第3章　C#语言基础

本章学习目标

- 掌握C#的基本数据类型。
- 掌握C#各种运算符、表达式的用法。
- 理解C#控制台程序的基本结构。
- 能够使用IF、WHILE、FOR等语句编写程序。

ASP.NET支持所有的.NET语言，目前.NET语言主要有C#（读作C Sharp）、VB.NET、JScript等。对熟悉C++的用户来说，C#更易学习；熟悉VB的用户，VB.NET也是不错的选择。在本书中，程序都是由C#语言实现，本章主要介绍C#语言。

C#是微软于2000年提出的一种源于C++、类似于Java的面向对象编程语言，适合于分布式环境中的组件开发。C#是专门为.NET设计的，也是.NET编程的首选语言。它吸收了C++、Visual Basic、Delphi、Java等语言的优点，体现了当今最新的程序设计技术的功能和精华。

3.1　C#语言的特点

Microsoft.NET（以下简称.NET）框架是微软提出的新一代Web软件开发模型，C#语言是.NET框架中新一代的开发工具。其主要特点如下。

1. C#语言是一种现代、面向对象的语言

C#语言简化了C++语言在类、命名空间、方法重载和异常处理等方面的操作，它摒弃了C++的复杂性，更易使用，更少出错。它使用组件编程，和VB一样容易使用。C#语法和C++、Java语法非常相似，如果读者用过C++和Java，学习C#语言应是比较轻松的。用C#语言编写的源程序，必须用C#语言编译器将C#源程序编译为微软中间语言（Microsoft Intermediate Language，MSIL）代码，形成扩展名为.exe或.dll的文件。中间语言代码不是CPU可执行的机器码，在程序运行时，必须由通用语言运行环境（Common Language Runtime，CLR）中的即时编译器（JUST IN Time，JIT）将中间语言代码翻译为CPU可执行的机器码，由CPU执行。CLR为C#语言中间语言代码运行提供了一种运行时的环境，C#语言的CLR和Java语言的虚拟机类似。虽然这种执行方法使运行速度变慢，但也带来其他一些无与伦比的优势。

1）通用语言规范

.NET系统包括如下语言：C#、C++、VB、J#，它们都遵守通用语言规范。任何遵守通用语言规范的语言源程序，都可编译为相同的中间语言代码，由CLR负责执行。只要为其他操作系统编制相应的CLR，中间语言代码也可在其他系统中运行。

2）自动内存管理

CLR内建垃圾收集器，当变量实例的生命周期结束时，垃圾收集器负责收回不被使用

的实例占用的内存空间。不必像 C 和 C++ 语言那样，用语句在堆中建立的实例，必须用语句释放实例占用的内存空间。也就是说，CLR 具有自动内存管理功能。

3）交叉语言处理

由于任何遵守通用语言规范的语言源程序，都可编译为相同的中间语言代码，不同语言设计的组件，可以互相通用，可以从其他语言定义的类派生出本语言的新类。由于中间语言代码由 CLR 负责执行，因此异常处理方法是一致的，这在调试一种语言调用另一种语言的子程序时，显得特别方便。

2. 较强的安全性

C＃语言不支持指针，一切对内存的访问都必须通过对象的引用变量来实现，只允许访问内存中允许访问的部分，这就防止了病毒程序使用非法指针访问私有成员，也避免指针的误操作产生的错误。CLR 执行中间语言代码前，要对中间语言代码的安全性、完整性进行验证，防止病毒对中间语言代码的修改。

系统中的组件或动态链接库可能要升级，由于这些组件或动态链接库都要在注册表中注册，由此可能带来一系列问题，例如，安装新程序时自动安装新组件替换旧组件，有可能使某些必须使用旧组件才可以运行的程序，使用新组件后不能运行。在.NET 中这些组件或动态链接库不必在注册表中注册，每个程序都可以使用自带的组件或动态链接库，只要把这些组件或动态链接库放到运行程序所在文件夹的子文件夹 bin 中，运行程序就自动使用在 bin 文件夹中的组件或动态链接库。由于不需要在注册表中注册，软件的安装也变得容易了，一般将运行程序及库文件复制到指定文件夹中即可。

3. 完全面向对象

C＃语言是完全面向对象的语言，它不像 C++ 语言，既支持面向过程程序设计，又支持面向对象程序设计。在 C＃中不存在全局函数、全局变量，所有的函数、变量和常量都必须定义在类中，避免了命名冲突。C＃语言不支持多重继承。

总之，C＃语言的特点可以归纳如下。

（1）语言简洁、易于理解，支持快速开发。

（2）可移植性好。

（3）面向对象，在 C＃中一切都是对象，所有对象都是由 Object 派生而来。

（4）类型安全。

（5）C＃的设计借鉴了多种语言，但最主要借鉴了 Java 和 C++ 。

3.2 程序结构

3.2.1 命名空间

在 C＃中，把系统中包含的内容按功能分成多个部分，每部分放在一个命名空间中，以避免不同部分内同名标识符的“命名冲突”。命名空间体现了标识符的逻辑分区管理策略。

1. C＃程序的基本框架

所有的 C＃程序都包括一个框架，其基本结构为：

```
using 命名空间;
```

```
[访问修饰符] class 类名
{
    …
    static void Main()
        {
            方法体
        }
}
```

