HZ BOOKS

原创
精品系列

ASP.NET 2.0
网站开发技术详解

方 兵 编著

附光盘

机械工业出版社
China Machine Press

阜阳师范学院图书馆
藏书

ASP.NET 2.0
网站开发技术详解

方兵 编著

机械工业出版社
China Machine Press

本书主要介绍 ASP .NET 2.0 技术，内容涵盖网站开发过程中涉及的所有必要的技术，包括网站的规划设计、数据库技术、网站实现时所用技术以及网站的管理布置等。着重讲述一些设计和开发原则、技巧以及一些容易引起混淆的概念和技术。

本书阐述系统全面，条理清晰，理论与实例并举，适合作为相关专业学生及工程技术人员的参考书。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

ASP .NET 2.0 网站开发技术详解/方兵编著．-北京：机械工业出版社，2007.8

ISBN 978-7-111-21889-0

Ⅰ.A…　Ⅱ.方…　Ⅲ.主页制作—程序设计　Ⅳ.TP393.092

中国版本图书馆 CIP 数据核字（2007）第 108370 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：王春华
北京京北制版厂印刷·新华书店北京发行所发行
2007 年 7 月第 1 版第 1 次印刷
186mm×240mm·20 印张
定价：45.00 元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

前　言

微软的.NET战略推出已经有6年多了，从.NET 1.0到1.1，再到现在的2.0，.NET正在走入软件业的主流，尽管目前它还只是广泛应用于服务器上，但显而易见，随着年底.NET 3.0的推出，.NET将从服务器领域走向客户端应用，Windows平台下的开发将由.NET统治。

什么是.NET

.NET是一个应用于一系列技术上的商标

微软将.NET视为数字化未来的一个远景和平台。如果更具体、更准确地看待这种创新，则是把.NET视为一个商标，一个微软已经应用于数种不同技术上的商标。这些技术有些是全新的，提供新的服务和新的可能性；另一些则允许我们以最新的方式来创建我们今天已经知道的各类Windows应用程序。当然，也有一些.NET家族成员只不过是挂着.NET牌子的现有技术的新版本而已。

.NET是软件成为一种服务的转移

.NET在这个方面的意义是最被广泛接受和理解的。“软件就是服务”的理念最初是在1997年左右由Oracle的CEO Larry Ellison以及SUN的CEO Scott McNealy在网络计算机的概念大行其道的时候提出的。不过Oracle和SUN并没有真正将这个概念变为现实，他们的视角更多地集中于资源集中化方面。不过，当初听到Ellison和McNealy这番见解的公司——包括微软，也认识到了这种见解道出了软件产业面临的一个巨大改变，.NET则是微软对这种概念和这种变化作出的自己的反应。

.NET是一个新的编程模型——也就是Internet平台

微软正在趋向于将.NET看作一个系统。在表面上，它包含了两种不同的编程模型：一个是Web服务编程模型，另一个是系统编程模型。

微软开始把.NET系统编程模型作为.NET整体的一个组成部分，计划最终以此代替现有的组件对象模型（Component Object Model，COM）以及Windows应用程序编程接口（API），这个现在还没有最终正式定名的模型使用一系列新的基础类。

.NET中最重要的新技术是Web Services。如其名称所示，Web Services提供了某些功能，我们可以通过网络加以调用。大多数拥有.NET商标的技术都可以在某种程度上直接支持Web Services。然而.NET绝非仅仅是Web Services而已，微软置于.NET商标下的技术包括：

.NET Framework：它包括通用语言运行时（Common Language Runtime，CLR）和.NET框架类库。CLR是构建一系列新应用程序的标准基础，.NET类库则为许多基于CLR的应用程序提供一个新的标准开发环境。这个类库包含的技术有：ASP.NET，最新一代的ASP（Active Server Pages）技术；ADO.NET，最新一代的ADO（ActiveX Data Objects）技术；新的WINFX编程模型，不久之前被命名为.NET Framework 3.0。这个编程模型包括：WPF，最新一代的图像引擎技术；WCF，架构于Web Services之上的通信交流框架；WWF，采用.NET以及其他一系列新技术构建起来的工

