第5章

歌词同步播放器

5.1 教学目标

5.1.1 知识目标

- 了解使用 Windows Media Player 控件进行音频播放和控制。
- 掌握 ProgressBar 控件的使用，能够实现播放进度的显示。
- 掌握字符串处理函数，如 Substring、Split、IndexOf、LastIndexOf、ToCharArray 等，能够灵活运用这些函数进行文本解析。
- 掌握泛型集合 List < T >的使用。
- 理解歌词同步显示的原理。
- 掌握 Timer 组件的使用，能够实现定时任务和同步功能。
- 理解对话框的使用，能够实现文件选择和用户交互。

5.1.2 能力目标

- 具备设计并实现一个简单的音频播放器应用程序的能力。
- 掌握使用 Windows Media Player 控件实现音频播放、暂停、停止等基本功能。
- 理解并能够实现播放进度的显示和同步。
- 具备解析 LRC 歌词文件的能力，能够实现歌词与音频的同步显示。
- 理解并能够实现音频与歌词文件的自动匹配和加载。

5.1.3 素质目标

- 培养读者的创新思维，鼓励在现有基础上进行功能扩展和优化。
- 培养读者的动手能力，通过实践项目巩固所学知识。
- 培养读者的用户体验意识，通过界面设计和功能实现提升软件的用户友好度。
- 培养读者的逻辑思维能力，通过编程实现复杂的媒体处理和同步功能。

5.2 案例简介

本案例旨在开发一个简单的歌词同步播放器应用程序，通过 Windows Forms 实现。该应用程序的主要功能如下。

1. 音频播放

支持常见音频格式的播放、暂停、停止等基本控制功能。

2. 进度显示

通过 ProgressBar 控件实现播放进度的可视化，同时显示总时长。

3. 歌词解析与同步

- 支持 LRC 歌词文件的解析。
- 实现歌词与音频的同步显示。

4. 音频与歌词文件的自动匹配

- 根据音频文件名自动在特定文件夹查找对应的歌词文件。
- 自动加载歌词。

5.3 知 识 点

5.3.1 ProgressBar

ProgressBar 控件即进度条控件，是一种常见的用户界面元素，常用于需要大量时间的场合，用它来指示当前处理进度、完成的百分比，而不至于让用户迷惑。例如，用于显示文件下载进度、显示批量文件格式的转换、显示播放器的当前播放进度、显示软件当前的安装进度等。

进度条也有诸多事件、方法和属性。不过其事件和方法一般较少使用或者没有使用的必要，因此下面主要介绍其属性。其主要属性如表 5-1 所示。

表 5-1 ProgressBar 控件的常用属性

属 性	说 明
Maximum	设置或返回进度条的最大值，默认值为 100
Minimum	设置或返回进度条的最小值，默认值为 0
Value	设置或返回进度条的当前值
Step	设置或返回一个值，该值用来决定每次调用 Performstep 方法时，Value 属性增加的幅度
Style	决定控件运行时的外观，该属性为枚举值。其取值有 Blocks、Continuous、Marquee。以 Blocks 使用体验最好，因为后面两种都只能向用户表达“程序还没死”这个概念，而不能表达“大概还需要等多久”这种概念

如下是实现文件复制进度的演示代码。

```
private void CopyMultiFiles(string[] fileNames) //fileNames 为待复制文件的数组
{
    progressBar1.Visible = true;
    progressBar1.Maximum = fileNames.Length;
    for (int i = 0; i < fileNames.Length; i++)
    {
        File.Copy(fileNames[i], fileNames[i] + ".Copy");
        progressBar1.Value = i+1;
    }
}
```

5.3.2　TrackBar

TrackBar 允许用户通过拖动滑块在一个范围内选择一个值。它通常用于调节音量、亮度、进度等场景。从外观上看，该控件体现为一个滑块和一个刻度。

TrackBar 控件提供了丰富的属性和事件，可以灵活地满足各种需求。

1. TrackBar 的常用属性

TrackBar 控件的常用属性如表 5-2 所示。

表 5-2　TrackBar 的一些常用属性

属　性	说　明
Minimum	获取或设置滑块的最小值(默认值为 0)
Maximum	获取或设置滑块的最大值(默认值为 10)
Value	获取或设置滑块的当前值
SmallChange	获取或设置滑块的小步长值(例如，按方向键时的变化量，默认值为 1)
LargeChange	获取或设置滑块的大步长值(例如，按 PageUp/PageDown 键时的变化量，默认值为 5)
Orientation	获取或设置滑块的方向(Horizontal 水平或 Vertical 垂直)
TickFrequency	获取或设置刻度线的间隔
TickStyle	获取或设置刻度线的显示方式(None、TopLeft、BottomRight、Both)，即刻度线相对于滑块的位置。 • None：无刻度线。 • TopLeft：刻度线位于滑块的上部(水平滑块)或者左部(垂直滑块)。 • BottomRight：刻度线位于滑块的下部(水平滑块)或者右部(垂直滑块)。 • Both：滑块的上下部(水平滑块)或左右部(垂直滑块)都有刻度线

2. TrackBar 的常用事件

TrackBar 的常用事件如表 5-3 所示。

表 5-3　TrackBar 的常用事件

事　件	说　明
ValueChanged	当滑块的值发生变化时触发
Scroll	当用户拖动滑块或通过键盘改变滑块值时触发

下面看一个简单示例，为了叙述尽量简洁，基本都以代码来呈现，用户只需要新建一个 Windows Forms 项目即可，代码如下。

```
TrackBar trackBar = null;
private void Form1_Load(object sender, EventArgs e)
{
    trackBar = new TrackBar();
    trackBar.Minimum = 0;
    trackBar.Maximum = 100;
    trackBar.Value = 50;
    trackBar.Orientation = Orientation.Horizontal;
    trackBar.TickFrequency = 10;
    trackBar.TickStyle = TickStyle.BottomRight;
    trackBar.LargeChange = 10;
```

```
        trackBar.SmallChange = 1;
        trackBar.Left = 10;
        trackBar.Width = this.Width - 20;
        trackBar.Scroll += TrackBar_Scroll;

        //添加到窗体
        this.Controls.Add(trackBar);
    }

    private void TrackBar_Scroll(object sender, EventArgs e)
    {
        this.Text = trackBar.Value.ToString();
    }
```

程序执行效果如图 5-1 所示。

图 5-1 TrackBar 控件演示

5.3.3 数组参数与 params

在前面讲授过函数，讲解过程中，参数的个数都是极少的。但在现实中，经常面临需要向函数传递大量参数的情况。此时数组参数就是一个可以考虑的选择。

如下演示数组求和函数。

```
static int Add(int[] nums)
{
    int sum = 0;
    for (int i = 0; i < nums.Length; i++)
        sum = sum + nums[i];
    return sum;
}
```

调用时，先创建数组，然后传入数组名即可。

```
int[] numbers = new int[5] { 1, 2, 3, 4, 5 };        //定义数组
Console.WriteLine(Add(numbers));                     //传入数组      结果 15
```

现在对前述数组求和函数稍做改造，即在数组参数前加上 params 修饰。

```
static int Add(params int[] nums)
{
    int sum = 0;
    for (int i = 0; i < nums.Length; i++)
        sum = sum + nums[i];
    return sum;
}
```

此时，虽然函数定义有了改变，但是前述调用方式仍然可以，且结果一样。

那加了 params 后，又有什么变化呢？其实，其好处在于调用时更为灵活，即无须事先创建好数组，而只需要在调用时根据需要直接输入数组元素即可。正因为如此，params 修饰的参数具备如下特点。

- params 型参数允许函数接收可变数量的参数。
- params 型参数必须是函数的最后一个参数。

因此，对于 params 型数组参数，除了可以采用传统的调用方式，也可以采用如下形式。

```
Console.WriteLine(Add(1,2,3,4,5));              //结果 15
```

5.3.4 字符串

字符串即 string 类型，也是一种极为常见的数据类型，在实际应用中被大量使用。字符串用双引号来表达，例如：

```
string s="中国 Zhongguo";
```

在 C# 中，默认采用 Unicode 字符编码，每个字符都算一个长度，因此字符串“ab”“12”“科技”的长度都为 2。字符串的长度可以通过属性 Length 来获取，其值即字符串中字符个数，如上 s.Length 的值为 10。

字符串具有不可变性。例如，若在前述的基础上给 s 重新赋值，即 s=“中国”，该赋值语句是创建了一个新字符串，并不是改变了原始字符串。基于此，字符串可以视为只读的字符数组，索引从 0 开始，如 s[2] 的值为 Z，但为其赋值则出错，如 s[2] ='z' 将会出错。