C#中必须且只能包含一个 Main 方法,置于某个类中。Main 方法是程序的入口点和出口点。系统从 Main 方法开始执行,到 Main 方法结束,也就意味着程序结束。

2. 命名空间的声明

命名空间的声明格式为:

```
namespace 命名空间名
{
    类型声明
}
```

使用时用"using 命名空间名;"导入。

注意:编译源程序时必须得到与之相匹配的动态链接库的支持。因此,一般在程序开头添加"命名空间"的引用,否则编译环境无法识别。

命名空间可以包含其他的命名空间,这种划分方法的优点类似于文件夹。但与文件夹不同的是,命名空间只是一种逻辑上的划分,而不是物理上的存储分类。

3. 命名空间的相关规则

(1) 用关键字 namespace 声明一个命名空间。声明时不允许使用任何访问修饰符,命名空间隐式地使用 public 修饰符。

(2) 全局命名空间应是源文件 using 语句后的第一条语句。在一个命名空间声明内部,还可以声明该命名空间的子命名空间。

(3) 在同一命名空间中,不允许出现同名命名空间成员或同名的类。

【例 3.1】 声明命名空间 N1,N2。其中 N1 为全局命名空间,N2 为 N1 的子命名空间。代码如下。

```
using System;
namespace N1.N2                          //同时定义命名空间 N1.N2
{  class A                               //类 A、B 在命名空间 N1.N2 中
       {  void f1(){ };  }
   class B
       {  void f2(){ };  }
}
```

也可以采用嵌套的方式声明以上命名空间:

```
using System;
namespace N1                             //声明 N1 为全局命名空间
```

```
{  namespace N2                          //(嵌套)声明 N1 的子命名空间 N2
   {  class A                            //在 N2 空间内定义的类不应重名
         {  void f1(){ };  }
      class B
         {  void f2(){ };  }
   }
}
```

4. 命名空间的使用

在程序中,需引用其他命名空间的类或函数,可以使用语句 using,使用形式如下。

```
using N1.N2;                             //告诉应用程序到哪里找到类 A
class WelcomeApp
{
    A a=new A();
    f1();
}
```

如果不使用 using 语句,应使用如下形式。

```
class WelcomeApp
{  N1.N2.A a=new N1.N2.A();              //表示类 A 在命名空间 N1.N2 中
   f1();
}
```

在一个命名空间中,可以声明一个或多个类型,包括类、接口、结构、枚举、委托等。即使未显式声明命名空间,也会创建默认命名空间,未命名的命名空间存在于每一个文件中。

5. 系统定义的命名空间

命名空间分为两类:用户自定义的命名空间和系统定义的命名空间。用户自定义的命名空间是在程序中自己定义的命名空间,而系统定义的命名空间是系统自动提供的,表 3.1 列出了系统定义的常用的命名空间。

表 3.1 系统定义常用的命名空间

命 名 空 间	说 明
System	定义通常使用的数据类型和数据转换的基本.NET 类
System.Collection	定义列表、队列、位数组和字符串表
System.Data	定义 ADO.NET 数据库结构
System.Drawing	提供对基本图形功能的访问
System.IO	允许读写数据列和文件
System.Net	提供对 Windows 网络功能的访问
System.Net.Sockets	提供对 Windows 套接字的访问
System.Runtime.Remoting	提供对 Windows 分布式计算平台的访问
System.Security	提供对 CLR 安全许可系统的访问

续表

命 名 空 间	说　　明
System. Text	ASCII、Unicode、UTF-7 和 UTF-8 字符编码处理
System. Threading	多线程编程
System. Timers	在指定的时间间隔引发一个事件
System. Web	浏览器和 Web 服务器功能
System. Web. Mail	发送邮件信息
System. Windows. Forms	创建使用标准 Windows 图形接口和基于 Windows 的应用程序
System. XML	提供对处理 XML 文档的支持

3.2.2 类

与 C++ 一样,C# 是一种面向对象的编程语言,它通过类、结构和接口来支持对象的封装、继承和多态等特征,还利用委托和事件来支持对象的消息响应。C# 中的类是一种至关重要的结构,类是相似的对象的一个组。类的全部成员都享有类的属性和行为。对象是类的实体。所以在程序中,使用类和对象的好处是可以模拟现实世界中的很多对象,这对于开发大型软件是相当有利的。

1. 类的定义

C# 的每一个程序包括至少一个自定义类,类的定义格式如下。

```
[访问修饰符] Class <类名>
{
    <实例变量>
    <方法>
}
```

