

- 数组字段必须用 **fixed** 关键字标记。
- 数组必须是一维 0 基数组。
- 数组的元素类型必须是以下类型之一：**Boolean**, **Char**, **SByte**, **Byte**, **Int16**, **Int32**, **UInt16**, **UInt32**, **Int64**, **UInt64**, **Single** 或 **Double**。

要和非托管代码进行互操作，而且非托管数据结构也有一个内联数组，就特别适合使用内联的数组。但内联数组也能用于其他地方。上述代码中的 **InlineArrayDemo** 方法提供了如何使用内联数组的一个例子。它执行和 **StackallocDemo** 方法一样的功能，只是用了不一样的方式。

第 17 章 委 托

本章内容：

- 初识委托
- 用委托回调静态方法
- 用委托回调实例方法
- 委托揭秘
- 用委托回调许多方法(委托链)
- 委托定义不要太多(泛型委托)
- C#为委托提供的简化语法
- 委托和反射

本章要讨论回调函数。回调函数是一种非常有用的编程机制，它的存在已经有很多年了。Microsoft .NET Framework 通过委托来提供回调函数机制。不同于其他平台(比如非托管 C++) 的回调机制，委托的功能要多得多。例如，委托确保回调方法是类型安全的(这是 CLR 最重要的目标之一)。委托还允许顺序调用多个方法，并支持调用静态方法和实例方法。

17.1 初识委托

C “运行时”的 `qsort` 函数获取指向一个回调函数的指针，以便对数组中的元素进行排序。在 Microsoft Windows 中，窗口过程、钩子过程和异步过程调用等都需要回调函数。在 .NET Framework 中，回调方法的应用更是广泛。例如，可以登记回调方法来获得各种各样的通知，例如未处理的异常、窗口状态变化、菜单项选择、文件系统变化、窗体控件事件和异步操作已完成等。

在非托管 C/C++ 中，非成员函数的地址只是一个内存地址。这个地址不携带任何额外的信息，比如函数期望收到的参数个数、参数类型、函数返回值类型以及函数的调用协定。简单地说，非托管 C/C++ 回调函数不是类型安全的(不过它们确实是一种非常轻量级的机制)。

.NET Framework 的回调函数和非托管 Windows 编程环境的回调函数一样有用，一样普遍。但是，.NET Framework 提供了称为委托的类型安全机制。为了理解委托，先来看看如何使

用它。以下代码^①演示了如何声明、创建和使用委托：

```
using System;
using System.Windows.Forms;
using System.IO;

// 声明一个委托类型，它的实例引用一个方法，
// 该方法获取一个 Int32 参数，返回 void
internal delegate void Feedback(Int32 value);

public sealed class Program {
    public static void Main() {
        StaticDelegateDemo();
        InstanceDelegateDemo();
        ChainDelegateDemo1(new Program());
        ChainDelegateDemo2(new Program());
    }

    private static void StaticDelegateDemo() {
        Console.WriteLine("----- Static Delegate Demo -----");
        Counter(1, 3, null);
        Counter(1, 3, new Feedback(Program.FeedbackToConsole));
        Counter(1, 3, new Feedback(FeedbackToMsgBox)); // 前缀"Program."可选
        Console.WriteLine();
    }

    private static void InstanceDelegateDemo() {
        Console.WriteLine("----- Instance Delegate Demo -----");
        Program p = new Program();
        Counter(1, 3, new Feedback(p.FeedbackToFile));
        Console.WriteLine();
    }

    private static void ChainDelegateDemo1(Program p) {
        Console.WriteLine("----- Chain Delegate Demo 1 -----");
        Feedback fb1 = new Feedback(FeedbackToConsole);
        Feedback fb2 = new Feedback(FeedbackToMsgBox);
        Feedback fb3 = new Feedback(p.FeedbackToFile);

        Feedback fbChain = null;
        fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
        fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
        fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
        Counter(1, 2, fbChain);

        Console.WriteLine();
        fbChain = (Feedback)
            Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
        Counter(1, 2, fbChain);
    }

    private static void ChainDelegateDemo2(Program p) {
        Console.WriteLine("----- Chain Delegate Demo 2 -----");
        Feedback fb1 = new Feedback(FeedbackToConsole);
        Feedback fb2 = new Feedback(FeedbackToMsgBox);
        Feedback fb3 = new Feedback(p.FeedbackToFile);

        Feedback fbChain = null;
        fbChain += fb1;
```

① 这个程序最好不要通过在 Visual Studio 中新建“Windows 窗体应用程序”项目来生成。用文本编辑器输入代码，另存为 *name.cs*。启动“VS2013 开发人员命令提示”，输入 **csc name.cs** 生成，输入 *name* 执行。这样可同时看到控制台和消息框的输出。——译注

```

        fbChain += fb2;
        fbChain += fb3;
        Counter(1, 2, fbChain);

        Console.WriteLine();
        fbChain -= new Feedback(FeedbackToMsgBox);
        Counter(1, 2, fbChain);
    }

    private static void Counter(Int32 from, Int32 to, Feedback fb) {
        for (Int32 val = from; val <= to; val++) {
            // 如果指定了任何回调, 就调用它们
            if (fb != null)
                fb(val);
        }
    }

    private static void FeedbackToConsole(Int32 value) {
        Console.WriteLine("Item=" + value);
    }

    private static void FeedbackToMsgBox(Int32 value) {
        MessageBox.Show("Item=" + value);
    }

    private void FeedbackToFile(Int32 value) {
        using (StreamWriter sw = new StreamWriter("Status", true)) {
            sw.WriteLine("Item=" + value);
        }
    }
}

```

下面来看看代码做的事情。在顶部, 注意看 **internal** 委托 **Feedback** 的声明。委托要指定一个回调方法签名。在本例中, **Feedback** 委托指定的方法要获取一个 **Int32** 参数, 返回 **void**。在某种程度上, 委托和非托管 C/C++ 中代表函数地址的 **typedef** 很相似。

Program 类定义了私有静态方法 **Counter**, 它从整数 **from** 计数到整数 **to**。方法的 **fb** 参数代表 **Feedback** 委托对象引用。方法遍历所有整数。对于每个整数, 如果 **fb** 变量不为 **null**, 就调用由 **fb** 变量指定的回调方法。传入这个回调方法的是正在处理的那个数据项的值, 也就是数据项的编号。设计和实现回调方法时, 可选择任何恰当的方式处理数据项。

17.2 用委托回调静态方法

理解 **Counter** 方法的设计及其工作方式之后, 再来看看如何利用委托回调静态方法。本节重点是上一节示例代码中的 **StaticDelegateDemo** 方法。

在 **StaticDelegateDemo** 方法中第一次调用 **Counter** 方法时, 为第三个参数(对应于 **Counter** 的 **fb** 参数)传递的是 **null**。由于 **Counter** 的 **fb** 参数收到的是 **null**, 所以处理每个数据项时都不调用回调方法。

StaticDelegateDemo 方法再次调用 **Counter**, 为第三个参数传递新构造的 **Feedback** 委托对象。委托对象是方法的包装器(wrapper), 使方法能通过包装器来间接回调。在本例中, 静态方法的完整名称 **Program.FeedbackToConsole** 被传给 **Feedback** 委托类型的构造器, 这就是要包装的方法。 **new** 操作符返回的引用作为 **Counter** 的第三个参数来传递。现在, 当

Counter 执行时，会为序列中的每个数据项调用 **Program** 类型的静态方法 **FeedbackToConsole**。**FeedbackToConsole** 方法本身的作用很简单，就是向控制台写一个字符串，显示正在进行处理的数据项。



注意 **FeedbackToConsole** 方法被定义成 **Program** 类型内部的私有方法，但 **Counter** 方法能调用 **Program** 的私有方法。这明显没有问题，因为 **Counter** 和 **FeedbackToConsole** 在同一个类型中定义。但即使 **Counter** 方法在另一个类型中定义，也不会出问题！简单地说，在一个类型中通过委托来调用另一个类型的私有成员，只要委托对象是由具有足够安全性/可访问性的代码创建的，便没有问题。

在 **StaticDelegateDemo** 方法中，对 **Counter** 方法的第三个调用和第二个调用几乎完全一致。唯一的区别在于 **Feedback** 委托对象包装的是静态方法 **Program.FeedbackToMsgBox**。**FeedbackToMsgBox** 构造一个字符串来指出正在处理的数据项，然后在消息框中显示该字符串。

这个例子中的所有操作都是类型安全的。例如，在构造 **Feedback** 委托对象时，编译器确保 **Program** 的 **FeedbackToConsole** 和 **FeedbackToMsgBox** 方法的签名兼容于 **Feedback** 委托定义的签名。具体地说，两个方法都要获取一个参数(一个 **Int32**)，而且两者都要有相同的返回类型(**void**)。将 **FeedbackToConsole** 的定义改为下面这样：

```
private static Boolean FeedbackToConsole(String value) {
    ...
}
```

C#编译器将不会编译以上代码，并报告以下错误：**error CS0123: "FeedbackToConsole"的重载均与委托"Feedback"不匹配。**

将方法绑定到委托时，C#和 CLR 都允许引用类型的协变性(covariance)和逆变性(contravariance)。协变性是指方法能返回从委托的返回类型派生的一个类型。逆变性是指方法获取的参数可以是委托的参数类型的基类。例如下面这个委托：

```
delegate Object MyCallback(FileStream s);
```

完全可以构造该委托类型的一个实例并绑定具有以下原型的方法：

```
String SomeMethod(Stream s);
```

在这里，**SomeMethod** 的返回类型(**String**)派生自委托的返回类型(**Object**)；这种协变性是允许的。**SomeMethod** 的参数类型(**Stream**)是委托的参数类型(**FileStream**)的基类；这种逆变性是允许的。

注意，只有引用类型才支持协变性与逆变性，值类型或 **void** 不支持。所以，不能把下面的方法绑定到 **MyCallback** 委托：

```
Int32 SomeOtherMethod(Stream s);
```

虽然 `SomeOtherMethod` 的返回类型(`Int32`)派生自 `MyCallback` 的返回类型(`Object`)，但这种形式的协变性是不允许的，因为 `Int32` 是值类型。显然，值类型和 `void` 之所以不支持，是因为它们的存储结构是变化的，而引用类型的存储结构始终是一个指针。幸好，试图执行不支持的操作，C#编译器会报错。

17.3 用委托回调实例方法

委托除了能调用静态方法，还能为具体的对象调用实例方法。为了理解如何回调实例方法，先来看看 17.1 节的示例代码中的 `InstanceDelegateDemo` 方法。

注意 `InstanceDelegateDemo` 方法构造了名为 `p` 的 `Program` 对象。这个 `Program` 对象没有定义任何实例字段或属性；创建它纯粹是为了演示。在 `Counter` 方法调用中构造新的 `Feedback` 委托对象时，向 `Feedback` 委托类型的构造函数传递的是 `p.FeedbackToFile`。这导致委托包装对 `FeedbackToFile` 方法的引用，这是一个实例方法(而不是静态方法)。当 `Counter` 调用由其 `fb` 实参标识的回调方法时，会调用 `FeedbackToFile` 实例方法，新构造的对象 `p` 的地址作为隐式的 `this` 参数传给这个实例方法。

`FeedbackToFile` 方法的工作方式类似于 `FeedbackToConsole` 和 `FeedbackToMsgBox`，不同的是它会打开一个文件，并将字符串附加到文件末尾。(方法创建的 `Status` 文件可在与可执行程序相同的目录中找到。)

再次声明，本例旨在演示委托可以包装对实例方法和静态方法的调用。如果是实例方法，委托要知道方法操作的是具体哪个对象实例。包装实例方法很有用，因为对象内部的代码可以访问对象的实例成员。这意味着对象可以维护一些状态，并在回调方法执行期间利用这些状态信息。

17.4 委托揭秘

从表面看，委托似乎很容易使用：用 C# 的 `delegate` 关键字定义，用熟悉的 `new` 操作符构造委托实例，用熟悉的方法调用语法来调用回调函数(用引用了委托对象的变量替代方法名)。

但实际情况比前几个例子演示的要复杂一些。编译器和 CLR 在幕后做了大量工作来隐藏复杂性。本节要解释编译器和 CLR 如何协同工作来实现委托。掌握这些知识有助于加深对委托的理解，并学会如何更高效地使用。另外，还要介绍通过委托来实现的一些附加功能。

首先重新审视这一行代码：

```
internal delegate void Feedback(Int32 value);
```

看到这行代码后，编译器实际会像下面这样定义一个完整的类：

```
internal class Feedback : System.MulticastDelegate {
    // 构造器
    public Feedback(Object @object, IntPtr method);
}
```

```
// 这个方法的原型和源代码指定的一样
public virtual void Invoke(Int32 value);

// 以下方法实现对回调方法的异步回调
public virtual IAsyncResult BeginInvoke(Int32 value,
    AsyncCallback callback, Object @object);
public virtual void EndInvoke(IAsyncResult result);
}
```

编译器定义的类有 4 个方法：一个构造器、**Invoke**、**BeginInvoke** 和 **EndInvoke**。本章重点解释构造器和 **Invoke**。**BeginInvoke** 和 **EndInvoke** 方法将留到第 27 章讨论。

事实上，可用 ILDasm.exe 查看生成的程序集，验证编译器真的会自动生成这个类，如图 17-1 所示。

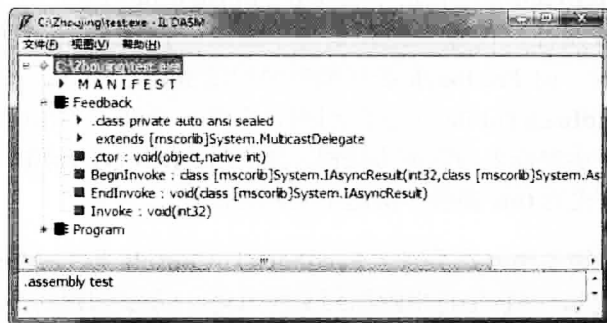


图 17-1 ILDasm.exe 显示了编译器为委托生成的元数据

在本例中，编译器定义了 **Feedback** 类，它派生自 FCL 定义的 **System.MulticastDelegate** 类型(所有委托类型都派生自 **MulticastDelegate**)。



重要提示 **System.MulticastDelegate** 派生自 **System.Delegate**，后者又派生自 **System.Object**。是历史原因造成有两个委托类。这实在是令人遗憾——FCL 本该只有一个委托类。没有办法，我们对这两个类都要有所了解。即使创建的所有委托类型都将 **MulticastDelegate** 作为基类，个别情况下仍会使用 **Delegate** 类(而非 **MulticastDelegate** 类)定义的方法处理自己的委托类型。例如，**Delegate** 类的两个静态方法 **Combine** 和 **Remove**(后文将解释其用途)的签名都指出要获取 **Delegate** 参数。由于你创建的委托类型派生自 **MulticastDelegate**，后者又派生自 **Delegate**，所以你的委托类型的实例是可以传给这两个方法的。

这个类的可访问性是 **private**，因为委托在源代码中声明为 **internal**。如果源代码改成使用 **public** 可见性，编译器生成的 **Feedback** 类也会变成公共类。要注意的是，委托类既可嵌套在一个类型中定义，也可在全局范围中定义。简单地说，由于委托是类，所以凡是能够定义类的地方，都能定义委托。

由于所有委托类型都派生自 **MulticastDelegate**，所以它们继承了 **MulticastDelegate** 的字段、

属性和方法。在所有这些成员中，有三个非公共字段是最重要的。表 17-1 总结了这些重要字段。

表 17-1 MulticastDelegate 的三个重要的非公共字段

字段	类型	说明
<code>_target</code>	<code>System.Object</code>	当委托对象包装一个静态方法时，这个字段为 <code>null</code> 。当委托对象包装一个实例方法时，这个字段引用的是回调方法要操作的对象。换言之，这个字段指出要传给实例方法的隐式参数 <code>this</code> 的值
<code>_methodPtr</code>	<code>System.IntPtr</code>	一个内部的整数值，CLR 用它标识要回调的方法
<code>_invocationList</code>	<code>System.Object</code>	该字段通常为 <code>null</code> 。构造委托链时它引用一个委托数组(详情参见下一节)

注意，所有委托都有一个构造器，它获取两个参数：一个是对象引用，另一个是引用了回调方法的整数。但如果仔细查看前面的源代码，会发现传递的是 `Program.FeedbackToConsole` 或 `p.FeedbackToFile` 这样的值。根据迄今为止学到的编程知识，似乎没有可能通过编译！

