第 2 章　C# 3.0 程序设计基础

在第一章里，了解了 ASP.NET 3.5 的特性和一些基本的.NET Framework 知识，不过如果要深入到 ASP.NET 3.5 应用程序开发，需要对开发语言有更加深入的了解。而在.NET 平台上，微软主推的编程语言就是 C#，本章将会从 C#的语法、结构和特性来讲解，以便读者能够深入的了解 C#程序设计。

2.1　C#程序

C#程序有自己的程序结构。C#编程语言类似 C++/Java 等面向对象编程语言，同样需要编写类、创建对象等。但是 C#依旧有与其他面向对象编程语言不同的特性，使用这些特性能够快速的正确的编写 C#宿主语言的应用程序，如 ASP.NET、WinForm 等。

2.1.1　C#程序的结构

在开始学习和编写 C#代码之前，首先应该了解 C#编程语言的结构，下列代码说明了 C#应用程序的基本结构。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                          //使用命名空间
namespace mycsharp                                          //程序代码命名空间
{
    class Program                                           //应用程序主类
    {
        static void Main(string[] args)                     //入口方法
        {
            Console.WriteLine("Hello World");               //输出 Hello World
            Console.ReadKey();                              //等待用户输入
        }
    }
}
```

其中，using 关键字的用途是引用微软的.NET 框架中现有的类库资源，该关键字出现在应用程序代码的开头，并使用在 cs 为后缀的文件中使用。using 关键字通常情况下会出现几次，其目的是引用类库中的各种资源，这些资源不仅包括代码中的 System,System.Collections.Generic,Linq，还包括其他.NET 框架的资源。

System 命名空间提供了构建应用程序所需的各种系统功能，例如 LINQ 的类库包括了构建 LINQ 应用程序的各种类库资源。.NET 中提供大量的命名空间，以便开发人员能够使用现有的类库进行应用程序的开发。同时，在代码中也可以看到在其中包含一个 mycsharp 的一个命名空间，示例代码如 namespace mycsharp。在当前程序中声明该命名空间，可以在其他的程序中引用这个命名空间，并使用此命名空间下的类和方法。

另外，Program 是一个类名。在 C#或其他的任何面向对象语言中（如 JAVA、C++）都需要编写类，类用于创建对象。在上述代码中，Program 是一个类的名称。

方法是用于描述类的行为。在上述示例第 9 行中，static void Main 是一个全局静态方法，它指示编译器从此处开始执行程序，相当于程序的入口，程序运行的时候会执行 Main 方法作为入口。在 C# Windows 编程中，大部分的应用程序必须在其组成程序的其中一个类中包含 Main 方法。

语句就是在 C#应用程序中包含的指令，通过使用分号进行分割，编译器通过分号来区分它们。一些编程语言只允许一行放置一条语句，但是 C#允许放置多个语句，也可以将一个语句拆分成多行。虽然 C#编译器支持这样的特性，但是还是推荐使用一行放置一个语句的，这样不仅提高了可读性，也便于书写。

括号“{”和“}”用来标识程序中代码的范围，如上述代码中 Main 方法囊括了 Main 方法的语句，Program 类囊括了类的方法，而 namespace mycsharp 命名空间囊括了此命名空间里的所有类。值得注意的是，Visual Studio 2008 为开发人员在编写程序的时候提供了诸多的智能提示，在完成一个类或一个变量时，系统会自动补全，而当鼠标放到一个大括号上的时候，编译器会指示开发人员此括号的范围，如图 2-1 所示。

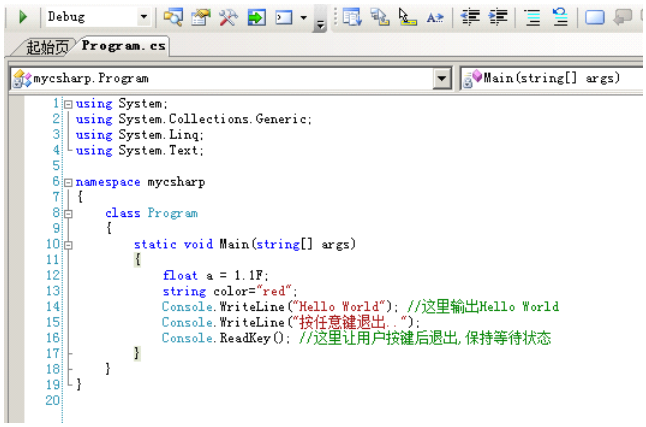

图 2-1　外围元素被标记

注意：在一个类内的所有方法都是独立的方法,所以每个大括号后面都不需要分号,同样对于命名空间里的所有类也是一样。

2.1.2　C# IDE 的代码设置

代码格式也是程序设计中一个非常重要的组成环节，他可以帮助用户组织代码和改进代码，也让代码具有可读性。具有良好可读性的代码能够让更多的开发人员更加轻松的了解和认知代码。按照约定的格式书写代码是一个非常良好的习惯，下面的代码示例说明了应用缩进、大小写敏感、空白区和注释等格式的原则。

```
using System;
using System.Collections.Generic;
using System.Linq;                                          //使用 LINQ 命名空间
using System.Text;
namespace mycsharp                                          //声明命名空间
{
    class Program                                           //主程序类
    {
```

```
        static void Main(string[] args)                                    //静态方法
        {
            Console.WriteLine("Hello World");                              //这里输出 Hello World
            Console.WriteLine("按任意键退出.."); Console.ReadKey();    //这里让用户按键后退出,保持等待状态
        }
    }
}
```

1．缩进

缩进可以帮助开发人员阅读代码，同样能够给开发人员带来层次感。读者可以从以上代码看出这一串代码让人能够很好的分辨区域，非常方便的就能找到 Main 方法的代码区域，这是因为括号都是有层次的。

缩进让代码保持优雅，同一语句块中的语句应该缩进到同一层次，这是一个非常重要的约定，因为它直接影响到代码的可读性。虽然缩进不是必须的，同样也没有编译器强制，但是为了在不同人员的开发中能够进行良好的协调，这是一个值得去遵守的约定。

2．大小写敏感

C#是一种对大小写敏感的编程语言。可能 php 等其他语言的开发人员不太适应大小写敏感，但是在 C#中，其语法规则的确是对字符串中字母的大小写敏感的，例如“C Sharp”、“c Sharp”、“c sHaRp”都是不同的字符串，在编程中应当注意。

3．空白

C#编译器会忽略到空白。使用空白能够改善代码的格式，提高代码的可读性。但是值得注意的是，编译器不对引号内的任何空白做忽略，在引号内的空格作为字符串存在。

4．注释

在 C/C++里，编译器支持开发人员编写注释，以便开发人员能够方便的阅读代码。当然，在 C#里也一样继承了这个良好的习惯。之所以这里说的是习惯，是因为编写注释同缩进一样，没有人强迫要编写注释，但是良好的注释习惯能够让代码更加优雅和可读，谁也不希望自己的代码在某一天过后自己也不认识了。

注释的写法是以符号“/*”开始,并以符号“*/”结束，这样能够让开发人员更加轻松的了解代码的作用，同时，也可以使用符号“//”双斜线来写注释，但是这样的注释是单行的，示例代码如下所示。

```
            /*
             * 多行注释
             * 本例演示了在程序中写注释的方法
               在注释内也可以不要开头的*号
             */
            //单行注释,一般对单个语句进行注释
```

5．布局风格

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");                              //这里输出 Hello World
        Console.WriteLine("按任意键退出.."); Console.ReadKey();    //这里让用户按键后退出,保持等待状态
    }
}
```

从以上代码可以看出，程序中使用了缩进、大小写敏感，空白区和注释等，但是这个代码风格依旧不是最好，可以修改代码让代码更加“好看”。这里能够将代码进行修正，修正后的示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");                    //这里输出 Hello World
        Console.WriteLine("按任意键退出..");
        Console.ReadKey();                                   //这里让用户按键后退出,保持等待状态
    }
}
```

这种布局风格让开发人员感觉到耳目一新，这样更能方便更多的开发人员阅读源代码。如果打开一千行或更多代码量的源文件时，其编码格式都是标准的风格的话，不管是谁再接手去阅读，都能尽快上手。不仅如此，在软件开发当中，应该规定好每个人都使用同样的布局风格，让团队能够协调运作。

2.2 变量

在任何编程语言中，无论是传统的面向过程还是面向对象都必须使用变量。因此，变量都有自己的数据类型，在使用变量的时候，必须使用相同的数据类型进行运算。在程序的运行中，计算中临时存储的数据都必须用到变量，变量的值也会放置在内存当中，由计算机运算后再保存到变量中，由此可见，变量在任何的应用程序开发中都是非常基础也是非常重要的。同样，在C#中也需要变量对数据进行存储，本节将会介绍C#的基本语法、数据类型、变量、枚举等。

2.2.1 定义

要声明一个变量就需要为这个变量找到一个数据类型，在C#中，数据类型由.NET Framework 和 C#语言来决定，表 2-1 列举了一些预定义的数据类型。

表 2-1 预定义数据类型

预定义类型	定义	字节数
byte	0~255之间的整数	1
sbyte	-128~127之间的整数	1
short	-32768~32767之间的整数	2
ushort	0~65535之间的整数	2
int	-2147483648~2147483647之间的整数	4
uint	0~4294967259之间的整数	4
long	-9223372036854775808~9223372036854775807之间的整数	8
ulong	0~18445744073709551615之间的整数	8
bool	布尔值,true of false	1
float	单精度浮点值	4
double	双精度浮点值	8
decimal	精确的十进制值,有28个有效单位	12
object	其他所有类型的基类	N/A
char	0~65535之间的单个Unicode字符	2
string	任意长度的Unicode字符序列	N/A

一个简单的声明变量的代码如下所示：

```
int s;                                          //声明整型变量
float myfloat;                                  //声明浮点型变量
```

上述代码声明了一个整型的变量 s，同时也声明了一个单精度浮点型变量 myfloat。

2.2.2 值类型

这种类型的对象总是直接通过其值使用，不需要对它进行引用。基于值类型的变量直接包含值。并且，所有的 C#局部变量都需要初始化后才可以使用，值类型同样如此，初始化代码如下所示。

```
int s;                                          //声明整型变量
s = new int();                                  //声明整型变量
s = 3;                                          //初始化变量
```

上式等同于如下代码。

