

调用这个公共初始化构造器。这样能减少生成的代码。下例演示了如何在 C# 中利用 **this** 关键字显式调用另一个构造器：

```
using System;
internal sealed class SomeType {
    // 不要显式初始化下面的字段
    private Int32    m_x;
    private String   m_s;
    private Double   m_d;
    private Byte     m_b;

    // 该构造器将所有字段都设为默认值,
    // 其他所有构造器都显式调用该构造器
    public SomeType() {
        m_x = 5;
        m_s = "Hi there";
        m_d = 3.14159;
        m_b = 0xff;
    }

    // 该构造器将所有的字段都设为默认值, 然后修改 m_x
    public SomeType(Int32 x) : this() {
        m_x = x;
    }

    // 该构造器将所有的字段都设为默认值, 然后修改 m_s
    public SomeType(String s) : this() {
        m_s = s;
    }

    // 该构造器首先将所有字段都设为默认值, 然后修改 m_x 和 m_s
    public SomeType(Int32 x, String s) : this() {
        m_x = x;
        m_s = s;
    }
}
```

8.2 实例构造器和结构(值类型)

值类型(**struct**)构造器的工作方式与引用类型(**class**)的构造器截然不同。CLR 总是允许创建值类型的实例, 并且没有办法阻止值类型的实例化。所以, 值类型其实并不需要定义构造器, C#编译器根本不会为值类型内联(嵌入)默认的空参构造器。来看下面的代码:

```
internal struct Point {
    public Int32 m_x, m_y;
}
internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;
}
```

为了构造一个 **Rectangle**, 必须使用 **new** 操作符, 而且必须指定构造器。在这个例子中, 调用的是 C#编译器自动生成的默认构造器。为 **Rectangle** 分配内存时, 内存中包含 **Point** 值类型的两个实例。考虑到性能, CLR 不会为包含在引用类型中的每个值类型字段都主动调用构造器。但是, 如前所述, 值类型的字段会被初始化为 **0** 或 **null**。

CLR 确实允许为值类型定义构造器。但必须显式调用才会执行。下面是一个例子。

```
internal struct Point {
```

```

public Int32 m_x, m_y;

public Point(Int32 x, Int32 y) {
    m_x = x;
    m_y = y;
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
        // 在 C# 中, 向一个值类型应用关键字 new,
        // 可以调用构造器来初始化值类型的字段
        m_topLeft = new Point(1, 2);
        m_bottomRight = new Point(100, 200);
    }
}

```

值类型的实例构造器只有显式调用才会执行。因此, 如果 **Rectangle** 的构造器没有使用 **new** 操作符来调用 **Point** 的构造器, 从而初始化 **Rectangle** 的 **m_topLeft** 字段和 **m_bottomRight** 字段, 那么两个 **Point** 字段中的 **m_x** 和 **m_y** 字段都将为 0。

前面展示的 **Point** 值类型没有定义默认的空参构造器。现在进行如下改写:

```

internal struct Point {
    public Int32 m_x, m_y;

    public Point() {
        m_x = m_y = 5;
    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
    }
}

```

现在, 构造新的 **Rectangle** 类时, 两个 **Point** 字段中的 **m_x** 和 **m_y** 字段会被初始化成多少? 是 0 还是 5? (提示: 小心上当!)

许多开发人员(尤其是那些有 C++ 背景的)都觉得 C# 编译器会在 **Rectangle** 的构造器中生成代码, 为 **Rectangle** 的两个字段自动调用 **Point** 的默认空参构造器。但是, 为了增强应用程序的运行时性能, C# 编译器不会自动生成这样的代码。实际上, 即便值类型提供了空参构造器, 许多编译器也永远不会生成代码来自动调用它。为了执行值类型的空参构造器, 开发人员必须增加显式调用值类型构造器的代码。

基于前面这一段的信息, 可以确定在前面示例代码中, **Rectangle** 类的两个 **Point** 字段的 **m_x** 和 **m_y** 字段会初始化为 0, 因为代码中没有在任何地方显式调用了 **Point** 的构造器。

但我说过, 这是一个容易让人上当的问题。这里的关键在于 C# 编译器不允许值类型定义空参构造器。所以, 前面的代码实际是编译不了的。试图编译上述代码时, C# 编译器会显示以下消息: **error CS0568: 结构不能包含显式的无参数构造器。**

C# 编译器故意不允许值类型定义空参构造器, 目的是防止开发人员对这种构造器在什么时

候调用产生迷惑。由于不能定义无参构造器，所以编译器永远不会生成自动调用它的代码。没有无参构造器，值类型的字段总是被初始化为 0 或 null。



注意 严格地说，只有当值类型的字段嵌套到引用类型中时，才保证被初始化为 0 或 null。基于栈的值类型字段则无此保证。但是，为了确保代码的“可验证性”(verifiability)，任何基于栈的值类型字段都必须在读取之前写入(赋值)。允许先读再写会造成安全漏洞。对于所有基于栈的值类型中的字段，C#和其他能生成“可验证”代码的编译器可以保证对它们进行“置零”，或至少保证在读取之前赋值，确保不会在运行时因验证失败而抛出异常。所以，你完全可以忽略本“注意”的内容，假定自己的值类型的字段都会被初始化为 0 或 null。

注意，虽然 C#不允许值类型带有无参构造器，但 CLR 允许。所以，如果不在乎前面描述的问题，可以使用另一编程语言(比如 IL 汇编语言)定义带有无参构造器的值类型。

由于 C#不允许为值类型定义无参构造器，所以编译以下类型时，C#编译器将显示消息：
error CS0573: "SomeValType.m_x": 结构中不能有实例字段初始值设定项。

```
internal struct SomeValType {
    // 不能在值类型中内联实例字段的初始化
    private Int32 m_x = 5;
}
```

另外，为了生成“可验证”代码，在访问值类型的任何字段之前，都需要对全部字段进行赋值。所以，值类型的任何构造器都必须初始化值类型的全部字段。以下类型为值类型定义了一个构造器，但没有初始化值类型的全部字段：

```
internal struct SomeValType {
    private Int32 m_x, m_y;

    // C#允许为值类型定义有参构造器
    public SomeValType(Int32 x) {
        m_x = x;
        // 注意 m_y 没有在这里初始化
    }
}
```

编译上述类型，C#编译器会显示消息：**error CS0171: 在控制返回到调用方之前，字段 "SomeValType.m_y" 必须完全赋值。**为了修正这个问题，需要在构造器中为 y 赋一个值(通常是 0)。下面是对值类型的全部字段进行赋值的一个替代方案：

```
// C#允许为值类型定义有参构造器
public SomeValType(Int32 x) {
    // 看起来很奇怪，但编译没问题，会将所有字段初始化为 0/null
    this = new SomeValType();

    m_x = x; // 用 x 覆盖 m_x 的 0
    // 注意 m_y 已初始化为 0
}
```

在值类型的构造器中，**this** 代表值类型本身的一个实例，用 **new** 创建的值类型的一个实例

可以赋给 **this**。在 **new** 的过程中，会将所有字段置为零。而在引用类型的构造器中，**this** 被认为是只读的，所以不能对它进行赋值。

8.3 类型构造器

除了实例构造器，CLR 还支持类型构造器(type constructor)，也称为静态构造器(static constructor)、类构造器(class constructor)或者类型初始化器(type initializer)。类型构造器可应用于接口(虽然 C#编译器不允许)、引用类型和值类型。实例构造器的作用是设置类型的实例的初始状态。对应地，类型构造器的作用是设置类型的初始状态。类型默认没有定义类型构造器。如果定义，也只能定义一个。此外，类型构造器永远没有参数。以下代码演示了如何在 C#中为引用类型和值类型定义一个类型构造器：

```
internal sealed class SomeRefType {
    static SomeRefType() {
        // SomeRefType 被首次访问时，执行这里的代码
    }
}

internal struct SomeValType{
    // C#允许值类型定义无参的类型构造器
    static SomeValType() {
        // SomeValType 被首次访问时，执行这里的代码
    }
}
```

可以看出，定义类型构造器类似于定义无参实例构造器，区别在于必须标记为 **static**。此外，类型构造器总是私有；C#自动把它们标记为 **private**。事实上，如果在源代码中显式将类型构造器标记为 **private**(或其他访问修饰符)，C#编译器会显示以下消息：**error CS0515: 静态构造函数中不允许出现访问修饰符**。之所以必须私有，是为了防止任何由开发人员写的代码调用它，对它的调用总是由 CLR 负责。

类型构造器的调用比较麻烦。JIT 编译器在编译一个方法时，会查看代码中都引用了哪些类型。任何一个类型定义了类型构造器，JIT 编译器都会检查针对当前 AppDomain，是否已经执行了这个类型构造器。如果构造器从未执行，JIT 编译器会在它生成的本机(native)代码中添加对类型构造器的调用。如果类型构造器已经执行，JIT 编译器就不添加对它的调用，因为它知道类型已经初始化好了。

现在，当方法被 JIT 编译完毕之后，线程开始执行它，最终会执行到调用类型构造器的代码。事实上，多个线程可能同时执行相同的方法。CLR 希望确保在每个 AppDomain 中，一个类型构造器只执行一次。为了保证这一点，在调用类型构造器时，调用线程要获取一个互斥线程同步锁。这样一来，如果多个线程试图同时调用某个类型的静态构造器，只有一个线程才可以获得锁，其他线程会被阻塞(blocked)。第一个线程会执行静态构造器中的代码。当第一个线程离开构造器后，正在等待的线程将被唤醒，然后发现构造器的代码已被执行过。因此，这些线程不会再次执行代码，将直接从构造器方法返回。除此之外，如果再次调用这样的方法，CLR 知道类型构造器已被执行过，从而确保构造器不被再次调用。



重要提示 虽然能在值类型中定义类型构造器，但永远都不要真的那么做，因为 CLR 有时不会调用值类型的静态类型构造器。下面是一个例子：

```
internal struct SomeValType {
    static SomeValType() {
        Console.WriteLine("这句话永远不会显示");
    }
    public Int32 m_x;
}

public sealed class Program {
    public static void Main() {
        SomeValType[] a = new SomeValType[10];
        a[0].m_x = 123;
        Console.WriteLine(a[0].m_x); // 显示 123
    }
}
```



注意 由于 CLR 保证一个类型构造器在每个 AppDomain 中只执行一次，而且(这种执行)是线程安全的，所以非常适合在类型构造器中初始化类型需要的任何单实例(Singleton)对象^①。

单个线程中的两个类型构造器包含相互引用的代码可能出问题。例如，假定 **ClassA** 的类型构造器包含了引用 **ClassB** 的代码，**ClassB** 的类型构造器包含了引用 **ClassA** 的代码。在这种情况下，CLR 仍然保证每个类型构造器的代码只被执行一次；但是，完全有可能在 **ClassA** 的类型构造器还没有执行完毕的前提下，就开始执行 **ClassB** 的类型构造器。因此，应尽量避免写会造成这种情况的代码。事实上，由于是 CLR 负责类型构造器的调用，所以任何代码都不应要求以特定的顺序调用类型构造器。

最后，如果类型构造器抛出未处理的异常，CLR 会认为类型不可用。试图访问该类型的任何字段或方法都会抛出 **System.TypeInitializationException** 异常。

类型构造器中的代码只能访问类型的静态字段，并且它的常规用途就是初始化这些字段。和实例字段一样，C# 提供了一个简单的语法来初始化类型的静态字段：

```
internal sealed class SomeType {
    private static Int32 s_x = 5;
}
```



注意 虽然 C# 不允许值类型为它的实例字段使用内联字段初始化语法，但可以为静态字段使用。换句话说，如果将前面定义的 **SomeType** 类型从 **class** 改为 **struct**，那么代码也能通过编译，而且会像你预期的那样工作。

① 每个 AppDomain 只能有一个实例，这种类型就是单实例类型。——译注

生成上述代码时，编译器自动为 **SomeType** 生成一个类型构造器，好像源代码本来是这样写的：

```
internal sealed class SomeType {
    private static Int32 s_x;
    static SomeType() { s_x = 5; }
}
```

使用 **ILDasm.exe** 查看类型构造器的 IL，很容易验证编译器实际生成的东西。类型构造器方法总是叫 **.cctor** (代表 class constructor)。

在以下代码中，可以看到 **.cctor** 方法是 **private** 和 **static** 的。另外，注意方法中的代码确实将值 **5** 加载到静态字段 **s_x** 中。

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size 7 (0x7)
    .maxstack 8
    IL_0000: ldc.i4.5
    IL_0001: stsfld      int32 SomeType::s_x
    IL_0006: ret
} // end of method SomeType::.cctor
```

类型构造器不应调用基类型的类型构造器。这种调用之所以没必要，是因为类型不可能有静态字段是从基类型分享或继承的。



注意 有的语言(比如 Java)希望在访问类型时自动调用它的类型构造器，并调用它的所有基类型的类型构造器。此外，类型实现的接口也必须调用接口的类型构造器。CLR 不支持这种行为。但是，使用由 **System.Runtime.CompilerServices.RuntimeHelpers** 提供的 **RunClassConstructor** 方法，编译器和开发人员可以实现这种行为。任何语言想要实现这种行为，可以告诉它的编译器在一个类型的类型构造器中生成代码，为所有基类型都调用这个方法。用 **RunClassConstructor** 方法调用一个类型构造器，CLR 就知道类型构造器之前是否执行过。如果是，CLR 不会再次调用它。

最后，假定有以下代码：

```
internal sealed class SomeType {
    private static Int32 s_x = 5;

    static SomeType() {
        s_x = 10;
    }
}
```

在这个例子中，C#编译器只生成一个类型构造器方法。它首先将 **s_x** 初始化为 **5**，再把它修改成 **10**。换言之，当 C#编译器为类型构造器生成 IL 代码时，它首先生成的是初始化静态字段所需的代码，然后才会添加你的类型构造器方法中显式包含的代码。



重要提示 偶尔有开发人员问我,是否可以在卸载类型时执行一些代码。首先要搞清楚的是,类型只有在 AppDomain 卸载时才会卸载。AppDomain 卸载时,用于标识类型的对象(类型对象)将成为“不可达”的对象(不存在对它的引用),垃圾回收器会回收类型对象的内存。这个行为导致许多开发人员认为可以为类型添加一个静态 **Finalize** 方法。当类型卸载时,就自动地调用这个方法。遗憾的是,CLR 并不支持静态 **Finalize** 方法。但也不是完全没有办法,要在 AppDomain 卸载时执行一些代码,可向 **System.AppDomain** 类型的 **DomainUnload** 事件登记一个回调方法。

8.4 操作符重载方法

有的语言允许类型定义操作符应该如何操作类型的实例。例如,许多类型(比如 **System.String**)都重载了相等(==)和不等(!=)操作符。CLR 对操作符重载一无所知,它甚至不知道什么是操作符。是编程语言定义了每个操作符的含义,以及当这些特殊符号出现时,应该生成什么样的代码。

例如在 C# 中,向基元(类型的)数字应用+符号,编译器生成将两个数加到一起的代码。将+符号应用于 **String** 对象,C#编译器生成将两个字符串连接到一起的代码。测试不等性时,C#使用!=符号,而 Microsoft Visual Basic 使用<>。最后,^在 C#中的含义为异或(XOR),在 Visual Basic 中则为求幂。

虽然 CLR 对操作符一无所知,但它确实规定了语言应如何公开操作符重载,以便由另一种语言的代码使用。每种编程语言都要自行决定是否支持操作符重载。如果决定支持,还要决定用什么语法来表示和使用它们。至于 CLR,操作符重载只是方法而已。

对编程语言的选择决定了你是否获得对操作符重载的支持,以及具体的语法是什么。编译源代码时,编译器会生成一个标识操作符行为的方法。CLR 规范要求操作符重载方法必须是 **public** 和 **static** 方法。另外,C#(以及其他许多语言)要求操作符重载方法至少有一个参数的类型与当前定义这个方法类型相同。之所以要进行这样的限制,是为了使 C#编译器能在合理的时间内找到要绑定的操作符方法。

以下 C#代码展示了一个在类中定义的操作符重载方法:

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) {...}
}
```

编译器为名为 **op_Addition** 的方法生成元数据方法定义项;这个方法定义项还设置了 **specialname** 标志,表明这是一个“特殊”方法。编程语言的编译器(包括 C#编译器)看到源代码中出现一个+操作符时,会检查是否有一个操作数的类型定义了名为 **op_Addition** 的 **specialname** 方法,而且该方法的参数兼容于操作数的类型。如果存在这样的方法,编译器就生成调用它的代码。不存在这样的方法就报告编译错误。

表 8-1 和表 8-2 总结了 C# 允许重载的一元和二元操作符，以及由编译器生成的对应的 CLS(Common Language Specification, 公共语言规范)方法名。下一节会解释表的第 3 列。

表 8-1 C#的一元操作符及其相容于 CLS 的方法名

C#操作符	特殊方法名	推荐的相容于 CLS 的方法名
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
!	op_LogicalNot	Not
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
(无)	op_True	IsTrue { get; }
(无)	op_False	IsFalse { get; }

