第 1 章

.NET 体系结构

我们不能孤立地使用 C#语言，而必须和.NET Framework 一起考虑。C#编译器专门用于.NET，这表示用 C#编写的所有代码总是在.NET Framework 中运行。对于 C#语言来说，可以得出两个重要的结论：

- C#的结构和方法论反映了.NET 基础方法论。
- 在许多情况下，C#的特定语言功能取决于.NET 的功能，或依赖于.NET 基类。

由于这种依赖性，在开始使用 C#编程前，了解.NET 的结构和方法论就非常重要了，这就是本章的目的。

本章首先了解在.NET 编译和运行所有的代码(包括 C#)时通常会出现什么情况。对这些内容进行概述之后，就要详细阐述 Microsoft 中间语言(Microsoft Intermediate Language，MSIL 或简称为 IL)，所有编译好的代码都要使用这种语言。本章特别要介绍 IL、通用类型系统(Common Type System，CTS)及公共语言规范(Common Language Specification，CLS)如何提供.NET 语言之间的互操作性。最后解释各种语言如何使用.NET，包括 Visual Basic 和 C++。

之后，我们将介绍.NET 的其他特性，包括程序集、命名空间和.NET 基类。最后本章简要探讨一下 C#开发人员可以创建的应用程序类型。

1.1 C#与.NET 的关系

C#是一种相当新的编程语言，C#的重要性体现在以下两个方面：

- 它是专门为与 Microsoft 的.NET Framework 一起使用而设计的。(.NET Framework 是一个功能非常丰富的平台，可开发、部署和执行分布式应用程序)。
- 它是一种基于现代面向对象设计方法的语言，在设计它时，Microsoft 还吸取了其他类似语言的经验，这些语言是近 20 年来面向对象规则得到广泛应用后才开发出来的。

有一个很重要的问题要弄明白：C#就其本身而言只是一种语言，尽管它是用于生成面

向.NET 环境的代码，但它本身不是.NET 的一部分。.NET 支持的一些特性，C#并不支持。而 C#语言支持的另一些特性，.NET 却不支持(例如运算符重载)！

但是，因为 C#语言是和.NET 一起使用的，所以如果要使用 C#高效地开发应用程序，理解 Framework 就非常重要，所以本章将介绍.NET 的内涵。

1.2 公共语言运行库

.NET Framework 的核心是其运行库的执行环境，称为公共语言运行库(CLR)或.NET 运行库。通常将在 CLR 的控制下运行的代码称为托管代码(managed code)。

但是，在 CLR 执行编写好的源代码之前，需要编译它们(在 C#中或其他语言中)。在.NET 中，编译分为两个阶段：

(1) 把源代码编译为 Microsoft 中间语言(IL)。

(2) CLR 把 IL 编译为平台专用的代码。

这个两阶段的编译过程非常重要，因为 Microsoft 中间语言(托管代码)是提供.NET 的许多优点的关键。

托管代码的优点

Microsoft 中间语言与 Java 字节代码共享一种理念：它们都是低级语言，语法很简单(使用数字代码，而不是文本代码)，可以非常快速地转换为内部机器码。对于代码来说，这种精心设计的通用语法，有很重要的优点。

1. 平台无关性

首先，这意味着包含字节代码指令的同一文件可以放在任一平台中，运行时编译过程的最后阶段可以很容易完成，这样代码就可以运行在特定的平台上。换言之，编译为中间语言就可以获得.NET 平台无关性，这与编译为 Java 字节代码就会得到 Java 平台无关性是一样的。

注意.NET 的平台无关性目前只是一种可能，因为在编写本书时，.NET 只能用于 Windows 平台，但人们正在积极准备，使它可以用于其他平台(参见 Mono 项目，它用于实现.NET 的开放源代码，参见 http://www.go-mono.com/)。

2. 提高性能

前面把 IL 和 Java 做了比较，实际上，IL 比 Java 字节代码的作用还要大。IL 总是即时编译的(称为 JIT 编译)，而 Java 字节代码常常是解释性的，Java 的一个缺点是，在运行应用程序时，把 Java 字节代码转换为内部可执行代码的过程会导致性能的损失(但在最近，Java 在某些平台上能进行 JIT 编译)。

JIT 编译器并不是把整个应用程序一次编译完(这样会有很长的启动时间)，而是只编译它调用的那部分代码(这是其名称由来)。代码编译过一次后，得到的内部可执行代码就存储起来，直到退出该应用程序为止，这样在下次运行这部分代码时，就不需要重新编译了。

Microsoft 认为这个过程要比一开始就编译整个应用程序代码的效率高得多，因为任何应用程序的大部分代码实际上并不是在每次运行过程中都执行。使用 JIT 编译器，从来都不会编译这种代码。

这解释了为什么托管 IL 代码的执行几乎和内部机器代码的执行速度一样快，但是并没有说明为什么 Microsoft 认为这会提高性能。其原因是编译过程的最后一部分是在运行时进行的，JIT 编译器确切地知道程序运行在什么类型的处理器上，可以利用该处理器提供的任何特性或特定的机器代码指令来优化最后的可执行代码。

传统的编译器会优化代码，但它们的优化过程是独立于代码所运行的特定处理器的。这是因为传统的编译器是在发布软件之前编译为内部机器可执行的代码。即编译器不知道代码所运行的处理器类型，例如该处理器是 x86 兼容处理器还是 Alpha 处理器，这超出了基本操作的范围。例如 Visual Studio 6 为一般的奔腾机器进行了优化，所以它生成的代码就不能利用奔腾 III 处理器的硬件特性。相反，JIT 编译器不仅可以进行 Visual Studio 6 所能完成的优化工作，还可以优化代码所运行的特定处理器。

3. 语言的互操作性

使用 IL 不仅支持平台无关性，还支持语言的互操作性。简而言之，就是能将任何一种语言编译为中间代码，编译好的代码可以与从其他语言编译过来的代码进行交互操作。

那么除了 C#之外，还有什么语言可以通过.NET 进行交互操作呢？下面就简要讨论其他常见语言如何与.NET 交互操作。

(1) Visual Basic 2005

Visual Basic 6 在升级到 Visual Basic .NET 2002 时，经历了一番脱胎换骨的变化，才集成到.NET Framework 的第一版中。Visual Basic 语言对 Visual Basic 6 进行了很大的演化，也就是说，Visual Basic 6 并不适合运行.NET 程序。例如，它与 COM 的高度集成，且只把事件处理程序作为源代码显示给开发人员，大多数后台代码不能用作源代码。另外，它不支持继承，Visual Basic 使用的标准数据类型也与.NET 不兼容。

Visual Basic 6 在 2002 年升级为 Visual Basic .NET，对 Visual Basic 进行的改变非常大，完全可以把 Visual Basic 当作是一种新语言。现有的 Visual Basic 6 代码不能编译为 Visual Basic 2005 代码(或 Visual Basic .NET 2002 和 2003 代码)，把 Visual Basic 6 程序转换为 Visual Basic 2005 时，需要对代码进行大量的改动，但大多数修改工作都可以由 Visual Studio 2005(Visual Studio 的升级版本，用于与.NET 一起使用)自动完成。如果把 Visual Basic 6 项目读到 Visual Studio 2005 中，Visual Studio 2005 就会升级该项目，也就是说把 Visual Basic 6 源代码重写为 Visual Basic 2005 源代码。虽然这意味着其中的工作已大大减轻，但用户仍需要检查新的 Visual Basic 2005 代码，以确保项目仍可正确工作，因为这种转换并不十分完美。

这种语言升级的一个副作用是不能再把 Visual Basic 2005 编译为内部可执行代码了。Visual Basic 2005 只编译为中间语言，就像 C#一样。如果需要继续使用 Visual Basic 6 编写程序，就可以这么做，但生成的可执行代码会完全忽略.NET Framework，如果继续把 Visual Studio 作为开发环境，就需要安装 Visual Studio 6。

(2) Visual C++ 2005

Visual C++ 6 有许多 Microsoft 对 Windows 的特定扩展。通过 Visual C++ .NET，又加入了更多的扩展内容，来支持.NET Framework。现有的 C++源代码会继续编译为内部可执行代码，不会有修改，但它会独立于.NET 运行库运行。如果让 C++代码在.NET Framework 中运行，就可以在代码的开头添加下述命令：

```
#using <mscorlib.dll>
```

还可以把标记/clr 传递给编译器，这样编译器假定要编译托管代码，因此会生成中间语言，而不是内部机器码。C++的一个有趣的问题是在编译托管代码时，编译器可以生成包含内嵌本机可执行代码的 IL。这表示在 C++代码中可以把托管类型和非托管类型合并起来，因此托管 C++代码：

```
class MyClass
{
```

定义了一个普通的 C++类，而代码：

```
__gc class MyClass
{
```

生成了一个托管类，就好像使用 C#或 Visual Basic 2005 编写类一样。实际上，托管 C++比 C#更优越的一点是可以在托管 C++代码中调用非托管 C++类，而不必采用 COM 交互功能。

如果在托管类型上试图使用.NET 不支持的特性(例如，模板或类的多继承)，编译器就会出现一个错误。另外，在使用托管类时，还需要使用非标准的 C++特性(例如上述代码中的__gc 关键字)。

因为 C++允许低级指针操作，C++编译器不能生成可以通过 CLR 内存类型安全测试的代码。如果 CLR 把代码标识为内存类型安全是非常重要的，就需要用其他一些语言编写源代码，例如 C# 或 Visual Basic 2005。

(3) Visual J# 2005

最新添加的语言是 Visual J# 2005。在.NET Framework 1.1 版本推出之前，用户必须下载相应的软件，才能使用 J#。现在 J#语言内置于.NET Framework 中。因此，J#用户可以利用 Visual Studio 2005 的所有常见特性。Microsoft 希望大多数 J++用户认为他们在使用.NET 时，将很容易使用 J#。J#不使用 Java 运行库，而是使用与其他.NET 兼容语言一样的基类库。这说明，与 C#和 Visual Basic 2005 一样，可以使用 J#创建 ASP.NET Web 应用程序、Windows 窗体、XML Web 服务和其他应用程序。

(4) 脚本语言

脚本语言仍在使用之中，但由于.NET 的推出，它们的重要性在降低。与此同时，JScript 升级到了 JScript.NET。现在 ASP.NET 页面可以用 JScript.NET 编写，可以把 JScript.NET 当作一种编译语言来运行，而不是解释性的语言，也可以编写强类型化的 JScript.NET 代码。有了 ASP.NET 后，就没有必要在服务器端的 Web 页面上使用脚本语言了，但 VBA 仍用作 Office 文档和 Visual Studio 宏语言。

(5) COM 和 COM+

从技术上讲，COM 和 COM+并不是面向.NET 的技术，因为基于它们的组件不能编译为 IL(但如果原来的 COM 组件是用 C++编写的，使用托管 C++，在某种程度上可以这么做)。但是，COM+仍然是一个重要的工具，因为其特性没有在.NET 中完全实现。另外，COM 组件仍可以使用——.NET 组合了 COM 的互操作性，从而使托管代码可以调用 COM 组件，COM 组件也可以调用托管代码(见第 33 章)。在一般情况下，把新组件编写为.NET 组件，大多是为了方便，因为这样可以利用.NET 基类和托管代码的其他优点。

1.3 中间语言

通过前面的学习，我们理解了 Microsoft 中间语言显然在.NET Framework 中有非常重要的作用。C#开发人员应明白，C#代码在执行前要编译为中间语言(实际上，C#编译器仅编译为托管代码)，这是有意义的，现在应详细讨论一下 IL 的主要特征，因为面向.NET 的所有语言在逻辑上都需要支持 IL 的主要特征。

下面就是中间语言的主要特征：

- 面向对象和使用接口
- 值类型和引用类型之间的巨大差别
- 强数据类型
- 使用异常来处理错误
- 使用特性(attribute)

下面详细讨论这些特征。

1.3.1 面向对象和接口的支持

.NET 的语言无关性还有一些实际的限制。中间语言在设计时就打算实现某些特殊的编程方法，这表示面向它的语言必须与编程方法兼容，Microsoft 为 IL 选择的特定道路是传统的面向对象的编程，带有类的单一继承性。

注意：

不熟悉面向对象概念的读者应参考附录 A，获得更多的信息。附录 A 可以从 www.wrox.com 上下载。

除了传统的面向对象编程外，中间语言还引入了接口的概念，它们显示了在带有 COM 的 Windows 下的第一个实现方式。.NET 接口与 COM 接口不同，它们不需要支持任何 COM 基础结构，例如，它们不是派生自 IUnknown，也没有对应的 GUID。但它们与 COM 接口共享下述理念：提供一个契约，实现给定接口的类必须提供该接口指定的方法和属性的实现方式。

前面介绍了使用.NET 意味着要编译为中间语言，即需要使用传统的面向对象的方法来编程。但这并不能提供语言的互操作性。毕竟，C++和 Java 都使用相同的面向对象的范型，但它们仍不是可交互操作的语言。下面需要详细探讨一下语言互操作性的概念。

首先，需要确定一下语言互操作性的含义。毕竟，COM 允许以不同语言编写的组件一起工作，即可以调用彼此的方法。这就足够了吗？COM 是一个二进制标准，允许组件实例化其他组件，调用它们的方法或属性，而无需考虑编写相关组件的语言。但为了实现这个功能，每个对象都必须通过 COM 运行库来实例化，通过接口来访问。根据相关组件的线程模型，不同线程上内存空间和运行组件之间要编组数据，这还可能造成很大的性能损失。在极端情况下，组件保存为可执行文件，而不是 DLL 文件，还必须创建单独的进程来运行它们。重要的是组件要能与其他组件通信，但仅通过 COM 运行库进行通信。无论 COM 是用于允许使用不同语言的组件直接彼此通信，或者创建彼此的实例，系统都把 COM 作为中间件来处理。不仅如此，COM 结构还不允许利用继承实现，即它丧失了面向对象编程的许多优势。

一个相关的问题是，在调试时，仍必须单独调试用不同语言编写的组件。这样就不可能在调试器上调试不同语言的代码了。语言互操作性的真正含义是用一种语言编写的类应能直接与用另一种语言编写的类通信。特别是：

- 用一种语言编写的类应能继承用另一种语言编写的类。
- 一个类应能包含另一个类的实例，而不管它们是使用什么语言编写的。
- 一个对象应能直接调用用其他语言编写的另一个对象的方法。
- 对象(或对象的引用)应能在方法之间传递。
- 在不同的语言之间调用方法时，应能在调试器中调试这些方法调用，即调试不同语言编写的源代码。

这是一个雄心勃勃的目标，但令人惊讶的是，.NET 和中间语言已经实现了这个目标。在调试器上调试方法时，Visual Studio 2005 IDE 提供了这样的工具(不是 CLR 提供的)。

1.3.2 相异值类型和引用类型

与其他编程语言一样，中间语言提供了许多预定义的基本数据类型。它的一个特性是值类型和引用类型有明显的区别。对于值类型，变量直接保存其数据，而对于引用类型，变量仅保存地址，对应的数据可以在该地址中找到。

在 C++中，引用类型类似于通过指针来访问变量，而在 Visual Basic 中，与引用类型最相似的是对象，Visual Basic 6 总是通过引用来访问对象。中间语言也有数据存储的规范：引用类型的实例总是存储在一个名为“托管堆”的内存区域中，值类型一般存储在堆栈中(但如果值类型在引用类型中声明为字段，它们就内联存储在堆中)。第 2 章“C#基础”讨论堆栈和堆，及其工作原理。

1.3.3 强数据类型

中间语言的一个重要方面是它基于强数据类型。所有的变量都清晰地标记为属于某个特定数据类型(在中间语言中没有 Visual Basic 和脚本语言中的 Variant 数据类型)。特别是中间语言一般不允许对模糊的数据类型执行任何操作。

例如，Visual Basic 6 开发人员习惯于传递变量，而无需考虑它们的类型，因为 Visual

Basic 6 会自动进行所需的类型转换。C++开发人员习惯于在不同类型之间转换指针类型。执行这类操作将大大提高性能，但破坏了类型的安全性。因此，这类操作只能在某些编译为托管代码的语言中的特殊情况下进行。确实，指针(相对于引用)只能在标记了的 C#代码块中使用，但在 Visual Basic 中不能使用(但一般在托管 C++中允许使用)。在代码中使用指针会立即导致 CLR 提供的内存类型安全性检查失败。

注意，一些与.NET 兼容的语言，例如 Visual Basic 2005，在类型化方面的要求仍比较松，但这是可以的，因为编译器在后台确保在生成的 IL 上强制类型安全。

尽管强迫实现类型的安全性最初会降低性能，但在许多情况下，我们从.NET 提供的、依赖于类型安全的服务中获得的好处更多。这些服务包括：

- 语言的互操作性
- 垃圾收集
- 安全性
- 应用程序域

下面讨论强数据类型化对这些.NET 特性非常重要的原因。

1. 语言互操作性中强数据类型的重要性

如果类派生自其他类，或包含其他类的实例，它就需要知道其他类使用的所有数据类型，这就是强数据类型非常重要的原因。实际上，过去没有任何系统指定这些信息，从而成为语言继承和交互操作的真正障碍。这类信息不只是在一个标准的可执行文件或 DLL 中出现。

假定将 Visual Basic 2005 类中的一个方法定义为返回一个整型——Visual Basic 2005 可以使用的标准数据类型之一。但 C#没有该名称的数据类型。显然，我们只能从该类中派生，再使用这个方法，如果编译器知道如何把 Visual Basic 2005 的整型类型映射为 C#定义的某种已知类型，就可以在 C#代码中使用返回的类型。这个问题在.NET 中是如何解决的？

(1) 通用类型系统(CTS)

这个数据类型问题在.NET 中使用通用类型系统(CTS)得到了解决。CTS 定义了可以在中间语言中使用的预定义数据类型，所有面向.NET Framework 的语言都可以生成最终基于这些类型的编译代码。

例如，Visual Basic 2005 的整型实际上是一个 32 位有符号的整数，它实际映射为中间语言类型 Int32。因此在中间语言代码中就指定这种数据类型。C#编译器可以使用这种类型，所以就不会有问题了。在源代码中，C#用关键字 int 来表示 Int32，所以编译器就认为 Visual Basic 2005 方法返回一个 int 类型的值。

通用类型系统不仅指定了基本数据类型，还定义了一个内容丰富的类型层次结构，其中包含设计合理的位置，在这些位置上，代码允许定义它自己的类型。通用类型系统的层次结构反映了中间语言的单一继承的面向对象方法，如图 1-1 所示。

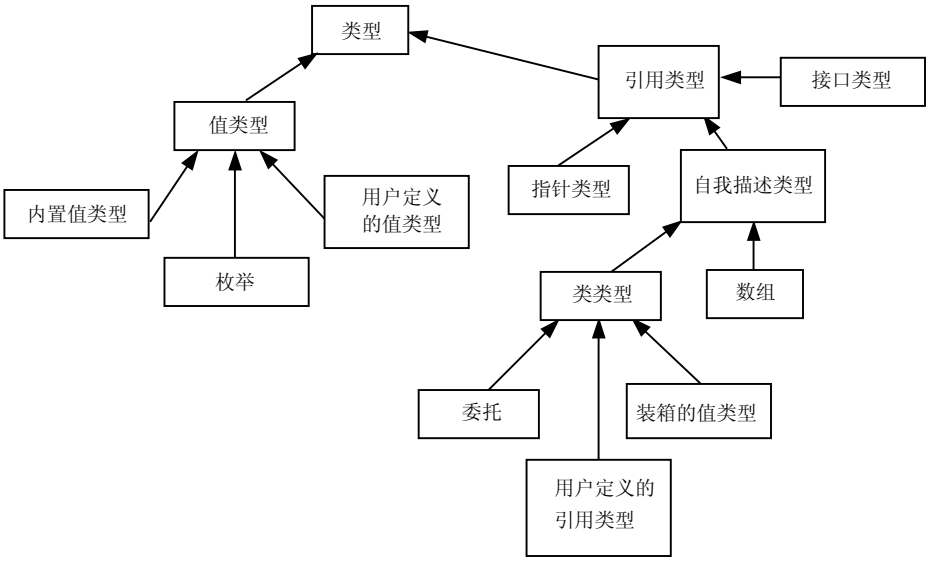

图　1-1

这个树形结构中的类型说明如表 1-1 所示。

表　1-1

类　　型	含　　义
Type	代表任何类型的基类
Value Type	代表任何值类型的基类
Reference Types	通过引用来访问，且存储在堆中的任何数据类型
Built-in Value Types	包含大多数标准基本类型，可以表示数字、Boolean 值或字符
Enumerations	枚举值的集合
User-defined Value Types	在源代码中定义，且保存为值类型的数据类型。在 C#中，它表示结构
Interface Types	接口
Pointer Types	指针
Self-describing Types	为垃圾回收器提供对它们本身有益的信息的数据类型(参见下一节)
Arrays	包含对象数组的类型
Class Types	可自我描述的类型，但不是数组
Delegates	用于把引用包含在方法中的类型
User-definedReference Types	在源代码中定义，且保存为引用类型的数据类型。在 C#中，它表示类
Boxed Value Types	值类型，临时打包放在一个引用中，以便于存储在堆中

这里没有列出内置的所有值类型，因为第 3 章将详细介绍它们。在 C#中，编译器识别

的每个预定义类型都映射为一个 IL 内置类型。这与 Visual Basic 2005 是一样的。

(2) 公共语言规范(CLS)

公共语言规范(Common Language Specification，CLS)和通用类型系统一起确保语言的互操作性。CLS 是一个最低标准集，所有面向.NET 的编译器都必须支持它。因为 IL 是一种内涵非常丰富的语言，大多数编译器的编写人员有可能把给定编译器的功能限制为只支持 IL 和 CLS 提供的一部分特性。只要编译器支持已在 CLS 中定义的内容，这就是很不错的。

提示：

编写非 CLS 兼容代码应该是完全可以接受的，只是在编写了这种代码后，就不能保证编译好的 IL 代码完全支持语言的互操作性。

下面的一个例子是有关区分大小写字母的。IL 是区分大小写的语言。使用这些语言的开发人员常常利用区分大小写所提供的灵活性来选择变量名。但 Visual Basic 2005 是不区分大小写的语言。CLS 就要指定 CLS 兼容代码不使用任何只根据大小写来区分的名称。因此，Visual Basic 2005 代码可以与 CLS 兼容代码一起使用。

这个例子说明了 CLS 的两种工作方式。首先是各个编译器的功能不必强大到支持.NET 的所有功能，这将鼓励人们为其他面向.NET 的编程语言开发编译器。第二，它提供如下保证：如果限制类只能使用 CLS 兼容的特性，就要保证用其他语言编写的代码可以使用这个类。

这种方法的优点是使用 CLS 兼容特性的限制只适用于公共和受保护的类成员和公共类。在类的私有实现方式中，可以编写非 CLS 代码，因为其他程序集(托管代码的单元，参见本章后面的内容)中的代码不能访问这部分代码。

这里不深入讨论 CLS 规范。在一般情况下，CLS 对 C#代码的影响不会太大，因为 C#中的非 CLS 兼容特性非常少。

2. 垃圾收集

垃圾收集器用来在.NET 中进行内存管理，特别是它可以恢复正在运行中的应用程序需要的内存。到目前为止，Windows 平台已经使用了两种技术来释放进程向系统动态请求的内存：

- 完全以手工方式使应用程序代码完成这些工作。
- 让对象维护引用计数。

让应用程序代码负责释放内存是低级高性能的语言使用的技术，例如 C++。这种技术很有效，且可以让资源在不需要时就释放(一般情况下)，但其最大的缺点是频繁出现错误。请求内存的代码还必须显式通知系统它什么时候不再需要该内存。但这是很容易被遗漏的，从而导致内存泄漏。

尽管现代的开发环境提供了帮助检测内存泄漏的工具，但它们很难跟踪错误，因为直到内存已大量泄漏从而使 Windows 拒绝为进程提供资源时，它们才会发挥作用。到那个时候，由于对内存的需求很大，会使整个计算机变得相当慢。

维护引用计数是 COM 对象采用的一种技术，其方法是每个 COM 组件都保留一个计

数，记录客户机目前对它的引用数。当这个计数下降到 0 时，组件就会删除自己，并释放相应的内存和资源。它带来的问题是仍需要客户机通知组件它们已经完成了内存的使用。只要有一个客户机没有这么做，对象就仍驻留在内存中。在某些方面，这是比 C++内存泄漏更为严重的问题，因为 COM 对象可能存在于它自己的进程中，从来不会被系统删除(在 C++内存泄漏问题上，系统至少可以在进程中断时释放所有的内存)。

.NET 运行库采用的方法是垃圾收集器，这是一个程序，其目的是清理内存，方法是所有动态请求的内存都分配到堆上(所有的语言都是这样处理的，但在.NET 中，CLR 维护它自己的托管堆，以供.NET 应用程序使用)，当.NET 检测到给定进程的托管堆已满，需要清理时，就调用垃圾收集器。垃圾收集器处理目前代码中的所有变量，检查对存储在托管堆上的对象的引用，确定哪些对象可以从代码中访问—— 即哪些对象有引用。没有引用的对象就不能再从代码中访问，因而被删除。Java 就使用与此类似的垃圾收集系统。

之所以在.NET 中使用垃圾收集器，是因为中间语言已用来处理进程。其规则要求，第一，不能引用已有的对象，除非复制已有的引用。第二，中间语言是类型安全的语言。在这里，其含义是如果存在对对象的任何引用，该引用中就有足够的信息来确定对象的类型。

垃圾收集器机制不能和诸如非托管 C++这样的语言一起使用，因为 C++允许指针自由地转换数据类型。

垃圾收集器的一个重要方面是它的不确定性。换言之，不能保证什么时候会调用垃圾收集器：.NET 运行库决定需要它时，就可以调用它(除非明确调用垃圾收集器)。但可以重写这个过程，在代码中调用垃圾收集器。

3. 安全性

.NET 很好地补足了 Windows 提供的安全机制，因为它提供的安全机制是基于代码的安全性，而 Windows 仅提供了基于角色的安全性。

基于角色的安全性建立在运行进程的账户的身份基础上，换言之，就是谁拥有和运行进程。另一方面，基于代码的安全性建立在代码实际执行的任务和代码的可信程度上。由于中间语言提供了强大的类型安全性，所以 CLR 可以在运行代码前检查它，以确定是否有需要的安全权限。.NET 还提供了一种机制，可以在运行代码前指定代码需要什么安全权限。

基于代码的安全性非常重要，原因是它降低了运行来历不明的代码的风险(例如代码是从 Internet 上下载来的)。即使代码运行在管理员账户下，也有可能使用基于代码的安全性，来确定这段代码是否仍不能执行管理员账户一般允许执行的某些类型的操作，例如读写环境变量、读写注册表或访问.NET 反射特性。

安全问题详见本书后面的第 19 章。

4. 应用程序域

应用程序域是.NET 中的一个重要技术改进，它用于减少运行应用程序的系统开销，这些应用程序需要与其他程序分离开来，但仍需要彼此通信。典型的例子是 Web 服务器应用程序，它需要同时响应许多浏览器请求。因此，要有许多组件实例同时响应这些同时运行的请求。

在.NET 开发出来以前，可以让这些实例共享同一个进程，但此时一个运行的实例就有

可能导致整个网站的崩溃；也可以把这些实例孤立在不同的进程中，但这样做会增加相关性能的系统开销。

到现在为止，孤立代码的唯一方式是通过进程来实现的。在运行一个新的应用程序时，它会在一个进程环境内运行。Windows 通过地址空间把进程分隔开来。这样，每个进程有 4GB 的虚拟内存来存储其数据和可执行代码(4GB 对应于 32 位系统，64 位系统要用更多的内存)。Windows 利用额外的间接方式把这些虚拟内存映射到物理内存或磁盘空间的一个特殊区域中，每个进程都会有不同的映射，虚拟地址空间块映射的物理内存之间不能有重叠，这种情况如图 1-2 所示。

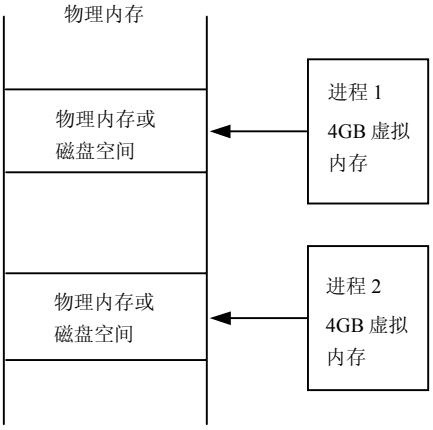

图 1-2

在一般情况下，任何进程都只能通过指定虚拟内存中的一个地址来访问内存——即进程不能直接访问物理内存，因此一个进程不可能访问分配给另一个进程的内存。这样就可以确保任何执行出错的代码不会损害其地址空间以外的数据(注意在 Windows 95/98 上，这些保护措施不像在 Windows NT/2000/XP/2003 上那样强大，所以理论上存在应用程序因写入不对应的内存而导致 Windows 崩溃的可能性)。

进程不仅是运行代码的实例相互隔离的一种方式，在 Windows NT/2000/XP/2003 系统上，它们还可以构成分配了安全权限和许可的单元。每个进程都有自己的安全标识，明确地表示 Windows 允许该进程可以执行的操作。

进程对确保安全有很大的帮助，而它们的一大缺点是性能。许多进程常常在一起工作，因此需要相互通信。一个常见的例子是进程调用一个 COM 组件，而该 COM 组件是可执行的，因此需要在它自己的进程上运行。在 COM 中使用代理时也会发生类似的情况。因为进程不能共享任何内存，所以必须使用一个复杂的编组过程在进程之间复制数据。这对性能有非常大的影响。如果需要使组件一起工作，但不希望性能有损失，唯一的方法是使用基于 DLL 的组件，让所有的组件在同一个地址空间中运行—— 其风险是执行出错的组件会影响其他组件。

应用程序域是分离组件的一种方式，它不会导致因在进程之间传送数据而产生的性能问题。其方法是把任何一个进程分解到多个应用程序域中，每个应用程序域大致对应一个应用程序，执行的每个线程都运行在一个具体的应用程序域中，如图 1-3 所示。

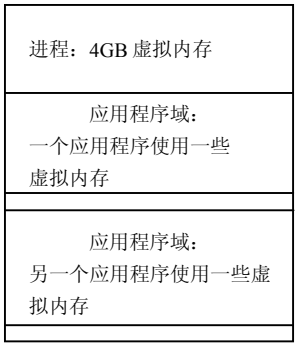

图 1-3

如果不同的可执行文件都运行在同一个进程空间中，显然它们就能轻松地共享数据，因为理论上它们可以直接访问彼此的数据。虽然在理论上这是可以实现的，但是 CLR 会检查每个正在运行的应用程序的代码，以确保这些代码不偏离它自己的数据区域，保证不发生直接访问其他进程的数据的情况。这初看起来是不可能的，如何告诉程序要做什么工作，而又不真正运行它？

实际上，这么做通常是可能的，因为中间语言拥有强大的类型安全功能。在大多数情况下，除非代码明确使用不安全的特性，例如指针，否则它使用的数据类型可以确保内存不会被错误地访问。例如，.NET 数组类型执行边界检查，以禁止执行超出边界的数组操作。如果运行的应用程序的确需要与运行在不同应用程序域中的其他应用程序通信或共享数据，就必须调用.NET 的远程服务。

被验证不能访问超出其应用程序域的数据(而不是通过明确的远程机制)的代码就是内存类型安全的代码，这种代码与运行在同一个进程中但应用程序域不同的类型安全代码一起运行是安全的。

1.3.4 通过异常处理错误

.NET Framework 可以根据异常使用相同的机制处理错误情况，这与 Java 和 C++是一样的。C++开发人员应注意到，由于 IL 有非常强大的类型系统，所以在 IL 中以 C++的方式使用异常不会带来相关的性能问题。另外，.NET 和 C#也支持 finally 块，这是许多 C++开发人员长久以来的愿望。

第 13 章会详细讨论异常。简要地说，代码的某些领域被看作是异常处理程序例程，每个例程都能处理某种特殊的错误情况(例如，找不到文件，或拒绝执行某些操作的许可)。这些条件可以定义得很宽或很窄。异常结构确保在发生错误情况时，执行进程立即跳到最合适的异常处理程序例程上，处理错误情况。

异常处理的结构还提供了一种方便的方式，当对象包含错误情况的准确信息时，该对象就可以传送给错误处理例程。这个对象包括给用户提供的相应信息和在代码的什么地方检测到错误的确切信息。

大多数异常处理结构，包括异常发生时的程序流控制，都是由高级语言处理的，例如 C#、Visual Basic 2005 和 C++，任何中间语言中的命令都不支持它。例如，C#使用 try{}、

catch{}和 finally{}代码块来处理它，详见第 13 章。

.NET 提供了一种基础结构，让面向.NET 的编译器支持异常处理。特别是它提供了一组.NET 类来表示异常，语言的互操作性则允许错误处理代码处理被抛出的异常对象，无论错误处理代码使用什么语言编写，都是这样。语言的无关性没有体现在 C++和 Java 的异常处理中，但在 COM 的错误处理机制中有一定限度的体现。COM 的错误处理机制包括从方法中返回错误代码以及传递错误对象。在不同的语言中，异常的处理是一致的，这是多语言开发的重要一环。

1.3.5 特性的使用

特性(attribute)是使用 C++编写 COM 组件的开发人员很熟悉的一个功能(使用 Microsoft 的 COM 接口定义语言(Interface Definition Language，IDL))。特性最初是为了在程序中提供与某些项相关的额外信息，以供编译器使用。

.NET 支持特性，因此现在 C++、C#和 Visual Basic 2005 也支持特性。但在.NET 中，对特性的革新是建立了一个机制，通过该机制可以在源代码中定义自己的特性。这些用户定义的特性将和对应数据类型或方法的元数据放在一起，这对于文档说明书十分有用，它们和反射技术一起使用，以根据特性执行编程任务。另外，与.NET 的语言无关性的基本原理一样，特性也可以在一种语言的源代码中定义，而被用另一种语言编写的代码读取。

本书的第 12 章详细介绍了特性。

1.4 程序集

程序集(assembly)是包含编译好的、面向.NET Framework 的代码的逻辑单元。本章不详细论述程序集，而在第 16 章中论述，下面概述其中的要点。

程序集是完全自我描述性的，也是一个逻辑单元而不是物理单元，它可以存储在多个文件中(动态程序集的确存储在内存中，而不是存储在文件中)。如果一个程序集存储在多个文件中，其中就会有一个包含入口点的主文件，该文件描述了程序集中的其他文件。

注意可执行代码和库代码使用相同的程序集结构。唯一的区别是可执行的程序集包含一个主程序入口点，而库程序集不包含。

程序集的一个重要特性是它们包含的元数据描述了对应代码中定义的类型和方法。程序集也包含描述程序集本身的元数据，这种程序集元数据包含在一个称为“程序集清单”的区域中，可以检查程序集的版本及其完整性。

注意：

ildasm 是一个基于 Windows 的实用程序，可以用于检查程序集的内容，包括程序集清单和元数据。第 16 章将介绍 ildasm。

程序集包含程序的元数据，表示调用给定程序集中的代码的应用程序或其他程序集不需要指定注册表或其他数据源，以确定如何使用该程序集。这与以前的 COM 有很大的区别，以前，组件的 GUID 和接口必须从注册表中获取，在某些情况下，方法和属性的详细

信息也需要从类型库中读取。

把数据分散在 3 个以上的不同位置上，可能会出现信息不同步的情况，从而妨碍其他软件成功地使用该组件。有了程序集后，就不会发生这种情况，因为所有的元数据都与程序的可执行指令存储在一起。注意，即使程序集存储在几个文件中，数据也不会出现不同步的问题。这是因为包含程序集入口的文件也存储了其他文件的细节、散列和内容，如果一个文件被替换，或者被塞满，系统肯定会检测出来，并拒绝加载程序集。

程序集有两种类型：共享程序集和私有程序集。

1.4.1 私有程序集

私有程序集是最简单的一种程序集类型。私有程序集一般附带在某个软件上，且只能用于该软件。附带私有程序集的常见情况是，以可执行文件或许多库的方式提供应用程序，这些库包含的代码只能用于该应用程序。

系统可以保证私有程序集不被其他软件使用，因为应用程序只能加载位于主执行文件所在文件夹或其子文件夹中的程序集。

用户一般会希望把商用软件安装在它自己的目录下，这样软件包没有覆盖、修改或加载另一个软件包的私有程序集的风险。私有程序集只能用于自己的软件包，这样，用户对什么软件使用它们就有了更多的控制。因此，不需要采取安全措施，因为这没有其他商用软件用某个新版本的程序集覆盖原来的私有程序集的风险(但软件是专门执行怀有恶意的损害性操作的情况除外)。名称也不会有冲突。如果私有程序集中的类正巧与另一个人的私有程序集中的类同名，是不会有问题的，因为给定的应用程序只能使用私有程序集的名称。

因为私有程序集完全是自含式的，所以安装它的过程就很简单。只需把相应的文件放在文件系统的对应文件夹中即可(不需要注册表项)，这个过程称为“0 影响(xcopy)安装”。

1.4.2 共享程序集

共享程序集是其他应用程序可以使用的公共库。因为其他软件可以访问共享程序集，所以需要采取一定的保护措施来防止以下风险：

- 名称冲突，另一个公司的共享程序集执行的类型与自己的共享程序集中的类型同名。因为客户机代码理论上可以同时访问这些程序集，所以这是一个严重的问题。
- 程序集被同一个程序集的不同版本覆盖——新版本与某些已有的客户机代码不兼容。

这些问题的解决方法是把共享程序集放在文件系统的一个特定的子目录树中，称为全局程序集高速缓存(GAC)。与私有程序集不同，不能简单地把共享程序集复制到对应的文件夹中，而需要专门安装到高速缓存中，这个过程可以用许多.NET 工具来完成，其中包含对程序集的检查、在程序集高速缓存中设置一个小的文件夹层次结构，以确保程序集的完整性。

为了避免名称冲突，共享程序集应根据私有密钥加密法指定一个名称(私有程序集只需要指定与其主文件名相同的名称即可)。该名称称为强名(strong name)，并保证其唯一性，

它必须由要引用共享程序集的应用程序来引用。

与覆盖程序集相关的问题，可以通过在程序集清单中指定版本信息来解决，也可以通过同时安装来解决。

1.4.3 反射

因为程序集存储了元数据，包括在程序集中定义的所有类型和这些类型的成员的细节，所以可以编程访问这些元数据。这个技术称为反射，第 12 章详细介绍了它们。该技术很有趣，因为它表示托管代码实际上可以检查其他托管代码，甚至检查它自己，以确定该代码的信息。它们常常用于获取特性的详细信息，也可以把反射用于其他目的，例如作为实例化类或调用方法的一种间接方式，如果把方法上的类名指定为字符串，就可以选择类来实例化方法，以便在运行时调用，而不是在编译时调用，例如根据用户的输入来调用(动态绑定)。

1.5 .NET Framework 类

至少从开发人员的角度来看，编写托管代码的最大好处是可以使用.NET 基类库。

.NET 基类是一个内容丰富的托管代码类集合，它可以完成以前要通过 Windows API 来完成的绝大多数任务。这些类派生自与中间语言相同的对象模型，也基于单一继承性。无论.NET 基类是否合适，都可以实例化对象，也可以从它们派生自己的类。

.NET 基类的一个优点是它们非常直观和易用。例如，要启动一个线程，可以调用 Thread 类的 Start()方法。要禁用 TextBox，应把 TextBox 对象的 Enabled 属性设置为 false。Visual Basic 和 Java 开发人员非常熟悉这种方式。它们的库都很容易使用，但对于 C++开发人员来说这是极大的解脱，因为他们多年来一直在使用诸如 GetDIBits()、RegisterWndClassEx()和 IsEqualIID()这样的 API 函数，以及需要传递 Windows 句柄的函数。

另一方面，C++开发人员总是很容易访问整个 Windows API，而 Visual Basic 6 和 Java 开发人员只能访问其语言所能访问的基本操作系统功能。.NET 基类的新增内容就是把 Visual Basic 和 Java 库的易用性和 Windows API 函数的丰富功能结合起来。但 Windows 仍有许多功能不能通过基类来使用，而需要调用 API 函数。但一般情况下，这只限于比较复杂的特性。基类库足以应付日常工作的使用。如果需要调用 API 函数，.NET 提供了所谓的“平台调用”，来确保对数据类型进行正确的转换，这样无论是使用 C#、C++或 Visual Basic 2005 进行编码，该任务都不会比直接从已有的 C++代码中调用函数更困难。

注意：

WinCV 是一个基于 Windows 的实用程序，可以用于浏览基类库中的类、结构、接口和枚举。本书将在第 14 章介绍 WinCV。

第 3 章主要介绍基类。完成了 C#语言语法的概述后，本书的其余内容将主要说明如何使用.NET 2.0 和.NET Framework 3.0 的.NET 基类库中的各种类，即各种基类是如何工作的。.NET 2.0 和 3.0 基类包括：

- IL 提供的核心功能，例如，通用类型系统中的基本数据类型，详见第 3 章。
- Windows GUI 支持和控件(第 28 和 31 章)
- Web 窗体(ASP.NET，第 32 和 33 章)
- 数据访问(ADO.NET，第 25~27 章)
- 目录访问(第 42 章)
- 文件系统和注册表访问(第 24 章)
- 网络和 Web 浏览(第 35 章)
- .NET 特性和反射(第 12 章)
- 访问 Windows 操作系统的各个方面(例如环境变量等，第 19 章)
- COM 互操作性(第 38 和 23 章)

附带说一下，根据 Microsoft 源文件，大部分.NET 基类实际上都是用 C#编写的！

1.6 命名空间

命名空间是.NET 避免类名冲突的一种方式。例如，命名空间可以避免下述情况：定义一个类来表示一个顾客，称此类为 Customer，同时其他人也在做相同的事(这有一个类似的场景——顾客有相当多的业务)。

命名空间不过是数据类型的一种组合方式，但命名空间中所有数据类型的名称都会自动加上该命名空间的名字作为其前缀。命名空间还可以相互嵌套。例如，大多数用于一般目的的.NET 基类位于命名空间 System 中，基类 Array 在这个命名空间中，所以其全名是 System.Array。

.NET 需要在命名空间中定义所有的类型，例如，可以把 Customer 类放在命名空间 YourCompanyName 中，则这个类的全名就是 YourCompanyName.Customer。

注意：

如果没有显式提供命名空间，类型就添加到一个没有名称的全局命名空间中。

Microsoft 建议在大多数情况下，都至少要提供两个嵌套的命名空间名，第一个是公司名，第二个是技术名称或软件包的名称，而类是其中的一个成员，例如 YourCompanyName.Sales Services.Customer。在大多数情况下，这么做可以保证类的名称不会与其他组织编写的类名冲突。

第 2 章将详细介绍命名空间。

1.7 用 C#创建.NET 应用程序

C#可以用于创建控制台应用程序：仅使用文本、运行在 DOS 窗口中的应用程序。在

进行单元测试类库、创建 Unix/Linux daemon 进程时，就要使用控制台应用程序。但是，我们常常使用 C#创建利用许多与.NET 相关的技术的应用程序，下面简要论述可以用 C#创建的不同类型的应用程序。

1.7.1 创建 ASP.NET 应用程序

ASP 是用于创建带有动态内容的 Web 页面的一种 Microsoft 技术。ASP 页面基本是一个嵌有服务器端 VBScript 或 JavaScript 代码块的 HTML 文件。当客户浏览器请求一个 ASP 页面时，Web 服务器就会发送页面的 HTML 部分，并处理服务器端脚本。这些脚本通常会查询数据库的数据，在 HTML 中标记数据。ASP 是客户建立基于浏览器的应用程序的一种便利方式。

但 ASP 也有缺点。首先，ASP 页面有时显示得比较慢，因为服务器端代码是解释性的，而不是编译的。第二，ASP 文件很难维护，因为它不是结构化的，服务器端的 ASP 代码和一般的 HTML 会混合在一起。第三，ASP 有时开发起来会比较困难，因为它不支持错误处理和语法检查。特别是如果使用 VBScript，并希望在页面中进行错误处理，就必须使用 On Error Resume Next 语句，通过 Err.Number 检查每个组件调用，以确保该调用正常进行。

ASP.NET 是 ASP 的全新修订版本，它解决了 ASP 的许多问题。但 ASP.NET 页面并没有替代 ASP，而是可以与原来的 ASP 应用程序在同一个服务器上并存。当然，也可以用 C#编写 ASP.NET。

后面的章节(第 32 和第 33 章)会详细讨论 ASP.NET，这里仅解释它的一些重要特性。

1. ASP.NET 的特性

首先，也是最重要的是，ASP.NET 页面是结构化的。这就是说，每个页面都是一个继承了.NET 类 System.Web.UI.Page 的类，可以重写在 Page 对象的生存期中调用的一系列方法，（可以把这些事件看成是页面所特有的，对应于原 ASP 的 global.asa 文件中的 OnApplication_Start 和 OnSession_Start 事件）。因为可以把一个页面的功能放在有明确含义的事件处理程序中，所以 ASP.NET 比较容易理解。

ASP.NET 页面的另一个优点是可以在 Visual Studio 2005 中创建它们，在该环境下，可以创建 ASP.NET 页面使用的业务逻辑和数据访问组件。Visual Studio 2005 项目(也称为解决方案)包含了与应用程序相关的所有文件。而且，也可以在编辑器中调试传统的 ASP 页面，在以前使用 Visual InterDev 时，把 InterDev 和项目的 Web 服务器配置为支持调试常常是一个让人头痛的问题。

最清楚的是，ASP.NET 的后台编码功能允许进一步采用结构化的方式。ASP.NET 允许把页面的服务器端功能单独放在一个类中，把该类编译为 DLL，并把该 DLL 放在 HTML 部分下面的一个目录中。放在页面顶部的后台编码指令将把该文件与其 DLL 关联起来。当浏览器请求该页面时，Web 服务器就会在页面的后台 DLL 中引发类中的事件。

最后，ASP.NET 在性能的提高上非常明显。传统的 ASP 页面是和每个页面请求一起解释，而 Web 服务器是在编译后高速缓存 ASP.NET 页面。这表示以后对 ASP.NET 页面的

请求就比 ASP 页面第一次执行的速度快得多。

ASP.NET 还易于编写通过浏览器显示窗体的页面，这在内联网环境中会使用。传统的方式是基于窗体的应用程序提供一个功能丰富的用户界面，但较难维护，因为它们运行在非常多的不同机器上。因此，当用户界面是必不可少的，并可以为用户提供扩展支持时，人们就会依赖基于窗体的应用程序。

2. Web 窗体

为了简化 Web 页面的结构，Visual Studio 2005 提供了 Web 窗体。它们允许以创建 Visual Basic 6 或 C++ Builder 窗口的方式图形化地建立 ASP.NET 页面；换言之，就是把控件从工具箱拖放到窗体上，再考虑窗体的代码，为控件编写事件处理程序。在使用 C#创建 Web 窗体时，就是创建一个继承于 Page 基类的 C#类，以及把这个类看作是后台编码的 ASP.NET 页面。当然不必使用 C#创建 Web 窗体，而可以使用 Visual Basic 2005 或另一种.NET 语言来创建。

过去，Web 开发的困难使一些开发小组不愿意使用 Web。为了成功地进行 Web 开发，必须了解非常多的不同技术，例如 VBScript、ASP、DHTML、JavaScript 等。把窗体概念应用于 Web 页面，Web 窗体就可以使 Web 开发容易许多。

Web 服务器控件

用于添加到 Web 窗体上的控件与 ActiveX 控件并不是同一种控件，它们是 ASP.NET 命名空间中的 XML 标记，当请求一个页面时，Web 浏览器会动态地把它们转换为 HTML 和客户端脚本。Web 服务器能以不同的方式显示相同的服务器端控件，产生一个对应于请求者特定 Web 浏览器的转换。这意味着现在很容易为 Web 页面编写相当复杂的用户界面，而不必担心如何确保页面运行在可用的任何浏览器上，因为 Web 窗体会完成这些任务。

可以使用 C#或 Visual Basic 2005 扩展 Web 窗体工具箱。创建一个新服务器端控件，仅是执行.NET 的 System.Web.UI.WebControls.WebControl 类而已。

3. XML Web 服务

目前，HTML 页面解决了 World Wide Web 上的大部分通信问题。有了 XML，计算机就可以用一种独立于设备的格式，在 Web 上彼此通信。将来，计算机可以使用 Web 和 XML 交流信息，而不是专用的线路和专用的格式，例如 EDI (Electronic Data Interchange)。XML Web 服务是为面向 Web 的服务而设计的，即远程计算机彼此提供可以分析和重新格式化的动态信息，最后显示给用户。XML Web 服务是计算机给 Web 上的其他计算机以 XML 格式显示信息的一种便利方式。

在技术上，.NET 上的 XML Web 服务是给请求的客户返回 XML 而不是 HTML 的 ASP.NET 页面。这种页面有后台编码的 DLL，它包含了派生自 WebService 类的类。Visual Studio 2005 IDE 提供的引擎简化了 Web 服务的开发。

公司选择使用 XML Web 服务主要有两个原因。第一是因为它们依赖于 HTTP，而 XML Web 服务可以把现有的网络(HTTP)用作传输信息的媒介。第二是因为 XML Web 服务使用 XML，该数据格式是自我描述的、非专用的、独立于平台的。

1.7.2 创建 Windows 窗体

C#和.NET 非常适合于 Web 开发，它们还为所谓的“胖客户端”应用程序提供了极好的支持，这种“胖客户端”应用程序必须安装在最终用户的机器上，来处理大多数操作，这种支持来源于 Windows 窗体。

Windows 窗体是 Visual Basic 6 窗体的.NET 版本，要设计一个图形化的窗口界面，只需把控件从工具箱拖放到 Windows 窗体上即可。要确定窗口的行为，应为该窗体的控件编写事件处理例程。Windows Form 项目编译为.EXE，该 EXE 必须与.NET 运行库一起安装在最终用户的计算机上。与其他.NET 项目类型一样，Visual Basic 2005 和 C#都支持 Windows Form 项目。第 28 章将详细介绍 Windows 窗体。

1.7.3 使用 Windows Presentation Foundation(WPF)

一种最新的技术叫做 Windows Presentation Foundation(WPF)。WPF 在建立应用程序时使用 XAML。XAML 表示可扩展的应用程序标记语言(Extensible Application Markup Language)。这种在 Microsoft 环境下创建应用程序的新方式在 2006 年引入，是.NET Framework 3.0 的一部分。要运行 WPF 应用程序，需要在客户机上安装.NET Framework 3.0。WPF 应用程序可用于 Windows Vista、Windows XP 和 Windows Server 2003(只有这些操作系统能安装.NET Framework 3.0)。

XAML 是用于创建窗体的 XML 声明，它代表 WPF 应用程序的所有可视化部分和操作。虽然可以编程利用 WPF 应用程序，但 WPF 是迈向声明性编程的一步，而声明性编程是编程业的趋势。声明性编程是指，不是利用编译语言，如 C#、VB 或 Java，通过编程来创建对象，而是通过 XML 类型的编程来声明所有的元素。第 31 章详细介绍了如何使用 XAML 和 C#建立这些新类型的应用程序。

1.7.4 Windows 控件

Web 窗体和 Windows 窗体的开发方式一样，但应为它们添加不同类型的控件。Web 窗体使用 Web 服务器控件，Windows 窗体使用 Windows 控件。

Windows 控件比较类似于 ActiveX 控件。在执行 Windows 控件后，它会编译为必须安装到客户机器上的 DLL。实际上，.NET SDK 提供了一个实用程序，为 ActiveX 控件创建包装器，以便把它们放在 Windows 窗体上。与 Web 控件一样，Windows 控件的创建需要派生于特定的类 System.Windows.Forms.Control。

1.7.5 Windows 服务

Windows 服务(最初称为 NT 服务)是一个在 Windows NT/2000/XP/2003 (但没有 Windows 9x)后台运行的程序。当希望程序连续运行，响应事件，但没有用户的明确启动操作时，就应使用 Windows 服务。例如 Web 服务器上的 World Wide Web 服务，它们监听来自客户的 Web 请求。

用 C#编写服务是非常简单的。System.ServiceProcess 命名空间中的.NET Framework 基类可以处理许多与服务相关的样本任务，另外，Visual Studio 2005 允许创建 C# Windows Service 项目，为基本 Windows 服务编写 C#源代码。第 22 章将详细介绍如何编写 C# Windows 服务。

1.7.6 Windows Communication Foundation(WCF)

通过基于 Microsoft 的技术，可以采用许多方式将数据和服务从一处移动到另一处。例如，可以使用 ASP.NET Web 服务、.NET Remoting、Enterprise Services 和用于初学者的 MSMQ。应采用哪种技术？这要考虑具体要达到的目标，因为每种技术都适合于不同的场合。

因此，Microsoft 把所有这些技术集成在一起，放在.NET Framework 3.0 中。现在只有一种移动数据的方式——Windows Communication Foundation(WCF)。WCF 允许建立好服务后，只要修改配置文件，就可以用多种方式提供该服务(甚至在不同的协议下)。WCF 是一种连接各种系统的强大的新方式。第 40 章将详细介绍 WCF。

1.8 C#在.NET 企业体系结构中的作用

C#需要.NET 运行库，在几年内大多数客户机—— 特别是大多数家用 PC—— 就可以安装.NET 了。而且，安装 C#应用程序在方式上类似于安装.NET 可重新分布的组件。因此，企业环境中会有许多 C#应用程序。实际上，C#为希望建立健全的 n 层客户机/服务器应用程序的公司提供了一个绝佳的机会。

C#与 ADO.NET 合并后，就可以快速而经常地访问数据库了，例如 SQL Server 和 Oracle 数据库。返回的数据集很容易通过 ADO.NET 对象模型来处理，并自动显示为 XML，一般通过办公室内联网来传输。

一旦为新项目建立了数据库模式，C#就会为执行一层数据访问对象提供一个极好的媒介，每个对象都能提供对不同的数据库表的插入、更新和删除访问。

因为这是第一个基于组件的 C 语言，所以 C#非常适合于执行业务对象层。它为组件之间的通信封装了杂乱的信息，让开发人员把注意力集中如何在把数据访问对象组合在一起，在方法中精确地强制执行公司的业务规则。而且使用特性，C#业务对象可以配备方法级的安全检查、对象池和由 COM+服务提供的 JIT 活动。另外，.NET 附带的实用程序允许新的.NET 业务对象与原来的 COM 组件交互。

要使用 C#创建企业应用程序，可以为数据访问对象创建一个 Class Library 项目，为业务对象创建另一个 Class Library 项目。在开发时，可以使用 Console 项目测试类上的方法。喜欢编程的人可以建立能自动从批处理文件中执行的 Console 项目，测试工作代码是否中断。

注意，C# 和 .NET 都会影响物理封装可重用类的方式。过去，许多开发人员把许多类放在一个物理组件中，因为这样安排会使部署容易得多；如果有版本冲突问题，就知道

在何处进行检查。因为部署.NET 企业组件仅是把文件复制到目录中，所以现在开发人员就可以把他们的类封装到逻辑性更高的离散组件中，而不会遇到 DLL Hell。

最后，用 C#编写的 ASP.NET 页面构成了用户界面的绝妙媒介。ASP.NET 页面是编译过的，所以执行得比较快。它们可以在 VS 2005 IDE 中调试，所以更加健壮。它们支持所有的语言特性，例如早期绑定、继承和模块化，所以用 C#编写的 ASP.NET 页面是很整洁的，很容易维护。

经验丰富的开发人员对大做广告的新技术和语言都持非常怀疑的态度，不愿意利用新平台，这仅仅是因为他们不愿意。如果读者是一位 IT 部门的企业开发人员，或者通过 World Wide Web 提供应用程序服务，即使一些比较奇异的特性如 XML Web 服务和服务器端控件不算在内，也可以确保 C#和.NET 至少提供了四个优点：

- 组件冲突将很少见，部署工作将更容易，因为同一组件的不同版本可以在同一台机器上并行运行，而不会发生冲突。
- ASP.NET 代码不再难懂。
- 可以利用.NET 基类中的许多功能。
- 对于需要 Windows 窗体用户界面的应用程序来说，利用 C#可以很容易编写这类应用程序。

在某种程度上，以前 Windows 窗体并未受到重视，因为没有 Web 窗体和基于 Internet 的应用程序。但如果用户缺乏 JavaScript、ASP 或相关技术的专业知识，Windows 窗体仍是方便而快速地创建用户界面的一种可行选择。记住管理好代码，使用户界面的逻辑与业务逻辑和数据访问代码分隔开来，这样才能在将来的某一刻把应用程序迁移到浏览器上。另外，Windows 窗体还为家用应用程序和一些小公司长期保留了重要的用户界面。Windows 窗体的新智能客户特性(很容易以在线和离线方式工作)将能开发出新的、更好的应用程序。

1.9 小结

本章介绍了许多基础知识，简要回顾了.NET Framework 的重要方面以及它与 C#的关系。首先讨论了所有面向.NET 的语言如何编译为中间语言，之后由公共语言运行库进行编译和执行。我们还讨论了.NET 的下述特性在编译和执行过程中的作用：

- 程序集和.NET 基类
- COM 组件
- JIT 编译
- 应用程序域
- 垃圾收集

图 1-4 简要说明了这些特性在编译和执行过程中是如何发挥作用的。

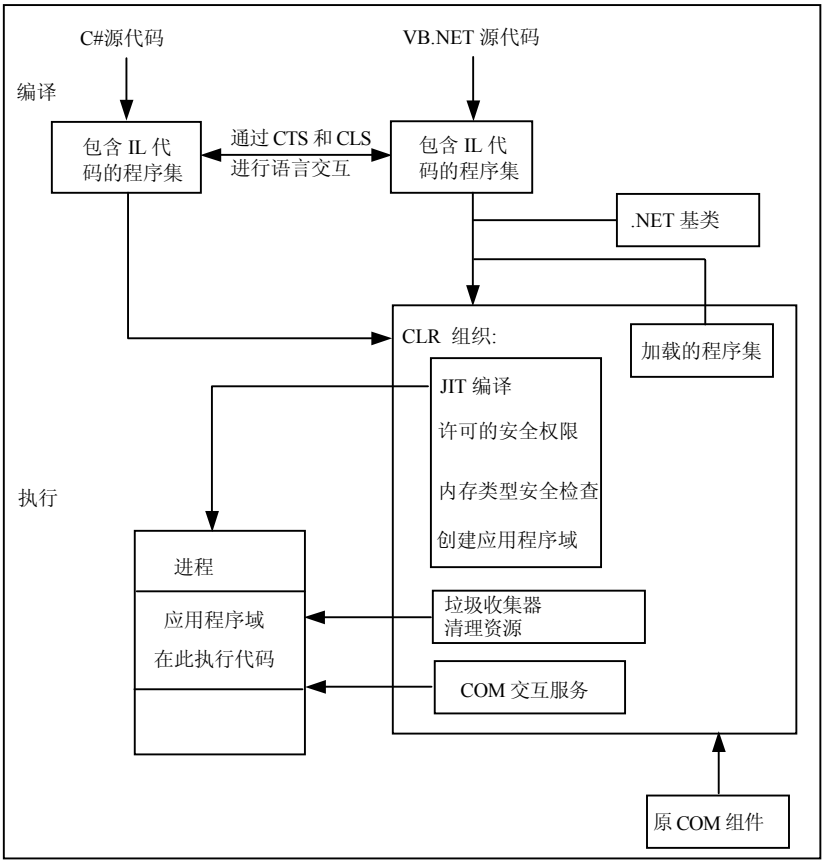

图　1-4

本章还讨论了 IL 的特性，特别是其强数据类型和面向对象的特性。探讨了这些特性如何影响面向.NET 的语言，包括 C#，并阐述了 IL 的强类型本质如何支持语言的互操作性，以及 CLR 服务，例如垃圾收集和安全性。

本章的最后讨论了 C#如何用作基于几个.NET 技术(包括 ASP.NET)的应用程序的基础。

第 2 章将介绍如何用 C#语言编写代码。

第 2 章

C# 基 础

理解了 C#的用途后，就可以学习如何使用它了。本章将介绍 C#的基础知识，并假定读者具备 C#编程的基本知识，这是后续章节的基础。本章的主要内容如下：

- 声明变量
- 变量的初始化和作用域
- C#的预定义数据类型
- 在 C#程序中使用循环和条件语句指定执行流
- 枚举
- 命名空间
- Main()方法
- 基本的命令行 C#编译器选项
- 使用 System.Console 执行控制台 I/O
- 在 C#和 Visual Studio .NET 中使用文档编制功能
- C#标识符和关键字
- C#编程的推荐规则和约定

阅读完本章后，读者就有足够的 C#知识编写简单的程序了，但还不能使用继承或其他面向对象的特征。这些内容将在本书后面的几章中讨论。

2.1 引言

如前所述，C#是一种面向对象的语言。在快速浏览 C#语句的基础时，我们假定读者已经很好地掌握了面向对象(OO)编程的概念。换言之，我们希望读者懂得类、对象、接口和继承的含义。如果读者以前使用过 C++或 Java，就应有很好的面向对象编程(OOP)的基础。但是，如果读者不具备 OOP 的背景知识，这个主题有许多很好的信息资源。本书的附录 A 就详细介绍了 OOP。附录 A 可以从 www.wrox.com 上下载。

如果读者对 Visual Basic6、C++或 Java 中的一种语言有丰富的编程经验，就应注意在

介绍 C#基础知识时，我们对 C#、C++、Java 和 Visual Basic6 进行了许多比较。但是，读者也许愿意阅读一本有关 C#和自己所选语言的比较的图书，来学习 C#。如果是这样，可以从 Wrox Press 网站(www.wrox.com)上下载不同的文档来学习 C#。

2.2 第一个 C#程序

下面采用传统的方式，编译并运行最简单的 C#程序，这是一个把信息写到屏幕上的控制台应用程序。

2.2.1 代码

在文本编辑器(例如 Notepad)中键入下面的代码，把它保存为.cs 文件(例如 First.cs)。Mian()方法如下所示：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class MyFirstCSharpClass
   {
      static void Main()
      {
         Console.WriteLine("This isn't at all like Java!");
         Console.ReadLine();
         return;
      }
   }
}
```

注意：

在后面的几章中，介绍了许多代码示例。编写 C#程序最常用的技巧是使用 Visual Studio 2005 生成一个基本项目，再添加自己的代码。但是，前面几章的目的是讲授 C#语言，为了简单起见，在第 14 章之前避免涉及 Visual Studio 2005。我们使代码显示为简单的文件，这样就可以使用任何文本编辑器键入它们，并在命令行上编译。

2.2.2 编译并运行程序

对源文件运行 C#命令行编译器(csc.exe)，编译这个程序：

```
csc First.cs
```

如果使用 csc 命令在命令行上编译代码，就应注意.NET 命令行工具，包括 csc，只有在设置了某些环境变量后才能使用。根据安装.NET(和 Visual Studio 2005)的方式，这里显示的结果可能与您机器上的结果不同。

注意：

如果没有设置环境变量，有两种解决方法。第一种方法是在运行 csc 之前，在命令行上运行批处理文件%Microsoft Visual Studio 2005%\Common7\Tools\vcvars32.bat。其中%Microsoft Visual Studio 2005 是安装 Visual Studio 2005 的文件夹。第二种方法(更简单)是使用 Visual Studio 2005 命令行代替通常的命令提示窗口。Visual Studio 2005 命令提示在“开始”菜单—“程序”—Microsoft Visual Studio 2005 - Microsoft Visual Studio Tools 子菜单下。它只是一个命令提示窗口，打开时会自动运行 vcvars32.bat。

编译代码，会生成一个可执行文件 First.exe。在命令行或 Windows Explorer 上，像运行任何可执行文件那样运行该文件，得到如下结果：

```
csc First.cs
    Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
    for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
    Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
First.exe
    This isn't at all like Java!
```

这些信息也许不那么真实！这个程序与 Java 有一些非常相似的地方，但有一两个地方与 Java 或 C++不同(如大写的 Main( )函数)。下面通过这个程序快速介绍 C#程序的基本结构。

2.2.3　详细介绍

首先对 C#语法作几个解释。在 C#中，与其他 C 风格的语言一样，大多数语句都以分号(;)结尾，语句可以写在多个代码行上，不需要使用续行字符(例如 Visual Basic 中的下划线)。用花括号({ ... })把语句组合为块。单行注释以两个斜杠字符开头(//)，多行注释以一个斜杠和一个星号(/*)开头，以一个星号和一个斜杠(*/)结尾。在这些方面，C#与 C++和 Java 一样，但与 Visual Basic 不同。分号和花括号使 C#代码与 Visual Basic 代码有完全不同的外观。如果您以前使用的是 Visual Basic，就应特别注意每个语句结尾的分号。对于使用 C 风格语言的新用户，忽略分号常常是导致编译错误的一个最主要的原因。

另一个方面是，C#是区分大小写的，也就是说，变量 myVar 与 MyVar 是两个不同的变量。

在上面的代码示例中，前几行代码是处理命名空间的(如本章后面所述)，命名空间是把相关类组合在一起的方式。Java 和 C++开发人员应很熟悉这个概念，但对于 Visual Basic6 开发人员来说是新概念。C#命名空间与 C++命名空间或 Java 的包基本相同，但 Visual Basic6 中没有对应的概念。namespace 关键字声明了应与类相关的命名空间。其后花括号中的所有代码都被认为是在这个命名空间中。编译器在 using 指令指定的命名空间中查找没有在当前命名空间中定义、但在代码中引用的类。这非常类似于 Java 中的 import 语句和 C++中的 using namespace 语句。

```
using System;
```

```
namespace Wrox.ProCSharp.Basics
{
```

在 First.cs 文件中使用 using 指令的原因是下面要使用一个库类 System.Console。using System 指令允许把这个类简写为 Console(类似于 System 命名空间中的其他类)。标准的 System 命名空间包含了最常用的.NET 类型。我们用 C#做的所有工作都依赖于.NET 基类，认识到这一点是非常重要的；在本例中，我们使用了 System 命名空间中的 Console 类，以写入控制台窗口。

注意：

几乎所有的 C#程序都使用 System 命名空间中的类，所以假定本章所有的代码文件都包含 using System;语句。

C#没有用于输入和输出的内置关键字，而是完全依赖于.NET 类。

接着，声明一个类 MyFirstClass。但是，因为该类位于 Wrox.ProCSharp.Basics 命名空间中，所以其完整的名称是 Wrox.ProCSharp.Basics.MyFirstCSharpClass：

```
class MyFirstCSharpClass
{
```

与 Java 一样，所有的 C#代码都必须包含在一个类中，C#中的类类似于 Java 和 C++中的类，大致相当于 Visual Basic6 中的类模块。类的声明包括 class 关键字，其后是类名和一对花括号。与类相关的所有代码都应放在这对花括号中。

下面声明方法 Main()。每个 C#可执行文件(例如控制台应用程序、Windows 应用程序和 Windows 服务)都必须有一个入口点—— Main 方法(注意 M 大写)：

```
static void Main()
{
```

这个方法在程序启动时调用，类似于 C++和 Java 中的 main 函数，或 Visual Basic6 模块中的 Sub Main。该方法要么没有返回值 void，要么返回一个整数(int)。C#方法对应于 C++ 和 Java 中的方法(有时把 C++中的方法称为成员函数)，它还对应于 Visual Basic 的 Function 或 Visual Basic 的 Sub，这取决于方法是否有返回值(与 Visual Basic 不同，C#的函数和子例程没有概念上的区别)。

注意，C#中的方法定义如下所示：

```
[modifiers] return_type MethodName([parameters])
{
   // Method body. NB. This code block is pseudo-code
}
```

第一个方括号中的内容表示可选关键字。修饰符(modifiers)用于指定用户所定义的方法的某些特性，例如可以在什么地方调用该方法。在本例中，有两个修饰符 public 和 static。修饰符 public 表示可以在任何地方访问该方法，所以可以在类的外部调用它。这与 C++和 Java 中的 public 相同，与 Visual Basic 中的 Public 相同。修饰符 static 表示方法不能在类的

实例上执行，因此不必先实例化类再调用。这是非常重要的，因为我们创建的是一个可执行文件，而不是类库。这与 C++和 Java 中的 static 关键字相同，但 Visual Basic 中没有对应的关键字(在 Visual Basic 中，Static 关键字有不同的含义)。把返回类型设置为 void，在本例中，不包含任何参数。

最后，看看代码语句。

```
Console.WriteLine("This isn't at all like Java!");
Console.ReadLine();
return;
```

在本例中，我们只调用了 System.Console 类的 WriteLine()方法，把一行文本写到控制台窗口上。WriteLine()是一个静态方法，在调用之前不需要实例化 Console 对象。

Console.ReadLine()读取用户的输入，添加这行代码会让应用程序等待用户按下回车键，之后退出应用程序。在 Visual Studio 2005 中，控制台窗口会消失。

然后调用 return 退出该方法(因为这是 Main 方法)。在方法的首部指定 void，因此没有返回值。Return 语句等价于 C++和 Java 中的 return，也等价于 Visual Basic 中的 Exit Sub 或 Exit Function。

对 C#基本语法有了大致的认识后，下面就要详细讨论 C#的各个方面了。因为没有变量是不可能编写出重要的程序的，所以首先介绍 C#中的变量。

2.3　变量

在 C#中声明变量使用下述语法：

```
datatype identifier;
```

例如：

```
int i;
```

该语句声明 int 变量 i。编译器不会让我们使用这个变量，除非我们用一个值初始化了该变量。但这个声明会在堆栈中给它分配 4 个字节，以保存其值。

声明 i 之后，就可以使用赋值运算符(=)给它分配一个值：

```
i = 10;
```

还可以在一行代码中声明变量，并初始化它的值：

```
int i = 10;
```

其语法与 C++和 Java 语法相同，但与 Visual Basic 中声明变量的语法完全不同。如果用户以前使用的是 Visual Basic6，应记住 C#不区分对象和简单的类型，所以不需要类似 Set 的关键字，即使是要把变量指向一个对象，也不需要 Set 关键字。无论变量的数据类型是什么，声明变量的 C#语法都是相同的。

如果在一个语句中声明和初始化了多个变量，那么所有的变量都具有相同的数据类型：

```
int x = 10, y =20;   // x and y are both ints
```

要声明类型不同的变量，需要使用单独的语句。在多个变量的声明中，不能指定不同的数据类型：

