```
String s = "Hi" + " " + "there.";
```

在上述代码中，由于所有字符串都是字面值，所以 C#编译器能在编译时连接它们，最终只将一个字符串(即**"Hi there."**)放到模块的元数据中。对非字面值字符串使用+操作符，连接则在运行时进行。运行时连接不要使用+操作符，因为这样会在堆上创建多个字符串对象，而堆是需要垃圾回收的，对性能有影响。相反，应该使用 **System.Text.StringBuilder** 类型(本章稍后详细解释)。

最后，C#提供了一种特殊的字符串声明方式。采取这种方式，引号之间的所有字符会都被视为字符串的一部分。这种特殊声明称为“逐字字符串”(verbatim string)，通常用于指定文件或目录的路径，或者与正则表达式配合使用。以下代码展示了如何使用和不使用逐字字符串字符(@)来声明同一个字符串：

```
// 指定应用程序路径
String file = "C:\\Windows\\System32\\Notepad.exe";

// 使用逐字字符串指定应用程序路径
String file = @"C:\Windows\System32\Notepad.exe";
```

两种写法在程序集的元数据中生成完全一样的字符串，但后者可读性更好。在字符串之前添加@符号使编译器知道这是逐字字符串。编译器会将反斜杠字符视为字面值(literal)而非转义符，使文件路径在源代码中更易读。

了解了如何构造字符串之后，接着探讨可以在 **String** 对象上执行的操作。

14.2.2 字符串是不可变的

String 对象最重要的一点就是不可变(immutable)。也就是说，字符串一经创建便不能更改，不能变长、变短或修改其中的任何字符。使字符串不可变有几方面的好处。首先，它允许在一个字符串上执行各种操作，而不实际地更改字符串：

```
if (s.ToUpperInvariant().Substring(10, 21).EndsWith("EXE")) {
  ...
}
```

ToUpperInvariant 返回一个新字符串；它没有修改字符串 **s** 的字符。在 **ToUpperInvariant** 返回的字符串上执行的 **Substring** 操作也返回新字符串。然后，**EndsWith** 对这个字符串进行检查。代码不会长时间引用由 **ToUpperInvariant** 和 **Substring** 创建的两个临时字符串，垃圾回收器会在下次回收时回收它们的内存。如果执行大量字符串操作，会在堆上创建大量 **String** 对象，造成更频繁的垃圾回收，从而影响应用程序性能。要高效执行大量字符串操作，建议使用 **StringBuilder** 类。

字符串不可变还意味着在操纵或访问字符串时不会发生线程同步问题。此外，CLR 可通过一个 **String** 对象共享多个完全一致的 **String** 内容。这样能减少系统中的字符串数量——从而节省内存——这就是所谓的“字符串留用”(string interning)[①]。

① MSDN 文档将 interning 翻译成“拘留”，专门供字符串留用的表称为“拘留池”。本书采用“留用”这一译法。这个技术的详情将在本章后面详细解释。——译注

出于对性能的考虑，**String** 类型与 CLR 紧密集成。具体地说，CLR 知道 **String** 类型中定义的字段如何布局，会直接访问这些字段。但为了获得这种性能和直接访问的好处，**String** 只能是密封类。换言之，不能把它作为自己类型的基类。如果允许 **String** 作为基类来定义自己的类型，就能添加自己的字段，而这会破坏 CLR 对于 **String** 类型的各种预设。此外，还可能破坏 CLR 团队因为 **String** 对象“不可变”而做出的各种预设。

14.2.3　比较字符串

“比较”或许是最常见的字符串操作。一般因为两个原因要比较字符串：判断相等性或者排序(通常是为了显示给用户看)。

判断字符串相等性或排序时，强烈建议调用 **String** 类定义的以下方法之一：

```
Boolean Equals(String value, StringComparison comparisonType)
static Boolean Equals(String a, String b, StringComparison comparisonType)

static Int32 Compare(String strA, String strB, StringComparison comparisonType)
static Int32 Compare(string strA, string strB, Boolean ignoreCase, CultureInfo culture)
static Int32 Compare(String strA, String strB, CultureInfo culture, CompareOptions options)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   StringComparison comparisonType)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   CultureInfo culture, CompareOptions options)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   Boolean ignoreCase, CultureInfo culture)

Boolean StartsWith(String value, StringComparison comparisonType)
Boolean StartsWith(String value,
   Boolean ignoreCase, CultureInfo culture)

Boolean EndsWith(String value, StringComparison comparisonType)
Boolean EndsWith(String value, Boolean ignoreCase, CultureInfo culture)
```

排序时应该总是执行区分大小写的比较。原因是假如只是大小写不同的两个字符串被视为相等，那么每次排序都可能按不同顺序排列，用户会感到困惑。

comparisonType 参数(上述大多数方法都有)要求获取由 **StringComparison** 枚举类型定义的某个值。该枚举类型的定义如下所示：

```
public enum StringComparison {
  CurrentCulture = 0,
  CurrentCultureIgnoreCase = 1,
  InvariantCulture = 2,
  InvariantCultureIgnoreCase = 3,
  Ordinal = 4,
  OrdinalIgnoreCase = 5
}
```

另外，前面有两个方法要求传递一个 **options** 参数，它是 **CompareOptions** 枚举类型定义的值之一：

```
[Flags]
public enum CompareOptions {
  None = 0,
  IgnoreCase = 1,
```

```
  IgnoreNonSpace = 2,
  IgnoreSymbols = 4,
  IgnoreKanaType = 8,
  IgnoreWidth = 0x00000010,
  Ordinal = 0x40000000,
  OrdinalIgnoreCase = 0x10000000,
  StringSort = 0x20000000
}
```

接受 **CompareOptions** 实参的方法要求显式传递语言文化。传递 **Ordinal** 或 **OrdinalIgnoreCase** 标志，这些 **Compare** 方法会忽略指定的语言文化。

许多程序都将字符串用于内部编程目的，比如路径名、文件名、URL、注册表项/值、环境变量、反射、XML 标记、XML 特性等。这些字符串通常只在程序内部使用，不向用户显示。出于编程目的而比较字符串时，应该总是使用 **StringComparison.Ordinal** 或者 **StringComparison.OrdinalIgnoreCase**。忽略语言文化是字符串比较最快的方式。

另一方面，要以语言文化正确的方式来比较字符串(通常为了向用户显示)，就应该使用 **StringComparison.CurrentCulture** 或者 **StringComparison.CurrentCultureIgnoreCase**。



重要提示 **StringComparison.InvariantCulture** 和 **StringComparison. InvariantCultureIgnoreCase** 平时最好不要用。虽然这两个值能保证比较时的语言文化正确性，但用来比较内部编程所需的字符串，所花的时间远超出序号比较[①]。此外，如果传递 **StringComparison.InvariantCulture**(固定语言文化)，其实就是不使用任何具体的语言文化。所以在处理要向用户显示的字符串时，选择它并不恰当。



重要提示 要在序号比较前更改字符串中的字符的大小写，应该使用 **String** 的 **ToUpperInvariant** 或 **ToLowerInvariant** 方法。强烈建议用 **ToUpperInvariant** 方法对字符串进行正规化(normalizing)，而不要用 **ToLowerInvariant**，因为 Microsoft 对执行大写比较的代码进行了优化。事实上，执行不区分大小写的比较之前，FCL 会自动将字符串正规化为大写形式。之所以要用 **ToUpperInvariant** 和 **ToLowerInvariant** 方法，是因为 **String** 类没有提供 **ToUpperOrdinal** 和 **ToLowerOrdinal** 方法。之所以不用 **ToUpper** 和 **ToLower** 方法，是因为它们对语言文化敏感。

以语言文化正确的方式比较字符串时，有时需要指定另一种语言文化，而不是使用与调用线程关联的那一种。这时可用前面列出的 **StartsWith**，**EndsWith** 和 **Compare** 方法的重载版本，它们都接受 **Boolean** 和 **CultureInfo** 参数。

① 传递 **StringComparison.Ordinal** 执行的就是序号比较，也就是不考虑语言文化信息，只比较字符串中的每个 **Char** 的 Unicode 码位。——译注



重要提示　除了前面列出的之外，**String** 类型还为 **Equals**，**StartsWith**，**EndsWith** 和 **Compare** 方法定义了其他几个重载版本。但是，Microsoft 建议避免使用这些额外的版本(也就是本书没有列出的版本)。除此之外，**String** 的其他比较方法——**CompareTo(IComparable** 接口所要求的)、**CompareOrdinal** 以及==和!=操作符——也应避免使用。之所以要避免使用这些方法和操作符，是因为调用者不显式指出以什么方式执行字符串比较，而你无法从方法名看出默认比较方式。例如，**CompareTo** 默认执行对语言文化敏感的比较，而 **Equals** 执行普通的序号(ordinal)比较。如果总是显式地指出以什么方式执行字符串比较，代码将更容易阅读和维护。

现在重点讲一下如何执行语言文化正确的比较。.NET Framework 使用 **System.Globalization.CultureInfo** 类型表示一个“语言/国家”对(根据 RFC 1766 标准)。例如，“en-US”代表美国英语，“en-AU”代表澳大利亚英语，而“de-DE”代表德国德语。在 CLR 中，每个线程都关联了两个特殊属性，每个属性都引用一个 **CultureInfo** 对象。两个属性的具体描述如下。

- **CurrentUICulture 属性**　该属性获取要向用户显示的资源。它在 GUI 或 Web 窗体应用程序中特别有用，因为它标识了在显示 UI 元素(比如标签和按钮)时应使用的语言。创建线程时，这个线程属性会被设置成一个默认的 **CultureInfo** 对象，该对象标识了正在运行应用程序的 Windows 版本所用的语言。而这个语言是用 Win32 函数 **GetUserDefaultUILanguage** 来获取的。如果应用程序在 Windows 的 MUI(多语言用户界面，Multilingual User Interface)版本上运行，可通过控制面板的“区域和语言”对话框来修改语言。在非 MUI 版本的 Windows 上，语言由安装的操作系统的本地化版本(或者安装的语言包)决定，而且这个语言不可更改。

- **CurrentCulture 属性**　不适合使用 **CurrentUICulture** 属性的场合就使用该属性，例如数字和日期格式化、字符串大小写转换以及字符串比较。格式化要同时用到 **CultureInfo** 对象的“语言”和“国家”部分。创建线程时，这个线程属性被设为一个默认的 **CultureInfo** 对象，其值通过调用 Win32 函数 **GetUserDefaultLCID** 来获取。可通过 Windows 控制面板的“区域和语言”对话框来修改这个值。

在许多计算机上，线程的 **CurrentUICulture** 和 **CurrentCulture** 属性都被设为同一个 **CultureInfo** 对象。也就是说，它们使用相同的“语言/国家信息。但也可把它们设为不同对象。例如，在美国运行的一个应用程序可能要用西班牙语来显示它的所有菜单项以及其他 GUI 元素，同时仍然要正确显示美国的货币和日期格式。为此，线程的 **CurrentUICulture** 属性要引用一个 **CultureInfo** 对象，该对象使用“es”(代表西班牙语)初始化。线程的 **CurrentCulture** 属性要引用另一个 **CultureInfo** 对象，用“en-US”初始化。

CultureInfo 对象内部的一个字段引用了一个 **System.Globalization.CompareInfo** 对象，该对象封装了语言文化的字符排序表信息(根据 Unicode 标准的定义)。以下代码演示了序号比

较和对语言文化敏感的比较的区别：

```
using System;
using System.Globalization;

public static class Program {
  public static void Main() {
    String s1 = "Strasse";
    String s2 = "Straße";
    Boolean eq;

    // Compare 返回非零值①
    eq = String.Compare(s1, s2, StringComparison.Ordinal) == 0;
    Console.WriteLine("Ordinal comparison: '{0}' {2} '{1}'", s1, s2,
      eq ? "==" : "!=");

    // 面向在德国(DE)说德语(de)的人群，
    // 正确地比较字符串
    CultureInfo ci = new CultureInfo("de-DE");

    // Compare 返回零值
    eq = String.Compare(s1, s2, true, ci) == 0;
    Console.WriteLine("Cultural comparison: '{0}' {2} '{1}'", s1, s2,
      eq ? "==" : "!=");
  }
}
```

生成并运行上述代码得到以下输出：

```
Ordinal comparison: 'Strasse' != 'Straße'
Cultural comparison: 'Strasse' == 'Straße'
```



注意 **Compare** 方法如果执行的不是序号比较就会进行“字符展开”(character expansion)，也就是将一个字符展开成忽视语言文化的多个字符。在前例中，德语 Eszet 字符“ß”总是展开成“ss”。类似地，“Æ”连字总是展开成“AE”。所以在上述代码中，无论传递什么语言文化，对 **Compare** 的第二个调用始终返回 0。

比较字符串以判断相等性或执行排序时，偶尔需要更多的控制。例如，比较日语字符串时就可能有这个需要。额外的控制通过 **CultureInfo** 对象的 **CompareInfo** 属性获得。前面说过，**CompareInfo** 对象封装了一种语言文化的字符比较表，每种语言文化只有一个 **CompareInfo** 对象。

调用 **String** 的 **Compare** 方法时，如果调用者指定了语言文化，就使用指定的语言文化；如果没有指定，就使用调用线程的 **CurrentCulture** 属性值。**Compare** 方法内部会获取与特定语言文化匹配的 **CompareInfo** 对象引用，并调用 **CompareInfo** 对象的 **Compare** 方法，传递恰当的选项(比如不区分大小写)。自然，如果需要额外的控制，可以自己调用一个特定 **CompareInfo** 对象的 **Compare** 方法。

---

① **Compare** 返回 **Int32** 值：非零表示不相等，零表示相等。非零和零分别对应 **true** 和 **false**，所以后面的代码将比较结果与 **0(false)**进行比较，将 **true** 或 **false** 结果赋给 **eq** 变量。——译注

CompareInfo 类型的 **Compare** 方法获取来自 **CompareOptions** 枚举类型[①]的一个值作为参数。该枚举类型定义的符号代表一组位标志(bit flag)，对这些位标志执行 OR 运算就能更全面地控制字符串比较。请参考文档获取这些符号的完整描述。

以下代码演示了语言文化对于字符串排序的重要性，并展示了执行字符串比较的各种方式：

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.Threading;

public static class Program {
   public static void Main() {
      String output = String.Empty;
      String[] symbol = new String[] { "<", "=", ">" };
      Int32 x;
      CultureInfo ci;

      // 以下代码演示了在不同语言文化中，
      // 字符串的比较方式也有所不同
      String s1 = "coté";
      String s2 = "côte";

      // 为法国法语排序字符串
      ci = new CultureInfo("fr-FR");
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine;

      // 为日本日语排序字符串
      ci = new CultureInfo("ja-JP");
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine;

      // 为当前线程的当前语言文化排序字符串
      ci = Thread.CurrentThread.CurrentCulture;
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine + Environment.NewLine;

      // 以下代码演示了如何将 CompareInfo.Compare 的
      // 高级选项应用于两个日语字符串。
      // 一个字符串代表用平假名写成的单词“shinkansen”(新干线)：
      // 另一个字符串代表用片假名写成的同一个单词
      s1 = "しんかんせん";  //("\u3057\u3093\u304B\u3093\u305b\u3093")
      s2 = "シンカンセン";  //("\u30b7\u30f3\u30ab\u30f3\u30bb\u30f3")

      // 以下是默认比较结果
      ci = new CultureInfo("ja-JP");
      x = Math.Sign(String.Compare(s1, s2, true, ci));
```

① 前面已经展示过 **CompareOptions** 枚举类型。

```
        output += String.Format("Simple {0} Compare: {1} {3} {2}",
           ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;

        // 以下是忽略日语假名的比较结果
        CompareInfo compareInfo = CompareInfo.GetCompareInfo("ja-JP");
         x = Math.Sign(compareInfo.Compare(s1, s2, CompareOptions.IgnoreKanaType));
        output += String.Format("Advanced {0} Compare: {1} {3} {2}",
           ci.Name, s1, s2, symbol[x + 1]);

        MessageBox.Show(output, "Comparing Strings For Sorting");
    }
}
```

生成并运行以上代码得到如图 14-1 所示的结果。

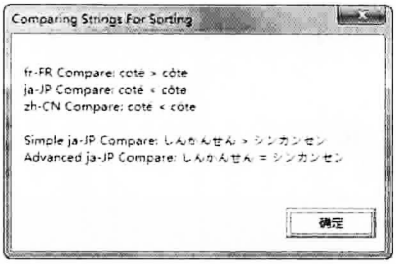


图 14-1　字符串排序结果



注意[①]　源代码不要用 ANSI 格式保存，否则日语字符会丢失。要在 Microsoft Visual Studio 中保存这个文件，请打开“另存文件为”对话框，单击“保存”按钮右侧的下箭头，选择“编码保存”，并选择“Unicode (UTF-8 带签名)–代码页 65001”。Microsoft C#编译器用这个代码页就能成功解析源代码文件了。

除了 **Compare**，**CompareInfo** 类还提供了 **IndexOf**，**LastIndexOf**，**IsPrefix** 和 **IsSuffix** 方法。由于所有这些方法都提供了接受 **CompareOptions** 枚举值的重载版本，所以能提供比 **String** 类定义的 **Compare**，**IndexOf**，**LastIndexOf**，**StartsWith** 和 **EndsWith** 方法更全面的控制。另外，FCL 的 **System.StringComparer** 类也能执行字符串比较，它适合对大量不同的字符串反复执行同一种比较。

14.2.4　字符串留用

如上一节所述，检查字符串相等性是应用程序的常见操作，也是一种可能严重损害性能的操作。执行序号(ordinal)相等性检查时，CLR 快速测试两个字符串是否包含相同数量的字符。答案否定，字符串肯定不相等；答案肯定，字符串则可能相等。然后，CLR 必须比较

① 中文版 Visual Studio 可忽略这个“注意”。——译注

每个单独的字符才能最终确认。而执行对语言文化敏感的比较时，CLR 必须比较所有单独的字符，因为两个字符串即使长度不同也可能相等。

此外，在内存中复制同一个字符串的多个实例纯属浪费，因为字符串是“不可变”(immutable)的。在内存中只保留字符串的一个实例将显著提升内存的利用率。需要引用字符串的所有变量只需指向单独一个字符串对象。

如果应用程序经常对字符串进行区分大小写的序号比较，或者事先知道许多字符串对象都有相同的值，就可利用 CLR 的**字符串留用**(string interning)机制来显著提升性能。CLR 初始化时会创建一个内部哈希表。在这个表中，键(key)是字符串，而值(value)是对托管堆中的 **String** 对象的引用。哈希表最开始是空的(理应如此)。**String** 类提供了两个方法，便于你访问这个内部哈希表：