当字符串中有特殊字符时，此时需要进行字符串的转义操作。例如，由于双引号是字符串的标记性字符，若字符串本身含有双引号，此时将会出错。例如：

```
string s="他对我说："编程时务必要有精益求精的态度"。";        //出错
```

字符转义通过“\”进行，常见的转义符如表 5-4 所示。

表 5-4 常见转义符

转义符号	含　义
\\	\
\n	换行
\t	制表符
\'	单引号
\"	双引号

此外，也可以利用字符数组来构建字符串。例如：

```
char[] chars = { 'C', 'h', 'i', 'n', 'a' };
string s = new string(chars); //"China"
```

5.3.5 常用字符串静态函数

本节介绍与字符串处理相关的静态函数。这些函数遵从如下的调用形式。

```
调用方法：string.方法名()
```

以下将详细介绍 string.Compare()、string.Format()、string.IsNullOrEmpty()和 string.Join()这 4 个常用的字符串函数。

1. string.Compare()

string.Compare()方法用于比较两个字符串，它返回一个整数，指示第一个字符串的顺序是小于、等于还是大于第二个字符串。

其典型重载形式如下。

```
int Compare(string strA, string strB)
int Compare(string strA, string strB, bool ignoreCase)
int Compare(string strA, string strB, StringComparison comparisonType)
```

其中：

- strA：第一个字符串。
- strB：第二个字符串。
- ignoreCase：如果为 true，则忽略大小写进行比较。
- comparisonType：指定比较的类型，如 StringComparison.Ordinal、StringComparison.OrdinalIgnoreCase 等。

当返回值为负数时，表明 strA 小于 strB；当返回值为正数时，则表明 strA 大于 strB；否则表明二者相等。

```
static void Main()
{
    string str1 = "Abc";
    string str2 = "abc";

    //区分大小写的比较
    int iResult = string.Compare(str1, str2, false);
    Console.WriteLine("区分大小写的比较结果：" + iResult);                    //输出：-1

    //不区分大小写的比较
    iResult = string.Compare(str1, str2, true);
    Console.WriteLine("不区分大小写的比较结果：" + iResult);                  //输出：0

    //使用 StringComparison.Ordinal 进行区分大小写的比较
    iResult = string.Compare(str1, str2, StringComparison.Ordinal);
    Console.WriteLine("使用 StringComparison.Ordinal：" + iResult);           //输出：-32

    //使用 StringComparison.OrdinalIgnoreCase 进行不区分大小写的比较
    iResult = string.Compare(str1, str2, StringComparison.OrdinalIgnoreCase);
    Console.WriteLine("使用 StringComparison.OrdinalIgnoreCase：" + iResult); //输出：0
}
```

注意：使用 string.Compare(str1,str2,StringComparison.Ordinal)进行字符串比较时，是一种区分大小写的二进制比较。由于"A"和"a"的 ASCII 码值差为 32，因此返回－32。

注意：使用 string.Compare(str1,str2,false)，其中，false 表示区分大小写。但该重载形式只在乎三种关系，即在前、在后和相等；而不在意究竟在前或者在后多少。由于"Abc"和"abc"在区分大小写的情况下不相等，且"Abc"在字典顺序中位于"abc"之前，因此返回－1。

2. string.Format()

string.Format 允许将多个对象或值插入字符串的特定位置，并控制它们的显示格式。string.Format 是 C＃中处理复杂字符串拼接和格式化的常用工具。

其常用重载形式如下。

```
public static string Format(string format, params object[] args)
```

其中：

- format：一个复合格式字符串，包含占位符{0}、{1}等，用于指定插入值的位置和格式。
- args：要插入格式字符串中的参数列表(可以是变量、常量或表达式)。

该函数的关键在于 format 参数，其形式为{index：formatString}，包含两部分，即指定参数位置的占位符和指定显示格式的格式化选项。

其中，占位符的形式为{index}，其中 index 是参数的索引(从 0 开始)，如{0}表示第一个参数，{1}表示第二个参数，以此类推。

格式化选项则较多，一些常见的格式化选项如表 5-5 所示。

表 5-5 常见的格式化选项

格　式	说　明
C 或 c	货币格式
D 或 d	十进制格式(适用于整数)
F 或 f	浮点数格式(可以指定小数位数，如 F2 表示保留两位小数)
N 或 n	数字格式(带千位分隔符，如 1,234.56)
P 或 p	百分比格式(如 0.123 格式化为 12.30％)
X 或 x	十六进制格式(适用于整数)
G 或 g	通用格式(根据值的类型自动选择最合适的格式)
0	数字占位符(如 0000 会将 12 格式化为 0012)
＃	数字占位符(如＃＃.＃＃会将 12.3 格式化为 12.3)

如下演示占位符输出及基本的数字格式化输出。

```
static void Main()
{
    string name = "小明";
    int age = 20;
    double height = 1.68;
```

```
        string result = string.Format("姓名: {0}, 年龄: {1}, 身高: {2:F2} ", name, age, height);
        Console.WriteLine(result);   //输出: 姓名: 小明, 年龄: 20, 身高: 1.68
    }
```

如下是格式化数字。

```
    static void Main()
    {
        int number = 12345;
        double price = 123.4567;

        string result = string.Format("数量: {0:N0}, 价格: {1:C2}", number, price);
        Console.WriteLine(result);   //数量: 12,345, 价格: ¥123.46
    }
```

注意: 价格符号的显示将因操作系统的不同而不同,例如,也有可能显示为 $123.46 等。

3. string.IsNullOrEmpty()

string.IsNullOrEmpty()方法用于确定指定的字符串是否为 null 或空字符串("")。这是一个常用的辅助方法,用于在处理字符串时进行空值检查。

其语法形式为

```
    public static bool IsNullOrEmpty(string value)
```

当返回 true 时,表明当前检测的字符串为 null 或空字符串。

```
    public static void Main()
    {
        string str1 = "";
        string str2 = null;
        string str3 = "    ";
        string str4 = "China";

        Console.WriteLine(string.IsNullOrEmpty(str1));      //输出: True
        Console.WriteLine(string.IsNullOrEmpty(str2));      //输出: True
        Console.WriteLine(string.IsNullOrEmpty(str3));      //输出: False
        Console.WriteLine(string.IsNullOrEmpty(str4));      //输出: False
    }
```

注意: 如果需要检查字符串是否为 null、空字符串或仅包含空白字符,可以使用 string.IsNullOrWhiteSpace()方法。该函数检查上述 str3 将返回 true,检查 str4 则仍然为 false。

4. string.Join()

string.Join()方法用于将字符串数组或集合中的元素连接为一个单一的字符串,并在元素之间插入指定的分隔符。

其常用重载形式为

```
    public static string Join(string separator, string[] values);
```

其中：

- separator：用于分隔元素的字符串。
- values：一个字符串数组。

```
public static void Main()
{
    string[] words = { "教育", "科技", "人才", "C#" };
    string joined = string.Join(" ", words);
    Console.WriteLine(joined);    //"教育 科技 人才 C#"

    //使用不同的分隔符
    joined = string.Join(", ", words);
    Console.WriteLine(joined);    //"教育, 科技, 人才, C#"
}
```

注意：string.Join()只能用于字符串数组或集合。如果需要连接其他类型的数组或集合，需要先将它们转换为字符串。

5.3.6 字符串插值 $

字符串插值从C# 6.0开始引入，是一种较string.Format更简洁、更易读的字符串格式化方法。$符号是实现字符串插值的关键，它允许在字符串字面量中直接嵌入表达式，从而简化了字符串的构建过程。其基本语法如下。

```
string result = $"文本{表达式}文本";
```

字符串插值具有如下特性。

- 可读性好：字符串插值使代码更接近自然语言，易于阅读和理解。
- 语法简洁：无须使用string.Format()方法和位置参数，代码更简洁。
- 支持表达式：可以在花括号内使用任意表达式，如方法调用、属性访问等。
- 类型安全：编译器会在编译时检查表达式的类型，减少运行时错误。

首先看一个与前文类似的格式化示例。

```
string name = "小明";
int age = 20;
string result= $"姓名:{name}, 年龄:{age}";
Console.WriteLine(result);       //"姓名: 小明, 年龄: 20"
```

可见，通过$实现了和string.Format相同的效果，但是其更为简洁。

现在看表达式的示例。

```
int a = 5;
int b = 10;
string sum = $"{a} + {b} = {a + b}";
Console.WriteLine(sum);       //5 + 10 = 15.
```

可见，$符号使得花括号{}内的内容会被解析为表达式，其结果将被嵌入最终的字符

串中。

既然可以执行表达式，自然也可以执行函数，例如：

```
public class Star
{
    public string Name { get; set; }
    public int Age { get; set; }

    public string Introduce()
    {
        return $"我叫{Name},现在{Age}岁";
    }
}
```

调用演示代码如下。

```
Star star = new Star { Name = "小明", Age = 20 };
string sResult = star.Introduce();
Console.WriteLine(sResult);        //我叫小明,现在 20 岁
```

5.3.7 常用字符串函数

本节介绍常用的字符串函数，这些函数遵从如下调用形式。

```
string s="";
s.函数()
```

如下根据字符串函数的功能，将其分为 4 类，如表 5-6 所示。

表 5-6 4 类常用字符串函数

函数类别	相关函数名字
转换类	ToCharArray、ToLower、ToUpper
查找类	Contains、EndsWith、StartsWith、IndexOf、LastIndexOf
编辑修改类	Trim、TrimEnd、TrimStart、Replace
提取类	Substring、Split

其中：

- 转换类：用于将字符串转换为其他表达形式。
- 查找类：用于在字符串中查找子字符串的位置或存在性。
- 编辑修改类：用于修改字符串内容。
- 提取类：用于从字符串中提取子字符串。

练一练：思考如下几个场景属于转换、查找、编辑、提取中的哪一种。

- 判断一个人是否姓王。
- 获取一个单姓姓名中的姓。
- 将张三丰改为张叁丰。
- 统计历年四级试卷中的高频词。

- 字符串中有很多常见应用,例如,一个文本文档中有多少个特定的关键词?一个文本文档中有多少个句子?

