C#编程魔法书

第二章 C#基本语法

按照微软官方的说法：“C# 中的关键组织结构概念包括程序、命名空间、类型、成员和程序集。 C# 程序由一个或多个源文件组成。程序声明类型，而类型则包含成员，并被整理到命名空间中。类型实例包括类和接口。成员实例包括字段、方法、属性和事件。编译完的 C# 程序实际上会打包到程序集中。程序集的文件扩展名通常为 .exe 或 .dll，具体取决于实现的是应用程序还是库。”本章自底向上来说明C#程序中的各个组成部分，最后再用几个实际的例子来将这些部分组装成一个完整的C#程序。

# 程序结构

一个C#程序由以下这些部分组成：

* 命名空间声明和引用的命名空间
* 类型定义
* 类型的成员方法和变量
* 类型的特性
* 主入口方法 – Main方法
* 语句和表达式
* 注释

我们再回过头看代码清单 1 - 1的代码，每条语句对应的组成部分：

* 代码的第一行是 **using System;** 通过**using**关键字在程序中引入了**System**命名空间中的类型和方法，通常情况下一个程序有多个**using**语句。
* 接下来是命名空间的声明，命名空间是一个类型的集合，在我们的例子里，命名空间*Com.ChinaPub.CSharpMagic*包含了类型*HelloworldApp*。
* 接下来是类型的定义，*HelloworldApp*这个类型包含了程序会用到的数据和方法。类型通常包含多个方法，方法定义了类型的行为。这个例子李*HelloworldApp*只包含了一个方法 –**Main**方法。
* 接下来定义了**Main**方法，是C#程序的入口。Main方法定义了程序的执行行为。
* 接下来的 /\* …… \*/ 是**注释**，编译器通常将注释当做空格处理，注释的作用是帮助其他程序员更好的理解代码的行为。
* **Main**方法中只有一条语句 **Console.WriteLine("Hello,world!");**用来执行，这条语句的作用是在命令行上打印一条消息：Hello, world!。*WriteLine*是类型*Console*的一个方法，而*Console*是*System*命名空间中的一个类型，通过前面的**using**关键字引入。

还有几点值得说明：

* C#是大小写敏感的，因此*main*和*Main*是两个不同的方法。
* C#中所有的语句都以英文字符分号（；）结尾。
* C#程序从**Main**方法开始入口执行。
* 与Java语言不同，C#源文件名不需要与类型名一致。

# 基础语法

## 注释

C#语言支持三种注释方式，其支持以双斜杠“//”开头的单行注释，也支持以“/\*”和“\*/”包含起来的多行注释，另外还支持以三个斜杠“///”开头的文档注释。C#编译器有一个编译选项，可以将源码中的文档注释提取出来单独保存到一个XML文件中，其他程序可以通过解析这个XML文件提供丰富的源码文档支持，如代码清单2 - 1。

代码清单2 - 1 C#注释示例

|  |
| --- |
| 1 // 源码位置：第二章\CommentDemo.cs  2 // 编译命令：csc /doc:commentdemo.xml CommentDemo.cs  3 using System;  4  5 namespace CommentDemo  6 {  7 ///<summary>  8 ///文档注释，这里可以写上对Program类型的注释说明  9 ///</summary>  10 class Program  11 {  12 ///<summary>  13 /// Main方法的注释说明，如：  14 /// Main方法是C#程序的入口方法  15 ///</summary>  16 ///<param name="args">这是参数args的注释说明</param>  17 static void Main(string[] args)  18 {  19 // 以双斜杠开头是单行注释  20 Console.WriteLine("Hello World!");  21 /\*  22 \* 这是多行注释  23 \* 下面的语句的作用是等待用户输入任意一行文本  24 \*/  25 Console.ReadLine();  26 }  27 }  28} |

要提取文档注释，可以使用csc.exe编译器的“/doc”选项指明提取出来的文档注释的保存位置，如：

|  |
| --- |
| csc /doc:commentdemo.xml CommentDemo.cs |

在命令行里一个个列出要提取文档注释的源码文件位置非常繁琐，因此一般通常使用Visual Studio IDE的工具来直接提取：

1. 在Visual Studio中右键单击需要提取文档注释的工程；
2. 在右键菜单中点击“属性”菜单；
3. 在属性面板中点击“生成”页签；
4. 在“生成”面板上拉到最下面，在“输出”节勾选“XML文档文件”复选框，如图2 - 1；
5. 再次生成工程，在bin文件夹的输出目录即可看到生成的文档注释XML文件，文件名一般以工程名命名。

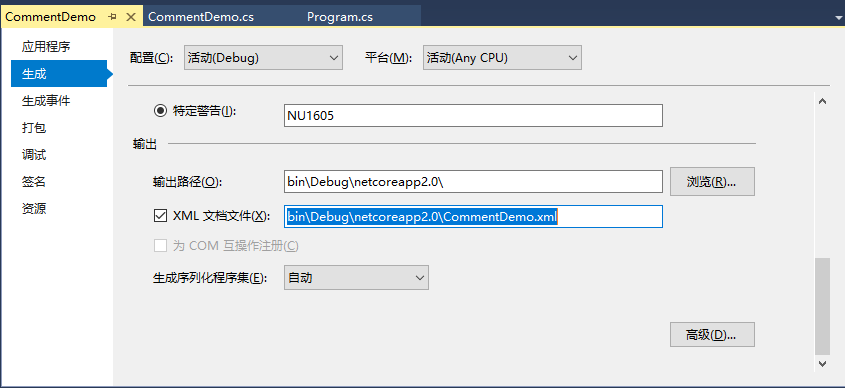


图2 - 1 Visual Studio中生成文档注释

## 类型与变量

程序是用来处理和计算数据的，因此每个程序都至少有两个基本元素，用以保存数据的元素和用来表达计算过程的语句。本节我们先讨论存储数据相关的元素：变量和常量。

由于计算机在内存或磁盘中保存数据，而内存或磁盘只能保存二进制格式的数据，不同的数据需要使用不同的方式表达，所以为了描述数据，编程语言提供了类型的概念。大部分面向对象的编程语言有两种类型系统：一种是编程语言自带的类型 – 基本类型（primitive types），基本类型一般就是简单类型，如字符，字符串和数字等，一种是由用户自建的类型（classes）。然而使用两种类型系统有不少的兼容性问题，例如要创建一个包含integer和double类型的数组，则不得不需要给Integer和double创建一个继承自相同基类的类型，一个封装integer，另一个封装double，这种封装是一个不必要的操作。因此C#语言里，所有类型都继承自System.Object类，所有新的类型在定义时，如果不显式指定继承关系，默认就是从System.Object类中继承。

然而所有类型都继承自同一个System.Object类型会带来较大的性能损耗，这是因为所有的对象都是在内存的堆（Heap）上创建，而操作系统一般会对栈做一些性能优化处理，例如将栈上的一些数据加载到CPU的寄存器上提高读写速度。因此为了能在栈上创建对象，C#的类型系统区分出两种类型：值类型（value types）和引用类型（reference types）。其中值类型在栈上创建，而引用类型在堆上创建。除了内存分配的区别以外，值类型包含的是实际数据，而引用类型类似C里的指针，包含的指向实际数据的内存地址。因此修改值类型的值是修改其包含的实际数据，而修改引用类型的值只是修改了内存地址，实际上是将值类型的值指向了另外一块实际数据，两者区别如图2 - 2所示：

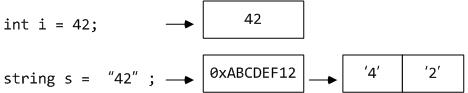


图2 - 2 值类型与引用类型

C#里值类型包括数字和字符等基本类型、枚举类型（enumerator）和结构体（structure）类型，而引用类型包括类（class）、数组（array）、接口（interface）和委托（delegate）。与大部分编程语言一样，C#也内置了一些基本类型，同时因为.NET Framework是一个跨语言的编程框架，其允许跨语言组件互用，即Visual Basic.NET程序可以调用C#编写的函数库组件，反之亦然，所以C#的内置类型在.NET Framework里有对应的类型，或者说C#的内置类型是.NET Framework对应类型的别名（alias），表2 - 1是C#内置的数据类型及其说明：

表2 - 1 C#内置基本类型说明

|  |  |  |
| --- | --- | --- |
| C# 类型 | .NET Framework类型 | 说明 |
| bool | System.Boolean | 值类型，取值为true或false的布尔值，默认值是false |
| byte | System.Byte | 值类型，取值范围为0到255的无符号字节，默认值是0。 |
| sbyte | System.SByte | 值类型，取值范围为-128到127的有符号字节，默认值是0。 |
| char | System.Char | 值类型，无符号16位Unicode字符，默认值是’\0’。 |
| decimal | System.Decimal | 值类型，高精度无四舍五入小数，常用于金融计算，默认值是 0.0m。 |
| double | System.Double | 值类型，双精度浮点数，默认值是0.0d。 |
| float | System.Single | 值类型，单精度浮点数，默认值是0.0f。 |
| int | System.Int32 | 值类型，有符号32位整数类型，默认值是0。 |
| uint | System.UInt32 | 值类型，无符号32位整数类型，默认值是0。 |
| long | System.Int64 | 值类型，有符号64位整数类型，默认值是0。 |
| ulong | System.UInt64 | 值类型，无符号64位整数类型，默认值是0。 |
| object | System.Object | 引用类型，所有类型的基类，默认值是null。 |
| short | System.Int16 | 值类型，有符号16位整数类型，默认值是0。 |
| ushort | System.UInt16 | 值类型，无符号6位整数类型，默认值是0。 |
| string | System.String | 引用类型，字符串对象，默认值是null。 |

程序在计算过程中，总是会随之产生中间结果，保存这些会变化的数据的元素称为变量（variable），而保存在程序运行过程中不会变化的数据称为常量（constant），例如π就是一个典型的常量。C#中变量命名只允许使用字母、数字和下划线这些字符，而且变量名只能以字母或者下划线开头，并且不能使用关键字命名，在Visual Studio IDE中编辑代码时，关键字默认会以蓝色字体显示[[1]](#footnote-1)。如“value1”，”\_value”这些都是合法的变量名，但是”1value“则不是合法的变量名。但有些场景下，变量名可能会跟关键字重复，例如在处理数据库表到C#对象映射，或者反序列化JSON的场景下，表字段或JSON对象的字段名跟关键字冲突，这个时候可以在关键字前面加“@”字符转义，如“@decimal”是一个合法的变量名，而“decimal”是C#中的关键字。“@”被称为原义识别符（verbatim identifier），还可以用在字符串中，用来将转义字符当来原始含义使用，如

|  |
| --- |
| @"c:\Docs\Source\a.txt" // 等同于："c:\\Docs\\Source\\a.txt"  @"""Ahoy!"" cried the captain." // 等同于："Ahoy!" cried the captain. |

在变量类型后面跟上变量名，就可以声明一个变量，也可以在类型后面跟上多个以逗号分隔的变量名来声明多个变量。在C#中，为了避免C/C++常见的使用未初始化内存 — 野指针的问题，是不允许在代码里使用未初始化变量的，使用未初始化变量会引发一个编译错误。因此在声明变量时，一般都会给变量赋予一个初始值，变量第一次赋值后，也被称作定义变量。这是因为对于从未赋值甚至是从未使用过的变量，编译器通常会在编译时抛弃掉这个变量。对于数字类型，有多种类型可以展现数字，如小数有三种类型，整数有更多的类型，需要在数字后面加一个后缀字母表示数字的类型，这个后缀字母大小写不敏感。如代码清单2 - 2的第7、8和9行。

* 后缀“d”表示双精度浮点数，也可以用“D”表示；“f”和“F”表示单精度浮点数，“m”和“M”表示高精度小数。
* 后缀“l”和“L”表示长整形**long**，“u”和“U”表示无符号整数。

C# 3.0开始，提供了一个语法糖优化，在定义变量时使用**var**关键字来由编译器推导出变量的类型，如第6行的变量index，编译器会自动推导出类型为int – 即最常用的整数类型。这个语法糖与JavaScript等动态语言不同，在这些动态语言中，变量是弱类型或者说没有类型的，**var**关键字只是定义了一个变量，而没有指明变量类型。C#中是强类型语言，**var**关键字的作用是在后面方便修改变量类型，具体的类型由编译器推导，在最终生成的IL语言中，定义的变量类型由编译器替程序员指明。

C#支持使用二进制、十进制和十六进制定义整形变量，在C# 7.0之后允许使用下划线“\_”来作为数字分隔符，如第11行byteValue1是采用十六进制赋值，byteValue2采用二进制赋值。

代码清单2 - 2 声明和定义变量

|  |
| --- |
| 1 int value;  2 int itemCount, employeeNumber;  3 // 因为value没有任何初始值，所以下面的语句会导致一个编译错误  4 Console.WriteLine("{0}", value);  5 int employeeCount = 10000;  6 var index = 0;  7 double defaultDoubleValue = 0.0, doubleValue = 0.0d;  8 float floatValue = 0.0f;  9 decimal decimalValue1 = 0.0m, decimalValue2 = 0.0M;  10 ulong ulongValue = 123ul;  11 byte byteValue1 = 0x00AB, byteValue2 = 0b1100\_1001; |

程序在计算过程中，总是会随之产生中间结果，保存这些会变化的数据的元素称为变量（variable），而保存在程序运行过程中不会变化的数据称为常量（constant），例如π就是一个典型的常量，定义常量时，除了需指明常量的类型，还需要用**const**关键字表明其是常量，试图修改一个常量会引发编译错误。在C#中还有一种常量，它的值在编写程序时不确定，但在启动时可以通过配置文件等外部信息获取，一旦赋值就不再允许更改，这种常量使用**readonly**关键字定义，如代码清单2 - 2的第9行。对于**const**常量定义，赋值时可以使用基于其它常量组成的计算表达式，C#编译器会自动在编译期计算并将结果赋值给常量。有些常量计算表达式的结果可能会超过常量类型的取值范围，这时编译器会自动报告编译错误，如代码清单2 - 2的第21行。但如果计算表达式里包含变量的化，由于变量的值在运行过程中可能会变更，编译器不会执行这个优化进而也无法捕捉到潜在的超出取值范围的错误，这种情况下可以使用**checked**关键字，在运行时由CLR执行引擎来捕捉这个错误，如代码清单2 - 2的第28行和31行。对于超过32位整数的取值范围的常量赋值表达式，C#编译器也会报告超出取值范围的编译错误，这种情况下可以使用unchecked关键字禁用这种检查，如第43行。

代码清单2 - 3 变量常量以及unchecked关键字等示例

|  |
| --- |
| 1 // 源码路径：第二章\DataDemo.cs  2 // 编译命令：csc DataDemo.cs  3 using System;  4  5 public class DataDemo  6 {  7 constdouble PI = 3.1415926; // 常量  8 // readonly常量，其值不能在编译器确定  9 static readonly long ticks = DateTime.Now.Ticks;  10  11 public static void Main()  12 {  13 Console.WriteLine($"PI:\t{PI}");  14 Console.WriteLine($"ticks:\t{ticks}");  15  16 constint nine = 9;  17  18 // 取消下面的注释，编译器会报告下列编译错误  19 // error CS0220: 在 checked 模式下，运算在编译时溢出  20 //  21 // int i1 = 2147483647 + nine;  22 // Console.WriteLine($"i1:\t{i1}");  23  24 int ten = 10;  25 int i2 = 2147483647 + ten;  26 Console.WriteLine($"i2:\t{i2}");  27  28 int i3 = checked(2147483647 + ten);  29 Console.WriteLine($"i3:\t{i3}");  30  31 checked  32 {  33 int i4 = 2147483647 + ten;  34 Console.WriteLine($"i4:\t{i4}");  35 }  36  37 // 取消下面的注释，编译器会报告下列编译错误  38 // error CS0220: 在 checked 模式下，运算在编译时溢出  39 //  40 // long l1 = 2147483647 + nine;  41 // Console.WriteLine($"l1:\t{l1}");  42  43 long l2 = unchecked(2147483647 + nine);  44 Console.WriteLine($"l2:\t{l2}");  45 }  46} |

## 装箱和拆箱

当把值类型变量赋值给引用类型变量时，就触发了一个装箱（boxing）操作，这是因为值类型直接保存变量数据，而引用类型保存内存地址，赋值时相当于创建了一个新的引用变量，将原来值类型的数据拷贝一份，新的变量指向这份数据，有点类似快递打包，因此被称为装箱。注意装箱操作和C/C++语言中的定义一个指向值类型变量的指针不一样，指针是指向原始的值类型数据，而装箱会将值类型的数据拷贝一份，引用变量指向的是这份拷贝，如代码清单2 - 3第12-13行演示的一样。而拆箱过程就是反过来，将引用变量封装的值类型数据取出来。

代码清单2 - 4 装箱拆箱过程示例

|  |
| --- |
| // 源码：第二章\BoxAndUnboxing.cs  1 using System;  2  3 public class BoxAndUnboxing  4 {  5 public static void Main()  6 {  7 int foo = 42; // 值类型  8 // 将值类型赋值给引用类型，引发装箱  9 object bar = foo;  10  11 // 修改foo的值不会影响bar的值  12 foo = 43;  13 Console.WriteLine(bar + "," + foo);  14  15 // 拆箱回值类型  16 int oof = (int)bar;  17 // 隐含两个装箱，一个拆箱操作  18 Console.WriteLine(oof + "," + (int) bar);  19  20  21 object[] arr = new object[] { foo, bar };  22 // 没有装箱和拆箱操作  23 Console.WriteLine(arr[0] + "," + arr[1]);  24 }  25} |

装箱过程涉及到在堆上分配内存和拷贝数据等操作，因此装箱和拆箱是有一些性能损耗的，在需要进行大量计算的程序如游戏里，这些小的性能损耗汇总起来也是很客观的，需要在编程的时候注意不必要的装箱拆箱操作，如代码清单2 - 3的第18行就有几个不必要的性能损耗，因为WriteLine方法接受字符串类型的参数，需要将oof转换成字符串也就是引用类型，转换过程有一个隐式的装箱操作，而bar在使用前执行的拆箱操作变回了值类型，到使用的时候又需要转换成引用类型，这些都是应该避免的性能损耗。使用ildasm程序打开编译后的BoxAndUnboxing.exe可执行文件，18行对应的IL代码如代码清单2 - 3所示。在IL\_002d处使用box指令将oof变量装箱，接着在IL\_0038处使用unbox.any指令拆箱bar变量，紧接着IL\_003d又使用box指令重新装箱后，才使用call指令调用String.Concat方法拼接出可以打印的字符串。

