第 3 章　C#语言基础

C#语言是一门简单、现代、优雅、面向对象、类型安全、平台独立的新型组件编程语言。其语法风格源自 C/C++家族，融合了 Visual Basic 的高效和 C/C++的强大。其优雅的语法风格，创新的语言特性，深受世界各地程序员的好评和喜爱。C#起源于 C 语言家族，因此，C，C++和 Java 的程序员能很快熟悉它。C#获得了 ECMA 和 ISO/IEC 的国际标准认证，它们分别是 ECMA-334 标准和 ISO/IEC 23270 标准。Microsoft 用于.NET 框架的 C#编译器就是根据这两个标准实现的。本章力求对 C#做一个简单明了的说明，目的是向各位摩丝提供对 C#语言的入门介绍，以便于能够快速上手编写程序。

3.1　这世界，我来了

几乎所有编程语言的第一课都是“Hello World”，MOL 也只好落一下俗套。这一节的内容是第一段程序，输出本土化的 Hello World。

3.1.1　Hello World 程序编写

打开 VS 后，我们先新建一个控制台应用程序，并命名为 LanguageTest（如图 3-1 所示），和第 1 章我们写的程序有所不同，新建的时候，模板要选择 Windows 和“控制台应用程序”。单击“确定”按钮，进入编程环境。

单击确定按钮之后，VS 会自动打开 Program.cs 文件，这个文件是“控件台应用程序”的主程序文件，它提供了程序的入口函数 Main（如图 3-2 所示），在代码第 13 行的位置写两句代码：

```
01  Console.WriteLine("这世界，我来了");
02  Console.ReadLine();
```

加入代码后，按下 F5 键运行程序，运行后的结果如图 3-3 所示。

3.1.2　Hello World 程序解析

大家可以看到，我们只写了两句话，这个程序就可以达到预期的要求。那么，这个程序里的代码到底是什么意思呢？听 MOL 给你细细道来。

```
using … （第 1-5 行）
```

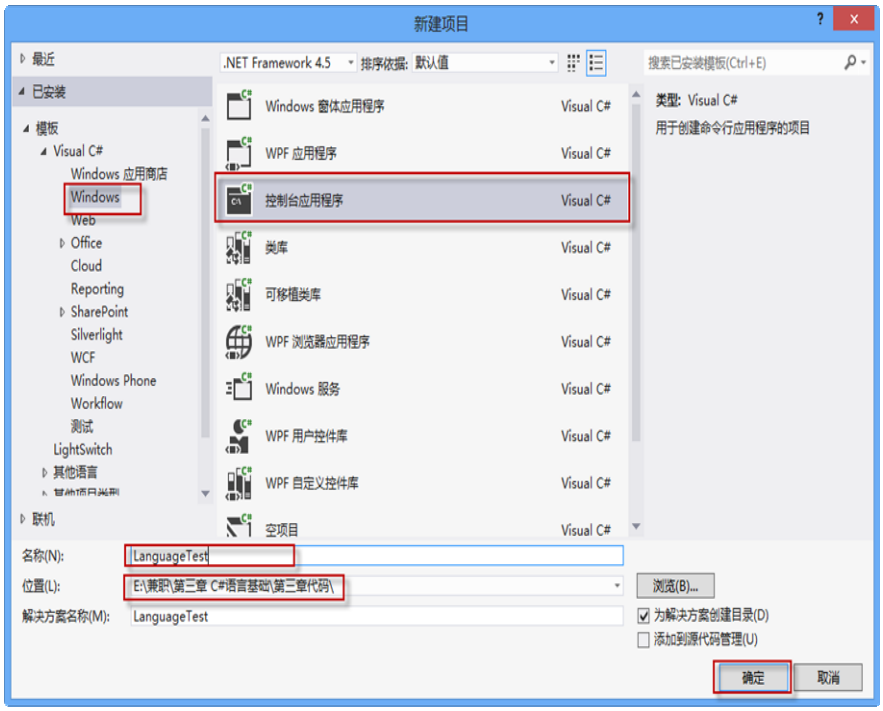

图 3-1　新建控制台应用程序

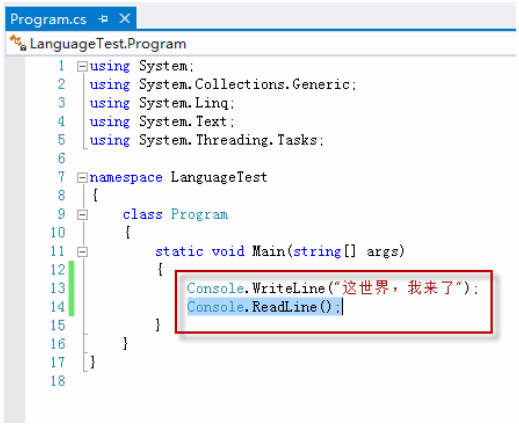

图 3-2　加入代码

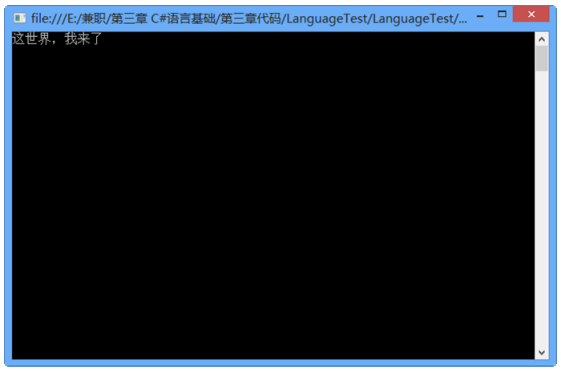

图 3-3　运行结果

在代码的最前面，是好几行 using 语句，这些 using 用来引入命名空间（namespace），相当于 C++中的 include 或者 java 中的 import。

```
namespace LanguageTest （第 7 行）
```

这是我们自己定义的一个命名空间。

```
Class Program（第 9 行）
```

声名一个类，名字叫 Program。这个程序所做的事情就是依靠它来完成的。

第 11 行，声明一个 Main()方法，用来作为程序的入口。

说明：C#程序的执行，总是从 Main()方法开始的。一个程序中不允许出现多个 Main()方法。

第 13 行调用 Console 类的 WriteLineo()方法输入一句话：“这世界，我来了”

第 14 行只是一个辅助行，但是如果没有这句话，运行的时候只会看到结果一闪而过。所以这句话的意思就是让程序等待，直到用户输入回车。

3.1.3　Hello World 程序总结

前面我们演示了一个简单的控制台应用程序的编写，并成功输出了“这世界，我来了”这样一句话。而且，我们也明白了程序中第一句话的意思。在接下来的几节中，我们将用控制台应用程序来讲解 C#的基础知识。

3.2　C#数据类型

C#是强类型语言，因此每个变量和对象都必须具有声明类型。所谓强类型语言，也称为强类型定义语言，它要求所有变量都必须先定义后使用。Java、.NET、python、C++等都是强制类型定义的。强类型已经强到了令人发指的地步，举例来说，你定义一个整数类型的变量，那么在你对它强制转换以前，它永远都是整数类型。而弱类型语言的数据类型可以被忽略，你定义一个变量，这个变量可以被赋值为整数，在程序执行的过程中还可以被赋值为其他类型。也就是一个变量可以赋不同类型的数据值。

C#语言的数据类型包括值类型和引用类型。值类型放在内存栈中，而引用类型放在内存堆中；在内存中存放的内容也不一样，值类型存放的是变量本身，而引用类型存放的是变量的引用，相当于 C++中的指针。这样说可能读者不太清楚，MOL 打个简单的比方：值类型相当于现金，你拿了就可以花；引用类型相当于银行卡，你必须把现金取出来才可以消费，所以值类型比引用类型执行得更快一点。

3.2.1　C#值类型

MOL 有个朋友是做销售工作的，他的工资水平浮动比较大，多则好几万，少则两千，但是他的工资只能用数目来描述，所以当我们把工资看成是一个变量的时候，它的类型就确定了，只能是数字类型。在 C#中，这种类型就是值类型数据。那么 C#中的值类型有哪些呢？

C#的值类型可以归类如下：

- 简单类型（Simple types ）；
- 结构类型（struct types）；
- 枚举类型（Enumeration types）。

3.2.2　简单类型

C#中的简单类型都有一些共同的特点：它们都是.NET 系统类型的别名；由简单类型组成的常量表达式仅在编译时而不是运行时受检测；简单类型可以按字面被初始化。下面我们简单说一下这些数据类型。

1．整型

C#中有 9 个整型，分别是 sbyte、byte、 short、 ushort、 int、 uint、 long、 ulong 和 char（单独一节讨论）。它们具有以下特性，如表 3-1 所示。

表 3-1　整型数据类型

类　　型	长　　度	取 值 范 围	符号
sbyte	8 位	–128～127	有
byte	16 位	0～255	无
short	16 位	–32768～32767	有
ushort	16 位	0～65535	无
int	32 位	–2 147 483 648～2 147 483 647	有
uint	32 位	0～4 294 967 295	无
long	64 位	－9 223 372 036 854 775 808～9 223 372 036 854 775 807	有
ulong	64 位	0～18 446 744 073 709 551 615	无

如果有哪位摩丝是从 VB 或 C 语言转过来的，那你一定要注意了。int 不再取决于一个机器的字（word）的大小，而 long 被设成 64 位。

2．布尔型

布尔数据类型有 true 和 false 两个布尔值，可以赋予 true 或 false 值一个布尔变量，或可以赋予一个表达式，其所求出的值等于两者之一：

```
bool boolMol=true;  //定义一个布尔类型的变量并赋值为 true
bool boolMol=(1>2); //定义一个布尔类型的变量并为其赋值，它的值是表达式 1>2 的结果
```

与 C 和 C++相比，在 C#中，bool 值不再为任何非零值。不要为了增加方便而把其他整型转换成布尔型，即不可以用 5.9 等非零的数字来描述一个 bool 值。

3．字符型

字符型为一个单 Unicode 字符。一个 Unicode 字符 16 位长，它可以用来表示世界上多种语言。可以按以下方法给一个字符变量赋值：

```
char charMol='a';
```

除此之外，可以通过十六进制转义符（前缀\x）或 Unicode 表示法给变量赋值（前缀\u）：

```
char charMol='\x0066';
```

说明：这句话中，我们使用了转义字符“\”，C#中定义了一些字母前加"\"表示常见的那些不能显示的 ASCII 字符，如\0,\t,\n 等，就称为转义字符，因为后面的字符，都不是它本来的 ASCII 字符意思了。关于转义字符，参见表 3-2。

不存在把 char 转换成其他数据类型的隐式转换。这就意味着，在 C#中把一个字符变量当作另外的整数数据类型看待是行不通的，这是 C 程序员必须改变习惯的另一个方面。但是，可以运用显式转换：

```
Char chSomeChar = (char)65;     //定义一个字符类型的变量，它的 ASCII 值为 65
Int SomeInt=(int)'A';           //定义一个字符类型的变量，它的值为 A
```

表 3-2　转义字符

转 义 字 符	含　　意
\’	单引号
\”	双引号
\\	反斜杠
\0	空
\a	警告（产生峰鸣）
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

4．浮点型

有两种数据类型被当作浮点型：float 和 double。它们的差别在于取值范围和精度。

- float：取值范围在 1.5x10^-45～ 3.4x10^38 之间，精度为 7 位数。
- double：取值范围在 5.0x10^-324 ～ 1.7x10^308 之间，精度为 15～16 位数。