```
int x = 10;
bool y = true;          // Creates a variable that stores true or false
int x = 10, bool y = true;   // This won't compile!
```

注意上面例子中的//和其后的文本，它们是注释。//字符串告诉编译器，忽略其后的文本。本章后面会详细讨论代码中的注释。

2.3.1 变量的初始化

变量的初始化是 C#强调安全性的另一个例子。简单地说，C#编译器需要用某个初始值对变量进行初始化，之后才能在操作中引用该变量。大多数现代编译器把没有初始化标记为警告，但 C#编译器把它当作错误来看待。这就可以防止我们无意中从其他程序遗留下来的内存中获取垃圾值。

C#有两个方法可确保变量在使用前进行了初始化：

- 变量是类或结构中的字段，如果没有显式初始化，在默认状态下创建这些变量时，其值就是 0。
- 方法的局部变量必须在代码中显式初始化，之后才能在语句中使用它们的值。此时，初始化不是在声明该变量时进行的，但编译器会通过方法检查所有可能的路径，如果检测到局部变量在初始化之前就使用了它的值，就会产生错误。

C#的方法与 C++的方法相反，在 C++中，编译器让程序员确保变量在使用之前进行了初始化，在 Visual Basic 中，所有的变量都会自动把其值设置为 0。

例如，在 C#中不能使用下面的语句：

```
public static int Main()
{
   int d;
   Console.WriteLine(d);   // Can't do this! Need to initialize d before use
   return 0;
}
```

注意在这段代码中，演示了如何定义 Main()，使之返回一个 int 类型的数据，而不是 void。在编译这些代码时，会得到下面的错误消息：

```
Use of unassigned local variable 'd'
```

同样的规则也适用于引用类型。考虑下面的语句：

```
Something objSomething;
```

在 C++中，上面的代码会在堆栈中创建 Something 类的一个实例。在 C#中，这行代码

仅会为 Something 对象创建一个引用，但这个引用还没有指向任何对象。对该变量调用方法或属性会导致错误。

在 C#中实例化一个引用对象需要使用 new 关键字。如上所述，创建一个引用，使用 new 关键字把该引用指向存储在堆上的一个对象：

```
objSomething = new Something();   // This creates a Something on the heap
```

2.3.2　变量的作用域

变量的作用域是可以访问该变量的代码区域。一般情况下，确定作用域有以下规则：

- 只要类在某个作用域内，其字段(也称为成员变量)也在该作用域内(在 C++、Java 和 Visual Basic 中也是这样)。
- 局部变量存在于表示声明该变量的块语句或方法结束的封闭花括号之前的作用域内。
- 在 for、while 或类似语句中声明的局部变量存在于该循环体内(C++程序员注意，这与 C++的 ANSI 标准相同。Microsoft C++编译器的早期版本不遵守该标准，在循环停止后这种变量仍存在)。

1. 局部变量的作用域冲突

大型程序在不同部分为不同的变量使用相同的变量名是很常见的。只要变量的作用域是程序的不同部分，就不会有问题，也不会产生模糊性。但要注意，同名的局部变量不能在同一作用域内声明两次，所以不能使用下面的代码：

```
int x = 20;
// some more code
int x = 30;
```

考虑下面的代码示例：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   public class ScopeTest
   {
      public static int Main()
      {
         for (int i = 0; i < 10; i++)
         {
            Console.WriteLine(i);
         }  // i goes out of scope here

         // We can declare a variable named i again, because
         // there's no other variable with that name in scope

         for (int i = 9; i >= 0; i--)
```

```
            {
               Console.WriteLine(i);
            }  // i goes out of scope here
            return 0;
         }
      }
   }
```

这段代码使用一个 for 循环打印出从 0~9 的数字，再打印从 9~0 的数字。重要的是在同一个方法中，代码中的变量 i 声明了两次。可以这么做的原因是在两次声明中，i 都是在循环内部声明的，所以变量 i 对于循环来说是局部变量。

下面是另一个例子：

```
      public static int Main()
      {
         int j = 20;
         for (int i = 0; i < 10; i++)
         {
            int j = 30;   // Can't do this - j is still in scope
            Console.WriteLine(j + i);
         }
         return 0;
      }
```

如果试图编译它，就会产生如下错误：

```
ScopeTest.cs(12,14): error CS0136: A local variable named 'j' cannot be
declared in this scope because it would give a different meaning to 'j', which
is already used in a 'parent or current' scope to denote something else
```

其原因是：变量 j 是在 for 循环开始前定义的，在执行 for 循环时应处于其作用域内，在 Main 方法结束执行后，变量 j 才超出作用域，第二个 j(不合法)则在循环的作用域内，该作用域嵌套在 Main 方法的作用域内。编译器无法区别这两个变量，所以不允许声明第二个变量。这也是与 C++不同的地方，在 C++中，允许隐藏变量。

2. 字段和局部变量的作用域冲突

在某些情况下，可以区分名称相同(尽管其完全限定的名称不同)、作用域相同的两个标识符。此时编译器允许声明第二个变量。原因是 C#在变量之间有一个基本的区分，它把声明为类型级的变量看作字段，而把在方法中声明的变量看作局部变量。

考虑下面的代码：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class ScopeTest2
   {
```

```
        static int j = 20;

        public static void Main()
        {
           int j = 30;
           Console.WriteLine(j);
           return;
        }
     }
  }
```

即使在 Main 方法的作用域内声明了两个变量 j，这段代码也会编译—— j 被定义在类级上，在该类删除前是不会超出作用域的(在本例中，当 Main 方法中断，程序结束时，才会删除该类)。此时，在 Main 方法中声明的新变量 j 隐藏了同名的类级变量，所以在运行这段代码时，会显示数字 30。

但是，如果要引用类级变量，该怎么办？可以使用语法 object.fieldname，在对象的外部引用类的字段或结构。在上面的例子中，我们访问静态方法中的一个静态字段(静态字段详见下一节)，所以不能使用类的实例，只能使用类本身的名称：

```
  …
        public static void Main()
        {
           int j = 30;
           Console.WriteLine(ScopeTest2.j);
        }
        ...
```

如果要访问一个实例字段(该字段属于类的一个特定实例)，就需要使用 this 关键字。this 的作用与 C++和 Java 中的 this 相同，与 Visual Basic 中的 Me 相同。

2.3.3　常量

在声明和初始化变量时，在变量的前面加上关键字 const，就可以把该变量指定为一个常量。顾名思义，常量是其值在使用过程中不会发生变化的变量：

```
  const int a = 100;   // This value cannot be changed
```

Visual Basic 和 C++开发人员非常熟悉常量。但 C++开发人员应注意，C#不支持 C++常量的所有细微的特性。在 C++中，变量不仅可以声明为常量，而且根据声明，还可以有常量指针，指向常量的变量指针、常量方法(不改变包含对象的内容)，方法的常量参数等。这些细微的特性在 C#中都删除了，只能把局部变量和字段声明为常量。

常量具有如下特征：

- 常量必须在声明时初始化。指定了其值后，就不能再修改了。

- 常量的值必须能在编译时用于计算。因此，不能用从一个变量中提取的值来初始化常量。如果需要这么做，应使用只读字段(详见第 3 章)。
- 常量总是静态的。但注意，不必(实际上，是不允许)在常量声明中包含修饰符 static。

在程序中使用常量至少有 3 个好处：

- 常量用易于理解的清楚的名称替代了含义不明确的数字或字符串，使程序更易于阅读。
- 常量使程序更易于修改。例如，在 C#程序中有一个 SalesTax 常量，该常量的值为 6%。如果以后销售税率发生变化，把新值赋给这个常量，就可以修改所有的税款计算结果，而不必查找整个程序，修改税率为 0.06 的每个项。
- 常量更容易避免程序出现错误。如果要把另一个值赋给程序中的一个常量，而该常量已经有了一个值，编译器就会报告错误。

2.4 预定义数据类型

前面介绍了如何声明变量和常量，下面要详细讨论 C#中可用的数据类型。与其他语言相比，C#对其可用的类型及其定义有更严格的描述。

2.4.1 值类型和引用类型

在开始介绍 C#中的数据类型之前，理解 C#把数据类型分为两种是非常重要的：

- 值类型
- 引用类型

下面几节将详细介绍值类型和引用类型的语法。从概念上看，其区别是值类型直接存储其值，而引用类型存储对值的引用。与其他语言相比，C#中的值类型基本上等价于 Visual Basic 或 C++中的简单类型(整型、浮点型，但没有指针或引用)。引用类型与 Visual Basic 中的引用类型相同，与 C++中通过指针访问的类型类似。

这两种类型存储在内存的不同地方：值类型存储在堆栈中，而引用类型存储在托管堆上。注意区分某个类型是值类型还是引用类型，因为这种存储位置的不同会有不同的影响。例如，int 是值类型，这表示下面的语句会在内存的两个地方存储值 20：

```
// i and j are both of type int
i = 20;
j = i;
```

但考虑下面的代码。这段代码假定已经定义了一个类 Vector，Vector 是一个引用类型，它有一个 int 类型的成员变量 Value：

```
Vector x, y;
x = new Vector ();
x.Value = 30;   // Value is a field defined in Vector class
```

```
    y = x;
    Console.WriteLine(y.Value);
    y.Value = 50;
    Console.WriteLine(x.Value);
```

要理解的重要一点是在执行这段代码后，只有一个 Vector 对象。x 和 y 都指向包含该对象的内存位置。因为 x 和 y 是引用类型的变量，声明这两个变量只保留了一个引用——而不会实例化给定类型的对象。这与在 C++中声明指针和 Visual Basic 中的对象引用是相同的——在 C++和 Visual Basic 中，都不会创建对象。要创建对象，就必须使用 new 关键字，如上所示。因为 x 和 y 引用同一个对象，所以对 x 的修改会影响 y，反之亦然。因此上面的代码会显示 30 和 50。

注意：

C++开发人员应注意，这个语法类似于引用，而不是指针。我们使用.(句点)符号，而不是->来访问对象成员。在语法上，C#引用看起来更类似于 C++引用变量。但是，抛开表面的语法，实际上它类似于 C++指针。

如果变量是一个引用，就可以把其值设置为 null，表示它不引用任何对象：

```
    y = null;
```

这类似于 Java 中把引用设置为 null，C++中把指针设置为 NULL，或 Visual Basic 中把对象引用设置为 Nothing。如果将引用设置为 null，显然就不可能对它调用任何非静态的成员函数或字段，这么做会在运行期间抛出一个异常。

在像 C++这样的语言中，开发人员可以选择是直接访问某个给定的值，还是通过指针来访问。Visual Basic 的限制更多：COM 对象是引用类型，简单类型总是值类型。C#在这方面类似于 Visual Basic：变量是值还是引用仅取决于其数据类型，所以，int 总是值类型。不能把 int 变量声明为引用(在第 6 章介绍装箱时，可以在类型为 object 的引用中封装值类型)。

在 C#中，基本数据类型如 bool 和 long 都是值类型。如果声明一个 bool 变量，并给它赋予另一个 bool 变量的值，在内存中就会有两个 bool 值。如果以后修改第一个 bool 变量的值，第二个 bool 变量的值也不会改变。这些类型是通过值来复制的。

相反，大多数更复杂的 C#数据类型，包括我们自己声明的类都是引用类型。它们分配在堆中，其生存期可以跨多个函数调用，可以通过一个或几个别名来访问。CLR 执行一种精细的算法，来跟踪哪些引用变量仍是可以访问的，哪些引用变量已经不能访问了。CLR 会定期删除不能访问的对象，把它们占用的内存返回给操作系统。这是通过垃圾收集器实现的。

把基本类型(如 int 和 bool)规定为值类型，而把包含许多字段的较大类型(通常在有类的情况下)规定为引用类型，C#设计这种方式的原因是可以得到最佳性能。如果要把自己的类型定义为值类型，就应把它声明为一个结构。

2.4.2　CTS 类型

如第 1 章所述，C#认可的基本预定义类型并没有内置于 C#语言中，而是内置于.NET

Framework 中。例如，在 C#中声明一个 int 类型的数据时，声明的实际上是.NET 结构 System.Int32 的一个实例。这听起来似乎很深奥，但其意义深远：这表示在语法上，可以把所有的基本数据类型看作是支持某些方法的类。例如，要把 int i 转换为 string，可以编写下面的代码：

```
string s = i.ToString();
```

应强调的是，在这种便利语法的背后，类型实际上仍存储为基本类型。基本类型在概念上用.NET 结构表示，所以肯定没有性能损失。

下面看看 C#中定义的类型。我们将列出每个类型，以及它们的定义和对应.NET 类型(CTS 类型)的名称。C#有 15 个预定义类型，其中 13 个是值类型，2 个是引用类型(string 和 object)。

2.4.3 预定义的值类型

内置的值类型表示基本数据类型，例如整型和浮点类型、字符类型和布尔类型。

1. 整型

C#支持 8 个预定义整数类型，如表 2-1 所示。

表 2-1

名 称	CTS 类 型	说 明	范 围
sbyte	System.SByte	8 位有符号的整数	–128~127 (-2^7~2^7-1)
short	System.Int16	16 位有符号的整数	–32 768~32 767 (-2^{15}~$2^{15}-1$)
int	System.Int32	32 位有符号的整数	–2 147 483 648~2 147 483 647(-2^{31}~$2^{31}-1$)
long	System.Int64	64 位有符号的整数	–9 223 372 036 854 775 808~9 223 372 036 854 775 807(-2^{63}~$2^{63}-1$)
byte	System.Byte	8 位无符号的整数	0~255(0~2^8-1)
ushort	System.Uint16	16 位无符号的整数	0~65535(0~$2^{16}-1$)
uint	System.Uint32	32 位无符号的整数	0~4 294 967 295(0~$2^{32}-1$)
ulong	System.Uint64	64 位无符号的整数	0~18 446 744 073 709 551 615(0~$2^{64}-1$)

Windows 的将来版本将支持 64 位处理器，可以把更大的数据块移入移出内存，获得更快的处理速度。因此，C#支持 8~64 位的有符号和无符号的整数。

当然，Visual Basic 开发人员会发现有许多类型名称是新的。C++和 Java 开发人员应注意：一些 C#类型的名称与 C++和 Java 类型一致，但其定义不同。例如，在 C#中，int 总是 32 位带符号的整数，而在 C++中，int 是带符号的整数，但其位数取决于平台(在 Windows 上是 32 位)。在 C#中，所有的数据类型都以与平台无关的方式定义，以备将来 C#和.NET 迁移到其他平台上。

byte 是 0~255(包括 255)的标准 8 位类型。注意，在强调类型的安全性时，C#认为 byte 类型和 char 类型完全不同，它们之间的编程转换必须显式写出。还要注意，与整数中的其他类型不同，byte 类型在默认状态下是无符号的，其有符号的版本有一个特殊的名称 sbyte。

在.NET 中，short 不再很短，现在它有 16 位，Int 类型更长，有 32 位。 long 类型最长，有 64 位。所有整数类型的变量都能赋予十进制或十六进制的值，后者需要 0x 前缀：

```
long x = 0x12ab;
```

如果对一个整数是 int、uint、long 或是 ulong 没有任何显式的声明，则该变量默认为 int 类型。为了把键入的值指定为其他整数类型，可以在数字后面加上如下字符：

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

也可以使用小写字母 u 和 l，但后者会与整数 1 混淆。

2. 浮点类型

C#提供了许多整型数据类型，也支持浮点类型，如表 2-2 所示。C 和 C++程序员很熟悉它们。

表　2-2

名　称	CTS 类 型	说　明	位　数	范围(大致)
float	System.Single	32 位单精度浮点数	7	$\pm1.5 \times 10^{-45}$~$\pm3.4 \times 10^{38}$
double	System.Double	64 位双精度浮点数	15/16	$\pm5.0 \times 10^{-324}$~$\pm1.7 \times 10^{308}$

float 数据类型用于较小的浮点数，因为它要求的精度较低。double 数据类型比 float 数据类型大，提供的精度也大一倍(15 位)。

如果在代码中没有对某个非整数值(如 12.3)硬编码，则编译器一般假定该变量是 double。如果想指定该值为 float，可以在其后加上字符 F(或 f)：

```
float f = 12.3F;
```

3. decimal 类型

另外，decimal 类型表示精度更高的浮点数，如表 2-3 所示。

表　2-3

名　称	CTS 类 型	说　明	位　数	范围(大致)
decimal	System. Decimal	128 位高精度十进制数 表示法	28	$\pm1.0\times10^{-28}$~$\pm7.9 \times 10^{28}$

CTS 和 C#一个重要的优点是提供了一种专用类型进行财务计算，这就是 decimal 类型，使用 decimal 类型提供的 28 位的方式取决于用户。换言之，可以用较大的精确度(带有美分)来表示较小的美元值，也可以在小数部分用更多的舍入来表示较大的美元值。但应注意，decimal 类型不是基本类型，所以在计算时使用该类型会有性能损失。

要把数字指定为 decimal 类型，而不是 double、 float 或整型，可以在数字的后面加上字符 M(或 m)，如下所示。

```
decimal d = 12.30M;
```

4. bool 类型

C#的 bool 类型用于包含布尔值 true 或 false，如表 2-4 所示。

表 2-4

名 称	CTS 类 型	值
bool	System.Boolean	true 或 false

bool 值和整数值不能互换。如果变量(或函数的返回类型)声明为 bool 类型，就只能使用值 true 或 false。如果试图使用 0 表示 false，非 0 值表示 true，就会出错。

5. 字符类型

为了保存单个字符的值，C#支持 char 数据类型，如表 2-5 所示。

表 2-5

名 称	CTS 类 型	值
char	System.Char	表示一个 16 位的(Unicode)字符

虽然这个数据类型在表面上类似于 C 和 C++中的 char 类型，但它们有重大区别。C++的 char 表示一个 8 位字符，而 C#的 char 包含 16 位。其部分原因是不允许在 char 类型与 8 位 byte 类型之间进行隐式转换。

尽管 8 位足够编码英语中的每个字符和数字 0~9 了，但它们不够编码更大的符号系统中的每个字符(例如中文)。为了面向全世界，计算机行业正在从 8 位字符集转向 16 位的 Unicode 模式，ASCII 编码是 Unicode 的一个子集。

char 类型的字面量是用单引号括起来的，例如'A'。如果把字符放在双引号中，编译器会把它看作是字符串，从而产生错误。

除了把 char 表示为字符字面量之外，还可以用 4 位十六进制的 Unicode 值(例如'\u0041')，带有数据类型转换的整数值(例如(char)65)，或十六进制数('\x0041')表示它们。它们还可以用转义序列表示，如表 2-6 所示。

表　2-6

转 义 序 列	字　　符
\'	单引号
\"	双引号
\\	反斜杠
\0	空
\a	警告
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符

C++开发人员应注意，因为 C#本身有一个 string 类型，所以不需要把字符串表示为 char 类型的数组。

2.4.4　预定义的引用类型

C#支持两个预定义的引用类型，如表 2-7 所示。

表　2-7

名　　称	CTS 类	说　　明
object	System.Object	根类型，CTS 中的其他类型都是从它派生而来的(包括值类型)
string	System.String	Unicode 字符串

1. object 类型

许多编程语言和类结构都提供了根类型，层次结构中的其他对象都从它派生而来。C#和.NET 也不例外。在 C#中，object 类型就是最终的父类型，所有内置类型和用户定义的类型都从它派生而来。这是 C#的一个重要特性，它把 C#与 Visual Basic 和 C++区分开来，但其行为与 Java 非常类似。所有的类型都隐含地最终派生于 System.Object 类，这样，object 类型就可以用于两个目的：

- 可以使用 object 引用绑定任何子类型的对象。例如，第 6 章将说明如何使用 object 类型把堆栈中的一个值对象装箱，再移动到堆中。对象引用也可以用于反射，此时必须有代码来处理类型未知的对象。这类似于 C++中的 void 指针或 Visual Basic 中的 Variant 数据类型。

- object 类型执行许多一般用途的基本方法，包括 Equals()、GetHashCode()、GetType()和 ToString()。用户定义的类需要使用一种面向对象技术——重写(见第 4 章)，提供其中一些方法的替代执行代码。例如，重写 ToString()时，要给类提供一个方法，给出类本身的字符串表示。如果类中没有提供这些方法的实现代码，编译器就会使用 object 类型中的实现代码，它们在类中的执行不一定正确。

后面的章节将详细讨论 object 类型。

2. string 类型

有 C 和 C++开发经验的人员可能在使用 C 风格的字符串时不太顺利。C 或 C++字符串不过是一个字符数组，因此客户机程序员必须做许多工作，才能把一个字符串复制到另一个字符串上，或者连接两个字符串。实际上，对于一般的 C++程序员来说，执行包装了这些操作细节的字符串类是一个非常头痛的耗时过程。Visual Basic 程序员的工作就比较简单，只需使用 string 类型即可。而 Java 程序员就更幸运了，其 String 类在许多方面都类似于 C#字符串。

C#有 string 关键字，在翻译为.NET 类时，它就是 System.String。有了它，像字符串连接和字符串复制这样的操作就很简单了：

```
string str1 = "Hello ";
string str2 = "World";
string str3 = str1 + str2; // string concatenation
```

尽管这是一个值类型的赋值，但 string 是一个引用类型。String 对象保留在堆上，而不是堆栈上。因此，当把一个字符串变量赋给另一个字符串时，会得到对内存中同一个字符串的两个引用。但是，string 与引用类型在常见的操作上有一些区别。例如，修改其中一个字符串，就会创建一个全新的 string 对象，而另一个字符串没有改变。考虑下面的代码：

```
using System;

class StringExample
{
   public static int Main()
   {
      string s1 = "a string";
      string s2 = s1;
      Console.WriteLine("s1 is " + s1);
      Console.WriteLine("s2 is " + s2);
      s1 = "another string";
      Console.WriteLine("s1 is now " + s1);
      Console.WriteLine("s2 is now " + s2);
      return 0;
   }
}
```

其输出结果为：

```
s1 is a string
```

```
s2 is a string
s1 is now another string
s2 is now a string
```

换言之，改变 s1 的值对 s2 没有影响，这与我们期待的引用类型正好相反。当用值"a string"初始化 s1 时，就在堆上分配了一个新的 string 对象。在初始化 s2 时，引用也指向这个对象，所以 s2 的值也是"a string"。但是现在要改变 s1 的值，而不是替换原来的值时，堆上就会为新值分配一个新对象。s2 变量仍指向原来的对象，所以它的值没有改变。这实际上是运算符重载的结果，运算符重载详见第 6 章。基本上，string 类实现为其语义遵循一般的、直观的字符串规则。

字符串字面量放在双引号中("...")；如果试图把字符串放在单引号中，编译器就会把它当作 char，从而引发错误。C#字符串和 char 一样，可以包含 Unicode、十六进制数转义序列。因为这些转义序列以一个反斜杠开头，所以不能在字符串中使用这个非转义的反斜杠字符，而需要用两个反斜杠字符(\\)来表示它：

```
string filepath = "C:\\ProCSharp\\First.cs";
```

即使用户相信自己可以在任何情况下都记住要这么做，但键入两个反斜杠字符会令人迷惑。幸好，C#提供了另一种替代方式。可以在字符串字面量的前面加上字符@，在这个字符后的所有字符都看作是其原来的含义——它们不会解释为转义字符：

```
string filepath = @"C:\ProCSharp\First.cs";
```

甚至允许在字符串字面量中包含换行符：

```
string jabberwocky = @"'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.";
```

那么 jabberwocky 的值就是：

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

2.5　流控制

本节将介绍 C#语言的重要语句：控制程序流的语句，它们不是按代码在程序中的排列位置顺序执行的。

2.5.1　条件语句

条件语句可以根据条件是否满足或根据表达式的值控制代码的执行分支。C#有两个控制代码分支的结构：if 语句，测试特定条件是否满足；switch 语句，它比较表达式和许多不同的值。

1. if 语句

对于条件分支，C#继承了 C 和 C++的 if···else 结构。对于用过程语言编程的人来说，其语法是非常直观的：

```
if (condition)
   statement(s)
else
   statement(s)
```

如果在条件中要执行多个语句，就需要用花括号({ ··· })把这些语句组合为一个块(这也适用于其他可以把语句组合为一个块的 C#结构，例如 for 和 while 循环)。

```
bool isZero;
if (i == 0)
{
   isZero = true;
   Console.WriteLine("i is Zero");
}
else
{
   isZero = false;
   Console.WriteLine("i is Non-zero");
}
```

其语法与 C++和 Java 类似，但与 Visual Basic 不同。Visual Basic 开发人员注意，C#中没有与 Visual Basic 的 EndIf 对应的语句，其规则是 if 的每个子句都只包含一个语句。如果需要多个语句，如上面的例子所示，就应把这些语句放在花括号中，这会把整组语句当作一个语句块来处理。

还可以单独使用 if 语句，不加 else 语句。也可以合并 else if 子句，测试多个条件。

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class MainEntryPoint
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Type in a string");
         string input;
         input = Console.ReadLine();
         if (input == "")
         {
            Console.WriteLine("You typed in an empty string");
         }
         else if (input.Length < 5)
```

```
            {
               Console.WriteLine("The string had less than 5 characters");
            }
            else if (input.Length < 10)
            {
               Console.WriteLine("The string had at least 5 but less than 10
                  characters");
            }
            Console.WriteLine("The string was " + input);
         }
      }
   }
```

添加到 if 子句中的 else if 语句的个数没有限制。

注意在上面的例子中，我们声明了一个字符串变量 input，让用户在命令行上输入文本，把文本填充到 input 中，然后测试该字符串变量的长度。代码还说明了在 C#中如何进行字符串处理。例如，要确定 input 的长度，可以使用 input.Length。

对于 if，要注意的一点是如果条件分支中只有一条语句，就无需使用花括号：

```
   if (i == 0)Let's add some brackets here.
      Console.WriteLine("i is Zero");        // This will only execute if i == 0
   Console.WriteLine("i can be anything"); // Will execute whatever the
                                           // value of i
```

但是，为了保持一致，许多程序员只要使用 if 语句，就加上花括号。

前面介绍的 if 语句还演示了用于比较数值的一些 C#运算符。特别注意，与 C++和 Java 一样，C#使用“==”对变量进行等于比较。此时不要使用“=”，“=”用于赋值。

在 C#中，if 子句中的表达式必须等于布尔值。C++程序员应特别注意这一点；与 C++不同，C#中的 if 语句不能直接测试整数(例如从函数中返回的值)，而必须明确地把返回的整数转换为布尔值 true 或 false，例如，比较值 0 和 null：

```
   if (DoSomething() != 0)
   {
      // Non-zero value returned
   }
   else
   {
      // Returned zero
   }
```

这个限制用于防止 C++中某些常见的运行错误，特别是在 C++中，当应使用“==”时，常常误输入“＝”，导致不希望的赋值。在 C#中，这常常会导致一个编译错误，因为除非在处理 bool 值，否则“＝”不会返回 bool。

2. switch 语句

switch…case 语句适合于从一组互斥的分支中选择一个执行分支。C++和 Java 程序员

应很熟悉它，该语句类似于 Visual Basic 中的 Select Case 语句。

其形式是 switch 参数的后面跟一组 case 子句。如果 switch 参数中表达式的值等于某个 case 子句旁边的某个值，就执行该 case 子句中的代码。此时不需要使用花括号把语句组合到块中；只需使用 break 语句标记每个 case 代码的结尾即可。也可以在 switch 语句中包含一个 default 子句，如果表达式不等于任何 case 子句的值，就执行 default 子句的代码。下面的 switch 语句测试 integerA 变量的值：

```
switch (integerA)
{
   case 1:
      Console.WriteLine("integerA =1");
      break;
   case 2:
      Console.WriteLine("integerA =2");
      break;
   case 3:
      Console.WriteLine("integerA =3");
      break;
   default:
      Console.WriteLine("integerA is not 1,2, or 3");
      break;
}
```

注意 case 的值必须是常量表达式——不允许使用变量。

C 和 C++程序员应很熟悉 switch…case 语句，而 C#的 switch…case 语句更安全。特别是它禁止所有 case 中的失败条件。如果激活了块中靠前的一个 case 子句，后面的 case 子句就不会被激活，除非使用 goto 语句特别标记要激活后面的 case 子句。编译器会把没有 break 语句的 case 子句标记为错误：

```
Control cannot fall through from one case label ('case 2:') to another
```

在有限的几种情况下，这种失败是允许的，但在大多数情况下，我们不希望出现这种失败，而且这会导致出现很难察觉的逻辑错误。让代码正常工作，而不是出现异常，这样不是更好吗？

但在使用 goto 语句时(C#支持)，会在 switch…cases 中重复出现失败。如果确实想这么做，就应重新考虑设计方案了。下面的代码说明了如何使用 goto 模拟失败，得到的代码会非常混乱：

```
// assume country and language are of type string
switch(country)
{
   case "America":
      CallAmericanOnlyMethod();
      goto case "Britain";
   case "France":
      language = "French";
      break;
```

```
      case "Britain":
         language = "English";
         break;
   }
```

但这有一种例外情况。如果一个 case 子句为空，就可以从这个 case 跳到下一个 case 上，这样就可以用相同的方式处理两个或多个 case 子句了(不需要 goto 语句)。

```
   switch(country)
   {
      case "au":
      case "uk":
      case "us":
         language = "English";
         break;
      case "at":
      case "de":
         language = "German";
         break;
   }
```

在 C#中，switch 语句的一个有趣的地方是 case 子句的排放顺序是无关紧要的，甚至可以把 default 子句放在最前面！因此，任何两个 case 都不能相同。这包括值相同的不同常量，所以不能这样编写：

```
   // assume country is of type string
   const string england = "uk";
   const string britain = "uk";
   switch(country)
   {
      case england:
      case britain:      // this will cause a compilation error
         language = "English";
         break;
   }
```

上面的代码还说明了 C#中的 switch 语句与 C++中的 switch 语句的另一个不同之处：在 C#中，可以把字符串用作测试变量。

2.5.2　循环

C#提供了 4 种不同的循环机制(for、while、do...while 和 foreach)，在满足某个条件之前，可以重复执行代码块。for、while 和 do...while 循环与 C++中的对应循环相同。

1. for 循环

C#的 for 循环提供的迭代循环机制是在执行下一次迭代前，测试是否满足某个条件，其语法如下：

```
for (initializer; condition; iterator)
   statement(s)
```

其中：

- initializer 是指在执行第一次迭代前要计算的表达式(通常把一个局部变量初始化为循环计数器)；
- condition 是在每次迭代新循环前要测试的表达式(它必须等于 true，才能执行下一次迭代)；
- iterator 是每次迭代完要计算的表达式(通常是递增循环计数器)。当 condition 等于 false 时，迭代停止。

for 循环是所谓的预测试循环，因为循环条件是在执行循环语句前计算的，如果循环条件为假，循环语句就根本不会执行。

for 循环非常适合于一个语句或语句块重复执行预定的次数。下面的例子就是 for 循环的典型用法，这段代码输出从 0~99 的整数：

```
for (int i = 0; i < 100; i = i+1)   // this is equivalent to
                        // For i = 0 To 99 in VB.
{
   Console.WriteLine(i);
}
```

这里声明了一个 int 类型的变量 i，并把它初始化为 0，用作循环计数器。接着测试它是否小于 100。因为这个条件等于 true，所以执行循环中的代码，显示值 0。然后给该计数器加 1，再次执行该过程。当 i 等于 100 时，循环停止。

实际上，上述编写循环的方式并不常用。C#在给变量加 1 时有一种简化方式，即不使用 i = i+1，而简写为 i++：

```
for (int i = 0; i < 100; i++)
{
//etc.}
```

C#的 for 循环语法比 Visual Basic 中的 For…Next 循环的功能强大得多，因为迭代器可以是任何语句。在 Visual Basic 中，只能对循环控制变量加减某个数字。在 C#中，则可以做任何事，例如，让循环控制变量乘以 2。

嵌套的 for 循环非常常见，在每次迭代外部的循环时，内部循环都要彻底执行完毕。这种模式通常用于在矩形多维数组中遍历每个元素。最外部的循环遍历每一行，内部的循环遍历某行上的每个列。下面的代码显示数字行，它还使用另一个 Console 方法 Console.Write()，该方法的作用与 Console.WriteLine()相同，但不在输出中添加回车换行符：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class MainEntryPoint
   {
```

```
        static void Main(string[ ] args)
        {
           // This loop iterates through rows...
           for (int i = 0; i < 100; i+=10)
           {
              // This loop iterates through columns...
              for (int j = i; j < i + 10; j++)
              {
                 Console.Write("  " + j);
              }
              Console.WriteLine();
           }
        }
     }
  }
```

尽管 j 是一个整数，但它会自动转换为字符串，以便进行连接。C++开发人员要注意，这比在 C++中处理字符串容易得多，Visual Basic 开发人员则已经习惯于此了。

C 程序员应注意上述例子中的一个特殊功能。在每次迭代外部的循环时，内部循环的计数器变量都要重新声明。这种语法不仅在 C#中可行，在 C++中也是合法的。

上述例子的结果是：

csc NumberTable.cs

```
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.0.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
 50  51  52  53  54  55  56  57  58  59
 60  61  62  63  64  65  66  67  68  69
 70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89
 90  91  92  93  94  95  96  97  98  99
```

尽管在技术上，可以在 for 循环的测试条件中计算其他变量，而不计算计数器变量，但这不太常见。也可以在 for 循环中忽略一个表达式(甚或所有表达式)。但此时，要考虑使用 while 循环。

2. while 循环

while 循环与 C++和 Java 中的 while 循环相同，与 Visual Basic 中的 While...Wend 循环

相同。与 for 循环一样，while 也是一个预测试的循环。其语法是类似的，但 while 循环只有一个表达式：

```
while(condition)
  statement(s);
```

与 for 循环不同的是，while 循环最常用于下述情况：在循环开始前，不知道重复执行一个语句或语句块的次数。通常，在某次迭代中，while 循环体中的语句把布尔标记设置为 false，结束循环，如下面的例子所示。

```
bool condition = false;
while (!condition)
{
  // This loop spins until the condition is true.
  DoSomeWork();
  condition = CheckCondition();   // assume CheckCondition() returns a bool
}
```

所有的 C#循环机制，包括 while 循环，如果只重复执行一条语句，而不是一个语句块，都可以省略花括号。许多程序员都认为最好在任何情况下都加上花括号。

3. do…while 循环

do...while 循环是 while 循环的后测试版本。它与 C++和 Java 中的 do...while 循环相同，与 Visual Basic 中的 Loop...While 循环相同，该循环的测试条件要在执行完循环体之后执行。因此 do...while 循环适合于至少执行一次循环体的情况：

```
bool condition;
do
{
  // This loop will at least execute once, even if Condition is false.
  MustBeCalledAtLeastOnce();
  condition = CheckCondition();
} while (condition);
```

4. foreach 循环

foreach 循环是我们讨论的最后一种 C#循环机制。其他循环机制都是 C 和 C++的最早期版本，而 foreach 语句是新增的循环机制(借用于 Visual Basic)，也是非常受欢迎的一种循环。

foreach 循环可以迭代集合中的每一项。现在不必考虑集合的概念，第 10 章将介绍集合。知道集合是一种包含其他对象的对象即可。从技术上看，要使用集合对象，就必须支持 IEnumerable 接口。集合的例子有 C#数组、System.Collection 命名空间中的集合类，以及用户定义的集合类。从下面的代码中可以了解 foreach 循环的语法，其中假定 arrayOfInts 是一个整型数组：

```
foreach (int temp in arrayOfInts)
```

```
    {
      Console.WriteLine(temp);
    }
```

其中，foreach 循环每次迭代数组中的一个元素。它把每个元素的值放在 int 型的变量 temp 中，然后执行一次循环迭代。

注意，foreach 循环不能改变集合中各项(上面的 temp)的值，所以下面的代码不会编译：

```
    foreach (int temp in arrayOfInts)
    {
      temp++;
      Console.WriteLine(temp);
    }
```

如果需要迭代集合中的各项，并改变它们的值，就应使用 for 循环。

2.5.3　跳转语句

C#提供了许多可以立即跳转到程序中另一行代码的语句，在此，先介绍 goto 语句。

1. goto 语句

goto 语句可以直接跳转到程序中用标签指定的另一行(标签是一个标识符，后跟一个冒号)：

```
    goto Label1;
      Console.WriteLine("This won't be executed");
    Label1:
      Console.WriteLine("Continuing execution from here");
```

goto 语句有两个限制。不能跳转到像 for 循环这样的代码块中，也不能跳出类的范围，不能退出 try...catch 块后面的 finally 块(第 13 章将介绍如何用 try...catch...finally 块处理异常)。

goto 语句的名声不太好，在大多数情况下不允许使用它。一般情况下，使用它肯定不是面向对象编程的好方式。但是有一个地方使用它是相当方便的——在 switch 语句的 case 子句之间跳转，这是因为 C#的 switch 语句在故障处理方面非常严格。前面介绍了其语法。

2. break 语句

前面简要提到过 break 语句——在 switch 语句中使用它退出某个 case 语句。实际上，break 也可以用于退出 for、foreach、while 或 do...while 循环，该语句会使控制流执行循环后面的语句。

如果该语句放在嵌套的循环中，就执行最内部循环后面的语句。如果 break 放在 switch 语句或循环外部，就会产生编译错误。

3. continue 语句

continue 语句类似于 break，也必须在 for、foreach、while 或 do...while 循环中使用。

但它只退出循环的当前迭代，开始执行循环的下一次迭代，而不是退出循环。

4. return 语句

return 语句用于退出类的方法，把控制权返回方法的调用者，如果方法有返回类型，return 语句必须返回这个类型的值，如果方法没有返回类型，应使用没有表达式的 return 语句。

2.6 枚举

枚举是用户定义的整数类型。在声明一个枚举时，要指定该枚举可以包含的一组可接受的实例值。不仅如此，还可以给值指定易于记忆的名称。如果在代码的某个地方，要试图把一个不在可接受范围内的值赋予枚举的一个实例，编译器就会报告一个错误。这个概念对于 Visual Basic 程序员来说是新的。C++支持枚举，但 C#枚举要比 C++枚举强大得多。

从长远来看，创建枚举可以节省大量的时间，减少许多麻烦。使用枚举比使用无格式的整数至少有如下三个优势：

- 如上所述，枚举可以使代码更易于维护，有助于确保给变量指定合法的、期望的值。
- 枚举使代码更清晰，允许用描述性的名称表示整数值，而不是用含义模糊的数来表示。
- 枚举使代码更易于键入。在给枚举类型的实例赋值时，VS .NET IDE 会通过 IntelliSense 弹出一个包含可接受值的列表框，减少了按键次数，并能够让我们回忆起可选的值。

定义如下的枚举：

```
public enum TimeOfDay
{
   Morning = 0,
   Afternoon = 1,
   Evening = 2
}
```

本例在枚举中使用一个整数值，来表示一天的每个阶段。现在可以把这些值作为枚举的成员来访问。例如，TimeOfDay.Morning 返回数字 0。使用这个枚举一般是把合适的值传送给方法，或在 switch 语句中迭代可能的值。

```
class EnumExample
{
   public static int Main()
   {
      WriteGreeting(TimeOfDay.Morning);
      return 0;
   }
```

```
      static void WriteGreeting(TimeOfDay timeOfDay)
      {
         switch(timeOfDay)
         {
            case TimeOfDay.Morning:
               Console.WriteLine("Good morning!");
               break;
            case TimeOfDay.Afternoon:
               Console.WriteLine("Good afternoon!");
               break;
            case TimeOfDay.Evening:
               Console.WriteLine("Good evening!");
               break;
            default:
               Console.WriteLine("Hello!");
               break;
         }
      }
   }
```

在 C#中，枚举的真正强大之处是它们在后台会实例化为派生于基类 System.Enum 的结构。这表示可以对它们调用方法，执行有用的任务。注意因为.NET Framework 的执行方式，在语法上把枚举当做结构是不会有性能损失的。实际上，一旦代码编译好，枚举就成为基本类型，与 int 和 float 类似。

可以获取枚举的字符串表示，例如使用前面的 TimeOfDay 枚举：

```
   TimeOfDay time = TimeOfDay.Afternoon;
   Console.WriteLine(time.ToString());
```

会返回字符串 Afternoon。

另外，还可以从字符串中获取枚举值：

```
   TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
   Console.WriteLine((int)time2);
```

这段代码说明了如何从字符串获取枚举值，并转换为整数。要从字符串中转换，需要使用静态的 Enum.Parse()方法，这个方法带 3 个参数，第一个参数是要使用的枚举类型，其语法是关键字 typeof 后跟放在括号中的枚举类名。typeof 运算符将在第 6 章详细论述。第二个参数是要转换的字符串，第三个参数是一个 bool，指定在进行转换时是否忽略大小写。最后，注意 Enum.Parse()方法实际上返回一个对象引用——我们需要把这个字符串显式转换为需要的枚举类型(这是一个拆箱操作的例子)。对于上面的代码，将返回 1，作为一个对象，对应于 TimeOfDay. Afternoon 的枚举值。在显式转换为 int 时，会再次生成 1。

System.Enum 上的其他方法可以返回枚举定义中的值的个数、列出值的名称等。详细信息参见 MSDN 文档。

2.7 数组

本章不打算详细介绍数组，因为第 5 章将详细论述数组和集合。但本章将介绍编写一维数组的句法。在声明 C#中的数组时，要在各个元素的变量类型后面，加上一组方括号(注意数组中的所有元素必须有相同的数据类型)。

注意：

Visual Basic 用户注意，C#中的数组使用方括号，而不是圆括号。C++用户很熟悉方括号，但应仔细查看这里给出的代码，因为声明数组变量的 C#语法与 C++语法并不相同。

例如，int 表示一个整数，而 int[]表示一个整型数组：

```
int[] integers;
```

要初始化特定大小的数组，可以使用 new 关键字，在类型名后面的方括号中给出数组的大小：

```
// Create a new array of 32 ints
int[] integers = new int[32];
```

所有的数组都是引用类型，并遵循引用的语义。因此，即使各个元素都是基本的值类型，integers 数组也是引用类型。如果以后编写如下代码：

```
int[] copy = integers;
```

该代码也只是把变量 copy 指向同一个数组，而不是创建一个新数组。

要访问数组中的单个元素，可以使用通常的语法，在数组名的后面，把元素的下标放在方括号中。所有的 C#数组都使用基于 0 的下标方式，所以要用下标 0 引用第一个变量：

```
integers[0] = 35;
```

同样，用下标值 31 引用有 32 个元素的数组中的最后一个元素：

```
integers[31] = 432;
```

C#的数组语法也非常灵活，实际上，C#可以在声明数组时不进行初始化，这样以后就可以在程序中动态地指定其大小。利用这项技术，可以创建一个空引用，以后再使用 new 关键字把这个引用指向请求动态分配的内存位置：

```
int[] integers;
integers = new int[32];
```

可以使用下面的语法查看数组包含多少个元素：

```
int numElements = integers.Length;   // integers is any reference to an array.
```

2.8　命名空间

如前所述，命名空间提供了一种组织相关类和其他类型的方式。与文件或组件不同，命名空间是一种逻辑组合，而不是物理组合。在 C#文件中定义类时，可以把它包括在命名空间定义中。以后，在定义另一个类，在另一个文件中执行相关操作时，就可以在同一个命名空间中包含它，创建一个逻辑组合，告诉使用类的其他开发人员：这两个类是如何相关的以及如何使用它们：

```
namespace CustomerPhoneBookApp
{
   using System;

   public struct Subscriber
   {
      // Code for struct here...
   }
}
```

把一个类型放在命名空间中，可以有效地给这个类型指定一个较长的名称，该名称包括类型的命名空间，后面是句点(.)和类的名称。在上面的例子中，Subscriber 结构的全名是 CustomerPhoneBookApp.Subscriber。这样，有相同短名的不同的类就可以在同一个程序中使用了。

也可以在命名空间中嵌套其他命名空间，为类型创建层次结构：

```
namespace Wrox
{
   namespace ProCSharp
   {
      namespace Basics
      {
         class NamespaceExample
         {
            // Code for the class here...
         }
      }
   }
}
```

每个命名空间名都由它所在命名空间的名称组成，这些名称用句点分隔开，首先是最外层的命名空间，最后是它自己的短名。所以 ProCSharp 命名空间的全名是 Wrox.ProCSharp，NamespaceExample 类的全名是 Wrox.ProCSharp.Basics.NamespaceExample。

使用这个语法也可以组织自己的命名空间定义中的命名空间，所以上面的代码也可以写为：

```
namespace Wrox.ProCSharp.Basics
```

```
{
   class NamespaceExample
   {
      // Code for the class here...
   }
}
```

注意不允许在另一个嵌套的命名空间中声明多部分的命名空间。

命名空间与程序集无关。同一个程序集中可以有不同的命名空间，也可以在不同的程序集中定义同一个命名空间中的类型。

2.8.1 using 语句

显然，命名空间相当长，键入起来很繁琐，用这种方式指定某个类也是不必要的。如本章开头所述，C#允许简写类的全名。为此，要在文件的顶部列出类的命名空间，前面加上 using 关键字。在文件的其他地方，就可以使用其类型名称来引用命名空间中的类型了：

```
using System;
using Wrox.ProCSharp;
```

如前所述，所有的 C#源代码都以语句 using System;开头，这仅是因为 Microsoft 提供的许多有用的类都包含在 System 命名空间中。

如果 using 指令引用的两个命名空间包含同名的类，就必须使用完整的名称(或者至少较长的名称)，确保编译器知道访问哪个类型，例如，类 NamespaceExample 同时存在于 Wrox. ProCSharp.Basics 和 Wrox.ProCSharp.OOP 命名空间中，如果要在命名空间 Wrox.ProCSharp 中创建一个类 Test，并在该类中实例化一个 NamespaceExample 类，就需要指定使用哪个类：

```
using Wrox.ProCSharp;

class Test
{
   public static int Main()
   {
      Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
      //do something with the nSEx variable
      return 0;
   }
}
```

因为 using 语句在 C#文件的开头，C 和 C++也把#include 语句放在这里，所以从 C++迁移到 C#的程序员常把命名空间与 C++风格的头文件相混淆。不要犯这种错误，using 语句在这些文件之间并没有建立物理链接。C#也没有对应于 C++头文件的部分。

公司应花一定的时间开发一种命名空间模式，这样其开发人员才能快速定位他们需要的功能，而且公司内部使用的类名也不会与外部的类库相冲突。本章后面将介绍建立命名

空间模式的规则和其他命名约定。

2.8.2　命名空间的别名

using 关键字的另一个用途是给类和命名空间指定别名。如果命名空间的名称非常长，又要在代码中使用多次，但不希望该命名空间的名称包含在 using 指令中(例如，避免类名冲突)，就可以给该命名空间指定一个别名，其语法如下：

```
using alias = NamespaceName;
```

下面的例子(前面例子的修订版本)给 Wrox.ProCSharp.Basics 命名空间指定 Introduction 别名，并使用这个别名实例化了一个 NamespaceExample 对象，这个对象是在该命名空间中定义的。注意命名空间别名的修饰符是::。因此将先从 Introduction 命名空间别名开始搜索。如果在相同的作用域中引入了一个 Introduction 类，就会发生冲突。即使出现了冲突，:: 操作符也允许引用别名。NamespaceExample 类有一个方法 GetNamespace()，该方法调用每个类都有的 GetType()方法，以访问表示类的类型的 Type 对象。下面使用这个对象来返回类的命名空间名：

```
using System;
using Introduction = Wrox.ProCSharp.Basics;
class Test
{
   public static int Main()
   {
      Introduction::NamespaceExample NSEx =
         new Introduction::NamespaceExample();
      Console.WriteLine(NSEx.GetNamespace());
      return 0;
   }
}

namespace Wrox.ProCSharp.Basics
{
   class NamespaceExample
   {
      public string GetNamespace()
      {
         return this.GetType().Namespace;
      }
   }
}
```

2.9　Main()方法

本章的开头提到过，C#程序是从方法 Main()开始执行的。这个方法必须是类或结构的静态方法，并且其返回类型必须是 int 或 void。

虽然显式指定 public 修饰符是很常见的，因为按照定义，必须在程序外部调用该方法，但我们给该方法指定什么访问级别并不重要，即使把该方法标记为 private，它也可以运行。

2.9.1 多个 Main()方法

在编译C#控制台或 Windows 应用程序时，默认情况下，编译器会在与上述签名匹配的类中查找 Main 方法，并使这个类方法成为程序的入口。如果有多个 Main 方法，编译器就会返回一个错误消息，例如，考虑下面的代码 MainExample.cs：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class Client
   {
      public static int Main()
      {
         MathExample.Main();
         return 0;
      }
   }

   class MathExample
   {
      static int Add(int x, int y)
      {
         return x + y;
      }

      public static int Main()
      {
         int i = Add(5,10);
         Console.WriteLine(i);
         return 0;
      }
   }
}
```

上述代码中包含两个类，它们都有一个 Main()方法。如果按照通常的方式编译这段代码，就会得到下述错误：

```
csc MainExample.cs
    Microsoft (R) Visual C# .NET Compiler version 8.00.40607.16
    for Microsoft (R) Windows (R) .NET Framework version 2.00.40607
    Copyright (C) Microsoft Corporation 2001-2003. All rights reserved.

    MainExample.cs(7,23): error CS0017: Program 'MainExample.exe' has more than one entry point
    defined: 'Wrox.ProCSharp.Basics.Client.Main()'
```

```
MainExample.cs(21,23): error CS0017: Program 'MainExample.exe' has more than one entry point
defined: 'Wrox.ProCSharp.Basics.MathExample.Main()'
```

但是，可以使用/main 选项，其后跟 Main()方法所属类的全名(包括命名空间)，明确告诉编译器把哪个方法作为程序的入口点：

```
csc MainExample.cs /main:Wrox.ProCSharp.Basics.MathExample
```

2.9.2　给 Main()方法传送参数

前面的例子只介绍了不带参数的 Main()方法。但在调用程序时，可以让 CLR 包含一个参数，将命令行参数转送给程序。这个参数是一个字符串数组，传统称为args(但 C#可以接受任何名称)。在启动程序时，可以使用这个数组，访问通过命令行传送过来的选项。

下面的例子 ArgsExample.cs 是在传送给 Main 方法的字符串数组中迭代，并把每个选项的值写入控制台窗口：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class ArgsExample
   {
      public static int Main(string[] args)
      {
         for (int i = 0; i < args.Length; i++)
         {
            Console.WriteLine(args[i]);
         }
         return 0;
      }
   }
}
```

通常使用命令行就可以编译这段代码。在运行编译好的可执行文件时，可以在程序名的后面加上参数，例如：

```
ArgsExample /a /b /c
```

```
/a
/b
/c
```

2.10　有关编译 C#文件的更多内容

前面介绍了如何使用 csc.exe 编译控制台应用程序，但其他类型的应用程序如何编译？如果要引用一个类库，该怎么办？MSDN 文档介绍了 C#编译器的所有编译选项，这里只

介绍其中最重要的选项。

要回答第一个问题，应使用/target 选项(常简写为/t)来指定要创建的文件类型。文件类型可以是表 2-8 所示的类型中的一种。

表 2-8

选 项	输 出
/t:exe	控制台应用程序 (默认)
/t:library	带有清单的类库
/t:module	没有清单的组件
/t:winexe	Windows 应用程序 (没有控制台窗口)

如果想得到一个可由.NET 运行库加载的非可执行文件(例如 DLL)，就必须把它编译为一个库。如果把 C#文件编译为一个模块，就不会创建任何程序集。虽然模块不能由运行库加载，但可以使用/addmodule 选项编译到另一个清单中。

另一个需要注意的选项是/out，该选项可以指定由编译器生成的输出文件名。如果没有指定/out 选项，编译器就会使用输入的 C#文件名，加上目标类型的扩展名来建立输出文件名(例如.exe 表示 Windows 或控制台应用程序，.dll 表示类库)。注意/out 和/t(或/target)选项必须放在要编译的文件名前面。

默认状态下，如果在未引用的程序集中引用类型，可以使用/reference 或/r 选项，后跟程序集的路径和文件名。下面的例子说明了如何编译类库，并在另一个程序集中引用这个库。它包含两个文件：

- 类库
- 控制台应用程序，该应用程序调用库中的一个类。

第一个文件 MathLibrary.cs 包含 DLL 的代码，为了简单起见，它只包含一个公共类 MathLib 和一个方法，该方法把两个 int 类型的数据加在一起：

```
namespace Wrox.ProCSharp.Basics
{
   public class MathLib
   {
      public int Add(int x, int y)
      {
         return x + y;
      }
   }
}
```

使用下述命令把这个 C#文件编译为. NET DLL：

```
csc /t:library MathLibrary.cs
```

控制台应用程序 MathClient.cs 将简单地实例化这个对象，调用其 Add 方法，在控制台窗口中显示结果：

```
using System;

namespace Wrox.ProCSharp.Basics
{
   class Client
   {
      public static void Main()
      {
         MathLib mathObj = new MathLib();
         Console.WriteLine(mathObj.Add(7,8));
      }
   }
}
```

使用/r 选项编译这个文件，使之指向新编译的 DLL：

```
csc MathClient.cs /r:MathLibrary.dll
```

当然，下面就可以像往常一样运行它了：在命令提示符上输入 MathClient，其结果是显示数字 15—— 加运算的结果。

2.11　控制台 I/O

现在，读者应基本熟悉了 C#的数据类型以及如何控制线程如何执行操作这些数据类型的程序。本章还要使用 Console 类的几个静态方法来读写数据，这些方法在编写基本的 C#程序时非常有效，下面就详细介绍它们。

要从控制台窗口中读取一行文本，可以使用 Console.ReadLine()方法，它会从控制台窗口中读取一个输入流(在用户按下回车键时停止)，并返回输入的字符串。写入控制台也有两个对应的方法，前面已经使用过它们：

- Console. Write()方法将指定的值写入控制台窗口。
- Console.WriteLine()方法类似，但在输出结果的最后添加一个换行符。

所有预定义类型(包括 object) 都有这些函数的各种形式(重载)，所以在大多数情况下，在显示值之前不必把它们转换为字符串。

例如，下面的代码允许用户输入一行文本，并显示该文本：

```
string s = Console.ReadLine();

Console.WriteLine(s);
```

Console.WriteLine()还允许用与 C 的 printf()函数类似的方式显示格式化的结果。要以这种方式使用 WriteLine()，应传入许多参数。第一个参数是花括号中包含标记的字符串，在这个花括号中，要把后续的参数插入到文本中。每个标记都包含一个基于 0 的索引，表示列表中参数的序号。例如，{0}表示列表中的第一个参数，所以下面的代码：

```
int i = 10;
int j = 20;
```

```
Console.WriteLine("{0} plus {1} equals {2}", i, j, i + j);
```

会显示：

```
10 plus 20 equals 30
```

也可以为值指定宽度，调整文本在该宽度中的位置，正值表示右对齐，负值表示左对齐。为此可以使用格式{n,w}，其中 n 是参数索引，w 是宽度值。

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n----\n {2,4}", i, j, i + j);
```

结果如下：

```
 940
+ 73
----
1013
```

最后，还可以添加一个格式字符串，和一个可选的精度值。这里没有列出格式字符串的完整列表，因为如第 8 章所述，我们可以定义自己的格式字符串。但用于预定义类型的主要格式字符串如表 2-9 所示。

表 2-9

字 符 串	说 明
C	本地货币格式
D	十进制格式，把整数转换为以 10 为基数的数，如果给定一个精度说明符，就加上前导 0
E	科学计数法(指数)格式。精度说明符设置小数位数(默认为 6)。格式字符串的大小写("e" 或 "E")确定指数符号的大小写
F	固定点格式，精度说明符设置小数位数，可以为 0
G	普通格式，使用 E 或 F 格式取决于哪种格式较简单
N	数字格式，用逗号表示千分符，例如 32,767.44
P	百分数格式
X	十六进制格式，精度说明符用于加上前导 0

注意格式字符串都不需要考虑大小写，除 e/E 之外。

如果要使用格式字符串，应把它放在给出参数个数和字段宽度的标记后面，并用一个冒号把它们分隔开。例如，要把 decimal 值格式化为货币格式，且使用计算机上的地区设置，其精度为两位小数，则使用 C2：

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n ------\n {2,9:C2}", i, j, i + j);
```

在美国，其结果是：

```
    $940.23
+   $73.70
-------------
  $1,013.93
```

最后一个技巧是，可以使用占位符来代替这些格式字符串，例如：

```
double d = 0.234;
Console.WriteLine("{0:#.00}", d);
```

其结果为 0.23，因为如果在符号(#)的位置上没有字符，就会忽略该符号(#)，如果 0 的位置上有一个字符，就用这个字符代替 0，否则就显示 0。

2.12　使用注释

本节的内容表面上看起来很简单——给代码添加注释。

2.12.1　源文件中的内部注释

在本章开头提到过，C#使用传统的 C 风格注释方式：单行注释使用// …，多行注释使用/* … */：

```
// This is a single-line comment
/* This comment
  spans multiple lines */
```

单行注释中的任何内容，即//后面的内容都会被编译器忽略。多行注释中/* 和 */之间的所有内容也会被忽略。显然不能在多行注释中包含*/组合，因为这会被当作注释的结尾。

实际上，可以把多行注释放在一行代码中：

```
Console.WriteLine(/*Here's a comment! */ "This will compile");
```

像这样的内联注释在使用时应小心，因为它们会使代码难以理解。但这样的注释在调试时是非常有用的，例如，在运行代码时要临时使用另一个值：

```
DoSomething(Width, /*Height*/ 100);
```

当然，字符串字面值中的注释字符会按照一般的字符来处理：

```
string s = "/* This is just a normal string */";
```

2.12.2　XML 文档说明

如前所述，除了 C 风格的注释外，C#还有一个非常好的功能，本章将讨论这一功能。根据特定的注释自动创建 XML 格式的文档说明。这些注释都是单行注释，但都以 3 个斜杠(///)开头，而不是通常的两个斜杠。在这些注释中，可以把包含类型和类型成员的文档说明的 XML 标识符放在代码中。

编译器可以识别表 2-10 所示的标识符。

表 2-10

标 识 符	说 明
<c>	把行中的文本标记为代码，例如<c>int i = 10;</c>
<code>	把多行标记为代码
<example>	标记为一个代码示例
<exception>	说明一个异常类(编译器要验证其语法)
<include>	包含其他文档说明文件的注释(编译器要验证其语法)
<list>	把列表插入到文档说明中
<param>	标记方法的参数(编译器要验证其语法)
<paramref>	表示一个单词是方法的参数(编译器要验证其语法)
<permission>	说明对成员的访问(编译器要验证其语法)
<remarks>	给成员添加描述
<returns>	说明方法的返回值
<see>	提供对另一个参数的交叉引用(编译器要验证其语法)
<seealso>	提供描述中的“参见”部分(编译器要验证其语法)
<summary>	提供类型或成员的简短小结
<value>	描述属性

要了解它们的工作方式，可以在上一节的 MathLibrary.cs 文件中添加一些 XML 注释，并称之为 Math.cs。我们给类及其 Add 方法添加一个<summary>元素，也给 Add 方法添加一个<returns>元素和两个<param>元素：

```
// Math.cs
namespace Wrox.ProCSharp.Basics
{

   ///<summary>
   ///   Wrox.ProCSharp.Basics.Math class.
   ///   Provides a method to add two integers.
   ///</summary>
   public class Math
   {
      ///<summary>
      ///   The Add method allows us to add two integers
      ///</summary>
      ///<returns>Result of the addition (int)</returns>
      ///<param name="x">First number to add</param>
      ///<param name="y">Second number to add</param>
      public int Add(int x, int y)
      {
         return x + y;
      }
```

```
    }
}
```

C#编译器可以把 XML 元素从特定的注释中提取出来，并使用它们生成一个 XML 文件。要让编译器为程序集生成 XML 文档说明，需在编译时指定/doc 选项，后跟要创建的文件名：

```
csc /t:library /doc:Math.xml Math.cs
```

如果 XML 注释没有生成格式正确的 XML 文档，编译器就生成一个错误。

上面的代码会生成一个 XML 文件 Math.xml，如下所示。

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>Math</name>
   </assembly>
   <members>
      <member name="T:Wrox.ProCSharp.Basics.Math">
         <summary>
            Wrox.ProCSharp.Basics.Math class.
            Provides a method to add two integers.
         </summary>
      </member>
      <member name=
            "M:Wrox.ProCSharp.Basics.Math.Add(System.Int32,System.Int32)">
         <summary>
            The Add method allows us to add two integers.
         </summary>
         <returns>Result of the addition (int)</returns>
         <param name="x">First number to add</param>
         <param name="y">Second number to add</param>
      </member>
   </members>
</doc>
```

注意，编译器为我们做了一些工作——它创建了一个<assembly>元素，并为该文件中的每个类型或类型成员添加一个<member>元素。每个<member>元素都有一个 name 特性，其中包含成员的全名，前面有一个字母表示其类型："T:"表示这是一个类型，"F:" 表示这是一个字段，"M:" 表示这是一个成员。

2.13　C#预处理器指令

除了前面介绍的常用关键字外，C#还有许多名为“预处理器指令”的命令。这些命令从来不会转化为可执行代码中的命令，但会影响编译过程的各个方面。例如，使用预处理器指令可以禁止编译器编译代码的某一部分。如果计划发布两个版本的代码，即基本版本和有更多功能的企业版本，就可以使用这些预处理器指令。在编译软件的基本版本时，使

用预处理器指令还可以禁止编译器编译与额外功能相关的代码。另外，在编写提供调试信息的代码时，也可以使用预处理器指令。实际上，在销售软件时，一般不希望编译这部分代码。

预处理器指令的开头都有符号#。

注意：

C++开发人员应知道在 C 和 C++中，预处理器指令是非常重要的，但是，在 C#中，并没有那么多的预处理器指令，它们的使用也不太频繁。C#提供了其他机制来实现许多 C++指令的功能，例如定制特性。还要注意，C#并没有一个像 C++那样的独立预处理器，所谓的预处理器指令实际上是由编译器处理的。尽管如此，C#仍保留了一些预处理器指令，因为这些命令对预处理器有一定的影响。

下面简要介绍预处理器指令的功能。

2.13.1 #define 和 #undef

#define 的用法如下所示：

```
#define DEBUG
```

它告诉编译器存在给定名称的符号，在本例中是 DEBUG。这有点类似于声明一个变量，但这个变量并没有真正的值，只是存在而已。这个符号不是实际代码的一部分，而只在编译器编译代码时存在。在 C#代码中它没有任何意义。

#undef 正好相反——删除符号的定义：

```
#undef DEBUG
```

如果符号不存在，#undef 就没有任何作用。同样，如果符号已经存在，#define 也不起作用。

必须把#define 和#undef 命令放在 C#源代码的开头，在声明要编译的任何对象的代码之前。

#define 本身并没有什么用，但与其他预处理器指令(特别是#if)结合使用时，它的功能就非常强大了。

注意：

这里应注意一般 C#语法的一些变化。预处理器指令不用分号结束，一般一行上只有一个命令。这是因为对于预处理器指令，C#不再要求命令用分号结束。如果它遇到一个预处理器指令，就会假定下一个命令在下一行上。

2.13.2 #if, #elif, #else 和#endif

这些指令告诉编译器是否要编译某个代码块。考虑下面的方法：

```
int DoSomeWork(double x)
```

```
    {
      // do something
      #if DEBUG
        Console.WriteLine("x is " + x);
      #endif
    }
```

这段代码会像往常那样编译，但 Console.WriteLine 命令包含在#if 子句内。这行代码只有在前面的#define 命令定义了符号 DEBUG 后才执行。当编译器遇到#if 语句后，将先检查相关的符号是否存在，如果符号存在，就编译#if 块中的代码。否则，编译器会忽略所有的代码，直到遇到匹配的#endif 指令为止。一般是在调试时定义符号 DEBUG，把与调试相关的代码放在#if 子句中。在完成了调试后，就把#define 语句注释掉，所有的调试代码会奇迹般地消失，可执行文件也会变小，最终用户不会被这些调试信息弄糊涂(显然，要做更多的测试，确保代码在没有定义 DEBUG 的情况下也能工作)。这项技术在 C 和 C++编程中非常普通，称为条件编译(conditional compilation)。

#elif (=else if)和#else 指令可以用在#if 块中，其含义非常直观。也可以嵌套#if 块：

```
#define ENTERPRISE
#define W2K

// further on in the file

#if  ENTERPRISE
  // do something
  #if W2K
    // some code that is only relevant to enterprise
    // edition running on W2K
  #endif
#elif PROFESSIONAL
  // do something else
#else
  // code for the leaner version
#endif
```

注意：

与 C++中的情况不同，使用#if 不是条件编译代码的唯一方式，C#还通过 Conditional 特性提供了另一种机制，详见第 12 章。

#if 和 #elif 还支持一组逻辑运算符!、==、!=和 ||。如果符号存在，就被认为是 true，否则为 false，例如：

```
#if W2K && (ENTERPRISE==false)   // if W2K is defined but ENTERPRISE isn't
```

2.13.3　#warning 和 #error

另外两个非常有用的预处理器指令是#warning 和#error，当编译器遇到它们时，会分别产生警告或错误。如果编译器遇到#warning 指令，会给用户显示#warning 指令后面的文本，

之后编译继续进行。如果编译器遇到#error 指令，就会给用户显示后面的文本，作为一个编译错误信息，然后会立即退出编译，不会生成 IL 代码。

使用这两个指令可以检查#define 语句是不是做错了什么事，使用#warning 语句可以让自己想起做过什么事：

```
#if DEBUG && RELEASE
  #error "You've defined DEBUG and RELEASE simultaneously! "
#endif

#warning "Don't forget to remove this line before the boss tests the code! "
  Console.WriteLine("*I hate this job*");
```

2.13.4 #region 和#endregion

#region 和 #endregion 指令用于把一段代码标记为有给定名称的一个块，如下所示。

```
#region Member Field Declarations
  int x;
  double d;
  Currency balance;
#endregion
```

这看起来似乎没有什么用，它不影响编译过程。这些指令的优点是它们可以被某些编辑器识别，包括 Visual Studio .NET 编辑器。这些编辑器可以使用这些指令使代码在屏幕上更好地布局。第 14 章会详细介绍它们。

2.13.5 #line

#line 指令可以用于改变编译器在警告和错误信息中显示的文件名和行号信息。这个指令用得并不多。如果编写代码时，在把代码发送给编译器前，要使用某些软件包改变键入的代码，就可以使用这个指令，因为这意味着编译器报告的行号或文件名与文件中的行号或编辑的文件名不匹配。#line 指令可以用于恢复这种匹配。也可以使用语法#line default 把行号恢复为默认的行号：

```
#line 164 "Core.cs"   // we happen to know this is line 164 in the file
                      // Core.cs, before the intermediate
                      // package mangles it.

// later on

#line default     // restores default line numbering
```

2.13.6 #pragma

#pragma 指令可以抑制或恢复指定的编译警告。与命令行选项不同，#pragma 指令可以在类或方法上执行，对抑制警告的内容和抑制的时间进行更精细的控制。下面的例子禁止

字段使用警告，然后在编译 MyClass 类后恢复该警告。

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

2.14　C#编程规则

本节介绍编写 C#程序时应注意的规则。

2.14.1　用于标识符的规则

本节将讨论变量、类、方法等的命名规则。注意本节所介绍的规则不仅是规则，也是C#编译器强制使用的。

标识符是给变量、用户定义的类型(例如类和结构)和这些类型的成员指定的名称。标识符区分大小写，所以 interestRate 和 InterestRate 是不同的变量。确定在 C#中可以使用什么标识符有两个规则：

- 它们必须以一个字母或下划线开头，但可以包含数字字符；
- 不能把 C#关键字用作标识符。

C#包含如表 2-11 所示的保留关键字。

表　2-11

abstract	do	In	protected	true
as	double	Int	public	try
base	else	Interface	readonly	typeof
bool	enum	Internal	ref	uint
break	event	Is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	volatile
const	for	operator	string	void
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	Implicit	private	throw	

如果需要把某一保留字用作标识符(例如，访问一个用另一种语言编写的类)，可以在标识符的前面加上前缀@符号，指示编译器其后的内容是一个标识符，而不是 C#关键字(所以 abstract 不是有效的标识符，而@abstract 是)。

最后，标识符也可以包含 Unicode 字符，用语法\uXXXX 来指定，其中 XXXX 是 Unicode 字符的四位十六进制代码。下面是有效标识符的一些例子：

- Name
- überfluß
- _Identifier
- \u005fIdentifier

最后两个标识符是相同的，可以互换(005f 是下划线字符的 Unicode 代码)，所以这些标识符在同一个作用域内不要声明两次。注意虽然从语法上看，标识符中可以使用下划线字符，但在大多数情况下，最好不要这么做，因为它不符合 Microsoft 的变量命名规则，这种命名规则可以确保开发人员使用相同的命名规则，易于阅读每个人编写的代码。

2.14.2 用法约定

在任何开发环境中，通常有一些传统的编程风格。这些风格不是语言的一部分，而是约定，例如，变量如何命名，类、方法或函数如何使用等。如果使用某语言的大多数开发人员都遵循相同的约定，不同的开发人员就很容易理解彼此的代码，这一般有助于程序的维护。例如，Visual Basic 6 的一个公共(但不统一)约定是，表示字符串的变量名以小写字母 s 或 str 开头，如 Dim sResult As String 或 Dim strMessage As String。约定主要取决于语言和环境。例如，在 Windows 平台上编程的 C++开发人员一般使用前缀 psz 或 lpsz 表示字符串：char *pszResult; char *lpszMessage;，但在 UNIX 机器上，则不使用任何前缀：char *Result; char *Message;。

从本书中的示例代码中可以总结出，C#中的约定是命名变量时不使用任何前缀：string Result; string Message;。

注意：

用带有前缀字母的变量名来表示某个数据类型，这种约定称为 Hungarian 表示法。这样，其他阅读该代码的开发人员就可以立即从变量名中了解它代表什么数据类型。在有了智能编辑器和 IntelliSense 之后，人们普遍认为 Hungarian 表示法是多余的。

但是，在许多语言中，用法约定是从语言的使用过程中逐渐演变而来的，Microsoft 编写的 C#和整个.NET Framework 都有非常多的用法约定，详见.NET/C# MSDN 文档说明。这说明，从一开始，.NET 程序就有非常高的互操作性，开发人员可以以此来理解代码。用法规则还得益于 20 年来面向对象编程的发展，因此相关的新闻组已经仔细考虑了这些用法规则，而且已经为开发团体所接受。所以我们应遵守这些约定。

但要注意，这些规则与语言规范是不同的。用户应尽可能遵循这些规则。但如果有很好的理由不遵循它们，也不会有什么问题。例如，不遵循这些用法约定，也不会出现编译错误。一般情况下，如果不遵循用法规则，就必须有一个说得过去的理由。规则应是一个

正确的决策，而不是让人头痛的东西。在阅读本书的后续内容时，应注意到在本书的许多示例中，都没有遵循该约定，这通常是因为某些规则适用于大型程序，而不适合于本书中的小示例。如果编写一个完整的软件包，就应遵循这些规则，但它们并不适合于只有 20 行代码的独立程序。在许多情况下，遵循约定会使这些示例难以理解。