如下转换类和查找类函数均以 string s = "中国 Zhongguo" 为例。

1. ToCharArray

将字符串转换为字符数组。

```
string s = "中国 Zhongguo";
char[] charArray = s.ToCharArray();

Console.WriteLine("字符数组内容:");
foreach (char c in charArray)
{
    Console.Write(c + " ");        //最终输出  中 国 Z h o n g g u o
}
```

2. ToLower

将字符串中的所有字符转换为小写。

```
string s = "中国 Zhongguo";
string sResult = s.ToLower();

Console.WriteLine("转换为小写:" + sResult);          //中国 zhongguo
```

3. ToUpper

将字符串中的所有字符转换为大写。

```
string s = "中国 Zhongguo";
string sResult = s.ToUpper();

Console.WriteLine("转换为大写:" + sResult);          //中国 ZHONGGUO
```

4. Contains(string s)

检查字符串中是否包含指定的子字符串。

```
string s = "中国 Zhongguo";
bool sResult1 = s.Contains("中国");
bool sResult2 = s.Contains("中心");

Console.WriteLine("是否包含 '中':" + sResult1);        //true
Console.WriteLine("是否包含 'Beijing':" + sResult2);   //false
```

5. bool EndsWith(string s)

检查字符串是否以指定的子字符串结尾。

```
string s = "中国 Zhongguo";
bool sResult1 = s.EndsWith("guo");
```

```
bool sResult2 = s.EndsWith("中");

Console.WriteLine("是否以 'guo' 结尾: " + sResult1);        //true
Console.WriteLine("是否以 '中' 结尾: " + sResult2);         //false
```

6. bool StartsWith(string s)

检查字符串是否以指定的子字符串开头。

```
string s = "中国 Zhongguo";
bool sResult1 = s.StartsWith("中国");
bool sResult2 = s.StartsWith("Zhong");

Console.WriteLine("是否以 '中国' 开头: " + sResult1);        //true
Console.WriteLine("是否以 'Zhong' 开头: " + sResult2);       //false
```

7. IndexOf

该函数有两种常用重载形式。

- IndexOf(string s)：在字符串中查找子字符串，返回第一次出现的位置(索引从 0 开始)，如果未找到则返回－1。
- IndexOf(string s,int start)：从指定的起始位置 start 处开始查找子字符串，返回第一次出现的位置(索引从 0 开始)，如果未找到则返回－1。

```
string s = "中国 Zhongguo";
int sResult1 = s.IndexOf("中国");
int sResult2 = s.IndexOf("中心");
int sResult3 = s.IndexOf("o");                    //等价于 s.IndexOf("o", 0);
int sResult4 = s.IndexOf("o", 5);

Console.WriteLine("'中国' 首次出现的位置: " + sResult1);                    //0
Console.WriteLine("'中心' 首次出现的位置: " + sResult2);                    //－1
Console.WriteLine("从位置 0 开始查找 'o' 首次出现的位置: " + sResult3);    //4
Console.WriteLine("从位置 5 开始查找 'o' 首次出现的位置: " + sResult4);    //9
```

此外，在查找时还可以设定附加选项，如是否区分大小写，该重载形式如下。

```
public int IndexOf(string text,StringComparison comparisonType)
```

该重载形式请自行学习。

8. LastIndexOf

该函数有两种常用重载形式。

- int LastIndexOf(string s)：从字符串的右侧开始查找子字符串，返回第一次出现的位置(索引从 0 开始)，如果未找到则返回－1。
- int LastIndexOf(string s,int start)：从指定的起始位置 start 处开始从右侧向左查找子字符串，返回第一次出现的位置(索引从 0 开始)，如果未找到则返回－1。

```
string s = "中国 Zhongguo";
int sResult1 = s.LastIndexOf("o");
int sResult2 = s.LastIndexOf("o", 7);

Console.WriteLine("'o'最后一次出现的位置: " + sResult1);                    //9
Console.WriteLine("从位置 7 开始向左查找 'o'出现的位置: " + sResult2);     //4
```

练一练：请说出如下各个 i 值为多少。

```
string s = "教育强国,科技强国,Talent 强国";
int i = s.IndexOf("强国");
i = s.LastIndexOf("强国");
i = s.IndexOf("强国",1);
i = s.LastIndexOf("强国",3);
```

练一练：如何利用 IndexOf 实现查找一个字符串中有多少个特定的字符(例如,红楼梦中出现多少个林黛玉)?

练一练：请自行学习 IndexOfAny 和 LastIndexOfAny。

下面开始讲解字符串编辑修改类函数,均以字符串 s="中国 Zhongguo＃"为例,注意,该字符串的第一个字符为空格,最后一个字符为＃。

9. Trim

移除字符串开头和结尾的空白字符(包括空格、制表符、换行符等)。

其典型重载形式如下。

```
public string Trim();
public string Trim(params char[] trimChars);                    //移除指定的字符

string s = " 中国 Zhongguo＃ ";
string sResult = s.Trim();

Console.WriteLine( $"原始字符串: '{s}'");
Console.WriteLine( $"Trim 后字符串: '{ sResult }'");       //'中国 Zhongguo＃'
```

可见,字符串中的第一个空格被成功去除。

如下为移除指定字符的示例。

```
string s = " 中国 Zhongguo＃ ";
string sResult = s.Trim(' ', '＃');

Console.WriteLine( $"Trim 后字符串: '{ sResult }'");       //'中国 Zhongguo'
```

可见,字符串开头的空格和字符串尾部的＃都被成功移除。

10. TrimStart

移除字符串开头的空白字符或指定字符。其语法形式如下。

```
public string TrimStart(params char[] trimChars);

string s = " 中国 Zhongguo＃ ";
```

```
string sResult = s.TrimStart();
Console.WriteLine($"TrimStart 后字符串: '{sResult}'");    //中国 Zhongguo#
```

字符串开头的空格被成功移除。

11. TrimEnd

移除字符串结尾的空白字符或指定字符。语法形式同 TrimStart。

```
string s = " 中国 Zhongguo#";
string sResult1 = s.TrimEnd();
string sResult2 = s.TrimEnd('#');
Console.WriteLine($"TrimEnd 后字符串: '{sResult1}'");    //" 中国 Zhongguo#"
Console.WriteLine($"TrimEnd 后字符串: '{sResult2}'");    //" 中国 Zhongguo"
```

12. Replace

替换字符串中的指定字符或子字符串。

该函数有以下两种常用重载形式。

```
public string Replace(char oldChar, char newChar)              //替换字符
public string Replace(string oldValue, string newValue)        //替换子字符串
```

不过由于字符必定是字符串,因此一般只需要使用第二种形式即可。

```
string s = " 中国 Zhongguo#";
//替换字符
string sResult = s.Replace('#', '!');
Console.WriteLine($"替换字符后: '{sResult}'");            //"中国 Zhongguo!"
//替换子字符串
sResult = s.Replace("Zhongguo", "China");
Console.WriteLine($"替换子字符串后: '{sResult}'");        //"中国 China#"
```

练一练:“今晚月亮好圆好圆啊”.Replace(“圆”,”弯”)的结果是什么?