表 8-2 C#的二元操作符及其相容于 CLS 的方法名

C#操作符	特殊方法名	推荐的相容于 CLS 的方法名
+	op_Addition	Add
-	op_Subtraction	Subtract
*	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod
&	op_BitwiseAnd	BitwiseAnd
	op_BitwiseOr	BitwiseOr
^	op_ExclusiveOr	Xor
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
=	op_Equality	Equals
!=	op_Inequality	Equals
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare

CLR 规范定义了许多额外的可重载的操作符,但 C#不支持这些额外的操作符。由于是非主流,所以此处不列出它们。如果对完整的列表感兴趣,请访问 CLI 的 ECMA 规范 (www.ecma-international.org/publications/standards/Ecma-335.htm)并阅读 Partition 1: Concepts and Architecture 的 Section 10.3.1(一元操作符)和 Section 10.3.2(二元操作符)。



注意 检查 Framework 类库(FCL)的核心数值类型(**Int32**, **Int64** 和 **UInt32** 等),会发现它们没有定义任何操作符重载方法。之所以不定义,是因为编译器会(在代码中)专门查找针对这些基元类型执行的操作(运算),并生成直接操作这些类型的实例的 IL 指令。如果类型要提供方法,而且编译器要生成代码来调用这些方法,方法调用就会产生额外的运行时开销。另外,方法最终都要执行一些 IL 指令来完成你希望的操作。这正是核心 FCL 类型没有定义任何操作符重载方法的原因。对于开发人员,这意味着假如选择的编程语言不支持其中的某个 FCL 类型,便不能对该类型的实例执行任何操作。

操作符和编程语言互操作性

操作符重载是很有用的工具,允许开发人员用简洁的代码表达自己的想法。但并不是所有编程语言都支持操作符重载。使用不支持操作符重载的语言时,语言不知道如何解释+操作符(除非类型是该语言的基元类型),编译器会报错。使用不支持操作符重载的编程语言时,语言应该允许你直接调用希望的 **op_***方法(例如 **op_Addition**)。

如果语言不支持在类型中定义+操作符重载,这个类型仍然可能提供了一个 **op_Addition** 方法。在这种情况下,可不可以 C#中使用+操作符来调用这个 **op_Addition** 方法呢?答案是否定的。C#编译器检测到操作符+时,会查找关联了 **specialname** 元数据标志的 **op_Addition** 方法,以确定 **op_Addition** 方法是要作为操作符重载方法使用。但由于现在这个 **op_Addition** 方法是由不支持操作符重载的编程语言生成的,所以方法没有关联 **specialname** 标记。因此,C#编译器会报告编译错误。当然,用任何编程语言写的代码都可以显式调用碰巧命名为 **op_Addition** 的方法,但编译器不会将一个+号的使用翻译成对这个方法的调用。



注意 FCL 的 **System.Decimal** 类型很好地演示了如何重载操作符并根据 Microsoft 的设计规范定义友好方法名。

Microsoft 操作符方法的命名规则之我见

操作符重载方法什么时候能调用，什么时候不能调用，这些规则会使人感觉非常迷惑。如果支持操作符重载的编译器不生成 **specialname** 元数据标记，规则会简单得多，而且开发人员使用提供了操作符重载方法的类型也会更轻松。支持操作符重载的语言都支持操作符符号语法，而且所有语言都支持显式调用各种 **op_**方法。我不理解为什么 Microsoft 非要把它搞得这么复杂。希望 Microsoft 在编译器未来的版本中放宽这些限制。

如果一个类型定义了操作符重载方法，Microsoft 还建议类型定义更友好的公共静态方法，并在这种方法的内部调用操作符重载方法。例如，根据 Microsoft 的设计规范，重载了 **op_Addition** 方法的类型应定义一个公共的、名字更友好的 **Add** 方法。表 8-1 和表 8-2 中的第 3 列展示了每个操作符推荐使用的友好名称。因此，前面的示例类型 **Complex** 应该像下面这样定义：

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
    public static Complex Add(Complex c1, Complex c2) {return(c1+c2); }
}
```

用任何语言写的代码都能调用 **Add** 这样的友好操作符方法，这一点毋庸置疑。Microsoft 要求类型提供友好方法名的设计规范使局面进一步复杂化。在我看来，这种额外的复杂性完全没必要。而且，除非 JIT 编译器能内联(直接嵌入)友好方法的代码，否则调用它们将导致额外的性能损失。内联代码可使 JIT 编译器优化代码，移除额外的方法调用，并提升运行时性能。

8.5 转换操作符方法

有时需要将对象从一种类型转换为另一种类型(例如将 **Byte** 转换为 **Int32**)。当源类型和目标类型都是编译器识别的基元类型时，编译器自己就知道如何生成转换对象所需的代码。

如果源类型或目标类型不是基元类型，编译器会生成代码，要求 CLR 执行转换(强制转型)。这种情况下，CLR 只是检查源对象的类型和目标类型(或者从目标类型派生的其他类型)是不是相同。但有时需要将对象从一种类型转换成全然不同的其他类型。例如，**System.Xml.Linq.XElement** 类允许将 XML 元素转换成 **Boolean**, **(U)Int32**, **(U)Int64**, **Single**, **Double**, **Decimal**, **String**, **DateTime**, **DateTimeOffset**, **TimeSpan**, **Guid** 或者所有这些类型(**String** 除外)的可空版本。另外，假设 FCL 包含了一个 **Rational**(有理数)类型，那么如果能将 **Int32** 或 **Single** 转换成 **Rational** 会显得很方便；反之亦然。

为了进行这样的转换，**Rational** 类型应该定义只有一个参数的公共构造器，该参数要求是源类型的实例。还应该定义无参的公共实例方法 **ToXxx**(类似于你熟悉的 **ToString** 方法)，

每个方法都将定义类型^①的实例转换成 **Xxx** 类型。以下代码展示了如何为 **Rational** 类型正确定义转换构造器和方法。

```
public sealed class Rational {
    // 由一个 Int32 构造一个 Rational
    public Rational(Int32 num){ ... }

    // 由一个 Single 构造一个 Rational
    public Rational(Single num){ ... }

    // 将一个 Rational 转换成一个 Int32
    public Int32 ToInt32(){ ... }

    // 将一个 Rational 转换成一个 Single
    public Single ToSingle(){ ... }
}
```

调用这些构造器和方法，使用任何编程语言的开发人员都能将 **Int32** 或 **Single** 对象转换成 **Rational** 对象，反之亦然。这些转换能给编程带来很多方便。设计类型时，应认真考虑类型需要支持的转换构造器和方法。

上一节讨论了某些编程语言如何提供操作符重载。事实上，有些编程语言(比如 C#)还提供了转换操作符重载。**转换操作符**是将对象从一种类型转换成另一种类型的方法。可以使用特殊的语法来定义转换操作符方法。CLR 规范要求转换操作符重载方法必须是 **public** 和 **static** 方法。此外，C#(以及其他许多语言)要求参数类型和返回类型二者必有其一与定义转换方法的类型相同。之所以要进行这个限制，是为了使 C#编译器能在一个合理的时间找到要绑定的操作符方法。以下代码为 **Rational** 类型添加了 4 个转换操作符方法：

```
public sealed class Rational {
    // 由一个 Int32 构造一个 Rational
    public Rational(Int32 num){ ... }

    // 由一个 Single 构造一个 Rational
    public Rational(Single num){ ... }

    // 将一个 Rational 转换成一个 Int32
    public Int32 ToInt32(){ ... }

    // 将一个 Rational 转换成一个 Single
    public Single ToSingle(){ ... }

    // 由一个 Int32 隐式构造并返回一个 Rational
    public static implicit operator Rational(Int32 num) {
        return new Rational(num);
    }

    // 由一个 Single 隐式构造并返回一个 Rational
    public static implicit operator Rational(Single num) {
        return new Rational(num);
    }
}
```

① 定义该方法的类型。——译注

```
// 由一个 Rational 显式返回一个 Int32
public static explicit operator Int32(Rational r) {
    return r.ToInt32();
}

// 由一个 Rational 显式返回一个 Single
public static explicit operator Single(Rational r) {
    return r.ToSingle();
}
}
```

对于转换操作符方法，编译器既可生成代码来隐式调用转换操作符方法，也可只有在源代码进行了显式转型时才生成代码来调用转换操作符方法。在 C# 中，**implicit** 关键字告诉编译器为了生成代码来调用方法，不需要在源代码中进行显式转型。相反，**explicit** 关键字告诉编译器只有在发现了显式转型时，才调用方法。

在 **implicit** 或 **explicit** 关键字之后，要指定 **operator** 关键字告诉编译器该方法是一个转换操作符。在 **operator** 之后，指定对象要转换成什么类型。在圆括号内，则指定要从什么类型转换。

像前面那样为 **Rational** 类型定义了转换操作符之后，就可以写出像下面这样的 C# 代码：

```
public sealed class Program {
    public static void Main() {
        Rational r1 = 5;           // Int32 隐式转型为 Rational
        Rational r2 = 2.5F;        // Single 隐式转型为 Rational

        Int32 x = (Int32) r1;      // Rational 显式转型为 Int32
        Single s = (Single) r2;    // Rational 显式转型为 Single
    }
}
```

在幕后，C# 编译器检测到代码中的转型，并内部生成 IL 代码来调用 **Rational** 类型定义的转换操作符方法。现在的问题是，这些方法的名称是什么？编译 **Rational** 类型并查看元数据，会发现编译器为定义每个转换操作符都生成了一个方法。**Rational** 类型的 4 个转换操作符方法的元数据如下：

```
public static Rational    op_Implicit(Int32 num)
public static Rational    op_Implicit(Single num)
public static Int32       op_Explicit(Rational r)
public static Single      op_Explicit(Rational r)
```

可以看出，将对象从一种类型转换成另一种类型的方法总是叫做 **op_Implicit** 或者 **op_Explicit**。只有在转换不损失精度或数量级的前提下(比如将个 **Int32** 转换成 **Rational**)，才能定义隐式转换操作符。如果转换会造成精度或数量级的损失(比如将 **Rational** 转换成 **Int32**)，就应该定义一个显式转换操作符。显式转换失败，应该让显式转换操作符方法抛出 **OverflowException** 或者 **InvalidOperationException** 异常。



注意 两个 `op_Explicit` 方法获取相同的参数，也就是一个 **Rational**。但两个方法的返回类型不同，一个是 **Int32**，另一个是 **Single**。这是仅凭返回类型来区分两个方法的例子。CLR 允许在一个类型中定义仅返回类型不同的多个方法。但只有极少数语言支持这个能力。你可能已经注意到了，C++、C#、Visual Basic 和 Java 语言都不允许在一个类型中定义仅返回类型不同的多个方法。个别语言(比如 IL 汇编语言)允许开发人员显式选择调用其中哪一个方法。当然，IL 汇编语言的程序员不应利用这个能力，否则定义的方法无法从其他语言中调用。虽然 C# 语言没有向 C# 程序员公开这个能力，但当一个类型定义了转换操作符方法时，C# 编译器会在内部利用这个能力。

C# 编译器提供了对转换操作符的完全支持。如果检测到代码中正在使用某个类型的对象，但实际期望的是另一种类型的对象，编译器就会查找能执行这种转换的隐式转换操作符方法，并生成代码来调用该方法。如果存在隐式转换操作符方法，编译器会在结果 IL 代码中生成对它的调用。如果编译器看到源代码是将对象从一种类型显式转换为另一种类型，就会查找能执行这种转换的隐式或显式转换操作符方法。如果找到一个，编译器就生成 IL 代码来调用它。如果没有找到合适的转换操作符方法，就报错并停止编译。



注意 使用强制类型转换表达式时，C# 生成代码来调用显式转换操作符方法。使用 C# 的 `as` 或 `is` 操作符时，则永远不会调用这些方法。(参见 4.2 节。)

为了真正理解操作符重载方法和转换操作符方法，强烈建议将 **System.Decimal** 类型作为典型来研究。**Decimal** 定义了几个构造器，允许将对象从各种类型转换为 **Decimal**。还定义了几个 **ToXxx** 方法，允许将 **Decimal** 转换成其他类型。最后，**Decimal** 类型还定义了几个转换操作符方法和操作符重载方法。

8.6 扩展方法

理解 C# 扩展方法最好的办法就是从例子中学习。14.3.2 节“**StringBuilder** 成员”会提到 **StringBuilder** 类提供的字符串处理方法比 **String** 类少。这是很奇怪的一件事，**StringBuilder** 类是可变的(**mutable**)，所以它应该是进行字符串处理的首选方式。现在，假定你想自己定义一些缺失的方法以方便操作 **StringBuilder**。例如，你也许想定义以下 **IndexOf** 方法：

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

```
    }
}
```

定义好这个方法后，可以在代码中使用它，如下所示：

```
StringBuilder sb=new StringBuilder("Hello. My name is Jeff."); //初始字符串
// 将句点更改为感叹号，获取!字符的索引(5)
Int32 index = StringExtensions.IndexOf(sb.Replace('.', '!'), '!');
```

上述代码工作起来没问题，但从程序员的角度看不理想。第一个问题是，要获取一个 **StringBuilder** 中的某个字符的索引，必须先知道 **StringExtensions** 类的存在。第二个问题是，代码没有反映出在 **StringBuilder** 对象上执行的操作的顺序，使代码很难写、读和维护。程序员希望先调用 **Replace**，再调用 **IndexOf**。但从左向右读最后一行代码，先看到的是 **IndexOf**，然后才看到 **Replace**。当然，可以像下面这样重写，使代码的行为看起来更容易理解：

```
// 首先，将句点更改成感叹号
sb.Replace('.', '!');

// 接着，获取!字符的索引(5)
Int32 index = StringExtensions.IndexOf(sb, '!');
```

但这两个版本都存在另一个不容忽视的问题，它影响了我们对代码行为的理解。使用 **StringExtensions** 显得“小题大做”，造成程序员无法专注于当前要执行的操作：**IndexOf**。如果 **StringBuilder** 类定义了自己的 **IndexOf** 方法，上述代码就可以重写为：

```
// 把句点更改为感叹号，获取!字符的索引(5)
Int32 index = sb.Replace('.', '!').IndexOf('!');
```

哇，是不是立刻就显得高端大气上档次了？一眼就能看出在 **StringBuilder** 对象中，是先将句点更改为感叹号，再获取感叹号的索引。

有了这个例子作为铺垫，就很容易理解 C# 扩展方法的意义了。它允许定义一个静态方法，并用实例方法的语法来调用。换言之，现在既能定义自己的 **IndexOf** 方法，又能避免上述三个问题。要将 **IndexOf** 方法转变成扩展方法，只需在第一个参数前添加 **this** 关键字：

```
public static class StringExtensions {
    public static Int32 IndexOf(this StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

现在，当编译器看到以下代码：

```
Int32 index = sb.IndexOf('X');
```

就首先检查 **StringBuilder** 类或者它的任何基类是否提供了获取单个 **Char** 参数、名为 **IndexOf** 的一个实例方法。如果是，就生成 IL 代码来调用它。如果没有找到匹配的实例方法，就继续检查是否有任何静态类定义了名为 **IndexOf** 的静态方法，方法的第一个参数的

类型和当前用于调用方法的那个表达式的类型匹配，而且该类型必须用 **this** 关键字标识。在本例中，表达式是 **sb**，类型是 **StringBuilder**。所以编译器会查找一个 **IndexOf** 方法，它有两个参数：一个 **StringBuilder**(用 **this** 关键字标记)，以及一个 **Char**。编译器找到了这个 **IndexOf** 方法，所以生成相应的 IL 代码来调用这个静态方法。

OK——这解释了编译器如何解决前面提到的、会影响代码理解的最后两个问题。但是，还没有说第一个问题是如何解决的：程序员怎么知道有这样的一个 **IndexOf** 方法，可以用它操作 **StringBuilder** 对象呢？这个问题是通过 Microsoft Visual Studio 的“智能感知”功能来解决的。在编辑器中输入句点符号，会弹出 Visual Studio 的“智能感知”窗口，列出当前可用的实例方法。现在，这个窗口还会列出可作用于句点左侧表达式类型的扩展方法。图 8-1 展示了 Visual Studio 的“智能感知”窗口；扩展方法的图标中有一个下箭头，方法旁边的“工具提示”表明该方法实际是一个扩展方法。这是相当实用的一个功能，因为现在可以轻松定义自己的方法来操作各种类型，其他程序员在使用这些类型的对象时，也能轻松地发现你的方法。

