

PROGRAMMER TO PROGRAMMER™



Professional C# 2008

# C# 高级编程

（第 2 版）

Wrox畅销书！C#经典名著！

2005年最权威的十大IT图书！

2005年度引进版科技类优秀图书！

2006年最受读者喜爱的十大技术开发类图书！

2007年最畅销的C#零售图书！

② C# 2008最新版

# 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：

- (1) C# 的结构和方法论反映了.NET 基础方法论。
- (2) 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。下面是本章的内容：

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)，.NET 上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。

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：平台无关性、提高性能和语言的互操作性。

### 1.2.1 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/>)。

### 1.2.2 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。

### 1.2.3 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 2008

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 2008 代码(或 Visual Basic .NET 2002、2003 和 2005 代码)，把 Visual Basic 6 程序转换为 Visual Basic 2008 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 2008(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 2008 中，Visual Studio 2008 就会升级该项目，也就是说把 Visual Basic 6 源代码重写为 Visual Basic 2008 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 2008 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 2008 编译为内部可执行代码了。Visual Basic 2008 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++ 2008

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 :

```
_ref class MyClass  
{
```

生成了一个托管类 , 就好像使用 C# 或 Visual Basic 2008 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。

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 2008。

## 3. 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 集成了 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 24 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

如前所述 , 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 B，获得更多的信息。

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，而无须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 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 2008，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 2008 类中的一个方法定义为返回一个 Integer--Visual Basic 2008 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 2008 的 Integer 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 2008 的 Integer 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 2008 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。

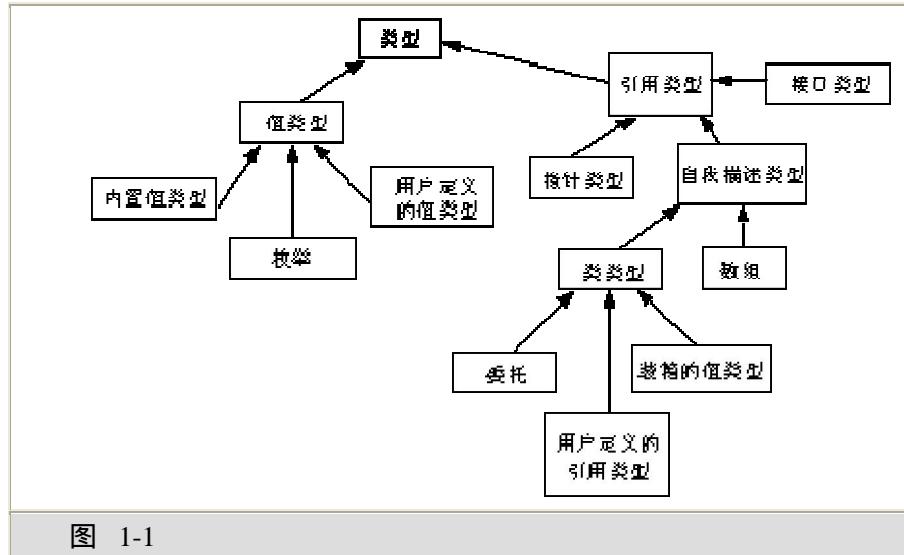


图 1-1

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-defined Reference 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 2008 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 2008 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 2008 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，就要保证用其他兼容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，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临时释放所有的内存)。

.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，换而言之，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。

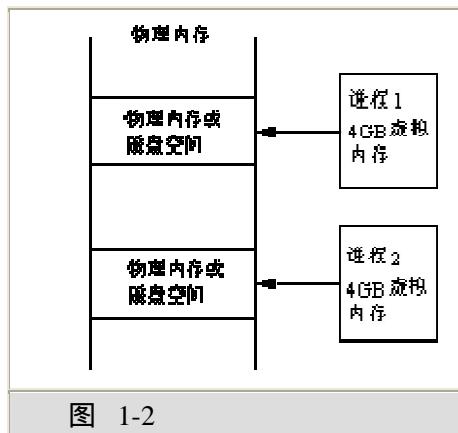
安全问题详见本书后面的第 20 章。

### 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。



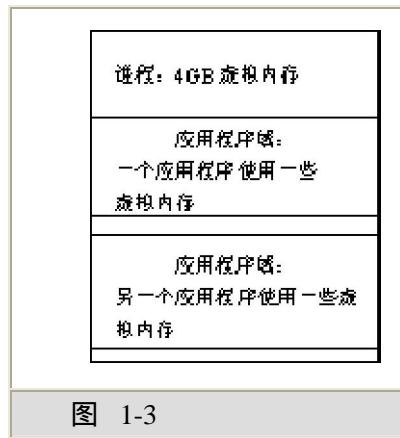
在一般情况下，任何进程都只能通过指定虚拟内存中的一个地址来访问内存--即进程不能直接访问物理内存，因此一个进程不可能访问分配给另一个进程的内存。这样就可以确保任何执行出错的代码不会损害其地址空间以外的数据(注意在 Windows 95/98 上，这些保护措施不像在 Windows NT/2000/XP/2003/Vista 上那样强大，所以理论上存在应用程序因写入不对称的内存而导致 Windows 崩溃的可能性)。

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/Vista 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中运行。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。

如果不同的可执行文件都运行在同一个进程中，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 14 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 2008 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 14 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异常处理代码处理被抛出的异常对象，无论异常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(在 Microsoft 的 COM 接口定义语言(Interface Definition Language，IDL)中使用特性)。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 2008 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 13 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 17 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 17 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好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 13 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，这仅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 2008 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 15 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 Framework 3.5 的.NET 基类库中的各种类，即各种基类是如何工作的。.NET 3.5 基类包括：

- IL 提提供的核心功能，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 31 和 34 章)
- Web 窗体(ASP.NET，第 37 和 38 章)
- 数据访问(ADO.NET，第 26~30 章)
- 目录访问(第 46 章)
- 文件系统和注册表访问(第 25 章)
- 网络和 Web 浏览(第 41 章)
- .NET 特性和反射(第 13 章)
- 访问 Windows 操作系统的各个方面(例如环境变量等，第 20 章)
- COM 互操作性(第 44 和 24 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类型检查。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 37、38 和 39 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在 Visual Studio 2008 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 2008 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的机器上。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 2008 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，就是创建一个继承自 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 2008 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。

### 3. 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 2008 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。

### 4. 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 2008 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 2008 和 C# 都支持 Windows Form 项目。第 31 章将详细介绍 Windows 窗体。

#### 1.7.3 使用 Windows Presentation Foundation(WPF)

有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 和 3.5 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 或 3.5。WPF 应用程序可用于 Windows Vista、Windows XP、Windows Server 2003 和 Windows Server 2008 (只有这些操作系统能安装.NET Framework 3.0 或 3.5)。

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 34 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/Vista (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 2008 允许创建 C# Windows Service 项目 ,为基本 Windows 服务编写 C# 源代码。第 23 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 和 3.5 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 42 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 对象模型或 LINQ 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 Visual Studio 2008 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。

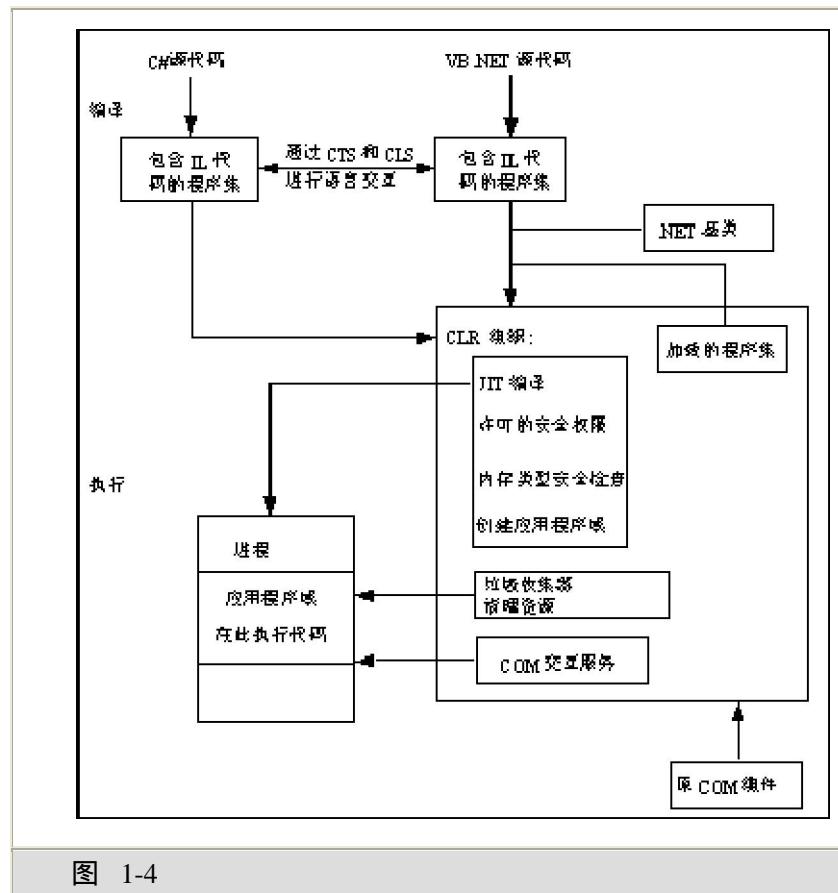


图 1-4

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。还讨论了 CLS 和 CTS，来帮助处理语言互操作性。

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

使用注释和文档编制功能

预编译器指令

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。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，就应该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，在继续之前先熟悉 OOP 基础是很有帮助的。

如果读者对 Visual Basic 6、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 Basic 6 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](http://www.wrox.com)) 上下载不同的文档来学习 C#。

### 2.2 第一个 C# 程序

下面编译并运行最简单的 C# 程序，这是一个简单的类，包含把信息写到屏幕上的控制台应用程序。

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 2008 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 2008。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。

#### 2.2.1 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)。Main() 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;  
}  
}  
}
```

## 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 2008)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 2008%\Common7\Tools\vcvars32.bat。其中%Microsoft Visual Studio 2008 是安装 Visual Studio 2008 的文件夹。第二种方法(更简单)是使用 Visual Studio 2008 命令行代替通常的命令提示窗口。Visual Studio 2008 命令提示在"开始"菜单 |"程序" | Microsoft Visual Studio 2008 | 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# Compiler version 9.00.20404  
for Microsoft (R) .NET Framework version 3.5  
Copyright (C) Microsoft Corporation. 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)。下面通过这个程序详细介绍 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 Basic 6 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 Basic 6 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。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。

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。

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 Basic 6 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()  
{\system32}
```

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 Basic 6 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 2008 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，所以也退出了应用程序)。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。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 Basic 6，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，这些文本仅为了让人们更好地理解程序，它们并不是程序的一部分。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，创建这些变量时，其值就默认是 0(类和结构在后面讨论)。

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'
```

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 类型推断

类型推断使用 var 关键字。声明变量的语法有些变化。编译器可以根据变量的初始化值“推断”变量的类型。例如：

```
int someNumber = 0;
```

就变成：

```
var someNumber = 0;
```

即使 someNumber 从来没有声明为 int，编译器也可以确定，只要 someNumber 在其作用域内，就是一个 int。编译后，上面两个语句是等价的。

下面是另一个小例子：

```
using System;
namespace Wrox.ProCSharp.Basics
{
    class Program
    {
        static void Main(string[] args)
        {
            var name = "Bugs Bunny";
            var age = 25;
            var isRabbit = true;

            Type nameType = name.GetType();
            Type ageType = age.GetType();
            Type isRabbitType = isRabbit.GetType();

            Console.WriteLine("name is type " + nameType.ToString());
            Console.WriteLine("age is type " + ageType.ToString());
        }
    }
}
```

```
Console.WriteLine("isRabbit is type " + isRabbitType.ToString());
}
}
}
```

这个程序的输出如下：

```
name is type System.String
age is type System.Int32
isRabbit is type System.Bool
```

需要遵循一些规则。变量必须初始化。否则，编译器就没有推断变量类型的依据。初始化器不能为空，且必须放在表达式中。不能把初始化器设置为一个对象，除非在初始化器中创建了一个新对象。第3章在讨论匿名类型时将详细探讨。

声明了变量，推断出了类型后，变量的类型就不能改变了。这与Visual Basic中使用的Variant类型不同。变量的类型建立后，就遵循其他变量类型遵循的强类型化规则。

### 2.3.3 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 2 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;
        Console.WriteLine(j);

        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字：

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

#### 2.3.4 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。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：

```
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。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 (-27~27-1)
short	System.Int16	16位有符号的整数	-32 768~32 767 (-215~215-1)
int	System.Int32	32位有符号的整数	-2 147 483 648~2 147 483 647(-231~231-1)
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(-263~263-1)
byte	System.Byte	8位无符号的整数	0~255(0~28-1)
ushort	System.UInt16	16位无符号的整数	0~65535(0~216-1)
uint	System.UInt32	32位无符号的整数	0~4 294 967 295(0~232-1)
ulong	System.UInt64	64位无符号的整数	0~18 446 744 073 709 551 615(0~264-1)

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	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$
double	System.Double	64 位双精度浮点数	15/16	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$

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	$\pm 1.0 \times 10^{-28} \sim \pm 7.9 \times 10^{28}$

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	说明	位数	值
bool	System.Boolean	表示 true 或 false	NA	true 或 false

bool 值和整数值不能相互隐式转换。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。这与 C++相同。

## 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 类型的数据组。

#### 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 6.0 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。object 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，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 。

也可以在上面的例子中给循环变量 i 使用类型推断功能。使用类型推断功能时 ,循环结构变成 :

```
for (var i = 0; i < 100; i++)  
...
```

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 和 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 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. 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。

这里也可以使用类型推断功能。此时，foreach 循环变成：

```
foreach (var temp in arrayOfInts)
...
```

temp 的类型推断为 int，因为这是集合项的类型。

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 14 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，Visual Studio 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;
        }
    }
}
```

```
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。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。全名常常称为完全限定的名称。

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型，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 NamespaceExample 类的一个对象，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;
    }
}
```

注意：

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，编译器会在类中查找与上述签名匹配的 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 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. 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.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
#ifndef 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#ifndef 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#ifndef 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#ifndef 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#ifndef 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#ifndef 块中，其含义非常直观。也可以嵌套#ifndef 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#ifndef 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 13 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 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 15 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编码。下面是有效标识符的一些例子：

Name

überflu?

\_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：

变量名用带有前缀字母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	Do	Loc	RGB
Add	Double	Local	Right
AddHandler	Each	Lock	RmDir
AddressOf	Else	LOF	Rnd
Alias	ElseIf	Log	RTrim
And	Empty	Long	SaveSettings
Ansi	End	Loop	Second
AppActivate	Enum	LTrim	Seek
Append	EOF	Me	Select
As	Erase	Mid	SetAttr
Asc	Err	Minute	SetException
Assembly	Error	MIRR	Shared
Atan	Event	MkDir	Shell
Auto	Exit	Module	Short
Beep	Exp	Month	Sign
Binary	Explicit	MustInherit	Sin
BitAnd	ExternalSource	MustOverride	Single
BitNot	False	MyBase	SLN
BitOr	FileAttr	MyClass	Space
BitXor	FileCopy	Namespace	Spc
Boolean	FileDateTime	New	Split
ByRef	FileLen	Next	Sqrt
Byte	Filter	Not	Static
ByVal	Finally	Nothing	Step
Call	Fix	NotInheritable	Stop
Case	For	NotOverridable	Str
Catch	Format	Now	StrComp
CBool	FreeFile	NPer	StrConv
CByte	Friend	NPV	Strict
CDate	Function	Null	String
CDbl	FV	Object	Structure
CDec	Get	Oct	Sub
ChDir	GetAllSettings	Off	Switch
ChDrive	GetAttr	On	SYD
Choose	GetException	Open	SyncLock
Chr	GetObject	Option	Tab
CInt	GetSetting	Optional	Tan
Class	GetType	Or	Text
Clear	GoTo	Overloads	Then
CLng	Handles	Overridable	Throw
Close	Hex	Overrides	TimeOfDay
Collection	Hour	ParamArray	Timer
Command	If	Pmt	TimeSerial
Compare	Iif	PPmt	TimeValue
Const	Implements	Preserve	To

Cos	Imports	Print	Today
CreateObject	In	Private	Trim
CShort	Inherits	Property	Try
CSng	Input	Public	TypeName
CStr	InStr	Put	TypeOf
CurDir	Int	PV	UBound
Date	Integer	QBColor	UCase
DateAdd	Interface	Raise	Unicode
DateDiff	Ipmt	RaiseEvent	Unlock
DatePart	IRR	Randomize	Until
DateSerial	Is	Rate	Val
DateValue	IsArray	Read	Weekday
Day	IsDate	ReadOnly	While
DDB	IsDBNull	ReDim	Width
Decimal	IsNumeric	Remove	With
Declare	Item	RemoveHandler	WithEvents
Default	Kill	Rename	Write
Delegate	Lcase	Replace	WriteOnly
DeleteSetting	Left	Reset	Xor
Dim	Lib	Resume	Year
Dir	Line	Return	

## 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，我们自己的类与基类的风格相同，有助于提高可读性和可维护性。

## 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（甚至 JavaScript）的开发人员能立即领悟的。

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、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

类成员

按值和引用传递参数

方法重载

构造函数和静态构造函数

只读字段

部分类

静态类

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 12 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 6 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 6 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。

#### (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# Compiler version 9.00.20404
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. 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递给方法的，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。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# Compiler version 9.00.20404  
for Microsoft (R) .NET Framework version 3.5  
Copyright (C) Microsoft Corporation. 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

通过值传递变量是默认的，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。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(方法名相同、但参数的个数和类型不同)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。

Visual Basic 6 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) 自动实现的属性

如果属性的 set 和 get 访问器中没有任何逻辑，就可以使用自动实现的属性。这种属性会自动实现基础成员变量。上例的代码如下：

```
public string ForeName { get; set; }
```

不需要声明 private string foreName。编译器会自动创建它。

使用自动实现的属性，就不能在属性设置中进行属性的有效性验证。所以在上面的例子中，不能检查 foreName 是否少于 20 个字符。但必须有两个访问器。尝试把该属性设置为只读属性，就会出错：

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

但是，每个访问器的访问级别可以不同。因此，下面的代码是合法的：

```
public string ForeName { get; private set; }
```

#### (4) 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，因为定义了一个带单个参数的构造函数，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 33 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]
```

当然，如果在周末执行代码，颜色设置就是 Green。

(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 匿名类型

第 2 章讨论了 var 关键字，用于表示隐式类型化的变量。var 与 new 关键字一起使用时，可以创建匿名类型。匿名类型只是一个继承了 Object 的、没有名称的类。该类的定义从初始化器中推断，类似于隐式类型化的变量。

如果需要一个对象包含某个人的姓氏、中间名和名字，则声明如下：

```
var captain = new {FirstName = "James", MiddleName = "T",
LastName = "Kirk"};
```

这会生成一个包含 FirstName、MiddleName 和 LastName 属性的对象。如果创建另一个对象，如下所示：

```
var doctor = new {FirstName = "Leonard", MiddleName = "",
LastName = "McCoy"};
```

Captain 和 doctor 的类型就是相同的。例如，可以设置 captain = doctor。

如果所设置的值来自于另一个对象，初始化器就可以简化。如果已经有一个包含 FirstName、MiddleName 和 LastName 属性的类，且有一个该类的实例 person，captain 对象就可以初始化为：

```
var captain = new (person.FirstName, person.MidleName,
person.LastName);
```

person 对象的属性名应投射为新对象名 captain。所以 captain 对象应有 FirstName、MiddleName 和 LastName 属性。

这些新对象的类型名是未知的。编译器为类型“伪造”了一个名称，但只有编译器才能使用它。我们不能也不应使用新对象上的任何类型引用，因为这不会得到一致的结果。

### 3.4 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，让人们试着在计算机上重新安排设备，并存储每个设备项的维数。使字段变为公共字段，就会违背编程规则，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。

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，如本章前面所述：

```
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 13 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。

### 3.4.1 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。

### 3.4.2 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。

### 3.4.3 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#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。

## 3.5 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 class、struct 或 interface 关键字的前面。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
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。

## 3.6 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();
```

## 3.7 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。

### 3.7.1 System.Object 方法

下面将简要总结每个方法的作用，下一节详细论述 ToString()方法。

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 12 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 13 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。

### 3.7.2 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 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
```

## 3.8 扩展方法

有许多方法扩展类。如果有类的源代码，继承（如第 4 章所述）就是给对象添加功能的好方法。但如果没有源代码，该怎么办？此时可以使用扩展方法，它允许改变一个类，但不需要类的源代码。

扩展方法是静态方法，是类的一部分，但实际上没有放在类的源代码中。假定上例中的 Money 类需要一个方法 AddToAmount ( decimal amountToAdd )。但是，由于某种原因，程序集最初的源代码不能直接修改。此时就可以创建一个静态类，把方法 AddToAmount 添加为一个静态方法。代码如下：

```
namespace Chapter3.Extensions
{
    public static class MoneyExtension
    {
        public static void AddToAmount(this Money money, decimal
            amountToAdd)
        {
            money.Amount += amountToAdd;
        }
    }
}
```

注意 AddToAmount 方法的参数。对于扩展方法，第一个参数是要扩展的类型，它放在 this 关键字的后面。这告诉编译器，这个方法是 Money 类型的一部分。在这个例子中，Money 是要扩展的类型。在扩展方法中，可以访问所扩展类型的所有公共方法和属性。

在主程序中，AddToAmount 方法看起来像是另一个方法。它没有显示第一个参数，也不能对它进行任何处理。要使用新方法，需要执行如下调用，这与其他方法相同：

```
cash1.AddToAmount(10M);
```

即使扩展方法是静态的，也要使用标准的实例方法语法。注意这里使用 cash1 实例变量来调用 AddToAmount，而没有使用类型名。

如果扩展方法与类中的某个方法同名，扩展方法就从来不会被调用。类中已有的实例方法优先。

## 3.9 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，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()。本章多次提到了继承，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。我们将讨论 C# 和 .NET Framework 如何处理继承。本章的主要内容如下：

- 继承的类型
- 实现继承
- 访问修饰符
- 接口

### 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 31 章讨论的 Windows Forms 类。第 31 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 31 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 12 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 Visual Studio)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#不能采用=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
{
```

要在方法或属性上使用 sealed 关键字，必须先在基类上把它声明为重写。如果基类上不希望有重写的方法或属性，就不要把它声明为 virtual。

### 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，并把控制权返回给 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了，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 GenericCustomer 提供一个带有参数的构造函数，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
<code>public</code>	所有的类型或成员	任何代码均可以访问该方法
<code>protected</code>	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
<code>internal</code>	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
<code>private</code>	所有的类型或成员	只能在它所属的类型中访问该方法
<code>protected internal</code>	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
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，方法和属性	密封类不能继承。对于属性和方法，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 的不兼容，因为 VB 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。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)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 ITransferBank Account 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

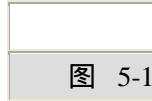
声明了数组后，就必须为数组分配内存，以保存数组的所有元素。数组是引用类型，所以必须给它分配堆上的内存。为此，应使用 new 运算符，指定数组中元素的类型和数量来初始化数组的变量。下面指定了数组的大小。

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。



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：

```
foreach (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、自动实现的属性 `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;  
    }  
    public string Firstname{ get; set; }  
    public string Lastname{ get; set; }  
    public override string ToString()  
    {  
    }
```

```
{  
    return String.Format("{0} {1}", 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 14 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。这个数组有足够的空间容纳两个引用。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。



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。



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。

在花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  $3 \times 3$  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  $3 \times 3$  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", "Fittipaldi"), 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

因为数组是引用类型，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();
```

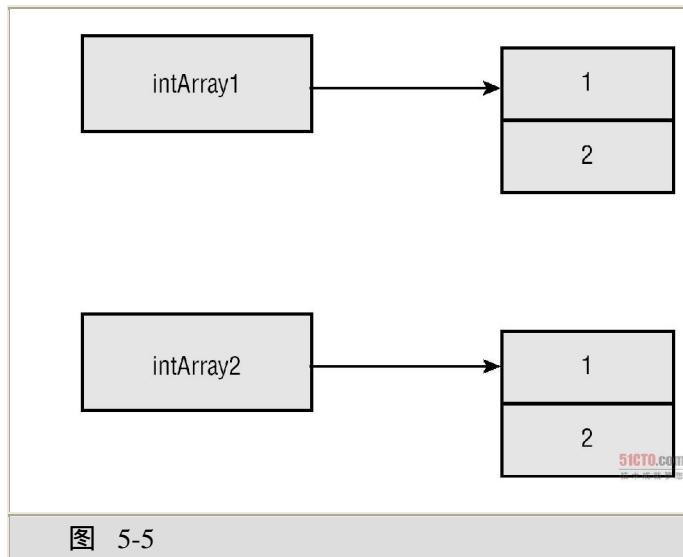


图 5-5

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();
```

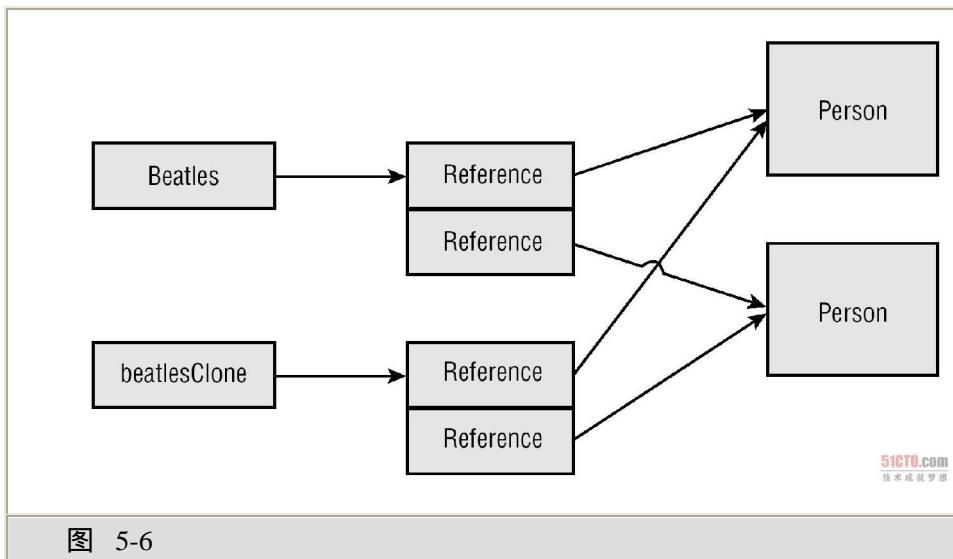


图 5-6

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

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。如果 LastName 的值相同，就比较 FirstName：

```
public class Person : IComparable  
{
```

```
public int CompareTo(object obj)
{
    Person other = obj as Person;
    int result = this.LastName.CompareTo(
        other.LastName);
    if (result == 0)
    {
        result = this.FirstName.CompareTo(
            other.FirstName);
    }
    return result;
}
//...
```

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.Sort (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 firstName 或 lastName 对 Person 对象排序。枚举 PersonCompareType 定义了与 PersonComparer 相当的排序选项：FirstName 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 firstName 还是 lastName 排序。

```
public class PersonComparer : IComparer
{
    public enum PersonCompareType
    {
        FirstName,
        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。第 19 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 Getenumerator()方法的 IEumerable 接口。Getenumerator()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。

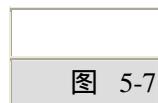


图 5-7

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 IEnumerable 接口的对象就足够了。

### 5.6.1 IEnumator 接口

foreach 语句使用 IEnumator 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

IEnumator 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在集合的开头。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 IEnumable 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.Collections;
namespace Wrox.ProCSharp.Arrays
{
    public class HelloCollection
    {
        public IEumerator GetEnumerator()
        {
            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;
                default:
                    return false;
            }
        }

        void IDisposable.Dispose()
        {
        }
    }
}
```

```
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，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 class 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 yield 语句的优点。

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 Visual Basic 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。

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	== != <> <= >=
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	??

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 12 章的讨论。还要注意，sizeof 运算符在.NET Framework 1.0 和 1.1 中使用，它需要不安全模式。自从.NET Framework 2.0 以来，就没有这个运算符了。

类 别	运 算 符
运算符关键字	sizeof(仅用于.NET Framework 1.0 和 1.1)
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 Visual Basic 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。Visual Basic 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 12 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) // true - x is incremented to 6 before the evaluation
{
    Console.WriteLine("This will execute");
}
if (x++ == 7) // false - x is incremented to 7 after the evaluation
{
    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;
```

下面介绍在 C# 代码中频繁使用的基本运算符和类型转换运算符。

### 6.1.2 条件运算符

条件运算符(?)也称为三元运算符，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 + " ";
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。下面修改代码 ,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。"兼容"表示对象是该类型 ,或者派生于该类型。例如 ,要检查变量是否与 object 类型兼容 ,可以使用下面的代码 :

```
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; //s2 = 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 12 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 13 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，不能随意合并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	
布尔 AND 运算符	&&
布尔 OR 运算符	
条件运算符	?:
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#中的一个特性，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 型变量包含的数据字节比 byte 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 14 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 编码为 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 14 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。还可以进行显式转换：

```
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值类型。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 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，下面的代码表示比较引用：

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

但是，如果把一些类看作值，其含义就会比较直观。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()的默认重写代码肯定足以应付绝大多数自定义的结构，但仍可以对自己的结构重写它，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 Visual Basic 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。所以编译器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版本一样。

#### 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(5.0, -1.0, -3.0)。此时，数学表达式总是写成 c=a+b，其中 a 和 b 是矢量，c 是结果矢量。这与使用 Vector 结构的方式是一样的。

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。

注意：

一般把运算符左边的参数命名为 lhs，运算符右边的参数命名为 rhs。

C#要求所有的运算符重载都声明为 public 和 static，这表示它们与它们的类或结构相关联，而不是与实例相关联，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  $\text{double } X = a*b$ ，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元素的值，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)？

浅度是比较对象是否指向内存中的同一个位置，而深度是比较对象的值和属性是否相等。应根据具体情况进相等检查，确定要进行什么比较。

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));
}
```

```
=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# 2008 Compiler version 3.05.20706.1  
for Microsoft (R) 2005 Framework version 3.5  
Copyright (C) Microsoft Corporation. All rights reserved.  
    Vectors3.cs(5,11):           warning           CS0660:  
'Wrox.ProCSharp.OOCSharp.Vector' defines  
operator   =   =   or   operator   !=   but   does   not   override  
Object.Equals(object o)  
    Vectors3.cs(5,11):           warning           CS0661:  
'Wrox.ProCSharp.OOCSharp.Vector' defines  
operator   =   =   or   operator   !=   but   does   not   override  
Object.GetHashCode()
```

在命令行上运行该示例 , 生成如下结果 :

```
Vectors3  
vect1==vect2 returns True  
vect1==vect3 returns False  
vect2==vect3 returns False  
    vect1!=vect2 returns False  
vect1!=vect3 returns True  
vect2!=vect3 returns True
```

### 3. 可以重载的运算符

并不是所有的运算符都可以重载。可以重载的运算符如表 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 = 45.63f;  
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 = 45.63f;  
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精确表示(像 1/3 这样的分数不能精确表示为小数，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精确表示。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.ToInt16()。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。此处的问题是溢出异常实际发生的位置根本不在 Main() 例程中--它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.ToInt16((value-dollars)*100);
        return new Currency(dollars, cents);
    }
}
```

注意，使用 Convert.ToInt16() 计算数字的美分部分，如上所示，但没有使用它计算数字的美元部分。在计算美元值时不需要使用 System.Convert，因为在此我们希望截去 float 值。

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.ToInt16() 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。

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。这是可行的，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 float，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 5.8 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。看看下面的代码：

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

目前，编译器只能采用一种方式来执行这个转换：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定制类型的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，应该就曾在许多 Windows API 中使用过回调。Visual Basic 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。

本章的主要内容如下：

- 委托
- 匿名方法
- 表达式
- 事件

### 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 类的构造函数必须带有一个参数，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递为委托的一个参数。这些将在本章后面讨论。

在 C 和 C++ 中，只能提取函数的地址，并传递为一个参数。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等全部细节。

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、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”。委托实现为派生自基类 System.MulticastDelegate 的类，System.MulticastDelegate 又派生自基类 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 {0}" + firstStringMethod());
    // With firstStringMethod initialized to x.ToString(),
    // the above statement is equivalent to saying
    // Console.WriteLine("String is {0}" + 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();
```

C# 2.0 使用委托推断扩展了委托的语法。为了减少输入量，只要需要委托实例，就可以只传送地址的名称。这称为委托推断。只要编译器可以把委托实例解析为特定的类型，这个 C# 特性就是有效的。下面的示例用 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)。

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。Currency 结构有自己的 ToString() 重载方法和一个与 GetCurrencyUnit() 的签名相同的静态方法，这样，就可以用同一个委托变量调用这些方法了：

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

```
public ushort Cents;
    public Currency(uint dollars, ushort cents)
{
this.Dollars = dollars;
this.Cents = cents;
}
public override string ToString()
{
return string.Format("${0}.{1,-2:00}", Dollars,Cents);
}

public static string GetCurrencyUnit()
{
return "Dollar";
}
public static explicit operator Currency (float value)
{
checked
{
uint dollars =(uint)value;
ushort cents =(ushort)((value-dollars)*100);
return new Currency(dollars,cents);
}
}
public static implicit operator float (Currency value)
{
return value.Dollars + (value.Cents/100.0f);
}
public static implicit operator Currency (uint value)
{
return new Currency(value, 0);
}
public static implicit operator uint (Currency value)
{
return value.Dollars;
}
}
```

下面就可以使用 GetAString 实例，代码如下所示：

```
private delegate string GetAString();
static void Main()
{
int x = 40;
GetAString firstStringMethod = x.ToString();
Console.WriteLine("String is {0}" + firstStringMethod());
    Currency balance = new Currency(34, 50);
    // firstStringMethod references an instance method
```

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

这段代码说明了如何通过委托来调用方法，然后重新给委托指定在类的不同实例上执行的不同方法，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。

### 7.1.3 简单的委托示例

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

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

```
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

#### 7.1.4 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 Comparison(object x, object y);
```

给 Sort 方法指定下述签名：

```
static public void Sort(object [] sortArray, Comparison comparison)
```

这个方法的文档说明强调，comparison 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。

设置完毕后，下面定义类 BubbleSorter：

```
class BubbleSorter
{
    static public void Sort(object [] sortArray, Comparison comparison)
    {
        for (int i=0 ; i<sortArray.Length ; i++)
        {
            for (int j=i+1 ; j<sortArray.Length ; j++)
            {
                if (comparison(sortArray[j], sortArray[i]))
                {
                    object temp = sortArray[i];
                    sortArray[i] = sortArray[j];
                    sortArray[j] = temp;
                }
            }
        }
    }
}
```

```
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("{0}, {1:C}", name, salary);
}
        public static bool CompareSalary(object x, object y)
{
Employee e1 = (Employee) x;
Employee e2 = (Employee) y;
return (e1.salary < e2.salary);
}
}
```

注意，为了匹配 Comparison 委托的签名，在这个类中必须定义 CompareSalary，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强制转换为 Employee 引用，才能进行比较。

注意：

这里除了把对象用作参数之外，还可以使用强类型化的泛型。第 9 章介绍了泛型和泛型委托。

下面编写一些客户端代码，完成排序：

```
using System;
namespace Wrox.ProCSharp.Delegates
{
delegate bool Comparison(object x, object y);
    class Program
{
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);

BubbleSorter.Sort(employees, Employee. CompareSalary);
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

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
```

### 7.1.5 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 = 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();
    Console.WriteLine("ProcessAndDisplayNumber called with value =
{0}"
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 14 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
```

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

在使用匿名方法时，必须遵循两个规则。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。

### 7.1.7 λ 表达式

C# 3.0 为匿名方法提供了一个新的语法：λ 表达式。λ 表达式可以用于委托类型。前面使用匿名方法的例子可以改为使用表达式：

```
using System;  
namespace Wrox.ProCSharp.Delegates  
{  
    class Program  
    {  
        delegate string DelegateTest(string val);  
        static void Main()  
        {  
            string mid = ", middle part,";
```

```
DelegateTest anonDel = param =>
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};

Console.WriteLine(anonDel("Start of string"));
}
```

运算符`=>`的左边列出了匿名方法需要的参数。这有几种编写方式。例如，如果需要在示例代码中把一个字符串参数定义为委托类型，一种方式是在括号中定义类型和变量名：

```
(string param)
```

在 表达式中，不需要给声明添加变量类型，因为编译器知道该类型：

```
(param)
```

如果只有一个参数，就可以删除括号：

```
param
```

表达式的右边列出了实现代码。在示例程序中，实现代码放在花括号中，类似于前面的匿名方法：

```
{
    param += mid;
    param += " and this was added to the string.";
    return param;
};
```

如果实现代码只有一行，也可以删除花括号和 `return` 语句，因为编译器会自动添加该语句。

例如，在下面的委托中，需要一个 `int` 参数，返回一个 `bool` 值：

```
public delegate bool Predicate(int obj)
```

可以声明一个委托变量，并指定一个 表达式。在 表达式中，左边定义了变量 `x`。这个变量的类型自动设置为 `int`，因为这是通过委托定义的。实现代码返回比较 `x>5` 的布尔结果。如果 `x` 大于 5，就返回 `true`，否则返回 `false`。

```
Predicate p1 = x => x > 5;
```

可以把这个 表达式传送给需要谓词参数的方法：

```
list.FindAll(x => x > 5);
```

这里列出了相同的 表达式，但把变量 `x` 的类型定义为 `int`，没有使用变量类型推断功能，还在实现代码中添加了 `return` 语句：

```
list.FindAll(int x) => { return x > 5; });
```

如果使用以前的语法，可以通过匿名方法完成相同的功能：

```
list.FindAll(delegate(int x) { return x > 5; }));
```

通过所有这些改变，C#编译器就创建出了相同的 IL 代码。

修改前面的 `SimpleDelegate` 示例，使用 表达式，可以删除类 `MathOperations`。`Main()`方法现在如下所示：

```
static void Main()
{
    DoubleOp multByTwo = val => val * 2;
```

```
DoubleOp square = val => 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，可以得到与上例相同的结果。但优点是它删除了类。

提示：

表达式可以用于委托是类型的任意地方。类型是 Expression 或 Expression<T>时，也可以使用 表达式。此时编译器会创建一个表达式树，详见第 11 章。

### 7.1.8 协变和抗变

委托调用的方法不需要与委托声明定义的类型相同。因此可能出现协变和抗变。

#### 1. 返回类型协变

方法的返回类型可以派生于委托定义的类型。在下面的示例中，委托 MyDelegate 定义为返回 DelegateReturn 类型。赋予委托实例 d1 的方法返回 DelegateReturn2 类型，DelegateReturn2 派生自 DelegateReturn，因此满足了委托的需求。这称为返回类型协变。

```
public class DelegateReturn
{
}
public class DelegateReturn2 : DelegateReturn
{
}
public delegate DelegateReturn MyDelegate1();
class Program
{
    static void Main()
    {
        MyDelegate1 d1 = Method1;
        d1();
    }
    static DelegateReturn2 Method1()
    {
        DelegateReturn2 d2 = new DelegateReturn2();
        return d2;
    }
}
```

#### 2. 参数类型抗变

术语“参数类型抗变”表示，委托定义的参数可能不同于委托调用的方法。这里是返回类型不同，因为方法使用的参数类型可能派生自委托定义的类型。在下面的示例代码中，委托使用的参数类型是 DelegateParam2，而赋予委托实例 d2 的方法使用的参数类型是 DelegateParam，

DelegateParam 是 DelegateParam2 的基类。

```
public class DelegateParam
{
}

public class DelegateParam2 : DelegateParam
{
}

    public delegate void MyDelegate2(DelegateParam2 p);

class Program
{
    static void Main()
    {
        MyDelegate2 d2 = Method2;
        DelegateParam2 p = new DelegateParam2();
        d2(p);
    }

    static void Method2(DelegateParam p)
    {
    }
}
```

## 7.2 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 Visual Basic 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。Visual Basic 和今天的.NET 把这些传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 Visual Basic 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，表示发生了某件有趣的事情；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。

### 7.2.1 从接收器的角度讨论事件

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，可以编写下面的代码。编译器会生成与前面相同的代码。

```
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，只要它派生自 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。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(button2_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.Show() 函数的调用：

```
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("This only happens in Button 2 click event");
}
```

如果使用 表达式，就不需要 Button\_Click 方法和 Button2\_Click 方法了。事件的代码如下：

```
buttonOne.Click += (sender, e) => labelInfo.Text = "Button One
was pressed";
buttonTwo.Click += (sender, e) => labelInfo.Text = "Button Two was
pressed";
buttonTwo.Click += (sender, e) =>
{
    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)。

## 7.2.2 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。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。

在一行代码中定义事件是 C#中的一个缩写方式，它可以定义添加和删除处理程序的方法，声明委托的一个变量。除了编写一行代码之外，还可以用下面的代码达到相同的效果。声明一个事件类型的变量以及添加和删除事件处理程序的方法。在定义添加和删除事件处理程序的方法时，其语法非常类似于属性。变量值的定义也类似于添加和删除事件处理程序。

```
private static ActionEventHandler action;  
public static event ActionEventHandler Action  
{  
    add  
    {  
        action += value;  
    }  
    remove  
    {  
        action -= value;  
    }  
}
```

提示：

如果不仅仅需要添加和删除事件处理程序，就可以使用定义事件的较长记号。例如，要为多个线程访问添加同步功能。WPF 控件就使用这种较长的记号给事件添加起泡和通道功能。事件的起泡和通道功能详见第 34 章。

基于 EventArgs 的新类 ActionCancelEventArgs 实际上派生自 CancelEventArgs，而 CancelEventArgs 派生自 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 ActionCancelEventArgs 类的代码：

```
public class ActionCancelEventArgs :  
System.ComponentModel.CancelEventArgs  
{  
    public ActionCancelEventArgs() : this(false) {}  
    public ActionCancelEventArgs(bool cancel) : this(false,  
String.Empty) {}  
    public ActionCancelEventArgs(bool cancel, string message) :  
base(cancel)  
{  
    this.message = message;
```

```
}
```

```
public string Message{ get; set;}
```

```
}
```

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 类中使用的信息都由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 OnEventName，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.ToString("yyyy-MM-dd HH:mm:ss");
        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)
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 ActionCancelEventArgs 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。

### 7.3 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。

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

下一章介绍字符串和正则表达式。

## 第 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.RegularExpressions 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，但如果沒有显式指定，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，才需要给字符串分配新内存，才可能移动包含的整个字符串。在添加额外的容量时，从经验来看，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.AppendFormat("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。



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");  
    Console.WriteLine("Not Encoded:\n" + greetingBuilder);  
    for(int i = 'z'; i>='a' ; i--)  
    {  
        char old1 = (char)i;  
        char new1 = (char)(i+1);  
        greetingBuilder = greetingBuilder.Replace(old1, new1);  
    }  
    for(int i = 'Z'; i>='A' ; i-- )  
    {  
        char old1 = (char)i;  
        char new1 = (char)(i+1);  
        greetingBuilder = greetingBuilder.Replace(old1, new1);  
    }  
    Console.WriteLine("Encoded:\n" + greetingBuilder);
```

这段代码使用了 `StringBuilder.Replace()`方法，它的功能与 `String.Replace()`一样，但不需要在过程中复制字符串。在上述代码中，为存储字符串而分配的总存储单元是 150 个字符，用于 `StringBuilder` 实例以及在最后一个 `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)

表 8-2 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 2008、10 Jun 2008、6/10/08 (美国)、10/6/08 (英国)或 10.06.2008 (德国)。

同样，第 6 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，但由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()` 会将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。



## 2. FormattableVector 示例

前面介绍了如何构造格式字符串，下面扩展本书第 6 章的 `Vector` 示例，以多种方式格式化矢量。这个示例的代码可以从 [www.wrox.com](http://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.RegularExpressions 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.RegularExpressions.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，包括可以在搜索操作中找到存储部分文本的工具性程序。本节仅介绍正则表达式的功能。

提示：

正则表达式的更多信息可参阅图书 Beginning Regular Expressions (ISBN : 978-0-7645-7489-4)。

假定应用程序需要把 US 电话号码转换为国际格式。在美国，电话号码的格式为 314-123-1234，常常写作(314)123-1234。在把这个国家格式转换为国际格式时，必须在电话号码的前面加上+1(美国的国家代码)，并给区号加上括号：+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 RegularExpressionsPlayaround，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 3.5 高级编程(第 5 版)》，清华大学出版社引进并出版：

```
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 3.5
Release of .NET, this book will give you the information you need to
master ASP.NET
and build a dynamic, successful, enterprise Web application.";
```

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	iatio n	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 1 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 3.5
        Release of .NET, this book will give you the information you need to
        master ASP.NET
        And build a dynamic, successful, enterprise Web application.";
    string pattern = @"\ba";
    MatchCollection matches = Regex.Matches(text, pattern,
        RegexOptions.IgnoreCase);
    WriteMatches(text, matches);
}
```

下面是一个简单的 Main()方法，可以编辑并选择一个 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 3.5
    Release of .NET, this book will give you the information you need to
    master ASP.NET
    And build a dynamic, successful, enterprise Web application.

    No. of matches: 1
Index: 291,    String: application,    Web application.
```

#### 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。

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String数据类型。String非常重要，这也是本书用一整章的篇幅介绍如何在应用程序中使用和处理String数据类型的原因。

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

最后，使用正则表达式进行高级的字符串处理是搜索和验证字符串的一种极佳工具。

下一章介绍C#的一个强大功能--泛型。

## 第 9 章 泛型

CLR 2.0 的一个新特性是泛型。在 CLR 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。

CLR 2.0(.NET 3.5 基于 CLR 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 从 2.0 开始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; }
    }
}
```

下面的代码把 `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。

`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;
        }
    }
    IEnumerable 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  
    }
```

```
    }
}

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;
    }
}
```

```
public string Title { get; set; }
public string Content { get; set; }
}
```

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 (TDокумент 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。泛型还有几种约束类型，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 T1 派生于泛型类型 T2。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# 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 I Enumerable<T> 接口，它定义了 GetEnumerator() 方法，以返回 I Enumerator<T>。对于 .NET 1.0 中的许多非泛型接口，.NET 从 2.0 开始定义了新的泛型版本，例如 I Comparable<T>：

```
public interface IComparable<T>
```

```
{
```

```
    int CompareTo(T other);
```

```
}
```

第 5 章中的非泛型接口 I Comparable 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;  
}
```

Account 类现在重构为执行接口 IAccount :

```
public class Account : IAccount  
{  
//...  
IAccount 接口定义了只读属性 Balance 和 Name :  
public interface IAccount  
{  
decimal Balance { get; }  
string Name { get; }  
}
```

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 Framework 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,  
TSummary>(IEnumerable<TInput> 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 Action 类型的参数中：

```
decimal amount = Algorithm.Accumulate<Account, decimal>(  
accounts,  
delegate(Account a, decimal d)  
{ return a.Balance + d; });
```

除了使用匿名方法之外，还可以使用 表达式把它传送给第二个参数：

```
decimal amount = Algorithm.Accumulate < Account, decimal > (  
accounts, (a, d) => a.Balance + d);
```

提示：

匿名方法和 表达式参见第 7 章。

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 = Algorithm.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。这里按照第二个 表达式的定义  $a => a.Balance > 2000$ ，只累加结余大于 2000 的账目：

```
decimal amount = Algorithm.AccumulateIf < Account, decimal  
> (
```

```
accounts, (a, d) => a.Balance + d, a => 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。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：

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

这样，就可以使用 表达式传送两个 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, (p1, p2)=>  
    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 表达式，其参数应匹配委托定义的参数：

```
Array.ForEach(persons, 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, p=>  
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;  
}  
public string Name {get; set}  
public string Team {get; set}  
}
```

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, p =>  
    new Racer(String.Format("{0} {1}", p.FirstName, p.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();  
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;
```

提示：

这里调用的 GetNullableType()方法只是任意返回可空 int 的方法的占位符。为了进行测试，可以把它实现为只返回 null 或返回任意整数值。

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
```

## 9.8 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>。  
下一章利用泛型来介绍集合类。

## 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 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

Lookup

HashSet

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 CLR 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 和 IList。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。

对集合和列表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。对于实现了这个接口的集合，可以获得元素个数，把集合复制到数组中 接口 ICollection<T> 扩展了接口 IEnumerable 的功能
ICollection<T>	Count , IsReadOnly , Add() , Clear() , Contains() , CopyTo() Remove()	ICollection<T> 接口是 ICollection 接口的泛型版本。这个接口的泛型版本可以添加和删除元素，获得元素个数
IList	IsFixedSize , IsReadOnly , Item , Add , Clear ,	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

	Contains , IndexOf , Insert , Remove , RemoveAt	
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 NotSupportedException 异常 比较接口 IList 的非泛型版本和泛型版本 ,会发现新的泛型接口只为提供了索引的集合定义了重要的方法和属性。其他方法重构到 ICollection<T>接口中
IDictionary	IsFixedSize , IsReadOnly , Item , Keys , Values , Add() , Clear() , Contains() , GetEnumerator() , Remove()	接口 IDictionary 由其元素包含键和值的非泛型集合实现
IDictionary< TKey , TValue >	Item , Keys , Values , Add() , ContainsKey () , Remove() , TryGetValue()	IDictionary< TKey , TValue > 由其元素包含键和值的泛型集合实现。这个接口比 IDictionary 简单
ILookup< TKey , TElement >	Count, Item, Contains()	ILookup< TKey , TElement > 是 .NET 3.5 中的一个新接口 ,一个键对应多个值的集合使用它。索引器为指定的键返回一个枚举
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。

提示 :

非泛型接口 ICollection 定义的一些属性用于同步不同线程对同一个集合的访问。这些属性不再用于新的泛型接口。其原因是这些属性会导致与同步相关的安全性问题 ,因为集合通常不仅仅必须同步。集合同步的内容可参见第 19 章。

表 10-2 列出了集合类和由这些类执行的集合接口。

表 10-2

集合类	集合接口
ArrayList	IList, ICollection, IEnumerable
Queue	ICollection, IEnumerable
Stack	ICollection, IEnumerable
BitArray	ICollection, IEnumerable
Hashtable	IDictionary, ICollection, IEnumerable
SortedList	IDictionary, ICollection, IEnumerable

List<T>	IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
Queue<T>	IEnumerable<T>, ICollection, IEnumerable
Stack<T>	IEnumerable<T>, ICollection, IEnumerable
LinkedList<T>	ICollection<T>, IEnumerable<T>, ICollection, IEnumerable
HashSet<T>	ICollection<T>, IEnumerable<T>, IEnumerable
Dictionary<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
SortedDictionary<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
SortedList<TKey, TValue>	IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
Lookup<TKey, TElement>	ILookup<TKey, TElement>, IEnumerable<IGrouping<TKey, TElement>>, IEnumerable

## 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 4 个字段：firstname、lastname、country 和获胜次数。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次数，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;
    }

        public string Firstname{ get; set; }
        public string Lastname{ get; set; }
        public string Country { get; set; }
        public int Wins { get; set; }

        public override string ToString()
```

```
{  
    return String.Format("{0} {1}",  
        FirstName, LastName);  
}  
    public string ToString(string format, IFormatProvider  
formatProvider)  
    {  
        switch (format.ToUpper())  
        {  
            case null:  
            case "N": //Name  
                return ToString();  
            case "F": //FirstName  
                return Firstname;  
            case "L": //LastName  
                return Lastname;  
            case "W": //Wins  
                return String.Format("{0}, Wins: {1}",  
                    ToString(), Wins);  
            case "C": // Country  
                return String.Format(  
                    "{0}, Country: {1}",  
                    ToString(), Country);  
            case "A": // All  
                return String.Format(  
                    "{0}, {1} Wins: {2}",  
                    ToString(), Country, Wins);  
            default:  
                throw new FormatException(String.Format(  
                    formatProvider,  
                    "Format {0} is not supported",  
                    format));  
        }  
    }  
    public string ToString(string format)  
    {  
        return ToString(format, null);  
    }  
    public int CompareTo(Racer other)  
    {  
        int compare = this.LastName.CompareTo(  
            other.LastName);  
        if (compare == 0)  
            return this.FirstName.CompareTo(  
                other.FirstName);  
        return compare;  
    }  
}
```

```
}
```

### 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。

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. 集合初始化器

C# 3.0 允许使用集合初始化器给集合赋值。集合初始化器的语法类似于第 5 章介绍的数组初始化器。使用集合初始化器，可以在初始化集合时，在花括号中给集合赋值：

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

提示：

集合初始化器是 C# 3.0 编程语言的一个特性，没有反映在已编译的程序集的 IL 代码中。编译器会把集合初始化器变成给初始化列表中的每一项调用 Add()方法。

#### 2. 添加元素

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

```
List intList = new List();
intList.Add(1);
intList.Add(2);

List stringList = new List();
stringList.Add("one");
stringList.Add("two");
```

变量 racers 定义为类型 List。使用 new 操作符创建相同类型的一个新对象。因为类 List 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 5 个一级方程式赛手，并添加到集合中。前 3 个用集合初始化器添加，后两个通过显式调用 Add()方法来添加。

```
Racer graham = new Racer("Graham", "Hill", "UK", 14);
Racer emerson = new Racer("Emerson", "Fittipaldi", "Brazil", 14);
Racer mario = new Racer("Mario", "Andretti", "USA", 12);
List racers = new List(20);
{graham, emerson, mario};

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

使用 List 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 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 } );
```

提示：

集合初始化器只能在声明集合时使用。AddRange()方法则可以在初始化集合后调用。

如果在实例化列表时知道集合的元素个数，也可以将执行 IEnumerable 的任意对象传送给类的构造函数。这非常类似于 AddRange()方法：

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

### 3. 插入元素

使用 Insert()方法可以在指定位置插入元素：

```
racers.Insert(3, new Racer("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。

### 4. 访问元素

执行了 IList 和 IList 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。指定 racers[3]，可以访问列表中的第 4 个元素：

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

可以用 Count 属性确定元素个数，再使用 for 循环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
{
    Console.WriteLine(racers[i]);
```

```
}
```

提示：

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

List 执行了接口 IEnumerable，所以也可以使用 foreach 语句迭代集合中的元素。

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

注意：

编译器解析 foreach 语句时，利用了接口 IEnumerable 和 IEnumerator，参见第 5 章。

除了使用 foreach 语句之外，List 类还提供了 ForEach()方法，它用 Action 参数声明。

```
public void ForEach(Action action);
```

ForEach()的执行代码如下。ForEach()迭代集合中的每一项，调用传递作为每一项的参数的方法。

```
public class List < T > : IList < T >
{
    private T[] items;
    //...
    public void ForEach(Action < T > action)
    {
        if (action == null) throw new ArgumentNullException("action");
        foreach (T item in items)
        {
            action(item);
        }
    }
    //...
}
```

为了给 ForEach 传递一个方法，Action 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 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);
    });

```

在 C# 3.0 中，还可以使用 表达式和以委托作为参数的方法。使用匿名方法执行的迭代也可以用

表达式定义：

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

注意：

匿名方法和表达式详见第 7 章。

## 5. 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，确定集合中的元素是否等于传送给 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 类中 Equals()方法的默认实现代码对值类型进行按位比较，对引用类型只比较其引用。

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。这个方法在搜索元素时使用下面将讨论的 Predicate<T>参数。要删除集合中的所有元素，可以使用 ICollection<T>接口定义的 Clear()方法。

## 6. 搜索

有不同的方式在集合中搜索元素。可以获得要查找的元素的索引，或者搜索元素本身。可以使用的方法有 IndexOf()、LastIndexOf()、FindIndex()、FindLastIndex()、Find()和 FindLast()。如果只检查元素是否存在，List<T>类提供了 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 racer)
    {
        if(racer == null) throw new ArgumentNullException("racer");
        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 表达式。结果与前面完全相同。现在 表达式定义了实现代码，来搜索 Country 属性设置为 Finland 的元素。

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

与 IndexOf()方法类似，使用 FindIndex()方法也可以指定搜索开始的索引和要迭代的元素个数。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(r=> 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(r=> 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。

## 7. 排序

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

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、国籍或获胜次数排序。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 表达式传送给 Sort() 方法，按获胜次数排序。两个参数的类型是 Racer，在其实现代代码中，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，所以获胜次数以降序方式排序。方法调用完后，完整的赛手列表就按赛手的获胜次数排序。

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

也可以调用 Reverse() 方法，倒转整个集合的顺序。

## 8. 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、国籍和获胜次数，而 Person 类型只包含姓名。为了进行转换，可以忽略赛手的国籍和获胜次数，但姓名必须转换：

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

```
public Person(string name)
{
    this.name = name;
}
public override string ToString()
{
    return name;
}
```

转换时调用了 racers.ConvertAll<Person>()方法。这个方法的变元定义为一个 表达式，其变元的类型是 Racer，返回类型是 Person。在 表达式的实现代码中，创建并返回了一个新的 Person 对象。对于 Person 对象，把 Firstname 和 Lastname 传送给构造函数：

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

转换的结果是一个列表，其中包含转换过来的 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>类执行了 IEnumerable 和 ICloneable 接口。Queue<T>泛型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 10-3 所示。

表 10-3

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()	把元素从队列复制到一个已有的数组中
ToArray()	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 19 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 10-4 所示。

表 10-4

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()	把元素从栈复制到一个已有的数组中。
ToArray()	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.WriteLine(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.WriteLine("First iteration: ");
foreach (char item in alphabet)
{
    Console.WriteLine(item);
}

Console.WriteLine("Second iteration: ");
while (alphabet.Count > 0)
{
    Console.WriteLine(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 集合类没有非泛型集合的类似版本。LinkedList 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 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 包含 ListNode 类型的元素的原因。使用 ListNode 类，可以获得列表中的下一个元素和上一个元素。表 10-5 描述了 ListNode 的属性。

表 10-5

ListNode 的属性	说 明
List	返回与节点相关的 LinkedList
Next	返回当前节点之后的节点。其返回类型是 ListNode
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 执行了 IEnumerable、ICollection、ICollection、IEnumerable、ISerializable 和 IDeserializationCallback 接口。这个类的成员如表 10-6 所示。

表 10-6

LinkedList 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。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。使用这个节点，可以逆序迭代集合中的其他节点
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 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，可以添加类型 LinkedListNode 或类型 T 的对象。如果传送 T 对象，则创建一个新 LinkedListNode 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() FindLast()	从链表的开头开始搜索传送给它的元素，并返回一个 LinkedListNode 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 和一个列表 List。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 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 包含 LinkedListNode 类型的元素。类 LinkedListNode 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 定义为 List<LINKEDLISTNODE>。为了快速访问每个优先级的最后一个文档，集合 List 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。

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 类型的集合包含所有的文档。List<LINKEDLISTNODE>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 documentList;
    // priorities 0..9
    private readonly List<LINKEDLISTNODE> priorityNodes;
    public PriorityDocumentManager()
    {
        documentList = new LinkedList();
        priorityNodes = new List<LINKEDLISTNODE>(10);
        for (int i = 0; i < 10; i++)
        {
            priorityNodes.Add(new LinkedListNode(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;
}
}
```

在 Main()方法中，PriorityDocumentManager 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 `ArgumentException` 类型的异常。如果索引器使用相同的键，就用新值替代旧值。

```
SortedList<string, string> books = new SortedList<string, string>();
books.Add(".NET 2.0 Wrox Box", "978-0-470-04840-5");
books.Add("Professional C# 2005 with .NET 3.0",
"978-0-470-12472-7");
books["Beginning Visual C# 2005"] = "978-0-7645-4382-1";
books["Professional C# 2008"] = "978-0-470-19137-6";
```

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 with .NET 3.0, 978-0-470-12472-7
```

Professional C# 2008, 978-0-470-19137-6

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
978-0-470-12472-7
978-0-470-19137-6
.NET 2.0 Wrox Box
Beginning Visual C# 2005
Professional C# 2005 with .NET 3.0
Professional C# 2008
```

SortedList< TKey, TValue > 类的属性如表 10-7 所示。

表 10-7

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递该比较器。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 > 类型类似于本章前面介绍的其他集合。见表 10-8。区别是 SortedList< T > 需要一个键和一个值。

表 10-8

SortedList 的方法	说 明
Add()	把带有键和值的元素放在有序表中
Remove()	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
RemoveAt()	
Clear()	删除有序表中的所有元素
ContainsKey()	ContainsKey() 和 ContainsValue() 方法检查有序表是否包含指定
ContainsValue()	

ContainsValue()	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。

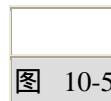


图 10-5

.NET Framework 提供了几个字典类。可以使用的最主要的类是 Dictionary< TKey, TValue >。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。

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 IEqualityComparer< T >. Equals()方法，或重写 Object 类的 Equals()方法。因为不同的键对象可能返回相同的散列码，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。IEqualityComparer<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并比较键。

## 10.7.2 字典示例

字典示例程序建立了一个员工字典。该字典是用 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 结构定义了在字典中使用的键，该结构的成员是表示员工的一个前缀字符和一个数字。这两个变量都是只读的，只能在构造函数中初始化。字典中的键不应改变，这是必须保证的。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 struct EmployeeId : IEquatable<EmployeeId>
{
    private readonly char prefix;
    private readonly int number;
    public EmployeeId(string id)
    {
        if (id == null) throw new ArgumentNullException("id");
        prefix = (id.ToUpper())[0];
        int numLength = id.Length - 1;
        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)
{
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 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格式字符串创建字符串表示，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()
{
return String.Format("{0}: {1,-20} {2:C}",
id.ToString(), name, salary);
}
}
```

}

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 Carl 和 Matt 所示：

```
static void Main()
{
    Dictionary<EmployeeId, Employee> employees =
        new Dictionary<EmployeeId, Employee>(31);
    EmployeeId idJeff = new EmployeeId("C7102");
    Employee jeff = new Employee(idJeff,
        "Jeff Gordon", 5164580.00m);
    employees.Add(idJeff, jeff);
    Console.WriteLine(jeff);

    EmployeeId idTony = new EmployeeId("C7105");
    Employee tony = new Employee(idTony,
        "Tony Stewart", 4814200.00m);
    employees.Add(idTony, tony);
    Console.WriteLine(tony);

    EmployeeId idDenny = new EmployeeId("C8011");
    Employee denny = new Employee(idDenny,
        "Denny Hamlin", 3718710.00m);
    employees.Add(idDenny, denny);
    Console.WriteLine(denny);

    EmployeeId idCarl = new EmployeeId("F7908");
    Employee carl = new Employee(idCarl,
        "Carl Edwards", 3285710.00m);
    employees[idCarl] = carl;
    Console.WriteLine(carl);

    EmployeeId idMatt = new EmployeeId("F7203");
    Employee matt = new Employee(idMatt,
        "Matt Kenseth", 4520330.00m);
    employees[idMatt] = matt;
    Console.WriteLine(matt);
}
```

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) > C7102
C007102: Jeff Gordon $5,164,580.00
Enter employee ID (format:A999999, X to exit) > F7908
F007908: Carl Edwards $3,285,710.00
Enter employee ID (format:A999999, X to exit) > X
```

### 10.7.3 Lookup 类

Dictionary< TKey, TValue>只为每个键支持一个值。新类 Lookup< TKey, TElement>是.NET 3.5 中新增的，它类似于 Dictionary< TKey, TValue>，但把键映射到一个值集上。这个类在程序集 System.Core 中实现，用 System.Linq 命名空间定义。

Lookup< TKey, TElement>的方法和属性如表 10-9 所示。

表 10-9

Lookup< TKey, TElement>的方法和属性	说 明
Count	属性 Count 返回集合中的元素个数
Item	使用索引器可以根据键访问特定的元素。因为同一个键可以对应多个值，所以这个属性返回所有值的枚举
Contain()	方法 Contains()根据是否用 key 参数传送元素，返回一个布尔值
ApplyResultSelector()	ApplyResultSelector()根据传送给它的转换函数，转换每一项，返回一个集合

Lookup< TKey, TElement>不能像一般的字典那样创建，而必须调用方法 ToLookup()，它返回一个 Lookup< TKey, TElement>对象。方法 ToLookup()是一个扩展方法，可以用于实现了 IEnumerable< T>的所有类。在下面的例子中，填充了一列 Racer 对象。List< T>实现了 IEnumerable< T>，所以可以在赛手列表上调用方法 ToLookup()。这个方法需要一个 Func< TSource, TKey>类型的委托 Func< TSource, TKey>类型定义了键的选择器。这里使用 表达式 r=>r.Country，根据国家来选择赛手。Foreach 循环只使用索引器访问来自奥地利的赛手。

提示：

扩展方法详见第 11 章，表达式参见第 7 章。

```
List < Racer > racers = new List < Racer > ();
racers.Add(new Racer("Jacques", "Villeneuve",
"Canada", 11));
racers.Add(new Racer("Alan", "Jones",
"Australia", 12));
racers.Add(new Racer("Jackie", "Stewart",
"United Kingdom", 27));
racers.Add(new Racer("James", "Hunt",
"United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham",
"Australia", 14));

    Lookup < string, Racer > lookupRacers =
(Lookup < string, Racer > )
racers.ToLookup(r => r.Country);
    foreach (Racer r in lookupRacers["Australia"])
{
    Console.WriteLine(r);
}
```

结果显示了来自奥地利的赛手：

```
Alan Jones
Jack Brabham
```

#### 10.7.4 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 10-10 所示。

表 10-10

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
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

自从.NET 2.0 以来，泛型字典优先于基于对象的字典，如表 10-11 所示。

表 10-11

泛型字典	说 明
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 > 实现为一个基于数组的链表，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 HashSet

.NET 3.5 在 System.Collections.Generic 命名空间中包含一个新的集合类：HashSet< T >。这个集合类包含不重复项的无序列表。这种集合称为“集(set)”。集是一个保留字，所以该类有另一个名称 HashSet< T >。这个名称很容易理解，因为这个集合基于散列值，插入元素的操作非常快，不需要像 List< T > 类那样重排集合。

HashSet< T > 类提供的方法可以创建合集和交集。表 10-12 列出了改变集的值的方法。

表 10-12

HashSet< T > 的修改方法	说 明
Add()	如果某元素不在集合中，Add()方法就把该元素添加到集合中。在其返回值 Boolean 中，返回元素是否添加的信息
Clear()	方法 Clear() 删除集合中的所有元素
Remove()	Remove()方法删除指定的元素
RemoveWhere()	RemoveWhere()方法需要一个 Predicate< T > 委托作为参数。删除满足谓词条件的所有元素
CopyTo()	CopyTo() 把集合中的元素复制到一个数组中
ExceptWith()	ExceptWith()方法把一个集合作为参数，从集中删除该集合中的所有元素
IntersectWith()	IntersectWith() 修改了集，仅包含所传送的集合和集中都有的元素
UnionWith()	UnionWith()方法把传送为参数的集合中的所有元素添加到集中

表 10-13 列出了仅返回集的信息、不修改元素的方法。

表 10-13

HashSet< T > 的验证方法	说 明
Contains()	如果所传送的元素在集合中，方法 Contains() 就返回 true
IsSubsetOf()	如果参数传送的集合是集的一个子集，方法 IsSubsetOf() 就返回 true
IsSupersetOf()	如果参数传送的集合是集的一个超集，方法 IsSupersetOf() 就返回 true
Overlaps()	如果参数传送的集合中至少有一个元素与集中的元素相同，Overlaps() 就返回 true
SetEquals()	如果参数传送的集合和集包含相同的元素，方法 SetEquals() 就返回 true

在示例代码中，创建了 3 个字符串类型的新集，并用一级方程式汽车填充。HashSet< T > 类实现了 ICollection< T > 接口。但是在该类中，Add() 方法是显式实现的，还提供了另一个 Add() 方法。Add() 方法的区别是返回类型，它返回一个布尔值，说明是否添加了元素。如果该元素已经在集中，就不添加它，并返回 false。

HashSet < string > companyTeams =
-----------------------------------

```
new HashSet<string>()
{ "Ferrari", "McLaren", "Toyota", "BMW",
"Renault", "Honda" };
HashSet<string> traditionalTeams =
new HashSet<string>()
{ "Ferrari", "McLaren" };
HashSet<string> privateTeams =
new HashSet<string>()
{ "Red Bull", "Toro Rosso", "Spyker",
"Super Aguri" };
if (privateTeams.Add("Williams"))
Console.WriteLine("Williams added");
if (!companyTeams.Add("McLaren"))
Console.WriteLine(
"McLaren was already in this set");
```

两个 Add()方法的输出写到控制台上：

```
Williams added
McLaren was already in this set
```

方法 IsSubsetOf()和 IsSupersetOf()比较集和实现了 IEnumerable<T>接口的集合，返回一个布尔结果。这里 IsSubsetOf()验证 traditionalTeams 中的每个元素是否都包含在 companyTeams 中，IsSupersetOf()验证 traditionalTeams 是否没有与 companyTeams 比较的额外元素。

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
Console.WriteLine("traditionalTeams is " +
"subset of companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
Console.WriteLine(
"companyTeams is a superset of " +
"traditionalTeams");
}
```

这个验证的结果如下：

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
Williams 也是一个传统队，因此这个队添加到 traditionalTeams
集合中：
```

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
Console.WriteLine("At least one team is " +
"the same with the traditional " +
"and private teams");
}
```

这有一个重叠，所以结果如下：

At least one team is the same with the traditional and private teams.

调用 UnionWith()方法，给变量 allTeams 填充了 companyTeams、PrivateTeams 和 traditionalTeams 的合集：

```
    HashSet < string > allTeams =  
    new HashSet < string > (companyTeams);  
    allTeams.UnionWith(privateTeams);  
    allTeams.UnionWith(traditionalTeams);  
    Console.WriteLine();  
    Console.WriteLine("all teams");  
    foreach (var team in allTeams)  
    {  
        Console.WriteLine(team);  
    }
```

这里返回所有的队，但每个队都只列出一次，因为集只包含唯一值：

```
Ferrari  
McLaren  
Toyota  
BMW  
Renault  
Honda  
Red Bull  
Toro Rosso  
Spyker  
Super Aguri  
Williams
```

方法 ExceptWith()从 allTeams 集中删除所有的私人队：

```
    allTeams.ExceptWith(privateTeams);  
    Console.WriteLine();  
    Console.WriteLine("no private team left");  
    foreach (var team in allTeams)  
    {  
        Console.WriteLine(team);  
    }
```

集合中的其他元素不包含私人队：

```
Ferrari  
McLaren  
Toyota  
BMW  
Renault  
Honda
```

## 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 10-14 所示。

表 10-14

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
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()方法执行二元 AND，只有两个输入数组的位都设置为 1，结果位才是 1。Or()方法执行二元 OR，只要有一个输入数组的位设置为 1，结果位就是 1。Xor()方法是异或操作，只有一个输入数组的位设置为 1，结果位才是 1

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.WriteLine("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：
Console.WriteLine(" not ");
DisplayBits(bits1);
bits1.Not();
Console.WriteLine(" = ");
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。

```
not 10111010 = 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 10-15 列出了 BitVector32 中与 BitArray 完全不同的成员。

表 10-15

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 类型的片断来获取和设置值。
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()，把第一个掩码传递为参数，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{000000000000000000000000000011101}
```

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 0xffff 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 0xffff，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   BBBB BBBB  
AAAAAAAAAAAAAA  
BitVector32.Section sectionA = BitVector32.CreateSection(0xffff);  
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));
```

方法 IntToBinaryString 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;  
}  
if (removeTrailingZero && sb.Length > 0 && sb[sb.Length - 1] == '0')  
{  
sb.Length -= 1;  
}  
return sb.ToString();  
}
```

```
    }
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。

表 10-16 列出了集合类及其执行不同操作的性能，例如添加、插入和删除元素。使用这个表可以选择性能最佳的集合类。左列是集合类，`Add` 列给出了在集合中添加元素所需的时间。`List < T >` 和 `HashSet < T >` 类把 `Add` 方法定义为在集合中添加元素。其他集合类用不同的方法把元素添加到集合中。例如 `Stack < T >` 类定义了 `Push()` 方法，`Queue < T >` 类定义了 `Enqueue()` 方法。这些信息也列在表

中。

如果单元格中有多个大 O 值，表示若集合需要重置大小，该操作就需要一定的时间。例如，在 List< T >类中，添加元素的时间是 O(1)。如果集合的容量不够大，需要重置大小，重置大小的操作就需要 O(n)。集合越大，重置大小操作的时间就越长。最好避免重置集合的大小，而应把集合的容量设置为一个可以包含所有元素的值。

如果单元格的内容是 na，就表示这个操作不能应用于这种集合类型。

表 10-16

集 合	Add	Insert	Remove	Item	Sort	Find
List<T>	如果集合必须重置大小，就是 O(1)或 O(n)	O(n)	O(n)	O(1)	O (n log n), 最坏情况 O(n ^ 2)	O(n)
Stack<T>	Push()，如果栈必须重置大小，就是 O(1)或 O(n)	na	Pop(), O(1)	na	na	na
Queue<T>	Enqueue()，如果队列必须重置大小，就是 O(1)或 O(n)	na	Dequeue(), O(1)	na	na	na
HashSet<T>	如果集必须重置大小，就是 O(1)或 O(n)	Add() O(1) 或 O(n)	O(1)	na	na	na
LinkedList<T>	AddLast() O(1)	AddAfter() O(1)	O(1)	na	na	O(n)
Dictionary< TKey, TValue >	O(1) 或 O(n)	na	O(1)	O(1)	na	na
SortedDictionary< TKey, TValue >	O(log n)	na	O(log n)	O(log n)	na	na
SortedList< TKey, TValue >	无序数据为 O(n)，如果必须重置大小，到列表的尾部就是 O(log n)	na	O(n)	读写是 O(log n)， 如果键在列表中， 就是 O(log n)；如 果键不在列表中， 就是 O(n)	na	na

## 10.11 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。集(其名称是 HashSet<T>)用于无序的唯一项。

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 章将详细介绍 LINQ，这是 C# 3.0 主要的新语言扩展。

## 第 11 章 Language Integrated Query

Language Integrated Query(LINQ)是 C# 3.0 和.NET 3.5 中最重要的新功能。LINQ 集成了 C# 编程语言中的查询语法，可以用相同的语法访问不同的数据源。LINQ 提供了不同数据源的抽象层，所以可以使用相同的语法。

本章介绍 LINQ 的核心功能和 C# 3.0 中支持新特性的语言扩展。本章的主要内容如下：

用 List<T>在对象上执行传统查询

扩展方法

表达式

LINQ查询

标准查询操作符

表达式树

LINQ提供程序

提示：

本章介绍 LINQ 的核心功能。读完本章后，在数据库中使用 LINQ 的内容可查阅第 27 章，查询 XML 数据的内容可参见第 29 章。

## 11.1 LINQ 概述

在介绍 LINQ 的特性之前，本节先用一个例子来说明在 LINQ 推出之前如何查询对象。在此过程中，将说明查询如何演变为 LINQ 查询。了解了这些步骤，就知道 LINQ 查询的意义了。

本章的示例基于一级方程式世界冠军。查询将搜索一个 Racer 对象列表。第一个查询按照比赛的顺序得到巴西所有一级方程式世界冠军。

### 11.1.1 使用 List<T>的查询

过滤和排序的第一个变体是在一个 List<T>类型的列表中搜索数据。在搜索开始之前，必须确定对象类型和对象列表。

给对象定义类型 Racer。Racer 定义了几个属性和一个重载的 ToString()方法，该方法以字符串格式显示赛手。这个类实现了接口 IFormattable，以支持格式字符串的不同变体，这个类还实现了接口 IComparable<Racer>，它根据 Lastname 为一组赛手排序。为了执行更高级的查询，类 Racer 不仅包含单值属性，如 Firstname、Lastname、Wins、Country 和 Starts，还包含多值属性，如 Cars 和 Years。Years 属性列出了赛手获得冠军的年份。一些赛手曾多次获得冠军。Cars 属性用于列出赛手在获得冠军的年份中使用的所有车型。

```
using System;
using System.Text;

namespace Wrox.ProCSharp.LINQ
{
    [Serializable]
    public class Racer : IComparable<Racer>, IFormattable
    {
```

```
public string FirstName {get; set; }

public string LastName {get; set; }

public int Wins {get; set; }

public string Country {get; set; }

public int Starts {get; set; }

public string[] Cars { get; set; }

public int[] Years { get; set; }

public override string ToString()

{

    return String.Format("{0} {1}", Firstname , Lastname);

}

public int CompareTo(Racer other)

{

    return this.lastname.CompareTo(other.lastname);

}

public string ToString(string format)

{

    return ToString(format, null);

}

public string ToString(string format, IFormatProvider formatProvider)

{

    switch (format)

    {

        case null:

        case "N":

            return ToString();

        case "F":

            return Firstname;
```

```
case "L":  
    return LastName;  
  
case "C":  
    return Country;  
  
case "S":  
    return Starts.ToString();  
  
case "W":  
    return Wins.ToString();  
  
case "A":  
    return String.Format("{0} {1}, {2}; " +  
        " starts: {3}, wins: {4}",  
        FirstName, LastName, Country,  
        Starts, Wins);  
  
default:  
    throw new FormatException(String.Format(  
        "Format {0} not supported", format));  
}  
}  
}  
}
```

类 Formula1 在 GetChampions()方法中返回一组赛手。这个列表包含了 1950 到 2007 年之间的所有一级方程式冠军。

```
using System;  
using System.Collections.Generic;  
namespace Wrox.ProCSharp.LINQ  
{  
    public static class Formula1  
    {  
        public static IList<Racer> GetChampions()
```

```
{  
  
List<Racer> racers = new List<Racer>(40);  
  
racers.Add(new Racer() { FirstName = "Nino",  
  
LastName = "Farina", Country = "Italy",  
  
Starts = 33, Wins = 5,  
  
Years = new int[] { 1950 },  
  
Cars = new string[] { "Alfa Romeo" }));  
  
racers.Add(new Racer() {  
  
FirstName = "Alberto",  
  
LastName = "Ascari", Country = "Italy",  
  
Starts = 32, Wins = 10,  
  
Years = new int[] { 1952, 1953 },  
  
Cars = new string[] { "Ferrari" }));  
  
racers.Add(new Racer() {  
  
FirstName = "Juan Manuel",  
  
LastName = "Fangio",  
  
Country = "Argentina", Starts = 51,  
  
Wins = 24, Years = new int[]  
  
{ 1951, 1954, 1955, 1956, 1957 },  
  
Cars = new string[] { "Alfa Romeo",  
  
"Maserati", "Mercedes",  
  
"Ferrari" }));  
  
racers.Add(new Racer() { FirstName = "Mike",  
  
LastName = "Hawthorn", Country = "UK",  
  
Starts = 45, Wins = 3,  
  
Years = new int[] { 1958 },  
  
Cars = new string[] { "Ferrari" }));  
  
racers.Add(new Racer() { FirstName = "Phil",  
  
LastName = "Hill", Country = "USA",
```

```
Starts = 48, Wins = 3,  
  
Years = new int[] { 1961 },  
  
Cars = new string[] { "Ferrari" } );  
  
racers.Add(new Racer() { FirstName = "John",  
  
    LastName = "Surtees", Country = "UK",  
  
    Starts = 111, Wins = 6,  
  
    Years = new int[] { 1964 },  
  
    Cars = new string[] { "Ferrari" } );  
  
racers.Add(new Racer() { FirstName = "Jim",  
  
    LastName = "Clark", Country = "UK",  
  
    Starts = 72, Wins = 25,  
  
    Years = new int[] { 1963, 1965 },  
  
    Cars = new string[] { "Lotus" } );  
  
racers.Add(new Racer() { FirstName = "Jack",  
  
    LastName = "Brabham",  
  
    Country = "Australia", Starts = 125,  
  
    Wins = 14,  
  
    Years = new int[] { 1959, 1960, 1966 },  
  
    Cars = new string[] { "Cooper",  
  
        "Brabham" } } );  
  
racers.Add(new Racer() { FirstName = "Denny",  
  
    LastName = "Hulme",  
  
    Country = "New Zealand", Starts = 112,  
  
    Wins = 8,  
  
    Years = new int[] { 1967 },  
  
    Cars = new string[] { "Brabham" } );  
  
racers.Add(new Racer() { FirstName = "Graham",  
  
    LastName = "Hill", Country = "UK",  
  
    Starts = 176, Wins = 14,
```

```
Years = new int[] { 1962, 1968 },  
  
Cars = new string[] { "BRM", "Lotus" }  
});  
  
racers.Add(new Racer() { FirstName = "Jochen",  
  
LastName = "Rindt", Country = "Austria",  
  
Starts = 60, Wins = 6,  
  
Years = new int[] { 1970 },  
  
Cars = new string[] { "Lotus" } }));  
  
racers.Add(new Racer() { FirstName = "Jackie",  
  
LastName = "Stewart", Country = "UK",  
  
Starts = 99, Wins = 27,  
  
Years = new int[] { 1969, 1971, 1973 },  
  
Cars = new string[] { "Matra",  
  
"Tyrrell" } }));  
  
racers.Add(new Racer() {  
  
FirstName = "Emerson",  
  
LastName = "Fittipaldi",  
  
Country = "Brazil", Starts = 143,  
  
Wins = 14, Years = new int[] { 1972,  
  
1974 },  
  
Cars = new string[] { "Lotus",  
  
"McLaren" } }));  
  
racers.Add(new Racer() { FirstName = "James",  
  
LastName = "Hunt", Country = "UK",  
  
Starts = 91, Wins = 10,  
  
Years = new int[] { 1976 },  
  
Cars = new string[] { "McLaren" } }));  
  
racers.Add(new Racer() { FirstName = "Mario",  
  
LastName = "Andretti", Country = "USA",
```

```
Starts = 128, Wins = 12,  
  
Years = new int[] { 1978 },  
  
Cars = new string[] { "Lotus" } );  
  
racers.Add(new Racer() { FirstName = "Jody",  
  
    LastName = "Scheckter",  
  
    Country = "South Africa", Starts = 112,  
  
    Wins = 10,  
  
    Years = new int[] { 1979 },  
  
    Cars = new string[] { "Ferrari" } );  
  
racers.Add(new Racer() { FirstName = "Alan",  
  
    LastName = "Jones",  
  
    Country = "Australia", Starts = 115,  
  
    Wins = 12,  
  
    Years = new int[] { 1980 },  
  
    Cars = new string[] { "Williams" } );  
  
racers.Add(new Racer() { FirstName = "Keke",  
  
    LastName = "Rosberg",  
  
    Country = "Finland", Starts = 114,  
  
    Wins = 5,  
  
    Years = new int[] { 1982 },  
  
    Cars = new string[] { "Williams" } );  
  
racers.Add(new Racer() { FirstName = "Niki",  
  
    LastName = "Lauda", Country = "Austria",  
  
    Starts = 173, Wins = 25,  
  
    Years = new int[] { 1975, 1977, 1984 },  
  
    Cars = new string[] { "Ferrari",  
        "McLaren" } } );  
  
racers.Add(new Racer() { FirstName = "Nelson",  
  
    LastName = "Piquet", Country = "Brazil",
```

```
Starts = 204, Wins = 23,  
  
Years = new int[] { 1981, 1983, 1987 },  
  
Cars = new string[] { "Brabham",  
"Williams" } );  
  
racers.Add(new Racer() { FirstName = "Ayrton",  
  
LastName = "Senna", Country = "Brazil",  
  
Starts = 161, Wins = 41,  
  
Years = new int[] { 1988, 1990, 1991 },  
  
Cars = new string[] { "McLaren" } );  
  
racers.Add(new Racer() { FirstName = "Nigel",  
  
LastName = "Mansell", Country = "UK",  
  
Starts = 187, Wins = 31,  
  
Years = new int[] { 1992 },  
  
Cars = new string[] { "Williams" } );  
  
racers.Add(new Racer() { FirstName = "Alain",  
  
LastName = "Prost", Country = "France",  
  
Starts = 197, Wins = 51,  
  
Years = new int[] { 1985, 1986, 1989,  
1993 },  
  
Cars = new string[] { "McLaren",  
"Williams" } );  
  
racers.Add(new Racer() { FirstName = "Damon",  
  
LastName = "Hill", Country = "UK",  
  
Starts = 114, Wins = 22,  
  
Years = new int[] { 1996 },  
  
Cars = new string[] { "Williams" } );  
  
racers.Add(new Racer() {  
  
FirstName = "Jacques",  
  
LastName = "Villeneuve",
```

```
Country = "Canada", Starts = 165,  
Wins = 11, Years = new int[] { 1997 },  
Cars = new string[] { "Williams" } );  
  
racers.Add(new Racer() { FirstName = "Mika",  
LastName = "Hakkinen",  
Country = "Finland", Starts = 160,  
Wins = 20, Years = new int[] { 1998,  
1999 },  
Cars = new string[] { "McLaren" } );  
  
racers.Add(new Racer() {  
FirstName = "Michael",  
LastName = "Schumacher",  
Country = "Germany", Starts = 250,  
Wins = 91,  
Years = new int[] { 1994, 1995, 2000,  
2001, 2002, 2003, 2004 },  
Cars = new string[] { "Benetton",  
"Ferrari" } });  
  
racers.Add(new Racer() {  
FirstName = "Fernando",  
LastName = "Alonso", Country = "Spain",  
Starts = 105, Wins = 19,  
Years = new int[] { 2005, 2006 },  
Cars = new string[] { "Renault" } );  
racers.Add(new Racer() { FirstName = "Kimi",  
LastName = "Räikkönen",  
Country = "Finland", Starts = 122,  
Wins = 15, Years = new int[] { 2007 },  
Cars = new string[] { "Ferrari" } });  
return racers;  
}  
}  
}
```

对于后面在多个列表中执行的查询，GetConstructorChampions()方法返回所有的制造商冠军。制造商冠军是从 1958 年开始设立的。

```
public static IList < Team >
    GetConstructorChampions()
{
    List < Team > teams = new List < Team > (20);
    teams.Add(new Team() { Name = "Vanwall",
        Years = new int[] { 1958 } });
    teams.Add(new Team() { Name = "Cooper",
        Years = new int[] { 1959, 1960 } });
    teams.Add(new Team() { Name = "Ferrari",
        Years = new int[] { 1961, 1964, 1975,
            1976, 1977, 1979, 1982, 1983, 1999,
            2000, 2001, 2002, 2003, 2004, 2007 } });
    teams.Add(new Team() { Name = "BRM",
        Years = new int[] { 1962 } });
    teams.Add(new Team() { Name = "Lotus",
        Years = new int[] { 1963, 1965, 1968,
            1970, 1972, 1973, 1978 } });
    teams.Add(new Team() { Name = "Brabham",
        Years = new int[] { 1966, 1967 } });
    teams.Add(new Team() { Name = "Matra",
        Years = new int[] { 1969 } });
    teams.Add(new Team() { Name = "Tyrrell",
        Years = new int[] { 1971 } });
    teams.Add(new Team() { Name = "McLaren",
        Years = new int[] { 1974, 1984, 1985,
            1988, 1989, 1990, 1991, 1998 } });
    teams.Add(new Team() { Name = "Williams",
        Years = new int[] { 1980, 1981, 1986,
            1987, 1992, 1993, 1994, 1996, 1997 } });
    teams.Add(new Team() { Name = "Benetton",
        Years = new int[] { 1995 } });
    teams.Add(new Team() { Name = "Renault",
        Years = new int[] { 2005, 2006 } });
    return teams;
}
```

现在进入对象查询的核心。首先，需要用 GetChampions()静态方法获得对象列表。该列表放在泛型类 List<T>中。这个类的 FindAll()方法接收一个 Predicate<T>委托，该委托可以实现为一个匿名方法。只返回 Country 属性设置为 Brazil 的赛手。接着，用 Sort()方法给得到的列表排序。不应按照 Lastname 属性排序，因为这是 Racer 类的默认排序方式，而可以传送一个类型为 Comparison<T>的委托，该委托也实现为一个匿名方法，来比较夺冠次数。使用 r2 对象，与 r1 比较，根据需要进行降序排序。foreach 语句最终迭代已排序的集合中的所有 Racer 对象。

```
private static void ObjectQuery()
{
    List<Racer> racers = new List<Racer>(Formula1.GetChampions());
    List<Racer> brazilRacers = racers.FindAll(
        delegate(Racer r)
    {
        return r.Country == "Brazil";
    });
    brazilRacers.Sort(
        delegate(Racer r1, Racer r2)
    {
        return r2.Wins.CompareTo(r1.Wins);
    });
    foreach (Racer r in brazilRacers)
    {
        Console.WriteLine("{0}: {1}", r);
    }
}
```

下面的列表显示了来自巴西的所有冠军，并按照夺冠次数排序：

Ayrton Senna, Brazil; starts: 161, wins: 41

Nelson Piquet, Brazil; starts: 204, wins: 23

Emerson Fittipaldi, Brazil; starts: 143, wins: 14

提示：

排序和过滤对象列表的内容详见第 10 章。

在前面的例子中，使用了 List<T>类的 FindAll() 和 Sort() 方法。使用任何集合都可以获得这两个方法的功能，而不仅仅是 List<T>。这需要使用扩展方法。扩展方法是 C# 3.0 的新增特性，这也是上述例子迈向 LINQ 的第一个变化。

### 11.1.2 扩展方法

扩展方法可以将方法写入最初没有提供该方法的类中。还可以把方法添加到实现某个接口的任何类中，这样多个类就可以使用相同的实现代码。

例如，String 类没有 Foo()方法。String 类是密封的，所以不能从这个类中继承。但可以执行一个扩展方法，如下所示：

```
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo invoked for {0}", s);
    }
}
```

扩展方法在静态类中声明，定义为一个静态方法，其中第一个参数定义了它扩展的类型。Foo()方法扩展了 String 类，因为它的第一个参数定义了 String 类型。为了区分扩展方法和一般的静态方法，扩展方法还需要给第一个参数使用 this 关键字。

现在就可以使用带 string 类型的 Foo()方法了：

```
string s = "Hello";
s.Foo();
```

结果在控制台上显示 Foo invoked for Hello，因为 Hello 是传送给 Foo()方法的字符串。

也许这看起来违反了面向对象的规则，因为给一个类型定义了新方法，但没有改变该类型。但实际上并非如此。扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种新语法。对于字符串，可以用如下方式调用 Foo()方法，获得相同的结果：

```
string s = "Hello";
StringExtension.Foo(s);
```

要调用静态方法，应在类名的后面加上方法名。扩展方法是调用静态方法的另一种方式。不必提供定义了静态方法的类名，相反，调用静态方法是因为它带的参数类型。只需导入包含该类的命名空间，就可以将 Foo()扩展方法放在 String 类的作用域中。

定义 LINQ 扩展方法的一个类是 System.Linq 命名空间中的 Enumerable。只需导入这个命名空间，就打开了这个类的扩展方法的作用域。下面列出了 Where() 扩展方法的实现代码。Where() 扩展方法的第一个参数包含了 this 关键字，其类型是 I Enumerable<T>。这样，Where 方法就可以用于实现了 I Enumerable<T> 的每个类型。例如数组和 List<T> 实现了 I Enumerable<T>。第二个参数是一个 Func<T, bool> 委托，它引用了一个返回布尔值、参数类型为 T 的方法。这个谓词在实现代码中调用，检查 I Enumerable<T> 源中的项是否应放在目标集合中。如果委托引用了该方法，yield return 语句就将源中的项返回给目标。

```
public static I Enumerable<T> Where<T>(this I Enumerable<T> source,  
                                         Func<T, bool> predicate)  
{  
    foreach (T item in source)  
    {  
        if (predicate(item))  
            yield return item;  
    }  
}
```

因为 Where() 实现为一个泛型方法，所以可以用于包含在集合中的任意类型。实现了 I Enumerable<T> 的集合都支持它。

提示：

这里的扩展方法在程序集 System.Core 的 System.Linq 命名空间中定义。

现在就可以使用 Enumerable 类中的扩展方法 Where()、OrderbyDescending() 和 Select() 了。这些方法都返回 I Enumerable<TSource>，所以可以使用前面的结果依次调用这些方法。通过扩展方法的参数，使用定义了委托参数的实现代码的匿名方法。

```
private static void ExtensionMethods()  
{  
    List < Racer > champions =  
        new List < Racer > (  
            Formula1.GetChampions());  
  
    I Enumerable < Racer > brazilChampions =  
        champions.Where(  
            delegate(Racer r)
```

```
{  
    return r.Country == "Brazil";  
}) .OrderByDescending(  
delegate(Racer r)  
{  
    return r.Wins;  
}) .Select(  
delegate(Racer r)  
{  
    return r;  
});  
  
foreach (Racer r in brazilChampions)  
{  
    Console.WriteLine("{0:A}", r);  
}  
}
```

### 11.1.3 表达式

C# 3.0 给匿名方法提供了一个新的语法—— 表达式。除了把匿名方法传送给 Where()、 OrderbyDescending() 和 Select() 方法之外，还可以使用 表达式。

这里把上面的例子改为使用 表达式。现在语法比较短，也更容易理解了，因为删除了 return 语句、参数类型和花括号。

表达式参见第 7 章。 表达式在 LINQ 中非常重要，所以下面复习一下该语法。详细信息可参见第 7 章。

比较 表达式和匿名委托，会发现许多类似之处。 运算符=>的左边是参数，不需要添加参数类型，因为它们是由编译器解析的。 运算符的右边定义了执行代码。在匿名方法中，需要花括号和 return 语句。在表达式中，不需要这些语法元素，因为它们是由编译器处理的。如果 运算符右边有多个语句，也可以使用花括号和 return 语句。

```
private static void LambdaExpressions()  
{
```

```
IEnumerable < Racer > brazilChampions =  
    Formula1.GetChampions().  
    Where(r => r.Country == "Brazil").  
    OrderByDescending(r => r.Wins).  
    Select(r => r);  
  
    foreach (Racer r in brazilChampions)  
    {  
        Console.WriteLine("{0:A}", r);  
    }  
}
```

提示：

使用无参的 表达式时，return语句和花括号是可选的。但在 表达式中仍可以使用这些语言结构。详见第 7章。

#### 11.1.4 LINQ查询

最后一个需要修改的是用新的 LINQ 查询记号定义查询。语句 from r in Formula1. GetChampions() Where r.Country == “Brazil” orderby r.Wins descending select r;是一个 LINQ 查询，子句 from、 where、 orderby、 descending 和 select 都是这个查询中的预定义关键字。编译器把这些子句映射到扩展方法上。这里的语法是使用扩展方法 Where()、 Orderby- Descending()和 Select()。 表达式传送给参数。

Where r.Country == "Brazil" is converted to Where(r => r.Country == "Brazil")  
.orderby r.Wins descending is converted to OrderByDescending(r => r.Wins).

```
private static void LinqQuery()  
{  
    var query = from r in Formula1.GetChampions()  
               where r.Country == "Brazil"  
               orderby r.Wins descending  
               select r;  
  
    foreach (Racer r in query)  
    {
```

```
        Console.WriteLine("{0:A}", r);
    }
}
```

提示：

LINQ查询是 C#语言中的一个简化查询记号。编译器编译查询表达式，调用扩展方法。查询表达式只是 C#中的一个语法，但不需要修改底层的 IL代码。

查询表达式必须以 from 子句开头，以 select 或 group 子句结束。在这两个子句之间，可以使用 where、 orderby、join、let 和其他 from 子句。

注意，变量 query 只指定了 LINQ 查询。该查询不是通过这个赋值语句执行的，只要使用 foreach 循环访问查询，该查询就会执行。详见后面的内容。

在前面的示例中，学习了 C# 3.0 语言特性，以及它们与 LINQ 查询的关系。下面深入探讨 LINQ 的特性。

### 11.1.5 推迟查询的执行

在运行期间定义查询表达式时，查询就不会运行。查询会在迭代数据项时运行。

再看看扩展方法 Where()。它使用 yield return 语句返回谓词为 true 的元素。因为使用了 yield return 语句，所以编译器会创建一个枚举器，在访问枚举中的项后，就返回它们。

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                         Func<T, bool> predicate)
{
    foreach (T item in source)
        if (predicate(item))
            yield return item;
}
```

这是一个非常有趣、也非常重要的结果。在下面的例子中，创建了一个 String 元素集合，用名称 arr 填充它。接着定义一个查询，从集合中找出以字母 J 开头的所有名称。集合也应是排好序的。在定义查询时，不会执行迭代。相反，迭代在 foreach 语句中进行，迭代所有的项。集合中只有一个元素 Juan 满足 where 表达式的要求，即以字母 J 开头。迭代完成后，将 Juan 写入控制台。之后在集合中添加四个新名称，再次执行迭代。

```
List<string> names = new List<string>
```

```
{ "Nino", "Alberto", "Juan", "Mike", "Phil" };

var namesWithJ = from n in names
    where n.StartsWith("J")
    orderby n
    select n;

Console.WriteLine("First iteration");

foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}

Console.WriteLine();

arr.Add("John");
arr.Add("Jim");
arr.Add("Jack");
arr.Add("Denny");

Console.WriteLine("Second iteration");

foreach (string name in namesWithJ)
{
    Console.WriteLine(name);
}
```

因为迭代在查询定义时不会执行，而是在执行每个 foreach 语句时执行，所以可以看到其中的变化，如应用程序的结果所示：

First iteration

Juan

Second iteration

Jack

Jim

John

Juan

当然，还必须注意，每次在迭代中使用查询时，都会调用扩展方法。在大多数情况下，这是非常有效的，因为我们可以检测出源数据中的变化。但是在一些情况下，这是不可行的。调用扩展方法 `ToArrayList()`、`ToEnumerable()`、`ToList()` 等可以改变这个操作。在下面的示例中，`ToList` 迭代集合，返回一个实现了 `IList<string>` 的集合。之后对返回的列表迭代两次，在这个过程中，数据源得到了新名称。

```
List<string> names = new List<string>
{
    { "Nino", "Alberto", "Juan", "Mike", "Phil" };
    IList<string> namesWithJ = (from n in names
                                where n.StartsWith("J")
                                orderby n
                                select n).ToList();
    Console.WriteLine("First iteration");
    foreach (string name in namesWithJ)
    {
        Console.WriteLine(name);
    }
    Console.WriteLine();
    names.Add("John");
    names.Add("Jim");
    names.Add("Jack");
    names.Add("Denny");
    Console.WriteLine("Second iteration");
    foreach (string name in namesWithJ)
    {
        Console.WriteLine(name);
    }
}
```

在结果中可以看到，两次迭代中输出保持不变，但集合中的值改变了：

First iteration

Juan

Second iteration

Juan

## 11.2 标准的查询操作符

Where OrderByDescending 和 Select 只是 LINQ 的几个查询操作符。 LINQ 查询为最常用的操作符定义了一个声明语法。还有许多标准查询操作符。

表 11-1列出了 LINQ 定义的标准查询操作符。

表 11-1

标准查询操作符	说 明
Where OfType<TResult>	过滤操作符定义了返回元素的条件。在 Where 查询操作符中，可以使用谓词，例如 表达式定义的谓词，来返回布尔值。 OfType<TResult> 根据类型过滤元素，只返回 TResult 类型的元素
Select 和 SelectMany	投射操作符用于把对象转换为另一个类型的对象。Select 和 SelectMany 定义了根据选择器函数选择结果值的投射
OrderBy , ThenBy OrderByDescending ThenByDescending Reverse	排序操作符改变所返回的元素的顺序。OrderBy 按升序排序，OrderByDescending 按降序排序。如果第一次排序的结果很类似，就可以使用 ThenBy 和 ThenByDescending 操作符进行第二次排序。Reverse 反转集合中元素的顺序

(续表 )

标准查询操作符	说 明
Join , GroupJoin	连接运算符用于合并不直接相关的集合。使用 Join 操作符，可以根据键选择器函数连接两个集合，这类似于 SQL 中的 JOIN。GroupJoin 操作符连接两个集合，组合其结果
GroupBy	组合运算符把数据放在组中。GroupBy 操作符组合有公共键的元素
Any , All , Contains	如果元素序列满足指定的条件，量词操作符就返回布尔值。Any , All 和 Contains 都是量词操作符。Any 确定集合中是否有满足谓词函数的元素；All 确定集合中的所有元素是否都满足谓词函数；Contains 检查某个元素是否在集合中。这些操作符都返回一个布尔值
Take , Skip, TakeWhile SkipWhile	分区操作符返回集合的一个子集。Take Skip TakeWhile 和 SkipWhile 都是分区操作符。使用它们可以得到部分结果。使用 Take 必须指定要从集合中提取的元素个数；Skip 跳过指定的元素个数，提取其他元素，TakeWhile 提取条件为真的元素
Distinct , Union Intersect , Except	Set 操作符返回一个集合。Distinct 从集合中删除重复的元素。除了 Distinct 之外，其他 Set 操作符都需要两个集合。Union 返回出现在其中一个集合中的元素。Intersect 返回两个集合中都有的元素。Except 返回只出现在一个集合中的元素
First FirstOrDefault Last LastOrDefault ElementAt ElementAtOrDefault Single SingleOrDefault	这些元素操作符仅返回一个元素。First 返回第一个满足条件的元素。FirstOrDefault 类似于 First，但如果找不到满足条件的元素，就返回类型的默认值。Last 返回最后一个满足条件的元素。ElementAt 指定了要返回的元素的位置。Single 只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异常
Count , Sum , Min , Max , Average ,	合计操作符计算集合的一个值。利用这些合计操作符，可以计算所有值的总和、元素的个数、值最大和最小的元素，平均值等

Aggregate	
ToArray	这些转换操作符将集合转换为数组、 I Enumerable、 IList、 IDictionary 等
ToEnumerable	
ToList	
ToDictionary	
toType<T>	
Empty, Range, Repeat	这些生成操作符返回一个新集合。使用 Empty，集合是空的，Range 返回一系列数字，Repeat 返回一个始终重复一个值的集合

下面是使用这些操作符的一些例子。

### 11.2.1 过滤

下面介绍一些查询的示例。

使用 Where 子句，可以合并多个表达式。例如，找出赢得至少 15 场比赛的巴西和奥地利赛手。传递给 where 子句的表达式的结果类型应是 bool：

```
var racers = from r in Formula1.GetChampions()
    where r.Wins > 15 &&
        (r.Country == "Brazil" ||
        r.Country == "Austria")
    select r;

foreach (var r in racers)
{
    Console.WriteLine("{0:A}", r);
}
```

用这个 LINQ 查询启动程序，会返回 Niki Lauda、Nelson Piquet 和 Ayrton Senna，如下：

Niki Lauda, Austria, Starts: 173, Wins: 25

Nelson Piquet, Brazil, Starts: 204, Wins: 23

Ayrton Senna, Brazil, Starts: 161, Wins: 41

并不是所有的查询都可以用 LINQ 查询完成。也不是所有的扩展方法都映射到 LINQ 查询子句上。高级查询需要使用扩展方法。为了更好地理解带扩展方法的复杂查询，最好看看简单的查询是如何映射的。使用扩展方法 Where() 和 Select()，会生成与前面 LINQ 查询非常类似的结果：

```
var racers = Formula1.GetChampions().  
    Where(r => r.Wins > 15 & &  
          (r.Country == "Brazil" ||  
           r.Country == "Austria")).  
    Select(r => r);
```

### 11.2.2 用索引来过滤

不能使用 LINQ查询的一个例子是 `Where()`方法的重载。在 `Where()`方法的重载中，可以传送第二个参数——索引。索引是过滤器返回的每个结果的计数器。可以在表达式中使用这个索引，执行基于索引的计算。下面的代码由 `Where()`扩展方法调用，它使用索引返回姓氏以 A开头、索引为偶数的赛手：

```
var racers = Formula1.GetChampions().  
    Where((r, index) =>  
          r.LastName.StartsWith("A") & &  
          index % 2 != 0);  
  
foreach (var r in racers)  
{  
    Console.WriteLine("{0}:{1}", r);  
}
```

姓氏以 A开头的赛手有 Alberto Ascari、 Mario Andretti和 Fernando Alonso。 Mario Andretti的索引是奇数，所以他不在结果中：

```
Alberto Ascari, Italy; starts: 32, wins: 10  
Fernando Alonso, Spain; starts: 105, wins: 19
```

### 11.2.3 类型过滤

为了进行基于类型的过滤，可以使用 `OfType()`扩展方法。这里数组数据包含 `string`和 `int`对象。使用 `OfType()`扩展方法，把 `string`类传送给泛型参数，就从集合中返回字符串：

```
object[] data = { "one", 2, 3, "four", "five", 6 };  
  
var query = data.OfType<string>();  
  
foreach (var s in query)
```

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

运行这段代码，就会显示字符串 one four和 five

```
one  
four  
five
```

#### 11.2.4 复合的 from子句

如果需要根据对象的一个成员进行过滤，而该成员本身是一个系列，就可以使用复合的 from子句。Racer类定义了一个属性 Cars，Cars是一个字符串数组。要过滤驾驶 Ferrari的所有冠军，可以使用如下所示的 LINQ查询。第一个 from子句访问从 Formula1.GetChampions()返回的 Racer对象，第二个 from子句访问 Racer类的属性 Cars，返回所有 string类型的赛车。接着在 Where子句中使用这些赛车过滤驾驶 Ferrari的所有冠军。

```
var ferrariDrivers = from r in  
    Formula1.GetChampions()  
    from c in r.Cars  
    where c == "Ferrari"  
    orderby r.LastName  
    select r.FirstName + " "  
    + r.LastName;
```

这个查询的结果显示了驾驶 Ferrari的所有一级方程式冠军：

```
Alberto Ascari  
Juan Manuel Fangio  
Mike Hawthorn  
Phil Hill  
Niki Lauda  
Jody Scheckter  
Michael Schumacher  
John Surtees
```

C#编译器把复合的 from子句和 LINQ查询转换为 SelectMany()扩展方法。SelectMany()可用于迭代序列的序列。示例中 SelectMany()方法的重载版本如下所示：

```
public static IEnumerable < TResult >
```

```
SelectMany < TSource, TCollection, TResult > (  
    this IEnumerable < TSource > source,  
    Func < TSource, IEnumerable < TCollection > > collectionSelector,  
    Func < TSource, TCollection, TResult >  
    resultSelector);
```

第一个参数是隐式参数，从 `GetChampions()`方法中接收 `Racer`对象序列。第二个参数是 `collectionSelector` 委托，它定义了内部序列。在 $\lambda$ 表达式 `r=>r.Cars` 中，应返回赛车集合。第三个参数是一个委托，现在为每个赛车调用该委托，接收 `Racer`和 `Car`对象。 $\lambda$ 表达式创建了一个匿名类型，它带 `Racer`和 `Car` 属性。这个 `SelectMany()`方法的结果是摊平了赛手和赛车的层次结构，为每辆赛车返回匿名类型的一个新对象集合。

这个新集合传送给 `Where()`方法，过滤出驾驶 `Ferrari`的赛手。最后，调用 `OrderBy()`和 `Select()`方法：

```
var ferrariDrivers = Formula1.GetChampions().  
    SelectMany(  
        r => r.Cars,  
        (r, c) => new { Racer = r, Car = c }).  
    Where(r => r.Car == "Ferrari").  
    OrderBy(r => r.Racer.LastName).  
    Select(r => r.Racer.FirstName + " " +  
        r.Racer.LastName);
```

把 `SelectMany()`泛型方法解析为这里使用的类型，所解析的类型如下所示。在这个例子中，数据源是 `Racer`类型，所过滤的集合是一个 `string`数组，当然所返回的匿名类型的名称是未知的，这里显示为 `TResult`：

```
public static IEnumerable < TResult >  
    SelectMany < Racer, string, TResult > (  
        this IEnumerable < Racer > source,  
        Func < Racer, IEnumerable < string > > collectionSelector,  
        Func < Racer, string, TResult > resultSelector);
```

查询仅从 LINQ查询转换为扩展方法，所以结果与前面的相同。

## 11.2.5 排序

要对序列排序，前面使用了 orderby子句。下面复习一下前面使用 orderby descending子句的例子。其中赛手按照赢得比赛的次数进行降序排序，赢得比赛的次数是用关键字选择器指定的：

```
var racers = from r in Formula1.GetChampions()
    where r.Country == "Brazil"
    orderby r.Wins descending
    select r;
```

orderby子句解析为 OrderBy()方法， orderby descending子句解析为 OrderBy Descending()方法：

```
var racers = Formula1.GetChampions().
    Where(r => r.Country == "Brazil").
    OrderByDescending(r => r.Wins).
    Select(r => r);
```

OrderBy()和 OrderByDescending ()方法返回 IOrderEnumerable<TSource> 这个接口派生于接口 IEnumerable<TSource>，但包含一个额外的方法 CreateOrderedEnumerable- <TSource>()。这个方法用于进一步给序列排序。如果根据关键字选择器来排序，两项的顺序相同，就可以使用 ThenBy()和 ThenByDescending ()方法继续排序。这两个方法需要 IOrderEnumerable<TSource>才能工作，但也返回这个接口。所以，可以添加任意多个 ThenBy()和 ThenByDescending ()方法，对集合排序。

使用 LINQ查询时，只需把所有用于排序的不同关键字 (用逗号分隔开)添加到 orderby子句中。这里，所有的赛手先按照国家排序，再按照姓氏排序，最后按照名字排序。添加到 LINQ查询结果中的 Take()扩展方法用于提取前 10个结果：

```
var racers = (from r in
    Formula1.GetChampions()
    orderby r.Country, r.LastName,
    r.FirstName
    select r).Take(10);
```

排序后的结果如下：

Argentina: Fangio, Juan Manuel

Australia: Brabham, Jack

Australia: Jones, Alan

Austria: Lauda, Niki

Austria: Rindt, Jochen

Brazil: Fittipaldi, Emerson

Brazil: Piquet, Nelson

Brazil: Senna, Ayrton

Canada: Villeneuve, Jacques

Finland: Häkkinen, Mika

使用 OrderBy() 和 ThenBy() 方法可以执行相同的操作：

```
var racers = Formula1.GetChampions().  
    OrderBy(r => r.Country).  
    ThenBy(r => r.LastName).  
    ThenBy(r => r.FirstName).  
    Take(10);
```

### 11.2.6 分组

要根据一个关键字值对查询结果分组，可以使用 group 子句。现在一级方程式冠军应按照国家分组，并列出一个国家的冠军数。子句 group r by r.Country into g 根据 Country 属性组合所有的赛手，并定义一个新的标识符 g，它以后用于访问分组的结果信息。group 子句的结果根据应用到分组结果上的扩展方法 Count() 来排序，如果冠军数相同，就根据关键字来排序，该关键字是国家，因为这是分组所使用的关键字。where 子句根据至少有两项的分组来过滤，select 子句创建一个带 Country 和 Count 属性的匿名类型。

```
var countries = from r in  
    Formula1.GetChampions()  
    group r by r.Country into g  
    orderby g.Count() descending, g.Key  
    where g.Count() >= 2  
    select new { Country = g.Key,
```

```
Count = g.Count() };

foreach (var item in countries)

{

    Console.WriteLine("{0, -10} {1}",

        item.Country, item.Count);

}
```

结果显示了带 Country和 Count属性的对象集合：

UK	9
Brazil	3
Australia	2
Austria	2
Finland	2
Italy	2
USA	2

要用扩展方法执行相同的操作，应把 groupby子句解析为 GroupBy()方法。在 GroupBy()方法的声明中，注意它返回实现了 IGrouping接口的对象枚举。IGrouping接口定义了 Key属性，所以在定义了对这个方法的调用后，可以访问分组的关键字：

```
public static IEnumerable < IGrouping < TKey, TSource > >

GroupBy < TSource, TKey > (

    this IEnumerable < TSource > source,

    Func < TSource, TKey > keySelector);
```

子句 group r by r.Country into g解析为 GroupBy(r => r.Country)，返回分组系列。分组系列首先用 OrderByDescending()方法排序，再用 ThenBy()方法排序。接着调用 Where()和 Select()方法。

```
var countries = Formula1.GetChampions().

    GroupBy(r => r.Country).

    OrderByDescending(g => g.Count()) .

    ThenBy(g => g.Key) .

    Where(g => g.Count() >= 2) .
```

```
Select(g => new { Country = g.Key,
    Count = g.Count() });

```

### 11.2.7 对嵌套的对象分组

如果分组的对象应包含嵌套的对象，就可以改变 select子句创建的匿名类型。在下面的例子中，所创建的国家不仅应包含国家名和赛手数量这两个属性，还应包含赛手名序列。这个序列用一个赋予 Racers 属性的 from/in 内部子句指定，内部的 from 子句使用分组标识符 g 获得该分组中的所有赛手，用姓氏对它们排序，再根据姓名创建一个新字符串：

```
var countries = from r in
    Formula1.GetChampions()
    group r by r.Country into g
    orderby g.Count() descending, g.Key
    where g.Count() >= 2
    select new
    {
        Country = g.Key,
        Count = g.Count(),
        Racers = from r1 in g
            orderby r1.LastName
            select r1.FirstName + " "
            + r1.LastName
    };
foreach (var item in countries)
{
    Console.WriteLine("{0, -10} {1}",
        item.Country, item.Count);
    foreach (var name in item.Racers)
    {
        Console.Write("{0}; ", name);
    }
}
```

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

结果应列出某个国家的所有冠军：

UK 9

Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill; Damon Hill;  
James Hunt; Nigel Mansell; Jackie Stewart; John Surtees;

Brazil 3

Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;

Australia 2

Jack Brabham; Alan Jones;

Austria 2

Niki Lauda; Jochen Rindt;

Finland 2

Mika Häkkinen; Keke Rosberg;

Italy 2

Alberto Ascari; Nino Farina;

USA 2

Mario Andretti; Phil Hill;

## 11.2.8 连接

使用 join子句可以根据特定的条件合并两个数据源，但之前要获得两个要连接的列表。在一级方程式比赛中，设有赛手冠军和制造商冠军。赛手从 GetChampions()方法中返回，制造商从 GetConstructorChampions()方法中返回。现在要获得一个年份列表，列出每年的赛手和制造商冠军。

为此，先定义两个查询，用于查询赛手和制造商团队：

```
var racers = from r in Formula1.GetChampions()  
            from y in r.Years  
            where y > 2003  
            select new  
            {  
                Year = y,  
                Name = r.FirstName + " " +  
                       r.LastName  
            };  
  
var teams = from t in
```

```
Formula1.GetConstructorChampions()  
from y in t.Years  
where y > 2003  
select new { Year = y,  
            Name = t.Name };
```

有了这两个查询，再通过子句 `join t in teams on r.Year equals t.Year`，根据赛手获得冠军的年份和制造商获得冠军的年份进行连接。`select`子句定义了一个新的匿名类型，它包含 `Year`、`Racer` 和 `Team` 属性。

```
var racersAndTeams =  
    from r in racers  
    join t in teams on r.Year equals t.Year  
    select new  
    {  
        Year = r.Year,  
        Racer = r.Name,  
        Team = t.Name  
    };  
  
Console.WriteLine("Year Champion " +  
    "Constructor Title");  
  
foreach (var item in racersAndTeams)  
{  
    Console.WriteLine("{0}: {1,-20} {2}",  
        item.Year, item.Racer, item.Team);  
}
```

当然，也可以把它们合并为一个 LINQ 查询，但这只是一种尝试：

```
int year = 2003;  
  
var racersAndTeams =  
    from r in
```

```
from r1 in Formula1.GetChampions()
from yr in r1.Years
where yr > year
select new
{
    Year = yr,
    Name = r1.FirstName + " " +
    r1.LastName
}
join t in
    from t1 in
        Formula1.GetConstructorChampions()
    from yt in t1.Years
    where yt > year
    select new { Year = yt,
        Name = t1.Name }
on r.Year equals t.Year
select new
{
    Year = r.Year,
    Racer = r.Name,
    Team = t.Name
};
```

结果显示了匿名类型中的数据：

Year	Champion	Constructor	Title
2004	Michael Schumacher	Ferrari	
2005	Fernando Alonso	Renault	
2006	Fernando Alonso	Renault	
2007	Kimi Räikkönen	Ferrari	

### 11.2.9 设置操作

扩展方法 Distinct()、Union()、Intersect()和 Except()都是设置操作。下面创建一个驾驶 Ferrari 的一级方程式冠军序列和驾驶 McLaren 的一级方程式冠军序列，然后确定是否有驾驶 Ferrari 和 McLaren 的冠军。当然，这里可以使用 Intersect() 扩展方法。

首先获得所有驾驶 Ferrari 的冠军。这只是一个简单的 LINQ 查询，其中使用复合的 from 子句访问属性 Cars，返回一个字符串对象序列。

```
var ferrariDrivers = from r in
    Formula1.GetChampions()
    from c in r.Cars
    where c == "Ferrari"
    orderby r.LastName
    select r;
```

现在建立另一个相同的查询，但 where 子句的参数不同，以获得所有驾驶 McLaren 的冠军。最好不要再次编写相同的查询。而可以创建一个方法，给它传送参数 car：

```
private static IEnumerable < Racer > GetRacersByCar(string car)
{
    return from r in Formula1.GetChampions()
        from c in r.Cars
        where c == car
        orderby r.LastName
        select r;
}
```

但是，因为该方法不需要在其他地方使用，所以应定义一个委托类型的变量来保存 LINQ 查询。变量 racerByCar 必须是一个委托类型，它需要一个字符串参数，返回 IEnumerable < Racer >，类似于前面实现的方法。为此，定义了几个泛型委托 Func<>，所以不需要声明自己的委托。把一个 λ 表达式赋予变量 racerByCar，λ 表达式的左边定义了一个 car 变量，其类型是 Func 委托的第一个泛型参数（字符串）。右边定义了 LINQ 查询，它使用该参数和 where 子句：

```
Func < string, IEnumerable < Racer > > racersByCar =
```

```
Car => from r in Formula1.GetChampions()
        from c in r.Cars
        where c == car
        orderby r.LastName
        select r;
```

现在可以使用 `Intersect()` 扩展方法，获得驾驶 Ferrari 和 McLaren 的所有冠军：

```
Console.WriteLine("World champion with " +
    "Ferrari and McLaren");
foreach (var racer in racersByCar("Ferrari").
    Intersect(racersByCar("McLaren")))
{
    Console.WriteLine(racer);
}
```

结果只有一个赛手 Niki Lauda：

```
World champion with Ferrari and McLaren
Niki Lauda
```

### 11.2.10 分区

扩展方法 `Take()` 和 `Skip()` 等的分区操作可用于分页，例如显示 5x 5 个赛手。

在下面的 LINQ 查询中，扩展方法 `Take()` 和 `Skip()` 添加到查询的最后。`Skip()` 方法先忽略根据页面的大小和实际的页数计算出的项数，再使用方法 `Take()` 根据页面的大小提取一定数量的项：

```
int pageSize = 5;
int numberPages = (int)Math.Ceiling(
    Formula1.GetChampions().Count() /
    (double)pageSize);

for (int page = 0; page < numberPages; page++)
{
    Console.WriteLine("Page {0}", page);

    var racers =
        (from r in Formula1.GetChampions()
```

```
orderby r.LastName
select r.FirstName + " " + r.LastName).
Skip(page * pageSize).Take(pageSize);

foreach (var name in racers)
{
    Console.WriteLine(name);
}
Console.WriteLine();

}
```

下面输出了前 3页：

Page 0

Fernando Alonso

Mario Andretti

Alberto Ascari

Jack Brabham

Jim Clark

Page 1

Juan Manuel Fangio

Nino Farina

Emerson Fittipaldi

Mika Häkkinen

Mike Hawthorn

Page 2

Phil Hill

Graham Hill

Damon Hill

Denny Hulme

James Hunt

分页在 Windows或 Web应用程序中非常有用，可以只给用户显示一部分数据。

提示：

这个分页机制的一个要点是，因为查询会在每个页面上执行，所以改变底层的数据会影响结果。在继续执行分页操作时，会显示新对象。根据不同的情况，这对于应用程序而言可能是一个优点。如果这个操作是不需要的，就可以只对原来的数据源分页，然后使用映射到原数据上的高速缓存。

使用 `TakeWhile()` 和 `SkipWhile()` 方法，还可以传送一个谓词，根据谓词的结果提取或跳过某些项。

#### 11.2.11 合计操作符

合计操作符如 `Count()`、`Sum()`、`Min()`、`Max()`、`Average()` 和 `Aggregate()`，不返回一个序列，而返回一个值。

`Count()` 扩展方法返回集合中的项数。下面的 `Count()` 方法应用于 `Racer` 的 `Years` 属性，过滤赛手，只返回获得冠军次数超过 3 次的赛手：

```
var query = from r in Formula1.GetChampions()
    where r.Years.Count() > 3
    orderby r.Years.Count() descending
    select new
    {
        Name = r.FirstName + " " +
        r.LastName,
        TimesChampion = r.Years.Count()
    };
foreach (var r in query)
{
    Console.WriteLine("{0} {1}", r.Name, r.TimesChampion);
}
```

结果如下：

```
Michael Schumacher 7
Juan Manuel Fangio 5
Alain Prost 4
```

Sum()方法汇总序列中的所有数字，返回这些数字的和。下面的 Sum()用于计算一个国家赢得比赛的总次数。首先根据国家对赛手分组，再在新创建的匿名类型中，给 Wins属性赋予某个国家赢得比赛的总次数。

```
var countries =  
    (from c in  
        from r in Formula1.GetChampions()  
        group r by r.Country into c  
        select new  
    {  
        Country = c.Key,  
        Wins = (from r1 in c  
                select r1.Wins).Sum()  
    }  
    orderby c.Wins descending, c.Country  
    select c).Take(5);  
  
foreach (var country in countries)  
{  
    Console.WriteLine("{0} {1}",  
        country.Country, country.Wins);  
}
```

根据获得一级方程式冠军的次数，最成功的国家是：

UK 138

Germany 91

Brazil 78

France 51

Finland 40

方法 Min()、Max()、Average()和 Aggregate()的使用方式与 Count()和 Sum()相同。Min()返回集合中的最小值，Max()返回集合中的最大值，Average()计算集合中的平均值。对于 Aggregate()方法，可以传送一个λ表达式，对所有的值进行汇总。

### 11.2.12 转换

本章前面提到，查询可以推迟到访问数据项时再执行。在迭代中使用查询，查询会执行。而使用转换操作符会立即执行查询，把结果放在数组、列表或字典中。

在下面的例子中，调用 `ToList()` 扩展方法，立即执行查询，把结果放在 `List<T>` 中：

```
List < Racer > racers =  
  
(from r in Formula1.GetChampions()  
  
where r.Starts > 150  
  
orderby r.Starts descending  
  
select r).ToList();  
  
foreach (var racer in racers)  
  
{  
  
    Console.WriteLine("{0} {0:S}", racer);  
  
}
```

把返回的对象放在列表中并没有这么简单。例如，对于集合中从赛车到赛手的快速访问，可以使用新类 `Lookup< TKey, TElement >`

提示：

`Dictionary< TKey, TValue >` 只支持一个键对应一个值。在 `System.Linq` 命名空间的类 `Lookup< TKey, TElement >` 中，一个键可以对应多个值。这些类详见第 10 章。

使用复合的 `from` 查询，可以摊平赛手和赛车序列，创建带有 `Car` 和 `Racer` 属性的匿名类型。在返回的 `Lookup` 对象中，键的类型应是表示汽车的 `string`，值的类型应是 `Racer`。为了进行这个选择，可以给 `ToLookup()` 方法的一个重载版本传送一个键和一个元素选择器。键选择器表示 `Car` 属性，元素选择器表示 `Racer` 属性。

```
ILookup < string, Racer > racers =  
  
(from r in Formula1.GetChampions()  
  
from c in r.Cars
```

```
select new
{
    Car = c,
    Racer = r
}).ToLookup(cr => cr.Car, cr => cr.Racer);

if (racers.Contains("Williams"))
{
    foreach (var williamsRacer in
        racers["Williams"])
    {
        Console.WriteLine(williamsRacer);
    }
}
```

用 Lookup类的索引器访问的所有 Williams冠军如下：

Alan Jones

Keke Rosberg

Nigel Mansell

Alain Prost

Damon Hill

Jacques Villeneuve

如果需要在未类型化的集合上使用 LINQ查询，例如 ArrayList，就可以使用 Cast()方法。在下面的例子中，基于 Object类型的 ArrayList集合用 Racer对象填充。为了定义强类型化的查询，可以使用 Cast()方法。

```
System.Collections.ArrayList list =
    new System.Collections.ArrayList(
        Formula1.GetChampions() as
        System.Collections.ICollection);
var query = from r in list.Cast < Racer > ()
```

```
where r.Country = "USA"
orderby r.Wins descending
select r;

foreach (var racer in query)
{
    Console.WriteLine("{0:A}", racer);
}
```

### 11.2.13 生成操作符

生成操作符 Range()、Empty() 和 Repeat() 不是扩展方法，而是返回序列的正常静态方法。在 LINQ to Objects 中，这些方法可用于 Enumerable 类。

有时需要填充一个范围的数字，此时就应使用 Range() 方法。这个方法把第一个参数作为起始值，把第二个参数作为要填充的项数。

```
var values = Enumerable.Range(1, 20);
foreach (var item in values)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

当然，结果如下所示：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

提示：

Range() 方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查询，返回一个 RangeEnumerator，其中只有一个 yield return 语句，来递增值。

可以把该结果与其他扩展方法合并起来，获得另一个结果，例如使用 Select() 扩展方法：

```
var values = Enumerable.Range(1, 20).
    Select(n => n * 3);
```

Empty()方法返回一个不返回值的迭代器，它可用于参数需要一个集合，且可以给参数传送空集合的情形。

Repeat()方法返回一个迭代器，该迭代器把同一个值重复特定的次数。

### 11.3 表达式树

在 LINQ to Objects 中，扩展方法需要将一个委托类型作为参数，这样就可以将 表达式赋予参数。表达式也可以赋予 Expression<T>类型的参数。Expression<T>类型指定，来自于 表达式的表达式树存储在程序集中。这样，就可以在运行期间分析表达式，并进行优化，以便于查询数据源。

下面看看一个前面使用的查询表达式：

```
var brazilRacers = from r in racers
    where r.Country == "Brazil"
    orderby r.Wins
    select r;
```

这个查询表达式使用了扩展方法 Where、OrderBy 和 Select。Enumerable 类定义了 Where 扩展方法，并将委托类型 Func<T, bool> 作为参数谓词。

```
public static IEnumerable<T> Where<T> (this IEnumerable<T> source,
                                         Func<T, bool> predicate);
```

这样，就把 表达式赋予谓词。这里 表达式类似于前面介绍的匿名方法。

```
Func<T, bool> predicate = r.Country == "Brazil";
```

Enumerable 类不是唯一一个定义了扩展方法 Where 的类。Queryable<T> 类也定义了 Where 扩展方法。这个类对 Where 扩展方法的定义是不同的：

```
public static IQueryable<T> Where<T> (this IQueryable<T> source,
                                         Expression<Func<T, bool>> predicate);
```

其中， 表达式赋予类型 Expression<T>，它的操作是不同的：

```
Expression<Func<T, bool>> predicate = r.Country == "Brazil";
```

除了使用委托之外，编译器还会把表达式树放在程序集中。表达式树可以在运行期间读取。表达式树从派生自抽象基类 Expression 的类中建立。Expression 类与 Expression<T> 不同。继承了 Expression 的

表达式类有 BinaryExpression ConstantExpression InvocationExpression LambdaExpression NewExpression NewArrayExpression TernaryExpression UnaryExpression等。编译器会从表达式中创建表达式树。

例如， 表达式 r.Country==“Brazil” 使用了 ParameterExpression MemberExpression ConstantExpression 和 MethodCallExpression, 来创建一个表达式树，将该树存储在程序集中。之后在运行期间使用这个树，创建一个用于底层数据源的优化查询。

方法 DisplayTree() 在控制台上图形化地显示表达式树。其中传送了一个 Expression 对象，并根据表达式类型，把表达式的一些信息写到控制台上。根据表达式的类型，递归调用方法 DisplayTree()。

提示：

在这个方法中，没有处理所有的表达式类型，只处理了下列示例表达式中使用的类型：

```
private static void DisplayTree(int indent,
    string message, Expression expression)
{
    string output = String.Format("{0} {1}" +
        "! NodeType: {2}; Expr: {3} ",
        "" .PadLeft(indent, ' > '), message,
        expression.NodeType, expression);

    indent++;

    switch (expression.NodeType)
    {
        case ExpressionType.Lambda:
            Console.WriteLine(output);
            LambdaExpression lambdaExpr =
                (LambdaExpression)expression;
            foreach (var parameter in
                lambdaExpr.Parameters)
            {
                DisplayTree(indent, "Parameter",
                    lambdaExpr.Parameters[i]);
            }
    }
}
```

```
    parameter);

}

DisplayTree(indent, "Body",
    lambdaExpr.Body);

break;

case ExpressionType.Constant:
    ConstantExpression constExpr =
        (ConstantExpression)expression;
    Console.WriteLine("{0} Const Value: " +
        "{1}", output, constExpr.Value);
    break;

case ExpressionType.Parameter:
    ParameterExpression paramExpr =
        (ParameterExpression)expression;
    Console.WriteLine("{0} Param Type: {1}",
        output, paramExpr.Type.Name);
    break;

case ExpressionType.Equal:
case ExpressionType.AndAlso:
case ExpressionType.GreaterThan:
    BinaryExpression binExpr =
        (BinaryExpression)expression;
    if (binExpr.Method != null)
    {
        Console.WriteLine("{0} Method: {1}",
            output, binExpr.Method.Name);
    }
else
{
```

```
        Console.WriteLine(output);

    }

    DisplayTree(indent, "Left",
                binExpr.Left);

    DisplayTree(indent, "Right",
                binExpr.Right);

    break;

case ExpressionType.MemberAccess:
    MemberExpression memberExpr =
        (MemberExpression)expression;

    Console.WriteLine("{0} Member Name: " +
        "{1}, Type: {2}", output,
        memberExpr.Member.Name,
        memberExpr.Type.Name);

    DisplayTree(indent, "Member Expr",
                memberExpr.Expression);

    break;

default:
    Console.WriteLine();

    Console.WriteLine("....{0} {1}",
        expression.NodeType,
        expression.Type.Name);

    break;
}
```

前面已经介绍了用于显示表达式树的表达式。这是一个 表达式，它使用一个 Racer参数，表达式体提取赢得比赛次数超过 6次的巴西赛手：

```
Expression < Func < Racer, bool > > expression =
    r => r.Country == "Brazil" && r.Wins > 6;
```

```
DisplayTree(0, "Lambda", expression);
```

下面看看结果。 表达式包含一个 Parameter和一个 AndAlso节点类型。 AndAlso节点类型的左边是一个 Equal节点类型，右边是一个 GreaterThan节点类型。 Equal节点类型的左边是 MemberAccess节点类型，右边是 Constant节点类型。

```
Lambda! NodeType: Lambda; Expr: r => ((r.Country = "Brazil")&&(r.Wins>6))  
> Parameter! NodeType: Parameter; Expr: r Param Type: Racer  
> Body! NodeType: AndAlso; Expr: ((r.Country = "Brazil") & & (r.Wins > 6))  
> > Left! NodeType: Equal; Expr: (r.Country = "Brazil") Method: op_Equality  
> > > Left! NodeType: MemberAccess; Expr: r.Country Member Name: Country, Type:  
String  
> > > Member Expr! NodeType: Parameter; Expr: r Param Type: Racer  
> > > Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil  
> > Right! NodeType: GreaterThan; Expr: (r.Wins > 6)  
> > > Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32  
> > > Member Expr! NodeType: Parameter; Expr: r Param Type: Racer  
> > > Right! NodeType: Constant; Expr: 6 Const Value: 6
```

使用类型 Expression<T>的一个例子是 LINQ to SQL。LINQ to SQL用 Expression<T>参数定义了扩展方法。这样，访问数据库的 LINQ提供程序就可以读取表达式，创建一个运行期间优化的查询，从数据库中获取数据。

## 11.4 LINQ 提供程序

.NET 3.5包含几个 LINQ提供程序。LINQ提供程序为特定的数据源实现了标准的查询操作符。LINQ提供程序也许会实现 LINQ定义的更多扩展方法，但至少要实现标准操作符。LINQ to XML不仅实现了专门用于 XML的方法，还实现了其他方法，例如 System.Xml.Linq命名空间的 Extensions类定义的方法 Elements() Descendants和 Ancestors。

LINQ提供程序的实现方案是根据命名空间和第一个参数的类型来选择的。实现扩展方法的类的命名空间必须是打开的，否则扩展类就不在作用域内。在 LINQ to Objects中定义的 Where()方法参数和在 LINQ to SQL中定义的 Where()方法参数是不同的。

LINQ to Objects中的 Where()方法是用 Enumerable类定义的：

```
public static IEnumerable < TSource > Where < TSource > (  
    this IEnumerable < TSource > source,  
    Func < TSource, bool > predicate);
```

在 System.Linq 命名空间中，还有另一个类实现了操作符 Where 这个实现代码由 LINQ to SQL 使用，这些代码在类 Queryable 中：

```
public static IQueryable<TSource> Where<TSource> (  
    this IQueryable<TSource> source,  
    Expression<Func<TSource, bool>> predicate);
```

这两个类都在 System.Linq 命名空间的 System.Core 程序集中实现。那么，它是如何定义的？使用了什么方法？无论是用 Func<TSource, bool> 参数传递，还是用 Expression<Func<TSource, bool>> 参数传递，表达式都是相同的。只是编译器的操作是不同的，它根据 source 参数来选择。编译器根据其参数选择最匹配的方法。在 LINQ to SQL 中定义的 DataContext 类的 GetTable() 方法返回 IQueryable<TSource>，因此 LINQ to SQL 使用类 Queryable 的 Where() 方法。

LINQ to SQL 提供程序使用表达式树，实现了接口 IQueryable 和 IQueryProvider。

## 11.5 小结

本章介绍了 C# 3.0 版本中最重要的改进。C# 在继续发展，在 C# 2.0 中，主要的新特性是泛型，它为类型安全的泛型集合类、泛型接口和委托奠定了基础。C# 3.0 的主要新特性是 LINQ，通过它可以使用与语言集成的语法查询任意数据源，只要该数据源有提供程序即可。

本章讨论了 LINQ 查询和查询所基于的语言结构，例如扩展方法和 表达式，还列出了各种 LINQ 查询操作符，它们不仅用于过滤数据源，给数据源排序，还用于执行分区、分组、转换、连接等操作。

LINQ 是一个非常深奥的主题，更多的信息可查阅第 27~29 章和附录 A，还可以下载其他第三方提供程序，例如 LINQ to MySQL、LINQ to Amazon、LINQ to Flickr 和 LINQ to SharePoint，无论使用什么数据源，都可以通过 LINQ 使用相同的查询语法。

另一个重要的概念是表达式树。表达式树允许在运行期间建立对数据源的查询，因为表达式树存储在程序集中。表达式树的用法详见第 27 章。

# 第 12 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

## 12.1 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。

### 12.1.1 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 位处理器上第 部分 C# 语言

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。

堆栈指针

存储单元

已用

未用

800000

799999

799998

799997

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。

## 294

### 第11 章内存管理和指针

```
{  
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。

### 12.1.2 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

**295**

第 部分 C# 语言

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。

```
200000
堆栈指针
未用堆
未用堆栈 199999
已用堆
已用堆栈
799996~799999
arabel
```

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。

```
堆栈指针
已用堆栈未用堆
未用堆栈
799996~799999
arabel
```

199999  
已用堆  
200032  
200000~200031  
arabel 实例  
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();
```

## 296

### 第11 章内存管理和指针

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。

### 12.1.3 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。

已使用  
空闲空间  
已使用  
已使用  
空闲空间  
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

## 297

### 第 部分 C# 语言

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，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。

## 12.2 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有不同的解决方法。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。

### 12.2.1 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，

**298**

第11 章内存管理和指针

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。

## 299

### 第 部 分 C# 语 言

#### 12.2.2 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收集器相关的问题。 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 **300**

#### 第11 章内存管理和指针

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。

#### 12.2.3 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)
{
301
第 部分 C# 语言
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

**302**

第11章内存管理和指针

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。

## 12.3 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。

### 12.3.1 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`DLL Import`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

**303**

第部分 C#语言

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`:

## 304

### 第11 章内存管理和指针

```
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。

0x12F8C4~ 0x12F8C7 x=20(-0 x 14)  
0x12F8C0~ 0x12F8C3 px=0 x 12F8C4  
0x12F8BC~ 0x12F8BF py=012F8C4

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。

## 306

### 第11 章内存管理和指针

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.WriteLine() 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;
```

## 307

### 第 部分 C# 语言

```
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

## 308

第11 章 内存管理和指针

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  $P + X * (\text{sizeof}(T))$ ,

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

数据类型	大小
char, sbyte	1
short, ushort	2
int, uint	4

## 309

第 部分 C# 语言

(续表)

long, ulong	8
float	4
double	8
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;
```

## 310

第11章内存管理和指针

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：

### 311

第 部分 C# 语言

```
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。

#### 12.3.2 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;
```

### 312

## 第11 章内存管理和指针

```
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

**313**

第 部 分 C# 语 言

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，可以证实用本章前面的“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;
    }
}
```

## 314

第11章内存管理和指针

```
}
```

```
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

**315**

## 第 部分 C# 语言

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
```

## 316

### 第11章内存管理和指针

```
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 ( $1 \text{ long} (=8) + 1 \text{ byte} (=1) = 9 \text{ 字节}$ )。这是前面讨论的对齐单词的结果。

#### 12.3.3 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，给它不必要的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

317

第 部分 C# 语言

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。

由 `stackalloc`

返回的指针

堆栈上分配的

连续存储单元

数组元素 0

数组元素 1

数组元素 2

等

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

## 318

### 第11 章 内存管理和指针

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是在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));
```

### 319

第 部分 C# 语言

```
}
```

```
}
```

```
}
```

```
}
```

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
```

## 12.4 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 13 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。

第 部分 C# 语言

## 13.1 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。

### 13.1.1 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

## 322

### 第12章 反射

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

### 323

第 部分 C# 语言

- 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。AttributeUsage 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  
**324**

### 第12 章 反射

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.
```

## 325

第 部分 C# 语言

### 13.1.2 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 `SupportsWhatsNewAttribute`。`LastModifiedAttribute` 特性可以用于标记最后一次修改数据项的时间, 它有两个必选参数 (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)]
}
```

## 326

第12 章反 射

```
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。

### 327

#### 第 部分 C# 语言

##### 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")]
```

## 328

### 第12章 反射

```
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。

## 13.2 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。

### 13.2.1 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 329

第 部分 C# 语言

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

Ful INName 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：

### 330

#### 第12 章 反 射

```
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。

### 13.2.2 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

**331**

第 部分 C# 语言

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 `MethodInfo` 对象获取所有公有实例成员的信息，参数信息不能通过 `MethodInfo` 对象来获得。为了获取该信息，需要引用 `MethodInfo` 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;
```

**332**

第12 章反 射

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)
```

### 333

第 部分 C# 语言

```
{}
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
```

### 13.2.3 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 What'sNewAttributes 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：

**334**

第12 章 反 射

```
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 WhatsNewAttributes 程序集的一个引用和支持 WhatsNewAttribute 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));
```

### 335

第 部分 C# 语言

#### 13.2.4 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)
    {
```

### 336

第12 章反 射

```
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.ToString());
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[] attrs = Attribute.GetCustomAttributes(type);
if (attrs.Length == 0)
{
AddToMessage("No changes to this class\n");
}
else
{
foreach (Attribute attrib in attrs)
{
WriteAttributeInfo(attrib);
}
}
```

