

(续表)

源 类 型	目 的 类 型
ushort	int、uint、long、ulong、float、double、decimal、BigInteger
int	long、float、double、decimal、BigInteger
uint	long、ulong、float、double、decimal、BigInteger
long、ulong	float、double、decimal、BigInteger
float	double、BigInteger
char	ushort、int、uint、long、ulong、float、double、decimal、BigInteger

注意，只能从较小的整数类型隐式地转换为较大的整数类型，不能从较大的整数类型隐式地转换为较小的整数类型。也可以在整数和浮点数之间转换，然而，其规则略有不同。尽管可以在相同大小的类型之间转换，如 `int/uint` 转换为 `float`，`long/ulong` 转换为 `double`，但是也可以从 `long/ulong` 转换回 `float`。这样做可能会丢失 4 个字节的数据，但这仅表示得到的 `float` 值比使用 `double` 得到的值精度低，编译器认为这是一种可以接受的错误，而其值的大小不会受到影响。无符号的变量可以转换为有符号的变量，只要无符号的变量值的大小在有符号的变量的范围之内即可。

在隐式地转换值类型时，可空类型需要考虑其他因素：

- 可空类型隐式地转换为其他可空类型，应遵循表 7-5 中非可空类型的转换规则。即 `int?` 隐式地转换为 `long?`、`float?`、`double?` 和 `decimal?`。
- 非可空类型隐式地转换为可空类型也遵循表 7-5 中的转换规则，即 `int` 隐式地转换为 `long?`、`float?`、`double?` 和 `decimal?`。
- 可空类型不能隐式地转换为非可空类型，此时必须进行显式转换，如下一节所述。这是因为可空类型的值可以是 `null`，但非可空类型不能表示这个值。

2. 显式转换

有许多场合不能隐式地转换类型，否则编译器会报告错误。下面是不能进行隐式转换的一些场合：

- `int` 转换为 `short` —— 会丢失数据。
- `int` 转换为 `uint` —— 会丢失数据。
- `uint` 转换为 `int` —— 会丢失数据。
- `float` 转换为 `int` —— 会丢失小数点后面的所有数据。
- 任何数字类型转换为 `char` —— 会丢失数据。
- `decimal` 转换为任何数字类型 —— 因为 `decimal` 类型的内部结构不同于整数和浮点数。
- `int?` 转换为 `int` —— 可空类型的值可以是 `null`。

但是，可以使用 `cast` 显式地执行这些转换。在把一种类型强制转换为另一种类型时，要有意地迫使编译器进行转换。类型强制转换的一般语法如下：

```
long val = 30000;
int i = (int)val;    // A valid cast. The maximum int is 2147483647
```

这表示,把强制转换的目标类型名放在要转换的值之前的圆括号中。对于熟悉 C 的程序员,这是类型强制转换的典型语法。对于熟悉 C++ 类型强制转换关键字(如 `static_cast`)的程序员,这些关键字在 C# 中不存在,必须使用 C 风格的旧语法。

这种类型强制转换是一种比较危险的操作,即使在从 `long` 转换为 `int` 这样简单的类型强制转换过程中,如果原来 `long` 的值比 `int` 的最大值还大,就会出问题:

```
long val = 3000000000;
int i = (int)val;           // An invalid cast. The maximum int is 2147483647
```

在本例中,不会报告错误,也得不到期望的结果。如果运行上面的代码,输出结果存储在 `i` 中,则其值为:

```
-1294967296
```

最好假定显式转换强制转换不会给出希望的结果。如前所述, C# 提供了一个 `checked` 运算符,使用它可以测试操作是否会导致算术溢出。使用 `checked` 运算符可以检查类型强制转换是否安全,如果不安全,就会迫使运行库抛出一个溢出异常:

```
long val = 3000000000;
int i = checked((int)val);
```

记住,所有的显式类型强制转换都可能不安全,在应用程序中应包含这样的代码,处理可能失败的类型强制转换。第 15 章将使用 `try` 和 `catch` 语句引入结构化异常处理。

使用类型强制转换可以把大多数基元数据类型从一种类型转换为另一种类型。例如,给 `price` 加上 0.5,再把结果强制转换为 `int`:

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

这么做的代价是把价格四舍五入为最接近的美元数。但在这个转换过程中,小数点后面的所有数据都会丢失。因此,如果要使用这个修改过的价格进行更多的计算,最好不要使用这种转换;如果要输出全部计算完或部分计算完的近似值,且不希望用小数点后面的多位数据去麻烦用户,这种转换就很好。

下面的例子说明了把无符号整数转换为 `char` 时,会发生的情况:

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

结果是 ASCII 码为 43 的字符,即“+”号。可以尝试数字类型(包括 `char`)之间的任何转换,这种转换是可行的,例如,把 `decimal` 转换为 `char`,或把 `char` 转换为 `decimal`。

值类型之间的转换并不仅限于孤立的变量。还可以把类型为 `double` 的数组元素转换为类型为 `int` 的结构成员变量:

```
struct Item Details
{
    public string Description;
    public int ApproxPrice;
```

```

    }

    //...

    double[] Prices = { 25.30, 26.20, 27.40, 30.00 };

    ItemDetails id;
    id.Description = "Hello there." ;
    id.ApproxPrice = (int) (Prices[0] + 0.5);

```

要把一个可空类型转换为非可空类型，或转换为另一个可空类型，其中可能会丢失数据，就必须使用显式的类型强制转换。甚至在底层基本类型相同的元素之间进行转换时，也要使用显式的类型强制转换，例如，`int?`转换为`int`，或`float?`转换为`float`。这是因为可空类型的值可以是`null`，非可空类型不能表示这个值。只要可以在两种等价的非可空类型之间进行显式的类型强制转换，对应可空类型之间显式的类型强制转换就可以进行。但如果从可空类型强制转换为非可空类型，且变量的值是`null`，就会抛出`InvalidOperationException`异常。例如：

```

int? a = null;
int b = (int)a;    // Will throw exception

```

小心地使用显式的类型强制转换，就可以把简单值类型的任何实例转换为几乎任何其他类型。但在进行显式的类型转换时有一些限制，就值类型来说，只能在数字、`char`类型和`enum`类型之间转换。不能直接把布尔型强制转换为其他类型，也不能把其他类型转换为布尔型。

如果需要在数字和字符串之间转换，就可以使用.NET类库中提供的一些方法。`Object`类实现了一个`ToString()`方法，该方法在所有的.NET预定义类型中都进行了重写，并返回对象的字符串表示：

```

int i = 10;
string s = i.ToString();

```

同样，如果需要分析一个字符串，以检索一个数字或布尔值，就可以使用所有预定义值类型都支持的`Parse()`方法：

```

string s = "100 ";
int i = int.Parse(s);
Console.WriteLine(i + 50);    // Add 50 to prove it is really an int

```

注意，如果不能转换字符串(例如，要把字符串`Hello`转换为一个整数)，`Parse()`方法就会通过抛出一个异常注册一个错误。第15章将介绍异常。

7.2.2 装箱和拆箱

第2章介绍了所有类型，包括简单的预定义类型(如`int`和`char`)和复杂类型(如从`object`类型中派生的类和结构)。下面可以像处理对象那样处理面值：

```

string s = 10.ToString();

```

但是，C#数据类型可以分为在栈上分配内存的值类型和在堆上分配内存的引用类型。如果`int`不过是栈上一个4字节的值，该如何在它上面调用方法？

C#的实现方式是通过一个有点魔术性的方式，即装箱(boxing)。装箱和拆箱(unboxing)可以把值类型转换为引用类型，并把引用类型转换回值类型。这包含在7.5节中，因为这是基本的操作，即

把值强制转换为 `object` 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“箱子”。

该转换可以隐式地进行，如上面的例子所述。还可以显式地进行转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

拆箱用于描述相反的过程，其中以前装箱的值类型强制转换回值类型。这里使用术语“强制转换”，是因为这种转换是显式进行的。其语法类似于前面的显式类型转换：

```
int myIntNumber = 20;
object myObject = myIntNumber;           // Box the int
int mySecondNumber = (int)myObject;      // Unbox it back into an int
```

只能对以前装箱的变量进行拆箱。当 `myObject` 不是装箱后的 `int` 型时，如果执行最后一行代码，就会在运行期间抛出一个异常。

这里有一个警告。在拆箱时，必须非常小心，确保得到的值变量有足够的空间存储拆箱的值中的所有字节。例如，C# 的 `int` 只有 32 位，所以把 `long` 值(64 位)拆箱为 `int` 时，会导致一个 `InvalidCastException` 异常：

```
long myLongNumber = 333333423;
object myObject = (object)myLongNumber;
int myIntNumber = (int)myObject;
```

7.3 比较对象的相等性

在讨论了运算符并简要介绍了相等运算符后，就应考虑在处理类和结构的实例时，“相等”意味着什么。理解对象相等的机制对逻辑表达式的编程非常重要，另外，对实现运算符重载和类型强制转换也非常重要，本章后面将讨论运算符重载。

对象相等的机制有所不同，这取决于比较的是引用类型(类的实例)的比较还是值类型(基元数据类型，结构或枚举的实例)。下面分别介绍引用类型和值类型的相等性。

7.3.1 比较引用类型的相等性

`System.Object` 定义了 3 个不同的方法，来比较对象的相等性：`ReferenceEquals()`和两个版本的 `Equals()`。再加上比较运算符(`==`)，实际上有 4 种进行比较相等性的方式。这些方法有一些细微的区别，下面就介绍这些方法。

1. `ReferenceEquals()`方法

`ReferenceEquals()`是一个静态方法，测试两个引用是否引用类的同一个实例，特别是两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以 `System.Object` 的实现代码保持不变。如果提供的两个引用引用同一个对象实例，则 `ReferenceEquals()`总是返回 `true`；否则就返回 `false`。但是它认为 `null` 等于 `null`：

```
SomeClass x, y;
```

```

x = new SomeClass();
y = new SomeClass();
bool B1 = ReferenceEquals(null, null); // returns true
bool B2 = ReferenceEquals(null, x);    // returns false
bool B3 = ReferenceEquals(x, y);        // returns false because x and y
                                         // point to different objects

```

2. 虚拟的 Equals()方法

Equals()虚拟版本的 System.Object 实现代码也可以比较引用。但因为这个方法是虚拟的，所以可以在自己的类中重写它，从而按值来比较对象。特别是如果希望类的实例用作字典中的键，就需要重写这个方法，以比较相关值。否则，根据重写 Object.GetHashCode()的方式，包含对象的字典类要么不工作，要么工作的效率非常低。在重写 Equals()方法时要注意，重写的代码不会抛出异常。同理，这是因为如果抛出异常，字典类就会出问题，一些在内部调用这个方法的.NET 基类也可能出问题。

3. 静态的 Equals()方法

Equals()的静态版本与其虚拟实例版本的作用相同，其区别是静态版本带有两个参数，并对它们进行相等性比较。这个方法可以处理两个对象中有一个是 null 的情况，因此，如果一个对象可能是 null，这个方法就可以抛出异常，提供额外的保护。静态重载版本首先要检查它传递的引用是否为 null。如果它们都是 null，就返回 true(因为 null 与 null 相等)。如果只有一个引用是 null，它就返回 false。如果两个引用实际上引用了某个对象，它就调用 Equals()的虚拟实例版本。这表示在重写 Equals()的实例版本时，其效果相当于也重写了静态版本。

4. 比较运算符(==)

最好将比较运算符看作严格的值比较和严格的引用比较之间的中间选项。在大多数情况下，下面的代码表示正在比较引用：

```
bool b = (x == y); // x, y object references
```

但是，如果把一些类看作值，其含义就会比较直观，这是可以接受的。在这些情况下，最好重写比较运算符，以执行值的比较。后面将讨论运算符的重载，但它的一个明显例子是 System.String 类，Microsoft 重写了这个运算符，以比较字符串的内容，而不比较它们的引用。

7.3.2 比较值类型的相等性

在比较值类型的相等性时，采用与引用类型相同的规则：ReferenceEquals()用于比较引用，Equals()用于比较值，比较运算符可以看作一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，进而才能对它们执行方法。另外，Microsoft 已经在 System.ValueType 类中重载了实例方法 Equals()，以便对值类型进行合适的相等性测试。如果调用 sA.Equals(sB)，其中 sA 和 sB 是某个结构的实例，则根据 sA 和 sB 是否在其所有的字段中包含相同的值，而返回 true 或 false。另一方面，在默认情况下，不能对自己的结构重载“=”运算符。在表达式中使用(sA == sB)会导致一个编译错误，除非在有问题的代码中为结构提供了“=”的重载版本。

另外，ReferenceEquals()在应用于值类型时，它总是返回 false，因为为了调用这个方法，值类型

需要装箱到对象中。即使使用下面的代码：

```
bool b = ReferenceEquals(v,v); // v is a variable of some value type
```

也会返回 `false`，因为在转换每个参数时，`v` 都会被单独装箱，这意味着会得到不同的引用。出于上述原因，调用 `ReferenceEquals()` 来比较值类型实际上没有什么意义，所以不能调用它。

尽管 `System.ValueType` 提供的 `Equals()` 的默认重写版本肯定足以应付绝大多数自定义的结构，但仍可以针对自己的结构再次重写它，以提高性能。另外，如果值类型包含作为字段的引用类型，就需要重写 `Equals()`，以便为这些字段提供合适的语义，因为 `Equals()` 的默认重写版本仅比较它们的地址。

7.4 运算符重载

本节将介绍为类或结构定义的另一种类型的成员：运算符重载。

C++ 开发人员应很熟悉运算符重载。但是，因为这个概念对 Java 和 Visual Basic 开发人员是全新的，所以这里要解释一下。C++ 开发人员可以直接跳到主要的运算符加载示例上。

运算符重载的关键是在对象上不能总是只调用方法或属性，有时还需要做一些其他工作，例如，对数值进行相加、相乘或逻辑操作(如比较对象)等。假定已经定义了一个表示数学矩阵的类。在数学领域中，矩阵可以相加和相乘，就像数字一样。所以可以编写下面的代码：

```
Matrix a, b, c;
// assume a, b and c have been initialized
Matrix d = c * (a + b);
```

通过重载运算符，就可以告诉编译器，“+”和“*”对 `Matrix` 对象进行什么操作，以编写上面的代码。如果不用支持运算符重载的语言编写代码，就必须定义一个方法，以执行这些操作。结果肯定不太直观，可能如下所示。

```
Matrix d = c.Multiply(a.Add(b));
```

学习到现在，像“+”和“*”这样的运算符只能用于预定义的数据类型，原因很简单：编译器知道所有常见的运算符对于这些数据类型的含义。例如，它知道如何把两个 `long` 加起来，或者如何对两个 `double` 执行相除操作，并生成合适的中间语言代码。但在定义自己的类或结构时，必须告诉编译器：什么方法可以调用，每个实例存储了什么字段等所有信息。同样，如果要对自定义的类使用运算符，就必须告诉编译器相关的运算符在这个类的上下文中的含义。此时就要定义运算符的重载。

要强调的另一个问题是重载不仅仅限于算术运算符。还需要考虑比较运算符 `==`、`<`、`>`、`!=`、`>=` 和 `<=`。例如，语句 `if(a==b)`。对于类，这个语句在默认状态下会比较引用 `a` 和 `b`。检测这两个引用是否指向内存中的同一个地址，而不是检测两个实例实际上是否包含相同的数据。对于 `string` 类，这种操作就会重写，于是比较字符串实际上就是比较每个字符串的内容。可以对自己的类进行这样的操作。对于结构，“`==`”运算符在默认状态下不做任何工作。试图比较两个结构，看看它们是否相等，就会产生一个编译错误，除非显式地重载了“`==`”，告诉编译器如何进行比较。

在许多情况下，重载运算符允许生成可读性更高、更直观的代码，包括：

- 在数学领域中，几乎包括所有的数学对象：坐标、矢量、矩阵、张量和函数等。如果编写一个程序执行某些数学或物理建模，就肯定会用类表示这些对象。
- 图形程序在计算屏幕上的位置时，也使用数学或相关的坐标对象。
- 表示大量金钱的类(例如，在财务程序中)。
- 字处理或文本分析程序也有表示语句、子句等的类，可以使用运算符合并语句(这是字符串连接的一种比较复杂的版本)。

但是，也有许多类型与运算符重载并不相关。不恰当地使用运算符重载，会使使用类型的代码很难理解。例如，把两个 `DateTime` 对象相乘，在概念上没有任何意义。

7.4.1 运算符的工作方式

为了解运算符是如何重载的，考虑一下在编译器遇到运算符时会发生什么情况很有用。用加法运算符(+)作为例子，假定编译器处理下面的代码：

```
int myInteger = 3;
uint myUnsignedInt = 2;
double myDouble = 4.0;
long myLong = myInteger + myUnsignedInt;
double myOtherDouble = myDouble + myInteger;
```

会发生什么情况：

```
long myLong = myInteger + myUnsignedInt;
```

编译器知道它需要把两个整数加起来，并把结果赋予一个 `long` 型变量。调用一个方法把数字加在一起时，表达式 `myInteger + myUnsignedInt` 是一种非常直观和方便的语法。该方法接受两个参数 `myInteger` 和 `myUnsignedInt`，并返回它们的和。所以编译器完成的任务与任何方法调用一样——它会根据参数类型查找最匹配的“+”运算符重载，这里是带两个整数参数的“+”运算符重载。与一般的重载方法一样，预定义的返回类型不会因为编译器所调用方法的哪个版本而影响编译器的选择。在本例中调用的重载方法接受两个 `int` 参数，返回一个 `int`，这个返回值随后会转换为一个 `long`。

下一行代码让编译器使用“+”运算符的另一个重载版本：

```
double myOtherDouble = myDouble + myInteger;
```

在这个例子中，参数是一个 `double` 类型的数据和一个 `int` 类型的数据，但“+”运算符没有带这种复合参数的重载形式，所以编译器认为，最匹配的“+”运算符重载是把两个 `double` 作为其参数的版本，并隐式地把 `int` 强制转换为 `double`。把两个 `double` 加在一起与把两个整数加在一起完全不同，浮点数存储为一个尾数和一个指数。把它们加在一起要按位移动一个 `double` 的尾数，从而使两个指数有相同的值，然后把尾数加起来，移动所得尾数的位，调整其指数，保证答案有尽可能高的精度。

现在，看看如果编译器遇到下面的代码，会发生什么：

```
Vector vect1, vect2, vect3;
// initialize vect1 and vect2
vect3 = vect1 + vect2;
vect1 = vect1 * 2;
```

其中, `Vector` 是结构, 稍后再定义它。编译器知道它需要把两个 `Vector` 实例加起来, 即 `vect1` 和 `vect2`。它会查找 “+” 运算符的重载, 重载的 “+” 运算符把两个 `Vector` 实例作为参数。

如果编译器找到这样的重载版本, 它就调用该运算符的实现代码。如果找不到, 它就要看看有没有可以用作最佳匹配的其他 “+” 运算符的重载, 例如某个运算符重载对应的两个参数是其他数据类型, 但可以隐式地转换为 `Vector` 实例。如果编译器找不到合适的运算符重载, 就会产生一个编译错误, 就像找不到其他方法调用的合适重载版本一样。

7.4.2 运算符重载的示例: `Vector` 结构

本节将开发一个结构 `Vector`, 来说明运算符重载, 这个 `Vector` 结构表示一个三维矢量。如果数学不是你的强项, 不必担心, 我们会使这个例子尽可能简单。三维矢量只是 3 个(double)数字的一个集合, 说明物体和原点之间的距离, 表示数字的变量是 `x`、`y` 和 `z`, `x` 表示物体与原点在 `x` 方向上的距离, `y` 表示它与原点在 `y` 方向上的距离, `z` 表示高度。把这 3 个数字组合起来, 就得到总距离。例如, 如果 `x=3.0`, `y=3.0`, `z=1.0`, 一般可以写作(3.0, 3.0, 1.0), 表示物体与原点在 `x` 方向上的距离是 3 个单位, 与原点在 `y` 方向上的距离是 3 个单位, 高度为 1 个单位。

矢量可以与矢量或数字相加或相乘。在这里我们使用术语 “标量”, 它是数字的数学用语——在 C# 中, 就是一个 `double`。相加的作用很明显。如果先移动(3.0, 3.0, 1.0)矢量对应的距离, 再移动(2.0, -4.0, -4.0)矢量对应的距离, 总移动量就是把这两个矢量加起来。矢量的相加指把每个坐标轴对应的元素分别相加, 因此得到(5.0, -1.0, -3.0)。此时, 数学表达式总是写成 `c=a+b`, 其中 `a` 和 `b` 是矢量, `c` 是结果矢量。这与使用 `Vector` 结构的方式一样。



这个例子将作为一个结构来开发, 而不是类, 但这并不重要。运算符重载用于结构和类时, 其工作方式是一样的。

下面是 `Vector` 的定义——包含成员字段、构造函数和重写的一个 `ToString()` 方法, 以便轻松地查看 `Vector` 的内容, 最后是运算符重载:



可从
wrox.com
下载源代码

```
namespace Wrox.ProCSharp.OOCSharp
{
    struct Vector
    {
        public double x, y, z;

        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public Vector(Vector rhs)
        {
            x = rhs.x;
            y = rhs.y;
            z = rhs.z;
        }
    }
}
```

```
public override string ToString()
{
    return "(" + x + ", " + y + ", " + z + ")";
}
```

代码下载 VectorStruct solution

这里提供了两个构造函数，通过传递每个元素的值，或者提供另一个复制其值的 `Vector`，来指定矢量的初始值。第二个构造函数带一个 `Vector` 参数，它通常称为复制构造函数，因为它们允许通过复制另一个实例来初始化一个类或结构实例。注意，为了简单起见，把字段设置为 `public`。也可以把它们设置为 `private`，编写相应的属性来访问它们，这样做不会改变这个程序的功能，只是代码会复杂一些。

下面是 `Vector` 结构的有趣部分——为“+”运算符提供支持的运算符重载：

```
public static Vector operator + (Vector lhs, Vector rhs)
{
    Vector result = new Vector(lhs);
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;

    return result;
}
```

运算符重载的声明方式与方法相同，但 `operator` 关键字告诉编译器，它实际上是一个自定义的运算符重载，后面是相关运算符的实际符号，在本例中就是“+”。返回类型是在使用这个运算符时获得的类型。在本例中，把两个矢量加起来会得到另一个矢量，所以返回类型也是 `Vector`。对于这个“+”运算符重载，返回类型与包含的类一样，但这种情况并不是必需的，在本示例中稍后将看到。两个参数就是要操作的对象。对于二元运算符(它带两个参数)，如“+”和“-”运算符，第一个参数是运算符左边的值，第二个参数是运算符右边的值。



一般把运算符左边的参数命名为 `lhs`，运算符右边的参数命名为 `rhs`。

C#要求所有的运算符重载都声明为 `public` 和 `static`，这表示它们与它们的类或结构相关联，而不是与某个特定实例相关联，所以运算符重载的代码体不能访问非静态类成员，也不能访问 `this` 标识符；这是可以的，因为参数提供了运算符执行其任务所需要知道的所有输入数据。

前面介绍了声明运算符“+”的语法，下面看看运算符内部的情况：

```
{
    Vector result = new Vector(lhs);
    result.x += rhs.x;
    result.y += rhs.y;
    result.z += rhs.z;

    return result;
}
```

这部分代码与声明方法的代码完全相同，显然，它返回一个矢量，其中包含前面定义的 lhs 和 rhs 的和，即把 x、y 和 z 成员分别相加。

下面需要编写一些简单的代码来测试 Vector 结构：

```
static void Main()
{
    Vector vect1, vect2, vect3;

    vect1 = new Vector(3.0, 3.0, 1.0);
    vect2 = new Vector(2.0, -4.0, -4.0);
    vect3 = vect1 + vect2;

    Console.WriteLine( " vect1 = " + vect1.ToString());
    Console.WriteLine( " vect2 = " + vect2.ToString());
    Console.WriteLine( " vect3 = " + vect3.ToString());
}
```

把这些代码另存为 Vectors.cs，编译并运行它，结果如下：

```
vect1 = ( 3, 3, 1 )
vect2 = ( 2, -4, -4 )
vect3 = ( 5, -1, -3 )
```

1. 添加更多的重载

矢量除了可以相加之外，还可以相乘、相减和比较它们的值。本节通过添加几个运算符重载，扩展了这个 Vector 例子。这并不是一个功能齐全的真实 Vector 类型，但足以说明运算符重载的其他方面了。首先要重载乘法运算符，以支持标量和矢量的相乘以及矢量和矢量的相乘。

矢量乘以标量只意味着矢量的元素分别与标量相乘，例如， $2 \times (1.0, 2.5, 2.0)$ 就等于 $(2.0, 5.0, 4.0)$ 。相关的运算符重载如下所示：



可从
wrox.com
下载源代码

```
public static Vector operator * (double lhs, Vector rhs)
{
    return new Vector(lhs * rhs.x, lhs * rhs.y, lhs * rhs.z);
}
```

[代码下载 VectorStructMoreOverloads.sln](#)

但这还不够，如果 a 和 b 声明为 Vector 类型，就可以编写下面的代码：

```
b = 2 * a;
```

编译器会隐式地把整数 2 转换为 double 类型，以匹配运算符重载的签名。但不能编译下面的代码：

```
b = a * 2;
```

编译器处理运算符重载的方式和方法重载是一样的。它会查看给定运算符的所有可用重载，找到与之最匹配的那个运算符重载。上面的语句要求第一个参数是 Vector，第二个参数是整数，或者可以隐式转换为整数的其他数据类型。我们没有提供这样一个重载。有一个运算符重载，其参数依次是一个 double 和一个 Vector，但编译器不能交换参数的顺序，所以这是不行的。还需要显式地定义

一个运算符重载，其参数依次是一个 `Vector` 和一个 `double`，有两种方式可以实现这样的运算符重载。第一种方式对矢量乘法进行分解，和处理所有运算符的方式一样，显式执行矢量相乘操作：

```
public static Vector operator * (Vector lhs, double rhs)
{
    return new Vector(rhs * lhs.x, rhs * lhs.y, rhs * lhs.z);
}
```

假定已经编写了实现相乘操作的代码，最好重用该代码：

```
public static Vector operator * (Vector lhs, double rhs)
{
    return rhs * lhs;
}
```

这段代码会告诉编译器，如果有 `Vector` 和 `double` 的相乘操作，编译器就颠倒参数的顺序，调用另一个运算符重载。本章的示例代码使用第2个版本，因为它看起来比较简洁同时阐述了该行为的思想。利用这个版本可以编写出可维护性更好的代码，因为不需要复制代码，就可在两个独立的重载中执行相乘操作。

下一个要重载的乘法运算符支持矢量相乘。在数学上，矢量相乘有两种方式，但这里我们感兴趣的是点积或内积，其结果实际上是一个标量。这就是我们介绍这个例子的原因，所以算术运算符不必返回与定义它们的类相同的类型。

在数学术语中，如果有两个矢量 (x, y, z) 和 (X, Y, Z) ，其内积就定义为 $x \cdot X + y \cdot Y + z \cdot Z$ 的值。两个矢量这样相乘很奇怪，但这实际上很有用，因为它可以用于计算各种其他的数。当然，如果要使用 `Direct3D` 或 `DirectDraw` 编写代码来显示复杂的 3D 图形，那么在计算对象放在屏幕上的什么位置时，中间常常需要编写代码来计算矢量的内积。这里我们关心的是使用 `Vector` 编写出 `double X = a · b`，其中 `a` 和 `b` 是两个矢量对象，并计算出它们的点积。相关的运算符重载如下所示：

```
public static double operator * (Vector lhs, Vector rhs)
{
    return lhs.x * rhs.x + lhs.y * rhs.y + lhs.z * rhs.z;
}
```

理解了算术运算符后，就可以用一个简单的测试方法来检验它们是否能正常运行：

```
static void Main()
{
    // stuff to demonstrate arithmetic operations
    Vector vect1, vect2, vect3;
    vect1 = new Vector(1.0, 1.5, 2.0);
    vect2 = new Vector(0.0, 0.0, -10.0);

    vect3 = vect1 + vect2;

    Console.WriteLine("vect1 = " + vect1);
    Console.WriteLine("vect2 = " + vect2);
    Console.WriteLine("vect3 = vect1 + vect2 = " + vect3);
    Console.WriteLine("2 * vect3 = " + 2 * vect3);
    vect3 += vect2;

    Console.WriteLine("vect3+=vect2 gives " + vect3);
}
```

```

    vect3 = vect1 * 2;

    Console.WriteLine("Setting vect3=vect1 * 2 gives " + vect3);

    double dot = vect1 * vect3;

    Console.WriteLine("vect1 * vect3 = " + dot);
}

```

运行代码(Vectors2.cs), 得到如下所示的结果:

VECTORS2

```

vect1 = ( 1, 1.5, 2 )
vect2 = ( 0, 0, -10 )
vect3 = vect1 + vect2 = ( 1, 1.5, -8 )
2*vect3 = ( 2, 3, -16 )
vect3 += vect2 gives ( 1, 1.5, -18 )
Setting vect3 = vect1 * 2 gives ( 2, 3, 4 )
vect1 * vect3 = 14.5

```

这说明, 运算符重载会给出正确的结果, 但如果仔细看看测试代码, 就会惊奇地注意到, 实际上它使用的是没有重载的运算符——相加赋值运算符(+):

```

    vect3 += vect2;

    Console.WriteLine("vect3 += vect2 gives" + vect3);

```

虽然“+”一般用作单个运算符, 但实际上它对应的操作分为两步: 相加和赋值。与 C++ 语言不同, C# 不允许重载“=”运算符, 但如果重载“+”运算符, 编译器就会自动使用“+”运算符的重载来执行“+=”运算符的操作。-=、&=、*=和/=等所有赋值运算符也遵循此规则。

2. 比较运算符的重载

本章前面介绍过, C# 中有 6 个比较运算符, 它们分为 3 对:

- ==和!=
- >和<
- >=和<=

C# 语言要求成对重载比较运算符。即, 如果重载了“=”, 也就必须重载“!="; 否则会产生编译错误。另外, 比较运算符必须返回布尔类型的值。这是它们与算术运算符的根本区别。例如, 两个数相加或相减的结果, 理论上取决于数的类型。已经看到两个 Vector 对象的相乘会得到一个标量。另一个例子是 .NET 基类 System.DateTime, 两个 DateTime 实例相减, 得到的结果不是一个 DateTime, 而是一个 System.TimeSpan 实例。相比之下, 如果比较运算得到的不是布尔类型的值, 就没有任何意义。



在重载“=”和“!="时, 还必须重载从 System.Object 中继承的 Equals() 和 GetHashCode() 方法, 否则会产生一个编译警告。原因是 Equals() 方法应实现与“=”运算符相同类型的相等逻辑。

除了这些区别外,重载比较运算符所遵循的规则与重载算术运算符相同。但比较两个数并不像想象的那么简单。例如,如果只比较两个对象引用,就是比较存储对象的内存地址。比较运算符很少进行这样的比较,所以必须编写代码重载运算符,比较对象的值,并返回相应的布尔结果。下面对 Vector 结构重载 “=” 和 “!=” 运算符。首先是实现 “=” 重载的代码:

```
public static bool operator == (Vector lhs, Vector rhs)
{
    if (lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z)
        return true;
    else
        return false;
}
```

这种方式仅根据矢量元素的值,来对它们进行相等性比较。对于大多数结构,这就是我们希望的,但在某些情况下,可能需要仔细考虑相等的含义。例如,如果有嵌入的类,那么是应比较引用是否指向同一个对象(浅度比较),还是应比较对象的值是否相等(深度比较)?

浅度比较是比较对象是否指向内存中的同一个位置,而深度比较是比较对象的值和属性是否相等。应根据具体情况进行相等检查,从而有助于确定要验证什么。



不要通过调用从 System.Object 中继承的 Equals() 方法的实例版本,来重载比较运算符。如果这么做,在 objA 是 null 时判断(objA=objB),就会产生一个异常,因为.NET 运行库会试图判断 null.Equals(objB)。采用其他方法(重写 Equals() 方法以调用比较运算符)比较安全。

还需要重载运算符 “!=”, 采用的方式如下:

```
public static bool operator != (Vector lhs, Vector rhs)
{
    return ! (lhs == rhs);
}
```

像往常一样,用一些测试代码检查重写方法的工作情况。这次定义 3 个 Vector 对象,并进行比较:



可从
wrox.com
下载源代码

```
static void Main()
{
    Vector vect1, vect2, vect3;

    vect1 = new Vector(3.0, 3.0, -10.0);
    vect2 = new Vector(3.0, 3.0, -10.0);
    vect3 = new Vector(2.0, 3.0, 6.0);

    Console.WriteLine("vect1==vect2 returns " + (vect1==vect2));
    Console.WriteLine("vect1==vect3 returns " + (vect1==vect3));
    Console.WriteLine("vect2==vect3 returns " + (vect2==vect3));

    Console.WriteLine();

    Console.WriteLine("vect1!=vect2 returns " + (vect1!=vect2));
    Console.WriteLine("vect1!=vect3 returns " + (vect1!=vect3));
    Console.WriteLine("vect2!=vect3 returns " + (vect2!=vect3));
}
```

编译这些代码(代码下载中的 Vectors3.cs 示例), 会得到以下编译器警告, 因为我们没有为 Vector 重写 Equals()。对于本例, 这并不重要, 所以忽略它。

```
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
for Microsoft (R) .NET Framework version 4.0
Copyright (C) Microsoft Corporation. All rights reserved.

Vectors3.cs(5,11): warning CS0660: 'Wrox.ProCSharp.OOCSharp.Vector' defines
operator == or operator != but does not override Object.Equals(object o)
Vectors3.cs(5,11): warning CS0661: 'Wrox.ProCSharp.OOCSharp.Vector' defines
operator == or operator != but does not override Object.GetHashCode()
```

在命令行上运行该示例, 生成如下结果:

```
VECTORS3

vect1==vect2 returns True
vect1==vect3 returns False
vect2==vect3 returns False

vect1!=vect2 returns False
vect1!=vect3 returns True
vect2!=vect3 returns True
```

3. 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 7-6 所示。

表 7-6

类 别	运 算 符	限 制
算术二元运算符	+, *, /, -, %	无
算术一元运算符	+, -, ++, --	无
按位二元运算符	&, , ^, <<, >>	无
按位一元运算符	!, ~, true, false	true 和 false 运算符必须成对重载
比较运算符	==, !=, >=, <, <=, >	必须成对重载
赋值运算符	+=, -=, *=, /=, >>=, <<=, %=, &=, =, ^=	不能显式地重载这些运算符, 在重写单个运算符 (如+, -, %等)时, 它们会被隐式地重写
索引运算符	[]	不能直接重载索引运算符。第 2 章介绍的索引器成员类型允许在类和结构上支持索引运算符。
数据类型转换运算符	()	不能直接重载类型强制转换运算符。用户定义的类型强制转换(本章的后面介绍)允许定义定制的类型强制转换行为

7.5 用户定义的类型强制转换

本章前面介绍了如何在预定义的数据类型之间转换数值，这通过类型强制转换过程来完成。C# 允许进行两种不同数据类型的强制转换：隐式强制转换和显式强制转换。

显式强制转换要在代码中显式地标记强制转换，其方法是在圆括号中写出目标数据类型：

```
int I = 3;
long l = I;           // implicit
short s = (short)I;   // explicit
```

对于预定义的数据类型，当类型强制转换可能失败或丢失某些数据时，需要显式强制转换。例如：

- 把 `int` 转换为 `short` 时，因为 `short` 可能不够大，不能包含对应 `int` 的数值。
- 把有符号的数据类型转换为无符号的数据类型时，如果有符号的变量包含一个负值，就会得到不正确的结果。
- 把浮点数转换为整数数据类型时，数字的小数部分会丢失。
- 把可空类型转换为非可空类型时，`null` 值会导致异常。

此时应在代码中进行显式强制转换，告诉编译器你知道这会有丢失数据的危险，因此编写代码时要把这种可能性考虑在内。

C# 允许定义自己的数据类型(结构和类)，这意味着需要某些工具支持在自定义的数据类型之间进行类型强制转换。方法是把类型强制转换运算符定义为相关类的一个成员运算符，类型强制转换运算符必须标记为隐式或显式，以说明希望如何使用它。我们应遵循与预定义的类型强制转换相同的规则，如果知道无论在源变量中存储什么值，类型强制转换总是安全的，就可以把它定义为隐式强制转换。然而，如果某些数值可能会出错，如丢失数据或抛出异常，就应把数据类型转换定义为显式强制转换。



如果源数据值会使类型强制转换失败，或者可能会抛出异常，就应把任何自定义类型强制转换定义为显式强制转换。

定义类型强制转换的语法类似于本章前面介绍的重载运算符。这并不是偶然的，类型强制转换在某种情况下可以看作是一种运算符，其作用是从源类型转换为目标类型。为了说明这种语法，下面的代码是从本节后面介绍的结构 `Currency` 示例中节选的：

```
public static implicit operator float (Currency value)
{
    // processing
}
```

运算符的返回类型定义了类型强制转换操作的目标类型，它有一个参数，即要转换的源对象。这里定义的类型强制转换可以隐式地把 `Currency` 型的值转换为 `float` 型。注意，如果数据类型转换声明为隐式，编译器就可以隐式或显式地使用这个转换。如果数据类型转换声明为显式，编译器就只能显式地使用它。与其他运算符重载一样，类型强制转换必须同时声明为 `public` 和 `static`。



C++开发人员应注意,这种情况与 C++中的用法不同,在 C++中,类型强制转换针对于类的实例成员。

7.5.1 实现用户定义的类型强制转换

本节将在示例 SimpleCurrency(和往常一样,其代码可以下载)中介绍隐式和显式的用户定义的类型强制转换的用法。在这个示例中,定义一个结构 Currency,它包含一个正的 USD(\$)金额。C#为此提供了 decimal 类型,但如果要进行比较复杂的财务处理,仍可以编写自己的结构和类来表示相应的金额,在这样的类上实现特定的方法。



类型强制转换的语法对于结构和类是一样的。本示例定义了一个结构,但如果把 Currency 声明为类,也是可以的。

首先, Currency 结构的定义如下所示。



可从
wrox.com
下载源代码

```
struct Currency
{
    public uint Dollars;
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
        this.Dollars = dollars;
        this.Cents = cents;
    }

    public override string ToString()
    {
        return string.Format(" ${0}.{1, - 2:00} ", Dollars,Cents);
    }
}
```

代码段 SimpleCurrency/Program.cs

Dollars 和 Cents 字段使用无符号的数据类型,可以确保 Currency 实例只能包含正值。这样限制,是为了在后面说明显式强制转换的一些要点。可以像这样使用一个类来存储公司员工的薪水信息。人们的薪水不会是负值!为了使类比较简单,我们把字段声明为 public,但通常应把它们声明为 private,并为 Dollars 和 Cents 字段定义相应的属性。

下面先假定要把 Currency 实例转换为 float 值,其中 float 值的整数部分表示美元。换言之,应编写下面的代码:

```
Currency balance = new Currency(10,50);
float f = balance; // We want f to be set to 10.5
```

为此,需要定义一种类型强制转换。给 Currency 的定义添加下述代码:

```
public static implicit operator float (Currency value)
{
    return value.Dollars + (value.Cents/100.0f);
}
```

这种类型强制转换是隐式的。在本例中这是一个合理的选择，因为在 `Currency` 的定义中，可以存储在 `Currency` 中的值也都可以存储在 `float` 中。在这种强制转换中，不应出现任何错误。



这里有一点欺骗性：实际上，当把 `uint` 转换为 `float` 时，精确度是降低，但 Microsoft 认为这种错误并不重要，因此把从 `uint` 到 `float` 的强制转换都当做隐式转换。

但是，如果把 `float` 型转换为 `Currency` 型，就不能保证转换肯定成功了：`float` 型可以存储负值，而 `Currency` 实例不能，`float` 型存储的数值的数量级要比 `Currency` 型的(`uint`) `Dollars` 字段大得多。所以，如果 `float` 值包含一个不合适的值，把它转换为 `Currency` 型就会得到意想不到的结果。因此，从 `float` 型转换到 `Currency` 型就应定义为显式转换。下面是我们的第一次尝试，这次不会得到正确的结果，但有助于解释原因：

```
public static explicit operator Currency (float value)
{
    uint dollars = (uint)value;
    ushort cents = (ushort)((value - dollars) * 100);
    return new Currency(dollars, cents);
}
```

下面的代码可以成功编译：

```
float amount = 45.63f;
Currency amount2 = (Currency)amount;
```

但是，下面的代码会抛出一个编译错误，因为它试图隐式地使用一个显式的类型强制转换：

```
float amount = 45.63f;
Currency amount2 = amount; // wrong
```

把数据类型强制转换声明为显式，就是警告开发人员要小心，因为可能会丢失数据。但这不是我们希望的 `Currency` 结构的行为方式。下面编写一个测试程序，并运行该示例。其中有一个 `Main()` 方法，它实例化了一个 `Currency` 结构，并试图进行几个转换。在这段代码的开头，以两种不同的方式计算 `balance` 的值(因为要使用它们来说明后面的内容)：

```
static void Main()
{
    try
    {
        Currency balance = new Currency(50,35);

        Console.WriteLine(balance);
        Console.WriteLine("balance is" + balance);
        Console.WriteLine("balance is (using ToString())" + balance.ToString());

        float balance2= balance;

        Console.WriteLine("After converting to float, =" + balance2);

        balance = (Currency) balance2;
```

```

        Console.WriteLine("After converting back to Currency, =" + balance);
        Console.WriteLine("Now attempt to convert out of range value of" +
            "- $50.50 to a Currency:");
    }
    checked
    {
        balance = (Currency) (- 50.50);
        Console.WriteLine("Result is" + balance.ToString());
    }
}
catch(Exception e)
{
    Console.WriteLine("Exception occurred:" + e.Message);
}
}

```

注意，所有的代码都放在一个 `try` 块中，来捕获在类型强制转换过程中发生的任何异常。在 `checked` 块中还添加了把超出范围的值转换为 `Currency` 的测试代码，试图捕获负值。运行这段代码，得到如下所示的结果：

SIMPLECURRENCY

```

50.35
Balance is $50.35
Balance is (using ToString()) $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of - $100.00 to a Currency:
Result is $4294967246.00

```

这个结果表示代码并没有像我们希望的那样工作。首先，从 `float` 转换回 `Currency` 得到一个错误的结果\$50.34，而不是\$50.35。其次，在试图转换明显超出范围的值时，没有生成异常。

第一个问题是由舍入错误引起的。如果类型强制转换用于把 `float` 转换为 `uint`，计算机就会截去多余的数字，而不是四舍五入它。计算机以二进制方式存储数字，而不是十进制，小数部分 0.35 不能用二进制小数来精确表示(像 $1/3$ 这样的分数不能精确地表示为十进制小数，它应等于循环小数 0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式精确地表示。把该数字乘以 100，就会得到一个小于 35 的数字，它截去了 34 美分。显然在本例中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能的四舍五入操作。幸运的是，Microsoft 编写了一个类 `System.Convert` 来完成该任务。`System.Convert` 对象包含大量的静态方法来完成各种数字转换，我们需要使用的是 `Convert.ToUInt16()`。注意，在使用 `System.Convert` 类的方法时会造成额外的性能损失，所以只应在需要时才使用它们。

下面看看为什么没有抛出期望的溢出异常。此处的问题是溢出异常实际发生的位置根本不在 `Main()` 例程中——它是在强制转换运算符的代码中发生的，该代码在 `Main()` 方法中调用，而且没有标记为 `checked`。

其解决方法是确保类型强制转换本身也在 `checked` 环境下进行。进行了这两个修改后，修订后的转换代码如下所示。

```

public static explicit operator Currency (float value)

```

```

checked
{
    uint dollars = (uint)value;
    ushort cents = Convert.ToUInt16((value - dollars) * 100);
    return new Currency(dollars, cents);
}

```

注意, 使用 `Convert.ToUInt16()` 计算数字的美分部分, 如上所示, 但没有使用它计算数字的美元部分。在计算美元值时不需要使用 `System.Convert`, 因为在此我们希望截去 `float` 值。

值得注意的是, `System.Convert` 类的方法还执行它们自己的溢出检查。因此对于本例的情况, 不需要把对 `Convert.ToUInt16()` 的调用放在 `checked` 环境下。但把 `value` 显式地强制转换为美元值仍需要 `checked` 环境。

这里没有给出这个新的 `checked` 强制转换的结果, 因为在本节后面还要对 `SimpleCurrency` 示例进行一些修改。

如果定义了一种使用非常频繁的类型强制转换, 其性能也非常好, 就可以不进行任何错误检查。如果对用户定义的类型强制转换和没有检查的错误进行了清晰的说明, 这也是一种合法的解决方案。

1. 类之间的类型强制转换

`Currency` 示例仅涉及与 `float` (一种预定义的数据类型) 来回转换的类。但类型转换不一定会涉及任何简单的数据类型。定义不同结构或类的实例之间的类型强制转换是完全合法的, 但有两个限制:

- 如果某个类派生自另一个类, 就不能定义这两个类之间的类型强制转换 (这些类型的类型转换已经存在)。
- 类型强制转换必须在源数据类型或目标数据类型的内部定义。

要说明这些要求, 假定有如图 7-1 所示的类层次结构。

换言之, 类 `C` 和 `D` 间接派生于 `A`。在这种情况下, 在 `A`、`B`、`C` 或 `D` 之间唯一的自定义类型强制转换就是类 `C` 和 `D` 之间的转换, 因为这些类并没有互相派生。这段代码如下所示 (假定希望类型强制转换是显式的, 这是在用户定义的类之间定义类型强制转换的通常情况):

```

public static explicit operator D(C value)
{
    // and so on
}
public static explicit operator C(D value)
{

```

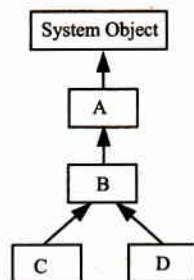


图 7-1

```
// and so on
}
```

对于这些类型强制转换，可以选择放置定义的地方——在 C 的类定义内部，或者在 D 的类定义内部，但不能在其他地方定义。C# 要求把类型强制转换的定义放在源类(或结构)或目标类(或结构)的内部。它的副作用是不能定义两个类之间的类型强制转换，除非至少可以编辑其中一个类的源代码。这是因为，这样可以防止第三方把类型强制转换引入类中。

一旦在一个类的内部定义了类型强制转换，就不能在另一个类中定义相同的类型强制转换。显然，对于每一种转换只能有一种类型强制转换，否则编译器就不知道该选择哪个类型强制转换了。

2. 基类和派生类之间的类型强制转换

要了解这些类型强制转换是如何工作的，首先看看源和目标的数据类型都是引用类型的情况。考虑两个类 `MyBase` 和 `MyDerived`，其中 `MyDerived` 直接或间接派生自 `MyBase`。

首先是从 `MyDerived` 到 `MyBase` 的转换，代码如下(假定可以使用构造函数)：

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

在本例中，是从 `MyDerived` 隐式地强制转换为 `MyBase`。这是可行的，因为对类型 `MyBase` 的任何引用都可以引用 `MyBase` 类的对象或派生自 `MyBase` 的对象。在 OO 编程中，派生类的实例实际上是基类的实例，但加上了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上定义了。

下面看看另一种方式，编写下面的代码：

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject;    // OK
MyDerived derivedCopy2 = (MyDerived) baseObject;        // Throws exception
```

上面的代码都是合法的 C# 代码(从句法的角度来看，它是合法的)，它说明了把基类强制转换为派生类。但是，在执行时最后一条语句会抛出一个异常。在进行类型强制转换时，会检查被引用的对象。因为基类引用原则上可以引用一个派生类的实例，所以这个对象可能是要强制转换的派生类的一个实例。如果是这样，强制转换就会成功，派生的引用被设置为引用这个对象。但如果有问题对象不是派生类(或者派生于这个类的其他类)的一个实例，强制转换就会失败，并抛出一个异常。

注意，编译器已经提供了基类和派生类之间的强制转换，这种转换实际上并没有对有问题对象进行任何数据转换。如果要进行的转换是合法的，它们也仅是把新引用设置为对对象的引用。这些强制转换在本质上与用户定义的强制转换不同。例如，在前面的 `SimpleCurrency` 示例中，我们定义了 `Currency` 结构和 `float` 数之间的强制转换。在 `float` 到 `Currency` 的强制转换中，实际上实例化了一个新的 `Currency` 结构，并用要求的值初始化它。在基类和派生类之间的预定义强制转换则不是这样。如果实际上要把 `MyBase` 实例转换为真实的 `MyDerived` 对象，该对象的值根据 `MyBase` 实例的内容来确定，就不能使用类型强制转换语法。最合适的选项通常是定义一个派生类的构造函数，它以基类的实例作为参数，让这个构造函数完成相关的初始化：

```
class DerivedClass: BaseClass
{
```

```
public DerivedClass(BaseClass rhs)
{
    // initialize object from the Base instance
}
// etc.
```

3. 装箱和拆箱数据类型强制转换

前面主要讨论了基类和派生类之间的数据类型强制转换，其中，基类和派生类都是引用类型。类似的规则也适用于强制转换值类型，尽管在转换值类型时，不可能仅仅复制引用，还必须复制一些数据。

当然，不能从结构或基元值类型中派生。所以基本结构和派生结构之间的强制转换总是基元类型或结构与 `System.Object` 之间的转换(理论上可以在结构和 `System.ValueType` 之间进行强制转换，但一般很少这么做)。

从结构(或基本类型)到 `object` 的强制转换总是一种隐式的强制转换，因为这种的强制转换是从派生类型到基本类型的转换，即第2章简要介绍的装箱过程。例如，`Currency` 结构：

```
Currency balance = new Currency(40,0);
object baseCopy = balance;
```

在执行上述隐式的强制转换时，`balance` 的内容被复制到堆上，放在一个装箱的对象上，`BaseCopy` 对象引用被设置为该对象。实际上在后台发生的情况是：在最初定义 `Currency` 结构时，.NET Framework 隐式地提供另一个(隐藏的)类，即装箱的 `Currency` 类，它包含与 `Currency` 结构相同的所有字段，但它是一个引用类型，存储在堆上。无论定义的这个值类型是一个结构，还是一个枚举，定义它时都存在类似的装箱引用类型，对应于所有的基元值类型，如 `int`、`double` 和 `uint` 等。不能也不必在源代码中直接通过编程访问某些装箱类，但在把一个值类型强制转换为 `object` 时，它们是在后台工作的对象。在隐式地把 `Currency` 转换为 `object` 时，会实例化一个装箱的 `Currency` 实例，并用 `Currency` 结构中的所有数据进行初始化。在上面的代码中，`baseCopy` 对象引用的就是这个已装箱的 `Currency` 实例。通过这种方式，就可以实现从派生类型到基本类型的强制转换，并且，值类型的语法与引用类型的语法一样。

强制转换的另一种方式称为拆箱。与在基本引用类型和派生引用类型之间的强制转换一样，这是一种显式的强制转换，因为如果要强制转换的对象不是正确的类型，就会抛出一个异常：

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject;           // OK
Currency derivedCopy2 = (Currency)baseObject;               // Exception thrown
```

上述代码的工作方式与前面的引用类型中的代码一样。把 `derivedObject` 强制转换为 `Currency` 会成功进行，因为 `derivedObject` 实际上引用的是装箱 `Currency` 实例——强制转换的过程是把已装箱的 `Currency` 对象的字段复制到一个新的 `Currency` 结构中。第二种强制转换会失败，因为 `baseObject` 没有引用已装箱的 `Currency` 对象。

在使用装箱和拆箱时，这两个过程都把数据复制到新装箱或拆箱的对象上，理解这一点非常重要。这样，例如，对装箱对象的操作就不会影响原始值类型的内容。

7.5.2 多重类型强制转换

在定义类型强制转换时必须考虑的一个问题是，如果在进行要求的数据类型转换时，C#编译器没有可用的直接强制转换方式，C#编译器就会寻找一种转换方式，把几种强制转换合并起来。例如，在 `Currency` 结构中，假定编译器遇到下面几行代码：

```
Currency balance = new Currency(10,50);
long amount = (long)balance;
double amountD = balance;
```

首先初始化一个 `Currency` 实例，再把它转换为一个 `long` 数。问题是不能定义这样的强制转换。但是，这段代码仍可以编译成功。因为编译器知道我们已经定义一个从 `Currency` 到 `float` 的隐式强制转换，而且它知道如何显式地从 `float` 强制转换为 `long`。所以它会把这行代码编译为中间语言(IL)代码，IL 代码首先把 `balance` 转换为 `float`，再把结果转换为 `long`。把 `balance` 转换为 `double` 型时，在上述代码的最后一行中也是这样。因为从 `Currency` 到 `float` 的强制转换和从 `float` 到 `double` 的预定义强制转换都是隐式的，所以可以在编写代码时把这种转换当作一种隐式转换。如果要显式地指定强制转换过程，则可以编写如下代码：

```
Currency balance = new Currency(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

但是，在大多数情况下，这会使代码变得比较复杂，因此是不必要的。相比之下，下面的代码会产生一个编译错误：

```
Currency balance = new Currency(10,50);
long amount = balance;
```

原因是编译器可以找到的最佳匹配的转换仍是首先转换为 `float`，再转换为 `long`。但从 `float` 到 `long` 的转换需要显式地指定。

并非所有这些转换都会带来太多的麻烦。毕竟转换的规则非常直观，主要是为了防止在开发人员不知情的情况下丢失数据。但是，在定义类型强制转换时如果不小心，编译器就有可能指定一条导致不期望的结果的路径。例如，假定编写 `Currency` 结构的其他小组成员要把一个 `uint` 转换为 `Currency`，其中该 `uint` 中包含了美分的总数(美分不是美元，因为我们不希望丢掉美元的小数部分)。为此应编写如下代码来实现强制转换：

```
public static implicit operator Currency (uint value)
{
    return new Currency(value/100u, (ushort)(value%100));
} // Do not do this!
```

注意，在这段代码中，第一个 100 后面的 `u` 可以确保把 `value/100u` 解释为一个 `uint` 数。如果写成 `value/100`，编译器就会把它解释为一个 `int` 型的值，而不是 `uint` 型的值。

在这段代码中清楚地标注了“不要这么做”。下面说明其原因。看看下面的代码段，它把包含 350 的一个 `uint` 转换为一个 `Currency`，再转换回 `uint`。那么在执行完这段代码后，`bal2` 中又将包含什么？

