事实上，如果使用了 **checked** 语句块，就可将+=操作符用于 **Byte**，稍微简化一下代码：

```
checked {                          // 开始 checked 块
   Byte b = 100;
   b += 200;                       // 该表达式会进行溢出检查
}                                  // 结束 checked 块
```



重要提示 由于 **checked** 操作符和 **checked** 语句唯一的作用就是决定生成哪个版本的加、减、乘和数据转换 IL 指令，所以在 **checked** 操作符或语句中调用方法，不会对该方法造成任何影响，如下例所示：

```
checked {
   // 假定 SomeMethod 试图把 400 加载到一个 Byte 中
   SomeMethod(400);
   // SomeMethod 可能会、也可能不会抛出 OverflowException 异常
   // 如果 SomeMethod 使用 checked 指令编译，就可能会抛出异常
   // 但这和当前的 checked 语句无关
}
```

根据我的经验，许多计算都会产生令人吃惊的结果。这一般是由于无效的用户输入，但也可能是由于系统的某个部分返回了程序员没有预料到的值。所以我对程序员有以下建议。

1. 尽量使用有符号数值类型(比如 **Int32** 和 **Int64**)而不是无符号数值类型(比如 **UInt32** 和 **UInt64**)。这允许编译器检测更多的上溢/下溢错误。除此之外，类库多个部分(比如 **Array** 和 **String** 的 **Length** 属性)被硬编码为返回有符号的值。这样在代码中四处移动这些值时，需要进行的强制类型转换就少了。较少的强制类型转换使代码更整洁，更容易维护。除此之外，无符号数值类型不符合 CLS。

2. 写代码时，如果代码可能发生你不希望的溢出(可能是因为无效的输入，比如要求使用最终用户或客户机提供的数据)，就把这些代码放到 **checked** 块中。同时捕捉 **OverflowException**，得体地从错误中恢复。

3. 写代码时，将允许发生溢出的代码显式放到 **unchecked** 块中，比如在计算校验和时。

4. 对于没有使用 **checked** 或 **unchecked** 的任何代码，都假定你希望在发生溢出时抛出一个异常，比如在输入已知的前提下计算一些东西(比如质数)，此时的溢出应被计为 bug。

开发应用程序时，打开编译器的**/checked+**开关进行调试性生成。这样系统会对没有显式标记 **checked** 或 **unchecked** 的代码进行溢出检查，所以应用程序运行起来会慢一些。此时一旦发生异常，就可以轻松检测到，而且能及时修正代码中的 bug。但是，为了正式发布而生成应用程序时，应使用编译器的**/checked-**开关，确保代码能更快运行，不会产生溢出异常。要在 Microsoft Visual Studio 中更改 Checked 设置，请打开项目的属性页，点击“生成”标签，单击“高级”，再勾选“检查运算上溢/下溢”，如图 5-1 所示。



图 5-1　在 Visual Studio 的“高级生成设置”对话框中指定编译器是否检查溢出

如果应用程序能容忍总是执行 **checked** 运算而带来的轻微性能损失，建议即使是为了发布而生成应用程序，也用**/checked** 命令行开关进行编译，这样可防止应用程序在包含已损坏的数据(甚至可能是安全漏洞)的前提下继续运行。例如，通过乘法运算来计算数组索引时，相较于因为数学运算的“回滚”[①]而访问到不正确的数组元素，抛出 **OverflowException** 异常才是更好的做法。



重要提示　**System.Decimal** 是非常特殊的类型。虽然许多编程语言(包括 C# 和 Visual Basic)将 **Decimal** 视为基元类型，但 CLR 不然。这意味着 CLR 没有知道如何处理 **Decimal** 值的 IL 指令。在文档中查看 **Decimal** 类型，可以看到它提供了一系列 **public static** 方法，包括 **Add**，**Subtract**，**Multiply**，**Divide** 等。此外，**Decimal** 类型还为+，-，*，/等提供了操作符重载方法。

编译使用了 **Decimal** 值的程序时，编译器会生成代码来调用 **Decimal** 的成员，并通过这些成员来执行实际运算。这意味着 **Decimal** 值的处理速度慢于 CLR 基元类型的值。另外，由于没有相应的 IL 指令来处理 **Decimal** 值，所以 **checked** 和 **unchecked** 操作符、语句以及编译器开关都失去了作用。如果对 **Decimal** 值执行的运算是不安全的，肯定会抛出 **OverflowException** 异常。

类似地，**System.Numerics.BigInteger** 类型也在内部使用 **UInt32** 数组来表示任意大的整数，它的值没有上限和下限。因此，对 **BigInteger** 执行的运算永远不会造成 **OverflowException** 异常。但如果值太大，没有足够多的内存来改变数组大小，对 **BigInteger** 的运算可能抛出 **OutOfMemoryException** 异常。

① 乘法运算可能产生一个较大的值，超出数组的索引范围。参见上一条关于“wrap”的注释。——译注

5.2 引用类型和值类型

CLR 支持两种类型：**引用类型**和**值类型**。虽然 FCL 的大多数类型都是引用类型，但程序员用得最多的还是值类型。引用类型总是从托管堆分配，C#的 **new** 操作符返回对象内存地址——即指向对象数据的内存地址。使用引用类型必须留意性能问题。首先要认清楚以下四个事实。

1. 内存必须从托管堆分配。
2. 堆上分配的每个对象都有一些额外成员，这些成员必须初始化。
3. 对象中的其他字节(为字段而设)总是设为零。
4. 从托管堆分配对象时，可能强制执行一次垃圾回收。

如果所有类型都是引用类型，应用程序的性能将显著下降。设想每次使用 **Int32** 值时都进行一次内存分配，性能会受到多么大的影响！为了提升简单和常用的类型的性能，CLR 提供了名为“值类型”的轻量级类型。值类型的实例一般在线程栈上分配(虽然也可作为字段嵌入引用类型的对象中)。在代表值类型实例的变量中不包含指向实例的指针。相反，变量中包含了实例本身的字段。由于变量已包含了实例的字段，所以操作实例中的字段不需要提领指针。值类型的实例不受垃圾回收器的控制。因此，值类型的使用缓解了托管堆的压力，并减少了应用程序生存期内的垃圾回收次数。

文档清楚指出哪些类型是引用类型，哪些是值类型。在文档中查看类型时，任何称为“类”的类型都是引用类型。例如，System.Exception 类、System.IO.FileStream 类以及 System.Random 类都是引用类型。相反，所有值类型都称为结构或枚举。例如，System.Int32 结构、System.Boolean 结构、System.Decimal 结构、System.TimeSpan 结构、System.DayOfWeek 枚举、System.IO.FileAttributes 枚举以及 System.Drawing.FontStyle 枚举都是值类型。

进一步研究文档，会发现所有结构都是抽象类型 **System.ValueType** 的直接派生类。**System.ValueType** 本身又直接从 **System.Object** 派生。根据定义，所有值类型都必须从 **System.ValueType** 派生。所有枚举都从 **System.Enum** 抽象类型派生，后者又从 **System.ValueType** 派生。CLR 和所有编程语言都给予枚举特殊待遇[①]。欲知枚举类型的详情，请参见第 15 章“枚举类型和位标志”。

虽然不能在定义值类型时为它选择基类型，但如果愿意，值类型可实现一个或多个接口。除此之外，所有值类型都隐式密封，目的是防止将值类型用作其他引用类型或值类型的基类型。例如，无法将 **Boolean**，**Char**，**Int32**，**UInt64**，**Single**，**Double**，**Decimal** 等作为基

① 将其视为“一等公民”，直接支持各种强大的操作。在非托管环境中，枚举就没这么“好命”了。——译注

类型来定义任何新类型。



重要提示　对于许多开发人员(比如非托管 C/C++开发人员)，最初接触引用类型和值类型时都觉得有些不解。在非托管 C/C++中声明类型后，使用该类型的代码会决定是在线程栈上还是在应用程序的堆中分配类型的实例。但在托管代码中，要由定义类型的开发人员决定在什么地方分配类型的实例，使用类型的人对此并无控制权。

以下代码和图 5-2 演示了引用类型和值类型的区别：

```
// 引用类型(因为'class')
class SomeRef { public Int32 x; }

// 值类型(因为'struct')
struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {
   SomeRef r1 = new SomeRef();      // 在堆上分配
   SomeVal v1 = new SomeVal();      // 在栈上分配
   r1.x = 5;                        // 提领指针
   v1.x = 5;                        // 在栈上修改
   Console.WriteLine(r1.x);         // 显示"5"
   Console.WriteLine(v1.x);         // 同样显示"5"
   // 图 5-2 的左半部分反映了执行以上代码之后的情况

   SomeRef r2 = r1;                 // 只复制引用(指针)
   SomeVal v2 = v1;                 // 在栈上分配并复制成员
   r1.x = 8;                        // r1.x 和 r2.x 都会更改
   v1.x = 9;                        // v1.x 会更改，v2.x 不变
   Console.WriteLine(r1.x);         // 显示"8"
   Console.WriteLine(r2.x);         // 显示"8"
   Console.WriteLine(v1.x);         // 显示"9"
   Console.WriteLine(v2.x);         // 显示"5"
   // 图 5-2 的右半部分反映了执行以上所有代码之后的情况
}
```

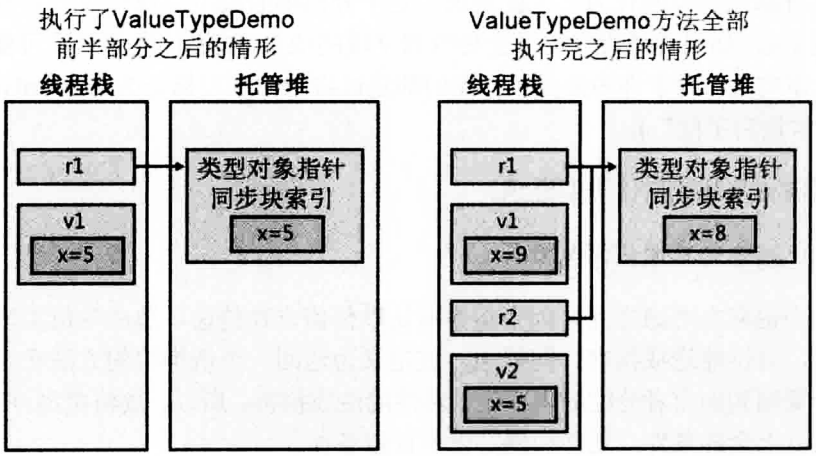


图 5-2　图解代码执行时的内存分配情况

上述代码中，**SomeVal** 类型用 **struct** 声明，而不是用更常用的 **class**。在 C#中，用 **struct** 声明的类型是值类型，用 **class** 声明的类型是引用类型。可以看出，引用类型和值类型的区别相当大。在代码中使用类型时，必须注意是引用类型还是值类型，因为这会极大地影响在代码中表达自己意图的方式。

上述代码中有这样一行：

```
SomeVal v1 = new SomeVal();     // 在栈上分配
```

因为这行代码的写法，似乎是要在托管堆上分配一个 **SomeVal** 实例。但 C#编译器知道 **SomeVal** 是值类型，所以会生成正确的 IL 代码，在线程栈上分配一个 **SomeVal** 实例。C#还会确保值类型中的所有字段都初始化为零。

上述代码还可以像下面这样写：

```
SomeVal v1;  // 在栈上分配
```

这一行生成的 IL 代码也会在线程栈上分配实例，并将字段初始化为零。唯一的区别在于，如果使用 **new** 操作符，C#会认为实例已初始化。以下代码更清楚地进行了说明：

```
// 这两行代码能通过编译，因为 C#认为
// v1 的字段已初始化为 0
SomeVal v1 = new SomeVal();
Int32 a = v1.x;

// 这两行代码不能通过编译，因为 C#不认为
// v1 的字段已初始化为 0
SomeVal v1;
Int32 a = v1.x; // error CS0170：使用了可能未赋值的字段"x"
```

设计自己的类型时，要仔细考虑类型是否应该定义成值类型而不是引用类型。值类型有时能提供更好的性能。具体地说，除非满足以下全部条件，否则不应将类型声明为值类型。

- 类型具有基元类型的行为。也就是说，是十分简单的类型，没有成员会修改类型的任何实例字段。如果类型没有提供会更改其字段的成员，就说该类型是**不可变**(immutable)类型。事实上，对于许多值类型，我们都建议将全部字段标记为 **readonly**(详情参见第 7 章“常量和字段”)。
- 类型不需要从其他任何类型继承。
- 类型也不派生出其他任何类型。

类型实例大小也应在考虑之列，因为实参默认以传值方式传递，造成对值类型实例中的字段进行复制，对性能造成损害。同样地，被定义为返回一个值类型的方法在返回时，实例中的字段会复制到调用者分配的内存中，对性能造成损害。所以，要将类型声明为值类型，除了要满足以上全部条件，还必须满足以下任意条件。

- 类型的实例较小(16 字节或更小)。
- 类型的实例较大(大于 16 字节)，但不作为方法实参传递，也不从方法返回。

值类型的主要优势是不作为对象在托管堆上分配。当然，与引用类型相比，值类型也存在自身的一些局限。下面列出了值类型和引用类型的一些区别。

- 值类型对象有两种表示形式：**未装箱**和**已装箱**，详情参见下一节。相反，引用类型总是处于已装箱形式。
- 值类型从 **System.ValueType** 派生。该类型提供了与 **System.Object** 相同的方法。但 **System.ValueType** 重写了 **Equals** 方法，能在两个对象的字段值完全匹配的前提下返回 **true**。此外，**System.ValueType** 重写了 **GetHashCode** 方法。生成哈希码时，这个重写方法所用的算法会将对象的实例字段中的值考虑在内。由于这个默认实现存在性能问题，所以定义自己的值类型时应重写 **Equals** 和 **GetHashCode** 方法，并提供它们的显式实现。本章末尾会讨论 **Equals** 和 **GetHashCode** 方法。
- 由于不能将值类型作为基类型来定义新的值类型或者新的引用类型，所以不应在值类型中引入任何新的虚方法。所有方法都不能是抽象的，所有方法都隐式密封(不可重写)。
- 引用类型的变量包含堆中对象的地址。引用类型的变量创建时默认初始化为 **null**，表明当前不指向有效对象。试图使用 **null** 引用类型变量会抛出 **NullReferenceException** 异常。相反，值类型的变量总是包含其基础类型的一个值，而且值类型的所有成员都初始化为 **0**。值类型变量不是指针，访问值类型不可能抛出 **NullReferenceException** 异常。CLR 确实允许为值类型添加“可空”(nullability)标识。可空类型将在第 19 章“可空值类型”详细讨论。
- 将值类型变量赋给另一个值类型变量，会执行逐字段的复制。将引用类型的变量赋给另一个引用类型的变量只复制内存地址。
- 基于上一条，两个或多个引用类型变量能引用堆中同一个对象，所以对一个变量执行的操作可能影响到另一个变量引用的对象。相反，值类型变量自成一体，对值类型变量执行的操作不可能影响另一个值类型变量。
- 由于未装箱的值类型不在堆上分配，一旦定义了该类型的一个实例的方法不再活动，为它们分配的存储就会被释放，而不是等着进行垃圾回收。

CLR 如何控制类型中的字段布局

为了提高性能，CLR 能按照它所选择的任何方式排列类型的字段。例如，CLR 可以在内存中重新安排字段的顺序，将对象引用分为一组，同时正确排列和填充数据字段。但在定义类型时，针对类型的各个字段，你可以告诉 CLR 是严格按照自己指定的顺序排列，还是按照 CLR 自己认为合适的方式重新排列。

为了告诉 CLR 应该怎样做，要为自己定义的类或结构应用 **System.Runtime.InteropServices.StructLayoutAttribute** 特性。可向该特性的构造器传递 **LayoutKind.Auto**，让 CLR 自动排列字段；也可传递 **LayoutKind.Sequential**，让 CLR 保持你的字段布局；也可传递 **LayoutKind.Explicit**，利用偏移量在内存中显式排列字段。如果不为自己定义的类型显式指定 **StructLayoutAttribute**，编译器会选择它自认为最好的布局。

注意，Microsoft C#编译器默认为引用类型(类)选择 **LayoutKind.Auto**，为值类型(结构)选择 **LayoutKind.Sequential**。显然，C#编译器团队认为和非托管代码互操作时会经常用到结构。为此，字段必须保持程序员定义的顺序。然而，假如创建的值类型不与非托管代码互操作，就应该覆盖 C#编译器的默认设定。下面是一个例子：

```
using System;
using System.Runtime.InteropServices;

// 让 CLR 自动排列字段以增强这个值类型的性能
[StructLayout(LayoutKind.Auto)]
internal struct SomeValType {
   private readonly Byte m_b;
   private readonly Int16 m_x;
   ...
}
```

StructLayoutAttribute 还允许显式指定每个字段的偏移量，这要求向其构造器传递 **LayoutKind.Explicit**。然后向值类型中的每个字段都应用 **System.Runtime.InteropServices.FieldOffsetAttribute** 特性的实例，向该特性的构造器传递 **Int32** 值来指出字段第一个字节距离实例起始处的偏移量(以字节为单位)。显式布局常用于模拟非托管 C/C++中的 union[①]，因为多个字段可起始于内存的相同偏移位置。下面是一个例子：

```
using System;
using System.Runtime.InteropServices;

// 开发人员显式排列这个值类型的字段
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValType {
    [FieldOffset(0)]
    private readonly Byte m_b;//m_b 和 m_x 字段在该类型的实例中相互重叠

    [FieldOffset(0)]
    private readonly Int16 m_x;//m_b 和 m_x 字段在该类型的实例中相互重叠
}
```

注意在类型中，一个引用类型和一个值类型相互重叠是不合法的。虽然允许多个引用类型在同一个起始偏移位置相互重叠，但这无法验证(unverifiable)。定义类型，在其中让多个值类型相互重叠则是合法的。但是，为了使这样的类型能够验证(verifiable)，所有重叠字节都必须能通过公共字段访问。

① union 是特殊的类，union 中的数据成员在内存中的存储相互重叠。每个数据成员都从相同内存地址开始。分配给 union 的存储区数量是包含它最大数据成员所需的内存数。同一时刻只有一个成员可以被赋值。——译注