编程风格的规则非常多。这里只介绍一些比较重要的规则，以及最适合于用户的规则。如果用户要让代码完全遵循用法规则，就需要参考 MSDN 文档说明。

1. 命名约定

使程序易于理解的一个重要方面是给对象选择命名的方式，包括变量名、方法名、类名、枚举名和命名空间的名称。

显然，这些名称应反映对象的功能，且不与其他名称冲突。在.NET Framework 中，一般规则也是变量名要反映变量实例的功能，而不是反映数据类型。例如，Height 就是一个比较好的变量名，而 IntegerValue 就不太好。但是，这种规则是一种理想状态，很难达到。在处理控件时，大多数情况下使用 ConfirmationDialog 和 ChooseEmployeeListBox 等变量名比较好，这些变量名说明了变量的数据类型。

名称的约定包括以下几个方面：

(1) 名称的大小写

在许多情况下，名称都应使用 Pascal 大小写命名形式。 Pascal 大小写形式是指名称中单词的第一个字母大写，如 EmployeeSalary、ConfirmationDialog、PlainTextEncoding。注意，命名空间、类、以及基类中的成员等的名称都应遵循该规则，最好不要使用带有下划线字符的单词，即名称不应是 employee_salary。其他语言中常量的名称常常全部都是大写，但在 C#中最好不要这样，因为这种名称很难阅读，而应全部使用 Pascal 大小写形式的命名约定：

```
const int MaximumLength;
```

我们还推荐使用另一种大小写模式：camel 大小写形式。这种形式类似于 Pascal 大小写形式，但名称中第一个单词的第一个字母不是大写：employeeSalary、confirmationDialog、plainTextEncoding。有三种情况可以使用 camel 大小写形式。

- 类型中所有私有成员字段的名称都应是 camel 大小写形式：

```
public int subscriberId;
```

但要注意成员字段名常常用一个下划线开头：

```
public int _subscriberId;
```

- 传递给方法的所有参数都应是 camel 大小写形式：

```
public void RecordSale(string salesmanName, int quantity);
```

- camel 大小写形式也可以用于区分同名的两个对象——比较常见的情况是属性封装一个字段：

```
    private string employeeName;

  public string EmployeeName
  {
    get
      {
        return employeeName;
      }
    }
```

如果这么做，则私有成员总是使用 camel 大小写形式，而公共的或受保护的成员总是使用 Pascal 大小写形式，这样使用这段代码的其他类就只能使用 Pascal 大小写形式的名称了(除了参数名以外)。

还要注意大小写问题。C#是区分大小写的，所以在 C#中，仅大小写不同的名称在语法上是正确的，如上面的例子。但是，程序集可能在 Visual Basic .NET 应用程序中调用，而 Visual Basic .NET 是不区分大小写的，如果使用仅大小写不同的名称，就必须使这两个名称不能在程序集的外部访问(上例是可行的，因为仅私有变量使用了 camel 大小写形式的名称)。否则，Visual Basic .NET 中的其他代码就不能正确使用这个程序集。

(2) 名称的风格

名称的风格应保持一致。例如，如果类中的一个方法叫 ShowConfirmationDialog()，另一个方法就不能叫 ShowDialogWarning()或 WarningDialogShow()，而应是 ShowWarningDialog()。

(3) 命名空间的名称

命名空间的名称非常重要，一定要仔细设计，以避免一个命名空间中对象的名称与其他对象同名。记住，命名空间的名称是.NET 区分共享程序集中对象名的唯一方式。如果软件包的命名空间使用的名称与另一个软件包相同，而这两个软件包都安装在一台计算机上，就会出问题。因此，最好用自己的公司名创建顶级的命名空间，再嵌套技术范围较窄、用户所在小组或部门、或类所在软件包的命名空间。Microsoft 建议使用如下的命名空间：<CompanyName>. <TechnologyName>，例如：

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

(4) 名称和关键字

名称不应与任何关键字冲突，这是非常重要的。实际上，如果在代码中，试图给某个对象指定与 C#关键字同名的名称，就会出现语法错误，因为编译器会假定该名称表示一个语句。但是，由于类可能由其他语言编写的代码访问，所以不能使用其他.NET 语言中的关键字作为对象的名称。一般说来，C++关键字类似于 C#关键字，不太可能与 C++混淆，Visual C++常用的关键字则用两个下划线字符开头。与 C#一样，C++关键字都是小写字母，如果要遵循公共类和成员使用 Pascal 风格的名称的约定，则在它们的名称中至少有一个字母是大写，因此不会与 C++关键字冲突。另一方面，Visual Basic 的问题会多一些，因为 Visual Basic 的关键字要比 C#的多，而且它不区分大小写，不能依赖于 Pascal 风格的名称来区分

类和成员。

表 2-12 列出了 Visual Basic 中的关键字和标准函数调用，无论对 C#公共类使用什么大小写组合，这些名称都不应使用。

表　2-12

Abs	Add	AddHandler	AddressOf	Alias
And	Ansi	AppActivate	Append	As
Asc	Assembly	Atan	Auto	Beep
Binary	BitAnd	BitNot	BitOr	BitXor
Boolean	ByRef	Byte	ByVal	Call
Case	Catch	CBool	CByte	CDate
CDbl	CDec	ChDir	ChDrive	Choose
Chr	CInt	Class	Clear	CLng
Close	Collection	Command	Compare	Const
Cos	CreateObject	CShort	CSng	Do
Double	Each	Else	ElseIf	Empty
End	Enum	EOF	Erase	Err
Error	Event	Exit	Exp	Explicit
ExternalSource	False	FileAttr	FileCopy	FileDateTime
FileLen	Filter	Finally	Fix	For
Format	FreeFile	Friend	Function	FV
Get	GetAllSettings	GetAttr	GetException	GetObject
GetSetting	GetType	GoTo	Handles	Hex
Hour	If	Iif	Implements	Imports
In	Inherits	Input	Loc	Local
Lock	LOF	Log	Long	Loop
LTrim	Me	Mid	Minute	MIRR
MkDir	Module	Month	MustInherit	MustOverride
MyBase	MyClass	Namespace	New	Next
Not	Nothing	NotInheritable	NotOverridable	Now
NPer	NPV	Null	Object	Oct
Off	On	Open	Option	Optional
Or	Overloads	Overridable	Overrides	ParamArray
Pmt	PPmt	Preserve	Print	Private
Property	Public	RGB	Right	RmDir
Rnd	RTrim	SaveSettings	Second	Seek

(续表)

Select	SetAttr	SetException	Shared	Shell
Short	Sign	Sin	Single	SLN
Space	Spc	Split	Sqrt	Static
Step	Stop	Str	StrComp	StrConv
Strict	String	Structure	Sub	Switch
SYD	SyncLock	Tab	Tan	Text
Then	Throw	TimeOfDay	Timer	TimeSerial
TimeValue	To	Today	Trim	Try
TypeName				

2. 属性和方法的使用

类中出现混乱的一个方面是一个数是用属性还是方法来表示。这没有硬性规定，但一般情况下，如果该对象的外观和操作都像一个变量，就应使用属性来表示它(属性详见第 3 章)，即：

- 客户机代码应能读取它的值，最好不要使用只写属性，例如，应使用 SetPassword() 方法，而不是 Password 只写属性。
- 读取该值不应花太长的时间。实际上，如果它是一个属性，通常表示读取过程花的时间相对较短。
- 读取该值不应有任何不希望的负面效应。设置属性的值，不应有与该属性不直接相关的负面效应。设置对话框的宽度会改变该对话框在屏幕上的外观，这是可以的，因为它与属性是相关的。
- 应可以用任何顺序设置属性。在设置属性时，最好不要因为还没有设置另一个相关的属性而抛出一个异常。例如，如果为了使用访问数据库的类，需要设置 ConnectionString、UserName 和 Password，应确保已经执行了该类，这样用户才能按照任何顺序设置它们。
- 顺序读取属性也应有相同的效果。如果属性的值可能会出现预料不到的改变，就应把它编写为一个方法。在监视汽车运动的类中，把 speed 编写为属性就不是一种好的方式，而应使用 GetSpeed()，另一方面，应把 Weight 和 EngineSize 编写为属性，因为对于给定的对象，它们是不会改变的。

如果要编码的对象满足上述所有条件，就应对它使用属性，否则就应使用方法。

3. 字段的用法

字段的用法非常简单。字段应总是私有的，但在某些情况下也可以把常量或只读字段设置为公有，原因是如果把字段设置为公有，就可以在以后扩展或修改类。

遵循上面的规则就可以编写出好的代码，而且这些规则应与面向对编程的风格一起使用。

Microsoft 在保持一致性方面相当谨慎，在编写.NET 基类时遵循了它自己的规则。在编写.NET 代码时应很好地遵循这些规则，对于基类来说，就是类、成员、命名空间的命名

方式和类层次结构的工作方式等，如果编写代码的风格与基类的编写风格相同，就不会犯什么错误。

2.15　小结

本章介绍了一些 C#基本语法，包括编写简单的 C#程序需要掌握的内容。我们讲述了许多基础知识，但其中有许多是熟悉 C 风格语言(甚或 JavaScript)的开发人员能立即领悟的。本章的主要内容包括：

- 变量的作用域和访问级别
- 声明各种数据类型的变量
- 在 C#程序中控制执行流
- 注释和 XML 文档说明
- 预处理器指令
- 用法规则和命名约定，在编写 C#代码时应遵循这些规则，使代码符合一般的.NET 规范，这样其他人就很容易理解您编写的代码了。

C#语法与 C++/Java 语法非常类似，但仍存在一些小区别。在许多领域，将这些语法与功能结合起来，会使编码更快速，例如高质量的字符串处理功能。C#还有一个强大的已定义类型系统，该系统基于值类型和引用类型的区别。下面两章将进一步介绍 C#的面向对象编程特性。

第 3 章

对象和类型

到目前为止，我们介绍了组成 C#语言的主要内容，包括变量的声明、数据类型和程序流语句，并简要介绍了一个只包含 Main()方法的完整小例子。但还没有介绍如何把这些内容组合在一起，构成一个完整的程序，其关键就在于对类的处理。这就是本章的主题。本章的主要内容如下：

- 类和结构的区别
- 字段、属性和方法
- 按值和引用传送参数
- 方法重载
- 构造函数和静态构造函数
- 只读字段
- Object 类，其他类型都从该类派生而来

第 4 章将介绍继承以及与继承相关的特性。

提示：

本章将讨论与类相关的基本语法，但假定您已经熟悉了使用类的基本原则，例如，知道构造函数和属性的含义，因此我们只是大致论述如何把这些原则应用于 C#代码。

本章介绍的这些概念不一定得到了大多数面向对象语言的支持。例如对象构造函数是您熟悉的、使用广泛的一个概念，但静态构造函数就是 C#的新增内容，所以我们将解释静态构造函数的工作原理。

3.1 类和结构

类和结构实际上都是创建对象的模板，每个对象都包含数据，并提供了处理和访问数据的方法。类定义了每个类对象(称为实例)可以包含什么数据和功能。例如，如果一个类表示一个顾客，就可以定义字段 CustomerID、FirstName、LastName 和 Address，以包含该

顾客的信息。还可以定义处理存储在这些字段中的数据的功能。接着，就可以实例化这个类的对象，以表示某个顾客，并为这个实例设置这些字段，使用其功能。

```
class PhoneCustomer
{
   public const string DayOfSendingBill ="Monday";
 public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

结构与类的区别是它们在内存中的存储方式(类是存储在堆(heap)上的引用类型，而结构是存储在堆栈(stack)上的值类型)、访问方式和一些特征(如结构不支持继承)。较小的数据类型使用结构可提高性能。但在语法上，结构与类非常相似，主要的区别是使用关键字 struct 代替 class 来声明结构。例如，如果希望所有的 PhoneCustomer 实例都存储在堆栈上，而不是存储在托管堆上，就可以编写下面的语句：

```
struct PhoneCustomerStruct
{
   public const string DayOfSendingBill = "Monday";
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

对于类和结构，都使用关键字 new 来声明实例：这个关键字创建对象并对其进行初始化。在下面的例子中，类和结构的字段值都默认为 0：

```
PhoneCustomer myCustomer = new PhoneCustomer();    //works for a class
PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct(); // works for a struct
```

在大多数情况下，类要比结构常用得多。因此，我们先讨论类，然后指出类和结构的区别，以及选择使用结构而不使用类的特殊原因。但除非特别说明，否则就可以假定用于类的代码也适用于结构。

3.2 类成员

类中的数据和函数称为类的成员。Microsoft 的正式术语对数据成员和函数成员进行了区分。除了这些成员外，类还可以包含嵌套的类型(例如其他类)。类中的所有成员都可以声明为 public(此时可以在类的外部直接访问它们)或 private(此时，它们只能由类中的其他代码来访问)。与 Visual Basic、C++和 Java 一样，C#在这个方面还有变化，例如 protected(表示成员仅能由该成员所在的类及其派生类访问)，第 4 章将详细解释各种访问级别。

3.2.1　数据成员

数据成员包含了类的数据—— 字段、常量和事件。数据成员可以是静态数据(与整个类相关)或实例数据(类的每个实例都有它自己的数据副本)。通常，对于面向对象的语言，类成员总是实例成员，除非用 static 进行了显式的声明。

字段是与类相关的变量。在前面的例子中已经使用了 PhoneCustomer 类中的字段：

一旦实例化 PhoneCustomer 对象，就可以使用语法 Object.FieldName 来访问这些字段：

```
PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = "Simon";
```

常量与类的关联方式同变量与类的关联方式一样。使用 const 关键字来声明常量。如果它们声明为 public，就可以在类的外部访问。

```
class PhoneCustomer
{
   public const string DayOfSendingBill = "Monday";
   public int CustomerID;
   public string FirstName;
   public string LastName;
}
```

事件是类的成员，在发生某些行为(例如改变类的字段或属性，或者进行了某种形式的用户交互操作)时，它可以让对象通知调用程序。客户可以包含所谓“事件处理程序”的代码来响应该事件。第 7 章将详细介绍事件。

3.2.2　函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(finalizer)、运算符以及索引器。

方法是与某个类相关的函数，它们可以是实例方法，也可以是静态方法。实例方法处理类的某个实例，静态方法提供了更一般的功能，不需要实例化一个类(例如 Console.WriteLine()方法)。下一节介绍方法。

属性是可以在客户机上访问的函数组，其访问方式与访问类的公共字段类似。C#为读写类上的属性提供了专用语法，所以不必使用那些名称中嵌有 Get 或 Set 的偷工减料的方法。因为属性的这种语法不同于一般函数的语法，在客户代码中，虚拟的对象被当做实际的东西。

构造函数是在实例化对象时自动调用的函数。它们必须与所属的类同名，且不能有返回类型。构造函数用于初始化字段的值。

终结器类似于构造函数，但是在 CLR 检测到不再需要某个对象时调用。它们的名称与类相同，但前面有一个~符号。C++程序员应注意，终结器在 C#中比在 C++中用得少得多，因为 CLR 会自动进行垃圾收集，另外，不可能预测什么时候调用终结器。第 7 章将介绍终结器。

运算符执行的最简单的操作就是+和－。在对两个整数进行相加操作时，严格地说，就是对整数使用+运算符。C#还允许指定把已有的运算符应用于自己的类(运算符重载)。第 5 章将详细论述运算符。

索引器允许对象以数组或集合的方式进行索引。第 5 章介绍索引器。

1. 方法

在 Visual Basic、C 和 C++中，可以定义与类完全不相关的全局函数，但在 C#中不能这样做。在 C#中，每个函数都必须与类或结构相关。

注意，正式的 C#术语实际上区分了函数和方法。在这个术语中，“函数成员”不仅包含方法，而且也包含类或结构的一些非数据成员，例如索引器、运算符、构造函数和析构函数等，甚至还有属性。这些都不是数据成员，字段、常量和事件才是数据成员。本章将详细讨论方法。

(1) 方法的声明

在 C#中，定义方法的语法与 C 风格的语言相同，与 C++和 Java 中的语法也相同。与 C++的主要语法区别是，在 C#中，每个方法都单独声明为 public 或 private，不能使用 public：块把几个方法定义组合起来。另外，所有的 C#方法都在类定义中声明和定义。在 C#中，不能像在 C++中那样把方法的实现代码分隔开来。

在 C#中，方法的定义包括方法的修饰符(例如方法的可访问性)、返回值的类型，然后是方法名、输入参数的列表(用圆括号括起来)和方法体(用花括号括起来)。

```
[modifiers] return_type MethodName([parameters])
{
   // Method body
}
```

每个参数都包括参数的类型名及在方法体中的引用名称。但如果方法有返回值，return 语句就必须与返回值一起使用，以指定出口点，例如：

```
public bool IsSquare(Rectangle rect)
{
   return (rect.Height == rect.Width);
}
```

这段代码使用了一个表示矩形的.NET 基类 System.Drawing.Rectangle。

如果方法没有返回值，就把返回类型指定为 void，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面写上一对空的圆括号()(就像本章前面的 Main()方法)。此时 return 语句就是可选的——当到达右花括号时，方法会自动返回。注意方法可以包含任意多个 return 语句：

```
public bool IsPositive(int value)
{
   if (value < 0)
```

```
        return false;
    return true;
}
```

(2) 调用方法

C#中调用方法的语法与 C++和 Java 中的一样，C#和 Visual Basic 的唯一区别是在 C#中调用方法时，必须使用圆括号，这要比 Visual Basic 6 中有时需要括号，有时不需要括号的规则简单一些。

下面的例子 MathTest 说明了类的定义和实例化、方法的定义和调用的语法。除了包含 Main()方法的类之外，它还定义了类 MathTest，该类包含两个方法和一个字段。

```
using System;

namespace Wrox.ProCSharp. MathTestSample
{
   class MainEntryPoint
   {
      static void Main()
      {
         // Try calling some static functions
         Console.WriteLine("Pi is " + MathTest.GetPi());
         int x = MathTest.GetSquareOf(5);
         Console.WriteLine("Square of 5 is " + x);

         // Instantiate at MathTest object
         MathTest math = new MathTest(); // this is C#'s way of
                                         // instantiating a reference type

         // Call non-static methods
         math.value = 30;
         Console.WriteLine(
            "Value field of math variable contains " + math.value);
         Console.WriteLine("Square of 30 is " + math.GetSquare());
      }
   }

   // Define a class named MathTest on which we will call a method
   class MathTest
   {
      public int value;

      public int GetSquare()
      {
         return value*value;
      }

      public static int GetSquareOf(int x)
      {
         return x*x;
```

```
        }

        public static double GetPi()
        {
           return 3.14159;
        }
     }
  }
```

运行 mathTest 示例，会得到如下结果：

csc MathTest.cs

```
Microsoft (R) Visual C# .NET Compiler version 8.00.40607.16
for Microsoft (R) .NET Framework version 2.0.40607
Copyright (C) Microsoft Corporation 2001-2003. All rights reserved.

MathTest.exe
Pi is 3.14159
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900
```

从代码中可以看出，MathTest 类包含一个字段和一个方法，该字段包含一个数字，该方法计算数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算把作为参数传入的数字的平方。

这个类有一些功能并不是 C#程序设计的好例子。例如，GetPi()通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

C++和 Java 开发人员应很熟悉这个例子的大多数语法。如果您有 Visual Basic 的编程经验，只需把 MathTest 类看作一个执行字段和方法的 Visual Basic 类模块。但无论使用什么语言，都要注意两个要点。

(3) 给方法传递参数

参数可以通过引用或值传递给方法。在变量通过引用传递给方法时，被调用的方法得到的就是这个变量，所以在方法内部对变量进行的任何改变在方法退出后仍旧发挥作用。而如果变量是通过值传送给方法的，被调用的方法得到的是变量的一个副本，也就是说，在方法退出后，对变量进行的修改会丢失。对于复杂的数据类型，按引用传递的效率更高，因为在按值传递时，必须复制大量的数据。

在 C#中，所有的参数都是通过值来传递的，除非特别说明。这与 C++是相同的，但与 Visual Basic 相反。但是，在理解引用类型的传递过程时需要注意。因为引用类型的对象只包含对象的引用，它们只给方法传递这个引用，而不是对象本身，所以对底层对象的修改会保留下来。相反，值类型的对象包含的是实际数据，所以传递给方法的是数据本身的副本。例如，int 通过值传递给方法，方法对该 int 的值所作的任何改变都没有改变原 int 对象的值。但如果数组或其他引用类型(如类)传递给方法，方法就会使用该引用改变这个数组中的值，而新值会反射到原来的数组对象上。

下面的例子 ParameterTest.cs 说明了这一点：

```
using System;

namespace Wrox.ProCSharp. ParameterTestSample
{
   class ParameterTest
   {
      static void SomeFunction(int[] ints, int i)
      {
         ints[0] = 100;
         i = 100;
      }

      public static int Main()
      {
         int i = 0;
         int[] ints = { 0, 1, 2, 4, 8 };
         // Display the original values
         Console.WriteLine("i = " + i);
         Console.WriteLine("ints[0] = " + ints[0]);
         Console.WriteLine("Calling SomeFunction...");

         // After this method returns, ints will be changed,
         // but i will not
         SomeFunction(ints, i);
         Console.WriteLine("i = " + i);
         Console.WriteLine("ints[0] = " + ints[0]);
         return 0;
      }
   }
}
```

结果如下：

csc ParameterTest.cs

```
Microsoft (R) Visual C# .NET Compiler version 8.00.40607.16
for Microsoft (R) .NET Framework version 2.0.40607
Copyright (C) Microsoft Corporation 2001-2003. All rights reserved.

ParameterTest.exe
i = 0
ints[0] = 0
Calling SomeFunction...
i = 0
ints[0] = 100
```

注意，i 的值保持不变，而在 ints 中改变的值在原来的数组中也改变了。

注意字符串是不同的，因为字符串是不能改变的(如果改变字符串的值，就会创建一个

全新的字符串)，所以字符串无法采用一般引用类型的行为方式。在方法调用中，对字符串所作的任何改变都不会影响原来的字符串。这一点将在第 8 章详细讨论。

(4) ref 参数

通过值传送变量是默认的，也可以迫使值参数通过引用传送给方法。为此，要使用 ref 关键字。如果把一个参数传递给方法，且这个方法的输入参数前带有 ref 关键字，则该方法对变量所作的任何改变都会影响原来对象的值：

```
static void SomeFunction(int[] ints, ref int i)
{
   ints[0] = 100;
   i = 100;      //the change to i will persist after SomeFunction() exits
}
```

在调用该方法时，还需要添加 ref 关键字：

```
SomeFunction(ints, ref i);
```

在 C#中添加 ref 关键字等同于在 C++中使用&语法指定按引用传递参数。但是，C#在调用方法时要求使用 ref 关键字，使操作更明确(因此有助于防止错误)。

最后，C#仍要求对传递给方法的参数进行初始化，理解这一点也是非常重要的。在传递给方法之前，无论是按值传递，还是按引用传递，变量都必须初始化。

(5) out 关键字

在 C 风格的语言中，函数常常能从一个例程中输出多个值，这是使用输出参数实现的，只要把输出值赋给通过引用传递给方法的变量即可。通常，变量通过引用传送的初值是不重要的，这些值会被函数重写，函数甚至从来没有使用过它们。

如果可以在 C#中使用这种约定，就会非常方便。但 C#要求变量在被引用前必须用一个初值进行初始化。尽管在把输入变量传递给函数前，可以用没有意义的值初始化它们，因为函数将使用真实、有意义的值初始化它们，但是这样做是没有必要的，有时甚至会引起混乱。但有一种方法能够简化 C#编译器所坚持的输入参数的初始化。

编译器使用 out 关键字来初始化。当在方法的输入参数前面加上 out 关键字时，传递给该方法的变量可以不初始化。该变量通过引用传送，所以在从被调用的方法中返回时，方法对该变量进行的任何改变都会保留下来。在调用该方法时，还需要使用 out 关键字，与在定义该方法时一样：

```
static void SomeFunction(out int i)
{
   i = 100;
}

public static int Main()
{
   int i; // note how i is declared but not initialized
   SomeFunction(out i);
   Console.WriteLine(i);
```

```
      return 0;
   }
```

out 关键字是 C#中的新增内容，在 Visual Basic 和 C++中没有对应的关键字，该关键字的引入使 C#更安全，更不容易出错。如果在函数体中没有给 out 参数分配一个值，该方法就不能编译。

(6) 方法的重载

C#支持方法的重载——方法的几个有不同签名(名称、参数个数、参数类型)的版本，但不支持 C++或 Visual Basic 中的默认参数。为了重载方法，只需声明同名但参数个数或类型不同的方法即可：

```
   class ResultDisplayer
   {
      void DisplayResult(string result)
      {
         // implementation
      }

      void DisplayResult(int result)
      {
         // implementation
      }
   }
```

因为 C#不直接支持可选参数，所以需要使用方法重载来达到此目的：

```
   class MyClass
   {
      int DoSomething(int x)   // want 2nd parameter with default value 10
      {
         DoSomething(x, 10);
      }

      int DoSomething(int x, int y)
      {
         // implementation
      }
   }
```

在任何语言中，对于方法重载来说，如果调用了错误的重载方法，就有可能出现运行错误。第 4 章将讨论如何使代码避免这些错误。现在，知道 C#在重载方法的参数方面有一些小区别即可：

- 两个方法不能仅在返回类型上有区别。
- 两个方法不能仅根据参数是声明为 ref 还是 out 来区分。

2. 属性

属性(property)不太常见，因为它们表示的概念是 C#从 Visual Basic 中提取的，而不是

从 C++/Java 中提取的。属性的概念是：它是一个方法或一对方法，在客户机代码看来，它们是一个字段。例如 Windows 窗体的 Height 属性。假定有下面的代码：

```
// mainForm is of type System.Windows.Form
mainForm.Height = 400;
```

执行这段代码，窗口的高度设置为 400，因此窗口会在屏幕上重新设置大小。在语法上，上面的代码类似于设置一个字段，但实际上是调用了属性访问器，它包含的代码重新设置了窗体的大小。

在 C#中定义属性，可以使用下面的语法：

```
public string SomeProperty
{
   get
   {
      return "This is the property value";
   }
   set
   {
      // do whatever needs to be done to set the property
   }
}
```

get 访问器不带参数，且必须返回属性声明的类型。也不应为 set 访问器指定任何显式参数，但编译器假定它带一个参数，其类型也与属性相同，并表示为 value。例如，下面的代码包含一个属性 ForeName，它设置了一个字段 foreName，该字段有一个长度限制。

```
private string foreName;

public string ForeName
{
   get
   {
      return foreName;
   }
   set
   {
      if (value.Length > 20)
         // code here to take error recovery action
         // (eg. throw an exception)
      else
         foreName = value;
   }
}
```

注意这里的命名模式。我们采用 C#的区分大小写模式，使用相同的名称，但公共属性采用 Pascal 大小写命名规则，而私有属性采用 camel 大小写命名规则。一些开发人员喜欢使用前面有下划线的字段名_foreName，这会为识别字段提供极大的便利。

VB6 程序员应注意，C#不区分 Visual Basic 6 中的 Set 和 Let，在 C#中，写入访问器总

是用关键字 set 标识。

(1) 只读和只写属性

在属性定义中省略 set 访问器，就可以创建只读属性。因此，把上面例子中的 ForeName 变成只读属性：

```
private string foreName;

public string ForeName
{
   get
   {
      return foreName;
   }
}
```

同样，在属性定义中省略 get 访问器，就可以创建只写属性。但是，这是不好的编程方式，因为这可能会使客户机代码的作者感到迷惑。一般情况下，如果要这么做，最好使用一个方法替代。

(2) 属性的访问修饰符

C#允许给属性的 get 和 set 访问器设置不同的访问修饰符，所以属性可以有公共的 get 访问器和私有或受保护的 set 访问器。这有助于控制属性的设置方式或时间。在下面的例子中，注意 set 访问器有一个私有访问修饰符，而 get 访问器没有任何访问修饰符。这表示 get 访问器具有属性的访问级别。在 get 和 set 访问器中，必须有一个具备属性的访问级别。如果 get 访问器的访问级别是 protected，就会产生一个编译错误，因为这会使两个访问器的访问级别都不是属性。

```
public string Name
{
   get
   {
      return _name;
   }
   set
   {
      _name = value;
   }
}
```

(3) 内联

一些开发人员可能会担心，在上一节中，我们列举了标准 C#编码方式导致了非常小的函数的许多情形，例如通过属性访问字段，而不是直接访问字段。这些额外的函数调用是否会增加系统开销，导致性能下降？其实，不需要担心这种编程方式会在 C#中带来性能损失。C#代码会编译为 IL，然后在运行期间进行正常的 JIT 编译，获得内部可执行代码。JIT 编译器可生成高度优化的代码，并在适当的时候内联代码(即用内联代码来替代函数调用)。

如果某个方法或属性的执行代码仅是调用另一个方法，或返回一个字段，则该方法或属性肯定是内联的。但要注意，在何处内联代码的决定完全由 CLR 做出。我们无法使用像 C++中 inline 这样的关键字来控制哪些方法是内联的。

3. 构造函数

在 C#中声明基本构造函数的语法与在 Java 和 C++中相同。下面声明一个与包含的类同名的方法，但该方法没有返回类型：

```
public class MyClass
{
   public MyClass()
   {
   }
   // rest of class definition
```

与 Java 和 C++相同，没有必要给类提供构造函数，在我们的例子中没有提供这样的构造函数。一般情况下，如果没有提供任何构造函数，编译器会在后台创建一个默认的构造函数。这是一个非常基本的构造函数，它只能把所有的成员字段初始化为标准的默认值(例如，引用类型为空引用，数字数据类型为 0，bool 为 false)。这通常就足够了，否则就需要编写自己的构造函数。

注意：

对于 C++程序员来说，C#中的基本字段在默认情况下初始化为 0，而 C++中的基本字段不进行初始化，不需要像 C++那样频繁地在 C#中编写构造函数。

构造函数的重载遵循与其他方法相同的规则。换言之，可以为构造函数提供任意多的重载，只要它们的签名有明显的区别即可：

```
   public MyClass()   // zero-parameter constructor
   {
      // construction code
   }
   public MyClass(int number)   // another overload
   {
      // construction code
   }
```

但注意，如果提供了带参数的构造函数，编译器就不会自动提供默认的构造函数，只有在没有定义任何构造函数时，编译器才会自动提供默认的构造函数。在下面的例子中，因为定义了一个带一个参数的构造函数，所以编译器会假定这是可以使用的唯一构造函数，不会隐式地提供其他构造函数：

```
public class MyNumber
{
   private int number;
   public MyNumber(int number)
```

```
    {
      this.number = number;
    }
}
```

上面的代码还说明，一般使用 this 关键字区分成员字段和同名的参数。如果试图使用无参数的构造函数实例化 MyNumber 对象，就会得到一个编译错误：

```
MyNumber numb = new MyNumber();   // causes compilation error
```

注意，可以把构造函数定义为 private 或 protected，这样不相关的类就不能访问它们：

```
public class MyNumber
{
  private int number;
  private MyNumber(int number)   // another overload
  {
    this.number = number;
  }
}
```

在这个例子中，我们并没有为 MyNumber 定义任何公共或受保护的构造函数。这就使 MyNumber 不能使用 new 运算符在外部代码中实例化(但可以在 MyNumber 上编写一个公共静态属性或方法，以进行实例化)。这在下面两种情况下是有用的：

- 类仅用作某些静态成员或属性的容器，因此永远不会实例化。
- 希望类仅通过调用某个静态成员函数来实例化(这就是所谓对象实例化的类代理方法)

(1) 静态构造函数

C#的一个新特征是也可以给类编写无参数的静态构造函数。这种构造函数只执行一次，而前面的构造函数是实例构造函数，只要创建类的对象，它都会执行。静态构造函数在 C++和 Visual Basic 6 中没有对应的函数。

```
class MyClass
{
  static MyClass()
  {
    // initialization code
  }
  // rest of class definition
}
```

编写静态构造函数的一个原因是，类有一些静态字段或属性，需要在第一次使用类之前，从外部源中初始化这些静态字段和属性。

.NET 运行库没有确保静态构造函数什么时候执行，所以不要把要求在某个特定时刻(例如，加载程序集时)执行的代码放在静态构造函数中。也不能预计不同类的静态构造函数按照什么顺序执行。但是，可以确保静态构造函数至多运行一次，即在代码引用类之前

执行。在 C#中，静态构造函数通常在第一次调用类的成员之前执行。

注意，静态构造函数没有访问修饰符，其他 C#代码从来不调用它，但在加载类时，总是由.NET 运行库调用它，所以像 public 和 private 这样的访问修饰符就没有意义了。同样，静态构造函数不能带任何参数，一个类也只能有一个静态构造函数。很显然，静态构造函数只能访问类的静态成员，不能访问实例成员。

注意，无参数的实例构造函数可以在类中与静态构造函数安全共存。尽管参数列表是相同的，但这并不矛盾，因为静态构造函数是在加载类时执行，而实例构造函数是在创建实例时执行，所以构造函数的执行不会有冲突。

如果多个类都有静态构造函数，先执行哪个静态构造函数是不确定的。此时静态构造函数中的代码不应依赖其他静态构造函数的执行情况。另一方面，如果静态字段有默认值，它们就在调用静态构造函数之前指定。

下面用一个例子来说明静态构造函数的用法，该例子基于包含用户设置的程序(用户设置假定存储在某个配置文件中)。为了简单一些，假定只有一个用户设置——BackColor，表示要在应用程序中使用的背景色。因为这里不想编写从外部数据源中读取数据的代码，所以假定该设置在工作日的背景色是红色，在周末的背景色是绿色。程序仅在控制台窗口中显示设置——但这足以说明静态构造函数是如何工作的。

```
namespace Wrox.ProCSharp.StaticConstructorSample
{
   public class UserPreferences
   {
      public static readonly Color BackColor;

      static UserPreferences()
      {
         DateTime now = DateTime.Now;
         if (now.DayOfWeek == DayOfWeek.Saturday
            || now.DayOfWeek == DayOfWeek.Sunday)
            BackColor = Color.Green;
         else
            BackColor = Color.Red;
      }

      private UserPreferences()
      {
      }
   }
}
```

这段代码说明了颜色设置如何存储在静态变量中，该静态变量在静态构造函数中进行初始化。把这个字段声明为只读类型，表示其值只能在构造函数中设置。本章后面将详细介绍只读字段。这段代码使用了 Microsoft 在 Framework 类库中支持的两个有用的结构 System.DateTime 和 System.Drawing.Color。DateTime 结构实现了静态属性 Now 和实例属性 DayOfWeek，Now 属性返回当前的时间，DayOfWeek 属性可以计算出某个日期是星期

几。Color(详见第 30 章)用于存储颜色，它实现了各种静态属性，例如本例使用的 Red 和 Green，返回常用的颜色。为了使用 Color 结构，需要在编译时引用 System.Drawing.dll 程序集，且必须为 System.Drawing 命名空间添加一个 using 语句：

```
using System;
using System.Drawing;
```

用下面的代码测试静态构造函数：

```
class MainEntryPoint
{
   static void Main(string[] args)
   {
      Console.WriteLine("User-preferences: BackColor is: " +
                        UserPreferences.BackColor.ToString());
   }
}
```

编译并运行这段代码，会得到如下结果：

StaticConstructor.exe

```
User-preferences: BackColor is: Color [Red]
```

(2) 从其他构造函数中调用构造函数

有时，在一个类中有几个构造函数，以容纳某些可选参数，这些构造函数都包含一些共同的代码。例如，下面的情况：

```
class Car
{
   private string description;
   private uint nWheels;
   public Car(string model, uint nWheels)
   {
      this.description = description;
      this.nWheels = nWheels;
   }

   public Car(string description)
   {
      this.description = description;
      this.nWheels = 4;
   }
// etc.
```

这两个构造函数初始化了相同的字段，显然，最好把所有的代码放在一个地方。C#有一个特殊的语法，称为构造函数初始化器，可以实现此目的：

```
class Car
{
   private string description;
```

```
      private uint nWheels;

      public Car(string description, uint nWheels)
      {
         this.description = description;
         this.nWheels = nWheels;
      }

      public Car(string description) : this(model, 4)
      {
      }
      // etc
```

这里，this 关键字仅调用参数最匹配的那个构造函数。注意，构造函数初始化器在构造函数之前执行。现在假定运行下面的代码：

```
   Car myCar = new Car("Proton Persona");
```

在本例中，在带一个参数的构造函数执行之前，先执行带 2 个参数的构造函数(但在本例中，因为带一个参数的构造函数没有代码，所以没有区别)。

C#构造函数初始化符可以包含对同一个类的另一个构造函数的调用(使用前面介绍的语法)，也可以包含对直接基类的构造函数的调用(使用相同的语法，但应使用 base 关键字代替 this)。初始化符中不能有多个调用。

在 C#中，构造函数初始化符的语法类似于 C++中的构造函数初始化列表，但 C++开发人员要注意，除了语法类似之外，C#初始化符所包含的代码遵循完全不同的规则。可以使用 C++初始化列表指定成员变量的初始值，或调用基类构造函数，而 C#初始化符中的代码只能调用另一个构造函数。这就要求 C#类在构造时遵循严格的顺序，但 C++就没有这个要求。这个问题详见第 4 章，那时就会看到，C#强制遵循的顺序只不过是良好的编程习惯而已。

3.2.3 只读字段

常量的概念就是一个包含不能修改的值的变量，常量是 C#与大多数编程语言共有的。但是，常量不必满足所有的要求。有时可能需要一些变量，其值不应改变，但在运行之前其值是未知的。C#为这种情形提供了另一个类型的变量：只读字段。

readonly 关键字比 const 灵活得多，允许把一个字段设置为常量，但可以执行一些运算，以确定它的初始值。其规则是可以在构造函数中给只读字段赋值，但不能在其他地方赋值。只读字段还可以是一个实例字段，而不是静态字段，类的每个实例可以有不同的值。与 const 字段不同，如果要把只读字段设置为静态，就必须显式声明。

如果有一个编辑文档的 MDI 程序，因为要注册，需要限制可以同时打开的文档数。现在假定要销售该软件的不同版本，而且顾客可以升级他们的版本，以便同时打开更多的文档。显然，不能在源代码中对最大文档数进行硬编码，而是需要一个字段表示这个最大文档数。这个字段必须是只读的——每次安装程序时，从注册表键或其他文件存储中读取。代码如下所示：

```
public class DocumentEditor
{
   public static readonly uint MaxDocuments;

   static DocumentEditor()
   {
      MaxDocuments = DosomethingToFindOutMaxNumber();
   }
```

在本例中，字段是静态的，因为每次运行程序的实例时，只需存储最大文档数一次。这就是在静态构造函数中初始化它的原因。如果只读字段是一个实例字段，就要在实例构造函数中初始化它。例如，假定编辑的每个文档都有一个创建日期，但不允许用户修改它(因为这会覆盖过去的日期)。注意，该字段也是公共的，我们不需要把只读字段设置为私有，因为按照定义，它们不能在外部修改(这个规则也适用于常量)。

如前所述，日期用基类 System.DateTime 表示。下面的代码使用带有 3 个参数(年份、月份和月份中的日)的 System.DateTime 构造函数，可以从 MSDN 文档中找到这个构造函数和其他 DateTime 构造函数的更多信息。

```
public class Document
{
   public readonly DateTime CreationDate;

   public Document()
   {
      // read in creation date from file. Assume result is 1 Jan 2002
      // but in general this can be different for different instances
      // of the class
      CreationDate = new DateTime(2002, 1, 1);
      }
   }
```

在上面的代码中，CreationDate 和 MaxDocuments 的处理方式与其他字段相同，但因为它们是只读的，所以不能在构造函数外部赋值：

```
void SomeMethod()
{
   MaxDocuments = 10; // compilation error here. MaxDocuments is readonly
}
```

还要注意，在构造函数中不必给只读字段赋值，如果没有赋值，它的值就是其数据类型的默认值，或者在声明时给它初始化的值。这适用于静态和实例只读字段。

3.3　结构

前面介绍了类如何封装程序中的对象，也介绍了如何将它们保存在堆中，通过这种方式可以在数据的生存期上获得很大的灵活性，但性能会有一定的损失。因托管堆的优化，

这种性能损失比较小。但是，有时仅需要一个小的数据结构。此时，类提供的功能多于我们需要的功能，由于性能的原因，最好使用结构。看看下面的例子：

```
class Dimensions
{
   public double Length;
   public double Width;
}
```

上面的示例代码定义了类 Dimensions，它只存储了一个项的长度和宽度。假定编写一个安排设备的程序，让人们试着重新安排计算机上的设备，并存储每个设备项的维数。看起来这样就会违背编程规则，使字段变为公共字段，但我们实际上并不需要类的全部功能。现在只有两个数字，把它们当作一对来处理，要比单个处理方便一些。既不需要很多方法，也不需要从类中继承，也不希望.NET 运行库在堆中遇到麻烦和性能问题，只需存储两个 double 类型的数据即可。

如本章前面所述，为此，只需修改代码，用关键字 struct 代替 class，定义一个结构而不是类：

```
   struct Dimensions
   {
      public double Length;
      public double Width;
   }
```

为结构定义函数与为类定义函数完全相同。下面的代码演示了结构的构造函数和属性：

```
   struct Dimensions
   {
      public double Length;
      public double Width;

      Dimensions(double length, double width)
      {
 Length= length;
Width= width;
}

      public double Diagonal
      {
         {
            get
            {
               return Math.Sqrt(Length* Length + Width* Width);
             }
         }
       }
```

在许多方面，可以把 C#中的结构看作是缩小的类。它们基本上与类相同，但更适合于

把一些数据组合起来的场合。它们与类的区别在于：

- 结构是值类型，不是引用类型。它们存储在堆栈中或存储为内联(inline)(如果它们是另一个保存在堆中的对象的一部分)，其生存期的限制与简单的数据类型一样。
- 结构不支持继承。
- 结构的构造函数的工作方式有一些区别。尤其是编译器总是提供一个无参数的默认构造函数，这是不允许替换的。
- 使用结构，可以指定字段如何在内存中布局(第 12 章在介绍属性时将详细论述这个问题)。

因为结构实际上是把数据项组合在一起，有时大多数甚至全部字段都声明为 public。严格说来，这与编写.NET 代码的规则相背——根据 Microsoft，字段(除了 const 字段之外)应总是私有的，并由公共属性封装。但是，对于简单的结构，许多开发人员都认为公共字段是可接受的编程方式。

注意：

C++开发人员要注意，C#中的结构在实现方式上与类大不相同。这与 C++的情形完全不同，在 C++中，类和结构是相同的对象。

下面将详细说明类和结构之间的区别。

3.3.1　结构是值类型

虽然结构是值类型，但在语法上常常可以把它们当作类来处理。例如，在上面的 Dimensions 类的定义中，可以编写下面的代码：

```
Dimensions point = new Dimensions();
point.Length = 3;
point.Width = 6;
```

注意，因为结构是值类型，所以 new 运算符与类和其他引用类型的工作方式不同。new 运算符并不分配堆中的内存，而是调用相应的构造函数，根据传送给它的参数，初始化所有的字段。对于结构，可以编写下述代码：

```
Dimensions point;
point.Length = 3;
point.Width = 6;
```

如果 Dimensions 是一个类，就会产生一个编译错误，因为 point 包含一个未初始化的引用——不指向任何地方的一个地址，所以不能给其字段设置值。但对于结构，变量声明实际上是为整个结构分配堆栈中的空间，所以就可以赋值了。但要注意下面的代码会产生一个编译错误，编译器会抱怨用户使用了未初始化的变量：

```
Dimensions point;
Double D = point.Length;
```

结构遵循其他数据类型都遵循的规则：在使用前所有的元素都必须进行初始化。在结

构上调用 new 运算符，或者给所有的字段分别赋值，结构就完全初始化了。当然，如果结构定义为类的成员字段，在初始化包含对象时，该结构会自动初始化为 0。

结构是值类型，所以会影响性能，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在堆栈中。在结构超出了作用域被删除时，速度也很快。另一方面，只要把结构作为参数来传递或者把一个结构赋给另一个结构(例如 A=B，其中 A 和 B 是结构)，结构的所有内容就被复制，而对于类，则只复制引用。这样，就会有性能损失，根据结构的大小，性能损失也不同。注意，结构主要用于小的数据结构。但当把结构作为参数传递给方法时，就应把它作为 ref 参数传递，以避免性能损失——此时只传递了结构在内存中的地址，这样传递速度就与在类中的传递速度一样快了。另一方面，如果这样做，就必须注意被调用的方法可以改变结构的值。

3.3.2 结构和继承

结构不是为继承设计的。不能从一个结构中继承，唯一的例外是结构(和 C#中的其他类型一样)派生于类 System.Object。因此，结构也可以访问 System.Object 的方法。在结构中，甚至可以重写 System.Object 中的方法—— 例如重写 ToString()方法。结构的继承链是：每个结构派生于 System.ValueType，System.ValueType 派生于 System.Object。ValueType 并没有给 Object 添加任何新成员，但提供了一些更适合结构的执行代码。注意，不能为结构提供其他基类：每个结构都派生于 ValueType。

3.3.3 结构的构造函数

为结构定义构造函数的方式与为类定义构造函数的方式相同，但不允许定义无参数的构造函数。这看起来似乎没有意义，其原因隐藏在.NET 运行库的执行方式中。下述情况非常少见：.NET 运行库不能调用用户提供的定制无参数构造函数，因此 Microsoft 采用一种非常简单的方式，禁止在 C#中的结构内使用无参数的构造函数。

前面说过，默认构造函数把所有的字段都初始化为 0，且总是隐式地给出，即使提供了其他带参数的构造函数，也是如此。也不能提供字段的初始值，以此绕过默认构造函数。下面的代码会产生编译错误：

```
struct Dimensions
{
   public double Length = 1;      // error. Initial values not allowed
   public double Width = 2;       // error. Initial values not allowed
}
```

当然，如果 Dimensions 声明为一个类，这段代码就不会有编译错误。

另外，可以像类那样为结构提供 Close()或 Dispose()方法。

3.4　部分类

partial 关键字允许把类、结构或接口放在多个文件中。一般情况下，一个类存储在单个文件中。但有时，多个开发人员需要访问同一个类，或者某种类型的代码生成器生成了一个类的某部分，所以把类放在多个文件中是有益的。

partial 关键字的用法是：把 partial 放在类、结构或接口的前面。在下面的例子中，TheBigClass 类位于两个不同的源文件 BigClassPart1.cs 和 BigClassPart2.cs 中：

```
//BigClassPart1.cs
partial class TheBigClass
{
  public void MethodOne()
  {
  }
}
```

```
//BigClassPart2.cs
partial class TheBigClass
{
  public void MethodTwo()
  {
  }
}
```

编译包含这两个源文件的项目时，会创建一个 TheBigClass 类，它有两个方法 MethodOne() 和 MethodTwo()。

如果声明类时使用了下面的关键字，这些关键字将应用于同一个类的所有部分：

- public
- private
- protected
- internal
- abstract
- sealed
- 基类
- new
- 一般约束

在嵌套的类型中，只要 partial 关键字位于 class 关键字的前面，就可以嵌套部分类。在把部分类编译到类型中时，会合并属性、XML 注释、接口、泛型类型的参数属性和成员。有如下两个源文件：

```
//BigClassPart1.cs
[CustomAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass
{
```

```
  public void MethodOne()
  {
  }
}
```

```
//BigClassPart2.cs
[AnotherAttribute]
partial class TheBigClass : IOtherBigClass
{
  public void MethodTwo()
  {
  }
}
```

编译后，源文件变成：

```
[CustomAttribute]
[AnotherAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass, IOtherBigClass
{
  public void MethodOne()
  {
  }

  public void MethodTwo()
  {
  }
}
```

3.5 静态类

本章前面讨论了静态构造函数，它们可以初始化静态的成员变量。如果类只包含静态的方法和属性，该类就是静态的。静态类在功能上与使用私有静态构造函数创建的类相同。不能创建静态类的实例。使用 static 关键字，编译器可以检查以后是否给该类添加了实例成员。如果是，就生成一个编译错误。这可以确保不创建静态类的实例。静态类的语法如下所示：

```
static class StaticUtilities
{
  public static void HelperMethod()
  {
  }
}
```

调用 HelperMethod()不需要 StaticUtilities 类型的对象。使用类型名即可进行该调用：

```
StaticUtilities.HelperMethod();
```

3.6　Object 类

前面提到，所有的.NET 类都派生于 System.Object。实际上，如果在定义类时没有指定基类，编译器就会自动假定这个类派生于 Object。本章没有使用继承，所以前面介绍的每个类都派生于 System.Object(如前所述，对于结构，这个派生是间接的：结构总是派生于 System.ValueType，System.ValueType 派生于 System.Object)。

其重要性在于，除了自己定义的方法和属性外，还可以访问为 Object 定义的许多公共或受保护的成员方法。这些方法可以用于自己定义的所有其他类中。

3.6.1　System.Object 方法

在 Object 中定义的方法如表 3-1 所示。

表　3-1

方　　法	访问修饰符	作　　用
string ToString()	public virtual	返回对象的字符串表示
intGetHashCode()	public virtual	在实现字典(散列表)时使用
bool Equals(object obj)	public virtual	对对象的实例进行相等比较
bool Equals(object objA,object objB)	public static	对对象的实例进行相等比较
Type GetType()	public	返回对象类型的详细信息
object MemberwiseClone()	protected	进行对象的浅表复制
void Finalize()	protected virtual	该方法是析构函数的.NET 版本

我们还没有完整地介绍 C#语言，所以用户还不能理解使用这些方法的方式。下面将简要总结每个方法的作用，但 ToString()方法要详细论述。

- ToString()方法：是获取对象的字符串表示的一种便捷方式。当只需要快速获取对象的内容，以用于调试时，就可以使用这个方法。在数据的格式化方面，它提供的选择非常少：例如，日期在原则上可以表示为许多不同的格式，但 DateTime.ToString()没有在这方面提供任何选择。如果需要更专业的字符串表示，例如考虑用户的格式化配置或文化(区域)，就应实现 IFormattable 接口(详见第 8 章)。
- GetHashTable()方法：如果对象放在名为映射(也称为散列表或字典)的数据结构中，就可以使用这个方法。处理这些结构的类使用该方法确定把对象放在结构的什么地方。如果希望把类用作字典的一个键，就需要重写 GetHashTable()方法。对该方法重载的执行方式有一些相当严格的限制，这些将在第 10 章介绍字典时讨论。
- Equals()(两个版本)和 ReferenceEquals()方法：如果把 3 个用于比较对象相等性的不同方法组合起来，就说明.NET Framework 在比较相等性方面有相当复杂的模式。这 3 个方法和比较运算符==在使用方式上有微妙的区别。而且，在重写带一个参数的虚拟 Equals()方法时也有一些限制，因为 System.Collections 命名空间中的一些

基类要调用该方法，并希望它以特定的方式执行。第 6 章在介绍运算符时将探讨这些方法的使用。

- Finalize()方法：第 11 章将介绍这个方法，它最接近 C++风格的析构函数，在引用对象被回收，以清理资源时调用。Finalize()方法的 Object 执行代码实际上什么也没有做，因而被垃圾收集器忽略。如果对象拥有对未托管资源的引用，则在该对象被删除时，就需要删除这些引用，此时一般要重写 Finalize()。垃圾收集器不能直接重写该方法，因为它只负责托管的资源，只能依赖用户提供的 Finalize()。
- GetType()方法：这个方法返回从 System.Type 派生的类的一个实例。这个对象可以提供对象所属类的更多信息，包括基本类型、方法、属性等。System.Type 还提供了.NET 反射技术的入口。这个主题详见第 12 章。
- MemberwiseClone()方法：这是 System.Object 中唯一没有在本书的其他地方详细论述的方法。不需要讨论这个方法，因为它在概念上相当简单，只是复制对象，返回一个对副本的引用(对于值类型，就是一个装箱的引用)。注意，得到的副本是一个浅表复制，即它复制了类中的所有值类型。如果类包含内嵌的引用，就只复制引用，而不复制引用的对象。这个方法是受保护的，所以不能用于复制外部的对象。该方法不是虚拟的，所以不能重写它的实现代码。

3.6.2 ToString()方法

第 2 章已经提到了 ToString()方法，它是快速获取对象的字符串表示的最便捷的方式。例如：

```
int i = -50;
string str = i.ToString();  // returns "-50"
```

下面是另一个例子：

```
enum Colors {Red, Orange, Yellow};
// later on in code...
Colors favoriteColor = Colors.Orange;
string str = favoriteColor.ToString();  // returns "Orange"
```

Object.ToString()声明为虚类型，在这些例子中，该方法的实现代码都是为 C#预定义数据类型重写过的代码，以返回这些类型的正确字符串表示。Colors 枚举是一个预定义的数据类型，它实际上实现为一个派生于 System.Enum 的结构，而 System.Enum 有一个相当聪明的 ToString()重写方法，来处理用户定义的所有枚举。

如果不在自己定义的类中重写 ToString()，该类将只继承 System.Object 执行方式——显示类的名称。如果希望 ToString()返回一个字符串，其中包含类中对象的值信息，就需要重写它。下面用一个例子 Money 来说明这一点。在该例子中，定义一个非常简单的类 Money，表示美元数。Money 是 decimal 类的包装器，提供了一个 ToString()方法。注意，这个方法必须声明为 override，因为它将替代(重写)Object 提供的 ToString()方法。第 4 章将详细讨论重写。该例子的完整代码如下所示(注意它还说明了如何使用属性封装字段)：

```
using System;

namespace Wrox.ProCSharp.OOCSharp
{
   class MainEntryPoint
   {
      static void Main(string[] args)
      {
         Money cash1 = new Money();
         cash1.Amount = 40M;
         Console.WriteLine("cash1.ToString() returns: " + cash1.ToString());
         Console.ReadLine();
      }
   }
   class Money
   {
      private decimal amount;

      public decimal Amount
      {
         get
         {
            return amount;
         }
         set
         {
            amount = value;
         }
      }
      public override string ToString()
      {
         return "$" + Amount.ToString();
      }
   }
}
```

这个例子仅说明了 C#的语法特性。C#已经有表示货币的预定义类型 decimal。所以在现实生活中，不必编写这样的类来重复该功能，除非要给它添加其他方法。在许多情况下，由于格式化要求，也可以使用 String.Format()方法(详见第 8 章)来表示货币字符串，而不是 ToString()。

在 Main()方法中，先实例化一个 Money 对象，再调用了 ToString()，执行该方法的重写版本。运行这段代码，会得到如下结果：

```
StringRepresentations
cash1.ToString() returns: $40
```

3.7 小结

本章介绍了 C#中声明和处理对象的语法，论述了如何声明静态和实例字段、属性、方法和构造函数。还讨论了 C#中新增的、其他语言的 OOP 模型中没有的新特性：静态构造函数提供了初始化静态字段的方式，利用结构可以定义高性能的类型，遵循更严格的特性集，不需要使用托管的堆。我们还阐述了 C#中的所有类型最终都派生于类 System.Object，这说明所有的类型都拥有一组基本的方法，包括 ToString()。

第 4 章将介绍 C#中的实现(implementation)继承和接口继承。

第 4 章 继 承

第 3 章介绍了如何使用 C#中的各个类，其重点是如何定义方法、构造函数、属性和单个类(或单个结构)中的其他成员。我们指出，所有的类最终都派生于 System.Object 类，但并没有说明如何创建继承类的层次结构。继承是本章的主题。我们将简要讨论 C#对继承的支持，然后详细论述如何在 C#中编码实现(implementation)继承和接口继承。注意，本章假定您已经熟悉了继承的基本概念，包括虚函数和重写。我们将重点阐述用于提供继承的语法和与继承相关的主题，例如虚函数，C#继承模型的其他方面是 C#所特有的，其他面向对象的语言都不具备。

4.1 继承的类型

首先介绍 C#在继承方面支持和不支持的功能。

4.1.1 实现继承和接口继承

在面向对象的编程中，有两种截然不同的继承类型：实现继承和接口继承。

- 实现继承：表示一个类型派生于一个基类型，拥有该基类型的所有成员字段和函数。在实现继承中，派生类型的每个函数采用基类型的实现代码，除非在派生类型的定义中指定重写该函数的实现代码。在需要给现有的类型添加功能，或许多相关的类型共享一组重要的公共功能时，这种类型的继承是非常有效的。例如第 28 章讨论的 Windows Forms 类。第 28 章也讨论了基类 System.Windows.Forms.Control，该类提供了常用 Windows 控件的非常复杂的实现代码，第 28 章还讨论了许多其他的类，例如 System. Windows.Forms.TextBox 和 System.Windows.Forms.ListBox，这两个类派生于 Control，并重写了函数，或提供了新的函数，以实现特定类型的控件。
- 接口继承：表示一个类型只继承了函数的签名，没有继承任何实现代码。在需要指定该类型具有某些可用的特性时，最好使用这种类型的继承。例如，某些类型可以指定从接口 System.IDisposable(详见第 11 章)中派生，从而提供一种清理资源

的方法 Dispose()。由于某种类型清理资源的方式可能与另一种类型的完全不同，所以定义通用的实现代码是没有意义的，此时就适合使用接口继承。接口继承常常被看做提供了一种契约：让类型派生于接口，来保证为客户提供某个功能。

在传统上，像 C++这样的语言在实现继承方面的功能非常强大。实际上，实现继承是 C++编程模型的核心。另一方面，VB6 不支持类的任何实现继承，但因其底层的 COM 基础体系，所以它支持接口继承。

在 C#中，既有实现继承，也有接口继承。它们没有强弱之分，因为这两种继承都完全内置于语言中，因此很容易为不同的情形选择最好的体系结构。

4.1.2 多重继承

一些语言如 C++支持所谓的“多重继承”，即一个类派生于多个类。使用多重继承的优点是有争议的：一方面，毫无疑问，可以使用多重继承编写非常复杂、但很紧凑的代码，如 C++ ATL 库。另一方面，使用多重实现继承的代码常常很难理解和调试(这也可以从 C++ ATL 库中看出)。如前所述，使健壮代码的编写容易一些，是开发 C#的重要设计目标。因此，C#不支持多重实现继承。而 C#又允许类型派生于多个接口。这说明，C#类可以派生于另一个类和任意多个接口。更准确地说，因为 System.Object 是一个公共的基类，所以每个 C#类(除了 Object 类之外)都有一个基类，还可以有任意多个基接口。

4.1.3 结构和类

第 3 章区分了结构(值类型)和类(引用类型)。使用结构的一个限制是结构不支持继承，但每个结构都自动派生于 System.ValueType。实际上还应更仔细一些：不能建立结构的类型层次，但结构可以实现接口。换言之，结构并不支持实现继承，但支持接口继承。事实上，定义结构和类可以总结为：

- 结构总是派生于 System.ValueType，它们还可以派生于任意多个接口。
- 类总是派生于用户选择的另一个类，它们还可以派生于任意多个接口。

4.2 实现继承

如果要声明一个类派生于另一个类，可以使用下面的语法：

```
class MyDerivedClass : MyBaseClass
{
   // functions and data members here
}
```

注意：

这个语法非常类似于 C++和 Java 中的语法，但是，C++程序员习惯于使用公共和私有继承的概念，要注意 C#不支持私有继承，因此基类名上没有 public 或 private 限定符。支持私有继承会大大增加语言的复杂性，实际上私有继承在 C++中也很少使用。

如果类(或结构)也派生于接口，则用逗号分隔开基类和接口：

```
public class MyDerivedClass : MyBaseClass, IInterface1, IInterface2
{
   //etc.

}
```

对于结构，语法如下：

```
public struct MyDerivedStruct : IInterface1, IInterface2
{
   //etc.
}
```

如果在类定义中没有指定基类，C#编译器就假定 System.Object 是基类。因此下面的两段代码生成相同的结果：

```
class MyClass : Object   //derives from System.Object
{
  //etc.
}
```

和

```
class MyClass   //derives from System.Object
{
  //etc.
}
```

第二种形式比较常用，因为它较简单。

C#支持 object 关键字，它用作 System.Object 类的假名，所以也可以编写下面的代码：

```
class MyClass : object   //derives from System.Object
{
  //etc.
}
```

如果要引用 Object 类，可以使用 object 关键字，智能编辑器(如 VS.NET)会识别它，因此便于编辑代码。

4.2.1 虚方法

把一个基类函数声明为 virtual，该函数就可以在派生类中重写了：

```
class MyBaseClass
{
  public virtual string VirtualMethod()
  {
    return "This method is virtual and defined in MyBaseClass";
  }
```

```
    }
```

也可以把属性声明为 virtual。对于虚属性或重写属性，语法与非虚属性是相同的，但要在定义中加上关键字 virtual，其语法如下所示：

```
    public virtual string ForeName
    {
       get { return foreName; }
       set { foreName = value; }
    }
    private string foreName;
```

为了简单起见，下面的讨论将主要集中于方法，但其规则也适用于属性。

C#中虚函数的概念与标准 OOP 概念相同：可以在派生类中重写虚函数。在调用方法时，会调用对象类型的合适方法。在 C#中，函数在默认情况下不是虚拟的，但(除了构造函数以外)可以显式地声明为 virtual。这遵循 C++的方式，即从性能的角度来看，除非显式指定，否则函数就不是虚拟的。而在 Java 中，所有的函数都是虚拟的。但 C#的语法与 C++的语法不同，因为 C#要求在派生类的函数重写另一个函数时，要使用 override 关键字显式声明：

```
    class MyDerivedClass : MyBaseClass
    {
       public override string VirtualMethod()
       {
          return "This method is an override defined in MyDerivedClass";
       }
    }
```

方法重写的语法避免了 C++中很容易发生的潜在运行错误：当派生类的方法签名无意中与基类版本略有差别时，派生类方法就不能重写基类方法了。在 C#中，这会出现一个编译错误，因为编译器会认为函数已标记为 override，但没有重写它的基类方法。

成员字段和静态函数都不能声明为 virtual，因为这个概念只对类中的实例函数成员有意义。

4.2.2 隐藏方法

如果签名相同的方法在基类和派生类中都进行了声明，但该方法没有声明为 virtual 和 override，派生类方法就会隐藏基类方法。

在大多数情况下，是要重写方法，而不是隐藏方法，因为隐藏方法会存在为给定类的实例调用错误方法的危险。但是，如下面的例子所示，C#语法可以确保开发人员在编译时收到这个潜在错误的警告，使隐藏方法更加安全。这也是类库开发人员得到的版本方面的好处。

假定有人编写了类 HisBaseClass：

```
    class HisBaseClass
    {
       // various members
    }
```

在将来的某一刻，要编写一个派生类，给 HisBaseClass 添加某个功能，特别是要添加

一个目前基类中没有的方法 MyGroovyMethod()：

```
class MyDerivedClass: HisBaseClass
{
   public int MyGroovyMethod()
   {
      // some groovy implementation
      return 0;
   }
}
```

一年后，基类的编写者决定扩展基类的功能。为了保持一致，他也添加了一个名为 MyGroovyMethod()的方法，该方法的名称和签名与前面添加的方法相同，但并不完成相同的工作。在使用基类的新方法编译代码时，程序在应该调用哪个方法上就会有潜在的冲突。这在C#中完全合法，但因为我们的MyGroovyMethod()与基类的MyGroovyMethod()不相关，运行这段代码的结果就可能不是我们希望的结果。C#已经为此设计了一种方式，可以很好地处理这种情况。

首先，系统会发出警告。在C#中，应使用 new 关键字声明我们要隐藏一个方法，如下所示：

```
class MyDerivedClass : HisBaseClass
{
   public new int MyGroovyMethod()
   {
      // some groovy implementation
      return 0;
   }
}
```

但是，我们的 MyGroovyMethod()没有声明为 new，所以编译器会认为它隐藏了基类的方法，但没有显式声明，因此发出一个警告(这也适用于把 MyGroovyMethod()声明为 virtual)。如果愿意，可以给我们的方法重命名。这么做，是最好的情形，因为这会避免许多冲突。但是，如果觉得重命名方法是不可能的(例如，已经为其他公司把软件发布为一个库，所以无法修改方法的名称)，则所有的已有客户机代码仍能正确运行，选择我们的 MyGroovyMethod()。这是因为访问这个方法的已有代码必须通过对 MyDerivedClass(或进一步派生的类)的引用进行选择。

已有的代码不能通过对 HisBaseClass 的引用访问这个方法，因为在对 HisBaseClass 的早期版本进行编译时，会产生一个编译错误。这个问题只会发生在将来编写的客户机代码上。C#会发出一个警告，告诉用户在将来的代码中可能会出问题——用户应注意这个警告，不要试图在将来的代码中通过对 HisBaseClass 的引用调用 MyGroovyMethod()方法，但所有已有的代码仍会正常工作。这是比较微妙的，但很好地说明了 C#如何处理类的不同版本。

4.2.3 调用函数的基类版本

C#有一种特殊的语法用于从派生类中调用方法的基类版本：base.<MethodName>()。例如，假定派生类中的一个方法要返回基类的方法返回的值的 90%，就可以使用下面的语法：

```
class CustomerAccount
{
   public virtual decimal CalculatePrice()
   {
      // implementation
      return 0.0M;
   }
}
class GoldAccount : CustomerAccount
{
   public override decimal CalculatePrice()
   {
      return base.CalculatePrice() * 0.9M;
   }
}
```

这个语法类似于 Java，但 Java 使用关键字 super 而不是 base。C++没有类似的关键字，但需要显式指定类名(CustomerAccount::CalculatePrice())。C++中对应于 base 的内容都比较模糊，因此 C++允许多重继承。

注意，可以使用 base.<MethodName>()语法调用基类中的任何方法，不必在同一个方法的重载中调用它。

4.2.4 抽象类和抽象函数

C#允许把类和函数声明为 abstract，抽象类不能实例化，而抽象函数没有执行代码，必须在非抽象的派生类中重写。显然，抽象函数也是虚拟的(但也不需要提供 virtual 关键字，实际上，如果提供了该关键字，就会产生一个语法错误)。如果类包含抽象函数，该类将也是抽象的，也必须声明为抽象的：

```
abstract class Building
{
   public abstract decimal CalculateHeatingCost();    // abstract method
}
```

C++开发人员要注意 C#中的一些语法区别。C#不支持采用=0 语法来声明抽象函数。在 C#中，这个语法有误导作用，因为可以在类声明的成员字段上使用=<value>，提供初始值：

```
abstract class Building
{
   private bool damaged = false;   // field
   public abstract decimal CalculateHeatingCost();   // abstract method
}
```

注意：

C++开发人员还要注意术语上的细微差别：在 C++中，抽象函数常常描述为纯虚函数，而在 C#中，仅使用抽象这个术语。

4.2.5 密封类和密封方法

C#允许把类和方法声明为 sealed。对于类来说，这表示不能继承该类；对于方法来说，这表示不能重写该方法。

```
sealed class FinalClass
{
   // etc
}
class DerivedClass : FinalClass      // wrong. Will give compilation error
{
   // etc
}
```

注意：

Java 开发人员可以把 C#中的 sealed 当作 Java 中的 final。

在把类或方法标记为 sealed 时，最可能的情形是：如果要对库、类或自己编写的其他类进行操作，则重写某些功能会导致错误。也可以因商业原因把类或方法标记为 sealed，以防第三方以违反注册协议的方式扩展该类。但一般情况下，在把类或方法标记为 sealed 时要小心，因为这么做会严重限制它的使用。即使不希望它能继承一个类或重写类的某个成员，仍有可能在将来的某个时刻，有人会遇到我们没有预料到的情形。.NET 基类库大量使用了密封类，使希望从这些类中派生出自己的类的第三方开发人员无法访问这些类。例如 string 就是一个密封类。

把方法声明为 sealed 也可以实现类似的目的，但很少这么做。

```
class MyClass
{
   public sealed override void FinalMethod()
   {
      // etc.
   }
}
class DerivedClass : MyClass
{
   public override void FinalMethod()     // wrong. Will give compilation error
   {
   }
}
```

在方法上使用 sealed 关键字是没有意义的，除非该方法本身是某个基类上另一个方法的重写形式。如果定义一个新方法，但不想让别人重写它，首先就不要把它声明为 virtual。但如果要重写某个基类方法，sealed 关键字就提供了一种方式，可以确保为方法提供的重写代码是最终的代码，其他人不能再重写它。

4.2.6 派生类的构造函数

第 3 章介绍了单个类的构造函数是如何工作的。这样，就产生了一个有趣的问题，在开始为层次结构中的类(这个类继承了其他类，也可能有定制的构造函数)定义自己的构造函数时，会发生什么情况？

假定没有为类定义任何显式的构造函数，这样编译器就会为所有的类提供默认的构造函数，在后台会进行许多操作，编译器可以很好地解决层次结构中的所有问题，每个类中的每个字段都会初始化为默认值。但在添加了一个我们自己的构造函数后，就要通过派生类的层次结构高效地控制构造过程，因此必须确保构造过程顺利进行，不要出现不能按照层次结构进行构造的问题。

为什么派生类会有某些特殊的问题？原因是在创建派生类的实例时，实际上会有多个构造函数起作用。要实例化的类的构造函数本身不能初始化类，还必须调用基类中的构造函数。这就是为什么要通过层次结构进行构造的原因。

为了说明为什么必须调用基类的构造函数，下面是手机公司 MortimerPhones 开发的一个例子。这个例子包含一个抽象类 GenericCustomer，它表示顾客。还有一个(非抽象)类 Nevermore60Customer，它表示采用特定付费方式(称为 Nevermore60 付费方式)的顾客。所有的顾客都有一个名字，由一个私有字段表示。在 Nevermore60 付费方式中，顾客前几分钟的电话费比较高，需要一个字段 highCostMinutesUsed，它详细说明了每个顾客该如何支付这些较高的电话费。抽象类 GenericCustomer 的定义如下所示：

```
abstract class GenericCustomer
{
   private string name;
   // lots of other methods etc.
}
class Nevermore60Customer : GenericCustomer
{
   private uint highCostMinutesUsed;
   // other methods etc.
}
```

不要担心在这些类中执行的其他方法，因为这里仅考虑构造过程。如果下载了本章的示例代码，就会发现类的定义仅包含构造函数。

下面看看使用 new 运算符实例化 Nevermore60Customer 时，会发生什么情况：

```
   GenericCustomer customer = new Nevermore60Customer();
```

显然，成员字段 name 和 highCostMinutesUsed 都必须在实例化 customer 时进行初始化。如果没有提供自己的构造函数，而是仅依赖默认的构造函数，name 就会初始化为 null 引用，highCostMinutesUsed 初始化为 0。下面详细讨论其过程。

highCostMinutesUsed 字段没有问题：编译器提供的默认 Nevermore60Customer 构造函数会把它初始化为 0。

那么 name 呢？看看类定义，显然，Nevermore60Customer 构造函数不能初始化这个值。

字段 name 声明为 private，这意味着派生的类不能访问它。默认的 Nevermore60Customer 构造函数甚至不知道存在这个字段。唯一知道这个字段的是 GenericCustomer 的其他成员，即如果对 name 进行初始化，就必须在 GenericCustomer 的某个构造函数中进行。无论类层次结构有多大，这种情况都会一直延续到最终的基类 System.Object 上。