当用两种浮点型执行运算时，可以产生以下的值：正 0 和负 0（从正数和负数两个方向无限趋近于 0），正无穷和负无穷，非数字值（Not-a-Number，缩写 NaN），非零值的有限数集；另一个运算规则为，当表达式中的一个值是浮点型时，所有其他的类型都要被转换成浮点型才能执行运算。

5．小数型（The decimal Type）

小数型是一种高精度、128 位数据类型，小数类型经常用于金融和货币的计算。它所表示的范围从大约 1.0x10^-28 到 7.9x10^28，具有 28 至 29 位有效数字。要注意，精度是以位数（digits）而不是以小数位（decimal places）表示，运算准确到 28 个小数位的最大值。

正如你所看到的，小数类型的取值范围比 double 的还窄，但它更精确。因此，没有 decimal 和 double 之间的隐式转换——往一个方向转换可能会溢出，往另外一个方向可能会丢失精度。你不得不运用显式转换。

当定义一个变量并赋值给它时，使用 m 后缀以表明它是一个小数型：

```
decimal decMyValue = 1.0m;              //定义一个小数类型的变量，赋值为 1.0
```

如果省略了 m，在变量被赋值之前，将被编译器认作为 double 型。

6．结构类型

一个结构（struct）类型可以声明构造函数、常数、字段、方法、属性、索引、操作符和嵌套类型。尽管列出来的功能看起来像一个成熟的类，但在 C#中，结构和类（class）的区别在于结构是一个值类型，而类是一个引用类型。与 C++相比，这里可以用结构关键字定义一个类。

使用结构的主要思想是用于创建小型的对象，如 Point 和 FileInfo 等。你可以节省内存，因为没有如类对象所需的那样有额外的引用产生。例如，当声明含有成千上万个对象的数组时，这会引起极大的差异。

下面的代码给大家示范一下，如何用一个结构来表示一个人的名字。

```
Struct week
{
    Public string firstname;                          //姓
    Public string secondname;                         //名
    Public week(string inputfir,string inputsec)   //赋值方法
    {
        firstname=inputfir;
        secondname=inputsec;
    }
}
```

7．枚举类型

当你想声明一个由指定常量集合组成的独特类型时，枚举类型正是你要寻觅的。最简单的形式，它看起来可能像这样：

```
enum MonthNames {January,February,March,April};
                                                    //定义一个枚举，包含了 1～4 月
```

默认情况下，枚举元素是 int 型，且第一个元素为 0 值。每一个连续的元素按 1 递增。如果想给第 1 个元素直接赋值，可以按如下方式把它设成 1：

```
enum MonthNames{January=1,February,March,April};
                                          //定义一个枚举，并设置 January 的值为 1
```

如果想赋任意值给每个元素——甚至相同的值，这也没有问题：

```
enum MonthNames { January=31, February=28, March=31, April=30 };
                                          //给枚举中的每个元素赋值
```

最后的选择是不同于 int 的数据类型。可以在一条语句中如此赋值：

```
enum MonthNames:byte{January=31,February=28,March=31,April=30};
```

说明：你可以使用的类型仅限于 long、int、short 和 byte。

3.2.3　引用类型

和值类型相比，引用类型不存储它们所代表的实际数据，但它们存储实际数据的引用。

在 C#中引用类型包括：

- 对象类型；
- 类类型；
- 接口；
- 字符串类型；
- 数组。

1．对象类型

对象类型是所有类型之母——它是其他类型最根本的基类。因为它是所有对象的基类，所以可把任何类型的值赋给它。例如，一个整型：

```
Object mol=123;
```

2．类（class）类型

类（class）类型是一个很神奇的类型，主要有属性和方法两个元素。属性就是这个类所包含的一些数据类型的变量，方法就是这个类的动作。举个例子，我们定义一个 MOL 的 class，MOL 有姓名、体重、性别，MOL 还会吃，还会编程。那么 MOL 这个类里面包含的属性有姓名、体重、性别。方法有吃、编程。

类还有构造函数和析构函数，构造函数就是用来开辟一块内存来放类的实例，析构函数就是把这块内存销毁。

类（class）和结构（Struct）这两种数据类型是很相似的，它们最主要的区别是类是引用类型，而结构是值类型。

类是面向对象的基础，在这里，MOL 以实例代码说明。下面的代码声明一个表示用户的类。

说明：面向对象并不是本书的重点，所以不会以太多的文字去说明，希望大家能利用各种资源去了解一下关于面向对象的知识。

```
Public class User
{
    String username;                    //用户名
    String password;                    //密码
    Bool sex;                           //性别
    Int age;                            //年龄
    Public GetuserName()
    {
        Console.writeLine(username);    //输入用户名
    }
}
```

3．接口

接口是一种引用类型，它只声明了抽象成员。说白了，就是只有声明没有实现，就好像一篇文章的大纲，却没有内容。每个人都可以用这些大纲写不同的文章。程序员可以使用接口，并且实现接口里声明的抽象成员。

注意：接口是不能实例化的。

接口中可以包括方法、属性和索引。它和类有什么不同呢？当定义一个类的时候，你可以派生自多个接口，但不能继承多个类。

那么，接口有什么用呢？比如一个接口派生了多类，那么我们只要知道接口里的方法，就可以通过任何一个继承自它的类去调用这些方法。

下面，我们用代码来说明接口的用法。

```
interface Iface                          //声明一个接口
{
string GetMyName();                      //声明接口里的方法
}
class Mol:iface                          //定义一个类 MOL MOL 继承自接口 iface
{
    string MolName="mol";                //定义一个类的属性用
    public string GetMyName()            //实现接口中的方法
    {
        return MolName;
    }
}
class ZhuGangLie:iface                   //定义一个类,这个类继承自接口 iface
{
        string ZhuName="猪刚鬣";          //姓名
        public string GetMyName()        //实现接口中的方法
        {
            return ZhuName;
        }
}
//调用方法
Mol person=new Mol();                //定义一个 Mol 类型的对象
console.WriteLine(person.GetMyName());
                                     //调用 GetMyName()方法，返回字符串并输出
ZhuGangLie pig=new ZhuGangLie();     //同上
console.WriteLine(pig.GetMyName());//同上
```

4．字符串类型

字符串数据类型当然是用来保存字符串喽。它继承自 Object，而且是被密封的，也就是给它做了个绝育手术，它不能再被继承了。它的用法十分简单：

```
string myName="Mol";             //声明一个字符串类型的变量并为其赋值
string AddStr="Mol "+"Love "+"Our country";
                                 //把 3 个字符串拼接成一个字符串并放在 AddStr 中
char word=AddStr[1];             //取出 AddStr 的第 2 个字符
if(myName==AddStr)               //判断两个字符串是否相同
{
    Console.WriteLine("这两个字符串相同");
}
else
{
    Console.WriteLine("这两个字符串不同");
}
```

5．数组

数组是一个可以容纳多个元素的类型，但是要容纳的元素必须是同一类型。如定义了一个 int[]的数组，那么它里面就只能放整数类型的元素了。

数组一个比较重要的概念就是“维数”，也就是我们经常提到的几维数组。它和我们线性代数中学的矩有的相似的地方。一维数组，就是数组中只放了一行数据；二维数组，就是数组中存放了多行多列……。需要注意的是，数组的维下标是从 0 开始的。如有一个 3 行 2 列的二维数组 Arr，要取它的第一行第二个数据的方法是：Arr[0][1]，0 表示第一行，1 表示第二列。

定义数组的方法有以下几种：

```
int[] intArr={"1","3","5"};
int[] arr=new int[]{1,2,3,4,5,6};                    //不定长
int[] arr2 = new int[3]{1,2,3};                      //定长
int[,] arr3 = new int[,]{{1,2,3},{1,2,3}};           //不定长
int[,] arr4 = new int[2,2]{{1,2},{1,2}};             //定长
//调用方法
Console.WriteLine(arr[0]+ "");
```

3.2.3　装箱和拆箱

由上面所讲的内容，我们可以知道，C#中所有的数据类型都是继承自 Object，所以，值类型和引用类型的值可通过显式或隐式操作相互转换。这个转换的过程就叫装箱和拆箱。

说明：显式转换通过表达式进行专一的数据类型转换操作，就是在“光天化日”之下进行的，我们可以看见代码。隐式转换没有专门的程序去进行类型转换，编译器帮我们做了转换的工作，这是“偷偷摸摸”进行的。我们看不见专门类型转换的代码。

1．装箱

装箱是值类型到 object 类型或到此值类型所实现的任何接口类型的隐式转换。对值类型装箱会在堆中分配一个对象实例，并将该值复制到新的对象中，如图 3-4 所示。

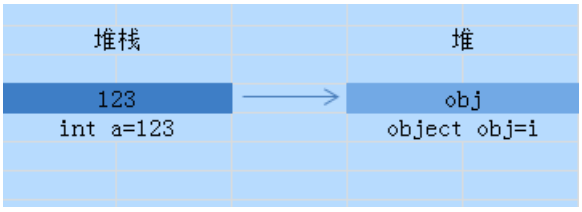

图 3-4　装箱

2．拆箱

拆箱（取消装箱）是从 object 类型到值类型或从接口类型到实现该接口的值类型的显式转换。取消装箱操作包括：

首先检查对象实例，确保它是给定值类型的一个装箱值（拆箱后没有转成原类型，编译时不会出错，但运行会出错，所以一定要确保这一点。用 GetType().ToString()判断时一定要使用类型全称，如 System.String，而不要用 String）。然后将该值从实例复制到值类型变量中。拆箱的操作如图 3-5 所示。

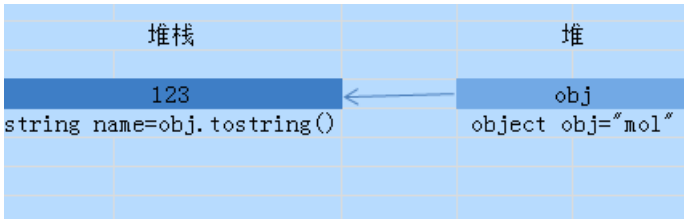

图 3-5　拆箱

说明：在进行拆箱操作的时候，一定要知道箱子里面放着的是什么类型，否则就会出错，或可能得到意想不到的结果。

3.3　变量与常量

在实际的编程中，经常需要用我们能看懂的字母或单词来描述内存中存放的数据，内存中的这些数据有些是一成不变的，有些会随着程序的执行它的值也发生变化。这些字母或者单词就是本节要讲的变量与常量。

3.3.1　常量

常量在应用程序的整个生命周期中从一而终，保持着同一个值不变。

定义常量的方法：

```
const string WriterName="Mol"; //定义一个常量，它是字符串类型，并为其赋值为 Mol
const int WriterAge=27;        //定义一个整型的常量，为它赋值为 27
```

常量在声明的时候就必须赋值，否则编译时就会报错。

如果在程序中有一些值经常会用到，那么就可以把它定义成常量，如圆周率、自然对数等。

3.3.2　变量

变量是指在程序运行过程中其值可以发生变化的量，定义变量的方法如下：

数据类型 变量名；

数据类型 变量名＝初值；

```
string testStr;                    //定义一个字符串变量
int testInt=0;                     //定义一个整型的变量
```