```
public static String Intern(String str);
public static String IsInterned(String str);
```

第一个方法 **Intern** 获取一个 **String**，获得它的哈希码，并在内部哈希表中检查是否有相匹配的。如果存在完全相同的字符串，就返回对现有 **String** 对象的引用。如果不存在完全相同的字符串，就创建字符串的副本，将副本添加到内部哈希表中，返回对该副本的引用。如果应用程序不再保持对原始 **String** 对象的引用，垃圾回收器就可释放那个字符串的内存。注意垃圾回收器不能释放内部哈希表引用的字符串，因为哈希表正在容纳对它们的引用。除非卸载 AppDomain 或进程终止，否则内部哈希表引用的 **String** 对象不能被释放。

和 **Intern** 方法一样，**IsInterned** 方法也获取一个 **String**，并在内部哈希表中查找它。如果哈希表中有匹配的字符串，**IsInterned** 就返回对这个留用(interned)字符串对象的引用。但如果没有，**IsInterned** 会返回 **null**，不会将字符串添加到哈希表中。

程序集加载时，CLR 默认留用程序集的元数据中描述的所有字面值(literal)字符串。Microsoft 知道可能因为额外的哈希表查找而显著影响性能，所以现在能禁用此功能。如果程序集用 **System.Runtime.CompilerServices.CompilationRelaxationsAttribute** 进行了标记，并指定了 **System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning** 标志值，那么根据 ECMA 规范，CLR 可能选择不留用那个程序集的元数据中定义的所有字符串。注意，为了提升应用程序性能，C#编译器在编译程序集时总是指定上述两个特性和标志。

即使程序集指定了这些特性和标志，CLR 也可能选择对字符串进行留用，但不要依赖 CLR 的这个行为。事实上，除非显式调用 **String** 的 **Intern** 方法，否则永远都不要以“字符串已留用”为前提来写代码。以下代码演示了字符串留用：

```
String s1 = "Hello";
String s2 = "Hello";
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 显示'False'

s1 = String.Intern(s1);
s2 = String.Intern(s2);
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 显示'True'
```

在第一个 **ReferenceEquals** 方法调用中，**s1** 引用堆中的**"Hello"**字符串对象，而 **s2** 引用堆中

的另一个**"Hello"**对象。由于引用不同，所以应该显示 **False**。但在 CLR 的 4.5 版本上运行，实际显示的是 **True**。这是由于这个版本的 CLR 选择忽视 C#编译器插入的特性和标志。程序集加载到 AppDomain 中时，CLR 对字面值(literal)字符串**"Hello"**进行留用，结果是 **s1** 和 **s2** 引用堆中的同一个**"Hello"**字符串。但如前所述，你的代码永远不要依赖这个行为，因为未来版本的 CLR 有可能会重视这些特性和标志，从而不对**"Hello"**字符串进行留用。事实上，使用 NGen.exe 实用程序编译这个程序集的代码，CLR 的 4.5 版本确实会使用这些特性和标志。

在第二个 **ReferenceEquals** 方法调用之前，**"Hello"**字符串被显式留用，**s1** 现在引用已留用的**"Hello"**。然后，通过再次调用 **Intern**，**s2** 引用和 **s1** 一样的**"Hello"**字符串。所以第二个 **ReferenceEquals** 调用保证结果是 **True**，无论程序集在编译时是否设置了特性和标志。

现在通过一个例子理解如何利用字符串留用来提升性能并减少内存耗用。以下 **NumTimesWordAppearsEquals** 方法获取两个参数：一个单词和一个字符串数组。每个数组元素都引用了一个单词。方法统计指定单词在单词列表中出现了多少次并返回计数：

```
private static Int32 NumTimesWordAppearsEquals(String word, String[] wordlist) {
  Int32 count = 0;
  for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
     if (word.Equals(wordlist[wordnum], StringComparison.Ordinal))
       count++;
  }
  return count;
}
```

这个方法调用了 **String** 的 **Equals** 方法，后者在内部比较字符串的各个单独字符，并核实所有字符都匹配。这个比较可能很慢。此外，**wordlist** 数组可能有多个元素引用了含有相同字符内容的不同 **String** 对象。这意味着堆中可能存在多个内容完全相同的字符串，并会在将来进行垃圾回收时幸存下来。

现在看一下这个方法的另一个版本。新版本利用了字符串留用：

```
private static Int32 NumTimesWordAppearsIntern(String word,String[] wordlist) {
  // 这个方法假定 wordlist 中的所有数组元素都引用已留用的字符串
  word = String.Intern(word);
  Int32 count = 0;
  for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
    if (Object.ReferenceEquals(word, wordlist[wordnum]))
      count++;
  }
  return count;
}
```

这个方法留用了单词，并假定 **wordlist** 包含对已留用字符串的引用。首先，假如一个单词在单词列表中多次出现，这个版本有助于节省内存。因为在这个版本中，**wordlist** 会包含对堆中同一个 **String** 对象的多个引用。其次，这个版本更快。因为比较指针就知道指定单词是否在数组中。

虽然 **NumTimesWordAppearsIntern** 方法本身比 **NumTimesWordAppearsEquals** 方法快，但在使用 **NumTimesWordAppearsIntern** 方法时，应用程序的总体性能是可能变慢的。这是因为所有字符串在添加到 **wordlist** 数组时(这里没有列出具体的代码)都要花时间进行留

用。但如果应用程序要为同一个 **worldlist** 多次调用 **NumTimesWordAppearsIntern**，这个方法真的能提升应用程序性能和内存利用率。总之，字符串留用虽然有用，但使用须谨慎。事实上，这正是 C#编译器默认不想启用字符串留用的原因。[①]

14.2.5　字符串池

编译源代码时，编译器必须处理每个字面值(literal)字符串，并在托管模块的元数据中嵌入。同一个字符串在源代码中多次出现，把它们都嵌入元数据会使生成的文件无谓地增大。

为了解决这个问题，许多编译器(包括 C#编译器)只在模块的元数据中只将字面值字符串写入一次。引用该字符串的所有代码都被修改成引用元数据中的同一个字符串。编译器将单个字符串的多个实例合并成一个实例，能显著减少模块的大小。但这并不是新技术，C/C++编译器多年来一直在采用这个技术(Microsoft 的 C/C++编译器称之为“字符串池”)。尽管如此，字符串池仍是提升字符串性能的另一种行之有效的方式，而你应注意到它的存在。

14.2.6　检查字符串中的字符和文本元素

虽然字符串比较对于排序或测试相等性很有用，但有时只是想检查一下字符串中的字符。**String** 类型为此提供了几个属性和方法，包括 **Length**，**Chars**(一个 C#索引器[②])，**GetEnumerator**，**ToCharArray**，**Contains**，**IndexOf**，**LastIndexOf**，**IndexOfAny** 和 **LastIndexOfAny**。

System.Char 实际代表一个 16 位 Unicode 码值，而且该值不一定就等于一个抽象 Unicode 字符。例如，有的抽象 Unicode 字符是两个码值的组合。U+0625(阿拉伯字母 Alef with Hamza below)和 U+0650(Arabic Kasra)字符组合起来就构成了一个抽象字符或者**文本元素**(text element)。

除此之外，有的 Unicode 文本元素要求用两个 16 位值表示。第一个称为“高位代理项”(high surrogate)，第二个称为“低位代理项”(low surrogate)。其中，高位代理项范围在 U+D800 到 U+DBFF 之间，低位代理项范围在 U+DC00 到 U+DFFF 之间。有了代理项，Unicode 就能表示 100 万个以上不同的字符。

美国和欧洲很少使用代理项，东亚各国则很常用。为了正确处理文本元素，应当使用 **System.Globalization.StringInfo** 类型。使用这个类型最简单的方式就是构造它的实例，向构造器传递一个字符串。然后可以查询 **StringInfo** 的 **LengthInTextElements** 属性来了解字符串中有多少个文本元素。接着就可以调用 **StringInfo** 的 **SubstringByTextElements** 方法来提取所有文本元素，或者提取指定数量的连续文本元素。

StringInfo 类还提供了静态方法 **GetTextElementEnumerator**，它返回一个 **System.Globalization.**

---

① 虽然编译器应用了特性并设置了不进行字符串留用的标志，但 CLR 选择忽视这些设置你也没有办法。——译注

② 或者说有参属性。——译注

TextElementEnumerator 对象，允许枚举字符串中包含的所有抽象 Unicode 字符。最后，可调用 **StringInfo** 的静态方法 **ParseCombiningCharacters** 来返回一个 **Int32** 数组。从数组长度就能可以字符串包含多少个文本元素。每个数组元素都是一个文本元素的起始码值索引。

以下代码演示了使用 **StringInfo** 类来处理字符串中的文本元素的各种方式：

```
using System;
using System.Text;
using System.Globalization;
using System.Windows.Forms;

public sealed class Program {
  public static void Main() {
    // 以下字符串包含组合字符
    String s = "a\u0304\u0308bc\u0327";
    SubstringByTextElements(s);
    EnumTextElements(s);
    EnumTextElementIndexes(s);
  }

  private static void SubstringByTextElements(String s) {
    String output = String.Empty;

    StringInfo si = new StringInfo(s);
    for (Int32 element = 0; element < si.LengthInTextElements; element++) {
      output += String.Format(
        "Text element {0} is '{1}'{2}",
        element, si.SubstringByTextElements(element, 1),
        Environment.NewLine);
    }
    MessageBox.Show(output, "Result of SubstringByTextElements");
  }

  private static void EnumTextElements(String s) {
    String output = String.Empty;

    TextElementEnumerator charEnum =
       StringInfo.GetTextElementEnumerator(s);
    while (charEnum.MoveNext()) {
      output += String.Format(
        "Character at index {0} is '{1}'{2}",
        charEnum.ElementIndex, charEnum.GetTextElement(),
        Environment.NewLine);
    }
    MessageBox.Show(output, "Result of GetTextElementEnumerator");
  }

  private static void EnumTextElementIndexes(String s) {
    String output = String.Empty;

    Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);
    for (Int32 i = 0; i < textElemIndex.Length; i++) {
      output += String.Format(
        "Character {0} starts at index {1}{2}",
        i, textElemIndex[i], Environment.NewLine);
    }
    MessageBox.Show(output, "Result of ParseCombiningCharacters");
  }
}
```

编译并运行上述代码，会显示如图 14-2、图 14-3 和图 14-4 所示的对话框。

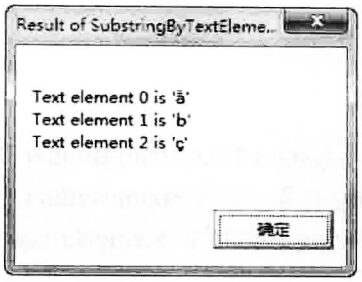


图 14-2　SubstringByTextElements 的结果

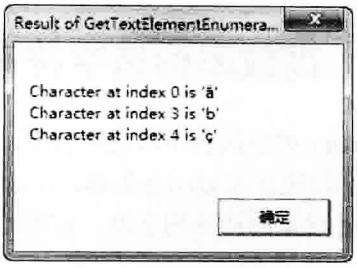


图 14-3　GetTextElementEnumerator 的结果

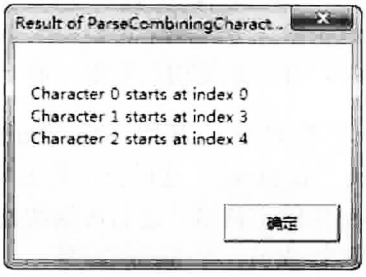


图 14-4　ParseCombiningCharacters 的结果

14.2.7　其他字符串操作

还可利用 **String** 类型提供的一些方法来复制整个字符串或者它的一部分。表 14-1 总结了这些方法。

表 14-1　用于复制字符串的方法

成员名称	方法类型	说明
Clone	实例	返回对同一个对象(**this**)的引用。能这样做是因为 **String** 对象不可变(immutable)。该方法实现了 **String** 的 **ICloneable** 接口
Copy	静态	返回指定字符串的新副本。该方法很少用，它的存在只是为了帮助一些需要把字符串当作 token 来对待的应用程序。通常，包含相同字符内容的多个字符串会被“留用”(intern)为单个字符串。该方法创建新字符串对象，确保即使字符串包含相同字符内容，引用 (指针)也有所不同
CopyTo	实例	将字符串中的部分字符复制到一个字符数组中
Substring	实例	返回代表原始字符串一部分的新字符串
ToString	实例	返回对同一个对象(**this**)的引用

除了这些方法，**String** 还提供了多个用于处理字符串的静态方法和实例方法，比如 **Insert**，**Remove**，**PadLeft**，**Replace**，**Split**，**Join**，**ToLower**，**ToUpper**，**Trim**，**Concat**，**Format** 等。使用所有这些方法时都请牢记一点，它们返回的都是新的字符串对象。这是由于字符串是不可变的。一经创建，便不能修改(使用安全代码的话)。

14.3 高效率构造字符串

由于 **String** 类型代表不可变字符串，所以 FCL 提供了 **System.Text.StringBuilder** 类型对字符串和字符进行高效动态处理，并返回处理好的 **String** 对象。可将 **StringBuilder** 想像成创建 **String** 对象的特殊构造器。你的方法一般应获取 **String** 参数而非 **StringBuilder** 参数。

从逻辑上说，**StringBuilder** 对象包含一个字段，该字段引用了由 **Char** 结构构成的数组。可利用 **StringBuilder** 的各个成员来操纵该字符数组，高效率地缩短字符串或更改字符串中的字符。如果字符串变大，超过了事先分配的字符数组大小，**StringBuilder** 会自动分配一个新的、更大的数组，复制字符，并开始使用新数组。前一个数组被垃圾回收。

用 StringBuilder 对象构造好字符串后，调用 StringBuilder 的 ToString 方法即可将 StringBuilder 的字符数组“转换”成 String。这样会在堆上新建 String 对象，其中包含调用 ToString 时存在于 StringBuilder 中的字符串。之后可继续处理 StringBuilder 中的字符串。以后可再次调用 ToString 把它转换成另一个 String 对象。

14.3.1 构造 StringBuilder 对象

和 **String** 类不同，CLR 不觉得 **StringBuilder** 类有什么特别。此外，大多数语言(包括 C#)都不将 **StringBuilder** 类视为基元类型。要像构造其他任何非基元类型那样构造 **StringBuilder** 对象：

```
StringBuilder sb = new StringBuilder();
```

StringBuilder 类型提供了许多构造器。每个构造器的职责是分配和初始化由每个 **StringBuilder** 对象维护的状态。下面解释了 **StringBuilder** 类的关键概念。

- **最大容量**
  一个 **Int32** 值，指定了能放到字符串中的最大字符数。默认值是 **Int32.MaxValue**(约 20 亿)。一般不用更改这个值，但有时需要指定较小的最大容量以确保永远不会创建超出特定长度的字符串。构造好之后，这个 **StringBuilder** 的最大容量就固定下来了，不能再变。

- **容量**
  一个 **Int32** 值，指定了由 **StringBuilder** 维护的字符数组的长度。默认为 **16**。如果事先知道要在这个 **StringBuilder** 中放入多少字符，那么构造 **StringBuilder** 对象时应该自己设置容量。
  向字符数组追加字符时，**StringBuilder** 会检测数组会不会超过设定的容量。如果会，**StringBuilder** 会自动倍增容量字段，用新容量来分配新数组，并将原始数组的字符复制到新数组中。随后，原始数组可以被垃圾回收。数组动态扩容会损害性能。要避免就要设置一个合适的初始容量。

- **字符数组**

  一个由 **Char** 结构构成的数组，负责维护“字符串”的字符内容。字符数总是小于或等于“容量”和“最大容量”值。可用 **StringBuilder** 的 **Length** 属性来获取数组中已经使用的字符数。**Length** 总是小于或等于 **StringBuilder** 的“容量”值。可在构造 **StringBuilder** 时传递一个 **String** 来初始化字符数组。不传递字符串，数组刚开始不包含任何字符——也就是说，**Length** 属性返回 **0**。

14.3.2　StringBuilder 的成员

和 **String** 不同，**StringBuilder** 代表可变(mutable)字符串。也就是说，**StringBuilder** 的大多数成员都能更改字符数组的内容，同时不会造成在托管堆上分配新对象。**StringBuilder** 只有以下两种情况才会分配新对象。

- 动态构造字符串，其长度超过了设置的“容量”。
- 调用 **StringBuilder** 的 **ToString** 方法。

表 14-2 总结了 **StringBuilder** 的成员。

表 14-2　StringBuilder 的成员

成员名称	成员类型	说明
MaxCapacity	只读属性	返回字符串能容纳的最大字符数(最大容量)
Capacity	可读/可写属性	获取或设置字符数组的长度(容量)。将容量设得比字符串长度小或者比 **MaxCapacity** 大将抛出 **ArgumentOutOfRangeException** 异常
EnsureCapacity	方法	保证字符数组至少具有指定的长度(容量)。如果传给方法的值大于 **StringBuilder** 的当前容量，当前容量会自动增大。如果当前容量已大于传给该属性的值，则不发生任何变化
Length	可读/可写属性	获取或设置“字符串”中的字符数。它可能小于字符数组的当前容量。将这个属性设为 **0**，会将 **StringBuilder** 的内容重置为空字符串
ToString	方法	这个方法的无参版本返回代表 **StringBuilder** 的字符数组的一个 **String**
Chars	可读/可写索引器属性	获取或设置字符数组指定索引位置的字符。在 C#中，这是一个索引器(有参属性)，要用数组语法([ ])来访问
Clear	方法	清除 **StringBuilder** 对象的内容，等同于把它的 **Length** 属性设为 **0**
Append	方法	在字符数组末尾追加一个对象；如有必要，数组会进行扩充。使用常规格式和与调用线程关联的语言文化将对象转换成字符串
Insert	方法	在字符数组中插入一个对象；如有必要，数组会进行扩充。使用常规格式和与调用线程关联的语言文化将对象转换成字符串
AppendFormat	方法	在字符数组末尾追加指定的零个或多个对象；如有必要，数组会进行扩充。使用调用者提供的格式化和语言文化信息，将这些对象转换成字符串。**AppendFormat** 是处理 **StringBuilder** 对象时最常用的方法之一

续表

成员名称	成员类型	说明
AppendLine	方法	在字符数组末尾追加一个行中止符或者一个带行中止符的字符串；如有必要，会增大数组的容量
Replace	方法	将字符数组中的一个字符替换成另一个字符，或者将一个字符串替换成另一个字符串
Remove	方法	从字符数组中删除指定范围的字符
Equals	方法	只有两个 **StringBuilder** 对象具有相同最大容量、字符数组容量和字符内容才返回 **true**
CopyTo	方法	将 **StringBuilder** 的字符内容的一个子集复制到一个 **Char** 数组中

使用 **StringBuilder** 的方法要注意，大多数方法返回的都是对同一个 **StringBuilder** 对象的引用，所以几个操作能连接到一起完成：

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").
  Replace(' ', '-').Remove(4, 3).ToString();
Console.WriteLine(s);      // 显示"Jeff-Richter"
```

String 和 **StringBuilder** 类提供的方法并不完全对应。例如，**String** 提供了 **ToLower**，**ToUpper**，**EndsWith**，**PadLeft**，**PadRight**，**Trim** 等方法；但 **StringBuilder** 类没有提供任何与之对应的方法。另一方面，**StringBuilder** 类提供了功能更全面的 **Replace** 方法，允许替换作为字符串一部分的字符或者子字符串(而不是一定要替换整个字符串)。由于两个类不完全对应，所以有时需要在 **String** 和 **StringBuilder** 之间转换以完成特定任务。例如，要构造字符串将所有字符转换成大写，再在其中插入一个子字符串，需要写下面这样的代码：

```
// 构造一个 StringBuilder 来执行字符串操作
StringBuilder sb = new StringBuilder();

// 使用 StringBuilder 来执行一些字符串操作
sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").Replace(" ", "-");

// 将 StringBuilder 转换成 String，以便将所有字符转换成大写
String s = sb.ToString().ToUpper();

// 清除 StringBuilder(分配新的 Char 数组)
sb.Length = 0;

// 将全部字符大写的 String 加载到 StringBuilder 中，执行其他操作
sb.Append(s).Insert(8, "Marc-");

// 将 StringBuilder 转换回 String
s = sb.ToString();

// 向用户显示 String
Console.WriteLine(s);      // "JEFFREY-Marc-RICHTER"
```

