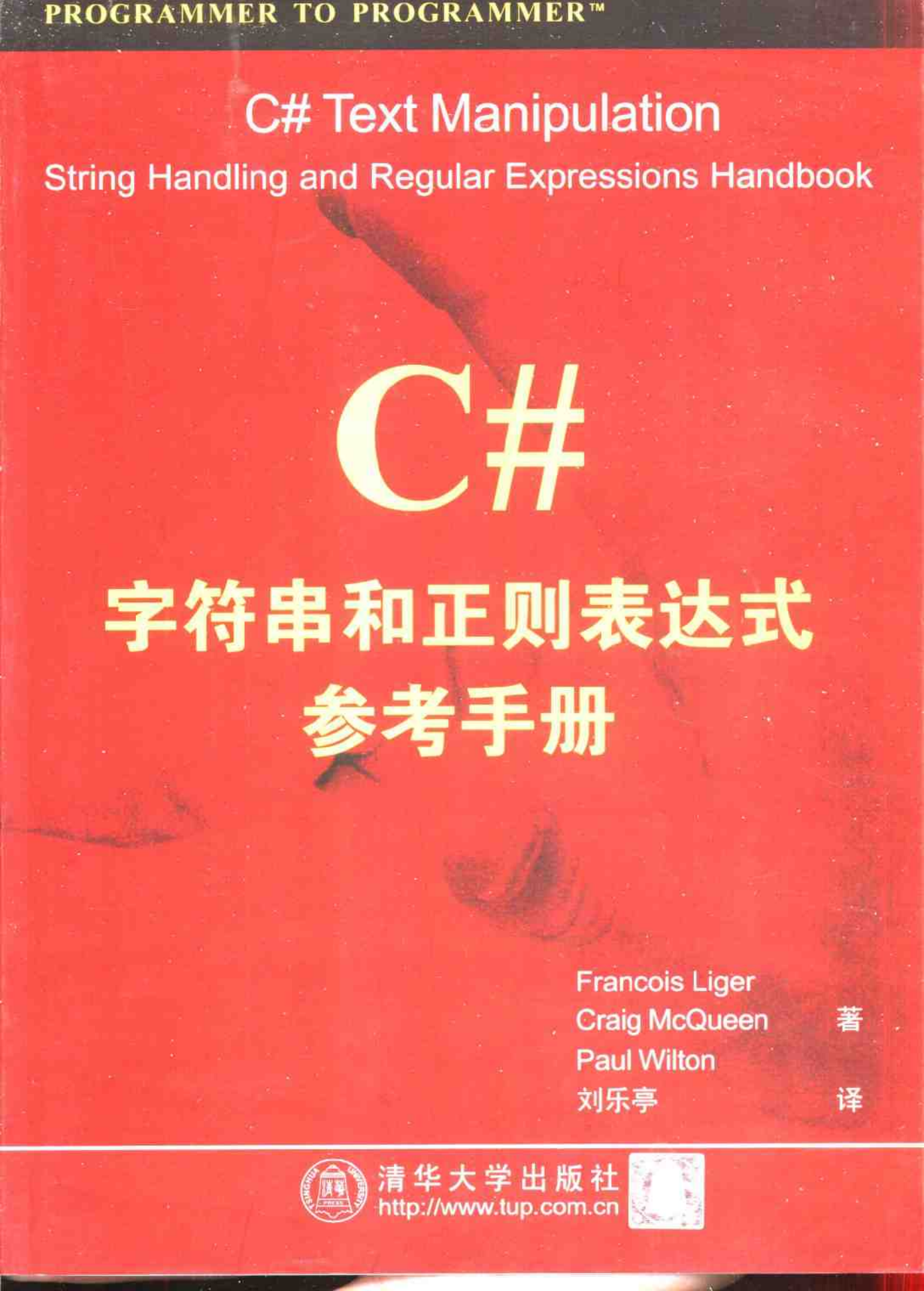
PROGRAMMER TO PROGRAMMER™

C# Text Manipulation

String Handling and Regular Expressions Handbook



C#

字符串和正则表达式参考手册

Francois Liger
Craig McQueen 著
Paul Wilton
刘乐亭 译

清华大学出版社
http://www.tup.com.cn

PROGRAMMER TO PROGRAMMER™

C# Text Manipulation

String Handling and Regular Expressions Handbook

文本操作是许多应用程序不可分割的一部分。.NET Framework 为处理文本提供了强大的功能。例如，.NET Framework 以 Unicode 形式存储字符串和字符，从而简便了应用程序的国际化。更为重要的是，由于 .NET Framework 的面向对象的特性，字符串也被视为对象，因此在应用程序中使用字符串时，一定要注意其用法。

.NET Framework 在 System.Text 命名空间中提供了许多强大的工具。String 类和 StringBuilder 类非常易于使用，其简单性是以使您在创建应用程序时忽略性能和效率的影响。此外，.NET Framework 还引入了正则表达式的文本匹配和操作的强大功能，这些功能能够让您使用简洁的语言分析、搜索和修改文本，从而进一步提高文本操作的效率。

在已由清华大学出版社出版的《VB.NET 字符串和正则表达式参考手册》一书中，主要阐述介绍了在 VB.NET 中操作文本的最佳方法。应广大读者的要求，我们又为 C# 开发人员翻译了这本设计和开发版的参考手册。

本书主要内容

- .NET Framework 中的字符串表示和管理
- 使用 StringBuilder 来提高应用程序的性能
- 运用不同的对象方法操作文本
- 如何安全地实现字符串和其他数据类型的转换
- 在 .NET Framework 中，如何使用 Unicode 表示文本，以实现国际化
- 如何使用正则表达式，包括优化文本操作的语法和模式匹配

Recommended Computer Book Categories	Programming
	C# .NET

ISBN 7-302-06327-3

9 787302 063278

定价：32.00 元

wrox

C#字符串和正则表达式参考手册

Francois Liger
Craig McQueen 著
Paul Wilton

刘乐亭 译

清华大学出版社

北京市版权局著作权合同登记号：01-2002-3187

内 容 简 介

文本操作几乎存在于任何应用程序中，合理地处理文本可以提高应用程序的性能。

本书详细阐述了.NET Framework 处理文本的方式，学习如何使用 String 类和 StringBuilder 类在.NET 中构建字符串，讲述在字符串和其他数据类型之间转换时所涉及的一些问题，并论述了如何用不同语言显示文本。本书还重点介绍了如何使用正则表达式匹配文本模式，描述了分组、替换和反向引用，并讨论了如何构建自己的正则表达式模式，以匹配具体的数据类型。最后，附录列出了 String 类和 StringBuilder 类的方法、属性和构造函数，以及许多正则表达式语法、选项和特殊字符。

本书适合于从事.NET 开发，并想在应用程序中提高文本处理效率的 C#开发人员。

图书在版编目(CIP)数据

C#字符串和正则表达式参考手册/(法)林格，(美)迈克昆廷，(英)威尔顿著；刘乐亭译.

—北京：清华大学出版社，2003

书名原文：C# Text Manipulation Handbook

ISBN 7-302-06327-3

I. C... II. ①林... ②迈... ③威... ④刘... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2003)第 009046 号

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)

http://www.tup.com.cn

责任编辑：李阳

印 刷 者：北京昌平环球印刷厂

发 行 者：新华书店总店北京发行所

开　　本：787×1092 1/16　**印张**：16　**字数**：332 千字

版　　次：2003 年 2 月第 1 版　2003 年 2 月第 1 次印刷

书　　号：ISBN 7-302-06327-3/TP・4775

印　　数：0001～4000

定　　价：32.00 元

前　言

本书主要讲述.NET Framework 处理文本，存储文本的不同方式，修改字符串的结果是什么，为什么需要一个单独的 StringBuilder 类，.NET Framework 把所有文本都存储为 Unicode 的意义是什么。本书也讲解操作文本的实现过程，同时解释各种可用的方法，比较它们的性能，以便您在选择操作文本的方式时做出更好的选择。在 C#中处理文本不像在 C++中处理文本那样复杂，但通过面向对象的架构，您可以控制处理 String 对象的方式，重要的是要理解.NET Framework 处理文本的方式。

处理字符串的一个最重要的类是 System.Text.RegularExpressions. Regex。这个类使所有的.NET 开发人员能够使用正则表达式提供的强大的模式匹配语言。“匹配”的意思是检查一个字符串是否遵守一定的模式，是否包含我们预期的数据。通常这要比给一个方法传递字符串，然后捕获一个异常(例如用无效数据更新数据库时发生的异常)更有效一些。本书的第 5～7 章将指导您学习正则表达式语言，解释每个组成部分的含义，以及如何构建复杂的表达式，以匹配文本中更复杂的模式。

没有正则表达式，即使像检查是否正确输入了电子邮件地址这样简单的任务也需要编写很多行代码。如果假设电子邮件地址已经存放在 myInput 字符串变量中，那么可以用下面这行代码生成一个布尔值，指明是否进行了成功的匹配：

```
bool IsMatch = Regex.IsMatch(myInput,
  a"^[a-z](\w|\d|[-.])+@([a-z\d-]+\.)+[a-z]{2,4}$",
  RegexOptions.IgnoreCase);
```

这看起来非常深奥，但您读完本书后，会发现它非常简单。为了简便起见，这个表达式没有对 myInput 变量做足够的限制(第 7 章会这样做)，但它指定了下面的模式：地址必须以字母开始，后面跟一个或多个字母、下划线、数字、连字符或句点，也可以是它们的组合，然后是“@”，它后面是字母的一个或多个模式，再后面是一个或多个字母、数字或连字符的重复模式(后跟一个句点)，并以一个 2～4 个字母的模式结束。例如，support@wrox.com 就是一个电子邮件地址，而 support@wrox 不是。

如果您有兴趣了解如何在 C#代码中使用该表达式语言，本书的作者会带领您学习这种模式匹配语言，因为他们对该语言都有丰富的经验。

本书读者对象

在 Wrox 参考手册系列丛书中，这是第一本面向 C#开发人员的书。您可以查阅 Wrox 网站，其中介绍了 C#参考手册系列中的其他书籍。

任何在应用程序中使用文本的开发人员都可以从本书中获益。本书代码的运行环境为 Microsoft C# .NET Standard Edition 或更高版本，但书中使用的代码和概念只适用于并只运行于.NET Framework SDK。

本书内容提要

本书讲解处理文本的各种方法。下面是各章内容的简介。

第 1 章　系统处理文本的方式

本章解释.NET Framework 处理文本的方式。讲述了各种较基础的内容，如字符串如何在内存中存储。在解释了文本的存储方式后，您就会明白为什么考虑处理文本的方法是很重要的。本章会介绍一些有关 MSIL(微软中间语言)的知识，但基本上是关于文本的概述，说明.NET 中内存的处理和管理(因此还有字符串占用内存的持续时间)。

第 2 章　String 类和 StringBuilder 类

本章将详述 String 类，解释.NET 中如何实现字符串。还将讲解 StringBuilder 类的工作原理，并比较这两个类的不同方法，了解执行某种操作的最快捷的方法是什么。学习 Microsoft 提供的共享源代码的 CLI(公共语言基础结构)实现方式，以期了解不同方法快慢的原因，以及在什么环境下使用看起来慢一些的方法更为合适。

第 3 章　字符串转换

本章讲述字符串和其他数据类型之间的转换所涉及的问题。讨论在不同的区域如何以不同的方式表示不同的数据类型，如数字、日期/时间，以及如何处理这些情况。我们还会探讨在数组和集合中存储字符串的问题，以及如何更好地操作以这种方式存储的字符串。

第 4 章　国际化

.NET Framework 中所有的文本都以 16 位的 Unicode 编码存放。本章对此进行了说明，并告诉您现在的字符至少以 16 位存储，此时该采取什么措施。我们将详细地讲述区域和不同的区域表示文本的方法，还将讲解如何操作 char 类型，因为在 Unicode 中，一个完整的字符可以由两个 16 位的 char 类型组成。本章会讨论如何以不同的语言来表

示文本，还将描述如何使用资源文件把应用程序中的文本保存在一个位置，以便把它们一次转换为另一种语言。

第 5 章　正则表达式

本章介绍正则表达式匹配语言，如何使用正则表达式匹配文本模式。首先学习如何完成使用 String 类的方法也能完成的工作，在熟悉了语法后，接着讲述该语言的一些独特功能。然后创建一个正则表达式测试器应用程序，它是测试表达式的一个非常有用的工具。

第 6 章　正则表达式的高级概念

上一章概述了匹配语言，而本章除了列举更多的表达式示例外，还将描述分组、替换和反向引用。分组允许表达式匹配成组的文本。这些组可以被捕获并赋予名称。反向引用可以用来提取前面的匹配，再为一个 HTML 结束标记匹配对应的开始标记。

第 7 章　正则表达式模式

本章详细介绍各种正则表达式模式，使用它们可以匹配具体的数据类型，如数字、日期等。本章将讲解这些模式的构成方式，帮助您构建自己的模式。其中一个模式是电子邮件地址验证程序，这是一个又长又复杂的表达式，非常有用。经过讲解，您就会明白构建匹配许多文本模式的表达式是何等的简单。

附录

本书有 5 个附录。前两个附录用表格列出了 String 类和 StringBuilder 类的方法、属性和构造函数。第 3 个附录包含了很多正则表达式语法、选项和特殊的字符。第 4 个附录是 C Sharp Today Web 站点(http://csharptoday.com)上的一篇文章。最后一个附录是有关本书的支持信息和下载代码的信息。

目　　录

第1章　系统处理文本的方式

文本是计算机应用程序中使用得最为频繁的数据类型。有效地操作和使用文本对于构建强大和可维护的应用程序至关重要。强大和可维护的应用程序一般运行顺利、节省内存，而且如果需要，还易于转换成不同的形式。但是如果使用不当，文本处理也可导致内存泄漏、性能下降、内存异常、难于将应用程序移植到其他语言中。

为了理解如何在 MS Windows 和.NET Framework 中使用文本，本章讲解以下问题：

- .NET Framework
- 作为数据类型的文本
- 字符和字符串
- 操作系统如何存储文本
- 典型的文本操作
- 常见的文本用法
- 国际化和本地化

本章介绍如何在典型的计算机操作系统中使用文本，具体讲解如何在.NET Framework 和 C#中使用文本。

本章给出了一些示例，介绍了在.NET Framework 之前执行文本处理任务的环境。学完本章后，您就可以理解计算机应用程序在处理文本时所面临的基本问题。本章还介绍了.NET 如何处理字符串的存储问题。本书的其他章节具体讲解了使用 C#处理文本的方法。

1.1　.NET Framework

为了理解系统如何处理文本，首先需要了解系统的各个特殊部分。.NET Framework 是 Win32 体系结构的重要改变，因此.NET Framework 有许多新概念。

.NET Framework 包括一个运行环境(也就是运行时)和一个类库。运行环境负责提供.NET 应用程序所需要的许多核心服务，其中包括内存分配和清理服务。因为字符串需要大量的内存管理工作，所以运行环境的实现将对使用字符串操作的应用程序的性能产生非常重要的影响。

1.1.1 公共语言运行时

运行环境通常称为公共语言基础结构(Common Language Infrastructure，CLI)。CLI是Microsoft提交给ECMA(欧洲计算机制造商协会)的一项标准，而ECMA是负责信息和通信系统标准化的机构。CLI的标准引用号是ECMA-335。

CLI定义了可执行代码和运行环境的规范。运行环境也称为虚拟运行系统(Virtual Execution System，VES)。Microsoft所实现的CLI是公共语言运行时(Common Language Runtime，CLR)。当讨论Microsoft专用的运行环境时，就使用CLR；当讨论CLR所遵守的标准时，就使用CLI。在本书中，大多数的讨论中使用了CLR，因为CLR是我们所使用的Microsoft实现的运行环境。

CLI的核心组成部分就是通用类型系统(Common Type System，CTS)。CTS定义了许多编程语言所支持的类型和操作。本书后面会详细讨论CTS。

与CTS相关的是公共语言规范(Common Language Specification，CLS)，CLS是提高语言互操作性的一组规则。.NET Framework的主要目标就是允许开发人员以任何喜爱的语言从事开发，并在相同的VES中运行这些程序。每一种语言都可以与其他语言交互。CLS允许工具实现人员编写出可以顺利过渡到其他语言的代码。

CLR被认为是在.NET中编写的应用程序的“管理器”。CLR确保应用程序符合安全规则，并向应用程序提供资源。在.NET中编写的应用程序称为托管代码(managed code)，这是由CLR管理代码的运行方式所决定的。

托管代码通过公共中间语言(Common Intermediate Language，CIL)和文件格式进行存储和传输。所有源代码语言都要编译成CIL指令集。因此，如果使用C#、J#、Visual Basic .NET或其他.NET语言编写，编译器都会将源代码转换为CIL指令。Microsoft实现的CIL称为MSIL。

托管数据就是由CLR自动分配和释放的数据，这些数据存储在托管堆(managed heap)中。通过垃圾收集机制可以自动释放数据。

托管堆

CLR提供了自动内存管理功能，它可以处理内存的分配和释放。对于开发人员来说这是非常便利的，因为在应用程序中，最常见、也最难跟踪的错误就是内存泄漏。当开发人员分配内存，但是忘记释放它时，就会发生内存泄漏。有时，在未确定何人负责释放内存时，也会出现内存泄漏。例如，如果给一个组件分配了一些内存，并把内存返回给调用者，那么由谁来负责内存的释放，内存又应该在何时释放呢？

因为必须给字符串分配内存，所以内存管理必然涉及到字符串的使用。

当第一次启动应用程序时，CLR就为应用程序预留了一块内存。此内存块就是托管堆。所有的引用类型都存储在堆中。稍后会讲到，String是引用类型，因此它存储在

堆中，并和堆中其他分配的内存块一样被处理。

图 1-1 演示了托管堆。

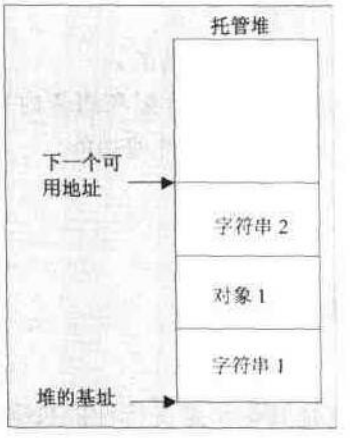


图　1-1

如图 1-1 所示，堆包含带有内存基址的一块内存区。内存区是连续的，存储区按线性方式分配。因为已经分配了堆内存，所以垃圾收集器在给系统返回分配的内存时，只能根据所要求的内存大小分配下一个可用的地址。这使内存的分配非常快捷。

内存分配非常简单，但是释放内存却要复杂得多。事实上，术语“垃圾收集”指释放不再使用的内存。计算机专家已经研究了执行垃圾收集的各种算法。.NET 文档简述了释放内存时垃圾收集器需要采取的步骤。在此不再重复。

1.1.2　.NET Framework 类库

.NET Framework 类库是支持.NET Framework 编程的类型的集合。类型包括类、结构和基类型。例如，System.Text.StringBuilder 类和 Char 结构都是类型。类库包含的类范围广泛，如用于 XML 处理、文件 I/O、绘图、线程，当然还有字符串操作的类。

在 System 命名空间中，有两个重要的类：

- Char——此结构是 Unicode 字符，具有转换字符和对字符分类(例如空白)的方法。
- String——此类是不变的字符串，具有操作字符串的方法。附录 A 给出了 String 类的构造函数、属性和方法的完整列表。

System.Text 命名空间包含重要的 StringBuilder 类，可以用于递增地构造字符串(附录 B 列出了 StringBuilder 类的所有构造函数、属性和方法)。该命名空间也包含一些可以将一种字符编码转换为另一种字符编码的类，比如 Encoding、ASCIIEncoding 和 UnicodeEncoding。

System.Text.RegularExpressions 命名空间包含各种类，用于构造和执行正则表达式，例如 Capture、CaptureCollection、Regex 和 RegexCompilationInfo。

简而言之，下面就是本章要讨论的要点：

- .NET Framework 类提供的文本操作功能。
- CLR 负责提供应用程序服务，包括分配和清理内存以支持字符串。
- 在 CLR 中实现作为垃圾收集的内存管理功能。

接下来详细介绍数据类型。

1.2 文本是一种数据类型

在编程语言中，数据类型就是具有预定义特征的数据。数据类型定义的规则，规定了数据应该具有什么样的值。大多数编程语言都有内置的数据类型，例如整型、浮点型和字符串型。

强类型化的编程语言，比如.NET 语言，实施了这种规则。如果试图给一种数据类型赋予与之不对应的值，就会收到错误消息。通常这些规则由编译器进行检查，这样数据类型错误就可以在早期捕获，不致于使之进入运行时代码(run-time code)。

除了规定数据的值之外，对数据类型能执行的操作也有限制。例如，对字符串数据类型执行除法操作就没有意义。

通常文本数据类型有最少的限定规则。因为有最少的规则控制文本可以携带的值，所以当程序员无法确认数据源中值的范围(比如接收用户的输入)时，就要使用文本数据类型。但是，由于文本数据无法有效存储，所以使用文本数据类型也会导致问题。

在某些编程语言中，文本数据类型非常难于处理。Visual C++就是一个例子。表 1-1 给出了当使用 Visual C++存储文本时可以选用的一些数据类型。

表 1-1 在 VC++中存储文本

选　项	示　例
字符数组	char myString[32];
宽字符数组	w_char myString[32];
标准 C++库	std::string myString;
MFC CString	CStr myString;
OLE Automation 字符串	BSTR myString;
BSTR 的类包装器	CComBSTR myString;

除了上表所列举的之外，还有在不同字符串类型之间进行转换和操作的函数。事实

上，表示字符串并不需要这么多方法，在此列举的众多方法只是历史上不断累积的产物。最初，字符存储在一个字节中。随着应用程序越来越国际化，在某些语言中所有的字符都要求使用两个字节来表示。

您可能注意到了，我们开始使用术语“字符串”表示文本。字符串是全称，表示字符的一个序列(或一串)。本章余下的内容使用字符串来代替文本。

因为.NET 是一种新的软件平台，所以重新定义了 String 数据类型，避免了用其他编程语言进行开发的混乱。

1.2.1　C#的数据类型

C#和其他.NET 语言中的所有数据类型都构建在 CTS 的基础之上。.NET Framework 的目标之一就是使程序员可以随意用各种与.NET 兼容的语言进行开发，而这些程序却在相同的基础平台上执行。CTS 定义了.NET 语言必须实现的数据类型。

1. 值类型和引用类型

.NET 中两种基本的数据类型是值类型和引用类型。

值类型包括整型、枚举、结构和字符。值类型直接存储在栈(stack)上，栈是操作系统分配的一个连续的内存区域，用于快速访问数据。CLR 对每一个正在运行的进程使用虚拟栈。因为值类型的容量是已知的，所以它可以存储在栈上。值类型可以直接访问，很少通过引用访问。当复制值类型时，值就被复制到另一个存储位置。

引用类型包括类、接口、数组和字符串。引用类型存储在托管堆中，托管堆是用于动态内存分配的内存区域，其中可以按任意次序分配和释放内存块。引用类型的容量直到运行时才能知道。这就是使用堆存储引用类型的原因。

可以通过对托管堆的引用(类似于指针)访问引用类型。这允许垃圾收集器跟踪引用，当引用不存在时，就释放存储区。当复制引用变量时，实际上是复制引用(也就是内存地址)，而不是值本身。

2. 基本数据类型

.NET Framework 类库是与 CLR 紧密集成的类型的集合。类库是面向对象的，这表示所有的类型都有属性和方法。一些类型使用得非常普遍，因此为描述这些类型提供了快捷方式。这些类型称为基本数据类型。

基本数据类型通过关键字识别，这些关键字就是 System 命名空间中预定义的类型的别名。基本数据类型与它的别名表示的结构类型没有区别：保留字 char 就等价于 System.Char。基本数据类型在其他类型的基础上提供附加操作：

- 所有的基本数据类型都允许通过写入字面量而创建值。例如，"A"就是 char 类型

的字面量。

- 可以使用基本数据类型声明常量。
- 当表达式的操作数都是基本数据类型常量时，编译器就可以在编译时求表达式的值。这种表达式就是常量表达式。

下面就是在 C#程序中声明的基本数据类型的用法示例：

```
char myChar = 'A';
string myString = "My String";
int myInt = 42;
```

string 类型是独特的基本数据类型：它是基本类型中惟一的引用类型(而不是值类型)。这在逻辑上就有漏洞：作为基本数据类型，字符串可以声明为常量。但是常量不能放在堆中，因为堆在运行时分配内存，而常量却在程序编译时定义。常量字符串是个特例，它需要运行被称为内置(interning)的进程，这一点在后面讨论。

在深入讨论字符串数据类型之前，首先介绍字符串的基本单元——字符数据类型。

1.2.2 字符和字符集

在 http://whatis.techtarget.com 上有字符的定义。这个 Web 站点给出的字符定义如下：

按信息术语，字符是具有拼音文字或象形文字含义的书面符号，通常用于组成文本的单词、描述数值或表达语法符号。在今天的信息技术中，字符通常指有限数目的符号，包括某种语言字母表中的字母，十进制数字系统中的数字和某些特殊符号，比如宏(&)和“at”符号(@)。

人们已经为字符开发了几种计算机编码标准。个人计算机中最常用到的标准是 ASCII。IBM 大型机系统使用 EBCDIC。后来出现的 Windows 系统开始支持新的标准 Unicode。

这个定义非常好，它告诉我们，字符表示与计算机用户进行通信的一个符号。惟一需要指出的是，因为字符在历史上由一个字节表示，所以字符有时也用于表示二进制信息。此定义还指出，有几种标准可以用于字符编码。

ASCII 是 PC 传统上使用的存储字符的方案。ASCII 字符存储为 7 位的二进制数字。因此一个 ASCII 字符可以代表 128 个值中的一个值。ASCII 格式非常陈旧，最初它是作为电报文本开发出来的，因此某些 ASCII 值是电报文本的控制字符(例如，ASCII 值 23 表示“传输文本结束”)。它只使用 7 位，因为在 20 世纪 70 年代，传输和媒介都不是很可靠，每个字符通常要和一个校验位一起发送。

由于现代计算机以字节存储信息，而每一个字节占 8 位，那么每一个字符就浪费一位存储空间。一位存储空间似乎不算什么，但是如果 8 位全部使用，而不是 7 位，就

可以多表示 1 倍的字符值。

随着计算机的发展，程序员决定利用这多余的一位，这就是 Extended ASCII。使用了这多余的一位，在 ASCII 字符集中就可以再增加 128 个值。这些值包括语音字符，数学符号和图形字符。

扩展的 ASCII 字符集稍有变化。如果在不同的平台之间转换字符，比如从 Windows 转换到 Macintosh，则扩展的 ASCII 字符就具有不同的意义。例如，在 MS Windows 下的字母 C 与 Macintosh 下的字母 Á 对应。扩展的字符集也称为代码页。

如果希望在不同的平台之间使用扩展的 ASCII 字符，则应通过使用代码页来标识字符集。在 Internet 上，一般使用 HTML 和 XML 作为在不同平台之间转换的文本文档。在 HTML 中，可以按如下方式规定代码页：

```
<META HTTP-EQUIV="Content-Type"
      CONTENT="text/html; charset=ISO-8859-1">
```

在此示例中，代码页是 ISO-8859-1，也称为 Latin1，涵盖了几乎所有的西欧语系，包括英语、法语、德语、意大利语和西班牙语。

在 XML 中，使用一个声明指令表示字符编码：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

注意：

任何 ISO 8859 字符集的前 128 个字符代码总是和 ASCII 字符集完全相同。

在 20 世纪 80 年代后期，许多研究人员开始研究使用相同的字符集表示不同语言的问题。因此，在 20 世纪 90 年代早期出现了 Unicode Consortium。Unicode Consortium 负责定义 Unicode 字符的行为和相互关系，并向实现者提供技术支持信息。其中的一些成员包括 Adobe、Apple、Microsoft、IBM 和 Sun。

Unicode Consortium 声明如下：

Unicode 对于每一个字符都提供了惟一的值，而不管使用什么平台，也不管什么程序和语言。

Unicode 标准定义了用于今天大多数语言的字符的代码。其支持的语言包括拉丁语、希腊语、斯拉夫语、亚美尼亚语、希伯来语、阿拉伯语、梵语、奥里雅语、泰米尔语和布莱叶盲语。除了字母之外，Unicode 还包括标点符号、读音符号、数学符号、技术符号、箭头符号和装饰符号。

Unicode Version 2.1 标准为世界上的各种字母表、象形文字集和符号集的几乎 39,000 个字符提供了代码。另外，还为将来可能出现的用法提供了 18,000 个预留代码。Unicode 标准还包括 6,400 个代码值，便于软件和硬件开发人员在其内部指定专用的字

符和符号。

Unicode 标准使用额外的一个字节表示了非常多的字符。Unicode 采用 2 字节的编码方式，可以代表 256×256 = 65,536 个字符。但是，提高使用的灵活性必然会耗费大量的存储空间。Unicode 文本文件的容量是 ASCII 文件容量的两倍，其内存空间也是如此。因此，如果存储空间有限，而且仅仅使用 ASCII 字符，就应该考虑所使用的编程语言如何表示字符的问题了。

最初的 Unicode 规范不能满足所有的字符要求。汉语就有 55,000 个字符。为了解决这个问题，Unicode Standard 定义了代理(surrogate)。代理或代理对是一对 16 位的 Unicode 代码值，表示单个的字符。通过使用代理，Unicode 可以支持一百多万个字符。这个问题将在第 4 章详细介绍。

C#中的字符

C#将字符存储为 Unicode。单个的 Unicode 字符由 char 值类型表示，char 值类型映射为 System.Char 结构。由于 Unicode 由两个字节表示，所以一个字符可以具有十六进制的 0x0000～0xFFFF 之间的任意值。

.NET 中的字符与之前的语言中的字符相比，最大的区别就是，.NET 中的字符定义为带有成员的结构。这表示，我们无需使用操作字符的函数，仅仅使用字符的方法即可。另外，与数据类型相关的常量(如最小值)绑定到对象本身，而不是绑定到别处列出的值。

char 结构中一个最有用的方法是 GetUnicodeCategory()。这个静态方法将传输给它的字符类型分为 30 种，每一种用 System.Globalization.UnicodeCategory 枚举来表示，其中有 UppercaseLetter、ParagraphSeparator、DashPunctuation 和 CurrencySymbol。

char 结构还有对字符分类的许多方法，比如 IsDigit()和 IsPunctuation()。下面是使用这些方法的示例：

```
string band;
band = "The Albion Band";
Console.WriteLine(char.IsWhiteSpace(band, 3));          // Output: True
Console.WriteLine(char.IsPunctuation('A'));             // Output: False
```

使用 IsWhiteSpace()方法，可以测试字符串中的第 4 个字符是否是空白字符(使用基于 0 的索引)。IsPunctuation()方法用来测试字母“A”是否是标点符号。注意定义时是使用单引号来界定 A 字符的。这是在 C#中使用的语言结构，它指定了所使用的对象的类型是 char，而不是 string。可参见.NET Framework 文档中对 char 结构所支持的全部方法的介绍。

把一个字符从 char 值类型转换为 int，就可以获得该字符的字符代码(也称为代码点)；给数字的 Tostring()方法添加一个格式控制符(详见第 3 章)，就可以在通常的

Unicode 16 进制格式中指定输出：

```
int charCode = (int) 'A';                          // charCode is set to 65.
Console.WriteLine(charCode.ToString("x4")); // Will output 0041
```

同样，把给定字符代码从 int 转换为 char，就可以返回该字符代码所表示的字符：

```
int charCode = 100;
char character = (char) charCode;
Console.WriteLine(character);   // Outputs "d"
```

如果希望了解字符在内存中的表示方式，可以将这些字符写入文件，然后使用二进制编辑器查看这些字符。文件中的字节就是在内存中存储的字节。下面的示例打开一个文件，以 3 种不同的编码方式将文本写入该文件。因为文件扩展名是.bin，所以此文件必须使用二进制编辑器打开：

```
using System;
using System.IO;
using System.Text;

class Binary
{
  static void Main(string[] args)
  {
    FileStream fs = new FileStream("text.bin", FileMode.OpenOrCreate);

    StreamWriter t = new StreamWriter(fs, Encoding.UTF8);
    t.Write("This is in UTF8");
    t.Flush();
    StreamWriter t2 = new StreamWriter(fs, Encoding.Unicode);
    t2.Write("This is in Unicode (UTF16)");
    t2.Flush();

    StreamWriter t3 = new StreamWriter(fs, Encoding.ASCII);
    t3.Write("This is in ASCII");
    t3.Flush();

    fs.Close();
  }
}
```

使用内置在 Visual Studio .NET 中的二进制编辑器，就可以看到图 1-2 所示的数据。

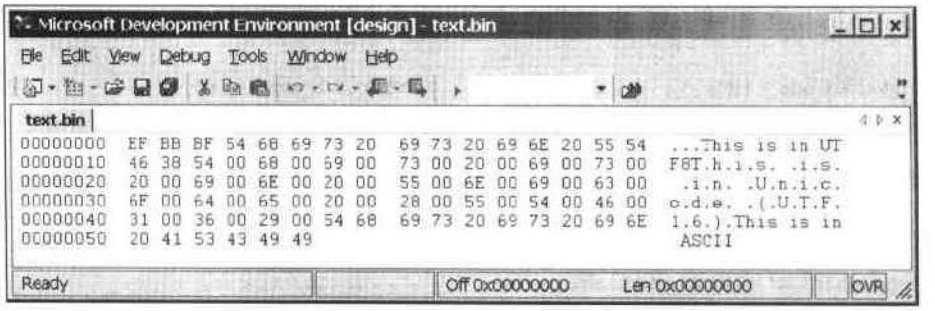


图　1-2

这里有几个要点。在所有 3 种编码方式中，低位字节都是相同的。低位字节是值的前十六位。例如，字母 T 在每一种编码方式中都由值 0x54 表示。Unicode 编码方式的每一个字符都有一个额外的字节，因此 Unicode 编码方式所占用的存储空间比 ASCII 和 UTF-8 所占用的存储空间多 1 倍。

前面介绍了字符，接下来介绍字符串。

1.2.3　字符串数据类型

字符串是字符序列。事实上，在 C 语言的最初版本中，字符串仅表示为字符数组。操作字节数组的函数也可以处理字符串。

.NET Framework 使用 String 类实现 string 数据类型。下面的代码给出了 string 和 string 数组的声明：

```
string myString = "My String!";
string[] arrString = {"to", "the", "max"};
```

字符串是独特的数据类型，因为它是惟一具有可变长度的基本数据类型，这就可能导致问题。计算机需要知道数据的长度，以便为之分配存储空间。为了讲解这个问题，可以参考下节内容“文本存储”。

1.3　文本存储

文本的存储取决于所使用的编程语言或软件平台。为了创建一个计算机应用程序，必须使用编程语言编写指令。编程语言本身就可以实现这些指令，也可以调用底层软件平台的服务来实现指令。

编译时，使用 C 编写的程序直接将指令转换为机器代码。机器代码不关心字符串的内容，它仅负责分配内存和复制字节。因此，操作字符串的大部分工作就由 C 语言

来完成。

在.NET 中，CLR 实现程序中所描述的功能。因此，CLR 必须理解文本分配和复制的方式。

文本的存储空间可以静态分配，也可以动态分配。如果静态分配，则当编译程序时，操作系统知道需要多少字符(也就是多少存储空间)。如果动态分配，操作系统直到运行时才为字符串分配存储空间。

编程语言通常使用两种方法记录字符串的长度。字符串要么在其开始处存储字符串的长度，要么包含一个特殊字符来表示结束标记。

当字符串的长度存储在其开始处时，字符串通常称为结构化的字符串或 Pascal 字符串。可以想见，Pascal 就是采用这种方法的编程语言。结构化字符串中的首个字节是长度字节，此字节给出了字符串中字符的数量。实际的字符数就是由长度字节规定的。

因为字符串的长度可以随时获取，所以对字符串的操作要快得多。例如，要判断字符串的长度，只需读取首个字节即可，而对于带结束符号的字符串，就必须扫描整个字符串才能找到结束标记。请考虑下面的伪代码：

```
for (int i =0; i < myString.Length; i++)
{
  // Perform some operation
}
```

如果字符串长度是 1,000，则 Pascal 字符串操作的次数就是 1,000。但是，对于带结束符号的字符串，操作的次数却是 StrLen(myString)*StrLen(myString) = 1,000,000！。尽管程序员无法得知编程语言如何实现操作，但是这个示例可以说明，理解以某种方式构造代码的含义非常重要。实现操作的较好方法如下：

```
int lengthString = myString.Length;
for (int i =0; i < lengthString; i++)
{
  // Perform some operation
}
```

对于带结束符号的字符串，这段代码可以以线性关系，而非二次方的关系减少操作的次数。

图 1-3 给出了 Pascal 字符串在内存中的存储方式。

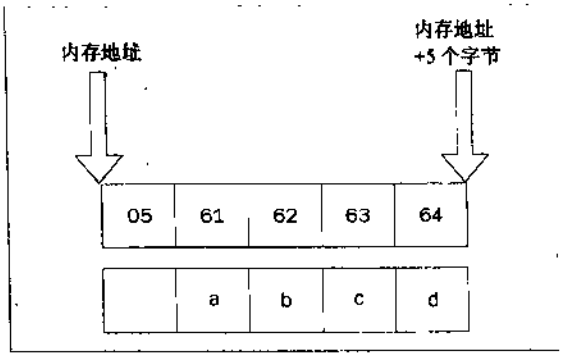


图　1-3

除了在字符串中加上前缀表示字符串长度之外，还可以在字符串中加入一个字符，表示字符串的终结。通常使用值为 0x00 的空字符，见图 1-4。

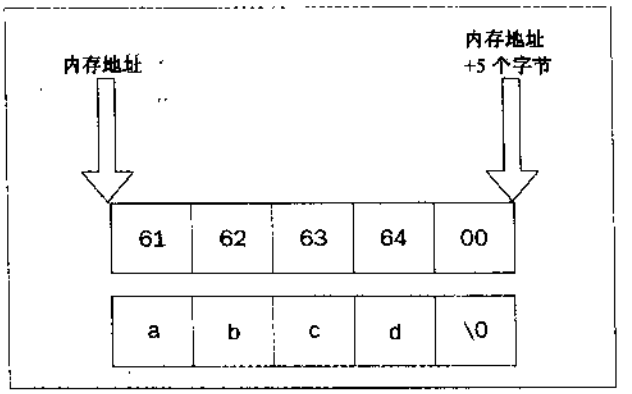


图　1-4

这种方法非常灵活，允许字符串有任意的长度，但是也存在缺点。为了判断字符串的长度，必须遍历每一个字符。另外，用户可能由于忘记使用结束字符串，或者改写了这个空字符而导致内存故障，这一点 C 程序员深有体会。

在每一种情况下，程序员都需要以对该种类型的字符串表示最有效的方式操作字符串。但是，如果把 Pascal 中的字符串操作方式用于 C 中以空字符终结的字符串，则会降低处理效率。因此，如果将在一种语言中执行字符串操作的代码直接翻译为另一种语言，就会降低效率。

由于字符串操作比较浪费系统资源，所以软件平台设计人员一直努力寻求提升其效率的技术。下面介绍其中一些技术。

1.3.1　高速缓存技术

为了使分配给字符串的内存空间最小，一些系统采用了高速缓存模式。Microsoft

Automation 采用了 Unicode 字符的 Automation 表示，也就是 BSTR(Binary Strings，二进制字符串)。

BSTR 以空字符为结束标记。但是，用户可以查询其长度，而无需扫描该字符串，因此这样的字符串可以包含嵌入的空字符。其长度值存储为 32 位的整数值，存储位置在内存中该字符串数据之前。BSTR 理论上的最大长度是 4,294,967,296 个字符，但是，操作系统可以根据所使用的资源决定实际可达到的长度。

Automation 可以高速缓存分配给 BSTR 的存储空间。这大大加快了存储空间分配和释放的速度。例如，如果应用程序分配一个 BSTR，然后释放，则释放后的自由内存空间就被 Automation 置入 BSTR 高速缓存中。如果之后应用程序分配了另一个 BSTR，则这一个 BSTR 就直接从高速缓存中获取存储空间。

空指针可作为 BSTR 变量的有效值。习惯上空指针被当作指向包含 0 个字符的 BSTR 的指针。

Automation 字符串高速缓存的大小为 60K，因此在这个容量范围内的连接操作的性能得到了优化。高速缓存被设计成以最有效的方式操作字符串。较长的字符串在堆中分配，对其进行连接要花费相当长的时间。这意味着首先连接较小的字符串可以显著提高操作性能，因为这样可以使用高速缓存。

例如，考虑下面的代码：

```
sLongString = sLongString + "." + Environment.NewLine
```

只有首先连接较小的字符串，才可能用到高速缓存，以提高性能：

```
sLongString = sLongString + ("." + Environment.NewLine)
```

1.3.2　内置

有时，应用程序需要反复重用某些字符串。利用这种特征可以大大节省内存空间，提高程序运行性能，为此就需要使用内置(interning)这种数据结构和算法集。

当重复使用相同的字符串时，内置可以提高程序运行性能。内置在前期会耗费一些资源成本，但是一旦就位，它可以支持字符串之间有效的相等性检查，因为它比较放在池中的对象，而不是字符序列。

CLR 自动维护一个称为“内置池”(intern pool)的表，此表包含程序中声明的每个惟一的字面字符串常量的单个实例，以及以编程方式创建的 System.String 类的任何惟一实例。

内置池实现为散列表。使用散列表意味着，一个字符串可以通过一个数字或“散列码”来表示。这样比较和搜索字符串就非常有效，因为这不是逐个字符地比较字符串，而仅仅是比较散列值。

内置池节省字符串存储空间。如果将一个字面字符串常量赋给几个变量，则每一个变量引用内置池中相同的常量，而不是引用具有相同值的 System.String 类的几个不同实例。

使用 StringBuilder 类创建的字符串可以使用 Intern()方法手工添加到内置池中。第 2 章将详细介绍内置的内容。

1.3.3 其他方法

对于专用的应用程序，存在有效存储文本的算法和数据结构。例如 tric，tric 是存储字符串的树结构，用于存储拼写检查程序和处理自然语言程序中容量巨大的词典。

下面给出一个简单示例，介绍 trie 的用法。考虑单词“exposition”和“exposure”。如果分开存储，则需要的存储空间是 18 个字符。但是通过观察可以发现，这两个单词的前 5 个字符是相同的。trie 首先存储相同的字符，然后使用指针指向各个单词中的其他字符。这就将存储的字符数量减少为 13 个字符。当要存储大量单词时，所节省的内存总量就非常可观。

某些可编程的 API，比如 Xalan-Java 2，实现一个 trie 类。如果该语言不是本身就支持此类，则程序员就必须构建此数据结构。

内置池的 CLR 实现就是散列表。但是它也可以实现为 trie。将 CLI 用作可用的共享源，它就可以做理论上的修改练习。

1.3.4 .NET 实现

前面介绍了存储字符串的各种方法。接下来给出一个示例，演示.NET 中字符串的存储方法。

下面是一个简单的程序，声明了一个字符串变量：

```
public class Class1
{
  static void Main(string[] args)
  {
    string myString = "hello string";
  }
}
```

当使用 ildasm.exe 工具查看时，实现此源代码的 IL 代码如下：

```
.method private hidebysig static void  Main(string[] args) cil managed
```

```
{
  .entrypoint
  // Code size       7 (0x7)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  ldstr      "hello string"
  IL_0005:  stloc.0
  IL_0006:  ret
} // end of method Class1::Main
```

可移植的可执行文件的作用仅仅是加载字符串。IL 代码只负责将字符 hello string 置入字符串对象，实际工作是由 CLI 实现完成的。

在 Shared Source CLI 的 string.cs 文件中，可以看到在外部声明的构造函数：

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
public extern String(char [] value);
```

为了观察实际的实现过程，必须将 C#代码转换为 C++代码。在 COMString.cpp 的代码中，可以找到实现构造函数的代码：

```
STRINGREF COMString::NewString(const WCHAR *pwsz)
{
  THROWSCOMPLUSEXCEPTION();
  if (!pwsz)
  {
    return NULL;
  }
  else
  {
    DWORD nch = (DWORD)wcslen(pwsz);
    if (nch==0) {
      return GetEmptyString();
    }
    _ASSERTE(!g_pGCHeap->IsHeapPointer((BYTE *) pwsz) ||
             !"pwsz can not point to GC Heap");
    STRINGREF pString = AllocateString( nch + 1);
    memcpyNoGCRefs(pString->GetBuffer(), pwsz, nch*sizeof(WCHAR));
    pString->SetStringLength(nch);
    _ASSERTE(pString->GetBuffer()[nch] == 0);
    return pString;
  }
}
```

AllocateString()调用返回 StringObject 的指针。C++ StringObject 类是.NET System.String 的内部表示。StringObject 实现的部分代码如下：

```
class StringObject : public Object
{
  friend class GCHeap;
  friend class JIT_TrialAlloc;

  private:
    DWORD    m_ArrayLength;
    DWORD    m_StringLength;
    WCHAR    m_Characters[0];
```

最后我们可以发现数据存储的位置，即 WCHAR 数组。WCHAR 是双字节的字符。可以看到，其中也存储了数组长度和字符串长度。数组长度是分配的内存大小，用 WCHAR 的个数表示。字符串长度就是字符串中 WCHAR 的个数。数组长度可能大于字符串长度，这表示字符串中存在未使用的存储空间。

m_StringLength 字段还有另一种作用。此值的两个高位表示字符串是否具有大于 0x7F 的字符。如果没有大于 0x7F 的字符，则表示这些字符属于标准的 ASCII 字符集。在这种情况下，系统就可以使用更快处理字符串的算法。其具体含义就是，如果字符串中的字符属于 ASCII 标准字符集，则系统就可以优化字符串处理性能。

可以看到，最终字符串存储为字符数组。其代码如下(其中字符串被缩短，以使之更具可读性)：

```
public static void Main()
{
    char[] myString = {'h', 'e', 'l', 'l', 'o'};
}
```

所生成的 IL 代码如下：

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       19 (0x13)
  .maxstack  3
  .locals init (char[] V_0)
  IL_0000:  ldc.i4.5
  IL_0001:  newarr     [mscorlib]System.Char
  IL_0006:  dup
  IL_0007:  ldtoken    field valuetype
```

```
'<PrivateImplementationDetails>'/'$$struct0x6000001-1'
'<PrivateImplementationDetails>'::'$$method0x6000001-1'
        IL_000c:  call       void
[mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray(class
[mscorlib]System.Array,
          valuetype [mscorlib]System.RuntimeFieldHandle)
          IL_0011:  stloc.0
          IL_0012:  ret
        } // end of method Class1::Main
```

<PrivateImplementationDetails>包含的代码比上述还多一些。看上去用了许多指令初始化字符串。每一个单独的字符都被加载到栈上，并存储在一个数组位置中。这个初始化字符串的方法看来有些笨拙。

另外，还可以显式地把字符数组指定为一个字符串，就像前面做的那样。这种方法更有吸引力，生成的 IL 代码也少得多。注意，使用这种方式会创建一个字符串对象，从而把 IL 指令的许多工作移交给了 CLR。

处理数组的方法也可以用于处理 char 数组。下面的示例演示了几个可以执行的操作：

```
using System;

public class Methods
{
  public static void Main(string[] args)
  {
    char[] myString = "hello string!".ToCharArray();
    Console.WriteLine(myString);

    // Create a substring
    for (int i = 6; i < myString.Length; i++)
    {
      Console.Write(myString[i]);
    }
    Console.WriteLine();

    // Find the last 'l' in the char array
    Console.WriteLine(Array.LastIndexOf(myString, 'l'));

    // Reverse the array
    Array.Reverse(myString);
    Console.WriteLine(myString);
```

```
        // Sort the array
        Array.Sort(myString);
        Console.WriteLine(myString);
        Console.ReadLine();
    }
}
```

在控制台上会显示如下输出：

```
hello string!
String!
3
!gnirts olleh
 !eghillnorst
```

有一些操作，比如 Sort()和 Reverse()，并不是 String 类和 StringBuilder 类的一部分，这些类在需要它们时，可以将字符串表示为字符数组。

1.4 字符串操作

字符串的操作一般可分为下面 5 大类：

- 连接字符串
- 从字符串中抽取子串
- 比较字符串
- 转换字符串
- 格式化字符串

下面详细介绍这些操作，然后学习如何组合使用这些操作，以用于下一节的常见文本操作。最后依据这些操作的性能(执行任务所需要的时间)和内存要求讨论操作的资源成本。

1.4.1 连接字符串

连接字符串可能是字符串最常用的操作。但是，它也是成本最为昂贵的操作，因为它需要占用大量的 CPU 资源来复制字节。字符串连接会给应用程序带来性能损失。

请看下面的代码示例：

```
string str1;
```

```
string str2;
string str3;

str1 = "abcde"
str2 = "fgh"
str3 = str1 + str2
```

图 1-5 演示了连接操作。

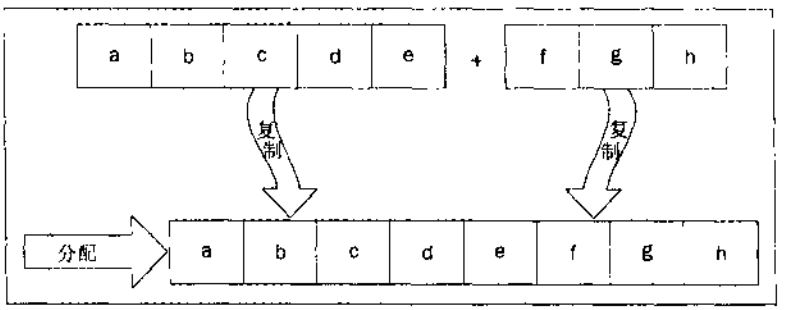


图　1-5

首先，必须分配新的内存空间才能容纳新的字符串。然后将字符从源字符串复制到目标字符串。

文章“HOWTO: Improve String Concatenation Performance”(KB Q170964)描述了通常的字符串连接的替代技术，使大型字符串的连接操作性能提升了 20 多倍。下面的简单循环在每一个迭代中都包含了一个字符串连接操作：

```
for (int i = 0;i <= N; i++)
{
  Dest = Dest + Source;
}
```

为了执行这些代码，必须执行下面这些步骤：

(1) 分配足够多的临时内存空间，以存放结果

(2) 将 Dest 复制到临时内存区的开始处

(3) 将 Source 复制到临时内存区的结束处

(4) 释放 Dest 旧的副本

(5) 为 Dest 分配内存区，使之足以存放结果

(6) 将临时数据复制到 Dest

下面就是 Microsoft 给出的结果：

```
Using standard concatenation
 1000 concatenations took 2348 ticks
```

```
2000 concatenations took 8954 ticks
3000 concatenations took 20271 ticks
4000 concatenations took 35103 ticks
5000 concatenations took 54453 ticks

Using pre-allocated storage and pseudo-concatenation
1000 pseudo-concatenations took 82 ticks
2000 pseudo-concatenations took 124 ticks
3000 pseudo-concatenations took 165 ticks
4000 pseudo-concatenations took 247 ticks
5000 pseudo-concatenations took 289 ticks
```

也就是说，优秀的字符串设计要使用“预分配存储区域和伪连接”技术。第 2 章将介绍.NET 如何使用 StringBuilder 类解决这个问题。

1.4.2 从字符串中提取子串

通常不是使字符串通过连接变得更大，而是使它们更小。Trim()方法用于清除字符串之前和之后的空格，程序员经常使用这个方法处理用户输入的文本。

有时需要从字符串中提取某些字符，这可以使用 Substring()方法来实现。实际上，使用 Trim()方法也可以提取子字符串，可以根据此方法的功能确定提取子字符串的位置。

在所有的情况下，这些功能都包括将字符串所有的字符或子串复制到另一个字符串。具体步骤如下：

(1) 判断要复制的字符数目

(2) 为这些字符分配存储空间

(3) 将字符复制到新的字符串中

由于涉及到了内存空间分配和复制操作，所以提取子串也非常耗费系统资源。

1.4.3 比较字符串

字符串比较是常见的操作。通常用户会输入一些文本，程序需要确定用户所输入的内容。一些字符串比较操作要求进行精确匹配。这是非常直接的操作，因为这是两个字符串之间逐个字符的简单比较。C#使用 op_Equality()方法完成这个任务。因为这个操作不涉及到内存分配或复制，所以此操作并不耗费系统资源。

在许多情况下，字符串比较用于验证工作。通常内容并不是最重要的，相反数据格

式却非常重要。为了确认电话号码格式是否为 XXX-XXX-XXXX，其中 X 的取值范围是 0～9，执行这项任务需要非常多的代码。

执行这种验证工作最好的方法是描述其应该具备的模型，然后针对模型验证文本。可以通过正则表达式实现模型匹配，所有的.NET 语言都通过 System.Text.RegularExpressions 命名空间支持正则表达式。本书第 5～7 章专门介绍这些内容，因为模式匹配提供了功能强大的比较工具。

1.4.4　字符串转换

非字符串的其他数据类型以有利于操作系统存储的形式表示。为了在字符串中显示这些数据类型，就需要将这些数据类型转换为字符。

.NET 使用 ToString()方法返回数据类型的字符串表示形式：

```
DateTime myDate = new DateTime(2002, 9,20);
Console.WriteLine(myDate.ToString());
```

这会在基于 US 的系统中显示 9/20/2002 12:00:00 AM。但是，根据用户所采用的实际数据格式的不同，其显示格式也可能有所不同。

将数据类型转换为字符串所采用的步骤如下：

(1) 基于源数据类型选择转换算法

(2) 判断转换后的字符串的大小

(3) 为字符串分配内存空间

(4) 执行转换算法，生成字符串

转换需要消耗多少内存资源取决于所使用的算法。转换也要耗费内存，因为除了算法要消耗资源外，决定字符串长度和在内存中的位置也要消耗内存。本书第 3 章将详细介绍.NET 中的字符串转换。

1.4.5　格式化字符串

常见的情况是将所有的信息收集到一个字符串中，一并传递给用户。另外，通常程序员也希望能控制将机器表示转换回字符数据的方式。这就是格式化工作，它非常类似于转换工作，只是程序员可以更好地控制转换的算法。

在.NET 中，为了实现格式化的字符串，可以使用 String.Format()方法。通过 Format()方法，可以规定格式化表达式或命名的格式。在 Visual Basic 6 中格式化字符串的难度在于，它一次只允许操作一段数据。程序员通常希望在一个字符串中放置许多段数据。C 中的格式化表达式功能更强大，它可以同时执行格式化字符串、执行转换、连接信息

等工作。第 2 章将学习.NET 如何在 C#中实现格式化表达式。

格式化方式可以根据用户的选择或语言设置有所不同。例如，可以按北美格式将 40 美元表示为$40.00，而按法国格式将 40 美元表示为 40.00 $。

格式化的资源成本类似于转换的资源成本，因为两者都需要执行算法，并需要分配一个字符串。

总之，引发内存分配和字节复制的字符串操作都是资源密集型的。扫描字符串以查找信息的字符串操作并不是资源密集型的。如果在循环中执行资源密集型的操作，则资源成本会显著地提高。下一节将演示有关这个问题的一些示例。