5.3　值类型的装箱和拆箱

值类型比引用类型“轻”，原因是它们不作为对象在托管堆中分配，不被垃圾回收，也不通过指针进行引用。但许多时候都需要获取对值类型实例的引用。例如，假定要创建 **ArrayList (System.Collections** 命名空间中定义的一个类型)对象来容纳一组 **Point** 结构，代码如下：

```
// 声明值类型
struct Point {
   public Int32 x, y;
}

public sealed class Program {
   public static void Main() {
      ArrayList a = new ArrayList();
      Point p;                  // 分配一个 Point(不在堆中分配)
      for (Int32 i = 0; i < 10; i++) {
         p.x = p.y = i;         // 初始化值类型中的成员
         a.Add(p);              // 对值类型装箱，将引用添加到 Arraylist 中
      }
      ...
   }
}
```

每次循环迭代都初始化一个 **Point** 的值类型字段，并将该 **Point** 存储到 **ArrayList** 中。但思考一下 **ArrayList** 中究竟存储了什么？是 **Point** 结构，**Point** 结构的地址，还是其他完全不同的东西？要知道正确答案，必须研究 **ArrayList** 的 **Add** 方法，了解它的参数被定义成什么类型。本例的 **Add** 方法原型如下：

```
public virtual Int32 Add(Object value);
```

可以看出 **Add** 获取的是一个 **Object** 参数。也就是说，**Add** 获取对托管堆上的一个对象的引用(或指针)来作为参数。但之前的代码传递的是 **p**，也就是一个 **Point**，是值类型。为了使代码正确工作，**Point** 值类型必须转换成真正的、在堆中托管的对象，而且必须获取对该对象的引用。

将值类型转换成引用类型要使用**装箱**机制。下面总结了对值类型的实例进行装箱时所发生的事情。

1. 在托管堆中分配内存。分配的内存量是值类型各字段所需的内存量，还要加上托管堆所有对象都有的两个额外成员(类型对象指针和同步块索引)所需的内存量。
2. 值类型的字段复制到新分配的堆内存。
3. 返回对象地址。现在该地址是对象引用；值类型成了引用类型。

C#编译器自动生成对值类型实例进行装箱所需的 IL 代码，但仍需理解内部发生的事情，对代码长度和性能心中有数。

C#编译器自动生成对值类型实例进行装箱所需的 IL 代码，但你仍然需要理解内部的工作机制才能体会到代码的大小和性能问题。

C#编译器检测到上述代码是向要求引用类型的方法传递值类型，所以自动生成代码对对象进行装箱。所以在运行时，当前存在于 **Point** 值类型实例 **p** 中的字段复制到新分配的 **Point** 对象中。已装箱 **Point** 对象(现在是引用类型)的地址返回并传给 **Add** 方法。**Point** 对象一直存在于堆中，直至被垃圾回收。**Point** 值类型变量 **p** 可被重用，因为 **ArrayList** 不知道关于它的任何事情。在这种情况下，已装箱值类型的生存期超过了未装箱值类型的生存期。



注意 FCL 现在包含一组新的泛型集合类，非泛型集合类已成为“昨日黄花”。例如，应该使用 **System.Collections.Generic.List<T>** 类而不是 **System.Collections.ArrayList** 类。泛型集合类对非泛型集合类进行了大量改进。例如，API 得到简化和增强，集合类的性能也得到显著提升。但最大的改进就是泛型集合类允许开发人员在操作值类型的集合时不需要对集合中的项进行装箱/拆箱。单这一项改进，就使性能提升了不少。这是因为托管堆中需要创建的对象减少了，进而减少了应用程序需要执行的垃圾回收的次数。另外，开发人员还获得了编译时的类型安全性，源代码也因为强制类型转换的次数减少而变得更清晰。所有这一切都将在第 12 章“泛型”详细解释。

知道装箱如何进行后，接着谈谈拆箱。假定要用以下代码获取 **ArrayList** 的第一个元素：

```
Point p = (Point) a[0];
```

它获取 **ArrayList** 的元素 0 包含的引用(或指针)，试图将其放到 **Point** 值类型的实例 **p** 中。为此，已装箱 **Point** 对象中的所有字段都必须复制到值类型变量 **p** 中，后者在线程栈上。CLR 分两步完成复制。第一步获取已装箱 **Point** 对象中的各个 **Point** 字段的地址。这个过程称为**拆箱**(unboxing)。第二步将字段包含的值从堆复制到基于栈的值类型实例中。

拆箱不是直接将装箱过程倒过来。拆箱的代价比装箱低得多。拆箱其实就是获取指针的过程，该指针指向包含在一个对象中的原始值类型(数据字段)。其实，指针指向的是已装箱实例中的未装箱部分。所以和装箱不同，拆箱不要求在内存中复制任何字节。知道这个重要区别之后，还应知道的一个重点是，往往紧接着土拆箱发生一次字段复制。

装箱和拆箱/复制显然会对应用程序的速度和内存消耗产生不利影响，所以应留意编译器在什么时候生成代码来自动进行这些操作。并尝试手动编写代码，尽量减少这种情况的发生。

已装箱值类型实例在拆箱时，内部发生下面这些事情。

1. 如果包含“对已装箱值类型实例的引用”的变量为 **null**，抛出 **NullReferenceException** 异常。
2. 如果引用的对象不是所需值类型的已装箱实例，抛出 **InvalidCastException** 异常。[①]

① CLR 还允许将值类型拆箱为相同值类型的可空版本。详情将在第 19 章讨论。

第二条意味着以下代码的工作方式和你想的可能不一样：

```
public static void Main() {
   Int32 x = 5;
   Object o = x;          // 对 x 装箱，o 引用已装箱对象
   Int16 y = (Int16) o;   // 抛出 InvalidCastException 异常
}
```

从逻辑上说，完全能获取 **o** 引用的已装箱 **Int32**，将其强制转型为 **Int16**。但在对对象进行拆箱时，只能转型为最初未装箱的值类型——本例是 **Int32**。以下是上述代码的正确写法：

```
public static void Main() {
   Int32 x = 5;
   Object o = x;                  // 对 x 进行装箱，o 引用已装箱对象
   Int16 y = (Int16)(Int32) o; // 先拆箱为正确类型，再转型
}
```

前面说过，一次拆箱操作经常紧接着一次字段复制。以下 C#代码演示了拆箱和复制：

```
public static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;       // 对 p 装箱；o 引用已装箱实例

   p = (Point) o;      // 对 o 拆箱，将字段从已装箱实例复制到栈变量中
}
```

最后一行，C#编译器生成一条 IL 指令对 **o** 拆箱(获取已装箱实例中的字段的地址)，并生成另一条 IL 指令将这些字段从堆复制到基于栈的变量 **p** 中。

再来看看以下代码：

```
public static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;          // 对 p 装箱；o 引用已装箱实例

   // 将 Point 的 x 字段变成 2
   p = (Point) o;         // 对 o 拆箱，将字段从已装箱的实例复制到栈变量中
   p.x = 2;               // 更改栈变量的状态
   o = p;                 // 对 p 装箱；o 引用新的已装箱实例
}
```

最后三行代码唯一的目的就是将 **Point** 的 **x** 字段从 **1** 变成 **2**。为此，首先要执行一次拆箱，再执行一次字段复制，再更改字段(在栈上)，最后执行一次装箱(在托管堆上创建全新的已装箱实例)。想必你已体会到了装箱和拆箱/复制对应用程序性能的影响。

有的语言(比如 C++/CLI)允许在不复制字段的前提下对已装箱的值类型进行拆箱。拆箱返回已装箱对象中的未装箱部分的地址(忽略对象的“类型对象指针”和“同步块索引”这两个额外的成员)。接着可利用这个指针来操纵未装箱实例的字段(这些字段恰好在堆上的已装箱对象中)。例如，上述代码用 C++/CLI 来写，效率会高很多，因为可直接在已装箱 **Point** 实例中修改 **Point** 的 **x** 字段的值。这就避免了在堆上分配新对象和复制所有字段两次！



重要提示 如果关心应用程序的性能，就应清楚编译器何时生成代码执行这些操作。遗憾的是，许多编译器都隐式生成代码来装箱对象，所以有时并不知道自己的代码会造成装箱。如果关心特定算法的性能，可用 ILDasm.exe 这样的工具查看方法的 IL 代码，观察 IL 指令 box 都在哪些地方出现。

再来看几个装箱和拆箱的例子：

```
public static void Main() {
   Int32 v = 5;          // 创建未装箱值类型变量
   Object o = v;         // o 引用已装箱的、包含值 5 的 Int32
   v = 123;              // 将未装箱的值修改成 123

   Console.WriteLine(v + ", " + (Int32) o); // 显示"123, 5"
}
```

能从上述代码中看出发生了多少次装箱吗？如果说 3 次，会不会觉得意外？让我们仔细分析一下代码，理解具体发生的事情。为了帮助理解，下面列出为这个 **Main** 方法生成的 IL 代码。我为这些代码加上了注释，方便你看清楚发生的每个操作：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       45 (0x2d)
  .maxstack 3
  .locals init ([0]int32 v,
           [1] object o)
  // 将 5 加载到 v 中
  IL_0000: ldc.i4.5
  IL_0001: stloc.0

  // 对 v 装箱，将引用指针存储到 o 中
  IL_0002: ldloc.0
  IL_0003: box          [mscorlib]System.Int32
  IL_0008: stloc.1

  // 将 123 加载到 v 中
  IL_0009: ldc.i4.s 123
  IL_000b: stloc.0

  // 对 v 装箱，将指针保留在栈上以进行 Concat(连接)操作
  IL_000c: ldloc.0
  IL_000d: box          [mscorlib]System.Int32

  // 将字符串加载到栈上以执行 Concat 操作
  IL_0012: ldstr  ", "

  // 对 o 拆箱：获取一个指针，它指向栈上的 Int32 字段
  IL_0017: ldloc.1
  IL_0018: unbox.any [mscorlib]System.Int32

  // 对 Int32 装箱，将指针保留在栈上以进行 Concat 操作
  IL_001d: box          [mscorlib]System.Int32

  // 调用 Concat
  IL_0022: call string [mscorlib]System.String::Conct(object,
                                                      object,
                                                      object)
```

```
  // 将从 Concat 返回的字符串传给 WriteLine
  IL_0027: call void [mscorlib]System.Console::WriteLine(string)

  // 从 Main 返回，终止应用程序
  IL_002c: ret
} // end of method App::Main
```

首先在栈上创建一个 **Int32** 未装箱值类型实例(**v**)，将其初始化为 **5**。再创建 **Object** 类型的变量(**o**)并初始化，让它指向 **v**。但由于引用类型的变量始终指向堆中的对象，所以 C#生成正确的 IL 代码对 **v** 进行装箱，将 **v** 的已装箱拷贝的地址存储到 **o** 中。接着，值 **123** 被放到未装箱值类型实例 **v** 中，但这个操作不会影响已装箱的 **Int32**，后者的值依然为 **5**。

接着调用 **WriteLine** 方法，**WriteLine** 要求获取一个 **String** 对象，但当前没有 **String** 对象。相反，现在有三个数据项：一个未装箱的 **Int32** 值类型实例(**v**)，一个 **String**(它是引用类型)，以及对已装箱 **Int32** 值类型实例的引用(**o**)，它要转型为未装箱的 **Int32**。必须以某种方式合并这些数据项来创建一个 **String**。

为了创建一个 **String**，C#编译器生成代码来调用 **String** 的静态方法 **Concat**。该方法有几个重载版本，所有版本执行的操作都一样，只是参数的数量不同。由于需要连接三个数据项来创建字符串，所以编译器选择 **Concat** 方法的以下版本：

```
public static String Concat(Object arg0, Object arg1, Object arg2);
```

为第一个参数 **arg0** 传递的是 **v**。但 **v** 是未装箱的值参数，而 **arg0** 是 **Object**，所以必须对 **v** **进行**装箱，并将已装箱的 **v** 的地址传给 **arg0**。对于 **arg1** 参数，字符串", "作为 **String** 对象引用传递。对于 **arg2** 参数，**o**(一个 **Object** 引用)会转型为 **Int32**。这要求执行拆箱(但不紧接着执行复制)，从而获取包含在已装箱 **Int32** 中的未装箱 **Int32** 的地址。这个未装箱的 **Int32** 实例必须再次装箱，并将新的已装箱实例的内存地址传给 **Concat** 的 **arg2** 参数。

Concat 方法调用指定的每个对象的 **ToString** 方法，将每个对象的字符串形式连接起来。从 **Concat** 返回的 **String** 对象传给 **WriteLine** 方法以显示最终结果。

应该指出，如果像下面这样写 **WriteLine** 调用，生成的 IL 代码将具有更高的执行效率：

```
Console.WriteLine(v + ", " + o);  // 显示"123, 5"
```

这和前面的版本几乎完全一致，只是移除了变量 **o** 之前的**(Int32)**强制转型。之所以效率更高，是因为 **o** 已经是指向一个 **Object** 的引用类型，它的地址可直接传给 **Concat** 方法。所以，移除强制转型避免了两次操作：一次拆箱和一次装箱。不妨重新生成应用程序，观察 IL 代码来体会避免的额外操作：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       35 (0x23)
  .maxstack 3
  .locals init ([0] int32 v,
           [1] object o)


  // 将 5 加载到 v 中
  IL_0000: ldc.i4.5
```

```
  IL_0001: stloc.0

  // 对 v 装箱，并将引用指针存储到 o 中
  IL_0002: ldloc.0
  IL_0003: box        [mscorlib]System.Int32
  IL_0008: stloc.1

  // 将 123 加载到 v 中
  IL_0009: ldc.i4.s 123
  IL_000b: stloc.0

  // 对 v 装箱，并将指针保留在栈上以进行 Concat (连接) 操作
  IL_000c: ldloc.0
  IL_000d: box        [mscorlib]System.Int32

  // 将字符串加载到栈上以执行 Concat 操作
  IL_0012: ldstr  ", "

  // 将已装箱 Int32 的地址加载到栈上以进行 Concat 操作
  IL_0017: ldloc.1

  // 调用 Concat
  IL_0018: call string [mscorlib]System.String::Conct(object,
                                                      object,
                                                      object)

  // 将从 Concat 返回的字符串传给 WriteLine
  IL_001d: call void [mscorlib]System.Console::WriteLine(string)

  // 从 Main 返回，终止这个应用程序
  IL_0022: ret
} // end of method App::Main
```

简单对比一下两个版本的 **Main** 方法的 IL 代码，会发现没有**(Int32)**转型的版本比有转型的版本小了 10 字节。第一个版本额外的拆箱/装箱步骤显然会生成更多的代码。更大的问题是，额外的装箱步骤会从托管堆中分配一个额外的对象，将来必须对其进行垃圾回收。这两个版本的结果一样，速度上的差别也并不明显。但是，假如在循环中发生额外的、不必要的装箱操作，就会严重影响应用程序的性能和内存消耗。

甚至可以这样调用 **WriteLine**，进一步提升上述代码的性能：

```
Console.WriteLine(v.ToString() + ", " + o);      // 显示"123, 5"
```

这会为未装箱的值类型实例 **v** 调用 **ToString** 方法，它返回一个 **String**。**String** 对象已经是引用类型，所以能直接传给 **Concat** 方法，不需要任何装箱操作。

下面是演示装箱和拆箱的另一个例子：

```
public static void Main() {
  Int32 v = 5;                  // 创建未装箱的值类型变量
  Object o = v;                 // o 引用 v 的已装箱版本

  v = 123;                      // 将未装箱的值类型修改成 123
  Console.WriteLine(v);         // 显示"123"
  v = (Int32) o;                // 拆箱并将 o 复制到 v
  Console.WriteLine(v);         // 显示"5"
}
```

上述代码发生了多少次装箱？答案是一次。之所以只发生一次装箱，是因为 **System.Console**

类已定义了获取单个 **Int32** 参数的 **WriteLine** 方法：

```
public static void WriteLine(Int32 value);
```

在前面对 **WriteLine** 的两次调用中，变量 **v(Int32** 未装箱值类型实例)以传值方式传给方法。虽然 **WriteLine** 方法也许会在它自己内部对 **Int32** 装箱，但这已经不在我们的控制范围之内了。最重要的是，我们已尽可能地在*自己*的代码中减少了装箱。

仔细研究一下 FCL，会发现许多方法都针对不同的值类型参数进行了重载。例如，**System.Console** 类型提供了 **WriteLine** 方法的几个重载版本：

```
public static void WriteLine(Boolean);
public static void WriteLine(Char);
public static void WriteLine(Char[]);
public static void WriteLine(Int32);
public static void WriteLine(UInt32);
public static void WriteLine(Int64);
public static void WriteLine(UInt64);
public static void WriteLine(Single);
public static void WriteLine(Double);
public static void WriteLine(Decimal);
public static void WriteLine(Object);
public static void WriteLine(String);
```

以下几个方法也有一组类似的重载版本：**System.Console** 的 **Write** 方法，**System.IO.BinaryWriter** 的 **Write** 方法，**System.IO.TextWriter** 的 **Write** 和 **WriteLine** 方法，**System.Runtime.Serialization.SerializationInfo** 的 **AddValue** 方法，**System.Text.StringBuilder** 的 **Append** 和 **Insert** 方法。大多数方法进行重载唯一的目的就是减少常用值类型的装箱次数。

但这些 FCL 类的方法不可能接受你自己定义的值类型。另外，即使是 FCL 中定义好的值类型，这些方法也可能没有提供对应的重载版本。调用方法并传递值类型时，如果不存在与值类型对应的重载版本，那么调用的肯定是获取一个 **Object** 参数的重载版本。将值类型实例作为 **Object** 传递会造成装箱，从而对性能造成不利影响。定义自己的类时，可将类中的方法定义为泛型(通过类型约束将类型参数限制为值类型)。这样方法就可获取任何值类型而不必装箱。泛型主题将在第 12 章讨论。

