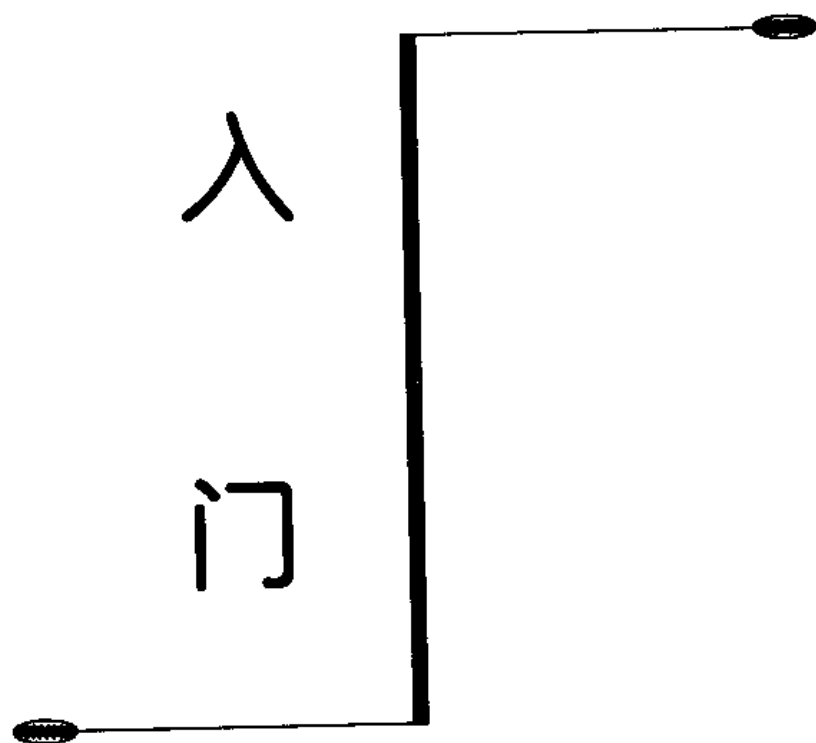


第 1 部分

入

门



第1章

VB.NET 快速入门

.NET 框架 (framework) 是一个用于创建、部署和运行服务及其他应用程序的环境 (emironment)。该环境涉及代码的重用及专业化、多语言开发、部署、管理和安全问题。.NET 框架将引导新一代软件, 这些软件将计算与通信相融合, 使开发者能够创造真正的与其他补充服务一体化的分布式应用程序。换句话说, 你现在可以创建 Web 服务, 如搜索引擎和债权计算器, 它将像我们了解的那样改变 Internet。这已不再是只与独立的 Web 站点或连接装置相关的问题了, 它是当今计算机和各种装置 (如掌上电脑、手表和移动电话) 彼此协作以创造出更好服务所面临的一个问题。例如, 调用某搜索引擎在你自己的应用程序中搜索、显示并处理结果。

传闻.NET 框架仅适用于构建 Web 站点, 但事实上并不是这样的。你同样可以用.NET 框架容易地创建出老式的 Windows 应用程序。如果 Microsoft 没有改变它的命名方式, 则.NET 框架就多少有些像 COM+ 2.0。虽然现在我们所使用的 .NET Framework 与 COM+ 差别很大, 但 NET 框架的确起源于 COM+。没有把它命名为 COM+ 2.0 也许是正确的。

虽然本书假定你已经了解了 .NET 框架。但在本章中仍旧提供了一条快速了解这些概念和术语的途径。尽管这些术语不是 VB.NET 所特有的, 但它们却是所有 .NET 编程语言所特有的, 因此如果打算在你所有的企业编程中都使用 VB.NET, 那么你就需要了解它们。

1.1 回顾编程概念

.NET 环境引入了许多编程概念, 这些概念甚至对于那些熟悉早期 Visual Basic 版本的人来说也非常新颖。事实上, 这些概念对于大多数 Windows 开发者来说都是很新颖的, 因为 VB.NET 这类的 .NET 编程工具, 适合企业在以 Web 为中心的应用程序环境中进行开发。这并不意味着 Windows 平台已经过时了, 而是说明你应该改变有关 Windows 编程的思考方式了。旧式的注册方式已经消失, 也就是说, 不再以 ActiveX/COM 服务器形式注册类型库 (type library)。 .NET 框架组件是自描述型的, 所有引用组件的信息都是元数据形式组件的一部分。

Visual Studio.NET 以 Microsoft.NET 框架为基础, 很显然 Visual Basic.NET 也是如此。VB.NET 是 Visual Studio.NET 程序包的一部分, 它是 Visual Basic.NET 的简易格式。Visual Basic 只是 Microsoft 用于 .NET 平台的语言之一。

最初, 一些新术语和概念看起来有点令人望而生畏, 但事实上它们并不那么可怕。如果

你想真正了解关于 .NET 编程概念和 .NET 框架的内容，那么只需阅读下去即可，因为本章将快速回顾这些基础知识。

1.1.1 快速浏览 .NET 框架组件

.NET 框架遵从用于交换数据的 CTS (Common Type System, 通用类型系统) 和用于语言互用性的 CLS (Common Language Specification, 通用语言规范)。简单地说，.NET 框架由三个主要部分组成：

- ASP.NET (Active Server Pages.NET)
- CLR (common Language Runtime, 通用语言运行时间)
- .NET 框架类库 (基类)

.NET 框架 由下列附件组成：

- 汇编
- 名称空间
- 管理组件
- CTS
- MSTL (MS Intermediate Language, MS 中间语言)
- 实时系统 (Just-In-Time, JIT)

下面几节将讨论这些组件。

ASP.NET

ASP.NET 是 ASP (Active Server Pages, 动态服务器页) 到管理空间 (managed space) 的演变，更确切地说就是管理代码执行 (详见本章后面“通用语言运行时间”和“管理数据”)。ASP.NET 是用于创建基于服务器的 Web 应用程序的框架。ASP.NET 页面通常会将外观呈现成现 (presentation) (HTML) 从逻辑或代码中分离开来。这意味着 HTML 会以 .aspx 扩展名保存在文本文件中，并将代码以 .vb 扩展名保存在文本文件中 (当你在 VB.NET 中运行逻辑/代码时)。

ASP.NET 在很大程度上都能与 ASP 语法兼容，这使得你能够将大多数现有的 ASP 代码移植到 .NET 框架，然后再逐一更新 ASP 文件。ASP.NET 有多种编程模型 (Programming model)，且能够以任意方式将这些模型混合使用。该编程模型如下所示：

- Web 表单 (Web form) 使你能够使用基于窗体的 UI 创建 Web 页。这些窗体与先前版本的 Visual Basic 窗体非常相像。这意味着你拥有适当的事件和出错处理能力。
- Web 服务 (Web service) 是实现远程服务器访问功能的一种很好的方法。Web 服务以 SOAP (Simple object Access Protocol, 简单对象存取协议) 为基础，而 SOAP 是一种基于 XML 的防火墙友好协议，它使用 HTTP 协议在 Internet 上进行数据传输。
- 标准 HTML (standard HTML) 使你能够遵照标准 HTML 3.2 或 4.0 版本创建网页。

CTS

.NET 环境以 CTS 为基础，这意味着所有的 .NET 语言都共享相同的数据类型。这使得在

不同的编程语言之间交换数据变得更为容易。如果你直接在源代码（例如，在不同的编程语言中创建的继承类）中交换数据，或使用 Web 服务或 COM+ 组件交换数据，都将不成问题。CTS 是在 CLR 中构建的。你是否遇到过与其他应用程序交换日期的问题？如果有，那么你一定懂得由 CTS 来设立适当的格式是多么有意义。比如，你不必担心这种格式 DD-MM-YYYY，只需传递一个 **DateTime** 数据类型即可。

CLS

CLS (common Language Specification, 通用语言规范) 是一组用于促进语言互用性的协定。这意味着为 .NET 平台开发的不同编程语言都必须遵守 CLS，以确保在不同的编程语言中开发的对象可以实现相互交流。这包括只揭示那些适用于 CLS 的特性和数据类型。只要输出方法、过程和数据类型适用于 CLS，你的内部对象、数据类型和特性就可以与 CLS 中的不同。换句话说，CLS 只不过是一种用于 .NET 框架中的语言互用性标准。

CLR

通用语言运行时间是 .NET 框架提供的运行时环境。它的职能是管理代码执行，即术语管理代码 (managed code)。Visual Basic.NET 和 C# 的编译器都生成管理代码，而 Visual C++ 编译器默认时生成非管理代码。这里所说的非管理代码是指用于 Windows 平台的代码，例如用于先前版本 Visual C++ 的编译器产生的代码。但是你可以用管理代码扩展 Visual C++。

对 JScript 的影响

这意味着 JScript 走到了尽头了吗？决非如此。如果你指的是在新编译过的 JScript 语言中的服务器端开发，那么这应该取决于开发者。众所周知，作为客户端代码执行，JScript 将继续存在的主要原因是它适用于所有的浏览器。但是，如果 CLR 被移植到足够的平台并得到足够的软件开发商支持，那么它很有可能在执行 Java 小程序方面与 Virtual Machine 竞争。Java 是一种已编译的语言，可以作为 .NET 语言使用或者与 Rational 软件中的 .NET 编译器一起提供。

VBScript、VBA 和 VSA

VBScript 和 VBA (Visual Basic for Application) 又怎样呢？这显然是与 VB.NET 入门相关的另一个问题。VSA (Visual Studio for Application) 是 VBA 的新版本，可用于定制和扩展以 Web 为基础的应用程序的功能。由于 VSA 是以 Visual Basic.NET 和 VBA 为基础的，所以它可以很好地进行 VB.NET 编程。VSA 实际上是 VBScript 和 VBA 的合并，它将脚本引擎运行时间用于 .NET 框架。即使在 .NET 和 Visual Basic 的早期版本的向后兼容性方面有所退步，该语言本身还是得到了显著的升级，这意味着需要改变某些现有的 VBA/VBScript 代码，以使它们能在 VSA 运行时环境中运行。

CLR 也能实现以下内部功能：

- 管理数据
- 自动完成垃圾的收集
- 共享一个源代码通用基础
- MSIL (Microsoft Intermediate Language, Microsoft 中间语言)

我们将在下面几小节中讨论这些特性。

管理数据

.NET 框架编译器主要生成管理代码，它由 CLR 进行管理。管理代码的意思是受管理的数据，这也意味着数据将始终由 CLR 来管理。管理代码有助于消除内存漏洞，但由于不再有确定的结果，所以你对数据的控制也会减弱，这正是 COM (+) 中有争议的一个优点。

自动收集垃圾

当对象由 CLR 管理（分配或释放）时，你对它们没有完全控制权。CLR 负责规划对象、引用对象，以及删除你不再使用的对象，而这个过程被称为自动收集垃圾（automatic garbage collection）。这与 Visual Basic 先前版本处理对象的过程（确定性结果）大不一样。因为先前版本的 VB 是以 COM 为基础，且使用 COM 模型来引用和计数对象，所以一旦应用程序实例化了一个对象（Set object = New Class），该对象引用的计数器就会加 1，当应用程序撤销了对象引用（Set object = Nothing 或当其超出了范围时），计数器就会减值。当计数器的值为 0 时，将自动从内存中释放对象。这是现在你必须注意的问题了，因为你不再拥有对象引用的完全控制权。如果你是 Java 程序员，就应该已经了解到了这一点。

源代码共享通用基础

因为所有适用于 CLR 的编译器都生成管理代码，所以源代码共享同一基础，该基础即为类型系统（CTS），而在某种程度上来说还有语言规范（CLS）。这意味着你可以继承使用由不同语言编写的类，这就是交叉语言继承（cross-language inheritance）的概念。这对于规模很大的开发组而言很有好处，因为这些小组中的开发者的技能可能会有很大差别。另一个主要优点是在需要进行调试时可以在同一环境中使用不同的编程语言安全地调试源代码！

中间语言编译

在编译代码时，代码会被编译成 MSIL，而 MSIL 会与代码中描述类型（类、接口和值类型）的元数据一起保存在 PE（Portable Executable）文件中。因为编译过的代码处于“中间状态”，所以不受平台的约束。也就是说，MSIL 代码可以在任何装有 CLR 的平台上执行。与 MSIL 一起保存在 PE 文件中的元数据允许代码对其自身进行描述，因而不需要类型库或接口定义语言（Interface Definition Language, IDL）文件。CLR 会在必要的时候从 PE 文件中查找元数据，并在文件执行时将其析取出来。

在运行时，CLR 的 JIT 编译器先把 MSIL 代码转换为机器代码，然后再执行机器代码。显然，机器代码适于安装了 CLR 的平台。JIT 编译器和 CLR 是由不同开发商提供的，我认为其中最知名的编译器就是由 Microsoft 制造的 Windows CLR。

1.1.2 JIT：虚拟机的另一种表达方式？

相信大家都听说过 Java 小程序所用的虚拟机（Virtual Machine）。简单来说，CLR 的 JIT

编译器在执行中间代码和不受平台限制方面与虚拟机相同。然而，CLR 的 JIT 编译器处理代码执行的方式比所谓的虚拟机更丰富。JIT 编译器是动态的，尽管它被用于特定的 OS，但在执行时还是会检测硬件并对硬件起作用。

JIT 编译器可以优化特定处理器的代码，该处理器是用在执行代码的系统之上的。也就是说，JIT 编译器只要一检测到 CPU，就会为特定的 CPU 优化代码。编译器不仅可以优化奔腾处理器的代码，还可以优化奔腾处理器某特定版本的代码，例如奔腾 IV。尽管目前管理代码的执行比非管理代码要慢一些，但它还是不错的。而且很快，管理代码的执行速度就会超过非管理代码的执行速度。

Java 虚拟机和 CLR 的另一个主要区别在于，前者可以被所有的新进程/应用程序调用，而后者却不能。

1.1.3 汇编和名称空间

由于 .NET 框架使用汇编和名称空间来对相关功能进行分组，因而你必须知道汇编和名称空间的真正含义。虽然不知道这些术语也可以开发一些非常简单的 .NET 应用程序，但那样就无法进一步深入学习。

汇编

汇编 (assembly) 是 .NET 框架应用程序的基石，也是运行时间的基本组成部分。所有使用 CLR 的应用程序都必须由一个或多个汇编组成。汇编为 CLR 提供了应用程序运行时所必需的信息。请注意，应用程序可以并经常由多个汇编组成，即汇编并不是应用程序部署的一个单元。你也可以根据 DLL 里的排序来考虑汇编。

虽然在本书中把汇编当作一个单独的实体，但事实上它可能由多个文件组成。将功能部署为单一的单元（即使超过一个文件）是对功能的一种逻辑收集。这样就必须通过某种途径使汇编成为一个整体。可以根据在某个文件中找到的类型库和信息考虑汇编。然而，汇编还包含了其他与组成应用程序相关的所有信息，因此它是自我描述型的。

由于汇编是通过汇编声明来自我描述的，所以我们不必再去想如何在应用程序中共享 DLL 以及如何处理自 Windows 产生以来一直存在的问题。然而，由于不再使用共享的 DLL，所以代码将占据更多的内存和磁盘空间，因为相同的功能现在可以很容易地在硬盘和内存中进行复制。然而，这也使得共享汇编并将其传播出去成为可能。

事实上，我们仍然可以在 .NET 框架内使用 COM+ Services (DLL 和 EXE)，甚至可以将 .NET 框架组件添加到 COM+ 应用程序中。但是本书只集中讨论 .NET 框架组件的使用。

正像前面所提及的那样，除了包含组成汇编的所有文件的索引外，汇编还包含了一个声明 (manifest)。该汇编声明也被称为汇编的元数据（与前面谈到的一样，元数据是用来描述主要数据的数据）。该声明内有表 1.1 中列举的所有组件。

表 1.1 汇编声明组件

项目	说明
标识	名称、版本、共享名和数字签名
文化、处理器和 OS	所支持的多种文化、处理器和 OS

续表

项目	说明
文件表格	作为汇编一部分的散列和其他所有文件的相对路径
参考汇编	所有外部独立性（即对其他汇编静态参考）的一个列表
类型参考	与如何为包含声明和实现方法的文件映射类型参考有关的信息
权限	为使运行更有效，汇编从运行时环境处请求的权限

除汇编声明组件之外，开发者也可以添加自定义汇编属性。这些纯信息属性可以包括标题和对汇编的描述。

名称空间

名称空间(namespace)是一条分组或组织相关类型(类、接口或值类型)的逻辑途径。.NET 名称空间可以在设计时提供方便，用于逻辑命名/分组。在运行时，它是建立名称作用域的汇编。

不同的.NET 框架类型使用带点语法的层次结构命名方案来命名。由于命名方案是分层的，所以应当有一个根名称空间。.NET 根名称空间被称作 **System**。例如，对于 **System** 名称空间中的第二级名称空间 **IO**，其全名为 **System.IO**。在编程中你将用到下列代码以使用 **System.IO** 名称空间内的类型：

```
Imports System.IO
```

这样就提供了一条访问名称空间内类型的方便途径。要声明一个对象以作为 **System IO** 名称空间中 **File** 类的一个实例，可以输入下列内容：

```
Dim objFile As File
```

当编译器遇到该声明语句后，就会在汇编内寻找 **File** 类。如果没有找到，则该编译器就在 **File** 类中添加 **System.IO.**，以查看是否能够在该名称空间内找到 **File** 类。当然，你也可以直接将名称空间加为类的前缀。所以，你也可以通过输入下列内容来在 **System.IO** 名称空间内声明一个对象作为 **File** 类的实例：

```
Dim objFile As System.IO.File
```

正如你所看到的，使用 **Imports** 语句可以大大减少键入的代码量。如果在类文件或模块文件顶部都使用 **Imports** 语句，则会使你的代码更易读且更易于维持。只需看一眼就知道用的是哪一个名称空间。

可以使用下列语法创建名称空间：

```
Namespace <NsName>
End Namespace
```

<NsName>是为名称空间所起的名称。所有要作为名称空间一部分的类型都必须位于名称空间块内。

此外，我们还需要利用名称空间来防止出现模棱两可的情况 [也被称作名称空间污染(namespace pollution)]。当两个不同的库包含相同的名称时就会发生这种情况，并且会导致

冲突。而名称空间可以帮助你解决这个问题。

如果两个名称空间的确是同一类型并且你要引用这个类型，编译器就将查找代码中引用的第一个名称空间，然后将其反馈给你。这就意味着如果你有一个模块文件或类文件，Imports 语句的位置将确定选择哪种类型。假设有两个名称空间（Test1 和 Test2），且这两个名称空间都实现 Show 类。而你需要引用 Test2 名称空间的 Show 类和这两个名称空间中的其他类，因此需要使用如下的 Imports 语句：

```
Imports Test1
Imports Test2
```

现在，如果要在作为 Import 语句的同一类文件或模块文件内创建对象作为 Show 类的实例，又会怎样呢？

```
Dim objTest As New Show
```

因为首先要引用 Test1 名称空间，而编译器会自动从其中挑选出 Show 类，所以，在这一特例中必须给类名加上其名称空间前缀，如下所示：

```
Dim objTest As New Test1.Show
```

名称空间可以覆盖多个汇编，这使得多个开发者可以在不同的汇编上进行工作，但是不能在同一个名称空间内创造类。可以随时访问当前汇编内的名称空间，也就是说，你不必导入它们。



名称空间是显式公用的，不能利用 Private 之类的访问修改程序对其进行改动。然而，用户对名称空间内类型的可访问性具有完全控制权。根据规定，名称空间的可见性为公用的。但是，放置名称空间的汇编决定了该名称空间的可访问性。这意味着名称空间对所有这样的工程是公用的，即这些工程引用了放置有该名称空间的汇编。

运行时环境（runtime environment）并不了解名称空间的任何信息，所以在访问某种类型时，CLR 需要该类型的全名以及包含该类型定义的汇编。有了这些信息，CLR 就可以载入汇编并访问该类型。

1.1.4 .NET 框架类库

.NET 框架的类库包含基类和基类的派生类。这些类包括许多功能，例如服务器控制、异常处理和数据访问进程等，这些正是本书的主题。使用 .NET 框架能够更快捷地构建出应用程序，这是因为你不必再从基础开始设计。类库实际上是一组统一并且面向对象的可扩展类库，也称作 API（Application Programming Interface，应用程序编程接口）。与先前版本的 Visual Basic 不同，那时许多系统和 OS 函数只能通过 Windows API 调用才可以访问。

1.2 获得合适的.NET 集成开发环境

IDE（Integrated Development Environment，在集成开发环境）中提出了许多新奇的特性，

下面将简要地介绍其中的一些特性：

- 为所有语言共享的 IDE
- 两种界面模式
- 内置 Web 浏览器功能
- 命令窗口
- 内置对象浏览器
- 集成调试程序
- 集成帮助系统
- 宏
- 升级部署工具
- 文本编辑器
- 服务器资源管理器
- 数据连接
- 工具箱
- 任务列表

1.2.1 所有语言共享 IDE

所有 .NET 语言共享同一个 IDE（请参阅图 1.1），这样所有开发者就可以在不同的编程语言中使用相同的工具，也意味着无论使用何种语言进行开发，都会获得相同的外观和感受。

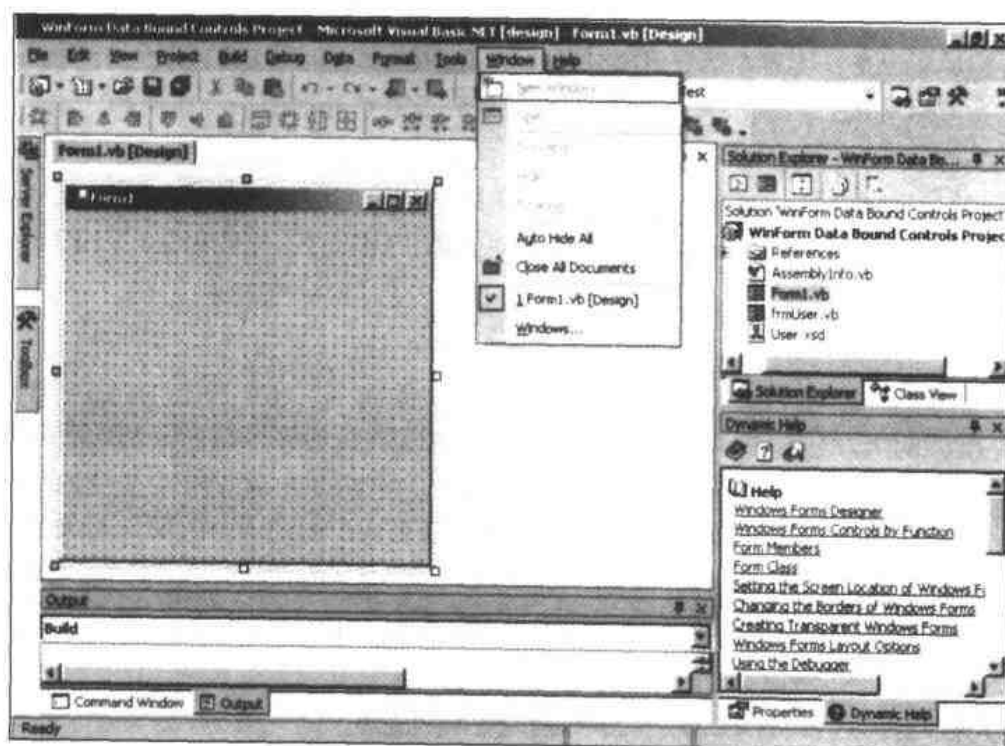


图 1.1 一个共享 IDE

1.2.2 两种界面模式

IDE 支持两种界面模式，用于调整窗口和窗格：

- MDI multiple document interface, 多文档界面模式：这是一种先前版本的 Visual Basic 所通常采用的旧模式，在这里可以用 MDI 父窗口来管理其上下文内的众多 MDI 子窗口。在图 1.2 中可以看到两个打开的窗口，(Start Page 和 Form1.vb [Design])，它们作为 MDI 子窗口重叠在一起。
- 文档标签模式 (tabs on documents mode)：这是默认模式，我个人更喜欢这种模式，因为它似乎更容易调整所有的窗口和窗格。单击适当的标签 Start Page 和 Form1.vb [Design]，便可打开图 1.1 所显示的窗口。

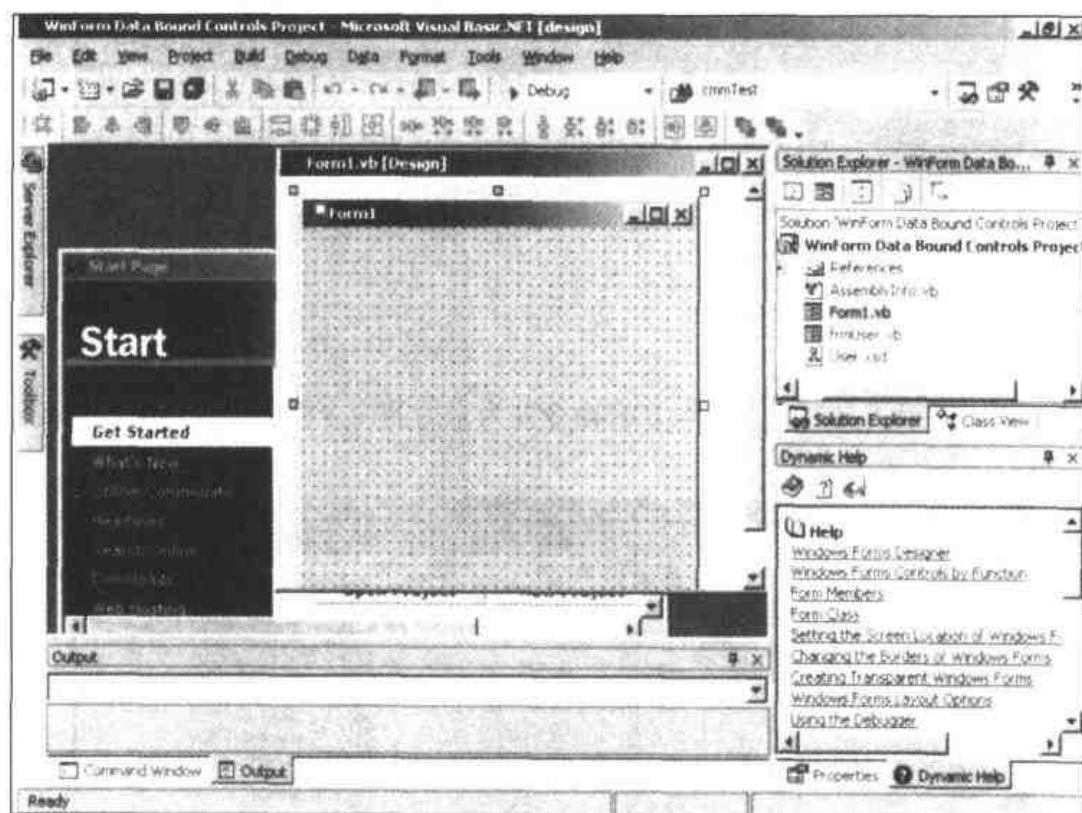


图 1.2 MDI 模式

1.2.3 内置 Web 浏览器功能

由于 IDE 具有内置 Web 浏览器性能，所以用户不必在其用于开发的机器上再安装其他浏览器。但是，我仍然建议至少要再安装一个以用于测试。内置 Web 浏览器的妙处在于当查看 Web 页时不必编译和运行整个工程。请参阅图 1.3 中的 Browse ~ Object Construction Sample 窗口，它在内置 Web 浏览器中显示了一个 HTML 页。

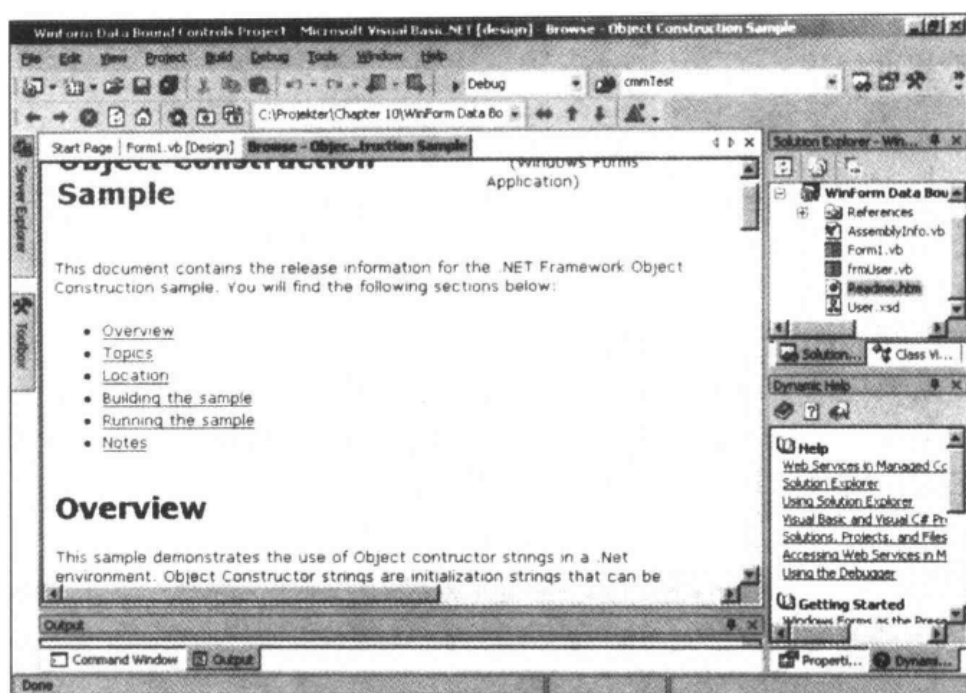


图 1.3 内置 Web 浏览器

1.2.4 命令窗口

命令窗口有两种模式：

- **command 模式**允许用户键入 IDE 命令并为常用的命令创建简短的别名（如图 1.4 所示）。

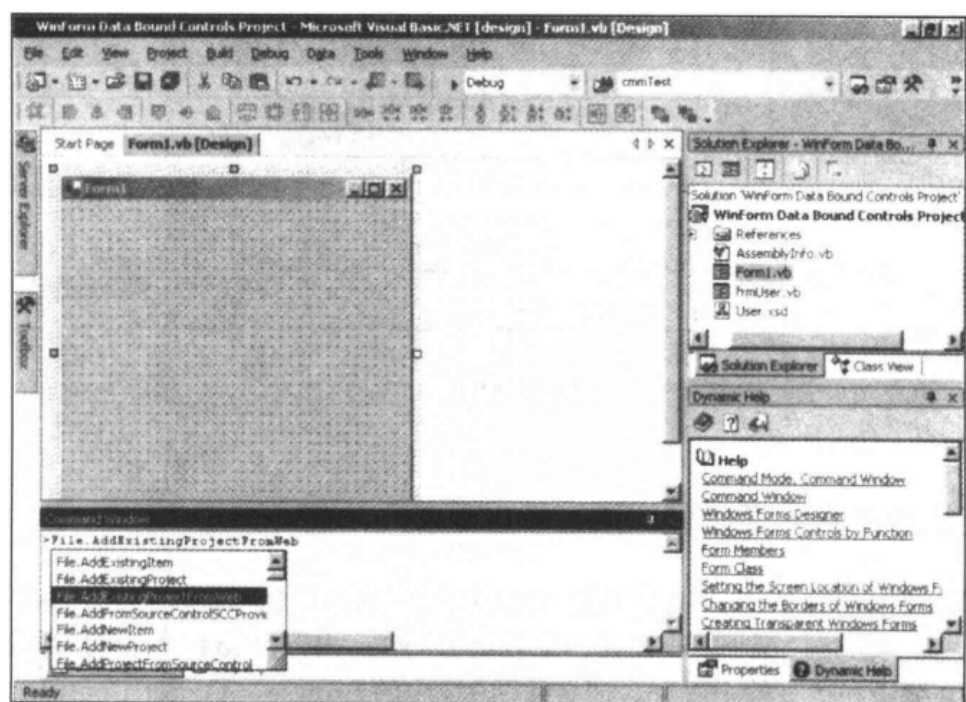


图 1.4 command 模式的命令窗口

- **immediate** 模式允许用户使用计算表达式（**evaluate expression**）来设置或读取变量的值，指定对象引用以及执行代码声明（如图 1.5 所示）。

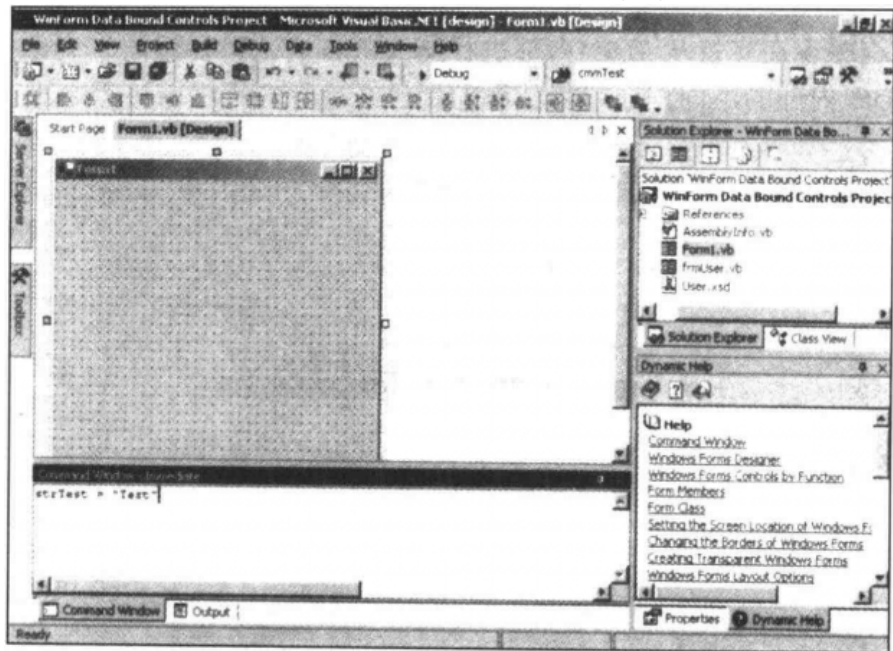


图 1.5 immediate 方式的命令窗口

1.2.5 内置对象浏览器

内置对象浏览器（**Built-in Object Browser**）感觉上与先前版本的 Visual Basic 中的类似，使你能够检测对象及其方法和属性。可以检测到的对象，也称作对象浏览器的浏览域，可以是部分外部组件或参考组件，或是当前解决方案中以工程形式出现的组件。浏览域中的组件包括 COM 组件和 .NET 框架组件（这点非常明显）。图 1.6 展示了对象浏览器。

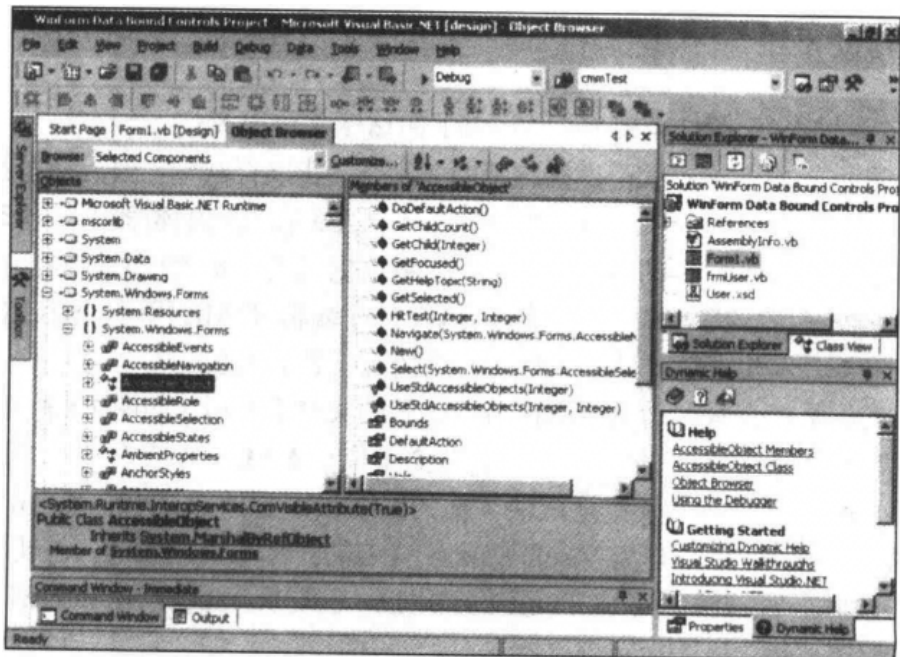


图 1.6 对象浏览器

1.2.6 集成调试程序

集成调试程序现在支持从 IDE 内部进行交叉语言调试。要想同时调试多个程序，可以通过运行 IDE 内部的不同程序或将其添加到已经运行的程序或进程之上（如图 1.7 所示）。

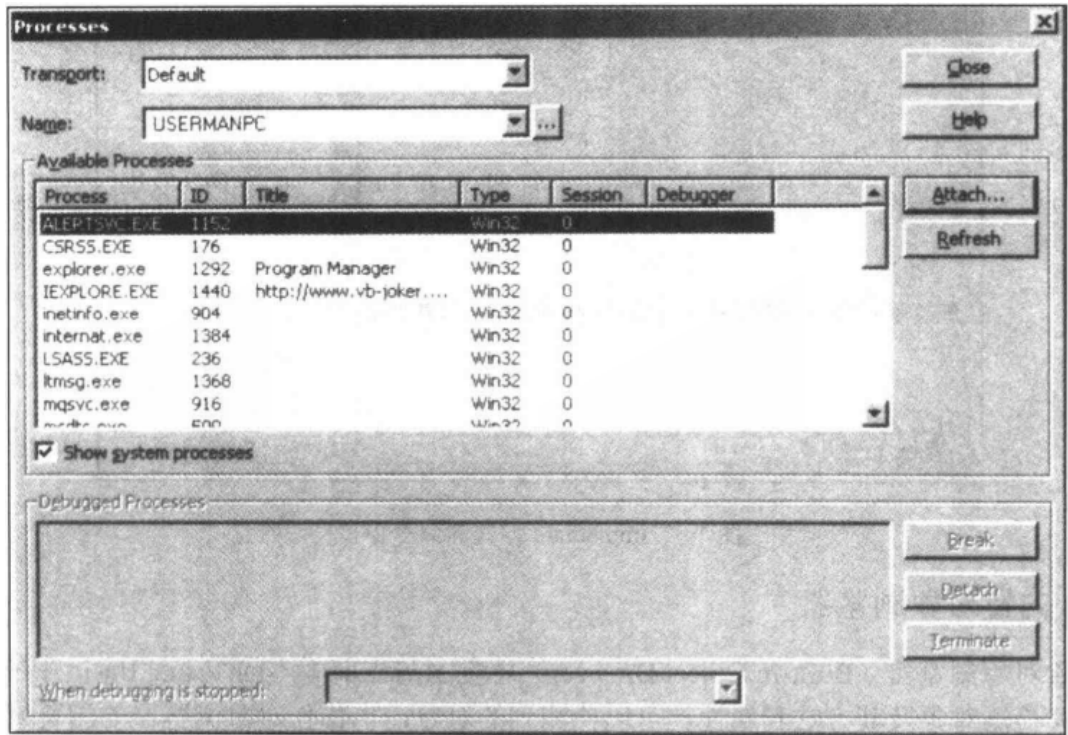


图 1.7 调试程序进程窗口

1.2.7 集成帮助系统

VS.NET 的帮助系统是比较先进的，它由下面这些部分组成：

- **动态帮助 (Dynamic Help)：**动态帮助非常好，我很喜欢它，因为动态帮助窗口的内容是基于当前上下文和选择而决定的，所以当你在代码编辑器中移动动态帮助窗口（其默认位置为 IDE 的右下角）时，它将根据当前选择改变其自身的内容。这种情况不仅仅在位于代码编辑器时会发生，而且在编辑 HTML 代码、使用 XML Designer、设计报告、编辑级联样式单 (CSS) 等情况下也会发生。

当你希望查看在动态帮助窗口中展示的某个主题时（如图 1.8 所示），只需单击该主题就会弹出相应的帮助文本，这些文本位于 MSDN 库应用程序中或者作为文档位于 IDE 内。究竟在那里还取决于你的帮助设置是外部显示还是内部显示。

- **MSDN：**MSDN 库是作为 Visual Studio.NET 的一部分而产生的，与辅助先前版本的 Visual Basic 时一样。VS.NET 完整的帮助系统以 Platform SDK 和 .NET 框架 SDK 归档的形式被包含进去。库浏览器现在已经有所改进，如今它是一种浏览器风格的应用程序，并链接到位于不同 Microsoft Web 站点上的相应信息。

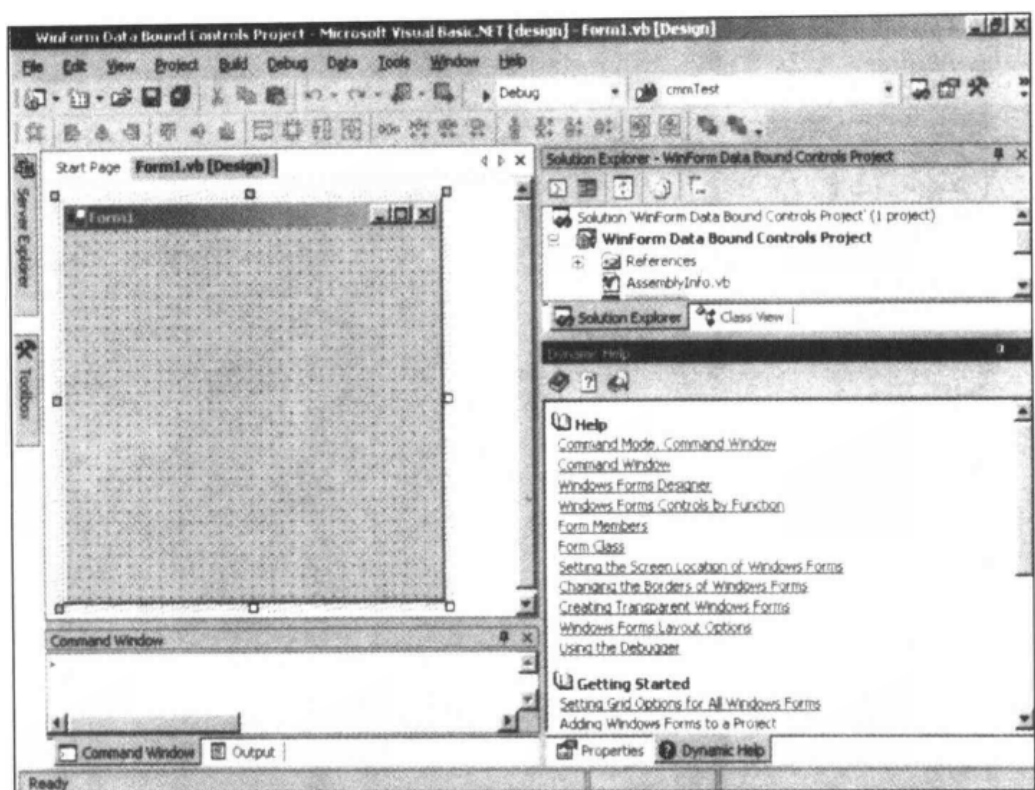


图 1.8 动态帮助窗口

1.2.8 宏

众所周知，开发过程中总要重复执行一些任务或一系列任务。那就是 VS.NET IDE 使用宏的原因，它可以自动完成这些重复的任务。可以用宏录制器 (recorder) 或宏 IDE 来创建/编辑宏。如果你曾经做过 VBA 编程，那么你一定觉得宏 IDE 似曾相识，因为它们非常相似。所以，它也是一种看起来和 VBA 很相似的语言。事实上，正如前面所讨论的那样，现在它被称作 Visual Studio for Applications，宏语言自然是基于 Visual Basic.NET 的。

1.2.9 升级部署工具

从先前版本的 VS 开始，VS.NET 的部署工具已经被大大地升级。在 VS.NET 中，你可以完成下面这些任务：

- 为用于远程调试的不同测试服务器部署应用程序等级
- 部署 Web 应用程序
- 利用 Microsoft Windows Installer 分配应用程序

1.2.10 文本编辑器

IDE 内的不同文本编辑器都具有相同的功能，这些功能是你期望 Windows 编辑器应具有。快捷方式也是一样的，并且大多数快捷键也都为大家所熟悉。

然而，还有一些内容是你从未见过的，包括在编辑器最左端显示的行号。虽然这些可选的行号不是代码或文本的一部分，但却可以给你带来方便（如图 1.9 所示）。

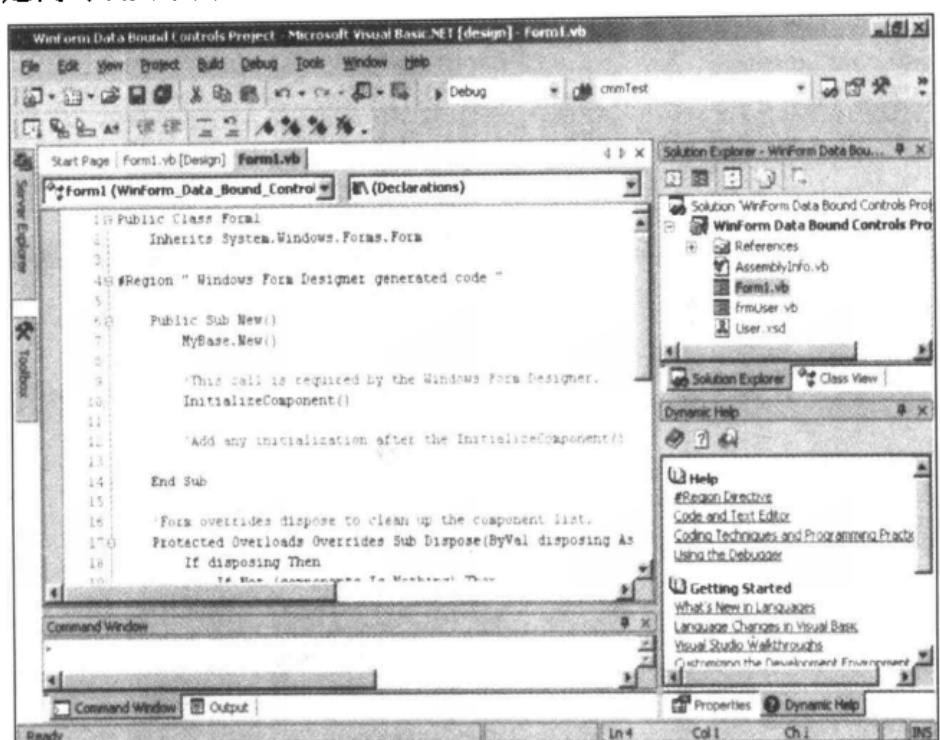


图 1.9 带有行号的文本编辑器

1.2.11 服务器资源管理器

服务器资源管理器窗口（默认状态下位于 IDE 左侧条上）用于操纵可以访问的所有服务器中的资源。这些资源包括：

- 数据库对象
- 性能计数器*
- 消息队列*
- 服务*
- Web 服务*
- 进程*
- 事件日志*

这里的大部分资源都是新的（标有*的资源都是新的），并可以直接把它们拖拽到 Web 表单或 Windows 表单中，然后在自己的代码内操作它们（如图 1.10 所示）。请注意，这些资源是否都可以从服务器资源管理器进行访问取决于所购买的 Visual Studio 版本。

1.2.12 数据连接

有多种与数据库的连接以及操作数据库对象的方法。我们会在第 3 章更详细地讨论这个主题，所以就不再这里在讨论了！

1.2.14 任务列表

任务列表窗口（默认状态下位于 IDE 底部）帮助用户组织和管理必须完成的各种任务以建立自己的解决方案。开发者和 CLR 都使用任务列表。这意味着开发者可以手动添加任务，但如果在编译时发生错误，CLR 也会添加任务。可以将某个任务以及文件名和行号一同添加到列表中，这样就可以通过双击该任务名称来访问问题或未完成的任务。

完成了某个特定的任务后，可以通过选择适当的复选框来表明任务已经完成（如图 1.12 所示）。

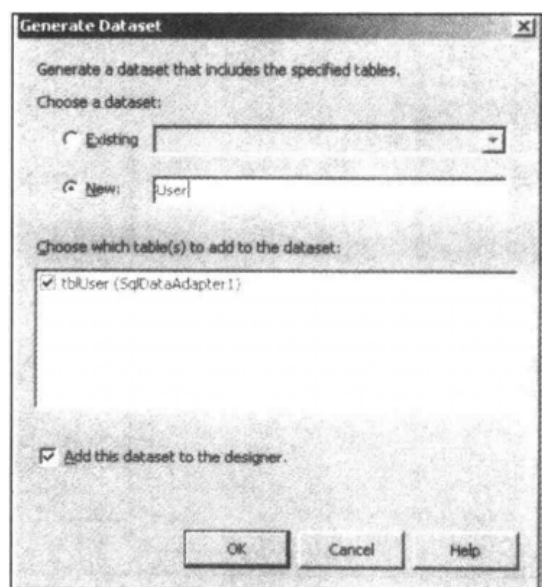


图 1.12 任务列表

1.3 小结

本章简要介绍了 Visual Basic.NET 中的新概念和术语，然后讨论汇编和名称空间，以及它们是如何满足 .NET 框架的要求的。通过本章的学习，我们了解到 CLR 用通用基础来统一 .NET 编程语言；通用语言规范是 .NET 语言必须遵守的一种标准；通用类型系统指定数据交换所使用的基类型；还介绍了基于服务器的 ASP（Active Server Pages，动态服务器页）编程以及框架类库。