### 337

第 部分 C# 语言

```
}
```

```
MethodInfo[] methods = type.GetMethods();
AddToMessage("CHANGES TO METHODS OF THIS CLASS:");
foreach (MethodInfo nextMethod in methods)
{
object[] attrs2 =
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 LastModified Attribute 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;
    }

338

第12 章 反射


}
// check that date is in range
DateTime modifiedDate = lastModifiedAttrib.DateModified;
if (modifiedDate < backDateTo)
{
    return;
}
AddToMessage(" MODIFIED: " +
    modifiedDate.ToString("yyyy-MM-dd HH:mm:ss"));
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

**339**

第 部分 C# 语言

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）。

### 13.3 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。—

# 第 14 章 错误和异常

错误的出现并不总是编写应用程序的人的原因，有时应用程序会因为终端用户的操作而发生错误。无论如何，我们都应预测应用程序和代码中出现的错误。

.NET Framework 改进了处理错误的方式。C# 处理错误的机制可以为每种错误提供定制的处理，并把识别错误的代码与处理错误的代码分离开来。

本章的主要内容如下：

    异常类

    使用 try-catch-finally 捕获异常

    创建用户定义的异常

学习完本章后，您将很好地掌握 C# 应用程序中的高级异常处理技术。

## 14.1 错误和异常处理

无论编码技术有多好，程序都必须能处理可能出现的错误。例如，在一些复杂的处理过程中，代码没有读取文件的许可，或者在发送网络请求时，网络可能会中断。在这种情况下，方法只返回相应的错误代码通常是不够的——可能方法调用嵌套了 15 级或者 20 级，此时，代码需要跳过所有的 15 或 20 级方法调用，才能完全退出任务，采取相应的措施。

C# 语言提供了处理这种情形的绝佳工具，称为异常处理机制。

注意：

在 VB6 中，错误处理工具的功能非常有限，主要是 On Error GoTo 语句。如果您有 VB6 的背景知识，就会发现 C# 异常打开了程序中处理错误的全新世界的大门。另一方面，Java 和 C++ 开发人员会比较熟悉异常的规则，因为这些语言处理错误的方式与 C# 相同。C++ 开发人员会留意异常是因为 C++ 可能会因此而降低性能，但在 C# 中就不是这样。在 C# 代码中使用异常一般不影响性能。VB 2005 开发人员会发现，在 C# 中处理异常非常类似于在 VB 中使用异常（但语法不同）。

第 部分 C# 语言

### 14.1.1 异常类

在 C# 中，当出现某个异常错误条件时，就会创建一个异常对象。这个对象包含有助于跟踪问题的信息。我们可以创建自己的异常类（详见后面的内容），但 .NET 提供了许多预定义的异常类。

异常基类

本节将快速总结 .NET 基类库中可以使用的一些异常。Microsoft 在 .NET 中定义了大量的异常类，这里不可能提供详尽的列表。图 13-1 所示的类结构图显示了其中的一些类，给

出了大致的模式。

图 13-1

这个图中的所有类都在 System 命名空间中，但 IOException 和派生于 IOException 的类除外，它们在 System.IO 命名空间中，这个命名空间处理文件数据的读写。一般情况下，异常没有特定的命名空间，异常类应放在生成异常的类所在的命名空间中，因此与 IO 相关的异常就在 System.IO 命名空间中。许多基类命名空间中都有异常类。

对于 .NET 类来说，一般的异常类 System.Exception 派生于 System.Object，通常不在代码中抛出这个 System.Exception 对象，因为它无法确定错误情况的本质。

在该层次结构中有两个重要的类，它们派生于 System.Exception：

System.SystemException——通常由 .NET 运行库生成，或者有着非常一般的本质、可以由几乎所有的应用程序生成。例如，如果 .NET 运行库检测到堆栈已满，就会

**342**

第13 章错误和异常

抛出 StackOverflowException 另一方面，如果检测到调用方法时参数不正确，可以在自己的代码中选择抛出 ArgumentException 或其子类。System.SystemException 的子类包括表示致命错误和非致命错误的异常。

System.ApplicationException——这个类非常重要，因为它是第三方定义的异常基类。如果自己定义的异常覆盖了应用程序独有的错误情况，就应使它们直接或间接派生于 System.ApplicationException。

其他可能用到的异常类包括：

StackOverflowException——如果分配给堆栈的内存区域已满，就会抛出这个异常。如果一个方法连续地递归调用它自己，就可能发生堆栈溢出。这一般是一个致命错误，因为它禁止应用程序执行除了中断以外的其他任务。在这种情况下，甚至也不可能执行 finally 块，通常用户自己不能处理像这样的错误，而应退出应用程序。

EndOfStreamException——这个异常通常是因为读到文件末尾而抛出的。第 35 章将解释流，流表示数据源之间的数据流。

OverflowException——如果要在 checked 环境下把包含值 -40 的 int 类型数据转换为 uint 数据，就会抛出这个异常。

我们不打算讨论图 13-1 中的所有其他异常类。

异常的类层次结构并不多见，因为其中的大多数类并没有给它们的基类添加任何功能。但是在异常处理时，添加继承类的一般原因是更准确地指定错误，所以不需要重写方法或添加新方法（但常常要添加额外的属性，以包含有关错误情况的额外信息）。例如，当传递了不正确的参数值时，可给方法调用使用 ArgumentException 基类， ArgumentNullException 派生于 ArgumentException 类，它专门用于传送参数值是 Null 的情况。

### 14.1.2 捕获异常

.NET Framework 提供了大量的预定义基类异常对象，该如何在代码中使用它们捕获错误？为了在 C# 代码中处理可能的错误，一般要把程序的相关部分分成 3 种不同类型的代码块：

try 块包含的代码组成了程序的正常操作部分，但这部分程序可能遇到某些严重的错误。

catch 块包含的代码处理各种错误，这些错误是 try 块中的代码执行时遇到的。这个块还可以用于记录错误。

finally 块包含的代码清理资源或执行要在 try 块或 catch 块末尾执行的其他操作。无论是否产生异常，都会执行 finally 块 \_\_\_\_\_，理解这一点是非常重要的。因为 finally 块包含了应总是执行的清理代码，如果在 finally 块中放置了 return 语句，编译器就会标记一个错误。例如，可以在 finally 块中关闭在 try 块中打开的连接。finally 块

是可选的。如果不需要清理代码（例如删除对象或关闭已打开的对象），就不需要包含此块。

那么，这些块是如何组合在一起捕获错误的？下面就是其步骤：

### 343

第 部分 C# 语言

(1) 程序流进入 try 块。

(2) 如果没有错误发生，就会正常执行操作。当程序流离开 try 块后，即使什么也没有发生，也会自动进入 finally 块（第 5 步）。但如果在 try 块中程序流检测到一个错误，程序流就会跳转到 catch 块（下一步）。

(3) 在 catch 块中处理错误。

(4) 在 catch 块执行完后，程序流会自动进入 finally 块：

(5) 执行 finally 块。

用于完成这些任务的 C# 语法如下所示：

```
try
{
    // code for normal execution
}
catch
{
    // error handling
}
finally
{
    // clean up
}
```

实际上，上面的代码还有几种变体：

可以省略 finally 块

可以提供任意多个 catch 块，处理不同类型的错误。但不应包含过多的 catch 块，以防降低应用程序的性能。

可以省略 catch 块——此时，该语法应不是标识异常，而是一种确保程序流在离开 try 块后执行 finally 块中的代码的方式，如果在 try 块中有几个出口，这是很有用的。这看起来很不错，实际上是有问题的。如果执行 try 块中的代码，则程序流如何在错误发生时切换到 catch 块上？如果检测到一个错误，代码就执行一定的操作，称为“抛出一个异常”；换言之，它实例化一个异常对象，并抛出这个异常：

```
throw new OverflowException();
```

这里实例化了 OverflowException 类的一个异常对象。只要计算机在 try 块中遇到一个 throw 语句，就会立即查找与这个 try 块对应的 catch 块。如果有多个与 try 块对应的 catch 块，计算机就会检查与 catch 块对应的异常类，确定正确的 catch 块。例如，当抛出一个 OverflowException 对象时，执行流就会跳转到下面的 catch 块上：

```
catch (OverflowException e)
{
    // exception handling here
}
```

换言之，计算机查找的 catch 块应表示同一个类（或基类）中匹配的异常类实例。

有了这些额外的信息，就可以扩展刚才介绍的 try 块。为了讨论方便，假定可能在 try

### 344

第13 章 错误和异常

块中发生两个严重错误：溢出和数组超出范围。假定代码包含两个布尔变量 Overflow 和 OutOfBounds，表示这两种错误情况是否存在。我们知道，存在表示溢出的预定义溢出类 OverflowException，同样，存在预定义的 IndexOutOfRangeException 类，用于处理数组超出范围错误。

现在，try 块如下所示：

```
try
{
    // code for normal execution
    if (Overflow == true)
        throw new OverflowException();
    // more processing
    if (OutOfBounds == true)
        throw new IndexOutOfRangeException();
    // otherwise continue normal execution
}
catch (OverflowException ex)
{
    // error handling for the overflow error condition
}
catch (IndexOutOfRangeException ex)
{
    // error handling for the index out of range error condition
}
finally
{
    // clean up
}
```

我们得到的 try 块看起来并不比 VB6 的 On Error GoTo 语句强多少，但可以更清晰地将不同的代码段分开。实际上，C# 的错误处理有一个更强大、更灵活的机制。

这是因为 throw 语句可以嵌套在 try 块的几个方法调用中，甚至在程序流进入其他方法时，也会继续执行同一个 try 块。如果计算机遇到一个 throw 语句，就会立即退出堆栈中所有的方法调用，查找 try 块的结尾和合适的 catch 块的开头，此时，中间方法调用中的所有局部变量都会出作用域。try..catch 结构最适合于本节开头描述的场合：错误发生在一个方法调用中，而该方法调用可能嵌套了 15 到 20 级，这些处理操作会立即停止。

从上面的论述可以看出，try 块在控制程序的执行流上有着重要的作用。但是，异常类是用于处理异常情况的，这是其名称的由来。不应该用异常来控制退出 do..while 循环的时间。

### 1. 执行多个 catch 块

要了解 try..catch..finally 块是如何工作的，最简单的方式是用两个示例来说明。第一个示例是 SimpleException。它多次要求用户键入一个数字，然后显示这个数字。为了便于

**345**

#### 第 部 分 C# 语 言

解释这个示例，假定该数字必须在 0 和 5 之间，否则程序就不能对该数字进行正确的处理。所以，如果用户键入超出该范围的数字，程序就抛出一个异常。

程序会继续要求用户输入数字，直到用户不再输入任何内容，但按下了回车键为止。

注意：

这段代码没有说明何时使用异常处理。前面已经提及，异常是用于处理异常情况的。

用户总是键入一些无聊的东西，所以这种情况不会真正发生。正常情况下，程序会处理不

正确的用户输入，进行即时检查，如果有问题，就要求用户重新键入。但是，在一个要求几分钟内读懂的小示例中生成异常是比较困难的，为了描述异常是如何工作的，后面将使用更真实的示例。

SimpleExceptions 的代码如下所示：

```
using System;
namespace Wrox.ProCSharp.AdvancedCSharp
{
    public class MainEntryPoint
    {
        public static void Main()
        {
            string userInput;
            while ( true )
            {
                try
                {
                    Console.Write("Input a number between 0 and 5 " +
                        "(or just hit return to exit)> ");
                    userInput = Console.ReadLine();
                    if (userInput == "")
                        break;
                }
                int index = Convert.ToInt32(userInput);
                if (index < 0 || index > 5)
                    throw new IndexOutOfRangeException(
                        "You typed in " + userInput);
                }
                Console.WriteLine("Your number was " + index);
            }
            catch (IndexOutOfRangeException ex)
            {
                Console.WriteLine("Exception: " +
                    "Number should be between 0 and 5. " + ex.Message);
            }
            catch (Exception ex)
            {

```

## 346

### 第13 章错误和异常

```
Console.WriteLine(
    "An exception was thrown. Message was:{0} " + ex.Message);
}
catch
{
    Console.WriteLine("Some other exception has occurred");
}
finally
{

```

```
Console.WriteLine("Thank you");
}
}
}
}
}
```

这段代码的核心是一个 while 循环，它连续使用 Console.ReadLine() 以请求用户输入。ReadLine() 返回一个字符串，所以程序首先要用 System.Convert.ToInt32() 方法把它转换为 int 型。System.Convert 类包含执行数据转换的各种有用的方法，并提供了 int.Parse() 方法的一个替代方法。一般情况下，System.Convert 包含执行各种类型转换的方法，C# 编译器把 int 解析为 System.Int32 基类的实例。

注意：

传递给 catch 块的参数只能用于该 catch 块。这就是为什么在上面的代码中，能在后续的 catch 块中使用相同的参数名 ex 的原因。

在上面的代码中，我们也检查一个空字符串，因为该空字符串是退出 while 循环的条件。注意这里用 break 语句退出 try 块和 while 循环——这是有效的。当然，当程序流退出 try 块时，会执行 finally 块中的 Console.WriteLine() 语句。尽管这里仅显示一句问候，仍需要执行关闭文件句柄，调用各种对象的 Dispose() 方法，以执行清理工作。一旦计算机退出了 finally 块，就会继续执行下一个语句，如果没有 finally 块，该语句也会执行。在本例中，我们返回 while 循环的开头，再次进入 try 块（除非执行 while 循环中 break 语句的结果是进入 finally 块，此时就会退出 while 循环）。

下面看看异常情况：

```
if (index < 0 || index > 5)
{
    throw new IndexOutOfRangeException("You typed in " + userInput);
}
```

在抛出一个异常时，需要选择要抛出的异常类型。可以使用类 System.Exception，但这个类是一个基类，最好不要把这个类的实例当作一个异常，因为它没有包含错误的任何信息。.NET Framework 定义了许多派生于 System.Exception 的其他异常类，每个类都对应于一种类型的异常情况，也可以定义自己的异常类。在抛出一个匹配特定错误情况的类实例时，应提供尽可能多的异常信息。在本例中，System.IndexOutOfRangeException 是最佳

**347**

第 部分 C# 语言

选择。IndexOutOfRangeException 有几个构造函数重载，我们选择的一个重载，其参数是一个描述错误的字符串。另外，也可以选择派生自己的定制异常对象，描述该应用程序环境中的错误情况。

假定用户这次键入了一个不在 0 到 5 范围内的数字，if 语句就会检测到一个错误，并实例化和抛出一个 IndexOutOfRangeException 对象。计算机会立即退出 try 块，查找处理 IndexOutOfRangeException 的 catch 块。它遇到的第一个 catch 块如下所示：

```
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(
        "Exception: Number should be between 0 and 5." + ex.Message);
}
```

由于这个 catch 块带一个合适类的参数，它就会传送给异常实例，并执行。在本例中，是显示错误信息和 Exception.Message 属性（它对应于给 IndexOutOfRangeException 的构造函数传递的字符串）。执行了这个 catch 块后，控制就切换到 finally 块，就好像没有发生过任何异常。

注意，本例还提供了另一个 catch 块：

```
catch (Exception ex)
{
    Console.WriteLine("An exception was thrown. Message was: " + ex.Message);
}
```

如果没有在前面的 catch 块中捕获到这类异常，这个 catch 块也能处理 `IndexOutOfRangeException` —— 基类的一个引用也可以指向派生于它的所有类实例，所有的异常都派生于 `System.Exception`。那么为什么不执行这个 catch 块？答案是计算机只执行它在 catch 块列表中找到的第一个合适的 catch 块。但为什么还要编写第二个 catch 块？不仅 try 块包含这段代码，还有另外 3 个方法调用 `Console.ReadLine()`、`Console.Write()` 和 `Convert.ToInt32()` 也包含这段代码，它们是 `System` 命名空间中的方法。这 3 个方法都可能抛出异常。

如果键入的内容不是数字，例如 `a` 或 `hello`，`Convert.ToInt32()` 方法就会抛出一个 `System.FormatException` 类的异常，表示传递给 `ToInt32()` 的字符串不能转换为 `int`。此时，计算机会跟踪这个方法调用，查找可以处理该异常的处理程序。第一个 catch 块带一个 `IndexOutOfRangeException`，不能处理这种异常。计算机接着查看第二个 catch 块，显然它可以处理这类异常，因为 `FormatException` 派生于 `Exception`，所以把 `FormatException` 实例作为参数传送给它。

示例的这种结构是非常典型的多 catch 块结构。最先编写的 catch 块用于处理非常特殊的错误情况，接着是比较一般的块，它们可以处理任何错误，我们没有为它们编写特定的错误处理程序。实际上，catch 块的顺序是很重要的，如果以相反的顺序编写这两个块，代码就不会编译，因为第二个 catch 块是不会执行的（Exception catch 块会捕获所有的异常），因此，最上面的 catch 块应用于最特殊的异常情况，最后是最一般的 catch 块。

但是，在上面的例子中还有第三个 catch 块：

```
catch
{
    348
    第13 章错误和异常
    Console.WriteLine("Some other exception has occurred");
}
```

这就是最一般的 catch 块—— 它不带参数，原因是这个 catch 块处理的是其他没有用 C# 编写的代码（甚或根本不是托管代码）抛出的异常。在 C# 语言中，只有派生于 `System.Exception` 的类实例，才能作为异常来抛出。但其他语言没有这个限制，例如，C++ 允许把任何变量作为异常来抛出。如果代码调用了用其他语言编写的库或程序集，抛出的异常就可能不是派生于 `System.Exception`。但在许多情况下，.NET PInvoke 机制会捕获这些异常，把它们转换为 .NET `Exception` 对象。但这个 catch 块做的工作并不多，因为我们不知道该异常表示什么类。

注意：

在上面的示例中，也可以不添加这个通用的 catch 处理程序，如果要调用其他不支持 .NET 但可能抛出异常的库，这么做是很有效的。这个示例包含它，主要是为了说明这个规则。

前面分析了示例的代码，现在可以试着运行它。下面的输出说明了不同的输入会得到不同的结果，并说明抛出了 `IndexOutOfRangeException` 和 `FormatException`：

#### SimpleExceptions

```
Input a number between 0 and 5 (or just hit return to exit)> 4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 0
```

```
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)> 10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Message was: Input string was not in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
```

## 2. 在其他代码中捕获异常

在上面的示例中，说明了两个异常的处理。一个异常是 `IndexOutOfRangeException`，它由我们自己的代码抛出，另一个异常是 `FormatException`，由一个基类抛出。如果检测到错误，或者某个方法因传递的参数有误而被错误调用，库中的代码常常就会抛出一个异常。但库中的代码很少捕获这样的异常。应由客户机代码来决定如何处理这些问题。

在调试时，异常经常从基类库中抛出，调试的过程在某种程度上是确定异常抛出的原因，并去除导致错误发生的缘由。主要目标是确保代码在执行后，异常只发生在非常少见的情况下，如果可能，应在代码中以适当的方式处理它。

## 349

### 第 部 分 C# 语 言

#### 3. System.Exception 属性

在示例中，我们只使用了异常对象的一个属性 `Message`。在 `System.Exception` 中还有许多其他属性，如表 13-1 所示。

表 13-1

##### 属 性 说 明

`Data` 这个属性可以给异常添加键 / 值语句，以提供异常的额外信息。这是 .NET Framework 2.0 中的一个新属性。

`HelpLink` 链接到一个帮助文件上，以提供该异常的更多信息。

`InnerException` 如果此异常是在 `catch` 块中抛出的，它就会包含把代码发送到 `catch` 块中的异常对象。

`Message` 描述错误情况的文本。

`Source` 导致异常的应用程序或对象名。

`StackTrace` 堆栈上方法调用的信息，它有助于跟踪抛出异常的方法。

`TargetSite` 描述抛出异常的方法的 .NET 反射对象。

在这些属性中，如果可以进行堆栈跟踪，`StackTrace` 和 `TargetSite` 是由 .NET 运行库自动提供的。`Source` 总是由 .NET 运行库提供为产生异常的程序集名称（但可以在代码中修改该属性，提供更专门的信息），`Data`、`Message`、`HelpLink` 和 `InnerException` 必须由抛出异常的代码提供，其方法是在抛出异常前设置这些属性。例如，抛出异常的代码如下所示：

```
if (ErrorCondition == true)
{
    Exception myException = new ClassmyException("Help!!!!");
    myException.Source = "My Application Name";
    myException.HelpLink = "MyHelpFile.txt";
    myException.Data["ErrorDate"] = DateTime.Now;
    myException.Data.Add("AdditionalInfo", "Contact Bill from the Blue Team");
    throw myException;
}
```

其中 ClassMyException 是抛出的异常类名。注意所有的异常类名通常以 Exception 结尾。 Data 属性可以用两种方式设置。

#### 4. 没有处理异常时所发生的情况

有时生成了一个异常后，代码中没有 catch 块能处理这类异常。前面的 SimpleExceptions 示例就说明了这种情况。例如，假定忽略 FormatException 和通用的 catch 块，只有处理 IndexOutOfRangeException 的块。此时，如果抛出一个 FormatException 异常，会发生什么情况呢？

答案是 .NET 运行库会捕获它。在本节的后面将介绍如何嵌套 try 块——实际上在本示例中，就有一个在后台处理的嵌套 try 块。.NET 运行库把整个程序放在另一个更大的 try 块中，每个 .NET 程序都会这么做。这个 try 块有一个 catch 处理程序，它可以捕获任何类

350

#### 第13 章错误和异常

型的异常。如果代码没有处理发生的异常，程序流就会退出程序，由 .NET 运行库中的 catch 块捕获它。但是，结果并不是你想像的那样。代码的执行会立即中断，并给用户显示一个对话框，说明代码没有处理异常，并给出 .NET 运行库能检索到的异常信息。至少异常会被捕获，这就是第 2 章在 Vector 示例程序抛出一个异常时发生的情况。

一般情况下，如果编写一个可执行程序，就应捕获尽可能多的异常，并以合理的方式处理它们。如果编写一个库，最好不要捕获异常（除非某个异常表示的是代码可以处理的情况），但要假定调用代码可以处理它们。当然，用户可能不想捕获任何 Microsoft 预定义的异常，而是抛出自己的异常对象，给客户机代码提供更特定的信息。

#### 5. 嵌套的 try 块

异常的一个特性是 try 块可以嵌套，如下所示：

```
try
{
// Point A
try
{
// Point B
}
catch
{
// Point C
}
finally
{
// clean up
}
// Point D
}
catch
{
// error handling
}
finally
{
// clean up
}
```

在上面的代码中，每个 try 块都只有一个 catch 块，但可以把多个 catch 块连接在一起。

下面详细讨论嵌套的 try 块如何工作。

如果抛出的异常在外层的 try 块中，且在内层 try 块的外部（标记为 A 或 D 的代码块），这种情况就与前面介绍的情况没有任何区别：异常由外层的 catch 块捕获，并执行外层的 finally 块，或者执行 finally 块，由 .NET 运行库处理异常。

如果异常是在内层 try 块（代码块 B）中抛出的，且有一个合适的内层 catch 块处理该异常，这又是我们熟悉的情况：在内层处理异常，执行内层的 finally 块，之后继续执行外层

**351**

第 部分 C# 语言

的 try 块（标记为 D 的代码块）。

现在假定异常是在代码块的内层 try 块中抛出的，但内层的 catch 块中没有合适的处理程序。这时通常就要执行内层的 finally 块，但 .NET 运行库只能退出内层的 try 块，才能搜索到合适的处理程序。下一个要搜索的区域显然是外层的 catch 块。如果系统在这里找到一个处理程序，就会执行该处理程序，再执行外层的 finally 块，如果没有找到合适的处理程序，就会继续搜索。在这里，执行的是外层的 finally 块，因为没有更多的 catch 块，所以控制权会返回 .NET 运行库。注意，不会执行外层 try 块中标记为 D 的代码。

如果异常是在 C 代码块中抛出的，就更有趣了。如果程序执行到代码块 C 中，就必须处理由代码块 B 抛出的异常。在 catch 块中抛出另一个异常是很正常的。此时，异常的处理就跟它是在外层 try 块中抛出的一样，程序流会立即退出内层的 catch 块，执行内层的 finally 块，在外层的 catch 中搜索处理程序。同样，如果在内层的 finally 块中抛出一个异常，搜索会在外层的 catch 块开始，执行最匹配的处理程序。

注意：

在 catch 和 finally 块中抛出异常是合理的。

尽管本例只有两个 try 块，但无论嵌套了多少个 try 块，规则都是一样的。在每个块中，.NET 运行库顺序执行 try 块，查找合适的处理程序，在每个步骤中，当退出 catch 块后，就会执行对应 finally 块中的清理代码，但不执行 finally 块外部的代码，直到找到合适的 catch 处理程序，并执行为止。

前面说明了嵌套 try 块的工作方式。下一个问题显然是为什么要这么做？这有两个原因：

修改所抛出的异常的类型。

在代码的不同地方处理不同类型的异常。

### (1) 修改异常的类型

当最初抛出的异常不足以说明问题时，修改异常的类型就非常重要了。通常的情况是抛出的异常（可能由 .NET 运行库抛出）是一种相当低级的异常，说明发生溢出（OverflowException 或传递给方法的参数不正确（派生于 ArgumentException 的一个类））。但是，由于抛出异常的环境，我们知道这暴露了一些底层的问题（例如，因为刚才读取的文件包含了不正确的数据，才发生了溢出异常）。此时处理程序对于第一个异常所能做的最佳处理就是抛出另一个异常，以便更准确地说明这个问题，让另一个 catch 块更恰当地处理它。也可以通过一个由 System.Exception 执行的属性 InnerException 来处理最初的异常。

InnerException 只包含另一个相关异常的引用——最终的处理例程需要这个额外的信息。当然，还应指出，异常可能在 catch 块中抛出。例如，可以从某个配置文件中读取数据，这个文件包含处理错误的详细指令——结果可能是这个文件不存在。

### (2) 在不同的地方处理不同的异常

嵌套 try 块的第二个原因是不同类型的异常可以在代码的不同地方处理。例如，在循环中，可能会发生各种异常。其中一些异常比较严重，需要退出整个循环，而另外一些则不太严重，只需退出这次迭代，进入循环的下一次迭代即可。在循环的内部有一个 try 块

**352**

## 第13 章错误和异常

就可以处理不太严重的异常，再在循环外面用一个外层的 try 块来处理比较严重的错误。

在下面的异常示例中，将解释具体的操作情况。

### 14.1.3 用户定义的异常类

下面介绍有关异常的第二个示例，这个示例叫作 `MortimerColdCall`，包含了两个嵌套的 try 块，说明了如何定义定制的异常类，再从 try 块中抛出另一个异常。

这个示例假定一家销售公司希望有更多的客户。该公司的销售部门打算给一些人打电话，希望他们成为自己的客户。用销售行业的行话来讲，就是“cold-call”一些人。为此，应有一个文本文件存储这些人的姓名，该文件应有正确的格式，第一行包含文件中的人数，后面的行包含这些人的姓名。换言之，正确的格式如下所示。

4

```
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

这个示例的目的是在屏幕上显示这些人的姓名（由销售人员读取），这就是为什么只把姓名放在文件中，但没有电话号码的原因。

程序要求用户输入文件的名称，然后读取文件，以显示其中的人名。这听起来是一个很简单的任务，但也会出现两个错误，需要退出整个过程：

用户可能键入不存在的文件名。这由 `FileNotFoundException` 异常来处理。

文件的格式可能不正确，这里可能有两个问题。首先，文件的第一行不是整数。

第二，文件中可能没有第一行指定的那么多人名。这两种情况都需要在一个定制异常中处理，我们已经专门为此编写了异常 `ColdCallFormatException`。还有其他问题，虽然不至于退出整个过程，但需要删除某个人名，继续处理文件中的下一个人名（因此这需要在内层的 try 块中处理）。一些人是工业间谍，为公司的竞争对手工作，显然，我们不希望让这些人知道我们要做的工作。因此应确定哪些人是工业间谍，搜索条件是他们的姓名以 B 开头。这些人应在第一次准备数据文件时从文件中删除，但万一被工业间谍混入，就需要检查文件中的每个姓名，如果检测到一个工业间谍，就应抛出一个 `LandLineSpyFoundException`，当然，这是另一个定制的异常对象。

最后，编写一个类 `ColdCallFileReader` 来执行这个示例，该类维护与 cold-call 文件的连接，并从中检索数据。我们将以非常安全的方式编写这个类，如果其方法调用不正确，就会抛出异常。例如，如果在文件打开前，调用了读取文件的方法，就会抛出一个异常。为此，我们编写了另一个异常类 `UnexpectedException`。

#### 1. 捕获用户定义的异常

首先是 `MortimerColdCall` 示例的 `Main()` 方法，它捕获用户定义的异常。注意，下面要调用 `System.IO` 命名空间和 `System` 命名空间中的文件处理类。

```
using System;
353
第 部分 C# 语言
using System.IO;
namespace Wrox.ProCSharp.AdvancedCSharp
{
    class MainEntryPoint
    {
        static void Main()
        {
            string fileName;
```

```
Console.WriteLine("Please type in the name of the file " +
"containing the names of the people to be cold-called > ");
fileName = Console.ReadLine();
ColdCallFileReader peopleToRing = new ColdCallFileReader();
try
{
    peopleToRing.Open(fileName);
    for (int i=0 ; i<peopleToRing.NPeopleToRing; i++)
    {
        peopleToRing.ProcessNextPerson();
    }
    Console.WriteLine("All callers processed correctly");
}
catch(FileNotFoundException ex)
{
    Console.WriteLine("The file {0} does not exist", fileName);
}
catch(ColdCallFormatException ex)
{
    Console.WriteLine(
        "The file {0} appears to have been corrupted", fileName);
    Console.WriteLine("Details of problem are: {0}", ex.Message);
    if (e.InnerException != null)
    {
        Console.WriteLine(
            "Inner exception was: {0}", ex.InnerException.Message);
    }
}
catch(Exception ex)
{
    Console.WriteLine("Exception occurred:\n" + ex.Message);
}
finally
{
    peopleToRing.Dispose();
}
Console.ReadLine();
}
```

### 354

#### 第13 章错误和异常

这段代码基本上是一个循环，处理文件中的人名。开始时，先让用户输入文件名，再实例化类 ColdCallFileReader 的一个对象，这个类稍后定义，用于处理文件中数据的读取。注意是在第一个 try 块的外部读取文件——这是因为这里实例化的变量需要在后面的 catch 和 finally 块中使用，如果在 try 块中声明它们，它们在 try 块的闭合花括号处就出了作用域。

在 try 块中打开文件 (使用 ColdCallFileReader.Open()方法)，并循环处理其中所有的人

名。ColdCallFileReader.ProcessNextPerson()方法会读取并显示文件中的下一个人名，而ColdCallFileReader.NpeopleToRing属性则说明文件中应有多少个人名(通过读取文件的第一行来获得)。

有3个catch块，其中两个用于处理FileNotFoundException和ColdCallFormatException异常，第三个则用于处理其他.NET错误。

在FileNotFoundException中，我们会为它显示一个信息，注意在这个catch块中，根本没有使用异常实例，原因是这个catch块用于说明应用程序的用户友好性。异常对象一般会包含技术信息，这些技术对开发人员是很有用的，但对于最终用户来说则没有什么用，所以在本例中我们将创建一个更简单的消息。

对于ColdCallFormatException处理程序，我们已经完成了编码，说明了如何提供更完整的技术信息，包括内层异常的细节。

最后，如果捕获到其他一般异常，就显示一个用户友好消息，而不是让这些异常由.NET运行库处理。注意我们选择不处理派生于System.Exception的异常，因为不直接调用非.NET的代码。

Finally块清理资源。在本例中，是指关闭已打开的文件。ColdCallFileReader.Dispose()方法完成了这个任务。

## 2. 抛出用户定义的异常

下面看看处理文件读取，以及抛出用户定义的异常的类ColdCallFileReader的定义。

这个类维护一个外部文件连接，所以需要确保它根据第4章有关释放对象的规则，正确地释放。这个类派生于IDisposable。

首先声明一些变量：

```
class ColdCallFileReader : IDisposable
{
    FileStream fs;
    StreamReader sr;
    uint nPeopleToRing;
    bool isDisposed = false;
    bool isOpen = false;
```

FileStream和StreamReader都在System.IO命名空间中，都是用于读取文件的基类。

FileStream用于连接文件，StreamReader则专门用于读取文本文件，执行方法

StreamReader()，该方法读取文件中的一行文本。第24章在深入讨论文件处理时将讨论StreamReader。

## 355

### 第 部分 C# 语言

isDisposed字段表示是否调用了Dispose()方法，我们选择执行ColdCallFileReader，这样，一旦调用了Dispose()方法，就不能重新打开文件连接，重新使用对象了。isOpen也用于错误检查——在本例中，检查StreamReader是否连接到打开的文件上。

打开文件和读取第一行的过程——告诉我们文件中有多少个人名——由Open()方法来处理：

```
public void Open(string fileName)
{
    if (isDisposed)
        throw new ObjectDisposedException("peopleToRing");
    fs = new FileStream(fileName, FileMode.Open);
    sr = new StreamReader(fs);
    try
    {
```

```
string firstLine = sr.ReadLine();
nPeopleToRing = uint.Parse(firstLine);
isOpen = true;
}
catch (FormatException e)
{
throw new ColdCallFormatException(
"First line isn't an integer", e);
}
```

与其他 `ColdCallFileReader` 方法一样，该方法首先检查在删除对象后，客户机代码是否不正确地调用了它，如果是，就抛出一个预定义的 `ObjectDisposedException` 对象。`Open()` 方法也会检查 `isDisposed` 字段，看看 `Dispose()` 是否已经被调用。因为调用 `Dispose()` 会告诉调用者现在已经处理完对象，所以，如果已经调用了 `Dispose()`，就说明有一个试图打开新文件连接的错误。

接着，这个方法包含前两个内层的 `try` 块，其目的是捕获因文件的第一行没有包含一个整数而抛出的错误。如果出现这个问题，.NET 运行库就抛出一个 `FormatException`，捕获并转换一个更有意义的异常，表示 cold-call 文件的格式有问题。注意 `System.FormatException` 表示与基本数据类型相关的格式问题，而不是与文件有关，所以在本例中它不是传递回调用例程的一个特别有用的异常。新抛出的异常会被最外层的 `try` 块捕获。注意这里不需要清理资源，所以不需要 `finally` 块。

如果一切正常，就把 `isOpen` 字段设置为 `true`，表示现在有一个有效的文件连接，可以从中读取数据。

`ProcessNextPerson()` 方法也包含一个内层 `try` 块：

```
public void ProcessNextPerson()
{
if (isDisposed)
throw new ObjectDisposedException("peopleToRing");
```

## 356

### 第13 章错误和异常

```
if (!isOpen)
throw new UnexpectedException(
"Attempt to access cold-call file that is not open");
try
{
string name;
name = sr.ReadLine();
if (name == null)
throw new ColdCallFormatException("Not enough names");
if (name[0] == 'B')
{
throw new LandLineSpyFoundException(name);
}
Console.WriteLine(name);
}
catch (LandLineSpyFoundException ex)
{
```

```
Console.WriteLine(ex.Message);
}
finally
{
}
}
```

这里可能存在两个与文件相关的错误 (假定有一个打开的文件连接，`ProcessNextPerson()`方法会先进行检查)。第一，读取下一个人名时，可能发现这是一个工业间谍。如果发生这种情况，在这个方法中就使用第一个 `catch` 块捕获异常。因为这个异常已经在循环中被捕获，所以程序流会继续在程序的 `Main` 方法中执行，处理文件中的下一个人名。

如果读取下一个人名，发现已经到达文件的末尾，也会发生错误。`StreamReader` 对象的 `ReadLine()` 方法的工作方式是：如果到达文件末尾，就会返回一个 `null`，而不是抛出一个异常。所以，如果找到一个 `null` 字符串，文件的格式就不正确，因为文件第一行上的数字要比文件中的实际人数多，如果发生这种错误，就抛出一个 `ColdCallFormatException`，它由外层的异常处理程序捕获 (使程序中断执行)。

这里还是不需要 `finally` 块，因为没有要清理的资源，但这次要放置一个空的 `finally` 块，表示在这里可以完成你希望完成的任务。

这个示例就要完成了。`ColdCallFileReader` 还有另外两个成员：`NPeopleToRing` 属性返回文件中假定的人数，`Dispose()` 方法可以关闭打开的文件。注意 `Dispose()` 方法仅返回它是否被调用——这是执行该方法的推荐方式。它还检查在关闭前是否有一个文件流要关闭。

这个例子说明了防御编码技术：

```
public uint NPeopleToRing
{
    get
    {
        if (isDisposed)
            throw new ObjectDisposedException("peopleToRing");
        if (!isOpen)
            throw new UnexpectedException(
                "Attempt to access cold-call file that is not open");
        return nPeopleToRing;
    }
}

public void Dispose()
{
    if (isDisposed)
        return;
    isDisposed = true;
    isOpen = false;
```

## 357

### 第 部 分 C# 语 言

```
if (fs != null)
{
    fs.Close();
    fs = null;
}
```

### 3. 定义异常类

最后，需要定义 3 个异常类。定义自己的异常是非常简单的，因为几乎不需要添加任何额外的方法。只需执行构造函数，确保基类构造函数正确调用即可。下面是 SalesSpyFoundException 的完整代码：

```
class SalesSpyFoundException : ApplicationException
{
    public SalesSpyFoundException(string spyName)
        : base("Sales spy found, with name " + spyName)
    {
    }

    public SalesSpyFoundException(
        string spyName, Exception innerException)
        : base(
            "Sales spy found with name " + spyName, innerException)
    {
    }
}
```

### 358

#### 第13 章错误和异常

注意，这个类派生于 ApplicationException，正是我们期望的定制异常。实际上，如果要更正式地创建它，可以把它放在一个中间类中，例如 ColdCallFileException，它派生于 ApplicationException，再从这个类派生出两个异常类，并确保处理代码确切地知道哪个异常处理程序处理哪个异常即可。但为了使这个示例比较简单，就不这么做了。

在 SalesSpyFoundException 中，处理的内容要多一些。假定传送给构造函数的信息仅是找到的间谍名，就把这个字符串转换为含义更明确的错误信息。我们还提供了两个构造函数，其中一个构造函数的参数只是一个信息，另一个构造函数的参数还有一个内层异常。在定义自己的异常类时，最好把这两个构造函数都包括进来（但以后将不能在示例中使用第二个 LandLine-SpyFoundException 构造函数）。

对于 ColdCallFormatException，规则是一样的，但不必对信息进行任何处理：

```
class ColdCallFormatException : ApplicationException
{
    public ColdCallFormatException(string message)
        : base(message)
    {
    }

    public ColdCallFormatException(
        string message, Exception innerException)
        : base(message, innerException)
    {
    }
}
```

最后是 `UnexpectedException`, 它看起来与 `ColdCallFormatException` 是一样的：

```
class UnexpectedException : ApplicationException
{
    public UnexpectedException(string message)
        : base(message)
    {
    }

    public UnexpectedException(string message, Exception innerException)
        : base(message, innerException)
    {
    }
}
```

下面准备测试该程序。首先，使用 `people.txt` 文件，其内容已经在前面列出了。它有 4 个名字 (与文件中第一行给出的数字匹配)，包括一个间谍。接着，使用下面的 `people2.txt` 文件，它有一个明显的格式错误：

49

```
George Washington
Benedict Armold
John Adams
```

### 359

第 部分 C# 语言

Thomas Jefferson

最后，使用该例子，但指定一个不存在的文件名 `people3.txt`，对这 3 个文件名运行程序 3 次，得到的结果如下：

**SolicitColdCall**

```
Please type in the name of the file containing the names of the people to
be cold
```

```
called > people.txt
```

```
George Washington
```

```
Sales spy found, with name Benedict Armold
```

```
John Adams
```

```
Thomas Jefferson
```

```
All callers processed correctly
```

**SolicitColdCall**

```
Please type in the name of the file containing the names of the people to
be cold
```

```
called > people2.txt
```

```
George Washington
```

```
Sales spy found, with name Benedict Armold
```

```
John Adams
```

```
Thomas Jefferson
```

```
The file people2.txt appears to have been corrupted
```

```
Details of the problem are: Not enough names
```

**SolicitColdCall**

```
Please type in the name of the file containing the names of the people to
be cold
```

```
called > people3.txt
```

The file people3.txt does not exist

这个小应用程序演示了处理程序中可能存在的错误和异常的许多不同方式。

## 14.2 小结

本章介绍了 C# 通过异常处理错误情况的机制，我们不仅可以输出代码中的一般错误代码，还可以用指定的方式处理最特殊的错误情况。有时一些错误情况是通过 .NET Framework 本身提供的，有时则需要编写自己的错误情况，如本章的例子所示。在这两种情况下，都可以采用许多方式来保护应用程序的工作流，使之不出现不必要的和危险的错误。

# 第 15 章 Visual Studio 2008

我们已经熟悉了 C# 语言，下面开始学习本书的应用部分，这部分将介绍如何使用 C# 编写各种应用程序。在此之前，需要了解如何使用 Visual Studio 以及 .NET 环境提供的一些功能来编写程序。

本章将介绍在 .NET 环境中编程的真正含义，讨论 Visual Studio，这是主要的开发环境，在该环境中可以编写、编译、调试和优化 C# 程序。本章还提供编写良好应用程序的规则。Visual Studio 是用于编写 Web 窗体、Windows 窗体、XML Web 服务的主要 IDE。Windows 窗体及用户界面代码的编写详见第 28 章。

本章还介绍建立面向 .NET Framework 3.0 的应用程序的步骤。通过 .NET Framework 3.0 类库提供的这类应用程序包括 Windows Presentation Foundation (WPF)、Windows Communication Foundation (WCF) 和 Windows Workflow Foundation (WF)。

## 15.1 使用 Visual Studio 2005

Visual Studio 2005 是一个全面集成的开发环境，用于编写、调试代码，把代码编译为程序集进行发布。实际上，Visual Studio 提供了一个非常专业的多文档界面应用程序，在该应用程序中可以进行与开发代码相关的任何操作，它提供了：

文本编辑器：在文本编辑器中，可以编写 C# 代码（以及 VB 2005、J# 和 C++ 代码）。这个文本编辑器相当复杂，例如，在键入语句时，它可以自动布局代码，如缩进代码行、匹配代码块的首尾括号、提供彩色编码的关键字等。在键入语句时，它还能执行一些语法检查，给可能产生编译错误的代码加上下划线，这也称为设计期间的调试。它还提供了 IntelliSense 功能。在开始键入时，IntelliSense 会自动显示类、字段或方法名。在开始键入方法的参数时，IntelliSense 也会显示可用重载方法的参数列表。下面的屏幕图 14-1 显示了这个功能，此时操作的是一个 .NET 基类 `ListBox`。

第 部分 Visual Studio

注意：

当 IntelliSense 列表框因某种原因不可见时，请按下快捷键 `Ctrl+Space`，可以在需要时打开 IntelliSense 列表框。

设计视图编辑器：它可以在项目中可视化地放置用户界面和数据访问控件。此时，Visual Studio 会自动在源文件中添加必要的 C# 代码，在项目中实例化这些控件（在 .NET 中，所有的控件实际上都是基类的实例）。

支持窗口：它们可以查看和修改项目的各个方面，例如，这些窗口可以显示源代码中的类以及 Windows 窗体和 Web 窗体类中的可用属性（和它们的初始值）。也可以使用这些窗口指定编译选项，例如代码需要引用哪些程序集。

图 14-1

在环境中编译：可以只选择一个菜单选项编译项目，而不必在命令行上运行 C# 编译器。Visual Studio 会调用 C# 编译器，把所有的相关命令行参数传递给编译器，例如要引用的程序集和要生成什么类型的程序集（例如可执行文件或库 .dll）。Visual Studio 还可以直接运行编译好的可执行文件，用户可以查看这些文件的运行情况是否正常，甚至可以选择不同的编译配置，例如，编译为发布版本或调试版本。

集成的调试程序：代码在第一次运行时，一般不会正确执行。也许在第二次、第三次才能正确运行。Visual Studio 无缝地链接到一个调试程序上，可以在该调试环境中设置断点，观察变量。

## 364

### 第14 章 Visual Studio 2005

集成的 MSDN 帮助：Visual Studio 可以在 IDE 中调用 MSDN 文档说明。例如，在文本编辑器中，如果不能确定某个关键字的含义，可以选择它，按下 F1 键，Visual Studio 就打开 MSDN，显示相关的主题。同样，如果不知道某个编译错误是什么意思，可以选择错误消息，按下 F1 键，打开 MSDN，系统就会显示该错误的信息。

访问其他程序：Visual Studio 还能调用许多其他工具来查看和修改计算机或网络的一些内容，而无需退出开发环境。利用这些工具，可以检查运行服务和数据库连接，直接查询 SQL Server 表，甚至打开 Internet Explorer 窗口，浏览 Web。当然，如果用户很熟悉 C++ 或 VB，就应很熟悉 Visual Studio 6 版本的开发环境，因此上面列出的许多特性就不是什么新内容了。Visual Studio 把以前在 Visual Studio 6 开发环境中可以使用的所有特性都组合起来，无论用户以前在 Visual Studio 6 中使用什么语言，在 Visual Studio 中都会发现一些新增功能。例如，在 Visual Basic 环境中，不能分别编译调试版本和发布版本。另一方面，如果用户有 C++ 编程经验，现在开始使用 C#，就可以获得许多数据访问支持，还可以通过单击把控件拖放到应用程序上，这些功能 Visual Basic 开发人员已经使用很久了，而在 C++ 中就是新增功能。在 C++ 开发环境中，只有最常用的用户界面控件才支持拖放操作。

#### 提示：

对于有 C++ 编程经验的人来说，Visual Studio 2005 去除了 Visual Studio 6 中的两个功能：edit-and-continue 调试和集成的配置器。Visual Studio 2005 还不包含功能完善的配置器应用程序。但在 System.Diagnostics 命名空间中有许多 .NET 类能帮助进行配置。perfmon 配置工具可以在命令行上使用（仅键入 perfmon），它有许多与 .NET 相关的新性能监视器。无论用户有什么编程背景，都会发现与 Visual Studio 6 相比，Visual Studio 2005 开发环境已经有了整体上的改进，包括增加一些新功能，一个跨语言的 IDE 和与 .NET 的集成，菜单和工具栏有一些新选项，许多选项都是 Visual Studio 6 已有的，但重新进行了命名。所以，用户需要花一些时间熟悉 Visual Studio 2005 的布局和命令。

Visual Studio 2002/2003 和 Visual Studio 2005 的区别仅限于 Visual Studio 2005 中几个新增的特性。在 Visual Studio 2005 中，最大的变化是可以修订代码，提取方法，对变量进行全局的重命名，对整个代码体执行其他快速操作。

在安装 Visual Studio 2005 时，会注意到一个最大的变化：这个新的 IDE 与 .NET Framework 2.0 一起工作。实际上，在安装 Visual Studio 2005 时，如果还没有安装 .NET Framework 2.0，就会自动安装它。Visual Studio 2005 不能与 .NET Framework 1.0 或 1.1 一起工作，如果仍要开发 1.0 或 1.1 版本的应用程序，就应在机器上安装 Visual Studio 2002 或 2003。安装 Visual Studio 2005 时，会安装 Visual Studio 的一个完整的新副本，但不会升级原来的 Visual Studio 2002 或 2003 IDE。Visual Studio 的这三个版本可以并行运行。如果试图使用 Visual Studio 2005 打开 Visual Studio 2002 或 2003 项目，IDE 就会提出警告：如果继续，会打开 Visual Studio Conversion Wizard（如图 14-2 所示），将该项目升级到 Visual Studio 2005。

升级向导从 Visual Studio 2003 迁移到 Visual Studio 2005 时进行了巨大的改进。这个新

向导可以对解决方案进行备份复制(如图 14-3 所示), 还可以备份源控件中包含的解决方案。

### 365

第 部分 Visual Studio

图 14-2

图 14-3

还可以在转换过程的最后一步生成一个转换报告。这个报告可以在 Visual Studio 的文档窗口中直接查看, 如图 14-4 所示(进行了一个简单的转换)。

在把解决方案从 Visual Studio 2003 升级到 Visual Studio 2005 之前, 应先在环境中测试程序, 确保应用程序不会因 .NET Framework 1.0/1.1 和 2.0 版本之间的变化而受到影响。如果安装了 .NET Framework 3.0, 应用程序在从 .NET Framework 2.0 升级到 3.0 时, 不需要完成转换过程。

本书适合于专业人员使用, 所以不会详细介绍 Visual Studio 2005 中的每个功能和菜单项。用户应自己熟悉该开发环境。介绍 Visual Studio 的主要目的是让用户熟悉建立和调试 C# 应用程序所涉及的所有概念, 这样才能更好地使用 Visual Studio 2005。图 14-5 显示了 Visual Studio 2005 的外观(注意, Visual Studio 的外观是高度可定制的, 在启动该开发环境时, 看到的可能是位置不同的窗口或内容不同的窗口)。

### 366

第14 章 Visual Studio 2005

图 14-4

图 14-5

下面几节将创建、编写和调试项目, 查看 Visual Studio 在每个阶段所能完成的操作。

### 367

第 部分 Visual Studio

#### 15.1.1 创建项目

安装好 Visual Studio 2005 后, 就可以开始编写第一个项目了。在 Visual Studio 中, 很少从一个空白文件开始, 从头键入 C# 代码, 就像本书前面章节那样(当然, 如果确实要从头开始编写代码, 或者创建一个包含许多项目的解决方案, 该 IDE 也提供了空应用程序项目选项)。编写项目的方式一般是先告诉 Visual Studio 要创建什么类型的项目, 然后 Visual Studio 会自动生成文件和 C# 代码, 给出该类型项目的基本框架。接着, 用户就可以在其中添加自己的代码了。例如, 如果要编写一个基于 Windows GUI 界面的应用程序(在 .NET 中, 这称为 Windows 窗体), Visual Studio 就会建立一个文件, 其中包含的 C# 源 \_\_\_\_\_ 代码创建了一个基本窗体, 这个窗体可以与 Windows 通信, 接收事件。它还可以最大化、最小化、重新设置大小, 用户只需在其中添加需要的控件和功能。如果应用程序要设计为命令行工具(控制台应用程序), Visual Studio 就会提供基本的命名空间、类和 Main() 方法。

最后, 在创建项目时, Visual Studio 还设置了提供给 C# 编译器的编译选项——表示项目是编译为命令行应用程序、库, 还是编译为 Windows 应用程序。它还告诉编译器需要引用的基类库(Windows GUI 应用程序需要引用许多与 Windows.Forms 相关的库, 控制台应用程序则不需要)。当然如果必要, 用户可以在编辑时, 修改这些设置。

在第一次启动 Visual Studio 时, 出现的窗口称为 Start Page, 如图 14-6 所示。这个 Start Page 是一个 HTML 页面, 其中包含各种链接, 通过它们可以进入有用的网站, 打开现有的项目, 或者启动一个新项目。

图 14-6

### 368

第14 章 Visual Studio 2005

图 14-6 显示了使用 Visual Studio 2005 打开的 Start Page, 其中有一个最近编辑的项目列表。单击其中的一个项目就可以打开它。

1. 选择项目类型

创建新项目时，可以在 Visual Studio 菜单上选择 File| New Project，打开 New Project 对话框，如图 14-7 所示，其中给出了可以创建的各种项目。

图 14-7

使用该对话框，可以选择 Visual Studio 为用户生成的某种初始框架文件和代码、编译选项，以及编译代码所使用的编译器：Visual C# VB 2005 Visual J# 或 Visual C++ 编译器。从这里可以看出，Microsoft 为 .NET 提供了多种语言集成。对于本例，我们选择了 C# 控制台应用程序。

注意：

这里不打算介绍不同类型的项目的所有选项。在 C# 方面，Visual Studio .NET 可以创建所有旧的 C++ 项目类型—— MFC 应用程序、 ATL 项目等。在 VB 2005 方面，选项有一些变化，例如，可以创建 VB 2005 命令行应用程序（控制台应用程序）、.NET 组件（类库）或者 .NET 控件（Windows 控件库），但不能创建基于 COM 的旧风格的控件（.NET 控件可以取代这种 ActiveX 控件）。

表 14-1 列出了 Visual C# Projects 下所有可用的选项。注意，在 Other Projects 选项下还有一些比较专业的 C# 模板项目。

369

第 部分 Visual Studio

表 14-1

如果 选 择 得到的 C# 代码和编译选项将生成  
Windows Application 响应事件的基本空窗体  
Class Library 可以由其他代码调用的 .NET 类  
Windows Control Library 可以由其他代码调用的 .NET 类，它有一个用户界面（类似于旧风格的 ActiveX 控件）  
ASP.NET Web Application 基于 ASP.NET 的网站：ASP.NET 页面和 C# 类生成的、从页面发送给浏览器的 HTML 响应  
ASP.NET Web Service 用作功能全面的 Web 服务的 C# 类  
ASP.NET Mobile Web Application 允许建立面向移动设备的 ASP.NET 页面的应用程序类型  
Web Control Library 可以由 ASP.NET 页面调用的控件，在浏览器上显示这个控件时，可生成给出该控件外观的 HTML 代码  
Console Application 在命令行提示符上或控制台窗口中运行的应用程序  
Windows Service 在 Windows NT 和 Windows 2000 的后台运行的服务  
Crystal Reports Windows Application 该项目用于创建带有 Windows 用户界面和 Crystal Report 示例的 C# 应用程序  
SQL Server Project 该项目用于创建在 SQL Server 中使用的类  
Pocket PC 2003 Application 该项目用于为 Pocket PC 2003 及以后版本创建 .NET Compact Framework 2.0 窗体应用程序  
Pocket PC 2003 Class Library 该项目用于为 Pocket PC 2003 及以后版本创建 .NET Compact Framework 2.0 类库（.dll）  
Pocket PC 2003 Control Library 该项目用于为 Pocket PC 2003 及以后版本创建 .NET Compact Framework 2.0 控件  
Pocket PC 2003 Console Application 该项目用于为 Pocket PC 2003 及以后版本创建 .NET Compact Framework 2.0 的非图形化应用程序  
Pocket PC 2003 Empty Project 用于为 Pocket PC 2003 及以后版本创建 .NET Compact Framework 2.0 应用程序的空项目  
Smartphone 2003 Application 该项目用于为 Smartphone 2003 及以后版本创建 .NET Compact Framework 1.0 窗体应用程序  
Smartphone 2003 Class Library 该项目用于为 Smartphone 2003 及以后版本创建 .NET

Compact Framework 1.0 类库 (.dll)

Smartphone 2003 Console Application 该项目用于为 Smartphone 2003 及以后版本创建 .NET

Compact Framework 1.0 的非图形化应用程序

Smartphone 2003 Empty Project 用于为 Smartphone 2003 及以后版本创建 .NET Compact Framework 1.0 应用程序的空项目

## 370

第14 章 Visual Studio 2005

(续表 )

如果 选 择 得到的C#代码和编译选项将生成

Windows CE 5.0 Application 该项目用于为 Windows CE 5.0 及以后版本创建 .NET Compact Framework 2.0 窗体应用程序

Windows CE 5.0 Class

Library

该项目用于为 Windows CE 5.0 及以后版本创建 .NET Compact Framework 2.0 类库 (.dll)

Windows CE 5.0 Control

Library

该项目用于为 Windows CE 5.0 及以后版本创建 .NET Compact Framework 2.0 控件

Windows CE 5.0 Console

Application

该项目用于为 Windows CE 5.0 及以后版本创建 .NET Compact Framework 2.0 命令行应用程序

Windows CE 5.0 Empty

Project

用于为 Windows CE 5.0 及以后版本创建 .NET Compact Framework

2.0 应用程序的空项目

Excel Workbook 该项目用于为新的或已有的 Excel 2003 工作簿创建托管代码扩展

Word Document 该项目用于为新的或已有的 Word 2003 文档创建托管代码扩展

Excel Template 该项目用于为新的或已有的 Excel 2003 模板创建托管代码扩展

Word Template 该项目用于为新的或已有的 Word 2003 模板创建托管代码扩展

Empty Project 没有任何代码。必须从头编写所有的代码，但在编写时仍拥有 Visual Studio 提供的全部功能

Empty Web Project 与 Empty Project 一样，但设置了编译选项，告诉编译器为 ASP.NET

页面生成代码

New Project In Existing

Folder

新的空项目文件。如果有一些 C#源代码 (例如，在文本编辑器中

键入的代码 )并要把它们转换为 Visual Studio 项目，就可以使用这

个选项

这个列表并不完整，本章在后面将补充 .NET Framework 3.0 项目。还有各种类型的初学者工具，例如屏幕保护工具、电影收集工具等。另外，还有一个测试应用程序可以创建包含测试的项目。

### 2. 新建的控制台项目

在上述对话框中选择 Console Application 选项，单击 OK 按钮，Visual Studio 就会提供许多文件，包括一个源文件 Program.cs，其中包含了最初的框架代码。图 14-8 显示了 Visual Studio 编写的代码。

可以看出，这是一个 C# 程序，但它实际上没有做任何工作，只是包含了 C# 可执行程序所必需的基本项：一个命名空间和一个包含 Main() 方法的类，其中 Main() 方法是程序的入口点。（严格说来，命名空间是不必要的，但不声明命名空间是一种不好的编程习惯）。按下 F5 键，或者选择 Debug 菜单中的 Start，这段代码就可以编译和运行。在这样做之前，

**371**

第 部分 Visual Studio

在程序中加入一行代码，让应用程序完成某个工作：

图 14-8

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the editors at Wrox Press");
}
```

如果编译并运行了该项目，就会显示一个控制台窗口，但该窗口几乎立即就消失了，用户几乎看不到输出的信息。原因是在创建该项目时，Visual Studio 记住了用户指定的设置，所以会把它编译并运行为控制台应用程序。然后，Windows 知道需要运行一个控制台应用程序，但没有运行该程序的控制台窗口。所以，Windows 就创建一个控制台窗口，并运行该程序。只要程序退出，Windows 就认为不再需要该控制台窗口，因此就即时删除了它。这些都是非常逻辑化的操作，但如果希望能看到项目的输出结果，这些操作对用户就没有什么帮助。

要避免这个问题，可以在 Main() 方法结束前插入下述代码：

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the editors at Wrox Press");
    Console.ReadLine();
}
```

这样，代码运行后，会显示其输出结果，之后执行 Console.ReadLine() 语句，用户按下

**372**

第14 章 Visual Studio 2005

回车键后，程序退出。这表示在用户按下回车键之前，控制台窗口一直会挂起。

注意这仅是在 Visual Studio 中试运行控制台应用程序的问题。如果编写的是一个 Windows 应用程序，该应用程序显示的窗口会自动停留在屏幕上，直到用户显式退出程序为止。同样，如果在命令行提示符上运行一个控制台应用程序，就没有窗口消失的问题。

### 3. 其他文件的创建

Program.cs 源代码文件不是 Visual Studio 创建的唯一文件。如果查看一下 Visual Studio 创建项目的文件夹，就会看到其中不仅有 C# 文件，还有如图 14-9 所示的完整目录结构。

图 14-9

文件夹 bin 和 obj 存储编译好的文件和中间文件，obj 的子文件夹存储各种临时或中间文件，bin 的子文件夹存储编译好的程序集。

注意：

传统上，VB 开发人员只是编写它们的代码，然后运行它们。代码在发布前，必须编译成可执行文件，但 VB 在调试中隐藏了这个过程。而在 C# 中，这个过程是显式的：要运行代码，必须先编译它，即在某处创建一个程序集。

在项目的主文件夹 ConsoleApplication1 中，剩余的文件都是由 Visual Studio 建立的，它们包含项目的信息（例如它包含的文件），这样，Visual Studio 就知道如何编译项目，在下一次打开该项目时，知道如何读取它。

**373**

第 部分 Visual Studio

### 15.1.2 解决方案和项目

项目和解决方案的一个重要区别是：

项目是一组要编译到单个程序集(在某些情况下，是单个模块)中的所有源文件和资源。例如，项目可以是类库，或一个Windows GUI应用程序。

解决方案是构成某个软件包(应用程序)的所有项目集。

为了说明这个区别，考虑一下在发布一个项目(该项目包含多个程序集)时的情况。例如，其中可能有一个用户界面，有某些定制控件和其他组件，它们都作为应用程序的库文件一起发布。不同的管理员甚至还有不同的用户界面。应用程序的不同部分都包含在一个独立的程序集中，因此，在Visual Studio看来，它们都是独立的项目。但我们要同时编写这些项目，使它们彼此连接起来。所以，把它们当作一个单元来编辑就很重要。在Visual Studio中，可以把所有的项目看作一个解决方案，把该解决方案当作是可以读入的单元，并允许用户在其上工作。

前面讨论了如何创建一个控制台项目。实际上，在前面的例子中，Visual Studio创建的是一个解决方案，这个解决方案只包含一个项目。可以在Visual Studio的一个窗口中查看它，该窗口称为Solution Explorer，它包含一个定义解决方案的树形结构，如图14-10所示。

图 14-10

图14-10说明了项目包含源文件Program.cs和另一个C#源文件AssemblyInfo.cs(在文件夹Properties中)，AssemblyInfo.cs包含程序集的描述信息和指定的版本信息(该文件详见第16章)。Solution Explorer也指定了项目通过命名空间引用的程序集。扩展Solution Explorer中的文件夹Reference，就可以看到它。

如果没有改变Visual Studio的任何默认设置，Solution Explorer就在屏幕的右上角。如果看不到它，可以选择View菜单中的Solution Explorer。

解决方案用扩展名为.sln的文件来表示，在本例中，就是ConsoleApplication1.sln。该项目由主文件夹中的各个文件来表示。如果试图使用Notepad编辑这些文件，就会发现它们大多数都是纯文本文件，为了与.NET和依赖于开放标准的.NET工具保持一致，它们大都是XML格式。

**374**

第14章 Visual Studio 2005

注意：

C++开发人员应认识到，Visual Studio解决方案对应于旧的C++项目工作区(存储在.dsw文件中)，Visual Studio项目对应于旧的C++项目(.dsp文件)。另一方面，VB开发人员应注意，解决方案对应于旧的VB项目组(.vbg文件)，.NET项目对应于旧的VB项目(.vbp文件)。Visual Studio与旧VB IDE的区别是，Visual Studio总是自动创建一个解决方案。在Visual Studio 6中，VB开发人员最初会得到一个项目，如果要得到项目组，就必须在IDE中显式指定。

#### 1. 给解决方案添加另一个项目

下面几节将说明Visual Studio如何处理Windows应用程序和控制台应用程序。为此，本节将创建一个Windows项目BasicForm，并把它添加到当前的解决方案ConsoleApplication1中。

注意：

最终我们将得到包含一个Windows应用程序和一个控制台应用程序的解决方案。这种情况并不常见——我们一般会在解决方案中包含一个应用程序和多个库，但这个解决方案可以演示更多的代码。如果用户编写的工具需要作为Windows应用程序和命令行工具来运行，就可以创建这样的解决方案。

创建新项目可以使用两种方式。一是进入File菜单，选择New Project选项(如前所述)。二是在File菜单中选择Add | New Project。如果从菜单中选择New Project选项，会打开熟悉的New Project对话框，但这次Visual Studio要在已有的ConsoleApplication1项目位置中创建新项目，如图14-11所示。

图 14-11

如果选中这个选项，就会添加一个新项目，ConsoleApplication1 解决方案现在就应包含一个控制台应用程序和一个Windows 应用程序。

### 375

第 部分 Visual Studio

注意：

为了保持 Visual Studio 的语言无关性，新项目不一定是 C# 项目，在同一个解决方案中，可以有 C# 项目、VB 2005 项目或 C++ 项目。但这里仍使用 C#，因为这是一本介绍 C# 的书。当然，ConsoleApplication1 对于这个解决方案来说，不再是一个合适的名称了。右击该名称，从弹出的菜单中选择 Rename，改变其名称。如果把它重命名为 DemoSolution，Solution Explorer 窗口就应如图 14-12 所示。

图 14-12

从这里可以看出，Visual Studio 会自动让新增的 Windows 项目引用那些对 Windows 窗体的功能来说非常重要的某些附加基类。

如果查看一下 Windows 资源管理器，就会发现解决方案的文件名已经改为 Demo-Solution.sln。一般情况下，如果要重新命名文件，最好在 Solution Explorer 中进行，因为 Visual Studio 会自动更新其他项目文件对该文件的引用。如果使用 Windows 资源管理器重命名文件，就会中断解决方案，因为 Visual Studio 不能定位它需要读取的所有文件，用户必须手工编辑项目和解决方案文件，以更新文件引用。

### 2. 设置启动项目

如果在解决方案中有多个项目，就必须确保该解决方案在某一刻只运行一个项目。在编译解决方案时，将编译其中的所有项目。但在按下 F5 键或选择 Start 时，必须告诉 Visual Studio 先运行哪个项目。如果有多个可执行文件，它调用了几个库，显然应先运行这个可执行文件。对于本例，项目中有两个独立的可执行文件，就必须逐个调试它们。

可以告诉 Visual Studio 应运行哪个项目，方法是在 Solution Explorer 中右击该项目，在弹出的菜单中选择 Set as Startup Project。这就告诉 Visual Studio .NET 哪个项目是当前的启动项目，因为它在 Solution Explorer 中是黑体显示，在图 14-12 上，就是 WindowsApplication1。

### 376

第14 章 Visual Studio 2005

#### 15.1.3 Windows 应用程序代码

在 Visual Studio 第一次创建应用程序时，Windows 应用程序包含的启动代码要比控制台应用程序多，因为创建一个窗口是一个比较复杂的过程。第 28 章将详细讨论 Windows 应用程序的代码，这里只给出 WindowsApplication1 项目中类 Form1 的代码，说明自动生成的代码有多少。

#### 15.1.4 读取 Visual Studio 6 项目

利用 C# 编写代码时，不需要读取旧的 Visual Studio 6 项目，因为 C# 代码在 Visual Studio 6 中不存在。但是，语言的互操作性是 .NET Framework 的一个重要部分，C# 代码可能要与 VB 代码或 C++ 代码一起使用。此时就需要编辑用 Visual Studio 6 创建的项目了。

Visual Studio 可以读取和升级 Visual Studio 6 项目和工作区，这不同于 C++、VB 和 J++ 项目。

在 Visual C++ 中，不需要修改源代码。所有的旧 C++ 代码使用新的 C++ 编译器仍可正常工作。显然它们不是托管代码，但仍可以编译为运行在 .NET 运行库外部的代码。如果代码要与 .NET Framework 集成起来，就需要编辑它。如果要让 Visual Studio 读取旧的 Visual C++ 项目，则只需增加一个新的解决方案文件和更新的项目文件。原来的 .dsw 和 .dsp 文件不变，这样，该项目就可以在需要时用 Visual Studio 6 编辑了。

对于 Visual Basic，就比较复杂了。如第 1 章所述，VB 2005 的设计与 VB 6 非常类

似，也有许多相同的语法，但在许多方面，它是一种新的语言。在 Visual Basic 6 中，源代码大都由控件的事件处理程序组成，实际上，实例化主窗口和许多控件的代码并不是 Visual Basic 代码的一部分，而是隐藏在后台中，是项目配置的一部分。而 Visual Basic 2005 则与 C# 的工作方式相同，把整个程序都放在源文件中，即显示主窗口和所有控件的代码都必须放在源文件中。与 C#一样，Visual Basic 2005 要求所有的代码都是面向对象的，是类的一部分，而 Visual Basic 6 则没有 .NET 中类的概念。如果要使用 Visual Studio 读取 Visual Basic 6 项目，在处理前必须把全部源代码升级为 Visual Basic 2005，这会对 Visual Basic 6 代码做许多修改。在很大程度上，Visual Studio 可以自动进行修改，创建一个新的 Visual Basic 2005 解决方案，它提供的源代码与原来的 VB6 代码大不相同，需要仔细检查生成的代码，以确保项目仍能正常工作。甚至还要找出 Visual Studio 注释掉的某些代码块，因为 Visual Studio 不能确定这些代码做什么工作，这部分代码必须手工编辑。

对于 Microsoft 来说，J++ 现在是一种过时的语言，.NET 不直接支持它。但为了让已有的 J++ 代码仍可继续使用，有几个工具可以让 J++ 代码在 .NET 中使用。Visual Studio 2005 包括 J# 开发环境，它可以处理 J++ 代码。还有一种可以自动把 J++ 代码转换为 C# 代码的工具，该工具类似于从 VB 6 到 VB 2005 的升级工具。这些工具组合为一个软件 JUMP (Java User Migration Path)，在编写本书时，它们还没有附

**377**

第 部分 Visual Studio

在 .NET 或 Visual Studio 中，但其详细情况可在通过

<http://msdn.microsoft.com/vjsharp/jump/default.asp> 获取。

### 15.1.5 项目的浏览和编码

本节将介绍在给项目添加代码时，Visual Studio 所提供的一些功能。

#### 1. 可折叠的编辑器

Visual Studio 的一个重要改进是，它把可折叠的编辑器当作默认的代码编辑器，如图 14-13 所示。

图 14-13

图 14-13 显示了前面生成的控制台应用程序的代码。但要注意窗口左边的小减号图标，它们标记出了编辑器认为是新代码块（或文档注释）的起点。单击这些图标，可以关闭对应的代码块视图，就像关闭树形控件中的节点一样，如图 14-14 所示。

在编辑时，可以只考虑要查看的那些代码块，关闭目前不想查看的代码块。而且，如果不喜欢单独关闭代码块的方式，可以用 C# 预处理器指令 #region 和 #endregion (详见本书前面的章节) 指定另一种方式。例如，假定要折叠 Main() 方法中的代码，可以先添加如图 14-15 所示的代码。

**378**

第14 章 Visual Studio 2005

图 14-14

图 14-15

代码编辑器会自动检测 #region 块，通过 #region 指令放置一个新的减号图标，如图 14-15 所示，以便关闭该代码区域。把这个代码块放在一个区域中，就可以用在 #region 指令中指

**379**

第 部分 Visual Studio

定的注释标记该区域，让编辑器关闭该代码块，如图 14-16 所示。但编译器会忽略这些指令，按正常情况编译 Main() 方法。

图 14-16

除了可折叠的编辑器之外，Visual Studio 的代码编辑器还拥有 Visual Studio 6 的全部功能，特别是它支持 IntelliSense 功能，这不仅减少了键入代码的工作量，还可以确保输入正

确的参数。C++开发人员会注意到，Visual Studio 的 IntelliSense 功能比 Visual Studio 6 的更强大，速度也更快。IntelliSense 功能在 Visual Studio 2005 中也得到了进一步的改进。它的智能化更强，可以记住用户喜欢的选项，并从这些选项开始，而不是从该功能提供的有时很长的列表的开头开始。

代码编辑器也对代码进行一些语法检查，在编译代码前用短波浪线划出大多数语法错误。把鼠标放在有下划线的文本上面，就会弹出一个小文本框，解释错误。这个功能VB 开发人员已使用了多年，叫做设计期间的调试功能，现在 C 和 C++ 开发人员也可以使用它了。

## 2. 其他窗口

除了代码编辑器外，Visual Studio 还提供了许多其他窗口，允许用户以不同的角度查看项目。

注意：

本节的其余部分将介绍许多其他的窗口。如果某个窗口在屏幕上不可见，可以进入 View 菜单，单击合适窗口的名称。要显示设计视图和代码编辑器，可以在 Solution Explorer 380

第14 章 Visual Studio 2005

中右击文件名，然后从弹出的窗口中选择 View Designer 或 View Code，也可以从 Solution Explorer 顶部的工具栏中选择。设计视图和代码编辑器共用同一个窗口。

### (1) Design View 窗口

如果设计一个用户界面应用程序，例如 Windows 应用程序、Windows 控件库或者 ASP.NET 应用程序，就可以使用设计视图 (Design View) 窗口，它会显示窗体的整体外观。设计视图窗口一般和工具箱窗口一起使用。工具箱包含许多可以拖放到程序中的 .NET 组件，如图 14-17 所示。

图 14-17

应用工具箱的规则与 Visual Studio 6 中所有的开发环境一样，但在 .NET 中，工具箱中的组件数量大大增加了。组件的种类在某种程度上取决于用户所编辑项目的类型。例如，在编辑 DemoSolution 解决方案中的 WindowsApplication1 项目时，可以使用的组件种类就比编辑 ConsoleApplication1 项目时多。最重要的组件种类如下：

数据访问组件：可以连接数据源，管理它们包含的数据。这类组件可处理 Microsoft SQL Server、Oracle 和 OleDb 数据源。

Windows Forms 控件 (标记为常用控件)：表示可视化控件，例如文本框、列表框和树形视图，用于处理胖客户应用程序。

381

第 部分 Visual Studio

Web Forms 控件 (标记为标准)：基本上与 Windows 控件的作用一样，但用于 Web 浏览器，把模拟控件的 HTML 输出结果发送给浏览器。

杂项组件：在机器上执行各种有用任务的 .NET 类，例如连接目录服务或事件日志。也可以把自己定制的组件类别添加到工具箱中，方法是右击任一类别，从弹出的菜单中选择 Add Tab。从该菜单中选择 Choose Items，就可以把其他工具放在工具箱中。这非常适合于添加自己喜欢的 COM 组件和 ActiveX 控件。在默认情况下，COM 组件和 ActiveX 控件不会显示在工具箱中。如果要添加一个 COM 控件，可以单击它，并拖放到项目上，就像操作 .NET 控件一样。Visual Studio 会自动添加所有必需的 COM 交互操作代码，以便项目调用该控件。此时，添加到项目中的实际上是 Visual Studio 在后台创建的一个 .NET 控件，它是所选 COM 控件的容器。

注意：

C++ 开发人员可能会把工具箱当作资源编辑器的 Visual Studio 版本。VB 开发人员刚开始不会认为该工具箱是新增的内容，因为在 Visual Studio 6 中就有工具箱。但是，Visual

Studio 中的工具箱对源代码的作用与 VB6 IDE 中的工具箱有显著的不同。

为了说明工具箱如何工作，把一个文本框放在基本窗体项目上。首先单击工具箱中的 TextBox 控件，再单击一次，把它放在设计视图的窗体上（也可以直接把它拖放到设计界面上）。设计视图如图 14-18 所示，该图也是编译并运行 WindowsApplication1 项目的结果。

图 14-18

在窗体的代码视图上，Visual Studio 2005 并没有像 IDE 的以前版本那样，实例化放在窗体上的 TextBox 对象。单击 Visual Studio Solution Explorer 中 Form1.cs 文件旁边的加号，

**382**

第14 章 Visual Studio 2005

会看到一个文件 Form1.Designer.cs，它用于窗体及其中控件的设计。在这个类文件中，Form1 类有一个新的成员变量：

```
public class Form1
{
    private System.Windows.Forms.TextBox textBox1;
```

在方法 InitializeComponent() 中还有一些初始化代码，该方法从 Form1 构造函数中调用：

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(13, 13);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(100, 20);
    this.textBox1.TabIndex = 0;
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(292, 273);
    this.Controls.Add(this.textBox1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
}
```

在某一方面，代码编辑器和设计视图是没有区别的，它们只是呈现了相同代码的不同视图。在设计视图上单击并添加 TextBox 时，编辑器会把上述代码放在 C# 源文件中。设计视图反映了这个变化，因为 Visual Studio 可以读取源代码，确定在应用程序启动时窗体中会有哪些控件。与 VB 查看控件的方式相比，这是一个重要的改变，在 VB 中，所有的控件都放在可视化的设计视图中。现在，由 C# 源代码控制应用程序，设计视图只是显示源代

码的另一种方式。如果使用 Visual Studio 编写 VB 2005 代码，也要遵循这个规则。  
还可以采用另外一种方式，如果把上述代码手工添加到 C# 源文件中，Visual Studio 也会自动从代码中检测到应用程序中有一个文本框控件，并会在设计视图的指定位置显示它。  
最好可视化地添加这些控件，让 Visual Studio 处理最初生成的代码，单击鼠标两次要比键入好几行代码快得多，也不容易出错！

### 383

#### 第 部分 Visual Studio

可视化添加这些控件的另一个原因是为了确定应用程序中有这些控件，Visual Studio 需要使相关的代码遵循某些条件——手工编写的代码可能不遵循这些条件。特别是 Initialize- Component() 方法包含初始化文本框的代码，在它的注释中警告用户不要修改代码。这是因为 Visual Studio 要查找这个方法，以确定应用程序在启动时有哪些控件。如果在代码的其他地方创建和定义了一个控件，Visual Studio 不知道有这个控件，因此就不能在设计视图或其他窗口中编辑它。

实际上，无论有什么警告，都可以在 InitializeComponent() 中修改代码，但应非常小心。例如，修改一些属性的值一般不会出什么问题，如让某个控件显示不同的文本，或者给它设置另一个大小。实际上，开发人员工作室非常擅长于处理在这个方法中添加的其他代码。但要注意，如果对 InitializeComponent() 进行了过多的修改，Visual Studio 就有可能识别不出使用代码添加的控件。在编译代码时，这不会影响到应用程序，但可能禁用 Visual Studio 为这些控件提供的某些编辑功能。因此，如果要进行其他重要的初始化，最好在 Form1 构造函数或其他方法中进行。

#### (2) 属性窗口

这是从旧 VB IDE 继承而来的另一个窗口。本书的第一部分说过，.NET 类可以执行属性。实际上，如第 28 章(讨论 Windows 窗体的建立)所述，表示窗体和控件的 .NET 基类有许多定义其操作和外观的属性。例如 Width Height Enabled(用户是否可以给该控件键入信息) 和 Text(控件所显示的文本)，Visual Studio 知道其中的许多属性。对于 Visual Studio 能通过读取源代码检测到的控件来说，属性窗口可以显示和编辑大多数属性的初值，如图 14-19 所示。

图 14-19

注意：

属性窗口也可以显示事件。单击窗口顶部的闪电图标，就可以查看 IDE 中当前选中的事件，或者查看在属性窗口的下拉列表中选择的事件。

### 384

#### 第14 章 Visual Studio 2005

在属性窗口的顶部，有一个列表框，从中可以选择要查看的控件。在本章的这个例子中，我们选择的是 Form1，即 WindowsApplication1 项目的主窗体类，把其文本编辑为“Basic Form-Hello!”。如果此时查看源代码，就会看到刚才的操作实际上是通过一个友好的用户界面编辑了源代码：

```
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(292, 273);
this.Controls.Add(this.textBox1);
this.Name = "Form1";
this.Text = "Basic Form - Hello";
this.ResumeLayout(false);
this.PerformLayout();
```

并不是属性窗口中的所有属性都会在源代码中显式指定。对于没有显式指定的属性，Visual Studio 会显示在创建窗体时给它们设置的默认值，这些默认值是在初始化窗体时设

置的。显然，如果在属性窗口中修改这些属性的值，源代码中就会出现一个显式设置该属性的语句，反之亦然。有趣的是，如果属性是从其初始值改变而来，这个属性在属性窗口的列表框中就会显示为黑体。有时双击属性窗口中的属性，会返回其初始值。

属性窗口提供了一种查看控件或窗口的外观和属性的简便方式。

注意：

属性窗口实现为一个System.Windows.Forms.PropertyGrid实例，该实例在内部使用第12章介绍的反射技术，来标识要显示的属性和属性值。

图 14-20

### (3) 类视图窗口

与属性窗口不同，类视图窗口最初是从C++(或J++)开发环境继承而来的一个窗口，如图14-20所示。它对于VB开发人员来说是新的，因为VB6不支持类的概念，而使用COM组件。Visual Studio实际上并没有把类视图看作是一个窗口，而是Solution Explorer的一个

385

第部分 Visual Studio

附加选项卡。默认情况下，类视图不会显示在Visual Studio Solution Explorer中。要打开类视图，应选择View | Other Windows | Class View。类视图如图14-20所示，显示了代码中命名空间和类的层次结构，给出了一个树形视图，用户可以展开该视图，以查看哪个命名空间包含了什么类，类包含了什么成员。

类视图的一个特性是，如果右击在源代码中可以访问的任一项，弹出菜单就会提供一个选项Go To Definition，单击它，就可以访问代码编辑器中该项的定义。还可以在类视图中双击该项(实际上是在源代码编辑器中右击需要的项，从打开的弹出菜单中选择相同的选项)来完成同样的操作。弹出菜单还允许给类添加字段、方法、属性或索引器。这意味着可以在一个对话框中指定相关信息，开发环境会自动添加对应的代码。这对于字段和方法可能不太有效，因为它们可以快速添加到代码中，但对于属性和索引器来说就非常有用了，因为它可以减少大量的键入工作。

### (4) 对象浏览器

在.NET环境中编程的一个重要优点是可以确定程序集中引用的基类和其他库中有什么方法和其他代码项。这个功能是通过对象浏览器来获得的。在Visual Studio 2005的View菜单中选择Object Brower，就可以访问这个窗口。

对象浏览器非常类似于类视图窗口，它也显示一个树形视图，该树形视图给出了应用程序的类结构，允许查看每个类的成员。用户界面则略有不同，因为它在一个单独的窗格中显示类成员，而不是在树形视图中显示。但真正的区别是它不仅可以查看项目中的命名空间和类，还可以查看项目所引用的所有程序集中的命名空间和类。图14-21显示了利用对象浏览器查看.NET基类中的SystemException类的情况。

图 14-21

386

第14章 Visual Studio 2005

在对象浏览器中必须注意的一点是，它先按照类所在的程序集对类进行分组，再按照命名空间对类进行分组。因为基类的命名空间常常分布在多个程序集中，所以在定位某个类时可能会遇到麻烦，除非知道该类在哪个程序集中。

对象浏览器可以查看.NET对象。如果由于某些原因要查看已安装的COM对象，就可以使用以前在C++IDE中使用的OLEView工具，它在文件夹C:\Program Files\Microsoft Visual Studio 8\Common7\Tools\Bin中，该文件夹还有几个其他类似的工具。

注意：

VB开发人员不应把.NET对象浏览器和VB 6 IDE的对象浏览器混为一谈，.NET对象浏览器可以查看.NET类，而VB 6中的对象浏览器则可以查看COM组件。如果要使用旧对象浏览器的功能，现在就应使用OLEView工具。

## (5) 服务器浏览器

服务器浏览器可以用于确定编码时计算机各个方面的情况，如图 14-22 所示。

图 14-22

从上面的屏幕图中可以看出，可以通过服务器浏览器访问数据库连接、服务信息和事件日志的信息。

服务器浏览器与属性窗口链接在一起，所以，如果打开 Services 节点，单击某个服务，该服务的属性就会显示在属性窗口中。

### 3. pin 按钮

在浏览 Visual Studio 时，前面介绍的许多窗口包含一些有趣的功能，很容易让人回想起工具栏的一些功能。除了代码编辑器外，它们都是固定的。另一个比较新的功能是在这些窗口处于浮动状态时，在每个窗口右上角的最小化按钮的旁边就会出现一个大头针图标，这个图标的工作方式就跟大头针一样，把窗口钉住。在窗口被钉住时（大头针垂直显示），它们的操作就像一般的窗口一样。但当它们没有被钉住时，（大头针水平显示），只要它们获得焦点，就会打开；而失去焦点时（用户单击了其他地方），它们就会无声无息地退到 Visual Studio 应用程序的主边界上（打开和关闭该按钮时，计算机的速度也有快慢变化）。

钉住或不钉住窗口是充分利用屏幕上有限空间的另一种方式。以前在 Windows 中这个功能用得不多，但在一些第三方应用程序例如 PaintShop Pro 中就使用了类似的概念。钉住的窗口在许多基于 Unix 的系统上已经有了一定的应用。

**387**

第 部分 Visual Studio

### 15.1.6 生成项目

本节介绍 Visual Studio 为生成项目提供的选项。

#### 1. 生成、编译和产生项目

在介绍各种生成选项前，首先澄清一个术语。在把源代码变成某类可执行的代码时，常常会使用 3 个不同的术语：编译、生成和产生。这三个术语的出现是因为直到最近，把源代码变成可执行代码的过程需要多个步骤（在 C++ 中仍是这样）。这在很大程度上是因为一个程序常常包含许多源文件，例如在 C++ 中，每个源文件都需要单独编译。这样就生成了对象文件，每个对象文件都包含一些可执行代码，但每个对象文件都仅与一个源文件相关。为了生成可执行代码，这些对象文件必须链接在一起，这个过程就称为链接。把过程组合起来通常称为（至少在 Windows 平台上）生成代码。但是，在 C# 中，编译器更为专业，能够把所有的源文件当作一个块来读取和处理。因此，就没有独立的链接阶段，在 C# 中，“编译（compile）”和“生成（build）”可以互换。

术语“产生（make）”的基本含义与“生成（build）”相同，但在 C# 中一般不使用。该术语来源于旧的大型计算机系统，在该系统上，当项目由许多源文件组成时，就编写一个独立的文件，其中包含的指令可以指示编译器如何生成项目：包括哪些文件，链接什么库等，这个文件一般称为产生文件（make file），该文件仍然是 Unix 上的标准。产生文件在 Windows 上一般不需要，但如果用户需要，也可以编写它们（或者让 Visual Studio 生成它们）。

#### 2. 调试和发布项目

有 1592 \_f1C++ 背景的开发人员都知道生成文件的概念，但 VB 开发人员就不一定知道了。在调试程序时，可执行代码进行的操作一般与实际发布该软件时所进行的操作大不相同，在发布时，除了代码正常工作外，可执行文件的尺寸应尽可能小，运行速度应尽可能快。但这些要求与调试代码时的要求并不相容。原因如下：

##### (1) 优化

要获得高性能，部分依赖于编译器对代码进行的许多优化，即编译器在编译过程中一直在监视源代码，找出某些代码，以一种不影响全局效果的方式修改它们，使其效率更高。例如，如果编译器遇到下面的源代码：

```
double InchesToCm(double ins)
```

```
{  
    return Ins*2.54;  
}  
  
// later on in the code  
Y = InchesToCm(X);
```

就可以用下面的代码替换它：

### 388

第14 章 Visual Studio 2005

```
Y = X * 2.54;
```

或者把下面的代码：

```
{  
    string Message = "Hi";  
    Console.WriteLine(Message);  
}
```

替换成：

```
Console.WriteLine("Hi");
```

因此，在过程中不必声明不必要的对象引用。

不能说C#编译器进行了什么优化工作，上面的两个例子在任何程序中都有可能出现，因为这类信息没有加以说明（对于托管语言如C#来说，上述优化很可能在JIT编译期间进行，而不是在C#编译器把源代码编译成程序集时进行）。从商业角度来看，编写编译器的公司通常不会对编译器使用的技巧进行过多的说明。这里还要强调，优化不影响源代码，它们只影响可执行代码的内容。但是，通过上面的例子，您应明白优化的内涵。

问题是，像上面例子中的优化可以使代码运行得更快，但它们对于调试来说，就没什么好处了。假定在第一个例子中，要在InchesToCm()方法内部设置一个断点，看看其中发生了什么。如果可执行代码没有InchesToCm()方法，因为编译器已删除了它，该怎么办？如果在Message变量上设置了一个监视点，但在编译好的代码中根本没有这个变量，该怎么办？

#### (2) 调试程序符号

在调试时，常常需要查看变量的值，因此需要通过该变量在源代码中的名称来指定它们。问题是可执行代码一般不包含这些变量的名称——编译器用内存地址取代了它们。.NET对这种情形进行了一定的改进。程序集中的某些对象是与其名称一起存储的，但只有小部分对象是这样，例如公共类和方法。这些名称仍在程序集进行JIT编译时被删除。如果在调试程序检查可执行代码时，要求调试程序给出变量HeightInInches的值，也不能得到我们希望的结果，它只能给出地址，根本就没有HeightInInches引用。因此，为了调试正确，用户必须在可执行代码中添加额外的调试信息。这些信息包括变量名和代码行号，让调试程序查找与源代码指令对应的可执行机器汇编语言指令。但不能在发布版本中放置这些信息，因为这是商业机密（调试信息可以让更多的人反汇编源代码），而且会增大可执行文件。

#### (3) 额外的源代码调试命令

在调试时，我们经常会遇到这样一个相关的问题：代码中有额外的命令行显示与调试相关的重要信息。显然，在发布软件前，必须从可执行代码中完全删除这些相关命令。可以手工完成这个任务，但如果仅是以某种方式标记这些语句，让编译器在编译代码时忽略它们，再发布软件，不是更容易吗？本书的第一部分已经讨论过，在C#中，可以定义一个合适的处理器符号，再把它和Conditional属性结合在一起使用，这就是所谓的条件编译。与最终销售的产品相比，即使把这些特性都加上，编译所有的商业软件和调试它们也是有细微差别的。Visual Studio可以把这些都考虑进去，因为如前所述，它存储了编译代

## 第 部分 Visual Studio

码时传递给编译器的所有信息，为了找出不同类型的生成文件，Visual Studio 只需要存储多套这类信息。不同的生成信息就称为配置。在创建一个项目时，Visual Studio 会自动提供两种配置，分别是 Debug 和 Release：

调试配置：通常会指定不进行任何优化，在可执行代码中给出额外的调试信息，

编译器假定给出了调试预处理器符号 Debug，除非在源代码中显式指定了

```
#undefined
```

发布配置：通常指定编译器要优化，在可执行代码中没有额外的调试信息，编译器假定没有给出任何调试预处理器符号。

也可以定义自己的配置。例如，建立专业级和企业级的生成版本，以发布两个版本的软件。过去，因为 Windows NT 支持 Unicode 字符编码，而 Windows 95 不支持，所以通常 C++ 项目有一个 Unicode 配置和一个 MBCS(多字节字符集) 配置。

### 3. 选择配置

一个很明显的问题是，因为 Visual Studio 存储了多个配置的信息，在生成一个项目时，该如何确定使用哪个配置？答案是总是有一个现行配置，在要求 Visual Studio 生成项目时，就使用这个配置（注意配置是每个项目的配置，而不是解决方案的配置）。

在默认情况下，创建一个项目时，调试配置就是现行配置。单击 Build 菜单选项，选择 Configuration Manager 项，就可以改变现行配置。还可以通过 Visual Studio 工具栏中的下拉菜单来改变现行配置。

### 4. 编辑配置

除了选择现行配置外，还可以查看和编辑配置，为此，需要在 Solution Explorer 中选择相应的项目，然后在 Project 菜单中选择 Properties，打开一个非常复杂的对话框（在 Solution Explorer 中右击项目名，在弹出的菜单中选择 Properties，也可以打开这个对话框）。该对话框包含一个树形视图，在该树形视图中可以选择许多不同的区域，以查看或编辑。这里不详细介绍所有的区域，只介绍其中两个最重要的区域。

图 14-23 显示了某个应用程序的可用属性的选项卡视图。这个屏幕图显示了本章前面创建的 ConsoleApplication1 项目的一般应用程序设置。

注意，可以选择要生成的程序集名和程序集的类型。其中的选项是控制台应用程序、Windows 应用程序和类库。当然，还可以改变程序集的类型（尽管这还有争议，但在 Visual Studio 最初生成项目时，为什么不能选择正确的项目类型呢？）。

屏幕图 14-24 显示了生成文件的配置属性。注意，对话框顶部的列表框允许指定要看的配置。此时可以查看调试配置——假定编译器定义了 DEBUG 和 TRACE 预处理器符号。如上所述，在调试配置中，代码没有优化，并会生成额外的调试信息。

一般情况下，用户不需要调整配置设置。如果需要使用它们，了解不同配置属性的区别是非常有用的。

## 390

第14 章 Visual Studio 2005

图 14-23

图 14-24

## 391

第 部分 Visual Studio

### 15.1.7 调试

在长时间讨论生成项目和生成配置后，我们还没有用大量的篇幅讨论代码的调试。原因是在 Visual Studio 中调试的规则和过程——设置断点和查看变量的值——与在 Visual Studio 6 IDE 中没有太大的区别。下面将简要介绍 Visual Studio 提供的调试功能，主要讨论对于某些开发人员来说是新内容的部分，我们还将论述如何处理异常，因为它们会给调试带来问题。

与 .NET 出现以前的语言一样，在 C# 中，调试所涉及的主要技术是设置断点，使用它们在代码的执行过程中检查某处发生的情况。

### 1. 断点

在 Visual Studio 中，可以在执行的代码中给任意一行设置断点。最简单的方式是在代码编辑器中单击该行，即在文档窗口左边的阴影区域中单击该行（或者选择该行，按下 F9 键），这样，就在该行设置了一个断点，只要代码执行到该行，就会中断，把控制权交给调试程序。在 Visual Studio 的以前版本中，断点在代码编辑器中用该行左边的一个大圆表示。Visual Studio 则把该行的文本和背景用另一种颜色来突出显示。再次单击该圆，就会删除断点。

如果程序并不适合于每次执行一行就中断一次，也可以设置条件断点。为此，可以单击 Debug | Windows | Breakpoints 菜单项，弹出一个对话框，该对话框要求用户给出要设置的断点信息，可以使用的选项有：

指定只有在对设置了断点的代码执行一定次数后，才能中断程序的执行。

指定执行某行一定的次数后，断点才起作用，例如执行该行 20 次后，断点才起作用（可用于调试大循环）。

给相关变量设置断点，而不是给指令设置断点。此时，监视的是变量值，只要变量的值发生了改变，就会触发断点。但是，使用这个选项会显著减慢代码的运行速度。在指令执行完后检查变量是否改变，会大大增加处理器时间。

### 2. 监视点

遇到断点时，通常要查看变量的值。最简单的方式是在代码编辑器中，把鼠标指针放在该变量名上，此时会显示一个小方框，其中给出了该变量的值。还可以扩展该方框，显示更详细的内容，如图 14-25 所示。

也可以使用监视窗口来查看变量的内容，它是一个带有标签的窗口，程序只有在调试器中运行时，该对话框才会出现，如图 14-26 所示。如果该对话框没有打开，可以选择 Debug | Windows | Autos。

类变量或结构变量的旁边有一个图标，单击它可以展开变量，查看其字段的值。

这个窗口的 3 个选项卡主要用于监视不同的变量：

Autos 监视程序运行时最后访问的变量。

Locals 监视当前执行的方法中的变量。

Watch 监视用户在 Watch 窗口中键入的变量。

## 392

第14 章 Visual Studio 2005

图 14-25

图 14-26

### 3. 异常

应用程序中的异常可以确保在该程序中以合适的方式处理错误情况。它们可以保证应用程序正常运行，不会给用户显示许多技术性的对话框。但在调试时，异常并没有那么强大的功能，这个问题有两个方面：

## 393

第 部分 Visual Studio

如果产生一个异常，则在调试时常常不希望由程序自动处理——有时处理异常就意味着退出，中断程序的执行！我们要让调试程序查找到发生异常的原因。当然，问题是如果要编写出健全的可以防范错误的好代码，程序就应能处理几乎所有的异常——包括要检测的错误！

如果产生了一个没有编写处理程序的异常，.NET 运行库还是会查找该异常的处理程序。此时它发现没有这样的处理程序，因而中断程序。这里没有调用堆栈，不能查看变量的值，因为它们都已出了作用域。

当然，可以在 catch 块中设置断点，但这常常没有什么帮助，因为在执行 catch 块时，按照定义，程序流会退出相应的 try 块，这样，要查看的变量值就出了作用域，找不出错误发生的原因。甚至不能查看堆栈跟踪，找出 throw 语句退出程序时执行了哪个方法，因为该方法已交出了控制权。当然，在 throw 语句上设置断点可以解决这个问题，但如果代码中有许多 throw 语句时，该如何编码？如何告诉编译器是哪个 throw 语句抛出了异常？实际上，Visual Studio 给这些问题提供了一个非常好的解决方案。查看一下 Debug 主菜单，其中有一个 Exceptions 菜单项，选择它会弹出 Exceptions 对话框，如图 14-27 所示，指定抛出异常时要执行什么操作。可以选择继续执行或者自动停止，开始调试代码，此时程序停止执行，调试程序开始调试 throw 语句。

图 14-27

这个工具的强大之处是用户可以根据抛出了哪个类的异常，来定制程序的行为。例如，在图 14-27 上，告诉 Visual Studio 在遇到由任一个 .NET 基类抛出的异常时，就中断执行，开始调试程序，但如果抛出了 AppDomainUnloadedException 异常，则不中断执行。Visual Studio 知道 .NET 基类中所有的异常类，也知道在 .NET 环境外部抛出的许多异常。Visual Studio 不会自动响应用户编写的定制异常类，但用户可以把自己的异常类手工添加到该列表中，指定哪个异常会立即停止执行程序。为此，只需单击图 14-27 中的 Add 按钮（在从树形结构中选择了一个顶级节点，该按钮就成为可用的），键入异常类的名称即可。

## 15.2 修订功能

许多开发人员在第一次为应用程序开发功能后，会改写应用程序，使之更易管理，可读性更高。这个过程称为修订。修订就是改写代码，提高可读性、性能，提供类型安全性，使应用程序更好地遵循标准 OO 面向对象 编程规则的过程。

394

第14 章 Visual Studio 2005

因此，Visual Studio 2005 的 C# 环境现在包含一组修订工具，这些工具位于 Visual Studio 菜单的 Refactoring 选项下。为了说明这些工具的作用，下面在 Visual Studio 中创建一个新类 Car：

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{
    public class Car
    {
        public string _color;
        public string _doors;
        public int Go()
        {
            int speedMph = 100;
            return speedMph;
        }
    }
}
```

现在，假定修订时希望修改代码，把 color 和 door 变量封装到 .NET 公共属性中。Visual Studio 2005 的修订功能可以在文档窗口中右击这些属性，选择 Refactor | Encapsulate Field 打开 Encapsulate Field 对话框，如图 14-28 所示。

图 14-28

在这个对话框中，提供属性名，单击OK按钮，就把选中的公共字段转换为一个私有字段，并把它封装到一个.NET 公共属性中。单击OK后，代码会改为(改写了两个字段)：

```
using System;
using System.Collections.Generic;
395
第 部分 Visual Studio
using System.Text;
namespace ConsoleApplication1
{
public class Car
{
private string _color;
public string Color
{
get { return _color; }
set { _color = value; }
}
private string _doors;
public string Doors
{
get { return _doors; }
set { _doors = value; }
}
public int Go()
{
int speedMph = 100;
return speedMph;
}
}
```

可以看出，这些向导不但能修订一个页面上的代码，也能修订整个应用程序的代码。  
它们还可以完成如下任务：

- 给方法、局部变量、字段等重命名
- 从选中的代码中提取方法
- 根据一组已有的类型成员提取接口
- 把局部变量提升为参数
- 对参数重命名或重新排序

Visual Studio 2005 提供的新修订功能可以使代码更简洁、可读性更高，结构化更强。

## 15.3 Visual Studio 2005 for .NET Framework 3.0

在默认情况下，Visual Studio 2005 不允许建立面向 .NET Framework 3.0 的应用程序。  
Visual Studio 2005 的默认安装仅面向 .NET Framework 2.0 要开始使用面向 .NET Framework 3.0 的新技术，需要安装另外几个软件包。  
.NET Framework 3.0 允许使用类库来建立新的应用程序类型，例如使用 Windows Presentation Foundation (WPF)、Windows Communication Foundation (WCF)、Windows Workflow Foundation (WF) 和 Windows CardSpace 的应用程序。

**396**

## 第14 章 Visual Studio 2005

要建立这些应用程序类型，首先需要安装 .NET Framework 3.0。安装 .NET Framework 3.0，也会在机器上安装 .NET Framework 2.0（假如没有安装 .NET Framework 2.0）。之后，就可以在 C:\WINDOWS\Microsoft.NET\Framework 中看到架构的这个版本。

注意，.NET Framework 3.0 不是 .NET Framework 的完全修订版本。例如，System.dll 或 System.Web.dll 就没有 3.0 版本，但在安装了 .NET Framework 2.0 版本后，就可以找到 .NET Framework 3.0 提供的三个组件所在的文件夹。这些文件夹的名称分别是 Windows Communication Foundation、Windows Workflow Foundation 和 WPF。如果使用的是 Windows Vista，则在 OS 的默认安装中已经安装了 .NET Framework 3.0。

如果在 Visual Studio 中使用 WPF、WCF 或 WF，就需要执行额外的步骤。在编写本书时，Microsoft 的 MSDN 网站上有两个产品可以帮助执行这些步骤：Visual Studio 2005 对 .NET Framework 3.0 的扩展（Windows Workflow Foundation）和 Visual Studio 2005 对 .NET Framework 3.0 的扩展（WCF 和 WPF）。据估计，这些扩展完全开发完毕时，用户将需要升级到 Visual Studio 的新版本（比 2005 更高的版本）。

要安装这些扩展，需要在机器上安装 Visual Studio 2005 之后，所有的 .NET Framework 3.0 项目类型就可以直接在 Visual Studio 中使用了。

### 15.3.1 .NET 3.0 的项目类型

在 Visual Studio 的 Orcas 之前的版本中，给各种 .NET 3.0 组件安装 Visual Studio 扩展，会提供各种新的项目类型。这些项目类型可以建立利用新技术的 .NET 应用程序。在 Visual Studio 中创建新项目时，注意有两个新的类别：其一是 .NET Framework 3.0，其二是 Workflow。如果选择 .NET Framework 3.0 选项，则可选的项目类型如图 14-29 所示。

图 14-29

在图 14-29 中，可以看出本节主要关注使用 Windows Presentation Foundation 和 Windows Communication Foundation 的项目。本节介绍的项目参见表 14-2。  
**397**

第 部 分 Visual Studio

表 14-2

项 目 说 明

Windows Application (WPF) 响应事件的基本空窗体。尽管该项目类型类似于 Windows Application 项目类型（Windows Forms），但这个 Windows Application 项目类型可以建立基于 XAML 的智能客户解决方案。

XAML Browser Application  
(WPF)

非常类似于用于 WPF 的 Windows Application，这个变体可以建立面向浏览器的、基于 XAML 的应用程序。

Custom Control Library  
(WPF)

可以建立在 .NET 3.0 应用程序中使用的定制服务器控件。使用交互操作技术，还可以在传统的 Windows 窗体应用程序中使用这些控件。WCF Service Library 可以创建一个服务库或一组服务，其端点通过 XML 配置文件来控制。

除了这四个项目类型之外，Workflow 部分还提供了另一组新项目类型。所有这些项目类型都关注的是 Windows Workflow Foundation (WF)。表 14-3 定义了这些新项目类型。

表 14-3

项 目 说 明

Sequential Workflow Console  
Application

可以创建利用顺序工作流的控制台应用程序

Sequential Workflow Library 可以创建利用顺序工作流的类库

Workflow Activity Library 作为用户建立的库的一个构造块，可以建立与这些项协作的操作

State Machine Workflow

Console Application

可以创建利用状态机工作流的控制台应用程序

State Machine Workflow

Library

可以创建利用状态机工作流的类库

Empty Workflow Project 关注工作流的空项目

从项目的Workflow部分提供的选项可以看出，基本上有两类项目：一类基于顺序工作流，一类基于状态机工作流。

顺序工作流会沿着某个顺序执行，例如工作流先执行步骤 A，再执行步骤 B，如果步骤 C 的结果为 true，就执行步骤 E。而状态机工作流是事件驱动的，它不是先执行步骤 A，再执行步骤 B。其过程应是：如果访问某个文件，就在工作流中完成这个步骤。新的 .NET 3.0 项目是相当新的，在许多方面都对完成步骤的方式有着革命性的改进。本书的其他章节，如第 31、40 和 41 章，将介绍这些新功能。

### 15.3.2 在 Visual Studio 中建立 WPF 应用程序

.NET Framework 3.0 给 Visual Studio 带来的一个主要变化是增加了 Windows Application (WPF) 项目类型。选择这个项目类型会创建一个 Windows.xaml 文件和一个 Windows.xaml.cs

**398**

第 14 章 Visual Studio 2005

文件。这个项目类型在 Solution Explorer 中默认创建的内容如图 14-30 所示。

图 14-30

在 Visual Studio 中，最大的变化出现在文档窗口中。创建这个项目后，文档窗口的默认视图如图 14-31 所示。

文档窗口有两个视图：设计视图和 XAML 视图。在设计视图中进行修改，会使 XAML 视图出现相应的变化，反之亦然。与传统的 Windows 窗体应用程序一样，WPF 应用程序也可以使用包含在 Visual Studio 工具箱中的控件。控件的这个新工具箱如图 14-32 所示。

图 14-31

**399**

第 部分 Visual Studio

图 14-32

### 15.3.3 在 Visual Studio 中建立 WF 应用程序

另一个完全不同的应用程序样式（在 Visual Studio 中建立应用程序时）是 Windows Workflow 应用程序类型。例如，从 New Project 对话框的 Workflow 部分选择 Sequential Workflow Console Application 项目类型，就会创建一个控制台应用程序，它的 Solution Explorer 视图如图 14-33 所示。

图 14-33

**400**

第 14 章 Visual Studio 2005

在建立使用 Windows Workflow Foundation 的应用程序时，一个很大的变化是它非常依赖于设计视图。仔细查看工作流（如图 14-34 所示），会发现它包含多个顺序步骤，甚至包含基于条件（如 if-else 语句）的操作。

图 14-34

## 15.4 小结

本章介绍了 .NET 环境中的一个最重要的编程工具 Visual Studio 2005。本章的大部分内容说明如何使用这个工具编写 C# 代码（以及 C++ 和 VB 2005 代码）。Visual Studio 2005 是编程界中最简单的开发环境之一。Visual Studio 很容易进行 RAD 开发，同时还可以深入了解创建应用程序的机制。本章详细介绍了如何使用 Visual Studio 完成各种任务，包括修订代码，读取 Visual Studio 6 项目、调试程序，以及许多可用于 Visual Studio 的窗口。

本章还介绍了可通过 .NET Framework 3.0 创建的新项目。这些新的项目类型关注的是 Windows Presentation Foundation、Windows Communication Foundation 和 Windows Workflow Foundation。

第 15 章将详细讨论部署。——

## 第 16 章 部署

编译源代码并完成测试后，开发过程并没有结束。在这个阶段，需要把应用程序提供给用户。无论是 ASP.NET 应用程序、智能客户应用程序还是用 Compact Framework 构建的应用程序，软件都必须部署到目标环境中。.NET Framework 使部署工作比以前容易得多，因为不再需要注册 COM 组件，编写新的注册表项。

本章将介绍可用于应用程序部署的选项，包括 ASP.NET 应用程序和智能客户应用程序的部署选项。本章的主要内容如下：

部署要求

简单的部署情况

基于 Windows 安装程序的项目

ClickOnce

### 16.1 部署的设计

部署常常是开发过程之后的工作，如果不下一定的工夫，可能会导致错误。为了避免在部署过程中出错，应在最初的设计阶段就对部署过程进行规划。任何部署问题，例如服务器的容量、桌面的安全性或从哪里加载程序集等，都应从一开始就纳入设计，这样部署过程才会比较顺利。

另一个必须在开发过程早期解决的问题是，在什么环境下测试部署。应用程序代码的单元测试和部署选项的测试可以在开发系统中进行，而部署必须在类似于目标系统的环境中测试。这是非常重要的，特别是目标计算机上不存在从属文件时。例如，第三方的库很早就安装在项目的开发计算机上，但目标计算机可能没有安装这个库。在部署软件包中很容易忘记包含这个库。在开发系统上进行的测试不可能发现这个错误，因为库已经存在了。对从属文件的说明可以帮助减少这种潜在的错误。

部署过程对于大型应用程序来说可能非常复杂。提前规划部署，在部署过程中可以节省时间和精力。

第 部分 Visual Studio

## 16.2 部署选项

本节概述 .NET 开发人员可以使用的部署选项。其中大多数选项将在本章的后面详细论述。

### 16.2.1 Xcopy 实用工具

Xcopy 实用工具允许把程序集或程序集组复制到应用程序文件夹中，从而减少了开发时间。由于程序集是自我包含的，元数据描述了包含在程序集中的内容，所以不需要在注册表中注册。每个程序集都跟踪它需要执行的其他程序集。默认情况下，程序集会在当前的应用程序文件夹中查找从属文件。把程序集移动到其他文件夹的过程将在本章后面讨论。

### 16.2.2 Copy Web 工具

如果开发的是 Web 项目，使用 Web 站点菜单中的 Copy Web 选项就会把运行应用程序所需要的组件复制到服务器上。

### 16.2.3 发布Web 站点

在发布 Web 站点时，会编译整个站点，然后复制到指定的位置。在预编译时，所有的源代码都会从最终的输出中删除，找出和处理所有编译错误。

### 16.2.4 部署项目

Visual Studio 2005 可以为应用程序创建安装程序。基于 Microsoft Windows Installer 技术有 4 种选择：创建合并模块；为客户应用程序创建安装程序；为 Web 应用程序创建安装程序；以及为基于智能设备 (Compact Framework) 的应用程序创建安装程序。还可以创建 cab 文件。部署项目为安装过程提供了极大的灵活性和可定制性。这 4 种部署方式对于大型应用程序都十分有用。

### 16.2.5 ClickOnce

ClickOnce 可以建立自动升级的、基于 Windows 的应用程序。ClickOnce 允许把应用程序发布到 Web 站点、文件共享、甚或 CD 上。在对应用程序进行升级、重新生成后，开发小组可以把它们发布到相同的位置或站点上。最终用户在使用应用程序时，程序会检查是否有更新版本，如果有，就进行更新。

## 16.3 部署的要求

基于 .NET 的应用程序一般都有运行要求。在执行任何托管的应用程序之前，CLR 对

404

第15 章部 署

目标平台都有一定的要求。

首先必须满足的要求是操作系统。目前下面的操作系统可以运行基于 .NET 的应用程序：

Windows 98

Windows 98, 第 2 版 (SE)

Windows Millennium Edition (ME)

Windows NT 4.0 (Service Pack 6a)

Windows 2000  
Windows XP Home  
Windows XP Professional  
Windows XP Professional TabletPC Edition  
Windows Vista

其次，必须支持下面的服务器平台：

Windows 2000 Server 和 Advanced Server  
Windows 2003 Server 系列

其他要求有Windows Internet Explorer 5.01 或更高版本，MDAC 2.6 或更高版本 (应用程序需要访问数据) 和用于 ASP.NET 应用程序的 Internet Information Services(IIS)。

在部署 .NET 应用程序时，还必须考虑硬件要求。硬件的最低要求是：

客户机：奔腾 90MHz, 32MB RAM  
服务器：奔腾 133MHz, 128MB RAM

要获得最佳性能，应增加 RAM, RAM越大，.NET 应用程序运行得就越好。服务器应用程序更是如此。

如果要运行使用 Windows Presentation Foundation (WPF)、Windows Communication Foundation (WCF)或Windows Workflow Foundation (WF)的 .NET 3.0 应用程序，要求就更严格。.NET 3.0 至少需要 Windows XP SP2 上述列表还应添加如下内容：

Windows XP Home (SP2)  
Windows XP Professional (SP2)  
Windows XP Professional TabletPC Edition (SP2)  
Windows Vista (不包括 IA64 平台)

支持下述服务器平台：

Windows 2003 Server Family (SP1)  
Windows Server “ Longhorn” IA64 版本

最低的硬件要求也有变化：

客户机：奔腾 400 MHz, 96 MB RAM

## 16.4 部署.NET 运行库

使用 .NET 开发应用程序时，需要依赖 .NET 运行库。这似乎很明显，但有时可以忽略

405

第 部分 Visual Studio

这一点。如果应用程序不使用任何 .NET 3.0 功能，就只需要安装 dotnetfx.exe 如果使用了 .NET 3.0 功能，还需要安装 dotnetfx3.exe

在下面对创建部署软件包的讨论中，运行库的安装是可选的。安装程序会检查是否安装了相应的运行库，如果没有，安装程序会从本地媒介中安装运行库，或者到指定的下载站点上下载并安装运行库。

## 16.5 简单的部署

如果在应用程序的初始设计阶段考虑了部署，那么部署就只是把一组文件复制到目标计算机上。对于Web 应用程序，就只需使用 Visual Studio 2005 中的一个简单的菜单选项。本节就讨论这种简单的部署情况。

为了了解如何设置各种部署选项，必须有一个要部署的应用程序。从 [www.wrox.com](http://www.wrox.com) 上下载的示例包含 3 个项目： SampleClientApp、 SampleWebApp 和 AppSupport。

SampleClientApp 是一个智能客户应用程序， SampleWebApp 是一个简单的Web 应用程序， AppSupport 是一个类库，它包含一个简单的类，该类返回一个包含当前日期和时间的字符串。 SampleClientApp 和 SampleWebApp 使用 AppSupport 的结果填充一个标签。为了使用

这些示例，首先加载并构建 AppSupport，然后在其他两个应用程序中，设置对新构建的 AppSupport.dll 的引用。

下面是 AppSupport 程序集的代码：

```
using System;
namespace AppSupport
{
    ///<summary>
    ///Simple assembly to return date and time string.
    ///</summary>
    public class Support
    {
        private Support()
        {
        }

        public static string GetDateTimeInfo()
        {
            DateTime dt = DateTime.Now;
            return string.Concat(dt.ToString("MM/dd/yyyy"), " ", dt.ToString("HH:mm:ss"));
        }
    }
}
```

这个简单的程序集足以演示可用的部署选项。

## 406

### 第15章部署

#### 16.5.1 Xcopy 部署

Xcopy 部署就是把一组文件复制到目标计算机上的一个文件夹中，然后在客户机上执行应用程序。这个术语来自于 DOS 命令 xcopy.exe。无论程序集的数目是多少，如果文件复制到同一个文件夹中，应用程序就会运行，不需要编辑配置设置或注册表。

为了理解 Xcopy 部署的工作原理，打开示例下载文件中的 SampleClientApp 解决方案 (SampleClientApp.sln)，把目标改为 Release，进行完整的编译。然后，使用“我的电脑”或文件管理器导航到项目文件夹 \SampleClientApp\bin\Release，双击 SampleClientApp.exe，运行应用程序。现在单击按钮，打开另一个对话框。这将验证应用程序是否能正常运行。

当然，这个文件夹是 Visual Studio 放置输出的地方，所以应用程序能正常工作。

创建一个新文件夹，命名为 ClientAppTest，把这两个文件从 Release 文件夹复制到这个新文件夹中，然后删除 Release 文件夹。再次双击 SampleClientApp.exe 文件，验证它是否正常工作。

Xcopy 部署只需把程序集复制到目标机器上，就可以部署功能完善的应用程序。这里使用的示例非常简单，但这并不意味着这个过程对较复杂的应用程序无效。实际上，使用这种方法对要部署的程序集的大小和数目没有限制。不想使用 Xcopy 部署的原因是它不能把程序集放在全局程序集缓存 (GAC) 中，或者不能在“开始”菜单中添加图标。如果应用程序仍依赖于某种类型的 COM 库，就不能很容易地注册 COM 组件。

#### 16.5.2 Xcopy 和 Web 应用程序

Xcopy 部署也可以用于 Web 应用程序，但文件夹结构有点不同。我们必须建立 Web 应用程序的虚拟目录，并配置合适的用户权限。这个过程通常需要使用 IIS 管理工具来完成。在建立虚拟目录后，Web 应用程序文件就可以复制到虚拟目录中。复制 Web 应用程序的文件有点困难，需要考虑两个配置文件和页面使用的图像。

#### 16.5.3 Copy Web 工具

一种较好的方法是使用 Copy Web 工具。在 Visual Studio 2005 的 Website | Copy Web Site 菜单项中就可以访问工具。它基本上是一个 FTP 客户程序，用于给远程位置来回传送文件。远程位置可以是任意 FTP 或 Web 站点，包括本地 Web 站点、IIS Web 站点和 Remote (Frontpage) Web 站点。Copy Web 工具的另一个特性是，它会把远程服务器上的文件与源站点上的文件同步。源站点总是 Visual Studio 2005 中当前打开的站点。如果当前项目有多个开发人员，就可以使用这个工具与本地开发站点保持同步。所进行的修改可以与用于测试的公共服务器进行同步。

#### 16.5.4 发布Web 站点

Web 项目的另一个部署选项是发布 Web 站点。发布 Web 站点就是预编译整个站点，并把编译好的版本放在指定的位置。该位置可以是文件共享、FTP 位置，或可以通过 HTTP 访问的

407

第 部分 Visual Studio

其他位置。编译过程会从程序集中去除所有的源代码，为部署创建 DLL 文件。这也包括 .ASPX 源文件中的标记。.ASPX 文件并不包含一般的标记，而是包含程序集的一个指针。每个 .ASPX 文件都与一个程序集相关。无论是模型、后台代码或单个文件，这个过程都会执行。

发布 Web 站点的优点是速度快，很安全。速度有所提高，是因为所有的程序集都已编译。否则，第一次访问页面时会有一个延迟，因为要编译和缓存页面和从属代码。安全性有所提高，是因为不部署源代码。另外，在部署前所有的源代码都进行了预编译，找出了所有的编译错误。

使用 Website | Publish Web Site 菜单项就可以发布 Web 站点。我们需要提供要发布的位置。这也可以是文件共享、FTP 位置、Web 站点或本地磁盘路径。在完成编译后，文件就放在指定的位置。在这里可以把文件复制到阶段服务器、测试服务器或产品服务器上。

## 16.6 Installer 项目

xcopy 部署使用起来很简单，但有时它缺乏一些功能。为了克服这个缺点，Visual Studio 2005 提供了 6 个 Installer 项目类型。其中 4 个类型基于 Windows Installer 技术，表 15-1 列出了这些项目类型。

表 15-1

项目 类型 说 明

Setup Project 用于安装客户应用程序、中间层应用程序和运行为 Windows 服务的应用程序

Web Setup Project 用于安装基于 Web 的应用程序

Merge Module Project 创建合并模块，这些模块可以和其他基于 Windows Installer 的安装应用程序一起使用

Cab Project 创建 cab 文件，通过旧式的部署技术进行发布

Setup Wizard 帮助创建部署项目

Smart Device CAB Project Pocket PC Smartphone 和其他基于 CE 的应用程序的 CAB 项目

Setup 和 Web Setup Project 非常相似。主要的区别是使用 Web Setup，项目会部署到 Web 服务器上的一个虚拟目录中，而使用 Setup Project，项目会部署到文件夹结构中。这两个项目类型都基于 Windows Installer，拥有基于 Windows Installer 的安装程序的所有功能。在创建包含在多个部署项目中的组件或功能库时，一般使用 Merge Module Project。创建合并模块时，可以设置专用于组件的配置项目，而无需在主部署项目的创建过程中考虑它们。Cab Project 类型仅为应用程序创建 Cab 文件。Cab 文件由旧式的安装技术以及一些基于 Web 的安装过程使用。Setup Wizard 项目类型逐步完成创建部署项目的步骤，在此过程中向用户询问特定的问题。下面的几节讨论如何创建这些部署过程，可以改变哪些设置和属性，可以增加什么定制内容。

408

## 第15 章部 署

### 16.6.1 Windows Installer

Windows Installer 是一个服务，负责管理在大多数Windows 操作系统上安装、更新、修复和删除应用程序。它是Windows ME, Windows 2000, Windows XP 和Windows Vista 的一部分，可以用于Windows 95, Windows 98 和Windows NT 4.0。Windows Installer 的当前版本是2.0。

Windows Installer 在数据库中跟踪应用程序的安装。在卸载应用程序时，Windows Installer 很容易跟踪和删除已添加的注册表设置、复制到硬盘上的文件，以及已添加的桌面和“开始”菜单图标。如果有某个文件仍被另一个应用程序引用，安装程序就会把它保留在硬盘上，不会使其他的应用程序中断。数据库还可以修复应用程序。如果注册表设置或与应用程序相关的 DLL 被破坏或不小心删除了，就可以修复安装。在修复过程中，安装程序会从上一次安装中读取数据库，并复制该安装。

Visual Studio 2005 中的部署项目可以创建Windows 安装软件包。部署项目允许访问大多数需要访问的内容，以便安装给定的应用程序。但是，如果需要更多的控制，就应查看 Windows Installer SDK，它在 Platform SDK 中，其中包含了为应用程序创建定制安装软件包的说明。下面几节将使用 Visual Studio 2005 部署项目创建这些安装软件包。

### 16.6.2 创建安装程序

为客户应用程序或Web 应用程序创建安装软件包并不困难。第一个任务是标识应用程序需要的所有外部资源，包括配置文件、COM 组件、第三方库、控件和图像。前面说过，在项目的文档说明中应包含一个从属文件列表。这个文档说明是非常有用的。Visual Studio 2005 可以询问程序集，提取该程序集的从属文件，但我们仍需要审查这些内容，以防遗漏。

另一个问题是，在过程的什么时候创建安装软件包。如果设置了一个自动构建过程，就可以把安装软件包的构建包含在项目成功构建的过程中。在耗时而复杂的大型项目中，过程的自动进行会大大减少出错的可能性，我们可以把部署项目包含在项目解决方案中。

Solution Property Pages 对话框中有一个 Configuration Properties 设置。使用这个设置可以选择要为各种构建配置包含的项目。如果选择 Release builds but not for the Debug builds 下面的 Build 复选框，安装软件包就只在创建发布版本时创建。这也是下面示例所使用的过程。

图 15-1 显示了 SampleClientApp 解决方案的 Solution Property Pages 对话框。其中显示了 Debug 配置，没有给安装项目选中 Build 复选框。

#### 1. 简单的客户应用程序

在下面的示例中，要为 SampleClientApp 解决方案创建一个安装程序（它包含在示例下载文件中，其中还包含已完成的安装程序项目）。

对于 SampleClientApp，创建两个部署项目。一个创建为独立的解决方案，另一个在原来的解决方案中创建，以便说明选择创建这两个部署项目的优缺点。

第一个示例将说明如何在独立的解决方案中创建部署项目。在开始创建部署项目之前，要确保部署的应用程序有一个发布版本。接着，在 Visual Studio 2005 中创建一个新项目。**409**

#### 第 部 分 Visual Studio

New Project 对话框中，选择左边的 Setup and Deployment Projects，再选择右边的 Setup Project，给它指定一个名称（例如 SampleClientStandaloneSetup）。此时，屏幕如图 15-2 所示。

图 15-1

图 15-2

在 Solution Explorer 窗口中，单击项目，再单击 Properties 窗口，就会看到一组属性。这些属性将在应用程序的安装过程中显示。其中一些属性还会显示在“添加/删除程序”控件

#### 410

#### 第15 章部 署

面板上。由于用户在安装过程中可以看到大多数属性（或者在“添加/删除程序”控件面板上

查看安装时可以看到它们），所以正确设置它们会使应用程序更专业。这个属性列表非常重  
要，特别是在应用程序要进行商业化部署时，更是如此。表 15-2 描述了这些属性及其值。

表 15-2

项目 属性 说 明

AddRemoveProgramIcon 显示在“添加/删除程序”对话框中的图标

Author 应用程序的编写者。这个属性设置通常与 Manufacturer 的相同，它显  
示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添  
加/删除程序”对话框的 SupportInfo 页面上的 Contact 字段中

Description 这是一个形式自由的文本字段，描述了要安装的应用程序或组件。这  
些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，  
以及“添加/删除程序”对话框的 SupportInfo 页面上的 Contact 字段中

DetectNewerInstalledVersion 这是一个布尔值，设置为 true 时，将检查是否已安装了应用程序的更  
新版本，如果是，就停止安装过程

InstallAllUsers 这是一个布尔值，设置为 true 时，将为计算机的所有用户安装应用程  
序。设置为 false 时，就只有当前用户能访问应用程序

Keywords 可用于在目标计算机上搜索 msi 文件。这些信息显示在 msi 软件包的  
Properties 对话框中的 Summary 页面上

Localization 用于字符串资源和注册设置的地域。这会影响安装程序的用户界面  
Manufacturer 生产应用程序或组件的公司名称。一般与 Author 属性中指定的信息  
相同。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary  
页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Publisher  
字段中，用作应用程序默认安装路径的一部分

ManufacturerURL 一个 Web 站点的 URL，该站点与要安装的应用程序或组件相关

PostBuildEvent 在构建结束后执行的命令

PreBuildEvent 在构建开始前执行的命令

ProductCode 应用程序或组件的唯一的字符串 GUID。Windows Installer 使用这个  
属性标识应用程序的后续升级或安装

ProductName 描述应用程序的名称，在“添加/删除程序”对话框中用作应用程序的  
描述，还用作默认安装路径的一部分：C:\Program Files\Manufacturer\  
ProductName

RemovePreviousVersions 一个布尔值，如果设置为 true，就检查计算机上是否安装了应用程  
序的以前版本。如果是，就调用以前版本的卸载功能，之后继续安装。

这个属性使用 ProductCode 和 UpgradeCode 确定是否卸载。

UpgradeCode 应相同，而 ProductCode 应不同

## 411

### 第 部 分 Visual Studio

(续表)

项目 属性 说 明

RunPostBuildEvent 运行 PostBuildEvent 的时间，其选项有 On successful build 或 Always

SearchPath 一个字符串，表示从属程序集、文件或合并模块的搜索路径。在开发  
机器上建立安装软件包时使用该属性

Subject 与应用程序相关的其他信息。这些信息显示在 msi 软件包的 Properties  
对话框中的 Summary 页面上

SupportPhone 为应用程序或组件提供支持的电话号码。这些信息显示在“添加/删除  
程序”对话框的 SupportInfo 页面上的 Support Information 字段中

SupportURL 为应用程序或组件提供支持的 URL。这些信息显示在“添加/删除程序”  
对话框的 SupportInfo 页面上的 Support Information 字段中

TargetPlatform 支持 Windows 的 32 或 64 位版本

Title 安装程序的标题。它显示在 msi 软件包的 Properties 对话框中的 Summary 页面上

UpgradeCode 一个字符串 GUID, 表示同一应用程序的不同版本共享的标识符。对于应用程序的不同版本或不同语言版本，不应修改 UpgradeCode, 该属性由 DetectNewerInstalledVersion 和 RemovePreviousVersions 使用 Version 安装程序、 cab 文件或合并模块的版本号。注意它不是要安装的应用程序的版本号

设置完属性后，就可以开始添加程序集了。在这个示例中，唯一要添加的程序集是主可执行文件 SampleClientApp.exe 为此，可以在 Solution Explorer 中右击项目，或从 Project 菜单中选择 Add, 此时有 4 个选项：

Project Output: 下一个示例探讨这个选项

File: 用于添加 readme 文本文件或不在构建过程中添加的其他文件

Merge Module: 独立创建的合并模块

Assembly: 使用这个选项可以选择要安装的程序集

本例选择 Assembly, 打开 Component Selector 对话框，该对话框类似于给项目添加引用的对话框。浏览至应用程序的 \bin\release 文件夹，选择 SampleClientApp.exe, 在 Component Selector 对话框中单击 OK, 现在可以看到 SampleClientApp.exe 在部署项目的 Solution Explorer 中。在 Detected Dependencies 部分，可以看到 Visual Studio 要求 SampleClientApp.exe 给出它需要的程序集。在本例中，会自动包括 AppSupport.dll。继续这个过程，直到应用程序中的所有程序集都显示在部署项目的 Solution Explorer 中为止。

接着，需要确定把程序集部署到什么地方。在默认情况下，在 Visual Studio 2005 中会显示文件系统编辑器，这个编辑器分为两个窗格，左边的窗格显示目标机器上文件系统的层次结构，右边的窗格则显示选中文件夹的详细视图。文件夹名称可能不是我们希望看到

## 412

### 第15 章部署

的，但这些文件夹用于目标机器，例如，文件夹 User's Programs Menu 映射为目标客户机的 C:\Documents and Settings\User Name\Start Menu\Programs 文件夹。

此时可以添加其他文件夹，例如特定的文件夹或定制的文件夹。要添加特定的文件夹，应确保目标机器上的文件系统在左边的窗格上突出显示，然后选择主菜单中的 Action 菜单。

Add Special Folder 菜单选项提供了可以添加的文件夹列表。例如，如果要在 Application 文件夹中添加一个文件夹，就可以在编辑器的左边窗格上选择 Application 文件夹，再选择 Action 菜单。这时就会出现一个可以创建新文件夹的 Add 菜单。给新文件夹重新命名，就会在目标机器上创建它。

我们要添加的一个特定文件夹是 GAC 文件夹。如果有几个不同的应用程序使用 AppSupport.dll，就可以把它安装到 GAC 中。为了把程序集添加到 GAC 中，程序集必须有一个强名称。把程序集添加到 GAC 的过程就是在 Special Folder 菜单中添加 GAC，再把要放在 GAC 中的程序集从当前文件夹拖放到 Global Assembly Cache 文件夹中。如果试图对没有强名称的程序集进行这个操作，部署项目就不能编译。

如果选择 Application 文件夹，在右边的窗格上，刚才添加的程序集就会自动添加到 Application 文件夹中。还可以把程序集移动到其他文件夹中，但程序集必须能找到对方（有关探测的更多信息请参阅第 16 章）。

如果要在用户的桌面或“开始”菜单上添加应用程序的快捷方式，就应把该快捷方式拖放到适当的文件夹中。要创建桌面快捷方式，就应进入 Application 文件夹，在编辑器的右边窗格上选择该应用程序，再进入 Action 菜单，选择 Create Shortcut 菜单项，创建应用程序的快捷方式。在创建好快捷方式后，把它拖放到 User's Desktop 文件夹中。现在安装应用程序，快捷方式就会显示在桌面上。一般情况下，应由用户决定是否需要应用程序的

快捷方式。要求用户输入信息并执行相应步骤的过程将在本章后面介绍。在“开始”菜单中创建菜单项的过程与此相同。另外，如果查看刚才创建的快捷方式的属性，就可以配置快捷方式的基本属性，例如参数和要使用的图标。应用程序图标是默认图标。  
在创建部署项目之前，需要检查一些项目属性。如果选择 Project 菜单，再选择 SampleClient- StandaloneSetup Properties，就会打开 Project Property Pages 对话框，其中的属性是针对当前配置的。在 Configuration 下拉框中选择配置后，就可以修改表 15-3 中的属性。

表 15-3

#### 属性说明

Output file name 在编译项目时生成的.msi 或.msm 文件的名称

Package file 这个属性允许指定如何打包文件。其选项有：

As loose uncompressed files: 所有的部署文件都在同一个目录下存储为.msi 文件；

In setup file: 文件被打包到.msi 文件中(默认设置)；

In cabinet files: 文件被打包到同一目录下的一个或多个 cab 文件中。选择这个选项时，CAB size 选项就是可用的

#### 413

第 部分 Visual Studio

(续表)

#### 属性说明

Prerequisites URL 可以指定在哪里查找.NET Framework 或 Windows Installer 2.0 等必需的程序。单击 Settings 按钮会显示一个对话框，其中包含安装过程中可以使用的技术：

Windows Installer 2.0

.NET Framework

Microsoft Visual J# .NET Redistributable Package 2.0

SQL Server 2005 Express Edition

Microsoft Data Access Components 2.8

还有一个选项，可以从预定义的 URL 上下载必需的程序，或从安装位置加载它们

Compression 这个属性指定所包含文件的压缩样式。其选项如下：

Optimized for speed: 用于较大的文件，安装时间较短(默认设置)

Optimized for size: 用于较小的文件，安装时间较长

None: 不压缩

CAB size Package file 属性设置为 In cabinet files 时，这个属性就会被激活。它不仅可以创建一个 cabinet 文件，还可以设置每个 cab 文件的最大尺寸

Authenticode Signature 在选中这个属性时，部署项目的输出就使用 Authenticode 标记，默认设置为不选中

Certificate file 用于签名的证书

Private key file 包含签名文件的数字加密键的私有键

Timestamp server URL Timestamp 服务器的 URL，也用于 Authenticode 标记

在设置完项目属性后，就应创建部署项目，为 SampleClientApp 应用程序创建安装程序。在建立项目后，就可以在 Solution Explorer 中右击项目名，测试安装了，此时可以在弹出的菜单中访问 Install 和 Uninstall 选项。如果一切正常，就可以成功安装和卸载 SampleClientApp 应用程序。

#### 同一个解决方案项目

上一个示例成功地创建了一个部署软件包，但有几个缺点。例如，在新程序集添加到原应用程序中时会发生什么情况？部署项目不会自动识别任何改动的地方，而必须添加新

程序集，再验证新的从属文件已包含进来。在较小的应用程序(如本例)中，这没有什么大不了。但在处理包含几十个甚至上百个程序集的应用程序时，这就可能很难维护了。Visual Studio 2005 为解决这个潜在的问题提供了一个简单的方法，即把部署项目包含在应用程序解决方案中，这样就可以把主项目的输出当作部署程序集了。下面以 SampleClientApp 为例来说明。

## 414

### 第15 章部 署

在 Visual Studio 2005 中打开 SampleClientApp 解决方案，使用 Solution Explorer 添加一个新项目，选择 Deployment and Setup Projects，再选择 Setup Project，之后按照上一节介绍的步骤进行。可以把这个项目命名为 SampleAppSolutionSetup 在前面的示例中，是在 Project 菜单中选择 Add | Assemblies，添加了程序集，这次在 Project 菜单中选择 Add | Project Output，这会打开 Add Project Output Group 对话框，如图 15-3 所示。

图 15-3

对话框的顶部有一个下拉列表框，其中显示了当前解决方案中的所有项目。选择主启动项目，然后从下面的列表中选择要包含在项目中的项，选项有 Documentation Primary Output Localized Resources Debug Symbols Content Files 和 Source Files。首先选择 Primary Output，这包括建立应用程序时的输出和所有的从属文件。对话框中还有一个下拉列表框，其中列出了有效的配置：Debug Release 以及自己添加的定制配置。这还确定了提取什么输出。对于部署，应使用 Release 配置。

在完成这些选择后，在 Solution Explorer 中就会给部署项目添加一个新条目。该条目的名称是 Primary output from SampleClientApp(Release .NET)。另外，文件 AppSupport.dll 将出现在从属文件列表中。与以前一样，不需要搜索从属程序集。

此时，仍在应用上一节讨论的所有项目属性。可以修改 Name Manufacturer cab file size 和其他属性。在设置好属性后，建立解决方案的 Release 版本，并测试安装。一切应像希望的那样正常工作。

为了理解把部署软件包添加到应用程序解决方案中的优点，下面把一个新项目添加到解决方案中。在本例中该项目名为 AppSupport1。在该项目中，有一个简单的测试方法，它返回字符串 Hello World。在 SampleTestApp 中设置一个对新增项目的引用，建立解决方案的另一个 Release 版本。部署项目会自动提取新的程序集，无需我们做任何工作。如果返回去，打开上一个示例中的独立部署项目，除非添加程序集，否则它是不会被提取的。

## 2. 简单的Web 应用程序

为 Web 应用程序创建安装软件包与创建客户安装软件包没有什么不同。下载的示例包

## 415

### 第 部分 Visual Studio

含一个 SampleWebApp 应用程序，它也利用了 AppSupport.dll 程序集。可以用与创建客户部署项目相同的方式创建部署项目，即创建独立的部署项目，或在原解决方案中创建部署项目。在这个示例中，我们在原解决方案中创建部署项目。

启动 SampleWebApp 解决方案，添加一个新的 Deployment and Setup 项目。这次要确保在 Templates 窗口中选择 Web Setup Project。如果查看该项目的属性视图，就会看到 Web 应用程序拥有与客户应用程序相同的所有属性。唯一新增的属性是 RestartWMService。这是一个布尔值，用于在安装过程中重新启动 IIS。如果使用 ASP.NET 组件，而且没有替换 ATL 或 ISAPI dll，就不需要修改该属性。

如果查看文件系统编辑器，就会注意其中只有一个文件夹。Web Application 文件夹就是我们的虚拟目录。默认情况下目录名就是部署项目名，位于 Web 根目录下。表 15-4 解释了可以在安装程序中设置的属性。上一节讨论的属性未包括在内。

表 15-4

### 属性说 明

AllowDirectoryBrowsing 布尔值，如果设置为 true，就允许以 HTML 格式列出虚拟目录中的文件

和子文件夹。该属性映射为 IIS 的 Directory Browsing 属性

AllowReadAccess 布尔值，如果设置为 true，就允许用户读取或下载文件。该属性映射为 IIS

的 Read 属性

AllowScriptSourceAcces

s

布尔值，如果设置为 true，就允许用户访问源代码，包括脚本。该属性映

射为 IIS 的 Script source access

AllowWriteAccess 布尔值，如果设置为 true，就允许用户修改可写文件中的内容。映射为 IIS

的 Write 属性

ApplicationProtection 确定运行在服务器上的应用程序的保护级别。有效值如下：

Low: 应用程序与 Web 服务运行在同一个进程中

Medium: 应用程序运行在同一个进程中，但不与 Web 服务运行在同一个进程中

High: 应用程序运行在它自己的进程中

该属性映射为 IIS 的 Application Protection 属性。如果 IsApplication 属性为 false，则不起作用

AppMappings 列出应用程序名以及与应用程序相关的文档或数据文件。该属性映射为 IIS 的 Application Mappings 属性

Condition Windows Installer 条件必须匹配所安装的项

DefaultDocument 用户第一次浏览站点时的默认文档或启动文档

ExecutePermissions 用户执行应用程序必须拥有的许可级别。有效值如下：

None: 只能访问静态内容

ScriptOnly: 只能访问脚本，包括 ASP

ScriptAndExecutables: 可以访问所有文件

该属性映射为 IIS 的 Execute Permissions

## 416

第15 章部 署

(续表 )

属 性 说 明

Index 布尔值，如果设置为 true，就允许给 Microsoft Indexing 服务的内容建立索引。该属性映射为 IIS 的 Index this resource 属性

IsApplication 布尔值，如果设置为 true，就让 IIS 为文件夹创建应用程序根目录

LogVisits 布尔值，如果设置为 true，就可以把对 Web 站点的访问记录到日志文件中。该属性映射为 IIS 的 Log visits 属性

Property 可以在安装期间访问的指定属性

VirtualDirectory 应用程序的虚拟目录，这相对于 Web 服务器

注意，大多数属性都是 IIS 的属性，可以在 IIS 管理工具中设置。所以有如下逻辑假设：

为了在安装程序中设置这些属性，安装程序在运行时需要拥有管理员权限。这里进行的设置可能会危害到安全，所以对做出的修改要进行很好的说明。

除了这些属性之外，创建部署项目的过程非常类似于前面的客户示例中所介绍的过

程。两个项目的主要区别是允许在安装过程中修改 IIS。可以看出，我们对 IIS 环境有很大的控制权。

### 3. Web 服务器上的客户

另一个安装情况是在 Web 站点上运行安装程序，或在 Web 站点上运行应用程序。如

果必须把应用程序部署给大量的用户，这就是两个很有吸引力的选项。在 Web 站点上部署，就不需要部署介质了，如 CD-ROM、DVD，甚或软盘。在 Web 站点甚至网络共享上运行

应用程序，就根本不需要发布安装程序了。

在Web 站点上运行安装程序是相当简单的，使用本章前面讨论的Web Bootstrapper 项目编译选项即可，此时需要提供安装文件夹的URL，在这个文件夹中，安装程序会查找需要的msi 和其他文件。设置好这个选项，并编译部署软件包后，就可以把它复制到 Setup folder URL 属性指定的Web 站点上。这样当用户导航到这个文件夹时，就可以运行安装程序，或者先下载再运行它。在这两种情况下，用户都必须连接到同一个站点，才能完成安装。

### 无干涉部署

还可以在 Web 站点或网络共享上运行应用程序。这个过程有点麻烦，这也是在设计应用程序时应考虑部署的一个主要原因。有时这称为无干涉部署 (Not Touch Deployment, NTD)。为了使这个过程顺利完成，应用程序代码必须以支持 NTD 的方式编写。利用 NTD 创建应用程序有两种方式：一种是把大多数应用程序代码放在 DLL 程序集中，DLL 放在 Web 服务器或网络的文件共享上。然后创建一个要在客户机上部署的小型应用程序可执行文件。这个存根程序 (Stub Program) 将使用 LoadFrom 方法调用一个 DLL 程序集，启动应用程序。存根程序唯一能看到的是 DLL 中的主入口。一旦加载了 DLL 程序集，应用程序就从同一个 URL 或网络共享中加载其他程序集。程序集首先会在应用程序目录中查找从属程序集，这是用于启动应用程序的 URL。在用户的客户机上，存根应用程序使用的代码如下所示。这个示

417

第 部分 Visual Studio

例调用 AppSupportII.dll 程序集，并把 TestMethod 调用的输出放在 label1 中。

```
Assembly testAssembly =  
Assembly.LoadFrom("http://localhost/AppSupport/AppSupportII.dll");  
Type type = testAssembly.GetType("AppSupportII.TestClass");  
object testObject = Activator.CreateInstance(type);  
label1.Text = (string)type.GetMethod("TestMethod").Invoke(testObject, null);
```

这个过程使用反射技术第一次从 Web 服务器上加载程序集。在这个示例中，Web 站点是本地机器 (localhost) 上的一个文件夹。接着，提取类的类型 (这里是 TestClass)。有了类型信息后，就可以使用 Activator.CreateInstance 方法创建对象。最后一步是获取 MethodInfo 对象 (GetMethod 的输出)，并调用 Invoke 方法。在比较复杂的应用程序中，这是应用程序的主入口点。从现在开始，就不再需要存根程序了。

另外，还可以把整个应用程序部署到 Web 站点上。对于这种方法，应创建一个简单的 Web 页面，其中包含一个到应用程序的安装可执行文件的链接，或者在用户桌面上有一个包含 Web 站点链接的快捷方式。单击这个链接，应用程序就会下载到用户的程序集下载缓存中，该缓存位于 Global Assembly Cache 中。应用程序从下载缓存中运行。每次请求新程序集时，都需要进入下载缓存，检查该程序集是否存在。如果不存在，就进入获取主应用程序的 URL。以这种方式部署应用程序的优点是，在能对应用程序进行更新时，该更新版本只需部署到一个地方。我们把新程序集放在 Web 文件夹中，当用户启动应用程序时，运行库会比较 URL 中的程序集版本和下载缓存中的程序集版本，如果在 URL 中找到新版本，就下载它，替换下载缓存中的当前版本。这样，用户将总是访问应用程序的最新版本。要对更新过程和安全性进行更多的控制，ClickOnce 是一个比较好的选项。

## 16.7 ClickOnce

ClickOnce 是一种允许应用程序自动升级的部署技术。应用程序发布到文件共享、Web 站点或 CD 这样的媒介上。之后，ClickOnce 应用程序就可以自动升级，而无需用户的干涉。ClickOnce 还解决了安全权限问题。一般情况下，要安装应用程序，用户需要有管理权限。而利用 ClickOnce，用户只要有运行应用程序所需的最低权限，就可以安装和运行应用程序。

### 16.7.1 ClickOnce 操作

ClickOnce 应用程序有两个基于 XML 的清单文件，其中一个是应用程序的清单，另一个是部署清单。这两个文件描述了部署应用程序所需的所有信息。

应用程序清单包含的应用程序信息有需要的权限、要包括的程序集和其他从属文件。

部署清单包含了应用程序的部署信息。应用程序清单的位置信息包含在部署清单中。这些清单的完整模式在 .NET SDK 文档说明中。

ClickOnce 有一些限制，例如，程序集不能添加到 GAC 中。表 15-5 比较了 ClickOnce 和 Windows Installer。

## 418

### 第15 章部署

表 15-5

#### ClickOnce Windows Installer

应用程序的安装位置 ClickOnce 应用程序缓存 Program Files 文件夹

给多个用户安装 否 是

安装共享文件 否 是

安装驱动程序 否 是

安装到 GAC 中 否 是

在“启动”组中添加应用程序 否 是

在菜单中添加应用程序 否 是

注册文件类型 否 是

访问注册表 否。有访问 HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run 权限是

文件的二进制修补 是 否

根据需要安装程序集 是 否

在一些情况下，使用 Windows Installer 比较好，但 ClickOnce 也适用于许多应用程序。

### 16.7.2 发布应用程序

ClickOnce 需要知道的信息都包含在两个清单文件中。为 ClickOnce 部署发布应用程序的过程就是生成清单，把文件放在正确的位置。清单文件可以在 Visual Studio 2005 中生成，还可以使用一个命令行工具 mage.exe，它还有一个带 GUI 的版本 mageUI.exe。