然而，C#编译器知道要构造的是委托，所以会分析源代码来确定引用的是哪个对象和方法。对象引用被传给构造器的 `object` 参数，标识了方法的一个特殊 `IntPtr` 值(从 `MethodDef` 或 `MemberRef` 元数据 token 获得)被传给构造器的 `method` 参数。对于静态方法，会为 `object` 参数传递 `null` 值。在构造器内部，这两个实参分别保存在 `_target` 和 `_methodPtr` 私有字段中。除此以外，构造器还将 `_invocationList` 字段设为 `null`，对这个字段的讨论将推迟到 17.5 节“用委托回调多个方法(委托链)”进行。

所以，每个委托对象实际都是一个包装器，其中包装了一个方法和调用该方法时要操作的对象。例如，在执行以下两行代码之后：

```
Feedback fbStatic = new Feedback(Program.FeedbackToConsole);
Feedback fbInstance = new Feedback(new Program().FeedbackToFile);
```

`fbStatic` 和 `fbInstance` 变量将引用两个独立的、初始化好的 `Feedback` 委托对象，如图 17-2 所示。

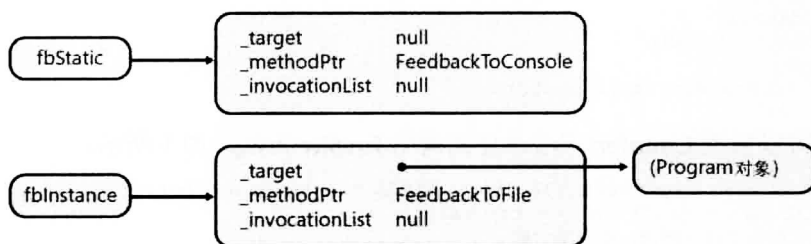


图 17-2 在两个变量引用的委托中，一个包装静态方法，另一个包装实例方法

知道委托对象如何构造并了解其内部结构之后，再来看看回调方法是如何调用的。为方便讨论，下面重复了 `Counter` 方法的定义：

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
```

```

    for (Int32 val = from; val <= to; val++) {
        // 如果指定了任何回调, 就调用它们
        if (fb != null)
            fb(val);
    }
}

```

注意 **if** 语句首先检查 **fb** 是否为 **null**。不为 **null** 就调用^①回调方法。**null** 检查必不可少, 因为 **fb** 只是可能引用了 **Feedback** 委托对象的变量; 它也可能为 **null**。这段代码看上去像是调用了一个名为 **fb** 的函数, 并向它传递一个参数(**val**)。但事实上, 这里没有名为 **fb** 的函数。再次提醒你注意, 因为编译器知道 **fb** 是引用了委托对象的变量, 所以会生成代码调用该委托对象的 **Invoke** 方法。也就是说, 编译器在看到以下代码时:

```
fb(val);
```

它将生成以下代码, 好像源代码本来就是这么写的一样:

```
fb.Invoke(val);
```

为了验证编译器生成代码来调用委托类型的 **Invoke** 方法, 可利用 **ILDasm.exe** 检查为 **Counter** 方法创建的 IL 代码。下面列出了 **Counter** 方法的 IL 代码。IL_0009 处的指令就是对 **Feedback** 的 **Invoke** 方法的调用。

```

.method private hidebysig static void Counter( int32 from,
                                                int32 'to',
                                                class Feedback fb) cil managed
{
    // Code size 23 (0x17)
    .maxstack 2
    .locals init (int32 val)
    IL_0000: ldarg.0
    IL_0001: stloc.0
    IL_0002: br.s IL_0012
    IL_0004: ldarg.2
    IL_0005: brfalse.s IL_000e
    IL_0007: ldarg.2
    IL_0008: ldloc.0
    IL_0009: callvirt instance void Feedback::Invoke(int32)
    IL_000e: ldloc.0
    IL_000f: ldc.i4.1
    IL_0010: add
    IL_0011: stloc.0
    IL_0012: ldloc.0
    IL_0013: ldarg.1
    IL_0014: ble.s IL_0004
    IL_0016: ret
} // end of method Program::Counter

```

其实, 完全可以修改 **Counter** 方法来显式调用 **Invoke** 方法, 如下所示:

```

private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // 如果指定了任何回调, 就调用它们
        if (fb != null)
            fb.Invoke(val);
    }
}

```

① 这里的“调用”是 **invoke**, 参考 8.6.2 节的译注对 **invoke** 和 **call** 的解释。——译注

前面说过, 编译器是在定义 **Feedback** 类的时候定义 **Invoke** 的。在 **Invoke** 被调用时, 它使用私有字段 **_target** 和 **_methodPtr** 在指定对象上调用包装好的回调方法。注意, **Invoke** 方法的签名和委托的签名匹配。由于 **Feedback** 委托要获取一个 **Int32** 参数并返回 **void**, 所以编译器生成的 **Invoke** 方法也要获取一个 **Int32** 并返回 **void**。

17.5 用委托回调多个方法(委托链)

委托本身就很有用, 再加上对委托链的支持, 用处就更大了! 委托链是委托对象的集合。可利用委托链调用集中的委托所代表的全部方法。为了理解这一点, 请参考 17.1 节的示例代码中的 **ChainDelegateDemo1** 方法。在 **Console.WriteLine** 语句之后, 我构造了三个委托对象并让变量 **fb1**, **fb2** 和 **fb3** 分别引用每个对象, 如图 17-3 所示。

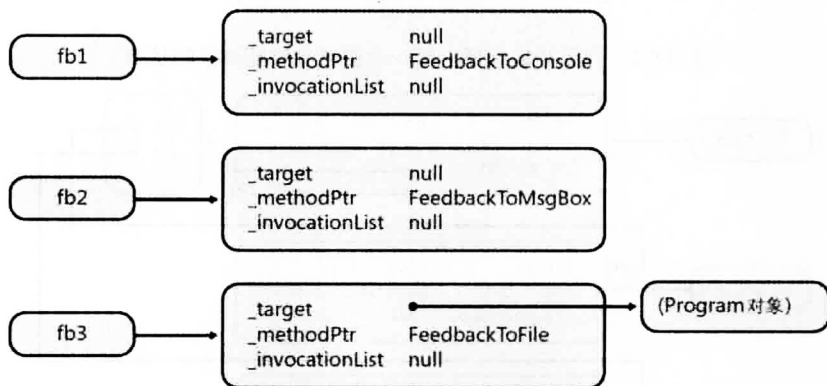


图 17-3 fb1, fb2 和 fb3 变量引用的委托对象的初始状态

指向 **Feedback** 委托对象的引用变量 **fbChain** 旨在引用委托链(或者说委托对象集合), 这些对象包装了可回调的方法。**fbChain** 初始化为 **null**, 表明目前没有要回调的方法。使用 **Delegate** 类的公共静态方法 **Combine** 将委托添加到链中:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
```

执行这行代码时, **Combine** 方法发现试图合并的是 **null** 和 **fb1**。在内部, **Combine** 直接返回 **fb1** 中的值, 所以 **fbChain** 变量现在引用 **fb1** 变量所引用的委托对象, 如图 17-4 所示。

再次调用 **Combine** 方法在链中添加第二个委托:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
```

在内部, **Combine** 方法发现 **fbChain** 已引用了一个委托对象, 所以 **Combine** 会构造一个新的委托对象。新委托对象对它的私有字段 **_target** 和 **_methodPtr** 进行初始化, 具体的值对于目前的讨论来说并不重要。重要的是, **_invocationList** 字段被初始化为引用一个委托对象数组。数组的第一个元素(索引 0)被初始化为引用包装了 **FeedbackToConsole** 方法的委托(也就是 **fbChain** 目前引用的委托)。数组的第二个元素(索引 1)被初始化为引用包装了 **FeedbackToMsgBox** 方法的委托(也就是 **fb2** 引用的委托)。最后, **fbChain** 被设为引用新建

的委托对象，如图 17-5 所示。

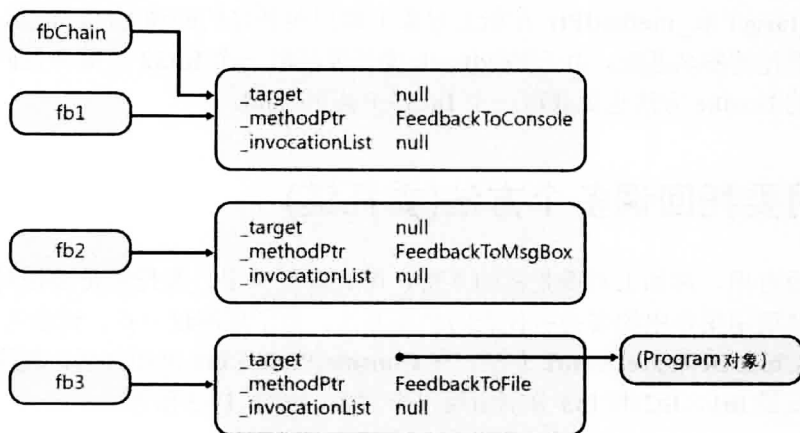


图 17-4 在委托链中插入第一个委托后委托对象的状态

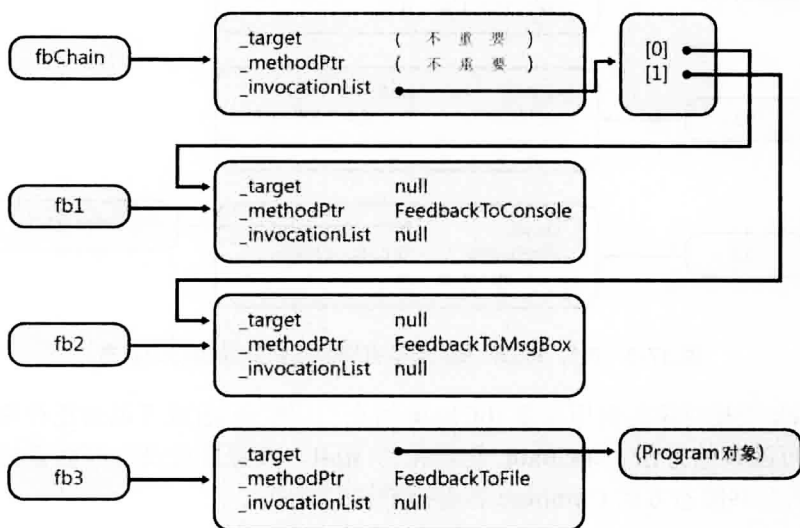


图 17-5 在委托链中插入第二个委托之后委托对象的状态

为了在链中添加第三个委托，我再次调用 **Combine** 方法。

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
```

同样地，**Combine** 方法发现 **fbChain** 已引用了一个委托对象，因而又构造一个新的委托对象，如图 17-6 所示。和前面一样，新委托对象对私有字段 **_target** 和 **_methodPtr** 进行初始化，具体的值就目前来说并不重要。**_invocationList** 字段被初始化为引用一个委托对象数组。该数组的第一个元素和第二个元素(索引 0 和 1)被初始化为引用 **fb1** 和 **fb2** 所引用的委托。数组的第三个元素(索引 2)被初始化为引用包装了 **FeedbackToFile** 方法的委托(这是 **fb3** 所引用的委托)。最后，**fbChain** 被设为引用这个新建的委托对象。注意，之前新建的委托及其 **_invocationList** 字段引用的数组现在可以进行垃圾回收。

在 **ChainDelegateDemo1** 方法中，用于设置委托链的所有代码执行完毕之后，我将 **fbChain**

变量传给 **Counter** 方法:

```
Counter(1, 2, fbChain);
```

Counter 方法内部的代码会在 **Feedback** 委托对象上隐式调用 **Invoke** 方法，具体已在前面讲述过了。在 **fbChain** 引用的委托上调用 **Invoke** 时，该委托发现私有字段 **_invocationList** 不为 **null**，所以会执行一个循环来遍历数组中的所有元素，并依次调用每个委托包装的方法。在本例中，**FeedbackToConsole** 首先被调用，随后是 **FeedbackToMsgBox**，最后是 **FeedbackToFile**。

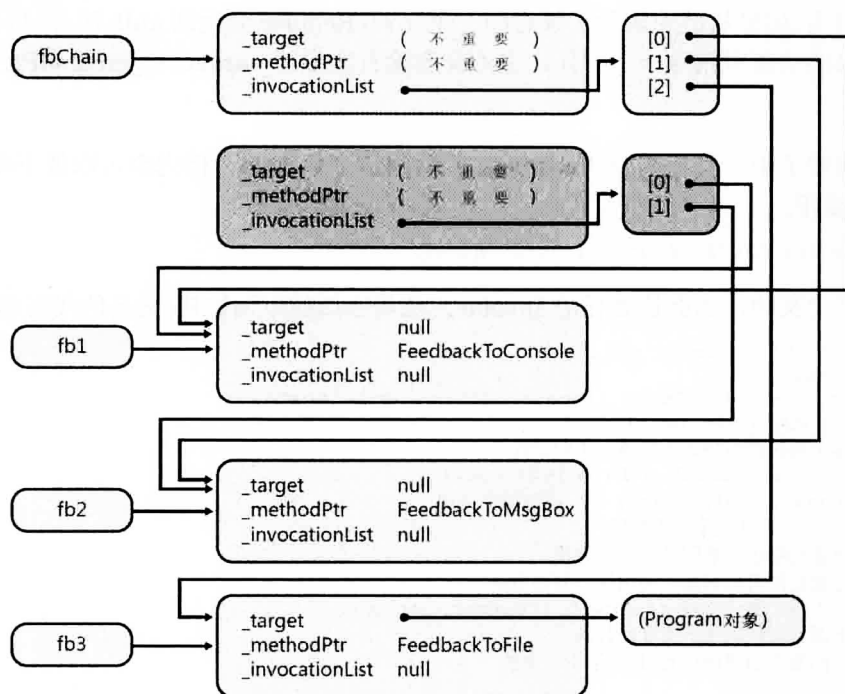


图 17-6 委托链完成后委托对象的最终状态

以伪代码的形式，**Feedback** 的 **Invoke** 方法基本上是像下面这样实现的：

```
public void Invoke(Int32 value) {
    Delegate[] delegateSet = _invocationList as Delegate[];
    if (delegateSet != null) {
        // 这个委托数组指定了应该调用的委托
        foreach (Feedback d in delegateSet)
            d(value); // 调用每个委托
    } else { // 否则就不是委托链
        // 该委托标识了要回调的单个方法，
        // 在指定的目标对象上调用这个回调方法
        _methodPtr.Invoke(_target, value);
        // 上面这行代码接近实际的代码，
        // 实际发生的事情用 C# 是表示不出来的
    }
}
```

注意，还可调用 **Delegate** 的公共静态方法 **Remove** 从链中删除委托。**ChainDelegateDemo1** 方法在结尾处对此进行了演示。

```
fbChain = (Feedback) Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
```

Remove 方法被调用时，它扫描第一个实参(本例是 **fbChain**)所引用的那个委托对象内部维护的委托数组(从末尾向索引 0 扫描)。**Remove** 查找的是其 **_target** 和 **_methodPtr** 字段与第二个实参(本例是新建的 **Feedback** 委托)中的字段匹配的委托。如果找到匹配的委托，并且(在删除之后)数组中只剩余一个数据项，就返回那个数据项。如果找到匹配的委托，并且数组中还剩余多个数据项，就新建一个委托对象——其中创建并初始化的 **_invocationList** 数组将引用原始数组中的所有数据项，当然被删除的数据项除外——并返回对这个新建委托对象的引用。如果从链中删除了仅有的一个元素，**Remove** 会返回 **null**。注意，每次 **Remove** 方法调用只能从链中删除一个委托，它不会删除有匹配的 **_target** 和 **_methodPtr** 字段的所有委托。

前面展示的例子中，委托类型 **Feedback** 的返回值都是 **void**。但完全可以像下面这样定义 **Feedback** 委托：

```
public delegate Int32 Feedback(Int32 value);
```

如果是这样定义的，那么该委托的 **Invoke** 方法就应该像下面这样(又是伪代码形式)：

```
public Int32 Invoke(Int32 value) {
    Int32 result;
    Delegate[] delegateSet = _invocationList as Delegate[];
    if (delegateSet != null) {
        // 这个委托数组指定了应该调用的委托
        foreach (Feedback d in delegateSet)
            result = d(value); // 调用每个委托
    } else { // 否则就不是委托链
        // 该委托标识了要回调的单个方法，
        // 在指定的目标对象上调用这个回调方法
        result = _methodPtr.Invoke(_target, value);
        // 上面这行代码接近实际的代码
        // 实际发生的事情用 C# 是表示不出来的
    }
    return result;
}
```

数组中的每个委托被调用时，其返回值被保存到 **result** 变量中。循环完成后，**result** 变量只包含调用的最后一个委托的结果(前面的返回值会被丢弃)，该值返回给调用 **Invoke** 的代码。