和其他大多数面向对象语言一样,类的定义包括关键字 Class,跟在其后的类名(标识符)以及花括号对,它们共同组成了类体,类成员就位于其中。

类成员一般分为以下三类。

(1) 数据成员;

(2) 函数成员;

(3) 嵌套类型。

数据成员包括成员变量、常量和事件。

成员变量用于表示数据。这个成员变量可以是实例变量(对象),也可以是静态变量(非对象)。成员变量可以声明为只读变量,这与常量数据成员紧密相关。但二者存在区别,只读变量在创建时给它赋值,并在此对象的生存期内存在,而常量在编译时被指定一个值,并在程序编译的整个生存期存在。

事件是可以被控件识别的操作,如单击“确定”按钮,选择某个单选按钮等。每一种控件有自己可以识别的事件,如窗体的加载、单击等事件。事件有系统事件和用户事件。系统事件由系统激发,如时间每隔 24 小时,银行储户的存款日期增加一天。用户事件由用户激发,如用户单击按钮,在文本框中显示特定的文本,事件驱动控件执行某项功能。

在.NET 框架中，事件是将事件发送者（触发事件的对象）与事件接收者（处理事件的方法）相关联的一种代理类，即事件机制是通过代理类来实现的。当一个事件被触发时，由该事件的代理来通知（调用）处理该事件的相应方法。

C＃中事件机制的工作过程如下。

（1）将实际应用中需通过事件机制解决的问题对象注册到相应的事件处理程序上，表示今后当该对象的状态发生变化时，该对象有权使用它注册的事件处理程序。

（2）当事件发生时，触发事件的对象就会调用该对象所有已注册的事件处理程序。

结合上面的叙述，将类成员概述介绍如下。

```
[访问修饰符] Class <class_name>
{
    <class_members>;
}
```

其中：

＜class_members＞包括数据成员、函数成员和嵌套类型。

数据成员有成员变量、常量和事件等。

函数成员有方法、属性、构造函数、析构函数、索引和操作符等。

嵌套类型有类、结构和枚举等。

2. C＃中的修饰符

C＃中的修饰符主要是用来控制作用域，限制访问权限的。C＃中有三类修饰符，分别是访问修饰符、类修饰符和成员修饰符。

1）访问修饰符

类的访问修饰符如表 3.2 所示。

表 3.2 类的访问修饰符

修 饰 符	说 明
public	访问无限制
protected	只可被包含类或其派生的类型访问
internal	只能被此程序访问
protected internal	只能被此程序或其包含类所派生的类型访问
private	只能被其包含类访问，为默认的

2）类修饰符

类修饰符如表 3.3 所示。

表 3.3 类修饰符

类的修饰符	说 明
abstract	用于修饰抽象类，抽象类不允许实例化
sealed	sealed 用于修饰最终类（密封类），最终类不允许派生

3) 成员修饰符

类成员修饰符如表3.4所示。

表3.4 类成员修饰符

修饰符	说明	修饰符	说明
abstract	定义抽象函数	override	定义重载
const	定义常量	readonly	定义只读属性
event	定义事件	static	用来声明静态成员
extern	告诉编译器在外部实现	virtual	定义虚函数

使用访问修饰符 private(私有)和 public(公有)等控制类的访问级别,默认情况为"私有",若要在其他项目或类中访问被定义的类,该类应声明为"公有"类。访问修饰符 private 和 public 也可用于声明类的成员。

C#的 public、protected、private 成员修饰符,每次只能修饰一个成员,直接位于成员声明的开始处。

例如,定义一个描述个人情况的类 Person。

代码如下。

```
using System;
class Person
{  private string name="张三";
   private int age=12;
   public void Display()
   {  Console.WriteLine("姓名:{0},年龄: {1}",name,age); }
   public void SetName(string PersonName)
   {  name=PersonName; }
   public void SetAge(int PersonAge)
   {  age=PersonAge; }
}
```

Person 类仅是一个用户新定义的数据类型,由它可生成 Person 类的实例,C#中称之为对象。

方法:

```
Person OnePerson=new Person();          //生成对象
```

或者:

```
Person OnePerson;                       //先建立变量
OnePerson=new Person();                 //再生成对象
```

Person()是 Person 类的构造函数,用于生成 Person 类的对象。

变量 OnePerson 是对 Person 类的对象的引用,不是 C 中的指针,不能像指针那样进行加减运算,也不能转换为其他类型地址,它是引用型变量,只能引用(代表)Person 对象。

C#类的继承是面向对象程序设计的主要特征之一,它可以重用代码,节省程序设计的

时间。继承就是在类之间建立一种相交关系，使得新定义的派生类的实例可以继承已有的基类的特征和能力，而且可以加入新的特性或者是修改已有的特性建立起类的新层次。

3. 类成员的种类

类成员的种类有：constant-declaration（常量声明）、field-declaration（字段声明）、method-declaration（方法声明）、property-declaration（属性声明）、event-declaration（事件声明）、indexer-declaration（索引器声明）、operator-declaration（运算符声明）、constructor-declaration（构造函数声明）、finalizer-declaration（终结器声明，即析构函数的定义）、static-constructor-declaration（静态构造函数声明）、type-declaration（类型声明）。

