持了一个类似于key-value对的数据结构，其中key是字符串的内容，而value则是字符串在托管堆上的引用（也可以理解为指针或地址）。当一个新的字符串对象需要分配时，CLR首先监测内部容器中是否已经存在该字符串对象，如果已经包含则直接返回已经存在的字符串对象引用；如果不存在，则新分配一个字符串对象，同时把其添加到内部容器中取。But，这里有一个例外，就是当程序员用new关键字显示地申明新分配一个字符串对象时，该机制将不会起作用。

从上面的描述中，我们可以看到字符串驻留池的本质是一个缓存，内部维持了一个键为字符串内容，值为该字符串在堆中的引用地址的键值对数据结构。我们可以通过下面一段代码来加深对于字符串驻留池的理解：

C#

```
class Program
{
    static void Main(string[] args)
    {
        // 01. 两个字符串对象，理论上引用应该不相等
        // 但是由于字符串池机制，二者指向了同一对象
        string a = "abcde";
        string b = "abcde";
        Console.WriteLine(object.ReferenceEquals(a, b));
        // 02. 由于编译器的优化，所以下面这个c仍然指向了同一引用地址
        string c = "a" + "bc" + "de";
        Console.WriteLine(object.ReferenceEquals(a, c));
        // 03. 显示地使用new来分配内存，这时候字符串池不起作用
        char[] arr = { 'a', 'b', 'c', 'd', 'e' };
        string d = new string(arr);
        Console.WriteLine(object.ReferenceEquals(a, d));
        Console.ReadKey();
    }
}
```

在上述代码中，由于字符串驻留池机制的使用，变量a、b、c都指向了同一个字符串实例对象，而d则使用了new关键字显示申明，因此字符串驻留池并没有对其起作用，其运行结果如下图所示：



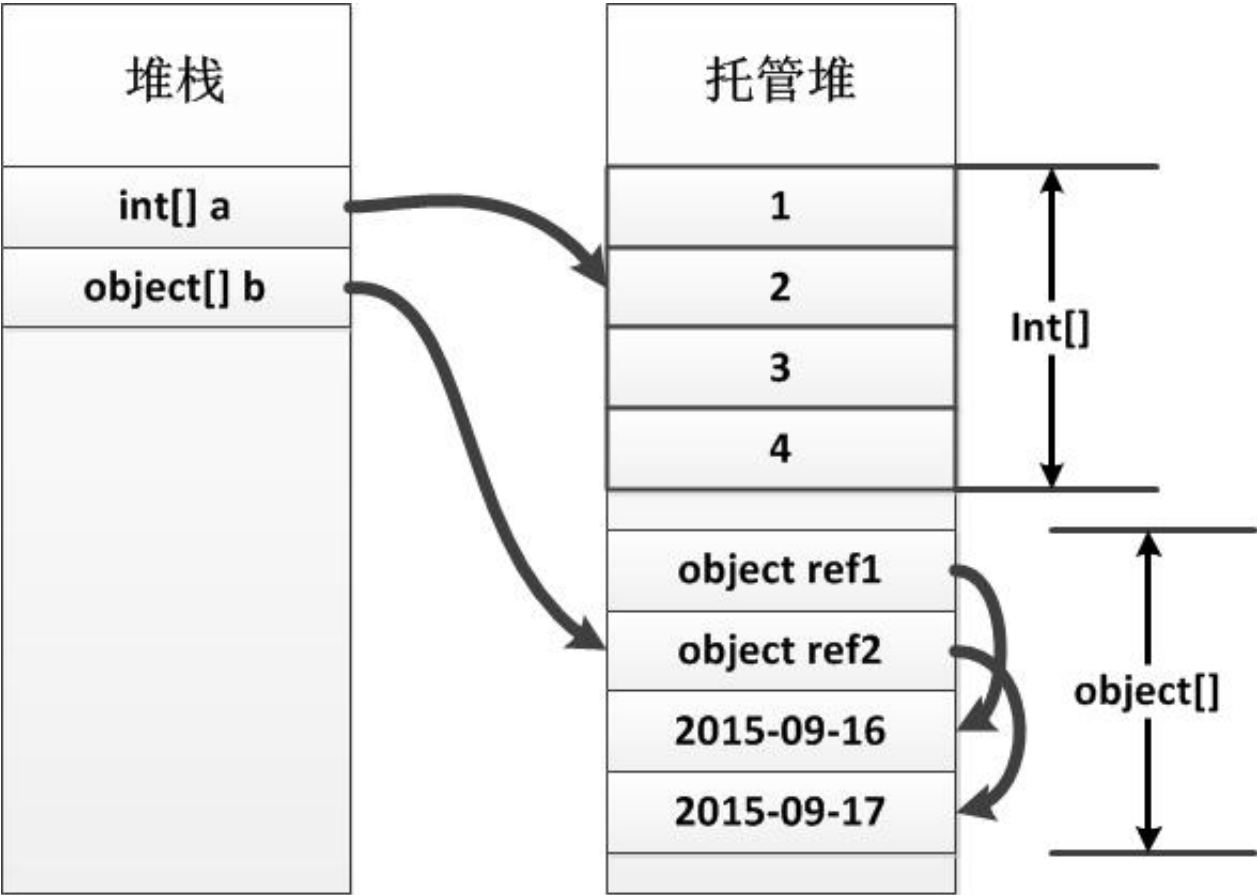
字符串驻留池的设计本意是为了改善程序的性能，因此在C#中默认是打开了字符串驻留池机制，But，.NET也为我们提供了字符串驻留池的开关接口，如果程序集标记了一个System.Runtime.CompilerServices.CompilationRelaxationsAttribute特性，并且指定了一个System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning标志，那么CLR不会采用字符串驻留池机制，其代码声明如下所示，但是我添加后一直没有尝试成功：

```
1
[assembly:
System.Runtime.CompilerServices.CompilationRelaxations(System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning)]
```