最后，我们简要介绍了共享 IDE 的一些新特性，例如文本编辑器内的行号和集成调试程序。

下一章是对 Visual Basic.NET 的简要介绍，内容包括数据库的设计以及如何与数据库进行对话。通过对其他概念的介绍你将了解如何建立一个标准化的数据库。

第 2 部分

数据库编程

第2章

与数据库对话

常用数据库术语及概念简介

本章介绍了书中其他部分将用到的术语和概念，你可以把本章作为下面几章的参考。我们将了解究竟什么是数据库，为什么要使用数据库，以及如何设计关系数据库等方面的内容。尽管本章并非想教给你设计关系数据库所需的全部知识，但是你可以学到一些基本常识。另外，如果你想查找更多有关数据库设计以及数据库标准化链接的文章，可以访问 Microsoft 产品服务站点：<http://support.microsoft.com/support/kb/articles/Q234/2/08.ASP>。

本章末尾的实用练习有助于加深你对此处所讲数据库概念的理解。

2.1 数据库究竟是什么？

就像电子数据库可以由许多不同的软件实现一样，数据库这一术语有许多不同定义。基于这一原因，我们在这里只是给出一个简单的定义。数据库（database）就是数据集合，而这些数据可以是任何信息。电话号码是数据，名字是数据，一本书中也包含着数据，诸如此类。在定义数据的上下文中，数据是关于什么内容的并不重要。数据库通常是以某一方式进行组织的。

电话簿就是一个数据库，它通常按照姓的字母排序来组织。电话簿的主要目的是方便人们查找电话号码。当给出一个电话号码时，通常可以与一个由名和姓构成的姓名、一个地址或电话号码“拥有”者的头衔联系起来。这实际上是拆分电话簿的过程，或者应该称为了解电话簿是如何组织的过程。在随后的章节中我们还会提到这一点。

正如你所看到的，数据库不一定是数据的电子集合，也不一定必须以某种特别的方式来组织。但本书仅讲述电子数据库。

2.2 为什么使用数据库？

当谈及使用数据库的原因时，一些键便会跃入脑海：存储、可访问性、组织以及操作。

- 存储 (storage): 对于存在于你的大脑之外的数据库, 都需要以某种形式来进行存储。如果需要的话, 可以将它存在电子磁盘上并随身携带, 或是交给其他人。数据库可以是 Microsoft Excel 电子数据表或 Microsoft Access 数据库这类形式。
- 可访问性 (accessibility): 如果不能访问数据库中的数据, 那么你的数据库对任何人来说都是没有意义的。一般情况下, 在使用电子数据库时, 例如由 Microsoft Access 创建的数据库, 访问它的方式可以有很多种选择。可以用 MS Access 前端程序来访问 MS Access 数据库, 也可以用能够与驱动程序通讯的应用程序来进行访问, 而这种驱动程序必须能够与 MS Access 数据库进行对话。很明显可访问性取决于数据库存储的格式, 但是为了便于讨论, 我们假定这一格式是为人所知的标准格式, 例如 dBase 或 MS Access。
- 组织 (organization): 电话簿最合理的组织方式就是按字母排序。然而, 电话簿中有许多重复数据, 它们所占用的空间可能是没有任何重复的数据所占空间的两倍或更多。但数据的重复可以方便你通过多种方式来查询数据。你可以按姓来查询列表, 也可以通过地址来查, 等等。谈及电子数据库时, 重复显然不是问题, 只要它可以正确地运行即可。可以使用索引和键来为顾客提供不同的数据访问方法, 而且还不必重复数据。索引和键起着数据指针的作用, 它们是避免数据重复的有效工具。
- 操作 (manipulation): 编辑电话簿不是一项轻松的任务。当一个电话号码不再存在时可以删掉它, 但改变电话号码的拥有者或者增添一个新电话号码时又该怎样? 你不得不在几处编辑并/或增加数据? 这样不起作用, 除非你乐于重做整个电话簿! 而在电子数据库中, 如果操作正确的话, 这将是一件很简单的事: 定位要编辑或删除的信息, 输入数据, 然后离开。这时, 该数据就会被添加或替换到所有相应的位置。好, 让我们先来做个总结, 下面是使用数据库的好处:
 - 当数据在某一地方时, 对数据进行定位和操作要简单得多。
 - 当数据都在同一地方并以电子方式将它们存储起来时, 数据组织会更简单。
 - 当所有数据都放在同一地方 (数据库中) 时, 可以从不同的应用程序和位置来访问这些数据。

2.3 数据库管理系统

DBMS (Database Management System, 数据库管理系统) 这一术语可以用于许多场合, 而且比其他的术语更恰当。一个完整的 DBMS 由硬件、软件、数据和用户构成。然而在本书中, DBMS 只是一个软件包。通过它可以管理一个或多个数据库, 对数据库中的数据进行操作, 并以不同的数据格式导出或导入。DBMS 不一定是像 MS SQL Server 或 Oracle 那样的数据库服务器系统, 它也可以是像 Microsoft Access 或 Lotus Approach 那样的所谓桌面数据库。桌面数据库并不需要服务器, 它只是一个文件 (或相关文件的集合), 可以通过用户自己的应用程序中适当的驱动程序或其自带的前端软件来进行访问。

本书中将交替使用术语数据库和 DBMS。

2.4 行和记录

虽然这两个概念十分简单，但在我刚开始摆弄数据库时经常被它们弄晕。如果你也被这些术语所困扰，那么就让本书来帮助你吧！行（row）和记录（record）其实就是一回事。在同一场合中遇到这两个术语的时候，请记住这一点。

2.5 列和字段

像行和记录一样，术语列（column）和字段（field）也是同义的。

2.6 关系型和层次型

当前，有两类数据库得到了广泛使用，即关系型（relational）和层次型（hierarchical）数据库。关系数据库是其中迄今为止最为流行的，或许你对它已经有所了解。但我在这里仍要提醒你一下，你也许在毫不知情的情况下使用了分层数据库！Windows Registry 和 Windows 2000 中的 Active Directory 都是分层数据库。

2.6.1 分层数据库

分层数据库（hierarchical database）是反向的树状结构，其顶端是根节点（请参阅图 2.1）。根节点有一个或多个子节点，这些子节点又都有自己的子节点，正如一棵有枝有叶的树。你可能一直在使用 Windows 资源文件管理器，它正是依靠树状结构来表示磁盘和文件的。Windows 资源管理器更像一个拥有多个数据库的分层 DBMS，也就是说，“我的电脑”是一个数据库，而本地计算机上的 C 盘驱动器是“我的电脑”这一根节点的一个子节点，C 盘内的文件夹又是 C 盘的子节点，以此类推。

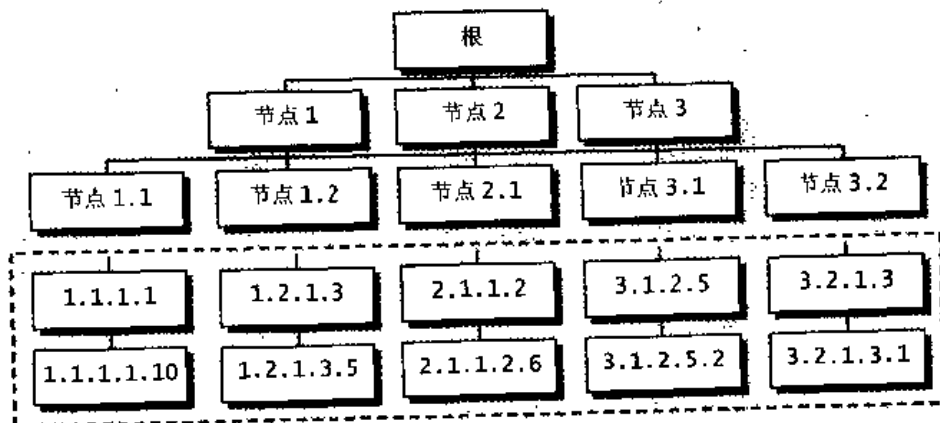


图 2.1 分层数据库

由于是直接链接的数据结构，所以对分层数据库的操作通常很快，但它们对于复杂的关系模型，例如在一个堆满原材料、产品、已装箱的产品、货架、成排的货架等物品的仓库系统里，就不够好了。这些项目都在一定程度上有所关联，要想避免数据重复和性能损失，就很难在分层数据库中表示出这些关系。

下面列出了一些分层数据库的使用场合：

- 文件目录系统
- 管理/组织等级

相信你自己还可以想出来一些。

分层数据库一般难以实现、建立和维护，通常需要系统程序员来完成这些工作。然而，IBM 提出了一种称为 IMS (Information Management System, 信息管理系统) 的分层数据库系统。你可以从 <http://www-4.ibm.com/software/data/ims/> 站点找到它。IMS 有一些与关系数据库系统（下面将讨论）中类似的工具，用于建立和实现分层数据库。虽然本书不会介绍分层数据库的实现方法，但将向你展示如何访问 Active Directory（也是分层数据库）。请参阅第 7 章以获得有关 Active Directory 和分层数据库的更多信息。

2.6.2 关系数据库

关系数据库 (relational database) 由以直接或间接方式相互关联的表构成。如果从对象角度考虑的话，表就代表对象，这些对象/表之间用所谓的关系联系起来。例如一个定货系统中，有产品、顾客及定单等等。在一个相当简单的定货系统中，用单独的表来分别存放与定单、顾客及产品相关的信息。一个顾客有零个或多个定单，而一个定单由一个或多个产品构成，现在你知道在表格间建立联系是多么容易了吧。请参阅本章后面的“关系数据库设计”部分，你会得到如何设计关系数据库的更多信息。

关系数据库模型

关系数据库模型 (Relational Database Model) 由 E. F. Codd “发明”，至今已成为数据库设计的实际标准。该模型的思想并不复杂，但是掌握它也有些难度。

关系数据库设计

在第一轮设计中就把数据库设计完全，这点很重要，因为在部署好之后，任何结构上的改变都会困难得多。把数据库设计好的另外一些原因是出于性能和可维护性方面的考虑。有时，在已经设计了一段时间以后，你也许会放弃进一步的设计，因为发现当前设计无法帮助你达成自己的追求目标。在设计自己的第一个数据库时，要仔细查看该设计，并依据发现的问题进行修改。你应该不断这样做，直到在该设计中再找不出任何问题。

虽然有许多工具可以生成关系数据库模型，但你仍需知道该如何构建关系数据库，以便充分利用这些工具。下面列出了一些可用工具的清单：

- ICT WizERD: <http://www.ict-computing.com/prod01.htm>
- Erwin: <http://www.cai.com/products/alm/erwin.htm>
- VisibleAnalyst@DBEngineer: <http://www.visible.com/dataapp/dappprods/vadbe.htm>

为数据库标识对象

在开始设计关系数据库时，首先要做的事就是坐下来找出数据库中应保留的对象。这里用**对象**一词来表示数据项，这些数据项代表顾客（**Customer 对象**）、定单（**Order 对象**）以及产品（**Product 对象**），而你在一般的定货系统中都会找到这些数据项。标识这些对象会有一定难度，尤其当你是关系数据库设计的新手时更为如此。如果的确这样，那么可以确定你标识的第一个对象，也可能是你惟一能够标识的对象，是 **Order 对象**。最重要的是要记住：虽然 **Order 对象** 是一个合理的对象，但这还不够。你还需要不断拆分对象，直到达到最底层为止。在定货系统示例中，一个定单由一个或多个 **OrderLine 对象** 和一个 **Customer 对象** 构成。**OrderLine 对象** 本身可以被分成更多对象，它是由产品类型和产品数量构成的。而 **Product 对象** 可派生 **Price 对象**。下面列出了该定货系统的一般对象：

- **Order**
- **Customer**
- **Product**
- **OrderLine**
- **Price**

然而，这里还有个问题。当把对象分成更小的对象时，操作条件应该是派生的对象是否有其自己的生命。让我们来看一下 **Price 对象**。它到底应该是一个对象，还是只是 **Product 对象** 的一个属性？这里所提到的属性指的是描述对象的信息。**Price 对象** 可能对应着不止一个 **Product 对象**，如果这样就会产生重复。虽然 **Price** 可以是一个单独的对象，但将它作为 **Product 对象** 的属性会更有意义。

请考虑下面的问题和答案：

- 一个 **Price** 对应着多个 **Product**，还是在 **Product 对象** 中 **Price** 会被重复？
Price 可能对应着不止一个 **Product**，如果这样就会产生重复。你可以把 **Price** 当作一个单独的对象，但如果将它作为 **Product 对象** 的属性可能会更有意义。尽管 **Price** 可能会重复，但必须考虑一下可维护性。当需要改变某个 **Product** 的 **Price** 时，改变 **Price 对象** 以及与其链接的多个 **Product 对象**，将会发生什么？所有关联 **Product** 的 **Price** 都会被改变。
- 一个 **Price** 只描述一个 **Product**，还是通配地描述所有 **Product**？
一个 **Price** 仅描述一个 **Product**。
- **Price** 是在发货时计算出的吗？
Price 是确定的，因此必须将其储存在某个地方。

在开始创建表时，有必要标识这些对象。迄今为止，所标识的对象都是实际的表。前面提到的属性，例如 **Price**，就是表中的字段。

关系

现在需要通过定义关系（relationship）来告诉 DBMS 各种表是如何关联在一起的。共有三种关系：

- 一对一
- 一对多
- 多对多

关系是建立在父表（parent table）和子表（child table）之间的。父表是主要对象，而子表是相关对象。

一对一关系

一对一关系用于在两个对象间创建直接关系，其中一个父表，另一个是子表。表 2.1 给出了此类关系的示例。

表 2.1 一对一关系

父表	子表
OrderLines	Products
Orders	Customers
Customer	Postal Code

一对多关系

一对多关系是最普遍的关系，作为一个数据库设计者，这些年你可能已经实现了许多这样的关系。这类关系存在于父表和子表之间，该父表可能有零个或多个子记录。表 2.2 给出了此类关系的示例。

表 2.2 一对多关系

父表	子表
Orders	OrderLines
Customers	Orders
Products	OrderLines

正如在前面一对一关系示例中所示，父表和子表经常颠倒。这意味着一对一关系中的父表在一对多关系中可能是子表。

多对多关系

当父表中的记录在子表中有许多相关记录，而子表中的记录在父表中也有许多相关记录时，就存在多对多关系。表 2.3 给出了这种关系的示例。

表 2.3 多对多关系

父表	关联表	子表
Orders	OrderLines	Products

父表和子表之间无法直接建立多对多关系的模型，而只能用第三个表来定义两个一对多关系，该表就被称作关联表。此表的某个记录中保存着父表记录的惟一 ID，也保存着子表记录的惟一 ID（请参阅图 2.2）。因此，要在关联表中寻找特定记录的话，需要知道父记录和子记录各自的惟一 ID。这两个 ID 是复合主键很好的候选。

图 2.2 应该读成：

- 一个定单包括一个或多个产品。
- 一个产品属于零个或多个定单。
- 一个定单行只隶属于一个定单，并且只包含一种产品。表 OrderLines 在这种情况下是一个关联表。

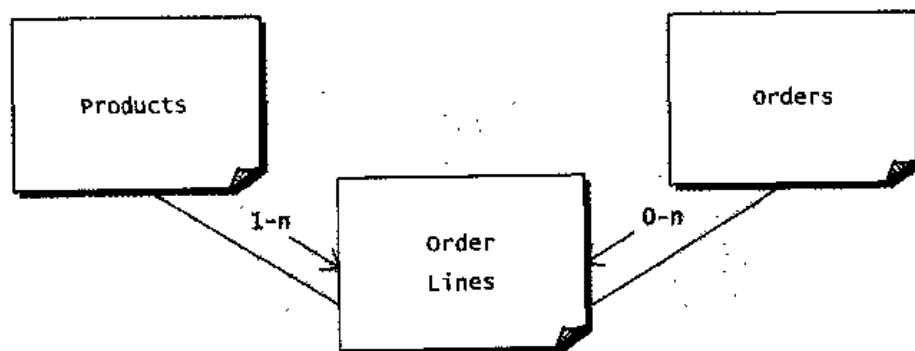


图 2.2 多对多关系

2.6.3 键

键（key）在关系数据库设计中非常重要，因为它们被用作记录的标识符，并用来排序查询输出。共有两类键：主键（primary key）和外键（foreign key）。

主键

一个表的主键总是惟一的，也就是说，表中任意两个记录在被指定为主键的字段中不可能有相同的值。这意味着主键惟一标识了表中的每个记录，而组成键或者是键一部分的字段不能为空（Null）。由不止一个字段构成的主键被称为复合主键（composite primary key）。主键常常是搜索键（lookup key），可用于查找记录，也可以在主键的字段中搜索值。根据关系的不同，主键常与子表中的外键（下面讨论）复合使用。在图 2.3 中，Orders 表中的主键标记为 PK。Orders 表中只能有一行与 OrderLines 表中的零行或多行相关。

外键

在关系中，外键用于表示子表中的搜索值。这意味着它直接对应于父表中主键的值，因此，可以用主键的值在子表中寻找相关记录。要注意，外键不同于主键，它可以包含空（Null）

值。虽然不推荐这样做，但这的确是可行的。在一对一关系中，外键必须是惟一的，否则关系中会包含重复。在图 2.3 中，OrderLines 表中的外键标记为 FK。OrderLines 表中可以有許多行与 Orders 表中的某行相关。请注意，在查找属于某售货员的 Orders 时，Orders 表中的 Salesman 字段是外键的候选。显然需要用到一个单独的 Salesman 表，该表将惟一的 SalesmanId 字段作为主键。

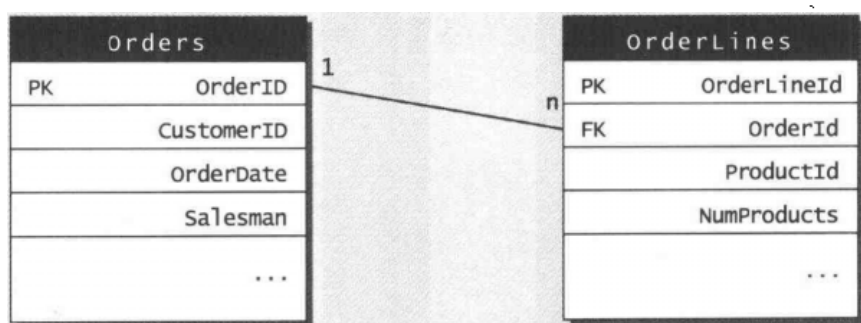


图 2.3 主键与外键

2.6.4 索引

你可以使用索引（index）来指定查找不是主键或外键的字段。当创建一个索引后，在构成该索引的字段中寻找某个值要快得多。虽然创建索引需要一些系统开销，但如果能仔细选择索引中所使用的字段，就会利大于弊。

这些系统开销会导致额外的磁盘空间消耗，而在添加或编辑记录时也会有一些性能损失。所以在试验索引的同时也需要做一些性能测试，看看使用索引会提高还是降低性能。在一个 Customer 表中，Name（姓名）字段就是一个索引候选。因为许多人都有相同的名字，所以它很容易重复，不能作为主键字段。但是顾客打来电话的时候，他或她可能会忘记自己的顾客 ID（可能是主键），这时就需要用他或她的名字来查找。

2.6.5 数据完整性

数据完整性（data integrity）表示的是数据库的内容是否“可信”，或者是否是最新的。换句话说，数据完整性意味着数据不会因为相关数据的更新而过时或者被孤立。可以通过校验下面的一两项来保证数据的完整性：

- 实体完整性
- 参考完整性
- 域完整性

实体完整性

实体完整性（entity integrity）规定主键中的值不能是 NULL。这一规定对复合主键（由

不止一个字段构成的键)和单字段主键都适用。特殊值 NULL 指的是一个空值或未分配的值。因为主键必须是惟一的,所以它们之中不能含有 NULL 值,因而在设计数据库时要应用实体完整性。

参考完整性

参考完整性 (referential integrity) 与关系有关。参考完整性保证了子表中记录的外键在父表中都作为主键存在。这意味着在强制参考完整性时,不能向子表中加入这样的记录:该记录拥有一个与父表中任意主键都不匹配的外键。

是否应用表 2.4 中的行为取决于在建立参考完整性时是否指定了限制或级联。级联 (cascade) 指的是这样一个过程:当在父表中更新或删除某个记录时,与之相关的表也同时得到更新。

表 2.4 参考完整性行为

任务	级联	限制
删除父表中的某个记录,该记录在子表中有相关记录	子表中的相关记录也被删除了	这种删除是不允许的
更新/改变父表中某个记录的主键,该记录在子表中有相关记录	子表中的相关记录也被更新了	这种更新/改变是不允许的

参考完整性保证了子表中的记录都不是孤立的,“孤立”是指子表中的记录在父表中没有相关记录。

域完整性

域完整性 (domain integrity) 保证了在适当位置某特定字段的值与域说明相符。域完整性是由应用于特定字段的物理和逻辑限制来保证的。当你试图向一个带域限制的字段中添加无效值时,这个字段不会接受该值。可以对 Customers 表中的 Phone Number 字段应用如下的域限制:

- 物理限制:数据类型——字符串;长度——13
- 逻辑限制:输入格式——(999) 9999999

2.6.6 标准化

实际上标准化并不是关系模型的一部分,但它的确是可以应用于关系数据库设计的一种过程。标准化 (normalization) 是设计数据库时用来进行分析的一种方法。标准化过程是一系列渐进的规则集,可称之为范式。标准化的目的是定义表,依据可预测结果来修改表,从而保证数据的完整性。可预测结果是指在修改数据时不会出现数据矛盾与/或冗余。范式定义于 20 世纪 70 年代,目的是解决表关系中的问题。这些问题也被称为异常 (anomaly)。

范式

由于范式 (normal form) 是渐进规则集,所以应该先用第一范式,然后再用第二范式,

依此类推（本节稍后将讨论不同层次的范式）。需要提醒你注意的是，在数据库设计中不一定要用到全部范式。许多经过标准化的数据库只符合前三种范式。作为一个数据库设计者，只要你觉得达到了目的，就可以终止标准化过程。

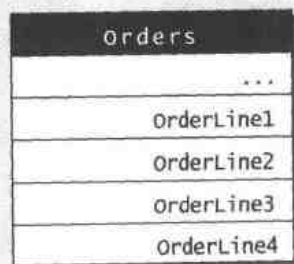
注意

当我在数年前初次遇到这些范式时也很难理解它们。而现在当我设计自己的数据库时，我一直在努力回想它们。如果只是设计出数据库，它们一般都符合前三种或前四种范式。事实上，如果你在一生中要设计许多数据库的话，那么我只能建议你熟记这些范式。请相信我，这是值得的。如果你已了解了标准化并将其应用到了数据库设计中，那么在本书剩下的章节中，你的工作就是找出错误！

第一范式

当把一组组数据放入不同的表中，并且用一对多关系连接起来时，就满足了第一范式（First Normal Form, 1NF）的要求。这方面的示例如图 2.4 所示，Orders 表是否有与这四个 OrderLine 对应的列？

别告诉我你从未犯过如图 2.4 中所示的错误。回想一下你起初设计的那些数据库实际上都是平面文件数据库，也就是说，数据库中的所有数据都放在同一个表或文件中。如果想符合第一范式，就应该按照图 2.5 那样设计表。



Orders	
...	
OrderLine1	
OrderLine2	
OrderLine3	
OrderLine4	

图 2.4 违反第一范式

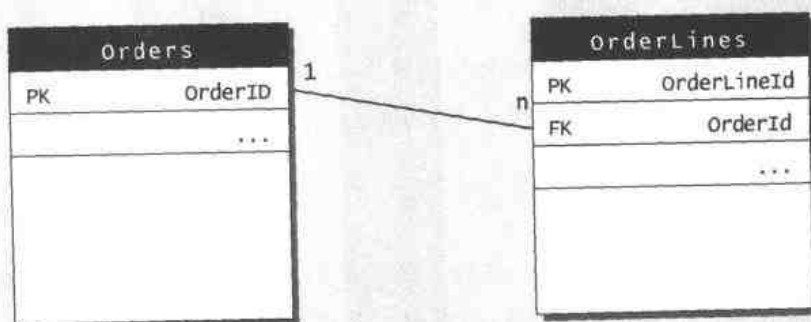


图 2.5 符合第一范式

第二范式

要符合第二范式（Second Normal Form, 2NF）的话，就必须先符合第一范式。实际上，

第二范式只应用于第一范式中那些具有复合主键的表。因此，如果数据库符合第一范式，并且在表中仅含有单字段主键，那么数据库就自动符合第二范式。然而，如果数据库中的确含有一个复合主键，那就需要消除对局部键的函数依赖。把违反范式的字段单独放进一个表中可以实现这一点。

例如，如果有一个 OrderLines 表，该表中包含图 2.6 所示的字段，那么 ProductDescription（产品描述）字段就违反了第二范式。因为 ProductDescription 字段只与 ProductId 字段有关而与 OrderId 字段无关，所以违反了第二范式。这意味着要符合第二范式，就必须把 ProductDescription 字段移至 Products 表中。你可以用 ProductId 作为查找值来获得产品描述。

OrderLines	
PK	OrderId
PK	ProductId
	...
	ProductDescription

图 2.6 违反第二范式

第三范式

可以通过符合第二范式和消除对非键字段的函数依赖来符合对第三范式（Third Normal Form, 3NF）。如在第二范式中那样，把违反范式的字段放入单独的表中就可以满足第三范式的条件。这意味着全部非键字段仅依赖于整个键。

在图 2.7 中，Location 字段不依赖于 ProductId 这一键段，却依赖于 Warehouse 这一非键字段。这违反了第三范式，应该创建一个新的 Warehouse 表以符合第三范式，该表中应该有 Warehouse 字段和 Location 字段。可以使用 Warehouse 字段作为查找值来得到 Warehouse 的位置。

第四范式

要符合第四范式（Fourth Normal Form, 4NF），表中所有记录就必须都符合第三范式，并且不能有多值依赖。可以这样解释术语多值依赖（multivalued dependency）：记录中可能没有两个或更多有关某实体的独立多值事实。有点困惑了？好，考虑一下以下内容：

- 一个顾客可以说一种或多种语言，一个或多个顾客可以说一种语言。
- 一个顾客可以有一种或多种付款方式，一种付款方式可适用于零个或多个顾客。

正如你所看到的，要处理两种多对多关系。

在图 2.8 所示的示例中，你可以为每个顾客存储一种付款方式和一种语言。惟一的问题是，语言和付款方式是多值的，也就是说，有不只一种语言和付款方式。这事实上意味着，某个说着英语和荷兰语、具有货到付款（Cash On Delivery，简称 COD）和信用卡两种支付方式的顾客，将需要不止一条顾客记录。实际上，如果想保存所有的组合（2 种语言×2 种付款方式），那么这个顾客需要四条记录。你能想像出一旦要进行改动，比方说换一种付款方式，就要更新全部记录是多么可怕吗？

Products	
PK	ProductId
	...
	warehouse
	Location

图 2.7 违反第三范式

Customers	
PK	CustomerId
	...
	Language
	Payment

图 2.8 违反第四范式

为了避免违反第四范式，需要用关联表来表示这两种多对多关系，正如图 2.9 中对 Customers-Payments（顾客-付款）关系所做的那样。请注意 CustomerPayments 表中的主键是复合的，也就是说，由两个字段构成，这两个字段都是单字段外键，分别对应 Customers 表和 Payments 表的关系。

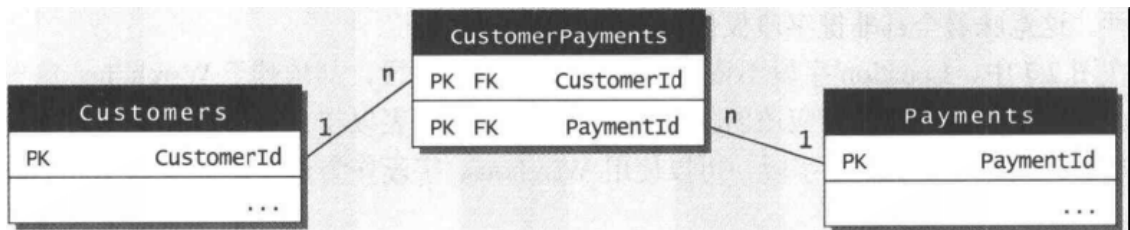


图 2.9 符合第四范式

需要符合第四范式来确保没有数据冗余并使数据更新更为容易。

第五范式

当某个记录中的信息无法从几个更小的记录中进行重构时，该记录就符合第五范式(Fifth Normal Form, 5NF)。这里更小的记录指的是比原记录包含的字段少的记录。第五范式与第四范式没有多大差别，但我还没在数据库中应用过第五范式。

2.6.7 反向标准化

反向标准化(denormalization)是标准化的反向过程，通常是因为标准化后有性能损失才这样做。然而，什么时候需要开始反向标准化处理并没有严格的规定。在开始进行标准化之前，请对自己当前的系统做一下基准测试以确立一个基准线。进行标准化之后，再进行

次基准测试，看看有没有性能提升或损失。假如性能损失太大，就要进行反向标准化处理，从而将在标准化过程中所做的变动都改回去。每次最好只做一个这种改变，以便测出每次改变可能产生的性能提升。

即使数据库设计是经过深思熟虑的，反向标准化也是一个重要方面，因为当你开始向数据库中添加数据时，性能最可能降低。

2.7 UserMan 数据库架构

在本书中建立的 UserMan 例程具有数据库特征。图 2.10 展示了该 SQL Server 数据库的架构。

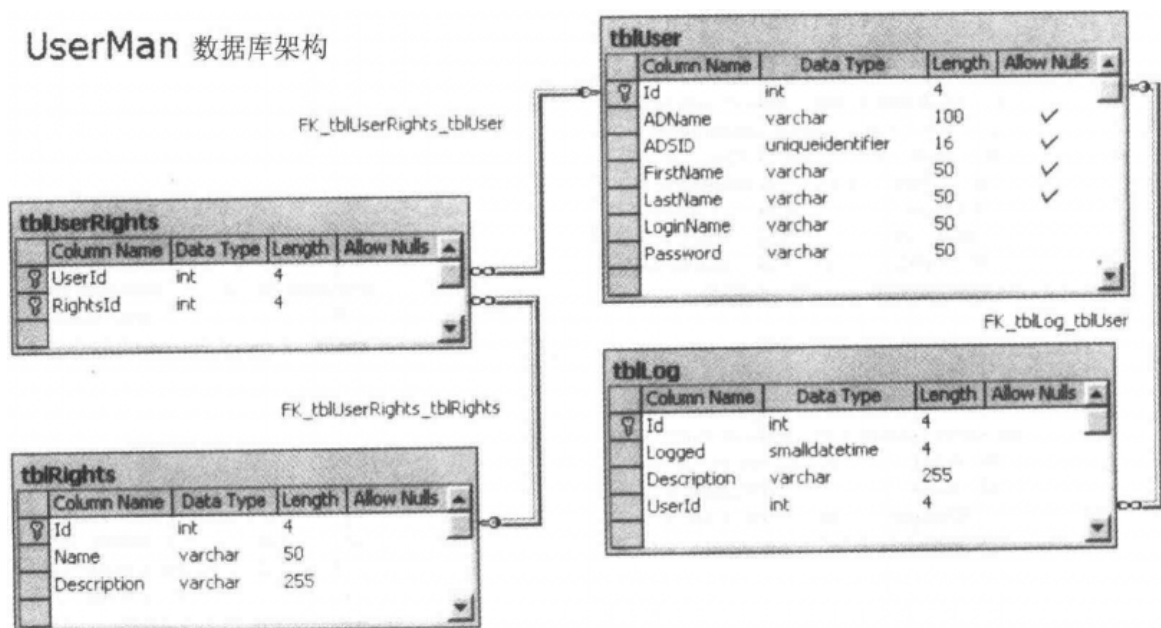


图 2.10 UserMan 数据库架构

如图 2.10 所示，该数据库包含四个表：

- **tblUser**：这是主要的表，其中存放着用户全部的标准信息，例如姓和名等等。
- **tblUserRights**：这是一个关联表，存放着 tblUser 表与 tblRights 表之间的多对多关系。
- **tblRights**：该表存放着系统分配给用户的不同权限，这些权限允许用户创建新用户，删除已有用户，等等。
- **tblLog**：该日志表中存放着用户在何时进行了何种操作的信息。我们可以通过它把用户紧紧控制起来！

练 习

检查 UserMan 数据库架构，看看依照本章所介绍的各种设计工具，可否找出某种办法来改进此设计。

如前所述，本书将继续分析该数据库，最后一章会给出有关如何扩展现有架构和例程的一些意见。

2.8 小结

本章解释了数据库的含义，以及怎样使用与何时使用数据库。简要指出了关系数据库与分层数据库间的区别。接着讨论了关系数据库的设计，并说明了建立关系数据库所涉及到的大多数概念，例如范式、键（主键与外键）、索引、表之间的各种关系类型以及如何从逻辑上实现这些关系。此外我们还通过对实体完整性、参考完整性和域完整性的讨论介绍了数据完整性。

本章最后一节详述了 UserMan 例程的数据库架构，本书各章都将以该数据库为例。

下一章将为你介绍 ADO.NET 中的大多数类，并解释它们的使用方法。

第 3A 章

ADO.NET 介绍：连接层

本章与其姊妹章节（第 3B 章）一起讲述了 ADO.NET。ADO.NET 中的类可以分成两组：连接层和非连接层。本章讲述连接层。

ADO（Active Data Objects）作为 Visual Basic 程序访问不同数据源的接口已有段时间了。我也可以把 ADO 看作应用软件的中间层，位于前端程序和数据源之间。ADO 2.7 是该产品的最新版本。

ADO.NET（Active Data Objects.NET）是 ADO 的一个全新的“能歌善舞”的版本。ADO.NET 被设计成基于 XML 的非连接式与分布式数据存取技术。这很好地符合了 Microsoft 的新“隐藏”战略，该战略使 ADO 可以在所有支持 XML 的平台上运行。这也意味着可以在 Internet 上将 ADO.NET 数据集从一个地方移值到另一个地方，因为 ADO.NET 可以穿透防火墙。实际上，完全可以说 ADO.NET 是一次革命，而不只是一次进步。

ADO 与 ADO.NET

本书中涉及到这两种数据存取技术。下面就是区分它们的方法：

- ADO 用来描述我们已经用了多年的 ADO 技术。该技术基于 COM，并且只能通过 COM Interop 使用。
- ADO.NET 是完全基于 .NET 框架的新版本，其内在技术与基于 COM 的 ADO 有很大不同。尽管名称中仍有 ADO，但两者是不同的技术。

然而，如你所料，我们还在使用 ADO，如果要在 .NET 框架中使用 ADO，则要用到 COM Interop（互用性）。在第 3B 章中你将看到 COM Interop。

由于 ADO.NET 是非连接式结构，所以多数 .NET 应用程序都使用它。然而，应该牢记一点，ADO.NET 针对的是 n 层的 Web 应用程序或分布于众多不同服务器上的应用程序。因此，如果仅是为 Windows 平台建立 n 层或客户机—服务器解决方案，最好还是使用 ADO。ADO.NET 的后续版本可能会改变这一点。

本章介绍了 ADO.NET 和 ADO，但 ADO 方面的内容不像 ADO.NET 那样详细。如需获得关于 ADO 的更多信息，请参阅 Apress 出版的图书：

- 《Serious ADO: Universal Data Access with Visual Basic》，Rob Macdonald 著。ISBN

1-893115-19-4。2000 年 4 月出版。

- 《ADO Examples and Best Practices》，William R. Vaughn 著。ISBN 1-893115-16-X。2000 年 5 月出版。

ADO 和 ADO.NET 中所有对象模型看起来都一样。若想访问一个数据源，则需要有一个提供程序。提供程序（provider）实际上是驱动程序的另一种说法，它是一个能够以二进制形式访问数据源的库（library）（有关细节请参阅本章稍后的“提供程序”部分）。有了提供程序后，需要从使用提供程序的应用程序中建立一个连接，以便访问数据源。参见图 3A.1 中的简化图片。既然已建立和打开了连接，就可以执行查询、检索，以及从数据源中删除、操作和更新数据。

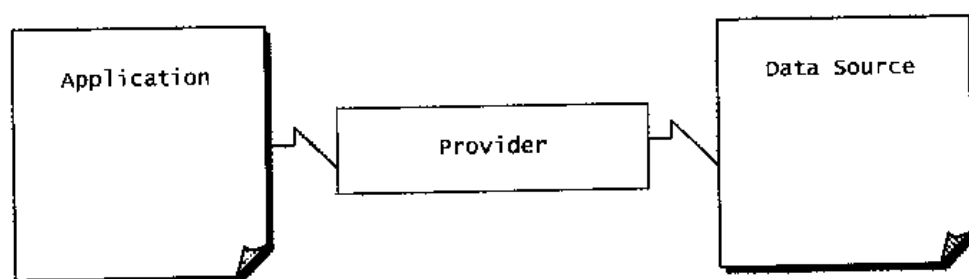


图 3A.1 提供程序的工作原理

要操作数据就需要有数据对象，在后面的部分中将会解释这点。

3A.1 数据关联的名称空间

VB.NET 中（或者更确切地说，就是在 VS.NET 中）有许多名称空间（namespace），它们在处理数据时是非常重要的（请参阅表 3A.1）。这些名称空间保存着多种数据类，为了使用这些类，必须把它们引入代码。实际上，可以在声明和实例化对象时，在这些类前面加上名称空间前缀。但是，首先仍需知道可以找到这些类的位置。



注意

如果不确定究竟什么是名称空间，请参阅第 1 章。

表 3A.1 数据关联的名称空间

名称空间	说明
System.Data	保存 ADO.NET 类和其他各种一般类，或者 .NET 数据提供程序中的子类
System.Data.SqlClient	保存 MS SQL Server 7.0 或更高版本中特有的类。是 MS SQL Server 7.0 或更高版本的 .NET 数据提供程序
System.Data.OleDb	描述用于访问 OLE DB 数据源的类集。也就是 OLE DB .NET Data Provider

可以如下使用 **Imports** 声明将名称空间引入代码中的某个类中：

```
Imports System.Data
```

也可以按照如下所示在声明与/或实例化对象时在类前面加上前缀，而不必引入名称空间：

```
Dim cnnUserMan As System.Data.SqlClient.SqlConnection ' 声明新连接
cnnUserMan = New System.Data.SqlClient.SqlConnection() ' 实例化连接
```

3A.2 提供程序

提供程序 (provider) 只是驱动程序 (driver) 的另一种说法，这意味着它是一个揭示 API (应用程序接口) 的二元库，可被应用程序调用。该库是一个 DLL 文件，但有时还依赖于其他 DLL。因此，一个提供程序可由多个文件构成。此处所讨论的提供程序被称为 OLE DB 提供程序，可以直接访问。然而，在使用 VB.NET 或先前版本的 VB 时，提供程序库所揭示的 API 已被预先封装在了 ADO 类中。因此，不需要为这些低级编程而着急，你所要做的只是在建立或者打开一个连接时指定提供程序的名称。

OLE DB 其实是 OLE DB 提供程序所使用的规范，而 OLE DB 提供程序的主要目的是将 OLE DB 指令翻译成目标数据源的语言，以及将目标数据源的语言翻译成 OLE DB 指令。

ADO 与 ADO.NET 之间的重要差别是：ADO 通过 COM Interop 调用 OLE DB 提供程序，而 ADO.NET 使用 **DataAdapter** 类通过 COM Interop 调用 OLE DB 提供程序 (参阅图 3A.2)。事实上，这样说并不完全正确。有时候 **DataAdapter** 对象会直接访问 DBMS 所提供的 API，正如 MS SQL Server 7.0 或其更新版本中的情况一样 (请参阅图 3A.3)。.NET 中的 SQL Server 数据提供程序使用称为表列数据流 (tabular data stream, TDS) 的专用协议与 SQL Server 对话。

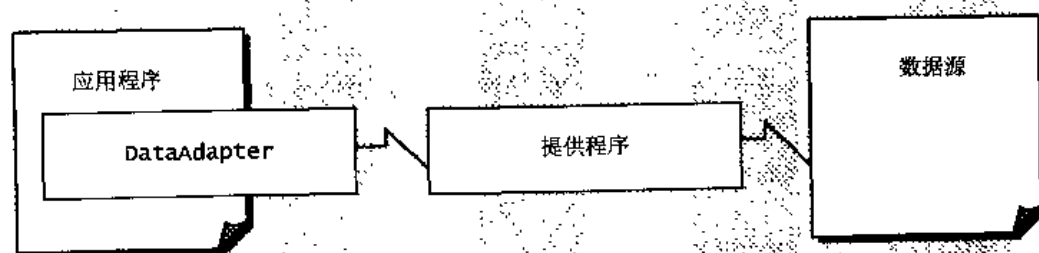


图 3A.2 DataAdapter 使用 OLE DB 提供程序访问数据源

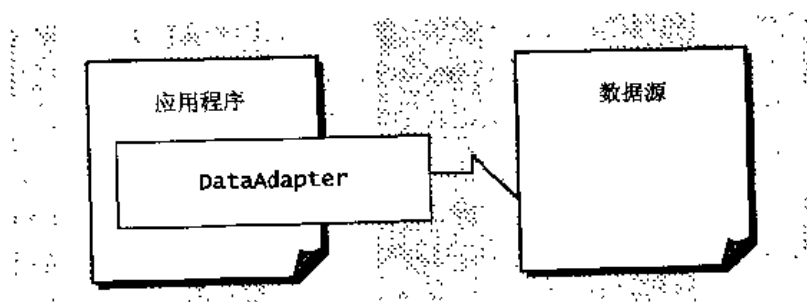


图 3A.3 DataAdapter 直接访问数据源

如果在编译时得知需要连接的某个数据源是具有特定 .NET 数据提供程序的数据仓库,那么就使用其提供程序,因为它已针对特定数据库做了优化。另一方面,如果是在创建一个通用数据包,不知道程序运行时将连接何种数据源,那么与之相配的是通用的 OLE DB.NET 数据提供程序。这样,现在已经我论述了特定的 .NET 数据提供程序,例如 SQL Server.NET 数据提供程序,以及用于任意与 OLE DB 兼容的提供程序的通用 OLE DB .NET 数据提供程序,但是还有第三种,也就是一定程度上被忽视了的 .NET 数据提供程序,由 ADO.NET 提供。这就是 ODBC.NET 数据提供程序。该提供程序专门用于访问 ODBC 数据源,并且只有在有合适的 ODBC 驱动程序时才能使用。本书没有对 ODBC 进行详细论述。

表 3A.2 列出的提供程序可以与 OLE DB.NET 数据提供程序一起操作。有关细节请参阅本章稍后的“.NET 数据提供程序”部分。

表 3A.2 与 OLE DB.NET 数据提供程序兼容的提供程序

提供程序名称	说明
SQLOLEDB	用于 MS SQL Server
MSDAORA	用于 Oracle
Microsoft.Jet.OLEDB.4.0	用于 JET 数据库 (MS Access)

3A.2.1 在连接时指定提供程序

虽然打开或者建立一个连接的方法有很多种,但下列方法可以指定要使用的提供程序:

```
cnnUserMan.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=C:\Northwind.mdb"
```

在连接 `cnnUserMan` (很明显,它已在别处声明和实例化了)的 **ConnectionString** 属性中指定想要使用 JET 提供程序 (`Provider=Microsoft.Jet.OLEDB.4.0`)。请参阅本章稍后的“ConnectionString 属性”一节。

3A.2.2 .NET 数据提供程序

.NET 数据提供程序是新型 ADO.NET 编程模型的一个核心部分。尽管一个 .NET 数据提供程序松散地对应着一个 OLE DB 提供程序,但它实际上与 ADO 对象模型更接近。这是因为 OLE DB 和 ADO 层已经融合在 .NET Framework 中,给予 .NET 数据提供程序更高的性能。之所以可以更高的性能源于:在使用 ADO.NET 时,实际上只用了一层。而在用 ADO 和 OLE DB 提供程序时,却使用了两层。

看待这方面的另一种方式是,只有先前知道自己究竟在做什么时,才会直接用低层 OLE DB API 来笨拙地修补程序。这样做是为了获得更高的性能。然而现在你不必再这么做了,朋友! ADO.NET 具有高层编程接口,由 ADO.NET 对象模型进行揭示,与直接使用 OLE DB 具有相同的性能。这不是很重要吗?但在你为此而乐昏了头之前,我必须指出决定性的一点:

OLE DB 规范使数据源获得了最大限度的可访问性。也就是说，它揭示了所有不同类型数据源的操作性，然而 .NET 数据提供者揭示了最小限度的操作性。因此 OLE DB 不会很快消失，并且只有在非连接的分布式访问数据源背景下，才会被 .NET 数据提供程序代替。对于其他大部分场合，当需要丰富的客户端数据操作时，可能仍需使用 ADO 与 OLE DB。

一个 .NET 数据提供程序实际上由以下 4 类数据构成：

- Connection（连接）
- DataAdapter（数据适配器）
- Command（命令）
- DataReader（数据阅读器）

下面几段详细论述了这些类。 .NET 数据提供程序是两层中的第一层，这两层组成了 ADO.NET 对象模型。第一层也被称为连接层（connected layer），另一层则被称为非连接层（disconnected layer）。非连接层是 **DataSet** 对象。有关这方面的详细信息，请参阅第 3B 章中“使用 DataSet 类”部分。

3A.2.3 Connection 类

Connection 类是 .NET 数据提供程序的一部分。在访问某个数据源时，需要有一个连接。但如果只是想在某个表中保存一些关联数据的话，就不需要了。在第 3B 章的“构建自己的 DataTable”一节中我们将会谈到这个话题。而本节并没有讨论 ADO 连接，仅讨论了 ADO.NET 连接。如果读者想了解 ADO Connection 对象的具体信息，可以读一下本章开始提到的几本书之一。

Microsoft 随 .NET Framework 一起提供的两个可控连接分别是：**OleDbConnection** 和 **SqlConnection**。可控（Managed）的意思是由 CLR 执行，并在 CLR 环境下运行。



有关细节请参阅第 1 章中“常用语言运行时”部分。

尽管 **OleDbConnection** 类可用于 SQL Server 数据源，但对 SQL Server 7.0 或以后的版本应使用 **SqlConnection**，因为它对连接这一特定数据源做了优化。对其他数据源则使用 **OleDbConnection**，除非你可以为自己的特定数据库找到第三方 .NET 数据提供程序（并因此找到特定的 connection 类）。

OleDbConnection

使用可控 **OleDbConnection** 来连接所有已启用 .NET 的应用程序。在需要连接不同数据源时，**OleDbConnection** 对创建要使用的通用数据封装尤其有用。**OleDbConnection** 用来通过 OLE DB 提供程序建立连接。

SqlConnection

SqlConnection 类仅用于 MS SQL Server 7.0 或其后续版本。它针对该特定数据库的连接

进行了特殊优化。除非要建立通用数据包，否则最好在已启用 .NET 的应用程序中使用此可控连接。

3A.2.4 ConnectionString 属性

在 **SqlConnection** 中，**ConnectionString** 属性与 OLE DB 连接字符串完全类似，而在 **OleDbConnection** 中，与 OLE DB 连接字符串完全兼容。该属性用于指定连接到数据源的字符串。只有在关闭了连接后，才可以设置和读取该属性。该属性的默认值为空。全部值都可以用单引号或双引号括住以进行指定，但这是可选的。各个值对之间 (value=) 必须用分号 (;) 隔开。



在任何时候，**ConnectionString** 中使用的分隔符都一样。即使你是使用不同字符集和分隔符 (等等) 的国际用户，也需要在该属性中用分号作为分隔符！

下面是一个 **ConnectionString** 属性的示例：

```
"Data Source=USERMANPC;User ID=UserMan;Password=userman;Initial
Catalog=UserMan"
```

大多数值名不区分大小写，但某些值，例如 **Password** 的值名则是区分大小写的。表 3A.3 以字母顺序列出了 **ConnectionString** 属性的值。

表 3A.3 ConnectionString 属性值

值名	默认值	请求	说明	示 例	特定提供程序
Addr 或 Address		是	要连接 SQL Server 的名称或网络地址 (例如，一个 IP 地址)	Addr='DBSERVER' 或 Address='10.0.0.1'	是 (作用于 SqlConnection 类，并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Application Name		不是	应用程序的名称	Application Name='UserMan'	是 (作用于 SqlConnection 类，并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
AttachDBFilename		不是	要连接的主要数据库文件名称。可连接数据库的名称包括了完整路径。该值一般用于基于文件的数据库，例如 Microsoft Access	AttachDBFilename='C:\Program Files\UserMan\Data\UserMan.mdb'	

续表

值名	默认值	请求	说 明	示 例	特定提供程序
Database		是	数据库的名称	Database= 'NorthWind'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDb Connection 类)
Data Source		是	基本上与 Addr 或 Address 相同, 但它也作用于 MS Access JET 数据库	Data Source='C:\Program Files\UserMan\Data\Use rMan.mdb' 或 Data Source='USERMANPC'	不是
Connect Timeout 或 Connection Timeout	15	不是	连接数据库时在终止连接尝试前等待的秒数。如果发生超时, 则表明出现了异常	Connection Timeout=15 或 Connect Timeout=15	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDb Connection 类)
Connection Lifetime	0	不是	可以用该值来指定终止连接的时间。该值仅与连接池 (Connection pooling) 有关, 并且该值 (秒) 指定了一个可接受的时间范围, 与创建连接和连接返回连接池的时间相比较。换句话说, 这意味着一个连接与该值相比较所生存的秒数 (返回连接池的时间减去创建时间)。如果 Connection Lifetime 值小于某个连接已生存的秒数, 该连接就会被终止。该值在负载平衡的情况中非常有用, 因为如果将 Connection Lifetime 设成一个小值, 那么连接时常在返回连接池时就被终止了。这意味着在重新建立该连接时, 它可能被创建在不同的服务器上, 这取决于服务器群的负载状况	Connection Lifetime=10	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)

续表

值名	默认值	请求	说 明	示 例	特定提供程序
Connection Reset	'true'	不是	该值决定了一个连接在被终止时是被重置还是只从连接池里删除。如果该值设为'false', 可以避免在需要该连接时再向服务器申请。应该清楚, 连接没有被重置, 因此该连接会保留它在归还给连接池之前所设置的任何值	Connection Reset='false'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Current Language		不是	这是 SQL Server 语言的记录名称。检查 SQL Server 文档以获得它所支持语言的清单	Current Language='English'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Enlist	'true'	不是	设置为'true'时, 在线程的当前事务处理中获得连接	Enlist='false'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
File Name		不是	用于一个 Microsoft Data Link File (数据连接文件, UDL)	File Name='Test.udl'	是 (仅作用于 OleDbConnection)
Initial Catalog (同 Database)					
Initial File Name (同 AttachDBFilename)					
Integrated Security	'false'	不是	指明一个连接是否需要加密。可能值'true'、'yes'和 'sspi'用于指定一个加密连接, 而默认值'false'和'no', 用于指定一个不加密连接。	Integrated Security='true'	是 (用于 SqlConnection 类)
Max Pool Size	100	不是	指定连接池中连接个数的最大值	Max Pool Size=200	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)

续表

值名	默认值	请求	说 明	示 例	特定提供程序
Min Pool Size	0	不是	指定连接池中连接个数的最小值	Min Pool Size=10	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Net Network Library	或 'dbmssoen'	不是	用于连接 SQL Server 的网络库。其值可以为 'dbmssoen' (TCP/IP)、'dbnmpntw' (Named Pipes)、'dbmsrpcn' (Multiprotocol)、'dbmsvinn' (Banyan Vines)、'dbmsadsn' (Apple Talk)、'dbmsgnet' (VIA)、'dbmsipcn' (Shared Memory) 或 'dbmsspxn' (IPX/SPX)。请注意, 在欲连接的系统和开始连接的系统上都必须安装有相应的库文件 (DLL)	Net='dbnmpntw'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Network Address (同 Addr 或 Address)					
Password 或 Pwd		是, 如果对数据库进行了保护	对应 User ID 的密码	Password='userma' 或 Pwd='userman'	不是
Persist Security Info	'false'	不是	帮助你保证诸如密码等敏感信息的安全。该值的行为依赖于连接的状态。其值为 'true' 或 'yes' 时, 不保证敏感信息的安全。当该值为 'false' 或 'no' 时则保证敏感信息的安全	Persist Security Info='true'	不是

续表

值名	默认值	请求	说 明	示 例	特定提供程序
Pooling	'true'	不是	如果该值设为'true',则连接对象都取自适当的池。如果池中没有任何可用连接对象,就创建一个新连接对象并加入池中	Pooling='false'	是 (作用于 SqlConnection 类, 并且 SQLOLEDB 提供程序作用于 OleDbConnection 类)
Provider		是	指明欲用来访问数据源的 OLE DB 提供程序名称	Provider=SQLOLEDB	是 (仅作用于 OleDbConnection 类)
Server (同 Addr 或 Address)					
Trusted_Connection (同 Integrated Security)					
User ID		是, 如果你的数据库进行了保护	希望用于连接的 User ID	User ID='UserMan'	不是
Workstation ID	客户机名或者开始连接的计算机名	不是	客户机名或者开始连接的计算机名	Workstation ID='USERMANPC'	不是

上表中的许多值都具有对应的属性, 例如 **ConnectionTimeout** 属性就可以单独进行设置 (有关细节, 请参阅“连接类属性”部分)。再提醒一次, 只有在连接被关闭后才可以设置这些属性! 因此, 如果已打开了一个连接, 就需要在设置某些属性前关闭该连接。在设置 **ConnectionString** 属性时, 同时对相应的属性进行设置。

在设置连接字符串时, 该字符串会被立即解析, 这意味着马上就能捕捉到任何语法错误及抛出的异常 (发生一个可俘获的错误)。在这里只捕捉到语法错误, 其他错误仅在试图打开连接时才能发现。

一旦打开了连接, 验证过的连接字符串将被作为连接对象的一部分返回, 连接对象的属性也随之更新。提醒你一下, 如果不设置 **Persist Security Info** 值或者把该值设为 'false', 在 **ConnectionString** 中就不会返回敏感信息。具有默认值的值也是如此, 它们不会在连接字符串中返回。如果想在设置 **ConnectionString** 属性后查看它的值, 知道这点就很重要了。

如果在 **ConnectionString** 属性中对一个值设置了不少一次, 那么只有最后出现的 **valuenam=value** 对起作用。同样, 如果使用了一个同义字, 像 **Pwd** 而不是 **Password**, 那么将返回“正式的”值名 (本例中返回 **Password**)。

最后一点, 值之间以及值名之间的空格将被去掉。但这并不十分正确, 因为如果使用单引号或双引号来定义值, 那么空格就将起作用!