1.5 字符串用法

文本有多种用途。下面介绍其中的一些用途，并研究与这些用途相关的系统开销。

1.5.1 构建字符串

字符串最常见的用法是构造一些信息，并发送到其他位置。这些用法包括：

- 构建日志项，写到文本文件中
- 构建 HTML 页，发送到浏览器
- 构建 XML 文档，置入消息队列中
- 为自动电子邮件构建消息
- 构建传输到数据库中的 SQL 语句

随着 Web 的日益普及，HTML 已经成为创建用户界面最常用的方法。由于 HTML 是基于文本的，所以程序员可以使用文本操作和存储来构建并操作 HTML。使用 VBScript 的 ASP 代码如下：

```
<%
  Dim strHTML

  strHTML = "<HTML><BODY>"
  strHTML = strHTML & "My name is: "
  strHTML = strHTML & Request.ServerVariables("SERVER_NAME")
  strHTML = strHTML & "</BODY></HTML>"

  Response.Write strHTML
%>
```

这种方法非常易于理解。使用这种方法可以添加所有希望发送的信息，然后将这些信息发送到客户机。但是，这种方法效率较低。当声明 StrHEML 变量时，系统无法得知它需要多少内存空间。仅当给变量赋值时，系统才知道应该分配多少内存空间。对于每一个附加的连接操作，操作系统必须分配更多的内存空间，然后复制字符。前面说过，内存分配和复制非常耗用资源和时间。

上面的代码段潜在地开始了 4 个互不相连的内存分配过程，这并不太困难。但是由于这是 Web 页，所以对于每一个页请求都要重复进行这些内存分配过程。因此，如果 Web 页每秒接收 5 条请求，则每秒就要发生 20 次内存分配过程。

另一种方法是在 C#中使用类似的代码，将所有的字符串片断置入一个数组，然后把这些字符串片断连接为一个字符串：

```
<%
  string[] arrString = new String[3];
  string strHTML;

  arrString[0] = "<HTML><BODY>";
  arrString[1] = "My name is: ";
  arrString[2] = Request.ServerVariables("SERVER_NAME");
  arrString[3] = "</BODY></HTML>";
  strHTML = arrString[0] + arrString[1] + arrString[2] + arrString[3];

  Response.Write(strHTML);
%>
```

这就减少了内存分配次数和复制次数。如果开发人员在创建字符串之前能估计到字符串的大小，就可以具体分配该内存空间。下面介绍一个 C 语言中的示例：

```
char strBuffer[256];

sprintf( strBuffer, "<HTML><BODY>My name is: %s </BODY></HTML>",
      strServerName );
```

在这个例子中，当声明字符串时(实际上是字符数组)，内存空间是固定的，也就是说无需分配内存空间。但是，如果 strServerName 使整个字符串大于 256 个字符，就会引发问题，因为这会使内存崩溃。

为了安全起见，并避免浪费内存空间，应该动态分配内存：

```
char *pBuffer;

pBuffer = malloc( sizeof(char)*(sizeof(strServerName)+ 39 ) );
```

```
sprintf( strBuffer, "<HTML><BODY>My name is: %s </BODY></HTML>",
    strServerName );
```

这种方法要求动态分配内存，但容易产生内存泄漏问题。程序员必须记住释放分配给字符串的内存空间。当程序员构建字符串时，此方法并不太直观。下一章将介绍如何使用 Stringbuilder 对象分段建立字符串，这比把数组的各个元素连接在一起要简洁得多。

1.5.2 分析字符串

除了构建字符串之外，开发人员还需要分解字符串。这就是字符串分析。分析就是将结构化的数据分解为较小的数据段，这个过程要依据描述数据布局方式的规则。程序员要面临的最常见的问题就是需要分析由定界符(如逗号)分隔的一行文本。

在.NET 中，可以使用 Split()方法。此方法对一个字符串对象执行操作，它有两个重载方法。其中一个只带有一个 char[]数组参数，该数组指定了每个定界符。另一个重载方法把 char[]数组作为第一个参数，另一个参数是一个整数，表示要返回的元素个数。该方法返回一个 string[]数组。

位置文件(positional file)是经常需要分析的另一种文本文件。在位置文件中，无需使用定界符分隔字段，因为字段的长度是固定的。EDI(电子数据交换)传输是与位置文件类似的一个示例。分析位置文件可以使用 Substring()方法，它的资源成本很高。

对于定界文件和位置文件而言，最好使用特定的工具进行分析。在 UNIX 平台中可以使用 awk 工具，此工具已经导入到许多其他操作系统中，包括 Windows。它支持正则表达式，因此此方法也常用于搜索-替换操作。

还有其他更复杂的文本类型需要分析。包括：

- XML 文档中的元素
- 计算机程序的源代码
- 字处理文档中的文本

这种类型的分析超出了编程语言的处理能力。通常使用工具来分析这种类型的文档。这些工具专门用于优化执行要解决的特殊的分析问题。

例如，可以使用 XML 分析器(例如 Microsoft 的 MSXML 分析器)来分析 XML 文档。XML 分析器通常使用两种方法分析——DOM 和 SAX 方法。DOM 方法分析表示 XML 文档的文本，在内存中将之存储为对象模型，使程序员可以非常容易地访问这些信息。SAX 是另一种分析方法，此方法不采用对象模型表示 XML 文档，而是通知调用代码何时分析某些文本元素。这两种方法各有千秋——DOM 可以将 XML 数据编写为完整的 XML 文档，而 SAX 便于定位和替换数据，因为它无需搜索完整的文档。

这些方法也可以用于处理文本文件。如果需要在文本文件中置入所有的数据，则可以将整个文件加载到内存中。但是，如果希望搜索特定的数据项，或者单独处理每一行数据，就需要一次在内存中加载一行数据。只要任务完成了，就可以中止处理进程，无需读取文件中的所有数据行。

1.6　国际化

术语国际化和本地化经常互换使用，但是它们的含义具有非常明显的区别。

国际化指规划和实现产品和服务，以便这些产品和服务可以适应各种语言和文化。国际化有时也称为“本地化实现”。Microsoft 采用术语 world-ready 表示国际化。有时 Internationalization 也称为 I18N，即 Internationalization 的首字母 I，18 指 18 个中间字符，而最后是字母 N。

本地化(localization)指将应用程序适应具体的语言、文化和指定的本地“外观”的过程。这包括翻译应用程序的文本，确保其格式(比如货币和日期)符合具体文化的习惯。

.NET 中的文化指用于规定所使用的语言、各种日期、时间和该地区具有的数字格式等内容的具体语言和地区。例如，en-US 规定使用的语言是 English，所使用的各种数据类型应适用于 US。如果使用 en-GB，则其语言就是相同的，但货币符号是£，日期格式是 dd/mm/yyyy，当然还有其他的地区区别。

理想情况下，所开发的产品或服务要便于实现本地化。国际化产品或服务较易于实现本地化。程序员的工作是保证软件的国际化。当在另一种语言中发布软件时，软件必须首先交给当地的专家进行处理。

国际化的核心思想就是确保任何语言的文本都可以置入用户界面中。当对文本进行国际化处理时，要遵循一些简单的规则。

如果字符串、字符、常量、屏幕位置、文件名称和文件路径是硬编码时，就非常难于跟踪和本地化。最好的方法是将所有可本地化处理的项单独放到资源文件中，并尽量减少编译时的依赖关系。

最好不要声明一个刚好容纳一个单词或一个句子的缓冲区。因为当翻译文本时，所翻译的文本可能会更大。例如，应用程序要为 OK 单词声明一个 2 字节的缓冲区，此 OK 单词作为 OK 按钮的文本。但是，在西班牙语中，单词 OK 被翻译为 Aceptar，这就会使应用程序的缓冲区溢出。

还要避免使用字符串合成。例如，考虑将 wrong file 和 wrong directory 翻译为意大利语。其结果分别是 file errato 和 cartella erratta。如果试图使用语法"wrong " + object 进行字符串合成，是不会起作用的。

前面提到的建议除了用于字符串外，还适用于字符集。本章前面已经介绍过字符集。

一种语言的字符需要显示，以使用该语言显示用户界面。必须留下足够的数据空间来容纳多字节的字符代码，比如日语中的汉字。某些语言需要更多的字符来表示概念，因此在用户界面中也就需要更多的空间。

将字符串置入资源文件中也非常重要。由于这种资源文件最可能采用翻译器，所以不在资源文件中置入令翻译器混淆的字符串非常重要。如果已经开发了国际化的应用程序，那么应用程序的本地化仅仅需要将用户界面翻译为本地语言即可。这意味着应用程序的可执行代码应该与文本分离。为此就要使用资源文件。

.NET 资源文件

.NET Framework 提供了 3 种方法来创建资源文件：分别将文本文件、XML 文件或二进制资源文件作为资源文件。文本文件和 XML 文件在插入到应用程序时就转化为二进制资源文件。可以使用 Resource File Generator(Resgen.exe)工具将这些文件编译为资源文件。

如果字符串是惟一存储的资源文件，则可以使用文本文件。文本存储为名-值对。其中可以添加注释，使用分号表示注释。下面是一段文本文件示例：

```
[strings]
;prompts
promptAge = Please enter your age.

;error messages
msgNotFound = File not found.
msgNoRecords = No records were found for that query.
```

使用文本文件的优点是它非常易于创建和理解。翻译器可以在 Notepad 或其他任何文本编辑器中编辑文件。其缺点是，如果文件中有了语法错误，则直到这些文件返回给开发人员，才会发现这些错误。

XML 文件也可以用作资源文件，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <xs:schema id="root" xmlns=""
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="data">
      <xs:complexType>
        <xs:sequence>
```

```
            <xs:element name="value" type="xs:string" minOccurs="0"
                        msdata:Ordinal="2" />
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" />
        </xs:complexType>
      </xs:element>
    </xs:schema>
    <data name="promptAge">
      <value>Please enter your age.</value>
    </data>
    <data name="msgNotFound">
      <value> File not found.</value>
    </data>
    <data name="msgNoRecords">
      <value> No records were found for that query.</value>
    </data>
  </root>
```

第一段代码是头文件，包含了文件实际内容的模式。翻译器仅处理内容，而不处理头文件。由于 XML 文件仍然是文本文件，所以也可以在 Notepad 或任何其他的文本编辑器中编辑。也可以使用其他一些工具编辑文件。例如 Visual Studio .NET 具有内置的 XML 编辑器，就可以编辑.resx 文件。

还可以使用其他 XML 编辑器。许多 XML 编辑器(比如 SoftQuad XMetal)非常易于使用。翻译器主要翻译文本，而无需关注文件的语法。另一个优点就是 XML 编辑器可以保证写入文件的数据格式符合 XML 格式(而且是有效的 XML 格式)。

1.7　小结

本章回顾了在计算机应用程序中使用文本的各方面内容。使用文本包括处理单个的字符和连续的字符集——也就是字符串。

许多字符串操作对应用程序的性能影响非常大，因为其中涉及到内存分配和字符复制。适当的字符串设计是保证程序员将发生问题的可能性降到最低程度。

本书余下的章节将介绍在 C#中操作文本的各种方法，并向您推荐实现文本操作的最佳方法。

第2章　String类和StringBuilder类

前一章讲述了大量的字符串处理问题。特别地，由于字符串是长度可变的数据类型，也会使性能和内存管理出现问题。本章讲述了 Microsoft .NET Framework 是如何利用 String 类和 StringBuilder 类处理这些字符串问题的。

为了学习 C#中文本的使用方法，本章讲述了下列内容：

- 如何利用 String 类
- 如何利用 StringBuilder 类
- C#中的文本操作
- 使用 String.Format 类
- 在 C#中使用文本

在 C#中执行文本操作的方法不止一种。本章讲解了各种情况下最适用的方法，其中有些方法是通过本章后半部分创建的一个性能测试应用程序来讲解的。

2.1　学习本章要用到的工具

要对 String 类和 StringBuilder 类的工作本质进行更好的理解，需要用到两类工具。

第一个工具是 Microsoft Shared Source CLI(Common Language Infrastructure，通用语言基础结构)实现，也称为 Rotor，可以在 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/mssharsourcecli.asp 获取。您不需要下载这个共享资源，除非您喜欢收集代码，因为本章有这些代码的摘录。

这是一个 Microsoft .NET Framework 实现的源代码，但不同于正式发布的.NET Framework 的实现代码。实际上，Microsoft 声明："这个实现的代码与 Microsoft 的商业 CLR 实现的代码有重大的不同，两者都促进了可移植性并使代码基址更易访问。"不过，这个源代码可以帮助了解 String 类和 StringBuilder 类是如何实现的。

第二个工具是 MSIL Disassembler (ildasm.exe)，本章中，它用来显示编译器生成的 Intermediate Language(IL，中间语言)。这个工具是由.NET Framework SDK 提供的。编译代码时，该编译器把 C#代码编译为 Microsoft 中间语言(MSIL)代码，再让 CLR 编译为本机代码。这个工具可以让用户在文件编译成本机代码之前看到这个文件的内容。

MSIL 包括了很多加载、存储、初始化和调用对象的方法的指令，还包含算术运算、逻辑运算、流程控制、内存直接访问、异常处理和所有.NET Framework 能够执行的操

作的指令。由于 MSIL 不是本机的程序集代码，它更易于读取和理解，所以 MSIL 可以让程序员理解代码将如何执行，因为程序员可以看到底层的、实际执行的指令。

代码执行之前，必须编译成本机代码，这是由即时(JIT)编译器完成的——Rotor 提供了这个编译器的实现。

2.2 文本结构

第 1 章介绍的一个重要问题是字符串操作的性能问题。这个问题简化为两个基本问题：内存分配和字节复制。所以，通过实现策略将内存分配和字节复制最小化，性能就会提高。

为了减少内存重新分配和字节复制的数量，可以分配比最初所需更多的空间。这样，当字符串增加时，就只需将新的字符复制到已经分配好的空间中。这个策略通常用来提高性能。当然，在这种情况下，也可能浪费内存。另外，在运行之前，我们通常不可能知道需要多少内存。开发人员几乎不能确定内存分配的数量是否充足，所以就需要一种策略来增加可用空间。大多数情况下，开发人员应该集中精力解决手边的业务问题，而不是去关心内存分配。内存分配策略应该集成到平台中。

Java 和.NET 平台在文本构建上采用了相似的方法。它们都有一个描述字符串的类，和一个建立字符串的类。命名也十分相似，Java 使用的是 String 和 StringBuffer，而.NET Framework 使用的是 String 和 StringBuilder。命名空间的位置也很相像。对于.NET，String 位于 System 命名空间，StringBuilder 位于 System.Text 命名空间。而 Java 中，它的 String 和 StringBuffer 都位于 java.lang 命名空间(它包含了 Java 核心类)。

.NET Framework 的 String 类型是不可变的。这就是说，一旦它被赋了一个值，就不能改变了。从编码的角度来看，似乎是值发生了改变，实际上是 CLR 在后台创建了另一个 String 对象，并把它赋予变量。不可改变的一个优点在于它是线程安全的。由于它不能被改变，就不用关心线程之间的读/写或者写/写冲突了。

当需要频繁改变一个字符串时，应使用 StringBuilder，因为 StringBuilder 会在任何字符串操作需要时，为字符进行内存的分配和再分配。

第 1 章提到，String 是基本数据类型中惟一的引用类型。所有其他的类型都是值类型。引用类型只存储指向托管堆中数据的一个指针。另外，当复制一个引用类型的对象时，会得到一个新的引用，但数据是相同的。而复制一个值类型会得到具有相同值的一组新数据。

把 System.String 定义为不可改变，它就可以具有值的语义，但仍保持为引用类型。例如，考虑下面的代码：

```
using System;
```

```
using System.Text;

namespace Wrox.Text.Chapter2
{
  class Immutable
  {
    static void Main(string[] args)
    {
      string s1 = "Rock";
      string s2 = s1;

      Console.WriteLine("s1 = " + s1);
      Console.WriteLine("s2 = " + s2);

      s2 = s2.Replace("ck", "ll");

      Console.WriteLine("s1 = " + s1);
      Console.WriteLine("s2 = " + s2);
    }
  }
}
```

第二行之后，str1 和 str2 都指向了堆中的同一组数据。调用 Replace()方法之后，它们包含了不同的引用。如果 str2 不是不可改变的，那么 str1 和 str2 都将包含值"Roll"。这是因为我们改变了堆的内容，并且 str1 和 str2 都指向了这个新的数据。另一种相反的情况为，Replace()方法创建了一个新的字符串(堆中的数据)，并把对它的一个引用赋给 str2。最初的 str2 保持原来的值，这样 str1 也就不会改变了。字符串的内容直到垃圾收集显式释放内存之后才会被重写。

2.3 String 类

System.String 在第 1 章中是作为基本引用类型介绍的。System 命名空间还包含了.NET 开发所用到的所有核心数据类型。它包括了基础类和定义通用的值类型和引用数据类型的基类。下面几节将详细讲解这些内容。

表 2-1 所示为本章涉及到的所有 String 类的公共成员，完整的表在附录 A 中给出。

表 2-1　String 类的公共成员

成员名称	成员类型	成员描述
Compare(String s1, String s2)	静态方法	按照当前的文化设置，对两个给定的字符串执行区分大小写的比较
Compare(String s1, String s2, Bool ignoreCase)	静态方法	按照当前的文化设置，对两个给定的字符串执行一次比较。如果 ignoreCase 为 True，就执行不区分大小写的比较
CompareOrdinal(String s1, String s2)	静态方法	不考虑当前文化信息，对给定的两个字符串执行一次比较
CompareTo(String s)	实例方法	对给定字符串与实例字符串执行一次区分大小写的，并考虑文化信息的比较
Copy(String s)	静态方法	返回一个与给定字符串具有相同值的新的字符串
CopyTo(Int sourceIndex, Char dest(), Int destIndex, Int count)	实例方法	从实例字符串将一个子串复制到给定的目的字符串
EndsWith(String s)	方法	如果实例字符串是以给定的字符串结束，就返回 True
Equals(String s)	方法	如果实例字符串与给定的对象具有相同的值，就返回 True
Format(IFormatProvider provider, String format, ParamArray args)	方法	返回一个包含有格式说明符的字符串，这个格式说明符将依照给定的格式提供器由参数数组替换
Intern(String s)	方法	返回一个对给定字符串的引用。如果字符串不是内置的，这个方法将会把它内置。内置字符串详见后面的内容
Replace(String oldString, String newString)	方法	在实例字符串中用 newString 替换所有出现过的 oldString
Split(ParamArray Separators)	方法	按照给定的字符分隔符将实例字符串分离
Substring (Int startPos, Int length)	方法	从指定位置开始返回一个指定长度的子串
ToString()	方法	返回一个对实例字符串的引用
ToString(IFormatProvider format)	方法	返回一个对实例字符串的引用

2.3.1 内置字符串

第 1 章提到，CLR 维护一个字符串的内置池(intern pool)。如果查看 CLI 的共享源代码实现，就会看到每一个 AppDomain 对应一个内置池。

为了弄明白字符串内置是如何进行的，考虑下面的例子：

```
const string s1 = "dog";

// Displays True if the string is interned.
Console.WriteLine(String.IsInterned(s1) != null);
```

图 2-1 是这个例子的图示。

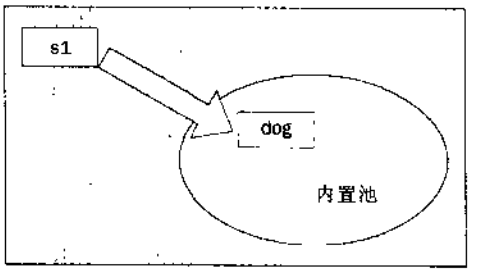


图 2-1

如果一个字符串是不可改变的，一如往常那样，被初始化为一个值，那么它就会如同一个常量。下面看看编译器是否会将它内置：

```
string s1 = "dog";

// Displays True if the string is interned.
Console.WriteLine(String.IsInterned(s1) != null);
```

代码执行后，会显示 True，这说明字符串已经被内置。再看看另外一个例子。在这个例子中，我们在最初的字符串声明中，把字符串作为一个单独的指令：

```
string s1;
s1 = "dog";

// Displays True if the string is interned.
Console.WriteLine(String.IsInterned(s1) != null);
```

编译器知道字符串将被固定，因此将它内置。结果，上面两段代码例子是完全相同的。那么，字符串何时不会被内置？看看下面的代码：

```
String s1 = "Dog";
```

```
// The next line displays True.
Console.WriteLine(String.IsInterned(s1) != null);

s1 = s1 + "s";

// The next line displays False.
Console.WriteLine(String.IsInterned(s1) != null);
```

在这里，字符串被检测了两遍，但返回的是两个不同的值。要记住 s1 是一个字符串的引用，而不是字符串真正的值。String 对象在声明时初始化为一个值。编译器将这个字符串内置，是因为它知道这个字符串在编译时的内容。新的代码行没有为先前的字符串添加"s"；它们创建了一个新的字符串，并将引用赋值给新的字符串。由于这个字符串是在运行时创建的，所以它不能自动内置，如图 2-2 所示。

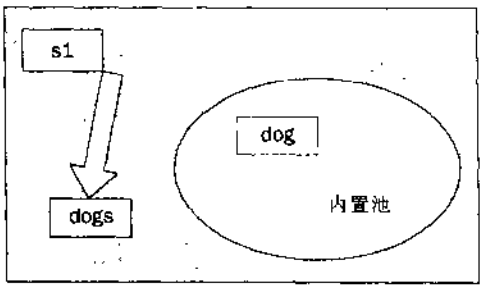


图　2-2

不过，您可以利用编程的方式将它内置：

```
String s1 = "Dog";

// The next line displays true.
Console.WriteLine(String.IsInterned(s1) != null);

s1 = s1 + "s";

String.Intern(s1);

// The next line displays true.
Console.WriteLine(String.IsInterned(s1) != null);
```

综上所述，系统如果在编译时知道一个字符串中的字符是什么，就会内置这个字符串。我们已经看到，被内置的字符串可以是一个常量，也可以不是。

另外一个要讨论的问题是，所有位于池中的字符串将会发生什么变化。看起来，那

些没有被引用的字符串可能造成内存丢失。不过，垃圾收集器能够检查出池中的哪些字符串不再被引用，并将它们从内置池中清除。例如，如果变量 s1 超出了作用域，只要应用程序中没有其他值为 Dog 的字符串变量，就不再会有任何对 Dog 的引用。

2.3.2 构建

在 C#中，初始化一个字符串的方法有许多。看看下面这个方法：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class Construction
  {
    static void Main(string[] args)
    {
      const string s1 = "My string 1.";
      string s2 = "My string 2.";
      string s3;
      s3 = "My string 3.";
    }
  }
}
```

由于字符串是不可改变的，所以任何声明之间并不存在功能上的区别，除了编译器不允许在代码中改变 s1 引用(因为它声明为 const)。在所有这 3 个例子中，一个字符串值一旦在堆中创建，它就不能再改变。任何对 String 对象的修改都会创建一个新的 String 对象，而包含在变量中的引用将指向这个对象。在上面的例子中，s1、s2 和 s3 都被内置。

这 3 个方法所做的事情似乎相同，但它们之间是否存在细微的差别呢？IL 展示了它们的不同之处。代码简化为下面的 IL。注意，IL 可能会有细微的不同，因为不同的编译器设置会创建不同的 IL 代码。不要为每一个 IL 指令的确切意义而烦恼；我们所关心的是这些源代码如何编译为 IL。编译上面的代码，并输入 ildasm Construction.exe，就可以看到上面代码的 IL 了。

```
method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
```

```
// Code size       13 (0xd)
.maxstack  1
.locals init (string V_0,
              string V_1)
IL_0000:  ldstr      "My string 2."
IL_0005:  stloc.0
IL_0006:  ldstr      "My string 3."
IL_000b:  stloc.1
IL_000c:  ret
} // end of method Construction::Main
```

首先要注意的是，“My string 1”已经不再存在。一个常量的作用和宏替换的作用相同。如果它没有被引用，就不会包括在 IL 代码中。如果我们改变代码来使用这个常量，这个值就会出现在 IL 代码中：

```
const string s1 = "My string 1.";
string s2 = "My string 2.";
string s3;
s3 = "My string 3.";

Console.WriteLine(s1);
```

将字符加载到一个 String 对象，然后字符串输出到控制台。

```
method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       23 (0x17)
  .maxstack  1
  .locals init (string V_0,
                string V_1)
  IL_0000:  ldstr      "My string 2."
  IL_0005:  stloc.0
  IL_0006:  ldstr      "My string 3."
  IL_000b:  stloc.1
  IL_000c:  ldstr      "My string 1."
  IL_0011:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0016:  ret
} // end of method Construction::Main
```

下一个字符串“My string 2.”执行 ldstr(加载字符串)操作。下文摘自公共语言基础结构(CLI)文档，标题为“Partition III – CIL Instruction Set”：

“ldstr 指令将一个代表字面量的新字符串对象作为字符串(必须是字符串自变量)存储在元数据中。ldstr 指令将分配内存并执行任何所需的格式转换，把文件中使用的格式转换成运行库所需要的字符串格式。CLI 保证两个 ldstr 指令的结果能够指向两个元数据标记，这两个标记能让相同序列的字符精确返回相同的字符串对象(这个过程称为“字符串内置”)。

文档中说，ldstr 将负责所有的困难工作。String 对象一旦创建，就立即存储在变量中。下面再来看一看“My string 3.”会有什么变化。关键字 string 是 String 类的一个别名。进入 IL 指令后，它们就变成完全相同的东西。惟一的区别是 Visual Stidio .NET 会突出显示 string，因为它是一个关键字，而不是 String 类名。

这样分析之后，可以看出 Const 是初始化字符串最有效的方法，因为它涉及的 IL 指令最少。

使用字符串构造函数的情况有两种。一种构造函数允许使用固定的字符构造字符串。例如，下面的代码创建了一个含有 4 个“D”的字符串：

```
string s2 = new string('D', 4);
```

似乎还可以创建如下的字符串：

```
string myString = new string("dog");
```

这个指令在 VB.NET 中可行，但在 C#中不行。C#编译器会给出如下结果：

```
CodeFile1.cs(11,23): error CS1502: The best overloaded method match for 'string.String(char*)'
has some invalid arguments
CodeFile1.cs(11,34): error CS1503: Argument '1': cannot convert from 'string' to 'char*'
```

编译器把“dog”转换为一个字符串基本类型，然后尝试调用以一个 String 对象作为参数的 string 构造函数。String 类的任何构造函数都不以 String 类型作为参数，所以必须使用 Clone()方法。解决这个问题的一个较麻烦的方式如下所示：

```
object myObj = "dog".Clone();
string myString = (string) myObj as string;
```

2.3.3 字符串的转义

众所周知，一些字符要表示为字符串十分困难。所有的字符串变量都是通过放入引号而完成赋值，所以，下面的代码将会引发一个语法错误：

```
string myString = "He said, "Dogs and cats."";
```

编译器会认为字符串是“He said,”，但不知道该如何处理剩下的字符，于是返回

一个语法错误。在 C#中，引号必须用一个转义序列来表示：

```
string myString = "He said, \"Dogs and cats.\"";
```

还有其他字符需要用转义序列来表示。表 2-2 列出了应用程序中所使用的转义序列。

表 2-2　应用程序中所使用的转义序列

转 义 序 列	Unicode 值	说　明
\0	x0000	空
\a	x0007	警告
\b	x0008	退格
\t	x0009	水平制表符
\n	x000A	换行
\v	x000B	垂直制表符
\f	x000C	换页
\r	x000D	回车
\"	x0022	双引号
\'	x0027	单引号
\\	x005C	斜杠

C#还允许使用'@'符号表示逐字的字符串字面量。在字符串的前面放置一个运算符，字符就会逐字被解释(引号转义序列例外)。例如，下面的代码：

```
Console.WriteLine(@"hello \t world");
```

会在输出中显示下述内容：

```
hello \t world
```

如果没有加上逐字运算符，就会插入一个制表符。

2.4　StringBuilder 类

我们已经讨论过，String 类的不可改变性使它更像一个值类型而不是一个引用类型。然而，这也有一个副作用，即每次执行字符串操作时，都会创建一个新的 String 对象。

StringBuilder 类解决了在对字符串进行重复修改的过程中创建大量对象的问题。StringBuilder 类以 Char 为单位向字符串分配空间，所以操作中不需要进行额外的内存

分配。

表 2-3 列出了 StringBuilder 类的一些属性和方法。

表 2-3　StringBuilder 类的成员

成员名称	成员类型	成员描述
Capacity	属性	当前为实例分配的字符数量。不一定与 Length 属性的值相同
Chars(int index)	属性	在字符串中的 index 位置取得或设置字符
Length	属性	字符串中字符的数量。可以设置为小于当前字符串值以截短它
MaxCapacity	属性	这个实例中可以被分配的字符的最大数目
Append(<type>)	方法	追加给定值的字符串表示。有针对每一个.NET 基本类型的重载方法。为简洁起见，不再列出
EnsureCapacity (Int capacity)	方法	如果当前容量小于指定容量，内存分配就会增加内存空间以达到指定容量
Insert(Int position, String s, Int count)	方法	在指定的 position，按 count 指定的次数插入 s
Replace(Char oldChar, Char newChar)	方法	在实例字符串中用 newChar 替换所有出现的 oldChar
Replace(String oldString, String newString)	方法	在实例字符串中用 newString 替换所有出现的 oldString
Replace(Char oldChar, Char newChar, Int startPos, Int count)	方法	在实例字符串的 startPos 位置和 count－1 位置之间用 newChar 替换所有出现的 oldChar
Replace(String oldString, String newString, Int startPos, Int count)	方法	在实例字符串的 startPos 位置和 count－1 位置之间用 newString 替换所有出现的 oldString
ToString()	方法	返回一个 String 对象的引用，它的值与包含在 StringBuilder 实例中的值相同

2.4.1　长度和容量

StringBuilder 类包含了 Length 属性和 Capacity 属性。与我们所期望的相反，Length 属性并不是只读的。如果 Length 设置为小于字符串长度，字符串就会被截短。如果设置为大于当前长度，就需要在末尾添加空格。下面摘自 StringBuilderLength.cs 的代码演示了这种行为。

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class StringBuilderLength
  {
    static void Main(string[] args)
    {
      StringBuilder sb = new StringBuilder("Hot dog!");
      Console.WriteLine(sb.ToString());   // Displays "Hot dog!"

      sb.Length = 3;
      Console.WriteLine(sb.ToString());   // Displays "Hot"

      sb.Length = 10;
      Console.Write(sb.ToString());       // Displays "Hot       "
      Console.WriteLine("(ends)");        // Indicates the string end
    }
  }
}
```

容量定义了用于文本处理的内存数量，默认容量为 16。如果知道字符串将会大于 16 个字符，就应该显式设置容量，这至少会避免一次内存的再分配。如果将一个字符串作为参数提供给构造函数，容量将会设置为最接近 2 的幂的值。例如，如果字符串长度为 17，容量就会是 32。Rotor 源代码在 comstringbuffer.cpp 文件中包含方法 CalculateCapacity。当用一个初始字符串创建一个新的 StringBuilder 时，就将调用这个方法。下面的 C#共享源 CLI 代码实现了容量计算：

```
//Double until we find something bigger than what we need.
while (newCapacity < requestedCapacity && newCapacity > 0)
{
```

```
            newCapacity*=2;
        }
```

在这个新字符串的例子中，requestedCapacity 就是字符串的长度。Capacity 属性与 Length 无关。不过，Capacity 总是等于或大于 Length。

下面的代码包含在 StringBuilderAppend.cs 中，演示了 Capacity 属性是怎样随着字符串的增加而变化的。Append()方法将指定字符的给定数量与字符串连接起来：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class StringBuilderAppend
  {
    static void Main(string[] args)
    {
      StringBuilder sb = new StringBuilder();
      Console.WriteLine(sb.Capacity + "\t" + sb.Length);

      sb.Append('a', 17);
      Console.WriteLine(sb.Capacity + "\t" + sb.Length);

      sb.Append('b', 16);
      Console.WriteLine(sb.Capacity + "\t" + sb.Length);

      sb.Append('c', 32);
      Console.WriteLine(sb.Capacity + "\t" + sb.Length);

      sb.Append('d', 64);
      Console.WriteLine(sb.Capacity + "\t" + sb.Length);
    }
  }
}
```

在控制台上生成了下面的输出：

```
16      0
32      17
64      33
128     65
256     129
```

在这种情况下，有四次内存再分配。如果初始容量设置为 256 个字符，就不会再分配内存了。在文件 stringbuilder.cs 中查看共享的源代码实现，可以确切地看到新容量是在哪里进行计算的。

```
newCapacity = (currentString.Capacity)*2;
if (newCapacity < requiredLength) newCapacity = requiredLength;
if (newCapacity > m_MaxCapacity) newCapacity = m_MaxCapacity;
```

可以利用属性 StringBuilder.MaxCapacity 获取最大容量。在 Microsoft .NET Framework 发布的实现方式中，这等价于 Int32.MaxValue，都是 2,147,483,647 (2^{31}-1)。这个范围很大，在到达这个最大极限之前，可能会遇到许多其他的问题，比如系统内存空间的缺乏。

默认容量、最大容量，以及容量如何增大，在各个实现中都是不同的。例如，共享的源代码实现的行为可能会与发布的实现不同(但默认容量仍旧是 16)。利用系统行为来优化性能是很重要的，但是要清楚，如果代码是在另外一个实现中运行，也许不能获取相同的效果。

如果编写了下面的代码：

```
if (myString.Capacity < myCapacity) myString.Capacity = myCapacity;
```

就可以用下面具有相同功能的代码替换它：

```
myString.EnsureCapacity(myCapacity)
```

如果当前容量小于新的容量，内存就会被再分配。当前容量可以减少，只要不小于 Length 即可。如果小于 Length，会抛出一个 System.ArgumentOutOfRangeException 异常。

2.4.2　ToString()方法

我们通过调用 ToString()方法取得了包含在 StringBuilder 类中的字符串。可以检索整个字符串，也可以检索部分字符串。下面的代码演示了这两种方法。

```
StringBuilder sb = new StringBuilder("This is my string.");
String s1 = sb.ToString();
String s2 = sb.ToString(5,5);

Console.WriteLine(s1);      // Displays "This is my string."
Console.WriteLine(s2);      // Displays "is my"
```

对 ToString()的第一次调用创建了一个新的 String 对象，且字符被复制到这个对象。这样就保持了 String 类的值的语义。不过，分配字符串还是有一些小的性能开销。

共享源代码 CLI 显示，第二个 ToString()方法调用了 String 类的 Substring()方法。最终发布的版本也许不是这样，因为许多类甚至都不是用 C#编写的，而是用本机代码编写的：

```
public String ToString(int startIndex, int length)
{
  return   m_StringValue.Substring(startIndex, length);
}
```

StringBuilder 的很多其他方法与 String 的方法是重复的。而且，一个类中的方法可以调用另一个类中的方法。下面几节将了解并比较它们的不同方法。

2.5 字符串操作

所有的编程语言都支持字符串操作。对字符串的操作有以下 4 类：

- 连接字符串
- 从字符串中提取子串
- 比较字符串
- 格式化字符串

我们将详细讲解这些操作，并了解如何将常见的文本操作组合在一起使用。操作的资源开销将会从性能(执行一个任务的时间)和内存两方面来讨论。

执行这些操作时，通常要在 String 类和 StringBuilder 类之间进行选择。这里提供了一些建议。

注意：

使用 Char 数组相对于 String 和 StringBuilder 类的优点在这里没有讨论。基于某些测试，发现对 Chars 的操作通常运行得比较慢。共享源代码 CLI 对于这种情况出现的原因提供了一个线索，那就是，Char 数组在操作执行之前要转换为 String 类型。只在操作可以更简单时使用 Char 数组，但即使这样，也有必要测试性能上的损失。

2.5.1 连接字符串

下面重温第 1 章中建立一个简单的 HTML 文档的连接示例。将其直接转化为 C#，

其代码如下：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class HTMLConsole
  {
    static void Main(string[] args)
    {
      string htmlString;
      Console.Write("Please enter your name: ");
      string userName = Console.ReadLine();

      htmlString = "<HTML><HEAD></HEAD><BODY>";
      htmlString += "My name is: ";
      htmlString += userName;
      htmlString += "</BODY></HTML>";

      Console.WriteLine(htmlString);
    }
  }
}
```

这样的代码非常普通，因为它是以大多数人的思维方式进行编写的。它是一种线性的编程方法：从一个字符串开始，逐个添加每一部分直到字符串结束。然而，它并不是最佳方法。从我们对 String 类的了解来看，系统进行的步骤如下所示：

(1) 编译器在第一次分配中将字符串内置。

(2) 在下一行代码中，执行一个连接。给一个新的字符串分配内存，把内置的文本和新的文本复制到字符串中。

(3) 在接下来的代码行中，执行更多的连接，再次创建新的字符串，复制字符。

为了构造 HTML 页，需要从内置池中取得一个字符串，创建 3 个新的字符串。我们以自己对系统如何存储字符串的认识为基础，再来看一下这个例子，争取提高性能：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
```

```
class HTMLConsoleImproved
{
  static void Main(string[] args)
  {
    const string prefixString =
      "<HTML><HEAD></HEAD><BODY>My name is: ";
    const string suffixString = "</BODY></HTML>";

    Console.Write("Please enter your name: ");
    string userName = Console.ReadLine();

    string htmlString = prefixString + userName + suffixString;

    Console.WriteLine(htmlString);
  }
 }
}
```

因为有些文本是我们预先知道的，所以声明为常量。这就保证了它能够放在内置池中。在运行时，它可以不分配内存。这个方法从内置池获取了两个字符串，并给一个字符串分配了内存。内存也会被保存，因为常量文本会存储一次，而不是对每一个运行代码的线程多次分配内存。

基于一些性能测试，将一个新的字符串赋值一百万次，这种方法的赋值速度几乎是前面那种方法的 3 倍。

另外一种方法是提供一个格式字符串。稍后将详细讲解格式字符串，但是现在我们要了解的一个重点是，替换参数会替换{0}说明符：

```
String htmlString = String.Format(
    "<HTML><BODY>My name is: {0} </BODY></HTML>", userName);
```

这种方法看起来相当不错。没有显式的字符串连接，并且文本容易读取。遗憾的是，它的执行并不是非常理想。Format()方法是很耗费资源的。它对于简单的字符串连接来说并不是一个好的方法。后面的章节将探讨这个方法在什么时候是最适用的。

最后，使用 StringBuilder 类来执行连接：

```
StringBuilder MHTML = new StringBuilder();

MHTML.Append("<HTML><HEAD></HEAD><BODY>My name is: ");
MHTML.Append(userName);
MHTML.Append("</BODY></HTML>");
```

```
Console.WriteLine(MHTML.ToString());
```

在这个例子中，没有显式地创建字符串。StringBuilder 类负责全部的内存分配。这个方法执行得相当不错，但是所耗费的时间仍然是第二种方法的两倍。不过，那是因为 StringBuilder 对象的创建比字符串的创建花费的时间长。如果将所有这些方法重新执行一遍，进行 10,000 次循环，就会发现一些更有趣的事情。StringBuilder 所花费的时间是内置池连接的百分之一。部分原因是，内置在这里并没有发挥太大的作用；它依然要为每一次循环创建一个新的字符串对象。不过，如果没有内置，就会耗费更长的时间，这取决于有多少内存可用，因为每次循环将会创建 3 倍于此的字符串对象，并且垃圾收集将会频繁地中断执行。

还可以把 Capacity 传递给 StringBuilder 的构造函数。通过向 StringBuilder 类暗示大约需要多少内存，可以使性能得到提高。这样也许可以不用执行任何再分配了。当把这个方法用于相同的性能测试时，循环所花的时间又少了百分之一。其任务只是把项目移到内存中，速度只取决于内存、主板和处理器。

还有其他执行连接的方法，但是这里讨论的是最主要的方法。最佳方法是保证字符串中不发生变化的部分声明为常量。所以，总的来说，对于单一的连接，不要使用 StringBuilder，因为创建对象所耗费的系统开销会超出连接所带来的性能上的收益。对于单一的连接，可能的话，应该使用常量和其他的内置字符串，因为这样可以将性能提高 3 倍。

2.5.2　从字符串中提取子串

StringBuilder 类没有支持子串的方法，因此必须使用 String 类来提取子串。该方法有两个重载形式：可以只提供起始位置，也可以提供起始位置和长度。如果没有指定长度，就返回从起始位置开始到字符串结束之间的所有字符。下面的例子说明了它的用法：

```
string myString = "It's been a long time.";

// Displays "been a long time."
Console.WriteLine(myString.Substring(5));

// Displays "been"
Console.WriteLine(myString.Substring(5, 4));
```

2.5.3 比较字符串

您可能会认为字符串的比较是一件简单的事情，但是，这里需要作出几个选择。String 类有 4 种比较字符串的方法：Compare()、CompareTo()、CompareOrdinal()和 Equals()。

Compare()方法是 CompareTo()方法的静态版本。只要使用“=”运算符，就会调用 Equals()方法，所以 Equals()方法与“=”是等价的。CompareOrdinal()方法对两个字符串进行比较，而不考虑本地语言或文化；文化会在第 4 章中详细介绍：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class CompareTiming
  {
    static void Main(string[] args)
    {
      const int NUM_ITER = 10000000;
      int result;
      bool bResult;

      // Interned and strings match
      string s1 = "satisfaction";
      string s2 = "satisfaction";

      // Not interned and strings do not match
      //string s1;
      //string s2;
      //s1 = "satisfaction";
      //s2 = "dissatisfaction";

      int startTime;
      int endTime;
      int count;

      // Compare() method
      startTime = Environment.TickCount;
```

```
for (count = 0; count < NUM_ITER; count++)
{
   result = String.Compare(s1, s2);
}
endTime = Environment.TickCount;
Console.WriteLine("Compare: " + (endTime - startTime));

// CompareTo() method
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER; count++)
{
   result = s1.CompareTo(s2);
}
endTime = Environment.TickCount;
Console.WriteLine("CompareTo: " + (endTime - startTime));

// CompareOrdinal() method
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER; count++)
{
   result = String.CompareOrdinal(s1, s2);
}
endTime = Environment.TickCount;
Console.WriteLine("CompareOrdinal: " + (endTime - startTime));

// Equals() instance method
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER; count++)
{
   bResult = s1.Equals(s2);
}
endTime = Environment.TickCount;
Console.WriteLine("Equals (instance): " + (endTime - startTime));

// Equals() static method
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER; count++)
{
   bResult = String.Equals(s1, s2);
}
```

```
            endTime = Environment.TickCount;
            Console.WriteLine("Equals (static): " + (endTime-startTime));
        }
    }
}
```

检测的结果如表 2-4 所示。实际的值在各个环境甚至执行之间都会不同，但是，函数之间的相对差异应该是相同的。按表中的要求，稍微调整代码，使字符串既不相等也非内置。

表 2-4　方法检测的结果

方　法	字符串是否匹配	内置字符串数目	执行时间(ms)
Compare()	是	2	3665
CompareTo()	是	2	3676
CompareOrdinal()	是	2	941
Equals() –实例	是	2	1051
Equals() – 共享	是	2	11
Compare()	否	2	2894
CompareTo()	否	2	2905
CompareOrdinal()	否	2	420
Equals() –实例	否	2	421
Equals() – 共享	否	2	541
Compare()	是	0	2894
CompareTo()	是	0	2904
CompareOrdinal()	是	0	421
Equals() –实例	是	0	420
Equals() – 共享	是	0	531
Compare()	否	1	2884
CompareTo()	否	1	2904
CompareOrdinal()	否	1	431
Equals() – 实例	否	1	420
Equals() – 共享	否	1	531

可以看到，结果会因为所使用的方法和所比较的字符串而有很大的不同。

观察这些结果会发现一些很有意思的现象。比如说 Compare()方法，它一直都比 CompareTo()方法要快一些。由于 Compare()方法是静态的，所以调用它的系统开销要小于调用实例方法，因为它不用创建实例。

CompareOrdinal()方法要比 Compare() 方法和 CompareTo()方法快得多。因为CompareOrdinal()方法不用考虑文化方面的信息，它没有执行转换和提供国际支持所需要的系统开销，但是其他两种方法都有。这个方法可以用于那些不会被本地化的字符串，比如各种操作系统和网络变量。

最后，Equals()方法的性能会因所比较的字符串而有很大不同。为什么这样，共享的源代码实现给出了提示：

```
public static bool Equals(String a, String b)
{
  if ((Object)a == (Object)b) return true;
  if ((Object)a == null || (Object)b == null) return false;
  return a.Equals(b);
}
```

在比较两个对象时，若比较它们的散列码，这个比较将会执行得最好，它将是一个非常迅速的操作。如果两个字符串是相同的，它们也会具有相同的散列码，因此相等测试进行得非常迅速。如果它们并不相同，调用会向前推进，逐个字符进行比较。.NET Framework 中肯定存在更多的代码，因为当两个字符串是内置并相同时，静态方法会获得一个巨大的性能提高。这很可能是因为它们的引用是相同的，所以，如果对此进行检测，这个操作将会格外地迅速。

通常，Equals()的实例方法或多或少会快一点，相等运算符是所有运算操作中最慢的一个。一个例外情况是，两个字符串都是内置的，并相等，在这种情况下静态方法要快得多。当选择要使用的方法时要记得这一点。

另外两个有用的比较方法是 EndsWith() 和 StartsWith()。在共享的源代码段中，这些方法使用的是 Compare()函数；因此它们具有相似的性能。另外，如果重写了上面所用的代码来比较这些方法，只有在开始测定时间之前声明了所有的变量，比如字符串长度，才会有速度的差异：

```
public bool EndsWith(String value)
{
  if (null == value) throw new ArgumentNullException("value");

  int valueLen = value.Length;
  int thisLen = this.Length;

  if (valueLen > thisLen) return false;

  return (0==Compare(this, thisLen - valueLen, value, 0, valueLen));
}
```

```
public bool StartsWith(String value)
{
  if (null == value) throw new ArgumentNullException("value");

  if (this.Length < value.Length) return false;

  return (0 == Compare(this, 0, value, 0, value.Length));
}
```

可以假设使用 Compare()和使用 EndsWith()或 StartsWith()相比，它并没有什么优势。

2.5.4 格式化

第 1 章中我们讨论过，格式化用于根据一个值的内部机器表示来创建该值的最终用户的视图。格式化可以描述为任何数据类型到字符串的转换。格式化的优势在于程序员可以指定转换如何进行。

所有基本数据类型都实现 IConvertible 接口。这个接口有一个 ToString()方法，它必须由实现该接口的类实现。这个方法与 System.Object 中所提供的重载不同，但是它们都返回实现接口的类的字符串表示。由于所有的基本数据类型都实现这个接口，所以可以随时检索该数据类型的字符串表示。而且，数字、日期和枚举基本类型也实现了 System.IFormattable 接口，这个接口包含了 ToString()方法的一个附加重载，这一节会详细介绍它。

重载的定义如下：

```
public string ToString(string format, IFormatProvider provider)
```

第一个参数是格式说明符，第二个是格式提供者。如果缺少 format 字符串，就调用方法 System.IConvertible.ToString()，它仅包含了 IFormatProvider 参数，并且使用一个默认格式。下一节将详细介绍这两个参数。如果没有提供任何参数，System.Object.ToString()将使用一个默认的格式说明符和一个默认的格式提供者。.NET 中丰富的格式化意味着没有必要逐字构造字符串。

格式化分为数字格式化(包含货币)、日期-时间格式化、枚举格式化和自定义格式化。对于所有的类型都有很多格式化选项。本章只列出了一些常用的选项。查看.NET Framework SDK 的说明文档，其中可以看到包含格式化字符串的所有有效选项。

1. 格式化数字

第一个例子，我们将使用 System.Math 类中定义的 Pi 常量。格式以默认值、科学

记数和定点记数的顺序列出：

```
double val = Math.PI;

Console.WriteLine(val.ToString());          // displays 3.14159265358979
Console.WriteLine(val.ToString("E"));       // displays 3.141593E+000
Console.WriteLine(val.ToString("F3"));      // displays 3.142
```

默认格式提供了信息，但是它对用户不够友好。科学记数是一个选项，但这种情况下它也帮不上什么忙。指定要显示的小数个数会使这个值看起来更规范。另外，该值被四舍五入，而不是截短。

许多开发人员也对十六进制的格式感兴趣。最终用户是否愿意看到用十六进制表示的数字是不确定的，但可以肯定的是，如果记录错误或跟踪声明，就希望用十六进制记录它们：

```
int val = 65535;

Console.WriteLine(val.ToString("x"));       // displays ffff
Console.WriteLine(val.ToString("X"));       // displays FFFF
```

也可以把数字格式化为百分数：

```
Single val = 0.653F;

Console.WriteLine(val.ToString("p"));       // displays 65.30 %
Console.WriteLine(val.ToString("p1"));      // displays 65.3 %
```

默认格式化会在数字和百分号之间放入一个空格。这是可以定制的，方法是改变格式化提供者：

```
Single val = 0.653F;
object myObj = NumberFormatInfo.CurrentInfo.Clone() as NumberFormatInfo;
NumberFormatInfo myFormat = myObj as NumberFormatInfo;

myFormat.PercentPositivePattern = 1;

Console.WriteLine(val.ToString("p", myFormat));     // displays 65.30%
Console.WriteLine(val.ToString("p1", myFormat));    // displays 65.3%
```

首先要注意的是，通过复制一个现存的格式化结构可以创建一个新的格式化结构。此外，NumberFormatInfo 类是 System.Globalization 命名空间的一个成员，因此该命名空间必须导入到程序中。格式化百分数的模式通过设置 PercentPositivePattern 属性而改

变。接着，把新的格式化结构传递给 ToString()方法。定制更多格式的演示将在以后的例子中提供。

货币格式化是数字格式化的一部分。结果的不同取决于文化设置。例如，UK 的货币符号是£：

```
double val = 12345.89;

Console.WriteLine(val.ToString());          // displays 12345.89
Console.WriteLine(val.ToString("C"));       // displays $12,345.89 in US
```

格式化具有很大的灵活性。下面的例子演示了一个没有意义的货币结构：

```
double val = 1234567.89;
int []groupSize = {2, 1, 3};
object myObj = NumberFormatInfo.CurrentInfo.Clone();
NumberFormatInfo myCurrency = myObj as NumberFormatInfo;
myCurrency.CurrencySymbol = "@";
myCurrency.CurrencyDecimalSeparator = "/";
myCurrency.CurrencyGroupSeparator = "-";
myCurrency.CurrencyGroupSizes = groupSize;

// displays @1-234-5-67/89
Console.WriteLine(val.ToString("C", myCurrency));
```

2. 格式化日期

格式化日期与格式化数字的程序是相同的；只有格式字符串不同。

注意：

本节中的输出形式取决于用户计算机的文化设置。

```
DateTime dateValue = DateTime.Now;

// Displays a date like 9/18/2002 12:14:40 PM
Console.WriteLine(dateValue.ToString());

// Displays date like Wednesday, September 18, 2002 12:14 PM
Console.WriteLine(dateValue.ToString("f"));

// Displays September, 2002
Console.WriteLine(dateValue.ToString("y"));
```

```
// Displays 12:14:40 PM
Console.WriteLine(dateValue.ToString("T"));
```

同样，您可以完全控制日期字符串的格式化方式，如下面的例子：

```
DateTime dateValue = DateTime.Now;
string []myDays = {"RelaxDay", "ToughDay", "BlahDay", "LazyDay",
                   "ProductiveDay", "PartyDay", "HomeDay"};

object myObj = DateTimeFormatInfo.CurrentInfo.Clone();
DateTimeFormatInfo myDateTime = myObj as DateTimeFormatInfo;

myDateTime.DayNames = myDays;
myDateTime.FullDateTimePattern = "dddd, dd MMMM yyyy HH:mm:ss";

// Displays LazyDay, 18 September 2002 12:19:47
Console.WriteLine(dateValue.ToString("F", myDateTime));
```

这个例子说明了两件事。首先，如何覆盖每周的日期，提供自己定义的日期；第二，如何显式提供一个格式化字符串来控制日期字符串连接在一起的方法。

3. 格式化枚举

最后看看从枚举到字符串的转换。考虑如下的枚举声明：

```
enum Music
{
  Rock = 1,
  Blues = 2,
  Jazz = 3,
  Classical = 4
}
```

可以获取枚举的字符串信息，如下面的代码所示：

```
Music myMusic = Music.Blues;

Console.WriteLine(myMusic.ToString());          // displays Blues
Console.WriteLine(myMusic.ToString("d"));       // displays 2
```

用同样的方法从系统枚举中获取文本：

```
DayOfWeek day = DayOfWeek.Friday;

Console.WriteLine(String.Format("My favorite day is {0:G}", day));
```

控制台会显示下面的内容：

```
My favorite day is Friday
```

格式化字符串“G”把枚举显示为一个字符串。

4. 格式化选择

由于 StringBuilder 和 String 都提供了格式化功能，您可能想知道该使用哪一个。查看一下 String.Format()方法的静态源代码，会发现它创建了一个 StringBuilder 对象，并调用 AppendFormat()方法：

```
public static String Format(IFormatProvider provider, String format,
                            params Object[] args)
{
  if (format == null || args == null)
    throw new ArgumentNullException((format==null)?"format":"args");
  StringBuilder sb =
       new StringBuilder(format.Length + args.Length * 8);
  sb.AppendFormat(provider, format, args);
  return sb.ToString();
}
```

由于 Format()是一个静态方法，调用它不会耗费很多的系统开销。所以您要选择最有意义的，并在自己所编写的代码中可读性最强的方法。

第 4 章还会介绍一些关于格式化的问题。在处理国际化的字符串时，字符串格式化是非常有用的，但这里就不详细介绍了。

2.6 字符串的使用

我们已经介绍了大量的字符串操作。现在把一些操作联合起来，创建一些实用的应用程序。

2.6.1 建立字符串

我们已经了解了建立一个包含 HTML 的字符串所遇到的问题。现在利用 String 类和 StringBuilder 类再看看那个例子。该例子演示了数据库中的数据检索并把数据显示在 HTML 表中。这并不是在演示完成这项任务的最好方法；而是在说明该如何建立字符串。ASP.NET 提供了显示数据的更好方法，比如利用 DataGrid 控件。

明白了这一点，回忆一下第 1 章以及本章前面所阐述的方法，它的问题是要进行太多的内存分配和字符串复制。下面的代码利用 String 类和 StringBuilder 类解决了这个问题。本例被编写为一个控制台应用程序，但是替换它的 Console.ReadLine()代码行和 Console.WriteLine()代码行，就很容易把它改为 ASP.NET 或 Windows 窗体应用程序：

```
class HTMLTableConstruction
{
  static void Main(string[] args)
  {
    // Connection string is specific to your database environment.
    const string CONN_STRING =
        "user id=sa;password=;database=pubs;server=localhost";
    string SQLQuery;
    StringBuilder sb = new StringBuilder(256);
    string authorName = "";

    // Retrieve author name - Replace with Request(fieldName) for
    // ASP.NET application
    Console.Write("Please enter author's last name: ");
    authorName = Console.ReadLine();