```
uint bal = 350;
Currency balance = bal;
```

```
uint bal2 = (uint)balance;
```

答案不是 350, 而是 3! 而且这是符合逻辑的。我们把 350 隐式地转换为 Currency, 得到的结果是 `balance.Dollars = 3, balance.Cents = 50`。然后编译器进行通常的操作, 为转换回 `uint` 指定最佳路径。`balance` 最终会被隐式地转换为 `float` 型(其值为 3.5), 然后显式地转换为 `uint` 型, 其值为 3。

当然, 在其他示例中, 转换为另一种数据类型后, 再转换回来有时会丢失数据。例如, 把包含 5.8 的 `float` 数转换为 `int` 数, 再转换回 `float` 数, 会丢失数字中的小数部分, 得到 5, 但原则上丢失数字的小数部分和一个整数被大于 100 的数整除的情况略有区别。Currency 现在成了一种相当危险的类, 它会对整数进行一些奇怪的操作。

问题是, 在转换过程中如何解释整数存在冲突。从 Currency 到 `float` 的强制转换会把整数 1 解释为 1 美元, 但从 `uint` 到 Currency 的强制转换会把这个整数解释为 1 美分, 这是很糟糕的一个示例。如果希望类易于使用, 就应确保所有的强制转换都按一种互相兼容的方式执行, 即这些转换直观上应得到相同的结果。在本例中, 显然要重新编写从 `uint` 到 Currency 的强制转换, 把整数值 1 解释为 1 美元:

```
public static implicit operator Currency (uint value)
{
    return new Currency(value, 0);
}
```

偶尔也会觉得这种新的转换方式可能根本不需要。但实际上这种转换方式可能非常有用。没有这种强制转换, 编译器在执行从 `uint` 到 Currency 的转换时, 就只能通过 `float` 来进行。此时直接转换的效率要高得多, 所以进行这种额外的转换强制会提高性能, 但需要确保它的结果与通过 `float` 进行转换得到的结果相同。在其他情况下, 也可以为不同的预定义数据类型分别定义强制转换, 让更多的转换隐式地执行, 而不是显式地执行, 但本例不是这样。

测试这种强制转换是否兼容, 应确定无论使用什么转换路径, 它是否都能得到相同的结果(而不是像在从 `float` 到 `int` 的转换过程中那样丢失数据)。Currency 类就是一个很好的示例。看看下面的代码:

```
Currency balance = new Currency(50, 35);
ulong bal = (ulong) balance;
```

目前, 编译器只能采用一种方式来完成这个转换: 把 Currency 隐式地转换为 `float`, 再显式地转换为 `ulong`。从 `float` 到 `ulong` 的转换需要显式转换, 本例就显式指定了这个转换, 所以编译是成功的。

但假定要添加另一种强制转换, 从 Currency 隐式地转换为 `uint`, 就需要修改 Currency 结构, 添加从 `uint` 到 Currency 的强制转换和从 Currency 到 `uint` 的强制转换。这段代码可以作为 SimpleCurrency2 示例:

```
public static implicit operator Currency (uint value)
{
    return new Currency(value, 0);
}

public static implicit operator uint (Currency value)
{
    return value.Dollars;
}
```

现在, 编译器从 Currency 转换到 `ulong` 可以使用另一条路径: 先从 Currency 隐式地转换为 `uint`,

再隐式地转换为 `ulong`。该采用哪条路径？C# 有一些严格的规则(本书不详细讨论这些规则，读者可参阅 MSDN 文档)，告诉编译器如何确定哪条是最佳路径。但最好自己设计类型强制转换，让所有的转换路径都得到相同的结果(但没有精确度的损失)，此时编译器选择哪条路径就不重要了(在本例中，编译器会选择 `Currency`→`uint`→`ulong` 路径，而不是 `Currency`→`float`→`ulong` 路径)。

为了测试 `SimpleCurrency2` 示例，给 `SimpleCurrency` 的测试程序添加如下代码：



可从
wrox.com
下载源代码

```
try
{
    Currency balance = new Currency(50,35);

    Console.WriteLine(balance);
    Console.WriteLine("balance is" + balance);
    Console.WriteLine("balance is (using ToString())" + balance.ToString());

    uint balance3 = (uint) balance;

    Console.WriteLine("Converting to uint gives" + balance3);
}
```

代码段 SimpleCurrency2/Program.cs

运行这个示例，得到如下所示的结果：

SIMPLECURRENCY2

```
50
balance is $50.35
balance is (using ToString()) $50.35
Converting to uint gives 50
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of - $50.50 to a Currency:
Result is $4294967246.00
```

这个结果显示了到 `uint` 的转换是成功的，但在转换过程中丢失了 `Currency` 的美分部分(小数部分)。把负的 `float` 类型强制转换为 `Currency` 也产生了预料中的溢出异常，因为 `float` 到 `Currency` 的强制转换本身定义了一个 `checked` 环境。

但是，这个输出结果也说明了进行强制转换时最后一个要注意的潜在问题：结果的第一行没有正确显示余额，显示了 50，而不是 \$50.35。在下面的代码中：

```
Console.WriteLine(balance);
Console.WriteLine(" balance is " + balance);
Console.WriteLine(" balance is (using ToString()) " + balance.ToString());
```

只有最后两行把 `Currency` 正确地显示为一个字符串。这是为什么？问题是在把类型强制转换和方法重载合并起来时，会出现另一个不希望的错误源。下面用倒序的方式解释这段代码。

第 3 行的 `Console.WriteLine()` 语句显式地调用 `Currency.ToString()` 方法，以确保 `Currency` 显示为一个字符串。第 2 行代码没有这么做。字符串 "balance is" 传递给 `Console.WriteLine()`，告诉编译器这个参数应解释为字符串。因此要隐式地调用 `Currency.ToString()` 方法。

但第 1 行的 `Console.WriteLine()` 方法只把原始 `Currency` 结构传递给 `Console.WriteLine()`。目前 `Console.WriteLine()` 有许多重载版本，但它们的参数都不是 `Currency` 结构。所以编译器会到处搜索，

看看它能把 `Currency` 强制转换为什么，为了与 `Console.WriteLine()` 的一个重载方法匹配。如上所示，`Console.WriteLine()` 的一个重载方法可以快速而高效地显示 `uint`，且其参数是一个 `uint`。因此应把 `Currency` 隐式地强制转换为 `uint`。

实际上，`Console.WriteLine()` 有另一个重载方法，它的参数是一个 `double` 数，结果显示该 `double` 数的值。如果仔细看看第一个 `SimpleCurrency` 示例的结果，就会发现该结果的第 1 行就是使用这个重载方法把 `Currency` 显示为一个 `double` 数。在这个示例中，没有直接把 `Currency` 强制转换为 `uint`，所以编译器选择 `Currency` → `float` → `double` 作为可用于 `Console.WriteLine()` 重载方法首选的强制转换方式。但在 `SimpleCurrency2` 中可以直接强制转换为 `uint`，所以编译器会选择该路径。

结论是：如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些强制转换方式进行数据转换，从而决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果存在任何疑问，最好指定显式地使用哪种强制转换。

7.6 小结

本章介绍了 C# 提供的标准运算符，描述了对象的相等机制，讨论了编译器如何把一种标准数据类型转换为另一种标准数据类型。还阐述了如何使用运算符重载在自己的数据类型上实现自定义运算符。最后，学习了运算符重载的一种特殊类型，即类型强制转换运算符，它允许用户指定如何将自定义类型的实例转换为其他数据类型。

第 8 章

委托、Lambda 表达式和事件

本章内容:

- 委托
- Lambda 表达式
- 事件

委托是寻址方法的 .NET 版本。在 C++ 中, 函数指针只不过是一个指向内存位置的指针, 它不是类型安全的。我们无法判断这个指针实际指向什么, 像参数和返回类型等项就更无从知晓了。而 .NET 委托完全不同, 委托是类型安全的类, 它定义了返回类型和参数的类型。委托类不仅包含对方法的引用, 也可以包含对多个方法的引用。

Lambda 表达式与委托直接相关。当参数是委托类型时, 就可以使用 Lambda 表达式实现委托引用的方法。

本章学习委托和 Lambda 表达式的基础知识, 说明如何通过 Lambda 表达式实现委托调用, 并阐述 .NET 如何将委托用作实现事件的方式。

8.1 委托

当要把方法传送给其他方法时, 需要使用委托。要了解它们的含义, 可以看看下面一行代码:

```
int i = int.Parse("99");
```

我们习惯于把数据作为参数传递给方法, 如上面的例子所示。所以, 给方法传递另一个方法听起来有点奇怪。而有时某个方法执行的操作并不是针对数据进行的, 而是要对另一个方法进行操作。更麻烦的是, 在编译时我们不知道第二个方法是什么, 这个信息只能在运行时得到, 所以需要把第二个方法作为参数传递给第一个方法。这听起来很令人迷惑, 下面用几个示例来说明:

启动线程和任务——在 C# 中, 可以告诉计算机并行运行某些新的执行序列同时运行当前的任务。这种序列就称为线程, 在其中一个基类 `System.Threading.Thread` 的一个实例上使用方法 `Start()`, 就可以启动一个线程。如果要告诉计算机启动一个新的执行序列, 就必须说明要在哪里启动该序列。必须为计算机提供开始启动的方法的细节, 即 `Thread` 类的构造函数必须带有一个参数, 该参数定义了线程调用的方法。

通用库类——许多库包含执行各种标准任务的代码。这些库通常可以自我包含。这样在编写库时, 就会知道任务该如何执行。但是有时在任务中还包含子任务, 只有使用该库的客户端代码才

知道如何执行这些子任务。例如，编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能对任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户端代码也必须告诉类如何比较要排序的特定对象。换言之，客户端代码必须给类传递某个可以调用并且进行这种比较的合适方法的细节。

- 事件——一般是通知代码发生了什么事情。GUI 编程主要处理事件。在引发事件时，运行库需要知道应执行哪个方法。这就需要把处理事件的方法作为一个参数传递给委托的。这些将在本章后面讨论。

在 C 和 C++ 中，只能提取函数的地址，并作为一个参数传递它。C 没有类型安全性。可以把任何函数传递给需要函数指针的方法。但是，这种直接方法不仅会导致一些关于类型安全性的问题，而且没有意识到：在进行面向对象编程时，几乎没有方法是孤立存在的，而是在调用方法前通常需要与类实例相关联。所以 .NET Framework 在语法上不允许使用这种直接方法。如果要传递方法，就必须把方法的细节封装在一种新类型的对象中，即委托。委托只是一种特殊类型的对象，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是一个或多个方法的地址。

8.1.1 声明委托

在 C# 中使用一个类时，分两个阶段。首先，需要定义这个类，即告诉编译器这个类由什么字段和方法组成。然后（除非只使用静态方法），实例化类的一个对象。使用委托时，也需要经过这两个步骤。首先必须定义要使用的委托，对于委托，定义它就是告诉编译器这种类型的委托表示哪种类型的方法。然后，必须创建该委托的一个或多个实例。编译器在后台将创建表示该委托的一个类。

定义委托的语法如下：

```
delegate void IntMethodInvoker(int x);
```

在这个示例中，定义了一个委托 `IntMethodInvoker`，并指定该委托的每个实例都可以包含一个方法的引用，该方法带有一个 `int` 参数，并返回 `void`。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所表示的方法的签名和返回类型等全部细节。



理解委托的一种好方式是把委托当作这样一件事情，它给方法的签名和返回类型指定名称。

假定要定义一个委托 `TwoLongsOp`，该委托表示的方法有两个 `long` 型参数，返回类型为 `double`。可以编写如下代码：

```
delegate double TwoLongsOp(long first, long second);
```

或者要定义一个委托，它表示的方法不带参数，返回一个 `string` 型的值，可以编写如下代码：

```
delegate string GetAString();
```

其语法类似于方法的定义，但没有方法体，定义的前面要加上关键字 `delegate`。因为定义委托基本上是定义一个新类，所以可以在定义类的任何相同地方定义委托，也就是说，可以在另一个类的内

部定义，也可以在任何类的外部定义，还可以在名称空间中把委托定义为顶层对象。根据定义的可见性，和委托的作用域，可以在委托的定义上应用任意常见的访问修饰符：public、private、protected 等：

```
public delegate string GetAString();
```



实际上，“定义一个委托”是指“定义一个新类”。委托实现为派生自基类 `System.MulticastDelegate` 的类，`System.MulticastDelegate` 又派生自基类 `System.Delegate`。C#编译器能识别这个类，会使用其委托语法，因此我们不需要了解这个类的具体执行情况。这是 C#与基类共同合作，使编程更易完成的另一个范例。

定义好委托后，就可以创建它的一个实例，从而用它存储特定方法的细节。



但是，在术语方面有一个问题。类有两个不同的术语：“类”表示较广义的定义，“对象”表示类的实例。但委托只有一个术语。在创建委托的实例时，所创建的委托的实例仍称为委托。必须从上下文中确定委托的确切含义。

8.1.2 使用委托

下面的代码段说明了如何使用委托。这是在 `int` 上调用 `ToString()` 方法的一种相当冗长的方式：



可从
wrox.com
下载源代码

```
private delegate string GetAString();

static void Main()
{
    int x = 40;
    GetAString firstStringMethod = new GetAString(x.ToString);
    Console.WriteLine("String is {0}", firstStringMethod());
    // With firstStringMethod initialized to x.ToString(),
    // the above statement is equivalent to saying
    // Console.WriteLine("String is {0}", x.ToString());
}
```

代码段 `GetAStringDemo/Program.cs`

在这段代码中，实例化了类型为 `GetAString` 的一个委托，并对它进行初始化，使它引用整型变量 `x` 的 `ToString()` 方法。在 C# 中，委托在语法上总是接受一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数并返回一个字符串的方法来初始化 `firstStringMethod` 变量，就会产生一个编译错误。注意，因为 `int.ToString()` 是一个实例方法(不是静态方法)，所以需要指定实例(`x`)和方法名来正确地初始化委托。

下一行代码使用这个委托来显示字符串。在任何代码中，都应提供委托实例的名称，后面的圆括号中应包含调用该委托中的方法时使用的任何等效参数。所以在上面的代码中，`Console.WriteLine()` 语句完全等价于注释语句中的代码行。

实际上，给委托实例提供圆括号与调用委托类的 `Invoke()` 方法完全相同。因为 `firstStringMethod` 是委

托类型的一个变量，所以 C# 编译器会用 `firstStringMethod.Invoke()` 代替 `firstStringMethod()`。

```
firstStringMethod();
firstStringMethod.Invoke();
```

为了减少输入量，只要需要委托实例，就可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个 C# 特性就是有效的。下面的示例用 `GetString` 委托的一个新实例初始化 `GetString` 类型的 `firstStringMethod` 变量：

```
GetString firstStringMethod = new GetString(x.ToString);
```

只要用变量 `x` 把方法名传送给变量 `firstStringMethod`，就可以编写出作用相同的代码：

```
GetString firstStringMethod = x.ToString;
```

C# 编译器创建的代码是一样的。由于编译器会用 `firstStringMethod` 检测需要的委托类型，因此它创建 `GetString` 委托类型的一个实例，用对象 `x` 把方法的地址传送给构造函数。



调用上述方法名时输入形式不能为 `x.ToString()` (不要输入圆括号)，也不能把它传给委托变量。输入圆括号调用一个方法。调用 `x.ToString()` 方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。

委托推断可以在需要委托实例的任何地方使用。委托推断也可以用于事件，因为事件基于委托 (参见本章后面的内容)。

委托的一个特征是它们的类型是安全的，可以确保被调用的方法的签名是正确的。但有趣的是，它们不关心在什么类型的对象上调用该方法，甚至不考虑该方法是静态方法，还是实例方法。



给定委托的实例可以引用任何类型的任何对象上的实例方法或静态方法——只要方法的签名匹配于委托的签名即可。

为了说明这一点，扩展上面的代码，让它使用 `firstStringMethod` 委托在另一个对象上调用其他两个方法，其中一个实例方法，另一个是静态方法。为此，使用本章前面定义的 `Currency` 结构。`Currency` 结构有自己的 `ToString()` 重载方法和一个与 `GetCurrencyUnit()` 的签名相同的静态方法。这样，就可以用同一个委托变量调用这些方法了：



可从
wrox.com
下载源代码

```
struct Currency
{
    public uint Dollars;
    public ushort Cents;

    public Currency(uint dollars, ushort cents)
    {
        this.Dollars = dollars;
        this.Cents = cents;
    }
}
```

```

public override string ToString()
{
    return string.Format("${0}.{1,2:00}", Dollars,Cents);
}

public static string GetCurrencyUnit()
{
    return "Dollar";
}

public static explicit operator Currency (float value)
{
    checked
    {
        uint dollars = (uint)value;
        ushort cents = (ushort)((value-dollars)*100);
        return new Currency(dollars, cents);
    }
}

public static implicit operator float (Currency value)
{
    return value.Dollars + (value.Cents/100.0f);
}

public static implicit operator Currency (uint value)
{
    return new Currency(value, 0);
}

public static implicit operator uint (Currency value)
{
    return value.Dollars;
}
}

```

代码段 GetAStringDemo/Currency.cs

下面就可以使用 **GetAString** 实例，代码如下所示：

```

private delegate string GetAString();

static void Main()
{
    int x = 40;
    GetAString firstStringMethod = x.ToString;
    Console.WriteLine("String is {0}", firstStringMethod());

    Currency balance = new Currency(34, 50);

    // firstStringMethod references an instance method
    firstStringMethod = balance.ToString;
    Console.WriteLine("String is {0}", firstStringMethod());

    // firstStringMethod references a static method
    firstStringMethod = new GetAString(Currency.GetCurrencyUnit);
    Console.WriteLine("String is {0}", firstStringMethod());
}

```

```
}
```

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上引用的不同方法，甚至可以指定静态方法，或者在类的不同类型的实例上引用的方法，只要每个方法的签名匹配委托定义即可。

运行应用程序，会得到委托引用的不同方法的输出结果：

```
String is 40
String is $34.50
String is Dollar
```

但是，我们实际上还没有说明把一个委托传递给另一个方法的具体过程，也没有得到任何特别有用的结果。调用 `int` 和 `Currency` 对象的 `ToString()` 的方法要比使用委托直观得多！但是，在真正领会到委托的用处前，需要用一个相当复杂的示例来说明委托的本质。下面就是两个委托的示例。第一个示例仅使用委托来调用两个不同的操作。它说明了如何把委托传递给方法，如何使用委托数组，但这仍没有很好地说明：没有委托，就不能完成很多工作。第二个示例就复杂得了，它有一个类 `BubbleSorter`，该类实现一个方法，按照升序排列一个对象数组。没有委托是很难编写出这个类。

8.1.3 简单的委托示例

在这个示例中，定义一个类 `MathsOperations`，它有两个静态方法，对 `double` 类型的值执行两个操作，然后使用该委托调用这些方法。这个数学类如下所示：



可从
wrox.com
下载源代码

```
class MathOperations
{
    public static double MultiplyByTwo(double value)
    {
        return value * 2;
    }

    public static double Square(double value)
    {
        return value * value;
    }
}
```

代码段 SimpleDelegate/MathOperations.cs

下面调用这些方法：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    delegate double DoubleOp(double x);

    class Program
    {
        static void Main()
        {
            DoubleOp[] operations =
            {
                MathOperations.MultiplyByTwo,
```

```

        MathOperations.Square
    };

    for (int i=0; i < operations.Length; i++)
    {
        Console.WriteLine("Using operations[{0}]:", i);
        ProcessAndDisplayNumber(operations[i], 2.0);
        ProcessAndDisplayNumber(operations[i], 7.94);
        ProcessAndDisplayNumber(operations[i], 1.414);
        Console.WriteLine();
    }
}

static void ProcessAndDisplayNumber(DoubleOp action, double value)
{
    double result = action(value);
    Console.WriteLine(
        "Value is {0}, result of operation is {1}", value, result);
}
}

```

代码段 SimpleDelegate/Program.cs

在这段代码中，实例化了一个委托数组 `DoubleOp` (记住，一旦定义了委托类，基本上就可以实例化它的实例，就像处理一般的类那样——所以把一些委托的实例放在数组中是可以的)。该数组的每个元素都初始化为由 `MathsOperations` 类实现的不同操作。然后遍历这个数组，把每个操作应用到 3 个不同的值上。这说明了使用委托的一种方式——把方法组合到一个数组中来使用，这样就可以在循环中调用不同的方法了。

这段代码的关键一行是把每个委托传递给 `ProcessAndDisplayNumber()` 方法，例如：

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

其中传递了委托名，但不带任何参数。假定 `operations[i]` 是一个委托，其语法是：

- `operations[i]` 表示“这个委托”。换言之，就是委托表示的方法。
- `operations[i](2.0)` 表示“实际上调用这个方法，参数放在圆括号中”。

`ProcessAndDisplayNumber()` 方法定义为把一个委托作为其第一个参数：

```
static void ProcessAndDisplayNumber(DoubleOp action, double value)
```

然后，在这个方法中，调用：

```
double result = action(value);
```

这实际上是调用 `action` 委托实例封装的方法，其返回结果存储在 `result` 中。运行这个示例，得到如下所示的结果：

```

SimpleDelegate
Using operations[0]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 15.88
Value is 1.414, result of operation is 2.828

```

```

Using operations[1]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 63.0436
Value is 1.414, result of operation is 1.999396

```

8.1.4 Action<T>和 Func<T>委托

除了为每个参数和返回类型定义一个新委托类型之外,还可以使用 **Action<T>**和 **Func<T>**委托。泛型 **Action<T>**委托表示引用一个 **void** 返回类型的方法。因为这个委托类存在不同的变体,所以可以传递至多 16 种不同的参数类型。没有泛型参数的 **Action** 类可调用没有参数的方法。**Action<in T>**调用带一个参数的方法,**Action<in T1, in T2>**调用带两个参数的方法,**Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8>**调用带 8 个参数的方法。

Func<T>委托可以以类似的方式使用。**Func<T>**允许调用带返回类型的方法。与 **Action<T>**类似,**Func<T>**也定义了不同的变体,至多也可以传递 16 个参数类型和一个返回类型。**Func<out TResult>**委托类型可以调用带返回类型且无参数的方法,**Func<in T, out TResult>**调用带一个参数的方法,**Func<in T1, in T2, in T3, in T4,out TResult>**调用带 4 个参数的方法。

上一节的示例声明了一个委托,其参数是 **double** 类型,返回类型是 **double**:

```
delegate double DoubleOp(double x);
```

除了声明自定义委托 **DoubleOp** 之外,还可以使用 **Func<in T, out TResult>**委托。可以声明一个该委托类型的变量,或者该委托类型的数组,如下所示:

```

Func <double, double>[] operations =
{
    MathOperations.MultiplyByTwo,
    MathOperations.Square
};

```

使用它,并将 **ProcessAndDisplayNumber()**方法作为参数:

```

static void ProcessAndDisplayNumber(Func<double, double>action,
    double value)
{
    double result = action(value);
    Console.WriteLine(
        "Value is {0}, result of operation is {1}", value, result);
}

```

8.1.5 BubbleSorter 示例

下面的示例将说明委托的真正用途。我们要编写一个类 **BubbleSorter**,它实现一个静态方法 **Sort()**,这个方法的一个参数是一个对象数组,把该数组按照升序重新排列。例如,假定传递给它的是 **int** 数组: {0, 5, 6, 2, 1}, 则返回的结果应是 {0, 1, 2, 5, 6}。

冒泡排序算法非常著名,是一种简单的排序方法。它适合于一小组数字,因为对于大量的数字(超过 10 个),还有更高效的算法。冒泡排序算法重复遍历数组,比较每一对数字,按照需要交换它们的位置,从而把最大的数字逐步移动到数组的最后。对于给 **int** 排序,进行冒泡排序的方法如下所示:

```

bool swapped = true;
do

```

```

    {
        swapped = false;
        for (int i = 0; i < sortArray.Length-1; i++)
        {
            if (sortArray[i] < sortArray[i+1])) // problem with this test
            {
                int temp = sortArray[i];
                sortArray[i] = sortArray[i + 1];
                sortArray[i + 1] = temp;
                swapped = true;
            }
        }
    } while (swapped);
}

```

它非常适合于 `int`，但我们希望 `Sort()` 方法能给任何对象排序。换言之，如果某段客户端代码包含 `Currency` 结构数组或自定义的其他类和结构，就需要对该数组排序。这样，上面代码中的 `if(sortArray[j] < sortArray[i])` 就有问题了，因为它需要比较数组中的两个对象，看看哪一个更大。可以对 `int` 进行这样的比较，但如何对没有实现“<”运算符的新类进行比较？答案是能识别该类的客户端代码必须在委托中传递一个封装的方法，这个方法可以进行比较。另外，不给 `temp` 变量使用 `int` 类型，而使用泛型类型就可以实现泛型方法 `Sort()`。

对于接受类型 `T` 的泛型方法 `Sort<T>()`，需要一个比较方法，其两个参数的类型是 `T`，`if` 比较的返回类型是布尔类型。这个方法可以从 `Func<T1, T2, TResult>` 委托中引用，其中 `T1` 和 `T2` 的类型相同：`Func<T, T, bool>`。

给 `Sort<T>` 方法指定下述签名：

```
static public void Sort<T> (IList<T> sortArray, Func<T, T, bool> comparison)
```

这个方法的文档说明，`comparison` 必须引用一个方法，该方法带有两个参数，如果第一个参数的值“小于”第二个参数，就返回 `true`。

设置完毕后，下面定义 `BubbleSorter` 类：



可从
wrox.com
下载源代码

```

class BubbleSorter
{
    static public void Sort<T> (IList<T> sortArray, Func<T, T, bool> comparison)
    {
        bool swapped = true;
        do
        {
            swapped = false;
            for (int i = 0; i < sortArray.Count-1; i++)
            {
                if (comparison(sortArray[i+1], sortArray[i]))
                {
                    T temp = sortArray[i];
                    sortArray[i] = sortArray[i + 1];
                    sortArray[i + 1] = temp;
                    swapped = true;
                }
            }
        } while (swapped);
    }
}

```

为了使用这个类, 需要定义另一个类, 从而建立要排序的数组。在本例中, 假定 Mortimer Phones 移动电话公司有一个员工列表, 要根据他们的薪水进行排序。每个员工分别由类 `Employee` 的一个实例表示, 如下所示:



可从
wrox.com
下载源代码

```
class Employee
{
    public Employee(string name, decimal salary)
    {
        this.Name = name;
        this.Salary = salary;
    }

    public string Name { get; private set; }
    public decimal Salary { get; private set; }

    public override string ToString()
    {
        return string.Format("{0}, {1:C}", Name, Salary);
    }

    public static bool CompareSalary(Employee e1, Employee e2)
    {
        return e1.Salary < e2.Salary;
    }
}
```

注意, 为了匹配 `Func<T, T, bool>` 委托的签名, 在这个类中必须定义 `CompareSalary`, 它的参数是两个 `Employee` 引用, 并返回一个布尔值。在实现比较的代码中, 根据薪水进行比较。

下面编写一些客户端代码, 完成排序:



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        static void Main()
        {
            Employee[] employees =
            {
                new Employee("Bugs Bunny", 20000),
                new Employee("Elmer Fudd", 10000),
                new Employee("Daffy Duck", 25000),
                new Employee("Wile Coyote", 1000000.38m),
                new Employee("Foghorn Leghorn", 23000),
                new Employee("RoadRunner", 50000)
            };
            BubbleSorter.Sort(employees, Employee.CompareSalary);

            foreach (var employee in employees)
```

```
Console.WriteLine(employee);
```

代码段 BubbleSorter/Program.cs

运行这段代码，正确显示按照薪水排列的 Employee，如下所示：

```
BubbleSorter
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
RoadRunner, $50,000.00
Wile Coyote, $1,000,000.38
```

8.1.6 多播委托

前面使用的每个委托都只包含一个方法调用。调用委托的次数与调用方法的次数相同。如果要调用多个方法，就需要多次显式调用这个委托。但是，委托也可以包含多个方法。这种委托称为多播委托。如果调用多播委托，就可以按顺序连续调用多个方法。为此，委托的签名就必须返回 void；否则，就只能得到委托调用的最后一个方法的结果。

可以使用返回类型为 void 的 Action<double> 委托：



可从
wrox.com
下载源代码

```
class Program
{
    static void Main()
    {
        Action<double>operations = MathOperations.MultiplyByTwo;
        operations += MathOperations.Square;
```

代码段 MulticastDelegates/Program.cs

在前面的示例中，因为要存储对两个方法的引用，所以实例化了一个委托数组。而这里只是在同一个多播委托中添加两个操作。多播委托可以识别运算符“+”和“+=”。另外，还可以扩展上述代码中的最后两行，如下所示：

```
Action<double>operation1 = MathOperations.MultiplyByTwo;
Action<double>operation2 = MathOperations.Square;
Action<double>operations = operation1 + operation2;
```

多播委托还识别运算符“-”和“-=”，以从委托中删除方法调用。



根据后面的内容，多播委托实际上是一个派生自 System.MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 System.Delegate。System.MulticastDelegate 的其他成员允许把多个方法调用链接为一个列表。

为了说明多播委托的用法，下面把 SimpleDelegate 示例转换为一个新示例 MulticastDelegate。现在需

要委托引用返回 `void` 的方法，就应重写 `MathOperations` 类中的方法，从而让它们显示其结果，而不是返回它们：



可从
wrox.com
下载源代码

```
class MathOperations
{
    public static void MultiplyByTwo(double value)
    {
        double result = value * 2;
        Console.WriteLine("Multiplying by 2: {0} gives {1}", value, result);
    }

    public static void Square(double value)
    {
        double result = value * value;
        Console.WriteLine("Squaring: {0} gives {1}", value, result);
    }
}
```

代码段 MulticastDelegates/MathOperations.cs

为了适应这个改变，也必须重写 `ProcessAndDisplayNumber()` 方法：

```
static void ProcessAndDisplayNumber(Action<double>action, double value)
{
    Console.WriteLine();
    Console.WriteLine("ProcessAndDisplayNumber called with value = {0}", value);
    action(value);
}
```

下面测试多播委托，其代码如下：

```
static void Main()
{
    Action<double>operations = MathOperations.MultiplyByTwo;
    operations += MathOperations.Square;

    ProcessAndDisplayNumber(operations, 2.0);
    ProcessAndDisplayNumber(operations, 7.94);
    ProcessAndDisplayNumber(operations, 1.414);
    Console.WriteLine();
}
```

现在，每次调用 `ProcessAndDisplayNumber()` 方法时，都会显示一条消息，说明它已经被调用。然后，下面的语句会按顺序调用 `action` 委托实例中的每个方法：

```
action(value);
```

运行这段代码，得到如下所示的结果：

```
MulticastDelegate

ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

ProcessAndDisplayNumber called with value = 7.94
```

```

Multiplying by 2: 7.94 gives 15.88
Squaring: 7.94 gives 63.0436

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.999396

```

如果正在使用多播委托，就应知道对同一个委托调用方法链的顺序并未正式定义。因此应避免编写依赖于以特定顺序调用方法的代码。

通过一个委托调用多个方法还可能导致一个大问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的其中一个方法抛出一个异常，整个迭代就会停止。下面是 `MulticastIteration` 示例。其中定义了一个简单的委托 `Action`，它没有参数并返回 `void`。这个委托打算调用 `One()` 和 `Two()` 方法，这两个方法满足委托的参数和返回类型要求。注意 `One()` 方法抛出了一个异常：



可从
wrox.com
下载源代码

```

using System;

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        static void One()
        {
            Console.WriteLine("One");
            throw new Exception("Error in one");
        }

        static void Two()
        {
            Console.WriteLine("Two");
        }
    }
}

```

代码段 `MulticastDelegateWithIteration/Program.cs`

在 `Main()` 方法中，创建了委托 `d1`，它引用方法 `One()`，接着把 `Two()` 方法的地址添加到同一个委托中。调用 `d1` 委托，就可以调用这两个方法。在 `try/catch` 块中捕获异常：

```

static void Main()
{
    Action d1 = One;
    d1 += Two;

    try
    {
        d1();
    }
    catch (Exception)
    {
        Console.WriteLine("Exception caught");
    }
}

```

委托只调用了第一个方法。因为第一个方法抛出了一个异常，所以委托的迭代会停止，不再调

用 `Two()` 方法。没有指定当调用方法的顺序时，结果会有所不同。

```
One
Exception Caught
```



错误和异常详见第 15 章。

在这种情况下，为了避免这个问题，应自己迭代方法列表。`Delegate` 类定义 `GetInvocationList()` 方法，它返回一个 `Delegate` 对象数组。现在可以使用这个委托调用与委托直接相关的方法，捕获异常，并继续下一次迭代。

```
static void Main()
{
    Action d1 = One;
    d1 += Two;

    Delegate[] delegates = d1.GetInvocationList();
    foreach (Action d in delegates)
    {
        try
        {
            d();
        }
        catch (Exception)
        {
            Console.WriteLine("Exception caught");
        }
    }
}
```

修改了代码后运行应用程序，会看到在捕获了异常后，将继续迭代下一个方法。

```
One
Exception caught
Two
```

8.1.7 匿名方法

到目前为止，要想使委托工作，方法必须已经存在(即委托是用它将调用的方法的相同签名定义的)。但还有另外一种使用委托的方式：即通过匿名方法。匿名方法是用作委托的参数的一段代码。

用匿名方法定义委托的语法与前面的定义并没有区别。但在实例化委托时，就有区别了。下面是一个非常简单的控制台应用程序，它说明了如何使用匿名方法：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        static void Main()
        {
```

```

string mid = ", middle part,";

Func<string, string>anonDel = delegate(string param)
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};
Console.WriteLine(anonDel("Start of string"));

```

代码段 AnonymousMethods/Program.cs

`Func<string, string>`委托接受一个字符串参数，返回一个字符串。`anonDel` 是这种委托类型的变量。不是把方法名赋予这个变量，而是使用一段简单的代码：它前面是关键字 `delegate`，后面是一个字符串参数：

可以看出，该代码块使用方法级的字符串变量 `mid`，该变量是在匿名方法的外部定义的，并把它添加到要传递的参数中。接着代码返回该字符串值。在调用委托时，把一个字符串作为参数传递，将返回的字符串输出到控制台上。

匿名方法的优点是减少了要编写的代码。不必定义仅由委托使用的方法。在为事件定义委托时，这是非常显然的(本章后面探讨事件)。这有助于降低代码的复杂性，尤其是定义了好几个事件时，代码会显得比较简单。使用匿名方法时，代码执行得不太快。编译器仍定义了一个方法，该方法只有一个自动指定的名称，我们不需要知道这个名称。

在使用匿名方法时，必须遵循两条规则。在匿名方法中不能使用跳转语句(`break`、`goto` 或 `continue`)跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

在匿名方法内部不能访问不安全的代码。另外，也不能访问在匿名方法外部使用的 `ref` 和 `out` 参数。但可以使用在匿名方法外部定义的其他变量。

如果需要用匿名方法多次编写同一个功能，就不要使用匿名方法。在本示例中，除了复制代码，编写一个指定的方法比较好，因为该方法只需编写一次，以后可通过名称引用它。

从 C# 3.0 开始，可以使用 **Lambda 表达式** 替代匿名方法。

8.2 Lambda 表达式

自 C# 3.0 开始，就可以使用一种新语法把实现代码赋予委托：**Lambda 表达式**。只要有委托参数类型的地方，就可以使用 **Lambda 表达式**。前面使用匿名方法的例子可以改为使用 **Lambda 表达式**：



Lambda 表达式的语法比匿名方法简单。如果所调用的方法有参数，且不需要参数，匿名方法的语法就比较简单，因为这样不需要提供参数。



可从
wrox.com
下载源代码

```

using System;

namespace Wrox.ProCSharp.Delegates
{

```

```

class Program
{
    static void Main()
    {
        string mid = ", middle part,";

        Func<string, string>lambda = param =>
        {
            param += mid;
            param += " and this was added to the string.";
            return param;
        };

        Console.WriteLine(lambda("Start of string"));
    }
}

```

代码段 LambdaExpressions/Program.cs

Lambda 运算符 “=>” 的左边列出了需要的参数。Lambda 运算符的右边定义了赋予 lambda 变量的方法的实现代码。

8.2.1 参数

Lambda 表达式有几种定义参数的方式。如果只有一个参数，只写出参数名就足够了。下面的 Lambda 表达式使用了参数 s。因为委托类型定义了一个 string 参数，所以 s 的类型就是 string。实现代码调用 String.Format() 方法来返回一个字符串，在调用该委托时，就把字符串写到控制台上：

```

Func<string, string>oneParam = s => String.Format(
    "change uppercase {0}", s.ToUpper());
Console.WriteLine(oneParam("test"));

```

如果委托使用多个参数，就把参数名放在花括号中。这里参数 x 和 y 的类型是 double，由 Func<double, double, double> 委托定义：

```

Func<double, double, double> twoParams = (x, y) => x * y;
Console.WriteLine(twoParams(3, 2));

```

为了方便，可以在花括号中给变量名添加参数类型：

```

Func<double, double, double> twoParamsWithTypes =
    (double x, double y) => x * y;
Console.WriteLine(twoParamsWithTypes(4, 2));

```

8.2.2 多行代码

如果 Lambda 表达式只有一条语句，在方法块内就不需要花括号和 return 语句，因为编译器会添加一条隐式的 return 语句。

```

Func<double, double> square = x => x * x;

```

添加花括号、return 语句和分号是完全合法的，通常这比不添加这些符号更容易阅读：

```

Func<double, double> square = x =>
{
    return x * x;
}

```

```

    {
        return x * x;
    }

```

但是，如果在 Lambda 表达式的实现代码中需要多条语句，就必须添加花括号和 return 语句：

```

Func<string, string> lambda = param =>
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};

```

8.2.3 Lambda 表达式外部的变量

通过 Lambda 表达式可以访问 Lambda 表达式块外部的变量。这是一个非常好的功能，但如果未正确使用，也会非常危险。

在下面的示例中，Func<int, int> 类型的 Lambda 表达式需要一个 int 参数，返回一个 int。该 Lambda 表达式的参数用变量 x 定义。实现代码还访问了 Lambda 表达式外部的变量 someVal。只要不认为在调用 f 时，Lambda 表达式创建了一个以后使用的新方法，这似乎没有什么问题。看看这个代码块，调用 f 的返回值应是 x 加 5 的结果，但似乎不是这样：

```

int someVal = 5;
Func<int, int> f = x => x + someVal;

```

假定以后要修改变量 someVal，于是调用 Lambda 表达式时，会使用 someVal 的新值。调用 f(3) 的结果是 10：

```

someVal = 7;
Console.WriteLine(f(3));

```

特别是，通过另一个线程调用 Lambda 表达式时，我们可能不知道进行了这个调用，也不知道外部变量的当前值是什么。

现在我们也可能会奇怪，如何在 Lambda 表达式的内部访问 Lambda 表达式外部的变量。为了理解这一点，看看编译器在定义 Lambda 表达式时做了什么。对于 Lambda 表达式 $x \Rightarrow x + \text{someVal}$ ，编译器会创建一个匿名类，它有一个构造函数来传递外部变量。该构造函数取决于从外部传递进来的变量个数。对于这个简单的例子，构造函数接受一个 int。匿名类包含一个匿名方法，其实现代码、参数和返回类型由 Lambda 表达式定义：

```

public class AnonymousClass
{
    private int someVal;
    public AnonymousClass(int someVal)
    {
        this.someVal = someVal;
    }
    public int AnonymousMethod(int x)
    {
        return x + someVal;
    }
}

```

使用 Lambda 表达式并调用该方法，会创建匿名类的一个实例，并传递调用该方法时变量的值。



Lambda 表达式可以用于类型是一个委托的任意地方。类型是 `Expression` 或 `Expression<T>` 时，也可以使用 Lambda 表达式。此时编译器会创建一个表达式树，详见第 11 章。

8.3 事件

事件基于委托，为委托提供了一种发布/订阅机制。在架构内到处都能看到事件。在 Windows 应用程序中，`Button` 类提供了 `Click` 事件。这类事件就是委托。触发 `Click` 事件时调用的处理程序方法需要定义，其参数由委托类型定义。

在本节的示例代码中，事件用于连接 `CarDealer` 类和 `Consumer` 类。`CarDealer` 类提供了一个新车到达时触发的事件。`Consumer` 类订阅该事件，以获得新车到达的通知。

8.3.1 事件发布程序

从 `CarDealer` 类开始，它基于事件提供一个订阅。`CarDealer` 类用 `event` 关键字定义了类型为 `EventHandler<CarInfoEventArgs>` 的 `NewCarInfo` 事件。在 `NewCar()` 方法中，触发 `NewCarInfo` 事件：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Delegates
{
    public class CarInfoEventArgs : EventArgs
    {
        public CarInfoEventArgs(string car)
        {
            this.Car = car;
        }

        public string Car { get; private set; }
    }

    public class CarDealer
    {
        public event EventHandler<CarInfoEventArgs> NewCarInfo;

        public void NewCar(string car)
        {
            Console.WriteLine("CarDealer, new car {0}", car);
            if (NewCarInfo != null)
            {
                NewCarInfo(this, new CarInfoEventArgs(car));
            }
        }
    }
}
```

代码段 EventsSample/CarDealer.cs

`CarDealer` 类提供了 `EventHandler<CarInfoEventArgs>` 类型的 `NewCarInfo` 事件。作为一个约定，事件一般使用带两个参数的方法，其中第一个参数是一个对象，包含事件的发送者，第二个参数提供了事件的相关信息。第二个参数随不同的事件类型而不同。`.NET 1.0` 为所有不同数据类型的事件定义了几百个委托。有了泛型委托 `EventHandler<T>` 后，这就不再需要委托了。`EventHandler<TEventArgs>` 定义了一个处理程序，它返回 `void`，接受两个参数。对于 `EventHandler<TEventArgs>`，第一个参数必须是 `object` 类型，第二个参数是 `T` 类型。`EventHandler<TEventArgs>` 还定义了一个关于 `T` 的约束：它必须派生自基类 `EventArgs`，`CarInfoEventArgs` 就派生自基类 `EventArgs`：

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

委托 `EventHandler<TEventArgs>` 的定义如下：

```
public delegate void EventHandler<TEventArgs> (object sender, TEventArgs e)
    where TEventArgs: EventArgs
```

在一行上定义事件是 C# 的简化记法。编译器会创建一个 `EventHandler<CarInfoEventArgs>` 委托类型的变量，并添加方法，以便从委托中订阅和取消订阅。该简化记法的较长形式如下所示。这非常类似于自动属性和完整属性之间的关系。对于事件，使用 `add` 和 `remove` 关键字添加和删除委托的处理程序：

```
private delegate EventHandler<CarInfoEventArgs> newCarInfo;
public event EventHandler<CarInfoEventArgs> NewCarInfo
{
    add
    {
        newCarInfo += value;
    }
    remove
    {
        newCarInfo -= value;
    }
}
```



如果不仅仅需要添加和删除事件处理程序，定义事件的长记法就很有用，例如，需要为多个线程访问添加同步操作。WPF 控件使用长记法给事件添加冒泡和隧道功能。事件的冒泡和隧道详见第 27 章。

`CarDealer` 类在 `NewCar` 方法中触发事件。使用 `NewCarInfo` 和花括号可以调用给事件订阅的所有处理程序。注意与多播委托一样，方法的调用顺序无法保证。为了更多地控制处理程序的调用，可以使用 `Delegate` 类的 `GetInvocationList()` 方法，访问委托列表中的每一项，并独立地调用每个方法，如上所示。

在触发事件之前，需要检查委托 `NewCarInfo` 是否不为空。如果没有订阅处理程序，委托就是空。

```
public void NewCar(string car)
{
    Console.WriteLine("CarDealer, new car {0}", car);
    if (NewCarInfo != null)
```

```

    {
        NewCarInfo(this, new CarInfoEventArgs(car));
    }
}

```

8.3.2 事件侦听器

`Consumer` 类用作事件侦听器。这个类订阅了 `CarDealer` 类的事件, 并定义了 `NewCarIsHere` 方法, 该方法满足 `EventHandler<CarInfoEventArgs>` 委托的要求, 其参数类型是 `object` 和 `CarInfoEventArgs`:



可从
wrox.com
下载源代码

```

using System;

namespace Wrox.ProCSharp.Delegates
{
    public class Consumer
    {
        private string name;

        public Consumer(string name)
        {
            this.name = name;
        }

        public void NewCarIsHere(object sender, CarInfoEventArgs e)
        {
            Console.WriteLine("{0}: car {1} is new", name, e.Car);
        }
    }
}

```

代码段 EventsSample/Consumer.cs

现在需要连接事件发布程序和订阅器。为此使用 `CarDealer` 类的 `NewCarInfo` 事件, 通过 “+” 创建一个订阅。消费者 `michael`(变量)订阅了事件, 接着消费者 `nick`(变量)也订阅了事件, 然后 `michael`(变量)通过 “-” 取消了订阅。



可从
wrox.com
下载源代码

```

namespace Wrox.ProCSharp.Delegates
{
    class Program
    {
        static void Main()
        {
            var dealer = new CarDealer();

            var michael = new Consumer("Michael");
            dealer.NewCarInfo += michael.NewCarIsHere;

            dealer.NewCar("Mercedes");

            var nick = new Consumer("Nick");
            dealer.NewCarInfo += nick.NewCarIsHere;

            dealer.NewCar("Ferrari");

            dealer.NewCarInfo -= michael.NewCarIsHere;

            dealer.NewCar("Toyota");
        }
    }
}

```

代码段 EventsSample/Program.cs

运行应用程序，一辆 Mercedes 到达，Michael 得到了通知。因为之后 Nick 也注册了该订阅，所以 Michael 和 Nick 都获得了新 Ferrari 的通知。接着 Michael 取消了订阅，所以只有 Nick 获得了 Toyota 的通知。

```
CarDealer, new car Mercedes
Michael: car Mercedes is new
CarDealer, new car Ferrari
Michael: car Ferrari is new
Nick: car Ferrari is new
CarDealer, new car Toyota
Nick: car Toyota is new
```

8.3.3 弱事件

通过事件，直接连接到发布程序和侦听器。但垃圾回收有一个问题。例如，如果侦听器不再直接引用，发布程序就仍有一个引用。垃圾回收器不能清空侦听器占用的内存，因为发布程序仍保有一个引用，会针对侦听器触发事件。

这种强连接可以通过弱事件模式来解决，即使用 `WeakEventManager` 作为发布程序和侦听器之间的中介。

前面的示例把 `CarDealer` 作为发布程序，把 `Consumer` 作为侦听器，下一节将修改这个示例，以使用弱事件模式。

1. 弱事件管理器

要使用弱事件，需要创建一个派生自 `WeakEventManager` 类的类。`WeakEventManager` 类在程序集 `WindowsBase` 的名称空间 `System.Windows` 中定义。

`WeakCarInfoEventManager` 类是弱事件管理器类，它管理 `NewCarInfo` 事件的发布程序和侦听器之间的连接。因为这个类实现了单一模式，所以只创建一个实例。静态属性 `CurrentManager` 创建了一个 `WeakCarInfoEventManager` 类型的对象(如果它不存在)，并返回对该对象的引用。`WeakCarInfoEventManager.CurrentManager` 用于访问 `WeakCarInfoEventManager` 类中的单一对象。

对于弱事件模式，弱事件管理器类需要静态方法 `AddListener()` 和 `StopListening()`。侦听器使用这些方法连接发布程序，和断开与发布程序的连接，而不是直接使用发布程序中的事件。侦听器还需要实现稍后介绍的接口 `IWeakEventListener`。通过 `AddListener()` 和 `Remove Listener()` 方法，调用 `WeakEventManager` 基类中的方法，来添加和删除侦听器。

对于 `WeakCarInfoEventManager` 类，还需要重写基类的 `StartListening()` 和 `StopListening()` 方法。添加第一个侦听器时调用 `StartListening()` 方法，删除最后一个侦听器时调用 `StopListening()` 方法。`StartListening()` 方法和 `StopListening()` 方法从弱事件管理器中订阅和取消订阅一个方法，以侦听发布程序中的事件。如果弱事件管理器类需要连接到不同的发布程序类型上，就可以在源对象中检查类型信息，之后进行类型强制转换。接着使用基类的 `DeliverEvent()` 方法，把事件传递给侦听器。`DeliverEvent()` 方法在侦听器中调用 `IWeakEventListener` 接口中的 `ReceiveWeakEvent()` 方法：



可从
wrox.com
下载源代码

```
using system. Windows
namespace Wrox.ProCSharp.Delegates
{
    public class WeakCarInfoEventManager: WeakEventManager
    {
        public static void AddListener(object source, IWeakEventListener listener)
        {
            CurrentManager.ProtectedAddListener(source, listener);
        }

        public static void RemoveListener(object source,
                                           IWeakEventListener listener)
        {
            CurrentManager.ProtectedRemoveListener(source, listener);
        }

        public static WeakCarInfoEventManager CurrentManager
        {
            get
            {
                WeakCarInfoEventManager manager =
                    GetCurrentManager(typeof(WeakCarInfoEventManager))
                    as WeakCarInfoEventManager;
                if (manager == null)
                {
                    manager = new WeakCarInfoEventManager();
                    SetCurrentManager(typeof(WeakCarInfoEventManager), manager);
                }
                return manager;
            }
        }

        protected override void StartListening(object source)
        {
            (source as CarDealer).NewCarInfo += CarDealer_NewCarInfo;
        }

        void CarDealer_NewCarInfo(object sender, CarInfoEventArgs e)
        {
            DeliverEvent(sender, e);
        }

        protected override void StopListening(object source)
        {
            (source as CarDealer).NewCarInfo -= CarDealer_NewCarInfo;
        }
    }
}
```

代码段 WeakEventsSample/WeakCarInfoEventManager.cs



WPF 使用弱事件模式和事件管理器类 `CollectionChangedEventManager`、`CurrentChangedEventManager`、`CurrentChangingEventManager`、`PropertyChangedEventManager`、`DataChangedEventManager` 和 `LostFocusEventManager`。

对于发布程序类 `CarDealer`，不需要做任何修改，其实现代码与前面相同。

2. 事件侦听器

侦听器需要改为实现 `IWeakEventListener` 接口。这个接口定义了 `ReceiveWeakEvent()` 方法，触发事件时，从弱事件管理器中调用这个方法。在该方法的实现代码中，应从触发的事件中调用 `NewCarIsHere()` 方法。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.Delegates
{
    public class Consumer : IWeakEventListener
    {
        private string name;

        public Consumer(string name)
        {
            this.name = name;
        }

        public void NewCarIsHere(object sender, CarInfoEventArgs e)
        {
            Console.WriteLine("{0}: car {1} is new", name, e.Car);
        }

        bool IWeakEventListener.ReceiveWeakEvent(Type managerType, object sender,
                                                    EventArgs e)
        {
            NewCarIsHere(sender, e as CarInfoEventArgs);
            return true;
        }
    }
}
```

代码段 WeakEventsSample/Consumer.cs

在 `Main` 方法中，连接发布程序和侦听器，该连接现在使用 `WeakCarInfoEventManager` 类的 `AddListener()` 和 `RemoveListener()` 静态方法。



可从
wrox.com
下载源代码

```
static void Main()
{
    var dealer = new CarDealer();

    var michael = new Consumer("Michael");
    WeakCarInfoEventManager.AddListener(dealer, michael);

    dealer.NewCar("Mercedes");

    var nick = new Consumer("Nick");
    WeakCarInfoEventManager.AddListener(dealer, nick);

    dealer.NewCar("Ferrari");

    WeakCarInfoEventManager.RemoveListener(dealer, michael);
}
```

```
dealer.NewCar("Toyota");
```

代码段 WeakEventsSample/Program.cs

实现了弱事件模式后,发布程序和侦听器就不再强连接了。当不再引用侦听器时,它就会被垃圾回收。

8.4 小结

本章介绍了委托、Lambda 表达式和事件的基本知识,解释了如何声明委托,如何给委托列表添加方法,如何实现通过委托和 Lambda 表达式调用的方法,并讨论了声明事件处理程序来响应事件的过程,以及如何创建自定义事件,使用引发事件的模式。

.NET 开发人员将大量使用委托和事件,特别是在开发 Windows 应用程序时。事件是 .NET 开发人员监控应用程序执行时出现的各种 Windows 消息的方式,否则就必须监控 WndProc,捕获 WM_MOUSEBUTTONDOWN 消息,而不是获取按钮的鼠标 Click 事件。

在设计大型应用程序时,使用委托和事件可以减少依赖性和层的耦合,并能开发出具有更高重用性的组件。

Lambda 表达式是委托的 C# 语言特性。通过它们可以减少需要编写的代码量。Lambda 表达式不仅仅用于委托,详见第 11 章。

下一章介绍字符串和正则表达式。

第 9 章

字符串和正则表达式

本章内容:

- 创建字符串
- 格式化表达式
- 正则表达式

从本书前面开始,我们一直在使用字符串,但可能没有意识到,在 C# 中 `string` 关键字的映射实际上指向 .NET 基类 `System.String`。`System.String` 是一个功能非常强大且用途非常广泛的基类,但它不是 .NET 库中唯一与字符串相关的类。本章首先复习一下 `System.String` 的特性,再介绍如何使用其他的 .NET 库类来处理字符串,特别是 `System.Text` 和 `System.Text.Regular Expressions` 名称空间中的类。本章主要介绍下述内容:

- **创建字符串**——如果多次修改一个字符串,例如,在显示字符串或将其传递给其他方法或应用程序前,创建一个较长的字符串,`String` 类就会变得效率低下。对于这种情况,应使用另一个类 `System.Text.StringBuilder`,因为它是专门为这种情况设计的。
- **格式化表达式**——这些格式化表达式将用于后面几章中的 `Console.WriteLine()` 方法。格式化表达式使用两个有效的接口 `IFormatProvider` 和 `IFormattable` 来处理。在自己的类上实现这两个接口,实际上就可以定义自己的格式化序列,这样, `Console.WriteLine()` 和类似的类就可以以指定的方式显示类的值。
- **正则表达式**——.NET 还提供了一些非常复杂的类来识别字符串,或从长字符串中提取满足某些复杂条件的子字符串。例如,找出字符串中所有重复出现的某个字符或一组字符,或者找出以 `s` 开头且至少包含一个 `n` 的所有单词,或者找出遵循雇员 ID 或社会安全号码结构的字符串。虽然可以使用 `String` 类,编写方法来完成这类处理,但这类方法编写起来比较繁琐。而使用 `System.Text.RegularExpressions` 名称空间中的类就比较简单, `System.Text.RegularExpressions` 专门用于完成这类处理。

9.1 System.String 类

在介绍其他字符串类之前,先快速复习一下 `String` 类中一些可用的方法。

`System.String` 是一个类,专门用于存储字符串,允许对字符串进行许多操作。由于这种数据类型非常重要,C# 提供了它自己的关键字和相关的语法,以便于使用这个类来轻松地处理字符串。

使用运算符重载可以连接字符串：

```
string message1 = "Hello"; // returns "Hello"
message1 += ", There"; // returns "Hello, There"
string message2 = message1 + "!"; // returns "Hello, There!"
```

C#还允许使用类似于索引器的语法来提取指定的字符：

```
string message = "Hello";
char char4 = message[4]; // returns 'o'. Note the string is zero-indexed
```