17.5.1 C#对委托链的支持

为方便 C# 开发人员，C# 编译器自动为委托类型的实例重载了 **+=** 和 **-=** 操作符。这些操作符分别调用 **Delegate.Combine** 和 **Delegate.Remove**。可用这些操作符简化委托链的构造。在 17.1 节的示例代码中，**ChainDelegateDemo1** 和 **ChainDelegateDemo2** 方法生成的 IL 代码完全一样。唯一的区别是 **ChainDelegateDemo2** 方法利用 C# 的 **+=** 和 **-=** 操作符简化了源代码。

要想证明两个方法生成的 IL 代码一样，可利用 **ILDasm.exe** 查看生成的 IL 代码。会看到 C# 编译器用 **Delegate** 类型的 **Combine** 和 **Remove** 公共静态方法调用分别替代了 **+=** 和 **-=** 操作符。

17.5.2 取得对委托链调用的更多控制

此时，想必你已理解了如何创建委托对象链，以及如何调用链中的所有对象。链中的所有项都会被调用，因为委托类型的 **Invoke** 方法包含了对数组中的所有项进行遍历的代码。这是一个很简单的算法。尽管这个简单的算法足以应付很多情形，但也有它的局限性。例如，除了最后一个返回值，其他所有回调方法的返回值都会被丢弃。但局限并不止于此。如果被调用的委托中有一个抛出了异常或阻塞了相当长一段时间，会出现什么情况呢？由于这个简单的算法是顺序调用链中的每一个委托，所以一个委托对象出现问题，链中后续的所有对象都调用不了。显然，这个算法还不够健壮。^①

由于这个算法有的时候不胜其任，所以 **MulticastDelegate** 类提供了一个实例方法 **GetInvocationList**，用于显式调用链中的每一个委托，并允许你使用需要的任何算法：

```
public abstract class MulticastDelegate : Delegate {
    // 创建一个委托数组，其中每个元素都引用链中的一个委托
    public sealed override Delegate[] GetInvocationList();
}
```

GetInvocationList 方法操作从 **MulticastDelegate** 派生的对象，返回包含 **Delegate** 引用的一个数组，其中每个引用都指向链中的一个委托对象。在内部，**GetInvocationList** 构造并初始化一个数组，让它的每个元素都引用链中的一个委托，然后返回对该数组的引用。如果 **_invocationList** 字段为 **null**，返回的数组就只有一个元素，该元素引用链中唯一的委托，即委托实例本身。

可以很容易地写一个算法来显式调用数组中每个对象。以下代码进行了演示：

```
using System;
using System.Reflection;
using System.Text;

// 定义一个 Light (灯) 组件
internal sealed class Light {
    // 该方法返回灯的状态
    public String SwitchPosition() {
        return "The light is off";
    }
}

// 定义一个 Fan (风扇) 组件
internal sealed class Fan {
    // 该方法返回风扇的状态
```

① 健壮性(鲁棒性)和可靠性是有区别的，两者对应的英文单词分别是 **robustness** 和 **reliability**。健壮性主要描述一个系统对于参数变化的不敏感性，而可靠性主要描述一个系统的正确性，也就是在你固定提供一个参数时，它应该产生稳定的、能预测的输出。例如一个程序，它的设计目标是获取一个参数并输出一个值。假如它能正确完成这个设计目标，就说它是可靠的。但在这个程序执行完毕后，假如没有正确释放内存，或者说系统没有自动帮它释放占用的资源，就认为这个程序及其“运行时”不具备健壮性或者鲁棒性。——译注

```

    public String Speed() {
        throw new InvalidOperationException("The fan broke due to overheating");
    }
}

// 定义一个 Speaker (扬声器) 组件
internal sealed class Speaker {
    // 该方法返回扬声器的状态
    public String Volume() {
        return "The volume is loud";
    }
}

public sealed class Program {

    // 定义委托来查询一个组件的状态
    private delegate String GetStatus();

    public static void Main() {
        // 声明空委托链
        GetStatus getStatus = null;

        // 构造 3 个组件，将它们的状态方法添加到委托链中
        getStatus += new GetStatus(new Light().SwitchPosition);
        getStatus += new GetStatus(new Fan().Speed);
        getStatus += new GetStatus(new Speaker().Volume);

        // 显示整理好的状态报告，反映这 3 个组件的状态
        Console.WriteLine(GetComponentStatusReport(getStatus));
    }

    // 该方法查询几个组件并返回状态报告
    private static String GetComponentStatusReport(GetStatus status) {

        // 如果委托链为空，就不进行任何操作
        if (status == null) return null;

        // 用下面的变量来创建状态报告
        StringBuilder report = new StringBuilder();

        // 获得一个数组，其中每个元素都是链中的委托
        Delegate[] arrayOfDelegates = status.GetInvocationList();

        // 遍历数组中的每一个委托
        foreach (GetStatus getStatus in arrayOfDelegates) {
            try {
                // 获得一个组件的状态字符串，把它附加到报告中
                report.AppendFormat("{0}{1}{1}", getStatus(),
                    Environment.NewLine);
            }
            catch (InvalidOperationException e) {
                // 在状态报告中为该组件生成一个错误记录
                Object component = getStatus.Target;
                report.AppendFormat(
                    "Failed to get status from {1}{2}{0} Error: {3}{0}{0}",
                    Environment.NewLine,
                    ((component == null) ? "" : component.GetType() + "."),
                    getStatus.Method.Name,
                    e.Message);
            }
        }
    }
}

```

```

    }
}

// 把整理好的报告返回给调用者
return report.ToString();
}
}

```

生成并运行以上代码，会得到如下所示的输出：

```

The light is off

Failed to get status from Fan.Speed
    Error: The fan broke due to overheating

The volume is loud

```

17.6 委托定义不要太多(泛型委托)

许多年前，Microsoft 在刚开始开发 .NET Framework 的时候引入了委托的概念。开发人员在 FCL 中添加类时，凡是有回调方法的地方都定义了新的委托类型。随着时间的推移，他们定义了许多委托。事实上，现在仅仅在 MSCorLib.dll 中，就有接近 50 个委托类型。下面只列出其中少数几个：

```

public delegate void TryCode(Object userData);
public delegate void WaitCallback(Object state);
public delegate void TimerCallback(Object state);
public delegate void ContextCallback(Object state);
public delegate void SendOrPostCallback(Object state);
public delegate void ParameterizedThreadStart(Object obj);

```

发现这几个委托的共同点了吗？它们其实都是一样的：这些委托类型的变量所引用的方法都是获取一个 **Object**，并返回 **void**。没理由定义这么多委托类型，留一个就可以了！

事实上，.NET Framework 现在支持泛型，所以实际只需几个泛型委托(在 **System** 命名空间中定义)就能表示需要获取多达 16 个参数的方法：

```

public delegate void Action(); // OK, 这个不是泛型
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);

```

所以，.NET Framework 现在提供了 17 个 **Action** 委托，它们从无参数到最多 16 个参数。如需获取 16 个以上的参数，就必须定义自己的委托类型，但这种情况极其罕见。除了 **Action** 委托，.NET Framework 还提供了 17 个 **Func** 函数，允许回调方法返回值：

```

public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate TResult Func<T1, ..., T16, TResult>(T1 arg1, ..., T16 arg16);

```

建议尽量使用这些委托类型，而不是在代码中定义更多的委托类型。这样可减少系统中的类型数量，同时简化编码。然而，如需使用 **ref** 或 **out** 关键字以传引用的方式传递参数，就可能不得不定义自己的委托：

```
delegate void Bar(ref Int32 z);
```

如果委托要通过 C# 的 **params** 关键字获取数量可变的参数，要为委托的任何参数指定默认值，或者要对委托的泛型类型参数进行约束，也必须定义自己的委托类型。

获取泛型实参并返回值的委托支持逆变和协变，而且建议总是利用这些功能，因为它们没有副作用，而且使你的委托适用于更多情形。欲知逆变和协变的详情，请参见 12.5 节“委托和接口的逆变和协变泛型类型实参”。

17.7 C#为委托提供的简化语法

许多程序员因为语法奇怪而对委托有抗拒感。例如下面这行代码：

```
button1.Click += new EventHandler(button1_Click);
```

其中的 **button1_Click** 是方法，看起来像下面这样：

```
void button1_Click(Object sender, EventArgs e) {  
    // 按钮单击后要做的的事情...  
}
```

第一行代码的思路是向按钮控件登记 **button1_Click** 方法的地址，以便在按钮被单击时调用方法。许多程序员认为，仅仅为了指定 **button1_Click** 方法的地址，就构造一个 **EventHandler** 委托对象，这显得有点儿不可思议。然而，构造 **EventHandler** 委托对象是 CLR 要求的，因为这个对象提供了一个包装器，可确保(被包装的)方法只能以类型安全的方式调用。这个包装器还支持调用实例方法和委托链。遗憾的是，很多程序员并不想仔细研究这些细节。程序员更喜欢像下面这样写代码：

```
button1.Click += button1_Click;
```

幸好，Microsoft C# 编译器确实为程序员提供了用于处理委托的一些简化语法。本节将讨论所有这些简化语法。但开始之前我要声明一点，后文描述的基本上只是 C# 的语法糖^①，这些简化语法为程序员提供了一种更简单的方式生成 CLR 和其他编程语言处理委托时所必需的 IL 代码。这些简化语法是 C# 特有的，其他编译器可能还没有提供额外的委托简化语法。

17.7.1 简化语法 1：不需要构造委托对象

如前所述，C# 允许指定回调方法的名称，不必构造委托对象包装器。例如：

```
internal sealed class AClass {
```

① 一般而言，越是高级的语言，提供的简化语法越多，以方便写程序，这就是所谓的“语法糖”。——译注

```

public static void CallbackWithoutNewingADelegateObject() {
    ThreadPool.QueueUserWorkItem(SomeAsyncTask, 5);
}

private static void SomeAsyncTask(Object o) {
    Console.WriteLine(o);
}
}

```

ThreadPool 类的静态 **QueueUserWorkItem** 方法期待一个 **WaitCallback** 委托对象引用, 委托对象中包装的是对 **SomeAsyncTask** 方法的引用。由于 C# 编译器能自己进行推断, 所以可以省略构造 **WaitCallback** 委托对象的代码, 使代码的可读性更佳, 也更容易理解。当然, 当代码编译时, C# 编译器还是会生成 IL 代码来新建 **WaitCallback** 委托对象——只是语法得到了简化而已。

17.7.2 简化语法 2: 不需要定义回调方法(lambda 表达式)

在前面的代码中, 回调方法名称 **SomeAsyncTask** 传给 **ThreadPool** 的 **QueueUserWorkItem** 方法。C# 允许以内联(直接嵌入)的方式写回调方法的代码, 不必在它自己的方法中写。例如, 前面的代码可以这样重写:

```

internal sealed class AClass {
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem( obj => Console.WriteLine(obj), 5 ),
    }
}

```

注意, 传给 **QueueUserWorkItem** 方法的第一个实参是代码(我把它倾斜显示了)! 更正式地说, 这是一个 C# lambda 表达式, 可通过 C# lambda 表达式操作符 `=>` 来轻松识别。lambda 表达式可在编译器预计会看到一个委托的地方使用。编译器看到这个 lambda 表达式之后, 会在类(本例是 **AClass**)中自动定义一个新的私有方法。这个新方法称为匿名函数, 因为方法名称由编译器自动创建, 而且你一般不知道这个名称。^①但可利用 **ILDasm.exe** 这样的工具检查编译器生成的代码。写完前面的代码并编译之后, 我通过 **ILDasm.exe** 看到 C# 编译器将该方法命名为 `<CallbackWithoutNewingADelegateObject>b__0`, 它获取一个 **Object**, 返回 **void**。

编译器选择的方法名以 `<` 符号开头, 这是因为在 C# 中, 标识符是不能包含 `<` 符号的; 这就确保了你不会碰巧定义一个编译器自动选择的名称。顺便说一句, 虽然 C# 禁止标识符包含 `<` 符号, 但 CLR 允许, 这是为什么不会出错的原因。还要注意, 虽然可将方法名作为字符串来传递, 通过反射来访问方法, 但 C# 语言规范指出, 编译器生成名称的方式是没有任何保证的。例如, 每次编译代码, 编译器都可能为方法生成一个不同的名称。

① 作者在这里故意区分了匿名函数和匿名方法。一般情况下, 两者可以互换着使用。如果非要区分, 那么编译器生成的全都是“匿名函数”, 这是最开始的叫法。从 C# 2.0 开始引入了“匿名方法”功能, 它的作用就是简化生成匿名函数而需要写的代码。在新的 C# 版本中(3.0 和以后), 更是建议用 lambda 表达式来进一步简化语法, 不再推荐使用 C# 2.0 引入的“匿名方法”。但归根结底, 所有这些语法糖都是为了更简单地生成匿名函数。——译注

注意 C#编译器向方法应用了 **System.Runtime.CompilerServices.CompilerGeneratedAttribute** 特性, 指出该方法由编译器生成, 而非程序员写的。=>操作符右侧的代码被放入编译器生成的方法中。



注意 写 lambda 表达式时没有办法向编译器生成的方法应用定制特性。此外, 不能向方法应用任何方法修饰符(比如 **unsafe**)。但这一般不会有什问题, 因为编译器生成的匿名函数总是私有方法, 而且方法要么是静态的, 要么是非静态的, 具体取决于方法是否访问了任何实例成员。所以, 没必要向方法应用 **public**, **protected**, **internal**, **virtual**, **sealed**, **override** 或 **abstract** 之类的修饰符。

最后, 如果写前面的代码并编译, C#编译器会将这些代码改写为下面这样(注释是我自己添加的):

```
internal sealed class AClass {
    // 创建该私有字段是为了缓存委托对象,
    // 优点: CallbackWithoutNewingADelegateObject 不会每次调用都新建一个对象
    // 缺点: 缓存的对象永远不会被垃圾回收
    [CompilerGenerated]
    private static WaitCallback <>9__CachedAnonymousMethodDelegate1;

    public static void CallbackWithoutNewingADelegateObject() {
        if (<>9__CachedAnonymousMethodDelegate1 == null) {
            // 第一次调用时, 创建委托对象, 并缓存它
            <>9__CachedAnonymousMethodDelegate1 =
                new WaitCallback(<CallbackWithoutNewingADelegateObject>b__0);
        }
        ThreadPool.QueueUserWorkItem(<>9__CachedAnonymousMethodDelegate1, 5);
    }

    [CompilerGenerated]
    private static void <CallbackWithoutNewingADelegateObject>b__0(Object obj) {
        Console.WriteLine(obj);
    }
}
```

lambda 表达式必须匹配 **WaitCallback** 委托: 获取一个 **Object** 并返回 **void**。但在指定参数名称时, 我简单地将 **obj** 放在=>操作符的左侧。在=>操作符右侧, **Console.WriteLine** 碰巧本来就返回 **void**。然而, 如果在这里放一个返回值不为 **void** 的表达式, 编译器生成的代码会直接忽略返回值, 因为编译器生成的方法必须用 **void** 返回类型来满足 **WaitCallback** 委托。

另外还要注意, 匿名函数被标记为 **private**, 禁止非类型内定义的代码访问(尽管反射能揭示出方法确实存在)。另外, 匿名函数被标记为 **static**, 因为代码没有访问任何实例成员(也不能访问, 因为 **CallbackWithoutNewingADelegateObject** 本身是静态方法)。不过, 代码可引用类中定义的任何静态字段或静态方法。下面是一个例子:

```
internal sealed class AClass {
    private static String sm_name; // 一个静态字段

    public static void CallbackWithoutNewingADelegateObject() {
```

```
ThreadPool.QueueUserWorkItem(
    // 回调代码可引用静态成员
    obj => Console.WriteLine(sm_name+ ": " + obj), 5);
}
```

如果 **CallbackWithoutNewingADelegateObject** 方法不是静态的，匿名函数的代码就可以包含对实例成员的引用。不包含实例成员引用，编译器仍会生成静态匿名函数，因为它的效率比实例方法高。之所以更高效，是因为不需要额外的 **this** 参数。但是，如果匿名函数的代码确实引用了实例成员，编译器就会生成非静态匿名函数：

```
internal sealed class AClass {
    private String m_name; // 一个实例字段

    // 一个实例方法
    public void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(
            // 回调代码可以引用实例成员
            obj => Console.WriteLine(m_name+ ": " + obj), 5);
    }
}
```

