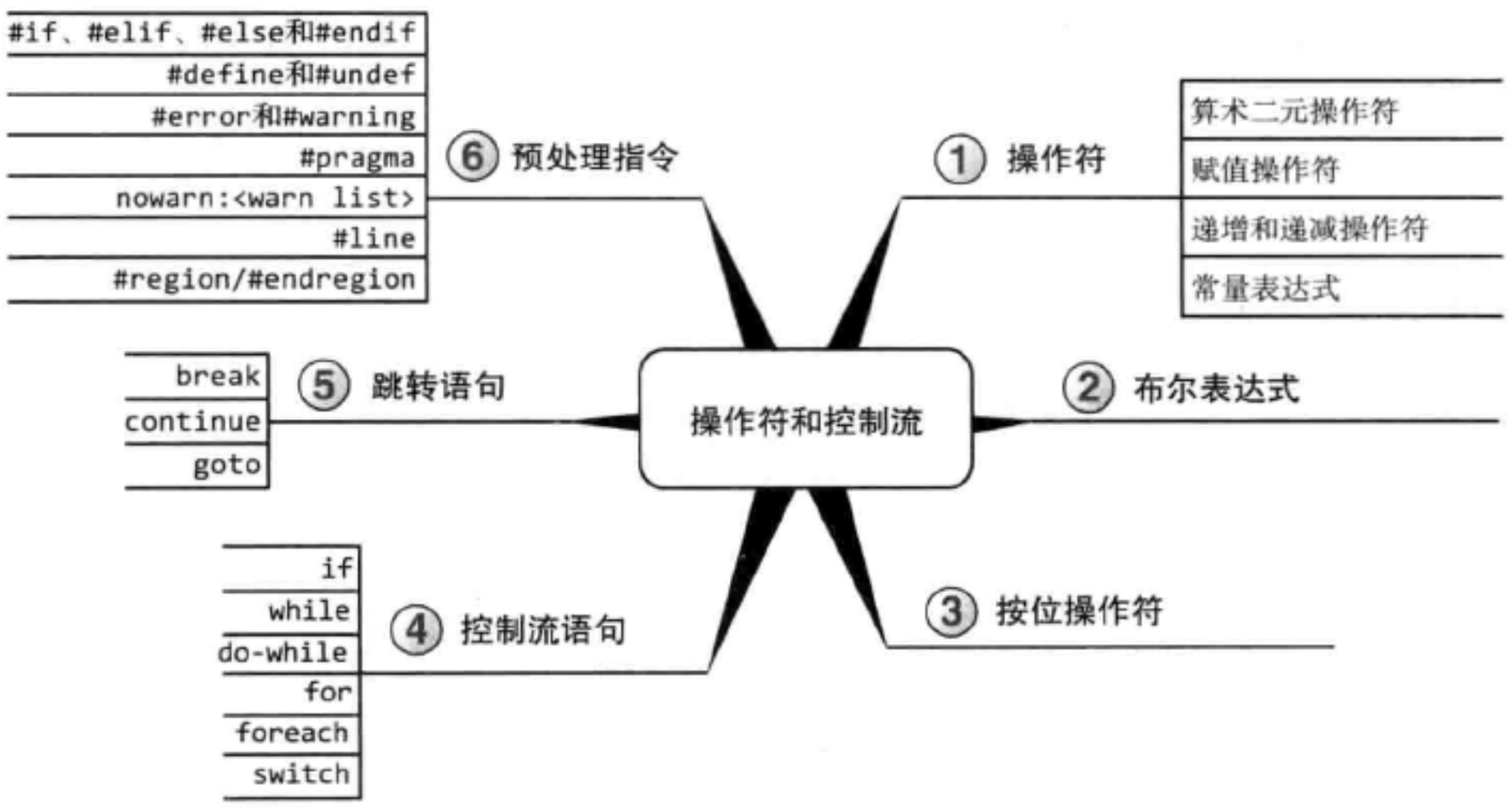
第3章

操作符和控制流

本章将讲述操作符、控制流语句和C#预处理器。利用操作符所提供的特定语法，可以针对计算中涉及的操作数执行各种不同的计算或操作。利用控制流语句，可以控制程序的条件逻辑，或多次重复一段代码。介绍了if控制流语句之后，本章将探讨布尔表达式的概念，这种表达式将嵌入许多控制流语句中。我们还会提到整数不能转型（即使是显式转型）为bool，并讨论了这个设计的好处。本章最后讨论C#预处理指令。



3.1 操作符

第2章学习了预定义数据类型，接着将更多地学习如何利用操作符结合这些数据类型来执行各种计算。例如，可以对已经声明好的变量执行计算。

初学者主题：操作符

操作符（operator）对一系列称为**操作数**（operand）的值（或变量）执行数学或逻辑运算/操作来生成新值（称为**结果**）。例如，代码清单3-1有两个操作数，也就是数字4和2，它们被减法操作符（-）组合到一起，结果赋给变量difference。

代码清单3-1 简单的操作符例子

```
int difference = 4 - 2;
```

通常将操作符划分为3大类：一元操作符、二元操作符和三元操作符，它们对应的操作数分别是1个、2个和3个。本节讨论了最基本的一些一元和二元操作符，三元操作符将在本章后面作简略介绍。

3.1.1 一元操作符正和负

有时需要改变数值变量的正负号。这时一元负操作符（-）就能派上用场。例如，代码清单3-2将当前的美国国债金额变成负值，指明这是欠款。

代码清单3-2 指定负值[①]

```
//National Debt to the Penny
decimal debt = -15236332233848.35M;
```

使用负操作符等价于从零减去操作数。

一元正操作符（+）对值几乎[②]没有影响。它在C#语言中是多余的，只是出于对称性的考虑才加进来。

3.1.2 二元算术操作符

二元操作符要求两个操作数。C#为二元操作符使用中缀表示法：操作符在左右操作数之间。每个二元表达式的结果要么赋给一个变量，要么以某种方式使用（例如用作为另一个表达式的操作数）。

语言对比：C++——仅有操作符的语句

和上面提到的规则相反，C++甚至允许像4+5；这样的二元表达式作为独立的语句使用。在C#中，只有调用、递增、递减和对象创建表达式才能作为独立的语句使用。

代码清单3-3是使用二元操作符（更准确地说，是二元算术操作符）的例子。算术操作符的每一边都有一个操作数，计算结果赋给一个变量。除了二元减法操作符（-），其他二元算术操作符还有加法（+）、除法（/）、乘法（*）和取余操作符（%，有时也称为取模操作符）。

---

① 这是2012年1月12日的美国国债数据，数据来自www.treasurydirect.gov。

② 一元+操作符定义为获取int、unit、long、ulong、float、double和decimal类型（及其可空版本）的操作数。用于其他类型（如short）时，操作数会根据实际情况转换为上述某个类型。

代码清单3-3 二元操作符的使用

```
class Division
{
  static void Main()
  {
      int numerator;
      int denominator;
      int quotient;
      int remainder;

      System.Console.Write("Enter the numerator: ");
      numerator = int.Parse(System.Console.ReadLine());

      System.Console.Write("Enter the denominator: ");
      denominator = int.Parse(System.Console.ReadLine());

      quotient = numerator / denominator;
      remainder = numerator % denominator;

      System.Console.WriteLine(
          "{0} / {1} = {2} with remainder {3}",
          numerator, denominator, quotient, remainder);
  }
}
```

输出3-1展示了结果。

输出3-1

```
Enter the numerator: 23
Enter the denominator: 3
23 / 3 = 7 with remainder 2
```

在上例突出显示的赋值语句中，除法和取余操作在赋值之前发生。操作符的执行顺序取决于它们的优先级和结合性。迄今为止用过的操作符的优先级如下。

（1）*、/和%具有最高优先级。

（2）+和-具有较低优先级。

（3）=在6个操作符中优先级最低。

所以，在上面的例子中，语句的行为符合预期，除法和取余先于赋值进行。

如果忘记对二元操作符的结果进行赋值，会出现如输出3-2所示的编译错误。

输出3-2

```
... error CS0201: Only assignment, call, increment, decrement,
and new object expressions can be used as a statement
```

初学者主题：圆括号、结合性、优先级和求值

当一个表达式包含了多个操作符时，可能会让人看不清楚每个操作符的操作数是什么。例如在表达式x+y*z中，很明显表达式x是+操作符的操作数，z是*操作符的操作数。但y是+还是*的操作数？

圆括号可以明确地将一个操作数与它所属的操作符相关联。如果希望y是被加数，

可以写(x+y)*z。如果希望它是被乘数，可以写x+(y*z)。

但包含多个操作符的表达式不一定非要添加圆括号。编译器能根据结合性和优先级从上下文中判断程序员省略了哪些圆括号。**结合性**（associativity）决定相似操作符的执行顺序，**优先级**（precedence）决定不相似操作符的执行顺序。

二元操作符可以“左结合”或“右结合”，具体取决于“位于中间”的表达式是从属于左边的操作符，还是从属于右边的操作符。例如，a-b-c被判定为(a-b)-c，而不是a-(b-c)。这是因为减法操作符为“左结合”。C#的大多数操作符都是左结合的，赋值操作符右结合。

对于不相似的操作符，要根据操作符的优先级决定位于中间的操作数从属于哪一边的操作符。例如，乘法操作符的优先级高于加法操作符，因此表达式x+y*z表示的是对x+(y*z)求值，而不是对(x+y)*z求值。

有时候，圆括号操作符并不改变表达式的求值结果。不过，使用圆括号来提高代码的可读性依然是一良好的编程的习惯。例如，在执行摄氏-华氏温度换算时，表达式(c*9.0/5.0)+32.0比c*9.0/5.0+32.0的可读性更好，即使在这个表达式中完全没有必要使用圆括号。

规范

要使用圆括号增加代码的易读性，尤其是在操作符优先级不是让人一目了然的时候。

很明显，对于相邻的两个操作符，高优先级的先于低优先级的先执行。例如x+y*z是先乘后加，乘法结果是加法的右操作数。但要注意，优先级和结合性只影响操作符自身的执行顺序，不影响操作数的求值顺序。

在C#中，操作数总是从左向右求值。在包含3个方法调用的表达式中，比如A()+B()*C()，首先求值A()，然后B()，然后C()，然后乘法操作符决定乘积，最后加法操作符决定和。不能因为C()是乘法操作数，A()是低优先级的加法运算的操作数，就认为对方法C()的调用先于对方法A()的调用。

语言对比：C++ —— 操作数求值顺序

和上述规则相反，C++规范允许不同的实现自行选择操作数求值顺序。对于A()+B()*C()这样的表达式，不同的C++编译器可选择以不同顺序求值函数调用，只要乘积是某个被加数即可。例如，可以选择先求值B()，再A()，再C()，再乘法，最后加法。

1. 将加法操作符用于字符串

操作符也可用于非数值类型。例如，可以使用加法操作符来拼接两个或者更多字符串，如代码清单3-4所示。

代码清单3-4 将二元操作符应用于非数值类型

```
class FortyTwo
{
  static void Main()
  {
      short windSpeed = 42;
      System.Console.WriteLine(
          "The original Tacoma Bridge in Washington\nwas "
          + "brought down by a "
          + windSpeed + " mile/hour wind.");
  }
}
```

输出3-3展示了代码清单3-4的结果。

输出3-3

```
The original Tacoma Bridge in Washington
was brought down by a 42 mile/hour wind.
```

由于不同语言文化的语句结构迥异，所以开发者注意不要对需要本地化的字符串使用加法操作符。在这种情况下，复合格式化更理想（参见第1章）。

规范

要用复合格式化而不是加法操作符来拼接字符串。

2. 在算术运算中使用字符

上一章介绍char类型的时候，曾提到，虽然char类型存储的是字符而不是数字，但它是整型（意味着它基于整数）。可以和其他整型一起参与算术运算。然而，不是基于存储的字符来解释char类型的值，而是基于它的基础值。例如，数字3包含Unicode值0x33（十六进制），换算成十进制值是51。数字4包含Unicode值0x34，或者十进制的52。如果像代码清单3-5那样，将3和4加到一起，会获得十六进制值0x167，即十进制的103，等价于字母g。

代码清单3-5 将加法操作符应用于char数据类型

```
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c);  // Writes out g.
```

输出3-4展示了代码清单3-5的结果。

输出3-4

```
g
```

可以利用char类型的这个特点判断两个字符相距多远。例如，字母f与字母c有3个字符的距离。为了获得这个值，可以用字母f减去字母c，如代码清单3-6所示。

代码清单3-6 判断两个字符之间的“距离”

```
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

输出3-5展示了代码清单3-6的结果。

输出3-5

```
3
```

3. 浮点类型的特殊性

浮点类型float和double有一些特殊性，比如它们处理精度的方式。本节通过一些具体的例子来帮助你认识浮点类型的一些特殊性。

float具有7位精度，能容纳值1 234 567和值0.123 456 7。然而，如果将这两个float值加到一起，结果会被取整为1 234 567，因为小数部分超过了float能够容纳的7位有效数字。这种类型的取整有时是致命的，尤其是在执行重复性计算或者检查相等性的时候（参见稍后的“高级主题：浮点类型造成非预期的不相等”）。

注意，二进制浮点类型内部实际存储的是二进制分数而不是十进制分数。所以，一次简单的赋值就可能引发精度问题，比如double number = 140.6F。140.6的准确值是分数703/5，但分母不是2的整数次幂，所以无法用二进制浮点数准确表示。实际上，值是用一个最接近16位float的，并且分母为2的整数次幂的分数来表示的。

由于double能容纳比float更精确的值，所以C#编译器实际会将这个表达式求值为double number = 140.600 0061 0351 6，这是一个最接近float类型的140.6的二进制分数，但表示成double比140.6稍大。

规范

避免在需要准确的十进制算术运算时使用二进制浮点类型，而是使用decimal浮点类型。

高级主题：浮点类型造成非预期的不相等

比较两个值是否相等的时候，浮点类型的不准确性可能造成非常严重的后果。有的时候，本来应该相等的值被错误地判断为不相等，如代码清单3-7所示。

代码清单3-7 浮点类型的不准确性造成的非预期不相等

```
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", decimalNumber, (decimal)doubleNumber1);
```

```
Trace.Assert((double)decimalNumber != doubleNumber1);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", (double)decimalNumber, doubleNumber1);

Trace.Assert((float)decimalNumber != floatNumber);
// Displays: (float)4.2M != 4.2F
System.Console.WriteLine(
    "(float){0}M != {1}F",
    (float)decimalNumber, floatNumber);

Trace.Assert(doubleNumber1 != (double)floatNumber);
// Displays: 4.20000006258488 != 4.20000028610229
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, (double)floatNumber);

Trace.Assert(doubleNumber1 != doubleNumber2);
// Displays: 4.20000006258488 != 4.2
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, doubleNumber2);

Trace.Assert(floatNumber != doubleNumber2);
// Displays: 4.2F != 4.2D
System.Console.WriteLine(
    "{0}F != {1}D", floatNumber, doubleNumber2);

Trace.Assert((double)4.2F != 4.2D);
// Display: 4.19999980926514 != 4.2
System.Console.WriteLine(
    "{0} != {1}", (double)4.2F, 4.2D);

Trace.Assert(4.2F != 4.2D);
// Display: 4.2F != 4.2D
System.Console.WriteLine(
    "{0}F != {1}D", 4.2F, 4.2D);
```

输出3-6展示了代码清单3-7的结果。

输出3–6

```
4.2 != 4.20000006258488
4.2 != 4.20000006258488
(float)4.2M != 4.2F
4.20000006258488 != 4.20000028610229
4.20000006258488 != 4.2
4.2F != 4.2D
4.19999980926514 != 4.2
4.2F != 4.2D
```

Assert()方法在参数求值为false的时候显示对话框提醒开发人员“断言失败”[①]。然而，上述代码中的所有Assert()语句都求值为true。所以，虽然值理论上应该相等，但由于浮点数的不准确性，它们被错误地判断为不相等。

① 为了使上述代码顺利编译，请添加using System.Diagnostics;指令。——译者注

规范

避免将二进制浮点类型用于相等性条件式。要么判断两个值之差是否在容差范围之内，要么使用decimal类型。

浮点类型还有其他特殊性。例如，整数除以零理论上应出错。int和decimal等数据类型确实会如此。然而，float和double允许结果是特殊值，如代码清单3-8和输出3-7所示。

代码清单3-8 浮点数除以零，显示“非数字”

```
float n=0f;
// Displays: NaN
System.Console.WriteLine(n / 0);
```

输出3-7

```
NaN
```

在数学中，特定的算术运算是未定义的，例如0除以它自己。在C#中，浮点0除以0会得到“Not a Number”（非数字）。试图打印这样的一个数，实际输出的就是“非数字”。类似地，获取负数的平方根（System.Math.Sqrt(-1)）也会得到“非数字”。

浮点数也可能溢出其边界。例如，float的上边界大约是3.4×10^{38}。一旦溢出这个边界，结果数就会存储为“正无穷大”，打印这个数，会输出“正无穷大”。类似地，float的下边界是-3.4×10^{38}，溢出这个边界会得到“负无穷大”。代码清单3-9分别生成正负无穷大，输出3-8展示了结果。

代码清单3-9 溢出一个浮点值的边界

```
// Displays: -Infinity
System.Console.WriteLine(-1f / 0);
// Displays: Infinity
System.Console.WriteLine(3.402823E+38f * 2f);
```

输出3-8

```
-Infinity
Infinity
```

进一步检查浮点数，发现它能包含非常接近零、但实际不是零的值。如果值超过float或double类型的阈值，值可能表示成“负零”或者“正零”，具体取决于数是负还是正，并在输出中表示成-0或者0。

3.1.3 复合赋值操作符

第1章讨论了简单的赋值操作符，它将操作符右边的值赋给左边的变量。复合赋值操作符将常见的二元操作符与赋值操作符结合，如代码清单3-10所示。

代码清单3-10 常见的递增计算

```
int x = 123;
x = x + 2;
```

在上述赋值运算中，首先计算x + 2，再将结果赋回x。由于这种形式的运算相当普遍，所以专门有一个操作符集成了计算和赋值。+=操作符使左边的变量递增右边的值，如代码清单3-11所示。

代码清单3-11 使用+=操作符

```
int x = 123;
x += 2;
```

上述代码等价于代码清单3-10。

还有其他复合赋值操作符提供了类似的功能。赋值操作符还可以和减法、乘法、除法和取余操作符合并，如代码清单3-12所示。

代码清单3-12 其他赋值操作符的例子

```
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

3.1.4 递增和递减操作符

C#提供了特殊的一元操作符来实现计数器的递增和递减。**递增操作符**（++）每次使一个变量递增1。所以，代码清单3-13的每一行代码的作用都是相同的。

代码清单3-13 递增操作符

```
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```

类似地，还可以使用**递减操作符**（--）使变量递减1。因此，代码清单3-14的每一行代码的作用都是相同的。

代码清单3-14 递减操作符

```
lines = lines - 1;
lines -= 1;
lines--;
```

初学者主题：循环中的递减示例

递增和递减操作符在循环（比如稍后要讲到的while循环）中经常用到。例如，代码清单3-15使用递减操作符，逆向遍历字母表中的每个字母。

代码清单3-15 降序显示每个字母的ASCII值

```
char current;
int unicodeValue;
```

```
// Set the initial value of current.
current='z';

do
{
  // Retrieve the Unicode value of current.
  unicodeValue = current;
  System.Console.Write("{0}={1}\t", current, unicodeValue);

  // Proceed to the previous letter in the alphabet;
  current--;
}
while(current>='a');
```

输出3-9展示了代码清单3-15的结果。

输出3–9

```
z=122   y=121   x=120   w=119   v=118   u=117   t=116   s=115   r=114
q=113   p=112   o=111   n=110   m=109   l=108   k=107   j=106   i=105
h=104   g=103   f=102   e=101   d=100   c=99   b=98   a=97
```

递增和递减操作符用于控制特定操作的执行次数。在这个例子中，还要注意递减操作符作用于字符（char）数据类型。只要数据类型支持“下一个值”和“上一个值”的概念，就适合使用递增和递减操作符。

以前说过，赋值操作符首先计算要赋的值，再执行赋值。赋值操作符的结果是所赋的值。递增和递减操作符与此相似。也是计算要赋的值，执行赋值，再返回结果值。所以赋值操作符可以和递增或递减操作符一起使用。但是，如果不仔细，可以得到令人困惑的结果，如代码清单3-16和输出3-10所示。

代码清单3–16 使用后缀递增操作符