3.2.3 结构

结构是使用 struct 关键字定义的，与类相似，都表示可以包含数据成员和函数成员的数据结构。

1. 结构与类的区别

C＃中的结构与类的区别主要在于以下几点。

（1）结构是值类型，而类是引用类型；

（2）结构是密封的（Sealed），因此不能被继承；

（3）结构不能继承类和其他的结构；

（4）结构隐式地继承了 System. ValueType 类型；

（5）结构的（无参数）默认构造函数不能被自定义的构造函数取代；

（6）结构的自定义的构造函数，必须初始化结构中全部成员的值；

（7）结构没有析构函数；

（8）不允许初始化结构的字段，但是可以初始化结构的常量成员。

总之，结构和类一样，可以声明构造函数、数据成员、方法、属性等。结构和类的最根本的区别是结构是值类型，类是引用类型。而且，结构不能从另外一个结构或者类派生，本身也不能被继承。

2. 结构的声明

结构的完整声明格式为：

```
[[属性]][结构修饰符] [partial] struct 标识符 [<类型参数列表>] [: 结构接口列表] [类型参数约束子句] {
    [结构成员声明 …]
}[;]
```

其中，结构的修饰符有：new、public、protected、internal 和 private。由于 C＃的结构不支持继承，所以没有类的 sealed 和 abstract 修饰符，也没有 static 修饰符。结构的默认修饰符为 public。

结构成员包括：

（1）数据成员：表示结构的数据项。

（2）方法成员：表示对数据项的操作。

例如，定义一个结构 Student 的格式为：

```
struct Student
{
    public int no;                    //声明结构型的数据成员
    public string name;
    public char sex;
    public int score;
};
```

结构成员声明与类的成员声明基本相同,只是没有 finalizer-declaration(终结器声明,即析构函数定义)。结构和类一样,使用 public、protected、private 等作为成员修饰符,每次只能修饰一个成员,也是直接位于成员声明的开始处。

例如:

```
//定义结构
struct Point {
    public int x,y;
    public Point(int x,int y) {
        this.x=x;
        this.y=y;
    }
}
//使用结构
Point a=new Point(10,10);
Point b=a;
a.x=100;
System.Console.WriteLine(b.x);
```

结构可以作为顶层类型,也可以作为类的成员,但是不能作为局部类型来定义。在 C# 的结构类型定义体后的分号是可选项。

3. 结构的特征

C# 的结构主要有以下特征。

(1) 结构是一种值类型,并且不需要堆分配。结构的实例化可以不使用 new 运算符。

(2) 在结构声明中,除非字段被声明为 const 或 static,否则无法初始化。结构类型永远不是抽象的,并且始终是隐式密封的,因此在结构声明中不允许使用 abstract 和 sealed 修饰符。

(3) 结构不能使用默认的构造函数,只能使用带参数的构造函数,当定义带参数的构造函数时,一定要完成结构所有字段的初始化,如果没有完成所有字段的初始化,编译时会发生错误。

(4) 由于结构不支持类与结构的继承,所以结构成员的声明可访问性不能是 protected 或 protected internal。结构中的函数成员不能是 abstract 或 virtual,因而 override 修饰符只适用于重写从 System. ValueType 继承的方法。

(5) 结构在赋值时进行复制。将结构赋值给新变量时,将复制所有数据,并且对新副本所做的任何修改不会更改原始副本的数据。

【例 3.2】 创建一个结构,主要用于描述学生信息。结构包括数据成员和方法成员的

定义及使用。

代码如下。

```
using System;
using System.Windows.Forms;
namespace TestStru
{
    public partial class TestStru : Form
    {
        struct Student                                  //声明结构
        {
            //声明结构的数据成员
            public int no;
            public string name;
            public char sex;
            public int score;
            public string Answer()                      //声明结构的方法成员
            {
                string result="该学生的信息如下：";
                result +="\n 学号：" +no;               //+no 相当于+this.no,下同
                result +="\n 姓名："+name;
                result +="\n 性别："+sex;
                result +="\n 成绩："+score;
                return result;                          //返回结果
            }
        };
        private void TestEnum_Load(object sender,EventArgs e)
        {
            Student s;                                  //使用结构
            s.no=101;
            s.name="黄海";
            s.sex='男';
            s.score=540;
            lblShow.Text=s.Answer();                    //显示该生信息
            lblShow.Text +="\n\n"+DateTime.Now;         //显示当前时间
        }
    }
}
```