在 Visual Studio 2005 中创建清单文件有两种方式。在 Project Properties 对话框的 Publish 选项卡底部有两个按钮，一个是 Publish Wizard，另一个是 Publish Now。Publish Wizard 要求回答几个应用程序的部署问题，然后生成清单文件，把所有需要的文件复制到部署位置。

Publish Now 按钮使用在 Publish 选项卡中设置的值，创建清单文件，并把文件复制到部署位置。

为了使用命令行工具 mage.exe，必须传送各个 ClickOnce 属性的值。使用 mage.exe 可以创建和更新清单文件。在命令提示中输入 mage.exe /help，就会显示传送所需值的语法。

mage.exe 的 GUI 版本 (mageUI.exe) 类似于 Visual Studio 2005 中的 Publish 选项卡。使用 GUI 工具可以创建和更新应用程序清单和部署清单文件。

ClickOnce 应用程序会显示在“添加/删除程序”控制面板选项上，这与其他安装的应用程序一样。一个主要区别是用户可以选择卸载应用程序或回退到以前的版本。ClickOnce 在 ClickOnce 应用程序缓存中保存以前的版本。

### 16.7.3 ClickOnce 设置

两个清单文件都有几个属性。最重要的属性是应用程序应从什么地方部署。必须指定应用程序的从属文件。Publish 选项卡上有一个 Application Files 按钮，单击它会打开一个对话框，输入应用程序需要的所有程序集。单击 Prerequisite 按钮会显示一个与应用程序一

## 419

### 第 部分 Visual Studio

起安装的通用预装程序列表。可以选择从发布应用程序的位置上安装预装程序，也可以从供应商的 Web 站点上安装预装程序。

单击 Update 按钮会显示一个对话框，其中包含了如何更新应用程序的信息。当有应用

程序的新版本时，可以使用 ClickOnce 更新应用程序。其选项包括：每次启动应用程序时检查是否有更新版本，或在后台检查更新版本。如果选择了后台选项，就可以输入两次检查的间隔时间。此时可以使用允许用户拒绝或接收更新版本的选项。它可用于在后台进行更新，这样用户就不知道进行了更新。下次运行应用程序时，会使用新版本替代旧版本。还可以给更新文件使用另一个位置存储。这样，原安装软件包在一个位置，用于给新用户安装应用程序，而所有的更新版本放在另一个位置上。

安装应用程序时，可以让它以在线模式或离线模式下运行。在离线模式下，应用程序可以从“开始”菜单中运行，就好像它是用 Windows Installer 安装的。在线模式表示应用程序只能在有安装文件夹的情况下运行。

#### 16.7.4 应用程序缓存

用 ClickOnce 发布的应用程序不能安装在 Program Files 文件夹中，它们会放在应用程序缓存中，应用程序缓存位于当前用户的 Document's and Settings 文件夹的 Local Settings 子文件夹下。控制部署的这个方面，可以把应用程序的多个版本同时放在客户机上。如果应用程序设置为在线运行，就会保留用户访问过的每个版本。对于设置为本地运行的应用程序，会保留当前版本和以前的版本。

所以，把 ClickOnce 应用程序回退到以前的版本是一个非常简单的过程。如果用户进入“添加/删除程序”控制面板选项，所显示的对话框将允许删除 ClickOnce 应用程序或回退到以前的版本。管理员可以修改清单文件，使之指向以前的版本。之后，下次用户启动应用程序时，会检查是否更新版本。应用程序不是查找要部署的新程序集，而是恢复以前的版本，但不需要用户的干涉。

#### 16.7.5 安全性

通过 Internet 或内联网部署的应用程序，其安全性或可信赖设置比安装到本地驱动器上的应用程序低。例如，如果应用程序从 Internet 上启动或部署，它就默认位于 Internet Security 区域。也就是说，它不能访问文件系统。如果应用程序是从文件共享中安装的，就在 Intranet 区域中运行。

如果应用程序需要的信赖度高于默认值，它就会要求用户获得运行应用程序所需的权限。这些权限在应用程序清单的 trustInfo 元素中设置。只需要授予这个设置中的权限。如果应用程序需要文件访问权限，就不需要授予 Full Trust 权限，只要文件访问权限即可。

另一个选项是使用 Trusted Application Deployment。Trusted Application Deployment 是给整个企业授予权限的方式，不需要提示用户。给每台客户机标记一个信任许可发行程序，这可以通过公钥加密法完成。一般一个公司只有一个信任许可发行程序。一定要把发行程序的私钥放在安全的地方。

### 420

#### 第15 章部 署

信任许可从发行程序上请求。所请求的信任级别是信任许可配置的一部分。还必须给许可发行程序提供用于标记应用程序的公钥。所创建的许可包含用于标记应用程序的公钥和许可发行程序的公钥。这个信任许可会嵌入到部署清单中。最后一步是用自己的密钥对标记部署清单。现在应用程序就可以部署了。

在客户打开部署清单时，Trust Manager 会确定 ClickOnce 应用程序是否有较高的信任级别。首先查看发行程序的许可。如果它是有效的，就比较许可中的公钥和用于标记应用程序的公钥。如果它们匹配，就给应用程序授予需要的权限。

#### 16.7.6 高级选项

前面讨论的安装过程功能非常强大，可以完成许多工作。在安装过程中还可以控制许多方面。例如，可以使用 Visual Studio 2005 中的各种编辑器建立条件安装，或者添加注册键和定制对话框。SampleClientSetupSolution 示例就启用了所有这些高级选项。

##### 1. 文件系统编辑器

文件系统编辑器允许指定组成应用程序的各种文件和程序集部署到目标机器的什么

地方。默认情况下会显示一组标准的部署文件夹。使用该编辑器可以添加任意多个定制和特定文件夹。还可以给应用程序添加桌面快捷方式和“开始”菜单快捷方式。组成部署的任何文件都必须在文件系统编辑器中引用。

## 2. 注册编辑器

注册编辑器允许给注册表添加键和数据。在第一次显示该编辑器时，会显示一组标准的主键：

HKEY\_CLASSES\_ROOT  
HKEY\_CURRENT\_USER  
HKEY\_LOCAL\_MACHINE  
HKEY\_USERS

HKEY\_CURRENT\_USER 和 HKEY\_LOCAL\_MACHINE 在 Software/[manufacturer] 键中包含了额外的条目，其中 manufacturer 是在部署项目的 Manufacturer 属性中输入的信息。要添加额外的键和值，应在编辑器的左边突出显示一个主键，从主菜单中选择 Action，再选择 New 选择要添加的键或值类型。重复这个步骤，直到得到了需要的所有注册设置为止。如果在左边的窗格中选择了 Registry on Target Machine 选项，再选择 Action 菜单，就会看到一个 Import 选项，该选项允许导入已定义好的 .reg 文件。

要为键创建默认值，必须先为该键输入值。然后在右边或值窗格上选择值名称。从 File 菜单中选择 Rename，并删除该名称。按下回车键，值名称就替换为 (Default)。还可以在该编辑器中为子键和值设置一些属性。前面还没有讨论的唯一属性是 DeleteAt Uninstall。设计良好的应用程序应在卸载过程中删除由该应用程序添加的所有键。默认设置是不删除键。

注意，维护应用程序设置的首选方法是使用基于 XML 的配置文件。与注册表项相比，

## 421

### 第 部 分 Visual Studio

这些文件提供了非常大的灵活性，更容易恢复和备份。

## 3. 文件类型编辑器

文件类型编辑器用于建立文件和应用程序之间的关系。例如，在双击 .doc 文件时，就会在 Word 中打开该文件。使用这个编辑器可以为应用程序创建这种关系。

为了添加关系，从 Action 菜单中选择 File Types on Target Machine，然后选择 Add File Type。在属性窗口中，可以设置关系的名称。在 Extension 属性中添加应与应用程序相关的文件扩展名。不要输入句点，可以用分号隔开多个扩展名，例如 ex1;ex2。在 Command 属性中选择省略号按钮。接着选择要与特定的文件类型相关的文件（一般是可执行文件）。注意任何一个扩展名都应只与一个应用程序相关。

默认情况下，编辑器会显示 &Open as the Document Action。我们还可以添加其他选项。

编辑器中显示的动作顺序就是用户右击文件类型时在弹出的菜单中显示的动作顺序。注意第一项总是默认动作。可以为动作设置 Argument 属性，这是用于启动应用程序的命令行参数。

## 4. 用户界面编辑器

有时在安装过程中要求用户提供更多的信息。用户界面编辑器可用于为一组预定义的对话框指定属性。该编辑器分为两个部分 Install 和 Admin。一个用于标准安装，另一个用于管理员安装。每个部分又分为 3 个子部分：Start、Progress 和 End。这些子部分表示安装过程的 3 个基本阶段，如图 15-4 所示。

图 15-4

## 422

### 第15 章部 署

表 15-6 列出了可以添加到项目中的对话框类型。

表 15-6

对话 框 说 明

Checkboxes 至多包含 4 个复选框。每个复选框都包含 Label、Value 和 Visible 属性

Confirm Installation 允许用户在安装开始之前确认各个设置

Customer Information 包含集合名称、公司名称和系列号的编辑字段。公司名称和系列号是可选的

Finished 在安装过程的最后显示

Installation Address 用于 Web 应用程序，显示一个对话框，让用户选择另一个安装 URL

Installation Folder 用于客户应用程序，显示一个对话框，让用户选择另一个安装文件夹

License Agreement 显示许可协议，该协议位于 LicenseFile 属性指定的文件中

Progress 在安装过程中显示一个进度指示器，说明当前的安装状态

RadioButtons 至多包含 4 个单选按钮，每个单选按钮都包含 Label 和 Value 属性

Read Me 显示 readme 信息，该信息包含在 ReadMe 属性指定的文件中

Register User 执行一个在注册过程中指导用户操作的应用程序，该应用程序必须在安装项目中提供

Splash 显示一个位图图像

TextBoxes 至多包含 4 个文本框，每个文本框都包含 Label、Value 和 Visible 属性

Welcome 包含两个属性 WelcomeText 和 CopyrightWarning，它们都是字符串属性

这些对话框还包含一个设置横幅位图的属性，大多数对话框还包含一个设置横幅文本的属性。还可以在编辑器窗口中向上或向下拖动对话框，改变它们的显示顺序。

获得了一些信息后，现在的问题就是如何使用它们。此时就要使用项目中大多数对象都包含的 Condition 属性了。Condition 属性必须是 true，安装步骤才能继续下去。例如，假定安装程序包含 3 个可选的安装组件。在这种情况下，就可以添加一个对话框，其中包含 3 个复选框。该对话框应在 Welcome 对话框之后、Confirm Installation 对话框之前的某个地方显示。修改每个复选框的 Label 属性，描述具体的动作。第一个动作是“安装组件 A”，第二个动作是“安装组件 B”，依次类推。在文件系统编辑器中选择表示组件 A 的文件。假定对话框中复选框的名称是 CHECKBOXA1，则文件的 Condition 属性就是 CHECKBOXA1=Checked，即如果 CHECKBOXA1 被选中，就安装文件，否则就不安装。

## 5. 定制动作编辑器

定制动作编辑器允许定义在安装的某些阶段进行的定制步骤。定制动作应事先创建好，它可以包含 DLL、EXE 脚本或 Installer 类。动作可以包含不能在标准部署项目中定义的特定步骤。动作应在部署的 4 个特定点执行。在第一次启动编辑器时，就会看到项目中的这 4 个点，如图 15-5 所示。

### 423

第 部分 Visual Studio

图 15-5

Install: 动作在安装阶段的最后执行

Commit: 动作在安装完成后执行，不记录错误

Rollback: 动作在回退阶段完成后执行

Uninstall: 动作在卸载完成后执行

要添加动作，首先选择要执行动作的安装阶段。再从 Action 菜单中选择 Add Custom Action 菜单选项，打开文件系统对话框，这表示包含动作的组件必须是部署项目的一部分。由于动作在要部署的目标机器上执行，所以应列在文件系统编辑器中。

在添加完动作后，可以从表 15-7 中选择一个或多个属性。

表 15-7

参 数命令行参数

Condition Windows Installer 条件，若要执行动作，该条件必须为 true

CustomDataAction 可用于动作的定制数据

EntryPoint 包含动作的定制 DLL 的入口。如果动作包含在可执行文件中，这个属性就不起作用

InstallerClass 布尔值，如果设置为 true, 就指定动作是一个 .NET 类 Project Installer

Name 动作的名称，默认为动作的文件名

SourcePath 动作在开发机器上的路径

## 424

### 第15 章部 署

由于动作是在部署项目外部开发的代码，所以可以给应用程序自由添加专业化的外观。但要注意这些动作都在相关的阶段完成后发生。如果选择 Install 阶段，动作就会在安装阶段完成后发生。如果要在该过程之前执行动作，就应创建一个启动条件。

#### 6. 启动条件编辑器

启动条件编辑器允许指定在安装继续之前必须满足的一些条件。启动条件可以分为不同的条件类型。基本启动条件是 File Search、Registry Search 和 Windows Installer Search。在编辑器第一次启动时，会看到两个组 Search Target Machine 和 Launch Conditions，如图 15-6 所示。一般需要进行搜索，根据该搜索的成功或失败来执行条件。这是通过设置搜索的 Property 属性来实现的。可以在安装过程中访问 Property 属性，在其他动作的 Condition 属性中也可以检查该属性。还可以在编辑器中添加 Launch Condition。在这个条件下把 Condition 属性设置为搜索的 Property 属性值。在条件中可以指定一个 URL，用于下载所搜索的文件、注册键或安装组件。注意在图 15-6 中，默认添加了一个 .NET Framework 条件。

图 15-6

File Search 搜索文件或文件类型。可以设置许多不同的、与文件相关的属性，来确定如何搜索文件，这些属性包括文件名、文件夹位置、各种日期值，版本信息和大小。还可以设置要搜索的子文件夹数目。

Registry Search 允许搜索键和值，还允许设置搜索的根键。

Windows Installer Search 搜索指定的安装组件。这个搜索由 GUID 执行。

## 425

### 第 部分 Visual Studio

启动条件编辑器提供了两个预打包的启动条件，一个是 .NET Framework 启动条件，它允许搜索运行库的特定版本，另一个启动条件是搜索 MDAC 的特定版本，该搜索使用注册表搜索，来查找相关的 MDAC 注册表项。

## 16.8 小结

部署软件对桌面软件的开发人员来说比较困难。随着 Web 站点越来越复杂，部署基于服务器的软件也变得困难起来。本章探讨了 Visual Studio 2005 和 .NET Framework 2.0 版本的部署选项和功能；使部署更容易完成，错误也较少。

在阅读完本章后，您应能创建部署软件包，解决几乎所有的部署问题。客户应用程序可以在本地部署，或通过 Internet 或内联网来部署。本章还介绍了部署项目的扩展特性和部署项目的配置方式。我们可以使用 No Touch Deployment 和 ClickOnce 部署应用程序。ClickOnce 的安全特性为部署客户应用程序提供了一种安全、有效的方式。使用部署项目安装 Web 应用程序还可以更轻松地完成 IIS 的配置。如果预编译应用程序，发布 Web 站点还有额外的好处。\_\_

## 第 31 章 Windows 窗体

基于 Web 的应用程序在过去几年非常流行。从管理员的角度来看，把所有的应用程序逻辑放在一个中央服务器上是非常吸引人的。但部署基于客户的软件会非常困难，特别是部署基于 COM 的客户软件。基于 Web 的应用程序的缺点是它们不能提供丰富的用户体验。.NET Framework 允许开发人员创建丰富、智能的客户应用程序，而且不再有部署问题和以前的 DLL Hell。无论选择 Windows 窗体还是 WPF(参见第 34 章)，客户应用程序都不再难以开发或部署。

Windows 窗体已经对 Windows 开发产生了影响。当应用程序处于初始设计阶段时，是建立基于 Web 的应用程序还是建立客户应用程序已经很难抉择了。Windows 客户应用程序开发起来非常快速和高效，它们可以为用户提供丰富的体验。

Visual Basic 开发人员对 Windows 窗体应比较熟悉。创建新窗体(也称为窗口或对话框)也采用把控件从工具箱拖放到窗体设计器上的方式。但是，如果您在创建消息泵和监视消息时使用的是 C 样式的传统 Windows 编程，或者您是一位 MFC 程序员，就会发现现在可以获得需要的低级内部功能了。现在可以重写 wndproc，捕获这些消息，但常常并不是真需要它们。

本章将主要介绍 Windows 窗体的如下方面：

Form 类

Windows 窗体的类层次结构

## System.Windows.Forms 命名空间中的控件和组件

菜单和工具栏

创建控件

创建用户控件

### 31.1 创建 Windows 窗体应用程序

首先需要创建一个 Windows 窗体应用程序。下面的示例创建了一个空白窗体，并把它显示在屏幕上。这个示例没有使用 Visual Studio 2008，而是在文本编辑器中输入代码，使用命令行编译器进行编译。下面是代码清单：

```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
    public class MyForm : System.Windows.Forms.Form
    {
        public MyForm()
        {
        }

        [STAThread]
        static void Main()
        {
            Application.Run(new MyForm());
        }
    }
}
```

在编译和运行这个示例时，会得到一个没有标题的小空白窗体。该窗体没有什么实际功能，但它却是一个 Windows 窗体。

代码中有两个地方需要注意。第一个是使用继承功能来创建 MyForm 类。下面的代码声明 MyForm 派生于 System.Windows.Forms。

```
public class MyForm : System.Windows.Forms.Form
```

Form 类是 System.Windows.Forms 命名空间的一个主要类。代码的其他部分如下：

```
[STAThread]
static void Main()
{
    Application.Run(new MyForm());
}
```

Main 是 C#客户应用程序的默认入口。一般在大型应用程序中，Main()方法不位于窗体中，而是位于类中，它负责完成需要的启动处理。在本例中，我们在项目属性对话框中设置启动的类名。注意属性[STAThread]，它把 COM 线程模型设置为单线程单元(Single-Threaded Apartment, STA)。COM 交互操作需要 STA 线程模型，默认认为添加到 Windows 窗体项目中。

Application.Run()方法负责启动标准的应用程序消息循环。它有 3 个重载版本：第一个重载版本不带参数，第二个重载版本把 ApplicationContext 对象作为其参数，本例中的第三个重载版本把窗体对象作为其参数。在这个示例中，MyForm 对象是应用程序的主窗体，这表示在关闭这个窗体时，应用程序就结束了。使用 ApplicationContext 类，可以对主消息循环何时结束和应用程序何时退出有更多的控制权。

Application 类包含一些非常有用的功能。它提供了一些静态方法和属性，用于控制应用程序的启动和停止过程，访问由应用程序处理的 Windows 消息。表 31-1 列出了其中一些比较有用的方法和属性。

表 31-1

方法/属性	说 明
CommonAppDataPath	对应用程序的所有用户都通用的数据路径。一般是 BasePath\Company Name\Product Name\Version，其中 BasePath 是 C:\Documents and Settings\username\ApplicationData。如果该路径不存在，就创建一个
ExecutablePath	这是启动应用程序的可执行文件的路径和文件名

(续表)

方法/属性	说 明
LocalUserAppDataPath	类似于 CommonAppDataPath，但这个属性支持漫游
MessageLoop	如果在当前线程上存在消息循环，就返回 True，否则返回 false
StartupPath	类似于 ExecutablePath，但不返回文件名
AddMessageFilter	用于预处理消息。在基于 IMessengerFilter 的对象上执行，消息可以从消息循环中过滤出来，或者在消息发送到循环中之前进行特殊的处理
DoEvents	类似于 Visual Basic 的 DoEvents 语句，允许处理队列中的消息
EnableVisualStyles	允许对应用程序的各种可视化元素使用 XP 可视化样式。它有两个重载版本，接收清单信息。一个重载版本的参数是清单流，另一个重载版本的参数是清单所在的完整名称和路径
Exit 和 ExitThread	Exit 结束所有当前运行的消息循环，并退出应用程序。ExitThread 只结束消息循环，关闭当前线程上的所有窗口

在 Visual Studio 2008 中生成这个示例时，它会是什么样子？首先要注意，创建了两个文件，其原因是 Visual Studio 2008 利用了 Framework 的部分类特性，把设计器生成的代码放在一个独立的文件中。使用默认名称 Form1，这两个文件就是 Form1.cs 和 Form1.Designer.cs。除非选择了 Project 菜单中的 Show All Files 选项，否则在 Solution Explorer 中看不到 Form1.Designer.cs。下面是 Visual Studio 2008 为两个文件生成的代码。第一个文件是 Form1.cs：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

```
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace VisualStudioForm
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

这很简单，其中包含一些 using 语句和一个简单的构造函数。接着是 Form1.Designer.cs 的代码：

```
namespace VisualStudioForm
{
    partial class Form1
    {
        /// < summary >
        /// Required designer variable.
        /// < /summary >
        private System.ComponentModel.IContainer components =
null;
        /// < summary >
        /// Clean up any resources being used.
        /// < /summary >
        /// < param name="disposing" > true if managed resources
should be
        disposed; otherwise, false. < /param >
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
        #region Windows Form Designer generated code
        /// < summary >
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// < /summary >
    }
}
```

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode =
System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
#endregion
}
```

窗体的设计器文件一般不应直接编辑。唯一的例外是要在 Dispose 方法中进行特殊的处理。 InitializeComponent 方法详见本章后面的内容。

首先，这个示例应用程序的代码比简单的命令行示例长。在类的开头有好几个 using 语句，在本例中大多数是不必要的。但保留它们并无大碍。类 Form1 派生于 System.Windows.Forms，与前面的 Notepad 示例一样，但代码从一开始就不同。Form1.Designer 文件的第一行代码如下：

```
private System.ComponentModel.Container components =
null;
```

在这个示例中，这行代码并没有做什么工作。当给窗体添加组件时，也就把该组件添加给了组件对象，该组件对象是一个容器。添加到这个容器中的原因与窗体的释放有关。窗体类支持 IDisposable 接口，因为它是在 Component 类中执行的。在组件添加到组件容器中时，容器将确保组件被正确地跟踪，并在释放窗体时释放它。如果查看代码中的 Dispose 方法，就可以看到它：

```
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
```

在此可以看到，在调用 Dispose 方法时，也会调用组件对象的 Dispose 方法，因为组件对象包含其他组件，所以它们也会被释放。

Form1 类的构造函数在 Form1.cs 中，如下所示：

```
public Form1()
{
    InitializeComponent();
}
```

注意对 InitializeComponent()的调用。InitializeComponent()在 Form1.Designer.cs 中，顾名思义，InitializeComponent()初始化了添加到窗体上的所有控件，还初始化了窗体的属性。在本示例中，InitializeComponent()如下所示：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode =
        System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
```

这是很基本的初始化代码。该方法与 Visual Studio 的设计器相关联。使用设计器修改窗体时，这些改动会在 InitializeComponent()中反映出来。如果在 InitializeComponent()中修改了任意类型的代码，下次在设计器中进行修改时，这些改动就会丢失。每次在设计器中进行修改后，InitializeComponent()都会重新生成。如果需要为窗体或窗体上的控件和组件添加其他初始化代码，就应在调用 InitializeComponent()后添加。InitializeComponent()还负责实例化控件，这样在 InitializeComponent()之前所有引用控件的调用都会失败，并生成一个空引用异常。

要在窗体上添加控件或组件，可以按下 Ctrl+Alt+X 或者在 Visual Studio 2008 的 View 菜单中选择 Toolbox。此时 Form1 应处于设计模式。在 Solution Explorer 中右击 Form1.cs，从弹出的菜单中选择 View Designer。选择 Button 控件，把它拖放到设计器的窗体上。也可以双击该控件，把它添加到窗体上。对 TextBox 控件进行相同的操作。

在窗体上添加了 TextBox 控件和 Button 控件后，InitializeComponent()就会扩展，包含如下代码：

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(77,
137);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(67,
```

```
75);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(100, 20);
this.textBox1.TabIndex = 1;
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F,
13F);
this.AutoScaleMode =
System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 264);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
this.PerformLayout(false);
this.PerformLayout();
}
```

如果查看该方法中的前 3 行代码 ,就会看到 TextBox 控件和 Button 控件被实例化了。注意给控件指定的名称 textBox1 和 button1。默认情况下 ,设计器会使用控件的名称 ,并在该名称的最后添加一个整数值。在添加另一个按钮时 ,设计器会使用名称 button2 ,依次类推。下一行代码是 SuspendLayout 和 ResumeLayout 对的部分。SuspendLayout()临时挂起控件第一次初始化时发生的布局事件。在该方法的最后 ,将调用 ResumeLayout()方法 ,把事件重置为正常状态。在包含许多控件的复杂窗体上 ,InitializeComponent()方法会非常长。

要修改控件的属性值 ,可以按下 F4 ,或从 View 菜单中选择 Properties Window。该窗口允许修改控件或组件的大多数属性。在 Properties 窗口中进行了修改后 ,Initialize Component()方法就会重新编写 ,以反映新属性值。例如 ,如果在 Properties 窗口中把 Text 属性改为 My Button ,InitializeComponent()就会包含如下代码 :

```
//
// button1
//
this.button1.Location = new System.Drawing.Point(77,
137);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button";
this.button1.UseVisualStyleBackColor = true;
```

如果使用的不是 Visual Studio 的编辑器，就需要在设计中包含 InitializeComponent() 类型函数。把所有这些初始化代码放在一个地方，有助于使构造函数更简洁，如果有多个构造函数，还需要确保能从每个构造函数中调用初始化代码。

### 类层次结构

在设计和构建定制控件时，理解层次结构是非常重要的。如果定制控件派生于已有的控件，例如对于带有额外属性和方法的文本框，就应使定制控件派生于文本框控件，再重写、添加属性和方法，以满足要求。但是，如果创建的控件与.NET Framework 中的已有控件不匹配，就必须从 3 个基类中派生：如果需要自动滚动功能，就从 Control 或 Scrollable Control 中派生，如果控件应是其他控件的容器，就应从 ContainerControl 类中派生。

本章的剩余内容将介绍其中的许多类，它们如何协同工作，以及如何使用它们建立具有专业化外观的客户应用程序。

## 31.2 Control 类

System.Windows.Forms 命名空间中有一个特殊的类，它是每个控件和窗体的基类，这个类就是 System.Windows.Forms.Control。Control 类执行核心功能，创建用户所见的界面。Control 类派生于 System.ComponentModel.Component 类。Component 类为 Control 类提供了必要的基础结构，在把控件拖放到设计界面上以及包含在另一个对象中时需要它。Control 类为派生于它的类提供了一个很长的功能列表。这个列表太长，不能在这里全部列出，所以我们仅介绍 Control 类提供的比较重要的功能。本章的后面在介绍基于 Control 类的特定控件时，将在一些示例代码中论述属性和方法。下面几小节将按照功能组合方法和属性，把相关的功能放在一起进行讨论。

### 31.2.1 大小和位置

控件的大小和位置由属性 Height、Width、Top、Bottom、Left、Right 以及辅助属性 Size 和 Location 确定。区别是 Height、Width、Top、Bottom、Left、Right 属性值都是一个整数，而 Size 的值使用一个 Size 结构来表示，Location 的值使用一个 Point 结构来表示。Size 结构和 Point 结构都包含 XY 坐标。Point 结构一般相对于一个位置，而 Size 结构是对象的高和宽。Size 和 Point 都位于 System.Drawing 命名空间。它们非常类似，因为它们都提供了 XY 坐标对，还拥有用于简单的比较和转换的重写运算符。例如，可以对两个 Size 结构执行相加操作。对于 Point 结构，加法运算符已进行了重写，可以把 Size 结构加到 Point 结构上，得到一个新的 Point。其结果是给某个位置加上某个距离值，得到一个新位置。如果动态创建窗体或控件，这是非常方便的。

Bounds 属性返回一个 Rectangle 对象，它表示一个控件区域。这个区域包含滚动条和标题栏。Rectangle 也位于 System.Drawing 命名空间。ClientSize 属性是一个 Size 结构，表示控件的客户区域，不包含滚动条和标题栏。

PointToClient 和 PointToScreen 方法是方便的转换方法，它们的参数是 Point 结构，返回一个 Point 结构。PointToClient 的 Point 参数表示屏幕坐标，该方法把屏幕坐标转换为基于当前客户对象的坐标。这非常便于进行拖放操作。PointToScreen 正好与之相反，它提取客户对象的坐标，把它们转换为屏幕

坐标。还有 RectangleToScreen 和 ScreenToRectangle 方法，它们具有相同的功能，只是用 Rectangle 结构代替 Point 结构。

Dock 属性确定子控件停放在父控件的哪条边上。DockStyle 枚举值用作其属性值。这个值可以是 Top、Bottom、Right、Left、Fill 和 None。Fill 会使控件的大小正好匹配父控件的客户区域。

Anchor 属性把子控件的一条边与父控件的一条边对齐，这与停靠不同，因为它不设置父控件的一条边，而是把到该边界的当前距离设置为常量。例如，如果把子控件的右边界与父控件的右边界对齐，并重新设置父控件的大小，子控件右边界到父控件右边界的距离将保持不变。Anchor 属性采用 AnchorStyles 枚举的值，其值是 Top、Bottom、Right、Left 和 None。通过设置该属性值，可以在重新设置父控件的大小时，动态地设置子控件的大小。这样，当用户重新设置窗体的大小时，按钮和文本框就不会被剪切或隐藏。

Dock 和 Anchor 属性与 Flow 和 Table 布局控件(详见本章后面的内容)一起使用时，可以创建非常复杂的用户窗口。对于包含许多控件的复杂窗体来说，窗口大小的重新设置比较困难。这些工具有助于完成这个任务。

### 31.2.2 外观

与控件外观相关的属性有 BackColor 和 ForeColor，它们把 System.Drawing.Color 对象作为其值。BackGroundImage 属性把基于 Image 的对象作为其值。System.Drawing.Image 是一个抽象类，用作 Bitmap 和 Metafile 类的基类。BackColorImageLayout 属性使用 ImageLayout 枚举设置图像在控件上的显示方式，其有效值是 Center、Tile、Stretch、Zoom 和 None。

Font 和 Text 属性处理文字的显示。要修改 Font 属性，需要创建一个 Font 对象。在创建 Font 对象时，要指定字体名称、字号和样式。

### 31.2.3 用户交互操作

用户交互操作最好描述为控件创建和响应的各种事件。一些比较常见的事件有 Click、DoubleClick、KeyDown、KeyPress、Validating 和 Paint。

鼠标事件 Click、DoubleClick、MouseDown、MouseUp、MouseEnter、MouseLeave 和 MouseHover 处理鼠标和控件的交互操作。如果处理 Click 和 DoubleClick 事件，每次捕获一个 DoubleClick 事件时，也会引发 Click 事件。如果处理不正确，就会出现我们不希望的结果。Click 和 DoubleClick 事件都把 EventArgs 作为其参数，而 MouseDown 和 MouseUp 事件把 MouseEventArgs 作为其参数。

MouseEventArgs 包含几个有用的信息，例如单击的按钮、按钮被单击的次数、鼠标轮制动器(鼠标轮上的凹槽)的数目和鼠标的当前 XY 坐标。如果可以访问这些信息，就必须处理 MouseDown 或 MouseUp 事件，而不是 Click 或 DoubleClick 事件。

键盘事件的工作方式与此类似：需要一些信息来确定处理什么事件。对于简单的情况，KeyPress 事件接收一个 KeyPressEventArg，它包含表示被按键的字符值 KeyChar。Handled 属性用于确定事件是否已处理。把 Handled 属性设置为 true，事件就不会由操作系统进行默认处理。如果需要被按的键的更多信息，则处理 KeyDown 或 KeyUp 事件会比较合适。它们都接收 KeyEventArg。KeyEventArg

中的属性包括 Ctrl、Alt 或 Shift 键是否被按下。KeyCode 属性返回一个 Keys 枚举值，表示被按下的键。与 KeyPressEventArgs.KeyChar 不同，KeyCode 属性指定键盘上的每个键，而不仅仅是字母数字键。KeyData 属性返回一个 Key 值，还设置修饰符。修饰符与值进行 OR 运算，指定是否同时按下了 Shift 或 Ctrl 键。KeyValue 属性是 Keys 枚举的整数值。Modifiers 属性包含一个 Keys 值，它表示被按下的修饰符键。如果选择了多个修饰符键，这些值就进行 OR 运算。键盘事件以下述顺序来引发：

- (1) KeyDown
- (2) KeyPress
- (3) KeyUp

Validating、Validated、Enter、Leave、GotFocus 和 LostFocus 事件都处理获得焦点(或被激活)和失去焦点的控件。在用户用 tab 键选择一个控件或用鼠标选择该控件时，该控件就获得了焦点。Enter、Leave、GotFocus 和 LostFocus 事件的功能似乎非常类似。GotFocus 和 LostFocus 事件是低级事件，与 Windows 消息 WM\_SETFOCUS 和 WM\_KILLFOCUS 相关。一般应尽可能使用 Enter 和 Leave 事件。Validating 和 Validated 事件在验证控件时发生。这些事件接收一个 CancelEventArgs，利用该参数，把 Cancel 属性设置为 true，就可以取消以后的事件。如果定制了验证代码，而且验证失败，就可以把 Cancel 属性设置为 true，且控件也不会失去焦点。Validating 事件在验证过程中发生，Validated 事件在验证过程后发生。这些事件的引发顺序如下：

- (1) Enter
- (2) GotFocus
- (3) Leave
- (4) Validating
- (5) Validated
- (6) LostFocus

理解这些事件的引发顺序是很重要的，可以避免不小心创建递归事件。例如，在控件的 LostFocus 事件中设置控件的焦点，就会创建一个消息死锁，且应用程序会停止响应。

#### 31.2.4 Windows 功能

System.Windows.Forms 命名空间是依赖 Windows 功能的少数几个命名空间之一。Control 类就是一个很好的示例。如果对 System.Windows.Forms.dll 进行反编译，就会看到 UnsafeNativeMethods 类的引用列表。.NET Framework 使用这个类封装所有的标准 Win32 API 调用。使用与 Win32 API 的交互操作，标准 Windows 应用程序的外观和操作方式就可以通过 System.Windows.Forms 命名空间获得。

支持与 Windows 交互操作的功能包括 Handle 和 IsHandleCreated 属性。Handle 属性返回一个包含控件 HWND(Windows 句柄)的 IntPtr。窗口句柄是唯一标识窗口的 HWND。可以将控件看作是一个窗口，所以它有相应的 HWND。可以使用 Handle 属性进行任意数量的 Win32 API 调用。

为了访问 Windows 消息，可以重写 WndProc 方法。该方法把一个 Message 对象作为其参数。Message 对象是 Windows 消息的一个简单封装器。它包含 HWnd、 LParam、 WParam、 Msg 和 Result 属性。如果希望由系统处理消息，就必须确保把消息传送给 base.WndProc(msg)方法。如果希望自己处理消息，就不需要传送消息。

### 31.2.5 杂项功能

一些条目较难分类，例如数据绑定功能。BindingContext 属性返回一个 BindingManagerBase 对象。DataBindings 集合包含一个 ControlBindingsCollection，它是控件的绑定对象集合，数据绑定详见第 32 章。

CompanyName、 ProductName 和 Product 版本提供了控件的初始数据及其当前版本。

Invalidate 方法允许使控件的一个区域失效，以进行重新绘制。可以使整个控件失效，或指定要失效的区域或矩形。这会把一个绘制消息传送给控件的 WindProc。还可以同时使子控件失效。

组成 Control 类的还有几十个属性、方法和事件。这个列表列出了比较常用的成员，希望您对可用的功能有一个大致的了解。

## 31.3 标准控件和组件

前一节介绍了控件常用的一些方法和属性。本节将讨论.NET Framework 提供的各种控件，解释每个控件提供的附加功能。下载的示例([www.wrox.com](http://www.wrox.com))包含一个示例应用程序 FormExample，这个应用程序是一个 MDI 应用程序(稍后讨论)，包含一个窗体 frmControls，其中包含许多具备基本功能的控件。图 31-1 显示了这个示例的外观。

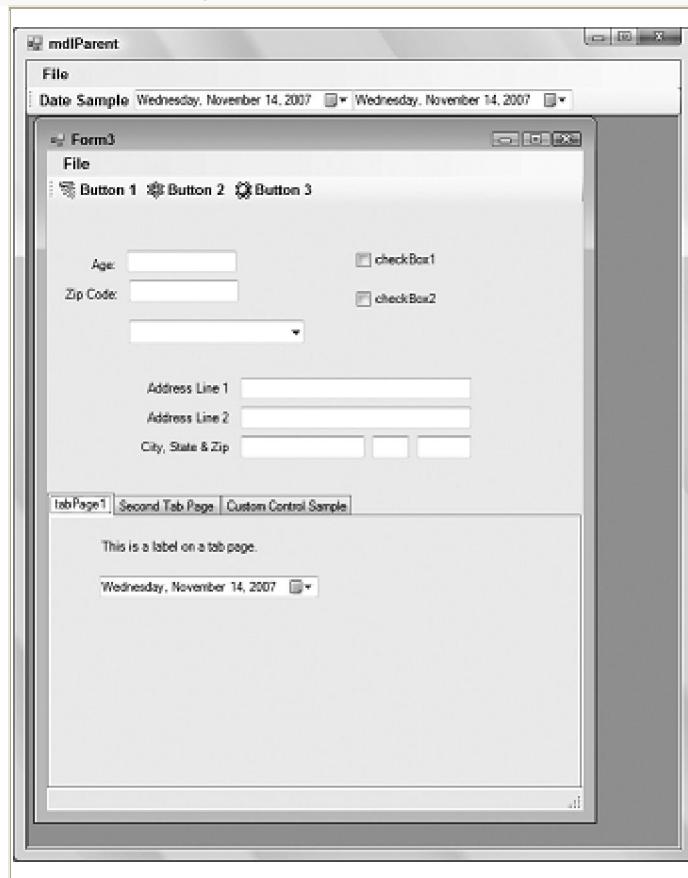


图 31-1

### 31.3.1 Button 控件

Button 类表示简单的命令按钮，派生自 ButtonBase 类。该类最常见的用法是编写处理按钮 Click 事件的代码。下面的代码执行 Click 事件的处理程序。在单击按钮时，会弹出一个显示按钮名称的消息框。

```
private void btnTest_Click(object sender,  
System.EventArgs e)  
{  
    MessageBox.Show(((Button)sender).Name + " was clicked.");  
}
```

在 PerformClick 方法中，可以模仿按钮上的 Click 事件，而无需用户单击按钮。NotifyDefault 方法把一个布尔值作为其参数，告诉按钮把它自己绘制为默认按钮。一般情况下，窗体上的默认按钮有略粗的边框。要把按钮标识为默认，可以把窗体上的 AcceptButton 属性设置为按钮。接着，在用户按下回车键时，就会引发默认按钮的单击事件。图 31-2 显示了标题为 Default 的默认按钮(注意黑色的边框)。

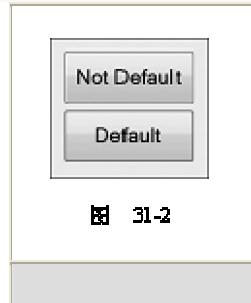


图 31-2

按钮可以包含图像和文本。图像通过 ImageList 对象或 Image 属性提供。ImageList 对象就是由放在窗体上的组件管理的一个图像列表。它们在本章的后面解释。

Text 和 Image 都包含 Align 属性，用以对齐按钮上的文本和图像。Align 属性使用 ContentAlignment 枚举的值。文本或图像可以与按钮的左、右、上、下边界对齐。

### 31.3.2 CheckBox 控件

CheckBox 控件也派生于 ButtonBase，用于接受来自用户的二状态或三状态响应。如果把 ThreeState 属性设置为 true，复选框的 CheckState 属性就可以是以下 3 个 CheckState 枚举值之一：

Checked：复选框有一个选中标记

Unchecked：复选框没有选中标记

Indeterminate：在这种状态下，复选框为灰显

Indeterminate 值只能在代码中设置，不能由用户设置。如果需要告诉用户选项还未设置，就可以使用这个值。如果希望使用布尔值，还可以使用 Checked 属性。

CheckedChanged 和 CheckStateChanged 事件在 CheckState 或 Checked 属性改变时发生。捕获的这些事件可以根据复选框的新状态设置其他值。在 frmControls 窗体类中，几个复选框的 CheckedChanged 事件由下面的方法处理：

```
private void checkBoxChanged(object sender, EventArgs e)
{
    CheckBox checkBox = (CheckBox)sender;
    MessageBox.Show(checkBox.Name + " new value is " +
    checkBox.Checked.ToString());
}
```

在每个复选框的 Checked 属性改变时，都会显示一个消息框，其中包含了改变的复选框名称和新值。

### 31.3.3 RadioButton 控件

最后一个派生自 ButtonBase 的控件是 RadioButton(单选按钮)。单选按钮一般用作一个组，有时称为选项按钮。单选按钮允许用户从几个选项中选择一个。当同一个容器中有多个 RadioButton 控件时，一次只能选择一个按钮。所以如果有 3 个选项，例如 Red、Green 和 Blue，如果 Red 选项被选中，而用户单击 Blue，则 Red 会自动取消选中。

Appearance 属性使用 Appearance 枚举值，即 Button 或 Normal。当选择 Normal 时，单选按钮看起来像一个小圆圈，在它的旁边有一个标签。选择按钮会填充圆圈，选择另一个按钮会取消对当前选中按钮的选择，使圆圈为空。当选中 Button 时，RadioButton 控件看起来像一个标准按钮，但工作方式类似于开关，选中是指焦点在位置中，取消选中是指正常状态或焦点在位置外。

CheckedAlign 属性确定圆圈与标签文本的相对位置，它可以在标签的顶部、左右两边或下方。

只要 Checked 属性的值改变，就会引发 CheckedChanged 事件。这样就可以根据控件的新值执行其他动作。

### 31.3.4 ComboBox 控件、ListBox 控件和 CheckedListBox 控件

ComboBox、ListBox 和 CheckedListBox 都派生于 ListControl 类。这个类提供了一些基本的列表管理功能。使用列表控件最重要的事是，给列表添加数据和选择数据。使用哪个列表一般取决于列表的用法和列表中数据的类型。如果需要选择多个选项，或用户需要在任意时刻查看列表中的几个项，最好使用 ListBox 和 CheckedListBox。如果一次只选择一个选项，就可以使用 ComboBox。

在使用列表框之前，必须先添加数据。为此，应给 ListBox.ObjectCollection 添加对象。这个集合可以使用列表的 Items 属性访问。由于该集合存储了对象，因此可以把任意有效的.NET 类型添加到列表中。要标识对象，需要设置两个重要的属性。第一个是 DisplayMember 属性，这个设置告诉列表控件，在列表中显示对象的哪个属性。另一个是 ValueMember 属性，它是要返回值的对象属性。如果在列表中添加了字符串，这两个属性就默认使用字符串值。示例应用程序中的 frmLists 窗体显示了如何把对象和字符串(也是对象)加载到列表框中。该例子使用 Vendor 对象作为列表数据。Vendor 对象

只包含两个属性 Name 和 PhoneNo。DisplayMember 属性设置为 Name，这告诉列表控件，把列表中 Name 属性的值显示给用户。

访问列表控件中的数据有两种方式，如下面的代码示例所示。列表中加载了 Vendor 对象，设置了 DisplayMember 和 ValueMember 属性。这段代码在示例应用程序的 frmLists 窗体中。

首先是 LoadList 方法，它给列表加载了 Vendor 对象或一个包含供应商姓名的简单字符串。选中一个选项按钮，看看在列表中加载了哪些值：

```
private void LoadList(Control ctrlToLoad)
{
    ListBox tmpCtrl = null;

    if (ctrlToLoad is ListBox)
        tmpCtrl = (ListBox)ctrlToLoad;

    tmpCtrl.Items.Clear();
    tmpCtrl.DataSource = null;

    if (radioButton1.Checked)
    {
        //load objects
        tmpCtrl.Items.Add(new Vendor("XYZ Company",
            "555-555-1234"));
        tmpCtrl.Items.Add(new Vendor("ABC Company",
            "555-555-2345"));
        tmpCtrl.Items.Add(new Vendor("Other Company",
            "555-555-3456"));
        tmpCtrl.Items.Add(new Vendor("Another Company",
            "555-555-4567"));
        tmpCtrl.Items.Add(new Vendor("More Company",
            "555-555-6789"));
        tmpCtrl.Items.Add(new Vendor("Last Company",
            "555-555-7890"));
        tmpCtrl.DisplayMember = "Name";
    }
    else
    {
        tmpCtrl.Items.Clear();
        tmpCtrl.Items.Add("XYZ Company");
        tmpCtrl.Items.Add("ABC Company");
        tmpCtrl.Items.Add("Other Company");
        tmpCtrl.Items.Add("Another Company");
        tmpCtrl.Items.Add("More Company");
        tmpCtrl.Items.Add("Last Company");
    }
}
```

在列表中加载了数据后，就可以使用 SelectedItem 和 SelectedIndex 属性获取数据。SelectedItem 返回当前选中的对象。如果列表设置为允许选择多个选项，就不能保证返回选中的选项。此时，应使用 SelectObject 集合。它包含列表中当前选中的所有选项。

如果需要特定索引的选项，可以使用 Items 属性访问 ListBox.ObjectCollection。这是一个标准的.NET 集合类，所以该集合中的项可以用与其他集合类相同的方式访问。

如果使用 DataBinding 填充列表，SelectedValue 属性就会返回选中对象内设置为 ValueMember 属性的属性值。如果 Phone 设置为 ValueMember，SelectedValue 就从选中的项中返回 Phone 值。要使用 ValueMember 和 SelectedValue，列表必须通过 DataSource 属性来加载。必须先使用对象加载 ArrayList 或其他基于 IList 的集合，再给列表赋予 DataSource 属性。下面的小例子演示了这个方法：

```
listBox1.DataSource = null;
System.Collections.ArrayList lst = new
System.Collections.ArrayList();
lst.Add(new Vendor("XYZ Company", "555-555-1234"));
lst.Add(new Vendor("ABC Company", "555-555-2345"));
lst.Add(new Vendor("Other Company", "555-555-3456"));
lst.Add(new Vendor("Another Company", "555-555-4567"));
lst.Add(new Vendor("More Company", "555-555-6789"));
lst.Add(new Vendor("Last Company", "555-555-7890"));
listBox1.Items.Clear();
listBox1.DataSource = lst;
listBox1.DisplayMember = "Name";
listBox1.ValueMember = "Phone";
```

使用 SelectedValue，但未使用 DataBinding，会导致 NullException 错误。

下面的代码显示了访问列表中数据的语法：

```
//obj is set to the selected Vendor object
obj = listBox1.SelectedItem;

//obj is set to the Vendor object with index of 3 (4th
object)
obj = listBox.Items[3];

//obj is set to the values of the Phone property of the
selected vendor object
//This example assumes that databinding was used to
populate the list
listBox1.ValueMember = "Phone";
obj = listBox1.SelectedValue;
```

注意，这些方法的返回类型都是 object。要使用 obj 的值，必须把它转换为正确的数据类型。

ComboBox 的 Items 属性返回 ComboBox.ObjectCollection。ComboBox 组合了编辑控件和列表框。把一个 DropDownListStyle 枚举值传送给 DropDownListStyle 属性，就可以设置 ComboBox 的样式。表 31-2 列出了 DropDownListStyle 的各个值。

表 31-2

值	说    明
DropDown	组合框的文本部分是可以编辑的，用户可以输入值。用户必须单击箭头按钮，才能显示列表
DropDownList	文本部分不能编辑。用户必须从列表中选择
Simple	类似于 DropDown，但列表总是可见的

如果列表中的值比较宽，就可以使用 DropDownListWidth 属性改变控件下拉部分的宽度。

MaxDropDownItems 属性设置在显示列表的下拉部分时的最大项数。

FindString 和 FindStringExact 方法是列表控件的另外两个有用的方法。FindString 在列表中查找以传入字符串开头的第一个字符串。FindStringExact 查找与传入字符串匹配的第一个字符串。它们都返回找到的值的索引，如果没有找到，就返回-1。它们还可以将要搜索的起始索引整数作为参数。

### 31.3.5 DateTimePicker 控件

DateTimePicker 允许用户在许多不同的格式中选择一个日期或时间值(或两者)。可以以任何标准时间日期格式显示基于 DateTime 的值。Format 属性使用 DateTimePickerFormat 枚举，它可以把格式设置为 Long、Short、Time 或 Custom。如果 Format 属性设置为 DateTimePickerFormat.Custom，就可以把 CustomFormat 属性设置为表示格式的字符串。

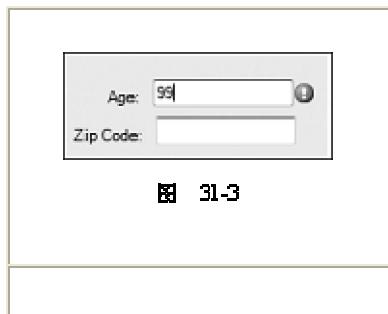
DateTimePicker 还包含 Text 属性和 Value 属性。Text 属性返回 DateTime 值的文本表示，Value 属性返回 DateTime 对象。还可以用 MinDate 和 Maxdate 属性设置日期所允许的最大值和最小值。

在用户单击向下箭头时，会显示一个日历，允许用户选择日历中的一个日期。DateTime Picker 还包含一些属性，这些属性可以设置标题、月份背景色和前景色，改变日期的外观。

ShowUpDown 属性确定控件上是否显示 UpDown 箭头。单击向上或向下箭头就可以改变当前突出显示的值。

### 31.3.6 ErrorProvider 组件

ErrorProvider 实际上并不是一个控件，而是一个组件。当把该组件拖放到设计器上时，它会显示在设计器下方的组件栏中。当存在一个错误条件或验证失败时，ErrorProvider 可以在控件的旁边显示一个图标。假定有一个 TextBox 控件用于输入年龄。业务规则是年龄值不能大于 65。如果用户试图输入大于 65 的年龄，就必须通知用户该年龄大于所允许的值，需要改变输入的值。有效值的检查在文本框的 Validated 事件中进行。如果验证失败，就调用 SetError 方法，传送引起错误的控件和一个字符串，将该错误告知用户。然后，一个图标开始闪烁，表示出现了一个错误，用户把鼠标放在该图标上时，会显示错误文本。图 31-3 显示了文本框中输入无效值时出现的图标。



可以为窗体上产生错误的每个控件创建一个 `ErrorHandler`，但如果窗体上有很多控件，这就会很麻烦。另一种选择是使用一个 `ErrorHandler`，在验证事件中调用被验证控件的 `IconLocation` 方法和一个 `ErrorIconAlignment` 枚举值。这个值设置图标与哪个控件对齐。接着调用 `SetError` 方法。如果没有错误，就调用 `SetError` 方法，并把空字符串作为它的错误字符串，清除 `ErrorHandler`。下面的示例说明了其工作原理：

```
private void txtAge_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
if (txtAge.TextLength > 0 && Convert.ToInt32(txtAge.Text)
> 65)
{
errMain.SetIconAlignment((Control)sender,
ErrorIconAlignment.MiddleRight);
errMain.SetError((Control)sender, "Value must be less
than 65.");
e.Cancel = true;
}
else
{
errMain.SetError((Control)sender, "");
}
}

private void txtZipCode_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
if (txtZipCode.Text.Length != 5)
{
errMain.SetIconAlignment((Control)sender, ErrorIconAlignment.MiddleRight);
errMain.SetError((Control)sender, "Must be 5 characters.");
e.Cancel = true;
}
else
{
errMain.SetError((Control)sender, "");
}
}
```

如果验证失败(例如 txtAge 中的年龄超过 65) ,就调用 ErrorProvider 组件 errMain 的 SetIcon 方法。该方法把图标放在验证失败的控件旁边。接着设置错误 ,这样 ,在用户把鼠标放在该图标上时 ,就会显示一个消息 ,告知用户哪个控件应为验证失败负责。

### 31.3.7 HelpProvider 组件

HelpProvider 类似于 ErrorProvider ,也是一个组件 ,而不是控件。HelpProvider 允许挂起控件 ,显示帮助主题。要把控件与 HelpProvider 关联起来 ,应调用 SetShowHelp 方法 ,给该方法传送该控件和一个确定是否显示帮助的布尔值。HelpNamespace 属性允许设置帮助文件。在设置 HelpNamespace 属性时 ,只要按下 F1 ,就会显示帮助文件 ,用 HelpProvider 注册的控件还会获得焦点。可以用 SetHelpKeyword 方法为帮助文件设置一个关键字。SetHelpNavigator 带一个 HelpNavigator 枚举值 ,用于确定显示帮助文件中的哪个元素。可以把它设置为特定的主题、索引、目录表或搜索页面。SetHelpString 把与帮助相关的文本字符串值关联到控件上。如果没有设置 HelpNamespace 属性 ,按下 F1 就会在弹出窗口中显示这个文本。下面在上一个示例中添加一个 HelpProvider :

```
helpProvider1.SetHelpString(txtAge, "Enter an age that  
is less than 65");  
  
helpProvider1.SetHelpString(txtZipCode, "Enter a 5 digit  
zip code");
```

### 31.3.8 ImageList 组件

ImageList 组件就是一个图像列表。一般情况下 ,这个属性用于存储一个图像集合 ,这些图像用作工具栏图标或 TreeView 控件上的图标。许多控件都包含 ImageList 属性。这个属性一般和 ImageIndex 属性一起使用。ImageList 属性设置为 ImageList 组件的一个实例 ,ImageIndex 属性设置为 ImageList 中应在控件中显示的图像的索引。使用 ImageIndex.Images 属性的 Add 方法可以把图像添加到 ImageList 组件中。Images 属性返回一个 ImageCollection。

两个最常用的属性是 ImageSize 和 ColorDepth。ImageSize 使用 Size 结构作为其值。其默认值是 16×16 ,但可以取 1~256 之间的任意值。ColorDepth 使用 ColorDepth 枚举作为其值。颜色深度值可以从 4 位~32 位。在.NET Framework 1.1 中 ,默认是 ColorDepth.Depth8Bit。

### 31.3.9 Label 控件

Label 控件一般用于给用户提供描述文本。文本可以与其他控件或当前系统状态相关。通常标签和文本框一起使用。标签为用户提供了在文本框中输入的数据类型的描述。标签控件总是只读的 ,用户不能修改 Text 属性的字符串值。但是 ,可以在代码中修改 Text 属性。UseMnemonic 属性允许用户启用访问键功能。在 Text 属性中 ,给一个字符前面加上宏符号&时 ,标签控件中的该字母就会加上下划线。按下 Alt 键和带有下划线的字母就会把焦点移动到 tab 顺序的下一个控件上。如果 Text 属性的文本包含一个宏符号 ,就应添加第二个宏符号 ,其后的字母将不带下划线。例如 ,如果标签文本是 Nuts & Bolts ,就应把属性设置为 Nuts && Bolts。由于标签控件是只读的 ,所以不能获得焦点。这就是焦点会移动到下一个控件上的原因。因此要记住 ,如果启用 Mnemonic ,就必须正确设置窗体上的 tab 顺序。

AutoSize 属性是一个布尔值，指定标签是否根据标签的内容自动设置其大小。在多语言应用程序中，Text 属性的长度会根据当前语言的不同而变化，此时就可以使用这个属性。

### 31.3.10 ListView 控件

ListView 控件允许以 4 种不同的方式显示条目。可以显示文本和可选的大图标、显示文本和可选的小图标、在垂直列表中显示文本和小图标、以及在详细视图中显示条目文本，并在列中显示子条目。这听起来应很熟悉，因为文件管理器的右边就用这种方式显示文件夹的内容。ListView 包含一个 ListViewItems 集合。ListViewItems 允许设置一个用于显示的 Text 属性，它的另一个属性 SubItems 包含在详细视图中显示的文本。

下面的示例演示了如何使用 ListView。这个示例包含国家的短列表。每个 CountryList 对象都包含国家名、国家简称和货币属性。下面是 CountryList 类的代码：

```
using System;

namespace FormSample
{
    public class CountryItem : System.Windows.Forms.ListViewItem
    {
        string _cntryName = "";
        string _cntryAbbrev = "";

        public CountryItem(string countryName,
                           string countryAbbreviation, string currency)
        {
            _cntryName = countryName;
            _cntryAbbrev = countryAbbreviation;
            base.Text = _cntryName;
            base.SubItems.Add(currency);
        }

        public string CountryName
        {
            get { return _cntryName; }
        }

        public string CountryAbbreviation
        {
            get { return _cntryAbbrev; }
        }
    }
}
```

注意 CountryList 类派生自 ListViewItem。这是因为我们只能给 ListView 控件添加基于 ListViewItem 的对象。在构造函数中，给 base.Text 属性传送国家名，给 base.SubItems 属性添加货币值。这会在列表中显示国家名，在 Details 视图的另一个单独的列中显示货币。

接着，需要在窗体的代码中给 ListView 控件添加几个 CountryItem 对象：

```
lvCountries.Items.Add(new CountryItem("United States", "US", "Dollar"));
lvCountries.Items[0].ImageIndex = 0;
lvCountries.Items.Add(new CountryItem("Great Britain", "GB", "Pound"));
lvCountries.Items[1].ImageIndex = 1;
lvCountries.Items.Add(new CountryItem("Canada", "CA", "Dollar"));
lvCountries.Items[2].ImageIndex = 2;
lvCountries.Items.Add(new CountryItem("Japan", "JP", "Yen"));
lvCountries.Items[3].ImageIndex = 3;
lvCountries.Items.Add(new CountryItem("Germany", "GM", "Deutch Mark"));
lvCountries.Items[4].ImageIndex = 4;
```

这里给 ListView 控件 lvCountries 的 Items 集合添加了一个新的 CountryItem。注意在把 CountryItem 添加到控件中后，才设置 CountryItem 的 ImageIndex 属性。有两个 ImageIndex 属性，一个用于大图标，一个用于小图标(SmallImageList 和 LargeImageList 属性)。要给两个 ImageList 指定不同的图像大小，应确保以相同的顺序给 ImageList 添加条目。这样，每个 ImageList 的索引就表示仅尺寸不同的相同图像。在本例中，ImageList 包含我们添加的每个国家的标记图标。

窗体的顶部是一个组合框 cbView，它列出了 4 个不同 View 枚举值。把这些条目添加到 cbView 中，如下所示：

```
cbView.Items.Add(View.LargeIcon);
cbView.Items.Add(View.SmallIcon);
cbView.Items.Add(View.List);
cbView.Items.Add(View.Details);
cbView.SelectedIndex = 0;
```

在 cbView 的 SelectedIndexChanged 事件中，添加下面这行代码：

```
lvCountries.View = (View) cbView.SelectedItem;
```

这行代码把 lvCountries 的 View 属性设置为在 ComboBox 控件中选中的新值。注意需要把它转换 View 类型，因为从 cbView 的 SelectedItem 属性中返回的是 object 类型。

最后，必须给 Column 集合添加列。这些列在 Details 视图中显示。在本例中添加了两列 Country Name 和 Currency。列的顺序是 ListViewItem 的 Text，然后是 ListViewItem.SubItems 集合中的每一项，

按照它们在集合中的顺序显示。添加列时，需要创建一个 `ColumnHeader` 对象，设置 `Text` 属性，还可以设置 `Width` 和 `Alignment` 属性。在创建 `ColumnHeader` 对象后，就可以把它添加到 `Column` 属性中。添加列的另一种方法是使用 `Columns.Add` 方法的重写版本，它允许传送 `Text`、`Width` 和 `Alignment` 值。下面是一个示例：

```
lvCountries.Column.Add("Country", 100, HorizontalAlignment.Left);
lvCountries.Column.Add("Currency", 100, HorizontalAlignment.Left);
```

如果把 `AllowColumnReorder` 属性设置为 `true`，用户就可以拖动列标题，重新安排列的顺序。

`ListView` 上的 `Checkboxes` 属性在 `ListView` 的条目旁边显示复选框，允许用户在 `ListView` 控件中选择多个条目。使用 `CheckedItems` 集合可以检查哪些项目被选中。

`Alignment` 属性设置 Large 和 Small 图标视图中图标的对齐方式。该值可以是 `ListViewAlignment` 枚举中的任意值，即 `Default`、`Left`、`Top`、`SnapToGrid`。`Default` 值允许用户把图标放在任意位置。在选择 `Left` 或 `Top` 时，条目应与 `ListView` 控件的左边或顶边对齐。在选择 `SnapToGrid` 时，条目会捕捉到 `ListView` 控件上不可见的栅格。`AutoArrange` 属性可以设置为布尔值，它会根据 `Alignment` 属性自动对齐图标。

### 31.3.11 PictureBox 控件

`PictureBox` 控件用于显示图像。图像可以是 BMP、JPEG、GIF、PNG、元文件或图标。`SizeMode` 属性使用 `PictureBoxSizeMode` 枚举确定图像在控件中的大小和位置。`SizeMode` 属性可以是 `AutoSize`、`CenterImage`、`Normal` 和 `StretchImage`。

设置 `ClientSize` 属性，可以改变 `PictureBox` 的显示区域大小。要加载 `PictureBox`，首先创建一个基于 `Iamge` 的对象。例如，要把 JPEG 文件加载到 `PictureBox` 中，需要编写如下代码：

```
Bitmap myJpeg = new Bitmap("mypic.jpg");
pictureBox1.Image = (Image) myJpeg;
```

注意需要转换回 `Image` 类型，因为这是 `Image` 属性所要求的。

### 31.3.12 ProgressBar 控件

`ProgressBar` 控件是较长操作的状态的可视化表示。它指示用户正在进行某个操作，用户应等待。`ProgressBar` 控件工作时要设置 `Minimum` 和 `Maximum` 属性。这些属性对应于进度指示器的最左端(`Minimum`)和最右端(`Maximum`)。设置 `Step` 属性，以确定每次调用 `PerformStep` 方法时数值的增量。还可以使用 `Increment` 方法，递增在方法调用中传入的值。`Value` 属性返回 `ProgressBar` 的当前值。

可以使用 `Text` 属性通知用户已完成了操作的百分数或还未处理的条目数。还有一个 `BackgroundImage` 属性可以定制进度条的外观。

### 31.3.13 TextBox 控件、RichTextBox 控件与 MaskedTextBox 控件

TextBox 控件是工具箱中最常用的控件之一。 TextBox、 RichTextBox 和 MaskedTextBox 控件都派生于 TextBoxBase。 TextBoxBase 提供了 MultiLine 和 Lines 属性， MultiLine 属性是一个布尔值，允许 TextBox 控件在多行中显示文本。文本框中的每一行都是字符串数组的一部分。这个数组通过 Lines 属性来访问。 Text 属性把整个文本框内容返回为一个字符串。 TextLength 是返回的文本字符串的总长。 MaxLength 属性把文本的长度限制为指定的数字。

SelectedText、 SelectionLength 和 SelectionStart 都处理文本框中当前选中的文本。选中的文本是控件获得焦点时突出显示的文本。

TextBox 控件增加了几个有趣的属性。 AcceptsReturn 属性是一个布尔值，允许 TextBox 把回车键接受为一个换行符，或者激活窗体上的默认按钮。这个属性设置为 true 时，按下回车键会在文本框中创建一个新行。 CharactorCasing 确定文本框中文本的大小写。 CharactorCasing 枚举包含 3 个值 Lower、 Normal 和 Upper。 Lower 会使所有的文本小写，Upper 则把所有的文本转变为大写，Normal 把文本显示为输入时的形式。 PasswordChar 属性用一个字符表示用户在文本框中输入文本时要显示给用户的内容，这通常用于输入密码和 PIN。 text 属性返回输入的文本，只有显示的内容会受这个属性的影响。

RichTextBox 是一个文本编辑控件，它可以处理特殊格式的文本。顾名思义，RichTextBox 控件使用 Rich Text Format(RTF) 处理特殊的格式。使用 Selection 属性 SelectionFont、 SelectionColor、 SelectionBullet 可以修改格式，使用 SelectionIndent、 SelectionRightIndent、 SelectionHangingIndent 可以修改段落的格式。所有 Selection 属性的工作方式都相同。如果有突出显示的文本段，对 Selection 属性的修改就会影响选中的文本。如果没有选中文本，这些修改就对当前插入点后面的文本起作用。

控件的文本可以使用 Text 属性或 Rtf 属性提取。 Text 属性只返回控件的文本，而 Rtf 属性返回带格式的文本。

LoadFile 方法可以用两种方式从文件中加载文本。它可以用一个表示文件名和路径的字符串，也可以使用一个流对象。还可以指定 RichTextBoxStreamType。表 31-3 列出了 RichTextBoxStreamType 的值。

表 31-3

值	说 明
PlainText	没有格式信息，包含 OLE 对象，允许使用空格
RichNoOleObjs	Rich 文本格式，但不包含 OLE 对象已经包含的空格
RichText	格式化的 RTF，且包含 OLE 对象
TextTextOleObjs	无格式文本，用文本替换 OLE 对象
UnicodePlainText	与 PlainText 相同，但编码为 Unicode

SaveFile 方法使用相同的参数，把控件中的数据存储在指定的文件中。如果文件已经存在，就覆盖它。

MaskedTextBox 可以限制用户在控件中输入的内容，它还可以自动格式化输入的数据。使用几个属性可以验证或格式化用户的输入。 Mask 属性包含覆盖字符串，覆盖字符串类似于格式字符串，使

用 Mask 字符串可以设置允许的字符数、允许字符的数据类型和数据的格式。基于 MaskedTextProvider 的类也提供了需要的格式化和验证信息。MaskedTextProvider 只能在它的构造函数中设置。

有 3 个不同的属性返回 MaskedTextBox 的文本。Text 属性返回控件的当前文本，它可以根据控件是否获得焦点而不同，而控件是否获得焦点取决于 HidePromptOnLeave 属性的值。该属性是一个字符串，告诉用户应输入什么内容。InputText 属性总是只返回用户输入的文本。OutputText 属性返回根据 IncludeLiterals 和 IncludePrompt 属性格式化的文本。例如，如果对电话号码进行覆盖，Mask 字符串就应包含括号和几个短横线。这些都是字面量字符，如果 IncludeLiteral 属性设置为 true，括号和短横线就应包含在 OutputText 属性中。

MaskedTextBox 控件还有几个额外的事件。OutputTextChanged 和 InputTextChanged 在 InputText 或 OutputText 改变时触发。

#### 31.3.14 Panel 控件

Panel 控件就是包含其他控件的控件。把控件组合在一起，放在一个面板上，将更容易管理这些控件。例如，可以禁用面板，从而禁用该面板上的所有控件。Panel 控件派生于 ScrollableControl，所以还可以使用 AutoScroll 属性。如果可用区域上有过多的控件要显示，就可以把它们放在一个面板上，并把 AutoScroll 属性设置为 true，这样就可以滚动所有的控件了。

面板在默认情况下不显示边框，但把 BorderStyle 属性设置为不是 none 的其他值，就可以使用面板通过边框可视化地组合相关的控件。这会使用户界面更友好。

Panel 是 FlowLayoutPanel、TableLayoutPanel、TabPage 和 SplitterPanel 的基类。使用这些控件，可以创建非常复杂或专业化的窗体或窗口。FlowLayoutPanel 和 TableLayoutPanel 对创建正确设置大小的窗体尤其有帮助。

#### 31.3.15 FlowLayoutPanel 和 TableLayoutPanel 控件

FlowLayoutPanel 和 TableLayoutPanel 是.NET Framework 的新增控件。顾名思义，面板可以采用 Web 窗体的方式给 Windows 窗体布局。FlowLayoutPanel 是一个容器，允许以垂直或水平的方式放置包含的控件。除了放置控件之外，还可以剪辑控件。放置的方向使用 FlowDirection 属性和 FlowDirection 枚举来设置。WrapContents 属性确定在重新设置窗体的大小时，控件是放在下一行、下一列，还是剪辑控件。

TableLayoutPanel 使用栅格结构控制控件的布局。所有的 Windows 窗体控件都是 TableLayoutPanel 的子控件，包括另一个 TableLayoutPanel。所以窗口的布局可以非常灵活，并可以动态设置。把一个控件添加到 TableLayoutPanel 上时，会给属性页面的 Layout 类别添加 4 个属性 Column、ColumnSpan、Row 和 RowSpan。与 Web 页面上的 html 表一样，可以给每个控件设置列和行间距。控件默认放置在表的单元格中心，但这可以使用 Anchor 和 Dock 属性改变。

行和列的默认样式可以使用 RowStyles 和 ColumnsStyles 集合改变。这两个集合分别包含 RowStyle 和 ColumnsStyle 对象。Style 对象有一个公共属性 SizeType。SizeType 使用 SizeType 枚举来确定列宽

或行高。该枚举值包含 AutoSize、Absolute 和 Percent。AutoSize 与其他同等控件共享该空间。Absolute 允许使用一组像素值来设置大小。Percent 要求控件把列或宽度设置为父控件的一个百分数。

行、列和子控件都可以在运行期间添加和删除。GrowStyle 属性使用 TableLayoutPanelGrowStyle 枚举值来设置在已填满的表中添加一个新控件时，是给表添加列、行，还是使表保持固定的大小。如果其值是 FixedSized，则在试图添加另一个控件时，就抛出一个 ArgumentException 异常。如果表中的单元格为空，控件就放在空单元格中。这个属性仅在表格已满，但要添加控件时起作用。

示例应用程序中的 frmPanel 窗体有 FlowLayoutPanels 和 TableLayoutPanels，并在其中放置了各种控件。试验这些控件，尤其要试用布局面板中控件的 Dock 和 Anchor 属性，这是理解其工作方式的最佳途径。

### 31.3.16 SplitContainer 控件

SplitContainer 控件把 3 个控件组合在一起，其中有两个面板控件，在它们之间有一个分隔栏。用户可以移动分隔栏，重新设置面板的大小。在重新设置面板的大小时，面板上的控件也可以重新设置大小。SplitContainer 的最佳示例是文件管理器。左面板包含文件的树型视图，右面板包含文件夹内容的列表视图。用户在分隔栏上移动鼠标时，光标就会改变，此时可以移动分隔栏。SplitContainer 可以包含任意控件，包括布局面板和其他 SplitContainer。因此，可以创建非常复杂、专业化很高的窗体。

分隔栏的移动和定位可以用 SplitterDistance 和 SplitterIncrement 属性控制。SplitterDistance 属性确定分隔栏与控件左边界或顶边的距离，SplitterIncrement 确定在拖动时分隔栏移动的像素值。面板可以使用 Panel1MinSize 和 Panel2MinSize 属性设置其最小尺寸，这些属性的单位也是像素。

Splitter 控件会引发与移动相关的两个事件 SplitterMoving 和 SplitterMoved。SplitterMoving 事件在移动过程中引发，SplitterMoved 在移动结束后引发。它们都接收一个 SplitterEventArgs。

SplitterEventArgs 的 SplitX 和 SplitY 属性表示 Splitter 左上角的 X 和 Y 坐标，X 和 Y 属性表示鼠标指针的 X 和 Y 坐标的。

### 31.3.17 TabControl 控件和 TabPages 控件

TabControl 允许把相关的组件组合到一系列选项卡页面上。TabControl 管理 TabPages 集合。有几个属性可以控制 TabControl 的外观。Appearance 属性使用 TabAppearance 枚举确定选项卡的外观。其值是 FlatButtons、Buttons 或 Normal。Multiline 属性的值是一个布尔值，确定是否显示多行选项卡。如果 Multiline 属性设置为 false，而有多个选项卡不能一次显示出来，就提供一组箭头，允许用户滚动查看剩余的选项卡。

TabPage 的 text 属性是在选项卡上显示的内容。Text 属性也在重写的构造函数中用作参数。

一旦创建了 TabPage 控件，它基本上就是一个容器控件，用于放置其他控件。Visual Studio 2008 中的设计器使用集合编辑器，很容易给 TabControl 控件添加 TabPage 控件。在添加每个页面时都可以设置各种属性。接着把其他子控件拖放到每个 TabPage 控件上。

通过查看 SelectedTab 属性可以确定当前的选项卡。每次选择新选项卡时，都会引发 SelectedIndex 事件。通过监听 SelectedIndex 属性，再用 SelectedTab 属性确认当前选项卡，就可以对每个选项卡进行特定的处理。

### 31.3.18 ToolStrip 控件

ToolStrip 控件是一个用于创建工具栏、菜单结构和状态栏的容器控件。ToolStrip 直接用于工具栏，还可以用作 MenuStrip 和 StatusStrip 控件的基类。

ToolStrip 控件在用于工具栏时，使用一组基于抽象类 ToolStripItem 的控件。ToolStripItem 可以添加公共显示和布局功能，并管理控件使用的大多数事件。ToolStripItem 派生于 System.ComponentModel.Component 类，而不是 Control 类。基于 ToolStripItem 的类必须包含在基于 ToolStrip 的容器中。

Image 和 Text 是要设置的最常见属性。Image 可以用 Image 属性设置，也可以使用 ImageList 控件，把它设置为 ToolStrip 控件的 ImageList 属性。然后就可以设置各个控件的 ImageIndex 属性。 ToolStripItem 上文本的格式化用 Font、 TextAlign 和 TextDirection 属性来处理。TextAlign 设置文本与控件的对齐方式，它可以是 ControlAlignment 枚举中的任一值，默认为 MiddleRight。TextDirection 属性设置文本的方向，其值可以是 ToolStripTextDirection 枚举中的任一值，包括 Horizontal、Inherit、Vertical270 和 Vertical90。Vertical270 把文本旋转 270°，Vertical90 把文本旋转 90°。

DisplayStyle 属性控制在控件上是显示文本、图像、文本和图像，还是什么都不显示。在 AutoSize 设置为 true 时，ToolStripItem 会重新设置其大小，确保只使用最少量的空间。

直接派生于 ToolStripItem 的控件如表 31-4 所示。

表 31-4

Tool Strip Items	说 明
ToolStripButton	表示用户可以选择的按钮
ToolStripLabel	在 ToolStrip 上显示不能选择的文本或图像。ToolStripLabel 还可以显示一个或多个超链接
ToolStripSeparator	用于分解和组合其他 ToolStripItems。选项根据功能来组合
ToolStripDropDownItem	显示下拉选项。是 ToolStripDropDownButton、 ToolStripMenuItem 和 ToolStripSplitButton 的基类
ToolStripControlHost	在 ToolStrip 上存放其他非 ToolStripItem 的派生控件。是 ToolStripComboBox、 ToolStripProgressBar 和 ToolStripTextBox 的基类

表 31-4 中的前两项 ToolStripDropDownItem 和 ToolStripControlHost 需要详细探讨。

ToolStripDropDownItem 是 ToolStripMenuItems 的基类，用于建立菜单结构。

ToolStripMenuItems 添加到 MenuStrip 控件上。如前所述，MenuStrips 派生于 ToolStrip 控件。在处理和扩展菜单项时，这是很重要的。因为工具栏和菜单派生于同一个类，所以很容易创建管理并执行命令的框架。

ToolStripControlHost 可以包含其他不派生自 ToolStripItem 的控件。可以直接放在 ToolStrip 中的控件是派生自 ToolStripItem 的控件。下面的示例演示了如何在 ToolStrip 中放置 DateTimePicker 控件：

```
public mdiParent()
{
    InitializeComponent();

    ToolStripControlHost _dateTimeCtl;
    _dateTimeCtl = new ToolStripControlHost(new DateTimePicker());
    ((DateTimePicker)_dateTimeCtl.Control).ValueChanged +=
    delegate {
        toolStripLabel1.Text =
        ((DateTimePicker)_dateTimeCtl.Control).Value.Subtract(DateTime.Now).ToString();
    };

    _dateTimeCtl.Width = 200;
    _dateTimeCtl.DisplayStyle = ToolStripItemDisplayStyle.Text;
    toolStrip1.Items.Add(_dateTimeCtl);
}
```

这是示例代码中 frmMain 窗体的构造函数。首先，声明并实例化一个 ToolStripControl Host。注意在实例化该控件时，应把窗体要包含的控件传送给构造函数。下一行代码启动 DateTimePicker 控件的 ValueChanged 事件。这个控件可以通过 ToolStripHostControl 的 Control 属性访问，并返回一个 Control 对象，因此需要将其类型转换为正确的类型。之后，就可以使用窗体包含的控件的属性和方法了。

提供更好封装性能的另一种方法是创建一个派生自 ToolStripControlHost 的新类。下面的代码是 DateTimePicker 工具栏控件的另一个版本 ToolStripDateTimePicker：

```
namespace FormsSample.SampleControls
{
    public class ToolStripDateTimePicker : System.Windows.Forms.ToolStripControlHost
    {
        //need to declare the event that will be exposed
        public event EventHandler ValueChanged;

        public ToolStripDateTimePicker() : base(new DateTimePicker())
        {
        }

        //create strong typed Control property.
        public new DateTime Control
        {
            get { return (DateTime)base.Control; }
        }

        //create a striong typed Value property
        public DateTime Value
```

```
{  
    get { return Control.Value; }  
}  
  
    protected override void  
OnSubscribeControlEvents(Control control)  
{  
    base.OnSubscribeControlEvents(control);  
    ((DateTimePicker)control).ValueChanged +=  
new EventHandler(HandleValueChanged);  
}  
  
    protected override void  
OnUnsubscribeControlEvents(Control control)  
{  
    base.OnUnsubscribeControlEvents(control);  
    ((DateTimePicker)control).ValueChanged -=  
new EventHandler(HandleValueChanged);  
}  
  
    private void HandleValueChanged (object sender,  
EventArgs e)  
{  
    if (ValueChanged != null)  
ValueChanged(this, e);  
}  
}
```

这个类的主要工作是提供 DateTimePicker 选中的属性、方法和事件。这样，主机应用程序就不必维护底层控件的引用了。提供事件的过程有点复杂。OnSubscribeControlEvents 方法用于为基于 ToolStripControlHost 的类(本例是 ToolStripDateTimePicker)同步被包含控件(在这里是 DateTimePicker)的事件。在这个例子中，ValueChanged 事件传送给 ToolStripDateTimePicker。其作用是允许控件的用户在主机应用程序中建立事件，就好像 ToolStripDateTimePicker 派生于 DateTimePicker，而不是 ToolStripControlHost 一样。下面的示例代码演示了这一点，它使用了 ToolStripDateTimePicker：

```
public mdiParent()  
{  
    ToolStripDateTimePicker otherDateTimePicker = new  
    ToolStripDateTimePicker();  
    otherDateTimePicker.Width = 200;  
    otherDateTimePicker.ValueChanged +=  
new EventHandler(otherDateTimePicker_ValueChanged);  
    toolStrip1.Items.Add(otherDateTimePicker);  
}
```

注意，在建立 ValueChanged 事件处理程序时，要引用 ToolStripDateTimePicker 类，而不是像前面的示例那样引用 DateTimePicker 控件。这个示例的代码与第一个例子相比简洁了许多，不仅如此，由于 DateTimePicker 封装在另一个类中，因此封装性能大大提高了，ToolStripDateTimePicker 在应用程序的其他部分或其他项目中的使用也简单了许多。

### 31.3.19 MenuStrip 控件

MenuStrip 控件是应用程序菜单结构的容器。如前所述，MenuStrip 派生于 ToolStrip 类。在建立菜单系统时，要给 ToolStrip 添加 ToolStripMenu 对象。这可以在代码中完成，也可以在 Visual Studio 的设计器中进行。把一个 ToolStrip 控件拖放到设计器的一个窗体中，MenuStrip 就允许直接在菜单项上输入菜单文本。

MenuStrip 控件只有两个额外的属性。GripStyle 使用 ToolStripGripStyle 枚举把栅格设置为可见或隐藏。

MdiWindowListItem 属性提取或返回 ToolStripMenuItem。这个 ToolStripMenuItem 是在 MDI 应用程序中显示所有已打开窗口的菜单。

### 31.3.20 ContextMenuStrip 控件

要显示弹出菜单 或在用户右击鼠标时显示一个菜单 就应使用 ContextMenuStrip 类。与 ToolStrip一样，ContextMenuStrip 也是 ToolStripMenuItems 对象的容器，但它派生于 ToolStripDropDownMenu。ContextMenu 的创建与 ToolStrip 相同，也是添加 ToolStripMenuItems，定义每一项的 Click 事件，执行某个任务。弹出菜单应赋予特定的控件，为此，要设置控件的 ContextMenuStrip 属性。在用户右击该控件时，就显示该菜单。

### 31.3.21 ToolStripMenuItem 控件

ToolStripMenuItem 是建立菜单结构的类。每个 ToolStripMenuItem 对象都表示菜单系统上的一个菜单选项。每个 ToolStripMenuItem 都有一个包含子菜单的 ToolStripItem Collection。这个功能继承自 ToolStripDropDownItem。

由于 ToolStripMenuItem 派生于 ToolStripItem，因此可以使用所有的格式化属性。图像在菜单文本的右边显示为小图标。菜单项的旁边可以有复选框标记，用 Checked 和 CheckState 属性设置该标记。

还可以给每个菜单项指定快捷键。快捷键一般包含两个按键，如 Ctrl+C (Copy 的快捷键)。在指定快捷键时，把 ShowShortCutKey 属性设置为 true，还可以在菜单上显示该快捷键。

在用户单击菜单项或使用定义好的快捷键时，应执行某个任务。为此，最常见的方式是处理 Click 事件。如果使用了 Checked 属性，还可以使用 CheckStateChanged 和 CheckedChanged 事件确定选中状态的变化。

### 31.3.22 ToolStripManager 类

菜单和工具栏结构可以很大，这样就难以管理。ToolStripManager 类可以创建较小、易于管理的菜单或工具栏结构，并在需要时合并它们。例如，如果窗体上有几个不同的控件，每个控件都必须显示一个弹出菜单。所有的控件都要使用几个菜单项，但每个控件还有几个独特的菜单项。可以在一个 ContextMenuStrip 上定义共有菜单选项，独特的菜单项可以预先定义，或在运行期间创建。对于需要弹出菜单的每个控件，应复制共有菜单选项，再使用 ToolStripManager.Merge 方法把独特的菜单选项与公有菜单选项合并起来。最后得到的菜单赋予控件的 ContextMenuStrip 属性。

### 31.3.23 ToolStripContainer 控件

ToolStripContainer 控件用于停放基于 ToolStrip 的控件。添加一个 ToolStripContainer，把 Docked 属性设置为 Fill，就在窗体的两侧边添加了一个 ToolStripPanel，在窗体的中间添加了一个 ToolStripContainerPanel。在 ToolStripPanels 中可以添加任意 ToolStrip (ToolStrip、MenuStrip 或 StatusStrip)。用户可以选择 ToolStrips，把它拖放到窗体的两边或底部。把 ToolStripPanels 的 Visible 属性设置为 false，就不能把 ToolStrip 放在面板上了。窗体中心的 ToolStripContainerPanel 可以用于放置窗体需要的其他控件。

## 31.4 窗体

本章的前面讨论了如何创建简单的 Windows 应用程序。该示例包含一个派生于 System.Windows.Forms.Form 的类。根据.NET Framework 说明文档，“窗体是应用程序中窗口的表示方式。”如果您具有 Visual Basic 背景，就会很熟悉术语“窗体”。如果您是使用 MFC 的 C++ 程序员，就可能习惯把窗体称为窗口、对话框或框架。无论怎样，窗体都是与用户交互的基本方式。我们已经介绍了 Control 类中一些较常见的属性、方法和事件，而且 Form 类派生于 Control 类，所以在 Form 类中存在这些属性、方法和事件。Form 类还在 Control 类的基础上添加了大量的功能，本节就介绍这些功能。

### 31.4.1 Form 类

Windows 客户应用程序可以包含一个窗体或上百个窗体。它们可以是基于 SDI(单文档界面 Single Document Interface)或 MDI(多文档界面，Multiple Document Interface)的应用程序。但无论怎样，System.Windows.Forms.Form 类都是 Windows 客户应用程序的核心。Form 类派生于 ContainerControl，ContainerControl 又派生于 ScrollableControl，ScrollableControl 则派生于 Control 类。因此可以假定，窗体可以是其他控件的容器，当所包含的控件在客户区域中显示不下时可以滚动显示，窗体可以拥有与其他控件相同的属性、方法和事件。所以，Form 类相当复杂。本节就介绍这些功能。

#### 1. 窗体的实例化和释放

理解创建窗体的过程是很重要的。我们要完成的工作取决于编写初始化代码的位置。对于实例，事件以如下顺序发生：

构造函数

Load

Activated

Closing

Closed

Deactivate

前 3 个事件在初始化过程中发生。根据初始化的类型，可以确定要关联哪个事件。这个类的构造函数在对象的实例化过程中执行。Load 事件在对象实例化后，窗体可见之前发生。它与构造函数的区别是窗体的可见性。在引发 Load 事件时，窗体已存在，但不可见。在构造函数的执行过程中，窗体还不存在，处在实例化过程中。Activated 事件在窗体处于可见状态并处于当前状态时发生。

有一种情形会略微改动一下这个事件执行顺序。如果在窗体构造函数执行的过程中，Visible 属性设置为 true，或调用了 Show 方法(它把 Visible 属性设置为 true)，就会立即引发 Load 事件。这也会使窗体可见，并处于当前状态，所以还会引发 Activate 事件。如果在设置 Visible 属性后还有代码，就执行这些代码。所以启动事件的执行顺序如下所示：

构造函数，执行到 Visible = true 为止

Load

Activate

构造函数，执行 Visible = true 之后的代码

这会产生一些未预料到的结果。最好在构造函数中进行尽可能多的初始化。

关闭窗体时会发生什么情况？Closing 事件可以取消处理，它把 CancelEventArgs 作为一个参数，如果把 Cancel 属性设置为 true，就会取消该事件，窗体仍处于打开状态。Closing 事件在窗体关闭时发生，而 Closed 事件在窗体关闭后发生。这两个事件都允许执行必要的清理工作。注意，Deactivate 事件在窗体关闭后发生，这是另一个可能产生难以查找的错误的来源。确保不在 Deactivate 事件中执行防止窗体被正常垃圾收集的操作。例如，设置对另一个对象的引用会使窗体仍未被释放。

如果调用 Application.Exit()方法，且当前有一个或多个窗体处于打开状态，就不会引发 Closing 和 Closed 事件。如果打开了正要清理的文件或数据库连接，这就是一个需要考虑的问题，此时应调用 Dispose 方法，所以另一种更好的方法是把大多数清理代码放在 Dispose 方法中。

与窗体启动相关的一些属性有 StartPosition、ShowInTaskbar 和 TopMost。StartPosition 可以是 FormStartPosition 枚举中的一个值：

CenterParent：窗体位于父窗体的客户区域中心。

CenterScreen：窗体位于当前屏幕的中心。

Manual：窗体的位置根据 Location 属性的值来确定。

WindowsDefaultBounds：窗体位于默认的 Windows 位置，使用默认的大小。

WindowsDefaultLocation：窗体位于默认的 Windows 位置，但其大小根据 Size 属性来定。

ShowInTaskbar 属性确定窗体是否应在任务栏上可见。如果窗体是一个子窗体，且只希望父窗体显示在任务栏上时，才使用这个属性。TopMost 属性指定窗体在应用程序启动时位于最上面，即使窗体没有立即获得焦点，也位于最上面。

为了让用户与应用程序交互，用户必须能看到窗体。利用 Show 和 ShowDialog 方法就可以实现这一点。Show 方法仅使窗体对用户可见。下面的代码演示了如何创建一个窗体，并把它显示给用户。假定要显示的窗体叫做 MyFormClass：

```
MyFormClass myForm = new MyFormClass();
myForm.Show();
```

这是非常简单的。但它的一个缺点是没有给调用代码发送任何通知，说明 MyForm 已处理完，并退出。有时这并不重要，Show 方法工作得很好。如果需要提供某种通知，使用 ShowDialog 方法是一种比较好的选择。

在调用 Show 方法后，Show 方法后面的代码会立即执行。在调用 ShowDialog 方法后，调用代码被暂停执行，等到调用 ShowDialog 方法的窗体关闭后再继续执行。不仅调用代码被暂停执行，而且窗体也可以返回一个 DialogResult 值。DialogResult 枚举是一组标识符，它们描述了对话框关闭的原因，包括 OK、Cancel、Yes、No 和其他几个标识符。为了让窗体返回一个 DialogResult 值，必须设置窗体的 DialogResult 属性，或者在窗体的一个按钮上设置 DialogResult 属性。

例如，假定应用程序的一部分要求提供客户的电话号码。窗体包含一个输入电话号码的文本框，和两个按钮 OK 和 Cancel。如果把 OK 按钮的 DialogResult 属性设置为 DialogResult.OK，把 Cancel 按钮的 DialogResult 属性设置为 DialogResult.Cancel，则在选择其中一个按钮时，窗体就会不可见，并给调用它的窗体返回相应的 DialogResult 值。现在注意窗体没有释放，只是把 Visible 属性设置为 false。这是因为仍必须从窗体中获取值。在这个示例中，我们需要电话号码。在窗体上为电话号码创建一个属性，这样父窗体就可以获取值，并调用窗体上的 Close 方法了。下面就是子窗体的代码：

```
namespace FormsSample.DialogSample
{
    partial class Phone : Form
    {
        public Phone()
        {
            InitializeComponent();

            btnOK.DialogResult = DialogResult.OK;
            btnCancel.DialogResult = DialogResult.Cancel;
        }

        public string PhoneNumber
        {
```

```
get { return textBox1.Text; }
set { textBox1.Text = value; }
}
}
}
```

首先要注意，不包含处理按钮单击事件的代码。因为设置了每个按钮的 DialogResult 属性，所以在单击 OK 或 Cancel 按钮后，窗体就消失了。添加的唯一属性是 PhoneNumber。下面的代码显示了父窗体中调用 Phone 对话框的方法：

```
Phone frm = new Phone();

frm.ShowDialog();
if (frm.DialogResult == DialogResult.OK)
{
    label1.Text = "Phone number is " + frm.PhoneNumber;
}
else if (frm.DialogResult == DialogResult.Cancel)
{
    label1.Text = "Form was canceled.";
}

frm.Close();
```

这看起来非常简单。创建新的 Phone 对象 frm，在调用 frm.ShowDialog()方法时，这个方法中的代码会停止执行，等待 Phone 窗体返回。接着检查 Phone 窗体的 DialogResult 属性。由于窗体还未释放，是不可见的，所以仍可以访问公共属性，其中一个公共属性就是 PhoneNumber。一旦获取了需要的数据，就可以调用窗体的 Close 方法。

一切正常，但如果返回的电话号码格式不正确，该怎么办？如果把 ShowDialog 放在循环中，就可以再次调用它，让用户重新输入值。这样就可以得到正确的值，注意，如果用户单击了 Cancel 按钮，还必须处理 DialogResult.Cancel：

```
Phone frm = new Phone();

while (true)
{
    frm.ShowDialog();
    if (frm.DialogResult == DialogResult.OK)
    {
        label1.Text = "Phone number is " + frm.PhoneNumber;
        if (frm.PhoneNumber.Length == 8 | frm.PhoneNumber.Length
== 12)
        {
            break;
        }
    }
}
```

```
{  
    MessageBox.Show("Phone number was not formatted  
    correctly.  
    Please correct entry.");  
}  
}  
  
else if (frm.DialogResult == DialogResult.Cancel)  
{  
    label1.Text = "Form was canceled."  
    break;  
}  
}  
  
frm.Close();
```

如果电话号码的长度没有通过简单的测试，Phone 窗体就会显示出来，让用户更正错误。

ShowDialog 框没有创建窗体的新实例，在窗体上输入的文本仍在该窗体上，所以如果必须重新设置窗体，就需要程序员自己完成。

## 2. 外观

用户首先看到的是应用程序的窗体。这应是应用程序中第一个也是最重要的功能。如果应用程序不解决业务问题，则其外观就无关紧要了。这并不是说，窗体和应用程序的整体 GUI 设计不应美观。像颜色组合、字体大小和窗口大小等的设计都可以使应用程序更容易让用户操作和接受。

有时不希望用户访问系统菜单。在单击窗口左上角的图标时，这个菜单就会显示出来。一般情况下，它包含恢复、最小化、最大化和关闭等选项。ControlBox 属性允许设置系统菜单的可见性。还可以用 MaximizeBox 和 MinimizeBox 属性设置最大化和最小化按钮的可见性。如果删除了所有的按钮，再把 Text 属性设置为空字符串("")，标题栏就会完全消失。

如果设置了窗体的 Icon 属性，但没有把 ControlBox 属性设置为 false，图标就会显示在窗体的左上角。通常应把 Icon 属性设置为 app.ico，这会使每个窗体的图标都与应用程序的图标相同。

FormBorderStyle 属性用于设置显示在窗体周围的边框类型。它使用 FormBorderStyle 枚举，其值是：

Fixed3D

FixedDialog

FixedSingle

FixedToolWindow

None

Sizable

## SizableToolWindow

大多数值的意义都一目了然，只有两个工具窗口边框除外。无论怎样设置 ShowInTaskBar 属性，工具窗口都不显示在任务栏上。当用户按下 Alt+Tab 时，工具窗口也不会显示在窗口列表中。默认设置是 Sizable。

除非明确要求，否则大多数 GUI 元素的颜色都应设置为系统颜色，而不是特定的颜色。这样，如果一些用户喜欢把所有的按钮设置为紫字绿底，应用程序就会采用这种颜色设置。为了把控件设置为使用特定的系统颜色，必须调用 System.Drawing.Color 类的 FromKnownColor 方法。FromKnownColor 方法将一个 KnownColor 枚举值作为其参数。在该枚举中定义了许多颜色和各种 GUI 元素颜色，例如 Control、ActiveBorder 和 Desktop。例如，如果窗体的背景色应总是匹配 Desktop 颜色，就应使用下面的代码：

```
myForm.BackColor = Color.  
FromKnownColor(KnownColor.Desktop);
```

如果用户改变桌面的颜色，窗体的背景色也会随之改变。这将给应用程序增加友好性。用户可以为桌面选择某种奇怪的颜色组合，但这由他们的偏好决定。

Windows XP 引入了一个叫做可视化样式的特性。当鼠标指针停在按钮、文本框、菜单和其他控件上或单击它们时，这些控件的外观会改变，并做出响应。调用 Application.EnableVisualStyles 方法，可以激活应用程序的可视化样式。这个方法必须在实例化任何类型的 GUI 之前调用，所以，它一般在 Main 方法中调用，如下面的示例所示：

```
[STAThread]  
static void Main()  
{  
    Application.EnableVisualStyles();  
    Application.Run(new Form1());  
}
```

这段代码允许支持可视化样式的各种控件采用可视化样式。由于 EnableVisualStyles 方法存在一个问题，所以必须在调用 EnableVisualStyles 之后立即添加 Application.DoEvents()方法。这应能解决工具栏上的图标在运行期间消失的问题。EnableVisualStyles 方法也可以在.NET Framework 1.1 中使用。

对于控件，还有一个必须完成的任务。大多数控件都包含 FlatStyle 属性，它把 FlatStyle 枚举作为其值。这个属性可以使用如下 4 个值：

Flat：当鼠标指针停在控件上时，控件显示为平面模式。

Flat3D：类似于 Flat，但当鼠标指针停在控件上时，控件显示为 3D 模式。

Standard：控件显示为 3D 模式。

System：控件的外观由操作系统控制。

为了在控件上使用可视化样式，FlatStyle 属性应设置为 FlatStyle.System。应用程序现在采用 XP 的外观和操作方式。

### 31.4.2 多文档界面

当应用程序可以显示同一类型窗体的多个实例，或以某种方式包含不同的窗体时，就应使用 MDI 类型的应用程序。例如可以同时显示多个编辑窗口的文本编辑器和 Microsoft Access，可以在 Access 中同时打开查询窗口、设计窗口和表窗口。这些窗口都不会超出主 Access 应用程序的边界。

包含本章示例的项目就是一个 MDI 应用程序，项目中的窗体 mdiParent 就是 MDI 父窗体。把 IsMdiContainer 设置为 true，会把该窗体设置为 MDI 父窗体。如果在设计器中创建窗体，注意其背景会变成暗灰色，说明这是一个 MDI 父窗体。仍可以给该窗体添加控件，但最好不要这么做。为了使子窗体成为 MDI 子窗体，子窗体需要知道其父窗体是哪个窗体。为此，应把子窗体的 MdiParent 属性设置为父窗体。在这个例子中，所有的子窗体都是用 ShowMdiChild 方法创建的，它的参数是要显示的子窗体的引用。把 MdiParent 属性设置为 this(表示引用 mdiParent 窗体)后，就显示该窗体。下面是 ShowMdiParent 方法的代码：

```
private void ShowMdiChild(Form childForm)
{
    childForm.MdiParent = this;
    childForm.Show();
}
```

MDI 应用程序的一个问题是，在任意时刻可以打开几个子窗体。对当前活动的子窗体的引用可以使用父窗体的 ActiveMdiChild 属性来提取。这就是 Window 菜单中 Current Active 菜单项的作用。单击该菜单项，会显示一个包含窗体名称和文本值的消息框。

子窗体可以调用 LayoutMdi 方法来安排。LayoutMdi 方法把 MdiLayout 枚举值作为参数。其值可以是 Cascade、TileHorizontal 和 TileVertical。

### 31.4.3 定制控件

使用控件和组件是使窗体软件包(如 Windows 窗体)的开发非常高效的一个主要因素。创建自己的控件、组件和用户控件会使开发更加高效。创建控件可以把功能封装在能多次使用的软件包中。

创建控件有许多方式。可以从头开始创建，从 Control、ScrollableControl 或 ContainerControl 中派生自己的类。除了给控件添加需要的功能之外，还必须重写 Paint 事件，完成绘制工作。如果控件是当前控件的一个改进版本，就必须从该控件中派生出要改进的控件。例如，如果需要一个 TextBox 控件，但要改变其背景色，设置其 ReadOnly 属性，则创建一个全新的 TextBox 控件就会浪费时间。从 TextBox 控件中派生，再重写 ReadOnly 属性即可。由于 TextBox 控件的 ReadOnly 属性没有标记为重写，所以必须使用 new 语句。下面的代码展示了新的 ReadOnly 属性：

```
public new bool ReadOnly
{
    get { return base.ReadOnly; }
```

```
set {
    if(value)
        this.BackgroundColor = Color.Red;
    else
        this.BackgroundColor =
Color.FromKnownColor(KnownColor.Window);

    base.ReadOnly = value;
}
}
```

对于属性 get，返回为基本对象设置的内容。把文本框设置为只读的属性在这里并不重要，应把这个功能传递给基本对象。在属性集中，检查传递的值是 true 还是 false。如果是 true，就要将颜色改为只读颜色(在本例中是 Red)；如果是 false，就把 BackGroundColor 设置为默认值。最后，把值传递给基本对象，让文本框成为只读文本框。可以看出，重写一个简单的属性，就可以给控件添加一个新功能。

### 1. 控件属性

可以给定制控件添加属性，改进设计期间控件的功能。表 31-5 描述了一些比较有用的属性。

表 31-5

属性名称	描述
BindableAttribute	在设计期间用于确定属性是否支持双向数据绑定
BrowsableAttribute	确定属性是否显示在可视化设计器中
CategoryAttribute	确定在属性窗口中，属性显示在哪个类别中。使用预定义的类别，或创建新的类别。默认值为 Misc
DefaultEventAttribute	指定类的默认事件

(续表)

属性名称	描述
DefaultPropertyAttribute	指定类的默认属性
DefaultValueAttribute	指定属性的默认值。一般是初始值
DescriptionAttribute	在选中属性时，这是显示在设计器窗口底部的文本
DesignOnlyAttribute	把属性标记为只能在设计模式下编辑

还有其他属性(与在设计期间使用的编辑器相关)以及一些在设计期间使用的高级功能。此外还应添加 Category 和 Description 属性，它们能帮助使用该控件的其他开发人员更好地理解属性的作用。为了添加 Intellisense 支持，应给每个属性、方法和事件都添加 XML 注释。在使用/doc 选项编译控件时，所生成的注释 XML 文件将为控件提供 Intellisense 支持。

### 2. 基于 TreeView 的定制控件

本节将介绍如何根据 TreeView 控件开发定制控件。这个控件显示驱动器中的文件结构。我们还将给它添加属性，设置基本文件夹或根文件夹，确定显示哪些文件和文件夹，并使用上一节讨论的各种属性。

与任何新项目一样，也必须为控件定义需求。下面是必须完成的基本需求列表：

读取文件夹和文件，并显示给用户

在树形层次结构视图中显示文件夹结构

可以隐藏文件

定义哪个文件夹是基本文件夹或根文件夹

返回当前选中的文件夹

提供延迟加载文件结构的功能

这应是一个好的起点。把 TreeView 控件作为新控件的基本控件，就可以满足上述的一个需求。

TreeView 控件以层次结构的形式显示数据。它显示的数据描述了列表中的对象，并可以带有图标。单击对象或使用箭头键，就可以展开和折叠这个列表。

在 Visual Studio 2008 中创建一个新的 Windows 控件库项目，命名为 FolderTree，删除类 UserControl1，添加一个新类，命名为 FolderTree。因为 FolderTree 派生于 TreeView，所以类声明由：

```
public class FolderTree
```

改为：

```
public class FolderTree : System.Windows.Forms.TreeView
```

此时就有了一个功能全面、且能工作的 FolderTree 控件了。该控件可以完成 TreeView 能完成的所有任务。

TreeView 控件包含一个 TreeNode 对象集合。我们不能直接把文件和文件夹加载到控件中，但有两种方式可以把加载的 TreeNode 映射到 TreeView 的 Node 集合和它表示的文件与文件夹中。

例如，在处理每个文件夹时，都会创建一个新的 TreeNode 对象，Text 属性设置为文件或文件夹的名称。如果在某一刻需要文件或文件夹的其他信息，就必须再次进入磁盘收集该信息，或把与文件或文件夹相关的其他数据存储在 Tag 属性中。

另一个方法是创建一个派生自 TreeNode 的新类。可以添加新属性和方法，且仍能使用 TreeNode 的基本功能。本例就使用这个方法，其设计更为灵活。如果需要新属性，就可以添加它们，无需中断已有的代码。

有两类对象必须加载到控件中：文件和文件夹。每类对象都有自己的特性。例如，文件夹有一个 DirectoryInfo 对象，它包含了额外的信息，而文件有一个 FileInfo 对象。由于有这些区别，所以我们使用两个类来加载 TreeView 控件：TreeNode 和 FolderNode。在项目中添加这两个类，每个类都派生于 TreeNode。下面是 TreeNode 的代码：

```
namespace FormsSample.SampleControls
```

```
{  
public class FileNode : System.Windows.FormsTreeNode  
{  
    string _fileName = "";  
    FileInfo _info;  
  
    public FileNode(string fileName)  
    {  
        _fileName = fileName;  
        _info = new FileInfo(_fileName);  
        base.Text = _info.Name;  
        if (_info.Extension.ToLower() == ".exe")  
            this.ForeColor = System.Drawing.Color.Red;  
  
    }  
  
    public string FileName  
    {  
        get { return _fileName; }  
        set { _fileName = value; }  
    }  
  
    public FileInfo FileNodeInfo  
    {  
        get { return _info; }  
    }  
}
```

正在处理的文件名被传送给 FileNode 的构造函数。在构造函数中，为文件创建 FileInfo 对象，并把它设置为成员变量 \_info。base.Text 属性设置为文件名。因为 FileNode 派生于 TreeNode，所以设置 TreeNode 的 Text 属性，这是显示在 TreeView 控件中的文本。

再添加两个属性来获取数据。FileName 返回文件名，FileNodeInfo 返回文件的 FileInfo 对象。

下面是 FolderNode 类的代码。它的结构非常类似于 FileNode，区别是用 DirectoryInfo 属性代替了 FileInfo 属性，用FolderPath 代替了 FileName。

```
namespace FormsSample.SampleControls  
{  
public class FolderNode : System.Windows.FormsTreeNode  
{  
    string _FolderPath = "";  
    DirectoryInfo _info;  
  
    public FolderNode(stringFolderPath)
```

```
{  
    _FolderPath =FolderPath;  
    _info = new DirectoryInfo(FolderPath);  
    this.Text = _info.Name;  
}  
  
    public string FolderPath  
{  
get { return _FolderPath; }  
set { _FolderPath = value; }  
}  
  
    public DirectoryInfo FolderNodeInfo  
{  
get { return _info; }  
}  
  
}
```

现在构建 FolderTree 控件。根据要求，我们需要一个属性来读取和设置 RootFolder，还需要 ShowFiles 属性，来确定文件是否显示在树中。SelectedFolder 属性返回树中当前突出显示的文件夹。

下面是 FolderTree 控件的代码：

```
using System;  
using System.Windows.Forms;  
using System.IO;  
using System.ComponentModel;  
  
namespace FolderTree  
{  
///<summary>  
/// Summary description for FolderTreeCtrl.  
///</summary>  
public class FolderTree : System.Windows.Forms.TreeView  
{  
  
    string _rootFolder ="";  
    bool _showFiles = true;  
    bool _inInit = false;  
  
    public FolderTree()  
    {  
  
    }  
  
    [Category("Behavior"),  
Description("Gets or sets the base or root folder of the tree"),  
DefaultValue("C:\\\\")]
```

```
public string RootFolder
{
    get { return _rootFolder; }
    set
    {
        _rootFolder = value;
        if(!_inInit)
            InitializeTree();
    }
}

[Category("Behavior"),
Description("Indicates whether files will seen in the list."),
DefaultValue(true)]
public bool ShowFiles
{
    get { return _showFiles; }
    set {_showFiles = value; }
}

[Browsable(false)]
public string SelectedFolder
{
    get
    {
        if(this.SelectedNode is FolderNode)
            return (FolderNode)this.SelectedNode.FolderPath;

        return "";
    }
}
```

我们添加了 3 个属性 ShowFiles、SelectedFolder 和 RootFolder。注意已经添加的属性。为 ShowFiles 和 RootFolder 设置 Category、Description 和 DefaultValue。这两个属性显示在设计模式下的属性浏览器中。SelectedFolder 在设计期间没有什么意义，所以选择 Browsable=false 属性。SelectedFolder 不在属性浏览器中显示，但由于它是一个公共属性，所以会在 Intellisence 中出现，并可以通过代码访问它。

接着，初始化文件系统的加载操作。初始化控件有点困难。在设计期间和运行期间进行初始化都必须仔细考虑。控件位于设计器中时，实际上是在运行。如果在构造函数中调用了一个数据库，这个调用就是在设计器中拖放控件时进行的。在 FolderTree 控件中，这就存在一个问题。

下面看看加载文件的方法：

```
private void LoadTree(FolderNode folder)
{
```

```
string[] dirs =
Directory.GetDirectories(folder.FolderPath);
foreach(string dir in dirs)
{
    folderNode tmpfolder = new FolderNode(dir);
    folder.Nodes.Add(tmpfolder);
    LoadTree(tmpfolder);
}
if (_showFiles)
{

    string[] files =
Directory.GetFiles(folder.FolderPath);
foreach(string file in files)
{
    FileNode fnode = new FileNode(file);
    folder.Nodes.Add(fnode);
}
}
}
```

showFiles 是一个布尔成员变量，在 ShowFiles 属性中设置。如果设置为 true，就在树中显示文件。现在唯一的问题是何时调用 LoadTree。我们有几种选择。可以在设置 RootFolder 属性时调用，这在一些情况下是很理想的，但在设计期间并不合适。控件在设计器中是激活的，所以在设置 rootNode 属性时，控件就会试图加载文件系统。

要解决这个问题，应查看 DesignMode 属性，如果控件在设计器中，它就返回 true。现在可以编写代码，初始化控件了：

```
private void InitializeTree()
{
    if (!this.DesignMode && _rootFolder != "")
    {
        FolderNode rootNode = new FolderNode(_rootFolder);
        LoadTree(rootNode);
        this.Nodes.Clear();
        this.Nodes.Add(rootNode);
    }
}
```

如果控件不处于设计模式，且\_rootFolder 也不是空字符串，就开始加载树。先创建 Root 节点，再把它传送给 LoadTree 方法。

另一个选择是执行公共方法 Init。在 Init 方法中，调用 LoadTree 方法。这个选择存在的问题是使用控件的开发人员需要调用 Init。根据情况的不同，这或许是一个可以接受的解决方法。

对于添加的灵活性，可以实现 ISupportInitialize 接口。ISupportInitialize 有两个方法 BeginInit 和 EndInit。在控件实现 ISupportInitialize 接口时，就会在 InitializeComponent 生成的代码中自动调用 BeginInit 和 EndInit 方法。这样就可以延缓初始化过程，直到设置完所有的属性为止。ISupportInitialize 还允许父窗体中的代码延缓初始化过程。如果在代码中设置 RootNode，先调用 BeginInit 就可以在控件加载文件系统之前设置 RootNode 属性和其他属性，或执行动作。在调用 EndInit 时，初始化控件。BeginInit 和 EndInit 方法如下所示：

```
#region ISupportInitialize Members

void ISupportInitialize.BeginInit()
{
    _inInit = true;
}

void ISupportInitialize.EndInit()
{
    if (_rootFolder != "")
    {
        InitializeTree();
    }

    _inInit = false;
}

#endregion
```

在 BeginInit 方法中，只是把成员变量 \_inInit 设置为 true。这个标志用于确定控件是否处于初始化过程，并在 RootFolder 属性中使用。如果在 InitializeComponent 类的外部设置 RootFolder 属性，树就需要重新初始化。在 RootFolder 属性中，检查 \_inInit 是 true 还是 false。如果是 true，就不需要查看初始化过程。如果 \_inInit 是 false，就调用 InitializeTree。还可以用一个公共方法 Init 完成这个任务。

在 EndInit 方法中，检查控件是否处于设计模式，\_rootFolder 是否被赋予了有效的路径。然后仅调用 InitializeTree。

为了添加专业化的外观，需要添加一个位图图像。当把控件添加到项目中时，这个图标将显示在工具箱中。位图图像应是 16×16 像素，16 色。可以用任意图形编辑器创建这个图像文件，只要正确设置了大小和颜色深度即可。甚至可以在 Visual Studio 2008 中创建这个文件：右击项目，选择 Add New Item。在列表中选择 Bitmap File，打开图像编辑器。在创建好位图文件之后，把它添加到项目中，确保它位于控件的命名空间中，且与控件同名。最后，把位图的 Build Action 设置为 Embedded Resource：在 Solution Explorer 中右击位图文件，选择 Properties。然后从 Build Action 属性中选择 Embedded Resource。

要测试控件，在同一个解决方案中创建 TestHarness 项目。TestHarness 是一个简单的 Windows 窗体应用程序，只包含一个窗体。在引用部分添加对 FolderTreeCtl 项目的引用。在 Toolbox 窗口中添加

对 FolderTreeCtl.DLL 的引用。FolderTreeCtl 现在应显示在工具箱中，且位图已添加为图标。单击该图标，把它拖放到 TestHarness 窗体上。把 RootFolder 设置为一个可用的文件夹，并运行解决方案。

这还不是一个完整的控件。还必须做一些工作，改进控件，使之成为功能全面、可用于产品的控件。例如，可以添加：

异常：如果控件试图加载用户不能访问的文件夹，就会引发一个异常。

后台加载：加载大的文件夹树是很费时间的。应改进初始化过程，以利用后台线程来加载大文件夹树。

颜色代码：可以让某些类型的文本显示为不同的颜色。

图标：可以添加一个 ImageList 控件，在加载时为每个文件或文件夹添加一个图标。

### 3. 用户控件

用户控件是 Windows 窗体中功能比较强大的一个特性。它们允许把用户界面设计封装到可重用的软件包中，以便以后把软件包插入其他项目。公司有若干个常用用户控件库是很常见的。不仅用户界面功能可以包含在用户控件中，而且常见的数据验证也可以包含在用户控件中，例如格式化电话号码或 id 号。在用户控件中可以包含一个预定义的项目列表，以便快速加载列表框或组合框。状态代码或国家代码也可以划在这个类别中。在用户控件中尽可能组合许多与当前应用程序不相关的功能，会使控件在公司中的用途更大。

本节将创建一个简单的地址用户控件，并添加两个事件，使控件可用于数据绑定。地址控件有几个文本字段用于输入两个地址行：城市、州和邮政编码。

要在当前项目中创建用户控件，只需在 Solution Explorer 中右击项目，选择 Add，再选择 Add New User Control。还可以创建一个新的 Control Library 项目，并添加用户控件。在启动新的用户控件后，就会在设计器上看到一个没有边框的窗体。在这里可以拖放组成用户控件的控件。用户控件实际上是把一个或多个控件添加到一个容器控件中，这有点类似于创建窗体。对于地址控件，需要 5 个文本框控件和 3 个标签控件。这些控件的布局可以任意，只要看着舒服即可，如图 31-4 所示。

这个示例中的 TextBox 控件命名如下：

txtAddress1

txtAddress2

txtCity

txtState

txtZip

在安排好了文本框控件，并指定了有效的名称后，就可以添加公共属性。我们希望把 TextBox 控件的可访问性设置为 public，而不是 private，但这并不好，因为这违背了封装要添加到属性中的功能的初衷。下面是必须添加的属性代码：

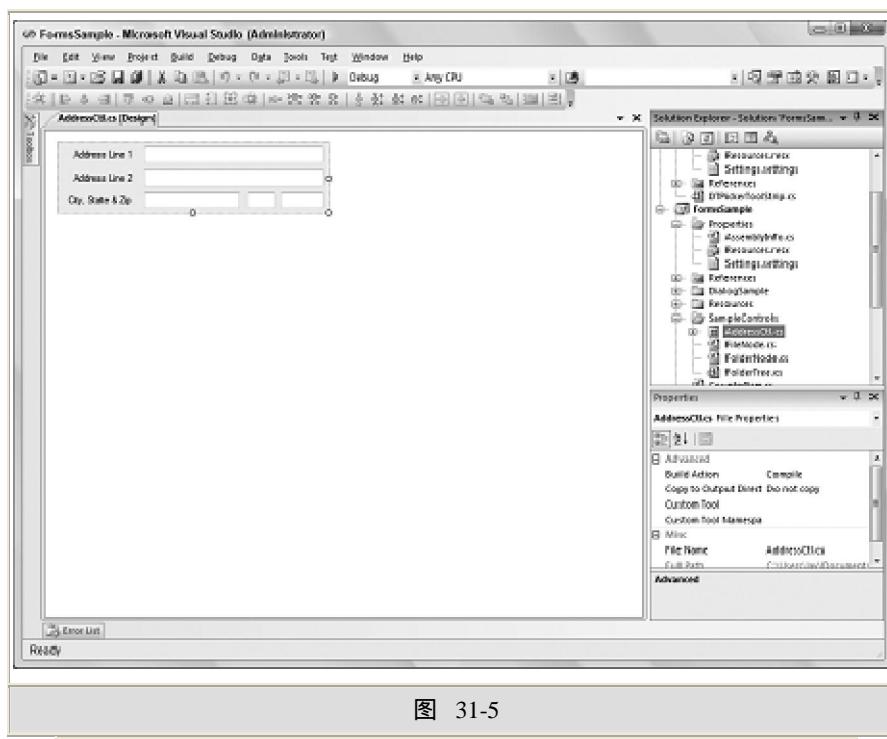


图 31-5

```
public string AddressLine1
{
    get{return txtAddress1.Text;}
    set{
        if(txtAddress1.Text != value)
        {
            txtAddress1.Text = value;
            if(AddressLine1Changed != null)
                AddressLine1Changed(this,
                    PropertyChangedEventArgs.Empty);
        }
    }
}

public string AddressLine2
{
    get{return txtAddress2.Text;}
    set{
        if(txtAddress2.Text != value)
        {
            txtAddress2.Text = value;
            if(AddressLine2Changed != null)
                AddressLine2Changed(this,
                    PropertyChangedEventArgs.Empty);
        }
    }
}

public string City
{
```

```
get{return txtCity.Text;}  
set{  
    if(txtCity.Text != value)  
    {  
        txtCity.Text = value;  
        if(CityChanged != null)  
            CityChanged(this, PropertyChangedEventArgs.Empty);  
    }  
}  
}  
  
public string State  
{  
get{return txtState.Text;}  
set{  
    if(txtState.Text != value)  
    {  
        txtState.Text = value;  
        if(StateChanged != null)  
            StateChanged(this, PropertyChangedEventArgs.Empty);  
    }  
}  
}  
  
public string Zip  
{  
get{return txtZip.Text;}  
set{  
    if(txtZip.Text != value)  
    {  
        txtZip.Text = value;  
        if(ZipChanged != null)  
            ZipChanged(this, PropertyChangedEventArgs.Empty);  
    }  
}  
}
```

属性的 get 部分非常简单，它返回相应 TextBox 控件的 Text 属性值。属性的 set 部分做的工作比较多。所有的 set 部分都以相同的方式工作。先检查属性值是否改变。如果新值与当前值相同，就快速退出。如果不相同，就把文本框的 Text 属性设置为新值，并测试事件是否实例化。要查找的事件是属性的修改事件。它采用特殊的命名格式 propertynameChanged，其中 propertyname 是属性的名称。在 AddressLine1 属性中，这个事件称为 AddressLine1Changed。该属性的声明如下：

```
public event EventHandler AddressLine1Changed;  
public event EventHandler AddressLine2Changed;  
public event EventHandler CityChanged;
```

```
public event EventHandler StateChanged;
public event EventHandler ZipChanged;
```

事件的作用是通知绑定操作，属性已改变。一旦进行验证，绑定操作就会确保把新值返回给控件所绑定的对象。要支持绑定，还需要做另一个工作。用户对文本框的修改将不再直接设置属性。所以在文本框改变时也必须引发 propertyNameChanged 事件。最简单的方式是监视 TextBox 控件的 TextChanged 事件。这个示例只包含一个 TextChanged 事件处理程序，且所有的文本框都使用它。检查控件名，确定是哪个控件引发了事件，并引发相应的 propertyNameChanged 事件。下面是事件处理程序的代码：

```
private void TextBoxControls_TextChanged(
object sender, System.EventArgs e)
{
switch(((textBox)sender).Name)
{
case "txtAddress1":
if(AddressLine1Changed != null)
AddressLine1Changed(this, EventArgs.Empty);

break;

case "txtAddress2":
if(AddressLine2Changed != null)
AddressLine2Changed(this, EventArgs.Empty);

break;

case "txtCity":
if(CityChanged != null)
CityChanged(this, EventArgs.Empty);

break;

case "txtState":
if(StateChanged != null)
StateChanged(this, EventArgs.Empty);

break;

case "txtZip":
if(ZipChanged != null)
ZipChanged(this, EventArgs.Empty);

break;
}
}
```

这个示例使用简单的 switch 语句来确定是哪个文本框引发了 TextChanged 事件。接着进行检查，验证事件是有效的，且不等于空。然后引发 TextChanged 事件。注意发送了一个空 EventArgs (EventArgs.Empty)。这些事件都已添加到属性中，以支持数据绑定，但这并不意味着使用该控件必须进行数据绑定。这些属性可以在代码中设置和读取，不必使用数据绑定。它们都被添加进来，所以用户控件可以在绑定可用时使用绑定。这只是使用户控件尽可能灵活的一种方式，以用于尽可能多的场合。

用户控件实际上是带有某些额外特性的控件，前一节讨论的所有在设计期间可能存在的问题在用户控件中也会出现。初始化用户控件可能会引发 FolderTree 示例中探讨的问题。所以在设计用户控件时必须小心，应避免出现使用该控件的其他开发人员不能访问数据存储等问题。

与控件的创建类似，属性也可以应用于用户控件。当把用户控件放在设计器中时，它的公共属性和方法会显示在属性窗口中。在地址用户控件的示例中，最好给地址属性添加 Category、Description 和 DefaultValue 特性。可以创建一个新的 AddressData 类别，并把默认值都设置为""。下面是这些特性应用于 AddressLine1 属性的示例：

```
[Category("AddressData")]
[Description("Gets or sets the AddressLine1 value"),
DefaultValue(" ")]
public string AddressLine1
{
    get{return txtAddress1.Text;}
    set{
        if(txtAddress1.Text != value)
        {
            txtAddress1.Text = value;
            if(AddressLine1Changed != null)
                AddressLine1Changed(this, EventArgs.Empty);
        }
    }
}
```

可以看出，要添加新的类别，只需在 Category 特性中设置文本，新类别就会自动添加。

仍有许多可以改进的地方。例如，可以在控件中包含一个州名和缩写列表。除了 state 属性之外，用户控件还可以显示州名和州的缩写。我们还应添加异常处理，并可以添加地址行的验证功能。确保大小写正确，弄清楚 AddressLine1 是否可选，在 AddressLine2 中是否应输入单元和房间号，而不是在 AddressLine1 中输入。

### 31.5 小结

本章介绍了建立基于 Windows 的客户应用程序的基础知识。解释了每个基本控件，讨论了 Windows.Forms 命名空间的层次结构，论述了控件的各种属性和方法。

我们还阐述了如何创建基本定制控件和基本用户控件。创建自己的控件是非常灵活的，无论如何强调都不过分。通过创建自己的定制控件工具箱，基于 Windows 的客户应用程序将更容易开发和测试，因为可以重用测试过的同一组件。

下一章介绍如何把数据源链接到窗体的控件上，这样可以使创建出来的窗体自动更新数据，使之与窗体上的数据同步。

## 第 32 章数 据 绑 定

第 26 章介绍了选择和修改数据的各种方式，本章接着第 25 章的内容，继续介绍如何把绑定到各种 Windows 控件上的数据显示给用户。本章主要内容如下：

使用 DataGridView 控件显示数据

.NET 数据绑定功能及其工作方式

如何使用 Server Explorer 创建连接，生成 DataSet 类(不需要编写代码)

如何对 DataGrid 中的数据行进行测试和反射

本章的示例代码可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

### 32.1 DataGridView 控件

.NET 的最初版本中的 DataGrid 控件有强大的功能，但在许多方面，它都不适用于商业应用程序，例如不能显示图像、下拉控件或锁定列等。该控件给人感觉只完成了一半，所以许多控件经销商都提供了定制的栅格控件，以克服这些缺陷，并提供更多的功能。

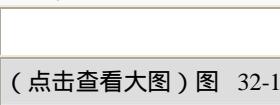
在.NET 2.0 中有了另一个栅格控件 DataGridView。它解决了 DataGrid 控件最初的许多问题，还增加了许多只能在插件产品中使用的功能。

新控件具有与 DataGrid 类似的绑定功能，可以绑定到 Array、DataTable、 DataView 或 DataSet 类，或者绑定到实现 IListSource 或 IList 接口的组件上。DataGridView 控件可以显示数据的许多视图。在最简单的情况下，设置 DataSource 和DataMember 属性，就可以显示数据(与 DataSet 类一样)。注意，这个新控件不是 DataGrid 的插件替代品，所以其编程接口完全不同于 DataGrid。这个控件还提供了更复杂的功能，本章将讨论这些功能。

### 32.1.1 显示列表数据

第 19 章介绍了选择数据和把数据放在一个数据表中的各种方式，但仅使用了 Console.WriteLine() 方法以非常基本的形式显示数据。

下面的示例将说明如何获取一些数据，并在 DataGridView 控件中显示，为此，建立一个新的应用程序 DisplayTabularData，如图 32-1 所示。



这个简单的应用程序从 Northwind 数据库的 customer 表中选择每个记录，在 DataGridView 控件中把它们显示给用户。其代码如下所示(不包含窗体和控件定义代码)：

```
using System;
using System.Configuration;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Windows.Forms;

namespace DisplayTabularData
{
    partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void getData_Click(object sender, EventArgs e)
        {
            string customers = "SELECT * FROM Customers";

            using (SqlConnection con =
new SqlConnection(ConfigurationSettings.
ConnectionStrings["northwind"].ConnectionString))
            {
                DataSet ds = new DataSet();
```

```
SqlDataAdapter da = new SqlDataAdapter(customers, con);
da.Fill(ds, "Customers");
dataGridView.AutoGenerateColumns = true;
dataGridView.DataSource = ds;
dataGridView.DataMember = "Customers";
}
}
}
}
```

窗体包含 `getData` 按钮，单击它会调用示例代码中的 `getData_Click` 方法。

这段代码使用  `ConfigurationManager` 类的属性 `.ConnectionStrings` 构建了 `SqlConnection` 对象。之后构建一个 `DataSet`，使用 `DataAdapter` 对象填充数据库表中的数据。然后设置 `DataSource` 和 `DataMember` 属性，在 `DataGridView` 控件中显示数据。注意 `AutoGenerateColumns` 属性也设置为 `true`，以确保给用户显示一些数据。如果这个标记没有指定，就需要自己创建所有的列。

### 32.1.2 数据源

`DataGridView` 控件是一种显示数据的非常灵活的方式。除了把 `DataSource` 设置为 `DataSet`，`DataMember` 设置为要显示的表名之外，`DataSource` 属性还可以设置为下述任何一个数据源：

数组(网格可以绑定到任何一个一维数组上)

`DataTable`

`DataView`

`DataSet 或 DataViewManager`

实现 `IListSource` 接口的组件

实现 `IList` 接口的组件

泛型集合类或派生于泛型集合类的对象

下面几节将给出这些数据源的示例。

#### 1. 显示数组中的数据

这初看起来非常简单，创建一个数组，填充一些数据，再在 `DataGridView` 控件上设置 `DataSource` 属性。下面是一些示例代码：

```
string[] stuff = new string[] {"One", "Two", "Three"};
dataGridView. DataBinding = stuff;
```

如果数据源包含多个表(例如使用 `DataSet` 或 `DataViewManager`)，就需要设置 `DataMember` 属性。

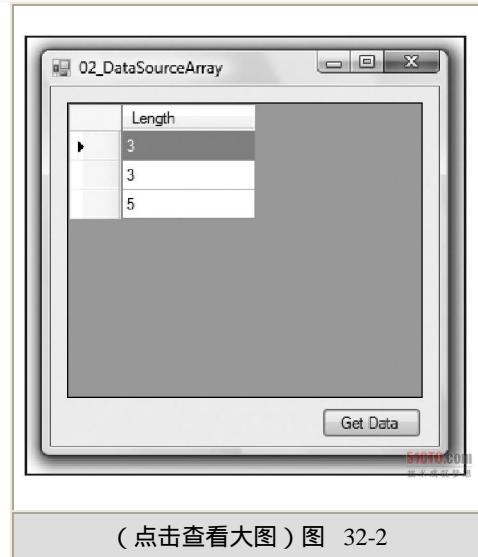
可以用这个数组代码替换上面示例中的 getData\_Click 事件处理程序，这段代码的结果如图 32-2 所示。

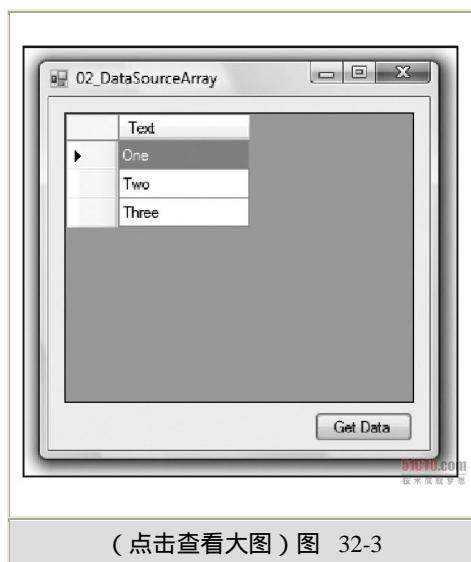
可以看出，网格显示出了数组中定义的字符串的长度，而不是这些字符串。原因是在把数组用作 DataGridView 控件的数据源时，网格会查找数组中对象的第一个公共属性，并显示这个值，而不会显示字符串值。字符串的第一个(也是唯一的)公共属性是其长度，所以就显示这个长度值。使用 TypeDescriptor 类的 GetProperties 方法可以获得任意类的属性列表，该方法返回的是一个 PropertyDescriptor 对象集合，接着，就可以在显示数据时使用它。.NET 的 PropertyGrid 控件在显示任意对象时，就使用这个方法。

在 DataGridView 中显示字符串的一种解决方法是创建一个包装器类，如下所示：

```
protected class Item
{
    public Item(string text)
    {
        _text = text;
    }
    public string Text
    {
        get { return _text; }
    }
    private string _text;
}
```

在数据源数组代码中添加这个 Item 类(从进行的各种处理来讲，它也可以是一个结构)的数组后，结果如图 32-3 所示。





(点击查看大图) 图 32-3

## 2. DataTable

有两种方式在 DataGridView 控件中显示 DataTable：

如果 DataTable 是独立的，就把控件的 DataSource 属性设置为这个表。

如果在 DataSet 中包含 DataTable，就把控件的 DataSource 属性设置为 DataSet，DataMember 属性设置为 DataSet 中的 DataTable 名。

图 32-4 所示为运行 DataSourceDataTable 示例代码的结果。

(点击查看大图) 图 32-4

注意，最后一列显示了一个复选框，而不是更常见的编辑控件。DataGridView 控件没有显示其他信息，而是从数据源中读取模式(在本例中是 Products 表)，根据列的类型推断应显示什么控件。与以前的 DataGrid 控件不同，DataGridView 控件还可以显示图像列、按钮和组合框。

在修改了 DataGrid 中的字段时，数据库中的数据不会改变，因为此时数据仅存储在本地计算机上-- 没有与数据库的活动连接。后面将讨论更新数据源中的数据。

## 3. 显示 DataView 中的数据

DataView 提供了一种过滤和排序 DataTable 中数据的一种方式。在从数据库中选择数据时，用户一般可以单击列标题，对数据排序。此外，还可以只过滤要显示在某些行中的数据，例如用户修改过的所有数据。DataView 允许过滤要显示给用户的 data 行，但不允许过滤 DataTable 中的数据列。

提示：

DataView 不允许过滤要显示的数据列，只允许过滤要显示的数据行。

根据现有的 DataTable 创建 DataView 的代码如下所示：

```
DataTable dv = new DataView(dataTable);
```

创建好后，就可以改变 DataView 上的设置，当该视图显示在 DataGrid 中时，这些设置会影响要显示的数据，以及对这些数据进行的操作。例如：

设置 AllowEdit = false 表示在数据行上禁用所有列的编辑功能。

设置 AllowNew = false 表示禁用新行功能。

设置 AllowDelete = false 表示禁用删除行的功能。

设置 RowStateFilter 只显示指定状态的行。

设置 RowFilter 可过滤数据行。

下一节将介绍如何使用 RowStateFilter 设置，其他选项都很容易理解。

#### (1) 通过数据过滤数据行

创建好 DataView 后，就可以通过设置 RowFilter 属性，来改变视图显示的数据。这个属性是一个字符串，可用作按照给定条件过滤数据的一种方式——该字符串的值就是过滤条件。其语法类似于一般 SQL 中的 WHERE 子句，但主要是对已经从数据库中选择出来的数据进行操作。

过滤子句的一些示例如表 32-1 所示。

表 32-1

子句	说明
UnitsInStock > 50	只显示 UnitsInStock 列大于 50 的行
Client = 'Smith'	只返回给定客户的记录
County LIKE 'C*'	返回 County 字段以 C 开头的所有记录——例如返回 Cornwall、Cumbria、Cheshire 和 Cambridgeshire 所有的行。可以使用 % 表示匹配一个字符的通配符，而 * 表示匹配 0 个或多个字符的通配符

运行库尽可能在过滤表达式中使用与源列相匹配的数据类型。例如，在前面的示例中，使用 UnitsInStock > '50' 表达式是合法的，尽管该列是一个整数列。但如果提供了一个无效的过滤字符串，就会产生 EvaluateException 异常。

#### (2) 根据状态过滤数据行

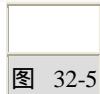
DataView 中的每一行都有一个定义好的行状态，它们的值如表 32-2 所示，这些状态也可以用于过滤用户查看的行。

表 32-2

DataViewRowState	说明
Added	列出新创建的所有行
CurrentRows	列出除了被删除的行以外的其他行
Deleted	列出最初被选中，且已经删除的行——不显示已经删除的新建行
ModifiedCurrent	列出所有已被修改的行，并显示这些行的当前值

ModifiedOriginal	列出所有已被修改的行，但显示这些行的初值，而不是当前值
OriginalRows	列出最初从数据源中选中的所有行，不包括新行。显示列的初值(即如果进行了修改，不显示当前值)
Unchanged	列出没有修改的行

图 32-5 显示了两个网格，一个网格显示已添加、删除或修改的数据行，另一个网格显示状态为表 32-2 中一种的行。



过滤器不仅可以用于可见的行，还可以用于这些行中列的状态。在进行 Modified- Original 或 ModifiedCurrent 选择时，这是很明显的。这两个状态都在第 20 章介绍过了，它们都是基于 DataRowVersion 枚举的。例如，如果用户更新了数据行中的一列，该行就会在选择 ModifiedOriginal 或 ModifiedCurrent 时显示出来，但其实际值可以是从数据库中选择出来的初值(如果选择了 ModifiedOriginal)，或者 DataColumn 中的当前值(如果选择了 ModifiedCurrent)。

### (3) 对数据行进行排序

除了过滤数据外，有时还需要对 DataView 中的数据进行排序。可以在 DataGridView 控件中单击列标题，这会按照升序或降序的顺序对该列进行排序，如图 32-6 所示。唯一的问题是控件只能对一列进行排序，而底层的 DataView 控件可以对多个列进行排序。

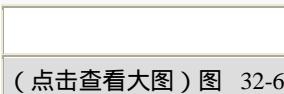
在对数据列进行排序时，可以单击列的标题(例如上面的 ProductName 列)，也可以通过代码排序，DataGridView 会显示一个箭头位图，表示对哪一列进行排序。

要编程设置列的排列顺序，可以使用 DataView 的 Sort 属性：

```
dataView.Sort = "ProductName";  
dataView.Sort = "ProductName ASC, ProductID DESC";
```

上面的第一行按照 ProductName 列对数据排序，如图 32-6 所示。第二行按照 ProductName 列对数据进行升序排序，再以 ProductID 的降序来排序。

DataView 支持对列进行升序或降序排序-- 默认为升序。如果选择对 DataView 中的多个列进行排序，DataGridView 就不会显示任何排序箭头。



网格中的每一列都是强类型化的，其排序顺序不是基于列的字符串表示，而是基于该列的数据。如果 DataGridView 有一个日期列，要对它进行排序，网格就会按日期来进行排序，而不是按日期字符串来进行排序。

## 4. 显示 DataSet 类中的数据

DataSet 有一个 DataGridView 不匹配 DataGrid 的特性：DataSet 定义时包含了表之间的关系。和以前的 DataGridView 示例一样，DataGrid 一次只能显示一个 DataTable，但在下面的示例 DataSourceDataSet 中，可以在屏幕上浏览 DataSet 中的关系。下面的代码可以根据 Northwind 数据库中的 Customers 和 Orders 表生成这样一个 DataSet。这个示例从两个 DataTable 中加载数据，然后在这些表之间创建了一个关系 CustomerOrders：

```
string orders = "SELECT * FROM Orders";
string customers = "SELECT * FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(orders, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Orders");
da = new SqlDataAdapter(customers, conn);
da.Fill(ds, "Customers");
ds.Relations.Add("CustomerOrders",
    ds.Tables["Customers"].Columns["CustomerID"],
    ds.Tables["Orders"].Columns["CustomerID"]);
```

创建好后，通过调用 SetDataBinding，就可以把 DataSet 绑定到 DataGrid 上：

```
dataGrid1.SetDataBinding(ds, "Customers");
```

这会得到如图 32-7 所示的屏幕图。

(点击查看大图) 图 32-7

与本章前面的 DataGridView 示例不同，每个记录的左边都有一个+号，这表示 DataSet 在 customers 和 orders 表之间有一个可导航的关系。在代码中可以定义许多这类关系。

单击+号，就会显示关系列表（如果关系已经显示出来，单击+号就会隐藏该关系）。单击关系名，就可以定位到链接的记录上，如图 32-8 所示，在本例中是列出选中客户的所有订单。

(点击查看大图) 图 32-8

DataGrid 控件的右上角还包含两个新图标。箭头允许用户导航回父行，显示上一页的内容。标题行显示父记录的细节，单击另一个按钮会隐藏或显示该箭头。

### 在 DataViewManager 中显示数据

DataViewManager 中显示的数据与 DataSet 中显示的数据相同，但在为 DataSet 创建 DataViewManager 时，会为每个 DataTable 创建一个单独的 DataView，再执行代码，根据过滤条件或者行的状态改变显示出来的行，如上面的 DataView 示例所示。即使代码不需要过滤数据，也可以把 DataSet 包装到 DataViewManager 中以进行显示，因为这样在修改源代码时可以使用更多的选项。

下面的示例根据上一例中的 DataSet 创建一个 DataViewManager , 然后改变 Customer 表中的 DataView , 使之只显示来自英国的客户 :

```
DataViewManager dvm = new DataViewManager(ds);
dvm.DataViewSettings["Customers"].RowFilter =
"Country='UK'";
dataGridView.SetDataBinding(dvm, "Customers");
```

如图 32-9 所示为 DataSourceDataViewManager 示例代码的运行结果。

(点击查看大图) 图 32-9

## 5. IListSource 和 IList 接口

DataGridView 还支持执行 IListSource 和 IList 接口之一的所有对象。 IListSource 只有一个方法 GetList() , 它返回一个 IList 接口 , 而 IList 比较有趣 , 它可由运行库中的许多类执行 , 执行这个接口的类有 Array、 ArrayList 和 StringCollection。

在使用 IList 时 , 对前面 Array 的警告也适用于集合中的对象-- 如果使用 StringCollection 作为 DataGridView 的数据源 , 网格就会显示字符串的长度 , 而不是我们希望显示的元素文本。

## 6. 显示泛型集合

除了已介绍的类型之外 , DataGridView 还可以绑定到泛型集合上。其语法与本章前面的示例相同 , 也是把 DataSource 属性设置为该集合 , 控件就会生成相应的显示结果。

所显示的列也基于对象的属性 : 在 DataGridView 中显示所有公共的可读字段。下面的示例显示了一个定义好的 list 类 :

```
class PersonList : List < Person >
{
}

class Person
{
    public Person( string name, Sex sex, DateTime dob )
    {
        _name = name;
        _sex = sex;
        _dateOfBirth = dob;
    }

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

```
    public Sex Sex
    {
        get { return _sex; }
        set { _sex = value; }
    }

    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }

    private string _name;
    private Sex _sex;
    private DateTime _dateOfBirth;
}

enum Sex
{
    Male,
    Female
}
```

这段代码显示了 Person 类的几个实例，它们是在 PersonList 类中构建的，如图 32-10 所示。

(点击查看大图) 图 32-10

在一些情况下，需要在网格中隐藏某些属性，此时可以使用 Browsable 特性，如上面的代码所示。标记为 non-browsable 的属性不会显示在属性网格中。

```
[Browsable(false)]
public bool IsEmployed
{
    ...
}
```

DataGridView 使用 Browsable 特性确定是显示还是隐藏一个属性。如果没有设置 Browsable 特性，就默认为显示属性。如果属性是只读的，网格控件就显示对象的值，但在网格中它是只读的。

在网格视图中进行的所有修改都会反映到底层对象上。例如，如果在上面的代码中，修改了用户界面上的人名，就会调用该属性的设置器方法。

## 32.2 DataGridView 类的层次结构

DataGridView 主要部分的类层次结构如图 32-11 所示。

(点击查看大图) 图 32-11

DataGridView 控件在显示数据时利用了派生自 DataGridViewColumn 的对象，如图 32-11 所示。现在，显示数据的选项比以前的 DataGrid 控件多。一个大的变化是在 DataGrid 中显示下拉列的这个功能，现在 DataGridView 以 DataGridViewComboBoxColumn 的形式提供。

在为 DataGridView 指定数据源时，默认要自动构建列。这些列是根据数据源中的数据类型创建的，所以布尔字段映射为 DataGridViewCheckBoxColumn。如果要自己完成列的创建，就可以把 AutoGenerateColumns 属性设置为 false，自己构建列。

下面的例子演示了如何构建列，并包含一个图像和一列 ComboBox。代码利用了一个 DataSet，把数据提取到两个 DataTable 中。第一个 DataTable 包含 Northwind 数据库中的雇员信息，第二个表包含 EmployeeID 列和自动生成的 Name 列，在显示 ComboBox 时要使用这两列：

```
using (SqlConnection con =
new SqlConnection (
ConfigurationSettings.ConnectionStrings["northwind"].ConnectionString ) )
{
    string select = "SELECT EmployeeID, FirstName, LastName, Photo,
IsNull(ReportsTo,0) as ReportsTo FROM Employees";

    SqlDataAdapter da = new SqlDataAdapter(select, con);

    DataSet ds = new DataSet();

    da.Fill(ds, "Employees");

    select = "SELECT EmployeeID, FirstName + ' ' + LastName as Name
FROM Employees UNION SELECT 0,'(None)'";

    da = new SqlDataAdapter(select, con);
da.Fill(ds, "Managers");

    // Construct the columns in the grid view
SetupColumns(ds);

    // Set the default height for a row
dataGridView.RowTemplate.Height = 100 ;

    // Then setup the datasource
dataGridView.AutoGenerateColumns = false;
dataGridView.DataSource = ds.Tables["Employees"];

}
```

这里要注意两个地方。第一个 select 语句用 0 替代 ReportsTo 列中的 null 值，数据库中有一行在这个字段中包含了 NULL 值，表示这个人没有上级。但是，在绑定数据时，ComboBox 需要这个列中

有值，否则在显示网格时会抛出一个异常。在这个例子中，选择显示 0，因为表中不存在 0-- 这通常称为 sentinel 值，因为它对应用程序有特殊的含义。

第二个 SQL 子句给 ComboBox 选择数据，包括创建值 Zero 和(None)时生成的行。在图 32-12 中，第二行显示了(None)。



图 32-12

定制的列用下面的函数创建：

```
private void SetupColumns(DataSet ds)
{
    DataGridViewTextBoxColumn forenameColumn = new
    DataGridViewTextBoxColumn();
    forenameColumn.DataPropertyName = "FirstName";
    forenameColumn.HeaderText = "Forename";
    forenameColumn.ValueType = typeof(string);
    forenameColumn.Frozen = true;
    dataGridView.Columns.Add(forenameColumn);

