.NET 中字符串比较的最佳做法

痕迹 CodeShare 1月19日

.NET 为开发本地化和全球化应用程序提供广泛支持，在执行排序和显示字符串等常见操作时，轻松应用当前区域性或特定区域性的约定。但排序或比较字符串并不总是区分区域性的操作。
例如，对于应用程序内部使用的字符串，通常应该跨所有区域性以相同的方式对其进行处理。如果将 XML 标记、HTML 标记、用户名、文件路径和系统对象名称等与区域性无关的字符串数据解释为区分区域性，则应用程序代码会遭遇细微的错误、不佳的性能，在某些情况下，还会遭遇安全性问题。

本文介绍 .NET 中的字符串排序、比较和大小写方法，针对如何选择适当的字符串处理方法提出建议，并提供有关字符串处理方法的其他信息。

对字符串用法的建议

使用 .NET 进行开发时，请遵循以下简要建议比较字符串：

- 使用为字符串操作显式指定字符串比较规则的重载。通常情况下，这涉及调用具有 StringComparison类型的参数的方法重载。
- 使用 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 进行比较，并以此作为匹配区域性不明确的字符串的安全默认设置。
- 将比较与 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 配合使用，以获得更好的性能。
- 向用户显示输出时，使用基于 StringComparison.CurrentCulture 的字符串操作。
- 当进行与语言（例如，符号）无关的比较时，使用非语言的 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 值，而不使用基于 CultureInfo.InvariantCulture 的字符串操作。
- 在规范化要比较的字符串时，使用 String.ToUpperInvariant 方法而非 String.ToLowerInvariant 方法。
- 使用 String.Equals 方法的重载来测试两个字符串是否相等。
- 使用 String.Compare 和 String.CompareTo 方法可对字符串进行排序，而不是检查字符串是否相等。
- 在用户界面，使用区分区域性的格式显示非字符串数据，如数字和日期。使用格式以固定区域性使非字符串数据显示为字符串形式。

比较字符串时，请避免采用以下做法：

- 不要使用未显式或隐式为字符串操作指定字符串比较规则的重载。

- 在大多数情况下，不要使用基于 StringComparison.InvariantCulture 的字符串操作。其中的一个少数例外情况是，保存在语言上有意义但区域性不明确的数据。
- 不要使用 String.Compare 或 CompareTo 方法的重载和用于确定两个字符串是否相等的返回值为 0 的测试。

显式指定字符串比较

重载 .NET 中大部分字符串操作方法。通常，一个或多个重载会接受默认设置，然而其他重载则不接受默认设置，而是定义比较或操作字符串的精确方式。大多数不依赖于默认设置的方法都包括 StringComparison类型的参数，该参数是按区域性和大小写为字符串比较显式指定规则的枚举。下表描述StringComparison 枚举成员。

StringComparison 成员	描述
CurrentCulture	使用当前区域性执行区分大小写的比较。
CurrentCultureIgnoreCase	使用当前区域性执行不区分大小写的比较。
InvariantCulture	使用固定区域性执行区分大小写的比较。
InvariantCultureIgnoreCase	使用固定区域性执行不区分大小写的比较。
Ordinal	执行序号比较。
OrdinalIgnoreCase	执行不区分大小写的序号比较。

例如， IndexOf 方法（它返回 String 对象中与某字符或字符串匹配的子字符串的索引）具有九种重载：

- 默认情况下，IndexOf(Char), IndexOf(Char, Int32)和 IndexOf(Char, Int32, Int32)对字符串中的字符执行序号（区分大小写但不区分区域性的）搜索。
- 默认情况下，IndexOf(String), IndexOf(String, Int32)和 IndexOf(String, Int32, Int32)对字符串中的子字符串执行区分大小写且区分区域性的搜索。
- IndexOf(String, StringComparison)、 IndexOf(String, Int32, StringComparison)和 IndexOf(String, Int32, Int32, StringComparison)，其中包括 StringComparison 类型的参数，该类型允许指定比较形式。

我们建议选择不使用默认值的重载，原因如下：

- 具有默认参数的一些重载（在字符串实例中搜索 Char 的重载）执行序号比较，而其他重载（在字符串实例中搜索字符串的重载）执行的是区分区域性的比较。要记住哪种方法使用哪个默认值并非易事，并很容易混淆重载。
- 依赖于方法调用默认值的代码的意图并不清楚。在下面依赖于默认值的示例中，很难了解开发人员对两个字符串的实际意图是执行序号比较还是语言比