作流引擎；以及对“构建和使用 Web Services”的其他支持等等。微软还发行了一个 .NET Framework 精简版，名为 .NET Compact Framework，用于小型设备，如个人数字助理（personal digital assistant，PDA）上。除此之外，还可以在 XBOX 360 的 XNA 中、微软的 Smart Watch 等等产品上看到 .NET 的影子。

Visual Studio .NET：支持多种可使用 .NET Framework 的编程语言，包括 Visual Basic；一个增强版的 C++；一个基于 .NET 的 Java 替代语言 J#，以及一个为 .NET Framework 量身打造的全新语言 C#。

.NET My Services：一组服务，允许用户存储和访问位于互联网中服务器上的个人信息，例如日程表和地址簿等。这些服务还提供诸如认证（Authentication）这样的通用功能，使客户能够证明自己的身份；还提供了一个“向不同设备上的客户发送消息”的方式。

.NET Enterprise servers：一系列软件服务器，包括 Exchange Server 2003、SharePoint Server 2003、Project Server 2003、BizTalk Server 2006、Application Center 2000、Commerce Server 2000、Host Integration Server 2000、SQL Server 2005 等等。除了几个称为 2003、2005 或 2006 的产品外，其他的很大程度上与这里说的 .NET 技术没有什么关联，但是显而易见，在未来的版本当中，它们将全部基于 .NET 技术构建，上面几个称 2003 的版本已经证明了这一点。

.NET 的特点

高效率开发

.NET Framework 为我们提供的这个庞大而又结构清晰的类型，使我们的编程变得异常轻松，另外，自动垃圾回收机制等一系列新的特性，可以让我们的程序员把更多的精力放在考虑如何实现客户所需要的业务逻辑上，而不为计算机在控制上内存如何分派之类的事情头痛。甚至无论是开发哪一种应用程序，无论是 C/S、B/S，还是智能设备亦或是数据库编程，你都可以使用最熟悉的一种编程语言，而不需要去学习诸如 C++、ASP、SQL 等各不相同的多用语言。.NET 还带来了多种语言之间的无缝集成，例如一个系统可以同时采用多种编程语言来开发，VB .NET 编写的类可以方便地再用 C#继承。这些都大幅度地提高了我们的开发效率。

多平台特性

尽管到目前为止 .NET 应用程序还只能运行于 Windows 平台上，但 .NET 天生就为跨平台应用做好了准备，据我们所知，微软自己还有第三方开发商都已经在为 .NET 程序在 Unix、OS2、Linux 等系统上运行而工作着（如开源项目 Mono）。我们还可以看到 .NET 应用程序将可以运行在 PDA 甚至手机上，以及 Vista 上将要出现的 XAML，将使我们进行应用程序开发的时候，不需要再考虑是 B/S 架构还是 C/S 架构。不久的将来，我们就可以只关心我们的应用程序如何满足客户的需求而不用考虑基于何种平台来开发。

无接触部署

借助于 .NET 的反射特性，.NET 应用程序可以精确地描述自身。这就使得无接触部署成为可能，.NET 应用程序无需在注册表中储存信息，只需简单的 XCOPY 便可正确地在用户的机器上运行，这将会使企业的部署成本大为降低。而在 .NET 2.0 中，Click Once 技术使我们的应用程序部署前所未有地简单，使 C/S 应用程序的部署不比 B/S 网页更困难，然而，C/S 将比 B/S 带来更好的用户体验特性。

试读结束，需要全本PDF请购买 www.ertongbook.com

消除 Dll Hell

同样是基于.NET 的反射特性，每一个应用程序都可以清楚地知道自己需要使用哪一个 Dll，同一个 Dll 的不同版本可以彼此和平共处，从而彻底消除让我们头痛的 Dll Hell。

可信赖计算

长期以来，微软系统的安全性问题一直备受诟病。比尔·盖茨决定改变这种现状。在.NET 中，这种安全性的考虑直接放到了代码级。通过一系列的技术，如代码访问安全（Code Access Security）、基于角色的安全、强名称（Strong Name）、权限和权限集等，最大限度地保证了系统的安全性。