仅仅因为 **StringBuilder** 没有提供 **String** 提供的所有操作就要像这样写代码，显然是不方便的，效率也很低。希望 Microsoft 将来能为 **StringBuilder** 添加更多的字符串操作，进一步完善这个类。

14.4　获取对象的字符串表示：ToString

经常都要获取对象的字符串表示。例如，可能需要向用户显示数值类型(比如 **Byte**，**Int32** 和 **Single**)或者 **DateTime** 对象。由于.NET Framework 是面向对象的平台，每个类型都有责任提供代码将实例的值转换成字符串表示。FCL 的设计者为此规划了统一的模式。本节将描述这个模式。

可调用 **ToString** 方法获取任何对象的字符串表示。**System.Object** 定义了一个 **public**、**virtual**、无参的 **ToString** 方法，所以在任何类型的实例上都能调用该方法。在语义上，**ToString** 返回代表对象当前值的字符串，该字符串应根据调用线程当前的语言文化进行格式化。例如，在数字的字符串表示中，应该使用与调用线程的语言文化关联的小数点符号、数字分组符号和其他元素。

System.Object 实现的 **ToString** 只是返回对象所属类型的全名。这个值用处不大，但对许多不能提供有意义的字符串的类型来说，这也是一个合理的默认值。例如，一个 **FileStream** 或 **Hashtable** 对象的字符串表示应该是什么呢？

任何类型要想提供合理的方式获取对象当前值的字符串表示，就应重写 **ToString** 方法。FCL 内建的许多核心类型(**Byte**，**Int32**，**UInt64**，**Double** 等)都重写了 **ToString**，能返回符合语言文化的字符串。在 Visual Studio 调试器中，鼠标移到变量上方会出现一条数据提示(datatip)。提示中的文本正是通过调用对象的 **ToString** 方法来获取的。所以，定义类时应该总是重写 **ToString** 方法，以提供良好的调试支持。

14.4.1　指定具体的格式和语言文化

无参 **ToString** 方法有两个问题。首先，调用者无法控制字符串的格式。例如，应用程序可能需要将数字格式化成货币、十进制、百分比或者十六进制字符串。其次，调用者不能方便地选择一种特定语言文化来格式化字符串。相较于客户端代码，服务器端应用程序在第二个问题上尤其麻烦。极少数时候，应用程序需要使用与调用线程不同的语言文化来格式化字符串。为了对字符串格式进行更多的控制，你重写的 **ToString** 方法应该允许指定具体的格式和语言文化信息。

为了使调用者能选择格式和语言文化，类型应该实现 **System.IFormattable** 接口：

```
public interface IFormattable {
  String ToString(String format, IFormatProvider formatProvider);
}
```

FCL 的所有基类型(**Byte**，**SByte**，**Int16/UInt16**，**Int32/UInt32**，**Int64/UInt64**，**Single**，**Double**，**Decimal** 和 **DateTime**)都实现了这个接口。此外，还有另一些类型(比如 **Guid**)也实现了它。最后，每个枚举类型定义都自动实现 **IFormattable** 接口，以便从枚举类型的实例获取一个有意义的字符串符号。

IFormattable 的 **ToString** 方法获取两个参数。第一个是 **format**，这个特殊字符串告诉方法

应该如何格式化对象。第二个是 **formatProvider**，是实现了 **System.IFormatProvider** 接口的一个类型的实例。该类型为 **ToString** 方法提供具体的语言文化信息。稍后还要详细讨论。

实现 **IFormattable** 接口的 **ToString** 方法的类型决定哪些格式字符串能被识别。如果传递的格式字符串无法识别，类型应抛出 **System.FormatException** 异常。

Microsoft 在 FCL 中定义的许多类型都能同时识别几种格式。例如，**DateTime** 类型支持用“d”表示短日期，用“D”表示长日期，用“g”表示常规(general)，用“M”表示月/日，用“s”表示可排序(sortable)，用“T”表示长时间，用“u”表示 ISO 8601 格式的协调世界时，用“U”表示长日期格式的协调世界时，用“Y”表示年/月。所有枚举类型都支持用“G”表示常规，用“F”表示标志(flag)，用“D”表示十进制，用“X”表示十六进制。第 15 章“枚举类型和位标志”将详细讨论枚举类型的格式化。

此外，所有内建数值类型都支持用“C”表示货币格式，用“D”表示十进制格式，用“E”表示科学记数法(指数)格式，用“F”表示定点(fix-point)格式，用“G”表示常规格式，用“N”表示数字格式，用“P”表示百分比格式，用“R”表示往返行程(round-trip)格式[①]，用“X”表示十六进制格式。事实上，数值类型还支持 picture 格式字符串[②]，它是考虑到在某些时候，简单的格式字符串可能无法完全满足需求。picture 格式字符串包含一些特殊字符，它们告诉类型的 **ToString** 方法具体要显示多少个数位、具体在什么放置一个小数分隔符以及具体有多少位小数等。欲知详情，请查阅文档中的“自定义数字格式字符串”主题。

对于大多数类型，调用 **ToString** 并为格式字符串传递 **null** 值完全等价于调用 **ToString** 并为格式字符串传递“G”。换言之，对象默认使用“常规格式”对自身进行格式化。实现类型时，要选择自己认为最常用的一种格式；这个格式就是“常规格式”。顺便说一句，无参的 **ToString** 方法假定调用者希望的是“常规格式”。

理解了格式字符串之后，接着研究一下语言文化的问题。字符串默认使用与调用线程关联的语言文化信息进行格式化。无参 **ToString** 方法就是这么做的；另外，为 **formatProvider** 参数传递 **null** 值，**IFormattable** 的 **ToString** 方法也这么做。

格式化数字(货币、整数、浮点数、百分比、日期和时间)适合应用对语言文化敏感的信息。**Guid** 类型的 **ToString** 方法只返回代表 **GUID** 值的字符串。生成 **Guid** 的字符串时不必考虑

---

① 文档中的“标准数字格式字符串”一节对 R 符号(文档中称为说明符)的解释是：往返行程说明符保证转换为字符串的数值再次被分析为相同的数值。使用此说明符格式化数值时，首先使用常规格式对其进行测试：**Double** 使用 15 位精度，**Single** 使用 7 位精度。如果此值被成功地分析回相同的数值，则使用常规格式说明符对其进行格式化。但是，如果此值未被成功地分析为相同数值，则它这样格式化：**Double** 使用 17 位精度，**Single** 使用 9 位精度。虽然精度说明符可以附加到往返行程格式说明符，但它将被忽略。使用此说明符时，往返行程优先于精度。此格式仅有浮点型(**Single** 和 **Double**)支持。——译注

② picture 确实可以理解成“图片”。例如，使用 picture 数值格式字符串**###,###**可以显示千分位分隔符。换言之，像画图那样指定确切的显示格式。——译注

语言文化，因为 **GUID** 只用于编程。

格式化数字时，**ToString** 方法检查为 **formatProvider** 参数传递的值。如果传递 **null**，**ToString** 读取 **System.Globalization.CultureInfo.CurrentCulture** 属性来判断与调用线程关联的语言文化。该属性返回 **System.Globalization.CultureInfo** 类型的一个实例。

利用这个对象，**ToString** 会读取它的 **NumberFormat** 或 **DateTimeFormat** 属性(具体取决于要格式化数字还是日期/时间)。这两个属性分别返回 **System.Globalization.NumberFormatInfo** 或 **System.Globalization.DateTimeFormatInfo** 类型的实例。**NumberFormatInfo** 类型定义了 **CurrencyDecimalSeparator**，**CurrencySymbol**，**NegativeSign**，**NumberGroupSeparator** 和 **PercentSymbol** 等属性。而 **DateTimeFormatInfo** 类型定义了 **Calendar**，**DateSeparator**，**DayNames**，**LongDatePattern**，**ShortTimePattern** 和 **TimeSeparator** 等属性。ToString 会在构造并格式化字符串时读取这些属性。

调用 **IFormattable** 的 **ToString** 方法可以不传递 **null**，而是传递一个对象引用，该对象的类型实现了 **IFormatProvider** 接口：

```
public interface IFormatProvider {
  Object GetFormat(Type formatType);
}
```

IFormatProvider 接口的基本思路是：当一个类型实现了该接口，就认为该类型的实例能提供对语言文化敏感的格式信息，与调用线程关联的语言文化应被忽略。

FCL 的类型只有少数实现了 **IFormatProvider** 接口。**System.Globalization.CultureInfo** 类型就是其中之一。例如，要为越南地区格式化字符串，就需要构造一个 **CultureInfo** 对象，并将那个对象作为 **ToString** 的 **formatProvider** 参数来传递。以下代码将以越南地区适用的货币格式来获取一个 **Decimal** 数值的字符串表示：

```
Decimal price = 123.54M;
String s = price.ToString("C", new CultureInfo("vi-VN"));
MessageBox.Show(s);
```

生成并运行上述代码，会显示如图 14-5 所示的消息框。



图 14-5　数值正确格式化以表示越南货币

在内部，**Decimal** 的 **ToString** 方法发现 **formatProvider** 实参不为 **null**，所以会像下面这样调用对象的 **GetFormat** 方法：

```
NumberFormatInfo nfi = (NumberFormatInfo)
   formatProvider.GetFormat(typeof(NumberFormatInfo));
```

ToString 正是采取这种方式从 **CultureInfo** 对象获取恰当的数字格式信息。数值类型(比如

Decimal)只请求数字格式信息。但其他类型(如 **DateTime**)可能像下面这样调用 **GetFormat**:

```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
    formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

实际上，由于 **GetFormat** 的参数能标识任何类型，所以该方法非常灵活，能请求任意类型的格式信息。.NET Framework 中的类型在调用 **GetFormat** 时，暂时只会请求数字或日期/时间信息，但未来可能会请求其他格式信息。

顺便说一句，如果对象不针对任何具体的语言文化而格式化，那么为了获取它的字符串表示，应调用 **System.Globalization.CultureInfo** 的静态 **InvariantCulture** 属性，并将返回的对象作为 **ToString** 的 **formatProvider** 参数来传递:

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
MessageBox.Show(s);
```

生成并运行上述代码，会出现如图 14-6 所示的消息框。注意，在生成的字符串中，第一个字符是¤，即国际通用货币符号(U+00A4)。



图 14-6　格式化数值来表示语言文化中性的货币值

用 **InvariantCulture** 格式化的字符串一般都不是向用户显示的。相反，一般将这种字符串保存到数据文件中供将来解析。

FCL 只有 3 个类型实现了 **IFormatProvider** 接口。第一个是前面解释过的 **CultureInfo**。另外两个是 **NumberFormatInfo** 和 **DateTimeFormatInfo**。在 **NumberFormatInfo** 对象上调用 **GetFormat**，方法会检查被请求的类型是不是一个 **NumberFormatInfo**。如果是就返回 this，否则返回 **null**。类似地，在 DateTimeFormatInfo 对象上调用 **GetFormat**，如果请求的是一个 **DateTimeFormatInfo** 就返回 **this**，否则返回 **null**。这两个类型实现 **IFormatProvider** 接口是为了简化编程。试图获取对象的字符串表示时，调用者通常要指定格式，并使用与调用线程关联的语言文化。因此，经常都要调用 **ToString**，为 **format** 参数传递一个字符串，并为 **formatProvider** 参数传递 **null**。为简化 ToString 调用，许多类型都提供了 **ToString** 方法的多个重载版本。例如，**Decimal** 类型提供了 4 个不同的 **ToString** 方法:

```
// 这个版本调用 ToString(null, null)
// 含义：采用常规数值格式，采用线程的语言文化信息
public override String ToString();

// 这个版本是 ToString 的真正实现
// 这个版本实现了 IFormattable 的 ToString 方法
// 含义：采用由调用者指定的格式和语言文化信息
public String ToString(String format, IFormatProvider formatProvider);

// 这个版本简单地调用 ToString(format, null)
```

```
// 含义：采用由调用者指定的格式，采用线程的语言文化信息
public String ToString(String format);

// 这个版本简单地调用 ToString(null, formatProvider)
// 这个版本实现了 IConvertible 的 ToString 方法
// 含义：采用常规格式，采用由调用者指定的语言文化信息
public String ToString(IFormatProvider formatProvider);
```

14.4.2　将多个对象格式化成一个字符串

到目前为止讲的都是一个单独的类型如何格式化它自己的对象。但有时需要构造由多个已格式化对象构成的字符串。例如，以下字符串由一个日期、一个人名和一个年龄构成：

```
String s = String.Format("On {0}, {1} is {2} years old.",
    new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);
Console.WriteLine(s);
```

生成并运行上述代码，而且“en-US”是线程当前的语言文化，就会看到以下输出：

```
On 4/22/2012 2:35:05 PM, Aidan is 9 years old.
```

String 的静态 **Format** 方法获取一个格式字符串。在格式字符串中，大括号中的数字指定了可供替换的参数。本例的格式字符串告诉 **Format** 方法将**{0}**替换成格式字符串之后的第一个参数(日期/时间)，将**{1}**替换成格式字符串之后的第二个参数(**"Aidan"**)，将**{2}**替换成格式字符串之后的第三个参数(**9**)。

在内部，**Format** 方法会调用每个对象的 **ToString** 方法来获取对象的字符串表示。返回的字符串依次连接到一起，并返回最终的完整字符串。看起来不错，但它意味着所有对象都要使用它们的常规格式和调用线程的语言文化信息来格式化。

在大括号内指定格式信息，可以更全面地控制对象格式化。例如，以下代码和上例几乎完全一致，只是为可替换参数 0 和 2 添加了格式化信息：

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
   new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);
Console.WriteLine(s);
```

生成并运行上述代码，同时“en-US”是线程当前的语言文化，会看到以下输出：

```
On Sunday, April 22, 2012, Aidan is 9.000000E+000 years old.
```

Format 方法解析格式字符串时，发现可替换参数 0 应该调用它的 **IFormattable** 接口的 **ToString** 方法，并为该方法的两个参数分别传递**"D"**和 **null**。类似地，**Format** 会调用可替换的参数 2 的 **IFormattable** 接口的 **ToString** 方法，并传递**"E"**和 **null**。假如可替换参数 0 和 2 的类型没有实现 **IFormattable** 接口，**Format** 会调用从 **Object** 继承(而且有可能重写)的无参 **ToString** 方法，并将默认格式附加到最终生成的字符串中。

String 类提供了静态 **Format** 方法的几个重载版本。一个版本获取实现了 **IFormatProvider** 接口的对象，允许使用由调用者指定的语言文化信息来格式化所有可替换参数。显然，这个版本的 **Format** 会调用每个对象的 **IFormattable.ToString** 方法，并将传给 **Format** 的任何 **IFormatProvider** 对象传给它。

如果使用 **StringBuilder** 而不是 String 来构造字符串，可以调用 **StringBuilder** 的 **AppendFormat** 方法。它的原理与 **String** 的 **Format** 方法相似，只是会格式化字符串并将其附加到 StringBuilder 的字符数组中。和 **String** 的 **Format** 方法一样，**AppendFormat** 也要获取格式字符串，而且也有获取一个 **IFormatProvider** 的版本。

System.Console 的 **Write** 和 **WriteLine** 方法也能获取格式字符串和可替换参数。但 **Console** 的 **Write** 和 **WriteLine** 方法没有重载版本能获取一个 **IFormatProvider**。要格式化符合特定语言文化的字符串，必须调用 **String** 的 **Format** 方法，首先传递所需的 **IFormatProvider** 对象，再将结果字符串传给 **Console** 的 **Write** 或 **WriteLine** 方法。但这应该不是一个大问题。正如前文所述，客户端代码极少需要使用有别于调用线程的其他语言文化来格式化字符串。

14.4.3 提供定制格式化器

现在应该很清楚了，.NET Framework 的格式化功能旨在提供更大的灵活性和更多的控制。但讨论还没有完。可以定义一个方法，在任何对象需要格式化成字符串的时候由 **StringBuilder** 的 **AppendFormat** 方法调用该方法。也就是说，**AppendFormat** 不是为每个对象调用 **ToString**，而是调用定制的方法，按照我们希望的任何方式格式化部分或全部对象。下面的讨论也适用于 **String** 的 **Format** 方法。

下面通过一个例子来解释这个机制。假定需要格式化用户在 Internet 浏览器中查看的 HTML 文本，希望所有 **Int32** 值都加粗显示。所以，每次将 **Int32** 值格式化成 **String** 时，都要用 **<B>**和**</B>**标记将字符串包围起来。以下代码演示了要做到这一点是多么容易：

```
using System;
using System.Text;
using System.Threading;

public static class Program {
  public static void Main() {
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}", "Jeff", 123, DateTime.Now);
    Console.WriteLine(sb);
  }
}

internal sealed class BoldInt32s : IFormatProvider, ICustomFormatter {
  public Object GetFormat(Type formatType) {
    if (formatType == typeof(ICustomFormatter)) return this;
    return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
  }

  public String Format(String format, Object arg, IFormatProvider formatProvider) {
    String s;

    IFormattable formattable = arg as IFormattable;

    if (formattable == null) s = arg.ToString();
    else s = formattable.ToString(format, formatProvider);

    if (arg.GetType() == typeof(Int32))
      return "<B>" + s + "</B>";
    return s;
```

```
  }
}
```

编译并运行上述代码，而且“en-US”是线程当前的语言文化，控制台上将显示以下输出(你的日期当然不同)：

```
Jeff <B>123</B> September 1
```

Main 构造了一个空白 **StringBuilder**，在其中附加了格式化好的字符串。调用 **AppendFormat** 时，第一个参数是 **BoldInt32s** 类的实例。该类实现了前面描述的 IFormatProvider 接口，另外还实现了 **ICustomFormatter** 接口：

```
public interface ICustomFormatter {
  String Format(String format, Object arg,
    IFormatProvider formatProvider);
}
```

任何时候只要 **StringBuilder** 的 **AppendFormat** 方法需要获取对象的字符串表示，就会调用这个接口的 **Format** 方法。可在方法内部通过一些巧妙的操作对字符串格式化进行全面控制。现在深入 **AppendFormat** 方法内部，看看它具体如何工作。以下伪代码展示了 **AppendFormat** 的工作方式：

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
  String format, params Object[] args) {

  // 如果传递了一个 IFormatProvider，
  // 就调查它是否提供了一个 ICustomFormatter 对象
  ICustomFormatter cf = null;

  if (formatProvider != null)
    cf = (ICustomFormatter)
      formatProvider.GetFormat(typeof(ICustomFormatter));

  // 在 StringBuilder 的字符数组中连续附加
  // 字面值(literal)字符(伪代码未显示)和可替换参数
  Boolean MoreReplaceableArgumentsToAppend = true;
  while (MoreReplaceableArgumentsToAppend) {
    // argFormat 引用从 format 参数获取的可替换格式字符串
    String argFormat = /* ... */;

    // argObj 引用来自 args 数组参数的对应元素
    Object argObj = /* ... */;

    // argStr 引用要在最终生成的字符串后面附加的格式化好的字符串
    String argStr = null;

    // 如果存在定制格式化器，就让它对传递的实参进行格式化
    if (cf != null)
      argStr = cf.Format(argFormat, argObj, formatProvider);

    // 如果不存在定制格式化器，或者它不对实参进行格式化，
    // 就尝试别的操作
    if (argStr == null) {
      // 实参的类型支持通过 IFormattable 接口来实现的
      // 富格式化操作(rich formatting)吗
      IFormattable formattable = argObj as IFormattable;
      if (formattable != null) {
        // 是：向类型的 IFormattable ToString 方法传递
        // 格式字符串和 formatProvider 参数
        argStr = formattable.ToString(argFormat, formatProvider);
```

```
      } else {
        // 否：使用线程的语言文化信息来获取默认格式
        if (argObj != null) argStr = argObj.ToString();
        else argStr = String.Empty;
      }
    }

    // 将 argStr 的字符附加到“字符数组”字段成员
    /* ... */

    // 检查剩余的参数是否需要格式化
    MoreReplaceableArgumentsToAppend = /* ... */;
  }
  return this;
}
```