理解了上面的问题后，就可以明白实例化派生类时会发生什么样的情况了。假定默认的构造函数在整个层次结构中使用：编译器首先找到它试图实例化的类的构造函数，在本例中是 Nevermore60Customer，这个默认 Nevermore60Customer 构造函数首先要做的是为其直接基类 GenericCustomer 运行默认构造函数，然后 GenericCustomer 构造函数为其直接基类 System.Object 运行默认构造函数，System. Object 没有任何基类，所以它的构造函数就执行，并把控制返回给 GenericCustomer 构造函数。现在执行 GenericCustomer 构造函数，把 name 初始化为 null，再把控制权返回给 Nevermore60Customer 构造函数，接着执行这个构造函数，把 highCostMinutesUsed 初始化为 0，并退出。此时，Nevermore60Customer 实例就已经成功地构造和初始化了。

构造函数的调用顺序是先调用 System.Object，再按照层次结构由上向下进行，直到到达编译器要实例化的类为止。还要注意在这个过程中，每个构造函数都初始化它自己的类中的字段。这是它的一般工作方式，在开始添加自己的构造函数时，也应尽可能遵循这个规则。

注意构造函数的执行顺序。基类的构造函数总是最先调用。也就是说，派生类的构造函数可以在执行过程中调用它可以访问的基类方法、属性和其他成员，因为基类已经构造出来的，其字段也初始化了。如果派生类不喜欢初始化基类的方式，但要访问数据，就可以改变数据的初始值，但是，好的编程方式应尽可能避免这种情况，让基类构造函数来处理其字段。

理解了构造过程后，就可以开始添加自己的构造函数了。

1. 在层次结构中添加无参数的构造函数

首先讨论最简单的情况，在层次结构中用一个无参数的构造函数来替换默认的构造函数后，看看会发生什么情况。假定要把每个人的名字初始化为<no name>，而不是 null 引用，修改 GenericCustomer 中的代码，如下所示：

```
public abstract class GenericCustomer
{
   private string name;
   public GenericCustomer()
      : base()  // we could omit this line without affecting the compiled code
   {
      name = "<no name>";
   }
```

添加这段代码后，代码运行正常。Nevermore60Customer 仍有自己的默认构造函数，所以上面描述的事件顺序仍不变，但编译器会使用定制的 GenericCustomer 构造函数，而不是生成默认的构造函数，所以 name 字段按照需要总是初始化为<no name>。

注意，在定制的构造函数中，在执行 GenericCustomer 构造函数前，添加了一个对基类构造函数的调用，使用的语法与前面解释如何让构造函数的不同重载版本互相调用时使用的语法相同。唯一的区别是，这次使用的关键字是 base，而不是 this，表示这是基类的构造函数，而不是要调用的类的构造函数。在 base 关键字后面的圆括号中没有参数，这是非常重要的，因为没有给基类构造函数传送参数，所以编译器会调用无参数的构造函数。其结果是编译器会插入调用 System.Object 构造函数的代码，这正好与默认情况相同。

实际上，可以把这行代码删除，只加上为本章中大多数构造函数编写的代码：

```
public GenericCustomer()
{
   name = "<no name>";
}
```

如果编译器没有在起始花括号的前面找到对另一个构造函数的任何引用，它就会假定我们要调用基类构造函数——这符合默认构造函数的工作方式。

base 和 this 关键字是调用另一个构造函数时允许使用的唯一关键字，其他关键字都会产生编译错误。还要注意只能指定一个其他的构造函数。

到目前为止，这段代码运行正常。但是，要通过构造函数的层次结构把级数弄乱的最好方法是把构造函数声明为私有：

```
private GenericCustomer()
{
   name = "<no name>";
}
```

如果试图这样做，就会产生一个有趣的编译错误，如果不理解构造是如何按照层次结构由上而下的顺序工作的，这个错误会让人摸不着头脑。

```
'Wrox.ProCSharp.GenericCustomer()' is inaccessible due to its protection level
```

有趣的是，该错误没有发生在 GenericCustomer 类中，而是发生在 Nevermore60Customer 派生类中。编译器试图为 Nevermore60Customer 生成默认的构造函数，但又做不到，因为默认的构造函数应调用无参数的 GenericCustomer 构造函数。把该构造函数声明为 private，它就不可能访问派生类了。如果为带有参数的 GenericCustomer 提供一个构造函数，但没有提供无参数的构造函数，也会发生类似的错误。在本例中，编译器不能为 GenericCustomer 生成默认构造函数，所以当编译器试图为派生类生成默认构造函数时，会再次发现它不能做到这一点，因为没有无参数的基类构造函数可调用。这个问题的解决方法是为派生类添加自己的构造函数——实际上不需要在这些构造函数中做任何工作，这样，编译器就不会为这些派生类生成默认构造函数了。

前面介绍了所有的理论知识，下面用一个例子来说明如何给类的层次结构添加构造函数。下一节为 MortimerPhones 样例添加带参数的构造函数。

2. 在层次结构中添加带参数的构造函数

首先是带一个参数的 GenericCustomer 构造函数，它仅在顾客提供其姓名时才实例化顾客：

```
abstract class GenericCustomer
{
   private string name;
   public GenericCustomer(string name)
   {
      this.name = name;
   }
```

到目前为止，代码运行一切正常，但刚才说过，在编译器试图为派生类创建默认构造函数时，会产生一个编译错误，因为编译器为 Nevermore60Customer 生成的默认构造函数会试图调用无参数的 GenericCustomer 构造函数，但 GenericCustomer 没有这样的构造函数。因此，需要为派生类提供一个构造函数，来避免这个错误：

```
class Nevermore60Customer : GenericCustomer
{
   private uint highCostMinutesUsed;
   public Nevermore60Customer(string name)
      :   base(name)
   {
   }
```

现在，Nevermore60Customer 对象的实例化只能在提供了包含顾客姓名的字符串后进行，这正是我们需要的。有趣的是 Nevermore60Customer 构造函数对这个字符串所做的处理。它本身不能初始化 name 字段，因为它不能访问基类中的私有字段，但可以把顾客姓名传送给基类，以便 GenericCustomer 构造函数处理。具体方法是，把先执行的基类构造函数指定为把顾客姓名当做参数的构造函数。除此之外，它不需要执行任何操作。

下面讨论如果要处理不同的重载构造函数和一个类的层次结构，会发生什么情况。假定 Nevermore60Customers 通过朋友联系到 MortimerPhones，即 MortimerPhones 公司中有一个人是朋友，因此可以获得折扣。这表示在构造一个 Nevermore60Customer 时，还需要传递联系人的姓名。在现实生活中，构造函数必须利用该姓名去完成更复杂的工作，例如处理折扣等，但这里只是把联系人的姓名存储到另一个字段中。

此时，Nevermore60Customer 定义如下所示：

```
class Nevermore60Customer : GenericCustomer
{
   public Nevermore60Customer(string name, string referrerName)
      : base(name)
   {
      this.referrerName = referrerName;
   }

   private string referrerName;
   private uint highCostMinutesUsed;
```

该构造函数将姓名作为参数，把它传递给 GenericCustomer 构造函数进行处理。referrerName 是一个变量，我们需要声明它，这样构造函数才能在其主体中处理这个参数。

但是，并不是所有的 Nevermore60Customers 都有联系人，所以还需要有一个不需此参数的构造函数(或为它提供默认值的构造函数)。实际上，我们指定如果没有联系人，referrerName 字段就设置为<None>。下面是这个带一个参数的构造函数：

```
public Nevermore60Customer(string name)
   : this(name, "<None>")
{
}
```

这样就正确建立了所有的构造函数。执行下面的代码时，检查事件链是很有益的：

```
GenericCustomer customer = new Nevermore60Customer("Arabel Jones");
```

编译器认为它需要带一个字符串参数的构造函数，所以它确认的构造函数就是刚才定义的那个构造函数，如下所示。

```
public Nevermore60Customer(string Name)
   : this(Name, "<None>")
```

在实例化 customer 时，就会调用这个构造函数。之后立即把控制权传送给对应的 Nevermore60Customer 构造函数，该构造函数带 2 个参数，分别是 Arabel Jones 和<None>。在这个构造函数中，把控制权依次传送给 GenericCustomer 构造函数，该构造函数带有 1 个参数，即字符串 Arabel Jones。然后这个构造函数把控制权传送给 System.Object 默认构造函数。现在执行这些构造函数，首先执行 System.Object 构造函数，接着执行 GenericCustomer 构造函数，初始化 name 字段。然后带有两个参数的 Nevermore60Customer 构造函数得到控制权，把联系人的姓名初始化为<None>。最后，执行 Nevermore60Customer 构造函数，该构造函数带有 1 个参数——这个构造函数什么也不做。

这个过程非常简洁，设计也很合理。每个构造函数都处理变量的初始化。在这个过程中，正确地实例化了类，以备使用。如果在为类编写自己的构造函数时遵循这个规则，即便是最复杂的类，也可以顺利地初始化，不会出现任何问题。

4.3 修饰符

前面已经遇到许多所谓的修饰符，即应用于类型或成员的关键字。修饰符可以指定方法的可见性，例如 public 或 private，还可以指定一项的本质，例如方法是 virtual 或 abstract。C#有许多访问修饰符，下面讨论完整的修饰符列表。

4.3.1 可见性修饰符

表 4-1 中的修饰符确定了是否允许其他代码访问某一项。

表　4-1

修　饰　符	应　用　于	说　　明
public	所有的类型或成员	任何代码均可以访问该方法
protected	类型和内嵌类型的所有成员	只有派生的类型能访问该方法
internal	类型和内嵌类型的所有成员	只能在包含它的程序集中访问该方法
private	所有的类型或成员	只能在它所属的类型中访问该方法
protected internal	类型和内嵌类型的所有成员	只能在包含它的程序集和派生类型的代码中访问该方法

注意，类型定义可以是内部或公共的，这取决于是否希望在包含类型的程序集外部访问它：

```
public class MyClass
{
//etc.
```

不能把类型定义为 protected、private 和 protected internal，因为这些修饰符对于包含在命名空间中的类型来说是没有意义的。因此这些修饰符只能应用于成员。但是，可以用这些修饰符定义嵌套的类型(即包含在其他类型中的类型)，因为在这种情况下，类型也具有成员的状态。下面的代码是合法的：

```
public class OuterClass
{
   protected class InnerClass
   {
        //etc.
    }
    //etc.
}
```

如果有嵌套的类型，内部的类型总是可以访问外部类型的所有成员，所以在上面的代码中，InnerClass 中的代码可以访问 OuterClass 的所有成员，甚至可以访问 OuterClass 的私有成员。

4.3.2　其他修饰符

表 4-2 中的修饰符可以应用于类型的成员，而且有不同的用途。在应用于类型时，其中的几个修饰符也是有意义的。

表　4-2

修　饰　符	应　用　于	说　　明
new	函数成员	成员用相同的签名隐藏继承的成员
static	所有的成员	成员不在类的具体实例上执行
virtual	仅类和函数成员	成员可以由派生类重写

(续表)

修 饰 符	应 用 于	说 明
abstract	仅函数成员	虚拟成员定义了成员的签名，但没有提供实现代码
override	仅函数成员	成员重写了继承的虚拟或抽象成员
sealed	类	成员重写了继承的虚拟成员，但继承该类的任何类都不能重写该成员。该修饰符必须与 override 一起使用
extern	仅静态[DllImport]方法	成员在外部用另一种语言实现

在这些修饰符中，internal 和 protected internal 是 C#和.NET Framework 新增的。internal 与 public 类似，但访问仅限于同一个程序集中的其他代码，换言之，在同一个程序中同时编译的代码。使用 internal 可以确保编写的其他类都能访问某一成员，但同时其他公司编写的其他代码不能访问它们。protected internal 合并了 protected 和 internal，但这是一种 OR 合并，而不是 AND 合并。protected internal 成员在同一个程序集的任何代码中都可见，在派生类中也可见，甚至在其他程序集中也可见。

4.4 接口

如前所述，如果一个类派生于一个接口，它就会执行某些函数。并不是所有的面向对象语言都支持接口，所以本节将详细介绍 C#接口的实现。

注意：

熟悉 COM 的开发人员应注意，尽管在概念上 C#接口类似于 COM 接口，但它们是不同的，底层的结构不同，例如，C#接口并不派生于 IUnknown。C#接口根据.NET 函数提供了一个契约。与 COM 接口不同，C#接口不代表任何类型的二进制标准。

下面列出 Microsoft 预定义的一个接口 System.IDisposable 的完整定义。IDisposable 包含一个方法 Dispose()，该方法由类执行，用于清理代码：

```
public interface IDisposable
{
   void Dispose();
}
```

上面的代码说明，声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的执行方式。一般情况下，接口中只能包含方法、属性、索引器和事件的声明。

不能实例化接口，它只能包含其成员的签名。接口不能有构造函数(如何构建不能实例化的对象？)或字段(因为这隐含了某些内部的执行方式)。接口定义也不允许包含运算符重载，但这不是因为声明它们在原则上有什么问题，而是因为接口通常是公共契约，包含运算符重载会引起一些与其他.NET 语言不兼容的问题，例如与 VB.NET 的不兼容，因为 VB.NET 不支持运算符重载。

在接口定义中还不允许声明成员上的修饰符。接口成员总是公共的，不能声明为虚拟或静态。如果需要，就应由执行的类来声明，因此最好通过执行的类来声明访问修饰符，就像上面的代码那样。

例如 IDisposable。如果类希望声明为公共类型，以便执行方法 Dispose()，该类就必须执行 IDisposable。在 C#中，这表示该类派生于 IDisposable。

```
class SomeClass : IDisposable
{
   // this class MUST contain an implementation of the
   // IDisposable.Dispose() method, otherwise
   // you get a compilation error
   public void Dispose()
   {
      // implementation of Dispose() method
   }
   // rest of class
}
```

在这个例子中，如果 SomeClass 派生于 IDisposable，但不包含与 IDisposable 中签名相同的 Dispose()实现代码，就会得到一个编译错误，因为该类破坏了实现 IDisposable 的契约。当然，编译器允许类有一个不派生于 IDisposable 的 Dispose()方法。问题是其他代码无法识别出 SomeClass 支持 IDisposable 特性。

注意：

IDisposable 是一个相当简单的接口，它只定义了一个方法。大多数接口都包含许多成员。

接口的另一个例子是 C#中的 foreach 循环。实际上，foreach 循环的内部工作方式是查询对象，看看它是否实现了 System.Collections.IEnumerable 接口。如果是，C#编译器就插入 IL 代码，使用这个接口上的方法迭代集合中的成员，否则，foreach 就会引发一个异常。第 10 章将详细介绍 IEnumerable 接口。但应注意，IEnumerable 和 IDisposable 在某种程度上都是有点特殊的接口，因为它们都可以由 C#编译器识别，在 C#编译器生成的代码中会考虑它们。显然，自己定义的接口就没有这个特权。

4.4.1 定义和实现接口

下面开发一个遵循接口继承规范的小例子来说明如何定义和使用接口。这个例子建立在银行账户的基础上。假定编写代码，最终允许在银行账户之间进行计算机转账业务。许多公司可以实现银行账户，但它们都是彼此赞同表示银行账户的所有类都实现接口 IBankAccount。该接口包含一个用于存取款的方法和一个返回余额的属性。这个接口还允许外部代码识别由不同银行账户执行的各种银行账户类。我们的目的是允许银行账户彼此通信，以便在账户之间进行转账业务，但还没有介绍这个功能。

为了使例子简单一些，我们把例子的所有代码都放在同一个源文件中，但实际上不同的银行账户类会编译到不同的程序集中，而这些程序集位于不同银行的不同机器上。第 37 章在讨论远程通信时，将介绍位于不同机器上的.NET 程序集如何通信。但那些内容对于

这里的例子来说过于复杂了。为了保留一定的真实性，我们为不同的公司定义不同的命名空间。

首先，需要定义 IBank 接口：

```
namespace Wrox.ProCSharp
{
   public interface IBankAccount
   {
      void PayIn(decimal amount);
      bool Withdraw(decimal amount);
      decimal Balance
      {
         get;
      }
   }
}
```

注意，接口的名称为 IBankAccount。接口名称传统上以字母 I 开头，以便知道这是一个接口。

注意：

如第 2 章所述，在大多数情况下，.NET 用法规则不鼓励采用所谓的 Hungarian 表示法，在名称的前面加一个字母，表示对象的类型，接口是 Hungarian 表示法推荐采用的几种名称之一。

现在可以编写表示银行账户的类了。这些类不必彼此相关，它们可以是完全不同的类。但它们都表示银行账户，因为它们都实现了 IBankAccount 接口。

下面是第一个类，一个由 Royal Bank of Venus 运行的存款账户：

```
namespace Wrox.ProCSharp.VenusBank
{
   public class SaverAccount : IBankAccount
   {
      private decimal balance;
      public void PayIn(decimal amount)
      {
         balance += amount;
      }
      public bool Withdraw(decimal amount)
      {
         if (balance >= amount)
         {
            balance -= amount;
            return true;
         }
         Console.WriteLine("Withdrawal attempt failed.");
         return false;
      }
```

```
        public decimal Balance
        {
           get
           {
              return balance;
           }
        }
        public override string ToString()
        {
           return String.Format("Venus Bank Saver: Balance = {0,6:C}", balance);
        }
     }
  }
```

这个类的实现代码的作用一目了然。其中包含一个私有字段 balance，当存款或取款时就调整这个字段。如果因为账户中的金额不足而取款失败，就会显示一个错误消息。还要注意，因为我们要使代码尽可能简单，所以不实现额外的属性，例如账户持有人的姓名。在现实生活中，这是最基本的信息，但对于本例来说，这是不必要的。

在这段代码中，唯一有趣的是类的声明：

```
  public class SaverAccount : IBankAccount
```

SaverAccount 派生于一个接口 IbankAccount，我们没有明确指出任何其他基类(当然这表示 SaverAccount 直接派生于 System.Object)。另外，从接口中派生完全独立于从类中派生。

SaverAccount 派生于 IBankAccount，表示它获得了 IBankAccount 的所有成员，但接口并不实际实现其方法，所以 SaverAccount 必须提供这些方法的所有实现代码。如果没有提供实现代码，编译器就会产生错误。接口仅表示其成员的存在性，类负责确定这些成员是虚拟还是抽象的(但只有在类本身是抽象的，这些成员才能是抽象的)。在本例中，接口方法不必是虚拟的。

为了说明不同的类如何实现相同的接口，下面假定 Planetary Bank of Jupiter 还实现一个类 Gold Account 来表示其银行账户：

```
  namespace Wrox.ProCSharp.JupiterBank
  {
     public class GoldAccount : IBankAccount
     {
        // etc
     }
  }
```

这里没有列出 GoldAccount 类的细节，因为在本例中它基本上与 SaverAccount 的实现代码相同。GoldAccount 与 VenusAccount 没有关系，它们只是碰巧实现相同的接口而已。

有了自己的类后，就可以测试它们了。首先需要一些 using 语句：

```
  using System;
  using Wrox.ProCSharp;
```

```
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;
```

然后需要一个 Main()方法：

```
namespace Wrox.ProCSharp
{
  class MainEntryPoint
  {
    static void Main()
    {
      IBankAccount venusAccount = new SaverAccount();
      IBankAccount jupiterAccount = new GoldAccount();
      venusAccount.PayIn(200);
      venusAccount.Withdraw(100);
      Console.WriteLine(venusAccount.ToString());
      jupiterAccount.PayIn(500);
      jupiterAccount.Withdraw(600);
      jupiterAccount.Withdraw(100);
      Console.WriteLine(jupiterAccount.ToString());
    }
  }
}
```

这段代码(如果下载本例子，它在 BankAccounts.cs 文件中)的执行结果如下：

```
C:>BankAccounts
Venus Bank Saver: Balance = £100.00
Withdrawal attempt failed.
Jupiter Bank Saver: Balance = £400.00
```

在这段代码中，一个要点是把引用变量声明为 IBankAccount 引用的方式。这表示它们可以指向实现这个接口的任何类的实例。但我们只能通过这些引用调用接口的方法——如果要调用由类执行的、不在接口中的方法，就需要把引用强制转换为合适的类型。在这段代码中，我们调用了 ToString()(不由 IBankAccount 实现)，但没有进行任何显式转换，这只是因为 ToString()是一个 System.Object 方法，C#编译器知道任何类都支持这个方法(换言之，从接口到 System.Object 的数据类型转换是隐式的)。第 6 章将介绍强制转换的语法。

接口引用完全可以看做是类引用—— 但接口引用的强大之处在于，它可以引用任何实现该接口的类。例如，我们可以构造接口数组，其中的每个元素都是不同的类：

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
accounts[1] = new GoldAccount();
```

但注意，如果编写了如下代码，就会生成一个编译错误：

```
accounts[1] = new SomeOtherClass(); // SomeOtherClass does NOT implement
                    // IBankAccount: WRONG!!
```

这会导致一个如下所示的编译错误：

```
Cannot implicitly convert type 'Wrox.ProCSharp.SomeOtherClass' to
 'Wrox.ProCSharp.IBankAccount'
```

4.4.2 派生的接口

接口可以彼此继承，其方式与类的继承相同。下面通过定义一个新接口 ITransferBankAccount 来说明这个概念，该接口的功能与 IBankAccount 相同，只是又定义了一个方法，把资金直接转到另一个账户上。

```
namespace Wrox.ProCSharp
{
   public interface ITransferBankAccount : IBankAccount
   {
      bool TransferTo(IBankAccount destination, decimal amount);
   }
}
```

因为 ITransferBankAccount 派生于 IBankAccount，所以拥有 IBankAccount 的所有成员和它自己的成员。这表示执行(派生于)ITransferBankAccount 的任何类都必须执行 IBankAccount 的所有方法和在 ITransferBankAccount 中定义的新方法 TransferTo()。没有执行所有这些方法就会产生一个编译错误。

注意，TransferTo()方法为目标账户使用了 IBankAccount 接口引用。这说明了接口的用途：在执行并调用这个方法时，不必知道转帐的对象类型，只需知道该对象执行 IBankAccount 即可。

下面演示 ITransferBankAccount：假定 Planetary Bank of Jupiter 还提供了一个当前账户。CurrentAccount 类的大多数执行代码与 SaverAccount 和 GoldAccount 的执行代码相同(这仅是为了使例子更简单，一般是不会这样的)，所以在下面的代码中，我们仅突出显示了不同的地方：

```
public class CurrentAccount : ITransferBankAccount
{
   private decimal balance;
   public void PayIn(decimal amount)
   {
      balance += amount;
   }
   public bool Withdraw(decimal amount)
   {
      if (balance >= amount)
      {
         balance -= amount;
         return true;
      }
      Console.WriteLine("Withdrawal attempt failed.");
      return false;
   }
   public decimal Balance
```

```
      {
         get
         {
            return balance;
         }
      }
      public bool TransferTo(IBankAccount destination, decimal amount)
      {
         bool result;
         if ((result = Withdraw(amount)) == true)
            destination.PayIn(amount);
         return result;
      }
      public override string ToString()
      {
         return String.Format("Jupiter Bank Current Account: Balance = {0,6:C}",
                                                              balance);
      }
   }
```

可以用下面的代码验证该类：

```
   static void Main()
   {
      IBankAccount venusAccount = new SaverAccount();
      ITransferBankAccount jupiterAccount = new CurrentAccount();
      venusAccount.PayIn(200);
      jupiterAccount.PayIn(500);
      jupiterAccount.TransferTo(venusAccount, 100);
      Console.WriteLine(venusAccount.ToString());
      Console.WriteLine(jupiterAccount.ToString());
   }
```

这段代码(CurrentAccount.cs)的结果如下所示，其中显示转账后正确的资金数：

```
C:>CurrentAccount
Venus Bank Saver: Balance = £300.00
Jupiter Bank Current Account: Balance = £400.00
```

4.5 小结

本章介绍了如何在 C#中进行继承。C#支持多接口继承和单一实现继承，还提供了许多有效的语法结构，以使代码更健壮，例如 override 关键字，它表示函数应在何时重写基类函数，new 关键字表示函数在何时隐藏基类函数，构造函数初始化器的硬性规则可以确保构造函数以健壮的方式进行交互操作。

第 5 章 数 组

如果需要使用同一类型的多个对象，就可以使用集合和数组。C#用特殊的记号声明和使用数组。Array 类在后台发挥作用，为数组中元素的排序和过滤提供了几个方法。

使用枚举器，可以迭代数组中的所有元素。

本章讨论如下内容：

- 简单数组
- 多维数组
- 锯齿数组
- Array 类
- 数组的接口
- 枚举

5.1 简单数组

如果需要使用同一类型的多个对象，就可以使用数组。数组是一种数据结构，可以包含同一类型的多个元素。

5.1.1 数组的声明

在声明数组时，应先定义数组中元素的类型，其后是一个空方括号和一个变量名。例如，下面声明了一个包含整型元素的数组：

```
int[] myArray;
```

5.1.2 数组的初始化

声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。为此，应使用 new 运算符，指定数组中元素的类型和数量

来初始化数组的变量。下面指定了数组的大小。

提示：

值类型和引用类型请参见第 3 章。

```
myArray = new int[4];
```

在声明和初始化后，变量 myArray 就引用了 4 个整型值，它们位于托管堆上，如图 5-1 所示。

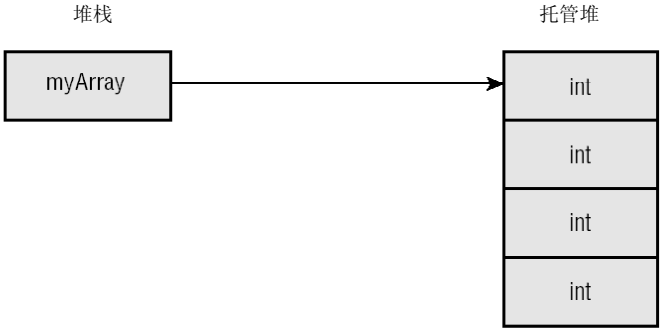

图 5-1

警告：

在指定了数组的大小后，如果不复制数组中的所有元素，就不能重新设置数组的大小。如果事先不知道数组中应包含多少个元素，就可以使用集合。集合请参见第 10 章。

除了在两个语句中声明和初始化数组之外，还可以在一个语句中声明和初始化数组：

```
int[] myArray = new int[4];
```

还可以使用数组初始化器为数组的每个元素赋值。数组初始化器只能在声明数组变量时使用，不能在声明数组之后使用。

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

如果用花括号初始化数组，还可以不指定数组的大小，因为编译器会计算出元素的个数：

```
int[] myArray = new int[] {4, 7, 11, 2};
```

使用 C#编译器还有一种更简化的形式。使用花括号可以同时声明和初始化数组，编译器生成的代码与前面的例子相同：

```
int[] myArray = {4, 7, 11, 2};
```

5.1.3 访问数组元素

数组在声明和初始化后，就可以使用索引器访问其中的元素了。数组只支持有整型参数的索引器。

提示：

在定制的类中，可以创建支持其他类型的索引器。创建定制索引器的内容请参见第 6 章。

通过索引器传送元素号，就可以访问数组。索引器总是以 0 开头，表示第一个元素。可以传送给索引器的最大值是元素个数减 1，因为索引从 0 开始。在下面的例子中，数组 myArray 用 4 个整型值声明和初始化。用索引器 0、1、2、3 就可以访问该数组中的元素。

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0];    // read first element
int v2 = myArray[1];    // read second element
myArray[3] = 44;      // change fourth element
```

警告：

如果使用错误的索引器值（不存在对应的元素），就会抛出 IndexOutOfRangeException 类型的异常。

如果不知道数组中的元素个数，则可以在 for 语句中使用 Length 属性：

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

除了使用 for 语句迭代数组中的所有元素之外，还可以使用 foreach 语句：

```
for (int val in myArray)
{
   Console.WriteLine(val);
}
```

提示：

foreach 语句利用了本章后面讨论的 IEnumerable 和 IEnumerator 接口。

5.1.4 使用引用类型

不但能声明预定义类型的数组，还可以声明定制类型的数组。下面用 Person 类来说明，这个类有两个构造函数、属性 Firstname 和 Lastname、以及 ToString()方法的一个重写：

```
public class Person
{
    public Person()
    }
    {

    public Person(string firstname, string lastname)
```

```
      "{
         this. firstname = firstname;
         this.lastname = lastname;
    }

    private string firstname;

    public string Firstname
    {
       get { return firstname;}
       set { firstname = value; }
    }

    private string lastname;

    public string Lastname
    {
       get { return lastname;}
       set { lastname = value; }
    }

    public override string ToString()
    {
       return firstname + " " + lastname;
    }
  }
```

声明一个包含两个 Person 元素的数组，与声明一个 int 数组类似：

```
Person[] myPersons = new Person[2];
```

但是必须注意，如果数组中的元素是引用类型，就必须为每个数组元素分配内存。若使用了数组中未分配内存的元素，就会抛出 NullReferenceException 类型的异常。

提示：

第 13 章介绍了错误和异常的详细内容。

使用从 0 开始的索引器，可以为数组的每个元素分配内存：

```
myPersons [0] = new Person("Ayrton", "Senna");
myPersons [1] = new Person("Michael", "Schumacher");
```

图 5-2 显示了 Person 数组中的对象在托管堆中的情况。myPersons 是一个存储在堆栈上的变量，该变量引用了存储在托管堆上的 Person 元素数组。这个数组有足够容纳两个引用的空间。数组中的每一项都引用了一个 Person 对象，而这些 Person 对象也存储在托管堆上。

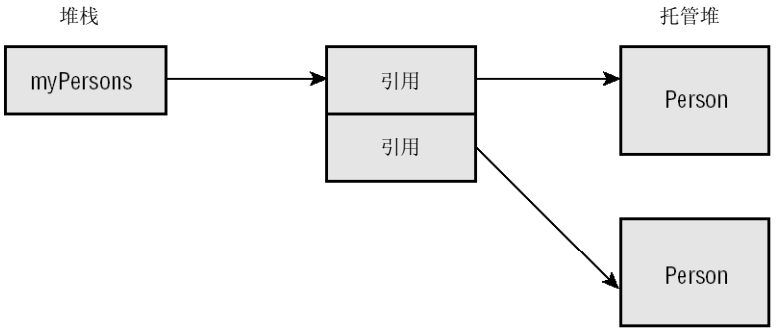

图 5-2

与 int 类型一样，也可以对定制类型使用数组初始化器：

```
Person[] myPersons = {new Person("Ayrton", "Senna"),
                      new Person("Michael", "Schumacher") };
```

5.2 多维数组

一般数组（也称为一维数组）用一个整数来索引。多维数组用两个或多个整数来索引。

图 5-3 是二维数组的数学记号，该数组有三行三列。第一行的值是 1、2 和 3，第三行的值是 7、8 和 9。

$$A = \begin{bmatrix} 1,2,3 \\ 4,5,6 \\ 7,8,9 \end{bmatrix}$$

图 5-3

在 C#中声明这个二维数组，需要在括号中加上一个逗号。数组在初始化时应指定每一维的大小（也称为阶）。接着，就可以使用两个整数作为索引器，来访问数组中的元素了：

```
int[,] twodim = new int[3, 3];
twodim[0,0] = 1;
twodim[0,1] = 2;
twodim[0,2] = 3;
twodim[1,0] = 4;
twodim[1,1] = 5;
twodim[1,2] = 6;
twodim[2,0] = 7;
twodim[2,1] = 8;
twodim[2,2] = 9;
```

提示：

数组声明之后，就不能修改其阶数了。

如果事先知道元素的值，也可以使用数组索引器来初始化二维数组。在初始化数组时，

使用一个外层的花括号，每一行用包含在外层花括号中的内层花括号来初始化。

```
int[,] twodim = {
                   {1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9},
                 };
```

提示：

使用数组初始化器时，必须初始化数组的每个元素，不能遗漏任何元素。

在中括号中使用两个逗号，就可以声明一个三维数组：

```
int[,,] threedim = {
                     { {1, 2}, {3, 4} },
                     { {5, 6}, {7, 8} },
                     { {9, 10}, {11, 12} },
                   };

Console.WriteLine(threedim[0,1,1]);
```

5.3 锯齿数组

二维数组的大小是矩形的，例如 3×3 个元素。而锯齿数组的大小设置是比较灵活的，在锯齿数组中，每一行都可以有不同的大小。

图 5-4 比较了有 3×3 个元素的二维数组和锯齿数组。图中的锯齿数组有 3 行，第一行有 2 个元素，第二行有 6 个元素，第三行有 3 个元素。

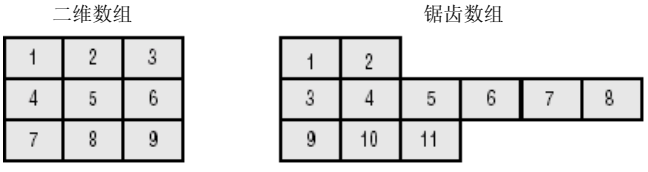

图 5-4

在声明锯齿数组时，要依次放置开闭括号。在初始化锯齿数组时，先设置该数组包含的行数。定义各行中元素个数的第二个括号设置为空，因为这类数组的每一行包含不同的元素数。之后，为每一行指定行中的元素个数：

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] {1, 2};
jagged[1] = new int[6] {3, 4, 5, 6, 7, 8};
jagged[2] = new int[3] {9, 10, 11};
```

迭代锯齿数组中所有元素的代码可以放在嵌套的 for 循环中。在外层的 for 循环中，迭代每一行，内层的 for 循环迭代一行中的每个元素：

```
for ( int row = 0; row < jagged.Length; row++)
{
   for ( int element = 0; element <jagged[row].Length; element++)
   {
      Console.WriteLine("row: {0}, element: [1], value: {2}",
          row, element, <jagged[row].[element]);
   }
}
```

该迭代显示了所有的行和每一行中的各个元素：

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
row: 2, element: 1, value: 9
row: 2, element: 2, value: 10
row: 2, element: 3, value: 11
```

5.4　Array 类

用括号声明数组是 C#中使用 Array 类的记号。在后台使用 C#语法，会创建一个派生于抽象基类 Array 的新类。这样，就可以使用 Array 类为每个 C#数组定义的方法和属性了。例如，前面就使用了 Length 属性，还使用 foreach 语句迭代数组。其实这是使用了 Array 类中的 GetEnumerator()方法。

5.4.1　属性

Array 类包含的如下属性可以用于每个数组实例。本章后面还将讨论其他更多的属性。

表　5-1

属　　性	说　　明
Length	Length 属性返回数组中的元素个数。如果是一个多维数组，该属性会返回所有阶的元素个数。如果需要确定一维中的元素个数，则可以使用 GetLength()方法
LongLength	Length 属性返回 int 值，而 LongLength 属性返回 long 值。如果数组包含的元素个数超出了 32 位 int 值的取值范围，就需要使用 LongLength 属性，来获得元素个数
Rank	使用 Rank 属性可以获得数组的维数

5.4.2 创建数组

Array 类是一个抽象类，所以不能使用构造函数来创建数组。但除了可以使用 C#语法创建数组实例之外，还可以使用静态方法 CreateInstance()创建数组。如果事先不知道元素的类型，就可以使用该静态方法，因为类型可以作为 Type 对象传送给 CreateInstance()方法。

下面的例子说明了如何创建类型为 int、大小为 5 的数组。CreateInstance()方法的第一个参数应是元素的类型，第二个参数定义数组的大小。可以用 SetValue()方法设置值，用 GetValue()方法读取值：

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
   intArray1.SetValue(33, i);
}

for (int i = 0; i < 5; i++)
{
   Console.WriteLine(intArray1.GetValue(i));
}
```

还可以将已创建的数组强制转换成声明为 int[]的数组：

```
int[] intArray2 = (int[])intArray1;
```

CreateInstance()方法有许多重载版本，可以创建多维数组和不基于 0 的数组。下面的例子就创建了一个包含 2×3 个元素的二维数组。第一维基于 1，第二维基于 0：

```
int[] lengths = {2, 3};
int[] lowerBounds = {1, 10};
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

SetValue()方法设置数组的元素，其参数是每一维的索引：

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaidi"), 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Ralf", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

尽管数组不是基于 0 的，但可以用一般的 C#记号将它赋予一个变量。只需注意不要超出边界即可：

```
Person[,] racers2 = (Person[,]) racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

5.4.3　复制数组

数组是引用类型，所以将一个数组变量赋予另一个数组变量，就会得到两个指向同一数组的变量。而复制数组，会使数组实现 ICloneable 接口。这个接口定义的 Clone()方法会创建数组的浅副本。

如果数组的元素是值类型，就会复制所有的值，如图 5-5 所示：

```
int intArray1 = {1, 2};
int intArray2 = (int[])intArray1.Clone();
```

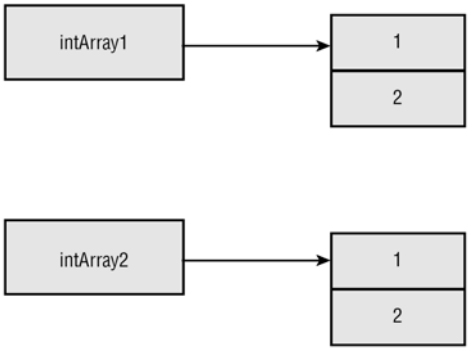

图　5-5

如果数组包含引用类型，则不复制元素，而只复制引用。图 5-6 显示了变量 beatles 和 beatlesClone，其中 beatlesClone 是通过在 beatles 上调用 Clone()方法来创建的。beatles 和 beatlesClone 引用的 Person 对象是相同的。如果修改 beatlesClone 中一个元素的属性，就会改变 beatles 中的对应对象。

```
Person[] beatles = {
              new Person("John", "Lennon"),
              new Person("Paul", "McCartney"),
            };
Person[] beatlesClone = (Person[])beatles.Clone();
```

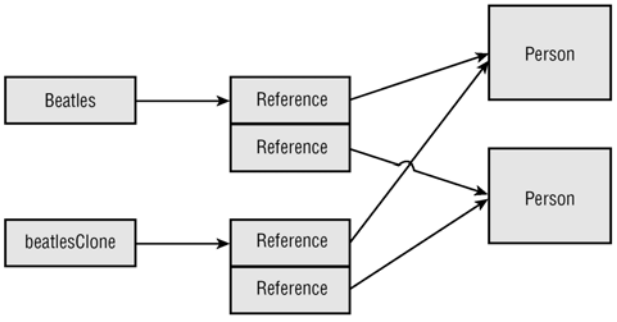

图　5-6

除了使用 Clone()方法之外，还可以使用 Array.Copy()方法创建浅副本。但 Clone()方法

和 Copy()方法有一个重要区别：Clone()方法会创建一个新数组，而 Copy()方法只是传送了阶数相同、有足够元素空间的已有数组。

提示：

如果需要包含引用类型的数组的深副本，就必须迭代数组，创建新对象。

5.4.4 排序

Array 类实现了对数组中元素的冒泡排序。Sort()方法需要数组中的元素实现 IComparable 接口。简单类型，如 System.String 和 System.Int32 实现了 IComparable 接口，所以可以对包含这些类型的元素排序。

在示例程序中，数组 name 包含 string 类型的元素，这个数组是可以排序的。

```
string names = {
             "Christina Aguillera",
             "Shakira",
             "Beyonce",
             "Gwen Stefani"
           };

Array.Sort(names);

foreach (string name in names)
{
   Console.WriteLine(name);
}
```

该应用程序的输出是排好序的数组：

```
Beyonce
Christina Aguillera
Gwen Stefani
Shakira
```

如果对数组使用定制的类，就必须实现 IComparable 接口。这个接口只定义了一个方法 CompareTo()，如果要比较的对象相等，该方法就返回 0。如果实例应排在参数对象的前面，该方法就返回小于 0 的值。如果实例应排在参数对象的后面，该方法就返回大于 0 的值。

修改 Person 类，使之执行 IComparable 接口。对 lastname 的值进行比较。lastname 是 string 类型，而 String 类已经实现了 IComparable 接口，所以可以使用 String 类中 CompareTo()方法的实现代码：

```
public class Person : IComparable
{
  public int CompareTo(object obj)
  {
    Person other = obj as Person;
    return this.lastname. CompareTo(other.lastname);
```

```
    }
    //…
```

现在可以按照姓氏对 Person 对象数组排序了：

```
Person[] persons = {
    new Person("Emerson", "Fittipaldi"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Michael", "Schumacher"),
};

Array.Sotr(persons);
foreach (Person p in persons)
{
  Console.WriteLine(p);
}
```

使用 Person 类的排序功能，会得到按姓氏排序的姓名：

```
Emerson Fittipaldi
Niki Lauda
Michael Schumacher
Ayrton Senna
```

如果 Person 对象的排序方式与上述不同，或者不能修改在数组中用作元素的类，就可以执行 IComparer 接口。这个接口定义了方法 Compare()。IComparable 接口必须由要比较的类来执行，而 IComparer 接口独立于要比较的类。这就是 Compare()方法定义了两个要比较的变元的原因。其返回值与 IComparable 接口的 CompareTo()方法类似。

类 PersonComparer 实现了 IComparer 接口，可以按照 firname 或 lastname 对 Person 对象排序。枚举 PersonCompareType 定义了与 PersonComparer 相当的排序选项：Firname 和 Lastname。排序的方式由类 PersonComparer 的构造函数定义，在该构造函数中设置了一个 PersonCompareType 值。Compare()方法用一个 switch 语句指定是按 firname 还是 lastname 排序。

```
public class PersonComparer : IComparer
{
  public enum PersonCompareType
  {
    Firname,
    Lastname
  }

  private PersonCompareType compareType;

  public PersonComparer(PersonCompareType compareType)
  {
     this. compareType = compareType;
    }
```

```
    public int Compare(object x, object y)
    {
       Person p1 = x as Person;
       Person p1 = y as Person;
       Switch (compareType)
       {
          case PersonCompareType.Firstname:
            return p1. Firstname. CompareTo(p2. Firstname);
          case PersonCompareType.Lastname:
            return p1. Lastname. CompareTo(p2. Lastname);
          default:
            throw new ArgumentException("unexpexted compare type")
        }
    }
}
```

现在，可以将一个 PersonComparer 对象传送给 Array.Sort()方法的第二个变元。下面是按名字对 persons 数组排序：

```
Array.Sort(persons,
new PersonComparer(PersonComparer. PersonCompareType.Firstname));
foreach (Person p in persons)
{
  Console.WriteLine(p);
}
```

persons 数组现在按名字排序：

```
Ayrton Senna
Emerson Fittipaldi
Michael Schumacher
Niki Lauda
```

提示：

Array 类还提供了 Sort 方法，它需要将一个委托作为变元。第 7 章将介绍如何使用委托。

5.5 数组和集合接口

Array 类实现了 IEumerable、ICollection 和 IList 接口，以访问和枚举数组中的元素。由于用定制数组创建的类派生于 Array 抽象类，所以能使用通过数组变量执行的接口中的方法和属性。

5.5.1 IEumerable 接口

IEumerable 是由 foreach 语句用于迭代数组的接口。这是一个非常特殊的特性，在下一节中讨论。

5.5.2 ICollection 接口

ICollection 接口派生于 IEumerable 接口，并添加了如表 5-2 所示的属性和方法。这个接口主要用于确定集合中的元素个数，或用于同步。

表 5-2

ICollection 接口的属性和方法	说 明
Count	Count 属性可确定集合中的元素个数，它返回的值与 Length 属性相同
IsSynchronized SyncRoot	IsSynchronized 属性确定集合是否是线程安全的。对于数组，这个属性总是返回 false。对于同步访问，SyncRoot 属性可以用于线程安全的访问。这两个属性都是在 ICollection 接口中定义的。第 18 章介绍了线程和同步，探讨了如何用集合实现线程安全性
CopyTo()	利用 CopyTo()方法可以将数组的元素复制到现有的数组中。它类似于静态方法 Array.Copy()

5.5.3 IList 接口

IList 接口派生于 ICollection 接口，并添加了下面的属性和方法。Array 类实现 IList 接口的主要原因是，IList 接口定义了 Item 属性，以使用索引器访问元素。IList 接口的许多其他成员是通过 Array 类抛出 NotSupportedException 异常实现的，因为这些不应用于数组。IList 接口的所有属性和方法如表 5-3 所示。

表 5-3

IList 接 口	说 明
Add()	Add()方法用于在集合中添加元素。对于数组，该方法会抛出 NotSupportedException 异常
Clear()	Clear()方法可清除数组中的所有元素。值类型设置为 0，引用类型设置为 null
Contains()	Contains()方法可以确定某个元素是否在数组中。其返回值是 true 或 false。这个方法会对数组中的所有元素进行线性搜索，直到找到所需元素为止
IndexOf()	IndexOf()方法与 Contains()方法类似，也是对数组中的所有元素进行线性搜索。不同的是，IndexOf()方法会返回所找到的第一个元素的索引
Insert() Remove() RemoveAt()	对于集合，Insert()方法用于插入元素，Remove()和 RemoveAt()可删除元素。对于数组，这些方法都抛出 NotSupportedException 异常
IsFixedSize	数组的大小总是固定的，所以这个属性总是返回 true
IsReadOnly	数组总是可以读/写的，所以这个属性返回 false。第 10 章将介绍如何从数组中创建只读属性
Item	Item 属性可以用整型索引访问数组

5.6 枚举

在 foreach 语句中使用枚举，可以迭代集合中的元素，且无需知道集合中的元素个数。图 5-7 显示了调用 foreach 方法的客户机和集合之间的关系。数组或集合执行带 GetEumerator()方法的 IEumerable 接口。GetEumerator()方法返回一个执行 IEumerable 接口的枚举。接着，foreach 语句就可以使用 IEumerable 接口迭代集合了。

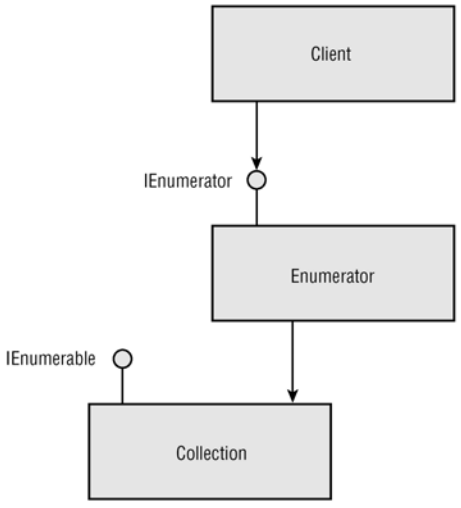

图 5-7

提示：

GetEnumerator()方法用 IEnumerable 接口定义。foreach 语句并不真的需要在集合类中执行这个接口。有一个名为 GetEnumerator()的方法，返回实现了 IEnumerator 接口的对象就足够了。

5.6.1 IEnumerator 接口

foreach 语句使用 IEnumerator 接口的方法和属性，迭代集合中的所有元素。这个接口中的属性和方法如表 5-4 所示。

表 5-4

IEnumerator 接口的方法和属性	说 明
MoveNext()	MoveNext()方法移动到集合的下一个元素上，如果有这个元素，该方法就返回 true。如果集合不再有更多的元素，该方法就返回 false
Current	属性 Current 返回光标所在的元素
Reset()	Reset()方法将光标重新定位于集合的开头。许多枚举会抛出 NotSupportedException 异常

5.6.2 foreach 语句

C#的 foreach 语句不会解析为 IL 代码中的 foreach 语句。C#编译器会把 foreach 语句转换为 IEnumerable 接口的方法和属性。下面是一个简单的 foreach 语句，它迭代 persons 数组中的所有元素，并逐个显示它们：

```
foreach (Person p in persons)
{
  Console.WriteLine(p);
}
```

foreach 语句会解析为下面的代码段。首先，调用 GetEnumerator()方法，获得数组的一个枚举。在 while 循环中——只要 MoveNext()返回 true——用 Current 属性访问数组中的元素：

```
IEnumerator enumerator = persons. GetEnumerator();
while (enumerator.MoveNext())
{
   Person p = (Person) enumerator.Current;
   Console.WriteLine(p);
}
```

5.6.3 yield 语句

C# 1.0 使用 foreach 语句可以轻松地迭代集合。在 C# 1.0 中，创建枚举器仍需要做大量的工作。C# 2.0 添加了 yield 语句，以便于创建枚举器。

yield return 语句返回集合的一个元素，并移动到下一个元素上。yield break 可停止迭代。

下面的例子是用 yield return 语句实现一个简单集合的代码。类 HelloCollection 包含 GetEnumerator()方法。该方法的实现代码包含两个 yield return 语句，它们分别返回字符串 Hello 和 World。

```
using System;
using System.Collection;

namespace Wrox.ProCAharp.Arrays
{
  public class HelloCollection
  {
    public IEumerator GetEumerator()
    {
    yield return "Hello";
    yield return "World";
    }
  }
}
```

警告：

包含 yield 语句的方法或属性也称为迭代块。迭代块必须声明为返回 IEnumerator 或 IEnumerable 接口。这个块可以包含多个 yield return 语句或 yield break 语句，但不能包含 return 语句。

现在可以用 foreach 语句迭代集合了：

```
  public class Program
  {
   HelloCollection helloCollection = new HelloCollection();
   foreach (string s in helloCollection)
    {
      Console.WriteLine(s);
    }
  }
}
```

使用迭代块，编译器会生成一个 yield 类型，其中包含一个状态机，如下面的代码所示。yield 类型执行 IEnumerator 和 IDisposable 接口的属性和方法。在下面的例子中，可以把 yield 类型看作内部类 Enumerator。外部类的 GetEnumerator()方法实例化并返回一个新的 yield 类型。在 yield 类型中，变量 state 定义了迭代的当前位置，每次调用 MoveNext()时，当前位置都会改变。MoveNext()封装了迭代块的代码，设置了 current 变量的值，使 Current 属性根据位置返回一个对象。

```
public class HelloCollection
{
public IEnumerator GetEnumerator()
  {
    Enumerator enumerator = new Enumerator();
    return enumerator;
}

public class Enumerator : IEnumerator, IDisposable
{
  private int state;
  private object current;

  public Enumerator(int state)
  {
    this.state = state;
}
bool System.Collections.IEnumerator.MoveNext()
{
  switch (state)
  {
     case 0:
       current = "Hello";
       state = 1;
```

```
                return true;
              case 1:
                current = "World";
                state = 2;
                return true;
              case 2:
                break;
            }

            return false;
          }

          void System.Collections.IEnumerator.Reset()
          {
            throw new NotSupportedException();
          }

          object System.Collections.IEnumerator.Current
          {
            get
            {
            return current;
            }
          }

          void IDisposable.Dispose()
          {
          }
      }
  }
```

现在使用 yield return 语句，很容易实现允许以不同方式迭代集合的类。类 MusicTitles 可以用默认方式通过 GetEnumerator()方法迭代标题，用 Reverse()方法逆序迭代标题，用 Subset()方法搜索子集：

```
  public class MusicTitles
  {
    string[] names = {
          "Tubular Bells", "Hergest Ridge",
          "Ommadawn", "Platinum");
    public IEnumerator GetEnumerator()
    {
       for (int i = 0; i < 4; i++)
       {
       yield return names[i];
        }
    }

    public IEnumerable Reverse()
    {
       for (int i = 3; i >= 0; i－)
       {
```

```
            yield return names[i];
        }
    }

    public IEnumerable Subset( int index, int length)
    {
        for (int i = index; i < index + length; i++)
        {
            yield return names[i];
        }
    }
}
```

迭代字符串数组的客户代码先使用 GetEnumerator()方法，该方法不必在代码中编写，因为这是默认使用的方法。然后逆序迭代标题，最后将索引和要迭代的元素个数传送给 Subset()方法，来迭代子集：

```
MusicTitles titles = new MusicTitles();
foreach(string title in titles)
{
  ConsoleWriteLine(title);
}
ConsoleWriteLine();

ConsoleWriteLine("reverse");
foreach(string title in titles.Reverse())
{
  ConsoleWriteLine(title);
}
ConsoleWriteLine();

ConsoleWriteLine("subset");
foreach(string title in titles.Subset(2, 2))
{
  ConsoleWriteLine(title);
}
```

使用 yield 语句还可以完成更复杂的任务，例如从 yield return 中返回枚举器。

在 TicTacToe 游戏中有 9 个域，玩家轮流在这些域中放置“十”字或一个圆。这些移动操作由 GameMoves 类模拟。方法 Cross()和 Circle()是创建迭代类型的迭代块。变量 cross 和 circle 在 GameMoves 类的构造函数中设置为 Cross()和 Circle()方法。这些域不设置为调用的方法，而是设置为用迭代块定义的迭代类型。在 Cross()迭代块中，将移动操作的信息写到控制台上，并递增移动次数。如果移动次数大于 9，就用 yield break 停止迭代；否则，就在每次迭代中返回 yield 类型 cross 的枚举对象。Circle()迭代块非常类似于 Cross()迭代块，只是它在每次迭代中返回 circle 迭代类型。

```
public calss GameMoves
{
  private IEnumerator cross;
```

```
    private IEnumerator circle;

    public GameMoves()
    {
      cross = Cross();
      circle = Circle();
  }

    private int move = 0;

    public IEnumerator Cross()
    {
       while (true)
       {
         Console.WriteLine("Cross, move {0}", move);
         move++;
         if (move > 9)
           yield break;
         yield return circle;
       }
    }

    public IEnumerator Circle()
    {
       while (true)
       {
         Console.WriteLine("Circle, move {0}", move);
         move++;
         if (move > 9)
           yield break;
         yield return cross;
       }
    }
  }
```

在客户程序中，可以以如下方式使用 GameMoves 类。将枚举器设置为由 game.Cross() 返回的枚举器类型，以设置第一次移动。enumerator.MoveNext 调用以迭代块定义的一次迭代，返回另一个枚举器。返回的值可以用 Current 属性访问，并设置为 enumerator 变量，用于下一次循环：

```
  GameMoves game = new GameMoves();
  IEnumerator enumerator = game.Cross();
  while (enumerator.MoveNext())
  {
    enumerator = (IEnumerator) enumerator.Current;
  }
```

这个程序的输出会显示交替移动的情况，直到最后一次移动：

```
Cross, move 0
```

```
Circle, move 1
Cross, move 2
Circle, move 3
Cross, move 4
Circle, move 5
Cross, move 6
Circle, move 7
Cross, move 8
```

5.7 小结

本章介绍了创建和使用简单数组、多维数组和锯齿数组的 C#记号。Array 类在 C#数组的后台使用，这样就可以用数组变量调用这个类的属性和方法。

我们还探讨了如何使用 IComparable 和 IComparer 接口给数组中的元素排序，描述了用 Array 类执行的 IEnumerable、ICollection 和 IList 接口的特性，最后论述了 C# 2.0 中新增的 yield 语句的优点。

数组和相关主题的更多信息，可参阅如下章节：第 6 章介绍了运算符和强制类型转换，其中探讨了定制索引器的创建。第 7 章讨论了委托和事件。Array 类的一些方法将委托用作参数。第 10 章介绍了本章探讨的集合类。集合类有较灵活的尺寸，第 10 章还介绍了其他容器，如字典和链表。

第 6 章

运算符和类型强制转换

前几章介绍了使用 C#编写程序所需要的大部分知识。本章将首先讨论基本语言元素，接着论述 C#语言的扩展功能。本章的主要内容如下：

- C#中的可用运算符
- 处理引用类型和值类型时相等的含义
- 基本数据类型之间的数据转换
- 使用装箱技术把值类型转换为引用类型
- 通过强制转换技术在引用类型之间转换
- 重载标准的运算符，以支持对定制类型的操作
- 给定制类型添加强制转换运算符，以支持无缝的数据类型转换

6.1 运算符

C 和 C++开发人员应很熟悉大多数 C#运算符，这里为新程序员和 VB 开发人员介绍最重要的运算符，并介绍 C#中的一些新变化。

C#支持表 6-1 所示的运算符，其中有 4 个运算符(sizeof、*、 ->、&)只能用于不安全的代码(这些代码绕过了 C#类型安全性的检查)，这些不安全的代码见第 11 章的讨论。

表 6-1

类　别	运　算　符
算术运算符	+ - * / %
逻辑运算符	& | ^ ~ && || !
字符串连接运算符	+
增量和减量运算符	++ --
移位运算符	<< >>
比较运算符	== != < > <= >=

(续表)

类 别	运 算 符
赋值运算符	= += -= *= /= %= &= |= ^= <<= >>=
成员访问运算符(用于对象和结构)	.
索引运算符(用于数组和索引器)	[]
数据类型转换运算符	()
条件运算符 (三元运算符)	?:
委托连接和删除运算符(见第 7 章)	+ -
对象创建运算符	new
类型信息运算符	sizeof (只用于不安全的代码) is typeof as
溢出异常控制运算符	checked unchecked
间接寻址运算符	* -> & (只用于不安全代码) []
命名空间别名限定符(见第 2 章)	::
空接合运算符	??

使用 C#运算符的一个最大缺点是，与 C 风格的语言一样，赋值(=)和比较(==)运算使用不同的运算符。例如，下述语句表示“x 等于 3”：

```
x = 3;
```

如果要比较 x 和另一个值，就需要使用两个等号(==)：

```
if (x == 3)
{

}
```

C#非常严格的类型安全规则防止出现常见的 C#错误，也就是在逻辑语句中使用赋值运算符代替比较运算符。在 C#中，下述语句会产生一个编译错误：

```
if (x = 3)
{

}
```

习惯使用宏字符&来连接字符串的 VB 程序员必须改变这个习惯。在 C#中，使用加号+连接字符串，而&表示两个不同整数值的按位 AND 运算。| 则在两个整数之间执行按位 OR 运算。VB 程序员可能还没有使用过%(取模)运算符，它返回除运算的余数，例如，如果 x 等于 7，则 x % 5 会返回 2。

在 C#中很少会用到指针，因此也很少用到间接寻址运算符(->)。使用它们的唯一场合是在不安全的代码块中，因为只有在此 C#才允许使用指针。指针和不安全的代码见第 11 章。

6.1.1　运算符的简化操作

表 6-2 列出了 C#中的全部简化赋值运算符。

表　6-2

运算符的简化操作	等　价　于
x++, ++x	x = x + 1
x– –,– –x	x = x – 1
x+= y	x = x + y
x – = y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y

为什么用两个例子来说明++增量和– –减量运算符？把运算符放在表达式的前面称为前置，把运算符放在表达式的后面称为后置。它们的执行方式有所不同。

增量或减量运算符可以作用于整个表达式，也可以作用于表达式的内部。当 x++和++x 单独占一行时，它们的作用是相同的，对应于语句 x = x + 1。但当它们用于表达式内部时，把运算符放在前面(++x)会在计算表达式之前递增 x，换言之，递增了 x 后，在表达式中使用新值进行计算。而把运算符放在后面(x++)会在计算表达式之后递增 x—— 使用 x 的原值计算表达式。下面的例子使用++增量运算符说明了它们的区别：

```
int x = 5;
if (++x == 6)
{
   Console.WriteLine("This will execute");
}
if (x++ == 7)
{
   Console.WriteLine("This won't");
}
```

第一个 if 条件得到 true，因为在计算表达式之前，x 从 5 递增为 6。第二个 if 语句中的条件为 false，因为在计算完整个表达式(x=6)后，x 才递增为 7。

前置运算符– –x 和后置运算符 x– –与此类似，但它们是递减，而不是递增。

其他简化运算符，如+= 和 – =需要两个操作数，用于执行算术、逻辑和按位运算，改

变第一个操作数的值。例如，下面两行代码是等价的：

```
x += 5;
x = x + 5;
```

6.1.2 三元运算符

三元运算符(?:)是 if...else 结构的简化形式。其名称的出处是它带有三个操作数。它可以计算一个条件，如果条件为真，就返回一个值；如果条件为假，则返回另一个值。其语法如下：

```
condition ? true_value : false_value
```

其中 condition 是要计算的 Boolean 型表达式，true_value 是 condition 为 true 时返回的值，false_value 是 condition 为 false 时返回的值。

恰当地使用三元运算符，可以使程序非常简洁。它特别适合于给被调用的函数提供两个参数中的一个。使用它可以把 Boolean 值转换为字符串值 true 或 false。它也很适合于显示正确的单数形式或复数形式，例如：

```
int x = 1;
string s = x.ToString() + " ";
s += (x == 1 ? "man" : "men");
Console.WriteLine(s);
```

如果 x 等于 1，这段代码就显示 1 man，如果 x 等于其他数，就显示其正确的复数形式。但要注意，如果结果需要用在不同的语言中，就必须编写更复杂的例程，以考虑到不同语言的不同语法。

6.1.3 checked 和 unchecked 运算符

考虑下面的代码：

```
byte b = 255;
b++;
Console.WriteLine(b.ToString());
```

byte 数据类型只能包含 0~255 的数，所以递增 b 的值会导致溢出。CLR 如何处理这个溢出取决于许多方面，包括编译器选项，所以只要有未预料到的溢出风险，就需要用某种方式确保得到我们希望的结果。

为此，C#提供了 checked 和 unchecked 运算符。如果把一个代码块标记为 checked，CLR 就会执行溢出检查，如果发生溢出，就抛出异常。如果修改代码，使之包含 checked 运算符：

```
byte b = 255;
checked
{
   b++;
```

```
    }
    Console.WriteLine(b.ToString());
```

运行这段代码，就会得到一个错误信息：

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow.
   at Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)
```

注意：
用/checked 编译器选项进行编译，就可以检查程序中所有未标记代码中的溢出。

如果要禁止溢出检查，可以把代码标记为 unchecked：

```
byte b = 255;
unchecked
{
   b++;
}
Console.WriteLine(b.ToString());
```

在本例中，不会抛出异常，但会丢失数据——因为 byte 数据类型不能包含 256，溢出的位会被丢掉，所以 b 变量得到的值是 0。

注意，unchecked 是默认值。只有在需要把几个未检查的代码行放在一个明确标记为 checked 的大代码块中，才需要显式使用 unchecked 关键字。

6.1.4　is 运算符

is 运算符可以检查对象是否与特定的类型兼容。例如，要检查变量是否与 object 类型兼容：

注意：
“兼容”表示对象是该类型，或者派生于该类型。

```
int i = 10;
if (i is object)
{
   Console.WriteLine("i is an object");
}
```

int 和其他 C#数据类型一样，也从 object 继承而来；表达式 i is object 将得到 true，并显示相应的信息。

6.1.5　as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，as 运算符就会返回值 null。如下面的代码所示，如果 object 引用不指向 string 实例，把 object 引用转换为 string 就会返回 null：

```
object o1 = "Some String";
object o2 = 5;

string s1 = o1 as string;     //s1 = "Some String"
string s2 = o2 as string;    //s1 = null
```

as 运算符允许在一步中进行安全的类型转换，不需要先使用 is 运算符测试类型，再执行转换。

6.1.6 sizeof 运算符

使用 sizeof 运算符可以确定堆栈中值类型需要的长度(单位是字节)：

```
unsafe
{
   Console.WriteLine(sizeof(int));
}
```

其结果是显示数字 4，因为 int 有 4 个字节。

注意，只能在不安全的代码中使用 sizeof 运算符。第 11 章将详细论述不安全的代码。

6.1.7 typeof 运算符

typeof 运算符返回一个表示特定类型的 System.Type 对象。例如，typeof(string)返回表示 System.String 类型的 Type 对象。在使用反射技术动态查找对象的信息时，这个运算符是很有效的。第 12 章将介绍反射。

6.1.8 可空类型和运算符

对于布尔类型，可以给它指定 true 或 false 值。但是，要把该类型的值定义为 undefined，该怎么办？此时使用可空类型可以给应用程序提供一个独特的值。如果在程序中使用可空类型，就必须考虑 null 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 null，其结果就是 null。例如：

```
int? a = null;

int? b = a + 4;      // b = null
int? c = a * 5;      // c = null
```

但是在比较可空类型时，只要有一个操作数是 null，比较的结果就是 false。即不能因为一个条件是 false，就认为该条件的对立面是 true，这在使用非可空类型的程序中很常见。例如：

```
int? a = null;
int? b = -5;

if (a >= b)
```

```
    Console.WriteLine("a >= b");
else
    Console.WriteLine("a < b");
```

注意：

null 值的可能性表示，不能随意比较表达式中的可空类型和非可空类型，详见本章后面的内容。

6.1.9　空接合运算符

空接合运算符(??)提供了一种快捷方式，可以在处理可空类型和引用类型时表示 Null 值。这个运算符放在两个操作数之间，第一个操作数必须是一个可空类型或引用类型，第二个操作数必须与第一个操作数的类型相同，或者可以隐含地转换为第一个操作数的类型。空接合运算符的计算如下：如果第一个操作数不是 null，则整个表达式就等于第一个操作数的值。但如果第一个操作数是 null，则整个表达式就等于第二个操作数的值。例如：

```
int? a = null;
int b;

b = a ?? 10;    // b has the value 10
a = 3;
b = a ?? 10;    // b has the value 3
```

如果第二个操作数不能隐含地转换为第一个操作数的类型，就生成一个编译错误。

6.1.10　运算符的优先级

表 6-3 显示了 C#运算符的优先级。表顶部的运算符有最高的优先级(即在包含多个运算符的表达式中，最先计算该运算符)：

表　6-3

组	运　算　符
初级运算符	() . [] x++ x－－ new typeof sizeof checked unchecked
一元运算符	+ － ! ~ ++x －－x 和数据类型转换
乘/除运算符	* / %
加/减运算符	+ –
移位运算符	<< >>
关系运算符	< > <= >= is as
比较运算符	== !=
按位 AND 运算符	&
按位 XOR 运算符	|
按位 OR 运算符	^

(续表)

组	运 算 符
布尔 AND 运算符	&&
布尔 OR 运算符	||
三元运算符	?:
赋值运算符	= += -= *= /= %= &= |= ^= <<= >>= >>>=

注意：

在复杂的表达式中，应避免利用运算符优先级来生成正确的结果。使用括号指定运算符的执行顺序，可以使代码更整洁，避免出现潜在的冲突。

6.2 类型的安全性

第 1 章提到中间语言(IL)可以对其代码强制加上强类型安全性。强类型支持.NET 提供的许多服务，包括安全性和语言的交互性。因为 C#这种语言会编译为 IL，所以 C#也是强类型的。这说明数据类型并不总是可互换的。本节将介绍基本类型之间的转换。

注意：

C#还支持在不同引用类型之间的转换，允许指定自己创建的数据类型如何与其他类型进行相互转换。这些论题将在本章后面讨论。

泛型是 C# 2.0 中的一个新特性，它可以避免对一些常见的情形进行类型转换，泛型详见第 9 章。

6.2.1 类型转换

我们常常需要把数据从一种类型转换为另一种类型。考虑下面的代码：

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

在编译这些代码时，会产生一个错误：

```
Cannot implicitly convert type 'int' to 'byte' (不能把 int 类型隐式地转换为 byte 类型)。
```

问题是，我们把两个 byte 型数据加在一起时，应返回 int 型结果，而不是另一个 byte。这是因为 byte 包含的数据只能为 8 位，所以把两个 byte 型数据加在一起，很容易得到不能存储在 byte 变量中的值。如果要把结果存储在一个 byte 变量中，就必须把它转换回 byte。C#有两种转换方式：隐式转换方式和显式转换方式。

1. 隐式转换方式

只要能保证值不会发生任何变化，类型转换就可以自动进行。这就是前面代码失败的原因：试图从 int 转换为 byte，而潜在地丢失了 3 个字节的数据。编译器不会告诉我们该怎么做，除非我们明确告诉它这就是我们希望的！如果在 long 型变量中存储结果，而不是 byte 型变量中，就不会有问题了：

```
byte value1 = 10;
byte value2 = 23;
long total;             // this will compile fine
total = value1 + value2;
Console.WriteLine(total);
```

这是因为 long 类型变量包含的数据字节比 byte 类型多，所以数据没有丢失的危险。在这些情况下，编译器会很顺利地转换，我们也不需要显式提出要求。

表 6-4 介绍了 C#支持的隐式类型转换。

表　6-4

源　类　型	目 的 类 型
sbyte	short、int、long、float、double、decimal
byte	short、ushort、int、uint、long、ulong、float、double、decimal
short	int、long、float、double、decimal
ushort	int、uint、long、ulong、float、double、decimal
int	long、float、double、decimal
uint	long、ulong、float、double、decimal
long、ulong	float、double、decimal
float	double
char	ushort、int、uint、long、ulong、float、double、decimal

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

在隐式转换值类型时，可空类型需要额外考虑：

- 可空类型隐式转换为其他可空类型，应遵循表 6-4 中非可空类型的转换规则。即 int?隐式转换为 long?、float?、double?和 decimal?。

- 非可空类型隐式转换为可空类型也遵循表 6-4 中的转换规则，即 int 隐式转换为 long?、float?、double?和 decimal?。
- 可空类型不能隐式转换为非可空类型，此时必须进行显式转换，如下一节所述。这是因为可空类型的值可以是 null，但非可空类型不能表示这个值。

2. 显式转换方式

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

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

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

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

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

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

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

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

```
-1294967296
```

最好假定显式数据转换不会给出希望的结果。如前所述，C#提供了一个 checked 运算符，使用它可以测试操作是否会产生算术溢出。使用这个运算符可以检查数据类型转换是否安全，如果不安全，就会让运行库抛出一个溢出异常：

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

记住，所有的显式数据类型转换都可能不安全，在应用程序中应包含这样的代码，处

理可能失败的数据类型转换。第 13 章将使用 try 和 catch 语句引入结构化异常处理。

使用数据类型转换可以把大多数数据从一种基本类型转换为另一种基本类型。例如：给 price 加上 0.5，再把结果转换为 int：

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

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

下面的例子说明了把一个无符号的整数转换为 char 型时，会发生的情况：

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

结果是 ASCII 数为 43 的字符，即+号。可以尝试数字类型之间的任何转换(包括 char)，这种转换是成功的，例如把 decimal 转换为 char，或把 char 转换为 decimal。

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

```
struct ItemDetails
{
   public string Description;
   public int ApproxPrice;
}

//...

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

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

要把一个可空类型转换为非可空类型，或转换为另一个可空类型，但可能会丢失数据，就必须使用显式转换。重要的是，在底层基本类型相同的元素之间进行转换时，就一定要使用显式转换，例如 int?转换为 int，或 float?转换为 float。这是因为可空类型的值可以是 null，非可空类型不能表示这个值。只要可以在两个非可空类型之间进行显式转换，对应可空类型之间的显式转换就可以进行。但如果从可空类型转换为非可空类型，且变量的值是 null，就会抛出 InvalidOperationException。例如：

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

使用显式的数据类型转换方式，并小心使用这种技术，就可以把简单值类型的任何实

例转换为另一种类型。但在进行显式的类型转换时有一些限制，例如值类型，只能在数字、char 类型和 enum 类型之间转换。不能直接把 Boolean 数据类型转换为其他类型，也不能把其他类型转换为 Boolean 数据类型。

如果需要在数字和字符串之间转换，.NET 类库提供了一些方法。Object 类有一个 ToString()方法，该方法在所有的.NET 预定义类型中都进行了重写，返回对象的字符串表示：

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

同样，如果需要分析一个字符串，获得一个数字或 Boolean 值，就可以使用所有预定义值类型都支持的 Parse 方法：

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

注意，如果不能转换字符串(例如要把字符串 Hello 转换为一个整数)，Parse 方法就会注册一个错误，抛出一个异常。第 13 章将介绍异常。

6.2.2 装箱和拆箱

第 2 章介绍了所有类型，包括简单的预定义类型，例如 int 和 char，以及复杂类型，例如从 Object 类型中派生的类和结构。下面可以像处理对象那样处理字面值：