```
int s;                                          //声明整型变量
s = 3;                                          //初始化变量
```

所有的值类型均隐式的派生自 System.ValueType，并且值类型不能派生出新的类。值的类型不能为 null，但是可空类型允许将 null 值赋给值类型，在上面的代码中，程序通过默认的构造函数给为变量 s 初始化并赋值。

2.2.3 引用类型

引用类型的变量又称为对象，是可存储对实际数据的引用。常见的引用类型有 class、interface、delegate、object 和 string。多个引用变量可以附加于一个对象，而且某些引用可以不附加于任何对象，如果声明了一个引用类型的变量却不给他赋给任何对象，那么它的默认值就是 null。相比之下，值类型的值不能为 null。

2.3 变量规则

声明变量并不是随意声明的，变量的声明有自己的规则。在 C#中，应用程序包含许多关键字，包括 int 等是不能够声明为变量名的，如 int int 是不允许的，在进行变量的声明和定义时，需要注意变量名称是否与现有的关键字重名。

2.3.1 命名规则和命名习惯

命名规则就是给变量取名的一种规则，一般来说，命名规则就是为了让开发人员给变量或者命名空间取个好名，不仅要好记，还要说明一些特性。在 C#里面，有常用的一些命名的习惯如下。

- Pascal 大小写形式：所有单词的第一个字母大写，其他字母小写。
- Camel 大小写形式：除了第一个单词，所有单词的第一个字母大写，其他字母小写。

当然，在其他编程中，不同的开发人员可能遇到了一些不一样的命名规则和命名习惯，但是在 C#中，推荐使用常用的一些命名习惯，这样能保证代码的优雅性和可读性。同时，也应该避免使用相同名称的命名空间或与系统命名相同的变量，如以下代码所示：

```
string int;                                     //系统会提示出错
```

运行上述代码时系统会提示错误，因为字符串“int”是一个关键字，当使用关键字做变量名时，编

译器会混淆该变量是变量还是关键字，所以系统会提示错误。所以，在变量声明时应该避免变量名称与关键字重名，如果变量名称与关键字重名，编译器就会报错，C#中常用的关键字如表 2-2 所示：

表 2-2　不应使用的关键字名称

AddHandler	AddressOf	Alias	And	Ansi	As
Assembly	Auto	BitAnd	BitNot	BitOr	BitXor
Boolean	ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate	CDec
CDbl	Char	CInt	Class	CLng	CObj
Const	CShort	CSng	CStr	CType	Date
Decimal	Declare	Default	Delegate	Dim	Do
Double	Each	Else	ElseIf	End	Enum
Erase	Error	Event	Exit	ExternalSource	False
Finally	For	Friend	Function	Get	GetType
Goto	Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let	Lib
Like	Long	Loop	Me	Mod	Module
MustInherit	MustOverride	MyClass	Namespace	MyBase	New
Next	Not	Nothing	NotInheritable	NotOverridable	Object
On	Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property	Protected
Public	RaiseEvent	ReadOnly	ReDim	Region	REM
RemoveHandler	Resume	Return	Select	Set	Shadows
Shared	Short	Single	Static	Step	Stop
String	Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode	Until
Variant	When	While	With	WithEvents	WriteOnly
Xor	eval	extends	instanceof	package	var

注意：标识符、参数名、函数名都不需要使用缩写。如果要使用缩写，超过两个字符以上的缩写都应该使用 Camel 大写格式。

2.3.2　声明并初始化变量

在程序代码编写中，需要大量的使用变量和读取变量的值，所以需要声明一个变量来表示一个值。这个变量可能描述是一个人的年龄，也可能是一辆车的颜色。在声明了一个变量之后，就必须给这个变量一个值，只有在给变量值之后能够说明这个变量被初始化。

1．语法

声明变量的语法非常简单，即在数据类型之后编写变量名，如一个人的年龄（age）和一辆车的颜色（color），声明代码如下所示。

```
int age;                                    //声明一个叫 age 的整型变量,代表年龄
string color;                               //声明一个叫 color 的字符串变量,代表颜色
```

上述代码声明了一个整型变量 age 和一个字符串型变量 color，由于年龄的值不会小于 0 也不会大于 100，所以在声明时可以使用数字类型进行声明。

2．初始化变量

变量在声明后还需要初始化，例如“我年龄 21 岁，很年轻，我想买一辆红色的车”，那么就需要对相应的变量进行初始化，示例代码如下所示。

```
int age;                                    //声明一个叫 age 的整型变量,代表年龄
string color;                               //声明一个叫 color 的字符串变量,代表颜色
```

```
age = 21;                                    //声明始化，年龄 21 岁
color = "red";                               //声明始化，车的颜色为红色
```

上述代码也可以合并为一个步骤简化编程开发，示例代码如下所示。

```
int age=1;                                   //声明并初始化一个叫 age 的整型变量,代表年龄
string color="red";                          //声明初始化
```

3．赋值

在声明了一个变量之后，就可以给这个变量赋值了，但是当编写以下代码就会出错，示例代码如下 。

```
float a = 1.1;                               //错误的声明浮点类型变量
```

当运行了以上代码后会提示错误信息：不能隐式地将 Double 类型转换为“float”类型;请使用“F”后缀创建此类型。从错误中可以看出，将变量后缀增加一个“F”即可，示例代码如下所示。

```
float a = 1.1F;                              //正确的声明浮点类型变量
```

运行程序，程序就能够编译并运行了。这是因为若无其他指定，C#编译器将默认所有带小数点的数字都是 Double 类型，如果要声明成其他类型，可以通过后缀来指定数据类型，如表 2-3 将展示一些可用的后缀，并且后缀可用小写。

表 2-3　可用的后缀表

后缀	描述
U	无符号
L	长整型
UL	无符号长整型
F	浮点型
D	双精度浮点型
M	十进制
L	长整型

4．转义字符

在开发过程当中，如果需要将单引号或者双引号输出，或将单引号等字符作为字符串输出，就会发现在字符串中单引号或者双引号等字符是不能直接进行输出呈现。为了解决这一问题，于是引入了转义字符，常用的转义字符表如表 2-4 所示。

表 2-4　转义字符表

换码序列	字符名称
\’	单引号
\”	双引号
\\	反斜杠
\0	空字符
\a	警报符
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

若在应用程序开发过程中，需要在程序里的字符串中编写一个双引号并进行输出，可以使用转义字符进行输出，示例代码如下所示。

```
string str=”this is \” ”;                    //使用转义字符
```

6．设置断点

在 Visual Studio .NET 开发环境中，为用户提供了在开发应用程序时查看变量值的工具，只需要在查看的变量上设置断点，以调试模式运行应用程序，就可以在调试窗口中查看变量的值。在代码编辑窗口单机左边的空白处可直接设置断点。断点以红色圆点标识。也可以在调试菜单中单击【切换断点】按钮，或使用快捷键【F9】键来设置断点，如图 2-2 所示。

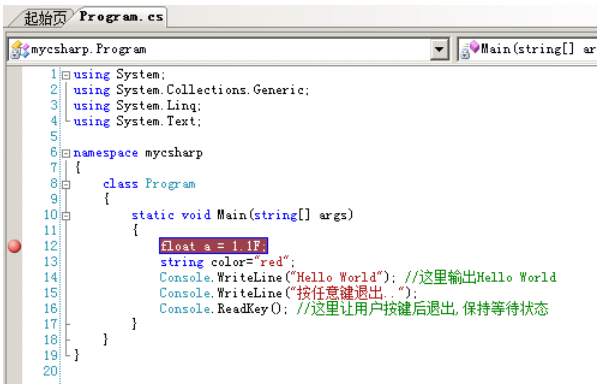

图 2-2　设置断点

按下【F5】键或在菜单栏中的调试菜单中单击【启动调试】按钮都可以运行程序。当程序开始运行，程序从 Main 入口运行并直至遇到断点，遇到断点后程序将停止运行，如图 2-3 所示。同时开发环境会高亮显示下一条即将执行的代码，同时调试查看窗口会显示，并呈现变量的当前值，如图 2-4 所示，。

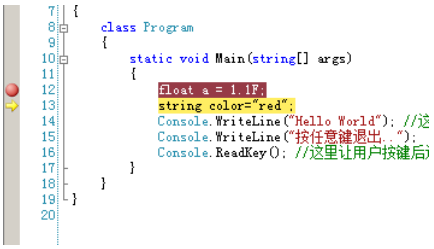

图 2-3　运行到断点,提示下一步执行的代码

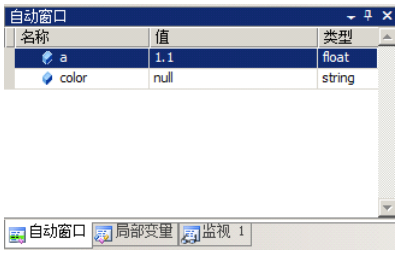

图 2-4　显示当前值

在调试完成后，可以通过快捷键【Shift+F5】停止调试，也可以在菜单栏中的【调试】菜单里的【停止调试】选项中停止应用程序的调试。如果需要继续执行，可以按下【F5】键或在调试菜单中选择【继续执行到下一个断点】选项进行执行。开发人员还可以通过使用快捷键【F10】，或在调试菜单中选择【逐过程】或【逐语句】每次只执行一条语句，方便对代码中变量变化的查看。

2.3.3　数组

数组是一个引用类型，开发人员能够声明数组并初始化数据进行相应的数组操作，数组是一种常用的数据存放方式。

1．数组的声明

数组的声明方法是在数据类型和变量名之间插入一组方括号，声明格式如下所示。

```
string[] groups;                                    //声明数组
```

以上语句声明了一个变量名为 groups 的数组，其数据类型为 string。声明了一个数组之后，并没有为此数组添加内容初始化，需要对数组初始化，才能使用数组。

2．数组的初始化

开发人员可以对数组进行显式的初始化，以便能够填充数组中的数据，初始化代码如下所示。

```
string[] groups={"asp.net","c#","control","mvc","wcf","wpf","linq"};          //初始化数组
```

值得注意的是，与平常的逻辑不同的是，数组的开始并不是 1，而是 0。以上初始化了 groups 数组，所以 groups[0]的值应该是“asp.net”而不是“c#”，相比之下，group[1]的值才应该是“c#”。

3．.NET 中数组的常用的属性和方法

在.NET 中，.NET 框架为开发人员提供了方便的方法来对数组进行运算，专注于逻辑处理的开发人员不需要手动实现对数组的操作。这些常用的方法如下所示。

- Length 方法用来获取数组中元素的个数。
- Reverse 方法用来反转数组中的元素，可以针对整个数组，或数组的一部分进行操作。
- Clone 方法用来复制一个数组。

对于数组的操作，可以使用相应的方法进行数据的遍历、查询和反转，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                                    //声明文本命名空间
namespace myArray                                                     //主应用程序类
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] groups={"asp.net","c#","control","mvc","wcf","wpf","linq"};          //初始化一个数组
            int count = groups.Length;                                //获取数组的长度
            Console.WriteLine("----------------------数组长度------------------------");
            Console.WriteLine(count.ToString());                      //输出数组的长度
            Console.WriteLine("----------------------原数组元素值------------------------");
            for (int i = 0; i < count; i++)                           //遍历输出数组元素
            {
                Console.WriteLine(groups[i]);                         //输出数组中的元素
            };
        }
    }
}
```