最后看提取类函数。

13. Substring

该函数具有如下两种常用重载形式。

```
public string Substring(int startIndex)
public string Substring(int startIndex,int count)
```

其中,第一种重载形式是从 startIndex 位置开始,提取此位置后所有的字符(包括当前位置的字符);第二种重载形式是从 startIndex 位置开始,提取 count 个字符。

```
string s = "精益求精练编程";
string sResult = s.Substring(2);
Console.WriteLine(sResult);                                    //求精练编程
sResult = s.Substring(2,2);
Console.WriteLine(sResult);                                    //求精
```

14. Split

该方法利用指定的字符作为分隔标记对字符串进行切割。其重载形式很多，较为典型的重载形式如下。

```
public string[] Split (params char[] separator)
```

该形式表示根据 separator 指定的字符切分字符串，返回切分后的字符串数组。separator 可以是不包含分隔符的空数组或空引用。另外，由于参数由 params 修饰，因此调用时可以直接传入各个用于切割的字符，而不必创建数组。

默认情况下，该函数遵从切西瓜原理，即一刀两段，两刀三段，n 刀将得到 n+1 段。

这也是 Split 函数诸多重载中唯一的带 params 参数的版本。

看一个简单示例。

```
string s = "中国 Zhongguo";
string [] sSegs = s.split('n')          //{"中国 Zho", "gguo"}
```

可见，一刀(n)得到了 2 段。

练一练：对如上字符串"中国 Zhongguo"，若采用字母 o 作为分隔标记进行切割将得到什么？

现在看一个稍微复杂的例子。如果想从字符串 s="a,b.6,,3_8,f."中取得各个字母和数字，该如何办呢？方法很简单，如以下代码所示。

```
s=" a,b.6,,3_8,f.";
string[] t=s.Split(',','.','_');        //对应上述 params 参数版本
Console.WriteLine(t.Length);            //8
```

运行程序，所得的数组有 8 个元素，如图 5-2 所示。

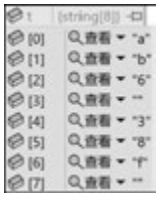

图 5-2 字符串切割演示

可见该方法可以有效切分出所需要的内容。但同时也有点小问题，切割后的 8 个元素中，有两个空字符串。

注意到原始字符串中，6 后面有两个连续的逗号，f 后有个句点，正是这两处导致最终结果中出现两个为空串的元素。

事实上，由于 char 数组必定是字符串数组，因而在不适用 params 版本的情况下，另一种典型的重载形式如下。

```
public string[] Split (string[] separator, StringSplitOptions options)
```

该重载形式利用指定的字符串数组元素进行切割，并可以通过 StringSplitOptions 控制切割后是否删除由于切割而得到的空元素。

当第二个参数取值 StringSplitOptions.None 时，返回值可以包括含有空字符串的数组元素。

当第二个参数取值 StringSplitOptions.RemoveEmptyEntries 时，其返回值不包括仅为空字符串的数组元素。

例如，如下演示英文文本中英文单词的提取。

```
string s = "I come from China,I am a student.";
char[] chars = new char[] { ',', '.', ' ' };
string[] sSeg = s.Split(chars, StringSplitOptions.RemoveEmptyEntries);

Console.WriteLine(string.Join("\n", sSeg));
```

执行程序,最终得到的数组中包含所有单词。

此外,该函数的其他重载形式如下。

```
public string[] Split (char[] separator, int count, StringSplitOptions options)
public string[] Split (string[] separator, int count, StringSplitOptions options)
```

下面看一个小的字符串示例,即如何从 LRC 歌词中分离出时间和相应的歌词。

```
//示例: LRC 歌词解析-此处仅使用了 4 句歌词,不过同样也适用更多歌词的情况
string s = "[00:36.00]记忆中的每一天\n[00:40.45]是最美丽的章节\n[00:43.69]揪心的思念\n
[00:46.92]堆积沉淀";
string[] sLines = s.Split('\n');
string sTemp = null, sMinute = null, sSecond = null, sLrc = null;
double dTime = 0;
for (int i = 0; i < sLines.Length; i++)
{
    sTemp = sLines[i];
    sMinute = sTemp.Substring(1, 2);    //分钟
    sSecond = sTemp.Substring(4, 5);
    sLrc = sTemp.Substring(10);
    dTime = Convert.ToInt32( sMinute ) * 60 +  double.Parse( sSecond);
    Console.WriteLine("行{0} 时间{1} 歌词{2}", i + 1, dTime, sLrc);
}
```

练一练:设字符串 s="教育强国,科技强国,人才强国。",请至少用三种方法,编程实现 s 中"强国"个数的统计。

5.3.8 Stopwatch 计时

在编程过程中,经常遇到需要统计程序耗时的情景。这可以通过 Stopwatch 来实现。该类位于命名空间 System.Diagnostics,其常用的属性和方法分别如表 5-7 和表 5-8 所示。

表 5-7 Stopwatch 常用属性

属 性	说 明
Elapsed	已经历了多久,为 TimeSpan 类型
ElapsedMilliseconds	已经历的毫秒数,long 型
ElapsedTicks	已经历的 Tick 数,long 型
IsRunning	Stopwatch 是否仍然在工作

表 5-8 Stopwatch 常用方法

方 法	说 明
Reset()	重置计时,即将表 5-7 中的属性置 0
Restart()	重启计时
Start()	启动计时
Stop()	停止计时

若要统计某段程序的执行耗时情况,一种最简单的使用方式是,在该代码块的前面调用 Stopwatch 实例对象的 Start()方法,而在代码块的后面调用该实例对象的 Stop()方法,此时再读取其 ElapsedMilliseconds 等属性即可知道程序执行耗费了多少时间。

```
static void Main(string[] args)
{
    Stopwatch sw = new Stopwatch();
    Console.WriteLine("开始计时");
    sw.Start();
    int s = 0;
    for (int i = 0; i < 10000000; i++)
        s += i;
    sw.Stop();
    Console.WriteLine("执行完毕,停止计时。程序执行耗时{0}毫秒",sw.ElapsedMilliseconds);
}
```

程序执行效果如图 5-3 所示。

图 5-3 利用 Stopwatch 计时

5.3.9 StringBuilder 与字符串高效操作

StringBuilder 与 string 对象相比的最大好处在于,在对 StringBuilder 对象进行追加、插入、替换、移除操作时,不会产生新对象,因此它适用于对字符串进行频繁操作的场合。

其最为常用的属性是 Length,表征了字符串的实际长度。

其常用方法如表 5-9 所示。

表 5-9 StringBuilder 常用方法

方 法	说 明
Append()	用于将文本或者对象的字符串表示形式添加到当前对象的结尾处
AppendFormat()	用于对追加部分字符串进行格式化
AppendLine()	将指定字符串的副本和默认的换行符追加到当前对象的末尾
Clear()	从当前 StringBuilder 实例中移除所有字符
Insert()	将指定的内容(字符、字符串)等插入当前实例中的指定位置
Remove()	将指定范围的字符从当前实例中删除
Replace()	将当前实例中指定的内容(字符、字符串)等替换为指定内容
ToString()	将当前的 StringBuilder 对象转换为字符串表达

示例如下。

```
StringBuilder sb = new StringBuilder();
sb = new StringBuilder("酒逢知己饮");
sb.AppendLine(" 诗向会人吟");
sb.AppendLine("人间四月天 麻城看杜鹃");
Console.WriteLine(sb.ToString());
sb.Replace("人","世");
Console.WriteLine(sb.ToString());
```

程序运行效果如图 5-4 所示。

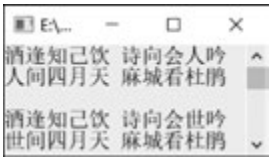

图 5-4 StringBuilder 示例

5.3.10 Path 类

Path 类是一个静态工具类，位于 System.IO 命名空间中，用于处理文件路径和目录路径。它提供了一系列静态方法，用于操作路径字符串，如获取文件名、获取扩展名、获取目录名等，此外也可以用于获取操作系统中一些特殊的目录或临时文件名等。

其常用属性如表 5-10 所示。

表 5-10 Path 类常用属性

属 性	说 明
PathSeparator	表示当多个路径字符串连接在一起时，用于分割各个文件或目录路径的字符。通常，当为搜索指定多个路径时，使用该属性。Windows 使用的默认字符为分号(;)
InvalidPathChars	是一个数组，包含不能用于路径字符串的字符。已被弃用
DirectorySeparatorChar	表示用于分隔路径字符串中各目录间的字符。Windows 使用的默认字符为反斜杠(\)；在 Linux 和 macOS 上，分隔符是/
VolumeSeparatorChar	表示用于将驱动器盘符与字符串路径的其余部分进行分隔的字符。Windows 使用的默认字符为冒号(:)

其常用方法较多，下面介绍其常用方法。

1. Path.Combine

将多个字符串合并为一个路径。

```
public static string Combine(params string[] paths)
```

简单示例如下。

```
string path1 = "C:\\Users";
string path2 = "hfcas";
string path3 = "Documents";
```

```
string sResult = Path.Combine(path1, path2, path3);
Console.WriteLine(sResult);        //输出: C:\Users\hfcas\Documents
```