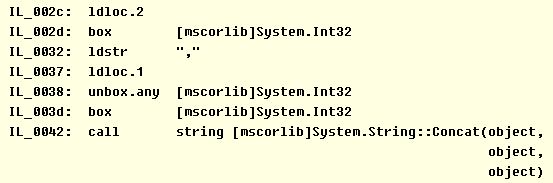


图2 - 3BoxAndUnboxing.exe的IL代码

## 浮点数类型的问题

计算机中保存数据的方式是以二进制格式保存的，而且通常的数据结构都是固定字节大小的，而且计算机内存是有限的，无法表现无限精度的数字，对于浮点数，特别是二进制浮点数，这往往会导致一些问题，如代码清单 2- 1：

代码清单 2- 1浮点数精度问题示例

|  |
| --- |
| 1 // 源码位置：第二章\FloatingPointDemo.cs 2 // 编译命令：csc FloatingPointDemo.cs 3 using System; 4 5 public class FloatingPointDemo 6 { 7     public static void Main() 8     { 9         float a, b, c; 10         a = 1.345f; 11         b = 1.123f; 12         c = a + b; 13         Console.WriteLine(c == 2.468f); 14         Console.WriteLine(c == 2.468); 15         Console.WriteLine((double) c == 2.468); 16 17         float z = 0.1f; 18         for (int i = 0; i< 10; ++i) 19             z += 0.1f; 20         Console.WriteLine( 21             $"0.1f \* 10 == 1.0f? {z == 1.0f}"); 22     } 23 } |

编译运行后，13行打印的结果是True，但第14行的打印结果却是False。发生这样的情况，跟计算机中浮点数的表示方法有很大关系。常见的**float**和**double**类型采用的是二进制方式表示浮点数的方式，其在内存中并不是存储浮点数的数值，存储的而是浮点数基于二进制表示的科学记数公式，如32位的**float**类型实际存储的是如图2 - 4的公式：

图2 - 4float类型的记数公式

其中：

s：代表符号位的值，占用1位，如果是0则代表正数，为1则代表负数；

e：指数部分，是英文Exponent的简写，占用8位；

m：尾数部分，是英文Mantissa的简写，占用23位。

我们先用下面的十进制的科学记数法来熟悉上面的公式，其中1.23是尾数部分（Mantissa），而6则是指数部分（Exponent）：

图2 - 5是浮点数0.15625在内存中按照的图2 - 4公式的二进制表现形式：

1. 符号位的值是0，表示这是一个正数；
2. 指数部分e是01111100，十进制是124，该值可以是正数 – 用来表示大于1的数，为负数时则表示小于1的小数 – 甚至是非常小的小数，因为指数部分没有符号位，因此通过减去127来计算得出符号。
3. 尾数部分的计算有些技巧，由于科学记数法里，尾数总是会归一化到1.xxx的形式，如111.00101101在十进制科学记数法里会记成尾数是 1.1100101101，指数是2的形式，而0.0001011011则会记成尾数是1.011011，指数是-4的形式。这样可以在存储的时候把尾数小数点前面的1省略掉，在计算的时候加回来，从而可以多表达一些数字。因此m的值按照公式的计算结果如下，值是1.25：  
    =   
   1 + 0 \* 2-1 + 1 \* 2-2 + 0 \* 2-3 + …… =   
   1.25。
4. 最后得到整个浮点数的值是：1.25 x 2-3 = 0.15625。

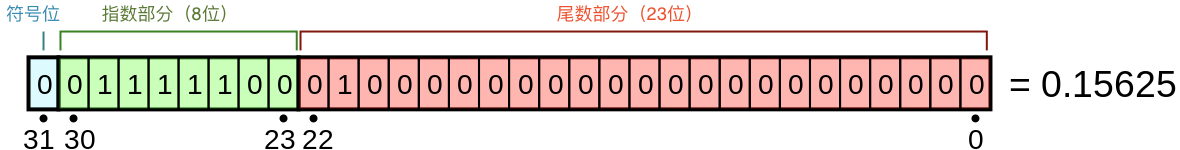


图2 - 5 float型浮点数在内存中的表现形式

64位的**double**类型的内存结构与之类似，记数公式为：

图2 - 6double类型的记数公式

其中指数部分e占用11位，尾数部分m占用52位。

回到代码清单 2- 1中的例子，在本书附带的示例代码中，笔者添加了代码清单 2- 2以打印**float**类型与**double**类型变量的内存表现形式，运行后可以查看到代码清单 2- 1中的变量a、b和c的表现形式分别如下所示，可以看到由于变量c和2.468f都是32位的float类型，两个变量值的二进制表现形式是完全一样的，这也就是对比能够成功的原因。然而将c与2.468对比时，C#中的小数如果不指定后缀的话，默认是64位的**double**类型，需要将c隐式转换成double类型才能做对比，转换后的c的二进制形式与2.468的二进制相差很大，结果导致对比失败。

* **a**: 0 01111111 01011000010100011110110
* **b**: 0 01111111 00011111011111001110111
* **c**: 0 10000000 00111011111001110110110
* **2.468f**： 0 10000000 00111011111001110110110
* **2.468**：  
  0 10000000000 0011101111100111011011001000101101000011100101011000
* **将float类型的2.468f转换成double类型的2.468：**0 10000000000 0011101111100111011011000000000000000000000000000000

代码清单 2- 2打印数字的二进制表现形式

|  |
| --- |
| 1 private static string ToBinaryString(float value) 2 { 3     byte[] bytes = BitConverter.GetBytes(value); 4     string s = ToBinaryString(bytes); 5     // 符号位 指数部分 尾数部分 6     return s.Substring(0, 1) + " " + s.Substring(1, 8) 7          + " " + s.Substring(9); 8 } 9 10 private static string ToBinaryString(byte[] bytes) 11 { 12     System.Text.StringBuilder builder = 13         new System.Text.StringBuilder(); 14     foreach (Byte b in bytes) 15     { 16         for (int i = 0; i< 8; i++) 17         { 18             builder.Insert(0, 19                 ((b >> i) & 1) == 1 ? "1" : "0"); 20         } 21     } 22 23     return builder.ToString(); 24 } |

代码清单 2- 1中第9行到第15行只演示了不同精度的float类型与double类型的比对问题，第17行到第21行演示了二进制浮点数的本身的精度问题，第20行的打印结果是False，具体原因就留给有兴趣的读者自己研究了。

除了上文描述的问题，不同语言对浮点数的处理精度也是不一样的，例如3 \* 0.3这个表达式，在笔者写作的MacBook Pro（操作系统：macOS 10.13.6）上的python和node中的运行结果都是0.8999999999999999，而在dotnetcore中的运行结果则是0.9。因此在对比两个二进制浮点数值时，不能采用 == 号来对比，而只能通过对比两个值相减结果的绝对值是否小于容许误差范围内来实现，如：

Math.Abs((3 \* 0.3) - 0.9) < 0.00000000001

为了解决上文描述的二进制浮点数的精度问题，.NET提供了Decimal类型，也称作高精度小数类型，用在如财务运算等对精度要求非常高的场景下。然而高精度带来的副作用就是其内存占用空间和运算速度相对于二进制浮点数都略逊一筹，decimal类型占用128位字节，运算速度要比float和double类型慢好几倍。

代码清单 2- 3高精度小数decimal类型示例

|  |
| --- |
| 1 // 源码位置：第二章\DecimalDemo.cs 2 // 编译命令：csc DecimalDemo.cs 3 using System; 4 5 public class DecimalDemo 6 { 7     public static void Main(string[] args) 8     { 9         int times = int.Parse(args[0]); 10        float f1 = 1.0f / times, f2 = 0.0f; 11        double d1 = 1.0d / times, d2 = 0.0d; 12        decimal c1 = 1.0m / times, c2 = 0.0m; 13 14        DateTime begin = DateTime.Now; 15        for (int i = 0; i<times; ++i) 16            f2 += f1; 17        TimeSpan period = DateTime.Now - begin; 18        Console.WriteLine( 19            $"float类型计算耗时：{period.TotalMilliseconds}" + 20            $"，最终结果：{f2}"); 21 22        begin = DateTime.Now; 23        for (int i = 0; i<times; ++i) 24            d2 += d1; 25        period = DateTime.Now - begin; 26        Console.WriteLine("double类型计算耗时：" + 27            $"{period.TotalMilliseconds}，最终结果：{d2}"); 28 29        begin = DateTime.Now; 30        for (int i = 0; i<times; ++i) 31            c2 += c1; 32        period = DateTime.Now - begin; 33        Console.WriteLine($"decimal类型计算耗时：" + 34            $"{period.TotalMilliseconds}，最终结果：{c2}"); 35 36        int[] bits = decimal.GetBits(c2); 37        Console.WriteLine( 38          "{0,8}  {1,10:X8}{2,10:X8}{3,10:X8}{4,10:X8}", 39          "值", "Bits[3]", "Bits[2]", "Bits[1]", "Bits[0]"); 40        Console.WriteLine( 41          "{0,8}  {1,10:X8}{2,10:X8}{3,10:X8}{4,10:X8}", 42          "---------", "--------", "--------", 43          "--------", "--------"); 44        Console.WriteLine( 45          "{0,8}  {1,10:X8}{2,10:X8}{3,10:X8}{4,10:X8}", 46          c2, bits[3], bits[2], bits[1], bits[0]); 47 48        Console.WriteLine(1.0m / 3.0m \* 3.0m); 49        Console.WriteLine(Math.Round(1.0m / 3.0m \* 3.0m)); 50    } 51 } |

在笔者写作的机器上编译代码清单 2- 3并执行第9行到第35行的浮点数加法10000000次的计算结果如图2 - 7所示。decimal类型的计算速度是double类型的6倍多，是float类型的近4倍，但是精度方面，仅仅是将1的一千万分之一累加一千万次，double和float类型的计算结果竟然不是1，特别是float类型的计算结果跟预期结构相差很大，而decimal类型的结果就相当精确了。

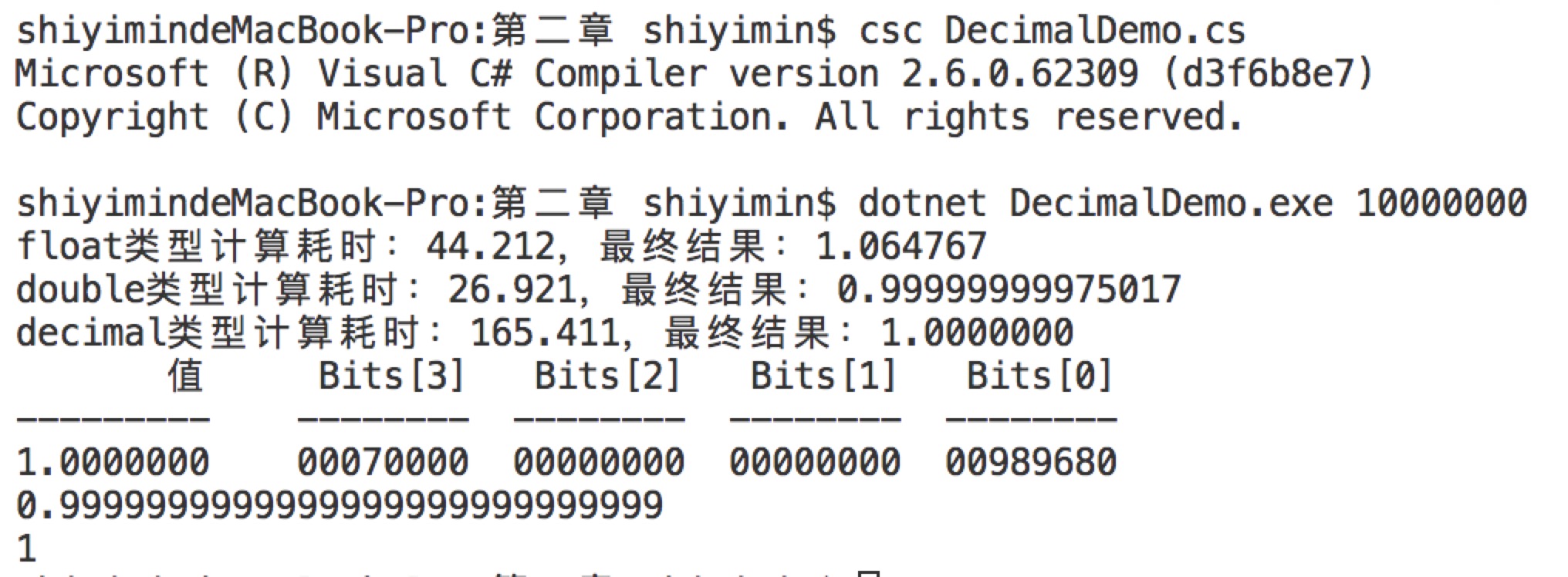


图2 - 7浮点数运行精度和速度对比

decimal类型的高精度得益于其是采用十进制的科学记数法公式，在内存中实际上保存的是下图的公式：

其中：

* 尾数部分m占用96位（第0 – 95位），即前面三个int32整数；
* 接下来的16位（第96 – 111位）暂未使用；
* 指数部分e占用8位（第112 – 119位），虽然占用了8位，但指数的取值范围只能是0 – 28，这是因为尾数部分表示的最大整数最多只有28个数字，即 296 = 7.92282 x 1028；
* 接下来的7位（第120 – 126位）暂未使用；
* 最后一位是符号位s（第127位），如果是0则代表正数，为1则代表负数；

代码清单 2- 3的第36行到第46行打印了decimal的内存表现形式，C#的decimal类型甚至能够记住小数点后0的个数，如代码清单 2- 3中36行打印的c2实际上是由0.0000001累加一千万次的结果，因此其指数部分的数值是7，而尾数部分的数值是十六进制值989680，即十进制的10,000,000。如果将36行打印的c2替换成1m，则GetBits返回的会是：00000000 00000000 00000000 00000001。

虽然采用十进制记数法的decimal类型的精度大大提高了，但是还是不能避免类似代码清单 2- 3中第48行的精度丢失问题，在处理时还是需要采用第49行演示的办法通过四舍五入来得到结果。

## 表达式和语句

程序是由序列的语句（statement）组成的，运行程序时，这些语句按照编码的顺序一条条执行。在编码时通常需要将多条语句组合在一起形成一个语句块，语句块使用“{”和“}”大括号包围。语句大部分时候会包含表达式（expression），表达式是可以计算出一个值的代码，如“1 + 2 / x”就是一个表达式，表达式由操作数（oprand）和操作符（operator）组成，操作数包含变量、常量、方法调用，而操作符包括加减乘除计算运算符等[[2]](#footnote-2)。虽然不是语法上的强制要求，一般的编程规范都建议在源码中，在表达式中，用空格隔开操作数和操作符，便于代码阅读，如“a + 3 \* PI – func(4)”这样的表达式。

在语句块里定义的变量称为局部变量（local variable），局部变量只在其定义的语句块内可见，因此语句块（block）也称为作用范围域（scope）。语句块可以嵌套任意多的语句块，也可以嵌套任意多层的语句块，定义在内层语句块里的局部变量将会覆盖外层语句块中定义的同名局部变量，如图2 - 4，为了避免这种意外的变量名覆盖问题，将局部变量尽可能定义在最里层的语句块是一个良好的编程习惯。

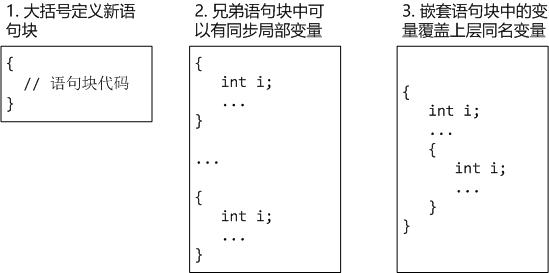


图2 - 8 语句块与局部变量

与其他编程语言类似，C#里的语句可以分成三大类：

* 选择语句：**if**和**switch**语句被称作选择语句，它们根据一个条件表达式的计算结果选择性的执行接下来的语句。
* 循环语句：**while**, **do**, **for**和**foreach**语句只要指定的条件表达式的计算结果为true，则重复执行一段语句块。
* 跳转语句：**goto**，**break**和**continue**语句无条件跳转到其它语句上执行。

### if语句

If语句是主要的决策选择语句，其可以匹配一个可选的else子句。C#里if语句的语法如表2 - 2，如果布尔表达式的结果计算为true，则选择first-embedded-statement执行，否则选择second-embedded-statement执行。first-embedded-statement和second-embedded-statement都是语句块。如果只包含单条语句，可以忽略包括语句块的大括号。

表2 - 2 if语句的语法

|  |
| --- |
| if ( boolean-expression )  first-embedded-statement  else  second-embedded-statement |

与C语言不同，C#中不能将整数隐式作为布尔值处理，这是因为经常有程序员利用这个“技巧”带来意外的程序的错误，所以C#里禁用这种做法，因此代码清单2 - 4的代码在C#中都是不允许的：

代码清单2 - 5 C#中不允许将整数当做布尔值处理

|  |
| --- |
| int x;  ...  if (x) ... // C#中必须写成 x != 0  if (x = 0) ... // C#中必须写成 x == 0 |

可以通过**else if**来串联多个**if**语句，通过这种串联可以支持到任意多的分支。然而串联的if语句是互斥的，即多个**else if**语句中只有第一个匹配成功的**else if**语句会执行，考虑代码清单2 - 5的例子，在命令行里输入6，程序只会打印“偶数”，而不会打印“3的倍数”这个消息，也就是说第12行的**else if**语句没有执行。

**if**语句之间也可以嵌套其它**if**子句，在使用嵌套**if**的时候需要注意**else**子句的匹配问题，如代码清单2 - 5里的第23 – 27行，虽然从代码缩进来看，**else**子句应该是跟第一个**if**子句匹配的，然而这仅仅是程序员一厢情愿的做法而已，编译器在匹配的时候是从最内层的**if**子句开始往外匹配，即匹配的是第二个**if**语句。通过这个例子，笔者有几个建议给到编程初学者：

* 写任何代码都要严格执行缩进，很多初学者舍不得在代码里多敲几个空格，然而往往这种早期偷懒的做法往往在很后期才会呈现出问题，需要耗费更多的时间去调查和修复问题，这种情况也被戏称为技术债务，早期偷得懒在后面需要连本带息的偿还。
* 任何语句块都应该用大括号包裹起来，笔者在代码清单2 - 4中第11 – 15行的做法是为了节省篇幅刻意为之，请读者避免效仿。