    // Remove any spaces that might exist before or after the name.
    authorName = authorName.Trim();
    // Create the SQL query.
    SQLQuery = String.Format("SELECT * FROM Authors " +
        "WHERE au_lname LIKE '{0}%'", authorName);

    // Intern the table tags so as to increase performance
    const string OPENTABLE = "<table>";
    const string CLOSETABLE = "</table>";
    const string OPENTABLEROW = "<tr>";
    const string CLOSETABLEROW = "</tr>";
    const string OPENTABLECELL = "<td>";
    const string CLOSETABLECELL = "</td>";
    const string OPENTABLEHEADER = "<th>";
    const string CLOSETABLEHEADER = "</th>";

    SqlConnection conn = new SqlConnection(CONN_STRING);
    conn.Open();
    SqlCommand cmd = new SqlCommand(SQLQuery, conn);
```

```
        SqlDataReader dr = cmd.ExecuteReader();

        int numFields = dr.FieldCount;

        sb.Append(OPENTABLE + "\n");

        // Create the table headings.
        for (int count = 0; count < numFields-1; count++)
        {
           sb.Append(OPENTABLEHEADER + dr.GetName(count) +
                CLOSETABLEHEADER);
        }

        // Iterate through the rows and add the data to the HTML table.
        while (dr.Read())
        {
           sb.Append("\n" + OPENTABLEROW);

           for (int count = 0; count < numFields-1; count++)
           {
              sb.Append(OPENTABLECELL + dr[count].ToString() +
                   CLOSETABLECELL);
           }

           sb.Append(CLOSETABLEROW);
        }

        conn.Close();
        sb.Append("\n" + CLOSETABLE + "\n");
        Console.WriteLine(sb.ToString());
     }
  }
```

第一个值得注意的代码段是 SQL 查询语句：

```
        // Remove any spaces that might exist before or after the name.
        authorName = authorName.Trim();

        // Create the SQL query.
        SQLQuery = String.Format("SELECT * FROM Authors " +
            "WHERE au_lname LIKE '{0}%'", authorName);
```

Trim()方法用来清除前导空格或尾部空格，这在 Web 窗体应用程序中是非常基本的。这里有一个小的性能损失，因为要为修整后的字符串创建一个新的字符串。接下来，利用 Format()方法创建 SQL 语句。这里也可以利用内置字符串的连接代替 Format()方法。但如果需要格式化插入 SQL 语句的文本，就要使用 Format()方法代替连接方法。然后在向一个命令对象传递 SQL 语句(命令在连接上执行)之前，定义几个常量，以保证系统将它们内置(虽然它无论如何都会被内置)。

```
SqlCommand cmd = new SqlCommand(SQLQuery, conn);
SqlDataReader      dr = cmd.ExecuteReader();

int numFields = dr.FieldCount;

sb.Append(OPENTABLE + "\n");

// Create the table headings.
for (int count = 0; count < numFields - 1; count++)
{
  sb.Append(OPENTABLEHEADER + dr.GetName(count) +
      CLOSETABLEHEADER);
}
```

HTML 是在一个 StringBuilder 对象内部建立的。注意，声明它时，它的默认容量指定为 256 个字符。记住，如果没有提供容量，默认值就是 16。字符的总数要比这个值大，所以可以提供一个较接近结果大小的值，来最小化内存再分配的数量。

```
StringBuilder sb = new StringBuilder(256);
```

利用 Append()方法可以把 HTML 变成 StringBuilder 对象的一部分。第一个 Append()方法包含了一个固定字符串和字符。编译器将识别它并把它简化为一个字符串。第二个 Append()调用中的字符串在运行之前不能求值，所以在这里会有性能损失。一旦头信息到位，就可以在这些行中进行迭代了。

```
// Iterate through the rows and add the data to the HTML table.
while (dr.Read())
{
  sb.Append("\n" + OPENTABLEROW);
  for (int count = 0; count < numFields-1; count++)
  {
    sb.Append(OPENTABLECELL + dr[count].ToString() +
        CLOSETABLECELL);
  }
```

```
    sb.Append(CLOSETABLEROW);
}
```

这个循环包含了大部分的工作，因为它里面有一个嵌套循环。大多数情况下，StringBuilder 对象把字符串复制到为字符串分配的内存。当它到达容量的极限时，就需要再分配内存和复制。考虑到它每次都将容量加倍，这应该不成问题。

下面的例子说明了 AppendFormat()方法可以用来代替字符串连接：

```
// Iterate through the rows and add the data to the HTML table.
while (dr.Read())
{
  sb.Append("\n" + OPENTABLEROW);

  for (count = 0; count < numFields-1; count++)
  {
    sb.AppendFormat("<td>{0}</td> ", dr[count].ToString());
  }

  sb.Append(CLOSETABLEROW);
}
```

指定要显示的字段格式是一个很有效的方法。它还具有更好的可读性。然而，格式化的字符串处理起来要慢于连接：

```
conn.Close();
sb.Append("\n" + CLOSETABLE + "\n");
Console.WriteLine(sb.ToString());
```

最后，完成 HTML，在一个 ASP.NET 应用程序中，该字符串将发送到客户端。在这个应用程序中，它只是输出到控制台。

2.6.2 标记

标记(tokenizing)是从文本中提取具体内容的过程。编译器广泛使用标记，因为它们需要根据分隔符分离出语言的关键字。在自然语言处理中，为了删除句子中的单词，也会用到标记。看一看下面这个语句：

```
"I just want the words, not the punctuation, and not the spaces, from this sentence."
```

一种方法是逐个字符地处理该语句。这种方式非常冗长且容易出错。虽然 Java 有

一个 StringTokenizer 类可以简化这个过程，但.NET Framework 中没有提供这样的类，所以用户必须编写更多的代码。下面的这些代码从句子中提取单词，并把它们输出到控制台。

```
class Tokenizing
{
  static void Main(string[] args)
  {
    string myString = "I just want the words, not the punctuation," +
                      " and not the spaces, from this sentence.";
    char [] separators = {' ', ',', '.', ':', ';', '?', '!'};

    int startPos = 0;
    int endPos = 0;

    do
    {
      endPos = myString.IndexOfAny(separators, startPos);

      if (endPos == -1) endPos = myString.Length;

      if (endPos != startPos)
          Console.WriteLine(myString.Substring(
                                  startPos, (endPos - startPos)));

      startPos = (endPos + 1);
    } while (startPos < myString.Length);
  }
}
```

在控制台中会看到下面这些输出内容：

```
I
just
want
the
words
not
the
punctuation
and
```

```
not
the
spaces
from
this
sentence
```

首先，声明变量。标识为记号分隔符的字符存储到一个数组中。声明每一个记号的开头和结尾位置的变量：

```
string myString = "I just want the words, not the punctuation," +
                  " and not the spaces, from this sentence.";
char [] separators = {' ', ',', '.', ':', ';', '?', '!'};

int startPos = 0;

int endPos = 0;
```

IndexOfAny()方法是这个标记程序中的关键部分。假设给定一个分隔符的列表和一个起始位置，它会找到含有一个分隔符的下一个字符的位置。如果没有发现分隔符，该方法就返回－1。这时，结束位置就设置为字符串的结尾：

```
do
{
  endPos = myString.IndexOfAny(separators, startPos);

  if (endPos == -1) endPos = myString.Length;
```

利用 Substring()方法将标记从字符串中提取出来并写入控制台。在搜索下一个标记之前，起始位置更新为结束位置的下一个字符：

```
  if (endPos != startPos)
      Console.WriteLine(myString.Substring(
                        startPos, (endPos - startPos)));

  startPos = (endPos + 1);
} while (startPos < myString.Length);
```

大部分例程都是很有效率的，因为是按线性操作字符串中的字符。开销最大的是 Substring()操作，它创建了一个新的字符串。

2.6.3　颠倒字符串次序

有时需要颠倒一个字符串的次序。有些语言把这作为内置的功能，但是.NET Framework 没有。下面这个例子展示了一个颠倒字符串次序的好方法：

```
class Reverse
{
  static void Main(string[] args)
  {
    string myString = "If there's a bustle in your hedgerow";
    char [] myChars = myString.ToCharArray();
    Array.Reverse(myChars);
    Console.WriteLine(myString);
    Console.WriteLine(myChars);
  }
}
```

代码会在控制台中显示下列内容：

```
If there's a bustle in your hedgerow
woregdeh ruoy ni eltsub a s'ereht fI
```

任何继承于 Array 的类都能利用 Reverse()方法为数组中的元素重新排序。要颠倒这个字符串，应从字符串中创建一个字符数组并调用 Reverse()方法。结果返回的数组是颠倒字符次序后的字符串。

不过，您会在第 4 章中发现，一个字符可以由一对 Char 组成。颠倒了它们两个的次序会造成这个字符无效。

2.6.4　插入、删除和替换

字符串有一类操作是插入、删除和替换字符串的一部分。String 类和 StringBuilder 类都支持这些操作。其中大部分操作都可以利用正则表达式完成。第 5～7 章介绍正则表达式。

对于这些例子，我们把下面这个示例文件(albums.txt)作为字符串的来源。下面的代码以 Unicode 格式读取文件。确保文本文件被保存为读取时的格式。例如记事本允许将代码保存为 Unicode：

```
Led Zeppelin, Led Zeppelin [I], 1969
Frank Zappa, We're Only in It for the Money, 1968
```

```
Beatles, Sgt. Pepper's Lonely Hearts Club Band, 1967
Jimi Hendrix, Are You Experienced?, 1967
```

下面的代码是加载数据并处理数据的测试工具。测试结果被发送给控制台：

```
class ProcessFile
{
  static void Main(string[] args)
  {
    const string NAME = "album.txt";
    Stream readLine;
    TextWriter writeLine;
    StringBuilder sb;
    readLine = File.OpenRead(NAME);
    writeLine = Console.Out;

    StreamReader readLineSReader =
        new StreamReader(readLine, Encoding.Unicode);

    readLineSReader.BaseStream.Seek(0, SeekOrigin.Begin);

    while (readLineSReader.Peek() > -1)
    {
      sb = new StringBuilder(readLineSReader.ReadLine());
      // insert string operation here
      Console.WriteLine(sb.ToString());
    }
  }
}
```

如果利用下面的代码在结尾再添加一列内容：

```
sb.Append(", Rock");
```

那么在控制台将显示下列内容：

```
Led Zeppelin, Led Zeppelin [I], 1969, Rock
Frank Zappa, We're Only in It for the Money, 1968, Rock
Beatles, Sgt. Pepper's Lonely Hearts Club Band, 1967, Rock
Jimi Hendrix, Are You Experienced?, 1967, Rock
```

第一列可以使用下面的代码删除：

```
sb.Remove(0, sb.ToString().IndexOf(',') + 1);
```

输出的结果为(注意在每行的开头仍有一个空格)：

```
 Led Zeppelin [I], 1969
We're Only in It for the Money, 1968
Sgt. Pepper's Lonely Hearts Club Band, 1967
Are You Experienced?, 1967
```

分隔符可以用下面的代码替换：

```
sb.Replace(",", " |");
```

这将会输出：

```
Led Zeppelin | Led Zeppelin [I] | 1969
Frank Zappa | We're Only in It for the Money | 1968
Beatles | Sgt. Pepper's Lonely Hearts Club Band | 1967
Jimi Hendrix | Are You Experienced? | 1967
```

这种情况下，会产生一个性能损失。为了使输出结果更加悦目，我们用空格和竖杠替换了逗号。这就是说系统不能完成字符对字符的简单替换。必须移动其他的字符，并分配更多的内存。

最后，可以利用下面的代码在每一行前面加上行号。这是以 m_lineNumber 已经在前面的某个地方声明并初始化为 0 作为前提的。

```
sb.Insert(0, lineNumber.ToString("000 "));
lineNumber++;
```

这样就得到了下面的输出结果：

```
000 Led Zeppelin, Led Zeppelin [I], 1969
001 Frank Zappa, We're Only in It for the Money, 1968
002 Beatles, Sgt. Pepper's Lonely Hearts Club Band, 1967
003 Jimi Hendrix, Are You Experienced?, 1967
```

在 String 类和 StringBuilder 类之间进行选择

对于插入、删除以及替换操作，可以在 String 类和 StringBuilder 类之间进行选择。表 2-5 显示了代码在 1.3 GHz Intel Celeron 处理器和 256MB 内存下执行时，这些操作的性能。

表 2-5　执行插入、删除和替换操作时使用 String 和 String Builder 类的性能比较

类	方　法	字符串是否内置	执行时间(ms)
String	Insert()/Remove()	是	691
String	Insert()/Remove()	否	851

(续表)

类	方 法	字符串是否内置	执行时间(ms)
StringBuilder	Insert()/Remove()	N/A	441
String	Replace()	是	981
String	Replace()	否	982
StringBuilder	Replace()	N/A	761

通过运行这些测试，可以看到进行 Replace()、Remove()和 Insert()操作时，StringBuilder 一直都比 String 迅速。如果使用 String 类，对一个内置字符串的 Insert()和 Replace()操作会比对一个非内置的字符串操作迅速，但是并不会带来显著的差异。

StringBuilder 类比 String 类执行得更好，主要原因在于，String 必须分配一个新的字符串并复制字符。StringBuilder 可以在相同的内存空间中执行这些操作。只有在 StringBuilder 达到容量极限，需要分配更多的内存时，性能才会受到影响。不过它要比 String 受到的影响小，而且通常它对内存的再分配要远远少于 String。和对连接进行的性能测试一样，只需知道创建一个 StringBuilder 会花费较长的时间。对下面的代码进行相似的测试，并且是在相同的机器上，使用相同数量的迭代。对一个 114 个字符的字符串来说，创建一个 StringBuilder 只多花费了 790ms。耗时多少取决于字符串的大小。

下面的代码用于获取前面表 2-5 的结果：

```
using System;
using System.Text;

namespace Wrox.Text.Chapter2
{
  class PerfTest
  {
    const int NUM_ITER = 1000000;

    static void Main(string[] args)
    {
      string s1 = "It's been a long time since I rock and rolled.";
      string s2;
      string s3;
      StringBuilder sb = new StringBuilder(s1);

      // Make sure we have a string that's not interned.
      s2 = s1.Replace('.', '!');

      int startTime;
      int endTime;
```

```
int count;

// INSERT/REMOVE - String, interned.
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER-1; count++)
{
   s3 = s1.Insert(12, "really ");
   s1 = s3.Remove(12, 7);
}
endTime = Environment.TickCount;
Console.WriteLine("INSERT/REMOVE - interned String: " +
      (endTime - startTime));

// INSERT - String, not interned.
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER-1; count++)
{
   s3 = s2.Insert(12, "really ");
   s1 = s3.Remove(12, 7);
}
endTime = Environment.TickCount;
Console.WriteLine("INSERT/REMOVE - non interned String: " +
      (endTime - startTime));

// INSERT - StringBuilder
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER-1; count++)
{
   sb.Insert(12, "really ", 1);
   sb.Remove(12, 7);
}
endTime = Environment.TickCount;
Console.WriteLine("INSERT/REMOVE - String Builder: " +
      (endTime - startTime));

// REPLACE - String, interned.
startTime = Environment.TickCount;
for (count = 0; count < NUM_ITER-1; count++)
{
   s3 = s1.Replace("rock and rolled", "had the blues");
```

```
        }
        endTime = Environment.TickCount;
        Console.WriteLine("REPLACE - interned String: " +
            (endTime - startTime));

        // REPLACE - String, not interned.
        startTime = Environment.TickCount;
        for (count = 0; count < NUM_ITER-1; count++)
        {
          s3 = s2.Replace("rock and rolled", "had the blues");
        }
        endTime = Environment.TickCount;
        Console.WriteLine("REPLACE - non interned String: " +
            (endTime - startTime));

        // REPLACE - StringBuilder
        startTime = Environment.TickCount;
        for (count = 0; count < NUM_ITER-1; count++)
        {
          sb.Replace("rock and rolled", "had the blues");
        }
        endTime = Environment.TickCount;
        Console.WriteLine("REPLACE - String Builder: " +
            (endTime - startTime));
      }
    }
  }
```

2.7　小结

本章详细介绍了利用 String 类和 StringBuilder 类对文本的处理。面对文本操作，通常有大堆的方法可以选择。本章讲述了 String 类和 StringBuilder 类的操作原理，以指导您做出最好的选择。

下面推荐几个优化字符串操作性能的技巧。

- 尽可能把字符串存储为常量，这会保证使用内置池，同时最小化所需机器指令的数量。
- 如果 String 类能够有效工作，就不要使用 StringBuilder。例如一个单独字符串的

赋值操作。

- 如果要循环建立一个庞大的字符数据块，就使用 StringBuilder。
- 如果需要国际化的字符串，就只能使用方法 Compare()。否则，使用 CompareOrdinal() 方法。
- 如果只需知道字符串是否相同，就应使用方法 Equals()，而不是 CompareOrdinal() 方法。
- 通常情况下使用方法 Equals()，而不是"="运算符。

本章没有对国际化进行详细讲解，它也对构建字符串处理方法有一定影响。解决本章所列出问题的另一个方法是使用正则表达式。C#中的国际化会在第 4 章中介绍，而正则表达式会放在 5～7 章中介绍。下一章介绍在数据类型之间移动字符串和将字符串存储到数组和集合时所遇到的问题。

第3章　字符串转换

编写代码时，常常需要转换信息的数据类型。例如，需要将用户输入的日期由字符串类型转换为实际日期表示法，以便进行计算；或者需要将一个数字转换为字符串，以便显示为两个小数位和一个货币符号的格式。.NET Framework 提供了完成这些任务的简便方法。

3.1　ToString()方法

继承于 System.Object 基类的每一个对象都包含一个 ToString()方法，这个方法提供了一个表示对象的字符串。这包括系统基本类型，如 Integer 或 Double 也都实现了 ToString()方法，因为它们也是对象。ToString()方法的默认实现返回的是对象派生类的实例类型的完全限定名。例如，下面的代码将返回一个含有“System.Object”的字符串，如图 3-1 所示。

```
Object obj = new Object();
MessageBox.Show(obj.ToString());
```

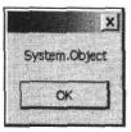


图　3-1

然而，这个方法可以重写，并且只要重写有意义，就应该重写，使之返回可读的、区分文化的字符串，代表当前对象。例如，int 类就是用这种方式实现了 ToString()方法。下面的代码提供了一个整型值的有效字符串表示，如图 3-2 所示。

```
int i = 123456;
MessageBox.Show(i.ToString());
```

图　3-2

同时需要注意，对所返回字符串的格式化需要更多控制的派生类应该实现 IFormattable 接口，对于 ToString()方法，这个接口使用的是线程的当前文化。下一章将介绍有关文化的更多内容，文化会指定将要使用的特定区域的信息。例如，对于 Double 类的实例来说，假设它的值为 0，在它的 ToString()方法调用后，可能会返回 0.00 或 0,00，到底返回哪种类型取决于当前文化。

知道了可以利用 ToString()方法把对象转换为字符串后，下面看看如何把数值转换为字符串。

3.2　把数值表示为字符串

先回顾一下.NET Framework 所支持的数值格式，以及数值数据，包括百分数和货币数量，是如何格式化并表示为字符串的。在探讨 NumberFormatInfo 类之前，先看一下.NET Framework 提供的一些有趣的数值特性。下面是一个把数字表示为字符串的最简单的方法：

```
12345.ToString();
```

它将把 12345 作为一个字符串输出。虽然这个方法很有用，但是格式化(或者说在此非常缺少格式化)并没有给人留下深刻印象。这是因为我们没有为字符串指明一个确切的格式，所以 ToString()方法就会使用默认的标准数字格式，没有小数分隔符说明，也没有分组。

在此，可以更进一步指定一些格式信息，比如：

```
12345.ToString("n");
```

这个例子中，我们提供了格式信息(值“n”)，以说明在把 12345 表示为一个字符串时，我们希望把它格式化为一个数字。格式化将会在线程级(thread level)考虑当前所选择的 CurrentCulture(文化的具体内容请参考第 4 章)，以提供表示这个数字的字符串。

如果当前文化是 en-US，会得到下面的结果：

```
12,345.00
```

如果当前文化是 fr-FR，其结果就会变成：

```
12 345,00
```

可以利用 String.Format()方法格式化数值结果，对于控制台应用程序，可以使用 Console.Write()方法，该方法会调用 String.Format()方法。格式由格式字符串指定。

下面的代码是利用货币格式(使用 ToString()时由“C”说明符选定)显示整数的实际值。由于作者位于美国，机器的配置也是 en-US 标准，所以用美元符号显示这个值，

其结果如图 3-3 所示。

```
using System;
using System.Windows.Forms;

namespace Wrox.Text.Chapter3
{
  class ToStringCulture
  {
    static void Main(string[] args)
    {
      int MyInt = 100;
      string MyString = MyInt.ToString("C");
      MessageBox.Show(MyString);
    }
  }
}
```

图 3-3

如果是在 UK，并且机器是标准的 English 配置，输入这些代码并运行，其输出将会是英镑的格式，如图 3-4 所示。

图 3-4

会发生这种情况是因为，如果没有把文化信息传递给 ToString()方法，应用程序就会使用在当前线程级设定的当前文化。

下面的代码是利用 CultureInfo 类指定 ToString()方法和格式字符串将使用的文化。这段代码创建了一个 CultureInfo 类的新实例 MyCulture，并利用字符串 fr-FR 把它初始化为法国文化。然后这个对象作为附加参数和前面例子用过的 C 字符串格式说明符一起传递给 ToString()方法，以生成一个法国货币值。

```
using System;
```

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

namespace Wrox.Text.Chapter3
{
  class ToStringCulture2
  {
    static void Main(string[] args)
    {
      int MyInt = 100;
      CultureInfo MyCulture = new CultureInfo("fr-FR");
      string MyString = MyInt.ToString("C", MyCulture);
      MessageBox.Show(MyString);
    }
  }
}
```

运行这段代码会得到如图 3-5 所示的结果。

图 3-5

把文化考虑在内是很有意思的。当前 CultureInfo 对象实现了 IFormatProvider 接口，并拥有一个成员 NumberFormatInfo，这个成员实际负责提供信息，如分组符号、组的实际大小、小数分隔符和其他信息。实际上，NumberFormatInfo 类实现 IFormatProvider 接口也是为应用程序提供格式化信息。

NumberFormatInfo 类

NumberFormatInfo 类定义了如何依据不同的文化，对数值进行格式化并显示。这个类包含了像货币、小数分隔符和其他数值符号这样的信息。

要为一个特定的文化创建 NumberFormatInfo，先要给它创建一个 CultureInfo 对象并检索其 CultureInfo.NumberFormat 属性。如果想为当前线程下的文化创建一个 NumberFormatInfo，就要利用 CurrentCultureInfo 属性。

要为一个不变的文化创建 NumberFormatInfo，对于只读类型，使用 InvariantInfo 属性，对于可写类型，使用 NumberFormatInfo 构造函数。注意，为一个中立文化(只处理

语言，没有区域/国家信息的文化)创建 NumberFormatInfo 是不可能的，因为没有指明与中立文化相关的格式化信息。

数值是利用存储在 NumberFormatInfo 属性中的标准模式或自定义模式格式化的。为了修改一个值的显示方式，NumberFormatInfo 必须是可写的，只有这样，自定义模式才能保存到它的属性中。由于与 CultureInfo 关联的默认 NumberFormatInfo 对象作为这个类的一个成员是不可写的，所以需要复制这个对象，用 Clone()方法创建一个可以修改的实例。

指定字符串格式的形式是 Ann，其中 A 是格式说明符，nn 是精度说明符。格式说明符控制应用于数值的格式化类型，而精度说明符控制有意义的数字或小数位的数目。

如果没有指定字符串格式，或设为 Nothing 或一个空串，就隐式地使用通用格式说明符(“G”)。

表 3-1 列出了所有标准的格式说明符和它们的用法示例。

表 3-1 格式说明符和用法示例

格式字符	格式说明	举例	输出
C 或 c	货币记数	1000.ToString("C"); (-1000).ToString("C");	$1,000.00 ($1,000.00)
D 或 d	小数记数	1000.ToString("D");	1000
E 或 e	科学记数(幂)	100000.ToString("E");	1.000000E+005
F 或 f	定点记数	1000.ToString("F4"); 1000.ToString("F0");	1000.0000 1000
G 或 g	通用记数	1000.ToString("G");	1000
N 或 n	数字形式	1000.ToString("N");	1,000.00
X 或 x	十六进制形式	1000.ToString("X"); 1000.ToString("x"); 0x1000.ToString("x");	3E8 3e8 1000
P 或 p	百分数	1.ToString("P");	100.00 %
r 或 R	往返。它可以保证字符串表示能转换回数字形式。它适用于 Double 类型而不适用于 Integer 类型	Double d = 1000.0; d.ToString("R");	1000

注意，格式字符不区分大小写，除了“x”和“X”，这种情况下格式字符的大小写会决定十六进制数字使用的大小写。上表中显示的格式化信息仅适用于 en-US 文化。如果使用另一种文化，其输出就会有所不同。

NumberFormatInfo 类实现了 ICloneable 接口，所以可以利用 Clone()方法复制

NumbeFormatInfo 对象。当为了提供自定义格式化，而打算自定义一个与 CultureInfo 关联的 NumberFormatInfo 类的实例时，这种方法十分有效。下面这个例子就是一个很好的证明。这个小程序检索当前文化并对它进行自定义，于是就有了新的货币符号“*”和新的组分隔符“|”。然后它以这种自定义的格式输出值 123456。

对于 CurrencyPositivePattern 需要一些解释。.NET Framework 使用一组由数字(实际上是枚举量)引用的模式，它说明了货币符号的位置(在字符串前面还是结尾)以及是否需要在货币符号前面插入空格。

```
using System;
using System.Globalization;
using System.Threading;
namespace Wrox.Text.Chapter3
{
  class ChangeCultureFormat
  {
    static void Main(string[] args)
    {
      // retrieve current culture information by cloning it so that
      // we can manipulate its NumberFormat member

      CultureInfo ci =
          (CultureInfo)Thread.CurrentThread.CurrentCulture.Clone();
      NumberFormatInfo nfi = ci.NumberFormat;

      // Set our format to "123|456*"
      nfi.CurrencyPositivePattern = 1;
      nfi.CurrencyGroupSeparator = "|";
      nfi.CurrencySymbol = "*";
      nfi.CurrencyDecimalDigits = 0;
      ci.NumberFormat = nfi;

      // Set the thread culture to our modified CultureInfo object
      Thread.CurrentThread.CurrentCulture = ci;

      // Display the actual value using the proper formatting
      Console.WriteLine(123456.ToString("C"));
    }
  }
}
```

执行这段代码将会输出如下结果：

```
123|456*
```

3.3 把日期和时间表示为字符串

DateTime 值类型表示日期和时间，其范围是从公元 1 年 1 月 1 日午夜 12:00:00(或者C.E.-通用纪元，"AD"的一种新的跨文化名称)到C.E. 9999年12月31日晚上11:59:59。

DateTime 值类型实现了 IFormattable 接口，允许用一个重载的 DateTime.ToString() 方法把它格式化为一个字符串。在.NET Framework 中，用来格式化 DateTime 对象的标准格式提供者类是 DateTimeFormatInfo。

DateTimeFormatInfo 类

这个类定义了 DateTime 值该如何依据不同的文化进行格式化并显示。它包含了日期模式、时间模式以及 AM/PM 指示符的信息。要为一个特定文化创建 DateTimeFormatInfo，先为它创建一个 CultureInfo，并检索其 CultureInfo.DateTimeFormat 属性。要为不变的文化创建一个 DateTimeFormatInfo，对于只读类型，可以使用 InvariantInfo 属性，对于可写类型，可以使用 DateTimeFormatInfo 构造函数。像 NumberFormatInfo 类一样，这个类也不可能为中立文化创建 DateTimeFormatInfo。

格式字符串如果只包含下表列出的单一格式说明符中的一种，就会解释为标准格式说明符。如果指定的格式字符是单个字符，并且不在下表中，就会有一个异常抛出。如果格式字符串比单个字符长(即便长出的部分是空白)，格式字符串就解释为自定义格式字符串。由这些格式说明符生成的模式会受到 Windows 控制面板中 Regional Options 设置的影响，因为这些设置影响了当前文化，除非用户的预设参数被覆盖。

与当前文化的 DateTimeFormat 属性关联的 DateSeparator 和 TimeSeparator 字符，定义了格式字符串显示的日期和时间分隔符。

然而，在使用 InvariantCulture 的情况下(被 r、s 和 u 说明符引用时)，与 DateSeparator 和 TimeSeparator 字符关联的字符就不会随着当前文化的改变而改变，依然附属于不变的文化。

.NET Framework 提供了支持日期和时间的许多格式(完整的列表可以访问 MSDN)。这里是一些最普通的格式，它们也被映射到 DateTimeFormatInfo 类的成员函数(与一些针对 en-US 文化的示例数据一起)：

- d：短日期模式(例如 5/20/2002)，也可以直接通过 DateTimeFormatInfo.ShortDate

Pattern 属性获得。

- D：长日期模式(例如，Monday, May 20, 2002)，也可以直接通过 DateTimeFormatInfo.LongDatePattern 属性获得。
- t：短时间模式(例如 3:51 PM)，也可以直接通过 DateTimeFormatInfo.ShortTimePattern 属性获得。
- T：长时间模式(例如 3:51:04 PM)，也可以直接通过 DateTimeFormatInfo.LongTimePattern 属性获得。

当然，除了标准格式，也支持自定义格式。下面的代码就演示了 DateTime 对象使用自定义格式字符串的方法：

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

namespace Wrox.Text.Chapter3
{
  class CustomDateTimeFormat
  {
    static void Main(string[] args)
    {
      DateTime dt = DateTime.Now;
      DateTimeFormatInfo dfi = new DateTimeFormatInfo();
      CultureInfo ci = new CultureInfo("nl-BE");

      // Use the DateTimeFormat from the culture associated with
      // the current thread.
      Console.WriteLine(dt.ToString("d"));
      Console.WriteLine(dt.ToString("m"));

      // Use the DateTimeFormat from the specific culture passed.
      Console.WriteLine(dt.ToString("F", ci));

      // Make up a new custom DateTime pattern, and use it
      dfi.MonthDayPattern = "MM-MMMM, ddd-dddd";
      Console.WriteLine(dt.ToString("m", dfi));

      // Reset the current thread to a different culture.
      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-BE");
      Console.WriteLine(dt.ToString("d"));
```

```
    }
  }
}
```

输出结果和下面给出的结果类似，这是由当前日期、时间和文化决定的：

```
10/7/2002
October 07
maandag 7 oktober 2002 11:19:42
10-October, Mon-Monday
7/10/2002
```

3.4 把其他对象表示为字符串

前面已经处理了把数字、日期以及时间表示为字符串的问题，但是，如果我们想创建一个类，并保证这个类也正确表示为一个字符串，该如何做呢？其实，这只是稍微复杂了一点，这个类将会从 Object 类继承，所以只需为类中的 ToString 方法提供重载就可以了。

现在创建一个含有 X 值和 Y 值的 Coordinate 类：

```
class Coordinate : Object
{
  private int mx;
  private int my;

  public Coordinate(int X, int Y)
  {
    mx = X;
    my = Y;
  }

  public override String ToString()
  {
    return "(" + mx.ToString() + ", " + my.ToString() + ")";
  }
}
```

这是一种相当简单和原始的方法。更为缜密的方法是启用格式化参数。例如，支持适用于数值格式的不同格式化选项，就能够把坐标表示为十六进制数字。

为此，只需重载一个实现了 IFormattable 接口的 ToString()方法，并且改变当前的 ToString()重载形式，以此实现格式化支持。这些工作是由下面的代码完成的：

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

namespace Wrox.Text.Chapter3
{
  class Coordinate : Object, IFormattable
  {
    private int mx;
    private int my;

    public Coordinate(int X, int Y)
    {
      mx = X;
      my = Y;
    }

    public override String ToString()
    {
      return ToString("g", CultureInfo.CurrentCulture);
    }

    public String ToString(String format, IFormatProvider fp)
    {
      // Implements IFormattable.ToString
      return String.Format("({0}, {1})",
                           mx.ToString(format, fp),
                           my.ToString(format, fp));
    }
  }

  class CoordinateExample
  {
    static void Main(string[] args)
    {
      Coordinate c = new Coordinate(1000, 2000);
      Console.WriteLine(c.ToString());
```

```
            Console.WriteLine(c.ToString("X", null));
            Console.WriteLine(c.ToString("N", new CultureInfo("en-GB")));
        }
    }
}
```

默认的 ToString()方法使用的是当前文化和通用数字格式 g。

这个应用程序的输出为：

```
(1000, 2000)
(3E8, 7D0)
(1,000.00, 2,000.00)
```

3.5 用字符串表示字符串

很多情况下需要把字符串转换为另一种字符串。这听起来可能很奇怪，下面先来看看为什么会出现这种情况。

假设有这么一段代码：

```
using System;

namespace Wrox.Text.Chapter3
{
  class StringConcatenation
  {
    static void Main(string[] args)
    {
      string[] list = {"car", "pen"};
      string message;

      foreach (string item in list)
      {
        message = "This is my new " + item;
        Console.WriteLine(message);
      }
    }
  }
}
```

这段代码的输出是：

```
This is my new car
This is my new pen
```

这看起来相当不错。然而，在很多情况下，当处理国际化字符串时，代码就会出错。这是因为代码认为所有的语言都应该像英语那样进行处理。实际上，许多语言都是以不同的方式进行处理的。

比如日语，它使用了不同的语序，动词放在句子的末尾。另外，在其他语言中，例如法语和意大利语，名词有词性之分(阳性或阴性)，形容词的变化依赖于所涉及的词性和名词数量的变化。

现在您就会明白为什么上面的代码不能保证始终正确运行。上面演示的这个连接的例子不是很好，我们需要一个更好的机制。方法 String.Format()正好提供了这个机制，下一章会对此进行介绍。

3.6　把字符串转换为其他类型

前面讨论了把一个数据由数值或日期时间对象格式化为字符串的方法。不过，有时也需要执行反向转换，就是把字符串转换成一个 DateTime 或数字。这正是 Parse()方法和 ParseExact()方法要完成的工作。

3.6.1　把字符串转换成数字

所有的数值类型(比如 Integer 和 Double)都提供 Parse()方法，它允许实现字符串到数字的转换，并考虑已经应用于字符串表示的格式设置。Parse()方法接受一个格式提供者，它允许指定并分析特定于文化的字符串。这就是说，也可以应用用来进行格式化的自定义设置。

如果没有指定格式提供者，就使用与当前线程相关联的提供者。如果想指定一个格式，必须要保证传递给 Parse()方法的 NumberStyles 枚举标志(由命名空间 System.Globalization 提供)能够真正地匹配将使用的格式，否则，就会抛出一个 FormatException 异常。例如，在 en-US 文化下，如果没有传递 NumberStyles.AllowThousands 枚举，那么包含逗号的字符串就不能利用 Parse()方法转换为整型值。

下面的代码是无效的，它会引发一个异常。下面的代码在分析包含非数值字符的字符串时，使用的是一种不正确的方法：

```
using System;
using System.Globalization;

namespace Wrox.Text.Chapter3
{
  class BadParsing
  {
    static void Main(string[] args)
    {
      CultureInfo myCultureInfo = new CultureInfo("en-US");
      string myString = "1,000";

      try
      {
        int myInt = int.Parse(myString, myCultureInfo);
        Console.WriteLine(myInt);
        // raise exception
      }

      catch(System.FormatException ex)
      {
        Console.WriteLine(
            "Commas cannot be included in string to convert");
        Console.WriteLine("Exception was: {0}", ex.ToString());
      }
    }
  }
}
```

只需稍微修改代码，在分析字符串时指定 NumberStyles.AllowThousand 标志，就可以保证代码的正常运行：

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

namespace Wrox.Text.Chapter3
{
  class GoodParsing
  {
    static void Main(string[] args)
```

```
    {
      CultureInfo myCultureInfo = new CultureInfo("en-US");
      string myString = "1,000";
      int myInt = int.Parse(myString, NumberStyles.AllowThousands,
                            myCultureInfo);
      Console.WriteLine(myInt);
    }
  }
}
```

也可以利用自定义格式进行分析。为了说明这一点，下面把利用自定义格式演示如何格式化数字时使用的那个例子扩展一下，给它加上分析功能。

这个小程序检索当前文化并对它进行自定义，把货币符号设置为“#”，组分隔符设置为“:”。程序将会输出一个整数(2364)，它是以一个插入这些字符的字符串为基础的。当处理一个能够从其他文化背景下的用户那里接收输入的 HTML 表单时，这种方法是极有价值的。

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

namespace Wrox.Text.Chapter3
{
  class CustomizeCulture
  {
    static void Main(string[] args)
    {
      // Retrieve current culture information by cloning it
      // so that we can manipulate its NumberFormat member
      CultureInfo ci =
      (CultureInfo)Thread.CurrentThread.CurrentCulture.Clone();
      NumberFormatInfo nfi = ci.NumberFormat;

      // Set our format to "2:364#"
      nfi.CurrencyPositivePattern = 1;
      nfi.CurrencyGroupSeparator = ":";
      nfi.CurrencySymbol = "#";
      nfi.CurrencyDecimalDigits = 0;
      ci.NumberFormat = nfi;
      // Set the thread's culture to our modified CultureInfo object
```

```
        Thread.CurrentThread.CurrentCulture = ci;

        // Display the actual value using the proper formatting
        Console.WriteLine(2364.ToString("C"));

        // Now parse a string using our custom format
        int myInt = int.Parse("2:364#",
                              NumberStyles.AllowCurrencySymbol |
                              NumberStyles.Currency |
                              NumberStyles.AllowThousands);
        Console.WriteLine("{0:N}", myInt);
      }
    }
}
```

调用 Parse()方法时，Currency、AllowCurrencySymbol 和 AllowThousands NumberStyles 标志全部被设置，这是为了匹配利用当前线程的文化指定的格式。

此时程序的输出为：

```
2:364#
2,364.00
```

3.6.2 把字符串转换为日期和时间

本节将讲述把字符串转换为日期和时间的技巧，因为日期在不同的区域有不同的格式。

分析日期和时间

DateTime 对象既包含 Parse()方法也包含 ParseExact()方法，两者都可以用来将日期和时间的字符串表示转换为 DateTime 对象。Parse()方法和 ParseExact()方法的不同之处在于：Parse()方法可以转换任何有效的字符串表示，而 ParseExact()方法只转换那些匹配所指定格式的字符串。如果使用的是标准的 DateTime 模式(在 DateTime 格式表中定义过的)，那么这两种方法都可以使用。

Parse()

下面的代码演示了如何利用 Parse()方法分析一个包含日期的字符串：

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

```
using System.Threading;
namespace Wrox.Text.Chapter3
{
  class Parse
  {
    static void Main(string[] args)
    {
      string myString = "10/28/2002";
      DateTime myDateTime = DateTime.Parse(myString);
      Console.WriteLine("{0:F}", myDateTime);
    }
  }
}
```

假设当前文化设置为 English(United States)，其输出为：

```
Monday, October 28, 2002 12:00:00 AM
```

应该注意的是，可以传递 CultureInfo，以便按照指定的文件分析日期/时间信息。如果对代码稍作修改，传递一个 English(United Kingdom)文化，就会得到不同的结果：

```
using System;
using System.Globalization;

namespace Wrox.Text.Chapter3
{
  class ParseUK
  {
    static void Main(string[] args)
    {
      CultureInfo myCultureInfo = new CultureInfo("en-GB");
      string myString = "28/10/2002";
      DateTime myDateTime = DateTime.Parse(myString, myCultureInfo);
      Console.WriteLine("{0:F}", myDateTime);
    }
  }
}
```

这段代码的输出为：

```
Monday 28 October 2002 00:00:00 AM
```

有一点很重要，因为日期的格式取决于当前所使用的文化，所以如果不提供文化信

息，就不能始终保证正确理解日期的含义。比如，02/01/2002 可以理解为 2002 年 1 月 2 日，也可以理解为 2002 年 2 月 1 日，这就取决于所使用的文化。由于这很容易导致存储不正确的日期，所以最好在任何日期处理代码中显式地设置文化信息，并保证日期存储为一个与文件无关的格式——这正是.NET Framework 中的不变文化所提供的功能(详细内容请参考第 4 章)。

Parse() 方法的另外一个特征是，默认情况下，在所传递的字符串中没有包含的任何日期或时间信息都会使用当前的日期和时间值替代。

ParseExact()

ParseExact()方法仅把指定的字符串模式转换为 DateTime 对象，如果所传递的字符串不匹配这个模式，就会有一个 FormatException 抛出。下面的代码示例中，会向 ParseExact()方法传递一个匹配于 en-US 短日期模式的字符串对象。

```
using System;
using System.Globalization;

namespace Wrox.Text.Chapter3
{
  class ParseExact
  {
    static void Main(string[] args)
    {
      CultureInfo myCultureInfo = new CultureInfo("en-US");
      string myString = "10/28/2002";
      DateTime myDateTime = DateTime.ParseExact(myString, "d",
                                                myCultureInfo);
      Console.WriteLine("{0:d}", myDateTime);
    }
  }
}
```

这段代码随着所设置的当前文化的不同而输出不同的结果。例如，在 US 文化下，输出是：10/28/2002。而在 UK 文化下，输出就变成了：28/10/2002。

3.7 在集合与数组之间移动字符串

本节简要讨论 System.Collections 命名空间的部分成员以及 System.Array 类型的对象。因为集合与数组存储成员的方式不同，需要对此进行一些讨论。

一个对象通常可以强制转换为一个它所继承的对象，也可以强制转换成它自己的数据类型。如果要对一个包含在数组或集合中的元素执行字符串操作，这个规则是很有用的。

3.7.1　数组

System.Array 对象可以定义为固定或可变的大小，对象本身并不包含任何其他对象的内容。如果定义一个值类型的数组，它就会包含值，但是因为 System.String 是一个引用类型，字符串数组将只包含对字符串位置的引用。另外，一个数组只能包含相同类型的元素。这是因为.NET Framework 更愿意知道数组的确切大小。值类型是固定长度的，所以，一个包含 10 个整数的数组，将占用 40 个字节的内存。这就是说，堆可以准确地分配数组所需要的空间。一个包含 10 个字符串的数组所占用的内存与一个包含 10 个整数的数组所占用的内存是相同的，因为引用也是 32 位的。由于所有的引用都是相同的，所以就可以定义一个 System.Object 类型的数组，它对于字符串和所有其他对象都将会分配相同数量的内存。

在下面的代码中可以看到它的应用。在数组中，GetValue(index)是用来返回指定下标的元素值，SetValue(item, index)用来设置或改变指定下标的元素值：

```
using System;

namespace Wrox.Text.Chapter3
{
  class ArrayExample
  {
    static void Main(string[] args)
    {
      string[] firstArray = {"what", "did", "you", "do"};

      for (int count = 0; count < firstArray.Length; count++)
      {
          Console.WriteLine("{0} - Type: {1}",
                            firstArray.GetValue(count),
                            firstArray.GetValue(count).GetType());
      }

      Console.WriteLine("Type of firstArray is {0}",
                        firstArray.GetType());
    }
```

```
        }
    }
```

这段代码有几处值得注意的地方。首先，数组的第一次定义分为几个阶段。在堆中有4个区域被分配，用于包含字符串“what”、“did”、“you”和“do”。其次，构造了一个限定为4个元素的Array对象。接着，把对4个字符串的引用插入到数组中的每一个元素。如果改变了这个数组中的任何一个元素，它都会创建一个新的String对象并插入一个对它的引用。

现在，看看第一个Console.WriteLine()。在此不需要把它转换成字符串类型，因为编译器能够在编译时识别出它是一个字符串。在接下来的代码中，我们利用一个格式字符串来输出GetType()的结果。因为GetType()返回一个System.Type，所以它必须转换成字符串，并且格式化指令要显式完成这个工作。这里的For循环4次，返回System.String。第二个Console.WriteLine()再次利用格式字符串返回数组类型。这里，返回的不是System.Array或System.Object，而是System.String[]，表示它是一个包含System.String类型元素的数组对象。由于GetValue()方法总是返回我们所期望的类型，所以数组是很容易处理的。

现在，如果要改变这个数组的大小，将会按照指定的大小创建一个新的Array对象，并把第一个数组中的内容复制到第二个数组的第一个元素中。这种做法效率极差，所以.NET Framework就提供了集合，以便存储不同类型的不同对象，像第2章讨论过的StringBuilder对象那样，尽量减少内存复制的数量。

3.7.2 ArrayList对象

ArrayList是数组环境下的StringBuilder。像StringBuilder一样，在使用默认构造函数时，初始化可以使用16个元素，而且它还允许把一个值传递到它的构造函数，这样就可以指定调整它的大小以包含不同数量的元素。ArrayList存储任何System.Object类型的对象——实际上是存储任何对象。这是很有利的一点，因为它意味着任何对象都可以包含进来，但同时也是一个负担，因为返回的对象总是需要转换成它的原始类型，这样编译器才不会抱怨开发人员希望执行的操作。不管怎么说，这还是一个好的办法。因为开发人员可能无法确定ArrayList中某个位置所包含的对象是什么类型。ArrayList像数组那样把它的元素存储到指定的下标中，但它可以包含不同的类型，如果需要添加更多的元素，那么内存分配就较少。与数组不同的是，ArrayList利用Add()方法添加一个新元素，利用Item()方法检索指定下标处的元素，还利用更多其他的方法，比如Insert()、Remove()、RemoveAt()、Sort()和IndexOf()执行更强大的操作。Insert()方法和RemoveAt()方法会使ArrayList中的内容上移或下移一位，这要依赖于方法，它们的执行是很耗费资源的。下面看一下ArrayList的使用：

```
using System;
using System.Collections;

namespace Wrox.Text.Chapter3
{
  class ArrayListExample
  {
    static void Main(string[] args)
    {
      ArrayList myList = new ArrayList();

      myList.Add("Humpty Dumpty sat on the wall");
      myList.Add("Humpty Dumpty had a great fall");
      myList.Add("All the king's horses and all the king's men");
      myList.Add("Couldn't put humpty together again");
      myList.Add(6);
      myList.Add(6.3);
      myList.Add(true);

      for(int count = 0; count < myList.Count; count++)
      {
        Console.WriteLine("{0}\tType: {1}",
                          myList[count],
                          myList[count].GetType());
      }
    }
  }
}
```

其输出如下所示：

```
Humpty Dumpty sat on the wall     Type: System.String
Humpty Dumpty had a great fall    Type: System.String
All the king's horses and all the king's men      Type: System.String
Couldn't put humpty together again        Type: System.String
6          Type: System.Int32
6.3        Type: System.Double
True       Type: System.Boolean
```

可以看到，返回的项目是我们所期望的类型，而且 String.Format()方法（由 Console.WriteLine 使用）可以很轻松地把其他类型转换为字符串。

3.7.3 IDictionary 对象

这些对象包括 HashTable 和所有其他键-值对集合。这些集合的工作方式与其他集合完全相同。键可以是任何对象。String.Format()可以处理包含在任何集合中的任何字符串。

3.8 小结

本章学习了字符串与其他数据类型之间进行转换的最佳方法，并考虑到了不同数据类型的输入方法可能不同，在不同区域的输出也会不同。详细讲解了从数字和日期到字符串的转换，它可能需要按照区域设置以完全不同的方式进行格式化。我们还学习了这些操作的反向操作，即把一个格式化的字符串转换成指定的数据类型。

最后一节讲述了集合包含字符串的方法——将其作为对对象的引用。这一节说明了引用 Array、ArrayList 和 IDictionary 集合的成员时所出现的问题。简单地说，这些对象可学习的内容很少。集合只是包含对独立 String 对象的引用。

下一章将详细讲述本章中所提到的特定文化的问题，并介绍如何在.NET Framework 中使用 Unicode。

第4章 国 际 化

创建一个真正的国际化应用程序所面临的主要难题之一是如何很好地处理文本，这涉及到很多方面，如满足本地化的需要，实现大小写，排序字符串，输出字符串和字符，处理 Unicode 的复杂元素(如代理对和组合字符)等。幸运的是，.NET Framework 所提供的一组类可以针对它所支持的各种文化正确地处理文本。简单地说，本章的主要内容为：

- Unicode——什么是 Unicode 以及如何在.NET 中编码
- 什么是文化
- 不同种类的文化以及如何处理它们
- 排序以及涉及不同文化的各种排序问题
- 处理 Unicode 字符，包括代理对和组合字符

通过本章的学习，您将准确地理解.NET 中字符串的处理，以及如何利用新的 Unicode 默认字符集来确保应用程序的国际化。您还将了解处理 Unicode 所涉及的问题，如不同语言中字母排序的不同方法。在探讨所有这些特性和.NET 提供的类之前，先研究一下.NET Framework 处理字符和字符串采用的标准 Unicode。

4.1 Unicode

虽然应用程序和用户可以处理文本，但目前的计算机还只能对数字直接进行操作——所以，计算机通过为每一个字符分配一个数值的方式来处理字母、数字和其他字符。这是通过编码实现的，编码就是把字符与任意值匹配起来的方式。传统上，根据 ASCII 标准，字符存储为 0～127 的二进制数。

后来，出现了其他编码方式，如 MS-DOS 或 Windows 代码页(代码页只是编码的另一个名称)，它们以 8 位二进制数存储字符，取值范围从 128～255，从而能够容纳字母表不同或更广泛的语言(如俄语、希腊语、西班牙语和法语)的附加字符。但是，并不是所有的附加字符都能够同时提供，不同的编码支持不同的语言。在 Microsoft Windows 中，用一种编码(代码页 1252)为所有的西欧语言提供支持，提供 ASCII 范围内的字符和西欧语言中使用的重音字符(如 é, ç, ü, ñ 等)。但是，这并没有把中欧和东欧语言所需要的所有字符都包含进来，所以需要用其他编码支持中欧和东欧语言(如俄语和希腊语)。

此外，还有支持东亚语言的一些编码(如日本使用的 Shift-JIS)，这些语言使用了许多其他字符。这些编码方案把用一个字节编码的字符和用两个字节编码的字符混合在一起，这使大部分的字符串操作(如查找下一个字符)更加困难。虽然所有这些编码都提供了处理一种、有时是几种语言的方法(只要使用相同的字符集编码)，但没有一种编码能够同时处理所有的语言。这是因为没有一种编码提供了足够的字符，这种局限已经产生了严重的后果。例如欧盟，就必须使用多种编码来支持它的各种语言。同一个值在不同编码中可能代表不同字符，同一个字符在不同编码中可能具有不同的值，这就增加了问题的复杂性，在不同计算机间交换数据时，它将产生很多的问题。

为了解决这些问题，需要一种更好的解决方法，这就是 Unicode。Unicode 标准是由 Unicode 协会(http://www.unicode.org)开发出来的，它能够为世界上所有的书面语言编码所有的字符。它为每一个字符命名并赋予一个惟一值。这个惟一的值称为码点(code point)，通常用一个前缀为“U”和“+”的十六进制数来表示。例如，码点 U+0042 表示 Unicode 标准中的字符 B，此字符命名为“LATIN CAPITAL LETTER B”。

Unicode 最初的目标是使用 16 位编码，为多于 65,000 个的字符提供码点。但是，65,000 个字符并不足以为全世界所有语言的字符编码。所以，Unicode 标准支持了附加机制，以表示 100 万个以上的字符，为现在和过去的所有语言的字符提供了足够的空间。这使 Unicode 标准稍微有些复杂，这部分内容将在“处理字符”一节中详述。

最后，Unicode 标准根据不同语言所使用的不同书写符号把字符从逻辑上分组。编码从 U+0000 开始为标准 ASCII 字符编码，然后从 U+0080 开始，代表了希腊语文字，古代斯拉夫文字，希伯来语文字，阿拉伯语文字以及其他大部分的文字——最终它应支持所有的文字。

编码格式

Unicode 标准化组织定义了 3 种编码格式(Unicode Transformation Formats 或 UTF 编码)，可以分别用 8 位(字节)，16 位(字)，32 位(双字)为相同的数据编码。3 种格式可以为相同的数据编码，数据可以方便地转换，而且转换时没有任何的数据丢失。

UTF-8 是一种通用的 HTML 和 XML 编码，因为它的编码方案与 ASCII 标准的字符向后兼容。使用 UTF-8，Unicode 字符可以转换为字节码。它的主要优点在于，对应于 ASCII 字符集的 Unicode 字符也是用单字节编码，而且与对应的 ASCII 字符具有相同的值。例如，Unicode 字符串“ABC”编码为一系列字节：0x41, 0x42, 0x43，就像用 ASCII 编码一样。超出 UTF-8 的 256 个字符范围外的 Unicode 字符(也就是码点大于 U+00FF 的字符)用 2 个或 3 个字节编码。.NET Framework 通过一个编码类对 UTF-8 编码提供支持。

UTF-16 是最常见的 Unicode 编码，人们在提到 Unicode 时，通常指的就是 UTF-16。

因为 Unicode 最初就是 UTF-16。它用 16 位的代码表示大部分的字符，有些字符用一对 16 位代码表示。这个主题将在“处理字符”一节中更深入地讨论。.NET Framework 中的字符串使用 UTF-16，但一般情况下这一点对开发者是透明的。

当开发者需要使用固定宽度的字符，就可以用到 UTF-32，它使用 32 位为所有的 Unicode 字符编码。但是，使用此格式的代价是很高的，因为每一个字符都占用 4 个字节，因而没有其他两个编码的用途广。.NET Framework 并没有内置对 UTF-32 格式的支持。

最后，要提到的另一个编码格式是 UTF-7。此编码用 7 位二进制数为所有 Unicode 字符编码。此格式使用转义字符为 ASCII 没有表示的字符编码。这些转义序列是少见的 ASCII 字符对，它表示接下来的 2 个或 3 个字符应解释为 ASCII 外的字符。这种格式实际上应该只用于 7 位传输(如邮件和新闻传输)的上下文中。.NET Framework 提供了一个编码类，使用它可以把.NET Framework 内部使用的 Unicode 转换为 UTF-7 序列。

4.2 .NET Framework 的编码类

对 Unicode 有了基本的了解后，下面介绍.NET Framework 所提供的编码类。System.Text 命名空间提供了一系列的编码类，可以用来在 Unicode (UTF-16)和其他编码间对字符串进行转换。为什么要进行转换？因为从已有的代码页转换为 Unicode，是在依赖操作系统上代码页的原应用程序和在.NET Framework 上开发的应用程序之间进行数据转换的一种方式。

Encoding 类

Encoding 类是其他编码类的基类。它提供了方法，可以使 Unicode 字符数组和字符串与使用 Windows 代码页编码的字节数组之间进行相互转换。简单地说，这种机制允许把 Unicode 字符转换为字节，因为不支持 Unicode 的 Windows 旧版本(如 Windows 98)开发的应用程序可以把这些字节识别为字符。

下面的代码示例演示了如何使用编码类在 Unicode 和代码页 1252(Windows Latin1，为英文使用的代码页)及 932(Shift-JIS，日文，为日文使用的代码页)间转换字符串。注意，使用 ConvertToEncoding()方法来转换日文字符，指定的代码页是 1252 还是 932，产生的结果是不同的。如果指定了代码页 1252，这些字符将无法识别，因为代码页 1252 中不支持日文，而只处理用于编写西欧语言的 Latin 文字。此时日文字符会转换为问号。但是如果指定代码页 932，将生成正确的日文字符。