    DataGridViewTextBoxColumn surnameColumn = new
    DataGridViewTextBoxColumn();
    surnameColumn.DataPropertyName = "LastName";
    surnameColumn.HeaderText = "Surname";
    surnameColumn.Frozen = true;
    surnameColumn.ValueType = typeof(string);
    dataGridView.Columns.Add(surnameColumn);

    DataGridViewImageColumn photoColumn = new
    DataGridViewImageColumn();
    photoColumn.DataPropertyName = "Photo";
    photoColumn.Width = 100;
    photoColumn.HeaderText = "Image";
    photoColumn.ReadOnly = true;
    photoColumn.ImageLayout =
    DataGridViewImageCellLayout.Normal;
    dataGridView.Columns.Add(photoColumn);

    DataGridViewComboBoxColumn reportsToColumn = new
    DataGridViewComboBoxColumn();
    reportsToColumn.HeaderText = "Reports To";
    reportsToColumn.DataSource = ds.Tables["Managers"];
    reportsToColumn.DisplayMember = "Name";
    reportsToColumn.ValueMember = "EmployeeID";
    reportsToColumn.DataPropertyName = "ReportsTo";
    dataGridView.Columns.Add(reportsToColumn);
}
```

ComboBox 在本例的最后创建，并使用传送过来的 DataSet 中的 Managers 表用作其数据源。该表包含 Name 和 EmployeeID 列，它们分别赋予 DisplayMember 和 ValueMember 属性。这两个属性定义了 ComboBox 的数据来自何方。

PropertyName 设置为组合框链接的主表中的列，它提供了列的初始值，如果用户从组合框中选择另一个值，就更新这个值。

这个例子还需要在更新数据库时正确处理空值。目前，如果在屏幕上选择了(None)，该例子仅把值 0 写入数据行。在 SQL Server 中，这会引发一个异常，因为这违反了 ReportsTo 列的外键码约束。为了解决这个问题，需要在把数据发送回 SQL Server 之前，预先处理它，即把行中值为 0 的 ReportsTo 列再设置为 NULL。

### 32.3 数据绑定

前面的示例都使用了 DataGrid 控件和 DataGridView 控件，这是在.NET 运行库中可以显示数据的两个控件。把控件链接到数据源的过程称为数据绑定。

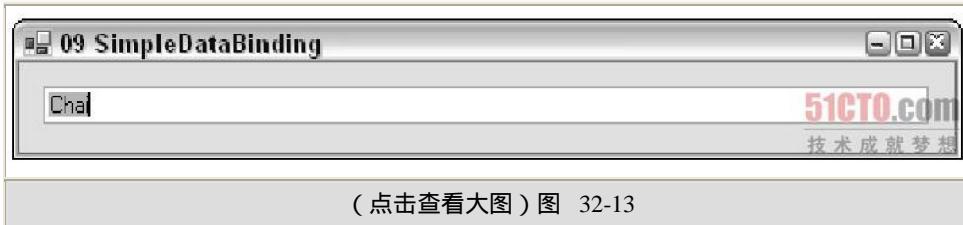
在 MFC 库中，把数据从类变量链接到一组控件上的过程称为对话框数据交换(Dialog Data Exchange ,DDX)。在.NET 中，可用于把数据绑定到控件上的功能更容易使用，也更强大。例如，在.NET 中，可以把数据绑定到控件的大多数属性上，而不仅仅是文本属性。还可以用类似的方式把数据绑定到 ASP.NET 控件上(参阅第 37 章)。

#### 32.3.1 简单的绑定

支持单一绑定的控件一般一次只显示一个值，例如文本框或单选按钮，下面的示例说明了如何把 DataTable 中的一列绑定到 TextBox 上：

```
DataSet ds = CreateDataSet();
textBox1.DataBindings.Add("Text", ds ,
"Products.ProductName");
```

用上面的方法 CreateDataSet() 从 Products 表中检索一些数据，并保存到返回的 DataSet 中，之后，第二行代码把控件(textBox1)的 Text 属性绑定到 Products.ProductName 列上。图 32-13 显示了这种数据绑定的结果。



(点击查看大图) 图 32-13

文本框显示了数据库中的一个字符串，要检查这个数据是否放在正确的列上并有正确的值，可以使用 SQL Server Management Studio 工具，验证 Products 表中的内容，如图 32-14 所示。



(点击查看大图) 图 32-14

屏幕上有一个文本框，既不能滚动到下一个或上一个记录上，也不能更新数据库，因此下一节介绍一个更真实的示例，说明如何使用其他对象绑定数据。

### 32.3.2 数据绑定对象

图 32-15 显示了数据绑定中使用的对象的类层次结构。本节将讨论 System.Windows.Forms 命名空间中的类 BindingContext、CurrencyManager 和 PropertyManager，说明在把数据绑定到窗体上的一个或多个控件上时，它们是如何交互的。带阴影的对象就是在绑定中使用的对象。

在前面的示例中，我们使用 TextBox 控件的 DataBindings 属性把 DataSet 的一列绑定到控件的 Text 属性上，DataBindings 属性是图 32-15 所示的 ControlBindingsCollection 的一个实例。

(点击查看大图) 图 32-15

```
textBox1.DataBindings.Add("Text", ds,  
"Products.ProductName");
```

这行代码给 ControlBindingsCollection 添加一个 Binding 对象。

#### 1. BindingContext

每个 Windows 窗体都有 BindingContext 属性，实际上，Form 派生自 Control，该属性是在 Control 中定义的，所以大多数控件都有这个属性。BindingContext 对象有一个 BindingManagerBase 实例集合，如图 32-16 所示。在对控件进行数据绑定时，就会创建这些实例，并把它们添加到绑定管理器对象中。



图 32-16

BindingContext 可以包含几个数据源，包装在 CurrencyManager 或 PropertyManager 中。使用哪个类取决于数据源本身。

如果数据源包含一个项目列表，例如 DataTable、DataView 或实现 IList 接口的对象，就使用 CurrencyManager，因为它可以在该数据源中保存当前位置。如果数据源只返回一个值，就把 PropertyManager 存储在 BindingContext 中。

只为给定的数据源创建一次 CurrencyManager 或 PropertyManager。如果把两个文本框绑定到 DataTable 的一个行上，则在 BindingContext 中只创建一个 CurrencyManager。

添加到窗体中的每个控件都链接到窗体的绑定管理器上，因此所有的控件都共享相同的实例。在最初创建一个控件时，其 BindingContext 属性为空。在把控件添加到窗体的 Controls 集合中时，就把 BindingContext 设置为该窗体的 Controls 集合。

要把控件绑定到一个列上，需要给其 DataBindings 属性添加一项，这是 ControlBindingsCollection 的一个实例。下面的代码可以创建一个新绑定：

```
textBox1.DataBindings.Add("Text", ds,
```

```
"Products.ProductName");
```

ControlBindingsCollection 的 Add()方法会从传递给它的参数中创建 Binding 对象的一个实例，并把它添加到绑定集合中，如图 32-17 所示。

图 32-17 显示了把 Binding 实例添加到控件中的情况。绑定把控件链接到数据源上，存储在 Form(或控件本身)的 BindingContext 中。数据源内部的改变会反映到控件上，控件中的改变也会反映到数据源上。

(点击查看大图) 图 32-17

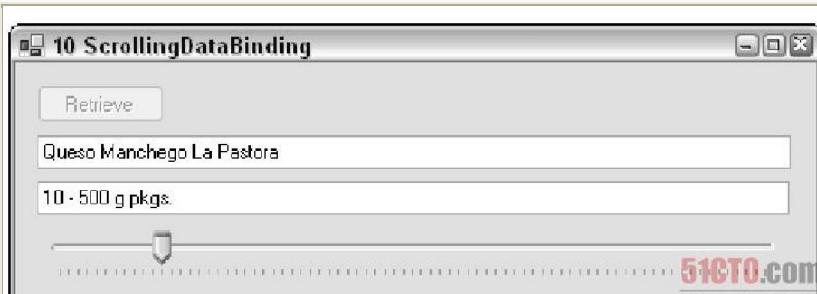
## 2. Binding 类

这个类把控件的一个属性链接到数据源的一个成员上。在改变该成员时，控件的属性会更新，以反映这个改变。反之亦然，如果文本框中的文本被更新，这个改变也会反映到数据源上。

可以把任何列绑定到控件的任何属性上，例如，可以把列绑定到一个文本框的 Text 属性中，也可以把另一个列绑定到文本框的 Color 属性上。可以把控件的属性绑定到完全不同的数据源上，例如，单元格的颜色在一个颜色表中定义，而实际的数据在另一个表中定义。

## 3. CurrencyManager 和 PropertyManager

在创建 Binding 对象时，如果这是第一次绑定数据源中的数据，就会创建对应的 CurrencyManager 或 PropertyManager 对象。这个类的作用是定义当前记录在数据源中的位置，在改变当前的记录时，需要调整所有的 ListBindings。图 32-18 显示了 Products 表中的两个字段，包含一种通过跟踪栏控件在记录之间移动的方式。



(点击查看大图) 图 32-18

ScrollingDataBinding 示例的代码如下所示：

```
namespace ScrollingDataBinding
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

```
}

    private DataSet CreateDataSet()
{
    string customers = "SELECT * FROM Products";
    DataSet ds = new DataSet();

        using (SqlConnection con = new SqlConnection (
ConfigurationSettings.ConnectionStrings["northwind"].ConnectionString))
{
    SqlDataAdapter da = new SqlDataAdapter(customers, con);

        da.Fill(ds, "Products");
}

    return ds;
}

    private void trackBar_Scroll(object sender, EventArgs e)
{
    this.BindingContext[ds, "Products"].Position = trackBar.Value;
}

    private void retrieveButton_Click(object sender, EventArgs e)
{
    retrieveButton.Enabled = false;

    ds = CreateDataSet();

    textName.DataBindings.Add("Text", ds, "Products.ProductName");
textQuan.DataBindings.Add("Text", ds, "Products.QuantityPerUnit");

    trackBar.Minimum = 0;
trackBar.Maximum = this.BindingContext[ds, "Products"].Count - 1;

    textName.Enabled = true;
textQuan.Enabled = true;
trackBar.Enabled = true;

}

    private DataSet ds;
}
}
```

滚动机制在 trackBar\_Scroll 事件处理程序中提供，该处理程序把 BindingContext 的位置设置为跟踪条的当前位置，改变 BindingContext 会更新显示在屏幕中的数据。

在 retrieveButton\_Click 事件中添加一个数据绑定表达式，就把数据绑定到两个文本框上，这里控件的 Text 属性设置为数据源中的字段。可以把控件的任意简单数据绑定到数据源的一项上，例如，可以绑定 text color、enabled 等属性。

在开始检索数据时，跟踪栏的最大位置就设置为记录的个数。接着在上面的滚动方法中，把 Products 数据表中 BindingContext 的位置设置为滚动条的位置，这样就可以有效地改变 DataTable 中的当前记录，绑定到当前行上的所有控件(在本例中是两个文本框)就会被更新。

本节介绍了如何绑定到各种数据源(例如数组、数据表、数据视图和各种其他数据容器)上，如何排序和过滤数据。下一节将讨论如何扩展 Visual Studio，以允许进行数据访问，得到与应用程序的更好集成。

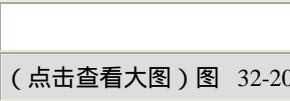
### 32.4 Visual Studio 和数据访问

本节讨论 Visual Studio 把数据集成到 GUI 中的一些新方式，具体地说，就是讨论如何创建连接，选择一些数据，生成一个 DataSet，再使用生成的所有对象创建一个简单的应用程序。Visual Studio 提供的工具可以使用 OleDbConnection 或 SqlConnection 类创建一个数据库连接，使用哪个类取决于要连接的数据库类型。定义了一个连接后，就可以创建一个 DataSet，在 Visual Studio 中给它填充数据，这会为 DataSet 生成一个 XSD 文件和.cs 代码，最终会创建一个类型安全的 DataSet。

#### 32.4.1 创建一个连接

首先，创建一个新的 Windows 应用程序，之后创建一个数据库连接。使用 Server Explorer，可以管理数据访问的各个方面，如图 32-19 所示。

在本例中，需要创建 Northwind 数据库的一个连接。从 Data Connections 项的弹出菜单中选择 Add Connection 选项，会打开一个向导，该向导允许选择数据库提供程序。这里选择.NET Framework Provider for SQL Server。Add Connection 对话框如图 32-20 所示。



根据.NET Framework 安装，示例数据库可能位于 SQL Server、MSDE(Microsoft SQL Server Data Engine)，或者在这两个地方都有。

要连接本地 MSDE 数据库(如果有)，键入(Local)\NETSDK 作为服务器的名称。要连接一般的 SQL Server 实例，应键入(local)或'.'，选择当前机器上的一个数据库，或者选择网络上需要的服务器名称。还需要输入用户名和密码来访问数据库。

从数据库的下拉列表中选择 Northwind 数据库，为了确保正确安装了所有的文件，单击 Test Connection 按钮，如果一切安装正确，就应显示一个包含确认信息的消息框。

Visual Studio 2005 在数据访问方面有许多变化，这些在用户界面的几个地方可以看出。我喜欢使用新的 Data 菜单，它可以查看已添加到项目中的数据源，添加新数据源，预览底层数据库(或其他数据源)中的数据。

下面的例子使用 Northwind 数据库连接生成一个用户界面，用于从 Employees 表中选择数据。第一步是从 Data 菜单中选择 Add New Data Source 打开一个向导，利用该向导完成后面的步骤。图 32-21 显示了 Data Sources Configuration 向导的一部分，从中可以为数据源选择合适的表。

(点击查看大图) 图 32-21

在向导中，可以选择数据源，它可以是数据库、本地数据库文件(例如.mdb 文件)、Web 服务或一个对象。接着向导要求用户根据所选择的数据源类型提供更详细的信息。对于数据库连接，要提供的信息有连接名称(它随后存储在应用程序配置文件中，如下面的代码所示)，接着选择提供数据的表、视图或存储过程。最后，这会在应用程序中生成一个强类型化的 DataSet。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<connectionStrings>
<add
name="SimpleApp.Properties.Settings.NorthwindConnection"
connectionString="Data Source=.;Integrated Security=True;
Initial Catalog=Northwind"
providerName="System.Data.SqlClient" />
</connectionStrings>
</configuration>
```

其中包括连接的名称、连接字符串和提供程序的名称(在生成连接对象时使用)。可以根据需要手工编辑这些信息。要给雇员数据显示一个用户界面，可以把选中的数据从 Data Sources 窗口拖放到窗体上。这会生成两种 User Interface 样式中的一种：利用前面介绍的 DataGridView 控件的网格样式 UI，或者一次只显示一个记录的 Details 视图。图 32-22 显示了 Details 视图。

把数据源拖放到窗体上会生成许多对象，它们有的可见，有的不可见。不可见的对象在窗体的组件区域中创建，包含 DataConnector、强类型化的 DataSet 和 TableAdapter(它包含用于选择或更新数据的 SQL)。可见的对象根据是选择 DataGridView 还是 Details 视图来创建。无论选择 DataGridView 还是 Details 视图，都会创建一个 DataNavigator 控件，用于给数据分页显示。图 32-23 显示了使用 DataGridView 控件生成的用户界面--Visual Studio 2008 的一个目标是简化数据访问，即不编写任何代码，就能生成功能全面的窗体。

在创建数据源时，要在解决方案中添加许多文件，要查看它们，需要单击 Solution Explorer 中的 Show All Files 按钮，然后就可以展开数据集节点，查看添加的文件。其中比较有趣的是.Designer.cs 文件，它包含用于填充数据集的 C# 源代码。

图 32-22

(点击查看大图) 图 32-23

在.Designer.cs 文件中定义了几个类，它们表示强类型化的数据集，其操作方式类似于标准的 DataAdapter 类。这个类在内部使用 DataAdapter 填充 DataSet。

#### 32.4.2 选择数据

生成的表数据适配器包含 SELECT、INSERT、UPDATE 和 DELETE 命令。显然，也可以调用存储过程，而不直接使用 SQL 命令。向导生成的代码也可以完成这个工作。Visual Studio 给 .Designer 文件添加如下代码：

```
private System.Data.SqlClient.SqlCommand  
m_DeleteCommand;  
  
private System.Data.SqlClient.SqlCommand  
m_InsertCommand;  
  
private System.Data.SqlClient.SqlCommand  
m_UpdateCommand;  
  
private System.Data.SqlClient.SqlDataAdapter m_adapter;
```

除了 Select 命令之外，为每个 SQL 命令定义一个对象和一个 SqlDataAdapter。在文件后面的 InitializeComponent() 方法中，向导生成了创建这些命令和数据适配器的代码。

在 Visual Studio 的以前版本中，为 Insert 和 Update 生成的命令也包含一个 select 子句，以便使数据与服务器上的数据再次同步。其实只是计算数据库中的所有字段(例如标识列和计算字段)。

向导生成的代码可以工作，但有时它并不是最佳的。对于产品系统，所有生成的 SQL 语句都应用存储过程的调用来替换。如果 INSERT 或 UPDATE 子句不需要重新同步数据，删除多余的 SQL 子句可以加速应用程序的运行。

#### 32.4.3 更新数据源

应用程序已经从数据库中选择了数据，本节将说明把数据的改变返回至数据库。如果您按照上一节的步骤进行，就应得到一个包含所有必要对象的应用程序。剩下的就只是激活工具栏上的 Save 按钮，编写一个事件处理程序，更新数据库。

在 IDE 上，从数据导航控件中选择 Save 按钮，把 Enabled 属性改为 true。然后双击按钮，生成一个事件处理程序。在这个事件处理程序中，把屏幕上数据的改变返回至数据库：

```
private void dataNavigatorSaveItem_Click(object sender,  
EventArgs e)  
{  
    employeesTableAdapter.Update(employeesDataset.Employees);  
}
```

Visual Studio 完成了大部分工作，我们只需使用表达适配器类的 Update 方法。表达适配器有 6 个 Update 方法，这个例子使用把 DataTable 作为参数的重写版本。

#### 32.4.4 其他常见的要求

在显示数据时，一个常见的要求是为给定的行提供一个弹出式菜单。这有很多解决方式，本例介绍的一种方式可以简化需要的代码，特别适合于下述情况：显示环境是 DataGrid，在该环境中显示了带有一些关系的 DataSet。问题是弹出的菜单与所选的行有关，该行可以来自于该 DataSet 中的任意一个源数据表。

因为弹出菜单的功能似乎很一般，所以这里利用一个支持建立菜单的基类(ContextDataRow)，每个数据行类都支持从这个基类派生来的一个弹出菜单。

用户右击 DataGrid 中一行的任何一部分，都可以查看该行，确定它是否派生于 ContextDataRow，如果是，就可以调用 PopupMenu() 函数。这可以使用一个接口来实现，但在本例中，使用基类会比较简单。

本例说明了如何生成 DataRow 和 DataTable 类，以便对数据进行类型安全的访问。这与前面 XSD 示例采用的方式相同。但这次代码是手工编写的，来说明在这种情况下如何使用定制属性和反射。

图 32-24 显示了这个示例的类层次结构。



这个示例的代码如下：

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;

public class ContextDataRow : DataRow
{
    public ContextDataRow(DataRowBuilder builder) :
        base(builder)
    {
    }

    public void PopupMenu(System.Windows.Forms.Control
parent, int x, int y)
{
    // Use reflection to get the list of popup menu
    // commands
    MemberInfo[] members = this.GetType().FindMembers(
        MemberTypes.Method,
        BindingFlags.Public | BindingFlags.Instance ,
        new System.Reflection.MemberFilter(Filter),
        null);
    if (members.Length > 0)
    {
    }
}
```

```
// Create a context menu

ContextMenu menu = new ContextMenu();

// Now loop through those members and generate
the popup menu
// Note the cast to MethodInfo in the foreach
foreach (MethodInfo meth in members)
{

    // Get the caption for the operation from the
    // ContextMenuAttribute

    ContextMenuAttribute[] ctx =
(ContextMenuAttribute[])
meth.GetCustomAttributes(typeof(ContextMenuAttribute),
true);

MenuCommand callback = new MenuCommand(this, meth);
MenuItem item = new MenuItem(ctx[0].Caption, new
EventHandler(callback.Execute));
item.DefaultItem = ctx[0].Default;
menu.MenuItems.Add(item);
}

System.Drawing.Point pt = new System.Drawing.Point(x,y);
menu.Show(parent, pt);
}

}

private bool Filter(MemberInfo member, object
criteria)
{
bool bInclude = false;

// Cast MemberInfo to MethodInfo

MethodInfo meth = member as MethodInfo;
if (meth != null)
if (meth.ReturnType == typeof(void))
{
ParameterInfo[] parms = meth.GetParameters();
if (parms.Length == 0)
{

    // Lastly check if there is a
ContextMenuAttribute on the
// method...

object[] attrs = meth.GetCustomAttributes
(typeof(ContextMenuAttribute), true);

```

```
bInclude = (atts.Length == 1);
}
}
return bInclude;
}
}
```

ContextDataRow 类派生自 DataRow，且该类只包含两个成员函数。第一个函数 PopupMenu 使用反射来查找对应于特定签名的方法，它还给用户显示这些选项的一个弹出菜单。第二个函数 Filter() 在枚举方法时由 PopupMenu 用作一个委托。如果成员函数符合相应的调用约定，它就只返回 true。

```
MethodInfo[] members =
this.GetType().FindMembers(MemberTypes.Method,
BindingFlags.Public | BindingFlags.Instance,
new System.Reflection.MemberFilter(Filter),
null);
```

这个语句用于过滤当前对象上的所有方法，仅返回与下述条件相匹配的方法：

成员必须是一个方法

成员必须是一个公共实例方法

成员必须没有返回值

成员必须不带参数

成员必须包含 ContextMenuAttribute

最后一个条件是一个定制属性，是专门为这个示例编写的。在完成 PopupMenu 方法的分析后讨论这个定制属性：

```
ContextMenu menu = new ContextMenu();
foreach (MethodInfo meth in members)
{
    // ... Add the menu item
}
System.Drawing.Point pt = new System.Drawing.Point(x,y);
menu.Show(parent, pt);
```

创建一个关联菜单实例，为每个满足上述条件的方法添加一个弹出菜单选项。接着，显示该菜单，如图 32-25 所示。

这个示例中的主要困难是下面这段代码，在弹出的菜单中，每个要显示的成员函数都要重复一次该段代码：

```
System.Type ctxtype = typeof(ContextMenuAttribute);
ContextMenuAttribute[] ctx = (ContextMenuAttribute[])
```

```
meth.GetCustomAttributes(ctxttype, true);
MenuCommand callback = new MenuCommand(this, meth);
MenuItem item = new MenuItem(ctx[0].Caption,
new EventHandler(callback.Execute));
item.DefaultItem = ctx[0].Default;
menu.MenuItems.Add(item);
```

(点击查看大图) 图 32-25

在弹出菜单中显示的每个方法都有一个 ContextMenuAttribute 属性。它为菜单选项定义了一个用户友好的名称，因为 C# 方法名不能有空格，所以在弹出菜单中使用英语要比使用某些内部代码更好。该属性从方法中获取，创建一个新菜单项，并把它添加到弹出式菜单的菜单项集合中。

这个示例代码还说明了简化的 Command 类的用法(一种常见的设计模式)。这个示例中使用的 MenuCommand 是在用户选择关联菜单中的一项时触发的，它会把调用传送给方法的接收器，在本例中是对象及所属的方法。这也有助于使接收器对象中的代码更容易与用户界面代码隔离开来。下面几节将解释这些代码。

### 1. 生成的表和行

本章前面的 XSD 示例显示的代码是在使用 Visual Studio 编辑器生成一组数据访问类时生成的。下面的类显示了 DataTable 需要的方法，这是为这个类生成的最小一组代码(它们都是手工生成的)：

```
public class CustomerTable : DataTable
{
    public CustomerTable() : base("Customers")
    {
        this.Columns.Add("CustomerID", typeof(string));
        this.Columns.Add("CompanyName", typeof(string));
        this.Columns.Add("ContactName", typeof(string));
    }
    protected override System.Type GetRowType()
    {
        return typeof(CustomerRow);
    }
    protected override DataRow
        NewRowFromBuilder(DataRowBuilder builder)
    {
        return(DataRow) new CustomerRow(builder);
    }
}
```

首先，DataTable 必须重写 GetRowType()方法，这是在生成表的新行时由.NET 在内部使用的。这个方法应返回用于表示每一行的类型。

其次，需要执行 NewRowFromBuilder()，它也是在为表创建新行时由运行库调用的。这对于最小执行方式是足够的。对应的 CustomerRow 类是相当简单的，它为行中的每一列执行属性，然后执行最终显示在关联菜单中的方法：

```
public class CustomerRow : ContextDataRow
{
    public CustomerRow(DataRowBuilder builder) :
        base(builder)
    {
    }

    public string CustomerID
    {
        get { return (string)this["CustomerID"]; }
        set { this["CustomerID"] = value; }
    }

    // Other properties omitted for clarity

    [ContextMenu("Blacklist Customer")]
    public void Blacklist()
    {
        // Do something
    }

    [ContextMenu("Get Contact",Default=true)]
    public void GetContact()
    {
        // Do something else
    }
}
```

该类派生于 ContextDataRow，包括相应的 getter/setter 方法，其属性名与每个字段是相同的，然后添加一些方法，在反射到类上时会使用这些方法：

```
[ContextMenu("Blacklist Customer")]
public void Blacklist()
{
    // Do something
}
```

每个要显示在关联菜单中的方法都有相同的签名，且包括定制的 ContextMenu 属性。

## 2. 使用属性

编写 ContextMenu 属性的理念是要能为给定的菜单选项提供一个自由文本名称，下面的示例还使用了一个 Default 标志，它用于指定默认的菜单选项。下面给出完整的属性类：

```
[AttributeUsage(AttributeTargets.Method,AllowMultiple=false,Inherited=true)]
public class ContextMenuAttribute : System.Attribute
```

```
{  
public ContextMenuAttribute(string caption)  
{  
Caption = caption;  
Default = false;  
}  
public readonly string Caption;  
}
```

类的 AttributeUsage 属性把 ContextMenuAttribute 标记为唯一可以在方法上使用的属性 ,它还定义了在任何给定的方法上只有该对象的一个实例。 Inherited=true 子句定义了属性是否可以放在一个超类方法上 ,且仍由子类反映出来。

可以把许多其他成员添加到这个属性上 ,例如 :

菜单选项的热键

要显示的图像

要显示在工具栏中的文本 ,当鼠标放在菜单选项上时 ,就会显示该文本

帮助上下文 ID

### 3. 分派方法

当菜单显示在.NET 中时 ,每个菜单选项都通过委托的方式链接到该选项的处理代码上。把菜单选项链接到代码上的机制中 ,有两个选项 :

执行与 System.EventHandler 有相同签名的方法 ,其定义如下 :

```
public delegate void EventHandler(object sender,  
EventArgs e);
```

定义一个代理类 ,它执行上述委托 ,把调用传送给接收的类。这称为 Command 模型 ,也是本例选择的模型。

Command 模型通过一个简单的中间类把调用的发送者和接收者分离开来。对于本例 ,该模型就矫枉过正了 ,但可以使每个 DataRow 上的方法更简单一些(因为它们不需要给委托传送参数) ,而且可扩展性更高 :

```
public class MenuCommand  
{  
public MenuCommand(object receiver, MethodInfo method)  
{  
Receiver = receiver;  
Method = method;  
}  
public void Execute(object sender, EventArgs e)
```

```
{  
    Method.Invoke(Receiver, new object[] {} );  
}  
public readonly object Receiver;  
public readonly MethodInfo Method;  
}
```

该类只提供了一个 EventHandler 委托(Execute 方法) , 它在接收器对象上调用所希望的方法。本例显示了两种不同类型的行-- Customers 表中的行和 Orders 表中的行。自然 , 这些类型的数据的处理选项都是不同的。图 32-25 显示了可以应用到 Customer 行上的操作。图 32-26 显示了可以应用到 Order 行上的操作。

(点击查看大图) 图 32-26

#### 4. 获得选中的行

这个示例的最后一个问题是如何处理 DataSet 中用户选择的行。首先考虑"它必须是 DataGrid 的一个属性" , 但在 DataGrid 上不能使用这个属性。查看一下从 MouseUp()事件处理程序中获得的测试信息 , 但只有在显示一个 DataTable 中的数据时 , 这才有一定的帮助。

下面介绍如何填充网格 :

```
dataGrid.SetDataBinding(ds, "Customers");
```

这个方法给 BindingContext 添加一个新的 CurrencyManager , 它表示当前数据表和 DataSet。现在 , DataGrid 有两个属性 DataSource 和DataMember , 在调用 SetDataBinding()时设置它们。本例中的 DataSource 是一个 DataSet , DataMember 是 Customers。

给定数据源和一个数据成员 , 以及窗体的 BindingContext , 就可以使用下面的代码查找当前行 :

```
protected void dataGrid_MouseUp(object sender,  
MouseEventArgs e)  
{  
    // Perform a hit test  
    if(e.Button == MouseButtons.Right)  
    {  
        // Find which row the user clicked on, if any  
        DataGrid.HitTestInfo hti = dataGrid.HitTest(e.X, e.Y);  
  
        // Check if the user hit a cell  
        if(hti.Type == DataGrid.HitTestType.Cell)  
        {  
            // Find the DataRow that corresponds to the cell  
            //the user has clicked upon
```

在调用 `dataGrid.HitTest()` 确定用户在什么地方单击后，就要为 DataGrid 检索 BindingManagerBase 实例了：

```
BindingManagerBase bmb =  
this.BindingContext[ dataGrid.DataSource,  
dataGrid.DataMember];
```

它使用了 DataGrid 的 DataSource 和 DataMember，给要返回的对象命名。现在只需查找用户单击的行，显示关联菜单。右击了一个行后，当前行的指示符一般不会移动，但这还不够，还要移动行的指示符，再显示弹出菜单。HitTestInfo 对象中有行号，所以只需移动 Bindind ManagerBase 对象的当前位置：

```
bmb.Position = hti.Row;
```

这会改变单元格的指示符，同时意味着在调用类来获取 Row 时，将返回当前行，而不是上次选择的行：

```
DataRowView drv = bmb.Current as DataRowView;  
if(drv != null)  
{  
    ContextDataRow ctx = drv.Row as ContextDataRow;  
    if(ctx != null) ctx.PopupMenu(dataGrid,e.X,e.Y);  
}
```

在 DataGrid 显示 DataSet 中的元素时，BindingManagerBase 集合中的 Current 对象是一个 DataRowView，它在上面的代码中用一个明确的类型转换进行了测试。如果成功，就可以获取 DataRowView 包装的实际行，执行另一个类型转换，确定它是否是 ContextDataRow，最终会弹出一个菜单。

在本例中，创建了两个数据表-- Customers 和 Orders，定义了这些表之间的关系，这样在用户单击 CustomerOrders 时，会得到一个订单的过滤列表。此时，DataGrid 把 DataMember 从 Customers 变为 Customers.CustomerOrders，BindingContext 索引器使用它获取要显示的数据。

### 32.5 小结

本章介绍了在.NET 上显示数据的一些方法，System.Windows.Forms 中有许多可用的类。本章重点介绍了如何使用 DataGridView 和 DataGrid 控件显示许多不同数据源中的数据，例如 Array、DataTable 或 DataSet。

在网格中显示数据并不总是很合适，所以本章也讨论了如何把一列数据链接到用户界面中的一个控件上。.NET 的绑定功能使这种用户界面非常容易获得支持，因为一般情况下，是把控件绑定到一列上，让.NET 处理其他任务。

下一章介绍用 GDI+绘图的功能。

### 第 33 章使用 GDI+绘图

在本书中，有 8 章内容介绍用户交互和.NET Framework，第 31 章主要介绍了如何显示对话框或 SDI、MDI 窗口，以及如何把各种控件放在这些窗口上，如按钮、文本框和列表框。第 32 章介绍在 Windows 窗体中使用许多 Windows 窗体控件处理各种数据源中的数据。

这些标准控件的功能非常强大，使用它们就可以获得许多应用程序的完整用户界面。但是，有时还需要在用户界面上有更大的灵活性。例如，要在窗口的确定位置以给定的字体绘制文本，或者显示图像，但不使用图像框控件，只使用形状和图形。这些都不能使用第 31 章介绍的控件来完成。要显示这种类型的输出，应用程序必须直接告诉操作系统需要在其窗口的什么地方显示什么内容。

本章主要介绍如何绘制以下内容：

绘图规则

直线、简单图形

.BMP 图像和其他图像文件

文本

处理打印

在这个过程中，还需要使用各种帮助对象，包括钢笔(用于定义直线的特性)、画笔(用于定义区域的填充方式)和字体(用于定义文本字符的图形)。我们还将介绍设备如何解释和显示不同的颜色。

下面首先讨论 GDI+技术。GDI+由.NET 基类集组成，这些基类可用于在屏幕上完成定制绘图，能把合适的指令发送到图形设备的驱动程序上，确保在监视器屏幕上显示正确的输出(或打印到硬拷贝中)。

### 33.1 理解绘图规则

本节讨论一些基本规则，只有理解了它们，才能开始在屏幕上绘图。首先概述 GDI，GDI+技术就建立在 GDI 上，然后说明它与 GDI+的关系。接着介绍几个简单的例子。

#### 33.1.1 GDI 和 GDI+

一般来说，Windows 的一个优点（实际上是现代操作系统的优点）是它可以让开发人员不考虑特定设备的细节。例如，不需要理解硬盘设备驱动程序，只需在相关的.NET 类中调用合适的方法（在.NET 推出之前，使用等价的 Windows API 函数），就可以编程读写磁盘上的文件。这个规则也适用于绘图。计算机在屏幕上绘图时，把指令发送给视频卡。问题是市面上有几百种不同的视频卡，大多数有不同的指令集和功能。如果把这个考虑在内，在应用程序中为每个视频卡驱动程序编写在屏幕上绘图的特定代码，这样的应用程序就根本不可能编写出来。这就是为什么在 Windows 最早期的版本中就有 Windows Graphical Device Interface (GDI) 的原因。

GDI+提供了一个抽象层，隐藏了不同视频卡之间的区别，这样就可以调用 Windows API 函数完成指定的任务了，GDI 会在内部指出在运行特定的代码时，如何让客户机的视频卡完成要绘制的图形。GDI 还可以完成其他任务。大多数计算机都有多个显示设备——例如，监视器和打印机。GDI 成功地使应用程序所使用的打印机看起来与屏幕一样。如果要打印某些东西，而不是显示它们，只需告诉系统输出的设备是打印机，再用相同的方式调用相同的 Windows API 函数即可。

可以看出，DC(设备环境)是一个功能非常强大的对象，在 GDI 下，所有的绘图工作都必须通过设备环境来完成。DC 甚至可用于不涉及在屏幕或其他硬件设备上绘图的其他操作，例如在内存中修改图像。

GDI 给开发人员提供了一个相当高级的 API，但它仍是一个基于旧的 Windows API 并且有 C 语言风格函数的 API，所以使用起来不是很方便。GDI+在很大程度上是 GDI 和应用程序之间的一层，提供了更直观、基于继承性的对象模型。尽管 GDI+基本上是 GDI 的一个包装器，但 Microsoft 已经能通过 GDI+提供新功能了，它还有一些性能方面的改进。

.NET 基类库的 GDI+部分非常大，本章不解释其特性。这是因为只要解释其中的几个类、方法和属性，就会把本章变成一个仅列出 GDI+类和方法的参考指南。而理解绘图的基本规则更重要；这样您应可以自己研究这些类。当然，关于 GDI+中类和方法的完整列表，可以参阅 SDK 文档说明。

注意：

有 VB6 背景的开发人员会发现，自己并不熟悉绘图过程涉及的概念，因为 VB6 的重点是处理绘图的控件。有 C++/MFC 背景的开发人员则比较熟悉这个领域，因为 MFC 要求开发人员使用 GDI 更多地控制绘图过程。但是，即使您具备很好的 GDI 背景知识，也会发现本章有许多新东西。

## 1. GDI+命名空间

表 33-1 列出了 GDI+基类的主要命名空间。

本章使用的几乎所有的类、结构等都包含在 System.Drawing 命名空间中。

表 33-1

命 名 空 间	说 明
System.Drawing	包含与基本绘图功能有关的大多数类、结构、枚举和委托
System.Drawing.Drawing2D	为大多数高级 2D 和矢量绘图操作提供了支持，包括消除锯齿、几何转换和图形路径

(续表)

命 名 空 间	说 明
System.Drawing.Imaging	帮助处理图像(位图、GIF 文件等)的各种类
System.Drawing.Printing	把打印机或打印预览窗口作为输出设备时使用的类
System.Drawing.Design	一些预定义的对话框、属性表和其他用户界面元素，与在设计期间扩展用户界面相关
System.Drawing.Text	对字体和字体系列执行更高级操作的类

## 2. 设备环境和 Graphics 对象

在 GDI 中，识别输出设备的方式是使用设备环境(DC)对象。该对象存储特定设备的信息，并能把 GDI API 函数调用转换为要发送给该设备的指令。还可以查询设备环境对象，确定对应的设备有什么功能(例如，打印机是彩色的，还是黑白的)，这样才能据此调整输出结果。如果要求设备完成它不能完成的任务，设备环境对象就会检测到，并采取相应的措施(这取决于具体的情形，例如抛出一个异常，或修改请求，获得与该设备的能力最相近的匹配)。

设备环境对象不仅可以处理硬件设备，还可以用作 Windows 的一个桥梁，因此能考虑到 Windows 绘图的要求或限制。例如，如果 Windows 知道只有一小部分应用程序窗口需要重新绘制，设备环境就可以捕获和撤销在该区域外绘图的工作。因为设备环境与 Windows 的关系非常密切，通过设备环境来工作就可以在其他方面简化代码。

例如，硬件设备需要知道在什么地方绘制对象，通常它们需要相对于屏幕(或输出设备)左上角的坐标。但应用程序常常要使用自己的坐标系统在自己窗口的客户区域(用于绘图的窗口)上绘图。而因为窗口可以放在屏幕上的任何位置，用户可以随时移动它，所以在两个坐标之间转换就是一个比较困难的任务。设备环境总是知道窗口在什么地方，并能自动进行这种转换。

在 GDI+中，设备环境包装在.NET 基类 System. Drawing.Graphics 中。大多数绘图工作都是调用 Graphics 实例的方法完成的。实际上，因为 Graphics 类负责处理大多数绘图操作，所以 GDI+中很少有操作不涉及到 Graphics 实例。理解如何处理这个对象是理解如何使用 GDI+在显示设备上绘图的关键。

### 33.1.2 绘制图形

下面用一个小示例 DisplayAtStartup 来说明如何在应用程序的主窗口中绘图。本章的示例都在 Visual Studio 2008 中创建为 C# Windows 应用程序。对于这种类型的项目，代码向导会提供一个类 Form1，它派生自 System.Windows.Form，表示应用程序的主窗口。还会生成一个类 Program(在 Program.cs 文件中)，表示应用程序的主起点。除非特别声明，否则在所有的示例中，新代码或修改过的代码都添加到向导生成的代码中(可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载示例代码)。

注意：

在.NET 的用法中，当说到显示各种控件的应用程序时，“窗口”大都用术语“窗体”来代替，表示一个矩形对象，它占据了屏幕上的一块区域，代表应用程序。在本章中，我们使用术语“窗口”，因为在手工绘图时，它更有意义。当谈到用于实例化窗体/窗口的.NET 类时，使用术语“窗体”。最后，“绘图”或“绘制”可以互换使用，描述在屏幕或其他显示设备上显示一些项的过程。

第一个示例创建一个窗体，并在启动窗体时在构造函数中绘制它。这并不是在屏幕上绘图的最佳方式，这个示例并不能在启动后按照需要重新绘制窗体。但利用这个示例，我们不必做太多的工作，就可以说明绘图的许多问题。

对于这个示例，启动 Visual Studio 2008，创建一个 Windows 应用程序，首先把窗体的背景色设置为白色。把这行代码放在 InitializeComponent()方法的后面，这样 Visual Studio 2008 就会识别该命令，并改变窗体的设计视图的外观。在 Visual Studio Solution Explorer 中单击 Show All Files 按钮，再展开 Form1.cs 文件旁边的加号，就可以看到 Form1.Designer.cs 文件，在这个文件中，包含了 InitializeComponent()方法。也可以使用设计视图设置背景色，这会自动添加相同的代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode =
        System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
}
```

接着，给 Form1 构造函数添加代码。使用窗体的 CreateGraphics()方法创建一个 Graphics 对象，这个对象包含绘图时需要使用的 Windows 设备环境。创建的设备环境与显示设备相关，也与这个窗口相关。

```
public Form1()
{
    InitializeComponent();

    Graphics dc = this.CreateGraphics();
    Show();
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
```

```
dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

然后调用 Show()方法显示窗口。之所以让窗口立即显示，是因为在窗口显示出来之前，我们不能做任何工作-- 没有可供绘图的地方。

最后，显示一个矩形，其坐标是(0,0)，宽度和高度是 50，再绘制一个椭圆，其坐标是(0, 50)，宽度是 80，高度是 50。注意坐标(x,y)表示从窗口的客户区域左上角开始向右的 x 个像素，向下的 y 个像素-- 这些是显示出来的图形的左上角坐标。

我们使用的 DrawRectangle()和 DrawEllipse()重载方法分别带 5 个参数。第一个参数是类 System.Drawing.Pen 的实例。Pen 是许多帮助绘图的支持对象中的一个，它包含如何绘制线条的信息。第一个 Pen 表示线条应是蓝色的，其宽度为 3 个像素；第二个 Pen 表示线条应是红色的，其宽度为 2 个像素。后面的 4 个参数是坐标和大小。对于矩形，它们分别表示矩形的左上角坐标(x,y)、其宽度和高度。对于椭圆，这些数值的含义是相同的，但它们是指椭圆假想的外接矩形，而不是椭圆本身。运行代码，会得到如-1 所示的图形。当然，本书不是彩页书，所以看不到颜色。

(点击查看大图) 图 33-1

这个屏幕图说明了两个问题。首先，用户可以很清楚地看到窗口客户区域的位置。这是一个白色的区域-- 该区域受到 BackColor 属性设置的影响。还要注意，矩形放在该区域的一角，因为我们指定了坐标(0,0)。其次，注意椭圆的顶部与矩形有轻度的重叠，这与代码中给出的坐标有点不同，这是因为 Windows 在重叠的区域上放置了矩形和椭圆的边框。在默认情况下，Windows 会试图把图形边框所在的线条放在中心位置-- 但这并不是总能做到的，因为线条是以像素为单位来绘制的，但每个图形的边框理论上位于两个像素之间。结果，1 个像素宽的线条就会正好位于图形顶边和左边的里面，而在右边和底边的外面。这样，从严格意义上讲，相邻图形的边框就会有一个像素的重叠。我们指定的线条宽度比较大，因此重叠区域也会比较大。设置 Pen.Alignment 属性(详见 SDK 文档说明)，就可以改变默认的操作方式，但这里使用默认的操作方式就足够了。

但如果运行这个示例，就会注意到窗体的执行过程有点奇怪。如果把它放在那里，或者用鼠标在屏幕移动该窗体，它就工作正常。但如果最小化该窗体，再恢复它，绘制好的图形就不见了。如果在示例中绘制另一个窗体，使之遮挡一部分图形，情况也是这样。如果在该窗体上拖动另一个窗口，使之只遮挡一部分图形，再把该窗口拖离这个窗体，临时被挡住的部分就消失了，只剩下一半椭圆或矩形了！

这是怎么回事？其原因是，如果窗口的一部分被隐藏了，Windows 通常会立即删除与其中显示的内容相关的所有信息。这是必需的，否则存储屏幕数据的内存量就会是个天文数字。一般的计算机在运行时，视频卡设置为显示 1024×768 像素、24 位彩色模式，这表示屏幕上的每个像素占据 3 个字节，则显示整个屏幕就需要 2.25MB(本章后面会说明 24 位颜色的含义)。但是，用户常常让任务栏上有 10 个或 20 个最小化窗口。下面考虑一种最糟糕的情况：20 个窗口，每个窗口如果没有最小化，就占用整个屏幕，如果 Windows 存储了这些窗口包含的可视化信息，当用户恢复它们时，它们就会有 45MB。

目前，比较好的图形卡有 64MB 的内存，可以应付这种情况，但在几年前图形卡有 4MB 的内存就不错了。剩余的部分需要存储在计算机的主内存中。许多人仍在使用旧机器，一些人甚至还在使用 4MB 的图形卡。很显然，Windows 不可能这样管理用户界面。

在窗口的某一部分消失时，那些像素也就丢失了。因为 Windows 释放了保存这些像素的内存。但要注意，窗口的一部分被隐藏了，当它检测到窗口不再被隐藏时，就请求拥有该窗口的应用程序重新绘制其内容。这个规则有一些例外——窗口的一小部分被挡住的时间比较短（例如，从主菜单中选择一个项目，该菜单项向下拉出，临时挡住了下面的窗口）。但一般情况下，如果窗口的一部分被挡住，应用程序就需要在以后重新绘制它。

这就是示例应用程序的一个问题。我们把绘图代码放在 Form1 的构造函数中，当应用程序启动时，就调用该函数一次，不能在以后需要时再次调用该构造函数，重新绘制图形。

在使用 Windows 窗体的服务器控件时，不需要知道这些，这是因为标准控件非常专业，能在 Windows 需要时重新绘制它们自己。这是编写控件时不需要担心实际绘图过程的原因之一。如果要应用程序在屏幕上绘图，还需要在 Windows 要求重新绘制窗口的全部或部分时，确保应用程序会正确响应。下一节将修改这个示例，完成应用程序的响应。

### 33.1.3 使用 OnPaint()绘制图形

上面的解释让您觉得绘制自己的用户界面是比较复杂的，实际上并非如此。让应用程序在需要时绘制自身是非常简单的。

Windows 会利用 Paint 事件通知应用程序完成一些重新绘制的要求。有趣的是，Form 类已经执行了这个事件的处理程序，因此不需要再添加处理程序了。Paint 事件的 Form1 处理程序处理虚方法 OnPaint()的调用，并给它传送一个参数 PaintEventArgs，这表示，我们只需重写 OnPaint()执行画图操作。

我们选择重写 OnPaint()，也可以为 Paint 事件添加自己的事件处理程序（例如 Form1\_Paint 方法）来得到相同的结果，其方式与为其他 Windows Form 事件添加处理程序一样。后一个方法更方便一些，因为可以通过 Visual Studio 2008 属性窗口添加新的事件处理程序，不必键入某些代码。但是我们采用重写 OnPaint()的方式要略为灵活一些，因为这样可以控制何时调用基类窗口进行处理，在文档说明中推荐采用这种方式。

下面创建一个 Windows 应用程序 DrawShapes 来完成这个操作。与以前一样，使用属性窗口把背景色设置为白色，再把窗体的文本改为 DrawShapes Sample，接着在 Form1 类中添加如下代码：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
```

```
}
```

注意，OnPaint()声明为 protected。OnPaint()一般在类的内部使用，所以类外部的其他代码不知道存在 OnPaint()。

PaintEventArgs 是一个派生自 EventArgs 的类，一般用于传送有关事件的信息。PaintEventArgs 有另外两个属性，其中比较重要的是 Graphics 实例，它们主要用于优化绘制窗口中需要绘制的部分。这样就不必调用 CreateGraphics()，在 OnPaint() 方法中获取设备环境了——用户总是可以得到设备环境。后面将介绍其他属性，它包含哪些窗口部分需要重新绘制的详细信息。

在 OnPaint() 的执行代码中，首先从 PaintEventArgs 中引用 Graphics 对象，再像以前那样绘制图形。最后调用基类的 OnPaint() 方法，这一步是非常重要的。我们重写了 OnPaint() 方法，完成了绘图工作，但 Windows 在绘图过程中可能会执行一些它自己的工作。这些工作都在.NET 基类的 OnPaint() 方法中完成。

注意：

对于这个示例，删除 base.OnPaint() 的调用似乎并没有任何影响，但不要试图删除这个调用。这样有可能阻止 Windows 正确执行任务，结果是无法预料的。

在应用程序第一次启动，窗口第一次显示出来时，也调用了 OnPaint()，所以不需要在构造函数中复制绘图代码。

运行这段代码，得到的结果将与前面的示例的结果相同，但现在，当最小化窗口或隐藏它的一部分时，应用程序会正确执行。

### 33.1.4 使用剪切区域

上一节的 DrawShapes 示例说明了在窗口中绘图的主要规则，但它并不是很高效。原因是它试图绘制窗口中的所有内容，而没有考虑需要绘制多少内容。如图 33-2 所示，运行 DrawShapes 示例，当该示例在屏幕上绘图时，打开另一个窗口，把它移动到 DrawShapes 窗体上，使之隐藏一部分窗体。

但移动上面的窗口时，DrawShapes 窗口会再次全部显示出来，Windows 通常会给窗体发送一个 Paint 事件，要求它重新绘制本身。矩形和椭圆都位于客户区域的左上角，所以在任何时候都是可见的，在本例中不需要重新绘制这部分，而只需要重新绘制白色背景区域。但是，Windows 并不知道这一点，它认为应引发 Paint 事件，调用 OnPaint() 方法的执行代码。OnPaint() 不必重新绘制矩形和椭圆。

(点击查看大图) 图 33-2

在本例中，没有重新绘制图形。原因是我们使用了设备环境。Windows 将利用重新绘制某些区域所需要的信息预先初始化设备环境。在 GDI 中，被标记出来的重绘区域称为无效区域，但在 GDI+ 中，该术语改为剪切区域，设备环境知道这个区域的内容，它截取在这个区域外部的绘图操作，且不把相关的绘图命令传送给图形卡。这听起来不错，但仍有一个潜在的性能损失。在确定是在无效区域外部绘图前，我们不知道必须进行多少设备环境处理。在某些情况下，要处理的任务比较多，因为计算哪

些像素需要改变为什么颜色，将会占用许多处理器时间(好的图形卡会提供硬件加速，对此有一定的帮助)。

其底线是让 Graphics 实例完成在无效区域外部的绘图工作，肯定会浪费处理器时间，减慢应用程序的运行。在设计优良的应用程序中，代码将执行一些检查，以查看需要进行哪些绘图工作，然后调用相关的 Graphics 实例方法。本节将编写一个新示例 DrawShapesWithClipping，修改 DisplayShapes 示例，只完成需要的重新绘制工作。在 OnPaint()代码中，进行一个简单的测试，看看无效区域是否与需要绘制的区域重叠，如果是，就调用绘图方法。

首先，需要获得剪切区域的信息。这需要使用 PaintEventArgs 的另一个属性。这个属性叫做 ClipRectangle，包含要重绘区域的坐标，并包装在一个结构实例 System.Drawing.Rectangle 中。Rectangle 是一个相当简单的结构，包含 4 个属性：Top、Bottom、Left 和 Right。它们分别包含矩形的上下的垂直坐标、左右的水平坐标。

接着，需要确定进行什么测试，以决定是否进行绘制。这里进行一个简单的测试。注意，在绘图过程中，矩形和椭圆完全包含在(0,0)到(80,130)的矩形客户区域中，实际上，点(82,132)就已经在安全区域中了，因为线条大约偏离这个区域一个像素。所以我们要看看剪切区域的左上角是否在这个矩形区域内。如果是，就重新绘制，如果不是，就不必麻烦了。

下面是代码：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;

    if (e.ClipRectangle.Top < 132 &&
        e.ClipRectangle.Left < 82)
    {
        Pen bluePen = new Pen(Color.Blue, 3);
        dc.DrawRectangle(bluePen, 0, 0, 50, 50);
        Pen redPen = new Pen(Color.Red, 2);
        dc.DrawEllipse(redPen, 0, 50, 80, 60);
    }
}
```

注意，显示的结果与以前显示的结果完全相同-- 但这次进行了早期测试，确定了哪些区域不需要绘制，提高了性能。还要注意这个是否进行绘图的测试是非常粗略的。还可以进行更精细的测试，确定矩形或者椭圆是否要重新绘制。这里有一个平衡。可以在 OnPaint()中进行更复杂的测试，以提高性能，也可以使 OnPaint()代码复杂一些。进行一些测试总是值得的，因为编写一些代码，可以更多地了解 Graphics 实例之外的绘制内容，Graphics 实例只是盲目地执行绘图命令。

### 33.2 测量坐标和区域

在上一个示例中，我们遇到了基本结构 Rectangle，它用于表示矩形的坐标。GDI+使用几个类似的结构来表示坐标或区域。下面介绍几个结构，它们都是在 System.Drawing 命名空间中定义的，如表 33-2 所示。

表 33-2

结 构	主要的公共属性
Point 和 PointF	X, Y
Size 和 SizeF	Width, Height
Rectangle 和 RectangleF	Left, Right, Top, Bottom, Width, Height, X, Y, Location, Size

注意，其中的许多对象都有许多其他属性、方法或运算符重载，这里没有列出来。本节只讨论最重要的成员。

### 33.2.1 Point 和 PointF 结构

从概念上讲，Point 在这些结构中是最简单的，在数学上，它等价于一个二维矢量，包含两个公共整型属性，表示它与某个特定位置的水平和垂直距离(在屏幕上)，如图 33-3 所示。

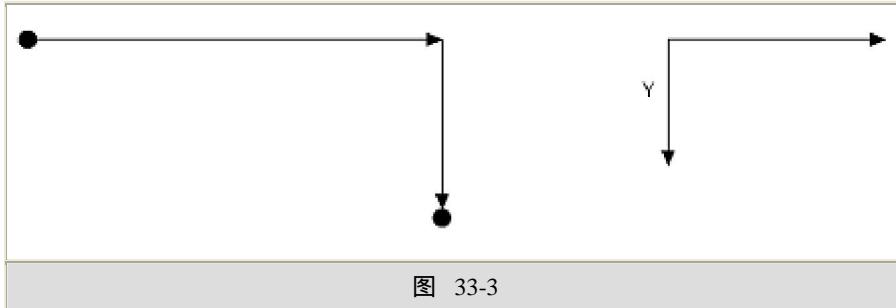


图 33-3

为了从点 A 到点 B，需要水平移动 20 个单位，并向下垂直移动 10 个单位，在图中标为 x 和 y，这就是它们的一般含义。我们可以创建一个 Point 结构，表示它们：

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X,
ab.Y);
```

X 和 Y 都是读写属性，因此可以在 Point 中设置这些值：

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X,
ab.Y);
```

注意，按照惯例，水平和垂直坐标表示为 x 和 y(小写)，但相应的 Point 属性是 X 和 Y(大写)，因为在 C# 中，公共属性的一般约定是名称以大写字母开头。

PointF 与 Point 完全相同，但 X 和 Y 属性的类型是 float，而不是 int。PointF 用于坐标不是整数值的情况。已经为这些结构定义了数据类型转换，这样就可以把 Point 隐式转换为 PointF，(注意，因

为 Point 和 PointF 是结构，这种转换实际上涉及到数据的复制)。但没有相应的逆过程，要把 PointF 转换为 Point，必须显式地复制值，或使用下面的 3 个转换方法 Round()、Truncate() 和 Ceiling()：

```
PointF abFloat = new PointF(33.5F, 10.9F);

// converting to Point
Point ab = new Point();
ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);

// but conversion back to PointF is implicit
PointF abFloat2 = ab;
```

下面看看测量单位。在默认情况下，GDI+把单位看作是屏幕(或打印机，无论图形设备是什么，都可以这样认为)上的像素，这就是 Graphics 对象方法把它们接收到的坐标看作其参数的方式。例如，点 new Point(20,10) 表示在屏幕上水平移动 20 个像素，向下垂直移动 10 个像素。通常这些像素从窗口客户区域的左上角开始测量，如上面的示例所示。但是，情况并不总是如此。例如，在某些情况下，需要以窗口的左上角(包括其边框)为原点来绘图，甚至以屏幕的左上角为原点。但在大多数情况下，除非文档说明书说明，否则都可以假定像素值是相对于客户区域的左上角。

在分析了滚动后，本章将在讨论 3 个坐标系统(世界、页面和设备坐标)时介绍这个主题。

### 33.2.2 Size 和 SizeF 结构

与 Point 和 PointF 一样，Size 也有两个变体。Size 结构用于 int 类型，SizeF 用于 float 类型，除此之外，Size 和 SizeF 是完全相同的。下面主要讨论 Size 结构。

在许多情况下，Size 结构与 Point 结构是相同的，它也有两个整型属性，表示水平和垂直距离-- 主要区别是这两个属性的名称不是 X 和 Y，而是 Width 和 Height。前面的图 33-3 可以表示为：

```
Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width,
ab.Height);
```

严格地讲，Size 在数学上与 Point 表示的含义相同；但在概念上它使用的方式略有不同。Point 用于说明实体在什么地方，而 Size 用于说明实体有多大。但是，Size 和 Point 是紧密相关的，目前甚至支持它们之间的显式转换：

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

例如，考虑前面绘制的矩形，其左上角的坐标是(0,0)，大小是(50,50)。这个矩形的大小是(50,50)，可以用一个 Size 实例来表示。其右下角的坐标也是(50,50)，但它由一个 Point 实例来表示。要理解这个区别，假定在另一个位置绘制该矩形，其左上角的坐标是(10,10)：

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

现在其右下角的坐标是(60,60)，但大小不变，仍是(50,50)。

Point 和 Size 结构的相加运算符都已经重载了，所以可以把一个 Size 加到 Point 结构上，得到另一个 Point 结构：

```
static void Main(string[] args)
{
    Point topLeft = new Point(10,10);
    Size rectangleSize = new Size(50,50);
    Point bottomRight = topLeft + rectangleSize;
    Console.WriteLine("topLeft = " + topLeft);
    Console.WriteLine("bottomRight = " + bottomRight);
    Console.WriteLine("Size = " + rectangleSize);
}
```

把这段代码作为控制台应用程序 PointAndSizes 来运行，会得到如图 33-4 所示的结果。

(点击查看大图) 图 33-4

注意，这个结果也说明了 Point 和 Size 的 ToString()方法已被重写，并以{X,Y}格式显示值。

也可以从一个 Point 减去某个 Size，得到另一个 Point，还可以把两个 Size 加在一起，得到另一个 Size。但不能把一个 Point 加到另一个 Point 上。Microsoft 认为 Point 相加在概念上没有意义，所以不支持加(+)运算符的任何重载版本进行这样的操作。

还可以在 Point 和 Size 之间进行显式的数据类型转换：

```
Point topLeft = new Point(10,10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;
```

在进行这样的数据类型转换时，s1.Width 被赋予 topLeft.X 的值，s1.Height 被赋予 topLeft.Y 的值，因此 s1 包含(10,10)。p1 最终的值与 topLeft 的值相同。

### 33.2.3 Rectangle 和 RectangleF 结构

这两个结构表示一个矩形区域(通常在屏幕上)。与 Point 和 Size 一样，这里只介绍 Rectangle 结构，RectangleF 与 Rectangle 基本相同，但它的属性类型是 float，而 Rectangle 的属性类型是 int。

Rectangle 可以看作由一个 Point 和一个 Size 组成 , 其中 Point 表示矩形的左上角 , Size 表示其大小。它的一个构造函数把 Point 和 Size 作为其参数。下面重新编写前面 DrawShapes 示例的代码 , 绘制一个矩形 :

```
Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);
```

这段代码也使用 Graphics.DrawRectangle() 的另一个重载方法 , 它的参数是 Pen 和 Rectangle 结构。

通过按顺序提供矩形的左上角水平和垂直坐标 , 宽度和高度(它们都是数字) , 可以构造一个 Rectangle :

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50)
```

Rectangle 包含几个读写属性 , 如表 33-3 所示 , 可以用不同的属性组合来设置或提取它的维数。

表 33-3

属 性	说 明
int Left	左边界的 x 坐标
int Right	右边界的 x 坐标
int Top	顶边的 y 坐标
int Bottom	底边的 y 坐标
int X	与 Left 相同
int Y	与 Top 相同
int Width	矩形的宽度
int Height	矩形的高度
Point Location	左上角
Size Size	矩形的大小

注意 , 这些属性都不是独立的 , 例如 , 设置 Width 会影响 Right 的值。

#### 33.2.4 Region

Region 表示屏幕上一个有复杂图形的区域。例如 , 图 33-5 中的阴影区域就可以用 Region 表示。

可以想象 , 初始化 Region 实例的过程相当复杂。从广义上看 , 可以指定哪些简单的图形组成这个区域 , 或指定绘制这个区域边界的路径。如果需要处理这样的区域 , 就应掌握 SDK 文档中的 Region 类。



图 33-5

### 33.3 调试须知

下面准备进行一些更高级的绘图工作。但首先介绍几个调试问题。如果在本章的示例中设置了断点，就会注意到调试图形例程不像调试其他程序那样简单。这是因为进入和退出调试程序常常会把 Paint 信息传送给应用程序。结果是在 OnPaint 重载方法上设置的断点会让应用程序一遍又一遍地绘制它本身，这样应用程序就不能完成任何工作。

这是很典型的一种情形。要明白为什么应用程序没有正确显示，可以在 OnPaint 上设置断点。应用程序会像期望的那样，遇到断点后，就会进入调试程序，此时在前景上会显示开发环境 MDI 窗口。如果把开发环境设置为满屏显示，以便更易于观察所有的调试信息，就会完全隐藏目前正在调试的应用程序。

接着，检查某些变量的值，希望找出某些有用的信息。然后按下 F5，告诉应用程序继续执行，完成某些处理后，看看应用程序在显示其他内容时会发生什么。但首先发生的是应用程序显示在前景中，Windows 检测到窗体再次可见，并提示给它发送了一个 Paint 事件。当然这表示程序遇到了断点。如果这就是我们希望的结果，那就很好。但更常见的是，我们希望以后在应用程序绘制了某些有趣的内容后再遇到断点，例如在选择某些菜单项，读取一个文件或者以其他方式改变显示的内容之后再遇到断点。这听起来就是我们需要的结果。我们根本没有在 OnPaint 中设置断点，应用程序也不会显示它在最初的启动窗口中显示的内容之外的其他内容。

有一种方式可以解决这个问题。如果有足够大的屏幕，最简单的方式就是平铺开发环境窗口，而不是把它设置为最大化，使之远离应用程序窗口，这样应用程序就不会被挡住了。但在大多数情况下，这并不是一个有效的解决方案，因为这样会使开发环境窗口过小（也可以使用第二个监视器）。另一个解决方案使用相同的规则，即把应用程序声明为在调试时放在最上层。方法是在 Form 类中设置属性 TopMost，这很容易在 InitializeComponent 方法中完成：

```
private void InitializeComponent()
{
    this.TopMost = true;
```

也可以在 Visual Studio 2008 的属性窗口中设置这个属性。

窗口设置为 TopMost 表示应用程序不会被其他窗口挡住（除了其他放在最上层的窗口）。它总是放在其他窗口的上面，甚至在另一个应用程序得到焦点时，也是这样。这是任务管理器的执行方式。

利用这个技巧时必须小心，因为我们不能确定 Windows 何时会决定应为某种原因引发 Paint 事件。如果在某些特殊的情况下，OnPaint 出了问题（例如，应用程序在选择某个菜单项后绘图，但此时出了问题）。最好的方式是在 OnPaint 中编写一些虚拟代码，测试某些条件，这些条件只在特殊的情况下才为 true。然后在 if 块中设置断点，如下所示。

```
protected override void OnPaint( PaintEventArgs e )
{
    // Condition() evaluates to true when we want to break
    if ( Condition() == true )
    {
        int ii = 0;    // <-- SET BREAKPOINT HERE!!!
```

```
}
```

上面的这段代码是设置条件断点的一种简捷方式。

### 33.4 绘制可滚动的窗口

前面的 DrawShapes 示例运行良好，因为需要绘制的内容正好适合最初的窗口大小。本节介绍如果绘制的内容不适合窗口的大小，需要做哪些工作。

下面扩展 DrawShapes 示例，以解释滚动的概念。为了使该示例更符合实际，首先创建一个 BigShapes 示例，该示例将矩形和椭圆画大一些。此时将使用 Point、Size 和 Rectangle 结构定义绘图区域，说明如何使用它们。进行了这样的修改后，Form1 类的相关部分如下所示：

```
// member fields
private Point rectangleTopLeft = new Point(0, 0);
private Size rectangleSize = new Size(200, 200);
private Point ellipseTopLeft = new Point(50, 200);
private Size ellipseSize = new Size(200, 150);
private Pen bluePen = new Pen(Color.Blue, 3);
private Pen redPen = new Pen(Color.Red, 2);

protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
    {
        Rectangle rectangleArea =
            new Rectangle (rectangleTopLeft, rectangleSize);
        Rectangle ellipseArea =
            new Rectangle (ellipseTopLeft, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

注意，这里还把 Pen、Size 和 Point 对象变成成员字段-- 这比每次需要绘图时都创建一个新 Pen 的效率高。

运行这个示例，得到如图 33-6 所示的结果。

这里有一个问题，图形在 300×300 像素的绘图区域中放不下。

一般情况下，如果文档太大，不能完全显示，应用程序就会添加滚动条，以便用户滚动窗口，查看其中选中的部分。这是另一个区域，在该区域中如果使用标准控件建立 Windows 窗体，就让.NET 运行库和基类来处理。如果窗体上有各种控件，Form 实例一般知道这些控件在哪里，如果其窗口比

较小，Form 实例就知道需要添加滚动条。Form 实例还会自动添加滚动条，不仅如此，它还可以正确绘制用户滚动到的部分屏幕。此时，用户不需要在代码中做什么工作。但在本章中，我们要在屏幕上绘制图形，所以要帮助 Form 实例确定何时能滚动。



图 33-6

添加滚动条是很简单的。Form 仍会处理所有的操作-- 因为它不知道绘图区域有多大。在上面的 BigShapes 示例中没有滚动条的原因是，Windows 不知道它们需要滚动条。我们需要确定的是，矩形的大小从文档的左上角(或者是在进行任何滚动前的客户区域左上角)开始向下延伸，其大小应足以包含整个文档。本章把这个区域称为文档区域，在图 33-7 中可以看出，本例的文档区域应是(250, 350)像素。

使用相关的属性 Form.AutoScrollMinSize 即可确定文档的大小。因此给 InitializeComponent()方法或 Form1 构造函数添加下述代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode =
        System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
    this.AutoScrollMinSize = new Size(250, 350);
}
```

另外，AutoScrollMinSize 属性还可以用 Visual Studio 2008 属性窗口设置。注意要访问 Size 类，需要添加下面的 using 语句：

```
using System.Drawing;
```

在应用程序启动时设置最小尺寸，并保持不变，在这个应用程序中是必要的，因为我们知道屏幕区域一般有多大。在运行该应用程序时，这个"文档"是不会改变大小的。但要记住，如果应用程序显示文件内容的操作，或者执行某些改变屏幕区域的操作，就需要在其他时间设置这个属性(此时，必须手工调整代码，Visual Studio 2008 属性窗口只能在构建窗体时设置属性的初始值)。

设置 MinScrollSize 只是一个开始，仅有它是不够的。如图 33-8 所示为示例应用程序目前的外观-开始时，让屏幕正确显示图形。

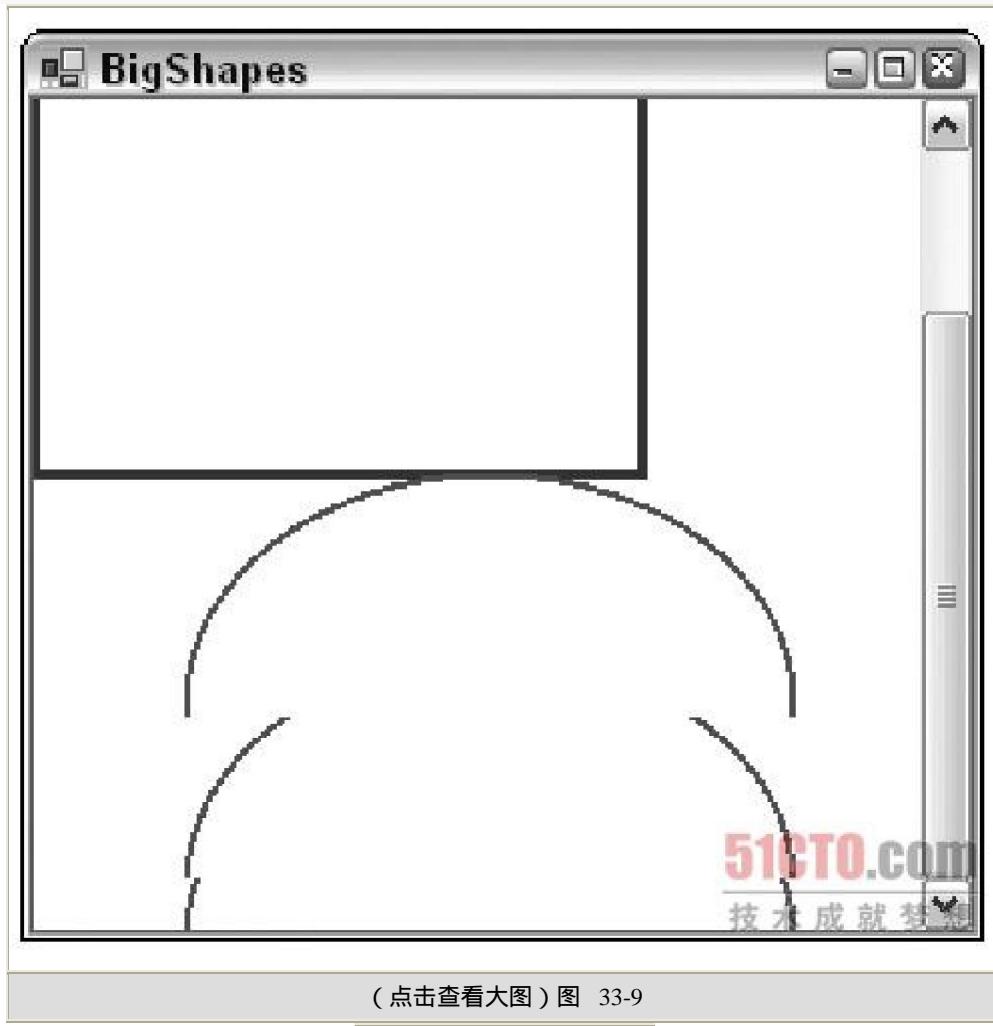
(点击查看大图) 图 33-7

(点击查看大图) 图 33-8

注意，不仅窗体正确地设置了滚动条，而且它们的大小也正确设置了，以指定文档正确显示的比例。可以试着在运行示例时重新设置窗口的大小，这样就会发现滚动条会正确响应，甚至如果使窗口变得足够大，不再需要滚动条时，滚动条就会消失。

但是，如果使用一个滚动条，并向下滚动它，会发生什么情况呢？如图 33-9 所示。显然，出现了错误！

出错的原因是我们没有在 OnPaint()重写方法的代码中考虑滚动条的位置。如果最小化窗口，再恢复它，重新绘制一遍窗口，就可以很清楚地看出这一点。结果如图 33-10 所示。



(点击查看大图) 图 33-9

(点击查看大图) 图 33-10

图形像以前一样进行了绘制，矩形的左上角嵌套在客户区域的左上角，就好像根本没有移动过滚动条一样。

在更正这个问题前，先介绍一下在这些屏幕图上发生了什么。

首先从 BigShapes 示例开始，如图 33-8 所示。在这个例子中，整个窗口刚刚重新进行了绘制。看看前面的代码，该代码的作用是使 graphics 实例用左上角坐标(0,0)(相对于窗口客户区域的左上角)绘制一个矩形——它是已经绘制过的。问题是，graphics 实例在默认情况下把坐标解释为是相对于客户窗口的，它不知道滚动条的存在。代码还没有尝试为滚动条的位置调整坐标。椭圆也是这样。

下面处理图 33-9 的问题。在滚动后，注意窗口上半部分显示正确，这是因为它们是在应用程序第一次启动时绘制的。在滚动窗口时，Windows 没有要求应用程序重新绘制已经显示在屏幕中的内容。Windows 只指出屏幕上目前显示的内容可以平滑地移动，以匹配滚动条的位置。这是一个非常高效的过程，因为它也能使用某些硬件加速来完成。在这个屏幕图中，有错误的是窗口下部的 1/3 部分。在应用程序第一次显示时，没有绘制这部分窗口，因为在滚动窗口前，这部分在客户区域的外部。这表示 Windows 要求 BigShapes 应用程序绘制这个区域。它引发 Paint 事件，把这个区域作为剪切的矩形。这也是 OnPaint() 重载方法完成的任务。

问题的另一种表达方式是我们把坐标表示为相对于文档开头的左上角-- 需要转换它们，使之相对于客户区域的左上角。图 33-11 说明了这一点。

为了使该图更清晰，我们向下向右扩展了该文档，超出了屏幕的边界，但这不会改变我们的推论，我们还假定其上还有一个水平滚动条和一个垂直滚动条。

在该图中，细矩形标记了屏幕区域的边框和整个文档的边框。粗线条标记了要绘制的矩形和椭圆。P 标记要绘制的某个随意点，这个点在后面会作为一个示例。在调用绘图方法时，提供 graphics 实例和从 B 点到 P 点的矢量，这个矢量表示为一个 Point 实例。我们实际上需要给出从点 A 到点 P 的矢量。



图 33-11

问题是，我们不知道从 A 点到 P 点的矢量。而知道从 B 点到 P 点的矢量，这是 P 相对于文档左上角的坐标-- 我们要在文档的 P 点绘图。还知道从 B 点到 A 点的矢量，这是滚动的距离，它存储在 Form 类的一个属性 AutoScrollPosition 中。但是不知道从 A 点到 P 点的矢量。

要解决这个问题，只需进行矢量相减即可。例如，要从 B 点到 P 点，可以水平移动 150 个像素，垂直向下移动 200 个像素。而要从 B 点到 A 点，就需要水平移动 10 个像素，垂直向下移动 57 个像素。这表示，要从 A 点到 P 点，需要水平移动 140 个像素( $=150-10$ )，垂直向下移动 143 个像素( $=200-57$ )。为了使之更简单，Graphics 类执行了一个方法来进行这些计算，这个方法是 TranslateTransform，我们给它传送水平和垂直坐标，表示客户区域的左上角相对于文档的左上角(AutoScrollPosition 属性，它是图中从 B 到 A 的矢量)，然后 Graphics 设备考虑客户区域相对于文档区域的位置，计算出这些坐标。

解释完之后，所需做的是把下面这行代码添加到绘图代码中：

```
dc.TranslateTransform(this.AutoScrollPosition.X,  
this.AutoScrollPosition.Y);
```

但在本示例中，它有点复杂，因为我们还要测试剪切区域，看看是否需要进行绘制工作。这个测试需要调整，把滚动的位置也考虑在内。完成后，该示例的整个绘图代码如下所示：

```
protected override void OnPaint( PaintEventArgs e )  
{  
    base.OnPaint(e);  
    Graphics dc = e.Graphics;
```

```
Size scrollOffset = new Size(this.AutoScrollPosition);

        if (e.ClipRectangle.Top+scrollOffset.Width <
350 ||
e.ClipRectangle.Left+scrollOffset.Height < 250)
{
    Rectangle rectangleArea = new Rectangle
(rectangleTopLeft+scrollOffset, rectangleSize);
    Rectangle ellipseArea = new Rectangle
(ellipseTopLeft+scrollOffset, ellipseSize);
    dc.DrawRectangle(bluePen, rectangleArea);
    dc.DrawEllipse(redPen, ellipseArea);
}
}
```

现在，滚动代码工作正常，最后得到正确的滚动屏幕图，如图 33-12 所示。



图 33-12

### 33.5 世界、页面和设备坐标

测量相对于文档区域左上角的位置和测量相对于屏幕(桌面)左上角的位置之间的区别非常重要，GDI+为它们指定了不同的名称：

世界坐标(World Coordinate)：要测量的点距离文档区域左上角的位置(以像素为单位)。

页面坐标(Page Coordinate)：要测量的点距离客户区域左上角的位置(以像素为单位)。

注意：

熟悉 GDI 的开发人员要注意，世界坐标对应于 GDI 中的逻辑坐标。页面坐标对应于设备坐标。还要注意，编写逻辑和设备坐标之间的转换代码在 GDI+中有了变化。在 GDI 中，转换是使用 Windows API 函数 LPtodP() 和 DPtoLP() 通过设备环境进行的，而在 GDI+中，由 Control 类来维护转换过程中所需要的信息，Form 和各种 Windows 窗体控件设备派生于 Control 类。

GDI+还有第 3 种坐标，即设备坐标(Device Coordinate)。设备坐标类似于页面坐标，但其测量单位不是像素，而是用户通过调用 Graphics.GraphicsUnit 属性指定的单位。它可以使用的单位除了默认的像素外，还包括英寸和毫米。本章虽然没有使用 PageUnit 属性，但它可用作获取设备的不同像素密度的方式。例如，在大多数监视器上，100 像素大约是 1 英寸。但是，激光打印机可以达到 1 200 dpi(点/英寸)-- 这表示一个 100 像素宽的图形在该激光打印机上打印时会比较小。把单位设置为英寸，指定图形为 1 英寸宽，就可以确保图形在不同的设备上有相同的大小。

```
Graphics dc = this.CreateGraphics();
dc.GraphicsUnit = GraphicsUnit.Inch;
```

GraphicsUnit 枚举中的可用值如下：

Display : 指定显示的测量单位

Document : 把文档单位(1/300 英寸)定义为测量单位

Inch : 把英寸定义为测量单位

Millimeter : 把毫米定义为测量单位

Pixel : 把像素定义为测量单位

Point : 把打印机的点数(1/72 英寸)定义为测量单位

World : 把世界坐标系统定义为测量单位

### 33.6 颜色

本节介绍如何在绘制图形时指定使用的颜色。

在 GDI+中，颜色用 System.Drawing.Color 结构的实例来表示。一般情况下，初始化这个结构后，就不能使用对应的 Color 实例对该结构进行操作了-- 只能把它传送给其他需要 Color 的调用方法。前面我们遇到过这种结构，在前面的每个示例中都设置了窗口客户区域的背景色，还设置了要显示的各种图形的颜色。Form.BackColor 属性返回一个 Color 实例。本节将详细介绍这个结构，特别是要介绍构建 Color 的几种不同方式。

#### 33.6.1 红绿蓝(RGB)值

监视器可以显示的颜色总数非常大-- 超过 160 万。其确切的数字是 2 的 24 次方 即 16 777 216。显然，我们需要对这些颜色进行索引，才能指定在给定的某个像素上要显示什么颜色。

给颜色进行索引的最常见方式是把它们分为红绿蓝成分，这种方式基于下述原则：人眼可以分辨的任何颜色都是由一定量的红色光、绿色光和蓝色光组成的。这些光称为成分(component)。实际上，如果每种成份的光分为 256 种不同的强度，它们提供了足够平滑的过渡，可以把人眼能分辨出来的图像显示为具有照片质量。因此，指定颜色时，可以给出这些成分的量，其值在 0~255 之间，其中 0 表示没有这种成分，255 表示这种成分的光达到最大的强度。

这给出了向 GDI+说明颜色的第一种方式。可以调用静态函数 Color.FromArgb()指定该颜色的红绿蓝值。Microsoft 没有为此提供构造函数，原因是除了一般的 RGB 成分外，还有其他方式表示颜色。因此，Microsoft 认为给定义的构造函数传递参数会引起误解：

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor =
Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

3 个参数分别是红绿蓝值。这个函数有许多重载方法，其中一些也允许指定 Alpha 混合值(这是方法名 FromArgb()中的 A)。Alpha 混合超出了本章的范围，但把它与屏幕上已有的颜色混合起来，可以描绘出半透明的颜色。这可以得到一些很漂亮的效果，常常用在游戏。

### 33.6.2 命名的颜色

使用 FromArgb()构造颜色是一种非常灵活的技巧，因为它表示我们可以指定人眼能辨识出的任何颜色。但是，如果要得到一种简单、标准、众所周知的纯色，例如红色或蓝色，命名想要的颜色是比较简单的。因此 Microsoft 还在 Color 中提供了许多静态属性，每个属性都返回一种命名的颜色。在下面的示例中，把窗口的背景色设置为白色时，就使用了其中一种属性：

```
this.BackColor = Color.White;  
  
// has the same effect as:  
// this.BackColor = Color.FromArgb(255, 255, 255);
```

有几百种这样的颜色。完整的列表参见 SDK 文档说明。它们包括所有的纯色：红、白、蓝、绿和黑等，还包括 MediumAquamarine、LightCoral 和 DarkOrchid 等颜色。还有一个 KnownColor 枚举，它列出了命名的颜色。

注意：

每种命名的颜色都表示一组 RGB 值，它们最初是多年前选择出来用在 Internet 上的。这种方式提供了色谱上一组有用的颜色，Web 浏览器可以辨识出这些颜色的名称-- 因此不必在 HTML 代码中显式写出它们的 RGB 值。几年前，这些颜色仍很重要，因为早期的浏览器不必准确地显示非常多的颜色，命名的颜色可以在大多数浏览器中准确地显示出来。目前它们已经不那么重要了，因为现代的 Web 浏览器能准确地显示任何 RGB 值。还有一些 Web 安全的调色板可以为开发人员提供可用于大多数浏览器的、非常丰富的颜色。

### 33.6.3 图形显示模式和安全的调色板

原则上监视器可以显示超过 160 万种 RGB 颜色，实际上这取决于如何在计算机上设置显示属性。在 Windows 中，传统上有 3 个主要的颜色选项(但有些机器还提供其他选项，这取决于硬件)：真彩色(24 位)、增强色(16 位)和 256 色。(在目前的一些图形卡上，真彩色是 32 位的，因为硬件进行了优化，但此时 32 位中只有 24 位用于该颜色)。

只有真彩色模式允许同时显示所有的 RGB 颜色。这听起来是最佳选择，但它是有代价的：完整的 RGB 值需要用 3 个字节来保存，这表示要显示的每个像素都需要用图形卡内存中的 3 个字节来保存。如果图形卡内存需要额外的费用(这种限制现在已经不像以前那样普遍了)，就可以选择其他模式。增强色模式用两个字节表示一个像素。每个 RGB 成分用 5 位就足够了。所以红色只有 32 种不同的强度，而不是 256 种。蓝色和绿色也是这样，总共有 65536 种颜色。这对于需要偶尔查看照片质量级的图像来说是足够的，但比较微妙的阴影区域会被破坏。

256 色模式给出的颜色更少。但是在这种模式下，可以选择任何颜色，系统会建立一个调色板，这是一个从 160 万 RGB 颜色中选择出来的 256 种颜色列表。在调色板中指定了颜色后，图形设备就只显示所指定的这些颜色。调色板在任何时候都可以改变-- 但图形设备每次只能在屏幕上显示 256 种不同的颜色。当获得高性能和视频内存需要额外的费用时，才使用 256 色模式。大多数计算机游戏都使用这种模式-- 它们仍能得到相当好的图形，因为调色板经过了非常仔细的选择。

一般情况下，如果显示设备使用增强色或 256 色模式，并要显示某种 RGB 颜色，它就会从能显示的颜色池中选择一种在数学上最接近的匹配颜色。因此知道颜色模式是非常重要的。如果要绘制某些涉及微妙阴影区域或照片质量级的图像，而用户没有选择 24 位颜色模式，就看不到期望的效果。如果要使用 GDI+ 进行绘制，就应该用不同的颜色模式测试应用程序(应用程序也可以编程设置给定的颜色模式，但本章不讨论这个问题)。

#### 33.6.4 安全调色板

下面简要介绍一下安全调色板。这是一种非常常见的默认调色板。它工作的方式是为每种颜色成分设置 6 个间隔相等的值，这些值分别是 0, 51, 102, 153, 204, 255。换言之，红色成分可以是这些值中的任一个。绿色成分和蓝色成分也一样。所以安全调色板中的颜色就包括(0,0,0) (黑色)、(153,0,0) (暗红色)、(0, 255, 102) (蓝绿色)等，这样就得到了  $6^3 = 216$  种颜色。这是一种让调色板包含色谱中间隔相等的颜色和所有亮度的简单方式，但实际上这是不可行的，因为数学上等间隔的颜色成分并不表示这些颜色的区别在人眼看来也是相等的。

如果把 Windows 设置为 256 色模式，默认的调色板就是安全调色板，其中添加了 20 种标准的 Windows 颜色和 20 种备用颜色。

### 33.7 画笔和钢笔

本节介绍两个辅助类，在绘制图形时需要使用它们。前面已经见过 Pen 类了，它用于告诉 graphics 实例如何绘制线条。相关的类是 System.Drawing.Brush，它告诉 graphics 实例如何填充区域。例如，Pen 用于绘制前面示例中矩形和椭圆的边框。如果需要把这些图形绘制为实心的，就要使用画笔指定如何填充它们。这两个类有一个共同点：很难对它们调用任何方法。用需要的颜色和其他属性构造一个 Pen 或 Brush 实例，再把它传送给需要 Pen 或 Brush 的绘图方法即可。

注意：

如果读者以前使用 GDI 编程，可能会注意到在前两个示例中，在 GDI+ 中使用 Pen 的方式是不同的。在 GDI 中，一般是调用一个 Windows API 函数 SelectObject()，它把钢笔关联到设备环境上。这个钢笔用于所有需要钢笔的绘图操作中，直到再次调用 SelectObject() 通知设备环境停止使用它为止。这个规则也适用于画笔和其他对象，例如字体和位图。而使用 GDI+，Microsoft 使用一种无状态的模式，其中没有默认的钢笔或其他帮助对象。只需给每个方法调用指定合适的帮助对象即可。

#### 33.7.1 画笔

GDI+有几种不同类型的画笔，本章不准备详细介绍它们，这里仅解释几个比较简单的画笔，读者掌握其要领即可。每种画笔都由一个派生自抽象类 System.Drawing.Brush 的类实例来表示。最简单的画笔 System.Drawing.SolidBrush 仅指定了区域用纯色来填充：

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
Brush solidFunnyOrangyBrownBrush =
new SolidBrush(Color.FromArgb(255,155,100));
```

另外，如果画笔是一种 Web 安全颜色，就可以用另一个类 System.Drawing.Brushes 构造出画笔。Brushes 是永远不能实例化的一个类(它有一个私有构造函数，禁止实例化)。它有许多静态属性，每个属性都返回指定颜色的画笔。可以像下面这样使用画笔：

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

比较复杂的一种画笔是影线画笔(hatch brush)，它通过绘制一种模式来填充区域，这种类型的画笔比较高级，所以在 Drawing2D 命名空间中，用 System.Drawing.Drawing2D.HatchBrush 类表示。Brushes 类不能帮助我们使用影线画笔，而需通过提供一个影线型式和两种颜色(前景色和背景色，背景色可以忽略，此时将使用默认的黑色)，来显式构造一个影线画笔。影线型式可以取自于枚举 System.Drawing.Drawing2D.HatchStyle，其中有许多 HatchStyle 值，其完整列表参阅 SDK 文档说明。为了让用户掌握这个概念，一般的型式包括 ForwardDiagonal、Cross、Diagonal Cross、SmallConfetti 和 ZigZag。构造影线画笔的示例如下所示。

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross,
Color.Azure);

// background color of CrossBrush is black

Brush brickBrush = new
HatchBrush(HatchStyle.DiagonalBrick,
Color.DarkGoldenrod, Color.Cyan);
```

GDI 只能使用实线和影线画笔，GDI+添加了两种新画笔：

System.Drawing.Drawing2D.LinearGradientBrush 用一种在屏幕上可变的颜色填充区域。

System.Drawing.Drawing2D.PathGradientBrush 与此类似，但其颜色沿着要填充的区域的路径而变化。

如果细心使用，这些画笔会得到一些惊人的效果。

### 33.7.2 钢笔

与画笔不同，钢笔只用一个类 System.Drawing.Pen 来表示。但钢笔比画笔复杂一些，因为它需要指定线条应有多宽(像素)，对于一条比较宽的线段，还要确定如何填充该线条中的区域。钢笔还可以

指定其他许多属性，本章不讨论它们，其中包括前面提到的 Alignment 属性，该属性表示相对于图形的边框，线条该如何绘制，以及在线条的末尾绘制什么图形(是否使图形光滑过渡)。

粗线条中的区域可以用纯色填充，或者使用画笔来填充。因此 Pen 实例可以包含 Brush 实例的引用。这是非常强大的，因为这表示我们可以绘制有影线填充或线性阴影的线条。构造 Pen 实例有四种不同的方式：可以传送一种颜色，或者传送一个画笔。这两个构造函数都会生成一个像素宽的钢笔。另外，还可以传送一种颜色或画笔，以及一个表示钢笔宽度的 float 类型的值。(该宽度必须是一个 float 类型的值，以允许执行绘图操作的 Graphics 对象使用非默认的单位，例如毫米或英寸，例如可以指定宽度是英寸的某个分数)。例如，可以构造如下的钢笔：

```
Brush brickBrush = new  
HatchBrush(HatchStyle.DiagonalBrick,  
Color.DarkGoldenrod, Color.Cyan);  
  
Pen solidBluePen = new Pen(Color.FromArgb(0, 0, 255));  
Pen solidWideBluePen = new Pen(Color.Blue, 4);  
Pen brickPen = new Pen(brickBrush);  
Pen brickWidePen = new Pen(brickBrush, 10);
```

另外，为了快速构造钢笔，还可以使用类 System.Drawing.Pens，它与 Brushes 类一样，包含许多存储好的钢笔。这些钢笔的宽度都是一个像素，使用通常的 Web 安全颜色，这样就可以用下述方式构建一个钢笔：

```
Pen solidYellowPen = Pens.Yellow;
```

### 33.8 绘制图形和线条

前面介绍了在屏幕上绘制规定的图形所需要的所有基类和对象。下面复习一些 Graphics 类可以使用的绘图方法，用一个小示例来介绍几个画笔和钢笔。

System.Drawing.Graphics 有很多方法，利用这些方法可以绘制各种线条、空心图形和实心图形。表 33-4 所示的列表并不完整，但给出了主要的方法，您应能据此掌握绘制各种图形的方法。

表 33-4

方 法	常 见 参 数	绘 制 的 图 形
DrawLine	钢笔、起点和终点	一段直线
DrawRectangle	钢笔、位置和大小	空心矩形
DrawEllipse	钢笔、位置和大小	空心椭圆
FillRectangle	画笔、位置和大小	实心矩形
FillEllipse	画笔、位置和大小	实心椭圆

DrawLines	钢笔、点数组	一组线，把数组中的每个点按顺序连接起来
DrawBezier	钢笔、4个点	通过两个端点的一条光滑曲线，剩余的两个点用于控制曲线的形状
DrawCurve	钢笔、点数组	通过点的一条光滑曲线
DrawArc	钢笔、矩形、两个角	由角度定义的矩形中圆的一部分
DrawClosedCurve	钢笔、点数组	与 DrawCurve 一样，但还要绘制一条用以闭合曲线的直线
DrawPie	钢笔、矩形、两个角	矩形中的空心楔形
FillPie	画笔、矩形、两个角	矩形中的实心楔形
DrawPolygon	钢笔、点数组	与 DrawLines 一样，但还要连接第一点和最后一点，以闭合绘制的图形

在结束绘制简单对象的主题前，用一个简单示例来说明使用画笔可以得到的各种可视效果。该示例是 ScrollMoreShapes，它是 ScrollShapes 的修正版本。除了矩形和椭圆外，我们还添加了一条粗线，用各种定制的画笔填充图形。前面解释了绘图的规则，所以这里只给出代码，而不作过多的注释。首先，因为添加了新画笔，所以需指定使用命名空间 System.Drawing.Drawing2D：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

接着是 Form1 类中的一些额外字段，其中包含了要绘制图形的位置信息，以及要使用的各种钢笔和画笔：

```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
new Size(200,150));
private Pen bluePen = new Pen(Color.Blue, 3);
private Pen redPen = new Pen(Color.Red, 2);
private Brush solidAzureBrush = Brushes.Azure;
private Brush solidYellowBrush = new SolidBrush(Color.Yellow);
static private Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
Color.DarkGoldenrod, Color.Cyan);
```

```
private Pen brickWidePen = new Pen(brickBrush, 10);
```

把 BrickBrush 字段声明为静态，就可以使用该字段的值初始化 BrickWidePen 字段了。C#不允许使用一个实例字段初始化另一个实例字段，因为还没有定义要先初始化哪个实例字段，如果把字段声明为静态字段就可以解决这个问题，因为只实例化了 Form1 类的实例，字段是静态字段还是实例字段就不重要了。

下面是 OnPaint()重写方法：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Point scrollOffset = this.AutoScrollPosition;
    dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);

    if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
        e.ClipRectangle.Left+scrollOffset.Y < 250)
    {
        dc.DrawRectangle(bluePen, rectangleBounds);
        dc.FillRectangle(solidYellowBrush, rectangleBounds);
        dc.DrawEllipse(redPen, ellipseBounds);
        dc.FillEllipse(solidAzureBrush, ellipseBounds);
        dc.DrawLine(brickWidePen, rectangleBounds.Location,
                   ellipseBounds.Location+ellipseBounds.Size);
    }
}
```

与以前一样，也将 AutoScrollMinSize 设置为(250,350)，结果如图 33-13 所示。



图 33-13

注意，粗对角线已在矩形和椭圆上绘制出来，它需要最后一个绘制。

### 33.9 显示图像

使用 GDI+最常想做的是显示文件中已有的图像。这确实要比绘制自己的用户界面简单多了，因为图像已经绘制好了。我们只需要加载文件，让 GDI+显示它即可。图像可以只包含一个线条或一个图标，也可以比较复杂，例如一张照片。对图像也可以执行某些操作，例如拉伸或旋转图像，也可以选择只显示图像的一部分。

本节将先给出一个示例，再讨论显示图像时需要注意的一些问题。可以这么做的原因是显示图像的代码是非常简单的。

我们需要.NET 的一个基类 System.Drawing.Image。Image 实例表示一个图像。读取图像仅需使用一行代码：

```
Image myImage = Image.FromFile("FileName");
```

FromFile()是 Image 的一个静态成员，是实例化图像的常用方式。文件可以是任何支持的图像文件格式，包括.bmp、.jpg、.gif 和.png。

显示图像是很简单的，假定有一个合适的 Graphics 实例，则调用 Graphics.DrawImage Unscaled() 或 Graphics.DrawImage()就足够了。这些方法都有许多重载方法，可以根据图像的位置和要绘制的大小非常灵活地处理用户提供的信息。下面要使用 DrawImage()：

```
dc.DrawImage(myImage, points);
```

在这行代码中，假定 dc 是一个 Graphics 实例，MyImage 是要显示的图像，points 是一个 Point 结构数组，其中 points[0]、points[1]和 points[2]是图像左上角、右上角和左下角的坐标。

注意：

熟悉 GDI 的开发人员可以从图像中看出 GDI 与 GDI+的最大区别。在 GDI 中，显示图像涉及到几个重要的步骤。如果图像是一个位图，加载它是很简单的，但如果它是其他类型的文件，加载它会涉及一系列 OLE 对象的调用。把加载的图像显示到屏幕上要获得它的一个句柄，把它放在一个内存设备环境中，然后在设备环境之间执行一个块传输。设备环境和句柄仍在后台，但如果要开始在代码中对图像进行复杂的编辑，就需要它们。简单的任务封装在 GDI+对象模型上。

下面用一个示例 DisplayImage 来说明显示图像的过程。这个示例在应用程序的主窗口中显示.jpg 文件。要使操作过程简单一些，.jpg 文件的路径硬编码到应用程序中(如果运行该示例，就需要改变该路径，以反映系统中文件的位置)。显示的.jpg 文件是圣彼得堡的日落图片。

与其他例子一样，DisplayImage 项目是 C# Visual Studio 2008 生成的一个标准的 Windows 应用程序。在 Form1 类中添加下述字段：

```
Image piccy;  
private Point [] piccyBounds;
```

然后在 Form1()构造函数中加载文件：

```
public Form1()  
{  
    InitializeComponent();  
    piccy =  
        Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.bmp");  
    this.AutoScrollMinSize = piccy.Size;  
    piccyBounds = new Point[3];  
    piccyBounds[0] = new Point(0,0); // top left  
    piccyBounds[1] = new Point(piccy.Width,0); // top right  
    piccyBounds[2] = new Point(0,piccy.Height); // bottom left  
}
```

注意，图像的大小(像素)是通过其 Size 属性来获得的，我们使用该属性设置文档区域。再建立一个 piccyBounds 数组，用于指定图像在屏幕上的位置。选择 3 个角的坐标，按实际大小来绘制图像，但如果要重新设置图像的大小、拉伸图像，或把图像变形为非矩形的平行四边形，可以改变 piccyBounds 数组中 Point 的值。

通过 OnPaint()重写方法显示图像：

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.ScaleTransform(1.0f, 1.0f);
    dc.TranslateTransform(this.AutoScrollPosition.X,
    this.AutoScrollPosition.Y);
    dc.DrawImage(piccy, piccyBounds);
}
```

最后，特别注意对向导生成的 Form1.Dispose()方法代码进行修改：

```
protected override void Dispose(bool disposing)
{
    piccy.Dispose();
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}
```

只要不再需要该图像了，就应立即删除它，因为图像在使用时一般会占用许多内存。在调用 Image.Dispose()后，Image 实例就不再引用任何图像，所以不再显示图像(除非加载了一个新图像)。

运行代码，会得到如图 33-14 所示的结果。



图 33-14

### 33.10 处理图像时的问题

显示图像是很简单的，但需要理解一些底层技术。

理解图像最重要的一点是，图像总是矩形的。这不只是方便了人们，其原因是其底层的技术。所有的现代图形卡都内置了硬件，可以非常高效地从内存的一个地方把像素块复制到另一个地方。假定像素块表示一个矩形区域，这个硬件加速操作可以虚拟为一个操作，而且执行速度非常快。实际上，

这是现代高性能图像的关键。这个操作称为位图块传输(或者 BitBlt)。Graphics.DrawImageUnscaled()在内部使用 BitBlt，这就是为什么能够看见一个大图像的原因，该图像也许包含上百万个像素，但几乎是立即就显示出来。如果计算机必须把图像一个像素一个像素地复制到屏幕上，该图像只能在几秒钟内逐渐画出来。

BitBlt 的效率非常高，所以图像的所有绘制和处理操作都是使用 BitBlt 完成的。甚至图像的某些编辑也是使用表示内存区域的设备环境之间图像的 BitBlt 完成的。在 GDI 中，Windows 32 API 函数 BitBlt()是最重要、使用最广泛的图像处理函数，而在 GDI+中，BitBlt 操作一般隐藏在 GDI+对象模型中。

图像的 BitBlt 区域不可能是矩形的，但很容易模拟类似的效果。一种方式是为了 BitBlt，把某种颜色标记为透明的。这样源图像中该颜色的区域不会覆盖目的设备中对应区域的现有颜色。在 BitBlt 过程中还可以指定，结果图像的每个像素会在进行 BitBlt 前，对源图像上和目的设备上该像素的颜色进行某些逻辑操作来形成(例如按位 AND)。这样的操作是由硬件加速来支持的，用于产生各种微妙的效果。注意 Graphics 对象执行另一个方法 DrawImage()，该方法类似于 DrawImageUnscaled()，但它有许多重载方法，可以指定 BitBlt 更复杂的形式，以便在绘图过程中使用。DrawImage()还可以只绘制图像的某个特定部分，或者对它执行其他操作，例如在绘图时缩放(缩放其大小)它。

### 33.11 绘制文本

到目前为止，本章还有一个非常重要的问题要讨论-- 显示文本。因为在屏幕上绘制文本通常比绘制简单图形更复杂。在不考虑外观的情况下，只显示一两行文本是非常简单的-- 它只需调用 Graphics 实例的一个方法 Graphics.DrawString()。但如果要显示一个文档，其中有许多文本，则事情就变得复杂多了，这有两个原因：

如果只考虑外观，则需要理解字体。图形的绘制需要使用画笔和钢笔作为帮助对象，绘制文本的过程则需要把字体作为帮助对象。理解字体不容易。

在窗口中需要仔细布局文本。用户通常期望文字一个跟一个地排列-- 排成一行，其间有一定的间隔。这是一个比想象中更困难的任务。开始时，一般事先不知道屏幕上文字之间的间隔有多大。这需要计算(使用方法 Graphics.MeasureString())。另外，屏幕上文字占用的间隔会影响文档中后续的文字在屏幕上的显示位置。如果应用程序自动换行，就需要在确定应在何处断开前仔细斟酌文字的大小。下次运行 Windows 的 Word 时，仔细看看 Word 是如何重新定位用户键入的文本的。这里有许多复杂的处理操作。任何 GDI+应用程序都不像 Word 那样复杂，但如果需要显示文本，仍需要考虑同样的问题。

总之，好的文本处理需要一定的技巧。假定知道字体和显示的位置，把一行文本显示在屏幕上的过程就非常简单。因此，下面介绍一个小示例，说明如何显示一些文本。在此之后，探讨字体和字体系列的一些规则，然后介绍一个非常真实的文本处理示例 CapsEditor。

### 33.12 简单的文本示例

这个示例 DisplayText 是常见的 Windows Forms。这次重写了 OnPaint()，添加了成员字段，如下所示：

```
private Brush blackBrush = Brushes.Black;
private Brush blueBrush = Brushes.Blue;
private Font haettenschweilerFont = new
Font("Haettenschweiler", 12);
private Font boldTimesFont = new Font("Times New Roman",
10, FontStyle.Bold);
private Font italicCourierFont = new Font("Courier", 11,
FontStyle.Italic |
FontStyle.Underline);

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.DrawString("This is a groovy string",
haettenschweilerFont, blackBrush,
10, 10);
    dc.DrawString("This is a groovy string " +
"with some very long text that will never fit in the box",
boldTimesFont, blueBrush,
new Rectangle(new Point(10, 40), new Size(100, 40)));
    dc.DrawString("This is a groovy string",
italicCourierFont, blackBrush,
new Point(10, 100));
}
```

运行这个示例，会得到如图 33-15 所示的结果。

这个示例说明了如何使用 Graphics.DrawString()方法绘制文本，DrawString()有许多重载方法，这里介绍其中的 3 个。但这些重载方法都需要用参数指定要显示的文本、字符串所使用的字体，以及用于构造各种直线和曲线以组成每个文本字符的画笔。其余的参数有另外两种指定方式。但一般情况下，可以指定一个 Point(或两个数字)或一个 Rectangle。

如果指定 Point，文本就从该 Point 的左上角开始，并向右延伸。如果指定 Rectangle，则 Graphics 实例就把字符串放在矩形的内部。如果文本在矩形内部容纳不下，就会被剪切，如图 33-15 中的第四行文本所示。把矩形传送给 DrawString()，表示绘图过程将持续较长时间，因为 DrawString()需要指定在什么地方放置换行符，但其结果应看起来好一些(如果字符串可以放在矩形中)。



图 33-15

这个示例还说明了构造字体的两种方法，一般需要字体的名称及其大小(高度)。也可以选择传送不同的样式以修改文本的绘制方式(黑体、下划线等)。

### 33.13 字体和字体系列

字体描述的是每个字母的显示方式。在一个文档中选择合适的字体和提供适当数量的不同字体，对于提高该文档的可读性是非常重要的。

大多数人在给字体命名时，会指定 Arial、Times New Roman(Windows 用户)或 Times、Helvetica(Mac OS 用户)等名称。实际上，这些都不是字体，它们是字体系列(font families)，字体系列以通俗的话来说，是文本的可视化风格。字体系列是应用程序整体外观中的一个关键因素，大多数人都习惯于识别常用字体系列的风格，甚至我们已经意识不到这一点了。

而字体应是像 Arial 9-point italic 这样的东西。换言之，字体添加了更多的信息，例如指定文本的大小，以及是否对该文本进行某种修改等，如文本是否为黑体、斜体、下划线，或者显示为小型大写字母或下标，这种修改在技术上称为风格，但在某些情况下该术语有误导作用，因为可视化外观主要取决于字体系列。

通过指定文本的高度，就可以测量其大小。高度以点为单位来测量，这是一个传统单位，表示 1/72 英寸(0.351mm)。例如，10 点的字母字体高度大约为 1/7 英寸或 3.5mm。但字体大小为 10 点的 7 行文本，在屏幕或纸上的高度不等于 1 英寸，因为还需要考虑行与行之间的间隔。

注意：

严格来讲，测量高度并不像这样简单，因为还需要考虑几个不同的高度。例如，较高字母，如 A 或 F 的高度(这是指我们讨论高度时字母的真实高度)，重音字母如? 或 ? 中的额外高度(内部前导)，以及字母的尾部超出底线的部分高度如 y 和 g(下行高度)。但是本章不讨论这些高度，一旦指定了字体系列和主要高度，这些高度就会自动确定。

在处理字体时，还会遇到其他常用于描述某种字体系列的术语：

Serif 字体系列在组成字符的许多线条尾部有一个小标记(这些标记称为 serif)。Times New Roman 就是这种字体的一个典型例子。

相反，Sans Serif 字体系列没有这些小标记。例如 Arial 和 Verdana。没有小标记常会使文本看起来比较生硬，所以 Sans Serif 字体常用于重要的文本。

True Type 字体系列以一种精确的数学方式定义组成字符的曲线形状。这表示可以使用相同的定义来计算如何在系列内部绘制任何大小的字体。目前，我们实际上使用的所有字体都是 True Type 字体。Windows 3.1 时代的一些旧字体系列是通过指定每种字体大小的每个字符位图来定义的，但现在最好不要使用这些字体。

Microsoft 提供了两个类来处理何时选择或处理字体：

System.Drawing.Font

System.Drawing.FontFamily

前面已经介绍了 Font 类的主要用法。在绘制文本时，实例化 Font 的一个实例，并把它传送给 DrawString 方法，以确定应该如何绘制文本。FontFamily 实例用于表示一个字体系列。

FontFamily 类的一个用法是：如果知道需要某种类型的字体(Serif、SansSerif 或 Monospace)，但不介意使用哪种字体，就可以使用该类。静态属性 properties GenericSerif, GenericSansSerif 和 GenericMonospace 返回满足这些条件的默认字体：

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

但一般来说，如果编写一个专业应用程序，就应以更专业的方式选择字体。可以执行绘图代码，检查哪些字体可用。然后选择合适的字体，例如从预定义的字体列表中选择第一个可用的字体。如果要让应用程序的用户友好性更高，列表中的第一个选项可能是用户上次运行软件时选择的字体。通常情况下，最好使用最常用的字体系列，例如 Arial 和 Times New Roman。但如果要使用某种不存在的字体显示文本，结果通常是不可预料的，Windows 仅是替代了标准系统字体，系统很容易绘制出文本，但看起来并不是非常友好，如果文档出现这种情况，会使人觉得软件的质量非常糟糕。

使用类 InstalledFontCollection 可以查看系统上的可用字体，该类在 System.Drawing.Text 命名空间中。这个类有一个属性 Families，该属性是一个包含系统上所有可用字体的数组：

```
InstalledFontCollection insFont = new  
InstalledFontCollection();  
FontFamily [] families = insFont.Families;  
foreach (FontFamily family in families)  
{  
  
    // do processing with this font family  
  
}
```

### 33.14 示例：枚举字体系列

下面介绍一个小示例 EnumFontFamilies，该示例列出系统上所有可用的字体系列，使用合适的字体(该字体系列的 10 点常规版本)显示它们的名称。运行这个示例，会得到如图 33-16 所示的结果。

图 33-16

当然，根据计算机上安装的字体，用户在运行这个示例时，可能会得到不同的结果。

对于这个示例，创建一个标准的 C# Windows 应用程序 EnumFont Families，首先添加要使用的另一个命名空间，我们要使用 InstalledFontCollection 类，它在 System.Drawing.Text 命名空间中定义：

```
using System.Drawing;  
using System.Drawing.Forms;  
using System.Drawing.Text;
```

然后在 Form1 类中添加如下常量：

```
private const int margin = 10;
```

margin 是文本与文档边缘之间的左边距和上边距的大小-- 它防止文本显示在客户区域边缘的右边。

因此，该示例以快捷方式显示字体系列，代码比较粗糙，在许多情况下并不是以真正应用程序中的方式执行任务。例如，我们为文档的大小硬编码了一个估计值(200,1500)，使用 Visual Studio 2008 属性窗口把 AutoScrollMinSize 属性设置为这个值。一般情况下必须查看要显示的文本，计算出文档的大小。下一节介绍这个过程。

下面是 OnPaint()方法：

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    int verticalCoordinate = margin;
    InstalledFontCollection insFont = new
    InstalledFontCollection();
    FontFamily [] families = insFont.Families;
    e.Graphics.TranslateTransform(AutoScrollPosition.X,
        AutoScrollPosition.Y);
    foreach (FontFamily family in families)
    {
        if (family.IsStyleAvailable(FontStyle.Regular))
        {
            Font f = new Font(family.Name, 12);
            Point topLeftCorner = new Point(margin,
                verticalCoordinate);
            verticalCoordinate += f.Height;
            e.Graphics.DrawString (family.Name, f,
                Brushes.Black,topLeftCorner);
            f.Dispose();
        }
    }
}
```

在这段代码中，首先使用 InstalledFontCollection 对象获得一个数组，其中包含所有可用的字体系列。对于每个系列，我们实例化大小为 12 点的一个 Font。为 Font 使用了一个简单的构造函数-- 有许多构造函数可以指定更多的选项。我们使用的构造函数带有两个参数：系列名称和字体的大小：

```
Font f = new Font(family.Name, 12);
```

这个构造函数构造了一个一般风格的字体。但是为了安全起见，在使用该字体显示文本前，先检查一下这种风格是否可用于每个字体系列，这是利用方法 `FontFamily.IsStyle Available()` 实现的，这种检查是非常重要的，因为并不是所有的字体都可以使用所有的风格：

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

FontFamily.IsStyleAvailable()带一个参数，即 FontStyle 枚举。该枚举包含许多标记，它们可以用按位 OR 运算符来组合。可能的标记有 Bold、Italic、Regular、Strikeout 和 Underline。

最后，使用 Font 类的一个属性 Height，该属性返回显示该字体文本需要的高度，以便算出行间距：

```
Font f = new Font(family.Name, 12);
Point topLeftCorner = new Point(margin,
verticalCoordinate);
verticalCoordinate += f.Height;
```

为了使事情简单化，OnPaint()的这个版本暴露出一些不太好的编程问题。首先，没有检查哪些文档区域需要绘制-- 而是显示所有的内容。另外，如前所述，实例化 Font 是一个计算密集型的过程，所以应保存字体，而不是每次调用 OnPaint()时都实例化新副本。这样设计出来的代码将需要花费相当长的时间来绘图。为了节省内存，帮助垃圾收集器，在完成操作后对每个 Font 实例调用 Dispose()。如果不这样，在 10 或 20 次绘图操作后，就会存在许多存储不再需要的字体的内存。

### 33.15 编辑文本文档：CapsEditor 示例

下面是本章中一个比较大的示例。CapsEditor 示例用于说明如何把前面介绍的绘图规则应用到真正的示例中。这个示例除了响应用户通过鼠标输入的信息外，不需要任何新资料，但它将说明如何管理文本的绘制，让应用程序具有高性能，并确保主窗口客户区域的内容总是最新的。

CapsEditor 程序允许用户读取文本文件，该文本逐行显示在客户区域中。如果用户双击任何一行文本，该行就会全部变为大写。这就是该示例的功能。即使只有这个有限的功能，也涉及到许多复杂的工作：确保把所有的信息都显示在正确的位置上，并考虑性能问题。特别是这里有一个新元素，文档的内容可以修改-- 用户选择菜单项，读取一个新文件，或用户双击一行，使之变为大写字母时，都要修改文件的内容。在第一种情况下，需要更新文档的大小，使滚动条仍能正确工作，并重新显示所有的内容。在第二种情况下，需要仔细检查文档的大小是否发生了变化，哪些文本需要重新显示。

首先看看 CapsEditor 的外观。第一次运行该应用程序时，没有加载文档，显示结果如图 33-17 所示。

图 33-17

File 菜单有两个菜单项：Open 和 Exit。Exit 退出应用程序，Open 调用 OpenFileDialog，读取用户选择的任何文件。图 33-18 是 CapsEditor 显示其源文件 Form1.cs 的情形(我们已经双击两行，把它们转换为大写)。

水平和垂直滚动条的大小也是正确的。滚动客户区域，使之正好显示整个文档。CapsEditor 不会给文本换行-- 即使没有这个功能，该示例也比较复杂了。它只显示文件被读取的各行文本。对文件的大小没有限制，但这里假定这是一个文本文件，不包含任何非打印字符。



图 33-18

首先添加一些 using 命令：

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

这是因为我们使用了 System.IO 命名空间中的 StreamReader 类。接着在 Form1 类中添加一些字段：

```
#region Constant fields
private const string standardTitle = "CapsEditor";
// default text in titlebar
private const uint margin = 10;
// horizontal and vertical margin in client area
#endregion

#region Member fields
// The 'document'
private readonly List < TextLineInformation >
    documentLines =
    new List < TextLineInformation > ();
private uint lineHeight; // height in pixels of one line
private Size documentSize; // how big a client area is
needed to
// display document
private uint nLines; // number of lines in document
private Font mainFont; // font used to display all lines
```

```
private Font emptyDocumentFont; // font used to display
empty message
private readonly Brush mainBrush = Brushes.Blue;
// brush used to display document text
private readonly Brush emptyDocumentBrush = Brushes.Red;
// brush used to display empty document message
private Point mouseDoubleClickPosition;
// location mouse is pointing to when double-clicked
private readonly OpenFileDialog fileOpenDialog = new
OpenFileDialog();
// standard open file dialog
private bool documentHasData = false;
// set to true if document has some data in it
#endregion
```

这些字段中的大多数都很容易理解。documentLines 字段是一个 List < TextLineInformation >，包含文件中已经读入的实际文本。实际上，这个字段包含了文档中的数据。documentLines 的每个元素都包含一行已读入的文本的信息。它是一个 List < TextLineInformation >，而不是常见的 C#数组，所以在读取文件时，可以给它动态地添加元素。

如前所述，每个 documentLines 元素都包含一行文本信息。这些信息实际上是另一个类 TextLineInformation 的实例：

```
class TextLineInformation
{
    public string Text;
    public uint Width;
}
```

TextLineInformation 看起来像在使用结构，而不是类，因为它只是把几个字段组合在一起。但它的实例总是作为 List < TextLineInformation >的元素来访问的，这样其元素就会存储为引用类型。

每个 TextLineInformation 实例都存储了一行文本-- 它可以看作是显示为一个项的最小单元，一般情况下，在 GDI+ 应用程序中，对于每个这样的项，都要存储该项的文本，以及显示它的世界坐标和其大小。(只要用户进行了滚动操作，页面坐标就会改变，而世界坐标通常只有在文档的其他部分以某种方式进行了修改时才会改变)。本例只需存储项的 Width 即可。其原因是此时的高度是选中字体的高度，它对于所有的文本行都一样，所以不必为每行文本存储高度。它只需要在字段 Form1.lineHeight 中存储一次即可，位置也是这样。在这里，x 坐标等于页边距，y 坐标很容易计算出来：

```
margin + lineHeight*(本行上面显示的行数)
```

如果要显示和操作单个单词，而不是整行文本，则每个单词的 x 坐标就必须使用该文本行上在该单词前面的所有单词的宽度来计算，但为了使这个计算简单一些，应把每行文本当作单一的项来看待。

下面处理主菜单。这部分应用程序是真正的 Windows 窗体(参见第 31 章),而不是 GDI+。在 Visual Studio 2008 中使用设计视图添加菜单项,把它们重新命名为 menuFile、menuFileOpen 和 menuFileExit。接着使用 Visual Studio 2008 属性窗口添加 File Open 和 File Exit 菜单项的事件处理程序。这些事件处理程序的名称是 Visual Studio 2008 生成的,即 menuFileOpen\_Click()和 menuFileExit\_Click()。

还需要在 Form1()构造函数中添加一些初始化代码:

```
public Form1()
{
    InitializeComponent();

    CreateFonts();
    fileOpenDialog.FileOk += delegate
    {
        LoadFile(fileOpenDialog.FileName);
    };
    fileOpenDialog.Filter =
    "Text files (*.txt)|*.txt|C# source files (*.cs)|*.cs";

}
```

这里添加的事件处理程序是在用户单击 FileOpen 对话框中的 OK 按钮时执行的。我们还给"打开文件"对话框设置了过滤器,只能加载文本文件。由于本示例选择了.txt 文件和 C#源文件,所以可以使用应用程序查看示例的源代码。

CreateFonts()是一个帮助方法,它挑选出要使用的字体:

```
private void CreateFonts()
{
    mainFont = new Font("Arial", 10);
    lineHeight = (uint)mainFont.Height;
    emptyDocumentFont = new Font("Verdana", 13,
    FontStyle.Bold);
}
```

处理程序的实际定义是标准的:

```
protected void OpenFileDialog_FileOk(object
    Sender, CancelEventArgs e)
{
    this.LoadFile(fileOpenDialog.FileName);
}

protected void menuFileOpen_Click(object sender,
    EventArgs e)
{
    fileOpenDialog.ShowDialog();
}
```

```
protected void menuFileExit_Click(object sender,
EventArgs e)
{
this.Close();
}
```

下面介绍 LoadFile()方法。这个方法处理文件的打开和读取操作，并确保引发 Paint 事件，使用新文件重新绘图：

```
private void LoadFile(string FileName)
{
StreamReader sr = new StreamReader(FileName);
string nextLine;
documentLines.Clear();
nLines = 0;
TextLineInformation nextLineInfo;
while ( (nextLine = sr.ReadLine()) != null)
{
nextLineInfo = new TextLineInformation();
nextLineInfo.Text = nextLine;
documentLines.Add(nextLineInfo);
++nLines;
}
sr.Close();
documentHasData = (nLines>0) ? true : false;

CalculateLineWidths();
CalculateDocumentSize();

this.Text = standardTitle + " - " + FileName;
this.Invalidate();
}
```

这个功能的大部分都是标准的文件读取过程(参见第 25 章)。注意在读取文件时，给 documentLines ArrayList 添加文本行，使这个数组最终按顺序包含每行文本的信息。在读取文件后，设置 documentHasData 标记，指定是否要显示信息。下一个任务是确定要显示内容的位置，之后确定显示文件的客户区域有多大-- 即用于设置滚动条的文档大小。最后，设置标题栏文本，并调用 Invalidate()。Invalidate()是 Microsoft 提供的一个非常重要的方法，所以下一节介绍这个方法的应用，之后介绍 CalculateLineWidths()和 CalculateDocument Size()方法的代码。

### 33.15.1 Invalidate()方法

Invalidate()是 System.Windows.Forms.Form 的一个成员，它把客户窗口区域标记为无效，因此在需要重新绘制时，它可以确保引发 Paint 事件。Invalidate()有两个重载方法：可以给它传送一个矩形，

指定(使用页面坐标)需要重新绘制哪个窗口区域 ,如果不提供任何参数 ,它就把整个客户区域标记为无效。

如果知道需要绘制某些内容 ,为什么不调用 OnPaint()或直接完成绘制任务的其他方法 ?一般情况下 ,最好不要直接调用绘图例程 ,如果代码要完成某些绘图任务 ,一般应调用 Invalidate()。其原因如下所示 :

绘图总是 GDI+应用程序执行的一种处理器密集型的任务。在其他工作的中间进行绘图会妨碍其他工作的进行。在前面的示例中 ,如果在 LoadFile()方法中直接调用一个方法来完成绘图 ,LoadFile()方法就将在绘图工作完成后才能返回。在这段时间里 ,应用程序不会响应其他事件。另一方面 ,通过调用 Invalidate() ,在从 LoadFile 返回之前 ,就可以让 Windows 引发一个 Paint 事件。接着 Windows 就可以检查等待处理的事件了。其内部的工作方式是事件被当作消息队列中一个消息。Windows 会定期检查该队列 ,如果其中有事件 ,Windows 就选择它 ,并调用相应的事件处理程序。现在 Paint 事件是队列中的唯一事件 ,所以 OnPaint()会被立即调用。但是 ,在一个比较复杂的应用程序中 ,可能会有其他事件 ,其中一些的优先权比 OnPaint()高。特别是如果用户已决定退出应用程序 ,该事件就会用消息 WM\_QUIT 来标记。

如果有一个比较复杂的多线程应用程序 ,就会希望用一个线程处理所有的绘图操作。使用 Invalidate()可以把所有的绘图操作传递到消息队列中 ,这有助于确保无论其他线程请求什么绘图操作 ,都由同一个线程完成所有的绘图操作(无论什么线程负责消息队列 ,都是由线程 Application.Run()处理绘图操作)。

还有一个与性能有关的原因。假定在某一时刻有几个不同的屏幕绘制请求 ,也许代码仅能修改文档 ,以确保显示更新的文档 ,而同时用户刚刚移开另一个覆盖部分客户区域的窗口。调用 Invalidate() ,可以让 Windows 注意到发生的事件。Windows 就会在需要时合并 Paint 事件 ,合并无效的区域 ,这样绘图操作就只执行一次。

最后 ,执行绘图的代码可能是应用程序中最复杂的代码部分 ,特别是当有一个比较专业化的用户界面时 ,就更是如此。需要长时间维护该代码的人员希望我们把所有的绘图代码都放在一个地方 ,且尽可能简单-- 如果程序的其他部分没有过多的路径进入该代码部分 ,维护就更容易。

其底线是最好把所有的绘图代码都放在 OnPaint()例程中 ,或者在该方法中调用的其他方法。但是要维持一个平衡。如果要在屏幕上替换一个字符 ,最好不要影响到已经绘制好的其他内容 ,此时可能不需要使用 Invalidate() ,而只需编写一个独立的绘图例程。

注意 :

在非常复杂的应用程序中 ,甚至可以编写一个完整的类 ,专门负责在屏幕上绘图。几年前 MFC 仍是 GDI 密集型应用程序的标准技术 ,MFC 就遵循这个模式 ,使用一个 C++类 C<ApplicationName>View 完成绘图操作。但即使是这样 ,这个类也有一个成员函数 OnDraw() ,用作大多数绘图请求的入口点。

### 33.15.2 计算项和文档的大小

下面返回 CapsEditor 示例 , 介绍在 LoadFile() 中调用的 CalculateLineWidths() 和 CalculateDocumentSize() 方法 :

```
private void CalculateLineWidths()
{
    Graphics dc = this.CreateGraphics();
    foreach (TextLineInformation nextLine in documentLines)
    {
        nextLine.Width = (uint)dc.MeasureString(nextLine.Text,
            mainFont).Width;
    }
}
```

这个方法仅遍历已经读取的每行文本 , 使用 Graphics.MeasureString() 方法处理和存储字符串需要的水平间距。存储这个值是因为 MeasureString() 要进行大量的计算。如果没有把 CapsEditor 示例编写得非常简单 , 就不能很容易地计算出每一行的高度和位置 , 而这个方法也肯定需要按计算所有项的方式来实现。

知道屏幕上每个项的大小后 , 就可以计算每个项的位置 , 并计算出文档的大小。高度基本上是每行文本的高度乘以行数。宽度则需要计算 , 遍历每行文本 , 确定哪一行最长。对于高度和宽度 , 也可以在显示的文档周围设置一个较小的页边距 , 使应用程序看起来更吸引人。

下面是计算文档大小的方法 :

```
private void CalculateDocumentSize()
{
    if (!documentHasData)
    {
        documentSize = new Size(100, 200);
    }
    else
    {
        documentSize.Height = (int)(nLines * lineHeight) +
            2 * (int)margin;
        uint maxLength = 0;
        foreach (TextLineInformation nextWord in documentLines)
        {
            uint tempLineLength = nextWord.Width + 2 * margin;
            if (tempLineLength > maxLength)
                maxLength = tempLineLength;
        }
        maxLength += 2 * margin;
        documentSize.Width = (int)maxLength;
    }
    this.AutoScrollMinSize = documentSize;
}
```

这个方法首先检查是否有数据要显示。如果没有，就使用硬编码的文档大小，该大小足以显示很大的红色<Empty Document>警告。如果要正确显示数据，就应使用 MeasureString()确定警告信息到底有多大。

计算出文档大小后，就设置 Form.AutoScaleMinSize 属性，告诉 Form 实例文档有多大。完成后，后台就会发生一些有趣的事。在设置这个属性的过程中，客户区域会标记为无效，引发 Paint 事件。改变文档的大小就意味着需要添加或修改滚动条，需要重新绘制整个客户区域。为什么说这很有趣？如果回过头来看看 LoadFile()的代码，就会发现在该方法中调用 Invalidate()是多余的。在设置文档大小时，肯定要使客户区域无效。在 LoadFile()方法中显式调用 Invalidate()，说明了在一般情况下应如何完成任务。实际上在这个示例中，再次调用 Invalidate()将重复请求 Paint 事件，这是不必要的。但是，这论证了前面论述的 Invalidate()给 Windows 一个优化性能的机会。第二个 Paint 事件实际上并不会被引发：Windows 发现队列中已经有一个 Paint 事件了，就会比较请求的无效区域，看看是否需要合并它们。在本例中，两个 Paint 事件都指定了整个客户区域，所以不需要合并，Windows 会撤消第二个 Paint 请求。当然，这个过程会占用处理器一定的时间，但与某些绘图操作所占用的时间相比，它可以忽略不计。

### 33.15.3 OnPaint()

前面介绍了 CapsEditor 如何加载文件，下面看看如何绘图：

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    int scrollPositionX = this.AutoScrollPosition.X;
    int scrollPositionY = this.AutoScrollPosition.Y;
    dc.TranslateTransform(scrollPositionX,
        scrollPositionY);

    if (!documentHasData)
    {
        dc.DrawString("<Empty document>", emptyDocumentFont,
            emptyDocumentBrush, new Point(20, 20));
        base.OnPaint(e);
        return;
    }

    // work out which lines are in clipping rectangle
    int minLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
            scrollPositionY);
    if (minLineInClipRegion == -1)
        minLineInClipRegion = 0;
    int maxLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom -
            scrollPositionY);
```

```
if (maxLineInClipRegion >= this.documentLines.Count ||  
    maxLineInClipRegion == -1)  
    maxLineInClipRegion = this.documentLines.Count - 1;  
  
    TextLineInformation nextLine;  
    for (int i = minLineInClipRegion; i <= maxLineInClipRegion; i++)  
    {  
        nextLine = (TextLineInformation)documentLines[i];  
        dc.DrawString(nextLine.Text, mainFont, mainBrush,  
                     this.LineIndexToWorldCoordinates(i));  
    }  
}
```

这个 OnPaint()重载方法的核心是一个循环，迭代文档的每行文本，它调用 Graphics.DrawString()绘制每行文本。其余的代码主要是优化绘图操作-- 通常是确定哪些部分需要绘制，而不是告诉 graphics 实例绘制所有的内容。

首先检查一下文档中是否有数据。如果没有，就显示一个消息，调用基类的 OnPaint()方法并退出。如果有数据，就查看剪切的矩形，即调用另一个方法 WorldYCoordinateTo LineIndex，后面再介绍这个方法，该方法带一个相对于文档顶部的 y 坐标，计算此时应显示文档中的哪些文本。

第一次调用方法 WorldYCoordinateToLineIndex()时，给它传送坐标值 e.ClipRectangle.Top - scrollPositionY，这是剪切区域的顶部，把它转换为世界坐标。如果返回值为-1，就假定需要从文档的开头开始(如果剪切区域位于顶部边距，就会出现这种情况)。

完成后，对剪切矩形的底部重复相同的过程，找出文档在剪切区域中的最后一行文本。第一行和最后一行的索引分别存储在 minLineInClipRegion 和 maxLineInClipRegion 中。这样就可以在这些值之间运行 for 循环，执行绘图操作了。在绘图循环中，需要通过 WorldYCoordinateToLineIndex()进行逆转换：给出一行文本的索引，要确定该行文本应绘制在什么位置。这个计算实际上是相当简单的，我们把它封装到另一个方法 LineIndexToWorldCoordinates()中，它返回该行文本左上角的坐标。返回的坐标是世界坐标，这很好，因为我们已在 Graphics 对象上调用了 TranslateTransform()，所以在要求显示文本时，需要给它传送世界坐标，而不是页面坐标。

#### 33.15.4 坐标转换

本节介绍在 CapsEditor 示例中编写的几个辅助方法，以帮助进行坐标转换。这些方法是上一节谈到的 WorldYCoordinateToLineIndex()和 LineIndexToWorldCoordinates()方法，还有几个其他的方法。

首先，LineIndexToWorldCoordinates()的参数是一个给定的行索引，它使用已知的页边距和行的高度，计算出该行左上角的世界坐标：

```
private Point LineIndexToWorldCoordinates(int index)  
{
```

```
Point TopLeftCorner = new Point  
(int)margin, (int)(lineHeight*index + margin));  
return TopLeftCorner;  
}
```

我们还使用一个方法在 OnPaint() 中进行逆转换。WorldYCoordinateToLineIndex() 可计算出行索引，但它只考虑了一个垂直的世界坐标。这是因为它是相对于剪切区域的顶部和底部来计算行索引的。

```
private int WorldYCoordinateToLineIndex(int y)  
{  
if (y < margin)  
return -1;  
return (int)((y-margin)/lineHeight);  
}
```

还有 3 个方法，它们在处理程序例程中调用，响应用户的双击鼠标操作。首先，用一个方法计算出要在给定世界坐标处显示的文本行的索引。与 WorldYCoordinateToLineIndex() 不同，这个方法考虑了 x 和 y 坐标，如果该坐标上没有文本行，它就返回 -1：

```
private int WorldCoordinatesToLineIndex(Point  
position)  
{  
if (!documentHasData)  
return -1;  
if (position.Y < margin || position.X < margin)  
return -1;  
int index =  
(int)(position.Y-margin)/(int)this.lineHeight;  
// check position isn't below document  
if (index >= documentLines.Count)  
return -1;  
// now check that horizontal position is within this line  
TextLineInformation theLine =  
(TextLineInformation)documentLines[index];  
if (position.X > margin + theLine.Width)  
return -1;  
  
// all is OK. We can return answer  
return index;  
}
```

最后，需要转换行索引和页面坐标(而不是世界坐标)，下面的方法完成这个任务：

```
private Point LineIndexToPageCoordinates(int  
index)  
{
```

```
return LineIndexToWorldCoordinates(index) +  
new Size(AutoScrollPosition);  
  
    private int PageCoordinatesToLineIndex(Point  
position)  
{  
    return WorldCoordinatesToLineIndex(position - new  
Size(AutoScrollPosition));  
}
```

注意在转换到页面坐标时，添加了 AutoScrollPosition，这是一个负值。

虽然这些方法看起来并不是很有趣，但它们说明了需要经常使用的一个技巧。在 GDI+中，系统常常会给出一些坐标(例如用户单击鼠标的坐标)，而我们需要确定在该坐标处要显示什么内容。或者相反-- 给出要显示的内容，确定它在什么地方显示。因此，如果编写一个 GDI+应用程序，就会发现编写完成坐标转换的方法是很有效的。

### 33.15.5 响应用户的输入

到目前为止，除了 CapsEditor 示例中的 File 菜单外，本章介绍的所有内容都是单向的：应用程序告诉用户一些信息，在屏幕上显示它们。当然，几乎所有的软件都是双向的：用户也可以与应用程序通信。下面就把这个功能添加到 CapsEditor 示例中。

让 GDI+应用程序响应用户输入，比编写代码在屏幕上绘图简单多了。实际上第 31 章已经介绍了如何处理用户输入。即重写 Form 类的方法，在相关的事件处理程序中调用。在引发 Paint 事件时，就是以这种方式调用 OnPaint()的。

当用户单击或移动鼠标时，可以重写的方法如表 33-5 所示。

表 33-5

方 法	何 时 调 用
OnClick(EventArgs e)	单击鼠标
OnDoubleClick(EventArgs e)	双击鼠标
OnMouseDown(MouseEventArgs e)	按下鼠标左键
OnMouseHover(MouseEventArgs e)	鼠标在移动后仍停留在某处
OnMouseMove(MouseEventArgs e)	移动鼠标
OnMouseUp (MouseEventArgs e)	释放鼠标左键

如果要检测用户什么时候键入文本，可以重写表 33-6 中的方法。

表 33-6

方 法	何 时 调 用
OnKeyDown(KeyEventArgs e)	按下一个键
OnKeyPress(KeyEventArgs e)	按下并释放一个键
OnKeyUp(KeyEventArgs e)	释放被按下的键

注意，其中的一些事件是重叠的。例如，如果用户按下鼠标按钮，就会引发 MouseDown 事件。如果按钮被立即释放，就会引发 MouseUp 事件和 Click 事件。另外，一些方法带有一个派生于 EventArgs 的参数，而不是 EventArgs 本身实例。这些派生类的实例可以给出特定事件的更多信息。MouseEventArgs 有两个属性 X 和 Y，它们给出鼠标按下时的设备坐标信息。KeyEventArgs 和 KeyPressEventArgs 的属性则指定与事件相关的键。

这就是用户响应方法。用户应考虑自己要完成的工作的逻辑。唯一要注意的是，编写 GDI+ 应用程序可能要比 Windows.Forms 应用程序做更多的逻辑工作，这是因为在 Windows.Forms 应用程序中，一般响应的是比较高级的事件(例如，文本框的 TextChanged 事件)。GDI+ 则相反，其中的事件都比较简单，用户单击鼠标，或按下键 H。应用程序的操作取决于一系列事件，而不是一个事件。例如，在 Word 中，为了选择一些文本，用户通常要单击鼠标左键，再移动鼠标，最后释放鼠标左键。应用程序接收到 MouseDown 事件，但仅有这个事件是不能完成什么任务的，只是记录下该鼠标单击时光标的位置。那么，当接收到 MouseDown 事件时，就可以检查刚才的记录，确定鼠标左键是否被按下，如果是，突出显示的文本就是用户选择的文本。当用户释放鼠标左键时，对应操作(在 OnMouseUp() 方法中)就需要检查一下是否有拖动操作，即鼠标按钮被按住并移动。只有这样，这个序列才算完成。

另外，因为某些事件有重叠，常常要选择让代码响应哪个事件。

一般规则是仔细考虑鼠标移动或单击和键盘事件的每个组合的逻辑，确保应用程序以直观的方式响应，让应用程序在各种情况下都按照期望的方式执行。我们的工作大多数是思考，而不是编码，但通过编码，可以更精细地完成任务，因为我们需要考虑用户输入的许多组合。例如，如果用户开始键入文本，而鼠标按钮处于按下状态，应用程序该如何响应？这听起来是不可能的，但迟早用户会尝试这么做的！

在 CapsEditor 示例中，为了使工作尽可能简单，并没有真正考虑任何组合的用户输入。只响应了用户的双击，此时把光标所在的那行文本变为大写字母。

这应是一个相当简单的任务，但有一个障碍，需要捕获 DoubleClick 事件。但上表说明，这个事件带一个 EventArgs 参数，不是 MouseEventArgs 参数，问题是如果要把一行文本正确标识为大写，我们需要知道用户双击时光标在什么地方，并需要一个 MouseEventArgs 参数。有两个解决办法，一个使用一个静态方法，由 Form1 对象 Control.mousePosition 执行，确定光标的位置：

```
protected override void OnDoubleClick(EventArgs e)
{
    Point MouseLocation = Control.mousePosition;
    // handle double click }
```

在大多数情况下，这个方法可以正常工作。但如果应用程序(甚至是其他一些有较高优先级的应用程序)在用户双击时进行某些计算密集型的工作，该方法就会有问题。此时要在等待大约半秒钟后，才调用 OnDoubleClick() 事件处理程序。我们不期望有这样的延迟，因为这会让用户感到很厌烦，但有时因为应用程序不能控制的原因(例如计算机较慢)，会偶尔发生这种情况。半秒的时间足够光标在屏幕上移动到其他位置了-- 此时会对完全错误的位置执行 Control.mousePosition！

比较好的方法是依赖于鼠标事件之间的重叠。双击鼠标的第一部分涉及到按下鼠标左键。这表示如果调用了 OnDoubleClick()，我们就知道 OnMouseDown()刚被调用，此时鼠标的位置不变。可以使用 OnMouseDown()重写方法记录光标的位置，为 OnDoubleClick()作准备。这就是在 CapsEditor 中采用的方法：

```
protected override void OnMouseDown(MouseEventArgs e)
{
    base.OnMouseDown(e);
    this.mousePosition = new Point(e.X, e.Y);
}
```

下面看看 OnDoubleClick()重写方法，该方法完成了许多工作：

```
protected override void OnDoubleClick(EventArgs e)
{
    int i =
    PageCoordinatesToLineIndex(this.mousePosition);
    if (i >= 0)
    {
        TextLineInformation lineToBeChanged =
        (TextLineInformation)documentLines[i];
        lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
        Graphics dc = this.CreateGraphics();
        uint newWidth = (uint)dc.MeasureString(lineToBeChanged.Text,
        mainFont).Width;
        if (newWidth > lineToBeChanged.Width)
            lineToBeChanged.Width = newWidth;
        if (newWidth + 2 * margin > this.documentSize.Width)
        {
            this.documentSize.Width = (int)newWidth;
            this.AutoScrollMinSize = this.documentSize;
        }
        Rectangle changedRectangle = new Rectangle(
        LineIndexToPageCoordinates(i),
        new Size((int)newWidth,
        (int)this.lineHeight));
        this.Invalidate(changedRectangle);
    }
    base.OnDoubleClick(e);
}
```

首先调用 PageCoordinatesToLineIndex()计算在用户双击时光标在哪行文本上。如果它返回-1，则光标没有在任何文本行上，所以什么也不做。当然，调用 OnDoubleClick()的基类版本，可以让 Windows 执行一些默认操作。

假定已经标识出了一行文本，就可以使用 `string.ToUpper()` 方法把它转换为大写。这是很容易的。比较难的部分是确定要在什么地方重新显示什么文本。幸运的是，因为这个示例非常简单，没有太多的组合。可以假定一个开头，把文本转换为大写通常要么让这行文本的宽度保持不变，要么增大其宽度。大写字母比小写字母大，所以，宽度是不会减小的。因为本例没有换行功能，所以文本行不会移动到下一行上，使其他文本向下移动。把文本行转换为大写的操作不会改变已显示的其他文本行的位置，这是一个非常大的简化！

接着代码使用 `Graphics.MeasureString()` 计算出文本的新宽度。这有两种可能性：

新宽度会使该文本更长，增大了整个文档的宽度。如果是这样，就需要把 `AutoScrollMinSize` 设置为新值，以便正确放置滚动条。

文档的大小没有变化。

在这两种情况下，都需要调用 `Invalidate()` 重新绘制屏幕。只有一行文本改变了，所以不希望重新绘制整个文档。而需要计算出仅包含被修改的文本行的矩形边界，并把这个矩形传送给 `Invalidate()`，确保只重新绘制该行文本。这就是以前代码的作用。`Invalidate()` 会在鼠标事件处理程序返回时调用 `OnPaint()`。在本章前面，曾提到在 `OnPaint()` 中设置断点的困难，如果运行示例，在 `OnPaint()` 中设置断点，捕获绘图操作，就会发现传送给 `OnPaint()` 的 `PaintEventArgs` 参数包含一个与指定矩形匹配的剪切区域。因为重载了 `OnPaint()` 方法，考虑了剪切区域，所以只重新绘制要求的文本行。

### 33.16 打印

前面主要介绍的是如何在屏幕上绘图。但有时还需要应用程序生成数据的硬拷贝。这就是本节的主题。我们将扩展 `CapsEditor` 示例，使之能进行打印预览，并打印出正在编辑的文档。

遗憾的是，本书没有详细讲述这个过程，所以这里执行的打印功能是非常简单的。通常，如果要让应用程序打印数据，要在主 `File` 菜单中添加 3 个菜单项：

`Page Setup`：允许用户选择各种选项，例如要打印哪个页面，要使用什么打印机等。

`Print Preview` 打开一个新窗体，显示打印机拷贝的模仿品。

`Print`：实际打印文档。

在本例中，为了使例子简单一些，不执行 `Page Setup` 菜单项。打印也只使用默认的设置。但要注意，如果要执行 `Page Setup`，Microsoft 已经编写了一个页面设置对话框类，以供使用。这个类是 `System.Windows.Forms.PrintDialog`。一般应编写一个事件处理程序，显示这个窗体，保存用户选择的设置。

在许多情况下，打印与在屏幕上显示是一样的：提供一个设备环境(`Graphics` 实例)，对该实例调用所有常见的显示命令。Microsoft 编写了许多类以帮助完成这个工作，我们需要的两个主要的类是 `System.Drawing.Printing.PrintDocument` 和 `System.Drawing.Printing.PrintPreview Dialog`。这两个类可以确保传送给设备环境的绘图命令能得到正确的处理，我们不必担心什么内容应打印到何处的问题。

打印/打印预览和显示在屏幕上有一些重要的区别。打印机不能滚动，但它们使用打印纸。所以需要确保找到一种合适的方式给文档分页，按要求绘制每一页。其他工作还有计算文档有多少内容可以放在一个页面上，因此计算出需要多少页，文档的每个部分应写到哪个页面上。

尽管上面描述得相当复杂，但打印过程是相当简单的。从编程的角度来看，需要采取的步骤大致如下：

打印：实例化一个 PrintDocument 对象，调用其 Print()方法。这个方法会在内部引发事件 PrintPage，发出打印第一个页面的信号。PrintPage 带一个参数 PrintPageEventArgs，它提供了页面大小和设置的信息，以及用于绘图命令的 Graphics 对象。因此应为这个事件编写一个事件处理程序，并执行该处理程序，以打印页面。这个事件处理程序还应把 PrintPageEventArgs 的布尔属性 HasMorePages 设置为 true 或 false，表示是否还有要打印的页面。PrintDocument.Print()方法将重复引发 PrintPage 事件，直到 HasMorePages 设置为 false 为止。

打印预览：此时，实例化 PrintDocument 对象和 PrintPreviewDialog 对象。使用属性 PrintPreviewDialog.Document 把 PrintDocument 关联到 PrintPreviewDialog 上，然后调用对话框的 ShowDialog()方法。这个方法会模式化地显示对话框，使之呈现为 Windows 标准打印预览窗体，显示文档的页面。在内部，则重复引发 PrintPage 事件，多次显示页面，直到 HasMorePages 属性为 false 为止。不需为此编写一个事件处理程序；而可以使用打印每个页面时使用的同一个事件处理程序，因为绘图代码在这两种情况下应是一样的(毕竟，打印预览的内容应与打印出来的版本看起来完全相同！)。

### 实现 Print 和 Print Preview

前面讨论了要执行的一般步骤，下面看看如何在代码中完成这些步骤。可以从 [www.wrox.com](http://www.wrox.com) 上下载 PrintingCapsEdit 项目，它包含 CapsEditor 项目，但进行了下述修改。

首先使用 Visual Studio 2008 设计视图在 File 菜单中添加两个新菜单项：Print 和 Print Preview。再使用属性窗口把这两个项命名为 menuFilePrint 和 menuFilePrintPreview，把它们设置为应用程序启动时是禁用的(只有打开了文档，才能进行打印！)。在主窗体的 LoadFile()方法中添加如下代码，激活这两个菜单项，LoadFile()方法负责把文件加载到 CapsEditor 应用程序中：