代码清单2 - 6 串联和嵌套if语句的问题

|  |
| --- |
| 1 // 源码位置：第二章\IfDemo.cs  2 using System;  3  4 public class IfDemo  5 {  6 public static void Main()  7 {  8 int input = Console.Read();  9 int value = input - '0';  10  11 if (value % 2 == 0) Console.WriteLine("偶数！");  12 else if (value % 3 == 0) Console.WriteLine("3的倍数！");  13 else if (value == 9) Console.WriteLine("9！");  14 else if (value == 0) Console.WriteLine("输入了0！");  15 else Console.WriteLine("奇数");  16  17 // 缩进不匹配是非常危险的  18 // 下面的else其实是匹配第二个if  19 if (value >= 0 && value <= 10)  20 if (value > 5)  21 Console.WriteLine("及格");  22 else  23 Console.WriteLine("超出成绩范围！");  24 }  25} |

### switch语句

如果需要处理复杂的条件逻辑，要嵌套多层**if**语句，这种情况可以考虑使用**switch**语句。switch语句在C#7.0之后有一个语法扩展，支持模式匹配，本节讨论C# 7.0之前的swtich语法，模式匹配将在后文探讨。switch语句的语法如表2 - 3，其由一到多个case标签组成，每个标签匹配一个常量，不能将多个常量组合在一起放到一个case标签上，每个常量必须要有其自身的case标签。**switch**语句块里面可以定义局部变量，作用范围就是整个switch语句块。C# 7.0之前，**switch**子句只能计算整数、char、枚举型和字符串这些类型的表达式，或者是用户自定义转换到这些类型的表达式。

注意：与Java、C和C++不同，C#的**switch**子句支持字符串，因此在**case**标签里匹配**null**值也是允许的。

表2 - 3 swtich语句的语法

|  |
| --- |
| switch ( expression )  {  case constant-expression:  statement-list;  default:  statement-list;  } |

在执行过程中，首先看有没有**case**标签的常量与**switch**子句的表达式计算结果匹配，如果有，则跳转到这个**case**标签下面的语句块里执行；如果没有**case**标签匹配，且有**default**标签的话，跳转到**default**语句块执行，否则退出**switch**语句。可以将多个case标签组合在一起，如代码清单2 - 6，每个case标签的常量必须唯一，否则会导致编译错误。

代码清单2 - 7 可以将多个case子句组合在一起

|  |
| --- |
| enum MonthName { January, February, ..., December }  MonthName current;  int monthDays;  ...  switch (current)  {  case MonthName.February:  monthDays = 28;  break;  case MonthName.April:  case MonthName.June:  case MonthName.September:  case MonthName.November:  monthDays = 30;  break;  default :  monthDays = 31;  break;  } |

与Java、C和C++不同，C#不允许从一个标签隐式滑落到下一个标签，即第一个**case**语句块里的最后一个语句执行完毕后，不能顺序执行下一个**case**语句块的语句，必须使用**break**语句、**goto**语句、**return**语句或者**throw**语句转移代码执行顺序。如表2 - 4左边的代码里，**break**语句是在**if**语句内部，有一个分支–即隐含的**else**分支没有覆盖，因此有隐式滑落的问题，这个问题会导致编译错误。右边的代码即是相应的修复，采用了三目运算表达式解决了左右代码的分支问题。

表2 - 4 隐式滑落的问题和相应的修复

|  |  |
| --- | --- |
| 错误的代码 | 正确的代码 |
| string suffix = "th";  switch (days % 10) {  case 1 :  if (days / 10 != 1) {  suffix = "st";  break;  }  //编译错误：将会滑落到下个标签  case 2 :  if (days / 10 != 1) {  suffix = "nd";  break;  }  //编译错误：将会滑落到下个标签  case 3 :  if (days / 10 != 1) {  suffix = "rd";  break;  }  //编译错误：将会滑落到下个标签  default :  suffix = "th";  //编译错误：将会滑落到下个标签  } | switch (days % 10) {  case1 :  suffix =  (days / 10 == 1) ? "th" : "st";  break;  case 2 :  suffix =  (days / 10 == 1) ? "th" : "nd";  break;  case 3 :  suffix =  (days / 10 == 1) ? "th" : "rd";  break;  default :  suffix = "th";  break;  } |

表2 - 4左边的隐式滑落还有一种修复办法，即采用**goto**语句来达到滑落的效果，如代码清单2 - 7：

代码清单2 - 8 使用goto语句实现switch的隐式滑落

|  |
| --- |
| switch (days % 10) {  case 1 :  if (days / 10 != 1) {  suffix = "st";  break;  }  gotocase 2;  case 2 :  if (days / 10 != 1) {  suffix = "nd";  break;  }  gotocase 3;  case 3 :  if (days / 10 != 1) {  suffix = "rd";  break;  }  gotodefault;  default :  suffix = "th";  break;  } |

### while和do语句

**while**语句是最简单的循环语句，只要**while**中的布尔表达式计算为**true**，则重复执行后面的语句块。表2 - 5是while语句的语法，注意第6行的update部分，很多编程初学者常犯的错误是忘记添加update部分更新循环的条件，如忘记添加表2 - 5第12行，导致无限循环造成程序假死问题。

表2 - 5 while循环的语法

|  |
| --- |
| 1 initializer;  2  3 while ( boolean-expression )  4 {  5 embedded-statement;  6 update;  7 } |

**do**语句与**while**语句类似，不同的是**do**语句中先执行要重复执行的语句块，在最后评估**while**中的表达式，如果为**true**则重复执行，否则退出循环。

代码清单2 - 9 while和do语句示例

|  |
| --- |
| 1 // 源码位置：第二章\whiledemo.cs  2 using System;  3  4 public class WhileDemo  5 {  6 public static void Main()  7 {  8 int i = 0;  9 while (i< 10)  10 {  11 Console.WriteLine(i);  12 i++;  13 }  14  15 i = 0;  16 do  17 {  18 Console.WriteLine(i);  19 i++;  20 } while (i< 10);  21 }  22} |

### for和foreach语句

在使用**while**循环的时候，更新控制循环是否继续执行的控制变量，即（表2 - 5第6行）的update部分通常和布尔表达式部分（ 表2 - 5第3行）距离较远，容易被忽略而导致编程错误。可以使用**for**语句来避免这种问题，**for**语句的语法如表2 - 6：

表2 - 6 for语句的语法

|  |
| --- |
| for ( initializer ; condition ; update )  embedded-statement |

注意：虽然表2 - 6中**for**语法的update子句写在要重复执行的embedded-statement子句之前，但实际上在运行时update子句是在embedded-statement子句后面执行的。

initializer、condition和update子句都是可选的，一个空的condition子句默认的计算结果是**true**，除非使用**break**语句跳出，否则会导致无限循环。**for**语句中的intializer子句可以定义只在for语句作用范围内可见的局部变量– 也称为循环变量，这点和**while**语句非常不同，**while**语句不能在condition子句里定义局部变量，只能在embedded-statement语句块中定义局部变量。在initializer和update子句中可以写多个表达式来定义和更新多个循环变量，但是condition子句只能计算一个结果，多个条件需要使用“&&”、“||”等布尔操作符来连接，如：

for ( int i = 0, j = 0; i % 2 == 0 && j % 3 == 0; i++, j++ )

循环经常用在遍历集合的场景下，使用for语句来遍历集合时，常常需要将当前遍历到的集合元素取出来保存到一个临时变量中，这样做需要多写一些代码，因此C#提供了一个语法糖foreach语句来简化遍历集合的代码。如代码清单2 - 10中第9行的for语句与第15行的foreach语句是等价的。C#不允许在使用foreach语句内更新循环变量，否则会报告编译错误，如代码清单2 - 10中第18行。

代码清单2 - 10 for和foreach语句示例

|  |
| --- |
| 1 // 源码位置：第二章\ForDemo.cs  2 using System;  3  4 public class ForDemo  5 {  6 public static void Main()  7 {  8 int[] numbers = newint[] { 1, 2, 3, 4, 5, 6, 7 };  9 for (int i = 0; i<numbers.Length; i++)  10 {  11 int number = numbers[i];  12 Console.WriteLine(number);  13 }  14  15 foreach (int number in numbers)  16 {  17 // 不能在foreach语句中更新循环用的变量number  18 // number++;  19 Console.WriteLine(numbers);  20 }  21 }  22} |

注意： foreach语句也可以遍历枚举类型的所有值，其通过Enum.GetValues()方法获取包含枚举所有值的数组来遍历。

### 跳转语句

最基本的跳转语句是goto语句，其将当前程序跳转到指定标签的代码上继续执行，要跳转的标签必须和goto语句在相同的作用范围内，而且多个goto语句可以跳转到相同的代码标签上，实际上所有的条件判断、循环等语句都可以使用goto语句实现。goto语句可以跳出当前代码块，但不能跳转进一个代码块，这个限制是为了避免goto语句跳过代码块中的初始化代码，在C++等其他编程语言实现中也遵循同样的规则。由于goto语句和要跳转的标签往往相距很远，这种跳转会极大的增加代码阅读的难度，因此在大部分编程技巧的书籍中，都不建议使用goto语句。

代码清单2 - 11 goto语句示例

|  |
| --- |
| if (number % 2 == 0)  goto Even;  Console.WriteLine("奇数");  goto End;  Even:  Console.WriteLine("偶数");  End: |

笔者建议仅在下列场景中使用goto语句：

* 在switch语句中使用goto跳转到case标签；
* 当if、循环语句有可能嵌套很深的情况下，可以使用goto语句改造代码减少语句嵌套。

**break**语句的作用是结束包含其的最里层的**switch**、**while**、**do**、**for**和**foreach**等语句，而**continue**语句的作用则是结束当前循环语句剩下未执行的语句，继续下一个循环，如代码清单2 - 12：

代码清单2 - 12 break和continue语句

|  |
| --- |
| int i = 0;  while (true)  {  Console.WriteLine(i);  i++;  if (i < 10)  continue;  else  break;  } |

## 异常

程序在执行过程中不可避免总是会碰到一些意外的情况，例如碰到用户输入的不合理的输入或者是打开了一个不存在的文件等等，需要向用户或者调用程序报告这些错误。在C语言等早期编程语言中，通常是采用错误码来报告错误，如代码清单2 - 13，这种做法有几个缺陷：

* 代码的核心业务逻辑与错误处理代码混在一起，导致代码更难阅读，如代码清单2 - 13的第2、3行，而且错误码的处理机制往往会导致程序员不得不使用goto语句来避免多层判断语句嵌套问题；
* 错误代码是一个数字，虽然在编程时可以通过将错误码定义为常量的方式增加代码的可读性，但是在调试程序时，只能看到数字，增加了调试程序排错的难度；
* 错误码容易被忽略，因为错误码只是一个函数的返回值，编程新手很容易忘记处理错误码，导致错误日积月累，真正展现错误的代码和引起错误的代码不在同一个地方，极大的增加了日后的代码排错难度。

代码清单2 - 13 C语言的错误码处理机制

|  |
| --- |
| 1 int errorCode = 0; // 通常0代表成功  2 FileInfo source = new FileInfo("code.cs");  3 if (errorCode == -1) goto Failed;  4 int length = (int)source.Length;  5 if (errorCode == -2) goto Failed;  6 char[] contents = newchar[length];  7 if (errorCode == -3) goto Failed;  8 // 处理完所有的可能错误，执行正常的业务逻辑，如打开文件  9  10 Failed:  11 // 如果有任何错误发生，执行一些清理工作，如关闭打开的文件 |

C#、Java等编程语言则采用异常的方式来报告错误，所有的异常类型均从Exception类继承，如图2 - 5，相对于错误码的形式，异常有以下优点：

* 异常的错误描述信息要比错误码更丰富也更容易理解，在调试期间，调试器对异常的支持也更完善。
* 异常无法被忽略，当异常触发时，程序会在触发异常的地方中止执行，并开始在程序调用栈上寻找能够捕捉和处理这个异常的代码块，如果找不到的话，停止整个程序的运行，在引发错误的第一现场停止程序执行便于程序员排错。

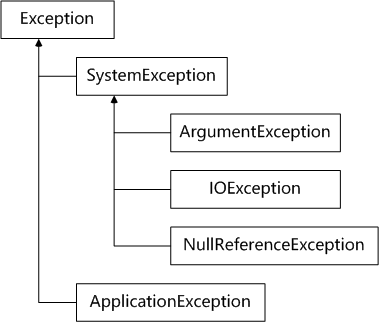


图2 - 9 异常继承结构例图

采用异常报告错误的方式，通常称为结构化错误处理，思路是将执行正常业务逻辑的代码和处理异常情况的代码分开，分别放在**try**和**catch**语句块中。这样执行业务逻辑的代码在**try**语句块中执行，其有可能碰到意外情况，使用**throw**关键字以抛出异常对象的方式报告错误。而处理这个错误的相关代码则放在**catch**语句块，**catch**的语法如下：

catch ( class-type identifier ) { … }

里面的class-type必须是System.Exception类或者继承自它的类型，identifier是保存捕捉到的异常的只读的局部变量，如果**catch**语句块中不需要用到异常信息，可以忽略它。在运行时，程序从try语句块顺序执行，当有异常抛出时，停止程序执行，并开始寻找能捕捉这个异常的**catch**块，如果在当前方法找不到匹配的**catch**块，则向上在调用这个方法的方法里递归寻找，直到寻找到一个匹配的**catch**块，则认为异常被成功捕捉，并在当前方法上继续执行。如果一直到Main方法都没有成功捕捉，则终止程序执行。

由于**try**语句块里面可以包含很多的语句，每条语句都有可能抛出一到多个不同的异常，所有**try**后面可以跟上多个**catch**语句块，每个**catch**语句块指明异常类型来捕捉特定种类的异常，如代码清单2 - 14中第43、47行分别尝试捕捉**OverflowException**和**DivideByZeroException**两个异常，这是因为**try**语句块中第31和33行用户输入的数字可能过大，超出了整形的取值范围，而第34行则可能发生除零错误，如果用户真的输入了不合理的值，则分别被这两个**catch**块捕捉并进入相应的语句块处理。在catch块里如果不需要用到被捕捉的异常，可以忽略保存异常的局部变量，如43和47行只指明了捕捉的异常类型，一般不建议在工作环境的编码中这样做，建议使用类似第51行的方式使用局部变量保存捕捉到的异常，并在日志里记录该异常的堆栈和消息等具体信息以便后续排错时调查。第51行也叫总异常捕捉块（general catch block），这是因为运行时按照代码排列顺序捕捉异常，最特定的异常放在最前面捕捉，其捕捉任何异常，所以放在最后使用。

代码清单2 - 14 在代码中捕捉异常示例

|  |
| --- |
| 1 // 源码位置：第二章\ExceptionDemo.cs  2 using System;  3  4 public class UserDefinedException : Exception  5 {  6 public UserDefinedException() : base() { }  7  8 public UserDefinedException(string message) : base(message) { }  9  10 public UserDefinedException(string message, Exception inner)  11 : base(message, inner) { }  12}  13  14public class ExceptionDemo  15{  16 static string ReadLine()  17 {  18 string str = Console.ReadLine();  19 if (string.IsNullOrWhiteSpace(str))  20 {  21 thrownew ArgumentNullException("至少需要输入一个数字！");  22 }  23 return str;  24 }  25  26 public static void Main()  27 {  28 try  29 {  30 Console.WriteLine("输入除数");  31 int i = int.Parse(ReadLine());  32 Console.WriteLine("输入被除数");  33 int j = int.Parse(ReadLine());  34 int k = i / j;  35 Console.WriteLine($"结果是：{k}");  36 }  37 catch (ArgumentException e)  38 {  39 // 再次抛出异常一般将原始异常保存在InnerException字段  40 thrownew UserDefinedException(  41 "系统错误，请参照InnerException获取具体的异常信息！", e);  42 }  43 catch (OverflowException)  44 {  45 Console.WriteLine("输入的数字太大！");  46 }  47 catch (DivideByZeroException)  48 {  49 Console.WriteLine("除零错误！");  50 }  51 catch (Exception e)  52 {  53 Console.WriteLine($"未知错误：{e.Message}，堆栈：\n{e.StackTrace}");  54 }  55 /\* 捕捉通用异常的语句也可以省略掉类型，如下面的catch块这样  56 catch  57 {  58 Console.WriteLine("未知错误！");  59 }  60 \*/  61 finally  62 {  63 Console.WriteLine("执行finally语句块中的代码");  64 }  65 }  66} |

在代码中碰到异常情况时，使用**throw**语句抛出异常通知这个错误，抛出的异常可以是.NET预定义的系统异常类型，也可以是用户通过继承Exception类型的自定义异常。代码清单2 - 14第21和第40行分别使用**throw**语句抛出系统定义异常和用户自定义异常。通常来讲，系统定义的异常在大部分场景下足够用了，抛出用户自定义异常的场景通常用在多组件之间通信，如Web服务等远程调用这种客户端组件调用服务器端组件的服务的场景，服务器端向客户端报告一个统一的异常，便于客户端捕捉异常并与本地代码的异常区分处理，这时新定义一个通用异常（比如取名ServerSideException）并将实际异常作为InnerException字段传递给客户端就显得很方便。代码清单2 - 14的第4行 – 12行演示了创建自定义异常的方法，其中3个构造函数是推荐的做法，第10行和第40行演示了将实际异常赋值给InnerException的方法，特别是第40行演示了在**catch**语句块里再次用**throw**语句抛出异常，一般称为重新抛出（re-throw）。

由于异常处理时会中止发生异常的代码后面的代码的执行，这往往会导致一些资源释放的问题，例如在执行文件读写操作时，在读写文件时发生异常，后面的释放文件资源的代码得不到执行，导致资源泄露，久而久之影响程序的性能和响应速度，所以C#提供了**finally**语句块，保证无论代码是否正常执行，都能执行**finally**块的代码，保证资源的释放操作，代码清单2 - 14的第61 – 64行演示了这个机制。