下面，我们通过一个程序来说明常量和变量的用法。这个程序的功能是：输入一个数，做为圆的半径，输入所对应的圆的面积。

```
const double pi = 3.14d;                          //定义一个常量 pi 为圆周率
static void Main(string[] args)                   //入口函数
{
    Console.WriteLine("请输入半径：\n\r");          //输出提示信息
    double r =Convert.ToDouble( Console.ReadLine());
                                                  //读取输入的数字并转换为 double 类型
   double re = pi * r * r;                        //计算圆面积
    Console.WriteLine("根据您输入的半径，计算出圆的面积是："+re);  //输入圆面积
    Console.ReadLine();
}
```

程序运行结果如图 3-6 所示。

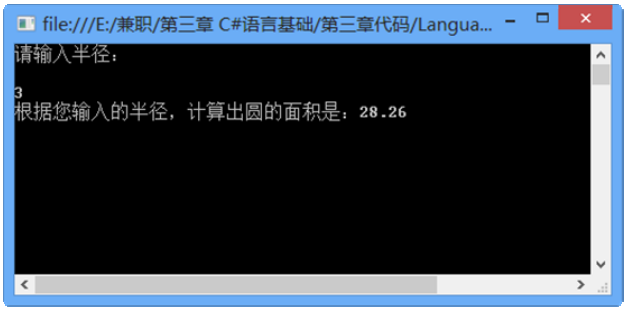

图 3-6　程序运行结果

注意：变量有自己的作用范围，超出了这个范围，就不能再使用这个变量了。如 A 语句块中定义一个变量 i，B 语句块中就不能对 i 进行访问。

3.4　C#中的表达式

知道了常量和变量，就算是在 C#编程基础的万里长征上走出了第一步，变量和常量如何使用？变量在程序里如何按照程序员的意愿得到相应的值？这就需要表达式和运算符一起来配合实现了。表达式与运算符是怎样的存在？听 MOL 给你细细道来。

表达式是由运算符和操作数组成的，如下代码所示：

```
int i=0;                                  //定义一个整型变量
i++;                                      //自加运算
string name="钓鱼岛";                      //定义一个初始值是"钓鱼岛"的字符串变量
name+="是中国的";                          //字符串连接
console.WriteLine(name);                  //输出
```

上面的 i 就是一个操作数，而且它是一个值类型的操作数，name 也是一个操作数，++、=、+=是运算符，name+="是中国的";就是一个表达式，因为在这一行里包括了操作数和运算符。

操作数就是我们前面所讲的常量和变量。那么运算符是什么呢？这是本节的重点。

3.5 运　算　符

运算符是一种专门用来处理数据运算的特殊符号，它非常重要，小到计算器，大到大型网站都离不开运算符。运算符主要分为：

- 算术运算符；
- 逻辑运算符；
- 字符串连接运算符；
- 增量和减量运算符；
- 移位运算符；
- 比较运算符；
- 赋值运算符；
- 成员访问运算符（用于对象和结构）；
- 索引运算符（用于数组和索引器）；
- 数据类型转换运算符；
- 条件运算符（三元运算符）；
- 委托连接和删除运算符；
- 对象创建运算符。

3.5.1 算术运算符

说到算数运算符，首先可以想到数学中一些常用的运算符，例如加、减、乘、除、求余。在 C# 这些运算符表示如表 3-3 所示。

表 3-3　C#中的算术运算符

运算符名称	C#中的符号
加	+
减	-
乘	*
除	/
取余	%

需要注意的是，算术运算符两边的数据类型最好相同，如果不同，会自动进行类型转换，而且是由低精度到高精度的转换。如：

```
double re=2+3.14d;              //定义 double 类型的变量，它的值是 2+3.14d 的结果
```

在上面的代码中，2 是整数类型，3.14d 是 double 类型，它们在进行加法运算的时候，首先把 2 转换成 double 类型，再和 3.14d 进行加运算，那么结果就是 5.14d。

下面的代码演示了四则运算的方法。

```
01  int addRe = 3 + 5;                  //定义一个整数类型的变量
```

```
02  Console.WriteLine("3+5 的结果是：" + addRe + "\n\r");
03  double jianRe = 90 - 56;        //定义一个 double 变量 值为 90-56 的结果
04  Console.WriteLine("\n\r90-56 的结果是：" + jianRe + "\n\r");
05  int MulRe = 4 * 8;              //定义一个整型变量 值为 4 和 8 的积
06  Console.WriteLine("4*8 的结果是：" + MulRe + "\n\r");
07  int decRe = 48 / 6;             //整型变量，值为 48 除 6 的整数部分
08  Console.WriteLine("48/6 的结果是：" + decRe + "\n\r");
09  int relRe = 50 % 6;             //整型变量，值为 50 除 6 的余数
10  Console.WriteLine("50/6 的余数是：" + relRe + "\n\r");
```

3.5.2　逻辑运算符

逻辑类型的运算符是最贴近生活的，我们在生活中经常会遇到诸如“如果明天不下雨并且上课的老师是美女，那么我不逃课”之类的情景。这里的“并且”就是逻辑运算里的一种。

下面我们要讲的逻辑运算有与、或、非、异或。C#中的表示如表 3-4 所示。

表 3-4　逻辑运算符

运算符名称	C#中的符号
与	&、&&
或	|、||
非	!
异或	^

这些运算符和我们在数理逻辑里的逻辑运算是一样的。

- &和&&：当且当运算符两边都为真时，结果为真。如果有一边为假，则结果为假。
- |和||：运算符两边只要有一边是真，则结果为真。
- !：这是个一元运算符，也就是说它这个运算符只能对一个元素进行操作，如：!a 就是对 a 进行取非运算。当 a 为真时，结果为假，当 a 为假时，结果为真。
- ^：这个操作符我们一般用不到，它的运算方法是当两边同为真或同为假时，结果为假；运算符两边 BOOL 值不一样时，结果为真。简称为“同假异真”

下面，MOL 列出了各种情况的运算结果，如表 3-5 所示。

表 3-5　逻辑运算符取值表

a	b	a&b、a&&b	a|b、a||b	!a	a^b
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	true
false	false	false	false	true	false

有些细心的摩丝一定发现了，“与”运算和“或”运算有两种表达式，那么这两种表达式到底有什么区别呢？

对于与运算来说，&和&&是两边的表达式都为真时，结果为真。不同之处只区别于一种情况，就是当运算符左边的表达式为 false 时，&&会直接返回 false。而&还要对右边的表达式进行扫描，再返回 false。

对于“或”运算来说，“|”和“||”都是运算符两边只要有 true 就返回 true。不同之外也只有一种情况，当运算符左边的表达式为 true 时，“||”会直接返回 true，而“|”还要对右边的表达式进行扫描再返回 true。

这样看来，&&和“||”要比&和“|”强悍，MOL 推荐大家使用&&和“||”，这样程序性能会更好一些。

3.5.3　字符串连接运算符

字符串连接运算符是“+”，没错，你没有听错，是加号。这是个神奇的符号，它不仅可以进行算术加运算，还可以进行字符串连接。如我们定义两个字符串 a 和 b，再把这两个字符串进行连接并赋值给字符串 c，那么 c 最后的结果就是字符串 a 和字符串 b 的连接。代码如下：

```
string a="mol ";                    //定义字符串 a
string b=" love mol's fans";        //定义字符串 b
string c=a+b;                       //字符串 a 和 b 进行连接后赋值给字符串 c，
                                    //结果为 mol love mol's fans
```

有一种特殊情况要注意，当“+”两边只要有字符串参加运算时，一律当成是字符串连接运算，如：

```
string strA="Mol have"              //定义字符串变量
int count=10000;                    //定义整型变量
string strB=" fans";                //定义第二个字符串变量
string re=strA+count+strB;
                    //+号表达式中有字符串参加运算，所以结果 Mol have 10000 fans
```

3.5.4　自增和自减运算

本节所讲的运算符，在大型的程序中用得很多。所谓自增，就是自动累加的意思，在 C#中就是自动加 1，依次类推，自减就是自动减 1。下面我们以示例代码来说明。

```
int a = 1;                  //定义一个整数类型变量 a
a++;                        //后置++
Console.WriteLine("1 的自增结果为"+a);
int b = 1;                  //定义一个整数类型变量 b
++b;                        //前置++
Console.WriteLine("1 的自增结果为" + b);
int c = 10;                 //定义一个整数类型变量 c
c--;                        //后置--
Console.WriteLine("10 的自减结果为" + c);
int d = 10;                 //定义一个整数类型变量 d
--d;                        //前置--
Console.WriteLine("10 的自减结果为" + d);
```

程序的运行结果如图 3-7 所示。

大家看到这里一定会纳闷了，在上面的代码中，自增自减运算符放在表达式前面和后

面的运算结果一样啊，那这两种形式是不是就是等价的呢？当然不是了，以自增为例来说明。++放在表达式前面，表示先对表达式进行自加运算，再使用表达式进行其他操作；而++放在表达式的后面，表示先使用表达式进行其他操作，再对表达式进行自加运算，如下面代码所示。

```
01  int testa = 1;
02  Console.WriteLine("输出 testa++:"+(testa++));
03  Console.WriteLine("输出 testa:"+testa);
04  int testb = 1;
05  Console.WriteLine("输出++testb"+(++testb));
06  Console.WriteLine("输出 testb:"+testb);
```

输出结果如图 3-8 所示。

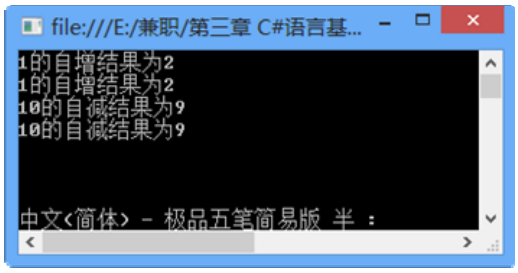

图 3-7　自增自减运算结果

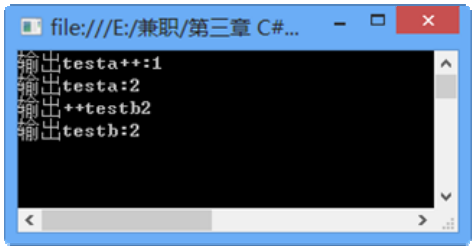

图 3-8　前置自增与后置自增的输出

代码解释：

第 2 行 Console.WriteLine(testa++);这句话是先把 testa 输出到屏幕上（所以屏幕上首先要输出 1），再对 testa 进行自加运算。相当于 console.WriteLine(testa)；testa=testa+1;这两句。显然，testa 在运算之后已经变成了 2。

第 3 行 Console.WriteLine(testa)就是把 testa 输出到屏幕上。经过上面的运算，testa 已经变成了 2，所以这个时候输出的就是 2。

第 5 行 Console.WriteLine(++testb);这句话是先对 testa 进行自增，然后再输出 testb，所以屏幕上会输出 2。相当于 testb=testb+1;console.WriteLine(testb);这两句。

第 6 行 Console.WriteLine(testa);由于 testa 的值没有再进行改变，所以输出到屏幕上还是 2。

由上面的代码和输出结果可以总结出，当自增运算符在前面时，就先进行自增运算再进行其他操作。当自增运算符在后面时，就是进行完其他操作时再进行自增运算。同理，自减运算也是一样的道理，MOL 在这里就不啰嗦了。