=>操作符左侧供指定传给 **lambda** 表达式的参数的名称。下例总结了一些规则：

```
// 如果委托不获取任何参数，就使用 ()
Func<String> f = () => "Jeff";

// 如果委托获取 1 个或更多参数，可显式指定类型
Func<Int32, String> f2 = (Int32 n) => n.ToString();
Func<Int32, Int32, String> f3 = (Int32 n1, Int32 n2) => (n1 + n2).ToString();

// 如果委托获取 1 个或更多参数，编译器可推断类型
Func<Int32, String> f4 = (n) => n.ToString();
Func<Int32, Int32, String> f5 = (n1, n2) => (n1 + n2).ToString();

// 如果委托获取 1 个参数，可省略(和)
Func<Int32, String> f6 = n => n.ToString();

// 如果委托有 ref/out 参数，必须显式指定 ref/out 和类型
Bar b = (out Int32 n) => n = 5;
```

对于最后一个例子，假定 **Bar** 的定义如下：

```
delegate void Bar(out Int32 z);
```

=>操作符右侧供指定匿名函数主体。通常，主体包含要么简单、要么复杂的表达式，并最终返回非 **void** 值。刚才的代码为所有 **Func** 委托变量赋的都是返回 **String** 的 **lambda** 表达式。匿名函数主体还经常只由一个语句构成。调用 **ThreadPool.QueueUserWorkItem** 时就是这种情况，我向它传递了调用 **Console.WriteLine**(返回 **void**)的一个 **lambda** 表达式。

如果主体由两个或多个语句构成，必须用大括号将语句封闭。在用了大括号的情况下，如果委托期待返回值，还必须在主体中添加 **return** 语句，例如：

```
Func<Int32, Int32, String> f7 = (n1, n2) => { Int32 sum = n1 + n2; return sum.ToString(); };
```



重要提示 lambda 表达式的主要优势在于，它从你的源代码中移除了一个“间接层”(a level of indirection)，或者说避免了迂回。正常情况下，必须写一个单独的方法，命名该方法，再在需要委托的地方传递这个方法名。方法名提供了引用代码主体的一种方式，如果要在多个地方引用同一个代码主体，单独写一个方法并命名确实是理想的方案。但如果只需在代码中引用这个主体一次，那么 lambda 表达式允许直接内联那些代码，不必为它分配名称，从而提高了编程效率。



注意 C# 2.0 问世时引入了一个称为匿名方法的功能。和 C# 3.0 引入的 lambda 表达式相似，匿名方法描述的也是创建匿名函数的语法。新规范(C#语言规范 7.14 节)建议开发人员使用新的 lambda 表达式语法，而不要使用旧的匿名方法语法，因为 lambda 表达式语法更简洁，代码更容易写、读和维护。当然，Microsoft C# 编译器仍然支持用这两种语法创建匿名函数，以兼容当年为 C# 2.0 写的代码。在本书中，我只解释并使用 lambda 表达式语法。

17.7.3 简化语法 3：局部变量不需要手动包装到类中即可传给回调方法

前面展示了回调代码如何引用类中定义的其他成员。但有时还希望回调代码引用存在于定义方法中的局部参数或变量。下面是一个有趣的例子：

```
internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {
        // 一些局部变量
        Int32[] squares = new Int32[numToDo];
        AutoResetEvent done = new AutoResetEvent(false);

        // 在其他线程上执行一系列任务
        for (Int32 n = 0; n < squares.Length; n++) {
            ThreadPool.QueueUserWorkItem(
                obj => {
                    Int32 num = (Int32) obj;

                    // 该任务通常更耗时
                    squares[num] = num * num;

                    // 如果是最后一个任务，就让主线程继续运行
                    if (Interlocked.Decrement(ref numToDo) == 0)
                        done.Set();
                }, n);
        }

        // 等待其他所有线程结束运行
        done.WaitOne();

        // 显示结果
        for (Int32 n = 0; n < squares.Length; n++)
```

```

        Console.WriteLine("Index {0}, Square={1}", n, squares[n]);
    }
}

```

这个例子生动地演示了 C#如何简单地实现一个非常复杂的任务。方法定义了一个参数 **numToDo** 和两个局部变量 **squares** 和 **done**。而且 **lambda** 表达式的主体引用了这些变量。

现在,想象 **lambda** 表达式主体中的代码在一个单独的方法中(确实如此,这是 CLR 要求的)。变量的值如何传给这个单独的方法?唯一的办法是定义一个新的辅助类,这个类要为打算传给回调代码的每个值都定义一个字段。此外,回调代码还必须定义成辅助类中的实例方法。然后, **UsingLocalVariablesInTheCallbackCode** 方法必须构造辅助类的实例,用方法定义的局部变量的值来初始化该实例中的字段。然后,构造绑定到辅助对象/实例方法的委托对象。



注意 当 **lambda** 表达式造成编译器生成一个类,而且参数/局部变量被转变成该类的字段后,变量引用的对象的生存期被延长了。正常情况下,在方法中最后一次使用参数/局部变量之后,这个参数/局部变量就会“离开作用域”,结束其生命期。但是,将变量转变成字段后,只要包含字段的那个对象不“死”,字段引用的对象也不会“死”。这在大多数应用程序中不是大问题,但有时要注意一下。

这项工作非常单调乏味,而且容易出错。但理所当然地,它们全部由 C#自动完成。写前面的代码时,C#编译器实际是像下面这样重写了代码(注释是我添加的):

```

internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {

        // 一些局部变量
        WaitCallback callback1 = null;

        // 构造辅助类的实例
        <>c__DisplayClass2 class1 = new <>c__DisplayClass2();

        // 初始化辅助类的字段
        class1.numToDo = numToDo;
        class1.squares = new Int32[class1.numToDo];
        class1.done = new AutoResetEvent(false);

        // 在其他线程上执行一系列任务
        for (Int32 n = 0; n < class1.squares.Length; n++) {
            if (callback1 == null) {
                // 新建的委托对象绑定到辅助对象及其匿名实例方法
                callback1 = new WaitCallback(
                    class1.<UsingLocalVariablesInTheCallbackCode>b__0);
            }

            ThreadPool.QueueUserWorkItem(callback1, n);
        }

        // 等待其他所有线程结束运行
        class1.done.WaitOne();
    }
}

```

```
// 显示结果
for (Int32 n = 0; n < class1.squares.Length; n++)
    Console.WriteLine("Index {0}, Square={1}", n, class1.squares[n]);
}

// 为避免冲突, 辅助类被指定了一个奇怪的名称,
// 而且被指定为私有的, 禁止从 AClass 类外部访问
[CompilerGenerated]
private sealed class <>c__DisplayClass2 : Object {

    // 回调代码要使用的每个局部变量都有一个对应的公共字段
    public Int32[] squares;
    public Int32 numToDo;
    public AutoResetEvent done;

    // 公共无参构造器
    public <>c__DisplayClass2 { }

    // 包含回调代码的公共实例方法
    public void <UsingLocalVariablesInTheCallbackCode>b__0(Object obj) {
        Int32 num = (Int32) obj;
        squares[num] = num * num;
        if (Interlocked.Decrement(ref numToDo) == 0)
            done.Set();
    }
}
```



重要提示 毫无疑问, C# 的 lambda 表达式功能很容易被程序员滥用。我开始使用 lambda 表达式时, 绝对是花了一些时间来熟悉它的。毕竟, 你在一个方法中写的代码实际不在这个方法中。除了有违直觉, 还使调试和单步执行变得比较有挑战性。但事实上, Visual Studio 调试器还是非常不错的。我对自己源代码中的 lambda 表达式进行单步测试时, 它处理得相当好。

我为自己设定了一个规则: 如果需要在回调方法中包含 3 行以上的代码, 就不使用 lambda 表达式。相反, 我会手动写一个方法, 并为其分配自己的名称。但如果使用得当, 匿名方法确实能显著提高开发人员的效率和代码的可维护性。在以下代码中, 使用 lambda 表达式感觉非常自然。没有它们, 这样的代码会很难写、读和维护。

```
// 创建并初始化一个 String 数组
String[] names = { "Jeff", "Kristin", "Aidan", "Grant" };

// 只获取含有小写字母 'a' 的名字
Char charToFind = 'a';
names = Array.FindAll(names, name => name.IndexOf(charToFind) >= 0);

// 将每个字符串的字符转换为大写
names = Array.ConvertAll(names, name => name.ToUpper());

// 显示结果
Array.ForEach(names, Console.WriteLine);
```

17.8 委托和反射

本章到目前为止，使用委托都要求开发人员事先知道回调方法的原型。例如，假定 **fb** 是引用了一个 **Feedback** 委托的变量(参见本章 17.1 节的示例程序)，那么为了调用这个委托，代码应该像下面这样写：

```
fb(item); // item 被定义 Int32
```

可以看出，编码时必须知道回调方法需要多少个参数，以及参数的具体类型。还好，开发人员几乎总是知道这些信息，所以像前面那样写代码是没有问题的。

不过在个别情况下，这些信息在编译时并不知道。第 11 章“事件”讨论 **EventSet** 类型时曾展示了一个例子。这个例子用字典来维护一组不同的委托类型。在运行时，为了引发事件，要在字典中查找并调用一个委托。但编译时不可能准确地知道要调用哪个委托，哪些参数必须传给委托的回调方法。

幸好 **System.Delegate.MethodInfo** 提供了一个 **CreateDelegate** 方法，允许在编译时不知道委托的所有必要信息的前提下创建委托。下面是 **MethodInfo** 为该方法定义的重载：

```
public abstract class MethodInfo : MethodBase {
    // 构造包装了一个静态方法的委托
    public virtual Delegate CreateDelegate(Type delegateType);

    // 构造包装了一个实例方法的委托; target 引用“this”实参
    public virtual Delegate CreateDelegate(Type delegateType, Object target);
}
```

创建好委托后，用 **Delegate** 的 **DynamicInvoke** 方法调用它，如下所示：

```
public abstract class Delegate {
    // 调用委托并传递参数
    public Object DynamicInvoke(params Object[] args);
}
```

使用反射 API(参见第 23 章“程序集加载和反射”)，首先必须获取引用了回调方法的一个 **MethodInfo** 对象。然后，调用 **CreateDelegate** 方法来构造由第一个参数 **delegateType** 所标识的 **Delegate** 派生类型的对象。如果委托包装了实例方法，还要向 **CreateDelegate** 传递一个 **target** 参数，指定作为 **this** 参数传给实例方法的对象。

System.Delegate 的 **DynamicInvoke** 方法允许调用委托对象的回调方法，传递一组在运行时确定的参数。调用 **DynamicInvoke** 时，它会在内部保证传递的参数与回调方法期望的参数兼容。如果兼容，就调用回调方法；否则抛出 **ArgumentException** 异常。**DynamicInvoke** 返回回调方法所返回的对象。

以下代码演示了如何使用 **CreateDelegate** 方法和 **DynamicInvoke** 方法：

```
using System;
using System.Reflection;
using System.IO;
```

```

using System.Linq;

// 下面是一些不同的委托定义
internal delegate Object TwoInt32s(Int32 n1, Int32 n2);
internal delegate Object OneString(String s1);

public static class DelegateReflection {
    public static void Main(String[] args) {
        if (args.Length < 2) {
            String usage =
                @"Usage: " +
                "{0} delType methodName [Arg1] [Arg2]" +
                "{0} where delType must be TwoInt32s or OneString"+
                "{0} if delType is TwoInt32s, methodName must be Add or Subtract" +
                "{0} if delType is OneString, methodName must be NumChars or Reverse"+
                "{0}" +
                "{0}Examples: " +
                "{0} {1} TwoInt32s Add 123 321" +
                "{0} {1} TwoInt32s Subtract 123 321" +
                "{0} {1} OneString NumChars \"Hello there\"" +
                "{0} {1} OneString Reverse \"Hello there\"";
            Console.WriteLine(usage, Environment.NewLine);
            return;
        }

        // 将 delType 实参转换为委托类型
        Type delType = Type.GetType(args[0]);
        if (delType == null) {
            Console.WriteLine("Invalid delType argument: " + args[0]);
            return;
        }

        Delegate d;
        try {
            // 将 Arg1 实参转换为方法
            MethodInfo mi =
                typeof(DelegateReflection).GetTypeInfo().GetDeclaredMethod(args[1]);

            // 创建包装了静态方法的委托对象
            d = mi.CreateDelegate(delType);
        }
        catch (ArgumentException) {
            Console.WriteLine("Invalid methodName argument: " + args[1]);
            return;
        }

        // 创建一个数组，其中只包含要通过委托对象传给方法的参数
        Object[] callbackArgs = new Object[args.Length - 2];

        if (d.GetType() == typeof(TwoInt32s)) {
            try {
                // 将 String 类型的参数转换为 Int32 类型的参数
                for (Int32 a = 2; a < args.Length; a++)
                    callbackArgs[a - 2] = Int32.Parse(args[a]);
            }
            catch (FormatException) {
                Console.WriteLine("Parameters must be integers.");
                return;
            }
        }

        if (d.GetType() == typeof(OneString)) {
            // 只复制 String 参数
            Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
        }
    }
}

```

```
try {  
    // 调用委托并显示结果  
    Object result = d.DynamicInvoke(callbackArgs);  
    Console.WriteLine("Result = " + result);  
}  
catch (TargetParameterCountException) {  
    Console.WriteLine("Incorrect number of parameters specified.");  
}  
}  
  
// 这个回调方法获取 2 个 Int32 参数  
private static Object Add(Int32 n1, Int32 n2) {  
    return n1 + n2;  
}  
  
// 这个回调方法获取 2 个 Int32 参数  
private static Object Subtract(Int32 n1, Int32 n2) {  
    return n1 - n2;  
}  
  
// 这个回调方法获取 1 个 String 参数  
private static Object NumChars(String s1) {  
    return s1.Length;  
}  
  
// 这个回调方法获取 1 个 String 参数  
private static Object Reverse(String s1) {  
    return new String(s1.Reverse().ToArray());  
}  
}
```

1. The first step in the process of the development of a new product is the identification of a market need.

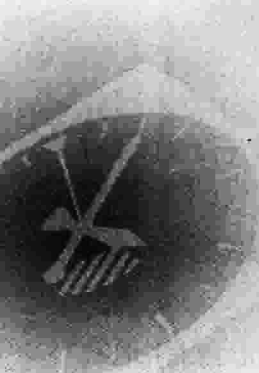
2. The second step is the selection of a product concept that meets the market need.

3. The third step is the development of a product prototype.

4. The fourth step is the testing of the product prototype.

5. The fifth step is the production of the product.

6. The sixth step is the distribution of the product.



第 18 章 定制特性

本章内容:

- 使用定制特性
- 定义自己的特性类
- 特性构造器和字段/属性数据类型
- 检测定制特性
- 两个特性实例的相互匹配
- 检测定制特性时不创建从 **Attribute** 派生的对象
- 条件特性类

本章讨论 Microsoft .NET Framework 提供的最具创意的功能之一: 定制特性(custom attribute)。利用定制特性, 可宣告式地为自己的代码构造添加注解来实现特殊功能。定制特性允许为几乎每一个元数据表记录项定义和应用信息。这种可扩展的元数据信息能在运行时查询, 从而动态改变代码的执行方式。使用各种 .NET Framework 技术(Windows 窗体、WPF 和 WCF 等), 会发现它们都利用了定制特性, 目的是方便开发者在代码中表达他们的意图。任何 .NET Framework 开发人员都有必要完全掌握定制特性。

18.1 使用定制特性

我们都知道能将 **public**, **private**, **static** 这样的特性应用于类型和成员。我们都同意应用特性具有很大的作用。但是, 如果能定义自己的特性, 会不会更有用? 例如, 能不能定义一个类型, 指出该类型能通过序列化来进行远程处理? 能不能将特性应用于方法, 指出执行该方法需要授予特定安全权限?