```
int count = 123;
int result;
result = count++;
System.Console.WriteLine(
  "result = {0} and count = {1}", result, count);
```

输出3–10

```
result = 123 and count = 124
```

赋给result的是count递增前的值。递增或递减操作符的位置决定了所赋的值是操作数计算之前还是之后的值。如果希望result的值包括递增（或递减）计算的结果，需要将操作符放在想要递增的变量之前，如代码清单3-17所示。

代码清单3–17 使用前缀递增操作符

```
int count = 123;
int result;
result = ++count;
System.Console.WriteLine(
  "result = {0} and count = {1}", result, count);
```

输出3-11展示了代码清单3-17的结果。

输出3-11

```
result = 124 and count = 124
```

在这个例子中，递增操作符出现在操作数之前，所以表达式生成的结果是递增后赋给变量的值。假定count为123，那么++count将124赋给count，表达式生成的结果124。相反，后缀形式count++会将124赋给count，表达式生成的结果递增前count所容纳的值，即123。无论操作符采用后缀还是前缀形式，变量count都会在表达式的结果生成之前递增，区别只是生成哪一个值。代码清单3-18展示了前缀和后缀操作符在行为上的差异，输出3-12展示了结果。

代码清单3-18 对比前缀和后缀递增操作符

```
class IncrementExample
{
  public static void Main()
  {
      int x = 123;
      // Displays 123, 124, 125.
      System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
      // x now contains the value 125.
      // Displays 126, 127, 128
      System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
      // x now contains the value 128.
  }
}
```

输出3-12

```
123, 124, 125
126, 127, 128
```

在代码清单3-18中，递增和递减操作符相对于操作数的位置影响了表达式的结果。前缀操作符的结果是变量递增/递减之后的值，而后缀操作符的结果是变量递增/递减之前的值。在语句中使用这些操作符应该小心。若心存疑虑，最好独立使用这些操作符（自成一个语句）。这样，不仅代码更易读，还可保证不犯错。

语言对比：C++——由实现定义的行为

前面说过，在C++中，表达式中的操作数可以按照任意顺序求值，而在C#中，总是从左向右。类似地，在C++中实现递增和递减时，可以任意选择是“修改后使用”，还是“使用后修改”。例如在C++中，对于M(x++, x++)这样的调用，假定x的初值是1，那么既可以调用M(1,2)，也可以调用M(2,1)，具体由编译器决定。C#则总是调用M(1,2)，因为C#做出了两点保证。第一，传给调用的实参总是从左向右计算。第二，总是先将被递增的值赋给变量，再使用表达式的值。这两点在C++中都不保证。

规范

避免混淆递增和递减操作符的用法。

在C、C++和C#之间移植使用递增和递减操作符的代码时要小心；C和C++的实现遵循的不一定是与C#相同的规则。

高级主题：线程安全的递增和递减

虽然递增和递减操作符简化了代码，但两者执行的都不是原子级别的运算。在操作符执行期间，可能发生线程上下文切换，可能造成竞争条件。可以用lock语句来防止出现竞争条件。然而，对于简单的递增和递减运算，一个代价没有那么高的替代方案是使用由System.Threading.Interlocked类提供的线程安全方法Increment()和Decrement()。这两个方法依赖于处理器的功能来执行快速的、线程安全的递增和递减运算（详情参见第19章）。

3.1.5 常量表达式和常量符号

上一章讨论了字面量，或者说直接嵌入代码的值。可以用操作符将多个字面值合并到**常量表达式**中。根据定义，常量表达式是C#编译器能在编译时完成求值的表达式（而不是在程序运行时才能求值），因其完全由常量操作数构成。然后，可以用常量表达式初始化常量符号，从而为常量值分配名称（类似于局部变量为存储位置分配名称）。例如，可用常量表达式计算一天中的秒数，结果赋给一个常量符号，并在其他表达式中使用该符号。

在代码清单3-19中，const关键字的作用就是声明常量符号。由于常量和“变量”相反——“常”意味着“不可变”——以后在代码中任何修改它的企图都会造成编译时错误。

代码清单3-19 声明常量

```
// ...
public long Main()
{
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;
    // ...
}
```

常量表达式：`60 * 60 * 24`

常量：`secondsPerWeek`

注意，赋给secondsPerWeek的也是常量表达式。表达式中的所有操作数都是常量，编译器能确定结果。

规范

不要使用常量表示将来可能改变的任何值。pi和金原子的质子数是常量。金价、公司名和程序版本号则应该是变量。

3.2 控制流程概述

本章后面的代码清单3-43展示了如何以一种简单的方式查看一个数的二进制形式。然而，即便如此简单的程序，不用控制流语句也是写不出来的。控制流语句控制着程序的执行路径。本节将讨论如何基于条件检查来改变语句的执行顺序。之后将学习如何通过循环结构来反复执行一组语句。

表3-1总结了所有控制流语句。注意，“常规语法结构”这一列给出的只是常见的语句用法，不是完整的词法结构（lexical structure）。

表3-1中的embedded-statement是指除了加标签的语句或声明之外的任何语句，但通常是语句块。

表3-1中的每个C#控制流语句都出现在了附录B的井字棋[①]程序中。这段程序会显示井字棋棋盘，提示每个玩家，并在每一次走棋之后更新。

表3–1 控制流语句

语 句	常规语法结构	示 例
if语句	**if**(*boolean-expression*) *embedded-statement*	`if (input == "quit")` `{` `  System.Console.WriteLine(` `  "Game end");` `  return;` `}`
	if(*boolean-expression*) *embedded-statement* **else** *embedded-statement*	`if (input == "quit")` `{` `  System.Console.WriteLine(` `  "Game end");` `  return;` `}` `else` `  GetNextMove();`
while语句	**while**(*boolean-expression*) *embedded-statement*	`while(count < total)` `{` `  System.Console.WriteLine(` `  "count = {0}", count);` `  count++;` `}`
do while语句	**do** *embedded-statement* **while**(*boolean-expression*);	`do` `{` `  System.Console.WriteLine(` `  "Enter name:");` `  input =` `  System.Console.ReadLine();` `}` `while(input != "exit");`
for语句	**for**(*for-initializer*; *boolean-expression*; for-iterator) *embedded-statement*	`for (int count = 1;` `  count <= 10;` `  count++)` `{` `  System.Console.WriteLine(` `  "count = {0}", count);` `}`

① 也有叫画圈打叉游戏的。

续表

语 句	常规语法结构	示 例
foreach语句	**foreach**(*type identifier* **in** *expression*) *embedded-statement*	**foreach** (char letter **in** email) { if(!insideDomain) { if (letter == '@') { insideDomain = true; } continue; } System.Console.Write(letter); }
continue语句	**continue**;	
switch语句	**switch**(governing-type-*expression*) { ... **case** const-expression: statement-list jump-statement **default**: statement-list jump-statement }	switch(input) { case "exit": case "quit": System.Console.WriteLine("Exiting app...."); break; case "restart": Reset(); goto case "start"; case "start": GetMove(); break; default: System.Console.WriteLine(input); break; }
break语句	**break**;	
goto语句	**goto** *identifier*;	
	goto case *const-expression*;	
	goto default;	

本章剩余部分将更详细地讨论每一条语句。讨论了if语句之后，在接触其他控制流语句之前，首先要解释代码块、作用域、布尔表达式以及按位操作符的概念。由于C#与其他语言具有很多相似性，部分读者可能发现这个表格非常熟悉。这部分读者可以直接跳到3.9节，或者直接跳到本章的小结部分。

3.2.1 if语句

if语句是C#中最常见的语句之一。它对称为**条件**（condition）的**布尔表达式**（返回true或false的表达式）进行求值，条件为true将执行后续语句（consequence-statement）。if语句可以有else子句，其中包含了在条件为false时将执行的替代语句（alternative-statement）。常规形式如下：

```
if (condition)
  consequence-statement
else
  alternative-statement
```

在代码清单3-20中，如果玩家输入1，程序将显示"Play against computer selected."；否则显示"Play against another player."。

代码清单3-20 if/else语句示例

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      string input;

      // Prompt the user to select a 1- or 2-player game.
      System.Console.Write (
          "1 - Play against the computer\n" +
          "2 - Play against another player.\n" +
          "Choose:"
      );
      input = System.Console.ReadLine();

      if(input=="1")
          // The user selected to play the computer.
          System.Console.WriteLine(
              "Play against computer selected.");
      else
          // Default to 2 players (even if user didn't enter 2).
          System.Console.WriteLine(
              "Play against another player.");
  }
}
```

3.2.2 嵌套if

代码有时需要多个if语句。代码清单3-21首先判断玩家是否输入小于或等于0的数选择退出，如果不是，就检查用户是否知道井字棋的最大走棋步数。

代码清单3-21 嵌套if语句

```
1.  class TicTacToeTrivia
2.  {
3.    static void Main()
4.    {
5.        int input;    // Declare a variable to store the input.
6.
7.        System.Console.Write(
8.            "What is the maximum number " +
9.            "of turns in tic-tac-toe?" +
10.           "(Enter 0 to exit.): ");
11.
12.       // int.Parse() converts the ReadLine()
13.       // return to an int data type.
14.       input = int.Parse(System.Console.ReadLine());
15.
16.       if (input <= 0)
17.               // Input is less than or equal to 0.
18.           System.Console.WriteLine("Exiting...");
19.       else
20.           if (input < 9)
21.               // Input is less than 9.
```

```
22.                 System.Console.WriteLine(
23.                     "Tic-tac-toe has more than {0}" +
24.                     " maximum turns.", input);
25.             else
26.                 if(input>9)
27.                     // Input is greater than 9.
28.                     System.Console.WriteLine(
29.                         "Tic-tac-toe has fewer than {0}" +
30.                         " maximum turns.", input);
31.                 else
32.                     // Input equals 9.
33.                     System.Console.WriteLine(
34.                         "Correct, tic-tac-toe " +
35.                         "has a max. of 9 turns.");
36.     }
37. }
```

输出3-13展示了代码清单3-21的结果。

输出3–13

```
What is the maximum number of turns in tic-tac-toe? (Enter 0 to exit.): 9
Correct, tic-tac-toe has a max. of 9 turns.
```

假定第14行显示提示时玩家输入9，那么程序的执行路径如下。

（1）第16行：检查input是否小于0。因为不是，所以跳到第20行。

（2）第20行：检查input是否小于9。因为不是，所以跳到第26行。

（3）第26行：检查input是否大于9。因为不是，所以跳到第33行。

（4）第33行：显示答案正确。

代码清单3-21使用了嵌套的if语句。为了分清嵌套，代码行进行了缩进。但正如在第1章学到的那样，空白不会影响执行路径。没有缩进和换行，代码执行起来是一样的。代码清单3-22展示了嵌套if语句的另一种形式，与代码清单3-21是等价的。

代码清单3–22 if/else连贯格式化

```
if (input < 0)
  System.Console.WriteLine("Exiting...");
else if (input < 9)
  System.Console.WriteLine(
      "Tic-tac-toe has more than {0}" +
      " maximum turns.", input);
else if(input < 9)
  System.Console.WriteLine(
      "Tic-tac-toe has less than {0}" +
      " maximum turns.", input);
else
  System.Console.WriteLine(
      "Correct, tic-tac-toe has a maximum " +
      " of 9 turns.");
```

后一种格式更常见。无论如何，都应该选择会获得最清晰代码的格式。

在上述代码清单中，每个if语句都省略了大括号。但正如马上就要讲到的那样，这与编码规范不符。规范提倡除了单行语句之外都使用代码块。

3.3 代码块

在前面的if语句示例中，if和else之后仅跟随了一个语句，也就是System.Console.WriteLine();语句，如代码清单3-23所示。

代码清单3-23 不带代码块的if语句

```
if(input < 9)
  System.Console.WriteLine("Exiting");
```

使用大括号，可以将多个语句合并成**代码块**，允许在符合条件时执行多个语句。例如代码清单3-24中突出显示的用于计算半径的代码块。

代码清单3-24 带有代码块的if语句

```
class CircleAreaCalculator
{
  static void Main()
  {
      double radius;  // Declare a variable to store the radius.
      double area;    // Declare a variable to store the area.

      System.Console.Write("Enter the radius of the circle: ");

      // double.Parse converts the ReadLine()
      // return to a double.
      radius = double.Parse(System.Console.ReadLine());

      if(radius>=0)
      {
          // Calculate the area of the circle.
          area = 3.14*radius*radius;
          System.Console.WriteLine(
              "The area of the circle is: {0}", area);
      }
      else
      {
          System.Console.WriteLine(
              "{0} is not a valid radius.", radius);
      }
  }
}
```

输出3-14展示了代码清单3-24的结果。

输出3-14

```
Enter the radius of the circle: 3
The area of the circle is: 28.26
```

在这个例子中，if语句检查radius（半径）是不是正数。如果是，就计算并显示圆的面积；否则显示消息指出半径无效。

注意第一个if之后跟随了两个语句，它们被封闭在一对大括号中。大括号将多个语句合并成代码块。

假如去掉代码清单3-24中用于创建代码块的大括号，在布尔表达式返回true的前提

下，只有紧接在if语句之后的那个语句才会执行。无论布尔表达式求值结果是什么，后续的语句都会执行。代码清单3-25展示了这种无效的代码。

代码清单3-25 依赖缩进，造成无效的代码

```
if(radius>=0)
  area = 3.14 * radius *radius;
  System.Console.WriteLine(
      "The area of the circle is: {0}", area);
```

在C#中，缩进仅用来增强代码的可读性。编译器会忽略它，所以上述代码在语义上等价于代码清单3-26。

代码清单3-26 语义上等价于代码清单3-25

```
if(radius>=0)
{
  area = 3.14*radius*radius;
}
System.Console.WriteLine(
  "The area of the circle is: {0}", area);
```

程序员必须防止此类不容易发现的错误。一种比较极端的做法是，无论如何都在控制流语句之后包括代码块，即使其中只有一个语句。事实上，设计规范是除非单行语句，否则不要去掉大括号 。

虽然比较少见，但也可独立使用代码块，它在语义上不属于任何控制流语句。换言之，大括号可自成一体（例如，没有条件或循环），这是完全合法的。

规范

避免在if语句中省略大括号，除非只有一行语句。

高级主题：数学常量

在代码清单3-25和代码清单3-26中，pi的值3.14是硬编码进去的，它非常粗略。在System.Math类中，pi和E有着更精确的定义。所以不是硬编码数学常量，而是使用System.Math.PI或者System.Math.E。

3.4 代码块、作用域和声明空间

代码块经常被称为**作用域**，但两个术语并不完全可以互换。具名事物的作用域是源代码的一个区域。在该区域中，可以使用非限定名称（前面不加限定前缀的名称）引用该事物。局部变量的作用域就是封闭它的代码块。这正是经常将代码块称为“作用域”的原因。

作用域经常和**声明空间**（declaration space）混淆。声明空间是具名事物的逻辑容器。该容器中不能存在同名的两个事物。代码块不仅定义了作用域，还定义了局部变量声明空间。在同一个声明空间中，不允许声明两个同名的局部变量。类似地，不能在同一个

类中声明两个签名都是Main()的方法（尽管对于方法的规则有一些放宽，在同一个声明空间中，允许存在签名不一样的两个同名的方法）。在声明局部变量的代码块外部，没有办法用局部变量的名称引用它；当局部变量出现在代码外部时，我们说局部变量“超出了作用域”。

简单地说，作用域决定一个名称引用什么事物，而声明空间决定同名的两个事物是否冲突。在代码清单3-27中，是在if语句主体声明局部变量message，这就将它的作用域限制在if主体。当后面的方法中使用它的名称时，这个局部变量就“超出了作用域”。要防止出现错误，必须在if语句外部声明该变量。

代码清单3-27 变量在其作用域外无法访问

```
class Program
{
    static void Main(string[] args)
    {
        int playerCount;
        System.Console.Write(
            "Enter the number of players (1 or 2):");
        playerCount = int.Parse(System.Console.ReadLine());
        if (playerCount != 1 && playerCount != 2)
        {
            string message =
                "You entered an invalid number of players.";
        }
        else
        {
            // ...
        }
        // Error:  message is not in scope.
        System.Console.WriteLine(message);
    }
}
```

输出3-15展示了代码清单3-27的结果。

输出3-15

```
...

...\Program.cs(18,26): error CS0103: The name 'message' does not exist
in the current context
```

声明空间中的每个局部变量名称必须是唯一的。声明空间覆盖了包含在最初声明局部变量的代码块中的所有子代码块。C#编译器禁止一个代码块中声明（或作为参数声明）的局部变量在其子代码块中重复声明。在代码清单3-27中，由于args和playerCount是在方法的代码块（方法主体）中声明的，所以在这个代码块的任何地方（包括子代码块）都不能再次声明它们。

message这个名称在这个局部变量的整个作用域内部引用这个局部变量。作用域是直接包含了局部变量声明的代码块。类似地，playerCount在声明了这个变量的整个代码块（包括if和else子代码块）中引用的都是同一个变量。

语言对比：C++ ——局部变量作用域

在C++中，对于块中声明的局部变量，它的作用域是从声明位置开始，到块尾结束。声明前对局部变量的引用会失败，因为局部变量此时不在作用域内。如果此时有另一个同名的事物在作用域中，C++会将名称解析成对那个事物的引用，而这可能不是你的原意。C#的规则稍有不同，对于声明局部变量的那个块，局部变量都“在作用域中”，但声明前引用它属于非法。换言之，此时可以成功找到局部变量，但对其的使用是错误的。只有在声明之后的位置使用它才合法。这是C#为了防止出现像C++中那些常见的不容易察觉之错误而设置的众多规则之一。

3.5 布尔表达式

if语句中包含在圆括号内的部分是**布尔表达式**（Boolean expression），称为**条件**。代码清单3-28突出显示了条件。

代码清单3-28 布尔表达式

```
if(input < 9)
{
    // Input is less than 9.
    System.Console.WriteLine(
        "Tic-tac-toe has more than {0}" +
         " maximum turns.", input);
}
// ...
```

布尔表达式在许多控制流语句中得到了运用，其关键特征在于总是求值为true或false。input<9要成为布尔表达式就必须返回bool值。例如，编译器不允许将布尔表达式写成x=42，因为它的作用是对x进行赋值，并返回新值，而不是检查x的值是否等于42。

语言对比：C++ ——错误地使用=来代替==

C#消除了C/C++的一个常见的编码错误。在C++中，代码清单3-29是不会出错的。

代码清单3-29 C++允许将赋值作为布尔表达式

```
if(input=9)    // Allowed in C++, not in C#.
  System.Console.WriteLine(
      "Correct, tic-tac-toe has a maximum of 9 turns.");
```

虽然上述代码表面上是检查input是否等于9，但正如第1章讲过的那样，=代表的是赋值操作符，而不是检查相等性的操作符。从赋值操作符返回的是赋给变量的值，本例中就是9。然而，作为int的9无法被判定为布尔表达式，所以是C#编译器不允许的。C 和C++将非零整数视为true，将零视为false。相反， C#要求条件必须是布尔类型，不允许整数。

3.5.1 关系操作符和相等性操作符

关系和相等性操作符用于判断一个值是否大于、小于或等于另一个值。表3-2总结了所有关系和相等性操作符。它们都是二元操作符。

表3-2 关系和相等性操作符

运 算 符	描 述	示 例
<	小于	input<9;
>	大于	input>9;
<=	小于或等于	input<=9;
>=	大于或等于	input>=9;
==	等于	input==9;
!=	不等于	input!=9;

在C#中使用相等性操作符==来测试相等性的语法和其他许多编程语言中的是一样的。例如，判断input是否等于9要使用input==9。相等性操作符使用两个等号，赋值操作符使用一个等号。C#中的惊叹号代表NOT，所以用于测试不等性的操作符是!=。