当 **Main** 调用 **AppendFormat** 时，**AppendFormat** 会调用我的格式提供器(format provider)的 **GetFormat** 方法，向它传递 **ICustomFormatter** 类型。在 **BoldInt32s** 类型中定义的 **GetFormat** 方法发现当前请求了 **ICustomFormatter**，所以会返回对它自身的引用，因为它已实现了该接口。调用我的 **GetFormat** 方法时，如果传递的是其他任何类型，就调用与线程关联的 **CultureInfo** 对象的 **GetFormat** 方法。

AppendFormat 需要格式化一个可替换参数时，会调用 **ICustomFormatter** 的 **Format** 方法。在本例中，**AppendFormat** 调用由 **BoldInt32s** 类型定义的 **Format** 方法。我的这个 **Format** 方法核实要格式化的对象是否支持通过 **IFormattable** 接口来实现的富格式化操作。如果对象不支持，就调用简单的、无参的 **ToString** 方法(从 **Object** 继承)来格式化对象。如果对象支持 **IFormattable**，就调用支持富格式化的 **ToString**，向它传递格式字符串和格式提供器。

现在已经有了格式化好的字符串。接着要核实对应的对象是不是一个 **Int32** 类型；如果是，就将格式化好的字符串放到**<B>**和**</B>**这两个 HTML 标记之间，然后返回新字符串。如果不是，就简单地返回格式化好的字符串，不做进一步处理。

14.5 解析字符串来获取对象：Parse

上一节解释了如何获取对象并得到它的字符串表示。本节讨论与之相反的操作，即如何获取字符串并得到它的对象表示。从字符串获得对象，这并不是一个常见的操作，但偶尔也会用到。Microsoft 觉得有必要规划一个正式的机制将字符串解析成对象。

能解析字符串的任何类型都提供了公共静态方法 **Parse**。方法获取一个 **String** 并返回类型的实例。从某种意义上说，**Parse** 扮演了一个工厂(factory)的角色。在 FCL 中，所有数值类型、**DateTime**、**TimeSpan** 以及其他一些类型(比如各种 SQL 数据类型)均提供了 **Parse** 方法。

来看看如何将字符串解析成数值类型。所有数值类型(**Byte**，**SByte**，**Int16/UInt16**，**Int32/UInt32**，**Int64/UInt64**，**Single**，**Double** 和 **Decimal**)都提供了至少一个 **Parse** 方法。以 **Int32** 类型的 **Parse** 方法为例(其他数值类型的 **Parse** 方法与此相似)：

```
public static Int32 Parse(String s, NumberStyles style,
  IFormatProvider provider);
```

从原型就能猜出方法具体如何工作。**String** 参数 **s** 是希望解析成 **Int32** 对象的一个数字的字符串表示。**System.Globalization.NumberStyles** 参数 **style** 是位标志(bit flag)集合，标识了 **Parse** 应在字符串查找的字符(也就是字符串 **s** 中允许的样式，有不允许的样式会抛出异常)。如本章前面所述，**IFormatProvider** 参数 **provider** 标识了一个对象，**Parse** 方法通过该对象获取语言文化特有的信息。

例如，以下代码造成 **Parse** 抛出 **System.FormatException** 异常，因为要解析的字符串包含前导空白字符：

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

要允许 **Parse** 跳过前导的空白字符，要像下面这样修改 **style** 参数：

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

请参考文档来完整地了解 **NumberStyles** 枚举类型定义的位符号和常见组合。

以下代码段展示了如何解析十六进制数：

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x); // 显示: "26"
```

这个 **Parse** 方法接受 3 个参数。为简化编程，许多类型都提供了额外的 **Parse** 重载版本，所以不需要传递如此多的实参。例如，**Int32** 提供了 **Parse** 方法的 4 个重载版本：

```
// 为 style 参数传递 NumberStyles.Integer,
// 并传递线程的语言文化提供者信息
public static Int32 Parse(String s);

// 传递线程的语言文化提供者信息
public static Int32 Parse(String s, NumberStyles style);

// 为 style 参数传递 NumberStyles.Integer
public static Int32 Parse(String s, IFormatProvider provider);

// 以下是刚才展示的版本
public static Int32 Parse(String s, NumberStyles style,
  IFormatProvider provider);
```

DateTime 类型也提供了一个 **Parse** 方法：

```
public static DateTime Parse(String s,
  IFormatProvider provider, DateTimeStyles styles);
```

这个方法与数值类型定义的 **Parse** 方法相似，只是 **DateTime** 的 **Parse** 方法获取的是由 **System.Globalization.DateTimeStyles** 枚举类型(而不是 **NumberStyles** 枚举类型)定义的位标志集合。参考文档来全面了解 **DateTimeStyles** 类型定义的位符号和常见组合。

为简化编程，**DateTime** 类型提供了 **Parse** 方法的 3 个重载版本：

```
// 传递线程的语言文化的提供者信息
// 并为 style 参数传递 DateTimeStyles.None
public static DateTime Parse(String s);

// 为 style 参数传递 DateTimeStyles.None
public static DateTime Parse(String s, IFormatProvider provider);
```

```
// 以下是刚才展示的版本
public static DateTime Parse(String s,
  IFormatProvider provider, DateTimeStyles styles);
```

对日期和时间的解析比较复杂。许多开发人员都感觉 **DateTime** 类型的 **Parse** 方法过于宽松，有时会解析不含日期或时间的字符串。有鉴于此，**DateTime** 类型还提供了 **ParseExact** 方法，它接受一个 picture 格式字符串，能准确描述应该如何格式化日期/时间字符串，以及如何对它进行解析。欲知详情，请在文档中查阅 **DateTimeFormatInfo** 类。



注意 一些开发人员向 Microsoft 报告了这样一个问题：如果应用程序频繁调用 **Parse**，而且 **Parse** 频繁抛出异常时(由于无效的用户输入)，应用程序的性能会显著下降。为此，Microsoft 在所有数值数据类型、**DateTime** 类型、**TimeSpan** 类型、甚至 **IPAddress** 类型中加入了 **TryParse** 方法。下面是 **Int32** 的 **TryParse** 方法的两个重载版本之一：

```
public static Boolean TryParse(String s, NumberStyles style,
   IFormatProvider provider, out Int32 result);
```

可以看出，方法会返回 **true** 或 **false**，指出传递的字符串是否能解析成 **Int32**。如果返回 **true**，以“传引用”的方式传给 **result** 参数的变量将包含解析好的数值。**TryXxx** 模式的详情将在第 20 章“异常和状态管理”中讨论。

14.6 编码：字符和字节的相互转换

Win32 开发人员经常要写代码将 Unicode 字符和字符串转换成“多字节字符集”(Multi-Byte Character Set，MBCS)格式。我个人就经常写这样的代码，这个过程很繁琐，还容易出错。在 CLR 中，所有字符都表示成 16 位 Uncode 码值，而所有字符串都由 16 位 Unicode 码值构成，这简化了运行时的字符和字符串处理。

但偶尔也想要将字符串保存到文件中，或者通过网络传输。如果字符串中的大多数字符都是英语用户用的，那么保存或传输一系列 16 位值，效率就显得不那么理想，因为写入的半数字节都只由零构成。相反，更有效的做法是将 16 位值编码成压缩的字节数组，以后再将字节数组解码回 16 位值的数组。

这种编码技术还使托管应用程序能和非 Unicode 系统创建的字符串进行交互。例如，要生成能由 Windows 95 日文版上运行的应用程序读取的文件，必须使用 Shift-JIS(代码页 932)保存 Unicode 文本。类似地，要用 Shift-JIS 编码将 Windows 95 日文版生成的文本文件读入 CLR。

用 **System.IO.BinaryWriter** 或者 **System.IO.StreamWriter** 类型将字符串发送给文件或网络流时，通常要进行编码。对应地，用 **System.IO.BinaryReader** 或者 **System.IO.StreamReader** 类型从文件或网络流中读取字符串时，通常要进行解码。不显式指定一种编码方案，所有

这些类型都默认使用 UTF-8[①]。但有时还是需要显式编码或解码字符串。即使不需要显式编码或解码，也能通过本节的学习，对流中的字符串读写有一个更清醒的认识。

幸好，FCL 提供了一些类型来简化字符编码和解码。两种最常用的编码方案是 UTF-16 和 UTF-8，如下所述。

- UTF-16 将每个 16 位字符编码成 2 个字节。不对字符产生任何影响，也不发生压缩——性能非常出色。UTF-16 编码也称为“Unicode 编码”。还要注意，UTF-16 可用于从“低位优先”(little-endian)转换成“高位优先”(big-endian)，或者从“高位优先”转换成“低位优先”。

- UTF-8 将部分字符编码成 1 个字节，部分编码成 2 个字节，部分编码成 3 个字节，再有部分编码成 4 个字节。值在 0x0080 之下的字符压缩成 1 个字节，适合表示美国使用的字符。0x0080～0x07FF 的字符转换成 2 个字节，适合欧洲和中东语言。0x0800 以及之上的字符转换成 3 个字节，适合东亚语言。最后，代理项对(surrogate pair)表示成 4 个字节。UTF-8 编码方案非常流行，但如果要编码的许多字符都具有 0x0800 或者之上的值，效率反而不如 UTF-16。

UTF-16 和 UTF-8 编码是目前最常用的编码方案。FCL 还支持下面这些不常用的。

- UTF-32 使用 4 个字节来编码所有字符。要写简单算法来遍历所有字符，同时不愿意花额外精力应付字节数可变的字符，就适合采用这种编码。例如，使用 UTF-32 根本不需要考虑代理项的问题，因为每个字符都是 4 字节。当然，UTF-32 的内存使用并不高效，所以很少用它将字符串保存到文件或者通过网络来传输字符串。这种编码方案通常在程序内部使用。还要注意，UTF-32 可用于“低位优先”和“高位优先”之间的相互转换。

- UTF-7 编码用于旧式系统。在那些系统上，字符可以使用 7 位值来表示。应该避免使用这种编码，因为它最终通常会使数据膨胀，而不是压缩。这种编码方案已被 Unicode 协会淘汰。

- ASCII 编码方案将 16 位字符编码成 ASCII 字符；也就是说，值小于 0x0080 的 16 位字符被转换成单字节。值超过 0x007F 的任何字符都不能被转换，否则字符的值会丢失。假如字符串完全由 ASCII 范围(0x00～0x7F)内的字符构成，ASCII 编码方案就能将数据压缩到原来的一半，而且速度非常快(高位字节被直接截掉)。但如果一些字符在 ASCII 范围之外，这种编码方案就不适合了，因为字符的值会丢失。

最后，FCL 还允许将 16 位字符编码到任意代码页。和 ASCII 一样，编码到代码页也是危险的，因为代码页表示不了的任何字符都会丢失。除非必须和使用其他编码方案的遗留文件或应用程序兼容，否则应该总是选择 UTF-16 或 UTF-8 编码。

---

① UTF 全称是 Unicode Transformation Format，即“Unicode 转换格式”。——译注

要编码或解码一组字符时，应获取从 **System.Text.Encoding** 派生的一个类的实例。抽象基类 **Encoding** 提供了几个静态只读属性，每个属性都返回从 **Encoding** 派生的一个类的实例。

下例使用 UTF-8 进行字符编码/解码。

```
using System;
using System.Text;

public static class Program {
  public static void Main() {
    // 准备编码的字符串
    String s = "Hi there.";

    // 获取从 Encoding 派生的一个对象，
    // 它知道怎样使用 UTF8 来进行编码/解码
    Encoding encodingUTF8 = Encoding.UTF8;

    // 将字符串编码成字节数组
    Byte[] encodedBytes = encodingUTF8.GetBytes(s);

    // 显示编好码的字节值
    Console.WriteLine("Encoded bytes: " +
      BitConverter.ToString(encodedBytes));

    // 将字节数组解码回字符串
    String decodedString = encodingUTF8.GetString(encodedBytes);

    // 显示解码的字符串
    Console.WriteLine("Decoded string: " + decodedString);
  }
}
```

上述代码的输出如下：

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
Decoded string: Hi there.
```

除了 **UTF8** 静态属性，**Encoding** 类还提供了以下静态属性：**Unicode**，**BigEndianUnicode**，**UTF32**，**UTF7**，**ASCII** 和 **Default**。**Default** 属性返回的对象使用用户当前的代码页来进行编码/解码。当前用户的代码页是在控制面板的“区域和语言选项”对话框中，通过“非 Unicode 程序中所使用的当前语言”区域的选项来指定的(查阅 Win32 函数 **GetACP** 了解详情)。但不鼓励使用 **Default** 属性，因为这样一来，应用程序的行为就会随着机器的设置而变。也就是说，一旦更改系统默认代码页，或者应用程序在另一台机器上运行，应用程序的行为就会改变。

除了这些属性，**Encoding** 还提供了静态 **GetEncoding** 方法，允许指定代码页(整数或字符串形式)，并返回可以使用指定代码页来编码/解码的对象。例如，可调用 **GetEncoding** 并传递**"Shift-JIS"**或者 **932**。

首次请求一个编码对象时，**Encoding** 类的属性或者 **GetEncoding** 方法会为请求的编码方案构造对象，并返回该对象。假如请求的编码对象以前请求过，**Encoding** 类会直接返回之前构造好的对象；不会为每个请求都构造新对象。这一举措减少了系统中的对象数量，也缓解了堆的垃圾回收压力。

除了调用 **Encoding** 的某个静态属性或者它的 **GetEncoding** 方法，还可构造以下某个类的实例：**System.Text.UnicodeEncoding**，**System.Text.UTF8Encoding**，**System.Text.UTF32Encoding**，**System.Text.UTF7Encoding** 或者 **System.Text.ASCIIEncoding**。但要注意，构造任何这些类的实例都会在托管堆中创建新对象，对性能有损害。

其中 4 个类(**UnicodeEncoding**，**UTF8Encoding**，**UTF32Encoding** 和 **UTF7Encoding**)提供了多个构造器，允许对编码和前导码[①]进行更多的控制(前导码有时也称为“字节顺序标记”，即 Byte Order Mark 或者 BOM)。在这 4 个类中，前 3 个类还提供了特殊的构造器，允许在对一个无效的字节序列进行解码的时候抛出异常。如果需要保证应用程序的安全性，防范无效的输入数据，就应当使用这些能抛出异常的类。

处理 **BinaryWriter** 或 **StreamWriter** 时，显式构造这些 **Encoding** 类型的实例是可以的。但 **ASCIIEncoding** 类仅一个构造器，没有提供更多的编码控制。所以如果需要 **ASCIIEncoding** 对象，请务必查询 **Encoding** 的 **ASCII** 属性来获得。该属性返回的是一个 **ASCIIEncoding** 对象引用。自己构造 **ASCIIEncoding** 对象会在堆上创建更多的对象，无谓地损害应用程序的性能。

一旦获得从 **Encoding** 派生的对象，就可调用 **GetBytes** 方法将字符串或字符数组转换成字节数组(**GetBytes** 有几个重载版本)。要将字节数组转换成字符数组或字符串，需要调用 **GetChars** 方法或者更有用的 **GetString** 方法(这两个方法都有几个重载版本)。前面的示例代码演示了如何调用 **GetBytes** 和 **GetString** 方法。

从 **Encoding** 派生的所有类型都提供了 **GetByteCount** 方法，它能统计对一组字符进行编码所产生的字节数，同时不实际进行编码。虽然 **GetByteCount** 的用处不是很大，但在分配字节数组时还是可以用一下的。另有一个 **GetCharCount** 方法，它返回解码得到的字符数，同时不实际进行解码。要想节省内存和重用数组，可考虑使用这些方法。

GetByteCount 和 **GetCharCount** 方法的速度一般，因为必须分析字符或字节数组才能返回准确的结果。如果更加追求速度而不是结果的准确性，可改为调用 **GetMaxByteCount** 或 **GetMaxCharCount** 方法。这两个方法获取代表字符数或字节数的一个整数，返回最坏情况下的值。[②]

从 **Encoding** 派生的每个对象都提供了一组公共只读属性，可查询这些属性来获取有关编码的详细信息。详情请参考文档。

以下程序演示了大多数属性及其含义，它将显示几个不同的编码的属性值：

```
using System;
using System.Text;

public static class Program {
  public static void Main() {
```

① reamble 在文档中翻译成“前导码”，可通过 **Encoding.GetPreamble** 方法获取。——译注

② 算如果对指定数量的字符/字节进行编码/解码，那么所产生的最大字节数/字符数。——译注

```
    foreach (EncodingInfo ei in Encoding.GetEncodings()) {
      Encoding e = ei.GetEncoding();
      Console.WriteLine("{1}{0}" +
        "\tCodePage={2}, WindowsCodePage={3}{0}" +
        "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +
        "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +
        "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",

        Environment.NewLine,
        e.EncodingName, e.CodePage, e.WindowsCodePage,
        e.WebName, e.HeaderName, e.BodyName,
        e.IsBrowserDisplay, e.IsBrowserSave,
        e.IsMailNewsDisplay, e.IsMailNewsSave);
    }
  }
}
```

运行上述程序将得到以下输出(为节省篇幅删除了部分内容):

```
IBM EBCDIC (美国-加拿大)
     CodePage=37, WindowsCodePage=1252
     WebName=IBM037, HeaderName=IBM037, BodyName=IBM037
     IsBrowserDisplay=False, IsBrowserSave=False
     IsMailNewsDisplay=False, IsMailNewsSave=False

OEM 美国
     CodePage=437, WindowsCodePage=1252
     WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
     IsBrowserDisplay=False, IsBrowserSave=False
     IsMailNewsDisplay=False, IsMailNewsSave=False

IBM EBCDIC (国际)
     CodePage=500, WindowsCodePage=1252
     WebName=IBM500, HeaderName=IBM500, BodyName=IBM500
     IsBrowserDisplay=False, IsBrowserSave=False
     IsMailNewsDisplay=False, IsMailNewsSave=False

阿拉伯字符(ASMO-708)
     CodePage=708, WindowsCodePage=1256
     WebName=ASMO-708, HeaderName=ASMO-708, BodyName=ASMO-708
     IsBrowserDisplay=True, IsBrowserSave=True
     IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode
     CodePage=1200, WindowsCodePage=1200
     WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
     IsBrowserDisplay=False, IsBrowserSave=True
     IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (Big-Endian)
     CodePage=1201, WindowsCodePage=1200
     WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
     IsBrowserDisplay=False, IsBrowserSave=False
     IsMailNewsDisplay=False, IsMailNewsSave=False

西欧字符(DOS)
     CodePage=850, WindowsCodePage=1252
     WebName=ibm850, HeaderName=ibm850, BodyName=ibm850
     IsBrowserDisplay=False, IsBrowserSave=False
     IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (UTF-8)
```

```
CodePage=65001, WindowsCodePage=1200
WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
IsBrowserDisplay=True, IsBrowserSave=True
IsMailNewsDisplay=True, IsMailNewsSave=True
```