为类型和方法创建和应用用户自定义的特性能带来极大的便利。当然, 编译器必须理解这些特性, 才能在最终的元数据中生成特性信息。由于编译器厂商一般不会发布其编译器产品的源代码, 所以 Microsoft 采取另一种机制提供对用户自定义特性的支持。这个机制称为定制特性。它的功能很强大, 在应用程序的设计时和运行时都能发挥重要作用。任何人都能定义和使用定制特性。另外, 面向 CLR 的所有编译器都必须识别定制特性, 并能在最终的元数据中生成特性信息。

关于自定义特性, 首先要知道它们只是将一些附加信息与某个目标元素关联起来的方式。

编译器在托管模块的元数据中生成(嵌入)这些额外的信息。大多数特性对编译器来说没有意义；编译器只是机械地检测源代码中的特性，并生成对应的元数据。

.NET Framework 类库(FCL)定义了几百个定制特性，可将它们应用于自己源代码中的各种元素。下面是一些例子。

- 将 **DllImport** 特性应用于方法，告诉 CLR 该方法的实现位于指定 DLL 的非托管代码中。
- 将 **Serializable** 特性应用于类型，告诉序列化格式化器^①一个实例的字段可以序列化和反序列化。
- 将 **AssemblyVersion** 特性应用于程序集，设置程序集的版本号。
- 将 **Flags** 特性应用于枚举类型，枚举类型就成了位标志(bit flag)集合。

以下 C# 代码应用了大量特性。在 C# 中，为了将定制特性应用于目标元素，要将特性放置于目标元素前的一对方括号中。代码本身做的事情不重要，重要的是对特性有一个认识。

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

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
internal sealed class OSVERSIONINFO {
    public OSVERSIONINFO() {
        OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
    }

    public UInt32 OSVersionInfoSize = 0;
    public UInt32 MajorVersion = 0;
    public UInt32 MinorVersion = 0;
    public UInt32 BuildNumber = 0;
    public UInt32 PlatformId = 0;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public String CSDVersion = null;
}

internal sealed class MyClass {
    [DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO ver);
}
```

在这里，**StructLayout** 特性应用于 **OSVERSIONINFO** 类，**MarshalAs** 特性应用于 **CSDVersion** 字段，**DllImport** 特性应用于 **GetVersionEx** 方法，而 **In** 和 **Out** 特性应用于 **GetVersionEx** 方法的 **ver** 参数。每种编程语言都定义了将定制特性应用于目标元素所采用的语法。例如，Microsoft Visual Basic .NET 要求使用一对尖括号(<>)而不是方括号。

CLR 允许将特性应用于可在文件的元数据中表示的几乎任何东西。不过，最常应用特性的

① “格式化器”是本书的译法，文档翻译成“格式化程序”。格式化器是实现了 **System.Runtime.Serialization.IFormatter** 接口的类型，它知道如何序列化和反序列化一个对象图。——译注

还是以下定义表中的记录项: **TypeDef**(类、结构、枚举、接口和委托), **MethodDef**(含构造器), **ParamDef**, **FieldDef**, **PropertyDef**, **EventDef**, **AssemblyDef** 和 **ModuleDef**。更具体地说, C#只允许将特性应用于定义以下任何目标元素的源代码: 程序集、模块、类型(类、结构、枚举、接口、委托)、字段、方法(含构造器)、方法参数、方法返回值、属性、事件和泛型类型参数。

应用特性时, C#允许用一个前缀明确指定特性要应用于的目标元素。以下代码展示了所有可能的前缀。许多时候, 即使省略前缀, 编译器也能判断特性要应用于什么目标元素(就像上例展示的那样)。但在其他时候, 必须指定前缀向编译器清楚表明我们的意图。下面倾斜显示的前缀是必须的。

```
using System;

[assembly: SomeAttr]           // 应用于程序集
[module: SomeAttr]           // 应用于模块

[type: SomeAttr]              // 应用于类型
internal sealed class SomeType<[typevar: SomeAttr] T> { // 应用于泛型类型变量

    [field: SomeAttr]          // 应用于字段
    public Int32 SomeField = 0;

    [return: SomeAttr]         // 应用于返回值
    [method: SomeAttr]         // 应用于方法
    public Int32 SomeMethod(
        [param: SomeAttr]      // 应用于参数
        Int32 SomeParam) { return SomeParam; }

    [property: SomeAttr]       // 应用于属性
    public String SomeProp {
        [method: SomeAttr]     // 应用于 get 访问器方法
        get { return null; }
    }

    [event: SomeAttr]          // 应用于事件
    [field: SomeAttr]          // 应用于编译器生成的字段
    [method: SomeAttr]         // 应用于编译器生成的 add & remove 方法
    public event EventHandler SomeEvent;
}
```

前面介绍了如何应用定制特性, 现在看看特性到底是什么。定制特性其实是一个类型的实例。为了符合“公共语言规范”(CLS)的要求, 定制特性类必须直接或间接从公共抽象类 **System.Attribute** 派生。C#只允许符合 CLS 规范的特性。查看文档会发现定义了以下类(参见前面的例子): **StructLayoutAttribute**, **MarshalAsAttribute**, **DllImportAttribute**, **InAttribute** 和 **OutAttribute**。所有这些类碰巧都在 **System.Runtime.InteropServices** 命名空间中定义。但特性类可以在任何命名空间中定义。进一步查看, 会发现所有这些类都从 **System.Attribute** 派生, 所有符合 CLS 规范的特性类都肯定从这个类派生。



注意 将特性应用于源代码中的目标元素时，C#编译器允许省略 **Attribute** 后缀以减少打字量，并提升源代码的可读性。本章许多示例代码都利用了 C#提供的这一便利。例如，许多源代码用的都是 **[DllImport(...)]**，而不是 **[DllImportAttribute(...)]**。

如前所述，特性是类的实例。类必须有公共构造器才能创建它的实例。所以，将特性应用于目标元素时，语法类似于调用类的某个实例构造器。除此之外，语言可能支持一些特殊的语法，允许设置与特性类关联的公共字段或属性。前面的例子将 **DllImport** 特性应用于 **GetVersionEx** 方法：

```
[DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
```

这一行代码的语法表面上看很奇怪，因为调用构造器时永远不会使用这样的语法。查阅 **DllImportAttribute** 类的文档，会发现它的构造器要求接受一个 **String** 参数。在这个例子中，"**Kernel32**"将传给这个参数。构造器的参数称为**定位参数**(positional parameter)，而且是强制性的；也就是说，应用特性时必须指定参数。

那么，另外两个“参数”是什么呢？这种特殊的语法允许在构造好 **DllImportAttribute** 对象后设置对象的任何公共字段或属性。在这个例子中，将"**Kernel32**"传给构造器并构造好 **DllImportAttribute** 对象之后，对象的公共实例字段 **CharSet** 和 **SetLastError** 分别设为 **CharSet.Auto** 和 **true**。用于设置字段或属性的“参数”称为**命名参数**(named parameter)。这种参数是可选的，因为在应用特性的实例时不一定要指定参数。稍后会解释是什么导致了实际地构造 **DllImportAttribute** 类的实例。

还要注意，可将多个特性应用于一个目标元素。例如，在本章的第一个示例程序中，**GetVersionEx** 方法的 **ver** 参数同时应用了 **In** 和 **Out** 这两个特性。将多个特性应用于单个目标元素时，注意特性的顺序无关紧要。另外，在 C#中，既可将每个特性都封闭到一对方括号中，也可在一对方括号中封闭多个以逗号分隔的特性。如果特性类的构造器不获取参数，那么圆括号可以省略。最后，就像前面说到的那样，**Attribute** 后缀也是可选的。下面代码行具有相同的行为，它们演示了应用多个特性时所有可能的方式：

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

18.2 定义自己的特性类

现在已经知道特性是从 **System.Attribute** 派生的一个类的实例，也知道了如何应用特性。接着研究如何定义定制特性类。假定你是 Microsoft 的员工，负责为枚举类型添加位标志(bit flag)支持，那么要做的第一件事就是定义一个 **FlagsAttribute** 类：

```
namespace System {
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
```

```

    }
}

```

注意，**FlagsAttribute** 类从 **Attribute** 继承。这使 **FlagsAttribute** 类成为符合 CLS 规范的定制特性。此外，注意类名有 **Attribute** 后缀；这是为了保持与标准的相容性，但这并不是必须的。最后，所有非抽象特性至少要包含一个公共构造器。上述代码中的 **FlagsAttribute** 构造器非常简单，不获取任何参数，也不做任何事情。



重要提示 应将特性想像成逻辑状态容器。也就是说，虽然特性类型是一个类，但这个类应该很简单。应该只提供一个公共构造器来接受特性的强制性(或定位性)状态信息，而且这个类可以提供公共字段和属性，以接受特性的可选(或命名)状态信息。类不应提供任何公共方法、事件或其他成员。

我通常不鼓励使用公共字段。特性也不例外，我同样不鼓励在这种类型中使用公共字段。使用属性要好得多。因为在更改特性类的实现方式时，属性能提供更大的灵活性。

现在的情况是 **FlagsAttribute** 类的实例能应用于任何目标元素。但事实上，这个特性应该只能应用于枚举类型，应用于属性或方法是没有意义的。为了告诉编译器这个特性的合法应用范围，需要向特性类应用 **System.AttributeUsageAttribute** 类的实例。下面是新的代码：

```

namespace System {
    [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
        }
    }
}

```

新版本将 **AttributeUsageAttribute** 的实例应用于特性。毕竟，特性类型本质上还是类，而类是可以应用特性的。**AttributeUsage** 特性是一个简单的类，可利用它告诉编译器定制特性的合法应用范围。所有编译器都内建了对该特性的支持，并会在定制特性应用于无效目标时报错。在这个例子中，**AttributeUsage** 特性指出 **Flags** 特性的实例只能应用于枚举类型的目标。

由于特性不过是类型，所以 **AttributeUsageAttribute** 类理解起来很容易。以下是该类的 FCL 源代码：

```

[Serializable]
[AttributeUsage(AttributeTargets.Class, Inherited=true)]
public sealed class AttributeUsageAttribute : Attribute {
    internal static AttributeUsageAttribute Default =
        new AttributeUsageAttribute(AttributeTargets.All);

    internal Boolean m_allowMultiple = false;
}

```

```

internal AttributeTargets m_attributeTarget = AttributeTargets.All;
internal Boolean m_inherited = true;

// 这是一个公共构造器
public AttributeUsageAttribute(AttributeTargets validOn) {
    m_attributeTarget = validOn;
}

internal AttributeUsageAttribute(AttributeTargets validOn,
    Boolean allowMultiple, Boolean inherited) {
    m_attributeTarget = validOn;
    m_allowMultiple = allowMultiple;
    m_inherited = inherited;
}

public Boolean AllowMultiple {
    get { return m_allowMultiple; }
    set { m_allowMultiple = value; }
}

public Boolean Inherited {
    get { return m_inherited; }
    set { m_inherited = value; }
}

public AttributeTargets ValidOn {
    get { return m_attributeTarget; }
}
}

```

如你所见，**AttributeUsageAttribute** 类有一个公共构造器，它允许传递位标志(bit flag)来指明特性的合法应用范围。**System.AttributeTargets** 枚举类型在 FCL 中是像下面这样定义的：

```

[Flags, Serializable]
public enum AttributeTargets {
    Assembly          = 0x0001,
    Module            = 0x0002,
    Class             = 0x0004,
    Struct            = 0x0008,
    Enum              = 0x0010,
    Constructor        = 0x0020,
    Method            = 0x0040,
    Property           = 0x0080,
    Field             = 0x0100,
    Event             = 0x0200,
    Interface          = 0x0400,
    Parameter          = 0x0800,
    Delegate           = 0x1000,
    ReturnValue        = 0x2000,
    GenericParameter   = 0x4000,
    All = Assembly | Module | Class | Struct | Enum |
        Constructor | Method | Property | Field | Event |
        Interface | Parameter | Delegate | ReturnValue |
        GenericParameter
}

```

AttributeUsageAttribute 类提供了两个附加的公共属性，即 **AllowMultiple** 和 **Inherited**。可在向特性类应用特性时选择设置这两个属性。

大多数特性多次应用于同一个目标是没有意义的。例如，将 **Flags** 或 **Serializable** 特性多次应用于同一个目标不会有任何好处。事实上，编译如下所示的代码：

```

[Flags][Flags]
internal enum Color {

```

```
Red
}
```

编译器会报告以下错误：

error CS0579: 重复的"Flags"特性。

但少数几个特性确实有必要多次应用于同一个目标。FCL 特性类 **ConditionalAttribute** 允许将它的多个实例应用于同一个目标元素。不将 **AllowMultiple** 明确设为 **true**，特性就只能向选定的目标元素应用一次。

AttributeUsageAttribute 的另一个属性是 **Inherited**，它指出特性在应用于基类时，是否同时应用于派生类和重写的方法。以下代码演示了特性的继承：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    Inherited=true)]
internal class TastyAttribute : Attribute {
}

[Tasty][Serializable]
internal class BaseType {

    [Tasty] protected virtual void DoSomething() { }
}

internal class DerivedType : BaseType {
    protected override void DoSomething() { }
}
```

在上述代码中，**DerivedType** 及其 **DoSomething** 方法都被视为 **Tasty**，因为 **TastyAttribute** 类被标记为可继承。但 **DerivedType** 不可序列化，因为 FCL 的 **SerializableAttribute** 类被标记为不可继承。

注意，.NET Framework 只认为类、方法、属性、事件、字段、方法返回值和参数等目标元素是可继承的。所以，定义特性类型时，只有在该特性应用于上述某个目标的前提下，才应该将 **Inherited** 设为 **true**。注意，可继承特性不会造成在托管模块中为派生类型生成额外的元数据。18.4 节“检测定制特性”将进一步讨论这方面的问题。



注意 定义自己的特性类时，如果忘记向自己的类应用 **AttributeUsage** 特性，编译器和 CLR 将假定该特性能应用于所有目标元素，向每个目标元素都只能应用一次，而且可继承。这些假定模仿了 **AttributeUsageAttribute** 类中的默认字段值。

18.3 特性构造器和字段/属性数据类型

定制特性类可定义构造器来获取参数。开发人员在应用特性类的实例时必须指定这些参数。还可在类中定义非静态公共字段和属性，使开发人员能为特性类的实例选择恰当的设置。

定义特性类的实例构造器、字段和属性时，可供选择的数据类型并不多。具体地说，只允

许 **Boolean**, **Char**, **Byte**, **SByte**, **Int16**, **UInt16**, **Int32**, **UInt32**, **Int64**, **UInt64**, **Single**, **Double**, **String**, **Type**, **Object** 或枚举类型。此外, 可使用上述任意类型的一维 0 基数组。但应尽量避免使用数组, 因为对于定制特性, 如果它的构造器要获取数组作为参数, 就会失去与 CLS 的相容性。

应用特性时必须传递一个编译时常量表达式, 它与特性类定义的类型匹配。在特性类定义了一个 **Type** 参数、**Type** 字段或者 **Type** 属性的任何地方, 都必须使用 C# **typeof** 操作符(如下例所示)。在特性类定义了一个 **Object** 参数、**Object** 字段或者 **Object** 属性的任何地方, 都可传递一个 **Int32**、**String** 或其他任何常量表达式(包括 **null**)。如果常量表达式代表值类型, 那么在运行时构造特性的实例时会对值类型进行装箱。以下是一个示例特性及其用法:

```
using System;

internal enum Color { Red }

[AttributeUsage(AttributeTargets.All)]
internal sealed class SomeAttribute : Attribute {
    public SomeAttribute(String name, Object o, Type[] types) {
        // 'name'引用一个 String
        // 'o'引用一个合法的类型(如有必要, 就进行装箱)
        // 'types'引用一个一维 0 基 Type 数组
    }
}

[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
internal sealed class SomeType {
}
```

逻辑上, 当编译器检测到向目标元素应用了定制特性时, 会调用特性类的构造器, 向它传递任何指定的参数, 从而构造特性类的实例。然后, 编译器采用增强型构造器语法所指定的值, 对任何公共字段和属性进行初始化。构造并初始化好定制特性类的对象之后, 编译器将它的状态序列化到目标元素的元数据表记录项中。



重要提示 为方便理解, 可以这样想象定制特性: 它是类的实例, 被序列化成驻留在元数据中的字节流。运行时可对元数据中的字节进行反序列化, 从而构造出类的实例。实际发生的事情是: 编译器在元数据中生成创建特性类的实例所需的信息。每个构造器参数都是 1 字节的类型 ID, 后跟具体的值。对构造器的参数进行“序列化”时, 编译器先写入字段/属性名称, 再跟上 1 字节的类型 ID, 最后是具体的值。如果是数组, 则会先保存数组元素的个数, 再跟上每个单独的元素。

18.4 检测定制特性

仅仅定义特性类没有用。确实可以定义自己想要的所有特性类, 并应用自己想要的所有实例。但这样除了在程序集中生成额外的元数据, 没有其他任何意义。应用程序代码的行为不会有任何改变。

第 15 章“枚举类型和位标志”描述了如何将 **Flags** 特性应用于枚举类型，从而改变 **System.Enum** 的 **ToString** 和 **Format** 方法的行为。方法的行为之所以改变，是因为它们会在运行时检查自己操作的枚举类型是否关联了 **Flags** 特性元数据。代码利用一种称为反射的技术检测特性的存在。这里只是简单地演示一下反射。第 23 章“程序集加载和反射”会完整地讨论这种技术。

假定你是 Microsoft 的员工，负责实现 **Enum** 的 **ToString** 方法，你会像下面这样实现它：

```
public override String ToString() {
    // 枚举类型是否应用了 FlagsAttribute 类型的实例？
    if (this.GetType().IsDefined(typeof(FlagsAttribute), false)) {
        // 如果是，就执行代码，将值视为一个位标志枚举类型
        ...
    } else {
        // 如果不是，就执行代码，将值视为一个普通枚举类型
        ...
    }
    ...
}
```