3.3 C#的数据结构

3.3.1 变量和常量

1. C#中的数据类型

C#是一种强类型语言，即程序中用到的每个存储单元都必须指明其数据类型。这似乎是一个较强的约束，但却给开发很大的帮助，这是由于C#编译器能够检查程序，以便使

程序中的数据使用正确的数据类型,这使开发人员在运行程序之前就能发现错误。

C#将所有的数据类型分成两大类,即值类型和引用类型。

这两种类型的区别在于:值类型的数据长度固定,存放于栈内,值类型存储的是自身的数值。引用类型的数据长度可变,存放于堆内,存储的是对数值的引用。一个值类型变量保存对应的实际值。int 类型就是一个典型例子。如果将一个变量 x 声明为 int 并赋值为 123,则说明数据类型是 int,标识符是 x,x 的值是 123。

一个引用变量包含一个对计算机存储器中对象的引用,它本身并不包含此对象。引用包括指向对象的位置。为了简要说明引用类型,在此举一个 string 类型。

一个 string 类型的变量本身并不包含一个字符串(string),但却声明它表示对计算机存储器中某个字符串的引用。

```
string myText;
```

声明 myText 为 string 类型。

在这里就是声明了一个引用变量,允许 myText 表示对一个字符串的引用。

例如,

```
myText="Hello World!";
```

此语句将字符串“Hello World!”的地址分配给 myText。

C#的值类型如表 3.5 所示。

表 3.5 C#的值类型

值 类 型	说 明
简单类型	有符号整型:sbyte、short、int 和 long
	无符号整型:byte、ushort、uint 和 ulong
	Unicode 字符型:char
	浮点型:float 和 double
	高精度小数型:decimal
	布尔型:bool
枚举类型	enum E {…} 形式的用户定义的类型
结构类型	struct S {…} 形式的用户定义的类型
可以为 null 的类型	其他所有具有 null 值的值类型的扩展

C#的引用类型如表 3.6 所示。

表 3.6 C#的引用类型

引用类型	说 明	引用类型	说 明
类类型	其他所有类型的最终基类:object	接口类型	interface I {…} 形式的用户定义的类型
	Unicode 字符串型:string	数组类型	一维和多维数组,例如 int[] 和 int[,]
	class C {…} 形式的用户定义的类型	委托类型	delegate int D(…) 形式的用户定义的类型

值类型和引用类型的主要区别如表 3.7 所示。

表 3.7 值类型和引用类型的区别

特 性	值 类 型	引 用 类 型
变量中保存的内容	实际数据	指向实际数据的引用指针
内存空间配置	栈(Stack)	受管制的堆(Managed Heap)
内存需求	较少	较多
执行效率	较快	较慢
内存释放时间点	执行超过定义变量的作用域时	由垃圾回收机制负责回收
可以为 null	不可以	可以

1) 整型

C#有 8 种整型,可以用来表示全部的整型数字。表 3.8 列出了相关整型。

表 3.8 整型

类型	.NET 类型	所占字节数	存 储 的 值
sbyte	System. SByte	1	−128～127 之间的整数
byte	System. Byte	1	0～255 之间的整数
short	System. Int16	2	−32 768～32 767 之间的整数
ushort	System. UInt16	2	0～65 535 之间的整数
int	System. Int32	4	2 147 483 648～2 147 483 647 之间的整数
uint	System. UInt32	4	0～4 294 967 259 之间的整数
long	System. Int64	8	9 223 372 036 854 775 808～9 223 372 036 854 775 807 之间的整数
ulong	System. Unt64	8	0～18 446 744 073 709 551 615 之间的整数

每个类型都有有符号和无符号两种形式。有符号类型能够存储符号,因此能够存储正数和负数,而无符号类型只能存储正数。

例如:

```
int intNumber=123;
long LongNumber=-123;
ulong UlongNumber=-68;
```

2) 浮点型

C#有三种浮点类型来表示浮点数,表 3.9 列出了相关内容。

表 3.9 浮点型

类 型	.NET 类型	所占字节数	存 储 的 值
float	System. Single	4	1.5×10^{-45}～3.4×10^{38}
double	System. Double	8	5.0×10^{-324}～1.7×10^{308}
decimal	System. Decimal	12	1.0×10^{-28}～7.9×10^{28}

注意：当赋值给浮点型时，必须在结尾加一个 f 或 F 字符。

例如：

```
float FloatNumber=12.3f;
```

当使用小数类型时，需要在结尾加一个 m 或 M 字符。

例如：

```
Decimal DecimalNumber=1234.5643m;
```

这是C#编译器的原因，它认为具有小数点的数字都是 double 类型的，除非声明它不是，即在数字后面加一个 F 或 M 字符分别表示数字是 float 或 decimal 类型。

3) 布尔型

C#中布尔型的值有 true 和 false 两种，如表 3.10 所示。

表 3.10 浮点型

类型	.NET 类型	所占字节数	存储的值
bool	System.Boolean	1	true 或 false

例如：

```
bool xyz=true;
```

4) 字符型

C#中 char 数据类型存储的是单独的字符值，如表 3.11 所示。

表 3.11 浮点型

类型	.NET 类型	所占字节数	存储的值
char	System.Char	2	一个 Unicode 字符，存储 0～65 535 之间的整数

例如：

```
char Mychar='W';
```

也可以把 Unicode 字符赋给字符串，Unicode 字符需要用到如下的转义字符：\u××××。其中，××××是一个 4 位的十六进制数，例如：

```
Mychar='\u0033';
```