3A.2.5 连接类属性

除了 **ConnectionString** 属性, 我们还可以通过值的对应属性对不同值单独进行设置。表 3A.4 和表 3A.5 以字母顺序列出了连接属性。请注意, 属性若具有 **Protected** 可访问性, 则仅在继承连接类的类中才可访问。这与 VB.NET 中全新的 OOP (Object Oriented Programming, 面向对象编程) 工具十分吻合。

表 3A.4 SqlConnection 类属性

属性名称	可访问性	ConnectionString 等价值	说 明	读	写
ConnectionTimeout	Public (公共的)	Connection Timeout		是	否
Container	Public	没有等价的 ConnectionString 值	该属性继承自 Component 类, 到达包含组件的 IContainer 接口。如果 Component 没有被密封在 IContainer 中, 则返回 Nothing	是	否
Database	Public	Database 或 Initial Catalog		是	否
DataSource	Public	Addr 或 Address		是	否
DesignMode (继承自 Component)	Protected	没有等价的 ConnectionString 值	该属性用来检查一个组件是否处于设计模式。如果该组件处于设计模式, 则返回 True , 否则返回 False 。请注意, 如果一个组件没有关联的 Isite , 则总是返回 False	是	否
Events (继承自 Component)	Protected	没有等价的 ConnectionString 值	该属性返回一个 EventHandlerList 对象, 对象中包含附在组件上的全部事件处理器	是	否
PacketSize	Public	没有等价的 ConnectionString 值	该属性指定了数据包的大小, 这些数据包用来通过网络与数据源通信。返回值是以字节数来计算的	是	否

续表

属性名称	可访问性	ConnectionString 等价值	说 明	读	写
ServerVersion	Public	没有等价的 ConnectionString 值	该属性接收来自 SQL Server 的字符串, 其中包含当前 SQL Server 的版本信息。如果 State 属性是 Closed, 那么在读取该属性时就会发生 Invalid OperationException (无效操作) 异常	是	否
Site (继承自 Component)	Public	没有等价的 ConnectionString 值	该属性取得或设置组件的位置。返回一个 ISite 对象, 但如果组件没有相关的 ISite 或该组件没有密封在一个 IContainer 中, 则返回值 Nothing。如果组件从 container (Icontainer) 中删掉的话, 也会返回 Nothing	是	否
State	Public	没有等价的 ConnectionString 值	该属性读取连接的当前状态, 其值可以为 Open 或 Closed。这些值取自 ConnectionState 的枚举值, 是 System.Data 名称空间的一部分	是	否
WorkstationId	Public	Workstation ID	该属性返回一个定义数据库客户端的字符串	是	否

表 3A.5 OleDbConnection 类属性

属性名	可访问性	ConnectionString 等价值	说 明	读	写
ConnectionTimeout	Public	Connection Timeout		是	否
Container	Public	没有等价的 ConnectionString 值	该属性继承自 Component 类, 获取包含组件的 Icontainer 接口。如果组件没有密封在 IContainer 中, 则返回 Nothing	是	否
Database	Public	Database 或 Initial Catalog		是	否
DataSource	Public	Addr 或 Address		是	否

续表

属性名称	可访问性	ConnectionString 等价值	说明	读	写
DesignMode (继承自 Component)	Protected	没有等价的 ConnectionString 值	该属性可以用来检查组件是否处于设计模式。如果处于设计模式, 则返回 True , 否则返回 False 。请注意, 如果一个组件没有关联的 ISite , 则总是返回 False	是	否
Events (继承自 Component)	Protected	没有等价的 ConnectionString 值	该属性返回一个 EventHandlerList 对象, 对象中包含附在组件上的全部事件处理器	是	否
Provider	Public	Provider		是	否
ServerVersion	Public	没有等价的 ConnectionString 值	该属性接受来自 DBMS 的字符串, 该字符串中包含着版本号。如果 State 属性为 Closed , 则当试图读取该属性时, 会抛出 InvalidOperationException 异常。如果 OLE DB 提供程序不支持该属性, 则返回一个空 String	是	否
Site (继承自 Component)	Public	没有等价的 ConnectionString 值	该属性读取或设置一个组件的 site 。返回一个 ISite 对象, 但如果 component 没有相关的 ISite 或该 component 没有密封在一个 IContainer 中, 则返回值 Nothing 。如果 component 被从 container (Icontainer) 中删掉的话, 也会返回 Nothing	是	否
State Public		没有等价的 ConnectionString 值	该属性读取连接的当前状态, 其值可以为 Open 或 Closed 。这些值取 ConnectionState 的枚举值, 是 System.Data 名称空间的一部分	是	否

解释 Component、Container、IContainer、ISite 以及 Site

尽管详细讨论 Component、Container、IContainer、ISite 和 Site 超出了本书的范围，但理解这些术语却是十分重要的。

- **Component (组件)**: 提供应用程序间共享的对象，或者可以操作其他应用程序提供的对象的组件，并把对象包含的组件显示给其他应用程序。任意给定类都必须实现 **IContainer** 接口以成为一个有效组件。
- **Container (容器)**: 一个包含零个或多个组件的对象。类必须实现 **IContainer** 接口以成为一个容器。
- **IContainer (容器接口)**: 为容器提供操作性的接口。该操作性包括添加、删除和检索组件。
- **ISite**: 由 **IContainer** 接口指定的接口程序，该接口为 site 提供操作性。
- **Site (站点)**: 把一个组件与一个容器结合在一起，并为容器提供了管理组件所需的操作性，以及一个容器和站点之间需要的通讯程序。如果一个类实现了 **ISite** 接口，那么它只能是一个站点。

3A.2.6 连接类方法

表 3A.6 与表 3A.7 列出了 **Connection** 类的方法。这些表仅是一些参考程序清单，因此，如果需要有关如何使用这些方法的信息，例如怎样打开一个连接，请阅读表后的解释。如果需要查看所有标为“Overloaded”的不同超载函数，同样请阅读表后的解释。有关细节，请参阅本章稍后“连接类异常处理”部分。

表 3A.6 SqlConnection 类方法

方法名	可访问性	说 明	返回值	可重载的
BeginTransaction()	Public	开始有关连接对象的事务处理。该方法是可重载的，详细信息请参阅本章稍后“事务处理”部分	实例化了的 SqlTransaction 对象	是
ChangeDatabase (ByVal vstrDatabase As String)	Public	该方法不能被忽略，它可以将所连接数据库转换为变量 vstrDatabase 指出的数据库。连接必须是已打开的，否则会出现 InvalidOperationException 异常	无	不是
Close()	Public	该方法没有参数，仅仅是关闭一个打开的连接。当调用了该方法以后，任何未进行的事务处理都将被返回	无	不是
CreateCommand()	Public	该方法创建一个 SqlCommand 。 SqlCommand 对象与连接对象相关	实例化了的 SqlCommand 对象	不是
Dispose() (继承自 Component)	Public	该方法可忽略，负责清除命令对象，并释放对象所使用的任意资源	无	是

续表

方法名	可访问性	说 明	返回值	可重载的
Equals()	Public	该方法继承自 Object 类, 可被用来判断两个连接是否相同	Boolean (布尔) 型数据	是
Finalize()	Protected	该方法继承自 Object 类, 被用来在 Garbage Collector (无用单元收集程序) 切断连接之前, 尝试释放资源并执行清除操作。然而, 因为无法保证该方法一定有效, 所以应该用 Dispose 方法观察。如果有必要的话, 该方法在派生类中必须被忽略, 因为默认情况下, 它什么也不做。这样的原因是, 如果 Garbage Collector 不得不处理运行 Finalize 操作的话, 它所需运行时间很可能要长的多	无	不是
GetHashCode() ()	Public	该方法继承自 Object 类, 返回连接的散列码 (hash-code)	Integer (整型) 类数据	不是
GetLifetimeService() ()	Public	该方法继承自 MarshalByRefObject 类, 返回一个终身服务 (lifetime service) 对象。该对象控制着当前连接类实例的终身政策。默认的终身服务返回一个 ILease 类的对象。该方法应仅用于与远程激活对象的连接。 Leasing Distributed Garbage Collector (LDGC) 负责将租用 (lease) (终身服务) 与远程激活的对象关联起来。一旦租用期满, 对象就被清除了	Object 类数据	不是
GetService (ByVal vtypService As Type)	Protected	该方法继承自 Component 类, 返回一个表示服务 (vtypService) 的对象, 该对象由组件提供	Object 类数据	不是
GetType()	Public	该方法继承自 Object 类, 返回对象的类型, 或更确切点说, 就是返回元数据, 并与对象所继承的类相关联。如果需要该类的名称, 可以用 cnmUserMan.GetType.ToString 读取	Type 类数据	不是
InitializeLifetimeService() ()	Public	该方法继承自 MarshalByRefObject 类, 用来向连接对象提供自己的租用, 从而控制连接的存在时间	Object 类数据	不是
MemberwiseClone() ()	Protected	该方法继承自 Object 类, 负责创建连接的副本。克隆出的连接是原连接的一个浅拷贝。“浅”指的是原连接被克隆, 并且原连接对于其他对象的引用也被克隆。这意味着获得了一个原连接的新拷贝, 与初始连接有同样的对象引用。这是与深拷贝相比较而言的, 在深拷贝中, 原连接被复制, 原连接所引用的对象也都被复制	Object 类数据	不是

续表

方法名	可访问性	说 明	返回值	可重载的
Open()	Public	该方法不能被忽略, 用于在当前属性设置下打开与数据源的连接。在不同情形下将抛出不同的异常(请参阅本章稍后“连接类异常处理”)		不是
ReferenceEquals (ByVal vobjEqual1 As Object, ByVal vobjEqual2 As Object)	Public	该方法用于比较索引, 因此它可以用于检查两个连接实例是否指向同一个连接	Boolean (布尔) 类数据	不是
ToString()	Public	该方法继承自 Object 类, 返回一个表示连接的字符串。如果用 MsgBox(cnnUserMan.ToString()) 来调用该方法, 消息框所显示的信息应包括文本 System.Data.SqlClient.SqlConnection 。该文本是对象 (SqlConnection) 的名称, 以及包含该对象的名称空间 (System.Data.SqlConnection) 的名称	String 类数据	不是

表 3A.7 OleDbConnection 类方法

方法名	可访问性	说 明	返回值	可重载的
BeginTransaction()	Public	开始对一个连接对象的事务处理。该方法是可重载的, 有关详细信息, 请参阅本章稍后“事务处理”部分	实例化了的 OleDbTransaction 对象	是
ChangeDatabase (ByVal vstrDatabase As String)	Public	该方法不可被忽略, 它负责将所连接的数据库变为变量 vstrDatabase 所指出的数据库。连接必须是已经打开的, 否则就会抛出 InvalidOperationException 异常。		
Close()	Public	该方法没有任何参数, 它仅仅用于关闭已打开的连接。调用该方法后, 任何未进行的事务处理都将被返回	没有	不是
CreateCommand()	Public	该方法创建一个 OleDbCommand 。 OleDbCommand 对象是与连接对象相关联的	已实例化的 OleDbCommand 对象	不是
Dispose() (继承自 Component)	Public	该方法可被忽略, 它负责清除命令对象并释放它所使用的资源	不是	

续表

方法名	可访问性	说 明	返回值	可重载的
Equals()	Public	该方法继承自 Object 类, 可用来判断两个连接是否相同。	Boolean (布尔) 型数据	是
Finalize()	Protected	该方法继承自 Object 类, 用来在 Garbage Collector (无用单元收集程序) 切断连接之前, 尝试释放资源并执行消除操作。然而, 因为无法保证该方法一定有效, 所以应该用 Dispose 方法观察一下。如果有必要的话, 在派生类中将其忽略。在默认情况下, 该方法什么也不做, 因而, 如果 Garbage Collector 不得不处理运行 Finalize 操作的话, 它所需的运行时间很可能要比原来长得多	没有	不是
GetHashCode()	Public	该方法继承自 Object 类, 返回连接的散列码	Integer (整型) 类数据	不是
GetLifetimeService()	Public	该方法继承自 MarshalByRefObject 类, 返回一个终身服务对象。该对象控制着当前连接类实例的终身政策。默认的终身服务返回一个 ILease 类的对象。该方法应仅用于与远程激活对象的连接。Leasing Distributed Garbage Collector (LDGC) 负责将一个租用 (终身服务) 与远程激活的对象关联起来。一旦租用期满, 对象就被清除	Object 类数据	不是
GetOleDbSchemaTable (ByVal vguiSchema As Guid, ByVal vobjRestrictions As Object)	Public	该方法用于为 vguiSchema 检索模式表和相关约束条件	DataTable 类数据	不是
GetService (ByVal vtypService As Type)	Protected	该方法继承自 Component 类, 返回一个表示服务 (vtypService) 的对象, 该对象由组件提供	Object 类数据	不是
GetType()	Public	该方法继承自 Object 类, 返回对象的类型。更确切点说, 是返回元数据, 与对象所继承的类相关联。如果需要该类的名称, 可以用 cnmUserMan.GetType.ToString 读取	Type 类数据	不是
InitializeLifetimeService()	Public	该方法继承自 MarshalByRefObject 类, 可用来向连接对象提供自己的租用, 从而控制连接的存在时间	Object 类数据	不是

续表

方法名	可访问性	说 明	返回值	可重载的
MemberwiseClone()	Protected	该方法继承自 Object 类，创建连接的一个副本。克隆出的连接是原连接的一个浅拷贝。“浅”指的是原连接被克隆，并且原连接对于其他对象的引用也被克隆。这意味着获得了一个原连接的新拷贝，与初始连接有同样的对象引用。这是与深拷贝相比较而言的，在深拷贝中，原连接被复制，原连接所引用的对象也都被复制	Object 类数据	不是
Open()	Public	该方法不能被忽略，用于在当前属性设置下打开与数据源的连接。在不同情形下将会抛出不同的异常（请参阅本章稍后“连接类异常处理”部分）		不是
ReferenceEquals (ByVal vobjEqual1 As Object, ByVal vobjEqual2 As Object)	Public	该方法用于比较索引，因此它可以用于检查两个连接实例是否指向同一个连接	Boolean （布尔）类数据	不是
ReleaseObjectPool()	Public Shared（全局共享）	当最后一个内在的 OLE DB 提供程序被释放时，该方法用来指明连接池可以清除了		不是
ToString()	Public	该方法继承自 Object 类，返回一个表示连接的字符串。如果用 MsgBox (cnnUserMan.ToString())来调用该方法，消息框所显示的信息应包括文本 System.Data.OleDb.OleDbConnection 。该文本是对象（ OleDbConnection ）的名称，以及包含该对象的名称空间（ System.Data.OleDb ）的名称	String 类数据	不是

3A.2.7 打开一个连接

一旦设置了打开连接所需的属性，就可以真正打开连接了。如果连接属性设置正确的话，打开连接就是十分简单的。有关示例请参阅程序清单 3A.1。

程序清单 3A.1 打开一个连接

```

1 ' 声明一个连接对象
2 Dim cnnUserMan As SqlConnection
3
4 ' 实例化连接对象
5 cnnUserMan = New SqlConnection()
```



```
6 ' 建立连接字符串
7 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
8 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
9 ' 打开连接
10 cnnUserMan.Open()
```

注意, 方法 **Open** 并无任何参数, 因此你需要确认已经设置了连接数据源所必需的属性。如果所需属性没有设置完全或者连接已经打开了, 那么就会抛出一个异常。相关细节请参阅本章稍后“**ConnectionString** 属性异常”部分。

3A.2.8 关闭一个连接

当不再需要某个连接时, 就应将其关闭, 以避免不必要地占用内存。如果连接属于某个连接池的话, 那么在关闭该连接的同时也将把它返回给连接池。可以如下所示关闭一个连接:

```
cnnUserMan.Close()
```

在调用 **Close** 方法时, 该方法会试着在连接关闭之前等候完成所有的事务处理。如果在一段时间后仍有某事务处理没有完成, 那么就会抛出一个异常。

3A.2.9 清除一个连接

可以调用 **Dispose** 方法来清除连接, 如下所示:

```
cnnUserMan.Dispose()
```

Dispose 方法会破坏对连接的引用, 并通知 **Garbage Collector** 可以运行了, 然后再破坏连接本身。当然, 这里假定该连接仅有一个引用。

3A.2.10 复制连接

有时用户可能希望或者需要复制连接对象。这相当容易, 但是对于如何进行复制就要小心了。有两种复制连接对象的方法:

- 浅拷贝 (shallow copying): 要克隆原始连接对象时可以用这种方法。这意味着在复制原始连接时, 它所具有的对象引用也会被复制。注意, 复制的是引用而非引用的对象本身。可以用受保护的 **MemberwiseClone** 方法实现浅拷贝。
- 深拷贝 (deep copying): 与浅拷贝不同, 在需要原始连接所引用的全部对象的副本时, 应该用深拷贝方法。这也称为 **cloning** (克隆), 因为得到的是两个完全相同的连接。如果需要这种拷贝, 你的连接类 (connection class) (继承自 **SqlConnection** 或 **OleDbConnection** 类) 必须先实现 **ICloneable** 接口, 然后就可以用该接口的方法 **Clone** 来执行实际的克隆了。

3A.2.11 比较两个连接对象

如果已经复制了一个连接对象或者将一个连接变量设为与另一个连接变量等价, 则很难

知道连接类的两个实例究竟是同一个连接，还是指向内存中同一个位置。在这里需要提醒你一下，从编程的角度来讲，这并不太难，可用下列内容进行比较：

- **Is** 运算符
- **Equals** 方法
- **ReferenceEquals** 方法

使用 Is 运算符进行比较

使用 Is 运算符可能是比较两个连接对象最简单的方法了。如下所示：

```
blnEqual = cnnUserMan1 Is cnnUserMan2
```

如果 **cnnUserMan1** 与 **cnnUserMan2** 指向内存中同一个位置，则 **Boolean** 变量 **blnEqual** 将被设为 **True**，否则被设为 **False**。

使用 Equals 方法进行比较

Equals 方法继承自 **Object** 类，实际上是用于比较值而非引用。当两个连接变量具有指向内存中同一个位置的引用时，它们实际上具有同样的“值”。因此，可以在连接对象的比较中使用 **Equals** 方法。**Equals** 方法有两种性质。一种用来检查由参数移值的连接对象是否等价于执行该方法的连接对象如下所示：

```
blnEqual = cnnUserMan1.Equals(cnnUserMan2)
```

如果 **cnnUserMan1** 与 **cnnUserMan2** 指向同一连接，则 **Boolean** 变量 **blnEqual** 将被设为 **True**，否则被设为 **False**。

第二种，也就是该方法的可重载版本，它把两个连接对象视为参数并检查它们是否相等，如下所示：

```
blnEqual = cnnUserMan1.Equals(cnnUserMan2, cnnUserMan3)
```

如果 **cnnUserMan2** 与 **cnnUserMan3** 指向同一连接，则 **Boolean** 变量 **blnEqual** 将被设为 **True**，否则被设为 **False**。通过该方法的可重载版本，可以真正同使用该方法的连接对象分离开对两个连接对象进行比较。

使用 ReferenceEquals 方法进行比较

ReferenceEquals 方法是用于比较引用的，因此它应优先于用来比较值的 **Equals** 方法。**ReferenceEquals** 方法的使用如下所示：

```
blnEqual = cnnUserMan1.ReferenceEquals(cnnUserMan2, cnnUserMan3)
```

如果 **cnnUserMan2** 与 **cnnUserMan3** 指向同一连接，则 **Boolean** 变量 **blnEqual** 将被设为 **True**，否则被设为 **False**。如果连接参数中包含 **Null**，则都将返回 **False**。



Equals 和 **ReferenceEquals** 都是继承的方法，其参数都是 **Object** 数据类型。这意味着可以使用自基本数据类型 **Object** 继承而来的任意数据类型进行比较。

3A.2.12 连接状态操作

当处理连接时，在试图获取或设置连接的某属性或执行某方法之前，检查一下连接的状态常常是一个很好的做法。可以使用 **Connection** 类的 **State** 属性来测定当前状态（相关细节请参阅本章前面“连接类属性”部分）。可以凭 **ConnectionState** 的枚举值检查 **State** 属性。有关 **ConnectionState** 枚举值成员的清单请参阅表 3A.8。

表 3A.8 ConnectionState 枚举值成员

名称	值	说明
Closed	0	连接已经关闭了
Open	1	连接是打开的
Connecting	2	当前，连接与一个数据源相连
Executing	4	连接正在执行一个命令
Fetching	8	连接正在从数据源检索数据
Broken	16	连接出错，不能使用。如果网络损坏，则常会出现该值。该状态下惟一有效的方法是 Close ，并且所有属性都是只读的

比较 State 与 ConnectionState

可以使用下列语句对 **State** 属性和 **ConnectionState** 的枚举值进行比较：

```
If CBool(cnnUserMan.State And ConnectionState.Open) Then
```

使用运算符 **And** 在两个枚举值间按位比较。这种情况是在检查连接是否打开。如果要查看连接是否关闭，可以这样做：

```
If CBool(cnnUserMan.State And ConnectionState.Closed) Then
```

3A.2.13 连接入池

连接入池（connection pooling）或应用程序间的连接共享与重用，可以自动应用于 ADO.NET 中的 OLE DB .NET 数据提供程序。这很重要，因为连接入池可以在连接被重用时保留资源。然而，SQL Server.NET 数据提供程序默认使用隐式入池模型，并且可以用 **ConnectionString** 属性来控制 **SqlConnection** 对象的隐式入池行为。

连接池仅在 **ConnectionString** 属性这一点上是独特的。这意味着在任何给定池中的全部连接都具有完全相同的连接字符串。必须知道，连接字符串中的空白将通向两个其他方面相同的连接而被添加到不同的连接池中。在程序清单 3A.2 中，由于连接 **cnnUserMan2** 的连接字符串中有空白，所以 **SqlConnection** 的两个实例将不会被加入同一个池。在分开 **Password** 和 **Data Source** 值名的分隔符（分号）后面，有一个额外的空格字符。

程序清单 3A.2 `ConnectionString` 属性中的空白

```
cnnUserMan1.ConnectionString = "User Id=UserMan;Password=userman;" & _
    "Data Source='USERMANPC';Initial Catalog='UserMan'"
cnnUserMan2.ConnectionString = "User Id=UserMan;Password=userman;" & _
    "Data Source='USERMANPC';Initial Catalog='UserMan'"
```

同一连接池测试

测试连接是否进入同一个池的一个方法是建立具有相同连接字符串的两个连接，并确保 **Max Pool Size**（最大池容量）的值设为 1。当打开第一个连接时，就会为该特定连接字符串建立一个新池，并添加该连接。当试图打开第二个连接时，对象入池程序（object pooler）会尝试将连接加入现存池中。由于现存池不允许再有任何连接（已达到最大池容量），该请求会被加入请求队列中。当过了 **ConnectionTimeout** 时期后，如果第一个连接没有返还给连接池（通过关闭连接），第二个连接就会发生超时。一旦建立了这种场景，还可以改变连接字符串，如果运行时中发生超时的话，就说明连接的确进入了同一个池。

`SqlConnection` 数据类型的连接入池

当设置 `SqlConnection` 类的 `ConnectionString` 属性时，必须将 **Pooling** 值设为 'true' 或不考虑该值。因为 **Pooling** 的默认值为 'true'，所以，在默认情况下打开连接时，`SqlConnection` 对象取自连接池。如没有池存在，就创建一个以保存打开的连接。

`ConnectionString` 属性的 **Min Pool Size**（最小池容量）与 **Max Pool Size**（最大池容量）值决定了一个池可以保存多少个连接。连接会不断加到池中，直到达到最大池容量。接着，对象入池程序会把请求加入请求队列中，如果在超时期限之前没有连接返还给连接池，就会发生超时。只有在使用 **Close** 方法关闭时，连接才返还给连接池。即使连接被终止，也是如此。一旦被终止的连接或无效连接被返还给连接池，对象入池程序就会从池中清除该连接。上面的操作会定期进行，这时对象入池程序会扫描被终止的连接。

从连接池中清除连接的另一种方法是指定 `ConnectionString` 属性中 **Connection Lifetime** 的值。该值默认为 0，意味着不会自动清除池中的连接。如果指定一个不同的值（以秒计），一旦连接被返还给连接池，就会将该值与创立连接的时间和当前时间（之差）相比较。如果连接存在时间超过了 **Connection Lifetime** 指明的值，那么将从池中清除该连接。

在连接池中重置连接

Connection Reset 值用于确定一个连接返回连接池时是否需要重置。默认值是 'true'，标志着连接返回池时会被重置。如果你曾在连接被打开的状态下修改了连接属性，这点就会很有意义。如果可以确定无论何时打开连接其状态都是一样的，那么就可以把该值设为 'false'，以避免到服务器的额外来回，从而减少网络负载并节省时间。

`OleDbConnection` 型连接入池

尽管 OLE DB .NET Data Provider 提供了自动入池，但仍可以手工忽略或是从编程方面禁用它。

禁用 OLE DB 入池

如果在连接字符串中指定了 **OLE DB Services** 的值，那么就可以禁用自动入池。下面的连接不会自动进行连接入池：

```
cnnUserMan.Open("Provider=SQLOLEDB;OLE DB Services=-4;Data  
Source=USERMANPC" & _"User ID=UserMan;Password=userman;Initial  
Catalog=UserMan")
```

阅读下面网址的“**OLE DB Programmer's Reference**”（OLE DB 程序员参考）文档可以了解有关 OLE DB Services 值的更多信息：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbabout_the_ole_db_documentation.asp。

清除对象入池

由于在创建了提供程序后，连接池就被缓存了，所以在使用连接完成操作后，要调用连接的 **ReleaseObjectPool** 方法，这点很重要。

使用 ReleaseObjectPool 方法

当最后一个提供程序被释放以后，**ReleaseObjectPool** 方法会告诉连接入池程序可以清除连接池了。关闭连接后，如果确定在 OLE DB Services 通常保持入池连接有效的期限内不再需要该连接了，就应调用该方法（请参阅程序清单 3A.3）。

程序清单 3A.3 调用 ReleaseObjectPool 方法

```
...  
' 打开连接  
cnnUserMan.Open()  
' 做你要做的事  
...  
' 关闭连接并释放池  
cnnUserMan.Close()  
cnnUserMan.ReleaseObjectPool()
```

3A.2.14 事务处理

事务处理（**transaction**）是一种将相关数据库操作编组的方法，这样，如果其中一个操作失败了，整组的事物处理就会全部失败。同样，如果它们都操作成功了，将永久性改变数据源。因此，事务处理是一个安全网，确保数据保持同步。

事务处理的一个典型例子是储蓄系统。假定你正从往来账户上转一笔钱到储蓄存款账户。该操作需要更新两个地方：在往来账户上记入这笔贷款，并且在储蓄存款账户上将该款项记入借方。我们假定该操作正在进行中，并且已经在往来账户上记入了贷款。这时，因为某种原因，系统崩溃了，该款项没有记入储蓄存款账户的借方。这很不好，并肯定会令人感到不快。如果用一个事务处理来执行该操作，那么一旦系统恢复备份并开始运行，记入往来账户的款项将被退回重来。虽然我在此处简化了整个过程，但为什么要使用事务处理读者应该很清楚了。

ADO.NET 提供了两种操作应用程序内事务处理的方法：手动和自动。

定义事务处理边界

所有的事务处理都有一个边界 (boundary)，它是一个事务处理中全部资源的范围或“周围边界”。边界内的资源共用同一事务处理标识符。这意味着全部资源，例如数据库服务器，都在内部被分配上该标识符，它也是事务处理边界内惟一的标识符。事务处理边界是抽象的，像一个看不到的框架。边界可以被扩展和缩小。至于怎样实现这项操作，则完全取决于应用是自动（隐式）事务处理还是手动（显式）事务处理。

手动事务处理

ADO.NET 中的 .NET 数据提供程序支持通过连接类 (connection class) 中的方法进行手动事务处理。在手动事务处理过程中，当试图执行一个处理方法前最好先检查一下连接的状态。详细信息请参阅本章前面的“连接状态操作”部分。

开始一个手动事务处理

若想开始一个事务处理则需要调用连接类的 **BeginTransaction** 方法。在执行参与事务处理的数据库操作之前，必须调用该方法。执行该调用的最简形式如程序清单 3A.4 所示。

程序清单 3A.4 以默认值开始事务处理

```
traUserMan = cnnUserMan.BeginTransaction()
```

连接对象 (cnnUserMan) 必须有效并且已经打开，否则将抛出 **InvalidOperationException** 异常。详细信息请参阅“开始事务处理方法的异常”。现在，traUserMan 保存着 **BeginTransaction** 方法所创建事务处理对象的一个引用。**BeginTransaction** 方法是可重载的，表 3A.9 与表 3A.10 展示了该方法的不同形式。

表 3A.9 SqlConnection 类的 BeginTransaction 方法

调用	说明
BeginTransaction() As SqlTransaction	该形式没有任何参数，应用于事务处理无嵌套的情形下，隔离等级为默认值（请参阅程序清单 3A.4）
BeginTransaction (ByVal venul isolationLevel As IsolationLevel) As SqlTransaction	将事务处理的隔离等级作为惟一参数。当想指定隔离等级时应使用该方法的这一形式（请参阅程序清单 3A.5）
BeginTransaction (ByVal vstrName As String) As SqlTransaction	将事务处理名称作为惟一参数。当采用默认隔离等级，要对事务处理进行嵌套，并为使识别而命名事务处理时，应使用该形式（请参阅程序清单 3A.6）。请注意，在 vstrName 中不能有空格，必须以字母或者以下字符之一开头：#（磅字符）或 _（下划线），之后的字符可以是数字（0~9）
BeginTransaction (ByVal venul isolationLevel As IsolationLevel , ByVal vstrName As String) As SqlTransaction	以事务处理的隔离等级和事务处理名称作为参数。要指定隔离等级，对事务处理进行嵌套，并且因此命名事务处理以便识别时，应使用该形式（请参阅程序清单 3A.7）。请注意，在 vstrName 中不能有空格，并且必须以字母或者以下字符之一开头：#（磅字符）或 _（下划线），之后的字符可以是数字（0~9）

表 3A.10 OleDbConnection 类的 BeginTransaction 方法

调用	说明
BeginTransaction() As SqlTransaction	该形式没有任何参数，应用于事务处理无嵌套的情形下，隔离等级为默认值（请参阅程序清单 3A-4）
BeginTransaction(ByVal venuIsolationLevel As IsolationLevel) As OleDbTransaction	将事务处理的隔离等级作为惟一参数。在想指定隔离等级时应使用该方法的这一形式。参见程序清单 3A.5，只需将例子中的 SqlConnection 和 SqlTransaction 换成 OleDbConnection 和 OleDbTransaction 即可

BeginTransaction 方法的不同形式都会返回一个继承自 **SqlTransaction** 或 **OleDbTransaction** 类的事务处理实例。隶属于 **System.Data** 名称空间的 **IsolationLevel** 的枚举值指定了局部事务处理锁定连接的行为。如果在事务处理过程中改变了隔离等级，那么服务器将对所有剩余语句应用新的隔离等级。有关 **IsolationLevel** 枚举值的概述，请参阅表 3A.11。

表 3A.11 IsolationLevel 枚举值的成员

名称	值	说明
Chaos	16	不能重写更高隔离等级的事务处理程序中的未决改变
ReadCommitted	4096	尽管共享锁定一直会持续到读操作结束 [这避免了肮脏读取 (dirty read)], 但数据仍可在事务处理结束之前改变。这会导致幻像数据或不可重复读取。这意味着无法保证所读取的数据与下次执行读取请求时相同
ReadUncommitted	256	可能发生肮脏读取，因为没有共享锁定起作用，也没有任何独占锁定被尊重
RepeatableRead	65536	查询所涉及的数据都被锁定，从而使其他用户无法更新数据。这样虽然可以防止不可重复读取 (nonrepeatable read)，但仍可能出现幻像数据
Serializable	1048576	用一个区域锁锁定 DataSet ，直到事务处理结束为止，防止其他用户在 DataSet 中更新或插入行（请参阅第 3B 章“使用 DataSet Class”）
Unspecified	-1	由于正在使用一个非指定的隔离等级，所以无法确定隔离等级

程序清单 3A.5 以非默认隔离等级开始事务处理

```

1 Dim cnnUserMan As SqlConnection
2 Dim traUserMan As SqlTransaction
3
4 ' 打开连接等
5 ...
6 ' 按指定隔离等级开始事务处理
7 traUserMan = cnnUserMan.BeginTransaction(IsolationLevel.ReadCommitted)

```

程序清单 3A.6 开始一个命名了的 SQL 事务处理

```

1 Dim cnnUserMan As SqlConnection

```

```

2 Dim traUserMan As SqlTransaction
3 Const STR_MAIN_TRANSACTION_NAME As String = "MainTransaction"
4
5 ' 打开连接等
6 ...
7 ' 开始一个命名了的事务处理
8 traUserMan = cnnUserMan.BeginTransaction(STR_MAIN_TRANSACTION_NAME)

```

程序清单 3A.7 以非默认隔离等级开始一个命名了的 SQL 事务处理

```

1 Dim cnnUserMan As SqlConnection
2 Dim traUserMan As SqlTransaction
3 Const STR_MAIN_TRANSACTION_NAME As String = "MainTransaction"
4
5 ' 打开连接等
6 ...
7 ' 以指定隔离等级开始一个命名了的事务处理
8 traUserMan=cnnUserMan.BeginTransaction(IsolationLevel.ReadCommitted, _
9 STR_MAIN_TRANSACTION_NAME)

```

在程序清单 3A.6 和程序清单 3A.7 中都使用了一个常量来命名事务处理，从而可以更容易地识别欲进行的事务处理。有关详细信息请参阅下节“事务处理嵌套及 `SqlTransaction` 中事务处理保存点的使用”。



没必要存储返回的事务处理对象，也就是说可以省略程序清单 3A.4、3A.5、3A.6 和 3A.7 中的 `traUserMan = bit`。在这里需要提醒你一下，因为以后可能要提交或者返回未决变化，所以保存返回事务处理对象就没什么意义了。这仅在保存了事务处理对象的引用时可能有意义。ADO 中返回和提交功能都是连接类的方法。而在 ADO.NET 中，这一功能是事务处理类的一部分。

事务处理嵌套及 `SqlTransaction` 中事务处理保存点的使用

该主题仅对 `SqlConnection` 和 `SqlTransaction` 类有效。如果你正在进行嵌套事务处理，那么最好给每个事务处理指定一个容易辨认的名称，以便能更容易地分辨出这些事务处理。虽然这里说的是“这些事务处理”，但在 ADO.NET 中，在一个连接中不能有多个事务处理。而在 ADO 中，对于一个连接而言，可以通过多次调用 `BeginTransaction` 方法并为每个事务处理指定名称来开始多个事务处理。ADO.NET 也以 `BeginTransaction` 方法开始。然而，`BeginTransaction` 方法所返回的事务处理对象具有 `Save` 方法，可用来达到几乎相同的目的。我认为 ADO.NET 中的事务处理嵌套变得更容易了，但是否真的如此，还要由读者自己判断。

下面我们来看看嵌套事务处理的新方法。它并不像嵌套那样，而只是将事务处理中可以返回到的特定点存储起来。`SqlTransaction` 类的 `Save` 方法用于存储事务处理中的参考点，如程序清单 3A.8 所示。

程序清单 3A.8 存储事务处理中的参考点

```

1 Dim cnnUserMan As SqlConnection
2 Dim traUserMan As SqlTransaction
3 Dim cmdUserMan As SqlCommand
4 Const STR_MAIN_TRANSACTION_NAME As String = "MainTransaction"
5 Const STR_FAMILY_TRANSACTION_NAME As String = "FamilyUpdates"
6 Const STR_ADDRESS_TRANSACTION_NAME As String = "AddressUpdates"
7
8 ' 打开连接等
9 ...
10 ' 开始命名了的事务处理
11 traUserMan = cnnUserMan.BeginTransaction(STR_MAIN_TRANSACTION_NAME)
12 ' 更新系列表
13 ...
14 ' 存储事务处理参考点
15 traUserMan.Save(STR_FAMILY_TRANSACTION_NAME)
16 ' 更新地址表
17 ...
18 ' 存储事务处理参考点
19 traUserMan.Save(STR_ADDRESS_TRANSACTION_NAME)
20 ' 返回地址表更新
21 traUserMan.Rollback(STR_FAMILY_TRANSACTION_NAME)

```

我们在程序清单 3A.8 中打开了连接，进行了事务处理，更新了系列表。此后，在下次数据库更新之前保存了一次参考点。这时，更新了地址表，并且又保存了一次参考点。最终我们得到一个真正的诀窍：返回到事务处理出现时指定的参考点，也就是系列表更新后的那一点（有关事务处理类 **Rollback** 方法的详细信息，请参阅“中止手动事务处理”）。执行了返回操作以后，地址表的更新被取消，而对系列表的更改仍在原地。说“仍在原地”指的是事务处理仍未被提交，但如果使用 `traUserMan.Commit()` 执行了事务处理类的 **Commit** 方法，那么系列表的更新将永久性生效了。

OleDbTransaction 中的事务处理嵌套

该主题仅对 **OleDbConnection** 和 **OleDbTransaction** 类有效。不像 ADO 中那样，可以通过多次调用 **BeginTransaction** 方法在一个连接上开始多个事务处理。在 ADO.NET 中，只能有一个事务处理，且该事物处理也用 **BeginTransaction** 方法开始。然而，**BeginTransaction** 方法返回的事务处理对象具有 **Begin** 方法，可用来实现嵌套。程序清单 3A.9 给出了一个使用 **OleDbTransaction** 类 **Begin** 方法的示例。

程序清单 3A.9 开始嵌套的 OleDb 事务处理

```

1 Dim cnnUserMan As OleDbConnection
2 Dim traUserManMain As OleDbTransaction
3 Dim traUserManFamily As OleDbTransaction
4 Dim traUserManAddress As OleDbTransaction
5
6 ' 打开连接等
7 ...