2. Path.GetFileName

获取路径中的文件名(包括扩展名)。

```
public static string GetFileName(string path)
```

示例如下。

```
string path = "C:\\Users\\hfcas\\Documents\\file.txt";
string sResult = Path.GetFileName(path);
Console.WriteLine(sResult);        //输出: file.txt
```

3. Path.GetFileNameWithoutExtension

获取路径中的文件名(不包括扩展名)。

```
public static string GetFileNameWithoutExtension(string path)
```

示例如下。

```
string path = "C:\\Users\\hfcas\\Documents\\file.txt";
string sResult = Path.GetFileNameWithoutExtension(path);
Console.WriteLine(sResult);        //输出: file
```

4. Path.GetExtension

获取路径中的文件扩展名。

```
public static string GetExtension(string path)
```

示例如下。

```
string path = "C:\\Users\\hfcas\\Documents\\file.txt";
string sResult = Path.GetExtension(path);
Console.WriteLine(sResult);        //输出: .txt
```

5. Path.GetDirectoryName

获取路径中的目录名。

```
public static string GetDirectoryName(string path)
```

示例如下。

```
string path = "C:\\Users\\hfcas\\Documents\\file.txt";
string sResult = Path.GetDirectoryName(path);
Console.WriteLine(sResult);        //输出: C:\Users\hfcas\Documents
```

6. Path.GetFullPath

将相对路径转换为绝对路径。

```
public static string GetFullPath(string path)
```

示例如下。

```
string relativePath = "file.txt";
string sResult = Path.GetFullPath(relativePath);

Console.WriteLine(sResult);
//输出: C:\My\Develop\WindowsFormsApp1\ConsoleApp1\bin\Debug\file.txt
```

注意：该方法不会检测文件的存在性。另外，其输出将根据具体情况而不同。

7. Path.GetTempPath

获取系统的临时文件夹路径。

```
public static string GetTempPath()
```

示例如下。

```
string sResult = Path.GetTempPath();
Console.WriteLine(sResult);        //输出: C:\Users\hfcas\AppData\Local\Temp\\
```

注意：该方法的输出将根据具体情况而不同。

8. Path.GetTempFileName

在系统的临时文件夹中创建一个唯一的临时文件，并返回其完整路径。

```
public static string GetTempFileName()
```

示例如下。

```
string sResult = Path.GetTempFileName();
Console.WriteLine(sResult);
//输出: C:\Users\hfcas\AppData\Local\Temp\tmpCF0A.tmp
```

注意：该方法的输出将根据具体情况而不同。

9. Path.ChangeExtension

更改路径中的文件扩展名。

```
public static string ChangeExtension(string path, string extension)
```

示例如下。

```
string path = "C:\\Users\\hfcas\\Documents\\file.txt";
string newPath = Path.ChangeExtension(path, ".docx");
Console.WriteLine(newPath);        //输出: C:\Users\hfcas\Documents\file.docx
```

10. Path.HasExtension

检查路径是否包含文件扩展名。

```
public static bool HasExtension(string path)
```

示例如下。

```
string path1 = "C:\\Users\\hfcas\\Documents\\file.txt";
string path2 = "C:\\Users\\hfcas\\Documents\\file";
Console.WriteLine(Path.HasExtension(path1));      //输出: True
Console.WriteLine(Path.HasExtension(path2));      //输出: False
```

11. Path.IsPathRooted

检查路径是否是绝对路径。

```
public static bool IsPathRooted(string path)
```

示例如下。

```
string path1 = "C:\\Users\\hfcas\\Documents\\file.txt";
string path2 = "bin\\file.txt";
Console.WriteLine(Path.IsPathRooted(path1));      //输出: True
Console.WriteLine(Path.IsPathRooted(path2));      //输出: False
```

练一练：请写出如下程序的输出或者对输出进行解释。

```
string path = @"C:\Program Files\ButSoft\Prince.exe";
Console.WriteLine(Path.GetDirectoryName(path));
Console.WriteLine(Path.GetExtension(path));
Console.WriteLine(Path.GetFileName(path));
Console.WriteLine(Path.GetFileNameWithoutExtension(path));
Console.WriteLine(Path.GetRandomFileName());
Console.WriteLine(Path.GetTempFileName());
Console.WriteLine(Path.GetTempPath());
```

5.3.11 泛型集合 List<T>

List<T>是一个泛型集合类，位于 System.Collections.Generic 命名空间中，因此需要应用该命名空间，与其对应的普通集合是 ArrayList。它提供了一种类型安全且高效的方式来存储和操作一组对象。List<T>相当于一个动态数组，可以根据需要自动调整大小，支持添加、删除、查找、排序等操作。

泛型集合具备类型安全的特点，操作高效，无须装箱和拆箱操作。

List<T>集合常用属性如表 5-11 所示。

表 5-11 List＜T＞常用属性

属　性	说　明
Count	获取集合中元素的数量
Capacity	获取或设置集合的容量，该属性值不小于 Count

注意：正如杯子，其容量为 50(Capacity)，但实际可能只装了 30(Count)。不过 List＜T＞是一个容量可以根据需求自动扩充的神奇杯子。

对 List＜T＞，其中元素的访问方式类似于数组，即通过索引来访问。

对于各类集合而言，其相关方法主要是通过增、删、改、查、排、转来展开的。这 6 类方法如表 5-12 所示。

表 5-12 List＜T＞常用方法

类　别	相 关 方 法
增	Add(T item)、AddRange(IEnumerable＜T＞collection)、Insert(int index, T item)、InsertRange(int index, IEnumerable＜T＞collection)
删	Remove(T item)、RemoveAt(int index)、RemoveRange(int index, int count)、Clear()
改	List[i] = x
查	Contains(T item)、IndexOf(T item)、IndexOf(T item, int index)、LastIndexOf(T item)、LastIndexOf(T item, int index)
排	Sort
转	ToArray

下面介绍 List＜T＞的这 6 类操作相关的方法。

1. Add(T item)

将单个元素添加到集合的末尾。

```
List<int> numbers = new List<int>();
numbers.Add(10);
numbers.Add(20);
Console.WriteLine(string.Join(", ", numbers));      //输出: 10, 20
```

2. AddRange(IEnumerable＜T＞collection)

将一组元素添加到集合的末尾。

```
List<int> numbers = new List<int>{ 10, 20 };
numbers.AddRange(new int[] { 30, 40 });
Console.WriteLine(string.Join(", ", numbers));      //输出: 10, 20, 30, 40
```

3. Insert(int index, T item)

在指定索引处插入元素。

```
List<int> numbers = new List<int>{ 10, 20, 30 };
numbers.Insert(1, 15);
Console.WriteLine(string.Join(", ", numbers));      //输出: 10, 15, 20, 30
```

4. Insert(int index,IEnumerable＜T＞ collection)

在指定索引处插入一组元素。

```
List＜int＞ numbers = new List＜int＞{ 10, 20, 30 };
numbers.InsertRange(1, new int[] { 15, 25 });
Console.WriteLine(string.Join(", ", numbers));          //输出: 10, 15, 25, 20, 30
```

5. Remove(T item)

移除集合中第一个匹配的元素,即按内容来删。

```
List＜string＞ fruits = new List＜string＞{ "Apple", "Banana", "Pear" };
fruits.Remove("Banana");
Console.WriteLine(string.Join(", ", fruits));           //输出: Apple, Pear
```

6. RemoveAt(int index)

移除指定索引处的元素,即按位置来删。

```
List＜string＞ fruits = new List＜string＞{ "Apple", "Banana", "Pear" };
fruits.RemoveAt(1);
Console.WriteLine(string.Join(", ", fruits));           //输出: Apple, Pear
```

7. RemoveRange(int index,int count)

从指定索引处开始移除指定数量的元素,即按位置来批量删。

```
List＜string＞ fruits = new List＜string＞{ "Apple", "Banana", "Pear", "Peach" };
fruits.RemoveRange(1, 2);
Console.WriteLine(string.Join(", ", fruits));           //输出: Apple, Peach
```

8. Clear

移除集合中的所有元素。

```
List＜string＞ fruits = new List＜string＞{ "Apple", "Banana", "Pear" };
fruits.Clear();
Console.WriteLine("元素数量: " + fruits.Count);       //输出: 0
```

9. Contains(T item)

查找集合是否包含指定元素。

```
List＜string＞ fruits = new List＜string＞{ "Apple", "Banana", "Pear" };
bool containsBanana = fruits.Contains("Banana");
Console.WriteLine(containsBanana);                      //输出: True
```