较，或者 `protocol` 和“http”之间存在的大小写差异是否会导致相等性测试返回 `false` 类型的参数的方法重载。

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http")) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```


一般情况下，我们建议调用不依赖于默认设置的方法，因为这会明确代码的意图。这进而使代码更具可读性且更易于调试和维护。下面的示例解决了前面示例中提出的问题。使用序号比较并且忽略大小写差异。

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase)) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```


字符串比较的详细信息

字符串比较是许多字符串相关操作的核心，特别是排序和相等性测试操作。字符串以确定的顺序进行排序：如果在排序的字符串列表中，“my”出现在“string”之前，则“my”必定小于或等于“string”。此外，比较可隐式确定相等性。对于认为是相等的字符串，比较操作将返回零。对此很好的解释是两个字符串都不小于对方。涉及到字符串的最有意义的操作包括这些步骤中的一个或两个步骤：与另一个字符串进行比较和执行明确的排序操作。

> [!NOTE]
> 可以下载排序权重表，这是一组文本文件，其中包含有关 Windows 操作系统排序和比较操作中所使用的字符权重的信息，也可以下载默认 Unicode 排序元素表，这是适用于 Linux 和 macOS 的最新版排序权重表。 Linux 和 macOS 上的特定排序权重表版本取决于系统上安装的 International Components for Unicode 库的版本。 有关 ICU 版本及它们所实现的 Unicode 版本的信息，请参阅下载 ICU。

但是，评估两个字符串的相等性或排序顺序不会生成一个正确的结果；其结果取决于用于比较这两个字符串的条件。特别是，序号或基于当前区域性或固定区域性（基于英语语言的区域设置不明确的区域性）的大小写和排序约定的字符串比较可能会产生不同的结果。

此外，使用不同 .NET 版本或在不同操作系统或不同的操作系统版本上使用 .NET 进行字符串比较时，返回的结果可能不同。有关详细信息，请参阅字符串和 Unicode 标准。

使用当前区域性的字符串比较

一个条件涉及在比较字符串时使用当前区域性的约定。基于当前区域性的比较使用线程的当前区域性或区域设置。如果用户未设置该区域性，则默认为“控制面板”中“区域

选项” 窗口中的设置。当数据与语言相关并反映区分区域性的用户交互时，应始终使用基于当前区域性的比较。

但是，当区域性发生更改时，.NET 中的比较和大小写行为也发生更改。如果执行应用程序的计算机与用于开发该应用程序的计算机具有不同的区域性，或者执行线程改变它的区域性，则会发生这种情况。此行为是有意而为之的，但许多开发人员不易察觉此行为。下面的示例说明了美国英语（“en-US”）与瑞典语（“sv-SE”）区域性在排序顺序中的差异。请注意，单词“ångström”、“Windows”和“Visual Studio”将出现在已排序的字符串数组的不同位置。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] values= { "able", "ångström", "apple", "Æble",
                         "Windows", "Visual Studio" };
      Array.Sort(values);
      DisplayArray(values);

      // Change culture to Swedish (Sweden).
      string originalCulture = CultureInfo.CurrentCulture.Name;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
      Array.Sort(values);
      DisplayArray(values);

      // Restore the original culture.
      Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);
    }

    private static void DisplayArray(string[] values)
    {
      Console.WriteLine("Sorting using the {0} culture:",
                        CultureInfo.CurrentCulture.Name);
      foreach (string value in values)
         Console.WriteLine("   {0}", value);