按 F5 运行后运行结果如图 2-5 所示。

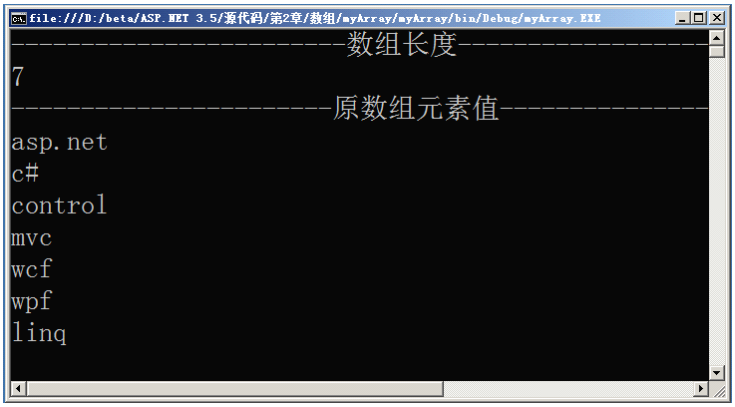

图 2-5　数组运行结果

从上述结果中可以看出，程序遍历了数组并将数组的内容全部输出。在进行数组中的内容输出时，需要使用循环语句进行输出数组的遍历和输出，循环语句的用法会在后面讲解。

2.3.4 声明并初始化字符串

字符串是计算机应用程序开发中常用的变量，在文本输出、字符串索引、字符串排序中都需要使用字符串。

1. 声明及初始化字符串

字符串类型（string）是程序开发中最常见的数据类型，如上一小节声明的数组中的任意一个元素都是一个字符串。由于数组也是有其数据类型的，所以声明的数组是一个字符串型的数组。字符串的声明方式和其他的数据类型声明方式相同，字符串变量的值必须在“”双引号之间，示例代码如下所示。

```
string str="Hello World!";                                    //声明字符串
```

在 2.3.2 中讲解了转义字符，当开发人员试图在字符串中间输入一些特殊符号的时候，会发现编译器报错，示例代码如下所示。

```
string str="Hello "World!";
```

在 Visual Studio 2008 中编写上述代码，会发现褐色的字符串被分开了，并且编译器报错“常量中有换行符”，因为字符串中的“ “”符号被当成是字符串的结束符号。为了解决这个问题，就需要用到转义字符。示例代码如下所示。

```
string str="Hello \"World!";                                  //使用转义字符
```

编译并运行，运行结果如图 2-6 所示。

图 2-6　使用转义字符初始化字符串

在程序中的开发中，经常需要引用和打开某个文件，打开文件的操作必须要引用文件夹的地址。例如要打开我的文档里的内容，就必须在地址栏敲击 D:\Users\Administrator\Documents，在编写程序的时候，“\”字符却无法编写在字符串中，同样也需要转义字符，示例代码如下所示。

```
string str="D:\\Users\\Administrator\\Documents";              //使用转义字符
```

编译并运行，运行结果如图 2-7 所示。

图 2-7　使用转义字符初始化字符串

2. 使用逐字符串

如果字符串初始化为逐字符串，编译器会严格的按照原有的样式输出，无论是转义字符中的换行符还是制表符，都会按照原样输出。逐字符串的声明只需要在双引号前加上字符“@”即可，示例代码如下所示。

```
string str=@"文件地址:D:\Users\Administrator\Documents \t";    //逐字符串
```

编译并运行上述代码，运行结果如图 2-8 所示。

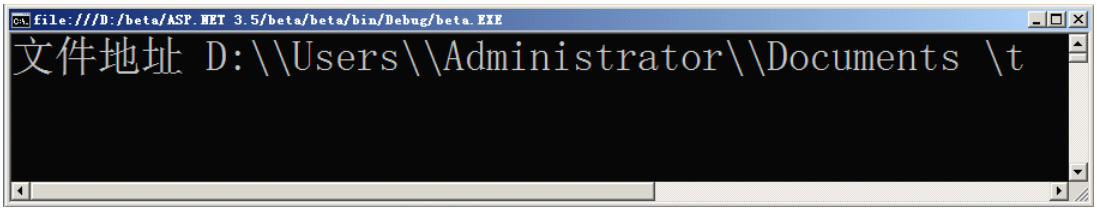

图 2-8　使用逐字符串

但是对于双引号而言，逐字符串依旧无法正确进行输出。若要使用逐字符串进行双引号的输出，则必须使用引号进行编写以便正确的输出双引号，示例代码如下所示。

```
string str=@"""";                                    //输出双引号
```

编译并运行，运行结果如图 2-9 所示。

图 2-9　逐字符串输出双引号

3．字符串格式化

在字符串操作时，很多地方需要用到字符串格式化，使用 Console.WriteLine 方法就能够实现字符串格式化，字符串格式化代码如下所示。

```
string str = "Guojing";                              //声明字符串
Console.WriteLine("Hi! Myname is {0},I love C#",str);  //字符串格式化输出
Console.ReadKey();                                   //等待用户按键
```

运行上述代码，其结果如图 2-10 所示。

图 2-10　字符串格式化

可以从运行结果看出，Console.WriteLine 方法中，前一个传递的参数中的{0}被后一个传递的参数 str 替换。例子中的“{0}”被称为占位符，用于标识一个参数，括号中的数字指定了参数的索引。同时，输出方法也可以格式化多个字符串，示例代码如下所示。

```
string str = "Guojing";
string str2 = "C#";
Console.WriteLine("Hi! Myname is {0},I love {1}",str,str2);   //格式化多个字符串输出
```

运行结果如图 2-11 所示。

图 2-11　多个占位符格式化字符串

2.3.5　操作字符串

在 C#中，为字符串提供了快捷和方便的操作，使用 C#提供的类能够进行字符串的比较、字符串的连接、字符串的拆分等操作，方便了开发人员进行字符串的操作。

1．比较字符串

如果需要比较字符串，有两种方式，一种是值比较，一种是引用比较。值比较可以直接使用运算符“==”进行比较，示例代码如下所示。

```
string str = "Guojing";                                    //声明字符串
string str2 = "C#";                                        //声明字符串
if (str == str2)                                           //使用“==”比较字符串
{
    Console.WriteLine("字符串相等");                        //输出不相等信息
}
else
{
    Console.WriteLine("字符串不相等");                      //输出相等信息
}
```

当判断两个字符串是否指向同一个对象时，可以使用 Compare 方法判定两个字符串是否指向同一个对象，示例代码如下所示。

```
string str = "Guojing";                                    //声明字符串
string str2 = "C#";                                        //声明字符串
if (str.CompareTo(str2) > 0)                               //使用 Compare 比较字符串
{
    Console.WriteLine("字符不串相等");                      //输出不相等信息
}
else
{
    Console.WriteLine("字符串相等");                        //输出相等信息
}
```

编译上述代码并运行，其结果如图 2-12 所示。

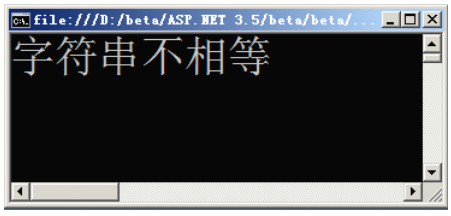

图 2-12　比较字符串

在上述代码运行后，如果字符串不相等，则输出“字符串不相等”字符，否则会输出“字符串相等”。

2．连接字符串

当一个字符串被创建，对字符串的操作的方法实际上是对字符串对象的操作。其返回的也是新的字符串对象，与 int 等数据类型一样，字符串也可以使用符号“+”进行连接，代码如下所示。

```
string str = "Guojing is A C# ";                                    //声明字符串
```

```
string str2 = "Programmer";                                    //声明字符串
Console.WriteLine(str+str2);                                   //连接字符串
```

在上述例子中，声明并初始化两个字符串型变量 str 和 str2，并输出 str+str2 的结果，运行结果如图2-13所示。

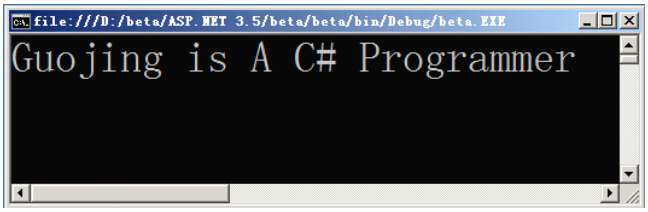

图 2-13　字符串连接

同样的，str 和 str2 也可以为新的字符串赋值，代码如下所示。

```
string mystr = str + str2;                                     //连接字符串
Console.WriteLine(mystr);                                      //输出字符串
```

上述代码运行结果同样如图 2-13 所示，这里就不再运行演示。

注意：在上述代码中，mystr 被初始化 str 和 str2 的“和”，但是在运算过程当中，str 和 str2 的值都没有被改变。

3．拆分字符串

能够连接一个字符串，同样也可以拆分一个字符串。.NET Framework 提供了若干方法供拆分字符串，示例代码如下所示。

```
string str = "Guojing is A C# Programmer";                     //声明字符串
Console.WriteLine(str.IndexOf("is").ToString());               //拆分字符串
Console.ReadKey();
```

编译运行后，可以看到返回的结果是 8，说明 is 是字符串从开始第 8 位才找到 is，若搜索不到查询的字符串，则返回-1。当字符串拆分成子字符串之后，可以通过 Split 方法对字符串进行分割，代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang";     //初始化字符串
string[] p = str.Split(',');                                   //使用 Split 方法分割并存入数组
for (int i = 0; i < p.Length; i++)                             //遍历显示
{
    Console.WriteLine(p[i]);                                   //输出字符串
}
```

上述代码第一句声明并初始化了一个字符串，第二句使用 Split 函数按照逗号来分割字符串，并存入到数组 p 内，然后遍历显示数组元素。

注意：使用 Split 函数的时候，通常情况下只能使用字符对字符串进行分割，所以使用的是单引号。

4．更改字符串大小写

在.NET 中，系统为开发人员提供了将字符串更改为大写或小写的方法，这两个方法分别为 ToUpper() 和 ToLower()。使用该方法能够进行字符串的大小写转换，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang";     //声明字符串
Console.WriteLine(str.ToUpper());                              //转换成大写
Console.WriteLine(str.ToLower());                              //转换成小写
Console.ReadKey();                                             //等待用户输入
```