表 14-3 总结了 **Encoding** 的所有派生类都提供的常用方法。

表 14-3　Encoding 的派生类提供的方法

方法名称	说明
GetPreamble	返回一个字节数组，指出在写入任何已编码字节之前，首先应该在一个流中写入什么字节。这些字节经常称为“前导码”(preamble)或“字节顺序标记”(Byte Order Mark，BOM)字节。开始从一个流中读取时，BOM 字节自动帮助检测当初写入流时采用的编码，以确保使用正确的解码器。对于从 **Encoding** 派生的一些类，这个方法返回 0 字节的数组——即没有前导码字节。显式构造 **UTF8Encoding** 对象，这个方法将返回一个 3 字节数组(包含 0xEF，0xBB 和 0xBF)。显式构造 **UnicodeEncoding** 对象，这个方法将返回一个 2 字节数组(包含 0xFE 和 0xFF)来表示“高位优先”(big-endian)编码，或者返回一个 2 字节数组(包含 0xFF 和 0xFE)来表示“低位优先”(little-endian)编码。默认为低位优先
Convert	将字节数组从一种编码(来源编码)转换为另一种(目标编码)。在内部，这个静态方法调用来源编码对象的 **GetChars** 方法，并将结果传给目标编码对象的 **GetBytes** 方法。结果字节数组返回给调用者
Equals	如果从 **Encoding** 派生的两个对象代表相同的代码页和前导码设置，就返回 **true**
GetHashCode	返回当前 **Encoding** 实例的哈希码

14.6.1　字符和字节流的编码和解码

假定现在要通过 **System.Net.Sockets.NetworkStream** 对象来读取一个 UTF-16 编码字符串。字节流通常以数据块(data chunk)的形式传输。换言之，可能是先从流中读取 5 个字节，再读取 7 个字节。UTF-16 的每个字符都由 2 个字节构成。所以，调用 **Encoding** 的 **GetString** 方法并传递第一个 5 字节数组，返回的字符串只包含 2 个字符。再次调用 **GetString** 并传递接着的 7 个字节，将返回只包含 3 个字符的字符串。显然，所有 code point[①]都会存储错误的值！

之所以会造成数据损坏，是由于所有 **Encoding** 派生都不维护多个方法调用之间的状态。要编码或解码以数据块形式传输的字符/字节，必须进行一些额外的工作来维护方法调用之间的状态，从而防止丢失数据。

字节块解码首先要获取一个 **Encoding** 派生对象引用(参见上一节)，再调用其 **GetDecoder** 方法。方法返回对一个新构造对象的引用，该对象的类型从 **System.Text.Decoder** 类派生。

① ode point 是一个抽象概念。可将每个字符都想像成一个抽象的 Unicode code point，可能需要使用多个字节来表示一个 code point。——译注

和 **Encoding** 类一样，**Decoder** 也是抽象基类。查阅文档，会发现找不到任何具体实现了 **Decoder** 的类。但 FCL 确实定义了一系列 **Decoder** 派生类。这些类在 FCL 内部使用。但是，**GetDecoder** 方法能构造这些类的实例，并将这些实例返回给应用程序代码。

Decoder 的所有派生类都提供了两个重要的方法：**GetChars** 和 **GetCharCount**。显然，这些方法的作用是对字节数组进行解码，工作方式类似于前面讨论过的 **Encoding** 的 **GetChars** 和 **GetCharCount** 方法。调用其中一个方法时，它会尽可能多地解码字节数组。假如字节数组包含的字节不足以完成一个字符，剩余的字节会保存到 **Decoder** 对象内部。下次调用其中一个方法时，**Decoder** 对象会利用之前剩余的字节再加上传给它的新字节数组。这样一来，就可以确保对数据块进行正确解码。从流中读取字节时 **Decoder** 对象的作用很大。

从 **Encoding** 派生的类型可用于无状态(中途不保持状态)编码和解码。而从 **Decoder** 派生的类型只能用于解码。以成块的方式编码字符串需要调用 **GetEncoder** 方法，而不是调用 **Encoding** 对象的 **GetDecoder** 方法。**GetEncoder** 返回一个新构造的对象，它的类型从抽象基类 **System.Text.Encoder** 派生。在文档中同样找不到谁具体实现了 **Encoder**。但 FCL 确实定义了一系列 **Encoder** 派生类。和从 **Decoder** 派生的类一样，这些类全都在 FCL 内部使用，只是 **GetEncoder** 方法能构造这些类的实例，并将这些实例返回给应用程序代码。

从 **Encoder** 派生的所有类都提供了两个重要方法：**GetBytes** 和 **GetByteCount**。每次调用，从 **Encoder** 派生的对象都会维护余下数据的状态信息，以便以成块的方式对数据进行编码。

14.6.2 Base-64 字符串编码和解码

写作本书时，UTF-16 和 UTF-8 编码已相当流行。另一个流行的方案是将字节序列编码成 Base-64 字符串。FCL 专门提供了进行 Base-64 编码和解码的方法。你可能以为这是通过一个从 **Encoding** 派生的类型来完成的。但考虑到某些原因，Base-64 编码和解码用 **System.Convert** 类型提供的一些静态方法来进行。

将 Base-64 字符串编码成字节数组需调用 **Convert** 的静态 **FromBase64String** 或 **FromBase64CharArray** 方法。类似地，将字节数组解码成 Base-64 字符串需调用 **Convert** 的静态 **ToBase64String** 或者 **ToBase64CharArray** 方法。以下代码演示了如何使用其中的部分方法：

```
using System;

public static class Program {
  public static void Main() {
    // 获取一组 10 个随机生成的字节
    Byte[] bytes = new Byte[10];
    new Random().NextBytes(bytes);

    // 显示字节
    Console.WriteLine(BitConverter.ToString(bytes));

    // 将字节解码成 Base-64 字节串，并显示字符串
    String s = Convert.ToBase64String(bytes);
    Console.WriteLine(s);
```

```
      // 将 Base-64 字符串编码回字节，并显示字节
      bytes = Convert.FromBase64String(s);
      Console.WriteLine(BitConverter.ToString(bytes));
   }
}
```

编译并执行上述代码得到以下输出(因为是随机生成的字节，所以你的输出可能不同)：

```
3B-B9-27-40-59-35-86-54-5F-F1
O7knQFk1hlRf8Q==
3B-B9-27-40-59-35-86-54-5F-F1
```

14.7　安全字符串

String 对象可能包含敏感数据，比如用户密码或信用卡资料。遗憾的是，**String** 对象在内存中包含一个字符数组。如果允许执行不安全或者非托管的代码，这些代码就可以扫描进程的地址空间，找到包含敏感数据的字符串，并以非授权的方式加以利用。即使 **String** 对象只用一小段时间就进行垃圾回收，CLR 也可能无法立即重用 **String** 对象的内存，致使 **String** 的字符长时间保留在进程的内存中(尤其是假如 **String** 对象是较老的一代[①])，造成机密数据泄露。此外，由于字符串不可变(immutable)，所以当你处理它们时，旧的副本会逗留在内存中，最终造成多个不同版本的字符串散布在整个内存空间中。

有的政府部门有严格的安全要求，对各种安全措施进行了非常具体的规定。为了满足这些要求，Microsoft 在 FCL 中增添了一个更安全的字符串类，即 **System.Security.SecureString**。构造 **SecureString** 对象时，会在内部分配一个非托管内存块，其中包含一个字符数组。使用非托管内存是为了避开垃圾回收器的“魔爪”。

这些字符串的字符是经过加密的，能防范任何恶意的非安全/非托管代码获取机密信息。利用以下任何一个方法，即可在安全字符串中附加、插入、删除或者设置一个字符：**AppendChar**，**InsertAt**，**RemoveAt** 和 **SetAt**。调用其中任何一个方法时，方法内部会解密字符，执行指定的操作，再重新加密字符。这意味着字符有一小段时间处于未加密状态。还意味着这些操作的性能会比较一般。所以，应该尽可能少地执行这些操作。

SecureString 类实现了 **IDisposable** 接口，允许以简单的方式确定性地摧毁字符串中的安全内容。应用程序不再需要敏感的字符串内容时，只需调用 **SecureString** 的 **Dispose** 方法。在内部，**Dispose** 会对内存缓冲区的内容进行清零，确保恶意代码无法获得敏感信息，然后释放缓冲区。**SecureString** 对象内部的一个字段引用了一个从 **SafeBuffer** 派生的对象，它负责维护实际的字符串。由于 **SafeBuffer** 类最终从 **CriticalFinalizerObject** 类派生[②]，所以字符串在垃圾回收时，它的字符内容保证会被清零，而且缓冲区会得到释放。和 **String** 对

---

① CLR 为了改善垃圾回收性能，引入了“代”(generation)的概念。简单地说，越新的对象，生命期越短。越老的对象，生命期越长。这些对象按照新老顺序，放入垃圾回收器专门分配的内存空间中(第 0 代，第 1 代和第 2 代)。——译注

② 第 21 章“自动内存管理(垃圾回收)”将讨论该抽象基类。

象不同，**SecureString** 对象在被回收之后，加密字符串的内容将不再存在于内存中。

知道了如何创建和修改 **SecureString** 对象之后，接着讨论如何使用它。遗憾的是，最新的 FCL 限制了对 **SecureString** 类的支持。也就是说，只有少数方法才能接受 **SecureString** 参数。在.NET Framework 4 中，以下情况允许将 **SecureString** 作为密码传递。

- 与加密服务提供程序(Cryptographic Service Provider，CSP)协作。参见 **System.Security.Cryptography.CspParameters** 类。
- 创建、导入或导出 X.509 证书。参见 **System.Security.Cryptography.X509Certificates.X509Certificate** 类和 **System.Security.Cryptography.X509Certificates.X509Certificate2** 类。
- 在特定用户帐户下启动新进程。参见 **System.Diagnostics.Process** 和 **System.Diagnostics.ProcessStartInfo** 类。
- 构造事件日志会话。参见 **System.Diagnostics.Eventing.Reader.EventLogSession** 类。
- 使用 **System.Windows.Controls.PasswordBox** 控件。参见该类的 **SecurePassword** 属性。

最后，可以创建自己的方法来接受 **SecureString** 对象参数。方法内部必须先让 **SecureString** 对象创建一个非托管内存缓冲区，它将用于包含解密过的字符，然后才能让该方法使用缓冲区。为了最大程度降低恶意代码获取敏感数据的风险，你的代码在访问解密过的字符串时，时间应尽可能短。结束使用字符串之后，代码应尽快清零并释放缓冲区。此外，绝对不要将 **SecureString** 的内容放到一个 **String** 中。否则，**String** 会在堆中保持未加密状态，只有经过垃圾回收，而且内存被重用的时候，它的字符内容才会被清零。**SecureString** 类特地没有重写 **ToString** 方法，目的就是避免泄露敏感数据。

下例演示了如何初始化和使用一个 **SecureString**(编译时要为 C#编译器指定**/unsafe** 开关选项)：

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

public static class Program {
  public static void Main() {
    using (SecureString ss = new SecureString()) {
      Console.Write("Please enter password: ");
      while (true) {
        ConsoleKeyInfo cki = Console.ReadKey(true);
        if (cki.Key == ConsoleKey.Enter) break;

        // 将密码字符附加到 SecureString 中
        ss.AppendChar(cki.KeyChar);
        Console.Write("*");
      }
      Console.WriteLine();

      // 密码已输入，出于演示的目的显示它
      DisplaySecureString(ss);
    }

    // using 之后，SecureString 被 dispose，内存中无敏感数据
```

```
  }

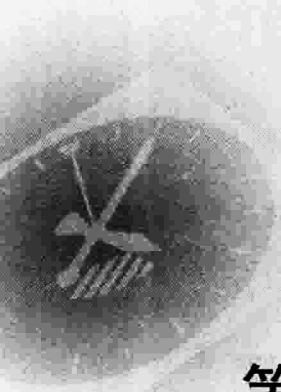
  // 这个方法是不安全的，因为它要访问非托管内存
  private unsafe static void DisplaySecureString(SecureString ss) {
    Char* pc = null;
    try {
      // 将 SecureString 解密到一个非托管内存缓冲区中
      pc = (Char*) Marshal.SecureStringToCoTaskMemUnicode(ss);

      // 访问包含已解密 SecureString 的非托管内存缓冲区
      for (Int32 index = 0; pc[index] != 0; index++)
        Console.Write(pc[index]);
    }
    finally {
      // 确定清零并释放包含已解密 SecureString 字符的非托管内存缓冲区
      if (pc != null)
        Marshal.ZeroFreeCoTaskMemUnicode((IntPtr) pc);
    }
  }
}
```

System.Runtime.InteropServices.Marshal 类提供了 5 个方法来将一个 **SecureString** 的字符解密到非托管内存缓冲区。所有方法都是静态方法，所有方法都接受一个 **SecureString** 参数，而且所有方法都返回一个 **IntPtr**。每个方法都另有一个配对的方法，必须调用配对方法来清零并释放内部缓冲区。表 14-4 总结了 **System.Runtime.InteropServices.Marshal** 类提供的将 **SecureString** 解密到内部缓冲区的方法以及对应的清零和释放缓冲区的方法。

表 14-4　Marshal 类提供的用于操纵安全字符串的方法

将 SecureString 解密到缓冲的方法	清零并释放缓冲区的方法
SecureStringToBSTR	**ZeroFreeBSTR**
SecureStringToCoTaskMemAnsi	**ZeroFreeCoTaskMemAnsi**
SecureStringToCoTaskMemUnicode	**ZeroFreeCoTaskMemUnicode**
SecureStringToGlobalAllocAnsi	**ZeroFreeGlobalAllocAnsi**
SecureStringToGlobalAllocUnicode	**ZeroFreeGlobalAllocUnicode**



第 15 章　枚举类型和位标志

本章内容：

- 枚举类型
- 位标志
- 为枚举类型添加方法

本章要讨论枚举类型和位标志。由于 Microsoft Windows 和许多编程语言多年来一直在使用这些结构，相信许多人已经知道了如何使用它们。不过，CLR 与 FCL 结合起来之后，枚举类型和位标志才真正成为面向对象的类型。而它们提供的一些非常“酷”的功能，我相信大多数开发人员并不熟悉。让我惊讶的是，这些新功能极大地简化了应用程序开发，个中缘由且听我娓娓道来。

15.1　枚举类型

枚举类型(enumerated type)定义了一组“符号名称/值”配对。例如，以下 **Color** 类型定义了一组符号，每个符号都标识一种颜色：

```
internal enum Color {
   White,          // 赋值 0
   Red,            // 赋值 1
   Green,          // 赋值 2
   Blue,           // 赋值 3
   Orange          // 赋值 4
}
```

当然，也可写程序用 0 表示白色，用 1 表示红色，以此类推。不过，不应将这些数字硬编码到代码中，而应使用枚举类型，理由至少有二。

- 枚举类型使程序更容易编写、阅读和维护。有了枚举类型，符号名称可在代码中随便使用，程序员不用费心思量每个硬编码值的含义(例如，不用念叨 white 是 0，或者 0 是 white)。而且，一旦与符号名称对应的值发生改变，代码也可以简单地重新编译，不需要对源代码进行任何修改。此外，文档工具和其他实用程序(比如调试程序)能向开发人员显示有意义的符号名称。

- 枚举类型是强类型的。例如，将 **Color.Orange** 作为参数传给要求 **Fruit** 枚举类型的方

法，编译器会报错。[①]

在 Microsoft.NET Framework 中，枚举类型不只是编译器所关心的符号，它还是类型系统中的“一等公民”，能实现很强大的操作。而在其他环境(比如非托管 C++)中，枚举类型是没有这个特点的。

每个枚举类型都直接从 **System.Enum** 派生，后者从 **System.ValueType** 派生，而 **System.ValueType** 又从 **System.Object** 派生。所以，枚举类型是值类型(详情参见第 5 章“基元类型、引用类型和值类型”)，可用未装箱和已装箱的形式来表示。但有别于其他值类型，枚举类型不能定义任何方法、属性或事件。不过，可利用 C#的“扩展方法”功能模拟向枚举类型添加方法，15.3 节“向枚举类型添加方法”展示了一个例子。

编译枚举类型时，C#编译器把每个符号转换成类型的一个常量字段。例如，编译器将前面的 **Color** 枚举类型看成是以下代码：

```
internal struct Color : System.Enum {
   // 以下是一些公共常量，它们定义了 Color 的符号和值
   public const Color White    = (Color) 0;
   public const Color Red      = (Color) 1;
   public const Color Green    = (Color) 2;
   public const Color Blue     = (Color) 3;
   public const Color Orange   = (Color) 4;

   // 以下是一个公共实例字段，包含 Color 变量的值，
   // 不能写代码来直接引用该字段
   public Int32 value__;
}
```

C#编译器不会实际地编译上述代码，因为它禁止定义从特殊类型 **System.Enum** 派生的类型。不过，可通过上述伪类型定义了解内部的工作方式。简单地说，枚举类型只是一个结构，其中定义了一组常量字段和一个实例字段。常量字段会嵌入程序集的元数据中，并可通过反射来访问。这意味着可以在运行时获得与枚举类型关联的所有符号及其值。还意味着可以将字符串符号转换成对应的数值。这些操作是通过 **System.Enum** 基类型来提供的，该类型提供了几个静态和实例方法，可利用它们操作枚举类型的实例，从而避免了必须使用反射的麻烦。下面将讨论其中一些操作。



重要提示 枚举类型定义的符号是常量值。所以当编译器发现代码引用了枚举类型的符号时，会在编译时用数值替换符号，代码不再引用定义了符号的枚举类型。这意味着运行时可能并不需要定义了枚举类型的程序集，编译时才需要。假如代码引用了枚举类型(而非仅仅引用类型定义的符号)，那么运行时就需要包含了枚举类型定义的程序集。可能会出现一些版本问题，因为枚举类型符号是常量，而非只读的值。7.1 节“常量”已解释过这些问题。

① **ruit** 枚举类型定义的应该是水果，而 **Color** 枚举类型定义的是颜色。虽然两个枚举类型中都有一个 **Orange**，但分别代表橙子和橙色。——译注

例如，System.Enum 类型有一个名为 GetUnderlyingType 的静态方法，而 System.Type 类型有一个名为 GetEnumUnderlyingType 的实例方法：

```
public static Type GetUnderlyingType(Type enumType);      // System.Enum 中定义
public Type GetEnumUnderlyingType();                      // System.Type 中定义
```

这些方法返回用于容纳一个枚举类型的值的基础类型。每个枚举类型都有一个基础类型，它可以是 **byte**，**sbyte**，**short**，**ushort**，**int**(最常用，也是 C#默认选择的)，**uint**，**long** 或 **ulong**。虽然这些 C#基元类型[①]都有对应的 FCL 类型，但 C#编译器为了简化本身的实现，要求只能指定基元类型名称。如果使用 FCL 类型名称(比如 **Int32**)，会显示以下错误信息：**error CS1008: 应输入类型 byte、sbyte、short、ushort、int、uint、long 或 ulong**。

以下代码演示了如何声明一个基础类型为 **byte(System.Byte)**的枚举类型：

```
internal enum Color : byte {
   White,
   Red,
   Green,
   Blue,
   Orange
}
```

基于这个 **Color** 枚举类型，以下代码显示了 **GetUnderlyingType** 的返回结果：

```
// 以下代码会显示"System.Byte"
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