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

namespace Wrox.Text.Chapter4
{
  class ConvertEncoding
  {
    [STAThread]
    static void Main(string[] args)
    {
      ConvertToEncoding("Hello, world", 1252);
      ConvertToEncoding("Hello, world", 932);

      String japaneseString = "\u307B\u308B\u305A\u3042\u306D";
      ConvertToEncoding(japaneseString, 1252);
      ConvertToEncoding(japaneseString, 932);

      Byte[] bytes = { 65, 66, 67, 233, 230 };
      ConvertFromEncoding(bytes, 1252);
    }

    static public void ConvertToEncoding(String text, int codePage)
    {
      // Get the encoding for the specified code page and
      // get the byte representation of the specified string
      Encoding targetEncoding = Encoding.GetEncoding(codePage);
      Byte[] targetBytes = targetEncoding.GetBytes(text);

      // display the bytes and their values
      String message = String.Format(
          "String \"{0}\" has been converted to: ", text);

      for (int count = 0; count < targetBytes.Length; count++)
      {
        message += String.Format("{0} ", targetBytes[count]);
      }

      MessageBox.Show(message);
    }
```

```
    static public void ConvertFromEncoding(Byte[] bytes, int codePage)
    {
      // Get the encoding for the specified code page and
      // convert the byte array to Unicode
      Encoding sourceEncoding = Encoding.GetEncoding(codePage);
      String targetString = sourceEncoding.GetString(bytes);

      // display the bytes and the string
      String byteString = String.Empty;
      for (int count = 0; count < bytes.Length; count++)
      {
        byteString += bytes[count].ToString() + " ";
      }

      String message = String.Format(
          "Bytes ¥"{0}¥" have been converted to string: {1}",
          byteString, targetString);
      MessageBox.Show(message);
    }
  }
}
```

如果安装了国际字体(在 Windows XP 系统下占用 230MB 的空间)，这段代码的输出将如图 4-1 和图 4-2 所示。

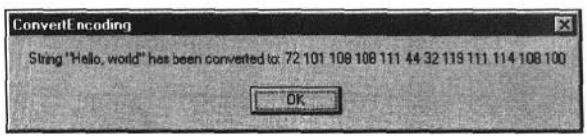


图 4-1

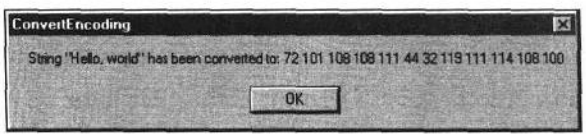


图 4-2

如图 4-1 和图 4-2 所示，字符串“Hello, world”在代码页 1252 (西欧) and 932 (日本)以相同的方式转换为字节。这是因为它们所转换的字节是这些字符的 ASCII 值，而 ASCII 字符存在于所有的代码页。

图 4-3 说明，在代码页 1252，日文字符串转换为值为 63 的字符(问号)。这是因为

这些字符不存在于此代码页，所以在从 Unicode 转换到代码页 1252 时，将无法解释这些字符。此时数据会丢失，因为不能被转换的字符会用问号来代替。

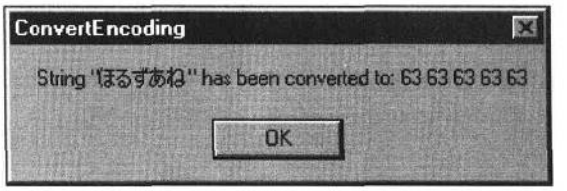


图 4-3

图 4-4 中的 Unicode 日文字符正确地转换为代码页 932 中的等价表示，因为支持日语的代码页 932 中包含这些字符。

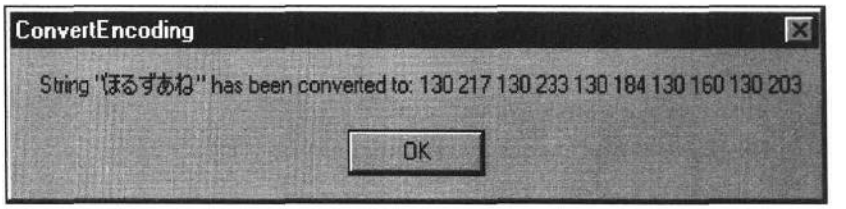


图 4-4

最后，转换还可以以相反方向进行，代码页 1252 中的字符能被成功转换为 Unicode 字符，如图 4-5 所示。

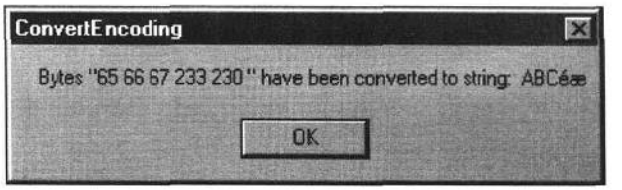


图 4-5

代码使用了方法 Encoding.GetEncoding()，传递一个代码页作为参数来获得用来格式化字节数组的编码。然后它使用了方法 Encoding.GetBytes()把 Unicode 字符数组转换为字节数组。与此方法执行相反操作的一个方法 GetChars()可以把字节数组转换为 Unicode 字符数组。

很多 Encoding 的派生类也在 System.Text 命名空间中提供，包括：

- ASCIIEncoding 类，用 7 位 ASCII 字符为 Unicode 字符编码，而且只支持范围在[U+0000-U+007F]的字符，即 ASCII 范围内的字符。如果试图转换此范围外的一个 Unicode 字符，将得到一个异常，并且字符也不会被编码。例如转换 é 会发生异常，因为它的值在 ASCII 范围之外。
- UnicodeEncoding 类，用两个连续字节为每个 Unicode 字符编码。不同的操作系

统在内部存储数据时使用不同的字节顺序，有的采用大数在前的顺序优先存储重要字节，有的采用小数在前的顺序最后存储重要字节。例如，字符 F 定义为 Unicode U+0046，在采用大数在前的顺序的机器中，编码为字节 0046，在采用小数在前的顺序的机器中，编码为字节 4600。

- UnicodeEncoding 类，提供了对这两种字节排序的支持，从而使这个类在.NET Framework(小数在前的顺序)与一些大型机和 Unix 系统(大数在前的顺序)交换数据时特别有用。
- UTF7Encoding 类，使用 UTF-7 为 Unicode 字符编码，支持所有的 Unicode 字符值，并把它们转换为 7 位的变长格式。
- UTF8Encoding 类，使用 UTF-8 为 Unicode 字符编码，也支持所有的 Unicode 字符，例如本书下载代码中就提供了这个类的例子。

可以像在示例中一样，使用 Encoding 类在计算机支持的代码页和 Unicode 之间转换数据。例如，可以用这种方式支持中华人民共和国的 GB-18030 编码。假设在计算机上安装了这个编码的正确支持文件(可从 Microsoft China 的 Web 站点 http://www.microsoft.com/china/windows2000/ downloads/18030.asp 下载)，将能够在基于.NET Framework 的应用程序中通过 Encoding 类使用此编码。

如果处理的是大量的数据，或者数据来自流而且只能在连续的块中获得，就需要使用 Decoder 和 Encoder 类来执行转换。这些类与方法 GetBytes()和 GetChars()不一样，它们可以保存支持跨越几个块的数据转换所需的状态信息。

4.3 处理字符串

在现实世界中，应用程序应只在绝对必要时才处理字符串，因为每一个操作可能都会涉及复杂性和性能问题。幸运的是，.NET Framework 在处理这种复杂性时提供了很多的帮助，使开发人员能用一种简明的方式创建支持国际化字符串的应用程序。但是，对于国际化字符串的操作还有一些普遍的准则，如严格的国际化字符串操作代价通常要高于不考虑文化正确性的操作。另外，处理整个字符串要比一个一个处理字符更容易，因为国际化字符串可能包含一些字符组合，而在不破坏数据的情况下，这些组合不能随意分离。

在学习标准字符串操作(如大写，小写和排序)前，需要先了解文化以及文化如何影响字符串操作。

4.3.1 CultureInfo 类

CultureInfo 类是 System.Globalization 命名空间中最重要的类。它提供了特定文化的信息，如文化名称，它的相关语言，国家/区域，日历和文化惯例。这个类还能够访问提供其他特定文化信息的类的实例，如 DateTimeFormatInfo、NumberFormatInfo、CompareInfo 和 TextInfo。这些对象包含特定文化的操作(如大小写，格式化日期和数字，比较字符串)所需的信息。所以，通过指定一个文化，就可以将一套通用的参数用于与用户的文化习俗相对应的字符串、日期和数字格式的信息。

1. 文化名称

文化名称派生于 RFC 1766 标准(www.ietf.org/rfc/rfc1766.txt)。这个标准使用了格式 language[-country/region]，其中 language 是一个小写的 2 字母(有时为 3 字母)代码，由 ISO-639-1 标准派生而来，country/region 是一个大写的 2 字母代码，派生于 ISO 3166 标准。下面给出一些例子：

- en 表示英语
- en-US 表示美国英语
- kok-IN 表示印第安语

一个文化名称可以包含也可以不包含 country/region 信息，而且名称可以只用语言信息来指定。例如，日语作为一种语言指定为 ja，作为口语指定为 ja-JP。

此外，一些文化名称具有的后缀可以指定文字，例如-Cyrl 代表古代斯拉夫文字，-Latn 代表拉丁文字，sr-SP-Cyrl 代表塞尔维亚(古代斯拉夫)(塞尔维亚)文字，sr-SP-Latn 代表塞尔维亚(拉丁)(塞尔维亚)文字。在写作本书时，这一点和 MSDN 文档不同。

2. 文化的不同类型

.NET Framework 实现了 200 多种文化，这些文化可以划分为以下类型：

- 惟一的不变文化
- 中立的文化
- 具体的文化

图 4-6 给出了文化树的一个子集，其中只显示了不变文化，中立的英国(en)和法国文化(fr)，与法国相关的具体的文化。

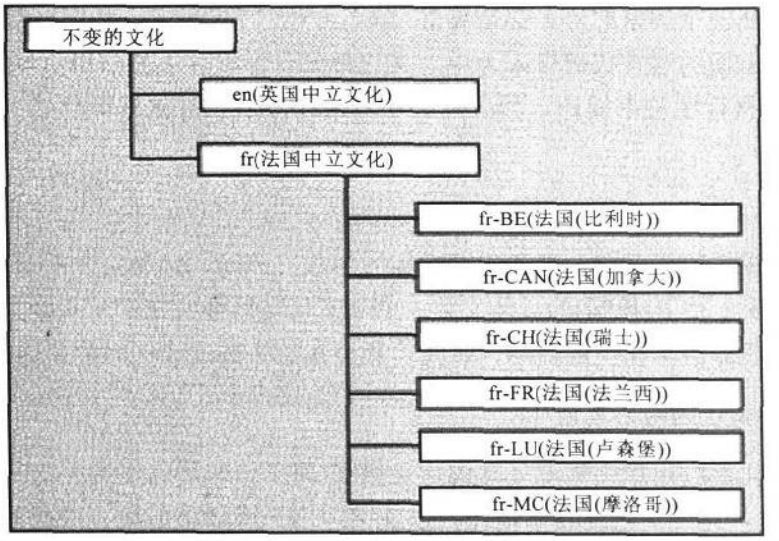


图 4-6

不变文化

不变文化是位于.NET Framework 所实现的文化树根部的一个惟一的文化。它与英语有关，但与一个国家或地区无关，是真实世界中并不存在的一种文化。它是一种完全不区分文化的文化。

这看上去很奇怪，但并不是没有逻辑，因为有许多的进程需要独立于文化的结果，如系统服务。不变文化的典型应用是处理 Web 服务的日期和时间存储，以及要传递给操作系统的文件名的大写。以一种文化独立的格式存储数据可保证已知格式不被修改(至少在相同版本的.NET Framework 中不被修改)。当不同文化的用户需要访问使用不变文化格式存储的数据时，可以为该用户适当地格式化此数据，方法是把数据转换为适合这个用户的文化。例如，在后端使用不变文化格式存储的数据，对于德国用户，可以在前端使用德国文化(Germany)进行分析和格式化，对于墨西哥用户，可以在前端使用西班牙文化(墨西哥)进行分析和格式化，这样，在前端可以得益于合适的文化，在后端则有相同的存储。

在其他情况下，比如直接用于用户界面元素，不变文化会产生在语言上不正确，或在文化上不适当的结果。所以，建议避免对用户界面元素使用不变文化。在前面的例子中，为德国用户或墨西哥用户使用不变文化来显示数据都是不恰当的。

中立文化

中立文化与语言相关，但与一个国家或地区无关。因此，中立文化非常适用于只与语言相关的操作，但不能用于检索信息，如日期和时间格式，因为各个国家和地区各不相同，即使在同一语言的范围内也可能不同。例如，美国和英国的日期和时间格式

就不相同，虽然两个国家的语言都是英语。

我们使用上面的语言代码格式为中立文化命名。例如，阿拉伯语用 ar 指定。中立文化不能用来执行字符串操作，如大写、小写或排序。执行这些操作时，需要创建一个具体文化。

具体文化

具体文化提供的信息与语言和国家或地区相关。例如，de 是一个中立的德语文化，de-DE 则是代表了德语(德国)的具体文化。中立文化与具体文化的主要区别在于后者提供了与具体地区或国家相关的日期、时间、货币和数字格式选项的附加信息。

3. CultureInfo 的两个角色

一个国际化的应用程序需要处理国际化数据(属于该应用程序数据的日期、时间、货币和字符等)。此外，它还要本地化，以使国际用户能以最简单、最高效的方式使用该应用程序。但是，对于一个国际化应用程序，本地化并不是必须的。

在两种情况下，CultureInfo 类都扮演了一个重要的角色——文化预设参数(culture preference)的提供者。因为这两个角色截然不同，所以即使很多情况下它们使用相同的文化，.NET Framework 还是为两个角色都实现了一个机制，也就是 CurrentCulture 和 CurrentUICulture。

CurrentCulture 和 CurrentUICulture 在应用程序的线程级设定，使多线程的应用程序在不同的线程使用不同的文化。如果没有指定它们的值，它们将从操作系统继承，所以开发人员总能拥有有效的 CurrentCulture 和 CurrentUICulture 定义。

CurrentUICulture

CurrentUICulture 的值决定了如何为一个窗体加载资源，用于对资源数据进行特定文化的查找。CurrentUICulture 值确定了窗体元素以什么语言显示。如果它设置为法语，用户界面就是用法语显示(假定本地化资源可用)。CurrentUICulture 值的惟一作用是，指定资源管理器加载与窗体相关的资源应使用的语言(如果窗体是可本地化的)。开发人员可以使用中立文化或具体文化来设定 CurrentUICulture。

CurrentCulture

CurrentCulture 的值决定了其他方面——日期格式、数字格式、字符串大小写和比较等。它只能设为具体文化，如 en-US 或 en-GB。这样就不必为 en 指定正确的货币符号了。

两种文化设定不需要具有相同的值。在应用程序的设计中，分别设定它们非常重要。例如，如果在为美国创建一个应用程序，但是想用西班牙语显示用户界面的菜单，而不是英语，可以把 CurrentCulture 设定为 en-US，以使数字格式正确，把 CurrentUICulture 设定为 es，以使本地化资源以西班牙语显示。另外，当运行一个没有本地化为日语的美国应用程序时，CurrentCulture 设为 ja-JP，以使日期、时间、货币和数字格式都为日

语格式，而 CurrentUICulture 设为 en 或 en-US，以使用户界面以英语显示。

4. 设置 CurrentCulture 和 CurrentUICulture

首先需要注意，在有些情况下不需要显式地设置这些文化。例如，如果开发一个 Windows 窗体应用程序，通常应该避免显式地设置这些值(至少是 CurrentCulture)，因为操作系统提供了与当前用户选择(CurrentCulture)和设置(CurrentCulture)匹配的隐含默认值。

隐含值

如果没有显式地设定 CurrentCulture 属性，那么系统将从 Windows 中的 GetUserDefaultLCID API 取得一个默认值。修改控制面板中的 Regional Options | Set Locale 选项会影响到此值。

如果 CurrentUICulture 属性没有在应用程序的代码中显式地设置，它将在应用程序启动时，由 Windows 资源使用的默认语言来设置。用户不能修改这个默认值，除非使用一个 Windows 版本的 MUI(Multilingual User Interface，多语言用户界面)，如果使用了 MUI，在控制面板中可以得到一个附加的选项来修改此默认值。

显式值

在某些情况下会调用默认值，还有一些情况需要开发人员显式地设置 CurrentUICulture 或 CurrentCulture。例如，如果开发一个服务器应用程序(如使用 ASP.NET)，默认的地区值将从服务器中获得，这些值很可能与用户在他们的客户机上所要求的值有差别。在这种情况下，显式地设置当前文化值和当前 UI 文化值是必需的。

把当前线程的文化属性设置为 CultureInfo 类的实例来完成这个工作：

```
using System.Globalization;
using System.Threading;
    ...
    Thread.CurrentThread.CurrentUICulture = New CultureInfo("en");
    Thread.CurrentThread.CurrentCulture = New CultureInfo("ja-JP");
```

在上面两个例子中，当前 UI 文化设置为 English，而当前文化设置为 Japanese。

现在您对文化和 CultureInfo 类已经有了更进一步的了解，下面开始研究字符串处理的细节。

4.3.2 大写和小写

有一点很重要，诸如大写、小写和字符串比较等操作在文化上是有区别的，这对语言的正确性和性能都有影响。特别地，当更加复杂的查找和计算在后台执行时会影响

性能，因此对给定的文化要执行正确的操作。

1. 大写

String 类提供了 ToUpper()方法以大写一个字符串。最简单的方式是无需任何参数就可以调用此方法：

```
myString = "This is a string"
myCapStr = myString.ToUpper();
```

这样的调用将使用在当前线程级指定的文化，并使用它来执行大写。如果选择的文化为 en-US，示例字符串将包含：

```
this is a string
THIS IS A STRING
```

如果当前线程的当前文化设定为 tr-TR(Turkish, Turkey)，那么输出则不同，因为土耳其语对字符“I”有不同的大写规则。土耳其语中有 4 个“I”，它们根据表 4-1 分别进行匹配。

表 4-1 土耳其语中的“I”

小　写	大　写
i (U+0069)	İ(U+0130)
1(U+0131)	I (U+0049)

在土耳其语中对字符“i”的大写转换不同，所以得到一个不同的输出字符串：

```
this is a string
THIS IS A STRING
```

调用 String.ToUpper()方法的另一种方式是为此方法传递一个 CultureInfo 参数。在这种情况下，CultureInfo 参数将代替当前线程的文化来执行大写操作。当在同一个应用程序中使用几个文化设置时，这是很有效的。

例如，下列代码：

```
myString = "this is a string";
myCapStr1 = myString.ToUpper(New CultureInfo("en-US"));
myCapStr2 = myString.ToUpper(New CultureInfo("tr-TR"));
```

产生的结果如下：

```
this is a string
THIS IS A STRING
THIS IS A STRING
```

2. 小写

String 类还提供了 String.ToLower()方法来执行相反的操作，小写。正如 ToUpper()方法，ToLower()带有一个参数 CultureInfo 或不带参数，如果不带参数，当前线程的文化将用来执行小写操作。

4.3.3 不需要区分文化的操作

大写和小写操作都是区分文化的，所以无法保证使用一种文化与使用另一种文化执行一个操作会得到相同的结果，这一点在上一小节已经介绍过。

虽然在处理用户界面字符串时需要区分文化的操作，以使字符串能根据正确的大小写规则显示给用户，但是仍然有一些情况，如处理系统对象时，会出现问题。例如，假定要验证一个路径，此路径名包含一些字符“I”，而开发人员有两个用来设定路径名称的字符串，一个是大写的，一个是小写的。如果两个字符串不匹配，就很难证实这两个字符串指向相同的文件，因为当前文化与原先用来生成路径名的文化具有不同的大小写规则。在这种情况下，保持转换规则相同是很重要的，这也是需要不变文化的原因。

4.3.4 排序

字符串能够以两种方式排序：一是不区分文化的方式，只考虑被排序字符的实际码点值，二是区分文化的方式，确保根据选定文化的排序规则执行排序。

下面举一个例子。这里有两个字符串“ciao”和“character”，首先对它们执行一次顺序排序，只检查码点的值，即与字符相关的数值，如表 4-2 所示。

表 4-2 字符串 ciao 和 character 码点的值

Ciao	Character
U+0063	U+0063
U+0069	U+0068
U+0061	U+0061
U+006F	U+0072
	U+0061
	U+0063
	U+0074
	U+0065
	U+0072

字符串 ciao 的第二个字符(i)的码点值大于字符串 character 第二个字符(h)的码点值。所以，顺序比较返回 ciao>character。

现在进行区分文化的排序。如果选择 en-US(English, United States)文化，根据字母表顺序，h 应该排在 i 前面，所以比较仍然返回 ciao>character。但是，如果文化修改为 cs-CZ(Czech, Czech Republic)，那么捷克语的规则规定，ch 在比较和排序时作为一个实体出现在 c 后，在这种情况下，两个字符串的比较结果为 ciao<character。

可以看出，使用特定文化的排序方法进行排序，能够很好地满足用户语言的要求。作为一个规则，用户界面元素，如列表框、组合框和其他表格形式的数据列表应该使用特定文化的排序方法，以满足用户的需要。

前面已经学习了理论，现在介绍如何把理论应用于.NET。

1. String.CompareOrdinal()方法

String 类提供了 CompareOrdinal()方法，此方法无需考虑当前文化就可以比较两个字符串。例如，下列代码段通过 CompareOrdinal()方法比较两个字符串并向命令行输出了结果：

```
using System;
using System.Globalization;

namespace Wrox.Text.Chapter4
{
  class CompareOrdinalMethod
  {
    [STAThread]
    static void Main(string[] args)
    {
      String firstString = "ciao";
      String secondString = "character";

      int result = String.CompareOrdinal(firstString, secondString);

      if (result < 0)
        Console.WriteLine("\"{0}\" < \"{1}\"",
                          firstString, secondString);
      else if(result == 0)
        Console.WriteLine("\"{0}\" = \"{1}\"",
                          firstString, secondString);
      else
        Console.WriteLine("\"{0}\" > \"{1}\"",
```

```
                                    firstString, secondString);
        }
      }
    }
```

此应用程序将产生如下输出：

```
"ciao" > "character"
```

在实际的应用程序中，在处理系统对象(例如文件名)时，可使用 String.Compare Ordinal()方法比较字符串，但不希望排序的顺序根据文化而改变，例如创建一个索引或散列表。

2. String.Compare()方法

Compare()方法实现了一次区分文化的字符串比较。此方法有一些重载形式，规定了比较是否区分文化，换句话说，是提供了一个 CultureInfo 对象来代替使用当前线程的文化。还可以指定字符串的下标和长度，以便有效地比较子串。

对上述例子使用 Compare()方法：

```
using System;
using System.Globalization;
namespace Wrox.Text.Chapter4
{
  class CompareMethod
  {
    [STAThread]
    static void Main(string[] args)
    {
      string firstString = "ciao";
      string secondString = "character";

      int result = String.Compare(firstString, secondString,
                                  true, new CultureInfo("cs-CZ"));

      if(result < 0)
        Console.WriteLine("\"{0}\" < \"{1}\"",
                          firstString, secondString);
      else if(result == 0)
        Console.WriteLine("\"{0}\" = \"{1}\"",
                          firstString, secondString);
      else
```

```
            Console.WriteLine("\"{0}\" > \"{1}\"",
                                    firstString, secondString);
        }
    }
}
```

在这个例子中，比较将产生的结果如下：

```
"ciao" < "character"
```

因为我们指定了文化 Czech (Czech Republic)来执行比较。注意，我们传递了文化 Czech(Czech Republic)作为 String.Compare()方法的一个参数，如“大写和小写”小节所示。并指定 String.Compare()方法的第三个参数值为 True，说明比较应区分大小写，对于 Compare()方法的这个重载形式，第三个参数是 IgnoreCase 标志。

3. String.CompareTo()方法

此方法与 Compare()类似，但它不能用作一个静态方法，而且没有提供任何方法来指定比较是否区分大小写。下述代码段通过 CompareTo()方法实现了比较：

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

namespace Wrox.Text.Chapter4
{
  class CompareToMethod
  {
    [STAThread]
    static void Main(string[] args)
    {
      String firstString = "ciao";
      String secondString = "character";

      Thread.CurrentThread.CurrentCulture = new CultureInfo("cs-CZ");
      int result = firstString.CompareTo(secondString);

      if(result < 0)
        Console.WriteLine("\"{0}\" < \"{1}\"",
                          firstString, secondString);
      else if(result == 0)
        Console.WriteLine("\"{0}\" = \"{1}\"",
                          firstString, secondString);
```

```
            else
                Console.WriteLine("\"{0}\" > \"{1}\"",
                                  firstString, secondString);
        }
    }
}
```

4. 使用 CompareInfo 类

对于具体文化，Compare()提供的区分文化的排序在某些场合可能满足不了更高级的排序要求。

例如，日语使用了两个假名表，平假名和片假名，它们各自包含了以相同方式发音，但以不同方式书写的字符。在处理日文文本时会出现这样的情况，发音方式相同的字符有时按相同字符处理，有时按不同方式处理。如果希望在代码中支持此功能，需要使用.NET Framework 提供的另一个类——CompareInfo。

CompareInfo 类实现了一套用于以区分文化的方式比较字符串的方法。与.NET Framework 中大部分类不一样，这个类的实例不能通过一个构造函数来创建，而是需要通过 GetCompareInfo()方法来创建。此方法有一些重载形式，这些重载方法可以指定一个整数作为 Windows LCID 标识符，或指定一个字符串作为一个文化标识符(CultureInfo)。在 MSDN 文档中可以找到一个包含 Windows LCID 的列表(http://support.microsoft.com/directory/article.asp?ID=KB; EN-US;Q224804&)，该列表提供了另一种排序顺序。

要为 en-US 文化创建一个 CompareInfo 对象，可以使用如下代码：

```
CompareInfo ci = CompareInfo.GetCompareInfo(1033);
```

或者：

```
CompareInfo ci = CompareInfo.GetCompareInfo("en-US");
```

检索 CompareInfo 对象的另一种方法是取得 CompareInfo 成员，它包含在 CultureInfo 类中(特别是当前文化中)。可以使用如下语法：

```
CompareInfo ci = CultureInfo.CurrentCulture.CompareInfo;
```

这个类中最有趣的方法是 Compare()，它也提供了一些重载形式。其中有一些很有意思，它们提供了参数 CompareOptions 对比较进行调整。下面的重载方法以两个字符串作为头两个参数，CompareOptions 作为第三个参数。

```
Compare(String string1, String string2, CompareOptions options);
```

CompareOptions 枚举包含了如表 4-3 所示的成员，可以组合起来自定义用于

CompareInfo. Compare()函数的比较选项。

表 4-3 CompareOptions 枚举所包含的成员列表

成员名称	描述
IgnoreCase	表明字符串比较必须忽略大小写。在 en-US 文化的例子中，如果设定了该值，abc 和 ABC 是相同的，否则就不相同
IgnoreKanaType	表明字符串比较必须忽略假名类型。 日语使用了两个假名表，平假名和片假名，它们表示了日语的语音。平假名用于本土日语的表达和措词，而片假名用于外来词，如“computer”。相同的语音可以用平假名和片假名来表达。如果指定了 IgnoreKanaType 标志，那么一个语音的平假名字符等于同一语音的片假名字符。在这里，“バナナ”(用片假名正确书写的 banana)等于“ばなな”(用平假名书写的 banana)
IgnoreNonSpace	表明字符串比较必须忽略非空格组合字符，如读音符号。如果设定了该值，“é”将等于“e”
IgnoreSymbols	表明字符串比较必须忽略符号，如空字符，标点，货币符号，百分号，数学符号和宏符号等，例如，阿拉伯语中的 Kashida，如果设定该值，Kashida 将被忽略
IgnoreWidth	表明字符串比较必须忽略字符串的宽度。例如，日语的片假名可以以全长也可以以半长输出，如果选择了此值，全长输出的片假名字符就与半长输出的字符相同。所以，"ﾊﾞﾅﾅ" 等于 "バナナ"
None	字符串比较的默认选项，和使用 String.Compare()进行的区分大小写的排序操作的结果是相同的
Ordinal	表明字符串比较必须按顺序进行。相当于“排序”一节所描述的顺序字符串比较
StringSort	表明字符串比较必须使用字符串排序算法，连字符、省略号，还有其他非字母数字字符出现在字母数字之前

4.4 处理字符

在操作国际化的字符串时，直接处理字符并不好。因为处理起来很复杂。虽然最初 Unicode 似乎是一种比代码页系统简单的模型，但是涉及到两类 Unicode 字符时处理仍然十分复杂。这两类字符是：代理对(surrogate pair)和组合字符(combining character)。

此时最好避免处理单个的字符，只处理字符串。但是，如果确实需要对单个字符操

作，那么下面的信息会有所帮助。在学习如何处理这种复杂性之前，应该注意，Unicode提供了标准中定义的每一个字符的信息。这些信息可以在http://www.unicode.org上找到。通过这些信息，可以了解哪些字符可能会出现问题。.NET Framework通过Char类发布有关Unicode字符的信息。

4.4.1 关于字符的必要信息

Char类提供了用于决定一个字符的角色及其属性的一些方法。基本方法为Char.GetUnicodeCategory()，可以指出字符所属的类。

以下的代码段显示了如何检索字符“a”的Unicode种类：

```
UnicodeCategory uc = Char.GetUnicodeCategory('a');
```

Char类还提供了许多IS方法，它们不精确地匹配了不同的Unicode类别。MSDN包含了一个可使用类别的列表(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemGlobalizationUnicodeCategoryClassTopic.asp)。

4.4.2 代理对

到目前为止，用于处理字符的Unicode与现有的代码页并没有什么不同。现在学习有关代理对和组合字符的复杂性。

Unicode常常被认为是以16位二进制数定义所有字符的一种简单机制。但是，事实却不是如此。Unicode能够提供65,536个字符，但仍然不足以表示世界上所有文字的字符。例如，中文就有多于85,000个字符，也就是说需要用多于16位的数来表示更多的字符。为了支持这些附加的字符，Unicode定义了一种机制，不必使用32位字符就可以使用附加的917,476个字符，同时能够保存与现有字符集的兼容性。这种机制称为代理对机制。

Unicode标准把一个代理对定义为单个抽象字符的编码字符表示，它包含两个代码单元，第一个单元是高端代理，第二个单元是低端代理。简单地说，要表示一个高于可以用16位编码的字符值，可以使用一对16位的字符(高端代理和低端代理)来表示相同的值。

1. 实现方法

在Unicode代码区高端，预留了一个1024个字符的区域(U+D800到U+DBFF)用于高端代理。另外一个预留区域U+DC00到U+DFFF用于低端代理。一个高端值和一个低端值组合在一起就构成了一个惟一的代理对。组合使用两个16位的Unicode值以及

它们在显示时对一个实际字符的惟一成对映射。在这两个预留区域内部没有分配其他的字符或符号。同样地，代理没有对应的解释，除非作为代理对的一部分出现时。孤立的一个代理是毫无意义的。所以，一个应用程序在预留区域遇到一个 Unicode 值时，它可能只是代理对的一半。

N 的 Unicode 值(0 到 10FFFF(16 进制)中的一个值)与一对代理字符之间的转换按如下方式定义。首先是高端代理：

```
(N – 0x10000) / 0x400 + 0xD800;
```

而低端代理可以通过下列表达式计算：

```
(N – 0x10000) % 0x400 + 0xDC00;
```

假设一个字符有一个标量值 0x2002F，匹配代理对如下，高端代理为 0xD840 ((0x2002F-0x10000)/0x400 + D800)，低端代理为 0xDC2F ((0x2002F-0x10000) % 0x400 + 0xDC00)(%是 C#中的取模运算符)。

2. 支持代理的原因

3.1 版本的 Unicode 标准引入了由代理表示的字符。表 4-4 列出了 3.2 版本的 Unicode 标准定义的范围。

表 4-4 Unicode(3.2 版本)范围列表

范围名称	起始端	终止端
Old Italic	U+10300	U+1032F
Gothic	U+10330	U+1034F
Deseret	U+10400	U+1044F
Byzantine Musical Symbols	U+1D000	U+1D0FF
Musical Symbols	U+1D100	U+1D1FF
Mathematical Alphanumeric Symbols	U+1D400	U+1D7FF
CJK Unified Ideographs Extension B	U+20000	U+2A6DF
CJK Compatibility Ideographs Supplement	U+2F800	U+2FA1F
Tags	U+E0000	U+E007F
Supplementary Private Use Area-A	U+F0000	U+FFFFF
Supplementary Private Use Area-B	U+100000	U+10FFFF

虽然表中定义的一些字符不经常使用，但是在 CJK Unified Ideographs Extension B 和 CJK Compatibility Ideographs Supplement 中获得的 CJK(China, Japan 和 Korea)字符对于东亚国家是必需的。中国，日本和韩国使用的汉字数量巨大，最初的 Unicode 只实

现了其中最常用的，没有实现它们的全部。后来，需要 Unicode 支持更多像这样的象形文字，而且也添加了更多的支持。首先是 CJK Extension A，它是主要的 Unicode 范围，然后是 CJK Extension B，可以通过代理对来访问。

使用 Unicode 代理对创建这样的字符确保了所有的字符仍然保留在相同的 Unicode 代码页中，避免了在以后移向更大的代码页，也就是说，不需要把所有的字符都移向 32 位，降低了成本。通过此方法，大概有一百万个代理对能被赋值。对字符的需要预计不会超出代理范围。

3. 处理代理对

在很大程度上，是 Microsoft 的.NET Framework 为 Unicode 字符串提供统一的支持，不管它们是否包含代理。也就是说，实际上开发人员只需要考虑代理对的几种情况。

代理对的形式是可预知的。它们总是由两个值组成，代理对的第一个成员表示一个高端的值，第二个成员表示一个低端的值。引用代理对也很容易。每一个代理对都是由两个 16 位的值组成。因为代理对总是由高位和低位组合而成，所以提供了一种简单的方法回撤到一个代理对序列的开始。以下几小节中给出了开发应用程序时所遇到的典型情况，并提供了处理代理时开发人员需要实现的支持级别。

4. 输出代理

Windows 窗体和 Web 窗体能够正确地输出代理，只要它们使用的一种字体包含了所输出字符的符号，如“Arial Unicode”。基于控制台的应用程序不输出代理，因为控制台没有提供任何能够正确显示符号的非均衡字体。

要正确输出一个 Windows 窗体控件代理，必须在窗体设计器中选择一种字体，否则控件的字体属性需要在运行时通过代码来修改。在下例中，为了选择一种字体，使用了一个 FontDialog 控件，然后把新选择的字体分配给一个显示代理对的标记。如果这个字体支持代理，控件可以正确输出它们。

如图 4-7 所示，一个使用.NET Framework 开发的 Windows 窗体应用程序显示一套代理字符。除了找到一种能够输出代理对所表示的字体外，输出并不需要特别的操作。

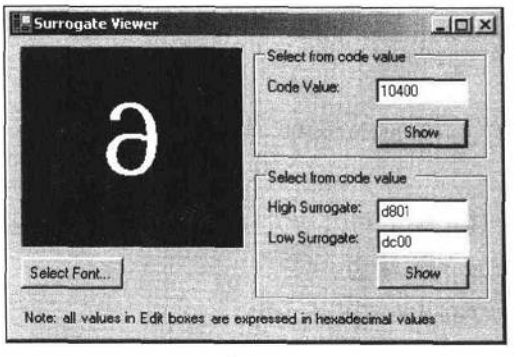


图 4-7

此例完整的源代码可以在下载代码中找到，但相关的代码如下所示。首先，处理 Select Font...按钮的方法为：

```
private void ButtonFont_Click(object sender, System.EventArgs e)
{
  FontDlg.Font = CharDisplay.Font;
  if(System.Windows.Forms.DialogResult.OK == FontDlg.ShowDialog())
    CharDisplay.Font = FontDlg.Font;
}
```

上述类并不复杂。它只是弹出了字体对话框，并通过修改 CharDisplay(显示字符的对话框)的 Font 属性来选择字体。最有趣的代码包含在如下的 Display()方法中。

```
private void Display(int codepoint)
{
  if(codepoint < 0x10000)
    CharDisplay.Text = "" + (char)codepoint;
  else
    CharDisplay.Text = "" + (char)GetHighSurrogate(codepoint) +
                       (char)GetLowSurrogate(codepoint);
}
```

在这里传递了字符完整的码点，如果它不是一个代理对(小于 0x10000 的代码值)，则在转换为 System.Char 并追加到一个空字符串末尾，以执行一个字符串的隐式转换之后输出。如果它是一个代理，就调用两个方法，GetHighSurrogate()和 GetLowSurrogate()，并分别返回两个 Char，同时把它们追加到文本框中，然后 Windows OS 和提供的字体把它们转换为所需字符。两个方法如下所示：

```
private int GetHighSurrogate(int codepoint)
{
  return (codepoint  - 0x10000) / 0x400 + 0xD800;
}

private int GetLowSurrogate(int codepoint)
{
  return (codepoint - 0x10000) % 0x400 + 0xDC00;
}
```

可以看出，这些方法只实现了与以前方法相同的功能，并返回了请求的高端代理和低端代理。与这些方法相反的方法为：

```
private int MakeCodePoint(int high, int low)
{
```

```
    return (high - 0xD800) * 0x400 + (low - 0xDC00) + 0x10000;
}
```

上述代码使用了相同的等式来检索高端代理和低端代理的代码值。两个 Show 按钮的事件代码如下所示：

```
private void ButtonCV_Click(object sender, System.EventArgs e)
{
  int ncp = int.Parse(ValueInput.Text,
      System.Globalization.NumberStyles.HexNumber);
  HighInput.Text =
      (ncp < 0x10000) ? "n/a" : GetHighSurrogate(ncp).ToString("X");
  LowInput.Text =
      (ncp < 0x10000) ? "n/a" : GetLowSurrogate(ncp).ToString("X");
  Display(ncp);
}

private void ButtonSP_Click(object sender, System.EventArgs e)
{
  int high = int.Parse(HighInput.Text,
      System.Globalization.NumberStyles.HexNumber);
  int low = int.Parse(LowInput.Text,
      System.Globalization.NumberStyles.HexNumber);

  // check that both high and low surrogates are in correct ranges
  if((high < 0xD800) || (high > 0xDBFF) || (low < 0xDC00) ||
      (low > 0xDFFF))
    ValueInput.Text = "n/a";
  else
  {
    int ncp = MakeCodePoint(high, low);
    ValueInput.Text = ncp.ToString("X");
    Display(ncp);
  }
}
```

第一个方法处理第一个 Show 按钮的单击事件，第二个方法处理第二个。ButtonCV_Click()首先检查文本框输入的值是否超出了代理对的范围。如果是的话，在相关的文本框输入“n/a”。如果不是，则使用上文所示的 GetHighSurrogate() 和 GetLowSurrogate()在文本框中输入正确的值。然后，使用 Display()方法在 CharDisplay 中显示此字符。

第二个方法比较简单，它使用 MakeCodePoint()组合了文本框中的两个值，并在输入前面提到的文本框前把此值传递给 Display()。

如前所述，在代码下载中可以找到完整的代码。

5. 输入代理

前面讨论了什么是代理对以及如何输出它们，现在看看如何输入代理对。

编程输入

如果采用编程的方式，代理的输入非常简单，因为它们和其他的 Unicode 字符一样被定义，而且使用相同的语法输入。例如，如下的 C#代码行创建了一个字符串，其中包含了一对代理所表示的一个字符：

```
String mySurrogatePairString;
mySurrogatePairString = "\uD840\uDC0B";
```

在 C#中，\u 格式通过其 Unicode 代码点值来表示一个 Unicode 字符。任何 Unicode 字符都可以采用这种方式来表示，例如，A 可以表示为\u0041，特别是，代理对只能以这种方式表示。在这种情况下，把字符串设置为一个代理对时，需要先指定高端代理值，再指定低端代理值。在没有提供日语输入支持的系统中，这可以在代码中输入一个日语字符串。

用户输入

用户输入依靠一个输入方法编辑器(Input Method Editor，IME)，使用它可以直接输入字符。IME 是 Windows 为东亚语言提供的一种机制，它可以通过输入音值，绘制字符，或直接通过键盘来输入字符。针对中文、日文和韩文的 IME 是不同的。如果用户已经安装了一个合适的 IME，就可以正确地输入代理字符。

6. 排序

在排序时代理不需要特殊的规定，因为对于这些字符，没有定义语言上很重要的排序顺序。所以，代理字符的排序顺序基本上定义为顺序排序，代理对只根据它们所代表的字符值进行排序。

4.4.3 组合字符

使用拉丁文字或其他文字的许多语言都拥有代表读音的字符。例如，“é”表示一个升调的“e”。有两种方式解释这种字符。第一种是把此字符当作已构造好的字符(“é”本身就是一个字符，有它自己的码点)。第二种是把“é”当作“e”(称为基字符)后跟一个非空格的升调字符。在这种情况下，“é”是把“e”和“´”组合在一起的复合字符。

已构建的字符序列通常由一个基字符(占据一个空格)和一个或多个非空格标记(在基字母的相同位置提供)组成。在这种情况下，字母实际上表示为一系列字符，这增加了处理字符串时的复杂性。

Unicode 标准提供了组合字符和复合字符来保持与已有标准(如 Latin 1)的兼容性，其中包括了许多复合字符，如"ü" 和 "ñ"。当字符是一个复合字符时，基字符在先，后跟一个或多个非空格字符。如果文本元素用多于一个的非空格标记编码，且标记在印刷上不互相影响，那么非空格标记的存储顺序就不重要。Unicode 标准规定了连续非空格字符如何应用于一个基字符。

1. 处理代理对和组合字符

就显示和排序而言，在处理代理和组合字符方面没有特殊的要求，但在处理字符串时如果天真地认为每一个元素都代表了一个字符，将会在插入和删除字符，或者仅仅是在遍历字符串以统计实际可显示的元素时，出现问题。

在一个结构良好的文本中，一个低端代理只能前缀一个高端代理，一个高端代理只能后缀一个低端代理。在处理组合字符时稍有不同，因为删除基字符并将非空格标记应用于前面的字符很容易导致错误。

如果是处理代理对，则如下字符串：

"AB𐓘C"

在内部以表 4-5 所示的方式表示。

表 4-5 字符串"AB𐓘C"的内部表示

0041	0042	D801	DC00	0043
A	B	𐓘		C

如果使用如下代码在上述字符串中检索头 3 个文本元素，将无法得到正确结果，因为我们将检索前两个字符和高端代理(但没有检索需要关联到高端代理才有意义的低端代理)。

```
string str;
str = "AB\uD801\uHDC00C";
string strFirstThree = str.Substring(0, 3);
```

为了正确执行操作，需要检索字符的附加信息(它们是否为代理)。所以，如果编写一个需要处理字符串中单个字符的程序，需要确定在删除一个字符或截取一个子串时，没有把高端代理和低端代理分离开来，就会导致一个代理对的分裂。检测代理并不难，因为执行范围检查将指出一个字符是否为代理。可以直接分析字符串并执行测试来手工完成这项工作，但.NET Framework 提供了两个方法来完成这些繁琐的工作。

更重要的是，这些方法还可以用于在手工实现处理字符串的代码时组合字符。

2. StringInfo.ParseCombiningCharacters()

StringInfo 类提供了方法 ParseCombiningCharacters()，它可以把一个字符串分析为文本元素。.NET Framework 把一个文本元素定义为一个文本单元，并作为一个单字符输出。文本元素可以是基字符、代理对或一个组合字符序列。

3. StringInfo.GetTextEnumerator()

StringInfo 类提供了另一个方法 GetTextEnumerator()，它可以实现与 ParseCombiningCharacters()方法相同的结果：把文本分析为文本元素。StringInfo 类通过 StringInfo.GetTextEnumerator()方法提供了对 TextEnumerator 类的访问。

下载代码中的 StringWalk 示例演示了如何使用 ParseCombiningCharacters() 方法或 GetTextElementEnumerator()方法来分析字符串，并检索字符串中的所有元素，包括代理对和组合字符。

4.5 格式化 Unicode 字符串

String.Format()方法可以通过在一个格式字符串中提供格式规范和附加对象来格式化一个字符。它根据格式说明符来格式化字符。对于带有国际字符的字符串的正确格式化，这通常是必需的。例如，可以把下面一行代码：

```
String message = "This is my new " + item;
```

重写为：

```
String message = String.Format("this is my new {0}", item);
```

虽然对性能会有所影响，因为 String.Format()方法比先前使用的字符串连接代价更高，但它使代码更加容易本地化为另一种语言。例如，如果要把已修改的代码转换为日语，只需要把格式化字符串修改为“これは私の新しい{0} です”，代码就能够正确处理日语。

String.Format()方法还可以用来格式化数字值、日期以及时间和字符串。例如下面的代码：

```
string message;
message = String.Format("Date is {0}, value is {1}, word is {2}",
    DateTime.Now, 1234, "car");
```

其输出如下：

```
Date is 18/05/2002 14:24:05, value is 1234, word is car
```

当然，还可以使用先前已介绍的数字、日期和时间值的格式化说明符，所以，如果将下面代码行：

```
message = String.Format("Date is {0}, value is {1}, word is {2}",
        DateTime.Now, 1234, "car");
```

替换为：

```
message = String.Format("Date is {0:d}, value is {1:C}, word is {2}",
    DateTime.Now, 1234, "car");
```

其输出如下(如果当前文化设为 English (United States))：

```
Date is 5/18/2002, value is $1,234.00, word is car
```

另外，CultureInfo(或实现 IFormatProvider 接口的其他类)可以作为一个参数传递给 String.Format()方法的一个重写形式。如果修改代码，传递 Swedish (Sweden)文化：

```
message = String.Format(New CultureInfo("sv-SE"),
    "Date is {0:d}, value is {1:C}, word is {2}", DateTime.Now, 1234,
    "car");
```

就会得到如下结果：

```
Date is 2002-05-18, value is 1.234,00 kr, word is car
```

最后，如果创建控制台应用程序，代码为：

```
message = String.Format("This is a {0}", "car");
Console.WriteLine(message);
```

可以由下列代码代替，它在内部使用了 String.Format()方法。

```
Console.WriteLine("This is a {0}", "car");
```

4.6 字符串用作资源

在开发国际化的应用程序时，常常需要本地化用户界面，以便让各个国家和地区(如德国和日本)的用户都能以他们自己的语言使用相同的应用程序。

本地化的一个笨拙方法是搜索源代码中的所有字符串，用翻译过来的字符串代替它们，再重新编译应用程序。这种方法需要做大量的工作，非常容易出错。

比较好的方法是把代码与用户界面元素分隔开来，把它们存储在代码能以直接的方式使用的资源中。这些资源独立于表示它们的语言。这种方法使应用程序的翻译、测

试和迁移都很容易，效率也较高。

.NET Framework 提供了一个资源机制，它实现了这种方法，并提供了其他帮助。

.NET 资源模型

.NET Framework 为资源提供了一种非常灵活的模型来容纳字符串和对象(在.NET Framework 中，任何可以系列化的对象都是资源)，这样就可以用不同的方式创建和使用它们。其基本模型如下所述。

1. .resx 文件

首先，需要定义资源，以创建它们。这是在.resx 文件中进行的，.resx 文件是一个 XML 文件。定义资源的方式非常简单，只需要一个键值对就可以定义资源。对于字符串，值就是字符串的内容，而键是定义名称的字符串，以后通过该名称，资源可以在代码中引用。下面是.resx 文件中的资源定义示例：

```
<data name="myFirstString">
  <value>Hello, this is the first string</value>
</data>
<data name="mySecondString">
  <value>Hi, this is my second string</value>
</data>
```

注意，如果只把字符串用作资源，就可以使用一个快捷方式来定义资源。可以把字符串放在一个包含对应格式的文本文件中，并使用.NET Framework SDK 提供的 ResXGen 工具生成一个对应的.resx。如果使用 Visual Studio .NET，就只需给当前工程添加一个程序集资源文件，再在数据视图中为该文件输入资源即可。

2. .resources 文件

定义好资源后，就需要编译它们，以便在应用程序中使用。.resources 文件包含在.resx 文件中定义的资源的二进制表示。

这个编译过程由 resgen 实用程序在命令行完成，或由 Visual Studio .NET 环境在编译工程时直接实现。在这个阶段中，资源包含在.resources 文件中，它与主程序集是分离的，这在某些情况下是非常理想的，但在许多情况下，把资源嵌入主程序集更有意义。

3. 内嵌的资源

在主程序集中内嵌资源只是命令行上的额外一步，使用 CSC (C#)编译器或 AL(程序集链接器)实用工具就可以完成。在 Visual Studio .NET 中，这一步会自动完成，只要把默认的 BuildAction 属性设置为 Embedded Resources，就会把一个程序集资源文件添加到工程上。与使用松散资源文件相比，使用内嵌资源有两个主要优点：一是部署简

单，二是更新容易。在部署时，部署较小的文件总是比较容易，在主程序集中内嵌资源，就可以发布单个包含代码和资源的文件。对于更新，程序集是阴影复制，也就是说，在用户使用程序集的同时可以更新程序集(这对于 Web 应用程序来说特别重要)。但是，.resources 文件不是阴影复制，如果用户在使用文件时，该文件就被锁定，不能更新，那么这会使更新非常复杂。

了解了如何创建资源后，下面看看如何在代码中使用它们。

4. 使用资源

在大多数应用程序中，使用资源仅需构建一个 ResourceManager 类，再使用这个类提供的方法。ResourceManager 类包含两个检索资源的方法、构造函数，还包含一些静态方法，用于创建基于文件的资源管理器。

为了检索资源，首先需要创建 ResourceManager 类的一个实例，接着通过 GetString() 和 GetObject()方法检索资源就非常简单了。这两个方法都把一个键名作为输入，分别返回一个字符串和一个对象。

下面的例子说明了如何创建 ResourceManager，检索在上述.resx 文件中输入的两个字符串资源：

```
using System;
using System.Resources;
using System.Windows.Forms;
using System.Reflection;

namespace ResourcesDemo
{
  class Demo
  {
    [STAThread]
    static void Main(string[] args)
    {
      ResourceManager rm = new ResourceManager(
          "ResourcesDemo.strings", Assembly.GetExecutingAssembly());
      MessageBox.Show(rm.GetString("myFirstString"));
      MessageBox.Show(rm.GetString("mySecondString"));
    }
  }
}
```

这个应用程序显示两个消息框，如图 4-8 所示。

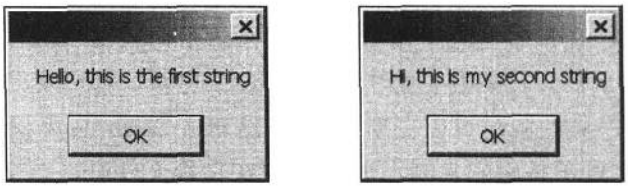


图 4-8

前面介绍了资源的用法，但没有讨论资源对本地化的益处，这是最初使用资源的原因。

5. 用资源进行本地化

使用资源对上述应用程序进行本地化的优点是，我们根本无需考虑代码。前面通过一个 ResourceManager 实例实现了资源的检索，下面就只需提供其他语言的资源，以获得应用程序的本地化版本。

创建该应用程序的法语版本，只需创建一个包含本地化字符串的.resx 文件，把.resx 文件编译为.resources 文件，再把.resources 文件内嵌到一个 DLL 中，这个 DLL 是只包含资源的程序集。

.NET Framework 为这些本地化资源使用一个命名约定(详见在线文档)，后面都遵循这个命名约定。

首先，创建一个.resx 文件 strings.fr.resx(最初的.resx 文件命名为 strings.resx)。其方法是复制已有的 strings.resx，再翻译字符串。现在就有了一个包含下述字符串的.resx 文件：

```
<data name="myFirstString">
  <value>Salut, ceci est ma première chaîne</value>
</data>
<data name="mySecondString">
  <value>Salut, ceci est ma seconde chaîne</value>
</data>
```

接着，Visual Studio .NET 会自动生成相应的.resources.dll 文件，并把它与主程序集一起放在 fr 文件夹中。如果在命令行上使用 C#，就需要使用 resgen 生成.resources 文件，再使用 AL 创建该 DLL。

如果在法语系统中使用该应用程序，就会看到它已经进行了本地化，如图 4-9 所示。

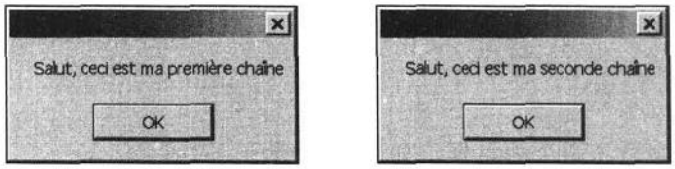


图 4-9

注意在英语系统中可以模拟这个过程，方法是在创建 ResourceManager 之前，用下面的代码把当前线程的 UICulture 设置为法国文化：

```
System.Threading.Thread.CurrentThread.CurrentUICulture =
    new System.Globalization.CultureInfo("fr");
```

.NET Framework 提供的资源模型还有其他优点，例如它提供了一个反馈机制，但这超出了本书的范围，详见.NET Framework SDK 文档。

4.7 小结

本章介绍了处理国际化的字符串和字符并不容易。.NET Framework 作了大量的工作为此提供支持，但对于某些情况仍需要非常小心。

如果您只想了解一些要点，就需要知道何时要处理文化和文化的敏感信息——特别是在处理与用户界面相关的操作时，如为在一个列表框中输出的字符串排序——同时要尽量避免处理单个字符，因为这将带来更高的复杂性。

第5章　正则表达式

正则表达式(regular expression，简写为 regexes)是用来操作和检验字符串数据的一种强大的工具。正则表达式是一串特殊的字符，它转换为某种算法，根据这个算法来匹配文本。对许多开发人员来说，正则表达式是新引入的概念，但在 Unix 系统甚至是 Web 脚本语言(如 JavaScript)中已经得到了广泛的应用。也许您对通配符，如 SQL 表达式中的“%”和“？”，或 MS DOS 中的“*”非常熟悉。通配符允许在查找时无需指定所有的字符，例如，“del *.txt”将删除所有扩展名为.txt 的文件。正则表达式提供了与通配符类似的方法，但功能要强大得多。

实际上，正则表达式有一套自己的简单语言，用于精确地描述要匹配的对象。只需使用一行代码正则表达式就可以匹配文本，如果不使用正则表达式可能就需要编写多行代码。但是，正则表达式也有它的缺点，它的创建可能比较复杂，而且含义比较模糊。但在这方面，它与许多成功的编程语言(如 Perl)是相同的。一旦习惯之后，正则表达式将非常容易使用。

使用正则表达式解决的最常见的问题是数据验证。ASP.NET 的 Web 窗体控件，如 Regular Expression Validator 控件，就是使用正则表达式来验证用户输入的数据。该控件使用一些标准正则表达式来验证邮政编码，E-mail 地址和 Internet 地址等内容。下一章将学习如何通过自定义的正则表达式验证自己的数据，尤其是因为 Validator 所提供的正则表达式无法覆盖到所有可能的情况。正则表达式允许根据字符模式进行匹配，如 3 个字母后跟 2 个数字，再后跟一个字母或另外两个数字。如果不使用正则表达式，这需要编写许多代码，但用正则表达式来实现就非常直接了。