上述代码调用 **Type** 的 **IsDefined** 方法，要求系统查看枚举类型的元数据，检查是否关联了 **FlagsAttribute** 类的实例。如果 **IsDefined** 返回 **true**，表明 **FlagsAttribute** 的一个实例已与枚举类型关联，**ToString** 方法认为值包含了一个位标志(bit flag)集合。如果 **IsDefined** 返回 **false**，**Format** 方法认为值是普通的枚举类型。

因此，在定义定制特性时，也必须实现一些代码来检测某些目标上是否存在该特性类的实例，然后执行一些逻辑分支代码。这样定制特性才能真正发挥作用。

FCL 提供了多种方式来检测特性的存在。如果通过 **System.Type** 对象来检测特性，可以像前面展示的那样使用 **IsDefined** 方法。但有时需要检测除了类型之外的其他目标(比如程序集、模块或方法)上的特性。为简化讨论，让我们聚焦于 **System.Reflection.CustomAttributeExtensions** 类定义的扩展方法。该类定义了三个静态方法来获取与目标关联的特性：**IsDefined**，**GetCustomAttributes** 和 **GetCustomAttribute**。每个方法都有几个重载版本。例如，每个方法都有一个版本能操作类型成员(类、结构、枚举、接口、委托、构造器、方法、属性、字段、事件和返回类型)、参数、模块和程序集。还有一些版本能指示系统遍历继承层次结构，在结果中包含继承的特性。表 18-1 简要总结了每个方法的用途，它们在元数据上反射以查找与 CLS 相容的定制特性类型的实例。

如果只想判断目标是否应用了一个特性，那么应该调用 **IsDefined**，因为它比另两个方法更高效。但我们知道，将特性应用于目标时，可以为特性的构造器指定参数，并可选择设置字段和属性。使用 **IsDefined** 不会构造特性对象，不会调用构造器，也不会设置字段和属性。

要构造特性对象，必须调用 **GetCustomAttributes** 或 **GetCustomAttribute** 方法。每次调用这两个方法，都会构造指定特性类型的新实例，并根据源代码中指定的值来设置每个实例的字段和属性。两个方法返回的都是对完全构造好的特性类实例的引用。

表 18-1 System.Reflection.CustomAttributeExtensions 定义三个静态方法

方法名称	说明
IsDefined	如果至少有一个指定的 Attribute 派生类的实例与目标关联, 就返回 true 。这个方法效率很高, 因为它不构造(反序列化)特性类的任何实例
GetCustomAttributes	返回应用于目标的指定特性对象的集合。每个实例都使用编译时指定的参数、字段和属性来构造(反序列化)。如果目标没有应用指定特性类的实例, 就返回一个空集合。该方法通常用于已将 AllowMultiple 设为 true 的特性, 或者用于列出已应用的所有特性
GetCustomAttribute	返回应用于目标的指定特性类的实例。实例使用编译时指定的参数、字段和属性来构造(反序列化)。如果目标没有应用指定特性类的实例, 就返回 null 。如果目标应用了指定特性的多个实例, 就抛出 System.Reflection.AmbiguousMatchException 异常。该方法通常用于已将 AllowMultiple 设为 false 的特性

调用上述任何方法, 内部都必须扫描托管模块的元数据, 执行字符串比较来定位指定的定制特性类。显然, 这些操作会耗费一定时间。假如对性能的要求比较高, 可考虑缓存这些方法的调用结果, 而不是反复调用来请求相同的信息。

System.Reflection 命名空间定义了几个类允许检查模块的元数据。这些类包括 **Assembly**, **Module**, **ParameterInfo**, **MemberInfo**, **Type**, **MethodInfo**, **ConstructorInfo**, **FieldInfo**, **EventInfo**, **PropertyInfo** 及其各自的 ***Builder** 类。所有类都提供了 **IsDefined** 和 **GetCustomAttributes** 方法。

反射类提供的 **GetCustomAttributes** 方法返回的是由 **Object** 实例构成的数组(**Object[]**), 而不是由 **Attribute** 实例构成的数组(**Attribute[]**)。这是由于反射类能返回不相容于 CLS 规范的特性类的对象。不过, 大可不必关心这种不一致性, 因为非 CLS 相容的特性是很稀少的。事实上, 我与 .NET Framework 打交道至今, 还没有见过一例。



注意 只有 **Attribute**, **Type** 和 **MethodInfo** 类才实现了支持 **Boolean inherit** 参数的反射方法。其他能检查特性的所有反射方法都会忽略 **inherit** 参数, 而且不会检查继承层次结构。要检查事件、属性、字段、构造器或参数是否应用了继承的特性, 只能调用 **Attribute** 的某个方法。

还要注意, 将一个类传给 **IsDefined**, **GetCustomAttribute** 或者 **GetCustomAttributes** 方法时, 这些方法会检测是否应用了指定的特性类或者它的派生类。如果只是想搜索一个具体的特性类, 应针对返回值执行一次额外的检查, 确保方法返回的正是想搜索的类。还可考虑将自己的特性类定义成 **sealed**, 减少可能存在的混淆, 并避免执行这个额外的检查。

以下示例代码列出了一个类型中定义的所有方法, 并显示应用于每个方法的特性。代码仅供演示, 平时不会像这样将这些定制特性应用于这些目标元素。

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
```

```

[assembly: CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name = "Jeff", Target = typeof(Program))]
public sealed class Program {
    [Conditional("Debug")]
    [Conditional("Release")]
    public void DoSomething() { }

    public Program() {
    }

    [CLSCompliant(true)]
    [STAThread]
    public static void Main()
    {
        // 显示应用于这个类型的特性集
        ShowAttributes(typeof(Program));

        // 获取与类型关联的方法集
        var members =
            from m in typeof(Program).GetTypeInfo().DeclaredMembers.OfType<MethodBase>()
            where m.IsPublic
            select m;

        foreach (MemberInfo member in members) {
            // 显示应用于这个成员的特性集
            ShowAttributes(member);
        }

        private static void ShowAttributes(MemberInfo attributeTarget) {
            var attributes = attributeTarget.GetCustomAttributes<Attribute>();

            Console.WriteLine("Attributes applied to {0}: {1}",
                attributeTarget.Name, (attributes.Count() == 0 ? "None" : String.Empty));

            foreach (Attribute attribute in attributes) {
                // 显示所应用的每个特性的类型
                Console.WriteLine(" {0}", attribute.GetType().ToString());

                if (attribute is DefaultMemberAttribute)
                    Console.WriteLine(" MemberName={0}",
                        ((DefaultMemberAttribute)attribute).MemberName);
                if (attribute is ConditionalAttribute)
                    Console.WriteLine(" ConditionString={0}",
                        ((ConditionalAttribute)attribute).ConditionString);
                if (attribute is CLSCompliantAttribute)
                    Console.WriteLine(" IsCompliant={0}",
                        ((CLSCompliantAttribute)attribute).IsCompliant);

                DebuggerDisplayAttribute dda = attribute as DebuggerDisplayAttribute;
                if (dda != null) {
                    Console.WriteLine(" Value={0}, Name={1}, Target={2}",
                        dda.Value, dda.Name, dda.Target);
                }
            }
            Console.WriteLine();
        }
    }
}

```

编译并运行上述应用程序得到以下输出:

```

Attributes applied to Program:
System.SerializableAttribute
System.Reflection.DefaultMemberAttribute

```

```

MemberName=Main
System.Diagnostics.DebuggerDisplayAttribute
Value=Richter, Name=Jeff, Target=Program

Attributes applied to DoSomething:
System.Diagnostics.ConditionalAttribute
ConditionString=Debug
System.Diagnostics.ConditionalAttribute
ConditionString=Release

Attributes applied to Main:
System.STAThreadAttribute
System.CLSCompliantAttribute
IsCompliant=True

Attributes applied to .ctor: None

```

18.5 两个特性实例的相互匹配

除了判断是否向目标应用了一个特性的实例,可能还需要检查特性的字段来确定它们的值。一个办法是老老实实写代码检查特性类的字段值。但 **System.Attribute** 重写了 **Object** 的 **Equals** 方法,会在内部比较两个对象的类型。不一致会返回 **false**。如果一致,Equals 会利用反射来比较两个特性对象中的字段值(为每个字段都调用 **Equals**)。所有字段都匹配就返回 **true**; 否则返回 **false**。可在自己的定制特性类中重写 **Equals** 来移除反射的使用,从而提升性能。

System.Attribute 还公开了虚方法 **Match**, 可重写它来提供更丰富的语义。**Match** 的默认实现只是调用 **Equal** 方法并返回它的结果。下例演示了如何重写 **Equals** 和 **Match**, 后者在一个特性代表另一个特性的子集的前提下返回 **true**。另外, 还演示了如何使用 **Match**。

```

using System;

[Flags]
internal enum Accounts {
    Savings    = 0x0001,
    Checking   = 0x0002,
    Brokerage  = 0x0004
}

[AttributeUsage(AttributeTargets.Class)]
internal sealed class AccountsAttribute : Attribute {
    private Accounts m_accounts;

    public AccountsAttribute(Accounts accounts) {
        m_accounts = accounts;
    }

    public override Boolean Match(Object obj)
    {
        // 如果基类实现了 Match, 而且基类不是
        // Attribute, 就取消对下面这行代码的注释
        // if (!base.Match(obj)) return false;

        // 由于 'this' 不为 null, 所以假如 obj 为 null,
        // 那么对象肯定不匹配
        // 注意: 如果你信任基类正确实现了 Match,
        // 那么下面这一行可以删除
        if (obj == null) return false;
    }
}

```

```

// 如果对象属于不同的类型, 肯定不匹配
// 注意: 如果你信任基类正确实现了 Match,
// 那么下面这一行可以删除
if (this.GetType() != obj.GetType()) return false;

// 将 obj 转型为我们的类型以访问字段
// 注意: 转型不可能失败, 因为我们知道
// 两个对象是相同的类型
AccountsAttribute other = (AccountsAttribute)obj;

// 比较字段, 判断它们是否有相同的值
// 这个例子判断'this'的帐户是不是
// other 的帐户的一个子集
if ((other.m_accounts & m_accounts) != m_accounts)
    return false;

return true; // 对象匹配
}

public override Boolean Equals(Object obj)
{
    // 如果基类实现了 Equals, 而且基类不是
    // Object, 就取消对下面这行代码的注释:
    // if (!base.Equals(obj)) return false

    // 由于'this 不为 null, 所以假如 object 为 null,
    // 那么对象肯定不相等
    // 注意: 如果你信任基类正确实现了 Equals,
    // 那么下面这一行可以删除
    if (obj == null) return false;

    // 如果对象属于不同的类型, 肯定不相等
    // 注意: 如果你信任基类正确实现了 Equals,
    // 那么下面这一行可以删除
    if (this.GetType() != obj.GetType()) return false;

    // 将 obj 转型为我们的类型以访问字段
    // 注意: 转型不可能失败, 因为我们知道
    // 两个对象是相同的类型
    AccountsAttribute other = (AccountsAttribute)obj;

    // 比较字段, 判断它们是否有相同的值
    // 这个例子判断'this'的帐户是不是
    // 与 other 的帐户相同
    if (other.m_accounts != m_accounts)
        return false;

    return true; // 对象相等
}

// 重写 GetHashCode, 因为我们重写了 Equals
public override Int32 GetHashCode() {
    return (Int32)m_accounts;
}

}

[Accounts(Accounts.Savings)]
internal sealed class ChildAccount { }

[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
internal sealed class AdultAccount { }

```

```

public sealed class Program
{
    public static void Main()
    {
        CanWriteCheck(new ChildAccount());
        CanWriteCheck(new AdultAccount());

        // 只是为了演示在一个没有应用 AccountsAttribute 的类型上,
        // 方法也能正确地工作
        CanWriteCheck(new Program());
    }

    private static void CanWriteCheck(Object obj) {
        // 构造 attribute 类型的一个实例, 并把它初始化成
        // 我们要显式查找的内容
        Attribute checking = new AccountsAttribute(Accounts.Checking);

        // 构造应用于类型的特性实例
        Attribute validAccounts = Attribute.GetCustomAttribute(
            obj.GetType(), typeof(AccountsAttribute), false);

        // 如果向精英应用了特性, 而且特性指定了
        // "Checking"帐户, 表明该类型可以开支票
        if ((validAccounts != null) && checking.Match(validAccounts)) {
            Console.WriteLine("{0} types can write checks.", obj.GetType());
        } else {
            Console.WriteLine("{0} types can NOT write checks.", obj.GetType());
        }
    }
}

```

编译并运行这个应用程序, 会得到以下输出:

```

ChildAccount types can NOT write checks.
AdultAccount types can write checks.
Program types can NOT write checks.

```

18.6 检测定制特性时不创建从 Attribute 派生的对象

本节将讨论如何利用另一种技术检测应用于元数据记录项的特性。在某些安全性要求严格的场合, 这个技术能保证不执行从 **Attribute** 派生的类中的代码。毕竟, 调用 **Attribute** 的 **GetCustomAttribute** 或者 **GetCustomAttributes** 方法时, 这些方法会在内部调用特性类的构造器, 而且可能调用属性的 **set** 访问器方法。此外, 首次访问类型会造成 CLR 调用类型的类型构造器(如果有的话)。在构造器、**set** 访问器方法以及类型构造器中, 可能包含每次查找特性都要执行的代码。这就相当于允许未知代码在 **AppDomain** 中运行, 所以存在安全隐患。

可用 **System.Reflection.CustomAttributeData** 类在查找特性的同时禁止执行特性类中的代码。该类定义了静态方法 **GetCustomAttributes** 来获取与目标关联的特性。方法有 4 个重载版本, 分别获取一个 **Assembly**, **Module**, **ParameterInfo** 和 **MemberInfo**。该类在 **System.Reflection** 命名空间(将在第 23 章“程序集加载和反射”讨论)中定义。通常, 先用 **Assembly** 的静态方法 **ReflectionOnlyLoad**(也在第 23 章讨论)加载程序集, 再用 **CustomAttributeData** 类分析这个程序集的元数据中的特性。简单地说, **ReflectionOnlyLoad**

以特殊方式加载程序集，期间会禁止 CLR 执行程序集中的任何代码；其中包括类型构造器。

CustomAttributeData 的 **GetCustomAttributes** 方法是一个工厂(factory)方法。也就是说，调用它会返回一个 **ICollection<CustomAttributeData>** 类型的对象，其中包含了由 **CustomAttributeData** 对象构成的集合。集合中的每个元素都是应用于指定目标的一个定制特性。可查询每个 **CustomAttributeData** 对象的只读属性，判断特性对象如何构造和初始化。具体地说，**Constructor** 属性指出构造器方法将如何调用。**ConstructorArguments** 属性以一个 **ICollection<CustomAttributeTypedArgument>** 实例的形式返回将传给这个构造器的实参。而 **NamedArguments** 属性以一个 **ICollection<CustomAttributeNamedArgument>** 实例的形式，返回将设置的字段/属性。注意，之所以说“将”，是因为不会实际地调用构造器和 set 访问器方法。禁止执行特性类的任何方法增强了安全性。

下面是之前例子的修改版本，它利用 **CustomAttributeData** 类来安全地获取应用于各个目标的特性：

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Collections.Generic;
using System.Linq;

[assembly: CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name = "Jeff", Target = typeof(Program))]
public sealed class Program
{
    [Conditional("Debug")]
    [Conditional("Release")]
    public void DoSomething() { }

    public Program() { }

    [CLSCompliant(true)]
    [STAThread]
    public static void Main()
    {
        // 显示应用于这个类型的特性集
        ShowAttributes(typeof(Program));

        // 获取与类型关联的方法集
        var members =
            from m in typeof(Program).GetTypeInfo().DeclaredMembers.OfType<MethodInfo>()
            where m.IsPublic
            select m;

        foreach (MemberInfo member in members) {
            // 显示应用于这个成员的特性集
            ShowAttributes(member);
        }

        private static void ShowAttributes(MemberInfo attributeTarget) {
            ICollection<CustomAttributeData> attributes =
                CustomAttributeData.GetCustomAttributes(attributeTarget);

            Console.WriteLine("Attributes applied to {0}: {1}",
```

```

        attributeTarget.Name, (attributes.Count == 0 ? "None" : String.Empty));

    foreach (CustomAttributeData attribute in attributes) {
        // 显示所应用的每个特性的类型
        Type t = attribute.Constructor.DeclaringType;
        Console.WriteLine(" {0}", t.ToString());
        Console.WriteLine("    Constructor called={0}", attribute.Constructor);

        IList<CustomAttributeTypedArgument> posArgs = attribute.ConstructorArguments;
        Console.WriteLine("    Positional arguments passed to constructor:" +
            ((posArgs.Count == 0) ? " None" : String.Empty));
        foreach (CustomAttributeTypedArgument pa in posArgs) {
            Console.WriteLine("        Type={0}, Value={1}", pa.ArgumentType, pa.Value);
        }

        IList<CustomAttributeNamedArgument> namedArgs = attribute.NamedArguments;
        Console.WriteLine("    Named arguments set after construction:" +
            ((namedArgs.Count == 0) ? " None" : String.Empty));
        foreach (CustomAttributeNamedArgument na in namedArgs) {
            Console.WriteLine("        Name={0}, Type={1}, Value={2}",
                na.MemberInfo.Name, na.TypedValue.ArgumentType, na.TypedValue.Value);
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}
}