## 数组

数组是一个由相同类型的连续元素组成的集合，每个元素通过一个整数索引读写，索引值从0开始。定义数组时需要指明数组类型，数组的轶（rank），变量名称。代码清单 2- 4中第22行就是定义一个数组的例子，C#要求在使用变量之前必须对其进行初始化，第9行和第11行演示了初始化一维数组的方法，两行代码的作用完全一样，第11行是一个语法糖。第13行演示了初始化二维数组的方法，C#要求初始化二维数组时每个维度的长度要完全一致，如果每个维度的长度不一会导致编译错误。第17行演示如何定义并初始化一个数组类型的数组，即数组的元素也是一个数组，变量jag粗看起来像一个二维数组，但实际上是一个一维数组，其元素是一个数组，而且每个元素的长度可以不一样，虽然有点像二维数组，但跟二维数组定长不一样（类似网格），也被称为锯齿型数组。这一点在33行和35行打印数组的维度（rank）和长度的时候也可以看出来它们的区别，二维数组grid的rank是2，长度是行数 \* 列数的结果10，而锯齿数组jag的rank是1，长度是2 – 两个数组元素。两种数组的元素访问也有一些细微的差别，grid的元素索引方式是grid[1, 1]，表示访问第二行第二列的元素，而jag的元素索引方式是jag[1][1]，表示先访问第二行的数组，再获取这个数组的第二个元素。

常见的编程实践当中很少会采用代码清单 2- 4中硬编码初始化数组的方式，通常是采用类似26行的方式在运行过程中计算数组的大小，动态创建并在循环中动态初始化它。

代码清单 2- 4数组使用示例

|  |
| --- |
| 1 // 源码位置：第二章\ArrayDemo.cs 2 // 编译命令：csc ArrayDemo.cs 3 using System; 4  5 public class ArrayDemo 6 { 7     static void Main(string[] args) 8     { 9         int[] row = new int[9] { 10             1, 2, 3, 4, 5, 6, 7, 8, 9 }; 11         // int[] row = { 12         //     9, 8, 7, 6, 5, 4, 3, 2, 1 }; 13         int[,] grid = new int[,] { 14             { 1, 1, 1, 1, 1 }, // 第一行 15             { 2, 2, 2, 2, 2 }  // 第二行 16         }; 17         int[][] jag = new int[][] { 18             new int[] {1, 1, 1 }, 19             new int[] {2, 2 } 20         }; 21  22         int[] column; 23         if (args.Length > 0) 24         { 25             int len = int.Parse(args[0]); 26             column = new int[len]; 27             column[len - 1] = len; 28         } 29  30         Console.WriteLine( 31             $"row[0]:{row[0]},rank:{row.Rank},length:{row.Length}."); 32         Console.WriteLine( 33             $"grid[1,4]:{grid[1,4]},rank:{grid.Rank},length:{grid.Length}."); 34         Console.WriteLine( 35             $"jag[1][1]:{jag[1][1]},rank:{jag.Rank},length:{jag.Length}."); 36     } 37 } |

## 方法与参数

在程序设计时，通常都会将程序分解成多个功能模块，这种分而治之的策略既容易编写和理解代码，也尽量减少出错的概率，方法（Method）就是程序里的最基本的功能模块，在C#中，方法必须定义在类（Class）或者结构体（Struct）里，这点与C++不同，C++允许方法定义在类的外面。方法接受一系列的输入，执行一些计算，返回计算结果。定义方法通常需要指明方法名，参数列表 – 其所需的输入数据，方法体（Method Body）- 执行计算的代码块，返回值的类型 – 计算结果，以及可选的一些方法修饰符。可以用在方法上的修饰符如表2 - 7所示，这些修饰符都是可选的，而且方法上可以同时定义多个修饰符来描述方法的特征：

表2 - 7可用在方法上的修饰符

|  |  |
| --- | --- |
| 修饰符 | 说明 |
| 访问修饰符 | 包括 public、private、internal和protected四个关键字，代表四种方法的外部可见性，它们的区别将在本章后文说明。 |
| 实现修饰符 | 包括abstract、extern、new、override、sealed、virtual、partial等关键字，表示方法的实现方式，它们的区别将在本章后文说明。 |
| static | 表示方法是一个静态成员，区别与实例成员，它们的区别在本章后文说明。 |
| unsafe | 表示方法运行在一个不是代码安全的环境中，例如方法里面用到了涉及指针的操作，在编译带有unsafe修饰符的方法时，需要传入/unsafe编译开关。 |

方法名的取名要求与变量名相同，参数列表使用圆括号包括起来，一个方法可以有零到多个参数，多个参数间使用英文逗号间隔，每个参数由参数类型和参数名组成，方法也可以没有参数，表示方法没有输入数据。方法体是普通的语句块（block），遵循本章前面小节讲到的语句块的一般规则，因此方法体里也可以定义局部变量，方法的参数也可以当做局部变量使用。方法如果有输出数据的话，需要指明返回值的类型，这个类型可以是任意的C#类型，如果方法不返回任何值，返回类型是void。在C#中，编译器不通过方法名来判断方法是否重复，而是通过方法签名（Method Signature）作为判断标准。方法签名包含方法名称、参数列表中的参数类型、修饰符和数量，方法签名不包含返回类型和参数名。在声明方法的类中，方法签名必须是唯一的。

虽然C#有值类型和引用类型，但是默认情况下，调用方法时，值类型和引用类型都是按值传递的。即当被调用的方法接受的是值类型的参数，则系统会拷贝这个值类型的值并传递给方法；当接受的是引用类型的参数，系统会拷贝这个引用类型的引用（或者不准确的可以称之为指针）的值并传递给方法。按值传递的结果是即使在被调用的方法修改了参数的值，也不会影响调用者里原始传入的参数的值。如果需要在被调用的方法中修改传入的参数的值，则可以采用引用传值的方法，只要在传入的参数前面加上**ref**关键字就指明是引用传值了。代码清单 2- 4第13行和20行分别演示了按值传递和引用传值的区别。从图2 - 10的运行结果可以看到传值引用调用后，PassByValue方法中对变量i和s的修改都没有效果，但是对类型tc的成员变量Value的修改保留下来了，这是因为tc是一个类型，传递的是tc的引用，对于其成员变量的修改是直接作用于tc的。而PassByRef方法接受的是引用传值参数，方法内对参数的修改都是直接作用于传递过来的变量i和s的。

引用传递除了**ref**关键字之外，还支持使用**out**关键字进行输出传递，如代码清单 2- 4中第24和第25行，一般采用out输出传递的原因是方法需要同时返回多个结果，典型的如

bool Int32.TryParse(string s, out int value)

这样的方法，其返回值用来向调用者通知方法调用是否成功，而实际解析出来的值通过value参数回传。**out**与**ref**的区别在于后者要求传入的参数调用前需要初始化，而前者不需要。如果把第9行的初始化i变量的赋值语句去掉，采用类似第23行的定义方式，编译器会报告使用未初始化变量的错误。25行是C# 7.0开始加上的语法糖，可以在一行方法调用中定义新的变量k，免去了前面版本调用**out**参数需要两行代码的麻烦。

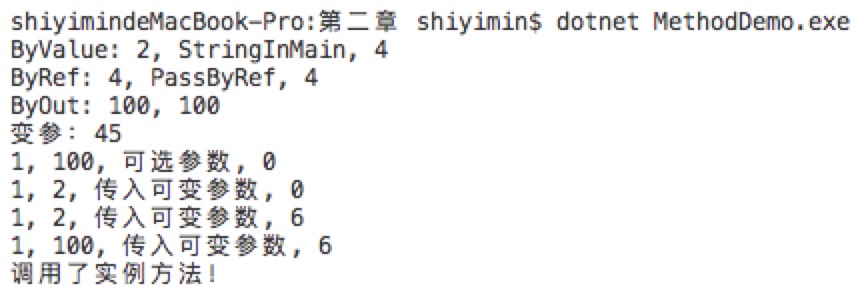


图2 - 10 不同类型方法的调用结果

C#还支持传递可变参数，这个特性在定义方法之前事先不知道参数个数的情形下很有用，典型的例子如我们常用的Console.WriteLine方法，使用**params**关键字，然后跟上可变参数类型的数组形式就可以定义一个可变参数。代码清单 2- 4中第62行定义的sum方法就是一个接受可变参数的方法，调用时采用第28行的方法传入任意个数的参数就能调用它，C#会将这些变长的参数列表整合成一个数组方便被调用的方法处理。

C# 4.0开始引入了可选参数的概念，即在定义方法时可以给几个参数设定默认值，调用者在调用方法时可以不传递这些参数，采用默认值即可。可选参数主要是为了支持调用Office自动化API等COM接口新增的语法糖，在一般的编程场景中，可选参数会导致后续接口升级的兼容性问题，建议读者尽量少用可选参数，特别是涉及到对外合作的公开接口，不建议使用可选参数。如果方法有多个可选参数时，调用者可以通过指定参数名称（named arguments）的方式给其中几个可选参数传值，代码清单 2- 4中第34行就是采用这种方式只给变量s传值，而optional参数依然采用默认值。第34行还演示了如何采用指定参数名称的方式给变参传值 – 将多个参数当成一个数组传递进去。代码清单 2- 4中第31和70行还演示了一个技巧，由于C#是支持Unicode的，可以用中文命名变量和方法。

代码清单 2- 4中定义的大部分方法都是静态（**static**）方法，不需要创建对象实例即可调用，第78行定义了一个实例方法，因此在38行调用它之前，需要创建一个实例才能调用。

代码清单 2- 5方法调用示例

|  |
| --- |
| 1 ﻿// 源码位置：第二章\MethodDemo.cs 2 // 编译命令：csc MethodDemo.cs 3 using System; 4  5 public class MethodDemo 6 { 7     static void Main(string[] args) 8     { 9         int i = 2; 10         string s = "StringInMain"; 11         TestClass tc = new TestClass(); 12         tc.Value = 2; 13         PassByValue(i, s, tc); 14         Console.WriteLine($"ByValue: {i}, {s}, {tc.Value}"); 15  16         i = 2; 17         s = "StringInMain"; 18         tc = new TestClass(); 19         tc.Value = 2; 20         PassByRef(ref i, ref s, ref tc); 21         Console.WriteLine($"ByRef: {i}, {s}, {tc.Value}"); 22  23         int j; 24         PassByOut(out j); 25         PassByOut(out int k); 26         Console.WriteLine($"ByOut: {j}, {k}"); 27  28         int ret = Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 0); 29         Console.WriteLine($"变参：{ret}"); 30  31         包含可选参数方法(1); 32         包含可选参数方法(1, 2, "传入可变参数"); 33         包含可选参数方法(1, 2, "传入可变参数", 1, 2, 3); 34         包含可选参数方法(1, s: "传入可变参数",  35             values: new int[] { 1, 2, 3 }); 36  37         MethodDemo md = new MethodDemo(); 38         md.InstanceMethod(); 39     } 40  41     private static void PassByValue( 42         int i, string s, TestClass tc) 43     { 44         i = i \* 2; 45         s = "PassByValue"; 46         tc.Value = tc.Value \* 2; 47     } 48  49     static void PassByRef( 50         ref int i, ref string s, ref TestClass tc) 51     { 52         i = i \* 2; 53         s = "PassByRef"; 54         tc.Value = tc.Value \* 2; 55     } 56  57     static void PassByOut(out int i) 58     { 59         i = 100; 60     } 61  62     static int Sum(params int[] values) 63     { 64         int result = 0; 65         for (var i = 0; i < values.Length; ++i) 66             result += values[i]; 67         return result; 68     } 69  70     static void 包含可选参数方法( 71         int required, int optional = 100,  72         string s = "可选参数", params int[] values) 73     { 74         Console.WriteLine( 75             $"{required}, {optional}, {s}, {Sum(values)}");  76     } 77  78     void InstanceMethod() 79     { 80         Console.WriteLine("调用了实例方法！"); 81     } 82 } 83  84 class TestClass 85 { 86     public int Value; 87 } |

前面的示例代码中经常见到的Main方法也是一个方法，Main方法是一个程序的入口方法-即操作系统从这个方法开始顺序执行程序代码，可执行的命令行（CUI）和桌面（GUI）程序都必须要有Main方法，与普通方法一样，Main方法也可以定义参数列表和返回值类型，但操作系统只能识别如下特征的Main方法作为入口方法：

* Main方法可以在类型（**class**）或者结构体（**struct**）中定义，但必须是静态（**static**）方法，但包含Main方法的类型或者结构体不必是静态的。
* Main方法可以是私有（**private**）方法，即操作系统可以忽略Main方法上的访问限制修饰符。
* Main方法的返回值可以是**void**或**int**类型，通常命令行（CUI）程序会采用int的返回类型，用来向操作系统汇报执行结果，如代码清单 2- 4中第16行和第27行针对执行成功和失败等不同情况分别返回了不同的状态值，这么做的原因是用在批处理程序中通过判断进程的返回值 – 即Main方法的返回值来选择性的执行后续的程序。返回0通常说明程序成功执行，失败采用非0值表示，这是因为成功只有一种情况，但是失败的原因有很多种，需要采用不同的状态码描述。代码清单 2- 5则演示了在bash脚本里执行程序并根据返回值来选择性的执行后面的脚本代码。Bash脚本多用在Mac、Linux和Unix等操作系统上，在Windows系统上也可以使用批处理（batch）脚本或者PowerShell脚本实现类似的效果，这方面的知识留给感兴趣的读者自己实践。虽然代码清单 2- 4和代码清单 2- 5演示的是命令行程序的情形，图形化程序一样可以采用类似的机制在用户关闭程序时将执行状态返回给操作系统。在C# 7.1之后，Main方法的返回值也可以是Task或者Task<int>，如果是后两个返回类型的话，可以加上async修饰符，Task和async修饰符将在后面的章节详细叙述。
* Main方法可以接受一个字符串数组（string[]）类型的参数，通常这些参数是在命令行中由操作系统传入的，程序通过解析字符串数组的每个元素来设置进程后续执行所需的数据。
* C#允许源码中包含多个Main方法，但是每个Main方法必须包含在不同的类型或者结构体中，在编译时，可以通过**csc**编译器的**/main**这个参数指定最终的入口方法，代码清单 2- 4的第2行的注释演示了/main参数的使用方法。

代码清单 2- 6Main方法的使用示例

|  |
| --- |
| 1 // 源码位置：第二章\MainDemo.cs 2 // 编译命令：csc /main:MainDemo1 MainDemo.cs 3 using System; 4 5 public class MainDemo1 6 { 7     public const int ARGUMENT\_ERROR = 1; 8     public const int I\_HATE\_ODD = 2; 9 10     public static int Main(string[] args) 11     { 12         if (args.Length != 2) 13         { 14             Console.WriteLine( 15                 "Usage: MainDemo <arg1> <arg2>"); 16             return ARGUMENT\_ERROR; 17         } 18 19         Console.WriteLine( 20             $"收到参数：【{args[0]}】与【{args[1]}】"); 21         var left = int.Parse(args[0]); 22         var right = int.Parse(args[1]); 23         var result = left + right; 24         if (result % 2 != 0) 25             return I\_HATE\_ODD; 26         else 27             return 0; 28     } 29 } 30 31 public class MainDemo2 32 { 33     public static void Main() 34     { 35         var rnd = new Random(); 36         var value = rnd.Next(); 37         if (value % 2 != 0) 38             throw new Exception("奇数触发异常！"); 39         else 40             Console.WriteLine("偶数正常执行！"); 41     } 42 } |

代码清单 2- 7 Bash脚本里执行dotnet程序并判断返回值

|  |
| --- |
| 1 #!/bin/bash  2  3 # dotnet MainDemo.exe  4 dotnet MainDemo.exe 1 2  5 # dotnet MainDemo.exe 3 3  6  7 case"$?"in  8 "1") echo "执行程序的参数错误！";;  9 "2") echo "结果是奇数！";;  10 "0") echo "执行成功！";;  11 esac |

## 方法重载

C#允许类型里有多个重名方法，但是方法签名 – 即参数数量或类型需要不同，称之为方法重载（method overloading）。方法重载一般用在需要不同类型输入数据，但是都执行相同的操作的场景下，如代码清单 2- 10中为相加多个数字这个操作，针对不同类型和数量的输入定义了多个重载方法，它们的名字都是“Add”，但要么是参数数量不同，要么就是参数类型不同，这样编译器就能将这些“Add”方法根据方法签名的不同而区分开来。这里值得注意的是第1行和第10行的两个“Add”重载方法，由于int可以隐式转换为double类型，所以为了采用double类型的重载版本，笔者在第15行特意将“2”采用double形式传入。另外方法的返回类型不作为方法签名看待，因此如果两个重名的方法，参数数量、顺序和类型完全相同，只是返回类型不同的话，编译器是会当作同名参数并报告编译错误的。

代码清单 2- 8方法重载示例

|  |
| --- |
| 1 static int Add(int x, int y) { return x + y; } 2  3 static int Add(int x, int y, int z) { return x + y + z; } 4  5 static double Add(string x, string y)  6 { 7     return double.Parse(x) + double.Parse(y); 8 } 9  10 static double Add(double x, double y) { return x + y; } 11  12 Console.WriteLine(Add(1, 2)); 13 Console.WriteLine(Add(1, 2, 3)); 14 Console.WriteLine(Add("1", "2")); 15 Console.WriteLine(Add(1, 2d)); |

## 操作符重载

跟C++类似，C#也提供了操作符重载的语法，为类型提供更直观的运算方式，这个语法可以用在需要大量科学计算的代码中，重载操作符跟定义方法类似，只是方法名以“**operator**”关键字开始，后面跟上要重载的操作符[[3]](#footnote-3)。如代码清单2 - 17的第29行和34行都重载了“+”操作符，但接受的操作数可以是不同类型，这样重载可以给操作符引入更多的语义。除了支持重载一般的预算操作符以外，C#还支持重载转换操作符，可以通过“**explicit**”和“**implicit**”两个关键字指明转换操作是强制（显式）还是隐式转换，如代码清单2 - 17的39行和44行，分别演示了两种转换的重载方式，两种转换的使用方式跟普通的原生类型想似，如第14行使用了隐式转换操作符将Number类型隐式转换成double类型后再计算表达式，而16行则使用了强制转换操作符。

代码清单2 - 15 操作符重载示例

|  |
| --- |
| 1 // 源码位置：第二章\OperatorOverloadingDemo.cs  2 // 编译命令：csc OperatorOverloadingDemo.cs  3 using System;  4  5 public class OperatorOverloadingDemo  6 {  7 static void Main()  8 {  9 Number a = new Number(1), b = new Number(2);  10 Number sum = a + b;  11 Console.WriteLine($"{sum}");  12 sum = sum + 3;  13 Console.WriteLine($"{sum}");  14 double dsum = a + b + 3.4d;  15 Console.WriteLine($"{dsum}");  16 Console.WriteLine($"{(short)sum}");  17 }  18 }  19  20 public class Number  21 {  22 private int Value { get; set; }  23  24 public Number(int v)  25 {  26 Value = v;  27 }  28  29 public static Number operator+(Number left, Number right)  30 {  31 return new Number(left.Value + right.Value);  32 }  33  34 public static Number operator+(Number left, int right)  35 {  36 return new Number(left.Value + right);  37 }  38  39 public static implicit operator double(Number num)  40 {  41 return (double)num.Value;  42 }  43  44 public static explicit operator short(Number num)  45 {  46 return (short)num.Value;  47 }  48  49 public override string ToString() { return Value.ToString(); }  50 } |