这个类可以完成许多常见的任务，如替换字符、删除空白和把字母变成大写形式等。可用的方法如表 9-1 所示。

表 9-1

方 法	作 用
Compare	比较字符串的内容，考虑文化背景(区域)，判断某些字符是否相等
CompareOrdinal	与 Compare 一样，但不考虑文化背景
Concat	把多个字符串实例合并为一个实例
CopyTo	把特定数量的字符从选定的下标复制到数组的一个全新实例中
Format	格式化包含各种值的字符串和如何格式化每个值的说明符
IndexOf	定位字符串中第一次出现某个给定子字符串或字符的位置
IndexOfAny	定位字符串中第一次出现某个字符或一组字符的位置
Insert	把一个字符串实例插入到另一个字符串实例的指定索引处
Join	合并字符串数组，创建一个新字符串
LastIndexOf	与 IndexOf 一样，但定位最后一次出现的位置
LastIndexOfAny	与 IndexOfAny 一样，但定位最后一次出现的位置
PadLeft	在字符串的左侧，通过添加指定的重复字符填充字符串
PadRight	在字符串的右侧，通过添加指定的重复字符填充字符串
Replace	用另一个字符或子字符串替换字符串中给定的字符或子字符串
Split	在出现给定字符的地方，把字符串拆分为一个子字符串数组
Substring	在字符串中检索给定位置的子字符串
ToLower	把字符串转换为小写形式
ToUpper	把字符串转换为大写形式
Trim	删除首尾的空白



表 9-1 并不完整，但可以让您明白字符串所提供的功能。

9.1.1 创建字符串

如上所述，string 类是一个功能非常强大的类，它实现许多很有用的方法。但是，string 类存在一个

问题：重复修改给定的字符串，效率会很低，它实际上是一个不可变的数据类型，一旦对字符串对象进行了初始化，该字符串对象就不能改变了。表面上修改字符串内容的方法和运算符实际上创建一个新字符串，根据需要，可以把旧字符串的内容复制到新字符串中。例如，下面的代码：

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

在执行这段代码时，首先，创建一个 `System.String` 类型的对象，并把它初始化为文本“Hello from all the guys at Wrox Press. ”。注意句号后面有一个空格。此时.NET 运行库会为该字符串分配足够的内存来保存这个文本(39 个字符)，再设置变量 `greetingText`，来表示这个字符串实例。

从语法上看，下一行代码是把更多的文本添加到字符串中。实际上并非如此，而是创建一个新字符串实例，给它分配足够的内存，以存储合并的文本(共 103 个字符)。把最初的文本“Hello from all the people at Wrox Press. ”复制到这个新字符串中，再加上额外的文本“We do hope you enjoy this book as much as we enjoyed writing it.”。然后更新存储在变量 `greetingText` 中的地址，使变量正确地指向新的字符串对象。现在没有引用旧的字符串对象——不再有变量引用它，下一次垃圾收集器清理应用程序中所有未使用的对象时，就会删除它。

这本身还不坏，但假定要对这个字符串编码，在字母表中，用 ASCII 码靠后的字符替代其中的每个字母(标点符号除外)，作为非常简单的加密模式的一部分。这就会把该字符串变成“Ifmmp gspn bmm uif hvst bu Xsyp Qsftt. Xf ep ipqf zpv fokpz uijt cppl bt nvdi bt xf fokpzfe xsjujoh ju.”。完成这个任务有好几种方式，但最简单、最高效的一种(假定只使用 `String` 类)是使用 `String.Replace()` 方法，该方法把字符串中指定的子字符串用另一个子字符串代替。使用 `Replace()`，对文本进行编码的代码如下所示：



可从
wrox.com
下载源代码

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";

for(int i = 'z'; i >= 'a'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingText = greetingText.Replace(old1, new1);
}

for(int i = 'Z'; i >= 'A'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingText = greetingText.Replace(old1, new1);
}

Console.WriteLine("Encoded:\n" + greetingText);
```

代码段 StringEncoder.cs



为了简单起见，这段代码没有把 Z 换成 A，也没有把 z 换成 a。这些字符分别编码为 “[” 和 “{”。

在本示例中, `Replace()` 方法以一种智能的方式工作, 在某种程度上, 它并没有创建一个新字符串, 除非它实际上要对旧字符串进行某些改变。原来的字符串包含 23 个不同的小写字母, 和 3 个不同的大写字母。所以 `Replace()` 就分配一个新字符串, 共 26 次, 每个新字符串都包含 103 个字符。因此加密过程需要在堆上有一个总共能存储 2678 个字符的字符串对象, 最终将等待被垃圾收集! 显然, 如果使用字符串频繁进行文字处理, 应用程序就会遇到严重的性能问题。

为了解决这类问题, Microsoft 提供了 `System.Text.StringBuilder` 类。`StringBuilder` 类不像 `String` 类那样能够支持非常多的方法。在 `StringBuilder` 类上可以进行的处理仅限于替换和追加或删除字符串中的文本。但是, 它的工作方式非常高效。

在使用 `String` 类构造一个字符串时, 要给它分配足够的内存来保存字符串。然而, `StringBuilder` 类通常分配的内存会比它需要的更多。开发人员可以选择指定 `StringBuilder` 要分配多少内存, 但如果没有指定, 在默认情况下就根据初始化 `StringBuilder` 实例时的字符串长度来确定内存的大小。`StringBuilder` 类有两个主要的属性:

- `Length` 指定字符串的实际长度;
- `Capacity` 指定字符串在分配的内存中的最大长度。

对字符串的修改就在赋予 `StringBuilder` 实例的内存块中进行, 这就大大提高了追加子字符串和替换单个字符的效率。删除或插入子字符串仍然效率低下, 因为这需要移动随后的字符串。只有执行扩展字符串容量的操作, 才需要给字符串分配新内存, 才可能移动包含的整个字符串。在添加额外的容量时, 从经验来看, 如果 `StringBuilder` 类检测到容量超出, 且容量没有设置新值, 就会使自己的容量翻倍。

例如, 如果使用 `StringBuilder` 对象构造最初的欢迎字符串, 就可以编写下面的代码:

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we enjoyed writing
it");
```



为了使用 `StringBuilder` 类, 需要在代码中引用 `System.Text` 类。

在这段代码中, 为 `StringBuilder` 类设置的初始容量是 150。最好把容量设置为字符串可能的最大长度, 确保 `StringBuilder` 类不需要重新分配内存, 因为其容量足够用了。该容量默认设置为 16。理论上, 可以设置尽可能大的数字, 足够给该容量传送一个 `int`, 但如果实际上给字符串分配 20 亿个字符的空间(这是 `StringBuilder` 实例理论上允许拥有的最大空间), 系统就可能会没有足够的内存。

执行上面的代码时, 它首先创建一个 `StringBuilder` 对象, 如图 9-1 所示。

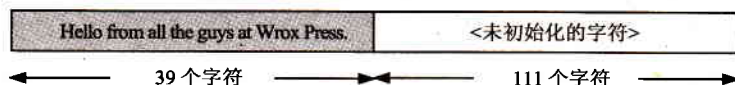


图 9-1

然后, 在调用 `AppendFormat()` 方法时, 其他文本就放在空的空间中, 不需要分配更多的内存。但是, 多次替换文本才能获得使用 `StringBuilder` 类所带来的高效性能。例如, 如果要以前面的方式

加密文本，就可以执行整个加密过程，无须分配更多的内存：

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we " + "enjoyed
    writing it");

Console.WriteLine("Not Encoded:\n" + greetingBuilder);

for(int i = 'z'; i >='a'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingBuilder = greetingBuilder.Replace(old1, new1);
}

for(int i = 'Z'; i >='A'; i--)
{
    char old1 = (char)i;
    char new1 = (char)(i+1);
    greetingBuilder = greetingBuilder.Replace(old1, new1);
}

Console.WriteLine("Encoded:\n" + greetingBuilder);
```

这段代码使用了 `StringBuilder.Replace()` 方法，它的功能与 `String.Replace()` 一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 `StringBuilder` 实例以及在最后一条 `Console.WriteLine()` 语句中执行字符串操作期间分配的内存。

一般，使用 `StringBuilder` 类可以执行字符串的任何操作，使用 `String` 类可以存储字符串或显示最终结果。

9.1.2 StringBuilder 成员

前面介绍了 `StringBuilder` 类的一个构造函数，它的参数是一个初始字符串及该字符串的容量。`StringBuilder` 类还有几个其他的构造函数，例如，可以只提供一个字符串：

```
StringBuilder sb = new StringBuilder("Hello");
```

或者用给定的容量创建一个空的 `StringBuilder` 类：

```
StringBuilder sb = new StringBuilder(20);
```

除了前面介绍的 `Length` 和 `Capacity` 属性外，还有一个只读属性 `MaxCapacity`，它表示对给定的 `StringBuilder` 实例的容量限制。在默认情况下，这由 `int.MaxValue` 给定(大约 20 亿，如前所述)。但在构造 `StringBuilder` 对象时，也可以把这个值设置为较低的值：

```
// This will both set initial capacity to 100, but the max will be 500.
// Hence, this StringBuilder can never grow to more than 500 characters,
// otherwise it will raise exception if you try to do that.
StringBuilder sb = new StringBuilder(100, 500);
```

还可以随时显式地设置容量，但如果把这个值设置为小于字符串的当前长度，或者超出了最大容量的某个值，就会抛出一个异常：

```
StringBuilder sb = new StringBuilder("Hello");
sb.Capacity = 100;
```

StringBuilder 类主要的方法如表 9-2 所示。

表 9-2

名 称	作 用
Append()	给当前字符串追加一个字符串
AppendFormat()	追加特定格式的字符串
Insert()	在当前字符串中插入一个子字符串
Remove()	从当前字符串中删除字符
Replace()	在当前字符串中, 用某个字符全部替换另一个字符, 或者用当前字符串中的一个子字符串全部替换另一个字符串
ToString()	返回当前强制转换为 System.String 对象的字符串(在 System.Object 中被重写)

其中一些方法还有几种格式的重载方法。



AppendFormat()方法实际上会在最终调用 Console.WriteLine()方法时调用, 它负责确定所有像{0:D}的格式化表达式应使用什么表达式替代。下一节讨论这个问题。

不能把 StringBuilder 强制转换为 String(隐式转换和显式转换都不行)。如果要把 StringBuilder 的内容输出为 String, 唯一的方式就是使用 ToString()方法。

前面介绍了 StringBuilder 类, 说明了使用它提高性能的一些方式。注意, 这个类并不总能提高性能。StringBuilder 类基本上应在处理多个字符串时使用。但如果只是连接两个字符串, 使用 System.String 类会比较好。

9.1.3 格式字符串

在前面的代码示例中编写了许多类和结构, 对这些类和结构实现 ToString()方法, 都是为了显示给定变量的内容。但是, 用户常常希望以各种可能的方式显示变量的内容, 在不同的文化或地区背景中有不同的格式。 .NET 基类 System.DateTime 就是最明显的一个示例: 可以把日期显示为 10 June 2010、10 Jun 2010、6/10/10(美国)、10/6/10(英国)或 10.06.2010(德国)。

同样, 在第 7 章中编写的 Vector 结构实现了 Vector.ToString()方法, 这是为了以(4, 56, 8)格式显示矢量。编写矢量另一个非常常用的方式类似 4i + 56j + 8k。如果要使类的用户友好性比较高, 就需要使用某些工具以用户希望的方式显示它们的字符串表示。 .NET 运行库定义了一种标准方式: 使用 IFormattable 接口。本节的主题就是说明如何把这个重要特性添加到类和结构上。

在显示一个变量时, 常常需要指定它的格式, 其中经常调用 Console.WriteLine()方法。因此, 本节把这个方法作为示例, 但这里的讨论适用于格式字符串的大多数情况。例如, 如果要在列表框或文本框中显示一个变量的值, 一般就使用 String.Format()方法来获得该变量的适当字符串表示, 但用于请求所需格式的格式说明符与传递给 Console.WriteLine()方法的格式相同。因此本节把 Console.WriteLine()方法作为一个示例来说明。首先看看在为基元类型提供格式字符串时会发生什么, 再看看如何把自己

的类和结构的格式说明符添加到过程中。

第2章在 `Console.WriteLine()` 和 `Console.WriteLine()` 方法中使用了格式字符串：

```
double d = 13.45;
int i = 45;
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

格式字符串本身大都由要显示的文本组成，但只要有要格式化的变量，它在参数列表中的下标就必须放在花括号中。在花括号中还可以有与该项的格式相关的其他信息。例如，可以包含：

- 该项的字符串表示要占用的字符数，这个信息的前面应有一个逗号。负值表示该项应左对齐，正值表示该项应右对齐。如果该项占用的字符数比给定的多，其内容就会完整地显示出来。
- 格式说明符也可以显示出来。它的前面应有一个冒号，表示应如何格式化该项。例如，把一个数字格式化为货币，或者以科学计数法显示。

第2章简要介绍了数字类型的常见格式说明符，表9-3再次引用该表。

表 9-3

格式说明符	应 用	含 义	示 例
C	数字类型	专用场合的货币值	\$4834.50 (USA) £4834.50 (UK)
D	只用于整数类型	一般的整数	4834
E	数字类型	科学计数法	4.834E+003
F	数字类型	小数点后的位数固定	4384.50
G	数字类型	一般的数字	4384.5
N	数字类型	通常是专用场合的数字格式	4,384.50 (UK/USA) 4 384,50 (欧洲大陆)
P	数字类型	百分比计数法	432,000.00%
X	只用于整数类型	十六进制格式	1120 (如果要显示 0x1120，就需要写上 0x)

如果要在整数上加上前导 0，就可以将格式说明符 0 重复尽可能多的次数。例如，格式说明符 0000 会把 3 显示为 0003，99 显示为 0099。

这里不能给出完整的列表，因为其他数据类型有自己的格式说明符。本节的主要目的是说明如何为自己的类定义格式说明符。

1. 字符串的格式化

为了说明如何格式化字符串，看看执行下面的语句会得到什么结果：

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

`Console.WriteLine()` 方法只是把参数的完整列表传送给静态方法 `String.Format()`。如果要在字符串中以其他方式格式化这些值，例如显示在一个文本框中，则也可以调用这个方法。实现带有 3 个

参数的 `WriteLine()` 重载方法的代码如下:

```
// Likely implementation of Console.WriteLine()

public void WriteLine(string format, object arg0, object arg1)
{
    this.WriteLine(string.Format(this.FormatProvider, format,
        new object[]{arg0, arg1}));
}
```

上面的代码依次调用了带有 1 个参数的重载方法 `WriteLine()`，仅显示了传递过来的字符串的内容，没有对它进行进一步的格式化。

`String.Format()` 方法现在需要用对应对象的合适字符串表示来替换每个格式说明符，构造最终的字符串。但是，如前所述，对于这个构建字符串的过程，需要 `StringBuilder` 实例，而不是 `String` 实例。在这个示例中，`StringBuilder` 实例是用字符串的第一部分(即文本 “The double is”)创建和初始化的。然后调用 `StringBuilder.AppendFormat()` 方法，传递第一个格式说明符 “{0,10:E}” 和相应的对象 `double`，把这个对象的字符串表示添加到构造好的字符串对象中，这个过程会继续重复调用 `StringBuilder.Append()` 和 `StringBuilder.AppendFormat()` 方法，直到得到了全部格式化好的字符串为止。

下面的内容比较有趣。`StringBuilder.AppendFormat()` 方法需要指出如何格式化对象，它首先检查对象，确定它是否实现 `System` 名称空间中的接口 `IFormattable`。只要试着把这个对象强制转换为接口，看看强制转换是否成功即可，或者使用 C# 的关键字 `is` 实现此测试。如果测试失败，`AppendFormat()` 方法就会调用对象的 `ToString()` 方法，所有的对象都从 `System.Object` 类继承了这个方法或重写了该方法。在前面给出的编写各种类和结构的示例中，执行过程都是这样，因为目前编写的类都没有实现这个接口。这就是在前面的章节中，`Object.ToString()` 的重写方法允许在 `Console.WriteLine()` 语句中显示结构和类(如 `Vector`)的原因。

但是，所有预定义的基元数字类型都实现这个接口，对于这些类型，特别是这个示例中的 `double` 和 `int`，就不会调用继承自 `System.Object` 类的基本 `ToString()` 方法。为了理解这个过程，需要了解 `IFormattable` 接口。

`IFormattable` 接口只定义了一个方法，该方法也命名为 `ToString()`，它带有两个参数，这与 `System.Object` 版本的 `ToString()` 方法不同，它不带参数。下面是 `IFormattable` 接口的定义：

```
interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider);
}
```

这个 `ToString()` 重载方法的第一个参数是一个字符串，它指定要求的格式。换言之，它是字符串的说明符部分，该部分放在字符串的 `{}` 中，该参数最初传递给 `Console.WriteLine()` 或 `String.Format()` 方法。例如，在本例中，最初的语句如下：

```
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

在判断第一个说明符 `{0,10:E}` 时，在 `double` 变量 `d` 上调用这个重载方法，传递给它的第一个参数是 `E`。`StringBuilder.AppendFormat()` 方法传递的总是显示在原始字符串的合适格式说明符内冒号后面的文本。

本书不讨论 `ToString()` 方法的第 2 个参数，它是实现 `IFormatProvider` 接口的对象引用。这个接口

提供了 `ToString()` 接口在格式化对象时需要考虑的更多信息——一般包括文化背景信息(.NET 文化背景类似于 Windows 时区; 如果格式化货币或日期, 就需要这些信息)。如果直接从源代码中调用这个 `ToString()` 重载方法, 就需要提供这样一个对象。但 `StringBuilder.AppendFormat()` 方法为这个参数传递一个空值。如果 `formatProvider` 为空, `ToString()` 方法就要使用系统设置中指定的文化背景信息。

现在回过头来看看本例。第一个要格式化的项是一个 `double` 类, 对此要求使用指数计数法, 格式说明符为 `E`。如前所述, `StringBuilder.AppendFormat()` 方法会确保该 `double` 数的确实现 `IFormattable` 接口, 因此要调用带有两个参数的 `ToString()` 重载方法, 其第一个参数是字符串 `"E"`, 第二个参数为空。现在该 `double` 数的这个方法在实现接口时, 会考虑要求的格式和当前的文化背景, 以合适的格式返回 `double` 的字符串表示。`StringBuilder.AppendFormat()` 方法则按照需要在返回的字符串中添加前导空格, 使之共有 10 个字符。

下一个要格式化的对象是 `int`, 它不需要任何特殊的格式(格式说明符是 `{1}`)。由于没有格式要求, 因此 `StringBuilder.AppendFormat()` 方法会给该格式化字符串传递一个空引用, 并适当地响应带有两个参数的 `int.ToString()` 重载方法。因为没有特殊的格式要求, 所以也可以调用不带参数的 `ToString()` 方法。

整个字符串格式化过程如图 9-2 所示。

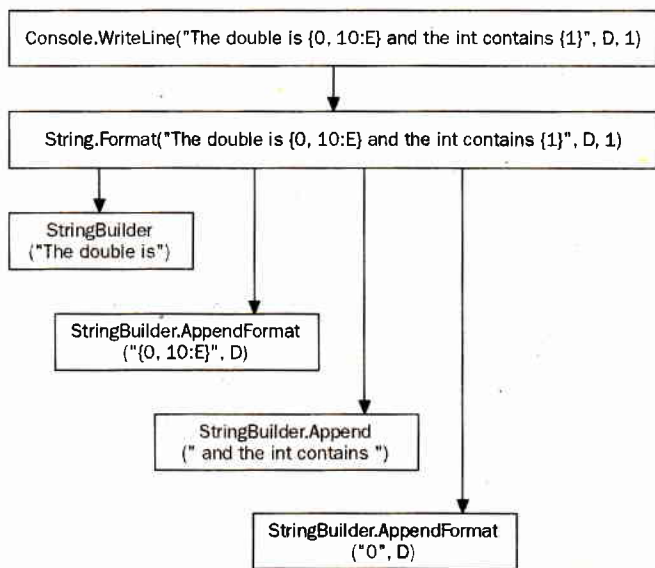


图 9-2

2. FormattableVector 示例

前面介绍了如何构造格式字符串, 下面扩展第 7 章的 `Vector` 示例, 以多种方式格式化矢量。这个示例的代码可以从 www.wrox.com 或者本书附赠光盘中找到, 文件名是 `FormattableVector.cs`。只要理解了所涉及的规则, 实际编写代码就相当简单了。我们只需要实现 `IFormattable` 接口, 提供由该接口定义的 `ToString()` 重载方法的实现代码即可。

要支持的格式说明符如下:

- `N` 应解释为一个请求, 以提供一个数字, 即矢量的模, 它是其成员的平方和, 在数学上等于 `Vector` 的长度的平方, 通常放在两个竖杠的中间: `||34.5||`。

- VE 应解释为以科学计数法显示每个成员的一个请求, 如说明符 E 应用于 double, 就可以表示为(2.3E+01, 4.5E+02, 1.0E+00)。
- IJK 应解释为以格式 23i + 450j + 1k 显示矢量的一个请求。
- 其他内容应仅返回 Vector 的默认表示方法(23, 450, 1.0)。

为了简单起见, 显示矢量时我们不以 IJK 和科学计数法的格式实现任何选项, 而是以不区分大小写的方式来测试说明符, 允许使用 ijk 而不是 IJK。注意, 使用什么字符串表示格式说明符完全取决于用户。

为此, 首先修改 Vector 结构的声明, 使之实现 IFormattable 接口:

```
struct Vector: IFormattable
{
    public double x, y, z;

    // Beginning part of Vector
```

下面添加带有两个参数的 ToString()重载方法的实现代码:

```
public string ToString(string format, IFormatProvider formatProvider)
{
    if (format == null)
    {
        return ToString();
    }

    string formatUpper = format.ToUpper();

    switch (formatUpper)
    {
        case "N":
            return "|" + Norm().ToString() + "|";
        case "VE":
            return String.Format("{0:E}, {1:E}, {2:E} ", x, y, z);
        case "IJK":
            StringBuilder sb = new StringBuilder(x.ToString(), 30);
            sb.AppendFormat(" i + ");
            sb.AppendFormat(y.ToString());
            sb.AppendFormat(" j + ");
            sb.AppendFormat(z.ToString());
            sb.AppendFormat(" k");
            return sb.ToString();
        default:
            return ToString();
    }
}
```

这就是我们要编写的代码。注意在调用任何方法前, 应防止使用格式字符串为空的参数。我们希望这个方法尽可能健壮。因为所有基元类型的格式说明符都不区分大小写, 所以其他开发人员也希望使用我们的类。对于格式说明符 VE, 需要把每个成员格式化为科学计数法, 所以再次使用 String.Format()方法。字段 x、y 和 z 都是 double 类型。对于 IJK 格式说明符, 把几个子字符串添加到字符串中, 因此使用 StringBuilder 对象来提高性能。

为了完整起见, 也可以再次使用前面开发的无参数的 ToString()重载方法:

```
public override string ToString()
{
    return "( " + x + ", " + y + ", " + z + " )";
}
```

最后, 需要添加一个 `Norm()` 方法, 计算矢量的平方(模), 因为在开发 `Vector` 结构时, 实际上没有提供这个方法:

```
public double Norm()
{
    return x*x + y*y + z*z;
}
```

下面用一些合适的测试代码测试可格式化的矢量:

```
static void Main()
{
    Vector v1 = new Vector(1,32,5);
    Vector v2 = new Vector(845.4, 54.3, -7.8);
    Console.WriteLine("\nIn IJK format,\nv1 is {0,30:IJK}\nv2 is {1,30:IJK}", v1, v2);
    Console.WriteLine("\nIn default format,\nv1 is {0,30}\nv2 is {1,30}", v1, v2);
    Console.WriteLine("\nIn VE format,\nv1 is {0,30:VE}\nv2 is {1,30:VE}", v1, v2);
    Console.WriteLine("\nNorms are:\nv1 is {0,20:N}\nv2 is {1,20:N}", v1, v2);
}
```

运行这个示例的结果如下所示:

```
FormattableVector
In IJK format,
v1 is 1 i + 32 j + 5 k
v2 is 845.4 i + 54.3 j + -7.8 k

In default format,
v1 is ( 1, 32, 5 )
v2 is ( 845.4, 54.3, -7.8 )

In VE format
v1 is ( 1.000000E+000, 3.200000E+001, 5.000000E+000 )
v2 is ( 8.454000E+002, 5.430000E+001, -7.800000E+000 )

Norms are:
v1 is || 1050 ||
v2 is || 717710.49 ||
```

这说明了选用的自定义格式说明符是正确的。

9.2 正则表达式

正则表达式作为小型技术领域的一部分在各种程序中都有着难以置信的作用, 但开发人员几乎很少使用它。正则表达式可以看作一种有特定功能的小型编程语言: 在大的字符串表达式中定位一个子字符串。它不是一种新技术, 最初它是在 `UNIX` 环境中开发的, 与 `Perl` 编程语言一起使用得比较多。Microsoft 把它移植到 `Windows` 中, 到目前为止它在脚本语言中用得比较多。但 `System`

`Text.RegularExpressions` 名称空间中的许多 .NET 类都支持正则表达式。 .NET Framework 的各个部分也使用正则表达式。例如，在 ASP.NET 验证服务器的控件中就使用了正则表达式。

对于不太熟悉正则表达式语言的读者，本节将主要解释正则表达式和相关的 .NET 类。如果您很熟悉正则表达式，就可以跳过本节，选择学习 .NET 基类的引用。注意，.NET 正则表达式引擎用于兼容 Perl 5 的正则表达式，但它有一些新功能。

9.2.1 正则表达式概述

正则表达式语言是一种专门用于字符串处理的语言。它包含两个功能：

- 一组用于标识字符类型的转义代码。您可能很熟悉 DOS 表达式中的 * 字符表示任意子字符串(例如，DOS 命令 `Dir Re*` 会列出名称以 `Re` 开头的文件)。正则表达式使用与 * 类似的许多序列来表示“任意一个字符”、“一个单词的中断”和“一个可选的字符”等。
- 一个系统，在搜索操作中，它把子字符串和中间结果的各个部分组合起来。

使用正则表达式，可以对字符串执行许多复杂而高级的操作，例如：

- 识别(可以是标记或删除)字符串中所有重复的单词，例如，把 “The computer books books” 转换为 “The computer books”。
- 把所有单词都转换为标题格式，例如，把 “this is a Title” 转换为 “This Is A Title”。
- 把长于 3 个字符的所有单词都转换为标题格式，例如，把 “this is a Title” 转换为 “This is a Title”。
- 确保句子有正确的大写形式。
- 区分 URI 的各个元素(例如，`http://www.wrox.com`，提取出协议、计算机名和文件名等)。

当然，这些都是可以在 C# 中用 `System.String` 和 `System.Text.StringBuilder` 的各种方法执行的任务。但是，在一些情况下，还需要编写相当多的 C# 代码。如果使用正则表达式，这些代码一般可以压缩为几行。实际上，是实例化了一个对象 `System.Text.RegularExpressions.RegEx`(甚至更简单，调用静态的 `RegEx()` 方法)，给它传递要处理的字符串和一个正则表达式(这是一个字符串，它包含用正则表达式语言编写的指令)，就可以了。

正则表达式字符串初看起来像是一般的字符串，但其中包含了转义序列和有特定含义的其他字符。例如，序列 `b` 表示一个字的开头和结尾(字的边界)，如果要表示正在查找以字符 `th` 开头的字，就可以编写正则表达式 `bth`(即字边界是序列 `t-h`)。如果要搜索所有以 `th` 结尾的单词，就可以编写 `th\b`(字边界是序列 `t-h`)。但是，正则表达式要比这复杂得多，包括可以在搜索操作中找到存储部分文本的工具性程序。本节仅简要介绍正则表达式的功能。



正则表达式的更多信息可参阅图书 *Beginning Regular Expressions*(ISBN: 979-0-7645-7489-4)。

假定应用程序需要把 US 电话号码转换为国际格式。在美国，电话号码的格式为 314-123-1234，常常写作(314)123-1234。在把这个国家格式转换为国际格式时，必须在电话号码的前面加上 +1(美国的国家代码)，并给区号加上圆括号：+1(314) 123-1234。在查找和替换时，这并不复杂，但如果要使用 `String` 类完成这个转换，就需要编写一些代码(这表示，必须使用 `System.String` 类的方法来编写代码)，而正则表达式语言可以构造一个短的字符串来表达上述含义。

所以，本节只有一个非常简单的示例，我们只考虑如何查找字符串中的某些子字符串，无须考虑如何修改它们。

9.2.2 RegularExpressionsPlayaround 示例

下面将开发一个小示例 `RegularExpressionsPlayaround`，通过实现并显示一些搜索的结果，说明正则表达式的一些功能，以及如何在 C# 中使用 .NET 正则表达式引擎。将在这个示例文档中使用的文本是引自 Wrox 出版社另一本有关 ASP.NET 的书籍 *Professional ASP.NET 4: in C# and VB* (2010 年出版，ISBN: 978-7-4705-0220-4)：



```
const string myText =
@"This comprehensive compendium provides a broad and thorough
investigation of all aspects of programming with ASP.NET. Entirely
revised and updated for the fourth release of .NET, this book will give
you the information you need to master ASP.NET and build a dynamic,
successful, enterprise Web application.";
```

代码段 `RegularExpressionsPlayaround.cs`



如果不考虑换行，则上面的代码是合法的 C# 代码——它说明了前缀为 @ 符号的逐字字符串的实用程序。

我们把这个文本称为输入字符串。为了说明 .NET 类的正则表达式，我们先进行一次纯文本的搜索，这次搜索不带任何转义序列或正则表达式命令。假定要查找所有的字符串 `ion`，把这个搜索字符串称为模式。使用正则表达式和上面声明的变量 `Text`，编写出下面的代码：

```
const string pattern = "ion";
MatchCollection myMatches = Regex.Matches(myText, pattern,
                                         RegexOptions.IgnoreCase |
                                         RegexOptions.ExplicitCapture);

foreach (Match nextMatch in myMatches)
{
    Console.WriteLine(nextMatch.Index);
}
```

在这段代码中，使用了 `System.Text.RegularExpressions` 名称空间中 `Regex` 类的静态方法 `Matches()`。这个方法的参数是一些输入文本、一个模式和 `RegexOptions` 枚举中的一组可选标志。在本例中，指定所有的搜索都不应区分大小写。另一个标记 `ExplicitCapture` 改变了收集匹配的方式，对于本例，这样可以使搜索的效率更高，其原因详见后面的内容(尽管它还有这里没有探讨的其他用法)。 `Matches()` 方法返回 `MatchCollection` 对象的引用。匹配是一个技术术语，表示在表达式中查找模式实例的结果，用 `System.Text.RegularExpressions.Match` 类来表示它。因此，我们返回一个包含所有匹配的 `MatchCollection`，每个匹配都用一个 `Match` 对象来表示。在上面的代码中，只是在集合中迭代，并使用 `Match` 类的 `Index` 属性，`Match` 类返回输入文本中匹配所在的索引。运行这段代码将得到 3 个匹配。表 9-4 描述了 `RegexOptions` 枚举的一些选项。

表 9-4

成 员 名	说 明
CultureInvariant	指定忽略字符串的文化背景
ExplicitCapture	修改收集匹配的方式，方法是确保把显式指定的匹配作为有效的搜索结果
IgnoreCase	忽略输入字符串的大小写
IgnorePatternWhitespace	在字符串中删除未转义的空白，使注释用英镑符号或短横线符号指定
Multiline	修改字符^和\$, 把它们应用于每一行的开头和结尾，而不仅仅应用于整个字符串的开头和结尾
RightToLeft	从右到左地读取输入字符串，而不是从左到右地读取(适合于一些亚洲语言或其他以这种方式读取的语言)
Singleline	指定句点的含义(.)，它原来表示单行模式，现在改为匹配每个字符

到目前为止，在前面的示例中，除了一些新的.NET 基类外，其他内容都不是新的。但正则表达式的功能主要取决于模式字符串。原因是模式字符串不必仅包含纯文本。如前所述，它还可以包含元字符和转义序列，其中元字符是给出命令的特定字符，而转义序列的工作方式与 C# 的转义序列相同，它们都是以反斜杠(\)开头的字符，且具有特殊的含义。

例如，假定要查找以 n 开头的字，那么可以使用转义序列\b，它表示一个字的边界(字的边界是以字母数字表中的某个字符开头，或者后面是一个空白字符或标点符号)。可以编写如下代码：

```
const string pattern = @"\bn";
MatchCollection myMatches = Regex.Matches(myText, pattern,
    RegexOptions.IgnoreCase |
    RegexOptions.ExplicitCapture);
```

注意字符串前面的符号@。要在运行时把\b 传递给.NET 正则表达式引擎，反斜杠(\)不应被 C# 编译器解释为转义序列。如果要查找以序列 ion 结尾的字，就可以使用下面的代码：

```
const string pattern = @"ion\b";
```

如果要查找以字母 a 开头，以序列 ion 结尾的所有字(在本例中它仅有一个匹配的字 application)，就必须在上面的代码中添加一些内容。显然，我们需要一个以\b a 开头，以 ion\b 结尾的模式，但中间的内容怎么办？需要告诉应用程序在 a 和 ion 中间的内容可以是任意长度的字符，只要这些字符不是空白即可。实际上，正确的模式如下所示。

```
const string pattern = @"\ba\S*ion\b";
```

使用正则表达式要习惯的一点是，对像这样怪异的字符序列见怪不怪。但这个序列的工作是非常逻辑化的。转义序列\S 表示任何不是空白字符的字符。*称为限定符，其含义是前面的字符可以重复任意次，包括 0 次。序列\S*表示任意个不是空白字符的字符。因此，上面的模式匹配以 a 开头以 ion 结尾的任何单个单词。

表 9-5 是可以使用的一些主要的特定字符或转义序列，但这个表并不完整，完整的列表请参考 MSDN 文档。

表 9-5

符 号	含 义	示 例	匹配的示例
^	输入文本的开头	^B	B, 但只能是文本中的第一个字符
\$	输入文本的结尾	X\$	X, 但只能是文本中的最后一个字符
.	除了换行符(\n)以外的所有单个字符	i.ation	isation、ization
*	可以重复 0 次或多次的前导字符	ra*t	rt、rat、raat 和 raaat 等
+	可以重复 1 次或多次的前导字符	ra+t	rat、raat 和 raaat 等(但不能是 rt)
?	可以重复 0 次或 1 次的前导字符	ra?t	只有 rt 和 rat 匹配
\s	任何空白字符	\sa	[space]a、\ta、\na (t 和 n 与 C#中的 t 和 n 含义相同)
\S	任何不是空白的字符	\SF	aF、rF、cF, 但不能是 tf
\b	字边界	ion\b	以 ion 结尾的任何字
\B	不是字边界的任意位置	\BX\B	字中间的任何 X

如果要搜索其中一个元字符, 就可以通过带有反斜杠的相应转义字符来表示。例如, “.” (一个句点) 表示除了换行字符以外的任何单个字符, 而 “\” 表示一个点。

可以把替换的字符放在方括号中, 请求匹配包含这些字符。例如, [l|c] 表示字符可以是 l 或 c。如果要搜索 map 或 man, 就可以使用序列 ma[n|p]。在方括号中, 也可以指定一个范围, 例如[a-z] 表示所有的小写字母, [A-E] 表示 A~E 之间的所有大写字母(包括字母 A 和 E), [0-9] 表示一个数字。如果要搜索一个整数(该序列只包含 0~9 的字符), 就可以编写[0-9]+



使用 “+” 字符表示至少要有这样一个数字, 但可以有多多个数字, 所以 9、83 和 854 等都是匹配的。

9.2.3 显示结果

本节编写一个示例 RegularExpressionsPlayaround, 看看正则表达式的工作方式。

该示例的核心是一个方法 WriteMatches(), 它把 MatchCollection 中的所有匹配以比较详细的格式显示出来。对于每个匹配结果, 它都会显示该匹配在输入字符串中的索引、匹配的字符串和一个略长的字符串, 其中包含匹配结果和输入文本中至多 10 个外围字符, 其中至多有 5 个字符放在匹配结果的前面, 至多 5 个字符放在匹配结果的后面(如果匹配结果的位置在输入文本的开头或结尾 5 个字符内, 则结果中匹配字符串前后的字符就会少于 5 个)。换言之, 如果要匹配的单词是 messaging, 靠近输入文本末尾的匹配结果应是 “and messaging of d”, 匹配结果的前后各有 5 个字符, 但位于输入文本的最后一个字 data 上的匹配结果就应是 “g of data” —— 匹配结果的后面只有一个字符。因为在该字符的后面是字符串的结尾。这个长字符串可以更清楚地表明正则表达式是在什么地方方找到匹配结果的:

```
static void WriteMatches(string text, MatchCollection matches)
```

```

Console.WriteLine("Original text was: \n\n" + text + "\n");
Console.WriteLine("No. of matches: " + matches.Count);

foreach (Match nextMatch in matches)
{
    int index = nextMatch.Index;
    string result = nextMatch.ToString();
    int charsBefore = (index < 5) ? index: 5;
    int fromEnd = text.Length-index-result.Length;
    int charsAfter = (fromEnd < 5) ? fromEnd: 5;
    int charsToDisplay = charsBefore + charsAfter + result.Length;

    Console.WriteLine("Index: {0}, \tString: {1}, \t{2}",
        index, result,
        text.Substring(index-charsBefore, charsToDisplay));
}
}

```

在这个方法中，处理过程是确定在较长的子字符串中有多少个字符可以显示，而无需超出输入文本的开头或结尾。注意在 `Match` 对象上使用了另一个属性 `Value`，它包含标识该匹配的字符串。而且，`RegularExpressionsPlayaround` 只包含名为 `Find1`、`Find2` 等的方法，这些方法根据本节中的示例执行某些搜索操作。例如，`Find2` 查找以 `a` 开头的任意字符串：

```

static void Find2()
{
    string text = @"This comprehensive compendium provides a broad and thorough
        investigation of all aspects of programming with ASP.NET. Entirely revised and
        updated for the 3.5 Release of .NET, this book will give you the information
        you need to master ASP.NET and build a dynamic, successful, enterprise Web
        application.";
    string pattern = @"^ba";
    MatchCollection matches = Regex.Matches(text, pattern,
        RegexOptions.IgnoreCase);
    WriteMatches(text, matches);
}

```

下面是一个简单的 `Main()` 方法，可以编辑它从而选择一个 `Find<n>()` 方法：

```

static void Main()
{
    Find1();
    Console.ReadLine();
}

```

这段代码还需要使用 `RegularExpressions` 名称空间：

```

using System;
using System.Text.RegularExpressions;

```

运行带有 `Find2()` 方法的示例，得到如下所示的结果：

```

RegularExpressionsPlayaround
Original text was:

```

This comprehensive compendium provides a broad and thorough investigation of all aspects of programming with ASP.NET. Entirely revised and updated for the 3.5 Release of .NET, this book will give you the information you need to master ASP.NET and build a dynamic, successful, enterprise Web application.

No. of matches: 1

Index: 291, String: application, Web application.

9.2.4 匹配、组合和捕获

正则表达式的一个很好的特性是可以把字符组合起来,其工作方式与C#中的复合语句一样。在C#中,可以把任意数量的语句放在花括号中,把它们组合在一起。其结果就像一个复合语句那样。在正则表达式模式中,也可以把任何字符组合起来(包括元字符和转义序列),像处理单个字符那样处理它们。唯一的区别是要使用圆括号,而不是花括号,得到的序列称为一组。

例如,模式(an)+定位任意重复的序列an。限定符“+”只应用于它前面的一个字符,但因为我们把字符组合起来了,所以它现在把重复的an作为一个单元来对待。这意味着,如果(an)+应用到输入文本bananas came to Europe late in the annals of history上,就会从bananas中识别出anan。另一方面,如果使用an+,则程序将从annals中选择ann,从bananas中选择出两个分开的an序列。表达式(an)+可以识别出an、anan、ananan等,而表达式an+可以识别出an、ann、annn等。



在上面的示例中,为什么(an)+从banana中选择的是anan,而没有把其中一个an作为一个匹配结果?因为匹配结果是不能重叠的。如果有可能重叠,在默认情况下就选择最长的匹配。

但是,组的功能要比这强大得多。在默认情况下,把模式的一部分组合为一组时,就要求正则表达式引擎按照该组来匹配,或按照整个模式来匹配。换言之,可以把组当作一个要匹配和返回的模式,如果要把字符串分解为各个部分,这种模式就非常有效。

例如,URI的格式是<protocol>://<address>[:<port>],其中端口是可选的。它的一个示例是http://www.wrox.com:4355。假定要从一个URI中提取协议、地址和端口,而且不管URI的后面是否直接有空白(但没有标点符号),那么可以使用下面的表达式:

```
\b(\S+):\/\/([^\:]+)(?:\:(\S+))?\b
```

该表达式的工作方式如下:首先,前导\b序列和尾随\b序列确保只需要考虑完全是字的文本部分。在这个文本部分中,第一组(\S+):会识别一个或多个不是空白的字符,其后是:。在HTTP URI的开头会识别出http://。花括号表示把http存储为一组。后面的序列([^\:]+)则在上述URI中识别字符串www.wrox.com,该组在遇到词的结尾(结束\b)时或标记另一组的冒号(:)时结束。

下一个组识别端口(本例是:4355)。后面的“?”表示该组在匹配过程中是可选的,如果没有:xxxx,就不会妨碍匹配的标记。这非常重要,因为端口号在URI中一般不指定,实际上,在大多数情况下,URI是没有端口号的。但是,事情会比较复杂。我们希望指定冒号可以出现,也可以不出现,但不希望把这个冒号也存储在组中。为此,可以嵌套两组:内部(\S+)组识别冒号后面的内容(本例中是4355)。外面组包含内部组,内部组的前面是一个冒号,该组又在序列“?:”的后面。这个序列表示该组不应保存(只需要保存4355,不需要保存:4355)。不要把这两个冒号混淆了,第一个冒号是序列

“?:”的一部分,表示不保存该组,第二个冒号是要搜索的文本。

在下面的字符串上运行该模式,得到的匹配是 `http://www.wrox.com`。

```
Hey I've just found this amazing URI at
http:// what was it --oh. yes http://www.wrox.com
```

在这个匹配过程中,找到了刚才提及的 3 组,还有第 4 组表示匹配本身。理论上,每个组都可以选择 0 次、1 次或多次匹配。单个的匹配就称为捕获。在第一组(S+)中,有一个捕获 `http`,第二组也有一个捕获 `www.wrox.com`,但第 3 组没有捕获,因为在这个 URI 中没有端口号。

注意,该字符串包含第二个 `http://`。虽然它匹配第一组,但它不会被该搜索过程捕获,因为整个搜索表达式不匹配于这部分文本。

虽然前面没有介绍使用组和捕获的任何 C# 示例,但下面提到的 .NET 类 `RegularExpressions` 就通过 `Group` 类和 `Capture` 类支持组和捕获。`GroupCollection` 类和 `CaptureCollection` 类分别表示组和捕获的集合,`Match` 类提供一个 `Groups` 属性,它返回相应的 `GroupCollection` 对象。`Group` 类也相应地实现一个 `Captures` 属性,该属性返回 `CaptureCollection` 对象。这些对象之间的关系如图 9-3 所示。

也许只希望把一些字符组合起来后,而不是每次都会返回一个 `Group` 对象。如果只是希望把一些字符组合起来,作为搜索模式的一部分,实例化对象就会浪费相当大的系统开销。对于单个组,可以以字符序列“?:”开头,禁止实例化对象,就像 URI 示例那样。而对于所有组,可以在 `Regex.Matches()` 方法上指定 `RegexOptions.ExplicitCaptures` 标志,如同前面的示例那样。

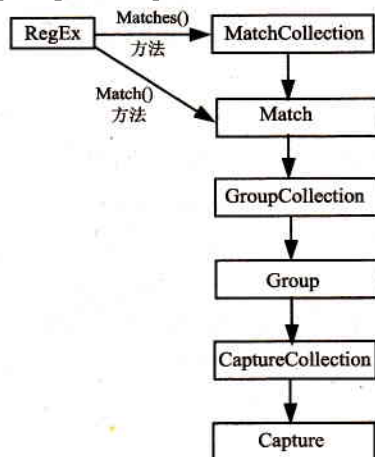


图 9-3

9.3 小结

在使用 .NET Framework 时,可用的数据类型相当多。在应用程序(特别是关注数据提交和检索的应用程序)中,最常用的一种类型就是 `String` 数据类型。`String` 非常重要,这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理 `String` 数据类型的原因。

过去在使用字符串时,常常需要通过连接来分解字符串。而在 .NET Framework 中,可以使用 `StringBuilder` 类完成许多这类任务,而且性能更好。

最后,使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种最佳工具。

第 10 章

集 合

本章内容:

- 理解集合接口和类型
- 使用列表、队列和栈
- 使用链表和有序列表
- 使用字典和集
- 使用位数组和位矢量
- 评估性能

第 6 章介绍了数组和 `Array` 类实现的接口。数组的大小是固定的。如果元素个数是动态的, 就应使用集合类。

`List<T>` 是与数组相当的集合类。还有其他类型的集合: 队列、栈、链表和字典。



.NET Framework 1.0 包含非泛型集合类, 如 `ArrayList` 和 `HashTable`。CLR 2.0 添加了对泛型类和泛型集合类的支持。本章的重点是新一组的集合类, 而忽略旧集合类, 因为旧集合类很少需要在新应用程序中使用。

10.1 集合接口和类型

大多数集合类都可在 `System.Collections` 和 `System.Collections.Generic` 名称空间中找到。泛型集合类位于 `System.Collections.Generic` 名称空间中; 专用于特定类型的集合类位于 `System.Collections.Specialized` 名称空间中。线程安全的集合类位于 `System.Collections.Concurrent` 名称空间中。

当然, 组合集合类还有其他方式。集合可以根据集合类实现的接口组合为列表、集合和字典。



接口 `IEnumerable` 和 `IEnumerator` 的内容详见第 6 章。

集合和列表实现的接口如表 10-1 所示。

表 10-1

接 口	说 明
IEnumerable<T>	如果将 foreach 语句用于集合, 就需要 IEnumerable 接口。这个接口定义了方法 GetEnumerator(), 它返回一个实现了 IEnumerator 接口的枚举
ICollection<T>	ICollection<T>接口由泛型集合类实现。使用这个接口可以获得集合中的元素个数(Count 属性), 把集合复制到数组中(CopyTo()方法), 还可以从集合中添加和删除元素(Add(), Remove(), Clear())
IList<T>	IList<T>接口用于可通过位置访问其中的元素列表, 这个接口定义了一个索引器, 可以在集合的指定位置插入或删除某些项(Insert()和 RemoveAt()方法)。IList<T>接口派生自 ICollection<T>接口
ISet<T>	ISet<T>接口是 .NET 4 中新增的。实现这个接口的集允许合并不同的集, 获得两个集的交集, 检查两个集是否重叠。ISet<T>接口派生自 ICollection<T>接口
IDictionary<TKey, TValue>	IDictionary<TKey, TValue>接口由包含键和值的泛型集合类实现。使用这个接口可以访问所有的键和值, 使用键类型的索引器可以访问某些项, 还可以添加或删除某些项
ILookup<TKey, TValue>	ILookup<TKey, TValue>接口类似于 IDictionary<TKey, TValue>接口, 实现该接口的集合有键和值, 且可以通过一个键包含多个值
IComparer<T>	接口 IComparer<T>由比较器实现, 通过 Compare()方法给集合中的元素排序
IEqualityComparer<T>	接口 IEqualityComparer<T>由一个比较器实现, 该比较器可用于字典中的键。使用这个接口, 可以对对象进行相等性比较。在 .NET 4 中, 这个接口也由数组和元组实现
IProducerConsumerCollection<T>	IProducerConsumerCollection<T>接口是 .NET 4 中新增的, 它支持新的线程安全的集合类

10.2 列表

.NET Framework 为动态列表提供了泛型类 List<T>。这个类实现了 IList、ICollection、IEnumerable、IList<T>、ICollection<T>和 IEnumerable<T>接口。

下面的例子将 Racer 类中的成员用作要添加到集合中的元素, 以表示一级方程式的一位赛车手。这个类有 5 个属性: Id、Firstname、Lastname、Country 和 Wins 的次数。在该类的构造函数中, 可以传递赛手的姓名和获胜次数, 以设置成员。重写 ToString()方法是为了返回赛手的姓名。Racer 类也实现了泛型接口 IComparable<T>, 为 Racer 类中的元素排序, 还实现了 IFormattable 接口。



```
[Serializable]
public class Racer: IComparable < Racer > , IFormattable
{
    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Country { get; set; }
    public int Wins { get; set; }
    public Racer(int id, string firstName, string lastName,
                string country = null, int wins = 0)
    {
        this.Id = id;
```

```

        this.FirstName = firstName;
        this.LastName = lastName;
        this.Country = country;
        this.Wins = wins;
    }

    public override string ToString()
    {
        return String.Format("{0} {1}", FirstName, LastName);
    }

    public string ToString(string format, IFormatProvider formatProvider)
    {
        if (format == null) format = "N";
        switch (format.ToUpper())
        {
            case "N": // name
                return ToString();
            case "F": // first name
                return FirstName;
            case "L": // last name
                return LastName;
            case "W": // Wins
                return String.Format("{0}, Wins: {1}", ToString(), Wins);
            case "C": // Country
                return String.Format("{0}, Country: {1}", ToString(), Country);
            case "A": // All
                return String.Format("{0}, {1} Wins: {2}", ToString(), Country, Wins);
            default:
                throw new FormatException(String.Format(formatProvider,
                    "Format {0} is not supported", format));
        }
    }

    public string ToString(string format)
    {
        return ToString(format, null);
    }

    public int CompareTo(Racer other)
    {
        int compare = this.LastName.CompareTo(other.LastName);
        if (compare == 0)
            return this.FirstName.CompareTo(other.FirstName);
        return compare;
    }
}

```

代码段 ListSamples/Racer.cs

10.2.1 创建列表

调用默认的构造函数，就可以创建列表对象。在泛型类 `List<T>` 中，必须为声明为列表的值指定类型。下面的代码说明了如何声明一个包含 `int` 的 `List<T>` 泛型类和一个包含 `Racer` 元素的列表。

`ArrayList` 是一个非泛型列表，它可以将任意 `Object` 类型作为其元素。

使用默认的构造函数创建一个空列表。元素添加到列表中后，列表的容量就会扩大为可容纳 4 个元素。如果添加了第 5 个元素，列表的大小就重新设置为包含 8 个元素。如果 8 个元素还不够，列表的大小就重新设置为包含 16 个元素。每次都会将列表的容量重新设置为原来的 2 倍。

```
var intList = new List<int>();
var racers = new List<Racer>();
```

如果列表的容量改变了，整个集合就要重新分配到一个新的内存块中。在 `List<T>` 泛型类的实现代码中，使用了一个 `T` 类型的数组。通过重新分配内存，创建一个新数组，`Array.Copy()` 方法将旧数组中的元素复制到新数组中。为节省时间，如果事先知道列表中元素的个数，就可以用构造函数定义其容量。下面创建了一个容量为 10 个元素的集合。如果该容量不足以容纳要添加的元素，就把集合的大小重新设置为包含 20 或 40 个元素，每次都是原来的 2 倍。

```
List<int> intList = new List<int>(10);
```

使用 `Capacity` 属性可以获取和设置集合的容量。

```
intList.Capacity = 20;
```

容量与集合中元素的个数不同。集合中的元素个数可以用 `Count` 属性读取。当然，容量总是大于或等于元素个数。只要不把元素添加到列表中，元素个数就是 0。

```
Console.WriteLine(intList.Count);
```

如果已经将元素添加到列表中，且不希望添加更多的元素，就可以调用 `TrimExcess()` 方法，去除不需要的容量。但是，因为重新定位需要时间，所以如果元素个数超过了容量的 90%，`TrimExcess()` 方法就什么也不做。

```
intList.TrimExcess();
```

1. 集合初始值设定项

还可以使用集合初始值设定项给集合赋值。集合初始值设定项的语法类似于第 6 章介绍的数组初始值设定项。使用集合初始值设定项，可以在初始化集合时，在花括号中给集合赋值：

```
var intList = new List<int>() {1, 2};
var stringList = new List<string>() {"one", "two"};
```



集合初始值设定项没有反映在已编译的程序集的 IL 代码中。编译器会把集合初始值设定项变成对于初始值设定项列表中的每一项调用 `Add()` 方法。

2. 添加元素

使用 `Add()` 方法可以给列表添加元素，如下所示。实例化的泛型类型定义了 `Add()` 方法的参数类型：

```
var intList = new List<int>();
```

```
intList.Add(1);
intList.Add(2);

var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

把 `racers` 变量定义为 `List<Racer>` 类型。使用 `new` 运算符创建相同类型的一个新对象。因为类 `List<T>` 用具体类 `Racer` 来实例化, 所以现在只有 `Racer` 对象可以用 `Add()` 方法添加。在下面的示例代码中, 创建了 5 个一级方程式赛车手, 并把它们添加到集合中。前 3 个用集合初始值设定项添加, 后两个通过显式调用 `Add()` 方法来添加。



可从
wrox.com
下载源代码

```
var graham = new Racer(7, "Graham", "Hill", "UK", 14);
var emerson = new Racer(13, "Emerson", "Fittipaldi", "Brazil", 14);
var mario = new Racer(16, "Mario", "Andretti", "USA", 12);

var racers = new List<Racer>(20) {graham, emerson, mario};

racers.Add(new Racer(24, "Michael", "Schumacher", "Germany", 91));
racers.Add(new Racer(27, "Mika", "Hakkinen", "Finland", 20));
```

代码段 ListSamples/Program.cs

使用 `List<T>` 类的 `AddRange()` 方法, 可以一次给集合添加多个元素。因为 `AddRange()` 方法的参数是 `IEnumerable<T>` 类型的对象, 所以也可以传递一个数组, 如下所示:

```
racers.AddRange(new Racer[] {
    new Racer(14, "Niki", "Lauda", "Austria", 25),
    new Racer(21, "Alain", "Prost", "France", 51)});
```



集合初始值设定项只能在声明集合时使用。`AddRange()` 方法则可以在初始化集合后调用。

如果在实例化列表时知道集合的元素个数, 就也可以将实现 `IEnumerable<T>` 类型的任意对象传递给类的构造函数。这非常类似于 `AddRange()` 方法:

```
var racers =
    new List<Racer>(new Racer[] {
        new Racer(12, "Jochen", "Rindt", "Austria", 6),
        new Racer(22, "Ayrton", "Senna", "Brazil", 41) });
```

3. 插入元素

使用 `Insert()` 方法可以在指定位置插入元素:

```
racers.Insert(3, new Racer(6, "Phil", "Hill", "USA", 3));
```