C# 编译器将枚举类型视为基元类型。所以可用许多熟悉的操作符(==，!=，<，>，<=，>=，+，-，^，&，|，~，++和--)来操纵枚举类型的实例。所有这些操作符实际作用于每个枚举类型实例内部的 **value__**实例字段。此外，C#编译器允许将枚举类型的实例显式转型为不同的枚举类型。也可显式将枚举类型实例转型为数值类型。

给定一个枚举类型的实例，可调用从 **System.Enum** 继承的 **ToString** 方法，把这个值映射为以下几种字符串表示：

```
Color c = Color.Blue;
Console.WriteLine(c);                    // "Blue" (常规格式)
Console.WriteLine(c.ToString());         // "Blue" (常规格式)
Console.WriteLine(c.ToString("G"));      // "Blue" (常规格式)
Console.WriteLine(c.ToString("D"));      // "3" (十进制格式)
Console.WriteLine(c.ToString("X"));      // "03" (十六进制格式)
```



注意　使用十六进制格式时，**ToString** 总是输出大写字母。此外，输出几位数取决于枚举的基础类型：**byte/sbyte** 输出 2 位数，**short/ushort** 输出 4 位数，**int/uint** 输出 8 位数，而 **long/ulong** 输出 16 位数。如有必要会添加前导零。

① 不要混淆“基础类型”和“基元类型”(参见 5.1 节“编程语言的基元类型”)。虽然枚举的基础类型就是这些基元类型(其实就是除 **Char** 之外的所有整型)，但英语中用 underlying type 和 primitive type 进行了区分，中文翻译同样要区分。简而言之，基元类型是语言的内建类型，编译器能直接识别。——译注

除了 **ToString** 方法，**System.Enum** 类型还提供了静态 **Format** 方法，可调用它格式化枚举类型的值：

```
public static String Format(Type enumType, Object value, String format);
```

个人倾向于调用 **ToString** 方法，因为它需要的代码更少，而且更容易调用。但 **Format** 有一个 **ToString** 没有的优势：允许为 **value** 参数传递数值。这样就不一定要有枚举类型的实例。例如，以下代码将显示“Blue”：

```
// 以下代码显示"Blue"
Console.WriteLine(Enum.Format(typeof(Color), 3, "G"));
```



注意 声明有多个符号的枚举类型时，所有符号都可以有相同的数值。使用常规格式将数值转换为符号时，**Enum** 的方法会返回其中一个符号，但不保证具体返回哪一个符号名称。另外，如果没有为要查找的数值定义符号，会返回包含该数值的字符串。

也可调用 **System.Enum** 的静态方法 **GetValues** 或者 **System.Type** 的实例方法 **GetEnumValues** 来返回一个数组，数组中的每个元素都对应枚举类型中的一个符号名称，每个元素都包含符号名称的数值：

```
public static Array GetValues(Type enumType);    // System.Enum 中定义
public Array GetEnumValues();                     // System.Type 中定义
```

该方法可以结合 **ToString** 方法使用以显示枚举类型中的所有符号名称及其对应数值，如下所示：

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n-----\t------");
foreach (Color c in colors) {
// 以十进制和常规格式显示每个符号
   Console.WriteLine("{0,5:D}\t{0:G}", c);
}
```

以上代码的输出如下：

```
Number of symbols defined: 5
Value   Symbol
-----   ------
    0   White
    1   Red
    2   Green
    3   Blue
    4   Orange
```

我个人不喜欢 **GetValues** 和 **GetEnumValues** 方法，因为两者均返回一个 **Array**，必须转型成恰当的数组类型。所以我总是定义自己的方法：

```
public static TEnum[] GetEnumValues<TEnum>() where TEnum : struct {
   return (TEnum[])Enum.GetValues(typeof(TEnum));
}
```

使用我的泛型 **GetEnumValues** 方法可获得更好的编译时类型安全性，而且上例的第一行代

码可简化成以下形式：

```
Color[] colors = GetEnumValues<Color>();
```

前面的讨论展示了可以对枚举类型执行的一些很“酷”的操作。在程序的 UI 元素(列表框、组合框等)中显示符号名称时，我认为经常使用的会是 **ToString** 方法(常规格式)，前提是字符串不需要本地化(因为枚举类型没有提供本地化支持)。除了 **GetValues** 方法，**System.Enum** 和 **System.Type** 类型还提供了以下方法来返回枚举类型的符号：

```
// 返回数值的字符串表示
public static String GetName(Type enumType, Object value);      // System.Enum 中定义
public String GetEnumName(Object value);                        // System.Type 中定义

// 返回一个 String 数组，枚举中每个符号都对应一个 String
public static String[] GetNames(Type enumType);                  // System.Enum 中定义
public String[] GetEnumNames();                                  // System.Type 中定义
```

前面讨论了用于查找枚举类型中的符号的多种方法。但还需要一个方法来查找与符号对应的值。例如，可利用这个操作转换用户在文本框中输入的一个符号。利用 **Enum** 提供的静态 **Parse** 和 **TryParse** 方法，可以很容易地将符号转换为枚举类型的实例：

```
public static Object Parse(Type enumType, String value);
public static Object Parse(Type enumType, String value, Boolean ignoreCase);
public static Boolean TryParse<TEnum>(String value, out TEnum result) where TEnum: struct;
public static Boolean TryParse<TEnum>(String value, Boolean ignoreCase, out TEnum result)
   where TEnum : struct;
```

以下代码演示了如何使用这些方法：

```
// 因为 Orange 定义为 4，'c'被初始化为 4
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);

// 因为没有定义 Brown，所以抛出 ArgumentException 异常
Color c = (Color) Enum.Parse(typeof(Color), "Brown", false);

// 创建值为 1 的 Color 枚举类型实例
Enum.TryParse<Color>("1", false, out c);

// 创建值为 23 的 Color 枚举类型实例
Enum.TryParse<Color>("23", false, out c);
```

以下是 **Enum** 的静态 **IsDefined** 方法和 **Type** 的 **IsEnumDefined** 方法：

```
public static Boolean IsDefined(Type enumType, Object value);   // System.Enum 中定义
public Boolean IsEnumDefined(Object value);                     // System.Type 中定义
```

可利用 **IsDefined** 方法判断数值对于某枚举类型是否合法：

```
// 显示"True"，因为 Color 将 Red 定义为 1
Console.WriteLine(Enum.IsDefined(typeof(Color), 1));

// 显示"True"，因为 Color 将 White 定义为 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "White"));

// 显示"False"，因为检查要区分大小写
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));

// 显示"False"，因为 Color 没有和值 10 对应的符号
Console.WriteLine(Enum.IsDefined(typeof(Color), 10));
```

IsDefined 方法被经常用于参数校验，如下例所示：

```
public void SetColor(Color c) {
   if (!Enum.IsDefined(typeof(Color), c)) {
   throw(new ArgumentOutOfRangeException("c", c, "无效颜色值。"));
   }
   // 将颜色设置为 White，Red，Green，Blue 或 Orange
   ...
}
```

参数校验是很有用的一个功能，因为其他人可能像下面这样调用 **SetColor**：

```
SetColor((Color) 547);
```

没有和值 547 对应的符号，所以 **SetColor** 方法抛出 **ArgumentOutOfRangeException** 异常，指出哪个参数无效，并解释为什么无效。



重要提示 **IsDefined** 方法很方便，但必须慎用。首先，**IsDefined** 总是执行区分大小写的查找，而且完全没有办法让它执行不区分大小写的查找。其次，**IsDefined** 相当慢，因为它在内部使用了反射。如果写代码来手动检查每一个可能的值，应用程序的性能极有可能变得更好。最后，只有当枚举类型本身在调用 **IsDefined** 的同一个程序集中定义时，才可使用 **IsDefined**。原因如下：假定 **Color** 枚举类型在一个程序集中定义，**SetColor** 方法在另一个程序集中定义。**SetColor** 方法调用 **IsDefined**，假如颜色是 **White**，**Red**，**Green**，**Blue** 或者 **Orange**，那么 **SetColor** 能正常执行。然而，假如 **Color** 枚举将来发生了变化，在其中包含了 **Purple**，那么 **SetColor** 现在就会接受 **Purple**，这是以前没有预料到的。因此，方法现在可能返回无法预料的结果。

最后，**System.Enum** 类型提供了一组静态 **ToObject** 方法。这些方法将 **Byte**，**SByte**，**Int16**，**UInt16**，**Int32**，**UInt32**，**Int64** 或 **UInt64** 类型的实例转换为枚举类型的实例。

枚举类型总是要与另外某个类型结合使用，一般作为类型的方法参数或返回类型、属性和字段使用。初学者经常提出的一个问题是：枚举类型是嵌套定义在需要它的类型中，还是和该类型同级？检查 FCL，会发现枚举类型通常与需要它的类同级。原因很简单，就是减少代码的录入量，使开发人员的工作变得更轻松。所以，除非担心名称冲突，否则你定义的枚举类型应该和需要它的类型同级。

15.2 位标志

程序员经常要和位标志(bit flag)集合打交道。调用 **System.IO.File** 类型的 **GetAttributes** 方法，会返回 **FileAttributes** 类型的一个实例。**FileAttributes** 类型是基本类型为 **Int32** 的枚举类型，其中每一位都反映了文件的一个特性(attribute)。**FileAttributes** 类型在 FCL 中的定义如下：

```
[Flags, Serializable]
public enum FileAttributes {
    ReadOnly          = 0x0001,
    Hidden            = 0x0002,
    System            = 0x0004,
    Directory         = 0x0010,
    Archive           = 0x0020,
    Device            = 0x0040,
    Normal            = 0x0080,
    Temporary         = 0x0100,
    SparseFile        = 0x0200,
    ReparsePoint      = 0x0400,
    Compressed        = 0x0800,
    Offline           = 0x1000,
    NotContentIndexed = 0x2000,
    Encrypted         = 0x4000
    IntegrityStream   = 0x08000,
    NoScrubData       = 0x20000
}
```

判断文件是否隐藏可执行以下代码：

```
String file = Assembly.GetEntryAssembly().Location;
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file, (attributes & FileAttributes.Hidden) != 0);
```



注意　**Enum** 类定义了一个 **HasFlag** 方法：

```
public Boolean HasFlag(Enum flag);
```

可利用该方法重写上述 **Console.WriteLine** 调用：

```
Console.WriteLine("Is {0} hidden? {1}", file,
   attributes.HasFlag(FileAttributes.Hidden));
```

但我建议避免使用 **HasFlag** 方法，理由是：由于它获取 **Enum** 类型的参数，所以传给它的任何值都必须装箱，产生一次内存分配。

以下代码演示了如何为文件设置只读和隐藏特性：

```
File.SetAttributes(file, FileAttributes.ReadOnly | FileAttributes.Hidden);
```

正如 **FileAttributes** 类型展示的那样，经常都要用枚举类型来表示一组可以组合的位标志。不过，虽然枚举类型和位标志相似，但它们的语义不尽相同。例如，枚举类型表示单个数值，而位标志表示位集合，其中一些位处于 on 状态，一些处于 off 状态[①]。

定义用于标识位标志的枚举类型时，当然应该显式为每个符号分配一个数值。通常，每个符号都有单独的一个位处于 on 状态。此外，经常都要定义一个值为 **0** 的 **None** 符号。还可定义一些符合来代表常见的位组合 (参见下面的 **ReadWrite** 符号)。另外，强烈建议向枚举类型应用定制特性类型 **System.FlagsAttribute**，如下所示：

```
[Flags] // C#编译器允许"Flags"或"FlagsAttribute"
internal enum Actions {
   None           = 0
   Read           = 0x0001,
```

① 进制 1 代表“on”，二进制 0 代表“off”。——译注

```
   Write          = 0x0002,
   ReadWrite      = Actions.Read | Actions.Write,
   Delete         = 0x0004,
   Query          = 0x0008,
   Sync           = 0x0010
}
```

由于 **Actions** 是枚举类型，所以在操纵位标志枚举类型时，可以使用上一节描述的所有方法。不过，假如其中一些方法的行为稍有区别，效果会更加理想。例如，假设有以下代码：

```
Actions actions = Actions.Read | Actions.Delete;     // 0x0005
Console.WriteLine(actions.ToString());               // "Read, Delete"
```

调用 **ToString** 时，它会试图将数值转换为对应的符号。现在的数值是 **0x0005**，没有对应的符号。不过，**ToString** 方法检测到 **Actions** 类型上存在**[Flags]**特性，所以 **ToString** 方法现在不会将该数值视为单独的值。相反，会把它视为一组位标志。由于 **0x0005** 由 **0x0001** 和 **0x0004** 组合而成，所以 **ToString** 会生成字符串**"Read, Delete"**。从 **Actions** 类型中删除**[Flags]**特性，**ToString** 方法将返回**"5"**。

上一节已讨论了 **ToString** 方法，指出它允许以 3 种方式格式化输出："G"(常规)、"D"(十进制)和"X"(十六进制)。使用常规格式来格式化枚举类型的实例时，首先会检查类型，看它是否应用了**[Flags]**这个特性。没有应用就查找与该数值匹配的符号并返回符号。如果应用了**[Flags]**特性，**ToString** 方法的工作过程如下所示。

1. 获取枚举类型定义的数值集合，降序排列这些数值。

2. 每个数值都和枚举实例中的值进行“按位与”计算，假如结果等于数值，与该数值关联的字符串就附加到输出字符串上，对应的位会被认为已经考虑过了，会被关闭(设为 0)。这一步不断重复，直到检查完所有数值，或直到枚举实例的所有位都被关闭。

3. 检查完所有数值后，如果枚举实例仍然不为 0，表明枚举实例中一些处于 on 状态的位不对应任何已定义的符号。在这种情况下，**ToString** 将枚举实例中的原始数值作为字符串返回。

4. 如果枚举实例原始值不为 0，返回符号之间以逗号分隔的字符串。

5. 如果枚举实例原始值为 0，而且枚举类型定义的一个符号对应的是 0 值，就返回这个符号。

6. 如果到达这一步，就返回“0”。

如果愿意，可定义没有**[Flags]**特性的 **Actions** 类型，并用"F"格式获得正确的字符串：

```
// [Flags]                    // 现在已经被注释掉
internal enum Actions {
   None        = 0
   Read        = 0x0001,
   Write       = 0x0002,
   ReadWrite   = Actions.Read | Actions.Write,
   Delete      = 0x0004,
   Query       = 0x0008,
   Sync        = 0x0010
}
```

```
Actions actions = Actions.Read | Actions.Delete;     // 0x0005
Console.WriteLine(actions.ToString("F"));            // "Read, Delete"
```

如果数值有一个位不能映射到一个符号，返回的字符串只包含一个代表原始数值的十进制数；字符串中不会有符号。

注意，枚举类型中定义的符号不一定是 2 的整数次方。例如，**Actions** 类型可定义一个名为 **All** 的符号，它对应的值是 0x001F[①]。如果 **Actions** 类型的一个实例的值是 0x001F，格式化该实例就会生成一个含有**"All"**的字符串。其他符号字符串不会出现。

前面讨论的是如何将数值转换成标志字符串(string of flag)。还可将以逗号分隔的符号字符串转换成数值，这是通过调用 **Enum** 的静态方法 **Parse** 和 **TryParse** 来实现的。以下代码演示了如何使用这些方法：

```
// 由于 Query 被定义为 8，所以'a'被初始化为 8
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);
Console.WriteLine(a.ToString()); // "Query"

// 由于 Query 和 Read 已定义，所以'a'被初始化为 9
Enum.TryParse<Actions>("Query, Read", false, out a);
Console.WriteLine(a.ToString()); // "Read, Query"

// 创建一个 Actions 枚举类型实例，其值为 28
a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString()); // "Delete, Query, Sync"
```

Parse 和 **TryParse** 方法在调用时，会在内部执行以下动作。

1. 删除字符串头尾的所有空白字符。
2. 如果字符串第一个字符是数字、加号(+)或减号(-)，该字符串会被认为是一个数字，方法返回一个枚举类型实例，其数值等于字符串转换后的数值。
3. 传递的字符串被分解为一组以逗号分隔的 token，每个 token 的空白字符都被删除。
4. 在枚举类型的已定义符号中查找每个 token 字符串。如果没有找到相应的符号，**Parse** 会抛出 **System.ArgumentException** 异常；而 **TryParse** 会返回 **false**。如果找到符号，就将它对应的数值与当前的一个动态结果进行“按位或”计算，再查找下一个符号。
5. 查找并找到了所有标记之后，返回这个动态结果。

永远不要对位标志枚举类型使用 **IsDefined** 方法。以下两方面原因造成该方法无法使用。

- 向 **IsDefined** 方法传递字符串，它不会将这个字符串拆分为单独的 token 来进行查找，而是试图查找整个字符串，把它看成是包含逗号的一个更大的符号。由于不能在枚举类型中定义含有逗号的符号，所以这个符号永远找不到。

---

① 计算可知，二进制 00000001(Read) | 00000010(Write) | 00000100(Delete) | 00001000(Query) | 00010000(Sync) = 00011111(All) = 十六进制 0x001F。——译注

- 向 **IsDefined** 方法传递一个数值，它会检查枚举类型是否定义了其数值和传入数值匹配的一个符号。由于位标志不能这样简单地匹配[①]，所以 **IsDefined** 通常会返回 **false**。

15.3 向枚举类型添加方法

本章早些时候曾指出，不能将方法定义为枚举类型的一部分。多年以来，我对此一直感到很“郁闷”，因为很多时候都需要为我的枚举类型提供一些方法。幸好，现在可以利用 C# 的扩展方法功能(参见第 8 章“方法”)模拟向枚举类型添加方法。

要为 **FileAttributes** 枚举类型添加方法，先定义一个包含了扩展方法的静态类，如下所示：

```
internal static class FileAttributesExtensionMethods {
   public static Boolean IsSet(this FileAttributes flags, FileAttributes flagToTest) {
      if (flagToTest == 0)
         throw new ArgumentOutOfRangeException("flagToTest", "Value must not be 0");
      return (flags & flagToTest) == flagToTest;
   }

   public static Boolean IsClear(this FileAttributes flags, FileAttributes flagToTest) {
      if (flagToTest == 0)
         throw new ArgumentOutOfRangeException("flagToTest", "Value must not be 0");
      return !IsSet(flags, flagToTest);
   }

   public static Boolean AnyFlagsSet(this FileAttributes flags, FileAttributes testFlags) {
      return ((flags & testFlags) != 0);
   }

   public static FileAttributes Set(this FileAttributes flags, FileAttributes setFlags) {
      return flags | setFlags;
   }

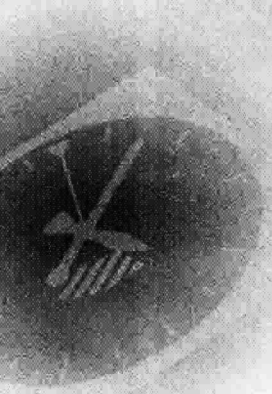
   public static FileAttributes Clear(this FileAttributes flags,
      FileAttributes clearFlags) {
      return flags & ~clearFlags;
}

   public static void ForEach(this FileAttributes flags,
      Action<FileAttributes> processFlag) {
      if (processFlag == null) throw new ArgumentNullException("processFlag");
      for (UInt32 bit = 1; bit != 0; bit <<= 1) {
         UInt32 temp = ((UInt32)flags) & bit;
         if (temp != 0) processFlag((FileAttributes)temp);
      }
   }
}
```

以下代码演示了如何调用其中的一些方法。从表面上看，似乎真的是在枚举类型上调用这些方法：