```
string s = 10.ToString();
```

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

C#的实现方式是通过一个有点魔术性的方式，即装箱(boxing)。装箱和拆箱(unboxing)可以把值类型转换为引用类型，或把引用类型转换为值类型。这已经在数据类型转换一节中介绍过了，即把值转换为 object 类型。装箱用于描述把一个值类型转换为引用类型。运行库会为堆上的对象创建一个临时的引用类型“box”。

该转换是隐式进行的，如上面的例子所述。还可以手工进行转换：

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

拆箱用于描述相反的过程，即以前装箱的值类型转换回引用类型。这里使用术语“cast”，是因为这种数据类型转换是显式进行的。其语法类似于前面的显式类型转换：

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

只能把以前装箱的变量再转换为值类型。当 o 不是装箱后的 int 型时，如果执行上面的代码，就会在运行期间抛出一个异常。

这里有一个警告。在拆箱时，必须非常小心，确保得到的值变量有足够的空间存储拆箱的值中的所有字节。例如，C#的 int 只有 32 位，所以把 long 值(64 位)拆箱为 int 时，会产生一个 InvalidCastException 异常：

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

6.3　对象的相等比较

在讨论了运算符，并简要介绍了相等运算符后，就应考虑在处理类和结构的实例时，“相等”意味着什么。理解对象相等比较的机制对编写逻辑表达式非常重要，另外，对实现运算符重载和数据类型转换也非常重要，本章的后面将讨论运算符重载。

对象相等比较的机制对于引用类型(类的实例)的比较和值类型(基本数据类型，结构或枚举的实例)的比较来说是不同的。下面分别介绍引用类型和值类型的相等比较。

6.3.1　引用类型的相等比较

System.Object 的一个初看上去令人惊讶的方面是它定义了 3 个不同的方法，来比较对象的相等性：ReferenceEquals()和 Equals()的两个版本。再加上比较运算符==，实际上有 4 种进行相等比较的方式。这些方法有一些微妙的区别，下面就介绍这些方法。

1. ReferenceEquals()方法

ReferenceEquals()是一个静态方法，测试两个引用是否指向类的同一个实例，即两个引用是否包含内存中的相同地址。作为静态方法，它不能重写，所以只能使用 System.Object 的实现代码。如果提供的两个引用指向同一个对象实例，ReferenceEquals()总是返回 true，否则就返回 false。但是它认为 null 等于 null：

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

2. 虚拟的 Equals()方法

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

这个方法的.NET 基类也可能出问题。

3. 静态的 Equals()方法

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

4. 比较运算符＝＝

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

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

表示比较引用。但是，如果把一些类看作值，其含义就会比较直观。在这些情况下，最好重写比较运算符，以执行值的比较。后面将讨论运算符的重载，但显然它的一个例子是 System.String 类，Microsoft 重写了这个运算符，比较字符串的内容，而不是它们的引用。

6.3.2 值类型的相等比较

在进行值类型的相等比较时，采用与引用类型相同的规则：ReferenceEquals()用于比较引用，Equals()用于比较值，比较运算符可以看作是一个中间项。但最大的区别是值类型需要装箱，才能把它们转换为引用，才能对它们执行方法。另外，Microsoft 已经在 System.ValueType 类中重载了实例方法 Equals()，以便对值类型进行合适的相等测试。如果调用 sA.Equals(sB)，其中 sA 和 sB 是某个结构的实例，则根据 sA 和 sB 是否在其所有的字段中包含相同的值，而返回 true 或 false。另一方面，在默认情况下，不能对自己的结构重载==运算符。在表达式中使用(sA==sB)会导致一个编译错误，除非在代码中为结构提供了==的重载版本。

另外，ReferenceEquals()在应用于值类型时，总是返回 false，因为为了调用这个方法，值类型需要装箱到对象中。即使使用下面的代码：

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

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

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

6.4　运算符重载

本节将介绍为类或结构定义的另一种类型的成员：运算符重载。

C++开发人员应很熟悉运算符重载。但是，因为这个概念对 Java 和 VB 开发人员来说是全新的，所以这里要解释一下。C++开发人员可以直接跳到主要示例上。

运算符重载的关键是在类实例上不能总是调用方法或属性，有时还需要做一些其他的工作，例如对数值进行相加、相乘或逻辑操作，如比较对象等。假定要定义一个类，表示一个数学矩阵，在数学中，矩阵可以相加和相乘，就像数字一样。所以可以编写下面的代码：

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

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

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

学习到现在，像+和*这样的运算符只能用于预定义的数据类型，原因很简单：编译器认为所有常见的运算符都是用于这些数据类型的，例如，它知道如何把两个 long 加起来，或者如何从一个 double 中减去另一个 double，并生成合适的中间语言代码。但在定义自己的类或结构时，必须告诉编译器：什么方法可以调用，每个实例存储了什么字段等所有的信息。同样，如果要在自己的类上使用运算符，就必须告诉编译器相关的运算符在这个类中的含义。此时就要定义运算符重载。

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

在许多情况下，重载运算符允许生成可读性更高、更直观的代码，包括：

- 在数学领域中，几乎包括所有的数学对象：坐标、矢量、矩阵、张量和函数等。如果编写一个程序执行某些数学或物理建模，肯定会用类表示这些对象。
- 图形程序在计算屏幕上的位置时，也使用数学或相关的坐标对象。
- 表示大量金钱的类(例如，在财务程序中)。
- 字处理或文本分析程序也有表示语句、子句等的类，可以使用运算符把语句连接在一起(这是字符串连接的一种比较复杂的版本)。

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

6.4.1 运算符的工作方式

为了理解运算符是如何重载的，考虑一下在编译器遇到运算符时会发生什么样的情况是很有用的——我们用相加运算符+作为例子来讲解。假定编译器遇到下面的代码：

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

会发生什么情况：

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

编译器知道它需要把两个整数加起来，并把结果赋予 long。调用一个方法把数字加在一起时，表达式 myInteger + myUnsignedInt 是一种非常直观、方便的语法。该方法带有两个参数 myInteger 和 myUnsignedInt，并返回它们的和。所以它完成的任务与任何方法调用是一样的—— 它会根据参数类型查找最匹配的+运算符重载，这里是带两个整数参数的+运算符重载。与一般的重载方法一样，预定义的返回类型不会因为调用的方法版本而影响编译器的选择。在本例中调用的重载方法带两个 int 类型参数，返回一个 int，这个返回值随后会转换为 long。

下一行代码让编译器使用+运算符的另一个重载版本：

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

在这个例子中，参数是一个 double 类型的数据和一个 int 类型的数据，但+运算符没有带这种复合参数的重载形式，所以编译器认为，最匹配的+运算符重载是把两个 double 作为其参数的版本，并隐式地把 int 转换为 double。把两个 double 加在一起与把两个整数加在一起完全不同，浮点数存储为一个尾数和一个指数。把它们加在一起要按位移动一个 double 的尾数，让两个指数有相同的值，然后把尾数加起来，移动所得尾数的位，调整其指数，保证答案有尽可能高的精度。

现在，看看如果编译器遇到下面的代码，会发生什么：

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

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

如果编译器找到这样的重载版本，就调用它的实现代码。如果找不到，就要看看有没有可以用作最佳匹配的其他+运算符重载，例如某个运算符重载的参数是其他数据类型，但可以隐式地转换为 Vector 实例。如果编译器找不到合适的运算符重载，就会产生一个编译错误，就像找不到其他方法调用的合适重载一样。

6.4.2　运算符重载的示例：Vector 结构

本节将开发一个结构 Vector，来演示运算符重载，这个 Vector 结构表示一个三维矢量。如果数学不是你的强项，不必担心，我们会使这个例子尽可能简单。三维矢量只是三个(double)数字的一个集合，说明物体和原点之间的距离，表示数字的变量是 x、y 和 z，x 表示物体与原点在 x 方向上的距离，y 表示它与原点在 y 方向上的距离，z 表示高度。把这 3 个数字组合起来，就得到总距离。例如，如果 x=3.0, y=3.0, z=1.0，一般可以写作(3.0, 3.0, 1.0)，表示物体与原点在 x 方向上的距离是 3，与原点在 y 方向上的距离是 3，高度为 1。

矢量可以与矢量或数字相加或相乘。在这里我们使用术语“标量”(scalar)，它是数字的数学用语——在 C#中，就是一个 double。相加的作用是很明显的。如果先移动(3.0, 3.0, 1.0)，再移动(2.0, －4.0, －4.0)，总移动量就是把这两个矢量加起来。矢量的相加是指把每个元素分别相加，因此得到(6.0, －1.0, －3.0)。此时，数学表达式总是写成 c=a+b，其中 a 和 b 是矢量，c 是结果矢量。这与使用 Vector 结构的方式是一样的。

注意：

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

下面是 Vector 的定义——包含成员字段、构造函数和一个 ToString()重写方法，以便查看 Vector 的内容，最后是运算符重载：

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

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

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

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

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

下面是 Vector 结构的有趣部分——为+运算符提供支持的运算符重载：

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

运算符重载的声明方式与方法的声明方式相同，但 operator 关键字告诉编译器，它实际上是一个运算符重载，后面是相关运算符的符号，在本例中就是+。返回类型是在使用这个运算符时获得的类型。在本例中，把两个矢量加起来会得到另一个矢量，所以返回类型就是 Vector。对于这个+运算符重载，返回类型与包含类一样，但这种情况并不是必需的。两个参数就是要操作的对象。对于二元运算符(带两个参数)，如+和－运算符，第一个参数是放在运算符左边的值，第二个参数是放在运算符右边的值。

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

前面介绍了声明运算符+的语法，下面看看运算符内部的情况：

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

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

下面需要编写一些简单的代码，测试 Vector 结构：

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

           vect1 = new Vector(3.0, 3.0, 1.0);
```

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

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

把这些代码保存为 Vectors.cs，编译并运行它，结果如下：

```
Vectors

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

1. 添加更多的重载

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

矢量乘以标量只是矢量的元素分别与标量相乘，例如，2 * (1.0, 2.5, 2.0)就等于(2.0, 5.0, 4.0)。相关的运算符重载如下所示。

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

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

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

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

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

编译器处理运算符重载的方式和处理方法重载的方式是一样的。它会查看给定运算符的所有可用重载，找到与之最匹配的那个运算符重载。上面的语句要求第一个参数是 Vector，第二个参数是整数，或者可以隐式转换为整数的其他数据类型。我们没有提供这样一个重载。有一个运算符重载，其参数是一个 double 和一个 Vector，但编译器不能改变参数的顺序，所以这是不行的。还需要显式定义一个运算符重载，其参数是一个 Vector 和一个 double，有两种方式可以定义这样一个运算符重载，第一种方式和处理所有运算符的方式一样，显式执行矢量相乘操作：

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

假定已经编写了执行相乘操作的代码，最好重复使用该代码：

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

这段代码会告诉编译器，如果有 Vector 和 double 的相乘操作，编译器就使参数的顺序反序，调用另一个运算符重载。在本章的示例代码中，我们使用第二个版本，它看起来比较简洁。利用这个版本可以编写出维护性更好的代码，因为不需要复制代码，就可在两个独立的重载中执行相乘操作。

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

在数学上，如果有两个矢量(x, y, z)和(X, Y, Z)，其内积就是 x*X + y*Y + z*Z 的值。两个矢量这样相乘是很奇怪的，但这是很有效的，因为它可以用于计算各种其他的数。当然，如果要使用 Direct3D 或 DirectDraw 编写代码来显示复杂的 3D 图形，在计算对象放在屏幕上的什么位置时，常常需要编写代码来计算矢量的内积，作为中间步骤。这里我们关心的是使用 Vector 编写出 double X = a*b，其中 a 和 b 是矢量，并计算出它们的点积。相关的运算符重载如下所示：

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

定义了算术运算符后，就可以用一个简单的测试方法来看看它们是否能正常运行：

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

   vect3 = vect1 + vect2;

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

   vect3 += vect2;

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

   vect3 = vect1*2;
```

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

        double dot = vect1*vect3;

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

运行代码(Vectors2.cs)，得到如下所示的结果：

Vectors2

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

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

```
        vect3 += vect2;

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

虽然+=一般用作单个运算符，但实际上其操作分为两部分：相加和赋值。与 C++不同，C#不允许重载=运算符，但如果重载+运算符，编译器就会自动使用+运算符的重载来执行+=运算符的操作。 -=、&=、*=和/=赋值运算符也遵循此规则。

2. 比较运算符的重载

C#中有 6 个比较运算符，它们分为 3 对：

- == 和 !=
- > 和 <
- >= 和 <=

C#要求成对重载比较运算符。如果重载了==，也必须重载！=，否则会产生编译错误。另外，比较运算符必须返回 bool 类型的值。这是它们与算术运算符的根本区别。两个数相加或相减的结果，理论上取决于数的类型。而两个 Vector 的相乘会得到一个标量。另一个例子是.NET 基类 System.DateTime，两个 DateTime 实例相减，得到的结果不是 DateTime，而是一个 System.TimeSpan 实例，但比较运算得到的如果不是 bool 类型的值，就没有任何意义。

注意：

在重载==和!=时，还应重载从 System.Object 中继承的 Equals()和 GetHashCode()方法，否则会产生一个编译警告。原因是 Equals()方法应执行与==运算符相同的相等逻辑。

除了这些区别外，重载比较运算符所遵循的规则与算术运算符相同。但比较两个数并不像想象的那么简单，例如，如果比较两个对象引用，就是比较存储对象的内存地址。比

较运算符很少进行这样的比较，所以必须编写运算符，比较对象的值，返回相应的布尔结果。下面给 Vector 结构重载==和!=运算符。首先是==的执行代码：

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

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

注意：

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

还需要重载运算符!=，采用的方式如下：

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

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

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

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

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

   Console.WriteLine();

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

编译这些代码(下载代码中的 Vectors3.cs)，会得到一个编译器警告，因为我们没有为 Vector 重写 Equals()，对于本例，这是不重要的，所以忽略它。

csc Vectors3.cs

```
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2006. All rights reserved.

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

在命令行上运行该示例，生成如下结果：

Vectors3

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

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

3. 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 6-5 所示。

表　6-5

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

6.5 用户定义的数据类型转换

本章前面介绍了如何在预定义的数据类型之间转换数值，这是通过数据类型转换过程来完成的。C#允许进行两种不同数据类型的转换：隐式转换和显式转换。

显式转换要在代码中显式标记转换，其方法是在圆括号中写出目标数据类型：

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

对于预定义的数据类型，当数据类型转换可能失败或丢失某些数据时，需要显式转换。例如：

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

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

C#允许定义自己的数据类型(结构和类)，这意味着需要某些工具支持在自己的数据类型之间进行类型转换。方法是把数据类型转换定义为相关类的一个成员运算符，数据类型转换必须标记为隐式或显式，以说明如何使用它。我们应遵循与预定义数据类型转换相同的规则，如果知道无论在源变量中存储什么值，数据类型转换总是安全的，就可以把它定义为隐式转换。另一方面，如果某些数值可能会出错，例如丢失数据或抛出异常，就应把数据类型转换定义为显式转换。

提示：

如果源数据值会使数据类型转换失败，或者可能会抛出异常，就应把定制数据类型转换定义为显式转换。

定义数据类型转换的语法类似于本章前面介绍的重载运算符。但它们是不一致的，数据类型转换在某种情况下可以看作是一种运算符，其作用是从源类型转换为目标类型。为了说明这个语法，下面的代码是从本节后面介绍的结构 Currency 示例中节选的：

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

运算符的返回类型定义了数据类型转换操作的目标类型，它有一个参数，即要转换的源对象。这里定义的数据类型转换可以隐式地把 Currency 的值转换为 float 型。注意，如

果数据类型转换声明为隐式，编译器可以隐式或显式地使用这个转换。如果数据类型转换声明为显式，编译器就只能显式地使用它。与其他运算符重载一样，数据类型转换必须声明为 public 和 static。

注意：

C++开发人员应注意，这种情况与 C++是不同的，在 C++中，数据类型转换是类的实例成员。

6.5.1　执行用户定义的类型转换

本节将在示例 SimpleCurrency(和往常一样，其代码可以下载)中介绍隐式和显式使用用户定义的数据类型转换。在这个示例中，定义一个结构 Currency，它包含一个正的 USD($)钱款。C#为此提供了 decimal 类型，但如果要进行比较复杂的财务处理，仍可以编写自己的结构和类来表示钱款，在这样的类上执行特定的方法。

注意：

数据类型转换的语法对于结构和类是一样的。我们的示例定义了一个结构，但如果把 Currency 声明为类，也是可以的。

首先，结构 Currency 的定义如下所示。

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

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

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

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

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

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

为此，需要定义一个数据类型转换。给 Currency 定义添加下述代码：

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

这个数据类型转换是隐式的。在本例中这是一个合理的选择，因为在 Currency 定义中，可以存储在 Currency 中的值也都可以存储在 float 中。在这个转换中，不应出现任何错误。

注意：

这里有一点欺骗性:实际上，当把 uint 转换为 float 时，会有精确度的损失，但 Microsoft 认为这种错误并不重要，因此把从 uint 到 float 的转换都当做隐式转换。

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

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

下面的代码可以成功编译：

```
float amount = 46.63f;
Currency amount2 = (Currency)amount;
```

但是，下面的代码会抛出一个编译错误，因为试图隐式地使用一个显式的数据类型转换：

```
float amount = 46.63f;
Currency amount2 = amount;   // wrong
```

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

```
static void Main()
{
  try
```

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

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

        float balance2= balance;

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

        balance = (Currency) balance2;

        Console.WriteLine("After converting back to Currency, = " + balance);
        Console.WriteLine("Now attempt to convert out of range value of " +
                          " - $100.00 to a Currency:");
        checked
        {
            balance = (Currency) (-50.5);
            Console.WriteLine("Result is " + balance.ToString());
        }
    }
    catch(Exception e)
    {
        Console.WriteLine("Exception occurred: " + e.Message);
    }
}
```

注意，所有的代码都放在一个 try 块中，来捕获在数据类型转换过程中发生的任何异常。在 checked 块中还添加了把超出范围的值转换为 Currency 的测试代码，所以，负值是肯定会被捕获的。运行这段代码，得到如下所示的结果：

SimpleCurrency

```
50.35
Balance is $50.35
Balance is (using ToString()) $50.35
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$100.00 to a Currency:
Result is $4294967246.60486
```

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

第一个问题是由圆整错误引起的。如果类型转换用于把 float 转换为 uint，计算机就会截去多余的数字，而不是圆整它。计算机以二进制方式存储数字，而不是十进制，小数部分 0.35 不能用二进制小数来表示(像 1/3 这样的分数不能表示为小数，它应等于循环小数

0.3333)。所以，计算机最后存储了一个略小于 0.35 的值，它可以用二进制格式表示。把该数字乘以 100，就会得到一个小于 35 的数字，截去了 34 美分。显然在本例中，这种由截去引起的错误是很严重的，避免该错误的方式是确保在数字转换过程中执行智能圆整操作。Microsoft 编写了一个类 System.Convert 来完成该任务。System.Convert 包含大量的静态方法来执行各种数字转换，我们需要使用的是 Convert.ToUInt16()。注意，在使用 System.Convert 方法时会产生额外的性能损失，所以只应在需要时才使用它们。

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

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

```
public static explicit operator Currency (float value)
{
  checked
  {
      uint dollars = (uint)value;
    ushort cents = Convert.ToUInt16((value - dollars)*100);
    return new Currency(dollars, cents);
  }
}
```

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

注意：

System.Convert 的方法还执行它自己的溢出检查。因此对于本例的情况，不需要把对 Convert.ToUInt16()的调用放在 checked 环境下。但把 value 显式转换为美元值仍需要 checked 环境。

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

注意：

如果定义了一个使用非常频繁的数据类型转换，其性能也非常好，就可以不进行任何错误检查，如果对用户定义的转换和没有检查的错误进行了清晰的说明，这也是一种合法的解决方案。

1. 类之间的数据类型转换

Currency 示例仅涉及到与 float(一种预定义的数据类型)来回转换的类。实际上任何简单数据类型的转换都是可以自定义的。定义不同结构或类之间的数据类型转换是允许的，但有两个限制：

- 如果某个类直接或间接继承了另一个类，就不能定义这两个类之间的数据类型转换(这些类型的类型转换已经存在)。
- 数据类型转换必须在源或目标数据类型的内部定义。

要说明这些要求，假定有如图 6-1 所示的类层次结构。

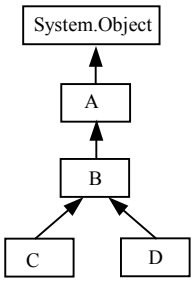

图　6-1

换言之，类 C 和 D 间接派生于 A。在这种情况下，在 A、B、C 或 D 之间唯一合法的类型转换就是类 C 和 D 之间的转换，因为这些类并没有互相派生。这段代码如下所示(假定希望数据类型转换是显式的，这是在用户定义的数据类型之间转换的通常情况)：

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

对于这些数据类型转换，可以选择放置定义的地方——在 C 的类定义内部，或者在 D 的类定义内部，但不能在其他地方定义。C#要求把数据类型转换的定义放在源类(或结构)或目标类(或结构)的内部。它的边界效应是不能定义两个类之间的数据类型转换，除非可以编辑它们的源代码。这是因为，这样可以防止第三方把数据类型转换引入类中。

一旦在一个类的内部定义了数据类型转换，就不能在另一个类中定义相同的数据类型转换。显然，只能有一个数据类型转换，否则编译器就不知道该选择哪个数据类型转换了。

2. 基类和派生类之间的数据类型转换

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

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

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

在本例中，是从 MyDerived 隐式地转换为 MyBase。这是因为对类 MyBase 的任何引用都可以引用类 MyBase 的对象或派生于 MyBase 的对象。在 OO 编程中，派生类的实例

实际上是基类的实例，但加上了一些额外的信息。在基类上定义的所有函数和字段也都在派生类上定义了。

下面看看另一种方式，编写下面的代码：

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

上面的代码都是合法的 C#代码(从句法的角度来看，是合法的)，是把基类转换为派生类。但是，最后的一个语句在执行时会抛出一个异常。在进行数据类型转换时，会检查被引用的对象。因为基类引用实际上可以引用一个派生类实例，所以这个对象可能是要转换的派生类的一个实例。如果是这样，转换就会成功，派生的引用被设置为引用这个对象。但如果该对象不是派生类(或者派生于这个类的其他类)的一个实例，转换就会失败，抛出一个异常。

注意，编译器已经提供了基类和派生类之间的转换，这种转换实际上并没有对对象进行任何数据转换。如果要进行的转换是合法的，它们也仅是把新引用设置为对对象的引用。这些转换在本质上与用户定义的转换不同。例如，在前面的 SimpleCurrency 示例中，我们定义了 Currency 结构和 float 之间的转换。在 float 到 Currency 的转换中，则实例化了一个新 Currency 结构，并用要求的值进行初始化。在基类和派生类之间的预定义转换则不是这样。如果要把 MyBase 实例转换为 MyDerived 对象，其值根据 MyBase 实例的内容来确定，就不能使用数据类型转换语法。最合适的选项通常是定义一个派生类的构造函数，它的参数是一个基类实例，让这个构造函数执行相关的初始化：

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

3. 装箱和拆箱数据类型转换

前面主要讨论了基类和派生类之间的数据类型转换，其中，基类和派生类都是引用类型。其规则也适用于转换值类型，但在转换值类型时，不是仅仅复制引用，还必须复制一些数据。

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

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

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

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

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

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

上述代码的工作方式与前面的引用类型一样。把 derivedObject 转换为 Currency 会成功进行，因为 derivedObject 实际上引用的是装箱 Currency 实例——转换的过程是把已装箱的 Currency 对象的字段复制到一个新的 Currency 结构中。第二个转换会失败，因为 baseObject 没有引用已装箱的 Currency 对象。

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

6.5.2　多重数据类型转换

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

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

首先初始化一个 Currency 实例，再把它转换为一个 long。问题是不能定义这样的转换。但是，这段代码仍可以编译成功。因为编译器知道我们要定义一个从 Currency 到 float 的隐式转换，而且它知道如何显式地从 float 转换为 long。所以它会把这行代码编译为中间语言代码，首先把 balance 转换为 float，再把结果转换为 long。上述代码的最后一行也是这样，把 balance 转换为 double 型时，因为从 Currency 到 float 的转换和从 float 到 double 的转换都是隐式的，就可以在代码中把这个转换当作一种隐式转换。如果要显式地指定转换过程，可以编写如下代码：

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

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

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

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

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

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

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

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

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

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

当然，转换为另一个数据类型后，再转换回来有时会丢失数据。例如，把包含 6.8 的 float 转换为 int，再转换回 float，会丢失数字中的小数部分，得到 5，但丢失数字中的小数部分和一个整数被 100 整除的情况略有区别。Currency 现在成了一种相当危险的类，它会对整数进行一些奇怪的操作。

问题是，在转换过程中如何解释整数是有矛盾的。从 Currency 到 float 的转换会把整数 1 解释为 1 美元，但从 uint 到 Currency 的转换会把这个整数解释为 1 美分，这是很糟糕的。如果希望类易于使用，就应确保所有的转换都按一种互相兼容的方式执行，即这些转换应得到相同的结果。在本例中，显然要重新编写从 uint 到 Currency 的转换，把整数值 1

解释为 1 美元：

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

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

测试这种转换是否成功，应确定无论使用什么转换路径，它都能得到相同的结果(而不是像在从 float 到 int 的转换过程中丢失数据那样)。Currency 类就是一个很好的示例。下面的代码：

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

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

但假定要添加另一个转换，从 Currency 隐式地转换为 uint，就需要修改 Currency 结构，添加从 uint 到 Currency 的转换和从 Currency 到 uint 的转换，这段代码可以下载，作为 SimpleCurrency2 示例：

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

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

现在，编译器从 Currency 转换到 ulong 可以使用另一条路径：先从 Currency 隐式地转换为 uint，再隐式地转换为 ulong。该采用哪条路径？ C#有一些规则(本书不详细讨论这些规则，读者可参阅 MSDN 文档说明)，告诉编译器如何确定哪条是最佳路径。但最好自己设计转换，让所有的转换都得到相同的结果(但没有精确度的损失)，此时编译器选择哪条路径就不重要了(在本例中，编译器会选择 Currency→uint→ulong 路径，而不是 Currency→float→ulong 路径)。

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

```
try
{
```

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

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

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

运行这个示例，得到如下所示的结果：

SimpleCurrency2

```
50
balance is $50.35
balance is (using ToString()) $50.35
Converting to uint gives 50
After converting to float, = 50.35
After converting back to Currency, = $50.34
Now attempt to convert out of range value of -$100.00 to a Currency:
Exception occurred: Arithmetic operation resulted in an overflow.
```

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

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

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

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

第三行的 Console.WriteLine()语句显式调用 Currency.ToString()方法，以确保 Currency 显示为一个字符串。第二行代码没有这么做。字符串"balance is "传送给 Console.WriteLine()，告诉编译器这个参数应解释为字符串，因此要隐式地调用 Currency.ToString()方法。

但第一行的 Console.WriteLine()方法只是把原来的 Currency 结构传送给 Console.WriteLine()。目前 Console.WriteLine()有许多重载，但它们的参数都不是 Currency 结构。所以编译器会到处搜索，看看它能把 Currency 转换为什么，以与 Console.WriteLine()的一个重载方法匹配。如上所示，Console.WriteLine()的一个重载方法可以快速而高效地显示 uint，且其参数是一个 uint。因此应把 Currency 隐式地转换为 uint。

实际上，Console.WriteLine()有另一个重载方法，它的参数是一个 double，结果是显示该 double 的值。如果仔细看看第一个 SimpleCurrency 示例的结果，就会发现该结果的第一行就是使用这个重载方法把 Currency 显示为一个 double。在这个示例中，没有直接把 Currency 转换为 uint，所以编译器选择 Currency → float → double 作为可用于

Console.WriteLine()重载方法的首选转换方式。但在 SimpleCurrency2 中可以直接转换为 uint，所以编译器会选择后者。

如果方法调用带有多个重载方法，并要给该方法传送参数，而该参数的数据类型不匹配任何重载方法，就可以迫使编译器确定使用哪些转换方式进行数据转换，决定使用哪个重载方法(并进行相应的数据转换)。当然，编译器总是按逻辑和严格的规则来工作，但结果可能并不是我们所期望的。如果可能会出问题，最好显式指定转换路径。

6.6　小结

本章介绍了 C#提供的标准运算符，描述了对象的相等机制，讨论了编译器如何把一种标准数据类型转换为另一种标准数据类型。还阐述了如何使用运算符重载在自己的数据类型上执行定制的运算符。最后，学习了运算符重载的一种特殊类型，即数据类型转换运算符，它允许用户指定如何将实例转换为其他数据类型。

第 7 章将介绍两个密切相关的成员类型：委托和事件，在自己的类型上也可以实现这两个成员类型，以支持基于事件的对象模型。

第 7 章

委托和事件

回调(callback)函数是 Windows 编程的一个重要部分。如果您具备 C 或 C++编程背景，就曾在许多 Windows API 中使用过回调。VB 添加了 AddressOf 关键字后，开发人员就可以利用以前一度受到限制的 API 了。回调函数实际上是方法调用的指针，也称为函数指针，是一个非常强大的编程特性。.NET 以委托的形式实现了函数指针的概念。它们的特殊之处是，与 C 函数指针不同，.NET 委托是类型安全的。这说明，C 中的函数指针只不过是一个指向存储单元的指针，我们无法说出这个指针实际指向什么，像参数和返回类型等就更无从知晓了。如本章所述，.NET 把委托作为一种类型安全的操作。本章后面将学习.NET 如何将委托用作实现事件的方式。

7.1 委托

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

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

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

- 启动线程—— 在 C#中，可以告诉计算机并行运行某些新的执行序列。这种序列就称为线程，在基类 System.Threading.Thread 的一个实例上使用方法 Start()，就可以开始执行一个线程。如果要告诉计算机开始一个新的执行序列，就必须说明要在哪里执行该序列。必须为计算机提供开始执行的方法的细节，即 Thread.Start()方法必须带有一个参数，该参数定义了要由线程调用的方法。

- 通用库类——有许多库包含执行各种标准任务的代码。这些库通常可以自我包含。这样在编写库时，就会知道任务该如何执行。但是有时在任务中还包含子任务，只有使用该库的客户机代码才知道如何执行这些子任务。例如编写一个类，它带有一个对象数组，并把它们按升序排列。但是，排序的部分过程会涉及到重复使用数组中的两个对象，比较它们，看看哪一个应放在前面。如果要编写的类必须能给任何对象数组排序，就无法提前告诉计算机应如何比较对象。处理类中对象数组的客户机代码也必须告诉类如何比较要排序的对象。换言之，客户机代码必须给类传递某个可以进行这种比较的合适方法的细节。
- 事件—— 一般是通知代码发生了什么事件。GUI 编程主要是处理事件。在发生事件时，运行库需要知道应执行哪个方法。这就需要把处理事件的方法传送为委托的一个参数。这些将在本章后面讨论。

在 C 和 C++中，只能提取函数的地址，并传送为一个参数。C 是没有类型安全性的。可以把任何函数传送给需要函数指针的方法。这种直接的方法会导致一些问题，例如类型的安全性，在进行面向对象编程时，方法很少是孤立存在的，在调用前，通常需要与类实例相关联。而这种方法并没有考虑到这个问题。所以.NET Framework 在语法上不允许使用这种直接的方法。如果要传递方法，就必须把方法的细节封装在一种新类型的对象中，即委托。委托只是一种特殊的对象类型，其特殊之处在于，我们以前定义的所有对象都包含数据，而委托包含的只是方法的地址。

7.1.1 在 C#中声明委托

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

定义委托的语法如下：

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

在这个示例中，定义了一个委托 IntMethodInvoker，并指定该委托的每个实例都包含一个方法的细节，该方法带有一个 int 参数，并返回 void。理解委托的一个要点是它们的类型安全性非常高。在定义委托时，必须给出它所代表的方法签名和返回类型的全部细节。

提示：

理解委托的一种好方式是把委托当作给方法签名和返回类型指定名称。

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

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

或者定义一个委托，它代表的方法不带参数，返回一个 string 型的值，则可以编写如下代码：

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

其语法类似于方法的定义，但没有方法体，定义的前面要加上关键字 delegate。因为定义委托基本上是定义一个新类，所以可以在定义类的任何地方定义委托，既可以在另一个类的内部定义，也可以在任何类的外部定义，还可以在命名空间中把委托定义为顶层对象。根据定义的可见性，可以在委托定义上添加一般的访问修饰符：public、 private 和 protected 等：

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

注意：

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

定义好委托后，就可以创建它的一个实例，以存储特定方法的细节。

注意：

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

7.1.2　在 C#中使用委托

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

```
private delegate string GetAString();

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

在这段代码中，实例化了类型为 GetAString 的一个委托，并对它进行初始化，使它引用整型变量 x 的 ToString()方法。在 C#中，委托在语法上总是带有一个参数的构造函数，这个参数就是委托引用的方法。这个方法必须匹配最初定义委托时的签名。所以在这个示例中，如果用不带参数、返回一个字符串的方法来初始化 firstStringMethod 变量，就会产生

一个编译错误。注意，int.ToString()是一个实例方法(不是静态方法)，所以需要指定实例(x)和方法名来正确初始化委托。

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

实际上，给委托实例提供括号与调用委托类的 Invoke()方法完全相同。firstStringMethod 是委托类型的一个变量，所以 C#编译器会用 firstStringMethod. Invoke()代替 firstStringMethod()。

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

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

提示：

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

为了说明这一点，我们扩展上面的代码，让它使用 firstStringMethod 委托在另一个对象上调用其他两个方法，其中一个是实例方法，另一个是静态方法。为此，再次使用本章前面定义的 Currency 结构。

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

   public Currency(uint dollars, ushort cents)
   {
      this.Dollars = dollars;
      this.Cents = cents;
   }
   public override string ToString()
   {
      return string.Format("${0}.{1,-2:00}", Dollars,Cents);
   }
   public static explicit operator Currency (float value)
   {
      checked
      {
         uint dollars =(uint)value;
         ushort cents =(ushort)((value-dollars)*100);
         return new Currency(dollars,cents);
       }
   }
   public static implicit operator float (Currency value)
```

```
        {
            return value.Dollars + (value.Cents/100.0f);
        }
    public static implicit operator Currency (uint value)
    {
       return new Currency(value, 0);
    }
        public static implicit operator uint (Currency value)
        {
            return value.Dollars;
        }
      }
```

Currency 结构已经有了自己的 ToString()重载方法。为了说明如何使用带有静态方法的委托，再增加一个静态方法，其签名与 Currency 的签名相同：

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

下面就可以使用 GetAString 实例，代码如下所示：

```
         private delegate string GetAString();

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

            Currency balance = new Currency(34, 50);

            // firstStringMethod references an instance method
            firstStringMethod = new GetAString(balance.ToString);
            Console.WriteLine("String is " + firstStringMethod());

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

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

运行应用程序，会得到委托引用的不同方法的结果：

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

但是，我们还没有说明把一个委托传递给另一个方法的具体过程，也没有给出任何有用的结果。调用 int 和 Currency 对象的 ToString()的方法要比使用委托直观得多！在真正领会到委托的用处前，需要用一个相当复杂的示例来说明委托的本质。下面就是两个委托的示例。第一个示例仅使用委托来调用两个不同的操作，说明了如何把委托传递给方法，如何使用委托数组，但这仍没有很好地说明：没有委托，就不能完成很多工作。第二个示例就复杂得多了，它有一个类 BubbleSorter，执行一个方法，按照升序排列一个对象数组，这个类没有委托是很难编写出来的。

7.2 委托推断

C# 2.0 用委托推断扩展了委托的语法。只要需要委托实例，就可以只传送地址的名称。前面的示例用 GetAString 委托的一个新实例初始化了 GetAString 类型的变量 firstStringMethod：

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

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

```
GetAString firstStringMethod = x.ToString;
```

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

注意：

不能调用 x.ToString()方法，把它传送给委托变量。调用 x.ToString()方法会返回一个不能赋予委托变量的字符串对象。只能把方法的地址赋予委托变量。

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

7.3 匿名方法

到目前为止，要想使委托工作，方法必须已经存在(即委托是用方法的签名定义的)。但使用委托还有另外一种方式：即通过匿名方法。匿名方法是用作委托参数的一个代码块。

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

```
using System;
```

```
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    delegate string DelegateTest(string val);

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

      delegateTest anonDel = delegate(string param)
      {
        param += mid;
        param += " and this was added to the string.";
        return param;
      };

      Console.WriteLine(anonDel("Start of string"));

    }
  }
}
```

委托 DelegateTest 在类 Program 中定义，它带一个字符串参数。有区别的是 Main 方法。在定义 anonDel 时，不是传送已知的方法名，而是使用一个简单的代码块：它前面是关键字 delegate，后面是一个参数：

```
delegate(string param)
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
```

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

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

在使用匿名方法时，必须遵循两个规则。在匿名方法中不能使用跳转语句跳到该匿名方法的外部，反之亦然：匿名方法外部的跳转语句不能跳到该匿名方法的内部。

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

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

7.3.1 简单的委托示例

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

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

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

下面调用这些方法：

```
using System;

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

   class MainEntryPoint
   {
      static void Main()
      {
         DoubleOp [] operations =
            {
               new DoubleOp(MathsOperations.MultiplyByTwo),
               new DoubleOp(MathsOperations.Square)
            };

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

      static void ProcessAndDisplayNumber(DoubleOp action, double value)
      {
         double result = action(value);
         Console.WriteLine(
```

```
            "Value is {0}, result of operation is {1}", value, result);
        }
    }
}
```

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

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

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

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

- operations[i]表示“这个委托”。换言之，就是委托代表的方法。
- operations[i](2.0)表示“调用这个方法，参数放在括号中”。

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

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

在这个方法中，调用：

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

这实际上是调用 action 委托实例封装的方法，其返回结果存储在 result 中。

运行这个示例，得到如下所示的结果：

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

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

如果在这个例子中使用匿名方法，就可以删除第一个类 MathOperations。Main 方法应如下所示：

```
static void Main()
{
  DoubleOp multByTwo = delegate(double val) {return val * 2;}
  DoubleOp square = delegate(double val) {return val * val;}

  DoubleOp [] operations = {multByTwo, square};
```

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

运行这个版本，结果与前面的例子相同。其优点是删除了一个类。

7.3.2 BubbleSorter 示例

下面的示例将说明委托的用途。我们要编写一个类 BubbleSorter，它执行一个静态方法 Sort()，这个方法的第一个参数是一个对象数组，把该数组按照升序重新排列。换言之，假定传递的是 int 数组：{0, 5, 6, 2, 1}，则返回的结果应是{0, 1, 2, 5, 6}。

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

```
    for (int i = 0; i < sortArray.Length; i++)
    {
      for (int j = i + 1; j < sortArray.Length; j++)
      {
        if (sortArray[j] < sortArray[i])  // problem with this test
        {
          int temp = sortArray[i];  // swap ith and jth entries
          sortArray[i] = sortArray[j];
          sortArray[j] = temp;
        }
      }
    }
```

它非常适合于 int，但我们希望 Sort()方法能给任何对象排序。换言之，如果某段客户机代码包含 Currency 结构数组或其他类和结构，就需要对该数组排序。这样，上面代码中的 if(sortArray[j] < sortArray[i])就有问题了，因为它需要比较数组中的两个对象，看看哪一个更大。可以对 int 进行这样的比较，但如何对直到运行期间才知道或确定的新类进行比较？答案是客户机代码知道类在委托中传递的是什么方法，封装这个方法就可以进行比较。

定义如下的委托：

```
    delegate bool CompareOp(object lhs, object rhs);
```

给 Sort 方法指定下述签名：

```
    static public void Sort(object [] sortArray, CompareOp gtMethod)
```

这个方法的文档说明强调，gtMethod 必须表示一个静态方法，该方法带有两个参数，如果第二个参数的值“大于”第一个参数(换言之，它应放在数组中靠后的位置)，就返回 true。

注意：

这里使用的是委托，但也可以使用接口来解决这个问题。.NET 提供的 IComparer 接口就用于此目的。但是这里使用委托是因为这种问题本身要求使用委托。

设置完毕后，下面定义类 BubbleSorter：

```
class BubbleSorter
{
   static public void Sort(object [] sortArray, CompareOp gtMethod)
   {
      for (int i=0 ; i<sortArray.Length ; i++)
      {
         for (int j=i+1 ; j<sortArray.Length ; j++)
         {
            if (gtMethod(sortArray[j], sortArray[i]))
            {
               object temp = sortArray[i];
               sortArray[i] = sortArray[j];
               sortArray[j] = temp;
            }
         }
      }
   }
}
```

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

```
class Employee
{
   private string name;
   private decimal salary;

   public Employee(string name, decimal salary)
   {
      this.name = name;
      this.salary = salary;
   }

   public override string ToString()
   {
      return string.Format(name + ", {0:C}", salary);
   }
```

```
        public static bool RhsIsGreater(object lhs, object rhs)
        {
           Employee empLhs = (Employee) lhs;
           Employee empRhs = (Employee) rhs;
           return (empRhs.salary > empLhs.salary);
        }
     }
```

注意，为了匹配 CompareOp 委托的签名，在这个类中必须定义 RhsIsGreater，它的参数是两个对象引用，而不是 Employee 引用。必须把这些参数的数据类型转换为 Employee 引用，才能进行比较。

下面编写一些客户机代码，完成排序：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
   delegate bool CompareOp(object lhs, object rhs);

   class MainEntryPoint
   {
      static void Main()
      {
         Employee [] employees =
            {
               new Employee("Bugs Bunny", 20000),
               new Employee("Elmer Fudd ", 10000),
               new Employee("Daffy Duck", 25000),
               new Employee("Wiley Coyote", (decimal)1000000.38),
               new Employee("Foghorn Leghorn", 23000),
               new Employee("RoadRunner'", 50000)};

         CompareOp employeeCompareOp = new CompareOp(Employee.RhsIsGreater);
         BubbleSorter.Sort(employees, employeeCompareOp);

         for (int i=0 ; i<employees.Length ; i++)
         {
            Console.WriteLine(employees[i].ToString());
}
      }
     }
   }
```

运行这段代码，正确显示按照薪水排列的 Employee，如下所示：

```
BubbleSorter
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
```

```
RoadRunner, $50,000.00
Wiley Coyote, $1,000,000.38
```

7.3.3　多播委托

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

下面的代码取自于 SimpleDelegate 示例。尽管其语法与以前相同，但实际上它实例化了一个多播委托 Operations：

```
   delegate void DoubleOp(double value);
//   delegate double DoubleOp(double value);   // can't do this now

   class MainEntryPoint
   {
      static void Main()
      {
         DoubleOp operations = new DoubleOp(MathOperations.MultiplyByTwo);
         operations += new DoubleOp(MathOperations.Square);
```

使用委托推断可以编写上面的代码。另外，这种方式也更容易理解：

```
DoubleOp operations = MathOperations.MultiplyByTwo;
operations += MathOperations.Square;
```

在前面的示例中，要存储对两个方法的引用，所以实例化了一个委托数组。而这里只是在一个多播委托中添加两个操作。多播委托可以识别运算符+和+=。还可以扩展上述代码中的最后两行，它们具有相同的效果：

```
   DoubleOp operation1 = MathOperations.MultiplyByTwo;
   DoubleOp operation2 = MathOperations.Square;
   DoubleOp operations = operation1 + operation2;
```

多播委托还识别运算符 - 和 -=，以从委托中删除方法调用。

注意：

根据后面的内容，多播委托是一个派生于 System.MulticastDelegate 的类，System.MulticastDelegate 又派生于基类 System.Delegate。System.MulticastDelegate 的其他成员允许把多个方法调用链接在一起，成为一个列表。

为了说明多播委托的用法，下面把 SimpleDelegate 示例改写为一个新示例 MulticastDelegate。现在需要把委托表示为返回 void 的方法，就应重写 MathOperations 类中的方法，让它们显示其结果，而不是返回它们：

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

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

为了适应这个改变，也必须重写 ProcessAndDisplayNumber：

```
static void ProcessAndDisplayNumber(DoubleOp action, double valueToProcess)
{
   Console.WriteLine("\nProcessAndDisplayNumber called with value = " +
                     valueToProcess);
   action(valueToProcess);
}
```

下面测试多播委托，其代码如下：

```
static void Main()
{
   DoubleOp operations = MathOperations.MultiplyByTwo;
   operations += MathOperations.Square;

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

现在，每次调用 ProcessAndDisplayNumber 时，都会显示一个信息，说明它已经被调用。然后，下面的语句会按顺序调用 action 委托实例中的每个方法：

```
action(value);
```

运行这段代码，得到如下所示的结果：

```
MulticastDelegate

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

ProcessAndDisplayNumber called with value = 7.94
```

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

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

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

通过一个委托调用多个方法还有一个大问题。多播委托包含一个逐个调用的委托集合。如果通过委托调用的一个方法抛出了异常，整个迭代就会停止。下面是 MulticastIteration 示例。其中定义了一个简单的委托 DemoDelegate，它没有参数，返回 void。这个委托调用方法 One()和 Two()，这两个方法满足委托的参数和返回类型要求。注意方法 One()抛出了一个异常：

```
using System;

namespace Wrox.ProCSharp.Delegates
{
   public delegate void DemoDelegate();

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

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

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

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

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

```
        }
    }
}
```

委托只调用了第一个方法。第一个方法抛出了异常，所以委托的迭代会停止，不再调用 Two()方法。当调用方法的顺序没有指定时，结果会有所不同。

```
One
Exception Caught
```

注意：
错误和异常详见第 13 章。

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

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

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

修改了代码后运行应用程序，会看到在捕获了异常后，将继续迭代下一个方法。

```
One
Exception caught
Two
```

7.4 事件

基于 Windows 的应用程序也是基于消息的。这说明，应用程序是通过 Windows 来通信的，Windows 又是使用预定义的消息与应用程序通信的。这些消息是包含各种信息的结构，应用程序和 Windows 使用这些信息决定下一步的操作。在 MFC 等库或 VB 等开发环境推出之前，开发人员必须处理 Windows 发送给应用程序的消息。VB 和今天的.NET 把这

些传送来的消息封装在事件中。如果需要响应某个消息，就应处理对应的事件。一个常见的例子是用户单击了窗体中的按钮后，Windows 就会给按钮消息处理程序(有时称为 Windows 过程或 WndProc)发送一个 WM_MOUSECLICK 消息。对于.NET 开发人员来说，这就是按钮的 Click 事件。

在开发基于对象的应用程序时，需要使用另一种对象通信方式。在一个对象中发生了有趣的事情时，就需要通知其他对象发生了什么变化。这里又要用到事件。就像.NET Framework 把 Windows 消息封装在事件中那样，也可以把事件用作对象之间的通信介质。

委托就用作应用程序接收到消息时封装事件的方式。在上一节介绍委托时，仅讨论了理解事件如何工作所需要的内容。但 Microsoft 设计 C#事件的目的是让用户无需理解底层的委托，就可以使用它们。所以下面开始从客户软件的角度讨论事件，主要考虑的是需要编写什么代码来接收事件通知，而无需担心后台上究竟发生了什么，从中可以看出事件的处理十分简单。之后，编写一个生成事件的示例，介绍事件和委托之间的关系。

本节的内容对 C++开发人员最有用，因为 C++没有与事件类似的概念。另一方面，C#事件与 VB 事件非常类似，但 C#中的语法和底层的实现有所不同。

注意：

这里的术语“事件”有两种不同的含义。第一，表示发生了某个有趣的事情；第二，表示 C#语言中已定义的一个对象，即处理通知过程的对象。在使用第二个含义时，我们常常把事件表示为 C#事件，或者在其含义很容易从上下文中看出时，就表示为事件。

7.4.1　从客户的角度讨论事件

事件接收器是指在发生某些事情时被通知的任何应用程序、对象或组件。当然，有事件接收器，就有事件发送器。发送器的作用是引发事件。发送器可以是应用程序中的另一个对象或程序集，在系统事件中，例如鼠标单击或键盘按键，发送器就是.NET 运行库。注意，事件的发送器并不知道接收器是谁。这就使事件非常有用。

现在，在事件接收器的某个地方有一个方法，它负责处理事件。在每次发生已注册的事件时，就执行这个事件处理程序。此时就要使用委托了。由于发送器对接收器一无所知，所以无法设置两者之间的引用类型，而是使用委托作为中介。发送器定义接收器要使用的委托，接收器将事件处理程序注册到事件中。连接事件处理程序的过程称为封装事件。封装 Click 事件的简单例子有助于说明这个过程。

首先创建一个简单的 Windows 窗体应用程序，把一个按钮控件从工具箱拖放到窗体上。在属性窗口中把按钮重命名为 buttonOne。在代码编辑器中把下面的代码添加到 Form1 构造函数中：

```
public Form1()
{
    InitializeComponent();
    buttonOne.Click += new EventHandler(Button_Click);
```

```
}
```

在 Visual Studio 中，注意在输入+=运算符之后，就只需按下 Tab 键两次，编辑器就会完成剩余的输入工作。在大多数情况下这很不错。但在这个例子中，不使用默认的处理程序名，所以应自己输入文本。

这将告诉运行库，在引发 buttonOne 的 Click 事件时，应执行 Button_Click 方法。EventHandler 是事件用于把处理程序(Button_Click)赋予事件(Click)的委托。注意使用+=运算符把这个新方法添加到委托列表中。这类似于本章前面介绍的多播示例。也就是说，可以为事件添加多个事件处理程序。由于这是一个多播委托，所以要遵循添加多个方法的所有规则，但是不能保证调用方法的顺序。下面在窗体上再添加一个按钮，把它重命名为 buttonTwo。把 buttonTwo 的 Click 事件也连接到同一个 Button_Click 方法上，如下所示：

```
buttonOne.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(Button_Click);
```

利用委托推断，可以编写代码。

```
buttonOne.Click += Button_Click;
buttonTwo.Click += Button_Click;
```

EventHandler 委托已在.NET Framework 中定义了。它位于 System 命名空间，所有在.NET Framework 中定义的事件都使用它。如前所述，委托要求添加到委托列表中的所有方法都必须有相同的签名。显然事件委托也有这个要求。下面是 Button_Click 方法的定义：

```
Private void Button_Click(object sender, Eventargs e)
{

}
```

这个方法有几个重要的地方。首先，它总是返回 void。事件处理程序不能有返回值。其次是参数。只要使用 EventHandler 委托，参数就应是 object 和 EventArgs。第一个参数是引发事件的对象，在这个例子中是 buttonOne 或 buttonTwo，这取决于被单击的按钮。把一个引用发送给引发事件的对象，就可以把同一个事件处理程序赋予多个对象。例如，可以为几个按钮定义一个按钮单击处理程序，接着根据 sender 参数确定单击了哪个按钮。

第二个参数 EventArgs 是包含有关事件的其他有用信息的对象。这个参数可以是任意类型，只要它派生于 EventArgs 即可。MouseDown 事件使用 MouseDownEventArgs，它包含所使用按钮的属性、指针的 X 和 Y 坐标，以及与事件相关的其他信息。注意，其命名模式是在类型的后面加上 EventArgs。本章的后面将介绍如何创建和使用基于 EventArgs 的定制对象。

方法的命名也应注意。按照约定，事件处理程序应遵循“object_event”的命名约定。object 就是引发事件的对象，而 event 就是被引发的事件。从可读性来看，应遵循这个命名约定。

本例最后在处理程序中添加了一些代码，以完成一些工作。记住有两个按钮使用同一个处理程序。所以首先必须确定是哪个按钮引发了事件，接着调用应执行的操作。在本例中，只是在窗体的一个标签控件上输出一些文本。把一个标签控件从工具箱拖放到窗体上，并将其命名为 labelInfo，然后在 Button_Click 方法中编写如下代码：

```
if(((Button)sender).Name == "buttonOne")
   labelInfo.Text = "Button One was pressed";
else
   labelInfo.Text = "Button Two was pressed";
```

注意，由于 sender 参数作为对象发送，所以必须把它的数据类型转换为引发事件的对象类型，在本例中就是 Button。本例使用 Name 属性确定是哪个按钮引发了对象，也可以使用其他属性。例如 Tag 属性就可以处理这种情形，因为它可以包含任何内容。为了了解事件委托的多播功能，给 buttonTwo 的 Click 事件添加另一个方法，使用默认的方法名。窗体的构造函数如下所示：

```
buttonOne.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(Button_Click);
buttonTwo.Click += new EventHandler(buttonTwo_Click);
```

如果让 Visual Studio 创建存根(stub)，就会在源文件的末尾得到如下方法。但是，必须添加对 MessageBox 函数的调用：

```
Private void buttonTwo_Click(object sender, EventArgs e)
{
  MessageBox.Show("This only happens in Button 2 click event");
}
```

如果使用匿名方法，就不需要 Button_Click 方法和 Button2_Click 方法了。事件的代码如下：

```
buttonOne.Click += delegate {labelInfo.Text = "Button One was pressed";};
buttonTwo.Click += delegate {labelInfo.Text = "Button Two was pressed";};
buttonTwo.Click += delegate
       {
          MessageBox.Show("This only happens in Button 2 click event");
       };
```

在运行这个例子时，单击 buttonOne 会改变标签上的文本。单击 buttonTwo 不仅会改变文本，还会显示消息框。注意，不能保证标签文本在消息框显示之前改变，所以不要在处理程序中编写具有依赖性的代码。

我们已经学习了许多概念，但要在接收器中编写的代码量是很小的。记住，编写事件接收器常常比编写事件发送器要频繁得多。至少在 Windows 用户界面上，Microsoft 已经编写了所有需要的事件发送器(它们都在.NET 基类中，在 Windows.Forms 命名空间中)。

7.4.2 生成事件

接收事件并响应它们仅是事件的一个方面。为了使事件真正发挥作用，还需要在代码中生成和引发事件。下面的例子将介绍如何创建、引发、接收和取消事件。

这个例子包含一个窗体，它会引发另一个类正在监听的事件。在引发事件后，接收对象就确定是否执行一个过程，如果该过程未能继续，就取消事件。本例的目标是确定当前时间的秒数是大于 30 还是小于 30。如果秒数小于 30，就用一个表示当前时间的字符串设置一个属性；如果秒数大于 30，就取消事件，把时间字符串设置为空。

用于生成事件的窗体包含一个按钮和一个标签。下载的示例代码把按钮命名为 buttonRaise，标签命名为 labelInfo，也可以给标签使用其他名称。在创建窗体，添加两个控件后，就可以创建事件和相应的委托了。在窗体类的类声明部分，添加如下代码：

```
public delegate void ActionEventHandler(object sender, ActionCancelEventArgs ev);
public static event ActionEventHandler Action;
```

这两行代码的作用是什么？首先，我们声明了一种新的委托类型 ActionEventHandler。必须创建一种新委托，而不使用.NET Framework 预定义的委托，其原因是后面要使用定制的 EventArgs 类。方法签名必须与委托匹配。有了一个要使用的委托后，下一行代码就定义事件。在本例中定义了 Action 事件，定义事件的语法要求指定与事件相关的委托。还可以使用在.NET Framework 中定义的委托。从 EventArgs 类中派生出了近 100 个类，应该可以找到一个自己能使用的类。但本例使用的是定制的 EventArgs 类，所以必须创建一个与之匹配的新委托类型。

基于 EventArgs 的新类 ActionCancelEventHandler 实际上派生于 CancelEventArgs，而 CancelEventArgs 派生于 EventArgs。CancelEventArgs 添加了 Cancel 属性，该属性是一个布尔值，它通知 sender 对象，接收器希望取消或停止事件的处理。在 ActionEventHandler 类中，还添加了 Message 属性，这是一个字符串属性，包含事件处理状态的文本信息。下面是 ActionCancelEventHandler 类的代码：

```
public class ActionCancelEventArgs : System.ComponentModel.CancelEventArgs
{
    string message = String.Empty;

    public ActionCancelEventArgs() : base() {}

    public ActionCancelEventArgs(bool cancel) : base(cancel) {}

    public ActionCancelEventArgs(bool cancel, string message) : base(cancel)
    {
        this.message = message;
    }

    public string Message
    {
```

```
        get {return message;}
        set {message = value;}
    }
}
```

可以看出，所有基于 EventArgs 的类都负责在发送器和接收器之间来回传送事件的信息。在大多数情况下，EventArgs 类中使用的信息都被事件处理程序中的接收器对象使用。但是，有时事件处理程序可以把信息添加到 EventArgs 类中，使之可用于发送器。这就是本例使用 EventArgs 类的方式。注意在 EventArgs 类中有两个可用的构造函数。这种额外的灵活性增加了该类的可用性。

目前声明了一个事件，定义了一个委托，并创建了 EventArgs 类。下一步需要引发事件。真正需要做的是用正确的参数调用事件，如本例所示：

```
ActionCancelEventArgs e = new ActionCancelEventArgs();
Action(this, e);
```

这非常简单。创建新的 ActionCancelEventArgs 类，并把它作为一个参数传递给事件。但是，这有一个小问题。如果事件不会在任何地方使用，该怎么办？如果还没有为事件定义处理程序，该怎么办？Action 事件实际上是空的。如果试图引发该事件，就会得到一个空引用异常。如果要派生一个新的窗体类，并使用该窗体，把 Action 事件定义为基事件，则只要引发了 Action 事件，就必须执行其他一些操作。目前，我们必须在派生的窗体中激活另一个事件处理程序，这样才能访问它。为了使这个过程容易一些，并捕获空引用错误，就必须创建一个方法 OnEvent Name，其中 EventName 是事件名。在这个例子中，有一个 OnAction 方法，下面是 OnAction 方法的完整代码：

```
protected void OnAction(object sender, ActionCancelEventArgs ev)
{
  if(Action != null)
  {
     Action(sender, ev);
  }
}
```

代码并不多，但完成了需要的工作。把该方法声明为 protected，就只有派生类可以访问它。事件在引发之前还会进行空引用测试。如果派生一个包含该方法和事件的新类，就必须重写 OnAction 方法，然后连接事件。为此，必须在重写代码中调用 base.OnAction()。否则就不会引发该事件。在整个.NET Framework 中都用这个命名约定，并在.NET SDK 文档中对这一命名规则进行了说明。

注意传送给 OnAction 方法的两个参数。它们看起来很熟悉，因为它们与需要传送给事件的参数相同。如果事件需要从另一个对象中引发，而不是从定义方法的对象中引发，就需要把访问修饰符设置为 internal 或 public，而不能设置为 protected。有时让类只包含事件声明，这些事件从其他类中调用是有意义的。仍可以创建 OnEventName 方法，但此时它们是静态方法。

目前，我们已经引发了事件，还需要一些代码来处理它。在项目中创建一个新类 BusEntity。本项目的目的是检查当前时间的秒数，如果它小于 30，就把一个字符串值设置为时间；如果它大于 30，就把字符串设置为::，并取消事件。下面是代码：

```
using System;
using System.IO;
using System.ComponentModel;

namespace Wrox.ProCSharp.Delegates
{
   public class BusEntity
   {
      string time = String.Empty;

      public BusEntity()
      {
         Form1.Action += new Form1.ActionEventHandler(Form1_Action);
      }

      private void Form1_Action(object sender, ActionCancelEventArgs e)
      {
         e.Cancel = !DoActions();
         if(e.Cancel)
            e.Message = "Wasn't the right time.";
      }

      private bool DoActions()
      {
         bool retVal = false;
         DateTime tm = DateTime.Now;

         if(tm.Second < 30)
         {
            time = "The time is " + DateTime.Now.ToLongTimeString();
            retVal = true;
         }
         else
            time = "";

         return retVal;
      }

      public string TimeString
      {
         get {return time;}
      }
   }
}
```

在构造函数中声明了 Form1.Action 事件的处理程序。注意其语法非常类似于前面 Click 事件的语法。由于声明事件使用的模式都是相同的，所以语法也应保持一致。还要注意如何获取 Action 事件的引用，而无需在 BusEntity 类中引用 Form1。在 Form1 类中，将 Action 事件声明为静态，这并不是必需的，但这样更易于创建处理程序。我们可以把事件声明为 public，但接着需要引用 Form1 的一个实例。

在构造函数中编写事件时，调用添加到委托列表中的方法 Form1_Action，并遵循命名标准。在处理程序中，需要确定是否取消事件。DoActions 方法根据前面描述的时间条件返回一个布尔值，并把_time 字符串设置为正确的值。

之后，把 DoActions 的返回值赋给 ActionCancelEventArgs 的 Cancel 属性。EventArgs 类一般仅在事件发送器和接收器之间来回传递值。如果取消了事件(ev.Cancel = true)，Message 属性就设置为一个字符串值，以说明事件为什么被取消。

如果再次查看 buttonRaise_Click 事件处理程序的代码，就可以看出 Cancel 属性的使用方式：

```
private void buttonRaise_Click(object sender, EventArgs e)
{
   ActionCancelEventArgs cancelEvent = new ActionCancelEventArgs();
   OnAction(this, cancelEvent);
   if(cancelEvent.Cancel)brackets
      labelInfo.Text = cancelEvent.Message;
   else
      labelInfo.Text = busEntity.TimeString;
}
```

注意创建了 ActionCancelEventArgs 对象。接着引发了事件 Action，并传递了新建的 ActionCancel EventArgs 对象。在调用 OnAction 方法，引发事件时，BusEntity 对象中 Action 事件处理程序的代码就会执行。如果还有其他对象注册了事件 Action，它们也会执行。记住，如果其他对象也处理这个事件，它们就会看到同一个 ActionCancelEventArgs 对象。如果需要确定是哪个对象取消了事件，而且如果有多个对象取消了事件，就需要在 ActionCancelEventArgs 类中包含某种基于列表的数据结构。

在与事件委托一起注册的处理程序执行完毕后，就可以查询 ActionCancelEventArgs 对象，确定它是否被取消了。如果是，labelInfo 就包含 Message 属性值；如果事件没有被取消，labelInfo 就会显示当前时间。

本节这基本上说明了如何利用事件和事件中基于 EventArgs 的对象，在应用程序中传递信息。

7.5　小结

本章介绍了委托和事件的基本知识，解释了如何声明委托，如何给委托列表添加方法，并讨论了声明事件处理程序来响应事件的过程，以及如何创建定制事件，使用引发事件的模式。

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

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

第 8 章

字符串和正则表达式

在本书的第一部分，我们一直在使用字符串，并说明 C#中 string 关键字的映射实际上指向.NET 基类 System.String。System.String 是一个功能非常强大且用途非常广泛的基类，但它不是.NET 中唯一与字符串相关的类。本章首先复习一下 System.String 的特性，再介绍如何使用其他的.NET 类来处理字符串，特别是 System.Text 和 System.Text.Regular Expressions 命名空间中的类。本章主要介绍下述内容：

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

8.1 System.String 类

在介绍其他字符串类之前，先快速复习一下 String 类上一些可用的方法。

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

字符串。

使用运算符重载可以连接字符串：

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

C#还允许使用类似于索引器的语法来提取指定的字符：

```
char char4 = message[4];   // returns 'a'. Note the char is zero-indexed
```

这个类可以完成许多常见的任务，例如替换字符、删除空白和把字母变成大写形式等。可用的方法如表 8-1 所示。

表 8-1

方 法	作 用
Compare	比较字符串的内容，考虑文化背景(区域)，确定某些字符是否相等
CompareOrdinal	与 Compare 一样，但不考虑文化背景
Concat	把多个字符串实例合并为一个实例
CopyTo	把特定数量的字符从选定的下标复制到数组的一个全新实例中
Format	格式化包含各种值的字符串和如何格式化每个值的说明符
IndexOf	定位字符串中第一次出现某个给定子字符串或字符的位置
IndexOfAny	定位字符串中第一次出现某个字符或一组字符的位置
Insert	把一个字符串实例插入到另一个字符串实例的指定索引处
Join	合并字符串数组，建立一个新字符串
LastIndexOf	与 IndexOf 一样，但定位最后一次出现的位置
LastIndexOfAny	与 IndexOfAny，但定位最后一次出现的位置
PadLeft	在字符串的开头，通过添加指定的重复字符填充字符串
PadRight	在字符串的结尾，通过添加指定的重复字符填充字符串
Replace	用另一个字符或子字符串替换字符串中给定的字符或子字符串
Split	在出现给定字符的地方，把字符串拆分为一个子字符串数组
Substring	在字符串中获取给定位置的子字符串
ToLower	把字符串转换为小写形式
ToUpper	把字符串转换为大写形式
Trim	删除首尾的空白

注意：

这个表并不完整，但可以让您明白字符串所提供的功能。

8.1.1　创建字符串

如上所述，string 类是一个功能非常强大的类，它执行许多很有用的方法。但是，string 类存在一个问题：重复修改给定的字符串，效率会很低，它实际上是一个不可变的数据类型，一旦对字符串对象进行了初始化，该字符串对象就不能改变了。表面上修改字符串内容的方法和运算符实际上是创建一个新的字符串，如果必要，可以把旧字符串的内容复制到新字符串中。例如，下面的代码：

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

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

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

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

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

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

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

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

注意：

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

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

为了解决这个问题，Microsoft 提供了 System.Text.StringBuilder 类。StringBuilder 不像 String 那样支持非常多的方法。在 StringBuilder 上可以进行的处理仅限于替换和添加或删除字符串中的文本。但是，它的工作方式非常高效。

在使用 String 类构造一个字符串时，要给它分配足够的内存来保存字符串，但 StringBuilder 通常分配的内存会比需要的更多。开发人员可以选择显式指定 StringBuilder 要分配多少内存，但如果没有显式指定，存储单元量在默认情况下就根据 StringBuilder 初始化时的字符串长度来确定。它有两个主要的属性：

- Length 指定字符串的实际长度；
- Capacity 是字符串占据存储单元的最大长度。

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

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

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

注意：

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

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

执行上面的代码，首先创建一个 StringBuilder 对象，如图 8-1 所示。

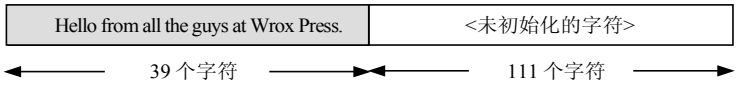

图　8-1

在调用 Append()方法时，其他文本就放在空的空间中，不需要分配更多的内存。但是，多次替换文本才能获得使用 StringBuilder 所带来的性能提高。例如，如果要以前面的方式加密文本，就可以执行整个加密过程，无须分配更多的内存：

```
StringBuilder greetingBuilder =
    new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.Append("We do hope you enjoy this book as much as we enjoyed
    writing it");
for(int i = 'z'; i>='a' ; i--)
{
   char old1 = (char)i;
   char new1 = (char)(i+1);
   greetingBuilder = greetingBuilder.Replace(old1, new1);
}

for(int i = 'Z'; i>='A' ; i--)
{
   char old1 = (char)i;
   char new1 = (char)(i+1);
   greetingBuilder = greetingBuilder.Replace(old1, new1);
}
Console.WriteLine("Encoded:\n" + greetingBuilder.ToString());
```

这段代码使用了 StringBuilder.Replace()方法，它的功能与 String.Replace()一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 StringBuilder 实例以及在最后一个 Console.WriteLine()语句中执行字符串操作期间分配的内存。

一般，使用 StringBuilder 可以执行字符串的操作，String 可以用于存储字符串或显示最终结果。

8.1.2　StringBuilder 成员

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

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

或者用给定的容量创建一个空的 StringBuilder：

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

除了前面介绍的 Length 和 Capacity 属性外，还有一个只读属性 MaxCapacity，它表

示对给定的 StringBuilder 实例的容量限制。在默认情况下，这由 int.MaxValue 给定(大约 20 亿，如前所述)。但在构造 StringBuilder 对象时，也可以把这个值设置为较低的值：

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

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

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

主要的 StringBuilder 方法如表 8-2 所示。

表 8-2

名 称	作 用
Append()	给当前字符串添加一个字符串
AppendFormat()	添加特定格式的字符串
Insert()	在当前字符串中插入一个子字符串
Remove()	从当前字符串中删除字符
Replace()	在当前字符串中，用某个字符替换另一个字符，或者用当前字符串中的一个子字符串替换另一字符串
ToString()	把当前字符串转换为 System.String 对象(在 System.Object 中被重写)

其中一些方法还有几种格式的重载方法。

注意：

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

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

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

8.1.3 格式化字符串

前面的代码示例中编写了许多类和结构，对这些类和结构执行 ToString()方法，都是为了显示给定变量的内容。但是，用户常常希望以各种可能的方式显示变量的内容，在不同的文化或地区背景中有不同的格式。.NET 基类 System.DateTime 就是最明显的一个示例：可以

把日期显示为 10 June 2006、10 Jun 2006、6/10/06 (美国)、10/6/06 (英国)或 10.06.2006 (德国)。

同样，第 3 章中编写的 Vector 结构执行 Vector.ToString()方法，是为了以(4, 56, 8)格式显示矢量。编写矢量的另一个非常常用的方式是 4i + 56j + 8k。如果要使类的用户友好性比较高，就需要使用某些工具以用户希望的方式显示它们的字符串表示。.NET 运行库定义了一种标准方式：使用接口 IFormattable，本节的主题就是说明如何把这个重要特性添加到类和结构上。

在显示一个变量时，常常需要指定它的格式，此时我们经常调用 Console.WriteLine()方法。因此，我们把这个方法作为示例，但这里的讨论适用于格式化字符串的大多数情况。例如，如果要在列表框或文本框中显示一个变量的值，一般要使用 String.Format()方法来获得该变量的合适字符串表示，但用于请求所需格式的格式说明符与传递给 Console.WriteLine()的格式相同，因此本节把 Console.WriteLine()作为一个示例来说明。首先看看在为基本类型提供格式字符串时会发生什么，再看看如何把自己的类和结构的格式说明符添加到过程中。

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

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

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

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

第 2 章简要介绍了数字类型的常见格式说明符，表 8-3 再次引用该表。

表　8-3

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

如果要在整数上加上前导 0，可以将格式说明符 0 重复所需的次数。例如，格式说明符 0000 会把 3 显示为 0003，99 显示为 0099。

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

1. 字符串的格式化

为了说明如何格式化字符串，看看执行下面的语句会得到什么结果：

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

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

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

public void WriteLine(string format, object arg0, object arg1)
{
   Console.WriteLine(string.Format(format, arg0, arg1));
}
```