5．常用的字符串操作

在 C#软件开发过程，字符串是使用率最高的数据类型之一，开发人员往往需要对字符串进行大量的操作。这里介绍一些经常使用的字符串操作如判断字符串是否为空，替换字符串中相应的字符等等。判断字符串是否为空会经常在程序中使用，以保证用户输入的完整性，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang";   //声明字符串
if (String.IsNullOrEmpty(str))                              //使用 String 类的静态方法
{
    Console.WriteLine("字符串为空");                         //输出字符串为空的信息
}
else
{
    Console.WriteLine("字符串不为空");                       //输出字符串不为空的信息
}
```

当需要对字符串执行替换操作时，可以使用 Replace 方法进行字符串的替换，示例代码如下所示。

```
string str = "BeiJing,Shanghai,GuangZhou,WuHan,ShenYang";   //声明字符串
str = str.Replace("ShenYang", "ShanXi");                    //使用 Replace 方法
Console.WriteLine(str);                                     //输出字符串
```

大多数应用程序都是对字符串进行操作，这里简单的介绍几种常用的字符串的操作，熟练掌握字符串的操作对应用程序开发有很大的好处。

2.3.6 创建和使用常量

常量是一般在程序开发当中不经常更改的变量，如π值、税率或者是数组的长度等。使用常量一般能够让代码更具可读性、更加健壮、便于维护。在程序开发当中，好的常量使用技巧对程序开发和维护都有好的影响，示例代码如下所示。

```
const double pi=3.1415926;                                  //常量 pi, π
static void Main(string[] args)                             //程序入口方法
{
    double r=2;                                             //声明 double 类型常量
    double round = 2 * pi * r * r;                          //使用常量
    Console.WriteLine(round.ToString());                    //输出变量值
    Console.ReadKey();                                      //等待用户输入
}
```

上述代码非常简单，就是计算一个圆的圆周率。当代码非常长的时候，程序也会非常干练，容易阅读，如果在程序中出现了以下代码，也能够理解该表达式的作用。示例代码如以下所示。

```
double Perimeter = 2 * pi * r;                              //使用常量
```

就算是其他的开发人员阅读到上述代码，也能够轻易的了解该语句的作用就是求圆的周长，因为在前面定义了常量 pi=3.1415926；当程序中用到这个变量的时候，立刻就能够知道程序的作用。声明变量的方法，只需要在普通的变量格式前加上 const 关键字即可，声明代码如下所示。

```
const double pi=3.1415926;                                  //声明 const 变量
const int max = 500;                                        //声明 const 变量
const long kilometer = 1000;                                //声明 const 变量
```

使用 const 声明的变量能够在程序中使用，但是值得注意的是，使用 const 声明的变量不能够在后面的代码中对该变量进行重新赋值。

注意：使用 const 声明的变量如果在后面的代码中进行重新赋值或更改，则编译器会提示错误。const 修饰符通常用于不常更改的变量的修饰。

2.3.7 创建并使用枚举

枚举类型是一组已命名的常量，它是一种用户自定义类型，开发人员可以自行创建枚举类型，声明枚举变量并初始化。枚举变量和普通的变量相比，确保了只将预定的值赋予变量，让代码更加容易维护 。在编写代码的时候，允许以简单容易辨认的名字作为变量名，使代码更具有可读性。同时，如果开发人员声明枚举变量，Visual Studio 2008 还能够提供的智能感知功能够让代码更加方便的输入。

1．声明及初始化枚举

如果需要创建枚举类型，就需要使用 enum 关键字，指定一个类型名称，如 int 等，然后列举出枚举可以使用的值，示例代码如下所示。

```
enum color {red,yellow,green,blue};                    //声明枚举
```

上述代码创建了一个 color 类型，然后声明了这个类型的变量，并使用枚举成员赋值，示例代码如下所示。

```
class Program
{
    enum color {red,yellow,green,blue};                //声明枚举
    static void Main(string[] args)                    //主程序入口方法
    {
        Console.WriteLine(color.green);                //查看枚举成员变量 green
        Console.ReadKey();                             //等待用户按键
    }
}
```

编译并运行上述代码，其输出结果为 green，说明在程序中已经对枚举变量中的成员初始化了。

2．使用枚举类型

枚举是用户自定义类型，所以在程序中可以引用用户的自定义类型进行自定义类型的变量的创建，示例代码如下所示。

```
color mycolor = color.green;                           //使用枚举类型
```

注意：枚举类型的定义只能在命名空间或类内声明，否则编译器会报错。

4．枚举成员的赋值和常用类型

声明并初始化枚举类型，也可以对枚举成员的值进行初始化，示例代码如下所示。

```
enum color {red=1,yellow=2,green=3,blue=1};            //枚举成员的赋值
```

不仅可以为枚举成员初始化，也可以为枚举成员定义基本类型，示例代码如下所示。

```
enum color:int   {red=1,yellow=2,green=3,blue=1};      //定义基本类型
```

2.3.8 类型转换

在应用程序开发当中，很多的情况都需要对数据类型进行转换，以保证程序的正常运行。类型转换是数据类型和数据类型之间的转换，在.NET 中，存在着大量的类型转换，常见的类型转换代码如下所示。

```
int i = 1;                                             //声明整型变量
Console.WriteLine(i);                                  //隐式转换输出
```

在上述代码中 i 是整型变量而 WriteLine 方法的参数是 Object 类型，但是 WriteLine 方法依旧能够正确输出是因为系统将 i 的类型在输出的时候转换成了字符型。在.NET 框架中，有隐式转换和显式转换，隐式转换是一种由 CLR 自动执行的类型转换，如上述代码中的，就是一种隐式的转换（开发人员不明

确指定的转换），该转换由 CLR 自动的将 int 类型转换成了 string 型。在.NET 中，CLR 支持许多数据类型的隐式转换，CLR 支持的类型转换列表如表 2-5 所示。

表 2-5　CLR支持的转换列表

从该类型	到该类型
sbyte	short,int,long,float,double,decimal
byte	short,ushort,int,uint,long,ulong,float,double,decimal
short	int,long,float,double,decimal
ushort	int,uint,long,ulong,float,double,decimal
int	long,float,double,decimal
uint	long,ulong,float,double,decimal
long,ulong	float,double,decimal
float	double
char	ushort,int,uint,long,ulong,float,double,decimal

显式转换是一种明确要求编译器执行的类型转换。在程序开发过程中，虽然很多地方能够使用隐式转换，但是隐式转换有可能存在风险，显式转换能够通过程序捕捉进行错误提示。虽然隐式也会提示错误，但是显式转换能够让开发人员更加清楚的了解代码中存在的风险并自定义错误提示以保证任何风险都能够及早避免，示例代码如下所示。

```
int i = 1;                                  //声明整型变量 i
float j = (float)i;                         //显式转换为浮点型
```

上述代码说明了显式转换的基本语法格式，具体语法格式如下所示。

```
type variable1=(cast-type)variable2;
```

注意：显式的转换可能导致数据的部分丢失，如 3.1415 转换为整型的时候会变成 3。

除了隐式的转换和显式的转换，还可以使用.NET 中的 Convert 类来实现转换，即使是两种没有联系的类型也可以实现转换。Convert 类的成员函数都是静态方法，当调用 Convert 类的方法时无需创建 Convert 对象，当使用显式的转换的时候，若代码如下所示，则编译器会报错。

```
string i = "1";                             //声明字符串变量
int j = (int)i;                             //显式转换为整型
Console.WriteLine(j);                       //隐式转换为字符串
```

但是明显的是，字符串变量 i 的值是有可能转换成整型变量值 1 的，Convert 类能够实现转换，示例代码如下所示。

```
string i = "1";                             //声明字符串变量
int j = Convert.ToInt32(i);                 //显式转换为整型
Console.WriteLine(j);                       //隐式转换为字符串
```

上述代码编译通过并能正常运行。Convert 类提供了诸多的转换功能，每个 Toxx 方法都将变量的值转换成相应.NET 简单数据类型的值，如 Int16、Int32、String 等。但是值得注意的是，并不是每个变量的值都能随意转换，示例代码如下所示。

```
string i = "haha";                          //声明字符串变量
int j = Convert.ToInt32(i);                 //错误的转换
```

上述代码中，i 的值是字符串“haha”，很明显，该字符串是无法转换为整型变量的。运行此代码后系统会抛出异常提示字符串“haha”不能够转换成整型常量。

2.4 编写表达式

在了解了 C#中的数据类型、变量的声明和初始化方式、以及类型转换等基本知识，就需要了解如

何进行表达式的编写。表达式在C#应用程序开发中非常的重要，本节将说明如何使用运算符创建和使用表达式。

2.4.1 表达式和运算符

表达式和运算符是应用程序开发中最基本也是最重要的一个部分，表达式和运算符组成一个基本语句，语句和语句之间组成函数或变量，这些函数或变量通过某种组合形成类。

1．定义

表达式是运算符和操作符的序列。运算符是个简明的符号，包括实际中的加减乘除，它告诉编译器在语句中实际发生的操作，而操作数既操作执行的对象。运算符和操作数组成完整的表达式。

2．运算符类型

在大部分情况下，对运算符类型的分类都是根据运算符所使用的操作数的个数来分类的，一般可以分为三类，这三类分别如下所示。

- 一元运算符：只使用一个操作数，如（!），自增运算符（++）等等，如 i++。
- 二元运算符：使用两个操作数，如最常用的加减法，i+j。
- 三元运算符：三元运算符只有（?:）一个。

除了按操作数个数来分以外，运算符还可以按照操作数执行的操作类型来分，如下所示。

- 关系运算符。
- 逻辑运算符。
- 算术运算符。
- 位运算符。
- 赋值运算符。
- 条件运算符。
- 类型信息运算符。
- 内存访问运算符。
- 其他运算符。

在应用程序开发中，运算符是最基本也是最常用的，它表示着一个表达式是如何进行运算的。常用的运算符如表 2-6 所示。

表 2-6　常用的运算符

运算符类型	运算符
元运算符	(x),x.y,f(x),a[x],x++,x--,new,typeof,sizeof,checked,uncheck
一元运算符	+,-,~!,++x,--x,(T)x,
算术运算符	+,-,*,/,%
位运算符	<<,>>,&,|,^,~
关系运算符	<,>,<=,>=,is,as
逻辑运算符	&,|,^
条件运算符	&&,||,?
赋值运算符	=,+=,-=,*=,/=,<<=,>>=,&=,^=,|=

正如表 2-5 中所示，C#编程中所需要使用到的运算符都能够通过相应的类别进行相应的分类，但其分类的标准并不是唯一的。

3．算术运算符

程序开发中常常需要使用算术运算符，算术运算符用于创建和执行数学表达式，以实现加、减、乘、除等基本操作，示例代码如下所示。