```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
    }
}
```

```
++nLines;
}
sr.Close();
if (nLines > 0)
{
documentHasData = true;
menuFilePrint.Enabled = true;
menuFilePrintPreview.Enabled = true;
}
else
{
documentHasData = false;
menuFilePrint.Enabled = false;
menuFilePrintPreview.Enabled = false;
}

CalculateLineWidths();
CalculateDocumentSize();

Text = standardTitle + " - " + FileName;
Invalidate();
}
```

上面突出显示的代码是添加到这个方法中的新代码。接着给 Form1 类添加一个成员字段：

```
public class Form1 : System.Windows.Forms.Form
{
private int pagesPrinted = 0;
```

这个字段用于指定目前正在打印的页面。使它成为一个成员字段，因为需要在 PrintPage 事件处理程序的调用之间记忆这个信息。

接着是用户选择 Print 或 Print Preview 菜单项时执行的事件处理程序：

```
private void menuFilePrintPreview_Click(object sender,
System.EventArgs e)
{
this.pagesPrinted = 0;
PrintPreviewDialog ppd = new PrintPreviewDialog();
PrintDocument pd = new PrintDocument();
pd.PrintPage += this.pd_PrintPage;
ppd.Document = pd;
ppd.ShowDialog();
}

private void menuFilePrint_Click(object sender,
System.EventArgs e)
```

```
{  
    this.pagesPrinted = 0;  
    PrintDocument pd = new PrintDocument();  
    pd.PrintPage += new PrintPageEventHandler  
(this.pd_PrintPage);  
    pd.Print();  
}
```

前面解释了涉及打印的主要过程，可以看出，这些事件处理程序仅执行了这个过程。在打印和打印预览中，都实例化了一个 PrintDocument 对象，并把一个事件处理程序关联到该对象的 PrintPage 事件上。对于打印，应调用 PrintDocument.Print()，而对于打印预览，则把 PrintDocument 对象关联到 PrintPreviewDialog 上，并调用打印预览对话框对象的 ShowDialog()方法。主要工作将在 PrintPage 事件的处理程序中完成。下面是该处理程序的代码：

```
private void pd_PrintPage(object sender,  
    PrintPageEventArgs e)  
{  
    float yPos = 0;  
    float leftMargin = e.MarginBounds.Left;  
    float topMargin = e.MarginBounds.Top;  
    string line = null;  
  
    // Calculate the number of lines per page.  
    int linesPerPage = (int)(e.MarginBounds.Height /  
        mainFont.GetHeight(e.Graphics));  
    int lineNo = this.pagesPrinted * linesPerPage;  
  
    // Print each line of the file.  
    int count = 0;  
    while(count < linesPerPage && lineNo < this.nLines)  
    {  
        line =  
            ((TextLineInformation)this.documentLines[lineNo]).Text;  
        yPos = topMargin + (count *  
            mainFont.GetHeight(e.Graphics));  
        e.Graphics.DrawString(line, mainFont, Brushes.Blue,  
            leftMargin, yPos, new StringFormat());  
        lineNo++;  
        count++;  
    }  
  
    // If more lines exist, print another page.  
    if(this.nLines > lineNo)  
        e.HasMorePages = true;  
    else  
        e.HasMorePages = false;
```

```
    pagesPrinted++;  
}
```

在声明了两个局部变量后，应先计算出一个页面上可以显示多少行文本——即页面的高度除以一行文本的高度，并取整。页面的高度可以从 PrintPageEventArgs.MarginBounds 属性获得，这个属性是一个 RectangleF 结构，该结构已初始化为页面的边界。一行文本的高度可以从 Form1.mainFont 字段中获得，该字段是显示文本所使用的字体。这里必须使用相同的字体进行打印。注意，对于 PrintingCapsEditor 示例，每页上的行数总是相同的，所以可以在第一次计算出该行数后，把它高速缓存起来。但是，该计算并不困难，在比较复杂的应用程序中，这个值可能会有变化，所以每次打印页面时重新计算它也是可以的。

我们还初始化了一个变量 lineNo，这是一个文档中文本行的基于 0 的索引，索引为 0 的文本行是文档中的第一行，这个信息是非常重要的，因为在原则上，可以调用 pd\_PrintPage() 方法打印任何页面，而不仅仅是打印第一页。lineNo 的值是每个页面中的行数和已打印的页面数的乘积。

接着运行一个循环，打印每一行文本。当打印完文档中的所有文本行，或者打印完本页面上的所有文本行时，这个循环就终止(只要满足这两个条件之一即可)。最后，检查是否还有要打印的文档，并据此设置 PrintPageEventArgs 的 HasMorePages 属性，递增 pagesPrinted 字段，这样下一次调用 PrintPage 事件处理程序时，就会打印出正确的页面了。

这个事件处理程序要注意的一点是不必担心绘图命令的发送目的地。我们使用所提供的 Graphics 对象和 PrintPageEventArgs。Microsoft 编写的 PrintDocument 类将在内部确保，如果进行打印，Graphics 对象就与打印机关联起来，如果进行打印预览，Graphics 对象就与屏幕上的打印预览窗体关联起来。

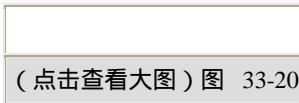
最后，需要确保为类型定义导入 System.Drawing.Printing 命名空间：

```
using System;  
using System.Collections;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Drawing.Printing;  
using System.Text;  
using System.Windows.Forms;  
using System.IO;
```

剩下的就是编译项目，并检查代码的工作情况。如果运行 CapsEdit，加载一个文本文档(与以前一样，为该项目选择 C# 源文件)，并选择 Print Preview，就可以显示出该文档的打印版本，如图 33-19 所示。

(点击查看大图) 图 33-19

在图 33-19 上，滚动到文档的第 5 页上，把预览设置为显示正常的尺寸。PrintPreviewDialog 提供了许多功能，这可以从窗体顶部的工具栏中看出来。可用的选项包括打印文档、缩小和放大文档、一次显示 2、3、4 或 6 页。这些选项的功能都是很完整的，不需要我们做任何工作。例如，如果把缩放改为自动，单击显示 4 页的选项(从右数的第 3 个工具栏)，就会得到如图 33-20 所示的结果。



### 33.17 小结

本章介绍了如何在显示设备上进行绘图操作，其中绘图操作是通过代码实现的，而不是由一些预定义的控件或对话框来实现的，这就是 GDI+的实质。GDI+是一个功能强大的工具，有许多.NET 基类都可用于在设备上绘图。绘图过程实际上是非常简单的，在大多数情况下，只使用几个 C#语句，就可以绘制文本和专业化的图形或图像。但是，管理绘图操作-- 后台的工作涉及到计算出要绘制的内容、确定在什么地方绘制，以及在任何给定的情况下，哪些内容需要重新绘制，哪些内容不需要重新绘制。这些工作都非常复杂，需要经过仔细的算法设计。因此，很好地理解 GDI+的工作方式，以及 Windows 采取什么操作来完成工作也是非常重要的。特别是，由于 Windows 的体系结构，在绘图过程中，应使窗口的区域失效，并依赖 Windows 通过引发 Paint 事件来响应。

本章并没有介绍绘图操作所涉及到的所有.NET 类，但如果理解了绘图操作的规则，就可以通过查看 SDK 文档说明中这些类的方法列表，实例化它们的实例，看看它们能做什么，以掌握它们。最后，绘图与其他编程一样，需要逻辑、仔细的思考和清晰的算法。应用它们，就能够编写出不依赖于标准控件的专业化用户界面。许多应用程序完全依赖于控件来设计其用户界面。这是很有效的，但这种应用程序看起来非常类似。通过添加一些 GDI+代码完成定制的绘图操作，可以使软件比较独特，比较新颖-- 这绝不仅仅有助于软件的销售！

下一章介绍最新的胖客户显示技术 WPF。

## 第 34 章 Windows Presentation Foundation

Windows Presentation Foundation(WPF)是.NET Framework 3.0 中的三个主要扩展之一。WPF 是为智能客户应用程序创建 UI 的一个新库。Windows 窗体控件基于 Windows 内置控件，利用了基于屏幕像素的 Windows 句柄。而 WPF 基于 DirectX。应用程序不再使用 Windows 句柄，更便于重新设置 UI 的大小，并内置了音频和视频的支持。

本章的主要论题如下：

WPF 概述

用作基本绘图元素的图形

利用转换功能实现缩放、旋转和倾斜

填充元素的不同笔刷

WPF 控件及其特性

如何用 WPF 面板定义布局

WPF 事件处理机制

样式、模板和资源

### 34.1 概述

WPF 的一个主要特性是设计人员和开发人员的工作很容易分开。设计人员的工作成果可以直接供开发人员使用。为此，必须理解 XAML。本章的第一个主题是概述 WPF，理解 XAML 的规则，讨论设计人员和开发人员如何合作。WPF 由几个包含了上千个类的程序集组成。因此用户可以在这些类中浏览，查找需要的类，大致了解 WPF 中的类层次结构和命名空间。

### 34.1.1 XAML

XML for Applications Markup Language(XAML)是一种 XML 语法 ,用于定义用户界面的层次结构。在下面的代码行中 ,声明了一个内容为 Click Me!、名为 button1 的按钮。<Button>元素指定使用 Button 类 :

```
<Button Name="button1">Click Me!</Button>
```

提示 :

XAML 元素总是有一个.NET 类。在特性和子元素中 ,可以设置属性的值 ,定义事件的处理程序方法。

为了测试简单的 XAML 代码 ,可以启动实用工具 XAMLPad.exe ( 参见图 34-1 ) ,在编辑字段中输入 XAML 代码。在 XAMLPad 中 ,可以在已准备好的<Page>和<Grid>元素中编写<Button>元素。利用 XAMLPad 可以立即查看 XAML 结果。

( 点击查看大图 ) 图 34-1

XAML 代码可以由 WPF 运行库解释 ,也可以编译为 BAML ( Binary Applications Markup Language ) ,在默认情况下 ,这是由 Visual Studio WPF 项目完成的。 BAML 添加为可执行文件的一个资源。

除了编写 XAML 之外 ,也可以用 C# 代码创建按钮。我们可以创建一个正常的 C# 控制台应用程序 ,添加对程序集 WindowsBase、PresentationCore 和 PresentationFramework 的引用 ,再编写下面的代码。在 Main() 方法中 ,从 System.Windows 命名空间中创建一个 Window 对象 ,设置 Title 属性。接着从 System.Windows.Controls 命名空间中创建一个 Button 对象 ,设置 Content 属性 ,将窗口的 Content 属性设置为该按钮。 Application 类的 Run() 方法负责处理 Windows 消息。

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.WPF
{
    class Program
    {
        [STAThread]
        static void Main()
        {
            Window mainWindow = new Window();
            mainWindow.Title = "WPF Application";
            Button button1 = new Button();
            button1.Content = "Click Me!";
            mainWindow.Content = button1;
            button1.Click += (sender, e) => MessageBox.Show("Button clicked");
        }
    }
}
```

```
Application app = new Application();
app.Run(mainWindow);
}
}
}
```

#### 提示：

Application 类也可以用 XAML 定义。在 Visual Studio WPF 项目中，打开 App.xaml 文件，它包含 Application 类的属性和 StartupUri。

运行应用程序，会得到一个包含按钮的窗口，如图 34-2 所示。

(点击查看大图) 图 34-2

可以看出，WPF 的编程非常类似于 Window 窗体的编程，其区别是按钮有 Content 属性，而不是 Text 属性。但是，与通过代码创建 UI 窗体相比，XAML 有一些非常好的优点。利用 XAML，设计人员和开发人员可以更好地合作。设计人员可以用 XAML 代码设计一个漂亮的 UI，开发人员在使用 C# 在后台代码中添加功能。使用 XAML 更便于将 UI 与功能分开。

使用后台代码和 XAML，可以在 C# 代码中直接与用 XAML 定义的元素交互操作。只需为该元素定义名称，将该名称用作变量，来修改属性，调用方法。

按钮有一个 Content 属性，而不是 Text 属性，因为按钮可以显示任意信息。可以给按钮添加文本、图形、列表框、视频-- 等任何元素。

#### 1. 将属性用作特性

在使用 XAML 之前，需要了解 XAML 语法的重要特性。使用 XML 特性可以指定类的属性。下面的例子说明了如何设置 Button 类的 Content 和 Background 属性。

```
<Button Content="Click Me!" Background="LightGreen" />
```

#### 2. 将属性用作元素

除了使用 XML 特性之外，属性也可以指定为子元素。指定 Button 元素的子元素，就可以直接设置 Content 的值。对于 Button 的其他属性，子元素名用外部元素的名称定义，之后是属性名：

```
<Button>
<Button.Background>
    LightGreen
</Button.Background>
    Click Me!
</Button>
```

在上面的例子中，不一定要使用子元素；使用 XML 特性，也可以得到相同的结果。但是，如果特性值比字符串还复杂，就不能再使用特性了。例如，背景不仅可以设置为一种简单的颜色，还可以设置为笔刷，如设置为线性渐变笔刷：

```
<Button>
<Button.Background>
<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
<GradientStop Color="Yellow" Offset="0.0" />
<GradientStop Color="Orange" Offset="0.25" />
<GradientStop Color="Red" Offset="0.75" />
<GradientStop Color="Violet" Offset="1.0" />
</LinearGradientBrush>
</Button.Background>
Click Me!
</Button>
```

### 3. 依赖属性

在用 WPF 编程时，常常会遇到“依赖属性”这个术语。WPF 元素是带有方法、属性和事件的类。WPF 元素的几乎每个属性都是依赖属性，这是什么意思？依赖属性可以依赖其他输入，例如主题和用户喜好。依赖属性与数据绑定、动画、资源和样式一起使用。

从编程的角度来看，要读写依赖属性，可以调用强类型化的属性，也可以给方法传送依赖属性对象。

只有派生自 DependencyObject 基类的类才能包含依赖属性。下面的类 MyDependencyObject 定义了依赖属性 SomeState。SomeStateProperty 是 DependencyProperty 类型的一个静态字段，它支持依赖属性。依赖属性使用 Register()方法通过 WPF 依赖属性系统来注册。Register()方法可以获取依赖属性的名称、依赖属性的类型和拥有者的类型。使用 DependencyObject 基类的 SetValue()方法，可以设置依赖属性的值。使用 GetValue()方法可以获取其值。依赖属性常常还有强类型化的访问权限。除了使用 DependencyObject 基类的方法之外，类 MyDependencyObject 还包含属性 SomeState，它从 set 和 get 存取器的实现代码中调用基类的方法。不应在 SetValue()和 GetValue()方法的执行代码中执行其他操作，因为不可能调用这些属性存取器。

```
public class MyDependencyObject : DependencyObject
{
    public static readonly DependencyProperty
    SomeStateProperty =
    DependencyProperty.Register("SomeState",
    typeof(String),
    typeof(MyDependencyObject));
    public string SomeState
    {
        get { return (string)this.GetValue(SomeStateProperty); }
        set { this.SetValue(SomeStateProperty, value); }
    }
}
```

```
}
```

```
}
```

#### 提示：

在 WPF 中，类 DependencyObject 位于层次结构的最高层。每个 WPF 元素都派生自这个基类。

#### 4. 附带属性

WPF 元素也可以从父元素中获得特性。例如，如果 Button 元素位于 Canvas 元素中，按钮的 Top 和 Left 属性把父元素的名称作为前缀。这种属性称为附带属性：

```
<Canvas>
<Button Canvas.Top="30" Canvas.Left="40">
    Click Me!
</Button>
</Canvas>
```

在后台代码中编写相同的功能有点不同，因为 Button 类没有 Canvas.Top 和 Canvas.Left 属性，但它包含在 Canvas 类中。

设置所有类都有的附带属性有一个命名模式。支持附带属性的类有静态方法 Set<Property> 和 Get<Property>，其中第一个参数是应用属性值的对象。Canvas 类定义了静态方法 SetLeft() 和 SetTop()，它们会获得与前面 XAML 代码相同的结果：

```
[STAThread]
static void Main()
{
    Window mainWindow = new Window();
    Canvas canvas = new Canvas();
    mainWindow.Content = canvas;
    Button button1 = new Button();
    canvas.Children.Add(button1);
    button1.Content = "Click Me!";
    Canvas.SetLeft(button1, 40);
    Canvas.SetTop(button1, 30);
    Application app = new Application();
    app.Run(mainWindow);
}
```

#### 提示：

附带属性可以实现为依赖对象。方法 DependencyProperty.RegisterAttached() 会注册附带属性。

#### 5. 标记扩展

在为元素设置值时，可以直接设置值，但有时标记扩展非常有帮助。标记扩展包含花括号，其后是定义了标记扩展类型的字符串标志。下面是一个 StaticResource 标记扩展：

```
<Button Name="button1" Style="{StaticResource key}">  
Content="Click Me" />
```

除了使用标记扩展之外，还可以使用子元素编写相同功能的代码：

```
<Button Name="button1">  
<Button.Style>  
<StaticResource ResourceKey="key" />  
</Button.Style>  
Click Me!  
</Button>
```

标记扩展主要用于访问资源和数据绑定，本章后面将讨论这两个主题。

### 34.1.2 设计人员和开发人员的合作

开发人员常常不仅要实现 Windows 应用程序，还负责应用程序的设计。尤其是应用程序仅用于内部使用，就更是如此。如果雇用懂得 UI 设计技巧的人来设计 UI，开发人员通常会得到设计人员的一个 JPG 文件，该文件描述了 UI 的外观。接着开发人员就要试图实现设计人员的规划。设计人员的一个简单修改，如给列表框和按钮指定不同的外观，就需要开发人员投入大量的精力设计定制的控件。因此，开发人员设计的 UI 与最初的设计大相径庭。

WPF 改变了这种情况。设计人员和开发人员可以使用相同的 XAML 代码。设计人员可以使用 Expression Blend 工具，开发人员则使用 Visual Studio 2008，但他们可以用同一个项目文件工作。在这个合作过程中，设计人员使用与 Visual Studio 中相同的项目文件在 Expression Blend 中启动一个项目，接着，开发人员编写后台代码，同时设计人员改进 UI。在开发人员改进功能的同时，设计人员还可以利用开发人员提供的功能，添加新的 UI 特性。

当然，还可以用 Visual Studio 启动应用程序，以后再用 Expression Blend 改进 UI。只是要小心，不要像处理 Windows 窗体那样设计 UI，因为这没有充分利用 WPF。

图 34-3 显示了用 WPF 创建的 Expression Blend。

(点击查看大图) 图 34-3

提示：

比较 Expression Blend 和 Visual Studio 扩展，Expression Blend 的优秀特性有定义样式、创建动画、使用图形等。为了合作，Expression Blend 可以使用开发人员编写的后台编码类，设计人员可以在 WPF 元素中指定与.NET 类的数据绑定。设计人员还可以在 Expression Blend 中启动完整的应用程序，来测试它。因为 Expression Blend 使用与 Visual Studio 相同的 MS-Build 文件，所以能编译后台编码的 C# 代码，运行应用程序。

### 34.1.3 类层次结构

WPF 包含上千个类，有很深的层次结构。为了帮助理解类之间的关系，图 34-4 在一个类图中列出了一些 WPF 类。表 34-1 描述了一些类及其功能。

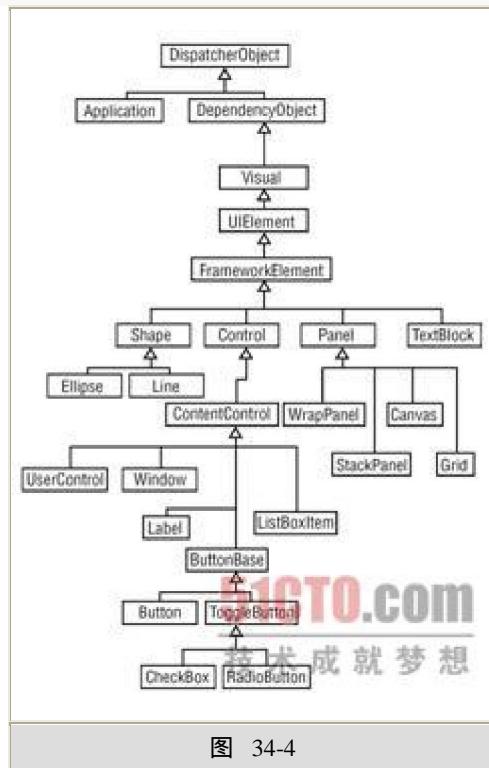


图 34-4

表 34-1

类	说 明
DispatcherObject	DispatcherObject 是一个抽象基类，用于绑定到一个线程上的类。与 Windows 窗体类似，WPF 也要求仅从创建线程中调用方法和属性。派生自 DispatcherObject 的类有一个关联的 Dispatcher 对象，它可用于切换线程
Application	在 WPF 应用程序中，会创建 Application 类的一个实例。这个类实现了 Singleton 模式，用于访问应用程序的窗口、资源和属性
DependencyObject	DependencyObject 是所有支持依赖属性的类的基类。依赖属性如前所述
Visual	所有可见元素的基类是 Visual。这个类包含点击测试和转换等特性

(续表)

类	说 明
UIElement	所有需要基本显示功能的 WPF 元素的抽象基类是 UIElement。这个类提供了鼠标移动、拖放、按键的通道和起泡事件；提供了可以由派生类重写的虚显示方法；以及布局方法。WPF 不再使用 Window 句柄，这个类就可以用作 Window 句柄
FrameworkElement	FrameworkElement 派生自基类 UIElement，实现了由基类定义的方法的默认代码
Shape	Shape 是所有图形元素的基类，例如 Line、Ellipse、Polygon、Rectangle
Control	Control 派生自 FrameworkElement，是所有用户交互元素的基类
Panel	Panel 派生自 FrameworkElement，是所有面板的抽象基类，这个类的 Children 属性用于面板中的所有 UI 元素，定义了安排子控件的方法。派生自 Panel 的类为子控件的布置方式定义了不同的类，例如 WrapPanel、StackPanel、Canvas、Grid
ContentControl	ContentControl 是所有有单个内容的控件的基类，如 Label、Button。内容控件的默认样式是受限制的，但可以使用模板改变其外观

可以看出，WPF 类有非常深的层次结构。本章和下一章将介绍提供核心功能的类，但不可能用两章的篇幅涵盖 WPF 的所有特性。

### 31.1.4 命名空间

Windows 窗体类和 WPF 类很容易混淆。Windows 窗体类位于 System.Windows.Forms 命名空间，而 WPF 类位于 System.Windows 命名空间及其子命名空间中，但不位于 System.Windows.Forms 命名空间。Windows 窗体的 Button 类的全称是 System.Windows.Forms.Button，而用于 WPF 的 Button 类的全称是 System.Windows.Controls.Button。Windows 窗体参见第 31 和 32 章。

WPF 的命名空间及其功能如表 34-2 所述。

表 34-2

命 名 空 间	说 明
System.Windows	这是 WPF 的核心命名空间，其中包含 WPF 的核心类，如 Application 类、用于依赖对象的类、DependencyObject 和 DependencyProperty，所有 WPF 元素的基类 FrameworkElement
System.Windows.Annotations	这个命名空间中的类用于用户在应用程序数据上创建的标识和记录，它们与文档分开存储。命名空间 System.Windows.Annotations.Storage 包含了存储标识的类
System.Windows.Automation	System.Windows.Automation 命名空间用于自动完成 WPF 应用程序。它有几个子命名空间。System.Windows.Automation.Peers 命名空间包含用于自动化的 WPF 元素，如 ButtonAutomationPeer 和 CheckBoxAutomationPeer。 如果创建定制的自动化提供程序，就需要 System.Windows.Automation.Provider 命名空间

(续表)

命 名 空 间	说 明
System.Windows.Controls	这个命名空间包含了所有 WPF 控件，如 Button、Border、Canvas、ComboBox、Expander、Slider、ToolTip、TreeView 等。在命名空间 System.Windows.Controls.Primitives 中，包含了在复杂控件中使用的类，如 Popup、ScrollBar、StatusBar、TabPanel 等
System.Windows.Converters	这个命名空间包含了用于数据转换的类。但它没有包含所有的转换类。核心转换类在 System.Windows 命名空间中定义
System.Windows.Data	这个命名空间由 WPF 数据绑定使用。其中的一个重要类是 Binding，它用于定义 WPF 目标元素和 CLR 源之间的绑定
System.Windows.Documents	在处理文档时，可以使用这个命名空间中的许多类。内容元素 FixedDocument 和 FlowDocument 可以包含这个命名空间中的其他元素。 System.Windows.Documents.Serialization 命名空间中的类可以将文档写入磁盘
System.Windows.Ink	Windows Tablet PC 和 Ultra Mobile PC 使用得越来越多。在这些 PC 上，ink 可以用于用户输入。System.Windows.Ink 命名空间包含处理 ink 输入的类
System.Windows.Input	这个命名空间包含的几个类用于命令处理、键盘输入、使用触针等
System.Windows.Interop	这个命名空间中的类用于集成 Win32 和 WPF
System.Windows.Markup	用于 XAML 标记代码的类位于这个命名空间
System.Windows.Media	要使用图像、音频和视频内容，可以使用这个命名空间中的类
System.Windows.Navigation	这个命名空间包含在窗口之间导航的类
System.Windows.Resources	这个命名空间包含资源的支持类
System.Windows.Shapes	UI 的核心类位于这个命名空间，如 Line、Ellipse、Rectangle 等
System.Windows.Threading	WPF 元素类似于绑定到单个线程上的 Windows 窗体控件。这个命名空间中的类

	可以处理多个线程，例如 Dispatcher 类就属于这个命名空间
System.Windows.Xps	XML Paper Specification(XPS)是一个新的文档规范，Microsoft Word 也支持该规范。在 System.Windows.Xps、System.Windows.Xps.Packaging 和 System.Windows.Xps.Serialization 命名空间中，包含了创建和传送 XPS 文档的类

## 34.2 形状

形状是 WPF 的核心元素。利用形状，可以绘制矩形、线条、椭圆、路径、多边形和多义线等二维图形，这些图形用派生自抽象类 Shape 的类表示。图形在 System.Windows.Shapes 命名空间中定义。

下面的 XAML 示例绘制了一个带腿的黄色笑脸，它用一个椭圆表示笑脸，两个椭圆表示眼睛，一个路径表示嘴，四个线条表示腿：

```
<Window x:Class="SimpleControls.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="WPF Samples" Height="260" Width="230">
<Canvas>
<Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
<Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
Stroke="Blue" StrokeThickness="3" Fill="White" />
<Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5"
Fill="Black" />
<Path Stroke="Blue" StrokeThickness="4" Data="M 62,125 Q 95,122
102,108" />
<Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
StrokeThickness="4" />
<Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue"
StrokeThickness="4" />
</Canvas>
</Window>
```

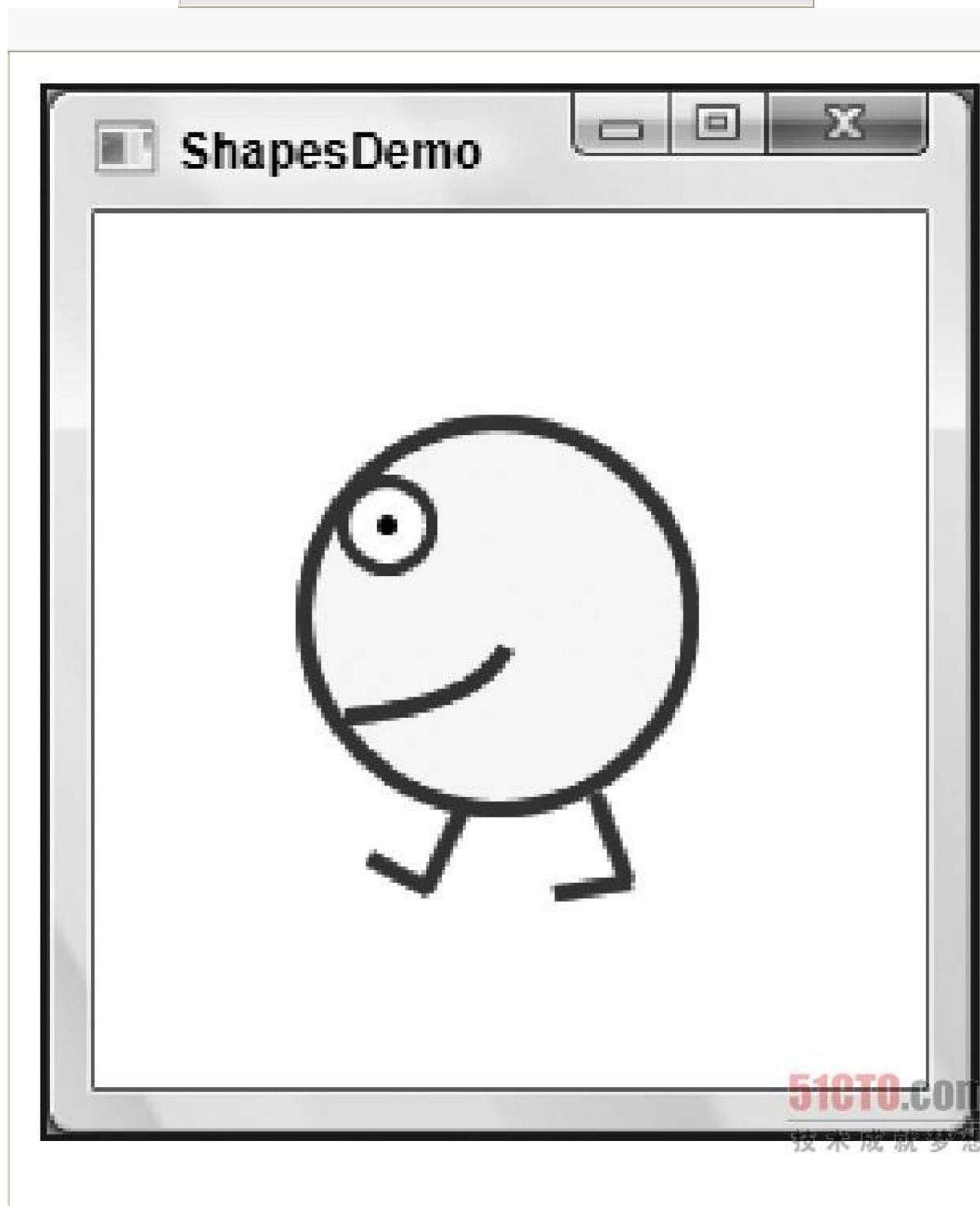
图 34-5 显示了这些 XAML 代码的结果。

无论是按钮还是线条、矩形等图形，所有这些 WPF 元素都可以通过编程来访问。把 Path 元素的 Name 属性设置为 mouth，就可以用变量名 mouth 以编程方式访问这个元素：

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
Data="M 62,125 Q 95,122 102,108" />
```

在 Path 元素的后台编码属性 Data 中 , mouth 设置为一个新的图形。为了设置路径 , Path 类支持 PathGeometry 和路径标记语法。字母 M 定义了路径的起点 , 字母 Q 指定了二次贝塞尔曲线的一个控制点和终点。运行应用程序 , 会看到如图 34-6 所示的窗口。

```
public Window1()
{
    InitializeComponent();
    mouth.Data = Geometry.Parse("M 62,125 Q 95,122 102,128");
}
```



(点击查看大图) 图 34-5



51CTO.COM  
技术成就梦想

(点击查看大图) 图 34-6

在本章的前面提到，按钮可以包含任何内容。对 XAML 代码做一点儿修改，将 Button 元素作为内容添加到窗口中，会使图形显示在按钮中，如图 34-7 所示。

(点击查看大图) 图 34-7

```
<Window x:Class="ShapesDemo.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
Title="ShapesDemo" Height="260" Width="230">
<Button Margin="5">
<Canvas Height="250" Width="220">
<Ellipse Canvas.Left="50" Canvas.Top="50" Width="100"
Height="100" Stroke="Blue" StrokeThickness="4"
Fill="Yellow" />
<Ellipse Canvas.Left="60" Canvas.Top="65" Width="25"
Height="25" Stroke="Blue" StrokeThickness="3"
Fill="White" />
<Ellipse Canvas.Left="70" Canvas.Top="75" Width="5"
Height="5" Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
Data="M 62,125 Q 95,122 102,108" />
<Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
StrokeThickness="4" />
<Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue"
StrokeThickness="4" />
</Canvas>
</Button>
</Window>
```

表 34-3 描述了 System.Windows.Shapes 命名空间中的图形。

表 34-3

Shape 类	说 明
Line	可以在坐标 X1,Y1 到 X2,Y2 之间绘制一条线
Rectangle	使用 Rectangle 类，可以指定 Width 和 Height，绘制一个矩形
Ellipse	使用 Ellipse 类，可以绘制一个椭圆
Path	使用 Path 类可以绘制一系列直线和曲线。Data 属性是 Geometry 类型。还可以使用派生自基类 Geometry 的类绘制图形，或使用路径标记语法来定义图形
Polygon	使用 Polygon 类可以绘制由线段连接而成的封闭图形。多边形由一系列赋予 Points 属性的 Point 对象定义
PolyLine	类似于 Polygon 类，使用 PolyLine 也可以绘制连接起来的线段，与多边形的区别是多义线不一定是封闭图形

### 34.3 变换

因为 WPF 基于 DirectX，DirectX 是基于矢量的，所以可以重置每个元素的大小。基于矢量的图形现在可以缩放、旋转和倾斜。即使没有手工计算位置，也可以进行单击测试（例如移动鼠标和单击鼠标）。

给 Canvas 元素的 LayoutTransform 属性添加 ScaleTransform 元素，如下所示，把整个笑脸的内容在 X 和 Y 方向上放大 2 倍。

```
<Canvas.LayoutTransform>
<ScaleTransform ScaleX="2" ScaleY="2" />
</Canvas.LayoutTransform>
```

旋转与缩放的执行方式相同。使用 RotateTransform 元素，可以定义旋转的角度：

```
<Canvas.LayoutTransform>
<RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

对于倾斜，可以使用 SkewTransform 元素。此时可以指定 X 和 Y 方向的倾斜角度：

```
<Canvas.LayoutTransform>
<SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

图 34-8 显示了这些变换的结果。这些图放在一个 StackPanel 中。从左到右，第一个图重置了大小，第二个图旋转了，第三个图倾斜了。为了更容易看出它们的区别，可以把 Canvas 元素的 Background 属性设置为不同的颜色。



图 34-8

#### 34.4 笔刷

本节介绍如何使用 WPF 提供的笔刷绘制背景和前景。本节将参考图 34-9，它显示了在 Button 元素的 Background 属性上使用各种笔刷的效果。



图 34-9

##### 34.4.1 SolidColorBrush

图 34-9 中的第一个按钮使用了 SolidColorBrush，顾名思义，这个笔刷使用纯色。完成的区域用同一种颜色绘制。

把 Background 属性设置为定义纯色的字符串，就可以定义纯色。该字符串会转换为一个 SolidColorBrush 元素。

```
<Button Height="30" Background="Purple">Solid
Color</Button>
```

当然，设置 Background 子元素，把 SolidColorBrush 元素添加为它的子内容，也可以得到这个效果。应用程序中的第二个按钮给背景使用了纯色 Yellow：

```
<Button Height="30" >
<Button.Background>
<SolidColorBrush>Yellow</SolidColorBrush>
</Button.Background>
Solid Color
</Button>
```

#### 34.4.2 LinearGradientBrush

对于平滑的颜色变化，可以使用 LinearGradientBrush，如第三个按钮所示。这个笔刷定义了 StartPoint 和 EndPoint 属性。使用这些属性可以为线性渐变指定两个坐标。默认的渐变方向是从 0,0 到 1,1 的对角线。定义其他值可以给渐变指定不同的方向。例如，StartPoint 指定为 0,0，EndPoint 指定为 0,1，就得到了一个垂直渐变。StartPoint 不变，EndPoint 指定为 1,0，就得到了一个水平渐变。

在这个笔刷中，可以用 GradientStop 元素定义指定偏移位置的颜色值。在各个偏移位置之间，颜色是平滑过渡的。

```
<Button Height="60">
<Button.Background>
<LinearGradientBrush StartPoint="0,0"
EndPoint="0.5,1.2">
<GradientStop Color="Red" Offset="0" /></GradientStop>
<GradientStop Color="Blue" Offset="0.2" />
</GradientStop>
<GradientStop Color="BlanchedAlmond" Offset="0.7" />
</GradientStop>
<GradientStop Color="DarkOrange" Offset="1" />
</GradientStop>
</LinearGradientBrush>
</Button.Background>
Linear Gradient Brush
</Button>
```

#### 34.4.3 RadialGradientBrush

使用 RadialGradientBrush 可以以放射方式产生平滑的颜色改变。在图 34-9 中，第四个按钮使用了 RadialGradientBrush。这个笔刷定义了从 GradientOrigin 点开始的颜色。

```
<Button Height="70" >
<Button.Background>
<RadialGradientBrush Center="0.5,0.5"
GradientOrigin="0.5,0.5"
RadiusX="0.5" RadiusY="0.5" SpreadMethod="Pad">
<GradientStop Color="White" Offset="0" />
<GradientStop Color="LightBlue" Offset="0.4" />
<GradientStop Color="DarkBlue" Offset="1" />
</RadialGradientBrush>
</Button.Background>
</Button>
```

```
</RadialGradientBrush>
</Button.Background>
Radial Gradient Brush
</Button>
```

#### 34.4.4 DrawingBrush

DrawingBrush 可以定义用笔刷绘制的图形。用笔刷绘制的图形在 GeometryDrawing 元素中定义。Geometry 属性中的 GeometryGroup 元素包含 Geometry 元素，例如 EllipseGeometry、LineGeometry、RectangleGeometry 和 CombineGeometry。

```
<Button Height="80">
<Button.Background>
<DrawingBrush>
<DrawingBrush.Drawing>
<GeometryDrawing Brush="LightBlue">
<GeometryDrawing.Geometry>
<GeometryGroup>
<EllipseGeometry RadiusX="30" RadiusY="30"
Center="20,20" />
<EllipseGeometry RadiusX="4" RadiusY="4"
Center="10,10" />
</GeometryGroup>
</GeometryDrawing.Geometry>
<GeometryDrawing.Pen>
<Pen>
<Pen.Brush>Red
</Pen.Brush>
</Pen>
</GeometryDrawing.Pen>
</GeometryDrawing>
</DrawingBrush.Drawing>
</DrawingBrush>
</Button.Background>
Drawing Brush
</Button>
```

#### 34.4.5 ImageBrush

要把图像加载到笔刷中，可以使用 ImageBrush 元素。在这个元素中，显示了 ImageSource 属性定义的图像。

```
<Button Height="100">
<Button.Background>
<ImageBrush
ImageSource=" C:\Windows\Web\Wallpaper\img21.bmp"
/>
```

```
</Button.Background>
<Button.Foreground>White</Button.Foreground>
Image Brush
</Button>
```

#### 34.4.6 VisualBrush

VisualBrush 可以在笔刷中使用其他 WPF 元素。下面给 Visual 属性添加一个 WPF 元素。图 34-9 中的第 7 个按钮包含一个矩形、一个椭圆和一个按钮。

```
<Button Height="100">
<Button.Background>
<VisualBrush >
<VisualBrush.Visual>
<StackPanel Background="White">
<Rectangle Width="25" Height="25"
Fill="LightCoral" Margin="2" />
<Ellipse Width="65" Height="20"
Fill="Aqua" Margin="5" />
<Button Margin="2">A Button</Button>
</StackPanel>
</VisualBrush.Visual>
</VisualBrush>
</Button.Background>
Visual Brush
</Button>
```

在 VisualBrush 中，还可以创建反射等效果。这里显示的按钮包含一个 StackPanel，它包含一个边框和一个矩形。边框包含一个 StackPanel，该 StackPanel 又包含一个标签和一个矩形。但这不是重点。第二个矩形是用 VisualBrush 填充的。这个笔刷定义了一个不透明值和一个变换。

Visual 属性绑定到 Border 元素上。变换是通过设置 VisualBrush 的 RelativeTransform 属性来完成的。这个变换使用了相对坐标。把 ScaleY 设置为-1，完成了 Y 向的反射。TranslateTransform 在 Y 向上移动变换，使反射效果位于原对象的下面。图 34-9 中的第 8 个按钮(VisualBrush2)显示了其效果。

提示：

这里使用的数据绑定和 Binding 元素详见下一章。

```
<Button Height="120">
<StackPanel>
<Border x:Name="reflected">
<Border.Background>Yellow</Border.Background>
<StackPanel>
<Label>Visual Brush 2</Label>
<Rectangle Width="70" Height="15" Margin="2"
Fill="BlueViolet" />
```

```
</StackPanel>
</Border>
<Rectangle Height="30">
<Rectangle.Fill>
<VisualBrush Opacity="0.35" Stretch="None"
Visual="{Binding ElementName=reflected}">
<VisualBrush.RelativeTransform>
<TransformGroup>
<ScaleTransform ScaleX="1" ScaleY="-1"
/>
<TranslateTransform Y="1" />
</TransformGroup>
</VisualBrush.RelativeTransform>
</VisualBrush>
</Rectangle.Fill>
</Rectangle>
</StackPanel>
</Button>
```

只要把 Visual 属性设置为 MediaElement , 就可以使用 VisualBrush 显示视频。对于 MediaControl , Source 属性应设置为 WMV 文件。在图 34-9 中 , 第 9 个按钮显示了 3 个女人 , 它就是显示视频的一个例子。但是在纸质媒介中 , 很难显示视频。读者可以自己试一试--如果使用 Windows Vista 的 Ultimate 版本 , 就可以在硬盘上找到这个视频。否则 , 可以选择另一个视频文件。

```
<Button Height="120">
<Button.Background>
<VisualBrush>
<VisualBrush.Visual>
<MediaElement x:Name="video"
Source="C:\Windows\ehome\ColorTint.wmv" />
</VisualBrush.Visual>
</VisualBrush>
</Button.Background>
</Button>
```

## 34.5 控件

可以给 WPF 使用上百个控件。为了更好地理解它们 , 我们把控件分为如下类别 :

简单控件

内容控件

有标题的内容控件

项控件

## 有标题的项控件

### 34.5.1 简单控件

简单控件是没有 Content 属性的控件。例如，Button 类可以包含任意图形、任意元素，这对于简单控件而言没有问题。表 34-4 列出了简单控件及其功能。

表 34-4

简单控件	说 明
PasswordBox	PasswordBox 控件用于输入密码。这个控件有用于输入密码的特殊属性，例如 PasswordChar 定义了在用户输入密码时显示的字符，Password 可以访问输入的密码。PasswordChanged 事件在修改密码时调用
ScrollBar	ScrollBar 控件包含一个 Thumb，用户可以在 Thumb 中选择一个值。如果文档在屏幕中放不下，就可以使用滚动条。一些控件包含滚动条，如果内容过多，就显示滚动条
ProgressBar	使用 ProgressBar 控件，可以指示某个时间较长的操作的进度
Slider	使用 Slider 控件，用户可以移动 Thumb，选择一个范围的值。ScrollBar、ProgressBar 和 Slider 派生自同一个基类 RangeBase
Textbox	Textbox 控件用于显示简单的无格式文本
RichTextbox	RichTextbox 控件通过 FlowDocument 类支持带格式的文本。RichTextBox 和 TextBox 派生自同一个基类 TextBoxBase

#### 提示：

尽管简单控件没有 Content 属性，但通过定义模板，完全可以定制这些控件的外观。模板详见本章后面的内容。

### 34.5.2 内容控件

ContentControl 有 Content 属性，利用 Content 属性，可以给控件添加任意内容。Button 类派生自基类 ContentControl，所以可以在这个控件中添加任意内容。在上面的例子中，Button 中有一个 Canvas 控件。表 34-5 列出了内容控件。

表 34-5

ContentControl 控件	说 明
Button	类 Button、RepeatButton、ToggleButton 和 GridViewColumnHeader 派生自同一个基类 ButtonBase。所有这些按钮都响应 Click 事件。RepeatButton 会重复响应 Click 事件，直到释放按钮为止
RepeatButton	
ToggleButton	
CheckBox	ToggleButton 是 CheckBox 和 RadioButton 的基类。这些按钮有开关状态。CheckBox 可以由用户选择和取消选择，RadioButton 可以由用户选择。清除 RadioButton 的选择必须通过编程来实现
RadioButton	
Label	Label 类表示控件的文本标签。这个类也支持访问键，例如菜单命令
Frame	Frame 控件支持导航。使用 Navigate() 方法可以导航到一个页面内容上。如果该内容是一个网页，就使用浏览器控件来显示
ListBoxItem	ListBoxItem 是 ListBox 控件中的一项
StatusBarItem	StatusBarItem 是 StatusBar 控件中的一项
ScrollViewer	ScrollViewer 是一个包含滚动条的内容控件，可以把任意内容放入这个控件，滚动条会在需要时显示

ToolTip	ToolTip 创建一个弹出窗口，显示控件的附加信息
UserControl	将 UserControl 类用作基类，可以为创建定制控件提供一种简单方式。但是，基类 UserControl 不支持模板
Window	Window 类可以创建窗口和对话框。使用这个类，会获得一个带有最小化/最大化/关闭按钮和系统菜单的框架。在显示对话框时，可以使用方法 ShowDialog()，方法 Show() 会打开一个窗口
NavigationWindow	类 NavigationWindow 派生自 Window 类，支持内容导航

只有 Frame 控件包含在下面 XAML 代码的 Window 中。Source 属性设置为 <http://www.wrox.com>，所以 Frame 控件导航到这个网站上，如图 34-10 所示。



图 34-10

```
<Window x:Class="FrameSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="FrameSample" Height="400" Width="400">
<Frame Source="http://www.wrox.com" />
</Window>
```

### 34.5.3 有标题的内容控件

带标题的内容控件派生自 HeaderContentControl 基类。HeaderContentControl 类又派生自基类 ContentControl。HeaderContentControl 类的 Header 属性定义了标题的内容，HeaderTemplate 属性可以对标题进行完全的定制。派生自基类 HeaderContentControl 的控件如表 34-6 所示。

表 34-6

HeaderContentControl	说 明
Expander	使用 Expander 控件，可以创建一个带对话框的“高级”模式，它在默认情况下不显示所有的信息，只有用户展开它，才会显示更多的信息。在未展开模式下，只显示标题信息，在展开模式下显示内容
GroupBox	GroupBox 控件提供了边框和标题来组合控件
TabItem	TabItem 控件是 TabControl 类中的项。TabItem 的 Header 属性定义了标题的内容，这些内容用 TabControl 的标签显示

Expander 控件的简单用法如下面的例子所示。Expander 控件的属性 Header 设置为 Click for more。这个文本用于显示扩展。这个控件的内容只有在控件展开时才显示。图 34-11 中的示例程序包含折叠的 Expander 控件，图 34-12 中的示例程序展开了 Expander 控件。

```
<Window x:Class="ExpanderSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Expander Sample" Height="300" Width="300">
<StackPanel>
<TextBlock>Short information</TextBlock>
<Expander Header="Click for more">
<Border Height="200" Width="200" Background="Yellow">
```

```
<TextBlock HorizontalAlignment="Center"  
VerticalAlignment="Center">  
    More information here!  
</TextBlock>  
</Border>  
</Expander>  
</StackPanel>  
</Window>
```

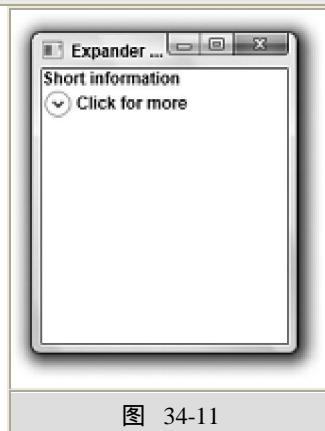


图 34-11



图 34-12

提示：

如果在展开 Expander 控件时，要修改该控件的标题文本，可以创建一个触发器，触发器详见本章后面的内容。

#### 34.5.4 项控件

类 ItemsControl 包含一列可以用 Items 属性访问的数据项。派生自 ItemsControl 的类如表 34-7 所示。

表 34-7

ItemsControl	说    明
Menu	类 Menu 和 ContextMenu 派生自抽象基类 MenuBase。把 MenuItem 元素放在数据项列表和相关的命令中，就可以给用户提供菜单
ContextMenu	

StatusBar	StatusBar 控件通常显示在应用程序的底部，为用户提供状态信息。可以把 StatusBarItem 元素放在 StatusBar 列表中
TreeView	要分层显示数据项，可以使用 TreeView 控件
ListBox	ListBox、ComboBox 和 TabControl 都有相同的抽象基类 Selector。这个基类可以从列表中选择数
ComboBox	据项。ListBox 显示列表中的数据项，ComboBox 有一个附带的 Button 控件，只有点击按钮，才会显示
TabControl	数据项。在 TabControl 中，内容可以排列为表格

### 34.5.5 带标题的项控件

HeaderItemsControl 是包含数据项和标题的控件的基类。类 HeaderItemsControl 派生自 ItemsControl。

派生自 HeaderItemsControl 的类如表 34-8 所示。

表 34-8

HeaderedItemsControl	说    明
MenuItem	菜单类 Menu 和 ContextMenu 包含 MenuItem 类型的数据项。菜单项可以连接到命令上，因为 MenuItem 类实现了接口 ICommandSource
TreeViewItem	TreeViewItem 类可以包含 TreeViewItem 类型的数据项
ToolBar	ToolBar 控件是一组控件(通常是 Button 和 Separator 元素)的容器。可以将 ToolBar 放在 ToolBarTray 中，它会重新安排 ToolBar 控件的位置

## 34.6 布局

为了定义应用程序的布局，可以使用派生自 Panel 基类中的类。这里讨论几个布局容器。布局容器要完成两个主要任务：测量和安排。在测量时，容器要求其子控件有合适的大小。由于控件的大小不一定合适，所以容器需要确定和安排其子控件的大小和位置。

### 34.6.1 StackPanel

Window 可以只包含一个元素，作为其内容。如果要包含多个元素，可以将 StackPanel 用作 Window 的一个子元素，在 StackPanel 中添加元素。StackPanel 是一个简单的容器控件，只能一个一个地显示元素。StackPanel 的显示方向可以是水平或垂直。类 ToolBarPanel 派生自 StackPanel。

```
<Window x:Class="LayoutSamples.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="300" Width="283">
<StackPanel Orientation="Vertical">
<Label>Label</Label>
<TextBox>TextBox</TextBox>
<CheckBox>Checkbox</CheckBox>
<CheckBox>Checkbox</CheckBox>
<ListBox>
<ListBoxItem>ListBoxItem One</ListBoxItem>
<ListBoxItem>ListBoxItem Two</ListBoxItem>
</ListBox>
```

```
<Button>Button</Button>
</StackPanel>
</Window>
```

在图 34-13 中，可以看到 StackPanel 垂直显示的子控件。

提示：

对于与 StackPanel 的数据绑定项，如果没有足够的空间显示所有的项，就可以使用 VirtualizingStackPanel。这个面板只生成显示的项。

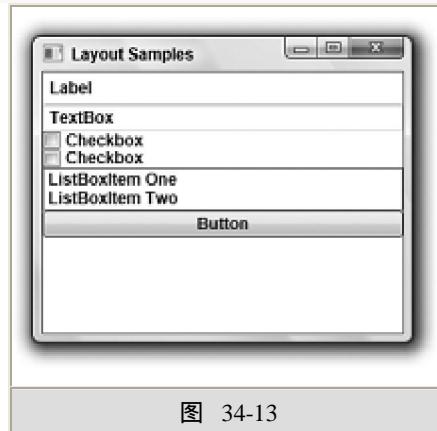


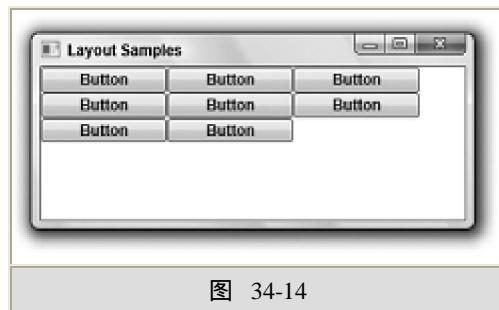
图 34-13

### 34.6.2 WrapPanel

WrapPanel 将子元素自左向右地排列，若一个水平行中放不下，就排在下一行。面板的排列方向可以是水平或垂直。

```
<Window x:Class="LayoutSamples.WrapPanelDemo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="160" Width="250">
<WrapPanel>
<Button Width="100">Button</Button>
</WrapPanel>
</Window>
```

图 34-14 显示了面板的排列结果。如果重新设置了应用程序的大小，按钮会重新排列，填满一行。



### 34.6.3 Canvas

Canvas 是一个允许显式指定控件位置的面板。它定义了相关的属性 Left、Right、Top 和 Bottom，这些属性可以由子元素在面板中定位时使用。

```
<Window x:Class="LayoutSamples.CanvasDemo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="300" Width="300">
<Canvas Background="LightBlue">
<Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
<TextBox Canvas.Top="30" Canvas.Left="130" Width="100"></TextBox>
<Button Canvas.Top="70" Canvas.Left="130">Click Me!</Button>
</Canvas>
</Window>
```

图 34-15 显示了 Canvas 面板的结果，其中定位了子元素 Label、TextBox 和 Button。



### 34.6.4 DockPanel

DockPanel 非常类似于 Windows 窗体的停靠功能。DockPanel 可以指定安排子控件的区域。

DockPanel 定义了附带属性 Dock，可以在控件的子控件中将它设置为 Left、Right、Top 和 Bottom。

图 34-16 显示了安排在 DockPanel 中的带边框的文本块。为了便于区别，为不同的区域指定了不同的颜色：

```
<Window x:Class="LayoutSamples.DockPanelDemo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="300" Width="300">
<DockPanel Background="LightBlue">
<Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
```

```
<TextBlock>Menu</TextBlock>
</Border>
<Border Height="25" Background="Aqua" DockPanel.Dock="Top">
<TextBlock>Toolbar</TextBlock>
</Border>
<Border Height="30" Background="LightSteelBlue"
DockPanel.Dock="Bottom">
<TextBlock>Status</TextBlock>
</Border>
<Border Width="80" Background="Azure" DockPanel.Dock="Left">
<TextBlock>Left Side</TextBlock>
</Border>
<Border Background="HotPink">
<TextBlock>Remaining Part</TextBlock>
</Border>
</DockPanel>
</Window>
```

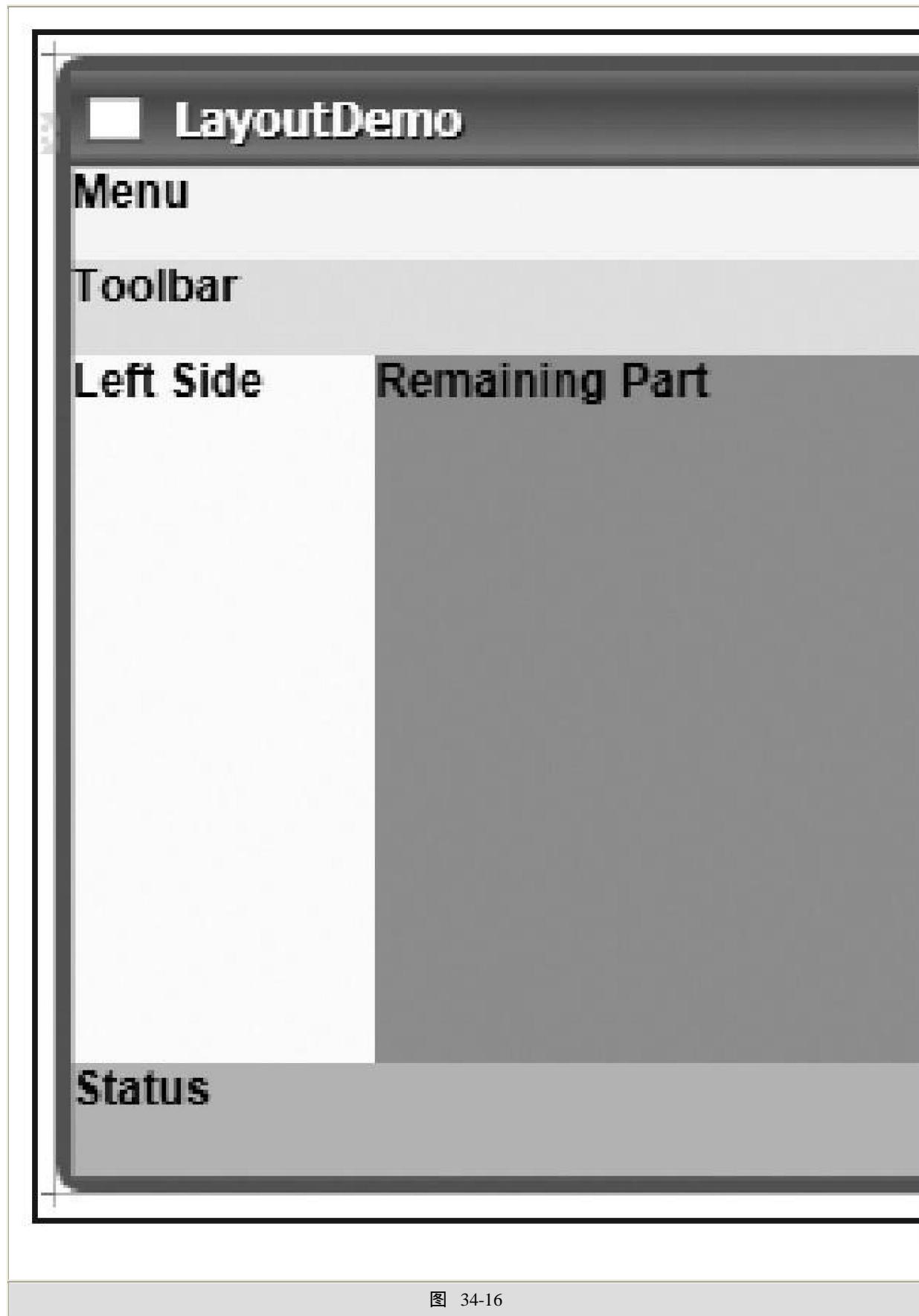


图 34-16

#### 34.6.5 Grid

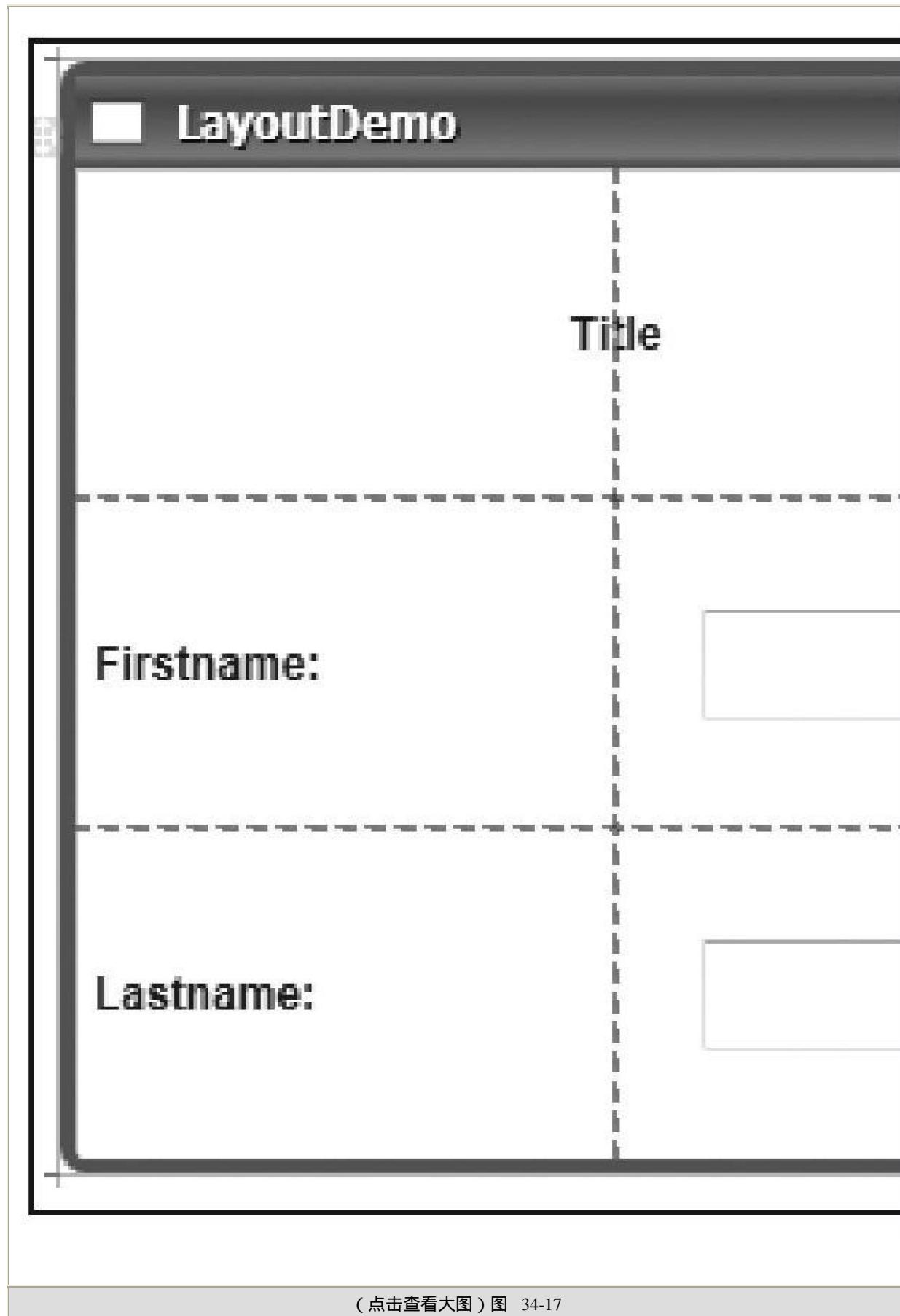
使用 Grid，可以在行和列中放置控件。对于每一列，可以指定 ColumnDefinition，对于每一行，可以指定 RowDefinition。下面的示例代码显示两列和三行。在每一列、每一行中，都可以指定宽度

或高度。ColumnDefinition 有一个 Width 依赖属性，RowDefinition 有一个 Height 依赖属性。可以以像素、厘米、英寸或点为单位定义高度和宽度，或者把它们设置为 Auto，根据内容来确定其大小。Grid 还允许根据具体情况指定大小，即根据可用的空间以及与其他行和列的相对位置，计算行和列的空间。在为列提供可用空间时，可以将 Width 属性设置为\*，要使某一列的空间是另一列的 2 倍，应指定 2\*。下面的示例代码定义了两列和三行，但没有定义列和行的其他设置，默认使用根据具体情况指定大小的设置。

这个 Grid 包含几个 Label 和 TextBox 控件。这些控件的父控件是 Grid，所以可以设置相关的属性 Column、ColumnSpan、Row 和 RowSpan。

```
<Window x:Class="LayoutSamples.GridDemo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Layout Samples" Height="300" Width="283">
<Grid ShowGridLines="True">
<Grid.ColumnDefinitions>
<ColumnDefinition />
<ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition />
<RowDefinition />
<RowDefinition />
</Grid.RowDefinitions>
<Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
VerticalAlignment="Center"
HorizontalAlignment="Center">Title</Label>
<Label Grid.Column="0" Grid.Row ="1" VerticalAlignment="Center">
Firstname:</Label>
<TextBox Grid.Column="1" Grid.Row="1" Width="100"
Height="30"></TextBox>
<Label Grid.Column="0" Grid.Row ="2" VerticalAlignment="Center">
Lastname:</Label>
<TextBox Grid.Column="1" Grid.Row="2" Width="100"
Height="30"></TextBox>
</Grid>
</Window>
```

在 Grid 中安排控件的结果如图 34-17 所示。为了便于看到列和行，ShowGridLines 属性设置为 true。



(点击查看大图) 图 34-17

提示：

要使 Grid 的每个单元格有相同的尺寸，可以使用 UniformGrid 类。

### 34.7 事件处理

WPF 类定义了能添加处理程序的事件，例如 MouseEnter、MouseLeave、MouseMove、Click 等。它们基于.NET 中的事件和委托机制。.NET 中的事件和委托机制详见第 7 章。

在 WPF 中，可以用 XAML 或在后台代码中指定事件处理程序。在 button1 中，XML 特性 Click 用于将方法 button\_Click 赋予单击事件。Button2 没有在 XAML 中指定事件处理程序：

```
<Button Name="button1" Click="button_Click">Button  
1</Button>  
<Button Name="button2" Button 2</Button>
```

在后台代码中，创建了 RoutedEventHandler 委托的一个实例，将 button\_Click 方法传送给该委托，从而指定了 Button2 的 Click 事件。在两个按钮中调用的方法 button\_Click () 定义了 RoutedEventHandler 委托描述的变元：

```
public Window1()  
{  
    InitializeComponent();  
    button2.Click += button_Click;  
}  
void OnButtonClick(object sender, RoutedEventArgs e)  
{  
    MessageBox.Show("Click Event");  
}
```

WPF 的事件处理机制基于.NET 事件，但扩展了起泡和通道特性。如前所述，Button 可以包含图形、列表框、另一个按钮等。如果 CheckBox 包含在 Button 中，单击了 CheckBox，会发生什么情况？事件应到达什么地方？答案是事件会起泡。首先，Click 事件会到达 CheckBox，然后向上起泡，到达 Button。这样，就可以用 Button 的 Click 事件处理包含在该 Button 中的所有元素的 Click 事件。

一些事件有通道事件，其他事件有起泡事件。通道事件先到达外部的元素，再沿着通道到达内部的元素。起泡事件先从内部元素开始，再到达外部的元素。通道和起泡事件通常是成对的。通道事件的前缀是 Preview，例如 PreviewMouseMove。这个事件从外部控件到达内部控件。在 PreviewMouseMove 事件后，会发生 MouseMove 事件。这个事件是一个起泡事件，它从内部控件到达外部控件。

将事件变元的 Handled 属性设置为 true，就可以关闭事件的通道和起泡功能。Handled 属性是 RoutedEventArgs 类的一个成员，所有参与通道和起泡特性的事件处理程序都有一个 RoutedEventArgs 类型或派生自 RoutedEventArgs 类型的事件变元。

提示：

如果将事件变元的 Handled 属性设置为 true，关闭事件的通道功能，则通道事件之后的起泡事件也不再会发生。

## 34.8 样式、模板和资源

设置 Button 元素的 FontSize 和 Background 属性，就可以定义 WPF 元素的外观和操作方式，如下所示：

```
<StackPanel>
<Button Name="button1" Width="150" FontSize="12"
Background="AliceBlue">
Click Me!
</Button>
</StackPanel>
```

除了定义每个元素的外观和操作方式之外，还可以定义用资源存储的样式。为了完全定制控件的外观，可以使用模板，再把它们存储到资源中。

### 34.8.1 样式

要定义样式，可以使用包含 Setter 元素的 Style 元素。使用 Setter，可以指定样式的 Property 和 Value，例如属性 Button.Background 和值 AliceBlue。

为了把样式赋予指定的元素，可以将样式赋予某一类型的所有元素，或者为该样式使用一个键。要把样式赋予某一类型的所有元素，可使用 Style 的 TargetType 属性，指定 x:Type 标记扩展(x>Type Button)，将样式赋予一个按钮。

```
<Window.Resources>
<Style TargetType="{x:Type Button}">
<Setter Property="Button.Background"
Value="LemonChiffon" />
<Setter Property="Button.FontSize" Value="18" />
</Style>
<Style x:Key="ButtonStyle">
<Setter Property="Button.Background" Value="AliceBlue"
/>
<Setter Property="Button.FontSize" Value="18" />
</Style>
</Window.Resources>
```

在下面的 XAML 代码中，button2 没有用元素属性定义样式，而是使用为 Button 类型定义的样式。button3 的 Style 属性用 StaticResource 标记扩展设置为{StaticResource ButtonStyle}，其中 ButtonStyle 指定了前面定义的样式资源的键值，所以 button3 的背景为 AliceBlue。

```
<Button Name="button2" Width="150">Click Me!</Button>
<Button Name="button3" Width="150"
Style="{StaticResource ButtonStyle}">
Click Me, Too!
</Button>
```

除了把按钮的 Background 设置为单个值之外，还可以将 Background 属性设置为定义了渐变色的 LinearGradientBrush，如下所示：

```
<Style x:Key="FancyButtonStyle">
    <Setter Property="Button.FontSize" Value="22" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.Background">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0.5,0"
                EndPoint="0.5,1">
                <GradientStop Offset="0.0" Color="LightCyan" />
                <GradientStop Offset="0.14" Color="Cyan" />
                <GradientStop Offset="0.7" Color="DarkCyan" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

button4 的样式采用青色的线性渐变效果：

```
<Button Name="button4" Width="200"
    Style="{StaticResource FancyButtonStyle}">
    Fancy!
</Button>
```

图 34-18 显示了这些按钮的效果。



图 34-18

### 34.8.2 资源

从样式示例可以看出，样式通常存储在资源中。可以在资源中定义任意元素，例如，前面为按钮的背景样式创建了笔刷，它本身就可以定义为一个资源，这样就可以在需要笔刷的地方使用它了。

下面的示例在 StackPanel 资源中定义了 LinearGradientBrush，它的键名是 MyGradientBrush。Button1 使用 StaticResource 标记扩展将 Background 属性赋予 MyGradientBrush 资源。图 34-19 显示了这段 XAML 代码的结果。

```
<Window x:Class="ResourcesSample.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Resources Sample" Height="100" Width="300"
>
```

```
<Window.Resources>
</Window.Resources>
<StackPanel>
<StackPanel.Resources>
<LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0.5,0"
EndPoint="0.5,1">
<GradientStop Offset="0.0" Color="LightCyan" />
<GradientStop Offset="0.14" Color="Cyan" />
<GradientStop Offset="0.7" Color="DarkCyan" />
</LinearGradientBrush>
</StackPanel.Resources>
<Button Name="button1" Width="200" Height="50" Foreground="White"
Background="{StaticResource MyGradientBrush}">
Click Me!
</Button>
</StackPanel>
</Window>
```



(点击查看大图) 图 34-19

这里，资源用 StackPanel 定义。在上面的例子中，资源用 Window 元素定义。基类 FrameworkElement 定义了 ResourceDictionary 类型的 Resources 属性。这就是资源可以用派生自 FrameworkElement 的所有类(任意 WPF 元素)来定义的原因。

资源是按层次结构来搜索的。如果用 Window 元素定义资源，它就会应用于 Window 的所有子元素。如果 Window 包含一个 Grid，该 Grid 又包含一个 StackPanel，资源是用 StackPanel 定义的，该资

源就会应用于 StackPanel 中的所有控件。如果 StackPanel 包含一个按钮，但只为该按钮定义了资源，这个样式就只对该按钮有效。

**警告：**

对于层次结构，需要注意是否为样式使用了没有 Key 的 TargetType。如果用 Canvas 元素定义了一个资源，并把样式的 TargetType 设置为应用于 TextBox 元素，该样式就会应用于 Canvas 中的所有 TextBox 元素。如果 Canvas 中有一个 ListBox，该样式甚至会应用于 ListBox 包含的 TextBox 元素。

如果需要将一个样式应用于多个窗口，就可以为应用程序定义样式。在一个 Visual Studio WPF 项目中，创建 App.xaml 文件，以定义应用程序的全局资源。应用程序样式对其中的每个窗口都有效；每个元素都可以访问为应用程序定义的资源。如果在父窗口中找不到资源，就可以在整个应用程序范围内搜索它。

```
<Application x:Class="StylingDemo.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

## 1. 系统资源

还有许多系统级的颜色和字体资源，可用于所有应用程序。这些资源用 SystemColors、SystemFonts 和 SystemParameters 类定义。

使用 SystemColors 可以获得边框、控件、桌面和窗口的颜色设置，例如，ActiveBorder Color、ControlBrush、DesktopColor、WindowColor、WindowBrush 等。

类 SystemFonts 返回菜单、状态栏、消息框等的字体设置，例如 CaptionFont、DialogFont、MenuFont、MessageBoxFont、StatusFont 等。

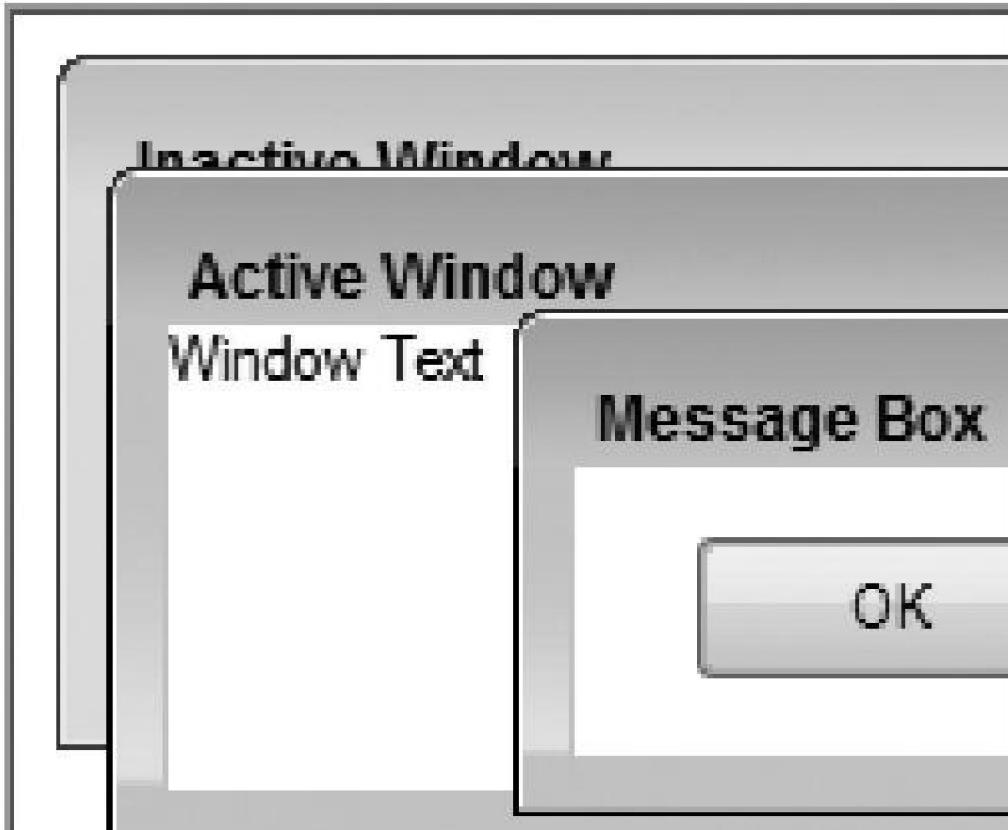
类 SystemParameters 设置了菜单按钮、光标、图标、边框标题、时间信息、键盘设置的大小。例如 BorderWidth、CaptionHeight、CaptionWidth、MenuButtonWidth、MenuPopupAnimation、MenuShowDelay、SmallIconHeight、SmallIconWidth 等。

在图 34-20 显示的对话框中，用户可以配置这些设置。在控制面板上，可以找到带 Personalization 设置的 Appearance 对话框。



## Appearance Settings

### Appearance



### Color scheme:

- Windows Aero**
- Windows Vista Basic
- Windows Standard
- Windows Classic
- High Contrast White
- High Contrast Black
- High Contrast #2

(点击查看大图) 图 34-20

## 2. 在代码中访问资源

要在后台代码中访问资源，基类 FrameworkElement 实现了方法 FindResource()，所以可以给每个 WPF 对象调用方法 FindResource()。

为此，button2 没有指定背景，但将 Click 事件赋予了方法 button1\_Click()。

```
<StackPanel Name="MyContainer">
<StackPanel.Resources>
<LinearGradientBrush x:Key="MyGradientBrush"
StartPoint="0.5,0" EndPoint="0.5,1">
<GradientStop Offset="0.0" Color="LightCyan" />
<GradientStop Offset="0.14" Color="Cyan" />
<GradientStop Offset="0.7" Color="DarkCyan" />
</LinearGradientBrush>
</StackPanel.Resources>
<Button Name="button2" Width="200" Height="50"
Click="button1_Click">
    Apply Resource Programmatically
</Button>
```

在方法 button1\_Click() 的实现代码中，给单击的按钮调用方法 FindResource()，按照层次结构搜索资源 MyGradientBrush，将该笔刷应用于控件的 Background 属性。

```
public void OnApplyResource(object sender,
RoutedEventArgs e)
{
    Control ctrl = sender as Control;
    ctrl.Background = ctrl.FindResource("MyGradientBrush")
as Brush;
}
```

警告：

如果方法 FindResource() 没有找到资源键，会抛出一个异常。如果不知道资源是否可用，就可以使用 TryFindResource() 方法来替代。如果找不到资源，TryFindResource() 方法就返回 null。

## 3. 动态资源

在 StaticResource 标记扩展中，资源是在加载期间搜索。如果在运行程序的过程中改变了资源，就应使用 DynamicResource 标记扩展。

下面的例子使用与前面相同的资源。button1 把资源用作 StaticResource，button3 使用 DynamicResource 标记扩展把资源用作 DynamicResource。button2 用于以编程方式改变资源。它的 Click 事件处理程序方法赋予了 button2\_Click。

```
<Button Name="button1" Width="200" Height="50"  
Background="{StaticResource MyGradientBrush}">  
    Static Resource  
</Button>  
  
<Button Name="button2" Width="200" Height="50"  
Click="button2_Click">  
    Change Resource  
</Button>  
  
<Button Name="button3" Width="200" Height="50"  
Background="{DynamicResource MyGradientBrush}">  
    Dynamic Resource  
</Button>
```

button2\_Click ()方法的实现代码清除了 StackPanel 的资源，用 MyGradientBrush 名称添加了一个新资源。这个新资源非常类似于在 XAML 代码中定义的资源，它只定义了不同的颜色。

```
public void button2_Click (object sender, RoutedEventArgs  
e)  
{  
    MyContainer.Resources.Clear();  
    LinearGradientBrush brush = new LinearGradientBrush();  
    brush.StartPoint = new Point(0.5, 0);  
    brush.EndPoint = new Point(0.5, 1);  
    GradientStopCollection stops = new  
    GradientStopCollection();  
    stops.Add(new GradientStop(Colors.White, 0.0));  
    stops.Add(new GradientStop(Colors.Yellow, 0.14));  
    stops.Add(new GradientStop(Colors.YellowGreen, 0.7));  
    brush.GradientStops = stops;  
    MyContainer.Resources.Add("MyGradientBrush", brush);  
}
```

如果运行应用程序，单击第三个按钮，动态修改资源，button4 就会立即获得新资源。用 StaticResource 定义的 button1 仍旧加载旧资源。

#### 警告：

DynamicResource 需要的性能资源比 StaticResource 更多，因为其资源总是在需要时加载。只有需要在运行期间进行修改，才给资源使用 DynamicResource。

#### 4. 触发器

使用触发器，可以动态修改控件的外观和操作方式，因为一些事件或属性值改变了。例如，用户在按钮上移动鼠标，按钮就会改变其外观。通常，这必须在 C# 代码中实现，但使用 WPF，也可以用 XAML 实现，而这只会影响 UI。

Style 类有一个 Triggers 属性，通过它可以指定属性触发器。下面的示例将两个 TextBox 元素放在 Canvas 面板中。利用 Window 资源定义一个样式 TextBoxStyle，TextBox 元素使用 Style 属性引用该样式。TextBoxStyle 指定，将 Background 设置为 LightBlue，FontSize 设置为 17。这是应用程序启动时 TextBox 元素的样式。使用触发器可以改变控件的样式。触发器在 Style.Triggers 元素中用 Trigger 元素定义。将一个触发器赋予属性 IsMouseOver，另一个触发器赋予属性 IsKeyboardFocused。这两个属性通过应用了样式的 TextBox 类定义。如果属性 IsMouseOver 的值是 true，就会引发触发器，将 Background 属性设置为 Red，FontSize 设置为 22。如果 TextBox 得到了焦点，属性 IsKeyboardFocused 就是 True，引发第二个触发器，将 TextBox 的 Background 属性设置为 Yellow。

```
<Window x:Class="TriggerSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="TriggerSample" Height="200" Width="400"
>
<Window.Resources>
<Style x:Key="TextBoxStyle" TargetType="{x:Type TextBox}">
<Setter Property="Background" Value="LightBlue" />
<Setter Property="FontSize" Value="17" />
<Style.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="Background" Value="Red" />
<Setter Property="FontSize" Value="22" />
</Trigger>
<Trigger Property="IsKeyboardFocused" Value="True">
<Setter Property="Background" Value="Yellow" />
<Setter Property="FontSize" Value="22" />
</Trigger>
</Style.Triggers>
</Style>
</Window.Resources>
<Canvas>
<TextBox Canvas.Top="80" Canvas.Left="30" Width="300"
Style="{StaticResource TextBoxStyle}" />
<TextBox Canvas.Top="120" Canvas.Left="30" Width="300"
Style="{StaticResource TextBoxStyle}" />
</Canvas>
</Window>
```

当引发触发器的原因不再有效时，就不必将属性值重置为初始值。例如不必定义 IsMouseOver = true 和 IsMouseOver=false 的触发器。只要引发触发器的原因不再有效，触发器进行的修改就会自动重置为初始值。

图 34-21 显示了触发器示例程序，其中，第一个文本框获得了焦点，第二个文本框使用样式的默认值指定背景和字号。



图 34-21

### 警告：

使用属性触发器，很容易改变控件的外观、字体、颜色、不透明度等。在鼠标滑过控件时，键盘设置焦点时，……都不需要编写任何代码。

Trigger 类定义了表 34-9 中的属性，以指定触发器的操作。

表 34-9

Trigger 属性	说 明
Property	使用属性触发器，Property 和 Value 属性用于指定触发器的引发时间，例如，Property =
Value	"IsMouseOver"， Value = "True"
Setters	一旦引发触发器，就可以使用 Setters 定义一个 Setter 元素集合，来改变属性值。Setter 类定义了属性 Property、TargetName 和 Value，以修改对象属性
EnterActions ExitActions	除了定义 Setters 之外，还可以定义 EnterActions 和 ExitActions。使用这两个属性，可以定义一个 TriggerAction 元素集合。EnterActions 在启动触发器时引发(此时属性触发器应用了 Property/Value 组合)。ExitActions 在触发器结束之前引发(此时还没有应用 Property/Value 组合)。 用这些操作指定的触发器操作派生自基类 TriggerAction，例如 SoundPlayerAction 和 BeginStoryboard。使用 SoundPlayerAction 可以开始播放声音，BeginStoryboard 用于动画，详见本章后面的内容。

### 提示：

属性触发器只是 WPF 中的一种触发器。另一种触发器是事件触发器。事件触发器在后面与动画一起论述。

## 5. 模板

本章前面介绍过，Button 控件可以包含任何内容，例如简单的文本，还可以给按钮添加一个 Canvas 元素，Canvas 元素可以包含图形。也可以给按钮添加 Grid、视频。按钮还可以完成更多的操作。

控件的外观、操作方式及其功能在 WPF 中是完全分离的。按钮有默认的外观，但可以用模板完全定制其外观。

表 34-10

模 板 类 型	说 明
ControlTemplate	使用 ControlTemplate 可以指定控件的可视化结构，重新设计其外观
ItemsPanelTemplate	对于 ItemsControl，可以赋予一个 ItemsPanelTemplate，以指定其项的布局。每个 ItemsControl 都有一个默认的 ItemsPanelTemplate。MenuItem 使用 WrapPanel，StatusBar 使用 DockPanel，ListBox 使用 VirtualizingStackPanel
DataTemplate	DataTemplates 非常适用于对象的图形化表示。给列表框指定样式，所以列表框中

	的项根据 ToString()方法的输出来显示。应用 DataTemplate，可以重写其操作，定义项的定制显示
HierarchicalData Template	HierarchicalDataTemplate 用于安排对象的树形结构。这个控件支持 HeaderedItemsControls，例如 TreeViewItem 和 MenuItem

下面的示例有几个按钮，然后逐步定制列表框，这样就可以立即看到改变的结果。首先，用两个非常简单的按钮作为开始，第一个按钮根本没有应用样式，第二个按钮引用了样式 ButtonStyle1，来改变 Background 和 FontSize。图 34-22 显示了第一个结果。

```
<Window x:Class="TemplateSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Template Sample" Height="300" Width="300">
<Window.Resources>
<Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
<Setter Property="Background" Value="Yellow" />
<Setter Property="FontSize" Value="18" />
</Style>
</Window.Resources>
<StackPanel>
<Button Name="button1" Height="50" Width="150">Default
Button</Button>
<Button Name="button2" Height="50" Width="150"
Style="{StaticResource ButtonStyle1}">Styled Button
</Button>
</StackPanel>
</Window>
```



图 34-22

现在，给资源添加新样式 ButtonStyle2。这个样式再次将 TargetType 设置为 Button 类型。Setter 现在指定了 Template 属性。而指定 Template 属性，可以完全改变按钮的外观。Template 属性的值通过 ControlTemplate 元素来定义。ControlTemplate 定义了控件的内容，允许访问控件的内容，如后面所述。这里，ControlTemplate 定义了一个包含两行的 Grid。该行根据具体情况指定大小，其中第一行的高度是第二行的两倍。接着定义了两个 Rectangle 元素。第一个矩形横跨两行，并将 Stroke 属性设置为 Green，获得绿色的边框，再设置 RadiusX 和 RadiusY 值，得到圆角矩形。第二个矩形位于第一

个矩形内部，其 Fill 属性设置为线性渐变笔刷。button3 的内容设置为 Template Button，并引用样式 ButtonStyle2。图 34-23 显示了采用新样式的 button3，但内容没有显示。

```
<Window x:Class="TemplateSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Template Sample" Height="300" Width="300"
>
<Window.Resources>
<!-- other styles -->
<Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate>
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="2*" />
<RowDefinition Height="*" />
</Grid.RowDefinitions>
<Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green"
/>
<Rectangle RadiusX="4" RadiusY="8" Margin="2">
<Rectangle.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="LightBlue" />
<GradientStop Offset="0.5" Color="#afff" />
<GradientStop Offset="1" Color="#6faa" />
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
</Grid>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<StackPanel>
<!-- other buttons -->
<Button Name="button3" Background="Yellow" Height="100" Width="220"
FontSize="24"
Style="{StaticResource ButtonStyle2}">Template Button
</Button>
</StackPanel>
</Window>
```



按钮现在的外观完全不同，但按钮的内容未在图 34-23 中显示出来。必须扩展前面创建的模板。模板中的第一个矩形现在把其 Fill 属性设置为 {TemplateBinding Background}。标记扩展

TemplateBinding 允许控件模板使用模板控件中的内容。这里，矩形用按钮定义的背景来填充。button3 定义了黄色背景，所以这个背景与控件模板中第二个矩形的背景合并起来。定义了第二个矩形后，使用 ContentPresenter 元素。该元素从模板控件中提取内容，根据定义放置这些内容，这里是放在两个行上，因为 Grid.RowSpan 设置为 2。如果定义了一个 ContentPresenter 元素，就必须用 ControlTemplate 设置 TargetType。使用 TemplateBinding 标记扩展 将 HorizontalAlignment、VerticalAlignment 和 Margin 属性设置为按钮定义的值，以指定内容在按钮中的位置。使用 ControlTemplate 还可以在资源中定义前面的触发器。图 34-24 显示了按钮的新结果，其中包括内容和合并了模板的背景。

```
<Style x:Key="ButtonStyle2" TargetType="{x:Type Button}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Button}">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="2*" />
<RowDefinition Height="*" />
</Grid.RowDefinitions>
<Rectangle Grid.RowSpan="2" RadiusX="4" RadiusY="8" Stroke="Green"
Fill="{TemplateBinding Background}" />
<Rectangle RadiusX="4" RadiusY="8" Margin="2">
<Rectangle.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="LightBlue" />
<GradientStop Offset="0.5" Color="#afff" />
<GradientStop Offset="1" Color="#6faa" />
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<ContentPresenter Grid.RowSpan="2"
```

```
HorizontalAlignment="{TemplateBinding  
HorizontalContentAlignment}"  
VerticalAlignment="{TemplateBinding  
VerticalContentAlignment}"  
Margin="{TemplateBinding Padding}" />  
</Grid>  
<ControlTemplate.Triggers>  
<Trigger Property="IsMouseOver" Value="True">  
<Setter Property="Foreground" Value="Aqua" />  
</Trigger>  
<Trigger Property="IsPressed" Value="True">  
<Setter Property="Foreground" Value="Black" />  
</Trigger>  
</ControlTemplate.Triggers>  
</ControlTemplate>  
</Setter.Value>  
</Setter>  
</Style>  
</Window.Resources>
```

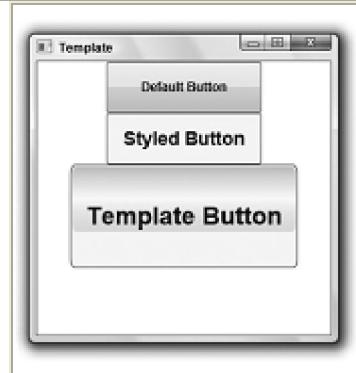


图 34-24

下面使用透明特性创建一个更有趣的按钮。样式 GelButton 设置了属性 Background、Height、Foreground、Margin 和 Template。模板是这个样式中最有趣的部分。该模板将 Grid 指定为一行一列。

在这个单元格中，有一个名为 GelBackground 的矩形。这个矩形是圆角矩形，因为指定了 RadiusX 和 RadiusY 值，该矩形的边框非常细，且采用线性渐变色，因为 StrokeThickness 设置为 0.35，还为这类笔触设置了线性渐变笔刷。

第二个矩形 GelShine 是一个高度为 15 像素的小矩形，因为设置了 Margin，所以在第一个矩形中可见。其笔触是透明的，所以该矩形没有边框。这个矩形只使用一个线性渐变填充笔刷，其颜色从部分透明的浅色变为完全透明。这会使该矩形具有微微发亮的效果。

在这两个矩形之后，再创建一个 ContentPresenter 元素，它为内容指定对齐方式，并从按钮中提取要显示的内容。

这样一个样式的按钮在屏幕上看起来很漂亮。但是，如果用鼠标单击它，或使鼠标滑过该按钮，它不会有任何动作。对于模板样式的按钮，必须给它指定触发器，使它在响应鼠标单击时显示不同的外观。属性触发器 IsMouseOver 为 Rectangle.Fill 属性定义一个新值，用辐射渐变笔刷给矩形填充另一种颜色。有新填充效果的矩形通过 TargetName 来引用。属性触发器 IsPressed 与前面的触发器非常类似，但用另一种颜色的辐射渐变笔刷填充矩形。图 34-25 显示了一个引用样式 GelButton 的按钮。图 34-26 显示了鼠标滑过时该按钮的外观，从中可以看出辐射渐变笔刷的效果。

```
<Style x:Key="GelButton" TargetType="{x:Type Button}">
<Setter Property="Background" Value="Black" />
<Setter Property="Height" Value="40" />
<Setter Property="Foreground" Value="White" />
<Setter Property="Margin" Value="3" />
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Button}">
<Grid>
<Rectangle Name="GelBackground" RadiusX="9" RadiusY="9"
Fill="{TemplateBinding Background}"
StrokeThickness="0.35">
<Rectangle.Stroke>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="White" />
<GradientStop Offset="1" Color="#666666" />
</LinearGradientBrush>
</Rectangle.Stroke>
</Rectangle>
<Rectangle Name="GelShine" Margin="2,2,2,0"
VerticalAlignment="Top"
RadiusX="6" RadiusY="6" Stroke="Transparent"
Height="15px">
<Rectangle.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="#ccffff" />
<GradientStop Offset="1" Color="Transparent" />
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<ContentPresenter Name="GelButtonContent"
VerticalAlignment="Center"
HorizontalAlignment="Center"
Content="{TemplateBinding Content}" />
</Grid>
<ControlTemplate.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="Rectangle.Fill"
TargetName="GelBackground">
```

```
<Setter.Value>
<RadialGradientBrush>
<GradientStop Offset="0" Color="Lime" />
<GradientStop Offset="1" Color="DarkGreen" />
</RadialGradientBrush>
</Setter.Value>
</Setter>
<Setter Property="Foreground" Value="Black" />
</Trigger>
<Trigger Property="IsPressed" Value="True">
<Setter Property="Rectangle.Fill"
TargetName="GelBackground">
<Setter.Value>
<RadialGradientBrush>
<GradientStop Offset="0" Color="#ffcc00" />
<GradientStop Offset="1" Color="#cc9900" />
</RadialGradientBrush>
</Setter.Value>
</Setter>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

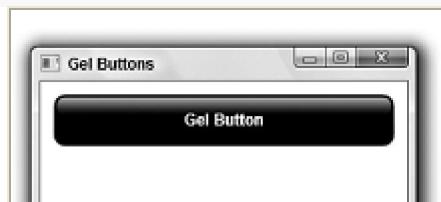


图 34-25



图 34-26

除了使用矩形按钮之外，还可以创建椭圆按钮。下面的例子将探讨如何使一种样式基于另一种样式。

用 Style 元素设置 BasedOn 属性，就可以使样式 RoundedGelButton 基于样式 GelButton。如果一种样式基于另一种样式，新样式就会获得基样式的所有设置，除非重新定义了设置。例如，RoundedGelButton 样式从 GelButton 样式中获得了 Foreground 和 Margin 设置，因为这些设置没有重新定义。如果改变了基样式中的一个设置，基于该样式的所有样式都会自动获得这个新设置值。

Height 和 Template 属性在新样式中重新定义了。其中，模板定义了两个 Ellipse 元素来替代矩形。外部椭圆的 GelBackground 定义了一个边框有渐变效果的黑色椭圆。第二个椭圆较小，其顶边距第一个椭圆 5 像素，底边距第一个椭圆 50 像素。这个椭圆也使用线性渐变效果，其颜色从一种浅色变为透明色，并指定了“发光”效果。还要为 IsMouseOver 和 IsPressed 指定触发器，改变第一个椭圆的 Fill 属性值。

在图 34-27 中，显示了基于 RoundedGelButton 样式的新按钮，它仍旧是一个按钮。

```
<Style x:Key="RoundedGelButton"
BasedOn="{StaticResource GelButton}"
TargetType="Button">
<Setter Property="Width" Value="100" />
<Setter Property="Height" Value="100" />
<Setter Property="Grid.Row" Value="2" />
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Button}">
<Grid>
<Ellipse Name="GelBackground" StrokeThickness="0.5"
Fill="Black">
<Ellipse.Stroke>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="#ff7e7e7e" />
<GradientStop Offset="1" Color="Black" />
</LinearGradientBrush>
</Ellipse.Stroke>
</Ellipse>
<Ellipse Margin="15,5,15,50">
<Ellipse.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="#aaffffff" />
<GradientStop Offset="1" Color="Transparent" />
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
<ContentPresenter Name="GelButtonContent"
VerticalAlignment="Center"
HorizontalAlignment="Center" Content="{TemplateBinding
Content}" />
</Grid>
```

```
<ControlTemplate.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="Rectangle.Fill"
TargetName="GelBackground">
<Setter.Value>
<RadialGradientBrush>
<GradientStop Offset="0" Color="Lime" />
<GradientStop Offset="1" Color="DarkGreen" />
</RadialGradientBrush>
</Setter.Value>
</Setter>
<Setter Property="Foreground" Value="Black" />
</Trigger>
<Trigger Property="IsPressed" Value="True">
<Setter Property="Rectangle.Fill"
TargetName="GelBackground">
<Setter.Value>
<RadialGradientBrush>
<GradientStop Offset="0" Color="#ffcc00" />
<GradientStop Offset="1" Color="#cc9900" />
</RadialGradientBrush>
</Setter.Value>
</Setter>
<Setter Property="Foreground" Value="Black" />
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

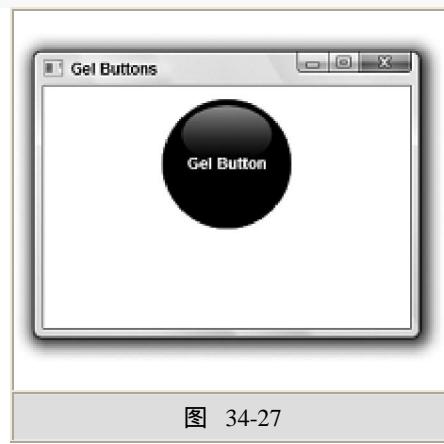


图 34-27

### 34.8.3 创建列表框的样式

创建按钮或标签的样式是一个简单的任务。如何改变包含一组元素的父元素的样式，例如列表框？列表框也有操作方式和外观。列表框可以显示一组元素，用户可以从列表中选择一个或多个元素。至于操作，ListBox 类定义了方法、属性和事件。列表框的外观与其操作是分开的。列表框元素有一个默认的外观，但可以通过创建模板，改变这个外观。

为了显示列表中的某些数据项，下面创建 Country 类，来表示名称和图像路径标志。Country 类定义了 Name 和 ImagePath 属性，用重写的 ToString() 方法表示默认的字符串：

```
public class Country
{
    public Country(string name)
        : this(name, null)
    {
    }
    public Country(string name, string imagePath)
    {
        this.name = name;
        this.imagePath = imagePath;
    }
    public string Name{ get; set; }
    public string ImagePath{ get; set; }
    public override string ToString()
    {
        return name;
    }
}
```

静态类 Countries 返回几个要显示的国家：

```
public static class Countries
{
    public static Country[] GetCountries()
    {
        List<Country> countries = new List<Country>();
        countries.Add(new Country("Austria",
            "Images/Austria.bmp"));
        countries.Add(new Country("Germany",
            "Images/Germany.bmp"));
        countries.Add(new Country("Norway",
            "Images/Norway.bmp"));
        countries.Add(new Country("USA", "Images/USA.bmp"));
        return countries.ToArray();
    }
}
```

在后台编码文件中，Window1 类的构造函数将 Window1 实例的 DataContext 属性设置为要从 Countries.GetCountries() 方法中返回的国家列表。DataContext 属性是数据绑定的一个特性，详见下一章。

```
public partial class Window1 : System.Windows.Window
{
    public Window1()
    {
        InitializeComponent();
        this.DataContext = Countries.GetCountries();
    }
}
```

在 XAML 代码中，定义了 countryList1 列表框。countryList1 没有使用样式，只使用了列表框元素的默认外观。属性 ItemsSource 设置为 Binding 标记扩展，它由数据绑定使用，在后台代码中，数据绑定用于一个 Country 对象数组。图 34-28 显示了列表框的默认外观。在默认情况下，只在一个简单的列表中显示 ToString() 方法返回的国家名称。

```
<Window x:Class="ListboxStyling.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Listbox Styling" Height="300" Width="300">
<StackPanel>
<ListBox Name="countryList1" ItemsSource="{Binding}" />
</StackPanel>
</Window>
```

Country 对象有名称和标志。当然，还可以在列表框中显示这两个值。为此，必须定义一个模板。

列表框元素包含 ListBoxItem 元素。使用 ItemTemplate 可以定义列表项的内容。样式 listBoxStyle1 定义了一个 ItemTemplate，其值为 DataTemplate。DataTemplate 用于将数据绑定到元素上。Binding 标记扩展可以用于 DataTemplate 元素。



图 34-28

DataTemplate 包含一个带三列的栅格。第一列包含字符串"Country："，第二列包含国家的名称，第三列包含该国家的标志。因为国家名称的长度是不同的，但看起来每个国家名称应有相同的大小，所以给第二列定义 SharedSizeGroup 属性。只有这一列使用共享大小的信息，因为设置了属性 Grid.IsSharedSizeScope。

行和列定义之后，就可以看到两个 TextBlock 元素。第一个 TextBlock 元素包含文本"Country："，第二个 TextBlock 元素绑定到 Country 类定义的 Name 属性上。

第三列的内容是一个包含栅格的 Border 元素。栅格包含一个带线性边框的矩形和一个绑定到 Country 类中 ImagePath 属性上的 Image 元素。图 34-29 显示了列表框中的国家，其效果与前面完全不同。

```
<Window.Resources>
<Style x:Key="listBoxStyle1" TargetType="{x:Type
ListBox}" >
```

```
<Setter Property="ItemTemplate">
<Setter.Value>
<DataTemplate>
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" />
<ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
<ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="60" />
</Grid.RowDefinitions>
<TextBlock FontSize="16" VerticalAlignment="Center" Margin="5" FontStyle="Italic" Grid.Column="0">Country:</TextBlock>
<TextBlock FontSize="16" VerticalAlignment="Center" Margin="5" Text="{Binding Name}" FontWeight="Bold" Grid.Column="1" />
<Border Margin="4,0" Grid.Column="2" BorderThickness="2" CornerRadius="4">
<Border.BorderBrush>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="#aaa" />
<GradientStop Offset="1" Color="#222" />
</LinearGradientBrush>
</Border.BorderBrush>
<Grid>
<Rectangle>
<Rectangle.Fill>
<LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
<GradientStop Offset="0" Color="#444" />
<GradientStop Offset="1" Color="#fff" />
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
<Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
</Grid>
</Border>
</Grid>
</DataTemplate>
</Setter.Value>
</Setter>
<Setter Property="Grid.IsSharedSizeScope" Value="True" />
```

```
</Style>  
</Window.Resources>
```



图 34-29



图 34-30

列表框中的数据项不一定是垂直排列，还可以给这个功能提供另一种视图。下一个样式 listBoxStyle2 定义了一个模板，使列表项水平显示，且带一个滚动条。

在前面的示例中，只创建了一个 ItemTemplate，来定义列表项在默认列表框中的外观。现在创建一个模板，定义另一个列表框。该模板包含一个 ControlTemplate 元素，它定义了列表框的元素。ControlTemplate 元素现在是一个包含 StackPanel 的 ScrollViewer-- 带滚动条的视图。列表项现在应水平排列，所以 StackPanel 的 Orientation 设置为 Horizontal。StackPanel 还包含用 ItemTemplate 定义的数据项，所以 StackPanel 元素的 IsItemsHost 设置为 true。IsItemsHost 属性可以用于包含一列数据项的所有 Panel 元素。

ItemTemplate 定义了 StackPanel 中数据项的外观。在其中创建一个包含两行的栅格，第一行包含绑定到 ImagePath 属性上的 Image 元素，第二行包含绑定到 Name 属性上的 TextBlock。

图 34-30 显示了使用 listBoxStyle2 样式的列表框，当视图太小，不能显示列表中的所有项时，滚动条会自动显示出来。

```
<Style x:Key="listBoxStyle2" TargetType="{x:Type
ListBox}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type ListBox}">
<ScrollViewer HorizontalScrollBarVisibility="Auto">
<StackPanel Name="StackPanell" IsItemsHost="True"
Orientation="Horizontal" />
</ScrollViewer>
</ControlTemplate>
</Setter.Value>
</Setter>
<Setter Property="VerticalAlignment" Value="Center" />
<Setter Property="ItemTemplate">
<Setter.Value>
<DataTemplate>
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="60" />
<RowDefinition Height="30" />
</Grid.RowDefinitions>
<Image Grid.Row="0" Width="48" Margin="2,2,2,1"
Source="{Binding ImagePath}" />
<TextBlock Grid.Row="1" FontSize="14"
HorizontalAlignment="Center"
Margin="5" Text="{Binding Name}" FontWeight="Bold" />
</Grid>
</DataTemplate>
</Setter.Value>
</Setter>
</Style>
```

显然，将控件的外观与其操作分开是很有好处的。也许用户有许多方式，能很好地显示列表中的数据项，使之满足应用程序的要求。也许用户只想在窗口中显示一定数量的数据项，先水平排列，一行放不下时，就继续排在下一行。此时就可以使用 WrapPanel。当然，可以给列表框的模板定义一个 WrapPanel，如 listBoxStyle3 所示。图 34-31 显示了使用 WrapPanel 的结果：

```
<Style x:Key="listBoxStyle3" TargetType="{x:Type  
ListBox}">  
    <Setter Property="Template">  
        <Setter.Value>  
            <ControlTemplate TargetType="{x:Type ListBox}">  
                <ScrollViewer VerticalScrollBarVisibility="Auto"  
                    HorizontalScrollBarVisibility="Disabled">  
                    <WrapPanel IsItemsHost="True" />  
                </ScrollViewer>  
            </ControlTemplate>  
        </Setter.Value>  
    </Setter>  
    <Setter Property="ItemTemplate">  
        <Setter.Value>  
            <DataTemplate>  
                <Grid>  
                    <Grid.ColumnDefinitions>  
                        <ColumnDefinition Width="140" />  
                    </Grid.ColumnDefinitions>  
                    <Grid.RowDefinitions>  
                        <RowDefinition Height="60" />  
                        <RowDefinition Height="30" />  
                    </Grid.RowDefinitions>  
                    <Image Grid.Row="0" Width="48" Margin="2,2,2,1"  
                        Source="{Binding ImagePath}" />  
                    <TextBlock Grid.Row="1" FontSize="14"  
                        HorizontalAlignment="Center"  
                        Margin="5" Text="{Binding Name}" />  
                </Grid>  
            </DataTemplate>  
        </Setter.Value>  
    </Setter>  
</Style>
```



图 34-31

提示：

下一章介绍了 DataTemplate 和数据绑定功能。

#### 34.9 小结

本章介绍了 WPF 的许多特性。WPF 便于分开开发人员和设计人员的工作。Microsoft Expression Blend 和 Visual Studio 都可以使用 XAML 代码。XAML 代码能很好地分开 UI 及其功能。所有的 UI 特性都可以使用 XAML 创建，其功能用后台代码创建。

我们还探讨了基于矢量图形的许多控件和容器。WPF 元素是矢量图形，所以可以缩放、倾斜和旋转。内容控件的内容非常灵活，所以事件处理机制基于起泡和通道事件。

可以使用不同类型的笔刷绘制背景和前景元素，例如可以使用纯色笔刷、线性渐变和辐射渐变笔刷、可视化笔刷完成反射功能或显示视频。

样式和模板可以定制控件的外观。触发器可以动态改变 WPF 控件的属性。连续改变 WPF 控件的属性值，就可以制作出动画。

下一章继续介绍 WPF，主要探讨动画、3D、数据绑定和其他几个特性。

## 第 35 章高 级 WPF

上一章介绍了 WPF 的一些核心功能，本章继续用 WPF 编程，介绍创建完整应用程序的一些重要方面，例如数据绑定和命令处理，还要探讨动画和 3D 编程。

本章的主要内容如下：

数据绑定

命令

动画

3D

Windows Forms 集成

### 35.1 数据绑定

上一章在给列表框设置样式时，介绍了数据绑定的几个特性。当然，数据绑定还有许多特性。WPF 数据绑定与 Windows Forms 数据绑定相比，进了一大步。本节介绍 WPF 中的数据绑定，讨论如下主题：

概述

用 XAML 绑定

简单的对象绑定

对象数据提供程序

列表绑定

绑定到 XML 上

#### 35.1.1 概述

在 WPF 数据绑定中，目标可以是 WPF 元素的任意依赖属性，CLR 对象的每个属性都可以是绑定源。WPF 元素实现为.NET 类，所以每个 WPF 元素都可以用作绑定源。图 35-1 显示了绑定源和绑定目标之间的连接。Binding 对象定义了该连接。

Binding 对象支持源与目标之间的几种绑定模式。绑定可以是单向的，即源信息传送给目标，但如果用户在用户界面上修改了该信息，源不会更新。要更新源，需要双向绑定。

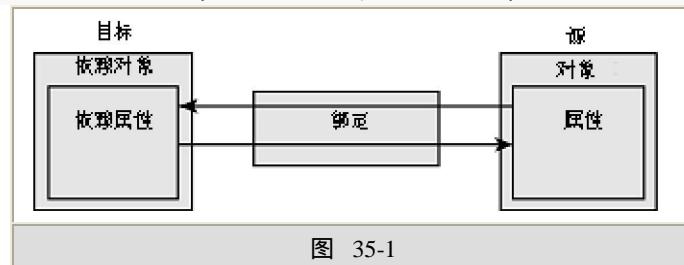


表 35-1 列出了绑定模式及其要求。

表 35-1

绑定模式	说    明
一次性	绑定从源指向目标，且仅在应用程序启动时，或数据内容改变时绑定一次。通过这种模式可以获得数据的快照
单向	绑定从源指向目标。可以用于只读数据，因为它不能在用户界面上修改数据。要更新用户界面，源必须实现接口 INotifyPropertyChanged
双向	在双向绑定中，用户可以在 UI 上修改数据。绑定是双向的——从源指向目标，从目标指向源。源对象需要实现读写属性，才能把改动的内容从 UI 更新到源对象上
指向源的单向	采用这种绑定模式，如果目标属性改变了，源对象也会更新

### 35.1.2 用 XAML 绑定

WPF 元素不仅是数据绑定的目标，还是绑定的源。可以把一个 WPF 元素的源属性绑定到另一个 WPF 元素的目标属性上。

下面的例子使用前面创建的笑脸，它是用 WPF 图形创建的，然后将其绑定到一个滑块上，以便在窗口中移动它。滑块是源元素，其名称是 slider。属性 Value 给出了滑块的位置值。数据绑定的目标是内部的 Canvas 元素。这个 Canvas 元素的名称是 FunnyFace，包含了绘制笑脸所需的所有图形。该 Canvas 元素包含在外部的 Canvas 元素中，所以可以设置关联的属性，在外部的 Canvas 元素中定位内部的 Canvas 元素。关联属性 Canvas.Left 设置为 Binding 标记扩展。在 Binding 标记扩展中，ElementName 设置为 slider，以引用 WPF 滑块元素，Path 设置为 Value，从 Value 属性中获取值。

```
<Window x:Class="DataBindingSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="DataBindingSample" Height="345" Width="310">
<StackPanel>
<Canvas Height="210" Width="280">
<Canvas Canvas.Top="0"
Canvas.Left="{Binding Path=Value, ElementName=slider}"
Name="FunnyFace" Height="210" Width="230">
<Ellipse Canvas.Left="20" Canvas.Top="50" Width="100" Height="100"
Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
<Ellipse Canvas.Left="40" Canvas.Top="65" Width="25" Height="25"
Stroke="Blue" StrokeThickness="3" Fill="White" />
<Ellipse Canvas.Left="50" Canvas.Top="75" Width="5" Height="5"
Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
Data="M 32,125 Q 65,122 72,108" />
<Line X1="94" X2="102" Y1="144" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="84" X2="103" Y1="169" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="62" X2="52" Y1="146" Y2="168" Stroke="Blue"
```

```
StrokeThickness="4" />
<Line X1="38" X2="53" Y1="160" Y2="168" Stroke="Blue"
StrokeThickness="4" />
</Canvas>
</Canvas>
<Slider Name="slider" Orientation="Horizontal" Value="10"
Maximum="100" />
</StackPanel>
</Window>
```

运行应用程序时，可以移动滑块，笑脸就会移动，如图 35-2 和 35-3 所示。

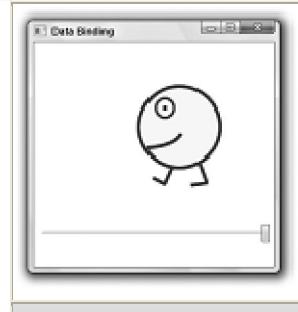


图 35-2

图 35-3

除了用 XAML 代码定义绑定信息之外，如这个例子使用 Binding 元数据扩展来定义，还可以使用后台代码。绑定的 XAML 版本如下：

```
<Canvas Canvas.Top="0"
Canvas.Left="{Binding Path=Value, ElementName=slider}"
Name="FunnyFace" Height="210" Width="230">
```

在后台代码中，必须创建一个新的 Binding 对象，设置 Path 和 Source 属性。Source 属性必须设置为源对象，这里是 WPF 对象 slider。Path 属性设置为一个 PropertyPath 实例，它用源对象的 Value 属性名初始化。对于目标，可以调用 SetBinding()方法来定义绑定。这里，目标是名为 FunnyFace 的 Canvas 对象。SetBinding()方法需要两个参数，第一个参数是一个依赖属性，第二个参数是绑定对象。Canvas.Left 属性应是绑定的，这样 DependencyProperty 类型的依赖属性才能用 Canvas.LeftProperty 字段访问：

```
Binding binding = new Binding();
binding.Path = new PropertyPath("Value");
binding.Source = slider;
FunnyFace.SetBinding(Canvas.LeftProperty, binding);
```

使用 Binding 类，可以配置许多绑定选项，如表 35-2 所示。

表 35-2

Binding 类的成员	说 明
Source	使用 Source 属性 , 可以定义数据绑定的源对象
RelativeSource	使用 RelativeSource 属性 , 可以指定与目标对象相关的源对象。当错误来源于同一个控件时 , 它可用于显示错误消息
ElementName	如果源对象是一个 WPF 元素 , 就可以用 ElementName 属性指定源对象
Path	使用 Path 属性 , 可以指定到源对象的路径。它可以是源对象的属性 , 也支持子元素的索引符和属性
XPath	使用 XML 数据源时 , 可以定义一个 XPath 查询表达式 , 来获得要绑定的数据
Mode	模式定义了绑定的方向。 Mode 属性是 BindingMode 类型。 BindingMode 是一个枚举 , 其值如下 : Default 、 OneTime 、 OneWay 、 TwoWay 、 OneWayToSource 。默认模式依赖于目标 : 对于文本框 , 双向绑定是默认值 ; 对于只读的标签 , 默认为单向 ; OneTime 表示数据仅从源中加载一次。 OneWay 还可以将对源对象的修改更新到目标对象中。 TwoWay 绑定表示 , 对 WPF 元素的修改可以写回源对象。 OneWayToSource 表示 , 从不读取数据 , 但总是从目标对象写入源对象
Converter	使用 Converter 属性 , 可以指定一个转换器类 , 来回转换 UI 的数据。转换器类必须实现接口 IValueConverter , 它定义了方法 Convert() 和 ConvertBack() 。 使用 ConverterParameter 属性可以给转换方法传递参数。转换器对文化很敏感 , 文化可以用 ConverterCulture 属性设置
Fallback Value	使用 Fallback Value 属性 , 可以定义一个在绑定没有返回值时使用的默认值
ValidationRules	使用 ValidationRules 属性 , 可以定义一个 ValidationRule 对象集合 , 在用 WPF 目标元素更新源对象之前检查该集合。 ExceptionValidationRule 类派生自 ValidationRule 类 , 负责检查异常

### 35.1.3 简单对象的绑定

要绑定 CLR 对象 , 只需使用.NET 类定义属性 , 如下面的例子就使用类 Book 定义了属性 Title 、 Publisher 、 Isbn 和 Authors :

```
public class Book
{
    public Book(string title, string publisher, string isbn,
    params string[] authors)
    {
        this.title = title;
        this.publisher = publisher;
        this.isbn = isbn;
        foreach (string author in authors)
        {
            this.authors.Add(author);
        }
    }

    public Book()
        : this("unknown", "unknown", "unknown")
    {
    }

    public string Title{ get; set; }
    public string Publisher{ get; set; }
    public string Isbn{ get; set; }
    public override string ToString()
    {
```

```
        return title;
    }

    private readonly List<string> authors = new
    List<string>();
    public string[] Authors
    {
        get { return authors.ToArray(); }
    }
}
```

在用户界面上，定义了几个标签和文本框控件，以显示图书信息。使用 Binding 标记扩展，将文本框控件绑定到 Book 类的属性上。在 Binding 标记扩展中，仅定义了 Path 属性，将它绑定到 Book 类的属性上。不需要定义源对象，因为源对象是通过指定 DataContext 来定义的，如下面的后台代码所示。对于 TextBox 元素，模式定义为默认值，即双向绑定：

```
<Window x:Class="ObjectBindingSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Object Binding Sample" Height="300" Width="340">

    <Grid Name="bookGrid" Margin="5,5,5,5" >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="30*" />
            <ColumnDefinition Width="70*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.Row="0">Title:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="0"
Text="{Binding Title}" />
        <Label Grid.Column="0" Grid.Row="1">Publisher:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="1"
Text="{Binding Publisher}" />
        <Label Grid.Column="0" Grid.Row="2">ISBN:</Label>
        <TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="2"
Text="{Binding Isbn}" />
        <Button Margin="5" Grid.Column="1" Grid.Row="4"
Click="bookButton_Click"
Name="bookButton">Open Dialog</Button>
    </Grid>
```

```
</Window>
```

在后台代码中定义了一个新的 Book 对象，并将其赋予 Grid 控件的 DataContext 属性。DataContext 是一个依赖属性，用基类 FrameworkElement 定义。指定 Grid 控件的 DataContext 属性表示，Grid 控件中的每个元素都默认绑定到同一个数据环境上。

```
public partial class Window1 : System.Windows.Window
{
    private Book book1 = new Book();
    public Window1()
    {
        InitializeComponent();
        book1.Title = "Professional C# 2005 with .NET C# 3.0";
        book1.Publisher = "Wrox Press";
        book1.Isbn = "978-0764575341";
        bookGrid.DataContext = book1;
    }
}
```

启动应用程序，就会看到图 35-4 所示的绑定数据。

为了实现双向绑定(对 WPF 元素的修改反映到 CLR 对象上)，定义了方法 OnOpenBookDialog()。这个方法指定为 bookButton 的 Click 事件，如下面的 XAML 代码所示。在执行时，会弹出一个消息框，显示 book1 对象的当前标题和 ISBN 号。图 35-5 显示了在运行过程中修改了一个输入后消息框的结果。

```
void OnOpenBookDialog(object sender, RoutedEventArgs e)
{
    string message = book1.Title;
    string caption = book1.Isbn;
    MessageBox.Show(message, caption);
}
```

图 35-4

图 35-5

#### 35.1.4 对象数据提供程序

除了在后台代码中定义对象之外，还可以用 XAML 定义对象实例。为此，必须引用在 XML 根元素的命名空间中声明的命名空间。XML 属性 xmlns:src="clr-namespace:Wrox. ProCsharp. WPF" 将.NET 命名空间 Wrox.ProCSharp.WPF 赋予 XML 命名空间别名 src。

现在，在 Window 资源中用 Book 元素定义 Book 类的一个对象。给 XML 属性 Title 和 Publisher 赋值，就设置了类 Book 的属性值。x:Key="theBook" 定义了资源的标识符，以便引用 book 对象。在 TextBox 元素中，Source 是用 Binding 标记扩展定义的，以引用 theBook 资源。

**提示：**

可以合并标记扩展。在下面的例子中，引用图书资源的 StaticResource 标记扩展包含在 Binding 标记扩展中。

```
<Window x:Class="ObjectBindingSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
Title="Object Binding Sample" Height="300" Width="340">
<Window.Resources>
<src:Book x:Key="theBook" Title="Professional C# 2008"
Publisher="Wrox Press" />
</Window.Resources>
<!-- ... -->
<TextBox Margin="5" Height="30" Grid.Column="1" Grid.Row="0"
Text="{Binding Source={StaticResource theBook}, Path=Title}" />
<!-- ... -->
```

**提示：**

如果要引用的.NET 命名空间在另一个程序集中，就必须把该程序集添加到 XML 声明中。

```
xmlns:system="clr-namespace:System;assembly=mscorlib".
```

除了直接在 XAML 代码中定义对象实例外，还可以定义一个对象数据提供程序，来引用类，调用方法。为了使用 ObjectDataProvider，最好创建一个返回要显示的对象的工厂类，如下面的 BookFactory 类所示：

```
public class BookFactory
{
    private List<Book> books = new List<Book>();
    public BookFactory()
    {
        books.Add(new Book("Professional C# 2008",
                           "Wrox Press", "978-0470191378"));
    }
    public Book GetTheBook()
    {
        return books[0];
    }
}
```

ObjectDataProvider 元素可以在资源段中定义。XML 属性 ObjectType 定义了类的名称，MethodName 指定了要调用以获得 book 对象的方法名：

```
<Window.Resources>
<ObjectDataProvider ObjectType="src:BookFactory"
MethodName="GetTheBook"
x:Key="theBook">
</ObjectDataProvider>
</Window.Resources>
```

可以用 ObjectDataProvider 类指定的属性如表 35-3 所示。

表 35-3

ObjectDataProvider	说    明
ObjectType	ObjectType 属性定义了要创建的实例类型
ConstructorParameters	使用 ConstructorParameters 集合可以在类中添加创建实例的参数
MethodName	MethodName 属性定义了由对象数据提供程序调用的方法名
MethodParameters	使用 MethodParameters 属性可以给通过 MethodName 属性定义的方法指定参数
ObjectInstance	使用 ObjectInstance 属性，可以获取和设置由 ObjectDataProvider 类使用的对象。例如，可以用编程方式指定已有的对象，以便用 ObjectDataProvider 实例化一个对象，而不是定义 ObjectType
Data	使用 Data 属性，可以访问用于数据绑定的底层对象。如果定义了 MethodName，则使用 Data 属性，可以访问从指定的方法返回的对象

### 35.1.5 列表绑定

绑定到列表上比绑定到简单对象上更灵活，但这两种绑定非常类似。可以在后台代码中将完整的列表赋予 DataContext，也可以使用 ObjectDataProvider 访问一个对象工厂，以返回一个列表。对支持绑定到列表上的元素(如列表框)，会绑定整个列表。对于只支持绑定一个对象的元素(如文本框)，就绑定当前的数据项。

使用 BookFactory 类，会返回一个 Book 对象列表：

```
public class BookFactory
{
    private List books = new List();
    public BookFactory()
    {
        books.Add(new Book("Professional C# 2008", "Wrox Press",
"978-0470191378", "Christian Nagel", "Bill Evjen",
"Jay Glynn", "Karli Watson", "Morgan Skinner"));
        books.Add(new Book("Professional C# 2005 with .NET 3.0",
"Wrox Press", "978-0-470-12472-7", "Christian Nagel",
"Bill Evjen", "Jay Glynn", "Karli Watson", "Morgan
Skinner"));
        books.Add(new Book("Professional C# 2005",
"Wrox Press", "978-0-7645-7534-1", "Christian Nagel",
```

```
"Bill Evjen", "Jay Glynn", "Karli Watson", "Morgan
Skinner",
"Allen Jones"));
books.Add(new Book("Beginning Visual C#",
"Wrox Press", "978-0-7645-4382-1", "Karli Watson",
"David Espinosa", "Zach Greenvoss", "Jacob Hammer
Pedersen",
"Christian Nagel", "John D. Reid", "Matthew Reynolds",
"Morgan Skinner", "Eric White"));
books.Add(new Book("ASP.NET Professional Secrets",
"Wiley", "978-0-7645-2628-2", "Bill Evjen",
"Thiru Thangarathinam", "Bill Hatfield", "Doug Seven",
"S. Srinivasa Sivakumar", "Dave Wanta", "Jason T. Roff"));
books.Add(new Book("Design and Analysis of Distributed
Algorithms",
" Wiley", "978-0-471-71997-7", "Nicolo Santoro"));
}
public List GetBooks()
{
return books;
}
}
```

在 WPF 后台代码中，类 Window1 的构造函数实例化了一个 BookFactory 类，并调用 GetBooks() 方法，将 Book 数组赋予 Window1 实例的 DataContext 属性：

```
public partial class Window1 : System.Windows.Window
{
private BookFactory factory = new BookFactory();
public Window1()
{
InitializeComponent();
this.DataContext = factory.GetBooks();
}
}
```

在 XAML 中，只需一个支持列表的控件，如列表框，绑定 ItemSource 属性，如下所示：

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="List Binding Sample" Height="300" Width="518"
>
```

```
Grid.Row="0" Grid.Column="0" Name="booksList"
ItemsSource="{Binding}" />
```

因为 Window 将 Book 数组赋予 DataContext，列表框放在 Window 中，所以列表框会用默认模板显示所有的图书，如图 35-6 所示。

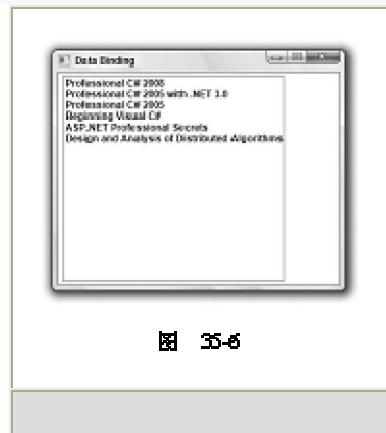


图 35-6

为了使列表框有更灵活的布局，必须定义一个模板，就像前一章为列表框定义样式那样。样式 listBoxStyle 包含的模板 ItemTemplate 定义了一个带标签元素的 DataTemplate。标签的内容绑定到 Title 上。数据项模板重复应用于列表中的每一项。

列表框元素指定了 Style 属性。ItemsSource 也设置为默认绑定。图 35-7 显示了使用新的列表框样式后应用程序的输出。

```
Style="{StaticResource listBoxStyle}" Grid.RowSpan="4"  
ItemsSource="{Binding}" />
```

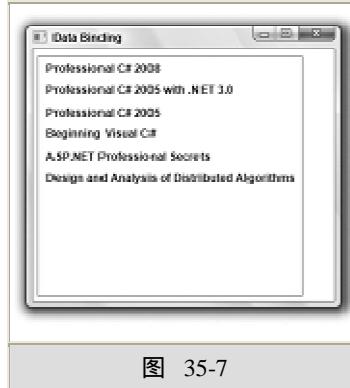


图 35-7

### 1. 主从绑定

除了显示列表中的所有元素之外，还应能显示选中项的详细信息。这不需要做太多的工作。只需将元素定义为显示当前选项即可。在示例程序中，用 Binding 标记扩展将三个标签元素分别定义为 Book 的属性 Title、Publisher 和 Isbn。这里必须对列表框进行一个重要的修改。在默认情况下，标签绑定到列表的第一项上。设置列表框的属性 IsSynchronizedWithCurrentItem = "True"，就会把列表框的选项设置为当前项。在图 35-8 中显示了结果：选中的项显示在详细信息标签中。

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Title="List Binding Sample" Height="300" Width="518"  
>  
<Window.Resources>  
<Style x:Key="listBoxStyle" TargetType="{x:Type ListBox}" >  
<Setter Property="ItemTemplate">  
<Setter.Value>  
<DataTemplate>  
<Label Content="{Binding Title}" />  
</DataTemplate>  
</Setter.Value>  
</Setter>  
</Style>  
<Style x:Key="labelStyle" TargetType="{x:Type Label}">  
<Setter Property="Width" Value="190" />  
<Setter Property="Height" Value="40" />  
<Setter Property="Margin" Value="5,5,5,5" />  
</Style>  
</Window.Resources>  
<DockPanel>  
<Grid >
```

```
<Grid.ColumnDefinitions>
<ColumnDefinition />
<ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition />
<RowDefinition />
<RowDefinition />
<RowDefinition />
</Grid.RowDefinitions>
<ListBox IsSynchronizedWithCurrentItem="True"
HorizontalAlignment="Left"
Margin="5" Style="{StaticResource listBoxStyle}"
Grid.RowSpan="4" ItemsSource="{Binding}" />
<Label Style="{StaticResource labelStyle}" Content="{Binding
Title}" Grid.Row="0" Grid.Column="1" />
<Label Style="{StaticResource labelStyle}" Content="{Binding
Publisher}" Grid.Row="1" Grid.Column="1" />
<Label Style="{StaticResource labelStyle}" Content="{Binding Isbn}"
Grid.Row="2" Grid.Column="1" />
</Grid>
</DockPanel>
</Window>
```

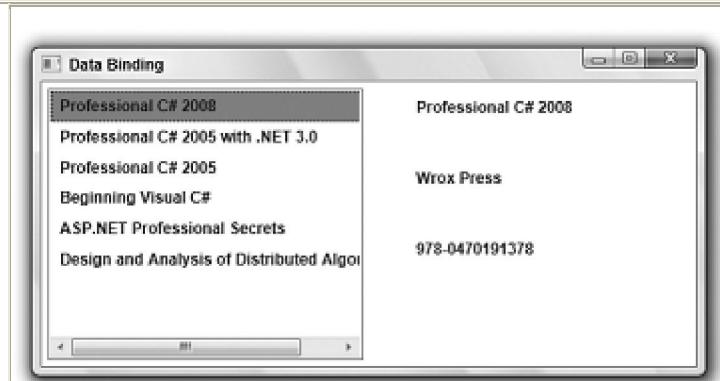


图 35-8

## 2. 值的转换

图书的作者还没有显示在结果中。如果将 Authors 属性绑定到标签元素上，就要调用 Array 类的 ToString()方法，它只返回类型的名称。一种解决方法是将 Authors 属性绑定到一个列表框上。对于列表框，可以定义一个模板，显示特定的视图。另一种解决方法是将 Authors 属性返回的字符串数组转换为一个字符串，再将该字符串用于绑定。

类 `StringArrayConverter` 可以将字符串数组转换为字符串。WPF 转换器类必须实现命名空间 `System.Windows.Data` 中的接口 `IValueConverter`。这个接口定义了方法 `Convert()` 和 `ConvertBack()`。在 `StringArrayConverter` 中，`Convert()` 方法会通过 `String.Join()` 方法把 `value` 变量中的字符串数组转换为字符串。`Join()` 方法的分隔符参数值从 `Convert()` 方法返回的 `parameter` 变量中提取。

提示：

String 类的方法的更多信息参见第 8 章。

```
public class StringArrayConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter,
        System.Globalization.CultureInfo culture)
    {
        string[] stringCollection = (string[])value;
        string separator = (string)parameter;
        return String.Join(separator, stringCollection);
    }

    public object ConvertBack(object value, Type targetType,
        object parameter,
        System.Globalization.CultureInfo culture)
    {
        string s = (string)value;
        char separator = (char)parameter;
        return s.Split(separator);
    }
}
```

在 XAML 代码中，`StringArrayConverter` 类可以声明为一个资源，以便在 `Binding` 标记扩展中引用它：

```
<Window x:Class="Wrox.ProCSharp.WPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
    Title="List Binding Sample" Height="300" Width="518"
    >

    <Window.Resources>
        <src:StringArrayConverter x:Key="stringArrayConverter" />
    <!-- ... -->
```

为了输出多个结果，声明一个 `TextBlock` 元素，其 `TextWrapping` 属性设置为 `Wrap`，以便显示多个作者。在 `Binding` 标记扩展中，`Path` 设置为 `Authors`，它定义为一个返回字符串数组的属性。`Converter` 属性指定字符串数组从资源 `stringArrayConverter` 中转换。转换器的 `Convert` 方法将 `ConverterParameter` 参数'作为输入来分隔多个作者。

```
<TextBlock Width="190" Height="50" Margin="5"
TextWrapping="Wrap"
Text="{Binding Path=Authors,
Converter={StaticResource stringArrayConverter},
ConverterParameter=' '}"
Grid.Row="3" Grid.Column="1" />
```

图 35-9 显示了图书的细节，包括作者。

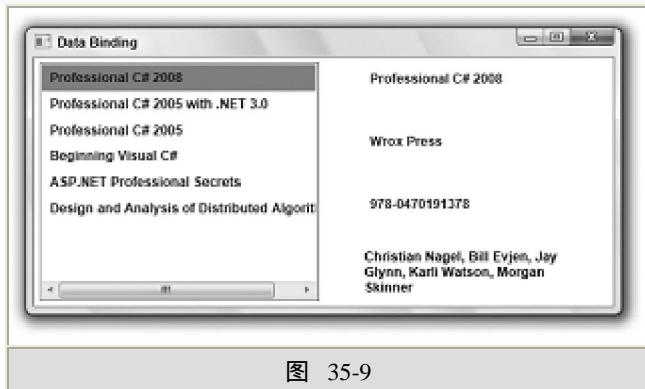


图 35-9

### 3. 动态添加列表项

如果列表项要动态添加，该怎么办？必须通知 WPF 元素：要在列表中添加元素。

在 WPF 应用程序的 XAML 代码中，要给 StackPanel 添加一个按钮元素。给 Click 事件指定方法 OnAddBook()：

```
<!-- ... -->
<DockPanel>
<StackPanel Orientation="Horizontal"
DockPanel.Dock="Bottom" Height="60">
<Button Click=" addBookButton_Click "
Name="addBookButton"
Margin="5" Width="80" Height="40">Add Book</Button>
</StackPanel>
<Grid >
<!-- ... -->
```

在方法 OnAddBook()中，包含了 addBookButton()方法的事件处理程序代码，这个方法将一个新的 Book 对象添加到列表中。如果用 BookFactory 测试应用程序(因为它已实现)，不会通知 WPF 元素：已在列表中添加了一个新对象。

```
void OnAddBook(object sender, RoutedEventArgs e)
{
    factory.AddBook(new Book(".NET 2.0 Wrox Box", "Wrox
Press",
"978-0-470-04840-5"));
}
```

赋予 DataContext 的对象必须实现接口 INotifyCollectionChanged。这个接口定义了由 WPF 应用程序使用的 CollectionChanged 事件。除了用定制的集合类实现这个接口之外，还可以使用泛型集合类 ObservableCollection<T>，该类在 WindowsBase 程序集的 System.Collections.ObjectModel 命名空间中定义。现在，把一个新数据项添加到集合中，这个新数据项会立即显示在列表框中。

```
public class BookFactory
{
    private ObservableCollection<Book> books = new
    ObservableCollection<Book>();
    // ...
    public void AddBook(Book b)
    {
        books.Add(b);
    }
    public ObservableCollection<Book> GetBooks()
    {
        return books;
    }
}
```

#### 4. 数据模板

上一章介绍了如何用模板来定制控件。还可以为数据类型定义模板，例如 Book 类。无论在何处使用 Book 类，都使用该模板定义默认外观。

在下面的例子中，在 Window 资源内部定义 DataTemplate。DataType 属性引用命名空间 Wrox.ProCSharp.WPF 中的 Book 类。该模板定义，在 StackPanel 中包含一个边框和两个标签元素。列表框元素并没有引用该模板。列表框定义的唯一一个属性是 ItemsSource，其值是默认的 Binding 标记扩展。因为 DataTemplate 没有定义键，所以由包含 Book 对象的所有列表使用。图 35-10 显示了使用数据模板的应用程序的结果。

```
<Window x:Class="Wrox.ProCSharp.WPF.DataTemplateDemo"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:src="clr-namespace:Wrox.ProCSharp.WPF"
Title="Data Template Sample" Height="300" Width="300"
>
<Window.Resources>
<DataTemplate DataType="{x:Type src:Book}">
<Border BorderBrush="Blue" BorderThickness="2"
Background="LightBlue"
Margin="10" Padding="15">
<StackPanel>
<Label Content="{Binding Path=Title}" />
<Label Content="{Binding Path=Publisher}" />
</StackPanel>
```

```
</Border>
</DataTemplate>
</Window.Resources>
<Grid>
<ListBox ItemsSource="{Binding}" />
</Grid>
</Window>
```

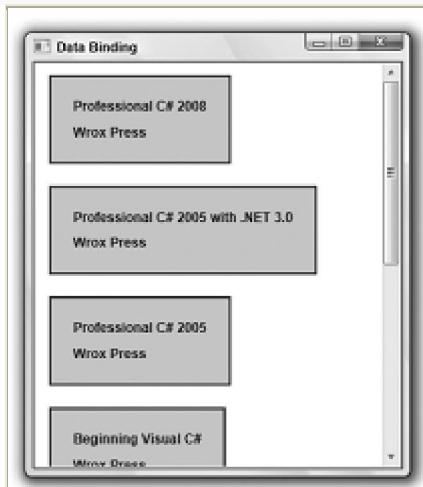


图 35-10

如果要对相同的数据类型使用另一个数据模板，可以创建一个数据模板选择器。数据模板选择器在派生于基类 DataTemplateSelector 的类中实现。

下面的数据模板选择器根据发布者选择另一个模板。在 Window 资源中，定义了这些模板。一个模板可以通过键名 WroxBookTemplate 来访问，另一个模板的键名是 WileyBookTemplate：

```
< DataTemplate x:Key="WroxBookTemplate"
  DataType="{x:Type src:Book}" >
  < Border BorderBrush="Blue" BorderThickness="2"
    Background="LightBlue"
    Margin="10" Padding="15" >
    < StackPanel >
      < Label Content="{Binding Path=Title}" />
      < Label Content="{Binding Path=Publisher}" />
    < /StackPanel >
  < /Border >
< /DataTemplate >

< DataTemplate x:Key="WileyBookTemplate"
  DataType="{x:Type src:Book}" >
  < Border BorderBrush="Yellow" BorderThickness="2"
    Background="LightGreen" Margin="10" Padding="15" >
    < StackPanel >
      < Label Content="{Binding Path=Title}" />
    < /StackPanel >
  < /Border >
< /DataTemplate >
```

```
< Label Content="{Binding Path=Publisher}" / >
< /StackPanel >
< /Border >
< /DataTemplate >
```

要选择模板，类 BookDataTemplateSelector 必须重写基类 DataTemplateSelector 中的 SelectTemplate 方法。其实现代码根据 Book 类中的 Publisher 属性选择模板：

```
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.WPF
{
    public class BookDataTemplateSelector : DataTemplateSelector
    {
        public override DataTemplate SelectTemplate(object item,
            DependencyObject container)
        {
            if (item != null && item is Book)
            {
                Window window = Application.Current.MainWindow;

                Book book = item as Book;
                switch (book.Publisher)
                {
                    case "Wrox Press":
                        return window.FindResource("WroxBookTemplate")
                            as DataTemplate;
                    case "Wiley":
                        return window.FindResource("WileyBookTemplate")
                            as DataTemplate;
                    default:
                        return window.FindResource("BookTemplate") as
                            DataTemplate;
                }
            }
            return null;
        }
    }
}
```

要在 XAML 代码中访问类 BookDataTemplateSelector，这个类必须在 Window 资源中定义：

```
< src:BookDataTemplateSelector
x:Key="bookTemplateSelector" / >
```

现在选择器类可以赋予 ListBox 的 ItemTemplateSelector 属性：

```
< ListBox ItemsSource="{Binding}"  
ItemTemplateSelector="{StaticResource  
bookTemplateSelector}" />
```

运行这个应用程序，可以看到不同发布者的不同数据模板，如图 35-11 所示。

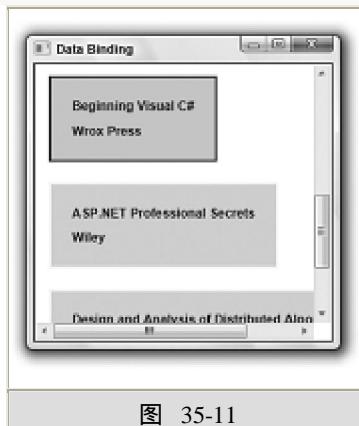


图 35-11

### 35.1.6 绑定到 XML 上

WPF 数据绑定还支持绑定到 XML 数据上。可以将 XmlDataProvider 用作数据源，使用 XPath 表达式绑定元素。为了分层次显示，可以使用 TreeView 控件，通过 HierarchicalData Template 为数据项创建视图。

下面包含 Book 元素的 XML 文件将用作后面例子的数据源：

```
<?xml version="1.0" encoding="utf-8" ?>  
<Books>  
  <Book isbn="978-0-470-12472-7">  
    <Title>Professional C# 2008</Title>  
    <Publisher>Wrox Press</Publisher>  
    <Author>Christian Nagel</Author>  
    <Author>Bill Evjen</Author>  
    <Author>Jay Glynn</Author>  
    <Author>Karli Watson</Author>  
    <Author>Morgan Skinner</Author>  
  </Book>  
  <Book isbn="978-0-7645-4382-1">  
    <Title>Beginning Visual C# 2008</Title>  
    <Publisher>Wrox Press</Publisher>  
    <Author>Karli Watson</Author>  
    <Author>David Espinosa</Author>  
    <Author>Zach Greenvoss</Author>  
    <Author>Jacob Hammer Pedersen</Author>  
    <Author>Christian Nagel</Author>  
    <Author>John D. Reid</Author>
```

```
<Author>Matthew Reynolds</Author>
<Author>Morgan Skinner</Author>
<Author>Eric White</Author>
</Book>
</Books>
```

与定义对象数据提供程序类似，也可以定义 XmlDataProvider。ObjectDataProvider 和 XmlDataProvider 都派生自同一个基类 DataSourceProvider。在示例的 XmlDataProvider 中，Source 属性设置为引用 XML 文件 books.xml。XPath 属性定义了一个 XPath 表达式，以引用 XML 根元素 Books。Grid 元素通过 DataContext 属性引用 XML 数据源。在栅格的 DataContext 属性中，所有的 Book 元素都需要列表绑定，所以 XPath 表达式设置为 Book。在栅格中，把列表框元素绑定到默认的数据环境中，并使用 DataTemplate 将标题包含在 TextBlock 元素中，作为列表框的数据项。在栅格中，还有三个标签元素，它们的数据绑定设置为 XPath 表达式，以显示标题、出版社和 ISBN 号。