关于装箱最后注意一点：如果知道自己的代码会造成编译器反复对一个值类型装箱，请改成用手动方式对值类型进行装箱。这样代码会变得更小、更快。下面是一个例子：

```
using System;

public sealed class Program {
  public static void Main() {
    Int32 v = 5; // 创建未装箱的值类型变量

#if INEFFICIENT
    // 编译下面这一行，v 被装箱 3 次，浪费时间和内存
    Console.WriteLine("{0}, {1}, {2}", v, v, v);
#else
    // 下面的代码结果一样，但无论执行速度，
    // 还是内存利用，都比前面的代码更胜一筹
    Object o = v;  // 对 v 进行手动装箱(仅 1 次)

    // 编译下面这一行不发生装箱
    Console.WriteLine("{0}, {1}, {2}", o, o, o);
```

```
#endif
   }
}
```

在定义了 **INEFFICIENT** 符号的前提下编译，编译器会生成代码对 **v** 装箱 3 次，造成在堆上分配 3 个对象！这太浪费了，因为每个对象都是完全相同的内容：**5**。在没有定义 **INEFFICIENT** 符号的前提下编译，**v** 只装箱一次，所以只在堆上分配一个对象。随后，在对 **Console.WriteLine** 方法的调用中，对同一个已装箱对象的引用被传递 3 次。第二个版本执行起来快得多，在堆上分配的内存也要少得多。

通过这些例子，很容易判断在什么时候一个值类型的实例需要装箱。简单地说，要获取对值类型实例的引用，实例就必须装箱。将值类型实例传给需要获取引用类型的方法，就会发生这种情况。但这并不是要对值类型实例装箱的唯一情况。

前面说过，未装箱值类型比引用类型更“轻”。这要归结于以下两个原因。

- 不在托管堆上分配。
- 没有堆上的每个对象都有的额外成员：“类型对象指针”和“同步块索引”。

由于未装箱值类型没有同步块索引，所以不能使用 **System.Threading.Monitor** 类型的方法(或者 C# **lock** 语句)让多个线程同步对实例的访问。

虽然未装箱值类型没有类型对象指针，但仍可调用由类型继承或重写的虚方法(比如 **Equals**，**GetHashCode** 或者 **ToString**)。如果值类型重写了其中任何虚方法，那么 CLR 可以非虚地调用该方法，因为值类型隐式密封，不可能有类型从它们派生，而且调用虚方法的值类型实例没有装箱。然而，如果重写的虚方法要调用方法在基类中的实现，那么在调用基类的实现时，值类型实例会装箱，以便能够通过 **this** 指针将对一个堆对象的引用传给基方法。

但在调用非虚的、继承的方法时(比如 **GetType** 或 **MemberwiseClone**)，无论如何都要对值类型进行装箱。因为这些方法由 **System.Object** 定义，要求 **this** 实参是指向堆对象的指针。

此外，将值类型的未装箱实例转型为类型的某个接口时要对实例进行装箱。这是因为接口变量必须包含对堆对象的引用(接口主题将在第 13 章“接口”中讨论)。以下代码对此进行了演示：

```
using System;

internal struct Point : IComparable {
   private Int32 m_x, m_y;

   // 构造器负责初始化字段
   public Point(Int32 x, Int32 y)
   {
      m_x = x;
      m_y = y;
   }

   // 重写从 System.ValueType 继承的 ToString 方法
```

```
    public override String ToString()
    {
        // 将 point 作为字符串返回。注意：调用 ToString 以避免装箱
        return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
    }

    // 实现类型安全的 CompareTo 方法
    public Int32 CompareTo(Point other)
    {
        // 利用勾股定理计算哪个 point 距离原点(0, 0)更远
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
          - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }

    // 实现 IComparable 的 CompareTo 方法
    public Int32 CompareTo(Object o) {
        if (GetType() != o.GetType()) {
            throw new ArgumentException("o is not a Point");
        }
        // 调用类型安全的 CompareTo 方法
        return CompareTo((Point) o);
    }
}

public static class Program
{
    public static void Main()
    {
        // 在栈上创建两个 Point 实例
        Point p1 = new Point(10, 10);
        Point p2 = new Point(20, 20);

        // 调用 ToString(虚方法)不装箱 p1
        Console.WriteLine(p1.ToString());     // 显示"(10, 10)"

        // 调用 GetType(非虚方法)时，要对 p1 进行装箱
        Console.WriteLine(p1.GetType()); // 显示"Point"

        // 调用 CompareTo 不装箱 p1
        // 由于调用的是 CompareTo(Point)，所以 p2 不装箱
        Console.WriteLine(p1.CompareTo(p2)); // 显示"-1"

        // p1 要装箱，引用放到 c 中
        IComparable c = p1;
        Console.WriteLine(c.GetType());   // 显示"Point"

        // 调用 CompareTo 不装箱 p1
        // 由于向 CompareTo 传递的不是 Point 变量，
        // 所以调用的是 CompareTo(Object)，它要求获取对已装箱 Point 的引用
        // c 不装箱是因为它本来就引用已装箱 Point
        Console.WriteLine(p1.CompareTo(c));   // 显示"0"

        // c 不装箱，因为它本来就引用已装箱 Point
        // p2 要装箱，因为调用的是 CompareTo(Object)
        Console.WriteLine(c.CompareTo(p2));   // 显示"-1"

        // 对 c 拆箱，字段复制到 p2 中
        p2 = (Point) c;

        // 证明字段已复制到 p2 中
```

```
        Console.WriteLine(p2.ToString());     // 显示"(10, 10)"
    }
}
```

上述代码演示了涉及装箱和拆箱的几种情形。

1. 调用 ToString

   调用 **ToString** 时 **p1** 不必装箱。表面看 **p1** 似乎必须装箱，因为 **ToString** 是从基类 **System.ValueType** 继承的虚方法。通常，为了调用虚方法，CLR 需要判断对象的类型来定位类型的方法表。由于 **p1** 是未装箱的值类型，所以不存在“类型对象指针”。但 JIT 编译器发现 **Point** 重写了 **ToString** 方法，所以会生成代码来直接(非虚地)调用 **ToString** 方法，而不必进行任何装箱操作。编译器知道这里不存在多态性问题，因为 **Point** 是值类型，没有类型能从它派生以提供虚方法的另一个实现。但假如 **Point** 的 **ToString** 方法在内部调用 **base.ToString()**，那么在调用 **System.ValueType** 的 **ToString** 方法时，值类型的实例会被装箱。

2. **调用 GetType**

   调用非虚方法 **GetType** 时 **p1** 必须装箱。**Point** 的 **GetType** 方法是从 **System.Object** 继承的。所以，为了调用 **GetType**，CLR 必须使用指向类型对象的指针，而这个指针只能通过装箱 **p1** 来获得。

3. 调用 CompareTo(第一次)

   第一次调用 **CompareTo** 时 **p1** 不必装箱，因为 **Point** 实现了 **CompareTo** 方法，编译器能直接调用它。注意向 **CompareTo** 传递的是一个 **Point** 变量(**p2**)，所以编译器调用的是获取一个 **Point** 参数的 **CompareTo** 重载版本。这意味着 **p2** 以传值方式传给 **CompareTo**，无需装箱。

4. 转型为 IComparable

   p1 转型为接口类型的变量 **c** 时必须装箱，因为接口被定义为引用类型。装箱 **p1** 后，指向已装箱对象的指针存储到变量 **c** 中。后面对 **GetType** 的调用证明 **c** 确实引用堆上的已装箱 **Point**。

5. 调用 CompareTo(第二次)

   第二次调用 **CompareTo** 时 **p1** 不必装箱，因为 **Point** 实现了 **CompareTo** 方法，编译器能直接调用。注意向 **CompareTo** 传递的是 **IComparable** 类型的变量 **c**，所以编译器调用的是获取一个 **Object** 参数的 **CompareTo** 重载版本。这意味着传递的实参必须是指针，必须引用堆上一个对象。幸好，**c** 确实引用一个已装箱 **Point**，所以 **c** 中的内存地址直接传给 **CompareTo**，无需额外装箱。

6. 调用 CompareTo(第三次)

   第三次调用 **CompareTo** 时，**c** 本来就引用堆上的已装箱 **Point** 对象，所以不装箱。由于 **c** 是 **IComparable** 接口类型，所以只能调用接口的获取一个 **Object** 参数的 **CompareTo** 方法。这意味着传递的实参必须是引用了堆上对象的指针。所以 **p2** 要装

箱，指向这个已装箱对象的指针将传给 **CompareTo**。

7. **转型为 Point**

将 **c** 转型为 **Point** 时，**c** 引用的堆上对象被拆箱，其字段从堆复制到 **p2**。**p2** 是栈上的 **Point** 类型实例。

我知道，对于引用类型、值类型和装箱的所有这些讨论很容易让人产生挫折感。但是，任何.NET Framework 开发人员只有在切实理解了这些概念之后，才能保障自己的长期成功。相信我，只有深刻理解了之后，才能更快、更轻松地构建高效率的应用程序。

5.3.1　使用接口更改已装箱值类型中的字段(以及为什么不应该这样做)

下面通过一些例子来验证自己对值类型、装箱和拆箱的理解程度。请研究以下代码，判断它会在控制台上显示什么：

```
using System;

// Point 是值类型
internal struct Point {
  private Int32 m_x, m_y;

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

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

  public override String ToString() {
    return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
  }
}

public sealed class Program {
  public static void Main() {
    Point p = new Point(1, 1);

    Console.WriteLine(p);

    p.Change(2, 2);
    Console.WriteLine(p);

    Object o = p;
    Console.WriteLine(o);

    ((Point) o).Change(3, 3);
    Console.WriteLine(o);
  }
}
```

程序其实很简单。**Main** 在栈上创建 **Point** 值类型的实例(**p**)，将它的 **m_x** 和 **m_y** 字段设为 **1**。然后，第一次调用 **WriteLine** 之前 **p** 要装箱。**WriteLine** 在已装箱 **Point** 上调用 **ToString**，并像预期的那样显示**(1, 1)**。然后用 **p** 调用 **Change** 方法，该方法将 **p** 在栈上的 **m_x** 和 **m_y** 字段值都更改为 **2**。第二次调用 **WriteLine** 时，再次对 **p** 进行装箱，像预料之中的那样显示**(2, 2)**。

现在，**p** 进行第 3 次装箱，**o** 引用已装箱的 **Point** 对象。第 3 次调用 **WriteLine** 再次显示**(2, 2)**，这同样是预料之中的。最后，我们希望调用 **Change** 方法来更新已装箱的 **Point** 对象中的字段。然而，**Object**(变量 **o** 的类型)对 **Change** 方法一无所知，所以首先必须将 **o** 转型为 **Point**。将 **o** 转型为 **Point** 要求对 **o** 进行拆箱，并将已装箱 **Point** 中的字段复制到线程栈上的一个临时 **Point** 中！这个临时 **Point** 的 **m_x** 和 **m_y** 字段会变成 **3** 和 **3**，但已装箱的 **Point** 不受这个 **Change** 调用的影响。第四次调用 **WriteLine** 方法，会再次显示**(2, 2)**。这是许多开发人员预料不到的。

有的语言(比如 C++/CLI)允许更改已装箱值类型中的字段，但 C#不允许。不过，可以用接口欺骗 C#，让它允许这个操作。下面是上例的修改版本：

```
using System;

// 接口定义了 Change 方法
internal interface IChangeBoxedPoint {
   void Change(Int32 x, Int32 y);
}

// Point 是值类型
internal struct Point : IChangeBoxedPoint {
   private Int32 m_x, m_y;

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

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

  public override String ToString() {
    return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
  }
}

public sealed class Program {
  public static void Main() {
    Point p = new Point(1, 1);

    Console.WriteLine(p);

    p.Change(2, 2);
    Console.WriteLine(p);

    Object o = p;
    Console.WriteLine(o);

    ((Point) o).Change(3, 3);
    Console.WriteLine(o);

    // 对 p 进行装箱，更改已装箱的对象，然后丢弃它
    ((IChangeBoxedPoint) p).Change(4, 4);
    Console.WriteLine(p);

    // 更改已装箱的对象，并显示它
    ((IChangeBoxedPoint) o).Change(5, 5);
    Console.WriteLine(o);
  }
}
```

上述代码和上一个版本几乎完全一致，主要区别是 **Change** 方法由 **IChangeBoxedPoint** 接口定义，**Point** 类型现在实现了该接口。**Main** 中的前 4 个 **WriteLine** 调用和前面的例子相同，生成的结果也一样 (这是我们预期的)。然而，**Main** 最后新增了两个例子。

在第一个例子中，未装箱的 **Point p** 转型为一个 **IChangeBoxedPoint**。这个转型造成对 **p** 中的值进行装箱。然后在已装箱值上调用 **Change**，这确实会将其 **m_x** 和 **m_y** 字段分别变成 **4** 和 **4**。但在 **Change** 返回之后，已装箱对象立即准备好进行垃圾回收。所以，对 **WriteLine** 的第 5 个调用会显示**(2, 2)**。许多开发人员预期的并不是这个结果。

在最后一个例子中，**o** 引用的已装箱 **Point** 转型为一个 **IChangeBoxedPoint**。这不需要装箱，因为 **o** 本来就是已装箱的 **Point**。然后调用 **Change**，它能正确修改已装箱 **Point** 的 **m_x** 和 **m_y** 字段。接口方法 **Change** 使我能够更改已装箱 **Point** 对象中的字段！现在调用 **WriteLine**，会像预期的那样显示**(5, 5)**。本例旨在演示接口方法如何修改已装箱值类型中的字段。在 C# 中，不用接口方法便无法做到。



重要提示　本章前面提到，值类型应该“不可变”（immutable）。也就是说，我们不应该定义任何会修改实例字段的成员。事实上，我建议将值类型的字段都标记为 **readonly**。这样，一旦不留神写一个试图更改字段的方法，编译时就会报错。前面的例子清楚揭示了我们为什么应该这样做。假如方法试图修改值类型的实例字段，调用这个方法就会产生非预期的行为。构造好值类型后，如果不调用任何会修改其状态的方法(或者如果根本不存在这样的方法)，就用不着操心什么时候发生装箱和拆箱/字段复制。如果值类型不可变，简单复制相同的状态就可以了(不用担心有方法会修改这些状态)，代码的任何行为都在你的掌控之中。

有许多开发人员审阅了本书内容。在阅读我的部分示例代码之后(比如前面的代码)，他们告诉我以后再也不敢使用值类型了。我必须声明，值类型的这些玄妙之处着实花了我好几天功夫进行调试，痛定思痛之余，我必须在之里着重强调，提醒大家注意，希望大家记住我描述的问题。这样，当代码真正出现这些问题的时候，我们就能够做到心中有数。虽然如此，但也不要因噎废食而惧怕值类型。它们很有用，有自己的适用场景。毕竟，程序偶尔还是需要 **Int32** 的。只是要注意，值类型和引用类型的行为会因为使用方式的不同而有明显差异。事实上，前例将 **Point** 声明为 **class** 而不是 **struct**，即可获得令人满意的结果。最后还要告诉你一个好消息，FCL 的核心值类型(**Byte**，**Int32**，**UInt32**，**Int64**，**UInt64**，**Single**，**Double**，**Decimal**，**BigInteger**，**Complex** 以及所有枚举)都是“不可变”的，所以在使用这些类型时，不会发生任何稀奇古怪的事情。

5.3.2　对象相等性和同一性

开发人员经常写代码比较对象。例如，有时要将对象放到集合，写代码对集合中的对象排

序、搜索或比较。本节将讨论相等性和同一性，还将讨论如何定义正确实现了对象相等性的类型。

System.Object 类型提供了名为 **Equals** 的虚方法，作用是在两个对象包含相同值的前提下返回 **true**。**Object** 的 **Equals** 方法是像下面这样实现的：

```
public class Object {
  public virtual Boolean Equals(Object obj) {

    // 如果两个引用指向同一个对象，它们肯定包含相同的值
    if (this == obj) return true;

    // 假定对象不包含相同的值
    return false;
  }
}
```

乍一看，这似乎就是 **Equals** 的合理实现：假如 **this** 和 **obj** 实参引用同一个对象，就返回 **true**。似乎合理是因为 **Equals** 知道对象肯定包含和它自身一样的值。但假如实参引用不同对象，**Equals** 就不肯定对象是否包含相同的值，所以返回 **false**。换言之，对于 **Object** 的 **Equals** 方法的默认实现，它实现的实际是**同一性**(identity)，而非**相等性**(equality)。

遗憾的是，**Object** 的 **Equals** 方法的默认实现并不合理，而且永远都不应该像这样实现。研究一下类的继承层次结构，并思考如何正确重写 **Equals** 方法，马上会发现问题出在哪里。下面展示了 **Equals** 方法应该如何正确地实现。

1. 如果 **obj** 实参为 **null**，就返回 **false**，因为调用非静态 **Equals** 方法时，**this** 所标识的当前对象显然不为 **null**。

2. 如果 **this** 和 **obj** 实参引用同一个对象，就返回 **true**。在比较包含大量字段的对象时，这一步有助于提升性能。

3. 如果 **this** 和 **obj** 实参引用不同类型的对象，就返回 **false**。一个 **String** 对象显然不等于一个 **FileStream** 对象。

4. 针对类型定义的每个实例字段，将 **this** 对象中的值与 **obj** 对象中的值进行比较。任何字段不相等，就返回 **false**。

5. 调用基类的 **Equals** 方法来比较它定义的任何字段。如果基类的 **Equals** 方法返回 **false**，就返回 **false**；否则返回 **true**。

所以，Microsoft 本应像下面这样实现 **Object** 的 **Equals** 方法：

```
public class Object {
  public virtual Boolean Equals(Object obj) {
    // 要比较的对象不能为 null
    if (obj == null) return false;

    // 如果对象属于不同的类型，则肯定不相等
    if (this.GetType() != obj.GetType()) return false;

    // 如果对象属于相同的类型，那么在它们的所有字段都匹配的前提下返回 true
    // 由于 System.Object 没有定义任何字段，所以字段是匹配的
```

```
    return true;
  }
}
```

但由于 Microsoft 没有像这样实现 **Equals**，所以 **Equals** 的实现规则远比想像的复杂。类型重写 **Equals** 方法时应调用其基类的 **Equals** 实现(除非基类就是 **Object**)。另外，由于类型能重写 **Object** 的 **Equals** 方法，所以不能再用它测试同一性。为了解决这个问题，**Object** 提供了静态方法 **ReferenceEquals**，其原型如下：

```
public class Object {
  public static Boolean ReferenceEquals(Object objA, Object objB) {
     return (objA == objB);
  }
}
```