.NET Framework 体系结构

.NET 是分层的、模块化的，并且是层次结构化的。.NET Framewok 的每一层都是一个抽象层。其中，.NET 语言是顶层，也是最为抽象的一层。而通用语言运行时则位于底层，它是最不抽象、最靠近本地环境的一层。这一点很重要，因为通用语言运行时需要与操作环境紧密合作来管理.NET 应用程序。.NET Framework 被分成了多个模块，每个模块都有它们各自特定的责任。最后由于高层只从底层请求服务，所以.NET 又是层次结构化的，如图 1 所示。

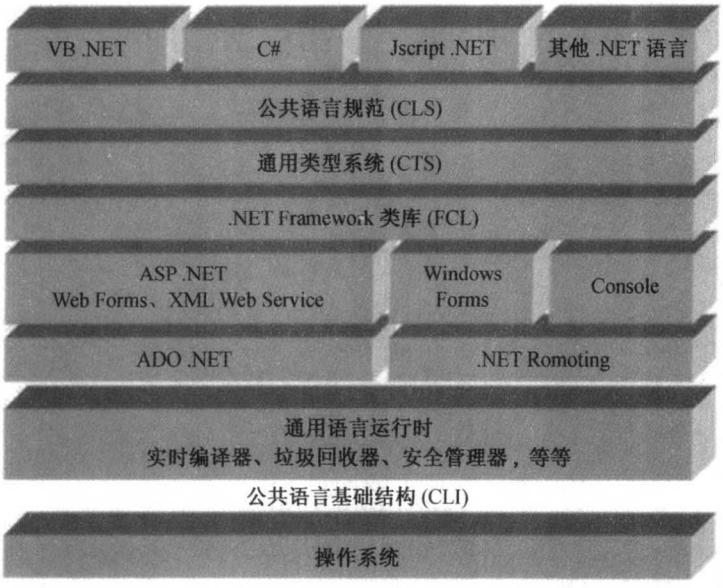

图 1　.NET Framework 体系结构

.NET 的未来

2006 年底，微软推出了新一代的操作系统 Vista，.NET 第一次上升到操作系统的核心 API 这样一个层面的地位，.NET 3.0（以前叫做 WinFX）与操作系统紧密结合，它改变了原有的编程机制。Vista 生成器最终会跟以前的 Win32 API（Application Programming Interface，应用程序编程接口）进行分离，取而代之的是可管理的 WinFX，而 WinFX 就是微软继 DOS、Win16、Win32 之后推出

的第四代 API。据外刊报道，以前利用 Win32 API 开发的软件，在微软承诺的维护期之后将不能运行。现在我们看到的是成千上万的 Win32 开发人员，我们在 Windows 上运行的软件几乎全部是使用 Win32 开发的。将来，我们看到的会是成千上万的 .NET 开发人员，在 Windows 上运行的软件将几乎全都是使用 .NET 开发的。

在非 PC 上，我们也将看到 .NET 出现在 PDA、手表等设备上，如果你对 .NET 了解得更多一些的话，还将可以在 XBOX360、电视机顶盒甚至机器人身上找到它的身影。

本书所讨论的 ASP .NET

本书所讨论的 .NET 仅仅是 .NET 中很小，目前却应用最广泛的一部分——ASP .NET，顺带还会有一些 ADO .NET 的内容，但是对于 .NET 的 Windows 开发、Web Services、.NET Remoting 等等的开发不做涉及。

另外，本书并不针对 .NET 的初学者，所以并不讲述 C#语言或者 ASP .NET 的语法之类的东西，而是针对有一定经验的开发人员，着重讲述一些设计和开发原则、技巧以及一些容易引起混淆的概念和技术。

为了保证定义的精确性，本书中部分概念的定义直接来自 MSDN Library，部分代码来自 PetShop 4.0。

目　录

第1章 .NET 2.0的新特性

相对于.NET 1.1来说，.NET 2.0有很多激动人心的新特性，比如泛型、可空类型，以及.NET Framework中新增的数量庞大的类库，等等。对于我们来说，有相当多的新知识需要去学习，然而幸运的是，根据所从事的开发方向的不同，我们只需要学习了解其中一部分就可以了。既然本书是为使用C#进行ASP.NET开发的人员准备的，那我们将只讲述相关的新特性，包括C#、ADO.NET和ASP.NET的内容。