```
FileAttributes fa = FileAttributes.System;
fa = fa.Set(FileAttributes.ReadOnly);
fa = fa.Clear(FileAttributes.System);
fa.ForEach(f => Console.WriteLine(f));
```

① 因为 bit flag 一般都要组合起来使用。——译注



第16章 数 组

本章内容：

- 初始化数组元素
- 数组转型
- 所有数组都隐式派生自 **System.Array**
- 所有数组都隐式实现 **IEnumerable**，**ICollection** 和 **IList**
- 数组的传递和返回
- 创建下限非零的数组
- 数组的内部工作原理
- 不安全的数组访问和固定大小的数组

数组是允许将多个数据项作为集合来处理的机制。CLR 支持一维、多维和交错数组(即数组构成的数组)。所有数组类型都隐式地从 **System.Array** 抽象类派生，后者又派生自 **System.Object**。这意味着数组始终是引用类型，是在托管堆上分配的。在应用程序的变量或字段中，包含的是对数组的引用，而不是包含数组本身的元素。下面的代码更清楚地说明了这一点：

```
Int32[] myIntegers;              // 声明一个数组引用
myIntegers = new Int32[100];     // 创建含有 100 个 Int32 的数组
```

第一行代码声明 **myIntegers** 变量，它能指向包含 **Int32** 值的一维数组。**myIntegers** 刚开始被设为 **null**，因为当时还没有分配数组。第二行代码分配了含有 100 个 **Int32** 值的数组，所有 **Int32** 都被初始化为 0。由于数组是引用类型，所以会在托管堆上分配容纳 100 个未装箱 **Int32** 所需的内存块。实际上，除了数组元素，数组对象占据的内存块还包含一个类型对象指针、一个同步块索引和一些额外的成员[①]。该数组的内存块地址被返回并保存到 **myIntegers** 变量中。

还可创建引用类型的数组：

```
Control[] myControls;            // 声明一个数组引用
myControls = new Control[50];    // 创建含有 50 个 Control 引用的数组
```

第一行代码声明 **myControls** 变量，它能指向包含 **Control** 引用的一维数组。**myControls**

① 这些额外的成员称为 overhead 字段或者说“开销字段”。——译注

刚开始被设为 **null**，因为当时还没有分配数组。第二行代码分配了含有 50 个 **Control** 引用的数组，这些引用全被初始化为 **null**。由于 **Control** 是引用类型，所以创建数组只是创建了一组引用，此时没有创建实际的对象。这个内存块的地址被返回并保存到 **myControls** 变量中。

图 16-1 展示了值类型的数组和引用类型的数组在托管堆中的情况。

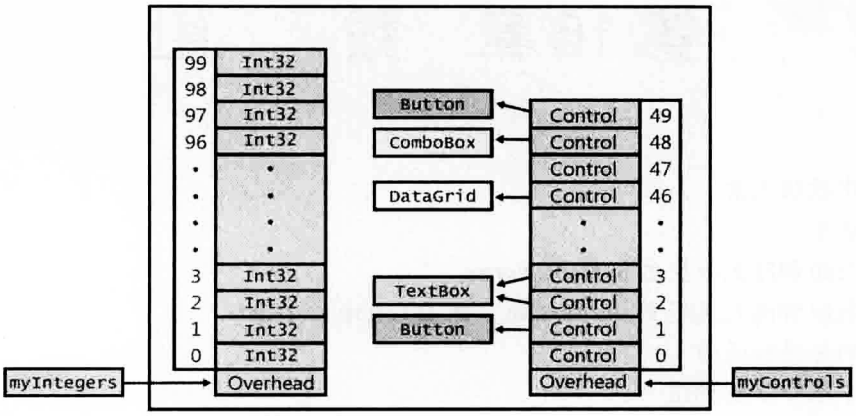


图 16-1　值类型和引用类型的数组在托管堆中的情况

在图 16-1 中，**Control** 数组显示了执行以下各行代码之后的结果：

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2];      // 两个元素引用同一个对象
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

为了符合“公共语言规范”(Common Language Specification，CLS)的要求，所有数组都必须是 0 基数组(即最小索引为 0)。这样就可以用 C#的方法创建数组，并将该数组的引用传给用其他语言(比如 Microsoft Visual Basic .NET)写的代码。此外，由于 0 基数组是最常用的数组(至少就目前而言)，所以 Microsoft 花了很大力气优化性能。不过，CLR 确实支持非 0 基数组，只是不提倡使用。对于不介意稍许性能下降或者跨语言移植问题的读者，本章后文要介绍如何创建和使用非 0 基数组。

注意在图 16-1 中，每个数组都关联了一些额外的开销信息。这些信息包括数组的秩[①]、数组每一维的下限(几乎总是 0)和每一维的长度。开销信息还包含数组的元素类型。本章后文将介绍查询这种开销信息的方法。

前面已通过几个例子演示了如何创建一维数组。应尽可能使用一维 0 基数组，有时也将这种数组称为 SZ[②]数组或向量(vector)。向量的性能是最佳的，因为可以使用一些特殊的 IL 指

① 即 rank，或称数组的维数。——译注

② SZ 是 single-dimension，zero-based(一维 0 基)的简称。——译注

令(比如 **newarr**，**ldelem**，**ldelema**，**ldlen** 和 **stelem**)来处理。不过，必要时也可使用多维数组。下面展示了几个多维数组的例子：

```
// 创建一个二维数组，由 Double 值构成
Double[,] myDoubles = new Double[10, 20];

// 创建一个三维数组，由 String 引用构成
String[,,] myStrings = new String[5, 3, 10];
```

CLR 还支持交错数组(jagged array)，即数组构成的数组。0 基一维交错数组的性能和普通向量一样好。不过，访问交错数组的元素意味着必须进行两次或更多次数组访问。下例演示了如何创建一个多边形数组，每个多边形都由一个包含 **Point** 实例的数组构成：

```
// 创建由多个 Point 数组构成的一维数组
Point[][] myPolygons = new Point[3][];

// myPolygons[0]引用一个含有 10 个 Point 实例的数组
myPolygons[0] = new Point[10];

// myPolygons[1]引用一个含有 20 个 Point 实例的数组
myPolygons[1] = new Point[20];

// myPolygons[2] 引用一个含有 30 个 Point 实例的数组
myPolygons[2] = new Point[30];

// 显示第一个多边形中的 Point
for (Int32 x = 0; x < myPolygons[0].Length; x++)
   Console.WriteLine(myPolygons[0][x]);
```



注意　CLR 会验证数组索引的有效性。换句话说，不能创建含有 100 个元素的数组(索引编号 0 到 99)，然后试图访问索引为-5 或 100 的元素。这样做会导致 **System.IndexOutOfRangeException** 异常。允许访问数组范围之外的内存会破坏类型安全性，而且会造成潜在的安全漏洞，所以 CLR 不允许可验证的代码这么做。通常，索引范围检查对性能的影响微乎其微，因为 JIT 编译器通常只在循环开始之前检查一次数组边界，而不是每次循环迭代都检查[①]。不过，如果仍然担心 CLR 索引检查造成的性能损失，可以在 C#中使用 unsafe 代码来访问数组。16.7 节“数组的内部工作原理”将演示具体做法。

16.1　初始化数组元素

前面展示了如何创建数组对象，如何初始化数组中的元素。C#允许用一个语句做这两件事情。例如：

---

① 不要混淆“循环”和“循环迭代”。例如以下代码：

```
Int32[] myArray = new Int32[100];
for (Int32 i = 0; i < myArray.Length; i++) myArray[i] = i;
```

“**for** 循环”总共要“循环迭代 100 次”，有时也简单地说“迭代 100 次”。——译注

```
String[] names = new String[] { "Aidan", "Grant" };
```

大括号中的以逗号分隔的数据项称为**数组初始化器**(array initializer)。每个数据项都可以是一个任意复杂度的表达式；在多维数组的情况下，则可以是一个嵌套的数组初始化器。上例只使用了两个简单的 **String** 表达式。

在方法中声明局部变量来引用初始化好的数组时，可利用 C#的“隐式类型的局部变量”功能来简化一下代码：

```
// 利用 C#的隐式类型的局部变量功能：
var names = new String[] { "Aidan", "Grant" };
```

编译器推断局部变量 **names** 是 **String[]**类型，因为那是赋值操作符(=)右侧的表达式的类型。

可利用 C#的隐式类型的数组功能让编译器推断数组元素的类型。注意，下面这行代码没有在 **new** 和**[]**之间指定类型。

```
// 利用 C#的隐式类型的局部变量和隐式类型的数组功能：
var names = new[] { "Aidan", "Grant", null };
```

在上一行中，编译器检查数组中用于初始化数组元素的表达式的类型，并选择所有元素最接近的共同基类来作为数组的类型。在本例中，编译器发现两个 **String** 和一个 **null**。由于 **null** 可隐式转型为任意引用类型(包括 **String**)，所以编译器推断应该创建和初始化一个由 **String** 引用构成的数组。但假如写以下代码：

```
// 使用 C#的隐式类型的局部变量和隐式类型的数组功能：(错误)
var names = new[] { "Aidan", "Grant", 123 };
```

编译器就会报错：**error CS0826: 找不到隐式类型数组的最佳类型**。这是由于两个 **String** 和一个 **Int32** 的共同基类是 **Object**，意味着编译器不得不创建 **Object** 引用的一个数组，然后对 **123** 进行装箱，并让最后一个数组元素引用已装箱的、值为 **123** 的一个 **Int32**。C#团队认为，隐式对数组元素进行装箱是一个代价高昂的操作，所以要在编译时报错。

作为初始化数组时的一个额外的语法奖励，还可以像下面这样写：

```
String[] names = { "Aidan", "Grant" };
```

注意，赋值操作符(=)右侧只给出了一个初始化器，没有 **new**，没有类型，没有**[]**。这个语法可读性很好，但遗憾的是，C#编译器不允许在这种语法中使用隐式类型的局部变量：

```
// 试图使用隐式类型的局部变量(错误)
var names = { "Aidan", "Grant" };
```

试图编译上面这行代码，编译器会报告以下两条消息。

- error CS0820：无法用数组初始值设定项初始化隐式类型的局部变量。
- error CS0622：只能使用数组初始值设定项表达式为数组类型赋值，请尝试改用 new 表达式。

虽然理论上可以通过编译，但 C#团队认为编译器在这里会为你做太多的工作。它要推断数

组类型，新建数组对象，初始化数组，还要推断局部变量的类型。

最后讲一下“隐式类型的数组”如何与“匿名类型”和“隐式类型的局部变量”组合使用。10.1.4 节“匿名类型”已讨论了匿名类型以及如何保证类型同一性。下面来看看以下代码：

```
// 使用 C#的隐式类型的局部变量、隐式类型的数组和匿名类型功能：
var kids = new[] {new { Name="Aidan" }, new { Name="Grant" }};

// 示例用法(用了另一个隐式类型的局部变量)：
foreach (var kid in kids)
   Console.WriteLine(kid.Name);
```

这个例子在一个数组初始化器中添加了两个用于定义数组元素的表达式。每个表达式都代表一个匿名类型(因为 **new** 操作符后没有提供类型名称)。由于两个匿名类型具有一致的结构(有一个 **String** 类型的 **Name** 字段)，所以编译器知道这两个对象具有相同的类型(类型的同一性)。然后，我使用了 C#的“隐式类型的数组”功能(在 **new** 和**[]**之间不指定类型)，让编译器推断数组本身的类型，构造这个数组对象，并初始化它内部的两个引用，指向匿名类型的两个实例。最后，将对这个数组对象的引用赋给 **kids** 局部变量，该变量的类型通过 C#的“隐式类型的局部变量”功能来推断。

第二行代码用 **foreach** 循环演示如何使用刚才创建的、用两个匿名类型实例初始化的数组。注意必须为循环使用隐式类型的局部变量(**kid**)。运行这段代码将得到以下输出：

```
Aidan
Grant
```

16.2　数组转型

对于元素为引用类型的数组，CLR 允许将数组元素从一种类型转型另一种。成功转型要求数组维数相同，而且必须存在从元素源类型到目标类型的隐式或显式转换。CLR 不允许将值类型元素的数组转型为其他任何类型。(不过，可用 **Array.Copy** 方法创建新数组并在其中填充元素来模拟这种效果。)[①]以下代码演示了数组转型：

```
// 创建二维 FileStream 数组
FileStream[,] fs2dim = new FileStream[5, 10];

// 隐式转型为二维 Object 数组
Object[,] o2dim = fs2dim;

// 二维数组不能转型为一维数组，编译器报错：
// error CS0030: 无法将类型"object[*,*]"转换为"System.IO.Stream[]"
Stream[] s1dim = (Stream[]) o2dim;

// 显式转型为二维 Stream 数组
Stream[,] s2dim = (Stream[,]) o2dim;

// 显式转型为二维 String 数组
// 能通过编译，但在运行时抛出 InvalidCastException 异常
```

① 注意，**Array.Copy** 方法执行的是浅拷贝。换言之，如果数组元素是引用类型，新数组将引用现有的对象。——译注

```
String[,] st2dim = (String[,]) o2dim;

// 创建一维 Int32 数组(元素是值类型)
Int32[] ildim = new Int32[5];

// 不能将值类型的数组转型为其他任何类型，编译器报错：
// error CS0030：无法将类型"int[]"转换为"object[]"
Object[] oldim = (Object[]) ildim;

// 创建一个新数组，使用 Array.Copy 将源数组中的每个元素
// 转型为目标数组中的元素类型，并把它们复制过去。
// 下面的代码创建元素为引用类型的数组，
// 每个元素都是对已装箱 Int32 的引用
Object[] obldim = new Object[ildim.Length];
Array.Copy(ildim, obldim, ildim.Length);
```

Array.Copy 的作用不仅仅是将元素从一个数组复制到另一个。**Copy** 方法还能正确处理内存的重叠区域，就像 C 的 **memmove** 函数一样。有趣的是，C 的 **memcpy** 函数反而不能正确处理重叠的内存区域。**Copy** 方法还能在复制每个数组元素时进行必要的类型转换，具体如下所述：

- 将值类型的元素装箱为引用类型的元素，比如将一个 **Int32[]**复制到一个 **Object[]**中。
- 将引用类型的元素拆箱为值类型的元素，比如将一个 **Object[]**复制到一个 **Int32[]**中。
- 加宽 CLR 基元值类型，比如将一个 **Int32[]**的元素复制到一个 **Double[]**中。
- 在两个数组之间复制时，如果仅从数组类型证明不了两者的兼容性，比如从 **Object[]** 转型为 **IFormattable[]**，就根据需要对元素进行向下类型转换。如果 **Object[]**中的每个对象都实现了 **IFormattable**，**Copy** 方法就能成功执行。

下面演示了 **Copy** 方法的另一种用法：

```
// 定义实现了一个接口的值类型
internal struct MyValueType : IComparable {
   public Int32 CompareTo(Object obj) {
      ...
   }
}

public static class Program {
   public static void Main() {
      // 创建含有 100 个值类型的数组
      MyValueType[] src = new MyValueType[100];

      // 创建 IComparable 引用数组
      IComparable[] dest = new IComparable[src.Length];

      // 初始化 IComparable 数组中的元素，
      // 使它们引用源数组元素的已装箱版本
      Array.Copy(src, dest, src.Length);
   }
}
```

地球人都能猜得到，FCL 频繁运用了 **Array** 的 **Copy** 方法。

有时确实需要将数组从一种类型转换为另一种类型。这种功能称为**数组协变性**(array

covariance)。但在利用它时要清楚由此而来的性能损失。假设有以下代码：

```
String[] sa = new String[100];
Object[] oa = sa;       // oa 引用一个 String 数组
oa[5] = "Jeff";         // 性能损失：CLR 检查 oa 的元素类型是不是 String；检查通过
oa[3] = 5;              // 性能损失：CLR 检查 oa 的元素类型是不是 Int32；发现有错，
                        // 抛出 ArrayTypeMismatchException 异常
```

在上述代码中，**oa** 变量被定义为 **Object[]**类型，但实际引用的是一个 **String[]**。编译器允许代码将 5 放到数组元素中，因为 5 是 **Int32**，而 **Int32** 派生自 **Object**。虽然编译能通过，但 CLR 必须保证类型安全。对数组元素赋值时，它必须保证赋值的合法性。所以，CLR 必须在运行时检查数组包含的是不是 **Int32** 元素。在本例中，答案是否定的，所以不允许赋值；CLR 抛出 **ArrayTypeMismatchException** 异常。



注意　如果只是需要将数组的某些元素复制到另一个数组，可选择 **System.Buffer** 的 **BlockCopy** 方法，它比 **Array** 的 **Copy** 方法快。但 **Buffer** 的 **BlockCopy** 方法只支持基元类型，不提供像 **Array** 的 **Copy** 方法那样的转型能力。方法的 **Int32** 参数代表的是数组中的字节偏移量，而非元素索引。设计 **BlockCopy** 的目的实际是将按位兼容(bitwise-compatible)[①]的数据从一个数组类型复制到另一个按位兼容的数据类型，比如将包含 Unicode 字符的一个 **Byte[]**(按字节的正确顺序)复制到一个 **Char[]**中。该方法一定程度上弥补了不能将数组当作任意类型的内存块来处理的不足。

要将一个数组的元素可靠地复制到另一个数组，应该使用 **System.Array** 的 **ConstrainedCopy** 方法。该方法要么完成复制，要么抛出异常，总之不会破坏目标数组中的数据。这就允许 **ConstrainedCopy** 在约束执行区域(Constrained Execution Region，CER)中执行。为了提供这种保证，**ConstrainedCopy** 要求源数组的元素类型要么与目标数组的元素类型相同，要么派生自目标数组的元素类型。另外，它不执行任何装箱、拆箱或向下类型转换。

16.3　所有数组都隐式派生自 System.Array

像下面这样声明数组变量：

```
FileStream[] fsArray;
```

CLR 会自动为 AppDomain 创建一个 **FileStream[]**类型。该类型隐式派生自 **System.Array**

① “按位兼容”因为英文原文是 bitwise-compatible，所以人们发明了 blittable 一词来表示这种类型。这种类型在托管和非托管内存中具有相同的表示。一部分基元类型是 blittable 类型。如果一维数组包含的是 blittable 类型，该数组也是 blittable 类型。另外，格式化的值类型如果只包含 blittable 类型，该值类型也是 blittable 类型。欲知详情，请在 MSDN 文档中搜索“可直接复制到本机结构中的类型和非直接复制到本机结构中的类型”这一主题。——译注

类型；因此，**System.Array** 类型定义的所有实例方法和属性都将由 **FileStream[]**继承，使这些方法和属性能通过 **fsArray** 变量调用。这极大方便了数组处理，因为 **System.Array** 定义了许多有用的实例方法和属性，比如 **Clone**，**CopyTo**，**GetLength**，**GetLongLength**，**GetLowerBound**，**GetUpperBound**，**Length**，**Rank** 等。

System.Array 类型还公开了很多有用的、用于数组处理的静态方法。这些方法均获取一个数组引用作为参数。一些有用的静态方法包括：**AsReadOnly**，**BinarySearch**，**Clear**，**ConstrainedCopy**，**ConvertAll**，**Copy**，**Exists**，**Find**，**FindAll**，**FindIndex**，**FindLast**，**FindLastIndex**，**ForEach**，**IndexOf**，**LastIndexOf**，**Resize**，**Reverse**，**Sort** 和 **TrueForAll**。这些方法中，每个都有多个重载版本。事实上，许多方法还提供了泛型重载版本，能保障编译时的类型安全性和良好的性能。鼓励大家查阅文档，体会这些方法究竟多么有用和强大！

16.4 所有数组都隐式实现 IEnumerable，ICollection 和 IList

许多方法都能操纵各种各样的集合对象，因为它们声明为允许获取 **IEnumerable**，**ICollection** 和 **IList** 等参数。可将数组传给这些方法，因为 **System.Array** 也实现了这三个接口。**System.Array** 之所以实现这些非泛型接口，是因为这些接口将所有元素都视为 **System.Object**。然而，最好是让 **System.Array** 实现这些接口的泛型形式，提供更好的编译时类型安全性和更好的性能。

不过，由于涉及多维数组和非 0 基数组的问题，CLR 团队不希望 **System.Array** 实现 **IEnumerable<T>**，**ICollection<T>**和 **IList<T>**。若在 **System.Array** 上定义这些接口，就会为所有数组类型启用这些接口。所以，CLR 没有那么做，而是耍了一个小花招：创建一维 0 基数组类型时，CLR 自动使数组类型实现 **IEnumerable<T>**，**ICollection<T>**和 **IList<T>**(T 是数组元素的类型)。同时，还为数组类型的所有基类型实现这三个接口，只要它们是引用类型。以下层次结构图对此进行了澄清：