3.5.5　移位运算符

移位运算符有两种，分别是左移<<和右移>>。

左移运算<<就是将第一个操作数向左移动第二个操作数所指定的位数。第二个操作数的类型必须是 int 类型或 uint 类型的。右移运算就是将第一个操作数向右移动第二个操作数所指定的位数。第二个操作数的类型必须是 int 类型或 uint 类型的。

这里所说的移位是针对二进制数来说的，如果不是二进制数，编译器会自动进行进制

转换再进行移位。如：2<<3;表示将十进制的 3 进行左移两位，先把 3 转换成二进制为 11，再进行左移两位，结果是二进制的 1100，换算成十进制就是 12。所以 2<<3 的结果是 12。右移运算同理。

3.5.6　比较运算符

和我们在数学中所学的一样，C#中两个对象进行比较，无非是这样几种情况：等于、不等于、大于、小于、大等于、小等于。

在 C#中，判断两个对象是否相等的符号是“==”即两个等号。如果==两边的对象相等，则返回 true，如果不相等，则返回 false。这个地方是初学者容易忽略的地方，很有可能会和后面讲到的赋值运算符“=”混淆。

- !=是判断两个对象是否不等。如果两个对象不等，则返回 true，如果两个对象相等，则返回 false。
- <是判断第一个数是否小于第二个数，如果第一个数小于第二个数，则返回 true，否则返回 false。
- >是判断第一个数是否小于第二个数，如果第一个数小于第二个数，则返回 true，否则返回 false。
- >=是判断第一个数是否大于等于第二个数，如果第一个数大于等于第二个数，则返回 true，否则返回 false。
- <=是判断第一个数是否小于等于第二个数，如果第一个数小于等于第二个数，则返回 true，否则返回 false。

以下代码演示如何使用逻辑运算符。

```
01  bool re;
02  int inta = 16;              //定义一个 int 类型的操作数 inta
03  int intb = 28;              //定义一个 int 类型的操作数 intb
04  re = (inta == intb);
                //判断 inta 和 intb 是否相等，如果相等，返回 true，否则返回 false
05  Console.WriteLine(re);
06  re=(inta!=intb);
                //判断 inta 和 intb 是否不等，如果不等，返回 true，否则返回 false
07  Console.WriteLine(re);
08  re=(inta>intb);
                //判断 inta 是否大于 intb，如果大于，返回 true，否则返回 false
09  Console.WriteLine(re);
10  re=(inta<intb);
                //判断 inta 是否小于 intb，如果小于，返回 true，否则返回 false
11  Console.WriteLine(re);
12  re=(inta<=intb);
                //判断 inta 是否小于等于 intb,如果小于等于,返回 true,否则返回 false
13  Console.WriteLine(re);
14  re=(inta>=intb);
                //判断 inta 是否大于等于 intb,如果大于等于,返回 true,否则返回 false
15  Console.WriteLine(re);
```

程序运行的结果如图 3-9 所示。

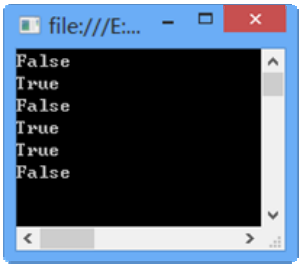

图 3-9　逻辑运算结果

3.5.7　赋值运算符

赋值运算符最常见的就是等号“=”，它表示将运算符右边的表达式的值存储在左边的操作数上。

下面所介绍的赋值运算符，本质就是算术运算符和等号运算符的结合。

- “+=”运算符，表示“+=”左边的操作数与右边的操作数进行加运算，再赋值给左边的变量。如 a+=b 就表示先对 a 和 b 进行加运算，再把结果赋值给 a。相当于 a=a+b。
- “-=”运算符，表示“-=”左边的操作数减去右边的操作数，再把结果赋值给左边变量。如 a-=b 就相当于 a=a-b。
- “*=”运算符，表示“*=”左边的操作数乘以右边的操作数，再把结果赋值给左边变量。如 a*=b 就相当于 a=a*b。
- “/=”运算符，表“示/=”左边的操作数除以右边的操作数，再把结果赋值给左边变量。如 a/=b 就相当于 a=a/b。
- “%=”运算符，表示“-=”左边的操作数对右边的操作数进行取余运算，再把结果赋值给左边变量。如 a%=b 就相当于 a=a%b。

示例代码如下：

```
01  int a = 10;
02  a += 1;          //对 a 进行+=赋值运算，结果为 11 并赋值给 a ,此时的 a 为 11
03  Console.WriteLine(a);
04  a -= 2;          //对 a 进行-=赋值运算，在运算之前 a 的值为 11，运算之后的结果为 9
                       并赋值给 a，此时的 a 为 9
05  Console.WriteLine(a);
06  a *= 3;          //对 a 进行*=赋值运算，在运算之前 a 的值为 9，运算之后的结果为 27
                       并赋值给 a，此时的 a 为 27
07  Console.WriteLine(a);
08  a /= 4;          //对 a 进行/=赋值运算，在运算之前 a 的值为 27，运算之后的结果为 6
                       并赋值给 a，此时的 a 为 6
09  Console.WriteLine(a);
10  a %= 5;          //对 a 进行%=赋值运算，在运算之前 a 的值为 6，运算之后的结果为 1
                       并赋值给 a，此时的 a 为 1
11  Console.WriteLine(a);
```

运算结果如图 3-10 所示。

说明：下面所介绍的运算符，都不是入门级的，如果你看不懂，没有关系。以后我们会深入介绍的。

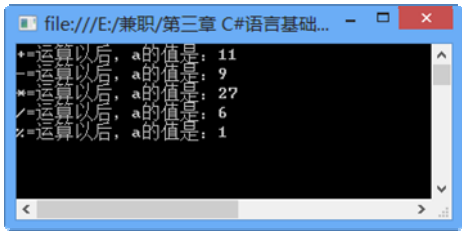

图 3-10　赋值运算结果

3.5.8　成员访问运算符

如果一个类（class）有自己的属性有方法，那么，我们要访问这个类的属性的方法的时候，就需要用到成员访问运算符“.”。如我们前面所用到的 Console.WriteLine("这世界，我来了。")；中的 Console 就是一个类，WriteLine()就是一个方法，这个方法可以在屏幕上输出一句话。我们要使用 WriteLine()方法的时候，就需要用到成员访问运算符了，格式为：类名.方法名/属性名。

3.5.9　索引运算符

索引运算符[]是针对数组和集合来操作的，如，[i]中的数字 i 表示的是下标或索引字符串。这就好比在一个走廊里有好几个房间，你要找到特定的房间，你可以说：“我要找第 3 个房间”，或者说“我要找门牌号是 2203 的文章”。这里的“第 3 个”中的 3 就是数字下标，而门牌号就是索引字符串。需要注意的是，当使用数字下标的时候，下标是从 0 开始的，如数组中的第一个元素下标是 0。数字下标索引经常用于数组和集合中，字符串索引经常用于集合中。

我们经常会在网址中传递参数，如网址为：http://192.168.1.28:7685/login.aspx?username=mol&password=molspassword，我们传递了两个参数 username 和 password。那么，我们在后台取这些参数的时候就需要用字符串索引。方法是：string username=Request.QueryString["username"]; ，取到传递的 username 的值。

数字下标索引的示例代码如下：

```
string[] tmp = { "字符串 1", "字符串 2", "字符串 3", "字符串 4", "字符串 5", "字符串 6", "字符串 7", "字符串 8", "字符串 9", "字符串 10" };    //定义一个字符串数组
string strFir = tmp[1];                                                    //取数组的第二个元素
Console.WriteLine("数组的第二个元素是："+strFir);
```

3.5.10　数据类型转换运算符

我们在编程的时候，经常会遇到不同类型的数据进行运算，有时编译器会自动对不同类型的操作数进行类型转换，这种转换叫隐式转换；有时编译器不能对类型进行转换，这时我们就需要用数据类型转换运算符()进行转换，这种转换叫显式转换。格式为：目的变量=（类型）源操作数。这样，我们就把源操作数转换成了需要的类型，如：

```
int a=(int)232.1f;          //把海战类型的 232.1 转换成整数类型，结果为 232
```

还有一种类型转换的方法，就是调用 Convert 类的方法。这个类有很多的方法用来进行类型转换，我们要用 Convert 的类型转换，实现把字符串“232”转换为整型的 232 应该

这样写：

```
int a=Convert.Toint32("232");            //字符串转整数
```

另外，每种类型都有自己的 Parse 方法，用来把字符串类型转换成自己的类型，如

```
double d=double.Parse("58");             //把字符串"58"转换成 double 类型
```

3.5.11　条件运算符

条件运算符是 C#中唯一的一个三元运算符，也就是说条件运算符要连接三个表达式。它的符号是?:，使用格式为：条件成立？返回结果 1：返回结果 2；意思是当条件成立的时候返回结果 1，否则返回结果 2。示例代码如下：

```
int re=(3-1)>0?10:-10;
```

这个表达式中，(3-1)>0 是条件，当它成立的时候返回 10，当它不成立的时候返回-10；相当于

```
int re;
if(3-1>0)                //如果 3-1>0
{
    re=10;               //给 re 赋值为 10
}
else                     //如果 3-1>0 不成立
{
    re=-10;              //给 re 赋值为-10
}
```

3.5.12　委托连接和删除运算符

大家都知道，在事件驱动的程序编写中，事件的发生会驱动程序的运行，如鼠标的单击会触发程序运行，那么程序是怎么知道鼠标单击要进行什么操作呢？比如我们想要单击按钮时，页面背景变灰，就可以这样写：按钮名称.单击事件+=（背景变灰的方法）；这样就注册好了一个事件，当我们单击按钮的时候，背景就会变灰。

移除事件用“-=”来进行，如按钮已经有了单击事件，现在想要移除按钮的单击事件，那么就可以这样写：按钮名称。单击事件-=（原有的方法）；这样，在单击按钮的时候就不会驱动任何程序运行。

3.5.13　对象创建运算符

在面向对象的编程中，经常需要创建一个对象，这个时候，我们就用到了 new 操作符，它的作用是开辟一块内存空间，这块内存空间存放了新建的一个对象。用法为：类型名 对象=new 类型名(); 。

到这里，C#的表达式与运算符就基本介绍完了，为什么是“基本”介绍完了呢？因为还有一些不常用到的运算符 MOL 并没有介绍，但是这并不影响我们的学习。本节学习完

以后，大家就对 C#中的表达式和运算符有一个感性认识。表达式与运算符是一颗颗的小珍珠，我们还需要一条条的线把它们串起来才能变成漂亮的项链，那么，这一条条的线是什么呢？MOL 接下来要给你介绍的就是它们，它们就是：流程控制语句。

3.6　流程控制语句

所谓语句就是构造所有 C#程序的过程构造块。语句可以声明局部变量或常量，调用方法，创建对象或将值赋于变量、属性、字段。就像我们前面经常用到的 Console.WriteLine(); 就是一条语句。C#中每一条语句都必须以分号结束。由一对大括号括起来的一系列语句叫代码块。

如果不进行任何控制，那么程序会按顺序从第一条语句执行到最后一条语句结束，然后退出程序。这显然是相当不实用的。我们需要应用程序可以在不同的情况下做出不同的反应，这样就需要进行流程控制。