关系和相等性操作符总是生成bool值，如代码清单3-30所示。

代码清单3-30 将关系操作符的结果赋给bool变量

```
bool result = 70 > 7;
```

井字棋程序（参见附录B）中使用相等性操作符判断玩家是否退出游戏。代码清单3-31的布尔表达式包含一个OR（||）逻辑操作符，下一节将详细讨论它。

代码清单3-31 在布尔表达式中使用相等性操作符

```
if (input == "" || input == "quit")
{
  System.Console.WriteLine("Player {0} quit!!", currentPlayer);
  break;
}
```

3.5.2 逻辑布尔操作符

逻辑操作符（logic operator）获取布尔操作数并生成布尔结果。可用逻辑操作符合并多个布尔表达式来构成更复杂的布尔表达式。逻辑操作符包括|、||、&、&&和^，分别对应于OR（或）、AND（与）和XOR（异或）。OR和AND的|和&版本很少用，原因稍后讨论。

1. OR操作符（||）

在代码清单3-31中，如果玩家输入quit，或直接按回车键而不输入任何值，就认为想退出程序。为了允许玩家以两种方式退出，程序使用了逻辑OR操作符||。

||操作符对两个布尔表达式进行求值，如果其中任何一个为true，就返回true，如代码清单3-32所示。

代码清单3-32 使用OR操作符

```
if((hourOfTheDay > 23) || (hourOfTheDay < 0))
  System.Console.WriteLine("The time you entered is invalid.");
```

注意，使用布尔OR操作符时，不一定每次都要对操作符两边的表达式进行求值。和C#中的所有操作符一样，OR操作符是从左向右求值的，所以假如左边求值为true，那么右边可以忽略。换言之，如果hourOfTheDay的值为33，那么(hourOfTheDay > 23)会返回true，所以OR操作符会忽略右侧表达式。这种**短路**（short circuiting）求值方式同样适用于布尔AND操作符。（注意，此处可以去掉圆括号，因为逻辑操作符的优先级高于关系操作符。但是，对子表达式使用圆括号增强了可读性。）

2. AND操作符（&&）

布尔AND操作符&&在两个操作数求值都为true的前提下才返回true。任何操作数为false，就会返回false。

代码清单3-33判断当前小时数是否大于10且小于24[①]。如果同时满足这两个条件，就输出一条消息表明当前是工作时间。和OR操作符一样，AND操作符也并非每次都要对右边的表达式进行求值。假如左边的操作数返回false，则不管右边的操作数是什么，最终的结果肯定为false，所以“运行时”会忽略右边的操作数。

代码清单3-33 使用AND操作符

```
if ((10 < hourOfTheDay) && (hourOfTheDay < 24))
  System.Console.WriteLine(
  "Hi-Ho, Hi-Ho, it's off to work we go.");
```

3. XOR操作符（^）

^符号是异或（Exclusive OR，XOR）操作符。若应用于两个布尔操作数，那么只有在两个操作数中仅有一个为true的前提下，XOR操作符才会返回true，如表3-3所示。

表3-3 XOR运算表

左操作数	右操作数	结　果
True	True	False
True	False	True
False	True	True
False	False	False

与布尔AND和OR操作符不同，布尔XOR操作符不支持短路运算。它始终都要检查两个操作数，因为除非确切地知道两个操作数的值，否则不能判定最终结果。注意，XOR操作符和!=操作符完全相同。

3.5.3 逻辑求反操作符

逻辑求反操作符（!）有时也称为NOT操作符，它的作用是反转一个bool数据类型的值。这是一元操作符，只需要一个操作数。代码清单3-34演示了它是如何工作的，输出3-16展示了结果。

代码清单3-34 使用逻辑求反操作符

```
bool valid = false;
bool result = !valid;
// Displays "result = True".
System.Console.WriteLine("result = {0}", result);
```

① 这是程序员的一般工作时间。

输出3-16

```
result = True
```

valid最开始的值为false。求反操作符对valid的值取反，将新值赋给result。

3.5.4 条件操作符

可用条件操作符取代if-else语句来选择两个值中的一个。条件操作符同时使用一个问号和一个冒号，其常规格式如下：

```
condition ? consequence : alternative
```

条件操作符是三元操作符，因为它需要3个操作数，即condition、consequence和alternative。（作为C#中唯一的三元操作符，条件操作符也经常被称为“三元操作符”。然而，用名称称呼它比用操作符的个数称呼它更易理解。）类似于逻辑操作符，条件操作符也支持短路求值。如果condition求值为true，则条件操作符只求值consequence；否则只求值alternative。操作符的结果是被求值的表达式。

代码清单3-35展示了如何使用条件操作符。该程序的完整清单列于附录B中。

代码清单3-35 条件操作符

```
public class TicTacToe
{
  public static string Main()
  {
     // Initially set the currentPlayer to Player 1;
     int currentPlayer = 1;

     // ...

     for (int turn = 1; turn <= 10; turn++)
     {
        // ...

        // Switch players
        currentPlayer = (currentPlayer == 2) ? 1 : 2;
     }
  }
}
```

这个程序的作用是交换当前玩家。为此，它检查当前值是否为2。这是条件语句的condition部分。如果结果为true，条件操作符?会返回consequence的值1；否则返回alternative的值2。和if语句不同，条件操作符的结果必须赋给某个变量（或者作为参数传递）。它不能单独作为一个语句使用。

> **规范**
>
> 考虑使用if/else语句，而不是使用过于复杂的条件表达式。

C#语言要求条件操作符的consequence和alternative表达式类型一致，而且在判定类型时不会检查表达式的上下文。例如，f ? "abc" : 123不是合法的条件表达式，因为

consequence和alternative分别是字符串和数字，它们相互之间不能转换。即使像object result = f ? "abc" : 123;这样的语句，C#编译器也会认为非法，因为兼容两个表达式的类型（这里是object）在条件表达式的外部。

3.5.5 空接合操作符

空接合操作符（null coalescing operator）??能简单地表示“如果这个值为空，就使用另一个值”，其形式如下。

```
expression1 ?? expression2;
```

??操作符支持短路求值。如果expression1不为null，就返回expression1的值，不对另一个表达式求值。如果expression1求值为null，就返回expression2的值。和条件操作符不同，空接合操作符是二元操作符。

代码清单3-36是使用空接合操作符的例子。

代码清单3-36 空接合操作符

```
string fileName = GetFileName();
// ...
string fullName = fileName ?? "default.txt";
// ...
```

如果fileName为null，就用空接合操作符将fullName设为"default.txt"。如果fileName不为null，fullName将获得fileName的值

空结合操作符能完美地“链接”。例如，对于表达式x ?? y ?? z，如果x不为null将返回x；否则，如果y不为null将返回y；否则返回z。也就是说，从左向右选出第一个非空表达式。如果之前所有表达式都为空，就选择最后一个表达式。

空结合操作符是C# 2.0和可空值类型一起引入的，它的操作数既可以是可空值类型，也可以是引用类型。

3.6 按位操作符

几乎所有编程语言都提供了一套按位操作符，用于对值的二进制格式进行处理。

初学者主题：位和字节

计算机的所有值都表示成1和0的二进制格式，这些1和0称为位或者**比特**（bit）。8位为一组，称为**字节**（byte）。在一个字节中，每一个连续的位都对应于2的一个乘幂。其中，最右边的位对应于2^0，最左边的位对应于2^7，如图3-1所示。

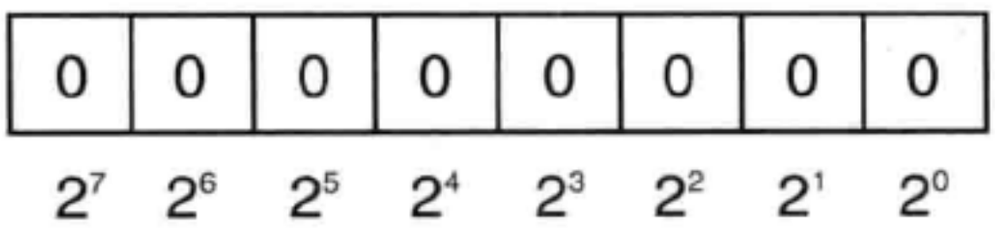


图3-1 对应的占位值

在许多情况下，尤其是在操作低级设备或系统服务的时候，信息是以二进制数据的形式获取的。操作这些设备和服务需要处理二进制数据。

如图3-2所示，每个框都对应于2的某个乘幂。字节（8位构成的一个数）的值是含有1的所有位的2的乘幂之和。

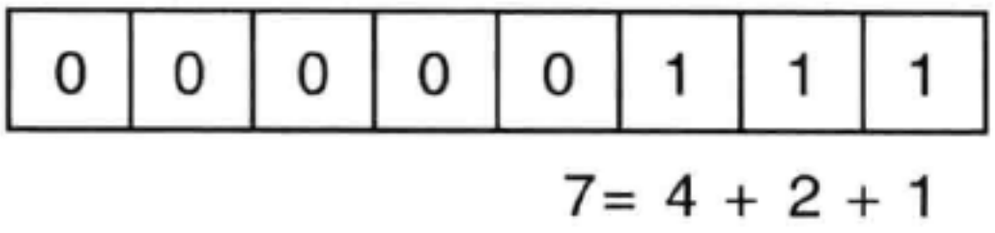


图3-2 计算无符号字节的值

对于有符号的数，二进制转换则有很大的不同。有符号的数（long、short、int）使用2的补数记数法表示。所以将负数加到正数上时，加法运算可以照常进行，就好像两个数都是正数。使用这种记数法，负数在行为上有别于正数。负数通过最左侧的1来标识。如果最左边的位置包含1，就要将含有0的位置加到一起，而不是将含有1的位置加到一起。每个位置都对应于负的（2的乘幂）。除此之外，还需要从结果中减去1。图3-3对此进行了演示。

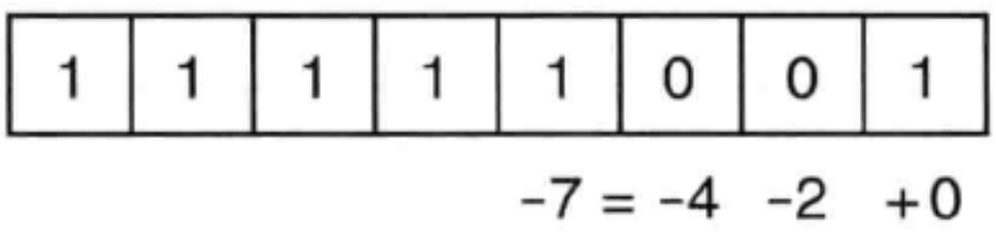


图3-3 计算负字节的值

所以，1111 1111 1111 1111对应于-1，1111 1111 1111 1001对应于-7，而1000 0000 0000 0000对应于16位整数能够容纳的最小负值。

3.6.1 移位操作符

有时要将一个数的二进制值向右或向左移位。左移时，所有位都向左移动由操作符右侧的操作数指定的位数。移位后在右边留下的空位由零填充。右移位操作符原理相似，只是朝相反方向移位；但如果是负数，左侧填充1而非0。两个移位操作符是>>和<<，分别称为右移位和左移位操作符。除此之外，还有复合移位和赋值操作符<<=和>>=。

例如int值-7，其二进制形式为1111 1111 1111 1111 1111 1111 1111 1001。代码清单3-37使其右移2个位置。

代码清单3-37 使用右移位操作符

```
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes
               // 11111111111111111111111111111110
// Write out "x is -2."
System.Console.WriteLine("x = {0}.", x);
```

输出3-17展示了代码清单3-36的结果。

输出3-17

```
x = -2.
```

向右移位时，最右侧的位在边界处“离开”，左边的负数位标识符右移两个位置，腾出来的空白位置用1填充。最终结果是-2。

虽然传说 x << 2比x * 4运算要快，但不要将移位操作符用于乘除法。20世纪70年代的一些C编译器可能确实如此，但现代编译器和微处理器都对算术运算进行了完美的优化。通过移位进行乘除法运算会令人迷惑，而且，假如维护代码的人忘记了移位操作符的优先级低于算术操作符，还很容易造成错误。

3.6.2 按位操作符

有时需要对两个操作数执行逐位的逻辑运算，比如AND、OR和XOR等，这分别是使用&、|和^操作符来实现的。

初学者主题：理解逻辑操作符

假定有两个数，如图3-4所示，按位操作符从最左边的位开始逐位进行逻辑运算，直到最右边的位为止。值1被视为true，值0被视为false。

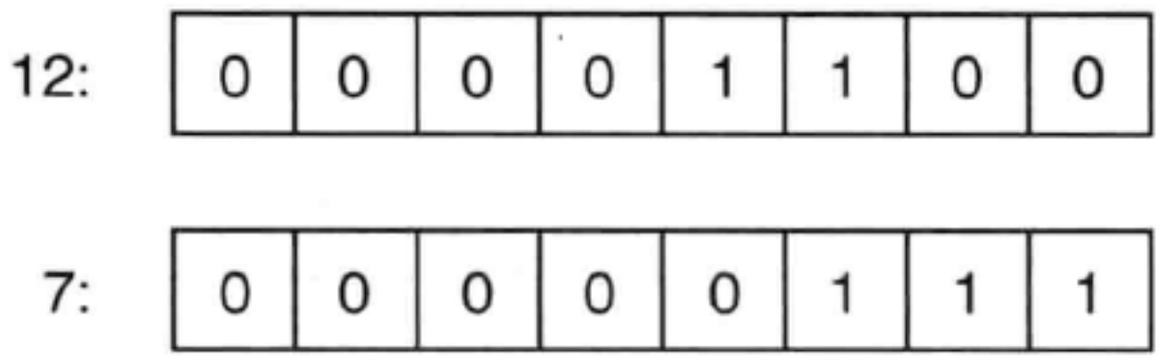


图3-4 12和7的二进制形式

所以，如果对图3-4的两个值执行按位AND运算，就会逐位比较第一个操作数（12）和第二个操作数（7），得到二进制值00000100，也就是十进制的4。另外，这两个值的按位OR运算结果是00001111，也就是十进制的15。XOR结果是00001011，也就是十进制的11。

代码清单3-38演示了如何使用这些按位操作符，结果如输出3-18所示。

代码清单3-38 使用按位操作符

```
byte and, or, xor;
and = 12 & 7;   // and = 4
or = 12 | 7;    // or = 15
xor = 12 ^ 7;   // xor = 11
System.Console.WriteLine(
    "and = {0} \nor = {1}\nxor = {2}",
    and, or, xor);
```

输出3-18

```
and = 4
or = 15
xor = 11
```

在代码清单3-38中，值7称为**掩码**（mask），作用是通过特定的操作符表达式，公开（expose）或消除（eliminate）第一个操作数中特定的位。注意，和AND（&&）操作符不同，&操作符总是两边求值，即使左边为false。类似地，OR操作符的|版本也不进行“短路求值”。即使左边的操作数为true，它也会对两边的操作数都进行求值。总之，AND和OR操作符的按位版本不进行“短路求值”。

为了将一个数转换成二进制形式，需要遍历它的每一位。代码清单3-39展示了如何将ulong整数转换成二进制形式的字符串，输出3-19展示了结果。

代码清单3-39 获取二进制形式的字符串表示

```
public class BinaryConverter
{
  public static void Main()
  {
      const int size = 64;
      ulong value;
      char bit;

      System.Console.Write ("Enter an integer: ");
      // Use long.Parse() so as to support negative numbers
      // Assumes unchecked assignment to ulong.
      value = (ulong)long.Parse(System.Console.ReadLine());

      // Set initial mask to 100....
      ulong mask = 1UL << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) != 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
      System.Console.WriteLine();
  }
}
```

输出3-19

```
Enter an integer: 42
00000000000000000000000000000000000000000000000000000000101010
```

注意，每次for循环（稍后就会讨论）都使用右移位赋值操作符创建与变量value中的每一位对应的掩码。使用按位操作符&，可以判断一个特定的位是否已被设置（是否设为1）。如果掩码测试生成非零结果，就将1写到控制台，否则写0。采用这种方式，最终就可以创建反映一个无符号长整数的二进制形式的输出。

还要注意，(mask & value) != 0中的圆括号是必需的，因为不相等测试的优先级高于AND操作符。如果不显式添加圆括号，这个表达式就等价于mask & (value != 0)，这没有任何意义，&的左操作数是一个ulong，而右操作数是一个bool。

3.6.3 按位赋值操作符

一点儿也不奇怪，按位操作符可以和赋值操作符合并，即&=、|=和^=。例如，可以让变量与一个数进行OR运算，将结果赋回给初始变量，代码清单3-40对此进行了演示。

代码清单3-40 使用逻辑赋值操作符

```
byte and = 12, or = 12, xor = 12;
and &= 7;   // and = 4
```

```
or  |= 7;   // or = 15
xor ^= 7;   // xor = 11
System.Console.WriteLine(
    "and = {0} \nor = {1}\nxor = {2}",
    and, or, xor);
```

输出3-20展示了代码清单3-40的结果。

输出3–20

```
and = 4
or = 15
xor = 11
```

使用fields &=mask这样的表达式将位映射与掩码合并，会从fields中消除mask中没有设置的位。相反，fields &= ~mask从fields中消除mask中已设置的位。

3.6.4 按位取反操作符

按位取反操作符（~）是对操作数的每一位取反，操作数可以是 int、uint、long 和 ulong 类型。例如，~1返回1111 1111 1111 1111 1111 1111 1111 1110，而~(1<<31)返回0111 1111 1111 1111 1111 1111 1111 1111。

3.7 控制流语句（续）

更详细地探讨了布尔表达式之后，就可以更清楚地描述C#支持的控制流语句。有经验的程序员已经熟悉了其中的许多语句，所以可以快速浏览本节的内容，找出C#特有的信息。尤其关注foreach循环，它对许多程序员来说是新的。

3.7.1 while和do/while循环

到目前为止学习的都是如何编写只执行一遍的程序。但是，计算机的关键优势之一是能多次执行相同的操作。为此，需要创建指令循环。本节讨论的第一个指令循环是while循环，因为它是最简单的条件循环。while语句的常规形式如下：

```
while (condition)
  statement
```

条件（condition）必须是布尔表达式，只要它求值为true，作为循环主体的语句（statement）就会反复执行。如果条件求值为false，代码执行就跳过循环主体，而从它之后的那一条语句执行。注意，循环主体会一直执行，即使它导致条件变成false。除非“在循环顶部”重新对条件求值，而且求值结果是false，否则循环不会退出。代码清单3-41用一个斐波那契计算器演示了while语句的用法。

代码清单3–41 while循环示例

```
class FibonacciCalculator
{
  static void Main()
  {
      decimal current;
      decimal previous;
      decimal temp;
```

```
        decimal input;

        System.Console.Write("Enter a positive integer:");

        // decimal.Parse convert the ReadLine to a decimal.
        input = decimal.Parse(System.Console.ReadLine());

        // Initialize current and previous to 1, the first
        // two numbers in the Fibonacci series.
        current = previous = 1;

        // While the current Fibonacci number in the series is
        // less than the value input by the user.
        while(current <= input)
        {
            temp = current;
            current = previous + current;
            previous = temp; // Executes even if previous
            // statement caused current to exceed input
        }

        System.Console.WriteLine(
          "The Fibonacci number following this is {0}",
          current);
    }
}
```

斐波那契数（Fibonacci number）是**斐波那契数列**（Fibonacci series）的成员，这个数列中的所有数都是数列中前两个数之和。数列最开头两个数是1和1。代码清单3-41中提示用户输入一个整数，接着使用while循环发现比用户输入的数大的第一个斐波那契数。

初学者主题：何时使用while循环

本章剩余的部分会讲到造成代码块反复执行的其他循环结构。术语*循环*（loop）指的是while结构中执行的语句（通常是一个代码块）。这是因为在达成退出条件之前，代码会一直循环执行。应该明白在什么时候选择什么循环结构。如果希望条件为true就一直执行某个操作，应该使用while结构。for主要用于重复次数已知的循环，比如从0～*n*的计数。do/while类似于while循环，区别在于它至少会循环一次。

do/while循环与while循环非常相似，只是它最适合需要循环1～*n*次的情况，而且*n*在循环开始前无法确定。do/while循环的一个典型应用就是反复提示用户输入。代码清单3-42是从井字棋程序中提取出来的。