```
<Window x:Class="XmlBindingSample.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="XmlBindingSample" Height="348" Width="498"
>
<Window.Resources>
<XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
<DataTemplate x:Key="listTemplate">
<TextBlock Text="{Binding XPath=Title}" />
</DataTemplate>
<Style x:Key="labelStyle" TargetType="{x:Type Label}">
<Setter Property="Width" Value="190" />
<Setter Property="Height" Value="40" />
<Setter Property="Margin" Value="5,5,5,5" />
</Style>
</Window.Resources>
<Grid DataContext="{Binding Source={StaticResource books},
XPath=Book}">
<Grid.ColumnDefinitions>
<ColumnDefinition />
<ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition />
<RowDefinition />
<RowDefinition />
<RowDefinition />
</Grid.RowDefinitions>
<ListBox IsSynchronizedWithCurrentItem="True" Margin="5,5,5,5"
Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
ItemTemplate="{StaticResource listTemplate}" />
```

```
<Label Style="{StaticResource labelStyle}" Content="{Binding  
XPath=Title}"  
Grid.Row="0" Grid.Column="1" />  
<Label Style="{StaticResource labelStyle}" Content="{Binding  
XPath=Publisher}"  
Grid.Row="1" Grid.Column="1" />  
<Label Style="{StaticResource labelStyle}" Content="{Binding  
XPath=@isbn}"  
Grid.Row="2" Grid.Column="1" />  
</Grid>  
</Window>
```

图 35-12 显示了 XML 绑定的结果。

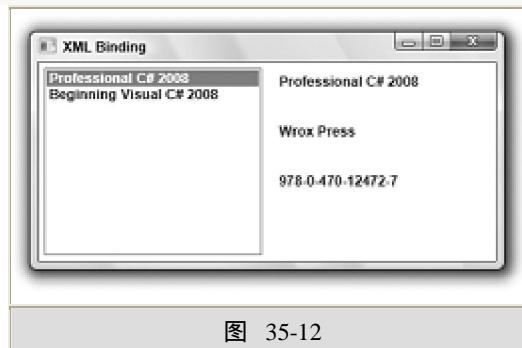


图 35-12

如果 XML 数据应以层次结构的方式显示，就可以使用 TreeView 控件。

### 35.1.7 绑定的验证

在把数据用于.NET 对象之前，有几个选项可验证用户的数据，这些选项如下：

处理异常

数据错误信息

定制验证规则

#### 1. 处理异常

这里演示的一个选项是如果在类 SomeData 中设置了无效值，这个.NET 类就抛出一个异常。属性 Value1 只接受大于等于 5 且小于 12 的值：

```
public class SomeData  
{  
    private int value1;  
    public int Value1 {  
        get  
        {  
            return value1;  
        }  
        set
```

```
{  
if (value < 5 || value > 12)  
throw new ArgumentException(  
"value must not be less than 5 or greater than 12");  
value1 = value;  
}  
}  
}
```

在 Window1 类的构造函数中，初始化 SomeData 类的一个新对象，并传送给 DataContext，用于数据绑定：

```
public partial class Window1 : Window  
{  
SomeData p1 = new SomeData() { Value1 = 11 };  
public Window1()  
{  
InitializeComponent();  
this.DataContext = p1;  
}
```

事件处理程序方法 buttonSubmit\_Click 显示一个消息框，列出 SomeData 实例的实际值：

```
private void buttonSubmit_Click(object sender,  
RoutedEventArgs e)  
{  
MessageBox.Show(p1.Value1.ToString());  
}
```

在简单数据绑定中，把文本框的 Text 属性绑定到 Value1 属性上。如果现在运行应用程序，试图把值改为某个无效值，单击 Submit 按钮，就可以验证该值永远不会改变。WPF 会捕获并忽略属性 Value1 的 set 访问器抛出的异常。

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1:  
< /Label >  
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"  
Text="{Binding Path=Value1}" / >
```

要在输入字段的内容发生变化时显示错误，可以把 Binding 标记扩展的 ValidatesOnException 属性设置为 True。输入一个无效值(设置该值时，会抛出一个异常)，文本框就会以红色线条框出，如图 35-13 所示。

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1:  
< /Label >  
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"
```

```
Text="{Binding Path=Value1,  
ValidatesOnExceptions=True}" />
```

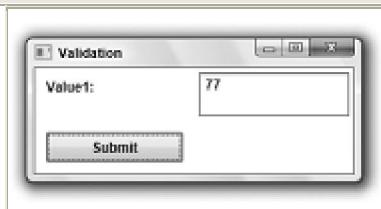


图 35-13

要以另一种方式给用户返回错误信息，可以把 Validation 类定义的关联属性 ErrorTemplate 赋予一个为错误定义 UI 的模板。标记错误的新模板用键 validationTemplate 表示。ControlTemplate 在已有的控件内容前面添加了一个红色的感叹号。

```
< ControlTemplate x:Key="validationTemplate" >  
< DockPanel >  
< TextBlock Foreground="Red" FontSize="20" > ! <  
/TextBlock >  
< AdornedElementPlaceholder />  
< /DockPanel >  
< /ControlTemplate >
```

用 Validation.ErrorTemplate 关联属性设置 validationTemplate 会激活带文本框的模板：

```
< Label Margin="5" Grid.Row="0" Grid.Column="0" > Value1:  
< /Label >  
< TextBox Margin="5" Grid.Row="0" Grid.Column="1"  
Text="{Binding Path=Value1,  
ValidatesOnExceptions=True}"  
Validation.ErrorTemplate="{StaticResource  
validationTemplate}" />
```

应用程序的新界面如图 35-14 所示。

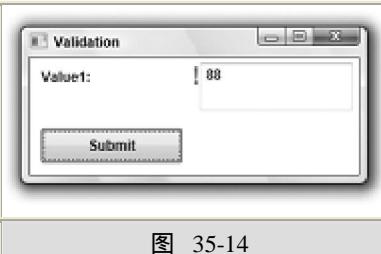


图 35-14

提示：

定制错误消息的另一个选项是注册 Validation 类的 Error 事件。这里必须把属性 NotifyOnValidationError 设置为 true。

错误信息可以在 Validation 类的 Errors 集合中访问。要在文本框的工具提示中显示错误信息，可以创建一个属性触发器，如下所示。只要 Validation 类的 HasError 属性设置为 True，就激活触发器。  
触发器设置文本框的 ToolTip 属性：

```
< Style TargetType="" >
< Style.Triggers >
< Trigger Property="Validation.HasError" Value="True" >
< Setter Property="ToolTip"
Value="{Binding RelativeSource=
{x:Static RelativeSource.Self},
Path=(Validation.Errors)[0].ErrorContent}" / >
< /Trigger >
< /Style.Triggers >
< /Style >
```

## 2. 数据错误信息

处理错误的另一种方式是确定.NET 对象是否执行了接口 IDataErrorInfo。

类 SomeData 现在改为执行接口 IDataErrorInfo。这个接口定义了属性 Error 和带字符串参数的索引器。在数据绑定的过程中验证 WPF 时，会调用索引器，把要验证的属性名传递为 columnName 参数。在执行代码中，会验证其值是否有效，如果无效就传送一个错误字符串。

下面验证属性 Value2，它是使用 C# 3.0 简单属性标记实现的。

```
public class SomeData : IDataErrorInfo
{
    private int value1;
    public int Value1 {
        get
        {
            return value1;
        }
        set
        {
            if (value < 5 || value > 12)
                throw new ArgumentException(
                    "value must not be less than 5 or greater than 12");
            value1 = value;
        }
    }
    public int Value2 { get; set; }

    string IDataErrorInfo.Error
    {
        get
        {
```

```
return null;
}
}
string IDataErrorInfo.this[string columnName]
{
get
{
if (columnName == "Value2")
{
if (this.Value2 < 0 || this.Value2 > 80)
return "age must not be less than 0 or greater than 80";
}
return null;
}
}
```

#### 提示：

在.NET 实体类中，索引器返回什么内容并不清楚，例如调用索引器，会从 Person 类型的对象中返回什么？因此最好在接口 IDataErrorInfo 中包含显式的执行代码。这样，这个索引器只能使用接口来访问，.NET 类可以有另一个执行方式，以实现其他目的。

如果把 Binding 类的 ValidationOnDataErrors 属性设置为 true，就在数据绑定的过程中使用接口 IDataErrorInfo。这里，改变文本框时，绑定机制会调用接口的索引器，把 Value2 传送给 columnName 变量：

```
< Label Margin="5" Grid.Row="1" Grid.Column="0" > Value2:
< /Label >
< TextBox Margin="5" Grid.Row="1" Grid.Column="1"
Text="{Binding Path=Value2,
ValidatesOnDataErrors=True}" / >
```

### 3. 定制验证规则

为了更多地控制验证，可以实现定制验证规则。实现了定制验证规则的类必须派生于基类 ValidationRule。在前面的两个例子中，也使用了验证规则。派生于 ValidationRule 抽象基类的两个类是 DataErrorValidationRule 和 ExceptionValidationRule。设置属性 Validates- OnDataErrors，使用接口 IDataErrorInfo，就激活了 DataErrorValidationRule。Exception- ValidationRule 处理异常，设置 ValidationOnException 会激活 ExceptionValidationRule。

下面实现一个验证规则，来验证正则表达式。类 RegularExpressionValidationRule 派生于基类 ValidationRule，重写了基类定义的抽象方法 Validate()。在其执行代码中，使用命名空间 System.Text.RegularExpressions 中的 RegEx 类验证 Expression 属性定义的表达式。

```
public class RegularExpressionValidationRule :  
ValidationRule  
{  
    public string Expression { get; set; }  
    public string ErrorMessage { get; set; }  
  
    public override ValidationResult Validate(object value,  
CultureInfo cultureInfo)  
{  
    ValidationResult result = null;  
    if (value != null)  
    {  
        Regex regEx = new Regex(Expression);  
        bool isMatch = regEx.IsMatch(value.ToString());  
        result = new ValidationResult(isMatch, isMatch ?  
null : ErrorMessage);  
    }  
    return result;  
}  
}
```

这里没有使用 Binding 标记扩展，而是把绑定作为 TextBox.Text 元素的一个子元素。绑定的对象现在定义了 Email 属性，它用简单的属性语法来实现。UpdateSourceTrigger 属性定义了源代码何时更新。更新源代码的选项如下：

属性值变化时更新，可以是用户输入的每个字符时更新

焦点失去时更新

显式指定更新时间

ValidationRules 是 Binding 类的一个属性，它包含 ValidationRule 元素。这里使用的验证规则是定制类 RegularExpressionValidationRule，其中 Expression 属性设置为一个正则表达式，用于验证输入是否是有效的电子邮件，ErrorMessage 属性给出 TextBox 的输入数据无效时显示的错误消息：

```
< Label Margin="5" Grid.Row="2" Grid.Column="0" > Email: < /Label  
>  
< TextBox Margin="5" Grid.Row="2" Grid.Column="1" >  
< TextBox.Text >  
< Binding Path="Email" UpdateSourceTrigger="LostFocus" >  
< Binding.ValidationRules >  
< src:RegularExpressionValidationRule  
Expression="([\w-\.]+)@([\[\[0-9\]\{1,3\}\.\[0-9\]\{1,3\}\.\[0-9\]\{1,3\}\.]\|(([\w-\.]+)\.)([a-zA-Z]\{2,4\}|[0-9]\{1,3\})(\.)?)$" ErrorMessage="Email is not valid" / >  
< /Binding.ValidationRules >  
< /Binding >
```

```
< /TextBox.Text >  
< /TextBox >
```

## 35.2 命令绑定

WPF 的 Menu 和 ToolBar 控件与 Windows 窗体的对应控件有相同的功能：启动命令。使用这些控件，可以添加事件处理程序，执行命令。启动命令还有其他方式：选择菜单，单击工具栏按钮，按下键盘上的某个键。为了处理这些不同的输入方式，WPF 提供了另一个功能：命令。

一些 WPF 控件还提供了预定义命令的执行代码，更便于获得某些功能。

WPF 通过命令类提供了一些预定义的命令：ApplicationCommands、EditingCommands、ComponentCommands 和 NavigationCommands。这些命令类都是静态类，其静态属性返回 RoutedUICommand 对象。例如，ApplicationCommands 的属性包括 New、Open、Save、SaveAs、Print 和 Close。许多应用程序都有这些命令。

为了说明这些命令的功能，创建一个简单的 WPF 项目，添加一个 Menu 控件，其菜单项包含 Undo、Redo、Cut、Copy 和 Paste。文本框 textContent 占用了窗口中的剩余空间，可以输入多行文本。在窗口中，创建了一个 DockPanel，来定义布局。使带有 MenuItem 元素的 Menu 控件停靠在顶边，标题设置为定义菜单的文本。\_ 定义了无需使用鼠标、可以直接用键盘访问菜单项的字母。按下 Alt 键时，标题文本中的字母下面就会出现下划线。Command 属性定义了与菜单项关联的命令。

```
<Window x:Class="Wrox.ProCSharp.WPF.WPFEditionWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="WPF Editor" Height="300" Width="300"  
>  
< DockPanel >  
< Menu DockPanel.Dock="Top" >  
< MenuItem Header="_Edit" >  
< MenuItem Name="editUndoMenu" Header="_Undo"  
          Command="ApplicationCommands.Undo" />  
< MenuItem Name="editRedoMenu" Header="_Redo"  
          Command="ApplicationCommands.Redo" />  
< Separator />  
< MenuItem Name="editCutMenu" Header="Cu_t"  
          Command="ApplicationCommands.Cut" />  
< MenuItem Name="editCopyMenu" Header="_Copy"  
          Command="ApplicationCommands.Copy" />  
< MenuItem Name="editPasteMenu" Header="_Paste"  
          Command="ApplicationCommands.Paste" />  
< /MenuItem >  
< /Menu >  
< TextBox Name="textContent" TextWrapping="Wrap"  
          AcceptsReturn="True" >
```

```
AcceptsTab="True" />
</DockPanel>
</Window>
```

这就是要为剪切板功能做的工作。 TextBox 类已经包含了这些预定义命令绑定的功能。启动应用程序，在文本框中输入文本时，就可以看到菜单项。在文本框中选择文本，可以使用 Cut 和 Copy 菜单项。图 35-15 显示了运行着的应用程序。

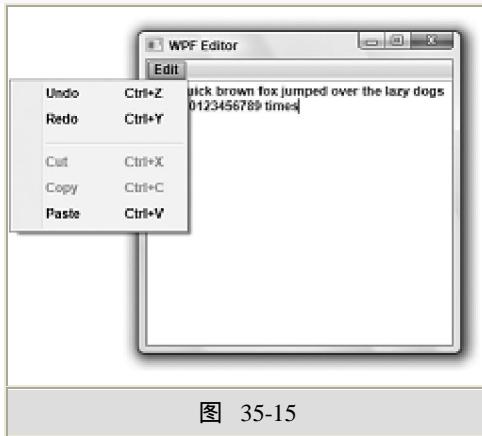


图 35-15

现在修改应用程序，添加前面没有用控件间定义的命令绑定。把打开和保存文件的命令添加到编辑器中。

为了使命令可以访问，应在 MenuItem 元素中添加更多的 MenuItem 元素，如下所示：

```
< MenuItem Header="_File" >
< MenuItem Name="fileNewMenu" Header="_New"
Command="ApplicationCommands.New" />
< MenuItem Name="fileOpenMenu" Header="_Open"
Command="ApplicationCommands.Open" />
< Separator />
< MenuItem Name="fileSave" Header="_Save"
Command="ApplicationCommands.Save" />
< MenuItem Name="fileSaveAs" Header="Save _As"
Command="ApplicationCommands.SaveAs" />
</ MenuItem >
```

命令也可以从工具栏中访问。对于ToolBar 元素，定义了可以从工具栏上访问的命令。为了安排工具栏，将 ToolBar 元素放在 ToolBarTray 中：

```
<ToolBarTray DockPanel.Dock="Top">
<ToolBar>
<Button Command="ApplicationCommands.New">
<Image Source="toolbargraphics/New.bmp" />
</Button>
<Button Command="ApplicationCommands.Open">
<Image Source="toolbargraphics/Open.bmp" />
</Button>
```

```
<Button Command="ApplicationCommands.Save">
<Image Source="toolbargraphics/Save.bmp" />
</Button>
</ToolBar>
</ToolBarTray>
```

现在需要定义命令绑定，把命令关联到事件处理程序上。命令绑定可以赋予派生自层次结构中最上面的 UIElement 基类的任意 WPF 类。可以定义 CommandBinding 元素，将命令绑定赋予 CommandBindings 属性。CommandBinding 类的 Command 属性可以指定一个实现了 ICommand 接口的对象、为事件 CanExecute 和 Executed 指定事件处理程序。示例程序把命令绑定赋予 Window 类。Executed 事件设置为实现命令功能的事件处理程序方法。如果命令不应在所有情况下都是可用的，就可以将事件 CanExecute 设置为一个处理程序，指定命令是否可用。

```
< Window.CommandBindings >
< CommandBinding Command="ApplicationCommands.New"
Executed="NewFileExecuted" / >
< CommandBinding Command="ApplicationCommands.Open"
Executed="OpenFileExecuted" / >
< CommandBinding Command="ApplicationCommands.Save"
Executed="SaveFileExecuted"
CanExecute="SaveFileCanExecute" / >
< CommandBinding Command="ApplicationCommands.SaveAs"
Executed="SaveAsFileExecuted"
CanExecute="SaveFileCanExecute" / >
< /Window.CommandBindings >
```

在处理程序方法的后台代码中，NewFileExecuted()会清空文本框，将文件名 untitled.txt 写入 Window 类的 Title 属性。在 OpenFileExecuted()中，创建了 Microsoft.Win32.OpenFileDialog，并显示为一个对话框。成功退出对话框后，会打开所选的文件，将其内容写入 TextBox 控件。

提示：

打开文件的对话框在 WPF 中没有预定义。可以选择文件和文件夹创建一个定制窗口，也可以使用 Microsoft.Win32 命名空间中的 OpenFileDialog 类，它封装在 Windows 对话框中。

```
public partial class Window1 : System.Windows.Window
{
    private string fileName;
    private readonly string defaultFileName;
    private const string appName = "WPF Editor";
    private bool isChanged = false;

    public Window1()
    {
        defaultFileName = System.IO.Path.Combine(
            Environment.GetFolderPath(
```

```
Environment.SpecialFolder.MyDocuments),  
@"untitled.txt");  
InitializeComponent();  
NewFile();  
}  
  
private void NewFileExecuted(object sender,  
ExecutedRoutedEventArgs e)  
{  
NewFile();  
}  
  
private void NewFile()  
{  
textContent.Clear();  
filename = defaultFilename;  
SetTitle();  
isChanged = false;  
}  
  
private void SetTitle()  
{  
Title = String.Format("{0} {1}",  
System.IO.Path.GetFileName(filename), appName);  
}  
private void OpenFileExecuted  
(object sender, ExecutedRoutedEventArgs e)  
{  
try  
{  
OpenFileDialog dlg = new OpenFileDialog();  
bool? dialogResult = dlg.ShowDialog();  
if (dialogResult == true)  
{  
filename = dlg.FileName;  
SetTitle();  
textContent.Text = File.ReadAllText(filename);  
}  
}  
catch (IOException ex)  
{  
MessageBox.Show(ex.Message, "Error WPF Editor",  
MessageBoxButton.OK, MessageBoxIcon.Error);  
}  
}
```

处理程序 SaveFileCanExecute()根据内容是否有变化，确定保存文件的命令是否可用：

```
private void SaveFileCanExecute(object sender,
    CanExecuteRoutedEventArgs e)
{
    if (isChanged)
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

应用程序打开了文件 sample.txt , 如图 35-16 所示。



图 35-16

### 35.3 动画

在动画中，可以使用移动的元素、通过颜色的改变、变换等制作顺畅的切换效果。WPF 使动画的制作非常简单。还可以连续改变任意依赖属性的值。不同的动画类可以根据其类型，连续改变不同属性的值。

动画的主要元素如下：

**时间线**：定义了值随时间的变化方式。有不同类型的时间线，可用于改变不同类型的值。所有时间线的基类都是 Timeline。为了连续改变 double，可以使用 DoubleAnimation 类。Int32Animation 类是 int 值的动画类。

**情节板**：用于合并动画。Storyboard 类派生自基类 TimelineGroup，TimelineGroup 又派生自基类 Timeline。使用 DoubleAnimation，可以连续改变 double，使用 Storyboard 类可以合并所有的动画。

**触发器**：通过触发器可以启动和停止动画。前面介绍了属性触发器。当属性值变化时，属性触发器就会启动。还可以创建事件触发器，当事件发生时，事件触发器就会启动。

**提示**：

动画类的命名空间是 System.Windows.Media.Animation。

### 35.3.1 时间线

时间线定义了值随时间变化的方式。第一个示例连续改变椭圆的大小。其中 DoubleAnimation 就是时间线，它改变了所使用的 double。Ellipse 类的 Triggers 属性设置为 EventTrigger。椭圆加载时，就触发用 EventTrigger 的 RoutedEvent 属性定义的事件触发器。BeginStoryboard 是启动故事板的启动操作。在故事板中，DoubleAnimation 元素用于连续改变 Ellipse 类的 Width 属性。动画在 3 秒内把椭圆的宽度从 100 改为 300，在之后的 3 秒内再改回来。

```
< Window x:Class="EllipseAnimation.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ellipse Animation" Height="300" Width="300" >
< Grid >
< Ellipse Height="50" Width="100" Fill="SteelBlue" >
< Ellipse.Triggers >
< EventTrigger RoutedEvent="Ellipse.Loaded" >
< EventTrigger.Actions >
< BeginStoryboard >
< Storyboard Duration="00:00:06" RepeatBehavior="Forever" >
< DoubleAnimation
  Storyboard.TargetProperty="(Ellipse.Width)"
  Duration="0:0:3" AutoReverse="True"
  FillBehavior="Stop" RepeatBehavior="Forever"
  AccelerationRatio="0.9" DecelerationRatio="0.1"
  From="100" To="300" / >
< /Storyboard >
< /BeginStoryboard >
< /EventTrigger.Actions >
< /EventTrigger >
< /Ellipse.Triggers >
< /Ellipse >
< /Grid >
< /Window >
```

图 35-17 和 35-18 显示了连续改变的椭圆的两个状态。

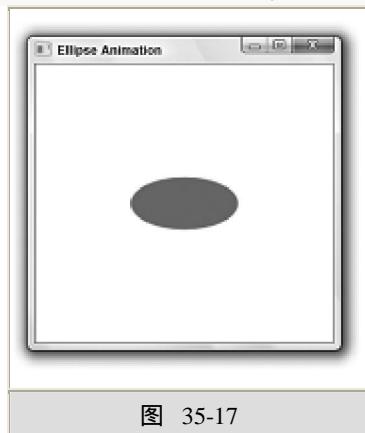


图 35-17

图 35-18

动画并不仅仅是连续显示在屏幕上的一般窗口动画。还可以给业务应用程序添加动画，使用户界面的响应更好。

下面的例子演示了一个相当好的动画，还说明了如何在样式中定义动画。在 Window 资源中，有一个用于按钮的样式 AnimatedButtonStyle。在模板中定义了一个矩形 outline。这个模板使用很细的笔触，其宽度设置为 0.4。

该模板为 IsMouseOver 属性定义了一个属性触发器。当鼠标滑过按钮时，就应用这个触发器的 EnterActions 属性。启动操作是 BeginStoryboard，它是一个触发器动作，可以包含并启动 Storyboard 元素。Storyboard 元素定义了一个 DoubleAnimation，可以连续改变 double 值。在这个动画中改变的属性值是矩形元素 outline 的 StrokeThickness 属性。该值平滑地改为 1.2，因为 By 属性指定，该属性变化的时间长度是 Duration 属性设置的 0.3 秒。在动画结束时，笔触的宽度重新设置为其初始值，因为 AutoReverse="True"。总之，只要鼠标滑过按钮，outline 的边框就在 0.3 秒内增加 1.2。图 35-19 显示了没有改变的按钮，图 35-20 显示了鼠标滑过按钮 0.3 秒后的按钮。在纸质媒介上，不可能显示平滑的动画和按钮外观的中间状态。

```
<Window x:Class="AnimationSample.ButtonAnimation"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Animation Sample" Height="300" Width="300">
<Window.Resources>
<Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
<Setter Property="Template">
<Setter.Value>
<ControlTemplate TargetType="{x:Type Button}">
<Grid>
<Rectangle Name="outline" RadiusX="9" RadiusY="9" Stroke="Black"
Fill="{TemplateBinding Background}" StrokeThickness="0.4">
</Rectangle>
<ContentPresenter VerticalAlignment="Center"
HorizontalAlignment="Center">
</ContentPresenter>
</Grid>
<ControlTemplate.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Trigger.EnterActions>
<BeginStoryboard>
<Storyboard>
<DoubleAnimation Duration="0:0:0.3" AutoReverse="True"
Storyboard.TargetProperty="(Rectangle.StrokeThickness)"
Storyboard.TargetName="outline" By="1.2" />
</Storyboard>
</BeginStoryboard>
</Trigger.EnterActions>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<Grid>
<Button Style="{StaticResource AnimatedButtonStyle}" Content="Click Me" />
</Grid>
</Window>
```

```
</BeginStoryboard>
</Trigger.EnterActions>
</Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Window.Resources>
<Grid>
<Button Style="{StaticResource MyButtonStyle}" Width="200"
Height="100">
Click Me!
</Button>
</Grid>
</Window>
```



图 35-19

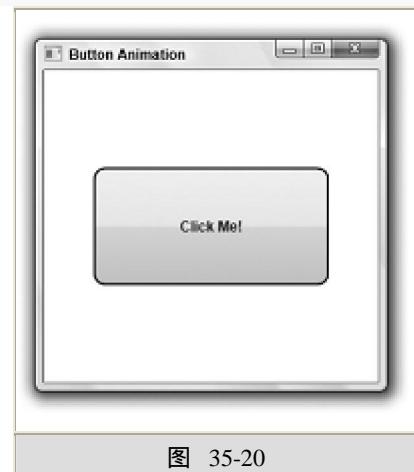


图 35-20

Timeline 可以完成的任务如表 35-4 所示。

表 35-4

Timeline 属性	说 明
-------------	-----

AutoReverse	使用 AutoReverse 属性，可以指定连续改变的值在动画结束后是否返回初始值
SpeedRatio	使用 SpeedRatio，可以改变动画的执行速度。在这个属性中，可以定义父子元素的相对关系。默认值为 1；将速率设置为较小的值，会使动画执行较慢；将速率设置为高于 1 的值，会使动画执行较快
BeginTime	使用 BeginTime，可以指定从触发器事件开始到动画开始之间的时间长度。其单位可以是天、小时、分钟、秒和几分之秒。根据 SpeedRatio，这可以不是真实的时间。例如，如果 SpeedRatio 设置为 2，开始时间设置为 6 秒，动画就在 3 秒后开始
AccelerationRatio	在动画中，值不一定是线性变化。可以指定 AccelerationRatio 或 Deceleration Ratio，定义加速度和减速度。这两个值的总和不能超过 1
DecelerationRatio	
Duration	使用 Duration 属性，可以指定动画执行一次的时间长度
RepeatBehavior	给 RepeatBehavior 属性指定一个 RepeatBehavior 结构，可以定义动画的重复次数或重复时间
FillBehavior	如果父元素的时间线有不同的持续时间，则 FillBehavior 属性就很重要。例如，如果父元素的时间线比实际动画的时间短，则将 FillBehavior 设置为 Stop 就表示动画停止。如果父元素的时间线比实际动画的时间长，HoldEnd 就会一直执行动画，直到连续改变的值重新设置为其初始值为止(假定 AutoReverse 设置为 true)

根据 Timeline 类的类型，还可以使用其他一些属性。例如，使用 DoubleAnimation，可以指定如表 35-5 所示的属性。

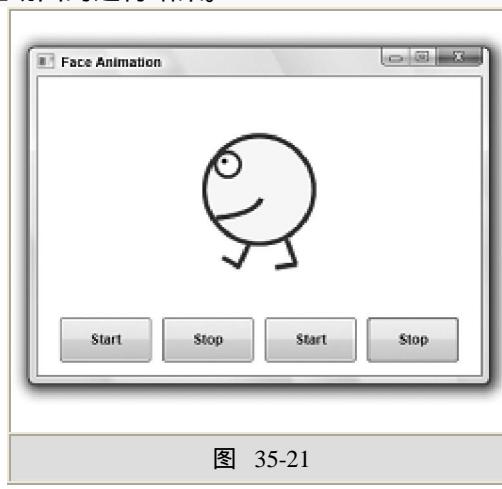
表 35-5

DoubleAnimation 属性	说 明
From , To	设置 From 和 To 属性，可以指定开始和结束动画的值
By	除了为动画定义开始值之外，还可以设置 By 属性，用绑定属性的当前值启动动画，该属性值会递增由 By 属性指定的值，直到动画结束为止

### 35.3.2 触发器

除了使用属性触发器之外，还可以定义一个事件触发器，来启动动画。下一个例子将为前面示例中的笑脸创建一个动画，一旦引发了按钮的 Click 事件，眼睛就会移动。这个例子还说明，可以在 XAML 和后台代码中启动动画。

图 35-21 显示了这个笑脸动画的运行结果。



在 Window 元素中，定义了一个 DockPanel 元素，来安排笑脸和按钮的位置。包含 Canvas 元素的栅格停靠在顶部。停靠在底部的是包含四个按钮的 StackPanel 元素。前两个按钮用于在后台代码中连续改变眼睛的位置，后两个按钮用于在 XAML 中连续改变眼睛的位置。

动画在<DockPanel.Triggers>段中定义。这里没有使用属性触发器，而使用了事件触发器。

RoutedEvent 和 SourceName 属性定义了按钮 startButtonXAML，该按钮的 Click 事件发生时，就启动第一个事件触发器。触发器的动作由 BeginStoryboard 元素定义，它启动所包含的 Storyboard。BeginStoryboard 定义了一个名称，它用于控制情节板的暂停、继续和停止动作。Storyboard 元素包含两个动画。第一个动画改变眼睛的 Canvas.Left 位置值，第二个动画改变 Canvas.Top 值。这两个动画有不同的时间值，使用指定的重复执行方式使眼睛的运动更有趣。

在按钮 stopButtonXAML 的 Click 事件发生时，就启动第二个事件触发器。在这里，情节板用 StopStoryboard 元素停止，该元素引用了起始情节板 beginMoveEye。

```
<Window x:Class="AnimatedFace.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Animated Face" Height="300" Width="406">
<DockPanel>
<Grid DockPanel.Dock="Top">
<!-- Funny Face -->
<Canvas Width="200" Height="200">
<Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
<Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
Stroke="Blue" StrokeThickness="3" Fill="White" />
<Ellipse Name="eye" Canvas.Left="67" Canvas.Top="72" Width="5"
Height="5" Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
Data="M 62,125 Q 95,122 102,108" />
<Line Name="LeftLeg" X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
StrokeThickness="4" />
<Line Name="LeftFoot" X1="68" X2="83" Y1="160" Y2="169"
Stroke="Blue"
StrokeThickness="4" />
<Line Name="RightLeg" X1="124" X2="132" Y1="144" Y2="166"
Stroke="Blue"
StrokeThickness="4" />
<Line Name="RightFoot" X1="114" X2="133" Y1="169" Y2="166"
Stroke="Blue"
StrokeThickness="4" />
</Canvas>
</Grid>
<StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal">
<Button Width="80" Height="40" Margin="20,5,5,5"
Name="startAnimationButton">Start</Button>
```

```
<Button Width="80" Height="40" Margin="5,5,5,5"
Name="stopAnimationButton">Stop</Button>
<Button Width="80" Height="40" Margin="5,5,5,5"
Name="startButtonXAML">Start</Button>
<Button Width="80" Height="40" Margin="5,5,5,5"
Name="stopButtonXAML">Stop
</Button>
</StackPanel>
<DockPanel.Triggers>
<EventTrigger RoutedEvent="Button.Click"
SourceName="startButtonXAML">
<BeginStoryboard Name="beginMoveEye">
<Storyboard Name="moveEye">
<DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
AutoReverse="True" By="8" Duration="0:0:1"
Storyboard.TargetName="eye"
Storyboard.TargetProperty="(Canvas.Left)" />
<DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
By="8"
Duration="0:0:5" Storyboard.TargetName="eye"
Storyboard.TargetProperty="(Canvas.Top)" />
</Storyboard>
</BeginStoryboard>
</EventTrigger>
<EventTrigger RoutedEvent="Button.Click"
SourceName="stopButtonXAML">
<StopStoryboard BeginStoryboardName="beginMoveEye" />
</EventTrigger>
</DockPanel.Triggers>
</DockPanel>
</Window>
```

除了在 XAML 的事件触发器中直接启动和停止动画之外，还可以在后台代码中控制动画。按钮 startAnimationButton 和 stopAnimationButton 分别关联了事件处理程序 OnStartAnimation 和 OnStopAnimation。在这些事件处理程序中，动画用 Begin()方法启动，用 Stop()方法停止。在 Begin() 方法中，第二个参数设置为 true，允许用停止请求来控制动画。

```
public partial class Window1 : System.Windows.Window
{
    public Window1()
    {
        InitializeComponent();
        startAnimationButton.Click += OnStartAnimation;
        stopAnimationButton.Click += OnStopAnimation;
    }
    void OnStartAnimation(object sender, RoutedEventArgs e)
```

```
{  
    moveEye.Begin(eye, true);  
}  
void OnStopAnimation(object sender, RoutedEventArgs e)  
{  
    moveEye.Stop(eye);  
}  
}
```

现在就可以启动应用程序，单击一个 Start 按钮，观看眼睛的移动了。

### 35.3.3 故事板

Storyboard 类继承自基类 Timeline，但可以包含几个时间线。Storyboard 类可以用于控制时间线。

表 35-6 描述了 Storyboard 类的方法。

表 35-6

Storyboard 类的方法	说 明
Begin()	Begin()方法启动与情节板关联的动画
BeginAnimation()	BeginAnimation()方法可以为一个依赖属性启动单个动画
CreateClock()	CreateClock()方法返回一个 Clock 对象，用于控制动画
Pause() , Resume()	使用 Pause()和 Resume()可以暂停、恢复动画的播放
Seek()	使用 Seek()方法，可以使动画移动到指定的时间处
Stop()	Stop()方法挂起时钟，停止动画

EventTrigger 类可以定义事件发生时的操作。表 35-7 描述了这个类的属性。

表 35-7

EventTrigger 类的属性	说 明
RoutedEventArgs	使用 RoutedEventArgs 属性，可以定义在触发器开始时的事件，例如按钮的 Click 事件
SourceName	SourceName 属性确定事件应连接到哪个 WPF 元素上

可以放在 EventTrigger 中的触发器动作如表 35-8 所示。在前面的例子中，使用了 BeginStoryboard 和 StopStoryboard 动作，表 35-8 列出了其他动作。

表 35-8

TriggerAction 类	说 明
SoundPlayerAction	使用 SoundPlayerAction，可以播放.wav 文件
BeginStoryboard	BeginStoryboard 启动由 Storyboard 定义的动画
PauseStoryboard	PauseStoryboard 暂停动画
ResumeStoryboard	ResumeStoryboard 重新启动暂停的动画
StopStoryboard	StopStoryboard 停止运行的动画
SeekStoryboard	SeekStoryboard 可以改变动画的当前时间
SkipStoryboardToFill	SkipStoryboardToFill 使动画向前移动到结束时间
SetStoryboardSpeedRatio	SetStoryboardSpeedRatio 可以改变动画的播放速度

### 35.4 在 WPF 中添加 3D 特性

本节介绍 WPF 中的 3D 特性，其中包含了开始使用该特性的信息。

提示：

WPF 中的 3D 特性在 System.Windows.Media.Media3D 命名空间中。

为了理解 WPF 中的 3D 特性，一定要知道坐标系统之间的区别。图 35-22 显示了 WPF 3D 中的坐标系统。原点位于中心。X 轴的正值在右边，负值在左边。Y 轴是垂直的，正值在上边，负值在下边。Z 轴在指向观察者的方向上定义了正值。

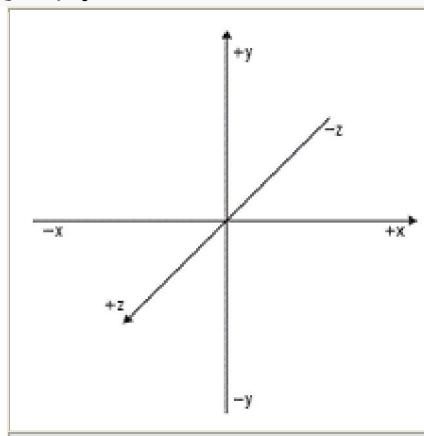


图 35-22

表 35-9 描述了最重要的类及其功能。

表 35-9

类	说 明
ViewPort3D	ViewPort3D 定义了 3D 对象的渲染表面。这个元素包含 3D 绘图的所有可见元素
ModelVisual3D	ModelVisual3D 包含在 ViewPort3D 中，它包含了所有可见元素。可以给完整的模型指定变换
GeometryModel3D	GeometryModel3D 包含在 ModelVisual3D 中，它包含网格和材质
Geometry3D	Geometry3D 是一个抽象基类，定义了几何形状。派生于 Geometry3D 的类是 MeshGeometry3D。使用 MeshGeometry3D 可以定义三角形的位置，建立 3D 模型
Material	Material 是一个抽象基类，定义了 MeshGeometry3D 指定的三角形的前边和后边。Material 包含在 GeometryModel3D 中。.NET 3.5 定义了几个材质类，例如 DiffuseMaterial、EmissiveMaterial 和 SpecularMaterial。根据材质的类型，以不同的方式计算灯光。EmissiveMaterial 利用灯光的计算，使材质发出等于笔刷颜色的光。DiffuseMaterial 使用漫射光，SpecularMaterial 定义了镜面发光模型。使用 MaterialGroup 类可以创建由其他材质合并而成的材质
Light	Light 是灯光的抽象基类。其派生类有 AmbientLight、DirectionalLight、PointLight 和 SpotLight。AmbientLight 是不自然的光，会近似照亮整个场景。使用这种光看不到边界。DirectionalLight 定义了定向光。太阳光就是一种定向光，光线来自一边，此时可以看到边界和阴影。PointLight 是一种位于指定位置的光，会照亮所有的方向。SpotLight 照亮指定的方向。这个光定义了一个圆锥，会得到一个发出光亮的区域
Camera	Camera 是摄像机的抽象基类，用于把 3D 场景映射为 2D 显示。其派生类是 PerspectiveCamera、OrthographicCamera 和 MatrixCamera。在 PerspectiveCamera 中，3D 对象离得越远就越小，这不同于 OrthographicCamera，在 Orthographic Camera 中，摄像机的

	距离对对象的大小没有影响。在 MatrixCamera 中，可以在矩阵中定义视图和变换
Transform3D	Transform3D 是 3D 变换的抽象基类。其派生类是 RotateTransform3D、ScaleTransform3D、TranslateTransform3D、MatrixTransform3D 和 Transform3D Group。TranslateTransform3D 允许在 x、y 和 z 向上变换对象，ScaleTransform3D 可以重置对象的大小。RotateTransform3D 可以在 x、y 和 z 向上把对象旋转指定的角度。Transform3DGroup 可以合并其他变换效果

## 三角形

本节从一个简单的 3D 示例开始。3D 模型由三角形组成，所以最简单的模型是一个三角形。三角形用 MeshGeometry3D 的 Positions 属性定义。3 个顶点都使用相同的 z 坐标 -4，x、y 坐标分别为 -1-1、1-1 和 01。属性 TriangleIndices 指定了逆时针的位置顺序。使用这个属性可以确定三角形的哪一边是可见的。三角形的一边显示了用 GeometryModel3D 类的 Material 属性定义的颜色，其他边显示了 BackMaterial 属性定义的颜色。

用于显示场景的摄像机位于坐标 0,0,0，其方向指向 0,0,-8。把摄像机的位置改变到左边，矩形就移动到右边，反之亦然。改变摄像机的 y 位置，矩形就会变大或变小。

这个场景中使用的光线是 AmbientLight，它用白色光照亮了整个场景。图 35-23 显示了三角形的效果。

```
< Window x:Class="Triangle3D.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="3D" Height="300" Width="300" >
< Grid >
< Viewport3D >
< Viewport3D.Camera >
< PerspectiveCamera Position="0 0 0" LookDirection="0 0 -8" / >
< /Viewport3D.Camera >

< ModelVisual3D >
< ModelVisual3D.Content >
< AmbientLight Color="White" / >
< /ModelVisual3D.Content >
< /ModelVisual3D >

< ModelVisual3D >
< ModelVisual3D.Content >
< GeometryModel3D >
< GeometryModel3D.Geometry >
< MeshGeometry3D
Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
TriangleIndices="0, 1, 2" / >
< /GeometryModel3D.Geometry >
< GeometryModel3D.Material >
< MaterialGroup >
```

```
< DiffuseMaterial >
< DiffuseMaterial.Brush >
< SolidColorBrush Color="Red" / >
< /DiffuseMaterial.Brush >
< /DiffuseMaterial >
< /MaterialGroup >
< /GeometryModel3D.Material >
< /GeometryModel3D >
< /ModelVisual3D.Content >
< /ModelVisual3D >
< /Viewport3D >
< /Grid >
< /Window >
```

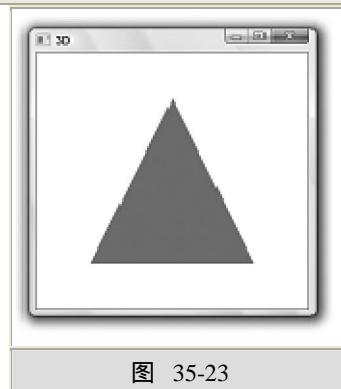


图 35-23

## 1. 改变光线

图 35-23 仅显示了一个简单的三角形，它与 2D 的效果相同。但是，下面将继续添加 3D 特性。例如，用 SpotLight 元素把环境光改为聚光灯，就可以看到三角形的另一个外观，使用聚光灯可以定义光源的位置和光线的照射方向。给光源的位置指定 -1 1 2，光就位于三角形的左边顶点处，其 y 坐标是三角形的高度。之后，光线向下向左照射。图 35-24 显示了三角形的新外观。

```
< ModelVisual3D >
< ModelVisual3D.Content >
< SpotLight Position="-1 1 -2" Color="White"
Direction="-1.5, -1, -5" / >
< /ModelVisual3D.Content >
< /ModelVisual3D >
```

图 35-24

## 2. 添加纹理

除了给三角形的材质使用纯色笔刷之外，还可以使用其他笔刷，例如 LinearGradientBrush，如下面的 XAML 代码所示。用 DiffuseMaterial 定义的 LinearGradientBrush 元素指定了黄色、橙色、红色、蓝色和紫罗兰色的渐变点。要把使用这种笔刷的对象的 2D 表面映射到 3D 几何体上，必须设置

TextCoordinates 属性。TextCoordinates 定义了 2D 点的集合，它可以映射到 3D 位置上。图 35-25 显示了示例应用程序中笔刷的 2D 坐标。三角形中的第一个位置-1-1 映射到笔刷坐标 0 1 上，右下角的位置 1 -1 映射到笔刷的 1 1 上，即紫罗兰色；0 1 映射到 0.5 0 上。图 35-26 显示了材质为渐变笔刷的三角形，这里也使用了环境光。

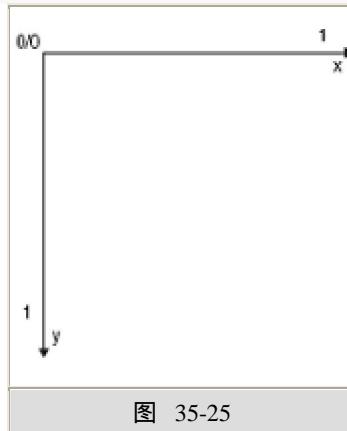


图 35-25

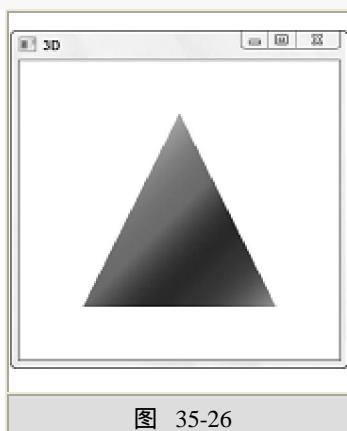


图 35-26

```
< ModelVisual3D >
< ModelVisual3D.Content >
< GeometryModel3D >
< GeometryModel3D.Geometry >
< MeshGeometry3D
    Positions="-1 -1 -4, 1 -1 -4, 0 1 -4"
    TriangleIndices="0, 1, 2"
    TextureCoordinates="0 1, 1 1, 0.5 0" / >
< /GeometryModel3D.Geometry >

< GeometryModel3D.Material >
< MaterialGroup >
< DiffuseMaterial >
< DiffuseMaterial.Brush >
< LinearGradientBrush StartPoint="0,0"
    EndPoint="1,1" >
< GradientStop Color="Yellow" Offset="0" / >
< GradientStop Color="Orange" Offset="0.25" / >
< GradientStop Color="Red" Offset="0.50" / >
< GradientStop Color="Blue" Offset="0.75" / >
```

```
< GradientStop Color="Violet" Offset="1" / >
< /LinearGradientBrush >
< /DiffuseMaterial.Brush >
< /DiffuseMaterial >
< /MaterialGroup >
< /GeometryModel3D.Material >
< /GeometryModel3D >
< /ModelVisual3D.Content >
< /ModelVisual3D >
```

#### 提示：

可以用类似的方式添加文本和其他控件。为此，只需用要绘制的元素创建 VisualBrush。VisualBrush 详见第 34 章。

### 3. 3D 对象

下面研究真正的 3D 对象：立方体。立方体由 5 个矩形组成：后面、前面、左面、右面和底面。每个矩形都由两个三角形组成，因为这是网格的核心。在 WPF 和 3D 术语中，网格用于描述建立 3D 形状的基本三角形。

下面是立方体中前面矩形的代码，该矩形由两个三角形组成。三角形的位置按 TriangleIndices 定义的逆时针设置。立方体的前面用红色的笔刷绘制，后面用灰色笔刷绘制。这两个笔刷都是 SolidColorBrush 类型，用 Window 的资源定义。

```
< !-- Front -- >
< GeometryModel3D >
< GeometryModel3D.Geometry >
< MeshGeometry3D
    Positions="-1 -1 1, 1 -1 1, 1 1 1, 1 1 1,
              -1 1 1, -1 -1 1"
    TriangleIndices="0 1 2, 3 4 5" / >
< /GeometryModel3D.Geometry >
< GeometryModel3D.Material >
< DiffuseMaterial Brush="{StaticResource redBrush}" / >
< /GeometryModel3D.Material >
< GeometryModel3D.BackMaterial >
< DiffuseMaterial Brush="{StaticResource grayBrush}" / >
< /GeometryModel3D.BackMaterial >
< /GeometryModel3D >
```

其他矩形非常类似，只是在不同的位置上。下面是立方体左面的 XAML 代码：

```
< !-- Left side -- >
< GeometryModel3D >
< GeometryModel3D.Geometry >
< MeshGeometry3D
```

```
Positions="-1 -1 1, -1 1 1, -1 -1 -1, -1 -1 -1, -1 1 1,  
-1 1 -1"  
TriangleIndices="0 1 2, 3 4 5" />  
</GeometryModel3D.Geometry>  
<GeometryModel3D.Material>  
<DiffuseMaterial Brush="{StaticResource redBrush}" />  
</GeometryModel3D.Material>  
<GeometryModel3D.BackMaterial>  
<DiffuseMaterial Brush="{StaticResource grayBrush}" />  
</GeometryModel3D.BackMaterial>  
</GeometryModel3D>
```

提示：

示例代码为立方体的每个面定义了一个 GeometryModel3D，仅是为了更好地理解代码。只要每个面都使用相同的材质，就可以定义一个网格，它包含立方体所有面的全部 10 个三角形。

所有的矩形都在 Model3DGroup 中组合，所以可以对立方体的所有面进行变换：

```
<!-- the model -->  
<ModelVisual3D>  
<ModelVisual3D.Content>  
<Model3DGroup>  
  
<!-- GeometryModel3D elements for every side of the box  
-->  
  
</Model3DGroup>
```

使用 Model3DGroup 的 Transform 属性，就可以变换这个组中的所有几何体。下面使用 RotateTransform3D 定义一个 AxisAngleRotation3D。要在运行期间旋转立方体，Angle 属性要绑定到 Slider 控件的值上。

```
<!-- Transformation of the complete model -->  
<Model3DGroup.Transform>  
<RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">  
</RotateTransform3D>  
<AxisAngleRotation3D x:Name="axisRotation" Axis="0, 0, 0" Angle="{Binding Path=Value, ElementName=axisAngle}" />  
</AxisAngleRotation3D>  
</RotateTransform3D>  
</Model3DGroup.Transform>
```

为了查看立方体，需要一个摄像机。这里使用 PerspectiveCamera，因此立方体离摄像机越远，就越小。摄像机的位置的方向在运行期间设置。

```
<!-- Camera -->
<Viewport3D.Camera>
<PerspectiveCamera x:Name="camera"
Position="{Binding Path=Text,
ElementName=textCameraPosition}"
LookDirection="{Binding Path=Text,
ElementName=textCameraDirection}" />
</Viewport3D.Camera>
```

应用程序使用两个不同的光源，其中一个光源是 DirectionalLight：

```
<!-- directional light -->
<ModelVisual3D>
<ModelVisual3D.Content>
<DirectionalLight Color="White"
x:Name="directionalLight">
<DirectionalLight.Direction>
<Vector3D X="1" Y="2" Z="3" />
</DirectionalLight.Direction>
</DirectionalLight>
</ModelVisual3D.Content>
</ModelVisual3D>
```

另一个光源是 SpotLight。使用这个光源可以突出显示立方体的一个特定区域。SpotLight 定义了属性 InnerConeAngle 和 OuterConeAngle，以指定完全照亮的区域：

```
<!-- spot light -->
<ModelVisual3D>
<ModelVisual3D.Content>
<SpotLight x:Name="spotLight"
InnerConeAngle="{Binding Path=Value,
ElementName=spotInnerCone}"
OuterConeAngle="{Binding Path=Value,
ElementName=spotOuterCone}"
Color="#FFFFFF"
Direction="{Binding Path=Text,
ElementName=spotDirection}"
Position="{Binding Path=Text,
ElementName=spotPosition}"
Range="{Binding Path=Value, ElementName=spotRange}" />
</ModelVisual3D.Content>
</ModelVisual3D>
```

运行应用程序，就可以改变立方体的旋转角度、摄像机和灯光，如图 35-27 所示。

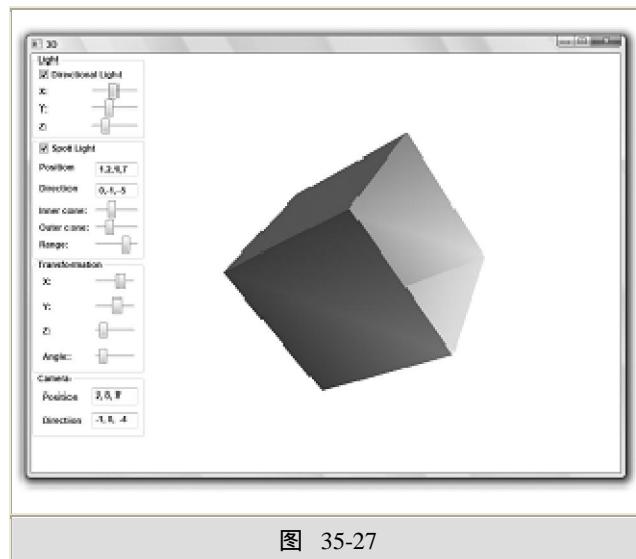


图 35-27

**提示：**

创建仅包含矩形或三角形的 3D 模型是很简单的。不应手工创建更复杂的模型，而应使用对应的工具。WPF 的 3D 工具在 [www.codeplex/3DTools](http://www.codeplex/3DTools) 上。

### 35.5 Windows 窗体集成

除了给 WPF 从头开始编写用户界面之外，还可以在 WPF 应用程序中使用已有的 Windows 窗体控件，创建要在 Windows 窗体应用程序中使用的新 WPF 控件。集成 Windows 窗体和 WPF 的最佳方式是创建控件，将它们集成到另一个应用程序中。

**警告：**

Windows 窗体和 WPF 的集成有一个很大的缺陷。如果集成了 Windows 窗体和 WPF，Windows 窗体控件仍显示以前的外观。Windows 窗体控件和应用程序并没有获得 WPF 的新外观。从用户界面的角度来看，最好完全重写用户界面。

**提示：**

要集成 Windows 窗体和 WPF，需要使用 WindowsFormsIntegration 程序集的 System.Windows.Forms.Integration 命名空间中的类。

#### 35.5.1 Windows 窗体中的 WPF 控件

可以在 Windows 窗体应用程序中使用 WPF 控件。WPF 元素是一个一般的.NET 类。但是，不能在 Windows 窗体代码中直接使用它；WPF 控件不是 Windows 窗体控件。集成可以利用 System.Windows.Forms.Integration 命名空间中的 ElementHost 封装类来进行。ElementHost 是一个 Windows 窗体控件，因为它派生自 System.Windows.Forms.Control，在 Windows 窗体应用程序中，可以像其他 Windows 窗体控件那样使用。ElementHost 包含和管理 WPF 控件。

下面是一个简单的 WPF 控件。在 Visual Studio 2008 中，可以创建一个定制的 WPF 用户控件库。示例控件派生自基类 UserControl，包含一个栅格和一个带定制内容的按钮。

```
<UserControl x:Class="WPFControl.UserControl1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
<Grid>
<Button>
<Canvas Height="230" Width="230">
<Ellipse Canvas.Left="50" Canvas.Top="50" Width="100" Height="100"
Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
<Ellipse Canvas.Left="60" Canvas.Top="65" Width="25" Height="25"
Stroke="Blue" StrokeThickness="3" Fill="White" />
<Ellipse Canvas.Left="70" Canvas.Top="75" Width="5" Height="5"
Fill="Black" />
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
Data="M 62,125 Q 95,122 102,108" />
<Line X1="124" X2="132" Y1="144" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="114" X2="133" Y1="169" Y2="166" Stroke="Blue"
StrokeThickness="4" />
<Line X1="92" X2="82" Y1="146" Y2="168" Stroke="Blue"
StrokeThickness="4" />
<Line X1="68" X2="83" Y1="160" Y2="168" Stroke="Blue"
StrokeThickness="4" />
</Canvas>
</Button>
</Grid>
</UserControl>
```

可以选择 Windows 窗体应用程序模板，创建一个 Windows 窗体应用程序。因为 WPF 用户控件项目在 Windows 窗体应用程序所在的解决方案中，所以可以把 WPF 用户控件从工具箱拖放到 Windows 窗体应用程序的设计界面上。这会添加对程序集 PresentationCore、PresentationFramework、WindowsBase、WindowsFormsIntegration 和包含 WPF 控件的程序集的引用。

在设计器生成的代码中，有一个引用 WPF 用户控件的变量和一个封装控件的 ElementHost 类型的对象。

```
private System.Windows.Forms.Integration.ElementHost
elementHost1;
private WPFControl.UserControl1 userControl11;
```

在 InitializeComponent 方法中，初始化了对象，把 WPF 控件实例赋予 ElementHost 类的 Child 属性：

```
private void InitializeComponent()
{
    this.elementHost1 = new
System.Windows.Forms.Integration.ElementHost();
```

```
this.userControl11 = new WPFControl.UserControl1();
this.SuspendLayout();
//
// elementHost1
//
this.elementHost1.Location = new
System.Drawing.Point(39, 44);
this.elementHost1.Name = "elementHost1";
this.elementHost1.Size = new System.Drawing.Size(259,
229);
this.elementHost1.TabIndex = 0;
this.elementHost1.Text = "elementHost1";
this.elementHost1.Child = this.userControl11;
//...
}
```

启动 Windows 窗体应用程序，WPF 控件和 Windows 窗体控件会显示在一个窗体中，如图 35-28 所示。

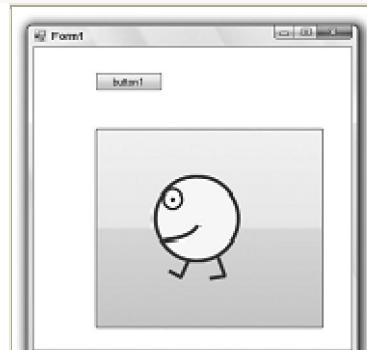


图 35-28

当然，还可以给 WPF 控件添加方法、属性和事件，再以与其他控件相同的方式使用它们。

### 35.5.2 WPF 应用程序中的 Windows 窗体控件

还可以用另一种方式集成 Windows 窗体和 WPF：把 Windows 窗体控件放在 WPF 应用程序中。与用于在 Windows 窗体中包含 WPF 控件的 ElementHost 类一样，现在需要一个封装器，它是一个包含 Windows 窗体控件的 WPF 控件。这个类的名称是 WindowsFormsHost，它也位于 WindowsFormsIntegration 程序集中。WindowsFormsHost 类派生自 HwndHost 和 FrameworkElement，因此可以用作 WPF 元素。

对于这种集成，应先创建一个 Windows 控件库。使用设计器给窗体添加一个文本框和一个按钮控件。修改按钮的 Text 属性，在后台代码中添加属性 ButtonText：

```
public partial class UserControl1 : UserControl
{
    public UserControl1()
    {
```

```
InitializeComponent();  
}  
  
public string ButtonText  
{  
    get { return button1.Text; }  
    set { button1.Text = value; }  
}
```

在 WPF 应用程序中，可以把 WindowsFormsHost 对象从工具箱添加到设计器上。这需要引用程序集 WindowsFormsIntegration、System.Windows.Forms 和包含 Windows 窗体控件的程序集。要在 XAML 中使用 Windows 窗体控件，必须添加一个 XML 命名空间别名，以引用.NET 命名空间。因为 Windows 窗体控件与 WPF 应用程序位于不同的程序集中，所以还必须将该程序集的名称添加到命名空间别名中。Windows 窗体控件现在可以包含在 WindowsFormsHost 元素中，如下所示。可以在 XAML 中，直接给属性 ButtonText 赋值，其方式与.NET Framework 元素类似：

```
<Window x:Class="WPFApplication.Window1"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:wf="clr-namespace:Wrox.ProCSharp.WPF;assembly=WindowsFormsControl"  
       Title="WPF Interop Application" Height="300" Width="300"  
>  
  
<Grid>  
    <Grid.RowDefinitions>  
        <RowDefinition />  
        <RowDefinition />  
    </Grid.RowDefinitions>  
    <WindowsFormsHost Grid.Row="0" Height="180">  
        <wf:UserControl1 x:Name="myControl" ButtonText="Click Me!" />  
    </WindowsFormsHost>  
    <StackPanel Grid.Row="1">  
        <TextBox Margin="5,5,5,5" Width="140" Height="30"/>  
        <Button Margin="5,5,5,5" Width="80" Height="40">WPF Button</Button>  
    </StackPanel>  
    </Grid>  
</Window>
```

图 35-29 显示了 WPF 应用程序的一个视图。当然，Windows 窗体控件仍像一个 Windows 窗体控件，没有获得 WPF 提供的重新设置大小和样式特性。

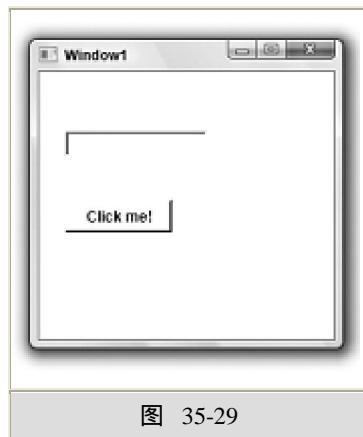


图 35-29

### 35.6 WPF 浏览器应用程序

VS2008 有另一个 WPF 项目模板：WPF 浏览器应用程序。这个应用程序可以运行在 IE 中，但我们在使用的.NET Framework 版本必须安装在客户机系统中。在该模板中可以获得用于浏览器的丰富客户机功能。但是，在 WPF 浏览器应用程序中，需要把.NET Framework 安装在客户机系统中，且仅支持 IE。

创建这样的项目类型，需要 XBAP(XAML Browser Application)文件。XBAP 是一个 XML 文件，定义了应用程序和它包含的程序集，用于 ClickOnce 部署。

XBAP 应用程序是一个部分信任的应用程序。只能在 Internet 权限中使用.NET 代码。

提示：

ClickOnce 参见第 16 章。

提示：

WPF 浏览器应用程序不同于 Silverlight。Silverlight 定义了一个 WPF 子集，它不需要把.NET Framework 安装在客户机系统中，但需要把一个插件安装在浏览器上，且支持不同的浏览器和不同的操作系统。Silverlight 1.0 不能用.NET 编程，只能使用 JavaScript 编程访问 XAML 元素。Silverlight 1.1 支持.NET Microframework。

### 35.7 小结

本章介绍了 WPF 的许多特性。WPF 的数据绑定特性比 Windows 窗体前进了一大步。可以把.NET 类的任意属性绑定到 WPF 元素的属性上。绑定模式定义了绑定的方向。可以绑定.NET 对象和列表，定义数据模板，为.NET 类创建默认的外观。

命令绑定可以把处理程序的代码映射到菜单和工具栏上。还可以用 WPF 进行复制和粘贴，因为这个技术的命令处理程序已经包含在 TextBox 控件中。

动画允许用户动态改变 WPF 元素的每个属性。动画可以非常有趣，使 UI 响应更快，更吸引人。

WPF 还可以把 3D 映射到屏幕的 2D 表面上，我们学习了如何创建 3D 模型，用不同的光源和摄像机观察它。

提示：

本章和上一章概述了 WPF，所提供的信息足以使用这个技术。WPF 的更多信息可参阅有关 WPF 的图书，例如 Chris Andrade 等编著的 Profession WPF Programming: .NET Development with the Windows Presentation Foundation(Wiley 出版社于 2007 年出版)。

## 第 36 章插件

插件可以在以后给应用程序添加功能。我们可以创建一个主机应用程序，随时间的推移给它添加越来越多的功能--这些功能可以是开发团队编写的，其他供应商也可以创建插件，扩展该应用程序。

目前，插件在许多不同的应用程序上使用，例如 IE 和 Visual Studio。IE 是一个主机应用程序，它提供了一个插件框架，许多公司都使用这个框架提供查看 Web 页面时的扩展程序。Shockwave Flash Object 可以查看带 Flash 内容的 Web 页面。Google 工具栏提供了特殊的 Google 功能，可以在 IE 中快速访问。Visual Studio 也有一个插件模型，可以用不同层次的扩展程序扩展 Visual Studio。

定制应用程序总是可以创建插件模型，以动态加载和使用程序集中的功能。利用插件模型时，需要考虑许多问题。如何检测新的程序集？如何解决版本问题？插件可以改变主机应用程序的稳定性吗？

.NET Framework 3.5 提供了一个框架，用程序集 System.AddIn 来保存和创建插件。这个框架也称为 Managed AddIn Framework (MAF)。

提示：

插件还有其他称呼，如 add-on 或 plug-in。

本章内容如下：

System.AddIn 体系结构

创建简单的插件

### 36.1 System.AddIn 体系结构

创建允许在运行期间添加插件的应用程序时，需要处理一些问题。例如，如何找到插件，如何解决版本问题，使主机应用程序和插件可以独立地升级。要解决这些问题，有几种方式。本节讨论插件的问题和 MAF 解决它们的体系结构：

插件的问题

管道体系结构

发现

激活

隔离

生存期

版本问题

#### 36.1.1 插件的问题

要创建一个主机应用程序，动态加载以后添加的程序集，必须解决几个问题，如表 36-1 所示。

表 36-1

插件问题	说明
发现	如何为主机应用程序查找新插件？这有几个不同的选项。一个选项是在配置文件中添加插件的信息。其缺点是安装新插件时，需要修改已有的配置文件。另一个选项是把包含插件的程序集复制到预定义的目录中，通过反射读取程序集的信息。 反射的更多内容可参见第 13 章
激活	程序集动态加载后，还不能使用 new 运算符创建它的实例。但可以用 Activator 类创建这类程序集。另外，如果插件加载到另一个应用程序域中或新进程中，还需要使用不同的激活选项。 程序集和应用程序域的更多内容可参见第 17 章
隔离	插件可能会使主机应用程序崩溃，读者可能见过 IE 因各种插件而崩溃的情况。根据主机应用程序的类型和插件的集成方式，插件可以加载到另一个应用程序域或另一个进程中
生存期	清理对象是垃圾回收器的一个工作。但是，垃圾回收器在这里没有任何帮助，因为插件可能在另一个应用程序域中或另一个进程中激活。把对象保存在内存中的其他方式有引用计数、租借和承办机制
版本	版本问题是插件的一个大问题。通常主机的一个新版本仍可以加载旧插件，而旧主机应有加载新插件的选项

下面探讨 MAF 的体系结构，说明这个框架如何解决这些问题。MAF 的设计目标如下：

应易于开发插件

在运行期间查找插件应很高效

开发主机程序应是一个很简单的过程，但不像开发插件那么容易

插件和主机应用程序应独立地升级

### 36.1.2 管道体系结构

MAF 体系结构基于一个包含 7 个程序集的管道。这个管道解决了插件的版本问题。因为管道中的程序集之间的依赖性很低，所以合同、主机程序和插件升级到新版本可以完全互不干扰。

图 36-1 显示了 MAF 体系结构的管道。其中心是合同程序集。这个程序集包含一个合同接口，其中列出了插件必须实现、可以由主机程序调用的方法和属性。合同的左边是主机端，右边是插件端。图中还显示了程序集之间的依赖性。最左端的主机程序集与合同程序集没有依赖性，插件程序集与合同程序集也没有依赖性，这两个程序集都没有实现合同定义的接口，只是有一个对视图程序集的引用。主机应用程序引用主机视图；插件引用插件视图。视图包含抽象的视图类，该类定义的方法和属性与合同相同。



图 36-1

图 36-2 显示了管道中类的关系。主机类与抽象的主机视图类有一个关联，并调用其方法。抽象的主机视图类由主机适配器实现。适配器在视图和合同之间建立连接。插件适配器实现了合同的方法和属性。这个适配器包含对插件视图的引用，把来自主机端的调用传送给插件视图。主机适配器类定

义了一个具体的类，它派生自主机视图的抽象基类，实现了方法和属性。这个适配器包含对合同的引用，把来自视图的调用传送给合同。

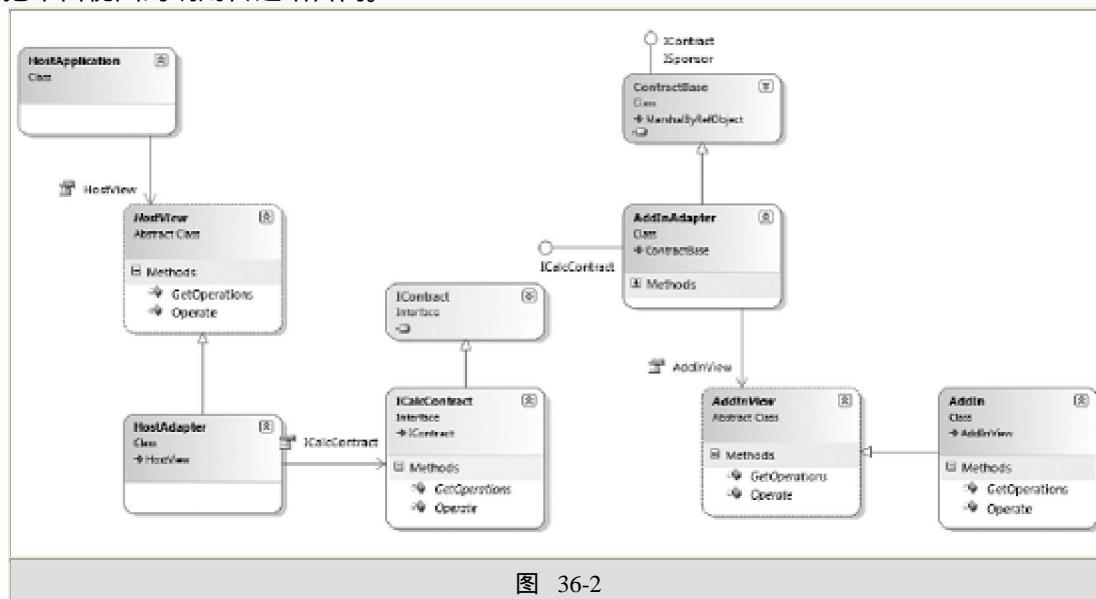


图 36-2

有了这个模型，插件端和主机端可以完全独立地升级了，只是需要使用映射层。例如，如果主机的一个新版本使用全新的方法和属性，合同就仍可以保持不变，只有适配器需要修改。也可以定义新的合同。适配器可以修改，也可以同时使用几个合同。

### 36.1.3 发现

如何为主机应用程序查找新插件？MAF 体系结构使用一个预定义的目录结构来查找插件和管道的其他程序集。管道的组成部分保存在这些子目录中：

HostSideAdapters

Contracts

AddInSideAdapters

AddInViews

AddIns

除了 AddIns 目录之外，其他目录都直接包含管道特定部分的程序集。AddIns 目录为每个插件程序集包含一个子目录。插件也可以保存在完全独立于其他管道组件的目录中。

管道的程序集需要使用反射来动态加载，才能获得插件的所有信息。而且，对于许多插件而言，这还会增加主机应用程序的启动时间。因此，MAF 使用一个高速缓存，来保存管道组件的信息。该高速缓存是由安装插件的程序创建的，如果主机应用程序有管道目录的写入权限，该高速缓存就由主机应用程序创建。

给管道组件高速缓存的信息是调用 AddInStore 类的方法来创建的。Update()方法查找还没有列在保存文件中的新插件。Rebuild()方法用插件的信息重建完全二进制的保存文件。

表 36-2 列出了 AddInStore 类的成员。

表 36-2

AddInStore 成员	说 明
Rebuild()	Rebuild()方法为管道的所有组件重建高速缓存。如果插件存储在另一个目录下，就可以使用 RebuildAddIns()重建插件的高速缓存
Update()	Rebuild()方法重建管道的完整高速缓存，Update()方法只用新管道组件更新高速缓存。
UpdateAddIns()	UpdateAddIns()方法只更新插件的高速缓存
FindAddIn()	这些方法都使用高速缓存查找插件。FindAddIns()方法返回匹配主机视图的所有插件集合。FindAddIn()方法返回一个特定的插件
FindAddIns()	

### 36.1.4 激活和隔离

AddInStore 类的 FindAddIns()方法返回表示插件的 AddInToken 对象集合。使用 AddInToken 类可以访问插件的信息，例如名称、描述、发布者和版本。使用 Activate()方法可以激活插件。表 36-3 列出了 AddInToken 类的属性和方法。

表 36-3

AddInToken 成员	说 明
Name、Publisher、Version、Description	AddInToken 类的 Name、Publisher、Version 和 Description 属性返回用特性 AddInAttribute 赋予插件的信息
AssemblyName	AssemblyName 返回包含插件的程序集名称
EnableDirectConnect	使用 EnableDirectConnect 属性可以设置一个值，主机程序应使用该值直接连接到插件上，而不使用管道的组件。只有插件和主机程序运行在同一个应用程序域，插件视图和主机视图的类型相同时，才能使用这个属性。该属性仍要求管道的所有组件都存在
QualificationData	插件可以用特性 QualificationDataAttribute 标记应用程序域和安全需求。插件可以列出安全需求和隔离需求。例如，[QualificationData ("Isolation", "NewAppDomain")] 表示插件必须保存在新进程中。可以从 AddInToken 中读取这些信息，激活有特定需求的插件。除了应用程序域和安全需求之外，还可以使用这个特性通过管道传送定制信息
Activate()	插件用 Activate()方法激活，利用这个方法的参数，可以定义插件是否加载到新应用程序域或新进程中。还可以定义插件获得的权限

一个插件可能使整个应用程序崩溃，例如 IE 可能因一个失败的插件而崩溃。根据应用程序类型和插件的类型，可以让插件运行在另一个应用程序域或另一个进程中，来避免这个问题。MAF 给出了几个选项。可以在新应用程序域或新进程中激活插件。新应用程序域还可以有有限的权限。

AddInToken 类的 Activate()方法有几个重载版本，在这些版本中，可以传送加载插件的环境参数。表 36-4 列出了不同的选项。

表 36-4

AddInToken.Activate() 的参数	说 明
AppDomain	可以传送一个加载插件的新应用程序域，这样可以使插件独立于主机应

	用程序，还可以从应用程序域中卸载插件
AddInSecurityLevel	如果插件应使用不同的安全级别来运行，就传送枚举 AddInSecurityLevel 的一个值，其值可以是 Internet、Intranet、FullTrust 和 Host
PermissionSet	如果预定义的安全级别不够安全，还可以给插件的应用程序域赋予 PermissionSet
AddInProcess	插件还可以运行在与主机应用程序不同的进程中。可以给 Activate()方法 传送一个新的 AddInProcess。如果所有的插件都卸载了，新进程就可以退出，否则新进程就继续运行。这个选项可以用 KeepAlive 属性设置
AddInEnvironment	传送 AddInEnvironment 对象是定义加载插件的应用程序域的另一个选项。在 AddInEnvironment 的构造函数中，可以传送一个 AppDomain 对象。还可以用 AddInController 类的 AddInEnvironment 属性获得插件的已有 AddInEnvironment

提示：

应用程序域详见第 17 章。

应用程序的类型也会限制可以使用的选项。WPF 插件目前不支持跨进程。Windows Forms 不能在不同的应用程序域之间连接 Windows 控件。

下面列出调用 AddInToken 的 Activate()方法时管道的执行步骤：

- (1) 用指定的权限创建应用程序域。
- (2) 用 Assembly.LoadFrom()方法把插件的程序集加载到新的应用程序域中。
- (3) 用反射调用插件的默认构造函数。因为插件派生于在插件视图中定义的基类，所以也加载了视图的程序集。
- (4) 接着构造插件端适配器的一个实例。插件的实例传送给适配器的构造函数，使适配器能连接合同和插件。插件适配器派生于基类 MarshalByRefObject，所以可以在应用程序域之间调用。
- (5) 激活代码给主机应用程序的应用程序域返回插件端适配器的一个代理。插件适配器实现了合同接口，所以该代理包含合同接口的方法和实现。
- (6) 主机端适配器的实例在主机应用程序的应用程序域中构造。插件端适配器的代理传送给该构造函数。激活代码会从插件令牌中查找主机端适配器的类型。

主机端适配器返回给主机应用程序。

### 36.1.5 合同

合同定义了主机端和插件端之间的界限。合同用一个接口来定义，该定义必须派生于基接口 IContract。合同必须仔细考虑，因为它根据需要支持灵活的插件场景。

合同没有版本支持，不能改变，所以插件以前的实现代码仍可以在新的主机程序中运行。新版本应通过定义新合同来创建。

合同的类型有一些限制，其原因是版本问题，而且应用程序域要从主机应用程序跨越到插件上。类型必须是安全的，且支持版本，能在边界(应用程序域或跨进程)之间传送，也能在主机程序和插件之间传送。

可以用合同传送的类型可以是：

基本类型

其他合同

可串行化的系统类型

简单的可串行化定制类型，包括基本类型、合同，以及没有实现代码的类型

接口 IContract 的成员如表 36-5 所示。

表 36-5

IContract 的成员	说 明
QueryContract()	使用 QueryContract()可以查询合同，验证是否也实现了另一个合同。插件可以支持几个合同
RemoteToString()	QueryContract()参数需要合同的字符串表示。RemoteToString()返回当前合同的字符串表示
AcquireLifetimeToken() RevokeLifetimeToken()	客户机调用 AcquireLifetimeToken()来保存对合同的引用。AcquireLifetimeToken()会递增引用计数。RevokeLifetimeToken()递减引用计数
RemoteEquals()	RemoteEquals()可用于比较两个合同引用

合同接口在 System.AddIn.Contract、System.AddIn.Contract.Collections 和 System.AddIn.Contract.Automation 命名空间中定义。表 36-6 列出了可以用于合同的合同接口。

表 36-6

合 同	说 明
IListContract<T>	IListContract<T>可用于返回一个合同列表
IEnumeratorContract<T>	IEnumeratorContract<T>用于枚举 IListContract<T>的元素
IServiceProviderContract	一个插件可以为其他插件提供服务。提供服务的插件称为服务提供程序，它实现了接口 IServiceProviderContract。通过 QueryService()方法，可以查询实现该接口的插件提供了什么服务

(续表)

合 同	说 明
IProfferServiceContract	IProfferServiceContract 是服务提供程序和 IServiceProviderContract 提供的接口。IProfferServiceContract 定义了方法 ProfferService()和 Revoke Service()。ProfferService()给所提供的服务添加了一个 IServiceProvider Contract，而 RevokeService()删除它
INativeHandleContract	这个接口允许使用 GetHandle()方法访问内部的 Windows 句柄。这个合同由 WPF 主机程序用于使用 WPF 插件

### 36.1.6 生存期

插件需要加载多长时间？使用多少时间？何时可以卸载应用程序域？这有几个选项。一个选项是使用引用计数。每次使用插件都会递增引用计数。如果引用计数递减到 0，就可以卸载插件。另一个选项是使用垃圾回收器。如果垃圾回收器在运行，且没有对对象的引用，该对象就是垃圾回收器的目标。.NET Remoting 使用了租约机制，是使对象保持激活状态的承办者。只要租期到了，就询问承办者该对象是否应继续保持激活状态。

卸载插件还有一个特殊的问题，因为插件运行在不同的应用程序域、不同的进程中。但垃圾回收器不能跨进程工作。MAF 使用一个混合的模型来管理生存期。在单个应用程序域中，使用垃圾回收机制。在管道内部使用一个隐式的承办机制，但引用计数可用于从外部控制承办者。

下面考虑一种情况：插件加载到另一个应用程序域中。在主机应用程序中，当不再需要引用时，垃圾回收器清理了主机视图和主机端适配器。而在插件端，合同定义了方法 AcquireLifetimeToken() 和 RevokeLifetimeToken()，来递增和递减承办者的引用计数。这两个版本不仅递增和递减一个值，还可以在某个团体频繁调用 RevokeLifetimeToken() 方法时，提早释放对象。而 AcquireLifetimeToken() 返回一个表示生存期令牌的标识符，这个标识符必须用于调用 RevokeLifetimeToken() 方法。所以这两个方法总是成对调用。

通常不必处理 AcquireLifetimeToken() 和 RevokeLifetimeToken() 方法的调用，而可以使用 ContractHandle 类，在构造函数中调用 AcquireLifetimeToken()，在终结器中调用 RevokeLifetimeToken()。

提示：

终结器详见第 12 章。

在插件加载到新应用程序域的情形中，当不再需要插件时，可以删除加载的代码。MAF 使用一个简单的模型把一个插件定义为应用程序域的拥有者，如果不再需要这个插件，就卸载应用程序域。如果在激活插件时创建了应用程序域，这个插件就是应用程序域的拥有者。如果应用程序域是以前创建的，就不会自动卸载。

ContractHandle 类在主机端适配器中用来增加插件的引用计数。这个类的成员如表 36-7 所示。

表 36-7

ContractHandle 成员	说 明
Contract	在 ContractHandle 类的构造过程中，可以指定一个实现了 IContract 的对象，来保存对它的引用。Contract 属性返回这个对象
Dispose()	Dispose()方法可以调用，而不是等待垃圾回收器执行清理操作，撤回生存期令牌
AppDomainOwner()	AppDomainOwner()是 ContractHandle 类的一个静态方法，如果插件拥有该方法传送的应用程序域，该方法就返回插件适配器
ContractOwnsAppDomain()	使用静态方法 ContractOwnsAppDomain()，可以验证指定的合同是否是应用程序域的拥有者。如果是，在删除合同时，会卸载应用程序域

### 36.1.7 版本问题

版本问题是插件的一个大问题。主机应用程序可以利用插件进一步开发。插件的一个要求是主机应用程序的新版本仍可以加载插件的旧版本。旧主机程序仍可以运行插件的新版本。那么，合同该如何修改呢？

System.AddIn 完全独立于主机应用程序和插件的实现，这是通过包含 7 部分的管道概念实现的。

### 36.2 插件示例

下面是一个主机应用程序的简单示例，它可以加载计算器插件。插件支持不同的计算操作。

我们需要创建一个解决方案，它包含 6 个库项目和一个控制台应用程序。示例应用程序的项目如表 36-8 所示。这个表列出了需要引用的程序集。在解决方案中引用了其他项目后，还需要把 Copy Local 属性设置为 False，这样程序集就不会复制。但 HostApp 控制台项目例外，它需要对 HostView 项目的引用。这个程序集必须复制，才能在主机应用程序中找到。另外，还需要修改所生成的程序集的输出路径，使程序集复制到管道的正确目录下。

表 36-8

项 目	引 用	输 出 路 径	说 明
CalcContract	System.AddIn.Contract	.. Pipeline \Contracts\	这个程序集包含与插件通信的合同。合同用接口定义
CalcView	System.AddIn	.. Pipeline \AddInViews\	CalcView 程序集包含一个由插件引用的抽象类，这是合同的插件端
CalcAddIn	System.AddIn.CalcView	.. Pipeline \AddIns \CalcAddIn\	CalcAddIn 是引用插件视图程序集的插件项目。这个程序集包含插件的实现代码

(续表)

项 目	引 用	输 出 路 径	说 明
CalcAddIn Adapter	System.AddIn System.AddIn.Contract CalcView CalcContract	.. Pipeline \AddInSideAdapters\	CalcAddInAdapter 连接插件视图和合同程序集，把合同映射到插件视图上
HostView			包含主机视图的抽象类的程序集不需要引用任何插件程序集，也没有解决方案中的其他项目引用
HostAdapter	System.AddIn System.AddIn.Contract HostView CalcContract	.. Pipeline \HostSideAdapters\	主机适配器把主机视图映射到合同上。因此需要引用这些项目
HostApp	System.AddIn HostView		主机应用程序激活插件

#### 36.2.1 计算器合同

下面实现合同程序集。合同程序集包含一个合同接口，该接口定义了在主机程序和插件之间通信的协议。

下面的代码是为计算器示例应用程序定义的合同。应用程序给合同定义了方法 GetOperations() 和 Operate()。GetOperations() 方法返回计算器插件支持的一组数学操作。数学操作是由 IOperationContract 接口定义的，IOperationContract 接口本身就是一个合同，定义了只读属性 Name 和 NumberOperands。

Operate() 方法调用插件中的操作，它需要 IOperation 接口定义的一个操作和通过 double 数组提供的操作数。

有了这个接口，插件就支持需要任意多个 double 操作数的操作，且返回一个 double。

属性 AddInContract 由 AddInStore 用于建立高速缓存。这个属性把类标记为插件合同接口。

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    [AddInContract]
    public interface ICalculatorContract : IContract
    {
        IListContract < IOperationContract > GetOperations();
        double Operate(IOperationContract operation, double[] operands);
    }
    public interface IOperationContract : IContract
    {
        string Name { get; }
        int NumberOperands { get; }
    }
}
```

### 36.2.2 计算器插件视图

插件视图重新定义了插件眼中的合同。该合同定义了接口 ICalculatorContract 和 IOperationContract。为此，插件视图定义了抽象类 Calculator 和具体的类 Operation。

在 Operation 中，没有每个插件都需要的特定实现代码，因为该类已经用插件视图程序集实现了。这个类用属性 Name 和 NumberOperands 描述了数学计算的一个操作。

抽象类 Calculator 定义了需要由插件实现的方法。虽然合同定义了需要在应用程序域和进程之间传送的参数和返回类型，但插件视图不需要。这里可以使用类型，以便于插件开发人员编写插件。GetOperations() 方法返回 IList<Operation>，而不是 IList<IOperationContract>，这与合同程序集不同。

AddInBase 属性把类标识为插件视图，用于存储。

```
using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.AddIns
```

```
{  
[AddInBase]  
public abstract class Calculator  
{  
public abstract IList < Operation > GetOperations();  
public abstract double Operate(Operation operation,  
double[] operand);  
}  
public class Operation  
{  
public string Name { get; set; }  
public int NumberOperands { get; set; }  
}  
}
```

### 36.2.3 计算器插件适配器

插件适配器把合同映射到插件视图上。这个程序集引用了合同和插件视图程序集。适配器的实现代码需要把合同中的方法 `IListContract<IOperationContract> GetOperations()` 映射到视图方法 `IList<Operation> GetOperations()` 上。

该程序集包含类 `OperationViewToContractAddInAdapter` 和 `CalculatorViewToContractAddInAdapter`。这两个类实现了接口 `IOperationContract` 和 `ICalculatorContract`。基接口 `IContract` 的方法可以通过派生于基类 `ContractBase` 来实现。这个基类提供了默认的实现代码。`OperationViewToContractAddInAdapter` 实现了 `IOperationContract` 接口的其他成员，并把调用传送给在构造函数中指定的 `Operation View`。

类 `OperationViewToContractAddInAdapter` 还包含静态帮助方法 `ViewToContractAdapter()` 和 `ContractToViewAdapter()`，前者把 `Operation` 映射到 `IOperationContract` 上，后者把 `IOperationContract` 映射到 `Operation` 上。

```
using System.AddIn.Pipeline;  
namespace Wrox.ProCSharp.AddIns  
{  
internal class OperationViewToContractAddInAdapter :  
ContractBase,  
IOperationContract  
{  
private Operation view;  
public OperationViewToContractAddInAdapter(Operation  
view)  
{  
this.view = view;  
}  
public string Name
```

```
{  
    get { return view.Name; }  
}  
public int NumberOperands  
{  
    get { return view.NumberOperands; }  
}  
public static IOperationContract  
ViewToContractAdapter(Operation view)  
{  
    return new OperationViewToContractAddInAdapter(view);  
}  
public static Operation ContractToViewAdapter(  
    IOperationContract contract)  
{  
    return (contract as  
        OperationViewToContractAddInAdapter).view;  
}  
}
```

类 CalculatorViewToContractAddInAdapter 非常类似于 OperationViewToContractAddIn- Adapter：它派生自 ContractBase，继承了 IContract 接口的默认实现代码，还实现了一个合同接口。但 ICalculatorContract 接口是用 GetOperations() 和 Operate() 方法实现的。

适配器的 Operate() 方法调用视图类 Calculator 的 Operate() 方法，其中 IOperationContract 需要转换为 Operation。这是使用类 OperationViewToContractAddInAdapter 的静态帮助方法 ContractViewToAdapter() 完成的。

GetOperations() 方法的实现需要把集合 IList<IOperationContract> 转换为 IList<Operation>。对于这个集合转换，类 CollectionAdapters 定义了转换方法 ToIList() 和 ToIListContract()。其中 ToIListContract() 方法用于这个转换。

属性 AddInAdapter 把类标识为插件端适配器，用于插件的存储。

```
using System.AddIn.Contract;  
using System.AddIn.Pipeline;  
namespace Wrox.ProCSharp.AddIns  
{  
    [AddInAdapter]  
    internal class CalculatorViewToContractAddInAdapter :  
        ContractBase,  
        ICalculatorContract  
    {  
        private Calculator view;  
        public CalculatorViewToContractAddInAdapter(Calculator view)
```

```
{  
    this.view = view;  
}  
public IListContract < IOperationContract > GetOperations()  
{  
    return CollectionAdapters.ToIListContract < Operation,  
        IOperationContract > (view.GetOperations(),  
        OperationViewToContractAddInAdapter.ViewToContractAdapter,  
        OperationViewToContractAddInAdapter.ContractToViewAdapter);  
}  
public double Operate(IOperationContract operation, double[]  
    operands)  
{  
    return view.Operate(  
        OperationViewToContractAddInAdapter.ContractToViewAdapter(  
            operation), operands);  
}  
}
```

#### 提示：

因为适配器类是由.NET 反射功能调用的，所以这些类可以使用内部的访问修饰符。这些类是要具体实现的，所以最好使用 internal 访问修饰符。

#### 36.2.4 计算器插件

插件现在包含具体的实现代码。它是用类 CalculatorV1 实现的。插件程序集依赖于插件视图程序集，因为它需要实现抽象类 Calculator。

属性 AddIn 把类标记为插件，用于插件的存储，并添加了发布者、版本和描述信息。在主机端，这些信息可以从 AddInToken 中访问。

CalculatorV1 在方法 GetOperations()中返回一组支持的操作。Operate()方法根据操作计算操作数。

```
using System;  
using System.AddIn;  
using System.Collections.Generic;  
namespace Wrox.ProCSharp.AddIns  
{  
    [AddIn("CalculatorAddIn", Publisher="Wrox Press",  
        Version="1.0.0.0",  
        Description="Sample AddIn")]  
    public class CalculatorV1 : Calculator  
    {  
        private List < Operation > operations;  
        public CalculatorV1()
```

```
{  
operations = new List < Operation > ();  
operations.Add(new Operation() { Name = "+" ,  
NumberOperands = 2 });  
operations.Add(new Operation() { Name = "-" ,  
NumberOperands = 2 });  
operations.Add(new Operation() { Name = "/" ,  
NumberOperands = 2 });  
operations.Add(new Operation() { Name = "*" ,  
NumberOperands = 2 });  
}  
public override IList < Operation > GetOperations()  
{  
return operations;  
}  
public override double Operate(Operation operation,  
double[] operand)  
{  
switch (operation.Name)  
{  
case "+":  
return operand[0] + operand[1];  
case "-":  
return operand[0] - operand[1];  
case "/":  
return operand[0] / operand[1];  
case "*":  
return operand[0] * operand[1];  
default:  
throw new InvalidOperationException(  
String.Format("invalid operation {0}" , operation.Name));  
}  
}  
}  
}
```

### 36.2.5 计算器主机视图

下面看看主机端的主机视图。与插件视图类似，主机视图也定义了一个抽象类，其方法类似于合同。但是，这里定义的方法是由主机应用程序调用的。

类 Calculator 和 Operation 都是抽象的，因为其成员由主机适配器实现。它们只需要定义由主机应用程序使用的接口：

```
using System.Collections.Generic;  
namespace Wrox.ProCSharp.AddIns  
{
```

```
public abstract class Calculator
{
    public abstract IList < Operation > GetOperations();
    public abstract double Operate(Operation operation,
        params double[] operand);
}
public abstract class Operation
{
    public abstract string Name { get; }
    public abstract int NumberOperands { get; }
}
```

### 36.2.6 计算机主机适配器

主机适配器程序集引用了主机视图和合同，把视图映射到合同上。类

OperationContractToViewHostAdapter 实现了抽象类 Operation 的成员。CalculatorContractToViewHostAdapter 实现了抽象类 Calculator 的成员。

在 OperationContractToViewHostAdapter 中，在构造函数中指定了对合同的引用。适配器类还包含一个 ContractHandle 实例，它添加了对合同的生存期引用，所以只要主机应用程序需要，插件就保持加载状态。

```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
    internal class OperationContractToViewHostAdapter : Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
        public override string Name
        {
            get
            {
                return Contract.Name;
            }
        }
        public override int NumberOperands
```

```
{  
get  
{  
return Contract.NumberOperands;  
}  
}  
}  
internal static class OperationHostAdapters  
{  
internal static IOperationContract ViewToContractAdapter  
(Operation view)  
{  
return  
((OperationContractToViewHostAdapter)view).Contract;  
}  
internal static Operation ContractToViewAdapter(  
IOperationContract contract)  
{  
return new  
OperationContractToViewHostAdapter(contract);  
}  
}  
}  
}
```

类 CalculatorContractToViewHostAdapter 实现了抽象主机视图类 Calculator 的成员，并把调用传递给合同。该类还有一个 ContractHandle 实例，它包含对合同的引用，这类似于插件端类型转换的适配器。但这次仅需要从插件适配器向主机端的类型转换。

属性 HostAdapter 把类标记为一个需要在 HostSideAdapters 目录下安装的适配器。

```
using System.Collections.Generic;  
using System.AddIn.Pipeline;  
namespace Wrox.ProCSharp.AddIns  
{  
[HostAdapter]  
internal class CalculatorContractToViewHostAdapter :  
Calculator  
{  
private ICalculatorContract contract;  
private ContractHandle handle;  
public CalculatorContractToViewHostAdapter  
(ICalculatorContract contract)  
{  
this.contract = contract;  
handle = new ContractHandle(contract);  
}
```

```
public override IList < Operation > GetOperations()
{
    return CollectionAdapters.ToIList < IOperationContract,
        Operation > (
            contract.GetOperations(),
            OperationHostAdapters.ContractToViewAdapter,
            OperationHostAdapters.ViewToContractAdapter);
}

public override double Operate(Operation operation,
    double[] operands)
{
    return contract.Operate
        (OperationHostAdapters.ViewToContractAdapter(
            operation), operands);
}
}
```

### 36.2.7 计算器主机

示例主机应用程序使用了 WPF 技术。这个应用程序的用户界面如图 36-3 所示。其顶部是可用的插件列表。左边是活动插件的操作。选择要调用的操作时，就会显示操作数。输入了操作数的值后，就可以调用插件的操作。

底部的按钮用于重建和更新插件存储器，以及退出应用程序。



图 36-3

下面的 XAML 代码显示了用户界面的树形结构。在 `ListBox` 元素中，给项模板使用不同的样式，为插件列表、操作列表和操作数列表指定特定的表示方式。

```
< DockPanel >
< GroupBox Header="AddIn Store" DockPanel.Dock="Bottom" >
< UniformGrid Columns="4" >
< Button x:Name="rebuildStore" Click="RebuildStore" Margin="5" > Rebuild < /Button >
< Button x:Name="updateStore" Click="UpdateStore" Margin="5" > Update < /Button >
< Button x:Name="refresh" Click="RefreshAddIns" Margin="5" > Refresh < /Button >
< Button x:Name="exit" Click="App_Exit" Margin="5" > Exit < /Button >
< /UniformGrid >
< /GroupBox >
< GroupBox Header="AddIns" DockPanel.Dock="Top" >
```

```
< ListBox x:Name="listAddIns" ItemsSource="{Binding}"  
Style="{StaticResource listAddInsStyle}" />  
< /GroupBox >  
< GroupBox DockPanel.Dock="Left" Header="Operations" >  
< ListBox x:Name="listOperations"  
ItemsSource="{Binding}"  
Style="{StaticResource listOperationsStyle}" />  
< /GroupBox >  
< StackPanel DockPanel.Dock="Right"  
Orientation="Vertical" >  
< GroupBox Header="Operands" >  
< ListBox x:Name="listOperands" ItemsSource="{Binding}"  
Style="{StaticResource listOperandsStyle}" />  
< /ListBox >  
< /GroupBox >  
< Button x:Name="buttonCalculate" Click="Calculate"  
IsEnabled="False"  
Margin="5" > Calculate < /Button >  
< GroupBox DockPanel.Dock="Bottom" Header="Result" >  
< Label x:Name="labelResult" />  
< /GroupBox >  
< /StackPanel >  
< /DockPanel >
```

#### 提示：

项模板的内容可参见第 35 章。

在后台代码中 `FindAddIns()` 方法在 `Window` 的构造函数中调用。`FindAddIns()` 方法使用 `AddInStore` 类获得 `AddInToken` 对象的集合，把它们传送给列表框 `listAddIns` 的 `DataContext` 属性，以显示它们。`AddInStore.FindAddIns()` 方法的第一个参数传送主机视图定义的抽象类 `Calculator`，从存储器中查找应用于合同的所有插件。第二个参数传送从应用程序配置文件中读取的管道目录。运行 Wrox 下载网站上的示例应用程序时，需要修改应用程序配置文件中的目录，以匹配自己的目录结构。

```
using System;  
using System.AddIn.Hosting;  
using System.AddIn.Pipeline;  
using System.IO;  
using System.Linq;  
using System.Windows;  
using System.Windows.Controls;  
using Wrox.ProCSharp.AddIns.Properties;  
namespace Wrox.ProCSharp.AddIns  
{  
    public partial class CalculatorHostWindow : Window  
    {
```

```
private Calculator activeAddIn = null;
private Operation currentOperation = null;
public CalculatorHostWindow()
{
    InitializeComponent();
    FindAddIns();
}
void FindAddIns()
{
    try
    {
        this.listAddIns.DataContext =
            AddInStore.FindAddIns(typeof(Calculator),
            Settings.Default.PipelinePath);
    }
    catch (DirectoryNotFoundException ex)
    {
        MessageBox.Show("Verify the pipeline directory in the "
        +
        "config file");
        Application.Current.Shutdown();
    }
}
//...
```

要更新 Add-In 存储器的高速缓存 ,UpdateStore()和 RebuildStore()方法应映射到 Update 和 Rebuild 按钮的 Click 事件上。在这些方法的实现代码中 , 使用了 AddInStore 类的 Update()和 Rebuild()方法。如果程序集存储在错误的目录下 , 这些方法就返回一个警告字符串数组。由于管道结构比较复杂 , 第一次把程序集复制到正确的目录下时 , 其项目配置不太可能完全正确。阅读这些方法返回的信息 , 可以清楚地了解错在何处。例如 , 信息"在程序集\Pipeline\AddInSideAdapters\CalcView.dll 中没有找到可用的 AddInAdapter 部分"表示 , 程序集 CalcView 存储在错误的目录下。

```
private void UpdateStore(object sender, RoutedEventArgs
e)
{
    string[] messages =
        AddInStore.Update(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
        "AddInStore Warnings", MessageBoxButton.OK,
        MessageBoxImage.Warning);
    }
}
private void RebuildStore(object sender, RoutedEventArgs
```

```
e)
{
    string[] messages =
    AddInStore.Rebuild(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
        "AddInStore Warnings", MessageBoxButton.OK,
        MessageBoxIcon.Warning);
    }
}
```

在图 36-2 中，可用插件的旁边有一个 Activate 按钮。单击这个按钮会调用处理方法 ActivateAddIn()。在这个方法的代码中，使用 AddInToken 类的 Activate()方法激活插件。其中插件加载到用 AddInProcess 类创建的一个新进程中。AddInProcess 类启动了进程 AddInProcess32.exe。把该进程的 KeepAlive 属性设置为 false，则只要最后一个插件引用被垃圾回收了，该进程就停止。参数 AddInSecurityLevel.Internet 使插件在有限的权限下运行。ActivateAddIn()的最后一个语句调用 ListOperations()方法，ListOperations()方法又调用插件的 GetOperations()方法。GetOperations()方法把返回的列表赋予列表框 listOperations 的 DataContext 属性，显示所有的操作。

```
private void ActivateAddIn(object sender, RoutedEventArgs
e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "ActivateAddIn invoked from the
wrong " +
    "control type");

    AddInToken addIn = el.Tag as AddInToken;
    Trace.Assert(el.Tag != null, String.Format(
    "An AddInToken must be assigned to the Tag property " +
    "of the control {0}", el.Name));
    AddInProcess process = new AddInProcess();
    process.KeepAlive = false;

    activeAddIn = addIn.Activate < Calculator > (process,
    AddInSecurityLevel.Internet);
    ListOperations();
}

void ListOperations()
{
    this.listOperations.DataContext =
    activeAddIn.GetOperations();
}
```

激活插件，在 UI 上显示操作列表后，用户就可以选择操作了。Operations 类别中按钮的 Click 事件赋予处理程序方法 OperationSelected()。在这个方法的代码中，检索赋予了按钮的 Tag 属性的 Operation 对象，获得操作所需要的操作数个数。为了让用户给操作数添加值，把一个 OperandUI 对象数组绑定到列表框 listOperands 上。

```
private void OperationSelected(object sender,
RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from
    " +
    "the wrong control type");
    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name));
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
    operands.Select((operand, index) =>
    new OperandUI()
    { Index = index + 1, Value = operand }).ToArray();
}
```

在 Calculate 按钮的 Click 事件中调用了 Calculate()方法。这里操作数从 UI 中提取，把操作和操作数传送给插件的 Operate()方法，结果显示为标签的内容：

```
private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI =
    (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui =>
    opui.Value).ToArray();
    labelResult.Content =
    activeAddIn.Operate(currentOperation,
    operands);
}
```

### 36.2.8 其他插件

艰苦的工作已经完成了。管道组件和主机应用程序都创建好了。管道现在可以工作了，还可以在主机应用程序中添加其他插件，例如下面的 Advanced Calculator 插件：

```
[AddIn("Advanced Calc", Publisher = "Wrox Press", Version  
= "1.1.0.0",  
Description = "Another AddIn Sample")]  
public class AdvancedCalculatorV1 : Calculator
```

### 36.3 小结

本章学习了.NET 3.5 技术的一个新概念：Managed AddIn Framework。MAF 使用管道概念使主机程序集和插件程序集的创建完全独立。清楚定义的合同隔离开了主机视图和插件视图。适配器可以使主机端和插件端独立地修改。

下一章是介绍用 ASP.NET 开发 UI 的 3 章中的第一章。

## 第 37 章 ASP.NET 页面

如果您是 C# 和 .NET 的新手，肯定不理解为什么本书要包含介绍 ASP.NET 的内容。这是一种全新的语言，对吗？但实际上并非如此。使用 C# 可以创建 ASP.NET 页面。

ASP.NET 是 .NET Framework 的一部分。在通过 HTTP 请求建立文档时，它可以在 Web 服务器上动态创建文档。该文档主要是 HTML 和 XHTML 文档，但也可以创建 XML 文档、CSS 文件、图像、PDF 文档，或者支持 MIME 类型的文档。

在某些方面，ASP.NET 类似于许多其他技术，例如 PHP、ASP、ColdFusion 等，但它们有一个重要的区别。顾名思义，ASP.NET 可以与 .NET Framework 完全集成，它包含了对 C# 的支持。

您可能使用过动态生成内容的 ASP 技术。这种技术使用脚本语言，例如 VBScript 或 JScript 来编程，结果却不是很好。但对于那些习惯于“正确的”已编译编程语言的人来说，这种技术很笨拙，肯定会导致性能的损失。

与更高级的编程语言相比，一个主要区别是 ASP.NET 提供了完整的服务器端对象模型，可以在运行期间使用。ASP.NET 可以在其环境中把页面上的所有控件作为对象来访问。在服务器端，还可以访问其他.NET 类，与许多有用的服务集成起来。在页面上使用的控件有许多功能，实际上可以完成 Windows Forms 类的几乎所有的功能，有非常大的灵活性。因此，生成 HTML 内容的 ASP.NET 通常称为 Web 窗体。

本章将详细介绍 ASP.NET，包括 ASP.NET 如何工作，ASP.NET 可以完成什么任务，以及什么地方适合使用 C#。下面是本章的主要内容：

ASP.NET 简介

如何使用服务器控件创建 ASP.NET Web 窗体

如何使用 ADO.NET 把数据绑定到 ASP.NET 控件上

应用程序配置

### 37.1 ASP.NET 概述

ASP.NET 使用 Internet Information Server(IIS) 来传送内容，以响应 HTTP 请求。ASP.NET 页面在.aspx 文件中，其基本结构如图 37-1 所示。

在 ASP.NET 处理过程中，可以访问所有的.NET 类、C#或其他语言创建的定制组件、数据库等。实际上，这与运行 C# 应用程序一样；在 ASP.NET 中使用 C# 就是在运行 C# 程序。

ASP.NET 文件可以包含下述内容：

服务器的处理指令

C#、VB.NET、JScript.NET 代码或.NET Framework 支持的其他语言的代码

对应已生成资源的窗体内容，例如 HTML

客户端的脚本代码

内嵌的 ASP.NET 服务器控件

实际上，ASP.NET 文件也可以很简单，如下所示。

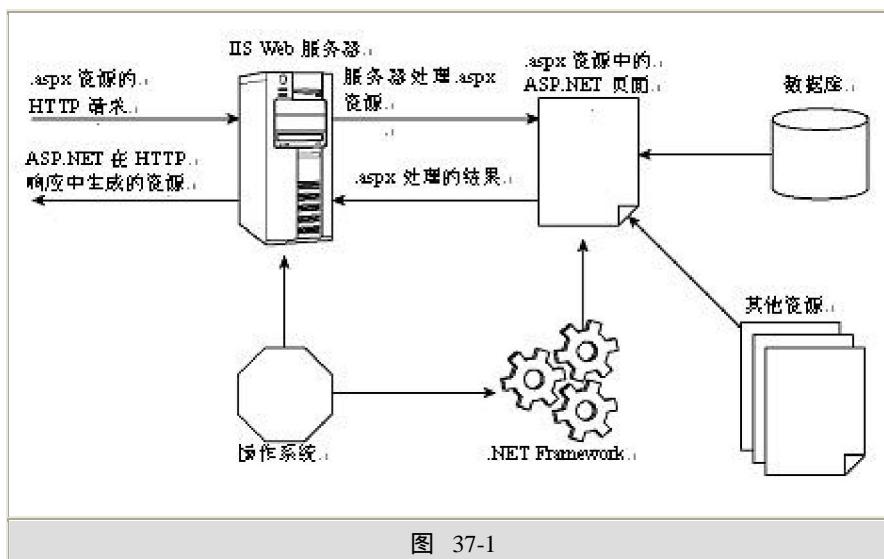


图 37-1

Hello!

结果很简单，返回一个只包含这个文本的 HTML 页面(因为 ASP.NET 页面的默认输出是 HTML)。

本章后面会提到，也可以把代码的某些部分分解为其他文件，这可以提供更合理的结构。

### ASP.NET 中的状态管理

ASP.NET 页面的一个重要属性是它们是无状态的。在默认情况下，在用户的请求之间，并没有信息存储在服务器上(但有一些方法可以完成存储信息的任务，详见下面的内容)。这初看起来有点奇怪，因为状态管理对于用户友好的交互会话是非常重要的。但是，ASP.NET 提供了一种相当好的方式来解决这个问题，使会话管理几乎是完全透明的。

简言之，Web 窗体上控件的状态信息(包括文本框中输入的数据、下拉列表中的选项等)存储在隐藏的 viewstate 字段中，这个字段是服务器生成的页面的一部分，并传送给用户。后续的操作称为回送(postback)，例如触发需要服务器端处理的事件，提交窗体数据，把这些信息传送回服务器。在服务器上，这些信息用于重新填充页面对象模型，以便操作它，就像在本地进行修改一样。

稍后详细介绍这个主题。

## 37.2 ASP.NET Web 窗体

如前所述，ASP.NET 中的许多功能是使用 Web 窗体实现的。稍后我们将创建一个简单的 Web 窗体，深入介绍这种技术。但这里先简要介绍 Web 窗体的设计。注意许多 ASP.NET 开发人员仅使用文本编辑器(例如 Notepad)创建文件。这里不推荐这么做，因为 Visual Studio 或 Web Developer Express 等 IDE 提供的优点是很重要的，只是使用 Notepad 等文本编辑器是创建文件的一种方法，所以这里提及它。如果使用文本编辑器，在把 Web 应用程序的哪些部分放在什么地方等方面有非常大的灵活性，例如，可以把所有代码都组合到一个文件中。把代码放在<script>标记中，在起始<script>标记中使用两个属性，如下所示：

```
<script language="c#" runat="server">  
// Server-side code goes here.
```

```
</script>
```

这里的 runat="server" 属性是很重要的，因为它指示 ASP.NET 引擎在服务器上执行这段代码，而不是把它传送给客户，因此可以访问前面讨论的环境。我们可以在服务器端脚本块中放置函数、事件处理程序等。

如果省略 runat="server" 属性，就是在提供客户端代码，如果使用本章后面要介绍的服务器端编码方式，就会失败。但是，可以使用<script>元素提供 JavaScript 等语言编写的客户端脚本。例如：

```
<script language="JavaScript" type="text/JavaScript">  
// Client-side code goes here; we can also use "vbscript".  
</script>
```

注意：

type 属性是可选的，但如果需要兼容 XHTML，它就是必需的。

在页面中添加 JavaScript 代码的功能也包含在 ASP.NET 中，这好像有点奇怪。但是，JavaScript 允许给 Web 页面添加动态的客户端操作，这是非常有用的。Ajax 编程就允许添加 JavaScript 代码，详见第 39 章。

可以在 Visual Studio 中创建 ASP.NET 文件。这是非常重要的，因为我们已经熟悉了在这个环境中进行 C# 编程。在这个环境中，Web 应用程序的默认项目设置提供了一种比单个.aspx 文件略微复杂的结构，使之更富于逻辑(更接近编程，而不像 Web 开发)。据此，本章将使用 Visual Studio 进行 ASP.NET 编程(而不是 Notepad)。

.aspx 文件也可以包含括在<% 和 %>标记中的代码块。但是，函数定义和变量声明不能放在这里。可以插入代码，当执行到块时就执行这些代码。当输出简单的 HTML 内容时，这是很有效的。这种方式类似于旧风格的 ASP 页面，但有一个重要的区别：代码是已经编译好的，不是解释性的。这样，性能会好得多。

下面举一个示例。要创建一个新的 Web 应用程序，在 Visual Studio 中，使用 File | New | Web Site 菜单项，打开一个对话框。在对话框中选择 Visual C# 语言类型和 ASP.NET Web Site 模板，现在要进行选择：Visual Studio 可以在几个不同的位置创建 Web 站点：

本地 IIS Web 服务器上

本地磁盘上，它配置为使用内置的 Visual Web Developer Web 服务器

可通过 FTP 访问的任意位置

支持 Front Page Server Extensions 的远程 Web 服务器上

不必考虑后两个选项，它们使用远程服务器，所以现在应选择前两项。一般情况下，IIS 是安装 ASP.NET Web 站点的最佳位置，因为它最接近部署 Web 站点时需要的配置。另一个选项使用内置的 Web 服务器，适合于测试，但有一些限制：

只有本地计算机能访问 Web 站点

访问 SMTP 等服务受到限制

安全模型与 IIS 不同：应用程序运行在当前用户的账户下，而不是运行在 ASP.NET 的特定账户下

最后一点需要澄清，因为在访问数据库或其他需要验证身份的数据时，安全性是非常重要的。在默认情况下，运行在 IIS 上的 Web 应用程序会在 Windows XP、2000 和 Vista Web 服务器的 ASPNET 账户下运行，或在 Windows Server 2003 的 NETWORK SERVICES 账户下运行。如果使用 IIS，这是可以配置的，但如果使用内置的 Web 服务器，就不能配置它。

为了便于演示，或者计算机上可能没有安装 IIS，则可以使用内置的 Web 服务器。在这个阶段不必担心安全性，只需选择它即可。

在 C:\ProCSharp\Chapter37 目录下使用 File System 选项创建一个新的 ASP.NET Web Site，称为 PCSWebApp1。如图 37-2 所示。

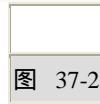


图 37-2

稍后，Visual Studio 应建立如下内容：

新的解决方案 PCSWebApp1，包含 C# Web 应用程序 PCSWebApp1

保留文件夹 App\_Data，包含数据文件，例如 XML 文件或数据库文件

Default.aspx，Web 应用程序中的第一个 ASP.NET 页面

Default.aspx.cs，Default.aspx 的后台代码类文件

Web.config，Web 应用程序的配置文件

这些都可以在 Solution Explorer 中看到，如图 37-3 所示。



图 37-3

可以在设计视图或源代码(HTML)视图中查看.aspx 文件。这与 Windows 窗体(参见第 31 章)完全相同。Visual Studio 中的起始视图是 Default.aspx 的设计或源代码视图(使用左下角的按钮可以切换视图)。设计视图如图 37-4 所示。



在窗体(当前为空)的下面，可以看到在窗体的 HTML 中光标当前的位置。这里光标在<form>元素的<div>元素中，<form>元素在页面的<body>元素的，显示为<form#form1>，用它的 id 属性表示。<div>元素也显示在设计视图中。

页面的源代码视图显示了在.aspx 文件中生成的代码：

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<div>
</div>
</form>
</body>
</html>
```

如果读者熟悉 HTML 语法，就会觉得这些代码很眼熟。这里列出了 HTML 页面中遵循 XHTML 模式的基本代码，并包含几行额外的代码。最重要的元素是<form>，它的 id 属性是 form1，包含了 ASP.NET 代码。这里最重要的属性是 runat。与本节前面的服务器端代码块一样，这个属性设置为 server，表示窗体的处理将在服务器上进行。如果没有包含这个属性，就不会在服务器端上完成任何处理，窗体也不会执行任何操作。在 ASP.NET 页面中，只有一个服务器端<form>元素。

这段代码中另一个比较重要的东西是顶部的<%@ Page %>标记，它定义了对于 C# Web 应用程序开发人员来说非常重要的页面特性。首先，language 属性指定在页面中使用 C# 语言，与前面的<script>块一样(Web 应用程序默认的语言是 VB，使用 Web.config 配置文件可以修改这个属性)。下面的三个属性 AutoEventWireup、CodeFile 和 Inherits 用于把 Web 窗体关联到后台代码文件中的一个类上，这里是 Default.aspx.cs 文件中的部分类\_Default。这就需要讨论 ASP.NET 代码模型了。

### 37.2.1 ASP.NET 代码模型

在 ASP.NET 中，布局(HTML)代码、ASP.NET 控件和 C#代码用于生成用户看到的 HTML。布局和 ASP.NET 代码存储在.aspx 文件中，也就是上一节的.aspx 文件。用于定制窗体操作的 C#代码包含在.aspx 文件中，也可以像前面的例子那样，放在单独的.aspx.cs 文件中，通常称为后台编码文件。

在处理 ASP.NET Web 窗体时，一般在用户请求页面时，预编译站点，此时会发生几个事件：

ASP.NET 处理器执行页面，确定必须创建什么对象，以实例化页面对象模型。

动态创建一个基类，包括页面上的控件成员和这些控件的事件处理程序(例如按钮单击事件)。

包含在.aspx 页面中的其他代码，与这个基类合并，构成完整的对象模型

编译所有的代码，并高速缓存起来，以备处理以后的请求

生成 HTML，返回给用户

在 Web 站点 PCSWebApp1 中，为 Default.aspx 生成的后台代码文件的内容最初非常少。首先看看需要在 Web 页面上使用的默认命名空间引用的集合：

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
```

在这些引用的下面，Default\_aspx 部分类的定义几乎是空的：

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}
```

这里可以使用 Page\_Load()事件处理程序添加加载页面时需要的代码。在添加事件处理程序时，这个类文件会包含越来越多的代码。注意没有把这个事件处理程序关联到页面上的代码，这是由 ASP.NET 运行库处理的。这要归功于 AutoEventWireup 属性，把它设置为 false，表示必须自己在代码中把事件处理程序与事件关联起来。

这个类是一个部分类定义，因为前面介绍的过程需要它。在预编译页面时，会从页面的 ASP.NET 代码中创建一个单独的部分类定义，这包括添加到页面上的所有控件。在设计期间，编译器会推断这个部分类定义，以便在后台代码中使用 IntelliSense，来引用页面上的控件。

### 37.2.2 ASP.NET 服务器控件

前面生成的代码并不能完成许多工作，所以下面就应添加一些内容。在 Visual Studio 中使用 Web 窗体设计器，它支持拖放操作，其方式与 Windows 窗体设计器相同。

可以添加到 ASP.NET 页面上的控件有 3 种类型：

HTML 服务器控件-- 这些控件模拟 HTML 元素，HTML 开发人员会很熟悉它们。

Web 服务器控件-- 这是一组新的控件，其中一些控件的功能与 HTML 控件相同，但它们的属性和其他元素有一个公共的命名模式，便于进行开发，而且可以与相似的 Windows 窗体控件保持一致。还有一些全新的、非常强大的控件，如本章后面所述。Web 服务器控件有几种类型，包括标准控件，如按钮、验证用户输入的验证控件、简化用户管理的登录控件，和处理数据源的一些较复杂的控件。

定制控件和用户控件-- 由开发人员定义的控件，我们可以用第 38 章介绍的许多方式来定义它们。

提示：

下一节列出了常用 Web 服务器控件及其使用说明的完整列表。下一章将介绍其他控件。本章没有介绍 HTML 控件。这些控件提供的功能，Web 服务器也能提供，而且 Web 服务器控件为熟悉编程的开发人员提供了一个功能比 HTML 更丰富的环境。学会如何使用 Web 服务器控件后，使用 HTML 服务器控件就不难了。也可以参阅清华大学出版社出版的《ASP.NET 2.0 高级编程》。

下面在上一节创建的 Web 站点 PCSWebApp1 中，添加两个 Web 服务器控件。所有的 Web 服务器控件都以下述 XML 元素的方式使用：

```
<asp:controlName runat="server"  
attribute="value">Contents</asp:controlName>
```

其中 controlName 是 ASP.NET 服务器控件的名称，attribute="value" 是一个或多个属性规范，Contents 指定控件的内容。一些控件可以使用属性和控件元素的内容来设置属性，例如 Label(用于显示简单文本)，其文本可以用两种方式指定。其他控件可以使用元素包含模式来定义它们的层次结构，例如 Table(定义一个表)可以包含 TableRow 元素，指定表中的行。

注意，控件的语法是基于 XML 的(它们也可以内嵌在非 XML 代码中，例如 HTML)。省略闭合标记、表示空元素的 />，或者重叠控件，都会产生错误。

最后，再看看 Web 服务器控件上的 runat="server" 属性。把它放在这里和放在其他地方是一样的，遗漏这个属性也会产生错误，结果将是一个不能运行的 Web 窗体。

第一个示例应简单一些。修改 Default.aspx 的 HTML 设计视图，代码如下。

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<div>
<asp:Label runat="server" ID="resultLabel" /><br />
<asp:Button runat="server" ID="triggerButton"
Text="Click Me" />
</div>
</form>
</body>
</html>
```

这里添加了两个 Web 窗体控件：标签和按钮。

注意：

在添加控件时，Visual Studio 的 IntelliSense 会提示代码输入项，这与 C# 代码编辑器一样。如果在隔开的视图中编辑代码，再同步视图，在源代码面板上编辑的元素会在设计面板上突出显示。

回过头来看看设计屏幕，其中已经添加了控件，并用它们的 ID 属性命名（ID 属性常常称为控件的标识符）。与 Windows 窗体一样，可以通过 Properties 窗口访问所有的属性、事件等，如果进行了修改，代码或设计会通过 Properties 窗口立即反馈回来。

注意：

也可以使用 CSS 属性窗口和其他样式窗口，给控件指定样式。但除非很熟悉 CSS，否则现在不要使用这个技术，而是应关注控件的功能。

我们添加的所有服务器控件都会自动成为对象模型的一部分，该对象模型是在这段后置代码中为窗体构建的。Windows 窗体开发人员可以即时得到这个对象模型，并开始认识到它与 Windows 窗体的类似性。

要让这个应用程序完成一些工作，应添加单击按钮的事件处理程序。可以在 Properties 窗口中为按钮输入一个方法名，也可以双击该按钮，得到默认的事件处理程序。如果双击按钮，就可以自动添加一个事件处理方法：

```
protected void triggerButton_Click(object sender,
```

```
EventArgs e)
{
}
```

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```
<div>
<asp:Label Runat="server" ID="resultLabel" /><br />
<asp:Button Runat="server" ID="triggerButton"
Text="Click Me"
OnClick="triggerButton_Click" />
</div>
```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click()中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```
void triggerButton_Click(object sender, EventArgs
e)
{
resultLabel.Text = "Button clicked!";
}
```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这就很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。



图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中)快速查看一下浏览器接收到的代码。<form>部分应如下所示：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE="
/>
</div>
<div>
<span id="resultLabel"></span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION"
id="__EVENTVALIDATION"
value="/wEWAqK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />
</div>
</form>
```

Web 服务器控件生成了 HTML，和分别代表和。还有一个名为 VIEWSTATE 的字段，把前面提到的窗体状态封装起来。在窗体传送回服务器以重新创建 UI，以及跟踪改变时使用这些信息。注意`元素已经进行了配置，通过 HTTP POST 操作(在 method 中指定)把数据传回 Default.aspx(在 action 中指定)，它还被赋予了一个名称 form1。`

在单击按钮，查看文本后，可再次浏览源 HTML(下面添加了必要的空格，使代码比较清晰)：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHRvbIB
jbGlja2VkJWRkZN3zQXZqDnF2ddEBw4Kj7MEqj9pJ" />
</div>
<div>
<span id="resultLabel">Button clicked!</span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAqLl3MPHAwLHpP+yCwFfz4kBL6+KP1xjb0lgrAageely" />
</div>
</form>
```

这次 viewstate 的值包含比较多的信息，因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上，这可能是一个非常长的字符串，但这是由系统在后台完成的，我们几乎可以不

考虑状态管理，只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时，可以禁用不需要保留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态，以提高性能，就可以禁用整个页面的 Viewstate。

注意：

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译，把 Default.aspx.cs 中的文本"Button clicked!"改为其他内容，保存文件，再次单击按钮。Web 页面上的文本会做相应的改变。

### 1. 控件面板

本节介绍可用控件，之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别，如图 37-6 所示。

注意，在控件的描述中使用了"属性"-- ASP.NET 代码中使用的属性与它同名。这里的引用并不完整，许多控件和属性都没有介绍，只介绍了最常用的属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation and Login 和 WebParts 类别在第 38 章介绍，AJAX 扩展控件在第 39 章介绍，Reporting 控件可以在 Web 页面上报告信息，包括 Crystal Reports，本书不讨论。



图 37-6

#### (1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI. WebControls. WebControl，而 System.Web.UI.WebControls.WebControl 又继承了 System.Web. UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类，而该基类又最终派生于 Control。因此，Web 服务器控件有许多共同的属性和事件，如果需要，就可以使用这些属性和事件。这里不可能介绍所有的元素，只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式，这是很容易控制的，例如属性 ForeColor、BackColor、Font 等，也可以使用 CSS(Cascading Style Sheet)类来控制。此时，应在一个独立的文件中，把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括：Width 和 Height，用于设置控件的大小；AccessKey 和 TabIndex，便于用户的交互操作；Enabled，设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件，在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件，使用 Parent 可以访问控件的容器。

对于事件，最常用的是继承来的 Load 事件，它执行控件的初始化，PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多，下一章将详细介绍它们，尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控件	说明
Label	显示简单文本，使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据，TextChanged 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮)就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本，Click 事件用于响应单击(服务器回送是自动的)。也可以使用 Command 事件响应单击，该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同，但把按钮显示为超链接

(续表)

控件	说明
ImageButton	显示一个图像，该图像放大一倍作为一个可单击的按钮，其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地，用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像，用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件，如果在链接后要执行其他处理，就应使用 LinkButton
DropDownList	允许用户选择一个列表项，可以直接从列表中选择，也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类)，SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变，这个控件也有 AutoPostBack 属性，所以选项的改变会触发一个回送操作
ListBox	允许用户从列表中选择一个或多个列表。把SelectionMode 设置为 Multiple 或 Single，可以确定一次选择多少个选项，Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中，与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送，CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同，例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下，它们都组合在一个组中，其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮，在这个组中，一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用，如果图像加载失败，由 AlternateText

	提供对应的文本
ImageMap	类似于 Image ,但在用户单击图像中的一个或多个热区时 ,可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供 ,例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、 TableRow 和 TableCell ,或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样 ,这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同 ,这个控件有一个 Click 事件 ,用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控件	说 明
HiddenField	用于提供隐藏的字段 ,以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能 ,但没有样式属性 ,只有一个 Text 属性(因为它派生于 Control ,而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性 ,但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.Date Time)来访问由用户选择的日期和月份 ,并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged 。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后 ,显示另一个图像。使用 Advertisement- File 属性指定描述图像的 XML 文件 , AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮 ,以选择要上传的文件。用户选择了文件之后 ,就可以使用 HasFile 属性确定是否选择了文件 ,然后使用后台代码中的 SaveAs() 方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤 ,按顺序或不按顺序显示给用户 ,并依赖此控件来维护状态
Xml	这是一个更复杂的文本显示控件 ,用于显示用 XSLT 样式表传输的 XML 内容 ,这些 XML 内容是使用 Document 、 DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的 ,XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件 ,每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定 ,如果视图改变了(可能因为单击了当前视图上的 Next 链接) ,就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出 ,但可以方便地把其他控件组合在一起 ,或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器 ,类似于 PlaceHolder ,但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View ,使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一起高速缓存的 Web 页面 ,这是一个与 ASP.NET 高速缓存相关的高级主题 ,本书不涉及
Localize	与 Literal 相同 ,但允许使用项目资源指定要在不同区域显示的文本 ,使文本本地化

## (2) 数据 Web 服务器控件

数据 Web 服务器控件分为两类：

数据源控件(SqlDataSource、 AccessDataSource、 ObjectDataSource、 XmlDataSource、 和 SiteMapDataSource)

数据显示控件(GridView、 DataList、 DetailsView、 FormView、 Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

表 37-2 描述了各种数据源控件。注意本节没有探讨属性，这主要是因为这些控件最好通过图形化的向导来配置。本章的后面将使用这些控件，使读者更好地理解它们的工作方式。

表 37-2

控 件	说 明
SqlDataSource	用作 SQL Server 数据库中存储的数据的管道。把这个控件放在页面上，就可以使用数据显示控件操作 SQL Server 数据。本章后面将使用这个控件
AccessDataSource	与 SqlDataSource 相同，但处理存储在 Microsoft Access 数据库中的数据
LinqDataSource	这个控件可以处理支持 LINQ 数据模型的对象
ObjectDataSource	这个控件可以处理存储在自己创建的对象中的数据，这些对象可能组合在一个集合类中。这是把定制的对象模型显示在 ASP.NET 页面上的非常快捷的方式
XmlDataSource	可以绑定到 XML 数据上。它可以绑定导航控件，例如 TreeView。利用这个控件，还可以使用 XSL 样式表传输 XML 数据
SiteMapDataSource	可以绑定到层次站点地图数据上。详见第 38 章的导航 Web 服务器控件

接着是数据显示控件，如表 37-3 所示。其中几个控件可以满足各种需求。一些控件的功能比另外一些控件强，但我们常常使用最简单的控件(例如不需要编辑数据项)。

表 37-3

控 件	说 明
GridView	以数据行的格式显示多个数据项(例如数据库中的行)，其中每一行包含表示数据字段的列。利用这个控件的属性，可以选择、排序和编辑数据项
DataList	显示多个数据项，可以为每一项提供模板，以任意指定的方式显示数据字段。与 GridView 一样，可以选择、排序和编辑数据项
DetailsView	以表格形式显示一个数据项，表中的每一行都与一个数据字段相关。这个控件可以添加、编辑和删除数据项
FormView	使用模板显示一个数据项。与 DetailsView 一样，这个控件也可以添加、编辑和删除

W	数据项
Repeater	与 DataList 相同 , 但不能选择和编辑数据
Repeater	显示报表服务数据的高级控件 , 本书不涉及
Viewer	

### (3) 验证 Web 服务器控件

验证控件可以在不编写任何代码的前提(在大多数情况下)下验证用户的输入。只要有回送 , 每个验证控件就会检查控件是否有效 , 并相应地改变 IsValid 属性的值。如果这个属性是 false , 被验证控件的用户输入就没有通过验证。包含所有控件的页面也有一个 IsValid 属性-- 如果页面中任一个有效性验证控件的 IsValid 属性设置为 false , 该页面的 IsValid 属性就是 false。可以在服务器端的代码上检查这个属性 , 并对它进行操作。

验证控件还有第二个功能。它们不仅可以在运行期间验证控件的有效性 , 还可以自动给用户输出有帮助的提示 , 把 ErrorMessage 属性设置为希望的文本 , 在用户试图回送无效的数据时 , 就会看到这些文本。

存储在 ErrorMessage 中的文本可以在验证控件所在的位置输出 , 也可以和页面上其他验证控件的信息一起输出在一个独立的位置。第二种方式可以使用 ValidationSummary 控件来获得 , 并把所有的错误信息和附加文本按照需要显示出来。

在支持这些控件的浏览器中 , 验证控件甚至可以生成客户端的 JavaScript 函数 , 来简化验证任务的执行。在某些情况下 , 是不会有回送的 , 因为验证控件在某些环境下禁止回送 , 输出错误信息 , 而不涉及服务器的执行。

所有的验证控件都继承于 BaseValidator , 所以它们共享几个重要的属性。最重要的是上面讨论的 ErrorMessage 属性 ; ControlToValidate 属性也是比较重要的 , 它指定要验证的控件的编程 ID。另一个重要的属性是 Display , 它确定是把文本放在验证汇总的位置上(该属性设置为 none) , 还是放在验证控件的位置上。也可以给错误信息留一些空间 , 即不显示这些错误信息(把 Display 设置为 Static) , 或者按照需要给这些信息动态分配空间 , 这会使页面的内容有轻微的改变(把 Display 设置为 Dynamic)。表 37-4 描述了各个验证控件。

表 37-4

控 件	说 明
RequiredFieldValidator	如果用户在 TextBox 等控件中输入数据 , 就检查这些数据
CompareValidator	用于检查输入的数据是否满足简单的要求。利用一个运算符集合 , 通过 Operator 和 ValueToCompare 属性进行验证。Operator 的值可以是 Equal、GreaterThan、GreaterThanOrEqual、LessThan、LessThanOrEqual、NotEqual 或 DataTypeCheck。DataTypeCheck 可以比较 ValueToCompare 的数据类型和控件中要验证的数据。ValueToCompare 是一个字符串属性 , 但根据其内容可以把它解释为另一种数据类型。要进一步比较控件 , 可以把 type 属性设置为 Currency、Date、Double、Integer 或 String
Range Validator	验证控件中的数据 , 看看其值是否在 MaximumValue 和 MinimumValue 属性值之间 , 其 Type 属性对应于每个 CompareValidator

RegularExpressionValidator	根据存储在 ValidationExpression 中的正则表达式验证字段的内容，可以用于验证邮政编码、电话号码、IP 号码等
----------------------------	--

(续表)

控件	说明
CustomValidator	使用定制函数验证控件中的数据。ClientValidationFunction 指定用于验证一个控件的客户端函数(这表示我们不能使用 C#)。这个函数应返回一个 Boolean 类型的值，表示验证是否成功。另外，还可以使用 ServerValidate 事件指定用于验证数据的服务器端函数。这个函数是一个 bool 类型的事件处理程序，其参数是一个包含要验证数据的字符串，而不是 EventArgs 参数。如果验证成功，就返回 true，否则返回 false
ValidationSummary	为所有设置了 ErrorMessage 的验证控件显示验证错误。通过设置 DisplayMode (BulletList、List 或 SingleParagraph) 和 HeaderText 属性，其显示的内容可以格式化；把 ShowSummary 设置为 false，就会禁止显示；把 ShowMessageBox 设置为 true，内容就会显示在弹出的消息框中

## 2. 服务器控件的示例

在这个示例中，要为一个 Web 应用程序(会议室登记工具)创建构架。与本书的其他示例一样，可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载示例应用程序的代码。现在仅介绍前端和简单的事件处理，后面将使用 ADO.NET 和数据绑定扩展这个示例，使之包含服务器端的事务逻辑。

要创建的 Web 窗体包含的字段有：用户名、事件名、会议室和参加者，以及可从中选择日期的一个日历(假定本例的作用是处理日常事件)。除了日历外，所有的字段都使用验证控件，只有日在服务器端验证，并提供一个默认日期，以防没有输入日期。

为了测试用户界面，窗体上也提供了一个 Label 控件，使用它可以显示提交的结果。

首先，在 Visual Studio 中，在 C:\ProCSharp\Chapter37\目录下创建一个新的 Web 站点，命名为 PCSWebApp2。然后修改 Default.aspx 中的代码：

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Meeting Room Booker</title>
</head>
<body>
<form id="form1" runat="server">
<div>
```

```
<h1 style="text-align: center;">  
Enter details and set a day to initiate an event.  
</h1>  
</div>
```

页面的标题是用 HTML 标记 `<h1>` 括起来的，以得到大型的标题格式的文本，之后，窗体的主体放在 HTML 标记 `<table>` 中。可以使用一个 Web 服务器控件表格，但这会导致不必要的复杂性，因为使用表格的目的仅仅是为了格式化显示而已，不是用于动态的 UI 元素(在设计 Web 窗体时，不要添加不必要的服务器控件)。这个表格有 3 列，第 1 列包含简单的文本标签，第 2 列包含对应于文本标签的 UI 字段(以及这些字段的验证控件)，第 3 列包含一个日期控件，可以从中选择日期，这个控件占用了 4 行。第 5 行包含一个跨越所有列的提交按钮，第 6 行包含一个 ValidationSummary 控件，在需要时可以显示错误信息(所有其他验证控件都设置了 `Display="None"`，因为它们都使用这个汇总来显示错误)。在表的下面是一个简单的标签，使用它可以显示结果，以后我们还将添加数据库访问。

```
<div style="text-align: center;">  
<table style="text-align: left; border-color: #000000;  
border-width: 2px;  
background-color: #ffff99e;" cellspacing="0"  
cellpadding="8" rules="none"  
width="540">  
<tr>  
<td valign="top">  
Your Name:</td>  
<td valign="top">  
<asp:TextBox ID="nameBox" Runat="server" Width="160px" />  
<asp:RequiredFieldValidator ID="validateName"  
Runat="server"  
ErrorMessage="You must enter a name."  
ControlToValidate="nameBox" Display="None" />  
</td>  
<td valign="middle" rowspan="4">  
<asp:Calendar ID="calendar" Runat="server"  
BackColor="White" />  
</td>  
</tr>  
<tr>  
<td valign="top">  
Event Name:</td>  
<td valign="top">  
<asp:TextBox ID="eventBox" Runat="server" Width="160px"  
/>  
<asp:RequiredFieldValidator ID="validateEvent"  
Runat="server"  
ErrorMessage="You must enter an event name."  
ControlToValidate="eventBox" Display="None" />
```

```
</td>  
</tr>
```

这个文件中的大多数 ASP.NET 代码都非常简单，许多代码只要浏览一遍就可以理解。特别要注意的是，在代码中用于选择会议室和多个会议参加者的列表项目是如何附加到控件的：

```
<tr>  
<td valign="top">  
Meeting Room:</td>  
<td valign="top">  
<asp:DropDownList ID="roomList" Runat="server"  
Width="160px">  
<asp:ListItem Value="1">The Happy Room</asp:ListItem>  
<asp:ListItem Value="2">The Angry Room</asp:ListItem>  
<asp:ListItem Value="3">The Depressing  
Room</asp:ListItem>  
<asp:ListItem Value="4">The Funked Out  
Room</asp:ListItem>  
</asp:DropDownList>  
<asp:RequiredFieldValidator ID="validateRoom"  
Runat="server"  
ErrorMessage="You must select a room."  
ControlToValidate="roomList" Display="None" />  
</td>  
</tr>  
<tr>  
<td valign="top">  
Attendees:</td>  
<td valign="top">  
<asp:ListBox ID="attendeeList" Runat="server"  
Width="160px"  
SelectionMode="Multiple" Rows="6">  
<asp:ListItem Value="1">Bill Gates</asp:ListItem>  
<asp:ListItem Value="2">Monica Lewinsky</asp:ListItem>  
<asp:ListItem Value="3">Vincent Price</asp:ListItem>  
<asp:ListItem Value="4">Vlad the Impaler</asp:ListItem>  
<asp:ListItem Value="5">Iggy Pop</asp:ListItem>  
<asp:ListItem Value="6">William  
Shakespeare</asp:ListItem>  
</asp:ListBox>
```

其中把 ListItem 对象与两个 Web 服务器控件关联起来。这些对象本身并不是 Web 服务器控件(它们仅继承了 System.Object)，因此不需要使用 Runat="server"。在处理页面时，使用<asp:ListItem>项创建 ListItem 对象，再把它们添加到父列表控件的 Items 集合中，这便于初始化列表，而无需编写代码(必

须创建一个 ListItemCollection 对象，添加 ListItem 对象，再把集合传送给列表控件)。当然，也可以编程完成这些工作：

```
<asp:RequiredFieldValidator  
ID="validateAttendees" Runat="server"  
ErrorMessage="You must have at least one attendee."  
ControlToValidate="attendeeList" Display="None" />  
</td>  
</tr>  
<tr>  
<td align="center" colspan="3">  
<asp:Button ID="submitButton" Runat="server"  
Width="100%"  
Text="Submit meeting room request" />  
</td>  
</tr>  
<tr>  
<td align="center" colspan="3">  
<asp:ValidationSummary ID="validationSummary"  
Runat="server"  
HeaderText="Before submitting your request:" />  
</td>  
</tr>  
</table>  
</div>  
<div>  
<p>  
Results:  
<asp:Label Runat="server" ID="resultLabel" Text="None."  
/>  
</p>  
</div>  
</form>  
</body>  
</html>
```

在设计视图上，创建的窗体如图 37-7 所示。这是一个功能全面的 UI，它可以在服务器请求之间维护它自己的状态，并验证用户输入。上述代码非常简洁，实际上，我们几乎不需要做什么工作，至少对于这个示例来说是这样，而只需把按钮单击事件与提交按钮关联起来。



图 37-7

实际并非如此。因为我们没有验证日历控件。这很简单，只需给它设置一个初始值。在页面的 Page\_Load() 事件处理程序中，可以设置该值：

```
private void Page_Load(object sender, System.EventArgs
```

```
e)
{
if (!this.IsPostBack)
{
calendar.SelectedDate = System.DateTime.Now;
}
}
```

我们把今天的日期作为初始值。注意首先检查页面的 IsPostBack 属性，看看是否会把调用 Page\_Load() 作为回送操作结果。如果正在进行回送，这个属性就应是 true，不必改变选中的日期(毕竟，我们不希望丢失用户的选择)。

要添加按钮单击处理程序，只需双击该按钮，并添加如下代码：

```
private void submitButton_Click(object sender,
System.EventArgs e)
{
if (this.IsValid)
{
resultLabel.Text = roomList.SelectedItem.Text +
" has been booked on " +
calendar.SelectedDate.ToString() +
" by " + nameBox.Text + " for " +
eventBox.Text + " event. ";
foreach (ListItem attendee in attendeeList.Items)
{
if (attendee.Selected)
{
resultLabel.Text += attendee.Text + ", ";
}
}
resultLabel.Text += " and " + nameBox.Text +
" will be attending.";
}
}
```

把 resultLabel 控件的 Text 属性设置为结果字符串，显示在主表格的下方。在 IE 中，这个提交结果应如图 37-8 所示，除非有错误，否则在这种情况下就应激活 ValidationSummary，如图 37-9 所示。



图 37-8



图 37-9

### 37.3 ADO.NET 和数据绑定

上一节创建的 Web 窗体应用程序有很好的功能，但只包含静态数据。另外，会议登记过程不包含永久的事件数据。为了解决这两个问题，可以使用 ADO.NET 访问存储在数据库中的数据，这样就可以存储和检索会议数据，以及会议室和参加者的列表。

数据绑定可以使检索数据的过程变得非常简单。像列表框(和一些更专业的控件)这样的控件可以使用这种技巧。它们可以绑定到执行 IEnumerable、 ICollection、 或 IListSource 接口的任何对象上，包括数据源 Web 服务器控件。

本节首先更新会议登记应用程序，使之支持数据；再使用一些其他支持数据的 Web 控件，介绍其他一些可以通过数据绑定完成的任务。

### 37.3.1 更新会议登记应用程序

为了区别于上一个示例，在 C:\ProCSharp\Chapter37\ 目录下创建一个新的 Web 站点 PCSWebApp3，复制前面创建的 PCSWebApp2 应用程序中的代码。在开始编写新代码前，先看看要访问的数据库。

#### 1. 数据库

在本例中，使用一个 Microsoft SQL Server Express 数据库 MeetingRoomBooker.mdf，该数据库可以从本书的代码中下载。企业级的应用程序应使用 SQL Server 数据库，但涉及到的技术是相同的，使用 SQL Server Express 会使测试更简单一些，代码也相同。

提示：

如果添加这个数据库的另一个版本，就需要在 Solution Explorer 的 App\_Data 文件夹中添加一个新数据库。为此，右击 App\_Data 文件夹，选择 Add New Item，再选择数据库，命名为 MeetingRoomBooker，然后单击 Add。这也会在 Server Explorer 窗口中配置一个数据连接，以供使用。之后就可以按照下一节的要求添加表，提供自己的数据。另外，要通过编写代码来使用下载的数据库，只需把它复制到 Web 站点的 App\_Data 文件夹下。

该数据库包含 3 个表：

Attendees 包含事件参加者的一个列表。

Rooms 包含会议室的一个列表。

Events 包含登记事件的一个列表。

#### (1) 参加者

Attendees 表包含表 37-5 所示的列。

该数据库包含 20 个参加者的信息，他们都有电子邮件地址。在一个比较高级的应用程序中，电子邮件会自动发送给已登记的参加者，我们把这个任务留给您来完成，其中使用的技巧可以在本书的其他地方找到。

表 37-5

列	类 型	说 明
ID	Identity , 主键	参加者的身份标识号码
Name	varchar , 必选 , 50 个字符	参加者的姓名
Email	varchar , 可选 , 50 个字符	参加者的电子邮件地址

## (2) 会议室

Rooms 表包含表 37-6 所示的列。

表 37-6

列	类 型	说 明
ID	Identity , 主键	房间标识号码
Room	varchar , 必选 , 50个字符	房间名

数据库中提供了 20 条记录。

## (3) 事件

Events 表包含表 37-7 所示的列。

表 37-7

列	类 型	说 明
ID	Identity , 主键	会议标识号码
Name	varchar , 必选 , 255 个字符	会议名称
Room	Int , 必选	会议室 ID.
AttendeeList	Text , 必选	参加者姓名列表
EventDate	DateTime , 必选	会议日期