## 预处理指令

跟C/C++语言类似，C#也定义了预处理指令，程序员通过在源码中使用这些预处理指令来修改编译器在编译期的行为。C#中的预处理指令都以“**#**”开始，预处理指令不是语句，后面不需要分号“;”结尾，表2 - 8列出了C#支持的预处理指令和相关说明。

表2 - 8C#支持的预处理指令集合

|  |  |
| --- | --- |
| 预处理指令 | 说明 |
| #define | 类似C/C++语言，定义一个宏变量，不推荐使用，推荐通过csc的/d开关在编译期动态定义，参考代码清单 2- 8的第2行注释。 |
| #undef | 取消一个宏变量定义。 |
| #if | 测试一个宏是否定义，或者宏的值是否满足某些条件来动态开启包括起来的代码，如代码清单 2- 8的第14行到第15行的代码只有定义了DEBUG宏的情况下才启用。  在使用#if时，可以使用“==”、“!=”、“&&”和“||”等逻辑判断操作符测试宏的值。 |
| #else | 与#if指令联合使用。 |
| #elif | 与#elif指令联合使用，作用类似判断语句中的else if。 |
| #endif | 指明一块宏测试预处理指令的结束位置。 |
| #pragma | 给编译器一些指令，支持两种指令：   * checksum，这个指令用在引入外部源文件时指明文件的校验码，只在Visual Studio调试ASP.NET页面时使用，平时编程接触不到。 * warning，临时禁用或者启用编译警告。如代码清单 2- 8的第9行使用warning disable临时禁用了编号为CS0219的编译警告，而第11行则使用warning restore恢复了CS0219警告。这个警告编号可以在csc编译器输出的警告信息中获取，如：“warning **CS0219**: The variable 'i' is assigned but its value is never used”。使用时可以省略警告编号前面的CS前缀。 |
| #line | 一般用在生成C#代码的场景中，使用该指令可以指明生成的C#代码的原始源码的位置，如ASP.NET里的aspx页面实际上会先被转换成C#源码再由csc编译，在生成的C#源码中嵌入#line指明原始aspx代码的位置，可以在后续的调试过程中提供aspx级别的调试体验。 |
| #error | 在代码中强制编译器产生一个编译错误。 |
| #warning | 在代码中强制编译器产生一个编译警告。 |
| #region | 用在IDE中，划分一段代码范围，在编辑代码时，可以将这块代码折叠起来。 |
| #endregion | 指明#region的结束位置。 |

根据预处理指令来动态启用不同的代码有很多实际应用，如在开发阶段可以通过#if预处理指令包含一些对开发排错，甚至对自动化测试提供支持的代码，但是在生产环境中通过使用不同的编译器开关移除掉这些冗余代码来提供程序的整体性能。

C#中有两种根据预处理宏启用不同代码的方法，一种是使用#if指令根据宏定义来动态启用，一种是采用Conditional特性，特性将在后文详细说明。笔者建议读者尽可能使用后者，因为#if的方案中，如果不符合测试条件，其所包括的代码会全部删除，如果在编程时不小心将一些被后续代码依赖的代码也包含在#if当中，轻则导致代码编译失败，重则在运行时才发现错误。例如将代码清单 2- 8中的第15行和20行的注释取消，并在编译时不定义DEBUG宏，则会造成编译失败。而后者Conditional的方案，通过在Conditional特性里指定需要满足条件的宏DEBUG – 参考23行，如果编译时定义了DEBUG宏，则调用这个方法的19行能够工作；而DEBUG没有定义的话，编译器会自动移除第19行的调用，进而避免了编译失败。

代码清单 2- 9预处理指令使用示例

|  |
| --- |
| 1 // 源码位置：第二章\PreprocessorDemo.cs 2 // 编译命令：csc /d:DEBUG PreprocessorDemo.cs 3 using System; 4  5 public class PreprocessorDemo 6 { 7     static void Main(string[] args) 8     { 9  #pragma warning disable 219 10         int i = 0; 11 #pragma warning restore CS0219 12  13 #if DEBUG 14         Console.WriteLine("定义了DEBUG宏"); 15         // int j = 1; 16 #else 17         Console.WriteLine("没有宏"); 18 #endif 19         OnlyDebugDefined(); 20         // Console.WriteLine(j + j); 21     } 22  23     [System.Diagnostics.Conditional("DEBUG")] 24     static void OnlyDebugDefined() 25     { 26         Console.WriteLine("只有在DEBUG定义了的情况下才有"); 27     } 28 } |

# 结构

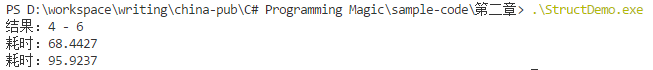
结构（**struct**）与类（**class**）非常相似，结构里可以定义构造函数，也可以定义成员变量和方法，然而结构是值类型，因此结构变量保存是其成员变量的实际数据 – 而类保存的是成员变量的引用，当将一个结构变量赋值给另外一个结构时，实际上是将前面结构的变量数据拷贝到后面结构中去。所有值类型都是密封的（**sealed**），结构也不例外，因此不能从结构里继承新的子结构，但是结构可以实现接口。由于接口是引用类型，将结构传递到要求的接口的方法或者变量时，就需要执行一次装箱（boxing）操作。

虽然定义结构的语法与类定义的语法很相似，但它们之间还是有如下区别：

* 定义结构时，不能初始化其成员变量，除非成员变量的修饰符是**const**或者**static**；
* 结构不能定义默认构造函数 – 原因参考后文代码清单2 - 15的说明，也不能定义一个终结器（**finalizer**）；
* 结构在栈上分配内存，或者直接嵌入在包含它的类型的内存中，而类在堆上分配内存；
* 结构可以不使用**new**操作符初始化，而类则必须用**new**初始化。

代码清单2 - 15演示了结构的一些基本用法，第9行、18 - 19行和27行分别演示了三种初始化结构的方法；第11行也演示了结构定义方法并被外部模块调用的例子；初始化结构体时，直接初始化变量要比使用带参构造函数能获得更好的性能，如第16行和第25行的两个循环分别使用这两种初始化方法创建结构，运行的结果如图2 - 11，可以看到使用构造函数的方法要比直接初始化变量的代码慢了不少，这也是C#编译器禁止在结构体中定义无参构造函数的原因之一。

图2 - 11 结构体初始化不同方式性能对比结果



代码清单2 - 16 结构使用示例

|  |
| --- |
| 1 // 源码位置：第二章\StructDemo.cs  2 // 编译命令：csc StructDemo.cs  3 using System;  4  5 public class StructDemo  6 {  7 static void Main()  8 {  9 Point point1 = new Point { X = 1, Y = 2 },  10 point2 = new Point { X = 3, Y = 4};  11 point1.Add(point2);  12 Console.WriteLine($"结果：{point1.X} - {point1.Y}");  13  14 Point[] points = new Point[10000000];  15 DateTime begin = DateTime.Now;  16 for (var i = 0; i<points.Length; ++i)  17 {  18 points[i].X = i;  19 points[i].Y = i;  20 }  21 var peroid = DateTime.Now - begin;  22 Console.WriteLine($"耗时：{peroid.TotalMilliseconds}");  23  24 begin = DateTime.Now;  25 for (var i = 0; i<points.Length; ++i)  26 {  27 points[i] = new Point(i, i);  28 }  29 peroid = DateTime.Now - begin;  30 Console.WriteLine($"耗时：{peroid.TotalMilliseconds}");  31 }  32 }  33  34 public struct Point  35 {  36 // public Point() {}  37  38 public Point(int x, int y)  39 {  40 X = x;  41 Y = y;  42 }  43  44 public int X;  45  46 public int Y;  47  48 public void Add(Point other)  49 {  50 X += other.X;  51 Y += other.Y;  52 }  53 } |

由于结构的使用方法与类非常相似，结构的使用场景比较有限，下面是笔者自己调研和体验的结构体的推荐使用场景：

1. 用在与非托管代码交互方面，由于C/C++的方法大量使用结构，而且结构的成员的内存布局不少都有特殊要求，在C#中可以通过结构来在托管代码层面构造与这些内存布局相同的结构。
2. 如果结构很小，例如大小不超过64个字节，可以使用结构，因为64位机器上引用（或者说指针）的大小也是64个字节，没有必要为了这么小的数据结构定义一个引用类型，浪费内存。
3. 如果一个数据结构会长期呆在内存，而且会有大量的实例保存在内存中，使用结构会比类节省垃圾回收的次数，从而提高性能[[4]](#footnote-4)，而且结构（**struct**）数组相对与对象（**object**）数组有更好的位置紧凑性（locality），现代CPU通常是将内存的一段数据批量加载到缓存中，这种硬件特性对频繁的读写结构数组的性能有很大的帮助。虽然CPU同样会批量加载对象数组中的一段数据进入缓存，但这些数据只是对象的引用，在读写时CPU还是需要根据引用去内存读写实际的数据，无法利用到这个硬件的优化。

在C# 2.0之后，添加了一个可空类型（**nullable types**）的语法糖，它是一个泛型，主要的作用是在值类型的基础上增加对**null**值的支持。在很多业务场景里，需要在值类型上添加null值的应用，如数据库中的表定义中支持可空列，那么在C#中如何使用值类型表示这些列就有很大的问题，以int类型为例，null和0是两个值，不能混为一谈。因此在此之前，要么采用引用类型，要么单独定义一个结构体同时保存值类型和是否有空值，使用引用类型的问题是没有值类型的性能上的好处，而结构体的问题是使用起来不是很直观。可空类型采用第二种方式 – 即结构体的方式，但是在C#语法上对其增加了额外的支持。

C#中定义一个可空类型有两种方法，一种是直接在值类型后面加上英文“?”，如int?就是值类型int的可空类型，另外一种方式是使用Nullable<T>结构体创建，如Nullable<int>也是int的可空类型。可空类型有两个成员变量，布尔型的HasValue表明当前变量中是否有值，而泛型Value则在HasValue为true的情况下，返回实际的数据，否则在HasValue为false的情况下，访问Value会导致系统抛出InvalidOperationException异常。

由于可空类型实际上是一个结构体，即值类型，因此其与引用类型相互赋值时，也是需要遵循装箱和拆箱的操作的，但其装箱规则略有不同：

* 当HasValue为false时，装箱操作获得是null引用；
* 当HasValue为true时，则将Value的值装箱，而不是将Nullable<T>变量装箱。

代码清单2 - 16演示了可空类型的一些使用方法，如第9行将null值直接赋值给可控类型，第10行演示了创建可空类型数组的语法，第14和15行很有意思，第12行的x1和x2的使用是一个结构体的用法，但取消14和15行的注释再次编译运行的话，程序会抛出NullReference异常，这是因为第12行是将x1和x2当做正常的结构体使用，HasValue是Nullable<T>结构体的一个成员变量；而14和15行，GetType()是System.Object类型的成员，在调用前，需要将x1和x2装箱，得到的是null值，导致了异常的抛出，同理在22行调用GetType方法，打印的是装箱Value值后的类型，即System.Int32。同时也代码清单2 - 16中可以看到C#中，Nullable<T>结构和T?的语义是一样的，两者可以混用。针对可空类型，C# 6.0开始甚至还提供了一个操作符语法糖 – “**??**”，它是三目预算符的一个简化，如16行中，如果x1的值是null，则使用default(int) – 即int型的默认值0，否则使用x1里的Value值赋值给i。即使有前面这些语法优化，但可空类型不能隐式转换成其对应的值类型，只能使用强制转换，如将23行的强制转换去掉，会导致编译错误。

代码清单2 - 17 可空类型使用示例

|  |
| --- |
| 1 // 源码位置：第二章\NullableTypeDemo.cs  2 // 编译命令：csc NullableTypeDemo.cs  3 using System;  4  5 public class NullableTypeDemo  6 {  7 static void Main()  8 {  9 int? x1 = null;  10 int?[] arr = new int?[10];  11 Nullable<int> x2 = null;  12 Console.WriteLine($"{x1.HasValue} - {x2.HasValue}");  13 Console.WriteLine($"{x1 == null} - {x2 == null}");  14 // Console.WriteLine($"{x1.GetType()}");  15 // Console.WriteLine($"{x2.GetType()}");  16 int i = x1 ?? default(int);  17  18 x1 = x2 = arr[0] = 10;  19 var x3 = x1 + x2;  20 Console.WriteLine($"{typeof(int?)} - {typeof(Nullable<int>)}");  21 Console.WriteLine(  22 $"{x1.GetType()} - {x2.GetType()} - {x3.GetType()}");  23 i = (int)x1;  24 Console.WriteLine($"{i}");  25 }  26 } |

# 面向对象编程

C语言这些面向过程的编程语言中，常常包含很多数据（变量）和方法，每个方法都可以访问一些变量，方法和变量的耦合度非常高。这种编程模型在小规模的程序里还好维护，随着程序规模的不断扩大，数据和方法越来越多，依赖某个数据的方法常常分散在很多.c源码文件中，修改这个数据往往变得非常困难 – 一不小心就导致依赖这个数据的某个方法无法正常工作。而面向对象编程的方式则是将数据和需要读写它的方法都封装到同一个类里，不仅更适合日常的思考方式，而且将耦合性高的数据和方法放在一起，也更容易维护。

面向对象编程最核心的概念就是类型（class），类型其实是一个分类的过程，是符合人们日常认知的一个过程。例如每辆车都有共通的行为：启动、加油门、刹车等；也都有共通的属性：四个轮子、方向盘等。当将一个物体称之为“车”的时候，其实指代的就是具备这些共通属性和行为的物体。类似的，我们在编程中将这些共通的属性和行为打包归类，避免有些代码反复出现，造成维护和阅读困难。“车”这个名词指代（归类）了具备共同属性和行为的同一类物品，在编程中，使用对象（object）或者实例（instance）来指代一辆具体的车。一个对象有三种特征组成：

* 身份（identity）：身份是将相同类型的一个具体的对象与其他对象区别开来的具体特征，在编程中，通常是对象的地址，如指针或者引用等等；
* 行为（behavior）：行为是让对象变得可以使用的基本特征，如车只有开动才有用，行为也是外部模块与对象交互的主要特征，在编程中，通常是对象的实例方法；
* 状态（state）：对象的行为改变通常意味着自身的状态的变化，状态可以分为外部可访问状态和内部维护状态，通过将状态的修改封装在类里，达到将对象与使用它的外部模块解耦，在编程中，状态由对象的实例变量表示。

## 定义和使用类

在C#中定义类的关键字是**class**，定义类型需要如下信息：

* 指明类型的名字，类名的命名规则与变量名相同，如代码清单 2- 9中分别在第6行和第51行定义了两个类型SavingAccount和ClassDemo。
* 通过访问修饰符指明类的外部可见性，最常用的访问修饰符是**public**和**private**，其它修饰符在后文说明，代码清单 2- 9中SavingAccount类型是public的，说明外部模块可以直接使用这个类型，而ClassDemo没有指明访问修饰符，则采用类型默认的访问修饰符internal。
* 定义类型的静态（static）和实例变量，表示类型自身和实例对象的状态。SavingAccount类型定义了一个静态变量InterestRate – 使用static关键字修饰，和几个实例变量Amount，InitialAmount和BeginDate。
* 定义类型的静态和实例方法，表示类型自身和实例对象的行为。如SavingAccount中的WithDraw方法是一个实例方法，而ClassDemo中的Main方法则是静态方法。

代码清单 2- 10类型使用示例

|  |
| --- |
| 1 // 源码位置：第二章\ClassDemo.cs 2 // 编译命令：csc ClassDemo.cs 3 using System; 4  5 public class SavingAccount 6 { 7     public static decimal InterestRate { get; private set; } = 0.03m; 8  9     public string AccountName { get; private set; } 10  11     public decimal Amount 12     { 13          get 14          { 15              var now = DateTime.Now; 16              int days = (int)(now - BeginDate).TotalDays; 17              return InitialAmount\* (1 + InterestRate / 365 \* days); 18          } 19     } 20  21     private decimal \_initialAmount; 22     public decimal InitialAmount 23     { 24          get { return \_initialAmount; } 25          private set  26          { 27              if (value <= 0) 28                 throw new InvalidOperationException("只接受正数！"); 29              \_initialAmount = value;  30         } 31     } 32  33     public DateTime BeginDate { get; private set; } 34  35     public SavingAccount(string name, decimal amount, DateTime? begin = null) 36     { 37         AccountName = name; 38         InitialAmount = amount; 39         BeginDate = begin ?? DateTime.Now; 40     } 41  42     public void WithDraw(decimal amount) 43     { 44         if (amount > this.Amount)  45             throw new ArgumentException("余额不足"); 46         InitialAmount = Amount - amount; 47         BeginDate = DateTime.Now; 48     } 49 } 50  51 class ClassDemo 52 { 53     static void Main() 54     { 55         var account1 = new SavingAccount("张三", 10000); 56         var account2 = 57             new SavingAccount("李四", 10000, new DateTime(2018, 1, 1)); 58  59         Console.WriteLine($"利率是：{SavingAccount.InterestRate \* 100}%"); 60         Console.WriteLine($"{account1.AccountName}的初始余额是" + 61             $"{account1.InitialAmount}，当前计息余额是{account1.Amount}"); 62         Console.WriteLine($"{account2.AccountName}的初始余额是" + 63             $"{account2.InitialAmount}，当前计息余额是{account2.Amount}"); 64  65         account1.WithDraw(9000); 66         account2.WithDraw(9000); 67  68         Console.WriteLine("取款后！"); 69         Console.WriteLine($"{account1.AccountName}的初始余额是" + 70             $"{account1.InitialAmount}，当前计息余额是{account1.Amount}"); 71         Console.WriteLine($"{account2.AccountName}的初始余额是" + 72             $"{account2.InitialAmount}，当前计息余额是{account2.Amount}"); 73     } 74 } |