二、常用集合和泛型2.1 int[]是值类型还是引用类型？

在.NET中的数组类型和C++中区别很大，.NET中无论是存储值类型对象的数组还是存储引用类型的数组，其本身都是引用类型，其内存也都是分配在堆上的。它们的共同特征在于：所有的数组类型都继承自System.Array，而System.Array又实现了多个接口，并且直接继承自System.Object。不同之处则在于存储值类型对象的数组所有的值都已经包含在数组内，而存储引用类型对象的数组，其值则是一个引用，指向位于托管堆中的实例对象。

下图直观地展示了二者内存分配的差别（假设object[]中存储都是DateTime类型的对象实例）：



在.NET中CLR会检测所有对数组的访问，任何视图访问数组边界以外的代码都会产生一个IndexOutOfRangeException异常。

2.2 数组之间如何进行转换?

数组类型的转换需要遵循以下两个原则:

- (1) 包含值类型的数组不能被隐式转换成其他任何类型;
- (2) 两个数组类型能够相互转换的一个前提是两者维数相同;

我们可以通过以下代码来看看数组类型转换的机制:

C#// 编译成功

```
string[] sz = { "a", "a", "a" };
object[] oz = sz;
// 编译失败, 值类型的数组不能被转换
int[] sz2 = { 1, 2, 3 };
object[] oz2 = sz;
// 编译失败, 两者维数不同
string[,] sz3 = { { "a", "b" }, { "a", "c" } };
object[] oz3 = sz3;
```

除了类型上的转换, 我们平时还可能会遇到内容转换的需求。例如, 在一系列的用户界面操作之后, 系统的后台可能会得到一个DateTime的数组, 而现在的任务则是将它们存储到数据库中, 而数据库访问层提供的接口只接受String[]参数, 这时我们要做的就是将DateTime[]从内容上转换为String[]对象。当然, 惯常做法是遍历整个源数组, 逐一地转换每个对象并且将其放入一个目标数组类型容器中, 最后再生成目标数组。But, 这里我们推荐使用Array.ConvertAll方法, 它提供了一个简便的转换数组间内容的接口, 我们只需指定源数组的类型、对象数组的类型和具体的转换算法, 该方法就能高效地完成转换工作。

下面的代码清楚地展示了普通的数组内容转换方式和使用Array.ConvertAll的数组内容转换方式的區別:

C#

```
class Program
{
    static void Main(string[] args)
    {
        String[] times = {"2008-1-1",
                           "2008-1-2",
                           "2008-1-3"};

        // 使用不同的方法转换
        DateTime[] result1 = OneByOne(times);
        DateTime[] result2 = ConvertAll(times);
        // 结果是相同的
        Console.WriteLine("手动逐个转换的方法: ");
        foreach (DateTime item in result1)
        {
```

```
        Console.WriteLine(item.ToString("yyyy-MM-dd"));
    }
    Console.WriteLine("使用Array.Convert方法: ");
    foreach (DateTime item2 in result2)
    {
        Console.WriteLine(item2.ToString("yyyy-MM-dd"));
    }
    Console.ReadKey();
}
// 逐个手动转换
private static DateTime[] OneByOne(String[] times)
{
    List<DateTime> result = new List<DateTime>();
    foreach (String item in times)
    {
        result.Add(DateTime.Parse(item));
    }
    return result.ToArray();
}
// 使用Array.ConvertAll方法
private static DateTime[] ConvertAll(String[] times)
{
    return Array.ConvertAll(times,
        new Converter<String, DateTime>
            (DateTimeToString));
}
private static DateTime DateTimeToString(String time)
{
    return DateTime.Parse(time);
}
}
```

从上述代码可以看出，二者实现了相同的功能，但是Array.ConvertAll不需要我们手动地遍历数组，也不需要生成一个临时的容器对象，更突出的优势是它可以接受一个动态的算法作为具体的转换逻辑。当然，明眼人一看就知道，它是以一个委托的形式作为参数传入，这样的机制保证了Array.ConvertAll具有较高的灵活性。

2.3 简述泛型的基本原理

泛型的语法和概念类似于C++中的template（模板），它是.NET 2.0中推出的众多特性中最为重要的一个，方便我们设计更加通用的类型，也避免了容器操作中的装箱和拆箱操作。