检查同一性(看两个引用是否指向同一个对象)务必调用 **ReferenceEquals**，不应使用 C#的==操作符(除非先把两个操作数都转型为 **Object**)，因为某个操作数的类型可能重载了==操作符，为其赋予不同于“同一性”的语义。

可以看出，在涉及对象相等性和同一性的时候，.NET Framework 的设计很容易使人混淆。顺便说一下，**System.ValueType**(所有值类型的基类)就重写了 **Object** 的 **Equals** 方法，并进行了正确的实现来执行值的相等性检查(而不是同一性检查)。**ValueType** 的 **Equals** 内部是这样实现。

1. 如果 **obj** 实参为 **null**，就返回 **false**。
2. 如果 **this** 和 **obj** 实参引用不同类型的对象，就返回 **false**。
3. 针对类型定义的每个实例字段，都将 **this** 对象中的值与 **obj** 对象中的值进行比较(通过调用字段的 **Equals** 方法)。任何字段不相等，就返回 **false**。
4. 返回 **true**。**ValueType** 的 **Equals** 方法不调用 **Object** 的 **Equals** 方法。

在内部，**ValueType** 的 **Equals** 方法利用反射(详情将在第 23 章“程序集加载和反射”讲述)完成上述步骤 3。由于 CLR 反射机制慢，定义自己的值类型时应重写 **Equals** 方法来提供自己的实现，从而提高用自己类型的实例进行值相等性比较的性能。当然，自己的实现不调用 **base.Equals**。

定义自己的类型时，你重写的 **Equals** 要符合相等性的 4 个特征。

- **Equals** 必须自反；**x.Equals(x)**肯定返回 **true**。
- **Equals** 必须对称；**x.Equals(y)**和 **y.Equals(x)**返回相同的值。
- **Equals** 必须可传递；**x.Equals(y)**返回 **true**，**y.Equals(z)**返回 **true**，则 **x.Equals(z)**肯定返回 **true**。
- **Equals** 必须一致。比较的两个值不变，**Equals** 返回值(**true** 或 **false**)也不能变。

如果实现的 **Equals** 不符合上述任何特征，应用程序就会行为失常。重写 **Equals** 方法时，可能还需要做下面几件事情。

- **让类型实现 System.IEquatable<T>接口的 Equals 方法**
  这个泛型接口允许定义类型安全的 **Equals** 方法。通常，你实现的 **Equals** 方法应获取一个 **Object** 参数，以便在内部调用类型安全的 **Equals** 方法。

- **重载==和!=操作符方法**
  通常应实现这些操作符方法，在内部调用类型安全的 **Equals**。

此外，如果以后要出于排序目的而比较类型的实例，类型还应实现 **System.IComparable** 的 **CompareTo** 方法和 **System.IComparable<T>**的类型安全的 **CompareTo** 方法。如果实现了这些方法，还可考虑重载各种比较操作符方法(<，<=，>，>=)，在这些方法内部调用类型安全的 **CompareTo** 方法。

5.4 对象哈希码

FCL 的设计者认为，如果能将任何对象的任何实例放到哈希表集合中，能带来很多好处。为此，**System.Object** 提供了虚方法 **GetHashCode**，它能获取任意对象的 **Int32** 哈希码。

如果你定义的类型重写了 **Equals** 方法，还应重写 **GetHashCode** 方法。事实上，如果类型重写 **Equals** 的同时没有重写 **GetHashCode**，Microsoft C#编译器会生成一条警告。例如，编译以下类型会显示警告消息：**warning CS0659: "Program"重写 Object.Equals(object o) 但不重写 Object.GetHashCode()**。

```
public sealed class Program {
  public override Boolean Equals(Object obj) { ... }
}
```

类型定义 **Equals** 之所以还要定义 **GetHashCode**，是由于在 **System.Collections.Hashtable** 类型、**System.Collections.Generic.Dictionary** 类型以及其他一些集合的实现中，要求两个对象必须具有相同哈希码才被视为相等。所以，重写 **Equals** 就必须重写 **GetHashCode**，确保相等性算法和对象哈希码算法一致。

简单地说，向集合添加键/值(key/value)对，首先要获取键对象的哈希码。该哈希码指出键/值对要存储到哪个哈希桶(bucket)中。集合需要查找键时，会获取指定键对象的哈希码。该哈希码标识了现在要以顺序方式搜索的哈希桶，将在其中查找与指定键对象相等的键对象。采用这个算法来存储和查找键，意味着一旦修改了集合中的一个键对象，集合就再也找不到该对象。所以，需要修改哈希表中的键对象时，正确做法是移除原来的键/值对，修改键对象，再将新的键/值对添加回哈希表。

自定义 **GetHashCode** 方法或许不是一件难事。但取决于数据类型和数据分布情况，可能并不容易设计出能返回良好分布值的哈希算法。下面是一个简单的哈希算法，它用于 **Point**

对象时也许还不错：

```
internal sealed class Point {
  private readonly Int32 m_x, m_y;
  public override Int32 GetHashCode() {
    return m_x ^ m_y; // 返回 m_x 和 m_y 的 XOR 结果
  }
  …
}
```

选择算法来计算类型实例的哈希码时，请遵守以下规则。

- 这个算法要提供良好的随机分布，使哈希表获得最佳性能。
- 可在算法中调用基类的 **GetHashCode** 方法，并包含它的返回值。但一般不要调用 **Object** 或 **ValueType** 的 **GetHashCode** 方法，因为两者的实现都与高性能哈希算法“不沾边”。
- 算法至少使用一个实例字段。
- 理想情况下，算法使用的字段应该不可变(immutable)；也就是说，字段应在对象构造时初始化，在对象生存期“永不言变”。
- 算法执行速度尽量快。
- 包含相同值的不同对象应返回相同哈希码。例如，包含相同文本的两个 **String** 对象应返回相同哈希码。

System.Object 实现的 **GetHashCode** 方法对派生类型和其中的字段一无所知，所以返回一个在对象生存期保证不变的编号。



重要提示　假如因为某些原因要实现自己的哈希表集合，或者要在实现的代码中调用 **GetHashCode**，记住千万不要对哈希码进行持久化，因为哈希码很容易改变。例如，一个类型未来的版本可能使用不同的算法计算对象哈希码。

有个公司没有把这个警告放在心上。在他们的网站上，用户可选择用户名和密码来创建账号。然后，网站获取密码 **String**，调用 **GetHashCode**，将哈希码持久性存储到数据库。用户重新登录网站，输入自己的密码。网站再次调用 **GetHashCode**，并将哈希码与数据库中存储的值比较，匹配就允许访问。不幸的是，公司升级到新版本 CLR 后，**String** 的 **GetHashCode** 方法发生了改变，现在返回不同的哈希码。结果是所有用户都无法登录！

5.5　dynamic 基元类型

C#是类型安全的编程语言。意味着所有表达式都解析成类型的实例，编译器生成的代码只执行对该类型有效的操作。和非类型安全的语言相比，类型安全的语言的优势在于：程序

员会犯的许多错误都能在编译时检测到，确保代码在尝试执行前是正确的。此外，能编译出更小、更快的代码，因为能在编译时进行更多预设，并在生成的 IL 和元数据中落实预设。

但程序许多时候仍需处理一些运行时才会知晓的信息。虽然可用类型安全的语言(比如 C#)和这些信息交互，但语法就会比较笨拙，尤其是在涉及大量字符串处理的时候。另外，性能也会有所损失。如果写的是纯 C#应用程序，只有在使用反射(详情参见第 23 章“程序集加载和反射”)的时候，才需要和运行时才能确定的信息打交道。但许多开发者在使用 C#时，都要和一些不是用 C#实现的组件进行通信。有的组件是.NET 动态语言，比如 Python 或 Ruby，有的是支持 **IDispatch** 接口的 COM 对象(可能用原生 C 或 C++实现)，也有的是 HTML 文档对象模型(Document Object Model，DOM)对象(可以用多种语言和技术实现)。构建 Microsoft Silverlight 应用程序时，与 HTML DOM 对象的通信尤其重要。

为了方便开发人员使用反射或者与其他组件通信，C#编译器允许将表达式的类型标记为 **dynamic**。还可将表达式的结果放到变量中，并将变量类型标记为 **dynamic**。然后，可以用这个 **dynamic** 表达式/变量调用成员，比如字段、属性/索引器、方法、委托以及一元/二元/转换操作符。代码使用 **dynamic** 表达式/变量调用成员时，编译器生成特殊 IL 代码来描述所需的操作。这种特殊的代码称为 payload(有效载荷)。在运行时，payload 代码根据 **dynamic** 表达式/变量引用的对象的实际类型来决定具体执行的操作。

以下代码进行了演示。

```
internal static class DynamicDemo {
   public static void Main() {
      dynamic value;
      for (Int32 demo = 0; demo < 2; demo++) {
         value = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
         value = value + value;
         M(value);
      }
   }

   private static void M(Int32 n) { Console.WriteLine("M(Int32): " + n); }
   private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

执行 **Main** 会得到以下输出：

```
M(Int32): 10
M(String): AA
```

要理解发生的事情，首先就得搞清楚+操作符。它的两个操作数的类型是 **dynamic**。由于 **value** 是 **dynamic**，所以 C#编译器生成 payload 代码在运行时检查 **value** 的实际类型，决定+操作符实际要做什么。

第一次对+操作符求值，**value** 包含 **5**(一个 **Int32**)，所以结果是 **10**(也是 **Int32**)。结果存回 **value** 变量。然后调用 **M** 方法，将 **value** 传给它。编译器针对 **M** 调用生成 payload 代码，以便在运行时检查传给 **M** 的实参的实际类型，并决定应该调用 **M** 方法的哪个重载版本。由于 **value** 包含一个 **Int32**，所以调用获取 **Int32** 参数的版本。

第二次对+操作符求值，**value** 包含"A"(一个 **String**)，所以结果是"AA"("A"和它自己连接)。然后再次调用 **M** 方法，将 **value** 传给它。这次 payload 代码判断传给 **M** 的是一个 **String**，所以调用获取 **String** 参数的版本。

如果字段、方法参数或方法返回值的类型是 **dynamic**，编译器会将该类型转换为 **System.Object**，并在元数据中向字段、参数或返回类型应用 **System.Runtime.CompilerServices.DynamicAttribute** 的实例。如果局部变量被指定为 **dynamic**，则变量类型也会成为 **Object**，但不会向局部变量应用 **DynamicAttribute**，因为它限制在方法内部使用。由于 **dynamic** 其实就是 **Object**，所以方法签名不能仅靠 **dynamic** 和 **Object** 的变化来区分。

泛型类(引用类型)、结构(值类型)、接口、委托或方法的泛型类型实参也可以是 **dynamic** 类型。编译器将 **dynamic** 转换成 **Object**，并向必要的各种元数据应用 **DynamicAttribute**。注意，使用的泛型代码是已经编译好的，会将类型视为 **Object**；编译器不在泛型代码中生成 payload 代码，所以不会执行动态调度。

所有表达式都能隐式转型为 **dynamic**，因为所有表达式最终都生成从 **Object** 派生的类型[①]。正常情况下，编译器不允许写代码将表达式从 **Object** 隐式转型为其他类型；必须显式转型。但是，编译器允许使用隐式转型语法将表达式从 **dynamic** 转型为其他类型：

```
Object o1 = 123;            // OK: 从 Int32 隐式转型为 Object(装箱)
Int32 n1 = o1;              // Error: 不允许从 Object 到 Int32 的隐式转型
Int32 n2 = (Int32) o1;      // OK: 从 Object 显式转型为 Int32(拆箱)

dynamic d1 = 123;           // OK: 从 Int32 隐式转型为 dynamic(装箱)
Int32 n3 = d1;              // OK: 从 dynamic 隐式转型为 Int32(拆箱)
```

从 **dynamic** 转型为其他类型时，虽然编译器允许省略显式转型，但 CLR 会在运行时验证转型来确保类型安全性。如果对象类型不兼容要转换成的类型，CLR 会抛出 **InvalidCastException** 异常。

注意，**dynamic** 表达式的求值结果是一个动态表达式。例如以下代码：

```
dynamic d = 123;
var result = M(d);          // 注意: 'var result'等同于'dynamic result'
```

代码之所以能通过编译，是因为编译时不知道调用哪个 **M** 方法，从而不知道 **M** 的返回类型，所以编译器假定 **result** 变量具有 **dynamic** 类型。为了对此进行验证，可以在 Visual Studio 中将鼠标放在 **var** 上；“智能感知”窗口会显示“**dynamic: 表示对象的操作将在运行时解析**”。如果运行时调用的 **M** 方法的返回类型是 **void**，将抛出 **Microsoft.CSharp.RuntimeBinder.RuntimeBinderException** 异常。

① 值类型当然要装箱。



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 关键字只能在方法内部声明局部变量，而 **dynamic** 关键字可用于局部变量、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 **var** 关键字的详情，请参见 9.2 节“隐式类型的局部变量”。

然而，从 **dynamic** 转换成另一个静态类型时，结果类型当然是静态类型。类似地，向类型的构造器传递一个或多个 **dynamic** 实参，结果是要构造的对象的类型：

```
dynamic d = 123;
var x = (Int32) d;              // 转换: 'var x'等同于'Int32 x'
var dt = new DateTime(d);       // 构造: 'var dt'等同于'DateTime dt'
```

如果 **dynamic** 表达式被指定为 **foreach** 语句中的集合，或者被指定为 **using** 语句中的资源，编译器会生成代码，分别将表达式转型为非泛型 **System.IEnumerable** 接口或 **System.IDisposable** 接口。转型成功，就使用表达式，代码正常运行。转型失败，就抛出 **Microsoft.CSharp.RuntimeBinder.RuntimeBinderException** 异常。



重要提示 **dynamic** 表达式其实是和 **System.Object** 一样的类型。编译器假定你在表达式上进行的任何操作都是合法的，所以不会生成任何警告或错误。但如果试图在运行时执行无效的操作，就会抛出异常。此外，Visual Studio 无法提供任何“智能感知”支持来帮助你写针对 **dynamic** 表达式的代码。虽然能定义对 **Object** 进行扩展的扩展方法(详情参见第 8 章“方法”)，但不能定义对 **dynamic** 进行扩展的扩展方法。另外，不能将 lambda 表达式或匿名方法(都在第 17 章“委托”中讨论)作为实参传给 **dynamic** 方法调用，因为编译器推断不了要使用的类型。

以下示例 C#代码使用 COM **IDispatch** 创建 Microsoft Office Excel 工作簿，将一个字符串放到单元格 A1 中：

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
   Application excel = new Application();
   excel.Visible = true;
   excel.Workbooks.Add(Type.Missing);
   ((Range)excel.Cells[1, 1]).Value
         = "Text in cell A1"; // 把这个字符串放到单元格 A1 中
}
```

没有 **dynamic** 类型，**excel.Cells[1, 1]**的返回值就是 **Object** 类型，必须先转型为 **Range** 类型才能访问其 **Value** 属性。但在为 COM 对象生成可由“运行时”调用的包装(wrapper)程序集时，COM 方法中使用的任何 **VARIANT** 实际都转换成 **dynamic**；这称为**动态化**(dynamification)。所以，由于 **excel.Cells[1, 1]**是 **dynamic** 类型，所以不必显式转型为 **Range**

类型就能访问其 **Value** 属性。动态化显著简化了与 COM 对象的互操作。下面是简化后的代码：

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    excel.Cells[1, 1].Value
          = "Text in cell A1"; // 把这个字符串放到单元格 A1 中
}
```

以下代码展示了如何利用反射在 **String** 目标("Jeffrey Richter")上调用方法("Contains")，向它传递一个 **String** 实参("ff")，并将 **Boolean** 结果存储到局部变量 **result** 中：

```
Object target = "Jeffrey Richter";
Object arg = "ff";

// 在目标上查找和希望的实参类型匹配的方法
Type[] argTypes = new Type[] { arg.GetType() };
MethodInfo method = target.GetType().GetMethod("Contains", argTypes);

// 在目标上调用方法，传递希望的实参
Object[] arguments = new Object[] { arg };
Boolean result = Convert.ToBoolean(method.Invoke(target, arguments));
```

可利用 C#的 **dynamic** 类型重写上述代码，并使用获得了显著简化的语法：

```
dynamic target = "Jeffrey Richter";
dynamic arg = "ff";
Boolean result = target.Contains(arg);
```

我早先指出 C#编译器会生成 payload 代码，在运行时根据对象实际类型判断要执行什么操作。这些 payload 代码使用了称为**运行时绑定器**(runtime binder)的类。不同编程语言定义了不同的运行时绑定器来封装自己的规则。C#“运行时绑定器”的代码在 Microsoft.CSharp.dll 程序集中，生成使用 **dynamic** 关键字的项目必须引用该程序集。编译器的默认响应文件 CSC.rsp 中已引用了该程序集。记住是这个程序集中的代码知道在运行时生成代码，在+操作符应用于两个 **Int32** 对象时执行加法，在+操作符应用于两个 **String** 对象时执行连接。

在运行时，Microsoft.CSharp.dll 程序集必须加载到 AppDomain 中，这会损害应用程序的性能，增大内存消耗。Microsoft.CSharp.dll 还会加载 System.dll 和 System.Core.dll。如果使用 **dynamic** 与 COM 组件互操作，还会加载 System.Dynamic.dll。payload 代码执行时，会在运行时生成动态代码；这些代码进入驻留于内存的程序集，即“匿名寄宿的 DynamicMethods 程序集”(Anonymously Hosted DynamicMethods Assembly)，作用是当特定 call site[①]使用具有相同运行时类型的动态实参发出大量调用时增强动态调度性能。

C#内建的动态求值功能所产生的额外开销不容忽视。虽然能用动态功能简化语法，但也要看是否值得。毕竟，加载所有这些程序集以及额外的内存消耗，会对性能造成额外影响。如果程序中只是一、两个地方需要动态行为，传统做法或许更高效。即调用反射方法(如果

① call site 是发出调用的地方，可理解成调用了一个目标方法的表达式或代码行。——译注

是托管对象)，或者进行手动类型转换(如果是 COM 对象)。

在运行时，C#的“运行时绑定器”根据对象的运行时类型分析应采取什么动态操作。绑定器首先检查类型是否实现了 **IDynamicMetaObjectProvider** 接口。如果是，就调用接口的 **GetMetaObject** 方法，它返回 **DynamicMetaObject** 的一个派生类型。该类型能处理对象的所有成员、方法和操作符绑定。**IDynamicMetaObjectProvider** 接口和 **DynamicMetaObject** 基类都在 **System.Dynamic** 命名空间中定义，都在 System.Core.dll 程序集中。

像 Python 和 Ruby 这样的动态语言，是为它们的类型赋予了从 **DynamicMetaObject** 派生的类型，以便从其他编程语言(比如 C#)中以恰当方式访问。类似地，访问 COM 组件时，C#的“运行时绑定器”会使用知道如何与 COM 组件通信的 **DynamicMetaObject** 派生类型。COM **DynamicMetaObject** 派生类型在 System.Dynamic.dll 程序集中定义。

如果在动态表达式中使用的一个对象的类型未实现 **IDynamicMetaObjectProvider** 接口，C#编译器会将对象视为用 C#定义的普通类型的实例，利用反射在对象上执行操作。

dynamic 的一个限制是只能访问对象的实例成员，因为 **dynamic** 变量必须引用对象。但有时需要动态调用运行时才能确定的一个类型的静态成员。我为此创建了 **StaticMemberDynamicWrapper** 类，它从 **System.Dynamic.DynamicObject** 派生。后者实现了 **IDynamicMetaObjectProvider** 接口。类内部使用了相当多的反射(这个主题将在第 23 章讨论)。以下是我的 **StaticMemberDynamicWrapper** 类的完整代码。