```
int a = 1;                                    //声明整型变量
int b = 2;                                    //声明整型变量
int c = a + b;                                //使用+运算符
int d = 1 + 2;                                //使用+运算符
int e = 1 + a;                                //使用+运算符
int f = b - a;                                //使用-运算符
int f = b / a;                                //使用/运算符
```

注意：当除数为 0，系统会抛出 DivideByZeroException 异常，在程序开发中应该避免出现逻辑错误，因为编译器不会检查逻辑错误，只有在运行中才会提示相应的逻辑错误并抛出异常。

在算术运算符中，运算符“%”代表求余数，示例代码如下所示。

```
int a = 10;                                   //声明整型变量
int b = 3;                                    //声明整型变量
Console.WriteLine((a%b).ToString());          //求 10 除以 3
```

上述代码实现了“求 10 除以 3”的功能，其运行结果为 1。在 C#的运算符中还包括自增和自减运算符，如“++”和“--”运算符。++和--运算符是一个单操作运算符，将目的操作数自增或自减 1。该运算符可以放置在变量的前面和变量的后面，都不会有任何的语法错误，但是放置的位置不同，实现的功能也不同，示例代码如下所示。

```
int a = 10;                                   //声明整型变量
int a2 = 10;                                  //声明整型变量
int b = a++;                                  //执行自增运算
int c = ++a2;                                 //执行自增运算
Console.WriteLine("a is " + a);               //输出 a 的值
Console.WriteLine("b is " + b);               //输出 b 的值
Console.WriteLine("c is " + c);               //输出 c 的值
```

运行结果如图 2-14 所示。

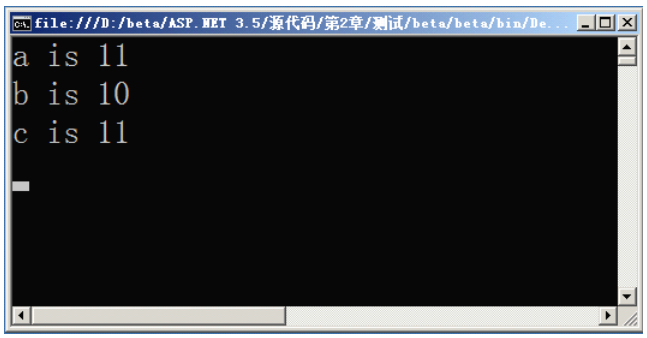

图 2-14　++运算符

由运行结果所示，变量 a、a2 为 10，在使用了++运算符后，a 和 a2 分别变为了 11，而 b 的赋值语句代码中使用的为后置自增运算符，示例代码如下所示。

```
int b = a++;                                  //b=10
```

当执行了上述代码后，b 的值为 10，而 a 会自增为 11，因为上述代码首先会将变量 a 的值赋值给 b 变量，赋值后再进行自增。而 c 的赋值语句中使用的为前置自增运算符，示例代码如下所示。

```
int c = ++a2;                                 //c=11
```

当执行了上述代码后，变量 c 的值为 11，是因为在执行自增操作时，首先进行自增，再将 a2 变量的值赋值给 c。当运算符在操作数后时，操作数会赋值给新的变量，然后再自增 1，当运算符在操作数

前时，操作数会先进行自增或自减，然后再赋值给新变量。

4．关系运算符

关系运算符用于创建一个表达式，该表达式用来比较两个对象并返回布尔值。示例代码如下所示。

```
string a="nihao";                                    //声明字符串变量 a
string b="nihao";                                    //声明字符串变量 b
if (a == b)                                          //使用比较运算符
{
    Console.WriteLine("相等");                        //输出比较相等信息
}
else
{
    Console.WriteLine("不相等");                      //输出比较不相等信息
}
```

关系运算符如“>”，“<”，“>=”，“<=”等同样是比较两个对象并返回布尔值，示例代码如下所示。

```
string a="nihao";                                    //声明字符串变量 a
string b="nihao";                                    //声明字符串变量 b
if (a == b)                                        //比较字符串，返回布尔值
{
    Console.WriteLine((a == b).ToString());          //输入比较后的布尔值
}
else
{
    Console.WriteLine((a == b).ToString());          //输入比较后的布尔值
}
```

编译并运行上述程序后，其输出为 true。若条件不成立，即如 a 不等于 b 的变量值，则返回 false。该条件所以可以直接编写在 if 语句中进行条件筛选和判断。

技巧：在使用判断的时候，可以直接使用表达式，只要表达式的返回值是布尔型的即可，同样也可以使用类型转换 Convert.ToBoolean 方法转换。

初学者很容易错误的使用关系运算符中的“==”号，因为初学者通常会将等于运算符编写为“=”号，示例代码如下所示。

```
if (a = b)                                           //使用布尔值布尔值
```

在这里，“=”号不等于“==”号，“=”号的意义是给一个变量赋值，而“==”号是比较两个变量的值是否相等，如果写成上述代码，虽然编译器不会报错，但是其运行过程就不是开发人员想象的流程。

5．逻辑运算符

逻辑运算符和布尔类型组成逻辑表达式。NOT 运算符“!”使用单个操作数，用于转换布尔值，即取非，示例代码如下所示。

```
bool myBool = true;                                  //创建布尔变量
bool notTrue = !myBool;                              //使用逻辑运算符
```

与其他编程语言相似的是，C#也使用 AND 运算符“&&”。该运算符使用两个操作数做与运算，当有一个操作数的布尔值为 false 时，则返回 false，示例代码如下所示。

```
bool myBool = true;                                  //创建布尔变量
bool notTrue = !myBool;                              //使用逻辑运算符取反
bool result = myBool && notTrue;                     //使用逻辑运算符计算
```

同样，C#中也使用“||”运算符来执行 OR 运算，当有一个操作数的布尔值为 true 时，则返回 true。当使用“&&”运算符和“||”运算符时，他们是短路（short-circuit）的，这也就是说，当一个布尔值能够由前一个或前几个操作数决定结果，那么就不取使用剩下的操作数继续运算而直接返回结果，示例代码如下所示。

```
bool myBool = true;                                          //创建布尔变量
bool notTrue = !myBool;                                      //使用逻辑运算符取反
bool result = myBool && notTrue;                             //使用逻辑运算符计算
bool other = true;                                           //创建布尔变量
if (result&&other)                                           //短路操作
{
    Console.WriteLine("true");                               //输出布尔值
}
else
{
    Console.WriteLine("false");                                  //输出布尔值
}
```

从上述代码可以看到，变量 other 的值为 true,而 result 的值为 false，那么 result&&other 语句中，会直接返回 false，说明条件失败。另外，在逻辑运算符中还包括 XOR 异或运算符“^”，该运算符确定是否操作数是否相同，若操作数的布尔值相同，则表达式将返回 false。

在 C#应用程序开发中，并不支持从整型直接转换为布尔值，虽然在 C/C++中能够直接编写数值进行逻辑判断，示例代码如下所示。

```
int result = 1;                                              //使用整型
if (result)                                                  //c++的转换
{
    cout<<"true";
}
else
{
    cout<<"false";
}
```

而上述代码如果在 C#中运行，编译器会报错，说明 C#并不支持显式的直接从 int 类型到 bool 类型的转换，但是使用 Convert 对象的静态方法可以实现同样的效果，代码如下所示。

```
int result = 1;
if (Convert.ToBoolean(result))                               //使用 Convert 静态对象
{
    Console.WriteLine("true");                               //输出布尔值
}
else
{
    Console.WriteLine("false");                              //输出布尔值
}
```

注意：在编程语言中，非 0 的数值都为 true，虽然 C#不支持隐式的转换 int 到 bool 类型，但是 Convert.ToBoolean 静态方法提供了实现，任何非 0 的参数都将返回 true。

6．位运算符

位运算符通常使用为模式来操作整数类型，这些位运算符非常实用。位运算符包括“<<”、“>>”、AND、OR 和 XOR。左移位运算符“<<”将整型中的位左移指定位数，每一次左移，整型变量的值将乘

以 2，代码如下所示。

```
int result = 4;
Console.WriteLine((result << 1).ToString());                    //左移并输出
```

运行结果为 8，操作原理如图 2-15 所示。

当使用“<<”运算符时，左移操作将舍弃移出的所有位，并用 0 来填充移入的位。同样，右移运算符“>>”也将操作数右移，每一次右移，整型变量的值将除以 2。AND 位运算符“&”通过逐位执行逻辑 AND 运算，从而生成新的操作数，AND 运算中，两个对应的值，若有一个值为 0，则全部为 0，原理如图 2-16 所示。

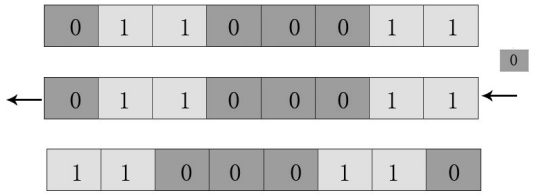

图 2-15　左移

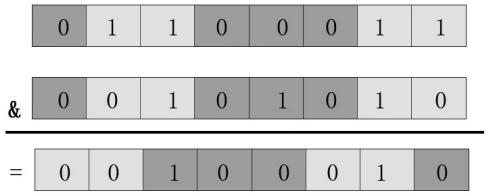

图 2-16　AND 位运算

OR 位运算符“|”的是用方法和原理和 AND 位运算符“&”基本相同，其区别在于使用的是 OR 运算，当有一个值为 1，则结果为 1，其原理如图 2-17 所示。

XOR 位运算符“^”的用法和 AND 位运算符类似，其区别在于当两个值相同时，执行计算的结果为 0，否则为 1，其原理图如 2-18 所示。

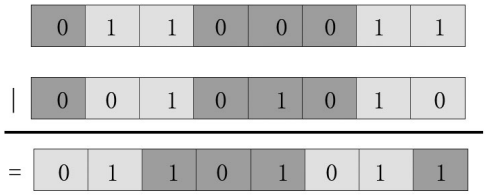

图 2-17　OR 位运算

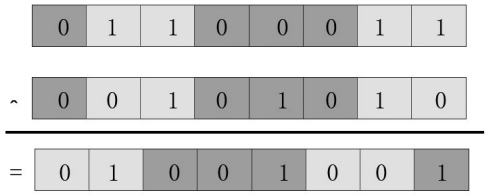

图 2-18　XOR 位运算

取补运算符“~”将生成整型类型的补码。如原值中的 1 将变为 0，而 0 则变为 1，原理图如 2-19 所示。

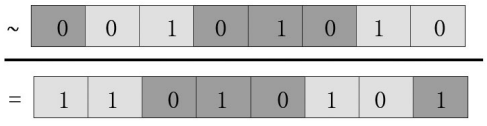

图 2-19　取补位运算

7．赋值运算符

C#提供了几种类型的赋值运算符，最常见的就是“=”运算符。C#还提供了组合运算符，如“+=”、“-=”、“*=”等。“=”运算符通常用来赋值，示例代码如下。