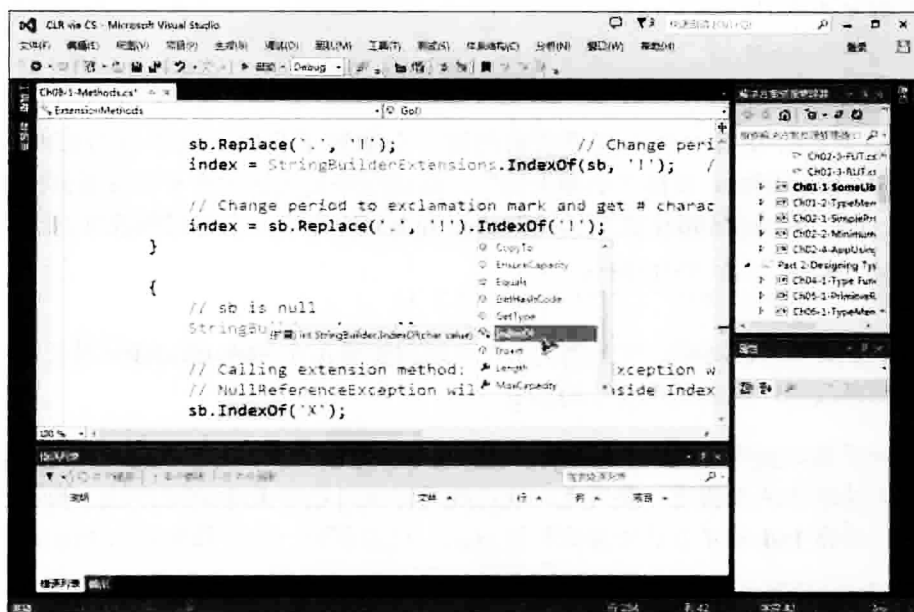


图 8-1 Visual Studio 的“智能感知”窗口能列出扩展方法

8.6.1 规则和原则

关于扩展方法，有一些附加的规则和原则需要注意。

- C#只支持扩展方法，不支持扩展属性、扩展事件、扩展操作符等。
- 扩展方法(第一个参数前面有 **this** 的方法)必须在非泛型的静态类中声明。然而，类名没有限制，可以随便叫什么名字。当然，扩展方法至少要有一个参数，而且只有第一个参数能用 **this** 关键字标记。

- C#编译器在静态类中查找扩展方法时，要求静态类本身必须具有文件作用域^①。如果静态类嵌套在另一个类中，C#编译器显示以下消息：**error CS1109: 扩展方法必须在顶级静态类中定义; StringBuilderExtensions 是嵌套类。**
- 由于静态类可以取任何名字，所以 C#编译器要花一定时间来寻找扩展方法，它必须检查文件作用域中的所有静态类，并扫描它们的所有静态方法来查找一个匹配。为增强性能，并避免找到非你所愿的扩展方法，C#编译器要求“导入”扩展方法。例如，如果有人 **Wintellect** 命名空间中定义了一个 **StringBuilderExtensions** 类，那么程序员为了访问这个类的扩展方法，必须在他的源代码文件顶部写一条 **using Wintellect;** 指令。
- 多个静态类可以定义相同的扩展方法。如果编译器检测到存在两个或多个扩展方法，就会显示以下消息：**error CS0121: 在以下方法或属性之间的调用不明确: "StringBuilderExtensions.IndexOf(string, char)" 和 "AnotherStringBuilderExtensions.IndexOf(string, char)"**。修正这个错误必须修改源代码。具体地说，不能再实例方法语法来调用这个静态方法。相反，必须使用静态方法语法。换言之，必须显式指定静态类的名称，明确告诉编译器要调用哪个方法。
- 使用这个功能须谨慎，一个原因是并非所有程序员都熟悉它。例如，用一个扩展方法扩展一个类型时，同时也扩展了派生类型。所以，不要将 **System.Object** 作为扩展方法的第一个参数，否则这个方法在所有表达式类型上都能调用，造成 Visual Studio 的“智能感知”窗口被填充太多垃圾信息。
- 扩展方法可能存在版本控制问题。如果 Microsoft 未来为他们的 **StringBuilder** 类添加了 **IndexOf** 实例方法，而且和我的代码调用的原型一样，那么在重新编译我的代码时，编译器会绑定到 Microsoft 的 **IndexOf** 实例方法，而不是我的静态 **IndexOf** 方法。这样我的程序就会有不同的行为。版本控制问题是使用扩展方法须谨慎的另一个原因。

8.6.2 用扩展方法扩展各种类型

前面演示了如何为 **StringBuilder** 类定义扩展方法。我要指出的一个问题是，由于扩展方法实际是对一个静态方法的调用，所以 CLR 不会生成代码对调用方法的表达式的值进行 **null** 值检查(不保证它非空):

```
// sb 是 null
StringBuilder sb = null;

// 调用扩展方法: NullReferenceException 异常不会在调用 IndexOf 时抛出,
// 相反, NullReferenceException 是在 IndexOf 内部的 for 循环中抛出的
sb.IndexOf('X');

// 调用实例方法: NullReferenceException 异常在调用 Replace 时抛出
sb.Replace('.', '!');
```

还要注意，可以为接口类型定义扩展方法，如下所示：

① 类要具有整个文件的作用域，而不能嵌套在某个类中而只具有该类的作用域。——译注

```
public static void ShowItems<T>(this IEnumerable<T> collection) {
    foreach (var item in collection)
        Console.WriteLine(item);
}
```

任何表达式，只要它最终的类型实现了 **IEnumerable<T>** 接口，就能调用上述扩展方法：

```
public static void Main() {
    // 每个 Char 在控制台上单独显示一行
    "Grant".ShowItems();

    // 每个 String 在控制台上单独显示一行
    new[] { "Jeff", "Kristin" }.ShowItems();

    // 每个 Int32 在控制台上单独显示一行
    new List<Int32>() { 1, 2, 3 }.ShowItems();
}
```



重要提示 扩展方法是 Microsoft 的 LINQ(Language Integrated Query, 语言集成查询)技术的基础。要想仔细研究提供了许多扩展方法的一个典型的类，请自行在文档中查看静态类 **System.Linq.Enumerable** 及其所有静态扩展方法。这个类中的每个扩展方法都扩展了 **IEnumerable** 或 **IEnumerable<T>** 接口。

还可为委托类型定义扩展方法，如下所示：

```
public static void InvokeAndCatch<TException>(this Action<Object> d, Object o)
    where TException : Exception {
    try { d(o); }
    catch (TException) { }
}
```

下面演示了如何调用它：

```
Action<Object> action = o => Console.WriteLine(o.GetType()); // 抛出 NullReferenceException
action.InvokeAndCatch<NullReferenceException>(null); // 吞噬 NullReferenceException
```

还可为枚举类型添加扩展方法(15.3 节“向枚举类型添加方法”展示了一个例子)。

最后，C#编译器允许创建委托(参见第 17 章“委托”)来引用一个对象上的扩展方法：

```
public static void Main () {
    // 创建一个 Action 委托 (实例) 来引用静态 ShowItems 扩展方法，
    // 并初始化第一个实参来引用字符串 "Jeff"
    Action a = "Jeff".ShowItems;
    .
    .
    // 调用 (Invoke) 委托，后者调用 (call)① ShowItems，
```

① “调用一个委托实例”中的“调用”对应的是 **invoke**，理解为“唤出”更恰当。它和后面的“在一个对象上调用方法”中的“调用”稍有不同，后者对应的是 **call**。在英语的语境中，**invoke** 和 **call** 的区别在于，在执行一个所有信息都已知的方法时，用 **call** 比较恰当。这些信息包括要引用的类型，方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用 **invoke** 就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译，只是在必要的时候附加英文原文提醒你区分。——译注

```
// 并向它传递对字符串"Jeff"的引用
a();
}
```

在上述代码中，C#编译器生成 IL 代码来构造一个 **Action** 委托。创建委托时，会向构造器传递应调用的方法，同时传递一个对象引用，这个引用应传给方法的隐藏 **this** 参数。正常情况下，创建引用静态方法的委托时，对象引用是 **null**，因为静态方法没有 **this** 参数。但在这个例子中，C#编译器生成特殊代码创建一个委托来引用静态方法(**ShowItems**)，而静态方法的目标对象是对**"Jeff"**字符串的引用。稍后，当这个委托被调用(**invoke**)时，CLR 会调用(**call**)静态方法，并向其传递对**"Jeff"**字符串的引用。这是编译器耍的小“花招”，但效果不错，而且只要你不细想内部发生的事情，整个过程还是感觉非常自然的。

8.6.3 ExtensionAttribute 类

扩展方法的概念要不是 C#特有的就好了！具体地说，我们希望程序员能用一种编程语言定义一组扩展方法，并让其他语言的程序员利用它们。要实现这一点，选择的编译器必须能够搜索静态类型和方法来寻找匹配的扩展方法。另外，速度慢了还不行。编译器必须快速完成上述搜索，将编译时间控制在合理范围内。

在 C#中，一旦用 **this** 关键字标记了某个静态方法的第一个参数，编译器就会在内部向该方法应用一个定制特性。该特性会在最终生成的文件的元数据中持久性地存储下来。该特性在 **System.Core.dll** 程序集中定义，它看起来像下面这样：

```
// 在 System.Runtime.CompilerServices 命名空间中定义
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class |
    AttributeTargets.Assembly)]
public sealed class ExtensionAttribute : Attribute {
}
```

除此之外，任何静态类只要包含至少一个扩展方法，它的元数据中也会应用这个特性。类似地，任何程序集只要包含了至少一个符合上述特点的静态类，它的元数据中也会应用这个特性。这样一来，如果代码调用了不存在实例的方法，编译器就能快速扫描引用的所有程序集，判断它们哪些包含了扩展方法。然后，在这些程序集中，可以只扫描包含了扩展方法的静态类。在每个这样的静态类中，可以只扫描扩展方法来查找匹配。利用这个技术，代码能以最快速度编译完毕。

8.7 分部方法

假定用某个工具生成了包含类型定义的 C#源代码文件，工具知道你想在代码的某些位置定制类型的行为。正常情况下，是让工具生成的代码调用虚方法来进行定制。工具生成的代码还必须包含虚方法的定义。另外，这些方法的实现是什么事情都不做，直接返回了事。现在，如果想定制类的行为，就必须从基类派生并定义自己的类，重写虚方法来实现自己想要的行为。下面是一个例子：

```
// 工具生成的代码，存储在某个源代码文件中：
internal class Base {
    private String m_name;

    // 在更改 m_name 字段前调用
    protected virtual void OnNameChanging(String value) {
    }

    public String Name {
        get { return m_name; }
        set {
            OnNameChanging(value.ToUpper()); // 告诉类要进行更改了
            m_name = value;                  // 更改字段
        }
    }
}

// 开发人员生成的代码，存储在另一个源代码文件中：
internal class Derived : Base {
    protected override void OnNameChanging(string value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

遗憾的是，上述代码存在两个问题。

- 类型必须是非密封的类。这个技术不能用于密封类，也不能用于值类型(值类型隐式密封)。此外，这个技术不能用于静态方法，因为静态方法不能重写。
- 效率问题。定义一个类型只是为了重写一个方法，这会浪费少量系统资源。另外，即使不想重写 **OnNameChanging** 的行为，基类代码仍需调用一个什么都不做、直接就返回的虚方法。另外，无论 **OnNameChanging** 是否访问传给它的实参，编译器都会生成对 **ToUpper** 进行调用的 IL 代码。

利用 C# 的分部方法功能，可以在解决上述问题的同时覆盖类的行为。以下代码使用分部方法实现和上述代码完全一样的语义：

```
// 工具生成的代码，存储在某个源代码文件中：
internal sealed partial class Base {
    private String m_name;

    // 这是分部方法的声明
    partial void OnNameChanging(String value);

    public String Name {
        get { return m_name; }
        set {
            OnNameChanging(value.ToUpper()); // 通知类要进行更改了
            m_name = value;                  // 更改字段
        }
    }
}

// 开发人员生成的代码，存储在另一个源代码文件中：
internal sealed partial class Base {

    // 这是分部方法的实现，会在 m_name 更改前调用
```

```
partial void OnNameChanging(String value) {
    if (String.IsNullOrEmpty(value))
        throw new ArgumentNullException("value");
}
}
```

这个新版本要注意以下几个问题。

- 类现在密封(虽然并非一定如此)。事实上,类可以是静态类,甚至可以是值类型。
- 工具和开发者所生成的代码真的是一个类型定义的两个部分。要更多地了解分部类型,请参见 6.5 节“分部类、结构和接口”。
- 工具生成的代码包含分部方法的声明。要用 **partial** 关键字标记,无主体。
- 开发者生成的代码实现这个声明。该方法也要用 **partial** 关键字标记,有主体。

编译上述代码后,可以获得和原始代码一样的效果。现在的好处在于,可以重新运行工具,在新的源代码文件中生成新的代码,但你自己的代码是存储在一个单独的文件中的,不会受到影响。另外,这个技术可用于密封类、静态类以及值类型。



注意 在 Visual Studio 编辑器中,如果输入 **partial** 并按空格键,“智能感知”窗口会列出当前类型定义的、还没有匹配实现的所有分部方法声明。可以方便地从窗口中选择一个分部方法。然后,Visual Studio 会自动生成方法原型。这个功能提高了编程效率。

但是,分部方法还提供了另一个巨大的提升。如果不想修改工具生成的类型的行为,那么根本不需要提供自己的源代码文件。如果只是对工具生成的代码进行编译,编译器会改变生成的 IL 代码和元数据,使工具生成的代码看起来变成下面这样:

```
// 如果没有分部方法的“实现”方法,
// 工具生成的代码在逻辑上就等价于下面的代码。
internal sealed class Base {
    private String m_name;

    public String Name {
        get { return m_name; }
        set {
            m_name = value; // 更改字段
        }
    }
}
```

也就是说,如果没有实现分部方法,编译器不会生成任何代表分部方法的元数据。此外,编译器不会生成任何调用分部方法的 IL 指令。而且,编译器不会生成对本该传给分部方法的实参进行求值的 IL 指令。在这个例子中,编译器不会生成调用 **ToUpper** 方法的代码。结果就是更少的元数据/IL,运行时的性能得到了提升。



注意 分部方法的工作方式类似于 `System.Diagnostics.ConditionalAttribute` 特性。然而，分部方法只能在单个类型中使用，而 `ConditionalAttribute` 能用于对另一个类型中定义的方法进行有选择的调用。

规则和原则

关于分部方法，有一些附加的规则和原则需要注意。

- 它们只能在分部类或结构中声明。
- 分部方法的返回类型始终是 **void**，任何参数都不能用 **out** 修饰符来标记。之所以有这两个限制，是因为方法在运行时可能不存在，所以不能将变量初始化为方法也许会返回的东西。类似地，不允许 **out** 参数是因为方法必须初始化它，而方法可能不存在。分部方法可以有 **ref** 参数，可以是泛型方法，可以是实例或静态方法，而且可标记为 **unsafe**。
- 当然，分部方法的声明和实现必须具有完全一致的签名。如果两者都应用了定制特性，编译器会合并两个方法的特性。应用于参数的任何特性也会合并。
- 如果没有对应的实现部分，便不能在代码中创建一个委托来引用这个分部方法。这同样是由于方法在运行时不存在。编译器报告以下消息：**error CS0762: 无法通过方法 "Base.OnNameChanging(string)" 创建委托，因为该方法是没有实现声明的分部方法。**
- 分部方法总是被视为 **private** 方法，但 C# 编译器禁止在分部方法声明之前添加 **private** 关键字。

第9章 参 数

本章内容:

- 可选参数和命名参数
- 隐式类型的局部变量
- 以传引用的方式向方法传递参数
- 向方法传递可变数量的参数
- 参数和返回类型的设计规范
- 常量性

本章重点在于向方法传递参数的各种方式, 包括如何可选地指定参数, 按名称指定参数, 按引用传递参数, 以及如何定义方法来接受可变数量的参数。

9.1 可选参数和命名参数

设计方法的参数时, 可为部分或全部参数分配默认值。然后, 调用这些方法的代码可以选择不提供部分实参, 使用其默认值。此外, 调用方法时可通过指定参数名称来传递实参。以下代码演示了可选参数和命名参数的用法:

```
using System;
public static class Program {
    private static Int32 s_n = 0;

    private static void M(Int32 x = 9, String s = "A",
        DateTime dt = default(DateTime), Guid guid = new Guid()) {

        Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);
    }

    public static void Main() {
        // 1. 等同于 M(9, "A", default(DateTime), new Guid());
        M();

        // 2. 等同于 M(8, "X", default(DateTime), new Guid());
        M(8, "X");

        // 3. 等同于 M(5, "A", DateTime.Now, Guid.NewGuid());
        M(5, guid: Guid.NewGuid(), dt: DateTime.Now);

        // 4. 等同于 M(0, "1", default(DateTime), new Guid());
```

```

        M(s_n++, s_n++.ToString());

        // 5. 等同于以下两行代码:
        // String t1 = "2"; Int32 t2 = 3;
        // M(t2, t1, default(DateTime), new Guid());
        M(s: (s_n++).ToString(), x: s_n++);
    }
}

```

运行程序得到以下输出:

```

x=9, s=A, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=8, s=X, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=5, s=A, dt=8/16/2012 10:14:25 PM, guid=d24a59da-6009-4aae-9295-839155811309
x=0, s=1, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=3, s=2, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000

```

如你所见, 如果调用时省略了一个实参, C#编译器会自动嵌入参数的默认值。对 **M** 的第 3 个和第 5 个调用使用了 C# 的命名参数功能。在这两个调用中, 我为 **x** 显式传递了值, 并指出要名为 **guid** 和 **dt** 的参数传递实参。

向方法传递实参时, 编译器按从左到右的顺序对实参进行求值。在对 **M** 的第 4 个调用中, **s_n** 的当前值(0)传给 **x**, 然后 **s_n** 递增。随后, **s_n** 的当前值(1)作为字符串传给 **s**, 然后继续递增到 2。使用命名参数传递实参时, 编译器仍然按从左到右的顺序对实参进行求值。在对 **M** 的第 5 个调用中, **s_n** 中的当前值(2)被转换成字符串, 并保存到编译器创建的临时变量(**t1**)中。接着, **s_n** 递增到 3, 这个值保存到编译器创建的另一个临时变量(**t2**)中。然后, **s_n** 继续递增到 4。最后在调用 **M** 时, 向它传递的实参依次是 **t2**, **t1**, 一个默认 **DateTime** 和一个新建的 **Guid**。

9.1.1 规则和原则

如果在方法中为部分参数指定了默认值, 请注意以下附加的规则和原则。

- 可为方法、构造器方法和有参属性(C#索引器)的参数指定默认值。还可为属于委托定义一部分的参数指定默认值。以后调用该委托类型的变量时可省略实参来接受默认值。
- 有默认值的参数必须放在没有默认值的所有参数之后。换言之, 一旦定义了有默认值的参数, 它右边的所有参数也必须有默认值。例如在前面的 **M** 方法定义中, 如果删除 **s** 的默认值("A"), 就会出现编译错误。但这个规则有一个例外: “参数数组”^①这种参数必须放在所有参数(包括有默认值的这些)之后, 而且数组本身不能有一个默认值。
- 默认值必须是编译时能确定的常量值。那么, 哪些参数能设置默认值? 这些参数的类型可以是 C# 认定的基元类型(参见第 5 章的表 5-1)。还包括枚举类型, 以及能设为 **null** 的任何引用类型。值类型的参数可将默认值设为值类型的实例, 并让它的所有字段都包含零值。可以用 **default** 关键字或者 **new** 关键字来表达这个意思; 两种语法将生成

① 在本章后面 9.4 节“向方法传递可变数量的参数”详细讨论。

完全一致的 IL 代码。在 **M** 方法中设置 **dt** 参数和 **guid** 参数的默认值时，分别使用的就是这两种语法。

- 不要重命名参数变量，否则任何调用者以传参数名的方式传递实参，它们的代码也必须修改。例如，在前面的 **M** 方法声明中，如果将 **dt** 变量重命名为 **dateTime**，对 **M** 的第三个调用就会造成编译器显示以下消息：**error CS1739: "M"的最佳重载没有名为 "dt"的参数。**
- 如果方法从模块外部调用，更改参数的默认值具有潜在的危险性。**call site**^①在它的调用中嵌入默认值。如果以后更改了参数的默认值，但没有重新编译包含 **call site** 的代码，它在调用你的方法时就会传递旧的默认值。可考虑将默认值 **0/null** 作为哨兵值使用，从而指出默认行为。这样一来，即使更改了默认值，也不必重新编译包含了 **call site** 的全部代码。下面是一个例子：

```
// 不要这样做：
private static String MakePath(String filename = "Untitled") {
    return String.Format(@"C:\{0}.txt", filename);
}

// 而要这样做：
private static String MakePath(String filename = null) {
    // 这里使用了空接合操作符(??)；详情参见第19章
    return String.Format(@"C:\{0}.txt", filename ?? "Untitled");
}
```

- 如果参数用 **ref** 或 **out** 关键字进行了标识，就不能设置默认值。因为没有办法为这些参数传递有意义的默认值。

使用可选或命名参数调用方法时，还要注意以下附加的规则和原则。

- 实参可按任意顺序传递，但命名实参只能出现在实参列表的尾部。
- 可按名称将实参传给没有默认值的参数，但所有必须的实参都必须传递(无论按位置还是按名称)，编译器才能编译代码。
- C#不允许省略逗号之间的实参，比如 **M(1, ,DateTime.Now)**。因为这会造成对可读性的影响，程序员将被迫去数逗号。对于有默认值的参数，如果想省略它们的实参，以传参数名的方式传递实参即可。
- 如果参数要求 **ref/out**，为了以传参数名的方式传递实参，请使用下面这样的语法：

```
// 方法声明：
private static void M(ref Int32 x) { ... }

// 方法调用：
Int32 a = 5;
M(x: ref a);
```

① **call site** 是发出调用的地方，可理解成调用了目标方法的表达式或代码行。——译注



注意 写 C# 代码和 Microsoft Office 的 COM 对象模型进行互操作时, C# 的可选参数和命名参数功能非常好用。另外, 调用 COM 组件时, 如果是传引用的方式传递实参, C# 还允许省略 **ref/out**, 进一步简化编码。但如果调用的不是 COM 组件, C# 就要求必须向实参应用 **ref/out** 关键字。

9.1.2 DefaultParameterValueAttribute 和 OptionalAttribute

默认和可选参数的概念要不是 C# 特有的就好了! 具体地说, 我们希望程序员能用一种编程语言定义一个方法, 指出哪些参数是可选的, 以及它们的默认值是什么。然后, 另一种语言的程序员可以调用该方法。要实现这一点, 所选的编译器必须允许调用者忽略一些实参, 还必须能确定这些实参的默认值。

在 C# 中, 一旦为参数分配了默认值, 编译器就会在内部向该参数应用定制特性 **System.Runtime.InteropServices.OptionalAttribute**。该特性会在最终生成的文件的元数据中持久性地存储下来。此外, 编译器向参数应用 **System.Runtime.InteropServices.DefaultParameterValueAttribute** 特性, 并将该属性持久性存储到生成的文件的元数据中。然后, 会向 **DefaultParameterValueAttribute** 的构造器传递你在源代码中指定的常量值。

之后, 一旦编译器发现某个方法调用缺失了部分实参, 就可以确定省略的是可选的实参, 并从元数据中提取默认值, 将值自动嵌入调用中。

9.2 隐式类型的局部变量

C# 能根据初始化表达式的类型推断方法中的局部变量的类型, 如下所示:

```
private static void ImplicitlyTypedLocalVariables() {
    var name = "Jeff";
    ShowVariableType(name);           // 显示: System.String

    // var n = null;                 // 错误, 不能将 null 赋给隐式类型的局部变量
    var x = (String)null;              // 可以这样写, 但意义不大
    ShowVariableType(x);              // 显示: System.String

    var numbers = new Int32[] { 1, 2, 3, 4 };
    ShowVariableType(numbers);        // 显示: System.Int32[]

    // 复杂类型能少打一些字
    var collection = new Dictionary<String, Single>() { { "Grant", 4.0f } };

    // 显示: System.Collections.Generic.Dictionary`2[System.String, System.Single]
    ShowVariableType(collection);

    foreach (var item in collection) {
        // 显示: System.Collections.Generic.KeyValuePair`2[System.String, System.Single]
        ShowVariableType(item);
    }
}
```

```

}

private static void ShowVariableType<T>(T t) {
    Console.WriteLine(typeof(T));
}

```

ImplicitlyTypedLocalVariables 方法中的第一行代码使用 C# 的 **var** 关键字引入了一个新的局部变量。为了确定 **name** 变量的类型，编译器要检查赋值操作符(=)右侧的表达式类型。由于“Jeff”是字符串，所以编译器推断 **name** 的类型是 **String**。为了证明编译器正确推断出类型，我写了 **ShowVariableType** 方法。这个泛型方法推断它的实参的类型并在控制台上显示。为方便阅读，我在 **ImplicitlyTypedLocalVariables** 方法内部以注释形式列出了每次调用 **ShowVariableType** 方法的显示结果。

ImplicitlyTypedLocalVariables 方法内部的第二个赋值(被注释掉了)会产生编译错误: **error CS0815: 无法将“<null>”赋予隐式类型的局部变量**。这是由于 **null** 能隐式转型为任何引用类型或可空值类型。因此，编译器不能推断它的确切类型。但在第三个赋值语句中，我证明只要显式指定了类型(本例是 **String**)，就可以将隐式类型的局部变量初始化为 **null**。这样做虽然可行，但意义不大，因为可以写 **String x = null;** 来获得同样效果。

第 4 个赋值语句反映了 C# “隐式类型局部变量”功能的真正价值。没有这个功能，就不得不在赋值操作符的左右两侧指定 **Dictionary<String, Single>**。这不仅需要打更多的字，而且以后修改了集合类型或者任何泛型参数类型，赋值操作符两侧的代码也要修改。

在 **foreach** 循环中，我用 **var** 让编译器自动推断集合中的元素的类型。这证明了 **var** 能很好地用于 **foreach**，**using** 和 **for** 语句。还可在试验代码时利用它。例如，可以用方法的返回值初始化隐式类型的局部变量。开发方法时可以灵活更改返回类型。编译器能察觉到返回类型的变化，并自动更改变量的类型！当然，如果使用变量的代码没有相应地进行修改，还是像使用旧类型那样使用它，就可能无法编译。

在 Microsoft Visual Studio 中，鼠标放到 **var** 上将显示一条“工具提示”，指出编译器根据表达式推断出来的类型。在方法中使用匿名类型时必须用到 C# 的隐式类型局部变量，详情参见第 10 章“属性”。

不能用 **var** 声明方法的参数类型。原因显而易见，因为编译器必须根据在 **call site** 传递的实参来推断参数类型，但 **call site** 可能一个都没有，也可能有好多个^①。除此之外，不能用 **var** 声明类型中的字段。C# 的这个限制是出于多方面的考虑。一个原因是字段可以被多个方法访问，而 C# 团队认为这个协定(变量的类型)应该显式陈述。另一个原因是一旦允许，匿名类型(第 10 章)就会泄露到方法的外部。

① 要么一个类型都推断不出来，要么多个推断发生冲突。——译注



重要提示 不要混淆 **dynamic** 和 **var**。用 **var** 声明局部变量只是一种简化语法，它要求编译器根据表达式推断具体数据类型。**var** 关键字只能声明方法内部的局部变量，而 **dynamic** 关键字适用于局部变量、字段和参数。表达式不能转型为 **var**，但能转型为 **dynamic**。必须显式初始化用 **var** 声明的变量，但无需初始化用 **dynamic** 声明的变量。欲知 C# 的 **dynamic** 类型的详情，请参见 5.5 节“**dynamic** 基元类型”。

9.3 以传引用的方式向方法传递参数

CLR 默认所有方法参数都传值。传递引用类型的对象时，对象引用(或者说指向对象的指针)被传给方法。注意引用(或指针)本身是传值的，意味着方法能修改对象，而调用者能看到这些修改。对于值类型的实例，传给方法的是实例的一个副本，意味着方法将获得它专用的一个值类型实例副本，调用者中的实例不受影响。



重要提示 在方法中，必须知道传递的每个参数是引用类型还是值类型，处理参数的代码显著有别。

CLR 允许以传引用而非传值的方式传递参数。C# 用关键字 **out** 或 **ref** 支持这个功能。两个关键字都告诉 C# 编译器生成元数据来指明该参数是传引用的。编译器将生成代码来传递参数的地址，而非传递参数本身。

CLR 不区分 **out** 和 **ref**，意味着无论用哪个关键字，都会生成相同的 IL 代码。另外，元数据也几乎完全一致，只有一个 bit 除外，它用于记录声明方法时指定的是 **out** 还是 **ref**。但 C# 编译器是将这两个关键字区别对待的，而且这个区别决定了由哪个方法负责初始化所引用的对象。如果方法的参数用 **out** 来标记，表明不指望调用者在调用方法之前初始化好了对象。被调用的方法不能读取参数的值，而且在返回前必须向这个值写入。相反，如果方法的参数用 **ref** 来标记，调用者就必须在调用该方法前初始化参数的值，被调用的方法可以读取值以及/或者向值写入。

对于 **out** 和 **ref**，引用类型和值类型的行为迥然有异。先看一看为值类型使用 **out** 和 **ref**：

```
public sealed class Program {
    public static void Main() {
        Int32 x;                // x 没有初始化
        GetVal(out x);           // x 不必初始化
        Console.WriteLine(x);    // 显示 "10"
    }

    private static void GetVal(out Int32 v) {
        v = 10; // 该方法必须初始化 v
    }
}
```

在上述代码中，**x** 在 **Main** 的栈帧^①中声明。然后，**x** 的地址传递给 **GetVal**。**GetVal** 的 **v** 是一个指针，指向 **Main** 栈帧中的 **Int32** 值。在 **GetVal** 内部，**v** 指向的那个 **Int32** 被更改为 **10**。**GetVal** 返回时，**Main** 的 **x** 就有了一个为 **10** 的值，控制台上会显示 **10**。为大的值类型使用 **out**，可提升代码的执行效率，因为它避免了在进行方法调用时复制值类型实例的字段。

下例用 **ref** 代替了 **out**：

```
public sealed class Program {
    public static void Main() {
        Int32 x = 5;           // x 已经初始化
        AddVal(ref x);         // x 必须初始化
        Console.WriteLine(x);  // 显示"15"
    }

    private static void AddVal(ref Int32 v) {
        v += 10; // 该方法可使用 v 的已初始化的值
    }
}
```

在上述代码中，**x** 也在 **Main** 的栈帧中声明，并初始化为 **5**。然后，**x** 的地址传给 **AddVal**。**AddVal** 的 **v** 是一个指针，指向 **Main** 栈帧中的 **Int32** 值。在 **AddVal** 内部，**v** 指向的那个 **Int32** 要求必须是已经初始化的。因此，**AddVal** 可在任何表达式中使用该初始值。**AddVal** 还可更改这个值，新值会“返回”调用者。在本例中，**AddVal** 将 **10** 加到初始值上。**AddVal** 返回时，**Main** 的 **x** 将包含 **15**，这个值会在控制台上显示出来。

综上所述，从 IL 和 CLR 的角度看，**out** 和 **ref** 是同一码事：都导致传递指向实例的一个指针。但从编译器的角度看，两者是有区别的。根据是 **out** 还是 **ref**，编译器会按照不同的标准来验证你写的代码是否正确。以下代码试图向要求 **ref** 参数的方法传递未初始化的值，结果是编译器报告以下错误：**error CS0165: 使用了未赋值的局部变量"x"**。

```
public sealed class Program {
    public static void Main() {
        Int32 x; // x 没有初始化

        // 下一行代码无法通过编译，编译器将报告：
        // error CS0165: 使用了未赋值的局部变量"x"
        AddVal(ref x);

        Console.WriteLine(x);
    }

    private static void AddVal(ref Int32 v) {
        v += 10; // 该方法可使用 v 的已初始化的值
    }
}
```

① 栈帧(stack frame)代表当前线程的调用栈中的一个方法调用。在执行线程的过程中进行的每个方法调用都会在调用栈中创建并压入一个 **StackFrame**。——译注



重要提示 经常有人问我,为什么 C# 要求必须在调用方法时指定 **out** 或 **ref**? 毕竟,编译器知道被调用的方法需要的是 **out** 还是 **ref**,所以应该能正确编译代码。事实上,C#编译器确实能自动采取正确的操作。但 C# 语言的设计者认为调用者应显式表明意图。只有这样,在 **call site**(调用位置)那里,才能清楚地知道被调用的方法是否需要传递的变量值进行更改。

另外,CLR 允许根据使用的是 **out** 还是 **ref** 参数对方法进行重载。例如,在 C# 中,以下代码是合法的,可以通过编译:

```
public sealed class Point {
    static void Add(Point p) { ... }
    static void Add(ref Point p) { ... }
}
```

两个重载方法只有 **out** 和 **ref** 的区别则不合法,因为两个签名的元数据形式完全相同。所以,不能在上述 **Point** 类型中再定义以下方法:

```
static void Add(out Point p) { ... }
```

试图在 **Point** 类型中添加这个 **Add** 方法,C#编译器会显示以下消息:

CS0663: "Add" 不能定义仅在 ref 和 out 上有差别的重载方法。

为值类型使用 **out** 和 **ref**,效果等同于以传值的方式传递引用类型。对于值类型,**out** 和 **ref** 允许方法操纵单一的值类型实例。调用者必须为实例分配内存,被调用者则操纵该内存(中的内容)。对于引用类型,调用代码为一个指针分配内存(该指针指向一个引用类型的对象),被调用者则操纵这个指针。正因为如此,仅当方法“返回”对“方法知道的一个对象”的引用时,为引用类型使用 **out** 和 **ref** 才有意义。以下代码对此进行了演示。

```
using System;
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs; // fs 没初始化

        // 打开第一个待处理的文件
        StartProcessingFiles(out fs);

        // 如果有更多需要处理的文件,就继续
        for (; fs != null; ContinueProcessingFiles(ref fs)) {

            // 处理一个文件
            fs.Read(...);
        }

        private static void StartProcessingFiles(out FileStream fs) {
            fs = new FileStream(...); // fs 必须在这个方法中初始化
        }
    }
}
```

```

    }

    private static void ContinueProcessingFiles(ref FileStream fs) {
        fs.Close();    // 关闭最后一个操作的文件