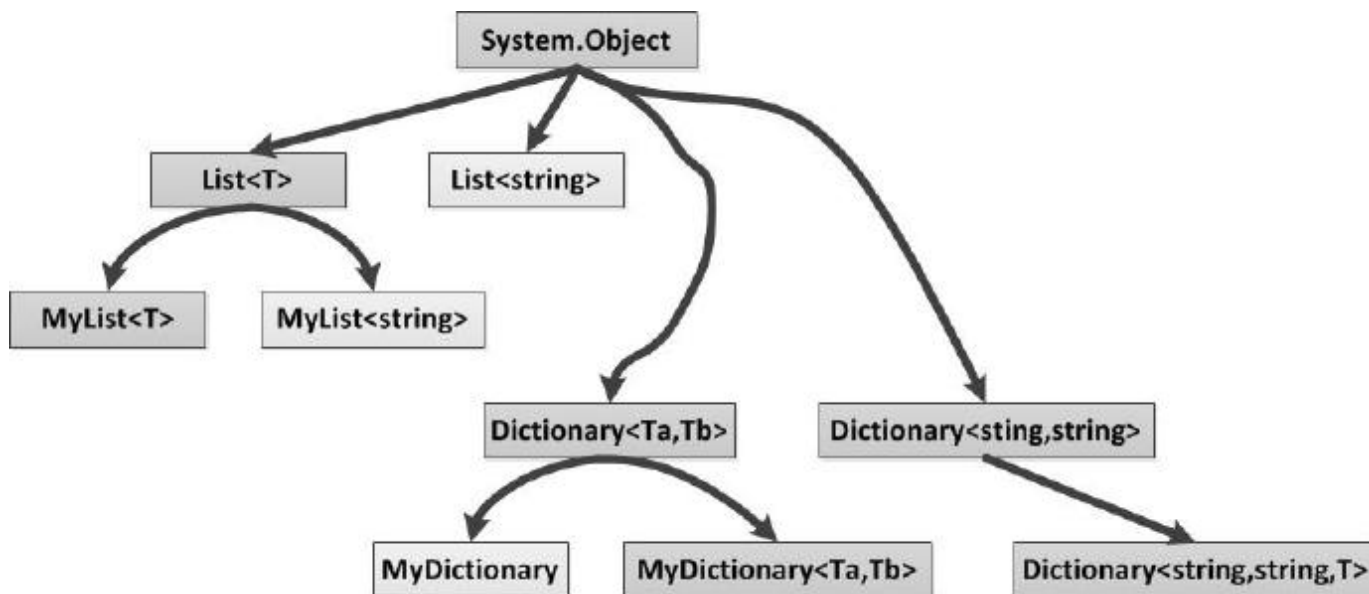
假如我们要实现一个排序算法，要求能够针对各种类型进行排序。按照以前的做法，我们需要对int、double、float等类型都实现一次，但是我们发现除了数据类型，其他的处理逻辑完全一致。这时，我们便可以考虑使用泛型来进行实现：

C#

```
public static class SortHelper<T> where T : IComparable
{
    public static void BubbleSort(T[] array)
    {
        int length = array.Length;
        for (int i = 0; i <= length - 2; i++)
        {
            for (int j = length - 1; j >= 1; j--)
            {
                // 对两个元素进行交换
                if (array[j].CompareTo(array[j - 1]) < 0)
                {
                    T temp = array[j];
                    array[j] = array[j - 1];
                    array[j - 1] = temp;
                }
            }
        }
    }
}
```

Tips: Microsoft在产品文档中建议所有的泛型参数名称都以T开头, 作为一个中编码的通用规范, 建议大家都能遵守这样的规范, 类似的规范还有所有的接口都以I开头。

泛型类型和普通类型有一定的区别, 通常泛型类型被称为开放式类型, .NET中规定开放式类型不能实例化, 这样也就确保了开放式类型的泛型参数在被指定前, 不会被实例化成任何对象 (事实上, .NET也没有办法确定到底要分配多少内存给开放式类型)。为开放式的类型提供泛型的实例导致了新的封闭类型的生成, 但这并不代表新的封闭类型和开放类型有任何继承关系, 它们在类结构图上是处于同一层次, 并且两者之间没有任何关系。下图展示了这一概念:



此外，在.NET中的System.Collections.Generic命名空间下提供了诸如List<T>、Dictionary<T>、LinkedList<T>等泛型数据结构，并且在System.Array中定义了一些静态的泛型方法，我们应该在编码实践时充分使用这些泛型容器，以提高我们的开发和系统的运行效率。

2.4 泛型的主要约束和次要约束是什么？

当一个泛型参数没有任何约束时，它可以进行的操作和运算是非常有限的，因为不能对实参进行任何类型上的保证，这时候就需要用到泛型约束。泛型的约束分为：主要约束和次要约束，它们都使实参必须满足一定的规范，C#编译器在编译的过程中可以根据约束来检查所有泛型类型的实参并确保其满足约束条件。

(1) 主要约束

一个泛型参数至多拥有一个主要约束，主要约束可以是一个引用类型、class或者struct。如果指定一个引用类型（class），那么实参必须是该类型或者该类型的派生类型。相反，struct则规定了实参必须是一个值类型。下面的代码展示了泛型参数主要约束：

C#

```
public class ClassT1<T> where T : Exception
{
    private T myException;
    public ClassT1(T t)
    {
        myException = t;
    }
    public override string ToString()
    {
        // 主要约束保证了myException拥有source成员
        return myException.Source;
    }
}

public class ClassT2<T> where T : class
{
    private T myT;
    public void Clear()
    {
        // T是引用类型，可以置null
        myT = null;
    }
}

public class ClassT3<T> where T : struct
{
    private T myT;
    public override string ToString()
```

```
{  
    // T是值类型, 不会发生NullReferenceException异常  
    return myT.ToString();  
}  
}
```