C#中的流程控制语句有以下几类：

- 选择语句；
- 迭代语句；
- 跳转语句；
- 异常处理语句。

下面，我们对这几种类型进行一一讲解。

3.6.1　选择语句

选择语句用 if 或者 if…else 来表示。所谓选择语句，就是根据输入条件的布尔值来判断进入哪个分支，格式为：

```
if(输入条件)
{
    代码块 1;
}
else
{
    代码块 2;
}
```

当输入条件的布尔值为 true，即输入条件为真时，执行代码块 1，否则执行代码块 2。

也可以没有 else 分支，格式为：

```
if(输入条件)
{
    代码块 1;
}
```

当输入条件为真时，执行代码块 1，否则不作为。

下面，我们用 if…else 来实现一个功能，这个功能是：如果明天不下雨，并且上课的老师是美女，那么不逃课。

从这里开始，MOL 将带你把自然语言转化为程序语言，就是把一些自然语言的需求进行整理，找出输入，并输入想要的结果。

以上面所说的需求为例，我们先把这句话的名词都挑出来，挑出来的名字有：明天、老师、美女。这样看来，我们好像无法下手，因为你不能把一个名词赋一个布尔类型的值，布尔类型只能表示一个状态。所以我们要对这句话进行修改，当然修改的前提是意思不变。

修改后，需求可以表述为：如果明天不是不下雨的状态是真，而且上课老师是美女的状态是真，那么我的结果是不逃课。这样，我们再进行提取名词，就会得到：“不下雨的状态”、“老师是美女的状态”、“结果”。这样，我们就要定义 3 个变量用来描述我们找到的 3 个名词，这 3 个变量分别是：

```
bool tomrrowState;              //不下雨的状态
bool teacherIsPerty;            //老师是美女的状态
string re;                      //结果
```

其中：

- tomrrowState 用来表示明天是否下雨，当它为 true 的时候，表示明天下雨，如果为 false 则表示明天不下雨。
- teacherIsPerty 用来表示老师是否是美女，当它为 true 的时候，表示老师是美女，如果为 false 则表示老师不是美女。
- re 用来表示我的状态，这里我们用字符串来保存这个状态。

为了程序更贴近实际，这里的天气情况我们需要用户来输入，程序取到用户的输入再赋值给对应的变量。于是我们就用到了新的语句：Console.ReadLine()，用来获取用户在屏幕上的输入。如：

```
string tmp=Console.ReadLine(); //读取用户的输入
```

为什么不直接把用户的输入赋值给前面定义好的 teacherIsPerty 变量呢？因为我们读取的用户输入是字符串类型的，还需要根据用户不同的输入来给变量赋不同的值。我们可以提示用户，如果老师是美女，请输入 yes，如果不是，请输入其他字母。这样，我们就可以根据用户输入的字符串来判断老师是否是美女了。

```
if(tmp=="yes")                  //如果用户的输入是 yes
{
    teacherIsPerty=true;        //给变量赋值为 true，说明老师是美女
}
else
{
    teacherIsPerty=false;       //给变量赋值为 false
}
```

明天的天气情况的判断也是同样的道理，这样我们就解决了输入的问题了。

MOL 还要提示各位摩丝，一个好的程序，一定有友好的提示，比如，我们应该在用户输入之前就提示用户：请输入明天的天气状态，如果不下雨，请输入 yes，否则输入其他字符串。

我们得到了天气状况，也知道了老师是不是美女，那应该判断明天是否逃课，当且仅当明天不下雨，并且老师是美女这两个状态都为 true 时，我们给变量 re 赋值为：“我今天

不逃课”，如果有一个状态不为 true，那么给变量 re 赋值为“ZZZ……”。这样，我们就有了下面的代码：

```
if(tomrrowState==true&& teacherIsPerty==true)      //如果条件成立
{
    re="我不逃课";                                  //给结果赋值
}
else
{
    re="ZZZ……";                                    //给结果赋值
}
```

这样，整个程序就完成了。

【示例 3-1】看看明天能逃课不？

```
01  bool teacherIsPerty;                    //老师是否是美女
02  bool tomrrowState;                      //明天是否下雨
03  string re;                              //我的状态
04  Console.WriteLine("本程序实现了以下功能，如果明天不下雨，并且上课老师是美女，
    那么我不逃课\n\r");
05  Console.WriteLine("明天是否下雨？如果不下雨，请输入 yes，否则输入其他字符串，
    按回车结束输入\n\r");
06  string tmp = Console.ReadLine();
07  if (tmp == "yes")
08  {
09      tomrrowState = true;
10  }
11  else
12  {
13      tomrrowState = false;
14  }
15  Console.WriteLine("老师是否是美女？如果是，请输入 yes，否则请输入其他字符串，
    按回车结束输入\n\r");
16  tmp = Console.ReadLine();
17  if (tmp == "yes")
18  {
19      teacherIsPerty = true;
20  }
21  else
22  {
23      teacherIsPerty = false;
24  }
25  if (teacherIsPerty == true && tomrrowState == true)
26  {
27      re = "我不逃课";
28  }
29  else
30  {
31      re = "ZZZ……";
32  }
33  Console.WriteLine("根据明天的天气情况和老师的表现，我的状态是："+re);
```

我们测试的结果为：如果明天不下雨，而且老师是美女，输出结果如图 3-11 所示。

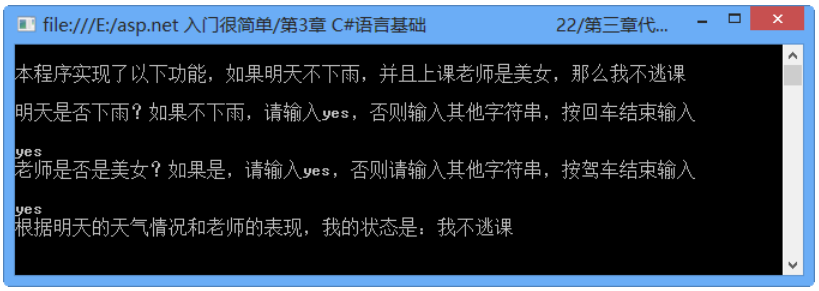

图 3-11　程序输出

如果明天不下雨，老师不是美女，输出结果如图 3-12 所示。

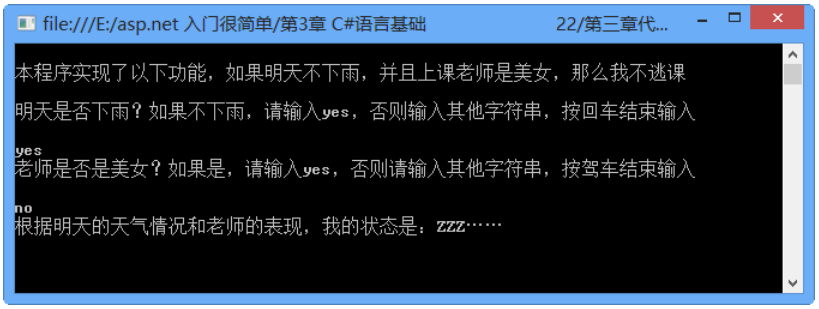

图 3-12　程序输出

说明：*程序源码位于光盘文件\源代码\第 3 章\代码 3-1。*

这个程序写完了，相信各位摩丝对选择语句也就入门了，没错，是入门了。因为简单的选择语句是可以变化多端的，如果程序需求很多，那么一个 if 语句就鞭长莫及了，这时就需要语句的嵌套。

所谓嵌套，就是在一个语句块中又包含了一个或多个语句块，对于 if 语句来说，是可以无限多次的嵌套的。下面，我们来做这样一个程序，这个程序的描述如下：

【示例 3-2】假设有 30 个摩丝来一睹 MOL 的尊容，但是 MOL 是个超级屌丝，居住的房间太小，只能一次接见 10 个人，那么摩丝们只能摇号了，第 1 号到第 10 号第一批进入房间，第 11 号到第 20 号，第二批进入房间，第 21 号到第 30 号第 3 批进入房间。每个摩丝不知道自己第几批进入房间，所以要把自己摇到的号输入，查看自己是第几批。

需求分析完了，我们就要进行自然语言到程序语言的转换，首先我们需要让摩丝输入自己的编号，再根据不同的编号返回第几批进入房间。实现的方法有很多，这里 MOL 所演示的是 if 嵌套的方法。

如果输入的编号介于 1～10 之间，那么输入第一批进入房间；否则，如果编号介于 11～20 之间，那么第二批进入房间，否则第三批进入房间。用程序语言来描述就是：

```
01  Console.WriteLine("请摩丝输入自己的编号，以回车结束输入\n\r");
02  string tmp = Console.ReadLine();    //定义一个变量 tmp 来接收用户的输入
03  int count = Convert.ToInt32(tmp);
                          //把用户输入的字符串转换成 int 类型以进行比较
04  if (count >= 1 && count <= 10)      //如果编号介于 1-10 之间
```

```
05  {
06      Console.WriteLine("请您在第一批进入\n\r");
07  }
08  else                                    //如果编号没有介于 1-10 之间
09  {
10      if (count >= 11 && count <= 20) //如果编号介于 11-20 之间
11      {
12          Console.WriteLine("请您在第二批进入\n\r");
13      }
14      else                                //其他情况
15      {
16          Console.WriteLine("请您在第三批进入\n\r");
17      }
18  }
19  Console.ReadLine();
```

程序运行结果如图 3-13 所示。

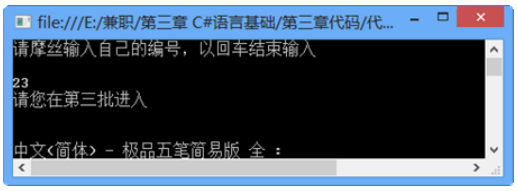

图 3-13　if 嵌套运算

上面的代码演示了如何使用 if 嵌套的方法。if 嵌套的方法不是最好的方法，却是最基础的方法，希望大家能熟练掌握。代码位置：光盘文件\源代码\第 3 章\代码 3-2。

3.6.2　选择语句 switch 分支语句

所谓 switch 语句就是一分支控制语句，它通过将控制命令传递给函数体内的一个 case 语句来处理多个选择和枚举。简单来说，就是当一个需求有很多种情况（情况到达 3 种以上）的时候，用 if…else 来实现就不太方便了。这个时候，switch 就派上了用场，它的使用格式如下：

```
switch(表达式)
{
    case "值 1":
        语句块 1;
        breadk;

    case "值 2":
        语句块 2;
        breadk;
…
    case "值 n":
        语句块 n;
        breadk;
    default:
        语句块 n+1;
        break;
}
```