代码清单3-42 do/while循环示例

```
// Repeatedly request player to move until he
// enter a valid position on the board.
bool valid;
do
{
  valid = false;

  // Request a move from the current player.
  System.Console.Write(
    "\nPlayer {0}: Enter move:", currentPlayer);
  input = System.Console.ReadLine();
```

```
        // Check the current player's input.
        // ...

    } while (!valid);
```

代码清单3-42在每次**迭代**（iteration）或循环开始的时候将valid设为false。接着，提示并获取用户输入的数。在代码清单3-42中，虽然这部分被省略了，但接下来的操作是检查输入是否正确。如果正确，就将true赋给valid。由于代码使用do/while语句，而不是while语句，所以至少提示用户输入一次。

do/while循环的常规形式如下：

```
do
  statement
while (condition);
```

和所有控制流语句一样，通常用代码块作为循环主体，以便循环执行多个语句。但是，也可将单一语句作为循环主体（标签语句和局部变量声明除外）。

3.7.2 for循环

for循环反复执行代码块，直到满足指定条件。区别在于，for循环有一套内建的语法规定了如何初始化、递增以及测试一个计数器的值。该计数器称为**循环变量**。由于递增操作在循环语法中有一席之地，所以递增和递减操作符经常作为for循环的一部分使用。

代码清单3-43展示了如何使用for循环显示整数的二进制形式。输出3-21展示了结果。

代码清单3-43 使用for循环

```
public class BinaryConverter
{
  public static void Main()
  {
      const int size = 64;
      ulong value;
      char bit;

      System.Console.Write ("Enter an integer: ");
      // Use long.Parse() so as to support negative numbers
      // Assumes unchecked assignment to ulong.
      value = (ulong)long.Parse(System.Console.ReadLine());

      // Set initial mask to 100....
      ulong mask = 1UL << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) > 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
  }
}
```

输出3-21

```
Enter an integer: -42
1111111111111111111111111111111111111111111111111111111111010110
```

代码清单3-43将一个位掩码应用于由用户输入的数的每一位，总共要应用64次。for循环头包含三个部分。第一部分声明并初始化变量count，第二部分描述for循环主体的执行条件，第三部分描述如何更新循环变量。for循环的常规形式如下。

```
for (initial ; condition ; loop)
  statement
```

下面对for循环的各个部分进行解释。

- initial（初始化）部分执行首次循环之前的初始化操作。在代码清单3-43中，它声明并初始化count变量。initial表达式不一定非要声明新变量。例如，可以在之前声明好变量，在for循环中只是初始化它。也可以完全省略该部分。这里声明的变量的作用域限于for语句头部和主体。
- for循环的condition（条件）部分指定循环结束条件。一旦该条件为false，就会终止循环，这和while循环是相似的。当且仅当条件求值为true，才会执行for循环主体。在上例中，循环在count大于或等于64的时候退出。
- loop表达式会在每次循环之后求值。在上例中，count++会在mask右移位（mask >>= 1）之后，但在对条件进行求值之前执行。第64次循环的时候，count会递增到64，造成条件变成false，因而终止循环。
- for循环的statement部分是在条件表达式为true的时候执行的代码。

在代码清单3-42中，for循环的执行步骤可以用以下伪代码来表示。

（1）声明count并将其初始化为0。

（2）如果count小于64，到步骤3；否则到步骤7。

（3）计算bit并显示它。

（4）对mask执行右移位。

（5）count递增1。

（6）回步骤2。

（7）继续执行循环主体之后的语句。

for语句头部的3个部分都可以省略。for(;;){ ... }完全有效，只要有办法从循环中退出以避免无限循环（如果条件部分缺失，则默认其值为常量true）。

initial和loop部分还支持多个循环变量，如代码清单3-44所示。

代码清单3-44 使用多个表达式的for循环

```
for(int x=0, y=5; ((x<=5) && (y>=0)); y--, x++)
{
  System.Console.Write("{0}{1}{2}\t",
      x, (x>y? '>' : '<'), y);
}
```

代码清单3-44的结果如输出3-22所示。

输出3-22

```
0<5     1<4     2<3     3>2     4>1     5>0
```

在这个例子中，initial部分包含了一个复杂的声明语句，声明并初始化两个循环变量，但它至少像是一个声明了多个局部变量的声明语句。loop部分则非比寻常，因为它是包含以逗号分隔的表达式列表，而非单一的表达式。

规范

如果发现正在写的for循环包含了复杂条件和多个循环变量，要考虑重构方法，以使控制流更容易理解。

for循环只不过是一种比写while循环更方便的方法。for循环能改写成while循环，如下例所示。

```
{
 initial;
  while(condition)
  {
    statement;
    loop;
  }
}
```

规范

假如事先知道循环次数，而且循环中需要用到控制循环次数的“计数器”，那么要使用for循环。

假如事先不知道循环次数，而且不需要计数器，那么要使用while循环。

3.7.3 foreach循环

C#语言的最后一个循环语句是foreach，它遍历数据项集合，设置循环变量来依次表示其中每一项。循环主体可对数据项执行指定操作。foreach循环的特点是每一项只被遍历一次：不会像其他循环那样出现计数错误，也不可能越过集合边界。

foreach语句的常规形式如下：

```
foreach(type variable in collection)
  statement
```

下面对foreach语句的各个部分进行解释。

- type为代表collection中每一项的variable声明数据类型。可将类型设为var，编译器将根据集合类型推断数据项的类型。
- variable是只读变量，foreach循环自动将collection中的下一项赋给它。variable的作用域限于循环主体。
- collection是代表多个数据项的表达式，比如数组。
- statement是每次循环都要执行的循环主体。

来看代码清单3-45展示的一个简单的foreach循环。

代码清单3-45 使用foreach循环判断剩余走棋

```
class TicTacToe       // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      // Hardcode initial board as follows
```

```
        // ---+---+---
        //  1 | 2 | 3
        // ---+---+---
        //  4 | 5 | 6
        // ---+---+---
        //  7 | 8 | 9
        // ---+---+---
        char[] cells = {
          '1', '2', '3', '4', '5', '6', '7', '8', '9'
        };

        System.Console.Write(
            "The available moves are as follows: ");

        // Write out the initial available moves
        foreach (char cell in cells)
        {
          if (cell != 'O' && cell != 'X')
          {
              System.Console.Write("{0} ", cell);
          }
        }
    }
}
```

输出3-23展示了代码清单3-45的结果。

输出3–23

```
The available moves are as follows: 1 2 3 4 5 6 7 8 9
```

执行到foreach语句时，将cells数组中的第一项，也就是值'1'赋给cell变量。然后执行foreach循环主体。if语句判断cell的值是否等于'O'或'X'，如果两者都不是，就在控制台上输出cell的值。下次循环将下个数组值赋给cell，以此类推。

必须记住，foreach循环期间禁止修改循环变量（这里是cell）。另外，循环变量在C# 5中的行为稍微有别于之前的版本。只有当循环主体中通过λ表达式或匿名方法使用了循环变量，才会体现出差别。详情参见第12章。

初学者主题：何时使用switch语句

有时需要在几个连续的if语句中比较同一个值，如代码清单3-46的input变量所示。

代码清单3–46 用if语句检查玩家的输入

```
// ...

bool valid = false;

// Check the current player's input.
if( (input == "1") ||
  (input == "2") ||
  (input == "3") ||
  (input == "4") ||
  (input == "5") ||
  (input == "6") ||
  (input == "7") ||
  (input == "8") ||
  (input == "9") )
{
```

```
        // Save/move as the player directed.
        // ...

        valid = true;
    }
    else if( (input == "") || (input == "quit") )
    {
        valid = true;
    }
    else
    {
        System.Console.WriteLine(
          "\nERROR:  Enter a value from 1-9. "
          + "Push ENTER to quit");
    }

    // ...
```

这段代码验证用户输入的文本，确定它是一步有效的井字棋走棋。例如，假定input的值是9，那么程序不得不执行9次求值。显然，更好的思路是只在一次求值之后就跳转到正确的代码。在这种情况下，应该使用switch语句。

3.7.4 switch语句

将一个值和许多不同的常量值比较时，switch语句比if语句更容易理解。switch语句的常规形式如下：

```
switch(expression)
{
    case constant:
        statements
    default:
        statements
}
```

下面对switch语句的各个部分进行解释。

- expression是要与不同常量比较的值。该表达式的类型决定了switch的“主导类型”（governing type）。允许的主导数据类型包括bool、sbyte、byte、short、ushort、int、uint、long、ulong、char、任何枚举（enum）类型（详情参见第8章）、上述所有值类型的可空类型以及string。
- constant是与主导类型兼容的任何常量表达式。
- 一个或多个case标签（或default标签），后跟一个或多个语句，这称为一个switch小节。上述常规形式包含了两个switch小节。代码清单3-47的switch语句则包含3个。
- statement是在expression的值等于某个标签指定的constant值时执行的一个或多个语句。这组语句的结束点必须是“不可到达”的[①]。换言之，不能“直通”或“贯穿”到下一个switch小节。所以，最后一个语句通常是跳转语句，比如break、return或goto。

① C#语言规范对结束点和可到达性的解释是这样的：“每条语句都有一个结束点（end point）。直观地讲，语句的结束点是紧跟在语句后面的那个位置。复合语句（包含嵌入语句的语句）的执行规则规定了当控制到达一个嵌入语句的结束点时所采取的操作。例如，当控制到达块中某个语句的结束点时，控制就转到该块中的下一个语句。如果执行流程可能到达某个语句，则称该语句是可到达的（reachable）。相反，如果某个语句不可能被执行，则称该语句是不可到达的（unreachable）。”——译者注

规范

不要使用continue作为跳转语句退出switch小节。如果switch语句是在一个循环中使用的，这样写是合法的。但是，这样做很容易对之后的switch小节中出现的break语句的意义感到迷惑。

switch语句至少要有一个switch小节；switch(x){}虽然合法，但会产生一个警告。另外，虽然在之前的规范中提到，在一般情况下应避免省略大括号，但有一个例外，就是要省略case和break语句的大括号，因为它们的作用是指示一个块的开始与结束。

代码清单3-47使用了一个switch语句，它在语义上等价于代码清单3-46的一系列if语句。

代码清单3-47 将if语句替换成switch语句

```
static bool ValidateAndMove(
  int[] playerPositions, int currentPlayer, string input)
{
  bool valid = false;

  // Check the current player's input.
  switch (input)
  {
    case "1" :
    case "2" :
    case "3" :
    case "4" :
    case "5" :
    case "6" :
    case "7" :
    case "8" :
    case "9" :
      // Save/move as the player directed.
      ...
      valid = true;
      break;

    case "" :
    case "quit" :
      valid = true;
      break;
    default :
      // If none of the other case statements
      // is encountered then the text is invalid.
      System.Console.WriteLine(
        "\nERROR:  Enter a value from 1-9. "
        + "Push ENTER to quit");
      break;
  }

  return valid;
}
```

在代码清单3-47中，input是要测试的表达式。由于input是字符串，所以主导类型是string。如果input的值是"1"、"2"、……、"9"，那么走棋是有效的（valid = true），

然后更改相应的单元格，使之与当前用户的标记（X或O）匹配。一旦遇到break语句，就会立即跳转到switch语句之后的语句。

下一个switch小节描述如何处理空字符串""或者"quit"。如果input等于这两个值之一，就将valid设为true。如果没有与测试表达式匹配的其他case标签，就执行default的switch小节。

switch语句有几件事情需要注意。

- 无switch小节的switch语句会产生编译器警告，但语句仍能通过编译。
- switch小节可以以任意顺序出现，default小节不一定非要出现在switch语句的最后。事实上，default的switch小节完全可以省略；它是可选的。
- C#要求每个switch小节（包括最后一个小节）的结束点“不可到达”。这意味着switch小节通常以break、return或goto结尾。

语言对比：C++——switch语句贯穿

在C++中，如果switch小节不以跳转语句结尾，那么控制会“贯穿”至下一个switch小节，并执行其中的代码。由于非预期的贯穿是C++中常见的错误，所以C#不允许控制从一个switch小节自然贯穿到下一个switch小节。C#的设计者认为，这样可以更好地防止这种常见的bug的源头并增强代码的可读性。如果希望switch小节执行另一个switch小节中的语句，可以显式使用goto语句来实现，详情参见3.8.3节。

3.8 跳转语句

循环的执行路径可以改变。事实上，可用跳转语句退出循环，或者跳过一次循环的剩余部分并开始下一次循环——即使循环条件当前仍然为true。本节介绍了让执行路径从一个位置跳转到另一个位置的几种方式。

3.8.1 break语句

C#使用break语句退出循环或者switch语句。任何时候遇到break语句，控制都会立即离开循环或switch。代码清单3-48演示了井字棋程序的foreach循环。

代码清单3-48 一旦发现赢家，就用break跳出循环

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      int winner=0;
      // Stores locations each player has moved.
      int[] playerPositions = {0,0};

      // Hardcoded board position
      //  X | 2 | O
      // ---+---+---
      //  O | O | 6
      // ---+---+---
      //  X | X | X
```

```
            playerPositions[0] = 449;
            playerPositions[1] = 28;

            // Determine if there is a winner
            int[] winningMasks = {
                  7, 56, 448, 73, 146, 292, 84, 273 };

            // Iterate through each winning mask to determine
            // if there is a winner.
            foreach (int mask in winningMasks)
            {
                if ((mask & playerPositions[0]) == mask)
                {
                    winner = 1;
                    break;
                }
                else if ((mask & playerPositions[1]) == mask)
                {
                    winner = 2;
                    break;
                }
            }

            System.Console.WriteLine(
                "Player {0} was the winner", winner);
        }
    }
```

输出3-24展示了代码清单3-48的结果。

输出3-24

```
Player 1 was the winner
```

代码清单3-48在发现一个玩家取得胜利后，就会使用break语句。break语句强迫它所在的循环（或者switch语句）终止执行，并转移到循环外部的下一个语句开始执行。就本例而言，假如位比较运算返回true（当前棋盘上已经有玩家取胜），就执行break语句，跳出当前foreach循环，并显示赢家。

初学者主题：用按位操作符处理棋子分布

井字棋程序示例（附录B）使用按位操作符判断哪个玩家取得胜利。首先，代码将每个玩家走的棋子位置保存到名为playerPositions的位映射中（这里使用的是一个数组，使两个玩家的位置都可以保存下来）。

最开始，两个playerPositions都是0。玩家每次走棋时，与落棋位置对应的位都设为1。例如，假定玩家选择在单元格3这个位置落棋，那么shifter就设为3-1。之所以要减去1，是因为C#数组是基于0的，应该将0而不是1视为第一个位置。接着，代码使用移位操作符设置position，也就是与单元格3对应的位，具体的代码是0000000000000001 << shifter。其中，shifter当前的值是2。最后，它将当前玩家的playerPositions设为0000000000000100（要减去1，因为索引是基于0的）。代码清单3-49使用“|=”将之前的走棋和当前走棋合并到一起。

代码清单3-49 设置与玩家每次走棋对应的位

```
int shifter;  // The number of places to shift
              // over in order to set a bit.
int position;  // The bit which is to be set.

// int.Parse() converts "input" to an integer.
// "int.Parse(input) - 1" because arrays
// are zero-based.
shifter = int.Parse(input) - 1;

// Shift mask of 00000000000000000000000000000001
// over by cellLocations.
position = 1 << shifter;

// Take the current player cells and OR them to set the
// new position as well.
// Since currentPlayer is either 1 or 2,
// subtract one to use currentPlayer as an
// index in a 0-based array.
playerPositions[currentPlayer-1] |= position;
```

之后，就可以遍历与棋盘上的取胜布局对应的每一个掩码，从而判断当前玩家是否得到了一个取胜布局，就像代码清单3-48展示的那样。

3.8.2 continue语句

循环主体可能有很多语句。在符合特定条件的时候，如果想中断当前循环，放弃执行剩余的语句，就可以使用continue语句跳到当前循环的末尾，并开始下一次循环。C#的continue语句允许退出当前循环（不管还剩下多少语句没有执行），并跳到循环条件。如果循环条件仍然为true，循环将继续执行。

代码清单3-50使用continue语句只显示电子邮件地址的域名部分。输出3-25展示了代码清单3-50的结果。

代码清单3-50 判断电子邮件地址的域名部分

```
class EmailDomain
{
  static void Main()
  {
      string email;
      bool insideDomain = false;
      System.Console.WriteLine("Enter an email address: ");

email = System.Console.ReadLine();

System.Console.Write("The email domain is: ");

// Iterate through each letter in the email address.
foreach (char letter in email)
{
    if (!insideDomain)
    {
        if (letter == '@')
        {
            insideDomain = true;
```

```
            }
            continue;
        }

        System.Console.Write(letter);
    }
  }
}
```

输出3-25

```
Enter an email address:
mark@dotnetprogramming.com
The email domain is: dotnetprogramming.com
```

在代码清单3-50中，在遇到电子邮件地址的域名部分之前，需要一直使用continue语句来跳转到电子邮件地址的下一个字符。

一般都可以使用if语句代替continue语句，这样做还能增强可读性。continue语句的问题在于，它在一次循环中提供了多个出口，从而影响了可读性。代码清单3-51重写了上面的例子，将continue语句替换成if/else构造，从而改善了可读性。

代码清单3-51 将continue替换成if语句

```
foreach (char letter in email)
{
  if (insideDomain)
  {
      System.Console.Write(letter);
  }
  else
  {
      if (letter == '@')
      {
          insideDomain = true;
      }
  }
}
```

3.8.3 goto语句

早期编程语言不像C#这样的现代语言那样具备完善的“结构化”控制流，它们要依赖简单的条件分支（if）和无条件分支（goto）语句来满足控制流的需求。这样得到的程序通常是难以理解的。许多资深程序员觉得goto语句在C#中继续存在很反常。然而，C#确实支持goto，而且只能利用goto在switch语句中实现贯穿。在代码清单3-52中，如果设置了/out选项，就使用goto语句跳转到default，/f选项的处理与此相似。

代码清单3-52 演示带有goto的switch语句

```
// ...
static void Main(string[] args)
{
  bool isOutputSet = false;
  bool isFiltered = false;

  foreach (string option in args)
```

```
{
    switch (option)
    {
        case "/out":
            isOutputSet = true;
            isFiltered = false;
            goto default;
        case "/f":
            isFiltered = true;
            isRecursive = false;
            goto default;
        default:
            if (isRecursive)
            {
                // Recurse down the hierarchy
                // ...