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

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

下面的内容比较有趣。StringBuilder.AppendFormat()需要指出如何格式化对象，它首先检查对象，确定它是否执行 System 命名空间中的接口 IFormattable。只要试着把这个对象转换为接口，看看转换是否成功即可，或者使用 C#关键字 is，也能实现此测试。如果测试失败，AppendFormat()只会调用对象的 ToString()方法，所有的对象都从 System.Object 继承了这个方法或重写了该方法。在前面给出的编写各种类和结构的示例中，执行过程都是这样，因为我们编写的类都没有执行这个接口。这就是在前面的章节中，Object.ToString()的重写方法允许在 Console.WriteLine()语句中显示类和结构如 Vector 的原因。

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

IFormattable 只定义了一个方法，该方法也叫作 ToString()，它带有两个参数，这与 System. Object 版本的 ToString()不同，它不带参数。下面是 IFormattable 的定义：

```
interface IFormattable
{
```

```
        string ToString(string format, IFormatProvider formatProvider);
    }
```

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

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

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

本书不讨论 ToString()的第 2 个参数，它是执行接口 IFormatProvider 的对象引用。这个接口提供了 ToString()在格式化对象时需要考虑的更多信息——一般包括文化背景信息(.NET 文化背景类似于 Windows 时区，如果格式化货币或日期，就需要这些信息)。如果直接从源代码中调用这个 ToString()重载方法，就需要提供这样一个对象。但 StringBuilder. AppendFormat()为这个参数传递一个空值。如果 formatProvider 为空，ToString()就要使用系统设置中指定的文化背景信息。

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

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

整个字符串格式化过程如图 8-2 所示。

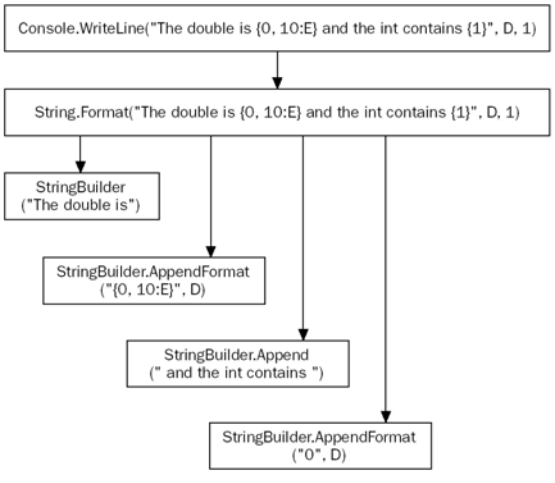

图　8-2

2. FormattableVector 示例

前面介绍了如何构造格式字符串，下面扩展本书前面的 Vector 示例，以多种方式格式化矢量。这个示例的代码可以从 www.wrox.com 上下载。只要理解了所涉及的规则，实际编写代码就相当简单了。我们只需要实现 IFormattable，提供由该接口定义的 ToString()重载方法即可。

要支持的格式说明符如下：

- N 应解释为一个请求，以提供一个数字，即矢量的模，它是其成员的平方和，在数学上等于 Vector 的长度的平方，通常放在两个竖杠的中间：||34.5||。
- VE 应解释为以科学计数法显示每个成员的一个请求，例如说明符 E 应用于 double，就可以表示为(2.3E+01, 4.5E+02, 1.0E+00)。
- IJK 应解释为以格式 23i + 450j + 1k 显示矢量的一个请求。
- 其他内容应仅返回 Vector 的默认表示方法(23, 450, 1.0)。

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

为此，首先修改 Vector 的声明，使之执行 IFormattable：

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

   // Beginning part of Vector
```

下面添加带有 2 个参数的 ToString()重载方法：

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

   string formatUpper = format.ToUpper();

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

```
            sb.AppendFormat (" k");
            return sb.ToString();
         default:
            return ToString();
      }
   }
```

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

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

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

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

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

下面用一些合适的测试代码测试可格式化的矢量：

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

运行这个示例的结果如下所示：

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

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

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

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

这说明了选用的定制格式说明符是正确的。

8.2 正则表达式

正则表达式在各种程序中都有着难以置信的作用，但并不是所有的开发人员都知道这一点。正则表达式可以看做一种有特定功能的小型编程语言：在大的字符串表达式中定位一个子字符串。它不是一种新技术，最初它是在 UNIX 环境中开发的，与 Perl 一起使用得比较多。Microsoft 把它移植到 Windows 中，到目前为止在脚本语言中用得比较多。但 System.Text.Regular- Expressions 命名空间中的许多.NET 类都支持正则表达式。.NET Framework 的各个部分都使用正则表达式，例如，在 ASP.NET 的验证服务器控件中就使用了正则表达式。

许多人都不太熟悉正则表达式语言，所以本节将主要解释正则表达式和相关的.NET 类。如果您很熟悉正则表达式，就可以跳过本节，学习.NET 基类的引用。注意，.NET 正则表达式引擎是为兼容 Perl 5 的正则表达式而设计的，但有一些新特性。

8.2.1 正则表达式概述

正则表达式语言是一种专门用于字符串处理的语言。它包含两个功能：

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

使用正则表达式，可以对字符串执行许多复杂而高级的操作，例如：

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

当然，这些都是可以在C#中用System.String和System.Text.StringBuilder的各种方法执行的任务。但是，在一些情况下，还需要编写相当多的C#代码。如果使用正则表达式，这些代码一般可以压缩为几行代码。实际上，是实例化了一个对象System.Text.Regular

Expressions. RegEx(甚至更简单：调用静态的RegEx()方法)，给它传送要处理的字符串和一个正则表达式(这是一个字符串，包含用正则表达式语言编写的指令)，就可以了。

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

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

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

8.2.2　RegularExpressionsPlayaround 示例

下面将开发一个小示例，执行并显示一些搜索的结果，说明正则表达式的一些特性，以及如何在 C#中使用.NET 正则表达式引擎。这个示例文档中使用的文本是引自另一本有关 ASP.NET 的 Wrox Press 书籍(《ASP.NET 2.0 高级编程》，清华大学出版社翻译出版)：

```
string Text =
@"This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entirely revised and updated for the 2.0
Release of .NET, this book will give you the information you need to master ASP.NET
and build a dynamic, successful, enterprise Web application.";
```

注意：

不考虑换行，则上面的表达式是合法的 C#代码——说明了使用字符串时应在前面加上符号@。

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

```
string Pattern = "ion";
MatchCollection Matches = Regex.Matches(Text, Pattern,
                                RegexOptions.IgnoreCase |
                                RegexOptions.ExplicitCapture);
foreach (Match NextMatch in Matches)
{
   Console.WriteLine(NextMatch.Index);
}
```

在这段代码中，使用了 System.Text.RegularExpressions 命名空间中 Regex 类的静态方法 Matches()。这个方法的参数是一些输入文本、一个模式和 RegexOptions 枚举中的一组可选标志。在本例中，指定所有的搜索都不应区分大小写。另一个标记 ExplicitCapture 改变了收集匹配的方式，对于本例，这样可以使搜索的效率更高，其原因详见后面的内容(尽管它还有这里没有介绍的其他用法)。Matches()返回 MatchCollections 对象的引用。匹配是一个技术术语，表示在表达式中查找模式实例的结果，用 System.Text.RegularExpressions.Match 来代表。因此，我们返回一个包含所有匹配的 MatchCollection，每个匹配都用一个 Match 对象来表示。在上面的代码中，只是在集合中迭代，使用 Match 类的 Index 属性，返回输入文本中匹配所在的索引。运行这段代码，将得到 3 个匹配。表 8-4 描述了 RegexOptions 枚举的一些选项。

表 8-4

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

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

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

```
string Pattern = @"\bn";
MatchCollection Matches = Regex.Matches(Text, Pattern,
                                RegexOptions.IgnoreCase |
                                RegexOptions.ExplicitCapture);
```

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

```
string Pattern = @"ion\b";
```

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

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

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

表 8-5 是可以使用的一些主要的特定字符或转义序列，但这个表并不完整，完整的列表请参考 MSDN 文档。

表　8-5

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

如果要搜索一个元字符，也可以通过带有反斜杠的转义字符来表示。例如，.(一个句点)表示除了换行字符以外的任何字符，而\.表示一个点。

可以把替换的字符放在方括号中，请求匹配包含这些字符。例如，[1|c]表示字符可以是 l 或 c。如果要搜索 map 或 man，可以使用序列 ma[n|p]。在方括号中，也可以指定一个范围，例如[a-z]表示所有的小写字母，[A-E]表示 A 到 E 之间的所有大写字母，[0-9]表示一个数字。如果要搜索一个整数(该序列只包含 0 到 9 的字符)，就可以编写[0-9]+(注意，使用+字符表示至少要有这样一个数字，但可以有多个数字，所以 9、 83 和 854 等都是匹配的)。

8.2.3 显示结果

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

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

```
static void WriteMatches(string text, MatchCollection matches)
{
   Console.WriteLine("Original text was: \n\n" + text + "\n");
   Console.WriteLine("No. of matches: " + matches.Count);
   foreach (Match nextMatch in matches)
   {
      int Index = nextMatch.Index;
      string result = nextMatch.ToString();
      int charsBefore = (Index < 5) ? Index : 5;
      int fromEnd = text.Length - Index - result.Length;
      int charsAfter = (fromEnd < 5) ? fromEnd : 5;
      int charsToDisplay = charsBefore + charsAfter + result.Length;

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

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

```
static void Find2()
{
string Text =
@"This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entity revised and updated for the 1.1
Release of .NET, this book will give you the information you need to master ASP.NET
And build a dynamic, successful, enterprise Web application.”;
```

```
    string pattern = @"\ba";
    MatchCollection matches = Regex.Matches(text, pattern,
      RegexOptions.IgnoreCase);
    WriteMatches(text, matches);
}
```

下面是一个简单的 Main()方法，可以编辑并选择一个 Find<n>()方法：

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

这段代码还使用了命名空间 RegularExpressions：

```
using System;
using System.Text.RegularExpressions;
```

运行带有 Find1()方法的示例，得到如下所示的结果：

```
RegularExpressionsPlayaround
Original text was:

This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entity revised and updated for the 1.1
Release of .NET, this book will give you the information you need to master ASP.NET
And build a dynamic, successful, enterprise Web application.

No. of matches: 1
Index: 291,     String: application,    Web application.
```

8.2.4　匹配、组合和捕获

正则表达式的一个很好的特性是可以把字符组合起来，其方式与 C#中的复合语句一样。在 C#中，可以把任意数量的语句放在花括号中，把它们组合在一起。其结果就像一个复合语句那样。在正则表达式模式中，也可以把任何字符组合起来(包括元字符和转义序列)，像处理一个字符那样处理它们。唯一的区别是要使用圆括号，而不是花括号，得到的序列称为一个组。

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

注意：

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

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

例如，URI 的格式是<protocol>://<address>:<port>，其中端口是可选的。它的一个示例是 http://www.wrox.com:4355。假定要从一个 URI 中提取协议、地址和端口，而且紧邻 URI 的后面可能有空白(但没有标点符号)，就可以使用下面的表达式：

```
\b(\S+)://(\S+)(?::(\S+))?\b
```

该表达式的工作方式如下：首先，前导和尾部的\b 序列确保只需要考虑完全是字的文本部分，在这个文本部分中，第一组(\S+)://会选择一个或多个不是空白的字符，其后是://。在 HTTP URI 的开头会选择出 http://。花括号表示把 http 存储为一个组。后面的序列(\S+)则在上述 URI 中选择 www.wrox.com，这个组在遇到词的结尾(结束\b)时或标记另一个组的冒号(：)时结束。

下一个组选择端口(本例是:4355)。后面的？表示这个组在匹配中是可选的，如果没有:xxxx，也不会妨碍匹配的标记。这是非常重要的，因为端口号在 URI 中一般不指定，实际上，在大多数情况下，URI 是没有端口号的。但是，事情会比较复杂。我们希望指定冒号可以出现，也可以不出现，但不希望把这个冒号也存储在组中。为此，可以嵌套两个组：内部的(\S+)组选择冒号后面的内容(本例中是 4355)，外面的组包含内部的组，前面是一个冒号，该组又在序列?:的后面。这个序列表示该组不应保存(只需要保存 4355，不需要保存:4355)。不要把这两个冒号混淆了，第一个冒号是序列?:的一部分，表示不保存这个组，第二个冒号是要搜索的文本。

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

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

在这个匹配中，找到了刚才提及的 3 个组，还有第四个组表示匹配本身。理论上，每个组都可以选择 0 次、1 次或多次匹配。单个的匹配就称为捕获。在第一个组(\S+)中，有一个捕获 http，第二个组也有一个捕获 www.wrox.com，但第三个组没有捕获，因为在这个 URI 中没有端口号。

注意，该字符串包含第二个 http://。虽然它匹配于第一个组，但不会被搜索出来，因为整个搜索表达式不匹配于这部分文本。

前面没有介绍使用组和捕获的任何 C#示例，下面提到的.NET 类 RegularExpressions 就通过 Group 和 Capture 类支持组和捕获。GroupCollection 和 CaptureCollection 分别表示组和捕获的集合，Match 类有一个方法 Groups()，它返回相应的 GroupCollection 对象，Group 类也相应地执行一个方法 Captures()，它返回 CaptureCollection 对象。这些对象之间的关系如图 8-3 所示。

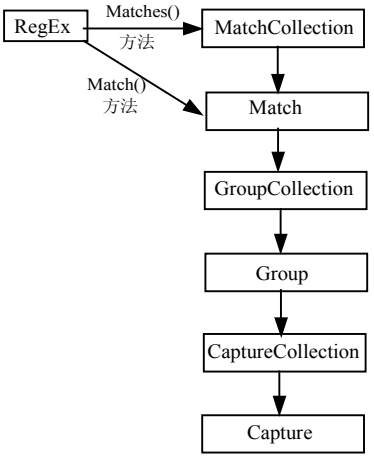

图　8-3

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

8.3　小结

在使用.NET Framework 时，可用的数据类型相当多。在应用程序(特别是关注数据提交和检索的应用程序)中，最常用的一个类型就是 Sting 数据类型。String 非常重要，这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理 String 数据类型的原因。

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

最后，使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种极佳工具。

第 9 章 泛 型

C#语言和 CLR 的一个最大变化是引入了泛型。在.NET 1.0 中，要创建一个灵活的类或方法，但该类或方法在编译期间不知道使用什么类，就必须以 Object 类为基础。而 Object 类在编译期间没有类型安全性，因此必须进行强制类型转换。另外，给值类型使用 Object 类会有性能损失。

.NET 2.0 提供了泛型。有了泛型，就不再需要 Object 类了。泛型类使用泛型类型，并可以根据需要用特定的类型替换泛型类型。这就保证了类型安全性：如果某个类型不支持泛型类，编译器就会生成错误。

泛型是一个很强大的特性，对于集合类而言尤其如此。.NET 1.0 中的大多数集合类都基于 Object 类型。.NET 2.0 提供了实现为泛型的新集合类。

泛型不仅限于类，本章还将介绍用于委托、接口和方法的泛型。

本章的主要内容如下：

- 泛型概述
- 创建泛型类
- 泛型类的特性
- 泛型接口
- 泛型方法
- 泛型委托
- Framework 的其他泛型类型

9.1 概述

泛型并不是一个全新的结构，其他语言中有类似的概念。例如，C++模板就与泛型相当。但是，C++模板和.NET 泛型之间有一个很大的区别。对于 C++模板，在用特定的类型实例化模板时，需要模板的源代码。相反，泛型不仅是 C#语言的一种结构，而且是 CLR 定义的。所以，即使泛型类是在 C#中定义的，也可以在 Visual Basic 中用一个特定的类型

实例化该泛型。

下面介绍泛型的优点和缺点，尤其是：

- 性能
- 类型安全性
- 二进制代码重用
- 代码的扩展
- 命名约定

9.1.1 性能

泛型的一个主要优点是性能。第 10 章介绍了 System.Collections 和 System.Collections.Generic 命名空间的泛型和非泛型集合类。对值类型使用非泛型集合类，在把值类型转换为引用类型，和把引用类型转换为值类型时，需要进行装箱和拆箱操作。

注意：

装箱和拆箱详见第 6 章，这里仅简要复习一下这些术语。

值类型存储在堆栈上，引用类型存储在堆上。C#类是引用类型，结构是值类型。.NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，int 可以赋予一个对象。从值类型转换为引用类型称为装箱。如果方法需要把一个对象作为参数，而且传送了一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以使用拆箱操作转换为值类型。在拆箱时，需要使用类型转换运算符。

下面的例子显示了 System.Collections 命名空间中的 ArrayList 类。ArrayList 存储对象，Add()方法定义为需要把一个对象作为参数，所以要装箱一个整数类型。在读取 ArrayList 中的值时，要进行拆箱，把对象转换为整数类型。可以使用类型转换运算符把 ArrayList 集合的第一个元素赋予变量 i1，在访问 int 类型的变量 i2 的 foreach 语句中，也要使用类型转换运算符：

```
ArrayList list = new ArrayList();
list.Add(44);   // boxing – convert a value type to a reference type

int i1 = (int)list[0];   // unboxing – convert a reference type to a value type

foreach (int i2 in list)
{
   Console.WriteLine(i2);   // unboxing
}
```

装箱和拆箱操作很容易使用，但性能损失比较大，迭代许多项时尤其如此。

System.Collections.Generic 命名空间中的 List<T>类不使用对象，而是在使用时定义类型。在下面的例子中，List<T>类的泛型类型定义为 int，所以 int 类型在 JIT 编译器动态生成的类中使用，不再进行装箱和拆箱操作：

```
List<int> list = new List<int>();
list.Add(44);   // no boxing - value types are stored in the List<int>

int i1 = list[0];   // no unboxing, no cast needed

foreach (int i2 in list)
{
   Console.WriteLine(i2);
}
```

9.1.2 类型安全

泛型的另一个特性是类型安全。与 ArrayList 类一样，如果使用对象，可以在这个集合中添加任意类型。下面的例子在 ArrayList 类型的集合中添加一个整数、一个字符串和一个 MyClass 类型的对象：

```
ArrayList list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

如果这个集合使用下面的 foreach 语句迭代，而该 foreach 语句使用整数元素来迭代，编译器就会编译这段代码。但并不是集合中的所有元素都可以转换为 int，所以会出现一个运行异常：

```
foreach (int i in list)
{
   Console.WriteLine(i);
}
```

错误应尽早发现。在泛型类 List<T>中，泛型类型 T 定义了允许使用的类型。有了 List<int>的定义，就只能把整数类型添加到集合中。编译器不会编译这段代码，因为 Add() 方法的参数无效：

```
List<int> list = new List<int>();
list.Add(44);
list.Add("mystring");   // compile time error
list.Add(new MyClass());   // compile time error
```

9.1.3 二进制代码的重用

泛型允许更好地重用二进制代码。泛型类可以定义一次，用许多不同的类型实例化。不需要像 C++模板那样访问源代码。

例如，System.Collections.Generic 命名空间中的 List<T>类用一个 int、一个字符串和一个 MyClass 类型实例化：

```
List<int> list = new List<int>();
list.Add(44);

List<string> stringList = new List<string>();
stringList.Add("mystring");

List<MyClass> myclassList = new List<MyClass>();
myClassList.Add(new MyClass());
```

泛型类型可以在一种语言中定义，在另一种.NET 语言中使用。

9.1.4 代码的扩展

在用不同的类型实例化泛型时，会创建多少代码？

因为泛型类的定义会放在程序集中，所以用某个类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JIT 编译器把泛型类编译为内部码时，会给每个值类型创建一个新类。引用类型共享同一个内部类的所有实现代码。这是因为引用类型在实例化的泛型类中只需要 4 字节的内存单元(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中。而每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

9.1.5 命名约定

如果在程序中使用泛型，区分泛型类型和非泛型类型会有一定的帮助。下面是泛型类型的命名规则：

- 泛型类型的名称用字母 T 作为前缀。
- 如果没有特殊的要求，泛型类型允许用任意类替代，且只使用了一个泛型类型，就可以用字符 T 作为泛型类型的名称。

```
public class List<T> { }

public class LinkedList<T> { }
```

- 如果泛型类型有特定的要求(例如必须实现一个接口或派生于基类)，或者使用了两个或多个泛型类型，就应给泛型类型使用描述性的名称：

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

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

public class SortedList<TKey, TValue> { }
```

9.2 创建泛型类

首先介绍一个一般的、非泛型的简化链表类，它可以包含任意类型的对象，以后再把

这个类转化为泛型类。

在链表中，一个元素引用其后的下一个元素。所以必须创建一个类，将对象封装在链表中，引用下一个对象。类 LinkedListNode 包含一个对象 value，它用构造函数初始化，还可以用 Value 属性读取。另外，LinkedListNode 类包含对链表中下一个元素和上一个元素的引用，这些元素都可以从属性中访问。

```
public class LinkedListNode
{
    private object value;
    public LinkedListNode(object value)
    {
        this.value = value;
    }

    public object Value
    {
        get { return value; }
    }

    private LinkedListNode next;
    public LinkedListNode Next
    {
        get { return next; }
        internal set { next = value; }
    }

    private LinkedListNode prev;
    public LinkedListNode Prev
    {
        get { return prev; }
        internal set { prev = value; }
    }
}
```

LinkedList 类包含 LinkedListNode 类型的 first 和 last 字段，它们分别标记了链表的头尾。AddLast()方法在链表尾添加一个新元素。首先创建一个 LinkedListNode 类型的对象。如果链表是空的，则 first 和 last 字段就设置为该新元素；否则，就把新元素添加为链表中的最后一个元素。执行 GetEnumerator()方法时，可以用 foreach 语句迭代链表。GetEnumerator()方法使用 yield 语句创建一个枚举器类型。

提示：

yield 语句参见第 5 章。

```
public class LinkedList : IEnumerable
{
    private LinkedListNode first;
    public LinkedListNode First
    {
```

```
            get { return first; }
        }

        private LinkedListNode last;
        public LinkedListNode Last
        {
            get { return last; }
        }

        public LinkedListNode AddLast(object node)
        {
            LinkedListNode newNode = new LinkedListNode(node);
            if (first == null)
            {
                first = newNode;
                last = first;
            }
            else
            {
                last.Next = newNode;
                last = newNode;
            }
            return newNode;
        }

        public IEnumerator GetEnumerator()
        {
            LinkedListNode current = first;
            while (current != null)
            {
                yield return current.Value;
                current = current.Next;
            }
        }
    }
```

现在可以给任意类型使用 LinkedList 类了。在下面的代码中，实例化了一个新 LinkedList 对象，添加了两个整数类型和一个字符串类型。整数类型要转换为一个对象，所以执行装箱操作，如前面所述。在 foreach 语句中执行拆箱操作。在 foreach 语句中，链表中的元素被强制转换为整数，所以对于链表中的第三个元素，会发生一个运行异常，因为它转换为 int 时会失败。

```
    LinkedList list1 = new LinkedList();
    list1.AddLast(2);
    list1.AddLast(4);
    list1.AddLast("6");
    foreach (int i in list1)
    {
    Console.WriteLine(i);
    }
```

下面创建链表的泛型版本。泛型类的定义与一般类类似，只是要使用泛型类型声明。之后，泛型类型就可以在类中用作一个字段成员，或者方法的参数类型。LinkedListNode 类用一个泛型类型 T 声明。字段 value 的类型是 T，而不是 object。构造函数和 Value 属性也变为接受和返回 T 类型的对象。也可以返回和设置泛型类型，所以属性 Next 和 Prev 的类型是 LinkedListNode<T>。

```
public class LinkedListNode<T>
{
    private T value;
    public LinkedListNode(T value)
    {
        this.value = value;
    }

    public T Value
    {
        get { return value; }
    }

    private LinkedListNode<T> next;
    public LinkedListNode<T> Next
    {
        get { return next; }
        internal set { next = value; }
    }

    private LinkedListNode<T> prev;
    public LinkedListNode<T> Prev
    {
        get { return prev; }
        internal set { prev = value; }
    }
}
```

LinkedList 类也改为泛型类。LinkedList<T>包含 LinkedListNode<T>元素。LinkedList 中的类型 T 定义了类型 T 的包含字段 first 和 last。AddLast()方法现在接受类型 T 的参数，实例化 LinkedListNode<T>类型的对象。

在.NET 2.0 推出后，IEnumerable 接口也有一个泛型版本 IEnumerable<T>。IEnumerable<T>派生于 IEnumerable，添加了返回 IEnumerator<T>的 GetEnumerator()方法，LinkedList<T>执行泛型接口 IEnumerable<T>。

提示：

枚举、接口 IEnumerable 和 IEnumerator 详见第 5 章。

```
public class LinkedList<T> : IEnumerable<T>
{
    private LinkedListNode<T> first;
    public LinkedListNode<T> First
```

```
        {
            get { return first; }
        }

        private LinkedListNode<T> last;
        public LinkedListNode<T> Last
        {
            get { return last; }
        }

        public LinkedListNode<T> AddLast(T node)
        {
            LinkedListNode<T> newNode = new LinkedListNode<T>(node);
            if (first == null)
            {
                first = newNode;
                last = first;
            }
            else
            {
                last.Next = newNode;
                last = newNode;
            }
            return newNode;
        }

        public IEnumerator<T> GetEnumerator()
        {
            LinkedListNode<T> current = first;

            while (current != null)
            {
                yield return current.Value;
                current = current.Next;
            }
          }

          IEnumerator IEnumerable.GetEnumerator()
          {
             return GetEnumerator();
          }
    }
```

使用泛型类 LinkedList<T>，可以用 int 类型实例化它，且无需装箱操作。如果不使用 AddLast()方法传送 int，就会出现一个编译错误。使用泛型 IEnumerable<T>，foreach 语句也是类型安全的，如果 foreach 语句中的变量不是 int，也会出现一个编译错误。

```
    LinkedList<int> list2 = new LinkedList<int>();
    list2.AddLast(1);
    list2.AddLast(3);
```

```
    list2.AddLast(5);
    foreach (int i in list2)
    {
        Console.WriteLine(i);
    }
```

同样，可以给泛型 LinkedList<T>使用 string 类型，将字符串传送给 AddLast()方法。

```
    LinkedList<string> list3 = new LinkedList<string>();
    list3.AddLast("2");
    list3.AddLast("four");
    list3.AddLast("foo");

    foreach (string s in list3)
    {
        Console.WriteLine(s);
    }
```

提示：

每个处理对象类型的类都可以有泛型实现方式。另外，如果类使用了继承，泛型非常有助于去除类型转换操作。

9.3 泛型类的特性

在创建泛型类时，需要一些其他 C#关键字。例如，不能把 null 赋予泛型类型。此时，可以使用 default 关键字。如果泛型类型不需要 Object 类的功能，但需要调用泛型类上的某些特定方法，就可以定义约束。

本节讨论如下主题：

- 默认值
- 约束
- 继承
- 静态成员

下面开始一个使用泛型文档管理器的示例。文档管理器用于从队列中读写文档。先创建一个新的控制台项目 DocumentManager，添加类 DocumentManager<T>。AddDocument()方法将一个文档添加到队列中。如果队列不为空，IsDocumentAvailable 只读属性就返回 true。

```
    using System;
    using System.Collections.Generic;

    namespace Wrox.ProCSharp.Generics
    {
        public class DocumentManager<T>
        {
            private readonly Queue<T> documentQueue = new Queue<T>();
```

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

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

9.3.1 默认值

现在给 DocumentManager<T>类添加一个 GetDocument()方法。在这个方法中，给类型 T 指定 null。但是，不能把 null 赋予泛型类型。原因是泛型类型也可以实例化为值类型，而 null 只能用于引用类型。为了解决这个问题，可以使用 default 关键字。通过 default 关键字，将 null 赋予引用类型，将 0 赋予值类型。

```
public T GetDocument()
{
    T doc = default(T);
    lock (this)
    {
    doc = documentQueue.Dequeue();
    }
    return doc;
}
```

注意：

default 关键字根据上下文可以有多种含义。switch 语句使用 default 定义默认情况。在泛型中，根据泛型类型是引用类型还是值类型，default 关键字用于将泛型类型初始化为 null 或 0。

9.3.2 约束

如果泛型类需要调用泛型类型上的方法，就必须添加约束。对于 DocumentManager<T>，文档的标题应在 DisplayAllDocuments()方法中显示。

Document 类执行带有 Title 和 Content 属性的 IDocument 接口：

```
public interface IDocument
{
    string Title { get; set; }
    string Content { get; set; }
}

public class Document : IDocument
{
    public Document()
    {
    }

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

    private string title;
    public string Title
    {
        get { return title; }
        set { title = value; }
    }

    private string content;
    public string Content
    {
        get { return content; }
        set { content = value; }
    }
}
```

要使用 DocumentManager<T>类显示文档，可以将类型 T 强制转换为 IDocument 接口，以显示标题：

```
public void DisplayAllDocuments()
{
    foreach (T doc in documentQueue)
    {
        Console.WriteLine((IDocument)doc).Title);
    }
}
```

问题是，如果类型 T 没有执行 IDocument 接口，这个类型转换就会生成一个运行异常。最好给 DocumentManager<TDocument>类定义一个约束：TDocument 类型必须执行 IDocument 接口。为了在泛型类型的名称中指定该要求，将 T 改为 TDocument。where 子句指定了执行 IDocument 接口的要求。

```
public class DocumentManager<TDocument>
    where TDocument : IDocument
{
```

这样，就可以编写 foreach 语句，让类型 T 包含属性 Title 了。Visual Studio IntelliSense 和编译器都会提供这个支持。

```
public void DisplayAllDocuments()
{
    foreach (TDocument doc in documentQueue)
    {
        Console.WriteLine(doc.Title);
    }
}
```

在 Main()方法中，DocumentManager<T>类用 Document 类型实例化，而 Document 类型执行了需要的 IDocument 接口。接着添加和显示新文档，检索其中一个文档：

```
static void Main()
{
    DocumentManager<Document> dm = new DocumentManager<Document>();
    dm.AddDocument(new Document("Title A", "Sample A"));
    dm.AddDocument(new Document("Title B", "Sample B"));

    dm.DisplayAllDocuments();

    if (dm.IsDocumentAvailable)
    {
        Document d = dm.GetDocument();
        Console.WriteLine(d.Content);
    }
}
```

DocumentManager 现在可以处理任何执行了 IDocument 接口的类。

在示例应用程序中，介绍了接口约束。还有几种约束类型，如表 9-1 所示。

表 9-1

约 束	说 明
where T : struct	使用结构约束，类型 T 必须是值类型
where T : class	类约束指定，类型 T 必须是引用类型
where T : IFoo	指定类型 T 必须执行接口 IFoo
where T : Foo	指定类型 T 必须派生于基类 Foo
where T : new()	这是一个构造函数约束，指定类型 T 必须有一个默认构造函数
where T : U	这个约束也可以指定，类型 T 派生于泛型类型 V。该约束也称为裸类型约束

注意：

在 CLR 2.0 中，只能为默认构造函数定义约束，不能为其他构造函数定义约束。

使用泛型类型还可以合并多个约束。where T : IFoo，new()约束和 MyClass<T>声明指定，类型 T 必须执行 IFoo 接口，且必须有一个默认构造函数。

```
public class MyClass<T>
   where T : IFoo, new()
{
   //...
```

提示：

在 C# 2.0 中，where 子句的一个重要限制是，不能定义必须由泛型类型执行的运算符。运算符不能在接口中定义。在 where 子句中，只能定义基类、接口和默认构造函数。

9.3.3 继承

前面创建的 LinkedList<T>类执行了 IEnumerable<T>接口：

```
public class LinkedList<T> : IEnumerable<T>
{
   //...
```

泛型类型可以执行泛型接口，也可以派生于一个类。泛型类可以派生于泛型基类：

```
public class Base<T>
{
}

public class Derived<T> : Base<T>
{
}
```

其要求是必须重复接口的泛型类型，或者必须指定基类的类型，如下所示：

```
public class Base<T>
{
}
public class Derived<T> : Base<string>
{
}
```

于是，派生类可以是泛型类或非泛型类。例如，可以定义一个抽象的泛型基类，它在派生类中用一个具体的类型实现。这允许对特定类型执行特殊的操作：

```
public abstract class Calc<T>
{
   public abstract T Add(T x, T y);
   public abstract T Sub(T x, T y);
}
public class SimpleCalc : Calc<int>
{
   public override int Add(int x, int y)
```

```
        {
            return x + y;
        }

        public override int Sub(int x, int y)
        {
            return x - y;
        }
    }
```

9.3.4 静态成员

泛型类的静态成员需要特别关注。泛型类的静态成员只能在类的一个实例中共享。下面看一个例子。StaticDemo<T>类包含静态字段 x：

```
public class StaticDemo<T>
{
public static int x;
}
```

由于对一个 string 类型和一个 int 类型使用了 StaticDemo<T>类，所以存在两组静态字段：

```
StaticDemo<string>.x = 4;
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // writes 4
```

9.4 泛型接口

使用泛型可以定义接口，接口中的方法可以带泛型参数。在链表示例中，就执行了 IEnumerable<T>接口，它定义了 GetEnumerator()方法，以返回 IEnumerator<T>。对于.NET 1.0 中的许多非泛型接口，.NET 2.0 定义了新的泛型版本，例如 IComparable<T>：

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

第 5 章中的非泛型接口 IComparable 需要一个对象，Person 类的 CompareTo()方法才能按姓氏给人员排序：

```
public class Person : IComparable
{
    public int CompareTo(object obj)
    {
        Person other = obj as Person;
        return this.lastname.CompareTo(other.lastname);
    }
//...
```

执行泛型版本时，不再需要将 object 的类型强制转换为 Person：

```
public class Person : IComparable<Person>
{
    public int CompareTo(Person other)
    {
        return this.lastname.CompareTo(other.lastname);
    }
//...
```

9.5 泛型方法

除了定义泛型类之外，还可以定义泛型方法。在泛型方法中，泛型类型用方法声明来定义。

Swap<T>方法把 T 定义为泛型类型，用于两个参数和一个变量 temp：

```
void Swap<T>(ref T x, ref T y)
{
   T temp;
   temp = x;
   x = y;
   y = temp;
}
```

把泛型类型赋予方法调用，就可以调用泛型方法：

```
int i = 4;
int j = 5;
Swap<int>(ref i, ref j);
```

但是，因为 C#编译器会通过调用 Swap 方法来获取参数的类型，所以不需要把泛型类型赋予方法调用。泛型方法可以像非泛型方法那样调用：

```
int i = 4;
int j = 5;
Swap(ref i, ref j);
```

下面的例子使用泛型方法累加集合中的所有元素。为了说明泛型方法的功能，下面的 Account 类包含 name 和 balance：

```
   public class Account
   {
      private string name;
      public string Name
      {
         get
         {
            return name;
         }
```

```
        }

        private decimal balance;
        public decimal Balance
        {
           get
           {
              return balance;
           }
        }

        public Account(string name, Decimal balance)
        {
           this.name = name;
           this.balance = balance;
        }
     }
```

应累加结余的所有账目操作都添加到 List<Account>类型的账目列表中：

```
           List<Account> accounts = new List<Account>();
           accounts.Add(new Account("Christian", 1500));
           accounts.Add(new Account("Sharon", 2200));
           accounts.Add(new Account("Katie", 1800));
```

累加所有 Account 对象的传统方式是用 foreach 语句迭代所有的 Account 对象，如下所示。foreach 语句使用 IEnumerable 接口迭代集合的元素，所以 AccumulateSimple()方法的参数是 IEnumerable 类型。这样，AccumulateSimple()方法就可以用于所有实现 IEnumerable 接口的集合类。在这个方法的实现代码中，直接访问 Account 对象的 Balance 属性：

```
     public static class Algorithm
     {
        public static decimal AccumulateSimple(IEnumerable e)
        {
           decimal sum = 0;
           foreach (Account a in e)
           {
              sum += a.Balance;
           }
           return sum;
        }
     }
```

Accumulate()方法的调用方式如下：

```
           decimal amount = Algorithm.AccumulateSimple(accounts);
```

第一个实现代码的问题是，它只能用于 Account 对象。使用泛型方法就可以避免这个问题。

Accumulate()方法的第二个版本接受实现了 IAccount 接口的任意类型。如前面的泛型

类所述，泛型类型可以用 where 子句来限制。这个子句也可以用于泛型方法。Accumulate()方法的参数改为 IEnumerable<T>。IEnumerable<T>是 IEnumerable 接口的泛型版本，由泛型集合类实现。

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> coll)
      where TAccount : IAccount
{
   decimal sum = 0;

   foreach (TAccount a in coll)
   {
      sum += a.Balance;
   }
   return sum;
}
```

将 Account 类型定义为泛型类型参数，就可以调用新的 Accumulate()方法：

```
decimal amount = Algorithm.Accumulate<Account>(accounts);
```

因为编译器会从方法的参数类型中自动推断出泛型类型参数，所以以如下方式调用 Accumulate()方法是有效的：

```
decimal amount = Algorithm.Accumulate(accounts);
```

泛型类型实现 IAccount 接口的要求过于严厉。这个要求可以使用泛型委托来改变。在下一节中，Accumulate()方法将改为独立于任何接口。

9.6 泛型委托

如第 7 章所述，委托是类型安全的方法引用。通过泛型委托，委托的参数可以在以后定义。

.NET 2.0 定义了一个泛型委托 EventHandler，它的第二个参数是 TEventArgs 类型，所以不再需要为每个新参数类型定义新委托了。

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

9.6.1 执行委托调用的方法

把 Accumulate()方法改为有两个泛型类型。TInput 是要累加的对象类型，TSummary 是返回类型。Accumulate 的第一个参数是 IEnumerable<T>接口，这与以前相同。第二个参数需要 Action 委托引用一个方法，来累加所有的结余。

在实现代码中，现在给每个元素调用 Action 委托引用的方法，再返回计算的总和：

```
public delegate TSummary Action<TInput, TSummary>(TInput t, TSummary u);
```

```
public static TSummary Accumulate<TInput, TOutput>(IEnumerable<T> coll,
     Action<TInput, TSummary> action)
{
   TSummary sum = default(TSummary);

   foreach (TInput input in coll)
   {
      sum = action(input, sum);
   }
   return sum;
}
```

Accumulate 方法可以通过匿名方法调用，该匿名方法指定，账目的结余应累加到第二个参数中：

```
decimal amount = Accumulate<Account, decimal>(
     accounts, delegate(Account a, decimal d) { return a.Balance + d; });
```

如果 Account 结余的累加需要进行多次，就可以把该功能放在一个 AccountAdder()方法中：

```
static decimal AccountAdder(Account a, decimal d)
{
   return a.Balance + d;
}
```

联合使用 AccountAdder 方法和 Accumulate 方法：

```
decimal amount = Accumulate<Account, decimal>(
     accounts, AccountAdder);
```

Action 委托引用的方法可以实现任何逻辑。例如，可以进行乘法操作，而不是加法操作。

Accumulate()方法和 AccumulateIf()方法一起使用，会更灵活。在 AccumulateIf()中，使用了另一个 Predicate<T>类型的参数。Predicate<T>委托引用的方法会检查某个账目是否应累加进去。在 foreach 语句中，只有谓词 match 返回 true，才会调用 action 方法：

```
public static TSummary AccumulateIf<TInput, TSummary>(
     IEnumerable<TInput> coll,
     Action<TInput, TSummary> action,
     Predicate<TInput> match)
{
   TSummary sum = default(TSummary);

   foreach (TInput a in coll)
   {
      if (match(a))
      {
         sum = action(a, sum);
```

```
            }
        }

        return sum;
    }
```

调用 AccumulateIf()方法可以实现累加和执行谓词。这里只累加结余大于 2000 的账目：

```
        decimal amount = Algorithm.AccumulateIf<Account, decimal>(
          accounts,
          delegate(Account a, decimal d) { return a.Balance + d; },
          delegate(Account a) {return a.Balance > 2000 ? });
```

9.6.2 对 Array 类使用泛型委托

第 5 章使用 IComparable 和 IComparer 接口，演示了 Array 类的几个排序技术。从.NET 2.0 开始，Array 类的一些方法把泛型委托类型用作参数。表 9-2 列出了这些方法、泛型类型和功能。

表 9-2

方 法	泛型参数类型	说 明
Sort()	int Comparison<T>(T x, T y)	Sort()方法定义了几个重载版本。其中一个重载版本需要一个 Comparison<T>类型的参数。Sort()使用委托引用的方法对集合中的所有元素排序
ForEach()	void Action<T>(T obj)	ForEach()方法对集合中的每一项调用由 Action<T>委托引用的方法
FindAll() Find() FindLast() FindIndex() FindLastIndex()	bool Predicate<T>(T match)	FindXXX()方法将 Predicate<T>委托作为其参数接受。由委托引用的方法会调用多次，并一个接一个地传送集合中的元素。Find()方法在谓词第一次返回 true 时停止搜索，并返回这个元素。FindIndex()返回查找到的第一个元素的索引。FindLast()和 FindLastIndex()以逆序方式对集合中的元素调用谓词，因此返回最后一项或最后一个索引。FindAll()返回一个新列表，其中包含谓词为 true 的所有项
ConvertAll()	TOutput Converter<TInput, TOutput>(TInput input)	ConvertAll()方法给集合中的每个元素调用 Converter<TInput, TOutput>委托，返回一列转换好的元素
TrueForAll()	bool Predicate<T>(T match)	TrueForAll()方法给每个元素调用谓词委托。如果谓词给每个元素都返回 true，则 TrueForAll()也返回 true。如果谓词为一个元素返回了 false，TrueForAll()就返回 false

下面看看如何使用这些方法。

Sort()方法把这个委托作为参数接受：

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

这样，就可以使用匿名委托传送两个 Person 对象，给数组排序。对于 Person 对象数组，参数 T 是 Person 类型：

```
Person[] persons = {
    new Person("Emerson", "Fittipaldi"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Michael", "Schumacher")
};
Array.Sort(persons,
    delegate(Person p1, Person p2)
    {
        return p1.Firstname.CompareTo(p2.Firstname);
    });
```

Array.ForEach()方法将 Action<T>委托作为参数，给数组的每个元素执行操作：

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

于是，就可以传送 Console.WriteLine 方法的地址，将每个人写入控制台。WriteLine()方法的一个重载版本将 Object 类作为参数类型。由于 Person 派生于 Object，所以它适合于 Person 数组：

```
Array.ForEach(persons, Console.WriteLine);
```

ForEach()语句的结果将 persons 变量引用的集合中的每个人都写入控制台：

```
Emerson Fittipaldi
Niki Lauda
Ayrton Senna
Michael Schumacher
```

如果需要更多的控制，则可以传送一个匿名方法，其参数应匹配委托定义的参数：

```
Array.ForEach(persons,
    delegate (Person p)
    {
        Console.WriteLine("{0}", p.Lastname);
    });
```

下面是写入控制台的姓氏：

```
Fittipaldi
Lauda
Senna
Schumacher
```

Array.FindAll()方法需要 Predicate<T>委托：

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

Array.FindAll()方法为数组中的每个元素调用谓词，并返回一个谓词是 true 的数组。在这个例子中，对于 Lastname 以字符串"S"开头的所有 Person 对象，都返回 true。

```
Person[] sPersons = Array.FindAll(persons,
    delegate (Person p)
    {
        return p.Lastname.StartsWith("S");
    });
```

迭代返回的集合 sPersons，并写入控制台，结果如下：

```
Ayrton Senna
Michael Schumacher
```

Array.ConvertAll()方法使用泛型委托 Converter 和两个泛型类型。第一个泛型类型 TInput 是输入参数，第二个泛型类型 TOutput 是返回类型。

```
public delegate TOutput Converter<TInput, TOutput>(TInput input);
```

如果一种类型的数组应转换为另一种类型的数组，就可以使用 ConvertAll()方法。下面是一个与 Person 类无关的 Racer 类。Person 类有 Firstname 和 Lastname 属性，而 Racer 类为赛手的姓名定义了一个属性 Name：

```
public class Racer
{
    public Racer(string name)
    {
        this.name = name;
    }

    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    private string team;
    public string Team
    {
        get { return team; }
        set { team = value; }
    }
}
```

使用 Array.ConvertAll()，很容易将 persons 数组转换为 Racer 数组。给每个 Person 元素调用委托。在每个 Person 元素的匿名方法的执行代码中，创建了一个新的 Racer 对象，

将 firstname 和 lastname 连接起来传送给带一个字符串参数的构造函数。结果是一个 Racer 对象数组：

```
Racer[] racers =
      Array.ConvertAll<Person, Racer>(
           persons,
           delegate(Person person)
           {
               return new Racer(person.Firstname + " " + person.Lastname);
           });
```

9.7 Framework 的其他泛型类型

除了 System.Collections.Generic 命名空间之外，.NET Framework 还有其他泛型类型。这里讨论的结构和委托都位于 System 命名空间中，用于不同的目的。

本节讨论如下内容：

- 结构 Nullable<T>
- 委托 EventHandler<TEventArgs>
- 结构 ArraySegment<T>

9.7.1 结构 Nullable<T>

数据库中的数字和编程语言中的数字有显著不同的特征，因为数据库中的数字可以为空，C#中的数字不能为空。Int32 是一个结构，而结构实现为值类型，所以它不能为空。

只有在数据库中，而且把 XML 数据映射为.NET 类型，才不存在这个问题。

这种区别常常令人很头痛，映射数据也要多做许多工作。一种解决方案是把数据库和 XML 文件中的数字映射为引用类型，因为引用类型可以为空值。但这也会在运行期间带来额外的系统开销。

使用 Nullable<T>结构很容易解决这个问题。在下面的例子中，Nullable<T>用 Nullable<int>实例化。变量 x 现在可以像 int 那样使用了，进行赋值或使用运算符执行一些计算。这是因为我们转换了 Nullable<T>类型的运算符。x 还可以是空。可以检查 Nullable<T>的 HasValue 和 Value 属性，如果该属性有一个值，就可以访问该值：

```
Nullable<int> x;
x = 4;
x += 3;
if (x.HasValue)
{
   int y = x.Value;
}
x = null;
```

因为可空类型使用得非常频繁，所以 C#有一种特殊的语法，用于定义这种类型的变量。定义这类变量时，不使用一般结构的语法，而使用？运算符。在下面的例子中，x1 和 x2

都是可空 int 类型的实例：

```
Nullable<int> x1;
int? x2;
```

可空类型可以与 null 和数字比较，如上所示。这里，x 的值与 null 比较，如果 x 不是 null，就与小于 0 的值比较：

```
int? x = GetNullableType(); Please use brackets here. Good coding standards. [CN]
can do it here
if (x == null)
{
   Console.WriteLine("x is null");
}
else if (x < 0)
{
   Console.WriteLine("x is smaller than 0");
}
```

可空类型还可以使用算术运算符。变量 x3 是变量 x1 和 x2 的和。如果这两个可空变量中有一个的值是 null，它们的和就是 null。

```
int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;
```

非可空类型可以转换为可空类型。从非可空类型转换为可空类型时，在不需要强制类型转换的地方可以进行隐式转换。这种转换总是成功的：

```
int y1 = 4;
int? x1 = y1;
```

但从可空类型转换为非可空类型可能会失败。如果可空类型的值是 null，把 null 值赋予非可空类型，就会抛出 InvalidOperationException 类型的异常。这就是进行显式转换时需要类型转换运算符的原因：

```
int? x1 = GetNullableType();
int y1 = (int)x1;
```

如果不进行显式类型转换，还可以使用接合运算符(coalescing operator)从可空类型转换为非可空类型。接合运算符的语法是??，为转换定义了一个默认值，以防可空类型的值是 null。这里，如果 x1 是 null，y1 的值就是 0。

```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

9.7.2 EventHandler<TEventArgs>

在 Windows Forms 和 Web 应用程序中，为许多不同的事件处理程序定义了委托。其中

一些事件处理程序如下：

```
public sealed delegate void EventHandler(object sender, EventArgs e);
public sealed delegate void PaintEventHandler(object sender, PaintEventArgs e);
public sealed delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

这些委托的共同点是，第一个参数总是 sender，它是事件的起源，第二个参数是包含事件特定信息的类型。

使用新的 EventHandler<TEventArgs>，就不需要为每个事件处理程序定义新委托了。可以看出，第一个参数的定义方式与以前一样，但第二个参数是一个泛型类型 TeventArgs。where 子句指定 TEventArgs 的类型必须派生于基类 EventArgs。

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

9.7.3 ArraySegment<T>

结构 ArraySegment<T>表示数组的一段。如果需要数组的一部分，就可以使用数组段。在 ArraySegment<T>中，包含了数组段的信息(偏移量和元素个数)。

在下面的例子中，变量 arr 定义为有 8 个元素的 int 数组。ArraySegment<int>类型的变量 segment 用于表示该整数数组的一段。该段用构造函数初始化，在这个构造函数中，传送了该数组、偏移量和元素个数。其中偏移量设置为 2，所以从第三个元素开始，元素个数设置为 3，所以 6 是数组段的最后一个元素。

数组段可以用 Array 属性访问。ArraySegment<T>还有 Offset 和 Count 属性，表示定义数组段的初始化了的值。for 循环用于迭代数组段。for 循环的第一个表达式初始化为迭代开始的偏移量。第二个表达式指定数组段中的元素个数，以确定迭代是否停止。在 for 循环中，数组段包含的元素用 Array 属性来访问：

```
int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};
ArraySegment<int> segment = new ArraySegment<int>(arr, 2, 3);

for (int i = segment.Offset; i < segment.Offset + segment.Count+1; i++)
{
  Console.WriteLine(segment.Array[i]);
}
```

在上面的例子中，ArraySegment<T>结构有什么用处？ArraySegment<T>可以作为参数传送给方法。这样，只要一个参数就可以定义数组、偏移量和元素个数，而不是 3 个参数。

WorkWithSegment()方法把 ArraySegment<string>作为参数。在这个方法的实现代码中，Offset、Count 和 Array 属性的用法与以前相同：

```
void WorkWithSegment(ArraySegment<string> segment)
{
  for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
  {
    Console.WriteLine(segment.Array[i]);
```

```
        }
    }
```

注意：

数组段不复制原数组的元素，但原数组可以通过 ArraySegment<T>访问。如果数组段中的元素改变了，这些变化也会反映到原数组中。

9.8 小结

本章介绍了.NET 2.0 中一个非常重要的特性：泛型。通过泛型类可以创建独立于类型的类，泛型方法是独立于类型的方法。接口、结构和委托也可以用泛型的方式创建。泛型引入了一种新的编程方式。我们介绍了算法(尤其是操作和谓词)如何用于不同的类，而且它们都是类型安全的。泛型委托可以去除集合中的算法。

.NET Framework 的其他类型包括 Nullable<T>、EventHandler<TEventArgs>和 ArraySegment<T>。

第10章 集　合

第 5 章介绍了数组和 Array 类执行的接口。数组的大小是固定的。如果元素个数是动态的，就应使用集合类。

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

本章介绍如何使用对象组。主要内容如下：

- 集合接口和类型
- 列表
- 队列
- 栈
- 链表
- 有序表
- 字典
- 带多个键的字典
- 位数组
- 性能

10.1 集合接口和类型

集合类可以组合为集合，存储 Object 类型的元素和泛型集合类。在.NET 2.0 之前，不存在泛型。现在泛型集合类通常是集合的首选类型。泛型集合类是类型安全的，如果使用值类型，是不需要装箱操作的。如果要在集合中添加不同类型的对象，且这些对象不是相互派生的，例如在集合中添加 int 和 string 对象，就只需基于对象的集合类。另一组集合类是专用于特定类型的集合，例如 StringCollection 类专用于 string 类型。

提示：
泛型的内容可参见第 9 章。

对象类型的集合位于 System.Collections 命名空间；泛型集合类位于 System.Collections.

Generic 命名空间；专用于特定类型的集合类位于 System.Collections.Specialized 命名空间。

当然，组合集合类还有其他方式。集合可以根据集合类执行的接口组合为列表、集合和字典。接口及其功能如表 10-1 所示。.NET 2.0 为集合类添加了新的泛型接口，例如 IEnumerable<T>和 IList<T>。这些接口的非泛型版本将一个对象定义为方法的参数，而其泛型版本使用泛型类型 T。

提示：

接口 IEnumerable、ICollection 和 IList 的内容详见第 5 章。

对集合非常重要的接口及其方法和属性如表 10-1 所示。

表 10-1

接 口	方法和属性	说 明
IEnumerable，IEnumerable<T>	GetEnumerator()	如果将 foreach 语句用于集合，就需要接口 IEnumerable。这个接口定义了方法 GetEnumerator()，它返回一个实现了 IEnumerator 的枚举。泛型接口 IEnumerable<T>继承了非泛型接口 IEnumerable，定义了一个返回 Enumerable<T>的 GetEnumerator 方法。因为这两个接口具有继承关系，所以对于每个需要 IEnumerable 类型参数的方法，都可以传送 Enumerable<T>对象
ICollection	Count, IsSynchronized, SyncRoot, CopyTo()	接口 ICollection 由集合类实现。使用这个接口的方法可以在集合中添加和删除元素
ICollection<T>	Count，IsReadOnly，Add()，Clear()，Contains()，CopyTo() Remove()	接口 ICollection<T>扩展了接口 IEnumerable 的功能
IList	IsFixedSize，IsReadOnly，Item，Add，Clear，Contains，IndexOf，Insert，Remove，RemoveAt	接口 IList 派生于接口 ICollection。IList 允许使用索引器访问集合，还可以在集合的任意位置插入或删除元素
IList<T>	Item，IndexOf Insert，Remove	与接口 ICollection<T>类似，接口 IList<T>也继承了接口 ICollection。 第 5 章提到，Array 类实现了这个接口，但添加或删除元素的方法会抛出 NotSupportedException 异常。在大小固定的只读集合（如 Array 类）中，这个接口定义的一些方法会抛出 NotSupportedExceptiuon 异常

(续表)

接口	方法和属性	说明
IDictionary	IsFixedSize，IsReadOnly，Item，Keys，Values，Add()，Clear()，Contains()，GetEnumerator()，Remove()	接口 IDictionary 或 IDictionary<TKey,TValue>由其元素包含键和值的集合实现
IDictionary<TKey，TValue>	Item，Keys，Values，Add()，ContainsKey ()，Remove()，TyrGetValue()	
IComparer<T>	Compare()	接口 IComparer<T>由比较器实现，通过 Compare()方法给集合中的元素排序
IEqualityComparer<T>	Equals()，GetHashCode()	接口 IEqualityComparer<T>由一个比较器实现，该比较器可用于字典中的键。使用这个接口，可以对对象进行相等比较。方法 GetHashCode()应为每个对象返回一个唯一值。如果对象相等，Equals()方法就返回 true，否则返回 false

10.2 列表

.NET Framework 为动态列表提供了类 ArrayList 和 List<T>。System.Collections.Generic 命名空间中的类 List<T>的用法非常类似于 System.Collections 命名空间中的 ArrayList 类。这个类实现了 IList、ICollection 和 IEnumerable 接口。第 9 章讨论了这些接口的方法，所以本节只探讨如何使用 List<T>类。

下面的例子将 Racer 类中的成员用作要添加到集合中的成员，以表示一级方程式的一位赛手。这个类有三个字段：firstname、lastname 和获胜者的数量。这些字段可以用属性来访问。在该类的构造函数中，可以传送赛手的姓名和获胜者的数量，以设置成员。方法 ToString()重写为返回赛手的姓名。类 Racer 也实现了泛型接口 IComparer<T>，为 Racer 元素排序。

```
[Serializable]
public class Racer : IComparable<Racer>, IFormattable
{
  public Racer()
```

```
      : this(String.Empty, String.Empty, String.Empty) {}

   public Racer(string firstname, string lastname, string country)
      : this(firstname, lastname, country, 0) {}

   public Racer(string firstname, string lastname, string country, int wins)
   {
      this.firstname = firstname;
      this.lastname = lastname;
      this.country = country;
      this.wins = wins;
   }

   private string firstname;
   public string Firstname
   {
      get {return firstname; }
      set { firstname = value; }
   }

   private string lastname;
   public string Lastname
   {
      get {return lastname; }
      set { lastname = value; }
   }

   private string country;
   public string Country
   {
      get {return country; }
      set { country = value; }
   }

    private int wins;
    public int Wins
   {
       get {return wins; }
       set { wins = value; }
   }

   public override string ToString()
   {
       return firstname + " " + lastname;
   }

   public string ToString(string format, IFormatProvider formatProvider)
   {
      switch (format)
      {
         case null:
```

```
            case "N":  //Name
               return ToString();
            case "F":  //FirstName
               return Firstname;
            case "L":  //LastName
               return Lastname;
            case "W":  //Wins
               return ToString() + " Wins: " + wins;
            case "C":  //Country
               return ToString() + " Country: " + country;
            case "A":  //All
               return ToString() + ", " + country + " Wins: " + wins;
            default:
               throw new FormatException(String.Format(formatProvider,
                  "Format {0} is not supported", format));
         }
      }

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

         public int CompareTo(Racer other)
      {
         return this.lastname. CompareTo(other.lastname);
      }
   }
```

10.2.1 创建列表

调用默认的构造函数，就可以创建列表对象。在泛型类 List<T>中，必须在声明中为列表的值指定类型。下面的代码说明了如何声明一个 包含 int 的 List<T>和一个包含 Racer 元素的列表。ArrayList 是一个非泛型列表，可以将任意 Object 类型作为其元素接受。

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

```
   ArrayList objectList = new ArrayList();

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

如果列表的容量改变了，整个集合就要重新分配到一个新的内存块中。在 List<T>的实现代码中，使用了一个 T 类型的数组。通过重新分配内存，创建一个新数组，Array.Copy()将旧数组中的元素复制到新数组中。为节省时间，如果事先知道列表中元素的个数，就可以用构造函数定义其容量。下面创建了一个容量为 10 个元素的集合。如果该容量不足以容

纳要添加的元素，就把集合的大小重新设置为 20，或 40，每次都是原来的 2 倍。

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

使用 Capacity 属性可以获取和设置集合的容量。

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

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

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

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

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

提示：

对于新的应用程序，通常可以使用泛型类 List<T>替代非泛型类 ArrayList，而且 ArrayList 类的方法与 List<T>非常相似，所以本节将只介绍 List<T>。

1. 添加元素

使用 Add()方法可以给列表添加元素，如下所示。实例化的泛型类型用 Add()方法定义了第一个参数的类型：

```
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);

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

变量 racers 定义为类型 List<Racer>。使用 new 操作符创建相同类型的一个新对象。因为类 List<T>用具体类 Racer 来实例化，所以现在只有 Racer 对象可以用 Add()方法添加。在下面的例子中，创建了 4 个一级方程式赛手，并添加到集合中。

```
List<Racer> racers = new List<Racer>(20);

Racer graham = new Racer("Graham", "Hill", "UK", 14);
Racers.Add(graham);
Racer emerson = new Racer("Emerson", "Fittipaldi", "Brazil", 14);
Racers.Add(emerson);
Racer mario = new Racer("Mario", "Andretti", "USA", 12);
Racers.Add(Mario);
```

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

使用 List<T>类的 AddRange()方法，可以一次给集合添加多个元素。AddRange()方法的参数是 IEnumerable<T>类型的对象，所以也可以传送一个数组，如下所示：

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

如果在实例化列表时知道集合的元素个数，也可以将执行 IEnumerable<T>的任意对象传送给类的构造函数。这非常类似于 AddRange()方法：

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

2. 插入元素

使用 Insert()方法可以在指定位置插入元素：

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

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

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

3. 访问元素

执行了 IList 和 IList<T>接口的所有类都提供了一个索引器，所以可以使用索引器，通过传送元素号来访问元素。第一个元素可以用索引值 0 来访问。

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

可以用 Count 属性确定元素个数，再使用 for 循环迭代集合中的每个元素，使用索引器访问每一项：

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

提示：

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

List<T>执行了接口 IEnumerable，所以也可以使用 foreach 语句迭代集合中的元素。

注意：

编译器解析 foreach 语句时，利用了接口 IEnumerable 和 IEnumerator，参见第 5 章。

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

除了使用 foreach 语句之外，List<T>类还提供了 ForEach()方法，它用 Action<T>参数声明。ForEach 迭代集合中的每一项，调用传送作为每一项的参数的方法。

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

为了给 ForEach 传送一个方法，Action<T>声明为一个委托，它定义了一个返回类型为 void、参数为 T 的方法。

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

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

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

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

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

也可以编写一个匿名方法，它将 Racer 对象作为参数。这里，格式 A 由 IFormattable 接口的 ToString()方法用于显示赛手的所有信息：

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

注意：

匿名方法详见第 7 章。

4. 删除元素

删除元素时，可以利用索引，或传送要删除的元素。下面的代码把 3 传送给 RemoveAt()，删除第 4 个元素：

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

也可以直接将 Racer 对象传送给 Remove()方法，删除这个元素。按索引删除比较快，因为必须在集合中搜索要删除的元素。Remove()方法先在集合中搜索，用 IndexOf()方法确定元素的索引，再使用该索引删除元素。IndexOf()方法先检查元素类型是否执行了 IEquatable 接口。如果是，就调用这个接口中的 Equals()方法，在集合中查找传送给 Equals()方法的元素。如果没有执行这个接口，就使用 Object 类的 Equals()方法比较元素。Object

类的 Equals()方法的默认实现代码对值类型进行按位比较，对引用类型只比较其引用。

注意：
第 6 章介绍了如何重写 Equals()方法。

这里从集合中删除了变量 graham 引用的赛手。变量 graham 是前面在填充集合时创建的。接口 IEquatable 和 Object.Equals()方法都没有在 Racer 类中重写，所以不能用要删除的元素内容创建一个新对象，再把它传送给 Remove()方法。

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

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

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

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

5. 搜索

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

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

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

使用方法 IndexOf()，还可以指定不需要搜索整个集合，但必须指定从哪个索引开始搜索以及要搜索的元素个数。

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

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

Predicate<T>类型是一个委托，它返回一个布尔值，需要把类型 T 作为参数。这个委托的用法与 ForEach()方法中的 Action 委托类似。如果 Predicate 返回 true，就表示有一个匹配，找到了一个元素。如果它返回 false，就表示没有找到元素，搜索将继续。

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

在 List<T>类中，把 Racer 对象作为类型 T，所以可以将一个方法（该方法将类型 Racer 定义为参数、且返回 bool）的地址传送给 FindIndex()方法。查找指定国家的第一个赛手时，可以创建如下所示的 FindCountry 类。Find()方法的签名和返回类型是通过 Predicate<T>委托定义的。Find()方法使用变量 country 搜索用 FindCountry 类的构造函数定义的一个国家。

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

    public bool FindCountryPredicate(Racer r)
    {
       if(r == 0) throw new ArgumentNullException("r");
       return r.Country == country;
    }
}
```

使用 FindIndex()方法可以创建 FindCountry()类的一个新实例，把一个国家字符串传送给构造函数，再传送 Find 方法的地址。FindIndex()方法成功完成后，index2 就包含集合中 Country 属性设置为 Finland 的第一项的索引。

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

除了用处理程序方法创建一个类之外，还可以在这里创建一个匿名方法。结果与前面完全相同。现在委托定义了匿名方法的实现代码，来搜索 Country 属性设置为 Finland 的元素。

```
int index3 = racers. FindIndex(
   delegate(Racer r)
{
  return r.Country == "Finland";
}
```

与 IndexOf()方法类似，使用 FindIndex()方法也可以指定搜索开始的索引和要迭代的元素个数。为了从集合中最后一个元素开始向前搜索，可以使用 FindLastIndex()方法。

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

```
Racer r = racers. Find(
   delegate(Racer r)
{
  return r.Firstname == "Niki";
});
```

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

这里调用了 FindAll()方法，返回 Wins 属性设置超过 20 的所有 racer 项。所有赢得 20 多场比赛的赛手都从 bigWinners 列表中引用。

```
List<Racer> bigWinners = racers.FindAll(
   delegate(Racer r)
{
  return r.Wins > 20;
}
};
```

用 foreach 语句迭代 bigWinners 变量，结果如下：

```
foreach(Racer r in bigWinners)
{
  Console.WriteLine("{0:A}", r);
}
```

```
Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25
Alain Prost, France Wins: 51
```

这个结果没有排序，但这是下一步要做的工作。

6. 排序

List<T>类可以对元素排序。Sort()方法定义了几个重载方法。可以传送给它的参数有泛型委托 Comparison<T>、泛型接口 IComparer<T>、泛型接口 IComparer<T>和一个范围值。

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

只有集合中的元素执行了接口 IComparable，才能使用不带参数的 Sort()方法。

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

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

如果需要按照元素类型不默认支持的方式排序，就应使用其他技术，例如传送执行了 IComparer<T>接口的对象。

类 RacerComparer 给 Racer 类型执行了接口 IComparer<T>。这个类允许按名字、姓氏、国籍或获胜人数排序。排序的种类用内部枚举类型 CompareType 定义。CompareType 用类 RacerComparer 的构造函数设置。接口 IComparer<Racer>定义了排序所需的方法 Compare()。在这个方法的实现代码中，使用了 string 和 int 类型的 CompareTo()方法。

```
public class RacerComparer : IComparer<Racer>
{
    public enum CompareType
    {
       Firstname,
       Lastname;
       Country;
       Wins
    }

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

    public int Compare(Racer x, Racer y)
    {
        if (x == null) throw new ArgumentNullException("x");
        if (y == null) throw new ArgumentNullException("y");
        int result;
        switch (compareType)
       {
           case compareType.Firstname:
              return x.Firstname.CompareTo(y.Firstname);
           case compareType.Lastname:
              return x.Lastname.CompareTo(y.Lastname);
           case compareType.Country:
              if ((result = x.Country.CompareTo(y. Country) == 0)
                  return x.Lastname.CompareTo(y.Lastname);
              else
                  return res;
           case compareType.Wins:
              return x.Wins.CompareTo(y.Wins);
           default:
               throw new ArgumentNullException("Invalid Compare Type");
        }
    }
}
```

现在，可以对类 RacerComparer 的一个实例使用 Sort()方法。传送枚举 RacerComparer.CompareType.Country，按属性 Country 对集合排序：

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

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

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

Comparison<T>是一个方法的委托，该方法有两个 T 类型的参数，返回类型为 int。如

果参数值相等，该方法就返回 0。如果第一个参数比第二个小，就返回小于 0 的值；否则，返回大于 0 的值。

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

现在可以把一个简单的匿名方法传送给 Sort()方法，按获胜者的人数排序。两个参数的类型是 Racer，在其实现代码中，使用 int 方法 CompareTo()比较 Wins 属性。在实现代码中，r2 和 r1 以逆序方式使用，所以获胜者的人数以降序方式排序。方法调用完后，完整的赛手列表就按赛手的姓名排序。

```
racers.Sort(delegate(Racer r1, Racer r2) {
   return r2.Wins.CompareTo(r1.Wins); });
```

也可以调用 Reverse()方法，倒转整个集合的顺序。

7. 类型转换

使用 List<T>类的 ConvertAll()方法，可以把任意类型的集合转换为另一种类型。ConvertAll()方法使用一个 Converter 委托，其定义如下：

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

泛型类型 TInput 和 TOutput 用于转换。TInput 是委托方法的变元，TOutput 是返回类型。

在这个例子中，所有的 Racer 类型都应转换为 Person 类型。Racer 类型包含姓名和汽车，而 Person 类型只包含姓名。为了进行转换，可以忽略赛手的国籍，但姓名必须转换：

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

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

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

转换时调用了 racers.ConvertAll<Person>()方法。这个方法的变元定义为一个匿名方法，其变元的类型是 Racer，返回类型是 Person。在匿名方法的实现代码中，创建并返回了一个新的 Person 对象。对于 Person 对象，把 Firstname 和 Lastname 传送给构造函数：

```
List<Person> persons = racers. ConvertAll<Person>(
  delegate(Racer r)
```

```
    {
        return new Person(r.Firstname + " " +r.Lastname);
    });
```

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

10.2.2 只读集合

集合创建好后，就是可读写的。当然，集合必须是可读写的，否则就不能给它填充值了。但是，在填充完集合后，可以创建只读集合。List<T>集合的方法 AsReadOnly 返回 ReadOnlyCollection<T>类型的对象。ReadOnlyCollection<T>类执行的接口与 List<T>相同，但所有修改集合的方法和属性都抛出 NotSupportedException 异常。

10.3 队列

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

注意：
本章的后面将使用链表的另一种实现方式，来定义优先级列表。

在.NET 的 System.Collections 命名空间中有非泛型类 Queue，在 System.Collections.Generic 命名空间中有泛型类 Queue<T>。这两个类的功能非常类似，但泛型类是强类型化的，定义了类型 T，而非泛型类基于 Object 类型。

在内部，Queue<T>类使用 T 类型的数组，这类似于 List<T>类型。另一个类似之处是它们都执行 ICollection 和 IEnumerable 接口。Queue 类执行了 ICollection、IEnumerable 和 ICloneable 接口。Queue<T>类没有执行泛型接口 ICollection<T>，因为这个接口用 Add()和 Remove()方法定义了在集合中添加和删除元素的方法。

队列与列表的主要区别是队列没有执行 IList 接口。所以不能用索引器访问队列。队列只允许添加元素，该元素会放在队列的尾部(使用 Enqueue()方法)，从队列的头部获取元素(使用 Dequeue()方法)。

图 10-1 显示了队列的元素。Enqueue()方法在队列的一端添加元素，Dequeue()方法在队列的另一端读取和删除元素。用 Dequeue()方法读取元素，将同时从队列中删除该元素。再调用一次 Dequeue()方法，会删除队列中的下一项。

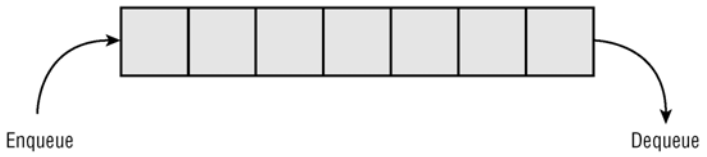

图 10-1

Queue 和 Queue <T>类的方法如表 10-2 所示。

表 10-2

Queue 和 Queue <T>类的成员	说 明
Enqueue()	在队列一端添加一个元素
Dequeue()	在队列的头部读取和删除一个元素。如果在调用 Dequeue()方法时，队列中不再有元素，就抛出 InvalidOperationException 异常
Peek()	在队列的头部读取一个元素，但不删除它
Count	返回队列中的元素个数
TrimExcess()	重新设置队列的容量。Dequeue()方法从队列中删除元素，但不会重新设置队列的容量。要从队列的头部去除空元素，应使用 TrimExcess()方法
Contains()	确定某个元素是否在队列中，如果是，就返回 true
CopyTo() ToArray()	把元素从队列复制到一个已有的数组中。ToArray()方法返回一个包含队列元素的新数组