1.1 C#语言的新特性

1.1.1 泛型

提到C# 2.0，很自然地首先想到的一个新特性就是泛型，因为这是我们最为津津乐道而且应用最广泛的新特性之一。其实泛型并不是一个新概念，早在C++时代，我们便已经熟知模板类。从某个角度来说，你可以认为泛型脱胎于模板类。当然，如果你不了解C++或者不了解模板类，那也没有关系，我们直接从泛型开始。

在没有泛型之前，我们常常要做的一件事情是类型的强制转换，请考虑如下的一段代码：

```
public object GetBigObject(object A,object B)
{
    ......
}
public void Main()
{
    int a=1;
    int b=2;
    int result=(int)GetBigObject(a,b);
}
```

上面这段代码尽管可能没有什么实际意义，但是我们可以发现，在实际工作当中常常会看到类似的代码。编程是严谨的，在开发时经常使用这样的代码会带来很多问题，比如：

（1）性能问题，在数据类型的这种转换当中几乎不可避免地要涉及到装箱拆箱的操作，而这样就会带来性能上的损失。

（2）类型转换错误，这种问题尤其是在调用别人编写的方法时容易出现，因为你常常不知道人家给你返回来的结果是什么，这个时候做强制转换往往会出错。

（3）对于类型转换错误，错误的发现被延迟到运行时才能够出现，而在编译阶段无法发现，这使得发现问题的时间被大大拖延，软件Bug往往会因此被漏过。

（4）这一点是承接前两点而来的，对于为别人提供接口的开发者而言，他无法告知调用者自己返回或者接收的数据是什么类型的。这种状况常常出现在我们使用Hashtable或者ArrayList来传递数据的时候，因为我们往往不知道这个Hashtable或者ArrayList里面装的是什么。当然你可以对它们作一个封装，可是自己开发一个集合类所需的开销是巨大的。在做接口开发的时候，我们的一

个原则是接口一定要明晰，要清楚地让调用者知道他应当向接口传入什么样的数据，接口又会返回什么样的数据。

由此，在C# 2.0中产生了泛型的概念。使用泛型，我们可以轻松解决以上问题。通过使用泛型类型参数T，可以编写其他客户端代码能够使用的单个类，而不致引入运行时强制转换或装箱操作的成本或风险。它使得：

（1）不再需要进行装箱拆箱的操作，从而显著提高性能。根据统计，在集合中的元素数多于百个时，避免装箱拆箱操作节省下的内存将远远超过为了编译泛型类而消耗的内存。

（2）有效避免了数据类型转换错误。

（3）在编译阶段就能够发现数据类型转换上出现的错误，从而及早解决问题。

（4）接口明晰，输入输出非常明确。

例如，我们编写如下的泛型方法：

```
public T GetBigObject(T A,T B)
{
    ……
}
```

这样，我们在调用的时候将不再需要进行类型的强制转换：

```
public void Main()
{
    int a=1;
    int b=2;
    int result=GetBigObject(a,b);
}
```

可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。

然而笔者认为，如果不是开发重用性非常强的模块的话，不需要自己设计和编写自己的泛型类，.NET Framework自身已经提供了相当数量的泛型类，利用这些泛型类，就可以很方便地使用泛型带来的强大特性了，通常不需要再去定义新的泛型类。

泛型最常见的用途是创建集合类，.NET Framework类库已经在System. Collections. Generic命名空间中提供了相当数量的泛型集合，其中最常用的就是Dictionary、List、Queue和Stack。应尽可能地使用这些类来代替普通的类，例如使用Dictionary来代替Hashtable，用List来代替ArrayList，等等。

建议在编写泛型类或泛型方法之类的对象时，最好为泛型参数T加上约束。在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用where上下文关键字指定的。如：

```
class EmployeeList<T>where T:Employee,IEmployee,System. IComparable<T>,new()
{
    //...
}
```