      Console.WriteLine();
    }
}
```



```
// The example displays the following output:
//       Sorting using the en-US culture:
//          able
//          Æble
//          ångström
//          apple
//          Visual Studio
//          Windows
//
//       Sorting using the sv-SE culture:
//          able
//          Æble
//          apple
//          Windows
//          Visual Studio
//          ångström
```

使用当前区域性的不区分大小写比较和区分区域性的比较是相同的，只不过前者忽略由线程的当前区域性指示的大小写。这种情况也可表明它的排序顺序。

以下方法默认利用使用当前区域性语义的比较：

- 不包括String.Compare 参数的 StringComparison 重载。
- String.CompareTo 重载。
- 默认 String.StartsWith(String) 方法和具有 String.StartsWith(String, Boolean, CultureInfo) null nullCultureInfo 重载。
- 默认 String.EndsWith(String) 方法和需要使用 nullCultureInfo 参数的 String.EndsWith(String, Boolean, CultureInfo) 方法。
- 接受String.IndexOf 作为搜索参数且不包含 String 参数的 StringComparison 重载。
- 接受String.LastIndexOf 作为搜索参数且不包含 String 参数的 StringComparison 重载。

总之，我们建议调用具有 xref:System.StringComparison 参数的重载，以便明确方法调用的意图。

当从语言角度解释非语言的字符串数据，或利用其他区域性的约定解释某个特定区域性中的字符串时，则会发生或大或小的错误。土耳其语 I 问题便是一个规范示例。

对于几乎所有拉丁字母来讲（包括美国英语），字符“i”(\u0069) 是字符“I”(\u0049) 的小写形式。此大小写规则快速成为在此类区域性中编程的人员的默认设置。但是，土耳其语（“tr-TR”）字母表中包含一个“带有点的 I”的字符“İ”(\u0130)，该字符是“i”的大写形式。土耳其语还包括一个小写“不带点的 i”字符，即为“ı”(\u0131)，该字符的大写形式为“I”。阿塞拜疆语（“az”）区域也会出现这种情况。

因此，关于将“i”变为大写或将“I”变为小写的假设并非在所有区域性中都是有效的。如果为字符串比较例程使用默认重载，则它们可能会因区域性不同而异。如果对非语言的数据进行比较，使用默认重载会产生不良后果，如以下对字符串“file”和“FILE”执行不区分大小写的比较尝试所示。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string fileUrl = "file";
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                       fileUrl.StartsWith("FILE", true, null));
      Console.WriteLine();

      Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                        fileUrl.StartsWith("FILE", true, null));
   }
}
// The example displays the following output:
//       Culture = English (United States)
//       (file == FILE) = True
//
//       Culture = Turkish (Turkey)
//       (file == FILE) = False
```


如果无意中在安全敏感设置中使用了区域性，则此比较会导致发生重大问题，如以下示例所示。如果当前区域性为美国英语，则 `IsFileURI("file:")` 等方法调用将返回 `true` ；但如果当前区域性为土耳其语，则将返回 `false` 。因此，在土耳其语系统中，有人可能会避开阻止访问以“FILE:”开头的不区分大小写的安全措施。

```
public static bool IsFileURI(String path)
{
   return path.StartsWith("FILE:", true, null);
}
```

在这种情况下，由于“file:”会被解释为非语言的、不区分区域性的标识符，因此，应按照下面的示例所示编写代码：

```
public static bool IsFileURI(string path)
{
   return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
}
```


序号字符串操作

在方法调用中指定 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 值表示非语言比较，这种比较忽略了自然语言的特性。利用 StringComparison 值调用的方法将字符串操作决策建立在简单的字节比较的基础之上，而不是按区域性参数化的大小写或相等表。在大多数情况下，这种方法最符合字符串的预期解释，并使代码更快更可靠。

序号比较就是字符串比较，在这种比较中，将比较每个字符串中的每个字节且不进行语言解释；例如，“windows”不匹配“Windows”。实质上，这是对 C 运行时 strcmp 函数的调用。当上下文指示应完全匹配字符串或要求保守匹配策略时，请使用这种比较。此外，序号比较是最快的比较操作，因为它在确定结果时不应用任何语言规

则。
.NET 中的字符串可以包括嵌入的空字符。序号比较与区分区域性的比较（包括使用固定区域性的比较）之间最明显的区别之一是对字符串中嵌入的空字符的处理方式。当使用 String.Compare 和 String.Equals 方法执行区分区域性的比较（包括使用固定区域性的比较）时，将忽略这些字符。因此，在区分区域性的比较中，包含嵌入的空字符的字符串可视为等于不包含空字符的字符串。

> [!IMPORTANT]
> 尽管字符串比较方法忽略嵌入的空字符，但是 String.Contains、 String.EndsWith、 String.IndexOf、 String.LastIndexOf和 String.StartsWith 等字符串搜索方法并不会忽略这些字符。