代码清单 2- 9演示了几个典型的面向对象封装的概念，首先将储蓄账户封装成SavingAccount类型，代表一类储蓄账户，每一个账户则是这个类的实例，如第55行和56行两个实例。不同账户的账户名、初始余额和创建日期一般是不同的，因此将这些状态封装成实例变量允许状态随着实例不同而不同。出于演示目的，假设所有储蓄账户的利率都是一样的，将利率封装为类型的静态变量体现这个概念，在计算每个账户的利息时，只要统一引用这个静态变量即可。由于实例变量封装的是不同实例的状态，因此在使用时需要通过对象实例来引用到他们，如第60行访问account1的不同实例变量的做法。而静态变量封装的状态不会因为实例的不同而改变，在引用时需要通过类型本身，如第59行的读取利率的做法。

实例方法和静态方法的差别也是类似的，由于实例方法会读写对象本身的状态，编译器默认会在实例方法里隐含一个参数this，指代实例本身，由于是隐含的，在读写实例变量或者调用实例方法时，通常可以忽略它，如44行和46行都可以读取实例本身的Amount变量。C#中有两种方式表示对象的状态，字段（field）和属性（property），其中字段与其它如C++和Java等面向对象编程语言的成员变量相似，内外模块根据访问修饰符设置可以直接读写字段的数值，如21行的\_initialAmount。而属性更像是其它编程语言中的读取/写入（getter/setter）方法的语法糖，程序员可以分别在读取写入字段时包含控制性的代码，如27行在改\_initialAmount时做的数值检查的代码，在写入方法里，编译器提供了一个隐含的变量**value[[5]](#footnote-5)**，用来保存外部修改属性值的数值。属性的读取和写入方法可以有不同的访问控制，如AccountName、InitialAmount和BeginDate这些属性读取方法是公开的，允许内外模块直接访问，然而写入方法是私有（private）的，只允许类型本身的实例方法修改，这样就提供了良好的数据封装性。C# 3.0之后，提供了一个名为自动实现属性（auto implemented properties）的语法糖优化，如第33行的BeginDate属性，对读取和写入方法采用不同的访问修饰符控制，而不需要提供类型InitialAmount属性的方法代码。第11行还演示了属性的另外一个用法 – 计算属性，即属性并没有一个实际的字段与之对应，每次访问都是通过内置的读取方法计算得出。C# 6.0更进一步，增加了自动实现属性赋值的语法糖，如第7行的InterestRate就是采用这种方法初始化的。

## 继承

面向对象编程除了将类型的状态的更新和与之高度耦合的行为代码封装到类型里以外，还提供继承的功能增进了代码复用。在使用一个代码库时，通过继承在一些类型的基础上添加一些自定义的功能。如在编程中经常会用到集合的概念，与其在每个集合类型（如任务列表类）都重复实现集合的方法，采用继承的做法是先定义集合的基类（base class），将所有集合类型都用到的状态和方法抽象泛化出来，其它集合类型作为子类（derived class）在这些泛化的状态和方法的基础上，增加业务特定的代码。由于子类会拥有基类除了private访问修饰符以外的所有成员，所以在基类上添加任何一个新的成员，会自动体现到子类中，极大的提高了代码维护的效率。C#与C++语言不同，C#不允许多重继承，即一个类型只能继承自一个基类，Object类型是所有C#类型的根本基类，继承的语法如下，子类后面使用英文冒号指明其基类即定义了继承关系：

class SubClass : BaseClass

由于子类会经常访问父类的一些成员或者方法，而这些成员或者方法采用**public**修饰符可能并不是很合适，就好比父母给孩子零花钱的小盒子，孩子可以去随意存取，但是外人不能这样做。而定义为private又阻止了其他类 – 包括子类访问，所以**protected**修饰符就是用来保护这种成员和方法的。

|  |  |
| --- | --- |
| 修饰符 | 说明 |
| public | 这种类型的成员是公开的，任何代码都可以访问它。 |
| private | 成员是私有的，只有本类或结构体中的代码可以访问它。 |
| protected | 成员只能被本类或者继承的子类访问。 |
| internal | 成员只能对定义在同一个装配件（assembly）中的代码公开，但是不是同一个装配件的代码不能访问它。 |
| protected internal | 是protected关键字和internal关键字两者的结合，即只对同一个装配件的代码公开，而其他装配件里只有从它继承的子类才能访问，这种情况非常少见。 |
| private protected | 只能被类型自己，和同一个装配件里从它继承的子类访问，这种情况非常少见。 |

## 虚方法和方法重写

继承是定义在类型层面的更新，继承通过将类型泛化，一方面便于代码的复用，一方面也便于外部代码操作类型。比如用户资金分别投资于活期存款，定期存款、基金和股票等资产上，这些资产都有一样的状态，如总额，购买时间等，也有相同的方法，如投资和赎回等。每个资产本身计算投资回报和投资风险的算法不同，针对每个资产设计一个类型是很合适的办法。在为用户呈现资产总额时，需要遍历这些资产计算累加资产总值，这个时候与其一个一个类型单独判断处理，不如为这些资产定义一个基类，并将用户的资产保存在基类数组里，计算资产总额时，遍历数组里的所有资产，计算得出总投资回报。由于每个资产的投资回报计算方式不同，不能使用统一的方法来处理，但在遍历资产数组计算时，有需要基类定义有计算投资回报率的方法供外部模块调用，这种场景下就可以采用虚函数处理。

代码清单 2- 10就演示了这种编程技巧，首先定义一个基类Account来泛化用户所有的投资账户的共同特征，而更明确的储蓄账户和股票账户分别用SavingAccount和StockAccount两个子类表示，而由于获取投资回报的计算方法根据账户不同而不同，所以将Account的当前余额Amount属性使用**virtual**关键字修饰作为一个虚方法（属性也可以这样处理），并提供一个默认的实现 – 即不考虑投资回报率，只返回初始余额。接下来在33行和52行分别给SavingAccount和StockAccount两个类型使用**override**关键字将虚方法重写，实现不同的投资回报率的计算公式。这样，在71行就可以使用一个Account[]类型的资产数组来统一表述用户的所有投资账户，在计算用户的总收益时，通过类似76行的循环就能得到每个账户的收益。这种对于调用者来看是统一处理的方式，而实际的代码确是不同的实现的做法，在面向对象编程中被称为多态。

子类和外部类型不同，外部类型只能访问类型的public或者internal修饰的成员，而子类除此之外，还可以读写基类中由protected关键字修饰的成员，这一点在第49行也演示了这个概念，通过这个关键字，基类可以允许子类读写有限的一些成员。如果不想其它模块从类型中派生新的类型，则可以使用**sealed**关键字来限制继承，C#编译器保证没有类型可以从**sealed**类型中继承。如代码清单 2- 10中的SavingAccount就不允许派生子类，而StockAccount则可以。**sealed**关键字的使用场景不多，一般出于安全性和性能优化的考虑才会考虑封闭类型的继承能力：

* 如果类型是用在插件化架构中，为了防止恶意继承类型，从而导致框架代码意外加载恶意代码，使用**sealed**可以防止这种情况发生。
* 另一方面，当一个类型采用了sealed关键字修饰，CLR在加载类型的时候不会再继续搜索其子类，会提升一些加载速度。

有时子类的一些字段和方法可能会跟基类的成员重名，默认情况下，在使用时，子类成员会覆盖基类的成员，编译器同时会打印一个编译警告指出这个问题。如果子类改名的风险较大，这个时候可以使用**new**关键字在子类中显式禁用这个编译器警告。这种情况下，当对象实例是子类类型时，调用重名成员就直接调用到子类的成员，而如果要调用基类的成员，则需要将子类实例显式转换成基类类型，第62行在StockAccount中定义了重名成员InitialAmount，在81行中，由于stock变量是StockAccount类型，所以调用的62行定义的InitialAmount成员，而82行的accounts[1]变量是Account类型，调用的是第9行定义的InitialAmount成员。一般在代码重构，或者实现其他代码库中定义的接口时，会碰到这种需要使用new关键字隐藏基类成员的情况。

代码清单 2- 11继承与虚函数示例

|  |
| --- |
| 1 // 源码位置：第二章\InheritanceDemo.cs 2 // 编译命令：csc InheritanceDemo.cs 3 using System; 4  5 public class Account 6 { 7     public string Name { get; protected set; } 8  9     public decimal InitialAmount { get; private set; } 10  11     public DateTime Begin { get; private set; } 12  13     public Account(string name, decimal initial) 14     { 15         Name = name; 16         InitialAmount = initial; 17         Begin = DateTime.Now.AddDays(-100); 18     } 19  20     public virtual decimal Amount 21     { 22         get { return InitialAmount; } 23     } 24 } 25  26 public sealed class SavingAccount : Account 27 { 28     public static decimal InterestRate { get; private set; } = 0.03m; 29  30     public SavingAccount(string name, decimal initial) 31         : base(name, initial) {} 32  33     public override decimal Amount 34     { 35         get 36         { 37             var days = (decimal)(DateTime.Now - Begin).TotalDays; 38             return InitialAmount\* 39                 (1 + InterestRate / 365 \* days); 40         } 41     } 42 } 43  44 public class StockAccount : Account 45 { 46     public StockAccount(string name, decimal initial) 47         : base(name, initial)  48     { 49         Name = name + "的股票账户"; 50     } 51  52     public override decimal Amount 53     { 54         get 55         { 56             var rnd = new Random(); 57             var earning = (decimal)rnd.NextDouble() - 0.5m; 58             return ((Account)this).InitialAmount \* (1 + earning); 59         } 60     } 61  62     public new decimal InitialAmount { get; set; } 63 } 64  65 public class InheritanceDemo 66 { 67     static void Main() 68     { 69         var name = "张三"; 70         var amount = 10000m; 71         var accounts = new Account[] { 72             new SavingAccount(name, amount), 73             new StockAccount(name, amount) 74         }; 75  76         foreach (var acc in accounts) 77             Console.WriteLine($"截止目前：{acc.Name}的余额是{acc.Amount}"); 78  79         var stock = (StockAccount)accounts[1]; 80         stock.InitialAmount = 99999; 81         Console.WriteLine($"股票账户初始余额：{stock.InitialAmount}，" + 82             $"而对应的账户数组中余额：{accounts[1].InitialAmount}"); 83     } 84 } |

## 抽象类和接口

在组件化编程中，组件之间往往互相依赖，组件之间需要一个契约来定义和描述这种依赖关系，通常这种契约是由接口（interface）来描述。接口定义和类型定义类似，只是将class关键字换成interface关键字，接口里也可以定义成员方法、属性和事件，但这些成员均没有方法体，这是因为接口是一个契约，只是告诉外部组件，本组件能够提供的服务，具体的服务往往由其他类型来实现。通过接口，外部组件只需要根据接口的内部定义和相关文档说明，就能与组件操作，好比我们用电，只需要将电器的插头接口电源插座就可用电，而不需要关心电源插座背后的电流是如何产生的情况一样。用电这个例子里，电源插座就是电网对外公开的接口，这个接口定义了电压，是双头插座，还是三头插座等契约规范，而具体的电流服务则有电网内部的其他模块实现，电器等外部模块只需要了解契约的规范，如是采用二头插头，还是三头插头来调用接口，就可以使用到电网的服务。由于接口是组件对外部组件的契约，所有的成员都是公开（public）的，所以接口的成员不能使用访问修饰符限定，而且接口本身也不能实例化。

接口只能包含方法的声明，不能包含任何方法体实现，C#中提供了抽象类（abstract class）的概念来补充接口的使用场景。与接口类似，不能直接实例化抽象类，但抽象类的方法可以包含方法体，抽象类里既可以有普通的成员方法，也可以有虚方法（virtual method），还可以包含抽象方法（abstract method），前两者可以有方法体，但后者类似接口中的成员方法，不能有方法体，但是可以使用访问修饰符限定。接口定义了外部模块使用本模块的契约，而抽象类定义了外部模块的子类型继承的契约。抽象类和接口之间的区别如代码清单 2- 10：

表2 - 9抽象类和接口的区别

|  |  |
| --- | --- |
| 抽象类 | 接口 |
| 抽象类的成员可以使用访问修饰符限定 | 接口成员不能使用访问修饰符限定，接口所有成员都是公开的。 |
| 一个类只能从一个类型继承下来，即C#中不允许多重继承。 | 一个类可以实现任意多个接口。 |
| 抽象类里的成员方法可以有方法体，只有标注为abstract的方法才没有方法体。 | 接口里只有方法的声明，方法不能有方法体。 |
| 抽象类可以有构造方法 | 接口不能有构造方法 |

由于抽象类里可以实现默认方法，所以可以在抽象类里为所有子类都要用到的共同方法提供默认的实现，这样可以避免大量的重复代码。抽象类一般用在将多个关系非常紧密的类型之间的抽象泛化，而接口则只需要类型声明自己能实现接口的契约即可。如代码清单 2- 10中，用户的资产可以使用抽象类和接口的理念做进一步的改造，这个例子里有两种账号，一个是储蓄账户，一个是股票账户，两种账户实际上有一点小区别，例如储蓄账户只能储蓄或提现，因为有利息所以有存款时间是一个很重要的属性；而股票账户可以支持买卖和提现，与股票账户类似的还有基金、黄金等账户，然而它们支持买卖，购买时间就显得不是很重要。基于上面的这些区别，将储蓄账户和股票账户从同一个基类继承下来显得不是很合适。代码清单 2- 11将两个账户分开，储蓄账户是独立的类型，如第12行的定义，而股票、基金等账户则抽象泛化成投资账户InvestAccount。因为投资账户这个概念过于宽泛，无法独立创建它的实例，所以这里将其定义为抽象类，然而各种投资账户的买卖都是类似的，基本都有买卖价格和数量，但实际的买卖撮合方式不同账户会略有不同，如股票的限价撮合交易，以一定的价格下单并不一定会根据下单数量完全成交，而是会根据市场上的对手单的情况决定成交数量，基金则有购买和赎回的份额限制。

代码清单 2- 11中为InvestAccount的买卖方法提供了一个默认实现，即维护账户中资产的数量和现金余额，然后具体的成交数量和撮合规则再定义抽象方法，由子类实现具体的逻辑，如54行的Buy方法和76行的BuyInternal方法。BuyInternal方法没有方法体，是抽象的，这样我们就能在代码层面给使用我们代码库的开发者定一个契约，只有实现（或者是重写override）了这个方法才能继承InvestAccount类型，如99行StockAccount的BuyInternal方法，这样最终使用代码库的开发者，只需要使用InvestAccount来统一管理所有的投资账户，而代码库的作者也可以通过抽象类和默认实现强制继承的子类按照其预想的意图来扩展InvestAccount类。我们将储蓄账户和投资账户的账户余额和提现的概念抽象成一个接口，这样即使不是从一个基类继承结构体系类型也能对外部提供类似的服务，如144行定义了IAccount接口，将SavingAccount、StockAccount和FundAccount这几个不在同一个类型继承架构中的类型的公共功能封装在一块，在144行的用户代码里，只要用户代码只关系获取账户余额（Amount）和提现（WithDraw）功能的话，就能统一放在数组里处理，如150行的foreach循环的处理方式。虽然144行定义的accounts变量是IAccount类型的数组，C#按照声明将每个元素当作IAccount接口类型处理，如果需要转换成对象实例也是允许的，一种方式是采用如154行的强制转换方式，但这种方式有一种风险是如果对象实例的类型不是要强制转换的类型，则会触发一个无效转换的异常（InvalidCastException），C#提供了**is**和**as**两个关键字来进行软性转换。**is**操作符用来判断对象实例是否转换到指定类型（包括父类和实现的接口），返回值是一个布尔值。**as**操作符则将对象实例转换到指定类型，如果转换不成功的话，则返回**null**值，而不是抛出一个异常，由于**as**操作符可能返回**null**值，所以其不能用来软性转换一个值类型，对于值类型之间的转换，如将子结构体的实例转换到父结构体类型，只能使用强制转换的方式。第159行和161行分别演示了**is**和**as**操作符的用法。

代码清单 2- 12接口与抽象类示例

|  |
| --- |
| 1 // 源码位置：第二章\InterfaceVsAbstractClassDemo.cs 2 // 编译命令：csc InterfaceVsAbstractClassDemo.cs 3 using System; 4  5 public interface IAccount 6 { 7     string Name { get; } 8     decimal Amount { get; } 9     void WithDraw(); 10 } 11  12 public sealed class SavingAccount : IAccount 13 { 14     public static decimal InterestRate {get; private set; } = 0.03m; 15     public string Name { get; private set; } 16     public decimal InitialAmount { get; private set; } 17     public DateTime Begin {get; private set;} 18  19     public SavingAccount(string name, decimal initial) 20     { 21         Name = name; 22         InitialAmount = initial; 23         Begin = DateTime.Now.AddDays(-100); 24     } 25  26     public decimal Amount 27     { 28         get 29         { 30             var days = (decimal)(DateTime.Now - Begin).TotalDays; 31             return InitialAmount \* 32                 (1 + InterestRate / 365 \* days); 33         } 34     } 35  36     public void WithDraw() 37     { 38         Console.WriteLine("SavingAccount.WithDraw!"); 39         InitialAmount = 0; 40     } 41 } 42  43 public abstract class InvestAccount : IAccount 44 { 45     public string Name { get; private set; } 46     public decimal InitialFund { get; protected set; } 47  48     public InvestAccount(string name, decimal initial) 49     { 50         Name = name; 51         Amount = InitialFund = initial; 52     } 53  54     public void Buy(decimal price, decimal amount) 55     { 56          var dealed = BuyInternal(price, amount); 57          Amount += dealed; 58          InitialFund -= price \* dealed; 59     } 60  61     public void Sell(decimal price, decimal amount) 62     { 63         if (price < 0) 64         { 65             Amount -= amount; 66             InitialFund += MarketPrice \* amount; 67         } 68         else 69         { 70             var dealed = SellInternal(price, amount); 71             Amount -= dealed; 72             InitialFund += price \* dealed; 73         } 74     } 75      76     protected abstract decimal BuyInternal(decimal price, decimal amount); 77     protected abstract decimal SellInternal(decimal price, decimal amount); 78     public decimal Amount { get; protected set;} 79     public abstract decimal MarketPrice { get; } 80  81     public virtual void WithDraw() 82     { 83         Sell(-1, Amount); 84         InitialFund = 0; 85         Console.WriteLine("InvestAccount.WithDraw，T + 0提现"); 86     } 87 } 88  89 public class StockAccount : InvestAccount 90 { 91     Random \_rnd = new Random(); 92     public override decimal MarketPrice 93     { 94         get { return (decimal)\_rnd.NextDouble(); } 95     } 96     public StockAccount(string name, decimal initial) 97         : base(name, initial) {} 98  99     protected override decimal BuyInternal(decimal price, decimal amount) 100     {    101         Console.WriteLine("StockAccount.BuyInternal"); 102         return amount \* (decimal)\_rnd.NextDouble(); 103     } 104  105     protected override decimal SellInternal(decimal price, decimal amount)  106     { 107         Console.WriteLine("StockAccount.SellInternal"); 108         return amount \* (decimal)\_rnd.NextDouble(); 109     } 110 } 111  112 public class FundAccount : InvestAccount 113 { 114     public override decimal MarketPrice { get { return 1m; } } 115  116     public FundAccount(string name, decimal initial) 117         : base(name, initial) {} 118  119     protected override decimal BuyInternal(decimal price, decimal amount) 120     { 121         Console.WriteLine("FundAccount.BuyInternal"); 122         return amount; 123     } 124  125     protected override decimal SellInternal(decimal price, decimal amount)  126     { 127         Console.WriteLine("FundAccount.SellInternal"); 128         return amount; 129     } 130  131     public override void WithDraw() 132     { 133         Console.WriteLine("FundAccount.WithDraw，需要 T + X"); 134         InitialFund = 0; 135     } 136 } 137  138 public class InterfaceVsAbstractClassDemo 139 { 140     static void Main() 141     { 142         var name = "张三"; 143         var amount = 10000m; 144         var accounts = new IAccount[] { 145             new SavingAccount(name, amount), 146             new StockAccount(name, amount), 147             new FundAccount(name, amount) 148         }; 149  150         foreach (var account in accounts) 151             account.WithDraw(); 152  153         Console.WriteLine("-------------------------------"); 154         var fund = (FundAccount)accounts[2]; 155         fund.Buy(1, 100); 156  157         for (var i = 0; i < accounts.Length; ++i) 158         { 159             if (accounts[i] is InvestAccount) 160             { 161                 var invest = accounts[i] as InvestAccount; 162                 invest.Buy(1, 200); 163                 invest.WithDraw(); 164             } 165         } 166     } 167 } |