下载的数据库中提供了几个会议。

## 2. 数据库的绑定

要绑定数据的两个控件是 attendeeList 和 roomList。在此之前，需要添加 Web 服务器控件 SqlDataSource，以映射要在 MeetingRoomBooker.mdf 数据库中访问的表。最快捷的方式是把它们从工具箱拖放到 Web 窗体 Default.aspx 上，通过配置向导配置它们。图 37-10 显示了如何为 SqlDataSource 控件 MRBAttendeeData 访问这个向导。

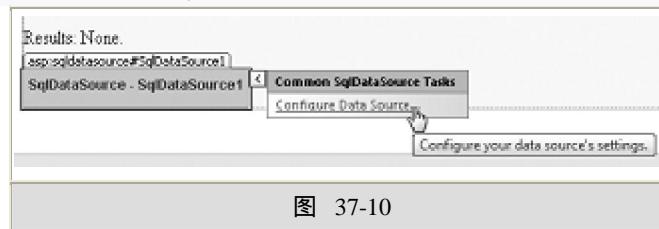


图 37-10

在数据源配置向导的第一个页面上，需要选择前面创建的数据库连接。接着，选择把连接字符串保存为 MRBConnectionString，然后从数据库的 Attendees 表中选择\*(所有字段)。

之后，把 SqlDataSource 控件的 ID 改为 MRBAttendeeData。还需要添加并配置另外两个 SqlDataSource 控件，以便用 MRBRoomData 和 MRBEVENTData 的 ID 值从 Rooms 和 Events 表中获得数据。这两个控件可以使用前面保存的 MRBConnectionString 进行连接。

添加了这些数据源之后，它们在窗体代码中的语法是非常简单的：

```
<asp:SqlDataSource ID="MRBAttendeeData" runat="server" ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>" SelectCommand="SELECT * FROM [Attendees]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBRoomData" runat="server" ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>" SelectCommand="SELECT * FROM [Rooms]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBEVENTData" runat="server" ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>" SelectCommand="SELECT * FROM [Events]"></asp:SqlDataSource>
```

连接字符串的定义在 web.config 文件中，本章后面将详细探讨这个文件。

接着，设置 roomList 和 attendeeList 控件的数据绑定属性。对于 roomList，需要的设置如下：

DataSourceID-MRBRoomData

DataTextField-Room

DataValueField-ID

同样，对于 attendeeList，需要的设置如下：

DataSourceID-MRBAttendeeData

DataTextField-Name

DataValueField-ID

也可以从代码中删除这些控件已有的硬编码列表项。

现在运行这个应用程序，从数据绑定控件中得到所有可用的参加者和会议室数据。稍后使用 MRBEVENTData 控件。

### 3. 定制日历控件

在把会议添加到数据库中之前 ,先修改一下日历的显示。最好用另一种颜色显示登记之前的日期 ,以防该日期被选中。这要求修改在日历中设置日期的方式 ,以及日期单元格的显示方式。

首先是日期选择。有 3 个地方需要查看会议登记的日期 ,并修改相应选择 :一是在 Page\_Load() 中设置初始日期时 ;二是在用户试图从日历中选择日期时 ;三是登记一个会议 ,并设置一个新的日期 ,以防用户在选择新日期前 ,在同一天连续登记两个会议。这些都是很常见的情况 ,也可以创建一个私有方法来执行这个计算。这个方法应接受一个试用日期作为参数 ,并返回要使用的日期 ,该日期可以与试用日期相同 ,也可以是试用日期之后的某个日期。

在添加这个方法之前 ,需要让代码访问 Events 表中的数据。为此可以使用 MRBEventData 控件 ,因为这个控件可以填充 DataView。所以 ,添加下面的私有成员和属性 :

```
private DataView eventData;

private DataView EventData
{
get
{
if (eventData == null)
{
eventData =
MRBEventData.Select(new DataSourceSelectArguments()) as
DataView;
}
return eventData;
}
set
{
eventData = value;
}
}
```

EventData 属性用需要的数据填充 eventData 成员 ,其结果缓存起来 ,供以后使用。这里使用 SqlDataSource.Select()方法获得 DataView。

把这个 GetFreeDate()方法添加到后台代码文件中 :

```
private System.DateTime GetFreeDate(System.DateTime
trialDate)
{
if (EventData.Count > 0)
{
System.DateTime testDate;
bool trialDateOK = false;
while (!trialDateOK)
{
trialDateOK = true;
```

```
foreach (DataRowView testRow in EventData)
{
    testDate = (System.DateTime)testRow[ "EventDate" ];
    if (testDate.Date == trialDate.Date)
    {
        trialDateOK = false;
        trialDate = trialDate.AddDays(1);
    }
}
}
}
}
return trialDate;
}
```

这段简单的代码使用 EventData DataView 提取会议数据。首先看看一般情况 :没有登记任何会议 ,此时返回该试用日期 ,以确认该日期 ,接着对 Event 表中的日期进行迭代 ,把该日期与试用日期比较。如果找到一个匹配 ,就给试用日期加一天 ,执行另一次搜索。

从 DataTable 中提取数据是相当简单的 :

```
testDate = (System.DateTime)testRow[ "EventDate" ];
```

把列数据转换为 System.DateTime , 这样会更精确。

使用 getFreeDate()的第一个地方是在 Page\_Load()后面。这表示只需对设置 SelectedDate 属性的代码稍加修改 :

```
if (!this.IsPostBack)
{
    System.DateTime trialDate = System.DateTime.Now;
    calendar.SelectedDate = getFreeDate(trialDate);
}
```

接着需要响应日历上的日期选择。为此 , 需要先为日历的 SelectionChanged 事件添加一个事件处理程序 , 强制检查现有会议的日期。双击设计器中的日历 , 添加如下代码 :

```
private void calendar_SelectionChanged(object
sender, System.EventArgs e)
{
    System.DateTime trialDate = calendar.SelectedDate;
    calendar.SelectedDate = getFreeDate(trialDate);
}
```

这段代码与 Page\_Load()相同。

执行这种检查的第三个地方是响应登记按钮的单击。后面会解释它 , 因为后面进行了许多改变。

接着把日历的日期单元格变为另一种颜色，以表示现存的会议。为此，需要给日期对象的 DayRender 事件添加一个事件处理程序。每次显示一个日期时，都会触发这个事件，并允许通过在处理程序中接收到的 DayRenderEventArgs 参数的 Cell 和 Date 属性，访问要显示的单元格对象和这个单元格的日期。我们需要比较要显示的单元格中的日期和 eventTable 对象中的日期，如果匹配，就可以使用 Cell.BackColor 属性为单元格着色：

```
void calendar_DayRender(object sender,
DayRenderEventArgs e)
{
if (EventData.Count > 0)
{
System.DateTime testDate;
foreach (DataRowView testRow in EventData)
{
testDate = (System.DateTime)testRow["EventDate"];
if (testDate.Date == e.Day.Date)
{
e.Cell.BackColor = System.Drawing.Color.Red;
}
}
}
}
```

这里使用红色，得到屏幕图 37-11。6 月的 12、15、22 日都有会议，所以用户选择了 24 日。



图 37-11

添加了日期选择逻辑后，就不可能选择显示为红色的一天，如果要选择这样的日期，就会选择该日期后面的某一天。例如，在图 37-6 的日历中单击 6 月 15 日，就会选择 16 日。

#### 4. 给数据库添加会议数据

submitButton\_Click() 事件处理程序目前从会议特性中组合了一个字符串，并在 resultLabel 控件中显示它。要给数据库添加一个会议，需要把创建出来的字符串重新格式化到一个 SQL INSERT 查询中，并执行它。

注意：

在开发环境中，不必过多地考虑安全性。通过 Web 站点解决方案添加一个 SQL Server 2005 Express 数据库，把 SqlDataSource 控件配置为使用该数据库，会自动建立一个连接字符串，它可用于写入数

据库。在比较高级的场合下，可以使用其他账户访问资源，例如域账户用于访问网络其他地方的 SQL Server 实例。ASP.NET 中有这个功能(通过模拟、COM+服务或其他方式获得)，但超出了本书的范围。在大多数情况下，只要正确配置连接字符串，能正常完成任务即可。

下面的许多代码都是很熟悉的：

```
void submitButton_Click(object sender, EventArgs e)
{
if (this.IsValid)
{
System.Text.StringBuilder sb = new System.Text.StringBuilder();
foreach (ListItem attendee in attendeeList.Items)
{
if (attendee.Selected)
{
sb.AppendFormat("{0} ({1}), ", attendee.Text, attendee.Value);
}
}
sb.AppendFormat(" and {0}", nameBox.Text);
string attendees = sb.ToString();
try
{
System.Data.SqlClient.SqlConnection conn =
new System.Data.SqlClient.SqlConnection(
ConfigurationManager.ConnectionStrings["MRBConnectionString"]
.ConnectionString);
System.Data.SqlClient.SqlCommand insertCommand =
new System.Data.SqlClient.SqlCommand("INSERT INTO [Events] "
+ "(Name, Room, AttendeeList, EventDate) VALUES (@Name, "
+ "@Room, @AttendeeList, @EventDate)", conn);
insertCommand.Parameters.Add(
"Name", SqlDbType.VarChar, 255).Value = eventBox.Text;
insertCommand.Parameters.Add(
"Room", SqlDbType.Int, 4).Value = roomList.SelectedValue;
insertCommand.Parameters.Add(
"AttendeeList", SqlDbType.Text, 16).Value = attendees;
insertCommand.Parameters.Add(
"EventDate", SqlDbType.DateTime, 8).Value =
calendar.SelectedDate;
}
```

这里最有趣的是如何使用下面的语法访问前面创建的连接字符串：

```
 ConfigurationManager.ConnectionStrings["MRBConnectionString"].ConnectionString
```

ConfigurationManager 类可以访问所有配置信息，它们都存储在 Web 应用程序的 Web.Config 配置文件中。该文件详见本章后面的内容。

创建了 SQL 命令后，就可以使用它插入新事件：

```
conn.Open();
int queryResult = insertCommand.ExecuteNonQuery();
conn.Close();
```

ExecuteNonQuery()返回一个整数，表示查询会影响表中的多少行。如果它等于 1，插入就是成功的。此时把一个成功的信息放在 resultLabel 中，清除 EventData，因为它现在已过期了。把日历选择改为一个新的、没有会议的日期。因为 GetFreeDate()使用 EventData，而 EventData 属性在没有数据时会自动刷新它自己，所以会刷新存储的会议数据：

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
}
```

如果 ExecuteNonQuery()返回的数字不是 1，就会有问题。在本例中不必担心，只需抛出一个异常，该异常会在一般的 catch 块中捕获，该 catch 块位于数据库访问代码中。它会在 resultLabel 中显示一个故障通知：

```
else
{
    throw new System.Data.DataException("Unknown data
error.");
}
}
catch
{
    resultLabel.Text = "Event not added due to DB access "
+ "problem.";
}
}
}
```

支持数据的会议登记应用程序就完成了。

### 37.3.2 数据绑定的更多内容

如前所述，Web 服务器控件有几个处理数据显示的控件：GridView、DataList、DetailsView、FormView 和 Repeater。在把数据输出到网页上时，这些都是非常有用的，因为它们会自动执行许多任务，否则将需要编写许多代码。

首先，介绍如何使用这些控件，在 PCSWebApp3 的底部显示一个会议列表。

把一个 GridView 控件从工具箱拖放到 Default.aspx 的底部 ,选择前面添加的 MRBEEventData 数据源 ,如图 37-12 所示。

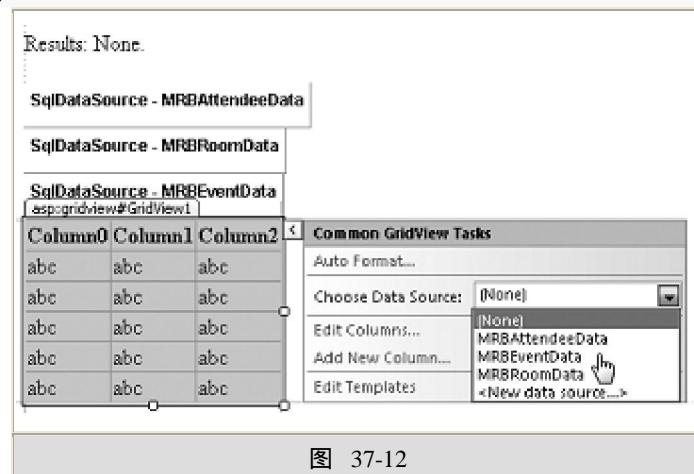


图 37-12

接着单击 Refresh Schema , 这就是在窗体底部显示会议列表所需做的工作 , 现在查看 Web 站点 , 就会看到会议 , 如图 37-13 所示。

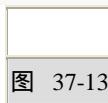


图 37-13

还可以对 submitButton\_Click() 做进一步的修改 , 确保在添加新记录时更新数据 :

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
}
```

所有的数据绑定控件都支持这个方法 , 如果调用顶层的(this) DataBind() 方法 , 窗体就会调用该方法。

注意 , 在图 37-13 中 , EventDate 字段的日期/时间显示有点麻烦。由于我们只查看日期 , 因此时间总是 12:00:00 , 其实这个信息不需要显示。下一节将学习如何以更友好的方式在 DataList 控件中显示这个日期信息。 DataGrid 控件包含许多属性 , 它们可以用于格式化显示的数据 , 但这部分内容由您自学。

### 1. 使用模板显示数据

许多数据显示控件可以使用模板来格式化要显示的数据。模板在 ASP.NET 中是 HTML 的参数化部分 , 用作某些控件的输出元素。它们可以定制如何将数据输出到浏览器上 , 不需要做太多的工作就可以得到专业级的显示结果。

有几个模板可用于定制列表的各个方面。对于 Repeater 和 DataList 来说，一个重要的模板是 <ItemTemplate>，它可以用于显示 Repeater、DataList 和 ListView 控件的每个数据项。在控件声明中声明这个模板(和其他模板)，例如：

```
<asp:DataList Runat="server" ... >  
<ItemTemplate>  
...  
</ItemTemplate>  
</asp:DataList>
```

在模板声明中，一般是输出 HTML 的部分内容，参数是绑定到控件的数据。在输出这些参数时，应使用一种特殊的语法：

```
<%# expression %>
```

expression 占位符是把参数绑定到页面或控件属性上的表达式，但它常常是由 Eval()或 Bind()表达式组成。通过指定表中的列，这个函数可以从绑定到控件的表中输出数据，Eval()使用下面的语法：

```
<%# Eval("ColumnName") %>
```

还有第二个可选参数，可以格式化返回的数据，它的语法与其他地方使用的字符串格式化表达式相同。该参数可以把日期字符串格式化为可读性更高的格式，这正是前面示例所缺乏的。

Bind()表达式与 Eval()相同，但可以把数据插入服务器控件的属性，例如：

```
<asp:Label RunAt="server" ID="ColumnDisplay" Text='<%#  
Bind("ColumnName") %>' />
```

注意，双引号可在 Bind()参数中使用，所以应使用单引号把属性值括起来。

表 37-8 列出了可用的模板以及它们的用法。

表 37-8

模 板	应 用 于	说 明
<ItemTemplate>	DataList, Repeater	列表项使用的模板
<HeaderTemplate>	DataList, DetailsView, FormView, Repeater	列表项前输出内容使用的模板
<FooterTemplate>	DataList, DetailsView, FormView, Repeater	列表项后输出内容使用的模板
<LayoutTemplate>	ListView	用于指定输出周围的项

(续表)

模 板	应 用 于	说 明
<SeparatorTemplate>	DataList, Repeater	列表中项之间使用的模板
<ItemSeparatorTemplate>	ListView	列表中项之间使用的模板
<AlternatingItemTemplat e>	DataList, ListView	其他项使用的模板，有助于查看

<SelectedItemTemplate>	DataList, ListView	列表中所选项使用的模板
<EditItemTemplate>	DataList, FormView, ListView	用于列表中正在编辑的项的模 板
<InsertItemTemplate>	FormView, ListView	用于列表中正在插入的项的模 板
<EmptyDataTemplate>	GridView, DetailsView, FormView	用于显示空项 , 例如 GridView 中没有记录时使用它
<PagerTemplate>	GridView, DetailsView, FormView	用于格式化分页
<GroupTemplate>	ListView	用于指定输出周围的项的组合
<GroupSeparatorTemplat e>	ListView	列表中项之间使用的模板
<EmptyItemTemplate>	ListView	使用分组的项时 , 该模板用于为 组中的空项提供输出。这个模板在组 中没有足够的项时使用

了解模板最简单的方式是举一个例子。

## 2. 使用模板

在 PCSWebApp3 的 Default.aspx 页面顶部扩展表格 , 使之包含一个 ListView , 显示存储在数据库中的每个会议。使这些会议成为可选择的 , 这样单击每个会议的名称 , 就在 FormView 控件中显示它们的信息。

首先需要为数据绑定控件创建新的数据源。每个数据绑定控件最好有自己的数据源。

ListView 控件需要 SqlDataSource 控件 MRBEVENTDATA2 ,MRBEVENTDATA2 与 MRBEVENTDATA 相同 , 但它只需返回 Name 和 ID 数据。需要的代码如下 :

```
<asp:SqlDataSource ID="MRBEVENTDATA2" Runat="server"  
SelectCommand="SELECT [ID], [Name] FROM [Events]"  
ConnectionString="<%$  
ConnectionStrings:MRBConnectionString %>">  
</asp:SqlDataSource>
```

FormView 控件的数据源 MRBEVENTDETAILDATA 比较复杂 , 但可以通过数据源配置向导方便地建立它。这个数据源使用 ListView 控件的选中项 EventList , 获取选中的项的数据。这可以使用 SQL 查询中的一个参数实现 , 如下所示 :

```
<asp:SqlDataSource ID="MRBEVENTDETAILDATA"  
Runat="server"  
SelectCommand="SELECT dbo.Events.Name, dbo.Rooms.Room,  
dbo.Events.AttendeeList,  
dbo.Events.EventDate FROM dbo.Events INNER JOIN dbo.Rooms  
ON dbo.Events.ID = dbo.Rooms.ID WHERE dbo.Events.ID = @ID"  
ConnectionString="<%$  
ConnectionStrings:MRBConnectionString %>">
```

```
<SelectParameters>
<asp:ControlParameter Name="ID" DefaultValue="-1"
ControlID="EventList"
PropertyName="SelectedValue" />
</SelectParameters>
</asp:SqlDataSource>
```

其中 ID 参数会得到在 Select 查询的@ID 处插入的值。ControlParameter 项从 EventList 的 SelectedValue 属性中提取这个值，如果没有选中的项，就使用-1。初看起来，这个语法有点古怪，但它非常灵活。一旦使用向导生成了这些对象，就不需要自己建立它们了。

下面需要添加 DataList 和 FormView 控件。修改 PCSWebApp3 项目的 Default.aspx 中的代码：

```
<tr>
<td align="center" colSpan=3>
<asp:ValidationSummary ID=validationSummary
Runat="server"
HeaderText="Before submitting your request:"/>
</td>
</tr>
<tr>
<td align="left" colspan="3" style="width: 40%; ">
<table cellspacing="4" style="width: 100%; ">
<tr>
<td colspan="2" style="text-align: center;">
<h2>Event details</h2>
</td>
</tr>
<tr>
<td style="width: 40%; background-color: #ccffcc;" valign="top">
<asp:ListView ID="EventList" runat="server"
DataSourceID="MRBEVENTData2" DataKeyNames="ID"
OnSelectedIndexChanged="EventList_SelectedIndexChanged" >
<LayoutTemplate>
<ul>
<asp:PlaceHolder ID="itemPlaceholder"
runat="server" />
</ul>
</LayoutTemplate>
<ItemTemplate>
<li>
<asp:LinkButton Text=' <%# Bind("Name") %>' runat="server" ID="NameLink" CommandName="Select" CommandArgument=' <%# Bind("ID") %>'>

```

```
CausesValidation="false" / >
</li>
</ItemTemplate>
<SelectedItemTemplate>
<li>
<b><%# Eval("Name") %></b>
</li>
</SelectedItemTemplate>
</asp:ListView>
</td>
<td valign="top">
<asp:FormView ID="FormView1" Runat="server"
DataSourceID="MRBEventDetailData">
<ItemTemplate>
<h3><%# Eval("Name") %></h3>
<b>Date:</b>
<%# Eval("EventDate", "{0:D}") %>
<br />
<b>Room:</b>
<%# Eval("Room") %>
<br />
<b>Attendees:</b>
<%# Eval("AttendeeList") %>
</ItemTemplate>
</asp:FormView>
</td>
</tr>
</table>
</td>
</tr>
</table>
```

我们添加了一个新的表行，其中包含一个表，该表中的一列是一个 ListView 控件，另一列是一个 FormView 控件。

ListView 使用<LayoutTemplate>输出一个项目列表，使用<ItemTemplate>和<Selected ItemTemplate>显示会议信息。在<LayoutTemplate>中，用 ID 属性为"itemPlaceholder"的 PlaceHolder 控件给数据项指定一个容器元素。为了提供选择，对会议名称链接执行 Select 命令，该会议名称链接显示在<ItemTemplate>中，这样就可以自动修改选择。我们还使用了 OnSelectedIndexChanged 事件，当 Select 命令修改选择时触发这个事件，以更新列表，用不同的风格显示选中的项。事件处理程序如下所示：

```
void EventList_SelectedIndexChanged(object sender,
EventArgs e)
{
```

```
EventList.DataBind();
}
```

还需要确保新会议添加到列表中：

```
if (queryResult == 1)
{
    resultLabel.Text = "Event Added.";
    EventData = null;
    calendar.SelectedDate =
        GetFreeDate(calendar.SelectedDate.AddDays(1));
    GridView1.DataBind();
    EventList.DataBind();
}
```

现在会议的详细信息就显示在表中，如图 37-14 所示。

使用模板和数据绑定控件可以完成许多任务，需要用一本书的篇幅来介绍。但是，这里介绍的内容已经足够您开始试用它们了。

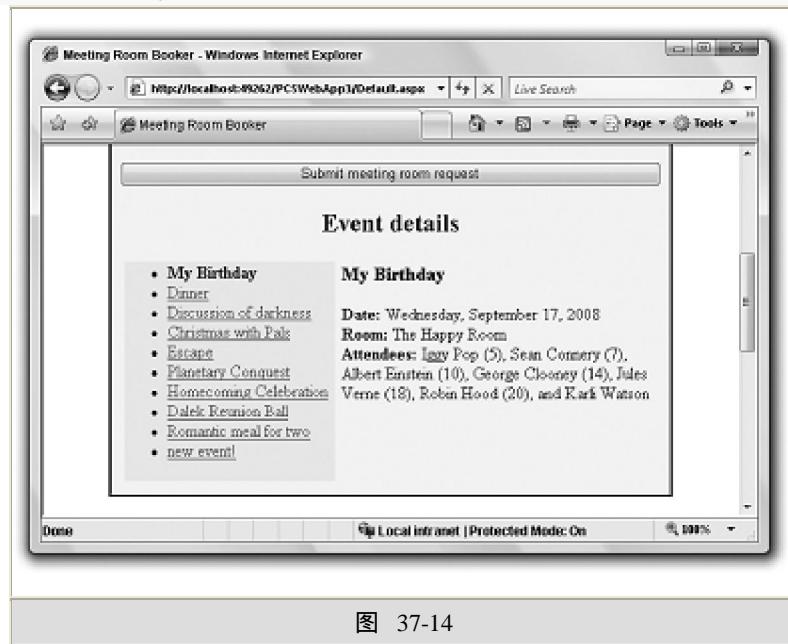


图 37-14

#### 37.4 应用程序配置

本章一直暗示一个事情，那就是应用程序都包含网页和配置设置。这是一个重要的概念，必须掌握，特别是为多个同步用户配置网站时，这个概念就更加重要了。

首先介绍一些术语和应用程序的生命期。应用程序定义为项目中的所有文件，由 web.config 文件配置。在第一次创建应用程序时，将创建一个 Application 对象，即收到第一个 HTTP 请求时创建该对象。此时还将触发 Application\_Start 事件，创建一个 HttpApplication 实例池。每个输入的请求都会接收到这样一个实例，执行请求的处理过程。注意，HttpApplication 对象不需要处理同步访问，与全局 Application 对象不同。所有的 HttpApplication 实例完成任务后，就触发 Application\_End 事件，应用程序终止执行，消除 Application 对象。

上面涉及到的事件处理程序(以及本章讨论的所有其他事件处理程序)可以在 Global.asax 文件中定义，该文件可以添加到任意 Web 站点项目中。生成的文件包含空格，用户可以在这些空格中填写信息，例如：

```
protected void Application_Start(Object sender,  
EventArgs e)  
{  
}
```

在单个用户使用 Web 应用程序时，会启动一个会话。与应用程序类似，会话将创建一个用户特定的 Session 对象，并触发 Session\_Start 事件。在一个会话中，每个请求都将触发 Application\_BeginRequest 和 Application\_EndRequest 事件。在一个会话中可以多次触发这两个事件，因为这些事件会访问应用程序中的不同资源。会话可以手动终止，如果没有接收到更多的请求，会话也会因超时而停止。会话终止将触发 Session\_End 事件，消除 Session 对象。

了解这个过程，可以执行几个操作，以简化应用程序。例如，如果应用程序的所有实例都使用一个资源密集型对象，就可以考虑在应用程序一级上实例化它。这将提高性能，减少多个用户使用的内存，因为在大多数请求中，不需要进行这个实例化。

可以使用的另一个技巧是存储会话级别的信息，以备单个用户在跨请求时使用。这些信息包括用户第一次连接(在 Session\_Start 事件处理程序中)时从数据库中提取的用户特定信息，在会话终止(通过超时或用户请求)后，才能使用这些信息。

这些技巧超出了本书的范围，读者可参阅《ASP.NET 2.0 高级编程》(清华大学出版社引进并已出版)，它们有助于理解该过程。

最后，看看 Web.Config 文件。Web 站点通常在其根目录下有这个文件(但不会在默认情况下创建它)，在其子目录下也有该文件，用于配置与该子目录相关的设置(如安全性)。本章开发的 Web 站点 PCSWebApp3 在添加已存储的数据库连接字符串时，会接收一个自动生成的 Web.Config 文件，如下所示：

```
<connectionStrings>  
<add name="MRBConnectionString" connectionString="Data  
Source=.\\SQLEXPRESS;  
AttachDbFilename=| DataDirectory |\\MeetingRoomBooker.mdf;  
Integrated Security=True;User Instance=True"  
providerName="System.Data.SqlClient" />  
</connectionStrings>
```

如果在调试模式下运行项目，就会在 Web.Config 文件中看到一些额外的设置。

可以手工编辑 Web.Config 文件，还可以使用一个工具配置 Web 站点(及其底层的配置文件)。这个工具在 Visual Studio Website 菜单的 ASP.NET Configuration 上。该工具如图 37-15 所示。



图 37-15

从文本中可以看出，这个工具可以配置许多设置，包括安全性。下一章将详细介绍这个工具。

### 37.5 小结

本章概述如何利用 ASP.NET 创建 Web 应用程序，介绍如何使用 C# 和 Web 服务器控件提供一个优秀的开发环境。我们还开发了一个会议登记应用程序，阐述了许多技术，例如各种服务器控件、ADO.NET 的数据绑定。

主要内容如下：

ASP.NET 概述，以及它与 .NET 开发环境的配合使用

ASP.NET 的基本语法，状态的管理，把 C# 代码集成到 ASP.NET 页面中

使用 Visual Studio 创建 ASP.NET Web 应用程序，存储和测试 Web 站点的选项

ASP.NET 开发人员可以使用的 Web 控件，它们如何传送动态或数据驱动的内容

使用事件处理器检测并执行用户与控件的交互操作，通过页面和显示事件定制控件

把数据绑定到 Web 控件上，使用模板和数据绑定表达式格式化要显示的数据

综合运用这些技巧，建立会议室登记应用程序

掌握了这些知识，就可以建立功能强大的 Web 应用程序了。但这只涉及 ASP.NET 的皮毛。在开始 Web 开发之前，应先了解更多的信息。第 38 章将扩展 ASP.NET 知识，学习更重要的 Web 主题，包括 master 页面、skinning 和个性化。

## 第 38 章 ASP.NET 开发

在进行 Web 开发时通常会出现这样的情况：即可用的工具的功能虽然强大，但不符合具体项目的需求，可能是给定控件的工作方式并不像所期望的那样，也可能是一部分代码本来的目的是能够在多个页面上重用，但是许多开发人员实现起来却相当复杂。在这些情况下，定制控件的建立就尤为迫切。简言之，定制控件可以把多个现有的控件包装在一起，这些现有控件还可能有指定布局的额外属性；定制控件也可以与现有的控件完全不同。使用定制控件与使用 ASP.NET 中的控件一样简单，能使 Web 站点的编码非常容易。

本章的第一部分将介绍控件开发人员可用的选项，并编写一个简单的用户控件，我们还将介绍构建高级控件的基础知识，但不详细探讨它们，这些主题需要一本书的篇幅来讨论。

接着介绍 Master 页面，这是 ASP.NET 2.0 的一个新技术，可以为 Web 站点提供模板。使用 Master 页面可以在 Web 站点上通过大量的重用代码，实现复杂的 Web 页面布局。本章还将说明如何使用 Web 导航服务器控件和 Master 页面，提供 Web 站点上一致的导航布局。

站点导航可以把用户分为不同的组，只允许某些用户(注册到站点上的用户，或站点管理员)访问某些部分。本章还将介绍 Web 站点的安全性和登录，通过 Web 登录服务器控件很容易实现该功能。

之后介绍一些高级样式设置技巧，即提供和选择 Web 站点的主题，主题把 Web 页面的显示与其功能分隔开。我们可以为站点提供 CSS 样式表，给 Web 服务器控件提供不同的样式。

最后使用 Web Part 定位和定制页面上的控件，让用户动态地个性化 Web 页面。

本章的主要内容如下：

用户控件和定制控件

Master 页面

站点导航

安全性

主题

Web Part

本章将开发一个大型示例应用程序，它包含本章和上一章介绍的所有技术。这个应用程序 PCSDemoSite 在本章的下载代码中。要包含所有的代码，会使本章非常长，但在学习其中的技术之前不需要运行它。相关的代码段会在需要时列出，其他代码(大多数是前面介绍的内容或简单代码)请读者自学。

### 38.1 用户控件和定制控件

在过去，实现定制控件是非常复杂的，尤其在大型系统中，由于使用定制控件需要复杂的注册过程，因此定制控件的实现就更为复杂。即使在简单的系统上，创建定制控件所需进行的编码也是一个相当复杂的过程。老版本 Web 语言的脚本编码功能也不能对手工编写的对象模型提供较好的访问，因此各个方面的性能都比较差。

.NET Framework 使用简单的编程技术，为定制控件的创建提供了一个理想的设置。ASP.NET 服务器控件的各个方面都可以随意定制，包括模板制作、客户端脚本编码等功能。但是，也不必为所有这些功能编写代码；控件越简单，创建就越容易。

另外，.NET 系统中固有的程序集动态查询功能使 Web 应用程序在新 Web 服务器上的安装如同复制包含代码的目录结构一样简单。要使用自己创建的控件，只需复制包含这些控件的程序集和其他代码即可。甚至可以把频繁使用的控件放在 Web 服务器上一个位于全局程序集缓存器(GAC)的程序集中，这样服务器上所有的 Web 应用程序就可以访问它们了。

本章将介绍两类不同的控件：

用户控件--即把现有的 ASP.NET 页转化为控件

定制控件--即组合几个控件的功能、扩展现有的控件以及从头创建新的控件

我们将创建一个简单的控件，显示一副扑克牌(黑桃、方块、红桃和梅花)，以便轻松地把它嵌入到其他 ASP.NET 页面中，以此来阐明用户控件的用法。对于定制控件，不打算详细介绍，只探讨基本规则和查找定制控件的地址。

### 38.1.1 用户控件

用户控件是用 ASP.NET 代码创建的控件，就像在标准的 ASP.NET Web 页面中创建控件一样，不同之处在于一旦创建了用户控件，就可以轻松地在多个 ASP.NET 页面中重用它们。

例如，假定已经创建了一个显示数据库中信息的页面，信息也许是关于订单的，就不必创建一个固定的页面去显示信息，而可以把相关的代码放到用户控件中，然后把该控件插入到任意多个不同的 Web 页面中。

此外，可以给用户控件定义属性和方法，例如，可以指定 Web 页面上显示数据库表时的背景色属性，或者指定一个方法，重新进行数据库查询，以检查数据库中的变化。

下面创建一个简单的用户控件，与其他章节一样，本章的示例项目也可以从 Wrox 网站 [www.wrox.com](http://www.wrox.com) 上下载。

#### 一个简单的用户控件

在 Visual Studio 中，在 C:\ProCSharp\Chapter38 目录下创建一个新 Web 站点 PCSUserC- WebApp1。一旦生成标准文件，就可以选择 Website | Add New Item... 菜单选项，添加名称为 PCSUserC1.ascx 的 Web 用户控件，如图 38-1 所示。

给项目添加的文件的扩展名为.ascx 和.ascx.cs ,它们的工作方式与前面的.aspx 文件非常相似。.ascx 文件包含 ASP.NET 代码 ,看起来与普通的.aspx 文件非常相似。.ascx.cs 文件是后台代码文件 ,它为用户控件定义了定制代码 ,定义的方式与在.aspx.cs 文件中定义窗体的方式一样。

与.aspx 文件相似 ,也可以在设计视图或源代码视图中查看.ascx 文件。在源代码视图中查看文件 ,可以发现一个重要的区别 :.ascx 文件没有显示 HTML 代码 ,特别是没有<form>元素 ,原因在于 :用户控件要插入到其他文件的 ASP.NET 窗体中 ,因此不需要自己的<form>标记。生成的代码如下所示。



这非常类似于在.aspx 文件中生成的<%@ Page %>指令 ,但指定了 Control ,而不是 Page ,CodeFile 属性指定了后台代码文件 ,Inherits 指定了在后台代码文件中页面继承的类名。.ascx.cs 文件中生成的代码与自动生成的.aspx.cs 文件一样 ,其中包含一个空的类定义和 Page\_Load() 事件处理程序。

本例的简单控件是一个显示图形的控件 ,显示的图形对应于扑克牌中的花色(即梅花、方块、红桃和黑桃)。这里所需的图形是 Visual Studio 附带的图形 ;它们在本章的下载代码中 ,位于 CardSuitImages 目录 ,其文件名分别是 CLUB.BMP、DIAMOND.BMP、HEART.BMP 和 SPADE.BMP。把这些图形文件复制到项目目录的新子目录 Images 中 ,以便在后面使用它们。如果不能访问这个目录 ,可以使用其他任意图形 ,因为它们对于代码的功能而言并不重要。

#### 注意 :

与 Visual Studio 的以前版本不同 ,在 Visual Studio 外部对 Web 站点结构进行的修改会自动反映到 IDE 上。只需单击 Solution Explorer 窗口中的 Refresh 按钮 ,就会看到新的 Images 目录和位图文件。

给新控件添加一些代码。在 PCSUserC1.ascx 的 HTML 视图中添加下列代码 :

```
<%@ Control Language="c#" AutoEventWireup="true"
CodeFile="PCSUserC1.ascx.cs"
Inherits="PCSUserC1" %>


|                                                                                  |                                                           |
|----------------------------------------------------------------------------------|-----------------------------------------------------------|
| <asp:image id="suitPic" imageurl="~/Images/club.bmp" runat="server"></asp:image> | <asp:label id="suitLabel" runat="server">Club</asp:label> |
|----------------------------------------------------------------------------------|-----------------------------------------------------------|


```

```
</table>
```

这段代码定义了控件的默认状态，即一个梅花图形和一个标签。图像路径前面的~表示"从 Web 站点的根目录开始"。在给控件添加功能之前，先把这个控件添加到项目的 Web 页面 WebForm1.aspx 上，测试这个默认状态。

为了在.aspx 文件中使用定制的控件，首先需要指定如何引用该控件，也就是说，如何在 HTML 中引用代表控件的标记名称。为此，在 Default.aspx 中代码的顶部使用<%Register%>指令，如下所示。

```
<%@ Register TagPrefix="PCS" TagName="UserC1"  
Src="PCSUserC1.ascx" %>
```

属性 TagPrefix 和 TagName 指定要使用的标记名称(指定的格式为<TagPrefix:TagName>)，使用属性 Src 指向包含用户控件的文件。现在，添加下面的元素，就可以使用控件了：

```
<form id="Form1" method="post" runat="server">  
<div>  
<PCS:UserC1 Runat="server" ID="myUserControl" />  
</div>  
</form>
```

这就是测试用户控件需要做的所有工作，运行项目的结果如图 38-2 所示。

(点击查看大图) 图 38-2

可以看出，这个控件在表格布局中组合了两个现有的控件，即图形控件和标签控件，因此它属于合成控件一类。

为了控制显示的花色图形，可以在元素上使用属性。用户控件元素上的属性会自动映射到用户控件的特性上，因此，只需给控件的后台代码 PCSUserC1.ascx.cs 添加一个特性。这个特性称为 Suit，让它接收合适的花色值。为了便于表示控件的状态，可以定义一个枚举，来保存 4 个花色名称。最佳方式是在 Web 站点上添加一个目录 App\_Code(App\_Code 是另一个"特殊"的目录，与 App\_Data 一样，它的功能取决于编程人员，这里是为 Web 应用程序保存其他代码文件。要添加这个目录，可以右击 Solution Explorer 中的 Web site，单击 Add ASP.NET Folder App\_Code)，然后在这个目录中添加一个.cs 文件 suit.cs，其代码如下：

```
using System;  
  
public enum suit  
{  
    club, diamond, heart, spade  
}
```

类 PCSUserC1 需要一个成员变量，以保存花色类型 currentSuit：

```
public partial class PCSUserC1 :  
System.Web.UI.UserControl  
{  
protected suit currentSuit;
```

再添加一个访问这个成员变量的属性 Suit :

```
public suit Suit  
{  
get  
{  
return currentSuit;  
}  
set  
{  
currentSuit = value;  
suitPic.ImageUrl = "~/Images/" + currentSuit.ToString()  
+ ".bmp";  
suitLabel.Text = currentSuit.ToString();  
}  
}
```

这里的 set 存取器把图形的 URL 设置为前面复制的一个文件，并把要显示的文本设置为花色名称。

下面需要给 Default.aspx 添加代码以访问这个新的属性。使用刚才添加的属性选择花色：

```
Club  
Diamond  
Heart  
Spade
```

还需要给列表的 SelectedIndexChanged 事件添加事件处理程序。双击设计视图中的单选按钮列表，就可以添加处理程序。

注意：

把列表的 Autopostback 属性设置为 true，是因为除非进行回送操作，否则将不在服务器上执行 suitList\_SelectedIndexChanged 事件处理程序，在默认状态下，这个控件也不会触发回送操作。

在 Default.aspx.cs 中，方法 suitList\_SelectedIndexChanged()需要以下代码：

```
public partial class Default
{
    protected void suitList_SelectedIndexChanged(object
sender, EventArgs e)
    {
        myUserControl.Suit = (suit)Enum.Parse(typeof(suit),
suitList.SelectedItem.Value);
    }
}
```

我们知道，元素<ListItem>上的 value 属性代表前面定义的枚举 suit 的有效值，因此简单地把这些值解析为枚举类型，并把它们用作用户控件的 Suit 属性值。使用简单的数据类型转换语法，就可以把返回的对象类型转换为 suit，因为这个类型不能通过隐式转换而得到。

在运行 Web 应用程序时，可以改变花色，如图 38-3 所示。

(点击查看大图) 图 38-3

接下来，给控件添加一些方法。这是非常简单的，只需给 PCSUserC1 类添加方法就可以了：

```
public void Club()
{
    Suit = suit.club;
}
public void Diamond()
{
    Suit = suit.diamond;
}
public void Heart()
{
    Suit = suit.heart;
}
public void Spade()
{
    Suit = suit.spade;
}
```

4 个方法 Club()、Diamond()、Heart()和 Spade()分别用于改变显示在屏幕上的扑克牌的花色。

在.aspx 页面上的 4 个 ImageButton 控件上调用这些函数：

```
</asp:RadioButtonList>  
  
<asp:ImageButton Runat="server" ID="clubButton"  
ImageUrl="~/Images/CLUB.BMP" OnClick="clubButton_Click"  
/>  
  
<asp:ImageButton Runat="server" ID="diamondButton"  
ImageUrl="~/Images/DIAMOND.BMP"  
OnClick="diamondButton_Click" />  
  
<asp:ImageButton Runat="server" ID="heartButton"  
ImageUrl="~/Images/HEART.BMP"  
OnClick="heartButton_Click" />  
  
<asp:ImageButton Runat="server" ID="spadeButton"  
ImageUrl="~/Images/SPADE.BMP"  
OnClick="spadeButton_Click" />  
  
</div>  
</form>
```

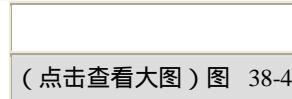
#### 事件处理程序如下：

```
protected void clubButton_Click(object sender,  
ImageClickEventArgs e)  
{  
myUserControl.Club();  
suitList.SelectedIndex = 0;  
}  
  
protected void diamondButton_Click(object sender,  
ImageClickEventArgs e)  
{  
myUserControl.Diamond();  
suitList.SelectedIndex = 1;  
}  
  
protected void heartButton_Click(object sender,  
ImageClickEventArgs e)  
{  
myUserControl.Heart();  
suitList.SelectedIndex = 2;  
}  
  
protected void spadeButton_Click(object sender,  
ImageClickEventArgs e)  
{  
myUserControl.Spade();  
suitList.SelectedIndex = 3;  
}
```

#### 注意：

这 4 个按钮可以使用一个事件处理程序，因为它们有相同的方法签名，都要通过传送给 sender 的值检测按下了哪个按钮，动态确定应调用 myUserControl 的哪个方法，动态设置哪个索引。但与需要的代码量相比，其性能差异不是很大，所以为了简单起见，还是把它们分开，放在 4 个事件处理程序中。

既然有了 4 个新按钮，就可以改变扑克牌的花色，如图 38-4 所示。



(点击查看大图) 图 38-4

完成了用户控件的创建后，使用`<%@Register%>`指令和为控件创建的两个源代码文件(PCSUserC1.ascx 和 PCSUserC1.ascx.cs)，就可以在其他 Web 页面中使用这个用户控件了。

### 38.1.2 PCSDemoSite 中的用户控件

在 PCSDemoSite 中，要把上一章的 Meeting Room Booker 应用程序转换为用户控件，以便于重用。为了查看这个控件，必须以 User1 的身份，用密码 User1!! 登录站点(本章后面将介绍系统的登录)，然后导航到 Meeting Room Booker 页面，如图 38-5 所示。

除了样式上有显著的变化之外，本章后面的主题还对该页面进行了如下大的改动：

用户名自动从用户信息中获取

在页面底部不显示额外的数据，后台代码文件中也相应地删除了 DataBind() 调用。

控件的下面没有显示得到的标签，用户查看在日历中添加的会议和会议列表，就可以获得足够的反馈，无需报告会议的添加是否成功。

包含用户控件的页面使用了 Master 页面。

要进行这些修改，代码的改动其实很简单，但这里不介绍它们。本章后面在进行登录时还会介绍这个控件。



图 38-5

### 38.1.3 定制控件

与用户控件相比，定制控件又进了一步，原因是定制控件完全包含在 C# 程序集中，不需要单独的 ASP.NET 代码。这意味着不需要在.aspx 文件中组装 UI。相反，完全可以控制输出的内容，即控件所生成的 HTML。

一般情况下，开发定制控件要比开发用户控件花费的时间更长，原因是定制控件的语法更复杂一些，通常需要编写更多的代码才能得到结果。在开发用户控件时，只需简单地把几个现有的控件组合在一起，而开发定制控件就不那么简单了。

为了使定制控件拥有最可定制化的功能，可以从 System.Web.UI.WebControls. WebControl 派生一个类，这样就创建了一个完全定制控件。此外，可以扩展现有控件的功能，创建一个派生的定制控件。最后，可以像上一节那样把几个现有的控件组合在一起，但要使用更有逻辑的结构，创建一个合成的定制控件。

以上述 3 种方法创建的定制控件都可以按同一种方式在 ASP.NET 页面中使用。我们只需把生成的程序集放在 Web 应用程序可以找到的地方，并使用<% Register%>指令注册要使用的元素名称。这里“Web 应用程序可以找到的地方”有两种情况：可以把程序集放在 Web 应用程序的 bin 目录下，如果希望服务器上的所有 Web 应用程序都可以访问它，就可以把它放在 GAC 中。如果只在一个 Web 站点上使用用户控件，就可以把控件的.cs 文件放在站点的 App\_Code 目录下。

<% Register%>指令的语法在定制控件中有一些变化：

```
<%@ Register TagPrefix="PCS"  
Namespace="PCSCustomWebControls"
```

```
Assembly="PCSCustomWebControls"%>
```

TagPrefix 选项的使用方式与以前一样，但不使用属性 TagName 和 Src，原因是所使用的定制控件程序集包含几个定制控件，每一个定制控件都以其类名来命名，所以 TagName 就变得多余了。此外可以使用.NET Framework 的动态查找功能去查找程序集，方法是给出程序集的名称和包含控件的命名空间。

在上面的代码示例中，程序要使用 PCSCustomWebControls.dll 程序集和 PCSCustomWebControls 命名空间中的控件，以及标记前缀 PCS。如果在这个命名空间中有名为 Control1 的控件，则可以通过下面的 ASP.NET 代码去使用它：

```
<PCS:Control1 Runat="server" ID="MyControl1"/>
```

<%Register%>指令的 Assembly 属性是可选的-- 如果站点的 App\_Code 目录下有定制控件，就可以忽略该属性，Web 站点会在该目录下查找控件。但 Namespace 属性不是可选的，必须在代码文件中包含定制控件的命名空间，否则 ASP.NET 运行库就找不到它们。

定制控件是可以嵌套使用的，例如在列表控件中可以嵌套<asp:ListItem>控件，以填充该列表控件：

```
<asp:DropDownList ID="roomList" Runat="server"  
Width="160px">  
    <asp:ListItem Value="1">The Happy Room</asp:ListItem>  
    <asp:ListItem Value="2">The Angry Room</asp:ListItem>  
    <asp:ListItem Value="3">The Depressing  
        Room</asp:ListItem>  
    <asp:ListItem Value="4">The Funked Out  
        Room</asp:ListItem>  
</asp:DropDownList>
```

可以以类似的方式创建一些控件，把它们解释为其他控件的子控件。这是比较高级的技术，本章不探讨。

### 定制控件示例

下面将理论用于实践。我们将使用 C:\ProCSharp\Chapter38\目录下的一个 Web 站点 PCSCustomCWebApp1，在其 App\_Code 目录下有一个定制控件，这个控件是已有 Label 控件的彩色版本，可以给文本中的每个字母设置不同的颜色。

RainbowLabel 控件的代码在 App\_Code\Rainbow.cs 文件中，首先是下述 using 语句：

```
using System;  
using System.Data;  
using System.Configuration;  
using System.Linq;  
using System.Web;  
using System.Web.Security;
```

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Drawing;
```

除了 System.Drawing 之外，这些都是把类文件添加到 Web 站点时的默认命名空间。

System.Drawing 命名空间用于 Color 枚举。该文件的类在私有数组 Colors 中包含一个私有颜色数组，这些颜色用于文本中的字母：

```
namespace PCSCustomWebControls
{
    public class RainbowLabel : Label
    {
        private Color[] colors = new Color[] {Color.Red,
        Color.Orange,
        Color.Yellow,
        Color.GreenYellow,
        Color.Blue, Color.Indigo,
        Color.Violet};
```

注意 PCSCustomWebControls 命名空间用于包含控件。如前所述，这是一个必须的命名空间，有了它，Web 页面才能正确引用控件。

为了迭代颜色，还要在私有属性 offset 中存储一个偏移整数值：

```
private int offset
{
    get
    {
        object rawOffset = ViewState["_offset"];
        if (rawOffset != null)
        {
            return (int)rawOffset;
        }
        else
        {
            ViewState["_offset"] = 0;
            return 0;
        }
    }
    set
    {
        ViewState["_offset"] = value;
    }
}
```

```
}
```

注意这个属性只是在一个成员字段中存储了一个值，这是 ASP.NET 维护状态的方式，如上一章所述。控件在每个回送操作中都会实例化，所以要存储值，必须使用视图状态。为了易于访问，只需使用 ViewState 集合，它可以存储能串行化的任意对象。如果没有这么做，在每次回送时，偏移值都会恢复其初始值。

要修改偏移值，可以使用 Cycle()方法：

```
public void Cycle()
{
    offset = ++offset;
}
```

这个方法只是递增在视图状态中为 offset 存储的值。

最后，重写定制控件最重要的方法 Render()。在这个方法中，要输出 HTML，而且这是一个实现起来非常复杂的方法。如果考虑显示控件的所有浏览器，以及可能影响显示的所有变量，这个方法就会非常大。幸好，对于本例来说，这个方法相当简单：

```
protected override void Render(HtmlTextWriter
output)
{
    string text = Text;
    for (int pos = 0; pos < text.Length; pos++)
    {
        int rgb = colors[(pos + offset) % colors.Length].ToArgb()
& 0xFFFFFFFF;
        output.Write(string.Format(
" < font color=\"#{0:X6}\" > {1} < /font > ", rgb,
text[pos]));
    }
}
```

这个方法允许访问输出流，以显示定制控件的内容。只有两种情况不需要实现这个方法：

设计一个没有可视化表示的控件(通常称为组件)

从已有的控件中派生，且不需要改变其显示特性。

定制控件还可以有定制方法、引发定制事件、响应子控件等。对于 RainbowLabel，不需要考虑这些。

下面修改 Default.aspx，以显示控件，访问 Cycle()，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<%@ Register TagPrefix="pcs"
Namespace="PCSCustomWebControls" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Untitled Page</title>
</head>
<body>
<form id="form1" runat="server">
<div>
<pcs:RainbowLabel runat="server" ID="rainbowLabel1"
Text="Multicolored label!" />
<asp:Button Runat="server" ID="cycleButton" Text="Cycle
colors"
OnClick="cycleButton_Click" />
</div>
</form>
</body>
</html>
```

Default.aspx.cs 中需要的代码非常简单：

```
public partial class _Default : System.Web.UI.Page
{
protected void cycleButton_Click(object sender, EventArgs e)
{
    rainbowLabel1.Cycle();
}
```

现在可以查看示例文本，给文本中的字母循环使用不同的颜色，如图 38-6 所示。

可以对定制控件做许多工作，实际上，其可能性是无穷的，但必须进行实践，才能发现更多的特性。



图 38-6

## 38.2 Master 页面

Master 页面可以使 Web 站点更容易设计。把所有(至少是大多数)的页面布局都放在一个文件中，就可以为站点的各个 Web 页面考虑更重要的事情了。

Master 页面在扩展名为.master 的文件中创建，并可以像其他站点内容那样，通过 Website | Add New Item...菜单项添加。初看起来，为 Master 页面生成的代码类似于标准.aspx 页面：

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile=
    "MyMasterPage.master.cs"
Inherits="MyMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
    <asp:ContentPlaceHolder id="head" runat="server" >
        </asp:ContentPlaceHolder >
    </head>
    <body>
        <form id="form1" runat="server">
            <div>
                <asp:ContentPlaceHolder ID="ContentPlaceHolder1"
                    Runat="server">
                </asp:ContentPlaceHolder>
            </div>
        </form>
    </body>
</html>
```

其区别如下：

使用<%@ Master %>指令，而不是<%@ Page %>指令，但属性是相同的。

在页面标题中放置一个 ID 为 head 的 ContentPlaceHolder 控件。

在页面中放置一个 ID 为 ContentPlaceHolder1 的 ContentPlaceHolder 控件。

这个 ContentPlaceHolder 控件使 master 页面非常有用。在一个页面中可以有任意多个 ContentPlaceHolder 控件，它们都由使用 master 页面的.aspx 页面用于"插入"内容。可以把默认内容插入 ContentPlaceHolder 控件，但.aspx 页面会重写这个内容。

.aspx 页面要使用 master 页面，需要修改<%@ Page %>指令，如下所示：

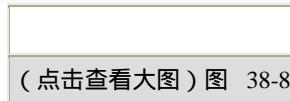
```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default"
    MasterPageFile="~/MyMasterPage.master"
    Title="Page Title" %>
```

这里添加了两个新属性：MasterPageFile 属性表示要使用的 master 页面，Title 属性设置 master 页面中<title>元素的内容。

把一个.aspx 页面添加到 Web 站点上时，就可以选择使用 master 页面，如图 38-7 所示。



如果选择使用 master 页面，就可以进入站点结构，查找要使用的 master 页面，如图 38-8 所示。



如果要使用默认的 master 页面内容，.aspx 页面就不必包含其他代码。实际上，包含 Form 控件是错误的，因为页面只能有一个 Form 控件，而 master 页面已经有了一个 Form 控件。

使用 master 页面的.aspx 页面可以包含不是指令的非根级内容、脚本元素和 Content 控件。可以有任意多个 Content 控件，每个 Content 控件都会把内容插入 master 页面的一个 ContentPlaceHolder 控件中。唯一要注意的是，确保 Content 控件的 ContentPlaceHolderID 属性匹配要插入内容的 ContentPlaceHolder 控件的 ID。所以，要把内容添加到前面的 master 页面上，只需编写下面的代码：

```
<%@ Page Language="C#"
  MasterPageFile "~/MyMasterPage.master"
  AutoEventWireup="true"
  CodeFile="Default2.aspx.cs" Inherits="Default2"
  Title="Untitled Page" %>
<asp:Content ID="Content1"
  ContentPlaceHolderID="head" Runat="Server" >
</asp:Content>
<asp:Content ID="Content1"
  ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server">
  Custom content!
</asp:Content>
```

master 页面的真正强大之处在于，把 ContentPlaceHolder 控件封装在其他页面内容中。例如导航控件、站点徽标和其他 HTML。可以为主要内容、边栏内容、脚标文本等提供多个 ContentPlaceHolder 控件。

如果不希望为特定的 ContentPlaceHolder 提供内容，就可以忽略页面上的 Content 控件。例如，可以从上面的代码中删除 Content1 控件，这不会影响最终的显示结果。

### 38.2.1 在 Web 页面中访问 Master 页面

给 Web 页面添加 master 页面时，有时需要在 Web 页面的代码中访问 master 页面。为此，可以使用 Page.Master 属性，它以 MasterPage 对象的形式返回对 master 页面的一个引用。可以把这个对象的

类型强制转换为 master 页面的类型，该类型由 master 页面文件定义(在上一节中，这个类称为 MyMasterPage)。有了这个引用后，就可以访问 master 页面类的任意公共成员了。

另外，还可以使用 MasterPage.FindControl()方法通过标识符定位 master 页面上的控件，以便处理 master 页面上内容占位符外部的内容。

一个典型用法是，如果定义了一个用于标准窗体的 master 页面，其中包含一个提交按钮，就可以在子页面上定位提交按钮，在 master 页面上为提交按钮提交事件处理程序。这样，就可以提供定制的验证逻辑，来响应窗体的提交了。

### 38.2.2 嵌套的 Master 页面

Master 页面选项 Select 也可以在创建新 Master 页面时使用。使用这个选项可以根据父 Master 页面创建嵌套的 Master 页面。例如，可以创建一个 Master 页面 MyNestedMasterPage，它使用 MyMasterPage：

```
< %@ Master Language="C#"
  MasterPageFile="~/MyMasterPage.master"
  AutoEventWireup="false"
  CodeFile="MyNestedMasterPage.master.cs"
  Inherits="MyNestedMasterPage" % >
< asp:Content ID="Content1" ContentPlaceHolderID="head"
  Runat="Server" >
<!-- Disabled for child controls. -->
</asp:Content >
< asp:Content ID="Content2"
  ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server" >
First nested place holder:
<asp:ContentPlaceHolder ID="NestedContentPlaceHolder1"
  runat="server" >
</asp:ContentPlaceHolder >
< br / >
< br / >
Second nested place holder:
<asp:ContentPlaceHolder ID="NestedContentPlaceHolder2"
  runat="server" >
</asp:ContentPlaceHolder >
</asp:Content >
```

使用这个 Master 页面的页面为 NestedContentPlaceHolder1 和 NestedContentPlaceHolder2 提供了内容，但不能直接访问 MyMasterPage 指定的 ContentPlaceHolder 控件。在这个例子中，MyNestedMasterPage 修改了 head 控件的内容，为 ContentPlaceHolder1 控件提供了一个模板。

创建一系列嵌套的 Master 页面，可以为页面提供其他布局，且不改变基本 Master 页面的某些方面。例如，根 Master 页面包含导航和基本布局，嵌套的 Master 页面可以为不同数量的列提供布局。

接着在站点的页面上使用嵌套的 Master 页面，使不同的页面在这些布局之间快速切换。

### 38.2.3 PCSDemoSite 中的 Master 页面

在 PCSDemoSite 中，使用了一个 master 页面 MasterPage.master(这是 master 页面的默认名称)，其代码如下所示：

```
<%@ Master Language="C#" AutoEventWireup="true"
CodeFile="MasterPage.master.cs"
Inherits="MasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<link rel="stylesheet" href="StyleSheet.css"
type="text/css" />
<title></title>
</head>
<body>
<form id="form1" runat="server">
<div id="header">
<h1><asp:literal ID="Literal1" runat="server"
text="<%$ AppSettings:SiteTitle %>" /></h1>
<asp:SiteMapPath ID="SiteMapPath1" Runat="server"
CssClass="breadcrumb" />
</div>
<div id="nav">
<div class="navTree">
<asp:TreeView ID="TreeView1" runat="server"
DataSourceID="SiteMapDataSource1" ShowLines="True" />
</div>
<br />
<br />
<asp:LoginView ID="LoginView1" Runat="server">
<LoggedInTemplate>
You are currently logged in as
<b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
<asp:LoginStatus ID="LoginStatus1" Runat="server" />
</LoggedInTemplate>
</asp:LoginView>
</div>
<div id="body">
<asp:ContentPlaceHolder ID="ContentPlaceHolder1"
Runat="server" />
</div>
</form>
```

```
<asp:SiteMapDataSource ID="SiteMapDataSource1"  
Runat="server" />  
</body>  
</html>
```

这里的许多控件前面都没有见过，稍后将介绍它们。这里要注意的是，`<div>`元素包含各个内容部分(标题、导航栏和页面体)，使用`<%$ AppSettings:SiteTitle %>`从 `Web.config` 文件中获得站点标题：

```
<appSettings>  
<add key="SiteTitle" value="Professional C# Demo Site"/>  
</appSettings>
```

还有 `StyleSheet.css` 的一个样式表链接：

```
<link rel="stylesheet" href="StyleSheet.css"  
type="text/css" />
```

这个 CSS 样式表包含此页面上`<div>`元素的基本布局信息，以及会议室登记工具控件的一部分。

```
div#header  
{  
position: absolute;  
top: 0px;  
left: 0px;  
height: 80px;  
width: 780px;  
padding: 10px;  
  
}  
  
div#nav  
{  
position: absolute;  
left: 0px;  
top: 100px;  
width: 180px;  
height: 580px;  
padding: 10px;  
  
}  
  
div#body  
{  
position: absolute;  
left: 200px;  
top: 100px;  
width: 580px;  
height: 580px;  
padding: 10px;
```

```
}
```

```
.mrbEventList
```

```
{
```

```
width: 40%;
```

```
}
```

注意，这个样式信息不包含颜色、字体等。这是主题中的样式表要获得的信息，主题详见本章后面的内容。这里只有布局信息，例如<div>的大小。

注意：

本章的站点尽可能遵循最佳实践方式。布局使用 CSS 而不是表格，很快就会成为 Web 站点布局的业界标准，所以要认真掌握。在上面的代码中，符号#用于格式化有指定 id 属性的<div>元素，.mrbEventList 格式化有指定 class 属性的 HTML 元素。

### 38.3 站点导航

有 3 个 Web 导航服务器控件 SiteMapPath、Menu 和 TreeView，可以用于为 Web 站点提供 XML 站点地图。如果使用另一个站点地图提供程序，站点地图就以另一种格式提供。一旦创建了这样一个数据源，这些 Web 导航服务器控件就可以自动为用户生成位置和导航信息。稍后介绍一个 XML 站点地图示例。

也可以使用 TreeView 控件显示其他结构化数据，但使用站点地图，可以提供导航信息的另一种视图。

Web 导航服务器控件如表 38-1 所示。

表 38-1

控件	说明
SiteMapPath	显示路径样式的信息，允许用户查看他们在站点结构中的位置，并导航到父区域中。 可以提供各种模板，例如 NodeStyle 和 CurrentNodeStyle，来定制路径信息的外观
Menu	通过 SiteMapDataSource 控件链接到站点地图信息上，可以查看完整的站点结构，其外观由模板定制
TreeView	可以在树型结构中显示层次化的数据，例如内容表。树中的节点存储在 Nodes 属性中，选中的节点存储在 SelectedNode 中。有几个事件可以在服务器端处理用户交互操作，包括 SelectedNodeChanged 和 TreeNode Collapsed。这个控件一般是数据绑定的

表 38-2

属性	说明
title	页面标题，用作站点地图中的链接文本
url	页面位置，用作站点地图中的超链接位置
roles	用户角色，允许查看菜单中的这个站点地图项
description	可选文本，用于站点地图的弹出式工具提示

站点有了 Web.sitemap 文件后，添加链接信息就只需在页面上放置如下代码：

```
<asp:SiteMapPath ID="SiteMapPath1" Runat="server" />
```

这将使用默认的提供程序和当前的 URL 位置，格式化父页面的链接列表。

添加一个菜单或树型视图菜单需要 SiteMapDataSource 控件，这也是非常简单的：

```
<asp:SiteMapDataSource ID="SiteMapDataSource1"  
Runat="server" />
```

如果使用定制的提供程序，唯一的区别是，可以通过 SiteMapProvider 属性指定该提供程序 ID。还可以使用 StartingNodeOffset 删除菜单数据的上一层(例如根级的 Home 项)，使用 ShowStartingNode="False" 将只删除顶级链接，使用 StartFromCurrentNode="True" 表示从当前位置开始，使用 StringNodeUrl 会重写根节点。

只要把 DataSourceID 设置为 SiteMapDataSource 的 ID，Menu 和 TreeView 控件就可以使用这个数据源中的数据。这两个控件都包含许多样式属性，并可以设置主题，详见本章后面的内容。

PCSDemoSite 中的导航

PCSDemoSite 的站点地图如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>  
<siteMap>  
  <siteMapNode url "~/Default.aspx" title="Home">  
    <siteMapNode url "~/About/Default.aspx" title="About" />  
    <siteMapNode url "~/MRB/Default.aspx" title="Meeting  
Room Booker"  
      roles="RegisteredUser,SiteAdministrator" />  
    <siteMapNode url "~/Configuration/Default.aspx"  
      title="Configuration"  
      roles="RegisteredUser,SiteAdministrator">  
      <siteMapNode url "~/Configuration/Themes/Default.aspx"  
        title="Themes"  
        roles="RegisteredUser,SiteAdministrator"/>  
    </siteMapNode>  
    <siteMapNode url "~/Users/Default.aspx" title="User  
Area"  
      roles="SiteAdministrator" />  
    <siteMapNode url "~/Login.aspx" title="Login Details" />  
  </siteMapNode>  
</siteMap>
```

PCSDemoSite 站点使用定制的提供程序从 Web.sitemap 中获得信息，这是必需的，因为默认的提供程序会忽略 roles 属性。这个定制提供程序在 Web 站点的 Web.config 文件中定义，如下所示：

```
<configuration>
  <xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
    ...
    <system.Web>
      ...
      <siteMap defaultProvider="CustomProvider">
        <providers>
          <add name="CustomProvider"
            description="SiteMap provider which reads in .sitemap XML
            files."
            type="System.Web.XmlSiteMapProvider, System.Web,
            Version=2.0.3600.0,
            Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
            siteMapFile="Web.sitemap" securityTrimmingEnabled="true" />
        </providers>
      </siteMap>
    ...
  ...
</configuration>
```

这个定制提供程序和默认提供程序的唯一区别是添加了 securityTrimmingEnabled = "true" , 它告诉提供程序 , 只为当前用户允许查看的节点提供数据。这种可见性是由用户的角色成员确定的 , 如下一节所述。

PCSDemoSite 中的 MasterPage.master 页面包含 SiteMapPath、 TreeView 导航显示和一个数据源 , 如下所示 :

```
<div id="header">
  <h1><asp:literal ID="Literal1" runat="server"
  text="<%$ AppSettings:SiteTitle %>" /></h1>
  <asp:SiteMapPath ID="SiteMapPath1" Runat="server"
  CssClass="breadcrumb" />
</div>
<div id="nav">
  <div class="navTree">
    <asp:TreeView ID="TreeView1" runat="server"
    DataSourceID="SiteMapDataSource1" ShowLines="True" />
  </div>
  <br />
  <br />
  <asp:LoginView ID="LoginView1" Runat="server">
    <LoggedInTemplate>
      You are currently logged in as
      <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
      <asp:LoginStatus ID="LoginStatus1" Runat="server" />
    </LoggedInTemplate>
  </asp:LoginView>
</div>
```

```
<div id="body">
<asp:ContentPlaceHolder ID="ContentPlaceHolder1"
Runat="server" />
</div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1"
Runat="server" />
```

唯一要注意的是，给 SiteMapPath 和 TreeView 提供了 CSS 类，以便于使用主题特性。

#### 38.4 安全性

在 Web 站点中，安全性和用户管理的实现常常是一个相当复杂的事情，这是有原因的，我们必须考虑许多因素，包括：

要实现哪种用户管理系统？用户要映射到 Windows 用户账户上吗？还是实现某种独立的管理系统？

如何实现登录系统？

是让用户在站点上注册吗？如何注册？

如何让一些用户只查看某些内容，只执行某些操作，而给另一些用户提供额外的权限？

忘记了密码，该怎么办？

在 ASP.NET 2.0 中，有一整套工具来处理这些问题，实际上，使用该工具只需几分钟就可以在站点上实现一个用户系统。我们有 3 种身份验证系统：

Windows 身份验证：用户有 Windows 账户，一般在内联网站点或 WAN 入口处使用

Forms 身份验证：Web 站点维护它自己的用户列表，完成自己的身份验证

Passport 身份验证：Microsoft 提供的一个集中式身份验证服务

完整讨论 ASP.NET 中的安全性至少需要一章的篇幅，但可以快速浏览一下实现安全性所涉及的工作。这里将只讨论 Forms 身份验证，因为这是最通用的系统，能很快建立和运转起来。

实现 Forms 身份验证的最快捷方式是使用 Website | ASP.NET Configuration 工具。上一章介绍过这个工具，它有一个 Security 选项卡，其中是一个安全向导。这个向导允许选择身份验证类型、添加角色、添加用户、保护站点的各个区域。

##### 38.4.1 使用安全向导添加 Forms 身份验证功能

为了便于说明，在 C:\ProCSharp\Chapter38\目录下创建一个新 Web 站点 PCSAuthenticationDemo。之后，打开 Website | ASP.NET Configuration 工具。进入 Security 选项卡。单击 Use the security setup wizard to configure security step by step 链接，阅读其中的信息后，单击第一步中的 Next，在第二步中，选择 From the internet，如-9 所示。

单击 Next，在确认使用默认的 Advanced provider settings 提供程序存储安全信息后，单击 Next。这个提供程序的信息可以通过 Provider 选项卡配置，在 Provider 选项卡中，可以选择把信息存储到其他地方，例如 SQL Server 数据库，但选择 access 数据库将便于演示。

选择 Enable roles for this Web site，如图 38-10 所示，单击 Next。

接着，添加一些角色，如图 38-11 所示。



Browse - ASP.NET Administration Tool Default.aspx Start Page

URL: http://localhost:43910/asp.netwebadminfiles/security/wizard/wizard.aspx

# ASP.NET Web Site Administration Tool

## Security Setup Wizard

Step 1: Welcome

Step 2: Select Access Method

Step 3: Data Store

**Step 4: Define Roles**

Step 5: Add New Users

Step 6: Add New Access Rules

Step 7: Complete

---

Security Management

### Define Roles

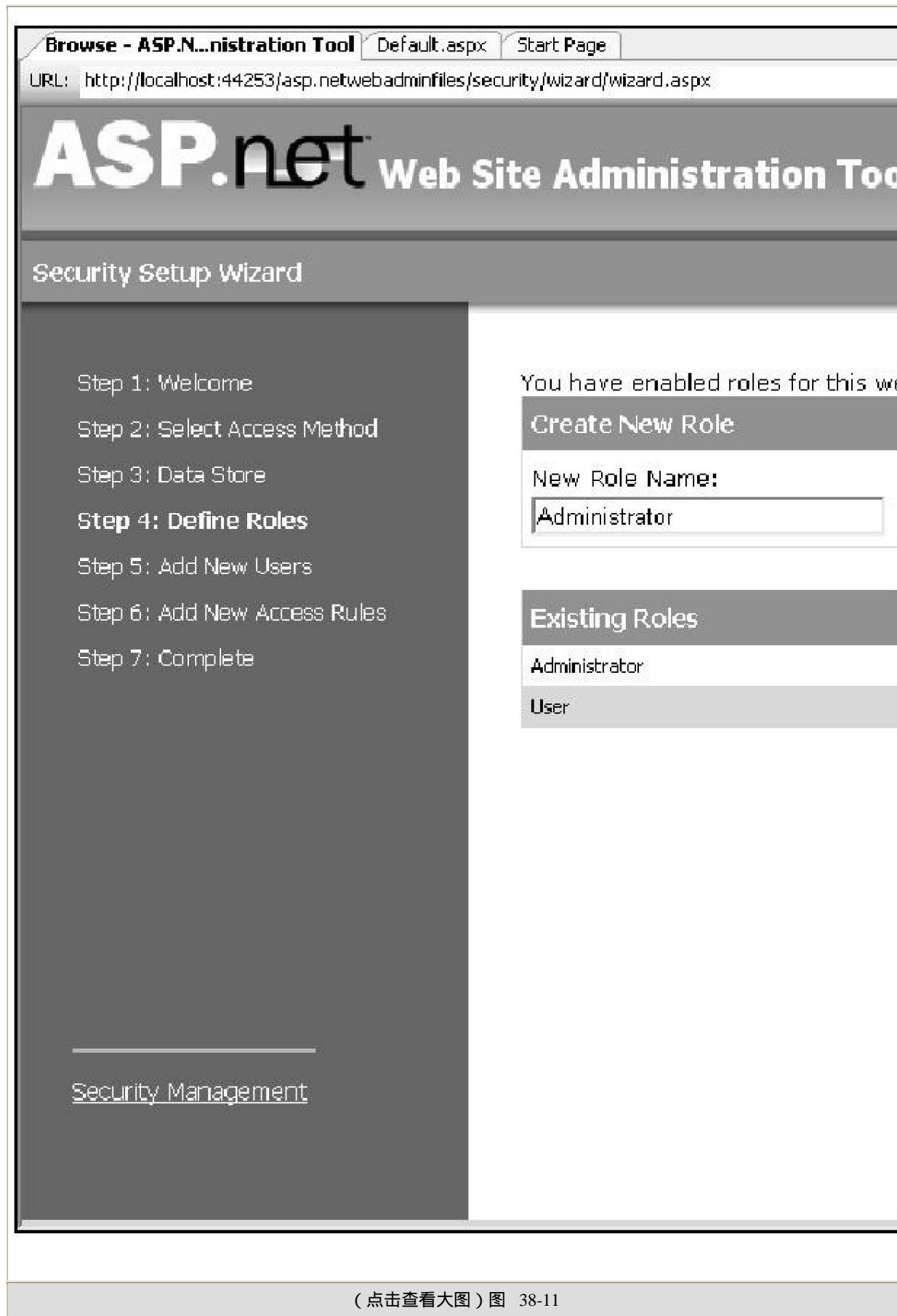
You can optionally add roles, or grant groups of users access to specific resources. You might create roles such as "manager" which will give different access to specific folders. Roles will have the access permissions assigned to them.

Type the name of the role that you want to create:

If you do not want to create roles, leave this field empty, uncheck the checkbox, and click Next to skip this step.

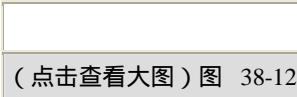
Enable roles for this Web site.

(点击查看大图) 图 38-10



(点击查看大图) 图 38-11

然后，添加一些用户，如图 38-12 所示。注意密码(在 machine.config 中定义)的默认安全角色是相当强的，密码至少有 7 个字符，至少包括 1 个符号和混合了大小写的字母。



提示：

在本章的下载代码中，为这个例子添加了两个用户。用户名分别是 User 和 Administrator，其密码均为 Pa\$\$w0rd。

单击 Next 按钮，可以定义站点的访问规则。在默认情况下，所有的用户和角色都可以访问站点的所有区域。在这个对话框中，可以限制角色、用户或匿名用户的访问区域。可以为站点的每个目录限制访问，因为这可以通过目录中的 Web.config 文件实现，如后面所示。现在跳过这一步，完成身份验证的设置。

最后一步是把用户赋予角色，这是通过 Security 选项卡上的 Manage users 链接完成的，在该选项卡上，可以编辑用户角色，如图 38-13 所示。

Browse - ASP.N... Administration Default.aspx Start Page

URL: http://localhost:44253/asp.netwebadminfiles/security/users/manageUsers.aspx

## ASP.NET Web Site Administration Tool

Home Security Application Provider

Click a row to select a user and then click **Edit user** to view or change his or her information. To assign roles to the selected user, select the appropriate check box.

To prevent a user from logging into your application but retain his or her information, change the user's status to inactive by clearing the check box.

### Search for Users

Search by: User name for:  Find User

Wildcard characters \* and ? are permitted.

A B C D E F G H I J K L M N O P Q R S

Active	User name	<a href="#">Edit user</a>	<a href="#">Delete user</a>	<a href="#">Edit roles</a>
<input checked="" type="checkbox"/>	Administrator	<a href="#">Edit user</a>	<a href="#">Delete user</a>	<a href="#">Edit roles</a>
<input checked="" type="checkbox"/>	User	<a href="#">Edit user</a>	<a href="#">Delete user</a>	<a href="#">Edit roles</a>

[Create new user](#)

(点击查看大图) 图 38-13

完成后，就有一个用户系统了，其中包含角色和用户。

现在就可以给 Web 站点添加几个控件，进行身份验证了。

### 38.4.2 实现登录系统

如果在运行安全向导后刷新 Solution Explorer，就会看到一个 Web.config 文件添加到项目中，其内容如下：

```
<roleManager enabled="true" />  
<authentication mode="Forms" />
```

这似乎不太像我们刚才做的工作，但注意许多信息存储在一个 SQL Express 数据库中，该数据库在 App\_Data 目录下，叫做 ASPNETDB.MDF。使用任意标准数据库管理工具都可以查看存储在这个文件中的数据，包括 Visual Studio。甚至可以直接在这个数据库中添加用户和角色。

在默认情况下，登录系统是通过 Web 站点根目录下的一个 Login.aspx 页面实现的。如果用户试图导航到无权访问的位置，他们就会自动重定向到这个页面，在成功登录后，就会进入希望的位置。

在 PCSAuthenticationDemo 站点中添加一个 Web 窗体 Login.aspx，把一个 Login 控件从工具箱拖放到这个窗体上。

这就是允许用户登录到 Web 站点上所需的全部工作。在浏览器上打开站点，导航到 Login.aspx，再输入在向导中添加的一个用户的信息，如图 38-14 所示。

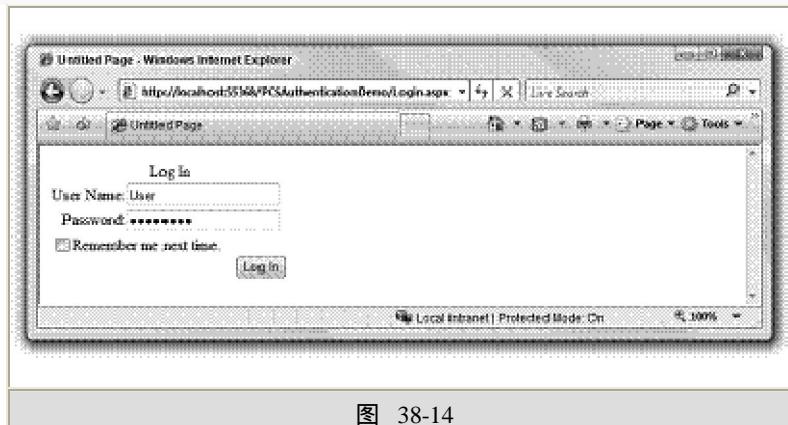


图 38-14

登录后，就会返回 Default.aspx，目前这是一个空页面。

### 38.4.3 Web 登录服务器控件

工具箱的 Login 部分包含几个控件，如表 38-3 所示。

表 38-3

控件	说明
Login	如前所示，这个控件允许用户登录 Web 站点。这个控件的大多数属性都用于给所提供的模板指定样式。还可以使用 DestinationPageUrl 强迫登录时重定向到指定的位置，使用 VisibleWhenLoggedIn 可以确定登录的用户是否能看到该控件，各种文本属性，如 CreateUserText，可以输出对用户有帮助的信息
LoginView	这个控件可以显示各种内容，这取决于用户是否登录，或用户的角色是什么。可以把内容放在<AnonymousTemplate>和<LoggedInTemplate>中，使用<RoleGroups>可以控制这个控件的输出
PasswordRecover	这个控件允许用户把密码邮寄给他自己，可以使用为用户定义的密码恢复问

ery	题。它的大多数属性也是用于格式化显示的，但 MailDefinition- Subject 属性用于配置发送给用户地址的电子邮件，SuccessPageUrl 属性在请求用户输入密码后重定向用户
LoginStatus	显示 Login 或 Logout 链接，这取决于用户是否登录，其文本和图像都可以定制
LoginName	给当前登录的用户输出用户名
CreateUserWiz ard	显示一个窗体，用户可以使用该窗体注册到站点上，并添加到用户列表中。与其他登录控件一样，它也有许多与布局格式相关的属性，但默认的格式已足够好了
ChangePasswor d	允许用户修改密码，它有 3 个字段，一个字段用于表示旧密码，其余两个字段表示新密码和确认密码。它也有许多样式属性

这些控件将在稍后的 PCSDemoSite 中使用。

#### 38.4.4 保护目录

最后要讨论的是如何限制对目录的访问。这可以通过前面介绍的 Site Configuration 工具实现，但手工完成也很简单。

给 PCSAuthenticationDemo 添加一个目录 SecureDirectory，在这个目录中添加 Web 页面 Default.aspx 和一个新的 Web.config 文件。用下面的代码替代 Web.config 的内容：

```
<?xml version="1.0" ?>
<configuration
    xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
    <system.web>
        <authorization>
            <deny users="?" />
            <allow roles="Administrator" />
            <deny roles="User" />
        </authorization>
    </system.web>
</configuration>
```

<authorization>元素可以包含一个或多个表示权限规则的<deny>或<allow>元素，每个元素都有一个 users 或 roles 属性，表示该规则应用于什么成员。规则是从上到下应用的，所以如果规则的成员关系有重叠，那么较特殊的规则一般应放在靠上的位置。在这个例子中，"?" 表示匿名用户，他们和 User 角色中的用户会被拒绝访问这个目录。注意如果这里的<allow>规则放在 User 角色的<deny>规则之前，则表示只有 User 和 Administrator 角色中的用户允许访问这个目录。所有的 users 角色都考虑在内，但规则的顺序仍是有用的。

现在，在登录到 Web 站点，导航到 SecureDirectory/Default.aspx 时，只有位于 Admin 角色中，才能访问该页面。其他用户或未通过身份验证的用户都会被重定向到登录页面。

#### 38.4.5 PCSDemoSite 中的安全性

PCSDemoSite 站点使用了前面介绍的 Login 控件、LoginView 控件、LoginStatus 控件、LoginName 控件、PasswordRecovery 控件和 ChangePassword 控件。

一个区别是包含了 Guest 角色，其结果是 guest 用户不能修改其密码，这使用 LoginView 比较合适，如 Login.aspx 所示：

```
<asp:Content ID="Content1"
ContentPlaceHolderID="ContentPlaceHolder1"
Runat="server">
<h2>Login Page</h2>
<asp:LoginView ID="LoginView1" Runat="server">
<RoleGroups>
<asp:RoleGroup Roles="Guest">
<ContentTemplate>
You are currently logged in as <b>
<asp:LoginName ID="LoginName1" Runat="server" /></b>.
<br />
<br />
<asp:LoginStatus ID="LoginStatus1" Runat="server" />
</ContentTemplate>
</asp:RoleGroup>
<asp:RoleGroup
Roles="RegisteredUser,SiteAdministrator">
<ContentTemplate>
You are currently logged in as <b>
<asp:LoginName ID="LoginName2" Runat="server" /></b>.
<br />
<br />
<asp:ChangePassword ID="ChangePassword1"
Runat="server">
</asp:ChangePassword>
<br />
<br />
<asp:LoginStatus ID="LoginStatus2" Runat="server" />
</ContentTemplate>
</asp:RoleGroup>
</RoleGroups>
<AnonymousTemplate>
<asp:Login ID="Login1" Runat="server">
</asp:Login>
<asp:PasswordRecovery ID="PasswordRecovery1"
Runat="Server" />
</AnonymousTemplate>
</asp:LoginView>
</asp:Content>
```

这里的视图会显示下述几个页面中的一个：

给匿名用户显示 Login 和 PasswordRecovery 控件。

给 Guest 用户显示 LoginName 和 LoginStatus 控件，如果需要，还会显示登录的用户名，并允许注销。

给 RegisteredUser 和 SiteAdministrator 用户显示 LoginName、LoginStatus 和 Change Password 控件。

该站点在各个目录中还包含各自的 Web.config 文件，以限制访问，也可以根据角色限制定向到其他地方。

注意：

为站点配置的用户显示在“关于”页面上，也可以添加自己的已配置用户。基本站点的用户(及其密码)是 User1(User1！！)、Admin/Admin!!)和 Guest(Guest!!)。

这里要注意，站点的根目录拒绝匿名用户，但 Themes 目录(详见下一节)重写了这个设置，允许匿名用户访问。这是必需的，因为如果没有重写该设置，匿名用户就会看到没有主题设置的站点，而主题文件是不能访问的。另外，根 Web.config 文件中的完整安全规范如下：

```
<configuration>
  <location path="StyleSheet.css">
    <system.web>
      <authorization>
        <allow users="?" />
      </authorization>
    </system.web>
  </location>
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
    ...
  </system.web>
</configuration>
```

其中，使用<location>元素重写了用 path 属性指定的文件的默认设置，这里是 StyleSheet.css 文件。<location>元素可以把任意<system.web>设置应用于指定的文件或目录，并可以把所有与目录相关的设置集中在一个地方(替代多个 Web.config 文件)。在上面的代码中，给匿名用户授予了访问 Web 站点中根样式表的权限，这是必需的，因为这个文件在 master 页面中定义了<div>元素的布局。不这么做，为匿名用户显示登录页面的 HTML 就很难读懂。

另一个要注意的地方是，在会议室登记工具控件的代码后置文件中，有如下 Page\_Load()事件处理程序：

```
void Page_Load(object sender, EventArgs e)
{
if (!this.IsPostBack)
{
nameBox.Text = Context.User.Identity.Name;
System.DateTime trialDate = System.DateTime.Now;
calendar.SelectedDate = GetFreeDate(trialDate);
}
}
```

其中，用户名是从当前的环境中提取出来的。在后台代码文件中，还要频繁使用 Context.User.IsInRole() 来检查访问。

### 38.5 主题

在 ASP.NET 页面中合并使用 master 页面和 CSS 样式表后，还要把窗体和函数分开，即把页面的外观和操作方式分开定义。利用主题，可以在一步中完成这个任务，并可以把所提供的几个主题之一动态应用于页面的外观。

主题包含如下内容：

主题的名称

可选的 CSS 样式表

可以给各个控件类型设置样式的 Skin 文件(.skin)

这些内容以两种不同的方式应用于页面：Theme 和 StyleSheetTheme：

Theme：所有的 skin 属性都应用于控件，重写页面上控件已有的所有属性

StyleSheetTheme：已有的控件属性优先于 skin 文件中定义的属性

CSS 样式表的工作方式与方法的使用方式相同，因为它们都以标准的 CSS 方式应用。

#### 38.5.1 把主题应用于页面

可以用几种声明或编程的方式把主题应用于页面，应用主题最简单的声明方式是在<%@ Page %>指令中使用 Theme 或 StyleSheetTheme 属性：

```
<%@ Page Theme="myTheme" ... %>
```

或者：

```
<%@ Page StyleSheetTheme="myTheme" ... %>
```

其中，myTheme 是给主题定义的名称。

另外，还可以在 Web 站点的 Web.config 文件中使用一项，给该站点上的所有页面指定要使用的主题：

```
<configuration>
  <xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
    <system.web>
      <pages Theme="myTheme" />
    </system.web>
  </configuration>
```

这里也可以使用 Theme 或 StyleSheetTheme 属性。还可以使用<location>元素重写某个页面或目录的这个设置，其方式与上一节中给安全信息使用该元素的方式相同。

如果采用编程方式，可以在页面的后台代码文件中应用主题，但只能在 Page\_PreInit()事件处理程序中应用主题，该事件在页面生存期的早期触发。在这个事件中，只需把 Page.Theme 或 Page.StyleSheetTheme 属性设置为要应用的主题名即可，例如：

```
protected override void OnPreInit(EventArgs e)
{
  Page.Theme = "myTheme";
}
```

通过代码应用主题时，可以动态应用一组主题中的一个主题文件。这个技巧将在 PCSDemoSite 中使用，如后面所示。

### 38.5.2 定义主题

主题在 ASP.NET 中的另一个特殊目录下定义，在这里是 App\_Themes。App\_Themes 目录可以包含任意多个子目录，每个子目录下有一个主题，子目录名就是主题名。

定义主题时，需要把主题的所有必要文件放在主题子目录下。对于 CSS 样式表，不需要考虑文件名，主题系统会自动查找扩展名为.css 的文件。同样，.skin 文件也可以有任意文件名，但最好使用多个.skin 文件，每个.skin 文件用于要设置样式的一个控件类型，每个.skin 文件的名称都用该控件名指定。

Skin 文件包含服务器控件的定义，其格式与标准 ASP.NET 页面中使用的格式相同。其区别是 skin 文件中的控件不会添加到页面上，它们只用于提取属性。按钮样式的定义一般放在 Button.skin 文件中，其内容如下所示：

```
<asp:Button Runat="server" BackColor="#444499"
  BorderColor="#000000"
  ForeColor="#ccccff" />
```

这个文件实际上是从 PCSDemoSite 的 DefaultTheme 主题中提取的，负责设置本章前面 Meeting Room Booker 页面中的按钮外观。

以这种方式为控件类型创建 skin 文件时，不需要使用 ID 属性。

### 38.5.3 PCSDemoSite 中的主题

Web 站点 PCSDemoSite 包含 3 个主题，可以在/Configuration/Themes/Default.aspx 页面上选择这 3 个主题--只要登录为 RegisteredUser 或 SiteAdministrator 角色的一个成员即可。这个页面如图 38-15 所示。



这里使用的主题是 DefaultTheme，也可以在这个页面上选择其他主题。图 38-16 显示了 BareTheme 主题。



这个主题适合于 Web 页面的可打印版本。BareTheme 目录并不包含文件，这里使用的文件是根样式表 StyleSheet.css。

图 38-17 显示了 LuridTheme 主题。



这个主题的颜色非常鲜亮，但很难阅读，它说明了站点的外观可以使用主题动态改变。更重要的是，像这样的主题可以提供 Web 站点的高对比度或大文本版本，以便于访问。

在 PCSDemoSite 上，当前选择的主题存储在会话状态中，所以在站点上浏览时，主题会保持不变。/Configuration/Themes/Default.aspx 的后台代码文件如下所示：

```
public partial class _Default : MyPageBase
{
    private void ApplyTheme(string themeName)
    {
        if (Session["SessionTheme"] != null)
        {
            Session.Remove("SessionTheme");
        }
        Session.Add("SessionTheme", themeName);
        Response.Redirect("~/Configuration/Themes", true);
    }

    void applyDefaultTheme_Click(object sender, EventArgs e)
    {
        ApplyTheme("DefaultTheme");
    }

    void applyBareTheme_Click(object sender, EventArgs e)
```

```
{  
    ApplyTheme( "BareTheme" );  
  
}  
  
void applyLuridTheme_Click(object sender, EventArgs e)  
{  
    ApplyTheme( "LuridTheme" );  
}  
}
```

这里的关键功能在 `ApplyTheme()` 中 , 它使用键 `SessionTheme` 把所选主题的名称放在会话状态中 , 并确定会话状态中是否已有一个主题 , 如果是 , 就删除它。

如前所述 , 主题必须在 `Page_PreInit()` 事件处理程序中应用 , 不能在所有页面使用的 master 页面中访问它 , 所以 , 如果要把选中的主题应用于所有的页面 , 有两个选项 :

在所有要应用主题的页面中 , 重写 `Page_PreInit()` 事件处理程序

为所有要应用主题的页面提供一个公共基类 , 再重写这个基类中的 `Page_PreInit()` 事件处理程序

PCSDemoSite 使用第二个选项 , 在 `Code/MyPageBase.cs` 中提供了一个公共页面基类 :

```
public class MyPageBase : Page  
{  
    protected override void OnPreInit(EventArgs e)  
    {  
        // theming  
        if (Session[ "SessionTheme" ] != null)  
        {  
            Page.Theme = Session[ "SessionTheme" ] as string;  
        }  
        else  
        {  
            Page.Theme = "DefaultTheme";  
        }  
  
        // base call  
        base.OnPreInit(e);  
    }  
}
```

这个事件处理程序检查 `SessionTheme` 中条目的会话状态 , 如果会话状态中有一项 , 就应用选中的主题 , 否则就使用 `DefaultTheme`。

注意这个类继承了通用的页面基类 `Page` , 这是必需的 , 否则页面就不能执行为一个 ASP.NET Web 页面。

为了使程序正常工作，还要为所有的 Web 页面指定这个基类。这有几种方式，最简单的方式是在页面的<@ Page %>指令或后台代码文件中指定。前者适合于简单的页面，但页面不能使用定制的后台代码文件，因为页面在其定制的后台代码文件中不再使用<@ Page %>指令。另一种方式是修改页面在后台代码文件中继承的类。在默认情况下，新页面继承了 Page，但可以改变这个继承。在前面主题选择页面的后台代码文件中，注意有如下代码：

```
public partial class _Default : MyPageBase
{
    ...
}
```

这里把 MyPageBase 指定为 Default 类的基类，所以使用在 MyPageBase.cs 中重写的方法。

### 38.6 Web Parts

ASP.NET 包含一组名为 Web Parts 的服务器控件，它们允许用户使 Web 页面个性化。在基于 SharePoint 的网站和 MSN 主页 <http://www.msn.com/> 上都可以看到这些控件。在使用 Web Parts 时，可以得到的功能如下：

给用户显示默认的页面布局。这个布局包含许多 Web Parts 组件，每个 Web Parts 都有标题和内容。

用户可以修改 Web Parts 在页面上的位置。

用户可以定制页面上 Web Parts 的外观，或者从页面中删除它们。

为用户提供一个 Web Parts 类别，允许用户将 Web Partst 拖放到页面上。

用户可以从页面中导出 Web Parts，再把它们导入另一个页面或站点上。

Web Parts 之间可以建立连接。例如，显示在一个 Web Parts 上的内容可以是显示在另一个 Web Parts 上的内容的图形表示。

用户进行的任何修改都可以在浏览站点的过程中保存下来。

ASP.NET 为使用 Web Parts 功能提供了一个完整的架构，包括管理和编辑控件。

Web Parts 的使用是一个复杂的课题，本节不描述 Web Parts 的所有功能，列出 Web Parts 组件提供的所有属性和方法，只概述 Web Parts，让读者理解它的基本功能。

#### 38.6.1 Web Parts 应用程序组件

工具箱的 Web Parts 部分包含 13 个控件，如图 38-18 所示(注意指针不是控件)。



图 38-18

这些控件如表 38-4 所示。该表还介绍了 Web Parts 页面的一些重要概念。

表 38-4

控件	说 明
WebPartManager	每个使用 Web Parts 的页面必须有一个(只能有一个)WebPartManager 控件的实例。可以把它放在 master 页面上。这个控件负责执行大部分 Web Parts 功能，不需要用户太多的干涉。只要根据自己需要的功能，将该控件放在 Web 页面上即可。对于更高级的功能，可以使用这个控件提供的许多属性和事件
ProxyWebPartManager	如果把 WebPartManager 控件放在 master 页面上，就很难在各个页面上配置它——实际上不能这么做。这与 Web Parts 之间静态连接的定义相关。ProxyWebPart Manager 控件可以在 Web 页面上声明性地定义静态连接，避免出现不能在同一个页面上放置两个 WebPartManager 控件的问题
WebPartZone	WebPartZone 控件用于定义可以包含 Web Parts 的页面区域。一般在页面上使用多个 WebPartZone 控件。例如，可以在页面的三列布局中使用三个 WebPartZone 控件。用户可以在 WebPartZone 区域之间移动 Web Parts，或者在一个 WebPartZone 中重新定位它们
CatalogZone	CatalogZone 控件允许用户把 Web Parts 添加到页面上。这个控件包含的控件派生于 CatalogZone，CatalogZone 提供了三个控件，本表的后面将介绍这三个控件。CatalogZone 及其包含的控件是否可见，取决于 WebPartManager 设置的当前显示模式
DeclarativeCatalogPart	DeclarativeCatalogPart 控件允许在线定义 Web Parts 控件。之后，这些控件就可以通过 CatalogZone 控件用于用户
PageCatalogPart	用户可以删除(关闭)显示在页面上的 Web Parts。为了检索它们，PageCatalogPart 控件提供了一组可以在页面上替换的、关闭的 Web Parts
ImportCatalogPart	ImportCatalogPart 控件允许通过 CatalogPart 接口把从页面中导出的 Web Part 再导入另一个页面
EditorZone	EditorZone 控件包含的控件允许用户根据 Web Parts 包含的控件，编辑 Web Parts 显示和行为的各个方面。它可以包含派生于 EditorPart 的控件，包括本表中后面的四个控件。与 CatalogZone 一样，这个控件的显示取决于当前显示模式
AppearanceEditorPart	这个控件允许用户修改 Web Parts 控件的外观和大小，并能隐藏它们
BehaviorEditorPart	这个控件允许用户使用它的许多属性配置 Web Parts 的行为，例如 Web Parts 是否可以关闭，Web Parts 的标题链接到什么 URL 上
LayoutEditorPart	这个控件允许用户修改 Web Parts 的布局属性，例如它包含在什么区域，在最小化状态下它是否显示
PropertyGridEditorPart	这是最一般的 Web Parts 编辑器控件，允许定义可以为定制 Web Parts 控件编辑的属性。之后，用户就可以编辑这些属性了
ConnectionsZone	这个控件允许用户在支持连接功能的 Web Parts 之间创建连接。与 CatalogZone 和 EditorZone 不同，这个控件中不放置任何其他控件。这个控件生成的用户界面取决于页面上可以进行连接的控件。这个控件的可见性取决于显示模式

注意，表 38-4 中的控件没有包含任何特定的 Web Part 控件。这是因为这些控件是我们自己创建的。任何放在 WebPartZone 区域中的控件会自动变成 Web Part，包括(最重要的)用户控件。使用用户控件，可以把其他控件组合在一起，提供 Web Part 控件的用户界面和功能。

### 38.6.2 Web Parts 示例

为了演示 Web Parts 的功能，可以查看本章下载代码中的示例 PCSWebParts。这个示例将使用 PCSAuthenticationDemo 例子的安全数据库。它有两个用户，其用户名是 User 和 Administrator，密码是 Pa\$\$w0rd。还可以登录为用户，处理页面上的 Web Parts，注销，之后再登录为另一个用户，以完全不同的方式处理 Web Parts。这两个用户的个性化会在站点访问的过程中保存下来。

登录到站点上后，显示的结果(用 User 登录)如图 38-19 所示。

图 38-19

这个页面包含如下控件：

1 个 WebPartManager 控件(没有可见的组件)

3 个 WebPartZone 控件

3 个 Web Parts(Date、Events 和 User Info)，分别放在 3 个 WebPartZone 控件中，其中两个 Web Parts 通过一个静态连接关联起来，如果修改 Date 中的日期，Events 中显示的日期就会更新。

改变显示模式的下拉列表。这个列表没有包含所有可能的显示模式，只包含了可用的显示模式。可用的模式是从 WebPartManager 控件中获得的，如后面所示。列出的模式有：

o Browse：这个模式是默认的，允许查看和使用 Web Parts。在这个模式下，每个 Web Part 都可以使用下拉菜单最小化和关闭，下拉菜单可以从每个 Web Part 的右上角访问。

o Design：在这个模式下，可以重新定位 Web Parts。

o Edit：在这个模式下，可以编辑 Web Part 属性。每个 Web Part 的下拉菜单中，有一个额外的菜单项 Edit。

o Catalog：在这个模式下，可以给页面添加新 Web Parts。

一个链接。将 Web Part 布局重置为默认情况(仅用于当前用户)

一个 EditorZone 控件(只在 Edit 模式下可见)

一个 CatalogZone 控件(只在 Catalog 模式下可见)

类别中可以添加到页面上的一个额外 Web Part

每个 Web Part 都在用户控件中定义。

为了演示布局的修改过程，可使用下拉列表将显示模式改为 Design。注意每个 WebPartZone 都带有一个 ID 值(分别是 LeftZone、CenterZone 和 RightZone)。还可以通过拖动标题来移动 Web Parts，在拖动过程中甚至可以看到反馈。如图 38-20 所示，其中显示了正在移动的名为 Date 的 Web Parts。

图 38-20

接着，在类别中添加一个新的 Web Part。将显示模式改为 Catalog，注意 CatalogZone 在页面的底部可见。单击 Declarative Catalog 链接，就可以在页面上添加一个 Links 控件，如图 38-21 所示。

图 38-21

注意这里还有一个 Page Catalog 链接。如果使用下拉菜单选择 Web Part，就会看到 Page Catalog 链接，它没有删除，只是隐藏了。

之后，把显示模式改为 Edit，再从 Web Part 的下拉列表中选择 Edit，如图 38-22 所示。



图 38-22

选择这个菜单项时，会打开 EditorZone 控件。在本例中，这个控件包含一个 AppearanceEditorPart 控件，如图 38-23 所示。

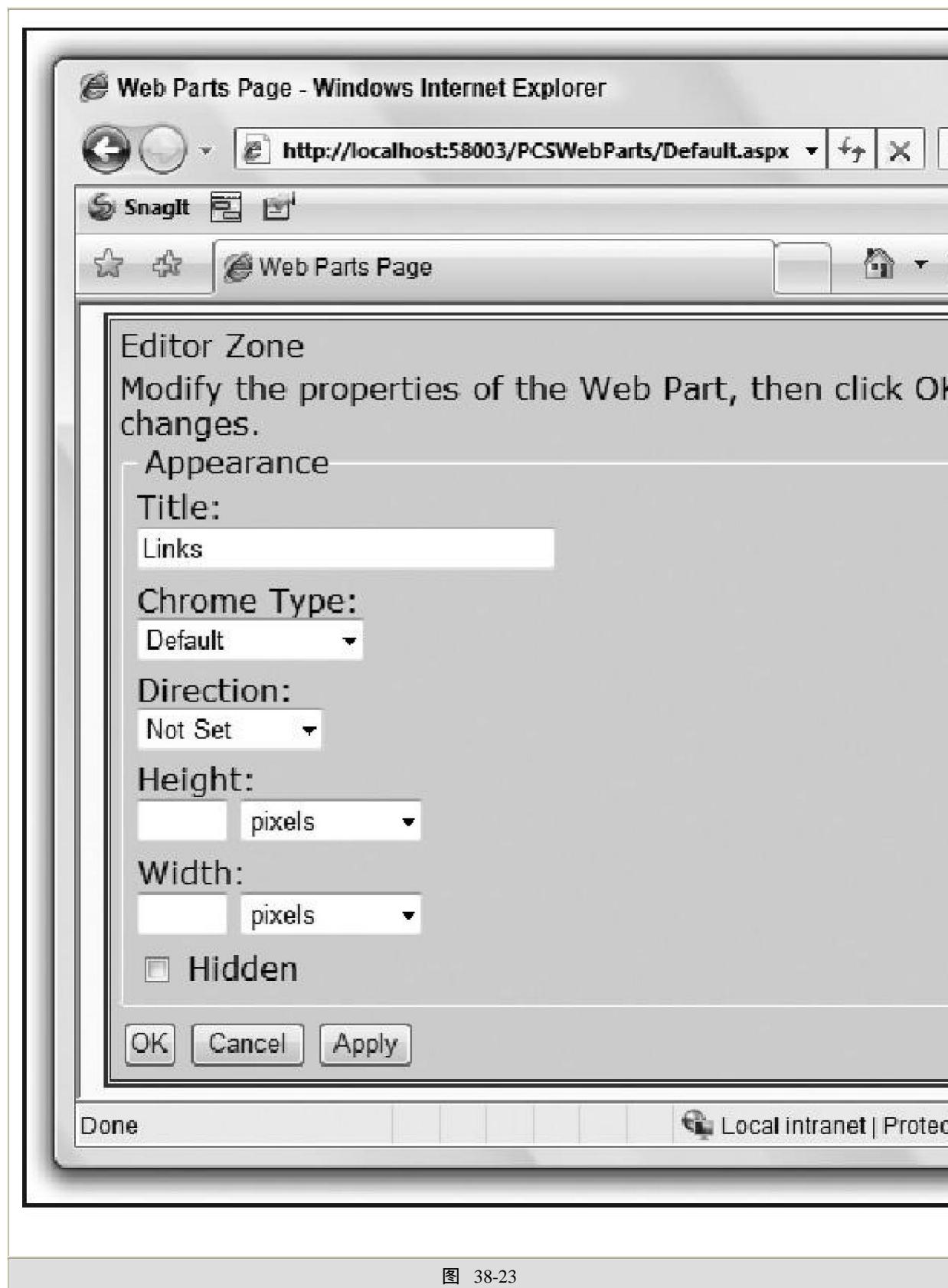


图 38-23

使用这个界面可以编辑和应用 Web Parts 的属性值。

完成了修改后，确认存储了它们，以便用户注销，再登录为另一个用户，之后切换回第一个用户。

现在，这个功能需要许多代码。实际上，本例的代码是相当简单的。查看 Web Parts 页面的代码。

<form>元素以一个 WebPartManager 控件开头：

```
<form id="form1" runat="server">
<asp:WebPartManager ID="WebPartManager1" runat="server"
OnDisplayModeChanged="WebPartManager1_DisplayModeChanged">
<StaticConnections>
<asp:WebPartConnection ID="dateConnection"
ConsumerConnectionPointID="DateConsumer"
ConsumerID="EventListControl1"
ProviderConnectionPointID="DateProvider"
ProviderID="DateSelectorControl1" />
</StaticConnections>
</asp:WebPartManager>
```

这个控件的 DisplayModeChanged 事件有一个处理程序 , 用于显示或隐藏页面底部的 <div> 编辑器。在 Date 和 Events 这两个 Web Parts 之间的静态连接也有一个规范 , 为此 , 要为用于这些 Web Parts 的两个用户控件中的连接定义端点 , 并引用这些端点。代码稍后列出。

接着定义标题、显示模式改变器和重置链接 :

```
<div class="mainDiv">
<h1>Web Parts Page</h1>
Display mode:
<asp:DropDownList ID="displayMode" runat="server"
AutoPostBack="True"
OnSelectedIndexChanged="displayMode_SelectedIndexChanged"
/>
<br />
<asp:LinkButton runat="server" ID="resetButton"
Text="Reset Layout"
OnClick="resetButton_Click" />
<br />
<br />
```

使用 WebPartManager1.SupportedDisplayModes 属性 , 在 Page\_Load() 事件处理程序中填充显示模式下拉列表。重置按钮使用 WebPartManager1.Personalization.ResetPersonalizationState() 方法给当前用户重置个性化状态。

接着是三个 WebPartZone 控件 , 每个控件都包含一个加载为 Web Part 的用户控件 :

```
<div class="innerDiv">
<div class="zoneDiv">
<asp:WebPartZone ID="LeftZone" runat="server">
<ZoneTemplate>
<uc1:DateSelectorControl ID="DateSelectorControl1"
runat="server"
title="Date" />
</ZoneTemplate>
```

```
</asp:WebPartZone>
</div>
<div class="zoneDiv">
<asp:WebPartZone ID="CenterZone" runat="server">
<ZoneTemplate>
<uc2:EventListControl ID="EventListControl1"
runat="server"
title="Events" />
</ZoneTemplate>
</asp:WebPartZone>
</div>
<div class="zoneDiv">
<asp:WebPartZone ID="RightZone" runat="server">
<ZoneTemplate>
<uc4:UserInfo ID="UserInfo1" runat="server" title="User
Info" />
</ZoneTemplate>
</asp:WebPartZone>
</div>
```

最后是 EditorZone 和 CatalogZone 控件 ,它们分别包含一个 AppearanceEditor 控件、PageCatalogPart 和 DeclarativeCatalogPart 控件 :

```
<asp:PlaceHolder runat="server" ID="editorPH"
Visible="false">
<div class="footerDiv">
<asp:EditorZone ID="EditorZone1" runat="server">
<ZoneTemplate>
<asp:AppearanceEditorPart ID="AppearanceEditorPart1"
runat="server" />
</ZoneTemplate>
</asp:EditorZone>
<asp:CatalogZone ID="CatalogZone1" runat="server">
<ZoneTemplate>
<asp:PageCatalogPart ID="PageCatalogPart1"
runat="server" />
<asp:DeclarativeCatalogPart
ID="DeclarativeCatalogPart1"
runat="server">
<WebPartsTemplate>
<uc3:LinksControl ID="LinksControl1" runat="server"
title="Links" />
</WebPartsTemplate>
</asp:DeclarativeCatalogPart>
</ZoneTemplate>
</asp:CatalogZone>
```

```
</div>
</asp:PlaceHolder>
</div>
</div>
</form>
```

DeclarativeCatalogPart 控件包含第四个用户控件，这是一个 Links 控件，用户可以把它添加到页面上。

Web Parts 的代码是相当简单的。例如，Links 控件只包含下述代码：

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="LinksControl.ascx.cs"
Inherits="LinksControl" %>
<a href="http://www.msn.com/">MSN</a>
<br />
<a href="http://www.microsoft.com/">Microsoft</a>
<br />
<a href="http://www.wrox.com/">Wrox Press</a>
```

不需要额外的标记，就可以使这个用户控件作为一个 Web Part 工作。这里唯一要注意的是，用户控件的<uc3:LinksControl>元素有一个 title 属性，但用户控件没有 Title 属性。这个属性由 DeclarativeCatalogPart 控件给 Web Part 推断要显示的标题(可以在运行期间用 AppearanceEditorPart 编辑)。

将一个接口引用从 DateSelectorControl 传送给 EventListControl(这些 Web Parts 使用的两个用户控件类)，就建立了 Date 和 Events 控件之间的连接。

```
public interface IDateProvider
{
    SelectedDatesCollection SelectedDates
    {
        get;
    }
}
```

DateSelectorControl 支持这个接口，所以可以使用 this 传送 IDateProvider 的一个实例。引用通过 DateSelectorControl 中的端点方法来传递，该引用是用 ConnectionProvider 属性修饰的：

```
[ConnectionProvider("Date Provider", "DateProvider")]
public IDateProvider ProvideDate()
{
    return this;
}
```

这就把 Web Part 标记为一个提供程序控件了。之后，就可以通过端点 ID 引用提供程序了，本例是 DateProvider。

要使用提供程序，可以使用 ConnectionConsumer 属性在 EventListControl 中指定一个使用方法：

```
[ConnectionConsumer( "Date Consumer", "DateConsumer" )]  
public void GetDate( IDateProvider provider )  
{  
    this.provider = provider;  
    IsConnected = true;  
    SetDateLabel();  
}
```

这个方法存储了一个传送过来的 IDateProvider 接口引用，设置一个标记，修改控件中的标签文本。

这个例子没有更多需要解释的地方。代码中有几个小装饰段，还有 Page\_Load() 中事件处理程序的信息，但这里不需要讨论它们。查看本章的下载代码，可以进一步研究它们。

但是 Web Parts 可以完成更多的工作。Web Parts 架构非常强大，功能非常丰富，需要一整本书的篇幅来探讨。但本节概述了 Web Parts，揭密了它们的一些功能。

### 38.7 小结

本章介绍了创建 ASP.NET 页面和 Web 站点的几个高级技术，并在示例 Web 站点 PCSDemoSite 中演示了这些技术。

首先探讨了如何使用 C# 创建可重用的 ASP.NET 服务器控件。其中讨论了如何从现有的 ASP.NET 页面中创建简单的用户控件，也讨论了如何从头创建定制控件。还研究了如何把上一章的会议室登记工具示例修改为一个用户控件。

接着介绍了 master 页面，如何为 Web 站点的页面提供模板，这是重用代码和简化开发的另一种方式。在 PCSDemoSite 中，有一个 master 页面包含 Web 导航服务器控件，允许用户在站点上浏览，并建立了主题的框架。本章后面介绍的主题非常适合于把功能与设计分开，是一个强大的可访问性技术。

我们还简要介绍了安全性，探讨了如何在 Web 站点上实现基于窗体的身份验证。

最后研究了 Web Parts，介绍了如何使用 Web Parts 服务器控件把基本应用程序和这个技术提供的某些功能集成在一起。

本章仅涉及 ASP.NET 2.0 功能的皮毛。例如，使用定制控件可以完成许多任务，模板和后期绑定的讨论是非常有趣的，并可以据此创建控件。但掌握了本章的内容后，就可以开始建立自己的定制控件了，还可以试验本章讨论的其他技术。

下一章介绍使用 Ajax 技术使 ASP.NET 应用程序更动态化的方式。

## 第 39 章 ASP.NET AJAX

Web 应用程序编程是一个不断变化和改进的主题。前面两章介绍了如何使用 ASP.NET 创建功能全面的 Web 应用程序，读者可能以为，前面已经探讨了创建自己的 Web 应用程序所需的所有工具。但是，如果花点时间查看当前的网站，就会注意到，最近的网站在使用方面比老网站好得多。许多目前最好的网站都提供了丰富的用户界面，其响应能力与 Windows 应用程序差不多。它们是使用客户端处理技术实现的，主要是 JavaScript 代码和一种新技术 Ajax。

出现这个变化，是因为客户用于浏览网站的浏览器和客户用于运行浏览器的计算机更强大了。Web 浏览器的当前版本，例如 Internet Explorer 7 和 Firefox，也支持各种标准。这些标准，包括 JavaScript，使 Web 应用程序提供的功能比使用普通的 HTML 提供的功能强大得多。前面的章节介绍了一些这方面的能力，例如使用层叠样式表(CSS)设置 Web 应用程序的样式。

本章介绍的 Ajax 并不是一个新技术，它只是一个标准的合并，以识别当前 Web 浏览器的丰富的潜在功能。

在支持 Ajax 的 Web 应用程序中，最重要的特性是 Web 浏览器能在操作的外部与 Web 服务器通信。这称为异步回送或部分页面的回送。实际上，这意味着用户可以与服务器端的功能和数据交互，而无需更新整个页面。例如，单击一个链接，移动到表的第二页数据上时，Ajax 可以只刷新表的内容，而不刷新整个 Web 页面。也就是说，需要的 Internet 通信量较少，从而使 Web 应用程序的响应比较快。本章的后面将介绍这个例子，还会举许多例子来说明 Ajax 在 Web 应用程序中的巨大作用。

本章将在代码中使用 Ajax 的 Microsoft 实现方式，它称为 ASP.NET AJAX。这个实现方式采用了 Ajax 模型，将它应用于 ASP.NET 架构。ASP.NET AJAX 提供了许多服务器控件和客户端技术，它们专用于 ASP.NET 开发人员，可以毫不费力地在 Web 应用程序中添加 Ajax 功能。

本章的内容如下：

首先学习 Ajax 和实现 Ajax 的技术。

学习 ASP.NET AJAX 及其组成部分，以及 ASP.NET AJAX 提供的功能。

介绍如何通过服务器端和客户端代码在 Web 应用程序中使用 ASP.NET AJAX。这是本章最大的一部分。

### 39.1 Ajax 的概念

Ajax 允许通过异步回送和动态的客户端 Web 页面处理，改进 Web 应用程序的用户界面。术语 "Ajax" 由 Jesse James Garrett 提出，是 Asynchronous JavaScript and XML 的缩写。

提示：

Ajax 不是一个缩写词，因此不能写作 AJAX。但是，在产品名称 ASP.NET AJAX 中它是大写，这是 Ajax 的 Microsoft 实现方式，如下一节所述。

根据定义，Ajax 显然涉及到 JavaScript 和 XML。但是，Ajax 编程需要使用其他技术，如表 39-1 所述。

表 39-1

技术	说 明
HTML/XHTML	HTML(超文本标记语言，Hypertext Markup Language)是显示和布局语言，由 Web 浏览器用于在图形化用户界面上显示信息。在前面两章中，学习了 HTML 如何实现这个功能，ASP.NET 如何生成 HTML 代码。可扩展的 HTML(XHTML)是使用 XML 结构的一个较严谨的 HTML 版本
CSS	CSS(层叠样式表)是 HTML 元素根据一个样式表中定义的规则设置样式的方式。它允许将样式同时应用于多个 HTML 元素，能在不修改 HTML 的情况下改变 Web 页面的外观。CSS 包含布局和样式信息，所以也可以使用 CSS 在页面上定位 HTML 元素。前面的章节还在例子中介绍了具体的操作
DOM	DOM(文档对象模型)是在层次结构中表示和处理(X)HTML 代码的一种方式。它允许访问页面上的“表 x 中第三行的第二列”，且无需使用比较基本的文本处理方式定位这个元素
JavaScript	JavaScript 是一个客户端脚本编辑技术，允许在 Web 浏览器上执行代码。JavaScript 的语法类似于其他基于 C 的语言，包括 C#，提供了变量、函数、分支代码、循环语句、事件处理程序和其他编程元素。但是与 C# 不同，JavaScript 不是强类型化的，JavaScript 代码的调试比较困难。对于 Ajax 编程，JavaScript 是一种关键技术，因为它允许利用 DOM 处理功能，动态修改 Web 页面
XML	XML 是标记数据的一种独立于平台的方式，对 Ajax 非常关键，它既是处理数据的方式，也是客户机和服务器之间的通信语言
XmlHttpRequest	自 Internet Explorer 5 以来，浏览器就把 XmlHttpRequest API 作为一种在客户机和服务器之间进行异步通信的方式。Microsoft 最初把它引入为一种技术，以访问通过 Internet 存储在 Exchange 服务器中的电子邮件，它使用的产品叫做 Outlook Web Access。后来它变成在 Web 应用程序中进行异步通信的标准方式，是支持 Ajax 的 Web 应用程序的一个核心技术。这个 API 的 Microsoft 实现方式称为 XMLHTTP，它利用所谓的 XMLHTTP 协议来通信

Ajax 还需要用服务器端代码处理部分页面的回送和完整页面的回送，这包括服务器控件的事件处理程序和 Web 服务(Web 服务详见第 37 章)。图 39-1 显示了这些技术如何在 Ajax Web 浏览器模型中联合使用，并与传统的 Web 浏览器模型进行比较。

在 AJAX 推出之前，表 39-1 中的前四个技术(HTML、CSS、DOM 和 JavaScript)用于创建所谓的动态 HTML(DHTML) Web 应用程序。这些应用程序比较著名有两个原因：它们提供的用户界面要好得多；它们一般只能用于一种类型的 Web 浏览器。

自 DHTML 推出以来，标准已有了改进，Web 浏览器的相关标准级别也提高了。但是，它们仍有区别，Ajax 解决方案必须考虑这些区别。也就是说，大多数开发人员实现 Ajax 解决方案还相当慢。只有开发出更抽象的 Ajax 架构(例如 ASP.NET AJAX)，创建支持 Ajax 的网站才是企业级开发的一个可行选项。

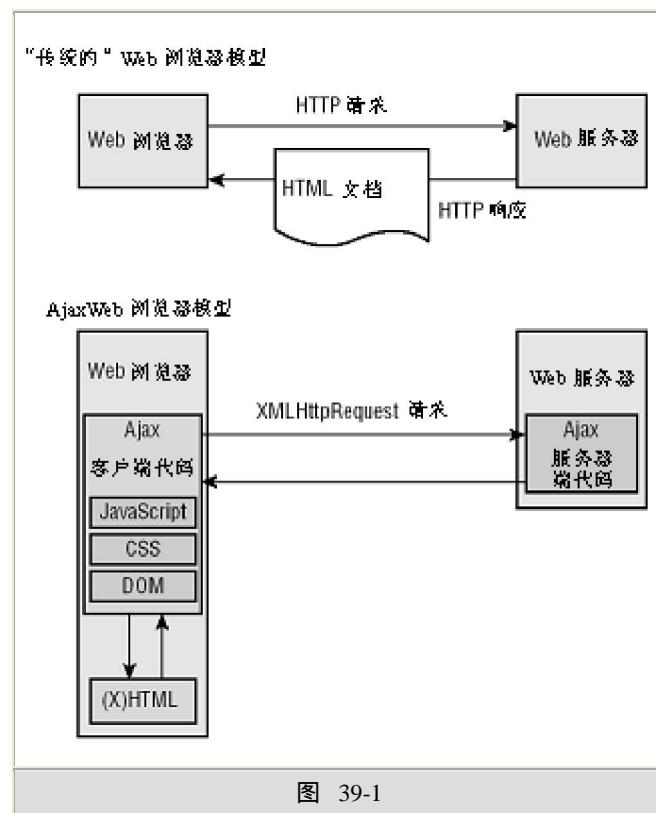


图 39-1

## 39.2 ASP.NET AJAX

ASP.NET AJAX 是 Ajax 架构的 Microsoft 实现方式，专用于 ASP.NET 开发人员。在 ASP.NET 的最新版本中，ASP.NET AJAX 是 ASP.NET 核心功能的一部分。网站 <http://ajax.asp.net> 上可以用于以前的 ASP.NET 版本，包含相关的文档说明、论坛和示例代码，可以用于我们使用的 ASP.NET 版本。

ASP.NET AJAX 提供了如下功能：

服务器端架构允许 ASP.NET Web 页面响应部分页面的回送操作。

ASP.NET 服务器控件便于实现 Ajax 功能。

HTTP 处理程序允许 ASP.NET Web 服务在部分页面的回送操作中，使用 JavaScript Object Notation(JSON)串行化功能与客户端代码通信。

Web 服务支持客户端代码访问 ASP.NET 应用程序服务，包括身份验证和个性化服务。

网站模板可用于创建支持 ASP.NET AJAX 的 Web 应用程序。

客户端的 JavaScript 库对 JavaScript 语法进行了许多改进，还提供了许多代码，来简化 Ajax 功能的实现。

这些服务器控件和服务器端的架构统称为 ASP.NET Extensions。ASP.NET AJAX 的客户端部分称为 AJAX 库。

还可以从网站 <http://ajax.asp.net> 上下载另外两个软件包：

ASP.NET AJAX Control Toolkit：这个下载软件包包含了由开发团体创建的其他服务器控件。这些控件是共享的，可以查看和修改它们。

Microsoft AJAX Library 3.5：这个下载软件包包含 JavaScript 客户端架构，它们由 ASP.NET AJAX 用于执行 Ajax 功能。如果开发的是 ASP.NET AJAX 应用程序，就不需要这个软件包。这个下载软件包适用于其他语言，例如 PHP，使用与 ASP.NET AJAX 相同的代码基执行 Ajax 功能。它超出了本章的范围。

提示：

还有一个下载软件包称为 Futures，过去用于给 ASP.NET AJAX 应用程序添加额外的、预先发布功能或原始功能。但在编写本书时，不清楚这个下载软件包是否在 VS 2008 中得到支持，所以本章不介绍它。

这两个下载软件包提供了功能丰富的架构，可以用于在自己的 ASP.NET Web 应用程序中添加 Ajax 功能。下面几节将介绍 ASP.NET AJAX 的各个组成部分。

### 39.2.1 核心功能

ASP.NET AJAX 的核心功能分为两个部分：AJAX 扩展和 AJAX 库。

#### 1. AJAX 扩展

在安装 ASP.NET AJAX 时，会在 GAC 中安装两个程序集：

System.Web.Extensions.dll：这个程序集包含 ASP.NET AJAX 功能，包括 AJAX 扩展和 AJAX 库 JavaScript 文件，它们可以通过 ScriptManager 组件(稍后介绍)来获得。

System.Web.Extensions.Design.dll：这个程序集包含用于 AJAX 扩展服务器控件的 ASP.NET Designer 组件，这些服务器控件由 ASP.NET Designer 在 Visual Studio 或 Visual Web 开发程序中使用。

ASP.NET AJAX 中的许多 AJAX 扩展组件都涉及支持部分页面的回送和用于 Web 服务的 JSON 串行化。这包括各种 HTTP 处理程序组件和对已有 ASP.NET 架构的扩展。这些功能都可以通过网站的 Web.config 文件来配置。还有用于其他配置的类和属性。但大多数配置都是透明的，用户很少需要改变支持 ASP.NET AJAX 的网站模板提供的默认设置。

与 AJAX 扩展的主要交互操作是使用服务器控件将 Ajax 功能添加到 Web 应用程序中。有几个服务器控件可以用各种方式增强 Web 应用程序。表 39-2 列出了一些服务器端组件。本章的后面将介绍它们。

表 39-2

控 件	说 明
ScriptManager	这个控件是 ASP.NET AJAX 功能的核心，使用部分页面回送功能的每个页面都需要它。它的主要作用是管理对 AJAX 库 JavaScript 文件的客户端引用，AJAX 库 JavaScript 文件位于 ASP.NET AJAX 程序集中。AJAX 库主要由 AJAX 扩展服务器控件使用，这些控件会生成自己的客户端代码。

	<p>这个控件还负责配置要在客户端代码中访问的 Web 服务。给 ScriptManager 控件提供 Web 服务的信息，就可以生成客户端类和服务器端类，来透明地管理与 Web 服务的异步通信。</p> <p>还可以使用 ScriptManager 控件维护对自己的 JavaScript 文件的引用，和 Futures CTP 中包含的其他 JavaScript 文件的引用</p>
UpdatePanel	<p>UpdatePanel 控件非常有用，也许是最常用的 ASP.NET 控件。这个控件与标准的 ASP.NET 占位符类似，可以包含其他控件。更重要的是，在部分页面的回送过程中，它还把页面的一个部分标记为可以独立于其他页面部分来更新的区域。</p> <p>UpdatePanel 控件包含的、产生回送操作的任意控件(如按钮控件)，都不会产生整个页面的回送操作，它们只执行部分页面的回送，只更新 UpdatePanel 的内容。</p> <p>在许多情况下，只需要这个控件实现 AJAX 功能。例如，可以把一个 GridView 控件放在 UpdatePanel 控件中，该控件的分页、排序和其他回送功能都在部分页面的回送过程中发挥作用</p>
UpdateProgress	<p>在部分页面的回送过程中，这个控件可以为用户提供反馈。在更新 UpdatePanel 时，可以为要显示的 UpdateProgress 控件提供一个模板。例如，可以使用浮点数的&lt;div&gt; 控件显示消息“ Updating... ”，告诉用户应用程序正在忙。注意部分页面的回送不会干扰 Web 页面的其他区域，其他区域仍可以响应</p>
Timer	<p>ASP.NET AJAX 的 Timer 控件是使 UpdatePanel 定期更新的一种有效方式。可以把这个控件配置为定期触发回送操作。如果这个控件包含在 UpdatePanel 控件中，则每次触发 Timer 控件时，都会更新该 UpdatePanel 控件。Timer 控件也有关联的事件，所以可以执行定期的服务器端处理</p>
AsyncPostBackTrigger	<p>这个控件可以在未包含在 UpdatePanel 中的控件里触发 UpdatePanel 的更新操作。例如，可以在 Web 页面的其他地方放置一个下拉列表，来更新包含 GridView 控件的 UpdatePanel</p>

AJAX 扩展还包含 ExtenderControl 抽象基类，来扩展已有的 ASP.NET 服务器控件。它由 ASP.NET 2.0 AJAX Futures CTP 中的各种类使用，如后面所述。

## 2. AJAX 库

在支持 ASP.NET AJAX 的 Web 应用程序中，AJAX 库包含的 JavaScript 文件由客户端代码使用。在这些 JavaScript 文件中包含许多功能，其中一些是改进 JavaScript 语言的通用代码，一些则专用于 Ajax 功能。AJAX 库包含的功能彼此互为基础，如表 39-3 所示。

表 39-3

功 能 层	说 明
浏览器兼容性	AJAX 库的最底层代码根据客户机的 Web 浏览器来映射各种 JavaScript 功能，这是必需的，因为 JavaScript 在不同浏览器中的实现方式是有区别的。提供这个功能层，其他层上的 JavaScript 代码就不必考虑浏览器的兼容性了，我们也可以编写独立于浏览器、在所有客户机环境中工作的代码
核心服务	这一层包含对 JavaScript 语言的增强，尤其是 OOP 功能。使用这一层的代码，可以使用 JavaScript 脚本文件定义命名空间、类、派生类和接口。C# 开发人员对此特别感兴趣，因为它使 JavaScript 代码的编写非常类似于用 C# 编写.NET 代码，且鼓励代码的重用
基类库	客户基类库(BCL)包含许多 JavaScript 类，它们为 AJAX 库层次结构中下层的类提供了底层功能。这些类中的大多数都不能直接使用
联网	联网层上的类允许客户端代码异步调用服务器端代码。这一层包含的基本架构可以调用

	URL，响应回调函数的结果。在大多数情况下，这些功能都不能直接使用，而应使用封装了该功能的类。这一层还包含用于 JSON 串行化和并行化的类，大多数联网类都在客户端的 System.Net 命名空间中
用户界面	这一层包含的类抽象了用户界面元素，如 HTML 元素和 DOM 事件。可以使用这一层的方法和属性编写独立于语言的 JavaScript 代码，来处理客户机上的 Web 页面。用户界面类包含在 System.UI 命名空间中
控件	AJAX 库的最后一层包含最高级代码，它们提供了 Ajax 操作和服务器控件功能。这包括动态生成代码，用于在客户端的 JavaScript 代码中调用 Web 服务

AJAX 库可以用于扩展和定制支持 ASP.NET AJAX 的 Web 应用程序的操作，但注意，不一定要这么做。要想在应用程序中不使用任何附加的 JavaScript，还有很长的路要走，只有需要更高级的功能，才需要这么做。如果要编写附带的客户端代码，使用 AJAX 库提供的功能会比较容易完成任务。

### 39.2.2 ASP.NET AJAX Control Toolkit

AJAX Control Toolkit 是附加服务器控件的一个集合，包括由 ASP.NET AJAX 团体编写的扩展控件。扩展控件可以在已有的 ASP.NET 服务器控件中添加功能，一般是给它附带一个客户端操作。例如，AJAX Control Toolkit 中的一个扩展器能在文本框中放置“watermark”文本，以扩展文本框控件，当用户还没有在文本框中添加任何内容时，就会显示该文本。这个扩展控件在服务器控件 TextBoxWatermark 中实现。

使用 AJAX Control Toolkit 可以给站点添加许多功能，它们超出了核心下载包的范围。这些控件可以使浏览操作更有趣，也许能为增强 Web 应用程序提供许多新思路。但是，AJAX Control Toolkit 独立于核心下载包，所以这些控件并没有获得与核心下载包中的控件相同的支持。

## 39.3 使用 ASP.NET AJAX

前面介绍了 ASP.NET AJAX 的组件部分，下面就开始探讨如何使用它们增强网站。本节将讨论支持 ASP.NET AJAX 的 Web 应用程序如何工作，如何使用该软件包中的各种功能。首先仔细研究一个简单的应用程序，然后在后续的章节中添加其他功能。

### 39.3.1 ASP.NET AJAX 网站示例

ASP.NET AJAX 模板包含了 ASP.NET AJAX 的所有核心功能。也可以使用 AJAX Control Toolkit Web Site 模板，以包含 AJAX Control Toolkit 中的控件。本示例要在 C:\ProCSharp\Chapter39 目录中创建一个使用默认 ASP.NET Web Site 模板的新网站 PCSAjaxWebApp1。

修改 Default.aspx 中的代码：

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
```

```
<title>Pro C# ASP.NET AJAX Sample</title>
</head>
<body>
<form id="form1" runat="server">
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<div>
<h1>Pro C# ASP.NET AJAX Sample</h1>
This sample obtains a list of primes up to a maximum value.
<br />
Maximum:
<asp:TextBox runat="server" id=".MaxValue" Text="2500" />
<br />
Result:
<asp:UpdatePanel runat="server" ID="ResultPanel">
<ContentTemplate>
<asp:Button runat="server" ID="GoButton" Text="Calculate " />
<br />
<asp:Label runat="server" ID="ResultLabel" />
<br />
<small>Panel render time: <%
=DateTime.Now.ToString() %></small>
</ContentTemplate>
</asp:UpdatePanel>
<asp:UpdateProgress runat="server" ID="UpdateProgress1">
<ProgressTemplate>
<div style="position: absolute; left: 100px; top: 200px;
padding: 40px 60px 40px 60px; background-color: lightyellow;
border: black 1px solid; font-weight: bold; font-size: larger;
filter: alpha(opacity=80); ">Updating...</div>
</ProgressTemplate>
</asp:UpdateProgress>
<small>Page render time: <% =DateTime.Now.ToString()
%></small>
</div>
</form>
</body>
</html>
```

切换到设计视图(注意 ASP.NET AJAX 控件，如 UpdatePanel 和 UpdateProgress，有可视化的设计组件)，双击 Calculate 按钮，添加一个事件处理程序。修改代码，如下所示：

```
protected void GoButton_Click(object sender, EventArgs e)
{
    int maxValue = 0;
    System.Text.StringBuilder resultText =
```

```
new System.Text.StringBuilder();
if (int.TryParse(MaxValue.Text, out maxValue))
{
    for (int trial = 2; trial <= maxValue; trial++)
    {
        bool isPrime = true;
        for (int divisor = 2; divisor <= Math.Sqrt(trial);
        divisor++)
        {
            if (trial % divisor == 0)
            {
                isPrime = false;
                break;
            }
        }
        if (isPrime)
        {
            resultText.AppendFormat("{0} ", trial);
        }
    }
}
else
{
    resultText.Append("Unable to parse maximum value.");
}
ResultLabel.Text = resultText.ToString();
}
```

保存修改的内容，按下 F5，运行项目。如果有提示，就在 Web.config 中启动调试功能。

在显示如图 39-2 所示的 Web 页面时，注意两个显示时间是相同的。

(点击查看大图) 图 39-2

单击 Calculate 按钮，显示小于等于 2500 的素数。除非在较慢的机器上运行，否则应立即得到结果。注意显示时间现在已经不同了，只有 UpdatePanel 中的显示时间改变了，如图 39-3 所示。

(点击查看大图) 图 39-3

最后，在最大值中添加一些 0，引入一个处理延迟(在较快的 PC 上添加三个 0 就足够了)，再次单击 Calculate 按钮。这次在显示结果之前，注意 UpdateProgress 控件显示一个部分透明的反馈消息，如图 39-4 所示。

(点击查看大图) 图 39-4

更新应用程序时，页面仍是可以响应的。例如，可以滚动页面。

提示：

更新完成时，浏览器的滚动位置设置为单击 Calculate 按钮之前的地方。在大多数情况下，部分页面的更新会很快执行完，这非常有利于可用性。

关闭浏览器，返回 Visual Studio。

### 39.3.2 支持 ASP.NET AJAX 的网站配置

学习了一个简单的支持 ASP.NET AJAX 的 Web 应用程序之后，就可以研究它的工作原理了。首先看看应用程序的 Web.config 文件，尤其是<configuration>的<system.web>配置段中的如下两个代码块：

```
< ?xml version="1.0"? >
< configuration >
...
< system.web >
< compilation debug="true" >
< assemblies >
< add assembly="System.Core, Version=3.5.0.0,
Culture=neutral,
PublicKeyToken=B77A5C561934E089" / >
< add assembly="System.Web.Extensions, Version=3.5.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" / >
< add assembly="System.Data.DataSetExtensions,
Version=3.5.0.0,
Culture=neutral, PublicKeyToken=B77A5C561934E089" / >
< add assembly="System.Xml.Linq, Version=3.5.0.0,
Culture=neutral,
PublicKeyToken=B77A5C561934E089" / >
< /assemblies >
< /compilation >
...
< compilation debug="true" >
< pages >
< controls >
< add tagPrefix="asp" namespace="System.Web.UI"
assembly="System.Web.Extensions, Version=3.5.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" / >
< add tagPrefix="asp"
namespace="System.Web.UI.WebControls"
assembly="System.Web.Extensions, Version=3.5.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" / >
< /controls >
< /pages >
< /compilation >
```

```
...
</system.web>
...
</configuration>
```

<compilation>中<assemblies>配置段的代码确保，ASP.NET AJAX 程序集 System.Web.Extensions.dll 从 GAC 中加载。<pages>中<controls>配置元素的代码引用这个程序集，将它包含的控件（在 System.Web.UI 和 System.Web.UI.WebControls 命名空间中）关联到标记前缀 asp 上。这两个配置段对于所有支持 ASP.NET AJAX 的 Web 应用程序都是必需的。

下面的两段<httpHandlers>和<httpModules>也是 ASP.NET AJAX 功能所需要的。<httpHandlers>段定义了三项内容：第一，Web 服务.asmx 的处理程序用 System.Web.Extensions 命名空间中的一个新类替代。这个新类可以通过 AJAX 库处理来自客户端调用的请求，包括 JSON 串行化和并行化。第二，添加一个处理程序，以使用 ASP.NET 应用程序服务。第三，给 ScriptResource.axd 资源添加一个新的处理程序。这个资源用于 ASP.NET AJAX 程序集中的 AJAX 库 JavaScript 文件，这样这些文件就不需要直接包含在应用程序中了。

```
<system.web>
...
<httpHandlers>
<remove verb="*" path="*.asmx" />
<add verb="*" path="*.asmx" validate="false"
      type="System.Web.Services.ScriptHandlerFactory,
      System.Web.Extensions, Version=1.0.61025.0,
      Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
<add verb="*" path="*_AppService.axd" validate="false"
      type="System.Web.Services.ScriptHandlerFactory,
      System.Web.Extensions, Version=1.0.61025.0,
      Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
<add verb="GET,HEAD" path="ScriptResource.axd"
      type="System.Web.Handlers.ScriptResourceHandler,
      System.Web.Extensions,
      Version=1.0.61025.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" validate="false" />
</httpHandlers>
...
</system.web>
```

<httpModules>段添加了一个新的 HTTP 模块，它在 Web 应用程序中添加了 HTTP 请求的其他处理代码。这将支持部分页面的回送。

```
<system.web>
...
```

```
<httpModules>
<add name="ScriptModule"
type="System.Web.Handlers.ScriptModule,
System.Web.Extensions,
Version=1.0.61025.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</httpModules>
</system.web>
```

其他的配置设置是通过<configSections>设置来确定的，<configSections>设置是<configuration>的第一个子元素。这一段这里没有列出，它必须包含进来，以便使用<system.web.extensions>和<system.webServer>段。

提示：

<system.web.extensions>段没有包含在默认的 ASP.NET Web Site 配置文件中，详见下一节。

下一个配置段<system.webServer>包含的设置与 IIS 7 Web 服务器相关 如果使用 IIS 的早期版本，就不需要这一段。这里没有列出这一段。

最后是一个< runtime >段：

```
< runtime >
< assemblyBinding
xmlns="urn:schemas-microsoft-com:asm.v1" >
< dependentAssembly >
< assemblyIdentity name="System.Web.Extensions"
publicKeyToken="31bf3856ad364e35" / >
< bindingRedirect oldVersion="1.0.0.0-1.1.0.0"
newVersion="3.5.0.0" / >
< /dependentAssembly >
< dependentAssembly >
< assemblyIdentity name="System.Web.Extensions.Design"
publicKeyToken="31bf3856ad364e35" / >
< bindingRedirect oldVersion="1.0.0.0-1.1.0.0"
newVersion="3.5.0.0" / >
< /dependentAssembly >
< /assemblyBinding >
< /runtime >
```

包含这段是为了确保与 ASP.NET AJAX 的旧版本兼容，除非安装了 ASP.NET AJAX 1.0 版本，否则它不会有影响。如果安装了 1.0 版本，这段就会启动第三方控件，绑定到 ASP.NET AJAX 的最新版本上。

## 1. 其他配置选项

<system.web.extensions>段包含的设置为 ASP.NET AJAX 提供了其他配置 ,这些配置都是可选的。在默认的 ASP.NET Web 应用程序模板中不包含它们 ,可以用这个配置段添加的大多数配置都与 Web 服务相关 ,包含在<webServices>元素中 ,该元素放在< scripting >元素中。首先 ,可以添加一段 ,通过 Web 服务访问 ASP.NET 身份验证访问(也可以选择强制 SSL) :

```
< system.web.extensions >
< scripting >
< webServices >
< authenticationService enabled="true" requireSSL =
"true|false"/ >
```

接下来 ,通过 profile Web 服务 ,启用和配置对 ASP.NET 个性化功能的访问。

```
< profileService enabled="true"
readAccessProperties="propertynam1,propertynam2"
writeAccessProperties="propertynam1,propertynam2" / >
```

最后一个与 Web 服务相关的设置是通过角色 Web 服务启用和配置对 ASP.NET 角色功能的访问。

```
< roleService enabled="true" / >
< /webServices >
```

最后 ,<system.web.extensions>段包含一个元素 ,允许配置异步通信的压缩和缓存 :

```
<scriptResourceHandler enableCompression="true"
enableCaching="true" />
</scripting>
</system.web.extensions>
```

## 2. AJAX Control Toolkit 的其他配置

要使用 AJAX Control Toolkit 中的控件 ,可以在 web.config 中添加如下配置 :

```
< controls >
...
< add namespace="AjaxControlToolkit"
assembly="AjaxControlToolkit"
tagPrefix="ajaxToolkit" / >
< /controls >
```

这将工具集中的控件映射到 ajaxToolkit 标记前缀上。这些控件包含在 AjaxControl- Toolkit.dll 程序集中 ,该程序集在 Web 应用程序的/bin 目录下。

还可以使用<%@ Register %>指令在 Web 页面上注册控件。

```
< %@ Register Assembly="AjaxControlToolkit"
Namespace="AjaxControlToolkit"
```

```
TagPrefix="ajaxToolkit" %>
```

### 39.3.3 添加 ASP.NET AJAX 功能

一旦通过网站模板将网站配置为使用 ASP.NET AJAX，或手工配置新 ASP.NET 网站或已有的 ASP.NET 网站，

在网站上添加 AJAX 功能的第一步是在 Web 页面上添加一个 ScriptManager 控件，之后，添加 UpdatePanel 等服务器控件，以启用部分页面的显示功能，再添加 Futures CTP 和 AJAX Control Toolkit 中的动态控件，给应用程序增加可用性和功能。还可以添加客户端代码，使用 AJAX 库进一步定制和增强应用程序的功能。

本节介绍可以使用服务器控件添加的功能。本章的后面将讨论客户端技术。

#### 1. ScriptManager 控件

如本章前面所述，ScriptManager 控件必须包含在使用部分页面回送和其他几个 ASP.NET AJAX 功能的所有页面上。

提示：

为了确保在 Web 应用程序的所有页面上都包含 ScriptManager 控件，必须将这个控件添加到应用程序使用的 master 页面上。

除了启用 ASP.NET AJAX 功能之外，还可以使用属性配置这个控件。在这些属性中，最简单的是 EnablePartialRendering，其默认值是 true。如果把这个属性设置为 false，就禁用了所有异步回送处理功能，例如 UpdatePanel 控件提供的页面回送功能。如果要给经理做一个演示，比较支持 AJAX 的网站和传统的网站，就可以这么做。

使用 ScriptManager 控件有几个原因，例如下面的情形：

确定是否把调用服务器端代码作为部分页面回送的结果

添加对其他客户端 JavaScript 文件的引用

引用 Web 服务

给客户返回错误消息

下面几节介绍这些配置选项。

#### (1) 检测部分页面的回送

ScriptManager 控件包含一个布尔属性 IsInAsyncPostBack。可以在服务器端代码中使用这个属性，检测部分页面是否正在回送。注意 ScriptManager 控件可能在 master 页面上。除了通过 master 页面访问这个控件之外，还可以使用静态方法 GetCurrent()，获得对当前 ScriptManager 实例的引用。例如：

```
ScriptManager scriptManager =
```

```
ScriptManager.GetCurrent(this);
if (scriptManager != null &&
scriptManager.IsInAsyncPostBack)
{
// Code to execute for partial-page postbacks.
}
```

必须将对 Page 控件的引用传送给 GetCurrent()方法。例如，如果在 ASP.NET Web 页面的 Page\_Load()事件处理程序中使用这个方法，就可以将 this 用作 Page 引用。另外，注意检查 null 引用，以避免异常。

## (2) 客户端 JavaScript 引用

除了在 HTML 页面的标题或页面的<script>元素中添加代码之外，还可以使用 ScriptManager 类的 Scripts 属性。这可以使脚本引用集中在一起，更便于维护它们。为此，可以给<UpdatePanel>控件元素添加一个<Scripts>子元素，再给<Scripts>添加<asp: ScriptReference>子控件元素。使用 ScriptReference 控件的 Path 属性引用定制脚本。