泛型参数有了主要约束后, 也就能够在类型中对其进行一定的操作了。

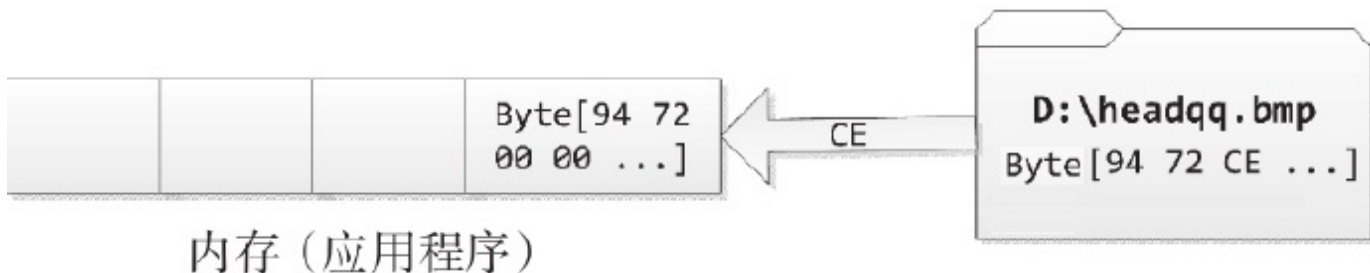
(2) 次要约束

次要约束主要是指实参实现的接口的限定。对于一个泛型, 可以有0到无限的次要约束, 次要约束规定了实参必须实现所有的次要约束中规定的接口。次要约束与主要约束的语法基本一致, 区别仅在于提供的不是一个引用类型而是一个或多个接口。例如我们为上面代码中的ClassT3增加一个次要约束:

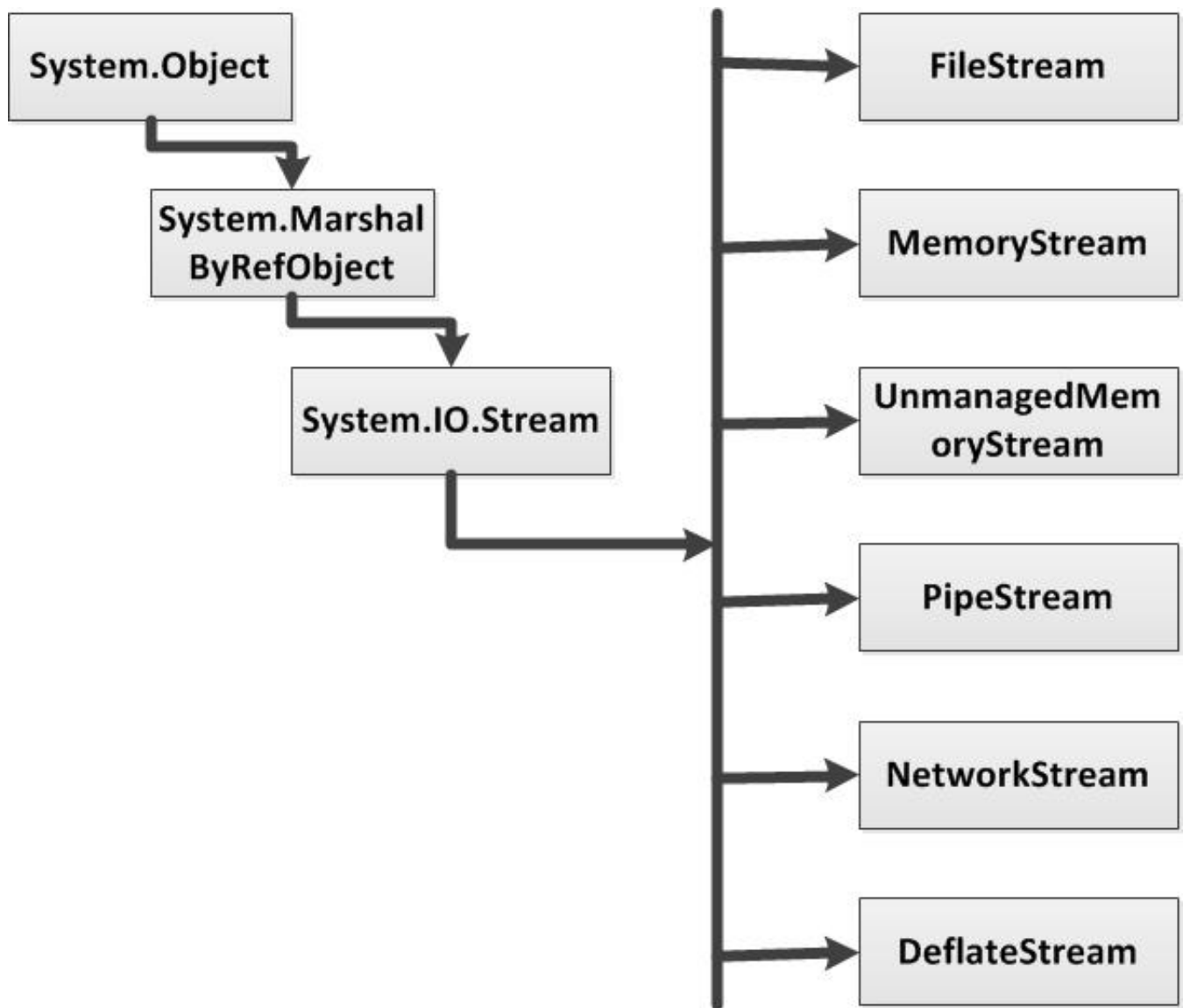
```
C#  
public class ClassT3<T> where T : struct, IComparable  
{  
    .....  
}
```

三、流和序列化3.1 流的概念以及.NET中有哪些常见的流?