10. IndexOf

该函数有两种常用重载形式。

IndexOf(T item):返回指定元素在集合中的第一个匹配项的索引,未找到则返回－1。

```
List<string> fruits = new List<string> { "Apple", "Banana", " Pear " };
int index = fruits.IndexOf("Banana");
Console.WriteLine(index);                                    //输出：1
```

IndexOf(T item,int index)：从指定索引处开始查找指定元素，返回第一个匹配项的索引，未找到则返回－1。

```
List<string> fruits = new List<string> { "Apple", "Banana", "Pear", "Banana" };
int index = fruits.IndexOf("Banana", 2);

Console.WriteLine(index);                                    //输出：3
```

11. LastIndexOf

该函数同样有两种常用重载形式。

LastIndexOf(T item)：返回指定元素在集合中的最后一个匹配项的索引，未找到则返回－1。

```
List<string> fruits = new List<string> { "Apple", "Banana", "Pear", "Banana" };
int lastIndex = fruits.LastIndexOf("Banana");
Console.WriteLine(lastIndex);                                //输出：3
```

LastIndexOf(T item,int index)：从指定索引处开始向左查找指定元素，返回最后一个匹配项的索引，未找到则返回－1。

```
List<string> fruits = new List<string> { "Apple", "Banana", "Cherry", "Banana" };
int lastIndex = fruits.LastIndexOf("Banana", 2);
Console.WriteLine(lastIndex);                                //输出：1
```

12. Sort

对集合中的元素进行排序，默认为升序排列。

```
List<int> numbers = new List<int> { 5, 3, 8, 1, 9 };
numbers.Sort();
Console.WriteLine(string.Join(", ", numbers));       //输出：1, 3, 5, 8, 9
```

13. ToArray

将集合转换为数组。

```
List<int> numbers = new List<int> { 1, 2, 3 };
int[] array = numbers.ToArray();
Console.WriteLine(string.Join(", ", array));         //输出：1, 2, 3
```

下面是一个综合性的例子。

```
using System.Collections.Generic;
static void Main()
{
```

```
        //创建一个 List<int> 集合
        List<int> numbers = new List<int>();

        //添加元素
        numbers.Add(10);
        numbers.Add(20);
        numbers.AddRange(new int[] { 30, 40 });

        //插入元素
        numbers.Insert(1, 15);
        numbers.InsertRange(3, new int[] { 25, 35 });

        //修改元素
        Numbers[3]=24;

        //输出集合
        Console.WriteLine("集合内容: " + string.Join(", ", numbers)); //输出: 10, 15, 20, 24,
                                                                      //35, 30, 40

        //移除元素
        numbers.Remove(24);
        numbers.RemoveAt(2);
        numbers.RemoveRange(3, 2);

        //输出集合
        Console.WriteLine("移除后集合内容: " + string.Join(", ", numbers)); //输出: 10, 15, 35

        //查找元素
        bool contains15 = numbers.Contains(15);
        int indexOf15 = numbers.IndexOf(15);
        int lastIndexOf15 = numbers.LastIndexOf(15);

        Console.WriteLine("是否包含 15: " + contains15);                    //输出: True
        Console.WriteLine("15 的索引: " + indexOf15);                       //输出: 1
        Console.WriteLine("15 的最后索引: " + lastIndexOf15);               //输出: 1

        //排序
        numbers.Sort();
        Console.WriteLine("排序后集合内容: " + string.Join(", ", numbers)); //输出: 10, 15, 35

        //转换为数组
        int[] array = numbers.ToArray();
        Console.WriteLine("数组内容: " + string.Join(", ", array));         //输出: 10, 15, 35

        //清空集合
        numbers.Clear();
        Console.WriteLine("清空后集合内容: " + string.Join(", ", numbers)); //输出:
    }
```

5.3.12 Dictionary<K,V>和 KeyValuePair<K,V>

泛型集合 Dictionary<K,V>是与普通集合 Hashtable 对应的版本,用于存储键值对型

数据，在 System.Collections.Generic 命名空间下。其中，键的取值一般以整型和字符串型较为常见，且键一定不能有重复；值的取值则非常广泛，可以是简单的值类型，也可以是内置引用类型，还可以是自定义类型。

Dictionary<K,V>中的元素操作涉及增加、删除、修改、查找、遍历等操作，其容量可以自动动态调整。其中，其遍历又分为值的遍历、键的遍历、键值对元素遍历。键值对元素同时遍历时，需要借助 KeyValuePair<K,V>来实现，KeyValuePair<K,V>表达的正好就是一个键值对元素，与 Hashtable 中元素遍历时的 DictionaryEntry 对应。

Dictionary<K,V>和 KeyValuePair<K,V>是用于处理键值对的重要数据结构。它们广泛应用于需要快速查找、存储和操作数据的场景。

其常用属性如表 5-13 所示。

表 5-13 Dictionary<K,V> 常用属性

属 性	含 义
Count	获取字典中键值对的数量
Keys	获取包含字典中所有键的集合
Values	获取包含字典中所有值的集合

其常用方法主要围绕其中键值对数据的增加、删除、修改、查找来进行，介绍如下。

1. Add(K key,V value)

向字典中添加一个新的键值对。如果键已存在，则会抛出异常。

```
Dictionary<string, int> students= new Dictionary<string, int>();
students.Add("小明", 20);
students.Add("大明", 30);
Console.WriteLine(string.Join(", ", students));        //输出：[小明, 20], [大明, 30]
```

2. Remove(K key)

移除具有指定键的键值对。如果键不存在，则返回 false。

```
Dictionary<string, int> students = new Dictionary<string, int>{ { "小明", 20 }, { "大明", 30 } };
bool isRemoved = students.Remove("小明");
Console.WriteLine(isRemoved);                          //输出：True
Console.WriteLine(string.Join(", ", students));        //输出：[大明, 30]
```

3. ContainsKey(K key)

检查字典中是否存在指定的键。

```
Dictionary<string, int> students = new Dictionary<string, int>{ { "小明", 20 }, { "大明", 30 } };
bool flag= students.ContainsKey("小明");
Console.WriteLine(flag);                               //输出：True
```

4. ContainsValue(V value)

检查字典中是否存在指定的值。

```
Dictionary<string, int> students = new Dictionary<string, int>{ { "小明", 20 }, { "大明", 30 } };
bool flag= students.ContainsValue(25);
Console.WriteLine(flag);                               //输出：True
```

5. TryGetValue(K key,out V value)

尝试获取指定键的值。如果键存在,则返回 true 并输出值;否则,返回 false。

```
Dictionary<string, int> students = new Dictionary<string, int>{ { "小明", 20 }, { "大明", 30 } };
if (students.TryGetValue("小明", out int age))
    Console.WriteLine("小明的年龄是: " + age);      //输出: 小明的年龄是: 20
else
    Console.WriteLine("未找到 小明");
```

6. Clear

移除字典中的所有键值对。

```
Dictionary<string, int> students = new Dictionary<string, int>{ { "小明", 20 }, { "大明", 30 } };
students.Clear();
Console.WriteLine("字典中的键值对数量: " + students.Count);      //输出: 0
```

下面看一个相对综合的例子,其中除了演示如上几个方法,还演示了对字典中键的遍历、值的遍历以及键值同时遍历。

```
using System.Collections.Generic;
static void Main()
{
    //创建一个 Dictionary<string, int> 字典
    Dictionary<string, int> students = new Dictionary<string, int>();

    //添加键值对
    students.Add("小明", 20);
    students.Add("大明", 30);
    students.Add("老明", 35);

    //输出字典内容,也就是键值对的同时遍历
    Console.WriteLine("字典内容:");
    foreach (var kv in students)
        Console.WriteLine($"{kv.Key}: {kv.Value}");

    //修改
    students["小明"]=25;

    //再次输出字典内容
    Console.WriteLine("字典内容:");
    foreach (var kv in students)
        Console.WriteLine($"{kv.Key}: {kv.Value}");

    //遍历字典的键
    Console.WriteLine("字典的键:");
    foreach (string key in students.Keys)
        Console.WriteLine(key);

    //遍历字典的值
    Console.WriteLine("字典的值:");
    foreach (int value in students.Values)
```

```
        Console.WriteLine(value);

    //移除键值对
    bool flag = students.Remove("大明");
    Console.WriteLine("是否成功移除 大明: " + flag);              //输出: True

    //检查键是否存在
    flag = students.ContainsKey("小明");
    Console.WriteLine("字典中是否包含 小明: " + flag);            //输出: True

    //检查值是否存在
    flag = students.ContainsValue(35);
    Console.WriteLine("字典中是否包含年龄 35: " + flag);          //输出: True

    //尝试获取值
    if (students.TryGetValue("老明", out int age))
        Console.WriteLine("老明 的年龄是: " + age);               //输出: 老明 的年龄是: 35
    else
        Console.WriteLine("未找到 老明");

    //清空字典
    students.Clear();
    Console.WriteLine("清空后键值对数量: " + students.Count);  //输出: 0
}
```