方法 `InsertRange()` 提供了插入大量元素的功能, 类似于前面的 `AddRange()` 方法。

如果索引大于集合中的元素个数, 就抛出 `ArgumentOutOfRangeException` 类型的异常。

4. 访问元素

实现了 `IList` 和 `IList<T>` 接口的所有类都提供了一个索引器，所以可以使用索引器，方法是通过传送元素号来访问元素。第一个元素可以用索引值 0 来访问。指定 `racers[3]`，可以访问列表中的第 4 个元素：

```
Racer r1 = racers[3];
```

可以用 `Count` 属性确定元素个数，再使用 `for` 循环遍历集合中的每个元素，并使用索引器访问每一项：

```
for (int i = 0; i < racers.Count; i++)
{
    Console.WriteLine(racers[i]);
}
```



可以通过索引访问的集合类有 `ArrayList`、`StringCollection` 和 `List<T>`。

因为 `List<T>` 集合类实现了 `IEnumerable` 接口，所以也可以使用 `foreach` 语句遍历集合中的元素。

```
foreach (Racer r in racers)
{
    Console.WriteLine(r);
}
```



编译器解析 `foreach` 语句时，利用了 `IEnumerable` 和 `IEnumerator` 接口，参见第 6 章。

除了使用 `foreach` 语句之外，`List<T>` 类还提供了 `ForEach()` 方法，该方法用 `Action<T>` 参数声明。

```
public void ForEach(Action<T> action);
```

实现 `ForEach()` 方法的代码如下。`ForEach()` 方法遍历集合中的每一项，调用作为每一项的参数传递的方法。

```
public class List<T> : IList<T>
{
    private T[] items;

    //...

    public void ForEach(Action<T> action)
    {
        if (action == null) throw new ArgumentNullException("action");

        foreach (T item in items)
        {
            action(item);
        }
    }
}
```

```

    }
    //...
}

```

为了给 `ForEach()` 方法传递一个方法, `Action<T>` 声明为一个委托, 该委托定义了一个返回类型为 `void`、参数为 `T` 的方法。

```
public delegate void Action<T> (T obj);
```

在 `Racer` 项的列表中, `ForEach()` 方法的处理程序必须声明为以 `Racer` 对象作为参数, 返回类型是 `void`。

```
public void ActionHandler(Racer obj);
```

因为 `Console.WriteLine()` 方法的一个重载版本将 `Object` 作为参数, 所以可以将这个方法的地址传送给 `ForEach()` 方法, 把该集合的每个赛手写入控制台:

```
racers.ForEach(Console.WriteLine);
```

也可以编写一个 `Lambda` 表达式, 它将 `Racer` 对象作为参数, 其实现代码使用了 `Console.WriteLine()` 方法。这里, 格式 `A` 由 `IFormattable` 接口的 `ToString()` 方法用于显示赛手的所有信息:

```
racers.ForEach(r => Console.WriteLine("{0:A}", r));
```



Lambda 表达式详见第 8 章。

5. 删除元素

删除元素时, 可以利用索引, 也可以传递要删除的元素。下面的代码通过把 3 传递给 `RemoveAt()` 方法, 删除第 4 个元素:

```
racers.RemoveAt(3);
```

也可以直接将 `Racer` 对象传递给 `Remove()` 方法, 来删除这个元素。按索引删除比较快, 因为必须在集合中搜索要删除的元素。`Remove()` 方法先在集合中搜索, 用 `IndexOf()` 方法获取元素的索引, 再使用该索引删除元素。`IndexOf()` 方法先检查元素类型是否实现了 `IEquatable<T>` 接口。如果是, 就调用这个接口的 `Equals()` 方法, 确定集合中的元素是否等于传递给 `Equals()` 方法的元素。如果没有实现这个接口, 就使用 `Object` 类的 `Equals()` 方法比较这些元素。`Object` 类中 `Equals()` 方法的默认实现代码对值类型进行按位比较, 对引用类型只比较其引用。



第 7 章介绍了如何重写 `Equals()` 方法。

这里从集合中删除了变量 `graham` 引用的赛手。变量 `graham` 是前面在填充集合时创建的。因为 `IEquatable<T>` 接口和 `Object.Equals()` 方法都没有在 `Racer` 类中重写, 所以不能用要删除元素的内容相

同创建一个新对象，再把它传递给 `Remove()` 方法。

```
if (!racers.Remove(gham))
{
    Console.WriteLine(
        "object not found in collection");
}
```

`RemoveRange()` 方法可以从集合中删除许多元素。它的第一个参数指定了开始删除的元素索引，第二个参数指定了要删除的元素个数。

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

要从集合中删除有指定特性的所有元素，可以使用 `RemoveAll()` 方法。这个方法在搜索元素时使用下面将讨论的 `Predicate<T>` 参数。要删除集合中的所有元素，可以使用 `ICollection<T>` 接口定义的 `Clear()` 方法。

6. 搜索

有不同的方式在集合中搜索元素。可以获得要查找的元素的索引，或者搜索元素本身。可以使用的方法有 `IndexOf()`、`LastIndexOf()`、`FindIndex()`、`FindLastIndex()`、`Find()` 和 `FindLast()`。如果只检查元素是否存在，`List<T>` 类就提供了 `Exists()` 方法。

`IndexOf()` 方法需要将一个对象作为参数，如果在集合中找到该元素，这个方法就返回该元素的索引。如果没有找到该元素，就返回 -1。`IndexOf()` 方法使用 `IEquatable<T>` 接口来比较元素。

```
int index1 = racers.IndexOf(mario);
```

使用 `IndexOf()` 方法，还可以指定不需要搜索整个集合，但必须指定从哪个索引开始搜索以及比较时要迭代的元素个数。

除了使用 `IndexOf()` 方法搜索指定的元素之外，还可以搜索有某个特性的元素，该特性可以用 `FindIndex()` 方法来定义。`FindIndex()` 方法需要一个 `Predicate` 类型的参数：

```
public int FindIndex(Predicate<T> match);
```

`Predicate<T>` 类型是一个委托，该委托返回一个布尔值，并且需要把类型 `T` 作为参数。这个委托的用法与 `ForEach()` 方法中的 `Action` 委托类似。如果 `Predicate<T>` 委托返回 `true`，就表示有一个匹配元素，并且找到了相应的元素。如果它返回 `false`，就表示没有找到元素，搜索将继续。

```
public delegate bool Predicate<T>(T obj);
```

在 `List<T>` 类中，把 `Racer` 对象作为类型 `T`，所以可以将一个方法(该方法将类型 `Racer` 定义为一个参数且返回一个布尔值)的地址传递给 `FindIndex()` 方法。查找指定国家的第一个赛手时，可以创建如下所示的 `FindCountry` 类。`FindCountryPredicate()` 方法的签名和返回类型通过 `Predicate<T>` 委托定义。`Find()` 方法使用变量 `country` 搜索用 `FindCountry` 类的构造函数定义的某个国家。

```
public class FindCountry
{
```

```

public FindCountry(string country)
{
    this.country = country;
}
private string country;

public bool FindCountryPredicate(Racer racer)
{
    if (racer == null) throw new ArgumentNullException("racer");

    return racer.Country == country;
}
}

```

使用 `FindIndex()` 方法可以创建 `FindCountry()` 方法的一个新实例，把表示一个国家的字符串传递给构造函数，再传递 `Find()` 方法的地址。`FindIndex()` 方法成功完成后，`index2` 就包含集合中赛手的 `Country` 属性设置为 `Finland` 的第一项的索引。

```

int index2 = racers.FindIndex(new FindCountry("Finland").
    FindCountryPredicate);

```

除了用处理程序方法创建类之外，还可以在这里创建 `Lambda` 表达式。结果与前面完全相同。现在在 `Lambda` 表达式定义了实现代码，来搜索 `Country` 属性设置为 `Finland` 的元素。

```

int index3 = racers.FindIndex(r => r.Country == "Finland");

```

与 `IndexOf()` 方法类似，使用 `FindIndex()` 方法也可以指定搜索开始的索引和要遍历的元素个数。为了从集合中的最后一个元素开始向前搜索某个索引，可以使用 `FindLastIndex()` 方法。

`FindIndex()` 方法返回所查找元素的索引。除了获得索引之外，还可以直接获得集合中的元素。`Find()` 方法需要一个 `Predicate<T>` 类型的参数，这与 `FindIndex()` 方法类似。下面的 `Find()` 方法搜索列表中 `Firstname` 属性设置为 `Niki` 的第一个赛手。当然，也可以实现 `FindLast()` 方法，查找与 `Predicate<T>` 类型匹配的最后一项。

```

Racer racer = racers.Find(r => r.FirstName == "Niki");

```

要获得与 `Predicate<T>` 类型匹配的所有项，而不是一项，可以使用 `FindAll()` 方法。`FindAll()` 方法使用的 `Predicate<T>` 委托与 `Find()` 和 `FindIndex()` 方法相同。`FindAll()` 方法在找到第一项后，不会停止搜索，而是继续迭代集合中的每一项，并返回 `Predicate<T>` 类型是 `true` 的所有项。

这里调用了 `FindAll()` 方法，返回 `Wins` 属性设置为大于 20 的整数的所有 `racer` 项。从 `bigWinners` 列表中引用所有赢得 20 多场比赛的赛手。

```

List<Racer> bigWinners = racers.FindAll(r => r.Wins > 20);

```

用 `foreach` 语句遍历 `bigWinners` 变量，结果如下：

```

foreach (Racer r in bigWinners)
{
    Console.WriteLine("{0:A}", r);
}
Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25

```

Alain Prost, France Wins: 51

这个结果没有排序, 但这是下一步要做的工作。

7. 排序

List<T>类可以使用 Sort()方法对元素排序。Sort()方法使用快速排序算法, 比较所有的元素, 直到整个列表排好序为止。

Sort()方法使用了几个重载的方法。可以传递给它的参数有泛型委托 Comparison<T>和泛型接口 IComparer<T>, 以及包含泛型接口 IComparer<T>的一个范围值。

```
public void List<T> .Sort();
public void List<T> .Sort(Comparison<T>);
public void List<T> .Sort(IComparer<T>);
public void List<T> .Sort(Int32, Int32, IComparer<T>);
```

只有集合中的元素实现了 IComparable 接口, 才能使用不带参数的 Sort()方法。

Racer 类实现了 IComparable<T>接口, 可以按姓氏对赛手排序:

```
racers.Sort();
racers.ForEach(Console.WriteLine);
```

如果需要按照元素类型不默认支持的方式排序, 就应使用其他技术, 例如传递一个实现了 IComparer<T>接口的对象。

RacerComparer 类为 Racer 类型实现了接口 IComparer<T>。这个类允许按名字、姓氏、国籍或获胜次数排序。排序的种类用内部枚举类型 CompareType 定义。CompareType 枚举类型用 RacerComparer 类的构造函数设置。IComparer<Racer>接口定义了排序所需的 Compare()方法。在这个方法的实现代码中, 使用了 string 和 int 类型的 CompareTo()方法。



可从
wrox.com
下载源代码

```
public class RacerComparer: IComparer<Racer>
{
    public enum CompareType
    {
        FirstName,
        LastName,
        Country,
        Wins
    }

    private CompareType compareType;
    public RacerComparer(CompareType compareType)
    {
        this.compareType = compareType;
    }

    public int Compare(Racer x, Racer y)
    {
        if (x == null) throw new ArgumentNullException("x");
        if (y == null) throw new ArgumentNullException("y");

        int result;
        switch (compareType)
```

```

    case CompareType.FirstName:
        return x.FirstName.CompareTo(y.FirstName);
    case CompareType.LastName:
        return x.LastName.CompareTo(y.LastName);
    case CompareType.Country:
        if ((result = x.Country.CompareTo(y.Country)) == 0)
            return x.LastName.CompareTo(y.LastName);
        else
            return result;
    case CompareType.Wins:
        return x.Wins.CompareTo(y.Wins);
    default:
        throw new ArgumentException("Invalid Compare Type");
}

```

代码段 ListSamples/RacerComparer.cs

现在，可以对 `RacerComparer` 类的一个实例使用 `Sort()` 方法。传递枚举 `RacerComparer.CompareType.Country`，按属性 `Country` 对集合排序：

```

racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
racers.ForEach(Console.WriteLine);

```

排序的另一种方式是使用重载的 `Sort()` 方法，该方法需要一个 `Comparison<T>` 委托：

```

public void List<T> .Sort(Comparison<T>);

```

`Comparison<T>` 是一个方法的委托，该方法有两个 `T` 类型的参数，返回类型为 `int`。如果参数值相等，该方法就必须返回 0。如果第一个参数比第二个小，它就必须返回一个小于 0 的值；否则，必须返回一个大于 0 的值。

```

public delegate int Comparison < T > (T x, T y);

```

现在可以把一个 `Lambda` 表达式传递给 `Sort()` 方法，按获胜次数排序。两个参数的类型是 `Racer`，在其实现代码中，使用 `int` 类型的 `CompareTo()` 方法比较 `Wins` 属性。在实现代码中，因为以逆序方式使用 `r2` 和 `r1`，所以获胜次数以降序方式排序。调用方法之后，完整的赛手列表就按赛手的获胜次数排序。

```

racers.Sort((r1, r2) => r2.Wins.CompareTo(r1.Wins));

```

也可以调用 `Reverse()` 方法，逆转整个集合的顺序。

8. 类型转换

使用 `List<T>` 类的 `ConvertAll<TOutput>()` 方法，可以把所有类型的集合转换为另一种类型。`ConvertAll<TOutput>()` 方法使用一个 `Converter` 委托，该委托的定义如下：

```

public sealed delegate TOutput Converter<TInput, TOutput>(TInput from);

```

泛型类型 `TInput` 和 `TOutput` 用于转换。`TInput` 是委托方法的参数，`TOutput` 是返回类型。

在这个例子中，所有的 `Racer` 类型都应转换为 `Person` 类型。`Racer` 类型包含姓氏、名字、国籍和获胜次数，而 `Person` 类型只包含名字。为了进行转换，可以忽略赛手的国籍和获胜次数，但姓名必须转换：

```
[Serializable]
public class Person
{
    private string name;

    public Person(string name)
    {
        this.name = name;
    }

    public override string ToString()
    {
        return name;
    }
}
```

转换时调用了 `racers.ConvertAll<Person>()` 方法。这个方法的参数定义为一个 `Lambda` 表达式，其参数的类型是 `Racer`，返回类型是 `Person`。在 `Lambda` 表达式的实现代码中，创建并返回了一个新的 `Person` 对象。对于 `Person` 对象，把 `Firstname` 和 `Lastname` 传递给构造函数：

```
List<Person> persons =
    racers.ConvertAll<Person>(
        r => new Person(r.Firstname + " " + r.Lastname));
```

转换的结果是一个列表，其中包含转换过的 `Person` 对象：类型为 `List<Person>` 的 `persons` 列表。

10.2.2 只读集合

创建集合后，它们就是可读写的。当然，集合必须是可读写的，否则就不能给它们填充值了。但是，在填充完集合后，可以创建只读集合。`List<T>` 集合的 `AsReadOnly()` 方法返回 `ReadOnlyCollection<T>` 类型的对象。`ReadOnlyCollection<T>` 类实现的接口与 `List<T>` 集合相同，但所有修改集合的方法和属性都抛出 `NotSupportedException` 异常。

10.3 队列

队列是其元素以先进先出(FIFO)的方式来处理的集合。先放入队列中的元素会先读取。队列的例子有在机场排的队列、人力资源部中等待处理求职信的队列和打印队列中等待处理的打印任务，以及按循环方式等待 CPU 处理的线程。另外，还常常有元素根据其优先级来处理的队列。例如，在机场的队列中，商务舱乘客的处理要优先于经济舱的乘客。这里可以使用多个队列，一个队列对应一个优先级。在机场，这很常见，因为商务舱乘客和经济舱乘客有不同的登记队列。打印队列和线程也是这样。可以为一组队列建立一个数组，数组中的一项代表一个优先级。在每个数组项中都有一个队列，其中按照 FIFO 的方式进行处理。



本章的后面将使用链表的另一种实现方式，来定义优先级列表。

队列使用 `System.Collections.Generic` 名称空间中的泛型类 `Queue<T>` 实现。在内部，`Queue<T>` 类使用 `T` 类型的数组，这类似于 `List<T>` 类型。它实现 `ICollection` 和 `IEnumerable<T>` 接口，但没有实现 `ICollection<T>` 接口，因为这个接口定义的 `Add()` 和 `Remove()` 方法不能用于队列。

因为 `Queue<T>` 类没有实现 `IList<T>` 接口，所以不能用索引器访问队列。队列只允许在队列中添加元素，该元素会放在队列的尾部(使用 `Enqueue()` 方法)，从队列的头部获取元素(使用 `Dequeue()` 方法)。

图 10-1 显示了队列的元素。`Enqueue()` 方法在队列的一端添加元素，`Dequeue()` 方法在队列的另一端读取和删除元素。用 `Dequeue()` 方法读取元素，将同时从队列中删除该元素。再次调用 `Dequeue()` 方法，会删除队列中的下一项。

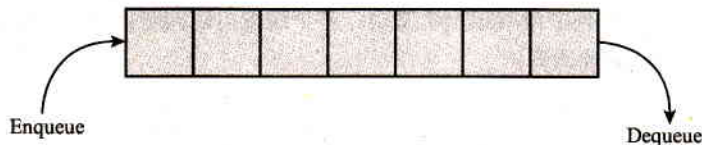


图 10-1

`Queue<T>` 类的方法如表 10-2 所示。

表 10-2

Queue<T>类的成员	说 明
Count	Count 属性返回队列中的元素个数
Enqueue()	Enqueue() 方法在队列一端添加一个元素
Dequeue()	Dequeue() 方法在队列的头部读取和删除一个元素。如果在调用 Dequeue() 方法时，队列中不再有元素，就抛出一个 <code>InvalidOperationException</code> 类型的异常
Peek()	Peek() 方法从在队列的头部读取一个元素，但不删除它
TrimExcess()	TrimExcess() 方法重新设置队列的容量。Dequeue() 方法从队列中删除元素，但它不会重新设置队列的容量。要从队列的头部去除空元素，应使用 TrimExcess() 方法

在创建队列时，可以使用与 `List<T>` 类型类似的构造函数。虽然默认的构造函数会创建一个空队列，但也可以使用构造函数指定容量。在把元素添加到队列中时，如果没有定义容量，容量就会递增，从而包含 4、8、16 和 32 个元素。类似于 `List<T>` 类，队列的容量也总是根据需要成倍增加。非泛型类 `Queue` 的默认构造函数与此不同，它会创建一个包含 32 项的空数组。使用构造函数的重载版本，还可以将实现了 `IEnumerable<T>` 接口的其他集合复制到队列中。

下面的文档管理应用程序示例说明了 `Queue<T>` 类的用法。使用一个线程将文档添加到队列中，用另一个线程从队列中读取文档，并处理它们。

存储在队列中的项是 `Document` 类型。`Document` 类定义了标题和内容：



可从
wrox.com
下载源代码

```
public class Document
{
    public string Title { get; private set; }
    public string Content { get; private set; }

    public Document(string title, string content)
    {
        this.Title = title;
        this.Content = content;
    }
}
```

代码段 QueueSample/Document.cs

DocumentManager 类是 **Queue<T>** 类外面的一层。**DocumentManager** 类定义了如何处理文档：用 **AddDocument()** 方法将文档添加到队列中，用 **GetDocument()** 方法从队列中获得文档。

在 **AddDocument()** 方法中，用 **Enqueue()** 方法把文档添加到队列的尾部。在 **GetDocument()** 方法中，用 **Dequeue()** 方法从队列中读取第一个文档。多个线程可以同时访问 **DocumentManager** 类，所以用 **lock** 语句锁定对队列的访问。



线程和 **lock** 语句参见第 20 章。

IsDocumentAvailable 是一个只读类型的布尔属性，如果队列中还有文档，它就返回 **true**，否则返回 **false**。



可从
wrox.com
下载源代码

```
public class DocumentManager
{
    private readonly Queue<Document> documentQueue = new Queue<Document>();

    public void AddDocument(Document doc)
    {
        lock (this)
        {
            documentQueue.Enqueue(doc);
        }
    }

    public Document GetDocument()
    {
        Document doc = null;
        lock (this)
        {
            doc = documentQueue.Dequeue();
        }
        return doc;
    }

    public bool IsDocumentAvailable
    {
        get
        {
            return documentQueue.Count > 0;
        }
    }
}
```

代码段 QueueSample/DocumentManager.cs

`ProcessDocuments` 类在一个单独的线程中处理队列中的文档。能从外部访问的唯一方法是 `Start()`。在 `Start()` 方法中，实例化了一个新线程。创建一个 `ProcessDocuments` 对象，来启动线程，定义 `Run()` 方法作为线程的启动方法。`ThreadStart` 是一个委托，它引用了由线程启动的方法。在创建 `Thread` 对象后，就调用 `Thread.Start()` 方法来启动线程。

使用 `ProcessDocuments` 类的 `Run()` 方法定义一个无限循环。在这个循环中，使用属性 `IsDocumentAvailable` 确定队列中是否还有文档。如果队列中还有文档，就从 `DocumentManager` 类中提取文档并处理。这里的处理仅是把信息写入控制台。在真正的应用程序中，文档可以写入文件、数据库，或通过网络发送。



可从
wrox.com
下载源代码

```
<public class ProcessDocuments
{
    public static void Start(DocumentManager dm)
    {
        new Thread(new ProcessDocuments(dm).Run).Start();
    }

    protected ProcessDocuments(DocumentManager dm)
    {
        documentManager = dm;
    }

    private DocumentManager documentManager;

    protected void Run()
    {
        while (true)
        {
            if (documentManager.IsDocumentAvailable)
            {
                Document doc = documentManager.GetDocument();
                Console.WriteLine("Processing document {0}", doc.Title);
            }
            Thread.Sleep(new Random().Next(20));
        }
    }
}
```

代码段 QueueSample/ProcessDocuments.cs

在应用程序的 `Main()` 方法中，实例化一个 `DocumentManager` 对象，启动文档处理线程。接着创建 1000 个文档，并添加到 `DocumentManager` 对象中：



可从
wrox.com
下载源代码

```
class Program
{
    static void Main()
    {
        var dm = new DocumentManager();
```

```
ProcessDocuments.Start(dm);

// Create documents and add them to the DocumentManager
for (int i = 0; i < 1000; i++)
{
    Document doc = new Document("Doc " + i.ToString(), "content");
    dm.AddDocument(doc);
    Console.WriteLine("Added document {0}", doc.Title);
    Thread.Sleep(new Random().Next(20));
}
}
```

代码段 QueueSample/Program.cs

在启动应用程序时，会在队列中添加和删除文档，输出如下所示：

```
Added document Doc 279
Processing document Doc 236
Added document Doc 280
Processing document Doc 237
Added document Doc 281
Processing document Doc 238
Processing document Doc 239
Processing document Doc 240
Processing document Doc 241
Added document Doc 282
Processing document Doc 242
Added document Doc 283
Processing document Doc 243
```

完成示例应用程序中描述的任务的真实程序可以处理用 Web 服务接收到的文档。

10.4 栈

栈是与队列非常类似的另一个容器，只是要使用不同的方法访问栈。最后添加到栈中的元素会最先读取。栈是一个后进先出(LIFO)的容器。

图 10-2 表示一个栈，用 Push()方法在栈中添加元素，用 Pop()方法获取最近添加的元素。

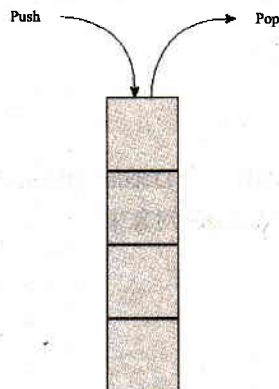


图 10-2

与 `Queue<T>` 类相同, `Stack<T>` 类实现 `IEnumerable<T>` 和 `ICollection` 接口。

`Stack<T>` 类的成员如表 10-3 所示。

表 10-3

Stack<T>类的成员	说 明
Count	返回栈中的元素个数
Push()	在栈顶添加一个元素
Pop()	从栈顶删除一个元素, 并返回该元素。如果栈是空的, 就抛出 <code>InvalidOperationException</code> 异常
Peek()	返回栈顶的元素, 但不删除它
Contains()	确定某个元素是否在栈中, 如果是, 就返回 <code>true</code>

在下面的例子中, 使用 `Push()` 方法把 3 个元素添加到栈中。在 `foreach` 方法中, 使用 `IEnumerable` 接口迭代所有的元素。栈的枚举器不会删除元素, 它只会逐个返回元素。



可从
wrox.com
下载源代码

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

foreach (char item in alphabet)
{
    Console.Write(item);
}

Console.WriteLine();
```

代码段 `StackSample/Program.cs`

因为元素的读取顺序是从最后一个添加到栈中的元素开始到第一个元素, 所以得到的结果如下:

CBA

用枚举器读取元素不会改变元素的状态。使用 `Pop()` 方法会从栈中读取每个元素, 然后删除它们。这样, 就可以使用 `while` 循环迭代集合, 检查 `Count` 属性, 确定栈中是否还有元素:

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');

Console.Write("First iteration: ");
foreach (char item in alphabet)
{
    Console.Write(item);
}
Console.WriteLine();

Console.Write("Second iteration: ");
while (alphabet.Count > 0)
{
    Console.Write(alphabet.Pop());
}
```

```
Console.WriteLine();
```

结果是两个 CBA，每次迭代对应一个 CBA。在第二次迭代后，栈变空，因为第二次迭代使用了 Pop() 方法：

```
First iteration: CBA
Second iteration: CBA
```

10.5 链表

LinkedList<T> 是一个双向链表，其元素指向它前面和后面的元素，如图 10-3 所示。

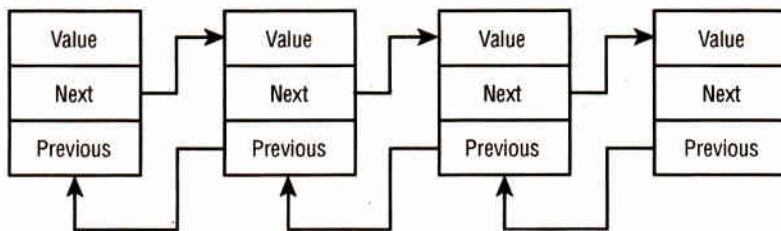


图 10-3

链表的优点是，如果将元素插入列表的中间位置，使用链表就会非常快。在插入一个元素时，只需修改上一个元素的 Next 引用和下一个元素的 Previous 引用，使它们引用所插入的元素。在 List<T> 类中，插入一个元素时，需要移动该元素后面的所有元素。

当然，链表也有缺点。链表的元素只能一个接一个地访问，这需要较长的时间来查找位于链表中间或尾部的元素。

链表不仅能在列表中存储元素。存储元素时，链表还必须存储每个元素存储下一个元素和上一个元素的信息。这就是 LinkedList<T> 包含 LinkedListNode<T> 类型的元素的原因。使用 LinkedListNode<T> 类，可以获得列表中的下一个元素和上一个元素。LinkedListNode<T> 定义了属性 List、Next、Previous 和 Value。List 属性返回与节点相关的 LinkedList<T> 对象，Next 和 Previous 属性用于遍历链表，访问当前节点之后和之前的节点。Value 返回与节点相关的元素，其类型是 T。

LinkedList<T> 类定义的成员可以访问链表中的第一个和最后一个元素 (First 和 Last)、在指定位置插入元素 (AddAfter()、AddBefore()、AddFirst() 和 AddLast() 方法)、删除指定位置的元素 (Remove()、RemoveFirst() 和 RemoveLast() 方法)、从链表的开头 (Find() 方法) 或结尾 (FindLast() 方法) 开始搜索元素。

示例应用程序使用了一个链表和一个列表。链表包含文档，这与上一个队列例子相同，但文档有一个额外的优先级。在链表中，文档按照优先级来排序。如果多个文档的优先级相同，这些元素就按照文档的插入时间来排序。

图 10-4 描述了示例应用程序中的集合。LinkedList<Document> 是一个包含所有 Document 对象的链表，该图显示了文档的标题和优先级。标题指出了文档添加到链表中的时间。第一个添加的文档的标题是 “One”。第二个添加的文档的标题是 “Two”，依此类推。可以看出，文档 One 和 Four 有相同的优先级 8，因为 One 在 Four 之前添加，所以 One 放在链表的前面。

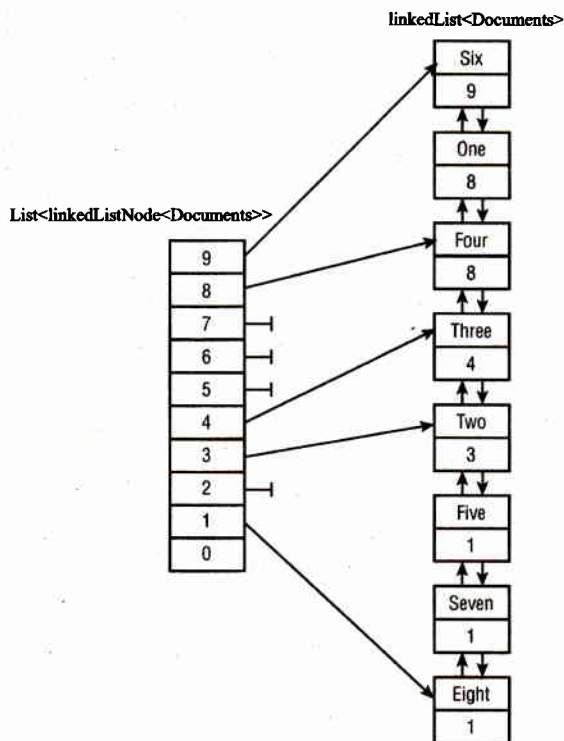


图 10-4

在链表中添加新文档时，它们应放在优先级相同的最后一个文档后面。集合 `LinkedList<Document>` 包含 `LinkedListNode<Document>` 类型的元素。`LinkedListNode<T>` 类添加 `Next` 和 `Previous` 属性，使搜索过程能从一个节点移动到下一个节点上。要引用这类元素，应把 `List<T>` 定义为 `List<LinkedListNode<Document>>`。为了快速访问每个优先级的最后一个文档，集合 `List<LinkedListNode>` 应最多包含 10 个元素，每个元素分别引用每个优先级的最后一个文档。在后面的讨论中，对每个优先级的最后一个文档的引用称为优先级节点。

在上面的例子中，`Document` 类扩展为包含优先级。优先级用类的构造函数设置：



```
public class Document
{
    public string Title { get; private set; }
    public string Content { get; private set; }
    public byte Priority { get; private set; }

    public Document(string title, string content, byte priority = 0)
    {
        this.Title = title;
        this.Content = content;
        this.Priority = priority;
    }
}
```

代码段 `LinkedListSample/Document.cs`

解决方案的核心是 `PriorityDocumentManager` 类。这个类很容易使用。在这个类的公共接口中，

可以把新的 Document 元素添加到链表中，可以检索第一个文档，为了便于测试，它还提供了一个方法，在元素链接到链表中时，该方法可以显示集合中的所有元素。

PriorityDocumentManager 类包含两个集合。LinkedList<Document> 类型的集合包含所有的文档。List<LinkedListNode<Document>> 类型的集合包含最多 10 个元素的引用，它们是添加指定优先级的新文档的入口点。这两个集合变量都用 PriorityDocumentManager 类的构造函数来初始化。列表集合也用 null 初始化：



```
public class PriorityDocumentManager
{
    private readonly LinkedList<Document> documentList;
    // priorities 0.9
    private readonly List<LinkedListNode<Document>> priorityNodes;

    public PriorityDocumentManager()
    {
        documentList = new LinkedList<Document>();

        priorityNodes = new List<LinkedListNode<Document>>(10);
        for (int i = 0; i < 10; i++)
        {
            priorityNodes.Add(new LinkedListNode<Document>(null));
        }
    }
}
```

代码段 LinkedListSample/PriorityDocumentManager.cs

在类的公共接口中，有一个 AddDocument() 方法。AddDocument() 方法只调用私有方法 AddDocumentToPriorityNode()。把实现代码放在另一个方法中的原因是，AddDocumentToPriorityNode() 方法可以递归调用，如后面所示。

```
public void AddDocument(Document d)
{
    if (d == null) throw new ArgumentNullException("d");

    AddDocumentToPriorityNode(d, d.Priority);
}
```

在 AddDocumentToPriorityNode() 方法的实现代码中，第一个操作是检查优先级是否在允许的优先级范围内。这里允许的范围是 0~9。如果传送了错误的值，就会抛出一个 ArgumentException 类型的异常。

接着检查是否已经有一个优先级节点与所传送的优先级相同。如果在列表集合中没有这样的优先级节点，就递归调用 AddDocumentToPriorityNode() 方法，递减优先级值，检查是否有低一级的优先级节点。

如果优先级节点的优先级值与所传送的优先级值不同，也没有比该优先级值更低的优先级节点，就可以调用 AddLast() 方法，将文档安全地添加到链表的末尾。另外，链表节点由负责指定文档优先级的优先级节点引用。

如果存在这样的优先级节点，就可以在链表中找到插入文档的位置。这里必须区分是存在指定优先级值的优先级节点，还是存在以较低的优先级值引用文档的优先级节点。对于第一种情况，可

以把新文档插入由优先级节点引用的位置后面。因为优先级节点总是引用指定优先级值的最后一个文档，所以必须设置优先级节点的引用。如果引用文档的优先级节点有较低的优先级值，情况就会比较复杂。这里新文档必须插入优先级值与优先级节点相同的所有文档的前面。为了找到优先级值相同的第一个文档，要通过一个 while 循环，使用 Previous 属性遍历所有的链表节点，直到找到一个优先级值不同的链表节点为止。这样，就找到了必须插入文档的位置，并可以设置优先级节点。

```
private void AddDocumentToPriorityNode(Document doc, int priority)
{
    if (priority > 9 || priority < 0)
        throw new ArgumentException("Priority must be between 0 and 9");

    if (priorityNodes[priority].Value == null)
    {
        --priority;
        if (priority >= 0)
        {
            // check for the next lower priority
            AddDocumentToPriorityNode(doc, priority);
        }
        else // now no priority node exists with the same priority or lower
            // add the new document to the end
        {
            documentList.AddLast(doc);
            priorityNodes[doc.Priority] = documentList.Last;
        }
        return;
    }
    else // a priority node exists
    {
        LinkedListNode<Document> prioNode = priorityNodes[priority];
        if (priority == doc.Priority)
            // priority node with the same priority exists
        {
            documentList.AddAfter(prioNode, doc);

            // set the priority node to the last document with the same priority
            priorityNodes[doc.Priority] = prioNode.Next;
        }
        else // only priority node with a lower priority exists
        {
            // get the first node of the lower priority
            LinkedListNode<Document> firstPrioNode = prioNode;

            while (firstPrioNode.Previous != null &&
                firstPrioNode.Previous.Value.Priority == prioNode.Value.Priority)
            {
                firstPrioNode = prioNode.Previous;
                prioNode = firstPrioNode;
            }

            documentList.AddBefore(firstPrioNode, doc);

            // set the priority node to the new value
            priorityNodes[doc.Priority] = firstPrioNode.Previous;
        }
    }
}
```

现在还剩下一个简单的方法没有讨论。`DisplayAllNodes()`方法只是在一个 `foreach` 循环中,把每个文档的优先级和标题显示在控制台上。

`GetDocument()`方法从链表中返回第一个文档(优先级最高的文档),并从链表中删除它:

```
public void DisplayAllNodes()
{
    foreach (Document doc in documentList)
    {
        Console.WriteLine("priority: {0}, title {1}", doc.Priority, doc.Title);
    }
}

// returns the document with the highest priority
// (that's first in the linked list)
public Document GetDocument()
{
    Document doc = documentList.First.Value;
    documentList.RemoveFirst();
    return doc;
}
}
```

在 `Main()`方法中, `PriorityDocumentManager` 类用于说明其功能。在链表中添加 8 个优先级不同的新文档,再显示整个链表:



```
static void Main()
{
    PriorityDocumentManager pdm = new PriorityDocumentManager();
    pdm.AddDocument(new Document("one", "Sample", 8));
    pdm.AddDocument(new Document("two", "Sample", 3));
    pdm.AddDocument(new Document("three", "Sample", 4));
    pdm.AddDocument(new Document("four", "Sample", 8));
    pdm.AddDocument(new Document("five", "Sample", 1));
    pdm.AddDocument(new Document("six", "Sample", 9));
    pdm.AddDocument(new Document("seven", "Sample", 1));
    pdm.AddDocument(new Document("eight", "Sample", 1));

    pdm.DisplayAllNodes();
}
}
```

代码段 `LinkedListSample/Program.cs`

在处理好的结果中,文档先按优先级排序,再按添加文档的时间排序:

```
priority: 9, title six
priority: 8, title one
priority: 8, title four
priority: 4, title three
priority: 3, title two
priority: 1, title five
priority: 1, title seven
priority: 1, title eight
```

10.6 有序列表

如果需要基于键对所需集合排序, 就可以使用 `SortedList<TKey, TValue>` 类。这个类按照键给元素排序。这个集合中的值和键都可以使用任意类型。

下面的例子创建了一个有序列表, 其中键和值都是 `string` 类型。默认的构造函数创建了一个空列表, 再用 `Add()` 方法添加两本书。使用重载的构造函数, 可以定义列表的容量, 传递实现了 `IComparer<TKey>` 接口的对象, 该接口用于给列表中的元素排序。

`Add()` 方法的第一个参数是键(书名), 第二个参数是值(ISBN 号)。除了使用 `Add()` 方法之外, 还可以使用索引器将元素添加到列表中。索引器需要把键作为索引参数。如果键已存在, `Add()` 方法就抛出一个 `ArgumentException` 类型的异常。如果索引器使用相同的键, 就用新值替代旧值。



`SortedList<TKey, TValue>` 类只允许每个键有一个对应的值, 如果需要每个键对应多个值, 就可以使用 `Lookup<TKey, TElement>` 类。



可从
wrox.com
下载源代码

```
var books = new SortedList<string, string>();
books.Add("C# 2008 Wrox Box", "978-0-470-047205-7");
books.Add("Professional ASP.NET MVC 1.0", "978-0-470-38461-9");

books["Beginning Visual C# 2008"] = "978-0-470-19135-4";
books["Professional C# 2008"] = "978-0-470-19137-6";
```

代码段 `SortedListSample/Program.cs`

可以使用 `foreach` 语句遍历该列表。枚举器返回的元素是 `KeyValuePair<TKey, TValue>` 类型, 其中包含了键和值。键可以用 `Key` 属性访问, 值可以用 `Value` 属性访问。

```
foreach (KeyValuePair<string, string> book in books)
{
    Console.WriteLine("{0}, {1}", book.Key, book.Value);
}
```

迭代语句会按键的顺序显示书名和 ISBN 号:

```
Beginning Visual C# 2008, 978-0-470-19135-4
C# 2008 Wrox Box, 978-0-470-047205-7
Professional ASP.NET MVC 1.0, 978-0-470-38461-9
Professional C# 2008, 978-0-470-19137-6
```

也可以使用 `Values` 和 `Keys` 属性访问值和键。因为 `Values` 属性返回 `ICollection<TValue>`, `Keys` 属性返回 `ICollection<TKey>`, 所以可以通过 `foreach` 语句使用这些属性:

```
foreach (string isbn in books.Values)
{
    Console.WriteLine(isbn);
}

foreach (string title in books.Keys)
```

```

{
    Console.WriteLine(title);
}

```

第一个循环显示值，第二个循环显示键：

```

978-0-470-19135-4
978-0-470-047205-7
978-0-470-38461-9
978-0-470-19137-6
Beginning Visual C# 2008
C# 2008 Wrox Box
Professional ASP.NET MVC 1.0
Professional C# 2008

```

如果尝试使用索引器访问一个元素，但所传递的键不存在，就会抛出一个 `KeyNotFoundException` 类型的异常。为了避免这个异常，可以使用 `ContainsKey()` 方法，如果所传递的键存在于集合中，这个方法就返回 `true`，也可以调用 `TryGetValue()` 方法，该方法尝试获得指定键的值。如果指定键对应的值不存在，该方法就不会抛出异常。

```

string isbn;
string title = "Professional C# 7.0";
if (!books.TryGetValue(title, out isbn))
{
    Console.WriteLine("{0} not found", title);
}

```

10.7 字典

字典表示一种非常复杂的数据结构，这种数据结构允许按照某个键来访问元素。字典也称为映射或散列表。字典的主要特性是能根据键快速查找值。也可以自由添加和删除元素，这有点像 `List<T>` 类，但没有在内存中移动后续元素的性能开销。

图 10-5 是字典的一个简化表示。其中 `employee-id` (如 B4711) 是添加到字典中的键。键会转换为一个散列。利用散列创建一个数字，它将索引和值关联起来。然后索引包含一个到值的链接。该图做了简化处理，因为一个索引项可以关联多个值，索引可以存储为一个树形结构。

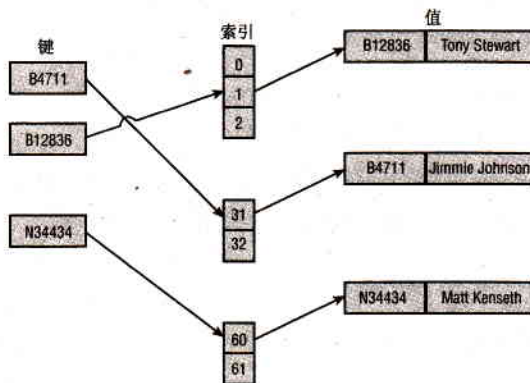


图 10-5

.NET Framework 提供了几个字典类。可以使用的最主要的类是 `Dictionary<TKey, TValue>`。

10.7.1 键的类型

用作字典中键的类型必须重写 `Object` 类的 `GetHashCode()` 方法。只要字典类需要确定元素的位置,它就要调用 `GetHashCode()` 方法。`GetHashCode()` 方法返回的 `int` 由字典用于计算在对应位置放置元素的索引。这里不介绍这个算法。我们只需知道,它涉及素数,所以字典的容量是一个素数。

`GetHashCode()` 方法的实现代码必须满足如下要求:

- 相同的对象应总是返回相同的值。
- 不同的对象可以返回相同的值。
- 它应执行得比较快,计算的开销不大。
- 它不能抛出异常。
- 它应至少使用一个实例字段。
- 散列代码值应平均分布在 `int` 可以存储的整个数字范围上。
- 散列代码最好在对象的生存期中不发生变化。



字典的性能取决于 `GetHashCode()` 方法的实现代码。

为什么要使散列代码值平均分布在整数的取值范围内?如果两个键返回的散列代码值会得到相同的索引,字典类就必须寻找最近的可用空闲位置来存储第二个数据项,这需要进行一定的搜索,以便以后检索这一项。显然这会降低性能,如果在排序的时候许多键都有相同的索引,这类冲突就更可能出现。根据 Microsoft 的部分算法的工作方式,当计算出来的散列代码值平均分布在 `int.MinValue` 和 `int.MaxValue` 之间时,这种风险会降低到最小。

除了实现 `GetHashCode()` 方法之外,键类型还必须实现 `IEquatable<T>.Equals()` 方法,或重写 `Object` 类的 `Equals()` 方法。因为不同的键对象可能返回相同的散列代码,所以字典使用 `Equals()` 方法来比较键。字典检查两个键 A 和 B 是否相等,并调用 `A.Equals(B)` 方法。这表示必须确保下述条件总是成立:



如果 `A.Equals(B)` 方法返回 `true`,则 `A.GetHashCode()` 和 `B.GetHashCode()` 方法必须总是返回相同的散列代码。

这似乎有点奇怪,但它非常重要。如果设计出某种重写这些方法的方式,使上面的条件并不总是成立,把这个类的实例用作键的字典就不能正常工作,而是会发生有趣的事情。例如,把一个对象放在字典中后,就再也检索不到它,或者试图检索某项,却返回了错误的项。



因此,如果为 `Equals()` 方法提供了重写版本,但没有提供 `GetHashCode()` 方法的重写版本,C#编译器就会显示一个编译警告。

对于 `System.Object`,这个条件为 `true`,因为 `Equals()` 方法只是比较引用,`GetHashCode()` 方法实

际上返回一个仅基于对象地址的散列代码。这说明，如果散列表基于一个键，而该键没有重写这些方法，这个散列表就能正常工作。但是，这么做的问题是，只有对象完全相同，键才被认为是相等的。也就是说，把一个对象放在字典中时，必须将它与该键的引用关联起来。也不能在以后用相同的值实例化另一个键对象。如果没有重写 `Equals()` 方法和 `GetHashCode()` 方法，在字典中使用类型时就不太方便。

另外，`System.String` 实现了 `IEquatable` 接口，并重载了 `GetHashCode()` 方法。`Equals()` 方法提供了值的比较，`GetHashCode()` 方法根据字符串的值返回一个散列代码。因此，在字典中把字符串用作键非常方便。

数字类型(如 `Int32`)也实现 `IEquatable` 接口，并重载 `GetHashCode()` 方法。但是这些类型返回的哈希代码只映射到值上。如果希望用作键的数字本身没有分布在可能的整数值范围内，把整数用作键就不能满足键值的平均分布规则，于是不能获得最佳的性能。`Int32` 并不适合在字典中使用。

如果需要使用的键类型没有实现 `IEquatable` 接口，并根据存储在字典中的键值重载 `GetHashCode()` 方法，就可以创建一个实现 `IEqualityComparer<T>` 接口的比较器。`IEqualityComparer<T>` 接口定义了 `GetHashCode()` 和 `Equals()` 方法，并将传递的对象作为参数，因此可以提供与对象类型不同的实现方式。`Dictionary<TKey, TValue>` 构造函数的一个重载版本允许传递一个实现了 `IEqualityComparer<T>` 接口的对象。如果把这个对象赋予字典，该类就用于生成散列代码并比较键。

10.7.2 字典示例

字典示例程序建立了一个员工字典。该字典用 `EmployeeId` 对象来索引，存储在字典中的每个数据项都是一个 `Employee` 对象，该对象存储员工的详细数据。

实现 `EmployeeId` 结构是为了定义了字典中使用的键，该结构的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。在构造函数中填充字段。重载 `ToString()` 方法是为了获得员工 ID 的字符串表示。与键类型的要求一样，`EmployeeId` 结构也要实现 `IEquatable` 接口，并重载 `GetHashCode()` 方法。



可从
wrox.com
下载源代码

```
[Serializable]
public class EmployeeIdException : Exception
{
    public EmployeeIdException(string message) : base(message) { }
}

[Serializable]
public struct EmployeeId : IEquatable<EmployeeId>
{
    private readonly char prefix;
    private readonly int number;

    public EmployeeId(string id)
    {
        if (id == null) throw new ArgumentNullException("id");

        prefix = (id.ToUpper())[0];
        int numLength = id.Length - 1;
        try
        {
            number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
        }
    }
}
```

```

    }
    catch (FormatException)
    {
        throw new EmployeeIdException("Invalid EmployeeId format");
    }
}

public override string ToString()
{
    return prefix.ToString() + string.Format("{0,6:000000}", number);
}

public override int GetHashCode()
{
    return (number ^ number << 16) * 0x15051505;
}

public bool Equals(EmployeeId other)
{
    if (other == null) return false;

    return (prefix == other.prefix && number == other.number);
}

public override bool Equals(object obj)
{
    return Equals((EmployeeId)obj);
}

public static bool operator ==(EmployeeId left, EmployeeId right)
{
    return left.Equals(right);
}

public static bool operator !=(EmployeeId left, EmployeeId right)
{
    return !(left == right);
}
}

```

代码段 DictionarySample/EmployeeId.cs

由 `IEquatable<T>` 接口定义的 `Equals()` 方法比较两个 `EmployeeId` 对象的值，如果这两个值相同，它就返回 `true`。除了实现 `IEquatable<T>` 接口中的 `Equals()` 方法之外，还可以重写 `Object` 类中的 `Equals()` 方法。

```

public bool Equals(EmployeeId other)
{
    if (other == null) return false;
    return (prefix == other.prefix && number == other.number);
}

```

由于数字是可变的，因此员工可以取 1~190000 的一个值。这并没有填满整数取值范围。`GetHashCode()` 方法使用的算法将数字向左移动 16 位，再与原来的数字进行异或操作，最后将结果乘以十六进制数 15 051 505。散列代码在整数取值区域上的分布相当均匀：

```

public override int GetHashCode()

```

```
{
    return (number ^ number << 16) * 0x15051505;
}
```



在 Internet 上,有许多更复杂的算法,它们能使散列代码在整数取值范围上更好地分布。也可以使用字符串的 GetHashCode()方法来返回一个散列。

Employee 类是一个简单的实体类,该实体类包含员工的姓名、薪水和 ID。构造函数初始化所有值,ToString()方法返回一个实例的字符串表示。ToString()方法的实现代码使用格式化字符串创建字符串表示,以提高性能。



可从
wrox.com
下载源代码

```
[Serializable]
public class Employee
{
    private string name;
    private decimal salary;
    private readonly EmployeeId id;

    public Employee(EmployeeId id, string name, decimal salary)
    {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public override string ToString()
    {
        return String.Format("{0}: {1, -20} {2:C}",
            id.ToString(), name, salary);
    }
}
```

代码段 DictionarySample/Employee.cs

在示例程序的 Main()方法中,创建一个新的 Dictionary<TKey, TValue>实例,其中键是 EmployeeId 类型,值是 Employee 类型。构造函数指定了 31 个元素的容量。注意容量一般是素数。但如果指定了一个不是素数的值,也不需要担心。Dictionary<TKey, TValue>类会使用传递给构造函数的整数后面紧接着的一个素数,来指定容量。用 Add()方法创建员工对象和 ID,并添加到字典中。除了 Add()方法外,还可以使用索引器,将键和值添加到字典中,如员工 Dale 和 Jeff 所示:



可从
wrox.com
下载源代码

```
static void Main()
{
    var employees = new Dictionary<EmployeeId, Employee>(31);

    var idKyle = new EmployeeId("T3755");
    var kyle = new Employee(idKyle, "Kyle Bush", 5443890.00m);
    employees.Add(idKyle, kyle);
    Console.WriteLine(kyle);

    var idCarl = new EmployeeId("F3547");
    var carl = new Employee(idCarl, "Carl Edwards", 5597120.00m);
    employees.Add(idCarl, carl);
    Console.WriteLine(carl);
}
```

```

var idJimmie = new EmployeeId("C3386");
var jimmie = new Employee(idJimmie, "Jimmie Johnson", 5024710.00m);
employees.Add(idJimmie, jimmie);
Console.WriteLine(jimmie);

var idDale = new EmployeeId("C3323");
var dale = new Employee(idDale, "Dale Earnhardt Jr.", 3522740.00m);
employees[idDale] = dale;
Console.WriteLine(dale);

var idJeff = new EmployeeId("C3234");
var jeff = new Employee(idJeff, "Jeff Burton", 3879540.00m);
employees[idJeff] = jeff;
Console.WriteLine(jeff);

```

代码段 DictionarySample/Program.cs

将数据项添加到字典中后, 在 `while` 循环中读取字典中的员工。让用户输入一个员工号, 把该号码存储在变量 `userInput` 中。用户输入 `X` 即可退出应用程序。如果输入的键在字典中, 就使用 `Dictionary<TKey, TValue>` 类的 `TryGetValue()` 方法检查它。如果找到了该键, `TryGetValue()` 方法就返回 `true`; 否则返回 `false`。如果找到了与键关联的值, 该值就存储在 `employee` 变量中, 并把该值写入控制台。



也可以使用 `Dictionary<TKey, TValue>` 类的索引器替代 `TryGetValue()` 方法, 来访问存储在字典中的值。但是, 如果没有找到键, 索引器会抛出一个 `KeyNotFoundException` 类型的异常。

```

while (true)
{
    Console.WriteLine("Enter employee id (X to exit) > ");
    var userInput = Console.ReadLine();
    userInput = userInput.ToUpper();
    if (userInput == "X") break;

    EmployeeId id;
    try
    {
        id = new EmployeeId(userInput);

        Employee employee;
        if (!employees.TryGetValue(id, out employee))
        {
            Console.WriteLine("Employee with id {0} does not exist",
                               id);
        }
        else
        {
            Console.WriteLine(employee);
        }
    }
    catch (EmployeeIdException ex)
    {
    }
}

```

```

        Console.WriteLine(ex.Message);
    }
}

```

运行应用程序，得到如下输出：

```

Enter employee id (X to exit) > C3386
C003386: Jimmie Johnson ? 5.024.710,00
Enter employee id (X to exit) > F3547
F003547: Carl Edwards ? 5.597.120,00
Enter employee id (X to exit) > X
Press any key to continue ...