同样可以把转义字符赋给字符型(char)，转义字符是有特殊意义的字符。表 3.12 列出了C#中允许的转义字符和它们的意义。

例如，给 OneChar 赋给一个垂直制表符，则用：

```
char OneChar='\t';
```

也可以赋一个十六进制数字给字符型，例如：

```
char MyHexchar='\xof';
```

表 3.12 转义字符

转义字符	含 义	Unicode 码	转义字符	含 义	Unicode 码
\'	单引号	\u0027	\t	水平制表符	\u0009
\"	双引号	\u0022	\f	走纸换页符	\u000C
\\	反斜线	\u005C	\n	换行	\u000A
\0	空字符	\u0000	\r	回车	\u000D
\a	警铃	\u0007	\b	退格	\u0008
\v	垂直制表符	\u000B			

5）预定义引用类型

C＃中有两种预定义引用类型，如表 3.13 所示。

表 3.13 预定义引用类型

类 型	.NET 类型	注 释
object	System. Object	root 类型：每一个. NET 类型都是由此继承
string	System. String	Unicode 字符串

Object 类型是 C＃所有数据类型的基类型，具有一些通用的方法；

String 类型可以方便地处理字符串操作，该类型的值需要放在双引号中。

2. 常量

在程序运行过程中，其值保持不变的量称为常量。常量类似于数学中的常数。常量可分为符号常量和直接常量两种形式。

常量是在程序运行期间其值不发生变化的量。常量包括符号常量和数值常量。

1）符号常量

符号常量是用一个标识符表示的常量。在 C＃中，符号常量有以下两种定义方法。

（1）使用 const 关键字声明（const 常量）

const 常量在编译时设置其值并且永远不能更改。编译时，编译器会把程序中的所有 const 常量全部替换为对应常数。

声明格式：

```
const 数据类型  常量名=值表达式;
```

例如：private const double PI＝3.1415926；

（2）使用 readonly 关键字声明（同上常量）

readonly 常量在程序运行期间只能被“初始化一次”，可以在声明语句中初始化，也可以在构造函数中初始化。初始化以后，用 readonly 声明的常量值就不能再更改。

声明格式：

```
readonly 数据类型  常量名=值表达式;
```

(3) 两种定义符号常量方法的区别

readonly 常量运行时初始化,const 常量编译时初始化。const 常量只能在声明中赋值,readonly 常量既可以在声明中赋值,也可以在构造函数中赋值。

通常常量名的第一个字母为大写。

2) 直接常量

所谓直接常量,就是在程序中直接给出的数据值。在 C# 中,直接常量包括整型常量、浮点型常量、小数型常量、字符型常量、字符串常量和布尔型常量。

(1) 整型常量

整型常量分为有符号整型常量、无符号整型常量和长整型常量,有符号整型常量写法与数学中的常数相同,直接书写,无符号整型常量在书写时添加 u 或 U 标志,长整型常量在书写时添加 l 或 L 标记。例如 3、3U、3L。

(2) 浮点型常量

浮点型常量分为单精度浮点型常量和双精度浮点型常量。单精度浮点型常量在书写时添加 f 或 F 标记,双精度浮点型常量添加 d 或 D 标记。例如 7f、7d。

注意:以小数形式直接书写而未加标记时,系统将自动解释成双精度浮点型常量。例如,9.0 即为双精度浮点型常量。

(3) 小数型常量

在 C# 中,小数型常量的后面必须添加 m 或 M 标记,否则就会被解释成标准的浮点型数据。C# 中的小数和数学中的小数是有区别的。

例如:5.0M

```
decimal y=99999999999999999999.99999m;
```

(4) 字符型常量

字符型常量是一个标准的 Unicode 字符,使用两个单引号来标记。例如'5'、'd'、'青'、'#'都是标准的字符型常量。

C# 还允许使用一种特殊形式的字符型常量,即以反斜杠“\”开头,后面跟字符的字符序列,这种字符型常量被称为转义字符常量。该形式的常量可以表示控制字符或不可见字符,当然也可以表示可见字符。例如,'\n'表示换行符,而'\x41'则表示字符 A。C# 中常用的转义字符如表 3.12 所示。