```

```

8 ' 开始主要事务处理
9 traUserManMain = cnnUserMan.BeginTransaction()
10 ' 更新系列表
11 ...
12 ' 开始嵌套的事务处理
13 traUserManFamily = traUserManMain.Begin()
14 ' 更新地址表
15 ...
16 ' 开始嵌套的事务处理
17 traUserManAddress = traUserManFamily.Begin()
18 ' 返回地址表更新
19 traUserManAddress.Rollback()

```

中止手动事务处理

如果由于某种原因需中止事务处理（也就是说，要返回自开始事务处理以来所做的全部更改），那么就需要调用事务处理类的 **Rollback** 方法。这可以确保数据源没有发生任何变化，更确切地说就是数据源返回其初始状态。可以如下进行调用：

```
traUserMan.Rollback()
```

你也可以用 **Rollback** 方法的重载形式来指定事务处理名称，如程序清单 3A.8 所示。**Rollback** 方法既不能在调用连接类的 **BeginTransaction** 方法之前使用，也不能在调用事务处理类 **Commit** 方法之后调用。否则，就会发生异常。

使用命名了的事务处理

尽管不移值事务处理的名称也可以调用 **Rollback** 方法，但建议在使用 **SqlTransaction** 类（**OleDbTransaction** 类没有这一特性）的时候使用事务处理名。开始事务处理时，使用一个允许将名称作为参数移值的重载 **BeginTransaction** 方法，如程序清单 3A.6、3A.7 和 3A.8 所示。这样就总是可以使用事务处理类的重载 **Rollback** 方法，并移值事务处理的名称。在使用 **Rollback** 方法的标准形式时，所有未决变化都将被返回。我个人认为如果总是在开始事务处理时为事务处理命名、保存参考点，并且返回事务处理，就会使代码更易读。

提交手动事务处理

完成对数据的操作后，就应该调用事务处理类的 **Commit** 方法来向数据源提交这些改变进行操作。通过调用 **BeginTransaction** 方法开始事务处理以来，对数据所作的全部改变都将被应用到数据源。事实上，这仅对于 **OleDbTransaction** 类是完全适用的。如果你使用的是 **SqlTransaction** 类，则可以保存多个参考点，并可以通过 **Rollback** 方法返回它们中的一个或者更多个。在这种情形下，那些被返回的操作显然不会被提交。

检定运行中的事务处理的隔离等级

如果无法确定一个运行中的事务处理的隔离等级，则可以事务处理类的 **IsolationLevel** 属性来检定该值，如程序清单 3A.10 所示。

程序清单 3A.10 检定运行中事务处理的隔离等级

```

' 以文本形式返回隔离等级
MsgBox(traUserMan.IsolationLevel.ToString)
' 以整型值形式返回隔离等级
intIsolationLevel = traUserMan.IsolationLevel

```

事务处理类检查

事务处理类只能由连接类的 **BeginTransaction** 方法实例化，而不能被继承。一旦用 **BeginTransaction** 方法开始了某个事务处理，就开始在 **BeginTransaction** 返回的事务处理对象上执行所有更进一步的事务处理操作。从表 3A.12 中可以看到 **SqlTransaction** 和 **OleDbTransaction**，两个事务处理类的惟一属性。

表 3A.12 **SqlTransaction** 和 **OleDbTransaction** 类的属性

属性名	可访问性	说明	读	写
IsolationLevel	Public	指定了事务处理的隔离等级。有关详细信息，请参阅本章前面介绍的内容	是	否

在表 3A.13 中可以看到来自 **System.Data.SqlClient** 名称空间的 **SqlTransaction** 和来自 **System.Data.OleDb** 名称空间的 **OleDbTransaction** 类的方法，但没有列出继承方法和可忽略方法。

表 3A.13 **SqlTransaction** 和 **OleDbTransaction** 类的方法

方法名	说明	返回值	可重载
Begin() (仅用于 OleDbTransaction)	开始一个新事务处理，将其嵌套于当前事务处理中。请参阅本章前面“ OleDbTransaction 中的事务处理嵌套”部分	OleDbTransaction 对象	是
Commit()	提交当前数据库事务处理。请参阅本章前面“提交手动事务处理”部分		不是
Rollback()	返回到未决变化以前，以使未决变化不被应用于数据库。请参阅本章前面“中止手动事务处理”部分		是
Save() (仅用于 SqlTransaction)	保存参考点，可以使用 Rollback 方法返回到该点		不是

手动事务处理小结

实际上，手动操作事务处理并不是个大任务。下面列出了需要记住的几步：

- 使用连接类的 **BeginTransaction** 方法开始事务处理。每个连接只能执行一次 **BeginTransaction**，否则也不会拥有并行事务处理。在同一个连接中再次使用 **BeginTransaction** 之前，需要递交运行中的事务处理。**BeginTransaction** 返回一个

回退或提交数据源变化所需的事务处理对象。

- 通过使用事务处理类的 **Rollback** 方法回滚事务处理。
- 通过使用事务处理类 **Commit** 方法提交事务处理。

自动事务处理

自动事务处理与手动事务处理的区别在于不一定非要应用它们。这里说“不一定”的意思是：如果使用 .NET 数据提供程序数据类的对象不在一个事务处理中，那么数据源也不在。即，如果应用程序使用了事务处理，那么就会自动使用数据源。

OLE DB .NET 和 SQL Server .NET 数据提供程序自动加入一个事务处理，并从 Windows 2000 Component Services 环境中获取事务处理的细节。

3A.2.15 连接类异常处理

本节说明了如何处理连接中的异常。异常或多或少与过去所说的可捕获错误相似，可以使用 **Try...Catch...End Try** 语句结构捕获它。如果使用的是 **SqlConnection** 类，那么有关异常校正，请在随后的“**SqlConnection** 异常”部分查找属性或方法。请注意，其中仅提到了非继承属性和方法的异常。因此，继承自 **Object** 类的 **Equals** 方法所导致的异常并没有列出。在本章的示例中仅列举了一些常见异常，而在本章稍后的“**OleDbConnection** 异常”部分将讨论 **OleDbConnection** 异常。

SqlConnection 异常

下面将展示如何处理在使用 **SqlConnection** 类过程中出现的异常。

为什么出现异常？

对于每种异常，我都会给出一段代码，以帮助你确定为什么会发生异常。当 **Try...Catch...End Try** 结构中 **Try** 部分的代码包含了多过程且可能导致异常的调用时，这里介绍的内容就很有用了。要知道，这里所讲到的只是简单的代码，只是为了帮助你了解一下如何进行错误处理。在所提供的示例中，告诉你如何发现是什么方法、什么属性等出现了异常。然而，有些方法和属性可能出现不止一类异常，当需要建立多捕获模块时就会出现这种情况，如下所示：

```
Try
...
Catch objInvalid As InvalidOperationException
...
Catch objArgument As ArgumentException
...
End Try
```

有关异常处理的详细信息，请参阅第 5 章。

BeginTransaction 方法异常

如果在无效的或已关闭的连接中调用 **BeginTransaction** 方法就会抛出 **InvalidOperationException** 异常。可以如下检查连接是否已关闭:

· 检查连接是否是打开的

```
If CBool(cnnUserMan.State And ConnectionState.Open) Then
```

在提交或回退第一个事务处理之前, 如果试图通过两次调用 **BeginTransaction** 方法进行并行事务处理, 也会抛出 **InvalidOperationException** 异常。

程序清单 3A.11 显示了如何检查是否由于 **BeginTransaction** 方法而导致了异常。

程序清单 3A.11 检查是否由于 BeginTransaction 导致了异常

```
1 ...  
2 Try  
3 ...' 开始事务处理  
4 Catch objException As Exception  
5     ' 检查是否由于 BeginTransaction 方法导致出现异常  
6     If objException.TargetSite.Name = "BeginTransaction" Then
```

ConnectionString 属性异常

如果在连接已被打开或中断时试图设置属性, 就会抛出 **InvalidOperationException** 异常。程序清单 3A.12 显示了如何检查是否因为 **ConnectionString** 属性而抛出了 **InvalidOperationException** 异常。

程序清单 3A.12 检查是否由于 ConnectionString 导致异常

```
1 ...  
2 Try  
3 ...' 设置连接字符串  
4 Catch objException As Exception  
5     ' 检查是否由于 ConnectionString 引起异常  
6     If objException.TargetSite.Name = "set_ConnectionString" Then
```

ConnectionTimeout 属性异常

当试图将该属性值设定成小于 0 值时就会抛出 **ArgumentException** 异常。实际上, 由于 **ConnectionTimeout** 属性是只读的, 所以不能直接设置它。然而, 可以通过 **ConnectionString** 属性中的 **Connection Timeout** 值设定该属性。**ConnectionString** 属性允许把该值名设定为小于 0 的值, 但要等到打开连接时才会验证。那时就会抛出 **ArgumentException** 异常。程序清单 3A.13 显示了如何检查是否因为 **ConnectionTimeout** 属性而抛出 **ArgumentException** 异常。

程序清单 3A.13 检查是否由于 ConnectionTimeout 导致异常

```
1 ...  
2 Try
```

```

3 ...' 设置 connection timeout
4 Catch objException As Exception
5     ' 检查是否将 connectiontimeout 设成了无效值
6     If objException.TargetSite.Name = "SetConnectTimeout" Then

```

Database 属性及 ChangeDatabase 方法异常

如果连接没有打开, 就会抛出 **InvalidOperationException** 异常。事实上, 由于 **Database** 属性是只读的, 所以不能直接设置它。然而, 可以通过 **ChangeDatabase** 方法或使用 Transact-SQL 语句来设置该属性。因此, 可以通过执行简单的 SQL 语句 USE master 并使用 **SqlCommand** 类的 **ExecuteNonQuery** 方法来从当前数据库切换到主数据库。此刻, **Database** 属性会动态更新, 而之后就可能抛出一个异常。



注意 SQL Server 中用的 Transact-SQL 是 ANSI SQL 标准在 Microsoft 中的语言。Transact-SQL 与 ANSI 完全兼容, 并具有许多增强了的性能。

程序清单 3A.14 显示了如何检查 **InvalidOperationException** 异常是否因为 **Database** 属性而抛出。

程序清单 3A.14 检查是否由于 Database 而导致了异常

```

1 ...
2 Try
3 ...' 设置 database
4 Catch objException As Exception
5     ' 检查我们是否试图在无效连接上改变数据库
6     If objException.TargetSite.Name = "ChangeDatabase" Then

```

Open 方法异常

当试图打开一个已打开或已中断连接时, 就会抛出 **InvalidOperationException** 异常。一旦发生了该异常, 只要关掉连接并再次打开就可以解决。程序清单 3A.15 显示了如何检查 **InvalidOperationException** 异常是否因为 **Open** 方法而抛出。

程序清单 3A.15 检查是否由于 Open 而导致了异常

```

1 ...
2 Try
3 ...' 打开数据库
4 Catch objException As Exception
5     ' 检查是否由于 Open 导致异常
6     If objException.TargetSite.Name = "Open" Then

```

OleDbConnection 异常

在使用 OLE DB .NET 数据提供程序时, 无论何时发生错误, 都会导致抛出

OleDbException 异常。该异常类不能被继承，其起源是 **ExternalException** 类。**OleDbException** 类至少有一个 **OleDbError** 类的实例。用户可以自己决定是否遍历 **OleDbException** 类包含的全部 **OleDbError** 类实例，以检查发生了什么错误。程序清单 3A.16 显示了具体的操作。该示例捕获了一个在试图打开一个连接时抛出的异常，但 **Catch** 区段代码可用于你遇到的所有 OLE DB 异常。

程序清单 3A.16 仔细检查 OleDbException 类

```

1 ...
2 Try
3     ' 打开连接
4     cnnUserMan.Open()
5 Catch objException As OleDbException
6     Dim objError As OleDbError
7
8     For Each objError In objException.Errors
9         MsgBox(objException.Message)
10    Next
11 End Try

```

程序清单 3A.16 显示了如何从 **OleDbException** 类的 **Errors** 类集中抽取所有 **OleDbError** 类实例。

检查 OleDbError 类

程序清单 3A.16 中列出的全部属性均用于显示错误信息，但在 **OleDbError** 类中还有一些重要的属性。有关全部属性的程序清单请参阅表 3A.14。顺便说一下，**OleDbError** 类是不可继承的类，它直接继承自 **Object** 类。

表 3A.14 OleDbError 类的属性

属性名	可访问性	说明
Message	Public	以一个 String 型变量返回错误的说明
NativeError	Public	以整型变量返回特定数据库的错误信息
Source	Public	以 String 型变量返回异常发生或产生异常的提供程序名称
SQLState	Public	以 String 型变量返回 ANSI SQL 标准的错误代码

表 3A.14 中的所有属性都是只读的，它们与 ADO 错误集中的属性有很多相同点。

检查 OleDbException 类

除了 **Errors** 集合之外，**OleDbException** 类还有其他一些属性和方法，如表 3A.15 和 3A.16 所示，这两个表都以字母排序的。

表 3A.15 OleDbException 类属性

属性名称	可访问性	说明	返回值	读	写
ErrorCode	Public	该属性继承自 ExternalException 类, 返回错误的错误标识数。该标识数是一个 HRESULT 或 Win32 错误代码	Integer 型数据	是	否
Errors	Public	这实际上是 OleDbError 类的集合。请参阅前面的“检查 OleDbError 类”部分	OleDbError Collection 型数据	是	否
HelpLink	Public	该属性继承自 Exception 类, 返回或设置指向 HTML 帮助文件的 URN 或 URL	String 型数据	是	是
HResult	Protected	该属性继承自 Exception 类, 返回或设置异常的 HRESULT。请参阅第 5 章中的一般错误处理部分		是	是
InnerException	Public	该属性继承自 Exception 类, 返回一个内部异常的引用。一般来说, 该属性保存有前面发生的异常或者引起当前异常的异常	Exception 型数据	是	否
Message	Public	该属性可被忽略, 也可参阅 OleDbError 类的 Message 属性		是	否
Source	Public	该属性可被忽略, 也可参阅 OleDbError 类的 Source 属性		是	否
StackTrace	Public	该属性继承自 Exception 类, 返回堆栈记录。可以用堆栈记录确定代码中发生错误的位置	String 型数据	是	否
TargetSite	Public	该属性继承自 Exception 类, 返回抛出异常的方法。有关 MethodBase 对象的详细信息, 可参阅第 5 章中的一般错误处理部分进行比较	MethodBase 型数据	是	否

表 3A.16 OleDbException 类的方法

方法名称	可访问性	说明	返回值
Equals()	Public	该方法继承自 Object 类, 在表 3A.6 中已有说明。请同时参阅本章前面“使用 Equals 方法进行比较”部分	Boolean 型数据
Finalize()	Protected	该方法继承自 Object 类, 在表 3A.6 中已有说明	None
GetBaseException()	Public	该方法继承自 Exception 类, 返回出现的原始异常。它在一个异常触发另一个异常等情形下有用。如果只抛出了一个异常, 那么返回的引用则指向当前异常	Exception 型数据
GetHashCode()	Public	该方法继承自 Object 类, 在表 3A.6 中已有说明。	Integer 型数据
GetObjectData(ByVal vobjInfo As SerializationInfo, ByVal vobjContext As StreamingContext)	Public	该方法继承自 Exception 类, 依据有关异常的信息设定 SerializationInfo 对象。有关 SerializationInfo 对象的详细信息, 请第 5 章一般错误处理部分	None

续表

方法名称	可访问性	说明	返回值
GetType()	Public	该方法继承自 Object 类，在表 3A.6 中已有说明	Type 型数据
MemberwiseClone()	Protected	该方法继承自 Object 类，在表 3A.6 中已有说明。请同时参阅本章前面“复制连接”部分	Object 型数据
ToString()	Public	该方法继承自 Object 类，在表 3A.6 中已有说明。用于 OleDbException 类时，返回充分证明了的异常名称。返回错误信息（ Message 属性）、内部异常名称（ InnerException 属性）和堆栈记录（ StackTrace 属性）	String 型数据

3A.3 使用命令对象

当需要对数据库进行查询时就要用到命令对象（command object）。命令对象是实现这项操作最简单和最容易的途径。在 ADO.NET 中，两个重要的命令类是 **OleDbCommand** 和 **SqlCommand**。**OleDbCommand** 类是 **System.Data.OleDb** 名称空间的一部分，而 **SqlCommand** 类是 **System.Data.SqlClient** 名称空间的一部分。



提示

如果无法确定代码中的类属于哪个名称空间，则可以，将光标移到该类上，而在弹出的工具提示栏中就会显示出该名称空间

3A.3.1 OleDbCommand 与 SqlCommand

如同在提供程序和连接类中一样，需要弄明白是否要连接到 MS SQL Server 7.0 或更高版本的数据源，或是其他的数据源。**SqlCommand** 用于 SQL Server，而 **OleDbCommand** 可用于任意其他 OLE DB 数据源。由于 **SqlCommand** 类针对 SQL Server 应用进行了优化，所以胜过 **OleDbCommand** 类。然而，可以你也根据自己的意愿以大致相同的方式使用 **OleDbCommand** 而非 **SqlCommand**。程序清单 3A.17 列出了如何声明和实例化 **SqlCommand**。

程序清单 3A.17 实例化 **SqlCommand**

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim strSQL As String
4
5 ' 连接实例化
6 cnnUserMan = New SqlConnection()
7 cnnUserMan.ConnectionString = "User ID=UserMan;" & _

```

```

8 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
9 ' 打开连接
10 cnnUserMan.Open()
11 ' 建立查询字符串
12 strSQL = "SELECT * FROM tblUser"
13 ' 命令实例化
14 cmmUserMan = New SqlCommand(strSQL, cnnUserMan)

```

在程序清单 3A.17 中所使用的是前面展示过的连接代码，因此不再多作解释。另外，这里声明了命令对象 `cmmUserMan` 和查询字符串 `strSQL`。一旦连接打开，就建立了查询字符串，然后使用该查询字符串和打开的连接实例化命令。这里，实例化命令对象的方法是四种不同的可用实例化过程之一，因为该方法是可重载的。

实例化 `SqlCommand` 对象

实例化 `SqlCommand` 对象的方法有四种（命令对象、查询字符串和连接字符串都已声明过了）：

- `cmmUserMan = New SqlCommand()` 对 `cmmUserMan` 进行了实例化，没有参数。如果用这种方式实例化命令，就需要在执行该命令或以其他方式使用命令前设置一些属性，例如 **Connection**。在实例化时，需要设置以下属性的初始值：

```

CommandText = ""
CommandTimeout = 30
CommandType = CommandType.Text
Connection = Nothing

```

- `cmmUserMan = New SqlCommand(ByVal vstrSQL As String)` 对 `cmmUserMan` 进行了实例化，参数为查询命令文本（`vstrSQL`）。此外，还需要通过设置 **Connection** 属性提供一个有效且打开了的连接。在实例化时，需要设置以下属性的初始值：

```

CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = Nothing

```

- `cmmUserMan = New SqlCommand(ByVal vstrSQL As String, ByRef rcnnUserMan As SqlConnection)` 对 `cmmUserMan` 进行了实例化，参数为查询命令文本（`vstrSQL`）和一个有效的并且已打开的连接 `rcnnUserMan`。在进行实例化时，需要设置以下属性的初始值：

```

CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = rcnnUserMan

```

- `cmmUserMan = New SqlCommand(ByVal vstrSQL As String, _ByRef rcnnUserMan As SqlConnection, ByRef rtraUserMan As`

SqlTransaction)对 cmmUserMan 进行了实例化, 参数为查询命令文本和一个连接对象。rcnnUserMan 必须是有效而且打开的连接, 而 rtraUserMan 必须是有效而且打开的事务处理, 并已经在 rcnnUserMan 上打开。当使用连接类的 **BeginTransaction** 方法开始事务处理时就要使用这种形式。在进行实例化时, 需要设置以下属性的初始值:

```
CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = rcnnUserMan
```

实例化 OleDbCommand 对象

下面列出了可用来实例化 OleDbCommand 对象的四种方法 (命令对象、查询字符串和连接字符串都已声明过了)

- cmmUserMan = New OleDbCommand() 对 cmmUserMan 进行了实例化, 没有参数。如果使用这种方式实例化命令, 则需要在执行该命令或以其他方式使用命令前设置一些属性, 例如 Connection。在进行实例化时, 需要设置以下属性的初始值:

```
CommandText = ""
CommandTimeout = 30
CommandType = CommandType.Text
Connection = Nothing
```

- cmmUserMan = New OleDbCommand(ByVal vstrSQL As String) 对 cmmUserMan 进行了实例化, 参数为查询命令文本 (vstrSQL)。此外, 还需要通过设置 **Connection** 属性提供一个有效且打开了的连接。在进行实例化时, 需要设置以下属性的初始值:

```
CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = Nothing
```

- cmmUserMan = New OleDbCommand(ByVal vstrSQL As String, ByRef rcnnUserMan As SqlConnection) 对 cmmUserMan 进行了实例化, 参数为查询命令文本 (vstrSQL) 和一个有效并且已打开的连接 rcnnUserMan。在进行实例化时, 需要设置以下属性的初始值:

```
CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = rcnnUserMan
```

- cmmUserMan = New OleDbCommand(ByVal vstrSQL As String, _ByRef rcnnUserMan As SqlConnection, ByRef rtraUserMan As SqlTransaction) 对 cmmUserMan 进行了实例化, 参数为查询命令文本和一个

连接对象。`rcnnUserMan` 必须是有效而且打开的连接，而 `rtraUserMan` 必须是有效而且打开的事务处理，并已经在 `rcnnUserMan` 上打开。当使用连接类的 **BeginTransaction** 方法开始事务处理时就要用到这种形式。在进行实例化时，需要设置以下属性的初始值：

```
CommandText = vstrSQL
CommandTimeout = 30
CommandType = CommandType.Text
Connection = rcnnUserMan
```

实际上，究竟选择哪种方式实例化命令对象（**SqlCommand** 或 **OleDbCommand**）还应取决于环境以及想要达到的目标。在多数场合下，你可能会选择 `cmmUserMan = New OleDbCommand()` 或 `cmmUserMan = New OleDbCommand(ByVal vstrSQL As String)`，除了选项 `cmmUserMan = New OleDbCommand(ByVal vstrSQL As String, ByRef rcnnUserMan As SqlConnection)` 中作为参数的连接以外，实质上都是一样的。当连接对象中已开始了一个事务处理时，通常使用 `cmmUserMan = New OleDbCommand(ByVal vstrSQL As String, ByRef rcnnUserMan As SqlConnection, ByRef rtraUserMan As SqlTransaction)`。

3A.3.2 命令类的属性

在表 3A.17 与表 3A.18 中以字母升序列出了所有全局命令类属性，并显示了等价的命令类构造函数参数（Command class constructor argument）（请参阅前面的“实例化 **SqlCommand** 对象”和“实例化 **OleDbCommand** 对象”部分）。请注意，一般在设置这些属性时只检查语法。在执行了 **Execute...** 方法后才进行真正的验证。

表 3A.17 SqlCommand 类的属性

属性名称	说明	命令构造函数的等价参数	默认值	读	写
CommandText	查询文本或欲执行的存储的程序名称。将该属性设置为与当前值相同的值时，略去分配。当 CommandType 属性被设置为 CommandType.StoredProcedure 时，需要把该属性设置成所存储程序的名称	VstrSQL	""（空字符串）	是	是
CommandTimeout	这是等候命令执行的秒数。如果在这段时间内没有执行命令，那么就会产生错误		30	是	是
CommandType	该属性决定了如何解释 CommandText 属性。下面列出的 CommandType 枚举值都是有效的。 StoredProcedure ：CommandText 属性必定保有一个所存储程序的名称， Text ：CommandText 属性解释为如同 SELECT 语句一样的查询，等等		Text	是	是

续表

属性名称	说明	命令构造函数的等价参数	默认值	读	写
Connection	与数据源的连接	RconnUserMan	Nothing	是	是
DesignTimeVisible	用于指明命令对象在 Windows Forms Designer (Windows 窗体设计程序) 控件中是否可见	False	False	是	是
Parameters	检索保存着 Transact-SQL 语句参数的 SqlParameterCollection		空集	是	否
Transaction	如果命令所附着的连接上已经开始了—个事务处理, 那么该属性就会保存这个事务处理对象	rtraUserMan	Nothing	是	是
UpdatedRowSource	<p>读取或设置在执行完命令后命令结果是否应用于源行以及应用的方式。该属性需要设置为下列 UpdateRowSource 的枚举值之一:</p> <p>Both: FirstReturnedRecord 与 OutputParameters 值的和。</p> <p>FirstReturnedRecord: 只有第一个返回记录中的数据被映射到数据集中改动了的行上。</p> <p>None: 返回的参数与/或行将被忽略。</p> <p>OutputParameters: 只有输出参数被映射到 Dataset 中改动了的行上</p>		Both (如果命令是自动生成的, 则为 None)	是	是

表 3A.18 OleDbCommand 类的属性

属性名称	说明	命令构造函数的等价参数	默认值	读	写
CommandText	查询文本或存储的欲执行程序的名称。将该属性设置为与当前值相同的值时, 则略去分配。当 CommandType 属性被设置为 CommandType.StoredProcedure 时, 需要把该属性设置成所存储程序的名称。只能在连接已打开并可用时设置该属性, 这意味着设置该属性时连接不能读取行	VstrSQL	"" (空字符串)	是	是
CommandTimeout	这是等候命令执行的秒数。如果在这段时间内没有执行命令, 则会产生错误		30	是	是
CommandType	<p>该属性决定了如何解释 CommandText 属性。下列值有效:</p> <p>StoredProcedure: CommandText 属性必定保存有一个所存储程序的名称。</p> <p>TableDirect: CommandText 属性必定保存有一个表的名称, 而该表的所有行都将被返回。这代替了类似 "SELECT * FROM TableName" 的查询。</p> <p>Text: CommandText 属性解释为如同 SELECT 语句一样的查询, 等等</p>		Text	是	是

续表

属性名称	说明	命令析构函数的等价参数	默认值	读	写
Connection	与数据源的连接	rcnnUserMan	Nothing	是	是
DesignTimeVisible	用于指明命令对象在 Windows Forms Designer (Windows 窗体设计程序) 控件中是否可见		False	是	是
Parameters	检索保存着 CommandText 属性参数的 OleDbParameterCollection		空集	是	否
Transaction	如果命令所附着的连接上已经开始了一个事务处理, 那么该属性就保存这个事务处理对象	rttraUserMan	Nothing	是	是
UpdatedRowSource	<p>获取或设置在执行完命令后命令结果是否应用于源行以及应用的方式。该属性需要设置为下列 OleDbParameterCollection 的枚举值之一:</p> <p>Both : FirstReturnedRecord 与 OutputParameters 值的和。</p> <p>FirstReturnedRecord: 只有第一个返回的记录中的数据被映射到数据集中改动了的行。</p> <p>None: 返回的参数与/或行将被忽略。行</p> <p>OutputParameters: 只有输出参数被映射到 Dataset 中改动了的行</p>		Both (如果命令是自动生成的, 则为 None)	是	是

3A.3.3 命令类的方法

表 3A.19 以字母持序列出了所有命令类 (SqlCommand 和 OleDbCommand) 的方法。

表 3A.19 命令类的方法

方法名称	说明
Cancel()	如果有命令在执行, 则取消当前执行命令。否则什么也不做
CreateParameter()	该方法可用于创建和实例化 SqlParameter 或 OleDbParameter 对象。与使用 <code>prmTest = New SqlParameter()</code> 实例化 SqlParameter 对象或使用 <code>prmTest = New OleDbParameter()</code> 实例化 OleDbParameter 对象等价。在这里需要提醒你一下, 这两种实例化参数对象的方法之间, 有不小的差别。 CreateParameter 方法把实例化了的参数对象加入参数集合, 使你节省了额外的代码行。可以使用 Parameters 属性检索参数集合
ExecuteNonQuery	执行 CommandText 属性中指定的查询。返回受影响行的号码。即使 CommandText 属性中包含一个返回行的语句, 也不会返回行。如果指定将行返回语句作为命令文本, 就会返回值-1。若指定了其他不能影响数据源中行的语句, 也会是如此

续表

方法名称	说明
<code>ExecuteReader()</code>	执行 CommandText 属性中指定的查询。返回一个 SqlDataReader 或 OleDbDataReader 型、只能向前的数据对象。在执行行返回语句时需要使用该方法，例如 SQL SELECT 语句。尽管可以用该方法执行无行返回的语句，但推荐使用 ExecuteNonQuery 方法来达到目的。这样做的原因是即使查询不返回行，该方法也会试图构建并返回一个仅能向前的数据对象。该方法是可重载的（请参阅“执行命令”）
<code>ExecuteScalar()</code>	该方法用于读取单一值，例如像 COUNT(*) 这样的总值。使用该方法比使用 ExecuteReader 方法开销小，编码也少（请参阅“执行命令”）
<code>ExecuteXmlReader()</code>	该方法用于以 XML 形式检索结果集合（请参阅“执行命令”）
<code>Prepare()</code>	该方法用于编译（预先的）命令。 CommandType 属性设为 Text 或 StoredProcedure 时可使用该方法。然而，如果 CommandType 属性设为 TableDirect ，将不用任何方法。该方法不带参数，可以如下所示简单地调用： <code>cmmUserMan.Prepare()</code>
<code>ResetCommandTimeout()</code>	将 CommandTimeout 属性重置为默认值。该方法不带参数，调用方法如下： <code>cmmUserMan.Prepare()</code>

3A.3.4 执行命令

现在你已经知道了如何建立并实例化命令对象，但还需要实际执行以完全利用命令对象的潜力。如你在命令类的不同方法中看到的一样，**ExecuteNonQuery**、**ExecuteReader**、**ExecuteScalar** 和 **ExecuteXmlReader** 方法就可以实现这一目的。如果你仔细读了这些方法的名称，就应明白该使用何种方法了（请参阅表 3A.20）。

表 3A.20 执行命令

方法名称	什么时候使用	示例
ExecuteNonQuery	与执行一个无返回行的命令，例如 DELETE 语句时，可以使用该方法。即便在使用该方法时用了行返回语句，也会因为结果集合被丢弃了，而没有任何意义	请参阅程序清单 3A.18 中所显示的如何向 tblUser 表中插入新用户（User99）
ExecuteReader	在执行一个行返回命令，例如 SELECT 语句时，使用该方法。返回行是在 SqlDataReader 中还是在 OleDbDataReader 中取决于使用的是哪种 .NET 数据提供程序。请注意， DataReader 类是只读、只能向前的数据类，这意味着它只能出于显示或类似目的而用来检索数据。不能更新 DataReader 里的数据！	请参阅程序清单 3A.19 中所显示的如何从 tblUser 表中检索全部用户
ExecuteScalar	当只想返回结果集第一行的第一列数据时使用该方法，该方法将忽略结果集中的其他数据。该方法不但速度快，而且比 ExecuteReader 方法的开销小。因此，当你知道只返回了一行，或正使用一个诸如 COUNT 的合计函数时，就使用该方法	请参阅程序清单 3A.20 中所显示的如何返回 tblUser 表的表行序号

续表

方法名称	什么时候使用	示例
ExecuteXmlReader (仅用于 SqlCommand)	该方法与 ExecuteReader 方法相似, 但返回行必须用 XML 表示	请参阅程序清单 3A.21 中所显示的如何把 tblUser 表中的所有行都以 XML 形式返回

程序清单 3A.18~3A.21 使用的都是 SQL Server .NET 数据提供程序, 如果你想使用其他数据源, 则只需将它换成 OLE DB .NET 数据提供程序即可。切记, 要依据准备使用的数据源来修改 **ConnectionString** 属性。

程序清单 3A.18 使用 ExecuteNonQuery 方法

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim strSQL As String
4
5 ' 连接实例化
6 cnnUserMan = New SqlConnection()
7 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
8 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
9 ' 打开连接
10 cnnUserMan.Open()
11 ' 建立查询字符串
12 strSQL = "INSERT INTO tblUser (LoginName) VALUES('User99')"

```

程序清单 3A.19 使用 ExecuteReader 方法

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim drdTest As SqlDataReader
4 Dim strSQL As String
5
6 ' 连接实例化
7 cnnUserMan = New SqlConnection()
8 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
9 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
10 ' 打开连接
11 cnnUserMan.Open()
12 ' 建立查询字符串
13 strSQL = "SELECT * FROM tblUser"
14 ' 实例化并执行命令
15 cmdUserMan = New SqlCommand(strSQL, cnnUserMan)
16 drdTest = cmdUserMan.ExecuteReader()

```


程序清单 3A.20 使用 ExecuteScalar 方法

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim intNumRows As Integer
4 Dim strSQL As String
5
6 ' 连接实例化
7 cnnUserMan = New SqlConnection()
8 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
9 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
10 ' 打开连接
11 cnnUserMan.Open()
12 ' 建立查询字符串
13 strSQL = "SELECT COUNT(*) FROM tblUser"
14 ' 命令实例化
15 cmdUserMan = New SqlCommand(strSQL, cnnUserMan)
16 ' 保存表的行号
17 intNumRows = CInt(cmdUserMan.ExecuteScalar().ToString)

```

程序清单 3A.21 使用 ExecuteXmlReader 方法

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim drdTest As XmlReader
4 Dim strSQL As String
5
6 ' 连接实例化
7 cnnUserMan = New SqlConnection()
8 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
9 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
10 ' 打开连接
11 cnnUserMan.Open()
12 ' 建立查询字符串来以 XML 形式返回结果集
13 strSQL = "SELECT * FROM tblUser FOR XML AUTO"
14 ' 命令实例化
15 cmdUserMan = New SqlCommand(strSQL, cnnUserMan)
16 ' 以 XML 形式检索行
17 drdTest = cmdUserMan.ExecuteXmlReader()

```

在程序清单 3A.21 中, 因为 FOR XML AUTO 子句出现在查询字符串末端, 所以结果集以 XML 形式检索。该子句命令 SQL Server 2000 以 XML 形式返回结果集。

3A.3.5 命令类异常处理

在处理命令类的属性和异常时可能会出现许多异常。下面讨论了最常见的异常及其处理

方法。

CommandTimeout 属性

当试图将 **CommandTimeout** 属性设为负值时会抛出 **ArgumentException** 异常。程序清单 3A.22 显示了在改动命令超时(command time-out)后, 如何检查是否抛出过 **ArgumentException** 异常。

程序清单 3A.22 检查在设定命令超时是否引起了异常

```
1 ...
2 Try
3 ...' 改变命令超时
4 Catch objException As ArgumentException
5     ' 检查是否试图将命令超时设为一个无效值
6     If objException.TargetSite.Name = "set_CommandTimeout" Then
```

CommandType 属性

如果试图把该属性设为无效命令类型, 就将抛出 **ArgumentException** 异常。程序清单 3A.23 显示了在改动命令类型时, 如何检查是否抛出了 **ArgumentException** 异常。

程序清单 3A.23 检查设定命令类型是否引起了异常

```
1 ...
2 Try
3 ...' 改变命令类型
4 Catch objException As ArgumentException
5     ' 检查是否试图将命令类型设为无效值
6     If objException.TargetSite.Name = "set_CommandType" Then
```

Prepare 方法

如果将连接变量设为 **Nothing** 或者连接没有打开, 就会抛出 **InvalidOperationException** 异常。程序清单 3A.24 显示了如何检查在准备命令时是否抛出了 **InvalidOperationException** 异常。

程序清单 3A.24 尝试准备命令时检查是否引起了异常

```
1 ...
2 Try
3 ...' 准备命令
4 Catch objException As InvalidOperationException
5     ' 检查我们是否试图在一个无效连接上准备命令
6     If objException.TargetSite.Name = "ValidateCommand" Then
```

UpdatedRowSource 属性

如果试图将源行更新属性(row source update property)设为 **UpdateRowSource** 枚举值以

外的值，就会抛出 **ArgumentException** 异常。程序清单 3A.25 显示了如何检查是否抛出了 **ArgumentException** 异常。

程序清单 3A.25 检查在试图改动源行更新时是否发生了异常

```
1 ...
2 Try
3 ... ' 改变源行更新
4 Catch objException As ArgumentException
5     ' 检查是否试图将源行更新值设为一个无效值。
6     If objException.TargetSite.Name = "set_UpdatedRowSource" Then
```

3A.4 使用 DataReader 类

DataReader 类是一个只读、只能向前的数据类。而且在内存使用方面，因为在内存中一次只有一行，所以它很高效。这与 **DataTable** 类（在第 3B 章中讨论）相反，后者为检索到的所有行分配内存。

只能用 **Command** 类的 **ExecuteReader** 方法实例化 **DataReader** 型对象（详细信息请参阅本章前面的“执行指令”部分）。在使用 **DataReader** 时，相关连接不能执行其他操作。这是因为该连接正在为 **DataReader** 服务，而且在连接为其他操作准备好以前，需要关闭 **DataReader**。**DataReader** 类是 ADO.NET 类中与 ADO **Recordset** 类最为类似的一个。如果你把 **Recordset** 与一个只读、只能向前的游标一起考虑的话，它确实如此。（它们的）区别在于如何读取下一行（请参阅本章稍后的“读取 **DataReader** 中的行”部分）。

3A.4.1 SqlDataReader 与 OleDbDataReader

同 **Connection** 和 **Command** 类一样，有两种重要的常规 **DataReader** 类：**SqlDataReader** 用于 MS SQL Server 7 或更高版本，而 **OleDbDataReader** 则用于其他数据源。请注意，如果使用了 **SqlConnection** 对象，就不能再对连接使用 **OleDbCommand** 与/或 **OleDbDataReader** 类。反过来也一样。换言之，不能混合使用 **OleDb** 与 **SQL** 数据类，它们不能一起操作！

3A.4.2 DataReader 对象的声明和实例化

DataReader 类无法继承说明了不能在 **DataReader** 类基础上创建自己的数据类。

下列范例代码是错的：

```
Dim drdTest As New SqlDataReader()
```

由于它试图用 **New** 关键词实例化 **drdTest**，所以会发生错误。程序清单 3A.26 显示了实例化一个 **SqlDataReader** 对象所需的操作。

程序清单 3A.26 **SqlDataReader** 对象的实例化

```
' 声明数据阅读器 (data reader)
```

```
Dim drdTest As SqlDataReader
' 执行命令并返回数据阅读器中的行
drdTest = cmmUserMan.ExecuteReader()
```

虽然在程序清单 3A.26 中没有声明、实例化以及打开命令对象的代码，但重用前面任一程序清单中的代码，你就可以得到功能完整的代码。这里只是为了强调实例化 **DataReader** 对象的方法，下面是一个看起来符合逻辑但无法运行的示例：

```
drdTest = New SqlDataReader ()
```

因为 **DataReader** 类没有构造函数，所以该代码不能运行。

3A.4.3 读取 DataReader 中的行

因为 **DataReader** 类是只能向前的数据类，所以如果需要读取全部返回行的话，就要从头至尾地按顺序读取行。程序清单 3A.27 显示了如何循环遍历一个填充了的数据阅读器。

程序清单 3A.27 循环遍历 DataReader 中的所有行

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmmUserMan As SqlCommand
3 Dim strSQL As String
4 Dim drdTest As SqlDataReader
5 Dim lngCounter As Long = 0
6
7 ' 连接实例化
8 cnnUserMan = New SqlConnection()
9 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
10 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
11 ' 打开连接
12 cnnUserMan.Open()
13 ' 建立查询字符串
14 strSQL = "SELECT * FROM tblUser"
15 ' 命令实例化
16 cmmUserMan = New SqlCommand(strSQL, cnnUserMan)
17 ' 执行命令并在数据阅读器中返回行
18 drdTest = cmmUserMan.ExecuteReader()
19 ' 遍历所有返回行
20 Do While drdTest.Read
21     lngCounter = lngCounter + 1
22 Loop
23
24 ' 显示返回行的数目
25 MsgBox(CStr(lngCounter))
```

程序清单 3A.27 循环遍历数据阅读器中的行，并计算了行的数目。很明显，这就是一个

显示如何顺序遍历 **Command** 类 **ExecuteReader** 方法返回的所有行的示例。

3A.4.4 关闭 DataReader

由于 **DataReader** 对象在处于打开状态时会占用连接, 所以一旦用完就应该马上将其关闭。可以使用 **Close** 方法关闭 **DataReader**, 如下所示:

```
drdTest.Close()
```

可以用 **IsClosed** 属性来检查数据阅读器是否已关闭, 如下所示:

```
If Not drdTest.IsClosed Then
```

3A.4.5 检查列中的 Null 值

在实例化数据阅读器并使用 **Read** 方法将其放在有效行上后, 就可以检查某个特定的列里是否包含 **Null** 值了。通过使用 **IsDBNull** 方法可以实现这一功能, 该方法把当前行中特定列的内容与 **DBNull** 类相比较。如果该列未包含已知值, 也称为 **Null** 值, **IsDBNull** 方法就返回 **True**, 否则返回 **False**。下面显示了如何实现该检查:

```
‘ 检查第 3 列 (0 数组) 是否包含 Null 值
```

```
If drdTest.IsDBNull(2) Then
```

3A.4.6 多结果处理

DataReader 类可以处理由命令类返回的多个结果。如果命令类通过指定由分号分隔的一批 SQL 语句来返回多个结果, 就可以用 **NextResult** 方法前移到下一个结果。请注意, 这只能向前移动。若能有一个 **PreviousResult** 方法就更好了, 可惜这是不可能的。

3A.4.7 DataReader 属性

DataReader 类具有以下属性, 如表 3A.21 所示 (以字母升序排列)。

表 3A.21 DataReader 类的属性

名称	说明
Depth	该只读属性用于分级的结果集, 例如 XML 数据。返回值指明了现在处于多少层节点, 或者当前元素在 XML 元素堆栈中的深度。SQL Server .NET 数据提供程序不支持该属性
FieldCount	该只读属性返回当前行中列的个数。如果 DataReader 没有连接数据源, 则抛出 NotSupportedException 异常
IsClosed	该属性只读, 用来指明数据阅读器是否已关闭。如果数据阅读器已关闭, 就返回 True , 反之返回 False

续表

名称	说明
Item	该只读属性用于指明要从什么列或什么域返回值。该属性不能被忽略，可用指明列名称的字符串指定，或者用指明列序号的数字指定。和所有数组一样，列也是从 0 开始编号，因此可用 <code>drdTest.Item(0)</code> 读取第一列。实际上，同样可以使用列的名称，因此如果第一列叫作 <code>Id</code> ，那么可以如下设置该属性： <code>drdTest.Item("Id")</code> 。请注意，没有对返回值应用任何格式化，这意味着无论列中包含什么内容，都将以它本来的格式返回
RecordsAffected	该只读属性检索受命令影响（插入、更新或删除）的记录序号。直到读完所有行并关闭了数据阅读器后才会设置该属性

3A.4.8 DataReader 方法

表 3A.22 以字母排列出了 `DataReader` 类的非派生方法和全局方法。请注意，所有标有星号(*)的方法都要求特定列中的数据是该方法所指明的数据类型。这是因为对于该特定列的内容不会做任何转换。如果试图在包含不同数据类型内容的列上使用某种方法，就会抛出 `InvalidCastException` 异常。而且，在继承类中，这些方法也不能被忽略。

表 3A.22 DataReader 类的方法

名称	说明	示例
<code>Close()</code>	关闭数据阅读器。必须在将连接用于其他用途之前调用该方法。这是因为只要数据阅读器开着，就会占用连接	<code>drdTest.Close()</code>
<code>*GetBoolean(ByVal vintOrdinal As Integer)</code>	返回特定列 <code>vintOrdinal</code> 的 Boolean 值。对应的 SQL Server 数据类型是 bit 。	<code>blnTest = drdTest.GetBoolean(0)</code>
<code>*GetByte(ByVal vintOrdinal As Integer)</code>	将特定列 <code>vintOrdinal</code> 的值以 Byte 型返回。对应的 SQL Server 数据类型是 tinyint 。	<code>bytTest = drdTest.GetByte(0)</code>
<code>*GetBytes(ByVal vintOrdinal As Integer, ByVal vintDataIndex As Integer, ByVal varrbytBuffer() As Byte, ByVal vintBufferIndex As Integer, ByVal vintLength As Integer)</code>	以一个 Byte 数组在 <code>varrbytBuffer</code> 参数中返回特定列 <code>vintOrdinal</code> 的值。特定列长度被当作该方法调用的结果返回。 <code>VintLength</code> 指定了字节数组复制的最大字节数， <code>VintBufferIndex</code> 是缓存的起始点。当字节被复制入缓存中时，从 <code>vintBufferIndex</code> 开始放置它们。从数据源的 <code>vintDataIndex</code> 点开始复制字节。当需要从数据库中读取大的图像时，可以使用该方法。该方法很大程度上与 ADO 中的 GetChunk 方法相对应。对应的 SQL Server 数据类型为 binary 、 image 和 varbinary	<pre>intNumBytes = drdSql.GetBytes (1, 0, Nothing, 0, Integer.MaxValue ReDim arrbytTest(intNum Bytes) drdSql.GetBytes(1 , 0, arrbytTest, 0, intNumBytes)</pre>

续表

名称	说明	示例
*GetChar(ByVal vintOrdinal As Integer)	以 Char 型返回特定列 (vintOrdinal) 的值。对应的 SQL Server 数据类型为 char	chrTest = drdTest.GetChar(0)
*GetChars(ByVal vintOrdinal As Integer, ByVal vintDataIndex As Integer, ByVal varchrBuffer() As Char, ByVal vintBufferIndex As Integer, ByVal vintLength As Integer)	以 Char 数组在 varchrBuffer 参数中返回特定列 (vintOrdinal) 的值。特定列长度被当作该方法调用的结果返回。VintLength 指定了字符数组复制的最大字符数。VintBufferIndex 是缓存的起始点。当字符被复制入缓存中时, 从 vintBufferIndex 开始放置它们。从数据源的 vintDataIndex 点开始复制字符。对应的 SQL Server 数据类型为 char 、 nchar 、 ntext 、 nvarchar 、 text 和 varchar	intNumChars = drdSql.GetChars(1, 0, Nothing, 0, Integer.MaxValue) ReDim arrchrTest (intNumChars) drdSql.GetChars(1 , 0, arrchrTest, 0, intNumChars)
GetData(ByVal vintOrdinal As Integer)	目前还不支持	
GetDataTypeName(ByVal vintOrdinal As Integer)	返回数据源中特定列所使用的数据类型名称	strDataType = drdTest.GetDataType Name(0)
*GetDateTime(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 DateTime 值。对应的 SQL Server 数据类型为 datetime 和 smalldatetime	dtmTest = drdTest.GetDateTi me(0)
*GetDecimal(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 Decimal 值。对应的 SQL Server 数据类型为 decimal 、 money 、 numeric 和 smallmoney	decTest = drdTest.GetDecimal (0)
*GetDouble(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 Double 值。对应的 SQL Server 数据类型为 float	dblTest = drdTest.GetDouble(0)
GetFieldType(ByVal vintOrdinal As Integer)	返回 vintOrdinal 所指定列的数据类型。返回一个 Type 型对象	typTest = drdTest.GetFieldT ype(0)
*GetFloat(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的浮点数。返回对象是 Single 型。对应的 SQL Server 数据类型为 real	sqlTest = drdTest.GetFloat (0)
*GetGuid(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的全局唯一标识符。返回对象为 Guid 型。对应的 SQL Server 数据类型为 uniqueidentifier	guiTest = drdTest.GetGuid (0)
*GetInt16(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 16 位有符号整数。返回一个 Short 型对象。对应的 SQL Server 数据类型为 smallint	shrTest = drdTest.GetInt16 (0)
*GetInt32(ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 32 位有符号整数。返回对象为 Integer 型。对应的 SQL Server 数据类型为 int	intTest = drdTest.GetInt32 (0)

续表

名称	说明	示例
*GetInt64 (ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 64 位有符号整数。返回对象为 Long 型。对应的 SQL Server 数据类型为 bigint	lngTest = drdTest.GetInt64 (0)
GetName (ByVal vintOrdinal As Integer)	返回 vintOrdinal 指定列的名称。返回对象为 String 型。该方法与 GetOrdinal 方法正相反	strTest = drdTest. GetName(0)
GetOrdinal (ByVal vstrName As String)	返回 vstrName 指定列的序号。返回对象为 Integer 型。该方法与 GetName 正相反	intTest = drdTest. GetOrdinal(0)
*GetSqlBinary (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的变长二进制数。返回对象为 SqlBinary 型。对应的 SQL Server 数据类型为 binary 和 image	sbnTest = drdTest.GetSqlBin ary(0)
*GetSqlBoolean (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的值。返回对象为 SqlBoolean 型, 并且只能是 True 、 False 或 Nothing 三者之一。对应的 SQL Server 数据类型为 bit 。尽管该方法和 GetBoolean 方法处理的 SQL Server 数据类型相同, 但返回值是不同的	sbtTest = drdTest.GetSqlBit (0)
*GetSqlByte (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 8 位无符号整数。返回对象为 SqlByte 型。对应的 SQL Server 数据类型为 tinyint	sbyTest = drdTest.GetSqlByte (0)
*GetSqlDateTime (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的日期和时间值。返回对象为 SqlDateTime 型。对应的 SQL Server 数据类型为 datetime 和 smalldatetime	sdtTest = drdTest.GetSqlDate Time(0)
*GetSqlDecimal (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 SqlDecimal 值	sdbTest = drdTest.GetSqlDou ble(0)
*GetSqlDouble (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 SqlDouble 值。对应的 SQL Server 数据类型为 float	sdbTest = drdTest.GetSqlDou ble(0)
*GetSqlGuid (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的全局唯一标识符。返回对象为 SqlGuid 型。对应的 SQL Server 数据类型为 uniqueidentifier	sguTest = drdTest.GetSqlGuid (0)
*GetSqlInt16 (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 16 位有符号整数。返回对象为 SqlInt16 型。对应的 SQL Server 数据类型为 smallint	s16Test = drdTest.GetSqlInt 16(0)

续表

名称	说明	示例
*GetSqlInt32 (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 32 位有符号整数。返回对象为 SqlInt32 型。对应的 SQL Server 数据类型为 int	s32Test = drdTest.GetSqlInt32(0)
*GetSqlInt64 (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 64 位有符号整数。返回对象为 SqlInt64 型。对应的 SQL Server 数据类型为 bigint	s64Test = drdTest.GetSqlInt64(0)
*GetSqlMoney (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 SqlMoney 值。对应的 SQL Server 数据类型为 money 和 smallmoney	smonTest = drdTest.GetSqlMoney(0)
*GetSqlSingle (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 SqlSingle 值。对应的 SQL Server 数据类型为 real	ssgTest = drdTest.GetSqlSingle(0)
*GetSqlString (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回在 vintOrdinal 所指定列中存储的 SqlString 值。对应的 SQL Server 数据类型为 char 、 nchar 、 ntext 、 nvarchar 、 text 和 varchar	sstTest = drdTest.GetSqlString(0)
GetSqlValue (ByVal vintOrdinal As Integer) (仅用于 SqlDataReader)	返回一个用于表示在 vintOrdinal 所指定列中存储的基础数据类型及其值的对象。返回对象是 Object 型。可以对该对象使用 GetType 、 ToString 及其他方法, 以获得所需信息	objTest = drdTest.GetSqlValue(0)
GetSqlValues (ByVal varrobjValues() As Object) (仅用于 SqlDataReader)	返回 varrobjValues 数组中当前列的所有属性 (attribute) 域。必须在执行该方法前初始化数组。返回的 Integer 是数组中对象的个数	Dim arrobjValues() As Object ReDim arrobjValues(Integer.MaxValue) intNumFields = drdTest.GetSqlValues(arrobjValues) ReDim Preserve arrobjValues(intNumFields)
*GetString (ByVal vintOrdinal As Integer)	返回在 vintOrdinal 所指定列中存储的 String 值。对应的 SQL Server 数据类型为 char 、 nchar 、 ntext 、 nvarchar 、 text 和 varchar	strTest = drdTest.GetString(0)
*GetTimeSpan (ByVal vintOrdinal As Integer) (仅用于 OleDbDataReader)	返回在 vintOrdinal 所指定列中存储的 TimeSpan 值	tmsTest = drdTest.GetTimeSpan(0)
GetValue (ByVal vintOrdinal As Integer)	以一个 Object 型变量返回在 vintOrdinal 所指定列中存储的值。返回值是 .NET 自己的格式	objTest = drdTest.GetValue(0)

续表

名称	说明	示例
GetValues(ByVal varobjValues () As Object)	返回 varobjValues 数组中当前行所有属性列的值。返回的 Integer 是数组中对象的个数。可以按照如下说使用 FieldCount 属性来测定数组大小： Dim arrobjValues() As Object ReDim arrobjValues(drdTest.FieldCount) drdTest.GetValues(arrobjValues)	
IsDBNull(ByVal vintOrdinal As Integer)	如果 vintOrdinal 所指定列等于 DBNull ，那么该方法返回 True ；否则返回 False 。DBNull 意味着列中不包含已知值，或者为 Null 值	If drdTest.IsDBNull(0) Then
NextResult()	该方法将数据阅读器放在命令所返回的结果集的下一个结果上。如果查询命令并没有包含一批行返回的 SQL 语句，或者当前结果是结果集的最后一个，那么该方法就不会奏效。如果还有结果，就返回 True ，否则返回 False 。在实例化时，将数据阅读器默认地放在第一个结果上	If drdTest.NextResult() Then
Read()	该方法将数据阅读器放在下一行上。请注意，实例化数据阅读器后，在可以访问数据阅读器中任意数据之前必须调用该方法。这是因为数据阅读器与 ADO 中的 Recordset 对象不同，不是默认地放在第一行上。如果数据阅读器放在了下一行上，就返回 True ，否则返回 False	If drdTest.Read() Then