```
int a,b,c;                                                      //声明三个整型变量
a = b = c = 1;                                                  //使用赋值运算符
```

上述代码声明并初始化 3 个整型变量 a、b、c 并初始化值这些变量的值为 1。加法赋值运算符“+=”将加法和赋值操作组合起来，先把第一个数值的值加上第二个数值的值再存放到第一个数值的，示例代

码如下所示。

```
a += 1;                                                    //进行自加运算
```

上述代码会将变量 a 的值加上 1 并再次赋值回 a，上述代码实现的功能和以下代码等效。

```
a = a + 1;                                                 //不使用+=运算符
```

同样，“-=”，“*=”，“/=”都是将第一个数值的值与第二个数值的值做操作再存放到第一个数值，同样也支持位运算符，示例代码如下所示。

```
a <<= 1;                                                   //移位运算
```

赋值运算符不仅支持加减乘除和位运算，同样也支持条件运算符，示例代码如下所示。

```
a &= 1;                                                    //条件运算
```

8．条件运算符

条件运算符“?:”需要三个操作数，示例代码如下所示。

```
bool ifisTrue=true;                                        //创建布尔值
string result = ifisTrue ? "true" : "false";               //使用三元条件运算符
Console.WriteLine(result.ToString());                      //输出布尔值
```

上述代码中，使用了条件运算符“?:”，条件运算符“?:”会执行第一个条件，若条件成立，则返回“:”运算符前的一个操作数的数值，否则返回“:”运算符后的操作数的数值。上述代码中，变量 ifisTrue 为 true，则返回“true”。

技巧：当记忆条件运算符的时候，“？”可以被记忆为“条件成立吗”，如果成立，则执行第一个，否则执行第二个，如(1>2)?1:2 可以解释成 1 大于 2 吗，大于则返回 1，不大于则返回 2，这样有助于记忆和阅读。

2.4.2 运算符的优先级

开发人员需要经常创建表达式来执行应用程序的计算，简单的有加减法，复杂的有矩阵、数据结构等，在创建表达式时，往往需要一个或多个运算符。在多个运算符之间的运算操作时，编译器会按照运算符的优先级来控制表达式的运算顺序，然后再计算求值。例如在生活中也常常遇到这样的计算，如 1+2*3。如果在程序开发中，编译器优先运算“+”运算符并进行计算就会造成错误的结果。

1．运算顺序

表达式中常用的运算符的运算顺序如表 2-7 所示。

表 2-7　运算符优先级

运算符类型	运算符
元运算符	X.y,f(x),a[x],x++,x--,new,typeof,checked,unchecked
一元运算符	+,-,!,~,++x,--x,(T)x
算术运算符	*,/,%
位运算符	<<,>>,&,|,^,~
关系运算符	<,>,<=,>=,is,as
逻辑运算符	&,^,|
条件运算符	&&,||,?
赋值运算符	=,+=,-=,*=,/=,<<=,>>=,&=,^=,|=

当执行运算 1+2*3 的时候，因为“+”运算符的优先级比“*”运算符的优先级低，则当编译器编译表达式并进行运算的时候，编译器会首先执行“*”运算符的乘法操作，然后执行“+”运算符的加法操作。当需要指定运算符的优先级，可以使用圆括号来告知编译器自定义运算符的优先级，示例代码如下所示。

```
c = a + b * c;                                    //先执行乘法
c = (a + b) * c;                                  //先执行加法
```

2．左结合和右结合

所有的二元运算符都是右两个操作数，除了赋值运算符以外其他的运算符都是左结合的，而赋值运算符是右结合，示例代码如下所示。

```
a + b + c;                                        //结合方式为(a+b)+c
a = b = c;                                        //结合方式为 a=(b=c)
```

2.5　使用条件语句

程序开发中，开发人员经常遇到选择性的问题，如用户是否注册。如果用户已经注册则允许用户登陆，否则就跳转到注册页面。这个时候，就需要在程序中使用条件语句。if 是最常用的条件语句，同时，if 还包括 if、if　else、if else if 等语句用于执行复杂的条件选择。

2.5.1　if 语句的使用方法

if 语句用于判断条件并按照相应的条件执行不同的代码块，if 语句包括多种呈现形式，这些形式分别是 if、if else、if else if。

1．声明 if 语句

if 语句的语法如下所示。

```
if(布尔值) 程序语句
```

当布尔值为 true，则会执行程序语句，当布尔值为 false 时，程序会跳过执行的语句执行，示例代码如下所示。

```
if (true)                                         //使用 if 语句
{
    console.writeline("ture");                    //为 true 的代码块
}
```

上述代码首先会判断 if 语句的条件，因为 if 语句的条件为 true，所以 if 语句会执行大括号内的代码，程序运行会输出字符串 true，如果将 if 内的条件改为 false，那么程序将不会执行大括号内的代码，从而不会输出字符串 true。

2．声明 if else 语句

if else 语句的语法如下所示。

```
if(布尔值) 程序语句 1 else 程序语句 2
```

同样，当布尔值为 true，则程序执行程序语句 1，但当布尔值为 false 时，程序则执行程序语句 2，示例代码如下所示。

```
if (true)                                         //使用 if 语句判断条件
{
    console.writeline("ture");                    //当条件为真时执行语句
}
else                                              //如果条件不成立则执行
{
    console.writeline("false");                   //当条件为假时执行语句
```

```
}
```

上述代码中 if 语句的条件为 true，所以 if 语句会执行第一个大括号中间的代码，而如果将 true 改为 false，则 if 语句会执行第二个大括号中的代码。

3．声明 if else if 语句

当需要进行多个条件判断是，可以编写 if else if 语句执行更多条件操作，示例代码如下所示。

```
if (month == "12")                                    //判断 month 是否等于 12
{
    console.writeline("winter");                      //输出 winter
}
else if (month == "7")                                //判断 month 是否等于 7
{
    console.writeline("summer");                      //输出 summer
}
else if (month == "3")                                //判断 month 是否等于 3
{
    console.writeline("spring");                      //输出 spring
}
else                                                  //当都不成立时执行
{
    console.writeline("we don't have this month");    //输出默认情况
}
```

上述代码会判断相应的月份，如果月份等于 12，就会执行相应的大括号中的代码，否则会继续进行判断，如果判断该月份即不是 3 月也不是 7 月，说明所有的条件都不复合，则会执行最后一段大括号中的代码。

4．使用布尔值和布尔表达式

在 if 语句编写中，if 语句的条件可以使用布尔值或布尔表达式，以便能够显式的进行判断，示例代码如下所示。

```
if (true)                                             //使用布尔值
{
    console.writeline("ture");                        //输出 ture
}
```

if 语句的条件不仅能够由单个布尔值或表达式组成，同样也可以由多个表达式组成，示例代码如下所示。

```
if (true||false)                                      //使用多个布尔值
{
    console.writeline("ture");                        //输出 ture
}
```

同样在编写 if 语句的条件时，可以使用复杂的布尔表达式进行条件约束，示例代码如下所示。

```
if ((a==b)&&(b==c))                                   //使用复杂的布尔表达式
{
    console.writeline("a 和 b 相等,b 和 c 也相等");    //输出相等信息
}
```

5．使用三元运算符

三元运算符(?:)是 if else 语句的缩略形式，比较后并返回布尔值，熟练使用该语句可以让代码变得更简练。

2.5.2 switch 选择语句的使用

switch 语句根据某个传递的参数的值来选择执行的代码。在 if 语句中，if 语句只能测试单个条件，如果需要测试多个条件，则需要书写冗长的代码。而 switch 语句能有效的避免冗长的代码并能测试多个条件。

1．声明 switch 选择语句

Switch 语句的语法如下所示。

```
switch (参数的值)
{
    case  参数的对应值 1:  操作 1; break;
    case  参数的对应值 2:  操作 2; break;
    case  参数的对应值 3:  操作 3; break;
}
```

从上述语法格式中可以看出 switch 的语法格式。在 switch 表达式之后跟一连串 case 标记相应的 switch 块。当参数的值为某个 case 对应的值的时候，switch 语句就会执行对应的 case 的值后的操作，并以 break 结尾跳出 switch 语句。若没有对应的参数时，可以定义 default 条件，执行默认代码，示例代码如下所示。

```
int x;
switch (x)                                                    //switch 语句
{
    case 0: Console.WriteLine("this is 0"); break;            //x=0 时执行
    case 1: Console.WriteLine("this is 1"); break;            //x=1 时执行
    case 2: Console.WriteLine("this is 2"); break;            //x=2 时执行
    default:Console.WriteLine(“这是默认情况”);break;
}
```

在上述代码中，当 x 等于 0 的时候，就会执行 case 0 的操作，就执行了 Console.WriteLine("this is 0")。如果 x 等于 1，语句就会执行 case 1 的操作。switch 不仅能够通过数字进行判断，还能够通过字符进行判断。

2．使用 break 跳出语句

从上述代码中可以看出，每一个操作后面都使用了一个 break 语句。在 C/C++中，程序员可以被允许不写 break 而贯穿整个 switch 语句，但是在 C#中不以 break 结尾是错误的，并且编译器不会通过。因为 C#的 switch 语句不支持贯穿操作，因为 C#中是希望避免在应用程序的开发中出现这样的错误。

注意：在 C#中，可以使用 goto 语句模拟，继续执行下一个 case 或 default。尽管在程序中可以这样做，但是会降低代码的可读性，所以不推荐使用 goto 语句。

3．switch 的执行顺序

switch 语句能够对相应的条件进行判断，示例代码下代码所示。