## 泛型

C# 2.0开始添加了对泛型的支持，泛型通过引入类型参数的概念，允许程序员编写通用类型和方法定义好规范，在使用时只要替换类型参数达到代码重用的效果。泛型经常用在集合类型里，在没有泛型之前，.NET Framework提供一个通用的可以包含任意类型元素的集合类 –ArrayList，里面的元素都以Object类型的形式保存，导致几个问题：

* 在使用时需要强制转换到实际的类型，由于元素都是以Object类型读写，在运行时万一将不一致的类型元素存入ArrayList，在强制转换的时候会失败，甚至触发转换失败的异常，导致不必要的编程错误。
* 如果元素是值类型的话，读写元素还涉及到装箱和拆箱操作，带来不必要的性能损耗。

泛型可以用在类型、接口、方法、事件和委托的定义上，而且.NET Framework在System.Collections.Generic命名空间里定义了很多泛型集合类，用来补充相应的System.Collections命名空间里的集合类，这些集合类我们将在后文讨论。是.NET Framework中泛型类List的源码的精简版[[6]](#footnote-6)，代码清单 2- 12演示了泛型类的定义 – 在类名后面加上尖括号“<>”，其中“T”就是类型参数，它是一个占位符，在使用List<T>类时，用具体的类型如int32、string或者自定义类型等替换它即可，即List<int>、List<string>或List<UserDefinedClass>，在运行时CLR会将占位符“T”替换成这些实际的类型使用– 如代码清单 2- 12中的intList和strList就是List<T>具象化的实例对象。

代码清单 2- 13 List泛型集合类的定义和使用

|  |
| --- |
| public class List<T> // ... 省略掉的代码 ... {     private T[] \_items;     private int \_size;     private int \_version;                     public void Add(T item) {         if (\_size == \_items.Length) EnsureCapacity(\_size + 1);         \_items[\_size++] = item;         \_version++;     }      // ... 省略掉的代码 ... }  var intList = newList<int>();  for (var i = 0; i <10; ++i) {  intList.Add(i);  }  var strList = newList<string>();  strList.Add("A string message"); |

泛型类可以定义任意多个类型参数，类型参数命名通常使用有意义的单词，并以T开头，如TInput、TOutput这样的名字。或者只用一个字母命名，而且选用从T开始的字母 ，如T、U等 – 如果类型参数超过两个，笔者强烈建议使用有意义的单词命名。C#允许使用**where**关键字在类型参数上添加一些限制条件，告诉CLR只有满足条件的类型才能具象化泛型类，这些限制条件参见表2 - 10：

表2 - 10泛型的类型参数的限制条件

|  |  |
| --- | --- |
| 限制条件 | 说明 |
| where T : struct | 类型参数T必须是一个值类型 –并包括除Nullable<T>以外的自定义结构体。 |
| where T : class | 类型参数必须是一个引用类型，如类型、接口、委托或者数组类型。 |
| where T : unmanaged | 类型参数不能是一个引用类型，而且类型里也不包括对引用类型的任何引用，这个限制条件在C# 7.3之后才引入。 |
| whereT : new() | 类型参数必须是有一个公开的无参构造函数，如果类型参数指定了多种限制条件，这个条件必须放到最后。 |
| where T :  <基类名称> | 类型参数必须是从指定的基类继承下来的子类 – 包括基类本身。 |
| where T :  <接口名称> | 类型参数必须实现了指定的接口，支持多个接口的限制条件，每个接口使用逗号分隔，而且接口本身也可以是一个泛型。 |
| where T : U | 如果有多个类型参数时，限制T必须是U或者从U继承下来的子类。 |

与C++的模版类不同，C++模版的具象过程是在编译期实现的，因此在模版类里使用类型参数的成员或方法在编译期是可以确定的。而C#的泛型是在运行时确定的，所以需要在编译期确定指定的类型参数是否有某些方法或成员，采用表2 - 10列出的限制条件就能让编译器确定这些信息。代码清单 2- 13列举了常用的类型参数的限制条件的使用示范，第1、30 - 33行分别演示了struct限制条件的使用，虽然限制条件名为struct，但并不限定于结构体，只要是值类型（Nullable类型除外）都是允许的。而与只允许值类型对应的就是第5、35 - 39行演示的只允许引用类型的class限制条件。第7行演示了new限制条件的用法，这个条件要求替换的类型必须有一个公开的无参构造函数，因此TestBaseClass就不能用在这个泛型里，如46行，但是值类型却可以，加入这个限制条件主要是为了便于在泛型类中初始化类型参数的实例的。由于很多场景下C#编译器不允许使用未初始化的变量，所以添加了default关键字来使用默认值来赋值，对于值类型default的作用就是将其内存全部置零，引用类型就是将其赋值为null，如第9行。第13行则演示了使用默认构造函数构造类型实例的办法，对于值类型其作用与default相同，而引用类型则会调用无参构造函数。

代码清单 2- 14泛型类型参数限制条件示例

|  |
| --- |
| 1 public class ValueTypeOnly<T> where T : struct {} 2  3 public struct TestStruct {} 4  5 public class RefTypeOnly<T> where T : class {} 6  7 public class NewableOnly<T> where T : new()  8 { 9     private T \_item = default(T); 10     public NewableOnly() 11     { 12         Console.WriteLine($"在new之前：{\_item}"); 13         \_item = new T(); 14         Console.WriteLine($"在new之后：{\_item}"); 15     } 16 } 17  18 public class TestBaseClass 19 { 20     protected TestBaseClass() {}  21 } 22  23 public class TestDeriveClass : TestBaseClass 24 { 25     public TestDeriveClass() : base()  26     { 27         Console.WriteLine("TestDeriveClass.ctor被调用了！"); 28     } 29 } 30 var vto1 = new ValueTypeOnly<int>(); 31 var vto2 = new ValueTypeOnly<TestStruct>(); 32 // TestBaseClass是引用类型，不能具象化ValueTypeOnly 33 // var vto3 = new ValueTypeOnly<TestBaseClass>(); 34  35 var rto1 = new RefTypeOnly<Object>(); 36 var rto2 = new RefTypeOnly<TestBaseClass>(); 37 // 值类型不能具象化ValueTypeOnly 38 // var rto3 = new RefTypeOnly<int>(); 39 // var rto4 = new RefTypeOnly<TestStruct>(); 40  41 var nbo1 = new NewableOnly<TestDeriveClass>(); 42 // 值类型是可以用在new() 限制条件中的 43 var nbo2 = new NewableOnly<TestStruct>(); 44 var nbo3 = new NewableOnly<int>(); 45 // TestBaseClass的构造函数是protected，不能使用 46 // var nbo4 = new NewableOnly<TestBaseClass>(); |

C# 7.3之后对泛型增加了更多的类型限制条件，它们分别是：

泛型除了可以用在类型上，还能用在其他地方，我们将在后文讨论集合类时再次探讨泛型的使用。

## 内嵌类、匿名类和多元组

C#中允许类型内部再定义类型和结构体，这种定义在其他类型或者结构体内部的类型称之为内嵌类（nested types），内嵌类的访问修饰符规则与类成员相似，默认访问修饰符是**private**，在定义时，可以给内嵌类指明如下的访问修饰符：

* 内嵌类（class）可以指定**public**，**protected**，**internal**，**protectedinternal**, **private**和 **privateprotected**。如果外部类是**sealed**的，那么就不能指定**protected**，**protectedinternal**和**privateprotected**修饰符。
* 内嵌结构题（struct）可以指定public，internal和private。

内嵌类可以访问包含它的外部类型的所有成员，包含**private**和**protected**的成员，也包括继承的**protected**的成员。代码清单 2- 12是内嵌类的定义实例，这个代码也是从.NET参考源码[[7]](#footnote-7)中提取出来的Hashtable的源码定义。

代码清单 2- 15定义内嵌类

|  |
| --- |
| public class Hashtable // : 省略实现的接口 {     private struct bucket     {         public Object key;         public Object val;         public int hash\_coll;     }      private bucket[] buckets; } |

当两个类强耦合，一个类基本上只会被另外一个类用到时，用内嵌类是一个非常好的方案，而且嵌套类通常随着包含类定义在同一个源文件中，便于后期代码维护。代码清单 2- 12中的bucket结构体是一个典型的内嵌类使用场景，其只表述哈希表内部结构，对于外部类型应该隐藏其实现方式。

如果嵌套类只会在一个方法内用到时– 这种应用场景主要用在后文要讨论的Linq中，单独定义嵌套类显得有点繁琐，C#提供了匿名类（anonymous types）这个语法糖，允许在方法里隐式定义一个嵌套类，编译器自动会在编译期创建类型定义。代码清单 2- 13第9行就定义了一个匿名类，由于匿名类没有预先定义，也就没有类名，所以使用**var**关键字来由编译器推导变量person的类型，对象实例时用new关键字不需要指定类型直接创建，类型的成员变量在创建时定义，编译器会根据这些信息为我们补足类型定义。

代码清单 2- 16定义匿名类

|  |
| --- |
| 1 // 源码位置：第二章\AnonymousTypeDemo.cs 2 // 编译命令：csc AnonymousTypeDemo.cs 3 using System; 4  5 class AnonymousTypeDemo 6 { 7     static void Main() 8     { 9         var person = new { Name = "张三", Age = 18 }; 10         Console.WriteLine($"姓名：{person.Name}。"); 11     } 12 } |

由于方法调用只能返回一个值，如果需要返回多个值，有几种方案：

* 使用**out**和**ref**关键字，其优点是快且直观，一个缺点是方法里定义了过多的输出（**out**）和传值引用（**ref**）参数影响代码的阅读与理解，另一个缺点是他们不能用在Linq和Lambda表达式。
* 定义一个类型或结构体将需要返回的值都封装在里面，其缺点是如果只有少数方法用类型，频繁定义类型略显繁琐。

在C# 7.0中，加入了元组（**tuples**）的语法糖用来将方法的多个返回值打包在一起返回。代码清单 2- 14演示了元组的定义和使用，定义和使用元组的语法有点类似匿名类，但元组的定义是将成员字段包含在“()”，而不是“{}”中。编译器有几种规则来定义元组的成员字段：

1. 直接指定成员字段名，如第10行的personTuple1的定义方式。
2. 不指定成员字段名，由编译器指派默认的以“Item”开头后面跟上字段在元组的索引组合成的名字，如第11行的personTuple2的第一个成员字段，它的名字是“Item1”，值是“张三”，第二个成员字段名是“Item2”，值是30。
3. 在C# 7.1开始，可以不指定成员字段名，由编译器通过赋值的局部变量名来命名，如第12行，personTuple3的第一个成员字段名是“name”，而第二成员字段还是以“Item”开头，然而由于其是第二个成员，因此名字是“Item2”而不是“Item1”。由于这个功能是最新的功能，在编译时需要加上C#的版本，如第2行注释里说明的编译命令。

其实在C# 7.0以前就提供了类似元组的ValueTuple的泛型结构体，最多可以包含8个成员字段，每个成员字段名就是以“Item”开头的，而编译器实际上也是在背后为程序里定义的每个元组都隐式定义了基于ValueTuple的类型，这也是元组成员默认名称以“Item”开头的原因。

注意：虽然编译器为元组提供了默认名称，在实际编程中强烈建议读者命名元组每个成员，增强代码的可读性，不要为了利用这一点编译器技巧而故意不做命名，这种由编译器生成默认名称的功能一般用在代码生成的场景里。

在赋值元组时，也遵循类似的规则，如第13行的copyTuple1的每个字段都定义了名字，因此跟personTuple1能一一对应；而第14行的copyTuple2没有定义名字，只是字段类型与personTuple1一致，copyTuple2的字段名则是以“Item”开头的默认规则。因为元组是基于ValueType类型的语法糖，所以它是一个值类型，也就有对应的可空类型，在第15和第23行代码演示元组与可空类型的互换。

第29行ReturnTuple这个方法返回的是元组类型，返回值的类型的定义规则与元组定义类似，这里我们返回的是一个包含字符串型name字段和整型age字段的元组，由于方法的返回值已经说明了元组的详细定义，所以在返回元组时，即可以通过指定字段名称赋值的方式 – 如31行，也可以让编译器推导赋值 – 如32行。

代码清单 2- 17元组（Tuple）使用示例

|  |
| --- |
| 1 // 源码位置：第二章\TupleDemo.cs 2 // 编译命令：csc /langversion:"7.1" TupleDemo.cs 3 using System; 4  5 public class TupleDemo 6 { 7     static void Main() 8     { 9         var name = "张三"; 10         var personTuple1 = ( name: "张三", age: 30 ); 11         var personTuple2 = ( "张三", 30 ); 12         var personTuple3 = ( name, 30 ); 13         (string name, int age) copyTuple1 = personTuple1; 14         (string, int) copyTuple2 = personTuple1; 15         (string name, int age)? nullableTuple = personTuple1; 16  17         Console.WriteLine($"{personTuple1.name}, {personTuple1.age}"); 18         Console.WriteLine($"{personTuple2.Item1}, {personTuple2.Item2}"); 19         Console.WriteLine($"{personTuple3.name}, {personTuple3.Item2}"); 20         Console.WriteLine($"{copyTuple1.name}, {copyTuple1.age}"); 21         Console.WriteLine($"{copyTuple2.Item1}, {copyTuple2.Item2}"); 22         Console.WriteLine( 23             $"{nullableTuple.Value.name}, {nullableTuple.Value.age}"); 24  25         var tuple4 = ReturnTuple(); 26         Console.WriteLine($"{tuple4.name}, {tuple4.age}"); 27     } 28  29     static (string name, int age) ReturnTuple() 30     { 31         // return (name: "张三", age: 30); 32         return ("张三", 30); 33     } 34 } |

嵌套类、匿名类和元组的功能有些重叠，之间的优劣性在网上也有不少的讨论，笔者参考一些资料个人的看法是[[8]](#footnote-8)：

* 匿名类和元组都可以用在方法内部，但前者是引用类型，后者是值类型。如果方法需要返回多个值，值的个数不是很多的话，使用元组会方便一些，而且元组的值类型属性意味着返回值拷贝的，而匿名类这种引用类型是在GC上分配的，如果要大量调用方法，使用元组可能有一些性能上的提升。然而元组在编译后字段名实际上还是会转换成“ItemX”的形式，无法使用反射技术通过源码中的字段名在运行时处理，而且也无法用在ASP.NET MVC中的razor页面里。
* 元组也可以在一些场景中替代嵌套类的作用，如像二维、三维坐标占用内存很小的成员属性，也可以考虑用元组来定义，这种做法可以利用元组即时定义字段的好处。代码清单 2- 15就演示这种字段的定义和使用方法。

代码清单 2- 18使用元组定义类型的成员字段

|  |
| --- |
| public class TypeHasTupleField {     public (string name, int age) Person { get; set; }      public int Id { get; set; } }  var vtf = new TypeHasTupleField() {     Person = (name: "张三", age: 30),     Id = 2 }; Console.WriteLine($"{vtf.Person.name}, {vtf.Person.age}"); |

## 解构

C# 7.0不仅支持在方法里通过元组返回多个数值，还增加将元组内的字段解构（deconstruct）到多个局部变量的语法糖，优化使用元组字段的方法。代码清单 2- 16中演示了C#支持的几种解构语法，其中第9、12、15和18 – 20行分别是几种不同的局部变量的定义和赋值手法，便于我们在编程时使用。如果方法返回的一部分字段在代码中不会用到的话，也可以使用下划线“\_”（discard）忽略掉，这个语法糖可以避免编译器在编译时报告某些局部变量并未使用的警告信息。可以使用“\_”忽略掉元组的任意多个字段，如37行的做法一样。当然至少要将一个元组字段赋值给局部变量，否则”(\_, \_, \_) = ReturnTuple()”的调用不如“ReturnTuple()”这样更直接了。

解构语法除了可以用在元组以外，还可以用在类型上，只要类型定义了一个公开的名为Deconstruct的实例方法，这个实例方法可以有多个重载，将需要解构的字段采用输出（**out**）参数赋值，编译器在编译时会自动根据解构语义匹配正确的重载版本，如54和60行分别定义了两种解构方式，在31和37行就可以直接跟解构元组一样解构类型。60行的第3个参数coord还演示了使用解构语法将多个字段解构成新的语义的方法。

使用扩展方法技术也可以将解构语法用在C# 7.0之前的编写的类型上，这个扩展方法的名称也必须是Deconstruct，第一个参数就是要解构的类型实例，剩下的均是输出参数用来定义解构的语义，在本书附带的示例代码DeconstructDemo.cs里有相应的例子，请读者自行参阅。