```
internal sealed class StaticMemberDynamicWrapper : DynamicObject {
   private readonly TypeInfo m_type;
   public StaticMemberDynamicWrapper(Type type) { m_type = type.GetTypeInfo(); }

   public override IEnumerable<String> GetDynamicMemberNames() {
      return m_type.DeclaredMembers.Select(mi => mi.Name);
   }

   public override Boolean TryGetMember(GetMemberBinder binder, out object result) {
      result = null;
      var field = FindField(binder.Name);
      if (field != null) { result = field.GetValue(null); return true; }

      var prop = FindProperty(binder.Name, true);
      if (prop != null) { result = prop.GetValue(null, null); return true; }
      return false;
   }

   public override Boolean TrySetMember(SetMemberBinder binder, object value) {
      var field = FindField(binder.Name);
      if (field != null) { field.SetValue(null, value); return true; }

      var prop = FindProperty(binder.Name, false);
      if (prop != null) { prop.SetValue(null, value, null); return true; }
      return false;
   }

   public override Boolean TryInvokeMember(InvokeMemberBinder binder, Object[] args,
      out Object result) {
      MethodInfo method = FindMethod(binder.Name,
         args.Select(c=>c.GetType()).ToArray());
      if (method == null) { result = null; return false; }
      result = method.Invoke(null, args);
      return true;
```

```
}

   private MethodInfo FindMethod(String name, Type[] paramTypes) {
      return m_type.DeclaredMethods.FirstOrDefault(mi => mi.IsPublic && mi.IsStatic
            && mi.Name == name
            && ParametersMatch(mi.GetParameters(), paramTypes));
   }

   private Boolean ParametersMatch(ParameterInfo[] parameters, Type[] paramTypes) {
      if (parameters.Length != paramTypes.Length) return false;
      for (Int32 i = 0; i < parameters.Length; i++)
         if (parameters[i].ParameterType != paramTypes[i]) return false;
      return true;
   }

   private FieldInfo FindField(String name) {
      return m_type.DeclaredFields.FirstOrDefault(fi => fi.IsPublic && fi.IsStatic
         && fi.Name == name);
}

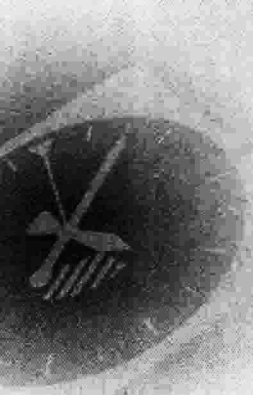
   private PropertyInfo FindProperty(String name, Boolean get) {
      if (get)
         return m_type.DeclaredProperties.FirstOrDefault(
            pi => pi.Name == name && pi.GetMethod != null &&
            pi.GetMethod.IsPublic && pi.GetMethod.IsStatic);

      return m_type.DeclaredProperties.FirstOrDefault(
         pi => pi.Name == name && pi.SetMethod != null &&
            pi.SetMethod.IsPublic && pi.SetMethod.IsStatic);
   }
}
```

为了动态调用静态成员，传递想要操作的 **Type** 来构建上述类的实例，将引用放到 **dynamic** 变量中，再用实例成员语法调用所需静态成员。下例展示如何调用 **String** 的静态 **Concat(String, String)**方法。

```
dynamic stringType = new StaticMemberDynamicWrapper(typeof(String));
var r = stringType.Concat("A", "B"); // 动态调用 String 的静态 Concat 方法
Console.WriteLine(r); // 显示"AB"
```





第 6 章　类型和成员基础

本章内容：

- 类型的各种成员
- 类型的可见性
- 成员的可访问性
- 静态类
- 分部类、结构和接口
- 组件、多态和版本控制

第 4 章和第 5 章重点介绍了类型以及所有类型的所有实例都支持的一组操作，并指出所有类型都可划分为引用类型或值类型。在本章及本部分后续的章节，将解释如何在类型中定义各种成员，从而设计出符合需要的类型。第 7 章～第 11 章将详细讨论各种成员。

6.1　类型的各种成员

类型中可定义 0 个或多个以下种类的成员。

- **常量**　常量是指出数据值恒定不变的符号。这种符号使代码更易阅读和维护。常量总与类型关联，不与类型的实例关联。常量逻辑上总是静态成员。相关内容在第 7 章“常量和字段”讨论。

- **字段**　字段表示只读或可读/可写的数据值。字段可以是静态的；这种字段被认为是*类型*状态的一部分。字段也可以是实例(非静态)；这种字段被认为是*对象*状态的一部分。强烈建议将字段声明为私有，防止类型或对象的状态被类型外部的代码破坏。相关内容在第 7 章讨论。

- **实例构造器**　实例构造器是将新*对象*的实例字段初始化为良好初始状态的特殊方法。相关内容在第 8 章“方法”讨论。

- **类型构造器**　类型构造器是将*类型*的静态字段初始化为良好初始状态的特殊方法。相关内容在第 8 章讨论。

- **方法**　方法是更改或查询类型或对象状态的函数。作用于类型称为静态方法，作用于

对象称为实例方法。方法通常要读写类型或对象的字段。相关内容在第 8 章讨论。

- **操作符重载** 操作符重载实际是方法，定义了当操作符作用于对象时，应该如何操作该对象。由于不是所有编程语言都支持操作符重载，所以操作符重载方法不是“公共语言规范”(Common Language Specification，CLS)的一部分。相关内容在第 8 章讨论。

- **转换操作符** 转换操作符是定义如何隐式或显式将对象从一种类型转型为另一种类型的方法。和操作符重载方法一样，并不是所有编程语言都支持转换操作符，所以不是 CLS 的一部分。相关内容在第 8 章讨论。

- **属性** 属性允许用简单的、字段风格的语法设置或查询类型或对象的逻辑状态，同时保证状态不被破坏。作用于类型称为静态属性，作用于对象称为实例属性。属性可以无参(非常普遍)，也可以有多个参数(相当少见，但集合类用得多)。相关内容在第 10 章“属性”讨论。

- **事件** 静态事件允许类型向一个或多个静态或实例方法发送通知。实例(非静态)事件允许对象向一个或多个静态或实例方法发送通知。引发事件通常是为了响应提供事件的类型或对象的状态的改变。事件包含两个方法，允许静态或实例方法登记或注销对该事件的关注。除了这两个方法，事件通常还用一个委托字段来维护已登记的方法集。相关内容在第 11 章“事件”讨论。

- **类型** 类型可定义其他嵌套类型。通常用这个办法将大的、复杂的类型分解成更小的构建单元(building block)以简化实现。

再次声明，本章宗旨并非详细描述各种成员，而是帮你打好基础，阐明这些成员的共性。

无论什么编程语言，编译器都必须能处理源代码，为上述每种成员生成元数据和 IL 代码。所有编程语言生成的元数据格式完全一致。这正是 CLR 成为“公共语言运行时”的原因。元数据是所有语言都生成和使用的公共信息。正是由于有了元数据，用一种语言写的代码才能无缝访问用另一种语言写的代码。

CLR 还利用公共元数据格式决定常量、字段、构造器、方法、属性和事件在运行时的行为。简单地说，元数据是整个 Microsoft .NET Framework 开发平台的关键，它实现了编程语言、类型和对象的无缝集成。

以下 C#代码展示了一个类型定义，其中包含所有可能的成员。代码能通过编译(有一些警告)，但不能代表你平常创建的类型。大多数方法没有实用价值，仅仅是为了示范编译器如何将类型及其成员转换成元数据。再次说明，后面几章会逐一对这些成员进行讨论。

```
using System;

public sealed class SomeType  {                                  // 1

   // 嵌套类
   private class SomeNestedType{}                                // 2

   // 常量、只读和静态可读/可写字段
   private const Int32 c_SomeConstant = 1;                       // 3
```

```
    private readonly String m_SomeReadOnlyField = "2";      // 4
    private static Int32 s_SomeReadWriteField = 3;          // 5

    // 类型构造器
    static SomeType(){}                                     // 6

    // 实例构造器
    public SomeType(Int32 x) { }                            // 7
    public SomeType() { }                                   // 8

    // 实例方法和静态方法
    private String InstanceMethod() {return null;}          // 9
    public static void Main(){}                             //10

    // 实例属性
    public Int32 SomeProp{                                  //11
       get{ return 0; }                                     //12
       set{  }                                              //13
    }

    // 实例有参属性(索引器)
    public Int32 this[String s] {                           //14
       get{return 0;}                                       //15
       set{}                                                //16
    }

    // 实例事件
    public event EventHandler SomeEvent;                    //17
}
```

编译这个类型，用 ILDasm.exe 查看元数据，将得到如图 6-1 所示的输出。

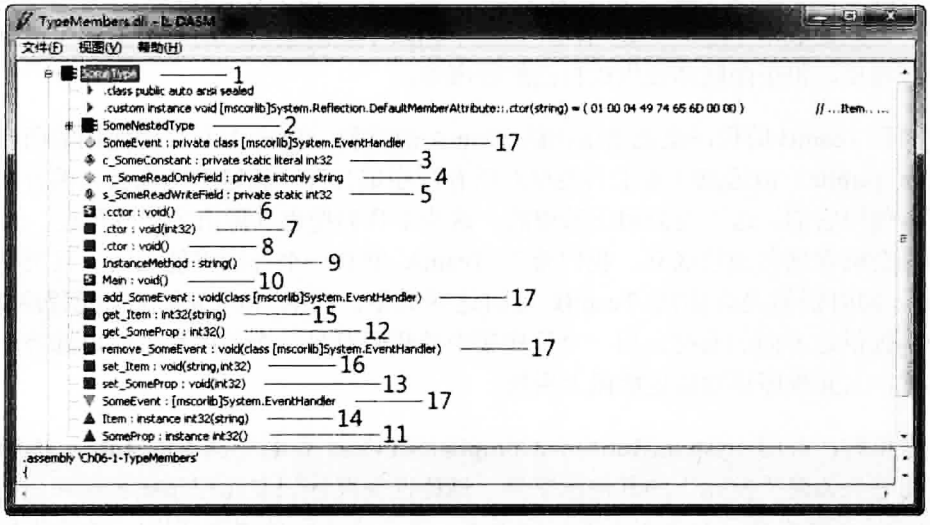


图 6-1　用 ILDasm.exe 查看 SomeType 的元数据

注意，源代码中定义的所有成员都造成编译器生成元数据。事实上，有的成员还造成编译器生成额外的成员和额外的元数据。例如，事件成员(17)造成编译器生成一个字段、两个方法和一些额外的元数据。目前不理解这些内容没有关系。但在学习后面几章时，希望你能回头看看这个例子，体会成员是如何定义的，它们对编译器生成的元数据有何影响。

6.2 类型的可见性

要定义文件范围的类型(而不是将类型定义嵌套到另一个类型中)时，可将类型的可见性指定为 **public** 或 **internal**。**public** 类型不仅对定义程序集中的所有代码可见，还对其他程序集中的代码可见。**internal** 类型则仅对定义程序集中的所有代码可见，对其他程序集中的代码不可见。定义类型时不显式指定可见性，C#编译器会帮你指定为 **internal**(限制比 **public** 大)。下面是几个例子。

```
using System;

// 以下类型的可见性为 public，既可由本程序集中的代码访问，
// 也可由其他程序集中的代码访问
public class ThisIsAPublicType { ... }

// 以下类型的可见性为 internal，只可由本程序集中的代码访问
internal class ThisIsAnInternalType { ... }

// 由于没有显式声明类型的可见性，所以以下类型的可见性为 internal
class ThisIsAlsoAnInternalType { ... }
```

友元程序集

假定下述情形：某公司的团队 TeamA 在某个程序集中定义了一组工具类型(utility type)，并希望公司的另一个团队 TeamB 的成员使用这些类型。但由于各种原因，比如时间安排、地理位置、不同的成本中心或报表结构，这两个团队不能将他们的所有类型都生成到一个程序集中；相反，每个团队都要生成自己的程序集。

为了使团队 TeamB 的程序集能使用团队 TeamA 的类型，TeamA 必须将他们的所有工具类型定义为 **public**。但这意味着工具类型对所有程序集公开。就连另一家公司的开发人员也能写代码使用它们。这不是公司所希望的。这些工具类型也许做出了一些预设，而 TeamB 写代码时会确保这些预设成立。我们希望 TeamA 能有一个办法将他们的工具类型定义为 **internal**，同时仍然允许团队 TeamB 访问这些类型。CLR 和 C#通过**友元程序集**(friend assembly)提供这方面的支持。用一个程序集中的代码对另一个程序集中的内部类型进行单元测试时，友元程序集功能也能派上用场。

生成程序集时，可用 **System.Runtime.CompilerServices** 命名空间中的 **InternalsVisibleTo** 特性标明它认为是“友元”的其他程序集。该特性获取标识友元程序集名称和公钥的字符串参数(传给该特性的字符串绝不能包含版本、语言文化和处理器架构)。注意当程序集认了“友元”之后，友元程序集就能访问该程序集中的所有 **internal** 类型，以及这些类型的 **internal** 成员。下例展示一个程序集如何将两个强命名程序集“Wintellect”和“Microsoft”指定为友元程序集：

```
using System;
using System.Runtime.CompilerServices; // 为了 InternalsVisibleTo 特性

// 当前程序集中的 internal 类型可由以下两个程序集中
```

```
// 的任何代码访问(不管什么版本或语言文化)
[assembly:InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]
[assembly:InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]

internal sealed class SomeInternalType { ... }
internal sealed class AnotherInternalType { ... }
```

从友元程序集访问上述程序集的 **internal** 类型很容易。例如，下面展示了公钥为“12345678...90abcdef”的友元程序集“Wintellect”如何访问上述程序集的 **internal** 类型 **SomeInternalType**。

```
using System;