流是一种针对字节流的操作, 它类似于内存与文件之间的一个管道。在对一个文件进行处理时, 本质上需要经过借助OS提供的API来进行打开文件, 读取文件中的字节流, 再关闭文件等操作, 其中读取文件的过程就可以看作是字节流的一个过程。



常见的流类型包括: 文件流、终端操作流以及网络Socket等, 在.NET中, System.IO.Stream类型被设计为作为所有流类型的虚基类, 所有的常见流类型都继承自System.IO.Stream类型, 当我们需要自定义一种流类型时, 也应该直接或者间接地继承自Stream类型。下图展示了在.NET中常见的流类型以及它们的类型结构:



从上图中可以发现，Stream类型继承自MarshalByRefObject类型，这保证了流类型可以跨越应用程序域进行交互。所有常用的流类型都继承自System.IO.Stream类型，这保证了流类型的同一性，并且屏蔽了底层的一些复杂操作，使用起来非常方便。

下面的代码中展示了如何在.NET中使用FileStream文件流进行简单的文件读写操作：

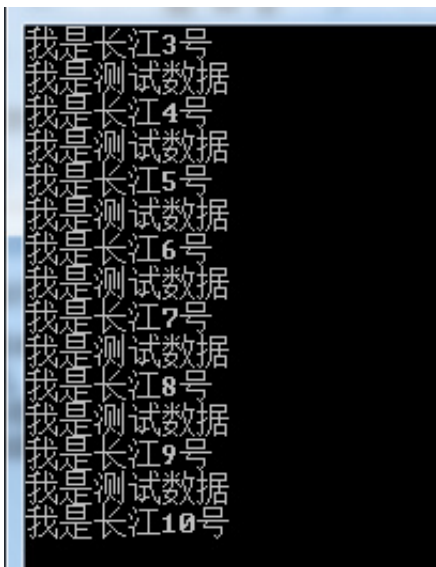
```
C#
class Program
{
    private const int bufferlength = 1024;
    static void Main(string[] args)
    {
        //创建一个文件，并写入内容
        string filename = @"C:\TestStream.txt";
        string filecontent = GetTestString();
        try
        {
            if (File.Exists(filename))
            {
```

```
        File.Delete(filename);
    }
    // 创建文件并写入内容
    using (FileStream fs = new FileStream(filename,
        FileMode.Create))
    {
        Byte[] bytes = Encoding.UTF8.GetBytes(filecontent);
        fs.Write(bytes, 0, bytes.Length);
    }
    // 读取文件并打印出来
    using (FileStream fs = new FileStream(filename,
        FileMode.Open))
    {
        Byte[] bytes = new Byte[bufferlength];
        UTF8Encoding encoding = new UTF8Encoding(true);
        while (fs.Read(bytes, 0, bytes.Length) > 0)
        {
            Console.WriteLine(encoding.GetString(bytes)
        );
        }
    }
    // 循环分批读取打印
    //using (FileStream fs = new FileStream(filename,
    FileMode.Open, FileAccess.Read))
    //{
    //    Byte[] bytes = new Byte[bufferlength];
    //    int bytesRead;
    //    do
    //    {
    //        bytesRead = fs.Read(bytes, 0,
    bufferlength);
    //        Console.WriteLine(Encoding.UTF8.GetString
    (bytes, 0, bytesRead));
    //    } while (bytesRead > 0);
    //}
    catch (IOException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadKey();
}
```

```
// 01. 取得测试数据
```

```
static string GetTestString()
{
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 10; i++)
    {
        builder.Append("我是测试数据\r\n");
        builder.Append("我是长江" + (i + 1) + "号\r\n");
    }
    return builder.ToString();
}
```

上述代码的执行结果如下图所示:



在实际开发中, 我们经常会遇到需要传递一个比较大的文件, 或者事先无法得知文件大小 (Length属性抛出异常), 因此也就不能创建一个尺寸正好合适的Byte[]数组, 此时只能分批读取和写入, 每次只读取部分字节, 直到文件尾。例如我们需要复制G盘中一个大小为4.4MB的mp3文件到C盘中去, 假设我们对大小超过2MB的文件都采用分批读取写入机制, 可以通过如下代码实现:

```
C#
class Program
{
    private const int BufferSize = 10240; // 10 KB
    public static void Main(string[] args)
    {
        string fileName = @"G:\My Musics\BlueMoves.mp3"; // Source 4.4 MB
        string copyName = @"C:\BlueMoves-Copy.mp3"; // Destination 4.4 MB
        using (Stream source = new FileStream(fileName, FileMode.Open,
        FileAccess.Read))
        {
            using (Stream target = new FileStream(copyName,
```

```

        FileMode.Create, FileAccess.Write))
        {
            byte[] buffer = new byte[BufferSize];
            int bytesRead;
            do
            {
                // 从源文件中读取指定的10K长度到缓存中
                bytesRead = source.Read(buffer, 0,
BufferSize);

                // 从缓存中写入已读取到的长度到目标文件中
                target.Write(buffer, 0, bytesRead);
            } while (bytesRead > 0);
        }
    }
    Console.ReadKey();
}
}