它可以用一棵树来类比，树根是表达式，表达式取不同的值，程序就走不同的树枝，如图 3-14 所示。

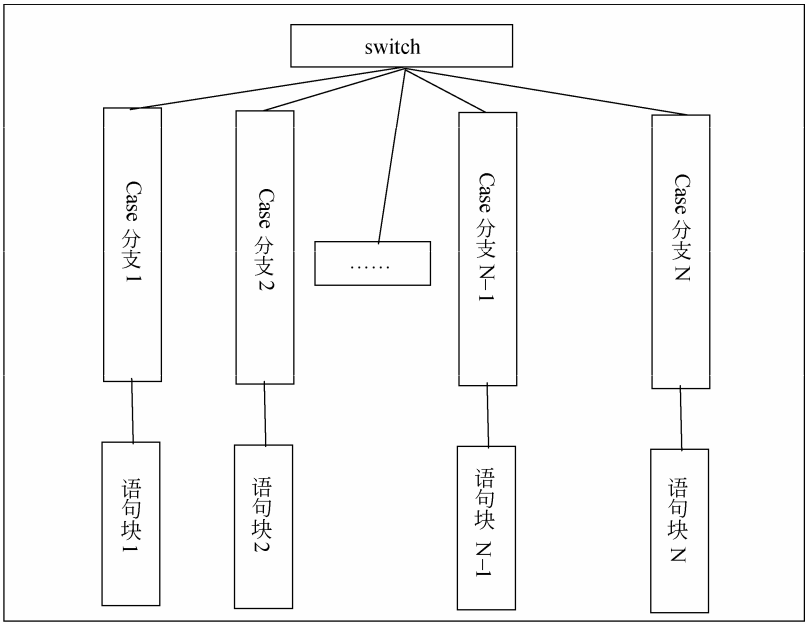

图 3-14　switch 图示

当表达式的值为“值 1”的时候执行语句块 1，当表达式的值为“值 2”的时候，执行语句块 2……当表达式的值不在我们所列举的值的范围内时，执行 default 分支。

大家可以看到，在上面的格式里，我们里引入了一个新的关键字 break，它的作用是使程序跳出分支，如果没有 break，那么程序会连同后面的语句块一起执行，这样显然不是我们想要的结果。

下面，我们通过一个例子来对 switch 加深认识。

【示例 3-3】根据用户输入的数字，在屏幕上打印出对应的英文星期。如用户输入 3，打印出 Wednesday。

我们先来分析一下需求。上面的需求完全可以用 if 来进行实现，但是这样需要写 7 个 if，而且当用户输入的数字不在 1～7 的范围内时，还需要多一个 if 进行判断，这样的代码显然是效率很低的，我们用 switch 来实现，代码如下：

```
01  Console.WriteLine("请输入您想要查询的英文星期的数字编号,如 1 对应星期 1\n\r");
02  string input = Console.ReadLine();
03  switch (input)
04  {
05     case "1":
06        Console.WriteLine("您要查询的英文星期为: Monday");
07        break;
08     case "2":
09        Console.WriteLine("您要查询的英文星期为: Tuesday");
10        break;
11     case "3":
12        Console.WriteLine("您要查询的英文星期为: Wednesday");
```

```
13          break;
14      case "4":
15          Console.WriteLine("您要查询的英文星期为：Thursday");
16          break;
17      case "5":
18          Console.WriteLine("您要查询的英文星期为：Friday");
19          break;
20      case "6":
21          Console.WriteLine("您要查询的英文星期为：Saturday ");
22          break;
23      case "7":
24          Console.WriteLine("您要查询的英文星期为：Sunday");
25          break;
26      default:
27          Console.WriteLine("亲，你的输入有误哦");
28          break;
29  }
```

为了进行比较，MOL 再用 if…else 方法来实现，代码如下：

```
01  if (input == "1")
02  {
03      Console.WriteLine("您要查询的英文星期为：Monday");
04  }
05  else
06  {
07      if (input == "2")
08      {
09          Console.WriteLine("您要查询的英文星期为：Tuesday");
10      }
11      else
12      {
13          if (input == "3")
14          {
15              Console.WriteLine("您要查询的英文星期为：Wednesday");
16          }
17          else
18          {
19              if (input == "4")
20              {
21                  Console.WriteLine("您要查询的英文星期为：Thursday");
22              }
23              else
24              {
25                  if (input == "5")
26                  {
27                      Console.WriteLine("您要查询的英文星期为：Friday");
28                  }
29                  else
30                  {
31                      if (input == "6")
32                      {
33                          Console.WriteLine("您要查询的英文星期为:Saturday ");
34                      }
35                      else
36                      {
37                          if (input == "7")
38                          {
39                              Console.WriteLine("您要查询的英文星期为：Sunday");
```

```
                        }
                        else
                        {
                            Console.WriteLine("亲，你的输入有误哦");
                        }
                    }
                }
            }
        }
    }
```

各位摩丝可以比较一下，同样一个功能，用 if 来实现不管是从逻辑上还是从代码数量上来看，都是很凌乱的。所以当条件判断超过 3 种的时候，推荐大家使用 switch 语句来实现。

说明：代码位置：光盘文件\源代码\第 3 章\代码 3-3。

上面的代码运行如图 3-15 所示。

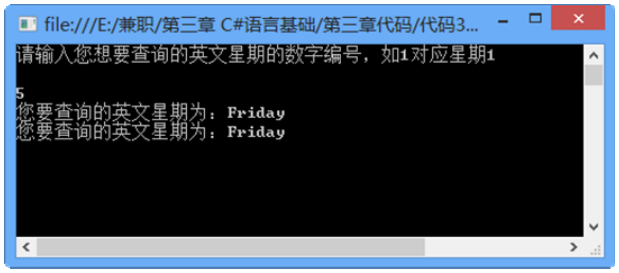

图 3-15　switch 代码和 if 代码比较

3.6.3　迭代语句 do…while

有时我们需要让一个语句块重复执行，直到某个条件符合。这时我们就用到了 do…while 迭代语句。

比如 MOL 出一道高难度的数学题：对 3X+5 进行求导，MOL 要根据用户输入的答案来判断用户答案是否正确，如果正确，则程序结束，如果不正确，就让用户重新输入答案，直到正确为止。代码如下：

```
bool right = false;
do
{
    Console.WriteLine("请输入对 3X+5 求导的结果\n\r");
    string tmp = Console.ReadLine();
    right = tmp == "3" ? true : false;
    //if (tmp == "3")        //注释掉的代码等同于上一句
    //{
    //    right = true;
    //}
    //else
    //{
    //    right = false;
```

```
14      //}
15  } while (right == false);
16  Console.ReadLine();
```

程序运行结果如图 3-16 所示。

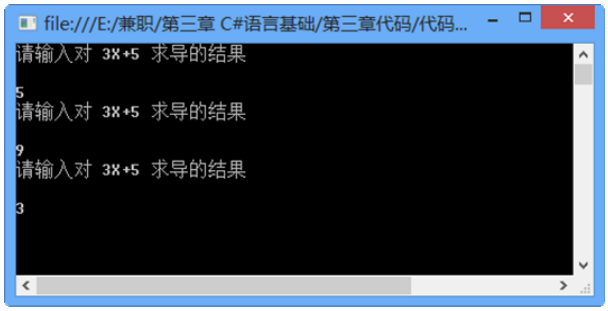

图 3-16　do…while 程序运行结果

对上面的代码解释一下。我们都知道，对 3X+5 进行求导的结果为 3，那么只有当用户输入为 3 的时候，变量 right 的值变为 true，否则 right 的值一直为 false。我们在循环代码最后写的是 while(right==false)，它表示如果 right 的值为 false，那么迭代语句会一直执行下去，直到 right 的值不为 false。也就是说，只有当用户输入为 3 的时候，right 的值才会为 true，这样程序才会结束。当用户输入不为 3 的时候，right 的值总为 false，那么程序就会一直执行，直到用户输入正确的结果。

总结一下，do…while 语句适用的情况是，程序需要等待输入的表达式到一定条件才会结束，也就是说，它适用于程序不知道用户什么时候才能给出正确输入的情况。这个迭代语句会在游戏编程中大量出现，如直到英雄击杀小兵，英雄的钱才会增加；直到对方的基地被推掉，游戏才会结束等。

3.6.4　迭代语句 for

如果需求中需要循环的次数是确定的，而且可以方便地通过下标来取到循环体中的每个元素，我们就使用 for 循环。如我们从小就背诵的 9×9 乘法表，如高斯同学小时候做的数学题：1+2+3+……+100；

for 语句的格式是：

```
for(标识初变量始值;标识变量判断条件;标识变量值变化)
{
    需要循环执行的语句块;
}
```

和 do…while 一样，我们需要让程序在希望的条件下退出循环并结束。所以我们需要一个标识变量来控制循环的进行，如果没有这个标识变量，那么程序将踏上死循环的不归路，然后你会发现电脑进入了假死状态。所以各位摩丝一定要注意，不管你使用什么样的循环语句，一定要避免死循环。

高斯同学在很小的时候就做出了一道数学题：1+2+3+……+100。所以他发明了一条高斯公式来折磨莘莘学子，自从有了计算机，我们不用记那么复杂的高斯公式，也一样可以

算出结果，代码如下（代码位置：“代码 3-5”）：

```
int re=0;                    //定义一个存放结果的变量
for (int flag = 1; flag <= 100; flag++)
                             //flag 从 1 开始计数，到 100 终止循环
{
    re += flag;              //把计数的变量加到结果变量中，从而实现了 1+2+3+……+100
}
Console.WriteLine("高斯计算出的结果为："+re);
```

代码解释：首先定义一个变量 re，用来存放结果的累加；然后进入 for 循环，在这个 for 循环中，定义了一个 flag 变量，这个变量表示循环的序数即第几次循环，我们只需要把这个序数加入到结果变量中就可以了，这样循环 100 次，就实现了从 1 加到了 100。运算结果如图 3-17 所示。

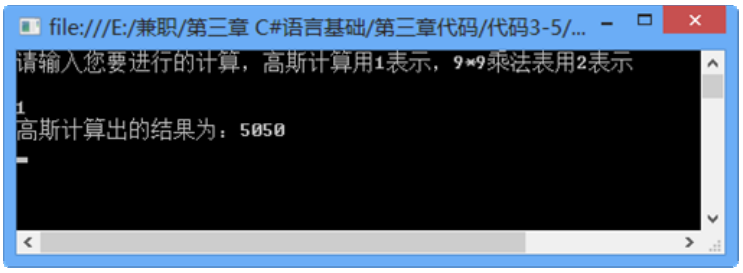

图 3-17　高斯同学计算的结果

通过这个例子，我们对 for 语句就有一个基本的认识，接下来，MOL 让大家的认识再加强一些，我们做一个 9×9 乘法表。先来分析一下 9×9 乘法表的需求。

我们可以把 9×9 乘法表看成是一个 9 行 9 列的表格，每一个单元格对应当前行数×当前列数的值。这样，我们需要两个标识变量，一个变量用来描述行数，一个变量用来描述列数。我们先对行进行循环，总共有 9 行，所以我们行变量的范围就是 1～9，用语句来描述就是 for(int i=1;i<=9;i++)；再对列进行循环，总共有 9 列，所以列变量的范围是 1～9，用语句来描述就是 for(int j=1;j<9;j++)；对于每一行，我们都要输入 9 列，所以列循环要放在行循环的语句块里，这样一个程序就形成了，代码如下：

```
for (int i = 1; i <= 9; i++)                //从 1 到 9 循环
{
    for (int j = 1; j <= i; j++)            //从 1 到 i 循环
    {
        Console.Write(i+"*"+j+"="+i*j+"\t");
    }
Console.WriteLine("\n\r");
}
```