下面的示例对字符串“Aa”与在“A”和“a”之间嵌入了多个空字符的相似字符串进行区分区域性的比较，并显示如何将这两个字符串视为相等的字符串：

```
using System;

public class Example
{
   public static void Main()
   {
      string str1 = "Aa";
      string str2 = "A" + new String('\u0000', 3) + "a";
      Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                        str1, ShowBytes(str1), str2, ShowBytes(str2));
      Console.WriteLine("   With String.Compare:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Compare(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Compare(str1, str2, StringComparison.InvariantCulture));

      Console.WriteLine("   With String.Equals:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Equals(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Equals(str1, str2, StringComparison.InvariantCulture));
   }

   private static string ShowBytes(string str)
   {
      string hexString = String.Empty;
      for (int ctr = 0; ctr < str.Length; ctr++)
      {
         string result = String.Empty;
         result = Convert.ToInt32(str[ctr]).ToString("X4");
         result = " " + result.Substring(0,2) + " " + result.Substring(2, 2);
         hexString += result;
      }
      return hexString.Trim();
   }
}
```



```
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Current Culture: 0
//          Invariant Culture: 0
//       With String.Equals:
//          Current Culture: True
//          Invariant Culture: True
```


但是，当使用序号比较时，这两个字符串不会视为相等，如下面的示例所示：

```
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                  str1, ShowBytes(str1), str2, ShowBytes(str2));
Console.WriteLine("   With String.Compare:");
Console.WriteLine("      Ordinal: {0}",
                  String.Compare(str1, str2, StringComparison.Ordinal));

Console.WriteLine("   With String.Equals:");
Console.WriteLine("      Ordinal: {0}",
                  String.Equals(str1, str2, StringComparison.Ordinal));
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 61):
//       With String.Compare:
//          Ordinal: 97
//       With String.Equals:
//          Ordinal: False
```


不区分大小写的序号比较是第二种最保守的方法。这些比较会忽略大多数的大小写；例如，“windows”会匹配“Windows”。在处理 ASCII 字符时，此策略等同于 StringComparison.Ordinal，只不过它会忽略常用的 ASCII 大小写。因此，[A, Z] (\u0041-\u005A) 中的任何字符都会匹配 [a,z] (\u0061-\007A) 中的相应字符。超出 ASCII 范围的大小写使用固定区域性的表。因此，下面的比较：

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

等效于（但会更快）这种比较：

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal);
```


这些比较仍非常快。

StringComparison.Ordinal 和 StringComparison.OrdinalIgnoreCase 均直接使用二进制值并最适合匹配。当不确定比较设置时，请使用这两个值中的其中一个。不过，由于它们执行逐字节比较，因此不会按照语言排序顺序（如英语词典）进行排序，而是按照二进制排序顺序。如果向用户显示结果，则在大多数上下文中结果都看上去不正常。
序号语义是不包括 String.Equals 参数（包括相等运算符）的 StringComparison 重载的默认项。总之，我们建议调用具有 StringComparison 参数的重载。

使用固定区域性的字符串操作

具有固定区域性的比较使用由静态 CompareInfo 属性返回的 CultureInfo.InvariantCulture 属性。此行为在所有系统中都相同；它会将其范围外的任何字符转换为其认为等效的固定字符。此策略对于在各个区域性中维护一组字符串行为很有用，但经常产生意外的结果。
具有固定区域性的不区分大小写的比较也使用由静态 CompareInfo 属性返回的静态 CultureInfo.InvariantCulture 属性以获取比较信息。所转换字符中的任何大小写差异都将被忽略。
使用 StringComparison.InvariantCulture 和 StringComparison.Ordinal 的比较对 ASCII 字符串产生相同的作用。但是， StringComparison.InvariantCulture 会做出

可能不适用于解释为一组字节的字符串的语言性决策。还可以使用 CultureInfo.InvariantCulture.CompareInfo 对象使 Compare 方法将一组特定的字符解释为等效字符。例如，下面的等效字符在固定区域性中是有效的：
InvariantCulture: a +° = å
如果 A 字符的小写拉丁字母“a”(\u0061) 旁边有上方组合圆圈字符“+ ° ”(\u030a)，A 字符就会被解释为，上方带有圆圈的小写拉丁字母“å”(\u00e5)。如下面的示例所示，此行为不同于序号比较。

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.InvariantCulture) == 0);

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.Ordinal) == 0);
// The example displays the following output:
//    Equal sort weight of a° and å using InvariantCulture: True
//    Equal sort weight of a° and å using Ordinal: False
```


当解释其中出现如“å”组合的文件名称、cookie 或其他内容时，序号比较仍会提供最透明和最合适的行为。
总的来说，固定区域性具有极少的对比较有用的属性。它会以与语言相关的方式执行比较，使其无法保证完整的符号等效性，但它并不是任何区域性中显示的选择。使用 StringComparison.InvariantCulture 进行比较的其中一个原因是为多个区域性相同的显示保留已排序的数据。例如，如果应用程序附带包含用于显示的已排序标识符列表的大型数据文件，则添加到此列表将需要使用固定条件样式排序插入。