            }
            else if (isFiltered)
            {
                // Add option to list of filters.
                // ...
            }
            break;
    }

}

// ...
}
```

输出3-26演示了如何执行代码清单3-52的代码。

输出3-26

```
C:\SAMPLES>Generate /out fizbottle.bin /f "*.xml" "*.wsdl"
```

如果要跳转到标签不是default的其他switch小节，可以使用goto case *constant*的语法；其中*constant*是要跳转到的目标case标签中指定的常量。如果要跳转到没有和switch小节关联的语句，需要在目标语句前添加标识符和冒号，并在goto语句中使用该标识符指示跳转。例如，可以编写一个加了标签的语句myLabel:Console.WriteLine();，然后用goto myLabel;跳转到那个标签语句。幸好，C#禁止通过goto跳转到代码块内部。只能用goto在代码块内部跳转，或者跳到一个封闭的代码块。通过这个限制，C#避免了在其他语言中可能遇到的大多数滥用goto的情况。

尽管C#对goto做了一些改善，但一般还是会认为使用goto是不“优雅”的，难以理解，而且造成结构很差的代码。要多次或者在不同情况下执行某个代码小节，要么使用循环，要么将代码抽取出来重构为一个方法。

规范

避免使用goto。

3.9 C#预处理指令

控制流语句中的条件表达式在运行时求值。相反，C#预处理器在编译时调用。预处理指令告诉C#编译器要编译哪些代码，并指出如何处理代码中的特定错误和警告。C#预处理指令还可告诉C#编辑器有关代码组织的信息。

语言对比：C++ —— 预处理

C和C++风格的语言用**预处理器**（preprocessor）对代码进行整理，根据特殊的记号来执行特殊的操作。预处理器指令通常告诉编译器如何编译文件中的代码，而并不参与实际的编译过程。相反，C#编译器将预处理器指令作为对源代码执行的常规词法分析的一部分。其结果就是，C#不支持更高级的预处理器宏，它最多只允许定义常量。事实上，“预处理器”一词在C++中显得很贴切，但在C#中就属于用词不当。

每个预处理指令都以#开头，而且必须在一行中写完。换行符（而不是分号）标志着预处理指令的结束。

表3-4总结了所有预处理指令。

表3-4 预处理指令

语句或表达式	常规语法结构	示 例
#if指令	`#if preprocessor-expression` `  code` `#endif`	`#if CSHARP2` `  Console.Clear();` `#endif`
#elif指令	`#if preprocessor-expression1` `  code` `#elif preprocessor-expression2` `  code` `#endif`	`#if LINUX` `...` `#elif WINDOWS` `...` `#endif`
#else指令	`#if` `  code` `#else` `  code` `#endif`	`#if CSHARP1` `...` `#else` `...` `#endif`
#define指令	`#define conditional-symbol`	`#define CSHARP2`
#undef指令	`#undef conditional-symbol`	`#undef CSHARP2`
#error指令	`#error preproc-message`	`#error Buggy implementation`
#warning指令	`#warning preproc-message`	`#warning Needs code review`
#pragma指令	`#pragma warning`	`#pragma warning disable 1030`
#line指令	`#line org-line new-line`	`#line 467` `"TicTacToe.cs"` `...` `#line default`
	`#line default`	
#region指令	`#region pre-proc-message` `  code` `#endregion`	`#region Methods` `  ...` `#endregion`

3.9.1 排除和包含代码

或许最常用的预处理指令就是控制何时以及如何包含代码的指令。例如，要使代码兼容C# 2.0（及以后版本）和C# 1.2的编译器，可以使用一个预处理指令指示在遇到1.2编译器时排除C# 2.0特有的代码。井字棋程序示例和代码清单3-53对此进行了演示。

代码清单3-53 遇到C# 1.x编译器就排除C# 2.0代码

```
#if CSHARP2
System.Console.Clear();
#endif
```

本例调用了System.Console.Clear()方法，这是2.0或更高版本的CLI才支持的方法。使用#if和#endif预处理指令，这行代码只有在定义了预处理符号CSHARP2的前提下才会编译。

预处理指令的另一个应用是处理不同平台之间的差异，比如用WINDOWS和LINUX #if指令将Windows和Linux特有的API包围起来。开发人员经常用这些指令取代多行注释（/*...*/），因为它们更容易通过定义恰当的符号或者通过搜索/替换来移除。预处理指令最后一个常见的用途是调试。如果用#if DEBUG指令将调试代码包围起来，那么大多数IDE都能在发布版中移除这些代码。IDE默认将DEBUG符号用于调试编译，将RELEASE符号用于发布版。

为了处理else-if条件，可以在#if指令中使用#elif指令，而不是创建两个完全独立的#if块，如代码清单3-54所示。

代码清单3-54 使用#if、#elif和#endif指令

```
#if LINUX
...
#elif WINDOWS
...
#endif
```

3.9.2 定义预处理符号

可以采取两种方式定义预处理符号。第一种方式是使用#define指令，如代码清单3-55所示。

代码清单3-55 #define的例子

```
#define CSHARP2
```

第二种方式是在编译时使用define选项，如输出3-27所示。

输出3-27

```
>csc.exe /define:CSHARP2 TicTacToe.cs
```

输出3-28展示了用Mono编译器如何实现与上面相同的功能。

输出3-28

```
>mcs.exe -define:CSHARP2 TicTacToe.cs
```

多个定义以分号分隔。使用define编译器选项的优点是不需要更改源代码，所以可用相同的源代码文件生成两套不同的二进制程序。

要取消符号定义，可以采取和使用#define相同的方式来使用#undef指令。

3.9.3 生成错误和警告

有时想要标记代码中潜在的问题。为此，可以插入#error和#warning指令来分别生成错误和警告消息。代码清单3-56使用井字棋程序警告代码无法防止玩家多次输入同一步棋。输出3-29展示了结果。

代码清单3-56 用#warning定义警告

```
#warning    "Same move allowed multiple times."
```

输出3-29

```
Performing main compilation...
...\tictactoe.cs(471,16): warning CS1030: #warning: '"Same move allowed multiple
times."'

Build complete -- 0 errors, 1 warnings
```

包含#warning指令后，编译器会主动发出警告，如输出3-29所示。利用这种警告，可以标记出代码中潜在的bug和可以改进的部分。它是提醒开发者任务尚未完结的好帮手。

3.9.4 关闭警告消息

警告可以指出代码中可能存在的问题，所以非常有用。然而，有的警告可以合理地忽略，所以有必要显示地关闭它们。C# 2.0以及之后的编译器提供了预处理指令#pragma来关闭或还原警告，如代码清单3-57所示。

代码清单3-57 使用预处理指令#pragma禁用#warning指令

```
#pragma warning disable  1030
```

注意，编译器输出时会在警告编号前附加CS前缀。但在用#pragma禁用警告时不要添加这个前缀。当没有预处理器命令时，这个对应警告错误的编号会被编译器删除。

重新启用警告仍然是使用#pragma指令，只是在warning之后添加restore选项，如代码清单3-58所示。

代码清单3-58 使用预处理指令#pragma还原警告

```
#pragma warning restore  1030
```

上述两条指令正好可以将一个特定的代码块包围起来——前提是已知该警告不适用于这个代码块。

经常被禁用的警告是CS1591。使用/doc编译器选项生成XML文档，但是并未注释程序中的所有public项，就会显示该警告。

3.9.5 nowarn:<warn list>选项

除了#pragma指令，C#编译器通常还支持nowarn:<warn list>选项。它可以获得与

#pragma相同的结果，只是不用把它加进源代码，而是把它作为编译器选项使用。除此之外，nowarn选项会影响整个编译过程，而#pragma指令只影响该指令所在的那个文件。例如在输出3-30中，我们在命令行上关闭了CS1591警告。

输出3-30

```
> csc /doc:generate.xml /nowarn:1591 /out:generate.exe Program.cs
```

3.9.6 指定行号

利用#line指令可以改变C#编译器在报告错误或警告时显示的行号。这个指令主要由自动生成C#代码的实用程序和设计器使用。在代码清单3-59中，真实行号显示在最左侧。

代码清单3-59 #line预处理指令

```
124         #line 113 "TicTacToe.cs"
125         #warning "Same move allowed multiple times."
126         #line default
```

在上例中，使用#line指令后，编译器会将实际发生在125行的警告报告在113行上发生，如输出3-31所示。

输出3-31

```
Performing main compilation...
...\tictactoe.cs(113,18): warning CS1030: #warning: '"Same move allowed multiple
times."'

Build complete -- 0 errors, 1 warnings
```

在#line指令后添加default，会反转之前的所有#line的效果，并指示编译器报告真实的行号，而不是之前使用#line指定的行号。

3.9.7 可视编辑器提示

C#提供了只有在可视代码编辑器中才有用的两个预处理指令，也就是#region和#endregion。像Microsoft Visual Studio .NET IDE这样的可视代码编辑器能够搜索源代码，找到这些指令，并相应地进行处理。C#允许使用#region指令声明代码区域。#region和#endregion必须成对使用，两个指令都可以选择在指令后面跟随一个描述性的字符串。除此之外，还可以将一个区域嵌套到另一个区域中。

代码清单3-60是井字棋程序的例子。

代码清单3-60 #region和#endregion预处理指令

```
...
#region Display Tic-tac-toe Board

#if CSHARP2
  System.Console.Clear();
#endif

// Display the current board;
border = 0;   //  set the first border (border[0] = "|")
```

```
// Display the top line of dashes.
// ("\n---+---+---\n")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
  // Write out a cell value and the border that comes after it.
  System.Console.Write(" {0} {1}", cell, borders[border]);

  // Increment to the next border;
  border++;
  // Reset border to 0 if it is 3.
  if (border == 3)
  {
      border = 0;
  }
}
#endregion Display Tic-tac-toe Board
...
```

下面的例子演示了在Microsoft Visual Studio .NET中是如何使用这些预处理指令的。Visual Studio .NET会检查上述代码，并提供树形控件来展开和折叠由#region和#endregion指令界定的代码区域（在代码编辑器窗口的左侧），如图3-5所示。

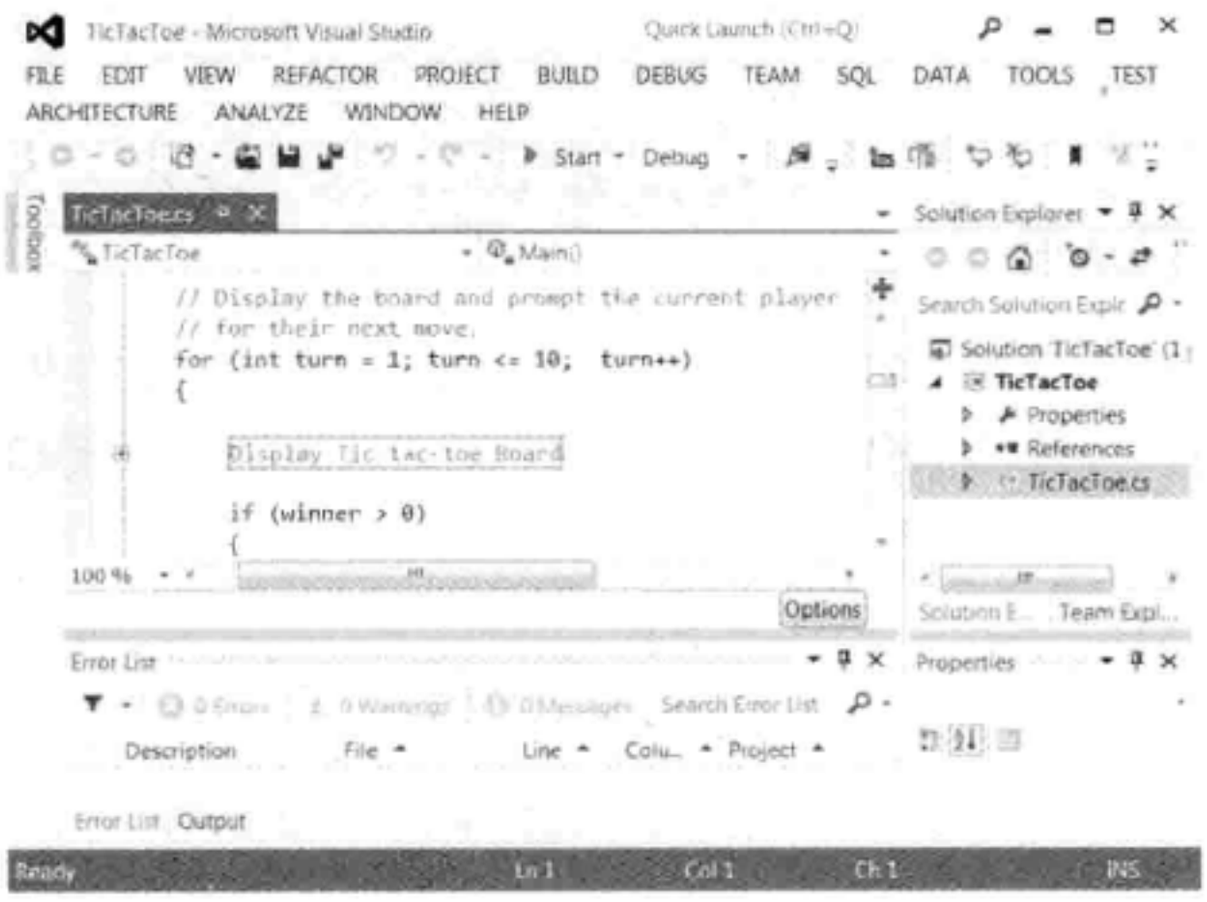


图3-5 Microsoft Visual Studio中折叠起来的区域

3.10 小结

本章首先介绍了C#赋值和算术操作符。接着讲解了如何使用操作符和const关键字声明常量表达式。然而，并没有按顺序讲解所有C#操作符。在讨论关系和逻辑比较操作符之前，先介绍了if语句以及代码块、作用域等重要概念。最后讨论的是按位操作符，重点强调了掩码的用法。还讨论了其他控制流语句，比如循环、switch和goto。本章最后讨论了C#预处理指令。

本章早些时候已讨论了操作符优先级，但表3-5的总结最全面，其中包括几个尚未讲到的操作符。

表3-5 操作符优先级*

类 别	运 算 符
主要	x.y f(x) a[x] x++ x-- new typeof(T) checked(x) unchecked(x) default(T) delegate{} ()
一元	+ - ! ~ ++x --x (T)x
乘	* / %
加	+ -
移位	<< >>
关系和类型测试	< > <= >= is as
相等性	== !=
逻辑AND	&
逻辑XOR	^
逻辑OR	|
条件AND	&&
条件OR	||
空接合	??
条件	?:
赋值	= *= /= %= += -= <<= >>= &= ^= |= =>

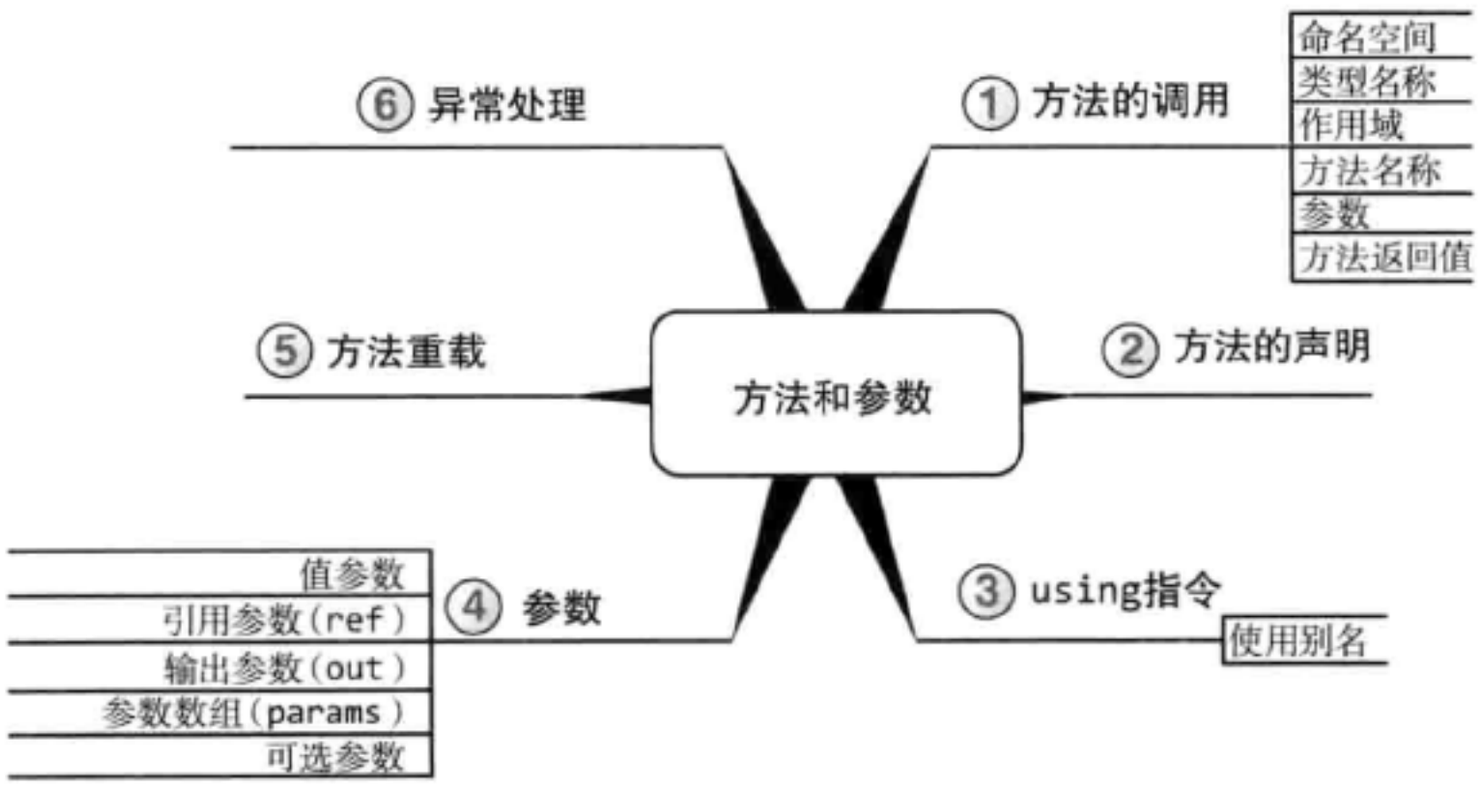
* 各行优先级从高到低排列。

要复习第1章至第3章的内容，或许最好的办法就是将附录B的井字棋程序彻底搞清楚。通过研究该程序，可以慢慢领悟如何将自己学到的东西合并成完整的程序。

第4章

方法和参数

基于目前学到的C#编程知识，应该能写一些简单、直观的程序，它们由一组语句构成，类似于20世纪70年代的那些程序。编程技术自20世纪70年代以来有了长足进步，随着程序变得越来越复杂，需要新的思维模式来管理这种复杂性。“过程式”或“结构化”编程的基本思路就是建立一个结构，在其中填充各式各样的语句，构成一个能独立发挥作用的单元。除此之外，通过结构化编程，可以将数据传给一组语句，并在这些语句执行完毕之后返回需要的结果。



除了方法定义和调用的基础知识，本章还讨论了一些更高级的概念，比如递归、方法重载、可选参数和命名参数。一直到本章末尾，我们讨论的所有方法都是静态方法。“静态”概念将在第5章详细讨论。

其实从第1章的HelloWorld程序开始，就已经接触到了方法。那个例子定义了Main()方

法。本章将更详细地学习方法的创建，其中包括如何用特殊的C#语法（ref和out）让参数向方法传递变量而不是值。本章最后介绍一些基本的错误处理技术。

4.1 方法的调用

初学者主题：什么是方法

到目前为止，在程序中写的所有语句其实都在一个名为Main()的方法内。随着程序逐渐变大，方法很快就会变得难以维护，可读性也会越来越差。

方法（method）组合一系列语句以执行特定操作或计算特定结果。它能够为构成程序的语句提供更好的结构和组织。假定要用Main()方法统计某个目录下源代码的行数。不是在一个巨大的Main()方法中写下所有代码，而是提供更简短的版本，隐藏每个方法的实现细节。代码清单4-1展示了一个例子。

代码清单4-1 将语句组合成方法

```
class LineCount
{
  static void Main()
  {
      int lineCount;
      string files;
      DisplayHelpText();
      files = GetFiles();
      lineCount = CountLines(files);
      DisplayLineCount(lineCount);
  }
  // ...
}
```

不是将所有语句都放到Main()中，而是把它们划分到多个方法中。例如，程序首先用一系列System.Console.WriteLine()语句显示帮助文本，这些语句全部放到名为DisplayHelpText()的方法中。类似地，用GetFiles()方法获取要统计行数的文件。最后，调用CountLines()方法实际统计行数，并调用DisplayLineCount()方法显示结果。一眼就能看清楚整个程序的结构，因为方法名清楚地描述了程序的实现步骤。

规范

要为方法名使用动词或动词短语。

方法总是和类型——通常是类——关联。类型将相关的方法分为一组。

方法通过**实参**接收数据，实参由方法的参数或形参[①]定义。参数是**调用者**（发出方法调用的代码）用于向被调用的方法（Write()、WriteLine()、GetFiles()、CountLines()等）传递数据的变量。在代码清单4-1中，files和lineCount分别是传给CountLines()和DisplayLineCount()方法的实参。方法通过**返回值**将数据返回给调用者。在代码清单4-1中，GetFiles()方法调用的返回值被赋给files。

① 以后不需要区分形参和实参的时候，一般以“参数”代之。——译者注

首先重新讨论一下已在第1章讲过的System.Console.Write()、System.Console.WriteLine()和System.Console.ReadLine()方法。这次要从方法调用的角度讨论它们，而不是将重点放在控制台输入和输出技术上面。代码清单4-2展示了这三个方法的应用。

代码清单4-2 一个简单的主法调用

```
class HeyYou
{
  static void Main()
  {
    string firstName;
    string lastName;

    System.Console.WriteLine("Hey you!");