(5) 字符串型常量

字符串常量表示若干个 Unicode 字符组成的字符序列,使用两个双引号来标记。例如,"5"、"abc"、"青海民族大学"都是字符串。

(6) 布尔型常量

布尔型常量只有两个:一个是 true,表示逻辑真;另一个是 false,表示逻辑假。

将字符串常量“hello”赋给字符串变量 str,使用语句:string str="hello";。

3. 变量

在程序运行的过程中,值可以改变的量称为变量。每个变量用一个变量名标识,变量命名应遵循标识符的命名规则,程序中通过变量名来引用变量的值。

变量代表了存储单元,且每个变量都有一个类型。变量的类型决定了该变量可以存储

的值类型。C＃是类型安全的语言，每个C＃编译器会保证存储在变量中的值总是恰当的类型。

在C＃中，使用变量的基本原则是，先定义，后使用。

1）变量的命名规则

C＃中给变量命名必须以字母或下划线开头，其后的字符可以是字母、下划线、数字和@。不能使用C＃中的关键字作为变量名，如 using、namespace 等，因为这些关键字对于C＃编译器而言有特定的含义。

C＃区分字母的大小写，因而在命名变量时，一定要使用正确的大小写，因为在程序中使用它们时，即使只有一个字母的大小写出错，也会引起编译错误。

2）变量的声明和初始化

要使用变量，就必须首先声明，即给变量指定名称和类型。只有声明了变量，才可以将它们用作存储单元，存取相应类型的数据。

变量的声明格式：

```
<type><name>;
```

其中，type 是变量的类型，name 是变量的名称。

以下是定义正确的变量：

```
int age;
double d;
bool isTeacher;
string sql;
char myC;
DateTime dt;
```

以下是定义不正确的变量：

```
char 2abc;
float class;
decimal Main;
int String;
string a-b;
```

在C＃的一条语句内，允许声明多个变量，但彼此之间应该用逗号隔开，而且声明的变量是同一种类型，例如：

```
int a,b,c;
```

在声明变量后就可以对变量进行相应的赋值初始化，可以使用运算符“＝”为变量分配一个值，例如：

```
double d=2.4;
string s="hello CSharp";
int productNumber=1001;
```

或者先声明变量再赋值：

```
int productNumber;
productNumber=1001;
```

在C#中,引用变量的值前都应该赋值。在没有给变量赋值之前就想引用变量的值是非常危险的,C#编译器会提示错误。

例如:

```
int productNumber;
System.Console.WriteLine(productNumber);     //引发相应的错误
```

在第二句中引用 productNumber 时,它还没有赋值,编译器就会提示相应的错误。

对于引用类型的变量同样是相同的,只是实例化对象的语法与简单的值类型赋值有一定的区别。例如:

```
TheObject myObject;
myobject=new TheObject;
```

第一行代码是为 TheObject 对象创建了一个引用,但是该引用并没有指向任何事物,所以任何对 TheObject 的操作都会失败。

第二行代码实际上是将该引用指向 TheObject 对象,利用 new 关键字创建一个新的对象,通过 myObject 对象的引用指向新创建的对象。

了解了变量及其赋值后,还应该注意变量的作用域问题。

变量的作用域是变量能够被访问的那块代码,也就是认为变量存在的区域。以下的块定义了一个 Age 变量:

```
{
    int Age=19;
}
```

其中,运用花括号表示开始和结尾。变量 Age 的作用域就局限于块内,如果在作用域外访问,就会发生编译错误。

3.3.2 运算符

变量和常量解决了数据的存储问题,运算符用于解决数据的处理方式,将变量、常量和运算符组合在一起,便形成了表达式。C#提供了大量的运算符,按照其处理操作数的不同大致可以分为三类,即一元运算符、二元运算符和三元运算符。C#的运算符主要有以下几种。

1. 算术运算符

C#的算术运算符如表3.14所示。

对部分运算符的说明如下。

1) 运算符"%"和"/"

对于取模运算符(%)来说,它是用于计算两个整数相除所得的余数。例如:

```
a=7% 4;
```

最终 a 的结果是3,因为7%4的余数是3。

表 3.14 算术运算符

运算符	说　明	表 达 式
+	执行加法运算(如果两个操作数是字符串,则该运算符用作字符串连接运算符,将一个字符串添加到另一个字符串的末尾)	操作数 1 + 操作数 2
−	执行减法运算	操作数 1 − 操作数 2
*	执行乘法运算	操作数 1 * 操作数 2
/	执行除法运算	操作数 1 / 操作数 2
%	取模运算符,获得进行除法运算后的余数	操作数 1 % 操作数 2
++	将变量的值加 1	变量++ 或++变量
−−	将变量的值减 1	变量−− 或−−变量
~	将一个数按位取反	~操作数

如果要求它们的商,则用以下语句:

```
b=7/4;
```

这样 b 就是它们的商了,应该是 1。

也许有的读者就不明白了,7/4 应该是 1.75,怎么会是 1 呢? 这里需要说明的是,当两个整数相除时,所得到的结果仍然是整数,没有小数部分。要想得到小数部分,可以写成 7.0/4 或者 7/4.0,也即把其中一个数变为非整数。

那么怎样由一个实数得到它的整数部分呢? 这就需要用强制类型转换。例如:

```
a=(int)(7.0/4);
```

因为 7.0/4 的值为 1.75,前面加上(int),就表示把结果强制转换成整型,这就得到了 1。

请思考一下 a=(float) (7/4);

最终 a 的结果是多少?