下面的例子说明了如何在 Web 应用程序的根文件夹下，添加对一个定制脚本文件 MyScript.js 的引用：

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
<Scripts>
<asp:ScriptReference Path("~/MyScript.js" />
</Scripts>
</asp:ScriptManager>
```

## (3) Web 服务引用

为了从客户端 JavaScript 代码中访问 Web 服务，ASP.NET AJAX 必须生成一个代理类。要控制这个操作，可以使用 ScriptManager 类的 Services 属性。与 Scripts 属性一样，也可以以声明方式指定这个属性，但这次要使用<Services>元素。给这个元素添加<asp: ServiceReference>控件。对于 Services 属性中的每个 ScriptReference 对象，都需要使用 Path 属性指定 Web 服务的路径。

ServiceReference 类也有一个 InlineScript 属性，它默认为 false。这个属性是 false 时，客户端代码向服务器发出请求，会得到一个代理类，来调用 Web 服务。为了改进性能(尤其是在一个页面上使用大量 Web 服务的情况)，可以将 InlineScript 设置为 true，这会在页面的客户端脚本中定义代理类。

ASP.NET Web 服务的文件扩展名是.asmx。如果不详细阅读本章，但希望在 Web 应用程序的根文件夹下添加对 Web 服务 MyService.asmx 的引用，应使用下面的代码：

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
<Services>
<asp:ServiceReference Path "~/MyService.asmx" />
</Services>
```

```
</asp:ScriptManager>
```

采用这种方式只能添加对本地 Web 服务的引用(即 Web 服务和调用代码在同一个 Web 应用程序中)。可以通过本地 Web 方法间接调用远程 Web 服务。

本章的后面将讨论如何从客户端 JavaScript 代码中异步调用 Web 方法 ,这些方法是以这种方式使用代理类生成的。

#### (4) 客户端错误消息

如果在部分页面的回送过程中抛出了异常 ,默认操作是将异常包含的错误消息放在客户端 JavaScript 警报消息框中。处理 ScriptManager 实例的 AsyncPostBackError 事件 ,可以定制要显示的消息。在这个事件的处理程序中 ,可以使用 AsyncPostBackEventArgs.Exception 属性访问抛出的异常 ,使用 ScriptManager.AsyncPostBackErrorMessage 属性设置显示给客户端的消息。这么做可以给用户隐藏异常细节。

如果要重写默认操作 ,以另一种方式显示消息 ,就必须使用 JavaScript 处理客户端对象 PageRequestManager 的 endRequest 事件 ,详见本章后面的内容。

## 2. 使用 UpdatePanel 控件

UpdatePanel 控件是编写支持 ASP.NET AJAX 的 Web 应用程序时最常用的控件。如本章前面的简单例子所述 ,这个控件可以封装 Web 页面的一部分 ,使之参与部分页面的回送操作。为此 ,要在页面上添加一个 UpdatePanel 控件 ,用需要的控件填充其子元素<ContentTemplate>。

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  <ContentTemplate>
    ...
  </ContentTemplate>
</asp:UpdatePanel>
```

根据 UpdatePanel 控件的 RenderMode 属性值 ,<ContentTemplate>模板的内容显示在<div>或<span>元素中。这个属性的默认值是 Block ,即显示在<div>元素中。要使用<span>元素 ,应将 RenderMode 属性设置为 Inline。

#### (1) 一个 Web 页面上的多个 UpdatePanel 控件

可以在一个页面上包含任意多个 UpdatePanel 控件。如果回送操作是由包含在页面上的任一个 UpdatePanel 控件的<ContentTemplate>模板中的控件引发 ,就进行部分页面的回送 ,而不是整个页面的回送。这会使所有的 UpdatePanel 控件根据其 UpdateMode 属性值进行更新。这个属性的默认值是 Always ,表示 UpdatePanel 为页面上的部分页面回送操作而更新 ,即使这个操作是由另一个 UpdatePanel 控件引发的 ,也是如此。如果把这个属性设置为 Conditional ,UpdatePanel 就仅在它包含的控件引发部分页面回送操作时更新 ,或者在启动了已定义的触发器时更新。触发器稍后介绍。

如果把 UpdateMode 属性设置为 Conditional , 还可以将 ChildrenAsTriggers 属性设置为 false , 禁止 UpdatePanel 包含的控件触发 UpdatePanel 的更新操作。但要注意 , 在这种情况下 , 这些控件仍会触发一个部分页面回送操作 , 它会使页面上的其他 UpdatePanel 进行更新。例如 , 这会使 UpdateMode 属性值为 Always 的 UpdatePanel 进行更新 , 如下面的代码所示 :

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  UpdateMode="Conditional"
  ChildrenAsTriggers="false">
    <ContentTemplate>
      <asp:Button runat="Server" ID="Button1" Text="Click Me" />
      <small>Panel 1 render time: <%
        =DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss") %></small>
    </ContentTemplate>
  </asp:UpdatePanel>
  <asp:UpdatePanel runat="Server" ID="UpdatePanel2">
    <ContentTemplate>
      <small>Panel 2 render time: <%
        =DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss") %></small>
    </ContentTemplate>
  </asp:UpdatePanel>
  <small>Page render time: <%
    =DateTime.Now.ToString("MM/dd/yyyy HH:mm:ss") %></small>
```

在这段代码中 , UpdatePanel2 控件的 UpdateMode 属性值设置为默认的 Always 。单击按钮时 , 会引发一个部分页面回送操作 , 但只更新 UpdatePanel2 。注意只更新了 "Panel2 render time" 标签。

## (2) 服务器端的 UpdatePanel 更新

有时页面上有多个 UpdatePanel 控件 , 可以不更新其中的一个 , 除非满足某些条件。在这种情况下 , 应将 UpdatePanel 的 UpdateMode 属性设置为 Conditional , 如上一节所述 , 再把 ChildrenAsTriggers 属性设置为 false 。接着 , 对于页面上引发部分页面回送操作的控件 , 在服务器端的事件处理程序中 , (有条件地)调用 UpdatePanel 的 Update() 方法 , 例如 :

```
protected void Button1_Click(object sender, EventArgs e)
{
  if (TestSomeCondition())
  {
    UpdatePanel1.Update();
  }
}
```

## (3) UpdatePanel 的触发器

给 Web 页面上其他地方的控件的 Triggers 属性添加触发器 , 就可以通过该控件更新 UpdatePanel 控件。触发器是 Web 页面上其他地方的控件的事件与 UpdatePanel 控件之间的关联。所有的控件都有

默认事件(如按钮控件的默认事件是 Click) , 所以可以不指定事件名。有两种触发器可以添加 , 它们用两个类表示 :

AsyncPostBackTrigger : 这个类会在指定控件的指定事件发生时 , 更新 UpdatePanel 控件。

PostBackTrigger : 这个类会在指定控件的指定事件发生时 , 更新整个页面。

一般使用 AsyncPostBackTrigger , 但如果希望 UpdatePanel 中的一个控件引发整个页面的回送操作 , 就可以使用 PostBackTrigger。

这两个触发器类有两个属性 ControlID 和 EventName , ControlID 指定了通过其标识符启动触发器的控件 , EventName 指定了控件中链接到触发器上的事件名。

为了扩展前面的例子 , 考虑下面的代码 :

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  UpdateMode="Conditional"
  ChildrenAsTriggers="false">
    <Triggers>
      <asp:AsyncPostBackTrigger ControlID="Button2" />
    </Triggers>
    <ContentTemplate>
      <asp:Button runat="Server" ID="Button1" Text="Click Me" />
      <small>Panel 1 render time: <%
        =DateTime.Now.ToString() %></small>
    </ContentTemplate>
  </asp:UpdatePanel>
  <asp:UpdatePanel runat="Server" ID="UpdatePanel2">
    <ContentTemplate>
      <asp:Button runat="Server" ID="Button2" Text="Click Me" />
      <small>Panel 2 render time: <%
        =DateTime.Now.ToString() %></small>
    </ContentTemplate>
  </asp:UpdatePanel>
  <small>Page render time: <%
    =DateTime.Now.ToString() %></small>
```

新的按钮控件 Button2 指定为 UpdatePanel1 中的一个触发器。单击这个按钮时 , 会更新 UpdatePanel1 和 UpdatePanel2。更新 UpdatePanel1 是因为启动了触发器 , 更新 UpdatePanel2 是因为它使用了 UpdateMode 的默认值 Always。

### 3. 使用 UpdateProgress

如前面的例子所示 , UpdateProgress 控件允许在部分页面的回送过程中给用户显示进度消息。使用 ProgressTemplate 属性可提供显示进度的模板 , 为此 , 一般使用控件的<ProgressTemplate>元素。

使用 AssociatedUpdatePanelID 属性将 UpdateProgress 控件与指定的 UpdatePanel 关联起来，就可以在页面上放置多个 UpdateProgress 控件。如果没有设置该属性(默认)，无论哪个 UpdatePanel 引发了部分页面回送操作，都显示 UpdateProgress 模板。

在执行部分页面回送操作时，显示 UpdateProgress 模板之前有一个延迟。这个延迟可以通过 DisplayAfter 属性来配置，DisplayAfter 是一个 int 属性，指定了延迟时间(单位是毫秒)，默认为 500 毫秒。

最后，可以使用布尔属性 DynamicLayout 指定在显示模板之前，是否为模板分配空间。这个属性的默认值是 true，此时页面上的空间是动态分配的，所以为了在线显示进度模板，需要删除其他控件。如果把这个属性设置为 false，就在显示模板之前，为模板分配空间，这样页面上其他控件的布局就不会改变。可以根据显示进度时要达到的效果设置这个属性。对于使用绝对坐标定位的进度模板，如前面的例子所示，应将这个属性设置为默认值。

#### 4. 使用扩展器控件

ASP.NET AJAX 的核心软件包包含一个类 ExtenderControl，它的作用是允许扩展其他 ASP.NET 服务器控件(即增加功能)。它广泛应用于 AJAX Control Toolkit，效果不错。可以使用 AJAX Control Toolkit 中的模板创建自己的扩展控件。ExtenderControl 控件的工作方式都是类似的：把它们放在页面上，与目标控件关联起来，添加进一步的配置。接着扩展器就会执行客户端代码，以添加功能。

为了了解扩展控件，在一个简单的例子中创建一个新的网站 PCSExtenderDemo，放在 C:\ProCSharp\Chapter39 目录下，把 AJAX Control Toolkit 程序集添加到网站的 bin 目录下，给 Default.aspx 添加如下代码：

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<%@ Register Assembly="AjaxControlToolkit"
Namespace="AjaxControlToolkit" TagPrefix="ajaxToolkit" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Color Selector</title>
</head>
<body>
<form id="form1" runat="server">
<asp:ScriptManager ID="ScriptManager1" runat="server" />
<div>
<asp:UpdatePanel runat="server" ID="updatePanel1">
<ContentTemplate>
<span style="display: inline-block; padding: 2px;">
My favorite color is:
```

```
</span>

<asp:Label runat="server" ID="favoriteColorLabel"
Text="green"
style="color: #00dd00; display: inline-block; padding:
2px;
width: 70px; font-weight: bold;" />
<ajaxToolkit:DropDownExtender runat="server"
ID="dropDownExtender1"
TargetControlID="favoriteColorLabel"
DropDownControlID="colDropDown" />
<asp:Panel ID="colDropDown" runat="server"
Style="display: none; visibility: hidden; width: 60px;
padding: 8px;
border: double 4px black; background-color: #ffffdd;
font-weight: bold;">
<asp:LinkButton runat="server" ID="OptionRed" Text="red"
OnClick="OnSelect" style="color: #ff0000;" /><br />
<asp:LinkButton runat="server" ID="OptionOrange"
Text="orange"
OnClick="OnSelect" style="color: #dd7700;" /><br />
<asp:LinkButton runat="server" ID="OptionYellow"
Text="yellow"
OnClick="OnSelect" style="color: #ddd000;" /><br />
<asp:LinkButton runat="server" ID="OptionGreen"
Text="green"
OnClick="OnSelect" style="color: #00dd00;" /><br />
<asp:LinkButton runat="server" ID="OptionBlue"
Text="blue"
OnClick="OnSelect" style="color: #0000dd;" /><br />
<asp:LinkButton runat="server" ID="OptionPurple"
Text="purple"
OnClick="OnSelect" style="color: #dd00ff;" />
</asp:Panel>
</ContentTemplate>
</asp:UpdatePanel>
</div>
</form>
</body>
</html>
```

还需要在这个文件的后台代码中添加如下事件处理程序：

```
protected void OnSelect(object sender, EventArgs e)
{
    favoriteColorLabel.Text = ((LinkButton)sender).Text;
    favoriteColorLabel.Style["color"] =
        ((LinkButton)sender).Style["color"];
```

}

在浏览器中，刚开始并没有显示很多内容，扩展器似乎没有什么作用，如图 39-5 所示。

(点击查看大图) 图 39-5

但是，把鼠标停在文本"green"上时，会动态显示一个下拉框。如果单击这个下拉框，就会显示一个列表，如图 39-6 所示。

单击下拉列表中的一个链接时，文本会改变(一个部分页面回送操作之后)。

对于这个简单的例子，要注意两点。第一，非常容易将扩展器与目标控件关联起来。第二，下拉列表用定制代码设置了样式，这表示可以在列表中放置任意内容。这个简单的扩展器是给 Web 应用程序添加功能的有效方式，使用起来也很简单。

(点击查看大图) 图 39-6

AJAX Control Toolkit 中的扩展器在不断增加和更新，所以请定期访问

<http://ajax.asp.net/ajaxtoolkit>。这个 Web 页面包含所有当前的扩展器的实时演示，可以看到它们的工作情况。

除了 AJAX Control Toolkit 提供的扩展器控件之外，还可以创建自己的扩展器控件。为了使这个过程尽可能简单，可以使用项目模板 ASP.NET AJAX Control Project。这个项目包含扩展器需要的所有基本功能，如用于扩展器的服务器端类和扩展器使用的客户端 JavaScript 文件。要创建有效的扩展器，必须使用 AJAX 库。

#### 39.3.4 使用 AJAX 库

AJAX 库有许多可进一步增强 Web 应用程序的功能。但是，为了增强 Web 应用程序，至少需要了解 JavaScript 的基本知识。本节将介绍 AJAX 库提供的一些功能，但这不是一个全面的教程。

使用 AJAX 库的基本规则与在 Web 应用程序中添加任意类型的客户端脚本一样，仍使用核心语言 JavaScript，与 DOM 交互。但是，在许多方面，AJAX 库都使工作更容易完成。本节将学习这些内容，为用户进一步试验 AJAX 库、参考在线 AJAX 库文档打下基础。

本节介绍的技术都在 PCSLibraryDemo 项目中演示，该项目将贯穿本章的剩余内容。

##### 1. 给 Web 页面添加 JavaScript

首先需要了解的是如何给 Web 页面添加客户端 JavaScript，这里有三个选项：

使用<script>元素，在 ASP.NET Web 页面上在线添加 JavaScript

将 JavaScript 添加到单独的 JavaScript 文件中，其扩展名是.js，再使用 ScriptManager 控件的<Scripts>子元素(首选)或从<script>元素中引用这些文件

## 从服务器端代码中生成 JavaScript , 例如后台代码或定制的扩展器控件

这些技术都有自己的优点。对于原型代码 , 在线编码是无可替代的 , 因为它非常快 , 易于使用。将 HTML 元素的客户端事件处理程序和带客户端函数的服务器控件关联起来是很容易的 , 因为所有的代码都在同一个文件中。

使用单独的文件有利于代码重用 , 因为可以创建自己的类库 , 这类似于已有的 AJAX 库 JavaScript 文件。

从后台代码中生成代码较难实现 , 因为我们通常不能像使用 C# 那样在编写 JavaScript 代码时访问 IntelliSense。但是 , 可以动态生成代码 , 以响应应用程序的状态 , 有时这是完成任务的唯一方式。

可以用 AJAX Control Toolkit 创建的扩展器包含一个独立的 JavaScript 文件 , 它用于定义操作 , 解决将客户端代码显示到服务器上的一些问题。

本章将使用在线编码技术 , 因为它最简单 , 允许我们只关注 JavaScript 功能。

### 2. 全局实用函数

AJAX 库提供的一个最常用的特性是封装了其他功能的全局函数集 , 包括 :

\$get() : 这个函数可以获得 DOM 元素的一个引用 , 它将其客户端 id 值提供为一个参数 , 可选的第二个参数指定了要搜索的父元素。

\$create() : 这个函数可以创建指定 JavaScript 类型的对象 , 同时进行初始化。可以给这个函数提供 1 ~ 5 个参数。第一个参数是要实例化的类型 , 它一般是由 AJAX 库定义的一个类型。其他参数分别指定了属性的初始值、事件处理程序、其他组件的引用和对象要关联的 DOM 对象。

\$addHandler() : 这个函数为给对象添加事件处理程序提供了一种缩写方式。

还有更多的全局函数 , 但这些是最常用的全局函数 , 尤其是 \$create() , 它可以大大减少初始化对象所需的代码量。

### 3. 使用 AJAX 库 JavaScript OOP 扩展

AJAX 库包含一个增强的架构 , 它定义了使用基于 OOP 的系统的类型 , 与 .NET Framework 技术紧密相关。可以创建命名空间 , 给命名空间添加类型 , 为类型添加构造函数、方法、属性和事件 , 甚至可以在类型定义上使用继承和接口。

本节将介绍如何使用这个功能的基本内容 , 但这里没有探讨事件和接口。这些结构超出了本章的范围。

#### (1) 定义命名空间

要定义命名空间 , 应使用 Type.registerNamespace() 函数 , 例如 :

```
Type.registerNamespace( "ProCSharp" );
```

注册了命名空间后，就可以给它添加类型了。

## (2) 定义类

定义类需要三步。第一，定义构造函数，第二，添加属性和方法，第三，注册该类。

要定义构造函数，需要使用命名空间和类名来定义一个函数，例如：

```
ProcSharp.Shape = function(color, scaleFactor) {  
    this._color = color;  
    this._scaleFactor = scaleFactor;  
}
```

这个构造函数带两个参数，使用它们设置本地字段(注意不一定要明确定义这些字段，只需设置它们的值)。

要添加属性和方法，应给它们赋予类的 Prototype 属性，如下所示：

```
ProcSharp.Shape.prototype = {  
    getColor : function() {  
        return this._color;  
    },  
    setColor : function(color) {  
        this._color = color;  
    },  
    getScaleFactor : function() {  
        return this._scaleFactor;  
    },  
    setScaleFactor : function(scaleFactor) {  
        this._scaleFactor = scaleFactor;  
    }  
}
```

这段代码提供 get 和 set 存取器定义了两个属性：

要注册类，应调用其 registerClass() 函数：

```
ProcSharp.Shape.registerClass('ProcSharp.Shape');
```

## (3) 继承

派生类的方式与创建类相同，但有一些小区别。在构造函数中使用 initializeBase() 函数初始化基类，以数组的形式传送参数：

```
ProcSharp.Circle = function(color, scaleFactor, diameter)  
{  
    ProcSharp.Circle.initializeBase(this, [color,  
    scaleFactor]);  
    this._diameter = diameter;
```

```
}
```

用前面的方式定义属性和方法：

```
ProCSharp.Circle.prototype = {  
    getDiameter : function() {  
        return this._diameter;  
    },  
    setDiameter : function(diameter) {  
        this._diameter = diameter;  
    },  
    getArea : function() {  
        return Math.PI * Math.pow((this._diameter *  
            this._scaleFactor) / 2, 2);  
    },  
    describe : function() {  
        var description = "This is a " + this._color + " circle  
        with an area of "  
        + this.getArea();  
        alert(description);  
    }  
}
```

注册类时，要把基类型提供为第二个参数：

```
ProCSharp.Circle.registerClass('ProCSharp.Circle',  
    ProCSharp.Shape);
```

将它们传送为其他参数，可以实现接口，但这里为了简单，没有提供其细节。

#### (4) 使用用户定义的类型

以这种方式定义了类之后，就可以通过简单的语法实例化和使用它们了。例如：

```
var myCircle = new ProCSharp.Circle('red', 1.0, 4.4);  
myCircle.describe();
```

这段代码会显示一个 JavaScript 警报框，如-7 所示。

(点击查看大图) 图 39-7

如果要测试一下，可以运行 PCSLibraryDemo 项目，单击 Text OOP Functionality 按钮。

#### 4. PageRequestManager 和 Application 对象

在 AJAX 库中，最有用的类是 PageRequestManager 和 Application。PageRequestManager 在 Sys.WebForms 命名空间中，Application 在 Sys 命名空间中。对于这两个类，重要的是它们提供的几

个事件可以与 JavaScript 事件处理程序关联起来。这些事件在页面生存期(用于 Application)或部分页面回送过程(用于 PageRequestManager)中非常有趣的点发生，可以在这些关键时刻执行操作。

AJAX 库定义事件处理程序的方式类似于.NET Framework 中的事件处理程序的定义方式。每个事件处理程序都有类似的签名，带两个参数。第一个参数是对生成事件的对象的引用。第二个参数是 Sys.EventArgs 类的一个实例或派生自这个类的一个子类的实例。PageRequestManager 和 Application 提供的许多事件都包括专门的事件变元类，它可用于确定事件的更多信息。表 39-4 按照事件的发生顺序列出了这些事件，先是加载页面，然后启动部分页面的回送操作，最后是关闭页面。

表 39-4

事 件	说 明
Application .init	这个事件在页面的生存期中是第一个发生的，它在加载了所有的 JavaScript 文件之后、创建应用程序中的对象之前发生
Application .load	这个事件在加载并初始化了应用程序中的对象后发生。这个事件经常关联一个事件处理程序，在页面第一次加载时执行操作。也可以为页面上的 pageLoad() 函数提供实现代码，该函数自动定义为这个事件的处理程序。使用 Sys.ApplicationLoadEventArgs 对象传送事件变元，该对象包含 IsPartialLoad 属性，用于确定是否已启动了部分页面的回送操作。用 get_IsPartialLoad() 存取器访问这个属性
PageRequestManager .initializeRequest	这个事件在部分页面的回送操作之前、创建请求对象之前发生。可以使用 Sys.WebForms.InitializeRequestEventArgs 事件变元属性，访问启动回送操作的元素(postBackElement)和底层的请求对象(request)
PageRequestManager .beginRequest	这个事件在部分页面的回送操作之前、创建请求对象之后发生。可以使用 Sys.WebForms.BeginRequestEventArgs 事件变元属性，访问启动回送操作的元素(postBackElement)和底层的请求对象(request)
PageRequestManager .pageLoading	这个事件在部分页面的回送操作之后、后续的处理开始之前发生。这个处理过程可以包含要删除或更新的<div>元素，该元素使用 panelsDeleting 和 panelsUpdating 属性，通过 sys.WebForms.PageLoadingEventArgs 对象来引用
PageRequestManager .pageLoaded	这个事件在部分页面的回送操作之后、处理 UpdatePanel 控件之后发生。这个处理过程可以包含要创建或更新的<div>元素，该元素使用 panelsCreated 和 panelsUpdated 属性，通过 WebForms.PageLoadedEventArgs 对象来引用
PageRequestManager .endRequest	这个事件完成部分页面的回送操作之后发生。传送给事件处理程序的 System.WebForms.EndRequestEventArgs 对象可以检测和处理服务器端错误(使用 error 和 errorHandled 属性)，通过 response 访问响应对象
Application .unload	这个事件在删除应用程序中的对象之前发生，以便执行最后的操作或清理任务

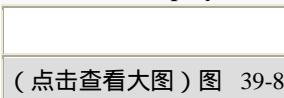
使用静态的 add\_xxx()方法，可以给 Application 对象的事件添加事件处理程序，例如：

```
Sys.Application.add_load(LoadHandler);
function LoadHandler(sender, args)
{
    // Event handler code.
}
```

PageRequestManager 的过程与此类似，但必须使用 get\_instance() 函数获得当前对象的一个实例，例如：

```
Sys.WebForms.PageRequestManager.getInstance().add_beginRequest(  
    BeginRequestHandler);  
  
function BeginRequestHandler(sender, args)  
{  
    // Event handler code.  
}
```

在 PCSLibraryDemo 应用程序中，为 PageRequestManager.endRequest 添加了一个事件处理程序。这个事件处理程序响应服务器端处理的错误，在 id 为 errorDisplay 的元素中显示一个错误消息。要测试这个方法，可以单击 Test Client-Side Error Display 按钮，如图 39-8 所示。



得到该结果的代码如下：

```
Sys.WebForms.PageRequestManager.getInstance().add_endRequest(EndRequestHandler);  
  
function EndRequestHandler(sender, args)  
{  
    if (args.get_error() != undefined)  
    {  
        var errorMessage = args.get_error().message;  
        args.set_errorHandled(true);  
        $get('errorDisplay').innerHTML = errorMessage;  
    }  
}
```

注意 EndRequestEventArgs 对象的 errorHandled 属性设置为 true，这会禁止执行默认操作，即使用 JavaScript 的 alert() 函数在对话框中显示错误消息。

在服务器上抛出一个异常，就生成了错误，如下所示：

```
protected void testErrorDisplay_Click(object sender,  
EventArgs e)  
{  
    throw new ApplicationException(  
        "This is the message set in the exception on the server.");  
}
```

还有许多情形可以使用事件处理技术，来处理 PageRequestManager 和 Application 的事件。

## 5. JavaScript 的调试

JavaScript 很难调试是出了名的。但这在 VS 的最新版本中得到了解决。现在可以像 C# 代码那样在 JavaScript 代码中添加断点，单步执行代码了。还可以在中断模式下查询典型状态，改变属性值等。编写 JavaScript 代码时使用的 IntelliSense 功能也在 VS 的最新版本中得到了很大改进。

有时还希望添加调试和跟踪代码，在代码执行过程中报告信息，例如使用 JavaScript 的 alert() 函数在对话框中显示信息。

有一些第三方工具可用于添加调试的客户端 UI，包括：

Fiddler：这个工具可以从 [www.fiddlertool.com](http://www.fiddlertool.com) 上获得，它可以记录计算机和 Web 应用程序之间的所有 HTTP 通信，包括部分页面的回送。还有一些工具可以查看在处理 Web 页面的过程中发生的事件的详细信息。

Nikhil 的 Web Development Helper：这个工具可以从 <http://projects.nikhilk.net/Projects/WebDevHelper.aspx> 上获得，它也可以记录 HTTP 通信。另外，这个工具包含许多专门用于 ASP.NET 和 ASP.NET AJAX 开发的实用程序，例如，可以查看视图状态，执行即时的 JavaScript 代码。后者特别适合于测试在客户机上创建的对象。Web Development Helper 还在发生 JavaScript 错误时显示其他错误信息，更便于跟踪 JavaScript 代码中的错误。

AJAX 库也提供了 Sys.Debug 类，给应用程序添加额外的调试特性。该类的一个最有用的特性是 Sys.Debug.traceDump() 函数，它可以分析对象，使用这个函数的一种方式是将一个 id 为 TraceConsole 的 textarea 控件放在 Web 页面上，接着，Debug 的所有输出就会发送到这个控件上。例如，可以使用 traceDump() 方法，将 Application 对象的信息输出到控制台上：

```
Sys.Application.add_load(LoadHandler);
function LoadHandler(sender, args)
{
    Sys.Debug.traceDump(sender);
}
```

这会得到如下输出：

```
traceDump {Sys._Application}
_updating: false
_id: null
_disposing: false
_creatingComponents: false
_disposableObjects {Array}
_components {Object}
_createdComponents {Array}
_secondPassComponents {Array}
_loadHandlerDelegate: null
_events {Sys.EventHandlerList}
_list {Object}
load {Array}
[0] {Function}
_initialized: true
_initializing: true
```

在这个输出中，可以看到该对象的所有属性。这个属性特别适合于 ASP.NET AJAX 开发。

## 6. 异步调用 Web 方法

ASP.NET AJAX 的一个最强大的特性是从客户端脚本中调用 Web 方法，这就允许访问数据、服务器端处理和其他功能。

本书的第 42 章介绍了 Web 方法，所以这里不详细介绍它。但是要讨论一些基本知识。简言之，Web 方法是可以在 Web 服务中提供，能通过 Internet 访问远程资源的方法。在 ASP.NET AJAX 中，还可以将 Web 方法用作服务器端 Web 页面的后台代码中的静态方法。在 Web 方法中使用参数和返回值的方式与其他方法类型相同。

在 ASP.NET AJAX 中，Web 方法是异步调用的。给 Web 方法传递参数，定义一个回调函数，当 Web 方法调用完成时，就会调用这个回调函数。该回调函数用于处理 Web 方法的响应。也可以提供另一个回调函数，以处理调用失败的情况。

在 PCSLibraryDemo 应用程序中，单击 Call Web Method 按钮时，就调用一个 Web 方法，如图 39-9 所示。

(点击查看大图) 图 39-9

在客户端脚本中使用 Web 方法之前，必须生成一个客户端代理类，以进行通信。为此，最简单的方式是在 ScriptManager 控件中，引用包含 Web 方法的 Web 服务的 URL：

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
<Services>
<asp:ServiceReference Path="~/SimpleService.asmx" />
</Services>
</asp:ScriptManager>
```

ASP.NET Web 服务使用扩展名.asmx，如上面的代码所示。为了使用客户端代理访问 Web 服务中的 Web 方法，必须给 Web 服务应用 System.Web.Script.Services.ScriptService 属性。

对于 Web 页面的后台代码中的 Web 方法，不需要这个属性，或者 ScriptManager 中的这个引用，但必须使用静态方法，给方法引用 System.Web.Services.WebMethod 属性。

生成了客户端代理后，就可以通过名称访问 Web 方法了，它定义为类中与 Web 服务同名的一个函数。在 PCSLibraryDemo 中，Web 服务 SimpleService.asmx 的一个 Web 方法是 Multiply()，它对两个 double 参数执行相乘操作。从客户端代码中调用这个方法时，要传送方法需要的两个参数(在例子中，从 HTML <input> 元素中获得)，可以传送一个或两个回调函数引用。如果传送一个引用，这个回调函数就在调用成功返回时使用这个引用。如果传送了两个引用，第二个引用就在 Web 方法失败时使用。

在 PCSLibraryDemo 中，使用了一个回调函数，它提取 Web 方法调用的结果，并将它赋予 id 为 webMethodResult 的<span>元素：

```
function callWebMethod()
{
    SimpleService.Multiply(parseFloat($get('xParam').value),
    parseFloat($get('yParam').value), multiplyCallBack);
}
function multiplyCallBack(result)
{
    $get('webMethodResult').innerHTML = result;
}
```

这个方法非常简单，但演示了从客户端代码中异步调用 Web 服务的便利性。

## 7. ASP.NET 应用程序服务

ASP.NET AJAX 包含 3 个专用的 Web 服务，用于访问 ASP.NET 应用程序服务。这些服务可以通过下面的客户端类来访问：

Sys.Services.AuthenticationService：这个服务包含的方法可以登录或注销用户，或确定用户是否已登录。

Sys.Services.ProfileService：这个服务可以获取和设置当前登录的用户的配置属性。配置属性在应用程序的 Web.config 文件中配置。

Sys.Services.RoleService：这个服务可以确定当前登录的用户的角色成员。

如果使用正确，这些类可以实现响应非常好的用户界面，其中包含身份验证、配置经成员功能。

这些服务超出了本章的范围，但应知道它们，它们很值得研究。

### 39.4 小结

本章介绍了如何使用 ASP.NET AJAX 增强 ASP.NET Web 应用程序。ASP.NET AJAX 包含的丰富功能使 Web 应用程序的响应更好、更动态，大大改进了用户体验。

首先学习了 Ajax 的概念、ASP.NET AJAX 的各个组件及其功能，了解了 AJAX 扩展和 AJAX 库的区别，这些组件联合起来提供 ASP.NET AJAX 核心功能的方式。还探讨了 AJAX Control Toolkit 和 ASP.NET 2.0 AJAX Futures CTP，这些都添加到这个核心功能中。

接着，论述了创建支持 ASP.NET AJAX 的 Web 应用程序的服务器端技术，如何在 ASP.NET Web 应用程序的 web.config 文件中配置 ASP.NET AJAX，如何使用 AJAX 扩展中的各种服务器控件，特别是学习了 ScriptManager、UpdatePanel(和触发器)、UpdateProgress 和扩展器控件，使用这些控件能快速、方便地给 Web 应用程序添加许多功能。

之后研究 AJAX 库。AJAX 库扩展并增强了 JavaScript，提供了许多可以添加到应用程序中的功能，但至少要了解 JavaScript 编程的基本知识。

我们学习了 AJAX 库给 JavaScript 添加的全局函数，如何使用 AJAX 库给 JavaScript 添加的 OOP 扩展，来定义命名空间和类。之后，研究了如何在页面的生存期和部分页面回送的过程中与客户端的事件交互，如何使用其中一个事件 PageRequestManager.endRequest，定制在部分页面回送的过程中发生的服务器错误在 Web 浏览器上的显示方式。

最后，陈述了客户端 Web 方法的调用，如何给这些方法调用使用异步模式，如何编写需要的代码，以调用简单的 Web 方法。还学习了通过 Web 服务访问 ASP.NET 应用程序服务(身份验证、配置和成员)的方式。

希望本章除能使读者对这个新技术感兴趣。Ajax 在 Web 上非常流行，ASP.NET AJAX 是将 Ajax 功能与 ASP.NET 应用程序集成起来的绝佳方式。这个产品也得到了非常好的支持，基于团体的版本，如 AJAX Control Toolkit，提供了更酷的功能，这些功能可以在应用程序中免费使用。

尽管必须学习 JavaScript 语言，但这是值得的。使用 ASP.NET AJAX 比仅使用 ASP.NET 可以使 Web 应用程序更好、功能更强、更动态。在 VS 的最新版本中，有一些使 ASP.NET AJAX 更容易使用的工具。

下一章介绍 Web 开发，学习如何使用 VS 中的代码扩展 Microsoft Office 应用程序，例如 Word、Excel 和 Outlook。

Visual Studio Tools for Office(VSTO)技术可以使用.NET Framework 定制和扩展 Microsoft Office 应用程序和文档 ,它包含的工具还可以使这个定制在 Visual Studio 中更容易完成 ,例如用于 Office ribbon 控件的可视化设计器。

VSTO 是微软公司发布的一系列产品中的最新产品 ,可以定制 Office 应用程序。用于访问 Office 应用程序的对象模型已经随时间逐步演化了。如果读者过去曾使用过它 ,就会熟悉它的某些部分。如果读者以前为 Office 应用程序编写过 VBA 插件 ,就为本章讨论的技术做好了准备(VSTO 可以与 VBA 交互操作)。但 VSTO 通过 Office Primary Interop Assemblies(PIAs)提供的、与 Office 交互的类已经扩展到 Office 对象模型之外。例如 ,VSTO 类包括.NET 数据绑定功能。

在 Visual Studio 2008 推出之前 ,VSTO 一直是一个独立下载的软件包 ,如果要开发 Office 解决方案 ,就可以得到它。在 Visual Studio 2008 中 ,VSTO 集成到 Visual Studio IDE 中。VSTO 的这个版本也称为 VSTO 3 ,包含了对 Office 2007 的全部支持 ,还包括许多新特性 ,例如可以与 Word 内容控件交互 ,前面提及的 ribbon 可视化设计器、VBA 集成等。

本章不需要 VSTO 或其以前版本的任何预备知识。内容如下 :

可以用 VSTO 创建的项目类型 ,在这些项目中可以包含的功能

应用于所有 VSTO 解决方案类型的基础技术

如何建立带定制 UI、VBA 交互操作功能和 ClickOnce 部署功能的 VSTO 解决方案

#### 40.1 VSTO 概述

VSTO 包含如下组件 :

一组项目模板 ,可用于创建各种类型的 Office 解决方案

设计器 ,支持 ribbons、动作面板和定制任务面板的可视化布局

建立在 Office Primary Interop Assemblies(PIAs)基础之上的类 ,它们还提供了扩展功能

VSTO 支持 Office 2003 和 2007 版。VSTO 类库有两种形式 ,各用于这两种 Office 版本 ,它们分别使用不同系列的程序集。由于它们比较简单(且功能集很丰富) ,所以本章主要介绍 2007 版。

VSTO 解决方案的一般体系结构如图 40-1 所示。

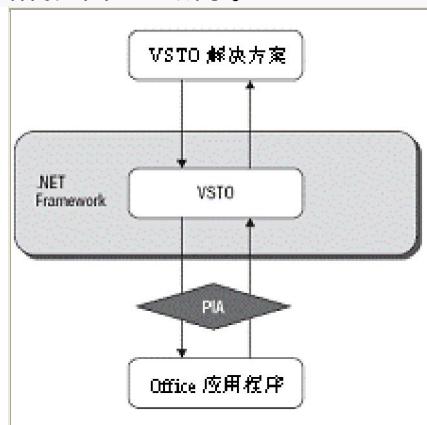


图 40-1

#### 40.1.1 项目类型

图 40-2 显示了 Visual Studio 中的项目模板。

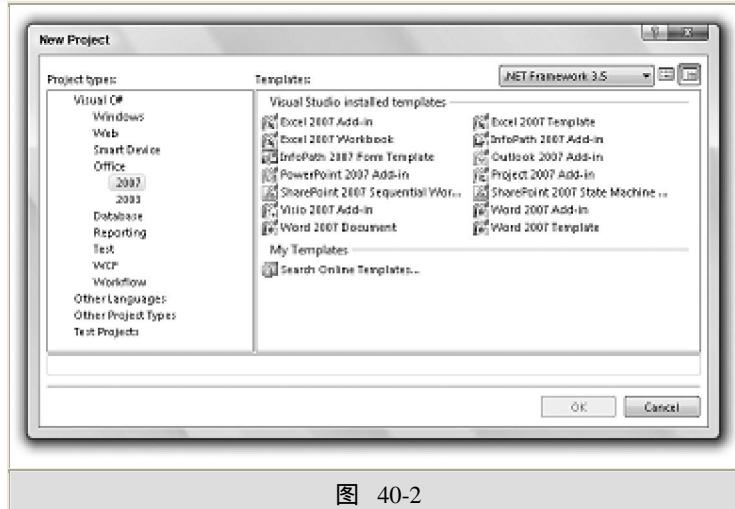


图 40-2

提示：

使用 VSTO 模板创建项目时，需要具备对 VBA 项目系统的访问权限。这是与 VBA 交互所必须的。

VSTO 项目模板可以分为如下类别：

文档级的定制

应用程序级的插件

SharePoint 工作流模板

InfoPath 窗体模板

一些项目类型有 2003 和 2007 版，但这里只介绍 2007 版。

本章主要讨论最常用的项目类型，即文档级的定制和应用程序级的插件。

##### 1. 文档级的定制

创建这种类型的项目时，会生成一个链接到单个文档上的程序集，例如 Word 文档、Word 模板或 Excel 工作簿。加载该文档时，关联的 Office 应用程序会检测到定制，加载程序集，使 VSTO 定制可以使用。

这类项目可以给某个业务线上的文档提供额外的功能，或者在文档模板中添加定制功能，为这类文档添加额外功能。所包含的代码可以操作文档和文档的内容，包括内嵌的对象。还可以提供定制菜单，包括可以用 Visual Studio Ribbon 设计器创建的 ribbon 菜单。

创建文档级的项目时，可以选择创建新文档，或者复制已有的文档，作为开发的起点。也可以选择要创建的文档类型。例如，对于 Word 文档，就可以选择创建.docx(默认)、.doc 或.docm 文档(.docm 是支持宏的文档)。其对话框如图 40-3 所示。

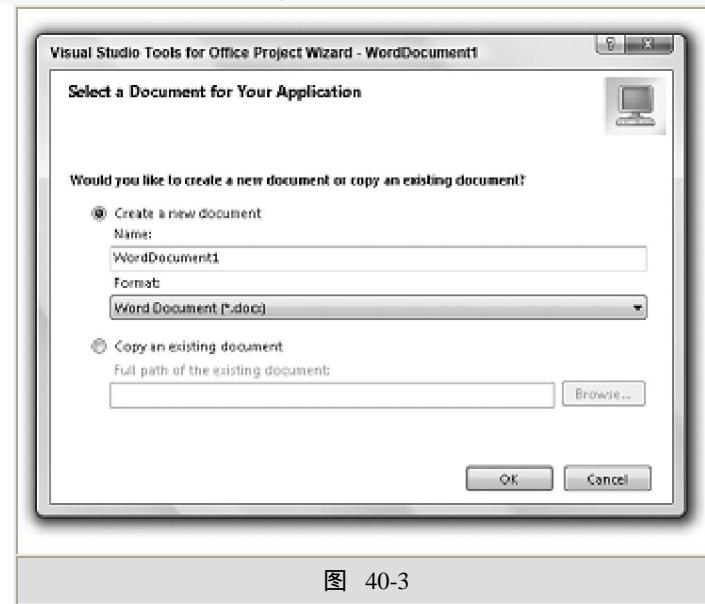


图 40-3

## 2. 应用程序级的插件

应用程序级的插件不同于文档级的定制，因为前者可用于整个目标 Office 应用程序。我们可以访问插件代码，其中可以包含菜单、文档操作等，而无论加载什么文档。

启动某个 Office 应用程序如 Word 时，它会寻找已在注册表中有数据项的关联插件，并加载需要的程序集。

## 3. SharePoint 工作流模板

这些项目提供了创建 SharePoint 工作流应用程序的模板。它们用于管理 SharePoint 进程中的文档流。创建了这类项目后，就可以在文档的生存期中，在重要的时刻执行定制代码。

## 4. InfoPath 窗体模板

这是用于 InfoPath 窗体的文档级定制的一种形式，但它们给 Word 和 Excel 文档定制使用略微不同的方法，所以通常要分为不同的类别。可以为 InfoPath 窗体创建模板，扩展 InfoPath 设计器的功能，为 InfoPath 窗体的设计人员和终端用户提供额外的功能和业务逻辑。

创建 InfoPath 窗体模板时，可以利用向导指定要创建的项目类型，如图 40-4 所示。

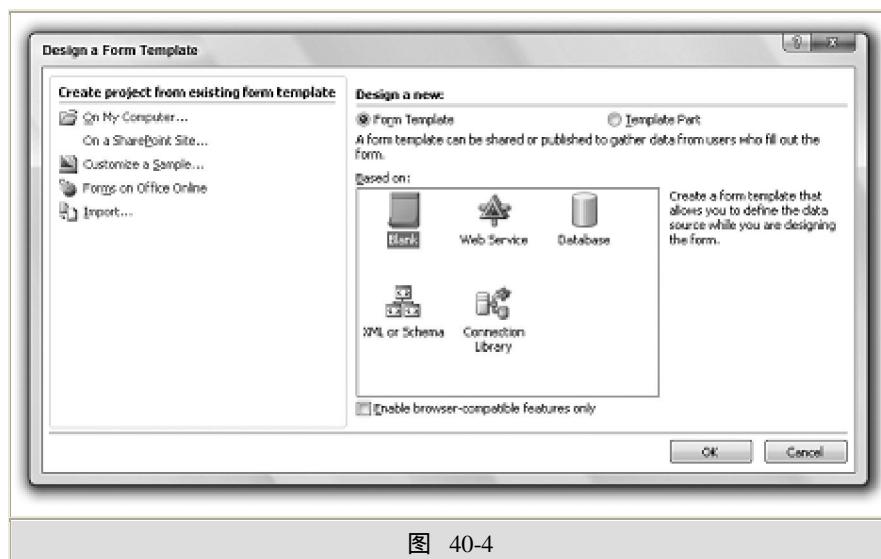


图 40-4

在图 40-4 中，这个向导为所创建的窗体提供了很大的灵活性：可以选择许多不同的起点(包括 SharePoint 站点上的窗体)。还可以创建完整的窗体或模板部分，把功能限制为与浏览器兼容的功能。

#### 40.1.2 项目特性

在各种 VSTO 项目类型中有几个可以使用的特性，例如交互面板和控件。我们使用的项目类型决定了可用的特性。表 40-1 根据项目类型列出了这些特性。

表 40-1

特 性	说 明
动作 面板	动作面板是保存在 Word 或 Excel 的动作面板中的对话框。可以在这里显示任意控件，这是扩展文档和应用程序的一种万能方式
数据 高速 缓存	数据的高速缓存可以在文档外部的高速缓存数据孤岛上存储在文档中使用的数据。这些数据孤岛可以从数据源中更新或手工更新，在数据源脱机或不可用时，允许 Office 文档访问数据
VBA 代码 的端点	如前所述，VSTO 支持与 VBA 的交互操作。在文档级的定制中，可以提供从 VBA 代码中调用的端点方法
主机 控件	主机控件是 Office 对象模型中已有控件的扩展封装器。可以操作这些对象，与它们建立数据绑定
智能 标记	智能标记是嵌入在 Office 文档中、有类型化内容的对象。它们在 Office 文档的内容中自动检测，例如，应用程序检测到相应的文本时，就会自动添加股票报价智能标记。可以创建自己的智能标记类型，定义可以在该标记上执行的操作
可视 化文档 设计器	处理文档定制项目时，要使用 Office 对象模型创建一个可视化的设计界面，以交互式地布置控件。设计器中显示的工具栏和菜单(如本章后面所述)具有全面的功能

#### 应用程序级的插件特性

特 性	说 明
定制任务面板	任务面板一般位于 Office 应用程序的一个边界上，提供了各种功能。例如，Word 的一个任务面板用于操作样式。与动作面板一样，它们也提供了很大的灵活性

跨应用程序的通信	为某个 Office 应用程序创建了插件后，就可以把这个功能提供给其他插件。例如可以在 Excel 中创建一个财务计算服务，再在 Word 中使用该服务——无需创建一个单独的插件
Outlook 窗体区域	可以创建在 Outlook 中使用的窗体区域

### 所有项目类型可用的特性

特    性	说    明
ClickOnce 部署	可以通过 ClickOnce 部署方法把自己创建的任意 VSTO 项目发布给终端用户，让用户检测对应用程序的程序集清单的变化，拥有文档级和应用程序级解决方案的最新版本
Ribbon 菜单	Ribbon 菜单在所有的 Office 应用程序中使用。VSTO 提供了创建定制 ribbon 菜单的两种方式，可以使用 XML 定义 ribbon，也可以使用 Ribbon 设计器，后者更容易使用，但采用 XML 版本可以保证向后兼容性

## 40.2 VSTO 基础

知道了 VSTO 包含的内容，下面该看看 VSTO 的特殊一面了，并学习如何建立 VSTO 项目。本节介绍的技术可以应用于所有的 VSTO 项目类型。

本节介绍如下内容：

Office 对象模型

VSTO 命名空间

主机项和主机控件

基本 VSTO 项目结构

Globals 类

事件处理

### 40.2.1 Office 对象模型

Office 应用程序的 2007 套装通过一个 COM 对象模型提供其功能。可以在 VBA 中直接使用这个对象模型，来控制 Office 功能的任意方面。Office 对象模型在 Office 97 中引入，之后有了许多演变，Office 中的功能也有许多改变。

Office 对象模型有数量巨大的类，其中一些类在 Office 应用程序的套装中使用，一些类专门用于某些应用程序。例如，Word 2007 对象模型包含 Documents 集合，它表示当前加载的对象，每个对象都用一个 Document 对象表示。在 VBA 代码中，可以根据名称或索引访问文档，调用方法对它们执行操作。例如，下面的 VBA 代码关闭了名称为 My Document 的文档，且不保存修改的内容：

```
Documents("My Document").Close  
SaveChanges:=wdDoNotSaveChanges
```

Office 对象模型包含命名的常量(例如上面代码中的 wdDoNotSaveChanges)和枚举，更便于使用。

### 40.2.2 VSTO 命名空间

VSTO 包含一个命名空间集合，该集合包含的类型可用于给 Office 对象模型编写程序。这些命名空间中的许多类和枚举直接映射到 Office 对象模型中的对象和枚举上。它们可以通过 Office PIAs 访问。VSTO 还包含不能直接映射的类型，或者与 Office 对象模型无关的类型。例如，有许多类用于 Visual Studio 中支持的设计器。

封装了 Office 对象模型中的对象或与它们通信的类型分别放在不同的命名空间中，这些命名空间包含了用于 Office 2003 和 2007 的类型。用于 Office 2007 开发的命名空间如表 40-2 所示。

表 40-2

命 名 空 间	说 明
Microsoft.Office.Core	这些命名空间包含 PIA 类的瘦封装器，所以提供了处理 Office 类的基本功能。在 Microsoft.Office.Interop 命名空间中有几个嵌套的命名空间，用于每个 Office 产品
Microsoft.Office.Tools	这个命名空间包含的基本类型提供了 VSTO 功能和用于嵌套命名空间中的许多类的基类。例如，这个命名空间包含了实现文档级定制中的动作面板所需的类，以及应用程序级插件的基类
Microsoft.Office.Tools.Excel	这些命名空间包含的类型用于与 Excel 应用程序和 Excel 文档交互
Microsoft.Office.Tools.Excel.*	
Microsoft.Office.Tools.Outlook	这个命名空间包含的类型用于与 Outlook 应用程序交互
Microsoft.Office.Tools.Ribbon	这个命名空间包含的类型用于处理和创建 Ribbon 菜单
Microsoft.Office.Tools.Word	这些命名空间包含的类型用于与 Word 应用程序和 Word 文档交互
Microsoft.Office.Tools.Word.*	
Microsoft.VisualStudio.Tools.*	这些命名空间提供的 VSTO 基础体系可以在 Visual Studio 中开发 VSTO 解决方案时使用

### 40.2.3 主机项和主机控件

主机项和主机控件是扩展文档级定制的类，使之更容易与 Office 文档交互。这些类简化了代码，因为它们提供了.NET 样式的事件，且进行了全面的管理。主机项和主机控件中的“主机”表示，这些类封装和扩展了通过 PIAs 访问的内部 Office 对象。

在使用主机项和主机控件时，常常需要使用底层的 PIA 交互操作类型。例如，如果创建了一个新的 Word 文档，就会接收到对交互操作 Word 文档类型的引用，而不是 Word 文档主机项。必须注意这一点，并据此编写代码。

Word 和 Excel 文档级定制都有主机项和主机控件。

#### 1. Word

Word 只有一个主机项 Microsoft.Office.Tools.Word.Document。这表示一个 Word 文档。这个类有许多方法和属性，可用于与 Word 文档交互。

Word 有 12 个主机控件，如表 40-3 所示，所有主机控件都在 Microsoft.Office.Tools.Word 命名空间中。

表 40-3

控件	说明
Bookmark	这个控件表示 Word 文档中的一个位置，它可以是单个位置，或一个字符串范围
XMLNode, XMLNodes	文档有一个关联的 XML 模式时使用这两个控件，它们允许通过文档内容的 XML 节点位置来引用文档内容。也可以用这两个控件操作文档的 XML 结构
ContentControl	这个类是本表中剩余 8 个控件的基类，允许处理 Word 内容控件。内容控件把内容表示为控件，或者启动文档中纯文本没有的功能
BuildingBlockGallery-ContentControl	这个控件允许添加和处理文档构造块，例如格式化的表、封面等
ComboBoxContentControl	这个控件表示格式化为组合框的内容
DatePickerContentControl	这个控件表示格式化为日期提取器的内容
DropDownListContentControl	这个控件表示格式化为下拉列表的内容
GroupContentControl	这个控件表示的内容是其他内容项的组合集合，包括文本和其他内容控件
PictureContentControl	这个控件表示一个图像
RichTextContentControl	这个控件表示一大块文本内容
PlainTextContentControl	这个控件表示一大块纯文本内容

## 2. Excel

Excel 有 3 个主机项和 4 个主机控件，它们都包含在 Microsoft.Office.Tools.Excel 命名空间中。

Excel 主机项如表 40-4 所示。

表 40-4

主机项	说明
Workbook	这个主机项表示整个 Excel 工作簿，它可以包含多个工作表和图表
Worksheet	这个主机项用于工作簿中的单个工作表
Chartsheet	这个主机项用于工作簿中的单个图表

Excel 主机控件如表 40-5 所示。

表 40-5

控件	说明
Chart	这个控件表示嵌入到工作表中的图表
ListObject	这个控件表示工作表中的一个列表
NamedRange	这个控件表示工作表中的一个命名区域
XmlMappedRange	这个控件在 Excel 电子表格有关联的模式时使用，用于处理映射到 XML 模式元素上的范围

### 40.2.4 基本的 VSTO 项目结构

第一次创建 VSTO 项目时，系统创建的文件随项目类型的不同而不同，但有一些共同的特性。本节介绍 VSTO 项目的组成。

## 1. 文档级定制的项目结构

创建文档级定制的项目时，在 Solution Explorer 中有一项表示文档类型。它可以是：

表示 Word 文档的.docx 文件

表示 Word 模板的.dotx 文件

表示 Excel 工作簿的.xlsx 文件

表示 Excel 模板的.xltx 文件

每个文档类型都有一个设计器视图和一个代码文件，如果在 Solution Explorer 中展开该项，就会看到它们。Excel 模板还包含子项，它们表示整个工作簿和工作簿中的每个工作表。这个结构可以在每个工作表或工作簿的基础上提供定制功能。

如果查看上述项目类型的隐藏文件，会看到几个设计器文件，查看这些设计器文件，还会看到模板生成的代码。每个 Office 文档项都在 VSTO 命名空间中有关联的类，代码文件中的类派生于这些类。这些类定义为部分类，这样定制代码会与可视化设计器生成的代码分隔开，类似于 Windows 窗体应用程序的结构。

例如，Word 文档模板提供了一个派生自主机项 Microsoft.Office.Tools.Word.Document 的类，其代码包含在 ThisDocument.cs 中，如下所示：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using
Microsoft.VisualStudio.Tools.Applications.Runtime;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;
namespace WordDocument1
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender,
System.EventArgs e)
        {
        }
        private void ThisDocument_Shutdown(object sender,
System.EventArgs e)
        {
        }
    #region VSTO Designer generated code
```

```
/// < summary >
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// < /summary >
private void InternalStartup()
{
    this.Startup += new
        System.EventHandler(ThisDocument_Startup);
    this.Shutdown += new
        System.EventHandler(ThisDocument_Shutdown);
}
#endregion
}
```

这些模板生成的代码包含两个主要命名空间的别名，在为 Word 创建文档级的定制时，需要使用这两个命名空间。Microsoft.Office.Core 用于主要的 VSTO Office 类，Microsoft.Office.Interop.Word 用于和 Word 相关的类。注意如果要使用 Word 主机控件，还要为 Microsoft.Office.Tools.Word 命名空间添加一个 using 语句。模板生成的代码还定义了两个事件处理程序 ThisDocument\_Startup() 和 ThisDocument\_Shutdown()，用于在加载和卸载文档时执行代码。

每个文档级定制项目类型的代码文件都有类似的结构，还定义了命名空间别名以及 VSTO 类中各个 Startup 和 Shutdown 事件的处理程序。以此为起点，可以添加对话框、动作面板、ribbon 控件、事件处理程序和定制代码，来定义定制操作。

在文档级的定制中，还可以通过文档设计器定制文档。根据所创建的解决方案类型，这可能需要给模板添加样板文件，给文档添加交互式内容或其他内容。设计器是 Office 应用程序的高效主机版本，使用它们可以像在应用程序中那样输入内容。还可以在文档中添加控件，例如主机控件和 Windows 窗体控件，以及这些控件的代码。

## 2. 应用程序级插件的项目结构

创建应用程序级插件时，在 Solution Explorer 中没有文档，而有一项表示创建插件所使用的应用程序。如果展开该项，会看到一个文件 ThisAddIn.cs。这个文件包含类 ThisAddIn 的部分定义，该类是插件的入口点。这个类派生于 Microsoft.Office.Tools.AddIn，它提供了编写插件的功能，实现了 Microsoft.VisualStudio.Tools.Office.IOfficeEntryPoint 接口，这是一个基础体系接口。

例如，Word 插件模板生成的代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Word = Microsoft.Office.Interop.Word;
```

```
using Office = Microsoft.Office.Core;
namespace WordAddIn1
{
    public partial class ThisAddIn
    {
        private void ThisAddIn_Startup(object sender,
System.EventArgs e)
        {
        }

        private void ThisAddIn_Shutdown(object sender,
System.EventArgs e)
        {
        }

        #region VSTO generated code
        /// < summary >
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// < /summary >
        private void InternalStartup()
        {
            this.Startup += new
System.EventHandler(ThisAddIn_Startup);
            this.Shutdown += new
System.EventHandler(ThisAddIn_Shutdown);
        }
        #endregion
    }
}
```

可以看出，这个结构非常类似于文档级定制使用的结构，它包含 Microsoft.Office.Core 和 Microsoft.Office.Interop.Word 命名空间的别名，提供了 Startup 和 Shutdown 事件的处理程序 (ThisAddIn\_Startup() 和 ThisAddIn\_Shutdown())。这些事件与文档级定制略有不同，因为它们在加载或卸载插件时触发，而不是在打开或关闭文档时触发。

定制应用程序级插件与文档级定制相同：添加 ribbon 控件、任务面板和其他代码。

#### 40.2.5 Globals 类

所有的 VSTO 项目类型都定义了 Globals 类，它提供了如下内容的全局访问权限：

对于文档级的定制，可以访问解决方案中的所有文档，这是通过其名称匹配文档类名的成员来实现的，例如 Globals.ThisWorkbook 和 Globals.Sheet1。

对于应用程序级的插件，可以访问插件对象。这是通过 Globals.ThisAddIn 实现的。

对于 Outlook 插件项目，可以访问所有的 Outlook 窗体区域。

可以访问解决方案中的所有 ribbon 控件，这是通过 Globals.Ribbons 属性定义的。

在后台，Globals 类是在解决方案的由各种设计器维护的代码文件中通过一系列部分定义来创建的。例如，在 Excel 工作簿项目中，默认的 Sheet1 工作表包含如下设计器生成的代码：

```
internal sealed partial class Globals
{
    private static Sheet1 _Sheet1;
    internal static Sheet1 Sheet1
    {
        get
        {
            return _Sheet1;
        }
        set
        {
            if (_Sheet1 == null)
            {
                _Sheet1 = value;
            }
            else
            {
                throw new System.NotSupportedException();
            }
        }
    }
}
```

这段代码把 Sheet1 成员添加到 Globals 类中。

#### 40.2.6 事件处理

本章前面介绍了主机项和主机控件类如何提供我们可以处理的事件。但交互操作类不是这样。我们只能使用几个事件，大多数事件都很难用于创建事件驱动的解决方案。为了响应事件，我们常常要关注主机项和主机控件提供的事件。

此处一个明显的问题是应用程序级的插件项目没有主机项和主机控件。在使用 VSTO 时，必须面对这个问题。但是，我们在插件中监听的大多数常用事件都关联到 ribbon 菜单和任务面板的交互操作中。我们用集成的 ribbon 设计器设计 ribbon 控件，响应 ribbon 控件生成的事件，使控件可以交互操作。任务面板常常实现为 Windows 窗体用户控件(也可以使用 WPF)，所以这里可以使用 Windows 窗体事件，来代替 PIA 交互操作事件。这表示，我们不会常常遇到如下情形：需要的功能没有可用的事件。

需要使用 PIA 提供的事件时，这些事件是通过 PIA 对象上的接口提供的。考虑一个 Word 插件项目。这个项目中的 ThisAddIn 类有一个属性 Application，利用它可以获得对 Office 应用程序的引用。这个属性的类型是 Microsoft.Office.Interop.Word.Application，它通过

Microsoft.Office.Interop.Word.ApplicationEvents4\_Event 接口提供事件。这个接口共提供了 29 个事件(对于像 Word 这样复杂的应用程序来说并不多)。我们可以处理 DocumentBeforeClose 事件 ,来响应 Word 文档的关闭请求。

### 40.3 建立 VSTO 解决方案

前面几节解释了 VSTO 项目的概念、结构和可以在各种项目类型中使用的特性。本节讨论如何实现 VSTO 解决方案。

图 40-5 列出了文档级定制解决方案的结构。

对于文档级的定制 ,至少要与一个主机项交互操作 ,该主机项一般包含几个主机控件。可以直接使用 Office 对象封装器 ,但在大多数情况下 ,应通过主机项和主机控件访问 Office 对象模型及其功能。

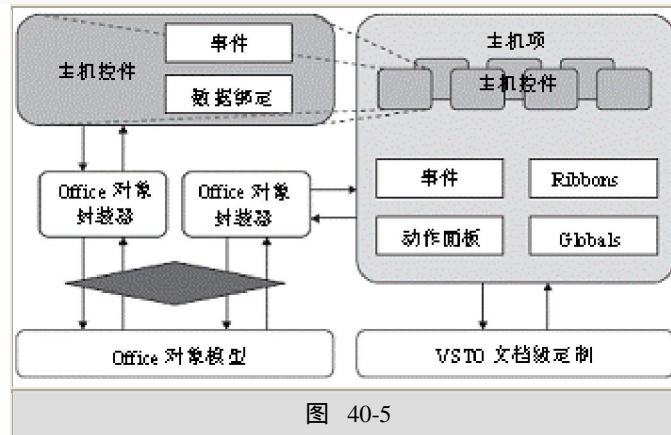


图 40-5

可以在代码中使用主机项和主机控件事件、数据绑定、ribbon 菜单、动作面板和全局对象。

图 40-6 列出了应用程序级插件解决方案的结构。

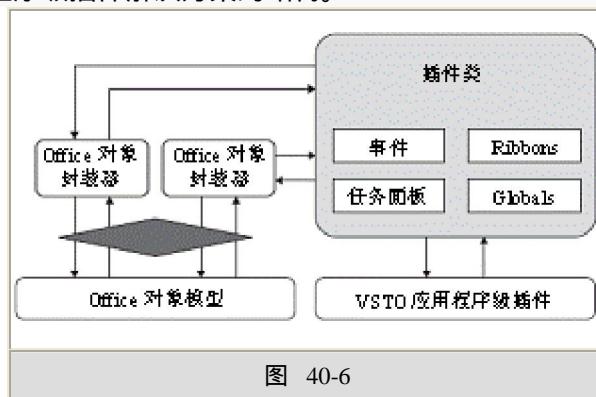


图 40-6

在这个略微简单的模型中 ,很可能要直接使用 Office 对象的瘦封装器 ,或者至少通过封装解决方案的插件类来使用。还要在代码中使用插件类的事件、ribbon 菜单、动作面板和全局对象。

本节介绍这两种应用程序类型以及如下主题 :

管理应用程序级插件

与应用程序和文档交互操作

UI 的定制

### 40.3.1 管理应用程序级插件

在创建应用程序级插件时，会发现 Visual Studio 执行了注册 Office 应用程序的所有步骤。这表示，添加了注册表项，在 Office 应用程序启动时，会自动定位和加载程序集。如果以后要添加或删除插件，就必须浏览 Office 应用程序设置，或者手工操作注册表。

例如，在 Word 中，必须打开 Office Button 菜单，单击 Word Options，选择 Add-Ins 选项卡，如图 40-7 所示。

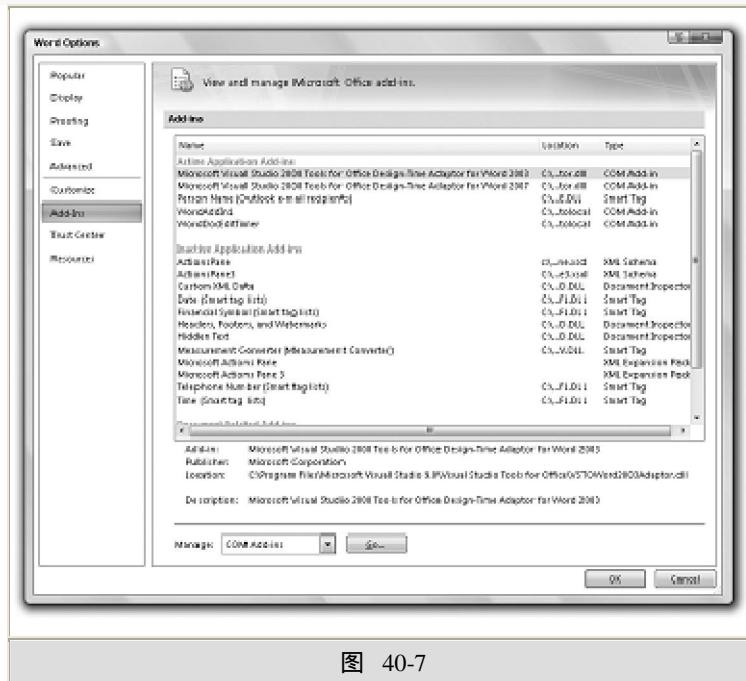


图 40-7

图 40-7 显示，用 VSTO 创建了两个插件：WordAddIn1 和 WordDocEditTimer。要添加或删除插件，必须在 Manage 下拉列表中选择 COM Add-Ins(默认选项)，单击 Go 按钮，打开如图 40-8 所示的对话框。



图 40-8

在 COM Add-Ins 对话框中取消对插件的选择，就会卸载插件，如图 40-8 所示。还可以使用 Add 和 Remove 按钮添加新插件或删除旧插件。

### 40.3.2 与应用程序和文档交互操作

无论创建什么类型的应用程序，都要与主机应用程序和/或主机应用程序中的文档交互操作。这包括使用下一节介绍的 UI 定制功能。还可能需要监控应用程序中的文档，这表示必须处理一些 Office 对象模型事件。例如，要监控 Word 中的文档，需要 Microsoft.Office.

Interop.Word.ApplicationEvents4\_Event 接口中如下事件的处理程序：

DocumentOpen : 打开文档时触发

NewDocument : 创建新文档时触发

DocumentBeforeClose : 保存文档时触发

另外，Word 第一次启动时，会加载一个文档，它可以是空白的新文档，也可以是已加载的旧文档。

提示：

本章的下载代码包含一个示例 WordDocEditTimer，它维护着 Word 文档的一个编辑时次数表。这个应用程序的部分功能是监控已加载的文档，其原因在后面解释。这个示例也使用了定制的任务面板和 ribbon 菜单，所以在介绍完这些主题后，再介绍这个示例。

在 Word 中，可以通过 ThisAddIn.Application.ActivateDocument 属性访问当前活动的文档，通过 ThisAddIn.Application.Documents 属性访问打开的文档集合。由于有了 Multiple Document Interface(MDI)，类似的属性也存在于其他 Office 应用程序中。可以通过 Microsoft.Office.Interop.Word.Document 类的属性操作文档的各个属性。

这里要注意，在开发 VSTO 解决方案时，必须处理的类和类成员的数量是相当大的。除非已经习惯了，否则很难找到需要的特性。例如，在 Word 中，当前活动的选择不是通过活动的文档获得的，而是通过应用程序获得的(利用 ThisAddIn.Application.Selection 属性)。其原因并不是很明显。

通过 Range 属性可以把选择应用于插入、读取或替换文本的操作。例如：

```
ThisAddIn.Application.Selection.Range.Text = "Inserted  
text";
```

但是，本章没有足够的篇幅来详细介绍对象库，读者可以在本章讨论到相关的内容时学习对象库。

#### 40.3.3 UI 的定制

在 VSTO 的最新版本中，最重要的方面是定制 UI 的功能和插件的灵活性。可以给已有的 ribbon 菜单中添加内容，添加全新的 ribbon 菜单，定制动作面板，添加全新的动作面板，集成 Windows 窗体、WPF 窗体和控件。

本节介绍这些主题。

##### 1. Ribbon 菜单

可以在本章介绍的所有 VSTO 项目中添加 ribbon 菜单。添加 ribbon 菜单时，会看到如图 40-9 所示的设计器窗口。



图 40-9

设计器可以给 Office 按钮菜单和 ribbon 菜单上的组添加控件(显示在图 40-9 的左上部)，来定制这个 ribbon 菜单。也可以添加其他组。

ribbon 中使用的类在 Microsoft.Office.Tools.Ribbon 命名空间中。这包括用于创建 ribbon 的派生类 OfficeRibbon。这个类可以包含 RibbonTab 对象，每个 RibbonTab 对象都包含了单个选项卡的内容。选项卡则包含了 RibbonGroup 对象，例如图 40-9 中的 group1 组。这些选项卡都可以包含各种控件。

选项卡上的组可以位于目标 Office 应用程序的一个全新选项卡上，或者位于一个已有的选项卡上。组在何处显示取决于 RibbonTab.ControlId 属性。这个属性有一个 ControlIdType 属性，它可以设置为 RibbonControlIdType.Custom 或 RibbonControlIdType.Office。如果使用 Custom，还必须把 RibbonTab.ControlId.CustomId 设置为 String 值，这是选项卡的标识符。这里可以使用任意标识符。但如果给 ControlIdType 使用 Office，就必须把 RibbonTab.ControlId.OfficeId 设置为一个 String 值，该值匹配在当前 Office 产品中使用的一个标识符。例如，在 Excel 中，可以把这个属性设置为 TabHome，把组添加到 Home 选项卡上，设置为 TabInsert 把组添加到 Insert 选项卡上，等。插件的默认属性是 TabAddIns，它由所有的插件共享。

提示：

可以使用许多选项卡，尤其在 Outlook 中；可以从 [www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en](http://www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en) 中下载包含完整列表的一系列电子表格。

决定了在何处放置 ribbon 组后，就可以添加如表 40-6 所示的控件了。

表 40-6

控 件	说 明
RibbonBox	这是一个容器控件，可用于布置组中的其他控件。可以把 BoxStyle 属性改为 RibbonBoxStyle.Horizontal 或 RibbonBoxStyle.Vertical，在 RibbonBox 中水平或垂直布置控件
RibbonButton	这个控件可用于在组中添加大按钮或小按钮，在按钮的旁边可以有或没有文本标签。把 ControlSize 属性设置为 RibbonControlSize.RibbonControlSizeLarge 或 RibbonControlSize.RibbonControlSizeRegular，就可以控制大小。按钮的 Click 事件处理程序可用于响应交互操作。还可以设置定制图像或存储在 Office 系统中的图像(详见本表后面的内容)
RibbonButtonGroup	这是一个容器控件，表示一组按钮。它可以包含 RibbonButton、RibbonGallery、RibbonMenu、RibbonSplitButton 和 RibbonToggleButton 控件
RibbonCheckBox	复选框控件，有 Click 事件和 Checked 属性
RibbonComboBox	组合框(合并了文本项和下拉列表)。列表项使用 Items 属性，输入的文本使用 Text 属性，TextChanged 事件用于响应交互操作
RibbonDropDown	这个容器可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。按钮和列表项格式化到下拉列表中。使用 SelectionChanged 事件响应交互操作

(续表)

控 件	说 明
RibbonEditBox	文本框，用户可用于输入或编辑 Text 属性中的文本。这个控件有 TextChanged 事件
RibbonGallery	与 RibbonDropDown 相同，这个控件也可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。这个控件使用 Click 和 ButtonClick 事件，来替代 RibbonDropDown 控件的 SelectionChanged 事件
RibbonLabel	显示简单的文本，用 Label 属性设置
RibbonMenu	弹出菜单，在设计视图中打开时，可以用其他控件填充该菜单，例如 RibbonButton

	和嵌套的 RibbonMenu 控件。处理菜单上的菜单项的事件
RibbonSeparator	一个简单的分隔符，用于定制组中的控件布局
RibbonSplitButton	合并了 RibbonButton 或 RibbonToggleButton 和 RibbonMenu 的控件。用 ButtonType 设置按钮的样式，该属性可以是 RibbonButtonType.Button 或 RibbonButtonType.ToggleButton。使用主按钮的 Click 事件或菜单中各按钮的 Click 事件来响应交互操作
RibbonToggleButton	一个按钮，可以处于选中或未选中状态，用 Checked 属性指定。这个控件也有 Click 事件

也可以设置组的 DialogBoxLauncher 属性，把一个图标显示在组的右下部。使用这个属性可以显示一个对话框，或者打开一个任务面板，或者执行其他操作。通过 GroupView Tasks 菜单可以添加或删除这个图标，如图 40-10 所示，该图还显示了表 40-5 中的其他一些控件，因为它们在设计视图中显示在 ribbon 上。



图 40-10

要给控件设置图像，例如给 RibbonButton 控件设置图像，就可以把 Image 属性设置为定制图像，ImageName 设置为图像名(以便在 OfficeRibbon.LoadImage 事件处理程序中优化图像的加载)，也可以使用内置的 Office 图像。为此，应把 OfficeImageId 属性设置为图像的 ID。

可以使用许多图像；还可以从 [www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en](http://www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en) 上下载包含这些图像的电子表格。图 40-11 显示了一个示例。



图 40-11

提示：

图 40-11 显示了 Developer ribbon 选项卡，通过 Popular 选项卡上的 Excel Options 对话框中的 Office 按钮可以打开它。

单击一个图像，就会打开一个对话框，指出该图像的 ID 是什么，如图 40-12 所示。

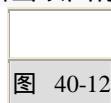


图 40-12

ribbon 设计器非常灵活，还可以提供希望出现在 Office ribbon 上的许多额外功能。但是，如果要进一步定制 UI，就要使用动作和任务面板，因为可以通过它们创建任意 UI 和功能。

## 2. 动作面板和定制的任务面板

使用动作和任务面板可以显示停放在 Office 应用程序界面的任务面板区域中的内容。任务面板在应用程序级的插件中使用，动作面板在文档级的定制中使用。任务和动作面板都必须继承自 UserControl 对象，这表示应使用 Windows 窗体创建一个 UI。如果把 WPF 窗体保存在 UserControl 的 ElementHost 控件上，还可以使用 WPF UI。这些控件的一个区别是可以通过 New Item Wizard 中的

Action Pane Template , 或使用简单的用户控件 , 把动作面板添加到文档级的定制上。任务面板必须添加为一般的用户控件。

要把动作面板添加到文档级定制中的一个文档上 , 应把动作面板类的一个实例添加到文档的 ActionsPane 属性的 Controls 集合中。例如 :

```
public partial class ThisWorkbook
{
    Private ActionsPaneControl1 actionsPane;
    private void ThisWorkbook_Startup(object sender,
    System.EventArgs e)
    {
        actionsPane = new ActionsPaneControl1();
        this.ActionsPane.Controls.Add(actionsPane);
    }
    ...
}
```

这段代码在加载文档(这里是 Excel 工作簿)时添加了动作面板。也可以在 ribbon 按钮事件处理程序中添加动作面板。

在应用程序级插件项目中 , 定制的任务面板通过 ThisAddIns.CustomTaskPanes.Add()方法属性添加。这个方法也允许命名任务窗口 , 例如 :

```
public partial class ThisAddIn
{
    Microsoft.Office.Tools.CustomTaskPane taskPane;
    private void ThisAddIn_Startup(object sender,
    System.EventArgs e)
    {
        taskPane = this.CustomTaskPanes.Add(new UserControl1(),
        "My Task Pane");
        taskPane.Visible = true;
    }
    ...
}
```

注意 Add()方法返回一个 Microsoft.Office.Tools.CustomTaskPane 类型的对象。可以通过这个对象的 Control 属性访问用户控件本身。还可以使用这个类型的其他属性 , 例如上面代码中的 Visible 属性 , 来控制任务面板。

此时 , 应注意 Office 应用程序的一个不太寻常的特性 , 尤其是 Word 和 Excel 之间的区别。由于历史的原因 , 尽管 Word 和 Excel 都是 MDI 应用程序 , 但这两个应用程序存储文档的方式是不同的。在 Word 中 , 每个文档都有一个唯一的父窗口 , 而在 Excel 中 , 每个文档都共享同一个父窗口。

在调用 CustomTaskPanes.Add()方法时，默认操作是把任务面板添加到当前活动的窗口中。在 Excel 中，这表示每个文档都显示该任务面板。因为它们都使用同一个父窗口。而在 Word 中，情况就不同了。如果希望任务面板显示给每个文档，就必须把它添加到包含文档的每个窗口中。

要把任务面板添加到特定的文档中，应给 Add()方法传送 Microsoft.Office.Interop.Word.Windows 类的一个实例，作为第三个参数。通过 Microsoft.Office.Interop.Word.Document.ActiveWindow 属性可以获得关联了文档的窗口。

下一节介绍如何完成这个操作。

#### 40.4 示例应用程序

如前所述，本章的示例代码包含一个应用程序 WordDocEditTimer，它维护着 Word 文档的一个编辑次数列表。本节将详细解释这个应用程序的代码，因为该应用程序演示了前面介绍的所有内容，还包含一些有益的提示。

这个应用程序的一般操作是只要创建或加载了文档，就启动一个链接到文档名称上的计时器。如果关闭文档，该文档的计时器就暂停。如果打开了以前计时的文档，计时器就恢复。另外，如果使用 Save As 把文档保存为另一个文件名，计时器就更新为使用新文件名。

这个应用程序是一个 Word 应用程序级的插件，使用一个定制任务面板和一个 ribbon 菜单。ribbon 菜单包含一个按钮和一个复选框，按钮用于开关任务面板，复选框用于暂停当前活动的文档的计时器。包含这些控件的组添加到 Home ribbon 选项卡的最后。任务面板显示一组活动的计时器。

这个用户界面如图 40-13 所示。

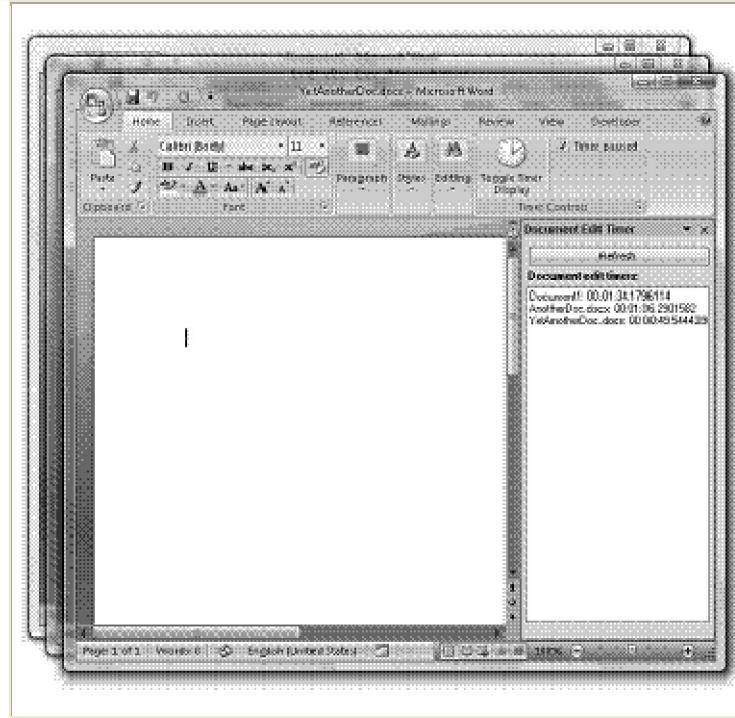


图 40-13

计时器通过 DocumentTimer 类来维护：

```
public class DocumentTimer
{
    public Word.Document Document { get; set; }
    public DateTime LastActive { get; set; }
    public bool IsActive { get; set; }
    public TimeSpan EditTime { get; set; }
}
```

这段代码保存了对 Microsoft.Office.Interop.Word.Document 对象的一个引用、总编辑时间、计时器是否激活，以及它上一次激活的时间。ThisAddIn 类维护这些对象的一个集合，这些对象与文档名关联起来：

```
public partial class ThisAddIn
{
    private Dictionary<string, DocumentTimer>
        documentEditTimes;
```

因此，每个计时器都可以通过文档引用或文档名来定位。这是必要的，因为文档引用可以跟踪文档名的变化(这里没有可用于监控文档名变化的事件)，文档名允许跟踪关闭、再次打开的文档。

ThisAddIn 类还维护一个 CustomTaskPane 对象列表(如前所述，Word 中的每个窗口都需要一个 CustomTaskPane 对象)：

```
private List<Tools.CustomTaskPane> timerDisplayPanes;
```

插件启动时，ThisAddIn\_Startup()方法执行了几个任务。首先它初始化两个集合：

```
private void ThisAddIn_Startup(object sender,
System.EventArgs e)
{
    // Initialize timers and display panels
    documentEditTimes = new Dictionary<string, DocumentTimer>();
    timerDisplayPanes = new
    List<Microsoft.Office.Tools.CustomTaskPane>();
```

接着通过 ApplicationEvents4\_Event 接口添加几个事件处理程序：

```
// Add event handlers
Word.ApplicationEvents4_Event eventInterface =
this.Application;
eventInterface.DocumentOpen += new
Microsoft.Office.Interop.Word
.ApplicationEvents4_DocumentOpenEventHandler(
eventInterface_DocumentOpen);
eventInterface.NewDocument += new
```

```
Microsoft.Office.Interop.Word
.ApplicationEvents4_NewDocumentEventHandler(
eventInterface_NewDocument);
eventInterface.DocumentBeforeClose += new
Microsoft.Office.Interop.Word
.ApplicationEvents4_DocumentBeforeCloseEventHandler(
eventInterface_DocumentBeforeClose);
eventInterface.WindowActivate += new
Microsoft.Office.Interop.Word
.ApplicationEvents4_WindowActivateEventHandler(
eventInterface_WindowActivate);
```

这些事件处理程序用于监控文档的打开、创建和关闭，并确保 ribbon 上的 Pause 复选框保持最新状态。后一个功能是使用 WindowsActivate 事件跟踪窗口的激活状态来实现的。

在这个事件处理程序中，最后一个任务是开始监控当前文档，把定制的任务面板添加到包含文档的窗口中：

```
// Start monitoring active document
MonitorDocument(this.Application.ActiveDocument);
AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
}
```

MonitorDocument()实用方法为文档添加一个计时器：

```
internal void MonitorDocument(Word.Document Doc)
{
// Monitor doc
documentEditTimes.Add(Doc.Name, new DocumentTimer
{
Document = Doc,
EditTime = new TimeSpan(0),
IsActive = true,
LastActive = DateTime.Now
}) ;
}
```

这个方法仅为文档创建了一个新的 DocumentTimer 对象。DocumentTimer 引用文档，其编辑次数是 0，且是在当前时间激活的。接着把这个计时器添加到 documentEditTimes 集合中，并关联到文档名中。

AddTaskPaneToWindow()方法把定制任务面板添加到窗口中。这个方法首先检查已有的任务面板，确保窗口中还没有任务面板。Word 中的另一个古怪的特性是如果在加载应用程序后，立即打开一个旧文档，默认的 Document1 文档就会消失，且不触发关闭事件。在访问包含任务面板的文档窗口时，这可能导致异常，所以该方法还检查表示是否出现该异常的 ArgumentNullException：

```
private void AddTaskPaneToWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    Tools.CustomTaskPane paneToRemove = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        try
        {
            if (pane.Window == Wn)
            {
                docPane = pane;
                break;
            }
        }
        catch (ArgumentNullException)
        {
            // pane.Window is null, so document1 has been unloaded.
            paneToRemove = pane;
        }
    }
}
```

如果抛出了一个异常，就从集合中删除错误的任务面板：

```
// Remove pane if necessary
timerDisplayPanes.Remove(paneToRemove);
```

如果窗口中没有任务面板，这个方法就添加一个：

```
// Add task pane to doc
if (docPane == null)
{
    Tools.CustomTaskPane pane = this.CustomTaskPanes.Add(
        new TimerDisplayPane(documentEditTimes),
        "Document Edit Timer",
        Wn);
    timerDisplayPanes.Add(pane);
    pane.VisibleChanged +=
        new EventHandler(timerDisplayPane_VisibleChanged);
}
```

添加的任务面板是 TimerDisplayPane 类的一个实例。稍后介绍这个类。它添加时使用的名称是 Document Edit Timer。另外，在调用 CustomTaskPanes.Add()方法后，还为得到的 CustomTaskPane 的 VisibleChanged 事件添加了一个处理程序，这样在第一次显示任务面板时，可以刷新显示：

```
private void timerDisplayPane_VisibleChanged(object sender, EventArgs e)
{
    // Get task pane and toggle visibility
    Tools.CustomTaskPane taskPane =
        (Tools.CustomTaskPane)sender;
    if (taskPane.Visible)
    {
        TimerDisplayPane timerControl =
            (TimerDisplayPane)taskPane.Control;
        timerControl.RefreshDisplay();
    }
}
```

TimerDisplayPane 类有一个 RefreshDisplay()方法，它在上面的代码中调用。这个方法刷新 timerControl 对象的显示。

接着的代码确保监控所有的文档。首先创建新文档时，调用 eventInterface\_New- Document()事件处理程序，调用 MonitorDocument()和前面介绍过的 AddTaskPaneTo- Window()方法监控文档。

```
private void eventInterface_NewDocument(Word.Document Doc)
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
```

新文档在计时器运行时启动，此时这个方法还清除了 ribbon 菜单中的 Pause 复选框。这是通过一个实用方法 SetPauseStatus()实现的，该方法在 ribbon 中定义：

```
// Set checkbox
Globals.Ribbons.TimerRibbon.SetPauseStatus(false);
```

在关闭文档之前，调用 eventInterface\_DocumentBeforeClose()事件处理程序。这个方法冻结了文档的计时器，更新了总编辑时间，清除了 Document 引用，删除了文档窗口中的任务面板(使用稍后介绍的 RemoveTaskPaneFromWindow()方法)，之后关闭窗口。

```
private void
eventInterface_DocumentBeforeClose(Word.Document Doc,
ref bool Cancel)
{
    // Freeze timer
    documentEditTimes[Doc.Name].EditTime += DateTime.Now
        - documentEditTimes[Doc.Name].LastActive;
    documentEditTimes[Doc.Name].IsActive = false;
```

```
documentEditTimes[Doc.Name].Document = null;
// Remove task pane
RemoveTaskPaneFromWindow(Doc.ActiveWindow);
}
```

打开文档时，调用 eventInterface\_DocumentOpen()方法。该方法完成了许多工作，因为在监控文档之前，这个方法必须查看计时器的名称，确定文档是否已有计时器：

```
private void eventInterface_DocumentOpen(Word.Document Doc)
{
if (documentEditTimes.ContainsKey(Doc.Name))
{
// Monitor old doc
documentEditTimes[Doc.Name].LastActive = DateTime.Now;
documentEditTimes[Doc.Name].IsActive = true;
documentEditTimes[Doc.Name].Document = Doc;
AddTaskPaneToWindow(Doc.ActiveWindow);
}
}
```

如果还没有监控文档，就为文档配置一个新监控器：

```
else
{
// Monitor new doc
MonitorDocument(Doc);
AddTaskPaneToWindow(Doc.ActiveWindow);
}
}
```

RemoveTaskPaneFromWindow()方法用于从窗口中删除任务面板。其代码首先检查特定的窗口中是否有任务面板：

```
private void RemoveTaskPaneFromWindow(Word.Window Wn)
{
// Check for task pane in window
Tools.CustomTaskPane docPane = null;
foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
{
if (pane.Window == Wn)
{
docPane = pane;
break;
}
}
}
```

如果找到了任务面板，就调用 CustomTaskPanes.Remove()方法删除它。还要从任务面板引用的本地集合中删除它。

```
// Remove document task pane
if (docPane != null)
{
    this.CustomTaskPanes.Remove(docPane);
    timerDisplayPanes.Remove(docPane);
}
```

这个类中的最后一个事件处理程序是 eventInterface\_WindowActivate()，在激活窗口时调用它。这个方法获得活动文档的计时器，选中 ribbon 菜单中的复选框，以更新文档的复选框：

```
private void eventInterface_WindowActivate(Word.Document Doc,
Word.Window Wn)
{
    // Ensure pause checkbox in ribbon is accurate, start by getting timer
    DocumentTimer documentTimer =
    documentEditTimes[this.Application.ActiveDocument.Name];
    // Set checkbox
    Globals.Ribbons.TimerRibbon.SetPauseStatus(!documentTimer.IsActive);
}
```

ThisAddIn 的代码还包含两个实用方法。第一个方法 ToggleTaskPaneDisplay()用于设置 CustomTaskPanes.Visible 属性，为当前活动的文档显示或隐藏任务面板。

```
internal void ToggleTaskPaneDisplay()
{
    // Ensure window has task window
    AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
    // toggle document task pane
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == this.Application.ActiveDocument.ActiveWindow)
        {
            docPane = pane;
            break;
        }
    }
    docPane.Visible = !docPane.Visible;
}
```

上述代码中的 ToggleTaskPaneDisplay()方法由 ribbon 控件上的事件处理程序调用，如后面所述。

最后，该类有另一个从 ribbon 菜单中调用的方法，它允许 ribbon 控件暂停或恢复文档的计时器：

```
internal void PauseOrResumeTimer(bool pause)
{
    // Get timer
    DocumentTimer documentTimer =
        documentEditTimes[this.Application.ActiveDocument.Name];
    if (pause & & documentTimer.IsActive)
    {
        // Freeze timer
        documentTimer.EditTime += DateTime.Now -
            documentTimer.LastActive;
        documentTimer.IsActive = false;
    }
    else if (!pause & & !documentTimer.IsActive)
    {
        // Resume timer
        documentTimer.IsActive = true;
        documentTimer.LastActive = DateTime.Now;
    }
}
```

这个类定义中的其他代码是 Shutdown 的空事件处理程序以及 VSTO 为关联 Startup 和 Shutdown 事件处理程序而生成的代码。

接着布置项目中的 ribbon，即 TimerRibbon，如图 40-14 所示。



图 40-14

这个 ribbon 包含一个 RibbonButton、一个 RibbonSeparator、一个 RibbonCheckBox 和一个 DialogBoxLauncher。按钮使用大显示样式，其 OfficeImageId 设置为 StartAfterPrevious，显示如图 40-13 所示的钟表图像。(这些图像在设计期间不可见)。ribbon 使用 TabHome 选项卡类型，其内容追加到 Home 选项卡上。

ribbon 有 3 个事件处理程序，每个处理程序都调用前面介绍的 ThisAddIn 中的一个实用方法：

```
private void group1_DialogLauncherClick(object sender,
    RibbonControlEventArgs e)
{
```

```
// Show or hide task pane
Globals.ThisAddIn.ToggleTaskPaneDisplay();
}

private void pauseCheckBox_Click(object sender,
RibbonControlEventArgs e)
{
// Pause timer
Globals.ThisAddIn.PauseOrResumeTimer(pauseCheckBox.Checked);
}

private void toggleDisplayButton_Click(object sender,
RibbonControlEventArgs e)
{
// Show or hide task pane
Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
```

ribbon 还包含自己的实用方法 SetPauseStatus() , 如前所述 , 该方法由 ThisAddIn 中的代码调用 , 以选中复选框或取消复选框的选中。

```
internal void SetPauseStatus(bool isPaused)
{
// Ensure checkbox is accurate
pauseCheckBox.Checked = isPaused;
}
```

这个解决方案中的另一个组件是任务面板中使用的 TimerDisplayPane 用户控件 , 这个控件的布局如图 40-15 所示。

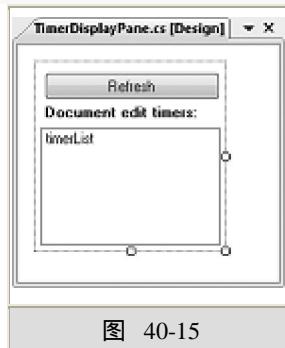


图 40-15

这个控件包含一个按钮、一个标签和一个列表框--这些都是很普通的显示控件 , 也可以用更漂亮的 WPF 控件替代它们。

该控件的代码保存了对文档计时器的一个本地引用 , 该引用在构造函数中设置 :

```
public partial class TimerDisplayPane : UserControl
{
private Dictionary < string, DocumentTimer >
documentEditTimes;
public TimerDisplayPane()
```

```
{  
    InitializeComponent();  
}  
public TimerDisplayPane(Dictionary<string,  
DocumentTimer>  
documentEditTimes) : this()  
{  
    // Store reference to edit times  
    this.documentEditTimes = documentEditTimes;  
}
```

按钮事件处理程序调用 RefreshDisplay()方法刷新计时器的显示：

```
private void refreshButton_Click(object sender, EventArgs  
e)  
{  
    RefreshDisplay();  
}
```

RefreshDisplay()方法也从 ThisAddIn 中调用，如前所述。考虑到该方法的任务，这是一个相当复杂的方法，它还检查被监控文档的列表，与已加载文档的列表比较，并解决出现的问题。这段代码在 VSTO 应用程序中常常是必不可少的，因为 COM Office 对象模型的接口偶尔不能像期望的那样工作。这里的规则是防御式编码。

该方法首先清除 timerList 列表框中的当前计时器列表：

```
internal void RefreshDisplay()  
{  
    // Clear existing list  
    this.timerList.Items.Clear();
```

接着检查监控器。这个方法迭代 Globals.ThisAddIn.Application.Documents 集合中的每个文档，确定文档是被监控、未被监控、或被监控了但在上次刷新时改变了文件名。

要找出被监控的文档，只需比较当前的文档名和键的 documentEditTimes 集合中的文档名：

```
// Ensure all docs are monitored  
foreach (Word.Document doc in  
Globals.ThisAddIn.Application.Documents)  
{  
    bool isMonitored = false;  
    bool requiresNameChange = false;  
    DocumentTimer oldNameTimer = null;  
    string oldName = null;  
    foreach (string documentName in documentEditTimes.Keys)  
{
```

```
if (doc.Name == documentName)
{
    isMonitored = true;
    break;
}
```

如果文档名不匹配，就比较文档引用，以检测对文档名的修改，如下面的代码所示：

```
else
{
    if (documentEditTimes[documentName].Document == doc)
    {
        // Monitored, but name changed!
        oldName = documentName;
        oldNameTimer = documentEditTimes[documentName];
        isMonitored = true;
        requiresNameChange = true;
        break;
    }
}
```

对于未监控的文档，需要创建一个新的监控器：

```
// Add monitor if not monitored
if (!isMonitored)
{
    Globals.ThisAddIn.MonitorDocument(doc);
}
```

名称改变的文档需要通过用于旧文档的监控器重新关联起来：

```
// Rename if necessary
if (requiresNameChange)
{
    documentEditTimes.Remove(oldName);
    documentEditTimes.Add(doc.Name, oldNameTimer);
}
```

调整了文档编辑计时器后，生成一个列表。代码还会检测引用的文档是否加载了，对于没有加载的文档，把 IsActive 属性设置为 false，暂停该文档的计时器。这也是防御性编程方式：

```
// Create new list
foreach (string documentName in documentEditTimes.Keys)
{
    // Check to see if doc is still loaded
```

```
bool isLoaded = false;
foreach (Word.Document doc in
    Globals.ThisAddIn.Application.Documents)
{
    if (doc.Name == documentName)
    {
        isLoaded = true;
        break;
    }
}
if (!isLoaded)
{
    documentEditTimes[documentName].IsActive = false;
    documentEditTimes[documentName].Document = null;
}
```

对于每个监控器，把一个列表项添加到列表框中，其中包含了文档名和总编辑时间：

```
// Add item
this.timerList.Items.Add(string.Format("{0}: {1}",
    documentName,
    documentEditTimes[documentName].EditTime +
    (documentEditTimes[documentName].IsActive ?
        (DateTime.Now -
        documentEditTimes[documentName].LastActive) :
        new TimeSpan(0))));
```

这就完成了这个例子中的代码。这个例子说明了如何使用 ribbon 和任务面板控件，如何维护多个 Word 文档中的任务面板，还演示了本章前面介绍的许多技术。

#### 40.5 VBA 交互操作性

Office 系统已经推出了多年，所以读者很熟悉 VBA 代码，在已有的应用程序中还使用了 VBA。在 VSTO 解决方案中可以重写 VBA 代码，但这并不总是切合实际。看到 VSTO 的功能后，读者可能希望用托管的 VSTO 代码替代已有的 VBA 功能，或者添加新功能。

VSTO 允许给 VBA 代码提供 VSTO 功能，以实现上述任务。为此，必须执行几个步骤，才能给 VBA 代码提供 COM 接口。这 9 步如下所示，还列出了下载代码的 ExcelVBAInterop 项目中的示例代码和屏幕图：

(1) 在开始给 VBA 提供 VSTO 代码之前，必须有一个包含 VBA 项目的文档。为了便于开发，最好在开始之前启动文档中的宏。接着，在 VSTO 中创建一个文档级的定制时，把该文档作为自己解决方案中文档的起点。

(2) 有了这个起点后，就可以用通常的方式编写访问应用程序和/或文档的代码了。这些代码不能通过 VSTO 项目访问，因为后面要提供一个 VBA 接口。所以应创建 VBA 可以调用的方法，例如：

```
public partial class ThisWorkbook :  
    ExcelVBAInterop.IThisWorkbook  
{  
    ...  
    public void NameSheet()  
    {  
        NamingDialog dlg = new NamingDialog();  
        if (dlg.ShowDialog() == DialogResult.OK)  
        {  
            ((Excel.Worksheet)this.ActiveSheet).Name =  
                dlg.SheetName;  
        }  
    }  
}
```

**提示：**

这些代码使用一个简单的定制对话框，稍后介绍它。这个对话框允许用户输入一个字符串，选择是否在表名中包含当前日期。

(3) 必须重写 GetAutomationObject()方法，为 VBA 代码的自动执行返回正确的对象，如下所示：

```
public partial class ThisWorkbook :  
    ExcelVBAInterop.IThisWorkbook  
{  
    ...  
    protected override object GetAutomationObject()  
    {  
        return this;  
    }  
}
```

(4) 因为 COM 系统通过接口来工作，所以必须通过接口提供要调用的方法。最简单的方法是右击代码，选择 Refactor | Extract Interface，接着选择要在接口中提供的方法，向导会完成剩余的工作，如图 40-16 所示。



图 40-16

(5) 还必须把 System.Runtime.InteropServices 命名空间中的属性添加到类中 ,使类显示在 COM 上 (参见第 24 章) :

```
using System.Runtime.InteropServices;
namespace ExcelVBAInterop
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public partial class ThisWorkbook :
        ExcelVBAInterop.IThisWorkbook
    {
        ...
    }
}
```

(6) 生成的接口还需要 ComVisible 属性 , 该接口必须是公共的 :

```
using System.Runtime.InteropServices;
namespace ExcelVBAInterop
{
    [ComVisible(true)]
    public interface IThisWorkbook
    {
        void NameSheet();
    }
}
```

(7) 把文档的 ReferenceAssemblyFromVbaProject 属性改为 true , 如图 40-17 所示。如果文档不包含 VBA 代码 , 就不能修改这个属性。改变它时 , 会接收到一个警告 , 说明项目运行时 , 添加到项目中的 VBA 代码会丢失 , 所以应备份修改的 VBA 代码。

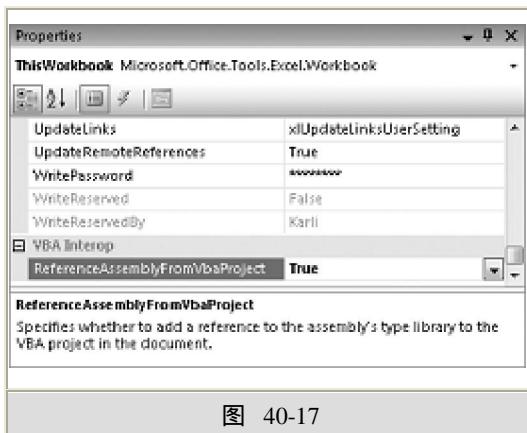


图 40-17

(8) 剩下的就是对文档中 VBA 代码的修改了。可以在此时运行项目，运行项目时可以通过 Developer 选项卡或按下 Alt+F11，来访问 VBA 代码。首先需要添加的是一个允许 VBA 访问 VSTO 代码的属性，如下所示(按添加了命名空间限制符的名称引用类)：

```
Property Get VSTOAssembly() As  
    ExcelVBAInterop.ThisWorkbook  
    Set VSTOAssembly = GetManagedObject(Me)  
End Property
```

(9) 接着就可以通过这个属性调用 VSTO 方法了：

```
Public Sub RenameSheet()  
    VSTOAssembly.NameSheet  
End Sub
```

完成了这些步骤后，就可以添加代码，调用接口上的方法，或手工调用它，如图 40-18 所示。

如果代码包含一个 UI，如本例所示，就会显示该 UI，且可以使用它。示例项目中的 UI 如图 40-19 所示。



图 40-18

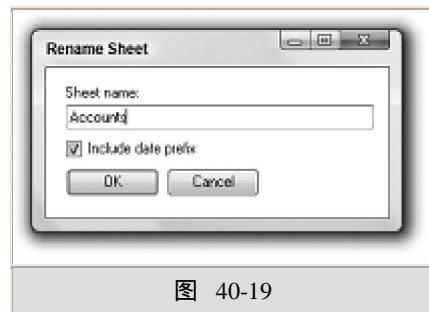


图 40-19

这样，就可以使 VSTO 代码用于 VBA 代码了。

#### 40.6 小结

本章学习了如何使用 VSTO 为 Office 产品创建托管的解决方案。

在本章的第一部分，介绍了 VSTO 项目的一般结构和可以创建的项目类型，还探讨了便于 VSTO 编程的特性。

接下来详细论述了 VSTO 解决方案中的一些特性，讨论了如何与 Office 对象模型通信，介绍了 VSTO 中的命名空间和类型，陈述了如何使用这些类型实现各种功能。之后研究了 VSTO 项目的一些编码特性，以及如何使用这些特性获得希望的结果。

然后，进行了一些实践。我们学习了如何在 Office 应用程序中管理插件，如何与 Office 对象模型交互操作，如何用 ribbon 菜单、任务面板和动作面板定制应用程序的 UI。

接着开发了一个示例应用程序，演示了前面学习的 UI 和交互操作技术。这个示例包含许多代码，还包含有用的技巧，例如如何在多个 Word 文档窗口中管理任务面板。

最后介绍了与 VBA 代码的交互操作。这部分介绍了如何通过 COM 交互操作性把托管代码提供给 VBA，并用另一个示例演示了这些技术。

这是第 部分的最后一章。第 部分的第一章将介绍如何在应用程序中使用 System.Net 命名空间中的类访问 Internet。

## 第 48 章 Syndication

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```
<div>
<asp:Label Runat="server" ID="resultLabel" /><br />
<asp:Button Runat="server" ID="triggerButton"
Text="Click Me"
OnClick="triggerButton_Click" />
</div>
```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click()中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```
void triggerButton_Click(object sender, EventArgs
e)
{
resultLabel.Text = "Button clicked!";
}
```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这就很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。



图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中)快速查看一下浏览器接收到的代码。<form>部分应如下所示：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
```

```
value=" /wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE="  
>  
</div>  
<div>  
<span id="resultLabel"></span><br />  
<input type="submit" name="triggerButton" value="Click Me"  
id="triggerButton" />  
</div>  
<div>  
<input type="hidden" name="__EVENTVALIDATION"  
id="__EVENTVALIDATION"  
value=" /wEWAqK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />  
</div>  
</form>
```

Web 服务器控件生成了 HTML , <span>和<input>分别代表<asp:Label>和<asp:Button>。还有一个名为 VIEWSTATE 的<input type="hidden">字段 , 把前面提到的窗体状态封装起来。在窗体传送回服务器以重新创建 UI , 以及跟踪改变时使用这些信息。注意<form>元素已经进行了配置 , 通过 HTTP POST 操作(在 method 中指定)把数据传送回 Default.aspx(在 action 中指定) , 它还被赋予了一个名称 form1。

在单击按钮 , 查看文本后 , 可再次浏览源 HTML(下面添加了必要的空格 , 使代码比较清晰) :

```
<form method="post" action="Default.aspx" id="form1">  
<div>  
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"  
value=" /wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHRvbIB  
jbGlja2VkiWRkZN3zQXZqDnF2ddEBw4Kj7MEqqj9pJ" />  
</div>  
<div>  
<span id="resultLabel">Button clicked!</span><br />  
<input type="submit" name="triggerButton" value="Click Me"  
id="triggerButton" />  
</div>  
<div>  
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"  
value=" /wEWAqLl3MPHAwLHpP+yCwFfz4kBL6+KPlxjB0lgrAageely" />  
</div>  
</form>
```

这次 viewstate 的值包含比较多的信息 ,因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上 ,这可能是一个非常长的字符串 ,但这是由系统在后台完成的 ,我们几乎可以不考虑状态管理 ,只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时 ,可以禁用不需要保

留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态，以提高性能，就可以禁用整个页面的 Viewstate。

注意：

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译，把 Default.aspx.cs 中的文本"Button clicked!"改为其他内容，保存文件，再次单击按钮。Web 页面上的文本会做相应的改变。

### 1. 控件面板

本节介绍可用控件，之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别，如图 37-6 所示。

注意，在控件的描述中使用了"属性"-- ASP.NET 代码中使用的属性与它同名。这里的引用并不完整，许多控件和属性都没有介绍，只介绍了最常用的属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation 和 Login 和 WebParts 类别在第 38 章介绍，AJAX 扩展控件在第 39 章介绍，Reporting 控件可以在 Web 页面上报告信息，包括 Crystal Reports，本书不讨论。



图 37-6

#### (1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI. WebControls. WebControl，而 System.Web.UI.WebControls.WebControl 又继承了 System.Web. UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类，而该基类又最终派生于 Control。因此，Web 服务器控件有许多共同的属性和事件，如果需要，就可以使用这些属性和事件。这里不可能介绍所有的元素，只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式，这是很容易控制的，例如属性 ForeColor、BackColor、Font 等，也可以使用 CSS(Cascading Style Sheet)类来控制。此时，应在一个独立的文件中，把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括：Width 和 Height，用于设置控件的大小；AccessKey 和 TabIndex，便于用户的交互操作；Enabled，设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件，在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件，使用 Parent 可以访问控件的容器。

对于事件，最常用的是继承来的 Load 事件，它执行控件的初始化，PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多，下一章将详细介绍它们，尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控件	说明
Label	显示简单文本，使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据，TextChanged 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮)就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本，Click 事件用于响应单击(服务器回送是自动的)。也可以使用 Command 事件响应单击，该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同，但把按钮显示为超链接

(续表)

控件	说明
ImageButton	显示一个图像，该图像放大一倍作为一个可单击的按钮，其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地，用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像，用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件，如果在链接后要执行其他处理，就应使用 LinkButton
DropDownList	允许用户选择一个列表项，可以直接从列表中选择，也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类)，SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变，这个控件也有 AutoPostBack 属性，所以选项的改变会触发一个回送操作
ListBox	允许用户从列表中选择一个或多个列表。把SelectionMode 设置为 Multiple 或 Single，可以确定一次选择多少个选项，Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中，与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送，CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同，例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下，它们都组合在一个组中，其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮，在这个组中，一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用，如果图像加载失败，由 AlternateText

	提供对应的文本
ImageMap	类似于 Image ,但在用户单击图像中的一个或多个热区时 ,可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供 ,例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、 TableRow 和 TableCell ,或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样 ,这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同 ,这个控件有一个 Click 事件 ,用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控件	说 明
HiddenField	用于提供隐藏的字段 ,以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能 ,但没有样式属性 ,只有一个 Text 属性(因为它派生于 Control ,而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性 ,但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.Date Time)来访问由用户选择的日期和月份 ,并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged 。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后 ,显示另一个图像。使用 Advertisement- File 属性指定描述图像的 XML 文件 , AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮 ,以选择要上传的文件。用户选择了文件之后 ,就可以使用 HasFile 属性确定是否选择了文件 ,然后使用后台代码中的 SaveAs() 方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤 ,按顺序或不按顺序显示给用户 ,并依赖此控件来维护状态
Xml	这是一个更复杂的文本显示控件 ,用于显示用 XSLT 样式表传输的 XML 内容 ,这些 XML 内容是使用 Document 、 DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的 ,XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件 ,每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定 ,如果视图改变了(可能因为单击了当前视图上的 Next 链接) ,就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出 ,但可以方便地把其他控件组合在一起 ,或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器 ,类似于 PlaceHolder ,但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View ,使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一起高速缓存的 Web 页面 ,这是一个与 ASP.NET 高速缓存相关的高级主题 ,本书不涉及
Localize	与 Literal 相同 ,但允许使用项目资源指定要在不同区域显示的文本 ,使文本本地化

## (2) 数据 Web 服务器控件

数据 Web 服务器控件分为两类：

数据源控件(SqlDataSource、 AccessDataSource、 ObjectDataSource、 XmlDataSource、 和 SiteMapDataSource)

数据显示控件(GridView、 DataList、 DetailsView、 FormView、 Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

### 48.1 System.ServiceModel.Syndication 命名空间概述

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```
<div>
<asp:Label Runat="server" ID="resultLabel" /><br />
<asp:Button Runat="server" ID="triggerButton"
Text="Click Me"
OnClick="triggerButton_Click" />
</div>
```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click()中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```
void triggerButton_Click(object sender, EventArgs
e)
{
resultLabel.Text = "Button clicked!";
}
```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这就很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。

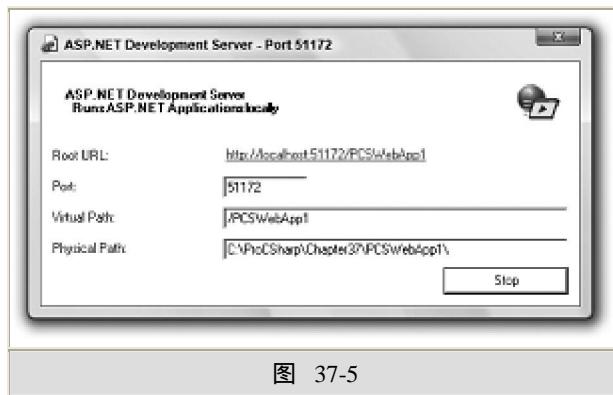


图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中)快速查看一下浏览器接收到的代码。`<form>`部分应如下所示：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE="
/>
</div>
<div>
<span id="resultLabel"></span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION"
id="__EVENTVALIDATION"
value="/wEWAqK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />
</div>
</form>
```

Web 服务器控件生成了 HTML，`<span>` 和 `<input>` 分别代表 `<asp:Label>` 和 `<asp:Button>`。还有一个名为 VIEWSTATE 的 `<input type="hidden">` 字段，把前面提到的窗体状态封装起来。在窗体传送回服务器以重新创建 UI，以及跟踪改变时使用这些信息。注意 `<form>` 元素已经进行了配置，通过 HTTP POST 操作(在 `method` 中指定)把数据传送回 Default.aspx(在 `action` 中指定)，它还被赋予了一个名称 `form1`。

在单击按钮，查看文本后，可再次浏览源 HTML(下面添加了必要的空格，使代码比较清晰)：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHRvbIB
jbGlja2VkiWRkZN3zQXZqDnF2ddEBw4Kj7MEqj9pJ" />
</div>
```

```
<div>
<span id="resultLabel">Button clicked!</span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgLl3MPHAwLHpP+yCwFfz4kBL6+KP1xjB0lgrAageely" />
</div>
</form>
```

这次 viewstate 的值包含比较多的信息 ,因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上 ,这可能是一个非常长的字符串 ,但这是由系统在后台完成的 ,我们几乎可以不考虑状态管理 ,只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时 ,可以禁用不需要保留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态 ,以提高性能 ,就可以禁用整个页面的 Viewstate。

注意 :

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译 ,把 Default.aspx.cs 中的文本"Button clicked!"改为其他内容 ,保存文件 ,再次单击按钮。Web 页面上的文本会做相应的改变。

### 1. 控件面板

本节介绍可用控件 ,之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别 ,如图 37-6 所示。

注意 ,在控件的描述中使用了"属性"-- ASP.NET 代码中使用的属性与它同名。这里的引用并不完整 ,许多控件和属性都没有介绍 ,只介绍了最常用的属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation and Login 和 WebParts 类别在第 38 章介绍 ,AJAX 扩展控件在第 39 章介绍 ,Reporting 控件可以在 Web 页面上报告信息 ,包括 Crystal Reports ,本书不讨论。



(1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI.WebControls.WebControl, 而 System.Web.UI.WebControls.WebControl 又继承了 System.Web.UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类, 而该基类又最终派生于 Control。因此, Web 服务器控件有许多共同的属性和事件, 如果需要, 就可以使用这些属性和事件。这里不可能介绍所有的元素, 只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式, 这是很容易控制的, 例如属性 ForeColor、BackColor、Font 等, 也可以使用 CSS(Cascading Style Sheet)类来控制。此时, 应在一个独立的文件中, 把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括: Width 和 Height, 用于设置控件的大小; AccessKey 和 TabIndex, 便于用户的交互操作; Enabled, 设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件, 在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件, 使用 Parent 可以访问控件的容器。

对于事件, 最常用的是继承来的 Load 事件, 它执行控件的初始化, PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多, 下一章将详细介绍它们, 尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控件	说明
Label	显示简单文本, 使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据, Text Changed 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮)就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本, Click 事件用于响应单击(服务器回送是自动的)。也可以使用 Command 事件响应单击, 该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同, 但把按钮显示为超链接

(续表)

控件	说明
ImageButton	显示一个图像, 该图像放大一倍作为一个可单击的按钮, 其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地, 用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像, 用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件, 如果在链接后要执行其他处理, 就应使用 LinkButton
DropDownList	允许用户选择一个列表项, 可以直接从列表中选择, 也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类), SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件

	可用于确定选项是否改变，这个控件也有 AutoPostBack 属性，所以选项的改变会触发一个回送操作
ListBox	允许用户从列表中选择一个或多个列表。把 SelectionMode 设置为 Multiple 或 Single，可以确定一次选择多少个选项，Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中，与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送，CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同，例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下，它们都组合在一个组中，其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮，在这个组中，一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用，如果图像加载失败，由 AlternateText 提供对应的文本
ImageMap	类似于 Image，但在用户单击图像中的一个或多个热区时，可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供，例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、 TableRow 和 TableCell，或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样，这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同，这个控件有一个 Click 事件，用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控件	说 明
HiddenField	用于提供隐藏的字段，以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能，但没有样式属性，只有一个 Text 属性(因为它派生于 Control，而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性，但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.DateTime)来访问由用户选择的日期和月份，并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后，显示另一个图像。使用 Advertisement-File 属性指定描述图像的 XML 文件，AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮，以选择要上传的文件。用户选择了文件之后，就可以使用 HasFile 属性确定是否选择了文件，然后使用后台代码中的 SaveAs()方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤，按顺序或不按顺序显示给用户，并依赖此控件来维护状态

Xml	这是一个更复杂的文本显示控件，用于显示用 XSLT 样式表传输的 XML 内容，这些 XML 内容是使用 Document、DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的，XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件，每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定，如果视图改变了(可能因为单击了当前视图上的 Next 链接)，就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出，但可以方便地把其他控件组合在一起，或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器，类似于 PlaceHolder，但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View，使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一起高速缓存的 Web 页面，这是一个与 ASP.NET 高速缓存相关的高级主题，本书不涉及
Localize	与 Literal 相同，但允许使用项目资源指定要在不同区域显示的文本，使文本本地化

## (2) 数据 Web 服务器控件

数据 Web 服务器控件分为两类：

数据源控件(SqlDataSource、 AccessDataSource、 ObjectDataSource、 XmlDataSource、 和 SiteMapDataSource)

数据显示控件(GridView、 DataList、 DetailsView、 FormView、 Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```
<div>
<asp:Label Runat="server" ID="resultLabel" /><br />
<asp:Button Runat="server" ID="triggerButton"
Text="Click Me"
OnClick="triggerButton_Click" />
</div>
```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click()中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```
void triggerButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = "Button clicked!";
}
```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。



图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中)快速查看一下浏览器接收到的代码。<form>部分应如下所示：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE="
/>
</div>
<div>
<span id="resultLabel"></span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION"
id="__EVENTVALIDATION"
value="/wEWAgK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />
</div>
```

```
</form>
```

Web 服务器控件生成了 HTML , <span>和<input>分别代表<asp:Label>和<asp:Button>。还有一个名为 VIEWSTATE 的<input type="hidden">字段 , 把前面提到的窗体状态封装起来。在窗体传送回服务器以重新创建 UI , 以及跟踪改变时使用这些信息。注意<form>元素已经进行了配置 , 通过 HTTP POST 操作(在 method 中指定)把数据传送到 Default.aspx(在 action 中指定) , 它还被赋予了一个名称 form1。

在单击按钮 , 查看文本后 , 可再次浏览源 HTML(下面添加了必要的空格 , 使代码比较清晰) :

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHRvbIB
jbGlja2VkiWRkZN3zQXZqDnF2ddEBw4Kj7MEqqj9pJ" />
</div>
<div>
<span id="resultLabel">Button clicked!</span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAglL3MPHAwLHpP+yCwFFfz4kBL6+KP1xjB0lgrAageely" />
</div>
</form>
```

这次 viewstate 的值包含比较多的信息 , 因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上 , 这可能是一个非常长的字符串 , 但这是由系统在后台完成的 , 我们几乎可以不考虑状态管理 , 只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时 , 可以禁用不需要保留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态 , 以提高性能 , 就可以禁用整个页面的 Viewstate。

注意 :

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译 , 把 Default.aspx.cs 中的文本"Button clicked!"改为其他内容 , 保存文件 , 再次单击按钮。Web 页面上的文本会做相应的改变。

## 1. 控件面板

本节介绍可用控件 , 之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别 , 如图 37-6 所示。

注意，在控件的描述中使用了“属性”-- ASP.NET 代码中使用的属性与它同名。这里的引用并不完整，许多控件和属性都没有介绍，只介绍了最常用的属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation and Login 和 WebParts 类别在第 38 章介绍，AJAX 扩展控件在第 39 章介绍，Reporting 控件可以在 Web 页面上报告信息，包括 Crystal Reports，本书不讨论。



图 37-6

### (1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI.WebControls.WebControl，而 System.Web.UI.WebControls.WebControl 又继承了 System.Web.UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类，而该基类又最终派生于 Control。因此，Web 服务器控件有许多共同的属性和事件，如果需要，就可以使用这些属性和事件。这里不可能介绍所有的元素，只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式，这是很容易控制的，例如属性 ForeColor、BackColor、Font 等，也可以使用 CSS(Cascading Style Sheet)类来控制。此时，应在一个独立的文件中，把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括：Width 和 Height，用于设置控件的大小；AccessKey 和 TabIndex，便于用户的交互操作；Enabled，设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件，在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件，使用 Parent 可以访问控件的容器。

对于事件，最常用的是继承来的 Load 事件，它执行控件的初始化，PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多，下一章将详细介绍它们，尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控件	说 明
Label	显示简单文本，使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据，Text Changed 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮)就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本，Click 事件用于响应单击(服

	务器回送是自动的)。也可以使用 Command 事件响应单击 , 该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同 , 但把按钮显示为超链接

(续表)

控件	说明
ImageButton	显示一个图像 , 该图像放大一倍作为一个可单击的按钮 , 其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地 , 用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像 , 用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件 , 如果在链接后要执行其他处理 , 就应使用 LinkButton
DropDownList	允许用户选择一个列表项 , 可以直接从列表中选择 , 也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类) , SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变 , 这个控件也有 AutoPostBack 属性 , 所以选项的改变会触发一个回送操作
ListBox	允许用户从列表中选择一个或多个列表。把 SelectionMode 设置为 Multiple 或 Single , 可以确定一次选择多少个选项 , Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中 , 与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送 , CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同 , 例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下 , 它们都组合在一个组中 , 其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同
RadioButtonList	创建一组单选按钮 , 在这个组中 , 一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用 , 如果图像加载失败 , 由 AlternateText 提供对应的文本
ImageMap	类似于 Image , 但在用户单击图像中的一个或多个热区时 , 可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供 , 例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、 TableRow 和 TableCell , 或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样 , 这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同 , 这个控件有一个 Click 事件 , 用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控件	说明

HiddenField	用于提供隐藏的字段，以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能，但没有样式属性，只有一个 Text 属性(因为它派生于 Control，而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性，但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.DateTime)来访问由用户选择的日期和月份，并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后，显示另一个图像。使用 Advertisement-File 属性指定描述图像的 XML 文件，AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮，以选择要上传的文件。用户选择了文件之后，就可以使用 HasFile 属性确定是否选择了文件，然后使用后台代码中的 SaveAs()方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤，按顺序或不按顺序显示给用户，并依赖此控件来维护状态
Xml	这是一个更复杂的文本显示控件，用于显示用 XSLT 样式表传输的 XML 内容，这些 XML 内容是使用 Document、DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的，XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件，每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定，如果视图改变了(可能因为单击了当前视图上的 Next 链接)，就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出，但可以方便地把其他控件组合在一起，或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器，类似于 PlaceHolder，但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View，使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一起高速缓存的 Web 页面，这是一个与 ASP.NET 高速缓存相关的高级主题，本书不涉及
Localize	与 Literal 相同，但允许使用项目资源指定要在不同区域显示的文本，使文本本地化

## (2) 数据 Web 服务器控件

数据 Web 服务器控件分为两类：

数据源控件(SqlDataSource、 AccessDataSource、 ObjectDataSource、 XmlDataSource、 和 SiteMapDataSource)

数据显示控件(GridView、 DataList、 DetailsView、 FormView、 Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

## 48.2 Syndication 阅读器

把一些代码添加到 Default.aspx 中，就可以把事件处理程序链接到按钮上：

```
<div>
<asp:Label Runat="server" ID="resultLabel" /><br />
<asp:Button Runat="server" ID="triggerButton"
Text="Click Me"
OnClick="triggerButton_Click" />
</div>
```

其中 OnClick 属性告诉 ASP.NET 运行库，在生成窗体的代码模型时，把按钮的单击事件包装到 triggerButton\_Click 方法中。

修改 triggerButton\_Click() 中的代码（注意标签控件类型是从 ASP.NET 代码中推断出来的，所以可以直接在后台代码中使用）：

```
void triggerButton_Click(object sender, EventArgs
e)
{
resultLabel.Text = "Button clicked!";
}
```

下面准备运行它。不需要建立项目，只需保存所有的内容，把 Web 浏览器指向 Web 站点的地址。如果使用 IIS，这就很简单，因为我们知道指向的 URL。但本例使用内置的 Web 服务器，所以需要启动运行。最快捷的方式是按下 Ctrl+F5，启动服务器，打开一个浏览器，并指向指定的 URL。

在运行内置的 Web 服务器时，系统栏中会显示一个图标。双击这个图标，会看到 Web 服务器执行的过程，并可以在需要时停止它，如图 37-5 所示。

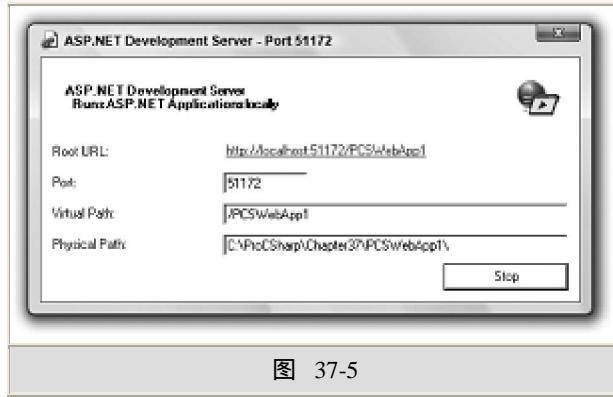


图 37-5

在图 37-5 中，可以看到 Web 服务器运行的端口和创建 Web 站点的 URL。

打开的浏览器应显示 Web 页面上的 Click Me 按钮。在单击这个按钮前，使用 Page | View Source (在 IE7 中) 快速查看一下浏览器接收到的代码。`<form>`部分应如下所示：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NWRkQw+7xydPDuBqgjPjjMHnYk872ZE="
/>
</div>
<div>
<span id="resultLabel"></span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION"
id="__EVENTVALIDATION"
value="/wEWAgK39qTFBwLHpP+yC4rCCl22/GGMaFwD017nokvyFZ8Q" />
</div>
</form>
```

Web 服务器控件生成了 HTML，`<span>` 和 `<input>` 分别代表 `<asp:Label>` 和 `<asp:Button>`。还有一个名为 VIEWSTATE 的 `<input type="hidden">` 字段，把前面提到的窗体状态封装起来。在窗体传回服务器以重新创建 UI，以及跟踪改变时使用这些信息。注意 `<form>` 元素已经进行了配置，通过 HTTP POST 操作(在 method 中指定)把数据传回 Default.aspx(在 action 中指定)，它还被赋予了一个名称 form1。

在单击按钮，查看文本后，可再次浏览源 HTML(下面添加了必要的空格，使代码比较清晰)：

```
<form method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHRvbib
jbGlja2VkiWRkZN3zQXZqDnF2ddEBw4Kj7MEqqj9pJ" />
</div>
<div>
<span id="resultLabel">Button clicked!</span><br />
<input type="submit" name="triggerButton" value="Click Me"
id="triggerButton" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgLl3MPHAwLHpP+yCwFFfz4kBL6+KP1xjB0lgrAageely" />
</div>
</form>
```

这次 viewstate 的值包含比较多的信息 ,因为 HTML 的结果不仅仅取决于 ASP.NET 页面的默认输出。在复杂的窗体上 ,这可能是一个非常长的字符串 ,但这是由系统在后台完成的 ,我们几乎可以不考虑状态管理 ,只要回送过程之间保存字段值即可。在 Viewstate 字符串过长时 ,可以禁用不需要保留状态信息的控件的 Viewstate。也可以禁用整个页面的 Viewstate。如果页面不需要在回送过程中保留状态 ,以提高性能 ,就可以禁用整个页面的 Viewstate。

注意 :

Viewstate 详见第 38 章。

为了说明不必手工进行任何编译 ,把 Default.aspx.cs 中的文本"Button clicked!"改为其他内容 ,保存文件 ,再次单击按钮。Web 页面上的文本会做相应的改变。

### 1. 控件面板

本节介绍可用控件 ,之后把它们组合到一个更丰富、更有趣的应用程序中。本节的内容对应于编辑 ASP.NET 页面时工具箱中的类别 ,如图 37-6 所示。

注意 ,在控件的描述中使用了"属性"-- ASP.NET 代码中使用的属性与它同名。这里的引用并不完整 ,许多控件和属性都没有介绍 ,只介绍了最常用的属性。本章介绍的控件在 Standard、Data 和 Validation 类别中。Navigation 和 Login 和 WebParts 类别在第 38 章介绍 ,AJAX 扩展控件在第 39 章介绍 ,Reporting 控件可以在 Web 页面上报告信息 ,包括 Crystal Reports ,本书不讨论。

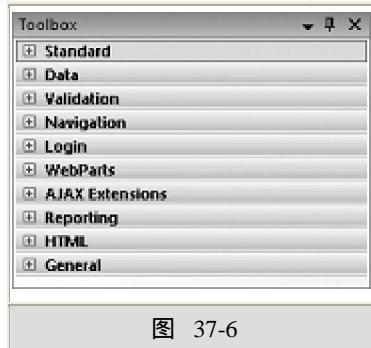


图 37-6

#### (1) 标准 Web 服务器控件

几乎所有的 Web 服务器控件(这个类别和其他类别)都继承了 System.Web.UI. WebControls. WebControl ,而 System.Web.UI.WebControls.WebControl 又继承了 System.Web. UI.Control。没有使用这个继承特性的 Web 服务器控件则直接派生于 Control 或更专门的基类 ,而该基类又最终派生于 Control。因此 ,Web 服务器控件有许多共同的属性和事件 ,如果需要 ,就可以使用这些属性和事件。这里不可能介绍所有的元素 ,只介绍 Web 服务器控件自身的属性和事件。

许多常用的继承属性主要用于处理显示格式 ,这是很容易控制的 ,例如属性 ForeColor、BackColor、Font 等 ,也可以使用 CSS(Cascading Style Sheet)类来控制。此时 ,应在一个独立的文件中 ,把字符串属性 CssClass 设置为 CSS 类的名称。还可以使用 CSS 属性窗口和样式管理窗口给 CSS 控件设置样式。其他属性包括 :Width 和 Height ,用于设置控件的大小 ;AccessKey 和 TabIndex ,便于用户的交互操作 ;Enabled ,设置控件的功能是否可以在 Web 窗体上使用。

一些控件还包含其他控件，在页面上建立控件层次结构。使用 Controls 属性就可以访问给定控件包含的控件，使用 Parent 可以访问控件的容器。

对于事件，最常用的是继承来的 Load 事件，它执行控件的初始化，PreRender 在控件输出 HTML 前进行最后一次修改。

可以使用的事件和属性很多，下一章将详细介绍它们，尤其是下一章将介绍更高级的样式设置技术。表 37-1 详细描述了标准 Web 服务器控件。

表 37-1

控件	说明
Label	显示简单文本，使用 Text 属性设置和编程修改显示的文本
TextBox	提供一个用户可以编辑的文本框。使用 Text 属性访问输入的数据，TextChanged 事件可处理回送的选择变化。如果要求进行自动回送(而不是使用按钮)就应把 AutoPostBack 属性设置为 true
Button	用户单击的标准按钮。Text 属性用于设置按钮上的文本，Click 事件用于响应单击(服务器回送是自动的)。也可以使用 Command 事件响应单击，该事件可以访问接收的附加属性 CommandName 和 CommandArgument
LinkButton	与 Button 相同，但把按钮显示为超链接

(续表)

控件	说明
ImageButton	显示一个图像，该图像放大一倍作为一个可单击的按钮，其属性和事件继承了 Button 和 Image
HyperLink	添加一个 HTML 超链接。用 NavigateUrl 设置目的地，用 Text 设置要显示的文本。也可以使用 ImageUrl 来指定要链接的图像，用 Target 指定要使用的浏览器窗口。这个控件没有非标准的事件，如果在链接后要执行其他处理，就应使用 LinkButton
DropDownList	允许用户选择一个列表项，可以直接从列表中选择，也可以键入前面的一或两个字母来选择。使用属性 Items 设置项目列表(这是一个包含 ListItem 对象的 ListItemCollection 类)，SelectedItem 和 SelectedIndex 属性可确定选择的内容。SelectedIndexChanged 事件可用于确定选项是否改变，这个控件也有 AutoPostBack 属性，所以选项的改变会触发一个回送操作
ListBox	允许用户从列表中选择一个或多个列表。把SelectionMode 设置为 Multiple 或 Single，可以确定一次选择多少个选项，Rows 确定要显示的选项个数。其他属性和事件与 DropDownList 控件相同
CheckBox	显示一个复选框。选择的状态存储在布尔属性 Checked 中，与复选框相关的文本存储在 Text 属性中。AutoPostBack 属性可以用于启动自动回送，CheckedChanged 事件则执行改变操作
CheckBoxList	创建一组复选框。属性和事件与其他列表控件相同，例如 DropDownList
RadioButton	显示一个单选按钮。一般情况下，它们都组合在一个组中，其中只有一个 RadioButton 控件是激活的。使用 GroupName 属性可以把 RadioButton 控件链接到一个组中。其他属性和事件与 CheckBox 相同

RadioButtonList	创建一组单选按钮，在这个组中，一次只能选择一个按钮。其属性和事件与其他列表控件相同
Image	显示一个图像。使用 ImageUrl 进行图像引用，如果图像加载失败，由 AlternateText 提供对应的文本
ImageMap	类似于 Image，但在用户单击图像中的一个或多个热区时，可以指定要触发的动作。要执行的动作可以是回送给服务器或重定向到另一个 URL 上。热区由派生于 HotSpot 的嵌入控件提供，例如 RectangleHotSpot 和 CircleHotSpot
Table	指定一个表。在设计期间可以使用它、 TableRow 和 TableCell，或者使用 TableRowCollection 类的 Rows 属性编程指定数据行。也可以在运行期间进行修改时使用这个属性。与 TableRow 和 TableCell 一样，这个控件有几个只能用于表格的格式属性
BulletedList	把一个选项列表格式化为一个项目符号列表。与其他列表控件不同，这个控件有一个 Click 事件，用于确定用户在回送期间单击了哪个选项。其他属性和事件与 DropDownList 相同

(续表)

控件	说 明
HiddenField	用于提供隐藏的字段，以存储不显示的值。这个控件可存储需要另一种存储机制才能发挥作用的设置。使用 Value 属性访问存储的值
Literal	执行与 Label 相同的功能，但没有样式属性，只有一个 Text 属性(因为它派生于 Control，而不是 WebControl)
Calendar	允许用户从图像日历中选择一个日期。这个控件有许多与格式相关的属性，但其基本功能是使用 SelectedDate 和 VisibleDate 属性(其类型是 System.DateTime)来访问由用户选择的日期和月份，并显示出来(总是包含 VisibleDate)。其关联的关键事件是 SelectionChanged。这个控件的回送是自动的
AdRotator	顺序显示几个图像。在每个服务器循环后，显示另一个图像。使用 Advertisement-File 属性指定描述图像的 XML 文件，AdCreated 事件在每个图像发回之前执行处理操作。也可以使用 Target 属性在单击一个图像时命名一个要打开的窗口
FileUpload	这个控件给用户显示一个文本框和一个 Browse 按钮，以选择要上传的文件。用户选择了文件之后，就可以使用 HasFile 属性确定是否选择了文件，然后使用后台代码中的 SaveAs()方法执行文件的上传
Wizard	这个高级控件用于简化用户在几个页面中输入数据的常见任务。可以给向导添加多个步骤，按顺序或不按顺序显示给用户，并依赖此控件来维护状态
Xml	这是一个更复杂的文本显示控件，用于显示用 XSLT 样式表传输的 XML 内容，这些 XML 内容是使用 Document、DocumentContent 或 DocumentSource 属性中的一个设置(取决于原始 XML 的格式)的，XSLT 样式表(可选)是使用 Transform 或 TransformSource 来设置的
MultiView	这个控件包含一个或多个 View 控件，每次只显示一个 View 控件。当前显示的视图用 ActiveViewIndex 指定，如果视图改变了(可能因为单击了当前视图上的 Next 链接)，就可以使用 ActiveViewChanged 事件检测出来
Panel	添加其他控件的容器。可以使用 HorizontalAlign 和 Wrap 指定内容如何安排
PlaceHolder	这个控件不显示任何输出，但可以方便地把其他控件组合在一起，或者用编程的方式把控件添加到给定的位置。被包含的控件可以使用 Controls 属性来访问
View	控件的容器，类似于 PlaceHolder，但主要用作 MultiView 的子控件。使用 Visible 属性可以指定是否显示给定的 View，使用 Activate 和 Deactivate 事件检测激活状态的变化
Substitution	指定一组不与其他输出一起高速缓存的 Web 页面，这是一个与 ASP.NET 高速缓存

	相关的高级主题，本书不涉及
Localize	与 Literal 相同，但允许使用项目资源指定要在不同区域显示的文本，使文本本地化

## (2) 数据 Web 服务器控件

数据 Web 服务器控件分为两类：

数据源控件(SqlDataSource、 AccessDataSource、 ObjectDataSource、 XmlDataSource、 和 SiteMapDataSource)

数据显示控件(GridView、 DataList、 DetailsView、 FormView、 Repeater 和 ReportViewer)

一般情况下，应把一个数据源控件(不可见)放在页面上，以链接数据源；然后添加一个绑定到数据源控件的数据显示控件，来显示该数据。一些更高级的数据显示控件，如 GridView，还可以编辑数据。

所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。这些类的方法，如 GetView()(或 GetHierarchicalView())，可以访问内部数据视图，还可以设置样式。

### 48.3 提供 SyndicationFeed

Office 系统已经推出了多年，所以读者很熟悉 VBA 代码，在已有的应用程序中还使用了 VBA。在 VSTO 解决方案中可以重写 VBA 代码，但这并不总是切合实际。看到 VSTO 的功能后，读者可能希望用托管的 VSTO 代码替代已有的 VBA 功能，或者添加新功能。

VSTO 允许给 VBA 代码提供 VSTO 功能，以实现上述任务。为此，必须执行几个步骤，才能给 VBA 代码提供 COM 接口。这 9 步如下所示，还列出了下载代码的 ExcelVBAInterop 项目中的示例代码和屏幕图：

(1) 在开始给 VBA 提供 VSTO 代码之前，必须有一个包含 VBA 项目的文档。为了便于开发，最好在开始之前启动文档中的宏。接着，在 VSTO 中创建一个文档级的定制时，把该文档作为自己解决方案中文档的起点。

(2) 有了这个起点后，就可以用通常的方式编写访问应用程序和/或文档的代码了。这些代码不能通过 VSTO 项目访问，因为后面要提供一个 VBA 接口。所以应创建 VBA 可以调用的方法，例如：

```
public partial class ThisWorkbook :  
ExcelVBAInterop.IThisWorkbook  
{  
...  
public void NameSheet()  
{  
NamingDialog dlg = new NamingDialog();  
if (dlg.ShowDialog() == DialogResult.OK)  
{
```

```
((Excel.Worksheet)this.ActiveSheet).Name =  
dlg.SheetName;  
}  
}  
}
```

**提示：**

这些代码使用一个简单的定制对话框，稍后介绍它。这个对话框允许用户输入一个字符串，选择是否在表名中包含当前日期。

(3) 必须重写 GetAutomationObject()方法，为 VBA 代码的自动执行返回正确的对象，如下所示：

```
public partial class ThisWorkbook :  
ExcelVBAInterop.IThisWorkbook  
{  
...  
protected override object GetAutomationObject()  
{  
return this;  
}  
}
```

(4) 因为 COM 系统通过接口来工作，所以必须通过接口提供要调用的方法。最简单的方法是右击代码，选择 Refactor | Extract Interface，接着选择要在接口中提供的方法，向导会完成剩余的工作，如图 40-16 所示。

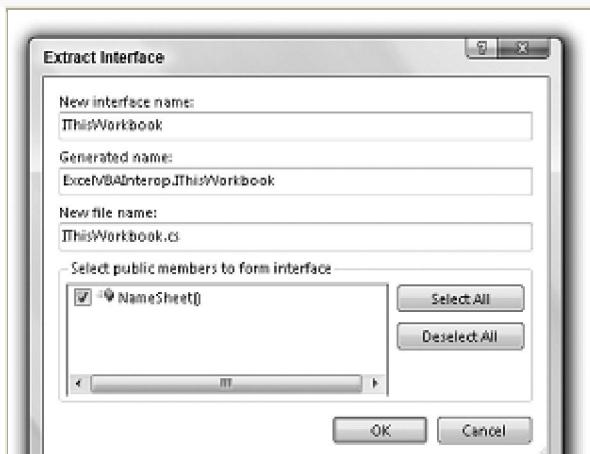


图 40-16

(5) 还必须把 System.Runtime.InteropServices 命名空间中的属性添加到类中，使类显示在 COM 上（参见第 24 章）：

```
using System.Runtime.InteropServices;  
namespace ExcelVBAInterop  
{
```

```
[ComVisible(true)]  
[ClassInterface(ClassInterfaceType.None)]  
public partial class ThisWorkbook :  
    ExcelVBAInterop.IThisWorkbook  
{  
    ...  
}
```

(6) 生成的接口还需要 ComVisible 属性，该接口必须是公共的：

```
using System.Runtime.InteropServices;  
namespace ExcelVBAInterop  
{  
    [ComVisible(true)]  
    public interface IThisWorkbook  
    {  
        void NameSheet();  
    }  
}
```

(7) 把文档的 ReferenceAssemblyFromVbaProject 属性改为 true，如图 40-17 所示。如果文档不包含 VBA 代码，就不能修改这个属性。改变它时，会接收到一个警告，说明项目运行时，添加到项目中的 VBA 代码会丢失，所以应备份修改的 VBA 代码。

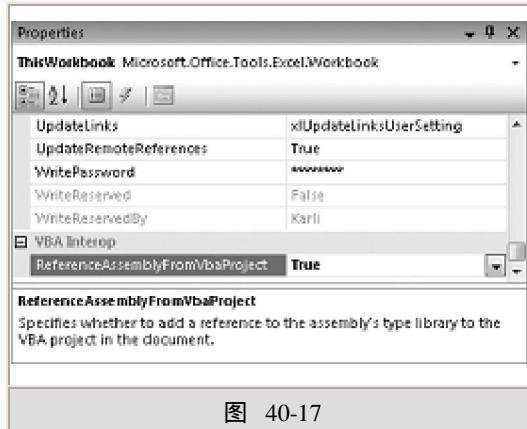


图 40-17

(8) 剩下的就是对文档中 VBA 代码的修改了。可以在此时运行项目，运行项目时可以通过 Developer 选项卡或按下 Alt+F11，来访问 VBA 代码。首先需要添加的是一个允许 VBA 访问 VSTO 代码的属性，如下所示(按添加了命名空间限制符的名称引用类)：

```
Property Get VSTOAssembly() As  
    ExcelVBAInterop.ThisWorkbook  
    Set VSTOAssembly = GetManagedClass(Me)  
End Property
```

(9) 接着就可以通过这个属性调用 VSTO 方法了：

```
Public Sub RenameSheet()
    VSTOAssembly.NameSheet
End Sub
```

完成了这些步骤后，就可以添加代码，调用接口上的方法，或手工调用它，如图 40-18 所示。

如果代码包含一个 UI，如本例所示，就会显示该 UI，且可以使用它。示例项目中的 UI 如图 40-19 所示。

#### 48.4 小结

本章学习了如何使用 VSTO 为 Office 产品创建托管的解决方案。

在本章的第一部分，介绍了 VSTO 项目的一般结构和可以创建的项目类型，还探讨了便于 VSTO 编程的特性。

接下来详细论述了 VSTO 解决方案中的一些特性，讨论了如何与 Office 对象模型通信，介绍了 VSTO 中的命名空间和类型，陈述了如何使用这些类型实现各种功能。之后研究了 VSTO 项目的一些编码特性，以及如何使用这些特性获得希望的结果。

然后，进行了一些实践。我们学习了如何在 Office 应用程序中管理插件，如何与 Office 对象模型交互操作，如何用 ribbon 菜单、任务面板和动作面板定制应用程序的 UI。

接着开发了一个示例应用程序，演示了前面学习的 UI 和交互操作技术。这个示例包含许多代码，还包含有用的技巧，例如如何在多个 Word 文档窗口中管理任务面板。

最后介绍了与 VBA 代码的交互操作。这部分介绍了如何通过 COM 交互操作性把托管代码提供给 VBA，并用另一个示例演示了这些技术。