    System.Console.Write("Enter your first name: ");

    firstName = System.Console.ReadLine();
    System.Console.Write("Enter your last name: ");
    lastName = System.Console.ReadLine();
    System.Console.WriteLine("Your full name is {0} {1}.",
        firstName, lastName);
  }
}
```

命名空间 System
类型名称 Console
方法名称 Write
实参 "Enter your first name: "

方法调用由方法名称和实参列表构成。完全限定的方法名称包括命名空间、类型名称和方法名称；每部分以句点符号分隔。稍后会讲到，调用方法时经常只使用方法完全限定名称的一部分。

4.1.1 命名空间

命名空间是一种分类机制，用于组合功能相关的所有类型。命名空间是分级的，级数可以任意，但是很少见到超过6级的命名空间。命名空间层级一般从公司名开始，然后是产品名，最后是功能领域。例如在Microsoft.Win32.Networking中，最外层的命名空间是Microsoft，它包含内层命名空间Win32，后者又包含嵌套更深的Networking命名空间。

命名空间主要用于按照功能领域组织类型，以便更容易地查找和理解它们。除此之外，命名空间还有助于防止类型名称发生冲突。两个都叫Button的类型只要在不同命名空间中，比如System.Web.UI.WebControls.Button和System.Windows.Controls.Button，编译器就能区分它们。

在代码清单4-2中，Console类型在System命名空间中。System命名空间包含了用于执行大量基本编程活动的类型。几乎所有C#程序都要使用System命名空间中的类型。表4-1总结了其他常用命名空间。

表4-1 常见命名空间

命名空间	描 述
System	包含基元类型，以及用于类型转换、数学计算、程序调用以及环境管理的类型
System.Collections	包含用于处理对象集合（比如列表和字典）的类型

续表

命名空间	描　述
System.Collections.Generics	包含使用泛型的强类型集合
System.Data	包含用于数据库处理的类型
System.Drawing	包含在显示设备上绘图和进行图像处理的类型
System.IO	包含用于文件和目录处理的类型，并提供了文件的处理、加载和保存功能
System.Linq	包含使用“语言集成查询”对集合中数据进行查询的类和接口
System.Text	包含用于处理字符串和各种文本编码的类型，并支持不同编码方式之间的转换
System.Text.RegularExpressions	包含用于处理正则表达式的类型
System.Threading	包含用于多线程编程的类型
System.Threading.Tasks	包含基于任务的异步操作的类型
System.Web	包含用于实现浏览器到服务器通信的类型（一般通过HTTP进行），功能用于支持ASP.NET
System.ServiceModel	包WCF包含在服务和客户端应用程序之间收发数据的类型，实现了在.NET和非.NET技术之间的分布式通信
System.Windows	含用WPF创建富用户界面的类型，WPF利用XAML进行声明式UI设计
System.Windows.Forms	包含创建富用户界面以及其中各种组件的类型
System.Xml	为XML处理提供了基于标准的支持

调用方法并非一定要提供命名空间。例如，假定要调用的方法与发出调用的方法在同一个命名空间中，就没有必要指定命名空间。本章稍后会讲解如何利用using指令避免每次调用方法都指定命名空间限定符。

规范

要为命名空间使用Pascal大小写风格。

考虑将源代码的文件目录结构组织成与命名空间的层次结构相匹配的形式。

4.1.2 类型名称

调用静态方法时，如果目标方法和调用者不在同一个类型中[①]（比如从HelloWorld.Main()中调用静态方法Console.WriteLine()），就需要使用类型名称限定符（即Console.）。但和命名空间一样，如果要调用的方法在当前类型中，C#就允许在调用该方法时省略类型名称（代码清单4-4展示了一个例子）。之所以不需要类型名称，是因为编译器能够根据调用位置推导出类型。显然，假如编译器不能进行这样的推导，就必须将类型名称作为方法调用的一部分。

类型本质上是对方法及其相关数据进行组合的一种方式。例如，Console类型包含了常用的Write()、WriteLine()和ReadLine()等方法。所有这些方法都在同一个“组”中，它们都从属于Console类型。

① 或者目标方法不在基类中。

4.1.3 作用域

上一章讲过，一个编程元素的“作用域”是指可以通过它的非限定名称引用到它的区域。对于某个类型中对一个方法的调用，如果这个方法是在这个类型中声明的，那么，对这个方法的调用就不需要使用类型限定符，因为对于整个包容类型，这个方法都“在作用域内”。类似地，一个类型对于声明了它的整个命名空间也都“在作用域内”。所以，在特定命名空间中的某个类型中的方法调用不需要在方法调用名称中指定该命名空间前缀。

4.1.4 方法名称

每个方法调用都包含一个方法名称。如前所述，方法名称可以用、也可以不用命名空间和类型名称加以限定。在方法名称之后是圆括号中的实参列表，每个实参以逗号分隔，对应于声明方法时指定的形参。

4.1.5 形参和实参

方法可接收任意数量的形参，每个形参都具有特定的数据类型。调用者为形参提供的值称为**实参**；每个实参都要和一个形参对应。以代码清单4-2使用的方法调用为例，它有3个实参：

```
System.Console.WriteLine(
    "Your full name is {1} {0}", lastName, firstName)
```

方法声明为接收3个参数。第一个参数是string类型，后面两个参数是object类型。(注意，这里传递的实参都是string类型。编译器之所以允许这样做，是因为所有字符串都兼容于object类型。)

4.1.6 方法返回值

与System.Console.WriteLine()相比，代码清单4-2中的System.Console.ReadLine()没有任何参数，因为该方法声明为不获取任何参数。然而，这个方法有一个**方法返回值**。利用返回值，可以将调用方法所产生的结果返回给调用者。因为System.Console.ReadLine()有返回值，所以可以将返回值赋给变量firstName。除此之外，还可以将方法的返回值作为另一个方法的实参使用，如代码清单4-3所示。

代码清单4-3 将方法返回值作为实参传给另一个方法调用

```
class Program
{
  static void Main()
  {
      System.Console.Write("Enter your first name: ");
      System.Console.WriteLine("Hello {0}!",
          System.Console.ReadLine());
  }
}
```

在代码清单4-3中，不是先为变量赋值，再在System.Console.WriteLine()调用中使用这个变量。相反，是在调用System.Console.WriteLine()时直接调用System.Console.ReadLine()方法。运行时，System.Console.ReadLine()方法将首先执行，其返回值将直接

传给System.Console.WriteLine()方法，而不是传给一个变量。

并非所有方法都返回数据，System.Console.Write()和System.Console.WriteLine()就是如此。稍后会讲到，这种方法指定了void返回类型。比如在HelloWorld的例子中，Main的返回类型就是void。

4.1.7 语句与方法调用的比较

代码清单4-3演示了语句和方法调用的差异。System.Console.WriteLine("Hello {0}!", System.Console.ReadLine());语句包含两个方法调用。语句通常包含一个或多个表达式，在本例中，每个表达式都是方法调用。所以，方法调用构成了语句的不同部分。

虽然在一个语句中包含多个方法调用能减少代码量，但不一定能增强可读性，而且很少能带来性能上的优势。开发者应该更加注重代码的可读性，而不要将过多的精力放在写简短的代码上。

> **注意**
>
> 通常，开发者应侧重于可读性，而不是在写出更短的代码方面耗费心机。为了使代码一目了然，进而在长时间里更容易维护，可读性是关键。

4.2 方法的声明

本节描述了如何声明方法来包含参数或返回类型。代码清单4-4演示了这些概念，输出4-1展示了结果。

代码清单4-4 声明方法

```
class IntroducingMethods
{
  static void Main()
  {
      string firstName;
      string lastName;
      string fullName;

      System.Console.WriteLine("Hey you!");

      firstName = GetUserInput("Enter your first name: ");
      lastName = GetUserInput("Enter your last name: ");

      fullName = GetFullName(firstName, lastName);

      DisplayGreeting(fullName);
  }

  static string GetUserInput(string prompt)
  {
      System.Console.Write(prompt);
      return System.Console.ReadLine();
  }

  static string GetFullName(string firstName, string lastName)
  {
```

```
        return firstName + " " + lastName;
    }

    static void DisplayGreeting(string name)
    {
        System.Console.WriteLine("Your full name is {0}.", name);
        return;
    }
}
```

输出4-1

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

代码清单4-4声明了4个方法。从Main()中，代码调用了GetUserInput()，然后调用了GetFullName()。两个方法都返回一个值，而且都要获取实参。最后，代码调用了DisplayGreeting()，它不返回任何数据。C#的每个方法都必须在某个类型中；本例的包容类型就是IntroducingMethods类。即使第1章就开始讨论的Main()方法，它也必须在一个类型中。

语言对比：C++/Visual Basic——全局方法

C#不支持全局方法，一切都必须在一个类型定义中。这正是Main()方法被标记为static的原因——static在这里等价于C++的全局方法和Visual Basic的“共享”方法。

初学者主题：用方法进行重构

将一组相关语句转移到一个方法中，而不是把它们留在一个较大的方法中，这是**重构**（refactoring）的一种形式。重构有助于减少重复代码，因为可以从多个位置调用方法，而不必在每个位置都重复这个方法的代码。重构还有助于增强代码的可读性。编码时应经常检查代码，找出可以重构的地方。尤其是那些不好理解的代码块，最好把它们转移到方法中，用有意义的方法名清晰地定义代码的行为。与简单地为一个代码块加上注释相比，重构的效果更好，因为只需看方法名就可清除地知道这个方法要做的事情。

例如，代码清单4-4的Main()方法具有与第1章的代码清单1-15的Main()方法一样的行为。虽然两者都很容易懂，但代码清单4-4显得更简洁。只需扫一眼Main()方法就可以理解这个程序是干什么的（暂时不去操心被调用的每个方法的实现细节）。

在Visual Studio中选定一组语句，右击并在弹出的菜单中选择“重构”，然后再选择“提取方法”，即可自动将一组语句转移到一个新方法中。

4.2.1 形式参数声明

来看看DisplayGreeting()和GetFullName()方法的声明。在方法声明的圆括号内可以添加**形式参数列表**（讨论泛型时会讲到，方法也可以有**类型参数列表**）。列表中的每个参数

都包含参数类型和参数名，每个参数以逗号分隔。

大多数参数的行为和命名规范与局部变量一致。换言之，参数名采用的是camel大小写风格。另外，不能在方法中声明与参数同名的局部变量，因为这会造成同名的两个“局部变量”。

规范

要为参数名使用camel大小写风格。

4.2.2 方法返回类型声明

GetUserInput()和GetFullName()方法除了指明要获取的参数，还包括了**方法返回类型**。很容易就可以辨别一个方法是否有返回值，因为在声明这个方法时，会在方法名之前添加一个数据类型。GetUserInput()和GetFullName()的返回数据类型都是string。虽然方法可以指定多个参数，但返回类型只能有一个。

具有返回类型的方法几乎总是包含一个或多个return语句，它将控制返回给调用者。return语句以return关键字开头，后跟计算返回值的表达式。例如，GetFullName()方法的return语句是return firstName + " " + lastName。return关键字后面的表达式必须兼容于方法的返回类型。

如果方法有返回类型，它的主体必须有“不可到达的结束点”。换言之，一个具有返回类型的方法不允许在不返回任何值的情况下将控制贯穿到方法的末尾。为了保证这一点，最简单的办法就是将return语句作为方法的最后一个语句。但这并非绝对，return语句并非只能在方法末尾出现。例如，方法中的if或switch语句可以包含return语句，如代码清单4-5所示。

代码清单4-5 方法中间的return语句

```
class Program
{
  static bool MyMethod()
  {
      string command = ObtainCommand();
      switch(command)
      {
          case "quit":
              return false;
          // ... omitted, other cases
          default:
              return true;
      }
  }
}
```

（注意，return语句将控制转移出switch，所以，在以return语句作为方法最后一个语句的方法中，不需要用break语句防止非法“贯穿”switch小节。）

在代码清单4-5中，方法的最后一个语句不是return语句，而是switch语句。但是，编译器判断这个方法的每条执行路径最终都是return语句，所以方法结束点“不可达”。这

样的方法是合法的，即使它不以return语句结尾。

如果return之后有“不可达”的语句，编译器会发出警告，指出有永远执行不到的语句。

虽然C#允许提前返回，但为了增强代码的可读性，以及使代码更易维护，应该尽可能地确定单一的退出位置，而不是在方法的多个代码路径中散布多个return语句。

指定void作为返回类型，表示方法没有返回值。所以，这种方法不支持向变量赋值，也无法在调用位置[①]作为参数传递。除此之外，在这种方法内部，return是可有可无的。如果真的指定了return，它之后不能有任何值。例如在代码清单4-4中，Main()的返回值是void，它没有使用任何return语句。而DisplayGreeting()最后使用了一个return语句，但return之后没有添加任何值。

语言对比：C++——头文件

和C++不同，C#类从来不将实现与声明分开。C#不区分头文件（.h）和实现文件（.cpp）。相反，声明和实现总是出现在同一个文件中。（C#确实支持名为“分部方法”的高级功能，允许将方法的声明和实现分开。但考虑到本章的目的，我们只讨论非分部方法。）这样做就不需要像C++那样在两个位置维护冗余的声明信息。

初学者主题：命名空间

如前所述，**命名空间**（namespace）是分类和组合相关类型的组织机制。在一个类型所在的命名空间中，能找到和它有关的其他类型。此外，重名的两个或更多类型只要在不同命名空间中，就没有歧义。

4.3 using指令

完全限定的命名空间名称可能很长、很笨拙。可将一个或多个命名空间的所有类型“导入”一个文件，这样在使用它们时就不需要进行完全限定。在C#中，程序员通过使用using指令（通常在文件的顶部）来实现的。例如在代码清单4-6中，Console没有附加System前缀。而是在代码清单的顶部使用了一个using System;语句。

代码清单4-6 using指令的例子

```
// The using directive imports all types from the
// specified namespace into the entire file.
using System;

class HelloWorld
{
  static void Main()
  {
      // No need to qualify Console with System
      // because of the using directive above.
      Console.WriteLine("Hello, my name is Inigo Montoya");
  }
}
```

① call site，就是发出调用的地方，可以理解成调用了一个目标方法的表达式或者代码行。——译者注

代码清单4-6的结果如输出4-2所示。

输出4-2

```
Hello, my name is Inigo Montoya
```

即使存在using System这样的using指令，也不能在使用System的某个子命名空间中的类型时省略System字样。例如，假定代码要访问System.Text中的StringBuilder类型，那么必须增加一个using System.Text;指令，或者使用System.Text.StringBuilder对类型进行完全限定，而不能仅仅写成Text.StringBuilder。简单地说，using指令不会导入任何**嵌套命名空间**中的类型。嵌套命名空间（由命名空间中的句点符号来标识）必须显式导入。

语言对比：Java——import指令中的通配符

Java允许使用通配符导入命名空间，例如：

```
import javax.swing.*;
```

与之相比，C#不允许在using指令中使用通配符，每个命名空间都必须显式地导入。

语言对比：Visual Basic .NET——项目范围的Imports指令

和C#不同，Visual Basic .NET允许为整个项目（而非只是单个文件）使用与using指令等价的Imports指令。换言之，Visual Basic .NET提供了using指令的一个命令行版本，它对整个项目的所有文件起作用。

通常，假如程序要用到一个命名空间中的许多类型，就应该考虑为该命名空间使用using指令，以避免对该命名空间中的所有类型都进行完全限定。正是这个原因，几乎所有文件都在顶部添加了using System指令。在本书剩余的部分，代码清单中经常会省略using System指令。但是，其他命名空间指令都会显式地包含。

使用using System指令的一个有趣结果是，可以使用不同的大小写形式来表示字符串数据类型：String或者string。前者的基础是using System指令，后者使用的是string关键字。两者在C#中都引用System.String数据类型，最终生成的CIL代码毫无区别[①]。

高级主题：嵌套的using指令

不仅可以在文件顶部使用using指令，还可以在命名空间声明的顶部包含它们。例如，假定声明了新命名空间EssentialCSharp，可以在这个声明的顶部添加using指令，如代码清单4-7所示。

① 我更喜欢使用string关键字，但不管选择哪一种表示方法，都应该在项目中保持一致。

代码清单4-7 在命名空间声明中指定using指令

```
namespace EssentialCSharp
{
  using System;

  class HelloWorld
  {
    static void Main()
    {
        // No need to qualify Console with System
        // because of the using directive above.
        Console.WriteLine("Hello, my name is Inigo Montoya");
    }
  }
}
```

输出4-3展示了结果。

输出4-3

```
Hello, my name is Inigo Montoya
```

在文件顶部放置using指令和在命名空间声明的顶部放置using指令的区别在于，后者的using指令只在声明的命名空间内有效。假如在EssentialCSharp命名空间前面或后面声明了新的命名空间，别的命名空间中的using System指令不会对新的命名空间起作用。不过，很少会这样写代码，尤其是根据约定，每个文件只应该有一个类型声明。

使用别名

还可利用using指令为命名空间或类型取一个别名。**别名**（alias）是在using指令起作用的范围内可以使用的替代名称。别名的两个最常见的用途是消除两个同名类型的歧义和缩写长名称。例如在代码清单4-8中，CountDownTimer别名引用了System.Timers.Timer类型。仅添加using System.Timers指令不足以完全限定Timer类型，原因是System.Threading也包含一个名为Timer的类型，所以在代码中直接使用Timer会造成歧义。

代码清单4-8 声明一个类型别名

```
using System;
using System.Threading;
using CountDownTimer = System.Timers.Timer;

class HelloWorld
{
  static void Main()
  {
      CountDownTimer timer;

      // ...
  }
}
```

代码清单4-8将全新的名称CountDownTimer作为别名。然而，可以将别名指定为Timer，如代码清单4-9所示。

代码清单4-9 声明同名的一个类型别名

```
using System;
using System.Threading;

// Declare alias Timer to refer to System.Timers.Timer to
// avoid code ambiguity with System.Threading.Timer
using Timer = System.Timers.Timer;

class HelloWorld
{
    static void Main()
    {
        Timer timer;

        // ...
    }
}
```

由于Timer现在是别名，所以“Timer”引用没有歧义。这时，如果要引用System.Threading.Timer类型，必须完全限定或者定义不同的别名。

4.4 Main()的返回值和参数

迄今为止，你看到的所有Main()方法采用的都是最简单的声明。在这些Main()方法声明中，不包含任何参数或返回类型。然而，C#支持在执行程序时提供命令行参数，并允许从Main()方法返回状态标识符。

“运行时”通过一个string数组参数将命令行参数传给Main()。要获取参数，访问数组就可以了，代码清单4-10对此进行了演示。程序的目的是下载指定URL位置的文件。第一个命令行参数指定了URL，第二个参数指定了存盘文件名。代码从一个switch语句开始，它根据参数数量（args.Length）采取不同的操作：

（1）如果没有两个参数，就显示一条错误消息，指出必须提供URL和文件名；

（2）如果有两个参数，表明用户既提供了资源的URL，也提供了下载的目标文件名。

代码清单4-10 将命令行参数传给Main()

```
using System;
using System.IO;
using System.Net;

class Program
{
  static int Main(string[] args)
  {
      int result;
      string targetFileName;
      string url;
      switch (args.Length)
      {
          default:
              // Exactly two arguments must be specified; give an error.
              Console.WriteLine(
                  "ERROR:  You must specify the "
                  + "URL and the file name");
```

```
            targetFileName = null;
            url = null;
            break;
        case 2:
            url = args[0];
            targetFileName = args[1];
            break;
    }

    if (targetFileName != null && url != null)
    {
        WebClient webClient = new WebClient();
        webClient.DownloadFile(url, targetFileName);
        result = 0;
    }
    else
    {
        Console.WriteLine(
            "Usage: Downloader.exe <URL> <TargetFileName>");
        result = 1;
    }
    return result;
  }