```

10.7.3 Lookup 类

`Dictionary<TKey, TValue>` 类支持每个键关联一个值。`Lookup<TKey, TElement>` 类非常类似于 `Dictionary<TKey, TValue>` 类，但把键映射到一个值集上。这个类在程序集 `System.Core` 中实现，用 `System.Linq` 名称空间定义。

`Lookup<TKey, TElement>` 类不能像一般的字典那样创建，而必须调用 `ToLookup()` 方法，该方法返回一个 `Lookup<TKey, TElement>` 对象。`ToLookup()` 方法是一个扩展方法，它可以用于实现了 `IEnumerable<T>` 接口的所有类。在下面的例子中，填充了一个 `Racer` 对象列表。因为 `List<T>` 类实现了 `IEnumerable<T>` 接口，所以可以在赛车列表上调用 `ToLookup()` 方法。这个方法需要一个 `Func<TSource, TKey>` 类型的委托，`Func<TSource, TKey>` 类型定义了键的选择器。这里使用 Lambda 表达式 `r => r.Country`，根据国家来选择赛车手。`foreach` 循环只使用索引器访问来自澳大利亚的赛车手。

扩展方法详见第 11 章，Lambda 表达式参见第 8 章。



可从
wrox.com
下载源代码

```

var racers = new List<Racer>();
racers.Add(new Racer("Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer("Alan", "Jones", "Australia", 12));
racers.Add(new Racer("Jackie", "Stewart", "United Kingdom", 27));
racers.Add(new Racer("James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham", "Australia", 14));

var lookupRacers = racers.ToLookup(r => r.Country);

foreach (Racer r in lookupRacers["Australia"])
{
    Console.WriteLine(r);
}

```

代码段 LookupSample/Program.cs

结果显示了来自澳大利亚的赛车手：

```

Alan Jones
Jack Brabham

```

10.7.4 有序字典

`SortedDictionary<TKey, TValue>`类是一个二叉搜索树，其中的元素根据键来排序。该键类型必须实现 `Comparable<TKey>` 接口。如果键的类型不能排序，则还可以创建一个实现了 `IComparer<TKey>` 接口的比较器，将比较器用作有序字典的构造函数的一个参数。

`SortedDictionary<TKey, TValue>`类和 `SortedList<TKey, TValue>`类的功能类似。但因为 `SortedList<TKey, TValue>`实现为一个基于数组的列表，而 `SortedDictionary<TKey, TValue>`类实现为一个字典，所以它们有不同的特征。

- `SortedList<TKey, TValue>`类使用的内存比 `SortedDictionary<TKey, TValue>`类少。
- `SortedDictionary<TKey, TValue>`类的元素插入和删除速度比较快。
- 在用已排好序的数据填充集合时，若不需要修改容量，`SortedList<TKey, TValue>`类就比较快。



`SortedList` 类使用的内存比 `SortedDictionary` 类少，但 `SortedDictionary` 类在插入和删除未排序的数据时比较快。

10.8 集

包含不重复元素的集合称为“集(set)”。.NET 4 包含两个集(`HashSet<T>`和 `SortedSet<T>`)，它们都实现 `ISet<T>`接口。`HashSet<T>`集包含不重复元素的无序列表，`SortedSet<T>`集包含不重复元素的有序列表。

`ISet<T>`接口提供的方法可以创建合集、交集，或者给出一个集是另一个集的超集或子集的信息。

在示例代码中，创建了 3 个字符串类型的新集，并用一级方程式汽车填充它们。`HashSet<T>`集实现 `ICollection<T>`接口。但是在该类中，`Add()`方法是显式实现的，还提供了另一个 `Add()`方法。`Add()`方法的区别是返回类型，它返回一个布尔值，说明是否添加了元素。如果该元素已经在集中，就不添加它，并返回 `false`。



可从
wrox.com
下载源代码

```
var companyTeams = new HashSet<string>()
    { "Ferrari", "McLaren", "Toyota", "BMW", "Renault" };
var traditionalTeams = new HashSet<string>()
    { "Ferrari", "McLaren" };
var privateTeams = new HashSet<string>()
    { "Red Bull", "Toro Rosso", "Force India", "Brawn GP" };

if (privateTeams.Add("Williams"))
    Console.WriteLine("Williams added");
if (!companyTeams.Add("McLaren"))
    Console.WriteLine("McLaren was already in this set");
```

代码段 SetSample/Program.cs

两个 `Add()`方法的输出写到控制台上：

```
Williams added
McLaren was already in this set
```

`IsSubsetOf()`和 `IsSupersetOf()`方法比较集并实现了 `IEnumerable<T>`接口的集合,并返回一个布尔结果。这里, `IsSubsetOf()`方法验证 `traditionalTeams` 集合中的每个元素是否都包含在 `companyTeams` 集合方法中, `IsSupersetOf()`方法验证 `traditionalTeams` 集合是否有与 `companyTeams` 集合比较的额外元素。

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
    Console.WriteLine("traditionalTeams is subset of companyTeams");
}

if (companyTeams.IsSupersetOf(traditionalTeams))
{
    Console.WriteLine("companyTeams is a superset of traditionalTeams");
}
```

这个验证的结果如下:

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
```

`Williams` 也是一个传统队,因此这个队添加到 `traditionalTeams` 集合中:

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
    Console.WriteLine("At least one team is the same with the " +
        "traditional and private teams");
}
```

因为有一个重叠,所以结果如下:

```
At least one team is the same with the traditional and private teams.
```

调用 `UnionWith()`方法,把引用新 `SortedSet<string>`的变量 `allTeams` 填充为 `companyTeams`、`PrivateTeams` 和 `traditionalTeams` 的合集:

```
var allTeams = new SortedSet<string>(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);

Console.WriteLine();
Console.WriteLine("all teams");
foreach (var team in allTeams)
{
    Console.WriteLine(team);
}
```

这里返回所有队,但每个队都只列出一次,因为集只包含唯一值。因为容器是 `SortedSet<string>`,所以结果是有序的:

```
BMW
Brawn GP
Ferrari
Force India
McLaren
```

```

Red Bull
Renault
Toro Rosso
Toyota
Williams

```

ExceptWith()方法从 allTeams 集中删除所有私有队:

```

allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("no private team left");
foreach (var team in allTeams)
{
    Console.WriteLine(team);
}

```

集合中的其他元素不包含私有队:

```

BMW
Ferrari
McLaren
Renault
Toyota

```

10.9 可观察的集合

如果需要集合中的元素何时删除或添加的信息,就可以使用 `ObservableCollection<T>` 类。这个类是为 WPF 定义的,这样 UI 就可以得知集合的变化,因此这个类在程序集 `WindowsBase` 中定义,同时用户需要引用这个程序集。这个类的名称空间是 `System.Collections.ObjectModel`。

`ObservableCollection<T>` 类派生自 `Collection<T>` 基类,该基类可用于创建自定义集合,并在内部使用 `List<T>` 类。重写基类中的虚方法 `SetItem()` 和 `RemoveItem()`,以触发 `CollectionChanged` 事件。这个类的用户可以使用 `INotifyCollectionChanged` 接口注册这个事件。

下面的示例说明了 `ObservableCollection<string>()` 方法的用法,其中给 `CollectionChanged` 事件注册了 `Data_CollectionChanged()` 方法。把两项添加到末尾,再插入一项,并删除一项:



可从
wrox.com
下载源代码

```

var data = new ObservableCollection<string>();
data.CollectionChanged += Data_CollectionChanged;
data.Add("One");
data.Add("Two");
data.Insert(1, "Three");
data.Remove("One");

```

代码段 `ObservableCollectionSample/Program.cs`

`Data_CollectionChanged()` 方法接收 `NotifyCollectionChangedEventArgs`,其中包含了集合的变化信息。`Action` 属性给出了是否添加或删除一项的信息。对于删除的项,会设置 `OldItems` 属性,列出删除的项。对于添加的项,则设置 `NewItems` 属性,列出新增的项。

```

static void Data_CollectionChanged(object sender,
    NotifyCollectionChangedEventArgs e)

```

```

{
    Console.WriteLine("action: {0}", e.Action.ToString());

    if (e.OldItems != null)
    {
        Console.WriteLine("starting index for old item(s): {0}",
            e.OldStartingIndex);
        Console.WriteLine("old item(s):");
        foreach (var item in e.OldItems)
        {
            Console.WriteLine(item);
        }
    }
    if (e.NewItems != null)
    {
        Console.WriteLine("starting index for new item(s): {0}",
            e.NewStartingIndex);
        Console.WriteLine("new item(s): ");
        foreach (var item in e.NewItems)
        {
            Console.WriteLine(item);
        }
    }
    Console.WriteLine();
}

```

运行应用程序，输出如下所示。先在集合中添加 **One** 和 **Two** 项，因此显示的 **Add** 动作的索引是 0 和 1。第 3 项 **Three** 插入在位置 1 上，所以显示的 **Add** 动作的索引是 1。最后删除 **One** 项，显示的 **Remove** 动作的索引是 0：

```

action: Add
starting index for new item(s): 0
new item(s):
One

action: Add
starting index for new item(s): 1
new item(s):
Two

action: Add
starting index for new item(s): 1
new item(s):
Three

action: Remove
starting index for old item(s): 0
old item(s):
One

```

10.10 位数组

如果需要处理的数字有许多位，就可以使用 **BitArray** 类和 **BitVector32** 结构。**BitArray** 类位于名

称空间 `System.Collections` 中, `BitVector32` 结构位于名称空间 `System.Collections.Specialized` 中。这两种类型最重要的区别是, `BitArray` 类可以重新设置大小, 如果事先不知道需要的位数, 就可以使用 `BitArray` 类, 它可以包含非常多的位。`BitVector32` 结构是基于栈的, 因此比较快。`BitVector32` 结构仅包含 32 位, 它们存储在一个整数中。

10.10.1 BitArray 类

`BitArray` 类是一个引用类型, 它包含一个 `int` 数组, 其中每 32 位使用一个新整数。这个类的成员如表 10-4 所示。

表 10-4

BitArray 类的成员	说 明
Count、Length	Count 和 Length 属性的 get 访问器返回数组中的位数。使用 Length 属性还可以定义新的数组大小, 重新设置集合的大小
Item、Get()、Set()	可以使用索引器读写数组中的位。索引器是布尔类型。除了使用索引器之外, 还可以使用 Get() 和 Set() 方法访问数组中的位
SetAll()	根据传送给该方法的参数, SetAll() 方法设置所有位的值
Not()	Not() 方法对数组中所有位的值取反
And()、Or()、Xor()	使用 And()、Or() 和 Xor() 方法, 可以合并两个 BitArray 对象。And() 方法执行二元 AND, 只有两个输入数组的位都设置为 1, 结果位才是 1。Or() 方法执行二元 OR, 只要有一个输入数组的位设置为 1, 结果位就是 1。Xor() 方法是异或操作, 只有一个输入数组的位设置为 1, 结果位才是 1

辅助方法 `DisplayBits()` 遍历 `BitArray`, 根据位的设置情况, 在控制台上显示 1 或 0:



可从
wrox.com
下载源代码

```
static void DisplayBits(BitArray bits)
{
    foreach (bool bit in bits)
    {
        Console.Write(bit ? 1 : 0);
    }
}
```

代码段 BitArraySample/Program.cs

说明 `BitArray` 类的示例创建了一个包含 8 位的数组, 其索引是 0~7。`SetAll()` 方法把这 8 位都设置为 `true`。接着 `Set()` 方法把对应于 1 的位设置为 `false`。除了 `Set()` 方法之外, 还可以使用索引器, 例如下面的第 5 个和第 7 个索引:

```
var bits1 = new BitArray(8);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.WriteLine("initialized: ");
DisplayBits(bits1);
```

```
Console.WriteLine();
```

这是初始化位的显示结果:

```
initialized: 10111010
```

Not()方法会对 BitArray 类的位取反:

```
Console.Write(" not ");
DisplayBits(bits1);
bits1.Not();
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

Not()方法的结果是对所有的位取反。如果某位是 true, 执行 Not()方法的结果就是 false, 反之亦然。

```
not 10111010 = 01000101
```

这里创建了一个新的 BitArray 类。在构造函数中, 因为使用变量 bits1 初始化数组, 所以新数组与旧数组有相同的值。接着把第 0、1 和 4 位的值设置为不同的值。在使用 Or()方法之前, 显示位数组 bits1 和 bits2。Or()方法将改变 bits1 的值:

```
var bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
DisplayBits(bits1);
Console.Write(" or ");
DisplayBits(bits2);
Console.Write(" = ");
bits1.Or(bits2);
DisplayBits(bits1);
Console.WriteLine();
```

使用 Or()方法时, 从两个输入数组中提取设置位。结果是, 如果某位在第一个或第二个数组中设置为 true, 该位在执行 Or()方法后就是 true:

```
01000101 or 10001101 = 11001101
```

下面使用 And()方法作用于位数组 bits1 和 bits2:

```
DisplayBits(bits2);
Console.Write(" and ");
DisplayBits(bits1);
Console.Write(" = ");
bits2.And(bits1);
DisplayBits(bits2);
Console.WriteLine();
```

And()方法只把在两个输入数组中都设置为 true 的位设置为 true:

```
10001101 and 11001101 = 10001101
```

最后使用 Xor()方法进行异或操作:

```

DisplayBits(bits1);
Console.Write(" xor ");
DisplayBits(bits2);
bits1.Xor(bits2);
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();

```

使用 `Xor()` 方法, 只有一个(不能是两个)输入数组的位设置为 1, 结果位才是 1。

```
11001101 xor 10001101 = 01000000
```

10.10.2 BitVector32 结构

如果事先知道需要的位数, 就可以使用 `BitVector32` 结构替代 `BitArray` 类。`BitVector32` 结构效率较高, 因为它是一个值类型, 在整数栈上存储位。一个整数可以存储 32 位。如果需要更多的位, 就可以使用多个 `BitVector32` 值或 `BitArray` 类。`BitArray` 类可以根据需要增大, 但 `BitVector32` 结构不能。

表 10-5 列出了 `BitVector32` 结构中 with `BitArray` 类完全不同的成员。

表 10-5

BitVector32 结构的成员	说 明
<code>Data</code>	<code>Data</code> 属性把 <code>BitVector32</code> 结构中的数据返回为整数
<code>Item</code>	<code>BitVector32</code> 的值可以使用索引器设置。索引器是重载的——可以使用掩码或 <code>BitVector32.Section</code> 类型的片断来获取和设置值
<code>CreateMask()</code>	这是一个静态方法, 用于为访问 <code>BitVector32</code> 结构中的特定位创建掩码
<code>CreateSection()</code>	这是一个静态方法, 用于创建 32 位中的几个片断

示例代码用默认构造函数创建了一个 `BitVector32` 结构, 其中所有的 32 位都初始化为 `false`。接着创建掩码, 以访问位矢量中的位。对 `CreateMask()` 方法的第一个调用创建了用来访问第一位的一个掩码。接着调用 `CreateMask()` 方法, 将 `bit1` 设置为 1。再次调用 `CreateMask()` 方法, 把第一个掩码作为参数传递给 `CreateMask()` 方法, 返回用来访问第二位(它是 2)的一个掩码。接着, 将 `bit3` 设置为 4, 以访问位编号 3。`bit4` 的值是 8, 以访问位编号 4。

然后, 使用掩码和索引器访问位矢量中的位, 并相应地设置字段:



可从
wrox.com
下载源代码

```

var bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);
bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);

```

代码段 BitArraySample/Program.cs

`BitVector32` 结构有一个重写的 `ToString()` 方法, 它不仅显示类名, 还显示 1 或 0, 来说明位是否

设置了，如下所示：

```
BitVector32{0000000000000000000000000000000011101}
```

除了用 `CreateMask()` 方法创建掩码之外，还可以自己定义掩码，也可以一次设置多位。十六进制值 `abcdef` 与二进制值 `1010 1011 1100 1101 1110 1111` 相同。用这个值定义的所有位都设置了：

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

在输出中可以验证设置的位:

```
BitVector32{00000000101010111100110111101111}
```

把 32 位分别放在不同的片断中非常有用。例如，IPv4 地址定义为一个 4 字节的数，该数存储在一个整数中。可以定义 4 个片断，把这个整数拆分开。在多播 IP 消息中，使用了几个 32 位的值。其中一个 32 位的值放在这些片断中：16 位表示源号，8 位表示查询器的查询内部码，3 位表示查询器的健壮变量，1 位表示抑制标志，还有 4 个保留位。也可以定义自己的位含义，以节省内存。

下面的例子模拟接收到值 0x79abcdef，把这个值传送给 BitVector32 结构的构造函数，从而相应地设置位：

```
int received = 0x79abcdef;

BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

在控制台上显示了初始化的位:

```
BitVector32{01111001101010111100110111101111}
```

接着创建 6 个片断。第一个片断需要 12 位，由十六进制值 0xffff 定义(设置了 12 位)。片断 B 需要 8 位，片断 C 需要 4 位，片断 D 和 E 需要 3 位，片断 F 需要两位。第一次调用 CreateSection() 方法只是接收 0xffff，为最前面的 12 位分配内存。第二次调用 CreateSection() 方法时，将第一个片断作为参数传递，从而使下一个片断从第一个片断的结尾处开始。CreateSection() 方法返回一个 BitVector32.Section 类型的值，该类型包含了该片断的偏移量和掩码。

```
// sections: FF EEE DDD CCCC BBBBBBBB
// AAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

把一个 `BitVector32.Section` 类型的值传递给 `BitVector32` 结构的索引器，会返回一个 `int`，它映射到位矢量的片断上。这里使用一个帮助方法 `IntToBinaryString()`，获得该 `int` 数的字符串表示：

```
Console.WriteLine("Section A: {0}",
    IntToBinaryString(bits2[sectionA], true));
Console.WriteLine("Section B: {0}",
    IntToBinaryString(bits2[sectionB], true));
Console.WriteLine("Section C: {0}",
```

```

        IntToBinaryString(bits2[sectionC], true));
Console.WriteLine("Section D: {0}",
    IntToBinaryString(bits2[sectionD], true));
Console.WriteLine("Section E: {0}",
    IntToBinaryString(bits2[sectionE], true));
Console.WriteLine("Section F: {0}",
    IntToBinaryString(bits2[sectionF], true));

```

IntToBinaryString()方法接收整数中的位，并返回一个包含0和1的字符串表示。在实现代码中遍历整数的32位。在迭代过程中，如果该位设置为1，就在 **StringBuilder** 的后面追加1，否则，就追加0。在循环中，移动一位，以检查是否设置了下一位。

```

static string IntToBinaryString(int bits, bool removeTrailingZero)
{
    var sb = new StringBuilder(32);
    for (int i = 0; i < 32; i++)
    {
        if ((bits & 0x80000000) != 0)
        {
            sb.Append("1");
        }
        else
        {
            sb.Append("0");
        }
        bits = bits << 1;
    }
    string s = sb.ToString();
    if (removeTrailingZero)
    {
        return s.TrimStart('0');
    }
    else
    {
        return s;
    }
}

```

结果显示了片断 A~F 的位表示，现在可以用传递给位矢量的值来验证：

```

Section A: 110111101111
Section B: 10111100
Section C: 1010
Section D: 1
Section E: 111
Section F: 1

```

10.11 并发集合

.NET 4 包含的新名称空间 **System.Collections.Concurrent** 有几个线程安全的集合类。线程安全的集合可防止多个线程以相互冲突的方式访问集合。

为了对集合进行线程安全的访问，定义了 `IProducerConsumerCollection<T>` 接口。这个接口中最重要的是 `TryAdd()` 和 `TryTake()`。`TryAdd()` 方法尝试给集合添加一项，但如果集合禁止添加项，这个操作就可能失败。为了给出相关信息，`TryAdd()` 方法返回一个布尔值，以说明操作是成功还是失败。`TryTake()` 方法也以这种方式工作，以通知调用者操作是成功还是失败，并在操作成功时返回集合中的项。下面列出了 `System.Collections.Concurrent` 名称空间中的类及其功能：

- `ConcurrentQueue<T>` —— 这个集合类用一种免锁定的算法实现，使用在内部合并到一个链表中的 32 项数组。访问队列元素的方法有 `Enqueue()`、`TryDequeue()` 和 `TryPeek()`。这些方法的命名非常类似于前面 `Queue<T>` 类的方法，只是给可能调用失败的方法加上了前缀 `Try`。因为这个类实现了 `IProducerConsumerCollection<T>` 接口，所以 `TryAdd()` 和 `TryTake()` 方法仅调用 `Enqueue()` 和 `TryDequeue()` 方法。
- `ConcurrentStack<T>` —— 非常类似于 `ConcurrentQueue<T>` 类，只是带有另外的元素访问方法。`ConcurrentStack<T>` 类定义了 `Push()`、`PushRange()`、`TryPeek()`、`TryPop()` 和 `TryPopRange()` 方法。在内部这个类使用其元素的链表。
- `ConcurrentBag<T>` —— 该类没有定义添加或提取项的任何顺序。这个类使用一个把线程映射到内部使用的数组上的概念，因此尝试减少锁定。访问元素的方法有 `Add()`、`TryPeek()` 和 `TryTake()`。
- `ConcurrentDictionary<TKey, TValue>` —— 这是一个线程安全的键值集合。`TryAdd()`、`TryGetValue()`、`TryRemove()` 和 `TryUpdate()` 方法以非阻塞的方式访问成员。因为元素基于键和值，所以 `ConcurrentDictionary<TKey, TValue>` 没有实现 `IProducerConsumerCollection<T>`。
- `ConcurrentXXX` —— 这些集合是线程安全的，如果某个动作不适用于线程的当前状态，它们就返回 `false`。在继续之前，总是需要确认添加或提取元素是否成功。不能相信集合会完成任务。
- `BlockingCollection<T>` —— 这个集合在可以添加或提取元素之前，会阻塞线程并一直等待。`BlockingCollection<T>` 集合提供了一个接口，以使用 `Add()` 和 `Take()` 方法来添加和删除元素。这些方法会阻塞线程，一直等到任务可以执行为止。

`Add()` 方法有一个重载版本，其中可以给该重载版本传递一个 `CancellationToken` 令牌。这个令牌允许取消被阻塞的调用。

如果不希望线程无限期地等待下去，且不希望从外部取消调用，就可以使用 `TryAdd()` 和 `TryTake()` 方法，在这些方法中，也可以指定一个超时值，它表示在调用失败之前应阻塞线程和等待的最长时间。

- `BlockingCollection<T>` —— 这是对实现了 `IProducerConsumerCollection<T>` 接口的任意类的修饰器，它默认使用 `ConcurrentQueue<T>` 类。还可以给构造函数传递任何其他实现了 `IProducerConsumerCollection<T>` 接口的类。

下面的代码示例简单演示了使用 `BlockingCollection<T>` 类和多个线程的过程。一个线程是生成器，它使用 `Add()` 方法给集合写入元素，另一个线程是使用者，它使用 `Take()` 方法从集合中提取元素：



可从
wtox.com
下载源代码

```
var sharedCollection = new BlockingCollection<int>();
var events = new ManualResetEventSlim[2];
var waits = new WaitHandle[2];
for (int i = 0; i < 2; i++)
{
```

```

        events[i] = new ManualResetEventSlim(false);
        waits[i] = events[i].WaitHandle;
    }

    var producer = new Thread(obj =>
    {
        var state =
            (Tuple<BlockingCollection<int>, ManualResetEventSlim>)obj;
        var coll = state.Item1;
        var ev = state.Item2;
        var r = new Random();

        for (int i = 0; i < 300; i++)
        {
            coll.Add(r.Next(3000));
        }
        ev.Set();
    });
    producer.Start(
        Tuple.Create<BlockingCollection<int>, ManualResetEventSlim>(
            sharedCollection, events[0]));

    var consumer = new Thread(obj =>
    {
        var state =
            (Tuple<BlockingCollection<int>, ManualResetEventSlim>)obj;
        var coll = state.Item1;
        var ev = state.Item2;

        for (int i = 0; i < 300; i++)
        {
            int result = coll.Take();
        }
        ev.Set();
    });
    consumer.Start(
        Tuple.Create<BlockingCollection<int>, ManualResetEventSlim>(
            sharedCollection, events[1]));

    if (!WaitHandle.WaitAll(waits))
        Console.WriteLine("wait failed");
    else
        Console.WriteLine("reading/writing finished");

```

代码段 ConcurrentSample/Program.cs



一旦可以使用多个线程，使用并发集合就非常有趣。它与 `CancellationToken` 令牌
的用法参见第20章。下一章也介绍了如何使用并发集合和并行 LINQ。

10.12 性能

许多集合类都提供了相同的功能，例如，`SortedList` 类与 `SortedDictionary` 类的功能几乎完全相

同。但是，其性能常常有很大区别。一个集合使用的内存少，另一个集合的元素检索速度快。在 MSDN 文档中，集合的方法常常有性能提示，给出了以大 O 记号表示的操作时间：

```
O(1)
O(log n)
O(n)
```

O(1)表示无论集合中有多少数据项，这个操作需要的时间都不变。例如，ArrayList 类的 Add() 方法就具有 O(1)行为。无论列表中有多个元素，在列表末尾添加一个新元素的时间都相同。Count 属性会给出元素个数，所以很容易找到列表末尾。

O(n)表示对于集合中的每个元素，需要增加的时间量都相同。如果需要重新给集合分配内存，ArrayList 类的 Add()方法就是一个 O(n)操作。改变容量，需要复制列表，复制的时间随元素的增加而线性增加。

O(log n)表示操作需要的时间随集合中元素的增加而增加，但每个元素需要增加的时间不是线性的，而是呈对数曲线。在集合中执行插入操作时，SortedDictionary<TKey,TValue>集合类具有 O(log n) 行为，而 SortedList<TKey,TValue>集合类具有 O(n)行为。这里 SortedDictionary <TKey,TValue>集合类要快得多，因为它在树形结构中插入元素的效率比列表高得多。

表 10-7 列出了集合类及其执行不同操作的性能，例如添加、插入和删除元素。使用这个表可以选择性能最佳的集合类。左列是集合类，Add 列给出了在集合中添加元素所需的时间。List<T>和 HashSet<T>类把 Add 方法定义为在集合中添加元素。其他集合类用不同的方法把元素添加到集合中。例如 Stack<T>类定义了 Push()方法，Queue<T>类定义了 Enqueue()方法。这些信息也列在表中。

如果单元格中有多个大 O 值，表示若集合需要重置大小，该操作就需要一定的时间。例如，在 List<T>类中，添加元素的时间是 O(1)。如果集合的容量不够大，需要重置大小，重置大小需要的时间长度就是 O(n)。集合越大，重置大小操作的时间就越长。最好避免重置集合的大小，而应把集合的容量设置为一个可以包含所有元素的值。

如果单元格的内容是 na(not applicatable)，就表示这个操作不能应用于这种集合类型。

表 10-16

集 合	Add	Insert	Remove	Item	Sort	Find
List<T>	如果集合必须重置大小，就是 O(1)或 O(n)	O(n)	O(n)	O(1)	O(n log n), 最坏的情况是 O(n ^ 2)	O(n)
Stack<T>	Push(), 如果栈必须重置大小，就是 O(1)或 O(n)	na	Pop()、O(1)	na	na	na
Queue<T>	Enqueue(), 如果队列必须重置大小，就是 O(1)或 O(n)	na	Dequeue()、O(1)	na	na	na
HashSet<T>	如果集必须重置大小，就是 O(1)或 O(n)	Add() O(1)或 O(n)	O(1)	na	na	na
LinkedList<T>	AddLast()、O(1)	AddAfter() O(1)	O(1)	na	na	O(n)
Dictionary <TKey, TValue>	O(1)或 O(n)	na	O(1)	O(1)	na	na

(续表)

集 合	Add	Insert	Remove	Item	Sort	Find
SortedDictionary <TKey, TValue>	$O(\log n)$	na	$O(\log n)$	$O(\log n)$	na	na
SortedList <TKey, TValue>	无序数据为 $O(n)$; 如果必须重置大小就是 $O(n)$; 到列表的尾部就是 $O(\log n)$	na	$O(n)$	读写是 $O(\log n)$; 如果键在列表中, 就是 $O(\log n)$; 如果键不在列表中, 就是 $O(n)$	na	na

10.13 小结

本章介绍了如何处理不同类型的集合。数组的大小是固定的, 但可以使用列表作为动态增长的集合。队列以先进先出的方式访问元素, 栈以后进先出的方式访问元素。链表可以快速插入和删除元素, 但搜索操作比较慢。通过键和值可以使用字典, 它的搜索和插入操作比较快。集用于唯一项, 可以是无序的(`HashSet<T>`), 也可以是有序的(`SortedSet<T>`)。ObservableCollection<T>类提供了在列表中的元素发生变化时触发的事件。

本章还介绍了许多接口和类及其在集合访问和排序上的用法。探讨了一些特殊的集合, 如 BitArray 和 BitVector32, 它们为处理带有位的集合进行了优化。

第 11 章将详细介绍 LINQ。

第 11 章

LINQ

本章内容:

- 用 `List<T>` 在对象上执行传统查询
- 扩展方法
- LINQ 查询操作符
- 平行 LINQ
- 表达式树

语言集查询(Language Integrated Query, LINQ)集成了 C# 编程语言中的查询语法, 可以用相同的语法访问不同的数据源。LINQ 提供了不同数据源的抽象层, 所以可以使用相同的语法。

本章介绍 LINQ 的核心功能和 C# 中支持新特性的语言扩展。



读完本章后, 在数据库中使用 LINQ 的内容可查阅第 31 章, 查询 XML 数据的内容可参见第 33 章。

11.1 LINQ 概述

在介绍 LINQ 的特性之前, 本节先介绍一个简单的 LINQ 查询。C# 提供了转换为方法调用的集成查询语言。本节会说明这个转换的过程, 以便用户使用 LINQ 的全部功能。

11.1.1 列表和实体

本章的 LINQ 查询在一个包含 1950~2008 年一级方程式锦标赛的集合上进行。这些数据需要使用实体类和列表来准备。

对于实体, 定义类型 `Racer`。`Racer` 定义了几个属性和一个重载的 `ToString()` 方法, 该方法以字符串格式显示赛手。这个类实现了 `IFormattable` 接口, 以支持格式字符串的不同变体, 这个类还实现了 `IComparable<Racer>` 接口, 它根据 `Lastname` 为一组赛手排序。为了执行更高级的查询, `Racer` 类不仅包含单值属性, 如 `Firstname`、`Lastname`、`Wins`、`Country` 和 `Starts`, 还包含多值属性, 如 `Cars` 和 `Years`。`Years` 属性列出了赛手获得冠军的年份。一些赛手曾多次获得冠军。`Cars` 属性用于列出赛手在获得冠军的年份中使用的所有车型。



可从
wrox.com
下载源代码

```
using System;
using System.Text;

namespace Wrox.ProCSharp.LINQ
{
    [Serializable]
    public class Racer: IComparable<Racer>, IFormattable
    {
        public Racer(string firstName = null, string lastName = null,
            string country = null, int starts = 0, int wins = 0,
            IEnumerable<int> years = null, IEnumerable<string> cars = null)
        {
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Country = country;
            this.Starts = starts;
            this.Wins = wins;
            var yearsList = new List<int>();
            foreach (var year in years)
            {
                yearsList.Add(year);
            }
            this.Years = yearsList.ToArray();
            var carList = new List<string>();
            foreach (var car in cars)
            {
                carList.Add(car);
            }
            this.Cars = carList.ToArray();
        }

        public string FirstName {get; set;}
        public string LastName {get; set;}
        public int Wins {get; set;}
        public string Country {get; set;}
        public int Starts {get; set;}
        public string[] Cars { get; private set; }
        public int[] Years { get; private set; }

        public override string ToString()
        {
            return String.Format("{0} {1}", FirstName, LastName);
        }

        public int CompareTo(Racer other)
        {
            if (other == null) throw new ArgumentNullException("other");
            return this.LastName.CompareTo(other.LastName);
        }

        public string ToString(string format)
        {
            return ToString(format, null);
        }

        public string ToString(string format, IFormatProvider formatProvider)
```

```

    {
        switch (format)
        {
            case null:
            case "N":
                return ToString();
            case "F":
                return FirstName;
            case "L":
                return LastName;
            case "C":
                return Country;
            case "S":
                return Starts.ToString();
            case "W":
                return Wins.ToString();
            case "A":
                return String.Format("{0} {1}, {2}; starts: {3}, wins: {4}",
                    FirstName, LastName, Country, Starts, Wins);
            default:
                throw new FormatException(String.Format(
                    "Format {0} not supported", format));
        }
    }
}

```

代码段 DataLib/Racer.cs

第二个实体类是 **Team**。这个类仅包含车队冠军的名字和获得冠军的年份：



可从
wrox.com
下载源代码

```

[Serializable]
public class Team
{
    public Team(string name, params int[] years)
    {
        this.Name = name;
        this.Years = years;
    }

    public string Name { get; private set; }
    public int[] Years { get; private set; }
}

```

代码段 DataLib/Team.cs

Formula1 类在 **GetChampions()** 方法中返回一组赛车手。这个列表包含了 1950~2008 年之间的所有一级方程式冠军。



可从
wrox.com
下载源代码

```

using System.Collections.Generic;

namespace Wrox.ProCSharp.LINQ
{
    public static class Formula1
    {
        private static List<Racer> racers;
    }
}

```

```

public static IList<Racer> GetChampions()
{
    if (racers == null)
    {
        racers = new List<Racer>(40);
        racers.Add(new Racer("Nino", "Farina", "Italy", 33, 5,
            new int[] { 1950 },
            new string[] { "Alfa Romeo" }));
        racers.Add(new Racer("Alberto", "Ascari", "Italy", 32, 10,
            new int[] { 1952, 1953 },
            new string[] { "Ferrari" }));
        racers.Add(new Racer("Juan Manuel", "Fangio", "Argentina", 51, 24,
            new int[] { 1951, 1954, 1955, 1956, 1957 },
            new string[] { "Alfa Romeo", "Maserati",
                "Mercedes", "Ferrari" }));
        racers.Add(new Racer("Mike", "Hawthorn", "UK", 45, 3,
            new int[] { 1958 },
            new string[] { "Ferrari" }));
        racers.Add(new Racer("Phil", "Hill", "USA", 48, 3,
            new int[] { 1961 },
            new string[] { "Ferrari" }));
        racers.Add(new Racer("John", "Surtees", "UK", 111, 6,
            new int[] { 1964 },
            new string[] { "Ferrari" }));
        racers.Add(new Racer("Jim", "Clark", "UK", 72, 25,
            new int[] { 1963, 1965 },
            new string[] { "Lotus" }));
        racers.Add(new Racer("Jack", "Brabham", "Australia", 125, 14,
            new int[] { 1959, 1960, 1966 },
            new string[] { "Cooper", "Brabham" }));
        racers.Add(new Racer("Denny", "Hulme", "New Zealand", 112, 8,
            new int[] { 1967 },
            new string[] { "Brabham" }));
        racers.Add(new Racer("Graham", "Hill", "UK", 176, 14,
            new int[] { 1962, 1968 },
            new string[] { "BRM", "Lotus" }));
        racers.Add(new Racer("Jochen", "Rindt", "Austria", 60, 6,
            new int[] { 1970 },
            new string[] { "Lotus" }));
        racers.Add(new Racer("Jackie", "Stewart", "UK", 99, 27,
            new int[] { 1969, 1971, 1973 },
            new string[] { "Matra", "Tyrrell" }));

        //...
    }
    return racers;
}
}

```

代码段 DataLib/Formu1.cs

对于后面在多个列表中执行的查询，`GetConstructorChampions()`方法返回所有的车队冠军的列表。车队冠军是从 1958 年开始设立的。

```

private static List<Team> teams;
public static IList<Team> GetConstructorChampions()

```

```

    {
        if (teams == null)
        {
            teams = new List<Team>()
            {
                new Team("Vanwall", 1958),
                new Team("Cooper", 1959, 1960),
                new Team("Ferrari", 1961, 1964, 1975, 1976, 1977, 1979, 1982,
                    1983, 1999, 2000, 2001, 2002, 2003, 2004, 2007, 2008),
                new Team("BRM", 1962),
                new Team("Lotus", 1963, 1965, 1968, 1970, 1972, 1973, 1978),
                new Team("Brabham", 1966, 1967),
                new Team("Matra", 1969),
                new Team("Tyrrell", 1971),
                new Team("McLaren", 1974, 1984, 1985, 1988, 1989, 1990, 1991,
                    1998),
                new Team("Williams", 1980, 1981, 1986, 1987, 1992, 1993, 1994,
                    1996, 1997),
                new Team("Benetton", 1995),
                new Team("Renault", 2005, 2006 )
            };
        }
        return teams;
    }
}

```

11.1.2 LINQ 查询

使用这些准备好的列表和实体，进行 LINQ 查询，例如查询来自巴西的所有世界冠军，并按照夺冠次数排序。为此可以使用 `List<T>` 类的方法，如 `FindAll()` 和 `Sort()` 方法。而使用 LINQ 的语法非常简单：



可从
wrox.com
下载源代码

```

private static void LinqQuery()
{
    var query = from r in Formula1.GetChampions()
                where r.Country == "Brazil"
                orderby r.Wins descending
                select r;

    foreach (Racer r in query)
    {
        Console.WriteLine("{0:A}", r);
    }
}

```

代码段 LINQIntro/Program.cs

这个查询的结果显示了来自巴西的所有世界冠军，并排好序：

```

Ayrton Senna, Brazil; starts: 161, wins: 41
Nelson Piquet, Brazil; starts: 204, wins: 23
Emerson Fittipaldi, Brazil; starts: 143, wins: 14

```

语句

```

from r in Formula1.GetChampions()
where r.Country == "Brazil"

```

```
orderby r.Wins descending
select r;
```

是一个 LINQ 查询，子句 `from`、`where`、`orderby`、`descending` 和 `select` 都是这个查询中预定义的关键字。

查询表达式必须以 `from` 子句开头，以 `select` 或 `group` 子句结束。在这两个子句之间，可以使用 `where`、`orderby`、`join`、`let` 和其他 `from` 子句。



变量 `query` 只指定了 LINQ 查询。该查询不是通过这个赋值语句执行的，只要使用 `foreach` 循环访问查询，该查询就会执行。详见 11.1.5 节的内容。

11.1.3 扩展方法

编译器会修改 LINQ 查询，以调用方法。LINQ 为 `IEnumerable<T>` 接口提供了各种扩展方法，以便用户在实现了该接口的任意集合上使用 LINQ 查询。

扩展方法可以将方法写入最初没有提供该方法的类中。还可以把方法添加到实现某个特定接口的任何类中，这样多个类就可以使用相同的实现代码。

例如，`String` 类没有 `Foo()` 方法。`String` 类是密封的，所以不能从这个类中继承。但可以执行一个扩展方法，如下所示：

```
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo invoked for {0}", s);
    }
}
```

扩展方法在静态类中声明，定义为一个静态方法，其中第一个参数定义了它扩展的类型。`Foo()` 方法扩展了 `String` 类，因为它的第一个参数定义为 `String` 类型。为了区分扩展方法和一般的静态方法，扩展方法还需要对第一个参数使用 `this` 关键字。

现在就可以使用带 `string` 类型的 `Foo()` 方法了：

```
string s = "Hello";
s.Foo();
```

结果在控制台上显示 `Foo invoked for Hello`，因为 `Hello` 是传递给 `Foo()` 方法的字符串。

也许这看起来违反了面向对象的规则，因为给一个类型定义了新方法，但没有改变该类型或派生自它的类型。但实际上并非如此。扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种新语法。对于字符串，可以用如下方式调用 `Foo()` 方法，获得相同的结果：

```
string s = "Hello";
StringExtension.Foo(s);
```

要调用静态方法，应在类名的后面加上方法名。扩展方法是调用静态方法的另一种方式。不必提供定义了静态方法的类名，相反，调用静态方法是因为它带的参数类型。只需导入包含该类的名

称空间, 就可以将 `Foo()` 扩展方法放在 `String` 类的作用域中。

定义 LINQ 扩展方法的一个类是 `System.Linq` 名称空间中的 `Enumerable`。只需导入这个名称空间, 就打开了这个类的扩展方法的作用域。下面列出了 `Where()` 扩展方法的实现代码。`Where()` 扩展方法的第一个参数包含了 `this` 关键字, 其类型是 `IEnumerable<T>`。这样, `Where()` 方法就可以用于实现了 `IEnumerable<T>` 的每个类型。例如数组和 `List<T>` 类实现了 `IEnumerable<T>` 接口。第二个参数是一个 `Func<T, bool>` 委托, 它引用了一个返回布尔值、参数类型为 `T` 的方法。这个谓词在实现代码中调用, 检查 `IEnumerable<T>` 源中的项是否应放在目标集合中。如果委托引用了该方法, `yield return` 语句就将源中的项返回给目标。

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
        if (predicate(item))
            yield return item;
}
```

因为 `Where()` 作为一个泛型方法实现, 所以它可以用于包含在集合中的任意类型。实现了 `IEnumerable<T>` 接口的任意集合都支持它。



这里的扩展方法在 `System.Core` 程序集的 `System.Linq` 名称空间中定义。

现在就可以使用 `Enumerable` 类中的扩展方法 `Where()`、`OrderByDescending()` 和 `Select()`。这些方法都返回 `IEnumerable<TSource>`, 所以可以使用前面的结果依次调用这些方法。通过扩展方法的参数, 使用定义了委托参数的实现代码的匿名方法。



可从
wrox.com
下载源代码

```
static void ExtensionMethods()
{
    var champions = new List<Racer>(Formula1.GetChampions());
    IEnumerable<Racer> brazilChampions =
        champions.Where(r => r.Country == "Brazil").
            OrderByDescending(r => r.Wins).
            Select(r => r);

    foreach (Racer r in brazilChampions)
    {
        Console.WriteLine("{0:A}", r);
    }
}
```

代码段 LINQIntro/Program.cs

11.1.4 推迟查询的执行

在运行期间定义查询表达式时, 查询就不会运行。查询会在迭代数据项时运行。

再看看扩展方法 `Where()`。它使用 `yield return` 语句返回谓词为 `true` 的元素。因为使用了 `yield return` 语句, 所以编译器会创建一个枚举器, 在访问枚举中的项后, 就返回它们。

```

public static IEnumerable<T> Where<T> (this IEnumerable<T> source,
Func<T, bool> predicate)
{
    foreach (T item in source)
        if (predicate(item))
            yield return item;
}

```

这是一个非常有趣也非常重要的结果。在下面的例子中，创建了 `String` 元素的一个集合，用名称 `arr` 填充它。接着定义一个查询，从集合中找出以字母 `J` 开头的所有名称。集合也应是排好序的。在定义查询时，不会进行迭代。相反，迭代在 `foreach` 语句中进行，在其中迭代所有的项。集合中只有一个元素 `Juan` 满足 `where` 表达式的要求，即以字母 `J` 开头。迭代完成后，将 `Juan` 写入控制台。之后在集合中添加 4 个新名称，再次进行迭代。

```

var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };

var namesWithJ = from n in names
                 where n.StartsWith("J")
                 orderby n
                 select n;

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}

```

因为迭代在查询定义时不会进行，而是在执行每个 `foreach` 语句时进行，所以可以看到其中的变化，如应用程序的结果所示：

```

First iteration
Juan

Second iteration
Jack
Jim
John
Juan

```

当然，还必须注意，每次在迭代中使用查询时，都会调用扩展方法。在大多数情况下，这是非常有效的，因为我们可以检测出源数据中的变化。但是在一些情况下，这是不可行的。调用扩展方法 `ToArray()`、`ToArrayable()`、`ToList()` 等可以改变这个操作。在示例中，`ToList` 遍历集合，返回一

个实现了 `IList<string>` 的集合。之后对返回的列表遍历两次，在两次迭代之间，数据源得到了新名称。

```
var names = new List<string>
    { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names
    where n.StartsWith("J")
    orderby n
    select n).ToList();

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}
```

在结果中可以看到，在两次迭代之间输出保持不变，但集合中的值改变了：

```
First iteration
Juan

Second iteration
Juan
```

11.2 标准的查询操作符

`Where`、`OrderByDescending` 和 `Select` 只是 LINQ 的几个查询操作符。LINQ 查询为最常用的操作符定义了一个声明语法。还有许多查询操作符可用于 `Enumerable` 类。

表 11-1 列出了 LINQ 定义的标准查询操作符。

表 11-1

标准查询操作符	说 明
<code>Where OfType<TResult></code>	筛选操作符定义了返回元素的条件。在 <code>Where</code> 查询操作符中，可以使用谓词，例如 <code>Lambda</code> 表达式定义的谓词，来返回布尔值。 <code>OfType<TResult></code> 根据类型筛选元素，只返回 <code>TResult</code> 类型的元素
<code>Select</code> 和 <code>SelectMany</code>	投射操作符用于把对象转换为另一个类型的新对象。 <code>Select</code> 和 <code>SelectMany</code> 定义了根据选择器函数选择结果值的投射

(续表)

标准查询操作符	说 明
OrderBy、ThenBy、 OrderByDescending、 ThenByDescending、 Reverse	排序操作符改变所返回的元素的顺序。OrderBy 按升序排序，OrderByDescending 按降序排序。如果第一次排序的结果很类似，就可以使用 ThenBy 和 ThenBy Descending 操作符进行第二次排序。Reverse 反转集合中元素的顺序
Join、GroupJoin	连接运算符用于合并不直接相关的集合。使用 Join 操作符，可以根据键选择器函数连接两个集合，这类似于 SQL 中的 JOIN。GroupJoin 操作符连接两个集合，组合其结果
GroupBy、ToLookup	组合运算符把数据放在组中。GroupBy 操作符组合有公共键的元素。ToLookup 通过创建一个一对多字典，来组合元素
Any、All、Contains	如果元素序列满足指定的条件，量词操作符就返回布尔值。Any、All 和 Contains 都是限定符操作符。Any 确定集合中是否有满足谓词函数的元素；All 确定集合中的所有元素是否都满足谓词函数；Contains 检查某个元素是否在集合中。这些操作符都返回一个布尔值
Take、Skip、 TakeWhile SkipWhile	分区操作符返回集合的一个子集。Take、Skip、TakeWhile 和 SkipWhile 都是分区操作符。使用它们可以得到部分结果。使用 Take 必须指定要从集合中提取的元素个数；Skip 跳过指定的元素个数，提取其他元素；TakeWhile 提取条件为真的元素
Distinct、Union Intersect、Except、Zip	Set 操作符返回一个集合。Distinct 从集合中删除重复的元素。除了 Distinct 之外，其他 Set 操作符都需要两个集合。Union 返回出现在其中一个集合中的唯一元素。Intersect 返回两个集合中都有的元素。Except 返回只出现在一个集合中的元素。Zip 是 .NET 4 新增的，它把两个集合合并为一个
First、FirstOrDefault、 Last、LastOrDefault、 ElementAt、 ElementAtOrDefault、 Single、SingleOrDefault	这些元素操作符仅返回一个元素。First 返回第一个满足条件的元素。FirstOrDefault 类似于 First，但如果没有找到满足条件的元素，就返回类型的默认值。Last 返回最后一个满足条件的元素。ElementAt 指定了要返回的元素的位置。Single 只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异常
Count、Sum、Min、 Max、Average、 Aggregate	聚合操作符计算集合的一个值。利用这些聚合操作符，可以计算所有值的总和、所有元素的个数、值最大和最小的元素，以及平均值等
ToArray、ToEnumerable ToList、ToDictionary Cast<TResult>	这些转换操作符将集合转换为数组：IEnumerable、IList、IDictionary 等
Empty、Range、 Repeat	这些生成操作符返回一个新集合。使用 Empty 时集合是空的；Range 返回一系列数字；Repeat 返回一个始终重复一个值的集合

下面是使用这些操作符的一些例子。

11.2.1 筛选

下面介绍一些查询的示例。

使用 Where 子句，可以合并多个表达式。例如，找出赢得至少 15 场比赛的巴西和奥地利车手。传递给 where 子句的表达式的结果类型应是布尔类型：



可从
Wrox.com
下载源代码

```
var racers = from r in Formula1.GetChampions()
              where r.Wins > 15 &&
                  (r.Country == "Brazil" || r.Country == "Austria")
              select r;
foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

代码段 Enumerablesample/Program.cs

用这个 LINQ 查询启动程序，会返回 Niki Lauda、Nelson Piquet 和 Ayrton Senna，如下：

```
Niki Lauda, Austria, Starts: 173, Wins: 25
Nelson Piquet, Brazil, Starts: 204, Wins: 23
Ayrton Senna, Brazil, Starts: 161, Wins: 41
```

并不是所有的查询都可以用 LINQ 查询完成。也不是所有的扩展方法都映射到 LINQ 查询子句上。高级查询需要使用扩展方法。为了更好地理解带扩展方法的复杂查询，最好看看简单的查询是如何映射的。使用扩展方法 Where() 和 Select()，会生成与前面 LINQ 查询非常类似的结果：

```
var racers = Formula1.GetChampions().
    Where(r => r.Wins > 15 &&
        (r.Country == "Brazil" || r.Country == "Austria")).
    Select(r => r);
```

11.2.2 用索引筛选

不能使用 LINQ 查询的一个例子是 Where() 方法的重载。在 Where() 方法的重载中，可以传递第二个参数——索引。索引是筛选器返回的每个结果的计数器。可以在表达式中使用这个索引，执行基于索引的计算。下面的代码由 Where() 扩展方法调用，它使用索引返回姓氏以 A 开头、索引为偶数的车手：

```
var racers = Formula1.GetChampions().
    Where((r, index) => r.LastName.StartsWith("A") && index % 2 != 0);
foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

姓氏以 A 开头的车手有 Alberto Ascari、Mario Andretti 和 Fernando Alonso。因为 Mario Andretti 的索引是奇数，所以他不在结果中：

```
Alberto Ascari, Italy; starts: 32, wins: 10
Fernando Alonso, Spain; starts: 132, wins: 21
```

11.2.3 类型筛选

为了进行基于类型的筛选, 可以使用 `OfType()` 扩展方法。这里数组数据包含 `string` 和 `int` 对象。使用 `OfType()` 扩展方法, 把 `string` 类传送给泛型参数, 就从集合中仅返回字符串:

```
object[] data = { "one", 2, 3, "four", "five", 6 };
var query = data.OfType<string>();
foreach (var s in query)
{
    Console.WriteLine(s);
}
```

运行这段代码, 就会显示字符串 `one`、`four` 和 `five`。

```
one
four
five
```

11.2.4 复合的 from 子句

如果需要根据对象的一个成员进行筛选, 而该成员本身是一个系列, 就可以使用复合的 `from` 子句。`Racer` 类定义了一个属性 `Cars`, 其中 `Cars` 是一个字符串数组。要筛选驾驶法拉利的所有冠军, 可以使用如下所示的 LINQ 查询。第一个 `from` 子句访问从 `Formylal.GetChampions()` 方法返回的 `Racer` 对象, 第二个 `from` 子句访问 `Racer` 类的 `Cars` 属性, 以返回所有 `string` 类型的赛车。接着在 `Where` 子句中使用这些赛车筛选驾驶法拉利的所有冠军。

```
var ferrariDrivers = from r in Formylal.GetChampions()
                    from c in r.Cars
                    where c == "Ferrari"
                    orderby r.LastName
                    select r.FirstName + " " + r.LastName;
```

这个查询的结果显示了驾驶法拉利的所有一级方程式冠军:

```
Alberto Ascari
Juan Manuel Fangio
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

C# 编译器把复合的 `from` 子句和 LINQ 查询转换为 `SelectMany()` 扩展方法。`SelectMany()` 方法可用于迭代序列的序列。示例中 `SelectMany()` 方法的重载版本如下所示:

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult> (
    this IEnumerable<TSource> source,
    Func<TSource,
    IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult>
    resultSelector);
```

第一个参数是隐式参数，它从 `GetChampions()` 方法中接收 `Racer` 对象序列。第二个参数是 `collectionSelector` 委托，其中定义了内部序列。在 `Lambda` 表达式 `r => r.Cars` 中，应返回赛车集合。第三个参数是一个委托，现在为每个赛车调用该委托，接收 `Racer` 和 `Car` 对象。`Lambda` 表达式创建了一个匿名类型，它有 `Racer` 和 `Car` 属性。这个 `SelectMany()` 方法的结果是摊平了赛手和赛车的层次结构，为每辆赛车返回匿名类型的一个新对象集合。

这个新集合传递给 `Where()` 方法，筛选出驾驶法拉利的赛手。最后，调用 `OrderBy()` 和 `Select()` 方法：

```
var ferrariDrivers = Formula1.GetChampions().
    SelectMany(
        r => r.Cars,
        (r, c) => new { Racer = r, Car = c }).
    Where(r => r.Car == "Ferrari").
    OrderBy(r => r.Racer.LastName).
    Select(r => r.Racer.FirstName + " " + r.Racer.LastName);
```

把 `SelectMany()` 泛型方法解析为这里使用的类型，所解析的类型如下所示。在这个例子中，数据源是 `Racer` 类型，所筛选的集合是一个 `string` 数组，当然所返回的匿名类型的名称是未知的，这里显示为 `TResult`：

```
public static IEnumerable<TResult> SelectMany <Racer, string, TResult> (
    this IEnumerable <Racer> source,
    Func<Racer, IEnumerable<string>> collectionSelector,
    Func<Racer, string, TResult> resultSelector);
```

查询仅从 LINQ 查询转换为扩展方法，所以结果与前面的相同。

11.2.5 排序

要对序列排序，前面使用了 `orderby` 子句。下面复习一下前面使用 `orderby descending` 子句的例子。其中赛手按照赢得比赛的次数进行降序排序，赢得比赛的次数用关键字选择器指定：



可从
wrox.com
下载源代码

```
var racers = from r in Formula1.GetChampions()
    where r.Country == "Brazil"
    orderby r.Wins descending
    select r;
```

代码段 `EnumerableSample/Program.cs`

`orderby` 子句解析为 `OrderBy()` 方法，`orderby descending` 子句解析为 `OrderByDescending()` 方法：

```
var racers = Formula1.GetChampions().
    Where(r => r.Country == "Brazil").
    OrderByDescending(r => r.Wins).
    Select(r => r);
```

`OrderBy()` 和 `OrderByDescending()` 方法返回 `IOrderEnumerable<TSource>`。这个接口派生自 `IEnumerable<TSource>` 接口，但包含一个额外的方法 `CreateOrderedEnumerable<TSource>()`。这个方法用于进一步给序列排序。如果根据关键字选择器来排序，其中有两项相同，就可以使用 `ThenBy()` 和 `ThenByDescending()` 方法继续排序。这两个方法需要 `IOrderEnumerable<TSource>` 接口才能工作，

但也返回这个接口。所以，可以添加任意多个 `ThenBy()` 和 `ThenByDescending()` 方法，对集合排序。

使用 LINQ 查询时，只需把所有用于排序的不同关键字(用逗号分隔开)添加到 `orderby` 子句中。这里，所有的赛车手先按照国家排序，再按照姓氏排序，最后按照名字排序。添加到 LINQ 查询结果中的 `Take()` 扩展方法用于提取前 10 个结果：

```
var racers = (from r in Formula1.GetChampions()
              orderby r.Country, r.LastName, r.FirstName
              select r).Take(10);
```

排序后的结果如下：

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

使用 `OrderBy()` 和 `ThenBy()` 方法可以执行相同的操作：

```
var racers = Formula1.GetChampions().
    OrderBy(r => r.Country).
    ThenBy(r => r.LastName).
    ThenBy(r => r.FirstName).
    Take(10);
```

11.2.6 分组

要根据一个关键字值对查询结果分组，可以使用 `group` 子句。现在一级方程式冠军应按照国家分组，并列出一个国家的冠军数。子句 `group r by r.Country into g` 根据 `Country` 属性组合所有的赛车手，并定义一个新的标识符 `g`，它以后用于访问分组的结果信息。`group` 子句的结果根据应用到分组结果上的扩展方法 `Count()` 来排序，如果冠军数相同，就根据关键字来排序，该关键字是国家，因为这是分组所使用的关键字。`where` 子句根据至少有两项的分组来筛选结果，`select` 子句创建一个带 `Country` 和 `Count` 属性的匿名类型。

```
var countries = from r in Formula1.GetChampions()
                group r by r.Country into g
                orderby g.Count() descending, g.Key
                where g.Count() >= 2
                select new {
                    Country = g.Key,
                    Count = g.Count()
                };

foreach (var item in countries)
{
    Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
}
```

结果显示了带 Country 和 Count 属性的对象集合:

```
UK 9
Brazil 3
Australia 2
Austria 2
Finland 2
Italy 2
USA 2
```

要用扩展方法执行相同的操作, 应把 `groupby` 子句解析为 `GroupBy()` 方法。在 `GroupBy()` 方法的声明中, 注意它返回实现了 `IGrouping` 接口的枚举对象。`IGrouping` 接口定义了 `Key` 属性, 所以在定义了对这个方法的调用后, 可以访问分组的关键字:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey> (
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

把子句 `group r by r.Country into g` 解析为 `GroupBy(r => r.Country)`, 返回分组序列。分组序列首先用 `OrderByDescending()` 方法排序, 再用 `ThenBy()` 方法排序。接着调用 `Where()` 和 `Select()` 方法。