```
int x;
switch (x)                                                    //switch 语句
{
    case 0: Console.WriteLine("this is 0"); break;            //x=0 时执行
    case 1: Console.WriteLine("this is 1"); break;            //x=1 时执行
    case 2: Console.WriteLine("this is 2"); break;            //x=2 时执行
    default: Console.WriteLine("no one"); break;              //都不满足时执行
}
```

从上述代码中分析，整型变量 x 被声明，并初始化，执行 switch 语句，当 x 的值为 0 的时候，case 0 之后的语句就会执行，即会执行 Console.WriteLine("this is 0");执行完毕后，语句执行 break 跳出 switch 语句。当 x 的值等于 3 的，switch 操作会发现在 case 中没有与之相等的标记，则会执行 default 标记并执行默认的代码块。

注意：在 switch 语句中，default 语句并不是必须的，但是编写 default 是可以为条件设置默认语句。

4．使用枚举类型

在 switch 表达式中，参数的类型必须为整型、字符型、字符串型、枚举类型或是能够隐式转换为以上类型的数据类型。在 switch 中常常会用到枚举类型，示例代码如下所示。

```
enum season { spring,summer,autumn,winter }                                    //声明枚举类型
static void Main(string[] args)
{
    season mySeason=season.summer;                                              //初始化
    switch (mySeason)
    {
        case season.spring: Console.WriteLine("this is spring"); break;      //mySeason=spring 时
        case season.summer: Console.WriteLine("this is summer"); break; //mySeason=summer 时
        case season.autumn: Console.WriteLine("this is autumn"); break; //mySeason=autumn 时
        case season.winter: Console.WriteLine("this is winter"); break;      //mySeason=winter 时
        default: Console.WriteLine("no one"); break;
    }
    Console.ReadKey();                                                          //等待用户按键
}
```

运行代码如图 2-20 所示。

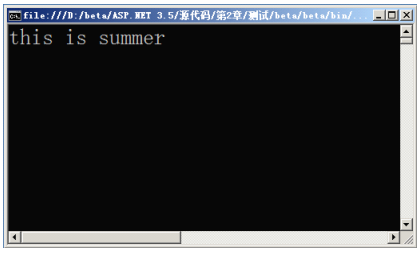

图 2-20　switch 中使用枚举类型

5．组合 case 语句

在某些情况下，一些条件所达成的效果是相同，这就要求在 switch 中往往需要对多个标记使用同一语句。switch 语句能够实现多个标记使用同一语句，代码如下所示。

```
switch (mySeason)
{
    case season.spring:
    case season.summer: Console.WriteLine("this is spring and summer"); break;//组合 case
    case season.autumn:
    case season.winter: Console.WriteLine("this is autumn and winter"); break;   //组合其他条件
    default: Console.WriteLine("no one"); break;                                 //默认 case
}
```

2.6 使用循环语句

程序开发中，经常需要对某个代码块执行循环，使编译器能够重复执行某个代码块来完成计算。循环能够减少代码量，避免重复输入相同的代码行，也能够提高应用程序的可读性。常见的循环语句有 for、while、do、for each。

2.6.1 for 循环语句

for 循环一般用于已知重复执行次数的循环，是程序开发中常用的循环条件之一，当 for 循环表达式中的条件为 true 时，就会一直循环代码块。因为循环的次数是在执行循环语句之前计算的，所以 for 循环又称作预测式循环。当表达式中的条件为 false 时，for 循环会结束循环并跳出。for 循环语法格式如下所示。

```
for(初始化表达式,条件表达式,迭代表达式)
    循环语句
```

for 循环的优点就是 for 循环的条件都位于同一位置，同样，循环的条件可以使用复杂的布尔表达式表示。for 循环表达式包含三个部分，即初始化表达式、条件表达式和迭代表达式。当 for 循环执行时，将按照以下顺序执行。

- 在 for 循环开始时，首先运行初始化表达式。
- 初始化表达式初始化后，则判断表达式条件。
- 若表达式条件成立，则执行循环语句。
- 循环语句执行完毕后，迭代表达式执行。
- 迭代表达式执行完毕后，再判断表达式条件并循环。

for 语句循环示意图 2-21 所示。

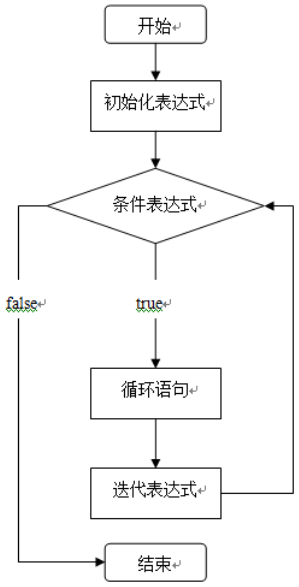

图 2-21 for 循环示意图

开发人员能够通过编写 for 循环语句进行代码块的重复，示例代码如下所示。

```
for (int i = 0; i < 100; i++)                                    //循环 100 次
```

```
{
    Console.WriteLine(i);                                    //输出 i 变量的值
}
```

技巧：for 循环即可做增量操作也可以做减量操作，如可以写为 for(int i=10;i>0;i--)，说明 for 循环的结构非常灵活，同样 for 循环的条件，迭代表达式也不仅仅局限与此。

for 循环还可以声明多个变量，在初始化表达式和迭代表达式中声明不只一个变量，示例代码如下所示。

```
for (int i = 0, j = 0; (i < 100) && (j < 100); i++, j++)    //多个条件循环
{
    Console.WriteLine("i is" + i);                           //输出 i 变量的值
    Console.WriteLine("j is" + j);                           //输出 j 变量的值
}
```

2.6.2 while 循环语句

while 语句同 for 语句一样都可以执行循环，但是 while 的使用更加灵活，开发人员可以在代码块执行前判断条件，也可以在代码块执行一次后再行判断条件。

1．while 语句的语法

while 语句是除了 if 语句以外另一个常用语句，while 语句的使用方法基本上和 if 语句相同，其区别就在于，if 语句一般需要先知道循环次数，而 while 语句即便不知道循环次数也可以使用。while 语句基本语法如下所示。

```
while(布尔值)
    执行语句
```

while 语句包括两个部分，布尔值和执行语句，while 语句执行步骤一般如下所示。

- 判断布尔值。
- 若布尔值为 true 则执行语句，否则跳过。

while 语句循环示意图如图 2-22 所示。

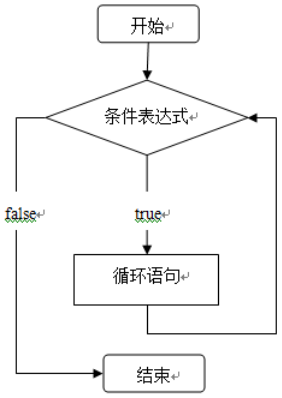

图 2-22　while 语句循环示意图

while 语句示例代码如下所示。

```
x = 100;                                                     //声明整型变量
while (x != 1)                                               //判断 x 不等于 1
{
    x--;                                                     //x 自减操作
```

```
}
```

上述代码，声明并初始化变量 x 等于 100，当判断条件 x!=1 成立时，则执行 x—操作，直到条件 x!=1 不成立时才跳过 while 循环。

2．continue 关键字：继续执行语句

在 while 语句中，可以使用 continue 语句来执行下一次迭代而不执行完所有的执行语句，示例代码如下所示。

```
int x, y;                                          //声明整型变量
x = 10;                                            //初始化 x
y = 10;                                            //初始化 y
while (x != 1)                                     //如果 x 不等于 1
{
    x--;                                           //x 自减操作
    Console.WriteLine(x);                          //输出 x
    continue;                                      //返回循环
    y--;                                           //y 自减操作（但不执行）
    Console.WriteLine(y);                          //输出 y（但不执行）
}
```

上述代码声明了 x，y 两个整型变量，并初始化值为 10，当 x 不等于 1 时执行 while 循环。在 while 循环语句中，当执行到 continue 关键字时则跳出并继续执行 while 循环而不执行 continue 关键字后的语句，如 y—语句。

2．break 关键字：跳出循环语句

break 关键字允许程序在 while 循环中跳出并终止循环，从而能够继续执行循环后的语句，示例代码如下所示。

```
while (x != 1)                                     //如果 x 不等于 1
{
    x--;                                           //x 自减操作
    Console.WriteLine(x);                          //输出 x
    if (x == 5)                                    //如果 x 等于 5
    {
        break;                                     //跳出循环
    }
}
```

上述代码判断 x 是否等于 5。x 如果等于 5，则 break 语句会生效并跳出循环，继续执行 while 循环语句之后的代码。

2.6.3 do while 循环语句

do while 循环和 while 循环十分相似，区别在于 do while 循环会执行一次执行语句，然后再判断 while 中的条件。这种循环成为后测试循环，当程序需要执行一次语句再循环的时候，do while 语句是非常实用的。do while 语句语法格式如下所示。

```
do
{执行语句}
while(布尔值);
```

do while 语句包含两个部分，执行语句和布尔值。与 while 循环语句不同的时，执行步骤首先执行一次执行语句，具体步骤如下所示。

- 执行一次执行语句。
- 判断布尔值。
- 若布尔值为 true，则继续执行，否则跳出循环。

while 语句循环示意图如图 2-23 所示。

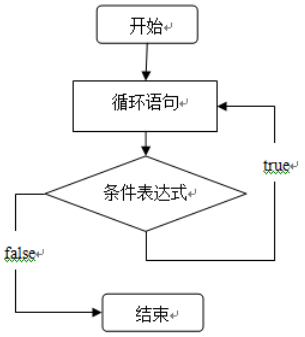

图 2-23　do while 语句循环示意图

do while 语句示例代码如下所示。

```
int x=90;                                    //声明整型变量
do                                           //首先执行一次代码块
{
    x ++;                                    //x 自增一次
    Console.WriteLine(x);                    //输出 x 的值
}
while (x < 100);                             //判断 x 是不是小于 100
```

上述代码在运行时会执行一次大括号内的代码块，执行完毕后才会进行相应的条件判断。

2.6.4　foreach 循环语句

for 循环语句常用的另一种用法就是对数组进行操作，C#还提供了 foreach 循环语句，如果想重复集合或者数组中的所有条目，使用 foreach 是很好的解决方案。foreach 语句语法格式如下