上面代码中用到了制表符“\t”，它的作用是为了美观，输出一个制表间隔，相当于在键盘上敲一下 Tab 键。代码运行后如图 3-18 所示。

这个乘法表做得有点四不像吧，我们所熟悉的乘法表是一个梯形的形状，也就是说乘数是要大于等于被乘数的。那么，我们就需要修改一下程序了，让程序在每一行循环的时候，不要输出 9 列，而是当乘数大于等于被乘数的时候才输出。代码中，表示被乘数的是行变量 i，表示乘数的是列变量 j，所以列循环的时候，循环范围就不是 1～9 之间了，而

是 1～i 之间。改进之后的代码为：

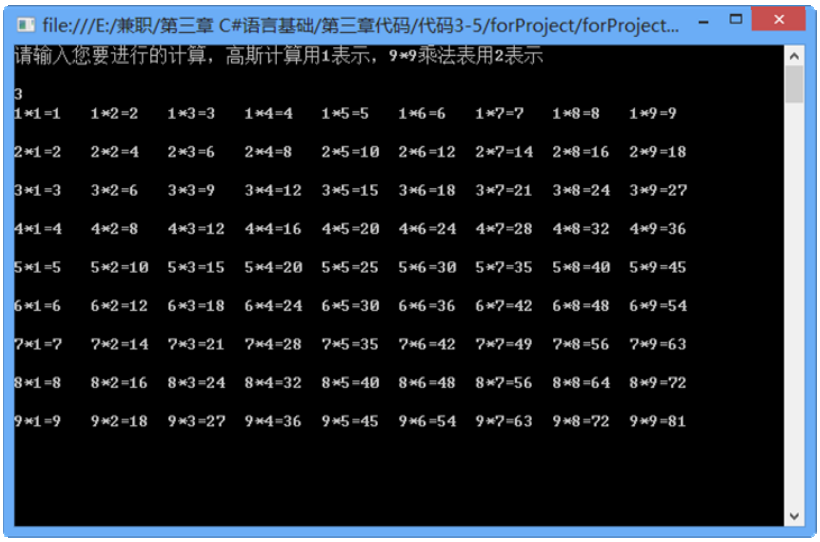

图 3-18　9×9 乘法表代码运行结果

```
for (int i = 1; i <= 9; i++)
{
    for (int j = 1; j <= i; j++)
    {
        Console.Write(i+"*"+j+"="+i*j+"\t");
    }
    Console.WriteLine("\n\r");
}
```

程序改进后输入的结果如图 3-19 所示。

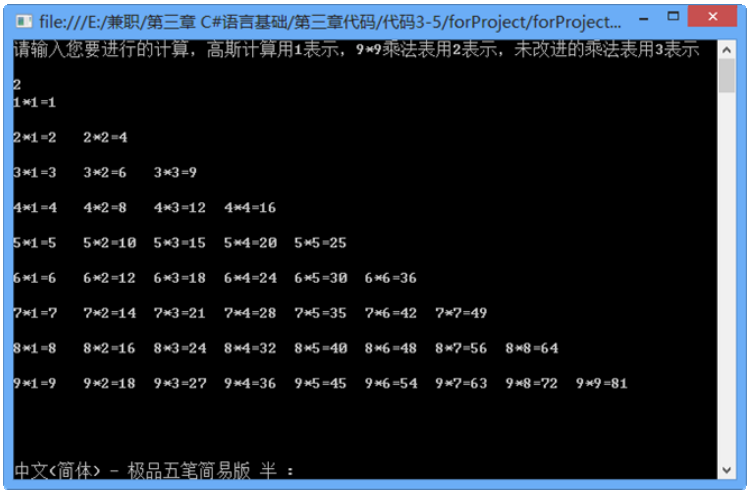

图 3-19　改进后的 9×9 乘法表

在“代码 3-5”的源文件中，MOL 把上面的代码全部集成到一个程序中，用 switch…case 来实现分支输出。希望大家能参照 MOL 写的代码自己再写一遍加深印象。

3.6.5　迭代语句 foreach

foreach 语句，就是对数组或对象集合中的每一个元素都进行一次语句块的处理，这个对象可以是空的，即一个元素都没有，也可以有成千上万个，但必须是有限的。foreach 和 for 的区别在于，for 是根据集合的下标来进行循环的，而 foreach 中没有下标的概念。说通俗一点，for 相当于哑巴吃饺子，心中有数，哑巴每吃一个饺子都会数一下，所以他知道自己吃的是第几个饺子；而 foreach 相当于狗熊掰棒子，它不知道自己掰了多少个，只知道把整片玉米地掰完为止。foreach 的语句格式为：

```
foreach(数据格式 变量 in 集合)
{
    对第一个变量进行操作;
}
```

这里的“数据格式”一定要和集合中元素的数据格式一样。

【示例 3-4】我们做一个程序，这个程序的需求说明是输出一个整数数组中的每一个元素。这个数组可以从任何途径获得，可以是从网上下载的，也有可能是服务系统传过来的等，为了方便，我们自己定义一个整数数组。代码如下（源码位置：光盘文件\源代码\第 3 章\代码 3-4）：

```
01  int[] arr = { 1,2,3,4,5,6,7,8,9};
02  foreach (int i in arr)          //针对数组中的每个 int 类型的元素
03  {
04      Console.WriteLine(i);
05  }
06  Console.ReadLine();
```

代码输出结果如图 3-20 所示。

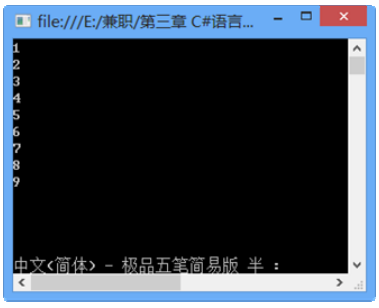

图 3-20　foreach 输出结果

有些摩丝肯定就会说了，这个功能用 for 也一样能实现，为啥还要多个 foreach？这不是画蛇添足吗？MOL 只能和你说，存在即合理。

for 循环多用来读取数据。如果是对象或集合、泛型集合用 for 循环读取数据，但只能通过下标来读取，所以很不方便。而且子典型的集合用 for 循环也是不可能读取的。同样 foreach 也无法通过下标来读取对象。所以两种循环是互补的。

3.6.6　迭代语句 while

有些摩丝看到这节的标题肯定就会说了：你刚上面不是刚讲过 while 吗，怎么又讲？稍安勿躁，此 while 非彼 while 也，怎么不一样法？听 MOL 细细道来。

while 语句就是执行一个语句块，直到指定的表达式计算为 false。那它和 do…while 有什么区别呢？我们再来回顾一下 do…while 的格式，它的格式如下。

```
do
```

```
{
    语句块;
}while(条件);
而我们这一节要讲的 while 的格式为
while(条件)
{
    语句块;
}
```

这是它们格式的区别。从它们格式的区别上不难看出，do…while 是先执行语句块，然后再判断，而 while 是先判断，如果条件满足才执行语句块。也就是说，do…while 最少执行一次，而 while 可能一次也不执行。除此之外，其他的都一样。这里 MOL 就不再写示例代码了。

3.6.7　跳转语句 break

break 用于终止最近的封闭循环或它所在的 switch 语句。控制传递给终止语句后面的语句。我们在前面讲 switch 的时候就用到了 break 来跳出 switch，它的第二个作用是终止最近的封闭循环。比如我们做一个 100 次的循环语句，当循环到第 10 次的时候，发现已经没有必要再循环下去了，这样，就可以用 break 来跳出循环，去执行循环体后面的语句。下面我们用例子来说明。

【示例 3-5】本例的需求是：有一个长度为 100 的数组，这个数组里存放的内容并不清楚，需要检索一下在这个数组里有没有一个 57 的整数，如果有，提示用户。

如果不用 break 跳转语句，我们完全可以实现，代码如下（源码位置：光盘文件\源代码\第 3 章\代码 3-5）：

```
01  for (int i = 0; i < 100; i++)           //循环 100 次
02  {
03      if (arr[i] == 57) flag = true;      //如果当前元素是 57 置标记为 true
04  }
05  if (flag)                               //如果标记为 true
06  {
07      Console.WriteLine("57 在本数组中\n\r");       //返回结果
08  }
09  else
10  {
11      Console.WriteLine("本数组中没有 57\n\r");     //返回结果
12  }
```

本段代码通过 for 循环，对数组中的每一个元素进行扫描，如果有某个元素是 57 的话，就把标识变量 flag 置为 true。程序输出如图 3-21 所示。

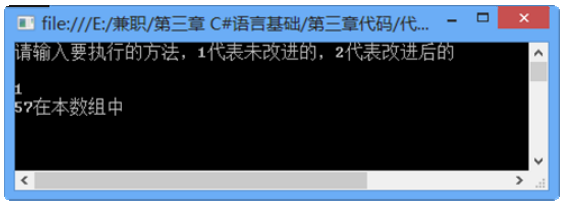

图 3-21　对数组进行扫描的结果

接下来，我们要对代码进行改进，改进的目的是去除不必要的循环，也就是说，当在数组中找到 57 这个数以后，就跳出 for 循环，从而节省 CPU 时间。其实改动很简单，只需要在循环体中的 if 判断语句块中加上 break 就可以了。

改进后的代码如下：

```
for (int i = 0; i < 100; i++)                    //循环 100 次
{
    if (arr[i] == 57)                            //如果当前元素是 57
    {
        flag = true;                             //置标记为 true
        break;                                   //跳出循环
    }
}
if (flag)                                        //如果数组中存在 57 这个元素
{
    Console.WriteLine("57 在本数组中\n\r");       //返回结果
}
else
{
    Console.WriteLine("本数组中没有 57\n\r");     //返回结果
}
```

在本例中，57 这个元素在数组的第 58 位，也就是说，比改进前要少执行 42 次。

如果我们把一次 for 循环的时间看成是一个时间片 t，那么没有改进的 for 循环要执行 100 次，就是 100t 的时间，改进后的时间为 58t 的时间。推而广之，对于一个 *n* 次的 for 循环来说，如果不使用 break 的话需要 nt 的时间，使用了 break 以后，需要(n+1)t/2 的时间（这个计算是概率统计的内容，如果大家没有学过可以不用深究），这样看来，break 还是功不可没的。当这个数组很长的时候，大家就可以感觉到 break 的优势了。有兴趣的摩丝可以做一个很长的数组来测试，这里 MOL 就不给大家演示了。

说明：在源码中，MOL 把 for 循环放在了 switch 中，也就是说大家将看到很多的 break，所以需要大家仔细一点，看清楚每一个 break 跳出的是哪个循环。

3.6.8　跳转语句 continue

大家一定玩过这样一个游戏：好几个人围成一圈，从某个人开始计数，当计到 7 或是 7 的倍数或数字中带 7，就要喊“过”，然后下一个人接着计数。在我们程序中也会遇到这样的情况，MOL 习惯把这种情况叫“淘米”，就好像给你一碗米，你要把里面的石子挑出来扔掉。对应到循环里就是对一个集合里的每一个元素进行扫描，遇到符合条件的元素就跳过，执行下一次循环。这就需要 continue 来配合了。

continue 语句就是将控制权传递给它所在的封闭迭代语句的下一次迭代。它的格式是：

```
迭代语句
{
    if(条件)
{
    continue;
}
```

```
语句块;
}
```