```
var countries = Formula1.GetChampions().
    GroupBy(r => r.Country).
    OrderByDescending(g => g.Count()).
    ThenBy(g => g.Key).
    Where(g => g.Count() >= 2).
    Select(g => new { Country = g.Key,
                    Count = g.Count() });
```

11.2.7 对嵌套的对象分组

如果分组的对象应包含嵌套的序列, 就可以改变 `select` 子句创建的匿名类型。在下面的例子中, 所返回的国家不仅应包含国家名和赛手数量这两个属性, 还应包含赛手名序列。这个序列用一个赋予 `Racers` 属性的 `from/in` 内部子句指定, 内部的 `from` 子句使用分组标识符 `g` 获得该分组中的所有赛手, 用姓氏对它们排序, 再根据姓名创建一个新字符串:



可从
wrox.com
下载源代码

```
var countries = from r in Formula1.GetChampions()
    group r by r.Country into g
    orderby g.Count() descending, g.Key
    where g.Count() >= 2
    select new
    {
        Country = g.Key,
        Count = g.Count(),
        Racers = from rl in g
            orderby rl.LastName
            select rl.FirstName + " " + rl.LastName
    };

foreach (var item in countries)
{
    Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
    foreach (var name in item.Racers)
    {
        Console.WriteLine("{0}; ", name);
    }
}
```

```

    }
    Console.WriteLine();
}

```

代码段 EnumerableSample/Program.cs

结果应列出某个国家的所有冠军：

```

UK 9
Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill; Damon Hill; James Hunt;
Nigel Mansell; Jackie Stewart; John Surtees;
Brazil 3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Australia 2
Jack Brabham; Alan Jones;
Austria 2
Niki Lauda; Jochen Rindt;
Finland 3
Mika Hakkinen; Kimi Räikkönen; Keke Rosberg;
Italy 2
Alberto Ascari; Nino Farina;
USA 2
Mario Andretti; Phil Hill;

```

11.2.8 连接

使用 `join` 子句可以根据特定的条件合并两个数据源，但之前要获得两个要连接的列表。在一级方程式比赛中，有赛手冠军和车队冠军。赛手从 `GetChampions()` 方法中返回，车队从 `GetConstructorChampions()` 方法中返回。现在要获得一个年份列表，列出每年的赛手冠军和车队冠军。

为此，先定义两个查询，用于查询赛手和车队：

```

var racers = from r in Formula1.GetChampions()
              from y in r.Years
              where y > 2003
              select new
              {
                  Year = y,
                  Name = r.FirstName + " " + r.LastName
              };

var teams = from t in
              Formula1.GetConstructorChampions()
              from y in t.Years
              where y > 2003
              select new
              {
                  Year = y,
                  Name = t.Name
              };

```

有了这两个查询，再通过子句 `join t in teams on r.Year equals t.Year`，根据赛手获得冠军的年份和车队获得冠军的年份进行连接。`select` 子句定义了一个新的匿名类型，它包含 `Year`、`Racer` 和 `Team` 属性。

```

var racersAndTeams =
    from r in racers
    join t in teams on r.Year equals t.Year
    select new
    {
        Year = r.Year,
        Racer = r.Name,
        Team = t.Name
    };
Console.WriteLine("Year Champion " + "Constructor Title");
foreach (var item in racersAndTeams)
{
    Console.WriteLine("{0}: {1,-20} {2}",
        item.Year, item.Racer, item.Team);
}

```

当然，也可以把它们合并为一个 LINQ 查询，但这只是一种尝试：

```

int year = 2003;
var racersAndTeams =
    from r in
        from rl in Formula1.GetChampions()
        from yr in rl.Years
        where yr > year
        select new
        {
            Year = yr,
            Name = rl.FirstName + " " + rl.LastName
        }
    join t in
        from tl in
            Formula1.GetConstructorChampions()
        from yt in tl.Years
        where yt > year
        select new
        {
            Year = yt,
            Name = tl.Name
        }
    on r.Year equals t.Year
    select new
    {
        Year = r.Year,
        Racer = r.Name,
        Team = t.Name
    };

```

结果显示了匿名类型中的数据：

Year Champion	Constructor Title
2004 Michael Schumacher	Ferrari
2005 Fernando Alonso	Renault
2006 Fernando Alonso	Renault
2007 Kimi Räikkönen	Ferrari
2008 Lewis Hamilton	Ferrari

11.2.9 集合操作

扩展方法 `Distinct()`、`Union()`、`Intersect()` 和 `Except()` 都是集合操作。下面创建一个驾驶法拉利的 F1 方程式冠军序列和驾驶迈凯轮的 F1 方程式冠军序列，然后确定是否有驾驶法拉利和迈凯轮的冠军。当然，这里可以使用 `Intersect()` 扩展方法。

首先获得所有驾驶法拉利的冠军。这只是一个简单的 LINQ 查询，其中使用复合的 `from` 子句访问 `Cars` 属性，该属性返回一个字符串对象序列。



可从
wrox.com
下载源代码

```
var ferrariDrivers = from r in
    Formula1.GetChampions()
    from c in r.Cars
    where c == "Ferrari"
    orderby r.LastName

select r;
```

代码段 `EnumerableSample/Program.cs`

现在建立另一个基本相同的查询，但 `where` 子句的参数不同，以获得所有驾驶迈凯轮的冠军。最好不要再次编写相同的查询。而可以创建一个方法，其中给它传递参数 `car`：

```
private static IEnumerable<Racer> GetRacersByCar(string car)
{
    return from r in Formula1.GetChampions()
           from c in r.Cars
           where c == car
           orderby r.LastName
           select r;
}
```

但是，因为该方法不需要在其他地方使用，所以应定义一个委托类型的变量来保存 LINQ 查询。`racerByCar` 变量必须是一个委托类型，该委托类型需要一个字符串参数，并返回 `IEnumerable<Racer>`，类似于前面实现的方法。为此，定义了几个泛型委托 `Func<>`，所以不需要声明自己的委托。把一个 Lambda 表达式赋予 `racerByCar` 变量。Lambda 表达式的左边定义了一个 `car` 变量，其类型是 `Func` 委托(字符串)的第一个泛型参数。右边定义了 LINQ 查询，它使用该参数和 `where` 子句：

```
Func<string, IEnumerable<Racer>> racersByCar =
    car => from r in Formula1.GetChampions()
           from c in r.Cars
           where c == car
           orderby r.LastName
           select r;
```

现在可以使用 `Intersect()` 扩展方法，获得驾驶法拉利和迈凯轮的所有冠军：

```
Console.WriteLine("World champion with Ferrari and McLaren");
foreach (var racer in racersByCar("Ferrari").Intersect(
    racersByCar("McLaren")))
{
    Console.WriteLine(racer);
}
```

结果只有一个赛手 Niki Lauda:

```
World champion with Ferrari and McLaren
Niki Lauda
```



集合操作通过调用实体类的 `GetHashCode()` 和 `Equals()` 方法来比较对象。对于自定义比较, 还可以传递一个实现了 `IEqualityComparer<T>` 接口的对象。在这里的示例中, `GetChampions()` 方法总是返回相同的对象, 因此默认的比较操作是有效的。

11.2.10 合并

`Zip()` 方法是 .NET 4 新增的, 允许用一个谓词函数把两个相关的序列合并为一个。

首先, 创建两个相关的序列, 它们使用相同的筛选(意大利国家)和排序方法。对于合并, 这很重要, 因为第一个集合中的第一项会与第二个集合中的第一项合并, 第一个集合中的第二项会与第二个集合中的第二项合并, 以此类推。如果两个序列的项数不同, `Zip()` 方法就在到达较小集合的末尾时停止。

第一个集合中的元素有一个 `Name` 属性, 第二个集合中的元素有 `LastName` 和 `Starts` 两个属性。

在 `racerNames` 集合上使用 `Zip()` 方法, 需要把第二个集合(`racerNamesAndStarts`)作为第一个参数。第二个参数的类型是 `Func<TFirst, TSecond, TResult>`。这个参数实现为一个 `Lambda` 表达式, 它通过参数 `first` 接收第一个集合的元素, 通过参数 `second` 接收第二个集合的元素。其实现代码创建并返回一个字符串, 该字符串包含第一个集合中元素的 `Name` 属性和第二个集合中元素的 `Starts` 属性:

```
var racerNames = from r in Formula1.GetChampions()
                  where r.Country == "Italy"
                  orderby r.Wins descending
                  select new
                  {
                      Name = r.FirstName + " " + r.LastName
                  };

var racerNamesAndStarts = from r in Formula1.GetChampions()
                           where r.Country == "Italy"
                           orderby r.Wins descending
                           select new
                           {
                               LastName = r.LastName,
                               Starts = r.Starts
                           };

var racers = racerNames.Zip(racerNamesAndStarts,
    (first, second) => first.Name + ", starts: " + second.Starts);
foreach (var r in racers)
{
    Console.WriteLine(r);
}
```

这个合并的结果是:

```
Alberto Ascari, starts: 32
```

Nino Farina, starts: 33

11.2.11 分区

扩展方法 `Take()` 和 `Skip()` 等的分区操作可用于分页，例如显示 5×5 个赛车。

在下面的 LINQ 查询中，把扩展方法 `Take()` 和 `Skip()` 添加到查询的最后。`Skip()` 方法先忽略根据页面大小和实际页数计算出的项数，再使用 `Take()` 方法根据页面大小提取一定数量的项：

```
int pageSize = 5;

int numberPages = (int)Math.Ceiling(Formula1.GetChampions().Count() /
    (double)pageSize);

for (int page = 0; page < numberPages; page++)
{
    Console.WriteLine("Page {0}", page);

    var racers =
        (from r in Formula1.GetChampions()
         orderby r.LastName
         select r.FirstName + " " + r.LastName).
        Skip(page * pageSize).Take(pageSize);

    foreach (var name in racers)
    {
        Console.WriteLine(name);
    }

    Console.WriteLine();
}
```

下面输出了前 3 页：

```
Page 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jim Clark

Page 1
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
Mika Hakkinen
Lewis Hamilton

Page 2
Mike Hawthorn
Phil Hill
Graham Hill
Damon Hill
Denny Hulme
```

分页在 Windows 或 Web 应用程序中非常有用，可以只给用户显示一部分数据。



这个分页机制的一个要点是，因为查询会在每个页面上执行，所以改变底层的数据会影响结果。在继续执行分页操作时，会显示新对象。根据不同的情况，这对于应用程序可能有利。如果这个操作是不需要的，就可以只对原来的数据源分页，然后使用映射到原始数据上的缓存。

使用 `TakeWhile()` 和 `SkipWhile()` 扩展方法，还可以传递一个谓词，根据谓词的结果提取或跳过某些项。

11.2.12 聚合操作符

聚合操作符(如 `Count()`、`Sum()`、`Min()`、`Max()`、`Average()` 和 `Aggregate()`)不返回一个序列，而返回一个值。

`Count()` 扩展方法返回集合中的项数。下面的 `Count()` 方法应用于 `Racer` 的 `Years` 属性，来筛选赛车手，只返回获得冠军次数超过 3 次的赛车手：

```
var query = from r in Formula1.GetChampions()
            where r.Years.Count() > 3
            orderby r.Years.Count() descending
            select new
            {
                Name = r.FirstName + " " + r.LastName,
                TimesChampion = r.Years.Count()
            };

foreach (var r in query)
{
    Console.WriteLine("{0} {1}", r.Name, r.TimesChampion);
}
```

结果如下：

```
Michael Schumacher 7
Juan Manuel Fangio 5
Alain Prost 4
```

`Sum()` 方法汇总序列中的所有数字，返回这些数字的和。下面的 `Sum()` 方法用于计算一个国家赢得比赛的总次数。首先根据国家对手分组，再在新创建的匿名类型中，把 `Wins` 属性赋予某个国家赢得比赛的总次数。

```
var countries =
    (from c in
        from r in Formula1.GetChampions()
        group r by r.Country into c
        select new
        {
            Country = c.Key,
            Wins = (from r1 in c
                    select r1.Wins).Sum()
        })
```

```

        orderby c.Wins descending, c.Country
        select c).Take(5);

    foreach (var country in countries)
    {
        Console.WriteLine("{0} {1}", country.Country, country.Wins);
    }

```

根据获得一级方程式冠军的次数，最成功的国家是：

```

UK 147
Germany 91
Brazil 78
France 51
Finland 42

```

方法 `Min()`、`Max()`、`Average()` 和 `Aggregate()` 的使用方式与 `Count()` 和 `Sum()` 相同。`Min()` 方法返回集合中的最小值，`Max()` 方法返回集合中的最大值，`Average()` 方法计算集合中的平均值。对于 `Aggregate()` 方法，可以传递一个 `Lambda` 表达式，该表达式对所有的值进行聚合。

11.2.13 转换

本章前面提到，查询可以推迟到访问数据项时再执行。在迭代中使用查询时，查询会执行。而使用转换操作符合立即执行查询，把查询结果放在数组、列表或字典中。

在下面的例子中，调用 `ToList()` 扩展方法，立即执行查询，得到的结果放在 `List<T>` 类中：



可从
wrox.com
下载源代码

```

List<Racer> racers =
    (from r in Formula1.GetChampions()
     where r.Starts > 150
     orderby r.Starts descending
     select r).ToList();

foreach (var racer in racers)
{
    Console.WriteLine("{0} {0:S}", racer);
}

```

代码段 `EnumerableSample/Program.cs`

把返回的对象放在列表中并没有这么简单。例如，对于集合类中从赛车到赛手的快速访问，可以使用新类 `Lookup<TKey, TElement>`。



`Dictionary<TKey, TValue>` 类只支持一个键对应一个值。在 `System.Linq` 名称空间的类 `Lookup<TKey, TElement>` 类中，一个键可以对应多个值。这些类详见第 10 章。

使用复合的 `from` 查询，可以摊平赛手和赛车序列，创建带有 `Car` 和 `Racer` 属性的匿名类型。在返回的 `Lookup` 对象中，键的类型应是表示汽车的 `string`，值的类型应是 `Racer`。为了进行这个选择，可以给 `ToLookup()` 方法的一个重载版本传递一个键和一个元素选择器。键选择器引用 `Car` 属性，元素选择器引用 `Racer` 属性。

```
var racers = (from r in Formula1.GetChampions()
```

```

        from c in r.Cars
        select new
        {
            Car = c,
            Racer = r
        }).ToLookup(cr => cr.Car, cr => cr.Racer);
if (racers.Contains("Williams"))
{
    foreach (var williamsRacer in
        racers["Williams"])
    {
        Console.WriteLine(williamsRacer);
    }
}

```

用 Lookup 类的索引器访问的所有 Williams 冠军如下:

```

Alan Jones
Keke Rosberg
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve

```

如果需要在非类型化的集合上(如 ArrayList)使用 LINQ 查询,就可以使用 Cast()方法。在下面的例子中,基于 Object 类型的 ArrayList 集合用 Racer 对象填充。为了定义强类型化的查询,可以使用 Cast()方法。

```

var list = new System.Collections.ArrayList(Formula1.GetChampions()
    as System.Collections.ICollection);

var query = from r in list.Cast<Racer>()
    where r.Country == "USA"
    orderby r.Wins descending
    select r;

foreach (var racer in query)
{
    Console.WriteLine("{0:A}", racer);
}

```

11.2.14 生成操作符

生成操作符 Range()、Empty()和 Repeat()不是扩展方法,而是返回序列的正常静态方法。在 LINQ to Objects 中,这些方法可用于 Enumerable 类。

有时需要填充一个范围的数字,此时就应使用 Range()方法。这个方法把第一个参数作为起始值,把第二个参数作为要填充的项数。

```

var values = Enumerable.Range(1, 20);
foreach (var item in values)
{
    Console.Write("{0} ", item);
}

Console.WriteLine();

```

当然，结果如下所示：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```



Range()方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查询，并返回一个 **RangeEnumerator**，其中只有一条 **yield return** 语句，来递增值。

可以把该结果与其他扩展方法合并起来，获得另一个结果，例如，使用 **Select()**扩展方法：

```
var values = Enumerable.Range(1, 20).Select(n => n * 3);
```

Empty()方法返回一个不返回值的迭代器，它可以用于需要一个集合的参数，其中可以给参数传递空集合。

Repeat()方法返回一个迭代器，该迭代器把同一个值重复特定的次数。

11.3 并行 LINQ

.NET 4 在 **System.Linq** 名称空间中包含一个新类 **ParallelEnumerable**，可以分解查询的工作使其分布在多个线程上。尽管 **Enumerable** 类给 **IEnumerable<T>** 接口定义了扩展方法，但 **ParallelEnumerable** 类的大多数扩展方法是 **ParallelQuery<TSource>** 类的扩展。一个重要的例外是 **AsParallel()** 方法，它扩展了 **IEnumerable<TSource>** 接口，返回 **ParallelQuery<TSource>** 类，所以正常的集合类可以以平行方式查询。

11.3.1 并行查询

为了说明并行 LINQ，需要一个大型集合。对于可以放在 CPU 的缓存中的小集合，并行 LINQ 看不出效果。在下面的代码中，用随机值填充一个大型的 **int** 数组：



可从
wrox.com
下载源代码

```
const int arraySize = 100000000;
var data = new int[arraySize];
var r = new Random();
for (int i = 0; i < arraySize; i++)
{
    data[i] = r.Next(40);
}
```

代码段 **ParallelLinqSample/Program.cs**

现在可以使用 LINQ 查询筛选数据，获取所筛选数据的总和。该查询用 **where** 子句定义了一个筛选器，仅汇总对应值小于 20 的项，接着调用聚合函数 **Sum()** 方法。与前面的 LINQ 查询的唯一区别是，这次调用了 **AsParallel()** 方法。

```
var sum = (from x in data.AsParallel()
           where x < 20
           select x).Sum();
```

与前面的 LINQ 查询一样，编译器会修改语法，以调用 **AsParallel()**、**Where()**、**Select()** 和 **Sum()**

方法。AsParallel()方法用 ParallelEnumerable 类定义,以扩展 IEnumerable<T>接口,所以可以对简单的数组调用它。AsParallel()方法返回 ParallelQuery<TSource>。因为返回的类型,所以编译器选择的 Where()方法是 ParallelEnumerable.Where(),而不是 Enumerable.Where()。在下面的代码中,Select()和 Sum()方法也来自 ParallelEnumerable 类。与 Enumerable 类的实现代码相反,对于 ParallelEnumerable 类,查询是分区的,以便多个线程可以同时处理该查询。数组可以分为多个部分,其中每个部分由不同的线程处理,以筛选其余项。完成分区的工作后,就需要合并,获得所有部分的总和。

```
var sum = data.AsParallel().Where(x => x < 20).Select(x => x).Sum();
```

运行这行代码会启动任务管理器,这样就可以看出系统的所有 CPU 都在忙碌。如果删除 AsParallel()方法,就不可能使用多个 CPU。当然,如果系统上没有多个 CPU,就不会看到并行版本带来的改进。

11.3.2 分区器

AsParallel()方法不仅扩展了 IEnumerable<T>接口,还扩展了 Partitioner 类。通过它,可以影响到创建的分区。

Partitioner 类用 System.Collection.Concurrent 名称空间定义,并且有不同的变体。Create()方法接受实现了 IList<T>类的数组或对象。根据这一点,以及类型的参数 loadBalance 和该方法的一些重载版本,会返回一个不同的 Partitioner 类型。对于数组, .NET 4 包含派生自抽象基类 OrderablePartitioner<TSource> 的 DynamicPartitionerForArray<TSource> 类和 StaticPartitionerForArray<TSource>类。

修改 11.3.1 节中的代码,手工创建一个分区器,而不是使用默认的分区分器:

```
var sum = (from x in Partitioner.Create(data, true).AsParallel()
           where x < 20
           select x).Sum();
```

也可以调用 WithExecutionMode()和 WithDegreeOfParallelism()方法,来影响并行机制。对于 WithExecutionMode()方法可以传递 ParallelExecutionMode 的一个 Default 值或者 ForceParallelism 值。默认情况下,并行 LINQ 避免使用系统开销很高的并行机制。对于 WithDegreeOfParallelism()方法,可以传递一个整数值,以指定应并行运行的最大任务数。

11.3.3 取消

.NET 4 提供了一种标准方式,来取消长时间运行的任务,这也适用于并行 LINQ。

要取消长时间运行的查询,可以给查询添加 WithCancellation()方法,并传递一个 CancellationToken 令牌作为参数。CancellationToken 令牌从 CancellationTokenSource 类中创建。该查询在单独的线程中运行,在该线程中,捕获一个 OperationCanceledException 类型的异常。如果取消了查询,就触发这个异常。在主线程中,调用 CancellationTokenSource 类的 Cancel()方法可以取消任务。

```
var cts = new CancellationTokenSource();

new Thread(() =>
{
    try
    {
```

```

        var sum = (from x in data.AsParallel().
                    WithCancellation(cts.Token)
                    where x < 80
                    select x).Sum();
        Console.WriteLine("query finished, sum: {0}", sum);
    }
    catch (OperationCanceledException ex)
    {
        Console.WriteLine(ex.Message);
    }
    }).Start();

    Console.WriteLine("query started");
    Console.Write("cancel? ");
    int input = Console.Read();
    if (input == 'Y' || input == 'y')
    {
        // cancel!
        cts.Cancel();
    }
}

```



关于取消和 CancellationToken 令牌的内容详见第 20 章。

11.4 表达式树

在 LINQ to Objects 中, 扩展方法需要将一个委托类型作为参数, 这样就可以将 Lambda 表达式赋予参数。Lambda 表达式也可以赋予 Expression<T>类型的参数。C#编译器根据类型给 Lambda 表达式定义不同的行为。如果类型是 Expression<T>, 编译器就从 Lambda 表达式中创建一个表达式树, 并存储在程序集中。这样, 就可以在运行期间分析表达式树, 并进行优化, 以便于查询数据源。

下面看看一个前面使用的查询表达式:



可从
wrox.com
下载源代码

```

var brazilRacers = from r in racers
                    where r.Country == "Brazil"
                    orderby r.Wins
                    select r;

```

代码段 ExpressionTreeSample/Program.cs

这个查询表达式使用了扩展方法 Where()、OrderBy() 和 Select()。Enumerable 类定义了 Where() 扩展方法, 并将委托类型 Func<T,bool> 作为参数谓词。

```

public static IEnumerable<TSource>Where<TSource>(this IEnumerable<TSource>source,
        Func<TSource, bool>predicate);

```

这样, 就把 Lambda 表达式赋予谓词。这里 Lambda 表达式类似于前面介绍的匿名方法。

```

Func<Racer, bool>predicate = r => r.Country == "Brazil";

```

Enumerable 类不是唯一一个定义了扩展方法 Where() 的类。Queryable<T> 类也定义了 Where() 扩

展方法。这个类对 `Where()` 扩展方法的定义是不同的:

```
public static IQueryable<TSource>Where<TSource>(this IQueryable<TSource> source,
    Expression<Func<TSource, bool>> predicate);
```

其中, 把 `Lambda` 表达式赋予类型 `Expression<T>`, 该类型的操作是不同的:

```
Expression<Func<Racer, bool>> predicate = r => r.Country == "Brazil";
```

除了使用委托之外, 编译器还会把表达式树放在程序集中。表达式树可以在运行期间读取。表达式树从派生自抽象基类 `Expression` 的类中构建。`Expression` 类与 `Expression<T>` 不同。继承自 `Expression` 类的表达式类有 `BinaryExpression`、`ConstantExpression`、`InvocationExpression`、`LambdaExpression`、`NewExpression`、`NewArrayExpression`、`TernaryExpression` 和 `UnaryExpression` 等。编译器会从 `Lambda` 表达式中创建表达式树。

例如, `Lambda` 表达式 `r.Country == "Brazil"` 使用了 `ParameterExpression`、`MemberExpression`、`ConstantExpression` 和 `MethodCallExpression`, 来创建一个表达式树, 并将该树存储在程序集中。之后在运行期间使用这个树, 创建一个用于底层数据源的优化查询。

`DisplayTree()` 方法在控制台上图形化地显示表达式树。其中传递了一个 `Expression` 对象, 并根据表达式的类型, 把表达式的一些信息写到控制台上。根据表达式的类型, 递归地调用 `DisplayTree()` 方法。



在这个方法中, 没有处理所有的表达式类型, 只处理了在下一个示例表达式中使用的类型:

```
private static void DisplayTree(int indent, string message,
    Expression expression)
{
    string output = String.Format("{0} {1} ! NodeType: {2}; Expr: {3} ",
        "", PadLeft(indent, '>'), message, expression.NodeType, expression);

    indent++;
    switch (expression.NodeType)
    {
        case ExpressionType.Lambda:
            Console.WriteLine(output);
            LambdaExpression lambdaExpr = (LambdaExpression)expression;
            foreach (var parameter in lambdaExpr.Parameters)
            {
                DisplayTree(indent, "Parameter", parameter);
            }
            DisplayTree(indent, "Body", lambdaExpr.Body);
            break;
        case ExpressionType.Constant:
            ConstantExpression constExpr = (ConstantExpression)expression;
            Console.WriteLine("{0} Const Value: {1}", output, constExpr.Value);
            break;
        case ExpressionType.Parameter:
            ParameterExpression paramExpr = (ParameterExpression)expression;
```

```

        Console.WriteLine("{0} Param Type: {1}", output,
                           paramExpr.Type.Name);
        break;
    case ExpressionType.Equal:
    case ExpressionType.AndAlso:
    case ExpressionType.GreaterThan:
        BinaryExpression binExpr = (BinaryExpression)expression;
        if (binExpr.Method != null)
        {
            Console.WriteLine("{0} Method: {1}.", output,
                               binExpr.Method.Name);
        }
        else
        {
            Console.WriteLine(output);
        }
        DisplayTree(indent, "Left", binExpr.Left);
        DisplayTree(indent, "Right", binExpr.Right);
        break;
    case ExpressionType.MemberAccess:
        MemberExpression memberExpr = (MemberExpression)expression;
        Console.WriteLine("{0} Member Name: {1}, Type: {2}", output,
                           memberExpr.Member.Name, memberExpr.Type.Name);
        DisplayTree(indent, "Member Expr", memberExpr.Expression);
        break;
    default:
        Console.WriteLine();
        Console.WriteLine("{0} {1}", expression.NodeType,
                           expression.Type.Name);
        break;
    }
}

```

前面已经介绍了用于显示表达式树的表达式。这是一个 Lambda 表达式，它有一个 `Racer` 参数，表达式体提取赢得比赛次数超过 6 次的巴西赛车手：

```

Expression<Func<Racer, bool>> expression =
    r => r.Country == "Brazil" && r.Wins > 6;

DisplayTree(0, "Lambda", expression);

```

下面看看结果。Lambda 表达式包含一个 `Parameter` 和一个 `AndAlso` 节点类型。`AndAlso` 节点类型的左边是一个 `Equal` 节点类型，右边是一个 `GreaterThan` 节点类型。`Equal` 节点类型的左边是 `MemberAccess` 节点类型，右边是 `Constant` 节点类型。

```

Lambda! NodeType: Lambda; Expr: r => ((r.Country == "Brazil")
    AndAlso (r.Wins > 6))
> Parameter! NodeType: Parameter; Expr: r Param Type: Racer
> Body! NodeType: AndAlso; Expr: ((r.Country == "Brazil")
    AndAlso (r.Wins > 6))
>> Left! NodeType: Equal; Expr: (r.Country == "Brazil") Method: op_Equality
>>> Left! NodeType: MemberAccess; Expr: r.Country
    Member Name: Country, Type: String
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer

```

```
> > > Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil
> > Right! NodeType: GreaterThan; Expr: (r.Wins > 6)
> > > Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32
> > > > Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
> > > Right! NodeType: Constant; Expr: 6 Const Value: 6
```

使用 `Expression<T>` 类型的一个例子是 ADO.NET Entity Framework 和 LINQ to SQL。这些技术用 `Expression<T>` 参数定义了扩展方法。这样，访问数据库的 LINQ 提供程序就可以读取表达式，创建一个运行期间优化的查询，从数据库中获取数据。

11.5 LINQ 提供程序

.NET 4 包含几个 LINQ 提供程序。LINQ 提供程序为特定的数据源实现了标准的查询操作符。LINQ 提供程序也许会实现比 LINQ 定义的更多扩展方法，但至少要实现标准操作符。LINQ to XML 不仅实现了专门用于 XML 的方法，还实现了其他方法，例如，`System.Xml.Linq` 名称空间中的 `Extensions` 类定义的 `Elements()`、`Descendants()` 和 `Ancestors()` 方法。

LINQ 提供程序的实现方案是根据名称空间和第一个参数的类型来选择的。实现扩展方法的类的名称空间必须是开放的，否则扩展类就不在作用域内。在 LINQ to Objects 中定义的 `Where()` 方法的参数和在 LINQ to SQL 中定义的 `Where()` 的方法参数不同。

LINQ to Objects 中的 `Where()` 方法用 `Enumerable` 类定义：

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

在 `System.Linq` 名称空间中，还有另一个类实现了操作符 `Where`。这个实现代码由 LINQ to SQL 使用。这些实现代码在 `Queryable` 类中可以找到：

```
public static IQueryable<TSource> Where<TSource> (this IQueryable<TSource> source,
    Expression<Func< TSource, bool>> predicate);
```

这两个类都在 `System.Linq` 名称空间的 `System.Core` 程序集中实现。那么，它是如何定义的？使用了什么方法？无论是用 `Func<TSource, bool>` 参数传递，还是用 `Expression<Func<TSource, bool>>` 参数传递，`Lambda` 表达式都相同。只是编译器的行为不同，它根据 `source` 参数来选择。编译器根据其参数选择最匹配的方法。在 ADO.NET Entity Framework 中定义的 `ObjectContext` 类的 `CreateQuery<T>()` 方法返回一个实现了 `IQueryable<TSource>` 接口的 `ObjectQuery<T>` 对象，因此 Entity Framework 使用类 `Queryable` 类的 `Where()` 方法。

11.6 小结

本章讨论了 LINQ 查询和查询所基于的语言结构，如扩展方法和 `Lambda` 表达式，还列出了各种 LINQ 查询操作符，它们不仅用于筛选数据源，给数据源排序，还用于执行分区、分组、转换、连接等操作。

使用并行 LINQ 可以轻松地并行化运行时间较长的查询。

另一个重要的概念是表达式树。表达式树允许在运行期间构建对数据源的查询，因为表达式树存储在程序集中。表达式树的用法详见第 31 章。LINQ 是一个非常深奥的主题，更多的信息可查阅第 31 章和第 33 章。还可以下载其他第三方提供程序，例如，LINQ to MySQL、LINQ to Amazon、LINQ to Flickr、LINQ to LDAP 和 LINQ to SharePoint。无论使用什么数据源，都可以通过 LINQ 使用相同的查询语法。

第 12 章

动态语言扩展

本章内容:

- 理解 Dynamic Language Runtime
- dynamic 类型
- DLR ScriptRuntime
- DynamicObject
- ExpandoObject

随着 Ruby、Python 等语言的成长, JavaScript 的使用主要集中在动态编程方面。在 .NET Framework 的以前版本中, var 关键字和匿名方法开辟出 C# 的“动态编程”路径。在版本 4 中, 增加了 dynamic 类型。尽管 C# 仍是一种静态的类型化语言, 但这些新增内容给它提供了一些开发人员期望的动态功能。

本章介绍 dynamic 类型及其使用规则, 并讨论 DynamicObject 的实现方式和使用方式。

12.1 DLR

C# 4 的动态功能是 Dynamic Language Runtime(动态语言运行时, DLR)的一部分。DLR 是添加到 CLR 的一系列服务, 它允许添加动态语言, 如 Ruby 和 Python, 并使 C# 具备和这些动态语言相同的某些动态功能。

在 CodePlex 网站上有一个开源的 DLR 版本, 这个版本也包含在 .NET 4 Framework 中, 还增加了对语言实现程序的一些支持。

在 .NET Framework 中, DLR 位于 System.Dynamic 名称空间和 System.Runtime.CompilerServices 名称空间的几个类中。

IronRuby 和 IronPython 是 Ruby 和 Python 语言的开源版本, 它们使用 DLR。Silverlight 也使用 DLR。通过包含 DLR, 可以给应用程序添加脚本编辑功能。脚本运行库允许给脚本传入变量和从脚本传出变量。

12.2 dynamic 类型

dynamic 类型允许编写忽略编译期间的类型检查的代码。编译器假定, 给 dynamic 类型的对象

定义的任何操作都是有效的。如果该操作无效，则在代码运行之前不会检测该错误，如下面的示例所示：

```
class Program
{
    static void Main(string[] args)
    {
        var staticPerson = new Person();
        dynamic dynamicPerson = new Person();
        staticPerson.GetFullName("John", "Smith");
        dynamicPerson.GetFullName("John", "Smith");
    }
}

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return string.Concat(FirstName, " ", LastName);
    }
}
```

这个示例没有编译，因为它调用了 `staticPerson.GetFullName()` 方法。因为 `Person` 对象上的方法不接受两个参数，所以编译器会提示出错。如果注释掉该行代码，这个示例就会编译。如果执行它，就会发生一个运行错误。所抛出的异常是 `RuntimeBinderException` 异常。`RuntimeBinder` 对象会在运行时判断该调用，确定 `Person` 类是否支持被调用的方法。这将在本章后面讨论。

与 `var` 关键字不同，定义为 `dynamic` 的对象可以在运行期间改变其类型。注意在使用 `var` 关键字时，对象类型的确定会延迟。类型一旦确定，就不能改变。动态对象的类型可以改变，而且可以改变多次，这不同于把对象的类型强制转换为另一种类型。在强制转换对象的类型时，是用另一种兼容的类型创建一个新对象。例如，不能把 `int` 强制转换为 `Person` 对象。在下面的示例中，如果对象是动态对象，就可以把它从 `int` 变成 `Person` 类型：



可从
wrox.com
下载源代码

```
dynamic dyn;

dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);

dyn = "This is a string";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);

dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine("{0} {1}", dyn.FirstName, dyn.LastName);
```

代码段 `Dynamic\Program.cs`

执行这段代码可以看出，`dyn` 对象的类型实际上从 `System.Int32` 变成 `System.String`，再变成 `Person`。如果 `dyn` 声明为 `int` 或 `string`，这段代码就不会编译。

对于 `dynamic` 类型有两个限制。动态对象不支持扩展方法，匿名函数(Lambda 表达式)也不能用作动态方法调用的参数，因此 LINQ 不能用于动态对象。大多数 LINQ 调用都是扩展方法，而 Lambda 表达式用作这些扩展方法的参数。

后台上的动态操作

在后台，这些是如何发生的？C#仍是一种静态的类型化语言，这一点没有改变。看看使用 `dynamic` 类型生成的 IL(中间语言)。

首先，看看下面的示例 C#代码：

```
using System;

namespace DeCompile
{
    class Program
    {
        static void Main(string[] args)
        {
            StaticClass staticObject = new StaticClass();
            DynamicClass dynamicObject = new DynamicClass();
            Console.WriteLine(staticObject.IntValue);
            Console.WriteLine(dynamicObject.DynValue);
            Console.ReadLine();
        }
    }

    class StaticClass
    {
        public int IntValue = 100;
    }

    class DynamicClass
    {
        public dynamic DynValue = 100;
    }
}
```

其中有两个类 `StaticClass` 和 `DynamicClass`。`StaticClass` 类有唯一一个返回 `int` 的字段。`DynamicClass` 有唯一一个返回 `dynamic` 对象的字段。`Main()`方法仅创建了这些对象，并输出方法返回的值。该示例非常简单。

现在注释掉 `Main()`方法中对 `DynamicClass` 类的引用：

```
static void Main(string[] args)
{
    StaticClass staticObject = new StaticClass();
    //DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

使用 `ildasm` 工具(参见第 18 章)，可以看到对于 `Main()`方法生成的 IL：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size 26 (0x1a)
    .maxstack 1
    .locals init ([0] class DeCompile.StaticClass staticObject)
    IL_0000: nop
    IL_0001: newobj instance void DeCompile.StaticClass::.ctor()
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: ldfld int32 DeCompile.StaticClass::IntValue
    IL_000d: call void [mscorlib]System.Console::WriteLine(int32)
    IL_0012: nop
    IL_0013: call string [mscorlib]System.Console::ReadLine()
    IL_0018: pop
    IL_0019: ret
} // end of method Program::Main
```

这里不讨论 IL 的细节，只看看这段代码，就可以看出其作用。第 0001 行调用了 `StaticClass` 构造函数，第 0008 行调用了 `StaticClass` 类的 `IntValue` 字段。下一行输出了其值。

现在注释掉对 `StaticClass` 类的引用，取消 `DynamicClass` 引用的注释：

```
static void Main(string[] args)
{
    //StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

再次编译应用程序，下面是生成的 IL：

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size 121 (0x79)
    .maxstack 9
    .locals init ([0] class DeCompile.DynamicClass dynamicObject,
        [1] class [Microsoft.CSharp]Microsoft.CSharp.
            RuntimeBinder.CSharpArgumentInfo[] CS$0$0000)
    IL_0000: nop
    IL_0001: newobj instance void DeCompile.DynamicClass::.ctor()
    IL_0006: stloc.0
    IL_0007: ldsfld class [System.Core]System.Runtime.CompilerServices. CallSite`1
        <class [mscorlib]
System.Action`3 < class
[System.Core]System.Runtime.CompilerServices.CallSite,class [mscorlib]
System.Type,object >> DeCompile.Program/'<Main> o__SiteContainer0'::'<>p__Site1'
    IL_000c: brtrue.s IL_004d
    IL_000e: ldc.i4.0
    IL_000f: ldstr "WriteLine"
    IL_0014: ldtoken DeCompile.Program
    IL_0019: call class [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
```

```

IL_001e: ldnull
IL_001f: ldc.i4.2
IL_0020: newarr [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.
CSharpArgumentInfo
IL_0025: stloc.1
IL_0026: ldloc.1
IL_0027: ldc.i4.0
IL_0028: ldc.i4.s 33
IL_002a: ldnull
IL_002b: newobj instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.
CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder.CSharpArgumentInfoFlags, string)
IL_0030: stelem.ref
IL_0031: ldloc.1
IL_0032: ldc.i4.1
IL_0033: ldc.i4.0
IL_0034: ldnull
IL_0035: newobj instance void [Microsoft.CSharp]Microsoft.
CSharp.RuntimeBinder.CSharpArgumentInfo::.ctor(valuetype
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.
CSharpArgumentInfoFlags,
string)

IL_003a: stelem.ref
IL_003b: ldloc.1
IL_003c: newobj instance void [Microsoft.CSharp]Microsoft.CSharp
RuntimeBinder.CSharpInvokeMemberBinder::.ctor(valuetype
Microsoft.CSharp) Microsoft.CSharp.RuntimeBinder.CSharpCallFlags,
string,

class [mscorlib]System.Type,

class [mscorlib]System.Collections.Generic.IEnumerable`1
<class [mscorlib]System.Type> ,

class [mscorlib]System.Collections.Generic.IEnumerable`1
<class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder. CSharpArgumentInfo>)
IL_0041: call class [System.Core]System.Runtime.CompilerServices.CallSite`1
<!> class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> ::Create(class [System.Core]
System.Runtime.CompilerServices.CallSiteBinder)
IL_0046: stsfld class [System.Core]System.Runtime. CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> DeCompile.Program/'<Main>
o__SiteContainer0'::'<>p__Site1'
IL_004b: br.s IL_004d
IL_004d: ldsfld class [System.Core]System.Runtime. CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object >> DeCompile.Program/'<Main>
o__SiteContainer0'::' <>p__Site1'
IL_0052: ldfld !0 class [System.Core]System.Runtime.
CompilerServices.CallSite`1

```

```

<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Target
IL_0057: ldsfld class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>> DeCompile.Program/'<Main>
    o _SiteContainer0'::'<>p _Site1'
IL_005c: ldtoken [mscorlib]System.Console
IL_0061: call class [mscorlib]System.Type
    [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
IL_0066: ldloc.0
IL_0067: ldfld object DeCompile.DynamicClass::DynValue
IL_006c: callvirt instance void class [mscorlib]System.Action`3
    <class [System.Core]System.Runtime.CompilerServices.CallSite, class
    [mscorlib]System.Type,object>::Invoke(!0,!1,!2)
IL_0071: nop
IL_0072: call string [mscorlib]System.Console::ReadLine()
IL_0077: pop
IL_0078: ret
} // end of method Program::Main

```

显然，C#编译器做了许多工作，以支持动态类型。在生成的代码中，会看到对 `System.Runtime.CompilerServices.CallSite` 类和 `System.Runtime.CompilerServices.CallSiteBinder` 类的引用。

`CallSite` 是在运行期间处理查找操作的类型。在运行期间调用动态对象时，必须找到该对象，看看其成员是否存在。`CallSite` 会缓存这个信息，这样查找操作就不需要重复执行。没有这个过程，循环结构的性能就有问题。

`CallSite` 完成了成员查找操作后，就调用 `CallSiteBinder()` 方法。它从 `CallSite` 中提取信息，并生成表达式树，来表示绑定器绑定的操作。

显然这需要做许多工作。优化非常复杂的操作时要格外小心。显然，使用 `dynamic` 类型是有用的，但它是有代价的。

12.3 包含 DLR ScriptRuntime

假定能给应用程序添加脚本编辑功能，并给脚本传入数值和从脚本传出数值，使应用程序可以利用脚本完成工作。这些都是在应用程序中包含 DLR 的 `ScriptRuntime` 而提供的功能。目前，`IronRuby`、`IronPython` 和 `JavaScript` 都支持包含在应用程序中的脚本语言。

有了 `ScriptRuntime`，就可以执行存储在文件中的代码段或完整的脚本。可以选择合适的语言引擎，或者让 DLR 确定使用什么引擎。脚本可以在自己的应用程序域或者在当前的应用程序域中创建。不仅可以给脚本传入数值并从脚本中传出数值，还可以在脚本中调用在动态对象上创建的方法。

这种灵活性为包含 `ScriptRuntime` 提供了无数种用法。下面的示例说明了使用 `ScriptRuntime` 的一种方式。假定有一个购物车应用程序，它的一个要求是根据某种标准计算折扣。这些折扣常常随着新销售策略的启动和完成而变化。处理这个要求有许多方式，本例将说明如何使用 `ScriptRuntime` 和少量 `Python` 脚本达到这个要求。

为了简单起见,本例是一个 Windows 客户端应用程序。它也可以是一个大型 Web 应用程序或任何其他应用程序的一部分。图 12-1 显示了这个应用程序的样例屏幕。

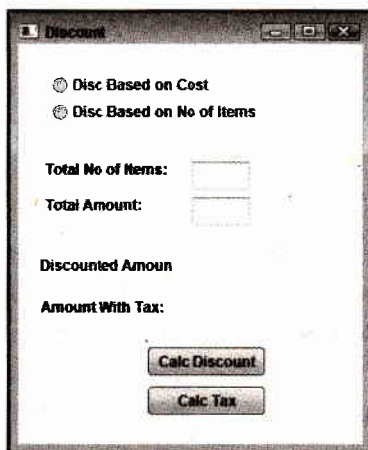


图 12-1

该应用程序提取所购买的物品数量和物品的总价,并根据所选的单选按钮使用某个折扣。在实际的应用程序中,系统使用略微复杂的方式确定要使用的折扣,但对于本例,单选按钮就足够了。

下面是计算折扣的代码:



```
private void button1_Click(object sender, RoutedEventArgs e)
{
    string scriptToUse;
    if (CostRadioButton.IsChecked.Value)
    {
        scriptToUse = "AmountDisc.py";
    }
    else
    {
        scriptToUse = "CountDisc.py";
    }
    ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
    ScriptEngine pythEng = scriptRuntime.GetEngine("Python");
    ScriptSource source = pythEng.CreateScriptSourceFromFile(scriptToUse);
    ScriptScope scope = pythEng.CreateScope();
    scope.SetVariable("prodCount", Convert.ToInt32(totalItems.Text));
    scope.SetVariable("amt", Convert.ToDecimal(totalAmt.Text));
    source.Execute(scope);
    label5.Content = scope.GetVariable("retAmt").ToString();
}
```

代码段 Window1.xaml.cs

第一部分仅确定要应用折扣的脚本 AmountDisc.py 或 CountDisc.py。AmountDisc.py 根据购买的金额计算折扣。

```
discAmt = .25
retAmt = amt
if amt > 25.00:
    retAmt = amt - (amt * discAmt)
```

能打折的最低购买金额是\$25.00。如果购买金额小于这个值，就不计算折扣，否则就使用 2.5% 的折扣率。

CountDisc.py 根据购买的物品数量计算折扣：

```
discCount = 5
discAmt = .1
retAmt = amt
if prodCount > discCount:
    retAmt = amt-(amt * discAmt)
```

在这个 Python 脚本中，购买的物品数量必须大于 5，才能给总价应用 10% 的折扣率。

下一部分是启动 ScriptRuntime 环境。这需要执行 4 个特定的步骤：创建 ScriptRuntime 对象、设置合适的 ScriptEngine 和创建 ScriptSource，以及创建 ScriptScope。

ScriptRuntime 对象是起点，也是包含 ScriptRuntime 的基础。它拥有包含环境的全局状态。ScriptRuntime 对象使用 CreateFromConfiguration() 静态方法创建。app.config 如下所示：

```
<configuration>
  <configSections>
    <section
      name="microsoft.scripting"
      type="Microsoft.Scripting.Hosting.Configuration.Section,
        Microsoft.Scripting,
        Version=0.9.6.10,
        Culture=neutral,
        PublicKeyToken=null"
      requirePermission="false" />
    </configSections>

    <microsoft.scripting>
      <languages>
        <language
          names="IronPython;Python;py"
          extensions=".py"
          displayName="IronPython 2.6 Alpha"
          type="IronPython.Runtime.PythonContext,
            IronPython,
            Version=2.6.0.1,
            Culture=neutral,
            PublicKeyToken=null" />
        </language>
      </languages>
    </microsoft.scripting>
  </configuration>
```

这段代码定义了“microsoft.scripting”的一部分，设置了 IronPython 语言引擎的几个属性。

接着，从 ScriptRuntime 中获取一个对 ScriptEngine 的引用。在本例中，指定需要 Python 引擎，但 ScriptRuntime 可以自己确定这一点，因为脚本的扩展名是 py。

ScriptEngine 完成了执行脚本代码的工作。执行文件或代码段中的脚本有几种方法。ScriptEngine 还提供了 ScriptSource 和 ScriptScope。

ScriptSource 对象允许访问脚本，它表示脚本的源代码。有了它，就可以操作脚本的源代码。从磁盘上加载它，逐行解析它，甚至把脚本编译到 CompiledCode 对象中。如果多次执行同一个脚本，

这就很方便。

`ScriptScope` 对象实际上是一个名称空间。要给脚本传入值或从脚本传出值, 应把一个变量绑定到 `ScriptScope` 上。本例调用 `SetVariable` 方法给 Python 脚本传入 `prodCount` 变量和 `amt` 变量。它们是 `totalItems` 文本框和 `totalAmt` 文本框中的值。计算出来的折扣使用 `GetVariable()` 方法从脚本中检索。在本例中, `retAmt` 变量包含了我们需要的值。

在 `CalcTax` 按钮中, 调用了 Python 对象上的方法。`CalcTax.py` 脚本是一个非常简单的方法, 它接受一个输入值, 加上 7.5% 的税, 再返回新值。代码如下:

```
def CalcTax(amount):
    return amount * 1.075
```

下面是调用 `CalcTax()` 方法的 C# 代码:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
    dynamic calcRate = scriptRuntime.UseFile("CalcTax.py");
    label6.Content = calcRate.CalcTax(Convert.ToDecimal(label5.Content)).ToString();
}
```

这是一个非常简单的过程。这里再次使用与前面相同的配置设置创建了 `ScriptRuntime` 对象。`calcRate` 是一个 `ScriptScope` 对象, 它定义为动态对象, 以便轻松地调用 `CalcTax()` 方法。这是使用动态类型简化编程工作的一个示例。

12.4 DynamicObject 和 ExpandoObject

如果要创建自己的动态对象, 该怎么办? 这有两种方法: 从 `DynamicObject` 中派生, 或者使用 `ExpandoObject`。使用 `DynamicObject` 需要做的工作较多, 因为必须重写几个方法。`ExpandoObject` 是一个可立即使用的密封类。

12.4.1 DynamicObject

考虑一个表示人的对象。一般应定义名字、中间名和姓氏等属性。现在假定要在运行期间构建这个对象, 且系统事先不知道该对象有什么属性或该对象可能支持什么方法。此时就可以使用基于 `DynamicObject` 的对象。需要这类功能的场合几乎没有, 但到目前为止, C# 语言还没有提供该功能。

先看看 `DynamicObject`:



可从
wrox.com
下载源代码

```
class WroxDynamicObject : DynamicObject
{
    Dictionary<string, object> _dynamicData = new Dictionary<string, object>();
    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        bool success = false;
        result = null;
        if (_dynamicData.ContainsKey(binder.Name))
        {
```

```

        result = _dynamicData[binder.Name];
        success = true;
    }
    else
    {
        result = "Property Not Found!";
        success = false;
    }
    return success;
}

public override bool TrySetMember(SetMemberBinder binder, object value)
{
    _dynamicData[binder.Name] = value;
    return true;
}

public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    dynamic method = _dynamicData[binder.Name];
    result = method((DateTime)args[0]);
    return result != null;
}
}

```

代码段 Dynamic\Program.cs

在这个示例中，重写了 3 个方法 TrySetMember()、TryGetMember() 和 TryInvokeMember()。

TrySetMember() 方法给对象添加了新方法、属性或字段。本例把成员信息存储在一个 Dictionary 对象中。传送给 TrySetMember() 方法的 SetMemberBinder 对象包含 Name 属性，它用于标识 Dictionary 中的元素。

TryGetMember() 方法根据 GetMemberBinder 对象的 Name 属性检索存储在 Dictionary 中的对象。这些方法如何使用？下面的代码使用了刚才新建的动态对象：

```

dynamic wroxDyn = new WroxDynamicObject();
wroxDyn.FirstName = "Bugs";
wroxDyn.LastName = "Bunny";
Console.WriteLine(wroxDyn.GetType());
Console.WriteLine("{0} {1}", wroxDyn.FirstName, wroxDyn.LastName);

```

看起来很简单，但在哪里调用了重写的方法？正是 .NET Framework 帮助完成了调用。DynamicObject 处理了绑定，我们只需引用 FirstName 和 LastName 属性即可，就好像它们一直存在一样。

如何添加方法？这很简单。可以使用上例中的 WroxDynamicObject，给它添加 GetTomorrowDate() 方法，该方法接受一个 DateTime 对象为参数，返回表示第二天的日期字符串。代码如下：

```

dynamic wroxDyn = new WroxDynamicObject();
Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();
wroxDyn.GetTomorrowDate = GetTomorrow;
Console.WriteLine("Tomorrow is {0}", wroxDyn.GetTomorrowDate(DateTime.Now));

```

这段代码使用 `Func<T, TResult>` 创建了委托 `GetTomorrow`。该委托表示的方法调用了 `AddDays`，给传入的 `Date` 加上一天，返回得到的日期字符串。接着把委托设置为 `wroxDyn` 对象上的 `GetTomorrowDate()` 方法。最后一行调用新方法，并传递今天的日期。

动态功能再次发挥了作用，对象上有了一个有效的方法。

12.4.2 ExpandoObject

`ExpandoObject` 的工作方式类似于上一节创建的 `WroxDynamicObject`，区别是不必重写方法，如下面的代码示例所示：

```
static void DoExpando()
{
    dynamic expObj = new ExpandoObject();
    expObj.FirstName = "Daffy";
    expObj.LastName = "Duck";
    Console.WriteLine(expObj.FirstName + " " + expObj.LastName);
    Func<DateTime, string>GetTomorrow = today => today.AddDays(1).ToShortDateString();
    expObj.GetTomorrowDate = GetTomorrow;
    Console.WriteLine("Tomorrow is {0}", expObj.GetTomorrowDate(DateTime.Now));

    expObj.Friends = new List<Person>();
    expObj.Friends.Add(new Person() { FirstName = "Bob", LastName = "Jones" });
    expObj.Friends.Add(new Person() { FirstName = "Robert", LastName = "Jones" });
    expObj.Friends.Add(new Person() { FirstName = "Bobby", LastName = "Jones" });

    foreach (Person friend in expObj.Friends)
    {
        Console.WriteLine(friend.FirstName + " " + friend.LastName);
    }
}
```

注意这段代码与前面的代码几乎完全相同，也添加了 `FirstName` 和 `LastName` 属性，以及 `GetTomorrow` 函数，但它还多做了一件事——把一个 `Person` 对象集合添加为对象的一个属性。

初看起来，这似乎与使用 `dynamic` 类型没有区别。但其中有两个微妙的区别非常重要。第一，不能仅创建 `dynamic` 类型的空对象。必须把 `dynamic` 类型赋予某个对象，例如，下面的代码是无效的：

```
dynamic dynObj;
dynObj.FirstName = "Joe";
```

与前面的示例一样，此时可以使用 `ExpandoObject`。

第二，因为 `dynamic` 类型必须赋予某个对象，所以如果执行 `GetType` 调用，它就会报告赋予了 `dynamic` 类型的对象类型。所以如果把它赋予 `int`，`GetType` 就报告它是一个 `int`。这不适用于 `ExpandoObject` 或派生自 `DynamicObject` 的对象。

如果需要控制动态对象中属性的添加和访问，则使该对象派生自 `DynamicObject` 是最佳选择。使用 `DynamicObject`，可以重写几个方法，准确地控制对象与运行库的交互方式。而对于其他情况，就应使用 `dynamic` 类型或 `ExpandoObject`。

12.5 小结

本章介绍了如何使用新的 `dynamic` 类型。还讨论了编译器在遇到 `dynamic` 类型时会做什么。我们探讨了 `Dynamic Language Runtime`, 可以把它包含在简单的应用程序中。并通过 `Python` 使用 `DLR`, 执行 `Python` 脚本, 传入传出脚本要使用的值。最后, 通过从 `DynamicObject` 派生一个类, 创建了自己的动态类型。

动态开发越来越流行, 它允许执行在静态的类型化语言中很难实现的操作。`dynamic` 类型和 `DLR` 允许 `C#` 程序员使用某些动态功能。

第 13 章

内存管理和指针

本章内容:

- 运行库在栈和堆上分配空间
- 垃圾回收
- 使用析构函数和 `System.IDisposable` 接口来释放非托管的资源
- C#中使用指针的语法
- 使用指针实现基于栈的高性能数组