```

上述代码中，设置了缓存buffer大小为10K，即每次只读取10K的内容长度到buffer中，通过循环的多次读写和写入完成整个复制操作。

3.2 如何使用压缩流？

由于网络带宽的限制、硬盘内存空间的限制等原因，文件和数据的压缩是我们经常会遇到的一个需求。因此，.NET中提供了对于压缩和解压的支持：GZipStream类型和DeflateStream类型，它们位于System.IO.Compression命名空间下，且都继承于Stream类型（对文件压缩的本质其实是针对字节的操作，也属于一种流的操作），实现了基本一致的功能。

下面的代码展示了GZipStream的使用方法，DeflateStream和GZipStream的使用方法几乎完全一致：

```

C#
class Program
{
    // 缓存数组的长度
    private const int bufferSize = 1024;
    static void Main(string[] args)
    {
        string test = GetTestString();
        byte[] original = Encoding.UTF8.GetBytes(test);
        byte[] compressed = null;
        byte[] decompressed = null;
        Console.WriteLine("数据的原始长度是:{0}", original.LongLength);
        // 1. 进行压缩
        // 1.1 压缩进入内存流
        using (MemoryStream target = new MemoryStream())

```



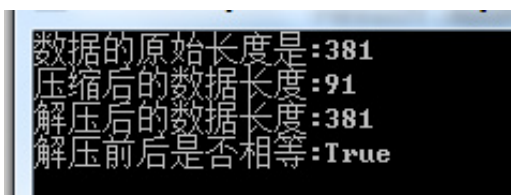
```
{
    using (GZipStream gzs = new GZipStream(target,
CompressionMode.Compress, true))
    {
        // 1.2 将数据写入压缩流
        WriteAllBytes(gzs, original, bufferSize);
    }
    compressed = target.ToArray();
    Console.WriteLine("压缩后的数据长度: {0}",
compressed.LongLength);
}
// 2. 进行解压缩
// 2.1 将解压后的数据写入内存流
using (MemoryStream source = new MemoryStream(compressed))
{
    using (GZipStream gzs = new GZipStream(source,
CompressionMode.Decompress, true))
    {
        // 2.2 从压缩流中读取所有数据
        decompressed = ReadAllBytes(gzs, bufferSize);
    }
    Console.WriteLine("解压后的数据长度: {0}",
decompressed.LongLength);
    Console.WriteLine("解压前后是否相等: {0}",
test.Equals(Encoding.UTF8.GetString(decompressed)));
}
Console.ReadKey();
}
// 01. 取得测试数据
static string GetTestString()
{
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 10; i++)
    {
        builder.Append("我是测试数据\r\n");
        builder.Append("我是长江" + (i + 1) + "号\r\n");
    }
    return builder.ToString();
}
// 02. 从一个流总读取所有字节
static Byte[] ReadAllBytes(Stream stream, int bufferlength)
{

```

```
Byte[] buffer = new Byte[bufferlength];
List<Byte> result = new List<Byte>();
int read;
while ((read = stream.Read(buffer, 0, bufferlength)) > 0)
{
    if (read < bufferlength)
    {
        Byte[] temp = new Byte[read];
        Array.Copy(buffer, temp, read);
        result.AddRange(temp);
    }
    else
    {
        result.AddRange(buffer);
    }
}
return result.ToArray();
}

// 03. 把字节写入一个流中
static void WriteAllBytes(Stream stream, Byte[] data, int bufferlength)
{
    Byte[] buffer = new Byte[bufferlength];
    for (long i = 0; i < data.LongLength; i += bufferlength)
    {
        int length = bufferlength;
        if (i + bufferlength > data.LongLength)
        {
            length = (int)(data.LongLength - i);
        }
        Array.Copy(data, i, buffer, 0, length);
        stream.Write(buffer, 0, length);
    }
}
```

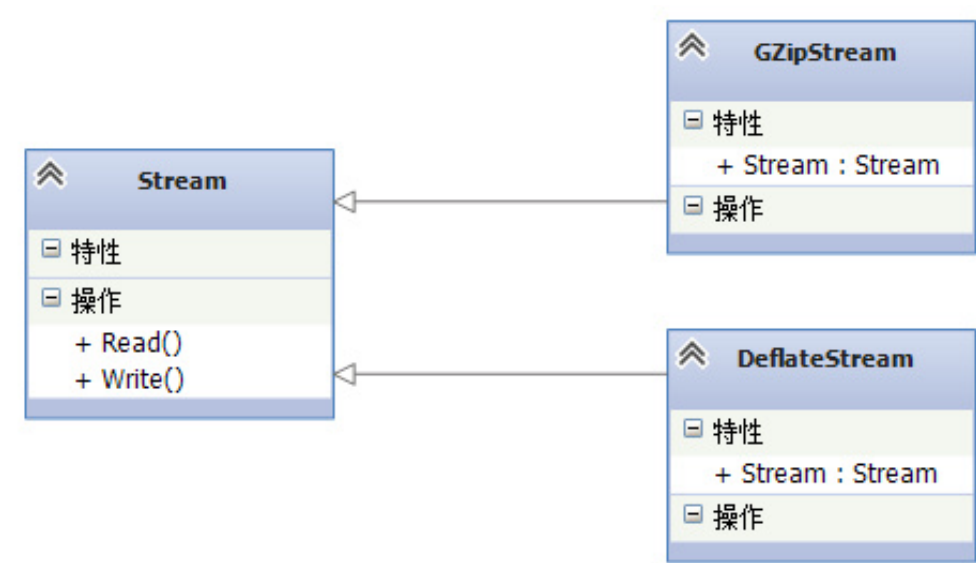
上述代码的运行结果如下图所示:



需要注意的是: 使用 GZipStream 类压缩大于 4 GB 的文件时将会引发异常。

通过GZipStream的构造方法可以看出, 它是一个典型的Decorator装饰者模式的应用, 所谓装饰者模式,

就是动态地给一个对象添加一些额外的职责。对于增加新功能这个方面，装饰者模式比新增一个之类更为灵活。就拿上面代码中的GZipStream来说，它扩展的是MemoryStream，为Write方法增加了压缩的功能，从而实现了压缩的应用。



扩展：许多资料表明.NET提供的GZipStream和DeflateStream类型的压缩算法并不出色，也不能调整压缩率，有些第三方的组件例如SharpZipLib实现了更高效的压缩和解压算法，我们可以在nuget中为项目添加该组件。

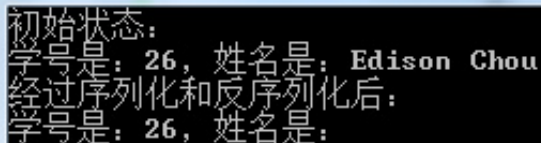


3.3 Serializable特性有什么作用？

通过上面的流类型可以方便地操作各种字节流，但是如何把现有的实例对象转换为方便传输的字节流，就需要使用序列化技术。对象实例的序列化，是指将实例对象转换为可方便存储、传输和交互的流。在.NET中，通过Serializable特性提供了序列化对象实例的机制，当一个类型被申明为Serializable后，它就能被诸如BinaryFormatter等实现了IFormatter接口的类型进行序列化和反序列化。

```
class Program
{
    static void Main(string[] args)
    {
        Person obj = new Person(26, "Edison Chou");
        Console.WriteLine("初始状态: ");
        Console.WriteLine(obj);
        // 序列化对象
        byte[] data = Serialize(obj);
        // 反序列化对象
        Person newObj = Deserialize(data);
        Console.WriteLine("经过序列化和反序列化后: ");
        Console.WriteLine(newObj);
        Console.ReadKey();
    }
    // 序列化对象
    static byte[] Serialize(Person p)
    {
        // 使用二进制序列化
        IFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream())
        {
            formatter.Serialize(ms, p);
            return ms.ToArray();
        }
    }
    // 反序列化对象
    static Person Deserialize(byte[] data)
    {
        // 使用二进制反序列化
        IFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream(data))
        {
            Person p = formatter.Deserialize(ms) as Person;
            return p;
        }
    }
}
```

上述代码的运行结果如下图所示:



```
初始状态:
学号是: 26, 姓名是: Edison Chou
经过序列化和反序列化后:
学号是: 26, 姓名是:
```

注意：当一个基类使用了Serializable特性后，并不意味着其所有子类都能被序列化。事实上，我们必须为每个子类都添加Serializable特性才能保证其能被正确地序列化。

3.4 .NET提供了哪几种可进行序列化操作的类型？

我们已经理解了如何把一个类型声明为可序列化的类型，但是万里长征只走了第一步，具体完成序列化和反序列化的操作还需要一个执行这些操作的类型。为了序列化具体实例到某种专用的格式，.NET中提供了三种对象序列格式化类型：BinaryFormatter、SoapFormatter和XmlSerializer。

(1) BinaryFormatter

顾名思义，BinaryFormatter可用于将可序列化的对象序列化成二进制的字节流，在前面Serializable特性的代码示例中已经展示过，这里不再重复展示。

(2) SoapFormatter

SoapFormatter致力于将可序列化的类型序列化成符合SOAP规范的XML文档以供使用。在.NET中，要使用SoapFormatter需要先添加对于SoapFormatter的引用：

```
C#
1
using System.Runtime.Serialization.Formatters.Soap;
```

Tips: SOAP是一种位于应用层的网络协议，它基于XML，并且是Web Service的基本协议。

(3) XmlSerializer

XmlSerializer并不仅仅针对那些标记了Serializable特性的类型，更为需要注意的是，Serializable和NonSerialized特性在XmlSerializer类型对象的操作中完全不起作用，取而代之的是XmlIgnore属性。XmlSerializer可以对没有标记Serializable特性的类型对象进行序列化，但是它仍然有一定的限制：

① 使用XmlSerializer序列化的对象必须显示地拥有一个无参数的公共构造方法；

因此，我们需要修改前面代码示例中的Person类，添加一个无参数的公共构造方法：

```
C#
static void Main(string[] args)
{
    Person obj = new Person(26, "Edison Chou", "CUIT");
    Console.WriteLine("原始对象为: ");
    Console.WriteLine(obj.ToString());
    // 使用SoapFormatter序列化对象
    byte[] data1 = SoapFormatter.Serialize(obj);
```

```

        Console.WriteLine("SoapFormatter序列化后: ");
        Console.WriteLine(Encoding.UTF8.GetString(data1));
        Console.WriteLine();
        // 使用XmlSerializer序列化对象
        byte[] data2 = XmlSerializer.Serialize(obj);
        Console.WriteLine("XmlSerializer序列化后: ");
        Console.WriteLine(Encoding.UTF8.GetString(data2));
        Console.ReadKey();
    }
}

```

示例运行结果如下图所示:



```

学号是: 26, 姓名是: Edison Chou, 大学是: CUIT
SoapFormatter序列化后:
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <a1:Person id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/UseSerializable/UseSerializable%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
      <_number>26</_number>
      <_univeristy id="ref-3">CUIT</_univeristy>
    </a1:Person>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

XmlSerializer序列化后:
<?xml version="1.0"?>
<Person xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <_name>Edison Chou</_name>
</Person>

```

3.5 如何自定义序列化和反序列化的过程?