internal sealed class Foo {
   private static Object SomeMethod() {
    // 这个"Wintellect"程序集能访问另一个程序集的 internal 类型，
    // 就好像那是 public 类型
    SomeInternalType sit = new SomeInternalType();
    return sit;
   }
}
```

由于程序集中的类型的 **internal** 成员能从友元程序集访问，所以要慎重考虑类型成员的可访问性，以及要将哪些程序集声明为友元。注意 C#编译器在编译友元程序集(不含 **InternalsVisibleTo** 特性的程序集)时要求使用编译器开关**/out:<*file*>**。使用这个编译器开关的原因在于，编译器需要知道准备编译的程序集的名称，从而判断生成的程序集是不是友元程序集。你或许以为 C#编译器能自己判断，因为平时都是它自己确定输出文件名。但事实上，在代码结束编译之前，C#编译器是不知道输出文件名的。因此，使用**/out:<*file*>**编译器开关能极大增强编译性能。

同样地，如果使用 C#编译器的**/t:module** 开关来编译模块(而不是编译成程序集)，而且该模块将成为某个友元程序集的一部分，那么还需要使用 C#编译器的**/moduleassemblyname:<*string*>**开关来编译该模块，它告诉编译器该模块将成为哪个程序集的一部分，使编译器设置模块中的代码，使它们能访问另一个程序集中的 **internal** 类型。

6.3　成员的可访问性

定义类型的成员(包括嵌套类型)时，可指定成员的**可访问性**。在代码中引用成员时，成员的可访问性指出引用是否合法。CLR 自己定义了一组可访问性修饰符，但每种编程语言在向成员应用可访问性时，都选择了自己的一组术语以及相应的语法。例如，CLR 使用 **Assembly** 表明成员对同一程序集内的所有代码可见，而 C#对应的术语是 **internal**。

表 6-1 总结了 6 个应用于成员的可访问性修饰符。从第一行到最后一行，按照从限制最大(**Private**)到限制最小(**Public**)的顺序排列。

当然，任何成员要想被访问，都必须在可见的类型中定义。例如，如果程序集 AssemblyA 定义了含有一个 **public** 方法的 **internal** 类型，则程序集 AssemblyB 中的代码不能调用该 **public** 方法，因为 **internal** 类型对 AssemblyB 来说不可见。

表 6-1 成员的可访问性

CLR 术语	C#术语	描述
Private	**private**	成员只能由定义类型或任何嵌套类型中的方法访问
Family	**protected**	成员只能由定义类型、任何嵌套类型或者不管在什么程序集中的派生类型中的方法访问
Family and Assembly	(不支持)	成员只能由定义类型、任何嵌套类型或者同一程序集中定义的任何派生类型中的方法访问
Assembly	**internal**	成员只能由定义程序集中的方法访问
Family or Assembly	**protected internal**	成员可由任何嵌套类型、任何派生类型(不管在什么程序集)或者定义程序集中的任何方法访问
Public	**public**	成员可由任何程序集的任何方法访问

编译代码时，编程语言的编译器检查代码是不是正确引用了类型和成员。如果代码不正确地引用了类型或成员，编译器会生成一条合适的错误消息。另外，在运行时将 IL 代码编译成本机 CPU 指令时，JIT 编译器也会确保对字段和方法的引用合法。例如，JIT 编译器如果检测到代码不正确地访问私有字段或方法，将分别抛出 **FieldAccessException** 或 **MethodAccessException** 异常。

通过对 IL 代码进行验证，可确保被引用成员的可访问性在运行时得到正确兑现——即使语言的编译器忽略了对可访问性的检查。另外极有可能发生的情况是：语言编译器编译的代码会访问另一个程序集中的另一个类型的 **public** 成员，但到运行时却加载了程序集的不同版本，新版本中的 **public** 成员变成了 **protected** 或 **private** 成员。

在 C#中，如果没有显式声明成员的可访问性，编译器通常(但并不总是)默认选择 **private**(限制最大的那个)。CLR 要求接口类型的所有成员都具有 **public** 可访问性。C#编译器知道这一点，因此禁止开发人员显式指定接口成员的可访问性；编译器自动将所有成员的可访问性设为 **public**。

> **更多信息** 参考 C#语言规范的“Declared Accessibility”(已声明可访问性)一节，完整地了解可在 C#中向类型和成员应用哪些可访问性，以及如何根据声明的上下文来选择默认的可访问性。

你也许注意到了，CLR 提供了称为 **Family and Assembly** 的可访问性。但 C#不支持。C#开发团队认为这种可访问性基本没用，所以决定放弃。

派生类型重写基类型定义的成员时，C#编译器要求原始成员和重写成员具有相同的可访问性。也就是说，如果基类成员是 **protected** 的，派生类中的重写成员也必须是 **protected** 的。但这是 C#的限制，不是 CLR 的。从基类派生时，CLR 允许放宽但不允许收紧成员的可访问性限制。例如，类可重写基类定义的 **protected** 方法，将重写方法设为 **public**(放宽限制)。

但不能重写基类定义的 **protected** 方法，将重写方法设为 **private**(收紧限制)。之所以不能在派生类中收紧对基类方法的访问，是因为 CLR 承诺派生类总能转型为基类，并获取对基类方法的访问权。如果允许派生类收紧限制，CLR 的承诺就无法兑现了。

6.4　静态类

有一些永远不需要实例化的类，例如 **Console**，**Math**，**Environment** 和 **ThreadPool**。这些类只有 **static** 成员。事实上，这种类唯一的作用就是组合一组相关的成员。例如，**Math** 类就定义了一组执行数学运算的方法。在 C#中，要用 **static** 关键字定义不可实例化的类。该关键字只能应用于类，不能应用于结构(值类型)。因为 CLR 总是允许值类型实例化，这是没办法阻止的。

C#编译器对静态类进行了如下限制。

- 静态类必须直接从基类 **System.Object** 派生，从其他任何基类派生都没有意义。继承只适用于对象，而你不能创建静态类的实例。
- 静态类不能实现任何接口，这是因为只有使用类的实例时，才可调用类的接口方法。
- 静态类只能定义静态成员(字段、方法、属性和事件)，任何实例成员都会导致编译器报错。
- 静态类不能作为字段、方法参数或局部变量使用，因为它们都代表引用了实例的变量，而这是不允许的。编译器检测到任何这样的用法都会报错。

下面是一个定义了静态成员的静态类。代码虽能通过编译(有一个警告)，但该类没有做任何有意义的事情。

```
using System;

   public static class AStaticClass {
     public static void AStaticMethod() { }

     public static String AStaticProperty {
       get { return s_AStaticField; }
       set { s_AStaticField = value; }
   }

   private static String s_AStaticField;

   public static event EventHandler AStaticEvent;
}
```

将上述代码编译成库(DLL)程序集，用 ILDasm.exe 查看会得到如图 6-2 所示的结果。如你所见，使用关键字 **static** 定义类，将导致 C#编译器将该类标记为 **abstract** 和 **sealed**。另外，编译器不在类型中生成实例构造器方法，你在图 6-2 中看不到实例构造器(.ctor)方法。



图 6-2 ILDasm.exe 表明静态类在元数据中是抽象密封类

6.5 分部类、结构和接口

本节要讨论分部类、结构和接口。**partial** 关键字告诉 C#编译器：类、结构或接口的定义源代码可能要分散到一个或多个源代码文件中。将类型源代码分散到多个文件的原因有三。

- **源代码控制**

  假定类型定义包含大量源代码，一个程序员把它从源代码控制系统中签出(check out)以进行修改。没有其他程序员能同时修改这个类型，除非之后执行合并(merge)。使用 **partial** 关键字可将类型的代码分散到多个源代码文件中，每个文件都可单独签出，多个程序员能同时编辑类型。

- **在同一个文件中将类或结构分解成不同的逻辑单元**

  我有时会创建一个类型来提供多个功能，使类型能提供完整解决方案。为简化实现，有时会在一个源代码文件中重复声明同一个分部类型。然后，分部类型的每个部分都实现一个功能，并配以它的全部字段、方法、属性、事件等。这样就可方便地看到组合以提供一个功能的全体成员，从而简化编码。与此同时，可以方便地将分部类型的一部分注释掉，以便从类中删除一个完整的功能，代之以另一个实现(通过分部类型的一个新的部分)。

- **代码拆分**

  在 Microsoft Visual Studio 中创建新项目时，一些源代码文件会作为项目一部分自动创建。这些源代码文件包含模板，能为项目开个好头。使用 Visual Studio 在设计图面上拖放控件时，Visual Studio 自动生成源代码，并将代码拆分到不同的源代码文件中。这提高了开发效率。很久以前，生成的代码是直接放到当前正在处理的那个源代码文件中的。这样做的问题在于，如果不小心编辑了一下生成的代码，设计器行为就可能失常。从 Visual Studio 2005 开始，新建窗体、控件等的时候，Visual Studio 自动创建两个源代码文件：一个用于你的代码，另一个用于设计器生成的代码。由于设计器的代码在单独的文件中，所以基本上杜绝了不小心编辑到它的可能。

要将 **partial** 关键字应用于所有文件中的类型。这些文件编译到一起时，编译器会合并代码，在最后的.exe 或.dll 程序集文件(或.netmodule 模块文件)中生成单个类型。“分部类型”功能完全由 C#编译器实现，CLR 对该功能一无所知，这解释了一个类型的所有源代码文件为什么必须使用相同编程语言，而且必须作为一个编译单元编译到一起。

6.6　组件、多态和版本控制

面向对象编程(Object-Oriented Programming，OOP)已问世多年。它在上个世纪 70 年代末、80 年代初首次投入应用时，应用程序规模还非常小，而且使应用程序运行起来所需的全部代码都由同一家公司编写。当然，那时确实有操作系统，应用程序也确实使用了操作系统的一些功能，但和今天的操作系统相比，那时的操作系统所提供的功能实在是太少了。

如今软件变得相当复杂，而且用户希望应用程序提供更丰富的功能，如 GUI、菜单、鼠标输入、手写板输入、打印输出、网络功能等。正是由于这个原因，操作系统和开发平台在这几年中取得了迅猛发展。另外，应用程序的开发也必须分工。不能再像以前那样，一个或几个开发人员就能写出一个应用程序需要的全部代码。这样做要么不可能，要么效率太低。现在的应用程序一般都包含了由许多不同的公司生成的代码。这些代码通过面向对象编程机制契合到一起。

组件软件编程(Component Software Programming，CSP)正是 OOP 发展到极致的成果。下面列举组件的一些特点。

- 组件(.NET Framework 称为程序集)有“已经发布”的意思。
- 组件有自己的标识(名称、版本、语言文化和公钥)。
- 组件永远维持自己的标识(程序集中的代码永远不会静态链接到另一个程序集中；.NET 总是使用动态链接)。
- 组件清楚指明它所依赖的组件(引用元数据表)。
- 组件应编档它的类和成员。C#语言通过源代码内的 XML 文档和编译器的**/doc** 命令行开关提供这个功能。
- 组件必须指定它需要的安全权限。CLR 的代码访问安全性(Code Access Security，CAS)机制提供这个功能。
- 组件要发布在任何“维护版本”中都不会改变的接口(对象模型)。“维护版本”(servicing version)代表组件的新版本，它向后兼容组件的原始版本。通常，“维护版本”包含 bug 修复、安全补丁或者一些小的功能增强。但在“维护版本”中，不能要求任何新的依赖关系，也不能要求任何附加的安全权限。

如最后一点所述，CSP 有很大一部分涉及版本控制。组件随着时间而改变，并根据不同的时间表来发布。版本控制使 CSP 的复杂性上升到了 OOP 无法企及的高度。(在 OOP 中，全

部代码都由一家公司编写和测试，并作为一个整体发布。)本节将重点放在组件的版本控制上。

.NET Framework 中的版本号包含 4 个部分：主版本号(major version)、次版本号(minor version)、内部版本号(build number)和修订号(revision)。例如，版本号为 1.2.3.4 的程序集，其主版本号为 1，次版本号为 2，内部版本号为 3，修订号为 4。major/minor 部分通常代表程序集的一个连续的、稳定的功能集，而 build /revision 部分通常代表对这个功能集的一次维护。

假定某公司发布了版本号为 2.7.0.0 的程序集。之后，为了修复该组件的 bug，他们可以生成一个新的程序集，并只改动版本号的 build/revision 部分，比如 2.7.1.34。这表明该程序集是维护版本，向后兼容原始版本(2.7.0.0)。

另一方面，假定该公司想生成程序集的新版本，而且由于发生了重大变化，所以不准备向后兼容程序集的原始版本。在这种情况下，公司实际是要创建全新组件，major/minor 版本号(比如 3.0.0.0)应该和原来的组件不同。



注意 此处只是说明应该如何看待版本号。遗憾的是，CLR 不以这种方式看待版本号。现在，CLR 将版本号看成是固定值，如果某个程序集依赖版本号为 1.2.3.4 的另一个程序集，CLR 只会尝试加载版本号为 1.2.3.4 的程序集(除非设置了绑定重定向)。

前面讨论了如何使用版本号更新组件的标识，从而反映出组件的新版本。下面要讨论如何利用 CLR 和编程语言(比如 C#)提供的功能来自动适应组件可能发生的变化。

将一个组件(程序集)中定义的类型作为另一个组件(程序集)中的一个类型的基类使用时，便会发生版本控制问题。显然，如果基类的版本(被修改得)低于派生类，派生类的行为也会改变，这可能造成类的行为失常。在多态情形中，由于派生类型会重写基类型定义的虚方法，所以这个问题显得尤其突出。

C#提供了 5 个能影响组件版本控制的关键字，可将它们应用于类型以及/或者类型成员。这些关键字直接对应 CLR 用于支持组件版本控制的功能。表 6-2 总结了与组件版本控制相关的 C#关键字，并描述了每个关键字如何影响类型或者类型成员的定义。

表 6-2 C#关键字及其对组件版本控制的影响

C#关键字	类型	方法/属性/事件	常量/字段
abstract	表示不能构造该类型的实例	表示为了构造派生类型的实例，派生类型必须重写并实现这个成员	(不允许)
virtual	(不允许)	表示这个成员可由派生类型重写	(不允许)
override	(不允许)	表示派生类型正在重写基类型的成员	(不允许)
sealed	表示该类型不能用作基类型	表示这个成员不能被派生类型重写，只能将该关键字应用于重写虚方法的方法	(不允许)
new	应用于嵌套类型、方法、属性、事件、常量或字段时，表示该成员与基类中相似的成员无任何关系		

6.6.3 节“对类型进行版本控制时的虚方法的处理”将演示这些关键字的作用和用法。但在讨论版本控制之前，先要讨论一下 CLR 实际如何调用虚方法。

6.6.1　CLR 如何调用虚方法、属性和事件

本节重点是方法，但我们的讨论也与虚属性和虚事件密切相关。属性和事件实际作为方法实现，本书后面会用专门的章来讨论它们。

方法代表在类型或类型的实例上执行某些操作的代码。在类型上执行操作，称为**静态方法**；在类型的实例上执行操作，称为**非静态方法**。所有方法都有名称、签名和返回类型(可为 **void**)。CLR 允许类型定义多个同名方法，只要每个方法都有一组不同的参数或者一个不同的返回类型。所以，完全能定义两个同名、同参数的方法，只要两者返回类型不同。但除了 IL 汇编语言，我没有发现任何利用了这一“特点”的语言。大多数语言(包括 C#)在判断方法的唯一性时，除了方法名之外，都只以参数为准，方法返回类型会被忽略。(C#在定义转换操作符方法时实际上放宽了此限制，详见第 8 章。)

以下 **Employee** 类定义了 3 种不同的方法：

```
internal class Employee {
   // 非虚实例方法
   public              Int32    GetYearsEmployed() { ... }

   // 虚方法(虚暗示实例)
   public virtual      String   GetProgressReport() { ... }

   // 静态方法
   public static       Employee Lookup(String name) { ... }
}
```

编译上述代码，编译器会在程序集的方法定义表中写入 3 个记录项，每个记录项都用一组标志(flag)指明方法是实例方法、虚方法还是静态方法。

写代码调用这些方法，生成调用代码的编译器会检查方法定义的标志(flag)，判断应如何生成 IL 代码来正确调用方法。CLR 提供两个方法调用指令。

- call
  该 IL 指令可调用静态方法、实例方法和虚方法。用 **call** 指令调用静态方法，必须指定方法的定义类型。用 **call** 指令调用实例方法或虚方法，必须指定引用了对象的变量。**call** 指令假定该变量不为 **null**。换言之，变量本身的类型指明了方法的定义类型。如果变量的类型没有定义该方法，就检查基类型来查找匹配方法。**call** 指令经常用于以非虚方式调用虚方法。

- callvirt
  该 IL 指令可调用实例方法和虚方法，不能调用静态方法。用 **callvirt** 指令调用实例方法或虚方法，必须指定引用了对象的变量。用 **callvirt** 指令调用非虚实例方法，变量的类型指明了方法的定义类型。用 **callvirt** 指令调用虚实例方法，CLR 调查发出调用的对象的实际类型，然后以多态方式调用方法。为了确定类型，发出调用的变量绝不能

是 **null**。换言之，编译这个调用时，JIT 编译器会生成代码来验证变量的值是不是 **null**。如果是，**callvirt** 指令造成 CLR 抛出 **NullReferenceException** 异常。正是由于要进行这种额外的检查，所以 **callvirt** 指令的执行速度比 **call** 指令稍慢。注意，即使 **callvirt** 指令调用的是非虚实例方法，也要执行这种 **null** 检查。

现在综合运用上述知识，看看 C#如何使用这些不同的 IL 指令：

```
using System;

public sealed class Program{
   public static void Main(){
      Console.WriteLine();     //调用静态方法

      Object o = new Object();
      o.GetHashCode();         //调用虚实例方法
      o.GetType();             //调用非虚实例方法
   }
}
```

编译上述代码，查看最后得到的 IL，结果如下：

```
.method public hidebysig static void Main() cil managed {
  .entrypoint
  // Code size 26 (0x1a)
  .maxstack 1
  .locals init (object o)
  IL_0000: call void System.Console::WriteLine()
  IL_0005: newobj instance void System.Object::.ctor()
  IL_000a: stloc.0
  IL_000b: ldloc.0
  IL_000c: callvirt instance int32 System.Object::GetHashCode()
  IL_0011: pop
  IL_0012: ldloc.0
  IL_0013: callvirt instance class System.Type System.Object::GetType()
  IL_0018: pop
  IL_0019: ret
} // end of method Program::Main
```

注意，C#编译器用 **call** 指令调用 **Console** 的 **WriteLine** 方法。这在意料之中，因为 **WriteLine** 是静态方法。接着用 **callvirt** 指令调用 **GetHashCode**，这也在意料之中，因为 **GetHashCode** 是虚方法。最后，C#编译器用 **callvirt** 指令调用 **GetType** 方法。这就有点出乎意料了，因为 **GetType** 不是虚方法。但这是可行的，因为对代码进行 JIT 编译时，CLR 知道 **GetType** 不是虚方法，所以在 JIT 编译好的代码中，会直接以非虚方式调用 **GetType**。

那么，为什么 C#编译器不干脆生成 **call** 指令呢？答案是 C#团队认为，JIT 编译器应生成代码来验证发出调用的对象不为 **null**。这意味着对非虚实例方法的调用要稍慢一点。这也意味着以下 C#代码将抛出 **NullReferenceException** 异常。注意，在另一些编程语言中，以下代码能正常工作。

```
using System;

public sealed class Program{
   public Int32 GetFive(){ return 5; }
   public static void Main(){
      Program p = null;
      Int32 x = p.GetFive();  // 在 C#中抛出 NullReferenceException 异常
   }
}
```

上述代码理论上并无问题。变量 **p** 确实为 **null**，但在调用非虚方法(**GetFive**)时，CLR 唯一需要知道的就是 **p** 的数据类型(**Program**)。如果真的调用 **GetFive**，**this** 实参值是 **null**。由于 **GetFive** 方法内部并未使用该实参，所以不会抛出 **NullReferenceException** 异常。但由于 C#编译器生成 **callvirt** 而不是 **call** 指令，所以上述代码抛出了 **NullReferenceException** 异常。



重要提示　将方法定义为非虚方法后，将来永远都不要把它更改为虚方法。这是因为某些编译器会用 call 而不是 callvirt 调用非虚方法。如果方法从非虚变成虚，而引用代码没有重新编译，会以非虚方式调用虚方法，造成应用程序行为无法预料。用 C#写的引用代码不会出问题，因为 C#用 callvirt 指令调用所有实例方法。但如果引用代码是用其他语言写的，就可能出问题。

编译器有时用 **call** 而不是 **callvirt** 调用虚方法。虽然刚开始有点难以理解，但以下代码证明了有时真的需要这样做：

```
internal class SomeClass {
   // ToString 是基类 Object 定义的虚方法
   public override String ToString() {