因为如果不做约束的话，很可能在泛型的内部又要做强制转换，这样，就会重新陷入没有泛型时我们所遇见的那一系列问题当中了。很难想象，如果不做约束，仅对Object类型的数据能做些什么操作。

下面我们列出6种类型的约束，见表1-1。

试读结束，需要全本PDF请购买 www.ertongbook.com

表 1-1 .NET 泛型的约束

约　束	说　明
T：结构	类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型
T：类	类型参数必须是引用类型，包括任何类、接口、委托或数组类型
T：new( )	类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，new( )约束必须最后指定
T：<基类名>	类型参数必须是指定的基类或派生自指定的基类
T：<接口名称>	类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的
T：U	为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。这称为裸类型约束

在使用泛型的时候，需要注意，不要为了使用泛型而使用泛型，除非在不得已的时候，否则就不要使用泛型，请考虑如下的代码：

```
public T ModifyData(T dataToModify)where T:DataSet
{
    ......
    return dataToModify;
}
```

实际上，上面这段代码没有任何意义，纯粹是为了使用泛型而使用泛型，因为 T 本身是引用类型，在使用的时候完全不需要强制类型转换，上面的方法如果改用下面的代码会更好一些：

```
public void ModifyData(DataSet dataToModify)
{
    ......
}
```

1.1.2 可空类型

可空类型允许变量包含未定义的值。在使用数据库和其他可能包含未含有具体值的元素的数据结构时，可以使用可空类型。对于我们使用数据库作为后端的应用程序来说，这一点尤其有用。举个例子，对于人员信息来说，姓名可能是必不可少的，而其他信息比如年龄、性别等用户可以不填，也就是说在数据库中可以为空。

从前在我们为人员信息生成了一个类型化的数据集之后，要从中读取一个可以为空的信息时，我们必须首先判断它是否为空，例如：

```
if(PersonInfoRow.IsAgeNull() == false)
{
    Label1.Text = personInfoRow.Age;
}
```

而要将该字段设为空，也必须调用专用的方法，如：

```
if(Text1.Text.Trim() == string.Empty)
{
    personInfoRow.Age = Text1.Text;
}
```

这种限制使得我们在开发过程中要相当小心，并且需要使用相当多的或许是不必要的代码来防止错误发生。在引入了可空类型之后，我们的开发就变得轻松多了。所要做的只是在类型声明后面加上一个？号：

```
int? number;
```

之后，便可以放心地将空值赋给它了：

```
number = null;
```

如果探究可空类型的原理，我们就会发现它其实是泛型的杰作。可空类型是 System. Nullable 结构的实例。可空类型可以表示其基础值类型正常范围内的值，再加上一个 null 值。例如，Nullable < Int32 >，读作“可空的 Int32”，可以被赋值为 -2 147 483 648 到 2 147 483 647 之间的任意值，也可以被赋值为 null 值。Nullable < bool > 可以被赋值为 true、false 或 null。过去，在需要使用三态数据的时候（三态数据在数据库里用得非常频繁，例如一个网站用户可以公开是男是女，也可以对自己的性别保密），我们常常需要自己定义一个枚举，现在使用 Nullable < bool > 就可以了。

在不想使用泛型的写法来写的时候，我们就可以用数据类型加上问号这种等效写法来写，如 Nullable < Int32 > 等效于 Int32? 或 int?，Nullable < bool > 等效于 bool?，等等。

在使用的时候要注意如下几点：

（1）因为可空类型可能为空，于是很多运算法则将不再简单适用，比如 A + B，如果 A 或者 B 当中有一个或者两个为可空类型，那么它们便不能简单相加，因为它们可能会为空，这时结果也就可能会为空，所以一定要先使用 HasValue 属性测试是否为空，或者使用 GetValueOrDefault 属性返回该基础类型所赋的值或默认值，再或者使用?? 运算符分配默认值，当前值为空的可空类型被赋值给非空类型时将应用该默认值，比如 int? x = null; int y = x?? -1;，但是切不可直接进行运算。

（2）可空类型可以强制转换为对应的基础类型，比如 int? x = 3; int y = (int)x;，但在转换之前一定要先测试是否不为空，如果为空的话就会出错。