我们用代码来进行说明，下面的代码要实现的功能是，把 1～100 中不能被 7 整除的数字取出来（源码位置：光盘文件\源代码第 3 章\代码 3-8）。

```
string re = "";                    //这个字符串变量是要存放不能被 7 整除的数
for (int i = 1; i <= 100; i++) //从 1～100 进行扫描
{
    if (i % 7 == 0) continue;    //如果能被 7 整除则跳过，进行下一个数字的扫描
    re += i+"\n\r";                //把数字附加到 re 变量上
}
Console.WriteLine(re);
Console.ReadLine();
```

在上面的代码中，我们在 for 循环体中进行了判断 if(i%7==0)，就是判断当前扫描的数字是否能被 7 整除，如果能，则 break 到下一次循环，如果不能就附加到 re 变量上，最后把 re 变量输出，程序运行的结果如图 3-22 所示。

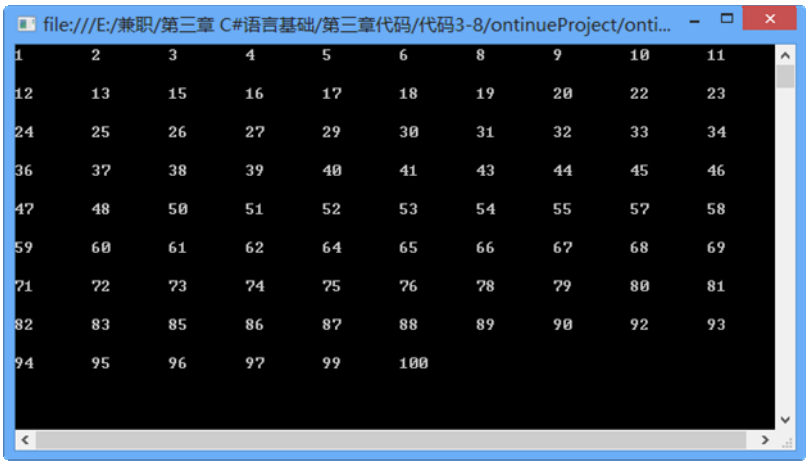

图 3-22　continue 程序输出

3.6.9　跳转语句 goto

goto 就是将程序的控制直接传递给标识语句。也就是说，不管程序现在是什么状况，只要碰到 goto，就立马跳走了。这样做的好处是程序员可以很随意地控制程序的运行流程，不好的地方就是 goto 严重地破坏了程序的逻辑，会引发不可预知的异常。所以一般不建议大家使用。下面 MOL 用一个例子说明 goto 的用法。

这个程序是老板发工资的程序，员工的基本工资是 3000，效益工资每个人都不同，但是老板一般都比较小气，会有一个上限工资，就是员工的工资不能超过一万。当有员工的基本工资超过一万的话，只给他发一万块。程序如下（源码位于光盘文件\源代码\第 3 章\代码 3-9）：

```
01  Console.WriteLine("请输入员工的效益工资\n\r");
02  int xiaoyi = Convert.ToInt32( Console.ReadLine());
```

```
03  if (xiaoyi < 7000) goto Lable1; goto Lable2;
04  Lable1: Console.WriteLine("员工应得的工资是："+(3000+xiaoyi)+"元\n\r");
05  Console.ReadLine();
06  return;
07  Lable2: Console.WriteLine("员工应得的工资是：1W 元\n\r");
08  Console.ReadLine();
```

代码运行如图 3-23 所示。

程序里的 Lable1 和 Lable2 就是标识，这个标识是我们自己定义的，你可以写任何字符，只要不和C#关键字冲突就可以了。标识就相当于路标，出现 goto 的时候，程序就知道往哪里跳转了。再次声明：goto 有风险，使用需谨慎。

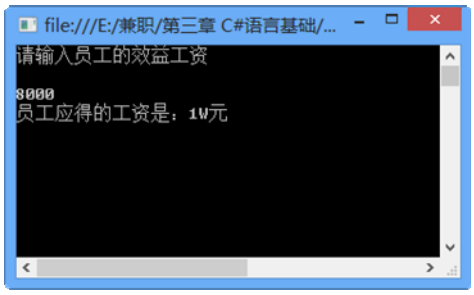

图 3-23　goto 运行结果

3.6.10　跳转语句 return

return 是终止它所在的方法的执行，并将控制交还给调用本方法的方法。此外，return 更重要的作用在于它还可以返回一个可选值。当然，如果方法为 void 类型，则可以省略 return 语句。

它的格式为：

```
数据类型 A 方法名 A()
{
    方法 A 语句块;
    return 数据类型 A 的值;
}
```

如，我们有一个方法是这样的：

```
string Name()
{
    string firstName="Mol";                  //定义字符串变量
    string MiddleName="White";               //定义字符串变量
    string re=MiddleName+" "+firstName;      //字符串连接
    return re;
    int a=1+1;
}
```

在本方法中，方法的返回类型是 string，在方法的语句块中，return 语句的作用是返回 White Mol 的一个字符串。int a=1+1;这句话在 return 的后面，所以它永远都不会被执行了。

我们还有一个方法如下：

```
void WriteName()
{
    string molsName=Name();     //方法调用的结果赋值给变量 molsName
    Console.WriteLine(molsName);
}
```

WriteName()方法体中调用 Name()方法，把 Name()方法的返回值赋值给变量 molsName。所以 string molsName=Name();这句话执行完成以后，molsName 变量的值就为“White Mol”。

Name()方法的最后有一句话是 int a=1+1;当 WriteName()方法调用 Name()方法时，Name()方法开始执行，到 return re;的时候，return 就把控制权交还给了 WriteName()方法，所以 Name()方法中的 int a=1+1 这句话永远都不会被执行。VS 也会给出提示：检测到无法执行的代码。

3.6.11　异常处理语句 try…catch…finally

我们编程的时候，不可避免地会遇到异常现象的出现，小到 1/0 这种除数为 0 的异常，大到读取文件时溢出的异常。如果不对这些异常进行处理，那么整个程序就会瘫痪。这个时候，我们就需要对异常进行处理。异常处理的格式为：

```
try
{
    语句块;
}
catch([异常类型 异常变量])
{
    处理异常的语句块;
}
[
finally
{
    不管任何情况都会执行的语句块;
}
]
```

说明：用中括号括起来的内容是可选内容。

下面，我们用例子来进行说明。

【示例 3-6】这个例子是用来计算一个班级中的平均分数的。其中，班级总分和班级人数需要用户手动输入。代码如下（源码位置：光盘文件\源代码\第 3 章\代码 3-6）：

```
01  try
02  {
03  Console.WriteLine("请输入班级总分");
04  int count = Convert.ToInt32(Console.ReadLine()); //得到用户输入的班级总分
05  Console.WriteLine("请输入班级人数");
06  int person = Convert.ToInt32(Console.ReadLine());      //得到班级人数
07  int ave = count / person;                              //计算平均分
08  Console.WriteLine("平均分是："+ave);                    //输出结果
09  return;
10  }
11  catch(Exception ee)                                    //捕获异常
12  {
13     Console.WriteLine(ee.Message);                      //输出异常信息
14  }
15  finally
16  {
17     Console.WriteLine("这是不管任何情况都能执行到的代码");
18     Console.ReadLine();
19  }
```

当我们输入正确数字的时候，程序正常执行，执行结果如图 3-24 所示。

当输入不正确的数字，或者直接输入字符串时，程序运行如图 3-25 所示。

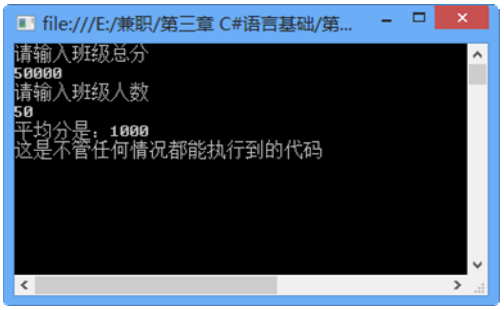

图 3-24　程序正常运行

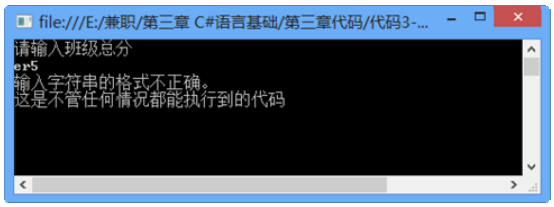

图 3-25　程序处理异常

这里本应该输入班级的总分，但是我们输入了字母，所以执行 int count = Convert.ToInt32(Console.ReadLine());的时候就会抛出异常，因为“er5”显然是不能够被转换成整数类型的。抛出异常后，程序就进入了 catch 语句块，把异常信息输出到屏幕上。最后进入 finally 语句块。

3.6.12　抛出异常 throw

这里所接收的异常是微软定义好的异常，当然我们也可以自己抛出异常，如，当用户输入的数字小于 0 时，抛出异常，异常信息为：输入的数字不能小于 0。这就用到了 throw，throw 经常被用来抛出自定义的异常。我们将 3.6.11 节的程序改进后，代码如下：

```
01  try
02  {
03  Console.WriteLine("请输入班级总分");
04  int count = Convert.ToInt32(Console.ReadLine());
05      if(count<0) throw(new Exception("输入的数字不能小于 0"));
06  Console.WriteLine("请输入班级人数");
07  int person = Convert.ToInt32(Console.ReadLine());
08  if (person < 0) throw (new Exception("输入的数字不能小于 0"));
09  int ave = count / person;
10  Console.WriteLine("平均分是: "+ave);
11  return;
12  }
13  catch(Exception ee)
14  {
15      Console.WriteLine(ee.Message);
16  }
17  finally
18  {
19      Console.WriteLine("这是不管任何情况都能执行到的代码");
20      Console.ReadLine();
21  }
```

程序运行的结果如图 3-26 所示。

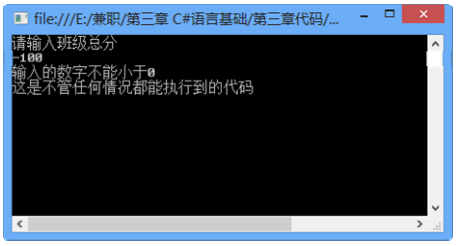

图 3-26　自定义异常运行结果

到这里，C#的流程控制语句基本介绍完毕，希望大家能把书上的例子自己动手写一遍。

3.7　小　　结

本章主要讲解了 C#中的数据类型的分类、定义和使用；还讲了常量和变量的用法。重点一是表达式与运算符，大家必须对常用的表达式和运算符做到烂熟于心才能进行下面的学习；重点二是流程控制语句。流程控制语句就像是一条主线把 C#中每一个分散的元素都串起来，可见流程控制语句是 C#语言的灵魂所在。希望大家能熟记常见的流程控制语句并能熟练运用。

3.8　习　　题

1．在屏幕上输出“我的名字是 MOL”。
2．C#的数据类型可以分为哪两种类型？简述之。
3．C#的两种数据类型有什么区别？
4．什么是装箱和拆箱？举例说明。
5．什么是变量？变量有什么特性？
6．什么是常量？常量有什么特性？
7．常量和变量分别在什么情况下使用？
8．列举常用的运算符，并说明它们的作用。
9．选择语句有几种？分别是什么？
10．迭代语句有几种？分别是什么？
11．跳转语句有种种？分别是什么？
12．在使用跳转语句的时候，需要注意什么？
13．如何避免程序出现不可预知的错误？
14．用程序实现四则运算。