        // 打开下一个文件; 如果没有更多的文件, 就“返回”null
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}

```

可以看出, 上述代码最大的不同就是定义了一些使用了 **out** 或 **ref** 引用类型参数的方法, 并用这些方法构造对象。新对象的指针将“返回”给调用者。还要注意, **ContinueProcessingFiles** 方法可对传给它的对象进行处理, 再“返回”一个新对象。之所以能这样做, 是因为参数标记了 **ref**。上述代码可简化为以下形式:

```

using System;
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs = null; // 初始化为 null (必要的操作)

        // 打开第一个待处理的文件
        ProcessFiles(ref fs);

        // 如果有更多需要处理的文件, 就继续
        for (; fs != null; ProcessFiles(ref fs)) {

            // 处理文件
            fs.Read(...);
        }

        private static void ProcessFiles(ref FileStream fs) {
            // 如果先前的文件是打开的, 就将其关闭
            if (fs != null) fs.Close(); // 关闭最后一个操作的文件

            // 打开下一个文件; 如果没有更多的文件, 就“返回”null
            if (noMoreFilesToProcess) fs = null;
            else fs = new FileStream (...);
        }
    }
}

```

下例演示了如何用 **ref** 关键字实现一个用于交换两个引用类型的方法:

```

public static void Swap(ref Object a, ref Object b) {
    Object t = b;
    b = a;
    a = t;
}

```

为了交换对两个 **String** 对象的引用, 你或许以为代码能像下面这样写:

```

public static void SomeMethod(){
    String s1="Jeffrey";
    String s2="Richter";

    Swap(ref s1,ref s2);
}

```

```

    Console.WriteLine(s1); // 显示"Richter"
    Console.WriteLine(s2); // 显示"Jeffrey"
}

```

但上述代码无法通过编译。问题在于，对于以传引用的方式传给方法的变量，它的类型必须与方法签名中声明的类型相同，换句话说，**Swap** 预期的是两个 **Object** 引用，而不是两个 **String** 引用。为了交换两个 **String** 引用，代码要像下面这样写：

```

public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";

    // 以传引用的方式传递的变量，
    // 必须和方法预期的匹配
    Object o1 = s1, o2 = s2;
    Swap(ref o1, ref o2);

    // 完后再将 Object 转型为 String
    s1 = (String) o1;
    s2 = (String) o2;

    Console.WriteLine(s1); // 显示"Richter"
    Console.WriteLine(s2); // 显示"Jeffrey"
}

```

这个版本的 **SomeMethod** 可以通过编译，并会如预期的一样执行。传递的参数之所以必须与方法预期的参数匹配，原因是保障类型安全。以下代码(幸好不会编译)演示了类型安全型是如何被破坏的：

```

internal sealed class SomeType {
    public Int32 m_val;
}

public sealed class Program {
    public static void Main() {
        SomeType st;

        // 以下代码将产生编译错误：
        // error CS1503: 参数"1":无法从"out SomeType"转换为"out object"
        GetAnObject(out st);

        Console.WriteLine(st.m_val);
    }

    private static void GetAnObject(out Object o) {
        o = new String('X', 100);
    }
}

```

在上述代码中，**Main** 显然预期 **GetAnObject** 方法返回一个 **SomeType** 对象。但是，由于 **GetAnObject** 的签名表示的是一个 **Object** 引用，所以 **GetAnObject** 可以将 **o** 初始化为任意类型的对象。在这个例子中，当 **GetAnObject** 返回 **Main** 时，**st** 引用一个 **String**，显然不是 **SomeType** 对象，所以对 **Console.WriteLine** 的调用肯定会失败。幸好，C#编译器不会编译上述代码，因为 **st** 是一个 **SomeType** 引用，而 **GetAnObject** 要求的是一个 **Object** 引用。

可用泛型来修正这些方法，使它们按你的预期来运行。下面修正了前面的 **Swap** 方法：

```
public static void Swap<T>(ref T a, ref T b) {
    T t = b;
    b = a;
    a = t;
}
```

重新了 **Swap** 后，以下代码(和以前展示过的完全相同)就能通过编译了，而且能完美运行：

```
public static void SomeMethod(){
    String s1="Jeffrey";
    String s2="Richter";

    Swap(ref s1,ref s2);
    Console.WriteLine(s1); // 显示"Richter"
    Console.WriteLine(s2); // 显示"Jeffrey"
}
```

要查看用泛型来解决这个问题的其他例子，请参见 **System.Threading** 命名空间中的 **Interlocked** 类的 **CompareExchange** 和 **Exchange** 方法。

9.4 向方法传递可变数量的参数

方法有时需要获取可变数量的参数。例如，**System.String** 类型的一些方法允许连接任意数量的字符串，还有一些方法允许指定一组要统一格式化的字符串。

为了接受可变数量的参数，方法要像下面这样声明：

```
static Int32 Add(params Int32[] values) {
    // 注意：如果愿意，可将 values 数组传给其他方法

    Int32 sum = 0;
    if (values != null) {
        for (Int32 x = 0; x < values.Length; x++)
            sum += values[x];
    }
    return sum;
}
```

除了 **params** 关键字，这个方法的一切对你来说都应该是非常熟悉的。**params** 只能应用于方法签名中的最后一个参数。暂时忽略 **params** 关键字，可以明显地看出 **Add** 方法接受一个 **Int32** 类型的数组，然后遍历数组，将其中的 **Int32** 值加到一起，结果 **sum** 返回给调用者。

显然，可以像下面这样调用该方法：

```
public static void Main() {
    // 显示 "15"
    Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 }));
}
```

数组能用任意数量的一组元素来初始化，再传给 **Add** 方法进行处理。尽管上述代码可以通过编译并能正确运行，但并不好看。我们当然希望能像下面这样调用 **Add** 方法：

```
public static void Main() {
    // 显示 "15"
```

```

    Console.WriteLine(Add(1, 2, 3, 4, 5));
}

```

由于 **params** 关键字的存在，所以的确能这样做。**params** 关键字告诉编译器向参数应用定制特性 **System.ParamArrayAttribute** 的一个实例。

C#编译器检测到方法调用时，会先检查所有具有指定名称、同时参数没有应用 **ParamArray** 特性的方法。找到匹配的方法，就生成调用它所需的代码。没有找到，就接着检查应用了 **ParamArray** 特性的方法。找到匹配的方法，编译器先生成代码来构造一个数组，填充它的元素，再生成代码来调用所选的方法。

上个例子并没有定义可获取 5 个 **Int32** 兼容实参的 **Add** 方法。但编译器发现在一个 **Add** 方法调用中传递了一组 **Int32** 值，而且有一个 **Add** 方法的 **Int32** 数组参数应用了 **ParamArray** 特性。因此，编译器认为这是一个匹配，会生成代码将实参保存到一个 **Int32** 类型的数组中，再调用 **Add** 方法并传递该数组。最终结果就是，你可以写代码直接向 **Add** 方法传递一组实参，而编译器会生成代码，像前面的第一个版本的方法调用那样，帮你构造和初始化一个数组来容纳这些实参。

只有方法的最后一个参数才可以用 **params** 关键字(**ParamArrayAttribute**)标记。另外，这个参数只能标识一维数组(任意类型)。可为这个参数传递 **null** 值，或传递对包含零个元素的一个数组的引用。以下 **Add** 调用能正常编译和运行，生成的结果是 **0**(和预期的一样)：

```

public static void Main() {
    // 以下两行都显示"0"
    Console.WriteLine(Add());           // 向 Add 传递 new Int32[0]
    Console.WriteLine(Add(null));       // 向 Add 传递 null: 更高效(因为不会分配数组)
}

```

前面所有例子都只是演示了如何写方法来获取任意数量的 **Int32** 参数。那么，如何写方法来获取任意数量、任意类型的参数呢？答案很简单：只需修改方法原型，让它获取一个 **Object[]** 而不是 **Int32[]**。以下方法显示传给它的每个对象的类型：

```

public sealed class Program {
    public static void Main() {
        DisplayTypes(new Object(), new Random(), "Jeff", 5);
    }

    private static void DisplayTypes(params Object[] objects) {
        if (objects != null) {
            foreach (Object o in objects)
                Console.WriteLine(o.GetType());
        }
    }
}

```

上述代码的输出如下：

```

System.Object
System.Random
System.String
System.Int32

```



重要提示 注意, 调用参数数量可变的方法对性能有所影响(除非显式传递 `null`)。毕竟, 数组对象必须在堆上分配, 数组元素必须初始化, 而且数组的内存最终需要垃圾回收。要减少对性能的影响, 可考虑定义几个没有使用 `params` 关键字的重载版本。关于这方面的范例, 请参考 `System.String` 类的 `Concat` 方法, 该方法定义了以下重载版本:

```
public sealed class String : Object, ... {
    public static string Concat(object arg0);
    public static string Concat(object arg0, object arg1);
    public static string Concat(object arg0, object arg1, object arg2);
    public static string Concat(params object[] args);
    public static string Concat(string str0, string str1);
    public static string Concat(string str0, string str1, string str2);
    public static string Concat(string str0, string str1, string str2,
                                string str3);
    public static string Concat(params string[] values);
}
```

如你所见, `Concat` 方法定义了几个没有使用 `params` 关键字的重载版本。这是为了改善常规情形下的性能。使用了 `params` 关键字的重载则用于不太常见的情形; 在这些情形下, 性能有一定的损失。但幸运的是, 这些情形本来就不常见。

9.5 参数和返回类型的设计规范

声明方法的参数类型时, 应尽量指定最弱的类型, 宁愿要接口也不要基类。例如, 如果要写方法来处理一组数据项, 最好是用接口(比如 `IEnumerable<T>`)声明参数, 而不要用强数据类型(比如 `List<T>`)或者更强的接口类型(比如 `ICollection<T>` 或 `IList<T>`):

```
// 好: 方法使用弱参数类型
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }
```

```
// 不好: 方法使用强参数类型
public void ManipulateItems<T>(List<T> collection) { ... }
```

原因是调用第一个方法时可传递数组对象、`List<T>` 对象、`String` 对象或者其他对象——只要对象的类型实现了 `IEnumerable<T>` 接口。相反, 第二个方法只允许传递 `List<T>` 对象, 不接受数组或 `String` 对象。显然, 第一个方法更好, 它更灵活, 适合更广泛的情形。

当然, 如果方法需要的是列表(而非任何可枚举的对象), 就应该将参数类型声明为 `IList<T>`。但仍然要避免将参数类型声明为 `List<T>`。声明为 `IList<T>`, 调用者可向方法传递数组和实现了 `IList<T>` 的其他类型的对象。

注意, 这里的例子讨论的是集合, 是用接口体系结构来设计的。讨论使用基类体系结构设计的类时, 概念同样适用。例如, 要实现对流中的字节进行处理的方法, 可定义以下方法:

```
// 好: 方法使用弱参数类型
```

```
public void ProcessBytes(Stream someStream) { ... }

// 不好: 方法使用强参数类型
public void ProcessBytes(FileStream fileStream) { ... }
```

第一个方法能处理任何流, 包括 **FileStream**, **NetworkStream** 和 **MemoryStream** 等。第二个只能处理 **FileStream** 流, 这限制了它的应用。

相反, 一般最好是将方法的返回类型声明为最强的类型(防止受限于特定类型)。例如, 方法最好返回 **FileStream** 而不是 **Stream** 对象:

```
// 好: 方法使用强返回类型
public FileStream OpenFile() { ... }

// 不好: 方法使用弱返回类型
public Stream OpenFile() { ... }
```

第一个方法是首选的, 它允许方法的调用者将返回对象视为 **FileStream** 对象或者 **Stream** 对象。但第二个方法要求调用者只能将返回对象视为 **Stream** 对象。总之, 要确保调用者在调用方法时有尽量大的灵活性, 使方法的适用范围更大。

有时需要不影响调用者的前提下修改方法的内部实现。在刚才的例子中, **OpenFile** 方法不太可能更改内部实现来返回除 **FileStream**(或 **FileStream** 的派生类型)之外的其他对象。但如果方法返回 **List<String>** 对象, 就可能想在未来的某个时候修改它的内部实现来返回一个 **String[]**。如果想保持一定的灵活性, 在将来更改方法返回的东西, 请选择一个较弱的返回类型。例如:

```
// 灵活: 方法使用较弱的返回类型
public IList<String> GetStringCollection() { ... }

// 不灵活: 方法使用较强的返回类型
public List<String> GetStringCollection() { ... }
```

在这个例子中, 即使 **GetStringCollection** 方法在内部使用一个 **List<String>** 对象并返回它, 但最好还是修改方法的原型, 使它返回一个 **IList<String>**。将来, **GetStringCollection** 方法可更改它的内部集合来使用一个 **String[]**。与此同时, 不需要修改调用者的源代码。事实上, 调用者甚至不需要重新编译。注意, 这个例子在较弱的类型中选择的是最强的那一个。例如, 它没有使用最弱的 **IEnumerable<String>**, 也没有使用较强的 **ICollection<String>**。^①

9.6 常量性

有的语言(比如非托管 C++)允许将方法或参数声明为常量, 从而禁止实例方法中的代码更改对象的任何字段, 或者更改传给方法的任何对象。CLR 没有提供这个功能, 许多程序员因此觉得很遗憾。既然 CLR 都不提供, 面向它的任何编程语言(包括 C#)自然也无法提供。

^① **IList** 继承自 **ICollection**, **ICollection** 继承自 **IEnumerable**。作者的意思是说, 在这三个比 **List<String>** 都弱的类型中, 选择的是最强的 **IList**。——译注

首先要注意，非托管 C++ 将实例方法或参数声明为 **const** 只能防止程序员用一般的代码来更改对象或参数。方法内部总是可以更改对象或实参的。这要么是通过强制类型转换来去掉“常量性”，要么通过获取对象/实参的地址，再向那个地址写入。从某种意义上说，非托管 C++ 向程序员撒了一个谎，使他们以为常量对象或实参不能写入(事实上可以)。

实现类型时，开发人员可以避免写操纵对象或实参的代码。例如，**String** 类就没有提供任何能更改 **String** 对象的方法，所以字符串是不可变(**immutable**)的。

此外，Microsoft 很难为 CLR 赋予验证常量对象/实参未被更改的能力。CLR 将不得不对每个写入操作进行验证，确定该写入针对的不是常量对象。这对性能影响很大。当然，如果检测到有违反常量性的地方，会造成 CLR 抛出异常。此外，如果支持常量性，还会给开发人员带来大量复杂性。例如，如果类型是不可变的，它的所有派生类型都不得不遵守这个约定。除此之外，在不可变的类型中，字段也必须不可变。

考虑到这些原因以及其他许多原因，CLR 没有提供对常量对象/实参的支持。

[illegible]

附件 4 项目可行性研究报告 附件 5 项目可行性研究报告 附件 6 项目可行性研究报告
 附件 7 项目可行性研究报告 附件 8 项目可行性研究报告 附件 9 项目可行性研究报告

[illegible]

— 41 —

第 10 章 属 性

本章内容：

- 无参属性
- 有参属性
- 调用属性访问器方法时的性能
- 属性访问器的可访问性
- 泛型属性访问器方法

本章讨论属性，它允许源代码用简化语法来调用方法。CLR 支持两种属性：**无参属性**，平时说的属性就是指它；**有参属性**，它在不同的编程语言中有不同的称呼。例如，C#将有参属性称为**索引器**，Microsoft Visual Basic 将有参属性称为**默认属性**。还要讨论如何使用“对象和集合初始化器”来初始化属性，以及如何用 C#的匿名类型和 **System.Tuple** 类型将多个属性打包到一起。

10.1 无参属性

许多类型都定义了能被获取或更改的状态信息。这种状态信息一般作为类型的字段成员实现。例如，以下类型定义包含两个字段：

```
public sealed class Employee {  
    public String Name;           // 员工姓名  
    public Int32 Age;             // 员工年龄  
}
```

创建该类型的实例后，可以使用以下形式的代码轻松获取(get)或设置(set)它的状态信息：

```
Employee e = new Employee();  
e.Name = "Jeffrey Richter";    // 设置员工姓名  
e.Age = 45;                     // 设置员工年龄  
  
Console.WriteLine(e.Name);     // 显示 "Jeffrey Richter"
```

这种查询和设置对象状态信息的做法十分常见。但我必须争辩的是，永远都不应该像这样实现。面向对象设计和编程的重要原则之一就是**数据封装**，意味着类型的字段永远不应该公开，否则很容易因为不恰当使用字段而破坏对象的状态。例如，以下代码可以很容易地破坏一个 **Employee** 对象：

```
e.Age=-5;    // 怎么可能有人是-5岁呢?
```

还有其他原因促使我们封装对类型中的数据字段的访问。其一，你可能希望访问字段来执行一些副作用^①、缓存某些值或者推迟创建一些内部对象^②。其二，你可能希望以线程安全的方式访问字段。其三，字段可能是一个逻辑字段，它的值不由内存中的字节表示，而是通过某个算法来计算获得。

基于上述原因，强烈建议将所有字段都设为 **private**。要允许用户或类型获取或设置状态信息，就公开一个针对该用途的方法。封装了字段访问的方法通常称为访问器(accessor)方法。访问器方法可选择对数据的合理性进行检查，确保对象的状态永远不被破坏。例如，我将上一个类重写为以下形式：