2) 运算符"++"和"−−"

对于运算符++或−−来说,++或−−在变量的前面还是后面对变量本身的影响是一样的,都是加 1 或者减 1,但是当把它们作为其他表达式的一部分时,两者就有区别了。运算符++或−−放在变量前面,那么在运算之前,变量先完成自增或自减运算;如果运算符放在变量的后面,那么自增自减运算是在变量参加表达式的运算后再运算。看下面的例子:

```
num1=4;
num2=8;
a=++num1;
b=num2++;
a=++num1;
```

这总地来看是一个赋值语句,把++num1 的值赋给 a,因为自增运算符在变量的前面,所以 num1 先自增加 1 变为 5,然后赋值给 a,最终 a 也为 5。b=num2++;这是把 num2++的值赋给 b,因为自增运算符在变量的后面,所以先把 num2 赋值给 b,b 应该为 8,然后 num2

自增加1变为9。

如果出现以下的情况怎么处理呢?

```
c=num1+++num2;
```

到底是c=(num1++)+num2;还是c=num1+(++num2);? 这要根据编译器来决定,不同的编译器可能有不同的结果。所以在以后的编程当中,应该尽量避免出现这种比较复杂的情况。

3) 字符串连接运算符"+"

字符串连接运算符"+"的功能是把两个字符串合并成一个字符串,例如:

```
string String1="Welcome " +" Welcome ";
```

则字符串变量string中存储的是字符串"Welcome Welcome"。

2. 比较(关系)运算符

比较(关系)运算符如表3.15所示。

表3.15 比较(关系)运算符

运算符	说 明	表 达 式
>	检查一个数是否大于另一个数	操作数1 > 操作数2
<	检查一个数是否小于另一个数	操作数1 < 操作数2
>=	检查一个数是否大于或等于另一个数	操作数1 >= 操作数2
<=	检查一个数是否小于或等于另一个数	操作数1 <= 操作数2
==	检查两个值是否相等	操作数1== 操作数2
!=	检查两个值是否不相等	操作数1 != 操作数2

比较(关系)运算符是对两个操作数进行比较,返回一个真值或假值。

注意:要区分用来比较两个值是否相等的运算符"==="和用来赋值的赋值运算符"="。

3. 逻辑运算符

逻辑运算符又叫布尔逻辑运算符,如表3.16所示。

表3.16 逻辑运算符

运算符	说 明	表 达 式
&&	对两个表达式执行逻辑"与"运算	操作数1 && 操作数2
||	对两个表达式执行逻辑"或"运算	操作数1 || 操作数2
!	对一个表达式执行逻辑"非"运算	!操作数
^	对两个表达式执行逻辑"异或"运算	操作数1 ^ 操作数2

其中,"&&"、"||"和"^"是二元运算符,"!"是一元运算符。

逻辑运算的结果只有两种取值:true或false。

在C#中,不能认为整数0是false,其他值是true。

例如：

```
bool isExist=false;
bool b=(i>0 && i<10);
```

又如，设置变量 i=5、j=4，则逻辑表达式 i! =j && i>=j 的结果为 true。

4. 赋值运算符和复合(快捷)赋值运算符

赋值运算符和复合赋值运算符如表 3.17 所示。

表 3.17 赋值运算符和复合(快捷)赋值运算符

运算符	说　明	表 达 式
=	是赋值运算符，将变量 2 的值赋予变量 1	变量 1=变量 2
+=	将变量 1 和变量 2 值的和赋予变量 1	变量 1+=变量 2
-=	将变量 1 的值减去变量 2 的值的结果赋予变量 1	变量 1-=变量 2
*=	将变量 1 的值乘以变量 2 的值的结果赋予变量 1	变量 1*=变量 2
/=	将变量 1 的值除以变量 2 的值的结果赋予变量 1	变量 1/=变量 2
%=	将变量 1 的值除以变量 2 的值的余数赋予变量 1	变量 1%=变量 2

赋值语句的作用是把某个常量或变量或表达式的值赋值给另一个变量。符号为“=”。这里并不是等于的意思，只是赋值，等于用“==”表示。

注意：赋值语句左边的变量在程序的其他地方必须声明。

例如：

```
int X1=10;
double Y2=2.34;
```

赋值运算符的右边的值可以是一个表达式，例如：

```
Y2=2.34*3;
```

复合(快捷)赋值运算符的用法，例如：

```
int X1=10;
```

X1+=5；等价于 X1=X1+5；也就是 X1=15；

5. 位运算符

位运算是对位进行比较和操作，其中位是取 0 或 1 的二进制位。共有 6 种位运算符，如表 3.18 所示。

表 3.18 位运算符

运算符	说　明	表 达 式
&	将操作数转换为二进制数，并按位进行与运算	操作数 1 & 操作数 2
|	将操作数转换为二进制数，并按位进行或运算	操作数 1 | 操作数 2
<<	将操作数转换为二进制数，然后左移 n 位	操作数<< n