（3）由于以上两点原因，建议不要随意使用可空类型，只在必要的时候才使用，使用时一定要小心谨慎，随时记得判断是否为空。

1.1.3 迭代器

迭代器是 C# 2.0 中的新功能。迭代器是方法、get 访问器或运算符，它使你能够在类或结构中支持 foreach 迭代，而不必实现整个 IEnumerable 接口。你只需提供一个迭代器，即可遍历类中的数据结构。当编译器检测到迭代器时，它将自动生成 IEnumerable 或 IEnumerable 接口的 Current、MoveNext 和 Dispose 方法。

这里要说明的一点是，实现迭代器有两种办法：

一种办法是需要创建迭代器的类型声明 IEnumerable 接口，然后实现其中的 GetEnumerator 方法，使用这种办法，编译器会自动生成 Current、MoveNext 和 Dispose 方法。然后在使用的时候，先调用 GetEnumerator()，然后使用 MoveNext 来轮询其中的项，用 Current 属性获取当前项。

另一种方法是不需要显示声明 IEnumerable 接口，而是直接在里面写一个返回 IEnumerable 接口的方法，不过这样做带来的一个问题就是只能使用简单迭代，也就是说只能使用 foreach 循环。请看下面的例子：

```
//声明集合:
public class SampleCollection
{
    public int[]items;

    public SampleCollection()
    {
        items = new int[5]{5,4,7,9,3};
    }
    public System. Collections. IEnumerable BuildCollection()
```

```
        {
            for(int i = 0;i < items.Length;i++)
            {
                yield return items[i];
            }
        }
    }

    class MainClass
    {
        static void Main()
        {
            SampleCollection col = new SampleCollection();

            //显示集合项目:
            System.Console.WriteLine("Values in the collection are:");
            foreach(int i in col.BuildCollection())
            {
                System.Console.Write(i + "");
            }
        }
    }
```

这种简单迭代的好处是使用起来比较方便，而且使用多个返回 IEnumerable 接口的方法可以使用不同的方式迭代。所以除非在相当必要的情况下，一般建议使用上述第二种办法创建迭代器。

1.1.4 匿名方法

与匿名方法相对的是命名方法。在平时开发当中，我们见到最多的是命名方法，在 .NET 2.0 之前，我们也只有命名方法的概念。命名方法最显著的特征是有方法签名，如下所示：

```
public System.Collections.IEnumerable BuildCollection()
    {
      ......
    }
```

因为有方法签名，于是我们可以在很多地方调用，也就是说可以实现代码复用，这是命名方法带给我们的一个非常便利的地方。

而匿名方法则不具有方法签名，不能在多个地方调用同一段代码，也就不能实现代码复用，但是它可以给我们带来另外一些便利：

（1）不必创建单独的方法，因此减少了实例化委托所需的编码系统开销。

（2）匿名方法中变量的作用范围不局限在方法内部，可以在不同的方法成员之间共享某个局部状态，这可能有一点儿难理解，我们来看下面的代码：

```
int n = 0;
Del d = delegate(){System.Console.WriteLine("Copy #:{0}", ++n);};
Del d = delegate(){System.Console.WriteLine("Copy #:{0}", --n);};
```

如果使用命名方法，除非将输入参数 n 以引用方式传入，否则将不能共享 n 的值。这里变量 n 被称为外部变量或捕获变量，与局部变量不同，外部变量的生命周期一直持续到引用该匿名方法的委托符合垃圾回收的条件为止。对 n 的引用是在创建该委托时捕获的。

（3）将代码段作为参数传递，要将代码块传递为委托参数，创建匿名方法则是惟一的方法。

例如：

```
//为单击事件创建处理程序
button1.Click += delegate(System.Object o, System.EventArgs e)
                 {System.Windows.Forms.MessageBox.Show("Click!");};
```