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

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

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

```
public class Document
{
   private string title;
   public string Title
   {
      get
      {
         return title;
       }
   }
```

```
        private string content;
        public string Content
        {
           get
           {
              return content;
           }
        }

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

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

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

提示：

线程和 lock 语句参见第 18 章。

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

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

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

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

```
        }

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

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

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

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

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

   private DocumentManager documentManager;

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

在应用程序的 Main()方法中，实例化一个 DocumentManager 对象，启动文档处理线程。

接着创建 1000 个文档，并添加到 DocumentManager 中：

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

      ProcessDocuments.Start(dm);

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

在启动应用程序时，会在队列中添加和删除文档，输出如下所示：

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

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

10.4 栈

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

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

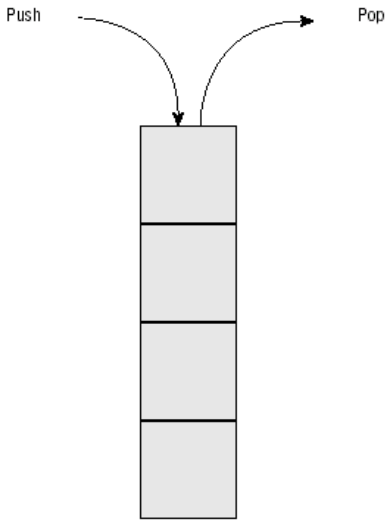

图 10-2

与 Queue 类相同，非泛型类 Statck 也执行了 ICollection、IEnumerable 和 ICloneable 接口；泛型类 Statck<T>实现了 IEnumerable<T>、ICollection 和 IEnumerable 接口。

Statck 和 Statck<T>类的成员如表 10-3 所示。

表 10-3

Statck 和 Statck<T>类的成员	说 明
Push()	在栈顶添加一个元素
Pop()	从栈顶删除一个元素，并返回该元素。如果栈是空的，就抛出 InvalidOperationException 异常
Peek()	返回栈顶元素，但不删除它
Count	返回栈中的元素个数
Contains()	确定某个元素是否在栈中，如果是，就返回 true
CopyTo() ToArray()	把元素从栈复制到一个已有的数组中。ToArray()方法返回一个包含栈中元素的新数组

在下面的例子中，使用 Push()方法把三个元素添加到栈中。在 foreach 方法中，使用 IEnumerable 接口迭代所有的元素。栈的枚举器不会删除元素，只会逐个返回元素。

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

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

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

```
CBA
```

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

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

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

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

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

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

10.5 链表

LinkedList<T>集合类没有非泛型集合的类似版本。LinkedList<T>是一个双向链表，其元素指向它前面和后面的元素，如图 10-3 所示。

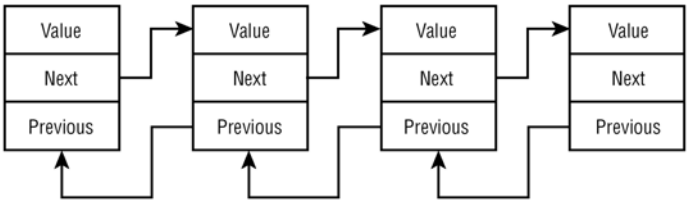

图 10-3

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

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

链表不仅能在列表中存储元素，还可以给每个元素存储下一个元素和上一个元素的信息。这就是 LinkedList<T>包含 LinkedListNode<T>类型的元素的原因。使用 LinkedListNode<T>类，可以获得列表中的下一个元素和上一个元素。表 10-4 描述了 LinkedListNode<T>的属性。

表 10-4

LinkedListNode<T>的属性	说 明
List	返回与节点相关的 LinkedList<T>
Next	返回当前节点之后的节点。其返回类型是 LinkedListNode<T>
Previous	返回当前节点之前的节点
Value	返回与节点相关的元素，其类型是 T

类 LinkedList<T>执行了 IEnumerable<T>、ICollection<T>、ICollection、IEnumerable、ISerializable 和 IDeserializationCallback 接口。这个类的成员如表 10-5 所示。

表 10-5

LinkedList<T>的成员	说 明
Count	返回链表中的元素个数
First	返回链表中的第一个节点。其返回类型是 LinkedListNode<T>。使用这个返回的节点，可以迭代集合中的其他节点
Last	返回链表中的最后一个元素。其返回类型是 LinkedListNode<T>
AddAfter() AddBefore() AddFirst() AddLast()	使用 AddXXX 方法可以在链表中添加元素。使用相应的 Add 方法，可以在链表的指定位置添加元素。AddAfter()需要一个 LinkedListNode<T>对象，在该对象中可以指定要添加的新元素后面的节点。AddBefore()把新元素放在第一个参数定义的节点前面。AddFirst()和 AddLast()把新元素添加到链表的开头和结尾 重载所有这些方法来接收任一个对象以添加类型 LinkedListNode<T>或类型 T。如果传送 T 对象，则创建一个新 LinkedListNode<T>对象
Remove() RemoveFirst() RemoveLast()	Remove()、RemoveFirst()和 RemoveLast()方法从链表中删除节点。RemoveFirst()删除第一个元素，RemoveLast()删除最后一个元素。Remove()需要搜索一个对象，从链表中删除匹配该对象的第一个节点
Clear()	从链表中删除所有的节点
Contains()	在链表中搜索一个元素，如果找到该元素，就返回 true，否则返回 false
Find()	从链表的开头开始搜索传送给它的元素，并返回一个 LinkedListNode<T>
FindLast()	与 Find()方法类似，但从链表的结尾开始搜索

示例应用程序使用了一个链表 LinkedList<T>和一个列表 List<T>。链表包含文档，这

与上一个例子相同，但文档有一个额外的优先级。在链表中，文档按照优先级来排序。如果多个文档的优先级相同，则这些元素就按照文档的插入时间来排序。

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

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

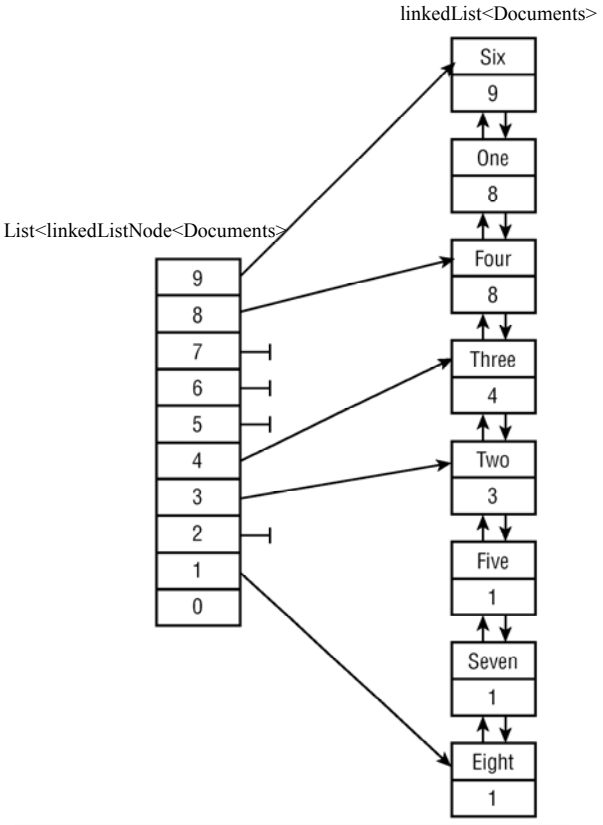

图 10-4

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

```
public class Document
{
```

```
        private string title;
        public string Title
        {
           get
           {
              return title;
           }
        }

        private string content;
        public string Content
        {
           get
           {
              return content;
           }
        }

        private byte priority;
        public byte Priority
        {
           get
           {
              return priority;
           }
        }

        public Document(string title, string content, byte priority)
        {
           this.title = title;
           this.content = content;
           this.priority = priority;
        }
     }
```

解决方案的核心是 PriorityDocumentManager 类。这个类很容易使用。在这个类的公共接口中，可以把新的 Document 元素添加到链表中，检索第一个文档，为了便于测试，它还提供了一个方法，在元素链接到链表中时，该方法可以显示集合中的所有元素。

PriorityDocumentManager 类包含两个集合。LinkedList<Document>类型的集合包含所有的文档。List<LinkedListNode<Document>>类型的集合包含最多 10 个元素的引用，它们是添加指定优先级的新文档的入口点。这两个集合变量都用 PriorityDocumentManager 类的构造函数来初始化。列表集合也用 null 初始化：

```
     public class PriorityDocumentManager
     {
        private readonly LinkedList<Document> documentList;

        // priorities 0..9
        private readonly List<LinkedListNode<Document>> priorityNodes;
```

```
public PriorityDocumentManager()
{
   documentList = new LinkedList<Document>();

   priorityNodes = new List<LinkedListNode<Document>>(10);
   for (int i = 0; i < 10; i++)
   {
      priorityNodes.Add(new LinkedListNode<Document>(null));
   }
}
```

在类的公共接口中，有一个方法 AddDocument()。它只是调用私有方法 AddDocumentToPriorityNode()。把实现代码放在另一个方法中的原因是，AddDocumentToPriorityNode()可以递归调用，如后面所示。

```
public void AddDocument(Document d)
{
    if (d == null) throw new ArgumentNullException("d");
    AddDocumentToPriorityNode(d, d.Priority);
}
```

在 AddDocumentToPriorityNode()的实现代码中，第一个操作是检查优先级是否在允许的范围内。这里允许的范围是 0~9。如果传送了错误的值，就会抛出 ArgumentException 类型的异常。

接着检查是否已经有一个优先级节点与所传送的优先级相同。如果在列表集合中没有这样的优先级节点，就递归调用 AddDocumentToPriorityNode()，递减优先级值，检查是否有低一级的优先级节点。

如果优先级节点的优先级值与所传送的优先级值不同，也没有比该优先级值更低的优先级节点，就可以调用 AddLast()方法，将文档安全地添加到链表的尾部。另外，链表节点由负责指定文档优先级的优先级节点引用。

如果存在这样的优先级节点了，就可以在链表中找到插入文档的位置。这里必须区分是存在指定优先级值的优先级节点，还是引用文档的优先级节点有较低的优先级值。对于第一种情况，可以把新文档插入由优先级节点引用的位置后面。因为优先级节点总是引用指定优先级值的最后一个文档，所以必须设置优先级节点的引用。如果引用文档的优先级节点有较低的优先级值，情况会比较复杂。这里新文档必须插入优先级值与优先级节点相同的所有文档的前面。为了找到优先级值相同的第一个文档，要通过一个 while 循环，使用 Previous 属性迭代所有的链表节点， 直到找到一个优先级值不同的链表节点为止。这样，就找到了文档的插入位置，并可以设置优先级节点。

```
private void AddDocumentToPriorityNode(Document doc, int priority)
{
   if (priority > 9 || priority < 0)
      throw new ArgumentException("Priority must be between 0 and 9");

   if (priorityNodes[priority].Value == null)
```

```
            {
               priority--;
               if (priority >= 0)
               {
                  // check for the next lower priority
                  AddDocumentToPriorityNode(doc, priority);
               }
               else // now no priority node exists with the same priority or lower
                    // add the new document to the end
               {
                  documentList.AddLast(doc);
                  priorityNodes[doc.Priority] = documentList.Last;
               }
               return;
            }
            else // a priority node exists
            {
               LinkedListNode<Document> priorityNode = priorityNodes[priority];
               if (priority == doc.Priority)    // priority node with the same
                                                // priority exists
               {
                  documentList.AddAfter(priorityNode, doc);

                  // set the priority node to the last document with the same priority
                  priorityNodes[doc.Priority] = priorityNode.Next;
               }
               else // only priority node with a lower priority exists
               {
                  // get the first node of the lower priority
                  LinkedListNode<Document> firstPriorityNode = priorityNode;
                  while (firstPriorityNode.Previous != null &&
                        firstPriorityNode.Previous.Value.Priority ==
                             priorityNode.Value.Priority)
                  {
                     firstPriorityNode = priorityNode.Previous;
                  }

                  documentList.AddBefore(firstPriorityNode, doc);

                  // set the priority node to the new value
                  priorityNodes[doc.Priority] = firstPriorityNode.Previous;
               }
            }
         }
```

现在还剩下一个简单的方法没有讨论。DisplayAllNodes()只是在一个 foreach 循环中，把每个文档的优先级和标题显示在控制台上。

GetDocument 方法从链表中返回第一个文档（优先级最高的文档），并从链表中删除它：

```
         public void DisplayAllNodes()
```

```
        {
           foreach (Document doc in documentList)
           {
              Console.WriteLine("priority: {0}, title {1}", doc.Priority, doc.Title);
           }
        }

        // returns the document with the highest priority (that's first
        // in the linked list)
        public Document GetDocument()
        {
           Document doc = documentList.First.Value;
           documentList.RemoveFirst();
           return doc;
        }
     }
```

PriorityDocumentManager 的 Main()方法用于演示其功能。在链表中添加 8 个优先级不同的新文档，再显示整个链表：

```
        static void Main
        {
           PriorityDocumentManager pdm = new PriorityDocumentManager();
           pdm.AddDocument(new Document("one", "Sample", 8));
           pdm.AddDocument(new Document("two", "Sample", 3));
           pdm.AddDocument(new Document("three", "Sample", 4));
           pdm.AddDocument(new Document("four", "Sample", 8));
           pdm.AddDocument(new Document("five", "Sample", 1));
           pdm.AddDocument(new Document("six", "Sample", 9));
           pdm.AddDocument(new Document("seven", "Sample", 1));
           pdm.AddDocument(new Document("eight", "Sample", 1));

           pdm.DisplayAllNodes();
         }
```

在处理好的结果中，文档先按优先级排序，再按添加文档的时间排序：

```
priority: 9, title six
priority: 8, title one
priority: 8, title four
priority: 4, title three
priority: 3, title two
priority: 1, title five
priority: 1, title seven
priority: 1, title eight
```

10.6 有序表

如果需要排好序的表，可以使用 SortedList<TKey, TValue>。这个类按照键给元素排序。

下面的例子创建了一个有序表，其中键和值都是 string 类型。默认的构造函数创建了一个空表，再用 Add()方法添加两本书。使用重载的构造函数，可以定义有序表的容量，传送执行了 IComparer<TKey>接口的对象，用于给有序表中的元素排序。

Add()方法的第一个参数是键（书名），第二个参数是值（ISBN 号）。除了使用 Add()方法之外，还可以使用索引器将元素添加到有序表中。索引器需要把键作为索引参数。如果键已存在，那么 Add()方法就抛出一个 ArgumentExeption 类型的异常。如果索引器使用相同的键，就用新值替代旧值。

```
SortedList<string, string> books = new SortedList<string, string>();
books.Add(".NET 2.0 Wrox Box", "978-0-470-04840-5");
books.Add("Professional C# 2005", "978-0-7645-7534-1");

books["Beginning Visual C# 2005"] = "978-0-7645-4382-1";
books["Professional C# with .NET 3.0"] = "978-0-470-12472-7";
```

可以使用 foreach 语句迭代有序表。枚举器返回的元素是 KeyValuePair<TKey, TValue>类型，其中包含了键和值。键可以用 Key 属性访问，值用 Value 属性访问。

```
foreach (KeyValuePair<string, string> book in books)
{
   Console.WriteLine("{0}, {1}", book.Key, book.Value);
}
```

迭代语句会按键的顺序显示书名和 ISBN 号：

```
.NET 2.0 Wrox Box, 978-0-470-04840-5
Beginning Visual C# 2005, 978-0-7645-4382-1
Professional C# 2005, 978-0-7645-7534-1
Professional C# with .NET 3.0, 978-0-470-12472-7
```

也可以使用 Values 和 Keys 属性访问值和键。Values 属性返回 IList<TValue>，Keys 属性返回 IList<TKey>，所以，可以在 foreach 中使用这些属性：

```
foreach (string isbn in books.Values)
{
   Console.WriteLine(isbn);
}

foreach (string title in books.Keys)
{
   Console.WriteLine(title);
}
```

第一个循环显示值，第二个循环显示键：

```
978-0-470-04840-5
978-0-7645-4382-1
978-0-7645-7534-1
978-0-470-12472-7
.NET 2.0 Wrox Box
```

```
Beginning Visual C# 2005
Professional C# 2005
Professional C# with .NET 3.0
```

SortedList<TKey, TValue>类的属性如表 10-6 所示。

表 10-6

SortedList 的属性	说 明
Capacity	使用 Capacity 属性可以获取和设置有序表能包含的元素个数。该属性与 List<T>类似：默认构造函数会创建一个空表，添加第一个元素会使有序表的容量变成 4 个元素，之后其容量会根据需要成倍增加
Comparer	Comparer 属性返回与有序表相关的比较器。可以在构造函数中传送该比较器。默认的比较器会调用 IComparable<TKey>接口的 CompareTo()方法来比较键。键类型执行了这个接口，也可以创建定制的比较器
Count	Count 属性返回有序表中的元素个数
Item	使用索引器可以访问有序表中的元素。索引器的参数类型由键类型定义
Keys	Keys 属性返回包含所有键的 IList<TKey>
Values	Values 属性返回包含所有值的 IList<TValue>

SortedList<T>类型的方法类似于本章前面介绍的其他集合。见表 10-7。区别是 SortedList<T>需要一个键和一个值。

表 10-7

SortedList 的方法	说 明
Add()	把带有键和值的元素放在有序表中
Remove()，RemoveAt()	Remove()方法需要从有序表中删除的元素的键。使用 RemoveAt()方法可以删除指定索引的元素
Clear()	删除有序表中的所有元素
ContainsKey()，ContainsValue()	ContainsKey()和 ContainsValue()方法检查有序表是否包含指定的键或值，并返回 true 或 false
IndexOfKey()，IndexOfValue()	IndexOfKey()和 IndexOfValue()方法检查有序表是否包含指定的键或值，并返回基于整数的索引
TrimExcess()	重新设置集合的大小，将容量改为需要的元素个数
TryGetValue()	使用 TryGetValue()方法可以尝试获得指定键的值。如果键不存在，这个方法就返回 false。如果键存在，就返回 true，并把值返回为 out 参数

注意：

除了泛型 SortedList<TKey, TValue>之外，还有一个对应的非泛型有序表 SortedList。

10.7 字典

字典表示一种非常复杂的数据结构，这种数据结构允许按照某个键来访问元素。字典也称为映射或散列表。字典的主要特性是能根据键快速查找值。也可以自由添加和删除元素，这有点像 List<T>，但没有在内存中移动后续元素的性能开销。

图 10-5 是字典的一个简化表示。其中 employee-id，如 B4711，是添加到字典中的键。键会转换为一个散列。利用散列创建一个数字，它将索引和值关联起来。然后索引包含一个到值的链接。该图做了简化处理，因为一个索引项可以关联多个值，索引可以存储为一个树形结构。

.NET Framework 提供了几个字典类。可以使用的最主要的类是 Dictionary<TKey, TValue>。这个类的属性和方法与上面的 SortedList<TKey, TValue>几乎完全相同，这里不再赘述。

10.7.1 键的类型

用作字典中键的类型必须重写 Object 类的 GetHashCode()方法。只要字典类需要确定元素的位置，就要调用 GetHashCode()方法。GetHashCode()方法返回的 int 由字典用于计算放置元素的索引。这里不介绍这个算法。我们只需知道，它涉及到素数，所以字典的容量是一个素数。

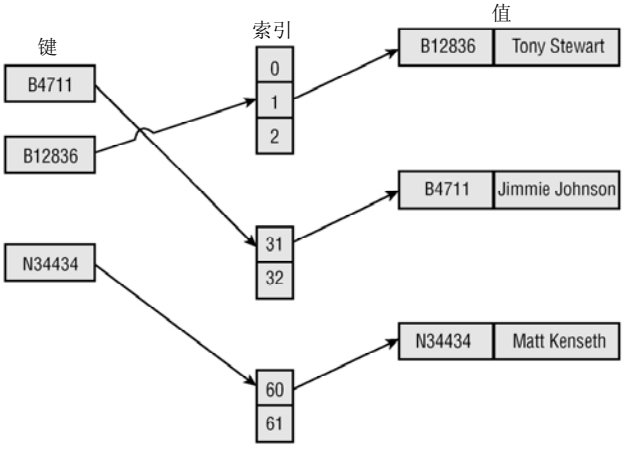

图 10-5

GetHashCode()方法的实现代码必须满足如下要求：

- 相同的对象应总是返回相同的值。
- 不同的对象可以返回相同的值。
- 应执行得比较快，计算的开销不大。
- 不能抛出异常。
- 应至少使用一个实例字段。
- 散列码值应平均分布在 int 可以存储的整个数字区域上。

- 散列码最好在对象的生存期中不发生变化。

提示：
字典的性能取决于 GetHashCode()方法的实现代码。

为什么要使散列码值平均分布在整数的取值区域内？如果两个键返回的散列会得到相同的索引，则字典类就必须寻找最近的可用自由单元来存储第二个数据项，这需要进行一定的搜索，以便以后检索这一项。显然这会降低性能，如果许多键都有相同的索引，这类冲突就比较多。根据 Microsoft 的部分算法的工作方式，计算出来的散列值平均分布在 int.MinValue 和 int.MaxValue 之间时，这种风险会降低到最小。

除了实现 GetHashCode()方法之外，键类型还必须执行 IEquality.Equals()方法，或重写 Object 类的 Equals()方法。因为不同的键对象可能返回相同的散列码，所以字典使用 Equals()方法来比较键。字典检查两个键 A 和 B 是否相等，并调用 A.Equals(B)。这表示必须确保下述条件总是成立：

提示
如果 A.Equals(B)返回 true，则 A.GetHashCode()和 B.GetHashCode()必须总是返回相同的散列码。

这似乎有点奇怪，但它非常重要。如果设计出某种重写这些方法的方式，使上面的条件并不总是成立，把这个类的实例用作键的字典就不能正常工作，而是会发生可笑的事情。例如，把一个对象放在字典中后，就再也找不到它，或者试图查找某项，却返回了错误的项。

注意：
因此，如果为 Equals()方法提供了重写版本，但没有提供 GetHashCode()的重写版本，C#编译器就会显示一个编译警告。

对于 System.Object，这个条件为 true，因为 Equals()方法只是比较引用，GetHashCode()总是返回一个仅基于对象地址的散列。这说明，如果散列表基于一个键，而该键没有重写这些方法，这个散列表就能正常工作。但是，这么做的问题是，只有对象完全相同，键才被认为是相等的。也就是说，把一个对象放在字典中时，必须将它与该键的引用关联起来。也不能在以后用相同的值实例化另一个键对象。如果没有重写 Equals()和 GetHashCode()，类型在字典中使用时就不太方便。

另外，System.String 执行了 IEquatable 接口，并重载了 GetHashCode()方法。Equals()提供了值的比较，GetHashCode()根据字符串的值返回一个散列。因此，字典中把字符串用作键是非常方便的。

数字类型，如 Int32，也执行了 IEquatable 接口，并重载了 GetHashCode()方法。但是这些类型返回的散列码只是映射到值上。如果希望用作键的数字本身没有分布在可能的整数值区域内，把整数用作键就不能满足键值平均分布、获得最佳性能的规则。Int32 并不适合在字典中使用。

如果所使用的键类型没有执行 IEquatable 接口，并根据存储在字典中的键值重载 GetHashCode()，就可以创建一个执行了 IEqualityComparer<T>接口的比较器。IEquality Comparer<T>接口定义了 GetHashCode()和 Equals()方法，并将对象作为参数，因此可以提供与对象类型不同的实现方式。Dictionary<TKey, TValue>构造函数的一个重载版本允许传送实现了 IEqualityComparer <T>接口的对象。如果把这个对象赋予字典，该类就用于生成散列码，比较键。

10.7.2 字典示例

字典示例程序建立了一个员工字典。该字典是用 EmployeeId 对象来索引的，存储在字典中的每个数据项都是一个 Employee 对象，该对象存储员工的详细数据。

EmployeeId 类定义了在字典中使用的键，该类的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。字段在构造函数中填充。ToString()方法重载为获得员工 ID 的字符串表示。与键类型的要求一样，EmployeeId 也要执行 IEquatable 接口，重写 GetHashCode()方法。

```
[Serializable]
public class EmployeeId : IEquatable<EmployeeId>
{
    private readonly char prefix;
    private readonly int number;

    public EmployeeId(string id)
    {
        if (id == null) throw new ArgumentNullException("id");

        prefix = (id.ToUpper())[0];
        int numLength = id.Length - 1;
        number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
    }
    public override string ToString()
    {
        return prefix.ToString() + string.Format("{0,6:000000}", number);
    }

    public override int GetHashCode()
    {
        return (number ^ number << 16) * 0x15051505;
    }
    public bool Equals(EmployeeId other)
    {
        if (other == null) return false;

        return (prefix == other.prefix && number == other.number);
    }
}
```

由 IEquatable<T>接口定义的 Equals()方法比较两个 EmployeeId 对象的值，如果这两个值相同，就返回 true。除了执行 IEquatable<T>接口中的 Equals()方法之外，还可以重写 Object 类中的 Equals()方法。

```
public bool Equals(EmployeeId other)
{
    if (other == null) return false;
    return (prefix == other.prefix && number == other.number);
}
```

由于数字是可变的，因此员工可以有 1～190000 之间的一个值。这并没有填满整数取值区域。GetHashCode()使用的算法将数字向左移动 16 位，再与原来的数字进行异或操作，最后将结果乘以十六进制数 15051505。散列码在整数取值区域上的分布相当均匀：

```
public override int GetHashCode()
{
    return (number ^ number << 16) * 0x15051505;
}
```

注意：

在 Internet 上，有许多更复杂的算法，能使散列码在整数取值区域上更好地分布。也可以使用字符串的 GetHashCode()方法来返回散列。

Employee 类是一个简单的实体类，包含员工的姓名、薪水和 ID。构造函数初始化了所有的值，方法 ToString()返回一个实例的字符串表示。ToString()方法的实现代码使用 StringBuilder 对象创建字符串表示，以提高性能。

```
[Serializable]
public class Employee
{
    private string name;
    private decimal salary;
    private readonly EmployeeId id;

    public Employee(EmployeeId id, string name, decimal salary)
    {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder(100);
        sb.AppendFormat("{0}: {1, -20} {2:C}", id.ToString(), name, salary);
        return sb.ToString();
    }
}
```

在示例程序的 Main()方法中，创建一个新的 Dictionary<TKey, TValue>实例，其中键是

EmployeeId 类型，值是 Employee 类型。构造函数指定了 31 个元素的容量。注意容量一般是素数。但如果指定了一个不是素数的值，也不需要担心。Dictionary<TKey, TValue>类会使用传送给构造函数的整数后面的一个素数，来指定容量。用 Add()方法创建员工对象和 ID，并添加到字典中。除了 Add()方法外，还可以使用索引器，将键和值添加到字典中，如员工 Jeff 和 Casey 所示：

```
static void Main()
{
    Dictionary<EmployeeId, Employee> employees =
        new Dictionary<EmployeeId, Employee>(31);

        EmployeeId idJimmie = new EmployeeId("B4711");
        Employee jimmie = new Employee(idJimmie, "Jimmie Johnson", 8909140.00m);
        employees.Add(idJimmie, jimmie);
        Console.WriteLine(jimmie);

        EmployeeId idTony = new EmployeeId("B12836");
        Employee tony = new Employee(idTony, "Tony Stewart", 7285280.00m);
        employees.Add(idTony, tony);
        Console.WriteLine(tony);

        EmployeeId idMatt = new EmployeeId("N34434");
        Employee matt = new Employee(idMatt, "Matt Kenseth", 6608920.00m);
        employees.Add(idMatt, matt);
        Console.WriteLine(matt);

        EmployeeId idJeff = new EmployeeId("J127");
        Employee jeff = new Employee(idJeff, "Jeff Gordon", 5975870.00m);
        employees[idJeff] = jeff;
        Console.WriteLine(jeff);

        EmployeeId idCasey = new EmployeeId("K100223");
        Employee casey = new Employee(idCasey, "Casey Mears", 5413340.00m);
        employees[idCasey] = casey;
        Console.WriteLine(casey);
```

将数据项添加到字典中后，在 while 循环中读取字典中的员工。让用户输入一个员工号，把该号码存储在变量 userInput 中。用户输入 X 即可退出程序。如果输入的键在字典中，就使用 Dictionary<TKey, TValue>类的 TryGetValue()方法检查它。如果找到了该键，TryGetValue()方法就返回 true，否则返回 false。如果找到了与键关联的值，该值就存储在 employee 变量中，并写入控制台。

注意：

也可以使用 Dictionary<TKey, TValue>类的索引器替代 TryGetValue()方法，来访问存储在字典中的值。但是，如果没有找到键，索引器会抛出 KeyNotFoundException 类型的异常。

```
while (true)
{
    Console.Write("Enter employee id (X to exit)> ");
```

```
        string userInput = Console.ReadLine();
        userInput = userInput.ToUpper();
        if (userInput == "X") break;

        EmployeeId id = new EmployeeId(userInput);

        Employee employee;
        if (!employees.TryGetValue(id, out employee))
        {
            Console.WriteLine("Employee with id {0} does not exist",
                id);
        }
        else
        {
                Console.WriteLine(employee);
            }
        }
    }
```

运行应用程序，得到如下输出：

```
Enter employee ID (format:A999999, X to exit)> J127
J000127: Jeff Gordon $5,975,870.00
Enter employee ID (format:A999999, X to exit)> N34434
N034434: Matt Kenseth $6,608,920.00
Enter employee ID (format:A999999, X to exit)> X
```

10.7.3 其他字典类

Dictionary<TKey, TValue>是 Framework 中的一个主要字典类，还有其他一些类，当然也有一些非泛型的字典类。

基于 Object 类型的字典和.NET 1.0 以来可以使用的字典类如表 10-8 所示。

表 10-8

非泛型字典	说 明
Hashtable	Hashtable 是.NET 1.0 中最常用的字典类。键和值都基于 Object 类型
ListDictionary	ListDictionary 位于命名空间 System.Collections.Specialized，如果使用的元素少于 10 个，它就比 Hashtable 快。ListDictionary 实现为链表
HybridDictionary	如果集合比较小，HybridDictionary 就使用 ListDictionary，当集合增大时，就改为使用 Hashtable。如果事先不知道元素个数，就可以使用 HybridDictionary
NameObjectCollectionBase	NameObjectCollectionBase 是一个抽象基类，将字符串类型的键关联到 Object 类型的值上。它可以用作定制字符串/Object 集合的基类这个类在内部使用 Hashtable

(续表)

非泛型字典	说 明

NameValueCollection	NameValueCollection 派生于 NameObjectCollection。它的键和值都是字符串类型。这个类有一个特性：多个值可以使用同一个键
Dictionary<TKey, TValue>	Dictionary<TKey, TValue>是一般用途的字典，将键映射到值上
SortedDictionary<TKey, TValue>	这是一个二叉搜索树，其中的元素根据键来排序。键的类型必须执行 IComparable<TKey>接口。如果键的类型不能排序，还可以创建一个执行了 IComparer<TKey>接口的比较器，将比较器用作有序字典的构造函数中的一个参数

SortedDictionary<TKey, TValue>和 SortedList<TKey, TValue>的功能类似。但因为 SortedList <TKey, TValue>实现为一个链表，而 SortedDictionary<TKey, TValue>实现为一个字典，所以它们有不同的特性。

- SortedList<TKey, TValue>使用的内存比 SortedDictionary<TKey, TValue>少。
- SortedDictionary<TKey, TValue>的元素插入和删除速度比较快。
- 在用已排好序的数据填充集合时，若不需要修改容量，SortedList<TKey, TValue>就比较快。

提示：

SortedList 使用的内存比 SortedDictionary 少，但 SortedDictionary 在插入和删除未排序的数据时比较快。

10.8 带多个键的字典

在.NET Framework 的字典中，每个键只支持一个值。不能为同一个键添加多个值。但是，可以创建基于 Dictionary<TKey, TValue>的定制字典，将一个列表用作值。

类 MultiDictionary<TKey, TValue>执行了接口 IDictionary<TKey, TValue>。这个类包含了泛型类 Dictionary，它的键 TKey 与外部的类相同，但其值定义为 List<TValue>类型。每个键映射为一个 List<TValue>类型的值。

```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.Collections
{
    public class MultiDictionary<TKey, TValue> : IDictionary<TKey, TValue>
    {
        private Dictionary<TKey, List<TValue>> dict =
        new Dictionary<TKey, List<TValue>>();
    //...
```

MultiDictionary<TKey, TValue>要求实现接口 IDictionary<TKey, TValue>的方法 Add()、ContainsKey()、Remove()、TryGetValue()，属性 Keys、Values，以及索引器。

Add()方法的实现代码首先用 TryGetValue()确定键是否已添加到字典中。如果键已在字典中，就把新值添加到从 TryGetValue()返回的列表中。否则，就创建一个新列表，再把值添加到这个新列表中，与列表相关的键添加到字典中。

```
public void Add(TKey key, TValue value)
{
   List<TValue> list;
   if (dict.TryGetValue(key, out list))
   {
      list.Add(value);
   }
   else
   {
      list = new List<TValue>();
      newList.Add(value);
      dict.Add(key, list);
   }
}
//...
```

方法 ContainsKey()、Remove()和属性 Keys 可以在向包含的字典发出请求时执行：

```
public bool ContainsKey(TKey key)
{
   return dict.ContainsKey(key);
}

public ICollection<TKey> Keys
{
   get
   {
      return dict.Keys;
   }
}

public bool Remove(TKey key)
{
   return dict.Remove(key);
}
//...
```

属性 Values 必须把 List<TValue>类型的内部值转换为 TValue 类型。在完成 while 循环后，列表 named values 就包含所有列表中的所有值，这些列表在包含它们的字典中存储为值。while 循环使用变量 dict 中的枚举器迭代所有的键，将所有的数据项添加到返回的列表中。

```
public ICollection<TValue> Values
{
   get
   {
      List<TValue> values = new List<TValue>();
```

```
            Dictionary<TKey, List<TValue>>.Enumerator enumerator = dict.
            GetEnumerator();
            while (enumerator.MoveNext())
            {
                values.AddRange(enumerator.Current.Value);
            }
            return values;
        }
    }
    //...
```

MultiDictionary 不支持索引器和 TryGetValue()方法，因为一个键可以关联多个值，所以不能返回一个值：

```
    bool IDictionary<TKey, TValue>.TryGetValue(TKey key, out TValue value)
    {
        throw new NotSupportedException("TryGetValue is not supported");
    }

    TValue IDictionary<TKey, TValue>.this[TKey key]
    {
        get
        {
            throw new NotSupportedException(
                "accessing elements by key is not supported");
        }
        set
        {
            throw new NotSupportedException(
                "accessing elements by key is not supported");
        }
    }
    //...
```

在这个类中，不能使用 TValue 类型的索引器，但可以使用 List<TValue>类型的索引器：

```
    public IList<TValue> this[TKey key]
    {
        get
        {
            return dict[key];
        }
    }
    //...
```

接口 IDictionary<TKey, TValue>派生于 ICollection<KeyValuePair<TKey, TValue>>，所以还必须实现方法 Add()、Clear()、Contains()、CopyTo()、Remove()，属性 Count、IsReadOnly。

变元类型为 KeyValuePair<TKey, TValue>的方法 Add()只需调用 Add()方法，其中的 TKey 和 TValue 要分别传送。Clear()方法在包含元素的字典上调用 Clear()。如果指定的键不存在，Contains()就返回 false，否则，list.Contains()就确定值是否在列表中，并返回结果。

CopyTo()把所有的数据项复制到一个数组中。属性 Count 返回所有值的个数。

```
public void Add(KeyValuePair<TKey, TValue> item)
{
    Add(item.Key, item.Value);
}
public void Clear()
{
    dict.Clear();
}

public bool Contains(KeyValuePair<TKey, TValue> item)
{
    List<TValue> list;
    if (!dict.TryGetValue(item.Key, out list))
    {
        return false;
    }
    else
    {
        return list.Contains(item.Value);
    }
}

public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex)
{
    if (array == null)
        throw new ArgumentNullException("array");
    if (arrayIndex < 0 || arrayIndex > array.Length)
        throw new ArgumentOutOfRangeException("array index out of range");
    if (array.Length - arrayIndex < this.Count)
        throw new ArgumentException("Array too small");

   Dictionary<TKey, List<TValue>>.Enumerator enumerator = dict.GetEnumerator();
    while (enumerator.MoveNext())
    {
        KeyValuePair<TKey, List<TValue>> mapPair = enumerator.Current;
        foreach (TValue val in mapPair.Value)
          {
           array[arrayIndex++] = new KeyValuePair<TKey,TValue>(mapPair.Key, val);
          }
    }
}

public int Count
{
    get
    {
        int count = 0;
      Dictionary<TKey, List<TValue>>.Enumerator enumerator = dict.GetEnumerator();
      while (enumerator.MoveNext())
      {
```

```
            KeyValuePair<TKey, List<TValue>> pair = enumerator.Current;
            count += pair.Value.Count;
          }

          return count;
        }
    }

    public bool IsReadOnly
    {
        get { return false; }
    }

    public bool Remove(KeyValuePair<TKey, TValue> item)
    {
        List<TValue> list;
        if (dict.TryGetValue(item.Key, out list))
        {
            return list.Remove(item.Value);
        }
        else
        {
            return false;
        }
    }
    //...
```

接口 IDictionary<TKey, TValue>也派生于 IEnumerable<KeyValuePair<TKey, TValue>>，所以还必须实现 GetEnumerator()方法。这里可以使用 yield return 语句。

注意：

yield return 语句详见第 5 章。

```
    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
    {
        Dictionary<TKey, List<TValue>>.Enumerator enumerateKeys = dict.GetEnumerator();
        while (enumerateKeys.MoveNext())
        {
            foreach (TValue val in enumerateKeys.Current.Value)
            {
                KeyValuePair<TKey, TValue> pair = new KeyValuePair<TKey, TValue>(
                    enumerateKeys.Current.Key, val);
                yield return pair;
            }
        }
    }
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
            return GetEnumerator();
        }
    }
```

```
}
```

MultiDictionary<TKey, TValue>可以像一般的字典那样创建，但可以多次使用同一个键。在这种情况下，可以将国家用作键，将 Racer 用作值。将两个澳大利亚赛手和两个英国赛手添加到字典中。这个字典的索引器返回一个用 foreach 循环迭代的列表。

```
MultiDictionary<string, Racer> racers = new MultiDictionary<string, Racer>();

racers.Add("Canada", new Racer("Jacqes", "Villeneuve", "Canada", 11));
racers.Add("Australia", new Racer("Alan", "Jones", "Australia", 12));
racers.Add("United Kingdom", new Racer("Jackie", "Stewart",
           "United Kingdom", 27));
racers.Add("United Kingdom", new Racer("James", "Hunt",
            "United Kingdom", 10));
racers.Add("Australia", new Racer("Jack", "Brabham", "Australia", 14));

foreach (Racer r in racers["Australia"])
{
   Console.WriteLine(r);
}
```

输出显示了澳大利亚赛手：

```
Alan Jones
Jack Brabham
```

10.9 位数组

如果需要处理许多位，就可以使用类 BitArray 和结构 BitVector32。BitArray 位于命名空间 System.Collections，BitVector32 位于命名空间 System.Collections.Specialized。这两种类型最重要的区别是，BitArray 可以重新设置大小，如果事先不知道需要的位数，就可以使用 BitArray，它可以包含非常多的位。BitVector32 是基于栈的，因此比较快。BitVector32 仅包含 32 位，存储在一个整数中。

10.9.1 BitArray

类 BitArray 是一个引用类型，包含一个 int 数组，每 32 位使用一个新整数。这个类的成员如表 10-9 所示。

表 10-9

BitArray 类的成员	说 明
Count, Length	Count 和 Length 的 get 访问器返回数组中的位数。使用 Length 属性还可以定义新的数组大小，重新设置集合的大小

(续表)

BitArray 类的成员	说 明

Item	可以使用索引器读写数组中的位。索引器是 bool 类型
Get(), Set()	除了使用索引器之外，还可以使用 Get 和 Set 方法访问数组中的位
SetAll()	根据传送给该方法的参数，设置所有位的值
Not()	倒转数组中所有位的值
And(), Or(), Xor()	使用 And()、Or 和 Xor()方法，可以合并两个 BitArray 对象。And()方法执行二进制的 AND，只有两个输入数组的位都设置为 1，结果位才是 1。Or()方法执行二进制的 OR，只要有一个输入数组的位设置为 1，结果位就是 1。Xor()方法是异或操作，只有一个输入数组的位设置为 1，结果位才是 1

注意：

第 6 章介绍了处理位的 C#运算符。

帮助方法 DisplayBits()迭代 BitArray，根据位的设置情况，在控制台上显示 1 或 0：

```
static void DisplayBits(BitArray bits)
{
    foreach (bool bit in bits)
    {
        Console.Write(bit ? 1 : 0);
    }
}
```

演示 BitArray 类的示例创建了一个包含 8 位的数组，其索引是 0~7。SetAll()方法把这 8 位都设置为 true。接着 Set()方法把 1 位设置为 false。除了 Set()方法之外，还可以使用索引器，例如下面的索引 5 和 7：

```
BitArray bits1 = new BitArray(8);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.Write("initialized:");
DisplayBits(bits1);
Console.WriteLine();
```

这是初始化位的显示结果：

```
initialized: 10111010
```

Not()方法会倒转 BitArray 的位：

```
DisplayBits(bits1);
bits1.Not();
Console.Write(" not ");
DisplayBits(bits1);
Console.WriteLine();
```

Not()方法的结果是所有的位全部倒转过来。如果某位是 true，执行 Not()方法的结果就

是 false，反之亦然。

```
10111010 not 01000101
```

这里创建了一个新的 BitArray。在构造函数中，使用变量 bits1 初始化数组，所以新数组与旧数组有相同的值。接着把位 0、1 和 4 的值设置为不同的值。在使用 Or()方法之前，显示位数组 bits1 和 bits2。Or()方法将改变 bits1 的值：

```
BitArray bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
DisplayBits(bits1);
Console.Write(" or ");
DisplayBits(bits2);
Console.Write(" = ");
bits1.Or(bits2);
DisplayBits(bits1);
Console.WriteLine();
```

使用 Or()方法时，从两个输入数组中提取设置位。结果是，如果某位在第一个或第二个数组中设置为 true，该位在执行 Or()方法后就是 true：

```
01000101 or 10001101 = 11001101
```

下面使用 And()方法处理 bits1 和 bits2：

```
DisplayBits(bits2);
Console.Write(" and ");
DisplayBits(bits1);
Console.Write(" = ");
bits2.And(bits1);
DisplayBits(bits2);
Console.WriteLine();
```

And()方法只把在两个输入数组中都设置为 true 的位设置为 true：

```
10001101 and 11001101 = 10001101
```

最后使用 Xor()方法进行异或操作：

```
DisplayBits(bits1);
Console.Write(" xor ");
DisplayBits(bits2);
bits1.Xor(bits2);
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

使用 Xor()方法，只有一个(不能是两个)输入数组的位设置为 1，结果位才是 1。

```
11001101 xor 10001101 = 01000000
```

10.9.2 BitVector32

如果事先知道需要的位数，就可以使用 BitVector32 结构替代 BitArray。BitVector32 效率较高，因为它是一个值类型，在整数栈上存储位。一个整数可以存储 32 位。如果需要更多的位，就可以使用多个 BitVector32 值或 BitArray。BitArray 可以根据需要增大，但 BitVector32 不能。

表 10-10 列出了 BitVector32 中与 BitArray 完全不同的成员。

表 10-10

BitVector32 的成员	说 明
Data	属性 Data 把 BitVector32 中的数据返回为整数
Item	BitVector32 的值可以使用索引器设置。索引器是重载的——可以使用掩码或 BitVector32. Section 类型的片断来确定和设置值。
CreateMask()	这是一个静态方法，用于为访问 BitVector32 中的特定位创建掩码
CreateSection()	这是一个静态方法，用于创建 32 位中的几个片断

示例代码用默认构造函数创建了一个 BitVector32，其中所有的 32 位都初始化为 false。接着创建掩码，以访问位矢量中的位。对 CreateMask()的第一个调用创建了一个访问第一位的掩码。接着调用 CreateMask()，将 bit1 设置为 1。再次调用 CreateMask()，把第一个掩码传送为参数，返回一个访问第二位(它是 2)的掩码。接着，将 bit3 设置为 4，以访问位号 3。bit4 的值是 8，以访问位号 4。

然后，使用掩码和索引器访问位矢量中的位，并设置字段：

```
BitVector32 bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);

bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);
```

BitVector32 有一个重写的 ToString()方法，它不仅显示类名，还显示 1 或 0，来说明位是否设置了，如下所示：

```
BitVector32{00000000000000000000000000011101}
```

除了用 CreateMask()方法创建掩码之外，还可以自己定义掩码，也可以一次设置多个位。十六进制值 abcdef 与二进制值 1010 1011 1100 1101 1110 1111 相同。用这个值定义的

所有位都设置了：

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

在输出中可以验证设置的位：

```
BitVector32{00000000101010111100110111101111}
```

把 32 位分别放在不同的片断中，是非常有用的。例如，IPv4 地址定义为一个 4 字节数，存储在一个整数中。可以定义 4 个片断，把这个整数拆分开。在多播 IP 消息中，使用了几个 32 位值。其中一个 32 位值放在这些片断中：16 位表示源号，8 位表示查询器的查询内部码，3 位表示查询器的健壮变量，1 位表示抑制标志，还有 4 个保留位。也可以定义自己的位含义，以节省内存。

下面的例子模拟接收到值 0x79abcdef，把这个值传送给 BitVector32 的构造函数，并设置位：

```
int received = 0x79abcdef;

BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

在控制台上显示了初始化的位：

```
BitVector32{01111001101010111100110111101111}
```

接着创建 6 个片断。第一个片断需要 12 位，由十六进制值 0xfff 定义(设置了 12 位)。片断 B 需要 8 位，C 片断需要 4 位，D 和 E 片断需要 3 位，F 片断需要 2 位。第一次调用 CreateSection()，只是接收 0xfff，为最前面的 12 位分配内存。第二次调用 CreateSection() 时，将第一个片断传送为变元，使下一个片断从第一个片断的结尾处开始。CreateSection() 返回一个 BitVector32. Section 类型的值，它包含了该片断的偏移量和掩码。

```
// sections: FF EEE DDD CCCC BBBBBBBB AAAAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xfff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

把一个 BitVector32. Section 传送给 BitVector32 的索引器，会返回一个 int，它映射到位矢量的片断上。这里使用一个帮助方法 IntToBinaryString()，获得 int 数的字符串表示：

```
Console.WriteLine("Section A: " + IntToBinaryString(bits2[sectionA], true));
Console.WriteLine("Section B: " + IntToBinaryString(bits2[sectionB], true));
Console.WriteLine("Section C: " + IntToBinaryString(bits2[sectionC], true));
Console.WriteLine("Section D: " + IntToBinaryString(bits2[sectionD], true));
Console.WriteLine("Section E: " + IntToBinaryString(bits2[sectionE], true));
Console.WriteLine("Section F: " + IntToBinaryString(bits2[sectionF], true));
```

IntoToBinaryString 接收整数中的位，返回一个包含 0 和 1 的字符串表示。在实现代码中迭代整数的 32 位。在迭代过程中，如果位设置为 1，就在 StringBuilder 的后面追加 1，否则，就追加 0。在循环中，移动一位，以检查是否设置了下一位。

```
static string IntToBinaryString(int bits, bool removeTrailingZero)
{
   StringBuilder sb = new StringBuilder(32);

   for (int i = 0; i < 32; i++)
   {
      if ((bits & 0x80000000) != 0)
      {
         sb.Append("1");
      }
      else
      {
         sb.Append("0");
      }
      bits = bits << 1;
   }
      string s = sb.ToString();
      if (removeTrailingZero)
      {
         return s.TrimStart('0');
      }
      else
      {
         return s;
      }
   }
```

结果显示了片断 A～F 的位表示，现在可以用传送给位矢量的值来验证了：

```
Section A: 110111101111
Section B: 10111100
Section C: 1010
Section D: 1
Section E: 111
Section F: 1
```

10.10 性能

许多集合类都提供了相同的功能，例如，SortedList 与 SortedDictionary 的功能几乎完全相同。但是，其性能常常有很大区别。一个集合使用的内存少，另一个集合的元素检索速度快。在 MSDN 文档中，集合的方法常常有性能提示，给出了以大 O 记号表示的操作时间：

```
O(1)
```

```
O(log n)
O(n)
```

O(1)表示无论集合中有多少数据项，这个操作需要的时间都不变。例如，ArrayList 的 Add()方法就是 O(1)。无论列表中有多少个元素，在列表尾部添加一个新元素的时间都是相同的。Count 属性会给出元素个数，所以很容易找到列表尾部。

O(n)表示对于集合中的每个元素，需要增加的时间量都是相同的。如果需要重新给集合分配内存，ArrayList 的 Add()方法就是 O(n)。改变容量，需要复制列表，复制的时间随元素的增加而线性增加。

O(log n)表示操作需要的时间随集合中元素的增加而增加，但每个元素需要增加的时间不是线性的，而是呈对数曲线。在集合中执行插入操作时，SortedDictionary<TKey,TValue>就是 O(log n)，而 SortedList<TKey,TValue>是 O(n)。这里 SortedDictionary<TKey,TValue>要快得多，因为它在树形结构中插入元素的效率比列表高得多。

10.11 小结

本章介绍了如何处理不同类型的集合。数组的大小是固定的，但可以使用列表作为动态增长的集合。队列以先进先出的方式访问元素，栈以后进先出的方式访问元素。链表可以快速插入和删除元素，但搜索操作比较慢。通过键和值可以使用字典，它的搜索和插入操作比较快。

本章还介绍了许多接口及其在集合访问和排序上的用法。探讨了一些特殊的集合，例如 BitArray 和 BitVector32，它们为处理位集合进行了优化。

第11章

内存管理和指针

本章介绍内存管理和内存访问的各个方面。尽管运行库负责为程序员处理大部分内存管理工作，但程序员仍必须理解内存管理的工作原理，了解如何处理未托管的资源。

如果很好地理解了内存管理和 C#提供的指针功能，也就能很好地集成 C#代码和原来的代码，并能在非常注重性能的系统中高效地处理内存。

本章的主要内容如下：

- 运行库如何在堆栈和堆上分配空间
- 垃圾收集的工作原理
- 如何使用析构函数和 System.IDisposable 接口来确保正确释放未托管的资源
- C#中使用指针的语法
- 如何使用指针实现基于堆栈的高性能数组

11.1 后台内存管理

C#编程的一个优点是程序员不需要担心具体的内存管理，尤其是垃圾收集器会处理所有的内存清理工作。用户可以得到像 C++语言那样的效率，而不需要考虑像在 C++中那样内存管理工作的复杂性。虽然不必手工管理内存，但如果要编写高效的代码，就仍需理解后台发生的事情。本节要介绍给变量分配内存时计算机内存中发生的情况。

注意：

本节的许多内容是没有经过事实证明的。您应把这一节看作是一般规则的简化向导，而不是实现的确切说明。

11.1.1 值数据类型

Windows 使用一个系统：虚拟寻址系统，该系统把程序可用的内存地址映射到硬件内存中的实际地址上，这些任务完全由 Windows 在后台管理，其实际结果是 32 位处理器上

的每个进程都可以使用 4GB 的内存——无论计算机上有多少硬盘空间。(在 64 位处理器上，这个数字会更大)。这个 4GB 内存实际上包含了程序的所有部分，包括可执行代码、代码加载的所有 DLL，以及程序运行时使用的所有变量的内容。这个 4GB 内存称为虚拟地址空间，或虚拟内存，为了方便起见，本章将它简称为内存。

4GB 中的每个存储单元都是从 0 开始往上排序的。要访问存储在内存的某个空间中的一个值，就需要提供表示该存储单元的数字。在任何复杂的高级语言中，例如 C#、VB、C++和 Java，编译器负责把人们可以理解的变量名称转换为处理器可以理解的内存地址。

在进程的虚拟内存中，有一个区域称为堆栈。堆栈存储不是对象成员的值数据类型。另外，在调用一个方法时，也使用堆栈存储传递给方法的所有参数的复本。为了理解堆栈的工作原理，需要注意在 C#中变量的作用域。如果变量 a 在变量 b 之前进入作用域，b 就会先出作用域。下面的代码：

```
{
   int a;
   // do something
   {
      int b;
      // do something else
   }
}
```

首先声明 a。在内部的代码块中声明了 b。然后内部的代码块终止，b 就出作用域，最后 a 出作用域。所以 b 的生存期会完全包含在 a 的生存期中。在释放变量时，其顺序总是与给它们分配内存的顺序相反，这就是堆栈的工作方式。

我们不知道堆栈在地址空间的什么地方，这些信息在进行 C#开发是不需要知道的。堆栈指针(操作系统维护的一个变量) 表示堆栈中下一个自由空间的地址。程序第一次运行时，堆栈指针指向为堆栈保留的内存块末尾。堆栈实际上是向下填充的，即从高内存地址向低内存地址填充。当数据入栈后，堆栈指针就会随之调整，以始终指向下一个自由空间。这种情况如图 11-1 所示。在该图中，显示了堆栈指针 800000(十六进制的 0xC3500)，下一个自由空间是地址 799999。

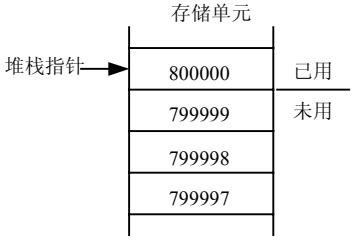

图 11-1

下面的代码会告诉编译器，需要一些存储单元以存储一个整数和一个双精度浮点数，这些存储单元会分别分配给 nRacingCars 和 engineSize，声明每个变量的代码表示开始请求访问这个变量，闭合花括号表示这两个变量出作用域的地方。

```
{
   int nRacingCars = 10;
   double engineSize = 3000.0;
   // do calculations;
}
```

假定使用如图 11-1 所示的堆栈。变量 nRacingCars 进入作用域，赋值为 10，这个值放在存储单元 799996~799999 上，这 4 个字节就在堆栈指针所指空间的下面。有 4 个字节是因为存储 int 要使用 4 个字节。为了容纳该 int，应从堆栈指针中减去 4，所以它现在指向位置 799996，即下一个自由空间 (799995)。

下一行代码声明变量 engineSize(这是一个 double)，把它初始化为 3000.0。double 要占用 8 个字节，所以值 3000.0 占据栈上的存储单元 799988~799995 上，堆栈指针减去 8，再次指向堆栈上的下一个自由空间。

当 engineSize 出作用域时，计算机就知道不再需要这个变量了。因为变量的生存期总是嵌套的，当 engineSize 在作用域中时，无论发生什么情况，都可以保证堆栈指针总是会指向存储 engineSize 的空间。为了从内存中删除这个变量，应给堆栈指针递增 8，现在指向 engineSize 使用过的空间。此处就是放置闭合花括号的地方。当 nRacingCars 也出作用域时，堆栈指针就再次递增 4，此时如果内存中又放入另一个变量，从 799999 开始的存储单元就会被覆盖，这些空间以前是存储 nRacingCars 的。

如果编译器遇到像 int i、j 这样的代码，则这两个变量进入作用域的顺序就是不确定的：两个变量是同时声明的，也是同时出作用域的。此时，变量以什么顺序从内存中删除就不重要了。编译器在内部会确保先放在内存中的那个变量后删除，这样就能保证该规则不会与变量的生存期冲突。

11.1.2　引用数据类型

堆栈有非常高的性能，但对于所有的变量来说还是不太灵活。变量的生存期必须嵌套，在许多情况下，这种要求都过于苛刻。通常我们希望使用一个方法分配内存，来存储一些数据，并在方法退出后的很长一段时间内数据仍是可以使用的。只要是用 new 运算符来请求存储空间，就存在这种可能性——例如所有的引用类型。此时就要使用托管堆。

如果以前编写过需要管理低级内存的 C++代码，就会很熟悉堆(heap)。托管堆和 C++使用的堆不同，它在垃圾收集器的控制下工作，与传统的堆相比有很显著的性能优势。

托管堆(简称为堆)是进程的可用 4GB 中的另一个内存区域。要了解堆的工作原理和如何为引用数据类型分配内存，看看下面的代码：

```
void DoWork()
{
   Customer arabel;
   arabel = new Customer();
   Customer otherCustomer2 = new EnhancedCustomer();
}
```

在这段代码中，假定存在两个类 Customer 和 EnhancedCustomer。EnhancedCustomer

类扩展了 Customer 类。

首先，声明一个 Customer 引用 arabel，在堆栈上给这个引用分配存储空间，但这仅是一个引用，而不是实际的 Customer 对象。arabel 引用占用 4 个字节的空间，包含了存储 Customer 对象的地址(需要 4 个字节把内存地址表示为 0 到 4GB 之间的一个整数值)。

然后看下一行代码：

```
arabel = new Customer();
```

这行代码完成了以下操作：首先，分配堆上的内存，以存储 Customer 实例(一个真正的实例，不只是一个地址)。然后把变量 arabel 的值设置为分配给新 Customer 对象的内存地址(它还调用合适的 Customer()构造函数初始化类实例中的字段，但我们不必担心这部分)。

Customer 实例没有放在堆栈中，而是放在内存的堆中。在这个例子中，现在还不知道一个 Customer 对象占用多少字节，但为了讨论方便，假定是 32 字节。这 32 字节包含了 Customer 实例字段，和.NET 用于识别和管理其类实例的一些信息。

为了在堆上找到一个存储新 Customer 对象的存储位置，.NET 运行库在堆中搜索，选取第一个未使用的、32 字节的连续块。为了讨论方便，假定其地址是 200000，arabel 引用占用堆栈中的 799996~799999 位置。这表示在实例化 arabel 对象前，内存的内容应如图 11-2 所示。

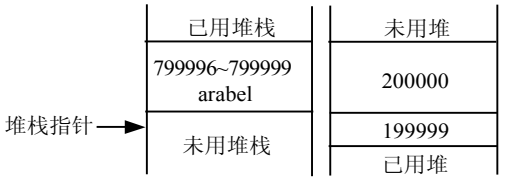

图 11-2

给 Customer 对象分配空间后，内存内容应如图 11-3 所示。注意，与堆栈不同，堆上的内存是向上分配的，所以自由空间在已用空间的上面。

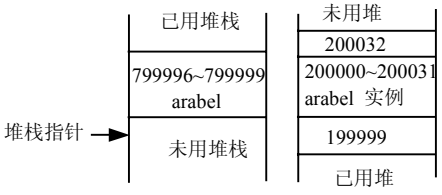

图 11-3

下一行代码声明了一个 Customer 引用，并实例化一个 Customer 对象。在这个例子中，需要在堆栈上为 mrJones 引用分配空间，同时，也需要在堆上为它分配空间：

```
Customer otherCustomer2 = new EnhancedCustomer();
```

该行把堆栈上的 4 字节分配给 otherCustomer2 引用，它存储在 799992~799995 位置上，而 otherCustomer2 对象在堆上从 200032 开始向上分配空间。

从这个例子可以看出，建立引用变量的过程要比建立值变量的过程更复杂，且不能避免性能的降低。实际上，我们对这个过程进行了过分的简化，因为.NET 运行库需要保存堆的状态信息，在堆中添加新数据时，这些信息也需要更新。尽管有这些性能损失，但仍有一种机制，在给变量分配内存时，不会受到堆栈的限制。把一个引用变量的值赋予另一个相同类型的变量，就有两个引用内存中同一对象的变量了。当一个引用变量出作用域时，它会从堆栈中删除，如上一节所述，但引用对象的数据仍保留在堆中，一直到程序停止，或垃圾收集器删除它为止，而只有在该数据不再被任何变量引用时，才会被删除。

这就是引用数据类型的强大之处，在 C#代码中广泛使用了这个特性。这说明，我们可以对数据的生存期进行非常强大的控制，因为只要有对数据的引用，该数据就肯定存在于堆上。

11.1.3　垃圾收集

由上面的讨论和图可以看出，托管堆的工作方式非常类似于堆栈，在某种程度上，对象会在内存中一个挨一个地放置，这样就很容易使用指向下一个空闲存储单元的堆指针，来确定下一个对象的位置。在堆上添加更多的对象时，也容易调整。但这比较复杂，因为基于堆的对象的生存期与引用它们的基于堆栈的变量的作用域不匹配。

在垃圾收集器运行时，会在堆中删除不再引用的所有对象。在完成删除动作后，堆会立即把对象分散开来，与已经释放的内存混合在一起，如图 11-4 所示。

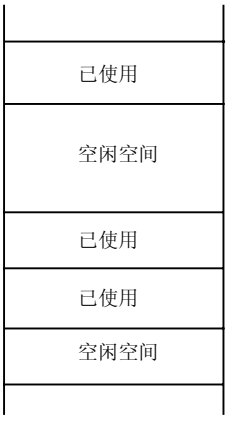

图　11-4

如果托管的堆也是这样，在其上给新对象分配内存就成为一个很难处理的过程，运行库必须搜索整个堆，才能找到足够大的内存块来存储每个新对象。但是，垃圾收集器不会让堆处于这种状态。只要它释放了能释放的所有对象，就会压缩其他对象，把它们都移动回堆的端部，再次形成一个连续的块。因此，堆可以继续像堆栈那样确定在什么地方存储

新对象。当然，在移动对象时，这些对象的所有引用都需要用正确的新地址来更新，但垃圾收集器也会处理更新问题。

垃圾收集器的这个压缩操作是托管的堆与旧未托管的堆的区别所在。使用托管的堆，就只需要读取堆指针的值即可，而不是搜索链接地址列表，来查找一个地方来放置新数据。因此，在.NET 下实例化对象要快得多。有趣的是，访问它们也比较快，因为对象会压缩到堆上相同的内存区域，这样需要交换的页面较少。Microsoft 相信，尽管垃圾收集器需要做一些工作，压缩堆，修改它移动的所有对象引用，致使性能降低，但这些性能会得到弥补。

注意：

一般情况下，垃圾收集器在.NET 运行库认为需要时运行。可以通过调用 System.GC.Collect()，强迫垃圾收集器在代码的某个地方运行，System.GC 是一个表示垃圾收集器的.NET 基类， Collect()方法则调用垃圾收集器。但是，这种方式适用的场合很少，例如，代码中有大量的对象刚刚停止引用，就适合调用垃圾收集器。但是，垃圾收集器的逻辑不能保证在一次垃圾收集过程中，所有未引用的对象都从堆中删除。

11.2 释放未托管的资源

垃圾收集器的出现意味着，通常不需要担心不再需要的对象，只要让这些对象的所有引用都超出作用域，并允许垃圾收集器在需要时释放资源即可。但是，垃圾收集器不知道如何释放未托管的资源(例如文件句柄、网络连接和数据库连接)。托管类在封装对未托管资源的直接或间接引用时，需要制定专门的规则，确保未托管的资源在回收类的一个实例时释放。

在定义一个类时，可以使用两种机制来自动释放未托管的资源。这些机制常常放在一起实现，因为每个机制都为问题提供了略为不同的解决方法。这两个机制是：

- 声明一个析构函数(或终结器)，作为类的一个成员
- 在类中执行 System.IDisposable 接口

下面依次讨论这两个机制，然后介绍如何同时实现它们，以获得最佳的效果。

11.2.1 析构函数

前面介绍了构造函数可以指定必须在创建类的实例时进行的某些操作，在垃圾收集器删除对象之前，也可以调用析构函数。由于执行这个操作，所以析构函数初看起来似乎是放置释放未托管资源、执行一般清理操作的代码的最佳地方。但是，事情并不是如此简单。

注意：

在讨论 C#中的析构函数时，在底层的.NET 结构中，这些函数称为终结器(finalizer)。在 C#中定义析构函数时，编译器发送给程序集的实际上是 Finalize()方法。这不会影响源代码，但如果需要查看程序集的内容，就应知道这个事实。

C++开发人员应很熟悉析构函数的语法，它看起来类似于一个方法，与包含类同名，

但前面加上了一个发音符号(~)。它没有返回类型，不带参数，没有访问修饰符。下面是一个例子：

```
class MyClass
{
   ~MyClass()
   {
      // destructor implementation
   }
}
```

C#编译器在编译析构函数时，会隐式地把析构函数的代码编译为 Finalize()方法的对应代码，确保执行父类的 Finalize()方法。下面列出了编译器为~MyClass()析构函数生成的 IL 的对应 C#代码：

```
protected override void Finalize()
{
   try
   {
      // destructor implementation
   }
   finally
   {
      base. Finalize();
    }
}
```

如上所示，在~MyClass()析构函数中执行的代码封装在 Finalize()方法的一个 try 块中。对父类 Finalize()方法的调用放在 finally 块中，确保该调用的执行。第 13 章会讨论 try 块和 finally 块。

有经验的 C++开发人员大量使用了析构函数，有时不仅用于清理资源，还提供调试信息或执行其他任务。C#析构函数的使用要比在 C++中少得多，与 C++析构函数相比，C#析构函数的问题是它们的不确定性。在删除 C++对象时，其析构函数会立即运行。但由于垃圾收集器的工作方式，无法确定 C#对象的析构函数何时执行。所以，不能在析构函数中放置需要在某一时刻运行的代码，也不应使用能以任意顺序对不同类实例调用的析构函数。如果对象占用了宝贵而重要的资源，应尽快释放这些资源，此时就不能等待垃圾收集器来释放了。

另一个问题是 C#析构函数的执行会延迟对象最终从内存中删除的时间。没有析构函数的对象会在垃圾收集器的一次处理中从内存中删除，但有析构函数的对象需要两次处理才能删除：第一次调用析构函数时，没有删除对象，第二次调用才真正删除对象。另外，运行库使用一个线程来执行所有对象的 Finalize()方法。如果频繁使用析构函数，而且使用它们执行长时间的清理任务，对性能的影响就会非常显著。

11.2.2 IDisposable 接口

在 C#中，推荐使用 System.IDisposable 接口替代析构函数。IDisposable 接口定义了一个模式(具有语言级的支持)，为释放未托管的资源提供了确定的机制，并避免产生析构函数固有的与垃圾函数器相关的问题。IDisposable 接口声明了一个方法 Dispose()，它不带参数，返回 void，Myclass 的方法 Dispose()的执行代码如下：

```
class Myclass : IDisposable
{
   public void Dispose()
   {
      // implementation
   }
}
```

Dispose()的执行代码显式释放由对象直接使用的所有未托管资源，并在所有实现 IDisposable 接口的封装对象上调用 Dispose()。这样，Dispose()方法在释放未托管资源的时间方面提供了精确的控制。

假定有一个类 ResourceGobbler，它使用某些外部资源，且执行 IDisposable 接口。如果要实例化这个类的实例，使用它，然后释放它，就可以使用下面的代码：

```
ResourceGobbler theInstance = new ResourceGobbler();

   // do your processing

theInstance.Dispose();
```

如果在处理过程中出现异常，这段代码就没有释放 theInstance 使用的资源，所以应使用 try 块(详见第 13 章)，编写下面的代码：

```
ResourceGobbler theInstance = null;
try

{
   theInstance = new ResourceGobbler();

    // do your processing
}
finally
{
   if (theInstance != null)
  {
theInstance.Dispose();
    }
}
```

即使在处理过程中出现了异常，这个版本也可以确保总是在 theInstance 上调用 Dispose()，总是释放由 theInstance 使用的资源。但是，如果总是要重复这样的结构，代码就很容易被混淆。C#提供了一种语法，可以确保在执行 IDisposable 接口的对象的引用超出

作用域时，在该对象上自动调用 Dispose()。该语法使用了 using 关键字来完成这一工作——但目前，在完全不同的环境下，它与命名空间没有关系。下面的代码生成与 try 块相对应的 IL 代码：

```
using (ResourceGobbler theInstance = new ResourceGobbler())
{
  // do your processing
}
```

using 语句的后面是一对圆括号，其中是引用变量的声明和实例化，该语句使变量放在随附的语句块中。另外，在变量超出作用域时，即使出现异常，也会自动调用其 Dispose()方法。如果已经使用 try 块来捕获其他异常，就会比较清晰，如果避免使用 using 语句，仅在已有的 try 块的 finally 子句中调用 Dispose()，还可以避免进行额外的缩进。

注意：

对于某些类来说，使用 Close()方法要比 Dispose()更富有逻辑性，例如，在处理文件或数据库连接时就是这样。在这些情况下，常常实现 IDisposable 接口，再执行一个独立的 Close()方法，来调用 Dispose()。这种方法在类的使用上比较清晰，还支持 C#提供的 using 语句。

11.2.3　实现 IDisposable 接口和析构函数

前面的章节讨论了类所使用的释放未托管资源的两种方式：

- 利用运行库强制执行的析构函数，但析构函数的执行是不确定的，而且，由于垃圾收集器的工作方式，它会给运行库增加不可接受的系统开销。
- IDisposable 接口提供了一种机制，允许类的用户控制释放资源的时间，但需要确保执行 Dispose()。

一般情况下，最好的方法是执行这两种机制，获得这两种机制的优点，克服其缺点。假定大多数程序员都能正确调用 Dispose()，同时把执行析构函数作为一种安全的机制，以防没有调用 Dispose()。下面是一个双重实现的例子：

```
public class ResourceHolder : IDisposable
{
   private bool isDispose = false;

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }

  protected virtual void Dispose(bool disposing)
  {
    if (!isDisposed)
    {
```

```
            if (disposing)
            {
                // Cleanup managed objects by calling their
                // Dispose() methods.
            }
            // Cleanup unmanaged objects
         }
         isDisposed=true;
      }

      ~ResourceHolder()
      {
         Dispose (false);
      }

      public void SomeMethod()
      {
         // Ensure object not already disposed before execution of any method
         if(isDisposed)
         {
            throw new ObjectDisposedException("ResourceHolder");
         }