运行结果如图 5-5 所示。

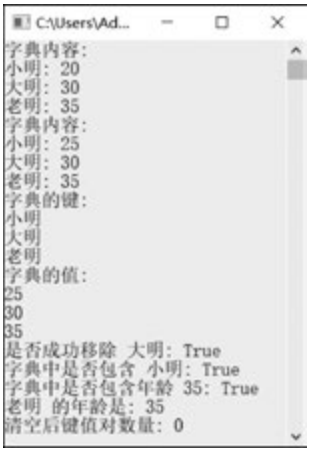

图 5-5 Dictionary<K,V>示例运行结果

5.4 拓展知识点

5.4.1 Windows Media Player 组件的引用

Windows Media Player 是一个常用于音视频等媒体文件播放的组件,功能强大,使用简单。在系统中有很多类似的组件,都可以通过引用而简化开发。因此,此处特地将完整的过程进行呈现。

首先打开需要引用相关组件或控件的项目，然后打开其工具箱，在工具箱空白处单击右键，单击“选择项”命令，如图 5-6 所示。

图 5-6　添加引用之“选择项”

在弹出的对话框中，根据需要选择相关选项，如相关组件。本例选择 Windows Media Player，如图 5-7 所示。

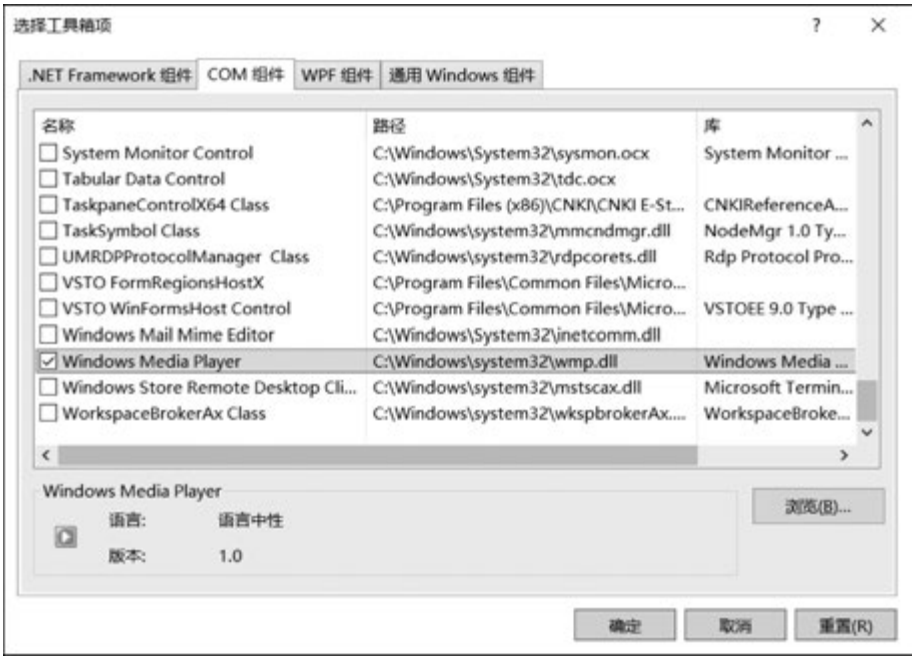

图 5-7　添加引用之选择组件

单击“确定”按钮即完成组件的添加，并显示于工具箱中，如图 5-8 所示。

此后，就可以像工具箱中其他的组件或控件一样使用了。当将该组件添加到窗体中时，将会自动添加两个引用，如图 5-9 所示。

图 5-8　添加引用之添加至工具箱

图 5-9　组件拖到窗体后自动添加的两个引用

此时，在编写代码时，在代码顶部添加如下引用。

```
using AxWMPLib;
using WMPLib;
```

5.4.2　使用 Windows Media Player 实现音频播放控制

Windows Media Player 控件提供了丰富的多媒体处理功能，使得开发者可以轻松地在应用程序中集成音频和视频播放功能。不过，本案例仅对其做最为基本的介绍。具体而言，典型的如实现音频文件的打开、播放、暂停、停止、音量调节以及播放进度等功能。

如下讲解均假设 Windows Media Player 控件被命名为 Player。

注意：在仅播放音频的情况下，一般应将 Windows Media Player 的 Visible 属性设置为 false。

1. 音频播放、暂停与停止

```
Player.URL = sFile;
Player.Ctlcontrols.play();                    //播放
Player.Ctlcontrols.pause();                   //暂停
Player.Ctlcontrols.stop();                    //停止
```

2. 音频媒体当前进度和总时长获取

```
double dDuration = Player.currentMedia.duration;          //总时长
doubule dCurrent = Player.Ctlcontrols.currentPosition;    //当前播放位置
```

3. 音量控制

音量控制是指改变音频的播放音量。可以通过调整 Windows Media Player 控件的 Volume 属性实现音量的调整。

```
//获取当前音量
int volume = Player.settings.volume;
//设置音量
Player.settings.volume =60;       //60%音量
```

5.4.3 LRC 歌词格式

LRC 是一种用于存储和显示同步歌词的文本文件格式。它通过在歌词文本中嵌入时间标签(timestamps),实现了歌词与音频播放的精确同步。LRC 文件通常与音频文件(如 MP3、WAV)同名,并存放在同一目录下。LRC 格式广泛应用于音乐播放器、卡拉 OK 系统以及各种多媒体应用中,为用户提供歌词滚动显示的功能。

LRC 文件由以下两部分组成。

- 元数据标签(可选):如[ti:标题],[ar:艺术家],[al:专辑],[by:制作人],[offset:偏移量]等。
- 歌词文本:每行歌词前有一个或多个时间标签,表示该行歌词的开始时间。

下面分别介绍。

1. 时间标签及歌词

时间标签用于表示歌词中每个单词或每行歌词的开始时间。常见的时间标签格式有两种,如下仅介绍简单的时间标签。

```
格式:[mm:ss.xx]
```

其中:

- mm:分钟(00～59)。
- ss:秒(00～59)。
- xx:百分之一秒(00～99)。

示例:

```
[00:12.00] 歌词第一行
[00:17.20] 歌词第二行
```

2. 元数据标签

LRC 文件可以包含多个元数据标签,用于描述歌词的元信息。常见的元数据标签如下。

- [ti:标题]:歌曲标题。
- [ar:艺术家]:艺术家名称。
- [al:专辑]:专辑名称。
- [by:制作人]:歌词制作人。

• [offset:偏移量]：时间偏移量(以 ms 为单位),用于调整歌词与音频的同步。

示例如下。

```
[ti:歌曲标题]
[ar:艺术家名称]
[al:专辑名称]
[by:制作人]
[offset:100]
```

例如,如下是经典《西游记》片尾曲《敢问路在何方》的 LRC 歌词。

```
[00:00.00]敢问路在何方
[00:08.00]作词：阎肃
[00:15.00]作曲：许镜清
[00:20.00]
[00:22.50]你挑着担 我牵着马
[00:29.70]迎来日出 送走晚霞
[00:37.10]踏平坎坷 成大道
[00:44.30]斗罢艰险 又出发 又出发
[00:51.80]啦啦……啦啦啦啦啦啦啦啦
[01:06.60]一番番春秋冬夏
[01:13.80]一场场酸甜苦辣
[01:21.10]敢问路在何方 路在脚下
[01:38.60]
[01:41.10]你挑着担 我牵着马
[01:48.30]翻山涉水 两肩霜花
[01:55.60]风云雷电 任叱咤
[02:02.90]一路豪歌 向天涯 向天涯
[02:10.40]啦啦……啦啦啦啦啦啦啦啦
[02:25.20]一番番春秋冬夏
[02:32.40]一场场酸甜苦辣
[02:39.70]敢问路在何方 路在脚下
[02:57.20]
[02:59.70]啦啦……啦啦啦啦啦啦啦啦
[03:14.50]一番番春秋冬夏
[03:21.70]一场场酸甜苦辣
[03:29.00]敢问路在何方 路在脚下
[03:46.50]敢问路在何方 路在脚下
```

由于 LRC 歌词是纯文本,因此可以自行解析。

另外,由于 LRC 文本程序明显的规律,因此一般都可以采用正则表达式进行解析。不过本案例中采用字符串函数来解析。此处不再赘述。

5.5 案例分析

该案例主要分为音频播放、进度显示、歌词解析、音频与歌词文件的自动匹配、歌词同步,简述如下。

1. 音频播放

• 使用 Windows Media Player 控件实现音频播放。

- 提供播放、暂停、停止等基本控制功能。

2. 进度显示

- 使用 ProgressBar 控件显示播放进度。
- 通过 Timer 组件定时更新 ProgressBar 控件的值，实现进度的实时显示。

假设做某事的总量为 X，当前的进度是做到了 x，因为 ProgressBar 控件的默认最小值(Minimum)为 0，最大值(Maximum)为 100，代表进度的是属性 Value，因此很容易计算出 Value＝100x/X，不过需要注意整除的问题。