匿名方法有以上这些优点，同时也会有一些局限性，在使用时一定要先权衡得失，再决定使用哪一种方式：

（1）匿名方法的概念使用是和代理联系在一起的，如果不是在使用代理的地方，那么便用不上匿名方法。

（2）匿名方法因为没有方法签名，所以不能在多处调用同一段代码，也就不能实现代码复用，因此不适合匿名方法块中代码比较多或者需要重用的情况。

（3）在目标在块外部的匿名方法块内使用跳转语句（如 goto、break 或 continue）是错误的。在目标在块内部的匿名方法块外部使用跳转语句（如 goto、break 或 continue）也是错误的。

1.1.5 分部类型

分部类型定义允许将单个类型（比如某个类）拆分为多个文件。Visual Studio 设计器使用此功能将它生成的代码与用户代码分离。

单纯这么说可能不容易理解，那么可以看看在 ASP .NET 2.0 中 aspx 网页的代码文件，可以发现每个 .cs 文件中都有 partial 关键字，而你再找不到从前由开发环境自动生成的控件声明、事件注册等代码，这些就是分部类的功劳。具体的原理我们后面再讲。同样，也可以在 ADO .NET 的数据集设计、Windows 窗体开发、Web 服务包装等地方找到分部类的应用。

或许通常不需要在自己写的类中添加 partial 关键字来定义分部类，除非遇到如下情况：

（1）处理大型项目时，需要使一个类分布于多个独立文件中，以便让多位程序员同时对该类进行处理。

（2）需要扩展 Visual Studio 自动生成的类，比如说为 DataSet 扩展功能，等等。

如果使用分部类，下面这些原则要记清楚：

（1）分布类的生成不依赖于文件名，而是依赖于类名，所以分部类在不同文件中的类名一定要相同，而且一定都要有 partial 关键字，不要以为其中某一个有了这个关键字就可以了，那样编译是通不过的。

（2）要成为同一类型的各个部分的所有分部类型定义都必须在同一程序集和同一模块（.exe 或 .dll 文件）中进行定义。分部定义不能跨越多个模块。

（3）要成为同一类型的各个部分的所有分部类型定义都必须在同一个命名空间下，如果处在不同的命名空间下，就会编译成不同的类。

（4）和上面的命名空间相同原则类似，如果是嵌套类型做分部，那么它们所嵌套于的类型必须是同一个。尽管可以处于不同的文件中，这取决于它们所嵌套于的类型是否也是分部类型。

（5）不能在成为同一类型的各个部分中编写签名完全相同的方法、属性、字段之类的成员，例如，不能在不同的部分中都写一个 DoSomething（int number）的方法，除非它们能成为方法重载。

（6）类名和泛型类型参数在所有的分部类型定义中都必须匹配。泛型类型可以是分部的。每个分部声明都必须以相同的顺序使用相同的参数名。

（7）可以在成为同一类型的各个部分中使用不同的修饰关键字，如 internal、public、abstract

等，编译时这些关键字将合并，但是它们彼此不能冲突，例如，不能在某一个部分中声明为 public，而在另一个部分中声明为 internal。

（8）如果在某一个部分中声明该类型是从某一个类型派生出来的，那么在其他部门中，要么可以声明它们继承自同一个类型，要么干脆就不声明继承关系，这是因为 .NET 中规定类只能是单继承。但是可以在不同的部分中声明不同的接口，编译时它们将合并在一起，从而使该类型同时实现多个接口。在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。最终类型是所有部分在编译时的组合。

综上所述，在使用分部类型时，有许多需要协调的地方，就像我们在开发时的团队协作，都是需要彼此协调，不能发生冲突。

1.1.6 属性访问器可访问性

属性或索引器的 get 和 set 部分称为“访问器”。在从前的版本中，这些访问器具有相同的可见性或访问级别：其所属属性或索引器的可见性或访问级别。不过，有时限制对其中某个访问器的访问会很有用。通常是在保持 get 访问器可公开访问的情况下，限制 set 访问器的可访问性。例如：

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

在上面的示例中，名为 Name 的属性定义了一个 get 访问器和一个 set 访问器。get 访问器接受该属性本身的可访问性级别（在此示例中为 public），而对于 set 访问器，则通过对该访问器本身应用 protected 访问修饰符进行显式限制。