}
```

代码清单4-10的结果如输出4-4所示。

输出4-4

```
>Downloader.exe
ERROR:  You must specify the URL to be downloaded
Downloader.exe <URL> <TargetFileName>
```

假如成功地获得了目标文件名，就可以用它来保存下载的文件。否则，应该显示帮助文本。Main()方法还会返回一个int，而不是像往常那样返回void。返回值对于Main()声明来说是可选的。但如果有返回值，程序就可以将状态码返回给调用者（如脚本或批处理文件）。根据约定，非零的返回值代表错误。

虽然所有命令行参数都可通过字符串数组传给Main()，但有时需要从非Main()的方法中访问那些参数。这时可用System.Environment.GetCommandLineArgs()方法返回由命令行参数构成的数组。

高级主题：消除多个Main()方法的歧义

假如一个程序的两个类都有Main()方法，可以在命令行上用csc.exe的/m开关指定包含入口点的类。

初学者主题：调用栈和调用点

代码执行时，方法可能调用其他方法，其他方法可能调用更多的方法，以此类推。在代码清单4-4展示的简单情况中，Main()调用了GetUserInput()，GetUserInput()调用了System.Console.ReadLine()，而System.Console.ReadLine()在内部会调用更多的方法。每次调用新方法，“运行时”都创建一个“栈帧”，其中包含的内容涉及传给新

调用的实参、新调用的局部变量以及方法返回时应该从哪里恢复等。这样形成的一系列栈帧称为**调用栈**（call stack）。随着程序复杂程度的提高，每个方法调用另一个方法时，这个调用栈都会变大。然而，当调用结束时，调用栈会发生收缩，直到调用另一个方法。我们用**栈展开**（stack unwinding）[①]这个词描述从调用栈中删除栈帧的过程。栈展开的顺序通常与方法调用的顺序相反。方法调用完毕之后，控制会返回**调用点**（call site），也就是最初发出方法调用的位置。

4.5 方法的参数

之前一直是通过方法的return语句返回数据。本节描述了如何通过方法参数返回数据，以及方法如何获取数量可变的参数。

4.5.1 值参数

参数默认是**传值**（pass by value）的。换言之，参数表达式的值会复制到目标参数中。例如在代码清单4-11中，在调用Combine()的时候，Main()使用的每个变量值都会复制到Combine()方法的参数中。输出4-5展示了这个代码清单的结果。

代码清单4-11 以传值方式传递变量

```
class Program
{
  static void Main()
  {
    // ...
    string fullName;
    string driveLetter = "C:";
    string folderPath  = "Data";
    string fileName    = "index.html";

    fullName = Combine(driveLetter, folderPath, fileName);

    Console.WriteLine(fullName);
    // ...
}

static string Combine(
    string driveLetter, string folderPath, string fileName)
{
    string path;
    path = string.Format("{1}{0}{2}{0}{3}",
        System.IO.Path.DirectorySeparatorChar,
        driveLetter, folderPath, fileName);
    return path;
  }
}
```

输出4-5

```
C:\Data\index.html
```

① unwind一般翻译成“展开”，但这并不是一个很好的翻译。wind和unwind源于生活。把线缠到线圈上称为wind；从线圈上松开称为unwind。同样地，调用方法时压入栈帧，称为wind；方法执行完毕，弹出栈帧，称为unwind。——译者注

Combine()方法返回之前，即使将null值赋给driveLetter、folderPath和fileName等变量，Main()中对应的变量仍会保持其初始值不变，因为在调用方法时，只是将变量的值复制了一份给方法。调用栈在一次调用的末尾“展开”的时候，复制的数据会被丢弃。

初学者主题：匹配调用者变量与参数名

在代码清单4-11中，调用者中的变量名与被调用方法中的参数名相匹配。这种匹配纯粹是为了增强可读性，名称是否匹配与方法调用的行为无关。被调用方法的参数和发出调用的方法的局部变量在不同声明空间中，相互之间没有任何关系。

高级主题：引用类型与值类型的对比

就本节来说，传递的参数是值类型还是引用类型并不重要。重要的是目标方法是否能将新值赋给调用者的初始变量。由于现在是生成调用者的变量的值的副本，所以怎么更改都影响不到调用者的变量。但不管怎样，都有必要理解值类型的变量和引用类型的变量的区别。

从名字就可以看出，对于引用类型的变量，它的值是对数据实际存储位置的引用。“运行时”如何表示引用类型变量的值，这是“运行时”的实现细节。一般都是用数据实际存储的内存地址来表示，但并非一定如此。

如果以传值方式传递一个引用类型的变量，引用（地址）本身会从调用者复制给方法参数。其结果就是，虽然目标方法更改不了调用者变量的值，即更改不了引用（地址）本身，但可以更改引用所指向的数据。

另一方面，对于值类型的参数，参数获得的是值的副本，所以在被调用的方法中更改参数不会影响调用者的原始变量。

4.5.2 引用参数（ref）

来看看代码清单4-12的例子，它调用方法来交换两个值，输出4-6展示了结果。

代码清单4-12 以传引用的方式传递变量

```
class Program
{
  static void Main()
  {
      // ...
      string first = "hello";
      string second = "goodbye";
      Swap(ref first, ref second);

      System.Console.WriteLine(
          @"first = ""{0}"", second = ""{1}""",
          first, second);
      // ...
  }

  static void Swap(ref string x, ref string y)
  {
      string temp = x;
      x = y;
      y = temp;
  }
}
```

输出4-6

```
first = "goodbye", second = "hello"
```

赋给first和second的值被成功地交换。这需要以**传引用**（pass by reference）的方式传递变量。将这个例子的Swap()调用与代码清单4-11的Combine()调用进行比较，不难发现，二者最明显的区别就是本例在参数的数据类型之前使用了关键字ref。这个关键字使参数以传引用的方式传递，使被调用的方法可以用新值来更新调用者的变量。

如果被调用的方法将参数指定为ref，那么调用者在调用这个方法的时候，提供的实参应该是附加了ref前缀的变量（而不是值）。这样调用者就显式地指定了目标方法可以对它接收到的任何ref参数进行重新赋值。除此之外，调用者应该对传引用的变量进行初始化，因为目标方法可能直接从ref参数读取数据而不先对其进行赋值。例如在代码清单4-12中，一开始就将first的值赋给temp——假定通过first传递的变量已由调用者进行了初始化。事实上，ref参数只是传递的变量的别名。换言之，只是为现有的变量分配了参数名，而不是创建新变量并将实参的值复制给它。

4.5.3 输出参数（out）

如前所述，用作ref参数的变量必须在传给被调用的方法之前赋值，因为被调用的方法可能从这个变量中读取值。例如，前面的Swap方法会对传给它的变量进行读取和写入。然而，方法经常要获取一个变量引用，并且只向变量中写入而不从中读取。这时更安全的做法是以传引用的方式传入一个未初始化的局部变量。

为此，代码需要用关键字out修饰参数类型。例如代码清单4-13的TryGetPhoneButton()方法，它返回与字符对应的电话按键。

代码清单4-13 仅传出的变量

```
class ConvertToPhoneNumber
{
  static int Main(string[] args)
  {
      char button;

      if(args.Length == 0)
      {
          Console.WriteLine(
              "ConvertToPhoneNumber.exe <phrase>");
          Console.WriteLine(
              "'_' indicates no standard phone button");
          return 1;
      }
      foreach(string word in args)
      {
          foreach(char character in word)
          {
              if(TryGetPhoneButton(character, out button))
              {
                  Console.Write(button);
              }
              else
              {
                  Console.Write('_');
```

```
            }
        }
    }
    Console.WriteLine();
    return 0;
}

static bool TryGetPhoneButton(char character, out char button)
{
    bool success = true;
    switch( char.ToLower(character) )
    {
    case '1':
        button = '1';
        break;
    case '2': case 'a': case 'b': case 'c':
        button = '2';
        break;

    // ...

    case '-':
        button = '-';
        break;
    default:
        // Set the button to indicate an invalid value
        button = '_';
        success = false;
        break;
    }
    return success;
  }
}
```

输出4-7展示了代码清单4-13的结果。

输出4-7

```
>ConvertToPhoneNumber.exe CSharpIsGood
2742774744663
```

在这个例子中，假如能成功判断与字符对应的电话按键，TryGetPhoneButton()方法就返回true。方法还会使用声明为out的button参数来返回对应的按钮。

out参数在功能上和ref参数完全一致，唯一的区别是，C#语言对别名变量的读写有不同的规定。如果参数被标记为out，编译器会核实在方法所有正常返回的代码路径中，是否都对该参数进行了赋值。例如，假定在某个代码执行路径中，没有对button进行赋值，编译器就会报错，指出代码没有对button进行初始化。在代码清单4-13中，方法最后将下划线字符赋给button，因为即使无法判断正确的电话按键，也必须对button进行赋值。

开发人员可以通过声明一个或多个out参数来克服方法只有一个返回类型的限制。例如，如果方法要返回两个值，就可以以方法返回值的形式正常返回一个值，另一个值写入别名变量并作为out参数传递。虽然这种做法既常见，也合法，但通常都还有更好的解决方案达到相同的目的。首先，如果一个方法需要返回两个或更多值，那么实际上应考虑改写

成两个方法，每一个方法返回一个值。如果这个方法一定要返回两个值，就可以考虑使用元组（tuple）类型（以便表示多个值）或者可空值类型（表示可能“无效”的值，如上例所示）。这样就可以避免以传引用的方式传递变量。

4.5.4 参数数组

在迄今为止讲到的例子中，参数数量都由目标方法声明所确定。但有时希望参数数量可变。以代码清单4-11的Combine()方法为例。那个方法传递了驱动器号、文件夹路径以及文件名（driveLetter、folderPath、fileName）等参数。假如路径中的文件夹数量大于1，调用者希望将额外的文件夹联结起来，以构成一个完整的路径，那么应该如何编写代码呢？也许最好的办法就是为文件夹传递一个字符串数组。然而，这会使调用代码变复杂，因为需要事先构造好一个数组，再将数组作为参数传递。

为了简化编码，C#提供了一个特殊的关键字，允许在调用方法时提供数量可变的参数，而不是由方法事先固定好参数数量。在具体讨论这种方法声明之前，先观察一下代码清单4-14的Main()方法中的调用代码。

代码清单4-14 传递一个长度可变的参数列表

```
using System.IO;

class PathEx
{
  static void Main()
  {
      string fullName;

      // ...

      // Call Combine() with four arguments
      fullName = Combine(
          Directory.GetCurrentDirectory(),
          "bin", "config", "index.html");
      Console.WriteLine(fullName);

      // ...

      // Call Combine() with only three arguments
      fullName = Combine(
          Environment.SystemDirectory,
          "Temp", "index.html");
      Console.WriteLine(fullName);

      // ...

      // Call Combine() with an array
      fullName = Combine(
          new string[] {
              "C:\\", "Data",
              "HomeDir", "index.html"} );
      Console.WriteLine(fullName);
      // ...
  }

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

```
    {
        string result = string.Empty;
        foreach (string path in paths)
        {
            result = System.IO.Path.Combine(result, path);
        }
        return result;
    }
}
```

输出4-8展示了代码清单4-14的结果。

输出4-8

```
C:\Data\mark\bin\config\index.html
C:\WINDOWS\system32\Temp\index.html
C:\Data\HomeDir\index.html
```

第一个Combine()调用指定了4个参数。第二个调用只指定了3个参数。在最后一个调用中，参数用一个数组来传递。换言之，Combine()方法可接受数量可变的参数，这些参数要么是以逗号分隔的字符串参数，要么是单个字符串数组。前者的语法称为方法调用的“展开”（expanded）形式，后者称为“正常”（normal）形式。

为了获得这样的效果，Combine()方法需要：

（1）在方法声明的最后一个参数之前添加params关键字；

（2）将最后一个参数声明为数组。

像这样声明了**参数数组**（parameter array）之后，每个参数都作为参数数组的成员来访问。Combine()方法遍历paths数组的每个元素，并调用System.IO.Path.Combine()。该方法自动合并路径中的不同部分，并能正确使用平台特有的目录分隔符。注意，PathEx.Combine()完全等价于Path.Combine()，只是PathEx.Combine()能处理数量可变的参数，而非只能处理两个。

参数数组要注意下面这些地方。

- 参数数组不一定是方法的唯一参数，但必须是方法声明中的最后一个参数。由于只有最后一个参数才可能是参数数组，所以方法最多只能有一个参数数组。
- 调用者可以指定和参数数组对应的零个实参，这会造成传递的参数数组包含零个数据项。
- 参数数组是类型安全的——实参的类型必须兼容于参数数组中元素的类型。
- 调用者可以显式地使用数组，而不是以逗号分隔的参数列表。最终生成的CIL代码是一样的。
- 假如目标方法的实现要求一个最起码的参数数量，请在方法声明中显式指定必须提供的参数。这样如果遗漏要求的参数，会导致编译器报错，而不需要依赖于运行时错误处理。例如，如果一个方法要求一个或多个整型实参，那就应该使用int Max(int first, params int[] operands)来声明这个方法，而不是使用int Max(params int[] operands)来声明，这样可以确保至少有一个整数实参传给Max()。

使用参数数组，可以将相同类型的、数量可变的多个参数传给方法。本章后面的4.7节讨论了如何支持不同类型的、数量可变的参数。

4.6 递归

“递归地调用方法”或者“用递归实现方法”意味着方法调用它自身。有时这是实现算法最简单的方式。代码清单4-15统计目录及其子目录中的所有C#源代码文件（*.cs）的代码行数。

代码清单4-15 返回目录中的所有.cs文件包含的代码行数

```
using System.IO;

public static class LineCounter
{
  // Use the first argument as the directory
  // to search, or default to the current directory.
  public static void Main(string[] args)
  {
      int totalLineCount = 0;
      string directory;
      if (args.Length > 0)
      {
          directory = args[0];
      }
      else
      {
          directory = Directory.GetCurrentDirectory();
      }
      totalLineCount = DirectoryCountLines(directory);
      System.Console.WriteLine(totalLineCount);
  }

  static int DirectoryCountLines(string directory)
  {
      int lineCount = 0;
      foreach (string file in
          Directory.GetFiles(directory, "*.cs"))
      {
          lineCount += CountLines(file);
      }

      foreach (string subdirectory in
          Directory.GetDirectories(directory))
      {
          lineCount += DirectoryCountLines(subdirectory);
      }

      return lineCount;
  }

  private static int CountLines(string file)
  {
      string line;
      int lineCount = 0;
      FileStream stream =
          new FileStream(file, FileMode.Open);①
      StreamReader reader = new StreamReader(stream);
      line = reader.ReadLine();

       while(line != null)
```

① 可用using语句改进这行代码，但暂时没有使用这个构造，因为到目前为止还没有讲到。

```
        {
            if (line.Trim() != "")
            {
                lineCount++;
            }
            line = reader.ReadLine();
        }

        reader.Close();  // Automatically closes the stream
        return lineCount;
    }
}
```

输出4-9展示了代码清单4-15的结果。

输出4-9

```
104
```

程序首先将第一个命令行参数传给DirectoryCountLines()，或直接使用当前目录（如果没有提供参数）。该方法首先遍历当前目录中的所有文件，累加每个.cs文件包含的源代码行数。处理好当前目录之后，代码将subdirectory传给Directory.CountLines()方法以处理每个子目录。同样的过程针对每个子目录反复进行，直到再也没有更多的子目录可供处理。

不熟悉递归的读者刚开始可能觉得非常繁琐。但事实上，递归通常都是最简单的编码模式，尤其是在和文件系统这样的层次化数据打交道的时候。不过，虽然可读性不错，但一般不是最快的实现。如果必须关注性能，开发者应该为递归实现寻求一种替代方案。具体如何选择，通常取决于想如何在可读性与性能之间取得平衡。

初学者主题：无限递归错误

用递归实现方法时，常见错误是在程序执行期间发生栈溢出。这通常是由于无限递归（infinite recursion）造成的。假如方法持续地调用自身，永远抵达不了标志递归结束的位置，就会发生无限递归。必须仔细地检查每个使用了递归的方法，验证递归调用是有限的而非无限的。

下面以伪代码的形式展示了一个常用的递归模式：

```
M(x)
{
  if x已达最小，不可继续分解 ①
    返回结果
else
    (1) 采取一些操作使问题变得更小
    (2) 递归调用M来解决更小的问题
    (3) 根据(1)和(2)计算结果
    返回结果
}
```

不遵守这个模式就可能出错。例如，如果不能将问题变得更小，或者不能处理所有可能的“最小”情况，就会递归个不停。

① 或者说：if满足递归结束条件（base case）。——译者注

4.7 方法重载

代码清单4-15调用了DirectoryCountLines()方法，它统计*.cs文件包含的源代码行数。然而，如果要统计*.h/*.cpp文件或*.vb文件的代码行数，DirectoryCountLines()就不起作用了。因此，我们希望有这样一个方法，它能获取文件扩展名作为参数，同时保留现有的方法定义，以便在默认情况下处理*.cs文件。

一个类中的所有方法都必须有唯一签名，C#依据方法名、参数数据类型或者参数数量的不同来定义唯一性。但是，在所有这些依据中，并不包括方法的返回类型；假如两个方法只是返回数据类型不同，那么会造成一个编译错误。假如一个类包含两个或者更多同名的方法，就会发生**方法重载**。对于重载的方法，参数的数量和/或数据类型肯定是不同的。

方法重载是一种**操作性多态**（operational polymorphism）。如果由于数据的变化造成同一个逻辑操作具有许多（“多”）形式（“态”），就会发生“多态”。以WriteLine()方法为例，可向它传递一个格式字符串和其他一些参数，也可只传递一个整数。两者的实现肯定不同。但在逻辑上，对于调用者，这个方法就是负责输出数据。至于方法内部如何实现，调用者并不关心。代码清单4-16展示了一个例子，输出4-10展示了结果。

代码清单4-16 使用重载统计.cs文件中的行数

```
using System.IO;

public static class LineCounter
{
    public static void Main(string[] args)
    {
        int totalLineCount;

        if (args.Length > 1)
        {
            totalLineCount =
                DirectoryCountLines(args[0], args[1]);
        }
        if (args.Length > 0)
        {
            totalLineCount = DirectoryCountLines(args[0]);
        }
        else
        {
            totalLineCount  = DirectoryCountLines();
        }

        System.Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        return DirectoryCountLines(
            Directory.GetCurrentDirectory());
    }

    static int DirectoryCountLines(string directory)
    {
        return DirectoryCountLines(directory, "*.cs");
    }
```

```
    static int DirectoryCountLines(
        string directory, string extension)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        int lineCount = 0;
        string line;
        FileStream stream =
            new FileStream(file, FileMode.Open);①
        StreamReader reader = new StreamReader(stream);
        line = reader.ReadLine();
        while(line != null)
        {
            if (line.Trim() != "")
            {
                lineCount++;
            }
            line = reader.ReadLine();
        }

        reader.Close();  // Automatically closes the stream
        return lineCount;
    }
}
```

输出4-10

```
>LineCounter.exe .\ *.cs
28
```

方法重载的作用是提供调用方法的多种方式。如上例所示，调用DirectoryCountLines()方法时可选择是否传递要搜索的目录以及文件扩展名。

本例修改了DirectoryCountLines()的无参数版本，让它调用单参数版本int DirectoryCountLines(string directory)。这是实现重载方法时经常采用的一种模式。它的基本思路是：开发者只需在一个方法中实现核心逻辑，其他所有重载版本都调用那个方法。如果核心实现发生了改变，那么只需要在一个位置修改，而不必在每个实现中都进行

① 可用using语句改进这行代码，但暂时没有使用这个构造，因为到目前为止还没有讲到。

修改。利用方法重载技术来支持可选的参数时，这种模式非常有用。注意这些参数的值在编译时不能确定，所以不适合用C# 4.0新增的“可选参数”功能来写。

4.8 可选参数

从C# 4.0开始，语言的设计者增添了对**可选参数**的支持。声明方法时将常量值赋给参数，以后调用方法时就不必每个参数都指定，如代码清单4-17所示。

代码清单4-17 使用了可选参数的方法

```
using System.IO;

public static class LineCounter
{
  public static void Main(string[] args)
  {
      int totalLineCount;

      if (args.Length > 1)
      {
          totalLineCount =
              DirectoryCountLines(args[0], args[1]);
      }
      if (args.Length > 0)
      {
          totalLineCount = DirectoryCountLines(args[0]);
      }
      else
      {
          totalLineCount  = DirectoryCountLines();
      }

      System.Console.WriteLine(totalLineCount);
  }

  static int DirectoryCountLines()
  {
      // ...
  }

  /*
    static int DirectoryCountLines(string directory)
    { ... }
  */

    static int DirectoryCountLines(
        string directory, string extension = "*.cs")
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
```

```
        Directory.GetDirectories(directory))
    {
        lineCount += DirectoryCountLines(subdirectory);
    }

    return lineCount;
}

private static int CountLines(string file)
{
    // ...
}
```