为方法调用选择 StringComparison 成员

下表概述了从语义字符串上下文到 StringComparison 枚举成员的映射：

数据	行为	相应 System.StringComparison value
区分大小写的内部标识符。 区分大小写的标准标识符（例如 XML 和 HTTP）。 区分大小写的安全相关设置。	字节完全匹配的非语言标识符。	Ordinal
不区分大小写的内部标识符。 不区分大小写的标准标识符（例如 XML 和 HTTP）。 文件路径。 注册表项和值。 环境变量。 资源标识符（例如，句柄名称）。 不区分大小写的安全相关设置。	无关大小写的非语言标识符；尤其是存储在大多数 Windows 系统服务中的数据。	OrdinalIgnoreCase
某些保留的、与语言相关的数据。 需要固定排序顺序的语言数据的显示。	仍与语言相关的区域性不明确数据。	InvariantCulture - 或 - InvariantCultureIgnoreCase
向用户显示的数据。 大多数用户输入。	需要本地语言自定义的数据。	CurrentCulture - 或 - CurrentCultureIgnoreCase

.NET 中的常见字符串比较方法

以下各节介绍最常用于执行字符串比较的方法。

String.Compare

默认解释：StringComparison.CurrentCulture。
作为字符串解释最核心的操作，应根据当前区域性检查这些方法调用的所有实例来确定是否应该从区域性（符号）解释或分离字符串。通常情况下，采用后者，并且应改用 StringComparison.Ordinal 比较。
System.Globalization.CompareInfo 属性返回的 CultureInfo.CompareInfo 类也包括利用 Compare 标记枚举的方式提供大量匹配选项（序号、忽略空白、忽略假名类型等）的 CompareOptions 方法。

String.CompareTo

默认解释：StringComparison.CurrentCulture。
此方法当前不提供指定 StringComparison 类型的重载。通常可以将此方法转换为建议

的 String.Compare(String, String, StringComparison) 形式。
实现 IComparable 和 IComparable 接口的类型实现此方法。由于它不提供 StringComparison 参数选项，因此实现类型经常使用户在其构造函数中指定 StringComparer。下面的示例定义 FileName 类，其类构造函数包括 StringComparer 参数。然后此 StringComparer 对象将用于 FileName.CompareTo 方法。

```
using System;

public class FileName : IComparable
{
   string fname;
   StringComparer comparer;

   public FileName(string name, StringComparer comparer)
   {
      if (String.IsNullOrEmpty(name))
         throw new ArgumentNullException("name");

      this.fname = name;

      if (comparer != null)
         this.comparer = comparer;
      else
         this.comparer = StringComparer.OrdinalIgnoreCase;
   }

   public string Name
   {
      get { return fname; }
   }

   public int CompareTo(object obj)
   {
      if (obj == null) return 1;

      if (! (obj is FileName))
         return comparer.Compare(this.fname, obj.ToString());
      else
         return comparer.Compare(this.fname, ((FileName) obj).Name);
   }
}
```


String.Equals

默认解释：StringComparison.Ordinal。
String 类可通过调用静态或实例 Equals 方法重载或使用静态相等运算符，测试是否相等。默认情况下，重载和运算符使用序号比较。但是，我们仍然建议调用显式指定 StringComparison 类型的重载，即使想要执行序号比较；这将更轻松地搜索特定字符串解释的代码。

String.ToUpper 和 String.ToLower

默认解释：StringComparison.CurrentCulture。
应谨慎使用这些方法，因为将字符串强制为大写或小写经常用作在不考虑大小写的情况下比较字符串的较小规范化。如果是这样，请考虑使用不区分大小写的比较。
还可以使用 String.ToUpperInvariant 和 String.ToLowerInvariant 方法。

ToUpperInvariant 是规范化大小写的标准方式。使用 StringComparison.OrdinalIgnoreCase 进行的比较在行为上是两个调用的组合：对两个字符串参数调用 ToUpperInvariant ，并使用 StringComparison.Ordinal执行比较。
通过向方法传递表示区域性的 CultureInfo 对象，重载也已可用于转换该特性区域性中的大写和小写字母。

Char.ToUpper 和 Char.ToLower

默认解释：StringComparison.CurrentCulture。
这些方法的工作原理类似于上一节中所述的 String.ToUpper 和 String.ToLower 方法。