```
public sealed class Employee {
    private String m_Name; // 字段现在是私有的
    private Int32 m_Age;    // 字段现在是私有的

    public String GetName() {
        return(m_Name);
    }

    public void SetName(String value) {
        m_Name = value;
    }

    public Int32 GetAge() {
        return(m_Age);
    }

    public void SetAge(Int32 value) {
        if (value < 0)
            throw new ArgumentOutOfRangeException("value",
                value.ToString(),
                "The value must be greater than or equal to 0");
        m_Age = value;
    }
}
```

虽然这只是一个简单的例子，但还是可以看出数据字段封装带来的巨大好处。另外可以看出，实现只读属性或只写属性是多么简单！只需选择不实现一个访问器方法即可。另外，将 **SetXxx** 方法标记为 **protected**，就可以只允许派生类型修改值。

但是，像这样进行数据封装有两个缺点。首先，因为不得不实现额外的方法，所以必须写更多的代码；其次，类型的用户必须调用方法，而不能直接引用字段名。

```
e.SetName("Jeffrey Richter");    // 更新员工姓名
String EmployeeName = e.GetName(); // 获取员工姓名
e.SetAge(41);                     // 更新员工年龄
e.SetAge(-5);                     // 抛出 ArgumentOutOfRangeException 异常
Int32 EmployeeAge = e.GetAge();   // 获取员工年龄
```

① 即 **side effect**：在计算机编程中，如果一个函数或表达式除了生成一个值，还会造成状态的改变，就说它会造成副作用；或者说会执行一个副作用。——译注

② 推迟创建对象是指在对象第一次需要时才真正创建它。——译注

我个人认为这些缺点微不足道。不过，编程语言和 CLR 还是提供了一个称为属性(property)的机制。它缓解了第一个缺点所造成的影响，同时完全消除了第二个缺点。

下面的类使用了属性，它与前面定义的类功能相同：

```
public sealed class Employee {
    private String m_Name;
    private Int32 m_Age;

    public String Name {
        get { return (m_Name); }
        set { m_Name = value; } // 关键字 value 总是代表新值
    }

    public Int32 Age {
        get { return (m_Age); }
        set {
            if (value < 0) // 关键字 value 总是代表新值
                throw new ArgumentOutOfRangeException("value",
                    value.ToString(),
                    "The value must be greater than or equal to 0");
            m_Age = value;
        }
    }
}
```

如你所见，属性使类型的定义稍微复杂了一些，但由于属性允许采用以下形式来写代码，所以额外的付出还是值得的：

```
e.Name = "Jeffrey Richter";           // set 员工姓名
String EmployeeName = e.Name;         // get 员工姓名
e.Age = 41;                           // set 员工年龄
e.Age = -5;                           // 抛出 ArgumentOutOfRangeException 异常
Int32 EmployeeAge = e.Age;            // get 员工年龄
```

可将属性想象成智能字段，即背后有额外逻辑的字段。CLR 支持静态、实例、抽象和虚属性。另外，属性可用任意“可访问性”修饰符来标记(详情参见第 6.3 节“成员的可访问性”)，而且可以在接口中定义(详情参见第 13 章“接口”)。

每个属性都有名称和类型(类型不能是 **void**)。属性不能重载，即不能定义名称相同、类型不同的两个属性。定义属性时通常同时指定 **get** 和 **set** 两个方法。但可省略 **set** 方法来定义只读属性，或省略 **get** 方法来定义只写属性。

经常利用属性的 **get** 和 **set** 方法操纵类型中定义的私有字段。私有字段通常称为支持字段(backing field)。但 **get** 和 **set** 方法并非一定要访问支持字段。例如，**System.Threading.Thread** 类型提供了 **Priority** 属性来直接和操作系统通信。**Thread** 对象内部没有一个关于线程优先级的字段。没有支持字段的另一种典型的属性是在运行时计算的只读属性。例如，以 0 结束的一个数组的长度或者已知高度和宽度的一个矩形的面积。

定义属性时，取决于属性的定义，编译器在最后的托管程序集中生成以下两项或三项。

- 代表属性 **get** 访问器的方法。仅在属性定义了 **get** 访问器方法时生成。
- 代表属性 **set** 访问器的方法。仅在属性定义了 **set** 访问器方法时生成。

- 托管程序集元数据中的属性定义。这一项必然生成。

以前面的 **Employee** 类型为例。编译器编译该类型时发现其中的 **Name** 和 **Age** 属性。由于两者都有 **get** 和 **set** 访问器方法，所以编译器在 **Employee** 类型中生成 4 个方法定义，这造成原始代码似乎是像下面这样写的：

```
public sealed class Employee {
    private String    m_Name;
    private Int32     m_Age;

    public String get_Name() {
        return m_Name;
    }
    public void set_Name(String value) {
        m_Name = value; // 实参 value 总是代表新设的值
    }

    public Int32 get_Age() {
        return m_Age;
    }

    public void set_Age(Int32 value) {
        if (value < 0) // value 总是代表新值
            throw new ArgumentOutOfRangeException("value",
                value.ToString(),
                "The value must be greater than or equal to 0");
        m_Age = value;
    }
}
```

编译器在你指定的属性名之前自动附加 **get_**或**set_**前缀来生成方法名。C#内建了对属性的支持。C#编译器发现代码试图获取或设置属性时，实际会生成对上述某个方法的调用。即使编程语言不直接支持属性，也可调用需要的访问器方法来访问属性。效果一样，只是代码看起来没那么优雅。

除了生成访问器方法，针对源代码中定义的每一个属性，编译器还会在托管程序集的元数据中生成一个属性定义项。在这个记录项中，包含了一些标志(flags)以及属性的类型。另外，它还引用了 **get** 和 **set** 访问器方法。这些信息唯一的作用就是在“属性”这种抽象概念与它的访问器方法之间建立起一个联系。编译器和其他工具可利用这种元数据信息(使用 **System.Reflection.PropertyInfo** 类来获得)。CLR 不使用这种元数据信息，在运行时只需要访问器方法。

10.1.1 自动实现的属性

如果只是为了封装一个支持字段而创建属性，C#还提供了一种更简洁的语法，称为**自动实现的属性**(Automatically Implemented Property，后文简称为 AIP)，例如下面的 **Name** 属性：

```
public sealed class Employee {
    // 自动实现的属性
    public String Name { get; set; }

    private Int32 m_Age;

    public Int32 Age {
```

```

get { return(m_Age); }
set {
    if (value < 0) // value 关键字总是代表新值
        throw new ArgumentOutOfRangeException("value",
            value.ToString(),
            "The value must be greater than or equal to 0");
    m_Age = value;
}
}
}

```

声明属性而不提供 **get/set** 方法的实现，C# 会自动为你声明一个私有字段。在本例中，字段的类型是 **String**，也就是属性的类型。另外，编译器会自动实现 **get_Name** 和 **set_Name** 方法，分别返回和设置字段中的值。

和直接声明名为 **Name** 的 **public String** 字段相比，AIP 的优势在哪里？事实上，两者存在一处重要的区别。使用 AIP，意味着你已经创建了一个属性。访问该属性的任何代码实际都会调用 **get** 和 **set** 方法。如果以后决定自己实现 **get** 和/或 **set** 方法，而不是接受编译器的默认实现，访问属性的任何代码都不必重新编译。然而，如果将 **Name** 声明为字段，以后又想把它更改为属性，那么访问字段的所有代码都必须重新编译才能访问属性方法。

- 我个人不喜欢编译器的 AIP 功能，所以一般会避免使用它。理由是：字段声明语法可能包含初始化部分，所以要在一行代码中声明并初始化字段。但没有简单的语法初始化 AIP。所以，必须在每个构造器方法中显式初始化每个 AIP。
- 运行时序列化引擎将字段名持久存储到序列化的流中。AIP 的支持字段名称由编译器决定，每次重新编译代码都可能更改这个名称。因此，任何类型只要含有一个 AIP，就没办法对该类型的实例进行反序列化。在任何想要序列化或反序列化的类型中，都不要使用 AIP 功能。
- 调试时不能在 AIP 的 **get** 或 **set** 方法上添加断点，所以不好检测应用程序在什么时候获取或设置这个属性。相反，手动实现的属性可设置断点，查错更方便。

还要注意，如果使用 AIP，属性必然是可读和可写的。换言之，编译器肯定同时生成 **get** 和 **set** 方法。这个设计的道理在于，只写字段不能读取值有什么用呢？类似地，只读字段肯定是有默认值的。另外，由于不知道编译器生成的支持字段到底是什么名字，所以代码只能用属性名访问属性。另外，任何访问器方法(**get** 或 **set**)要显式实现，两个访问器方法都必须显式实现，就不能使用 AIP 功能了。换言之，AIP 是作用于整个属性的；要么都用，要么都不用。不能显式实现一个访问器方法，而让另一个自动实现。

10.1.2 合理定义属性

我个人不喜欢属性，我还希望 Microsoft .NET Framework 及其编程语言不要提供对属性的支持。理由是属性看起来和字段相似，但本质是方法。这造成了大量误解。程序员在看到貌似访问字段的代码时，会做出一些对属性来说不成立的假定，具体如下所示。

- 属性可以只读或只写，而字段访问总是可读和可写的(一个例外是 **readonly** 字段仅在构

造器中可写)。如果定义属性，最好同时为它提供 **get** 和 **set** 访问器方法。

- 属性方法可能抛出异常；字段访问永远不会。
- 属性不能作为 **out** 或 **ref** 参数传给方法，而字段可以。例如以下代码是编译不了的：

```
using System;

public sealed class SomeType {
    private static String Name {
        get { return null; }
        set {}
    }

    static void MethodWithoutParam(out String n) { n = null; }

    public static void Main() {
        // 对于下一行代码，C#编译器将报告以下错误消息：
        // error CS0206：属性或索引器不得作为 out 或 ref 参数传递
        MethodWithoutParam(out Name);
    }
}
```

- 属性方法可能花较长时间执行，字段访问则总是立即完成。许多人使用属性是为了线程同步，这就可能造成线程永远终止。所以，要线程同步就不要使用属性，而要使用方法。此外，如果你的类可以被远程访问(例如从 **System.MarshalByRefObject** 派生)，调用属性方法会非常慢。在这种情况下应该优先使用方法而不是属性。我个人认为从 **MarshalByRefObject** 派生的类永远都不应该使用属性。
- 连续多次调用，属性方法每次都可能返回不同的值，字段则每次都返回相同的值。**System.DateTime** 类的只读属性 **Now** 返回当前日期和时间。每次查询这个属性都返回不同的值。这是一个错误，Microsoft 现在很想修正这个错误，将 **Now** 改成方法而不是属性。**Environment** 的 **TickCount** 属性是 Microsoft 犯错的另一个例子。
- 属性方法可能造成明显的副作用^①，字段访问则永远不会。类型的使用者应该能按照他/她选择的任何顺序设置类型定义各个属性，而不会造成类型中(因为设置顺序的不同)出现不同的行为。
- 属性方法可能需要额外的内存，或者返回的引用并非指向对象状态一部分，造成对返回对象的修改作用不到原始对象身上。而查询字段返回的引用总是指向原始对象状态的一部分。使用会返回一个拷贝的属性很容易引起混淆，文档也经常没有专门说明。^②

① 这里的副作用(side effect)是指，访问属性时，除了单纯设置或获取属性，还会造成对象状态的改变。如果存在多个副作用，程序的行为就要依赖于历史；或者说要依赖于求值顺序。如果以不同的顺序设置属性，类型每一次的行为都不同，那么显然是不合理的。——译注

② 指开发人员在文档中没有清楚地指明这是属性，而且返回的是一个 **copy**，造成别人在使用这个类时产生混淆。——译注

属性和 Visual Studio 调试器

Microsoft Visual Studio 允许在调试器的监视窗口中输入一个对象的属性。这样一来，每次遇到一个断点，调试器都会调用属性的 **get** 访问器方法，并显示返回值。这个技术在查错时很有用，但也有可能造成 bug，并损害调试性能。例如，假定为网络共享中的文件创建了一个 **FileStream**，然后将 **FileStream** 的 **Length** 属性添加到调试器的监视窗口中。现在，每次遇到一个断点，调试器都会调用 **Length** 的 **get** 访问器方法，该方法内部向服务器发出一个网络请求来获取文件的当前长度！

类似地，如果属性的 **get** 访问器方法有一个 side effect，那么每次抵达断点，都会执行这个 side effect。例如，假定属性的 **get** 访问器方法每次调用时都递增一个计数器，这个计数器每次抵达断点时也会递增。考虑到可能有这些问题，Visual Studio 允许为监视窗口中显示的属性关闭属性求值。要在 Visual Studio 中关闭属性求值，请选择“工具”|“选项”|“调试”|“常规”。然后，在如图 10-1 所示的列表框中，清除勾选“启用属性求值和其他隐式函数调用”。注意，即使清除了这个选项，仍可将属性添加到监视窗口，然后手动强制 Visual Studio 对它进行求值。为此，单击监视窗口“值”列中的强制求值圆圈即可。

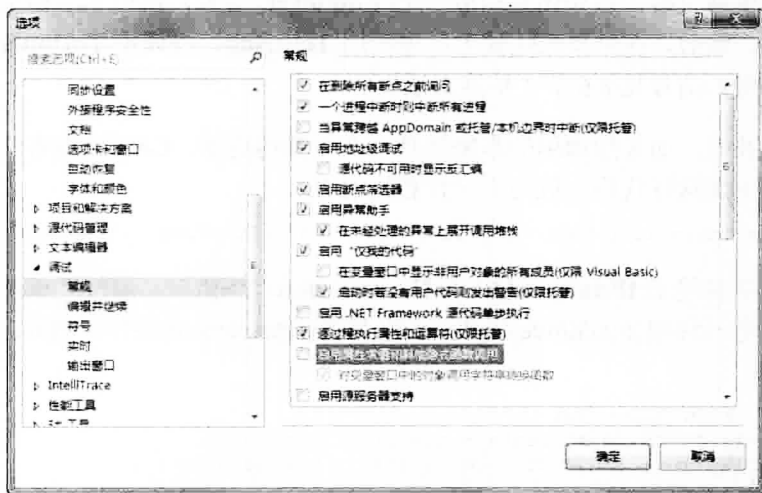


图 10-1 Visual Studio 的常规调试设置

我发现现在的人对属性的依赖有过之而无不及，经常是不管有没有必要都使用属性。仔细研究一下上面这个属性和字段差别列表，你会发现只有在极个别的情况下，定义属性才真正有用，同时不会造成开发人员的混淆。属性唯一的好处就是提供了简化的语法。和调用普通方法(非属性中的方法)相比，属性不仅不会提升代码的性能，还会妨碍对代码的理解。要是我参与 .NET Framework 以及编译器的设计，根本就不会提供属性；相反，我会让程序员老老实实在地实现 **GetXxx** 和 **SetXxx** 方法。然后，编译器可以提供一些特殊的、简化的语

法来调用这些方法。但是，我希望编译器使用有别于字段访问的语法，使程序员能真正理解他们正在做什么——是在调用一个方法！

10.1.3 对象和集合初始化器

经常要构造一个对象并设置对象的一些公共属性(或字段)。为了简化这个常见的编程模式，C#语言支持一种特殊的对象初始化语法。下面是一个例子：

```
Employee e = new Employee() { Name = "Jeff", Age = 45 };
```

这个语句做了好几件事情，包括构造一个 **Employee** 对象，调用它的无参构造器，将它的公共 **Name** 属性设为"Jeff"，并将公共 **Age** 属性设为 45。事实上，这一行代码等价于以下几行代码。可以检查两者的 IL 代码来加以验证。

```
Employee e = new Employee();
e.Name = "Jeff";
e.Age = 45;
```

对象初始化器语法真正的好处在于，它允许在表达式的上下文(相对于语句的上下文)中编码，允许组合多个函数，进而增强了代码的可读性。例如，现在可以写这样的代码：

```
String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

这个语句做的事情更多，首先还是构造一个 **Employee** 对象，调用它的构造器，再初始化两个公共属性。然后，在结果表达式上，先调用 **ToString**，再调用 **ToUpper**。要深入了解函数的组合使用，请参见 8.6 节“扩展方法”。

作为一个小的补充，如果想调用的本来就是一个无参构造器，C#还允许省略起始大括号之前的圆括号。下面这行代码生成与上一行相同的 IL：

```
String s = new Employee { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

如果属性的类型实现了 **IEnumerable** 或 **IEnumerable<T>** 接口，属性就被认为是集合，而集合的初始化是一种相加(additive)操作，而非替换(replacement)操作。例如，假定有下面这个类定义：

```
public sealed class Classroom {
    private List<String> m_students = new List<String>();
    public List<String> Students { get { return m_students; } }

    public Classroom() {}
}
```

现在可以写代码来构造一个 **Classroom** 对象，并像下面这样初始化 **Students** 集合：

```
public static void M() {
    Classroom classroom = new Classroom {
        Students = { "Jeff", "Kristin", "Aidan", "Grant" }
    };