所有 GetSql... 方法（如 GetSqlBinary）都只与 SqlDataReader 类相关，且隶属于 System.Data.SqlTypes 名称空间。其中的多数方法都有相应的、可以处理全部数据阅读器类的方法。它们处理数据源（SQL Server）中相同的数据类型，但返回的数据类型不同。因此，究竟选择哪种方法取决于应用程序要处理的数据类型：是需要 SQL Server 自身的数据类型，还是 .NET Framework 的数据类型？

3A.4.9 处理 DataReader 异常

DataReader 类的属性和方法可能会抛出表 3A.23 所示的异常。

表 3A.23 按字母升序排列的 DataReader 异常

名称	说明	抛出异常的属性 / 方法
ArgumentException	参数超出范围时抛出该异常	
InvalidCastException	如果试图使用 Get* 方法隐式计算或转换数据库的值，就会抛出该异常	所有被标星号 (*) 的方法。
InvalidOperationException	当试图执行在当前状态对象中无效的操作（属性或方法）时，就会抛出该异常	
NotSupportedException	试图在一个已关闭或者未连接的 DataReader 对象上使用 I/O 属性或方法时，就回抛出该异常	

3A.4.10 什么时候使用 DataReader 类

DataReader 类应该用于需要在内存消耗上开销最小的情形。很明显, 如果想返回几十万行数据, 就不会使用 **DataReader** 对象, 因为可能始终无法循环遍历所有行。

如果是仅看一行或几行返回数据, 则只需要读取一、两个字段或列, 那么 **DataReader** 无疑是一个好的选择。**DataReader** 类是只读的, 因此, 如果需要更新数据的话, 就不应该使用 **DataReader** 类。

3A.4.11 XmlReader

其实在说只有两个 **DataReader** 类的时候, 我撒了谎, 因为还有 **XmlReader** 类。很明显, 该类用于处理 XML 格式的数据。实际上, **XmlReader** 类是 `System.Xml` 名称空间的一部分, 必须被忽略。在 .NET Framework 中已经做到了这点。虽然仍可以选择由自己忽略 **XmlReader** 类, 但已有三个类整体实现了 **XmlReader** 类, 它们是 **XmlTextReader**、**XmlValidatingReader** 和 **XmlNodeReader**。

- **XmlTextReader**: 该类用于快速、无缓存、只向前的流访问。该类检查 XML 的格式完好性, 但如果需要数据验证, 就必须使用 **XmlValidatingReader** 类。
- **XmlValidatingReader**: 该类提供 XML 数据的 XDR、XSD 和 DTD 模式验证 (schema validation)。
- **XmlNodeReader**: **XmlNodeReader** 类读取 **XmlNavigator** 类中的 XML 数据, 这意味着它可以读取存储在 **XmlDocument** 或 **XmlDataDocument** 中的数据。

对于这三个派生的 XML 阅读器, 本节不再探究更多细节, 但仍将仔细考察其父类 **XmlReader**。

3A.4.12 XmlReader 的属性

XmlReader 类具有表 3A.24 所示的属性 (以字母顺序排列):

表 3A.24 **XmlReader** 类的属性

名称	说明
AttributeCount	返回当前节点的属性个数, 包括默认属性。仅 Element 、 DocumentType 和 XmlDeclaration 类节点有属性, 这意味着该属性只与这三类节点有关。如果已使用了命令类的 ExecuteXmlReader 方法, 那么 AttributeCount 属性将返回当前行非 Null 值的个数。该属性只读
BaseURI	返回当前节点的基础 URI。如果没有基础 URI, 就返回空字符串。该属性只读, 并且如果由命令类的 ExecuteXmlReader 方法返回 XmlReader , 那么该属性就没有任何实在意义
CanResolveEntity	该属性只读, 返回一个说明阅读器是否可以解析和分解实体的值
Depth	返回值指明当前处于多少层节点或在 XML 元素堆栈中的深度。该属性只读, 并且必须被忽略

续表

名称	说明
EOF	该属性只读, 返回一个 Boolean 值, 指明 XmlReader 是否位于流的末尾
HasAttributes	该属性只读, 返回一个 Boolean 值, 指明当前节点是否有属性
HasValue	该只读属性返回一个 Boolean 值, 指明当前节点是否可以保存 Value 型属性值
IsDefault	该属性只读, 返回一个 Boolean 值, 指明当前节点是不是由默认值生成的属性, 该属性在模式或 DTD 中进行定义
IsEmptyElement	该属性只读, 返回一个 Boolean 值, 指明当前节点是否一个空元素。可以如下定义空元素: <code><DataElement/></code>
Item()	该属性只读, 返回属性的值, 属性由 index 指明, 如 (Index As Integer); 或由 name 指明, 如 (Name As String); 或由 LocalName 与 NamespaceURI 属性指明, 如 (LocalName As String, NamespaceURI As String)。有关详细信息, 请参阅 LocalName 和 NamespaceURI 属性项。多数情况下可忽略 Item() 属性, 只简单地使用圆括号即可
LocalName	该属性只读, 返回当前节点的名称, 没有名称空间前缀。对于没有名称的节点类型, 返回空字符串。如果使用了命令类的 ExecuteXmlReader 方法, 那么 LocalName 属性将返回表的名称
Name	该属性只读, 返回当前节点的限定名, 或者返回带名称空间前缀的名称
NamespaceURI	该属性只读, 返回当前节点的名称空间 URI
NameTable	该属性只读, 返回与实现相关的 XmlNameTable
NodeType	该只读属性返回当前节点的类型。返回数据类型为 XmlNodeType
Prefix	该属性返回当前节点的名称空间前缀, 该属性只读
QuoteChar	该属性返回属性节点中用于封闭值的引号。该属性只读, 并且返回一个单引号 (') 或者双引号 (")。该属性只应用于属性节点
ReadState	该属性返回流的读取状态。该属性只读并返回 ReadState 枚举值中的一项。可以返回下列项: Closed : XmlReader 上已执行了 Close 方法 EndOfFile : 已经到达了数据流末尾 (成功的) Error : 发生了错误。这意味着 XmlReader 不能继续读取流了 Initial : 还没有调用 Read 方法, 因此, 阅读器在它的初始位置上 Interactive : 正在执行一个读取操作
Value	该只读属性返回当前节点的文本值。如果当前节点没有值可返回, 就返回空字符串
XmlLang	该只读属性返回正在使用的 xml:lang 作用域。请注意该属性值也是 NameTable 属性所返回 XmlNameTable 中的一部分
XmlSpace	该只读属性以 XmlSpace 枚举值返回正在使用的 xml:space 作用域。 XmlSpace 枚举值有如下成员: Default : xml:space 作用域是“默认值”。 None : 没有 xml:space 作用域。 Preserve : xml:space 作用域是“保护的”

3A.4.13 XmlReader 的方法

表 3A.25 按字母排列出了 **XmlReader** 类的非继承方法和公共方法。请注意, 所有加星号 (*) 的方法都要求特定列数据与该方法指明的数据类型一致, 这是因为不会对特定列做任何转换。如果试图将任何方法用于包含不同数据类型内容的列上, 就会抛出 **InvalidCastException** 异常。而且在继承类中, 这些方法都不能被忽略。

表 3A.25 XmlReader 类的方法

名称	说明	示例
Close()	重置所有属性, 并将 ReadState 属性改为 Closed	<code>xrdTest.Close()</code>
GetAttribute()	该可重载方法返回一个属性的值。请参阅示例代码, 返回 tblUser 表中当前节点或当前行的 LoginName 的代码。在别处已经用命令类的 ExecuteXmlReader 方法实例化了阅读器	<code>xrdTest.GetAttribute(3), xrdTest.GetAttribute ("LoginName")</code>
IsStartElement()	该可重载方法返回一个 Boolean 值, 指明当前目录节点是一个开始标签还是一个空元素标签	<code>If xrdTest.IsStart Element() Then</code>
LookupNamespace (ByVal vstrPrefix As String)	在 XmlReader 类的派生实现里必须忽略该方法。它在当前元素范围内解析 vstrPrefix	<code>xrdTest.LookupNamespace ("MyNameSpace")</code>
MoveToAttribute()	该可重载方法移动到与传递参数匹配的属性。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 则该方法便可用来在当前行的列之间移动	<code>xrdTest.MoveToAttribute (0), xrdTest.MoveToAttribute ("LoginName")</code>
MoveToContent()	该方法检查当前节点是否是一个内容节点, 并且跳到下一个内容节点, 如果找不到内容节点则跳到 EOF。如果当前节点是一个属性节点, 那么就将阅读器的位置移回到拥有该属性节点的元素	<code>xrdTest.MoveToContent()</code>
MoveToElement()	该方法必须被覆盖, 移动到当前属性节点所在的元素。基于当前属性是否为一个元素所有, 会返回一个 Boolean 值。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法可与 MoveToAttribute 方法联合作用, 以移动到一个特定属性/列中, 并且退回到拥有该属性/列的元素/行。最后返回一个 Boolean 值, 指明该移动是否成功	<code>xrdTest.MoveToElement()</code>
MoveToFirstAttribute()	该方法用于移动到元素的第一个属性中。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法可用于移动到当前行的第一列中。最后返回一个 Boolean 值, 指明该移动是否成功	<code>xrdTest.MoveToFirstAttr ibute()</code>

续表

名称	说明	示例
MoveToNextAttribute()	该方法用于移动到元素中下一个属性。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法可用于移动到当前行的下一列	<code>xrdTest.MoveToNextAttribute()</code>
Read()	该方法用于读取流中的下一个节点。必须在调用其他属性或方法之前调用该方法。实际上这并不十分正确, 因为如果阅读器还没有被放到第一行或者第一个元素中, 那么就可以从一些属性中返回初始值	<code>xrdTest.Read()</code>
ReadAttributeValue()	该方法把属性节点分解为 Text 和 EntityReference 型节点。返回一个 Boolean 值, 指明是否返回节点。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法就没有实际意义	<code>xrdTest.ReadAttributeValue()</code>
ReadElementString()	该可重载方法用于读取简单的文本元素	<code>xrdTest.ReadElementString()</code> or <code>xrdTest.ReadElementString("ElementName")</code>
ReadEndElement()	该方法用于检查当前节点是否为结尾标签。如果当前节点是结尾标签, 阅读器的位置就前进到下一个节点。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法就没有什么实际意义了	<code>xrdTest.ReadEndElement()</code>
ReadInnerXml()	ReadInnerXml 方法以字符串形式返回当前节点的所有内容。返回的字符串中包括置标文本 (markup text)。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法就没有什么实际意义	<code>xrdTest.ReadInnerXml()</code>
ReadOuterXml()	ReadOuterXml 方法以字符串形式返回当前节点及其所有子节点的全部内容。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法将以 XML 节点的形式返回当前行。请注意, 只有非空列才会作为属性返回	<code>xrdTest.ReadOuterXml()</code>
ReadStartElement()	该方法用于检查当前节点是否为元素。如果当前节点是元素, 阅读器的位置就前进到下一个节点。如果 XmlReader 已经被命令类的 ExecuteXmlReader 方法返回, 那么该方法就没有什么实际意义	<code>xrdTest.ReadStartElement()</code>

续表

名称	说明	示例
ReadString()	该方法必须被覆盖，用于读取元素内容并以字符串形式将其返回	xrdTest.ReadString()
ResolveEntity()	该方法用在 EntityReferences 节点上，用于解析该类节点的实体引用	xrdTest.ResolveEntity()
Skip()	跳过当前元素或移动到下一个元素。根据阅读器所在位置，Skip 方法可以实现 Read 方法的功能。当阅读器位于一个叶节点上时，就能做到这点。请注意，该方法可以检查 XML 格式的完好性	xrdTest.Skip()

3A.4.14 声明与实例化 XmlReader 对象

XmlReader 类必须被覆盖，这意味着不能创建 **XmlReader** 类的实例。

下面的示例是错的：

```
Dim xrdTest As New XmlReader()
```

它是错的，因为它试图用关键词 **New** 来实例化 xrdTest。在程序清单 3A.28 中显示了如何正确地进行实例化。

程序清单 3A.28 实例化 XmlReader 对象

```
' 声明 xml 阅读器
Dim xrdTest As XmlReader
' 执行命令并在 xml 阅读器中返回行
xrdTest = cnnUserMan.ExecuteXmlReader()
```

程序清单 3A.28 中没有声明、实例化以及打开命令对象的代码，但如果要使用前面任一程序清单中的代码，则必须有功能完全的代码。

3A.4.15 读取 XmlReader 中的行

因为 **XmlReader** 是一个只能向前的数据类，所以如果需要读取所有返回行的话，就需要从头到尾按顺序读取行。程序清单 3A.29 显示了如何循环遍历一个填充了的 XML 阅读器。

程序清单 3A.29 循环遍历 XmlReader 中的所有

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUserMan As SqlCommand
3 Dim strSQL As String
4 Dim xrdTest As XmlReader
5 Dim lngCounter As Long = 0
6
7 ' 连接实例化
```

```

8 cnnUserMan = New SqlConnection()
9 cnnUserMan.ConnectionString = "User ID=UserMan;" & _
10 "Server=USERMANPC;Password=userman;Initial Catalog=UserMan"
11 ' 打开连接
12 cnnUserMan.Open()
13 ' 建立查询字符串
14 strSQL = "SELECT * FROM tblUser"
15 ' 命令实例化
16 cmdUserMan = New SqlCommand(strSQL, cnnUserMan)
17 ' 执行命令并在数据阅读器中返回行
18 xrdTest = cmdUserMan.ExecuteReader()
19 ' 循环遍历所有返回行
20 Do While xrdTest.Read
21     lngCounter = lngCounter + 1
22 Loop
23
24 ' 显示返回行的个数
25 MsgBox(CStr(lngCounter))

```

程序清单 3A.29 中的代码循环遍历数据阅读器中的行并计算了行的个数，是一个演示了如何顺序地遍历 **Command** 类 **ExecuteXmlReader** 方法所返回的全部行的示例。

3A.4.16 关闭 XmlReader

因为当 **XmlReader** 对象处于打开状态时会占用连接，所以一旦用完 XML 阅读器就应该马上关闭它。可以如下使用 **Close** 方法关闭 XML 阅读器：

```
xrdTest.Close()
```

3A.5 DataAdapter 说明

DataAdapter 类用于从数据源读取数据并填充 **DataSet** 以及相关类，例如 **DataTable** 有关这些非连接数据类的详细信息，请参阅第 3B 章。**DataAdapter** 类也负责将所做改变移值给数据源。换句话说，数据适配器（data adapter）是位于 ADO.NET 中非连接部分与连接部分之间的“连接器”类。实际上，可以十分恰当地说，**DataAdapter** 类是对数据源中的数据进行操作的工具箱。

数据适配器用 **Connection** 对象连接数据源，然后再用 **Command** 对象从数据源中检索数据并将数据送还给数据源。除了 **DataReader** 类以外，ADO.NET 中的所有数据访问都要经过数据适配器。事实上，这一点还可以有不同的表达：所有非连接的数据访问都要通过数据适配器进行。数据适配器是数据源与数据集之间的桥梁！

当需要从数据源检索数据或者向数据源发送数据时，数据适配器使用 **Command** 对象完成，而你则必须指定这些命令对象。现在，这一切与 ADO 中的操作大不相同，在 ADO 中，你几乎不能控制查询的执行方式。由你指定用于在数据源中选择、更新和删除行的命令对象。通过创建命令对象并给它们分配合适的数据适配器属性来实现这一点。数据适配器属性有：

SelectCommand、**InsertCommand**、**UpdateCommand** 或 **DeleteCommand**。

共有两种 **DataAdapter** 类：**OleDbDataAdapter** 类与 **SqlDataAdapter** 类。

3A.5.1 DataAdapter 属性

DataAdapter 类的属性如表 3A.26 所示（按字母排序）。请注意，该表只显示了公共方法和非继承方法。事实上，这不完全对，因为只忽略了 **Object** 这样继承自基础类的属性。这意味着显示的是继承自 **DataAdapter** 类的属性。所列属性对于 **SqlDataAdapter** 和 **OleDbDataAdapter** 都是相同的。

表 3A.26 DataAdapter 类的属性

名称	说明
AcceptChangesDuringFill	这个可读/写属性返回或设置 Boolean 值，该值指明在 DataRow 类加入 DataTable 中以后，是否对它使用了 AcceptChanges 方法。其默认值为 True
DeleteCommand	这个可读/写属性返回或设置从数据源中删除行时要使用的 SQL 语句
InsertCommand	这个可读/写属性返回或设置向数据源插入行时要使用的 SQL 语句
MissingMappingAction	这个可读/写属性返回或设置一个值，该值指明未映射的原始表或原始列是否应使用原始名称解析，或是抛出一个异常。该属性决定了在原始表或原始列没有映射地址时所采取的操作。其有效值是 MissingMappingAction 枚举值的所有成员，默认值为 Passthrough
MissingSchemaAction	这个可读/写属性返回或设置一个值，该值指明在数据集中丢失的原始表、原始列和关系，是加入数据集模式中忽略，还是抛出异常。这意味着该属性决定了当数据源中的表、列和/或关系在数据集中丢失时，应采取什么行动。有效值为 MissingSchemaAction 枚举值的所有成员。默认值为 Add
SelectCommand	这个可读/写属性返回或设置从数据源选择/取出行时要使用的 SQL 语句
TableMappings	该只读属性返回一个值，该值指明数据源中的表应该怎样映射到数据集中的表。其返回值是 DataTableMapping 对象的集合，默认值为空集合
UpdateCommand	这个可读/写属性返回或设置在数据源中更新行时要使用的 SQL 语句

3A.5.2 DataAdapter 的方法

表 3A.27 以字母升序列出了 **DataAdapter** 类的非继承方法和公共方法。除特别声明以外，列出的方法对于 **SqlDataAdapter** 和 **OleDbDataAdapter** 类来说都是相同的。

表 3A.27 DataAdapter 类的方法

名称	说明	示例
Fill() (仅用于 SqlDataAdapter)	Fill 方法是可重载的, 继承自 DbDataAdapter 类。该方法用于在数据集中增加和/或更新行。这意味着在数据集中会复制数据源中被请求的数据	相关例子请参阅第 3B 章“使用 DataAdapter 填充 Data Set ”部分
Fill() (仅用于 OleDbDataAdapter)	Fill 方法是可重载的, 而且该方法的一些可重载形式继承自 DbDataAdapter 类。该方法用于在数据集中添加和/或更新行, 数据集是基于 ADO Recordset 或 ADO Record 对象中的行, 或者基于来自数据源的行	相关示例请参阅第 3B 章“使用 DataAdapter 填充 Data Set ”部分
FillSchema()	FillSchema 方法是可重载的, 继承自 DbDataAdapter 类。该方法用于向数据集中添加 DataTable 对象。根据传递参数配置该 DataTable 对象	相关示例请参阅第 3B 章“使用 DataTable 类”部分
GetFillParameters()	GetFillParameters 方法继承自 DbDataAdapter 类。该方法读取执行 SELECT 语句时使用的全部参数, 应将返回值存储在 IDataParameter 型数组中	<code>arrprmSelect = dadUser.GetFillParameters()</code>
Update()	Update 方法是可重载的, 继承自 DbDataAdapter 类。该方法根据数据集中的变化用来更新数据源。这意味着与 InsertCommand 、 UpdateCommand 和 DeleteCommand 属性相关的命令对象在每次插入、修改和删除行时都会被执行	相关示例请参阅第 3B 章“使用 DataAdapter 更新 Data Source ”部分

3A.5.3 实例化 DataAdapter

和其他对象一样, 在使用 **DataAdapter** 之前必须将其实例化。使用可重载的构造函数 **New** 可以实例化 **DataAdapter** 类。程序清单 3A.30 和程序清单 3A.31 展示了部分示例。

程序清单 3A.30 实例化 SqlDataAdapter

```

1 Const strSQLUser As String = "SELECT * FROM tblUser"
2 Const strConnection As String = "Data Source=USERMANPC;" & _
3   "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5 Dim cnnUserMan As SqlConnection
6 Dim cmdUser As SqlCommand
7 Dim dadDefaultConstructor As SqlDataAdapter
8 Dim dadSqlCommandArgument As SqlDataAdapter
9 Dim dadSqlConnectionArgument As SqlDataAdapter
10 Dim dadStringArguments As SqlDataAdapter
11
12 ' 实例化并打开连接
13 cnnUserMan = New SqlConnection(strConnection)
14 cnnUserMan.Open()
```

```

15 ' 命令实例化
16 cmmUser = New SqlCommand(strSQLUser)
17
18 ' 数据适配器的实例化
19 dadDefaultConstructor = New SqlDataAdapter()
20 dadSqlCommandArgument = New SqlDataAdapter(cmmUser)
21 dadSqlConnectionArgument = New SqlDataAdapter(strSQLUser, cnnUserMan)
22 dadStringArguments = New SqlDataAdapter(strSQLUser, strConnection)
23 ' 数据适配器的实例化
24 dadDefaultConstructor.SelectCommand = cmmUser
25 dadDefaultConstructor.SelectCommand.Connection = cnnUserMan

```

程序清单 3A.31 实例化 OleDbDataAdapter

```

1 Const strSQLUser As String = "SELECT * FROM tblUser"
2 Const strConnection As String = "Provider=SQLOLEDB;Data Source=USERMANPC" & _
3   ";User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5 Dim cnnUserMan As OleDbConnection
6 Dim cmmUser As OleDbCommand
7 Dim dadDefaultConstructor As OleDbDataAdapter
8 Dim dadOleDbCommandArgument As OleDbDataAdapter
9 Dim dadOleDbConnectionArgument As OleDbDataAdapter
10 Dim dadStringArguments As OleDbDataAdapter
11
12 ' 实例化并打开连接
13 cnnUserMan = New OleDbConnection(strConnection)
14 cnnUserMan.Open()
15 ' 命令的实例化
16 cmmUser = New OleDbCommand(strSQLUser)
17
18 ' 数据适配器的实例化
19 dadDefaultConstructor = New OleDbDataAdapter()
20 dadOleDbCommandArgument = New OleDbDataAdapter(cmmUser)
21 dadOleDbConnectionArgument = New OleDbDataAdapter(strSQLUser, cnnUserMan)
22 dadStringArguments = New OleDbDataAdapter(strSQLUser, strConnection)
23 ' 数据适配器的实例化
24 dadDefaultConstructor.SelectCommand = cmmUser
25 dadDefaultConstructor.SelectCommand.Connection = cnnUserMan

```

如果对数据适配器使用默认构造函数（第 7 行、第 19 行、第 24 行和第 25 行），就不得不为要执行的动作指定命令对象。这意味着，如果要从数据源中检索行，则至少需要设置 **SelectCommand** 属性（有关如何设置命令属性的详细信息，请参阅下一节“设定命令属性”）。在第 24 行中，命令属性设成了一个有效命令对象。然而，这并非必要的，因为实际上还可以指定一个要选择的字符串（STR-SQL-VSER）。

对于从数据源中检索行，其他三个构造函数不需要进一步实例化。然而，如果想插入、删除与更新数据源，则必须指定对应的命令属性（有关如何指定命令属性的详细信息，请参

阅下一节“设定命令属性”。

最终使用哪种方法取决于你自己，但如果所有的命令对象都使用同一个连接，那么为什么不在实例化数据适配器时指定它（命令对象）呢？

正如在程序清单 3A.30 和程序清单 3A.31 中看到的一样，实例化 **OleDbDataAdapter** 和 **SqlDataAdapter** 对象之间没有什么本质的区别。这两个类之间的区别在填充数据集后才会显示出来。有关如何填充数据集的详细信息，请参阅第 3B 章“使用 **DataAdapter** 填充 **Data Set**”部分。

3A.5.4 设定命令属性

如果只是用数据适配器从数据源中检索数据，那么就不需要设定数据适配器的命令属性，前提是已经用数据适配器析构函数指定了 **SelectCommand** 属性（请参阅前面“实例化 **DataAdapter**”部分）。然而，如果需要从数据源中插入、更新和/或删除行，以及/或者在实例化数据适配器时没有设定 **SelectCommand** 属性，那么将就必须在使用数据适配器前设定这些属性。**SelectCommand** 用于从数据源中检索行，而其他三个命令属性用于在调用 **Update** 方法时更新数据源。

这 4 个命令属性分别是：

- **SelectCommand**
- **InsertCommand**
- **DeleteCommand**
- **UpdateCommand**

ADO 中没有提供这些用来充分控制如何向/从数据源移值数据的选择。如果对命令对象分配了这些属性，还可以控制这些对象在发生诸如出错之类事件时的动作。所以这里需要多用一些代码，但当涉及到数据源的流控制时，你的确可以自如地进行控制。

这些属性的设置非常简单。如果已创建了命令对象，那么只需要给它们分配对应的属性即可，如程序清单 3A.32 和程序清单 3A.33 所示。

程序清单 3A.32 设定 **SqlDataAdapter** 的命令属性

```

1 Const strSQLUserSelect As String = "SELECT * FROM tblUser"
2 Const strSQLUserDelete As String = "DELETE FROM tblUser WHERE Id=@Id"
3 Const strSQLUserInsert As String = "INSERT INTO tblUser(FirstName, " & _
4   "LastName, LoginName) VALUES(@FirstName, @LastName, @LoginName)"
5 Const strSQLUserUpdate As String = "UPDATE tblUser SET FirstName=" & _
6   "@FirstName, LastName=@LastName, LoginName=@LoginName WHERE Id=@Id"
7 Const strConnection As String = "Data Source=USERMANPC;" & _
8   "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
9
10 Dim cnnUserMan As SqlConnection
11 Dim cmdUserSelect As SqlCommand
12 Dim cmdUserDelete As SqlCommand
13 Dim cmdUserInsert As SqlCommand
14 Dim cmdUserUpdate As SqlCommand

```

```

15 Dim dadUserMan As SqlDataAdapter
16 Dim prmsSQLDelete, prmsSQLUpdate, prmsSQLInsert As SqlParameter
17 ' 实例化并打开连接
18 cnnUserMan = New SqlConnection(strConnection)
19 cnnUserMan.Open()
20 ' 命令的实例化
21 cmdUserSelect = New SqlCommand(strSQLUserSelect, cnnUserMan)
22 cmdUserDelete = New SqlCommand(strSQLUserDelete, cnnUserMan)
23 cmdUserInsert = New SqlCommand(strSQLUserInsert, cnnUserMan)
24 cmdUserUpdate = New SqlCommand(strSQLUserUpdate, cnnUserMan)
25
26 ' 实例化数据适配器
27 dadUserMan = New SqlDataAdapter(strSQLUserSelect, cnnUserMan)
28 ' 设定数据适配器的命令属性
29 dadUserMan.SelectCommand = cmdUserSelect
30 dadUserMan.InsertCommand = cmdUserInsert
31 dadUserMan.DeleteCommand = cmdUserDelete
32 dadUserMan.UpdateCommand = cmdUserUpdate
33
34 ' 添加删除命令参数
35 cmdUserDelete.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
36 "FirstName")
37 cmdUserDelete.Parameters.Add("@LastName", SqlDbType.VarChar, 50, "LastName")
38 cmdUserDelete.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
39 "LoginName")
40 prmsSQLDelete = dadUserMan.DeleteCommand.Parameters.Add("@Id", _
41 SqlDbType.Int, Nothing, "Id")
42 prmsSQLDelete.Direction = ParameterDirection.Input
43 prmsSQLDelete.SourceVersion = DataRowVersion.Original
44 ' 添加更新命令参数
45 cmdUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
46 "FirstName")
47 cmdUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, 50, "LastName")
48 cmdUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
49 "LoginName")
50 prmsSQLUpdate = dadUserMan.UpdateCommand.Parameters.Add("@Id", _
51 SqlDbType.Int, Nothing, "Id")
52 prmsSQLUpdate.Direction = ParameterDirection.Input
53 prmsSQLUpdate.SourceVersion = DataRowVersion.Original
54 ' 添加插入命令属性
55 cmdUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
56 "FirstName")
57 cmdUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, 50, "LastName")
58 cmdUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
59 "LoginName")

```

程序清单 3A.33. 设定 OleDbDataAdapter 的命令属性

```

1 Const strSQLUserSelect As String = "SELECT * FROM tblUser"
2 Const strSQLUserDelete As String = "DELETE FROM tblUser WHERE Id=@Id"

```

```

3 Const strSQLUserInsert As String = "INSERT INTO tblUser(FirstName, " & _
4   "LastName, LoginName) VALUES(@FirstName, @LastName, @LoginName)"
5 Const strSQLUserUpdate As String = "UPDATE tblUser SET FirstName=" & _
6   "@FirstName, LastName=@LastName, LoginName=@LoginName WHERE Id=@Id"
7 Const strConnection As String = "Provider=SqlOleDb;Data Source=USERMANPC" & _
8   ";User ID=UserMan;Password=userman;Initial Catalog=UserMan"
9
10 Dim cnnUserMan As OleDbConnection
11 Dim cmdUserSelect As OleDbCommand
12 Dim cmdUserDelete As OleDbCommand
13 Dim cmdUserInsert As OleDbCommand
14 Dim cmdUserUpdate As OleDbCommand
15 Dim dadUserMan As OleDbDataAdapter
16 Dim prmSQLDelete, prmSQLUpdate, prmSQLInsert As OleDbParameter
17 ' 实例化并打开连接
18 cnnUserMan = New OleDbConnection(strConnection)
19 cnnUserMan.Open()
20 ' 命令的实例化
21 cmdUserSelect = New OleDbCommand(strSQLUserSelect, cnnUserMan)
22 cmdUserDelete = New OleDbCommand(strSQLUserDelete, cnnUserMan)
23 cmdUserInsert = New OleDbCommand(strSQLUserInsert, cnnUserMan)
24 cmdUserUpdate = New OleDbCommand(strSQLUserUpdate, cnnUserMan)
25
26 ' 实例化数据适配器
27 dadUserMan = New OleDbDataAdapter(strSQLUserSelect, cnnUserMan)
28 ' 设定数据适配器的命令属性
29 dadUserMan.SelectCommand = cmdUserSelect
30 dadUserMan.InsertCommand = cmdUserInsert
31 dadUserMan.DeleteCommand = cmdUserDelete
32 dadUserMan.UpdateCommand = cmdUserUpdate
33
34 ' 添加删除命令参数
35 cmdUserDelete.Parameters.Add("@FirstName", OleDbType.VarChar, 50, _
36   "FirstName")
37 cmdUserDelete.Parameters.Add("@LastName", OleDbType.VarChar, 50, "LastName")
38 cmdUserDelete.Parameters.Add("@LoginName", OleDbType.VarChar, 50, _
39   "LoginName")
40 prmSQLDelete = dadUserMan.DeleteCommand.Parameters.Add("@Id", _
41   OleDbType.Integer, Nothing, "Id")
42 prmSQLDelete.Direction = ParameterDirection.Input
43 prmSQLDelete.SourceVersion = DataRowVersion.Original
44 ' 添加更新命令参数
45 cmdUserUpdate.Parameters.Add("@FirstName", OleDbType.VarChar, 50, _
46   "FirstName")
47 cmdUserUpdate.Parameters.Add("@LastName", OleDbType.VarChar, 50, "LastName")
48 cmdUserUpdate.Parameters.Add("@LoginName", OleDbType.VarChar, 50, _
49   "LoginName")
50 prmSQLUpdate = dadUserMan.UpdateCommand.Parameters.Add("@Id", _
51   OleDbType.Integer, Nothing, "Id")

```

```
52 prmSQLUpdate.Direction = ParameterDirection.Input
53 prmSQLUpdate.SourceVersion = DataRowVersion.Original
54 ' 添加插入命令参数
55 cmdmUserInsert.Parameters.Add("@FirstName", OleDbType.VarChar, 50, _
56 "FirstName")
57 cmdmUserInsert.Parameters.Add("@LastName", OleDbType.VarChar, 50, "LastName")
58 cmdmUserInsert.Parameters.Add("@LoginName", OleDbType.VarChar, 50, _
59 "LoginName")
```

程序清单 3A.32 和程序清单 3A.33 之间只有一些细小的差别, 并且都归因于所使用的数据提供程序的不同。除设置数据适配器的命令属性外, 还配置了命令对象的参数, 用以处理 UserMan 数据库中的 tblUser 表。其中专门使用了指定参数, 因为我坚信这样做会使代码更容易读懂。

3A.6 小结

本章介绍了两种不同的数据访问技术, ADO 与 ADO.NET。ADO.NET 用于建立分布式多级 Web 应用, 而 ADO 是在多机种网络环境中, 进行传统 Windows 客户机-服务器/多级应用编程的最好办法。在多机种网络环境中, 你每时每刻都链接在本地网上。此外, 本章还介绍了 ADO.NET 连接层的对象模型, 并在需要时与 ADO 中的对象模型做了比较。本章既对 ADO.NET 连接层进行了介绍, 也在一定程度上对 ADO 做了介绍, 同时可作为 ADO.NET 这种数据存取技术中各种连接类、方法以及属性的参考。

本章包括了以下话题:

- **Connection、Command、DataReader 以及 DataAdapter 数据类:** 将 ADO.NET 类与 ADO 中的等价值做了比较, 并给出了何时使用哪种数据类的建议。
- **自动与手动事务处理:** 展示了事务处理边界, 以及在 **Connection** 和 **Transaction** 类中使用它们的方法。
- **.NET 框架中的两个 .NET 数据提供程序 (OLE DB.NET 提供程序和 SQL Client.NET 提供程序)** 都有其自己的相关数据类集合, 且彼此之间不可互换。换句话说, 你不能将 OLE DB .NET 数据提供程序的连接与 SQL Server .NET 数据提供程序的命令类混合起来, 反之亦然。
- **SQL Server .NET 数据提供程序与 OLE DB .NET 数据提供程序:** 讨论了访问 SQL Server 7.0 或更新版本时 SQL Server .NET 数据提供程序的使用方法, 以及访问带有 OLE DB 提供程序的其他数据源时, OLE DB .NET 数据提供程序的使用方法。

下一章将讲述 ADO.NET 非连接层中的类。

第 3B 章

ADO.NET 介绍：非连接层

本章与其姊妹篇第 3A 章都讲述了 ADO.NET 的主要内容。ADO.NET 中的类可分为两组：连接层和非连接层。本章专门讲述非连接层。

如果你是初学 ADO.NET，我建议你在继续学习本章之前先阅读第 3A 章。本章也将为你提供详尽的参考。

3B.1 使用 DataSet 类

DataSet 类相当复杂，非常难以处理和理解。至少要对其有进一步的研究并意识到不需要了解其全部内容的时候，你才会理解这个概念。我将在这里讲述 **DataSet** 类的全部内容，你可以从中选择自己所需的信息。**DataSet** 类是 **System.Data** 名称空间的一部分。

先讲述最重要的内容：数据集在结构上与关系型数据库关系非常密切。说 **DataSet** 类似于关系型数据库，是指 **DataSet** 类实际上揭示了一个分层对象模型（这与分层数据库无关）。该对象模型由表、行和列组成，还包括关系和约束。实际上 **DataSet** 和 **DataTable** 类处理包含以下对象的集合：

- **DataSet** 类处理 **Tables** 集合（包含数据类型为 **DataTable** 的对象）和 **Relations** 集合（包含数据类型为 **DataRelation** 的对象）。
- **DataTable** 类处理 **Rows** 集合（包含数据类型为 **DataRow** 的对象）和 **Columns** 集合（包含数据类型为 **DataColumn** 的对象）。**Rows** 集合处理表中的所有行，而 **Columns** 集合则是表的实际模式。

为充分挖掘 **DataSet** 类的潜力，至少需要结合 **DataTable** 类来一起使用。

简单地说，**DataSet** 类是一种本地容器或内存中的高速缓存，用于存储或缓存从数据库中检索到的数据。可以称之为虚拟数据存储器，因为从数据库中检索到的所有数据，包括数据的模式，均存储在名为数据集的非连接高速缓存中。**DataSet** 的秘密在于尽管断开了到数据源的连接，但还是可以使用 **DataSet** 中的数据，就像使用数据库中的数据一样。可以将 **DataSet** 中的数据看作实际数据的副本。现在你可能怀疑更新实际的数据是否有困难。回答是既困难也容易。说困难是因为这比使用旧的 ADO **Recordset** 类的要求更高，说容易是因为有多种选项可以实现这种数据更新。稍后我们将对此进行讨论。

那该如何从数据库中获取数据并存入 **DataSet** 中呢？这是 ADO.NET 的连接部分

DataAdapter 类的工作。**DataAdapter** 是直接与数据源通讯(在 SQL Server 7.0 或更高版本中)或使用底层的 OLE DB 提供程序来与数据库对话的类。在第 3A 章介绍了 **DataAdapter** 类。

关于 **DataSet** 类需要注意的是, **DataSet** 类并不是子类, 或者说 **DataSet** 类可与所有类型的数据源共同工作, 不存在 **SqlDataSet** 或 **OleDbDataSet** 类。

3B.1.1 Recordset 与 DataSet

如果你对 ADO 非常熟悉, 就应知道 **Recordset** 对象并理解 **Recordset** 对象如何与各种不同的指针一起工作。尽管在 ADO.NET 中并没有直接与 **Recordset** 对象等价的对象, 但我在这里还是会给出了一些生成 **DataSet** 类及其相关的数据类 **DataTable** 的示例, 这些类在行为上与 **Recordset** 对象相似。在最初设计 ADO.NET 时, 一个主要的障碍是寻找一种将连接的 ADO **Recordset** 类转变为非连接状态的方法。在 ADO.NET 中, ADO **Recordset** 类被映射为几种不同的类。包括非连接的 **DataSet** 类, 它使用连接的 **DataAdapter** 类来检索数据。尽管 ADO 通过远程数据服务 (RDS) 引入了非连接记录集 (recordset) 的概念, 但设置的连接部分依然在幕后起作用。

简单地说, **DataSet** 类实际上就是非连接 **Recordset** 对象的集合, **Recordset** 对象对 **DataTable** 类完全开放。因此, 如果喜欢使用 ADO **Recordset** 对象, 但又想同时使用 ADO.NET, 则可以选择 **DataTable** 类。**DataTable** 类的工作方式与 ADO **Recordset** 相同。详细信息请参阅本章后面的“使用 **DataTable** 类”部分。

数据源独立性

DataSet 的强大之处在于它完全独立于数据源。换句话说, **DataSet** 是一种数据容器或高速缓存, 存储从数据源复制的数据。**DataSet** 是非连接的, 因而可以独立于数据源, 这一事实使 **DataSet** 成为包含来自多个数据源 (例如来自各种数据库的表) 的理想容器。

使用 XML 格式

当把数据从数据源 (例如数据库) 中移动到数据集 (或其他途径) 时, 所采用的格式就是 XML。这是 ADO.NET 的一个关键概念, 一切都是基于 XML 的。由于这种 XML 格式, 你不仅可以跨进程和机器边界来传输数据, 甚至还可以跨越使用防火墙的网络来传输数据。的确, XML 使得这一切都成为可能。现在你真正拥有了一种, 用来操作从所有支持 XML 的数据源传输到 Internet 上的数据的方法。

这种用于传输数据的格式就是 XML, 但这并非 ADO.NET 使用 XML 的惟一地方。XML 是 ADO.NET 中所有数据的基础格式, 而如果需要持续进行或将数据串行化成一个文件, 则所使用的格式仍是 XML。这意味着你可以使用所有支持 XML 的阅读器来读取这种持续的文件。

在继续学习之前, 你应该先弄清楚一点: 尽管 ADO.NET 中的底层或基础的数据格式是 XML, 但在数据集中的数据并不使用 XML 来表达。当你在数据集和数据源之间交换数据时, ADO.NET 数据 API 会自动处理 XML 数据的创建过程, 但数据集里的数据却采用一种更高效的格式。因此实际上并不需要为使用 ADO.NET 而了解有关 XML 的任何内容。但我建议你学习 XML 的基础知识, 因为 XML 是 Visual Studio.NET 密不可分的一部分。

类型化与非类型化数据集

数据集可以是类型化或非类型化的。它们之间的区别在于类型化的数据集有一个模式，而非类型化的数据集没有。你可以在应用程序中选择使用任意一种数据集，但是需要知道类型化的数据集在 Visual Studio 中有更多的支持，因而有更多的工具便于你使用。

类型化数据集通过强类型编程可以更容易地访问表字段的内容。强类型编程使用来自底层数据模式的信息。这意味着你直接针对声明过的对象进行编程，而不是针对实际试图操作的数据。类型化数据集有一个对 XML 模式文件的引用，该模式文件 (*.xsd) 描述了所有包含在数据集中的表的结构。



有关如何创建类型化数据集的详细内容请参阅第 4 章。

在程序清单 3B.1 中，你会看到强类型如何改变正常访问表中列内容的方式。

程序清单 3B.1 强类型与弱类型

```
1 ' 显示来自 ADO 记录集的值
2 MsgBox(rstUser.Fields("FirstName").Value)
3 ' 使用强类型显示来自 ADO.NET 数据集的值
4 MsgBox(dstUser.tblUser(0).FirstName)
5 使用弱类型显示来自 ADO.NET 数据集的值
6 MsgBox(dstUser.Tables("tblUser").Columns(3).ToString)
```

正如在程序清单 3B.1 中所看到的那样，使用强类型时语法简单到直接使用表名和字段名来引用表和字段（参阅第 4 行）。rstUser 记录集和 dstUser 数据集均已在他处被声明、实例化并被打开或填充过了。

3B.1.2 DataSet 属性

表 3B.1 中按字母顺序显示了 DataSet 类的各种属性。请注意这里显示的仅仅是公共的、非继承的属性。

表 3B.1 DataSet 类属性

名称	说明
CaseSensitive	该属性可读写，决定了包含在 Tables 集合中的 DataTable 对象中的字符串比较是否区分大小写。默认值是 False。该属性还会影响筛选、排序和搜索操作所执行的方式。在设置该属性的同时，还默认地为 Tables 集合中的 DataTable 对象设置了相同的属性
DataSetName	返回或设置数据集的名称
DefaultViewManager	返回数据集中的视图。对该视图可进行筛选或排序，你可以通过它进行搜索或导航。返回值的数据类型是 DataViewManager
EnforceConstraints	该属性值决定了执行更新操作时是否强制应用约束规则。默认值是 True。如果无法强制应用一个或更多约束，则抛出 ConstraintException 异常

续表

名称	说明
ExtendedProperties	返回定制属性或定制用户信息的集合。返回的数据类型是 PropertyCollection 类。你可以使用集合的 Add 方法向属性集合添加信息，如下所示： <code>dstUserMan.ExtendProperties.Add("New Property", "New Property Value")</code>
HasErrors	返回一个 Boolean 值，指明在数据集表的行中是否存在错误。在检查任何拥有 HasErrors 属性的单个表之前都可以使用该属性
Locale	该属性可读写，用来保存区域信息以比较表中的字符串。其值是 CultureInfo 类型，默认时为 Nothing
Namespace	该属性可读写，用来保存数据集的名称空间。将 XML 文档读入数据集或将数据集写入 XML 文档时都可使用该属性
Prefix	该属性可读写，用来保存数据集名称空间的前缀。该前缀用于标识 XML 文档中哪些元素属于该数据集对象的名称空间。用 NameSpace 属性设置名称空间
Relations	从数据集中返回关系集合。返回值的数据类型为 DataRelationCollection ，负责控制 DataRelation 类型的对象。如果不存在数据关系则返回 Nothing
Tables	从数据集中返回表的集合。返回值的数据类型是 DataTableCollection ，负责控制 DataTable 类型的对象。如果不存在 DataTable 则返回 Nothing

3B.1.3 DataSet 方法

表 3B.2 按字母顺序列出了数据集公共的、非继承的方法。

表 3B.2 DataSet 类的方法

名称	说明	示例
AcceptChanges()	该方法接受或提交自从上次调用该方法或加载数据集时起对该数据集所的全部更改。请注意，该方法对 Tables 集合中的所有表都调用 AcceptChanges 方法，该过程对每个 DataTable 的 Rows 集合的所有 DataRow 对象依次调用 AcceptChanges 方法。因此如果需要仅接受对特定表的更改，则可以在该表中调用 AcceptChanges 方法	<code>dstUser.AcceptChanges()</code>
BeginInit()	BeginInit 方法用于指示数据集尚未初始化，此方法不能被覆盖。该方法开始对表单上的数据集进行初始化。实际上，其他组件也可使用该方法。这种初始化发生在运行时，与 EndInit 方法一起运行，以确保数据集在完全初始化之前未被使用	<code>dstUser.BeginInit()</code>
Clear()	该方法可以清除数据集中的数据，这意味着所有表中的行都将被移除。该方法不会清除数据结构，只是清除数据本身	<code>dstUser.Clear()</code>

续表

名称	说明	示例
Clone()	Clone 方法用于克隆或复制数据集的数据结构, 不包括数据本身, 只是克隆或复制表、模式、关系和约束。如果需要同时复制数据, 则需要使用 Copy 方法	<code>dstClone = dstUser.Clone()</code>
Copy()	Copy 方法用于克隆数据集的数据结构以及从数据集中将数据复制到新的数据集中。如果仅需要克隆数据结构, 则应该使用 Clone 方法	<code>dstCopy = dstUser.Copy()</code>
EndInit()	EndInit 方法用来与 BeginInit 方法结合使用。一旦调用该方法, 则表示该数据集已被完全初始化	<code>dstUser.EndInit()</code>
GetChanges()	该重载方法用于恢复数据集的副本, 该数据集包含自从上次调用 AcceptChanges 方法或加载该数据集时起所做的所有更改。使用该属性时可以不带任何参数, 也可以通过指定 DataRowState 枚举的成员来指明需要返回对数据集所做的何种更改	<code>dstChanges = dstUser.GetChanges()</code> 或 <code>dstAdded = dstUser.GetChanges(DataRowState.Added)</code>
GetXml()	GetXml 方法将数据集中的数据作为字符串变量的 XML 数据返回	<code>strXMLData = dstUser.GetXml()</code>
GetXmlSchema()	GetXmlSchema 方法为字符串变量数据集中的数据返回 XSD 模式	
HasChanges()	该方法可用于检测数据集中的数据是否有过更改。该方法是重载方法, 其中的一种变体形式将 DataRowState 枚举中的一个成员作为其参数。这样你就可以指定是否仅想检测特定的更改, 如添加行。系统将返回一个 Boolean 类型的值, 以指示是否发生了更改	<code>blnChanges = dstUser.HasChanges()</code> 或 <code>blnChanges = dstUser.HasChanges(DataRowState.Added)</code>
InferXmlSchema(ByVal Schema, ByVal varstrURI() As String)	这种重载方法将 XML 模式从 XmlReader 、 Stream 、 TextReader 或某文件中推断或复制到数据集中。varstrURI 参数是名称空间 URI 的数组, 用以阻止推断/复制	<code>dstUser.InferXmlSchema(xrdUser, arrstrURIExclude)</code>
Merge()	Merge 方法是重载方法, 用于将数据集与 DataRow 对象的数组形式的其他数据并入其他数据集或 DataTable 中	<code>dstUser.Merge(arrdrwMerge)</code> 或 <code>dstUser.Merge(dtbMerge)</code> 或 <code>dstUser.Merge(dstMerge)</code>
ReadXml()	这该重载方法用于将 XML 模式和数据读入数据集。可从 Stream 、文件、 TextReader 或 XmlReader 中读取 XML 模式和数据。如果仅想读取 XML 模式, 则可使用 ReadXmlSchema 方法	<code>dstUser.ReadXml(stmUser)</code> 或 <code>dstUser.ReadXml(strXMLFile)</code> 或 <code>dstUser.ReadXml(trdUser)</code> 或 <code>dstUser.ReadXml(xrdUser)</code>
ReadXmlSchema()	该重载方法用于将 XML 模式读入数据集。可从 XmlReader 、 Stream 、文件或 TextReader 中读取 XML 模式。如果想读取 XML 模式和数据, 则可使用 ReadXml 方法	<code>dstUser.ReadXmlSchema(xrdUser)</code> 或 <code>dstUser.ReadXmlSchema(stmUser)</code> 或 <code>dstUser.ReadXmlSchema(strXMLFile)</code> 或 <code>dstUser.ReadXmlSchema(trdUser)</code>

续表

名称	说明	示例
RejectChanges()	该方法用来拒绝或回滚自从上次调用 AcceptChanges 方法或加载 DataSet 时起对数据集所做的更改。请注意, 该方法将在 Tables 集合中针对所有表调用 RejectChanges 方法, 该过程对每个 DataTable 的 Rows 集合中的所有 DataRow 对象依次调用 RejectChanges 方法。因此, 如果仅需要拒绝对特定表的更改, 则可针对该表调用 RejectChanges 方法	<code>dstUser.RejectChanges()</code>
Reset()	该方法将数据集重置为初始状态, 而其本身能被覆盖	<code>dstUser.Reset()</code>
WriteXml()	这种重载方法用于从数据集中编写 XML 模式和数据。可将 XML 模式和数据写入 XmlWriter 、 Stream 、文件或 TextWriter 中。如果仅编写 XML 模式, 则可使用 WriteXmlSchema 方法	<code>dstUser.WriteXml(xwrUser)</code> 或 <code>dstUser.WriteXml(stmUser)</code> 或 <code>dstUser.WriteXml(strXMLFile)</code> 或 <code>dstUser.WriteXml(twrUser)</code>
WriteXmlSchema()	这种重载方法用于从数据集中编写 XML 模式。可将 XML 模式写入 XmlWriter 、 Stream 、文件或 TextWriter 。如果想编写 XML 模式和数据, 则可使用 WriteXml 方法	<code>dstUser.WriteXmlSchema(xrdUser)</code> 或 <code>dstUser.WriteXmlSchema(stmUser)</code> 或 <code>dstUser.WriteXmlSchema(strXMLFile)</code> 或 <code>dstUser.WriteXmlSchema(trdUser)</code>

3B.1.4 实例化 DataSet

实例化 **DataSet** 对象没有什么问题, 可以在声明 **DataSet** 时对其进行实例化, 如下所示:

```
Dim dstUnnamed As New DataSet()
Dim dstNamed As New DataSet("UserManDataSet")
```

从以上两行代码中可以看出, 两者的惟一区别在于赋给 **dstNamed** 数据集的名称不同。给数据集这样命名在数据集持续为 XML 格式时特别有用, 因为此后可以给 XML 文档元素赋予易于识别的名称。

也可以先声明变量然后再进行实例化, 如下所示:

```
Dim dstUnnamed As DataSet
Dim dstNamed As DataSet

dstUnnamed = New DataSet()
```

```
dstNamed = New DataSet("UserManDataSet")
```

3B.1.5 使用 DataAdapter 填充 DataSet

一旦设置好数据适配器和数据集，就可以填充数据集了。**DataAdapter** 类的 **Fill** 方法用于填充和刷新 **DataSet** 对象。**Fill** 方法是重载方法，对 **OleDbDataAdapter** 类的支持比支持 **SqlDataAdapter** 的方法变体形式更多。这是因为 **OleDbDataAdapter** 类支持填充来自 ADO **Recordset** 对象或 ADO **Record** 对象的数据集。确实，你可能会对旧的 ADO **Recordset** 对象和新的 ADO **Record** 对象感到困惑。在表 3B.3 中显示了不同变体形式的 **Fill** 方法。此处的示例代码假定已经实例化和初始化了各种对象。从前面的程序清单中可以看到如何进行实例化和初始化，因为它们原本就是采用相同的代码构建的。

表 3B.3 重载的 **Fill** 方法 (**DataAdapter**) 的各种版本

示例代码	说明
<pre>intNumRows = dadUserMan.Fill(ds tUser)</pre>	<p>该种变体形式用来填充 dstUser 数据集并保存 intNumRows 返回的行数。由于未指定表名，所以 Tables 集合中的新数据表被称为“Table”。可以使用类似下面的语句来检查表名：MsgBox(dstUserMan.Tables(0).TableName)</p>
<pre>intNumRows = dadUserMan.Fill(ds tUser, "tblUser")</pre>	<p>该种变体形式用来填充 dstUser 数据集并返回 intNumRows 中返回的行数。由于已指定源表名，所以数据集中 Tables 集合的新数据表被称为“tblUser”，就像在数据源中一样。可以使用如下语句检查表名：MsgBox(dstUserMan.Tables(0).TableName)</p>
<pre>intNumRows = dadUserMan.Fill(ds tUser, 0, 2, "tblUser")</pre>	<p>该种变体形式用来填充 dstUser 数据集并返回在 intNumRows 中返回的行数。由于已指定数据源表名，所以数据集中 Tables 集合的新数据表被称为“tblUser”，就像在数据源中一样。中间的两个参数 (0 和 2) 表明只返回索引号为 0~2 的行。如果你不希望返回所有行，就应该使用这种方法。可以在 intNumRows 变量中检查返回行的数目</p>
<pre>intNumRows = dadUserMan.Fill(dt bUser)</pre>	<p>该种变体形式实际上并不用来填充数据集，而是填充 DataTable 对象。返回的行数保存在 intNumRows 中。由于未指定源表名，所以该数据表没有名称。可以在对其实例化时或在实例化 DataTable 之后使用类似 dtbUser = New DataTable("tblUser") 的语句指定 DataTable 名。通过设置 TableName 属性可以实现这一点</p>
<pre>intNumRows = dadUserMan.Fill(dt bUser, rstADO) (仅限 于OleDbDataAdapter)</pre>	<p>该种变体形式实际上并不是用来填充数据集，而是填充 DataTable 对象。返回的行数保存在 intNumRows 中，ADO Recordset 对象中的行则被复制到 DataTable 中。此外，还可以指定 ADO Record 对象代替 rstADO Recordset 对象</p>
<pre>intNumRows = dadUserMan.Fill(ds tUser, rstADO, "tblUser") (仅限于 OleDbDataAdapter)</pre>	<p>该种变体形式用来填充 dstUser 数据集并返回 intNumRows 中返回的行数，该行数保存在 intNumRows 中。由于已指定源表名，所以数据集中 Tables 集合的新数据表被称为“tblUser”。而 ADO Recordset 对象的行则被复制到数据表中。此外，还可以指定 ADO Recordset 对象代替 rstADO Recordset 对象</p>

我并不想泛泛而谈如何使用 ADO **Recordset** 和 ADO 类，也许这是向读者显示完整代码示例的一个好机会，该示例打开了一个 ADO **Recordset**，并使用该 **Recordset** 填充 **DataSet**。程序清单 3B.2 显示了如何操作。

程序清单 3B.2 使用 ADO **Recordset** 填充 **DataSet**

```
1 Public Sub FillDataSetFromRecordset()  
2   Const STR_SQL_USER_SELECT As String = "SELECT * FROM tblUser"  
3   Const STR_CONNECTION As String = "Provider=SQLOLEDB;" & _  
4     "Data Source=USERMANPC;User ID=UserMan;Password=userman;" & _  
5     "Initial Catalog=UserMan"  
6  
7   Dim cnnUserMan As OleDbConnection  
8   Dim dadUserMan As OleDbDataAdapter  
9   Dim dstUserMan As DataSet  
10  
11  Dim rstUser As ADODB.Recordset  
12  Dim cnnADOUserMan As ADODB.Connection  
13  
14  Dim intNumRows As Integer  
15  
16  ' 实例化并打开连接  
17  cnnUserMan = New OleDbConnection(STR_CONNECTION)  
18  cnnUserMan.Open()  
19  cnnADOUserMan = New ADODB.Connection()  
20  cnnADOUserMan.Open(STR_CONNECTION)  
21  
22  ' 实例化数据适配器  
23  dadUserMan = New OleDbDataAdapter(STR_SQL_USER_SELECT, cnnUserMan)  
24  
25  ' 实例化 dataset  
26  dstUserMan = New DataSet()  
27  ' 实例化 recordset  
28  rstUser = New ADODB.Recordset()  
29  
30  ' 填充记录集  
31  rstUser.Open(STR_SQL_USER_SELECT, cnnADOUserMan)  
32  ' 填充数据集  
33  intNumRows = dadUserMan.Fill(dstUserMan, rstUser, "tblUser")  
34 End Sub
```

在程序清单 3B.2 中使用了一些 ADO 类，为此，必须从工程中添加对 ADO COM 库的引用。有关该操作的详细信息请参阅本章后面的“COM Interop”。编写程序清单 3B.2 中的代码并不需要很多技巧，我在这里打开了两个连接，一个 ADO.NET 连接和一个 ADO 连接。之后再实例化该数据适配器和记录集，填充记录集并使用该记录集填充数据集。可以使用 ADO **Record** 对象代替 **Recordset** 对象并/或填充 **DataTable** 来代替 **DataSet**。有关如何使用 **DataAdapter** 的 **Fill** 方法以实现上述过程的详细信息，请查看表 3B.3。

3B.1.6 使用 DataAdapter 更新数据源

完成对数据集中数据的操作之后,即可更新数据源。可以使用 **DataAdapter** 类的 **Update** 方法来更新数据源。该方法负责检查 **Rows** 集合中每个 **DataRow** 对象的 **RowState** 属性。**Rows** 集合是 **DataTable** 对象的成员,而 **DataTable** 对象包含在数据集的 **Tables** 集合中。因此数据适配器在数据集 **Tables** 集合的所有表中开始循环,对每个表,数据适配器在 **Rows** 集合中都用循环来检查 **RowState** 属性。如果插入、更新或删除了某行, **DataAdapter** 就会使用一种命令属性来处理更新。这意味着如果要插入新行,就使用 **InsertCommand** 属性;如果要更新已存在的行,就使用 **UpdateCommand** 属性;如果要删除已存在的行,则使用 **DeleteCommand** 属性。

在程序清单 3B.3 中设置了一个数据适配器、一个数据集和一些命令对象来处理 **UserMan** 数据库的 **tblUser** 表。之后便在表中添加、修改和删除了一行。然后用 **HasChange** 方法检查数据集中的数据是否发生了更改。如果发生了更改,则使用 **GetChange** 方法将所有改动过的行加载到 **dstChanges** 数据集中。如果此期间发生错误则拒绝更改,否则使用 **Update** 方法更改数据源。程序清单 3B.3 中的代码并不能处理更新数据源时所发生的错误,欲了解如何处理错误,请查看第 5 章的示例代码和有关建议。

程序清单 3B.3 将更改写回到数据源

```

1 Public Sub UpdateSqlDataSet()
2     Const STR_SQL_USER_SELECT As String = "SELECT * FROM tblUser"
3     Const STR_SQL_USER_DELETE As String = "DELETE FROM tblUser WHERE Id=@Id"
4     Const STR_SQL_USER_INSERT As String = "INSERT INTO tblUser(FirstName" & _
5         ", LastName, LoginName) VALUES(@FirstName, @LastName, @LoginName)"
6     Const STR_SQL_USER_UPDATE As String = "UPDATE tblUser SET FirstName=" & _
7         "@FirstName, LastName=@LastName, LoginName=@LoginName WHERE Id=@Id"
8     Const STR_CONNECTION As String = "Data Source=USERMANPC" & _
9         ";User ID=UserMan;Password=userman;Initial Catalog=UserMan"
10
11     Dim cnnUserMan As SqlConnection
12     Dim cmdUserSelect As SqlCommand
13     Dim cmdUserDelete As SqlCommand
14     Dim cmdUserInsert As SqlCommand
15     Dim cmdUserUpdate As SqlCommand
16     Dim dadUserMan As SqlDataAdapter
17     Dim dstUserMan, dstChanges As DataSet
18     Dim drwUser As DataRow
19     Dim prmSQLDelete, prmSQLUpdate, prmSQLInsert As SqlParameter
20
21     ' 实例化并打开连接
22     cnnUserMan = New SqlConnection(STR_CONNECTION)
23     cnnUserMan.Open()
24
25     ' 实例化命令
26     cmdUserSelect = New SqlCommand(STR_SQL_USER_SELECT, cnnUserMan)

```



```
27  cmmUserDelete = New SqlCommand(STR_SQL_USER_DELETE, cnnUserMan)
28  cmmUserInsert = New SqlCommand(STR_SQL_USER_INSERT, cnnUserMan)
29  cmmUserUpdate = New SqlCommand(STR_SQL_USER_UPDATE, cnnUserMan)
30
31  ' 实例化数据适配器
32  dadUserMan = New SqlDataAdapter(STR_SQL_USER_SELECT, cnnUserMan)
33  ' 设置数据适配器命令属性
34  dadUserMan.SelectCommand = cmmUserSelect
35  dadUserMan.InsertCommand = cmmUserInsert
36  dadUserMan.DeleteCommand = cmmUserDelete
37  dadUserMan.UpdateCommand = cmmUserUpdate
38
39  ' 添加 Delete 命令参数
40  cmmUserDelete.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
41    "FirstName")
42  cmmUserDelete.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
43    "LastName")
44  cmmUserDelete.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
45    "LoginName")
46  prmsQLDelete = dadUserMan.DeleteCommand.Parameters.Add("@Id", _
47    SqlDbType.Int, Nothing, "Id")
48  prmsQLDelete.Direction = ParameterDirection.Input
49  prmsQLDelete.SourceVersion = DataRowVersion.Original
50  ' 添加 Update 命令参数
51  cmmUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
52    "FirstName")
53  cmmUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
54    "LastName")
55  cmmUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
56    "LoginName")
57  prmsQLUpdate = dadUserMan.UpdateCommand.Parameters.Add("@Id", _
58    SqlDbType.Int, Nothing, "Id")
59  prmsQLUpdate.Direction = ParameterDirection.Input
60  prmsQLUpdate.SourceVersion = DataRowVersion.Original
61  ' 添加 Insert 命令参数
62  cmmUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
63    "FirstName")
64  cmmUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
65    "LastName")
66  cmmUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
67    "LoginName")
68
69  ' 实例化 dataset
70  dstUserMan = New DataSet()
71  ' 填充数据集
72  dadUserMan.Fill(dstUserMan, "tblUser")
73
74  ' 添加新行
75  drwUser = dstUserMan.Tables("tblUser").NewRow()
```

```

76 drwUser("FirstName") = "New User"
77 drwUser("LastName") = "New User LastName"
78 drwUser("LoginName") = "NewUser"
79 dstUserMan.Tables("tblUser").Rows.Add(drwUser)
80
81 ' 更新已存在的行 (索引号为 5)
82 dstUserMan.Tables("tblUser").Rows(5)(5) = "OldUser"
83
84 ' 删除索引号为 6 的行
85 dstUserMan.Tables("tblUser").Rows(6).Delete()
86
87 ' 检查数据集中的数据是否发生改变
88 If dstUserMan.HasChanges() Then
89     ' 将所有发生更改的行保存在新数据集中
90     dstChanges = dstUserMan.GetChanges()
91     ' 检查发生更改的行是否有错误
92     If dstChanges.HasErrors() Then
93         ' 拒绝更改
94         dstUserMan.RejectChanges()
95     Else
96         ' 更新数据源
97         dadUserMan.Update(dstChanges, "tblUser")
98     End If
99 End If
100 End Sub

```

3B.1.7 从 DataSet 中清除数据

在向数据集或结构的组成元素中添加表、关系、约束等内容后，经常需要从数据集中清除所有数据。使用 **Clear** 方法可以轻松地实现这一点，如程序清单 3B.4 所示：

程序清单 3B.4 从 DataSet 中清除数据

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5
6 ' 实例化并打开连接
7 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
8     "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
9 cnnUserMan.Open()
10 ' 实例化命令
11 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
12 ' 实例化并初始化数据适配器
13 dadUser = New SqlDataAdapter()
14 dadUser.SelectCommand = cmdUser
15 ' 实例化数据集

```

```
16 dstUser = New DataSet()
17 ' 填充数据集
18 dadUser.Fill(dstUser, "tblUser")
19 ' 输入你的资料
20 ...
21 ' 从数据集中清除数据
22 dstUser.Clear()
```

3B.1.8 复制 DataSet

有时因为各种原因需要复制数据集。如果出于测试目的而需要对一些数据进行处理, 复制数据集便是一个好办法, 它可以保证原始数据完整无缺。根据你的实际需要, 有两种方法可以实现这一点: 仅复制数据结构, 或者复制数据结构及其数据。我将在下面两小节分别进行讲述。

复制 DataSet 的数据结构

你是否平时也有复制或者克隆数据集的数据结构的需要? 数据集的 **Clone** 方法可以完成这种操作。有关示例请参阅程序清单 3B.5。

程序清单 3B.5 从 DataSet 中克隆数据结构

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dstClone As DataSet
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据集
12 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dstUser = New DataSet()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmdUser
17 ' 填充数据集
18 dadUser.Fill(dstUser, "tblUser")
19 ' 克隆数据集
20 dstClone = dstUser.Clone()
```

复制 DataSet 的数据和结构

当需要从数据集中完整复制其中包含的数据结构和数据时, 可以使用 **Copy** 方法。有关示例请参阅程序清单 3B.6。

程序清单 3B.6 从 DataSet 中复制数据结构和数据

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dstCopy As DataSet
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据集
12 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dstUser = New DataSet()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmdUser
17 ' 填充数据集
18 dadUser.Fill(dstUser, "tblUser")
19 ' 复制数据集
20 dstCopy = dstUser.Copy()

```

3B.1.9 将 DataSet 中的数据与其他数据进行合并

有时需要将数据集中的数据与其他形式的数据库合并 (Combine) 在一起。例如, 现有一个 **DataSet**, 该 DataSet 已用 **DataAdapter** 的 **Fill** 方法填充过数据结构和数据, 准备将其与通过编程手段创建好的 **DataSet** 或 **DataTable** 合并。开始操作该数据之后, 你可能想合并这两个对象中的数据。将这些数据合并到数据集中就实现这一目标。使用 **DataSet** 的 **Merge** 方法可以合并类似 **DataRow** 对象的数组、**DataTable** 对象或 **DataSet** 对象的数据。合并后最终的数据集将替换执行 **Merge** 方法的数据集中的数据。有关使用 **Merge** 方法的示例, 请参阅程序清单 3B.7、程序清单 3B.8 和程序清单 3B.9。

程序清单 3B.7 将 DataSet 对象与 DataRow 对象的数组进行合并

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dtbUser As DataTable
6 Dim arrdrwUser(0) As DataRow
7 Dim drwUser As DataRow
8
9 ' 实例化并打开连接
10 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
11 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
12 cnnUserMan.Open()

```

```
13 ' 实例化命令、数据集和数据表
14 cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
15 dstUser = New DataSet()
16 dtbUser = New DataTable()
17 ' 实例化并初始化数据适配器
18 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
19 dadUser.SelectCommand = cmmUser
20 ' 填充数据集
21 dadUser.Fill(dstUser, "tblUser")
22 ' 创建新行并填充数据
23 drwUser = dstUser.Tables("tblUser").NewRow()
24 drwUser("LoginName") = "NewUser1"
25 drwUser("FirstName") = "New"
26 drwUser("LastName") = "User"
27 arrdrwUser.SetValue(drwUser, 0)
28 ' 合并数据集和数据行数组
29 dstUser.Merge(arrdrwUser)
```

程序清单 3B.8 合并两个 DataSet 对象

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmmUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dstCopy As DataSet
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9   "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据集
12 cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dstUser = New DataSet()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmmUser
17 ' 填充数据集
18 dadUser.Fill(dstUser, "tblUser")
19 ' 复制数据集
20 dstCopy = dstUser.Copy()
21 ' 利用该数据集输入你的资料
22 ...
23 ' 合并两个数据集
24 dstUser.Merge(dstCopy)
```

程序清单 3B.9 合并 DataSet 对象和 DataTable 对象

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmmUser As SqlCommand
```

```

3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dtbUser As DataTable
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9   "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令、数据集和数据表
12 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dstUser = New DataSet()
14 dtbUser = New DataTable()
15 ' 实例化并初始化数据适配器
16 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
17 dadUser.SelectCommand = cmdUser
18 ' 填充数据集和数据表
19 dadUser.Fill(dstUser, "tblUser")
20 dadUser.Fill(dtbUser)
21 ' 利用该数据集和数据表输入你的资料
22 ...
23 ' 合并数据集和数据表
24 dstUser.Merge(dtbUser)

```

请注意，在程序清单 3B.7~3B.9 中选择创建了数据结构，并使用数据适配器的 **Fill** 方法填充了数据表。你可以自己创建数据结构，并填入来自各种数据源的数据，随后将其与数据集合并。有关如何操作的更多信息请参阅本章后面的“使用 **DataTable** 类”一节。

3B.1.10 检测和处理对 **DataSet** 中数据所做的更改

有时需要知道数据集中的数据是否已经更改。这里所说的更改包括插入新行、删除行和修改行。**DataSet** 类的 **HasChanges** 方法便可以用于检测。该方法实际上是为 **Tables** 集合中的单个 **DataTable** 对象而设计的，但是如果想知道数据集中的数据是否发生更改，就需要使用该数据集的方法。

HasChanges 方法是重载方法，从程序清单 3B.10 和程序清单 3B.11 中可以了解到如何使用该方法的各种变体形式。

程序清单 3B.10 检测 **DataSet** 对象中对数据进行的所有更改

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dstChanges As DataSet
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _

```

```
9  "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据集
12 cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dstUser = New DataSet()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmmUser
17 ' 填充数据集
18 dadUser.Fill(dstUser, "tblUser")
19 ' 使用数据集输入你的资料
20 ...
21 ' 检查数据集中是否有数据发生更改
22 If dstUser.HasChanges() Then
23     ' 将所有更改保存在新数据集中
24     dstChanges = dstUser.GetChanges()
25 End If
```

在程序清单 3B.10 中, 所有数据更改均被简单地保存在新数据集中。显然, 这并不会影响原来的数据, 然而一旦将更改隔离开, 就可以操作数据和检查错误了。在更新数据源时这一点尤其有用。不要忘记该 **DataSet** 与数据源之间是非连接的, 任何更改在显式更新数据源之前都不会被写回数据源。

因为程序清单 3B.11 中的代码与程序清单 3B.10 中的代码基本相同, 所以我们取了其中的一小段, 但它显示了如何将不同的更改筛选到不同的数据集中。

程序清单 3B.11 检测 DataSet 对象中不同的数据更改

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmmUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser As DataSet
5 Dim dstChanges As DataSet
6 Dim dstAdditions As DataSet
7 Dim dstDeletions As DataSet
8
9 ' 实例化并打开连接
...
21. ' 检查数据集中的数据是否发生更改
22 If dstUser.HasChanges() Then
23. ' 将所有修改过的行保存在新数据集中
24     dstChanges = dstUser.GetChanges(DataRowState.Modified)
25     ' 将所有添加的行保存在新数据集中
26     dstAdditions = dstUser.GetChanges(DataRowState.Added)
27     ' 将所有删除的行保存在新数据集中
28     dstDeletions = dstUser.GetChanges(DataRowState.Deleted)
29 End If
```

3B.1.11 接受或拒绝对数据集中数据的更改

对数据集中的数据进行更改后，可以选择拒绝和接受更改。**DataSet** 类有两种方法用来处理这个问题：**AcceptChanges** 方法和 **RejectChanges** 方法。需要注意的是，这些方法是对数据集中的数据产生作用，而不是数据源本身。这要归根于数据集是非连接的以及因此而导致的数据集不与数据源交互这一事实。

AcceptChanges 方法

使用该方法时，对 **Tables** 集合里的数据表（data table）中的行所做的任何更改均会被接受。在表集合中针对每个 **DataTable** 对象分别调用 **AcceptChanges** 方法就可以实现这一点。当对某个数据表调用 **AcceptChanges** 方法时，该数据表在 **Rows** 集合中会针对每个 **DataRow** 对象调用 **AcceptChanges** 方法。接着检查每个数据行的 **RowState** 属性。如果行状态是 **Added** 或 **Modified**，则 **RowState** 属性被更改为 **Unchanged**。而状态为 **Deleted** 的行将从各自的数据表中被删除。如果在调用 **AcceptChanges** 方法时，**DataRow** 处于编辑状态，则该行将结束编辑模式。有关如何使用 **AcceptChanges** 方法以及该方法是如何影响 **DataSet** 的示例，请参考下一节中的程序清单 3B.12。如果在调用数据适配器的 **Update** 方法之前调用 **AcceptChanges** 方法，则不会将更改写回到数据源中，因为你只是接受了更改，但这些更改却仍被标记为 **Unchanged**！



如果数据集中包含 **ForeignKeyConstraint** 对象，那么一旦调用 **AcceptChanges** 方法就会强制应用 **AcceptRejectRule** 属性。**AcceptRejectRule** 属性用来判定更改或删除是否适用于某个关系。

RejectChanges 方法

当使用 **RejectChanges** 方法时，对 **Tables** 集合里的数据表中的行所做的任何更改均会被拒绝。在表集合中针对每个 **DataTable** 对象分别调用 **RejectChanges** 方法就可实现这一点。当针对某个数据表调用 **RejectChanges** 方法时，该数据表在 **Rows** 集合中会对每个 **DataRow** 对象调用 **RejectChanges** 方法。然后检查每个数据行的 **RowState** 属性。如果行状态为 **Added**，则会从各自的数据表中被删除。如果行状态是 **Modified** 或 **Deleted**，则将 **RowState** 属性改为 **Unchanged**，并恢复该行的原始内容。如果在调用 **RejectChanges** 方法时，**DataRow** 处于编辑状态，则取消编辑模式。有关如何使用 **RejectChanges** 方法以及该方法是如何影响 **DataSet** 的示例，请参考程序清单 3B.12。

程序清单 3B.12 接受或拒绝对 **DataSet** 对象中的数据所做的更改

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dstUser, dstChanges As DataSet
```



```
5 Dim drwUser As DataRow
6 Dim intCounter As Integer
7
8 ' 实例化并打开连接
9 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
10 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
11 cnnUserMan.Open()
12 ' 实例化命令和数据集
13 cmmUser = New SqlCommand("SELECT * FROM tblUser",cnnUserMan)
14 dstUser = New DataSet()
15 ' 实例化并初始化数据适配器
16 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
17 dadUser.SelectCommand = cmmUser
18 ' 填充数据集
19 dadUser.Fill(dstUser, "tblUser")
20 ' 使用来自 user 表的模式创建新的数据行
21 drwUser = dstUser.Tables("tblUser").NewRow()
22 ' 在数据行的各列中输入值
23 drwUser("LoginName") = "NewUser1"
24 drwUser("FirstName") = "New"
25 drwUser("LastName") = "User"
26 ' 向 user 表添加数据行
27 dstUser.Tables("tblUser").Rows.Add(drwUser)
28 ' 检查数据集中是否有数据发生更改
29 If dstUser.HasChanges() Then
30     ' 将所有发生更改的行保存在新数据集中
31     dstChanges = dstUser.GetChanges()
32     ' 检查发生更改的行中是否包含错误
32     If dstChanges.HasErrors() Then
33         ' 在拒绝更改前显示所有行的行状态
34         For intCounter = 0 To dstUser.Tables(0).Rows.Count - 1
35             MsgBox("HasErrors=True, Before RejectChanges, RowState=" & _
36                 dstUser.Tables(0).Rows(intCounter).RowState.ToString & _
37                 ", LoginName=" & _
38                 dstUser.Tables(0).Rows(intCounter)("LoginName").ToString)
39         Next
40         ' 拒绝对数据集所做的更改
41         dstUser.RejectChanges()
42         ' 在拒绝更改后显示所有行的状态
43         For intCounter = 0 To dstUser.Tables(0).Rows.Count - 1
44             MsgBox("HasErrors=True, After RejectChanges, RowState=" & _
45                 dstUser.Tables(0).Rows(intCounter).RowState.ToString & _
46                 ", LoginName=" & _
47                 dstUser.Tables(0).Rows(intCounter)("LoginName").ToString)
48         Next
49     Else
50         ' 在接受更改前显示所有行的行状态
51         For intCounter = 0 To dstUser.Tables(0).Rows.Count - 1
52             MsgBox("HasErrors=False, Before AcceptChanges, RowState=" & _
```

```

53         dstUser.Tables(0).Rows(intCounter).RowState.ToString & _
54         ", LoginName=" & _
55         dstUser.Tables(0).Rows(intCounter)("LoginName").ToString)
56     Next
57     ' 接受对数据集所做的更改
58     dstUser.AcceptChanges()
59     ' 在接受更改后显示所有行的行状态
60     For intCounter = 0 To dstUser.Tables(0).Rows.Count - 1
61         MsgBox("HasErrors=False, After AcceptChanges, RowState=" & _
62             dstUser.Tables(0).Rows(intCounter).RowState.ToString & _
63             ", LoginName=" & _
64             dstUser.Tables(0).Rows(intCounter)("LoginName").ToString)
65     Next
66 End If
67 End If

```

程序清单 3B.12 显示了如何使用 **DataSet** 类的 **AcceptChanges** 方法和 **RejectChanges** 方法，还说明了如何检查已更改行的改动和错误。在本示例中，**HasErrors** 方法返回的是 **False**，说明更改被接受。你也可以适用地调整示例代码以便拒绝更改，并改为使用 **RejectChanges** 方法。

3B.2 使用 DataTable 类

DataTable 类用于处理包含在 **DataSet** 类的 **Tables** 集合中表的内容。**DataTable** 类是 **System.dat** 名称空间的一部分，**DataTable** 则是对某个表内数据的内存缓存。需要注意的是，与 **DataSet** 类相似，**DataTable** 类也不是子类。换句话说，**DataSet** 类可与任意类型的数据源共同工作。



不存在 **SqlDataTable** 或 **OleDbDataTable** 类。

3B.2.1 DataTable 属性

表 3B.4 按字母顺序显示了 **DataTable** 类的属性。请注意，这里仅列举了非继承的公共属性。

表 3B.4 **DataTable** 类属性

名称	说明
CaseSensitive	该属性指明在表中进行字符串比较时是否区分大小写。该属性设置或返回一个 Boolean 值。如果 DataTable 是数据集的一部分，则该属性的值便被赋予该数据集上的 CaseSensitive 属性。然而，如果程序化地创建了 DataTable ，则该属性被默认设置为 False
ChildRelations	ChildRelations 属性为数据表返回子关系的集合。该属性只读，其返回的数据类型是 DataRelationCollection 。如果不存在关系则返回 Nothing

续表

名称	说明
Columns	该属性返回组成数据表的列/字段的 DataColumnCollection 集合。该属性只读。如果不存在列则返回 Nothing
Constraints	Constraints 属性返回属于数据表的约束集合。该属性只读, 返回的数据类型是 ConstraintCollection 。如果不存在约束则返回 Nothing
DataSet	该属性只读, 返回表所属的数据集。这意味着返回的对象的数据类型为 DataSet
DefaultView	该属性只读, 返回 DataView 对象 (表的一个定制视图)。返回的数据视图可用于筛选、排序和搜索数据表。有关 DataView 类的更多信息, 请参阅本章后面的“使用 DataView 视图”一节
DisplayExpression	DisplayExpression 属性用于返回和设置表达式, 表达式返回的值描述了 UI 中的表。该属性可用于动态创建基于当前数据的文本
ExtendedProperties	该属性只读, 返回一个由定制用户信息组成的 PropertyCollection 集合。可使用该属性的 Add 方法向数据表中添加定制信息
HasErrors	该属性只读, 返回一个 Boolean 值, 指明在数据表的行中是否发生了错误
Locale	Locale 属性用于返回或设置 CultureInfo 对象。该对象提供的区域信息用于比较数据表的字符串。这意味着可以指定与数据表中包含的数据相匹配的区域, 从而确保特定的字符能够正确地排序, 字符串比较也根据区域规则来执行。默认情况下, 该属性被设置为数据集的 Locale 属性值。然而, 如果数据表不属于某数据集, 则该属性的值将被设置为当前系统的
MinimumCapacity	该属性返回或设置数据表的初始化尺寸。其数据类型是 Integer , 默认值为 25。该值指定的是在创建额外的资源以容纳更多的行之前, 数据表最初所能处理的行数。当性能问题非常关键时应该设置该属性, 因为它可以使在开始填充数据表之前更快地分配资源。因此当性能问题非常关键时, 可将该属性设置为适合所返回的行数的最小值
Namespace	Namespace 属性返回或设置名称空间, 用于将数据表中的数据表示为 XML
ParentRelations	ParentRelations 属性为数据表返回父级关系的集合, 作为 DataRelationCollection 对象, 该属性只读。如果不存在父级关系则返回 Nothing
Prefix	该属性可读写, 用来处理数据表名称空间的前缀。当数据表被表示为 XML 时会用到该名称空间前缀。该属性的数据类型为 String
PrimaryKey	PrimaryKey 属性可读写, 返回或设置 DataColumn 对象的数组, 这些对象是数据表的主键。如果试图设置已是外键的列, 则会抛出 DataException 异常
Rows	该只读属性返回 DataRowCollection 对象, 用来处理组成数据表中数据的所有 DataRow 对象。如果数据表中不存在行则返回 Nothing
TableName	该属性可读写, 用来处理数据表的名称。在数据集的 Tables 集合中查找表时会用到该名称属性。该属性的数据类型为 String

3B.2.2 DataTable 方法

表 3B.5 按字母顺序列出了 **DataTable** 类非继承的公共方法。

表 3B.5 **DataTable** 类方法

名称	说明	示例
AcceptChanges()	该方法接受或提交自从上次调用该方法或加载该数据表时起所做的所有更改。由于该方法更改的是数据表中所有发生更改的行的行状态，所以直到对该数据适配器调用 Update 方法后才会调用该方法。如果在更新数据集之前调用了 AcceptChanges 方法，那么由于发生更改的行的 RowState 属性将改为 Unchanged ，所以更改不会被写回到数据集，看上去好像仍未发生更改一样	<code>dtbUser.AcceptChanges()</code>
BeginInit()	BeginInit 方法用于指示数据表还未初始化。用该方法开始对那些用于表单或其他组件的数据表进行初始化。初始化发生在运行时并与 EndInit 方法一起执行，以确保数据表在完全初始化之前未被使用	<code>dtbUser.BeginInit()</code>
BeginLoadData()	BeginLoadData 方法与 EndLoadData 搭配使用。该方法在使用 LoadDataRow 方法加载数据时负责关闭索引维护、约束和通知	<code>dtbUser.BeginLoadData()</code>
Clear()	该方法从数据表中清空所有行。如果某行在其他表中有子行，而该数据表与当前数据表之间存在强制性关系，则抛出一个异常	<code>dtbUser.Clear()</code>
Clone()	Clone 方法用于克隆或复制数据表的数据结构。但不包含数据本身，仅包括模式、关系和约束。如果还需要复制数据，则需使用 Copy 方法	<code>dtbClone = dtbUser.Clone()</code>
Compute(ByVal vstrExpression As String, ByVal vstrFilter As String)	Compute 方法用于计算当前行中符合 vstrFilter 条件的 vstrExpression 。请注意， vstrExpression 表达式必须包含聚合函数 (aggregate function)，如 Sum 或 Count	<code>objCompute = dtbUser.Compute("COUNT (FirstName)", "LastName IS NOT NULL")</code>
Copy()	Copy 方法用于克隆数据表的数据结构，并将数据从当前数据表中复制到新数据表中。如果仅需克隆数据结构，则应该使用 Clone 方法	<code>dtbCopy = dtbUser.Copy()</code>
EndInit()	EndInit 方法用于与 BeginInit 方法搭配使用。如果调用该方法，则说明数据集已被完全初始化	<code>dtbUser.EndInit()</code>
EndLoadData()	EndLoadData 方法在 BeginLoadData 方法关闭索引维护、约束和通知后予以返回。在使用 LoadDataRow 方法加载数据时，要使用 EndLoadData 和 BeginLoadData 方法	<code>dtbUser.EndLoadData()</code>

续表

名称	说明	示例
GetChanges()	该重载方法用于获取数据表的副本, 该副本包含自上次调用 AcceptChanges 方法和加载该数据表时起对数据表所做的所有更改。使用该方法可以不带任何参数, 也可以通过指定 DataRowState 枚举的一个成员来指明要在数据表中返回何种更改	<code>dtbChanges = dtbUser. GetChanges() or dtbAdded = dtbUser. GetChanges(DataRowSt ate.Added)</code>
GetErrors()	GetErrors 方法返回 DataRow 对象的数组。该数组包含数据表中所有包含错误的行	<code>arrdrwErrors = dtbUser.GetErrors()</code>
ImportRow(ByVal vdrwImport As DataRow)	ImportRow 方法将 DataRow 对象复制到 DataTable 中。该副本包括原始的和当前的值、错误以及 DataRowState 值。简单地说, DataRow 对象中的一切内容均被复制	<code>dtbUser.ImportRow(d rwImport)</code>
LoadDataRow(ByVal varobjValues() As Object, ByVal vbInAcceptChanges As Boolean)	LoadDataRow 方法使用 varobjValues 数组中的值查找指定的行。该数组中此值用于匹配主键列。如果找到匹配的行, 则更新该值。否则使用 varobjValues 值创建新行	<code>drwLoad = dtbUser. LoadDataRow(arobjV alues, False)</code>
NewRow()	NewRow 方法用于创建具有与数据表相同模式的新 DataRow 对象	<code>drwNew = dtbUser. NewRow()</code>
RejectChanges()	该方法拒绝或回滚自从上次调用 AcceptChanges 方法或加载该 DataTable 时起对该数据表所做的更改	<code>dtbUser.RejectChang es()</code>
Select()	Select 方法是重载方法, 用于检索 DataRow 对象的数组。返回的数据行按主键排序。如果没有主键, 则按照各行添加到数据表中的顺序进行排序。实际上, 如果使用的重载版本不将排序规则作为参数, 那么将按各行的添加顺序进行排序	<code>arrdrwAllDataRows = dtbUser.Select() or arrdrwFirstNameUnso rtedDataRows = dtb User.Select("FirstN ame = 'John'") 或 arrdrwFirstName SortedDataRows = dtbUser.Select ("FirstName = 'John'", "LastName ASC") 或 arrdrwFirst Name SortedOriginalDataR ows = dtbUser.Select ("FirstName = 'John'", "LastName ASC", Data View Row State. OriginalRows)</code>

3B.2.3 声明并实例化 DataTable

有多种方法可以实例化 **DataTable** 对象。可以使用重载的类构造函数或者引用数据集 **Tables** 集合中指定的表。以下是在声明 **DataTable** 时进行实例化的方法:

```
Dim dtbNoArguments As New DataTable()
Dim dtbTableNameArgument As New DataTable("TableName")
```

必要时，也可以对其进行声明和实例化，如下所示：

```
Dim dtbNoArguments As DataTable
Dim dtbTableNameArgument As DataTable

dtbNoArguments = New DataTable()
dtbTableNameArgument = New DataTable("TableName")
```

我在这里使用了两种不同的构造函数，一个不带参数，另一个则将表名作为惟一的参数。你也可以先声明 **DataTable** 对象，然后再令该对象引用填充过的数据集中的 **Tables** 集合里的表，如下所示：

```
Dim dtbUser As DataTable
dtbUser = dstUser.Tables("tblUser")
```

3B.2.4 构建自己的 DataTable

有时需要存储具有类似表结构的临时数据，这需要几组具有相同结构的数据序列。由于类似表的结构，因而可以选择 **DataTable** 作为其存储形式，不过还有其他形式。程序清单 3B.13 阐述了如何从类似 UserMan 数据库的 **tblUser** 表中从头创建数据结构。

程序清单 3B.13 构建自己的 DataTable

```
1 Dim dtbUser As DataTable
2 Dim drwUser As DataRow
3 Dim dclUser As DataColumn
4 Dim arrdclPrimaryKey(0) As DataColumn
5
6 dtbUser = New DataTable("tblUser")
7
8 ' 创建表结构
9 dclUser = New DataColumn()
10 dclUser.ColumnName = "Id"
11 dclUser.DataType = Type.GetType("System.Int32")
12 dclUser.AutoIncrement = True
13 dclUser.AutoIncrementSeed = 1
14 dclUser.AutoIncrementStep = 1
15 dclUser.AllowDBNull = False
16 ' 向数据表结构中添加列
17 dtbUser.Columns.Add(dclUser)
18 ' 向 PK 数组中添加列
19 arrdclPrimaryKey(0) = dclUser
20 ' 设置主键
21 dtbUser.PrimaryKey = arrdclPrimaryKey
```

```
22
23 dclUser = New DataColumn()
24 dclUser.ColumnName = "ADName"
25 dclUser.DataType = Type.GetType("System.String")
26 ' 向数据表结构中添加列
27 dtbUser.Columns.Add(dclUser)
28
29 dclUser = New DataColumn()
30 dclUser.ColumnName = "ADSID"
31 dclUser.DataType = Type.GetType("System.Guid")
32 ' 向数据表结构中添加列
33 dtbUser.Columns.Add(dclUser)
34
35 dclUser = New DataColumn()
36 dclUser.ColumnName = "FirstName"
37 dclUser.DataType = Type.GetType("System.String")
38 ' 向数据表结构中添加列
39 dtbUser.Columns.Add(dclUser)
40
41 dclUser = New DataColumn()
42 dclUser.ColumnName = "LastName"
43 dclUser.DataType = Type.GetType("System.String")
44 ' 向数据表结构中添加列
45 dtbUser.Columns.Add(dclUser)
46
47 dclUser = New DataColumn()
48 dclUser.ColumnName = "LoginName"
49 dclUser.DataType = Type.GetType("System.String")
50 dclUser.AllowDBNull = False
51 dclUser.Unique = True
52 ' 向数据表结构中添加列
53 dtbUser.Columns.Add(dclUser)
54 dclUser = New DataColumn()
55 dclUser.ColumnName = "Password"
56 dclUser.DataType = Type.GetType("System.String")
57 dclUser.AllowDBNull = False
58 ' 向数据表结构中添加列
59 dtbUser.Columns.Add(dclUser)
```

程序清单 3B.13 中的示例代码使用了 **DataTable** 类、**DataColumn** 类以及 **DataRow** 类。**DataRow** 类将在本章后面予以讨论。可以通过使用 **DataTable** 的 **Rows** 集合中的 **Add** 方法和/或向 **DataSet** 中添加数据表，程序清单中所创建的数据表可用来存储数据。

3B.2.5 填充 DataTable

有多种方法可以填充 **DataTable**。可以通过创建 **DataRow** 对象来手动向数据表中添加行以及设置列值，然后再使用 **Rows** 属性的 **Add** 方法将其添加到数据表中。程序清单 3B.13 中

的示例演示了如何创建自己的 **DataTable**。

此外，还可以用 **DataAdapter** 类的 **Fill** 方法来实现这一点，在下一节的程序清单 3B.14 中给出了相关范例。

3B.2.6 从 DataTable 中清除数据

在向数据表或数据表的结构元素中添加关系、约束等元素时，经常需要一种能从数据表中清除所有数据的方法。通过使用 **Clear** 方法即可完成此操作，如程序清单 3B.14 所示。

程序清单 3B.14 从 DataTable 中清除数据

```

1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dtbUser As DataTable
5
6 ' 实例化并打开连接
7 cnnUserMan = New SqlConnection("Data Source=USERMANFC;" & _
8   "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
9 cnnUserMan.Open()
10 ' 实例化命令
11 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
12 ' 实例化并初始化数据适配器
13 dadUser = New SqlDataAdapter()
14 dadUser.SelectCommand = cmdUser
15 ' 实例化数据表
16 dtbUser = New DataTable("tblUser")
17 ' 填充数据表
18 dadUser.Fill(dtbUser)
19 ' 输入你的资料
20 ...
21 ' 从数据表中清除数据
22 dtbUser.Clear()

```

3B.2.7 复制 DataTable

有时需要复制数据表。如果是出于测试目的而对某些数据进行处理，则为了保持原始数据的完整性，可对数据的副本执行操作。根据你的实际需要，有两种方法可以做到这一点：可以仅复制数据结构（与数据集类似），也可以复制数据结构及其数据。有关这些技巧，请参阅以下两小节中的示例代码。

复制 DataTable 的数据结构

如果需要克隆数据结构，则可使用 **Clone** 方法。有关克隆数据结构的示例，请查看程序清单 3B.15。

程序清单 3B.15 从 DataTable 中克隆数据结构

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dtbUser As DataTable
5 Dim dtbClone As DataTable
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9     "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据表
12 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dtbUser = New DataTable()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmdUser
17 ' 填充数据表
18 dadUser.Fill(dtbUser)
19 ' 克隆数据表
20 dtbClone = dtbUser.Clone()
```

复制 DataTable 的数据和结构

当需要完整地复制数据表中的数据结构及其包含的数据时, 可使用 **Copy** 方法, 如程序清单 3B.16 中所示。

程序清单 3B.16 从 DataTable 中复制数据结构和数据

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dtbUser As DataTable
5 Dim dtbCopy As DataTable
6
7 ' 实例化并打开连接
8 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
9     "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
10 cnnUserMan.Open()
11 ' 实例化命令和数据表
12 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
13 dtbUser = New DataTable()
14 ' 实例化并初始化数据适配器
15 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
16 dadUser.SelectCommand = cmdUser
17 ' 填充数据表
18 dadUser.Fill(dtbUser)
19 ' 复制数据表
```

```
20 dtbCopy = dtbUser.Copy()
```

3B.2.8 搜索 DataTable 并检索经筛选的数据视图

虽然 **DataTable** 类中没有用于查找指定行的方法，但可以借助 **DefaultView** 属性做到这一点。该属性（或者应该说是类，因为该属性也返回或设置 **DataView** 对象）具有 **RowFilter** 属性，**RowFilter** 属性与 ADO **Recordset** 对象的 **Filter** 属性作用相似。有关在 **tblUser** 表中筛选姓为 **Doe** 的所有用户的示例，请查看程序清单 3B.17。

程序清单 3B.17 搜索 **DataTable** 类

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmdUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dtbUser As DataTable
5 Dim intCounter As Integer
6 ' 实例化并打开连接
7 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
8 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
9 cnnUserMan.Open()
10 ' 实例化命令和数据表
11 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
12 dtbUser = New DataTable()
13 ' 实例化并初始化数据适配器
14 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
15 dadUser.SelectCommand = cmdUser
16 ' 填充数据表
17 dadUser.Fill(dtbUser)
18 ' 筛选数据表视图
19 dtbUser.DefaultView.RowFilter = "LastName = 'Doe'"
20
21 ' 对数据表视图中的所有行进行循环
22 For intCounter = 0 To dtbUser.DefaultView.Count - 1
23     MsgBox(dtbUser.DefaultView(0).Row("LastName").ToString())
24 Next
```

请注意，在像上述程序清单中那样对数据表进行筛选时，数据表中可见行的数量不会改变。如果在筛选前后检查数据表（使用 **dtbUser.Rows.Count**）的行数，则会发现二者完全相同。

虽然在前面曾提到使用 **DataTable** 类本身的任何方法或属性都不能进行搜索，但这并不完全正确。你可以使用 **DataTable** 类的 **Select** 方法，但是该方法仅适用于在单独的 **DataRow** 对象数组中检索指定的行并对其进行处理。因此，倘若要像程序清单 3B.17 中那样检索所有姓为 **Doe** 的用户，又希望这些行在 **DataRow** 对象的数组中，就只有使用以下方法：