在代码清单4-17中，DirectoryCountLines()方法的单参数版本已被移除（注释掉），但Main()方法似乎仍在调用该方法（只指定一个参数）。如果调用时不指定extension（扩展名）参数，就使用声明时赋给extension的值（本例中是*.cs）。这样一来，如果愿意，调用代码就可以不指定一个值，避免了在C# 3.0和更早的版本中对一个额外的重载版本的需求。注意，可选参数一定放在所有必须的参数（无默认值的参数）后面。另外，默认值必须是常量，或者说必须是能在编译时确定的值，这一点极大限制了“可选参数”的应用。例如，不能像下面这样声明方法：

```
DirectoryCountLines(
  String directory = Environment.CurrentDirectory,
  String extension = “*.cs”)
```

这是由于Environment.CurrentDirectory不是常量。相反，由于"*.cs"是常量，所以C#允许它作为可选参数的默认值。

规范

要尽量为所有参数提供好的默认值。
要提供简单的方法重载，其必需的参数的数量要少。
考虑从最简单到最复杂来组织重载。

C# 4.0新增的另一个方法调用功能是**命名参数**。利用命名参数，调用者可显式地为一个参数赋值，而不是像以前那样只能依据参数顺序来决定哪个值赋给哪个参数，如代码清单4-18所示。

代码清单4-18 调用方法时指定参数名

```
class Program
{
  static void Main()
  {
      DisplayGreeting(
          firstName: "Inigo", lastName: "Montoya");
  }

  public void DisplayGreeting(
      string firstName,
      string middleName = default(string),
```

```
        string lastName = default(string))
    {
        // ...
    }
}
```

在代码清单4-18中，从Main()中调用DisplayGreeting()时，是将值赋给一个命名参数。调用时，在两个可选参数中（middleName和lastName），只指定了lastName。如果一个方法有大量参数，而且其中许多都是可选的（访问Microsoft COM库时，这是很常见的一种情况），那么命名参数语法肯定能带来不少便利。但要注意的是，这个便利的代价是牺牲方法接口的灵活性。过去（至少就C#来说），参数名可以自由更改，不会造成调用代码无法编译的情况。但在添加了命名参数后，参数名就成为方法接口的一部分。更改名称会导致使用命名参数的代码无法编译。

规范

要将参数名视为API的一部分。如果API之间的版本兼容性很重要，就要避免更改参数名。

对于有经验的C#开发人员，这是一个令人吃惊的限制。然而，这个限制自.NET 1.0开始就作为CLS的一部分存在下来了。另外，Visual Basic一直都支持用命名参数调用方法。所以，库开发人员应该早已养成了不更改参数名的习惯，这样才能成功地与其他.NET语言进行互操作，不会因为版本的变化而造成自己开发的库失效。C# 4.0只是像其他许多.NET语言早就要求的那样，对参数名的更改进行了相同的限制。

方法重载、可选参数和命名参数这几种技术一起使用，将难以一眼看出最终调用的是哪个方法。只有在所有参数（可选参数除外）都恰好有一个对应的实参（不管是根据名称还是位置），而且该实参具有兼容类型的情况下，才说一个调用**适用于**（兼容于）一个方法。虽然这限制了可调用方法的数量，但不足以唯一性地标识方法。为了进一步区分方法，编译器只使用调用者显式标识的参数，忽略调用者没有指定的所有可选参数。所以，假如由于一个方法有可选参数，造成两个方法都适用，编译器最终选择的是无可选参数的方法。

高级主题：方法解析

当编译器必须从一系列“适用”的方法中选择一个最适合某个特定调用的方法时，会选择拥有最具体的参数类型的那个方法。假定有两个适用的方法，每个都要求将实参隐式地转换成形参的类型，最终选择的是形参类型是更具体的派生类型的那个方法。

例如，如果调用者传递的是一个int，那么接受double的方法将优先于接受object的方法。这是由于double比object更具体。因为，有不是double的object，但没有不是object的double，所以double更具体。

如果有多个适用的方法，但无法从中挑选出最具唯一性的，编译器就会报错，指明调用存在歧义。

例如，给定以下方法：

```
static void Method(object thing){}
static void Method(double thing){}
static void Method(long thing){}
static void Method(int thing){}
```

调用Method(42)会被解析成Method(int thing)，因为存在着一个从实参类型到形参类型完全匹配的方法。如果删除该方法，那么重载解析会选择long版本，因为long比double和object更具体。

C#规范包含额外的规则来决定byte、ushort、uint、ulong和其他数值类型之间的隐式转换。但在写程序时，最好是使用显式转型，以方便别人理解你想调用哪个目标方法。

4.9 用异常实现基本错误处理

本节将探讨如何利用**异常处理**机制来处理错误报告。

方法利用异常处理将有关错误的信息传给调用者，同时不需要使用返回值或者显式提供任何参数。代码清单4-19略微修改了第1章的HeyYou程序。这次不是请求用户输入姓氏，而是请求输入年龄。

代码清单4-19 将string转换成int

```
using System;

class ExceptionHandling
{
  static void Main()
  {
      string firstName;
      string ageText;
      int age;

      Console.WriteLine("Hey you!");

      Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      Console.Write("Enter your age: ");
      ageText = Console.ReadLine();
      age = int.Parse(ageText);

      Console.WriteLine(
          "Hi {0}!  You are {1} months old.",
          firstName, age*12);
  }
}
```

输出4-11展示了代码清单4-19的结果。

输出4-11

```
Hey you!
Enter your first name: Inigo
Enter your age: 42
Hi Inigo!  You are 504 months old.
```

System.Console.ReadLine()的返回值存储到ageText变量中，然后传给int数据类型提供的Parse()方法。该方法获取代表数字的string值，把它转换为int类型。

初学者主题：42作为字符串和整数

C#要求每个非空值都有一个与它关联的良好定义的类型。换言之，不仅数据的值是紧要的，与之关联的类型也是紧要的。所以，字符串42和整数42完全不同。字符串由4和2这两个字符构成，而int是数值42。

基于转换好的字符串，System.Console.WriteLine()语句以月份为单位打印年龄（age*12）。

但是，用户完全有可能输入一个无效的整数字符串。例如，输入“forty-two”会发生什么？Parse()方法不能完成这样的转换。它希望用户输入只包含数字的字符串。假如Parse()方法接收到一个无效的值，它需要某种方式将这一事实反馈给调用者。

4.9.1 捕捉错误

为了通知调用者参数无效，int.Parse()会**引发异常**。引发异常会终止执行当前分支，跳到调用栈中用于处理异常的第一个代码块。

但是，由于当前尚未提供任何异常处理，所以程序会向用户报告发生了**未处理的异常**。假定系统中没有注册任何调试器，错误信息会出现在控制台上，如输出4-12所示。

输出4-12

```
Hey you!
Enter your first name: Inigo
Enter your age: forty-two

Unhandled Exception: System.FormatException: Input string was
        not in a correct format.
    at System.Number.ParseInt32(String s, NumberStyles style,
        NumberFormatInfo info)
    at ExceptionHandling.Main()
```

显然，像这样的错误消息并不是特别有用。为了解决这个问题，有必要提供一个机制对错误进行恰当的处理，例如向用户报告一条更有意义的错误消息。

这个过程称为**捕捉异常**。代码清单4-20展示了具体的语法，输出4-13展示了结果。

代码清单4-20 捕捉异常

```
using System;

class ExceptionHandling
{
  static int Main()
  {
      string firstName;
      string ageText;
      int age;
      int result = 0;
```

```
Console.Write("Enter your first name: ");
firstName = Console.ReadLine();

Console.Write("Enter your age: ");
ageText = Console.ReadLine();

try
{
    age = int.Parse(ageText);
    Console.WriteLine(
        "Hi {0}!  You are {1} months old.",
        firstName, age*12);
}
catch (FormatException )
{
    Console.WriteLine(
        "The age entered, {0}, is not valid.",
        ageText);
    result = 1;
}
catch(Exception exception)
{
    Console.WriteLine(
        "Unexpected error:  {0}", exception.Message);
    result = 1;
}
finally
{
    Console.WriteLine("Goodbye {0}",
        firstName);
}

return result;
  }
}
```

输出4-13

```
Enter your first name: Inigo
Enter your age: forty-two
The age entered, forty-two, is not valid.
Goodbye Inigo
```

首先要用**try块**将可能引发异常的代码（age=int.Parse()）包围起来。这个块以try关键字开始。try关键字告诉编译器：开发者认为块中的代码有可能引发异常；如果真的引发了异常，那么某个catch块要尝试处理这个异常。

try块之后必须紧跟着一个或多个catch块（或/和一个finally块）。catch块（参见稍后的“高级主题：常规catch”）可选择指定异常的数据类型。只要数据类型与异常类型匹配，对应的catch块就会执行。但是，假如一直找不到合适的catch块，引发的异常就会变成一个未处理的异常，就好像没有进行异常处理一样。

图4-1展示了最终的程序流程。

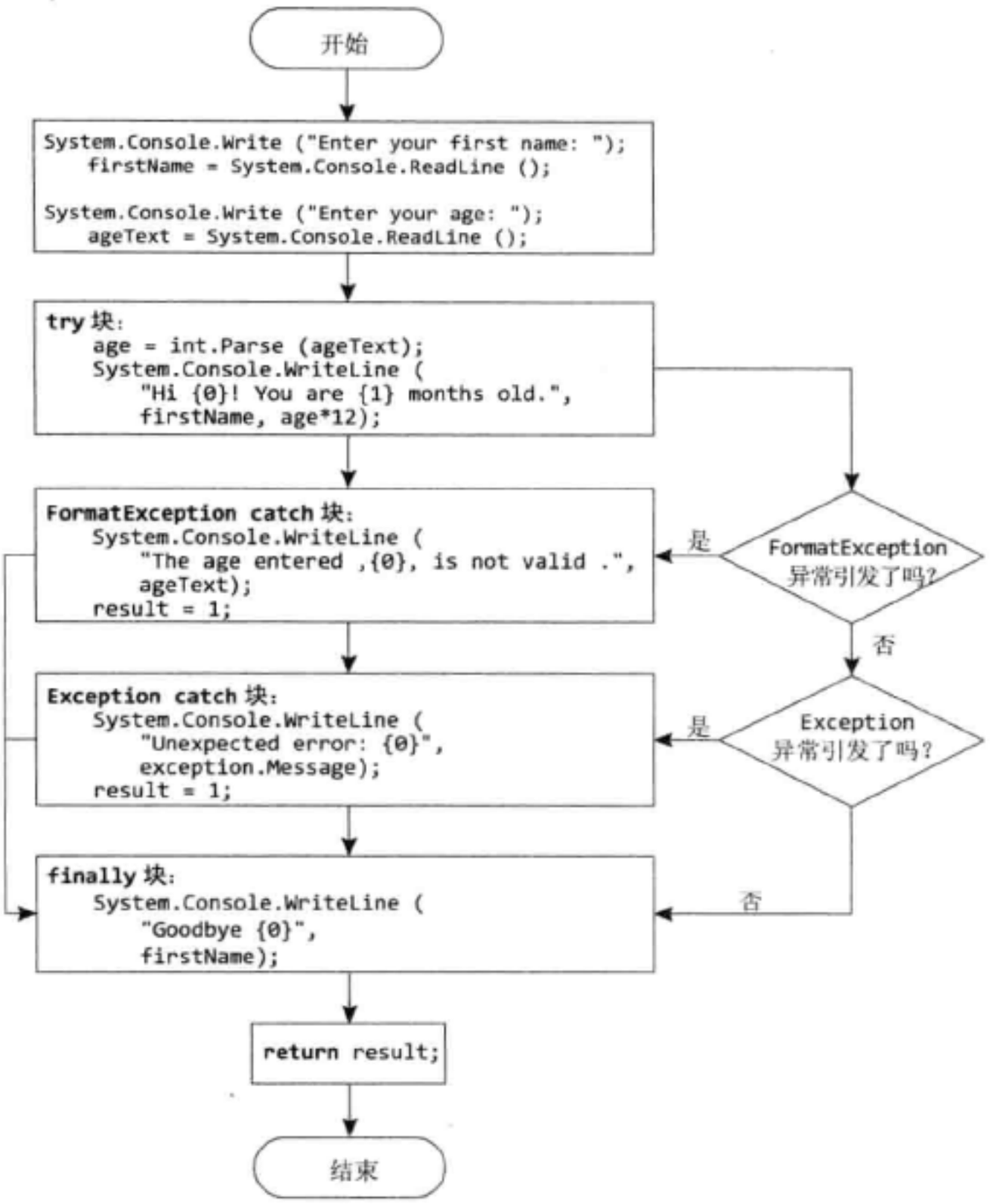


图4-1 异常处理控制流程

例如，假定输入“forty-two”，int.Parse()会引发System.FormatException类型的异常，控制会跳转到后面的一系列catch块（System.FormatException表明字符串格式不正确，无法进行解析）。由于第一个catch块就与int.Parse()引发的异常类型匹配，所以会执行这个块中的代码。但假如try块中的语句引发的是不同类型的异常，执行的就是第二个catch块，因为所有异常都是System.Exception类型。

如果没有System.FormatException这个catch块，那么即使int.Parse引发的是一个System. FormatException异常，也会执行System.Exception catch块。这是由于System.FormatException也是System.Exception类型（System.FormatException是泛化异常类System.Exception的一个更具体的实现）。

处理异常的顺序非常重要。catch块必须按照从最具体到最不具体排列。System. Exception数据类型是最不具体的，所以它应该最后出现。System. FormatException排在第一，因为它是代码清单4-20所处理的最具体的异常。

无论控制是正常地离开try块还是由于try块中的代码引发异常而离开的，只要控制离开try块，finally块就会执行。finally块的作用是提供一个最终位置，在其中放入无论是否发生异常都要执行的代码。finally块最适合用来执行资源清理。事实上，完全可以只写一个try块和一个finally块，而不写任何catch块。无论try块是否引发异常，甚至无论是否写了一个catch块来处理异常，finally块都会执行。代码清单4-21演示了一个try/finally块，输出4-14展示了结果。

代码清单4-21 有finally块但无catch块

```
using System;

class ExceptionHandling
{
  static int Main()
  {
      string firstName;
      string ageText;
      int age;
      int result = 0;

Console.Write("Enter your first name: ");
firstName = Console.ReadLine();

Console.Write("Enter your age: ");
ageText = Console.ReadLine();

try
{
    age = int.Parse(ageText);
    Console.WriteLine(
        "Hi {0}!  You are {1} months old.",
        firstName, age*12);
}
finally
{
    Console.WriteLine("Goodbye {0}",
        firstName);
}

        return result;
    }
  }
```

输出4-14

```
Enter your first name: Inigo
Enter your age: forty-two

Unhandled Exception: System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo
info)
   at ExceptionHandling.Main()
Goodbye Inigo
```

细心的读者可能已经注意到此处的一些有趣的事情了：“运行时”是先报告未处理的异常，然后才运行finally块。如何解释这种看似非常规的行为呢？

首先，这个行为是合法的，因为对于未处理的异常，“运行时”的行为是由具体实现定义的，任何行为都合法！“运行时”选择这个特定的行为是因为它知道在运行finally块之前，异常就已经是未处理的了。“运行时”已检查了调用栈上的所有栈帧，发现没有任何一个栈帧关联了能和引发的异常相匹配的catch块。

一旦“运行时”判断某个异常将成为未处理的异常，就会检查机器上是否安装了调试器，因为用户可能是软件开发人员，正要对这种失败进行分析。如果是，就在运行finally块之前为用户提供将调试器与进程连接的机会。如果没有安装调试器，或者用户拒绝调试

问题，默认的行为就是在控制台上打印出未处理的异常，再看是否有哪些finally块可供运行。注意，由于这是“由具体实现定义的”，所以，在这种情况下，“运行时”并非一定要运行finally块。实现可以选择这样做也以可不这样做。

规范

避免从finally块显式地引发异常（因方法调用而隐式地引发的异常可以被接受）。

要优先使用try/finally而不是try/catch块来实现资源清理代码。

要在抛出的异常中描述异常为什么发生。如有可能，还要说明如何防范。

高级主题：Exception类继承

从C# 2.0开始，所有异常都派生自System.Exception类（从其他语言引发的异常类型如果不是从System.Exception派生的，会自动由一个对象“封装”）。所以，它们都可以用catch(System.Exception exception)块进行处理。然而，一个更好的做法是编写一个专门的catch块来处理更具体的派生类型（如System.FormatException），从而获取有关异常的具体信息，有的放矢地处理，并避免使用大量条件逻辑来判断具体发生了什么类型的异常。

这正是C#规定catch块必须从“最具体”到“最不具体”排列的原因。例如，用于捕捉System. Exception的catch语句不能出现在捕捉System.FormatException的catch语句之前，因为System.FormatException比System.Exception更具体。

一个方法可以引发许多异常类型。表4-2总结了.NET Framework中的一些较为常见的类型。

表4-2 常见的异常类型

异常类型	描 述
System.Exception	这是最“基本”的异常，其他所有异常类型都从它派生
System.ArgumentException	传给方法的参数无效
System.ArgumentNullException	不应该为null的参数为null
System.ApplicationException	应避免用这个异常类型。初衷是想区分系统异常和应用程序异常。貌似合理，但实际行不通
System.FormatException	参数格式不符合调用的方法的参数规范
System.IndexOutOfRangeException	试图访问不存在的数组元素或其他集合元素
System.InvalidCastException	因无效的类型转换引发的异常
System.NotImplementedException	虽然找到了对应的方法签名，但该方法尚未完全实现
System.NullReferenceException	试图通过一个空的引用访问一个对象
System.ArithmeticException	发生了无效的数学运算，但其中不包括被零除
System.ArrayTypeMismatchException	试图将类型有误的元素存储到数组中
System.StackOverflowException	表明发生了非预期的深递归

高级主题：常规catch

可以指定一个不获取任何参数的catch块，如代码清单4-22所示。

代码清单4-22 常规catch块

```
...
try
{
    age = int.Parse(ageText);
    System.Console.WriteLine(
        "Hi {0}!  You are {1} months old.",
        firstName, age*12);
}
catch (System.FormatException exception)
{
    System.Console.WriteLine(
        "The age entered ,{0}, is not valid.",
        ageText);
    result = 1;
}
catch(System.Exception exception)
{
    System.Console.WriteLine(
        "Unexpected error:  {0}", exception.Message);
    result = 1;
}
catch
{
    System.Console.WriteLine(
        "Unexpected error!");
    result = 1;
}
finally
{
    System.Console.WriteLine("Goodbye {0}",
        firstName);
}
...
```

没有指定数据类型的catch块称为常规catch块（generic catch block），它等价于获取object数据类型的catch块，例如catch(object exception){...}。由于所有类最终都从object派生，所以没有数据类型的catch块必须放到最后。

常规catch块很少使用，因为没有办法捕捉有关异常的任何信息。此外，C#不允许引发object类型的一个异常，只有使用C++这样的语言写的库才允许任意类型的异常。

从C# 2.0开始，行为稍微有别于之前版本的C#。在C# 2.0中，假如遇到用另一种语言编写的代码，而且它会引发不是从System.Exception类派生的异常，那么该异常对象会被封装到一个System.Runtime.CompilerServices.RuntimeWrappedException中，后者从System.Exception派生。换言之，在C#程序集中，所有异常（无论它们是否从System.Exception派生）都会表现得和从System.Exception派生一样。

结果就是，用于捕捉System.Exception的catch块会捕捉之前的块没有捕捉到的所有异常， 同时，System.Exception catch块之后的一个常规catch块永远得不到调用。所以，从C# 2.0开始，假如在捕捉System.Exception的catch块之后添加了一个常规catch块，编译器就会报告一条警告消息[①]，指出常规catch块永远不会执行。

① 具体的警告消息是“上一个catch子句已捕获所有异常。引发的所有非异常均被包装在System.Runtime.CompilerServices.RuntimeWrappedException中”，其中的“非异常”翻译有误，实际应为“非System.Exception派生的异常”。——译者注