    // 显示教室中的 4 个学生
    foreach (var student in classroom.Students)
        Console.WriteLine(student);
}
```

编译上述代码时，编译器发现 **Students** 属性的类型是 **List<String>**，而且这个类型实现了 **IEnumerable<String>** 接口。现在，编译器假定 **List<String>** 类型提供了一个名为 **Add** 的方法(因为大多数集合类都提供了 **Add** 方法将数据项添加到集合)。然后，编译器生成代码来调用集合的 **Add** 方法。所以，上述代码会被编译器转换成这样：

```
public static void M() {
    Classroom classroom = new Classroom();
    classroom.Students.Add("Jeff");
    classroom.Students.Add("Kristin");
    classroom.Students.Add("Aidan");
    classroom.Students.Add("Grant");

    // 显示教室中的 4 个学生
    foreach (var student in classroom.Students)
        Console.WriteLine(student);
}
```

如果属性的类型实现了 **IEnumerable** 或 **IEnumerable<T>**，但未提供 **Add** 方法，编译器就不允许使用集合初始化语法向集合中添加数据项；相反，编译器报告以下消息：**error CS0117: "System.Collections.Generic.IEnumerable<string>" 不包含 "Add" 的定义。**

有的集合的 **Add** 方法要获取多个实参，比如 **Dictionary** 的 **Add** 方法：

```
public void Add(TKey key, TValue value);
```

通过在集合初始化器中嵌套大括号的方式，可向 **Add** 方法传递多个实参，如下所示：

```
var table = new Dictionary<String, Int32> {
    { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }
};
```

它等价于以下代码：

```
var table = new Dictionary<String, Int32>();
table.Add("Jeffrey", 1);
table.Add("Kristin", 2);
table.Add("Aidan", 3);

table.Add("Grant", 4);
```

10.1.4 匿名类型

利用 C# 的匿名类型功能，可以用很简洁的语法来自动声明不可变(**immutable**)的元组类型。元组^①类型是含有一组属性的类型，这些属性通常以某种方式相互关联。在以下代码的第一行中，我定义了含有两个属性(**String** 类型的 **Name** 和 **Int32** 类型的 **Year**)的类，构造了该类型的实例，将 **Name** 属性设为 **"Jeff"**，将 **Year** 属性设为 **1964**。

```
// 定义类型，构造实例，并初始化属性
var ol = new { Name = "Jeff", Year = 1964 };

// 在控制台上显示属性: Name=Jeff, Year=1964
Console.WriteLine("Name={0}, Year={1}", ol.Name, ol.Year);
```

① 元组 (tuple) 一词来源于对顺序的抽象: single, double, triple, quadruple, quintuple, *n*-tuple。

第一行代码创建了匿名类型，我没有在 **new** 关键字后指定类型名称，所以编译器会自动创建类型名称，而且不会告诉我这个名称具体是什么(这正是匿名的含义)。这行代码使用上一节讨论的“对象初始化器”语法来声明属性，同时初始化这些属性。另外，由于我(开发人员)不知道编译时的类型名称，也就不知道变量 **o1** 声明为什么类型。但这不是问题，因为可以像第 9 章讨论过的那样使用 C# 的“隐式类型局部变量”功能(**var**)。它的作用是告诉编译器根据赋值操作符(=)右侧的表达式推断类型。

现在，让我们将注意力放在编译器实际做的事情上面。遇到下面这样的一行代码时：

```
var o = new { property1 = expression1, ..., propertyN = expressionN };
```

编译器会推断每个表达式的类型，创建推断类型的私有字段，为每个字段创建公共只读属性，并创建一个构造器来接受所有这些表达式。在构造器的代码中，会用传给它的表达式的求值结果来初始化私有只读字段。除此之外，编译器还会重写 **Object** 的 **Equals**，**GetHashCode** 和 **ToString** 方法，并生成所有这些方法中的代码。最终，编译器生成的类看起来像这样：

```
[CompilerGenerated]
internal sealed class <>f__AnonymousType0<...>: Object {
    private readonly tl fl;
    public tl pl { get { return fl; } }

    ...

    private readonly tn fn;
    public tn pn { get { return fn; } }

    public <>f__AnonymousType0<...>(tl a1, ..., tn an) {
        fl = a1; ...; fn = an; // 设置所有字段
    }

    public override Boolean Equals(Object value) {
        // 任何字段不匹配就返回 false，否则返回 true
    }

    public override Int32 GetHashCode() {
        // 返回根据每个字段的哈希码生成的一个哈希码
    }

    public override String ToString() {
        // 返回“属性名=值”对的以逗号分隔的列表
    }
}
```

编译器会生成 **Equals** 和 **GetHashCode** 方法，因此匿名类型的实例能放到哈希表集合中。属性是只读的，而非可读可写，目的是防止对象的哈希码发生改变。如果对象在哈希表中作为键使用，更改它的哈希码会造成再也找不到它。编译器会生成 **ToString** 方法来帮助进行调试。在 **Visual Studio** 调试器中，可将鼠标指针放在引用了匿名类型实例的一个变量上方。随后，**Visual Studio** 会调用 **ToString** 方法，并在一个提示窗口中显示结果字符串。顺便说一句，在编辑器中写代码时，**Visual Studio** 的“智能感知”功能会提示属性名，这是非常好用的一个功能。

编译器支持用另外两种语法声明匿名类型中的属性，能根据变量推断属性名和类型：

```
String Name = "Grant";
DateTime dt = DateTime.Now;

// 有两个属性的一个匿名类型
// 1. String Name 属性设为"Grant"
// 2. Int32 Year 属性设为 dt 中的年份
var o2 = new { Name, dt.Year };
```

在这个例子中，编译器判断第一个属性应该叫 **Name**。由于 **Name** 是局部变量的名称，所以编译器将属性类型设为与局部变量相同的类型：**String**。对于第二个属性，编译器使用字段/属性的名称：**Year**。**Year** 是 **DateTime** 类的一个 **Int32** 属性，所以匿名类型中的 **Year** 属性也是一个 **Int32**。现在，当编译器构造这个匿名类型的一个实例时，会将实例的 **Name** 属性设为 **Name** 局部变量中的值，使 **Name** 属性引用同一个“Grant”字符串。编译器还要将实例的 **Year** 属性设为与 **dt** 的 **Year** 属性返回的值相同。

编译器在定义匿名类型时是非常“善解人意”的。如果它看到你在源代码中定义了多个匿名类型，而且这些类型具有相同的结构，那么它只会创建一个匿名类型定义，但创建该类型的多个实例。所谓“相同的结构”，是指在这些匿名类型中，每个属性都有相同的类型和名称，而且这些属性的指定顺序相同。在前面的几个例子中，变量 **o1** 和 **o2** 就是同类型的，因为在定义匿名类型的两行代码中，都是先一个 **String** 类型的 **Name** 属性，再一个 **Int32** 类型的 **Year** 属性。

由于两个变量(**o1** 和 **o2**)的类型相同，所以可以做一些非常“酷”的事情，比如检查两个对象是否包含相等的值，并将对一个对象的引用赋给正在指向另一个对象的变量，如下所示：

```
// 一个类型允许相等测试和赋值操作
Console.WriteLine("Objects are equal: " + o1.Equals(o2));
o1 = o2; // 赋值
```

另外，由于这种类型的同一性，所以可以创建一个隐式类型的数组(详情参见 16.1 节“初始化数组元素”)，在其中包含一组匿名类型的对象。

```
// 之所以能这样写，是因为所有对象都是相同的匿名类型
var people = new[] {
    o1, // o1 的定义参见本节开头
    new { Name = "Kristin", Year = 1970 },
    new { Name = "Aidan", Year = 2003 },
    new { Name = "Grant", Year = 2008 }
};
```

```
// 下面展示如何遍历匿名类型的对象构成的数组 (var 是必须的)
foreach (var person in people)
    Console.WriteLine("Person={0}, Year={1}", person.Name, person.Year);
```

匿名类型经常与 LINQ(Language Intergrated Query, 语言集成查询)配合使用。可用 LINQ 执行查询，从而生成由一组对象构成的集合，这些对象都是相同的匿名类型。然后，可以对结果集中的对象进行处理。所有这些都是同一个方法中发生的。下例展示了如何返回“我的文档”文件夹中过去 7 天修改过的所有文件：

```
String myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var query =
    from pathname in Directory.GetFiles(myDocuments)
    let LastWriteTime = File.GetLastWriteTime(pathname)
    where LastWriteTime > (DateTime.Now - TimeSpan.FromDays(7))
    orderby LastWriteTime
    select new { Path = pathname, LastWriteTime }; // 匿名类型的对象构成的集合

foreach (var file in query)
    Console.WriteLine("LastWriteTime={0}, Path={1}", file.LastWriteTime, file.Path);
```

匿名类型的实例不能泄露到方法外部。方法原型不能接受匿名类型的参数，因为无法指定匿名类型。类似地，方法也不能返回对匿名类型的引用。虽然可以将匿名类型的实例视为一个 **Object** (所有匿名类型都从 **Object** 派生)，但没办法将 **Object** 类型的变量转型回匿名类型，因为不知道在匿名类型在编译时的名称。要传递元组，应考虑使用下一节讨论的 **System.Tuple** 类型。

10.1.5 System.Tuple 类型

在 **System** 命名空间，Microsoft 定义了几个泛型 **Tuple** 类型，它们全部从 **Object** 派生，区别只在于元数^① (泛型参数的个数)。下面演示了最简单和最复杂的 **Tuple** 类型：

```
// 这是最简单的：
[Serializable]
public class Tuple<T1> {
    private T1 m_Item1;
    public Tuple(T1 item1) { m_Item1 = item1; }
    public T1 Item1 { get { return m_Item1; } }
}

// 这是最复杂的：
[Serializable]
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> {
    private T1 m_Item1; private T2 m_Item2; private T3 m_Item3; private T4 m_Item4;
    private T5 m_Item5; private T6 m_Item6; private T7 m_Item7; private TRest m_Rest;
    public Tuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7,
        TRest rest) {
        m_Item1 = item1; m_Item2 = item2; m_Item3 = item3; m_Item4 = item4;
        m_Item5 = item5; m_Item6 = item6; m_Item7 = item7; m_Rest = rest;
    }

    public T1 Item1 { get { return m_Item1; } }
    public T2 Item2 { get { return m_Item2; } }
    public T3 Item3 { get { return m_Item3; } }
    public T4 Item4 { get { return m_Item4; } }
    public T5 Item5 { get { return m_Item5; } }
    public T6 Item6 { get { return m_Item6; } }
}
```

① 元数的英文是 **arity**。在计算机编程中，一个函数或运算(操作)的元数是指函数获取的实参或操作数的个数。它源于像 **unary(arity=1)**、**binary(arity=2)**、**ternary(arity=3)** 这样的单词。——译注

```

public T7 Item7 { get { return m_Item7; } }
public TRest Rest { get { return m_Rest; } }
}

```

和匿名类型相似，**Tuple** 创建好之后就不可变了(所有属性都只读)。虽然这里没有显示，但 **Tuple** 类还提供了 **CompareTo**、**Equals**、**GetHashCode** 和 **ToString** 方法以及 **Size** 属性。此外，所有 **Tuple** 类型都实现了 **IStructuralEquatable**、**IStructuralComparable** 和 **IComparable** 接口，所以可以比较两个 **Tuple** 对象，对它们的字段进行比对。请参考文档更多地了解这些方法和接口。

下面的示例泛型方法 **MinMax** 用一个 **Tuple** 类型向调用者返回两样信息：

```

// 用 Item1 返回最小值 & 用 Item2 返回最大值
private static Tuple<Int32, Int32> MinMax(Int32 a, Int32 b) {
    return new Tuple<Int32, Int32>(Math.Min(a, b), Math.Max(a, b));
}

// 下面展示了如何调用方法，以及如何使用返回的 Tuple
private static void TupleTypes() {
    var minmax = MinMax(6, 2);

    // Min=2, Max=6
    Console.WriteLine("Min={0}, Max={1}", minmax.Item1, minmax.Item2);
}

```

当然，很重要的一点是 **Tuple** 的生产者(写它的人)和消费者(用它的人)必须对 **Item#** 属性返回的内容有一个清楚的理解。对于匿名类型，属性的实际名称是根据定义匿名类型的源代码来确定的。对于 **Tuple** 类型，属性一律被 Microsoft 称为 **Item#**，我们无法对此进行任何改变。遗憾的是，这种名字没有任何实际的含义或意义，所以要由生产者和消费者为它们分配具体含义。这还影响了代码的可读性和可维护性。所以，应该在自己的代码添加详细的注释，说明每个 **Item#** 代表着什么。

编译器只能在调用泛型方法时推断泛型类型，调用构造器时不能。因此，**System** 命名空间还包含了一个非泛型静态 **Tuple** 类，其中包含一组静态 **Create** 方法，能根据实参推断泛型类型。这个类扮演了创建 **Tuple** 对象的一个工厂的角色，它存在的唯一意义便是简化你的代码。下面用静态 **Tuple** 类重写了刚才的 **MinMax** 方法：

```

// 用 Item1 返回最小值 & 用 Item2 返回最大值
private static Tuple<Int32, Int32> MinMax(Int32 a, Int32 b) {
    return Tuple.Create(Math.Min(a, b), Math.Max(a, b)); // 更简单的语法
}

```

要创建多于 8 个元素的 **Tuple**，可为 **Rest** 参数传递另一个 **Tuple**，如下所示：

```

var t = Tuple.Create(0, 1, 2, 3, 4, 5, 6, Tuple.Create(7, 8));
Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}",
    t.Item1, t.Item2, t.Item3, t.Item4, t.Item5, t.Item6, t.Item7,
    t.Rest.Item1.Item1, t.Rest.Item1.Item2);

```



注意 除了匿名类型和 **Tuple** 类型, 还可研究一下 **System.Dynamic.ExpandoObject** 类(在 **System.Core.dll** 程序集中定义)。这个类和 C# 的 **dynamic** 类型(参见 5.5 节“**dynamic** 基元类型”)配合使用, 就可用另一种方式将一系列属性(键/值对)组合到一起。虽然实现不了编译时的类型安全性, 但语法看起来不错(虽然得不到“智能感知”支持), 而且还可以在 C# 和 Python 这样的动态语言之间传递 **ExpandoObject** 对象。以下是使用了一个 **ExpandoObject** 的示例代码:

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6;           // 添加一个 Int32 'x' 属性, 其值为 6
e.y = "Jeff";      // 添加一个 String 'y' 属性, 其值为 "Jeff"
e.z = null;        // 添加一个 Object 'z' 属性, 其值为 null

// 查看所有属性及其值:
foreach (var v in (IDictionary<String, Object>)e)
    Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);

// 删除 'x' 属性及其值:
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

10.2 有参属性

在上一节中, 属性的 **get** 访问器方法不接受参数, 因此称为**无参属性**。由于用起来就像访问字段, 所以很容易理解。除了这些与字段相似的属性, 编程语言还支持**有参属性**, 它的 **get** 访问器方法接受一个或多个参数, **set** 访问器方法接受两个或多个参数。不同编程语言以不同方式公开有参属性。另外, 不同编程语言对有参属性的称呼也不同。C# 称为**索引器**, Visual Basic 则称为**默认属性**。本节主要讨论 C# 如何使用有参属性来公开索引器。

C# 使用数组风格的语法来公开有参属性(索引器)。换句话说, 可将索引器看成是 C# 开发人员对 **[]** 操作符的重载。下面是一个示例 **BitArray** 类, 它允许用数组风格的语法来索引由该类的实例维护的一组二进制位。

```
using System;

public sealed class BitArray {
    // 容纳了二进制的私有字节数组
    private Byte[] m_byteArray;
    private Int32 m_numBits;

    // 下面的构造器用于分配字节数组, 并将所有位设为 0
    public BitArray(Int32 numBits) {
        // 先验证实参
        if (numBits <= 0)
            throw new ArgumentOutOfRangeException("numBits must be > 0");

        // 保存位的个数
        m_numBits = numBits;
    }
}
```

```

// 为位数组分配字节
m_byteArray = new Byte[(numBits + 7) / 8];
}

// 下面是索引器(有参属性)
public Boolean this[Int32 bitPos] {

    // 下面是索引器的 get 访问器方法
    get {
        // 先验证实参
        if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos");

        // 返回指定索引处的位的状态
        return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0;
    }

    // 下面是索引器的 set 访问器方法
    set {
        if ((bitPos < 0) || (bitPos >= m_numBits))
            throw new ArgumentOutOfRangeException("bitPos",
                bitPos.ToString());

        if (value) {
            // 将指定索引处的位设为 true
            m_byteArray[bitPos / 8] = (Byte)
                (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
        } else {
            // 将指定索引处的位设为 false
            m_byteArray[bitPos / 8] = (Byte)
                (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
        }
    }
}
}
}

```

BitArray 类的索引器用起来很简单:

```

// 分配含 14 个位的 BitArray 数组
BitArray ba = new BitArray(14);

// 调用 set 访问器方法, 将编号为偶数的所有位都设为 true
for (Int32 x = 0; x < 14; x++) {
    ba[x] = (x % 2 == 0);
}