```
Dim arrdrwFilter() As DataRow
arrdrwFilter = dtbUser.Select("LastName = 'Doe'")
```

请注意，**Select** 方法还有其他用途，例如检索所有未发生更改的行。你可以指定想要返回的行的 **RowState** 属性。请在前面的表 3B.5 中查看 **Select** 方法。

3B.3 使用 DataView 类

当数据库中有多个视图时则使用 **DataView** 类。**DataView** 类是 **System.Data** 名称空间的一部分。如前所述，该类可在默认的视图之外创建数据表的其他视图。使用该类可以筛选、排序、搜索和导航甚至编辑数据表中的行。

与 **DataSet** 和 **DataTable** 类相似，**DataView** 不是子类，或者更准确地说，**DataView** 可以与任意类型的数据源一起操作。



不存在 **SqlDataView** 或 **OleDbDataView** 类！

3B.3.1 DataView 属性

表 3B.6 按字母顺序显示了 **DataView** 类的属性。请注意，这里仅包含了其非继承的公共属性。

表 3B.6 DataView 类属性

名称	说明
AllowDelete	AllowDelete 属性返回或设置一个 Boolean 值，指明是否允许数据视图中的删除操作
AllowEdit	AllowEdit 属性返回或设置一个 Boolean 值，指明是否允许在数据视图中编辑行
AllowNew	AllowNew 属性返回或设置一个 Boolean 值，指明是否允许使用 AddNew 方法向数据视图添加新行
ApplyDefaultSort	该属性返回或设置一个 Boolean 值，指明是否使用默认的排序方式
Count	该属性只读，返回视图中可见行的行数。这里所说的可见行是指不受 RowFilter 和 RowStateFilter 属性设置影响的行
DataViewManager	DataViewManager 属性只读，返回 DataViewManager ，它与此数据视图甚或拥有此数据集的数据视图管理器相关联，并因此创建了此数据视图。如果不存在 DataViewManager ，则返回 Nothing
Item	该属性只读，用于指定想要从数据表中返回的行的索引。实际上，并不需要指定 Item 属性，仅使用括号即可。举例来说， dvwUser(0) 和 dvwUser(0) 都会检索索引为 0 的行
RowFilter	RowFilter 属性是一个 String 属性，该属性检索或设置用来在 DataView 中筛选可见行的表达式。有关如何使用该属性的示例，请参阅程序清单 3B.17
RowStateFilter	该属性检索或设置用于 DataView 的行状态筛选器。这意味着可以根据行状态（例如 Unchanged 、 Added 或 Deleted ）对行进行筛选。该值必须是 DataViewRowState 枚举的一个成员

续表

名称	说明
Sort	Sort 属性检索或设置表的列排序和排序规则。 Sort 属性的数据类型是 String 。可以指定用逗号分隔列，后接排序规则（ASC 或 DESC 分别表示升序或降序），如下所示： dvwUser.Sort = "LastName, FirstName ASC"
Table	该属性返回或设置源 DataTable ，此处指的是为数据视图提供数据的数据表。仅在当前值为 Nothing 时设置该属性。

3B.3.2 DataView 方法

表 3B.7 按升序列出了 **DataView** 类中的所有非继承的公共方法。

表 3B.7 DataView 类方法

名称	说明	示例
AddNew()	AddNew 方法向 DataView 中添加一个新行。返回值的数据类型是 DataRowView	drvNew = dvwUser.AddNew()
BeginInit()	BeginInit 方法用于指明数据视图仍未初始化，该方法不可被覆盖。该方法开始对数据视图进行初始化，该视图用于表单或由其他组件使用。初始化发生在运行时，并与 EndInit 方法一起使用，以确保数据视图在完全初始化之前未被使用	dvwUser.BeginInit()
Delete(ByVal vintIndex As Integer)	该方法用于删除符合指定索引 vintIndex 的行。如果在删除某行后感到后悔，则可以调用 DataTable 的 RejectChanges 方法来撤消删除。使用 Find 方法可以对指定行进行定位	dvwUser.Delete(5)
EndInit()	EndInit 方法用于与 BeginInit 方法搭配使用。调用该方法则表明数据视图已被完全初始化	dvwUser.EndInit()
Find()	该重载方法用于通过查找一个或更多主键值来对 DataView 中的行进行定位	intIndex = dvwUser.Find(objValue) or intIndex = dvwUser. Find(arrobjValues)
GetEnumerator()	该方法返回用于在列表中导航的枚举数，该方法不能被覆盖	enmDataView = dvwUser.GetEnumerator()

3B.3.3 声明并实例化 DataView

有多种方法可以实例化 **DataView** 对象，可以使用重载的类构造函数或者引用 **DataTable** 对象的 **DefaultView** 属性完成。下面讲述了如何在声明时实例化数据视图：

```
Dim dvwNoArguments As New DataView()
```

```
Dim dvwTableArgument As New DataView(dstUser.Tables("tblUser"))
```

必要时, 也可以先声明然后再实例化数据视图, 如下所示:

```
Dim dvwNoArguments As DataView
Dim dvwTableArgument As DataView

dvwNoArguments = New DataView()
dvwTableArgument = New DataView(dstUser.Tables("tblUser"))
```

在这里使用了两种不同的构造函数, 一种不带任何参数, 另一种则仅将数据表作为唯一的参数。你也可以首先声明 **DataView** 对象, 然后再让该对象引用 **DataTable** 对象的 **DefaultView** 属性, 如下所示:

```
Dim dvwUser As DataView
dvwUser = dstUser.DefaultView()
```

3B.3.4 搜索 DataView

可以使用 **Find** 方法查找指定的行。该方法是重载方法, 以对象或对象的数组作为其唯一的参数。有关在 UserMan 数据库的 tblUser 表中查找 ID 为 1 的用户的示例代码, 请参阅程序清单 3B.18。

程序清单 3B.18 搜索 DataView 类

```
1 Dim cnnUserMan As SqlConnection
2 Dim cmmUser As SqlCommand
3 Dim dadUser As SqlDataAdapter
4 Dim dtbUser As DataTable
5 Dim dvwUser As DataView
6 Dim objPKValue As Object
7 Dim intIndex As Integer
8
9 ' 实例化并打开连接
10 cnnUserMan = New SqlConnection("Data Source=USERMANPC;" & _
11 "User ID=UserMan;Password=userman;Initial Catalog=UserMan")
12 cnnUserMan.Open()
13 ' 实例化命令和数据表
14 cmmUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
15 dtbUser = New DataTable()
16 ' 实例化并初始化数据适配器
17 dadUser = New SqlDataAdapter("SELECT * FROM tblUser", cnnUserMan)
18 dadUser.SelectCommand = cmmUser
19 ' 填充数据表
20 dadUser.Fill(dtbUser)
21 ' 筛选数据表视图
22 dtbUser.DefaultView.RowFilter = "LastName = 'Doe'"
23 ' 创建新的数据视图
24 dvwUser = dtbUser.DefaultView
```

```

25 ' 指定排序规则
26 dvwUser.Sort = "Id ASC"
27 ' 查找 Id 为 1 的用户
28 objPKValue = 1
29 intIndex = dvwUser.Find(objPKValue)

```

即使可以在对象数组的对象上指定多个值，数组中的值也只是试图与 **DataView** 中的主键列相匹配。如果数据视图只有一个主键列，那么指定多个值并无必要。

3B.4 使用 DataRow 类

DataRow 类用于描述 **DataTable** 类中单独的行。可以仅使用 **DataRow** 对象本身，但其唯一的实际用途便是与 **DataTable** 类搭配使用。**DataRow** 类与 **DataColumn** 类一起组成了 **DataTable** 类的主要构造块。

这很有意义，因为数据库中的表包含了行和列。那么为什么要使这种对象模型呢？

现在应考虑该类的属性和方法了。表 3B.8 显示了 **DataRow** 类中的所有非继承公共属性。

表 3B.8 DataRow 类属性

属性名	描述
HasErrors	HasErrors 属性返回一个 Boolean 值，指明列集合中是否包含错误。该属性只读
Item	该重载属性返回或设置存储在指定列中的数据。该属性是 DataRow 类的默认属性，这也就是访问指定列时无需使用该属性的原因。直接使用括号就足够了。这说明 <code>drwUser.Item(2)</code> 和 <code>drwUser(2)</code> 作用相同
ItemArray	ItemArray 为 DataRow 对象的所有列返回或设置值，由对象数组实现
RowError	该属性为 DataRow 对象返回或设置定制的错误描述。可以使用该属性指明特定的行中是否有错误
RowState	RowState 属性只读，返回 DataRow 的当前状态。返回的值是 DataRowState 枚举中的一个值。行状态供 GetChanges 和 HasChanges 方法使用
Table	该属性只读，返回该行所属的 DataTable

表 3B.9 显示了 **DataRow** 类中的所有非继承公共方法。

表 3B.9 DataRow 类的方法

方法名	说明	示例
AcceptChanges()	该方法提交自从上次调用 AcceptChanges 或加载该行时起对 DataRow 所做的更改	<code>drwUser.AcceptChanges()</code>
BeginEdit()	BeginEdit 方法用于开始对 DataRow 的编辑操作。当该行处于编辑模式时，所有事件均被禁用。这意味着可以更改该行的内容，而不需要任何激活事件或检验规则触发器。该方法应与 EndEdit 方法和/或 CancelEdit 方法搭配使用	<code>drwUser.BeginEdit()</code>

续表

方法名	说明	示例
CancelEdit()	该方法用于取消对当前行的编辑。需与 BeginEdit 方法搭配使用	<code>drwUser.CancelEdit()</code>
ClearErrors()	该方法用于清除 DataRow 中的所有错误。其中包括 RowError 属性和使用 SetColumnError 方法设置的错误	<code>drwUser.ClearErrors()</code>
Delete()	该方法用于删除 DataRow 。实际上, 这并不完全正确。如果该行的 RowState 是 Added , 则该行会被删除, 但是如果不是 Added , 则该行的 RowState 会被更改为 Deleted 。这意味着直到针对该数据集调用了 AcceptChanges 方法或 Update 方法之后, 该行才会被真正删除。这也说明可以调用 RejectChanges 方法来撤消删除。如果试图删除已标记为 Deleted 的行, 则会抛出一个 DeletedRowInaccessibleException 异常	<code>drwUser.Delete()</code>
EndEdit()	该方法用于结束当前的编辑, 应与 BeginEdit 方法搭配使用	<code>drwUser.EndEdit()</code>
GetChildRows()	该重载方法用于检索 DataRow 的子行。使用 DataRelation 对象、 DataRelation 对象的名称、 DataRelation 对象与 DataRowVersion 对象或是 DataRelation 对象的名称与 DataRowVersion 对象就可以实现这点	<pre> arrdrwChildRows = drwUser.GetChildRows (drlUser) 或 arrdrwChildRows = drwUser.GetChildRows (strRelationName) 或 arrdrwChildRows = drwUser.GetChildRows (drlUser, drvUser) 或 arrdrwChildRows = drwUser.GetChildRows (strRelationName, drvUser) </pre>
GetColumnError()	该重载方法使用 DataColumn 对象、 Integer 或 String 返回指定列的错误说明	<pre> strError = drwUser.GetColumnErr or(dtcName) or strError = drwUser. GetColumnError(intCo lumn) or strError = drwUser.GetColumnErr or(strColumn) </pre>
GetColumnsInError()	GetColumnsInError 方法用于返回有错误的 DataColumn 对象的数组。应该在调用该方法之前使用 DataRow 对象的 HasErrors 方法来判断该行中是否存在错误	<pre> arrdtcError = drwUser.GetColumnsIn Error() </pre>

续表

方法名	说明	示例
GetParentRow()	该重载方法用于检索 DataRow 的父行。使用 DataRelation 对象、 DataRelation 对象的名称、 DataRelation 对象与 DataRowVersion 对象或者 DataRelation 对象的名称与 DataRowVersion 对象就可以实现这点	<pre>drwParent = drwUser.GetParentRow (drlUser) 或 drwParent = drwUser.GetParentRow (strRelationName) 或 drwParent = drwUser.GetParentRow (drlUser, drvUser) 或 drwParent = drwUser.GetParentRow (strRelationName, drvUser)</pre>
GetParentRows()	该重载方法用于检索 DataRow 的多个父行。使用 DataRelation 对象、 DataRelation 对象的名称、 DataRelation 对象与 DataRowVersion 对象或者 DataRelation 对象的名称与 DataRowVersion 对象就可以实现这点	<pre>arrdrwParent = drwUser.GetParentRow s(drlUser) 或 arrdrwParent = drwUser.GetParentRow s(strRelationName) 或 arrdrwParent = drwUser.GetParentRow s(drlUser, drvUser) 或 arrdrwParent = drwUser.GetParentRow s(strRelationName, drvUser)</pre>
HasVersion(ByVal vdrvVersion As DataRowVersion)	该方法返回 Boolean 值，指明指定的版本 (vdrvVersion) 是否存在。vdrvVersion 参数必须是 DataRowVersion 枚举的一个成员	<pre>blnExist = drwUser.HasVersion(D ataRowVersion.Default t)</pre>
IsNull()	该重载方法返回 Boolean 值，指明指定的列中是否包含空值 (Nothing)	<pre>blnNull = drwUser.IsNull(dtcNu ll) 或 blnNull = drwUser.IsNull(intCo lumn) 或 blnNull = drwUser.IsNull(strCo lumn) 或 blnNull = drwUser.IsNull(dtcNu ll, drvVersion)</pre>
RejectChanges()	拒绝自从上次调用 AcceptChanges 或从加载 DataRow 时起对该行所做的全部更改	<pre>drwUser.RejectChanges()</pre>

续表

方法名	说明	示例
SetColumnError()	SetColumnError 方法是重载方法，用于为指定列设置错误说明	drwUser.SetColumnError(dtcError, strError) 或 drwUser.SetColumnError(intColumn, strError) 或 drwUser.SetColumnError(strColumn, strError)
SetParentRow()	该重载方法用于设置 DataRow 的父行。使用 DataRow 对象或同时使用 DataRow 对象与 DataRelation 对象都可以实现这点	drwUser.SetParentRow(drwParent) or drwUser.GetParentRow(drwParent, drlParent)
SetUnspecified(dtcUnspecified)	该方法将 dtcUnspecified DataColumn 的值设置为 unspecified	drwUser.SetUnspecified(dtcUnspecified)

3B.4.1 声明并实例化 DataRow

只有一种方法可以实例化 **DataRow** 对象。**DataRow** 对象没有构造函数，因而需要借助 **DataTable**。**DataTable** 的 **NewRow** 方法正是所需的方法：

```
Dim drwUser As DataRow
drwUser = dtbUser.NewRow()
```

3B.4.2 构建自己的 DataRow

在程序清单 3B.13 中可以了解到如何创建自己的 **DataRow**。该清单还显示了如何创建类似于 UserMan 数据库中 tblUser 表的 **DataTable**。

3B.5 指针

指针是为使用关系型数据库而设计的，它们仅仅是指向结果集中指定行的指示器。在 ADO 和 ADO.NET 中，该结果集通常被描述为类似于 ADO **Recordset** 对象、ADO.NET **DataReader** 类或 **DataTable** 类的数据库类。ADO 中有许多方法可以处理和使用指针，而在 ADO.NET 中，可选的方法却受到很多限制。这是由 ADO.NET 的非连接结构及其数据源的独立性所决定的，但在其后续版本中应该会有所改进。

3B.5.1 指针类型

指针有许多类型。下面简单讲述了在 ADO 和 ADO.NET 中用到的几种，列举如下：

- 只进指针

- 静态指针
- 动态指针
- 键集指针

只进指针 (forward-only cursor)

该指针类型只需要最小的开销。正如其名称所描述的那样，使用这种指针在结果集中一次只能前进一行。但这并不完全正确，可以关闭处理结果集的数据类，然后再将其打开，指针就会移到第一行或 BOF（文件的开头）处，位于第一行之前（参阅图 3B.1）。

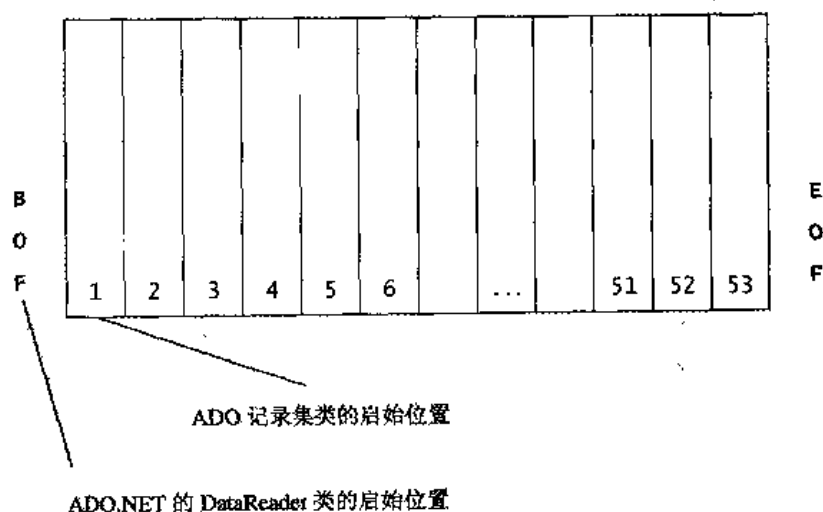


图 3B.1 BOF（文件的开头）

只进指针默认为动态的，也就是说访问时行是从数据源中读取该行，因此在生成结果集后已提交到该行的更改将会生效。但在 ADO.NET **DataReader** 类中并非如此。对打开的 **DataReader** 对象的基础行所做的更改不会生效。

该指针类型是 ADO.NET **DataReader** 类使用的惟一类型，也是 ADO **Recordset** 类使用的四种指针类型之一。

静态指针 (static cursor)

静态指针如其名称所述是静态的。这意味着结果集的内容是静态的，或者打开结果集后其内容不会发生更改。打开结果集之后就无法检测对其所做的更改，尽管由静态指针自身所做的更改通常会被检测出来并反映到结果集中（这取决于具体的实现过程）。

与只进指针不同，静态指针既能向前滚动也能向后滚动。该指针是 ADO **Recordset** 类使用的四种指针类型之一。

动态指针 (dynamic cursor)

动态指针非常适合处理并发问题，因为它可以检测到打开指针后对结果集所做的更改。

动态指针可以向前和向后滚动，由于它是这里提到的四种指针中最耗资源的一种，所以能使用其他类型的指针时就不要使用动态指针。

该指针是 ADO **Recordset** 类使用的四种指针类型之一。

键集指针(Keyset cursor)

键集指针可被视作静态指针和动态指针的混合物，因为键集指针同时具备静态指针和动态指针的特有功能。它可以检测对数据列值的更改。由指针所做的插入将附加到结果集上，然而，如果关闭该指针之后又打开它，那么由其他指针所做的插入将只是可见状态。在有键集指针时的删除情况则正好相反。由其他指针所做的删除将会被检测到，但由键集指针本身所做的删除却无法检测。

该指针是 ADO **Recordset** 类使用的四种指针类型之一。

3B.5.2 指针位置

根据所处位置的不同，指针可分为两种：客户端指针和服务器端指针。正如其名称所描述的那样，它们指的是对客户端和服务端上数据的指示器。这两种类型各有千秋，下面我们将分别予以讨论。

客户端指针

客户端指针（或称本地指针）用于在本地或客户机的结果集中执行导航。这意味着在查询中选择的所有行都必须被传输到客户端。这种传输非常耗费资源，具体取决于行数、网络流量和客户机的存储空间。存储空间可以是内存，也可以是磁盘空间，这取决于指针类型和/或客户机上的可用空间。

服务器端指针

服务器端指针（或称之为远程指针）用于在位于远程机器或服务器上的结果集中执行导航。ADO.NET 当前并没有提供对使用服务器端指针的内在支持，因此只能选择 ADO。其原因可能是由于各种数据源的实现方式大相径庭。这意味着要隐去处理多种不同数据源的复杂性时，公开服务器端指针（以及其他服务器端的功能）的操作会非常困难。我们所希望的是无论访问何种数据源，对服务器端指针来说，其行为和操作均完全相同。

相信在今后的版本中还会增加一些服务器端的功能。前面已提醒读者注意，**DataReader** 类使用的是服务器端指针。务必记住，服务器端指针并不能实际控制 **DataReader** 类，而只能一次前进一行，因此期望得到的服务器端指针的功能实际上并不具备。

ADO.NET（除 **DataReader** 以外）仅使用客户端指针，然而你可以指定 ADO 首选的指针位置。简单地说，如果需要服务器端指针的功能，那么 ADO 是你当前的惟一选择。实际上，如果要寻求服务器端的处理方式，请参考在第 6 章所介绍的有关存储过程和触发器的使用。

3B.6 COM Interop

由于 ADO 以前的版本均以 COM 组件构建，因而为了访问 COM 组件，还需要使用 COM

Interop。为使用 COM，COM Interop 提供了对 COM 组件的接口，同时也可借此通过 COM 组件访问 COM Interop。在本节中我们将只讨论 COM-to-.NET 的使用方法。

无论何时访问 COM 组件，均需使用 COM Interop。关于 COM Interop 的内容本节无法尽述，如果需要更多信息，请参考 Dan Appleman 编写的《Moving to Visual Basic.NET: Strategies, Concepts and Code》，作者 Dan Appleman，ISBN 1893115976，Apress 出版社 2001 年出版。

有多种在 .NET 框架中使用 COM 组件的方法，但各种方法都是将 COM 组件公开给管理代码。CLR 期望所有类型在汇编过程中均被定义为元数据，这也适用于 COM 类型。因此现在的工作是将 COM 类型转换成元数据，或者为 COM 类型生成元数据。

下面将介绍生成元数据的两种简单方法，建议你针对以前版本的 ADO 选择其中之一。第一种方法需要针对该 COM 组件在命令行中运行类型库导入程序 **TlbImp.exe**，以便在汇编中生成元数据。请注意，该组件必须包含一个类型库。如果不包含，那么很可能有一个单独的类型库文件用于导入 (*.tlb)。任何写管理代码的客户都可使用该汇编结果。假设你想导入 ADO 2.7 库（通常位于 Program Files\Common Files\System\Ado 文件夹下），需要打开命令提示符，进入 ADO 文件夹并执行以下命令：**TlbImp msado27.tlb**。输出是 ADODB.DLL 汇编，该汇编处理 ADODB 名称空间，相信你应对此比较熟悉。添加对工程的引用可以引用该汇编。有关生成元数据的信息请参考生成元数据的方法。如果需要查看所有可用的命令选项，则只需从命令行中运行不带参数的 TlbImp 命令即可。系统将显示所有选项。

第二种生成元数据的方法需要在工程中添加一个对 COM 组件的引用，如图 3B.2 所示。这种方法中，IDE 会生成元数据。可以通过单击 [Project] 菜单上的 [Add Reference] 命令来访问 [Add Reference] 对话框。

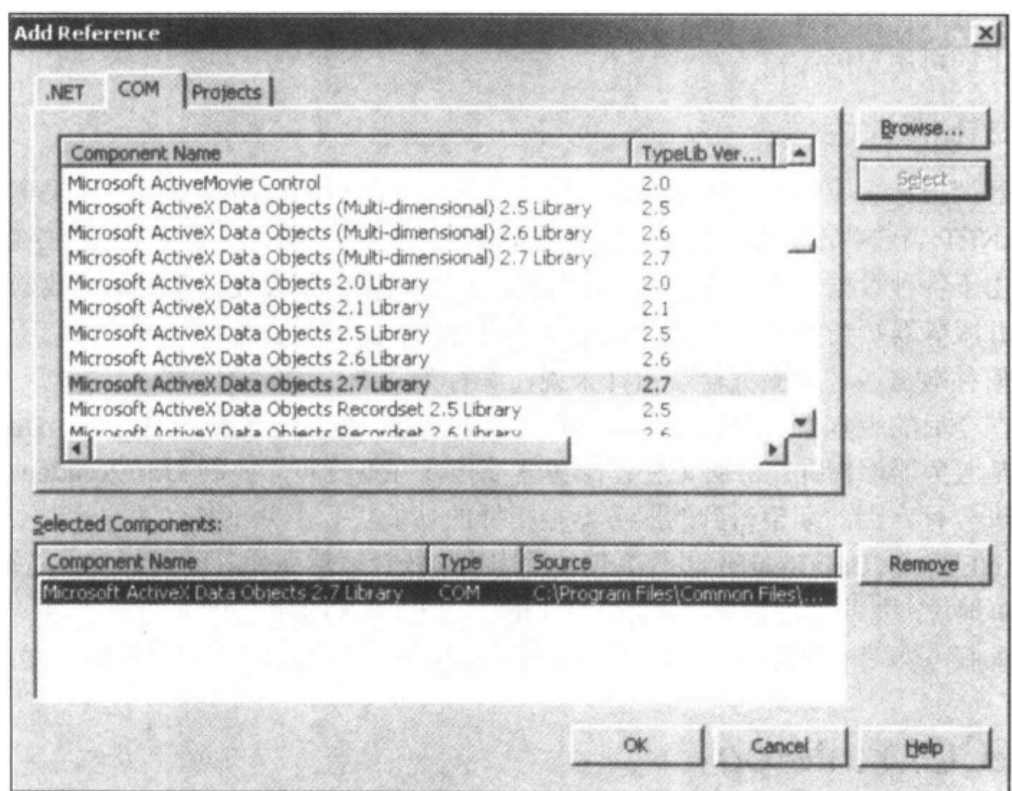


图 3B.2 [Add Reference] 对话框

可以使用上述的任一方法来访问所有 COM 类型，而所有 COM 组件仍作为应用程序的一部分。请记住，将这些组件全部移植到 .NET 类是明智之举，因为使用 COM Interop 的开销很大。一旦在工程中添加了对 COM 组件的引用，就可以使用对象浏览器来查看所有类型。

虽然建议你执行所有操作，但实际上并没有必要这样做。之所以说没有必要是因为 Microsoft 已决定将一些 Primary Interop Assemblies 集成到 VS.NET 中，其中包括了 ADO。这意味着已经存在针对 ADO COM 库的 .NET 框架包装。因此你所需做的只是添加对包装的引用。可以通过单击 [Project] 菜单上的 [Add Reference] 命令来访问 [Add Reference] 对话框。在 [Add Reference] 对话框中的 .NET 选项卡上可以找到组件名为 adodb 的包装。从列表中选择该包装，然后单击 [Select]，最后单击 [OK]。

3B.7 小结

本章与上一章一起介绍了两种不同的数据访问技术：ADO 和 ADO.NET。ADO.NET 用于构建分布式多层 Web 应用程序，而 ADO 是在异构网络环境下（在这种环境下，你时时刻刻都位于 LAN 链接之上）进行传统 Windows 客户机/服务器多层应用程序编程的最佳工具。本章介绍了这种针对 ADO.NET 非连接层的对象模型，并适当地将其与 ADO 对象模型进行了比较。此外还介绍了 ADO.NET 非连接层并在一定程度上涉及了 ADO，本章是对该数据访问技术的各种类、方法和属性的参考。

除上面提到的以外，本章还讲述了以下内容：

- 讨论了 **DataSet**、**DataTable**、**DataRow** 和 **DataRow** 数据类。对 ADO.NET 的一些类与 ADO 中的对应类进行了比较，并对何时应该使用何种数据类提出了一些建议。
- 本章还讨论了指针和 COM Interop，后者是使用 ADO 2.7 或更早版本的必要条件。下一章将介绍 IDE 与数据相关的各种功能，例如如何创建数据库工程。

第 4 章

以数据库观点介绍 IDE

与数据库相关的不同设计器和工程介绍

本章将介绍 IDE (Integrated Development Environment, 集成开发环境) (尤其是 IDE 与数据库的联系), 并讨论创建数据库工程的方法, 以及用不同设计器完成复杂工作的方法, 换句话说, 就是如何用 IDE 来完成某些任务, 而不必使用代码或其他外部工具。

本章包含的几个实用练习将引导你创建数据库工程并添加脚本、查询和命令文件。相关内容可以查看正文中的练习。

4.1 使用服务器资源管理器

服务器资源管理器位于 IDE 的左部, 如果将鼠标光标移到 [Server Explorer] 选项卡上或按下 [Ctrl] + [Alt] + [S], 则都将显示服务器资源管理器。如果单击 IDE 的其他部分 (如 Code Editor), 服务器资源管理器就会再次隐藏。服务器资源管理器窗口包含与创建的数据连接和所连接的服务器相关的树状视图。



服务器资源管理器窗口显示的资源与当前打开的工程无关。

当第一次打开 Visual Studio.NET IDE 时, 服务器资源管理器不会显示任何数据连接或服务, 如图 4.1 所示。

4.1.1 处理数据连接

如果需要操作数据库或创建强类型数据集, 则先要创建数据连接。根据访问权限的不同, 可以从服务器资源管理器完成大多数数据库任务。

添加数据连接

通过在 [Data Connections] 节点上单击鼠标右键并从弹出菜单中选取 [Add Connection]

命令, 可以完成数据连接。单击右键后将出现 [Data Link Properties] 对话框。而单击 [Provider] 选项卡后将看到如图 4.2 所示的对话框。如果你在 VB6 Data View 中创建过数据连接, 那么应该非常熟悉该对话框。

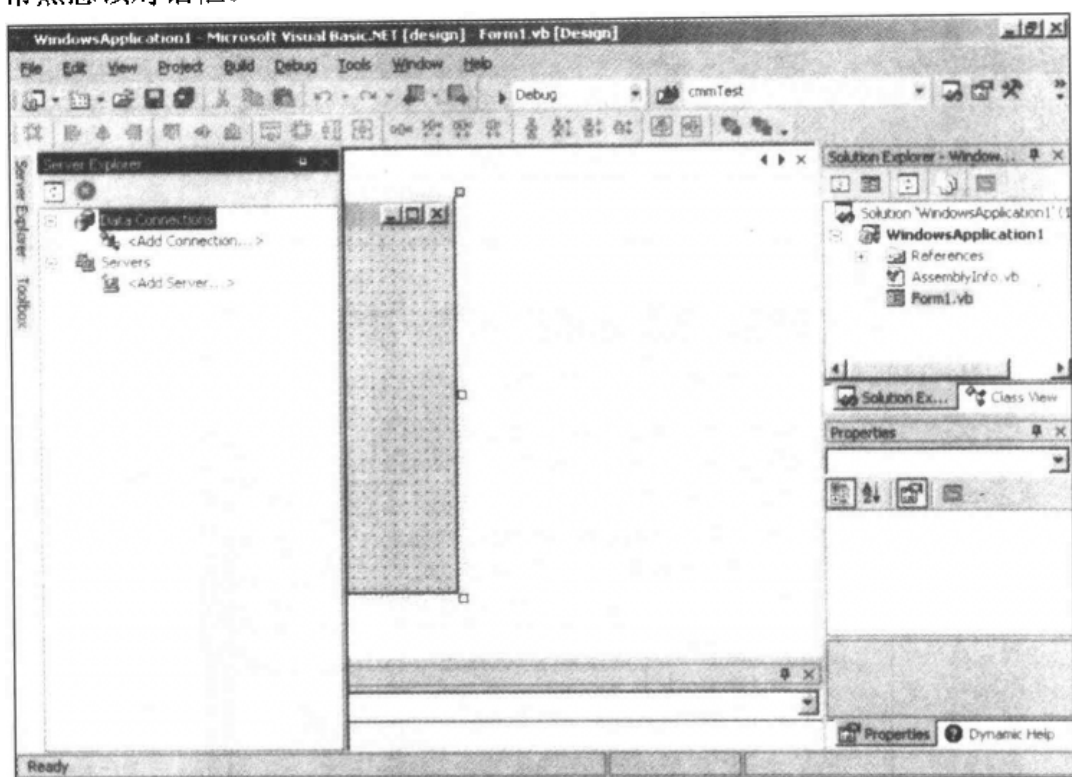


图 4.1 空白的服务器资源管理器窗口

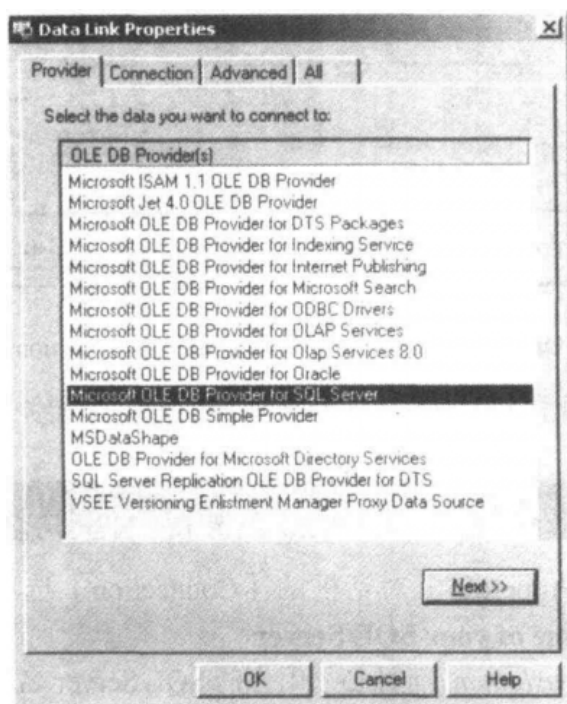


图 4.2 [Data Link Properties] 对话框中的 [Provider] 选项卡

若要连接到某数据源，则必须在 [Data Link Properties] 对话框的 [Provider] 选项卡上选取合适的提供程序 (provider)，然后再单击 [Next] 按钮。此时将显示如图 4.3 所示的 [Connection] 选项卡。

练 习

在 [Data Link Properties] 对话框中的 [Provider] 选项卡上，选取 [Microsoft OLE DB Provider for SQL Server]。

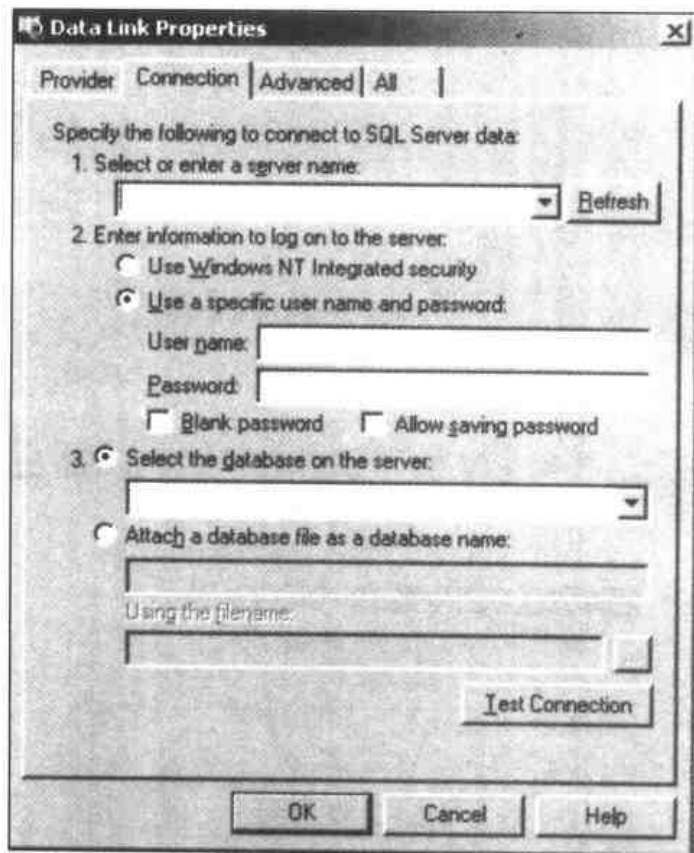


图 4.3 [Data Link Properties] 对话框中的 [Connection] 选项卡

在 [Connection] 选项卡上，必须输入与要连接的数据库相关的详细资料。

练 习

1) 在 [Data Link Properties] 对话框的 [Connection] 选项卡中输入下列文本：
USERMANPC for the name of your SQL Server

请参阅图 4.4。密码为 *userman*，而且必须用你的 SQL Server 名称代替为 *USERMANPC*。

2) 单击 [OK] 按钮。

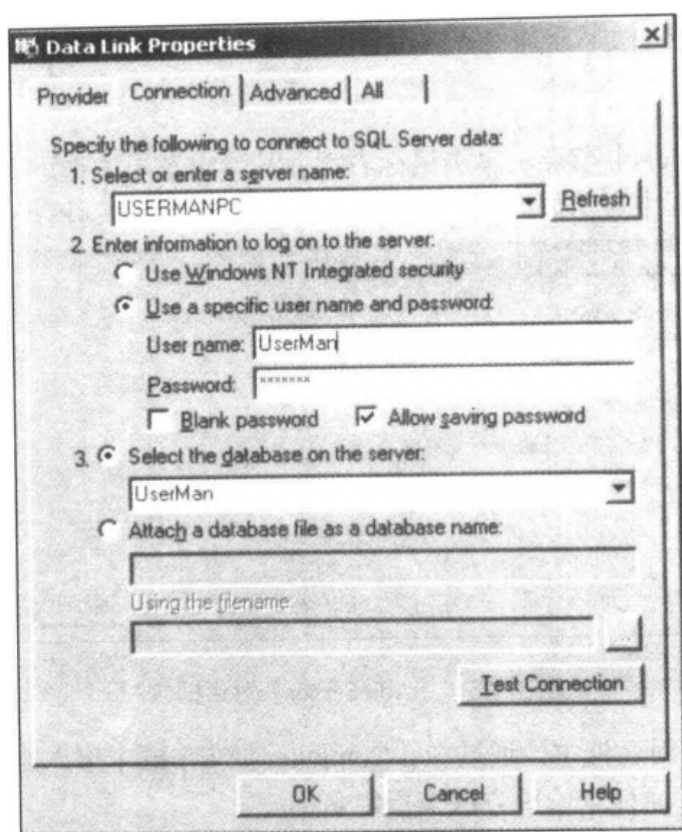


图 4.4 [Data Link Properties] 对话框中 [Connection] 选项卡上的 SQL Server 设置

删除数据连接

当使用完某个数据连接后，应该从服务器资源管理器中将其删除。有三种方法可以从服务器资源管理器中删除数据连接。首先必须在树状视图中选取合适的节点，然后再执行下列所示的一种操作：

- 按下 [Delete] 键。
- 在树状视图节点上单击鼠标右键，并从弹出菜单中选取 [Delete] 命令。
- 从 [Edit] 菜单中选取 [Delete] 命令。

请注意，这个过程只是删除了连接，而没有删除与之连接的数据库。如果需要删除 SQL Server 数据库，请参阅本章后面的“删除/撤销 SQL Server 数据库”部分。

创建数据库对象

如果要创建数据库对象，如表、图表、视图、存储过程或函数，则应先选取对应的树状视图节点，然后再在该节点上单击鼠标右键，并选取 [New...] 命令。有关创建数据库对象的更多信息请参阅第 6 章，第 2 章包含了建立关系数据库的有关信息。

4.1.2 处理服务器

在服务器资源管理器中的 [Servers] 节点下，可以添加能够访问的任何服务器。因为服务器的资源都显示在服务器资源管理器窗口中，更易于控制和操作，所以至少应添加当前工

程将要访问的所有服务器。

添加服务器

可以通过在 [Servers] 节点上单击鼠标右键并从弹出菜单中选取 [Add Server] 命令来添加服务器，在该过程中显示如图 4.5 所示的 [Add Server] 对话框。

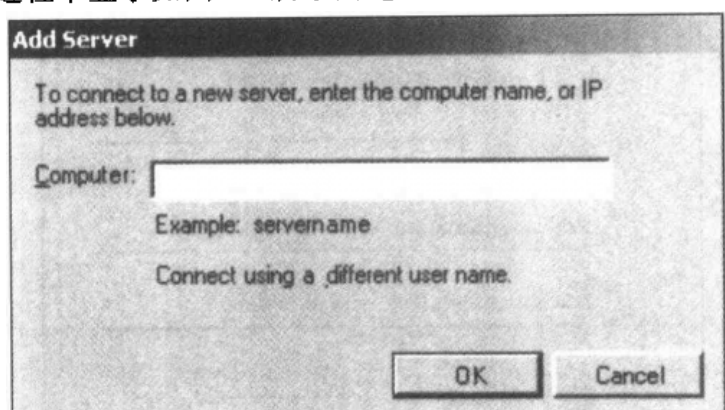


图 4.5 [Add Server] 对话框

在 [Add Server] 对话框中，通过在 [Computer] 文本框中输入服务器的名称就可以添加该服务器。可以用下面任意一种方法来指定名称：

- 如果服务器在局域网中，可以使用 NETBIOS 名称，例如 USERMANPC。
- 使用服务器的 IP 地址，例如 192.129.192.15。

练 习

在 [Add Server] 对话框的 [Name] 文本框中输入服务器名称，并单击 [OK] 按钮。

如果要以不同的用户身份连接服务器，则必须单击 [Connect using a different user name]（使用不同用户名连接），这时显示如图 4.6 所示的 [Connect As] 对话框。

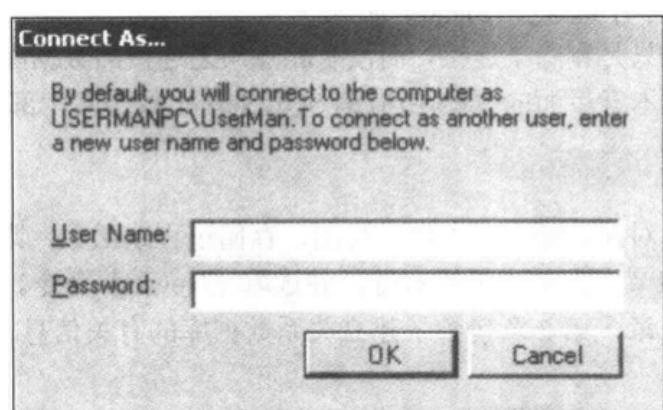


图 4.6 [Connect As] 对话框

在 [Connect As] 对话框中有两个文本框必须填写：

- **User Name:** 键入要以该用户身份连接的用户名。如果该用户在不同的域中, 那么必须指定域名, 其格式为 DomainName\UserName (域名\用户名)。

- **Password:** 输入与用户名对应的密码。如果该用户没有密码, 则该文本框为空。

向服务器资源管理器添加服务器之后, 通过展开树状节点 (如果节点未展开, 可以单击“+”图标进入下一层节点) 就可以查看该服务器提供的所有资源了。

在服务器中有许多不同资源, 这里只介绍了与数据库相关的部分, 包括消息队列和 SQL Server 数据库。

使用服务器资源

下面介绍如何使用你所选择的服务器中与数据库相关的资源。

使用消息队列

第 8 章将深入介绍消息队列, 因此在这里不会讨论太多细节。我将演示如何从服务器资源管理器创建、操作和删除消息队列。另外需要注意的是: 只能以相关消息队列访问服务器上的消息队列, 也就是不支持工作组消息队列设置。工作组设置是不相关的消息队列, 即: 对消息存储而言, 工作组设置与服务器毫不相关。

虽然可以看见何种消息取决于你的权限, 但是可以在服务器资源管理器中看到三种类型的消息队列:

- **Private queue (私有队列):** 这种队列在本地计算机上注册, 且不是目录服务的一部分。一般而言, 其他应用程序不能定位这种队列。欲了解更多信息请参阅第 8 章。
- **Public queue (公有队列):** 这种队列在目录服务中注册。它可以由其他消息队列应用程序进行定位。欲了解更多信息请参阅第 8 章。
- **System queue (系统队列):** 这种队列一般由操作系统用于内部消息。欲了解更多信息请参阅第 8 章。

图 4.7 显示了服务器资源管理器中展开的 [Message Queue] 节点。

创建消息队列

要创建新队列, 可在队列节点上单击鼠标右键, 并从弹出菜单中选取 [Create Queue]。这时将显示如图 4.8 所示的 [Create Message Queue] 对话框。

通过在文本框中键入文本给消息队列命名。若选中 [Make queue transactional] 复选框, 则只接受作为事务处理一部分的消息。请注意, 在连接的服务器上创建消息队列需要一定权限。单击 [OK] 按钮以创建队列。

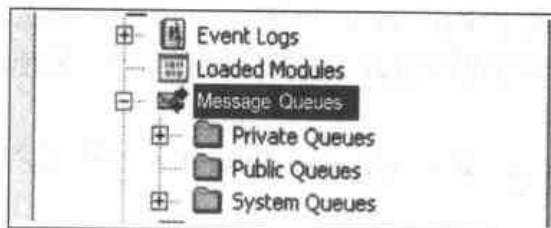


图 4.7 服务器资源管理器中展开的 [Message Queue] 节点

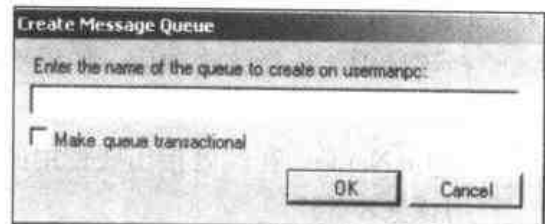


图 4.8 [Create Message Queue] 对话框

删除消息队列

如果不再使用某个消息队列，则可以将其删除。只需在该消息队列上单击鼠标右键，并从弹出菜单中选取 [Delete] 命令即可，注意，该过程中要求确认删除操作。该队列中的所有消息都将被永久删除。

从消息队列中删除消息

如果需要从队列中清除一个或多个消息，可用两种方法从服务器资源管理器完成这项操作。可以一次删除一个消息，也可以从队列中删除所有消息。可以用下述方法从队列中清除所有消息：展开队列节点并在 [Queue Message] 节点上单击鼠标右键，从弹出菜单中选取 [Clear Message] 命令，然后确认删除操作。要从队列中清除单个消息，可以选择该消息，再单击鼠标右键，然后从弹出菜单中选取 [Delete] 命令。

使用 SQL Server 数据库

使用 SQL Server 资源可以查看特定服务器是否运行了 SQL Server。通过单击 “+” 图标可以展开 [SQL Server Databases] 节点。如果单击了 SQL Server，那么在 SQL Server 服务未运行时，将自动启动 SQL Server 服务。这与使用 [Data Connections] 不同，在 [Data Connections] 中若想添加连接就必须启动该服务。

创建 SQL Server 数据库

如果还没有创建需要连接的数据库，则可以按照下面的方法进行创建。在服务器资源管理器窗口展开运行 SQL Server 的服务器，然后再展开 [SQL Server Databases]，并在要创建数据库的 SQL Server 上单击鼠标右键。此时将弹出菜单，从中选取 [New Database] 命令。将显示如图 4.9 所示的 [Create Database] 对话框，在这里可以指定数据库的属性。

如果你曾经在 SQL Server 7.0（或更新版本）中使用过服务器资源管理器，那么可能会比较熟悉该对话框的内容。如下所示：

- **Server:** 这是运行 SQL Server 的服务器的名称。如果需要改变该项内容，则必须取消该对话框并打开所要求的服务器中的相应对话框。
- **New Database Name:** 这是新数据库的名称。
- **Use Windows NT Authentication:** 如果要使用 Windows NT 身份验证来连接到 SQL Server，则应选择该选项。不管使用什么用户名进行连接，用户名都将作为运行 SQL Server 的服务器进行身份验证的基础（本书没有例举该选项的用法）。
- **Use SQL Server Authentication:** 如果要使用 SQL Server 自身的身份验证，那么应选择该选项。选择该选项后，如果不打算使用系统管理员账户，则必须创建要使用的登录名。
- **Login Name:** 在此处指出要使用的登录名。如果使用 Windows NT 身份验证，则禁用该文本框。
- **Password:** 键入与登录名对应的密码。如果使用 Windows NT 身份验证，则禁用该文本框。

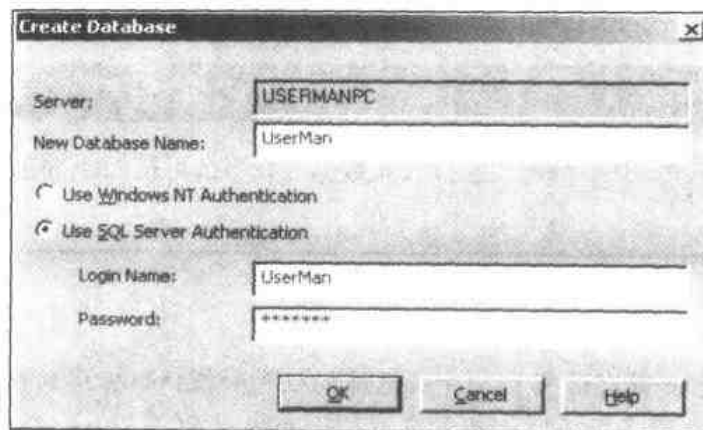


图 4.9 【Create Database】对话框

练 习

你可能已经在 SQL Server 中创建了 UserMan 数据库。但是，如果想创建新数据库，只需按照下列说明来填写【Create Database】对话框：

- 1) 在【Server】组合框中键入或选择服务器名称。
- 2) 在【New Database Name】文本框中键入 **Test**。
- 3) 选择【Use SQL Server Authentication】选项。
- 4) 在【Login Name】文本框中键入具有创建数据库权限的登录名。
- 5) 键入与第 4 步中所使用的登录账户对应的密码。

通过单击【OK】按钮来创建新数据库并更新显示，使之包含新的数据库。在本章前面的“处理数据连接”部分，我们已经介绍了连接 SQL Server 数据库方面的内容。

注意

在用这种方法创建 SQL Server 数据库时，新的数据库将使用默认值，而这些值都在服务器中进行设置。如果要用非默认值或属性创建数据库，则必须使用 SQL 语句 **CREATE DATABASE** 编写代码，或者从 SQL Server Enterprise Manager 的 Microsoft Management Console (MMC) 创建。另外，可以先创建上面所示的数据库，然后再通过 SQL 语句 **ALTER DATABASE** 的代码来改变某些默认值，如改变数据库和/或日志文件的位置。

无法用服务器资源管理器创建基于文件的数据库，如 MS Access JET 数据库。要创建基于文件的数据库，需要使用相应的前端工具（这种情况下为 MS Access）。或者用合适的提供程序从代码中执行 SQL 语句 **CREATE DATABASE**。

删除/撤销 SQL Server 数据库

不能从服务器资源管理器窗口自动删除或撤销 SQL Server 数据库。必须从 SQL Server

Enterprise Manager 或通过使用 SQL 语句 **DROP DATABASE** 的代码来完成这项操作。

练 习

如果你在前面的练习中已经创建了 Test 数据库,那么可按照下面的提示删除或撤销该数据库。



实际上,如果有访问 master 数据库的权利和撤销数据库的权限,那么就可以按照下面的步骤从服务器资源管理器窗口删除和撤销 SQL Server 数据库:

- 1) 打开所讨论的服务器上除 master 数据库之外的数据库。正如前文所叙述,你的登录名必须对该数据库具有合适的权限。
- 2) 展开 [Tables] 节点,并在该节点上单击鼠标右键。
- 3) 从弹出菜单中选取 [Retrieve Data from Table] 命令。
- 4) 在 [Query] 工具栏上单击 [Show SQL Pane] 按钮 (检查工具的提示)。
- 5) 从 SQL 窗格中删除 SQL 语句 **SELECT *....**
- 6) 输入 **USE master**,并在 [Query] 工具栏上单击 [Run Query] 按钮。
- 7) 在得到的对话框中单击 [OK] 按钮。
- 8) 删除 SQL 窗格的内容,并输入 **DROP DATABASE databasename** (其中 databasename 是准备删除或撤销的数据库的实际名称),单击 [Run Query] 按钮。
- 9) 单击 [OK] 按钮。。

删除服务器

如果某个服务器不会再被使用,那么就应该从服务器资源管理器中将其删除。有三种方法可用来从当前工程中删除服务器。首先必须在树状视图中选取合适的节点,然后再完成下列任一步骤:

- 按下 [Delete] 键。
- 在树状视图上单击鼠标右键,并从弹出菜单中选取 [Delete] 命令。
- 从 [Edit] 菜单中选取 [Delete] 命令。

4.2 数据库工程一览

数据库工程用于存储连接、SQL 脚本和命令文件 (如批处理脚本和/或计划脚本执行)。除此之外,通过数据库工程还可以使用 Visual SourceSafe 来处理不同版本的数据库对象。换句话说,数据库工程用于直接操作数据库对象!

可以按照下面的步骤创建数据库工程:

1. 选取 File/New/Project 菜单命令 (或按下 [Ctrl] + [Shift] + [N]),此时将显示如图

4.10 所示的 [New Project] 对话框。

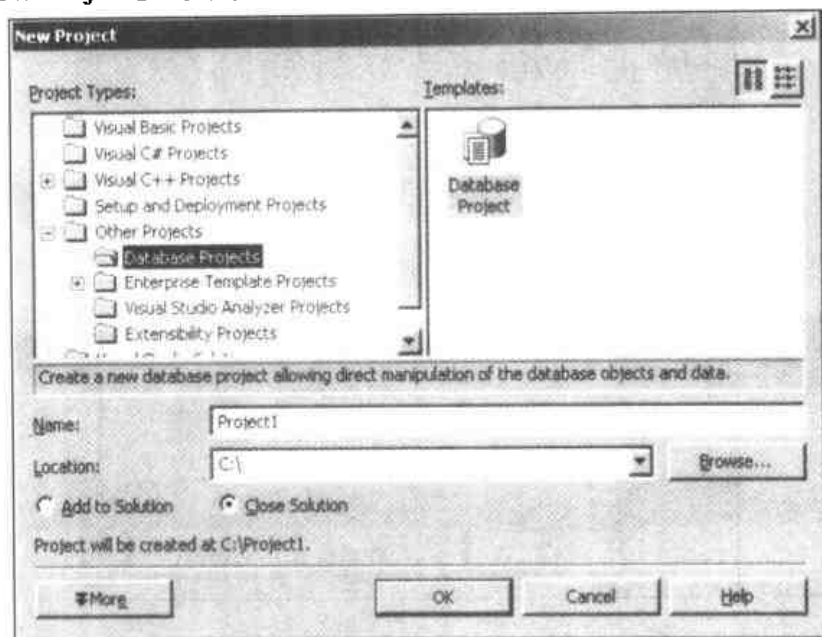


图 4.10 [New Project] 对话框

2. 在 [New Project] 对话框中，展开 [Other Projects] 节点并选择 [Database Projects]。
3. 在 [Name] 文本框中输入工程名称，并在 [Location] 文本框中指定工程位置。
4. 最后，如果要将新工程添加到当前解决方案中，或者要关闭当前解决方案并创建新的解决方案，那么就需要进行指定。可通过选择 [Add to Solution] 选项或 [Close Solution] 选项来进行指定。请注意：只有在当前打开了工程的情况下，才能使用 [Add to Solution] 选项和 [Close Solution] 选项。

练 习

在 [New Project] 对话框的 [Name] 文本框中输入 UserMan DB Project，将该工程保存到硬盘上。

5. 单击 [OK] 按钮，显示 [Add Database Reference] 对话框。该对话框如图 4.11 所示。

[Add Database Reference] 对话框中有来自服务器资源管理器的所有数据连接的列表，详情请参阅本章前面的“处理数据连接”部分，在其中选择期望的数据连接。如果列表中没有显示要引用的数据库，则可以通过单击 [Add New Reference] 按钮进行添加。此时将显示 [Data Link Properties] 对话框，本章前面的“添加数据连接”部分已经对此进行了介绍。单击 [OK] 按钮，新工程将出现在解决方案资源管理器中。

练 习

在 [Name] 文本框中输入 UserMan DB Project。

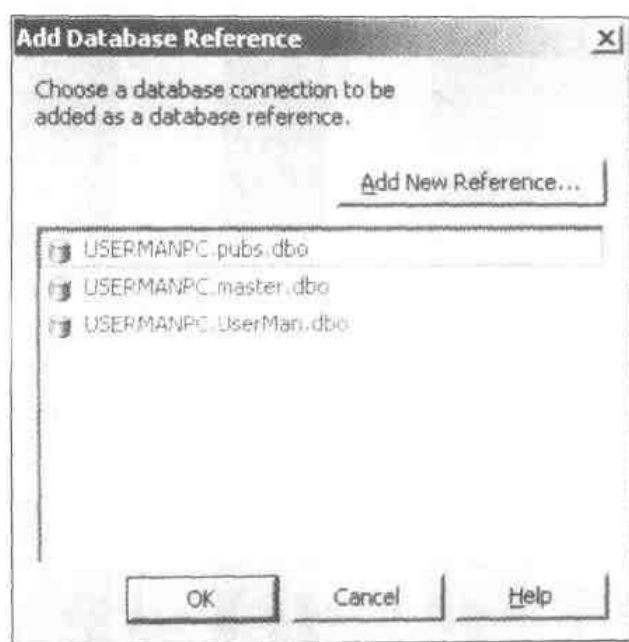


图 4.11 [Add Database Reference] 对话框

6. 如果要向工程中添加更多的数据库引用，可以在解决方案资源管理器中的 [Database References] 节点上单击鼠标右键，并在弹出菜单中选取 [Add Reference] 命令。



向数据库工程添加数据库引用时，如果服务器资源管理器的 [Data Connections] 中不存在该数据库引用，则该数据库引用将自动添加到 [Data Connections] 节点。

如果工程中有多个数据库引用，可以将其中一个设置为默认引用。指定默认引用的结果是：当创建对象（例如脚本）时，添加到脚本的表的列表将由默认数据库中的表组成。另外，当在工程中运行脚本和查询时，也要使用默认数据库引用。要设置默认数据库引用，可以在工程文件夹的根节点上单击鼠标右键，并从弹出菜单中选取 [Set Default Reference] 命令。此时将显示 [Set Default Reference] 对话框，其外形非常类似于图 4.11 所示的 [Add Database Reference] 对话框。选择作为默认值的数据库，并单击 [OK] 按钮。完成这些操作后，你会注意到解决方案资源管理器中的 [Database References] 节点已经改变了。默认数据库引用有其新的图标，表示它是默认引用。

4.2.1 创建数据库工程文件夹

如果在解决方案资源管理器中查看数据库工程（确保数据库工程已经展开），则会看到已经创建了三个子文件夹。这些文件夹用于对数据库进行分组，如下所述：

- Change Scripts 文件夹用于处理删除和更新脚本，等等。
- Create Scripts 文件夹用于创建类似数据库或表的脚本。
- Queries 文件夹用于保存返回行或单个/标量值及类似的脚本。

如果需要更多文件夹，或者只想创建自己的文件夹来对数据库对象分组，那么按照下列方法操作将会感到很自由。需要做的就是：在解决方案资源管理器中的数据库工程节点上单击鼠标右键，并从弹出菜单中选取 [New Folder] 命令。新文件夹将自动添加到解决方案资源管理器中。你可以为新文件夹命名，并可以像在普通文件系统中一样为其创建子文件夹。



虽然可以按照你自己的想法来对数据库对象进行分组，但是应遵循常识性的文件夹命名方案，如用于默认文件夹的名称，并确保数据库对象位于合适的文件夹，以便于访问。这样更容易查找指定的数据库对象，而且更易于创建仅包含来自于一个文件夹的脚本的命令文件。

4.2.2 删除数据库工程文件夹

可以为数据库对象创建文件夹，也可以删除文件夹。在需要删除的文件夹上单击鼠标右键，从弹出菜单中选取 [Remove] 命令，并确认文件夹删除操作，即可删除该文件夹。实际上，可以选择只从工程中移走文件夹，或是从硬盘上删除文件夹并从工程中移走该文件夹。



在删除文件夹时必须小心，因为此时会删除文件夹中的所有数据库对象。

4.2.3 向数据库工程添加数据库对象

可以向数据库工程添加新的和已存在的数据库对象。有关如何完成这两种任务的信息，请参阅下面的部分。

添加新的数据库对象

如果要向工程中的任意文件夹（包括工程根文件夹）中添加新的数据库对象，则可以在所期望的文件夹上单击鼠标右键，并从弹出菜单中选取 [Add New Item] 命令。在弹出菜单中还有其他选项也可用于该任务，但是 [Add New Item] 命令包含了 [Add SQL Script] 和 [Add Query] 命令。选择该命令后，将弹出 [Add New Item] 对话框（请参阅图 4.12）。

从图 4.12 可以看出，可以向数据库工程添加几种不同的数据库对象。在向工程添加新的数据库对象时，必须基于模板进行添加。[Add New Item] 对话框右边的窗格显示了可用的模板。除了 Database Query 模板外，其他模板在单击 [Open] 按钮后将打开 SQL Editor 进行编辑（有关使用这项功能的更多信息，请参阅本章后面的“使用 SQL Editor 编辑脚本”部分）。如果基于 Database Query 模板创建了新的数据库对象，那么单击 [Open] 按钮后将显示 Query Designer（有关使用 Query Designer 的更多信息请参阅本章后面的“使用 Query Designer 设计查询”部分）。

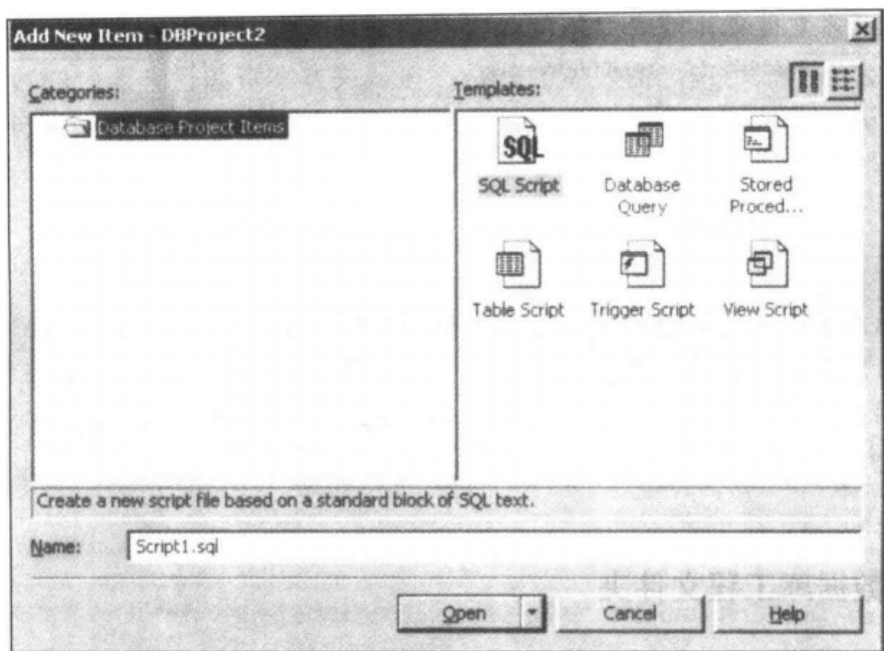


图 4.12 [Add New Item] 对话框

在创建脚本或查询时，Query Designer 和 SQL Editor 在某些方面是一致的。即，有些任务可由两种工具中的任意一种完成，究竟选择哪种工具完全取决于你的偏爱。Query Designer 具有拖放功能，易于使用，而使用 SQL Editor 可以完成更加复杂的任务。表 4.1 列举了可用模板，描述了创建特定数据库对象的最佳模板，并介绍了工具的使用方法。

表 4.1 数据库对象模板

模板名称	默认工具	说明
Database Query	Query Designer	使用数据库查询来创建返回行的查询、删除和更新查询，产生/创建表查询，以及插入查询。也可用 SQL Editor 完成该任务，但是 Query Designer 更适用于该任务。Query Designer 具有查询网格和拖放功能，因此更易于使用
SQL Script	SQL Editor	如果要创建脚本（其他模板没有包含的脚本），则应使用 SQL Script 模板
Stored Procedure Script	SQL Editor	使用 Stored Procedure 模板创建存储过程，用于在服务器端快速处理重复的查询或函数。有关存储过程的更多信息请参阅第 6 章
Table Script	SQL Editor	使用 Table Script 模板产生 SQL 脚本 CREATE TABLE。我个人认为用 Query Designer 完成该任务会更简单，但是，有时你可能要先添加已创建表的脚本，然后再用 SQL Editor 进行编辑
Trigger Script	SQL Editor	Trigger Script 模板用于为服务器端处理或数据操作的确认而创建触发器。有关触发器的更多信息请参阅第 6 章
View Script	SQL Editor	View Script 模板用于创建视图以加速返回行的查询。有关视图的更多信息请参阅第 6 章

在 IDE 中运行脚本

创建完脚本后，便可以在 IDE 中测试脚本。在解决方案资源管理器的脚本上单击鼠标右键，并从弹出菜单中选取 [Run] 命令，则将在默认数据库上执行该脚本。如果要在不同的数据库上执行该脚本，可从上述弹出菜单中选取 [Run On] 命令。此时将显示 [Run On] 对话框，如图 4.13 所示。

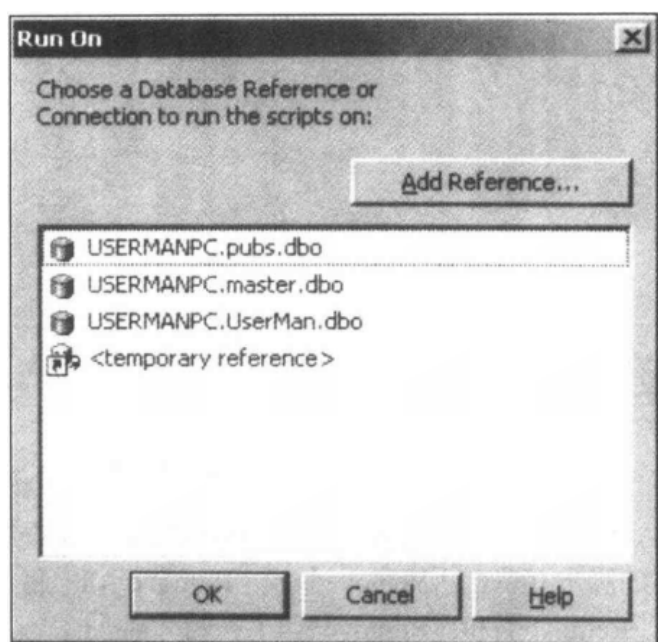


图 4.13 [Run On] 对话框

[Run On] 对话框与图 4.11 中的 [Add Reference] 对话框非常相似，但是其临时引用（由 <temporary reference> 表示）有一点不同。你可以用临时引用来创建临时数据库连接，该连接会运行脚本并在脚本结束时销毁。

在选择了所需的引用后，单击 [OK] 按钮，从而在所选择的引用上运行该脚本。

添加命令文件

如果要同时执行几个脚本，则最好把这些脚本放在同一个命令文件中。这样，就可以通过执行该命令文件来执行一批脚本。实际上，因为可以在命令行中执行命令文件，所以它对计划执行来说也是很好的方法。这意味着将一个脚本放在命令文件中也是有意义的。命令文件是扩展名为 *.cmd 的 Windows 命令文件。

在创建命令文件时，需要先在解决方案资源管理器中选择放置该命令文件的文件夹，然后再单击鼠标右键。当从弹出的菜单中选取 [Create Command File] 命令后，将出现 [Create Command File] 对话框（请参阅图 4.14）。

在 [Create Command File] 对话框中，可以给定命令文件名称并选择脚本作为命令文件的一部分。[Available Scripts] 中列出的脚本是从创建命令文件的文件夹中获取的。你无法从其他文件夹中选择脚本，而这正是将脚本组织在正确的文件夹中尤为重要原因（这同样适用于查询）。

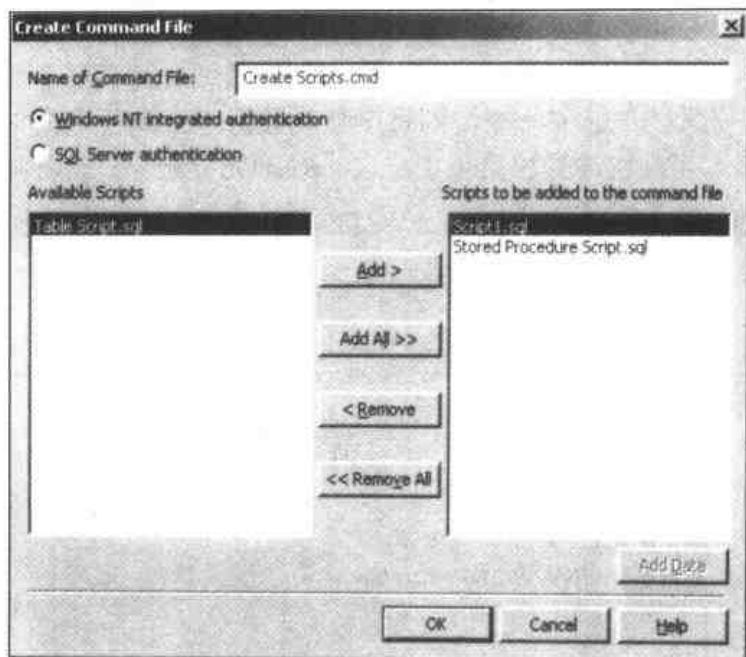


图 4.14 「Create Command File」对话框

使用 [Add] 或 [Add All] 按钮可以向命令文件添加脚本。在添加一个或多个脚本后，可以通过选择不想包含的脚本并单击 [Remove] 或 [Remove All] 将其移走。注意，只有向命令文件添加脚本后，[Remove] 和 [Remove All] 按钮才可见。

最后要做的是选择使用 Windows NT 身份验证，还是 SQL Server 身份验证。如果选择了 Windows NT 身份验证，那么将以登录用户或分配给计划任务服务的用户账户的权限执行脚本。但是，如果选择了 SQL Server 身份验证，那么当执行脚本时必须在命令提示符中提供登录名和密码。在完成这些操作后单击 [OK] 按钮，命令文件将出现在首次创建文件时用鼠标右键单击的文件夹中。

当执行命令文件时，需要提供服务器名称和数据库名称，如：