```

编译并运行上述应用程序，将获得以下输出：

```

Attributes applied to Program:
System.SerializableAttribute
    Constructor called=Void .ctor()
    Positional arguments passed to constructor: None
    Named arguments set after construction: None

```

```

System.Diagnostics.DebuggerDisplayAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
        Type=System.String, Value=Richter
    Named arguments set after construction:
        Name=Name, Type=System.String, Value=Jeff
        Name=Target, Type=System.Type, Value=Program

```

```

System.Reflection.DefaultMemberAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
        Type=System.String, Value=Main
    Named arguments set after construction: None

```

```

Attributes applied to DoSomething:
System.Diagnostics.ConditionalAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
        Type=System.String, Value=Release
    Named arguments set after construction: None

```

```

System.Diagnostics.ConditionalAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
        Type=System.String, Value=Debug
    Named arguments set after construction: None

```

Attributes applied to Main:

```

System.STAThreadAttribute
  Constructor called=Void .ctor()
  Positional arguments passed to constructor: None
  Named arguments set after construction: None

System.CLSCompliantAttribute
  Constructor called=Void .ctor(Boolean)
  Positional arguments passed to constructor:
    Type=System.Boolean, Value=True
  Named arguments set after construction: None
Attributes applied to .ctor: None

```

18.7 条件特性类

定义、应用和反射特性能带来许多便利，所以开发人员越来越频繁地使用这些技术。特性简化了对代码的注释，还能实现丰富的功能。近来，开发人员越来越喜欢在设计和调试期间利用特性来辅助开发。例如，Microsoft Visual Studio 代码分析工具(FxCopCmd.exe)提供了一个 **System.Diagnostics.CodeAnalysis.SuppressMessageAttribute**，可将它应用于类型和成员，从而阻止报告特定的静态分析工具规则冲突(rule violation)。该特性仅对代码分析工具有用；程序平常运行时不会关注它。没有使用代码分析工具时，将 **SuppressMessage** 特性留在元数据中会使元数据无谓地膨胀，这会使文件变得更大，增大进程的工作集，损害应用程序的性能。假如有一种简单的方式，使编译器只有在使用代码分析工具时才生成 **SuppressMessage** 特性，结果会好很多。幸好，利用条件特性类真的能做到这一点。

应用了 **System.Diagnostics.ConditionalAttribute** 的特性类称为条件特性类。下面是一个例子：

```

// #define TEST
#define VERIFY

using System;
using System.Diagnostics;

[Conditional("TEST")] [Conditional("VERIFY")]
public sealed class CondAttribute : Attribute {
}

[Cond]
public sealed class Program {
    public static void Main() {
        Console.WriteLine("CondAttribute is {0} applied to Program type.",
            Attribute.IsDefined(typeof(Program),
                typeof(CondAttribute)) ? "" : "not ");
    }
}

```

编译器如果发现向目标元素应用了 **CondAttribute** 的实例，那么当含有目标元素的代码编译时，只有在定义 **TEST** 或 **VERIFY** 符号的前提下，编译器才会在元数据中生成特性信息。不过，特性类的定义元数据和实现仍存在于程序集中。

第 19 章 可空值类型

本章内容:

- C#对可空值类型的支持
- C#的空接合操作符
- CLR 对可空值类型的特殊支持

我们知道值类型的变量永远不会为 **null**；它总是包含值类型的值本身。事实上，这正是“值类型”一词的由来。遗憾的是，这在某些情况下会成为问题。例如，设计数据库时，可将一个列的数据类型定义成一个 32 位整数，并映射到 FCL(Framework Class Library)的 **Int32** 数据类型。但是，数据库中的一个列可能允许值为空；也就是说，该列在某一行上允许没有任何值。用 Microsoft .NET Framework 处理数据库数据可能变得很困难，因为在 CLR 中，没有办法将 **Int32** 值表示成 **null**。



注意 Microsoft ADO.NET 的表适配器(table adapter)确实支持可空类型。遗憾的是 **System.Data.SqlTypes** 命名空间中的类型没有用可空类型替换，部分原因是类型之间没有“一对一”的对应关系。例如，**SqlDecimal** 类型最大允许 38 位数，而普通的 **Decimal** 类型最大只允许 29 位数。此外，**SqlString** 类型支持它自己的本地化和比较选项，而普通的 **String** 类型并不支持这些。

下面是另一个例子：Java 的 **java.util.Date** 类是引用类型，所以该类型的变量能设为 **null**。但 CLR 的 **System.DateTime** 是值类型，**DateTime** 变量永远不能设为 **null**。如果用 Java 写的一个应用程序想和运行 CLR 的 Web 服务交流日期/时间，那么一旦 Java 程序发送 **null**，就会出问题，因为 CLR 不知道如何表示 **null**，也不知道如何操作它。

为了解决这个问题，Microsoft 在 CLR 中引入了可空值类型的概念。为了理解它们是如何工作的，先来看看 FCL 中定义的 **System.Nullable<T>** 结构。以下是 **System.Nullable<T>** 定义的逻辑表示：

```
[Serializable, StructLayout(LayoutKind.Sequential)]
public struct Nullable<T> where T : struct {

    // 这两个字段表示状态
    private Boolean hasValue = false;    // 假定 null
    internal T value = default(T);      // 假定所有位都为零
```

```

public Nullable(T value) {
    this.value = value;
    this.HasValue = true;
}

public Boolean HasValue { get { return hasValue; } }

public T Value {
    get {
        if (!hasValue) {
            throw new InvalidOperationException(
                "Nullable object must have a value.");
        }
        return value;
    }
}

public T GetValueOrDefault() { return value; }

public T GetValueOrDefault(T defaultValue) {
    if (!HasValue) return defaultValue;
    return value;
}

public override Boolean Equals(Object other) {
    if (!HasValue) return (other == null);
    if (other == null) return false;
    return value.Equals(other);
}

public override int GetHashCode() {
    if (!HasValue) return 0;
    return value.GetHashCode();
}

public override string ToString() {
    if (!HasValue) return "";
    return value.ToString();
}

public static implicit operator Nullable<T>(T value) {
    return new Nullable<T>(value);
}

public static explicit operator T(Nullable<T> value) {
    return value.Value;
}

```

可以看出，该结构能表示可为 **null** 的值类型。由于 **Nullable<T>** 本身是值类型，所以它的实例仍然是“轻量级”的。也就是说，实例仍然可以在栈上，而且实例的大小和原始值类型基本一样，只是多了一个 **Boolean** 字段。注意 **Nullable** 的类型参数 **T** 被约束为 **struct**。这是由于引用类型的变量本来就可以为 **null**，所以没必要再去照顾它。

现在，要在代码中使用一个可空的 **Int32**，就可以像下面这样写：

```

Nullable<Int32> x = 5;
Nullable<Int32> y = null;
Console.WriteLine("x: HasValue={0}, Value={1}", x.HasValue, x.Value);
Console.WriteLine("y: HasValue={0}, Value={1}", y.HasValue, y.GetValueOrDefault());

```

编译并运行上述代码，将获得以下输出：

```
x: HasValue=True, Value=5
y: HasValue=False, Value=0
```

19.1 C#对可空值类型的支持

C#允许使用相当简单的语法初始化上述两个 `Nullable<Int32>` 变量 **x** 和 **y**。事实上，C#开发团队的目的是将可空值类型集成到 C# 语言中，使之成为“一等公民”。为此，C#提供了更清晰的语法来处理可空值类型。C#允许用问号表示法来声明并初始化 **x** 和 **y** 变量：

```
Int32? x = 5;
Int32? y = null;
```

在 C# 中，`Int32?` 等价于 `Nullable<Int32>`。但 C# 在此基础上更进一步，允许开发人员在可空实例上执行转换和转型^①。C# 还允许向可空实例应用操作符。以下代码对此进行了演示：

```
private static void ConversionsAndCasting() {
    // 从非可空 Int32 隐式转换为 Nullable<Int32>
    Int32? a = 5;

    // 从 'null' 隐式转换为 Nullable<Int32>
    Int32? b = null;

    // 从 Nullable<Int32> 显式转换为非可空 Int32
    Int32 c = (Int32) a;

    // 在可空基元类型之间转型
    Double? d = 5;        // Int32 转型为 Double? (d 是 double 值 5.0)
    Double? e = b;        // Int32? 转型为 Double? (e 是 null)
}
```

C# 还允许向可空实例应用操作符，例如：

```
private static void Operators() {
    Int32? a = 5;
    Int32? b = null;

    // 一元操作符 (+ ++ - -- ! ~)
    a++;        // a = 6
    b = -b;     // b = null

    // 二元操作符 (+ - * / % & | ^ << >>)
    a = a + 3;   // a = 9
    b = b * 3;   // b = null;

    // 相等性操作符 (== !=)
    if (a == null) { /* no */ } else { /* yes */ }
    if (b == null) { /* yes */ } else { /* no */ }
    if (a != b) { /* yes */ } else { /* no */ }

    // 比较操作符 (< > <= >=)
    if (a < b)   { /* no */ } else { /* yes */ }
}
```

下面总结了 C# 如何解析操作符。

① 作者在这里区分了转换和转型。例如，从 `Int32` 的可空版本到非可空版本(或相反)，称为“转换”。但是，涉及不同基元类型的转换，就称为“转型”或“强制类型转换”。——译注

- 一元操作符(+, ++, -, --, !, ~)

操作数是 **null**, 结果就是 **null**。

- 二元操作符(+, -, *, /, %, &, |, ^, <<, >>)

两个操作数任何一个为 **null**, 结果就是 **null**。但有一个例外, 它发生在将 **&** 和操作符应用于 **Boolean?** 操作数的时候。在这种情况下, 两个操作符的行为和 SQL 的三值逻辑一样。对于这两个操作符, 如果两个操作数都不是 **null**, 那么操作符和平常一样工作。如果两个操作数都是 **null**, 结果就是 **null**。特殊行为仅在其中之一为 **null** 时发生。下表列出了针对操作数的 **true**, **false** 和 **null** 三个值的各种组合, 两个操作符的求值情况。

操作数 1 → 操作数 2 ↓	true	false	null
true	& = true = true	& = false = true	& = null = true
false	& = false = true	& = false = false	& = false = null
null	& = null = true	& = false = null	& = null = null

- 相等性操作符(==, !=)

两个操作数都是 **null**, 两者相等。一个操作数是 **null**, 两者不相等。两个操作数都不是 **null**, 就比较值来判断是否相等。

- 关系操作符(<, >, <=, >=)

两个操作数任何一个为 **null**, 结果就是 **false**。两个操作数都不是 **null**, 就比较值。

注意, 操作可空实例会生成大量代码。例如以下方法:

```
private static Int32? NullableCodeSize(Int32? a, Int32? b) {
    return (a + b);
}
```

编译这个方法会生成相当多的 IL 代码, 而且操作可空类型的速度慢于非可空类型。编译器生成的 IL 代码等价于以下 C# 代码:

```
private static Nullable<Int32> NullableCodeSize(
    Nullable<Int32> a, Nullable<Int32> b) {
    Nullable<Int32> nullable1 = a;
    Nullable<Int32> nullable2 = b;
    if (!nullable1.HasValue & nullable2.HasValue) {
        return new Nullable<Int32>();
    }
    return new Nullable<Int32>(
        nullable1.GetValueOrDefault() + nullable2.GetValueOrDefault());
}
```

最后要说明的是, 可定义自己的值类型来重载上述各种操作符。8.4 节“操作符重载方法”已对此进行了讨论。使用自己的值类型的可空实例, 编译器能正确识别它并调用你重载的

操作符(方法)。以下 **Point** 值类型重载了 **=** 和 **!=** 操作符:

```
using System;

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

    public static Boolean operator==(Point p1, Point p2) {
        return (p1.m_x == p2.m_x) && (p1.m_y == p2.m_y);
    }

    public static Boolean operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }
}
```

然后可以使用 **Point** 类型的可空实例, 编译器能自动调用你重载的操作符(方法):

```
internal static class Program {
    public static void Main() {
        Point? p1 = new Point(1, 1);
        Point? p2 = new Point(2, 2);

        Console.WriteLine("Are points equal? " + (p1 == p2).ToString());
        Console.WriteLine("Are points not equal? " + (p1 != p2).ToString());
    }
}
```

生成并运行上述代码得到以下输出:

```
Are points equal? False
Are points not equal? True
```

19.2 C#的空接合操作符

C#提供了一个“空接合操作符”(null-coalescing operator), 即**??**操作符, 它要获取两个操作数。假如左边的操作数不为 **null**, 就返回这个操作数的值。如果左边的操作数为 **null**, 就返回右边的操作数的值。利用空接合操作符, 可以方便地设置变量的默认值。

空接合操作符的一个好处在于, 它既能用于引用类型, 也能用于可空值类型。以下代码演示了如何使用**??**操作符:

```
private static void NullCoalescingOperator() {
    Int32? b = null;

    // 下面这行等价于:
    // x = (b.HasValue) ? b.Value : 123
    Int32 x = b ?? 123;
    Console.WriteLine(x); // "123"

    // 下面这行等价于:
    // String temp = GetFilename();
    // filename = (temp != null) ? temp : "Untitled";
    String filename = GetFilename() ?? "Untitled";
}
```

有人争辩说**??**操作符不过是**?:**操作符的“语法糖”而已, 所以 C#编译器团队不应该将这个操作符添加到语言中。实际上, **??**提供了重大的语法上的改进。第一个改进是**??**操作符能

更好地支持表达式:

```
Func<String> f = () => SomeMethod() ?? "Untitled";
```

相比下一行代码, 上述代码更容易阅读和理解。下面这行代码要求进行变量赋值, 而且用一个语句还搞不定:

```
Func<String> f = () => { var temp = SomeMethod();  
return temp != null ? temp : "Untitled";};
```

第二个改进是??在复合情形中更好用。例如, 下面这行代码:

```
String s = SomeMethod1() ?? SomeMethod2() ?? "Untitled";
```

它比下面这一堆代码更容易阅读和理解:

```
String s;  
var sm1 = SomeMethod1();  
if (sm1 != null) s = sm1;  
else {  
    var sm2 = SomeMethod2();  
    if (sm2 != null) s = sm2;  
    else s = "Untitled";  
}
```

19.3 CLR 对可空值类型的特殊支持

CLR 内建对可空值类型的支持。这个特殊的支持是针对装箱、拆箱、调用 **GetType** 和调用接口方法提供的, 它使可空类型能无缝地集成到 CLR 中, 而且使它们具有更自然的行为, 更符合大多数开发人员的预期。下面深入研究一下 CLR 对可空类型的特殊支持。

19.3.1 可空值类型的装箱

假定有一个逻辑上设为 **null** 的 **Nullable<Int32>** 变量。将其传给期待一个 **Object** 的方法, 就必须对其进行装箱, 并将对已装箱 **Nullable<Int32>** 的引用传给方法。但对表面上为 **null** 的值进行装箱不符合直觉——即使 **Nullable<Int32>** 变量本身非 **null**, 它只是在逻辑上包含了 **null**。为了解决这个问题, CLR 会在装箱可空变量时执行一些特殊代码, 从表面上维持可空类型的“一等公民”地位。

具体地说, 当 CLR 对 **Nullable<T>** 实例进行装箱时, 会检查它是否为 **null**。如果是, CLR 不装箱任何东西, 直接返回 **null**。如果可空实例不为 **null**, CLR 从可空实例中取出值并进行装箱。也就是说, 一个值为 5 的 **Nullable<Int32>** 会装箱成值为 5 的已装箱 **Int32**。以下代码演示了这一行为:

```
// 对 Nullable<T> 进行装箱, 要么返回 null, 要么返回一个已装箱的 T  
Int32? n = null;  
Object o = n; // o 为 null  
Console.WriteLine("o is null={0}", o == null); // "True"  
  
n = 5;  
o = n; // o 引用一个已装箱的 Int32  
Console.WriteLine("o's type={0}", o.GetType()); // "System.Int32"
```