   // 编译器使用 IL 指令'call'，
   // 以非虚方式调用 Object 的 ToString 方法

   // 如果编译器用'callvirt'而不是'call'，
   // 那么该方法将递归调用自身，直至栈溢出
   return base.ToString();
   }
}
```

调用虚方法 **base.ToString** 时，C#编译器生成 **call** 指令来确保以非虚方式调用基类的 **ToString** 方法。这是必要的，因为如果以虚方式调用 **ToString**，调用会递归执行，直至线程栈溢出，这显然不是你所期望的。

编译器调用值类型定义的方法时倾向于使用 **call** 指令，因为值类型是密封的。这意味着即使值类型含有虚方法也不用考虑多态性，这使调用更快。此外，值类型实例的本质保证它永不为 **null**，所以永远不抛出 **NullReferenceException** 异常。最后，如果以虚方式调用值类型中的虚方法，CLR 要获取对值类型的类型对象的引用，以便引用(类型对象中的)方法表，这要求对值类型装箱。装箱对堆造成更大压力，迫使进行更频繁的垃圾回收，使性能受到影响。

无论用 **call** 还是 **callvirt** 调用实例方法或虚方法，这些方法通常接收隐藏的 **this** 实参作为方法第一个参数。**this** 实参引用要操作的对象。

设计类型时应尽量减少虚方法数量。首先，调用虚方法的速度比调用非虚方法慢。其次，JIT 编译器不能内嵌(inline)虚方法，这进一步影响性能。第三，虚方法使组件版本控制变得更脆弱，详情参见下一节。第四，定义基类型时，经常要提供一组重载的简便方法(convenience method)。如果希望这些方法是多态的，最好的办法就是使最复杂的方法成为

虚方法，使所有重载的简便方法成为非虚方法。顺便说一句，遵循这个原则，还可在改善组件版本控制的同时，不至于对派生类型产生负面影响。下面是一个例子：

```
public class Set {
  private Int32 m_length = 0;

  // 这个重载的简便方法是非虚的
  public Int32 Find(Object value) {
    return Find(value, 0, m_length);
  }

  // 这个重载的简便方法是非虚的
  public Int32 Find(Object value, Int32 startIndex) {
    return Find(value, startIndex, m_length - startIndex);
  }

  // 功能最丰富的方法是虚方法，可以被重写
  public virtual Int32 Find(Object value, Int32 startIndex, Int32 endIndex)
  {
     // 可被重写的实现放在这里…
  }

     // 其他方法放在这里
}
```

6.6.2 合理使用类型的可见性和成员的可访问性

使用.NET Framework 时，应用程序是由多个公司生产的多个程序集所定义的类型构成的。这意味着开发人员对所用的组件以及其中定义的类型几乎没有什么控制权。开发人员通常无法访问源代码(甚至可能不知道组件用什么编程语言创建)，而且不同组件的版本发布一般都基于不同的时间表。除此之外，由于多态和受保护成员，基类开发人员必须信任派生类开发人员所写的代码。当然，派生类的开发人员也必须信任从基类继承的代码。设计组件和类型时，应慎重考虑这些问题。

本节描述了设计类型时应如何思考这些问题，具体就是如何正确设置类型的可见性和成员的可访问性来取得最优结果。

首先，定义新类型时编译器应默认生成密封类，使它不能作为基类使用。但是，包括 C# 编译器在内的许多编译器都默认生成非密封类，只是允许开发人员使用关键字 **sealed** 将类显式标记为密封。我认为现在的编译器使用了错误的默认设定。不过，亡羊补牢，为时不晚，希望将来的编译器能改正这一错误。密封类之所以比非密封类更好，有以下三个方面的原因。

- **版本控制**
  如果类最初密封，将来可在不破坏兼容性的前提下更改为非密封。但如果最初非密封，将来就不可能更改为密封，因为这将中断派生类。另外，如果非密封类定义了非密封虚方法，必须在新版本的类中保持虚方法调用顺序，否则可能中断派生类。

- **性能**
  如上一节所述，调用虚方法在性能上不及调用非虚方法，因为 CLR 必须在运行时查找

对象的类型，判断要调用的方法由哪个类型定义。但是，如果 JIT 编译器看到使用密封类型的虚方法调用，就可采用非虚方式调用虚方法，从而生成更高效的代码。之所以能这么做，是因为密封类自然不会有派生类。例如，在下面的代码中，JIT 编译器可采用非虚方式调用虚方法 **ToString**：

```
using System;
public sealed class Point {
   private Int32 m_x, m_y;

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

   public override String ToString() {
      return String.Format("({0}, {1})", m_x, m_y);
   }

   public static void Main() {
      Point p = new Point(3, 4);

      // C#编译器在此生成 callvirt 指令，
      // 但 JIT 编译器将优化这个调用，并生
      // 成代码来非虚地调用 ToString。这
      // 是因为 p 的类型是 Point，而 Point
      // 是密封类
      Console.WriteLine(p.ToString());
   }
}
```

- **安全性和可预测性**

  类必须保护自己的状态，不允许被破坏。当类处于非密封状态时，只要它的任何数据字段或者在内部对这些字段进行处理的方法是可以访问的，而且不是私有的，派生类就能访问和更改基类的状态。另外，派生类既可重写基类的虚方法，也可直接调用这个虚方法在基类中的实现。一旦将某个方法、属性或事件设为 **virtual**，基类就会丧失对它的行为和状态的部分控制权。所以，除非经过了认真考虑，否则这种做法可能导致对象的行为变得不可预测，还可能留下安全隐患。

密封类的问题是可能给类型的用户造成巨大的不便。有时，开发人员希望从现有类型派生出一个类，在其中添加额外字段或状态信息来满足自己应用程序的需要。他们甚至希望在派生类中定义辅助方法(helper method)或简便方法(convenience method)来操纵这些额外的字段。虽然 CLR 没有提供机制允许你用辅助方法或字段扩展已生成的类型，但可利用 C#的扩展方法(第 8 章)模拟辅助方法，还可利用 **ConditionalWeakTable** 类(第 21 章)模拟跟踪状态。

以下是我自己定义类时遵循的原则。

- 定义类时，除非确定要将其作为基类，并允许派生类对它进行特化[①]，否则总是显式地指定为 **sealed** 类。如前所述，这与 C#以及其他许多编译器的默认方式相反。另外，我默认将类指定为 **internal** 类，除非我希望在程序集外部公开这个类。幸好，如果不显式指定类型的可见性，C#编译器默认使用的就是 **internal**。如果我真的要定义一个可

① 特化（specialization）是指继承了基类的东西不算，还对这些东西进行特殊处理。——译注

由其他人继承的类，同时不希望允许特化，那么我会重写并密封继承的所有虚方法。

- 类的内部，我总是毫不犹豫地将数据字段定义为 **private**。幸好，C#默认就将字段标记为 **private**。事实上，我情愿 C#强制所有字段都标记为 **private**，根本不允许 **protected**，**internal** 和 **public** 等等。状态一旦公开，就极易产生问题，造成对象的行为无法预测，并留下安全隐患。即使只将一些字段声明为 **internal** 也会如此。即使在单个程序集中，也很难跟踪引用了一个字段的所有代码，尤其是假如代码由几个开发人员编写，并编译到同一个程序集中。

- 在类的内部，我总是将自己的方法、属性和事件定义为 **private** 和非虚。幸好，C#默认也是这样的。当然，我会将某个方法、属性和事件定义为 **public**，以便公开类型的某些功能。我会尽量避免将上述任何成员定义为 **protected** 或 **internal**，因为这会使类型面临更大的安全风险。即使迫不得已，我也会尽量选择 **protected** 或 **internal**。**virtual** 永远最后才考虑，因为虚成员会放弃许多控制，丧失独立性，变得彻底依赖于派生类的正确行为。

- OOP 有一条古老的格言，大意是当事情变得过于复杂时，就搞更多的类型出来。当算法的实现开始变得复杂时，我会定义一些辅助类型来封装独立的功能。如果定义的辅助类型只由一个“超类型”使用，我会在“超类型”中嵌套这些辅助类型。这样除了可以限制范围，还允许嵌套的辅助类型中的代码引用“超类型”中定义的私有成员。但是，Visual Studio 的代码分析工具(FxCopCmd.exe)强制执行了一条设计规则，即对外公开的嵌套类型必须在文件或程序集范围中定义，不能在另一个类型中定义。之所以会有这个规则，是因为一些开发人员觉得引用嵌套类型时，所用的语法过于繁琐。我赞同该规则，自己绝不会定义公共嵌套类型。

6.6.3 对类型进行版本控制时的虚方法的处理

如前所述，在组件软件编程(Component Software Programming，CSP)环境中，版本控制是非常重要的问题。第 3 章已讨论了部分版本控制问题。那一章解释了强命名程序集，并讨论了管理员如何确保应用程序绑定到和生成/测试时一样的程序集。但是，还有其他版本控制问题会造成源代码兼容性问题。例如，如果类型要作为基类型使用，增加或修改它的成员时务必非常小心。下面来看一些例子。

假定 CompanyA 定义了 **Phone** 类型：

```
namespace CompanyA {
   public class Phone {
      public void Dial() {
         Console.WriteLine("Phone.Dial");
         //在这里执行拨号操作
      }
   }
}
```

再假定 CompanyB 定义了 **BetterPhone** 类型，使用 CompanyA 的 **Phone** 类型作为基类型：

```
namespace CompanyB {
```

```
    public class BetterPhone : CompanyA.Phone {
        public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

CompanyB 编译上述代码时，C#编译器生成以下警告消息：

```
warning CS0108:"CompanyB.BetterPhone.Dial()"隐藏了继承的成员
                    "CompanyA.Phone.Dial()"。如果是有意隐藏，请使用关键字 new
```

该警告告诉开发人员 **BetterPhone** 类正在定义个 **Dial** 方法，它会隐藏 **Phone** 类定义的 **Dial**。新方法可能改变 **Dial** 的语义(这个语义是 CompanyA 最初创建 **Dial** 方法时定义的)。

编译器就潜在的语言不匹配问题发出警告，这是一个令人欣赏的设计。编译器甚至贴心地告诉你如何消除这条警告，办法是在 **BetterPhone** 类中定义 **Dial** 时，在前面加一个 **new** 关键字。以下是改正的 **BetterPhone** 类：

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // 新的 Dial 方法变得与 Phone 的 Dial 方法无关了
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

现在，CompanyB 能在其应用程序中使用 **BetterPhone.Dial**。以下是 CompanyB 可能写的一些示例代码：

```
public sealed class Program{
    public static void Main(){
        CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
        phone.Dial();
    }
}
```

运行上述代码，输出结果如下所示：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
```

输出符合 CompanyB 的预期。调用 **Dial** 方法时，会调用由 **BetterPhone** 类定义的新 **Dial**。在新 **Dial** 中，先调用虚方法 **EstablishConnection**，再调用基类型 **Phone** 中的 **Dial** 方法。

现在，假定几家公司计划使用 CompanyA 的 **Phone** 类型。再假定这几家公司都认为在 **Dial** 方法中建立连接的主意非常好。CompanyA 收到这个反馈，决定对 **Phone** 类进行修订：

```
namespace CompanyA{
   public class Phone{
      public void Dial(){
         Console.WriteLine("Phone.Dial");
         EstablishConnection();
         // 在这里执行拨号操作
      }

      protected virtual void EstablishConnection(){
         Console.WriteLine("Phone.EstablishConnection");
         // 在这里执行建立连接的操作
      }
   }
}
```

现在，一旦 CompanyB 编译它的 **BetterPhone** 类型(从 CompanyA 的新版本 **Phone** 派生)，编译器将生成以下警告消息：

```
warning CS0114:  "CompanyB.BetterPhone.EstablishConnection()"将隐藏继承的
                      成员"CompanyA.Phone.EstablishConnection()"。若要使当前成员
                      重写该实现，请添加关键字 override。否则，添加关键字 new。
```

编译器警告 **Phone** 和 **BetterPhone** 都提供了 **EstablishConnection** 方法，而且两者的语义可能不一致。只是简单地重新编译 **BetterPhone**，可能无法获得和使用第一个版本的 **Phone** 类型时相同的行为。

如果 CompanyB 认定 **EstablishConnection** 方法在两个类型中的语义不一致，CompanyB 可以告诉编译器使用 **BetterPhone** 类中定义的 **Dial** 和 **EstablishConnection** 方法，它们与基类型 **Phone** 中定义的 **EstablishConnection** 方法没有关系。CompanyB 可以为 **EstablishConnection** 方法添加 **new** 关键字来告诉编译器这一点。

```
namespace CompanyB {
   public class BetterPhone : CompanyA.Phone {

      // 保留关键字 new，指明该方法与基类型的
      // Dial 方法没有关系
      public new void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      // 为这个方法添加关键字 new，指明该方法与基类型的
      // EstablishConnection 方法没有关系
      protected new virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // 在这里执行建立连接的操作
      }
   }
}
```

在这段代码中，关键字 **new** 告诉编译器生成元数据，让 CLR 知道 **BetterPhone** 类型的 **EstablishConnection** 方法应被视为由 **BetterPhone** 类型引入的新函数。这样 CLR 就知道 **Phone** 和 **BetterPhone** 这两个类中的方法无任何关系。

执行相同的应用程序代码(前面列出的 **Main** 方法中的代码)，输出结果如下所示：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

这个输出表明，在 **Main** 方法中调用 **Dial**，调用的是 **BetterPhone** 类定义的新 **Dial** 方法。后者调用了同样由 **BetterPhone** 类定义的虚方法 **EstablishConnection**。**BetterPhone** 的 **EstablishConnection** 方法返回后，将调用 **Phone** 的 **Dial** 方法。**Phone** 的 **Dial** 方法调用了 **EstablishConnection**，但由于 **BetterPhone** 的 **EstablishConnection** 使用 **new** 进行了标识，所以不认为 **BetterPhone** 的 **EstablishConnection** 是对 **Phone** 的虚方法 **EstablishConnection** 的重写。最终结果是，**Phone** 的 **Dial** 方法调用了 **Phone** 的 **EstablishConnection** 方法——这正是我们所期望的。



注意　如果编译器像原生 C++编译器那样默认将方法视为重写，**BetterPhone** 的开发者就不能使用 **Dial** 和 **EstablishConnection** 方法名了。这极有可能造成整个源代码 base 的连锁反应，破坏源代码和二进制兼容性。这种波及面太大的改变是我们不希望的，尤其是中大型的项目。但是，如果更改方法名只会造成源代码发生适度更新，就应该更改方法名，避免 **Dial** 和 **EstablishConnection** 方法的两种不同的含义使开发人员产生混淆。

还有一个办法是，**CompanyB** 可以获得 **CompanyA** 的新版本 **Phone** 类型，并确定 **Dial** 和 **EstablishConnection** 在 **Phone** 中的语义正好是他们所希望的。这种情况下，**CompanyB** 可通过完全移除 **Dial** 方法来修改他们的 **BetterPhone** 类型。另外，由于 **CompanyB** 现在希望告诉编译器，**BetterPhone** 的 **EstablishConnection** 方法和 **Phone** 的 **EstablishConnection** 方法是相关的，所以必须移除 **new** 关键字。但是，仅仅移除 **new** 关键字还不够，因为编译器目前还无法准确判断 **BetterPhone** 的 **EstablishConnection** 方法的意图。为了准确表示意图，**CompanyB** 的开发人员还必须将 **BetterPhone** 的 **EstablishConnection** 方法由 **virtual** 改变为 **override**。以下代码展示了新版本的 **BetterPhone** 类。

```
namespace CompanyB{
    public class BetterPhone:CompanyA.Phone{

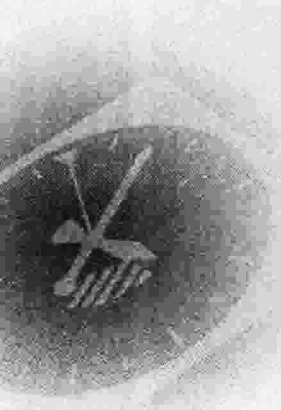
        // 删除 Dial 方法(从基类继承 Dial)

        // 移除关键字'new'，将关键字'virtual'修改为'override'，
        // 指明该方法与基类的 EstablishConnection 方法的关系
        protected override void EstablishConnection(){
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

执行相同的应用程序代码(前面列出的 **Main** 方法中的代码)，输出结果如下所示：

```
Phone.Dial
BetterPhone.EstablishConnection
```

该输出结果表明，在 **Main** 中调用 **Dial** 方法，调用的是由 **Phone** 定义、并由 **BetterPhone** 继承的 **Dial** 方法。然后，当 **Phone** 的 **Dial** 方法调用虚方法 **EstablishConnection** 时，实际调用的是 **BetterPhone** 类的 **EstablishConnection** 方法，因为它重写了由 **Phone** 定义的虚方法 **EstablishConnection**。



第 7 章　常量和字段

本章内容：

- 常量
- 字段

本章介绍如何向类型添加数据成员，具体就是常量和字段。

7.1　常量

常量是值从不变化的符号。定义常量符号时，它的值必须能在编译时确定。确定后，编译器将常量值保存到程序集元数据中。这意味着只能定义编译器识别的基元类型的常量。在 C#中，以下类型是基元类型，可用于定义常量：**Boolean**，**Char**，**Byte**，**SByte**，**Int16**，**UInt16**，**Int32**，**UInt32**，**Int64**，**UInt64**，**Single**，**Double**，**Decimal** 和 **String**。然而，C# 也允许定义非基元类型的常量变量(constant variable)，前提是把值设为 **null**：

```
using System;

public sealed class SomeType {
   // SomeType 不是基元类型，但 C#允许
   // 值为 null 的这种类型的常量变量
   public const SomeType Empty = null;
}
```