String.StartsWith 和 String.EndsWith

默认解释：StringComparison.CurrentCulture。
默认情况下，这两种方法执行区分区域性的比较。

String.IndexOf 和 String.LastIndexOf

默认解释：StringComparison.CurrentCulture。
这些方法的默认重载如何执行比较方面缺乏一致性。包含 String.IndexOf 参数的所有 String.LastIndexOf 和 Char 方法都执行序号比较，但是包含 String.IndexOf 参数的默认 String.LastIndexOf 和 String 方法都执行区分区域性的比较。
如果调用 String.IndexOf(String) 或 String.LastIndexOf(String) 方法并向其传递一个字符串以在当前实例中查找，那么我们建议调用显式指定 StringComparison 类型的重载。包括 Char 参数的重载不允许指定 StringComparison 类型。

间接执行字符串比较的方法

将字符串比较作为核心操作的一些非字符串方法使用 StringComparer 类型。
StringComparer 类型包含六个返回 StringComparer 实例的静态属性，这些实例的 StringComparer.Compare 方法可执行以下类型的字符串比较：

- 使用当前区域性的区分区域性的字符串比较。此 StringComparer 对象由 StringComparer.CurrentCulture 属性返回。
- 使用当前区域性的不区分区域性的比较。此 StringComparer 对象由 StringComparer.CurrentCultureIgnoreCase 属性返回。
- 使用固定区域性的单词比较规则的不区分区域性的比较。此 StringComparer 对象由 StringComparer.InvariantCulture 属性返回。
- 使用固定区域性的单词比较规则的不区分大小写和不区分区域性的比较。此 StringComparer 对象由 StringComparer.InvariantCultureIgnoreCase 属性返回。
- 序号比较。此 StringComparer 对象由 StringComparer.Ordinal 属性返回。

- 不区分大小写的序号比较。此 StringComparer 对象由 StringComparer.OrdinalIgnoreCase 属性返回。

Array.Sort 和 Array.BinarySearch

默认解释：StringComparison.CurrentCulture。
当在集合中存储任何数据，或将持久数据从文件或数据库中读取到集合中时，切换当前区域性可能会使集合中的固定条件无效。Array.BinarySearch 方法假定已对数组中要搜索的元素排序。若要对数组中的任何字符串元素进行排序， Array.Sort 方法会调用 String.Compare 方法以对各个元素进行排序。如果对数组进行排序和搜索其内容的时间范围内区域性发生变化，那么使用区分区域性的比较器会很危险。例如在下面的代码中，是在由 Thread.CurrentThread.CurrentCulture 属性。如果在调用 StoreNames 和 DoesNameExist之间更改了区域性（尤其是数组内容保存在两个方法调用之间的某个位置），那么二进制搜索可能会失败。

```
// Incorrect.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names); // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name) >= 0);  // Line B.
}
```


建议的变体将显示在下面使用相同序号（不区分区域性）比较方法进行排序并搜索数组的示例中。在这两个示例中，更改代码会反映在标记 `Line A` 和 `Line B` 的代码行中。

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.Ordinal);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.Ordinal) >= 0);
}
```


如果此数据永久保留并跨区域性移动，并且使用排序来向用户显示此数据，则可以考虑使用 StringComparison.InvariantCulture，其语言操作可获得更好的用户输出且不受区域性更改的影响。下面的示例修改了前面两个示例，使用固定区域性对数组进行排序和搜索。

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.InvariantCulture);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.InvariantCulture) >
}
```


集合示例：哈希表构造函数

哈希字符串提供了第二个运算示例，该运算受比较字符串的方式影响。
下面的示例实例化 Hashtable 对象，方法是向其传递由 StringComparer 属性返回的 StringComparer.OrdinalIgnoreCase 对象。由于派生自 StringComparer 的类 StringComparer 实现 IEqualityComparer 接口，其 GetHashCode 方法用于计算哈希表中的字符串的哈希代码。

```
const int initialTableCapacity = 100;
Hashtable h;

public void PopulateFileTable(string directory)
{
   h = new Hashtable(initialTableCapacity,
                     StringComparer.OrdinalIgnoreCase);

   foreach (string file in Directory.GetFiles(directory))
         h.Add(file, File.GetCreationTime(file));
}

public void PrintCreationTime(string targetFile)
{
   Object dt = h[targetFile];
   if (dt != null)
   {
      Console.WriteLine("File {0} was created at time {1}.",
         targetFile,
         (DateTime) dt);
   }
   else
   {
      Console.WriteLine("File {0} does not exist.", targetFile);
   }
}
```