本章介绍内存管理和内存访问的各个方面。尽管运行库负责为程序员处理大部分内存管理工作，但程序员仍必须理解内存管理的工作原理，了解如何高效地处理非托管的资源。

如果很好地理解了内存管理和 C#提供的指针功能，也就能很好地集成 C#代码和原来的代码，并能在非常注重性能的系统高效地处理内存。

13.1 后台内存管理

C#编程的一个优点是程序员不需要担心具体的内存管理，垃圾回收器会自动处理所有的内存清理工作。用户可以得到像 C++语言那样的效率，而不需要考虑像在 C++中那样内存管理工作的复杂性。虽然不必手动管理内存，但仍需理解后台发生的事情。理解程序在后台如何处理内存有助于提高应用程序的速度和性能。本节要介绍给变量分配内存时在计算机的内存中发生的情况。



本节不详细介绍许多主题的相关内容。您应把这一节看作是一般过程的简化向导，而不是实现的确切说明。

13.1.1 值数据类型

Windows 使用一个系统：虚拟寻址系统，该系统把程序可用的内存地址映射到硬件内存中的实际地址上，这些任务完全由 Windows 在后台管理。其实际结果是 32 位处理器上的每个进程都可以使用 4GB 的内存——无论计算机上实际有多少硬盘空间(在 64 位处理器上，这个数字会更大)。这个 4GB 的内存实际上包含了程序的所有部分，包括可执行代码、代码加载的所有 DLL，以及程序运行时使用的所有变量的内容。这个 4GB 的内存称为虚拟地址空间，或虚拟内存。为了方便起见，本章将

它简称为内存。

4GB 中的每个存储单元都是从 0 开始往上排序的。要访问存储在内存的某个空间中的一个值，就需要提供表示该存储单元的数字。在任何复杂的高级语言中，如 C#、VB、C++ 和 Java，编译器负责把人们可以理解的变量名转换为处理器可以理解的内存地址。

在进程的虚拟内存中，有一个区域称为栈。栈存储不是对象成员的值数据类型。另外，在调用一个方法时，也使用栈存储传递给方法的所有参数的副本。为了理解栈的工作原理，需要注意在 C# 中变量的作用域。如果变量 a 在变量 b 之前进入作用域，b 就会首先超出作用域。下面的代码：

```
{
    int a;
    // do something
    {
        int b;
        // do something else
    }
}
```

首先声明 a。在内部代码块中声明了 b。然后内部代码块终止，b 就超出作用域，最后 a 超出作用域。所以 b 的生存期完全包含在 a 的生存期中。在释放变量时，其顺序总是与给它们分配内存的顺序相反，这就是栈的工作方式。

还要注意，b 在另一个代码块中(通过另一对嵌套的花括号来定义)。因此，它包含在另一个作用域中。这称为块作用域或结构作用域。

我们不知道栈具体在地址空间的什么地方，这些信息在进行 C# 开发是不需要知道的。栈指针(操作系统维护的一个变量)表示栈中下一个空闲存储单元的地址。程序第一次开始运行时，栈指针指向为栈保留的内存块末尾。栈实际上是向下填充的，即从高内存地址向低内存地址填充。当数据入栈后，栈指针就会随之调整，以始终指向下一个空闲存储单元。这种情况如图 13-1 所示。在该图中，显示了栈指针 800 000(十六进制的 0xC3500)，下一个空闲存储单元是地址 799 999。

下面的代码会告诉编译器，需要一些存储空间以存储一个整数和一个双精度浮点数，这些存储单元分别称为 nRacingCars 和 engineSize，声明每个变量的代码行表示开始请求访问这个变量，在其中声明变量闭合花括号中的代码块标识这两个变量超出作用域的地方。

```
{
    int nRacingCars = 10;
    double engineSize = 3000.0;
    // do calculations;
}
```

假定使用如图 13-1 所示的栈。nRacingCars 变量进入作用域，赋值为 10，这个值放在存储单元 799 996~799 999 上，这 4 个字节就在栈指针所指空间的下面。有 4 个字节是因为存储 int 要使用 4 个字节。为了容纳该 int，应从栈指针对应的值中减去 4，所以它现在指向位置 799 996，即下一个空闲单元(799 995)。

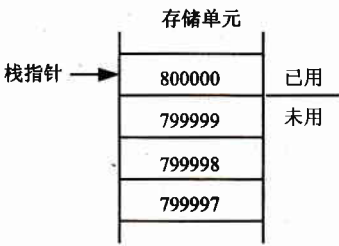


图 13-1

下一行代码声明变量 `engineSize`(这是一个 `double`)，把它初始化为 3 000.0。一个 `double` 数要占用 8 个字节，所以值 3 000.0 放在栈上的存储单元 799 988~799 995 上，栈指针对应的值减去 8，再次指向栈上的下一个空闲单元。

当 `engineSize` 超出作用域时，计算机就知道不再需要这个变量了。因为变量的生存期总是嵌套的，当 `engineSize` 在作用域中时，无论发生什么情况，都可以保证栈指针总是会指向存储 `engineSize` 的空间。为了从内存中删除这个变量，应给栈指针对应的值递增 8，现在它指向 `engineSize` 末尾紧接着的空间。此处就是放置闭合花括号的地方。当 `nRacingCars` 也超出作用域时，栈指针对应的值就再次递增 4。从栈中删除 `engineSize` 和 `nRacingCars` 之后，此时如果在作用域中又放入另一个变量，从 799 999 开始的存储单元就会被覆盖，这些空间以前是存储 `nRacingCars` 的。

如果编译器遇到 `int i, j` 这样的代码行，则这两个变量进入作用域的顺序是不确定的。两个变量是同时声明的，也是同时超出作用域的。此时，变量以什么顺序从内存中删除就不重要了。编译器在内部会确保先放在内存中的那个变量后删除，这样就能保证该规则不会与变量的生存期冲突。

13.1.2 引用数据类型

尽管栈有非常高的性能，但对于所有的变量它还是不太灵活。变量的生存期必须嵌套，在许多情况下，这种要求都过于苛刻。通常我们希望使用一个方法分配内存，来存储一些数据，并在方法退出后的很长一段时间内数据仍是可用的。只要是用 `new` 运算符来请求分配存储空间，就存在这种可能性——例如对于所有的引用类型。此时就要使用托管堆。

如果读者以前编写过需要管理低级内存的 C++ 代码，就会很熟悉堆(heap)。托管堆和 C++ 使用的堆不同，它在垃圾回收器的控制下工作，与传统的堆相比有很显著的优势。

托管堆(简称为堆)是处理器的可用 4GB 中的另一个内存区域。要了解堆的工作原理和如何为引用数据类型分配内存，看看下面的代码：

```
void DoWork()
{
    Customer arabel;
    arabel = new Customer();
    Customer otherCustomer2 = new EnhancedCustomer();
}
```

在这段代码中，假定存在两个类 `Customer` 和 `EnhancedCustomer`。`EnhancedCustomer` 类扩展了 `Customer` 类。

首先，声明一个 `Customer` 引用 `arabel`，在栈上给这个引用分配存储空间，但这仅是一个引用，而不是实际的 `Customer` 对象。`arabel` 引用占用 4 个字节的空间，足够包含在其中存储 `Customer` 对象的地址(需要 4 个字节把 0~4GB 之间的内存地址表示为一个整数值)。

· 然后看下一行代码：

```
arabel = new Customer();
```

这行代码完成了以下操作：首先，它分配堆上的内存，以存储 `Customer` 对象(一个真正的对象，不只是一个地址)。然后把变量 `arabel` 的值设置为分配给新 `Customer` 对象的内存地址(它还调用合适的 `Customer()` 构造函数初始化类实例中的字段，但此处我们不必担心这部分)。

`Customer` 实例没有放在栈中，而是放在堆中。在这个例子中，现在还不知道一个 `Customer` 对

象占用多少字节,但为了讨论方便,假定是 32 个字节。这 32 个字节包含了 `Customer` 实例的字段,和 .NET 用于识别和管理其类实例的一些信息。

为了在堆上找到存储新 `Customer` 对象的一个存储位置, .NET 运行库在堆中搜索,选取第一个未使用的且包含 32 个字节的连续块。为了讨论方便,假定其地址是 200 000, `arabel` 引用占用栈中的 799 996~799 999 位置。这表示在实例化 `arabel` 对象前,内存的内容应如图 13-2 所示。

给 `Customer` 对象分配空间后,内存的内容应如图 13-3 所示。注意,与栈不同,堆上的内存是向上分配的,所以空闲空间在已用空间的上面。



图 13-2

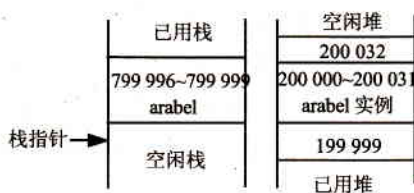


图 13-3

下一行代码声明了一个 `Customer` 引用,并实例化一个 `Customer` 对象。在这个例子中,需要在栈上为 `otherCustomer2` 引用分配空间,同时,也需要在堆上的一行单码中为 `mrJones` 对象分配空间:

```
Customer otherCustomer2 = new EnhancedCustomer();
```

该行把栈上的 4 个字节分配给 `otherCustomer2` 引用,它存储在 799 992~799 995 位置上,而 `otherCustomer2` 对象在堆上从 200 032 开始向上分配空间。

从这个例子可以看出,建立引用变量的过程要比建立值变量的过程更复杂,且不能避免性能的系统开销。实际上,我们对这个过程进行了过分的简化,因为 .NET 运行库需要保存堆的状态信息,在堆中添加新数据时,这些信息也需要更新。尽管有这些性能开销,但仍有一种机制,在给变量分配内存时,不会受到栈的限制。把一个引用变量的值赋予另一个相同类型的变量,就有两个引用内存中同一对象的变量了。当一个引用变量超出作用域时,它会从栈中删除,如上一节所述,但引用对象的数据仍保留在堆中,一直到程序终止,或垃圾回收器删除它为止,而只有在该数据不再被任何变量引用时,它才会被删除。

这就是引用数据类型的强大之处,在 C# 代码中广泛使用了这个特性。这说明,我们可以对数据的生存期进行非常强大的控制,因为只要保持对数据的引用,该数据就肯定存在于堆上。

13.1.3 垃圾回收

由上面的讨论和图 13-2 和图 13-3 可以看出,托管堆的工作方式非常类似于栈,在某种程度上,对象会在内存中一个挨一个地放置,这样就很容易使用指向下一个空闲存储单元的堆指针,来确定下一个对象的位置。在堆上添加更多的对象时,也容易调整。但这比较复杂,因为基于堆的对象的生存期与引用它们的基于栈的变量的作用域不匹配。

在垃圾回收器运行时,它会从堆中删除不再引用的所有对象。在完成删除操作后,堆会立即把对象分散开来,与已经释放的内存混合在一起,如图 13-4 所示。

如果托管的堆也是这样，在其上给新对象分配内存就成为一个很难处理的过程，运行库必须搜索整个堆，才能找到足够大的内存块来存储每个新对象。但是，垃圾回收器不会让堆处于这种状态。只要它释放了能释放的所有对象，就会把其他对象移动回堆的端部，再次形成一个连续的内存块。因此，堆可以继续像栈那样确定在什么地方存储新对象。当然，在移动对象时，这些对象的所有引用都需要用正确的新地址来更新，但垃圾回收器也会处理更新问题。

垃圾回收器的这个压缩操作是托管的堆与非托管的旧堆的区别所在。使用托管的堆，就只需要读取堆指针的值即可，而不需遍历地址的链表，来查找一个地方来放置新数据。因此，在.NET 下实例化对象要快得多。有趣的是，访问它们也比较快，因为对象会压缩到堆上相同的内存区域，这样需要交换的页面较少。Microsoft 相信，尽管垃圾回收器需要做一些工作，压缩堆，修改它移动的所有对象引用，致使性能降低，但这些性能会得到更多弥补。



图 13-4



一般情况下，垃圾收集器在.NET 运行库认为需要它时运行。可以调用 `System.GC.Collect()` 方法，强迫垃圾回收器在代码的某个地方运行，`System.GC` 类是一个表示垃圾回收器的.NET 类，`Collect()` 方法初始化垃圾回收器。但是，GC 类适用的场合很少，例如，代码中有大量的对象刚刚取消引用，就适合调用垃圾收集器。但是，垃圾回收器的逻辑不能保证在一次垃圾收集过程中，所有未引用的对象都从堆中删除。

垃圾回收器运行时，它实际上会降低应用程序的性能，因为在垃圾回收器完成其任务之前，应用程序不可能继续运行。使用.NET 垃圾回收器可以减小这个问题的影响，因为它是一个世代 (generational) 垃圾回收器。

创建对象时，会把这些对象放在托管堆上。堆的第一部分称为第 0 代。创建新对象时，会把它们移动到堆的这个部分中。因此，这里驻留了最新的对象。

对象会继续放在这个部分，直到第一个对象集合在垃圾回收的过程中生成。这个清理过程之后仍保留的对象会被压缩，然后移动到堆的下一部分上或世代部分——第 1 代对应的部分。

此时，第 0 代对应的部分为空，所有的新对象都再次放在这一部分上。在垃圾回收过程中遗留下来的旧对象放在第 1 代对应的部分上。老对象的这种移动会再次发生。接着重复下一次回收过程。这意味着，第 1 代中在垃圾回收过程中遗留下来的对象会移动到堆的第 2 代，位于第 0 代的对象会移动到第 1 代，第 0 代仍用于放置新对象。



有趣的是，在给对象分配内存空间时，如果超出了第 0 代对应的部分的容量，或者调用了 `GC.Collect()` 方法，就会进行垃圾回收。

这个过程极大地提高了应用程序的性能。一般而言，最新的对象通常是可以回收的对象，而且

可能也会回收大量比较新的对象。如果这些对象在堆中的位置是相邻的,垃圾回收过程就会更快。另外,相关的对象相邻放置也会使程序执行得更快。

在.NET中,垃圾回收提高性能的另一个领域是架构处理堆上较大对象的方式。在.NET下,较大对象有自己的托管堆,称为大对象堆。使用大于 85 000 个字节的对象时,它们就会放在这个特殊的堆上,而不是主堆上。.NET 应用程序不知道两者的区别,因为这是自动完成的。其原因是在堆上压缩大对象是比较昂贵的,因此驻留在大对象堆上的对象不执行压缩过程。

13.2 释放非托管的资源

垃圾回收器的出现意味着,通常不需要担心不再需要的对象,只要让这些对象的所有引用都超出作用域,并允许垃圾回收器在需要时释放内存即可。但是,垃圾回收器不知道如何释放非托管的资源(例如文件句柄、网络连接和数据库连接)。托管类在封装对非托管资源的直接或间接引用时,需要制定专门的规则,确保非托管的资源在回收类的一个实例时释放。

在定义一个类时,可以使用两种机制来自动释放非托管的资源。这些机制常常放在一起实现,因为每种机制都为问题提供了略有不同的解决方法。这两种机制是:

- 声明一个析构函数(或终结器),作为类的一个成员
- 在类中实现 `System.IDisposable` 接口

下面依次讨论这两种机制,然后介绍如何同时实现它们,以获得最佳的效果。

13.2.1 析构函数

前面介绍了构造函数可以指定必须在创建类的实例时进行的某些操作。相反,在垃圾回收器销毁对象之前,也可以调用析构函数。由于执行这个操作,因此析构函数初看起来似乎是放置释放非托管资源、执行一般清理操作的代码的最佳地方。但是,事情并不是如此简单。



在讨论 C# 中的析构函数时,在底层的.NET 体系结构中,这些函数称为终结器 (finalizer)。在 C# 中定义析构函数时,编译器发送给程序集的实际上是 `Finalize()` 方法。它不会影响源代码,但如果需要查看程序集的内容,就应知道这个事实。

C++ 开发人员应很熟悉析构函数的语法。它看起来类似于一个方法,与包含的类同名,但有一个前缀波浪形符(~)。它没有返回类型,不带参数,没有访问修饰符。下面是一个例子:

```
class MyClass
{
    ~MyClass()
    {
        // destructor implementation
    }
}
```

C# 编译器在编译析构函数时,它会隐式地把析构函数的代码编译为等价于 `Finalize()` 方法的代码,从而确保执行父类的 `Finalize()` 方法。下面列出了等价于编译器为 `~MyClass()` 析构函数生成的 IL 的 C# 代码:

```
protected override void Finalize()
{
    try
    {
        // destructor implementation
    }
    finally
    {
        base.Finalize();
    }
}
```

如上所示, 在 `~MyClass()` 析构函数中实现的代码封装在 `Finalize()` 方法的一个 `try` 块中。对父类的 `Finalize()` 方法的调用放在 `finally` 块中, 确保该调用的执行。第 15 章会讨论 `try` 块和 `finally` 块。

有经验的 C++ 开发人员大量使用了析构函数, 有时不仅用于清理资源, 还提供调试信息或执行其他任务。C# 析构函数要比 C++ 析构函数的使用少得多。与 C++ 析构函数相比, C# 析构函数的问题是它们的不确定性。在销毁 C++ 对象时, 其析构函数会立即运行。但由于使用 C# 时垃圾回收器的工作方式, 无法确定 C# 对象的析构函数何时执行。所以, 不能在析构函数中放置需要在某一时刻运行的代码, 也不应使用能以任意顺序对不同类的实例调用的析构函数。如果对象占用了宝贵而重要的资源, 应尽快释放这些资源, 此时就不能等待垃圾回收器来释放了。

另一个问题是 C# 析构函数的实现会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾回收器的一次处理中从内存中删除, 但有析构函数的对象需要两次处理才能销毁: 第一次调用析构函数时, 没有删除对象, 第二次调用才真正删除对象。另外, 运行库使用一个线程来执行所有对象的 `Finalize()` 方法。如果频繁使用析构函数, 而且使用它们执行长时间的清理任务, 对性能的影响就会非常显著。

13.2.2 IDisposable 接口

在 C# 中, 推荐使用 `System.IDisposable` 接口替代析构函数。`IDisposable` 接口定义了一种模式(具有语言级的支持), 该模式为释放非托管的资源提供了确定的机制, 并避免产生析构函数固有的与垃圾回收器相关的问题。`IDisposable` 接口声明了一个 `Dispose()` 方法, 它不带参数, 返回 `void`, `MyClass` 类的 `Dispose()` 方法的实现代码如下:

```
class MyClass: IDisposable
{
    public void Dispose()
    {
        // implementation
    }
}
```

`Dispose()` 方法的实现代码显式地释放由对象直接使用的所有非托管资源, 并在所有也实现 `IDisposable` 接口的封装对象上调用 `Dispose()` 方法。这样, `Dispose()` 方法为何时释放非托管资源提供了精确的控制。

假定有一个 `ResourceGobbler` 类, 它依赖于使用某些外部资源, 且实现 `IDisposable` 接口。如果要实例化这个类的实例, 使用它, 然后释放它, 就可以使用下面的代码:

```
ResourceGobbler theInstance = new ResourceGobbler();

// do your processing

theInstance.Dispose();
```

但是，如果在处理过程中出现异常，这段代码就没有释放 `theInstance` 使用的资源，所以应使用 `try` 块(详见第 15 章)，编写下面的代码：

```
ResourceGobbler theInstance = null;

try
{
    theInstance = new ResourceGobbler();
    // do your processing
}
finally
{
    if (theInstance != null)
    {
        theInstance.Dispose();
    }
}
```

即使在处理过程中出现了异常，这个版本也可以确保总是在 `theInstance` 上调用 `Dispose()` 方法，总是释放 `theInstance` 使用的任意资源。但是，如果总是要重复这样的结构，代码就很容易被混淆。C# 提供了一种语法，可以确保在实现 `IDisposable` 接口的对象的引用超出作用域时，在该对象上自动调用 `Dispose()` 方法。该语法使用了 `using` 关键字来完成这一工作——该关键字在完全不同的环境下，它与名称空间没有关系。下面的代码生成与 `try` 块等价的 IL 代码：

```
using (ResourceGobbler theInstance = new ResourceGobbler())
{
    // do your processing
}
```

`using` 语句的后面是一对圆括号，其中是引用变量的声明和实例化，该语句使变量放在随后的语句块中。另外，在变量超出作用域时，即使出现异常，也会自动调用其 `Dispose()` 方法。然而，如果已经使用 `try` 块来捕获其他异常，就会比较清晰，如果避免使用 `using` 语句，仅在已有的 `try` 块的 `finally` 子句中调用 `Dispose()` 方法，还可以避免进行额外的代码缩进。



对于某些类，使用 `Close()` 方法要比 `Dispose()` 方法更富有逻辑性，例如，在处理文件或数据库连接时就是这样。在这些情况下，常常实现 `IDisposable` 接口，再实现一个独立的 `Close()` 方法，`Close()` 方法只调用 `Dispose()` 方法。这种方法在类的使用上比较清晰，还支持 C# 提供的 `using` 语句。

13.2.3 实现 `IDisposable` 接口和析构函数

前面的章节讨论了类所使用的释放非托管资源的两种方式：

从上述代码可以看出, `Dispose()` 方法有第二个 `protected` 重载方法, 它带一个布尔参数, 这是真正完成清理工作的方法。`Dispose(bool)` 方法由析构函数和 `IDisposable.Dispose()` 方法调用。这种方式的重点是确保所有的清理代码都放在一个地方。

传递给 `Dispose(bool)` 方法的参数表示 `Dispose(bool)` 方法是由析构函数调用, 还是由 `IDisposable.Dispose()` 方法调用——`Dispose(bool)` 方法不应从代码的其他地方调用, 其原因是:

- 如果使用者调用 `IDisposable.Dispose()` 方法, 该使用者就指定应清理所有与该对象相关的资源, 包括托管和非托管的资源。
- 如果调用了析构函数, 原则上所有的资源仍需要清理。但是在这种情况下, 析构函数必须由垃圾回收器调用, 而且用户不应试图访问其他托管的对象, 因为我们不再能确定它们的状态了。在这种情况下, 最好清理已知的非托管资源, 希望任何引用的托管对象还有析构函数, 这些析构函数执行自己的清理过程。

`isDisposed` 成员变量表示对象是否已被清理, 并确保不试图多次清理成员变量。它还允许在执行实例方法之前测试对象是否已清理, 如 `SomeMethod()` 方法所示。这个简单的方法不是线程安全的, 需要调用者确保在同一时刻只有一个线程调用方法。要求使用者进行同步是一个合理的假定, 在整个 .NET 类库中(例如, 在集合类中)反复使用了这个假定。第 20 章将讨论线程和同步。

最后, `IDisposable.Dispose()` 方法包含一个对 `System.GC.SuppressFinalize()` 方法的调用。`GC` 类表示垃圾回收器, `SuppressFinalize()` 方法则告诉垃圾回收器有一个类不再需要调用其析构函数了。因为 `Dispose()` 方法已经完成了所有需要的清理工作, 所以析构函数不需要做任何工作。调用 `SuppressFinalize()` 方法就意味着垃圾回收器认为这个对象根本没有析构函数。

13.3 不安全的代码

如前面的章节所述, C# 非常擅长于对开发人员隐藏大部分基本内存管理, 因为它使用了垃圾回收器和引用。但是, 有时需要直接访问内存。例如, 由于性能问题, 要在外部(非 .NET 环境)的 DLL 中访问一个函数, 该函数需要把一个指针当作参数来传递(许多 Windows API 函数就是这样)。本节将论述 C# 直接访问内存的内容的功能。

13.3.1 用指针直接访问内存

下面把指针当作一个新论题来介绍, 而实际上, 指针并不是新东西。因为在代码中可以自由使用引用, 而引用就是一个类型安全的指针。前面已经介绍了表示对象和数组的变量实际上存储相应数据(引用)的内存地址。指针只是一个以与引用相同的方式存储地址的变量。其区别是 C# 不允许直接访问在引用变量中包含的地址。有了引用后, 从语法上看, 变量就可以存储引用的实际内容。

C# 引用主要用于使 C# 语言易于使用, 防止用户无意中执行某些破坏内存中内容的操作。另一方面, 使用指针, 就可以访问实际内存地址, 执行新类型的操作。例如, 给地址加上 4 个字节, 就可以查看甚至修改存储在新地址中的数据。

下面是使用指针的两个主要原因:

- **向后兼容性**——尽管 .NET 运行库提供了许多工具，但仍可以调用本地的 Windows API 函数。对于某些操作这可能是完成任务的唯一方式。这些 API 函数都是用 C 语言编写的，通常要求把指针作为其参数。但在许多情况下，还可以使用 `DllImport` 声明，以避免使用指针，例如，使用 `System.IntPtr` 类。
- **性能**——在一些情况下，速度是最重要的，而指针可以提供最优性能。假定用户知道自己在做什么，就可以确保以最高效的方式访问或处理数据。但是，注意在代码的其他区域中，不使用指针，也可以对性能进行必要的改进。请使用代码配置文件，查找代码中的瓶颈，代码配置文件随 VS 一起安装。

但是，这种低级的内存访问也是有代价的。使用指针的语法比引用类型的语法更复杂。而且，指针使用起来比较困难，需要非常高的编程技巧和很强的能力，仔细考虑代码所完成的逻辑操作，才能成功地使用指针。如果不仔细，使用指针就很容易在程序中引入细微的、难以查找的错误。例如，很容易重写其他变量，导致栈溢出，访问某些没有存储变量的内存区域，甚至重写 .NET 运行库所需要的代码信息，因而使程序崩溃。

另外，如果使用指针，就必须授予代码运行库的代码访问安全机制的高级别信任，否则就不能执行它。在默认的代码访问安全策略中，只有代码运行在本地计算机上，这才是可能的。如果代码必须运行在远程地点，如 Internet，用户就必须给代码授予额外的许可，代码才能工作。除非用户信任您和代码，否则他们不会授予这些许可，第 21 章将讨论代码访问安全性。

尽管有这些问题，但指针在编写高效的代码时是一种非常强大和灵活的工具



这里强烈建议不要使用指针，因为如果使用指针，代码不仅难以编写和调试，而且无法通过 CLR 施加的内存类型安全检查(详见第 1 章)。

1. 用 `unsafe` 关键字编写不安全的代码

因为使用指针会带来相关的风险，所以 C# 只允许在特别标记的代码块中使用指针。标记代码所用的关键字是 `unsafe`。下面的代码把一个方法标记为 `unsafe`：

```
unsafe int GetSomeNumber()
{
    // code that can use pointers
}
```

任何方法都可以标记为 `unsafe`——无论该方法是否应用了其他修饰符(例如，静态方法、虚方法等)。在这种方法中，`unsafe` 修饰符还会应用到方法的参数上，允许把指针用作参数。还可以把整个类或结构标记为 `unsafe`，这表示假设所有的成员都是不安全的：

```
unsafe class MyClass
{
    // any method in this class can now use pointers
}
```

同样，可以把成员标记为 `unsafe`：

```
class MyClass
{
    unsafe int* pX; // declaration of a pointer field in a class
}
```

也可以把方法中的一块代码标记为 **unsafe**:

```
void MyMethod()
{
    // code that doesn't use pointers
    unsafe
    {
        // unsafe code that uses pointers here
    }
    // more 'safe' code that doesn't use pointers
}
```

但要注意, 不能把局部变量本身标记为 **unsafe**:

```
int MyMethod()
{
    unsafe int *pX; // WRONG
}
```

如果要使用不安全的局部变量, 就需要在不安全的方法或语句块中声明和使用它。在使用指针前还有一步要完成。C#编译器会拒绝不安全的代码, 除非告诉编译器代码包含不安全的代码块。标记所用的关键字是 **unsafe**。因此, 要编译包含不安全代码块的文件 **MySource.cs**(假定没有其他编译器选项), 就要使用下述命令:

```
csc /unsafe MySource.cs
```

或者

```
csc -unsafe MySource.cs
```



如果使用 Visual Studio 2005、2008 或 2010, 就可以在项目属性窗口的 **Build** 选项卡中找到编译不安全代码的选项。

2. 指针的语法

把代码块标记为 **unsafe** 后, 就可以使用下面的语法声明指针:

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

这段代码声明了 4 个变量, **pWidth** 和 **pHeight** 是整数指针, **pResult** 是 **double** 型指针, **pFlags** 是字节型的数组指针。我们常常在指针变量名的前面使用前缀 **p** 来表示这些变量是指针。在变量声明中, 符号 ***** 表示声明一个指针, 换言之, 就是存储特定类型的变量的地址。



C++开发人员应注意，C++与C#中的语法差异。C#语句中的“`int* pX, pY;`”对应于C++语句中的“`int *pX, *pY;`”。在C#中，*符号与类型相关，而与变量名无关。

声明了指针类型的变量后，就可以用与一般变量相同的方式使用它们，但首先需要学习另外两个运算符：

- `&`表示“取地址”，并把一个值数据类型转换为指针，例如，`int`转换为`*int`。这个运算符称为寻址运算符。
- `*`表示“获取地址的内容”，把一个指针转换为值数据类型(例如，`*float`转换为`float`)。这个运算符称为“间接寻址运算符”(有时称为“取消引用运算符”)。

从这些定义中可以看出，`&`和`*`的作用是相反的。



假设符号`&`和`*`也表示按位 AND(`&`)和乘法(`*`)运算符，那么如何以这种方式使用它们？答案是在实际使用时它们是不会混淆的，用户和编译器总是知道在什么情况下这两个符号有什么含义，因为按照新指针的定义，这些符号总是以一元运算符的形式出现——它们只作用于一个变量，并出现在代码中变量的前面。另一方面，按位 AND 和乘法运算符是二元运算符，它们需要两个操作数。

下面的代码说明了如何使用这些运算符：

```
int x = 10;
int* pX, pY;
pX = & x;
pY = pX;
*pY = 20;
```

首先声明一个整数 `x`，其值是 10。接着声明两个整数指针 `pX` 和 `pY`。然后把 `pX` 设置为指向 `x`(换言之，把 `pX` 的内容设置为 `x` 的地址)。然后把 `pX` 的值赋予 `pY`，所以 `pY` 也指向 `x`。最后，在语句 `*pY = 20` 中，把值 20 赋予 `pY` 指向的地址包含的内容。实际上是把 `x` 的内容改为 20，因为 `pY` 指向 `x`。注意在这里，变量 `pY` 和 `x` 之间没有任何关系。只是此时 `pY` 碰巧指向存储 `x` 的存储单元而已。

要进一步理解这个过程，假定 `x` 存储在栈的存储单元 `0x12F8C4~0x12F8C7` 中(十进制就是 124332~124335，即有 4 个存储单元，因为一个 `int` 占用 4 个字节)。因为栈向下分配内存，所以变量 `pX` 存储在 `0x12F8C0~0x12F8C3` 的位置上，`pY` 存储在 `0x12F8BC~0x12F8BF` 的位置上。注意，`pX` 和 `pY` 也分别占用 4 个字节。这不是因为一个 `int` 占用 4 个字节，而是因为在 32 位处理器上，需要用 4 个字节存储一个地址。利用这些地址，在执行完上述代码后，栈应如图 13-5 所示。

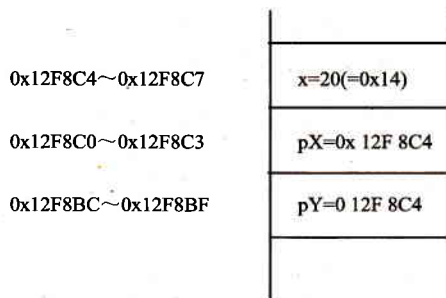


图 13-5



这个示例使用 `int` 来说明该过程，其中 `int` 存储在 32 位处理器中栈的连续空间上，但并不是所有的数据类型都会存储在连续的空间中。原因是 32 位处理器最擅长于在 4 个字节的内存块中检索数据。这种计算机上的内存会分解为 4 个字节的块，在 Windows 上，每个块有时称为 `DWORD`，因为这是 32 位无符号 `int` 数在 .NET 出现之前的名字。这是从内存中获取 `DWORD` 的最高效的方式——跨越 `DWORD` 边界存储数据通常会降低硬件的性能。因此，.NET 运行库通常会给某些数据类型填充一些空间，使它们占用的内存是 4 的倍数。例如，`short` 数据占用两个字节，但如果把一个 `short` 放在栈中，栈指针仍会向下移动 4 个字节，而不是两个字节，这样，下一个存储在栈中的变量就仍从 `DWORD` 的边界开始存储。

可以把指针声明为任意一种值类型——即任何预定义的类型 `uint`、`int` 和 `byte` 等，也可以声明为一个结构。但是不能把指针声明为一个类或数组，因为这么做会使垃圾回收器出现问题。为了正常工作，垃圾回收器需要知道在堆上创建了什么类的实例，它们在什么地方。但如果代码开始使用指针处理类，就很容易破坏堆中 .NET 运行库为垃圾回收器维护的与类相关的信息。在这里，垃圾回收器可以访问的任何数据类型称为托管类型，而指针只能声明为非托管类型，因为垃圾回收器不能处理它们。

3. 将指针强制转换为整数类型

由于指针实际上存储了一个表示地址的整数，因此任何指针中的地址都可以和任何整数类型之间相互转换。指针到整数类型的转换必须是显式指定的，隐式的转换是不允许的。例如，编写下面的代码是合法的：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

把指针 `pX` 中包含的地址强制转换为一个 `uint`，存储在变量 `y` 中。接着把 `y` 强制转换回一个 `int*`，存储在新变量 `pD` 中。因此 `pD` 也指向 `x` 的值。

把指针的值强制转换为整数类型的主要目的是显示它。虽然 `Console.WriteLine()` 和 `Console.WriteLine()` 方法没有带指针的重载方法，但是必须把强制指针的值转换为整数类型，这两个方法才能接受和显示它们：

```
Console.WriteLine("Address is " + pX); // wrong -- will give a
// compilation error
Console.WriteLine("Address is " + (uint)pX); // OK
```

可以把一个指针强制转换为任何整数类型，但是，因为在 32 位系统上，一个地址占用 4 个字节，把指针强制转换为除了 `uint`、`long` 或 `ulong` 之外的数据类型，肯定会导致溢出错误(`int` 数也可能导致这个问题，因为它的取值范围是 -20 亿~20 亿，而地址的取值范围是 0~40 亿)。C# 是用于 64 位

处理器的，一个地址占用 8 个字节。因此在这样的系统上，把指针强制转换为非 `ulong` 类型，就可能导致溢出错误。还要注意，`checked` 关键字不能用于涉及指针的转换。对于这种转换，即使在设置 `checked` 的情况下，发生溢出时也不会抛出异常。NET 运行库假定，如果使用指针，就知道自己要做什么，不必担心可能出现的溢出。

4. 指针类型之间的强制转换

也可以在指向不同类型的指针之间进行显式的转换。例如：

```
byte aByte = 8;
byte* pByte = &aByte;
double* pDouble = (double*)pByte;
```

这是一段合法的代码，但如果要执行这段代码，就要小心了。在上面的示例中，如果要查找指针 `pDouble` 指向的 `double` 值，就会查找包含 1 个字节(`aByte`)的内存，和一些其他内存，并把它当作包含一个 `double` 值的内存区域来对待——这不会得到一个有意义的值。但是，可以在类型之间转换，实现 `Union` 类型的等价形式，或者把指针强制转换为其他类型，例如把指针转换为 `sbyte`，检查内存的单个字节。

5. void 指针

如果要维护一个指针，但不希望指定它指向的数据类型，就可以把指针声明为 `void`：

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

`void` 指针的主要用途是调用需要 `void*` 参数的 API 函数。在 C# 语言中，使用 `void` 指针的情况并不是很多。特殊情况下，如果试图使用 `*` 运算符取消引用 `void` 指针，编译器就会标记一个错误。

6. 指针算术的运算

可以给指针加减整数。但是，编译器很智能，知道如何执行这个操作。例如，假定有一个 `int` 指针，要在其值上加 1。编译器会假定我们要查找 `int` 后面的存储单元，因此会给该值加上 4 个字节，即加上一个 `int` 占用的字节数。如果这是一个 `double` 指针，加 1 就表示在指针的值上加 8 个字节，即一个 `double` 占用的字节数。只有指针指向 `byte` 或 `sbyte`(都是 1 个字节)时，才会给该指针的值加上 1。

可以对指针使用运算符 `+`、`-`、`+=`、`-=`、`++` 和 `--`，这些运算符右边的变量必须是 `long` 或 `ulong` 类型。



不允许对 `void` 指针执行算术运算。

例如，假定有如下定义：

```
uint u = 3;
byte b = 8;
double d = 10.0;
```

```
uint * pUInt= & u;      // size of a uint is 4
byte * pByte = & b;     // size of a byte is 1
double * pDouble = & d; // size of a double is 8
```

下面假定这些指针指向的地址是:

- pUInt: 1243332
- pByte: 1243328
- pDouble: 1243320

执行这段代码后:

```
++pUInt;      // adds (1*4) = 4 bytes to pUInt
pByte -= 3;    // subtracts (3*1) = 3 bytes from pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

指针应包含的内容是:

- pUInt: 1243336
- pByte: 1243325
- pDouble2: 1243352



给类型为 T 的指针加上数值 X , 其中指针的值为 P , 则得到的结果是 $P + X * (\text{sizeof}(T))$ 。



使用这条规则时要小心。如果给定类型的连续值存储在连续的存储单元中, 指针加法就允许在存储单元之间移动指针。但如果类型是 `byte` 或 `char`, 其总字节数不是 4 的倍数, 连续值就不是默认地存储在连续的存储单元中。

如果两个指针都指向相同的数据类型, 则也可以把一个指针从另一个指针中减去。此时, 结果是一个 `long`, 其值是指针值的差被该数据类型所占用的字节数整除的结果:

```
double* pD1 = (double*)1243324; // note that it is perfectly valid to
                                // initialize a pointer like this.
double* pD2 = (double*)1243300;
long L = pD1 - pD2;              // gives the result 3 (=24/sizeof(double))
```

7. sizeof 运算符

这一节将介绍如何确定各种数据类型的大小。如果需要在代码中使用某种类型的大小, 就可以使用 `sizeof` 运算符, 它的参数是数据类型的名称, 返回该类型占用的字节数。例如:

```
int x = sizeof(double);
```

这将设置 x 的值为 8。

使用 `sizeof` 的优点是不必在代码中硬编码数据类型的大小, 使代码的移植性更强。对于预定义的数据类型, `sizeof` 返回表 13-1 所示的值。

表 13-1

<code>sizeof(sbyte) = 1;</code>	<code>sizeof(byte) = 1;</code>
<code>sizeof(short) = 2;</code>	<code>sizeof(ushort) = 2;</code>
<code>sizeof(int) = 4;</code>	<code>sizeof(uint) = 4;</code>
<code>sizeof(long) = 8;</code>	<code>sizeof(ulong) = 8;</code>
<code>sizeof(char) = 2;</code>	<code>sizeof(float) = 4;</code>
<code>sizeof(double) = 8;</code>	<code>sizeof(bool) = 1;</code>

也可以对自己定义的结构使用 `sizeof`，但此时得到的结果取决于结构中的字段类型。不能对类使用 `sizeof`。

8. 结构指针：指针成员访问运算符

结构指针的工作方式与预定义值类型的指针的工作方式完全相同。但是这有一个条件：结构不能包含任何引用类型，这是因为前面介绍的一个限制——指针不能指向任何引用类型。为了避免这种情况，如果创建一个指针，它指向包含任何引用类型的任何结构，编译器就会标记一个错误。

假定定义了如下结构：

```
struct MyStruct
{
    public long X;
    public float F;
}
```

就可以给它定义一个指针：

```
MyStruct* pStruct;
```

然后对其进行初始化：

```
MyStruct Struct = new MyStruct();
pStruct = & Struct;
```

也可以通过指针访问结构的成员值：

```
( * pStruct ).X = 4;
( * pStruct ).F = 3.4f;
```

但是，这个语法有点复杂。因此，C#定义了另一个运算符，用一种比较简单的语法，通过指针访问结构的成员，它称为指针成员访问运算符，其符号是一个短划线，后跟一个大于号，它看起来像一个箭头：`->`。



C++开发人员认识指针成员访问运算符。因为 C++ 使用这个符号完成相同的任务。

使用这个指针成员访问运算符，上述代码可以重写为：

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

也可以直接把合适类型的指针设置为指向结构中的一个字段：

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

或者

```
long* pL = &(pStruct-> X);
float* pF = &(pStruct-> F);
```

9. 类成员指针

前面说过，不能创建指向类的指针，这是因为垃圾回收器不维护关于指针的任何信息，只维护关于引用的信息，因此创建指向类的指针会使垃圾回收器不能正常工作。

但是，大多数类都包含值类型的成员，可以为这些值类型成员创建指针，但这需要一种特殊的语法。例如，假定把上面示例中的结构重写为类：

```
class MyClass
{
    public long X;
    public float F;
}
```

然后就可以为它的字段 X 和 F 创建指针了，方法与前面一样。但这么做会产生一个编译错误：

```
MyClass myObject = new MyClass();
long* pL = &(myObject.X); // wrong -- compilation error
float* pF = &(myObject.F); // wrong -- compilation error
```

尽管 X 和 F 都是非托管类型，但它们嵌入在一个对象中，这个对象存储在堆上。在垃圾回收的过程中，垃圾回收器会把 MyObject 移动到内存的一个新单元上，这样，pL 和 pF 就会指向错误的存储地址。由于存在这个问题，因此编译器不允许以这种方式把托管类型的成员的地址分配给指针。

解决这个问题的方法是使用 **fixed** 关键字，它会告诉垃圾回收器，可能有引用某些对象的成员的指针，所以这些实例不能移动。如果要声明一个指针，则使用 **fixed** 的语法如下所示：

```
MyClass myObject = new MyClass();
fixed (long* pObject = &(myObject.X))
{
    // do something
}
```

在关键字 **fixed** 后面的圆括号中，定义和初始化指针变量。这个指针变量(在本例中是 pObject)的作用域是花括号标识的 **fixed** 块。这样，垃圾回收器就知道，在执行 **fixed** 块中的代码时，不能移动 myObject 对象。

如果要声明多个这样的指针，可以在同一个代码块前放置多条 **fixed** 语句：

```

MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
fixed (float* pF = &(myObject.F))
{
    // do something
}

```

如果要在不同的阶段固定几个指针，就可以嵌套整个 **fixed** 块：

```

MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
{
    // do something with pX
    fixed (float* pF = &(myObject.F))
    {
        // do something else with pF
    }
}

```

如果这些变量的类型相同，就也可以在同一条 **fixed** 块中初始化多个变量：

```

MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X))
{
    // etc.
}

```

在上述情况中，是否声明不同的指针，让它们指向相同或不同对象中的字段，或者指向与类实例无关的静态字段，这一点并不重要。

13.3.2 指针示例: PointerPlayground

下面给出一个使用指针的示例: **PointerPlayground**。它执行一些简单的指针操作，显示结果，还允许查看内存中发生的情况，并确定变量存储在什么地方：



可从
wtoox.com
下载源代码

```

using System;

namespace PointerPlayground
{
    class MainEntryPoint
    {
        static unsafe void Main()
        {
            int x=10;
            short y = -1;
            byte y2 = 4;
            double z = 1.5;
            int* pX = &x;
            short* pY = &y;
            double* pZ = &z;

            Console.WriteLine(
                "Address of x is 0x{0:X}, size is {1}, value is {2}",
                (uint) &x, sizeof(int), x);
        }
    }
}

```

```

Console.WriteLine(
    "Address of y is 0x{0:X}, size is {1}, value is {2}",
    (uint) &y, sizeof(short), y);
Console.WriteLine(
    "Address of y2 is 0x{0:X}, size is {1}, value is {2}",
    (uint) &y2, sizeof(byte), y2);
Console.WriteLine(
    "Address of z is 0x{0:X}, size is {1}, value is {2}",
    (uint) &z, sizeof(double), z);
Console.WriteLine(
    "Address of pX= &x is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint) &pX, sizeof(int*), (uint)pX);
Console.WriteLine(
    "Address of pY= &y is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint) &pY, sizeof(short*), (uint)pY);
Console.WriteLine(
    "Address of pZ= &z is 0x{0:X}, size is {1}, value is 0x{2:X}",
    (uint) &pZ, sizeof(double*), (uint)pZ);

*pX = 20;
Console.WriteLine("After setting *pX, x = {0}", x);
Console.WriteLine(" *pX = {0}", *pX);

pZ = (double*)pX;
Console.WriteLine("x treated as a double = {0}", *pZ);

Console.ReadLine();
}
}

```

代码段 PointerPlayground/Program.cs

这段代码声明了 4 个值变量：

- `int x`
- `short y`
- `byte y2`
- `double z`

它还声明了指向其中 3 个值的指针：pX、pY 和 pZ。

然后显示这 3 个变量的值，以及它们的大小和地址。注意在获取 pX、pY 和 pZ 的地址时，我们查看的是指针的指针，即值的地址的地址！还要注意，与显示地址的常见方式一致，在 `Console.WriteLine()` 命令中使用 `{0:X}` 格式说明符，确保该内存地址以十六进制格式显示。

最后，使用指针 pX 把 x 的值改为 20，执行一些指针类型强制转换，如果把 x 的内容当作 `double` 类型，就会得到无意义的结果。

编译并运行这段代码，得到下面的结果，其中列出了用 `/unsafe` 标志进行编译和不用 `/unsafe` 标志进行编译的结果：

```

csc PointerPlayground.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground.cs(7,26): error CS0227: Unsafe code may only appear if

```

```

compiling with /unsafe

csc /unsafe PointerPlayground.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground
Address of x is 0x12F4B0, size is 4, value is 10
Address of y is 0x12F4AC, size is 2, value is -1
Address of y2 is 0x12F4A8, size is 1, value is 4
Address of z is 0x12F4A0, size is 8, value is 1.5
Address of pX= &x is 0x12F49C, size is 4, value is 0x12F4B0
Address of pY= &y is 0x12F498, size is 4, value is 0x12F4AC
Address of pZ= &z is 0x12F494, size is 4, value is 0x12F4A0
After setting *pX, x = 20
*pX = 20
x treated as a double = 2.86965129997082E-308

```

检查这些结果,可以证实 13.1 节描述的栈操作,即栈向下给变量分配内存。注意,这还证实了栈中的内存块总是按照 4 个字节的倍数进行分配。例如, y 是一个 short 数(其大小为 2 字节),其地址是 1 242 284(十进制),表示为该变量分配的存储单元是 1 242 284~1 242 287。如果.NET 运行库严格地逐个排列变量,则 y 应只占用两个存储单元,即 1 242 284 和 1 242 285。

下一个示例 `PointerPlayground2` 介绍指针的算术,以及结构指针和类成员指针。开始时,定义一个结构 `CurrencyStruct`,它把货币值表示为美元和美分,再定义一个等价的类 `CurrencyClass`:



```

internal struct CurrencyStruct
{
    public long Dollars;
    public byte Cents;

    public override string ToString()
    {
        return "$" + Dollars + "." + Cents;
    }
}

internal class CurrencyClass
{
    public long Dollars;
    public byte Cents;

    public override string ToString()
    {
        return "$" + Dollars + "." + Cents;
    }
}

```

代码段 `PointerPlayground.sln`

定义好结构和类后,就可以对它们应用指针了。下面的代码是一个新的示例。这段代码比较长,我们对此将做详细讲解。首先显示 `CurrencyStruct` 结构的字节数,创建它的两个实例和一些指针,然后使用 `pAmount` 指针初始化一个 `CurrencyStruct` 结构 `amount1` 的成员,显示变量的地址:

```

public static unsafe void Main()
{

```

```

Console.WriteLine(
    "Size of CurrencyStruct struct is " + sizeof(CurrencyStruct));
CurrencyStruct amount1, amount2;
CurrencyStruct* pAmount = & amount1;
long* pDollars = & (pAmount->Dollars);
byte* pCents = & (pAmount->Cents);
Console.WriteLine("Address of amount1 is 0x{0:X}", (uint)&amount1);
Console.WriteLine("Address of amount2 is 0x{0:X}", (uint)&amount2);
Console.WriteLine("Address of pAmount is 0x{0:X}", (uint)&pAmount);
Console.WriteLine("Address of pDollars is 0x{0:X}", (uint)&pDollars);
Console.WriteLine("Address of pCents is 0x{0:X}", (uint)&pCents);
pAmount->Dollars = 20;
* pCents = 50;
Console.WriteLine("amount1 contains " + amount1);

```

现在根据栈的工作方式，执行一些指针操作。因为变量是按顺序声明的，所以 `amount2` 存储在 `amount1` 后面的地址中，`sizeof(CurrencyStruct)` 运算符返回 16 (见后面的屏幕输出)，所以 `CurrencyStruct` 结构占用的字节数是 4 的倍数。在递减了 `Currency` 指针后，它就指向 `amount2`：

```

--pAmount;    // this should get it to point to amount2
Console.WriteLine("amount2 has address 0x{0:X} and contains {1}",
    (uint)pAmount, * pAmount);

```

在调用 `Console.WriteLine()` 语句时，它显示了 `amount2` 的内容，但还没有对它进行初始化。显示出来的东西就是随机的垃圾——在执行该示例内存中存储在该单元中的内容。但这有一个要点：一般情况下，C# 编译器会禁止使用未初始化的变量，但在开始使用指针时，就很容易绕过许多通常的编译检查。此时我们这么做，是因为编译器无法知道我们实际上要显示的是 `amount2` 的内容。因为知道了栈的工作方式，所以可以说出递减 `pAmount` 的结果是什么。使用指针算法，可以访问编译器通常禁止访问的各种变量和存储单元，因此指针算法是不安全的。

接下来在 `pCents` 指针上进行指针运算。`pCents` 指针目前指向 `amount1.Cents`，但此处的目的是使用指针算法让它指向 `amount2.Cents`，而不是直接告诉编译器我们要做什么。为此，需要从 `pCents` 指针所包含的地址中减去 `sizeof(Currency)`：

```

// do some clever casting to get pCents to point to cents
// inside amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Address of pCents is now 0x{0:X}", (uint)&pCents);

```

最后，使用 `fixed` 关键字创建一些指向类实例中字段的指针，使用这些指针设置这个实例的值。注意，这也是我们第一次查看存储在堆中(而不是栈)的项的地址：

```

Console.WriteLine("\nNow with classes");
// now try it out with classes
CurrencyClass amount3 = new CurrencyClass();

fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
    Console.WriteLine(
        "amount3.Dollars has address 0x{0:X}", (uint)pDollars2);
}

```

```

Console.WriteLine(
    "amount3.Cents has address 0x{0:X}", (uint) pCents2);
*pDollars2 = -100;
Console.WriteLine("amount3 contains " + amount3);
}

```

编译并运行这段代码，得到如下所示的结果：

```

csc /unsafe PointerPlayground2.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayground2
Size of CurrencyStruct struct is 16
Address of amount1 is 0x12F4A4
Address of amount2 is 0x12F494
Address of pAmount is 0x12F490
Address of pDollars is 0x12F48C
Address of pCents is 0x12F488
amount1 contains $20.50
amount2 has address 0x12F494 and contains $0.0
Address of pCents is now 0x12F488

Now with classes
amount3.Dollars has address 0xA64414
amount3.Cents has address 0xA6441C
amount3 contains $-100.0

```

注意在这个结果中，显示了未初始化的 `amount2` 的值，`CurrencyStruct` 结构的字节数是 16，大于其字段的字节数(一个 `long` 数占用 8 个字节，加上 1 个字节等于 9 个字节)。

13.3.3 使用指针优化性能

前面用许多篇幅介绍了使用指针可以完成的各种任务，但在前面的示例中，仅是处理内存，让有兴趣的人们了解实际上发生了什么事，并没有帮助人们编写出更好的代码！本节将应用我们对指针的理解，用一个示例来说明使用指针可以大大提高性能。

1. 创建基于栈的数组

本节将探讨指针的一个主要应用领域：在栈中创建高性能、低系统开销的数组。第 2 章介绍了 C# 如何支持数组的处理。C# 很容易使用一维数组和矩形或锯齿形多维数组，但有一个缺点：这些数组实际上都是对象，它们是 `System.Array` 的实例。因此数组存储在堆上，这会增加系统开销。有时，我们希望创建一个使用时间比较短的高性能数组，不希望有引用对象的系统开销。而使用指针就可以做到，但指针只对于一维数组比较简单。

为了创建一个高性能的数组，需要使用另一个关键字：`stackalloc`。`stackalloc` 命令指示 .NET 运行库在栈上分配一定量的内存。在调用 `stackalloc` 命令时，需要为它提供两条信息：

- 要存储的数据类型
- 需要存储的数据项数

例如，要分配足够的内存，以存储 10 个 `decimal` 数据项，可以编写下面的代码：

```
decimal * pDecimals = stackalloc decimal[10];
```

注意, 这条命令只分配栈内存。它不会试图把内存初始化为任何默认值, 这正好符合我们的目的。因为要创建一个高性能的数组, 给它不必要地初始化相应值会降低性能。

同样, 要存储 20 个 `double` 数据项, 可以编写下面的代码:

```
double* pDoubles = stackalloc double[20];
```

虽然这行代码指定把变量的个数存储为一个常数, 但它等于在运行时计算的一个数字。所以可以把上面的示例写为:

```
int size;
size = 20;    // or some other value calculated at run-time
double * pDoubles = stackalloc double[size];
```

从这些代码段中可以看出, `stackalloc` 的语法有点不寻常。它的后面紧跟要存储的数据类型名(该数据类型必须是一个值类型), 之后把需要的项数放在方括号中。分配的字节数是项数乘以 `sizeof(数据类型)`。在这里, 使用方括号表示这是一个数组。如果给 20 个 `double` 数分配存储单元, 就得到了一个有 20 个元素的 `double` 数组, 最简单的数组类型是逐个存储元素的内存块, 如图 13-6 所示。

在图 13-6 中, 显示了 `stackalloc` 返回的指针, `stackalloc` 总是返回分配数据类型的指针, 它指向新分配内存块的顶部。要使用这个内存块, 可以取消对已返回指针的引用。例如, 给 20 个 `double` 数分配内存后, 把第一个元素(数组的元素 0)设置为 3.0, 可以编写下面的代码:

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
```

要访问数组的下一个元素, 可以使用指针算法。如前所述, 如果给一个指针加 1, 它的值就会增加它指向的数据类型的字节数。在本例中, 就会把指针指向已分配的内存块中的下一个空闲存储单元。因此可以把数组的第二个元素(元素编号为 1)设置为 8.4:

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

同样, 可以用表达式 `*(pDoubles+X)` 访问数组中下标为 X 的元素。

这样, 就得到一种访问数组中元素的方式, 但对于一般目的, 使用这种语法过于复杂。C# 为此定义了另一种语法。对指针应用方括号时, C# 为方括号提供了一种非常精确的含义。如果变量 `p` 是任意指针类型, `X` 是一个整数, 表达式 `p[X]` 就被编译器解释为 `*(p+X)`, 这适用于所有的指针, 不仅仅是用 `stackalloc` 初始化的指针。利用这个简捷的记号, 就可以用一种非常方便的语法访问数组。实际上, 访问基于栈的一维数组所使用的语法与访问基于堆的、由 `System.Array` 类表示的数组完全相同:

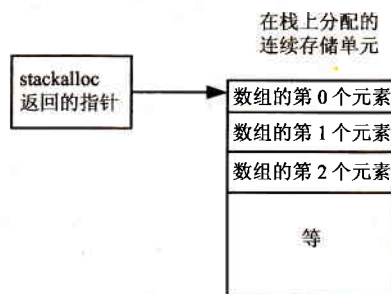


图 13-6

```
double* pDoubles = stackalloc double [20];
pDoubles[0] = 3.0; // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4; // pDoubles[1] is the same as *(pDoubles+1)
```



把数组的语法应用于指针并不是新东西。自从开发出 C 和 C++ 语言以来，它就是这两种语言的基础部分。实际上，C++ 开发人员会把这里用 `stackalloc` 获得的、基于栈的数组完全等同于传统的基于栈的 C 和 C++ 数组。这种语法和指针与数组的链接方式是 C 语言在 20 世纪 70 年代后期流行起来的原因之一，也是指针的使用成为 C 和 C++ 中一种流行的编程技巧的主要原因。

尽管高性能的数组可以用与一般 C# 数组相同的方式访问，但需要注意：在 C# 中，下面的代码会抛出一个异常：

```
double[] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

抛出异常的原因是：使用越界的下标来访问数组：下标是 50，而允许的最大下标是 19。但是，如果使用 `stackalloc` 声明了一个等价的数组，对数组进行边界检查时，这个数组中就没有封装任何对象，因此下面的代码不会抛出异常：

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

在这段代码中，我们分配了足够的内存来存储 20 个 `double` 类型的数。接着把 `sizeof(double)` 存储单元的起始位置设置为该存储单元的起始位置加上 `50*sizeof(double)` 个存储单元，来保存双精度值 3.0。但这个存储单元超出了刚才为 `double` 数分配的内存区域。谁也不知道这个地址存储了什么数据。最好是只使用某个当前未使用的内存，但所重写的存储单元也有可能在是栈上用于存储其他变量，或者是某个正在执行的方法的返回地址。因此，使用指针获得高性能的同时，也会付出一些代价：需要确保自己知道在做什么，否则就会抛出非常古怪的运行错误。

2. QuickArray 示例

下面用一个 `stackalloc` 示例 `QuickArray` 来结束关于指针的讨论。在这个示例中，程序仅要求用户提供为数组分配的元素数。然后代码使用 `stackalloc` 给 `long` 型数组分配一定的存储单元。这个数组的元素是从 0 开始的整数的平方，结果显示在控制台上：

```
using System;

namespace QuickArray
{
    internal class Program
    {
        private static unsafe void Main()
        {
            Console.WriteLine("How big an array do you want? \n > ");
            string userInput = Console.ReadLine();
            uint size = uint.Parse(userInput);
```

```
long * pArray = stackalloc long[(int) size];
for (int i = 0; i < size; i++)
{
    pArray[i] = i * i;
}

for (int i = 0; i < size; i++)
{
    Console.WriteLine("Element {0} = {1}", i, * (pArray + i));
}

Console.ReadLine();
}
```

运行这个示例，得到如图 13-7 所示的结果：



图 13-7

13.4 小结

要想成为真正优秀的 C#程序员，必须牢固掌握存储单元和垃圾回收的工作原理。本章描述了 CLR 管理以及在堆和栈上分配内存的方式，讨论了如何编写正确地释放非托管资源的类，并介绍如何在 C#中使用指针，这些都是很难理解的高级主题，初学者常常不能正确实现。

本章为第 15 章的错误处理和第 20 章的何时使用线程打下了基础。下一章介绍 C#中的反射技术。

第 14 章

反 射

本章内容:

- 使用自定义特性
- 在运行期间使用反射检查这些元数据
- 从支持反射的类中构建访问点

本章讨论自定义特性和反射。自定义特性允许把自定义元数据与程序元素关联起来。这些元数据是在编译过程中创建的,并嵌入到程序集中。反射是一个普通术语,它描述了在运行过程中检查和处理程序元素的功能。例如,反射允许完成以下任务:

- 枚举类型的成员
- 实例化新对象
- 执行对象的成员
- 查找类型的信息
- 查找程序集的信息
- 检查应用于某种类型的自定义特性
- 创建和编译新程序集

这个列表列出了许多功能,包括.NET Framework 类库提供的一些最强大、最复杂的功能。但本章不可能介绍反射的所有功能,仅讨论最常用的功能。

为了说明自定义特性和反射,我们将开发一个示例,说明公司如何定期升级软件,自动记录升级的信息。在这个示例中,要定义几个自定义特性,表示程序元素最后修改的日期,以及发生了什么变化。然后使用反射开发一个应用程序,它在程序集中查找这些特性,自动显示软件自某个给定日期以来升级的所有信息。

本章要讨论的另一个示例是一个应用程序,该程序从数据库中读取信息或把信息写入数据库,并使用自定义特性,把类和属性标记为对应的数据库表和列。然后在运行期间从程序集中读取这些特性,使程序可以自动从数据库的相应位置检索或写入数据,无需为每个表或每一列编写特定的逻辑。

14.1 自定义特性

前面介绍了如何在程序的各个数据项上定义特性。这些特性都是 Microsoft 定义好的,作为.NET Framework 类库的一部分,许多特性都得到了 C#编译器的支持。对于这些特殊的特性,编译器可以以特殊的方式定制编译过程,例如,可以根据 StructLayout 特性中的信息在内存中布置结构。

.NET Framework 也允许用户定义自己的特性。显然, 这些特性不会影响编译过程, 因为编译器不能识别它们, 但这些特性在应用于程序元素时, 可以在编译好的程序集中用作元数据。

这些元数据在文档说明中非常有用。但是, 使自定义特性非常强大的因素是使用反射, 代码可以读取这些元数据, 使用它们在运行期间作出决策, 也就是说, 自定义特性可以直接影响代码运行的方式。例如, 自定义特性可以用于支持对自定义许可类进行声明性的代码访问安全检查, 把信息与程序元素关联起来, 程序元素由测试工具使用, 或者在开发可扩展的架构时, 允许加载插件或模块。

14.1.1 编写自定义特性

为了理解编写自定义特性的方式, 应了解一下在编译器遇到代码中某个应用了自定义特性的元素时, 该如何处理。以数据库为例, 假定有一个 C# 属性声明, 如下所示。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    get {
        // etc.
```

当 C# 编译器发现这个属性(property)应用了一个 `FieldName` 特性时, 首先会把字符串 `Attribute` 追加到这个名称的后面, 形成一个组合名称 `FieldNameAttribute`, 然后在其搜索路径的所有名称空间(即在 `using` 语句中提及的名称空间)中搜索有指定名称的类。但要注意, 如果用一个特性标记数据项, 而该特性的名称以字符串 `Attribute` 结尾, 编译器就不会将该字符串加到组合名称中, 而是不修改该特性名。因此, 上面的代码等价于:

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    get {
        // etc.
```

编译器会找到含有该名称的类, 且这个类直接或间接派生自 `System.Attribute`。编译器还认为这个类包含控制特性用法的信息。特别是属性类需要指定:

- 特性可以应用到哪些类型的程序元素上(类、结构、属性和方法等)
- 它是否可以多次应用到同一个程序元素上
- 特性在应用到类或接口上时, 是否由派生类和接口继承
- 这个特性有哪些必选和可选参数

如果编译器找不到对应的特性类, 或者找到一个这样的特性类, 但使用特性的方式与特性类中的信息不匹配, 编译器就会产生一个编译错误。例如, 如果特性类指定该特性只能应用于类, 但我们把它应用到结构定义上, 就会产生一个编译错误。

继续上面的示例, 假定定义了一个 `FieldName` 特性:

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false,
    Inherited=false)]
public class FieldNameAttribute: Attribute
{
    private string name;
```

```

public FieldNameAttribute(string name)
{
    this.name = name;
}
}

```

下面几节讨论这个定义中的每个元素。

1. AttributeUsage 特性

要注意的第一个问题是特性(attribute)类本身用一个特性——`System.AttributeUsage` 特性来标记。这是 Microsoft 定义的一个特性, C#编译器为它提供了特殊的支持(您可能认为 `AttributeUsage` 根本不是一个属性, 它更像一个元特性, 因为它只能应用到其他特性上, 不能应用到类上)。 `AttributeUsage` 主要用于标识自定义特性可以应用到哪些类型的程序元素上。这些信息由它的第一个参数给出, 该参数是必选的, 其类型是枚举类型 `AttributeTargets`。在上面的示例中, 指定 `FieldName` 特性只能应用到属性(property)上——这是因为我们在前面的代码段中把它应用到属性上。 `AttributeTargets` 枚举的成员如下:

- All
- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter(仅.NET 2.0 及更高版本提供)
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

这个列表列出了可以应用该特性的所有程序元素。注意在把特性应用到程序元素上时, 应把特性放在元素前面的方括号中。但是, 在上面的列表中, 有两个值不对应于任何程序元素: `Assembly` 和 `Module`。特性可以应用到整个程序集或模块中, 而不是应用到代码中的一个元素上, 在这种情况下, 这个特性可以放在源代码的任何地方, 但需要用关键字 `assembly` 或 `module` 作为前缀:

```

[assembly:SomeAssemblyAttribute(Parameters)]
[module:SomeAssemblyAttribute(Parameters)]

```

在指定自定义特性的有效目标元素时, 可以使用按位 OR 运算符把这些值组合起来。例如, 如果指定 `FieldName` 特性可以同时应用到属性和字段上, 可以编写下面的代码:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
AllowMultiple=false,
Inherited=false)]
public class FieldNameAttribute: Attribute
```

也可以使用 `AttributeTargets.All` 指定自定义特性可以应用到所有类型的程序元素上。`AttributesUsage` 特性还包含另外两个参数: `AllowMultiple` 和 `Inherited`。它们用不同的语法来指定: `<ParameterName>=<ParameterValue>`, 而不是只给出这些参数的值。这些参数是可选的, 根据需要, 可以忽略它们。

`AllowMultiple` 参数表示一个特性是否可以多次应用到同一项上, 这里把它设置为 `false`, 表示如果编译器遇到下述代码, 就会产生一个错误:

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{
    // etc.
```

如果把 `Inherited` 参数设置为 `true`, 就表示应用到类或接口上的特性也可以自动应用到所有派生的类或接口上。如果特性应用到方法或属性上, 它就可以自动应用到该方法或属性等的重写版本上。

2. 指定特性参数

下面介绍如何指定自定义特性接受的参数。在编译器遇到下述语句时:

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    // etc.
```

编译器会检查传递给特性的参数(在本例中, 是一个字符串), 并查找该特性中带这些参数的构造函数。如果编译器找到一个这样的构造函数, 编译器就会把指定的元数据传递给程序集。如果编译器找不到, 就生成一个编译错误。如后面所述, 反射会从程序集中读取元数据(属性), 并实例化它们表示的特性类。因此, 编译器需要确保存在这样的构造函数, 才能在运行期间实例化指定的特性。

在本例中, 仅为 `FieldNameAttribute` 类提供了一个构造函数, 而这个构造函数有一个字符串参数。因此, 在把 `FieldName` 特性应用到一个属性上时, 必须为它提供一个字符串参数, 如上面的代码所示。

如果可以选择特性提供的参数类型, 就可以提供构造函数的不同重载方法, 尽管一般是仅提供一个构造函数, 使用属性来定义任何其他可选参数, 下面将介绍可选参数。

3. 指定特性的可选参数

在 `AttributeUsage` 特性中, 可以使用另一种语法, 把可选参数添加到特性中。这种语法指定可选参数的名称和值, 它通过特性类中的公共属性或字段起作用。例如, 假定修改 `SocialSecurityNumber` 属性的定义, 如下所示:

```
[FieldName("SocialSecurityNumber", Comment="This is the primary key field")]
public string SocialSecurityNumber
{
    // etc.
```

在本例中, 编译器识别第二个参数的语法<ParameterName>=<ParameterValue>, 并且不会把这个参数传递给 `FieldNameAttribute` 类的构造函数, 而是查找一个有该名称的公共属性或字段(最好不要使用公共字段, 所以一般情况下要使用特性), 编译器可以用这个属性设置第二个参数的值。如果希望上面的代码工作, 就必须给 `FieldNameAttribute` 类添加一些代码:

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false,
    Inherited=false)]
public class FieldNameAttribute: Attribute
{
    private string comment;
    public string Comment
    {
        get
        {
            return comment;
        }
        set
        {
            comment = value;
        }
    }
    // etc
}
```

14.1.2 自定义特性示例: WhatsNewAttributes

本节开始编写前面描述过的示例 `WhatsNewAttributes`, 该示例提供了一个特性, 表示最后一次修改程序元素的时间。这个示例比前面所有的示例都复杂, 因为它包含 3 个不同的程序集:

- `WhatsNewAttributes` 程序集, 它包含特性的定义。
- `VectorClass` 程序集, 它包含所应用的特性的代码。
- `LookUpWhatsNew` 程序集, 它包含显示已改变的数据项详细信息的项目。

其中, 只有 `LookUpWhatsNew` 程序集是目前为止使用的一个控制台应用程序, 其余两个程序集都是库, 它们都包含类的定义, 但都没有程序的入口点。对于 `VectorClass` 程序集, 我们使用了 `VectorAsCollection` 示例, 但从中删除了入口点和测试代码类, 只剩下 `Vector` 类。这些类详见本章后面的内容。

在命令行上编译, 以此管理 3 个相关的程序集要求较高的技巧。尽管分别给出编译这 3 个源文件的命令, 但也可以编辑代码示例(可以从 Wrox 网站 www.wrox.com 上下载), 作为一个组合的 Visual Studio 解决方案, 详见第 16 章。下载的文件包含所需的 Visual Studio 2010 解决方案文件。

1. WhatsNewAttributes 库程序集

首先从核心的 `WhatsNewAttributes` 程序集开始。其源代码包含在 `WhatsNewAttributes.cs` 文件中, 该文件位于本章示例代码中 `WhatsNewAttributes` 解决方案的 `WhatsNewAttributes` 项目中。编译的语法

非常简单：在命令行上，给编译器提供 `target:library` 标记即可。要编译 `WhatsNewAttributes` 程序集，输入：

```
csc /target:library WhatsNewAttributes.cs
```

`WhatsNewAttributes.cs` 文件定义了两个特性类 `LastModifiedAttribute` 和 `SupportsWhatsNewAttribute`。`LastModifiedAttribute` 特性可以用于标记最后一次修改数据项的时间，它有两个必选参数（这两个参数传递给构造函数）：修改的日期和包含描述修改信息的字符串。它还有一个可选参数 `issues`（表示存在一个公共属性），它可以用来描述该数据项的任何重要问题。

在现实生活中，或许想把特性应用到任何对象上。为了使代码比较简单，这里仅允许将它应用于类和方法，并允许它多次应用到同一项上（`AllowMultiple=true`），因为可以多次修改某一项，每次修改都需要用一个不同的特性实例来标记。

`SupportsWhatsNew` 是一个较小的类，它表示不带任何参数的特性。这个特性是一个程序集的特性，它用于把程序集标记为通过 `LastModifiedAttribute` 维护的文档。这样，以后查看这个程序集的程序会知道，它读取的程序集是我们使用自动文档过程生成的那个程序集。这部分示例的完整源代码如下所示：



可从
wrox.com
下载源代码

```
using System;

namespace WhatsNewAttributes
{
    [AttributeUsage(
        AttributeTargets.Class | AttributeTargets.Method,
        AllowMultiple=true, Inherited=false)]
    public class LastModifiedAttribute: Attribute
    {
        private readonly DateTime _dateModified;
        private readonly string _changes;

        public LastModifiedAttribute(string dateModified, string changes)
        {
            dateModified = DateTime.Parse(dateModified);
            changes = changes;
        }

        public DateTime DateModified
        {
            get { return dateModified; }
        }

        public string Changes
        {
            get { return changes; }
        }

        public string Issues { get; set; }
    }

    [AttributeUsage(AttributeTargets.Assembly)]
    public class SupportsWhatsNewAttribute: Attribute
    {
    }
}
```

代码段 WhatsNewAttributes.cs

从上面的描述可以看出，上面的代码非常简单。但要注意，不必将 `set` 访问器提供给 `Changes` 和 `DateModified` 属性，不需要这些访问器是因为，在构造函数中这些参数都设置为必选参数。需要 `get` 访问器，是为了以后可以读取这些特性的值。

2. VectorClass 程序集

本节就使用这些特性，我们用前面的 `VectorAsCollection` 示例的修订版本来说明。注意这里需要引用刚才创建的 `WhatsNewAttributes` 库，还需要使用 `using` 语句指定相应的名称空间，这样编译器才能识别这些特性：

```
using System;
using System.Collections;
using System.Text;
using WhatsNewAttributes;

[assembly: SupportsWhatsNew]
```

在这段代码中，添加了一行用 `SupportsWhatsNew` 特性标记程序集本身的代码。

下面考虑 `Vector` 类的代码。我们并不是真的要修改这个类中的某些主要内容，只是添加两个 `LastModified` 特性，以标记出本章对 `Vector` 类进行的操作。把 `Vector` 定义为一个类，而不是结构，以简化后面显示特性所编写的代码(在 `VectorAsCollection` 示例中，`Vector` 是一个结构，但其枚举器是一个类。于是，在查看程序集时，这个示例的下一迭代必须同时考虑类和结构。这会使例子比较复杂)。

```
namespace VectorClass
{
    [LastModified("14 Feb 2010", "IEnumerable interface implemented " +
        "So Vector can now be treated as a collection")]
    [LastModified("10 Feb 2010", "IFormattable interface implemented " +
        "So Vector now responds to format specifiers N and VE")]
    class Vector: IFormattable, IEnumerable
    {
        public double x, y, z;

        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        [LastModified("10 Feb 2010",
            "Method added in order to provide formatting support")]
        public string ToString(string format, IFormatProvider formatProvider)
        {
            if (format == null)
            {
                return ToString();
            }
        }
    }
}
```

再把包含的 `VectorEnumerator` 类标记为 `new`:

```
[LastModified("14 Feb 2010",
             "Class created as part of collection support for Vector")]
private class VectorEnumerator: IEnumerator
{
```

为了在命令行上编译这段代码, 应输入下面的命令:

```
csc /target:library /reference:WhatsNewAttributes.dll VectorClass.cs
```

上面是这个示例的代码。目前还不能运行它, 因为我们只有两个库。在描述了反射的工作原理后, 就介绍这个示例的最后一部分, 从中可以查看和显示这些特性。

14.2 反射

本节先介绍 `System.Type` 类, 通过这个类可以访问关于任何数据类型的信息。然后简要介绍 `System.Reflection.Assembly` 类, 它可以用于访问给定程序集的相关信息, 或者把这个程序集加载到程序中。最后把本节的代码和上一节的代码结合起来, 完成 `WhatsNewAttributes` 示例。

14.2.1 `System.Type` 类

这里使用 `Type` 类只为了存储类型的引用:

```
Type t = typeof(double);
```

我们以前把 `Type` 看作一个类, 但它实际上是一个抽象的基类。只要实例化了一个 `Type` 对象, 实际上就实例化了 `Type` 的一个派生类。尽管一般情况下派生类只提供各种 `Type` 方法和属性的不同重载, 但是这些方法和属性返回对应数据类型的正确数据, `Type` 有与每种数据类型对应的派生类。它们一般不添加新的方法或属性。通常, 获取指向任何给定类型的 `Type` 引用有 3 种常用方式:

- 使用 C# 的 `typeof` 运算符, 如上述代码所示。这个运算符的参数是类型的名称(但不放在引号中)。
- 使用 `GetType()` 方法, 所有的类都会从 `System.Object` 继承这个方法。

```
double d = 10;
Type t = d.GetType();
```

在一个变量上调用 `GetType()` 方法, 而不是把类型的名称作为其参数。但要注意, 返回的 `Type` 对象仍只与该数据类型相关: 它不包含与该类型的实例相关的任何信息。如果引用了一个对象, 但不能确保该对象实际上是哪个类的实例, 这个方法就很有用。

- 还可以调用 `Type` 类的静态方法 `GetType()`:

```
Type t = Type.GetType("System.Double");
```

`Type` 是许多反射功能的入口。它实现许多方法和属性, 这里不可能列出所有的方法和属性, 而主要介绍如何使用这个类。注意, 可用的属性都是只读的: 可以使用 `Type` 确定数据的类型,

但不能使用它修改该类型！

1. Type 的属性

由 Type 实现的属性可以分为下述 3 类：

- 许多属性都可以获取包含与类相关的各种名称的字符串，如表 14-1 所示。

表 14-1

属 性	返 回 值
Name	数据类型名
FullName	数据类型的完全限定名(包括名称空间名)
Namespace	在其中定义数据类型的名称空间名

- 属性还可以进一步获取 Type 对象的引用，这些引用表示相关的类，如表 14-2 所示。

表 14-2

属 性	返回对应的 Type 引用
BaseType	该 Type 的直接基本类型
UnderlyingSystemType	该 Type 在 .NET 运行库中映射到的类型(某些 .NET 基类实际上映射到由 IL 识别的特定预定义类型)

- 许多布尔属性表示这种类型是一个类，还是一个枚举等。这些特性包括 IsAbstract、IsArray、IsClass、IsEnum、IsInterface、IsPointer、IsPrimitive(一种预定义的基元数据类型)、IsPublic、IsSealed 和 IsValueType。例如，使用一种基元数据类型：

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract);    // writes false
Console.WriteLine(intType.IsClass);       // writes false
Console.WriteLine(intType.IsEnum);        // writes false
Console.WriteLine(intType.IsPrimitive);    // writes true
Console.WriteLine(intType.IsValueType);    // writes true
```

或者使用 Vector 类：

```
Type vecType = typeof(Vector);
Console.WriteLine(vecType.IsAbstract);    // writes false
Console.WriteLine(vecType.IsClass);       // writes true
Console.WriteLine(vecType.IsEnum);        // writes false
Console.WriteLine(vecType.IsPrimitive);    // writes false
Console.WriteLine(vecType.IsValueType);    // writes false
```

也可以获取在其中定义该类型的程序集的引用，该引用作为 System.Reflection.Assembly 类的实例的一个引用来返回：

```
Type t = typeof (Vector);
Assembly containingAssembly = new Assembly(t);
```

2. 方法

System.Type 的大多数方法都用于获取对应数据类型的成员信息：构造函数、属性、方法和事件等。它有许多方法，但它们都有相同的模式。例如，有两个方法可以获取数据类型的方法的细节信息：GetMethod()和 GetMethods()。GetMethod()方法返回 System.Reflection.MethodInfo 对象的一个引用，其中包含一个方法的细节信息。GetMethods()返回这种引用的一个数组。其区别是 GetMethods()方法返回所有方法的细节信息；而 GetMethod()方法返回一个方法的细节信息，其中该方法包含特定的参数列表。这两个方法都有重载方法，重载方法有一个附加的参数，即 BindingFlags 枚举值，该值表示应返回哪些成员，例如，返回公有成员、实例成员和静态成员等。

例如，GetMethods()最简单的一个重载方法不带参数，返回数据类型所有公共方法的信息：

```
Type t = typeof(double);
MethodInfo[] methods = t.GetMethods();
foreach (MethodInfo nextMethod in methods)
{
    // etc.
}
```

Type 的成员方法如表 14-3 所示，遵循同一个模式。

表 14-3

返回的对象类型	方法(名称为复数形式的方法返回一个数组)
ConstructorInfo	GetConstructor(), GetConstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()
PropertyInfo	GetProperty(), GetProperties()

GetMember()和 GetMembers()方法返回数据类型的任何成员或所有成员的详细信息，不管这些成员是构造函数、属性和方法等。

14.2.2 TypeView 示例

下面用一个短小的示例 TypeView 来说明 Type 类的一些功能，这个示例可以用来列出数据类型的所有成员。本例主要说明对于 double 型 TypeView 的用法，也可以修改该样例中的一行代码，使用其他的数据类型。TypeView 提供的信息要比在控制台窗口中显示的信息多得多，所以我们将打破常规，在一个消息框中显示这些信息。对于一个 double 数运行 TypeView 示例，结果如图 14-1 所示。

该消息框显示了数据类型的名称、全名和名称空间，以及底层类型和基类的名称。然后，它迭代该数据类型的所有公有实例成员，显

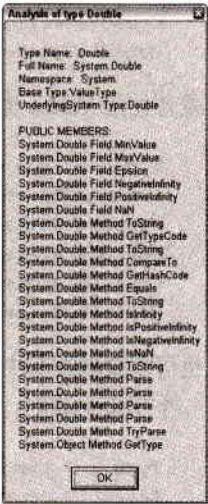


图 14-1

示所声明类型的每个成员、成员的类型(方法、字段等)以及成员的名称。声明类型是实际声明类型成员的类的名称(例如,如果在 `System.Double` 中定义或重载它,该声明类型就是 `System.Double`,如果成员继承自某个基类,该声明类型就是相关基类的名称)。

`TypeView` 不会显示方法的签名,因为我们是通过 `MemberInfo` 对象获取所有公有实例成员的详细信息,参数的相关信息不能通过 `MemberInfo` 对象来获得。为了获取该信息,需要引用 `MemberInfo` 和其他更特殊的对象,即需要分别获取每一种类型的成员的详细信息。

`TypeView` 会显示所有公有实例成员的详细信息,但对于 `double` 类型,仅定义了字段和方法。对于本示例,把 `TypeView` 编译为一个控制台应用程序,可以从控制台应用程序中显示消息框。但是,使用消息框就意味着需要引用基类程序集 `System.Windows.Forms.dll`,它包含 `System.Windows.Forms` 名称空间中的类,在这个名称空间中,定义了我们需要的 `MessageBox` 类。下面列出 `TypeView` 的代码。开始时需要添加几条 `using` 语句:

```
using System;
using System.Reflection;
using System.Text;
using System.Windows.Forms;
```

需要 `System.Text` 的原因是我们要使用 `StringBuilder` 对象构建在消息框中显示的文本,以及消息框本身的 `System.Windows.Forms`。全部代码都放在 `MainClass` 一个类中,这个类包含两个静态方法和一个静态字段, `StringBuilder` 的一个实例叫作 `OutputText`, `OutputText` 用于创建在消息框中显示的文本。`Main()`方法和类的声明如下所示:

```
class MainClass
{
    static StringBuilder OutputText = new StringBuilder();

    static void Main()
    {
        // modify this line to retrieve details of any
        // other data type
        Type t = typeof(double);

        AnalyzeType(t);
        MessageBox.Show(OutputText.ToString(), "Analysis of type "
                        + t.Name);

        Console.ReadLine();
    }
}
```

实现的 `Main()`方法首先声明一个 `Type` 对象,来表示我们选择的数据类型,再调用方法 `AnalyzeType()`, `AnalyzeType()`方法从 `Type` 对象中提取信息,并使用该信息构建输出文本。最后在消息框中显示输出。使用 `MessageBox` 类非常直观:只需调用其静态方法 `Show()`,给它传递两个字符串,两个字符串分别为消息框中的文本和标题。这些都由 `AnalyzeType()`方法来完成:

```
static void AnalyzeType(Type t)
{
    AddToOutput("Type Name: " + t.Name);
    AddToOutput("Full Name: " + t.FullName);
    AddToOutput("Namespace: " + t.Namespace);
}
```

```

Type tBase = t.BaseType;

if (tBase != null)
{
    AddToOutput("Base Type:" + tBase.Name);
}

Type tUnderlyingSystem = t.UnderlyingSystemType;

if (tUnderlyingSystem != null)
{
    AddToOutput("UnderlyingSystem Type:" + tUnderlyingSystem.Name);
}

AddToOutput("\nPUBLIC MEMBERS:");
MemberInfo [] Members = t.GetMembers();

foreach (MemberInfo NextMember in Members)
{
    AddToOutput(NextMember.DeclaringType + " " +
        NextMember.MemberType + " " + NextMember.Name);
}
}

```

实现 `AnalyzaType()` 方法, 仅需调用 `Type` 对象的各种属性, 就可以获得我们需要的类型名称的相关信息, 再调用 `GetMembers()` 方法, 获得一个 `MemberInfo` 对象数组, 该数组用于显示每个成员的信息。注意这里使用了一个辅助方法 `AddToOutput()`, 该方法创建要在消息框中显示的文本:

```

static void AddToOutput(string Text)
{
    OutputText.Append("\n" + Text);
}

```

使用下面的命令编译 `TypeView` 程序集:

```
csc /reference:System.Windows.Forms.dll Program.cs
```

14.2.3 Assembly 类

`Assembly` 类在 `System.Reflection` 名称空间中定义, 它允许访问给定程序集的元数据, 它也包含可以加载和执行程序集(假定该程序集是可执行的)的方法。与 `Type` 类一样, `Assembly` 类包含非常多的方法和属性, 这里不可能逐一论述。下面仅介绍完成 `WhatsNewAttributes` 示例所需要的方法和属性。

在使用 `Assembly` 实例做一些工作前, 需要把相应的程序集加载到正在运行的进程中。为此, 可以使用静态成员 `Assembly.Load()` 或 `Assembly.LoadFrom()`。这两个方法的区别是 `Load()` 方法的参数是程序集的名称, 运行库会在各个位置上搜索该程序集, 试图找到该程序集, 这些位置包括本地目录和全局程序集缓存。而 `LoadFrom()` 方法的参数是程序集的完整路径名, 它不会在其他位置搜索该程序集:

```

Assembly assembly1 = Assembly.Load("SomeAssembly");
Assembly assembly2 = Assembly.LoadFrom
    ("C:\My Projects\Software\SomeOtherAssembly");

```

这两个方法都有许多其他重载版本，它们提供了其他安全信息。加载了一个程序集后，就可以使用它的各种属性进行查询，例如，查找它的全名：

```
string name = assembly1.FullName;
```

1. 查找在程序集中定义的类型

`Assembly` 类的一个功能是它可以获得在相应程序集中定义的所有类型的详细信息，只要调用 `Assembly.GetTypes()` 方法，它就可以返回一个包含所有类型的详细信息的 `System.Type` 引用数组，然后就可以按照上一节的方式处理这些 `Type` 引用了：

```
Type[] types = theAssembly.GetTypes();
foreach (Type definedType in types)
{
    DoSomethingWith(definedType);
}
```

2. 查找自定义特性

用于查找在程序集或类型中定义了什么自定义特性的方法取决于与该特性相关的对象类型。如果要确定程序集从整体上关联了什么自定义特性，就需要调用 `Attribute` 类的一个静态方法 `GetCustomAttributes()`，给它传递程序集的引用：

```
Attribute[] definedAttributes =
    Attribute.GetCustomAttributes(assembly1);
// assembly1 is an Assembly object
```



这是相当重要的。以前您可能想知道，在定义自定义特性时，为什么必须费尽周折为它们编写类，以及为什么 Microsoft 没有更简单的语法。答案就在于此。自定义特性确实与对象一样，加载了程序集后，就可以读取这些特性对象，查看它们的属性，调用它们的方法。

`GetCustomAttributes()` 方法用于获取程序集的特性，它有两个重载方法：如果在调用它时，除了程序集的引用外，没有指定其他参数，该方法就会返回为这个程序集定义的所有自定义特性。当然，也可以通过指定第二个参数来调用它，第二个参数是表示感兴趣的特性类的一个 `Type` 对象，在这种情况下，`GetCustomAttributes()` 方法就返回一个数组，该数组包含指定类型的所有特性。

注意，所有特性都作为一般的 `Attribute` 引用来获取。如果要调用为自定义特性定义的任何方法或属性，就需要把这些引用显式转换为相关的自定义特性类。调用 `Assembly.GetCustomAttributes()` 的另一个重载方法，可以获得与给定数据类型相关的自定义特性的详细信息，这次传递的是一个 `Type` 引用，它描述了要获取的任何相关特性的类型。另一方面，如果要获得与方法、构造函数和字段等相关的特性，就需要调用 `GetCustomAttributes()` 方法，该方法是 `MethodInfo`、`ConstructorInfo` 和 `FieldInfo` 等类的一个成员。

如果只需要给定类型的一个特性，就可以调用 `GetCustomAttribute()` 方法，它返回一个 `Attribute` 对象。在 `WhatsNewAttributes` 示例中使用 `GetCustomAttribute()` 方法，是为了确定程序集中是否有

SupportsWhatsNew 特性。为此，调用 `GetCustomAttributes()` 方法，传递对 `WhatsNewAttributes` 程序集的一个引用和 `SupportWhatsNewAttribute` 特性的类型。如果有这个特性，就返回一个 `Attribute` 实例。如果在程序集中没有定义任何实例，就返回 `null`。如果找到两个或多个实例，`GetCustomAttributes()` 方法就抛出一个 `System.Reflection.AmbiguousMatchException` 异常：

```
Attribute supportsAttribute =
    Attribute.GetCustomAttributes(assembly1,
        typeof(SupportsWhatsNewAttribute));
```

14.2.4 完成 WhatsNewAttributes 示例

现在已经有足够的知识来完成 `WhatsNewAttributes` 示例了。为该示例中的最后一个程序集 `LookUpWhatsNew` 编写源代码，这部分应用程序是一个控制台应用程序，它需要引用其他两个程序集 `WhatsNewAttributes` 和 `VectorClass`。这是一个命令行应用程序，但仍可以像前面的 `TypeView` 示例那样在消息框中显示结果，因为结果是许多文本，所以不能显示在一个控制台窗口屏幕截图中。

这个文件的名称为 `LookUpWhatsNew.cs`，编译它的命令是：

```
csc /reference:WhatsNewAttributes.dll /reference:VectorClass.dll LookUpWhatsNew.cs
```

在这个文件的源代码中，首先指定要使用的名称空间 `System.Text`，因为需要再次使用一个 `StringBuilder` 对象：

```
using System;
using System.Reflection;
using System.Windows.Forms;
using System.Text;
using WhatsNewAttributes;

namespace LookUpWhatsNew
{
```

`WhatsNewChecker` 类包含主程序入口点和其他方法。我们定义的所有方法都在这个类中，它还有两个静态字段：`outputText` 和 `backDateTo`。`outputText` 字段包含在准备阶段创建的文本，这个文本要写到消息框中，`backDateTo` 字段存储了选择的日期——自从该日期以来进行的所有修改都要显示出来。一般情况下，需要显示一个对话框，让用户选择这个日期，但我们不想编写这种代码，以免转移读者的注意力。因此，把 `backDateTo` 字段硬编码为日期 2010 年 2 月 1 日。在下载这段代码时，很容易修改这个日期：

```
internal class WhatsNewChecker
{
    private static readonly StringBuilder outputText = new StringBuilder(1000);
    private static DateTime backDateTo = new DateTime(2010, 2, 1);

    static void Main()
    {
        Assembly theAssembly = Assembly.Load("VectorClass");
        Attribute supportsAttribute =
            Attribute.GetCustomAttribute(
                theAssembly, typeof(SupportsWhatsNewAttribute));
        string name = theAssembly.FullName;
```

```

        AddToMessage("Assembly: " + name);

        if (supportsAttribute == null)
        {
            AddToMessage(
                "This assembly does not support WhatsNew attributes");
            return;
        }
        else
        {
            AddToMessage("Defined Types:");
        }

        Type[] types = theAssembly.GetTypes();

        foreach (Type definedType in types)
            DisplayTypeInfo(definedType);

        MessageBox.Show(outputText.ToString(),
            "What's New since " + backDateTo.ToLongDateString());
        Console.ReadLine();
    }

```

Main()方法首先加载 **VectorClass** 程序集，验证它是否真的用 **SupportsWhatsNew** 特性标记。我们知道，**VectorClass** 程序集应用了 **SupportsWhatsNew** 特性，虽然才编译了该程序集，但进行这种检查还是必要的，因为用户可能希望检查这个程序集。

验证了这个程序集后，使用 **Assembly.GetTypes()**方法获得一个数组，其中包括在该程序集中定义的所有类型，然后在这个数组中遍历它们。对每种类型调用一个方法——**DisplayTypeInfo()**，它给 **outputText** 字段添加相关的文本，包括 **LastModifiedAttribute** 类的任何实例的详细信息。最后，显示带有完整文本的消息框。**DisplayTypeInfo()**方法如下所示：

```

private static void DisplayTypeInfo(Type type)
{
    // make sure we only pick out classes
    if (!(type.IsClass))
    {
        return;
    }

    AddToMessage("\nclass " + type.Name);

    Attribute [] attribs = Attribute.GetCustomAttributes(type);

    if (attribs.Length == 0)
    {
        AddToMessage("No changes to this class\n");
    }
    else
    {
        foreach (Attribute attrib in attribs)
        {
            WriteAttributeInfo(attrib);
        }
    }
}

```

```

MethodInfo [] methods = type.GetMethods();
AddToMessage("CHANGES TO METHODS OF THIS CLASS:");

foreach (MethodInfo nextMethod in methods)
{
    object [] attribs2 =
        nextMethod.GetCustomAttributes(
            typeof(LastModifiedAttribute), false);

    if (attribs2 != null)
    {
        AddToMessage(
            nextMethod.ReturnType + " " + nextMethod.Name + "()");
        foreach (Attribute nextAttrib in attribs2)
        {
            WriteAttributeInfo(nextAttrib);
        }
    }
}
}

```

注意,在这个方法中,首先应检查所传递的 `Type` 引用是否表示一个类。因为,为了简化代码,指定 `LastModified` 特性只能应用于类或成员方法,如果该引用不是类(它可能是一个结构、委托或枚举),那么进行任何处理都是浪费时间。

接着使用 `Attribute.GetCustomAttributes()` 方法确定这个类是否有相关的 `LastModifiedAttribute` 实例。如果有,就使用辅助方法 `WriteAttributeInfo()` 把它们的详细信息添加到输出文本中。

最后使用 `Type.GetMethods()` 方法遍历这种数据类型的所有成员方法;然后对每个方法进行相同的处理(类似于对类执行的操作):检查每个方法是否有相关的 `LastModifiedAttribute` 实例,如果有,就用 `WriteAttributeInfo()` 方法显示它们。

下面的代码显示了 `WriteAttributeInfo()` 方法,它负责确定为给定的 `LastModifiedAttribute` 实例显示什么文本,注意因为这个方法的参数是一个 `Attribute` 引用,所以需要先把该引用强制转换为 `LastModifiedAttribute` 引用。之后,就可以使用最初为这个特性定义的属性获取其参数。在把该特性添加到要显示的文本中之前,应检查特性的日期是否是最近的:

```

private static void WriteAttributeInfo(Attribute attrib)
{
    LastModifiedAttribute lastModifiedAttrib =
        attrib as LastModifiedAttribute;

    if (lastModifiedAttrib == null)
    {
        return;
    }

    // check that date is in range
    DateTime modifiedDate = lastModifiedAttrib.DateModified;

    if (modifiedDate < backDateTo)
    {
        return;
    }
}

```

```

AddToMessage(" MODIFIED: " +
    modifiedDate.ToLongDateString() + ":");
AddToMessage(" " + lastModifiedAttrib.Changes);

if (lastModifiedAttrib.Issues != null)
{
    AddToMessage(" Outstanding issues:" +
        lastModifiedAttrib.Issues);
}
}

```

最后，是辅助方法 AddToMessage():

```

static void AddToMessage(string message)
{
    outputText.Append("\n" + message);
}
}

```

运行这段代码，得到如图 14-2 所示的结果。

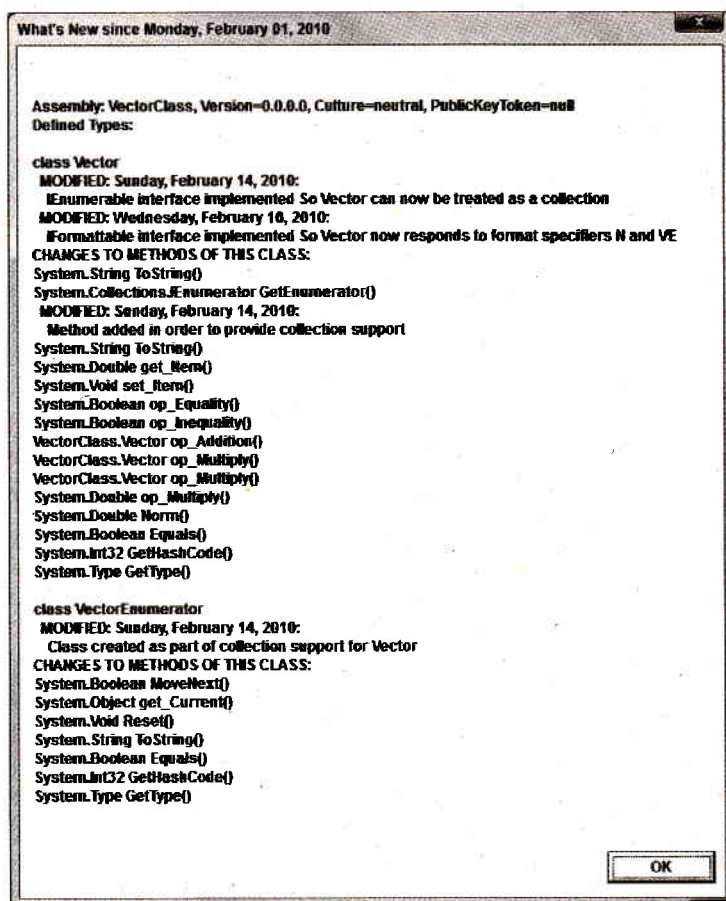


图 14-2

注意，在列出 VectorClass 程序集中定义的类型时，实际上选择了两个类：Vector 类和内嵌的

`VectorEnumerator` 类。还要注意, 这段代码把 `backDateTo` 日期硬编码为 2 月 1 日, 实际上选择的是日期为 2 月 14 日的特性(添加集合支持的时间), 而不是 2 月 10 日(添加 `IFormattable` 接口的时间)。

14.3 小结

本章没有介绍反射的全部内容, 反射需要一整本书来讨论。我们只介绍了 `Type` 和 `Assembly` 类, 它们是访问反射所提供的扩展功能的主要入口点。

另外, 本章还探讨了反射的一个常用方面: 自定义特性, 它比其他方面更常用。介绍了如何定义和应用自己的自定义特性, 以及如何在运行期间检索自定义特性的信息。

第 15 章介绍异常和结构化的异常处理。

第 15 章

错误和异常

本章内容:

- 异常类
- 使用 `try...catch...finally` 捕获异常
- 创建用户定义的异常

错误的出现并不总是编写应用程序的人的原因,有时应用程序会因为应用程序的最终用户引发或运行代码的环境而发生错误。无论如何,我们都应相应地预测应用程序和代码中出现的错误。

.NET Framework 改进了处理错误的方式。C#处理错误的机制可以为每种错误提供自定义处理方式,并把识别错误的代码与处理错误的代码分离开来。

学习完本章后,您将很好地掌握 C#应用程序中的高级异常处理技术。

无论编码技术有多好,程序都必须能处理可能出现的任何错误。例如,在一些复杂的代码处理过程中,代码没有读取文件的许可,或者在发送网络请求时,网络可能会中断。在这种异常情况下,方法只返回相应的错误代码通常是不够的——可能方法调用嵌套了 15 级或者 20 级,此时,程序需要跳过所有的 15 或 20 级方法调用,才能完全退出任务,并采取相应的应对措施。C#语言提供了处理这种情形的最佳工具,称为异常处理机制。



Java 和 C++开发人员会比较熟悉异常的规则,因为这些语言处理错误的方式与 C#相似。C++开发人员会留意异常是因为 C++可能会因此而降低性能,但在 C#中就不是这样。在 C#代码中使用异常一般不影响性能。Visual Basic 开发人员会发现,在 C#中处理异常非常类似于在 Visual Basic 中使用异常(除了语法不同)。

15.1 异常类

在 C#中,当出现某个特殊的异常错误条件时,就会创建(或抛出)一个异常对象。这个对象包含有助于跟踪问题的信息。我们可以创建自己的异常类(详见后面的内容),但.NET 提供了许多预定义的异常类。

本节将简要介绍在.NET 基类库中可用的一些异常。Microsoft 在.NET 中定义了大量的异常类,这里不可能提供详尽的列表。在图 15-1 类的层次结构图显示了其中的一些类,它们给出了大致的模式。

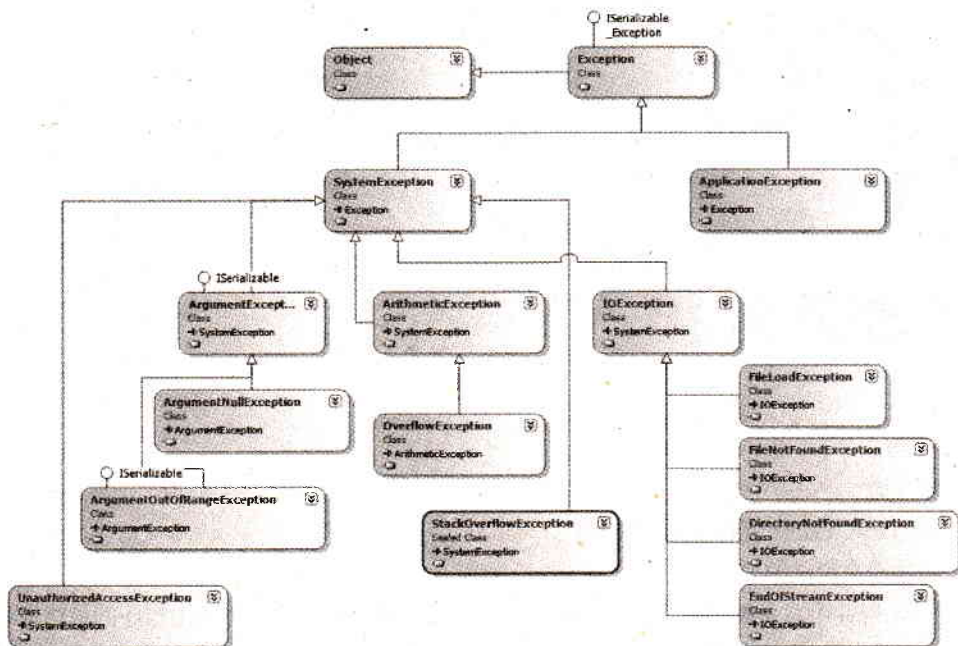


图 15-1

图 15-1 中的所有类都在 `System` 名称空间中，但 `IOException` 类和派生于 `IOException` 的类除外，它们在 `System.IO` 名称空间中。`System.IO` 名称空间处理文件数据的读写。一般情况下，异常没有特定的名称空间，异常类应放在生成异常的类所在的名称空间中，因此与 IO 相关的异常就在 `System.IO` 名称空间中。在许多基类名称空间中都有异常类。

对于 .NET 类，一般的异常类 `System.Exception` 派生自 `System.Object`，通常不在代码中抛出 `System.Exception` 泛型对象，因为它们无法确定错误情况的本质。

在该层次结构中两个重要的类，它们派生自 `System.Exception` 类：

- **System.SystemException**——该类用于通常由 .NET 运行库抛出的异常，或者有着非常一般的本质并且可以由几乎所有的应用程序抛出的异常。例如，如果 .NET 运行库检测到栈已满，它就会抛出 `StackOverflowException` 异常。另一方面，如果检测到调用方法时参数不正确，就可以在自己的代码中选择抛出 `ArgumentException` 异常或其子类异常。`System.SystemException` 异常的子类包括表示致命错误和非致命错误的异常。
- **System.ApplicationException**——这个类非常重要，因为它是第三方定义的异常基类。如果自己定义的任何异常覆盖了应用程序独有的错误情况，就应使它们直接或间接派生自 `System.ApplicationException` 异常。

其他可能用到的异常类包括：

- **StackOverflowException**——如果分配给栈的内存区域已满，就会抛出这个异常。如果一个方法连续地递归调用它自己，就可能发生栈溢出。这一般是一个致命错误，因为它禁止应用程序执行除了中断以外的其他任务。在这种情况下，甚至也不可能执行 `finally` 块，通常用户自己不能处理像这样的错误，而应退出应用程序。
- **EndOfStreamException**——这个异常通常是因为读到文件末尾而抛出的。第 24 章解释流，流表示数据源之间的数据流。

- **OverflowException**——如果要在 `checked` 环境下把包含值 -40 的 `int` 类型数据强制转换为 `uint` 数据，就会抛出这个异常。

我们打算讨论图 15-1 中的其他异常类。

异常类的层次结构并不多见，因为其中的大多数类并没有给它们的基类添加任何功能。但是在处理异常时，添加继承类的一般原因是更准确地指定错误，所以不需要重写方法或添加新方法(尽管常常要添加额外的属性，以包含有关错误情况的额外信息)。例如，当传递了不正确的参数值时，可给方法调用使用 **ArgumentException** 基类，**ArgumentNullException** 异常类派生于 **ArgumentException** 异常类，它专门用于处理所传递的参数值是 `Null` 的情况。

15.2 捕获异常

.NET Framework 提供了大量的预定义基类异常对象，该如何在代码中使用它们捕获错误情况？为了在 C# 代码中处理可能的错误情况，一般要把程序的相关部分分成 3 种不同类型的代码块：

- **try** 块包含的代码组成了程序的正常操作部分，但这部分程序可能遇到某些严重的错误。
- **catch** 块包含的代码处理各种错误情况，这些错误是执行 **try** 块中的代码时遇到的。这个块还可以用于记录错误。
- **finally** 块包含的代码清理资源或执行通常要在 **try** 块或 **catch** 块末尾执行的其他操作。无论是否抛出异常，都会执行 **finally** 块，理解这一点非常重要。因为 **finally** 块包含了应总是执行的清理代码，如果在 **finally** 块中放置了 `return` 语句，编译器就会标记一个错误。例如，使用 **finally** 块时，可以关闭在 **try** 块中打开的连接。**finally** 块是完全可选的。如果不需要清理代码(如删除对象或关闭已打开的对象)，就不需要包含此块。

那么，这些块是如何组合在一起捕获错误情况的？下面就是其步骤：

- (1) 执行的程序流进入 **try** 块。
- (2) 如果在 **try** 块中没有错误发生，在块中就会正常执行操作。当程序流到达 **try** 块末尾后，如果存在一个 **finally** 块，程序流就会自动进入 **finally** 块(第(5)步)。但如果在 **try** 块中程序流检测到一个错误，程序流就会跳转到 **catch** 块(下一步)。
- (3) 在 **catch** 块中处理错误。
- (4) 在 **catch** 块执行完后，如果存在一个 **finally** 块，程序流就会自动进入 **finally** 块：
- (5) 执行 **finally** 块(如果存在)。

用于完成这些任务的 C# 语法如下所示：

```
try
{
    // code for normal execution
}
catch
{
    // error handling
}
finally
{
    // clean up
}
```

实际上，上面的代码还有几种变体：

- 可以省略 `finally` 块，因为它是可选的
- 可以提供任意多个 `catch` 块，处理不同类型的错误。但不应包含过多的 `catch` 块，以防降低应用程序的性能。
- 可以一起省略 `catch` 块——此时，该语法应不是标识异常，而是一种确保程序流在离开 `try` 块后执行 `finally` 块中的代码的方式。如果在 `try` 块中有几个出口点，这很有用。

这看起来很不错，实际上是有问题的。如果运行 `try` 块中的代码，则程序流如何在错误发生时切换到 `catch` 块？如果检测到一个错误，代码就执行一定的操作，称为“抛出一个异常”；换言之，它实例化一个异常对象类，并抛出这个异常：

```
throw new OverflowException();
```

这里实例化了 `OverflowException` 类的一个异常对象。只要计算机在 `try` 块中遇到一条 `throw` 语句，就会立即查找与这个 `try` 块对应的 `catch` 块。如果有多个与 `try` 块对应的 `catch` 块，计算机就会查找与 `catch` 块对应的异常类，确定正确的 `catch` 块。例如，当抛出一个 `OverflowException` 异常对象时，执行的程序流就会跳转到下面的 `catch` 块：

```
catch (OverflowException ex)
{
    // exception handling here
}
```

换言之，计算机查找的 `catch` 块应表示同一个类(或基类)中匹配的异常类实例。

有了这些额外的信息，就可以扩展刚才介绍的 `try` 块。为了讨论方便，假定可能在 `try` 块中发生两个严重错误：溢出和数组超出范围。假定代码包含两个布尔变量 `Overflow` 和 `OutOfBounds`，它们分别表示这两种错误情况是否存在。我们知道，存在表示溢出的预定义溢出异常类 `OverflowException`；同样，存在预定义的 `IndexOutOfRangeException` 异常类，用于处理超出范围的数组。

现在，`try` 块如下所示：

```
try
{
    // code for normal execution

    if (Overflow == true)
    {
        throw new OverflowException();
    }

    // more processing

    if (OutOfBounds == true)
    {
        throw new IndexOutOfRangeException();
    }

    // otherwise continue normal execution
}
catch (OverflowException ex)
```

```

{
    // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
    // error handling for the index out of range error condition
}
finally
{
    // clean up
}

```

我们得到的 `try` 块看起来并不比 Visual Basic 6 中的 `On Error GoTo` 语句强多少, 但可以更清晰地 将不同的代码段分开。实际上, C# 的错误处理有一个更强大、更灵活的机制。

这是因为 `throw` 语句可以嵌套在 `try` 块的几个方法调用中, 甚至在程序流进入其他方法时, 也会 继续执行同一个 `try` 块。如果计算机遇到一条 `throw` 语句, 就会立即退出栈上所有的方法调用, 查找 `try` 块的结尾和合适的 `catch` 块的开头, 此时, 中间方法调用中的所有局部变量都会超出作用域。 `try...catch` 结构最适合于本节开头描述的场所: 错误发生在一个方法调用中, 而该方法调用可能嵌套 了 15 或 20 级, 这些处理操作会立即停止。

从上面的论述可以看出, `try` 块在控制执行的程序流上有重要的作用。但是, 异常是用于处理异常情 况的, 这是其名称的由来。不应该用异常来控制退出 `do...while` 循环的时间。

15.2.1 实现多个 catch 块

要了解 `try...catch...finally` 块是如何工作的, 最简单的方式是用两个示例来说明。第一个示例是 `SimpleExceptions`。它多次要求用户输入一个数字, 然后显示这个数字。为了便于解释这个示例, 假 定该数字必须在 0~5 之间, 否则程序就不能对该数字进行正确的处理。所以, 如果用户输入超出该 范围的数字, 程序就抛出一个异常。

程序会继续要求用户输入更多数字, 直到用户不再输入任何内容, 按回车键为止。



这段代码没有说明何时使用异常处理。前面已经提及, 异常用于处理异常情况。 用户总是输入一些无聊的东西, 所以这种情况不会真正发生。正常情况下, 程序会处 理不正确的用户输入, 方法是进行即时检查, 如果有问题, 就要求用户重新输入。但 是, 在一个要求几分钟内读懂的小示例中生成异常是比较困难的, 为了描述异常是如 何工作的, 后面将使用更真实的示例。

`SimpleExceptions` 的代码如下所示:



可从
wrox.com
下载源代码

```

using System;

namespace Wrox.ProCSharp.AdvancedCSharp
{
    public class MainEntryPoint
    {
        public static void Main()
        {
            while (true)
            {

```