         // method implementation…
      }
   }
```

可以看出，Dispose()有第二个 protected 重载方法，它带一个 bool 参数，这是真正完成清理工作的方法。Dispose(bool)由析构函数和 IDisposable.Dispose()调用。这个方式的重点是确保所有的清理代码都放在一个地方。

传递给 Dispose(bool)的参数表示 Dispose(bool)是由析构函数调用，还是由 IDisposable.Dispose()调用——Dispose(bool)不应从代码的其他地方调用，其原因是：

- 如果客户调用 IDisposable.Dispose()，该客户就指定应清理所有与该对象相关的资源，包括托管和非托管的资源。
- 如果调用了析构函数，原则上所有的资源仍需要清理。但是在这种情况下，析构函数必须由垃圾收集器调用，而且不应访问其他托管的对象，因为我们不再能确定它们的状态了。在这种情况下，最好清理已知的未托管资源，希望引用的托管对象还有析构函数，执行自己的清理过程。

isDisposed 成员变量表示对象是否已被删除，并允许确保不多次删除成员变量。它还允许在执行实例方法之前测试对象是否已释放，如 SomeMethod()所示。这个简单的方法不是线程安全的，需要调用者确保在同一时刻只有一个线程调用方法。要求客户进行同步是一个合理的假定，在整个.NET 类库中反复使用了这个假定(例如在集合类中)。第 18 章将讨论线程和同步。

最后，IDisposable.Dispose()包含一个对 System.GC.SuppressFinalize()方法的调用。GC 表示垃圾收集器，SuppressFinalize()方法则告诉垃圾收集器有一个类不再需要调用其析构

函数了。因为 Dispose()已经完成了所有需要的清理工作，所以析构函数不需要做任何工作。调用 SuppressFinalize()就意味着垃圾收集器认为这个对象根本没有析构函数。

11.3　不安全的代码

如前面的章节所述，C#非常擅长于隐藏基本内存管理，因为它使用了垃圾收集器和引用。但是，有时需要直接访问内存，例如由于性能问题，要在外部(非.NET 环境)的 DLL 中访问一个函数，该函数需要把一个指针当作参数来传递(许多 Windows API 函数就是这样)。本节将论述 C#直接访问内存内容的功能。

11.3.1　指针

下面把指针当作一个新论题来介绍，而实际上，指针并不是新东西，因为在代码中可以自由使用引用，而引用就是一个类型安全的指针。前面已经介绍了表示对象和数组的变量实际上包含存储相应数据(引用)的内存地址。指针只是一个以与引用相同的方式存储地址的变量。其区别是 C#不允许直接访问引用变量包含的地址。有了引用后，从语法上看，变量就可以存储引用的实际内容。

C#引用主要用于使 C#语言易于使用，防止用户无意中执行某些破坏内存中内容的操作，另一方面，使用指针，就可以访问实际内存地址，执行新类型的操作。例如，给地址加上 4 字节，就可以查看甚至修改存储在新地址中的数据。

下面是使用指针的两个主要原因：

- 向后兼容性。尽管.NET 运行库提供了许多工具，但仍可以调用内部的 Windows API 函数。 对于某些操作来说，这可能是完成任务的唯一方式。这些 API 函数都是用 C 语言编写的，通常要求把指针作为其参数。但在许多情况下，还可以使用 DllImport 声明，以避免使用指针，例如使用 System.IntPtr 类。
- 性能。在一些情况下，速度是最重要的，而指针可以提供最优性能。假定用户知道自己在做什么，就可以确保以最高效的方式访问或处理数据。但是，注意在代码的其他区域中，不使用指针，也可以对性能做必要的改进。请使用代码配置文件，查找代码中的瓶颈，代码配置文件随 VS2005 一起安装。

但是，这种低级内存访问也是有代价的。使用指针的语法比引用类型更复杂。而且，指针使用起来比较困难，需要非常高的编程技巧和很强的能力，仔细考虑代码所完成的逻辑操作，才能成功地使用指针。如果不仔细，使用指针很容易在程序中引入微妙的难以查找的错误。例如很容易重写其他变量，导致堆栈溢出，访问某些没有存储变量的内存区域，甚至重写.NET 运行库所需要的代码信息，因而使程序崩溃。

另外，如果使用指针，就必须为代码获取代码访问安全机制的高级别信任，否则就不能执行。在默认的代码访问安全策略中，只有代码运行在本地机器上，这才是可能的。如果代码必须运行在远程地点，例如 Internet，用户就必须给代码授予额外的许可，代码

才能工作。除非用户信任您和代码，否则他们不会授予这些许可，第 19 章将讨论代码访问安全性。

尽管有这些问题，但指针在编写高效的代码时是一种非常强大和灵活的工具，这里就介绍指针的使用。

注意：

这里强烈建议不要使用指针，因为如果使用指针，代码不仅难以编写和调试，而且无法通过 CLR 的内存类型安全检查(详见第 1 章)。

1. 编写不安全的代码

因为使用指针会带来相关的风险，所以 C#只允许在特别标记的代码块中使用指针。标记代码所用的关键字是 unsafe。下面的代码把一个方法标记为 unsafe：

```
unsafe int GetSomeNumber()
{
   // code that can use pointers
}
```

任何方法都可以标记为 unsafe——无论该方法是否应用了其他修饰符(例如，静态方法、虚拟方法等)。在这种方法中，unsafe 修饰符还会应用到方法的参数上，允许把指针用作参数。还可以把整个类或结构标记为 unsafe，表示所有的成员都是不安全的：

```
unsafe class MyClass
{
   // any method in this class can now use pointers
}
```

同样，可以把成员标记为 unsafe：

```
class MyClass
{
   unsafe int *pX;   // declaration of a pointer field in a class
}
```

也可以把方法中的一个代码块标记为 unsafe：

```
void MyMethod()
{
   // code that doesn't use pointers
   unsafe
   {
      // unsafe code that uses pointers here
   }
   // more 'safe' code that doesn't use pointers
}
```

但要注意，不能把局部变量本身标记为 unsafe：

```
int MyMethod()
{
   unsafe int *pX;   // WRONG
}
```

如果要使用不安全的局部变量，就需要在不安全的方法或语句块中声明和使用它。在使用指针前还有一步要完成。C#编译器会拒绝不安全的代码，除非告诉编译器代码包含不安全的代码块。标记所用的关键字是 unsafe。因此，要编译包含不安全代码块的文件 MySource.cs(假定没有其他编译器选项)，就要使用下述命令：

```
csc /unsafe MySource.cs
```

或者

```
csc -unsafe MySource.cs
```

注意：

如果使用 Visual Studio 2005，就可以在项目属性窗口中找到编译不安全代码的选项。

2. 指针的语法

把代码块标记为 unsafe 后，就可以使用下面的语法声明指针：

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

这段代码声明了 4 个变量，pWidth 和 pHeight 是整数指针，pResult 是 double 型指针，pFlags 是 byte 型的指针数组。我们常常在指针变量名的前面使用前缀 p 来表示这些变量是指针。在变量声明中，符号*表示声明一个指针，换言之，就是存储特定类型的变量的地址。

提示：

C++开发人员应注意，这个语法与 C#中的语法是不同的。C#语句中的 int* pX, pY; 对应于 C++ 语句中的 int *pX, *pY; 在 C#中，*符号与类型相关，而不是与变量名相关。

声明了指针类型的变量后，就可以用与一般变量的方式使用它们，但首先需要学习另外两个运算符：

- & 表示“取地址”，并把一个值数据类型转换为指针，例如 int 转换为*int。这个运算符称为寻址运算符。
- * 表示“获取地址的内容”，把一个指针转换为值数据类型(例如，*float 转换为 float)。这个运算符称为“间接寻址运算符”(有时称为“取消引用运算符”)。

从这些定义中可以看出，&和*的作用是相反的。

注意：

符号&和*也表示按位 AND(&)和乘法(*)运算符，那么如何以这种方式使用它们？答案是在实际使用时它们是不会混淆的:用户和编译器总是知道在什么情况下这两个符号有什么含义，因为按照新指针的定义，这些符号总是以一元运算符的形式出现——它们只作用

于一个变量，并出现在代码中变量的前面。另一方面，按位 AND 和乘法运算符是二元运算符，它们需要两个操作数。

下面的代码说明了如何使用这些运算符：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
```

首先声明一个整数 x，其值是 10。接着声明两个整数指针 pX 和 pY。然后把 pX 设置为指向 x(换言之，把 pX 的内容设置为 x 的地址)。把 pX 的值赋予 pY，所以 pY 也指向 x。最后，在语句*pY = 20 中，把值 20 赋予 pY 指向的地址。实际上是把 x 的内容改为 20，因为 pY 指向 x。注意在这里，变量 pY 和 x 之间没有任何关系。只是此时 pY 碰巧指向存储 x 的存储单元而已。

要进一步理解这个过程，假定 x 存储在堆栈的存储单元 0x12F8C4 到 0x12F8C7 中(十进制就是 1243332 到 1243335，即有 4 个存储单元，因为 int 占用 4 字节)。因为堆栈向下分配内存，所以变量 pX 存储在 0x12F8C0 到 0x12F8C3 的位置上，pY 存储在 0x12F8BC 到 0x12F8BF 的位置上。注意，pX 和 pY 也分别占用 4 字节。这不是因为 int 占用 4 字节，而是因为在 32 位处理器上，需要用 4 字节存储一个地址。利用这些地址，在执行完上述代码后，堆栈应如图 11-5 所示。

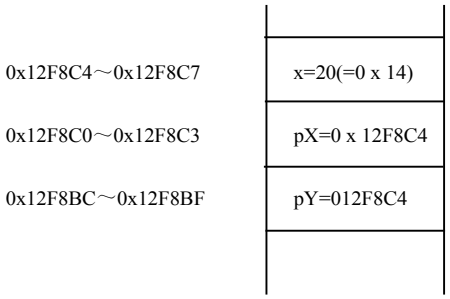

图 11-5

注意：

这个示例使用 int 来说明该过程，其中 int 存储在 32 位处理器中堆栈的连续空间上，但并不是所有的数据类型都会存储在连续的空间中。原因是 32 位处理器最擅长于在 4 字节的内存块中获取数据。这种机器上的内存会分解为 4 字节的块，在 Windows 上，每个块都时常称为 DWORD，因为这是 32 位无符号 int 在.NET 出现之前的名字。这是从内存中获取 DWORD 的最高效的方式——跨越 DWORD 边界存储数据通常会降低硬件的性能。因此，.NET 运行库通常会给某些数据类型加上一些空间，使它们占用的内存是 4 的倍数。例如，short 数据占用 2 字节，但如果把一个 short 放在堆栈中，堆栈指针仍会减少 4，而不是 2，这样，下一个存储在堆栈中的变量就仍从 DWORD 的边界开始存储。

可以把指针声明为任意一种数据类型——即任何预定义的数据类型 uint、int 和 byte 等，也可以声明为一个结构。但是不能把指针声明为一个类或数组，因为这么做会使垃圾收集器出现问题。为了正常工作，垃圾收集器需要知道在堆上创建了什么类实例，它们在什么地方。但如果代码使用指针处理类，将很容易破坏堆中.NET 运行库为垃圾收集器维护的与类相关的信息。在这里，垃圾收集器可以访问的数据类型称为托管类型，而指针只能声明为非托管类型，因为垃圾收集器不能处理它们。

3. 将指针转换为整数类型

由于指针实际上存储了一个表示地址的整数，所以任何指针中的地址都可以转换为任何整数类型。指针到整数类型的转换必须是显式指定的，隐式的转换是不允许的。例如，编写下面的代码是合法的：

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

把指针 pX 中包含的地址转换为一个 uint，存储在变量 y 中。接着把 y 转换回 int*，存储在新变量 pD 中。因此 pD 也指向 x 的值。

把指针的值转换为整数类型的主要目的是显示它。Console.Write()和 Console.WriteLine()方法没有带指针的重载方法，所以必须把指针转换为整数类型，这两个方法才能接受和显示它们：

```
Console.WriteLine("Address is" + pX);   // wrong – will give a
                                        // compilation error
Console.WriteLine("Address is" + (uint) pX);   // OK
```

可以把一个指针转换为任何整数类型，但是，因为在 32 位系统上，地址占用 4 字节，把指针转换为不是 uint、long 或 ulong 的数据类型，肯定会导致溢出错误(int 也可能导致这个问题，因为它的取值范围是－20 亿~20 亿，而地址的取值范围是 0~40 亿)。C#是用于 64 位处理器的，地址占用 8 字节。因此在这样的系统上，把指针转换为非 ulong 的类型，就可能导致溢出错误。还要注意，checked 关键字不能用于涉及指针的转换。对于这种转换，即使在设置 checked 的情况下，发生溢出时也不会抛出异常。.NET 运行库假定，如果使用指针，就知道自己要做什么，并希望出现溢出。

4. 指针类型之间的转换

也可以在指向不同类型的指针之间进行显式的转换。例如：

```
byte aByte = 8;
```

```
byte* pByte= &aByte;
double* pDouble = (double*)pByte;
```

这是一段合法的代码，但如果要执行这段代码，就要小心了。在上面的示例中，如果要查找指针 pDouble 指向的 double，就会查找包含 1 字节(aByte)的内存，并和一些其他内存合并在一起，把它当作包含一个 double 的内存区域来对待——这不会得到一个有意义的值。但是，可以在类型之间转换，实现类型的统一，或者把指针转换为其他类型，例如把指针转换为 sbyte，检查内存的单个字节。

5. void 指针

如果要使用一个指针，但不希望指定它指向的数据类型，就可以把指针声明为 void：

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

void 型指针的主要用途是调用需要 void*型参数的 API 函数。在 C#语言中，使用 void 指针的情况并不是很多。特殊情况下，如果试图使用*运算符间接引用 void 指针，编译器就会标记一个错误。

6. 指针的算法

可以给指针加减整数。但是，编译器很智能，知道如何执行这个操作。例如，假定有一个 int 指针，要在其值上加 1。编译器会假定我们要查找 int 后面的存储单元，因此会给该值加上 4 字节，即加上 int 的字节数。如果这是一个 double 指针，加 1 就表示在指针的值上加 8 字节，即 double 的字节数。只有指针是指向 byte 或 sbyte(都是 1 字节)，才会给该指针的值加上 1。

可以对指针使用运算符+、－、+=、－=、++和－－，这些运算符右边的变量必须是 long 或 ulong 类型。

注意：
不允许对 void 指针执行算术运算。

例如，假定有如下定义：

```
uint u = 3;
byte b = 8;
double d = 10.0;
uint* pUint= &u;       // size of a uint is 4
byte* pByte = &b;      // size of a byte is 1
double* pDouble = &d;   // size of a double is 8
```

下面假定这些指针的地址是：

- pUint：1243332

- pByte: 1243328
- pDouble: 1243320

执行这段代码后：

```
++pUint;            // adds (1*4)= 4 bytes to pUint
pByte -= 3;          // subtracts (3*1)=3 bytes from pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 bytes (4*8 bytes)
```

指针应包含的内容是：

- pUint: 1243336
- pByte: 1243325
- pDouble2: 1243352

提示：

给类型为 T 的指针加上 X，其中指针的值为 P，则得到的结果是 P + X*(sizeof(T))。

注意：

使用这个规则时要小心。如果给定类型的连续值存储在连续的存储单元中，指针加法就允许在存储单元中移动指针。但如果类型是 byte 或 char，其总字节数就不是 4 的倍数，连续值就不是默认地存储在连续的存储单元中。

如果两个指针都指向相同的数据类型，也可以把一个指针从另一个指针中减去。此时，结果是一个 long，其值是指针值的差被该数据类型所占用的字节数整除的结果：

```
double* pD1 = (double*)1243324;    // note that it is perfectly valid to
                                   // initialize a pointer like this.
double* pD2 = (double*)1243300;
long L = pD1-pD2;              // gives the result 3 (=24/sizeof(double))
```

7. sizeof 运算符

这一节将介绍如何确定各种数据类型的大小。如果需要在代码中使用类型的大小，就可以使用 sizeof 运算符，它的参数是数据类型的名称，返回该类型占用的字节数。例如：

```
int x = sizeof(double);
```

这将设置 x 的值为 8。

使用 sizeof 的优点是不必在代码中硬编码数据类型的大小，使代码的移植性更强。对于预定义的数据类型，sizeof 返回表 11-1 所示的值。

表　11-1

sizeof(sbyte) = 1;	sizeof(byte) = 1;
sizeof(short) = 2;	sizeof(ushort) = 2;
sizeof(int) = 4;	sizeof(uint) = 4;

(续表)

sizeof(long) = 8;	sizeof(ulong) = 8;
sizeof(char) = 2;	sizeof(float) = 4;
sizeof(double) = 8;	sizeof(bool) = 1;

也可以对自己定义的结构使用 sizeof，但此时得到的结果取决于结构中的字段。不能对类使用 sizeof。它只能用于不安全的代码块。

8. 结构指针：指针成员访问运算符

结构指针的工作方式与预定义值类型的指针的工作方式是一样的。但是这有一个条件：结构不能包含任何引用类型，这是因为前面介绍的一个限制—— 指针不能指向任何引用类型。为了避免这种情况，如果创建一个指针，它指向包含引用类型的结构，编译器就会标记一个错误。

假定定义了如下结构：

```
struct MyStruct
{
   public long X;
   public float F;
}
```

就可以给它定义一个指针：

```
MyStruct* pStruct;
```

对其进行初始化：

```
MyStruct Struct = new MyStruct();
pStruct = &Struct;
```

也可以通过指针访问结构的成员值：

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

但是，这个语法有点复杂。因此，C#定义了另一个运算符，用一种比较简单的语法，通过指针访问结构的成员，该语法称为指针成员访问运算符，其符号是一个短划线，后跟一个大于号：->。

注意：

C++开发人员会认识指针成员访问操作符。因为 C++使用这个符号完成相同的任务。

使用这个指针成员访问运算符，上述代码可以重写为：

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

也可以直接把合适类型的指针设置为指向结构中的一个字段：

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

或者

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

9. 类成员指针

前面说过，不能创建指向类的指针，这是因为垃圾收集器不维护指针的任何信息，只维护所引用的信息，因此创建指向类的指针会使垃圾收集器不能正常工作。

但是，大多数类都包含值类型的成员，可以为这些值类型成员创建指针，但这需要一种特殊的语法。例如，假定把上面示例中的结构重写为类：

```
class MyClass
{
   public long X;
   public float F;
}
```

然后就可以为它的字段 X 和 F 创建指针了，方法与前面一样。但这么做会生成一个编译错误：

```
MyClass myObject = new MyClass();
long* pL = &( myObject.X);   // wrong--compilation error
float* pF = &( myObject.F);   // wrong--compilation error
```

X 和 F 都是非托管类型，它们嵌入在一个对象中，存储在堆上。在垃圾收集的过程中，垃圾收集器会把 MyObject 移动到内存的一个新单元上，这样， pL 和 pF 就会指向错误的存储单元。由于存在这个问题，所以编译器不允许以这种方式把托管类型的成员地址分配给指针。

解决这个问题的方法是使用 fixed 关键字，它会告诉垃圾收集器，类实例的某些成员有指向它们的指针，所以这些实例不能移动。如果要声明一个指针，使用 fixed 的语法如下所示：

```
MyClass myObject = new MyClass();
fixed (long* pObject = &( myObject.X))
{
   // do something
}
```

在关键字 fixed 后面的圆括号中，定义和初始化指针变量。这个指针变量(在本例中是 pObject)的作用域是花括号标记的 fixed 块。这样，垃圾收集器就知道，在执行 fixed 块中的代码时，不能移动 MyObject 对象。

如果要声明多个这样的指针，可以在同一个代码块前放置多个 fixed 语句：

```
MyClass myObject = new MyClass();
fixed (long* pX = &( myObject.X))
fixed (float* pF = &( myObject.F))
{
   // do something
}
```

如果要在不同的阶段固定几个指针，还可以嵌套整个 fixed 块：

```
MyClass myObject = new MyClass();
fixed (long* pX = &( myObject.X))
{
   // do something with pX
   fixed (float* pF = &( myObject.F))
   {
      // do something else with pF
   }
}
```

也可以在同一个 fixed 语句中初始化多个变量，但这些变量的类型必须相同：

```
MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &( myObject.X), pX2 = &( myObject2.X))
{
   // etc.
}
```

在上述情况中，是否声明不同的指针，让它们指向相同或不同对象中的字段，或者指向不与类实例相关的静态字段，这一点是不重要的。

11.3.2 指针示例：PointerPlayaround

下面给出一个使用指针的示例：PointerPlayaround。它执行一些简单的指针操作，显示结果，还允许查看内存中发生的情况，并确定变量存储在什么地方：

```
using System;

namespace Wrox.ProCSharp.Memory
{
   class MainEntryPoint
   {
      static unsafe void Main()
      {
         int x=10;
         short y =-1;
         byte y2 = 4;
         double z = 1.5;
         int* pX = &x;
         short* pY = &y;
```

```
            double* pZ = &z;

            Console.WriteLine(
               "Address of x is 0x{0:X}, size is {1}, value is {2}",
               (uint)&x, sizeof(int), x);
            Console.WriteLine(
               "Address of y is 0x{0:X}, size is {1}, value is {2}",
               (uint)&y, sizeof(short), y);
            Console.WriteLine(
               "Address of y2 is 0x{0:X}, size is {1}, value is {2}",
               (uint)&y2, sizeof(byte), y2);
            Console.WriteLine(
               "Address of z is 0x{0:X}, size is {1}, value is {2}",
               (uint)&z, sizeof(double), z);
            Console.WriteLine(
               "Address of pX=&x is 0x{0:X}, size is {1}, value is 0x{2:X}",
               (uint)&pX, sizeof(int*), (uint)pX);
            Console.WriteLine(
               "Address of pY=&y is 0x{0:X}, size is {1}, value is 0x{2:X}",
               (uint)&pY, sizeof(short*), (uint)pY);
            Console.WriteLine(
               "Address of pZ=&z is 0x{0:X}, size is {1}, value is 0x{2:X}",
               (uint)&pZ, sizeof(double*), (uint)pZ);

            *pX = 20;
            Console.WriteLine("After setting *pX, x = {0}", x);
            Console.WriteLine("*pX = {0}", *pX);

            pZ = (double*)pX;
            Console.WriteLine("x treated as a double = {0}", *pZ);

            Console.ReadLine();
         }
      }
   }
```

这段代码声明了 4 个值变量：

- int x
- short y
- byte y2
- double z

还声明了指向这 3 个值的指针：pX、pY、pZ。

然后显示这 3 个变量的值，以及它们的大小和地址。注意在获取 pX, pY 和 pZ 的地址时，我们查看的是指针的指针，即值的地址的地址！还要注意，与显示地址的常见方式一致，在 Console.WriteLine()命令中使用{0:X}格式说明符，确保该内存地址以十六进制格式显示。

最后，使用指针 pX 把 x 的值改为 20，执行一些指针转换，如果把 x 的内容当作 double

类型，就会得到无意义的结果。

编译运行这段代码，在得到的结果中，我们将列出用/unsafe 标志进行编译和不用/unsafe 标志进行编译的结果：

```
csc PointerPlayaround.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50215.33
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50215
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

PointerPlayaround.cs(7,26): error CS0227: Unsafe code may only appear if
      compiling with /unsafe

csc /unsafe PointerPlayaround.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50215.33
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50215
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

PointerPlayaround
Address of x is 0x12F4B0, size is 4, value is 10
Address of y is 0x12F4AC, size is 2, value is -1
Address of y2 is 0x12F4A8, size is 1, value is 4
Address of z is 0x12F4A0, size is 8, value is 1.5
Address of pX=&x is 0x12F49C, size is 4, value is 0x12F4B0
Address of pY=&y is 0x12F498, size is 4, value is 0x12F4AC
Address of pZ=&z is 0x12F494, size is 4, value is 0x12F4A0
After setting *pX, x = 20
*pX = 20
x treated as a double = 2.86965129997082E-308
```

检查这些结果，可以证实本章前面的“后台内存管理”一节描述的堆栈操作，即堆栈给变量向下分配内存。注意，这还证实了堆栈中的内存块总是按照 4 字节的倍数进行分配。例如，y 是一个 short(其大小为 2 字节)，其地址是 1242284(十进制)，表示为该变量分配的内存区域是 1242284~1242287。如果.NET 运行库严格地逐个排列变量，则 y 应只占用 2 个存储单元 1242284 和 1242285。

下一个示例 PointerPlayaround2 介绍指针的算术，以及结构指针和类成员指针。开始时，定义一个结构 CurrencyStruct，把货币值表示为美元和美分，再定义一个对应的类 CurrencyClass：

```
struct CurrencyStruct
{
   public long Dollars;
   public byte Cents;

   public override string ToString()
   {
      return "$" + Dollars + "." + Cents;
   }
```

```
    }

    class CurrencyClass
    {
       public long Dollars;
       public byte Cents;

       public override string ToString()
       {
          return "$" + Dollars + "." + Cents;
       }
    }
```

定义好了结构和类后，就可以对它们应用指针了。下面的代码是一个新的示例。这段代码比较长，我们对此将做详细讲解。首先显示 CurrencyStruct 结构的字节数，创建它的两个实例和一些指针，再使用 pAmount 指针初始化一个 CurrencyStruct 结构 amount1，显示变量的地址：

```
public static unsafe void Main()
{
   Console.WriteLine(
      "Size of Currency struct is " + sizeof(CurrencyStruct));
   CurrencyStruct amount1, amount2;
   CurrencyStruct* pAmount = &amount1;
   long* pDollars = &(pAmount->Dollars);
   byte* pCents = &(pAmount->Cents);

   Console.WriteLine("Address of amount1 is 0x{0:X}", (uint)&amount1);
   Console.WriteLine("Address of amount2 is 0x{0:X}", (uint)&amount2);
   Console.WriteLine("Address of pAmount is 0x{0:X}", (uint)&pAmount);
   Console.WriteLine("Address of pDollars is 0x{0:X}", (uint)&pDollars);
   Console.WriteLine("Address of pCents is 0x{0:X}", (uint)&pCents);
   pAmount->Dollars = 20;
   *pCents = 50;
   Console.WriteLine("amount1 contains " + amount1);
```

现在根据堆栈的工作方式，执行一些指针操作。因为变量是按顺序声明的，所以 amount2 存储在 amount1 后面的地址上，sizeof(CurrencyStruct)返回 16(见后面的的屏幕输出)，所以 CurrencyStruct 占用的字节数是 4 的倍数。在递减了 Currency 指针后，它就指向 amount2：

```
   --pAmount;   // this should get it to point to amount2
   Console.WriteLine("amount2 has address 0x{0:X} and contains {1}",
      (uint)pAmount, *pAmount);
```

在调用 Console.WriteLine()语句时，它显示了 amount2 的内容，但还没有对它进行初始化。显示出来的东西就是随机的垃圾—— 在执行该示例前存储在内存中该单元的内容。但这有一个要点：一般情况下，C#编译器会禁止使用未初始化的值，但在开始使用指针时，就很容易绕过许多通常的编译检查。此时我们这么做，是因为编译器无法知道我们实际上要显示

的是 amount2 的内容。因为知道了堆栈的工作方式，所以可以说出递减 pAmount 的结果是什么。使用指针算法，可以访问各种编译器通常禁止访问的变量和存储单元，因此指针算法是不安全的。

接下来在 pCents 指针上进行指针运算。pCents 目前指向 amount1.Cents，但此处的目的是使用指针算法让它指向 amount2.Cents，而不是直接告诉编译器我们要做什么。为此，需要从 pCents 指针所包含的地址中减去 sizeof(Currency)：

```
// do some clever casting to get pCents to point to cents
// inside amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Address of pCents is now 0x{0:X}", (uint)&pCents);
```

最后，使用 fixed 关键字创建一些指向类实例中字段的指针，使用这些指针设置这个实例的值。注意，这也是我们第一次查看存储在堆中(而不是堆栈)的项的地址：

```
Console.WriteLine("\nNow with classes");
// now try it out with classes
CurrencyClass amount3 = new CurrencyClass();

fixed(long* pDollars2 = &(amount3.Dollars))
fixed(byte* pCents2 = &(amount3.Cents))
{
   Console.WriteLine(
      "amount3.Dollars has address 0x{0:X}", (uint)pDollars2);
   Console.WriteLine(
      "amount3.Cents has address 0x{0:X}", (uint) pCents2);
   *pDollars2 = -100;
   Console.WriteLine("amount3 contains " + amount3);
}
```

编译并运行这段代码，得到如下所示的结果：

```
csc /unsafe PointerPlayaround2.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50215.33
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50215
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

PointerPlayaround2
Size of CurrencyStruct struct is 16
Address of amount1 is 0x12F4A4
Address of amount2 is 0x12F494
Address of pAmount is 0x12F490
Address of pDollars is 0x12F48C
Address of pCents is 0x12F488
amount1 contains $20.50
amount2 has address 0x12F494 and contains $0.0
Address of pCents is now 0x12F488
```

```
Now with classes
amount3.Dollars has address 0xA64414
amount3.Cents has address 0xA6441C
amount3 contains $-100.0
```

注意在这个结果中，显示了未初始化的 amount2 值，CurrencyStruct 结构的字节数是 16，大于其字段的字节数(1 long(=8) + 1 byte(=1) = 9 字节)。这是前面讨论的对齐单词的结果。

11.3.3　使用指针优化性能

前面用许多篇幅介绍了使用指针可以完成的各种任务，但在前面的示例中，仅是处理内存，让有兴趣的人们了解底层发生了什么事，并没有帮助人们编写出好的代码！本节将应用我们对指针的理解，用一个示例来说明使用指针可以大大提高性能。

1. 创建基于堆栈的数组

本节将介绍指针的一个主要应用领域：在堆栈中创建高性能、低系统开销的数组。第 2 章介绍了 C#如何支持数组的处理。C#很容易使用一维数组和矩形或锯齿形多维数组，但有一个缺点：这些数组实际上都是对象，是 System.Array 的实例。因此数组只能存储在堆上，会增加系统开销。有时，我们希望创建一个使用时间比较短的高性能数组，不希望有引用对象的系统开销。而使用指针就可以做到，但指针只能用于一维数组。

为了创建一个高性能的数组，需要使用另一个关键字：stackalloc。stackalloc 命令指示.NET 运行库分配堆栈上一定量的内存。在调用它时，需要为它提供两条信息：

- 要存储的数据类型
- 需要存储的数据项数。

例如，分配足够的内存，以存储 10 个 decimal 数据项，可以编写下面的代码：

```
decimal* pDecimals = stackalloc decimal [10];
```

注意，这个命令只是分配堆栈内存而已。它不会试图把内存初始化为任何默认值，这正好符合我们的目的。因为这是一个高性能的数组，给它不必要地初始化值会降低性能。

同样，要存储 20 个 double 数据项，可以编写下面的代码：

```
double* pDoubles = stackalloc double [20];
```

虽然这行代码指定把变量的个数存储为一个常数，但它是在运行时计算的一个数字。所以可以把上面的示例写为：

```
int size;
size = 20;   // or some other value calculated at run-time
double* pDoubles = stackalloc double [size];
```

从这些代码段中可以看出，stackalloc 的语法有点不寻常。它的后面紧跟要存储的数据类型名(该数据类型必须是一个值类型)，之后把需要的变量个数放在方括号中。分配的字

节数是变量个数乘以 sizeof(数据类型)。在这里，使用方括号表示这是一个数组。如果给 20 个 double 数据分配存储单元，就得到了一个有 20 个元素的 double 数组，最简单的数组类型是逐个存储元素的内存块，如图 11-6 所示。

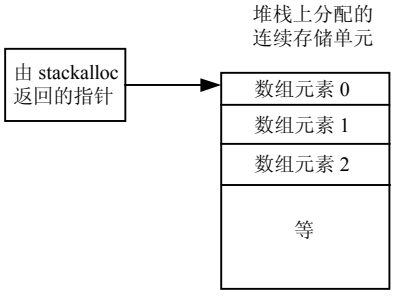

图 11-6

在图 11-6 中，显示了一个由 stackalloc 返回的指针，stackalloc 总是返回分配数据类型的指针，它指向新分配内存块的顶部。要使用这个内存块，可以取消对返回指针的引用。例如，给 20 个 double 数据分配内存后，把第一个元素(数组中的元素 0)设置为 3.0，可以编写下面的代码：

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
```

要访问数组的下一个元素，可以使用指针算法。如前所述，如果给一个指针加 1，它的值就会增加其数据类型的字节数。在本例中，就会把指针指向下一个空闲存储单元。因此可以把数组的第二个元素(数组中元素号为 1)设置为 8.4：

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

同样，可以用表达式*(pDoubles+X)获得数组中下标为 X 的元素。

这样，就得到一种访问数组中元素的方式，但对于一般目的，使用这种语法过于复杂。C#为此定义了另一种语法。对指针应用方括号时，C#为方括号提供了一种非常明确的含义。如果变量 p 是任意指针类型，X 是一个整数，表达式 p[X]就被编译器解释为*(p+X)，这适用于所有的指针，不仅仅是用 stackalloc 初始化的指针。利用这个简捷的记号，就可以用一种非常方便的方式访问数组。实际上，访问基于堆栈的一维数组所使用的语法与访问基于堆的、由 System.Array 类表示的数组是一样的：

```
double *pDoubles = stackalloc double [20];
pDoubles[0] = 3.0;   // pDoubles[0] is the same as *pDoubles
pDoubles[1] = 8.4;   // pDoubles[1] is the same as *(pDoubles+1)
```

注意：

把数组的语法应用于指针并不是新东西。自从开发出 C 和 C++语言以来，它们就是这两种语言的基础部分。实际上，C++开发人员会把这里用 stackalloc 获得的、基于堆栈的数组完全等同于传统的基于堆栈的 C 和 C++数组。这个语法和指针与数组的链接方式是 C 语

言在 70 年代后期流行起来的原因之一，也是指针的使用成为 C 和 C++中一种大众化编程技巧的主要原因。

高性能的数组可以用与一般 C#数组相同的方式访问，但需要注意：在 C#中，下面的代码会抛出一个异常：

```
double [] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

抛出异常的原因是：使用越界的下标来访问数组：下标是 50，而允许的最大下标是 19。但是，如果使用 stackalloc 声明了一个相同的数组，对数组进行边界检查时，这个数组中没有封装任何对象，因此下面的代码不会抛出异常：

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

在这段代码中，我们分配了足够的内存来存储 20 个 double 类型的数据。接着把 sizeof(double)存储单元的起始位置设置为该存储单元的起始位置加上 50*sizeof(double)存储单元，来保存双精度值 3.0。但这个存储单元超出了刚才为 double 分配的内存区域。谁也不知道这个地址存储了什么数据。最好是只使用某个当前未使用的内存，但所重写的空间也有可能在是堆栈上用于存储其他变量，或者是某个正在执行的方法的返回地址。因此，使用指针获得高性能的同时，也会付出一些代价：需要确保自己知道在做什么，否则就会抛出非常古怪的运行时错误。

2. 示例 QuickArray

下面用一个 stackalloc 示例 QuickArray 来结束关于指针的讨论。在这个示例中，程序仅要求用户提供为数组分配的元素数。然后代码使用 stackalloc 给 long 型数组分配一定的存储单元。这个数组的元素是从 0 开始的整数的平方，结果显示在控制台上：

```
using System;

namespace Wrox.ProCSharp.Memory
{
   class MainEntryPoint
   {
      static unsafe void Main()
      {
         Console.Write("How big an array do you want? \n> ");
         string userInput = Console.ReadLine();
         uint size = uint.Parse(userInput);

         long* pArray = stackalloc long [(int)size];
         for (int i=0 ; i<size ; i++)
            pArray[i] = i*i;

         for (int i=0 ; i<size ; i++)
            Console.WriteLine("Element {0} = {1}", i, *(pArray+i));
```

```
            }
        }
    }
}
```

运行这个示例，得到如下所示的结果：

```
QuickArray
How big an array do you want?
> 15
Element 0 = 0
Element 1 = 1
Element 2 = 4
Element 3 = 9
Element 4 = 16
Element 5 = 25
Element 6 = 36
Element 7 = 49
Element 8 = 64
Element 9 = 81
Element 10 = 100
Element 11 = 121
Element 12 = 144
Element 13 = 169
Element 14 = 196
```

11.4 小结

要想成为真正优秀的 C#程序员，必须牢固掌握存储单元和垃圾收集的工作原理。本章描述了 CLR 管理以及在堆和堆栈上分配内存的方式，讨论了如何编写正确释放未托管资源的类，并介绍如何在 C#中使用指针，这些都是很难理解的高级主题，初学者常常不能正确实现。

第12章 反　　射

反射是一个普通术语，描述了在运行过程中检查和处理程序元素的功能。例如，反射允许完成以下任务：

- 枚举类型的成员
- 实例化新对象
- 执行对象的成员
- 查找类型的信息
- 查找程序集的信息
- 检查应用于类型的定制特性
- 创建和编译新程序集

这个列表列出了许多功能，包括.NET Framework 类库提供的一些最强大、最复杂的功能。但本章不可能介绍反射的所有功能，仅讨论最常用的功能。

首先讨论定制特性，定制特性允许把定制的元数据与程序元素关联起来。这些元数据是在编译过程中创建的，并嵌入到程序集中。接着就可以在运行期间使用反射的一些功能检查这些元数据了。

在介绍了定制特性后，本章将探讨支持反射的一些基类，包括 System.Type 和 System.Reflection.Assembly 类，它们可以访问反射提供的许多功能。

为了演示定制特性和反射，我们将开发一个示例，说明公司如何定期升级软件，自动解释升级的信息。在这个示例中，要定义几个定制特性，表示程序元素最后修改或创建的日期，以及发生了什么变化。然后使用反射开发一个应用程序，在程序集中查找这些特性，自动显示软件自某个给定日期以来升级的所有信息。

本章要讨论的另一个示例是一个应用程序，该程序读写数据库，并使用定制特性，把类和特性标记为对应的数据库表和列。然后在运行期间从程序集中读取这些特性，使程序可以自动从数据库的相应位置检索或写入数据，无需为每个表或列编写特定的逻辑。

12.1 定制特性

前面介绍了如何在程序的各个数据项上定义特性。这些特性都是 Microsoft 定义好的，作为.NET Framework 类库的一部分，许多特性都得到了 C#编译器的支持。对于这些特性，编译器可以以特殊的方式定制编译过程，例如，可以根据 StructLayout 特性中的信息在内存中布置结构。

.NET Framework 也允许用户定义自己的特性。显然，这些特性不会影响编译过程，因为编译器不能识别它们，但这些特性在应用于程序元素时，可以在编译好的程序集中用作元数据。

这些元数据在文档说明中非常有用。但是，使定制特性非常强大的因素是使用反射，代码可以读取这些元数据，使用它们在运行期间作出决策，也就是说，定制特性可以直接影响代码运行的方式。例如，定制特性可以用于支持对定制许可类进行声明代码访问安全检查，把信息与程序元素关联起来，由测试工具使用，或者在开发可扩展的架构时，允许加载插件或模块。

12.1.1 编写定制特性

为了理解编写定制特性的方式，应了解一下在编译器遇到代码中某个应用了定制特性的元素时，该如何处理。以数据库为例，假定有一个 C#属性声明，如下所示。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
   get {
      // etc.
```

当 C#编译器发现这个属性有一个特性 FieldName 时，首先会把字符串 Attribute 添加到这个名称的后面，形成一个组合名称 FieldNameAttribute，然后在其搜索路径的所有命名空间(即在 using 语句中提及的命名空间)中搜索有指定名称的类。但要注意，如果用一个特性标记数据项，而该特性的名称以字符串 Attribute 结尾，编译器就不会把该字符串加到组合名称中，而是不修改该特性名。因此，上面的代码实际上等价于：

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
   get {
   // etc.
```

编译器会找到含有该名称的类，且这个类直接或间接派生自 System.Attribute。编译器还认为这个类包含控制特性用法的信息。特别是属性类需要指定：

- 特性可以应用到哪些程序元素上(类、结构、属性和方法等)
- 它是否可以多次应用到同一个程序元素上
- 特性在应用到类或接口上时，是否由派生类和接口继承

- 这个特性有哪些必选和可选参数

如果编译器找不到对应的特性类，或者找到一个这样的特性类，但使用特性的方式与特性类中的信息不匹配，编译器就会产生一个编译错误。例如，如果特性类指定该特性只能应用于字段，但我们把它应用到结构定义上，就会产生一个编译错误。

继续上面的示例，假定定义了一个 FieldName 特性：

```
[AttributeUsage(AttributeTargets.Property,
   AllowMultiple=false,
   Inherited=false)]
public class FieldNameAttribute : Attribute
{
   private string name;
   public FieldNameAttribute(string name)
   {
      this.name = name;
   }
}
```

下面几节讨论这个定义中的每个元素。

1. AttributeUsage 特性

要注意的第一个问题是特性(attribute)类本身用一个特性 System.AttributeUsage 来标记。这是 Microsoft 定义的一个特性，C#编译器为它提供了特殊的支持(AttributeUsage 根本不是一个特性，它更像一个元特性，因为它只能应用到其他特性上，不能应用到类上)。AttributeUsage 主要用于表示定制特性可以应用到哪些类型的程序元素上。这些信息由它的第一个参数给出，该参数是必选的，其类型是枚举类型 AttributeTargets。在上面的示例中，指定 FieldName 特性只能应用到属性(property)上——这是因为我们在前面的代码段中把它应用到属性上。AttributeTargets 枚举的成员如下：

- All
- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter(仅.NET 2.0 提供)
- Interface
- Method
- Module
- Parameter
- Property

- ReturnValue
- Struct

这个列表列出了可以应用该特性的所有程序元素。注意在把特性应用到程序元素上时，应把特性放在元素前面的方括号中。但是，在上面的列表中，有两个值不对应于任何程序元素：Assembly 和 Module。特性可以作为一个整体应用到程序集或模块中，而不是应用到代码中的一个元素上，在这种情况下，这个特性可以放在源代码的任何地方，但需要用关键字 assembly 或 module 来做前缀：

```
[assembly: SomeAssemblyAttribute(Parameters)]
[module: SomeAssemblyAttribute(Parameters)]
```

在指定定制特性的有效目标元素时，可以使用按位 OR 运算符把这些值组合起来。例如，如果指定 FieldName 特性可以应用到属性和字段上，可以编写下面的代码：

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
  AllowMultiple=false,
  Inherited=false)]
public class FieldNameAttribute : Attribute
```

也可以使用 AttributeTargets.All 指定特性可以应用到所有类型的程序元素上。AttributesUsage 特性还包含另外两个参数 AllowMultiple 和 Inherited。它们用不同的语法来指定：<AttributeName>=<AttributeValue>，而不是只给出这些参数的值。这些参数是可选的，如果需要，可以忽略它们。

AllowMultiple 参数表示一个特性是否可以多次应用到同一项上，这里把它设置为 false，表示如果编译器遇到下述代码，就会产生一个错误：

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{

  // etc.
```

如果 Inherited 参数设置为 true，就表示应用到类或接口上的特性也可以自动应用到所有派生的类或接口上。如果特性应用到方法或属性上，也可以自动应用到该方法或属性的重载上。

2. 指定特性参数

下面介绍如何指定定制特性的参数。在编译器遇到下述语句时：

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{

  // etc.
```

会检查传送给特性的参数(在本例中，是一个字符串)，并查找该特性中带这些参数的

构造函数。如果找到一个这样的构造函数，编译器就会把指定的元数据传送给程序集。如果找不到，就生成一个编译错误。如后面所述，反射会从程序集中读取元数据，并实例化它们表示的特性类。因此，编译器需要确保存在这样的构造函数，才能在运行期间实例化指定的特性。

在本例中，仅为 FieldNameAttribute 提供了一个构造函数，而这个构造函数有一个字符串参数。因此，在把 FieldNameAttribute 特性应用到一个属性上时，必须为它提供一个字符串参数，如上面的代码所示。

如果可以选择特性的参数类型，当然可以提供构造函数的不同重载方法，但一般是仅提供一个构造函数，使用属性来定义其他可选参数，下面将介绍可选参数。

3. 指定特性的可选参数

在 AttributeUsage 特性中，可以使用另一个语法，把可选参数添加到特性中。这个语法指定可选参数的名称和值，处理特性类中的公共属性或字段。例如，假定修改 SocialSecurityNumber 属性的定义，如下所示：

```
[FieldName("SocialSecurityNumber", Comment="This is the primary key field")]
public string SocialSecurityNumber
{

  // etc.
```

在本例中，编译器识别第二个参数的语法<ParameterName>=<ParameterValue>，所以不会把这个参数传递给 FieldNameAttribute 构造函数，而是查找一个有该名称的公用属性或字段(最好不要使用公用字段，所以一般情况下要使用属性)，编译器可以用这个属性设置第二个参数的值。如果希望上面的代码工作，必须给 FieldNameAttribute 添加一些代码：

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false,
  Inherited=false)]
public class FieldNameAttribute : Attribute
{
  private string comment;
  public string Comment
  {
    get
    {
      return comment;
    }
    set
    {
      comment = value;
    }
   }
    // etc.
```

12.1.2 定制特性示例：WhatsNewAttributes

本节开始编写前面描述过的示例 WhatsNewAttributes，该示例提供了一个特性，表示最后一次修改程序元素的时间。这个示例比前面所有的示例都复杂，因为它包含 3 个不同的程序集：

- WhatsNewAttributes 程序集，它包含特性的定义。
- VectorClass 程序集，包含所应用的特性的代码。
- LookUpWhatsNew 程序集，包含显示已改变的数据项信息的项目。

当然，只有 LookUpWhatsNew 是前面使用的一个控制台应用程序，其余两个程序集都是库文件，它们都包含类的定义，但都没有程序的入口。对于 VectorClass 程序集，我们使用了 VectorAsCollection 示例，但删除了入口和测试代码类，只剩下 Vector 类。

在命令行上编译，以此管理 3 个相关的程序集要求较高的技巧，所以我们分别给出编译这 3 个源文件的命令。也可以编辑代码示例，(可以从 Wrox Press 网站上下载)，组合为一个 Visual Studio 2005 解决方案，详见第 14 章。下载的文件包含所需的 Visual Studio 2005 解决方案文件。

1. WhatsNewAttributes 库程序集

首先从核心的 WhatsNewAttributes 程序集开始。其源代码包含在文件 WhatsNewAttributes.cs 中，该文件位于本章示例代码的 WhatsNewAttributes 解决方案的 WhatsNewAttributes 项目中。编译为库的语法非常简单：在命令行上，给编译器提供标记 target:library 即可。要编译 WhatsNewAttributes，键入：

```
csc /target:library WhatsNewAttributes.cs
```

WhatsNewAttributes.cs 文件定义了两个特性类 LastModifiedAttribute 和 SupportsWhatsNew-Attribute。LastModifiedAttribute 特性可以用于标记最后一次修改数据项的时间，它有两个必选参数(该参数传递给构造函数)；修改的日期和包含描述修改的字符串。它还有一个可选参数 Issues (表示存在一个公共属性)，它可以描述该数据项的任何重要问题。

在现实生活中，或许想把特性应用到任何对象上。为了使代码比较简单，这里仅允许将它应用于类和方法，并允许它多次应用到同一项上(AllowMultiple=true)，因为可以多次修改一个项，每次修改都需要用一个不同的特性实例来标记。

SupportsWhatsNew 是一个较小的类，表示不带任何参数的特性。这个特性是一个程序集的特性，用于把程序集标记为通过 LastModifiedAttribute 维护的文档说明书。这样，以后查看这个程序集的程序会知道，它读取的程序集是我们使用自动文档说明过程生成的那个程序集。这部分示例的完整源代码如下所示：

```
using System;
namespace Wrox.ProCSharp.WhatsNewAttributes
{
   [AttributeUsage(
      AttributeTargets.Class | AttributeTargets.Method,
      AllowMultiple=true, Inherited=false)]
```

```
    public class LastModifiedAttribute : Attribute
    {
       private DateTime dateModified;
       private string changes;
       private string issues;

       public LastModifiedAttribute(string dateModified, string changes)
       {
          this.dateModified = DateTime.Parse(dateModified);
          this.changes = changes;
       }

       public DateTime DateModified
       {
          get
          {
             return dateModified;
          }
       }

       public string Changes
       {
          get
          {
             return changes;
          }
       }

       public string Issues
       {
          get
          {
             return issues;
          }
          set
          {
             issues = value;
          }
       }
    }

    [AttributeUsage(AttributeTargets.Assembly)]
    public class SupportsWhatsNewAttribute : Attribute
    {
    }
}
```

从上面的描述可以看出，上面的代码非常简单。但要注意，不必将 set 访问器提供给 Changes 和 DateModified 属性，不需要这些访问器是因为在构造函数中，这些参数都是必选参数。需要 get 访问器，是因为以后可以读取这些特性的值。

2. VectorClass 程序集

本节就使用这些特性，我们用前面的 VectorAsCollection 示例的修订版本来说明。注意这里需要引用刚才创建的 WhatsNewAttributes 库，还需要使用 using 语句指定相应的命名空间，这样编译器才能识别出这些特性：

```
using System;
using System.Collections;
using System.Text;
using Wrox.ProCSharp.WhatsNewAttributes;

[assembly: SupportsWhatsNew]
```

在这段代码中，添加了一行用 SupportsWhatsNew 特性标记程序集本身的代码。

下面考虑 Vector 类的代码。我们并不是真的要修改这个类中的任何内容，只是添加两个 LastModified 特性，以标记出本章对 Vector 类进行的操作。把 Vector 定义为一个类，而不是结构，以简化后面显示特性所编写的代码(在 VectorAsCollection 示例中，Vector 是一个结构，但其枚举器是一个类。于是，这个示例的下一个版本在查看程序集时，必须同时考虑类和结构。这会使例子比较复杂)。

```
namespace Wrox.ProCSharp.VectorClass
{
   [LastModified("14 Feb 2007", "IEnumerable interface implemented " +
      "So Vector can now be treated as a collection")]
   [LastModified("10 Feb 2007", "IFormattable interface implemented " +
      "So Vector now responds to format specifiers N and VE")]
   class Vector : IFormattable, IEnumerable
   {
      public double x, y, z;

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

      [LastModified("10 Feb 2002",
                    "Method added in order to provide formatting support")]
      public string ToString(string format, IFormatProvider formatProvider)
      {
         if (format == null)
         {
            return ToString();
         }
```

再把包含的 VectorEnumerator 类标记为 new：

```
   [LastModified("14 Feb 2007",
                 "Class created as part of collection support for Vector")]
```

```
    private class VectorEnumerator : IEnumerator
    {
```

为了在命令行上编译这段代码，应键入下面的命令：

```
csc /target:library /reference:WhatsNewAttributes.dll VectorClass.cs
```

上面是这个示例的代码。目前还不能运行它，因为我们只有两个库。在描述了反射的工作原理后，就介绍这个示例的最后一部分，查找和显示这些特性。

12.2 反射

本节先介绍 System.Type 类，通过这个类可以访问任何给定数据类型的信息。然后简要介绍 System.Reflection.Assembly 类，它可以用于访问给定程序集的信息，或者把这个程序集加载到程序中。最后把本节的代码和上一节的代码结合起来，完成 WhatsNewAttributes 示例。

12.2.1 System.Type 类

在本书中的许多场合中都使用了 Type 类，但它只存储类型的引用：

```
Type t = typeof(double)
```

我们以前把 Type 看作一个类，但它实际上是一个抽象的基类。只要实例化了一个 Type 对象，就实例化了 Type 的一个派生类。Type 有与每种数据类型对应的派生类，但一般情况下派生的类只提供各种 Type 方法和属性的不同重载，返回对应数据类型的正确数据。一般不增加新的方法或属性。获取指向给定类型的 Type 引用有 3 种常用方式：

- 使用 C#的 typeof 运算符，如上所示。这个运算符的参数是类型的名称(不放在引号中)。
- 使用 GetType()方法，所有的类都会从 System.Object 继承这个类。

```
double d = 10;
Type t = d.GetType();
```

在一个变量上调用 GetType()，而不是把类型的名称作为其参数。但要注意，返回的 Type 对象仍只与该数据类型相关：它不包含与类型实例相关的任何信息。如果有一个对象引用，但不能确保该对象实际上是哪个类的实例，这个方法也是很有用的。

- 还可以调用 Type 类的静态方法 GetType()：

```
Type t = Type.GetType("System.Double");
```

Type 是许多反射技术的入口。它执行许多方法和属性，这里不可能列出所有的方法和属性，而主要介绍如何使用这个类。注意，可用的属性都是只读的：可以使用 Type 确定

数据的类型，但不能使用它修改该类型！

1. Type 的属性

由 Type 执行的属性可以分为下述 3 类：

- 有许多属性都可以获取包含与类相关的各种名称的字符串，如表 12-1 所示。

表 12-1

属 性	返 回 值
Name	数据类型名
FullName	数据类型的完全限定名(包括命名空间名)
Namespace	定义数据类型的命名空间名

- 属性还可以进一步获取 Type 对象的引用，这些引用表示相关的类，如表 12-2 所示。

表 12-2

属 性	返回对应的 Type 引用
BaseType	这个 Type 的直接基本类型
UnderlyingSystemType	这个 Type 在 .NET 运行库中映射的类型 (某些.NET 基类实际上映射由 IL 识别的特定预定义类型)

- 许多 Boolean 属性表示这个类型是一个类、还是一个枚举等。这些属性包括 IsAbstract、IsArray、IsClass、IsEnum、IsInterface、IsPointer、IsPrimitive(一种预定义的基本数据类型)、IsPublic、IsSealed 和 IsValueType

例如，使用一个基本数据类型：

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract);      // writes false
Console.WriteLine(intType.IsClass);         // writes false
Console.WriteLine(intType.IsEnum);          // writes false
Console.WriteLine(intType.IsPrimitive);     // writes true
Console.WriteLine(intType.IsValueType);     // writes true
```

或者使用 Vector 类：

```
Type intType = typeof(Vector);
Console.WriteLine(intType.IsAbstract);   // writes false
Console.WriteLine(intType.IsClass);      // writes true
Console.WriteLine(intType.IsEnum);       // writes false
Console.WriteLine(intType.IsPrimitive);  // writes false
Console.WriteLine(intType.IsValueType);  // writes false
```

也可以获取定义类型的程序集的引用，该引用作为 System.Reflection.Assembly 类实例的一个引用来返回：

```
Type t = typeof (Vector);
Assembly containingAssembly = new Assembly(t);
```

2. 方法

System.Type 的大多数方法都用于获取对应数据类型的成员信息：构造函数、属性、方法和事件等。它有许多方法，但它们都有相同的模式。例如，有两个方法可以获取数据类型的方法信息：GetMethod() 和 GetMethods()。GetMethod() 方法返回 System.Reflection.MethodInfo 对象的一个引用，其中包含一个方法的信息。GetMethods() 返回这种引用的一个数组。其区别是 GetMethods()返回所有方法的信息，而 GetMethod() 返回一个方法的信息，其中该方法包含特定的参数列表。这两个方法都有重载方法，该重载方法有一个附加的参数，即 BindingFlags 枚举值，表示应返回哪些成员，例如，返回公有成员、实例成员和静态成员等。

例如，GetMethods()最简单的一个重载方法不带参数，返回数据类型所有公共方法的信息：

```
Type t = typeof(double);
MethodInfo [] methods = t.GetMethods();
foreach (MethodInfo nextMethod in methods)
{
   // etc.
       }
```

Type 的成员方法如表 12-3 所示遵循同一个模式。

表 12-3

返回的对象类型	方法(名称为复数形式的方法返回一个数组)
ConstructorInfo	GetConstructor(), GetConstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()
PropertyInfo	GetProperty(), GetProperties()

GetMember()和 GetMembers()方法返回数据类型的一个或所有成员的信息，这些成员可以是构造函数、属性和方法等。最后要注意，可以调用这些成员，其方式是调用 Type 的 InvokeMember()方法，或者调用 MethodInfo, PropertyInfo 和其他类的 Invoke()方法。

12.2.2 TypeView 示例

下面用一个短小的示例 TypeView 来说明 Type 类的一些功能，这个示例可以列出数据类

型的所有成员。本例中主要介绍 double 型的 TypeView 用法，也可以修改该样列中的一行代码，使用其他的数据类型。TypeView 提供的信息要比在控制台窗口中显示的信息多得多，所以我们将打破常规，在一个消息框中显示这些信息。运行 double 型的 TypeView 示例，结果如图 12-1 所示。

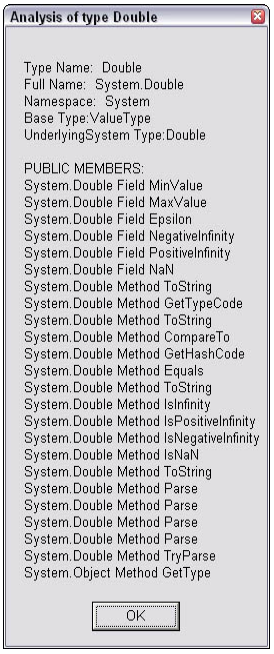

图 12-1

该消息框显示了数据类型的名称、全名和命名空间，以及底层类型和基类的名称。然后迭代该数据类型的所有公有实例成员，显示所声明类型的每个成员、成员的类型(方法、字段等)以及成员的名称。声明类型是实际声明类型成员的类名(换言之，如果在 System.Double 中定义或重载，该声明类型就是 System.Double，如果成员继承了某个基类，该声明类就是相关基类的名称)。

TypeView 不会显示方法的签名，因为我们是通过 MemberInfo 对象获取所有公有实例成员的信息，参数信息不能通过 MemberInfo 对象来获得。为了获取该信息，需要引用 MemberInfo 和其他更特殊的对象，即需要分别获取每一个成员类型的信息。

TypeView 会显示所有公有实例成员的信息，但对于 double 来说，仅定义了字段和方法。把 TypeView 编译为一个控制台应用程序，可以在控制台应用程序中显示消息框。但是，使用消息框就意味着需要引用基类程序集 System.Windows.Forms. dll，它包含 System.Windows.Forms 命名空间中的类，在这个命名空间中，定义了我们需要的 MessageBox 类。下面列出 TypeView 的代码。开始时需要添加两条 using 语句：

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Reflection;
```

需要 System.Text 的原因是我们要使用 StringBuilder 对象建立在消息框中显示的文本，以及消息框本身的 System.Windows.Forms。全部代码都放在类 MainClass 中，这个类包含两个静态方法和一个静态字段，StringBuilder 的一个实例叫作 OutputText，用于创建在消息框中显示的文本。Main 方法和类的声明如下所示：

```
class MainClass
{
   Static StringBuilder OutputText = new StringBuilder();

   static void Main()
   {
      // modify this line to retrieve details of any
      // other data type
      Type t = typeof(double);

      AnalyzeType(t);
      MessageBox.Show(OutputText.ToString(), "Analysis of type "
                                          + t.Name);
      Console.ReadLine();
   }
```

Main()方法首先声明一个 Type 对象，表示我们选择的数据类型，再调用方法 AnalyzeType()，从 Type 对象中提取信息，并使用该信息建立输出文本。最后在消息框中显示输出。使用 MessageBox 类是非常直观的：只需调用其静态方法 Show()，给它传递两个字符串，分别为消息框中的文本和标题。这些都由 AnalyzeType()来完成：

```
   static void AnalyzeType(Type t)
   {
      AddToOutput("Type Name:  " + t.Name);
      AddToOutput("Full Name:  " + t.FullName);
      AddToOutput("Namespace:  " + t.Namespace);

      Type tBase = t.BaseType;

      if (tBase != null)
      {
        AddToOutput("Base Type:" + tBase.Name);
       }

      Type tUnderlyingSystem = t.UnderlyingSystemType;

      if (tUnderlyingSystem != null)
       {
        AddToOutput("UnderlyingSystem Type:" + tUnderlyingSystem.Name);
       }

      AddToOutput("\nPUBLIC MEMBERS:");
      MemberInfo [] Members = t.GetMembers();

      foreach (MemberInfo NextMember in Members)
```

```
            {
               AddToOutput(NextMember.DeclaringType + " " +
               NextMember.MemberType + " " + NextMember.Name);
            }
         }
```

执行这个方法，仅需调用 Type 对象的各种属性，就可以获得我们需要的类型名称的信息，再调用 GetMembers()方法，获得一个 MemberInfo 对象数组，该数组用于显示每个成员的信息。注意这里使用了一个辅助方法 AddToOutput()，该方法创建要在消息框中显示的文本：

```
         static void AddToOutput(string Text)
         {
            OutputText.Append("\n" + Text);
         }
```

使用下面的命令编译 TypeView 程序集：

```
csc /reference:System.Windows.Forms.dll TypeView.cs
```

12.2.3 Assembly 类

Assembly 类是在 System.Reflection 命名空间中定义的，它允许访问给定程序集的元数据，它也包含可以加载和执行程序集(假定该程序集是可执行的)的方法。与 Type 类一样，Assembly 类包含非常多的方法和属性，这里不可能逐一论述。下面仅介绍完成示例 WhatsNewAttributes 所需要的方法和属性。

在使用 Assembly 实例做一些工作前，需要把相应的程序集加载到运行进程中。为此，可以使用静态成员 Assembly.Load()或 Assembly.LoadFrom()。这两个方法的区别是 Load()的参数是程序集的名称，运行库会在各个位置上搜索该程序集，这些位置包括本地目录和全局程序集高速缓存。而 LoadFrom()的参数是程序集的完整路径名，不会在其他位置搜索该程序集：

```
      Assembly assembly1 = Assembly.Load("SomeAssembly");
      Assembly assembly2 = Assembly.LoadFrom
         (@"C:\My Projects\Software\SomeOtherAssembly");
```

这两个方法都有许多其他重载，它们提供了其他安全信息。加载了一个程序集后，就可以使用它的各种属性，例如查找它的全名：

```
      string name = assembly1.FullName;
```

1. 查找在程序集中定义的类型

Assembly 类的一个特性是可以获得在相应程序集中定义的所有类型的信息，只要调用 Assembly.GetTypes()方法，就可以返回一个包含所有类型信息的 System.Type 引用数组，然后就可以按照上一节的方式处理这些 Type 引用了：

```
Type[] types = theAssembly.GetTypes();

foreach(Type definedType in types)
{
   DoSomethingWith(definedType);
}
```

2. 查找定制特性

用于查找在程序集或类型中定义了什么定制特性的方法取决于与该特性相关的对象类型。如果要确定程序集中有什么定制特性，就需要调用 Attribute 类的一个静态方法 GetCustomAttributes()，给它传递程序集的引用：

```
Attribute [] definedAttributes =
          Attribute.GetCustomAttributes(assembly1);
          // assembly1 is an Assembly object
```

注意：

这是相当重要的。以前您可能想知道，在定义定制特性时，必须为它们编写类，为什么 Microsoft 没有更简单的语法。答案就在于此。定制特性与对象一样，加载了程序集后，就可以读取这些特性对象，查看它们的属性，并且调用它们的方法。

GetCustomAttributes()在用于获取程序集的特性时，有两个重载方法：如果在调用它时，除了程序集的引用外，没有指定其他参数，该方法就会返回为这个程序集定义的所有定制特性。当然，也可以通过指定第二个参数来调用它，第二个参数表示特性类的一个 Type 对象，在这种情况下，GetCustomAttributes()就返回一个数组，该数组包含该特性类的所有特性。

注意，所有的特性都作为一般的 Attribute 引用来获取。如果要调用为定制特性定义的任何方法或属性，就需要把这些引用显式转换为相关的定制特性类。调用 Assembly.GetCustomAttributes()的另一个重载方法，可以获得与给定数据类型相关的定制特性信息，这次传递的是一个 Type 引用，它描述了要获取的任何相关特性的类型。另一方面，如果要获得与方法、构造函数和字段等相关的特性，就需要调用 GetCustomAttributes()方法，该方法是类 MethodInfo、 ConstructorInfo 和 FieldInfo 等的一个成员。

如果只需要给定类型的一个特性，就可以调用GetCustomAttribute()方法，它返回一个Attribute对象。在WhatsNewAttributes示例中使用GetCustomAttribute()方法，是为了确定程序集中是否有特性SupportsWhatsNew。为此，调用GetCustomAttributes()，传递对WhatsNew Attributes程序集的一个引用和SupportWhatsNewAttribute特性的类型。如果有这个特性，就返回一个Attribute实例。如果在程序集中没有定义任何实例，就返回null。如果找到两个或多个实例， GetCustomAttribute() 方法就抛出一个异常 System.Reflection.AmbiguousMatchException：

```
Attribute supportsAttribute =
       Attribute.GetCustomAttributes(assembly1,
       typeof(SupportsWhats NewAttribute));
```

12.2.4 完成 WhatsNewAttributes 示例

现在已经有足够的知识来完成 WhatsNewAttributes 示例了。为该示例中的最后一个程序集 LookUpWhatsNew 编写源代码，这部分应用程序是一个控制台应用程序，它需要引用其他两个程序集 WhatsNewAttributes 和 VectorClass。这是一个命令行应用程序，但仍可以象前面的 TypeView 示例那样在消息框中显示结果，因为结果是许多文本，所以不能显示在一个控制台窗口屏幕上。

这个文件的名称为LookUpWhatsNew.cs，编译它的命令是：

```
csc /reference:WhatsNewAttributes.dll /reference:VectorClass.dll
LookUpWhatsNew.cs
```

在这个文件的源代码中，首先指定要使用的命名空间 System.Text，因为需要使用一个 StringBuilder 对象：

```
using System;
using System.Reflection;
using System.Windows.Forms;
using System.Text;
using Wrox.ProCSharp.VectorClass;
using Wrox.ProCSharp.WhatsNewAttributes;

namespace Wrox.ProCSharp.LookUpWhatsNew
{
```

类 WhatsNewChecker 包含主程序入口和其他方法。我们定义的所有方法都在这个类中，它还有两个静态字段：outputText 和 backDateTo。outputText 包含在准备阶段创建的文本，这个文本要写到消息框中，backDateTo 存储了选择的日期——自从该日期以来的所有修改都要显示出来。一般情况下，需要显示一个对话框，让用户选择这个日期，但我们不想编写这段代码，以免转移读者的注意力。因此，把 backDateTo 硬编码为日期 2007 年 2 月 1 日。在下载这段代码时，很容易修改这个日期：

```
   class WhatsNewChecker
   {
      static StringBuilder outputText = new StringBuilder(1000);
      static DateTime backDateTo = new DateTime(2007, 2, 1);

      static void Main()
      {
         Assembly theAssembly = Assembly.Load("VectorClass");
         Attribute supportsAttribute =
            Attribute.GetCustomAttribute(
               theAssembly, typeof(SupportsWhatsNewAttribute));
         string Name = theAssembly.FullName;

         AddToMessage("Assembly: " + Name);
         if (supportsAttribute == null)
         {
```

```
            AddToMessage(  "This assembly does not support WhatsNew attributes");
            return;
         }
         else
            AddToMessage("Defined Types:");

         Type[] types = theAssembly.GetTypes();
         foreach(Type definedType in types)
            DisplayTypeInfo(theAssembly, definedType);

         MessageBox.Show(outputText.ToString(),
            "What\'s New since " + backDateTo.ToLongDateString());
         Console.ReadLine();
      }
```

Main()方法首先加载 VectorClass 程序集，验证它是否真的用 SupportsWhatsNew 特性来标记。我们知道，VectorClass 应用了 SupportsWhatsNew 特性，虽然才编译了该程序集，但进行这种检查还是必要的，因为用户可能希望检查这个程序集。

验证了这个程序集后，使用 Assembly.GetTypes()方法获得一个数组，其中包括在该程序集中定义的所有类型，然后在这个数组中迭代。对每种类型调用一个方法 DisplayTypeInfo()，给 outputText 字段添加相关的文本，包括 LastModifiedAttribute 实例的信息。最后，显示带有完整文本的消息框。DisplayTypeInfo()方法如下所示：

```
static void DisplayTypeInfo(Assembly theAssembly, Type type)
{
    // make sure we only pick out classes
    if (!(type.IsClass))
    {
        return;
    }

    AddToMessage("\nclass " + type.Name);

    Attribute [] attribs = Attribute.GetCustomAttributes(type);
    if (attribs.Length == 0)
    {
        AddToMessage("No changes to this class\n");
    }
    else
    {
        foreach (Attribute attrib in attribs)
        {
            WriteAttributeInfo(attrib);
        }
```

```
            }

            MethodInfo [] methods = type.GetMethods();
            AddToMessage("CHANGES TO METHODS OF THIS CLASS:");

            foreach (MethodInfo nextMethod in methods)
            {
                object [] attribs2 =
                    nextMethod.GetCustomAttributes(
                        typeof(LastModifiedAttribute), false);
                if (attribs != null)
                {
                    AddToMessage(
                        nextMethod.ReturnType + " " + nextMethod.Name + "()");
                    foreach (Attribute nextAttrib in attribs2)
                    {
                        WriteAttributeInfo(nextAttrib);
                    }
                }
            }
        }
```

注意，在这个方法中，首先应检查参数 Type 引用是否表示一个类。为了简化代码，指定 LastModified 特性只能应用于类或成员方法，如果该引用不是类(它可能是一个结构、委托或枚举)，进行任何处理都是浪费时间。

接着使用 Attribute.GetCustomAttributes()方法确定这个类是否有相关的 LastModifiedAttribute 实例。如果有，就使用帮助方法 WriteAttributeInfo()把它们的信息添加到输出文本中。

最后使用 Type.GetMethods()方法迭代这个数据类型的所有成员方法，然后对类的每个方法进行相同的处理：检查每个方法是否有相关的 LastModifiedAttribute 实例，如果有，用 WriteAttributeInfo()显示方法它们。

下面的代码显示了 WriteAttributeInfo()方法，它负责确定为给定的 LastModifiedAttribute 实例显示什么文本，注意这个方法的参数是一个 Attribute 引用，所以需要先把该引用转换为 LastModifiedAttribute 引用。之后，就可以使用最初为这个特性定义的属性获取其参数。在把该特性添加到要显示的文本中之前，应检查特性的日期是否是最近的：

```
        static void WriteAttributeInfo(Attribute attrib)
        {

           LastModifiedAttribute lastModifiedAttrib =
              attrib as LastModifiedAttribute;
           if (lastModifiedAttrib == null)
           {
              return;
```

```
         }

         // check that date is in range
         DateTime modifiedDate = lastModifiedAttrib.DateModified;

         if (modifiedDate < backDateTo)
         {
            return;
         }

         AddToMessage("  MODIFIED:  " +
            modifiedDate.ToLongDateString() + ":");
         AddToMessage("    " + lastModifiedAttrib.Changes);

         if (lastModifiedAttrib.Issues != null)
         {
            AddToMessage("   Outstanding issues:" +
               lastModifiedAttrib.Issues);
          }
      }
```

最后，是辅助方法 AddToMessage()：

```
      static void AddToMessage(string message)
      {
         outputText.Append("\n" + message);
      }
   }
}
```

运行这段代码，得到如图 12-2 所示的结果。

图 12-2

注意，在列出 VectorClass 程序集中定义的类型时，实际上选择了两个类：Vector 和内嵌的 VectorEnumerator 类。还要注意，这段代码把 backDateTo 日期硬编码为 2 月 1 日，实际上选择的是日期为 2 月 14 日的特性(添加集合支持的时间)，而不是 1 月 14 日(添加 IFormattable 接口的时间)。

12.3 小结

本章没有介绍反射的全部内容，反射需要一整本书来讨论。我们只介绍了 Type 和 Assembly 类，它们是访问反射所提供的扩展功能的主要入口。

另外，本章还探讨了反射的一个常用方面：定制特性。介绍了如何定义和应用自己的定制特性，以及如何在运行期间检索定制属性的信息。

第 13 章介绍异常和结构化的异常处理。