19.3.2 可空值类型的拆箱

CLR 允许将已装箱的值类型 **T** 拆箱为一个 **T** 或者 **Nullable<T>**。如果对已装箱值类型的引用是 **null**, 而且要把它拆箱为一个 **Nullable<T>**, 那么 CLR 会将 **Nullable<T>** 的值设为 **null**。以下代码演示了这个行为:

```
// 创建已装箱的 Int32
Object o = 5;

// 把它拆箱为一个 Nullable<Int32> 和一个 Int32
Int32? a = (Int32?) o; // a = 5
Int32 b = (Int32) o;    // b = 5

// 创建初始化为 null 的一个引用
o = null;

// 把它"拆箱"为一个 Nullable<Int32> 和一个 Int32
a = (Int32?) o; // a = null
b = (Int32) o;  // NullReferenceException
```

19.3.3 通过可空值类型调用 GetType

在 **Nullable<T>** 对象上调用 **GetType**, CLR 实际会“撒谎”说类型是 **T**, 而不是 **Nullable<T>**。以下代码演示了这一行为:

```
Int32? x = 5;

// 下面这行会显示 "System.Int32", 而非 "System.Nullable<Int32>"
Console.WriteLine(x.GetType());
```

19.3.4 通过可空值类型调用接口方法

以下代码将 **Nullable<Int32>** 类型的变量 **n** 转型为接口类型 **IComparable<Int32>**。**Nullable<T>** 不像 **Int32** 那样实现了 **IComparable<Int32>** 接口, 但 C# 编译器允许这样的代码通过编译, 而且 CLR 的校验器也认为这样的代码可验证, 从而允许使用更简洁的语法:

```
Int32? n = 5;
Int32 result = ((IComparable) n).CompareTo(5); // 能顺利编译和运行
Console.WriteLine(result);                      // 0
```

假如 CLR 不提供这一特殊支持, 要在可空值类型上调用接口方法, 就必须写很繁琐的代码。首先要转型为已拆箱的值类型, 然后才能转型为接口以发出调用:

```
Int32 result = ((IComparable) (Int32) n).CompareTo(5); // 很繁琐
```

陳敬衡與陳敬堂父子 (續前)

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。陳敬堂在文中提到，陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係 (續前)

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

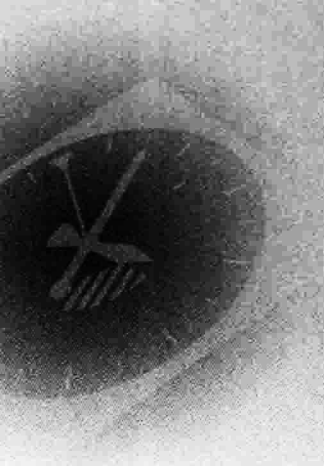
陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係 (續前)

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

陳敬衡與陳敬堂父子關係，可從陳敬堂於1911年所寫的「陳敬衡與陳敬堂父子關係」一文中得知。

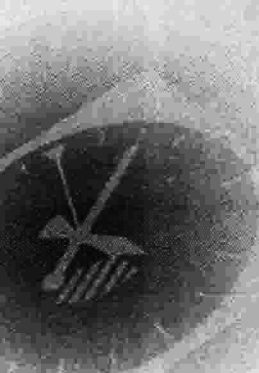


第IV部分 核心机制

- ▶ 第 20 章 异常和状态管理
- ▶ 第 21 章 托管堆和垃圾回收
- ▶ 第 22 章 CLR 寄宿和 AppDomain
- ▶ 第 23 章 程序集加载和反射
- ▶ 第 24 章 运行时序列化
- ▶ 第 25 章 与 WinRT 组件互操作

藥VI部 避心肝藥

避心肝藥之作用	第 12 章	4
避心肝藥之適應症	第 13 章	4
避心肝藥之禁忌症	第 14 章	4
避心肝藥之副作用	第 15 章	4
避心肝藥之用法	第 16 章	4
避心肝藥之貯藏	第 17 章	4
避心肝藥之包裝	第 18 章	4



第 20 章 异常和状态管理

本章内容：

- 定义“异常”
- 异常处理机制
- **System.Exception** 类
- FCL 定义的异常类
- 抛出异常
- 定义自己的异常类
- 用可靠性换取开发效率
- 设计规范和最佳实践
- 未处理的异常
- 对异常进行调试
- 异常处理的性能问题
- 约束执行区域(CER)
- 代码协定

本章重点在于错误处理，但并非仅限于此。错误处理要分几个部分。首先要定义到底什么是错误。然后要讨论如何判断代码正在经历一个错误，以及如何从错误中恢复。这个时候，状态就成为一个要考虑的问题，因为错误常常在不恰当的时候发生。代码可能在状态改变的中途发生错误。这时需要将一些状态还原为改变之前的样子。当然，还要讨论代码如何通知调用者有错误发生。

在我看来，异常处理是 CLR 最薄弱的一个环节，造成开发人员在写托管代码时遇到许多问题。经过多年的发展，Microsoft 确实进行了一系列显著的改进来帮助开发人员处理错误。但我认为在获得一个真正良好、可靠的系统之前，Microsoft 仍有大量工作要做。针对未处理的异常、约束执行区域(constrained execution region, CER)、代码协定、运行时包装的异常以及未捕捉的异常，本章要讨论处理它们时的改进。

20.1 定义“异常”

设计类型时要想好各种使用情况。类型名称通常是名词，例如 **FileStream** 或者 **StringBuilder**。然后要为类型定义属性、方法、事件等。这些成员的定义方式(属性的数据

类型、方法的参数、返回值等)就是类型的编程接口。这些成员代表类本身或者类型实例能执行的行动。行动成员通常用动词表示,例如 **Read**, **Write**, **Flush**, **Append**, **Insert** 和 **Remove** 等。当行动成员不能完成任务时,就应抛出异常。



重要提示 异常是指成员没有完成它的名称所宣称的行动。

例如以下类定义:

```
internal class Account {
    public static void Transfer(Account from, Account to, Decimal amount) {
        from -= amount;
        to += amount;
    }
}
```

Transfer 方法接受两个 **Account** 对象和一个代表账号之间转账金额的 **Decimal** 值。显然, **Transfer** 方法的作用是从一个账户扣除钱,把钱添加到另一个账户中。**Transfer** 方法可能因为多种原因而失败。例如, **from** 或 **to** 实参可能为 **null**; **from** 或 **to** 实参引用的可能不是活动账户; **from** 账户可能没有足够的资金; **to** 账户的资金可能过多,以至于增加资金时导致账户溢出; **amount** 实参为 0、负数或者小数超过两位。

Transfer 方法在调用时,它的代码应检查前面描述的种种可能。检测到其中任何一种可能都不能转账,应抛出异常来通知调用者它不能完成任务。事实上, **Transfer** 方法的返回类型为 **void**。这是由于 **Transfer** 方法没有什么有意义的值需要返回。这一点很容易想得通:方法正常返回^①表明转账成功,失败就抛出一个有意义的异常。

面向对象编程极大提高了开发人员的开发效率,因为可以写这样的代码:

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E"); // true
```

这行代码将多个操作链接到一起^②。我很容易写这行代码,其他人也很容易阅读和维护,因为它的意图很明显:获取字符串,取出一部分,全部大写那个部分,然后检查那个部分是否以"E"结尾。出发点不错,但有一个重要的前提:没有操作失败,中途不出错。但错误总是可能发生的,所以需要以一种方式处理错误。事实上,许多面向对象的构造——构造器、获取和设置属性、添加和删除事件、调用操作符重载和调用转换操作符等——都没办法返回错误代码,但它们仍然需要报告错误。Microsoft .NET Framework 和所有编程语言通过异常处理来解决这个问题。

① 指返回到调用位置,而不是返回一个值。——译注

② 事实上,利用 C# 的“扩展方法”,可以将更多本来不能链接的方法链接到一起。



重要提示 许多开发人员都错误地认为异常和某件事情的发生频率有关。例如，一个设计文件 **Read** 方法的开发人员可能会这样想：“读取文件最终会抵达文件尾。由于抵达文件尾总是会发生，所以我设计这个 **Read** 方法返回一个特殊值来报告抵达了文件尾；我不让它抛出异常。”问题在于，这是设计 **Read** 方法的开发人员的想法，而非调用 **Read** 方法的开发人员的想法。

设计 **Read** 方法的开发人员不可能知道这个方法的所有调用情形。所以，开发人员不可能知道 **Read** 的调用者是不是每次都会一路读取到文件尾。事实上，由于大多数文件包含的都是结构化数据，所以一路读取直至文件尾的情况是很少发生的。

20.2 异常处理机制

本节介绍异常处理机制，以及进行异常处理所需的 C# 构造，但不打算罗列过多的细节。本章旨在提供何时以及如何使用异常处理的设计规范。要更多地了解异常处理机制和相关的 C# 语言构造，请参考文档和 C# 语言规范。另外，.NET Framework 异常处理机制是用 Microsoft Windows 提供的结构化异常处理 (Structured Exception Handling, SEH) 机制构建的。对 SEH 的讨论有很多，包括我自己的《Windows 核心编程 (第 5 版)》一书，其中有 3 章内容专门讨论 SEH。

以下 C# 代码展示了异常处理机制的标准用法，可通过它对异常处理代码块及其用途产生初步认识。代码后面的各小节将正式描述 **try**、**catch** 和 **finally** 块及其用途，并提供关于它们的一些注意事项。

```
private void SomeMethod() {
    try {
        // 需要得体地进行恢复和/或清理的代码放在这里
    }
    catch (InvalidOperationException) {
        // 从 InvalidOperationException 恢复的代码放在这里
    }
    catch (IOException) {
        // 从 IOException 恢复的代码放在这里
    }
    catch {
        // 从除了上述异常之外的其他所有异常恢复的代码放在这里
        ...
        // 如果什么异常都捕捉，通常要重新抛出异常。本章稍后将详细解释
        throw;
    }
    finally {
        // 这里的代码对始于 try 块的任何操作进行清理
        // 这里的代码总是执行，不管是不是抛出了异常
    }
    // 如果 try 块没有抛出异常，或者某个 catch 块捕捉到异常，但没有抛出或
```

```
// 重新抛出异常，就执行下面的代码
```

```
...
```

这段代码只是使用各种异常处理块的一种可能的方式。不要被这些代码吓到——大多数方法都只有一个 **try** 块和一个匹配 **finally** 块，或者一个 **try** 块和一个匹配的 **catch** 块。像本例那样有这么多 **catch** 块是很少见的，这里列出它们仅仅是为了演示。

20.2.1 try 块

如果代码需要执行一般性的资源清理操作，需要从异常中恢复，或者两者都需要，就可以放到 **try** 块中。负责清理的代码应放到一个 **finally** 块中。**try** 块还可包含也许会抛出异常的代码。负责异常恢复的代码应放到一个或多个 **catch** 块中。针对应用程序能从中安全恢复的每一种异常，都应该创建一个 **catch** 块。一个 **try** 块至少要有有一个关联的 **catch** 块或 **finally** 块，单独一个 **try** 块没有意义，C#也不允许。



重要提示 开发人员有时不知道应该在一个 **try** 块中放入多少代码。这具体取决于状态管理。如果在一个 **try** 块中执行多个可能抛出同一个异常类型的操作，但不同的操作有不同的异常恢复措施，就应该将每个操作都放到它自己的 **try** 块中，这样才能正确地恢复状态。

20.2.2 catch 块

catch 块包含的是响应一个异常需要执行的代码。一个 **try** 块可以关联 0 个或多个 **catch** 块。如果 **try** 块中的代码没有造成异常的抛出，CLR 永远不会执行它的任何 **catch** 块。线程将跳过所有 **catch** 块，直接执行 **finally** 块(如果有的话)。**finally** 块执行完毕后，从 **finally** 块后面的语句继续执行。

catch 关键字后的圆括号中的表达式称为**捕捉类型**。C#要求捕捉类型必须是 **System.Exception** 或者它的派生类型。例如，上述代码包含用于处理 **InvalidOperationException** 异常(或者从它派生的任何异常)和 **IOException** 异常(或者从它派生的任何异常)的 **catch** 块。最后一个 **catch** 块没有指定捕捉类型，能处理除了前面的 **catch** 块指定的之外的其他所有异常；这相当于捕捉 **System.Exception** (只是 **catch** 块大括号中的代码访问不了异常信息)。



注意 用 Microsoft Visual Studio 调试 **catch** 块时，可在监视窗口中添加特殊变量名称 **\$exception** 来查看当前抛出的异常对象。

CLR 自上而下搜索匹配的 **catch** 块，所以应该将较具体的异常放在顶部。也就是说，首先出现的是派生程度最大的异常类型，接着是它们的基类型(如果有的话)，最后是 **System.Exception**(或者没有指定任何捕捉类型的 **catch** 块)。事实上，如果弄反了这个顺序，

将较具体的 **catch** 块放在靠近底部的位置, C#编译器会报错, 因为这样的 **catch** 块是不可达的。

在 **try** 块的代码(或者从 **try** 块调用的任何方法)中抛出异常, CLR 将搜索捕捉类型与抛出的异常相同(或者是它的基类)的 **catch** 块。如果没有任何捕捉类型与抛出的异常匹配, CLR 会去调用栈^①更高的一层搜索与异常匹配的捕捉类型。如果都到了调用栈的顶部, 还是没有找到匹配的 **catch** 块, 就会发生未处理的异常。本章后面将更深入地探讨未处理的异常。

一旦 CLR 找到匹配的 **catch** 块, 就会执行内层所有 **finally** 块中的代码。所谓“内层 **finally** 块”是指从抛出异常的 **try** 块开始, 到匹配异常的 **catch** 块之间的所有 **finally** 块。^②注意, 匹配异常的那个 **catch** 块所关联的 **finally** 块尚未执行, 该 **finally** 块中的代码一直要等到这个 **catch** 块中的代码执行完毕才会执行。

所有内层 **finally** 块执行完毕之后, 匹配异常的那个 **catch** 块中的代码才开始执行。**catch** 块中的代码通常执行一些对异常进行处理的操作。在 **catch** 块的末尾, 我们有以下三个选择。

- 重新抛出相同的异常, 向调用栈高一层的代码通知该异常的发生。
- 抛出一个不同的异常, 向调用栈高一层的代码提供更丰富的异常信息。
- 让线程从 **catch** 块的底部退出^③。

本章稍后将针对每一种技术的使用时机提供一些指导方针。选择前两种技术将抛出异常, CLR 的行为和之前说的一样: 回溯调用栈, 查找捕捉类型与抛出的异常的类型匹配的 **catch** 块。

选择最后一种技术, 当线程从 **catch** 块的底部退出后, 它将立即执行包含在 **finally** 块^④ (如果有的话)中的代码。**finally** 块的所有代码执行完毕后, 线程退出 **finally** 块, 执行紧跟在 **finally** 块之后的语句。如果不存在 **finally** 块, 线程将从最后一个 **catch** 块之后的语句开始执行。

C#允许在捕捉类型后指定一个变量。捕捉到异常时, 该变量将引用抛出的 **System.Exception** 派生对象。**catch** 块的代码可通过引用该变量来访问异常的具体信息(例如异常发生时的堆栈跟踪^⑤)。虽然这个对象可以修改, 但最好不要这么做, 而应把它当成是只读的。本章稍后将解释 **Exception** 类型以及可以在该类型上进行哪些操作。

① 文档翻译成“调用堆栈”。——译注

② 前面说过, 异常处理涉及调用栈的知识。内层没有找到合适的 **catch**, 就跑去上一层。以此类推, 直至栈顶。不要产生“**finally** 块怎么跑到 **try** 和 **catch** 之间”的误解。请用“立体”思维来看待寻找合适 **catch** 的过程。——译注

③ 此退出(fall out of the bottom of the catch block)非彼退出。不是说要终止线程, 而是说执行正常地“贯穿”**catch** 块的底部, 并执行匹配的 **finally** 块。——译注

④ 这个才是与 **catch** 关联的 **finally** 块, 也就是常说的 **try-catch-finally** 中的 **finally**。

⑤ 对方法调用的跟踪称为堆栈跟踪(stack trace)。堆栈跟踪列表提供了一种循着调用序列跟踪到异常发生处的手段。另外要注意, 虽然本书大多数时候会将 **stack** 翻译成“栈”而不是“堆栈”, 但为了保持和文档及习惯说法的一致, 偶尔还是不得不将一些 **stack** 翻译成“堆栈”。——译注