因为属性访问器的概念比较简单，所以没有什么太多需要注意的地方，只是要注意，使用访问器实现接口时，访问器不能具有访问修饰符。但是，如果使用一个访问器（如 get）实现接口，则另一个访问器可以具有访问修饰符。

1.1.7 其他新特性

C# 2.0 还有其他一系列的新特性，因为不是经常用，这里不再作详细描述，只作简单说明，如果读者有兴趣了解详细信息，请自行参看 SDK 文档。

1. 命名空间别名限定符 命名空间别名限定符（::）对访问命名空间成员提供了更多控制。global:: 别名允许访问可能被代码中的实体隐藏的根命名空间。

2. 静态类 若要声明那些包含不能实例化的静态方法的类，静态类就是一种安全而便利的方式。C# 1.2 版要求将类构造函数定义为私有的，以防止类被实例化。

3. 外部程序集别名 通过 extern 关键字的这种扩展用法引用包含在同一程序集中的同一组件的不同版本。

4. 委托中的协变和逆变 现在传递给委托的方法在返回类型和参数方面可以具有更大的灵活性。

5. 友元程序集 程序集可以提供对其他程序集的非公共类型的访问。

1.2 ADO .NET 的新特性

和 .NET 1.1 相比，ADO .NET 2.0 作了大幅度的改进，使其变得更方便、更智能、更兼容和可复用度更高。

1.2.1 托管提供程序的新特性

1. 服务器枚举 在 .NET 1.1 里想要实现服务器枚举是一件非常麻烦的事情，所以我们通常不做服务器枚举，而是要求用户手工输入连接字符串，最多也就是把计算机名、实例名、数据库名等分为几个部分让用户分别输入，然后再拼接成一个连接字符串而已，很容易想像，这样的做法是很不友好的。在 .NET 2.0 里，这样的状况得到了改善。SQL Server 2000 和 SQL Server 2005 均允许应用程序在当前的网络中查找 SQL Server 实例。SqlDataSourceEnumerator 类向应用程序开发人员公开此信息，提供包含所有可见服务器信息的 DataTable。这个返回的表包含网络上可用服务器实例的列表。

例如，使用下面的代码我们可以检索到网络中可用的服务器信息：

```
System.Data.Sql.SqlDataSourceEnumerator instance = System.Data.Sql.SqlDataSourceEnumerator.Instance
System.Data.DataTable dataTable = instance.GetDataSources();
```

GetDataSources 方法即是用来检索包含有关所有可见 SQL Server 2000 或 SQL Server 2005 实例的信息的。

通过方法调用返回的表包含表 1-2 所示，所有列均包含 string 值。

表 1-2 服务器枚举返回的数据结构

列	说 明
ServerName	服务器的名称
InstanceName	服务器实例的名称。如果服务器作为默认实例运行，则为空白
IsClustered	指示服务器是否属于群集
Version	服务器的版本（对于 SQL Server 2000，为 8.00.x，对于 SQL Server 2005，为 9.00.x）

2. 异步处理 在 .NET 2.0 中，对于数据操作新增加了一些用于异步处理的方法，比如 SqlCommand.BeginExecuteNonQuery 方法等，这些使得我们在系统处理大量的数据时不一定必须等待操作完成，关于这个异步处理，如果你原来就对 .NET 中的异步处理很熟悉的话，其实没有更多需要注意的地方。

3. 数据的批处理 如果了解 DataAdapter 的原理，你就会知道，在 .NET 1.1 当中，DataAdapter 在执行 Update 方法对数据进行更新时，对数据行进行遍历，并依次向数据库发送命令来执行。而在 .NET 2.0 中，DataAdapter 可以将 DataSet 或 DataTable 中的 INSERT、UPDATE 和 DELETE 操作分组发向服务器，从而减少与服务器的往返次数，以大大提高性能。

4. SQL Server 通知 允许 .NET Framework 应用程序向 SQL Server 发送命令，如果执行相同命令，生成的结果集与最初检索到的结果集不同，请求生成通知。这样在数据库数据发生更改时，应用程序就能够得到通知。我们可以通过如下 3 种方式来实现：

试读结束，需要全本PDF请购买 www.ertongbook.com