此外，还有一种极其简单的处理方式，即将 ProgressBar 的 Maximum 属性赋值为任务总量 X，则其属性 Value＝x。

通过 Timer 组件定时更新 ProgressBar 控件的值，实现进度的实时显示。

3. 歌词解析

- 使用字符串处理函数解析 LRC 歌词文件，提取时间和歌词信息。
- 使用 List＜T＞存储解析后的时间和歌词。
- 使用 Timer 组件定时获取当前播放时间点，查找对应的歌词并显示，实现歌词与音频的同步显示。

LRC 歌词本质上就是一个文本文件，且具备固定的格式，因此对歌词文件的解析即通过字符串函数对字符串进行操作，从其中抽取指定的信息。LRC 歌词每一行的格式大致如下。

```
[01:02.50]《C＃程序设计》教学示范
```

需要从其中分析出时间和歌词两项信息。这可以从多种途径来获得。例如：

思路 1：先将整个歌词按换行符分割，所得每一行即如上述示例歌词所示。在上述歌词中，格式完全固定，即第 2、3 位为分钟部分，第 5～9 位为秒数部分，从第 11 位开始为歌词部分。而从字符串中取出指定位置的内容使用 Substring 函数即可。

思路 2：一次切割到位。利用字符'[',':',']','\n'来进行切割(Split)，将得到形如“分钟 秒 歌词 分钟 秒 歌词…”的数组，很明显，按 3 为递增周期即可获取时间和歌词了。

另外，本案例中用来存储时间和歌词是通过 List＜T＞来完成的。

4. 音频与歌词文件的自动匹配

- 使用 Path 类获取音频文件的目录和文件名。
- 根据音频文件名自动查找对应的歌词文件，支持歌词文件与音频文件位于同一目录或固定目录的情况。

具体而言，可以通过其 GetDirectoryName 方法获取目录；通过 GetExtension 获取文件扩展名；通过 GetFileName 获取包括扩展名的文件名；通过 GetFileNameWithoutExtension 获取不带后缀的文件名等。

一般歌词要么和音频文件放置于同一个文件夹，要么放置于一个固定的文件夹，且和对应的音频文件同名。无论哪种情况，只要将音频文件的扩展名替换为歌词文件的扩展名，然后结合路径(文件夹)信息，即可获得相应的歌词文件全路径。例如，若音频的全路径为 C:\Music\娃哈哈.mp3，歌词集中存放文件夹为 C:\Lyrics\，则对应歌词文件可能为 C:\Music\娃哈哈.lrc 或 C:\Lyrics\娃哈哈.lrc。

5. 歌词同步

本项目使用的方式是：在 Timer_Tick 中通过 Windows Media Player 的 Ctlcontrols.currentPosition 获取当前播放的时间点，然后通过该时间去分析获取当前时间点应该显示的歌词，将其显示出来，从而实现歌词同步。

注意：本项目为了简单，不过多涉及 Windows Media Player 控件，而将重心集中在书本相关知识点方面，因此并未涉及该控件的诸多事件如 PlayStateChange 等，而是使用了不太常规的但对通过教材来学习的读者来说更易理解的方式来实现。

5.6 控　　件

相关控件及属性设置如表 5-14 所示。

表 5-14　相关控件及属性设置

名称及类型	属　　性	属　性　值	作　　用
Form1	Text	音频播放器	
	StartPosition	CenterScreen	启动时屏幕居中
label1	AutoSize	False	用于显示歌词
	TextAlign	MiddleCenter	
button1	Text	打开	选择待播放的文件
progressBar1			用于显示播放进度
trackBar1	Maximum	100	最大音量
	TickStyle	None	不需要刻度
openFileDialog1			用于选择音频文件
timer1			实现进度和歌词同步
Player(Windows Media Player)	Visible	False	播放控件

5.7 核 心 代 码

该项目的核心在于实现对歌词的解析，解析类 LRCParser 的代码如下。

```
class LRCParser
{
    private List<double> listTimes = new List<double>();    //存储分解后的时间
    private List<string> listLRCs = new List<string>();      //存储分解后的歌词

    /// <summary>
    /// 分析传入的歌词,将其分解为时间和对应的歌词两个列表
    /// </summary>
    /// <param name="sLRC">歌词</param>
    public LRCParser(string sLRC)
    {
        //歌词示例：[00:00.00]《C#程序设计》教学示范
        string[] sLines = sLRC.Split('\n');
        string sLine = null;
```

```
            string sMin = null, sSec = null, sL = null;
            double dTemp = 0;
            for (int i = 0; i < sLines.Length; i++)          //逐行处理
            {
                sLine = sLines[i];
                sMin = sLine.Substring(1, 2);                //获取分钟部分
                sSec = sLine.Substring(4, 2);                //获取秒数部分
                sL = sLine.Substring(10);                    //获取歌词
                dTemp = 60 * Convert.ToInt32(sMin) + Convert.ToDouble(sSec);
                listTimes.Add(dTemp);
                listLRCs.Add(sL);
            }
        }

        /// <summary>
        /// 根据传入的时间,获取该时间点应该显示的歌词
        /// </summary>
        /// <param name="dTime">所传入的时间</param>
        /// <returns>返回值为所传入时间对应的歌词</returns>
        public string GetLrcByTime(double dTime)
        {
            string sRet = null;
            for (int i = listTimes.Count - 1; i >= 0; i--)
            {
                if (dTime > listTimes[i])
                {
                    sRet = listLRCs[i];
                    break;
                }
            }
            return sRet;
        }
    }
```

下面是窗体中的代码,首先是变量声明,代码如下。

```
LRCParser parser = null;
```

“打开”按钮用于选择待播放的文件,然后播放,其核心代码如下。

```
private void Button1_Click(object sender, EventArgs e)
{
    //分别代表待播放文件、歌词文件、音频文件扩展名、歌词
    string sFile = null,sLrcFile=null,sExt=null,sLrc=null;
    DialogResult dlgResult = openFileDialog1.ShowDialog();
    if (dlgResult == DialogResult.OK)
    {
        sFile = openFileDialog1.FileName;                    //待播放的文件
        //根据音频文件自动处理同目录下的歌词文件
        sExt = Path.GetExtension(sFile);                     //获取当前音频文件扩展名
        sLrcFile = sFile.Replace(sExt, ".lrc");
```

```
            if (File.Exists(sLrcFile))
            {
                sLrc = File.ReadAllText(sLrcFile, Encoding.Default);   //注意编码
                parser = new LRCParser(sLrc);                          //解析歌词
                Player.URL = sFile;
                Player.Ctlcontrols.play();                             //开始播放
                timer1.Interval = 50;                                  //开启定时器
                timer1.Enabled = true;
            }
            else
            {
                label1.Text = "没有匹配的歌词文件";
            }
        }
    }
```

上面只是实现音频文件播放，歌词的同步是靠定时器实现的，代码如下。

```
private void Timer1_Tick(object sender, EventArgs e)
{
    label1.Text = parser.GetLrcByTime(Player.Ctlcontrols.currentPosition);
    progressBar1.Maximum = Convert.ToInt32(Player.currentMedia.duration);
    progressBar1.Value = Convert.ToInt32(Player.Ctlcontrols.currentPosition);
}
```

音量调整代码如下。

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    Player.settings.volume = trackBar1.Value;
}
```

程序运行效果如图 5-10 所示。

图 5-10　音频播放器运行效果

5.8　思 考 拓 展

(1) 实现 5.5 节中的另外几种思路，包括 ProgressBar 控件的进度问题，以及歌词的解析思路。

(2) 本项目中使用的 LRC 歌词较为理想化，即没有 LRC 歌词前不带时间戳的那些行，

也不考虑某些带循环特征的歌词，不过和本例相比并无本质的难度差异，请自行实现。

(3) 在上述案例的基础上，实现列表播放，如顺序播放、随机播放等。

(4) 利用 Windows Media Player 控件的诸多事件如 PlayStateChange 等以及其属性 currentPlaylist 等实现一个列表播放的播放器。

5.9 总　　结

通过本案例，读者不仅学习了 Windows Forms 应用程序中媒体播放功能的实现，还学习了 ProgressBar 控件、字符串处理函数、Path 类、泛型集合 List < T >、Timer 组件和对话框的使用。通过解析 LRC 歌词文件，读者能够实现歌词与音频的同步显示。此外，通过思考和拓展，读者可以在现有基础上进一步优化和扩展功能，提升音频播放器的功能性和用户体验。