正则表达式并不是在数据验证时才有用，在需要操作基于文本的数据的任何场合，都可以使用正则表达式。例如，有一个字符串包含了 HTML 标记，而我们要提取每一个标记，然后提取每一个标记的属性值。正则表达式使这项工作变得非常简单，只需要几行代码就能完成。

本章将循序渐进地介绍.NET 中的正则表达式，以及它所支持的正则表达式语法。下一章将介绍更高级的特性。

5.1　System.Text.RegularExpressions 命名空间

前面谈了很多理论，下面进入实践阶段。在.NET 中实现正则表达式的关键是 System.Text.RegularExpressions 命名空间，它包含了下面列出的 8 个类：

- Regex——包含了正则表达式，以及使用正则表达式的各种方法。
- MatchCollection——包含了一个正则表达式找到的所有匹配项。
- Match——包含了一次匹配中所有匹配的文本。
- GroupCollection——包含了一次匹配中的所有分组。
- Group——包含一个分组集合中一个组的细节。
- CaptureCollection——包含一个组的所有 Capture 对象。
- Capture——返回组内一次捕获所匹配的字符串。
- RegexCompilationInfo——提供了把 Regex 编译为一个独立程序集所需的细节。

上述类中最重要的是 Regex 类；其他的类提供了专门的实用函数，但并不常用。本章介绍类 Regex、Match 和 MatchCollection。它们的功能都由 System.dll 程序集提供。

5.2　Regex 类

Regex 类不仅可以用来创建正则表达式，而且提供了许多有用的方法，以使用正则表达式来操作字符串数据。例如，搜索字符模式或进行复杂的查找和替换。如果要把一个正则表达式重复用于不同的字符串，就可以创建一个 Regex 对象。但是，许多类的方法都是静态方法，也就是说，如果正则表达式只使用一次，那么直接使用所需方法要比创建一个 Regex 对象(类的一个实例)更加有效。如果要使用静态的 Replace()方法，可以编写如下代码：

```
Regex.Replace(string input,
              String pattern,
              string replacement);
```

Regex 类支持许多选项，这些选项改变了正则表达式语法工作的方式。表 5-1 给出了常用的一些选项，其他的选项将在接下来的两章介绍。

5.2.1　RegexOptions 枚举

表 5-1 列出了较常用的 Regex 选项。

表 5-1　Regex 类所支持的常用选项

标　志	描　述
IgnoreCase	使模式匹配忽略大小写。默认情况区分大小写
RightToLeft	从右到左查找输入字符串。默认的顺序是从左到右，符合英语等语言自左至右的阅读习惯，但不同于阿拉伯语和希伯来语自右至左的阅读习惯

（续表）

标　志	描　述
None	不设定标志。这是默认的选项，表示搜索是区分大小写的，且按从左到右的顺序进行
MultiLine	指定了^和$可以匹配行的开头和结尾，以及字符串的开头和结尾。也就是说，使用换行符分隔，在每一行都能得到不同的匹配。但是，字符“.”（在后面 5.5.1 节将介绍）仍然不匹配换行符
SingleLine	规定特殊字符“.”匹配任一字符，换行符除外。默认情况下特殊字符“.”不匹配换行符。常和选项 MultiLine 一起使用

只有掌握了正确的正则表达式语法，上面的一些选项才有意义。默认状态下所有的标志都没有设定。也就是说默认的查找是区分大小写的，查找顺序是从左到右。

可以使用 RegexOptions 枚举来设定标志，例如：

```
RegexOptions.IgnoreCase
```

把标志传递给正则表达式的构造函数，或在静态方法中，作为一个参数传递给方法：

```
Regex.IsMatch(MyString, "ABC",
              RegexOptions.IgnoreCase | RegexOptions.RightToLeft);
```

为了设置多个标志，只需使用“|”字符把每一个标志放在一起执行 OR 操作，因此上例中的标志选项设定为 IgnoreCase 和 RightToLeft。它执行了一次按位操作，返回方法能够正确解释的一个不同的值。要删除代码中所有的标志，可以使用 None。

5.2.2　类构造函数

这里介绍创建 Regex 对象的不同方法以及它们的参数，其中特别介绍了参数 RegexOptions，以及如何使用它来规定诸如查找顺序，是否忽略大小写等问题。

两个主要的构造函数为：

```
Regex(string pattern);

Regex(string pattern, RegexOptions options);
```

第一个参数是要匹配的正则表达式。举一个简单的例子，假设要匹配 ABC：

```
using System;
using System.Text.RegularExpressions;

namespace Wrox.Text.Chapter5
```

```
{
  class ABC
  {
    static void Main(string[] args)
    {
      Regex myRegEx = new Regex("ABC");
      Console.WritcLine(myRegEx.IsMatch("The first three letters of "
          + "the alphabet are ABC"));
    }
  }
}
```

这会输出 True，因为该句子中包含 ABC。

如果要匹配 ABC 或 abc 及其他任意的大小写组合，都可以设定不区分大小写选项标志并传递给构造函数来实现。

```
Regex myRegEx = new Regex("ABC", RegexOptions.IgnoreCase);
Console.WriteLine(myRegEx.IsMatch("Hello abc there"));
```

还需要注意，默认情况下 Regex 对象匹配的是 ASCII 文本。

5.2.3　IsMatch()方法

IsMatch()方法可以测试字符串，看它是否匹配正则表达式的模式。如果发现了一次匹配，那么该方法返回 True，否则返回 False。IsMatch()有一个静态的重载方法，使用它时可以无需显式地创建一个 Regex 对象。

该方法有 4 个重载形式：

```
public bool Regex.IsMatch(string input);

public bool Regex.IsMatch(string input,
                          int startat);

public static bool Regex.IsMatch(string input,
                                 string pattern);

public static bool Regex.IsMatch(string input,
                                 string pattern,
                                 RegexOptions options);
```

后两个重载方法是静态方法，而前两个用于对 Regex 对象实例执行操作。除了

RegexOptions 枚举(在前面的 5.2.2 节中介绍过它)外，后两个重载方法的参数都是字符串。int 型参数可以用来指定开始匹配的字符串位置。由此，测试一个字符串是否包含单词“Wrox Press”，应使用 IsMatch()方法：

```
string inputString = "Welcome to the publishers, Wrox Press Ltd";

if(Regex.IsMatch(inputString, "wrox press", RegexOptions.IgnoreCase))
{
  Console.WriteLine("Match Found");
}
else
{
  Console.WriteLine("No Match Found");
}
```

如果输入字符串包含了“wrox press”，执行上面的代码后会发现一次匹配。我们指定了选项 IgnoreCase，匹配将忽略大小写，所以“Wrox Press”也可以匹配。这里只使用了正则表达式一次，所以使用静态方法，从而不需要实例化一个新的 Regex 对象。如果要保存状态(正则表达式的细节，选项以及匹配的细节)，可以创建一个 Regex 对象。

5.2.4 Replace()方法

这个方法也有一个静态的重载方法，它用指定的字符串代替一个匹配模式。此方法有 10 种变体，这里只介绍基本的几种，其他更高级的选项将在下一章讲述。

作为一个静态方法，它有以下的重载方法：

```
public static string Regex.Replace(string input,
                                   string pattern,
                                   string replacement);

public static string Regex.Replace(string input,
                                   string pattern,
                                   string replacement,
                                   RegexOptions options);
```

使用静态方法，如果想用“Wrox Press”替换“wrox press”的所有实例，可以使用如下代码：

```
string inputString = "Welcome to the publishers wrox press ltd";
inputString = Regex.Replace(inputString, "wrox press", "Wrox Press");
```

```
Console.WriteLine (inputString);
```

如果使用非静态方法，可以指定替换次数的最大值以及开始下标：

```
Public string Replace(string input,
                      string replacement);

Public string Replace(string input,
                      string replacement,
                      int count);

Public string Replace(string input,
                      string replacement,
                      int count,
                      int startat);
```

如果要在使用其中的一个重载方法时替换所产生的所有匹配，可以把 -1 传递给参数 count。

例如，字符串是"123,456,123, 123,789,123,888"，要使用 xxx 替换 456 之后的 123，但最多只替换两次，代码如下：

```
string inputString = "123,456,123,123,789,123,888";
Regex regExp = new Regex("123");
inputString = regExp.Replace(inputString, "xxx", 2, 4);

Console.WriteLine(inputString);
```

其输出如下：

```
123,456,xxx,xxx,789,123,888
```

第一个 123 不会被替换，因为最后一个参数值是 4，表示搜索是从下标为 4 的字符开始进行，也就是 456 中的 4。在 456 和 789 之间的两个 123 将替换为 xxx。但是，最后一个 123 也不会替换，因为第三个参数指定了最多只做两次替换。

5.2.5　Split()方法

此方法在每次发现匹配的位置拆分字符串。它返回一个字符串数组。该方法也有静态的重载方法，以及用于 Regex 实例的方法。两个静态方法为：

```
Public static string[] Split(string input,
                             string replacement);
```

```
Public static string[] Split(string input,
                             string pattern,
                             RegexOptions options);
```

假设有一个使用逗号定界符的字符串，可以用如下代码拆分字符串：

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

namespace Wrox.Text.Chapter5
{
  class Split
  {
    static void Main(string[] args)
    {
      string inputString = "123,ABC,456,DEF,789";
      string[] splitResults;
      splitResults = Regex.Split(inputString, ",");
      StringBuilder resultsString = new StringBuilder(32);

      foreach (string stringElement in splitResults)
      {
        resultsString.Append(stringElement + "\n");
      }

      MessageBox.Show(resultsString.ToString());
    }
  }
}
```

这段代码将会产生如图 5-1 所示的消息框。

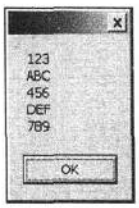


图 5-1

字符串在逗号处被拆分，但不包括逗号，每一个拆分出来的字符串都放在一个数组元素中。

该方法的公有非静态方法类似于 Replace( )方法的共享方法，利用它们可以指定拆分的最大次数和模式匹配的开始位置：

```
Public string[] Split(string input);

Public string[] Split(string input,
                      int count);

Public string[] Split(string input,
                      int count,
                      int startat);
```

如果把 0 作为最大拆分次数的参数 count 的值，将对所有匹配执行拆分操作，这与默认的操作相同。

5.3　Match 类和 MatchCollection 类

利用 Match 类和 MatchCollection 类，可以获得通过一个正则表达式实现的每一次匹配的细节。Match 类表示一次匹配，而 MatchCollection 类是一个 Match 对象的集合，其中每一个对象都代表了一次成功的匹配。我们需要使用 Regex 对象的 Match()或 Matches()方法来检索匹配。

首先介绍 Regex.Match()方法，它有以下的重载方法：

```
public Match Match(string input);

public Match Match(string input,
                   int startat);

public Match Match(string input,
                   int startat,
                   int length);

public static Match Match(string input,
                   string pattern);

public static Match Match(string input,
                   string pattern,
                         RegexOptions options);
```

前3个方法是实例方法，后两个是静态方法。所有方法都会返回一个Match对象，其中包含了匹配的细节。注意，返回的Match对象只代表了实现的第一次匹配，而不是所有的匹配。

下面的示例使用了上述第二个重载方法：

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

namespace Wrox.Text.Chapter5
{
  class MatchMethod
  {
    static void Main(string[] args)
    {
      string inputString = "A sailor went to sea to sea, "
          + "to see what he could see could see.";

      Regex myRegex = new Regex("se.");
      Match matchMade = myRegex.Match(inputString, 20);

      while (matchMade.Success)
      {
        MessageBox.Show(matchMade.Value);
        matchMade = matchMade.NextMatch();
      }
    }
  }
}
```

输入字符串取自一首民歌的歌词，歌词中重复的单词有助于强调要表达的意义。正则表达式是“se.”。字符“.”在下文中会介绍，它可以匹配任何字符。此正则表达式通常将匹配所有以se打头的串，如see或sea。但是，当使用Regex类的Match( )方法时：

```
Match matchMade = myRegex.Match(inputString, 20);
```

我们指定了匹配从第21个字符开始。此外，它只发现第一个匹配，在本例中是第二个“sea”。虽然本例中在进行匹配之前我们就能看出将有一次成功匹配，但在实践中常常无法事先得知，此时应使用Match类的Success属性。在一个While循环中使用Success属性，在一次匹配成功后，将继续循环。

```
while (matchMade.Success)
```

```
{
  MessageBox.Show(matchMade.Value);
  matchMade = matchMade.NextMatch();
}
```

使用 Match.NextMatch()方法可以得到下一次匹配。该方法将从 Match 对象的最近一次成功匹配后的字符开始搜索。表 5-2 和表 5-3 详述了 Match 类的一些更有用的方法和属性。

表 5-2　Match 类的一些实用方法

方　法	描　述
NextMatch()	返回一个新的 Match 对象，它带有下一次匹配的结果。从上一次匹配结束的位置开始执行(最后一个匹配字符的下一个字符)
Result()	返回传递的替换模式的展开式。和下一章所述的分组一起使用。它将对所有不同正则表达式的替换模式进行分组，并作为一个整体返回
Synchronized()	返回一个 Match 实例，等价于一个可以在多线程应用程序中的多线程间安全共享的实例

表 5-3　Match 类的一些实用属性

属　性	描　述
Captures	返回一个捕获对象集合，每个对象代表捕获组所匹配的每一次捕获。分组和捕获将在下一章中介绍
Groups	返回一个分组对象集合，每个对象代表被正则表达式匹配的每一个 Regex 组
Index	在输入字符串中第一个匹配字符的位置
Length	匹配的长度
Success	根据匹配是否成功返回 True 或 False
Value	从输入字符串中获得匹配的子串

最后学习 MatchCollection 类。使用 Regex.Matches()方法，可以得到 MatchCollection 对象的一个引用。这个集合类将包含分别代表每一次正则表达式匹配的 Match 对象。它在处理多次匹配时尤其有用，而且可以代替 Match.NextMatch()方法。MatchCollection 类有两个有用的属性 Count 和 Item。Count 返回匹配的个数，Item 允许通过下标或使用 foreach 循环访问集合中的每一个 Match 对象。

要得到一个 MatchCollection，需要使用 Regex.Matches()方法，它有如下重载方法，它们都返回一个 MatchCollection 对象：

```
public MatchCollection Matches(string input);
```

```
public MatchCollection Matches(string input,
                               int startat);

public static MatchCollection Matches(string input,
                                      string pattern);

public static MatchCollection Matches(string input,
                                      string pattern,
                                      RegexOptions options);
```

下例使用了静态的 Matches( )方法来访问第三次匹配，即 see：

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

namespace Wrox.Text.Chapter5
{
  class MatchCollectionClass
  {
    static void Main(string[] args)
    {
      string userInputString = "A sailor went to sea to sea, "
          + " to see what he could see could see.";
      MatchCollection matchesMade =
          Regex.Matches(userInputString, "se.");

      if(matchesMade.Count >= 3)
      {
        MessageBox.Show(matchesMade[2].Value);
      }
    }
  }
}
```

我们把 matchesMade 定义为一个 MatchCollection 对象，然后把它设置为 Regex 方法 Matches( )返回的 MatchCollection 对象。这里使用了静态的 Matches()方法，而没有显式创建一个 Regex 对象。

然后，使用 Count 属性进行检查，发现有 3 次或更多次的匹配，输出第三次匹配的细节，也就是集合中下标为 2 的对象。

5.4　Regex 测试器示例

在上一节中，介绍了 Regex 对象的一些方法。本小节将创建一个应用程序，它将使正则表达式的创建和测试非常简单。虽然只是一个简单的示例，但它也是非常有用的，而且能用来测试后续章节中的正则表达式。图 5-2 给出了此应用程序的界面。

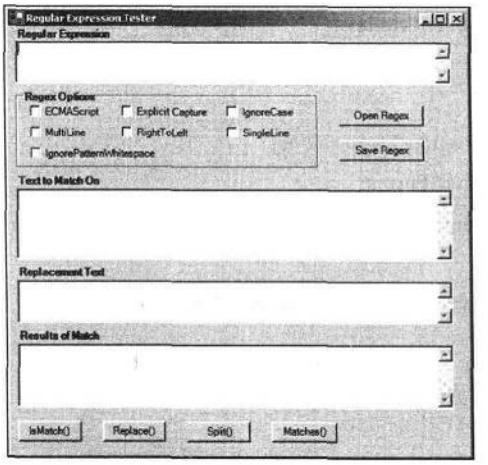


图　5-2

此程序允许输入正则表达式，设定所有的选项，然后使用方法 IsMatch()、Replace()、Split()和 Matches()进行测试。

例如，如果在正则表达式文本框中输入\d\d\d[a-z]，选中 IgnoreCase 复选框，在 Text to Match On 文本框中输入 456A 321B 789 78AA 444E，并单击 Match( )按钮，会看到如图 5-3 所示的输出。

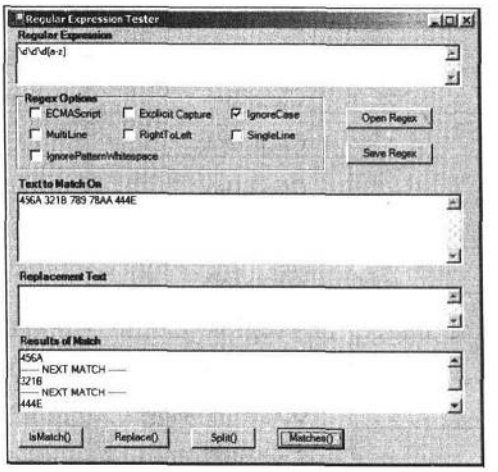


图　5-3

稍后会介绍正则表达式语法\d和[a-z]，但从正则表达式测试器中可以推断出，\d 用来匹配一个数字字符，[a-z]匹配 a 到 z 之间的任何一个字母。设定 IgnoreCase 标志规定匹配不区分大小写。下一章只介绍选项 MultiLine、IgnoreCase 和 SingleLine。

需要创建一个新的 Windows 应用程序(RegexTester)，并更改初始的类定义，其代码如下所示：

```
using System;
using System.ComponentModel;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;
using System.Windows.Forms;
using System.Drawing;

public class RegexTesterForm : System.Windows.Forms.Form
{
```

表 5-4 列出了要添加到窗体的控件。

表 5-4　要向窗体添加的控件

名　称	控件类型	文　本
TestRegexButton	Button	IsMatch()
ReplaceButton	Button	Replace()
SplitButton	Button	Split()
MatchesButton	Button	Matches()
OpenRegexButton	Button	Open Regex
SaveRegexButton	Button	Save Regex
OptionsGroup	GroupBox	Regex Options
SingleLineChkBox	CheckBox	SingleLine
RightToLeftChkBox	CheckBox	RightToLeft
MultiLineChkBox	CheckBox	MultiLine
IgnorePatternWhiteSpaceChkBox	CheckBox	IgnorePatternWhiteSpace
IgnoreCaseChkBox	CheckBox	IgnoreCase
ExplicitCaptureChkBox	CheckBox	ExplicitCapture
ECMAScriptChkBox	CheckBox	ECMAScript
openFileDialog1	OpenFileDialog	N/A
saveFileDialog1	SaveFileDialog	N/A

对于表 5-5 中的控件，把 MultiLine 属性设置为 True，把 Scrollbars 属性设置为 Vertical。

表 5-5　附加控件

名　称	控 件 类 型
RegexTextBox	TextBox
InputTextBox	TextBox
ReplacementTextBox	TextBox
ResultsTextBox	TextBox

在图 5-3 中，可以看到添加上表中的附加控件以及标记控件的位置，以及需要加粗的文本。Open Regex 和 Save Regex 按钮是为了方便而设置的，它们能保存更复杂的正则表达式。我们从添加代码以保存正则表达式开始。双击 Save Regex 按钮，并修改它的 Click 事件处理程序：

```
private void SaveRegexButton_Click(object sender. System.EventArgs e)
{
  saveFileDialog1.ShowDialog();
}
```

这会显示 Save File 对话框，用户可以为保存正则表达式的文本文件选择驱动器、目录和文件名。如果用户单击 OK 来保存文件，就触发 FileOK 事件，所以这里添加保存文件的代码。双击 SaveFileDialog1 按钮，并修改它的事件处理程序：

```
private void saveFileDialog1_FileOk(object sender,
        System.ComponentModel.CancelEventArgs e)
{
  StreamWriter streamWriterRegex =
      File.CreateText(saveFileDialog1.FileName);
  streamWriterRegex.Write(this.RegexTextBox.Text);
  streamWriterRegex.Close();
}
```

从用户选择的 SaveFileDialog.FileName 属性中获取文件路径，然后使用 StreamWriter 类创建一个新的文本文件。使用它的 Write()方法把正则表达式保存到文件中。

打开一个正则表达式，涉及一个与保存它类似的方法。首先，需要添加代码，以便在单击 OpenRegexButton 按钮时显示 open 对话框：

```
private void OpenRegexButton_Click(object sender,
                                   System.EventArgs e)
```

```
{
  openFileDialog1.ShowDialog();
}
```

如果接着单击 OK 按钮，就会触发 OpenFileDialog1 的 FileOk 事件。现在，插入从文件中读取正则表达式的代码：

```
private void openFileDialog1_FileOk(object sender,
        System.ComponentModel.CancelEventArgs e)
{
  StreamReader streamReaderRegex =
      File.OpenText(openFileDialog1.FileName);
  this.RegexTextBox.Text = streamReaderRegex.ReadToEnd();
  streamReaderRegex.Close();
}
```

这与保存正则表达式的方法相似，但这次使用的是 StreamReader 对象从文本中读取，并把 txtRegex.Text 属性设为 Stream Reader 对象的值。

刚才介绍了正则表达式的打开和保存操作，现在看看如何根据用户选取的复选框确定需要设定的 Regex 选项。因为这是所有的 Regex 测试按钮都需要做的设置，所以创建窗体类的一个私有方法 GetSelectedRegexOptions()，它由其他按钮的 Click 事件调用：

```
private RegexOptions GetSelectedRegexOptions()
{
  RegexOptions selectedRegexOptions = RegexOptions.None;

  if(this.IgnoreCaseChkBox.Checked)
      selectedRegexOptions |= RegexOptions.IgnoreCase;

  if(this.ExplicitCaptureChkBox.Checked)
      selectedRegexOptions |= RegexOptions.ExplicitCapture;

  if(this.ECMAScriptChkBox.Checked)
      selectedRegexOptions |= RegexOptions.ECMAScript;

  if(this.IgnorePatternWhitespaceChkBox.Checked)
      selectedRegexOptions |= RegexOptions.IgnorePatternWhitespace;

  if(this.MultiLineChkBox.Checked)
      selectedRegexOptions |= RegexOptions.Multiline;
  if(this.RightToLeftChkBox.Checked)
```

```
        selectedRegexOptions |= RegexOptions.RightToLeft;

    if(this.SingleLineChkBox.Checked)
        selectedRegexOptions |= RegexOptions.Singleline;

    return selectedRegexOptions;
}
```

此方法只是用于选中每一个复选框的 if 语句列表。如果选中了一个复选框，那么它将与 RegexOption 一起执行 OR 运算。

顶部的代码如下：

```
RegexOptions selectedRegexOptions = RegexOptions.None;

if(this.IgnoreCaseChkBox.Checked)
  selectedRegexOptions |= RegexOptions.IgnoreCase;
```

把变量 selectedRegexOptions 定义为 RegexOptions 类型，它将保存选项集的最终记数。最初把它设置为 RegexOptions.None，表示没有设置任何选项。

在 If 语句中，检查用户是否选中了复选框 chkIgnoreCase，如果是，就使用一个逻辑 OR 赋值运算符把 selectedRegexOptions 设置为 IgnoreCase，使 RegexOptions 包含选项 IgnoreCase (|=)。注意，这里用的是逻辑 OR 而不是布尔 OR。它增加了已设定的现有选项，但并不删除这些选项。

函数的其他部分以相同的方式执行，直到所有的复选框和 Regex 选项都已按照用户请求进行了设定。然后返回保存在 selectedRegexOptions 中的结果。

每一个按钮都与在顶部框输入的一个正则表达式的方法名称相同。下面先介绍最简单的一个按钮 IsMatch()。把下面的代码添加到它的 Click 事件中：

```
private void TestRegexButton_Click(object sender,
                                   System.EventArgs e)
{
  try
  {
    RegexOptions selectedRegexOptions =
        this.GetSelectedRegexOptions();
    Regex testRegex = new Regex(this.RegexTextBox.Text,
        selectedRegexOptions);

    if(testRegex.IsMatch(this.InputTextBox.Text))
    {
      this.ResultsTextBox.ForeColor = Color.Black;
      this.ResultsTextBox.Text = "MATCH FOUND";
```

```
        }
        else
        {
          this.ResultsTextBox.Text = "NO MATCH FOUND";
          this.ResultsTextBox.ForeColor = Color.Red;
        }
      }

      catch (ArgumentException ex)
      {
        this.ResultsTextBox.ForeColor = Color.Red;
        this.ResultsTextBox.Text =
            "There was an error in your regular expression:\r\n"
            + ex.Message;
      }
    }
```

代码放在一个 try...catch 语句中，这在这种情况下尤其有用，因为如果使用正则表达式时出现了语法错误，Catch 子句就捕获异常 ArgumentException，并把它输出到 ResultsTextBox。出错消息通常会给出足够的细节，提示正则表达式语法出错的位置。

在 try 子句的开头，我们创建了一个新的变量，来保存 RegexOptions 枚举，并使用上面创建的 GetSelectedRegexOptions()方法把它初始化为用户选择的选项。

然后，使用 IsMatch()方法查看正则表达式在 txtInputText 文本框中是否有匹配。如果发现了一个匹配，就返回 True，并把 ResultsTextBox 设置为"MATCH FOUND"。如果没有发现匹配，则输出红色字母"NO MATCH FOUND"。

现在看看 Replace()按钮以及它的 Click 事件代码：

```
    private void ReplaceButton_Click(object sender, System.EventArgs e)
    {
      try
      {
        RegexOptions selectedRegexOptions =
            this.GetSelectedRegexOptions();
        Regex replaceRegex = new Regex(this.RegexTextBox.Text,
            selectedRegexOptions);

        this.ResultsTextBox.ForeColor = Color.Black;
        this.ResultsTextBox.Text = replaceRegex.Replace(
            this.InputTextBox.Text, this.ReplacementTextBox.Text);
      }
```

```
        catch (ArgumentException ex)
        {
          this.ResultsTextBox.ForeColor = Color.Red;
          this.ResultsTextBox.Text = "There was an error in "
              + "your regular expression:\r\n" + ex.Message;
        }
      }
```

这里再次使用一个 try...catch 语句，来捕获正则表达式语法中的错误，并输出到 ResultsTextBox。在 try 子句中，我们获得了 RegexOptions，并使用 RegexTextBox 文本框中的正则表达式创建了一个新的 Regex 对象。但是接下来，使用 Replace()方法，用 ReplacementTextBox 文本框内的文本替换正则表达式在 txtInputText 的文本中实现的匹配，然后在 ResultsTextBox 文本框中输出结果。

接下来要介绍的是 Split( )按钮以及它的 Click 事件处理代码：

```
private void SplitButton_Click(object sender, System.EventArgs e)
{
  try
  {
    RegexOptions selectedRegexOptions =
        this.GetSelectedRegexOptions();
    Regex splitRegex = new Regex(this.RegexTextBox.Text,
        selectedRegexOptions);

    String[] splitResults;
    splitResults = splitRegex.Split(this.InputTextBox.Text);
    StringBuilder resultsString = new
        StringBuilder(this.InputTextBox.Text.Length);

    foreach (String stringElement in splitResults)
        resultsString.Append(stringElement + Environment.NewLine);

    this.ResultsTextBox.ForeColor = Color.Black;
    this.ResultsTextBox.Text = resultsString.ToString();
  }

  catch (ArgumentException ex)
  {
    this.ResultsTextBox.ForeColor = Color.Red;
```

```
            this.ResultsTextBox.Text = "There was an error in "
                + "your regular expression:\r\n" + ex.Message;
        }
    }
```

这里的 try...catch 子句和上文所述的作用相同，它捕获正则表达式中的所有错误。接着得到所选择的 RegexOptions，并使用它和正则表达式字符串创建一个新的 Regex 对象。

我们声明一个数组 splitResults，它保存 Split()方法返回的结果，把每一个经划分的数据项作为字符串数组中的一个字符串返回。然后，对 Regex 对象执行 Split()方法，并把结果存储在 splitResults 数组中。

得到了保存在数组中的结果后，现在需要使用 foreach 循环遍历每一个数组元素，并把结果追加到 resultsString StringBuilder 对象中。该对象初始化为经划分的字符串的大小，减少了内存分配的空间。

```
            foreach (String stringElement in splitResults)
                resultsString.Append(stringElement + Environment.NewLine);
```

最后，把结果输出到文本框 ResultsTextBox 上：

```
            this.ResultsTextBox.Text = resultsString.ToString();
```

要介绍的最后一个按钮是 Matches()以及它的事件代码：

```
    private void MatchesButton_Click(object sender, System.EventArgs e)
    {
      try
      {
        RegexOptions selectedRegexOptions = GetSelectedRegexOptions();
        Regex matchesRegex = new Regex(this.RegexTextBox.Text,
            selectedRegexOptions);

        MatchCollection matchesFound;
        matchesFound = matchesRegex.Matches(this.InputTextBox.Text);

        String nextMatch = "------- NEXT MATCH -------\r\n";
        StringBuilder resultsString = new StringBuilder(64);

        foreach (Match matchMade in matchesFound)
            resultsString.Append(matchMade.Value
                + (Environment.NewLine + nextMatch));
```

```
            this.ResultsTextBox.ForeColor = Color.Black;
            this.ResultsTextBox.Text = resultsString.ToString();
        }

        catch (ArgumentException ex)
        {
            this.ResultsTextBox.ForeColor = Color.Red;
            this.ResultsTextBox.Text = "There was an error in "
                + "your regular expressions:\r\n" + ex.Message;
        }
    }
```

和先前的单击事件一样，我们还是使用了一个 try...catch 语句来捕获创建新 Regex 对象时所出现的正则表达式异常。但是，这里想获得的是正则表达式每一次匹配的文本。为此，需要使用对象 MatchCollection 和 Match 来保存每一次匹配的文本。变量 matchesFound 设置为 Matches()方法所返回的 MatchCollection 对象。这个集合对象包含了每一次匹配的 Match 对象。使用一个 foreach 循环来遍历此集合，在循环中把每一个 Match.Value 属性都追加到 resultsString StringBuilder 对象之后，并使用一个字符串来表示匹配的界限。把 Environment.NewLine 和 nextMatch 组合起来，因为这将增加字符串高速缓存的机会，不需要创建一个单独的字符串对象来标记删除，在内容放到 StringBuilder 对象中后，垃圾回收器会立即回收它们。

最后，代码把结果输出到 ResultsTextBox 文本框中。

5.5　正则表达式基础语法

到目前为止，除了两次小的例外，我们只使用 Regex 方法进行了简单的正则表达式字符匹配。例如，匹配字符串“ABC”。但是，这些操作所实现的功能与 String 类及其对应方法也没什么两样。如果只想匹配这样简单的字符，使用 String 类的方法会更加高效；Regex 方法更强大，但也意味着需要更强大的处理能力。

但是，正则表达式的强大不是表现在特定的字符匹配上，而是在字符类型的模式匹配上。例如匹配空字符、字母、数字、发音字符和字母或数字的范围。

5.5.1　匹配不同类型的字符

表 5-6 列出了正则表达式语法中的特殊字符类。这些特殊字符可以替代一个字符，比如，\d 可以匹配任何数字。

表 5-6 以正则表达式语法表示的字符类

字 符 类	匹配的字符	示 例
\d	从 0～9 的任一数字	\d\d 可以匹配 72，但不匹配 aa 或 7a
\D	非数字字符	\D\D\D 匹配 abc，但不匹配 123
\w	任一单词字符，如 A-Z、a-z、0-9，和下划线字符	\w\w\w\w 匹配 Ab_2，但不匹配£$%* 或 Ab_@
\W	非单词字符	\W 可以匹配@，但不匹配 a
\s	任一空白字符，包括了制表符，换行符，回车符，换页符和垂直制表符	匹配所有传统的空字符，包括用 HTML，XML 和其他标准定义的字符
\S	任一非空白字符	每一个非空字符：A%&g3;等
.	任一字符	“.”匹配任一字符，换行符除外
[...]	括号中的任一字符	[abc]匹配单个字符 a、b 或 c，但不能匹配其他字符 [a-z]匹配 a～z 的任一字符
[^...]	非括号中的任一字符	[^abc]匹配 a、b、c 除外的任一字符，但可以匹配 A、B 或 C [^a-z]匹配非 a～z 的任一字符，但可以匹配所有的大写字母

要匹配格式为 1-800-888-5474 的电话号码，正则表达式应该为：

```
\d-\d\d\d-\d\d\d-\d\d\d\d
```

\d 匹配数字类中的任一字符，“-”匹配一个破折号。可以使用下面的代码做一个测试：

```
Console.WriteLine(Regex.IsMatch("1-800-888-5474",
    @"\d-\d\d\d-\d\d\d-\d\d\d\d"));
```

注意在正则表达式字符串的前面插入了一个@字符。这是因为 C#可以识别字符串中的一些转义字符，例如\n 表示换行，当遇到\d 字符时，C#会报告一个错误。没有引用的字符串字面字符，就必须在每个正则表达式转义字符的前面插入额外的一个\字符。运行这段代码，将会得到一个显示 True 的对话框以表示有效。如果改变了测试的字符串：

```
Console.WriteLine(Regex.IsMatch("X-800-888-5474",
    @"\d-\d\d\d-\d\d\d-\d\d\d\d"));
```

就会显示 False。当然，也可以在新的正则表达式测试应用程序中测试这个正则表

达式。要匹配一个格式为“Oct 31 2002”的日期，正则表达式应该为：

```
[a-zA-Z][a-zA-Z][a-zA-Z] \d\d \d\d\d\d
```

[a-zA-Z]匹配从 a 到 z 的任一大写或小写字母。所以，这里要匹配的模式为 a 到 z 或 A 到 Z 之间的 3 个字母，后面是一个空格，两个数字，再跟一个空格，最后是 4 个数字。

在这种情况下，最好使用选项 IgnoreCase，进行不区分大小写的匹配，这样就可以用[a-z]或[A-Z]来代替[a-zA-Z]。

如果要在日期数字和空格中间允许破折号出现，如“Oct 31 2002”，就应修改正则表达式：

```
[a-z][a-z][a-z][ -]\d\d[ -]\d\d\d\d
```

注意，前面的正则表达式中有许多的重复字符。在上面的正则表达式中，出现了 3 个[a-z]，中间出现了两个\d，最后又出现了 4 个\d。正则表达式语法有专门的重复字符来处理这种情况，这正是稍后一小节“指定重复字符”要讨论的主题。

注意：

在代码中使用正则表达式时，@字符的一种替代方法是在所有以\开头的正则表达式前面加上额外的一个\，这样 C#就不会认为这是一个字符串转义字符了，而不是把它当做一个正则表达式字符。所以\d 应写为\\d，\s 应写为\\s，等等。这同样适用于本书后面介绍的正则表达式语法。

5.5.2　指定匹配位置

现在介绍在正则表达式中，如何指定匹配开始、结束和在中间的位置。例如，使用^可以指定匹配必须从字符串的开头进行，或使用\b 指定单词的范围。前面的正则表达式允许在匹配字符之前或之后的任意位置开始进行匹配。

定位字符允许规定匹配模式在字符串中发现的位置。这个操作称为“锚定”模式。也就是说，锚定可以使正则表达式的一端固定在字符串中要进行匹配的一个点上。所以，进行锚定操作可以防止对整个字符串进行匹配操作。

表 5-7 列出了 8 个特殊的定位字符。

表 5-7　定位字符

定位字符	描　述
^	其后的模式必须在字符串的开始处，如果是一个多行字符串，应位于任一行的开始。对于多行文本(包含回车符的字符串)，需要设定 Multiline 标志

(续表)

定位字符	描述
$	前面的模式必须在字符串的末尾处，如果是一个多行字符串，应该在任一行的末尾
\A	前面的模式必须在字符串的开始处；多行标志被忽略
\z	前面的模式必须在字符串的末尾处；多行标志被忽略
\Z	前面的模式必须位于字符串的末尾，或是位于换行符前
\b	匹配一个单词边界，实质上是单词字符和非单词字符间的点。单词字符是[a-zA-Z0-9_]中的任一字符。位于一个单词的开始
\B	匹配一个非单词边界的位置，不在一个单词的开始

如果使用正则表达式对用户数据进行验证，通常需要确定用户只输入了我们想要的数据，而且在数据之前和之后没有其他数据。例如，“1234 4567 1234 1232”是一个有效的信用卡号码。但是，如果输入 XXXX1234 4567 1234 1232，这个输入就是无效的。要解决这个问题，可以在正则表达式的开始使用一个“^”字符以保证卡号出现在字符串的开始位置，使用$字符以确保模式出现在字符串的结束位置。添加了这些字符，就可以保证卡号模式既是字符串的开始，又是字符串的结尾，因而是字符串惟一的内容。所以，用来匹配卡号的正则表达式变为：

```
^\d\d\d\d \d\d\d\d \d\d\d\d \d\d\d\d$
```

这个正则表达式使用了特殊字符“^”和“$”，因而对选项 MultiLine 的设定就变得很重要。如果设定了 MultiLine，那么“^”除了匹配字符串的开始位置，还可以匹配后跟\r 或\n (Cr/Lf)的位置，使用“$”除了可以匹配字符串的结束位置，还可以匹配\n 或\r 前面的位置。

如果没有设定 MultiLine，"^"将只匹配一个字符串开始的位置，$只匹配一个字符串结束的位置，最多匹配第一个\n 或\r。重复字符在上一小节电话号码匹配的例子中已经提到。接下来描述它们。

5.5.3 指定重复字符

重复字符不但可以压缩正则表达式，而且可以指定一个字符以及字符组出现的频率。下面先了解基本的重复字符。如表 5-8 所示。

表 5-8　基本的重复字符

重复字符	含　义	举　例
{n}	匹配前面的字符 n 次	x{2}匹配 xx，但不匹配 x 或 xxx
{n,}	匹配前面的字符 n 次或更多	x{2,}匹配 2 个或更多的 x，如 xx, xxx, xxxx, xxxxx, ...
{n,m}	匹配前面的字符最少 n 次，最多 m 次。如果 n 为 0，可以不指定	x{2,4}匹配了 xx，xxx 和 xxxx，但不匹配 x 或 xxxxx
?	匹配前面的字符 0 次或 1 次，可以省略	x?匹配 x 或空
+	匹配前面的字符 1 次或多次	x+匹配 x，或 xx，或更多的 x
*	匹配前面的字符 0 次或多次	x*匹配 0 个或多个 x

学习了上表和前一节关于定位正则表达式的内容，下面修改先前的电话号码和日期的正则表达式。下面是原来的电话号码正则表达式：

```
\d-\d\d\d-\d\d\d-\d\d\d\d
```

可以修改为：

```
^\d-\d{3}-\d{3}-\d{4}$
```

修改过的正则表达式将精确地匹配相同的模式，读取一个数字，后跟 1 个破折号，3 个数字，1 个破折号，3 个数字，一个破折号，最后是 4 个数字。这样也实现了对字符串的定位，所以在模式的两端不会出现其他字符。

下面是短日期匹配表达式：

```
[a-zA-Z][a-zA-Z][a-zA-Z][ -]\d\d[ -]\d\d\d\d
```

可以修改为：

```
^[a-zA-Z]{3}[ -]\d\d[ -]\d{4}$
```

我们没有把表达式中间的\d\d 修改为\d{2}，因为修改后增加了一个字符，而增加表达式的长度是无意义的。

比{n}更有用的是特殊的重复字符“?”、“+”和“*”。比如说要匹配一个 16 位的信用卡号码。用户输入卡号时，有时输入完整的号码而不插入空格，有时用空格把号码以 4 个数字为单位进行分组，就像卡上所输出的格式一样。这就提出了一个问题，在执行正则表达式模式匹配前，需要使用 Replace()方法去除空格，或者使用两个正则表达式来匹配每一种情况，如下所示：

```
\d{16}
```

和

```
\d{4} \d{4} \d{4} \d{4}
```

使用重复字符“?”(它匹配前一个字符 0 次或 1 次)，就可以把两个正则表达式组合在一起：

```
^\d{4} ?\d{4} ?\d{4} ?\d{4}$
```

在数字的空格后跟?，表示空格必须出现 0 次或 1 次。除此之外，只有通过两端定位使模式与字符串完全相同，才能够实现匹配。

贪婪和懒惰的表达式

前面所有的限定词都可以用“贪婪”来形容，换句话说，正则表达式引擎将会尽可能地匹配更多的字符。我们可以这样理解，正则表达式引擎遇到一个重复字符时，只要表达式的特定成分允许，它将从左到右开始搜索。例如，\d*3 将进行数字匹配，一直到没有更多的数字能够匹配为止。获得了尽可能多的数字后，引擎将试图匹配 3。如果找不到，或不能立即找到(因为它已经匹配了所有的数字)，就从上一次匹配中删除一个字符并再作一次尝试，这个过程会一直重复，直到匹配到 3 为止，然后接着进行。从这个例子中可以看出来，只要指定了数量，引擎就会首先尽可能多地获得字符，然后当无法匹配一个字符时，才吝惜地把字符一次一个地释放掉。可以在图 5-4 中看到此工作流程。它使用了正则表达式\d*3，并试图在字符串“123456789fgfd”中进行匹配。

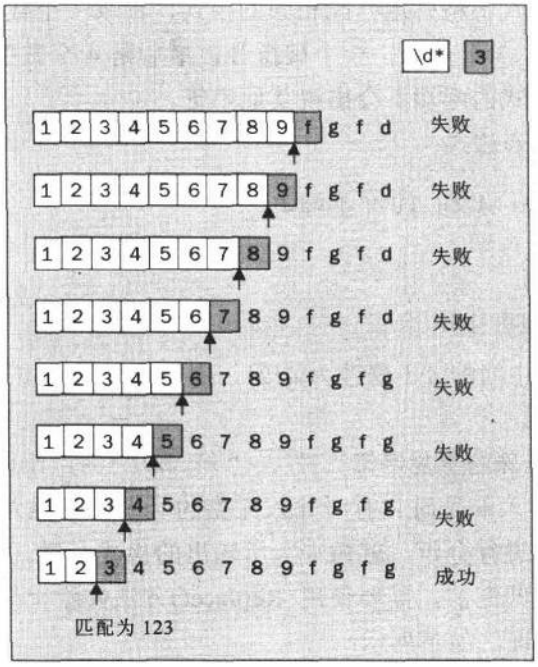


图　5-4

每个决策点放在一个正则表达式引擎使用的内部栈中。这意味着一旦失败，引擎会返回到上一个决策点并进行下一次决策。在图 5-4 中，就是每次试图少匹配一个数字。如果所有的选择都考虑过了，栈顶元素被弹出，并返回到上一个决策点。如果栈中没有决策点了，就代表没有可匹配的元素，IsMatch()方法返回 False。

说明此操作的最好方法是举一个例子。下面是一个不同的例子——匹配 ISBN。出版商和个人经常使用 ISBN 的不同形式。它们之间相同的地方在于都使用了 10 个数字，而且最后一位是“X”(最后一位是校验数字，因此可能有 11 位数字)。在接受这些不同的形式之前，允许用任意数目的“-”或空格字符在序列中的任意位置分隔数字。所以，对于一本书，可以使用“1-861008-23-6”或“1-86100-823-6”作为 ISBN。可以使用下面的正则表达式：

```
^(\d[- ]*){9}[\dxX]$
```

这个表达式分解后会变得非常简单。首先开头必须是一个数字，然后是 0 个或更多的连字符或任意个空字符。重复此序列至少 8 次(因为{9}是重复序列至少 8 次的一种简略表达)，括号表示一个分组；下一章将介绍分组。最后一个符号表明可以有一个数字，“x”或“X”。我们将使用如下 ISBN 进行匹配：

```
1-861-0008  236
```

如果仔细观察这个序列，会发现出现了过多的数字，其中增加了一个多余的 0。这个例子中只使用了一个限定符，下面看一下正则表达式是如何进行匹配的，这最好用下面的一个编号列表来表示：

(1) 匹配 1。

(2) 匹配“-”，但还可以匹配空字符，所以这个位置被放入栈中。

(3) 匹配 8。

(4) “-”没有找到。引擎移向了下一个决策点，而且没有发现空字符，因为使用了字符"*"，这是有效的，从而匹配。没有更进一步的决策点，所以栈中没有多余的字符。

(5) 匹配 6。

(6) 没有找到“-”。引擎移向下一个决策点，而且没有发现空格字符，因为使用了字符“*”，这是有效的，从而匹配。没有更进一步的决策点，所以栈中没有多余的字符。

(7) 匹配 1。

(8) 匹配“-”，但还可以选择匹配空字符，所以这个位置被放入栈中。

(9) 匹配 0。

(10) 没有找到“-”。引擎移向下一个决策点而且没有发现空字符，因为使用了字符“*”，这是有效的，从而匹配。没有更进一步的决策点，所以栈中没有多余的字符。

(11) 匹配 0。

(12) 没有找到“-”。引擎移向下一个决策点而且没有发现空字符，因为使用了字符“*”，这是有效的，所以匹配。没有更进一步的决策点，所以栈中没有多余的内容。

(13) 匹配 0。

(14) 没有找到“-”。引擎移向下一个决策点而且没有发现空字符，因为使用了字符“*”，这是有效的，所以匹配。没有更进一步的决策点，所以栈中没有多余的字符。

(15) 匹配 8。

(16) 这和前面的情况不同。没有找到“-”，所以移向下一个选择，0 个或多个空字符。它试图获得尽可能多的字符(在这里是两个空格字符)。然后它把控制权传递给下一条指令，并停留在栈中匹配 2 个空字符的决策点上。

(17) 匹配 2。

(18) “-”没有找到。引擎移向下一个决策点而且没有发现空字符，因为使用了字符“*”，这是有效的，从而匹配。没有更进一步的决策点，所以栈中没有多余的字符。

(19) 匹配 3，但还可以选择 x 或 X，所以此决策点放入栈中。

现在，匹配引擎已经行进到字符串的末尾，但表达式中仍有一个字符，所以它必须退回决策点，选择一个不同的字符来观察是否匹配。在这种情况下，有 4 个决策点可以返回：上表中的 2、8、16 和 19。匹配引擎返回最后一个决策点 19，并观察“x”或“X”是否匹配。因为数字是 3，它们不匹配，所以此决策点从栈中被删除。

现在移向16，在这里匹配更少的空字符——一个空格符。它成功地匹配后，开始再次向前移动。还有更多的决策点，但是，因为没有空字符，所以这个点留在栈中。引擎想匹配一个数字但没有成功，所以返回上一个点(16)。它试图匹配没有空字符的字符，而且把此决策点从栈中删除，因为没有更多可供选择的决策点。它再次失败，因为在下一个位置有一个空格字符，而正则表达式引擎仍需要一个数字。

现在它返回了栈中的下一个决策点，也就是决策点 8。“-”在这里匹配，但其他的决策没有尝试，所以在这里它要试着这么做。但是，由于那里没有空字符，仍然是一个“-”，所以这次匹配失败。它把这个点从栈中删除，并移向了上一个决策点 2。它尝试对这个字符进行同样的操作，但再次失败。栈中没有更多的决策点可以返回，所以匹配失败。

这种确定失败的方法漫长而且复杂。但是，因为引擎必须在表达式的多个点作出选择，所以这种方式是正则表达式引擎所必需的。但是，还可以改进正则表达式，以减少决策点的个数。这一点必须要考虑，因为正则表达式引擎是从上一次的决策点启动，在字符串和正则表达式的一些选择中，正则表达式将多次执行匹配，失败后就会返回，直到到达字符串的末尾。换句话说，即使没有匹配期望的所有字符，它仍然可以成功。但可以阻止通配符的“贪婪”。

为了使用通配符进行非贪婪的匹配，即懒惰匹配，只需使用前面介绍的限定词，但在后面加上一个“？”。区别在于，使用非贪婪的重复，正则表达式引擎将进行尽可

能少的匹配。另外一个例子可以说明这一点。

假设要匹配文本文件中的一个单行文本。在文件中有下列文本：

```
This is the first line.
This the second line.
The final line.
```

我们希望每一次匹配捕获的都是一个整行。因此，要搜索一个或多个空字符或非空字符，匹配应从一行的开始起到行的末尾结束。在第一次尝试中，使用的正则表达式为：

```
^[\w\W]+$
```

下面使用正则表达式测试器，并选中其中的复选框 MultiLine，如图 5-5 所示。

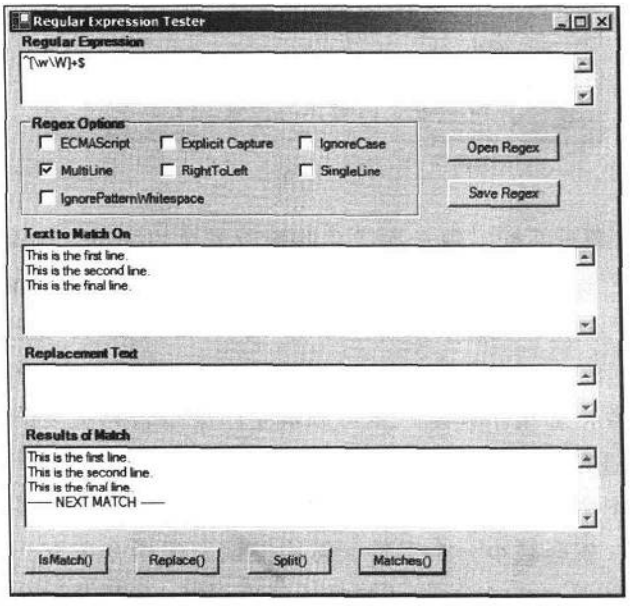


图　5-5

Matches()按钮将显示实际匹配的子串。在单击 Matches()按钮时，将只出现一次匹配，匹配的是整个字符串。我们希望匹配从一行的开头进行，在末尾结束，但只匹配了一次。

问题在于 Regex 引擎是贪婪的。而且，它所实现的第一次匹配是有效的，它从行首开始，中间包含了一个或多个单词或非单词字符，并在行尾结束。问题是它匹配了过多的内容，包括中间的所有行。在这里，选项 MultiLine 是不相关的。问题在于贪婪的重复字符“+”。解决的方法是使用非贪婪的重复指令，这里是“+?”。把正则表达式修改为：

```
^[\w\W]+?$
```

并再次单击 Matches()按钮，则每一行将分别包含在一次匹配中，如图 5-6 所示。

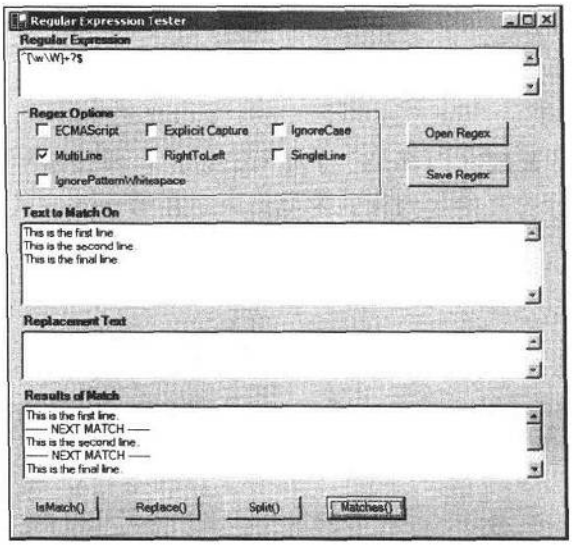


图　5-6

现在，Regex 引擎只匹配实现一次成功匹配所需要的字符。也就是说，引擎从第一行开始，匹配行尾之前的所有字符，然后终止。虽然选项 MulitiLine 规定，字符\W 可以匹配一个换行符，但是，当在其后加上字符“$”时，它就会终止。

何时使用懒惰匹配并没有严格的规定，但通常对于要搜索的字符却非常明确。观察指令所要匹配的对象，如果不匹配，就可以决定匹配是否为贪婪的。如果组合使用通配符和字符“.”，通常应使用非贪婪的匹配，否则在引擎移向下一条正则表达式指令前，会匹配字符串中所有的剩余字符。贪婪的指令知道它能匹配多少字符，如果以后出现一次匹配失败，就会减少一个字符。然而懒惰匹配指令在下一条指令匹配失败之前只作尽可能少的尝试，它只有在找到匹配时才再次尝试。如果 Regex 引擎在使用一个懒惰的通配符时遇到一个不能匹配的字符，或者在试图匹配下一个字符时到达了字符串的末尾，就会失败，并返回决策栈的上一个决策点。

使用贪婪和懒惰匹配的情况是不同的，需要考虑在字符串中将匹配多少字符，字符串总共有多少字符，可能遇到多少重复的字符。接下来再看看 ISBN 的例子。把正则表达式修改为：

```
^(\d[-\s]*?){9}[\dxX]$
```

还使用相同的字符串：

```
1-861-0008  236
```

这里匹配 0，再匹配 0 个或多个空字符，首先从 0 开始。匹配例程的区别是，当遇

到 2 个空字符时，它首先匹配第一个，但因为下一个要匹配的字符是数字，匹配将会失败。也就是说，如果匹配了两个空字符，则因为没有更多的空字符，所以将使栈为空，从而没有更多的决策点可以选择。从性能方面考虑，贪婪式搜索通常会提供更快的匹配速度，但因为它的贪婪，也意味着开发人员可能无法得到期望的匹配结果。但是，考虑这些运算符在建立表达式时所起的作用，您应该能够作出更明智的决策。

5.5.4　指定替换

前面了解了如何使用[]来指定一些替换。但是，它只能匹配单个字符。替换字符(|)提供了更大的选择范围。

假设要使用正则表达式匹配“Paul”或“John”。如果没有替换字符，就需要两个正则表达式，进行两次不同的比较。如果使用替换字符，就只需要一个表达式，如下所示：