由于常量值从不变化，所以常量总是被视为类型定义的一部分。换言之，常量总是被视为静态成员，而不是实例成员。定义常量将导致创建元数据。

代码引用常量符号时，编译器在定义常量的程序集的元数据中查找该符号，提取常量的值，将值嵌入生成的 IL 代码中。由于常量的值直接嵌入代码，所以在运行时不需要为常量分配任何内存。除此之外，不能获取常量的地址，也不能以传引用的方式传递常量。这些限制意味着常量不能很好地支持跨程序集的版本控制。因此，只有确定一个符号的值从不变化才应定义常量。(将 **MaxInt16** 定义为 **32767** 就是一个很好的例子)。下面来演示我刚才所说的内容。首先，请输入以下代码，并将其编译成一个 DLL 程序集。

```
using System;

public sealed class SomeLibraryType {
   // 注意：C#不允许为常量指定 static 关键字，
```

```
    // 因为常量总是隐式为 static
    public const Int32 MaxEntriesInList = 50;
}
```

接着用以下代码生成一个应用程序程序集[①]：

```
using System;

public sealed class Program {
    public static void Main() {
        Console.WriteLine("Max entries supported in list: "
            + SomeLibraryType.MaxEntriesInList);
    }
}
```

注意代码引用了在 **SomeLibraryType** 类中定义的 **MaxEntriesInList** 常量。编译器生成应用程序代码时，会注意到 **MaxEntriesInList** 是值为 **50** 的常量符号，所以会将 **Int32** 值 **50** 嵌入应用程序的 IL 代码，如下所示。事实上，在生成了应用程序程序集之后，运行时根本不会加载 DLL 程序集，可以把它从磁盘上删除。

```
. .method public hidebysig static void Main() cil managed
{

  .entrypoint
  // Code size       25 (0x19)

  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr      "Max entries supported in list: "
  IL_0006: ldc.i4.s   50
  IL_0008: box        [mscorlib]System.Int32
  IL_000d: call       string [mscorlib]System.String::Concat(object, object)
  IL_0012: call       void [mscorlib]System.Console::WriteLine(string)
  IL_0017: nop
  IL_0018: ret
} // end of method Program::Main
```

这个例子清楚地展示了版本控制问题。如果开发人员将常量 **MaxEntriesInList** 的值更改为 **1000**，并且只是重新生成 DLL 程序集，那么应用程序程序集不受任何影响。应用程序要获得新值，也必须重新编译。如果希望在运行时从一个程序集中提取另一个程序集中的值，那么不应该使用常量，而应该使用 **readonly** 字段，详情参见下一节。

7.2 字段

字段是一种数据成员，其中容纳了一个值类型的实例或者对一个引用类型的引用。表 7-1 总结了可应用于字段的修饰符。

如表 7-1 所示，CLR 支持类型(静态)字段和实例(非静态)字段。如果是类型字段，容纳字段数据所需的动态内存是在类型对象中分配的，而类型对象是在类型加载到一个 AppDomain

① 用 csc.exe 的/r 开关来引用刚才的.dll 文件。

时创建的(参见第 22 章“CLR 寄宿和 AppDomain”)。那么，什么时候将类型加载到一个 AppDomain 中呢？这通常是在引用了该类型的任何方法首次进行 JIT 编译的时候。如果是实例字段，容纳字段数据所需的动态内存是在构造类型的实例时分配的。

表 7-1　字段修饰符

CLR 术语	C#术语	说明
Static	**static**	这种字段是类型状态的一部分，而不是对象状态的一部分
Instance	(默认)	这种字段与类型的一个实例关联，而不是与类型本身关联
InitOnly	**readonly**	这种字段只能由一个构造器方法中的代码写入
Volatile	**volatile**	编译器、CLR 和硬件不会对访问这种字段的代码执行“线程不安全”的优化措施。只有以下类型才能标记为 **volatile**：所有引用类型，**Single**，**Boolean**，**Byte**，**SByte**，**Int16**，**UInt16**，**Int32**，**UInt32**，**Char**，以及基础类型为 **Byte**，**SByte**，**Int16**，**UInt16**，**Int32** 或 **UInt32** 的所有枚举类型。**volatile** 字段将在第 29 章“基元线程同步构造”讨论[①]

由于字段存储在动态内存中，所以它们的值在运行时才能获取。字段还解决了常量存在的版本控制问题。此外，字段可以是任何数据类型，不像常量那样仅仅局限于编译器内置的基元类型。

CLR 支持 **readonly** 字段和 **read/write** 字段。大多数字段都是 **read/write** 字段，意味着在代码执行过程中，字段值可多次改变。但 **readonly** 字段只能在构造器方法中写入。(构造器方法只能调用一次，即对象首次创建时。)编译器和验证机制确保 **readonly** 字段不会被构造器以外的任何方法写入。注意，可利用反射来修改 **readonly** 字段。

现在，让我们以 7.1 节“常量”的代码为例，使用一个静态 **readonly** 字段来修正版本控制问题。下面是新版本 DLL 程序集的代码：

```
using System;

public sealed class SomeLibraryType {
   // 字段和类型关联必须使用 static 关键字
   public static readonly Int32 MaxEntriesInList = 50;
}
```

这是唯一需要修改的，应用程序的代码不必修改。但是，为了观察新的行为，必须重新生成它。当应用程序的 **Main** 方法运行时，CLR 将加载 DLL 程序集(现在运行时需要该程序集了)，并从分配给它的动态内存中提取 **MaxEntriesInList** 字段的值。当然，该值是 **50**。

① 文档将 volatile 翻译为“可变”。其实它是“短暂存在”、“易变”的意思，因为可能有多个线程想都对这种字段进行修改，所以“易变”或“易失”更佳。——译注

假设 DLL 程序集的开发人员将 **50** 改为 **1000**，并重新生成程序集。当应用程序代码重新执行时，它将自动提取字段的新值 **1000**。应用程序不需要重新生成，可以直接运行(尽管性能会受到一点影响)。要注意的是，当前假定的是 DLL 程序集的新版本没有进行强命名，而且应用程序的版本策略是让 CLR 加载这个新版本。

下例演示了如何定义一个与类型本身关联的 **readonly** 静态字段。还定义了 **read/write** 静态字段，以及 **readonly** 和 **read/write** 实例字段。

```
public sealed class SomeType {
   // 这是一个静态 readonly 字段：在运行时对这个类进行初始化时，
   // 它的值会被计算并存储到内存中
   public static readonly Random s_random = new Random();

   // 这是一个静态 read/write 字段
   private static Int32 s_numberOfWrites = 0;

   // 这是一个实例 readonly 字段
   public readonly String Pathname = "Untitled";

   // 这是一个实例 read/write 字段
   private System.IO.FileStream m_fs;

   public SomeType(String pathname) {
      // 该行修改只读字段 pathname，
      //在构造器中可以这样做
      this.Pathname = pathname;
   }

   public String DoSomething() {
      // 该行读写静态 read/write 字段
      s_numberOfWrites = s_numberOfWrites + 1;

      // 该行读取 readonly 实例字段
      return Pathname;
   }
}
```

在上述代码中，许多字段都是内联初始化的[①]。C#允许使用这种简便的内联初始化语法来初始化类的常量、**read/write** 字段和 **readonly** 字段。第 8 章“方法”会讲到，C#实际是在构造器中对字段进行初始化的，字段的内联初始化只是一种语法上的简化。另外，在 C#中初始化字段时，如果使用内联语法，而不是在构造器中赋值，有一些性能问题需要考虑。这些性能问题也将在第 8 章讨论。

① 内联(inline)初始化是指在代码中直接赋值来初始化，而不是将对构造器的调用写出来。——译注

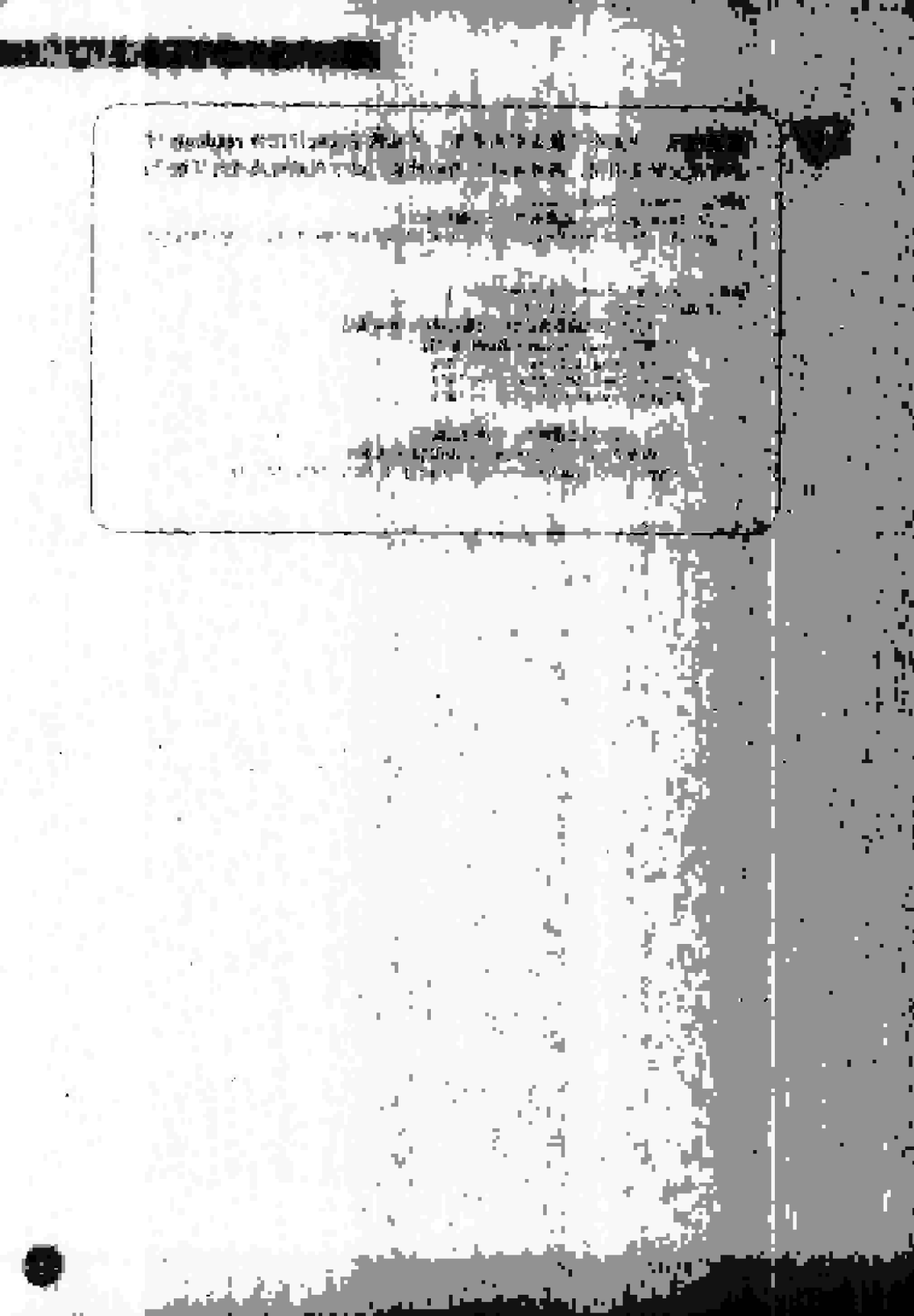


重要提示　当某个字段是引用类型，并且该字段被标记为 **readonly** 时，不可改变的是引用，而非字段引用的对象。以下代码对此进行了演示：

```
public sealed class AType {
    // InvalidChars 总是引用同一个数组对象
    public static readonly Char[]InvalidChars = new Char[] {'A','B','C'};
}

public sealed class AnotherType {
    public static void M() {
        // 下面三行代码是合法的，可通过编译，并可成功
        // 修改 InvalidChars 数组中的字符
        AType.InvalidChars[0] = 'X';
        AType.InvalidChars[1] = 'Y';
        AType.InvalidChars[2] = 'Z';

        // 下一行代码是非法的，无法通过编译，
        // 因为不能让 InvalidChars 引用别的什么东西
        AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };
    }
}
```



第8章 方 法

本章内容：

- 实例构造器和类(引用类型)
- 实例构造器和结构(值类型)
- 类型构造器
- 操作符重载方法
- 转换操作符方法
- 扩展方法
- 分部方法

本章重点讨论你将来可能遇到的各种方法，包括实例构造器和类型构造器。还会讲述如何定义方法来重载操作符和类型转换(以进行隐式和显式转型)。还会讨论扩展方法，以便将自己的实例方法在逻辑上“添加”到现有类型中。还会讨论分部方法，允许将类型的实现分散到多个组成部分中。

8.1 实例构造器和类(引用类型)

构造器是将类型的实例初始化为良好状态的特殊方法。构造器方法在“方法定义元数据表”中始终叫做**.ctor**(constructor 的简称)。创建引用类型的实例时，首先为实例的数据字段分配内存，然后初始化对象的附加字段(类型对象指针和同步块索引)[①]，最后调用类型的实例构造器来设置对象的初始状态。

构造引用类型的对象时，在调用类型的实例构造器之前，为对象分配的内存总是先被归零。没有被构造器显式重写的所有字段都保证获得 **0** 或 **null** 值。

和其他方法不同，实例构造器永远不能被继承。也就是说，类只有类自己定义的实例构造器。由于永远不能继承实例构造器，所以实例构造器不能使用以下修饰符：**virtual**，**new**，**override**，**sealed** 和 **abstract**。如果类没有显式定义任何构造器，C#编译器将定义一个默认

① 这些附加的字段称为 overhead fields，“overhead”是开销的意思，意味着是创建对象时必须的“开销”。——译注

(无参)构造器。在它的实现中，只是简单地调用了基类的无参构造器。

例如下面这个类：

```
public class SomeType {
}
```

它等价于：

```
public class SomeType {
   public SomeType() : base() { }
}
```

如果类的修饰符为 **abstract**，那么编译器生成的默认构造器的可访问性就为 **protected**；否则，构造器会被赋予 **public** 可访问性。如果基类没有提供无参构造器，那么派生类必须显式调用一个基类构造器，否则编译器会报错。如果类的修饰符为 **static(sealed** 和 **abstract)**[①]，编译器根本不会在类的定义中生成默认构造器。

一个类型可以定义多个实例构造器。每个构造器都必须有不同的签名，而且每个都可以有不同的可访问性。为了使代码“可验证”(verifiable)，类的实例构造器在访问从基类继承的任何字段之前，必须先调用基类的构造器。如果派生类的构造器没有显式调用一个基类构造器，C#编译器会自动生成对默认的基类构造器的调用。最终，**System.Object** 的公共无参构造器会得到调用。该构造器什么都不做，会直接返回。由于 **System.Object** 没有定义实例数据字段，所以它的构造器无事可做。

极少数时候可以在不调用实例构造器的前提下创建类型的实例。一个典型的例子是 **Object** 的 **MemberwiseClone** 方法。该方法的作用是分配内存，初始化对象的附加字段(类型对象指针和同步块索引)，然后将源对象的字节数据复制到新对象中。另外，用运行时序列化器(runtime serializer)反序列化对象时，通常也不需要调用构造器。反序列化代码使用 **System.Runtime.Serialization.FormatterServices** 类型的 **GetUninitializedObject** 或者 **GetSafeUninitializedObject** 方法为对象分配内存，期间不会调用一个构造器。详情参见第 24 章“运行时序列化”。



重要提示 不要在构造器中调用虚方法。原因是假如被实例化的类型重写了虚方法，就会执行派生类型对虚方法的实现。但在这个时候，尚未完成对继承层次结构中的所有字段的初始化（被实例化的类型的构造器还没有运行呢）。所以，调用虚方法会导致无法预测的行为。归根到底，这是由于调用虚方法时，直到运行时之前都不会选择执行该方法的实际类型。

C#语言用简单的语法在构造引用类型的实例时初始化类型中定义的字段：

```
internal sealed class SomeType {
   private Int32 m_x = 5;
```

① 静态类在元数据中是抽象密封类。——译注

```
}
```

构造 **SomeType** 的对象时，它的 **m_x** 字段被初始化为 **5**。这是如何发生的呢？检查一下 **SomeType** 的构造器方法(也称作**.ctor**)的 IL 代码就明白了，如下所示：

```
.method public hidebysig specialname rtspecialname
        instance void .ctor() cil managed
{
  // Code size 14 (0xe)
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: ldc.i4.5
  IL_0002: stfld  int32    SomeType::m_x
  IL_0007: ldarg.0
  IL_0008: call            instance void [mscorlib]System.Object::.ctor()
  IL_000d: ret
} // end of method SomeType::.ctor
```

可以看出，**SomeType** 的构造器把值 **5** 存储到字段 **m_x**，再调用基类的构造器。换句话说，C#编译器提供了一个简化的语法，允许以“内联”(其实就是嵌入)方式初始化实例字段。但在幕后，它会将这种语法转换成构造器方法中的代码来执行初始化。这同时提醒我们注意代码的膨胀效应。如以下类定义所示：

```
internal sealed class SomeType {
   private Int32      m_x = 5;
   private String     m_s = "Hi there";
   private Double     m_d = 3.14159;
   private Byte       m_b;

   // 下面是一些构造器
   public SomeType()           {...}
   public SomeType(Int32 x)    {...}
   public SomeType(String s)   {...; m_d=10;}
}
```

编译器为这三个构造器方法生成代码时，在每个方法的开始位置，都会包含用于初始化 **m_x**，**m_s** 和 **m_d** 的代码。在这些初始化代码之后，编译器会插入对基类构造器的调用。再然后，会插入构造器自己的代码。例如，对于获取一个 **String** 参数的构造器，编译器生成的代码首先初始化 **m_x**，**m_s** 和 **m_d**，再调用基类(**Object**)的构造器，再执行自己的代码(最后是用值 **10** 覆盖 **m_d** 原先的值)。注意，即使没有代码显式初始化 **m_b**，**m_b** 也保证会被初始化为 **0**。



注意　编译器在调用基类构造器前使用简化语法对所有字段进行初始化，以维持源代码给人留下的“这些字段总是有一个值”的印象。但假如基类构造器调用了虚方法并回调由派生类定义的方法，就可能出问题。在这种情况下，使用简化语法初始化的字段在调用虚方法之前就初始化好了。

由于有三个构造器，所以编译器生成三次初始化 **m_x**，**m_s** 和 **m_d** 的代码——每个构造器一次。如果有几个已初始化的实例字段和许多重载的构造器方法，可考虑不是在定义字段时初始化，而是创建单个构造器来执行这些公共的初始化。然后，让其他构造器都显式