```
foreach (局部变量 in 集合)
    执行语句;
```

- for each 语句执行顺序如下所示。
- 集合中是否存在元素。
- 若存在，则用集合中的第一个元素初始化局部变量。
- 执行控制语句。
- 集合中是否还有剩余元素，若存在，则将剩余的第一个元素初始化局部变量。
- 若不存在，结束循环。

foreach 循环示意图如图 2-24 所示。

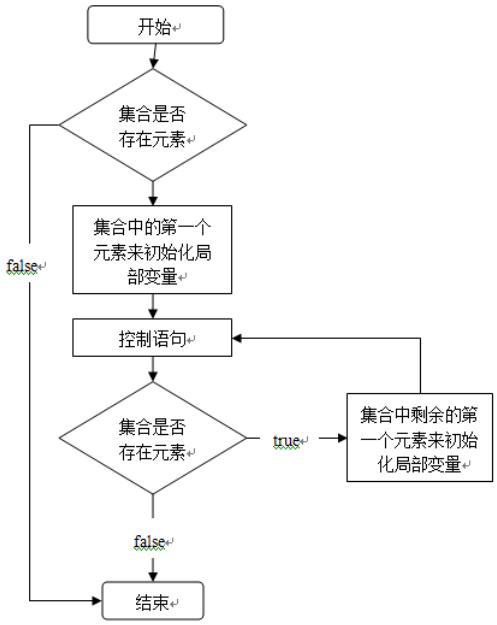

图 2-24　foreach 语句循环示意图

foreach 语句示例代码如下所示。

```
string[] str = { "hello", "world", "nice", "to", "meet", "you" };    //定义数组变量
foreach (string s in str)                                            //如果存在元素则执行循环
{
    Console.WriteLine(s);                                            //输出元素
}
```

上述代码声明了数组 str，并对 str 数组进行遍历循环。运行结果如图 2-25 所示。

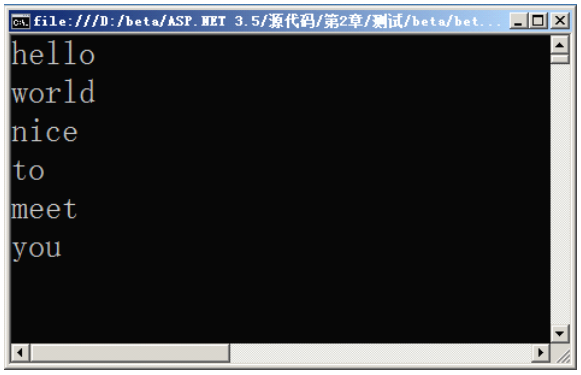

图　2-25 foreach 语句循环示例

注意：在使用 foreach 语句的时候，局部变量的数据类型应该与集合或数组的数据类型相同，否则编译器会报错。

2.7　异常处理语句

在传统的 ASP 开发过程中，要发现错误是非常复杂和困难的，常常错误发生后，很难找到错误的代码行。C#为处理程序执行期间可能出现的异常情况提供内置支持，这些异常由正常控制流之外的代码

处理。常用的异常语句包括 throw，try，catch 等。

2.7.1 throw 异常语句

throw 语句用于发出在程序执行期间出现的异常情况的信号、引发异常的是一个对象，该对象的类是从 System.Exception 派生的。通常 throw 语句与 try-catch 或 try-final 语句一起使用。示例代码如下所示。

```
int x = 1;                                          //声明整型变量 x
int y = 0;                                          //声明整型变量 y
if (y == 0)                                         //如果 y 等于 0
{
    throw new ArgumentException();                  //抛出异常
}
Console.WriteLine("除数不能为 0");                  //输出错误信息
```

上述代码使用 throw 语句引发异常并向用户输出了异常信息。

2.7.2 try-catch 异常语句

try-catch 语句由一个 try 和一个或多个 catch 子句构成，这些子句可以指定不同的异常处理应用程序。当 try 块中的代码异常，则会执行 catch 块的保护代码，在应用程序开发当中，try-catch 语句能够处理异常并返回给用户友好的错误提示，示例代码如下所示。

```
int x = 1;                                          //声明整型变量 x
int y = 0;                                          //声明整型变量 y
try                                                 //尝试处理代码块
{
    x = x / y;                                      //出现异常
}
catch                                               //捕获异常
{
    Console.WriteLine("除数不能为空");              //抛出异常
}
```

上述代码试图用一个整型变量除以一个值为 0 的整型变量，不使用 try-catch 捕捉异常，则系统会抛出异常跳转到开发环境或代码块。使用 try-catch，系统同样会抛出异常，但是开发人员能够通过程序捕捉异常并自定义输出异常。同样，它也可以接受从 System.Exception 派生的对象传递过来的参数，示例代码如下所示。

```
int x = 1;                                          //声明整型变量
int y = 0;                                          //声明整型变量
try                                                 //尝试处理代码块
{
    x = x / y;                                      //进行除法计算
}
catch(Exception ee)                                 //使用 Exception 对象
{
    Console.WriteLine("除数不能为空,具体错误信息如下所示\n");   //输出错误信息
    Console.WriteLine(ee.ToString());               //捕获代码
}
```

运行结果如图 2-26 所示。

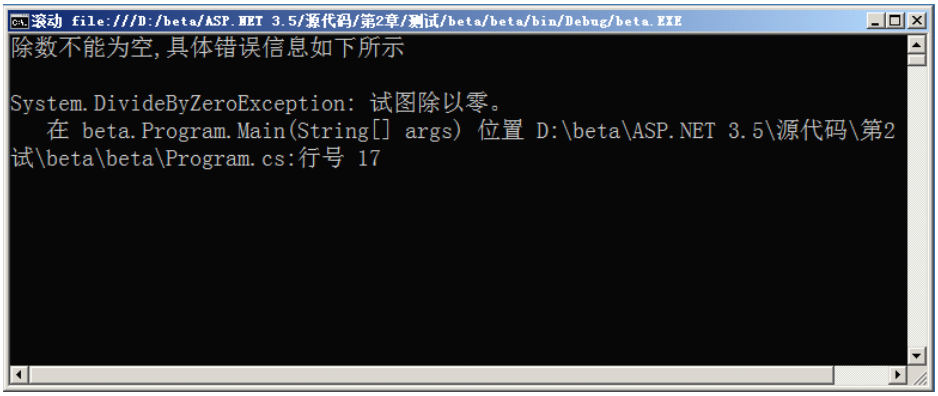

图 2-26　try-catch 语句使用示例

在运行结果中，程序详细的输出了异常的信息，此错误的信息由程序捕捉，并不会停止应用程序。

注意：try-catch 能够捕捉应用程序中的错误信息，但是 try-catch 会对程序的性能造成影响，在程序开发当中，应避免不必要的 try-catch 语句的出现。

2.7.3　try-finally 异常语句

catch 用于处理应用程序语句中出现的异常，而 finally 语句用于清除 try 块中分配的任何资源，以及运行应用程序中任何发生异常也必须执行的代码。finally 语句经常和 catch 语句搭配使用，示例代码如下所示。

```
int x = 1;                                          //声明整型变量 x
int y = 0;                                          //声明整型变量 y
try                                                 //尝试处理代码块
{
    x = x / y;                                      //进行除法计算
}
finally                                             //继续执行程序块
{
    Console.WriteLine("系统已自动停止");              //依旧输出错误信息
}
```

上述代码试图用一个整型变量除以一个值为 0 的整型变量，当异常发生时，系统会抛出异常，但是 finally 语句也被执行。

2.7.4　try-catch-finally 异常语句

try-catch-final 语句能够使应用程序更加健壮。try-finally 语句依旧会抛出异常，而 try-catch-finally 语句能够捕获异常并执行 finally 语句中的控制语句，try-catch-finally 语句结构和很灵活，示例代码如下所示。

```
int x = 1;                                          //声明整型变量 x
int y = 0;                                          //声明整型变量 y
try                                                 //尝试处理代码块
{
    x = x / y;                                      //进行除法计算
```

```
}
catch (Exception ee)                                    //捕获异常信息
{
    Console.WriteLine("除数不能为空,具体错误信息如下所示");    //抛出异常
    Console.WriteLine(ee.ToString());                   //输出异常信息
}
finally                                                 //继续执行程序块
{
    Console.WriteLine("系统已自动停止");                  //继续执行程序
}
```

上述代码试图用一个整型变量除以一个值为 0 的整型变量，当异常发生时，catch 捕获并抛出异常，捕获异常后，finally 语句也被执行。运行结果如图 2-27 所示。

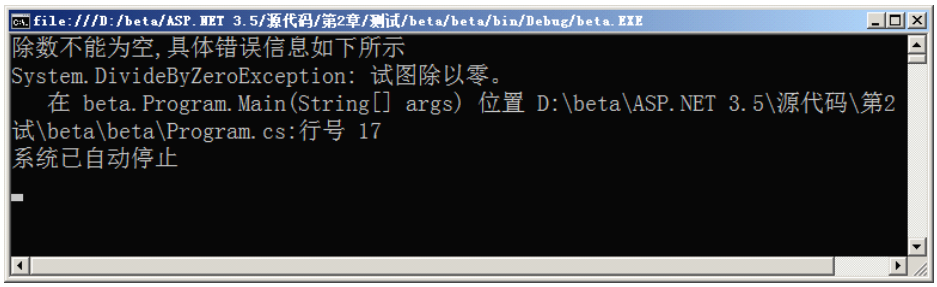

图 2-27　try-catch-finally 语句运行示例

2.8　小结

本章介绍了 C#语言的基本知识，包括变量、变量规则、表达式、条件语句、循环语句以及异常处理，本章主要讲解了：

- 变量：介绍了变量的概念、变量的声明以及初始化。
- 变量规则：介绍了变量的命名、规则。
- 表达式：介绍了表达式的创建和使用方法。
- 条件语句：介绍了 if、if else、if else if、switch 等条件语句的使用方法。
- 循环语句：介绍了 for，while、do while、foreach 等循环语句的使用方法。
- 异常处理：介绍了异常以及 throw、try-catch、try-finally、try-catch-finally 语句的使用方法。

C#语言的特性参考了 Java 的技术规则，在 C#中也有一个虚拟机，叫公共语言运行环境(CLR)。C#的体系结构与 Windows 的体系结构十分相同，因此 C#很容易被开发人员使用并熟悉，能够很快的适应开发。C#和 C++也有很多的相似之处，学习 C++的开发人员也能够适应 C#的学习和开发。C#比 C++又有了更多的增强功能，如类型安全，事件处理，随便帐集，代码安全性，垃圾回收等。

C#是基于.NET 体系的，也是.NET 体系中的风云人物。学好 C#，读者不仅能够开发 ASP.NET 应用程序，也能够开发 WinForm、WPF、WCF 等应用程序。这些应用程序的开发原理上都是相通的，所以，掌握好 C#基础是非常必要的。