```
Paul|John
```

替换字符与分组在一起使用更有效(下一章会介绍分组)。面对所有的替换决策，引擎将首先匹配第一个选择，然后从左到右一一匹配。

5.5.5　特殊字符

上一小节讨论了许多正则表达式。例如，如果要匹配任一单词字符，可以使用\w，如果要一次或多次匹配前一个字符，可以使用“+”。但是，如果要匹配一个真正的加号而不是表示匹配 0 个或多个字符的特殊字符“+”，或者匹配无法输入的字符，例如非标准字符，如 tab 键和回车符，该怎么办？这可以使用转义序列来指定这些字符。“\”字符表示其后的字符是一个转义字符序列，该序列后面的字符表示每个转义序列表示的字符，所以\t 表示制表符，这些转义字符如表 5-9 所示。

表 5-9　转义序列

转义序列	描　述
\\	匹配字符“\”
\.	匹配字符“.”
*	匹配字符“*”
\+	匹配字符“+”
\?	匹配字符“?”
\\|	匹配字符“|”

（续表）

转义序列	描　述
\(	匹配字符“(”
\)	匹配字符“)”
\{	匹配字符“{”
\}	匹配字符“}”
\^	匹配字符“^”
\$	匹配字符“$”
\n	匹配换行符
\r	匹配回车符
\t	匹配 tab 键
\v	匹配垂直制表符
\f	匹配换页符
\nnn	匹配一个三位八进制数指定的 ASCII 字符，例如\103 匹配一个大写的 C
\xnn	匹配一个二位十六进制数指定的 ASCII 字符，例如\x43 匹配 C
\unnnn	匹配一个 4 位十六进制数指定的 Unicode 字符
\cV	匹配一个控制字符，例如，\cV 匹配 Ctrl-V

假设有这样的字符串，“The price is $1.95 and it is available now”，要匹配“$”并用一个“£”来替换。美元符号必须转义，否则将解释为模式必须在字符串的末尾。

```
string myString = "The price is $1.95 and it is available now";
Console.WriteLine(Regex.Replace(myString, @"\$", "£"));
```

另一个例子是只包含一个 URL 的一个字符串，要把它划分成独立的部分，例如将 www.subdomain.domain.com 划分成：

```
www
subdomain
domain
com
```

需要在字符“.”处划分输入字符串。但是，“.”是正则表达式语法的一部分，所以要通知 Regex 分析器那是一个实际的字符“.”，而不是正则表达式的语法。此时可以在“.”前面添加一个“\”进行转义，即正则表达式为“\.”。

5.6　小结

本章介绍了正则表达式基础语法和正则表达式在.NET Framework 中的实现。

本章的主要内容如下：

- 在.NET 中，正则表达式的功能由命名空间 System.Text.RegularExpressions 提供，并包含在 System.dll 中。
- 最重要的类是 Regex 类。利用它可以创建和使用正则表达式来测试字符模式，搜索、替换和划分字符串。该类有许多静态方法，如果在代码中只想使用一次正则表达式，那么使用这些方法可以代替创建一个 Regex 对象。
- 正则表达式语法不但可以匹配实际的字符，还允许匹配字符类型，例如数字、字母和空字符。
- 字符匹配的次数可以由重复字符来规定。
- 替代字符“|”是一个逻辑 OR 语句，因为它允许匹配一个或多个模式。
- 指定键盘不能输入的字符，或者匹配的字符是正则表达式语法的一部分时，可以使用特殊的字符转义。

下一章将介绍更高级的正则表达式语法，还要介绍.NET 正则表达式所支持的更多样的特性，同时还要学习 System.Text.RegularExpressions 命名空间中的其他类。

第6章　正则表达式的高级概念

本章将介绍上一章没有涉及的高级正则表达式语法，以及.NET Framework 提供的用于处理正则表达式的其他类，如 Group 和 GroupCollection 类。对于更复杂的概念，如分组、在替换中使用分组和反向引用，我们将给出一些示例。

6.1　分组、替换和反向引用

本节将介绍.NET 中的类 Group 和 GroupCollection，替换以及反向引用。之所以把所有这些内容安排在一章来学习，是因为它们彼此都是相关的。Groups 用于说明一个字符模式应看作一个不同的项。例如，假设要匹配某人的全名，虽然最终匹配的结果应该是全名，但通常把它分为姓和名两部分。通过类 Group 和 GroupCollection，可以分别访问姓和名两部分。

替换和反向引用允许使用通过组匹配的字符。替换允许组用于替换操作，并作为替换模式的文本的一部分。假设某人不慎把一个单词重复输入了两次，例如“and and”，要用一个“and”来替换，就应把匹配的单词放在一个组中，然后用这个组替换整个匹配(“and and”替换为“and”)。在下文中将介绍替换的具体流程。

反向引用允许稍后在正则表达式中重用通过组匹配的模式，也就是引用先前匹配的组。例如，如果要匹配一个 HTML 元素的开始和结束标记，很明显首先匹配开始标记，例如“ ”，然后匹配对应的结束标记(“ ”)。但是，我们预先并不知道开始标记是什么，所以无法指定结束标记的内容，除非使用了反向引用。现在，假设拥有如下 HTML 字符串：

```
<P>some text</P><H2>some heading</H2>
```

在这个简单的例子中，HTML 开始标记由一个字母后跟一个可选数字组成(显而易见，实际的 HTML 有更复杂的标记，但这里只使用简单的例子)。要匹配开始标记，可以使用下面的正则表达式：

```
<[a-zA-Z]\d?>
```

该正则表达式在输入字符串中产生两次匹配：<P> and <H2>。

对于第一个匹配，结束标记一定是</P>，第二个一定是</H2>。重要的是字符 P 和 H2，因为它们必须在结束标记中得到匹配，所以，需要在开始标记匹配模式中对它们

分组：

```
<([a-zA-Z]\d?)>
```

方括号指定了模式的分组。所以，现在对于每一次匹配有一个组，在示例中，该组将包含 P 和 H2。

所以，在结束标记中，需要用第一个组中的内容和“<”/及“>”进行匹配，它们是</P> 和</H2>。

所以，完整的正则表达式应该为：

```
<([a-zA-Z]\d?)>[^<]*<\1>
```

此表达式匹配开始标记，确保了在一个正则表达式组中捕获标记类型。只要没有遇到字符“<”(即还没遇见一个新的开始或结束标记)，就搜索所有的字符。关闭标记应匹配<\，然后是 group1 的内容(\1 表示 group1)，最后是“>”。

6.1.1　简单的分组

在下面的小节中，介绍可以用于正则表达式的分组字符。

1. 捕获：( )

此分组字符组合模式在括号内匹配的字符。它是一个捕获组，也就是说被模式匹配的字符成为最终匹配的一部分，除非设定了选项 ExplicitCapture，此选项意味着默认情况下字符不是匹配的一部分。

假设有如下输入字符串：ABC1 DEF2 XY。下面的正则表达式匹配了从 A 到 Z 的 3 个字符，然后是一个数字：

```
([A-Z]{3})\d
```

此表达式产生了如下两个匹配：

```
1st Match = ABC1
2nd Match = DEF2
```

每一次匹配都含有一个组：

```
1st match's 1st group = ABC
2nd match's 1st group = DEF
```

有了反向引用，就可以通过正则表达式中的数字访问组，也可以通过.NET 的类 Group 和 GroupCollection 访问组。如果设定了选项 ExplicitCapture，就不能获得被组捕获的内容。

2. 非捕获: (?:)

此分组符号组合了模式在括号中匹配的字符。它是一个非捕获组，这意味着模式所匹配的字符将不作为一个组来捕获，但它将构成最后匹配结果的一部分。它与上文组类型基本完全相同，不同的是设置了 ExplicitCapture 选项。

例如，假设有如下输入字符串：1A BB SA 1 C。下面的正则表达式匹配了一个数字或一个 A 到 Z 的字母，后跟任意一个单词字符：

```
(?:\d|[A-Z])\w
```

此正则表达式将产生 3 个匹配：

```
1st Match = 1A
2nd Match = BB
3rd Match = SA
```

但是，没有组被捕获。

3. 通过名称捕获: (?<name>)

此分组符号组合了模式在方括号内匹配的字符，并把在尖括号中指定的名称作为组的名称。在正则表达式中，可以使用名称进行反向引用，这样就不用使用数字了。即使设定了选项 ExplicitCapture，它也是一个捕获组，这意味着在组内匹配的字符可以用于反向引用，或者可以通过 Group 类来访问。

考虑如下输入字符串：

```
Characters in Seinfeld include Jerry Seinfeld, Elaine Benes, Cosmo Kramer and George
Costanza
```

下面的正则表达式匹配了他们的姓和名，并在一个组 lastName 中捕获了姓：

```
\b[A-Z][a-z]+ (?<lastName>[A-Z][a-z]+)\b
```

此表达式产生了 4 个匹配：

```
1st match = Jerry Seinfeld
2nd match = Elaine Benes
3rd match = Cosmo Kramer
4th match = George Costanza
```

在每一次匹配中都有一个组 lastName：

```
1st match lastName group = Seinfeld
2nd match lastName group = Benes
3rd match lastName group = Kramer
```

```
4th match lastName group = Costanza
```

无论是否设定了选项 ExplicitCapture，组都将被捕获。

4. 比较简单的组

上文所介绍的最简单的组是捕获组。在括号内放置正则表达式的一部分，就可以通知正则表达式引擎把模式当作一个不同的实体来处理。当与选择字符“|”一起使用时，这最得尤其重要。例如，如果要匹配一个人的称呼，如 Mr、Mrs、Miss、Ms、Dr，然后是他们的名字，可以使用如下表达式：

```
(Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*
```

在括号中有 5 个可能出现的匹配，Mr、Mrs、Miss、Ms 或 Dr。括号中的模式只要求匹配其中的一个。紧跟组的是一个空格，然后是一个大写字母，然后是 0 个或多个小写字母。使用上一章介绍的正则表达式测试器程序，可以得到如图 6-1 所示的窗口。

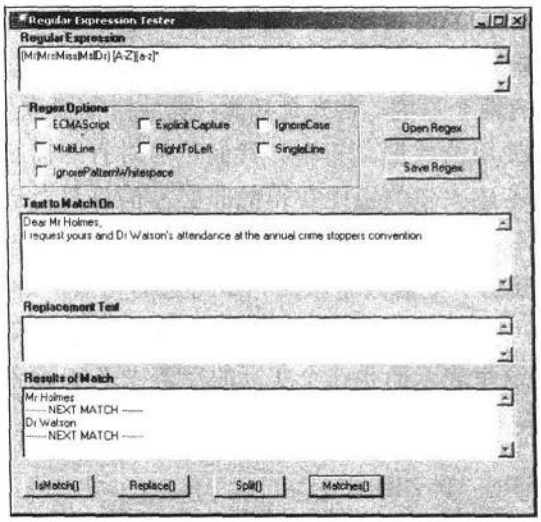


图　6-1

单击 Matches()按钮时，将发现如下两个匹配模式：

```
Mr Holmes
Dr Watson
```

将此与一个没有组的相同的正则表达式作比较：

```
Mr|Mrs|Miss|Ms|Dr [A-Z][a-z]*
```

此表达式的问题在于，它将匹配 Mr、Mrs、Miss 或 Ms，此外，它还会匹配一个 Dr 后跟一个姓。所以，如果修改了测试器中的正则表达式并再次单击 Matches()按钮，将得到如下匹配：

Mr

Dr Watson

通过增加括号，通知 Regex 引擎把选项作为一个组处理，从而匹配其中的一个项。注意，使用捕获组，组所匹配的模式变为了最后匹配的一部分。实际上，如下文所述，可以访问一个组所实现的单个匹配。正如上一章所提到的，这称为贪婪匹配，因为它“饥饿”地消费字符并存储它们。但是，可以指定不想捕获组。在示例中，需要使用组，以使替换能够正确实现。我们对组的结果没有特别的兴趣，它只是作为整个匹配的一部分。在这种情况下，可以使用非捕获或非贪婪的组(?:)。

如果修改了上一个关于姓的例子，使它包括一个非捕获组，将会得到如下表达式：

```
(?:Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*
```

如果在正则表达式测试器中测试该表达式，就会得到与第一次相同的匹配(Mr Holmes 和 Dr Watson)，但是，正则表达式引擎没有保存组所匹配的特殊字符以备将来使用，稍后在学习获得组信息时讨论以上情况。另一个使用非捕获组的办法是设定选项 ExplicitCapture，此选项使所有组在默认情况下都为非捕获。

选择使用非捕获组有两个原因。首先是效率，捕获组需要额外的工作和资源，所以不应该捕获不需要或不使用的数据。其次，我们可能对有些捕获组的结果发生兴趣，对其他的不感兴趣。从不感兴趣的组中获得的非必需信息只会造成混乱，所以最好使它们成为非捕获的，而不是修改代码忽略它们。

可以对表达式作的最后一处修改是添加一个可选的句点，如将“Mr”改为“Mr.”，等。记住，在 Regex 语法中，字符“.”具有特殊的意义，它表示匹配任一字符。所以，必须通知 Regex 分析器这里表示的是一个实际的字符“.”，而不是正则表达式语法中的特殊字符。可以使用这样的表达式：

```
(?:Mr|Mrs|Miss|Ms|Dr)\.? [A-Z][a-z]*
```

字符\.通知 Regex 分析器这里表示了一个字符“.”，“?”表示字符“.”可以出现 0 次或 1 次。

讨论的最后一个组是一个命名的捕获组——(?<name>)。这允许通过组名称引用上一次匹配的组。此组在当前还没有什么用处，所以在下文介绍反向引用时再讨论它。

6.1.2 Group 类和 GroupCollection 类

.NET Framework 允许访问组中的被捕获字符。它提供了 Group 类，此类代表了一次匹配中的一个组，还提供了 GroupCollection 类，它包含了 Group 对象，对应于一次成功匹配中的每一个组。

要获得一次匹配中所有组的集合，可以使用 Match 类的 Groups 属性。此属性返回 GroupCollection 对象，所以可以获得每一次匹配中的每个组。

假设有如下正则表达式：

```
(\d\d)\s([A-Z][A-Z])
```

此表达式将匹配 2 个数字，后跟 1 个空字符，然后是两个大写字母。给 2 个数字添加括号，构成第一个捕获组，给两个[A-Z]块添加括号，形成第二个捕获组。注意如果在代码中使用正则表达式，就必须用\对字符进行转义，所以\d 就变成\\d，\s 就变成\\s。另外，在上一章中，可以使用@符号显式引用字符串。如果输入字符串“12 AB 34 CD 56 EF”，则可以得到如下匹配：

```
12 AB
34 CD
56 EF
```

每一次匹配的都是两个组。例如，第一次匹配的两个组为 12 和 AB。可以访问这些组，因为每一个 Match 对象都包含了 Groups 属性，它返回 GroupCollection 对象：

```
Regex matchRegex = new Regex (@" (\d\d) \s ( [A-z] [A-z] ") ;
Match matchMade = matchRegex.Match("12 AB 34 CD 56 EF") ;
groupcollection matchGroups = matchMade.Groups
```

现在，matchGroups 集合将为正则表达式中的每一个正则表达式组保存一个 Group 对象。可以通过它的下标访问每一个组，但是要注意组集合中的第一个组总是整个的匹配。因此即使根本没有定义组，GroupCollection 仍至少拥有一个 Group 对象，下面的代码演示了这种情况：

```
Console.WriteLine(matchGroups[0].Value);
Console.WriteLine(matchGroups[1].Value);
Console.WriteLine(matchGroups[2].Value);
```

对于第一次匹配(12 AB)，它将得到如下内容：

```
12 AB
12
AB
```

对于第二次匹配(34 CD)，它将得到如下内容：

```
34 CD
34
CD
```

在上文中已经提到，可以通过正则表达式语法(?<name>)为组命名。命名之后，就可以通过组名和 index 来访问组。下面把正则表达式修改为：

```
(?<numberGroup>\\d\\d)\\s(?<letterGroup>[A-Z][A-Z])
```

现在，可以通过 index 和名称访问组：

```
Console.WriteLine(matchGroups[1].Value);                    // 12
Console.WriteLine(matchGroups["numberGroup"].Value);        // 12
Console.WriteLine(matchGroups[2].Value);                    // AB
Console.WriteLine(matchGroups["letterGroup"].Value);        // AB
```

注意：

正则表达式的组名是区分大小写的，所以 letterGroup 和 LetterGroup 是不同的。

请注意使用一个非捕获组的情况：

```
(?:\d\d)\s([A-Z][A-Z])
```

现在，matchGroups(1).Value 的第一个匹配将为 AB，因为第一个组捕获的字符没有被存储，以便以后在使用 GroupCollection 类的代码中引用，或用于反向引用或替换。

下面修改正则表达式测试器，并添加一个按钮，此按钮在单击时显示在一个正则表达式中匹配的组。首先，需要打开工程 Regular Expression Tester，然后向窗体添加一个按钮 GroupsButton。窗体如图 6-2 所示。

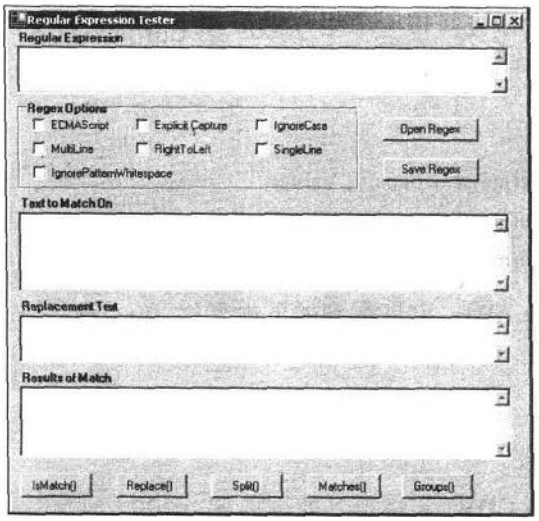


图　6-2

现在，需要为新按钮 GroupsButton 添加 Click 事件代码：

```
private void GroupsButton_Click(object sender, System.EventArgs e)
```

```
{
  try
  {
    RegexOptions selectedRegexOptions = GetSelectedRegexOptions();
    Regex matchesRegex = new Regex(this.RegexTextBox.Text,
                                   selectedRegexOptions);
    MatchCollection matchesFound;
    matchesFound = matchesRegex.Matches(this.InputTextBox.Text);
    StringBuilder resultsString = new StringBuilder(64);
    GroupCollection matchGroups;

    foreach(Match matchMade in matchesFound)
    {
      matchGroups = matchMade.Groups;
      foreach(Group matchGroup in matchGroups)
          resultsString.Append(" (" + matchGroup.Value + ")");
      resultsString.Append(Environment.NewLine + Environment.NewLine);
      resultsString.Append("------- NEXT MATCH -------\r\n");
    }

    this.ResultsTextBox.ForeColor = System.Drawing.Color.Black;
    this.ResultsTextBox.Text = resultsString.ToString();

  }

  catch(ArgumentException ex)
  {
    this.ResultsTextBox.ForeColor = System.Drawing.Color.Red;
    this.ResultsTextBox.Text = "There was an error in your "
        + "regular expression:\r\n" + ex.Message;
  }
}
```

大部分的代码都与 Matches()按钮的代码非常相似。主要的改变是 foreach 循环的插入，通过循环遍历了一次匹配内的每一个组：

```
foreach(Group matchGroup in matchGroups)
    resultsString.Append(" (" + matchGroup.Value + ")");
```

首先，定义了保存单个组和组集合细节的变量。然后，对每一次匹配，执行一次内部循环 foreach，遍历一次匹配中的每一个组。把 matchGroups 设定为 GroupCollection

对象，此对象由 Match 对象的 Groups 属性返回。然后，遍历每一个组(记住，即使没有定义组，完整匹配就是组集合中的第一项内容)。把每一个组的值都添加到结果字符串中，以便在 Results of Match 文本框中输出。

下面创建匹配称呼和姓的一个正则表达式，但只返回姓作为一个捕获组：

```
(?:Mr|Mrs|Miss|Ms|Dr) ([A-Z][a-z]*)
```

打开 Regular Expression Tester 工程，得到了图 6-3 所示的窗体。

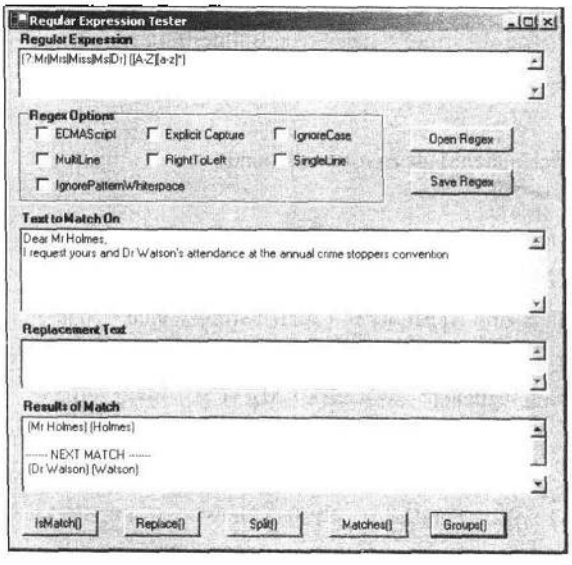


图　6-3

从结果可以看到，每一次匹配的 GroupCollection 都在第一个组中包含了整个的匹配，组在这里只匹配了姓。因为使用了替换字符，需要使称呼部分出现在一个组中，但是我们对它所捕获的字符并不感兴趣，感兴趣的只是所捕获的姓中的字符，也就是第二个组。所以，在开括号后使用字符，使替换组成为一个非捕获组。

使用非捕获组的另一种方法规定：在默认状态下使用 ExplicitCapture 选项，则所有的组都应该是非捕获组。这可以在 Regular Expression Tester 中通过选中相关的复选框并单击 Groups()按钮来完成。这一次，所有的组都没匹配，只输出了整个的匹配。选中了复选框 ExplicitCapture 后，就需要通过命名来指定何时想捕获一个组。可以把正则表达式修改为：

```
(Mr|Mrs|Miss|Ms|Dr) (?<lastName>[A-Z][a-z]*)
```

现在，得到了想要的结果，即使选中了 ExplicitCapture 复选框，第二个组也会被捕获，因为通过命名已经显式地通知了此选项来捕获分组。

6.1.3　替换

上一章介绍了 Regex 类的 Replace()方法。只要实现匹配，此方法都将执行一次全局搜索并替换输入字符串，而且用指定的文本来替换匹配的文本。但是，有时候也想在替换中使用一个正则表达式的匹配文本。假设有一个姓名清单，想用称呼和“X”替换所有的称呼和姓，如用“Mr X”替换“Mr Washington”。能够匹配称呼和姓的正则表达式为：

```
(Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*
```

问题在于不清楚使用什么称呼，所以不能简单地使用如下代码，这些代码将用 Mr X 代替所有的称呼：

```
string userInputString = "Dr Watson, Mr Holmes, and Mrs Smith";
userInputString = Regex.Replace(userInputString,
    "(Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*", "Mr X");
Console.WriteLine(userInputString);
```

很明显，要使用被第一个匹配称呼的组所捕获的文本，并把它替换到替换文本中。可以通过$\groupNumber 来做这项工作，groupNumber 表示组出现在正则表达式中的位置。需要把它添加到替换文本中，所以代码将变为：

```
string userInputString = "Dr Watson, Mr Holmes, and Mrs Smith";
userInputString = Regex.Replace(userInputString,
    "(Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*", "$1 X");
Console.WriteLine(userInputString);
```

可以使用 Regular Expression Tester 来做演示，图 6-4 中使用了按钮 Replace()。

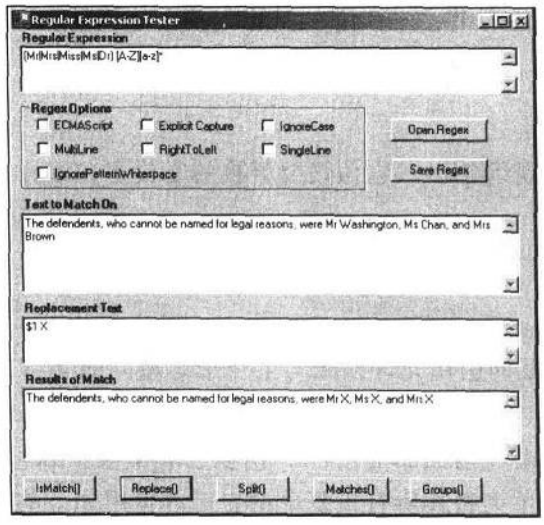


图　6-4

从上图中可以了解，后跟一个数字的$指明了应该用于替换文本中的组号码，本例中是第一个组。表 6-1 列出了许多可以使用的替换字符。

表 6-1　附加的替换字符

字　符	描　述
$group	用 group 指定的组号作替换
${name}	替换由组(?<name>)匹配的最后一个子串
$$	替换字符$
$&	替换整个的匹配
$`	替换输入字符串在匹配前的所有文本
$'	替换输入字符串在匹配后的所有文本
$+	替换最后捕获的组
$_	替换整个的输入字符串

上一个例子已经使用了$group。也可以使用$+作为最后一个捕获的组，因为匹配中惟一捕获的组也是第一个组，称呼组。

使用组号码的另一个方法是为组命名，并使用组名称作替换。正则表达式应该为：

```
(?<title>Mr|Mrs|Miss|Ms|Dr) [A-Z][a-z]*
```

替换字符串应为：

```
${title} X
```

记住，正则表达式和替换文本是区分大小写的，所以如果用 title 为组命名，使用 Title 作为替换模式中的名称将无法工作。此时不会产生错误，但将看到如下内容：

```
The defendants, who can't be named for legal reasons, were ${Title} X, ${Title} X,
and ${Title} X.
```

Regex 引擎没有发现组 Title，它假设要替换的是文本{Title}。

6.1.4　反向引用

在正则表达式中，反向引用允许匹配与上一个组相同的字符。一个常见的例子是匹配重复的单词。匹配上一个组的语法是\groupNumber 或\k<groupName>。但是，在代码中使用反向引用时，需要在前面加上“\”，所以它就变成了\\k<groupName>。

假设只用一个初始单词替换所有重复单词。可以使用如下的正则表达式，如图 6-5 所示。

(\b[a-zA-Z]+\b)\s\1

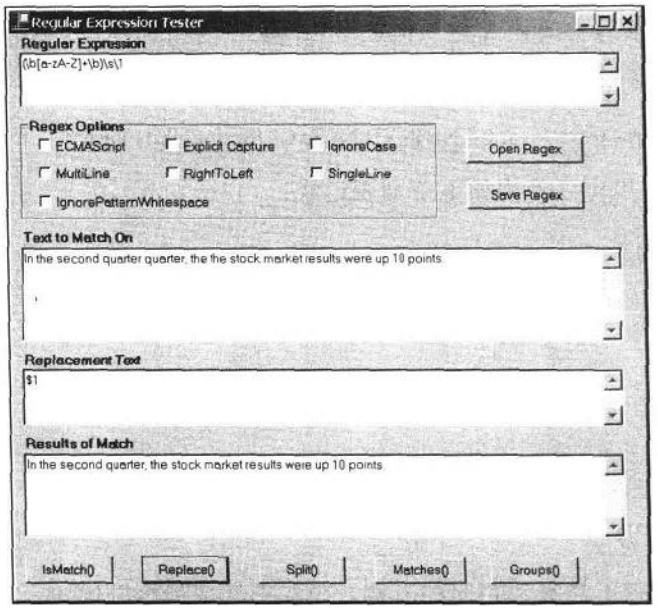


图　6-5

该表达式将匹配一个或多个字母，通过在表达式一边添加一个词界来限制，所有的匹配字符都在一个捕获组中。然后，匹配了一个空字符，并通过反向引用匹配了第一个组所匹配的内容。在 Regular Expression Tester 中，替换文本为$1。用这个匹配替换第一个组匹配的文本。通过单击 Replace()按钮，纠正了句子中的两个重复词的问题。

可以为组命名并通过名称反向引用此组，而不是使用一个号码：

(?<firstWord>\b[a-zA-Z]+\b)\s\k<firstWord>

注意在使用反向引用时，指定八进制数会使反向引用变得混乱。\1 到\9 总是用作反向引用，任何以 0 打头的数字，如\040，将作为一个八进制数处理，它代表了一个 ASCII 字符。如果一个数字没有以 0 打头，它将作为一个反向引用处理，前提是有一个匹配组用于反向引用。如果拥有 11 个组，那么\11 将匹配第 11 个组。如果只有 10 个组，那么\11 作为代表一个 ASCII 字符的八进制数被处理。

6.1.5　高级组

上面介绍了 3 种基本和常见的组：捕获的、非捕获的和通过名称捕获的组。除此之外还有很多高级组需要讨论。

1. 正声明: (?=)

(?=)规定了括号中的模式必须出现在声明的右侧。模式将不构成最后匹配的一部分。例如，考虑如下的输入字符串：

```
The languages were Java, C#.NET, VB.NET, C, JScript.NET, Pascal
```

对此输入字符串可以使用如下正则表达式：

```
\S+(?=\.NET)
```

得到如下结果：

```
C#
VB
JScript
```

2. 负声明：(?!)

(?!)规定了模式不能紧邻声明的右侧。模式不构成最终匹配的一部分。考虑如下的输入字符串：

```
123A 456c 789 111C
```

对此输入字符串可以使用如下正则表达式：

```
\d{3}(?![A-Z])
```

获得如下结果：

```
456
789
```

3. 反向正声明: (?<=)

(?<=)规定了括号中的模式必须出现在声明的左侧。模式不构成最终匹配的一部分。考虑如下的输入字符串：

```
New Mexico, West Virginia, Washington, New England
```

可以对此输入字符串使用如下正则表达式：

```
(?<=New )([A-Z][a-z]+)
```

得到如下结果：

```
Mexico
```

England

4. 反向负声明: (?<!)

(?<!)规定了模式不能紧邻声明的左侧。模式仍然不构成最终匹配的一部分。考虑如下的输入字符串：

123A 456F 789C 111A

对此输入字符串可以使用如下正则表达式：

(?<!1)\d{2}[A-Z]

得到如下结果：

56F
89C

5. 非回溯: (?>)

(?>)防止了 Regex 引擎在失败时回溯(backtracking)以试图匹配，它称为贪婪的子表达式。假设要匹配所有以“ing”结尾的单词。输入字符串如下：

He was very trusting

要得到完整的文本，可以使用如下正则表达式：

.*ing

但是，如果不进行回溯，将无法实现匹配：

(?>.*)ing

6. 比较高级组

严格地讲，这些只是声明而不是分组；它们声明一个模式必须存在于一个特定的位置。声明组匹配的模式不是最终匹配的一部分，因而它们不能用于反向引用。

头两个描述的高级组匹配声明右侧的一个模式；它们预先检测模式是否存在。正声明需要模式的存在，整个的匹配才能成功。相反，负声明查看模式是否不存在，如果存在，则不能实现匹配，而且整个的匹配将失败。

在 Regular Expression Tester 中输入如下字符串：

A1 B 2 C D3 E4

可以使用如下的正声明：

[A-Z](?=\d)

得到如下匹配：

A
D
E

注意，即使声明的模式(这里只是\d)必须匹配，它也不作为最终匹配结果的一部分。可以把正则表达式修改为一个负声明：

```
[A-Z](?!\d)
```

修改后得到的匹配为：

B
C

换句话说，没有跟随数字时，才实现一个匹配。

下面的两个组是相同的声明，不同的是它们查看紧邻声明的左侧，寻找匹配的模式。正声明要求：为了整个匹配的成功，模式必须存在，而负声明要求：为了成功匹配，模式一定不存在。

使用与正声明示例相同的输入字符串，但把正声明改为反向正声明：

```
(?<=[A-Z])\d
```

此声明将匹配前缀为一个大写字母(从 A 到 Z)的任意数字，产生的匹配如下：

1
3
4

修改为反向负声明：

```
(?<![A-Z])\d
```

此声明匹配没有前缀一个大写字母(从 A 到 Z)的任一数字，并输出：

2

最后一个组是非回溯组，这或许并不容易理解。正则表达式引擎将尽力实现一次匹配。现在演示一个具体的示例，假设要匹配以.com 结尾的 URL，可以创建如下正则表达式：

```
www\.(.*)\.com
```

此表达式匹配 www，后跟一个句点，然后是一个捕获组内的一个或多个任意字符，后跟另一个句点，最后是 com。可以使用 Regular Expression Tester 来进行测试，如图 6-6 所示。

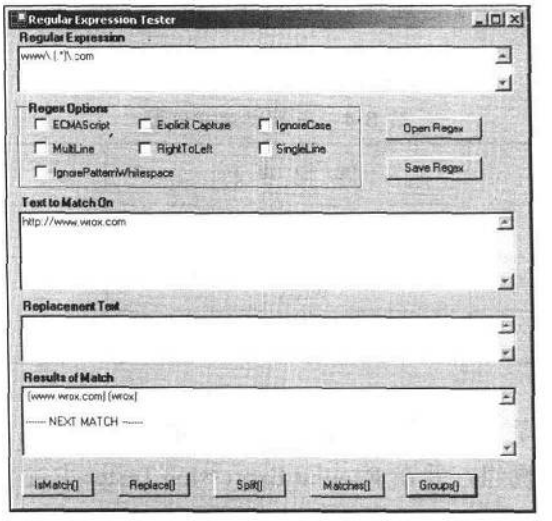


图　6-6

在上图中，单击 Groups()按钮，就可以看到包含.*的第一个组匹配 wrox。直观地看，“.”应该匹配任一字符，而“*”匹配一个或多个任意字符，所以您或许想知道为什么组没有匹配.com。

要实现一次成功的匹配，就必须匹配完整的正则表达式，正则表达式引擎不能只忽略.com。正则表达式引擎就像一个只看结果的约会中介，非常渴望实现匹配。所以，引擎会回溯 —— 直到实现一次匹配。下面介绍具体的流程。

表 6-2 给出了第一次的匹配尝试。

表 6-2　正则表达式匹配

正则表达式	匹 配 字 符	是 否 匹 配
www.	www.	是
(.*)	wrox.com	是
.com		否

第一次匹配将失败，因为组(.*)匹配了所有的剩余字符，而.com 将无法匹配。在接下来的匹配中，Regex 引擎回溯了一个字符，如表 6-3 所示。

表 6-3　正则表达式匹配

正则表达式	匹 配 字 符	是 否 匹 配
www.	www.	是
(.*)	wrox.co	是
.com	m	否

仍然无法实现完整的匹配。现在，Regex 引擎一直回溯字符，直到最终实现一次匹配，如表 6-4 所示。

表 6-4 正则表达式匹配

正则表达式	匹配字符	是否匹配
www.	www.	是
(.*)	wrox	是
.com	.com	是

所有的回溯操作都是艰难而且低效的。非回溯组允许关闭回溯操作以提高效率。把正则表达式修改为非回溯组，并在 Regular Expression Tester 中进行测试：

```
www\.(?>.*)\.com
```

结果没有实现任何匹配。问题在于 Regex 引擎只测试了第一个选项(.*匹配包括.com 的所有字符)，失败后，分组的回溯又被关闭，所以没有实现任何匹配。在这种情况下，修改正则表达式：

```
www\.(?>[^.]*)\.com
```

修改后的表达式可以成功匹配，因为它会搜索所有字符直到找到一个句点。.com 没有通过括号实现匹配。这种正则表达式更加有效，因为它不需要进行回溯操作。但是，它不会匹配如 www.domain.subdomain.com 这样的子域。同样，非回溯组也是一个非捕获组——它的值没有被捕获以供将来使用。

6.2 在正则表达式中作决策

本章学习了使用替换字符 | 作简单的决策。正则表达式提供了两个更高级的决策指令。

expression 选项可以选择如下语法中的一种：

```
(?(expression)yes|no)

(?(?=expression)yes|no)
```

如果匹配了 expression，那么正则表达式将试图匹配 yes。否则，就试图匹配 no 正则表达式。正则表达式 no 的是可选的。有一点很重要，即作决策的模式 expression 的长度为 0，代表了 yes 或 no 的正则表达式将从与 expression 相同的位置开始匹配。

下面看看如下的输入字符串：

```
1A CB 3A 5C 3B
```

考虑如下正则表达式：

```
(?(\d)\dA|[A-Z]B)
```

使用此正则表达式可以得到下面的结果：

```
1A
CB
3A
```

name 选项可以使用如下语法：

```
(?(name)yes|no)
```

如果组中的 name 正则表达式被匹配，则试图匹配正则表达式 yes。否则，试图匹配正则表达式 no。表达式 no 是可选的。

使用如下输入字符串来示范：

```
77-77A 69-AA 57-B
```

使用如下正则表达式：

```
(\d7)?-(?(1)\d\d[A-Z]|[A-Z][A-Z])
```

此正则表达式将产生如下匹配：

```
77-77A
-AA
```

如果忘记了 yes 或 no 正则表达式的匹配开始位置与做测试的表达式模式的位置相同，那么决策表达式的语法可能很容易混淆，例如下面的正则表达式：

```
(?(\d)A|B)
```

尝试对如下输入字符串使用该表达式：

```
1A CB 3A 5C 3B
```

使用 Regular Expression Tester，其结果如图 6-7 所示。

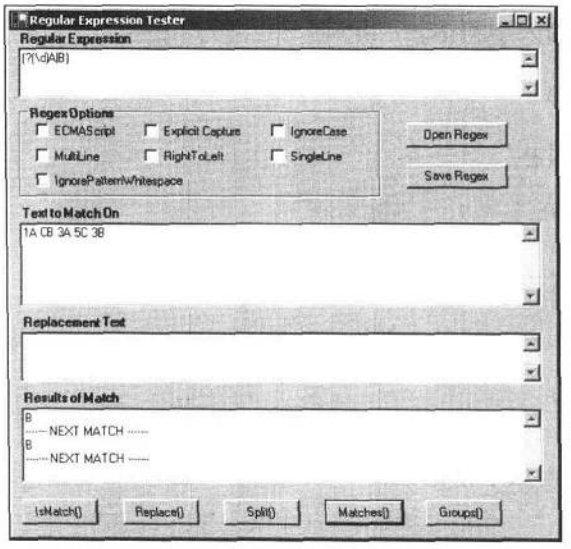


图　6-7

假设存在 1A 和 3A，为什么它们没有匹配呢？原因是决定 yes 或 no 测试（\d）的模式是成功的，但正则表达式 yes(这里指的是一个字符 A)仍然没有匹配。yes 测试与决策测试在相同的位置起动，测试需要一个数字，但 A 显然不是数字，所以即使决策表达式是成功的，yes 也无法成功匹配。要确保正则表达式在上例中发挥作用，需要把它修改为：

```
(?(\d)\dA|B)
```

匹配结果为：

```
1A
B
3A
B
```

重要的是，yes 正则表达式必须至少匹配与决策正则表达式相同的模式，否则将匹配失败。匹配不必完全相同，只要匹配的是相同的模式。例如，表达式的例子仍然适用：

```
(?(\d)\wA|B)
```

这是因为\w 将匹配一个单词字符和包括数字的多个单词字符。

6.3　在正则表达式内设定选项

正则表达式的语法允许在一个组内修改正则表达式。例如，假设要设定一个不区分大小写的组：

```
(?i:[a-z])
```

现在，无论创建正则表达式时全局选项是什么，[a-z]都将匹配大写字母 A 到 Z。如果想删除一个选项，只需要使用符号“-”。例如，使用下面的表达式使匹配区分大小写：

```
(?-i:[a-z])
```

表 6-5 列出了可以内联(inline)设定的 5 个选项。

表 6-5　可以内联设定的 5 个正则表达式选项

正则表达式选项	标　志	描　　述
ExplicitCapture	N	此选项规定，只有显式命名或编号的组才是有效的捕获
IgnoreCase	I	此选项指定不区分大小写的匹配
IgnorePatternWhitespace	X	此选项规定，非转义的空字符被排除在模式之外，并启用了前缀一个#的注释
MultiLine	M	指定多行模式，修改了字符^和$的含义
SingleLine	S	此选项规定，只有显式命名或编号的组才是有效的捕获

可以一次设定任意多的选项，例如：

```
(?n-i:[a-z])
```

这样就把组设为 ExplicitCapture 选项并区分大小写。

6.4　正则表达式引擎的规则

现在已经介绍了本书所涉及的所有正则表达式语法。最后介绍一些正则表达式引擎匹配的规则。如果想得心应手地使用正则表达式，需要牢记一些基本规则：

规则 1：正则表达式引擎会对输入字符串尽快地开始匹配。它一次搜索一个字符，直到发现一次匹配。

```
Input string = 123 ABC 456 DEF
Regex = [A-Z]*
```

本例从 ABC 开始匹配。

规则 2：发现一个匹配的开头后，正则表达式引擎将继续匹配，直到遇到一个不被模式接收的字符。

```
Input string = 123 ABC 456 DEF
Regex = [A-Z]*
```

第一次匹配从 ABC 开始，当遇到模式不接受的空格时结束。

规则 3：Regex 引擎非常贪婪——只要模式允许，它将匹配尽可能多的字符。

```
Input String = 'Dr Watson's watch'
Regex = '.*'
```

实现的匹配是整个字符串。表达式模式匹配一个单引号，后跟任意字符，然后以另一个单引号结束。所以'Dr Watson'也是匹配的，引擎不需要继续匹配其他的字符，它已经有匹配的模式了。但是，正则表达式引擎继续匹配(见规则 2——匹配一直进行到出现一个无效字符)，并发现还能匹配：

```
s watch'
```

添加一个“?”，使“*”成为一个非贪婪的限定符：

```
Input String = 'Dr Watson's watch'
Regex = '.*?'
```

这时，正则表达式将只匹配：

```
'Dr Watson'
```

正则表达式遇到了第二个'，现在已经得到了足够的匹配字符，并终止。也就是说，它是非贪婪的，只获取足够一次匹配的字符，就不再捕获多余字符。

规则 4：Regex 引擎渴望实现匹配，所以将在需要时回溯以实现匹配。

```
Input string = 'Hello world' said K & R.
Regex = '.*'
```

匹配结果为：

```
'Hello World'
```

但是，如果没有回溯操作将无法得到匹配。规则 3 说明表达式引擎在寻找它能实现的最大匹配。.*表示匹配一个字符 0 次或者多次。所以，world 后的“'”已经足够完成一次匹配，但 Regex 引擎想实现更多的匹配，并在规则 2 不适用时继续搜索(只要没有发现不匹配的字符)。表达式引擎继续向右搜索，直到到达输入字符串的末端，因为字符“.”能够匹配所有的字符。但是还有一个问题，模式规定了在模式尾部必须有一个字符“'”，

但因为它使用.*获得了所有的字符，这意味着“!”将无法匹配。所以在这里，渴望实现匹配的引擎一次回溯一个字符，直到回溯到 world 后的“!”，从而完成匹配。

使用一个非回溯组禁用回溯操作，会发生下列情况：

```
Input string = 'Hello world' said K & R.
Regex = '(?>.*)'
```

这一次，将没有任何匹配实现，因为.*匹配了右侧直到结束的所有字符，而且规定了不使用回溯，所以引擎无法根据“Hello World”实现匹配。

规则 5：Regex 引擎总是选择第一个选项。

当使用字符“|”指定不同选择时，正则表达式引擎将试图匹配第一个选项，如果无法匹配，再匹配第二个，然后第三个等。

```
Input string = 1234 123 3456
Regex = (\d{2}|\d{3}|\d{4})
```

匹配结果如下：

```
12
34
12
34
56
```

可以看到，即使可以选择匹配 1234、123 或 3456 作为完整匹配，它还是匹配了第一个选项，并在每一次匹配两个数字。

牢记这 5 条规则将有助于正确使用正则表达式，因为正则表达式经常会出现一些令人意想不到的结果。

6.5　小结

本章的主要内容如下：

- 使用正则表达式语法如何对一个模式进行分组。
- 如何使用.NET Framework 中的组并通过 Group 类访问它们的值。
- 如何在一个 Replace()方法的替换部分使用组。
- 如何在正则表达式内部使用先前匹配的组。

下一章将使用前面所学的正则表达式语法来创建很多实用的、真实的示例。

第7章　正则表达式模式

本章将详细介绍一些正则表达式模式的实用示例。这一章不仅会提供一个有用的模式库，还会说明正则表达式是如何编译的。本章将成为学习如何构造正则表达式的非常有用的向导，同时，还会提供一组处理常见数据的正则表达式。

7.1　验证字符

首先利用一些基本且常用的正则表达式来核查一个字符串是否包含无效字符。第一组方法用来检查字母、数字和空格组成的序列的有效性。表 7-1 所示为验证字符的模式及其描述。

表 7-1　验证字符的模式及其描述

模　式	描　述
[^a-zA-Z\\d]	检查文本不包含字母表的字母、数字或者空格
[^a-zA-Z\\d]	检查文本不包含字母表的字母或者数字
[^a-zA-Z]	检查文本不包含字母表的字母或者空格

在用户输入了字符串，并想确保它仅包含我们需要的字符时，这些模式是非常有用的。上面每一个表达式的使用方法都是非常相似的，例如，检测变量 userInputString 是否包含除字母表中的字母、数字和空格之外的字符：

```
if (Regex.IsMatch(userInputString, "[^a-zA-Z\\d ]"))
{
  // Invalid Chars found
}
else
{
   //Valid chars only found
}
```

但通常只应匹配现存内容而不是不存在的内容。相比还没有出现的字符，匹配已经存在的或者将会出现的字符会更为简单。不过，如果先前的一个匹配失败，这些模式可以用于匹配无效字符。在下面简单的匹配例子中，我们将像上面的例子那样进行否定匹配。

7.2　验证数字

这一节学习验证各种数字类型的不同方法。

7.2.1　只包含数字

首先利用一个简单的正则表达式来验证传递过来的字符串是否只包含数值型字符，其中验证数字的模式及描述如表 7-2 所示。

表 7-2　验证数字的模式及其描述

模　式	描　述
[^\\d.]	检查字符串是否包含除了数字和小数点之外的字符

对这个模式来说，任何非数字或非小数点的字符都是无效的。注意，通常“.”是一个特殊的正则表达式字符，它可以匹配任何字符，如果要匹配一个真正的“.”，需要将它转义为“\.”。不过，当利用方括号来指定要匹配的字符时，这就不必要了。

可以利用下面的代码来检测 userInputString 中是否存在无效字符：

```
if (Regex.IsMatch(userInputString, "[^\\d.]"))
{
  // Invalid Chars found
}
else
{
   //Valid chars only found
}
```

7.2.2　只包含整型数

下面的正则表达式用来检查整数：

```
^((\+|-)\d)?\d*$
```

与前面的匹配字符的简单表达式不同，这个表达式匹配的是一个指定的模式。它指定这个字符串可以以符号“+”或“-”开始，但是一旦以符号“+”或“-”开始，该符号后就必须至少有一个数字，这个数字之后可以有零个或多个数字。末尾的“$”确

保了最后一个数字之后不会再有其他字符。注意，由于“+”是正则表达式语言的一部分，它表示匹配一个或多个字符，所以必须在第一组中将其转义为\+，以告知正则表达式引擎我们需要一个真正的“+”，而不是正则表达式语法中的“+”。

下面这些整数是有效的：

```
+1
234
-2445234
```

而以下这些是无效的：

```
++123
ABC
1.23
1234C
```

先前，如果发现了一个匹配，说明存在无效字符。现在，如果发现一个匹配，则表示这个输入字符串有效。检查有效字符串比先前检查无效字符串更有意义，因为不只检查某些字符是否存在，还要指明字符必须遵守特有的顺序，所以+123 是有效的，而123+就是无效的。使用这个表达式的代码如下：

```
if (Regex.IsMatch(userInputString, "^((\\+|-)\\d)?\\d*$"))
{
  // Input string valid
}
else
{
  // Invalid input string
}
```

7.2.3　只包含浮点数

假设检查正或负浮点数：

```
^(?:\+|-)?\d+(?:\.\d+)?$
```

因为浮点数可以等同于整数(1 是一个有效的整数和浮点数)，所以正则表达式对于整数和浮点数来说都是有效的。

在这里，正则表达式规定数字、负号和小数点是允许的。表达式^(?:\+|-)?规定字符串必须以字符开头(有效的数字不能位于其他字符的中间)。它还规定在非捕获组中，可以以符号“+”或“-”作为开头。需要把“+”和“-”放在一个组中，因为“|”或

者匹配左侧的正则表达式模式“+”，或者匹配右侧的正则表达式模式“-”。如果不把“\+|-”放在一个组中，就会得到错误的结果——模式会查找只带有“+”号或者只带有“-”号的数字。由于我们对捕获的结果并不感兴趣，所以使用了非捕获组——一个替代方法是当用到 Regex 对象时设置 ExplictCapture 选项。

接下来，利用\d+指定符号后面可以跟一个或多个数字。这样就匹配了小数点前面的一个或多个数字。最后是(\.\d+)?$，用于匹配小数点和它后面的数字，是正则表达式的一部分。它匹配一个句点字符；这个字符是正则表达式语言中的一部分，所以当要匹配的是一个真正的句点字符时，需要将它转义。接下来，匹配一个或多个数字。表达式组的最后有一个“?”，它说明这个组必须匹配零次或多次。基本上，这个组允许成功匹配没有小数点的数字。

```
if (Regex.IsMatch(userInputString, "^(?:\\+|-)?\\d+(?:\\.\\d+)?$"))
{
  // Input string valid
}
else
{
  // Invalid input string
}
```

7.3　验证电话号码

到目前为止，对数据的验证非常简单。然而，对电话号码的验证具有挑战性。问题在于：

- 国家与国家之间的电话号码是不同的
- 输入一个有效的号码有多种方法，例如添加或者不添加国内或国际的代码

有一种区分电话号码格式的方法，即把电话号码进行分解，将每一部分放入独立的方格中，例如，一个方格中放入国际拨号代码，一个方格中放入地区代码，一个方格中放入号码本身，如果需要，还可以在一个方格中放入分机号码。

另外还有一种方法，即利用正则表达式来匹配各种不同的国际电话号码。这里需要指定的不止是有效字符，还需要指定数据的格式。例如，下面的这些格式是有效的：

```
+1 (123) 123 4567
+1123123 456
+44 (123) 123 4567
+44 (123) 123 4567 ext 123
```

+44 20 7893 4567

注意：

可以在 http://phonebooth.interocitor.net/wtng/找到全球电话号码系统的更多信息。

正则表达式需要处理的号码(可以用空格分隔)见表 7-3。

表 7-3　正则表达式需要处理的号码及其说明

号　码	说　明
国际号码	符号“+”后跟 1～3 个数字(可选)
地区号码	2~5 个数字，有时放在圆括号中(必选)
实际的订阅者号码	3~10 个数字，有时带有空格(必选)
分机号码	2~5 个数字，前面带有 x、xtn、extn、pax、pbx 或者 extension，有时放在圆括号中

很明显，肯定还有未考虑到的国家，这就需要针对消费者和伙伴所在的地点进行处理了。下面的正则表达式比前面的那个复杂得多：

```
^(\+\d{1,3} ?)?(\(\d{1,5}\)|\d{1,5}) ?\d{3} ?\d{0,7}
( (x|xtn|ext extn|extension)? \.? ?\d{1,5})?$
```

需要为这个表达式设置不区分大小写的选项，以及显式捕获的选项。虽然它很复杂，但是把它分解之后，就非常简单了。实际上，它的长度决定了它必须被分为两行；但应把它输入到一行上。

从匹配国际拨号代码的模式开始：

```
(\+\d{1,3} ?)?
```

现在，要匹配一个加号(\+)，这个加号后面跟有 1～3 个数字(\d{1,3})和一个可选空格(?)。由于“+”字符是一个特殊字符，要在它前面添加一个“\”字符，以说明匹配的是一个真正的“+”字符。这些字符放在一个圆括号中，说明它们是一组字符。这里允许出现空格，并把这组完整的字符进行零次或多次匹配——这是由这个组的闭圆括号之后的“?”字符指定的。

接下来，是匹配地区代码的模式：

```
(\(\d{1,5}\)|\d{1,5})
```

这个模式包含在圆括号中，说明它是一个字符组，匹配圆括号中 1～5 个数字((\d{1,5}))，或仅仅匹配 1～5 个数字(\d{1,5})。由于圆括号在正则表达式语法中是特殊字符，所以要想匹配真正的圆括号，就需要把“\”字符放在它们的前面。同时要注意对管道符号(pipe symbol)(|)的使用，它的含义是“OR”或者“匹配这两种模式中的任一种”。

接下来，匹配订阅者号码：

```
?\d{3} ?\d{0,7}
```

注意，在第一个“?”之前有一个空格—— 这个空格和问号的意思是，“匹配零个或一个空格”。这个符号后面是 3 个或 4 个数字(\d{3})，以 US 的标准应该有 3 个数字，而以 UK 的标准应该有 4 个数字。然后是另一个“零个或一个空格”，最后是 0～7 个数字(\d{0,7})。

最后，添加处理一个可选分机号码的部分：

```
( (x|xtn|ext|extn|extension)? \.? ?\d{1,5})?$
```

这个组是可选的，因为它的圆括号后面有一个问号。这个组本身是检验空格的，后面可以选择带有 x、ext、xtn、extn 或者 extension 中的一个，接下来是零个或一个句点(注意\字符，由于“.”是正则表达式语法的一个特殊字符)，再后面是零个或一个空格，最后是 2～5 个数字。

把这 4 个模式放在一起，就构建出了除开始和结束语法之外的完整的正则表达式了：记住正则表达式以“^”开始，以“$”结束。“^”字符指定模式必须在字符串起始位置开始匹配，而“$”字符指定模式必须在字符串末尾结束匹配。也就是说，字符串必须完全匹配这个模式——在这个匹配模式的前面或后面没有任何其他字符。

于是，就可以把这个解释过的正则表达式用到下面的代码上了，记住，需要在现有的“\”前面添加另一个“\”：

```
if (Regex.IsMatch(userInputString,
                  "^(\\+\\d{1,3} ?)?(\\(\\d{1,5}\\)|" +
                  "\\d{1,5}) ?\\d{3} ?\\d{0,7}" +
                  "( (x|xtn|ext|extn|extension)? \\.? ?\\d{1,5})?$",
                  RegexOptions.IgnoreCase |
                  RegexOptions.ExplicitCapture))
{
  MessageBox.Show("Valid");
}
else
{
  MessageBox.Show("Invalid");
}
```

注意，在这个例子中设置 IgnoreCase 选项是很重要的；否则，正则表达式将无法匹配 ext 部分。

注意：

如第 5 章所述，把“\”用作转义字符的另一个方法是在正则表达式的前面加上@字符。

7.4 验证邮政编码

前面成功检验了世界范围内的电话号码，但要对邮政编码进行检验则是一个较大的挑战。下面创建一个仅检验 US 和 UK 邮政编码的方法。如果要检验其他国家的邮政编码，就需要修改代码。用一个正则表达式验证多个邮政编码的可管理性较差，如果对需要检验的每一个国家的邮政编码分别使用一个正则表达式，就会容易一些。但为了达到这个目的，我们把为 UK 和 US 设计的正则表达式联合起来：

```
^(\d{5}(-\d{4})?|[a-z][a-z]?\d\d? ?\d[a-z][a-z])$
```

这个代码行实际上分为两部分：第一部分检验 US 的邮政编码，第二部分检验 UK 的邮政编码。先看美国的邮政编码。

美国邮政编码可以表示为两种格式中的一种——5 五个数字(12345)，或者是 5 个数字后面加上一个短划线和 4 个数字(12345－1234)。匹配这两种格式的邮政编码正则表达式如下：

```
\d{5}(-\d{4})?
```

这个表达式匹配 5 个数字，后跟一个可选的非捕获组，用来匹配一个短划线和它后面的 4 个数字。

对一个匹配 UK 邮政编码的正则表达式，先来研究一下它的各种格式。UK 邮政编码的格式是，开头为一个或两个字母，它们后面是一个或两个数字，之后是一个可选空格，接着是一个数字，然后是两个字母。另外，伦敦中部的一些邮政编码，例如：SE2V 3ER，第一部分的末尾有一个字母。目前，只有一些以 SE，WC 和 W 开头的编码是这种情况，但这是会改变的。UK 邮政编码的一些有效示例有：CH3 9DR、PR29 1XX、M27 1AE、WC1V 2ER 和 C27 3AH。

了解了这些，可以列出如下模式：