对于某些类型, 序列化和反序列化往往有一些特殊的操作或逻辑检查需求, 这时就需要我们能够主动地控制序列化和反序列化的过程。 .NET中提供的Serializable特性帮助我们非常快捷地声明了一个可序列化的类型 (因此也就缺乏了灵活性), 但很多时候由于业务逻辑的要求, 我们需要主动地控制序列化和反序列化的过程。 因此, .NET提供了ISerializable接口来满足自定义序列化需求。

下面的代码展示了自定义序列化和反序列化的类型模板:

```

C#
[Serializable]
public class MyObject : ISerializable
{
    protected MyObject(SerializationInfo info, StreamingContext context)
    {

```

```
        // 在此构造方法中实现反序列化
    }

    public virtual void GetObjectData(SerializationInfo info, StreamingContext
context)
    {
        // 在此方法中实现序列化
    }
}
```

如上代码所示，GetObjectData和特殊构造方法都接收两个参数：SerializationInfo 类型参数的作用类似于一个哈希表，通过key/value对来存储整个对象的内容，而StreamingContext 类型参数则包含了流的当前状态，我们可以根据此参数来判断是否需要序列化和反序列化类型独享。

如果基类实现了ISerializable接口，则派生类需要针对自己的成员实现反序列化构造方法，并且重写基类中的GetObjectData方法。

下面通过一个具体的代码示例，来了解如何在.NET程序中自定义序列化和反序列化的过程：

①首先我们需要一个需要被序列化和反序列化的类型，该类型有可能被其他类型继承

C#

```
[Serializable]
public class MyObject : ISerializable
{
    private int _number;
    [NonSerialized]
    private string _name;
    public MyObject(int num, string name)
    {
        this._number = num;
        this._name = name;
    }
    public override string ToString()
    {
        return string.Format("整数是: {0}\r\n字符串是: {1}", _number,
_name);
    }
    // 实现自定义的序列化
    protected MyObject(SerializationInfo info, StreamingContext context)
    {
        // 从SerializationInfo对象（类似于一个HashTable）中读取内容
        this._number = info.GetInt32("MyObjectInt");
        this._name = info.GetString("MyObjectString");
    }
}
```


// 实现自定义的反序列化

```
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    // 将成员对象写入SerializationInfo对象中
    info.AddValue("MyObjectInt", this._number);
    info.AddValue("MyObjectString", this._name);
}
}
```

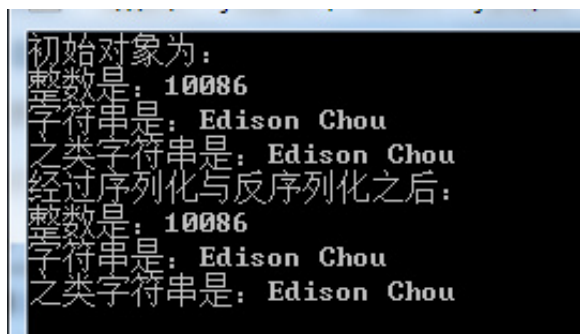
②随后编写一个继承自MyObject的子类，并添加一个私有的成员变量。需要注意的是：子类必须负责序列化和反序列化自己添加的成员变量。

C#

```
class Program
{
    static void Main(string[] args)
    {
        MyObjectSon obj = new MyObjectSon(10086, "Edison Chou");
        Console.WriteLine("初始对象为: ");
        Console.WriteLine(obj.ToString());
        // 序列化
        byte[] data = Serialize(obj);
        Console.WriteLine("经过序列化与反序列化之后: ");
        Console.WriteLine(DeSerialize(data));
        Console.ReadKey();
    }
    // 序列化对象-BinaryFormatter
    static byte[] Serialize(MyObject p)
    {
        // 使用二进制序列化
        IFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream())
        {
            formatter.Serialize(ms, p);
            return ms.ToArray();
        }
    }
    // 反序列化对象-BinaryFormatter
    static MyObject DeSerialize(byte[] data)
    {
        // 使用二进制反序列化
        IFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream(data))
```

```
{  
  
    MyObject p = formatter.Deserialize(ms) as MyObject;  
    return p;  
}  
  
}
```

上述代码的运行结果如下图所示:



从结果图中可以看出, 由于实现了自定义的序列化和反序列化, 从而原先使用Serializable特性的默认序列化和反序列化算法没有起作用, MyObject类型的所有成员经过序列化和反序列化之后均被完整地还原了, 包括申明了NonSerialized特性的成员。

参考资料

- (1) 朱毅, 《进入IT企业必读的200个.NET面试题》
- (2) 张子阳, 《.NET之美: .NET关键技术深入解析》
- (3) 王涛, 《你必须知道的.NET》

合作联系

Email: bd@jobbole.com

QQ: 2302462408 (加好友请注明来意)

更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 分享和发现有价值的内容与观点
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI, 网页, 交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享