```
Object
  Array (非泛型 IEnumerable, ICollection, IList)
    Object[]              (IEnumerable, ICollection, IList of Object)
      String[]            (IEnumerable, ICollection, IList of String)
      Stream[]            (IEnumerable, ICollection, IList of Stream)
        FileStream[]      (IEnumerable, ICollection, IList of FileStream)
      .
      .     (其他引用类型的数组)
      .
```

所以，如果执行以下代码：

```
FileStream[] fsArray;
```

那么当 CLR 创建 **FileStream[]**类型时，会自动为这个类型实现 **IEnumerable<FileStream>**，**ICollection<FileStream>**和 **IList<FileStream>**接口。此外，**FileStream[]**类型还会为基类型实现接口：**IEnumerable<Stream>**，**IEnumerable<Object>**，**ICollection<Stream>**，**ICollection<Object>**，**IList<Stream>**和 **IList<Object>**。由于所有这些接口都由 CLR 自动

实现，所以在存在这些接口的任何地方都可以使用 **fsArray** 变量。例如，可将 **fsArray** 变量传给具有以下任何一种原型的方法：

```
void M1(IList<FileStream> fsList) { … }
void M2(ICollection<Stream> sCollection) { … }
void M3(IEnumerable<Object> oEnumerable) { … }
```

注意，如果数组包含值类型的元素，数组类型不会为元素的基类型实现接口。例如，如果执行以下代码：

```
DateTime[] dtArray; // 一个值类型的数组
```

那么 **DateTime[]** 类型只会实现 **IEnumerable<DateTime>**，**ICollection<DateTime>** 和 **IList<DateTime>**接口，不会为 **DateTime** 的基类型(包括 **System.ValueType** 和 **System.Object**)实现这些泛型接口。这意味着 **dtArray** 变量不能作为实参传给前面的 **M3** 方法。这是因为值类型的数组在内存中的布局与引用类型的数组不同。数组内存的布局请参见本章前面的描述。

16.5　数组的传递和返回

数组作为实参传给方法时，实际传递的是对该数组的引用。因此，被调用的方法能修改数组中的元素。如果不想被修改，必须生成数组的拷贝并将拷贝传给方法。注意，**Array.Copy** 方法执行的是浅拷贝。换言之，如果数组元素是引用类型，新数组将引用现有的对象。

类似地，有的方法会返回对数组的引用。如果方法构造并初始化数组，返回数组引用是没有问题的。但假如方法返回的是对字段所维护的一个内部数组的引用，就必须决定是否想让该方法的调用者直接访问这个数组及其元素。如果是，就可以返回数组引用。但更常见的情况是，你并不希望方法的调用者获得这个访问权限。所以，方法应该构造一个新数组，并调用 **Array.Copy** 返回对新数组的引用。再次提醒，**Array.Copy** 执行的是对原始数组的浅拷贝。

如果定义返回数组引用的方法，而且数组中不包含元素，那么方法既可以返回 **null**，也可以返回对包含零个元素的一个数组的引用。实现这种方法时，Microsoft 强烈建议让它返回后者，因为这样能简化调用该方法时需要写的代码[①]。例如，以下代码很容易理解。而且即使没有可供遍历的约会(即 **appointments** 数组中没有元素)，也能正确运行：

```
// 这段代码更容易写，更容易理解
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0; a < appointments.Length; a++) {
   // 对 appointments[a]执行操作
}
```

以下代码也能在没有约会的前提下正确运行，但写起来麻烦一些，而且不好理解：

```
// 这段代码写起来麻烦一些，而且不好理解
```

① 因为不需要执行 **null** 值检测。——译注

```
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
   for (Int32 a = 0, a < appointments.Length; a++) {
      // 对 appointments[a]执行操作
   }
}
```

将方法设计为返回对含有 0 个元素的一个数组的引用，而不是返回 **null**，该方法的调用者就能更轻松地使用该方法。顺便提一句，对字段也应如此。如果类型中有一个字段是数组引用，应考虑让这个字段始终引用数组，即使数组中不包含任何元素。

16.6 创建下限非零的数组

前面提到过，能创建和操作下限非 0 的数组。可以调用数组的静态 **CreatInstance** 方法来动态创建自己的数组。该方法有若干个重载版本，允许指定数组元素的类型、数组的维数、每一维的下限和每一维的元素数目。**CreateInstance** 为数组分配内存，将参数信息保存到数组的内存块的开销(overhead)部分，然后返回对该数组的引用。如果数组维数是 2 或 2 以上，就可以把 **CreateInstance** 返回的引用转型为一个 **ElementType[]**变量(**ElementType** 要替换为类型名称)，以简化对数组中的元素的访问。如果只有一维，C#要求必须使用该 **Array** 的 **GetValue** 和 **SetValue** 方法访问数组元素。

以下代码演示了如何动态创建由 **System.Decimal** 值构成的二维数组。第一维代表 2005 到 2009(含)年份，第二维代表 1 到 4(含)季度。代码遍历动态数组中的所有元素。我本来可以将数组的上下限硬编码到代码中，这样能获得更好的性能。但我最后决定使用 **System.Array** 的 **GetLowerBound** 和 **GetUpperBound** 方法来演示它们的用法：

```
using System;

public sealed class DynamicArrays {
   public static void Main() {
      // 我想创建一个二维数组 [2005..2009][1..4]
      Int32[] lowerBounds = { 2005, 1 };
      Int32[] lengths     = {    5, 4 };
      Decimal[,] quarterlyRevenue = (Decimal[,])
         Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);

      Console.WriteLine("{0,4} {1,9} {2,9} {3,9} {4,9}",
         "Year", "Q1", "Q2", "Q3", "Q4");
      Int32 firstYear    = quarterlyRevenue.GetLowerBound(0);
      Int32 lastYear     = quarterlyRevenue.GetUpperBound(0);
      Int32 firstQuarter = quarterlyRevenue.GetLowerBound(1);
      Int32 lastQuarter  = quarterlyRevenue.GetUpperBound(1);

      for (Int32 year = firstYear; year <= lastYear; year++) {
         Console.Write(year + " ");
         for (Int32 quarter = firstQuarter; quarter <= lastQuarter; quarter++) {
            Console.Write("{0,9:C} ", quarterlyRevenue[year, quarter]);
         }
         Console.WriteLine();
      }
   }
}
```

编译并运行这段代码将得到以下输出：

```
Year   Q1      Q2      Q3      Q4
2005   $0.00   $0.00   $0.00   $0.00
2006   $0.00   $0.00   $0.00   $0.00
2007   $0.00   $0.00   $0.00   $0.00
2008   $0.00   $0.00   $0.00   $0.00
2009   $0.00   $0.00   $0.00   $0.00
```

16.7　数组的内部工作原理

CLR 内部实际支持两种不同的数组。

- 下限为 0 的一维数组。这些数组有时称为 SZ(single-dimensional, zero-based，一维 0 基)数组或向量(vector)。
- 下限未知的一维或多维数组。

可执行以下代码来实际地查看不同种类的数组(注释显示了输出)

```
using System;

public sealed class Program {
   public static void Main() {
   Array a;

   // 创建一维的 0 基数组，不包含任何元素
   a = new String[0];
   Console.WriteLine(a.GetType()); // "System.String[]"

   // 创建一维 0 基数组，不包含任何元素
   a = Array.CreateInstance(typeof(String),
      new Int32[] { 0 }, new Int32[] { 0 });
   Console.WriteLine(a.GetType()); // "System.String[]"

   // 创建一维 1 基数组，其中不包含任何元素
   a = Array.CreateInstance(typeof(String),
      new Int32[] { 0 }, new Int32[] { 1 });
   Console.WriteLine(a.GetType()); // "System.String[*]" <-- 这个显示很奇怪，不是吗?

   Console.WriteLine();

   // 创建二维 0 基数组，其中不包含任何元素
   a = new String[0, 0];
   Console.WriteLine(a.GetType());  // "System.String[,]"

   // 创建二维 0 基数组，其中不包含任何元素
   a = Array.CreateInstance(typeof(String),
      new Int32[] { 0, 0 }, new Int32[] { 0, 0 });
   Console.WriteLine(a.GetType()); // "System.String[,]"

   // 创建二维的 1 基数组，其中不包含任何元素
   a = Array.CreateInstance(typeof(String),
      new Int32[] { 0, 0 }, new Int32[] { 1, 1 });
   Console.WriteLine(a.GetType()); // "System.String[,]"
   }
}
```

每个 **Console.WriteLine** 语句后都有一条指出其输出的注释。对于一维数组，0 基数组显示

的类型名称是 **System.String[]**，但 1 基数组显示的是 **System.String[*]**。*符号表明 CLR 知道该数组不是 0 基的。注意，C#不允许声明 **String[*]**类型的变量，因此不能使用 C#语法来访问一维非 0 基数组。尽管可以调用 **Array** 的 **GetValue** 和 **SetValue** 方法来访问这种数组的元素，但速度会比较慢，因为有方法调用的开销。

对于多维数组，0 基和 1 基数组会显示同样的类型名称：**System.String[,]**。在运行时，CLR 将所有多维数组都视为非 0 基数组。这自然会让人觉得类型名称应该显示为 **System.String[*,*]**。但是，对于多维数组，CLR 决定不使用*符号。这是由于假如使用*，那么它们始终都会存在，而大量的*会使开发人员产生混淆。

访问一维 0 基数组的元素比访问非 0 基一维或多维数组的元素稍快。这是多方面的原因造成的。首先，有一些特殊 IL 指令，比如 **newarr**，**ldelem**，**ldelema**，**ldlen** 和 **stelem**，用于处理一维 0 基数组，这些特殊 IL 指令会导致 JIT 编译器生成优化代码。例如，JIT 编译器生成的代码假定数组是 0 基的，所以在访问元素时不需要从指定索引中减去一个偏移量。其次，一般情况下，JIT 编译器能将索引范围检查代码从循环中拿出，导致它只执行一次。以下面这段常见的代码为例：

```
using System;

public static class Program {
   public static void Main() {
      Int32[] a = new Int32[5];
      for(Int32 index = 0; index < a.Length; index++) {
         // 对a[index]执行操作
      }
   }
}
```

对于以上代码，首先注意在 **for** 循环的测试表达式中对数组的 **Length** 属性的调用。由于 **Length** 是属性，所以查询长度实际是方法调用。但 JIT 编译器知道 **Length** 是 **Array** 类的属性，所以在生成的代码中，实际只调用该属性一次，结果存储到一个临时变量中。每次循环迭代检查的都是这个临时变量。这就加快了 JIT 编译的代码的速度。但有的开发人员低估了 JIT 编译器的“本事”，试图自己写一些“高明”的代码来“帮助”JIT 编译器。任何自作聪明的尝试几乎肯定会对性能造成负面影响，还会使代码更难阅读，妨碍可维护性。在上述代码中，最好保持对数组 **Length** 属性的调用，而不要自己用什么局部变量来缓存它的值。

其次要注意，JIT 编译器知道 **for** 循环要访问 **0** 到 **Length - 1** 的数组元素。所以，JIT 编译器会生成代码，在运行时测试所有数组元素的访问都在数组有效范围内。具体地说，JIT 编译器会生成代码来检查是否**(0 >= a.GetLowerBound(0)) && ((Length - 1) <= a.GetUpperBound(0))**。这个检查在循环之前发生。如果在数组有效范围内，JIT 编译器不会在循环内部生成代码验证每一次数组访问都在有效范围内。这样一来，循环内部的数组访问变得非常快。

遗憾的是，正如前文所述，访问“非 0 基一维数组”或“多维数组”的速度比不上一维 0 基数组。对于这些数组类型，JIT 编译器不会将索引检查从循环中拿出来，所以每次数组访

问都要验证指定的索引。此外，JIT 编译器还要添加代码从指定索引中减去数组下限，这进一步影响了代码执行速度，即使此时使用的多维数组碰巧是 0 基数组。所以，如果很关心性能，考虑用由数组构成的数组(即交错数组)代替矩形数组。

C#和 CLR 还允许使用 unsafe(不可验证)代码访问数组。这种技术实际能在访问数组时关闭索引上下限检查。这种不安全的数组访问技术适合以下元素类型的数组：**SByte**、**Byte**、**Int16**、**UInt16**、**Int32**、**UInt32**、**Int64**、**UInt64**、**Char**、**Single**、**Double**、**Decimal**、**Boolean**、枚举类型或者字段为上述任何类型的值类型结构。

这个功能很强大，但使用须谨慎，因为它允许直接内存访问。访问越界(超出数组上下限)不会抛出异常，但会损坏内存中的数据，破坏类型安全性，并可能造成安全漏洞！有鉴于此，包含 unsafe 代码的程序集必须被授予完全信任，或至少启用“跳过验证”安全权限。

以下 C#代码演示了访问二维数组的三种方式(安全、交错和不安全)：

```
using System;
using System.Diagnostics;

public static class Program {
   private const Int32 c_numElements = 10000;

   public static void Main() {
      // 声明二维数组
      Int32[,] a2Dim = new Int32[c_numElements, c_numElements];

      // 将二维数组声明为交错数组(向量构成的向量)
      Int32[][] aJagged = new Int32[c_numElements][];
      for (Int32 x = 0; x < c_numElements; x++)
         aJagged[x] = new Int32[c_numElements];

      //  1: 用普通的安全技术访问数组中的所有元素
      Safe2DimArrayAccess(a2Dim);

      //  2: 用交错数组技术访问数组中的所有元素
      SafeJaggedArrayAccess(aJagged);

      //  3: 用 unsafe 技术访问数值中的所有元素
      Unsafe2DimArrayAccess(a2Dim);
   }

   private static Int32 Safe2DimArrayAccess(Int32[,] a) {
      Int32 sum = 0;
      for (Int32 x = 0; x < c_numElements; x++) {
         for (Int32 y = 0; y < c_numElements; y++) {
            sum += a[x, y];
         }
      }
      return sum;
   }

   private static Int32 SafeJaggedArrayAccess(Int32[][] a) {
      Int32 sum = 0;
      for (Int32 x = 0; x < c_numElements; x++) {
         for (Int32 y = 0; y < c_numElements; y++) {
            sum += a[x][y];
         }
      }
      return sum;
   }
```

```
   private static unsafe Int32 Unsafe2DimArrayAccess(Int32[,] a) {
      Int32 sum = 0;
      fixed (Int32* pi = a) {
         for (Int32 x = 0; x < c_numElements; x++) {
            Int32 baseOfDim = x * c_numElements;
            for (Int32 y = 0; y < c_numElements; y++) {
               sum += pi[baseOfDim + y];
            }
         }
      }
      return sum;
   }
}
```

Unsafe2DimArrayAccess 方法标记了 **unsafe** 修饰符，这是使用 C#的 **fixed** 语句所必须的。编译这段代码要求在运行 C#编译器时指定**/unsafe** 开关，或者在 Microsoft Visual Studio 的项目属性页的“生成”选项卡中勾选“允许不安全代码”。

写代码时，不安全数据访问技术有时或许是你的最佳选择，但要注意该技术的三处不足。

- 相较于其他技术，处理数组元素的代码更复杂，不容易读和写，因为要使用 C# **fixed** 语句，要执行内存地址计算。
- 计算过程中出错，可能访问到不属于数组的内存。这会造成计算错误，损坏内存数据，破坏类型安全性，并可能造成安全漏洞。
- 因为这些潜在的问题，CLR 禁止在降低了安全级别的环境(如 Microsoft Silverlight)中运行不安全代码。

16.8 不安全的数组访问和固定大小的数组

不安全的数组访问非常强大，因为它允许访问以下元素。

- 堆上的托管数组对象中的元素(上一节对此进行了演示)。
- 非托管堆上的数组中的元素。第 14 章的 SecureString 示例演示了如何调用 **System.Runtime.InteropServices.Marshal** 类的 **SecureStringToCoTaskMemUnicode** 方法来返回一个数组，并对这个数组进行不安全的数组访问。
- 线程栈上的数组中的元素。

如果性能是首要目标，请避免在堆上分配托管的数组对象。相反，在线程栈上分配数组。这是通过 C#的 **stackalloc** 语句来完成的(它在很大程度上类似于 C 的 **alloca** 函数)。**stackalloc** 语句只能创建一维 0 基、由值类型元素构成的数组，而且值类型绝对不能包含任何引用类型的字段。实际上，应该把它的作用看成是分配一个内存块，这个内存块可以使用不安全的指针来操纵。所以，不能将这个内存缓冲区的地址传给大部分 FCL 方法。当然，栈上分配的内存(数组)会在方法返回时自动释放；这对增强性能起了一定作用。使用这个功能要求为 C#编译器指定**/unsafe** 开关。

以下代码中的 **StackallocDemo** 方法演示如何使用 C# **stackalloc** 语句：

```
using System;

public static class Program {
   public static void Main() {
      StackallocDemo();
      InlineArrayDemo();
   }

   private static void StackallocDemo() {
      unsafe {
         const Int32 width = 20;
         Char* pc = stackalloc Char[width];  // 在栈上分配数组

         String s = "Jeffrey Richter";  // 15 个字符

         for (Int32 index = 0; index < width; index++) {
            pc[width - index - 1] =
               (index < s.Length) ? s[index] : '.';
         }

         // 下面这行代码显示".....rethciR yerffeJ"
         Console.WriteLine(new String(pc, 0, width));
      }
   }

   private static void InlineArrayDemo() {
      unsafe {
         CharArray ca;     // 在栈上分配数组
         Int32 widthInBytes = sizeof(CharArray);
         Int32 width = widthInBytes / 2;

         String s = "Jeffrey Richter";   // 15 个字符

         for (Int32 index = 0; index < width; index++) {
            ca.Characters[width - index - 1] =
                (index < s.Length) ? s[index] : '.';
         }

         // 下面这行代码显示".....rethciR yerffeJ"
         Console.WriteLine(new String(ca.Characters, 0, width));
      }
   }
}

internal unsafe struct CharArray {
   // 这个数组内联(嵌入)到结构中
   public fixed Char Characters[20];
}
```

通常，由于数组是引用类型，所以结构中定义的数组字段实际只是指向数组的指针或引用；数组本身在结构的内存的外部。不过，也可像上述代码中的 **CharArray** 结构那样，直接将数组嵌入结构。在结构中嵌入数组需满足以下几个条件。

- 类型必须是结构(值类型)；不能在类(引用类型)中嵌入数组。
- 字段或其定义结构必须用 **unsafe** 关键字标记。