```
([a-z][a-z]?\d\d?|[a-z]{2}\d[a-z]) ?\d[a-z][a-z]
```

为了“匹配其中的任一种”，利用|字符把这两种模式联合起来，并用圆括号组合。然后在模式的起始部分添加“^”字符，在它的末尾部分添加“$”字符，以确保字符串中的惟一信息是邮政编码。虽然邮政编码应该大写，但小写也是有效的格式，所以在使用正则表达式时，也设置了不区分大小写的选项：

```
^(\d{5}(-\d{4})?|([a-z][a-z]?\d\d?|[a-z{2}\d[a-z]] ?\d[a-z][a-z]])$
```

正则表达式的使用与前面的例子是相同的，但在使用字符串时要添加另一个“\”字符：

```
if (Regex.IsMatch(userInputString,
                  "^(\\d{5}(-\\d{4})?|([a-z][a-z]" +
                  "\\d\\d?|[a-z]{2}\\d[a-z]) ?" +
                  "\\d[a-z][a-z])$",
                  RegexOptions.IgnoreCase |
                  RegexOptions.ExplicitCapture))
{
  MessageBox.Show("Valid");
}
else
{
  MessageBox.Show("Invalid");
}
```

7.5　验证电子邮件地址

在开始用一个正则表达式匹配电子邮件地址之前，先来看一下有效电子邮件地址的类型：

- someone@mailserver.com
- someone@mailserver.info
- someone.something@mailserver.com
- someone.something@subdomain.mailserver.com
- someone@mailserver.co.uk
- someone@subdomain.mailserver.co.uk
- someone.something@mailserver.co.uk
- someone@mailserver.org.uk
- some.one@subdomain.mailserver.org.uk

如果去 SMTP RFC (http://www.ietf.org/rfc/rfc0821.txt)调查，还会得到如下的地址类型：

- someone@123.113.209.32
- """Paul Wilton"""@somedomain.com

这个列表很长，包含要处理的各种格式。最好把它们分解开来。首先，要注意上面提到的两种类型。这两种类型不经常使用，但是也会提供。实际上，RFC 是不确切的，建议电子邮件的 IP 地址类型写成：someone@[123.113.209.32]。然而，RFC 还允许将域名提供给双方，所以，应该使用上面列举的更为通用的地址形式。对于后面的这种类型，只有很少的电子邮件服务器支持它们。不过，基于完整性，我们还是把它包括在内。首

先，需要把电子邮件地址分解成独立的部分，然后，再关注“@”符号后面的内容。

7.5.1 验证 IP 地址

这是最简单的，所以首先考虑它。电子邮件地址的“@”符号后面可以是一个域名，也可以是一个 IP 地址。IP 地址是由 4 组十进制的 8 字节数字组成，每组数字由圆点隔开。地址不以零结尾。正则表达式是一种文本模式匹配语言，无法检测数字的范围，故只能指定这些数字如何出现。由于它们可重复，下面先处理一个数字的匹配问题：

```
(1??\d{1,2}|2[0-4]\d|25[0-5])
```

第一个选项包含了可能匹配的最多数字。它定义了一个懒惰选项 1，之后是一个或两个数字。这允许匹配 0～199 之间的数字。选择了一个懒惰通配符，因为不以 1 开头的数字比以 1 开头的数字多，所以首先匹配这些数字会更有意义。然后是一个选择字符|，它表示第一个数字是 2，它的后面是 0～4 之间的数字，再后面是 200～249 之间的数字。最后一组指定可以使用 25，后跟 0～5 之间的任何数字。然后把它们放在一个组中，指定它是一个完整的表达式，或用句点括起来，说明它是一个完整的表达式。

匹配这个 IP 地址的一部分是非常麻烦的。表达式如下：

```
^((1??\d{1,2}|2[0-4]\d|25[0-5])\.){3}(1??\d{1,2}|2[0-4]\d|25[0-5])$
```

首先指定字符串的开头，然后额外添加一个组括号，因为它要检查模式是否重复。这个模式与前面的那个是相同的，模式后面有一个句号。这个匹配进行得很好。然而，如果要划分出 IP 地址的组成，那是做不到的。除非去掉这个模式的重复。而且，“\.”必须包含在一个非捕获组内。如果这是一个更大的表达式的一部分，就应删除“^”和“$”。

这个正则表达式很好地匹配一个 IP 地址，但是您可能还想包含几个以 10、127 或者以 192.168 开头的地址，作为私有地址范围起始的地址。这取决于应用程序的用途。

7.5.2 验证域名

自从允许使用 Unicode 域名之后，事情就变得复杂起来。不过，电子邮件 RFC 还不允许使用这种域名，所以我们仍遵守传统定义，看一看如何利用 ASCII 码来描述一个域。域名由一个被圆点隔开的单词列表组成，最后一个单词有 2～4 个字符。以前通常会有这种情况：如果使用了两个字母的国家名，那么在它之前的域名最少会有两部分——一个用于组的域(.co, .ac 等)和它前面的一个具体的域名。不过，随着.tv 名的消失，情况又有所变化。我们可以将域名制定得非常详细，提供允许的顶级域(Top－Level Domain，TLD)，但是这将使正则表达式变得非常庞大，而执行 DNS 查找时会

更加有效。

域名的每一部分都有它必须要遵守的规则。它可以包含任何字母、数字或连字符，但必须以字母开头。但也有例外的情况，在域名的任一点，都可以使用后面带有数字的#符号，数字代表那个字母的 ASCII 码，或者 Unicode 形式的 16 位 Unicode 值。了解了这些，就可以建立正则表达式了，首先是名称部分，假设设置了 IgnoreCase 选项：

```
([a-z]|#\d+)([a-z0-9-]|#\d+)*([a-z0-9]|#\d+)
```

它把这个域分成了 3 个部分。每一部分都可以使用#nn 符号，所以不必提及它们。RFC 没有指定这里能够包含多少个数字，所以我们也不指定。第一部分必须只包含一个 ASCII 字母；第二部分必须包含零个或多个字母、数字或连字符；第三部分必须包含一个字母或数字。顶级域有更多的限制，如下所示：

```
[a-z]{2,4}
```

这就限制为 2，3 或 4 个字母的顶级域。用句点将它们连在一起：

```
^(([a-z]|#\d+?)([a-z0-9-]|#\d+?)*([a-z0-9]|#\d+?)\.)+([a-z]{2,4})$
```

域名在字符串的开始和结尾部分都被锚定了。首先添加一个附加组支持一个或多个“name.”部分，然后，在它自己组的末端指定一个 2～4 个字母的域名。将大多数通配符设置为懒惰。因为很多模式都是相似的，所以这样做是有意义的，否则就需要太多的回溯。不过，第二组使用了贪婪通配符，它会进行尽可能多的字符匹配，直到到达一个它不能匹配的字符。然后，它仅仅回溯一个位置，让第三组进行匹配。这种情况要比懒惰匹配能更有效地利用资源，因为它会不断地进行匹配。每个名称进行一次回溯是可以接受的额外处理量。

7.5.3　验证个人地址

现在可以验证@符号前面的部分了。RFC 指定它能够包含任何代码为 33～126 之间的 ASCII 字符。假设只匹配 ASCII 码，所以引擎只对 128 个字符进行匹配。在这种情况下，排除所要求的值更简单：

```
[^<>()\[\]\\.,;:@"\x00-\x20\x7F]+
```

这个表达式允许使用任意数目的字符，只要这些字符不是包含在方括号内的那些字符。字符“[”、“]”和“\”必须转义。不过，RFC 允许其他类型的匹配。使用“\”字符可以转义前面的任何字符。也就是说，允许在个人地址中使用回车符。然而，不太可能有人使用这种类型的正则表达式，所以，这里不使用 SingleLine 选项。不过，需要允许其他被排斥的字符，所以添加一个 | 匹配：

\\.

另外，还有一种用作信箱名的格式，是一个带引号的字符串。它的格式是任意字符串，包括空格，放在 3 对双引号字符中。这个字符串能够包含除回车符、换行符、双引号或反斜线之外的任何字符。它的正则表达式如下：

```
"""[^\x0A\x0D"\\]+"""
```

但可以利用反斜线转义带引号的字符串中的字符。不过，由于不允许使用回车符，所以需要转义的惟一字符是“\”：

```
"""([^\x0A\x0D"\\]|\\\\)+"""
```

信箱名的完整正则表达式如下：

```
^(([^<>()\[\]\\.,;:@"\x00-\x20\x7F]|
\\.)+|("""([^\x0A\x0D"\\]|\\\\)+"""))$
```

很明显，这个表达式应该位于一行，并且必须匹配全部的字符串。

7.5.4 验证完整的地址

看完了前面的内容，就能够建立完整电子邮件地址的正则表达式了。首先，检查“@”符号之前的所有字符，包括“@”符号：

```
^(([^<>()\[\]\\.,;:@"\x00-\x20\x7F]|
\\.)+|("""([^\x0A\x0D"\\]|\\\\)+"""))@
```

这是比较简单的。现在检验域或 IP 地址部分。首先检验域名：

```
^(([^<>()\[\]\\.,;:@"\x00-\x20\x7F]|\\.)+|
("""([^\x0A\x0D"\\]|\\\\)+"""))@(([a-z]|#\d+?)([a-z0-9-]|#\d+?)*
([a-z0-9]|#\d+?)\.)+([a-z]{2,4})
```

这个表达式用于匹配传统的电子邮件地址类型。可是，我们也希望它用于 IP 地址，这是很容易做到的。如果单击 ExplicitCapture 选项，并如下所示添加命名的组，它就可以匹配人名和域名了(或 IP 地址)。

```
^(?<person>([^<>()\[\]\\.,;:@"\x00-\x20\x7F]|\\.)+|
("""([^\x0A\x0D"\\]|\\\\)+"""))@(?<domain>(([a-z]|#\d+?)
([a-z0-9-]|#\d+?)*([a-z0-9]|#\d+?)\.)+([a-z]{2,4})|((1??\d{1,2}|
2[0-4]\d|25[0-5])\.){3}(1??\d{1,2}|2[0-4]\d|25[0-5]))$
```

这是一个相当长的表达式，但是在下载的代码中它是一个枚举的一部分。它会匹配任何类型的电子邮件地址，除了那些转义了回车符的地址。在正则表达式检测器中检

测一下，它将运行良好。我们必须在域名和 IP 地址中选出一个放在它自己的组中，以使每一个都能匹配到字符串的结尾部分。

7.6　分析 SMTP 日志文件

本节包含一个分析系统文件的示例。在这个示例中，我们要利用正则表达式从 SMTP 日志文件中提取信息——更精确地说，是从 IIS 的 SMTP 日志文件中提取。为了运行这个示例，日志文件应有如下的扩展属性：

- date-time
- c-ip
- cs-username
- cs-method
- cs-uri-query
- sc-bytes
- cs-bytes
- time-taken

这些属性是在 IIS 或 Exchange 2000 的交换管理器上 SMTP 虚拟服务器属性的 Extended Properties 选项卡中设置的，如图 7-1 所示。

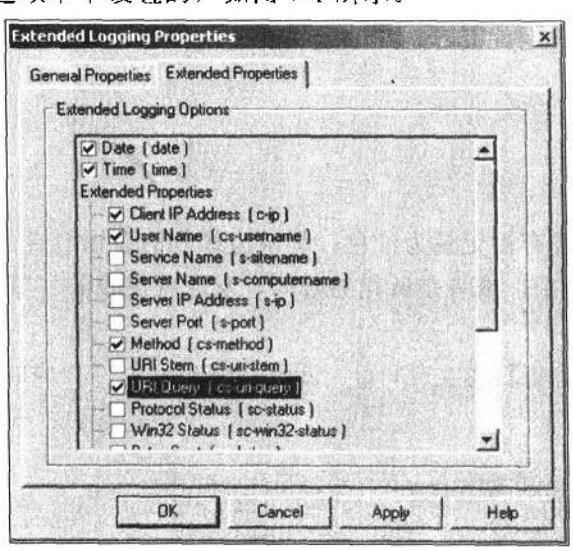


图　7-1

这个程序允许用户选择一个日志文件，并会显示一个 SMTP 会话和发生的动作的列表，并把所有潜在的可疑会话(例如，黑客的攻击)标记为红色。如图 7-2 所示。

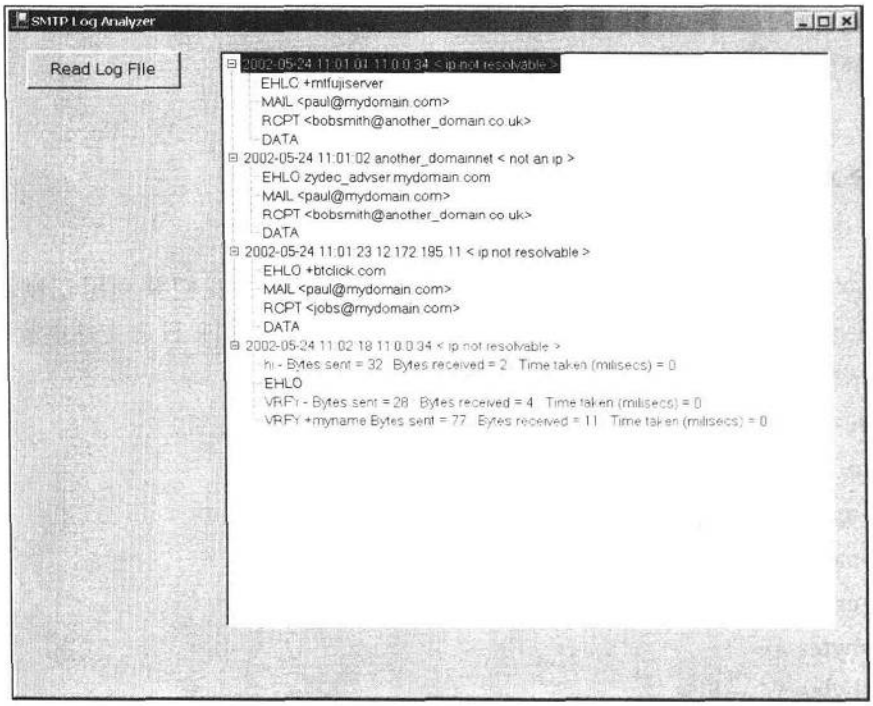


图　7-2

大部分 SMTP 会话是由下面这些内容组成的：

HELO or EHLO
MAIL
RCPT
DATA
QUIT

所以，那些会话没有被记录为红色。记录为红色的会话包含了除上述 SMTP 命令之外的一个 SMTP 命令，或者是所花的时间、发送或接收的字节的大小不同于通常的大小。

首要任务是自己分析日志文件，看一看这些信息是怎样构成的；下面是一个日志文件的示例：

```
#Software: Microsoft Internet Information Services 5.0
#Version: 1.0
#Date: 2002-05-24 11:01:01
#Fields: date time c-ip cs-username cs-method cs-uri-query sc-bytes cs-bytes time-taken
2002-05-24 11:01:01 11.0.0.34 mtfujiserver EHLO +mtfujiserver 331 17 170
2002-05-24 11:01:01 11.0.0.34 mtfujiserver MAIL +FROM:+<paul@mydomain.com> 42 30 40
```

2002-05-24 11:01:01 11.0.0.34 mtfujiserver RCPT +TO:+<bobsmith@another_domain.co.uk> 32 30 0
2002-05-24 11:01:01 11.0.0.34 mtfujiserver DATA +<000001c20312$8844cc60$0400000a@mtfujiserver> 129 501 370
2002-05-24 11:01:02 - OutboundConnectionResponse – 220+another_domainnet+Mail+Service+ESMTP+(oceanus.uk.another_domain.net) 54 0 781
2002-05-24 11:01:02 another_domainnet OutboundConnectionCommand EHLO zydec_advser.mydomain.com 4 0 781
2002-05-24 11:01:02 another_domainnet OutboundConnectionResponse – 250-oceanus.uk.another_domain.net+Hello+host113-113-204-18.in-addr.btopenworld.com+[213.123.214.118] 92 0 821
2002-05-24 11:01:02 another_domainnet OutboundConnectionCommand MAIL FROM:<paul@mydomain.com> 4 0 821
2002-05-24 11:01:02 another_domainnet OutboundConnectionResponse – 250+<paul@mydomain.com>+is+syntactically+correct 48 0 851
2002-05-24 11:01:02 another_domainnet OutboundConnectionCommand RCPT TO:<bobsmith@another_domain.co.uk> 4 0 861
2002-05-24 11:01:04 11.0.0.34 mtfujiserver QUIT mtfujiserver 74 4 0
2002-05-24 11:01:23 12.172.195.11 btclick.com EHLO +btclick.com 336 16 30
2002-05-24 11:01:23 12.172.195.11 btclick.com MAIL +FROM:<paul@mydomain.com> 42 38 0
2002-05-24 11:01:23 12.172.195.11 btclick.com RCPT +TO:<jobs@mydomain.com> 30 27 0
2002-05-24 11:01:23 12.172.195.11 btclick.com DATA +<000101c20312$955140a0$0400000a@mtfujiserver> 129 664 311
2002-05-24 11:01:26 12.172.195.11 btclick.com QUIT btclick.com 74 4 0
2002-05-24 11:01:43 another_domainnet OutboundConnectionResponse – 250+<bobsmith@another_domain.co.uk>+verified 34 0 41389
2002-05-24 11:01:43 another_domainnet OutboundConnectionCommand DATA - 4 0 41399
2002-05-24 11:01:43 another_domainnet OutboundConnectionResponse – 354+Enter+message,+ending+with+"."+on+a+line+by+itself 54 0 41419
2002-05-24 11:01:43 another_domainnet OutboundConnectionResponse – 250+OK+id=17BCpT-000G0E-00 26 0 41519
2002-05-24 11:01:43 another_domainnet OutboundConnectionCommand QUIT - 4 0 41529
2002-05-24 11:01:43 another_domainnet OutboundConnectionResponse – 221+oceanus.uk.another_domain.net+closing+connection 43 0 41549
2002-05-24 11:02:18 11.0.0.34 - hi - 32 2 0
2002-05-24 11:02:22 11.0.0.34 - EHLO - 331 4 0
2002-05-24 11:02:26 11.0.0.34 - VRFY - 28 4 0

```
2002-05-24 11:02:31 11.0.0.34 - VRFY +myname 77 11 0
2002-05-24 11:02:36 11.0.0.34 - QUIT - 77 11 4186
```

首先要注意，每一个单独的命令都独立成行。上面的例子看起来不是这样，那是因为受到本书宽度的限制；每一行都以日期开头。数据的每一个元素(date、time、client-ip，等)都由空格隔开。分别从每一行获取数据应该是相当容易的。比较棘手的就是保证每一个会话的所有信息都聚集到一起——它特别麻烦，因为服务器具有大量的并发会话，正如上面那个日志中的那样。这就意味着我们无法按行分隔独立的会话；例如，日志的顶部是一个发送电子邮件的标准 SMTP 会话：

```
2002-05-24 11:01:01 11.0.0.34 mtfujiserver EHLO +mtfujiserver 331 17 170
2002-05-24 11:01:01 11.0.0.34 mtfujiserver MAIL +FROM:+<paul@mydomain.com> 42 30 40
2002-05-24 11:01:01 11.0.0.34 mtfujiserver RCPT +TO:+<bobsmith@another_domain.co.uk>
32 30 0
2002-05-24 11:01:01 11.0.0.34 mtfujiserver DATA
+<000001c20312$8844cc60$0400000a@mtfujiserver> 129 501 370
```

然而，在此出现的不是一个 QUIT 命令行，日志现在包含的是另外一个独立并发会话的数据：

```
2002-05-24 11:01:02 - OutboundConnectionResponse –
220+another_domainnet+Mail+Service+ESMTP+(oceanus.uk.another_domain.net) 54 0 781
```

这个会话持续了几行，第一个会话的最后一行又突然出现：

```
2002-05-24 11:01:04 11.0.0.34 mtfujiserver QUIT mtfujiserver 74 4 0
```

这意味着正则表达式必须分别匹配每一行和行中的数据，并且要以某种方式保证一个单独会话的各行包含在一个匹配内。可以编写这样一个表达式，但是，它会极其复杂，跨许多行，并很难维护。有时使用正则表达式，需要知道什么时候限制复杂性——否则，最后会得到几大篇正则表达式，它非常复杂，人类基因组的测试工作也“望尘莫及”。

下面在一个匹配中捕获每一行，并把每一个数据项放到它自己的组中。那就意味着必须使用代码将这些会话放置在一起，不过它要比使用正则表达式容易得多——而且也更加有效。

```
^(?<date>[^ #]+) (?<time>[^ ]+) (?<clientAddress>[^ -]+)
(?<clientName>[^ ]+) (?<method>[^ -]+) (?<uriQuery>[^ ]+)
(?<sentBytes>[\d]+) (?<receivedBytes>[\d]+) (?<timeTaken>[\d]+)
```

页面宽度的限制将会使单行的正则表达式分成 3 行。在正则表达式中有 9 个组，每个组都会有一个名称，说明组所包含的数据。组的名称并不是必需的，但它易于代码

的创建和理解。注意，前两行的末尾处有一个空格字符。

这个正则表达式在表 7-4 中分解。这个表显示了当它匹配下面这个日志行时所出现的情况：

2002-05-24 11:01:01 11.0.0.34 mtfujiserver EHLO +mtfujiserver 331 17 170

表 7-4　正则表达式的详解

正则表达式	描　　述	匹配的内容
^	稍后将在代码中指定 MultiLine 选项；这说明^匹配的是每一行的开头。没有设置这个选项时，它只会匹配输入字符串的开头	每一个新行的开头
(?<date>[^ #]+)	这是一个捕获组，名为 date，它的模式是一个或多个空格或散列(#)字符。“#”字符在日志中表示一个注释行，所以代码将忽略这一行。空格是日志数据字段之间的分隔符。由于这是一个命名的组，即使设置了 ExplicitCapture 选项，它也总是捕获这个组	2002-05-24
(?<time>[^]+)	这个组名为 time，匹配一个或多个非空格字符	11:01:01
(?<clientAddress>[^ -]+)	clientAddress 组匹配一个或多个非空格或非“-”的字符。当没有数据时，IIS 插入一个“-”。我们对没有客户端地址的日志不感兴趣，因为这表示接收到的信息，而不是 SMTP 命令	11.0.0.34
(?<clientName>[^]+)	clientName 组匹配一个或多个非空格字符	mtfujiserver
(?<method>[^ -]+)	method 组匹配一个或多个非空格或者非“-”的字符	EHLO
(?<uriQuery>[^]+)	uriQuery 组匹配一个或多个非空格字符	+mtfujiserver
(?<sentBytes>[\d]+)	sentBytes 组匹配一个或多个数字	331
(?<receivedBytes>[\d]+)	receivedBytes 组匹配一个或多个数字	17
(?<timeTaken>[\d]+)	timeTaken 组匹配一个或多个数字	170

分析了正则表达式后，下面创建一个新的 Windows 工程，利用正则表达式来分析日志。

打开 Visual Studio .NET，创建一个新的 Windows 工程 SMTPLogAnalyze。添加一个按钮、一个打开文件对话框和一个树型视图控件。将按钮命名为 cmdReadLogFile，打开文件对话框和树型视图控件使用它们的默认名称 OpenFileDialog1 和 TreeView1。

开始部分的代码如下：

```
using System;
using System.Drawing;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Collections;
using System.Text.RegularExpressions;
using System.IO;
using System.Net;

namespace CSharpSMTPLogAnalyser
{
  public class frmSMTPLogAnalyzer : System.Windows.Forms.Form
  {
```

把代码添加到 cmdReadLogFile.Click 事件中来检验代码本身，这些代码将显示打开文件对话框，允许用户选择要分析的日志文件：

```
private void cmdReadLogFile_Click(object sender, System.EventArgs e)
{
  OpenFileDialog1.ShowDialog();
}
```

现在需要为 OpenFileDialog1.FileOk 事件添加代码；这些代码读取文件并分析文件。代码非常多，所以这里分几次列出它们。下面从读取日志文件的代码开始：

```
private void OpenFileDialog1_FileOk(object sender,
        System.ComponentModel.CancelEventArgs e)
{
  try
  {
    // Clear any pre-existing details;
    TreeView1.Nodes.Clear();

    // Create a hashtable object to hold details of sessions;
    Hashtable ipIndex = new Hashtable();
    // read in the file to be analysed; need to make a temporary
    // copy of the file because IIS locks the latest log entry;
    String logFileText;
    String filePath = OpenFileDialog1.FileName;
    String tempFile = Path.GetTempFileName();
```

```
File.Copy(filePath, tempFile, true);
StreamReader streamReaderRegex =
        File.OpenText(tempFile);
logFileText = streamReaderRegex.ReadToEnd();

streamReaderRegex.Close();
File.Delete(tempFile);
```

放在 try...catch 语句开头的代码，先是清除了 TreeView1，然后创建了一个新的 Hashtable 对象。稍后将使用这个对象存储 SMTP 会话的详细资料和它们在 TreeView 节点中的位置。

接下来，获取用户选择的日志文件名，并取得了一个临时文件名。不能在 IIS 日志文件中直接读取文件，因为如果是当天的日志文件，并且 IIS 正在使用它，它就会被锁定——任何打开它的尝试都会引发一个错误。解决这个问题的最简单的方法，也是这里选择的方法，就是对日志文件做一个临时副本，然后读取它的信息，最后再将这个临时文件删除。

利用 File 对象的 Copy()共享方法复制这个临时文件，然后利用 StreamReader 和 File 打开这个临时文件。使用 StreamReader，读到文件的结尾，将其关闭，最后将它删除。

现在需要创建正则表达式，来匹配日志的每一行，并在一个具体的命名组中捕获它的数据：

```
// get the individual log lines and data within them;
Regex logLinesRegex = new Regex("^(?<date>[^ #]+) (?<time>[^ ]+) " +
                                "(?<clientAddress>[^ -]+) " +
                                "(?<clientName>[^ ]+) " +
                                "(?<method>[^ -]+) " +
                                "(?<uriQuery>[^ ]+) " +
                                "(?<sentBytes>[\\d]+) " +
                                "(?<receivedBytes>[\\d]+) " +
                                "(?<timeTaken>[\\d]+)",
                                RegexOptions.Multiline);

Regex getEmailAddressDataRegex = new Regex("<[^>]+>",
    RegexOptions.Compiled);

MatchCollection logLineMatches;

string clientAddress;
string clientHostName = "";
```

```
string method;
string uriQuery;
TreeNode logNode;
Boolean suspectSession = false;

logLineMatches = logLinesRegex.Matches(logFileText);
```

现在获取了日志数据，每一个匹配都包含了一行日志，匹配中的每一个组都包含了需要的数据。现在必须循环执行这些匹配，提取数据，并把它追加到 TreeView 控件的节点上。然而，这样做要复杂一些，因为需要确定将一个会话的所有行都放在该会话的正确节点下。因为有并发的 SMTP 会话，并且日志中一个会话的各行按次序运行，所以各行之间可能会有其他的会话。代码需要考虑到这些情况，并跟踪每个会话，跟踪会话是在早先创建的 ipIndex Hashtable 对象中，借助客户端 IP 地址来完成的。每次发现一个新的会话，代码都会在 TreeView 控件的 Nodes 属性中对该节点的索引做一个注释。之后，向相同的节点添加新的日志行，直到日志中这个会话结束。

再来看看代码，用 foreach 循环来迭代 logLinesMatches MatchCollection 对象中的每一次匹配：

```
// Go through each line matched in the log;
foreach (Match logLineMatch in logLineMatches)
{
  // Obtain the data in uriQuery, method and clientaddress;
  uriQuery = logLineMatch.Groups["uriQuery"].Value;
  method = logLineMatch.Groups["method"].Value;
  clientAddress = logLineMatch.Groups["clientAddress"].Value;
```

在循环顶部，获取了该日志行的 uriQuery、method 和 clientAddress 值。这些详细信息都放在正则表达式匹配的组中；由于每一组都有名称，所以可以按照名称提取它们，而不必记住这些组的顺序。如果有相当多的组，记顺序很容易出错。

下面的代码试图提取出异常的 SMTP 会话。它判断会话异常的依据在于，SMTP 会话包含了不常使用的命令，或者会话特别庞大，或者它占用了相当长的时间。检验非常简单，如果程序进一步开发，它会变得更加复杂：

```
// If the SMTP method is ! mail, rcpt, helo, ehlo || data
// or if the time taken is more than a minute
// or the sent bytes are more than 1mb;
// or the received bytes more than a 1mb;
// then this might be a suspect line;
if (method != "MAIL" && method != "RCPT" && method != "HELO" &&
        method != "EHLO" && method != "DATA" ||
```

```
    ( Int32.Parse(logLineMatch.Groups["timeTaken"].Value) > 60000
    ||
    Int32.Parse(logLineMatch.Groups["sentBytes"].Value) > 1048576
    ||
    Int32.Parse(logLineMatch.Groups["receivedBytes"].Value) > 1048576))
  {
    uriQuery = uriQuery + " Bytes sent = " +
        logLineMatch.Groups["sentBytes"].Value;

    uriQuery = uriQuery + " : Bytes received = " +
        logLineMatch.Groups["receivedBytes"].Value;

    uriQuery = uriQuery +" : Time taken (milisecs) = " +
        logLineMatch.Groups["timeTaken"].Value;

    suspectSession = true;
  }
  else
  {
    suspectSession = false;
  }
```

如果会话可疑，就向 uriQuery 数据添加额外的信息，帮助用户弄清楚为什么会话是可疑的。然后，把 Boolean 变量 suspectSession 设置为 True。稍后，代码就会检查这个变量，并把这个可疑会话突出显示为红色。

如果行中包含了一个 MAIL(邮件发出的地址)或者 RCPT(接收邮件的地址)命令，那么提取涉及到的邮件地址并在分析中显示是很有帮助的。这就是下面的代码完成的任务：

```
// if (the mail || rcpt methods are used) {
// extract the e-mail address the mail is;
// being sent to or from;
if (method == "MAIL" || method == "RCPT")
{
  uriQuery = getEmailAddressDataRegex.Match(uriQuery).Value;
}
else if (uriQuery == "-" || method == "DATA")
{
  uriQuery = "";
}
```

我们得到了所需的数据，现在，该向 TreeView 添加新的节点了。新的节点有 3 种

可能性：

- 它是一个新的会话
- 它是一个现有会话的 QUIT 命令
- 它是现有会话的另外一行

在下面的 If 语句中，代码会检验是否有一个现有的会话节点。如果这个节点存在，它会接下来检查 SMTP QUIT 命令。如果它就是 QUIT 命令，就不需要添加日志行了，因为所有会话必须有一个 QUIT 命令——即使会话已经超时。另一种情况是仅把 IP 地址从 ipIndex 清除，ipIndex 包含了日志中目前打开的所有会话。如果它不是 QUIT 命令，那么代码需要找到那个会话的顶层节点的索引，并在它的下面添加一个新的节点，其中包含最新的日志行和详细资料。注意，如果变量 suspectSession 为 True，就必须把当前日志行和会话节点行都突出为红色。完成这些任务的代码如下：

```
// Has this IP been found before?;
if (ipIndex.ContainsKey(clientAddress))
{
  // if (QUIT method found) { current session has ended for that IP
  // and we need to remove details of its node from the session index;
  if (method == "QUIT")
  {
    ipIndex.Remove(clientAddress);
  }
  else
  {
    // only add the line if the method was ! empty
    // IIS indicates no data with a minus sign;
    if (method != "-")
    {
      TreeView1.Nodes[Int32.Parse(
          ipIndex[clientAddress].ToString())].Nodes.Add(method +
          " " + uriQuery);

      // if (this is one of the lines noted as being suspect)
      // make the line and the session line appear in red;
      // and add extra data for info
      if (suspectSession)
      {
        TreeView1.Nodes[Int32.Parse(
            ipIndex[clientAddress].ToString())].LastNode.ForeColor
            = Color.Red;
```

```
                TreeView1.Nodes[
                    Int32.Parse(ipIndex[clientAddress].ToString())].ForeColor
                    = Color.Red;
            }
        }
    }
}
```

另外一种情况是，这是一个新的会话，需要获取使用 DNS 的服务器名，如果失败，要向 ipIndex 添加这个会话的 IP 地址，然后向 TreeView 控件添加一个新的顶层节点，接着向它添加日志行的节点的详细资料：

```
else
{
    // This is a new session - create a new top level node;
    // and record session's location in tree;
    logNode = new TreeNode();
    try
    {
        clientHostName = " <" +
            Dns.GetHostByAddress(clientAddress).HostName + ">";
    }
    catch (Exception ex)
    {
        if (ex is System.FormatException)
        {
            clientHostName = " < not an ip >";
        }
        else if (ex is System.Net.Sockets.SocketException)
        {
            clientHostName = " < ip not resolvable >";
        }
    }

    logNode.Text = logLineMatch.Groups["date"].Value + " " +
                   logLineMatch.Groups["time"].Value + " " +
                   clientAddress +
                   clientHostName;

    logNode.Nodes.Add(method + " " + uriQuery);
```

```
            TreeView1.Nodes.Add(logNode);
            ipIndex.Add(clientAddress, TreeView1.GetNodeCount(false) - 1);

            // Display suspect sessions in red;
            if (suspectSession)
            {
              TreeView1.Nodes[Int32.Parse(ipIndex[
                  clientAddress].ToString())].LastNode.ForeColor = Color.Red;

              TreeView1.Nodes[Int32.Parse(ipIndex[
                  clientAddress].ToString())].ForeColor = Color.Red;
            }
          }
```

最后关闭 if 和 foreach 语句，为开始的 try 块提供一个 catch 块，并关闭 OpenFileDialog1_FileOk()方法、窗体和命名空间：

```
              }
            }
            catch (Exception ex)
            {
              MessageBox.Show("Error occurred - " + ex.Message);
            }
          }
        }
      }
```

这个应用程序可以用来分析所有的 SMTP 日志文件。

7.7 HTML 标记

这一节学习如何匹配 HTML 标记。开始要创建一个正则表达式，用以识别字符串是否包含了 HTML 开始或结束标记。当接受用户在一个 Web 页面的输入时，如果要避免他们插入自己的 HTML 标记，比如图像、链接或 JavaScript 的标记，就需要用到这个正则表达式。我们可以进行更深层次的识别，并允许使用某些 HTML 标记，比如格式化标记，但是现在只查找任何一种类型的标记。

第二个正则表达式不是用来验证数据的，而是把一个 HTML 文件拆分成组件标记和属性。我们将创建一个示例，即一个 Web 页的 HTML 的树型视图。

和创建正则表达式一样，第一个任务是写出需要匹配的模式和模式的变化；对于

HTML 来说就是下面的模式：

```
<tag>Text to retain</tag>
<tag>
<tag tagattribute1="somevalue">
<tag tagattribute1="somevalue" tagattribute2="somevalue">
<tag tagattribute1="somevalue">Text to retain</tag>
<tag tagattribute1=somevalue>Text to retain</tag>
<tag tagattribute1='somevalue' tagattribute2='somevalue'>
  Text to retain
</tag>
<tag></tag>
<tag attribute1="somevalue" attribute2="somevalue></tag>
```

下面先看看如何在一个输入字符串内部匹配这些标记。

7.7.1　从用户输入中清除 HTML

匹配开始或结束 HTML 标记的正则表达式如下：

```
(<[a-z]+[^>]*>)|(</[a-z\d]+>)
```

事实上，它是带有替换字符的两个正则表达式，允许模式匹配其中任一个 HTML 标记。左边的模式：

```
(<[a-z]+[^>]*>)
```

匹配一个 HTML 开始标记。它查找一个“<”字符，后跟 a 到 z 之间的任意一个或多个字母，接着是零个或多个任意字符，只要不是结束标记字符“>”。那么，为什么不能用下面的表达式：

```
(<[^>]+>)
```

它可以匹配任何 HTML 开始标记，不过，它也可以匹配“<”字符和没有 HTML 的情况。例如：

```
The value was < 22,000 but always > 10,000
```

这将导致一个错误匹配。通常对于正则表达式，既定义想匹配什么，也定义不想匹配什么。要求“<”之后紧跟字母 a-z，会减少错误匹配的机会。

正则表达式中用来匹配结束标记的第二部分如下：

```
(</[a-z\d]+>)
```

结束标记没有属性，所以正则表达式可以简化。它匹配一个"<"字符，后跟一个"/"，然后是 a 到 z 之间的一个或个字母或者数字。最后，它匹配"> "字符。

可以使用这个正则表达式查看是否存在一个匹配，如果有，意味着用户已经在输入中插入了 HTML：

```
if (Regex.IsMatch(userInputString,
                  "(<[a-z]+[^>]*>)|(</[a-z\\d]+>)",
                  RegexOptions.IgnoreCase |
                  RegexOptions.ExplicitCapture))
{
  MessageBox.Show("HTML FOUND");
}
else
{
  MessageBox.Show("HTML NOT FOUND");
}
```

也可以利用 Replace()方法将这些讨厌的标记全部清除，只留下普通文本。

7.7.2 提取所有 HTML 标记

这一节介绍正则表达式是如何从 HTML 页面中提取详细信息的。在这个示例中，我们将建立一个 TreeView 控件，该控件给出了一个 Web 页及其标记的详细描述。不过，在分析一个 HTML 页面的具体组成部分时，正则表达式本身是很有用的。

提取 HTML 标记及其属性的正则表达式要复杂多了：

```
<(?<outertag>[a-z]+[\d]?)(?<attributes> [^>]*)*>
(?<innerhtml>(<(?<innertag>[a-z]+[\d]?)[^>]*>.*?</\k<innertag>>|
<[a-z]+[\d]?[^>]*>|(?>[^<]*))*(?=</\k<outertag>>))?
```

正则表达式的目的在于把每一个标记和它的内容都分解为一个单独的匹配，但是在那个匹配内，有一个组匹配 HTML 标记类型，另外一个组匹配标记的属性，所以如果 HTML 如下所示：

```
<p>
  Welcome to Wrox
</p>
<img src="ThePlane.jpg" width="149">
<hr>
<p align="left">my text</p>
```

就会创建表 7-5 中的匹配和组。

表 7-5　匹配和组

完 全 匹 配	组 1 外部标记	组 2 属性	组 3 内部 html
Welcome to Wrox	P		Welcome to Wrox
<img src="ThePlane.jpg" width="149" height="323">	Img	src="ThePlane.jpg" width="149" height="323"	
<hr>	Hr		
my text	P	align="left"	my text

要解决的一个最大问题是所嵌套的标记，例如：

```
<p>
  <p>some text<img src="myimg.jpg">
  <a href="somepage.htm">click here</a></p>
</p>
```

问题是强制正则表达式找出正确的结束标记。在本例中，第一个 的结束标记其实就是最后一个 标记。现在需要找出处理这个问题的方法。我们所使用的技术是在外部标记内使用 HTML；使用这种技术，正则表达式引擎除了匹配最后一个 标记之外别无选择。

为运行正则表达式，需要设置下面的选项：

- IgnoreCase
- ExplicitCapture
- SingleLine

我们不关心 HTML 的大小写，所以设置选项 IgnoreCase。而且，由于只关注 4 个组的内容，所以为了便捷和高效，设置了 ExplicitCapture 选项，只有命名的组才存储它们的匹配。SingleLine 选项允许“.”字符匹配新行，因为一个标记的 HTML 可能会在多行出现。

这个正则表达式非常庞大，所以要对它进行分解，首先分解出匹配第一个开始标记的正则表达式模式：

```
<(?<outertag>[a-z]+[\d]?)(?<attributes> [^>]*)*>
```

这个表达式匹配像 和<img src="ThePlane.jpg" width="149">这样的标记。

首先，它匹配一个 HTML 标记的“<”字符，然后试着匹配一个名为 outertag 的组。这个组将要匹配形如 h1、 p 和 img 的标记类型。我们使用这个组获取最初的结果，然

后匹配结束标记类型。接下来是 attributes 组：这个组可以匹配任意 HTML 标记属性，例如 src="myfile.jpg"。若把它放在它自己的组中，就很容易在 Visual Basic .NET 代码中访问匹配的结果。

一些单元素标记只有一个开始标记，例如<img>标记，所以前面匹配的模式就足够了。但大多数的元素有一个结束标记，所以需要模式的其余部分：

```
(?<innerhtml>(<(?<innertag>[a-z]+[\d]?)[^>]*>.*?</\k<innertag>>|
<[a-z]+[\d]? [^>]*>|(?>[^<]*))*(?=</\k<outertag>>))?
```

稍后分析这个表达式，它可以匹配任何 HTML 开始和结束标记对，或是任何单独的 HTML 标记，或是任何后跟结束标记的普通文本，它还匹配前面匹配过的外部开始标记。

外部组就是 innerHTML 组；它匹配外部开始标记和最后的结束标记之间的所有内容，所以如果有下面的 HTML：

```
<p>
  <p>some text<img src="myimg.jpg">
  <a href="somepage.htm">click here</a></p>
</p>
```

那么 innerhtml 组将匹配如下内容：

```
<p>some text<img src="myimg.jpg">
<a href="somepage.htm">click here</a></p>
```

如果外部结束标记存在，innerhtml 组只执行一次匹配，但外部结束标记既没有在组内捕获，也没有在匹配中捕获。

把这个组分解。在 innerhtml 组内是另外一个非捕获组，非捕获是因为使用了 ExplicitCapture 选项。组内有 3 个独立的模式，每个模式都用“|”字符隔开，正则表达式的“|”字符相当于逻辑 OR。这 3 个组和它们匹配的内容如表 7-6 所示。

表 7-6 正则表达式模式和它们的匹配

正则表达式	描 述	示 例 匹 配
(?<innertag>[a-z]+[\d]?) [^>]*>.*?</\k<innertag>>	匹配一个开始 HTML 标记，一个结束 HTML 标记，和它们两者中间的内容	 some text
<[a-z]+[\d]?[^>]*>	它匹配一个没有结束标记的 HTML 开始标记	<img>
(?>[^<]*)	它匹配 HTML 标记之外的任何内容	Some plain text

“|”字符首先试着匹配左边的表达式，若没有找到要匹配的内容，再匹配右边的

表达式。我们可以利用这个规则，因为它意味着模式会首先寻找一个带有结束标记的 HTML 开始标记，比如 some text 。如果没有这样的匹配，它就会寻找一个不带匹配结束标记的 HTML 标记，比如<img src="image.gif">。如果仍然没有匹配，它就会寻找任何内容，匹配纯文本或换行字符。

从表中的第一个正则表达式开始：

```
(?<innertag>[a-z]+[\d]?)[^>]*>.*?<\k<innertag>>
```

这与匹配外部 HTML 标记的正则表达式十分相似。首先，创建一个名为 innertag 的显式捕获组；innertag 将会捕获开始标记，稍后在模式中用它匹配一个相同类型的结束标记。

然后，要匹配 a 到 z 之间的一个或多个字母，后跟一个可选的数字，即匹配像 h1 和 h2 这样的标记。在 innertag 组之后，进行一个非贪婪匹配，它会寻找零个或多个任意类型的字符，包括换行字符，因为我们设置了 SingleLine 选项。使用“*”的非贪婪模式，因为要限制正则表达式引擎匹配的内容。这里有一个危险：当还有更多的 HTML 标记要单独匹配时，它会匹配它能匹配的任何内容，包括我们不希望匹配的内容，比如字符串的剩余字符。

最后，在结尾处匹配一个与开始标记类型相同的结束标记：

```
<\k<innertag>>
```

表中的第二行是匹配单个开始标记的正则表达式：

```
<[a-z]+[\d]?[^>]*>
```

这几乎与前面的 innertag 组的模式相同，只是不用获取更多的组，因为这里没有结束标记可以匹配。

下面是表中的最后一个正则表达式：

```
(?>[^<]*)
```

它只是匹配了前面没有被匹配的内容，除了“<”字符之外，它会捕获标记之间的任何文本。

我们已经分析了 innerhtml 组；现在只剩下下面的内容了：

```
)*(?=<\k<outertag>>))?
```

Innerhtml 组之后是一个“*”，它告知正则表达式引擎对组的内容进行零次或多次匹配。所以，如果 HTML 如下所示：

```
<p>
  <h1><img>some text<p>more text</p></h1>
</p>
```

第一个<p>标记由外部标记匹配模式匹配，所有的内部 html，包括<h1>标记、<img>标记、文本、内部<p>标记，这个标记的文本等，都会由 innerhtml 组匹配。如果没有“*”，就只匹配<h1>标记。

正则表达式的最后，是匹配与外部 HTML 标记相同类型的结束标记的模式：

```
<(?<outertag>[a-z]+[\d]?)(?<attributes> [^>]*)*>
(?<innerhtml>(<(?<innertag>[a-z]+[\d]?)[^>]*>.*?</\k<innertag>>|
<[a-z]+[\d]?[^>]*>|(?>[^<]*))*(?=</\k<outertag>>))?
```

下一节将把这个正则表达式转变成一个有用的 C#代码示例。

7.7.3 HTML 提取示例

我们要在.NET 中创建一个新的应用程序，以利用刚刚讨论的一些新概念。我们将加载一个 HTML 页面，然后利用上面的正则表达式把 HTML 分解成单独的标记，并把结果显示在一个 TreeView 中。如图 7-3 所示。

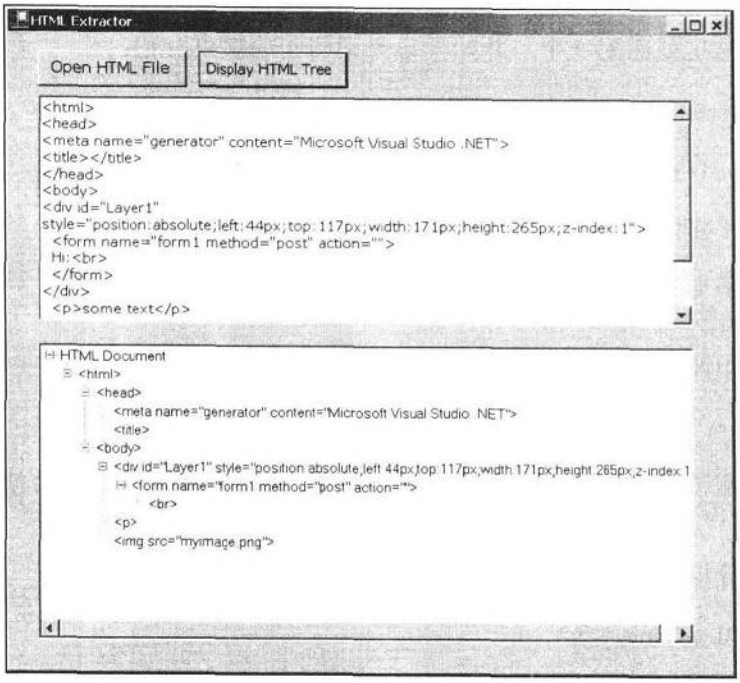


图　7-3

启动 Visual Studio .NET，并启动一个名为 HTMLExtractor 的新 Windows 应用程序。创建一个带控件的窗体。

各个控件的相关属性如表 7-7 所示。

表 7-7　控件的属性

控件类型	属　性
Button	Name =cmdOpenHTML Text = Open HTML File
Button	Name = cmdDisplayHTML Text = Display HTML Tree
Form	Name = frmHTMLExtractor
OpenFileDialog	Name=OpenFileDialog1 Filter = Web Pages|*.htm;*.html;
TextBox	Name=txtInputText MultiLine= True ScrollBars = Vertical
TreeView	Name = HTMLTreeView

首先，在窗体类的顶部，告知编译器要使用 System.Text.RegularExpressions 和 System.IO 命名空间：

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text.RegularExpressions;
using System.IO;

namespace CSharpHTMLExtractor
{
```

在 frmHTMLExtractor 类内部，需要声明两个变量；第一个变量包含了 Regex 对象，第二个变量对已进行的匹配进行记数：

```
public class frmHTMLExtractor : System.Windows.Forms.Form
{
  Int32 intMatchesMade = 0;
  Regex extractHTMLRegex =
      new Regex("<(?<outertag>[a-z]+[\\d]?)(?<attributes> [^>]*)*>" +
"(?<innerhtml>(<(?<innertag>[a-z]+[\\d]?)[^>]*>.*?</\\k<innertag>>|" +
      "<[a-z]+[\\d]?[^>]*>|(?>[^<]*))*(?=</\\k<outertag>>))?",
```

```
        RegexOptions.IgnoreCase |
        RegexOptions.Compiled |
        RegexOptions.ExplicitCapture |
        RegexOptions.Singleline);
```

在上面的代码中，设置了正则表达式进行正确匹配所必需的 3 个选项：IgnoreCase、ExplicitCapture 和 SingleLine。而且，通过设置 Compiled 选项，告诉正则表达式引擎把正则表达式转换成更适合计算机使用的模式，而不是更加“人性化”的语法。如果清除了 Compiled 选项，分析 Web 页所需的时间将会显著增加(大约 50%)。

现在添加读取 HTML 文件的代码；首先是单击按钮的代码：

```
private void cmdOpenHTML_Click(object sender, System.EventArgs e)
{
  OpenFileDialog1.ShowDialog();
}
```

利用 OpenFileDialog1.ShowDialog()方法，代码获取了将要分析的 HTML 文件的名称。下面是 OpenFileDialog1.FileOk 事件的代码：

```
private void OpenFileDialog1_FileOk(object sender,
    System.ComponentModel.CancelEventArgs e)
{
  String filePath = OpenFileDialog1.FileName;
  StreamReader streamReaderRegex = File.OpenText(filePath);
  this.txtInputText.Text = streamReaderRegex.ReadToEnd();
  streamReaderRegex.Close();
}
```

System.IO.StreamReader 类用来读取所选择的文件，并把 txtInputText 框的 Text 属性设置为文件的内容。接下来，需要向 cmdDisplayHTML 按钮的单击事件添加代码；这个事件启动匹配和树的创建：

```
private void cmdDisplayHTML_Click(object sender, System.EventArgs e)
{
  intMatchesMade = 0;
  HTMLTreeView.Nodes.Clear();
  HTMLTreeView.Nodes.Add(populateTagNode(this.txtInputText.Text,
      "HTML Document"));
  HTMLTreeView.ExpandAll();
  MessageBox.Show("Completed " + intMatchesMade + " matches");
}
```

重设 MatchesMade 变量，该变量对已找到的 HTML 标记计数，然后清除 TreeView 控件，为新的 HTML 作准备。接下来，向树添加一个新的树节点，也就是 populateTagNode()方法返回的值，其代码如下所示：

```
private   TreeNode populateTagNode(String sInputString,
    String sTitleText)
{

  TreeNode htmlTagNode = new TreeNode();

  try
  {
    MatchCollection matchesFound;

    TreeNode htmlSubTagNode;
    String sTag;
    matchesFound = extractHTMLRegex.Matches(sInputString);
    htmlTagNode.Text = sTitleText;

    foreach (Match matchMade in matchesFound)
    {
      intMatchesMade = intMatchesMade + 1;
      sTag = "<" + matchMade.Groups[1].Value +
            matchMade.Groups[2].Value + ">";
      htmlSubTagNode = populateTagNode(matchMade.Groups[3].Value,
          sTag);
      htmlTagNode.Nodes.Add(htmlSubTagNode);
    }
  }

  catch (ArgumentException ex)
  {
    MessageBox.Show("The following error occurred "
        + "\r\n" + ex.Message);
  }

  return htmlTagNode;
}
```

这个方法是递归的；它通过获取外部标记，不断地提取嵌套的 HTML 标记，然后

持续调用它自身获取内部标记。下面用一个示例来说明这段代码：

```
<html>
<body>
  <p>some text</p>
  <img src="myimage.png">
</body>
</html>
```

当匹配上面的 HTML 时，只有一个匹配，就是整个的 HTML。不过，第一组是<html>标记，第二组是标记的属性(如果这个标记有属性)，第三组包含了开始<html>标记和结束<html>标记之间的全部 HTML 标记。函数创建了一个名为 html 的 TreeNode。

接下来，这个方法进行自我调用，但是这次传递的不是整个的 HTML 输入字符串，而仅仅是第三组中的 HTML——外部的开始和结束标记之间的 HTML。它将会被分解，这次还是只有一个匹配，body 标记。再次创建一个 TreeNode 并再次调用 populateTagNode()方法，但只提取<body>标记内的 HTML。这一次有两个匹配，<p>标记和 <img>标记。如果嵌套更深，将重复调用 populateTagNode()方法，直到提取出全部的 HTML，并且为每个标记创建一个节点。这个应用程序现在很简单，但是它可以扩展其他的用途。例如，可以检查遗漏的结束标记，或显示笨拙的格式化的 HTML。

7.8 小结

本章利用了在前面两章学到的知识，并把它们付诸实践。我们学习了一系列的正则表达式，从简单的用于数据验证的正则表达式，到较为复杂的用于验证电子邮件地址的正则表达式，还有能够分析日志文件和 HTML 页的多行正则表达式。

我们学习了处理正则表达式的方法，首先是查看要进行匹配的数据。通常，当查看一些数据时，开发人员可以立即辨别出它的模式；例如，不用太费气力就可以猜出 person@somedomain.com 的模式。使用正则表达式，需要不断重复地选择模式：先是如何构建这个模式，然后是使用什么信息来识别它。这些做完之后，还需要找出有效模式的所有组合，重要的是那些不想匹配的模式。进行完这些准备之后，再分阶段构造正则表达式，这样它们就不像乍看起来那么难了。

读完整本书，您已经学习了如何存储文本，如何使用各种工具来构造、匹配、追加和替换字符串。您现在可以在应用程序中游刃有余地操作文本了。接下来的附录提供了 String 类和 StringBuilder 类的方法的快速索引，以及正则表达式模式中用到的各种字符的索引。希望这本书对您将来的工作有所帮助。

附录A String 类

附录 A 包含了 String 类中所有的方法、属性和构造函数。您可以在第 2 章中找到更详细的相关内容。

A.1 构造函数

表 A-1 列出了一个构建新 String 对象实例的各种重载方法。

表 A-1 构建 String 对象实例的各种重载方法

构 造 函 数	描 述
String(char[] charArray)	用一个字符数组 charArray 的内容创建一个新的字符串
String(char c, int count)	创建由指定的字符 c 重复 count 次组成的一个新字符串
String(char[] charArray, int pos, int count)	创建一个新字符串，该字符串由在字符串 charArray 中定义的一个子串组成，子串的开始位置由 pos 定义，包含的字符个数由 count 定义

A.2 属性

表 A-2 列出了 String 类的属性。

表 A-2 String 类的属性

属 性	描 述
Empty	表示空字符串的只读静态常量
Chars(int index)	返回字符串中 index 位置的字符
Length	返回字符串中的字符数