代码清单 2- 19元组解构示例

|  |
| --- |
| 1 // 源码位置：第二章\DeconstructDemo.cs 2 // 编译命令：csc DeconstructDemo.cs 3 using System; 4  5 public class DeconstructDemo 6 { 7     static void Main() 8     { 9         (string name1, int age1) = ReturnTuple(); 10         Console.WriteLine($"{name1}，{age1}"); 11  12         var (name2, age2) = ReturnTuple(); 13         Console.WriteLine($"{name2}，{age2}"); 14  15         (string name3, var age3) = ReturnTuple(); 16         Console.WriteLine($"{name3}，{age3}"); 17  18         string name4 = null; 19         int age4 = 0; 20         (name4, age4) = ReturnTuple(); 21         Console.WriteLine($"{name4}，{age4}"); 22  23         var (name5, \_) = ReturnTuple(); 24         Console.WriteLine($"{name5}"); 25  26         Coordinate coord = new Coordinate() 27         { 28             Latitude = 100.0, 29             Longitude = 150.0 30         }; 31         var (lat, lon) = coord; 32         Console.WriteLine($"{lat} - {lon}"); 33  34         var (lat1, \_) = coord; 35         Console.WriteLine($"{lat1}"); 36  37         (\_, \_, var latlon) = coord; 38         Console.WriteLine(latlon); 39     } 40  41     static (string name, int age) ReturnTuple() 42     { 43         // return (name: "张三", age: 30); 44         return ("张三", 30); 45     } 46 } 47  48 public class Coordinate 49 { 50     public double Latitude { get; set; } 51  52     public double Longitude { get; set; } 53  54     public void Deconstruct(out double lat, out double lon) 55     { 56         lat = Latitude; 57         lon = Longitude; 58     } 59  60     public void Deconstruct(out double lat, out double lon, out string coord) 61     { 62         lat = Latitude; 63         lon = Longitude; 64         coord = string.Format($"({Latitude},{Longitude})"); 65     } 66 } |

## 析构方法和IDisposable接口

前文介绍了C# 7.0之后才引入的解构(deconstruct)语法糖，中文的翻译与另一个语法糖析构(destructor)方法很像 – 甚至英文也很相似。当然两者是有本质的区别的，析构方法是自C# 1.0就存在的语法糖，作用与C++语言中的析构方法有点类似，也是在释放对象时执行清理操作的。由于C#程序是托管程序，一般可由CLR管理的如内存等资源，都不需要显式释放，在程序执行时由GC自动处理，但是对于一些资源，如数据库连接、通过操作系统打开的文件等，CLR无法管理它们，因此也被称为非托管资源，析构方法就是用来在C#程序中告诉CLR释放这些非托管资源的方法。代码清单 2- 17中41和49行就演示了析构方法的定义，它与C++的析构方法的语法非常相似，但是用法却不尽相同。首先需要注意的是析构方法没有访问修饰符，而且在C#中不能直接调用它– 不像C++那样有一个delete关键字，其作用实质上是重写了Object类型的Finalize方法，通过ildasm反编译程序也可以看出这一点。C#编译器不允许直接重写Object类型的Finalize方法，只能通过析构方法的语法来重写，通过这个限制，C#编译器保证子类的析构方法能够调用到基类的析构方法，确保资源的有效释放，如运行本程序，会发现在第14行的分割线之前，对象dc的析构方法包含了对其基类DestructorBaseClass的析构方法的调用。

代码清单 2- 20析构方法和IDisposable接口示例

|  |
| --- |
| 1 // 源码位置：第二章\DesctructorDemo.cs 2 // 编译命令：csc DesctructorDemo.cs 3 using System; 4  5 public class DesctructorDemo 6 { 7     static void Main() 8     { 9         UseDestructorObjects(); 10  11         GC.Collect(); 12         GC.WaitForPendingFinalizers();  13  14         Console.WriteLine("------- 分割线 ---------"); 15         UseDispose1(); 16         GC.Collect(); 17         GC.WaitForPendingFinalizers(); 18     } 19  20     static void UseDestructorObjects() 21     { 22         var dbc = new DestructorBaseClass(); 23         var dc = new DestructorClass(); 24         dbc.DummyMethod(); 25         dc.DummyMethod(); 26     } 27  28     static void UseDispose1() 29     { 30         using (var dc = new DestructorClass()) 31         { 32             dc.DummyMethod(); 33         } 34     } 35 } 36  37 public class DestructorBaseClass 38 { 39     public void DummyMethod() {} 40  41     ~DestructorBaseClass() 42     { 43         Console.WriteLine("DestructorBaseClass destructor"); 44     } 45 } 46  47 public class DestructorClass : DestructorBaseClass, IDisposable 48 { 49     ~DestructorClass() 50     { 51         Console.WriteLine("DestructorClass destructor"); 52     } 53  54     public void Dispose() 55     { 56         GC.SuppressFinalize(this); 57         Console.WriteLine("DestructorClass Dispose"); 58     } 59 } |

CLR的垃圾收集程序（GC）会对定义了析构方法的对象做一些特殊处理，对于这些对象，当GC认为其不再被程序所使用的时候，首先将其放到一个Finalizer队列当中，然而这些对象不会被释放掉，在第二次GC的时候再依次执行队列中每个对象的析构方法再被释放掉，也就是说定义了析构方法的对象在内存中存活的时间较其他对象会长，也就潜在增加了内存的负担。在代码清单 2- 17中也演示了这个问题，第11行在程序中强制触发了一次GC操作，然后在第12行调用WaitForPendingFinalizer方法等待GC执行完毕对象的析构方法才释放掉对象，如果将第12行注释掉再执行，会观察到析构方法的执行有时会在14行 – 也就是分割线之后，这是因为GC的执行是不可提前预知的。

由于使用析构方法会导致对象在内存存活的时间会稍微长一些，所以.NET Framework又定义了一个特殊的接口 –IDisposable，这个接口只有一个方法Dispose，类型可以通过实现这个接口让用户程序显式释放对象的非托管资源。IDisposable接口与析构方法的功能重复，如果对象同时实现了两者，在Dispose方法里需要调用SuppressFinalize方法 – 如第56行，一来避免非托管资源的双重释放，二来告诉GC不要将对象放到Finalizer队列中减少对象在内存中的存活时间。一般建议将Dispose方法用在try … finally 语句的finally语句块中，确保无论程序是否正常执行，都能释放掉非托管资源，使用方法参见本书附带示例代码的Destructor.cs中的UseDispose2方法。C#为了鼓励程序员编写良好的代码，甚至提供了一个语法糖来做这件事情，如代码清单 2- 17第30行的using语句块，在语句块的结尾 – 即第33行编译器会生成finally语句块并调用Dispose方法。在实际编程中，推荐读者使用IDisposable接口来显式释放非托管资源，尽量不要用析构方法。

# 委托和事件

委托（**delegate**）有点类似C/C++中的函数指针或回调方法，但相对来说类型更安全，而且委托可以调用静态方法和实例方法 – 这个特性在C++里实现难度较大，创建委托的语法是：

<access modifier> delegate <returnType> MethodName ([parameters])

代码清单2 - 18中第7行定义名为OrderBy的委托，其接受两个int型参数，并返回一个布尔值，因此其只能匹配接受两个int型参数并返回布尔值的方法，如第25行和30行的Asc和Desc方法；定义好委托后，将其当做一个普通类型作为方法的参数定义，表明该方法接受该委托类型标记（signature）的回调方法，如第9行的Sort方法的order参数；创建委托的语法类似新建一个类的实例，将需要用委托封装的方法名传递给委托就能创建，如第43行；而使用委托与调用方法的语法一样，如15行对order这个委托实例的调用，如果不是第9行的参数类型指明这是一个委托，阅读代码时很容易将其当做一个普通方法对待。

代码清单2 - 18使用委托

|  |
| --- |
| 1 // 源码位置：第二章\DelegateDemo.cs  2 // 编译命令：csc DelegateDemo.cs  3 using System;  4  5 public class DelegateDemo  6 {  7 public delegate bool OrderBy(int left, int right);  8  9 private static void Sort(int[] items, OrderBy order)  10 {  11 for (var i = 0; i < items.Length; ++i)  12 {  13 for (var j = i; j < items.Length; ++j)  14 {  15 if (order(items[i], items[j]))  16 {  17 int swap = items[i];  18 items[i] = items[j];  19 items[j] = swap;  20 }  21 }  22 }  23 }  24  25 private static bool Asc(int left, int right)  26 {  27 return left > right;  28 }  29  30 private static bool Desc(int left, int right)  31 {  32 return left < right;  33 }  34  35 static void Main(string[] args)  36 {  37 var items = new int[10];  38 var rnd = new Random();  39 for (var i = 0; i < items.Length; ++i)  40 items[i] = rnd.Next(0, 100);  41  42 // var order = new OrderBy(Asc);  43 var order = new OrderBy(Desc);  44 Sort(items, order);  45 for (var i = 0; i < items.Length; ++i)  46 Console.Write($"{items[i]} ");  47 Console.WriteLine();  48 }  49 } |

代码清单2 - 18也演示了委托的典型应用场景，即方法回调，更进一步细分可以分为异步处理和补足通用方法的代码：

* **异步处理**：当程序调用长时间才能完成的方法时，与其被动等待这个方法的完成再继续后面代码的执行，可以使用委托将长时间的方法放到一个独立的线程中运行，同时继续后面代码的执行，当线程执行完毕后，通过委托通知程序回来处理执行结果。
* **补足通用方法**：这也是代码清单2 - 18演示的场景，排序有多种排序条件，如升序和降序，对一些复杂类型的排序，可能有根据不同字段排序的条件。但排序算法都是固定的（代码清单2 - 18中为了演示使用了性能最差也最简单的冒泡排序），采用委托可以只提供一个通用的排序算法，在算法的使用过程中，可以根据实际的需求来动态修改排序的条件，如42和43行分别代表升序和降序的排序条件。

跟**decimal**类型相似，.NET Framework中也有System.MulticastDelegate这个类与委托对应，代码清单2 - 18中，使用委托时就像定义了一个新的类型那样可以直接使用，实际上所有C#的委托都继承自这个类型，因此一个委托实例拥有MulticastDelegate类的相同成员方法。MulticastDelegate顾名思义，这种委托可以触发多个方法调用。代码清单2 - 19中第24和28行分别演示了使用Combine和Remove方法向委托增加和删减方法调用的语法，由于这两个方法都只接受Delegate类型的参数，所以需要用类似创建对象新实例的方法，通过**new**关键字将方法转换成委托。针对Combine和Remove方法，C#提供了一个语法糖，即可以用“**+**”和“**-**”操作符来增减委托的方法调用列表，如第32、35和41行分别演示了这个语法优化，有兴趣的读者可以用ildasm工具来看一下代码清单2 - 19的反编译结果，会发现在源码中虽然我们可以使用“**+**”和“**-**”操作符直接使用具体的方法作为操作数修改委托，但在实际的IL代码中，C#还是在后台默默使用**new**关键字将方法转换成了一个委托实例。

虽然代码清单2 - 19中演示的都是将静态方法赋值给委托的例子，但也可以将实例方法赋值给委托，将实例方法赋值给委托的方法与将静态方法赋值给委托的方法相同，然而在通过委托回调实例方法时，Delegate类型的Target字段保存了实例方法所需要的**this**实例。

代码清单2 - 19 MulticastDelegate使用示例

|  |
| --- |
| 1 // 源码位置：第二章\MultiCastDelegateDemo.cs  2 // 编译命令：csc MultiCastDelegateDemo.cs  3 using System;  4  5 public delegate void GreetingDelegate(string value);  6  7 public class MultiCastDelegateDemo  8 {  9 static void SayHello(string value)  10 {  11 Console.WriteLine("Hello {0}", value);  12 }  13  14 static void SayGoodbye(string value)  15 {  16 Console.WriteLine("Bye {0}", value);  17 }  18  19 static void Main()  20 {  21 GreetingDelegate singleDel1 = SayHello;  22 singleDel1("singleDel1");  23  24 GreetingDelegate multiDel1 = (GreetingDelegate)Delegate.Combine(  25 singleDel1, new GreetingDelegate(SayGoodbye));  26 multiDel1("multiDel1");  27  28 GreetingDelegate singleDel2 = (GreetingDelegate)Delegate.Remove(  29 multiDel1, singleDel1);  30 singleDel2("singleDel2");  31  32 GreetingDelegate multiDel2 = singleDel1 + SayGoodbye;  33 multiDel2("multiDel2");  34  35 GreetingDelegate singleDel3 = multiDel2 - SayGoodbye;  36 singleDel3("singleDel3");  37  38 GreetingDelegate multiDel3 = SayHello;  39 multiDel3("multiDel3");  40  41 multiDel3 += SayGoodbye;  42 multiDel3("multiDel3");  43 }  44 } |

委托的一个主要作用就是作为类型的回调方法使用，在一些场景中经常需要用到这种回调方法来通知外部类型。如在窗体（WinForm）程序和ASP.NET WebForm程序中，当用户点击了某个按钮，或者输入了一些文本，需要将这些事件及时通知到感兴趣的类型中，以便它们处理做出回应。C#提供了事件（**event**）的语法来支持这些场景，触发事件的类型被称作发布者（publisher），而监听事件并做出响应的类型被称作订阅者（subscriber），这种发布者-订阅者模式是很常见的编程模式。如在窗体（Winform）程序中，当用户点击一个按钮时，按钮控件类型会触发一个点击事件，按钮控件就是发布者，而监听点击事件并做出响应的类型，如执行一个网络下载动作，这个类型时订阅者。

事件也是基于委托的，在类型中定义事件需要指明事件的委托类型，这个委托只接受两个参数，第一个object类型的参数用来传递触发事件的发布者实例。第二个参数可以是任意类型，用来在发布者和订阅者之间传递处理事件所需的数据。例如对于输入文本事件，这个参数可以传递用户输入的文本。但对于有些事件，如按钮点击事件，可能不需要传递数据，但C#语法还是要求定义第二个参数。为了规范事件的使用，.NET Framework定义了EventHandler委托和其泛型版EventHandler<T>，并定义了EventArgs类型用来传递事件数据，开发人员可以从这个类型中继承新的事件参数类型。

代码清单 2- 21中第12和第19行分别演示了监听事件和取消事件订阅的方法，在25行为文本输入事件定义了新的类型用来传递用户输入的文本，在C#不支持泛型之前，我们只能像第33行那样要么定义一个新的委托类型，要么就是使用通用的EventHandler类型，然后在事件处理程序中将EventArgs强制转换成需要的类型，有了泛型支持后，只需要像37行那样就可以了。第45和51行则分别演示了在发布者触发事件时，将数据传递给订阅者的方法，第51行需要特别说明一下，虽然按钮点击事件不需要传递任何数据，但这里笔者还是创建了一个EventArgs实例，而不是传递null值，这是为了避免事件订阅者不做null值检查引起异常抛出从而导致整个程序意外终止的情况。

代码清单 2- 21 事件示例

|  |
| --- |
| 1 // 源码位置：第二章\EventsDemo.cs 2 // 编译命令：csc EventsDemo.cs 3 using System; 4  5 public class EventsDemo 6 { 7     static void Main() 8     { 9         var publisher = new Publisher(); 10         var subscriber = new Subscriber(); 11  12         publisher.ButtonClicked += subscriber.ProcessClick; 13         publisher.OnEditing += subscriber.ProcessNewText; 14  15         publisher.RaiseTextInput(); 16         publisher.RaiseButtonClicked(); 17  18         Console.WriteLine("演示移除事件监听程序的方法"); 19         publisher.OnEditing -= subscriber.ProcessNewText; 20         publisher.RaiseTextInput(); 21         publisher.RaiseButtonClicked(); 22     } 23 } 24  25 public class TextInputEventArgs : EventArgs 26 { 27     public TextInputEventArgs(string text) { Text = text; } 28     public string Text { get; } 29 } 30  31 public class Publisher 32 { 33     public delegate void TextInputEvent(object sender,TextInputEventArgs args); 34  35     // 使用泛型版的EventArgs<T>可以免掉每次定义新委托的麻烦 36     // 下面注释的事件定义与更下面一行的事件定义等价 37     // public event EventHandler<TextInputEventArgs> OnEditing; 38     public event TextInputEvent OnEditing; 39  40     public event EventHandler ButtonClicked; 41  42     public void RaiseTextInput() 43     { 44         if (OnEditing != null) 45             OnEditing(this, new TextInputEventArgs("Some text")); 46     } 47  48     public void RaiseButtonClicked() 49     { 50         if (ButtonClicked != null) 51             ButtonClicked(this, new EventArgs()); 52     } 53 } 54  55 public class Subscriber 56 { 57     public void ProcessNewText(object sender, TextInputEventArgs args) 58     { 59         Console.WriteLine("Got new text: {0}", args.Text); 60     } 61  62     public void ProcessClick(object sender, EventArgs args) 63     { 64         Console.WriteLine("Button clicked"); 65     } 66 } |

# 小结

本章快速叙述了C#的基础语法，这些语法足够开发大型程序，并适用于很多互联网开发中，也是C# 2.0就基本涵盖的语法。C#后续版本中也添加了不少新的语法和新的编程范式，笔者将在后文结合实际例子或者具体的编程范式来介绍这些新的语法。

1. C#所有的关键字参考在线文档：<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>。 [↑](#footnote-ref-1)
2. 所有C#运算符请参考在线文档：<https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/>。 [↑](#footnote-ref-2)
3. 所有可以重载的操作符参考在线文档：<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators>。 [↑](#footnote-ref-3)
4. 关于这个非常特殊的使用场景，StackExchange的工程师有一篇很好的博客说明了这个优化：<https://blog.marcgravell.com/2011/10/assault-by-gc.html>。 [↑](#footnote-ref-4)
5. value在属性的写入方法里是作为一个关键字处理的，在写入方法里不能定义跟value同名的变量，但在其它方法里，value则作为普通变量处理了。 [↑](#footnote-ref-5)
6. 完整源码位置：https://referencesource.microsoft.com/mscorlib/system/collections/generic/list.cs.html [↑](#footnote-ref-6)
7. .NET Reference Source：是微软开源的.NET框架源码，地址在：https://referencesource.microsoft.com/ [↑](#footnote-ref-7)
8. 更详尽的对比，有兴趣的读者请参阅：http://download.microsoft.com/download/5/4/B/54B83DFE-D7AA-4155-9687-B0CF58FF65D7/tuples-anon-types-concrete.pdf [↑](#footnote-ref-8)