// 调用 get 访问器方法显示所有位的状态
for (Int32 x = 0; x < 14; x++) {
    Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}

```

在这个示例 **BitArray** 类中, 索引器获取 **Int32** 类型的参数 **bitPos**。所有索引器至少要有一个参数, 但可以有更多。这些参数(和返回类型)可以是除了 **void** 之外的任意类型。在 **System.Drawing.dll** 程序集的 **System.Drawing.Imaging.ColorMatrix** 类中, 提供了有多个参数的一个索引器的例子。

经常要创建索引器来查询关联数组^①中的值。**System.Collections.Generic.Dictionary** 类型就

① 关联数组(associative array)使用字符串索引(称为键)来访问存储在数组中的值(称为值)。——译注

提供了这样的一个索引器，它获取一个键，并返回与该键关联的值。和无参属性不同，类型可提供多个重载的索引器，只要这些索引器的签名不同。

和无参属性的 **set** 访问器方法相似，索引器的 **set** 访问器方法同样包含了一个隐藏参数，在 C# 中称为 **value**。该参数代表想赋给“被索引元素”的新值。

CLR 本身不区分无参属性和有参属性。对 CLR 来说，每个属性都只是类型中定义的一对方法和一些元数据。如前所述，不同的编程语言要求用不同语法来创建和使用有参属性。将 **this[...]** 作为表达索引器的语法，这纯粹是 C# 团队自己的选择。也正是因为这个选择，所以 C# 只允许在对象的实例上定义索引器。C# 不支持定义静态索引器属性，虽然 CLR 是支持静态有参属性的。

由于 CLR 以相同的方式对待有参和无参属性，所以编译器会在托管程序集中生成以下两项或三项。

- 代表有参属性 **get** 访问器的方法。仅在属性定义了 **get** 访问器方法时生成。
- 代表有参属性 **set** 访问器的方法。仅在属性定义了 **set** 访问器方法时生成。
- 托管程序集元数据中的属性定义。这一项必然生成。没有专门的有参属性元数据定义表，因为对于 CLR 来说，有参属性不过就是属性。

对于前面的 **BitArray** 类，编译器在编译索引器时，原始代码似乎是像下面这样写的：

```
public sealed class BitArray {

    // 下面是索引器的 get 访问器方法
    public Boolean get_Item(Int32 bitPos) { /* ... */ }

    // 下面是索引器的 set 访问器方法
    public void set_Item(Int32 bitPos, Boolean value) { /* ... */ }
}
```

编译器在索引器名称之前附加 **get_** 或者 **set_** 前缀，从而自动生成这些方法的名称。由于 C# 的索引器语法不允许开发人员指定索引器名称，所以 C# 编译器团队不得不为访问器方法选择一个默认名称；他们最后选择了 **Item**。因此，编译器生成的方法名就是 **get_Item** 和 **set_Item**。

查看文档时，留意类型是否提供了名为 **Item** 的属性，从而判断该类型是否提供了索引器。例如，**System.Collections.Generic.List** 类型提供了名为 **Item** 的公共实例属性，它就是 **List** 的索引器。

用 C# 编程永远看不到 **Item** 这个名称，所以一般无需关心编译器选择的是什么名称。但如果为类型设计的索引器要由其他语言的代码访问，就可能需要更改索引器的 **get** 和 **set** 访问器方法所用的默认名称 **Item**。C# 允许向索引器应用定制特性 **System.Runtime.CompilerServices.IndexerNameAttribute** 来重命名这些方法，如下所示：

```
using System;
using System.Runtime.CompilerServices;

public sealed class BitArray {

    [IndexerName("Bit")]
    public Boolean this[Int32 bitPos] {
        // 这里至少要定义一个访问器方法
    }
}
```

现在, 编译器将生成名为 **get_Bit** 和 **set_Bit** 的方法, 而不是 **get_Item** 和 **set_Item**。编译时, C#编译器会注意到 **IndexerName** 特性, 它告诉编译器如何对方法和属性的元数据进行命名。特性本身不会进入程序集的元数据^①。

以下 Visual Basic 代码演示了如何访问这个 C#索引器:

```
' 构造 BitArray 类型的实例
Dim ba as New BitArray(10)

' Visual Basic 用()而不是[]指定数组元素
Console.WriteLine(ba(2)) ' 显示 True 或 False

' Visual Basic 还允许通过索引器的名称来访问它
Console.WriteLine(ba.Bit(2)) ' 显示的内容和上一行代码相同
```

C#允许一个类型定义多个索引器, 只要索引器的参数集不同。在其他语言中, **IndexerName** 特性允许定义多个相同签名的索引器, 因为索引器各自可以有不同的名称。C#不允许这样做, 是因为它的语法不是通过名称来引用索引器, 编译器不知道你引用的是哪个索引器。编译以下 C#代码将导致编译器报错: **error C0111: 类型"SomeType"已定义了一个名为 "this"的具有相同参数类型的成员。**

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {
    // 定义 get_Item 访问器方法
    public Int32 this[Boolean b] {
        get { return 0; }
    }

    // 定义 get_Jeff 访问器方法
    [IndexerName("Jeff")]
    public String this[Boolean b] {
        get { return null; }
    }
}
```

显然, C#将索引器看成是对[]操作符的重载, 而[]操作符不能用来消除具有不同方法名和相同参数集的有参属性的歧义。

顺便说一句, **System.String** 类型是改变了索引器名称的一个例子。**String** 的索引器的名称

① 这个原因造成 **IndexerNameAttribute** 类不是 CLI 和 C#语言的 ECMA 标准的一部分。

是 **Chars**，而不是 **Item**。这个只读属性允许从字符串中获得一个单独的字符。对于不用[] 操作符语法来访问这个属性的编程语言，**Chars** 是更有意义的名称。

选择主要的有参属性

C#对索引器的限制引起以下两个问题。

- 如果定义类型的语言允许多个有参属性应该怎么办？
- 从 C#中如何使用这个类型？

这两个问题的答案是类型必须选择其中一个有参属性名来作为默认(主要)属性,这要求向类本身应用 **System.Reflection.DefaultMemberAttribute** 的一个实例。要说明的是,该特性可应用于类、结构或接口。在 C#中编译定义了有参属性的类型时,编译器会自动向类型应用该特性的实例,同时会考虑到你可能应用了 **IndexerName** 特性的情况。在 **DefaultMemberAttribute** 特性的构造器中,指定了要作为类型的默认有参属性使用的名称。

因此,如果在 C#中定义包含有参属性的类型,但没有指定 **IndexerName** 特性,那么在向该类型应用的 **DefaultMember** 特性中,会将默认成员的名称指定为 **Item**。如果向有参属性应用了 **IndexerName** 特性,那么在向该类型应用的 **DefaultMember** 特性中,会将默认成员的名称指定为由 **IndexerName** 特性指定的字符串名称。记住,如果代码中含有多个名称不一样的有参属性,C#将无法编译。

对于支持多个有参属性的编程语言,必须从中选择一个属性方法名,并用 **DefaultMember** 特性来标识。这是 C#代码唯一能访问的有参属性。

C#编译器发现代码试图获取或设置索引器时,会生成对其中一个方法的调用。有的语言不支持有参属性。要从这种语言中访问有参属性,必须显式调用需要的索引器方法。对于 CLR,无参和有参属性是无区别的,所以可用相同的 **System.Reflection.PropertyInfo** 类来发现有参属性和它的访问器方法之间的关联。

10.3 调用属性访问器方法时的性能

对于简单的 **get** 和 **set** 访问器方法,JIT 编译器会将代码内联(**inline**,或者说嵌入)。这样一来,使用属性(而不是使用字段)就没有性能上的损失。内联是指将方法(目前说的是访问器方法)的代码直接编译到调用它的方法中。这就避免了在运行时发出调用所产生的开销,代价是编译好的方法变得更大。由于属性访问器方法包含的代码一般很少,所以对内联会使生成的本机代码变得更小,而且执行得更快。



注意 JIT 编译器在调试代码时不会内联属性方法，因为内联的代码会变得难以调试。这意味着在程序的发行版本中，访问属性时的性能可能比较快；在程序的调试版本中，可能比较慢。字段访问在调试和发布版本中，速度都很快。

10.4 属性访问器的可访问性

有时希望为 **get** 访问器方法指定一种可访问性，为 **set** 访问器方法指定另一种可访问性。最常见的情形是提供公共 **get** 访问器和受保护 **set** 访问器。

```
public class SomeType {
    private String m_name;
    public String Name {
        get { return m_name; }
        protected set { m_name = value; }
    }
}
```

如上述代码所示，**Name** 属性本身声明为 **public** 属性，意味着 **get** 访问器方法是公共的，所有代码都能调用。而 **set** 访问器方法声明为 **protected**，只能从 **SomeType** 内部定义的代码中调用，或者从 **SomeType** 的派生类的代码中调用。

定义属性时，如果两个访问器方法需要不同的可访问性，C# 要求必须为属性本身指定限制最小的可访问性。然后，两个访问器只能选择一个来使用限制较大的。在前面的例子中，属性本身声明为 **public**，而 **set** 访问器方法声明为 **protected**（限制较 **public** 大）。

10.5 泛型属性访问器方法

既然属性本质上是方法，而 C# 和 CLR 允许泛型方法，所以有时可能想在定义属性时引入它自己的泛型类型参数（而非使用包容类型的泛型类型参数），但 C# 不允许。之所以属性不能引入它自己的泛型类型参数，最主要的原因是概念上说不通。属性本来应该表示可供查询或设置的某个对象特征。一旦引入泛型类型的参数，就意味着有可能改变查询/设置行为。但属性不应该和行为沾边。公开对象的行为——无论是不是泛型——都应该定义方法而非属性。

第 11 章 事 件

本章内容：

- 设计公开事件的类型
- 编译器如何实现事件
- 设计侦听事件的类型
- 显式实现事件

本章讨论可以在类型中定义的最后一种成员：事件。定义了事件成员的类型允许类型(或类型的实例)通知其他对象发生了特定的事情。例如，**Button** 类提供了 **Click** 事件。应用程序中的一个或多个对象可接收关于该事件的通知，以便在 **Button** 被单击之后采取特定操作。我们用事件这种类型成员来实现这种交互。具体地说，定义了事件成员的类型能提供以下功能。

- 方法能登记它对事件的关注。
- 方法能注销它对事件的关注。
- 事件发生时，登记了的方法将收到通知。

类型之所以能提供事件通知功能，是因为类型维护了一个已登记方法的列表。事件发生后，类型将通知列表中所有已登记的方法。

CLR 事件模型以委托为基础。委托是调用^①回调方法的一种类型安全的方式。对象凭借回调方法接收它们订阅的通知。本章会开始使用委托，但委托的完整细节是在第 17 章“委托”中讲述的。

为了帮你完整地理解事件在 CLR 中的工作机制，先来描述事件很有用的一个场景。假定要设计一个电子邮件应用程序。电子邮件到达时，用户可能希望将该邮件转发给传真机或寻呼机。先设计名为 **MailManager** 的类型来接收传入的电子邮件，它公开 **NewMail** 事件。

① 这个“调用”(invoke)理解为“唤出”更恰当。它和普通的“调用”(call)稍有不同。在英语的语境中，invoke 和 call 的区别在于，在执行一个所有信息都已知的方法时，用 call 比较恰当。这些信息包括要引用的类型、方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用 invoke 就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译，只是在必要的时候附加英文原文提醒你区分。——译注

其他类型(如 **Fax** 和 **Pager**)的对象登记对于该事件的关注。**MailManager** 收到新电子邮件会引发该事件, 造成邮件分发给每一个已登记的对象。每个对象都用它们自己的方式处理邮件。

应用程序初始化时只实例化一个 **MailManager** 实例, 然后可以实例化任意数量的 **Fax** 和 **Pager** 对象。图 11-1 展示了应用程序如何初始化, 以及新电子邮件到达时发生的事情。

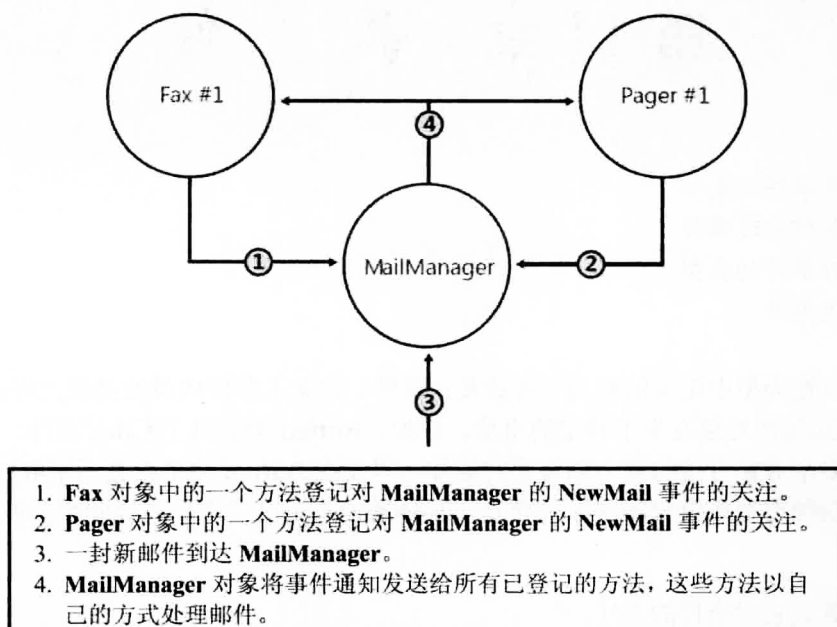


图 11-1 设计使用了事件的应用程序

图 11-1 的应用程序首先构造 **MailManager** 的一个实例。**MailManager** 提供了 **NewMail** 事件。构造 **Fax** 和 **Pager** 对象时, 它们向 **MailManager** 的 **NewMail** 事件登记自己的一个实例方法。这样当新邮件到达时, **MailManager** 就知道通知 **Fax** 和 **Pager** 对象。**MailManager** 将来收到新邮件时会引发 **NewMail** 事件, 使所有已登记的方法都有机会以自己的方式处理邮件。

11.1 设计要公开事件的类型

开发人员通过连续几个步骤定义公开了一个或多个事件成员的类型。本节详细描述了每个必要的步骤。**MailManager** 示例应用程序(可从 <http://wintellect.com> 下载)展示了 **MailManager** 类型、**Fax** 类型和 **Pager** 类型的所有源代码。注意, **Pager** 类型和 **Fax** 类型几乎完全相同。

11.1.1 第一步: 定义类型来容纳所有需要发送给事件通知接收者的附加信息

事件引发时, 引发事件的对象可能希望向接收事件通知的对象传递一些附加信息。这些附加信息需要封装到它自己的类中, 该类通常包含一组私有字段, 以及一些用于公开这些字

段的只读公共属性。根据约定, 这种类型应该从 **System.EventArgs** 派生, 而且类名以 **EventArgs** 结束。本例将该类命名为 **NewMailEventArgs** 类, 它的各个字段分别标识了发件人(**m_from**)、收件人(**m_to**)和主题(**m_subject**)。

// 第一步: 定义一个类型来容纳所有应该发送给事件通知接收者的附加信息

```
internal class NewMailEventArgs : EventArgs {

    private readonly String m_from, m_to, m_subject;

    public NewMailEventArgs(String from, String to, String subject) {
        m_from = from; m_to = to; m_subject = subject;
    }

    public String From { get { return m_from; } }
    public String To { get { return m_to; } }
    public String Subject { get { return m_subject; } }
}
```



注意 **EventArgs** 类在 Microsoft .NET Framework 类库(FCL)中定义, 其实现如下:

```
[ComVisible(true), Serializable]
public class EventArgs {
    public static readonly EventArgs Empty = new EventArgs();
    public EventArgs() { }
}
```

可以看出, 该类型的实现非常简单, 就是一个让其他类型继承的基类型。许多事件都没有附加信息需要传递。例如, 当一个 **Button** 向已登记的接收者通知自己被单击时, 调用回调方法就可以了。定义不需要传递附加数据的事件时, 可直接使用 **EventArgs.Empty**, 不用构造新的 **EventArgs** 对象。

11.1.2 第二步: 定义事件成员

事件成员使用 C# 关键字 **event** 定义。每个事件成员都要指定以下内容: 可访问性标识符(几乎肯定是 **public**, 这样其他代码才能访问该事件成员); 委托类型, 指出要调用的方法的原型; 以及名称(可以是任何有效的标识符)。以下是我们的 **MailManager** 类中的事件成员:

```
internal class MailManager {

    // 第二步: 定义事件成员
    public event EventHandler<NewMailEventArgs> NewMail;
    ...
}
```

NewMail 是事件名称。事件成员的类型是 **EventHandler<NewMailEventArgs>**, 意味着“事件通知”的所有接收者都必须提供一个原型和 **EventHandler<NewMailEventArgs>** 委托类型匹配的回调方法。由于泛型 **System.EventHandler** 委托类型的定义如下:

```
public delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e);
```

所以方法原型必须具有以下形式:

```
void MethodName(Object sender, NewMailEventArgs e);
```