A.3 方法

表 A-3 包含了 String 类中所有的方法以及这些方法的重载方法。表 A-3 中还指明

了该方法是实例方法还是静态方法。

表 A-3 string 类的方法

方 法	访问级别	描 述
Clone()	实例	返回对 String 对象的一个引用。这与对象通常实现的 Clone()方法不同，但却是很合适的，因为 System.String 是不可变的，所以不需要副本
Compare(string s1, string s2)	静态	根据当前文化对两个特定字符串执行一次区分大小写的比较
Compare(string s1, string s2, bool ignoreCase)	静态	根据当前文化对两个特定字符串执行一次比较。如果 ignoreCase 为 true，将执行一个不区分大小写的比较
Compare(string s1, string s2, bool ignoreCase, CultureInfo culture)	静态	根据所提供的文化设置对两个特定字符串执行一次比较。如果 ignoreCase 为 true，则执行不区分大小写的比较
Compare(string s1, int index1, string s2, int index2, int length)	静态	对由给定参数定义的两个子串执行一次区分大小写的比较
Compare(string s1, int index1, string s2, int index2, int length, bool ignoreCase)	静态	对由给定参数定义的两个子串执行一次比较。如果 ignoreCase 为 true，则执行不区分大小写的比较
Compare(string s1, int index1, string s2, int index2, int length, bool ignoreCase, CultureInfo culture)	静态	根据所提供的文化设置，对由给定参数定义的两个子串进行比较。如果 ignoreCase 为 true，执行一次不区分大小写的比较

（续表）

方　法	访问级别	描　述
CompareOrdinal(string s1, string s2)	静态	不考虑文化对两个给定的字符串进行比较
CompareOrdinal(string s1, int index1, string s2, int index2, int length)	静态	不考虑文化对两个给定的子串进行比较
CompareTo(Object obj)	实例	对带有实例字符串的特定对象进行一次区分大小写、区分文化的比较
CompareTo(string s)	实例	对带有实例字符串的特定字符串进行区分大小写、区分文化的比较
Concat(Object obj)	静态	创建给定对象的一个字符串表示
Concat(ParamArray objects)	静态	根据给定数组中的对象连接创建一个字符串
Concat(ParamArray strings)	静态	根据给定数组中的字符串连接创建一个字符串
Concat(Object obj1, Object obj2)	静态	根据两个给定对象的连接创建一个字符串
Concat(string s1, string s2)	静态	根据两个给定字符串创建一个字符串
Concat(Object obj1, Object obj2, Object obj3)	静态	根据 3 个给定对象的连接创建一个字符串
Concat(string s1, string s2, string s3)	静态	根据 3 个给定字符串的连接创建一个字符串
Concat(string s1, string s2, string s3, string s4)	静态	根据 4 个给定字符串的连接创建一个字符串
Copy(string s)	静态	返回一个与给定字符串具有相同值的新字符串

（续表）

方　法	访问级别	描　述
CopyTo(int sourceIndex, char dest(), int destIndex, int count)	实例	从实例字符串中复制一个子串到给定的目标字符串
EndsWith(string s)	实例	如果实例字符串以特定字符串结尾，则返回true
Equals(string s)	实例	如果实例字符串与特定对象具有相同值，则返回true
Equals(string s1, string s2)	静态	如果提供的两个字符串具有相同的值，则返回true
Format(string format, Object arg)	静态	返回一个字符串，由特定参数所替换的格式说明组成
Format(string format, ParamArray args())	实例	返回一个字符串，由参数数组所替换的格式说明组成
Format(IFormatProvider provider, string format, ParamArray args())	实例	返回一个字符串，根据特定格式提供者由参数数组所替换的格式说明组成
Format(string format, Object arg1, Object arg2)	实例	返回一个字符串，由指定的两个参数所替换的格式说明组成
Format(string format, Object arg1, Object arg2, Object arg3)	静态	返回一个字符串，由指定的3个参数所替换的格式说明组成
GetEnumerator()	实例	返回一个对象，可以用来遍历字符串中的字符
GetHashCode()	实例	返回字符串实例的惟一的散列代码
GetType()	实例	返回字符串实例的类型
GetTypeCode()	实例	返回字符串类的类型代码
IndexOf(char c)	实例	返回首次遇到的指定字符的下标
IndexOf(string s)	实例	返回首次遇到的指定字符串的下标

(续表)

方 法	访问级别	描 述
IndexOf(char c, int startPos)	实例	返回从指定位置开始首次遇到的指定字符的下标
IndexOf(string s, int startPos)	实例	返回从指定位置开始首次遇到的指定字符串的下标
IndexOf(char c, int startPos, int count)	实例	从指定位置向后搜索 count 个字符，返回首次遇到的指定字符的下标
IndexOf(string s, int startPos, int count)	实例	从指定位置开始向后搜索 count 个字符，返回首次遇到的特定字符串的下标
IndexOfAny(char[] chars)	实例	返回特定数组中首次遇到的任意字符的下标
IndexOfAny(char[] chars, int startPos)	实例	从指定位置开始，返回特定数组中首次遇到的任意字符的下标
IndexOfAny(char[] chars, int startPos, int count)	实例	从指定位置开始向后搜索 count 个字符，返回在指定数组中首次遇到的任意字符的下标
Insert(int startPos, string s)	实例	在实例字符串中的指定位置插入指定的字符串
Intern(string s)	实例	返回一个特定字符串的引用。如果字符串不是内置的，那么此方法将内置该字符串
IsInterned(string s)	实例	如果特定字符串在内置池中，则返回对它的一个引用。否则返回 Nothing
Join(string separator, String[] arrStr)	实例	返回一个字符串，由特定数组中的字符串连接而成，每个字符间都有特定的分隔符
Join(string separator, String[] arrStr, int startIndex, int count)	实例	返回一个字符串，由特定数组中的字符串连接而成，每个字符间都有特定的分隔符。只使用 count 个的数组元素，使用的第一个字符串为 startIndex
LastIndexOf(char c)	实例	返回最后一次遇到的指定字符的下标
LastIndexOf(string s)	实例	返回最后一次遇到的指定字符串的下标
LastIndexOf(char c, int startPos)	实例	从指定位置开始，返回最后一次遇到的指定字符的下标

（续表）

方　法	访问级别	描　述
LastIndexOf(string s, int startPos)	实例	从指定位置开始，返回最后一次遇到的指定字符串的下标
LastIndexOf(char c, int startPos, int count)	实例	从指定位置开始并向后搜索 count 个字符，返回最后一次遇到的特定字符的下标
LastIndexOf(string s, int startPos, int count)	实例	从指定位置开始并向后搜索 count 个字符，返回最后一次遇到的指定字符串的下标
LastIndexOfAny(char[] chars)	实例	返回指定数组中最后一次遇到的任意字符的下标
LastIndexOfAny(char[] chars, int startPos)	实例	从指定位置开始，返回指定数组中最后一次遇到的任意字符的下标
LastIndexOfAny(char[] chars, int startPos, int count)	实例	从指定位置开始并向后搜索 count 个字符，返回最后一次遇到的特定数组中任意字符的下标
PadLeft(int totalWidth)	实例	在实例字符串的左端添加所需数目的空格，以使字符串的长度达到 totalWidth
PadLeft(int totalWidth, char c)	实例	在实例字符串的左端添加所需数目的指定字符，以使字符串长度达到 totalWidth
PadRight(int totalWidth)	实例	在实例字符串右端添加所需数目的空格，以使字符串长度达到 totalWidth
PadRight(int totalWidth, char c)	实例	在实例字符串的右端添加所需数目的指定字符，以使字符串长度达到 totalWidth
Remove(int startPos, int count)	实例	从位置 startPos 开始删除 count 个字符
Replace(char oldchar, char newchar)	实例	在实例字符串中，用 newchar 替换所有的 oldchar
Replace(string oldString, string newString)	实例	在实例字符串中，用 newString 替换所有的 oldString
Split(ParamArray Separators)	实例	根据指定的字符串分隔符来划分实例字符串
Split(ParamArray separators, int maxElements)	实例	根据指定的字符分隔符分裂实例字符串，最多返回 maxElements 个字符

(续表)

方 法	访问级别	描 述
StartsWith(string s)	实例	如果实例字符串以指定的字符串开始，返回 true
SubString(int startPos)	实例	从指定的位置开始返回一个子串
SubString(int startPos, int length)	实例	从指定位置开始返回一个指定长度的子串
TocharArray()	实例	返回一个字符数组，字符值由字符串实例提供
TocharArray(int startPos, int length)	实例	返回一个字符数组，其值为字符串实例中从 startPos 开始的 length 个字符的值
ToLower()	实例	返回实例字符串的一个小写版本
ToLower(CultureInfo culture)	实例	根据特定的文化设置返回实例字符串的一个小写版本
ToString()	实例	返回对实例字符串的一个引用
ToString(IFormatProvider Format)	实例	返回对实例字符串的一个引用
ToUpper()	实例	返回实例字符串的一个大写版本
ToUpper(CultureInfo culture)	实例	根据特定的文化设置返回实例字符串的一个大写版本
Trim()	实例	删除字符串实例前导和结尾的空格字符
Trim(ParamArray chars)	实例	删除字符串实例中开始和结尾位置所有的指定字符
TrimEnd(ParamArray chars)	实例	删除字符串实例中结尾位置所有的指定字符。如果没有指定字符，就删除空白字符
TrimStart(ParamArray chars)	实例	删除字符串实例中开始位置所有的指定字符。如果没有指定字符，就删除空白字符

附录B StringBuilder 类

此附录列出了 StringBuilder 类所有的方法、属性和构造函数。在第 2 章中可以找到关于这个类更多的细节。

B.1 构造函数

表 B-1 列出了在创建一个新 StringBuilder 对象时所调用的构造函数。

表 B-1 StringBuilder 的构造函数

构 造 函 数	描 述
StringBuilder()	使用 0 字符的初始长度值创建一个新的 StringBuilder 对象，其默认容量为 16 个字符
StringBuilder(int capacity)	用 0 字符的初始长度值和指定的初始 capacity 创建一个新的 StringBuilder 对象
StringBuilder(string s)	使用 s 的字符串值创建一个新的 StringBuilder 对象
StringBuilder(int capacity, int maxCapacity)	使用 0 字符的初始长度值和指定的初始 capacity 创建一个新的 StringBuilder 对象。容量不能大于已定义的 maxCapacity
StringBuilder(string s, int capacity)	使用 s 的字符串值和指定的初始 capacity 创建一个新的 StringBuilder 对象
StringBuilder(string s, int startPos, int length, int capacity)	使用 startPos 和 length 定义的 s 的指定子字符串值，创建一个新的 StringBuilder 对象

B.2 属性

表 B-2 列出了所有 StringBuilder 实例的属性。

表 B-2　StringBuilder 实例属性

属　性	描　述
Capacity	当前为实例分配的字符数，不需要与 Length 属性值相同
Chars(int index)	取得或设定字符串中指定位置 index 的值
Length	字符串中的字符数。可以设置为小于当前字符串长度的值，为此进行截短操作
MaxCapacity	可以为此实例分配的最大的字符数

B.3　方法

表 B-3 列出了 StringBuilder 实例的所有方法。其中没有静态方法。

表 B-3　StringBuilder 实例的方法

方　法	描　述
Append(<type>)	追加指定值的字符串表示。对于每一个.NET 基本类型都有一个重载方法。为了简单，这里没有列出所有的重载方法
Append(Char value, int count)	追加指定字符 count 次
Append(Char charArray(), int startPos, int count)	追加由指定参数定义的 Char()数组的一个子串
Append(string s, int startPos, int count)	追加指定参数定义的 s 的一个子串
AppendFormat (string specification, Object value)	追加由指定的规范和值定义的一个格式化的字符串
AppendFormat (string specification, ParamArray values())	追加由指定规范和值定义的一个格式化的字符串
AppendFormat (IFormatProvider provider, string specification, Object value)	根据指定的格式化提供者，追加由指定规范和值定义的一个格式化的值

(续表)

方　法	描　述
AppendFormat (string specification, Object value1, Object value2)	追加由指定规范和值定义的一个格式化的字符串
AppendFormat (string specification, Object value1, Object value2, Object value3)	追加由指定规范和值定义的一个格式化的字符串
EnsureCapacity (int capacity)	如果当前容量小于指定容量，为当前容量分配内存，直到和指定容量相同
Equals(StringBuilder sb)	如果当前实例和指定 StringBuilder 具有相同的值，返回 true
GetHashCode()	返回 StringBuilder 实例的一个惟一的散列代码
GetType()	返回 StringBuilder 实例的类型
Insert(int position, <type>)	在指定的位置 position 插入作为第 2 个参数提供的值。对于每一个.NET 基本类型都有一个重载方法。为了简略，这里没有把它们全部列出
Insert(int position, string s, int count)	在指定的位置 position 插入 s count 次
Insert(int position, Char charArray(), int startPos, int count)	在指定 Char 数组的指定位置插入，由指定参数定义的一个子串
Remove(int startPos, int count)	从 startPos 开始删除 count 个字符
Replace(Char oldChar, Char newChar)	在实例字符串中用 newChar 替换所有的 oldChar
Replace(string oldString, string newString)	在实例字符串中用 newString 替换所有的 oldString
Replace(Char oldChar, Char newChar, int startPos, int count)	在实例字符串的 startPos 和 count - 1 之间用 newChar 替换所有的 oldChar

(续表)

方　法	描　述
Replace(string oldString, string newString, int startPos, int count)	在实例字符串的 startPos 和 count -1 之间用 newString 替换所有的 oldString
ToString()	返回对一个 String 对象的引用，它的值与包含在 StringBuilder 实例中的值相同
ToString(int startPos, int count)	返回对一个 String 对象的引用，此对象的值是 StringBuilder 中由指定参数定义的子串

附录C　正则表达式语法

本附录详细地列出了能在正则表达式中使用，以匹配文本的各种字符。当需要解释一个现有的正则表达式时，此附录可以作为一个快捷的参考。

C.1　匹配字符

表 C-1 列出了匹配字符。

表 C-1　匹配字符

字　符　类	匹配的字符	举　例
\d	从 0～9 的任一数字	\d\d 匹配 72，但不匹配 aa 或 7a
\D	任一非数字字符	\D\D\D 匹配 abc，但不匹配 123
\w	任一单词字符，包括 A-Z, a-z, 0-9 和下划线_	\w\w\w\w 匹配 Ab_2，但不匹配£$%* 或 Ab_@
\W	任一非单词字符	\W 匹配@，但不匹配 a
\s	任一空白字符，包括制表符，换行符，回车符，换页符和垂直制表符	匹配在 HTML，XML 和其他标准定义中的所有传统空白字符
\S	任一非空白字符	空白字符以外的任意字符，如 A%&g3;等
.	任一字符	“.”匹配除换行符以外的任一字符(除非设置了 MultiLine 选项)
[...]	括号中的任一字符	[abc]将匹配一个单字符，a, b 或 c。 [a-z]将匹配从 a 到 z 的任一字符
[^...]	不在括号中的任一字符	[^abc]将匹配除了 a、b、c 之外的任一字符，可以匹配 A、B、C。 [^a-z]将匹配不属于范围 a 到 z 的任一字符，但可以匹配所有的大写字母

C.2　重复字符

表 C-2 列出了重复字符。

表 C-2　重复字符

重复字符	含　义	举　例
{n}	匹配前面的字符 n 次	x{2}匹配 xx，但不匹配 x 或 xxx
{n,}	匹配前面的字符至少 n 次	x{2,}匹配 2 个或更多的 x，如 xx, xxx, xxxx, xxxxx,...
{n,m}	匹配前面的字符至少 n 次，至多 m 次。如果 n 为 0，此参数为可选参数	x{2,4}匹配 xx, xxx 和 xxxx,但不匹配 x 或 xxxxx
?	匹配前面的字符 0 次或 1 次，实质上也是可选的	x?匹配 x 或零个 x
+	匹配前面的字符一次或多次	x+匹配 x 或 xx 或大于 0 的任意多的 x
*	匹配前面的字符 0 次或更多次	x*匹配 0 个，1 个或更多个 x

C.3　定位字符

表 C-3 列出了各种定位字符。它们可以应用于字符或组合，放在字符串的左端或右端。

表 C-3　定位字符

定位字符	描　述
^	随后的模式必须位于字符串的开始位置，如果是一个多行字符串，则必须位于行首。对于多行文本(包含回车符的一个字符串)来说，需要设置多行标志
$	前面的模式必须位于字符串的末端，如果是一个多行字符串，必须位于行尾
\A	前面的模式必须位于字符串开始位置，忽略多行标志
\z	前面的模式必须位于字符串的末端，忽略多行标志
\Z	前面的模式必须位于字符串的末端，或者位于一个换行符前
\b	匹配一个单词边界，也就是一个单词字符和非单词字符中间的点。要记住一个单词字符是[a-zA-Z0-9_]中的一个字符。位于一个单词的词首
\B	匹配一个非单词字符边界位置，不是一个单词的词首

C.4 分组字符

表 C-4 列出了各种分组字符，利用它们可以把指定的匹配进行分组，还可以检索这些指定的匹配。

表 C-4 分组字符

分组字符	定义	举例
()	此字符可以组合括号内模式所匹配的字符。它是一个捕获组，也就是说模式匹配的字符作为最终匹配的一部分，除非设置了 ExplicitCapture 选项——默认状态下字符不是匹配的一部分	输入字符串为：ABC1 DEF2 XY 匹配 3 个从 A 到 Z 的字符和 1 个数字的正则表达式： ([A-Z]{3})\d 将产生两次匹配： Match 1 = ABC1 Match 2 = DEF2 每次匹配对应一个组： Match1 的第 1 个组＝ABC Match2 的第 1 个组＝DEF 有了反向引用，就可以通过它在正则表达式中的编号以及 C#和类 Group、GroupCollection 来访问组。如果设置了 ExplicitCapture 选项，就不能使用组所捕获的内容
(?:)	此字符组合括号内模式所匹配的字符。它是一个非捕获分组，这意味着模式所匹配的字符将不作为一个组来捕获，但它构成了最终匹配结果的一部分。它基本上与上面的组类型相同，但设定了选项 ExplicitCapture	输入字符串为：1A BB SA 1 C 匹配一个数字或一个 A 到 Z 的字母，接着是任意单词字符的正则表达式为： (?:\d|[A-Z])\w 它将产生 3 次匹配： 第 1 次匹配＝1A 第 2 次匹配＝BB 第 3 次匹配＝SA 但是没有组被捕获

（续表）

分组字符	定义	举例
(?<name>)	此选项组合括号内模式所匹配的字符，并用尖括号中指定的值为组命名。在正则表达式中，可以使用名称进行反向引用，而不必使用编号。即使不设置 ExplicitCapture 选项，它也是一个捕获组。这意味着反向引用可以利用组内匹配的字符，或者通过 Group 类访问	输入字符串为： Characters in Sienfeld included Jerry Seinfeld, Elaine Benes, Cosmo Kramer and George Costanza 能够匹配它们的姓名，并在一个组 lastName 中捕获姓的正则表达式为： \b[A-Z][a-z]+ (?<lastName>[A-Z][a-z]+)\b 它产生了 4 次匹配： First Match = Jerry Seinfeld Second Match = Elaine Benes Third Match = Cosmo Kramer Fourth Match = George Costanza 每一次匹配都对应了一个 lastName 组： 第 1 次匹配: lastName group = Seinfeld 第 2 次匹配: lastName group = Benes 第 3 次匹配: lastName group = Kramer 第 4 次匹配: lastName group = Costanza 不管是否设置了选项 ExplictCapture，组都将被捕获
(?=)	正声明。声明的右侧必须是括号中指定的模式。此模式不构成最终匹配的一部分	正则表达式\S+(?=.NET)要匹配的输入字符串为： The languages were Java, C#.NET, VB.NET, C, JScript.NET, Pascal 将产生如下匹配： C# VB JScript
(?!)	负声明。它规定模式不能紧临着声明的右侧。此模式不构成最终匹配的一部分	\d{3}(?![A-Z])要匹配的输入字符串为： 123A 456 789 111C 将产生如下匹配： 456 789

(续表)

分组字符	定　义	举　例
(?<=)	反向正声明。声明的左侧必须为括号内的指定模式。此模式不构成最终匹配的一部分	正则表达式(?<=New)([A-Z][a-z]+)要匹配的输入字符串为： The following states, New Mexico, West Virginia, Washington, New England 它将产生如下匹配： Mexico England
(?<!)	反向正声明。声明左侧不能是括号内的指定模式。模式将不构成最终匹配的一部分	正则表达式(?<!1)\d{2}[A-Z]要匹配的输入字符串如下： 123A 456F 789C 111A 它将实现如下匹配： 56F 89C
(?>)	非回溯组。防止Regex引擎回溯并且防止实现一次匹配	假设要匹配所有以“ing”结尾的单词。输入字符串如下： He was very trusting 正则表达式为： .*ing 它将实现一次匹配——单词trusting。“.”匹配任意字符，当然也匹配“ing”。所以，Regex引擎回溯一位并在第2个“t”停止，然后匹配指定的模式“ing”。 但是，如果禁用回溯操作： (?>.*)ing 将实现0次匹配。“.”能匹配所有的字符，包括“ing”——这意味着模式末端的“ing”不能匹配，从而匹配失败

C.5 决策字符

表C-5列出的字符强迫处理器执行一次if...else决策。

表 C-5　决策字符

字　符	描　述	举　例
(?(regex)yes_regex|no_regex)	如果表达式 regex 匹配，那么将试图匹配表达式 yes。否则匹配表达式 no。正则表达式 no 是可选参数。注意，作出决策的模式宽度为 0。这意味着表达式 yes 或 no 将从与 regex 表达式相同的位置开始匹配	正则表达式(?(\d)\dA|[A-Z]B)要匹配的输入字符串为： “1A CB 3A 5C 3B” 它实现的匹配是： 1A CB 3A
(?(group name or number)yes_regex|no_regex)	如果组中的正则表达式实现了匹配，那么试图匹配 yes 正则表达式。否则，试图匹配正则表达式。no 正则表达式 no 是可选的参数	正则表达式(\d7)?-(?(1)\d\d[A-Z]|[A-Z][A-Z]) 要匹配的输入字符串为： 77-77A 69-AA 57-B 它实现的匹配为： 77-77A -AA

C.6　替换字符

表 C-6 列出了可以作为替换字符串的一部分输入的字符。

表 C-6　替换字符

字　符	描　述
$group	用 group 指定的组号替换
${name}	替换被一个(?<name>)组匹配的最后子串
$$	替换一个字符$
$&	替换整个的匹配
$`	替换输入字符串匹配之前的所有文本
$'	替换输入字符串匹配之后的所有文本
$+	替换最后捕获的组
$_	替换整个的输入字符串

C.7　转义序列

表 C-7 列出了正则表达式中可能需要的转义字符。

表 C-7　转义字符

转义字符	描　述
\\	匹配实际的字符“\”
\.	匹配字符“.”
*	匹配字符“*”
\+	匹配字符“+”
\?	匹配字符“?”
|	匹配字符“|”
\(	匹配字符“(”
\)	匹配字符“)”
\{	匹配字符“{”
\}	匹配字符“}”
\^	匹配字符“^”
\$	匹配字符“$”
\n	匹配换行符
\r	匹配回车符
\t	匹配制表符
\v	匹配垂直制表符
\f	匹配换页符
\nnn	匹配一个 8 进制数字 nnn 指定的 ASCII 字符。所以\103 匹配一个大写的 C
\xnn	匹配一个 16 进制数字 nn 指定的 ASCII 字符，所以\x43 匹配 C
\unnnn	匹配由 4 位 16 进制数字(由 nnnn 表示)指定的 Unicode 字符
\cV	匹配一个控制字符，例如\cV 匹配 Ctrl-V

C.8　选项标志

表 C-8 列出了可以在模式中设置的正则表达式选项。

表 C-8　选项标志

选项标志	名　称
I	IgnoreCase
M	Multiline
N	ExplicitCapture
S	Singleline
X	IgnorePatternWhitespace

选项本身的含义如表 C-9 所示。

表 C-9　可在模式中设置的正则表达式选项

标　志	描　述
IgnoreCase	使模式匹配不区分大小写。默认的选项是匹配区分大小写
RightToLeft	从右到左搜索输入字符串。默认是从左到右以符合英语等语言的阅读习惯，但不符合阿拉伯语或希伯来语的阅读习惯
None	不设置标志。这是默认选项
MultiLine	指定^ 和 $可以匹配行首和行尾，以及字符串的开始和结尾。这意味着可以匹配每个用换行符分隔的行。但是，字符“.”仍然不匹配换行符
SingleLine	规定特殊字符“.”匹配任意的字符，包括换行符。默认情况下，特殊字符“.”不匹配换行符。通常与 MultiLine 选项一起使用
ECMAScript	ECMA(European Computer Manufacturer's Association，欧洲计算机生产商协会)已经定义了正则表达式应该如何实现，而且已经在 ECMAScript 规范中实现，这是一个基于标准的 JavaScript。这个选项只能与 IgnoreCase 和 MultiLine 标志一起使用。与其他任何标志一起使用，ECMAScript 都将产生异常
IgnorePatternWhiteSpace	此选项从使用的正则表达式模式中删除所有非转义空白字符。它使表达式能跨越多行文本，但必须确保对模式中所有的空白字符进行了转义。如果设置了此选项，还可以使用"#"字符来注释正则表达式
Compiled	它把正则表达式编译为更接近机器代码的代码。这样速度更快，但不允许对它进行任何修改

附录D　在C#中管理文化上正确的日期和时间格式

本书讨论了如何使用 C#和.NET Framework 进行文本处理。本附录则给出了一篇发表在 C Sharp Today 网站(http://www.csharp.today.com)上的文章，这篇文章最初在 2001 年 11 月发表，已经对.NET 1.0 进行了全面更新，您可以在网站上找到这篇文章。

这篇文章以 C#为编程语言，全面阐述了.NET DateTimeFormatInfo 类的各种功能。其目的是帮助程序员编写出在世界任何地方都能运行正确的代码。明明可以一次就能写好程序，为什么要编写两次？因为我们先编写一些简单的、立即就能用的程序，这是使用两个控制台程序完成的。之后，再介绍两个例子，用 ASP.NET 对日期和日历进行几种处理。接着，讨论 DateSquares 示例，该示例最初出现在 Wrox 图书《C# Programming With the Public Beta》 (ISBN 1-86100-487-7)的第 13 章开头，说明如何进行简单的修改，使输出结果在文化上是正确的。

D.1　使用 DateTime 的控制台示例

控制台示例提供了学习如何处理某些新概念或编程任务的快捷方式。最初，DateTimeFormatInfo 如何利用 DateTime 结构并不明显，DateTime 结构在.NET 中是时间和日期的基本容器。DateTime 的实例存储了时间的一个特定时刻。使用 DateTime 的方法和属性，可以给该实例增加或减少一个时间段，改变该时刻。还可以分析、比较、转换和格式化其内容。DateTime 继承于 IFormattable 接口，DateTimeFormatInfo 也利用 IFormattable 接口来发挥作用。DateTime 值使用标准或自定义的模式进行格式化，这些模式存储在 DateTimeFormatInfo 实例的属性中。

DateTime 从其继承的 DateTimeFormatInfo 中获得了许多内置的格式化方法。所以，即使不直接调用 DateTimeFormatInfo 类，也可以管理国际化的日期格式。输出格式由线程的隐含 CultureInfo 决定，如果需要，还可以覆盖该隐含 CultureInfo。在任何现代操作系统上运行的每个线程都有一个关联的默认地区，如果需要，可以覆盖该地区，在.NET 中，地区由 CultureInfo 类管理。

如果您不熟悉 CultureInfo 类的用法，可以阅读文章 Internationalization with the .NET CultureInfo Class & C# (http://www.csharptoday.com/content.asp?id=1635)。CultureInfo 会

影响 System.Globalization 命名空间中大多数成员的行为，对此类有基本的了解，将有助于理解本文章。

D.1.1　第一个控制台示例

在第一个示例中，使用 DateTime 的功能提供国际化格式。主要介绍如何使用最常用的格式，即短日期和时间格式，以及长日期和时间格式。本附录还提供了代码，以说明如何生成运行该示例所用的文化中 DateTime 的所有格式，但需要使用 DateTimeFormatInfo 确定这么多信息的用法。我们还将演示如何修改默认的文化设置，以便在其他地区查看结果。

下面创建一个 C#控制台工程 ConsoleDateTime。在类文件中，添加如下代码：

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

namespace ConsoleDateTime
{
  class Class1
  {
    static void Main(string[] args)
    {
      // get the current time
      DateTime dt = DateTime.Now;

      // display the name
      Console.WriteLine("Current Thread Culture : {0}",
          Thread.CurrentThread.CurrentCulture.Name);

      // show the output from ToString() (not internationalized)
      Console.WriteLine("DateTime.ToString: {0}", dt.ToString());

      // show culturally correct short and long dates and times
      Console.WriteLine("DateTime.ToshortDateString: {0}",
          dt.ToShortDateString());
      Console.WriteLine("DateTime.ToLongDateString: {0}",
          dt.ToLongDateString());
      Console.WriteLine("DateTime.ToShortTimeString: {0}",
```

```
            dt.ToShortTimeString());
        Console.WriteLine("DateTime.ToLongTimeString: {0}",
            dt.ToLongTimeString());

        // enumerate all possible formats
        // not all of these are culturally useful
        string [] dtfmts = dt.GetDateTimeFormats();
        for (int i = 0; i < dtfmts.GetUpperBound(0); i++)
            Console.WriteLine("{0}", dtfmts[i]);
        foreach(string s in dtfmts)
            Console.WriteLine("{0}", s);
    }
  }
}
```

运行结果如图 D-1 所示。

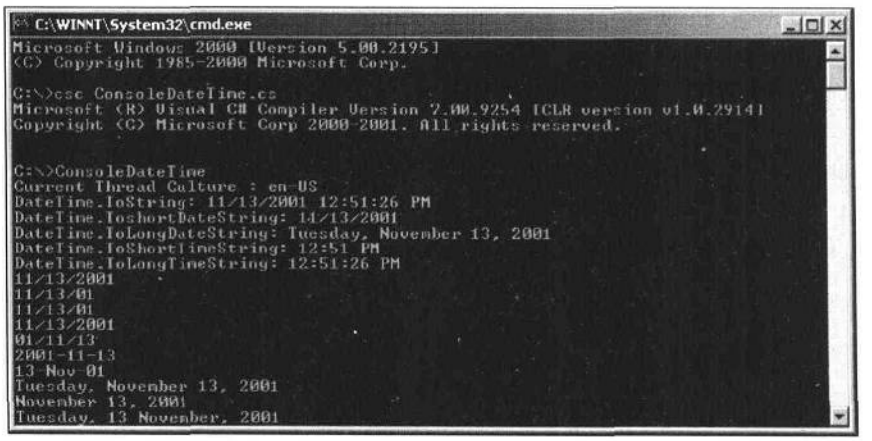


图　D-1

这个程序的运行结果是可以扩展的。该结果取决于启动线程的文化设置，该设置在结果的第一行输出为 RFC 1766 标识符。在 http://www.ietf.org/rfc/rfc1766.txt?number=1766 可以查阅有关 RFC 1766 样式标识符的更多内容。在第一次运行完程序之后，从控制面板的区域选项中选择一个新区域，修改默认的文化设置，如图 D-2 所示。

选择一个靠近默认地区的地区(因为使用控制台输出结果，如果选择的地区完全不同，就只能看到问号，而看不到文本)。例如，在我的系统上，通常用 English (United States) 地区设置进行开发。如果选择 French (France)，它的控制台代码页与 English (United States)类似，程序运行时就可以获得有效的输出，而不必添加任何额外的语言支持，如图 D-3 所示。

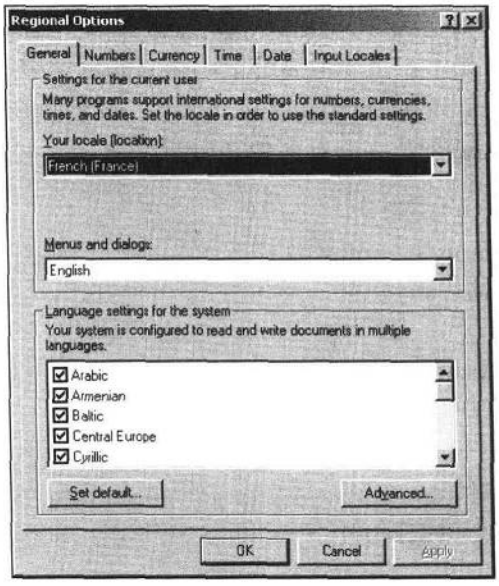


图　D-2

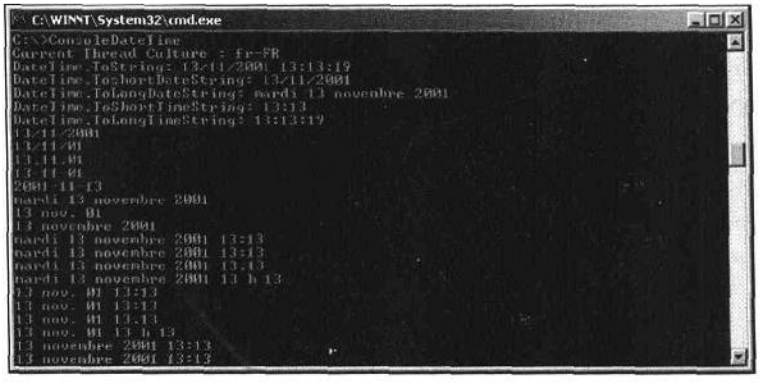


图　D-3

这里进行的地区设置确定了启动线程的 CultureInfo 实例，这就是获得不同输出的原因。

后面显示的许多格式(但不是全部)都可以正常使用。其中一些有特殊的用途，需要更深入地讨论 DateTimeFormatInfo。另外，在一些应用程序中，必须修改所使用的日历类型，才能获得特定的格式。DateTimeFormatInfo 可以把它们全部挑选出来，但在大多数情况下，DateTime 的默认短日期字符串和长日期字符串方法足以在当前线程中返回有效的默认结果。在以后的文章中将深入论述 DateTimeFormatInfo。

下面的一个示例遍历在当前地区可用的日历，并在一个消息框中显示长日期(许多地区使用不同的日历来标记重要的国家事件或节日，这里将涉及其中的一些)。在消息框中提供输出时，可以更好地查看结果，特别是在查看“外来的”地区时，就更是如

此。但是，必须安装额外的语言支持(详见下面的内容)。下面还提供了代码。我们使用一个控制台程序，但输出显示在一个消息框中。注意显式使用 DateTimeFormatInfo 来获得相应的长日期模式，并用作 DateTime 格式化器。我们还在工程中添加了一行代码 'using System.Windows.Forms; '和一个对应的.dll 引用。

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

namespace ConsoleDateTimeCalendars
{
  class Class1
  {
    static void Main(string[] args)
    {
      // Get the current date
      DateTime dt = DateTime.Now;

      // Create a writable cultureinfo according to the thread locale
      CultureInfo ci = CultureInfo.CreateSpecificCulture(
          CultureInfo.CurrentCulture.Name);

      // show the name
      Console.WriteLine("Current Culture: {0}", ci.DisplayName);

      // get the associated DateTimeFormatInfo instance
      DateTimeFormatInfo dtf = ci.DateTimeFormat;

      // enumerate the calendars
      Calendar [] cal = ci.OptionalCalendars;
      for (int i = 0; i < cal.GetLength(0); i++) {

        // select in each calendar
        dtf.Calendar = cal[i];

        // show the result
        MessageBox.Show(dt.ToString(dtf.LongDatePattern, dtf),
            cal[i].ToString());
      }
```

```
            foreach(Calendar c in cal) {
                // select in each calendar
                dtf.Calendar = c;
                // show the result
                MessageBox.Show(dt.ToString(dtf.LongDatePattern, dtf),
                    c.ToString());
            }
        }
    }
}
```

运行代码，会返回一系列消息框，其标题是日历的名称，消息则是一个长日期格式例如，如果在机器上把地区设置为日本(使用 Regional Options 对话框)，其输出结果如图 D-4 所示，这个地区有 3 个日历。第一个日历是 Japanese Era 日历，输出为 Heisei 13 年 9 月 22 日，与 Gregorian 日历相比，它只有年份不同，日期和月份相同。如果选择日本，就必须支持这个日历。

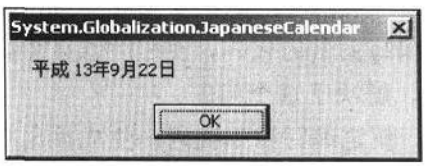


图　D-4

第二个结果是使用英语的 Gregorian 日历，如图 D-5 所示。

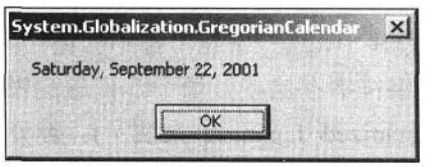


图　D-5

最后一个结果是使用日语的 Gregorian 日历，如图 D-6 所示。输出为 2001 年 9 月 22 日(刚才我们介绍了 3 个有用的 kanji(Chinese)字符)。

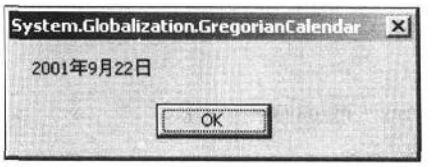


图　D-6

如果设置阿拉伯地区，还会看到更多的日历。其中的大多数都有一个 Hijri 阴历和

几个 Gregorian 日历，分别使用英语、法语、法语和/或英语的音译。在国际化时会产生令人惊讶的结果的其他地区有中国(台湾)、泰国、韩国、波斯和以色列。以色列也有 13 个月。下一篇文章将介绍如何查找这些信息。

如果不能清晰地查看这些结果，或在系统上找不到这些地区，就应在系统上安装这些语言的支持软件。这是非常容易的。使用 Regional Options 对话框，选择要使用的语言(参阅上面的 Regional Options 对话框图示)，并使用手边的安装 CD。

D.1.2 管理字符串格式的注意事项

在上面的代码示例中，注意隐含使用了 String 类来管理像这个例子这样的格式化。

```
Console.WriteLine("Current Culture: {0}", ci.Name);
```

程序员会犯的一个最大错误是使用字符串连接来建立消息。把多个片段组装为一个句子会很难翻译，因为结果完全依赖于单词的顺序和自然语言的基本语法规则。但像 VB、C#和 Java 这样的编程语言很容易出错，而当程序或网站必须翻译为几种语言时，其代价是很高的，必须重新编写大量的代码，使程序独立于底层的用户接口。而 C#(和 Java)提供了一个很好的机制，解决了这个问题。

下面看一个简单的例子。假定编写如下代码，生成输出"My name is Bill"：

```
string myname = "Bill";
string greeting = "My name is ";
Console.WriteLine(greeting + myname);
```

在一个编写得非常好的程序中，这些字符串都不会在代码中出现，而是放在一个消息表或资源文件中。但即使字符串存储在资源中，问题也没有解决。下面把希望得到的输出翻译为日语(使用 Romanized Japanese 来演示)。翻译的一个结果是"Watashi no imeno wa Bill desu"，这类似于"As for my name it Bill is"。但是，如果试图通过连接来建立消息，就必须重新编写代码，把 3 个部分连接在一起，而不是连接两个部分，因为变量 myname 的值必须放在动词的前面：

```
string myname = "Bill";
string greeting = "Watashi no imeno wa ";
string tobe = " desu"
Console.WriteLine(greeting + myname + tobe);
```

这样的词序在 Japanese 中很常见，German 在一定程度上也使用这样的词序。许多其他语言也使用这样的词序，但英语不是。也就是说，必须修改代码来显示消息，这是很糟糕的解决方法，而且也是不必要的。

该如何解决这个问题？其实很简单，把问候字符串改为：

```
string greeting = "Hello, my name is {0}";
```

翻译为日语，就是"Watashi no imeno wa {0} desu"。现在使用相同的代码可以处理最初的字符串和翻译后的字符串，即

```
Console.WriteLine(greeting, myname);
```

在不同的语言中使用多种替代时，这种技术非常有用。例如，在 C 和 C++中使用 printf 系列，在 Win32 中使用 FormatMessage，在 MFC 中使用 CString::FormatMessage，在 Java 中使用 MessageFormat 类。

D.2　ASP 的一些示例

如果您对 C#感兴趣，可能阅读过上面提到的 Wrox 图书 C# Programming with the Public Beta。

在第 13 章的前面，给出了创建 Web 工程的步骤。之后，把默认的.aspx 文件重新命名为 DateSquares.aspx，并添加一些 C#代码，以显示日期，然后为 1～10 这些数字创建一个方块列表。下面就是添加的代码：

```
<h1>First ASP .NET Page</h1>
Today's date is
<% = DateTime.Now.ToString() %>

<p>
First 10 squares are
<%
int i;
for (i = 1; i <= 10; i++) {
  Response.Write("<br> ", + i.ToString() + " squared = " +
  (i*i).ToString());
}
%>
```

这里有许多有趣的国际化问题(包括通过字符串连接功能把 HTML 和程序输出组合起来，保证网站不能翻译为另一种语言！)，第一个引起我们注意的是显示日期的代码，如上面突出显示的代码所示。在运行代码时，会得到所期望的结果：显示文化上正确的日期。但是，从一开始就正确地进行通常比较容易，而且节省了产品的投资和开发时间，特别适合于为其他地区发布的产品分期清偿投资。

接着看看上述日期代码的错误，说明如何用较少的编程工作来修复该错误。问题在于输出。无论使用什么系统，在系统上进行什么用户地区设置，上述代码行的输出总是US样式的日期和时间字符串，即9/10/2001 7:04:35 PM。字段之间的分隔符是'/'和':'，时间按12小时计算，更糟糕的是，日期是模糊的，它是9月10日，还是10月9日？如果用户生活在US，就知道应是9月10日，但欧洲大陆的大多数人会认为是10月9日。如果使用两个数字的年份，这可能更让人糊涂。例如，9/10/01甚至可以看作是Heisei 9年10月1日。这似乎很有趣，但如果程序是为世界上最重要的经济实体之一日本的用户开发的，就必须处理这个问题。最后，程序的输出显示了太多的信息。通常用户只需要日期或时间，而很少同时需要两者。下面用两个示例来说明如何处理这些问题。

D.2.1 修正 DateSquares

在第一个示例中，重新编写 DateSquares.aspx。我们把日期部分用一行代码来代替，使输出与地区有一定的关联。再把文本"Today's date is "移动到一个 String.Format 调用中，使之更容易维护和翻译，之后在 ToString 语句中使用一个格式化器。这里间接使用了 DateTimeFormatInfo，因为 CultureInfo 是它的一个成员。最后，把方块表用服务器上所有地区的列表来代替，并显示地区名称、国家名称及完整的日期和时间。在混合非常多的语言时，应把页面编码方式(文档属性是'charset')设置为 utf-8，如图 D-7 所示。

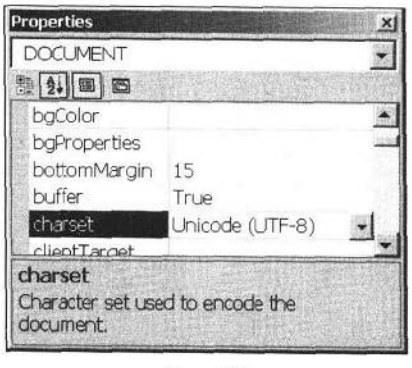


图 D-7

下面是代码。注意需要在头文件中使用 import 语句，以访问 Globalization 命名空间和 charset 声明。还要注意如何使用 String 类解决最初代码的字符串连接问题，并隔离 HTML 语句。下面是完整的页面：

```
<%@ Page language="c#" Codebehind="datesquares.aspx.cs"
    AutoEventWireup="false" Inherits="AspIntlDemo.datesquares" %>
<%@ Import Namespace="System.Globalization" %>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>

<HEAD>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
  <meta name="CODE_LANGUAGE" Content="C#">
  <meta name=vs_defaultClientScript
        content="JavaScript (ECMAScript)">
  <meta name=vs_targetSchema
        content="http://schemas.microsoft.com/intellisense/ie5">
</HEAD>

<body>
  <h1> First.ASP.Net Page </h1>
  <% = String.Format("Today's date is {0}",
       DateTime.Now.ToString(CultureInfo.CurrentCulture)) %>

  <p>
  <%
    DateTime dt = DateTime.Now;
    CultureInfo [] ci =
        CultureInfo.GetCultures(CultureTypes.SpecificCultures);
    for (int i = 0; i < ci.GetLength(0); i++) {
      DateTimeFormatInfo dtf = ci[i].DateTimeFormat;
      Response.Write(String.Format("<br>{0} {1} {2}",
                     ci[i].EnglishName,
                     ci[i].NativeName,
                     dt.ToString(dtf.FullDateTimePattern, dtf)));
    }
  %>
</body>
</HTML>
```

为什么要列举特定的文化，而不是所有的文化？特定的文化包含了语言和地区，而父文化只包含语言。但父文化与 DateTimeFormatInfo 类没有关联。这么做的一个原因是 DateTimeFormatInfo 包含工作日和月份的名称，还包含大量与语言和地区相关的格式。例如，en-US(United States) 和 en-UK (United Kingdom)的父文化相同，都是 en，但第一个工作日是不同的，分别是 Sunday 和 Monday。默认的短日期格式也有不同的

顺序(如 15/8/2001 和 08/15/2001)。

程序的输出是可以扩展的。本文包含了实际生成的页面(datesquares_aspx.htm)，使用浏览器可以打开它，查看输出结果。根据本地系统上的字体和国际化支持设置，该输出结果可能有所不同，如图 D-8 所示。

图　D-8

最后，看看获得文化上正确显示的日期所需要的步骤，而避免字符串连接所带来的缺陷并不需要非常多的代码。这个例子的惟一问题是输出显示了服务器可以完成的工作，但没有考虑用户的喜好。所以下面介绍另一个示例，它读取用户的语言设置，并据此设置一些日期、时间和日历信息。

D.2.2　另一个日期示例

在这个示例中，要在浏览器中读取用户的语言设置，找到这些语言设置后，为它们每一个创建一个特定的文化。对于每个文化，枚举所有的日历，为每个日历，显示完整的日期和时间字符串。本例还以两种格式显示文化的名称(原来的名称和 UI 显示)以及日历名称。

您可能知道如何在浏览器中设置语言。在 Internet Explorer 中，选择 Tools | Internet Options | Languages | Language Preference。在下面的屏幕图 D-9 中，我在浏览器中设置的语言有相关的 RFC 1766 标识符(en-US、ar-EG 和 ru 等)，这些标识符可以读取，并用

作 CultureInfo 示例的构造函数，或传递给静态方法 CreateSpecificCulture。例如，在 CultureInfo 构造函数中使用 ru，就会得到一个父文化，它不能用于日期和时间信息。但是，CreateSpecificCulture 会把 Russia 作为默认国家，创建一个可写的 Russian 语言 CultureInfo 对象，这正是我们想要的结果。

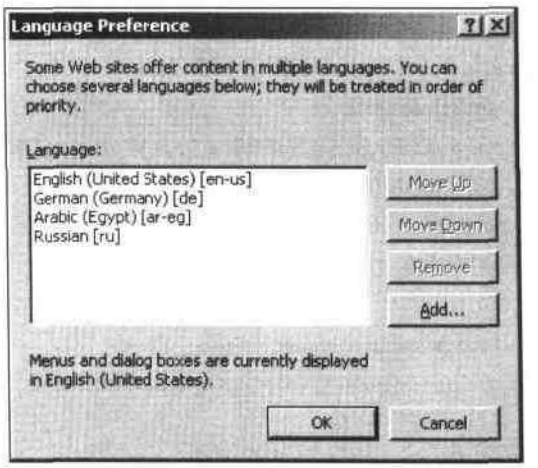


图　D-9

下面是生成结果的源代码。注意使用 String 类隔离 HTML 格式，并分析 RF C 1766 标识符。为了简洁起见，删除了样板文件的大多数内容和 HTML 头语句，只留下了重要的 Import Namespace 指令：

```
<%@ Page language="c#" %>
<%@ Import Namespace="System.Globalization" %>
<body >
<%
  // get current date and time
  DateTime dt = DateTime.Now;

  // retrieve the server variables
  NameValueCollection coll = Request.ServerVariables;

  // get the accept_language string
  string s = coll.Get("HTTP_ACCEPT_LANGUAGE");

  // parse the strings and get the RFC 1766 ID's
  // which are comma separated.
  char [] sep = {','};
```

```
// split the strings into an array and iterate over each element
string [] langstr = s.Split(sep);
for (int i = 0; i < langstr.GetLength(0); i++) {
   // the substrings may have additional data separated by ';'
   int pos = langstr[i].IndexOf(";");

   // use the 'name' variable obtained (RFC 1766 Id)
   // to create a CultureInfo instance
   string name = pos > 0 ? langstr[i].Substring(0, pos) : langstr[i];
   CultureInfo ci = CultureInfo.CreateSpecificCulture(name);
   // get the associated DateTimeFormatInfo object
   DateTimeFormatInfo dtf = ci.DateTimeFormat;

   // enumerate all the possible calendars in this culture
   System.Globalization.Calendar [] cal = ci.OptionalCalendars;

   // Show the user's languages in local and native formats
   Response.Write(String.Format("<h2>{0} : {1}</h2>",
        ci.DisplayName, ci.NativeName));

   // For each calendar, display some information
   for (int j = 0; j < cal.GetLength(0); j++) {
      // select the calendar into the DateTimeFormatInfo object
      dtf.Calendar = cal[j];

      // display the name of the calendar in bold type
      Response.Write(String.Format("<STRONG>{0}</STRONG>",
           cal[j].ToString()));

      // if the calendar is Gregorian append the type in parentheses
      if (cal[j].GetType().IsInstanceOfType(new GregorianCalendar()))
           Response.Write(String.Format("<STRONG> ({0})</STRONG><br>",
           ((GregorianCalendar)cal[j]).CalendarType.ToString()));

      else
         Response.Write("<br>");
         // Finally show the fully localized date for this calendar
         Response.Write(String.Format("{0}<br>",
              dt.ToString(dtf.FullDateTimePattern, dtf)));
   }
```

```
    }
%>
</body>
```

下面是查找 Russian 和 Egyptian–Arabic 文化的输出，分别如图 D-10 和 D-11 所示。

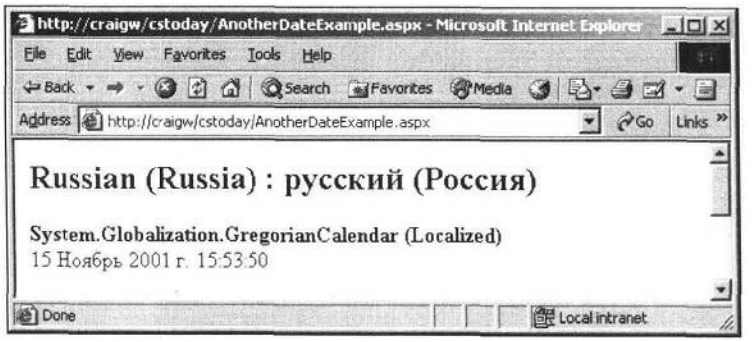


图　D-10

图　D-11

D.3　小结

提供自动正确显示的日期和时间格式是很简单的，因为编写代码不涉及到文化。在大多数情况下，DateTime 中的短和长日期格式化器足以用文化上正确的方式显示日期和时间。如果需要显示更详细的信息，必须显式使用 DateTimeFormatInfo 类，这个类

也是许多附带信息的一个库，其中包含工作日的名称、月份名称以及缩写的名称，一周的第一天、日历和许多特定格式，包括可以分析的格式。

附录E　技术支持、勘误表和代码下载

我们总是想知道您对本书的看法，您喜欢哪些内容，不喜欢哪些内容，这些信息都将有助于我们下一次做得更好。如果您有什么意见和建议，请向 feedback@wrox.com 发邮件。但是，您一定要在您的信中注明本书的书名。

这是我们的第一本 C#参考手册，它几乎是从《VB.NET 字符串和正则表达式参考手册》直接翻译过来的。希望您让我们知道如何才能使我们做得更好。

E.1　如何下载本书的示例代码

在您登录到 Wrox.com 站点 http://www.wrox.com/时，只需使用 Search 工具或使用书名列表就可以找到本书。接着单击本书的信息页面上的 Download Code 链接，就可以获得所有的源代码。

从该站点上下载的文件已经使用 WinZip 进行了压缩。在把文件保存到硬盘的一个文件夹中时，需要使用解压缩程序如 WinZip 对该文件解压缩。在 ZIP 文件中有一个文件夹结构和一个解释该结构的 HTML 文件，并给出了其他信息，包括 E-mail 支持和建议您深入阅读的材料。

E.2　勘误表

尽管我们已经尽了各种努力来保证本书正文或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他您避免受挫，当然，这还有助于提供更高质量的信息。请给 support@wrox.com 发电子邮件，我们就会检验您的信息，如果是正确的，就把它传送到该书的勘误表页面上。

要在网站上找到勘误表，可以登录 Wrox 网站 http://www.wrox.com/ACON1.asp?ISBN=1861008236，然后在本书的信息页面上，单击 Book Errata 链接。

E.3　电子邮件支持

如果您希望直接就本书的问题向对本书知之甚多的专家咨询，可以向

support@wrox. com 发电子邮件，在电子邮件的“主题”(Subject)栏内，加上本书的名称和 ISBN 的最后 4 位号码。典型的电子邮件应该包括下列内容：

- 在“主题”栏加上书的名称、ISBN 的最后 4 位数字(8236)和问题所在的页码号。
- 在邮件的正文中加上您的姓名、联系信息和问题。

我们不会发给您垃圾邮件。我们只需要详细的情况以节省您和我们的宝贵时间。当您发送电子邮件时，它会直接到达以下支持链：

- 客户支持——您的消息会传送到我们的客户支持人员，他们有常见问题的文件，会迅速回答关于本书和网站的一般性问题。
- 编辑支持——更深的问题会转发到负责本书的技术编辑处。他或她具有编程语言或特殊产品的经验，能够回答某个主题的详细技术问题。解决了问题后，技术编辑会把它传送到该书的勘误表页面上。
- 作者支持——最后，在编辑都不能回答问题的情况下(这种情况很少出现)，这些问题将转发到作者。我们试图保护作者不要从写作(或编程工作)中分心，但是，我们也很愿意将特殊的问题转发给他们。所有的 Wrox 作者都会帮助支持他们的书籍。他们对客户和编辑回复电子邮件，所有的您都会从中受益。

Wrox 支持过程只能提供直接与已出版的图书相关的问题。对于超出此范围的问题可以通过 http://p2p.wrox.com/ 论坛的团体列表来提供支持。

E.4 p2p.wrox.com

P2P 邮件列表是为作者和同行的讨论而设立的。我们在邮件列表、论坛和新闻组中提供“程序员到程序员的支持”(programmer to programmer support)，还包括一对一的电子邮件支持系统。如果把问题发送给 P2P，就可以相信，您的问题不仅仅是由支持专家解答，而且还要提供给我们邮件列表中的许多 Wrox 作者和其他业界专家。在 p2p.wrox.com 上，可以从许多不同的列表中获得帮助，不仅在阅读本书时获得帮助，还可以在开发应用程序时获得帮助。

要订阅一个邮件列表，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com。

(2) 从左侧的菜单栏中选择合适的列表。

(3) 单击想加入的邮件列表。

(4) 按照指示订阅和填写电子邮件地址和密码。

(5) 回复接收到的确认电子邮件。

(6) 使用订阅管理器加入更多的列表，设置自己的邮件设置。