```
CommandFileName.cmd ServerName DatabaseName
```

添加已存在的数据库对象

如果已经创建了一个或多个要添加到工程中的数据库对象，只需使用弹出菜单进行添加即可。要在文件夹中添加对象，可在该文件夹上单击鼠标右键，然后从弹出菜单中选取 [Add Existing Item] 命令。这时将显示 [Add Existing Item] 对话框，在这里可以浏览并选择数据库对象。在选择需要的对象时，按下 [Ctrl] 键就可以从同一文件夹中同时选取多个数据库对象。选择了期望的对象后，单击 [Open] 按钮，所选对象会立即添加到工程中，并显示在更新的解决方案资源管理器窗口上。



注意

在向工程添加对象时，没有已存在项的确认操作，因此应确定添加了正确类型的数据库对象。

4.3 使用 Database Designer 设计数据库

Database Designer 是用于设计数据库的可视化工具。可用来创建表，包括键、索引、限制和表间的关系。Database Designer 会创建图表，通过该图表能够可视化地添加和操作数据库对象。数据库图表以图形形式描述数据库，显示数据库结构。这意味着数据库图表只能控制一个数据库中的数据库对象。即可以在一个数据库中创建几个图表，但是一个图表只能包含一个数据库。

4.3.1 创建数据库图表

数据库图表 (database diagram) 是创建数据库的可视化工具。可以添加或创建表，添加表间的关系，并执行设计数据库时几乎所有常用的任务。可以从服务器资源管理器创建数据库图表。若想展开 [Data Connections] 节点并创建图表的数据库。则需在 [Database Diagrams] 节点单击鼠标右键，并从弹出菜单中选取 [New Diagram] 命令。这时将显示如图 4.15 所示的 [Add Table] 对话框。在该对话框中，可以从要创建图表的数据库中选择期望的表并单击 [Add] 按钮。然后通过单击 [Close] 按钮关闭对话框，继续图表操作。



只有 SQL Server 和 Oracle 数据库可以使用数据库图表，Microsoft Access (JET Engine) 数据库无法使用。

练习

创建新的数据库图表，并在 UserMan 数据库中添加以 tbl 开头的表。

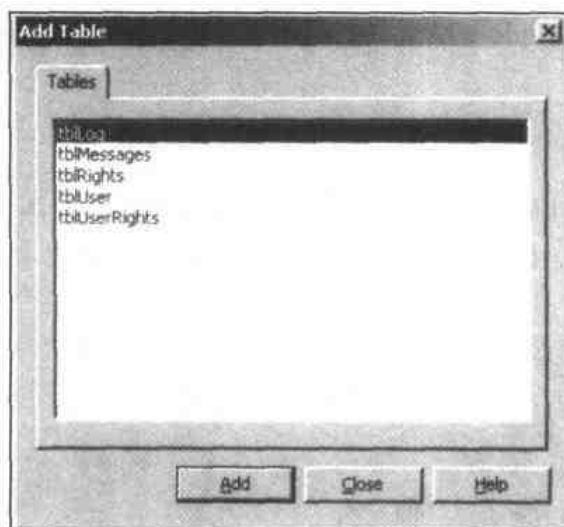


图 4.15 [Add Table] 对话框

打开图表后，所选的表将自动添加到图表中。如果设置了表之间的关系，则会自动显示这些关系（请参阅图 4.16）。

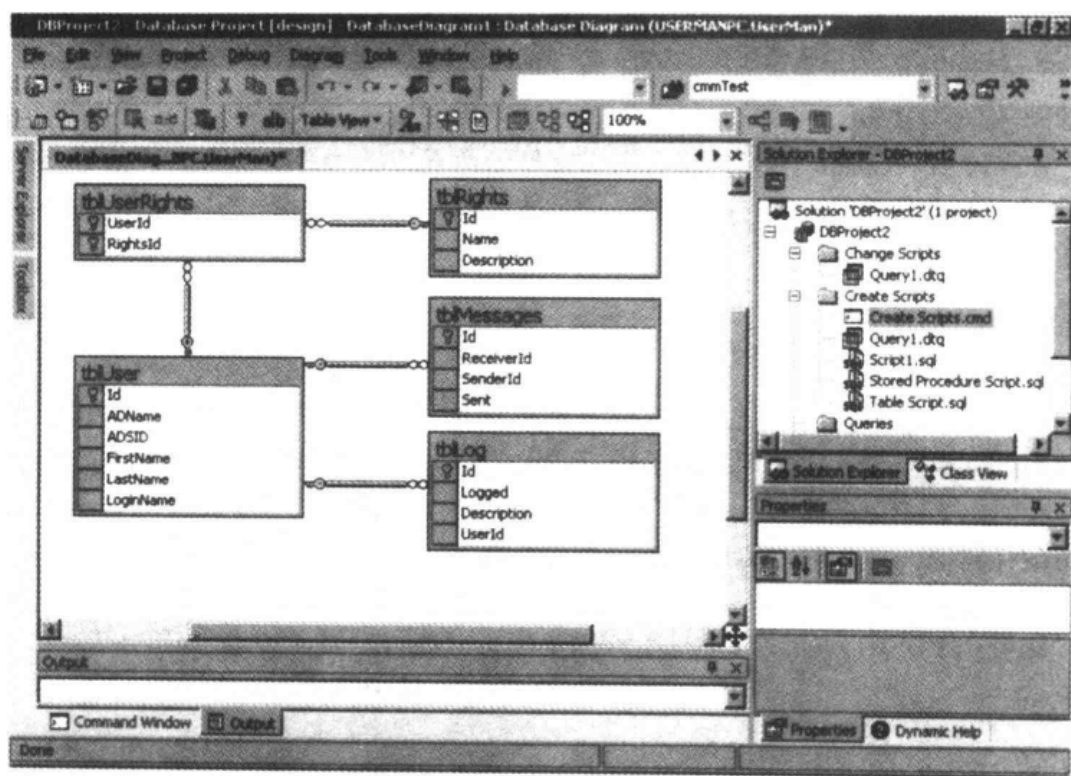


图 4.16 数据库图表

添加表

如果要向图表中添加多个表，则可以在图表中的任意空白处单击鼠标右键，然后从弹出菜单中选取 [Add Table] 命令。这时将显示如图 4.15 所示的 [Add Table] 对话框。选择要添加到图表中的表并单击 [Add] 按钮。然后单击 [Close] 按钮关闭对话框，继续数据库图表操作。

删除或移走表

如果向图表中添加了实际不需要的表，则应该将其移走。从图表中移走表和从数据库中删除表是完全不同的操作，因此我们将详细介绍。下面是从图表中移走表的方法：

1. 在要操作的表上单击鼠标右键。
2. 从弹出菜单中选取 [Remove Table from Diagram] 命令。

请注意：这项任务没有确认操作！如果你实际上要从数据库中删除该表，那么可以按下面的步骤执行：

1. 在要操作的表上单击鼠标右键。
2. 从弹出菜单中选取 [Delete Table from Database] 命令。
3. 在确认对话框中单击 [Yes] 按钮。

创建新表

如果没有创建要添加到数据库图表中的表，那么可以从图表内创建表。在图表的任意空白处单击鼠标右键并从弹出菜单中选取 [New Table] 命令。此时将显示 [Choose Name] 对话框（参看图 4.17），在该对话框中输入新表的名称，然后单击 [OK] 按钮。

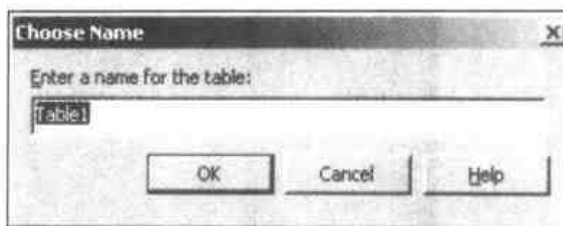


图 4.17 [Choose Name] 对话框

练习

在 [Choose Name] 对话框的文本框中键入 tblTest，并单击 [OK] 按钮。

当输入表的名称并单击 [OK] 按钮后，新表将显示在数据库图表中（参看图 4.18）。

对于新表中的每一个字段或列，均可指定其列名、数据类型和长度，并指出该列是否允许 Null（空）值。要添加新列，只需在 [Column Name]、[Data Type]、[Length] 和 [Allows Nulls] 属性中填写适当值即可，然后再将光标移到下面的行。这看起来很神秘并容易使人迷惑：在表设计视图中用行表示的表如何由列构成？现在先暂时放下这个问题，你最终将会理解这一点。

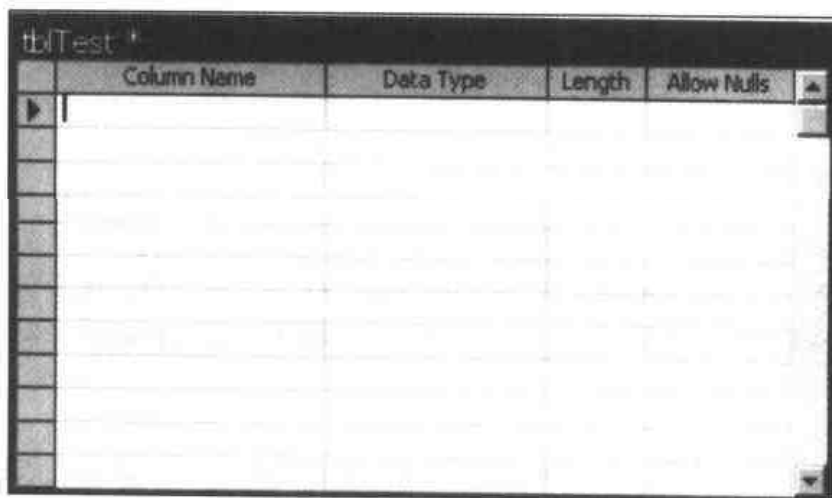


图 4.18 数据库图表中的新表

练习

填写图 4.19 所示的列属性并按下 [Ctrl] + [S]。

首先，你可能会注意到 [Data Type] 列属性是特定于 DBMS（数据库管理系统）的。这意味着数据类型下拉列表中所列举的是对创建图表的数据库有效的数据类型。

当输入列及列属性后，按下 [Ctrl] + [S] 保存图表并创建新表。虽然用 Database Designer（数据库图表）创建和编辑表非常方便，但是使用 Table Designer 可能更有帮助。更多信息

请参阅本章后面的“使用 Table Designer”部分。

Column Name	Data Type	Length	Allow Nulls
Id	int	4	
Name	varchar	50	
Description	varchar	255	✓

图 4.19 列属性

添加关系

关系用于表示不同表的数据之间的联系。创建表后，只需将一个表的字段拖拽到另一个表的相关字段就可以创建关系。此时将显示 [Create Relationship] 对话框，可以在此处定义关系。



对关系的详细说明请参阅第2章。

删除关系

如果由于某些原因关系作废了，则应该删除该关系。具体方法是，在关系上单击鼠标右键，并从弹出菜单中选取 [Delete Relationship from Database] 命令。这里需要确认该删除操作。与对数据库图表所做的其他改变一样，直到保存图表时才将删除操作保存到数据库中。

编辑数据库属性

如果要改变数据库的属性，如表的键或者两个表之间的关系，只需在表或关系上单击鼠标右键并从弹出菜单中选取 [Property Pages] 命令。这时将显示 [Property Pages] 对话框，在此处可以编辑表的键、索引和约束条件，以及表之间的关系。

数据库图表综述

相信你对旧版本的 MS 数据库工具也存在相同的问题：一旦向图表添加了许多表，要得到图表的完整视图几乎是不可能的。下列功能对你可能有所帮助。

- 放大以查看一个表，或者缩小以查看整个图表：使用快捷菜单就可进行缩放。在图表的任意空白处单击鼠标右键，从弹出菜单中选取 [Zoom] 命令，并指定要缩放的比例。如果以前曾改变了缩放比例，那么在菜单上将选中当前的缩放比例。在 [Zoom] 子菜单中有个命令值得特别注意，[To Fit] 命令。该命令会自动选择缩放比例，从

而使整个数据库图表显示在当前视图中。

- 移动视图 (view port), 或者改变图表的可视区域: 将视图图标放在图表的右下角 (参看图 4.20)。如果单击视图图标, 则可以在全景窗口中看到整个图表, 如图 4.21 所示。如果按住鼠标左键, 就可在图表周围移动视点。视图是由圆点组成的长方形, 当第一次单击视点图标时在全景窗口中可以看到该长方形。

练 习

将数据库图表放大到 200%, 并单击视图图标。按住鼠标左键并在数据库图表周围移动视图。当选定了图表的适当区域后, 放开鼠标左键。现在, 看到的数据库图表区域应该与用全景窗口选取的区域一致。

- 排列所有表, 使相关的表整齐且有秩序地排列: 在图表的任意空白处单击鼠标右键, 并选取 [Arrange Tables] 命令。
- 显示关系名称: 在图表的任意空白处单击鼠标右键, 并选取 [Show Relationship Labels] 命令。此时将在关系旁显示名称标签。
- 向图表添加描述性的文本标签: 在图表的任意空白处单击鼠标右键, 并选取 [New Text Annotation] 命令。在图表中单击鼠标右键的位置将打开文本字段。当输入文本之后, 单击图表的其他部分便可完成文本注释的编辑工作。使用这些文本注释来添加描述性文本可以增加图表的可读性。不过这些文本注释不是数据库的组成部分。如果要编辑已存在的文本注释, 则只需在图表上单击该文本就可以打开文本字段并进行编辑。
- 改变表视图: 如果要查看比所显示的属性更多或更少的表属性, 则可以选择所讨论的表, 并在其中一个表上单击鼠标右键。然后从弹出菜单中选取 [TableView] 命令, 并单击要改变的视图。在设计数据库时, Standard View (标准视图) 非常有用, 而 Column Name (列名) 视图则很好地描绘了表的轮廓。
- 自动设定表的大小: 如果一个或多个表的大小与行数和/或列数不相符, 那么可以选中这些表, 并在其中一个表上单击鼠标右键, 然后选取 [Autosize Selected Tables] 命令。

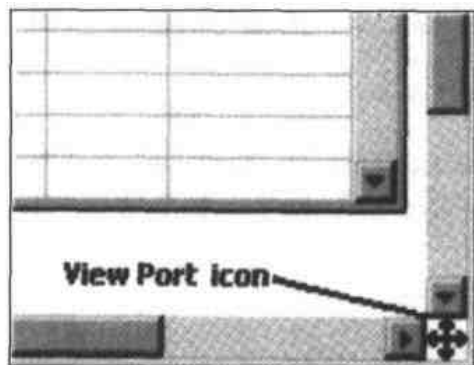


图 4.20 数据库图表视图图标

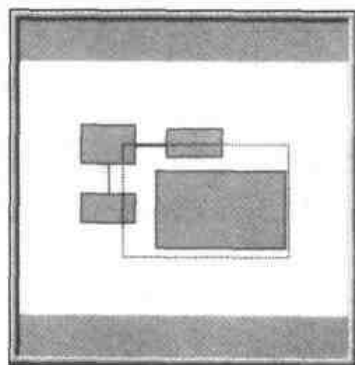


图 4.21 数据库图表全景窗口

4.3.2 保存数据库图表

打开数据库图表后，可以随时通过按 [Ctrl] + [S] 来保存图表。在图表保存到数据库之前会对其进行验证。如果已存在的数据与新的关系和/或约束条件冲突，就无法保存该图表，并且会出现描述该错误详情的对话框。在保存图表之前必须先更正错误，然后再向数据库保存所做的修改。

4.4 使用 Table Designer

到目前为止，Table Designer 是在数据库中创建新表的最全面的工具。虽然可以使用 Database Designer 向图表和数据库添加新表，但是使用 Table Designer 可以更好地从总体观察表。

要使用 Table Designer 创建新表，则必须打开服务器资源管理器，展开要创建表的数据库，并在该数据库的 [Tables] 节点上单击鼠标右键。然后，通过从弹出菜单中选取 [New Table] 命令打开 Table Designer，如图 4.22 所示。

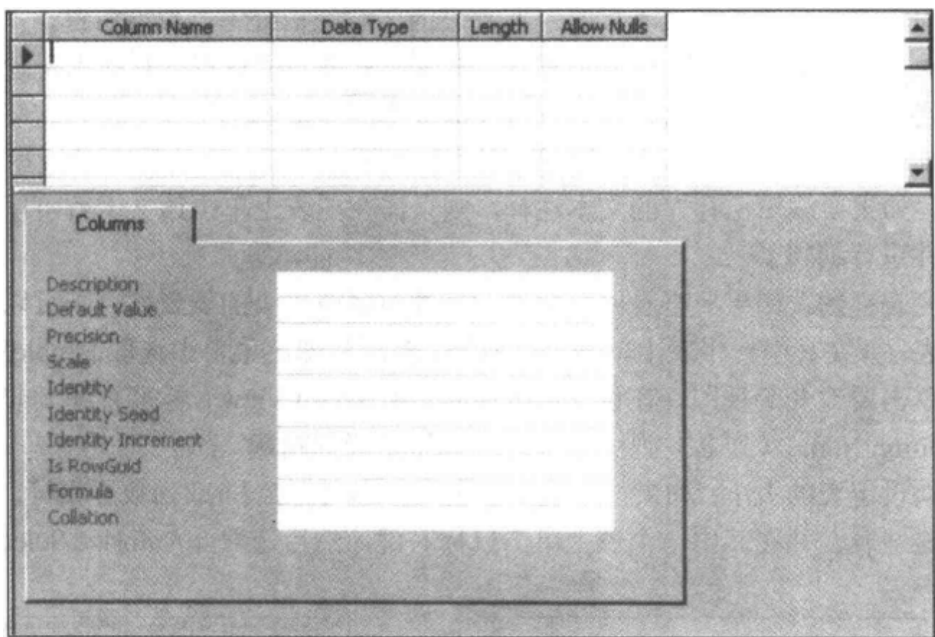


图 4.22 Table Designer

Table Designer 起初看上去与使用 Database Designer 的 [New Table] 功能时所见到的表视图非常相似（请参阅本章前面的“添加表”部分），但实际上 Table Designer 更复杂一些。

4.4.1 添加列

要向表中添加新列，则需在 [Column Name] 列中输入列的名称，且必须在 [Data Type]

属性中为该列填写期望的数据类型。请注意，在下拉列表中所列举的数据类型是专用于 DBMS 的，这意味着列表中包含的是对连接的数据库有效的数据类型。如果指定的数据类型具有不同的长度，如 SQL Server 中的 varchar 数据类型，那么还要求指定 [Length] 属性。在创建新列时（通过输入列名实现），默认情况下总会选中 [Allow Nulls] 属性。如果允许某列为 Null 值，那么该列则不能作为键段。因此，在向表添加数据之前，要确保已经清除了键段的 [Allow Nulls] 属性，该属性在以后很难修改。

迄今为止，我们所讨论的内容都与使用 Database Designer 添加新表非常相似。但也有很大的差异：Table Designer 底部的 [Columns] 属性窗格（请参看图 4.22）。根据连接的数据库不同，[Columns] 属性窗格也各异。在图 4.22 所示的是使用 SQL Server 的情况。但是有些属性是相同的，如下面的一些情况：

- **Description:** 使用该属性添加列的说明，如存储什么值。在数据库生命周期中，对以后接管数据库的用户而言，这些信息非常有价值。
- **Default Value:** 如果某行中没有为特定列赋值，那么 Default Value（默认值）用于指定该值。可用作允许 Null 值的地方，此处很难用代码处理。

4.4.2 设置主键

如果要为列设置主键，可以在 [Column Name] 旁的栅格上单击鼠标右键。接着，单击 [Set Primary Key] 设置该列为主键（请参看图 4.23）。如果需要组合的主键，那么要先选择组成主键的所有列，然后再在 [Column Name] 旁的栅格上单击鼠标右键。在 [Column Name] 旁的栅格上单击鼠标右键之前，可以用在列表中选择多项的方法选择多个列，即：在逐个选择列时，按住 [Ctrl] 键。如果列是连续的，则可以先选择第一列，然后按住 [Shift] 键同时选择最后一列。

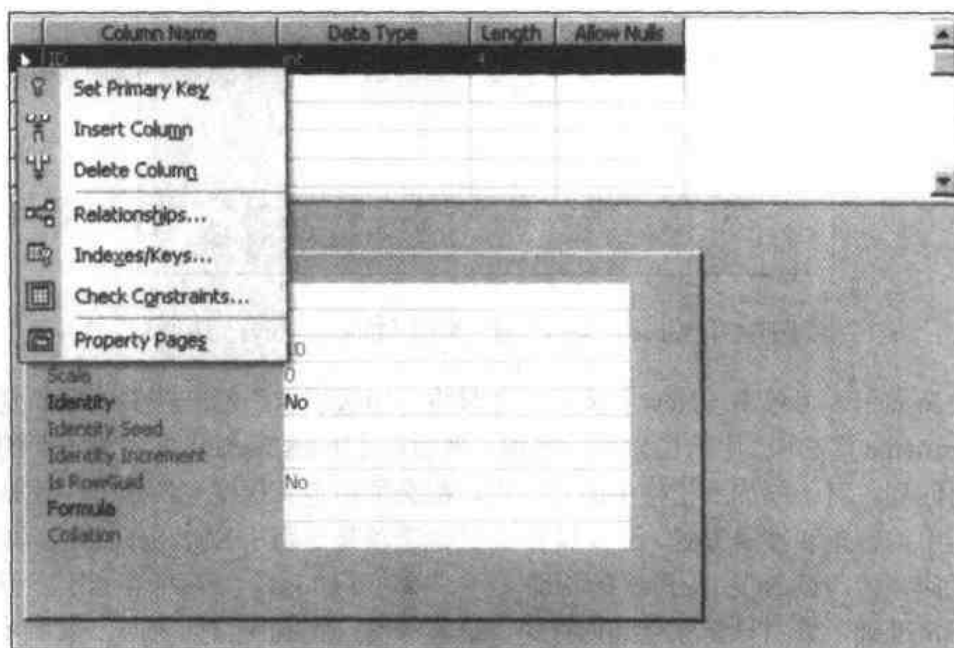


图 4.23 设置主键

4.4.3 添加索引和键

在用于搜索表的列上创建索引是个不错的主意。要创建索引，可在 Table Designer 中的表栅格上任意位置单击鼠标右键，并从弹出菜单中选取 [Indexes/Keys] 命令。此时将显示带有 [Indexes/Keys] 选项卡的 [Property Pages] 对话框（请参看图 4.24）。请注意，如果已经创建了索引和/或主键，那么选项卡可能会有所不同。此时会有 [Selected Index] 列表框，而且其中一个存在的索引处于选中状态。另外，该选项卡的内容也是专用于 DBMS 的，因此会根据所连接的数据库不同而有所变化。图 4.24 中的 [Indexes/Keys] 选项卡显示了连接到 SQL Server 数据库。

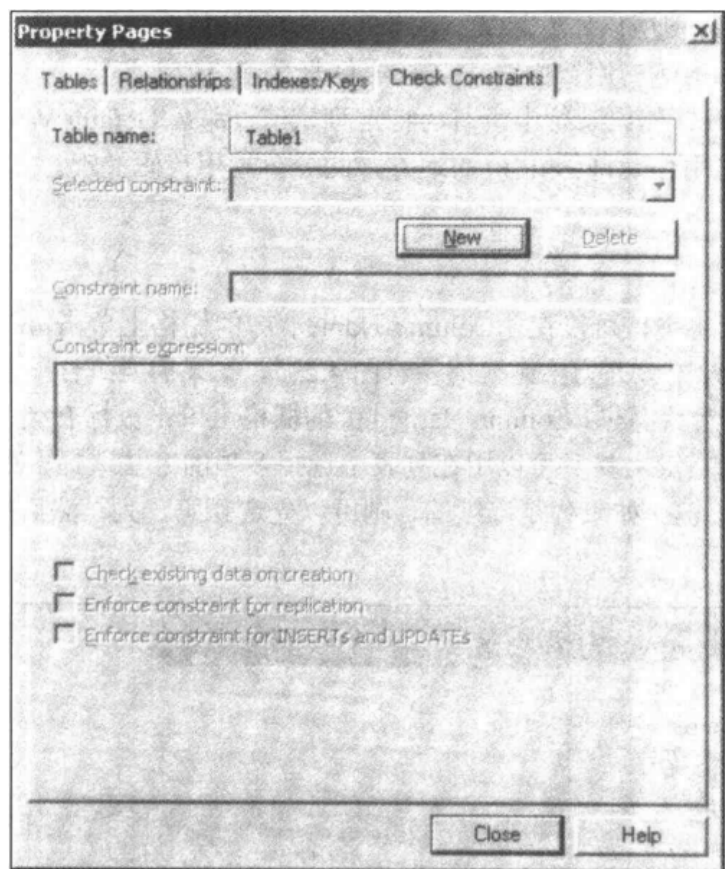


图 4.24 [Property Pages] 对话框的 [Indexes/Keys] 选项卡

要创建新索引，可单击 [New] 按钮。这时将启用选项卡上的一些文本框和复选框。首先在 [Index name] 文本框中给出索引的名称，然后在 [Index name] 文本框下面的栅格中添加组成索引的列。对于添加到栅格中的每一列，都必须指定其排序顺序是升序还是降序。这取决于索引中的数据和搜索方式。图 4.24 中的其他选项是专用于 SQL Server 的。如果你对 these 选项有疑问，请阅读 SQL Server 的帮助文件。单击 [Close] 按钮保存索引。但是，这么说并不是非常准确，索引只是保存到内存中。这意味着：直到保存图表时，索引才会保存到数据库。为避免混乱（不是必须的），应只将主键指定为键。因此，如果要添加关系中所使用的外键，则必须向所讨论的列添加索引。

4.4.4 添加约束条件

有时必须保证表中特定列的值在特定范围内。如果是这样，那么最好创建约束条件，从而强化规则或确保值在要求的范围之内。要创建约束条件，可以在 Table Designer 的表栅格上任意位置单击鼠标右键，再从弹出菜单中选取 [Check Constraints] 命令。此时将显示带有 [Check Constraints] 选项卡的 [Property Pages] 对话框（参看图 4.25）。

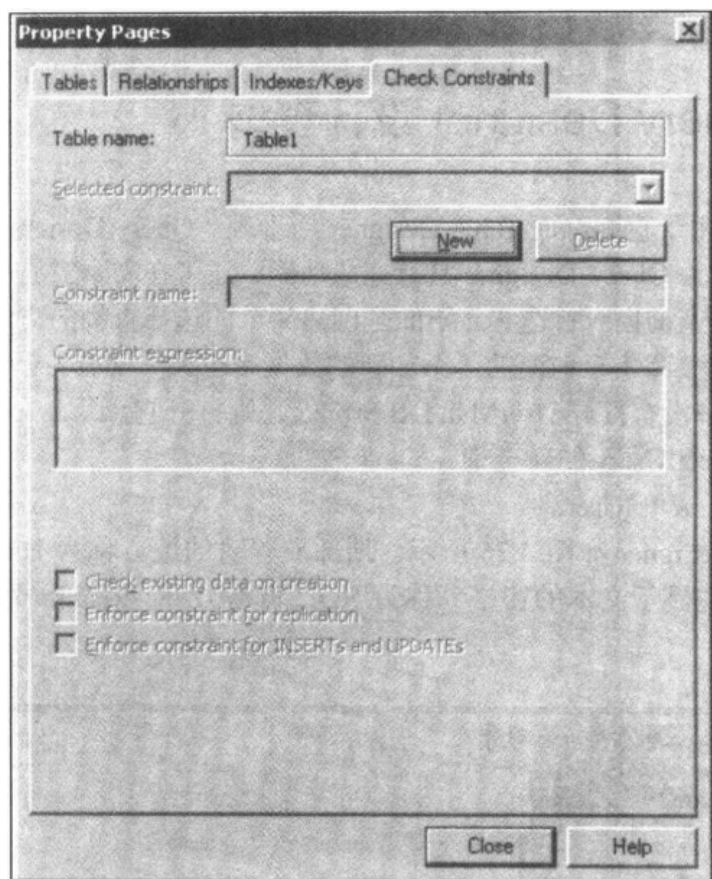


图 4.25 [Property Pages] 对话框的 [Check Constraints] 选项卡

要创建新约束条件，则单击 [New] 按钮。此时将启用选项卡上的一些文本框和复选框。首先在 [Constraint name] 文本框中给出约束名称，然后在 [Constraint expression] 文本框中输入约束表达式。下面是约束表达式的示例：

```
Test < > ''
```

上述表达式的含义是：字段 Test 不能为空字符串。下面是对选项卡上复选框的说明：

- **Check existing data on creation:** 保存表时，验证所有存在的数据是否与约束相冲突。与索引不同，在表中可能存在与列约束不一致的值。
- **Enforce constraint for replication:** 当表复制到另一个数据库中时，该选项指示强制使用该约束条件。
- **Enforce constraint for INSERTs and UPDATES:** 插入或更新数据时，该选项指示强制

使用该约束。这意味着：如果与约束不一致，插入或更新操作将失败。
单击 [Close] 按钮，约束将保存到内存中。

4.4.5 创建关系

虽然 Table Designer 是创建表的首选工具，但它并不是最适合创建关系的工具。用 Table Designer 创建表后，应打开数据库图表并使用 Database Designer 创建关系。有关该工具使用方法的信息，请参阅本章前面的“添加关系”部分。

4.5 使用 Query Designer 设计查询

Query Designer 用于查询，而 Table Designer 用于表。Query Designer 是简化查询创建操作的可视化工具。因为 Query Designer 具有拖放功能，还可在文本窗格中手工输入查询，所以，即使是最简单的查询也应首选它。下面是创建简单的选择查询的方法（如果不清楚执行下述任务的方法，请参阅本章前面的“添加新的数据库对象”部分）：

1. 如果在 IDE 中没有打开 UserMan DB Project，则打开它。
2. 添加名为 Select Users 的新查询。
3. 向查询中添加表 tblUser。

现在的 Query Designer 如图 4.26 所示。顶部放置表的地方称为 Diagram 窗格，下面是 Grid 窗格，接下来是显示文本的窗格（称为 SQL 窗格），而 Results 窗格位于设计器的底部。

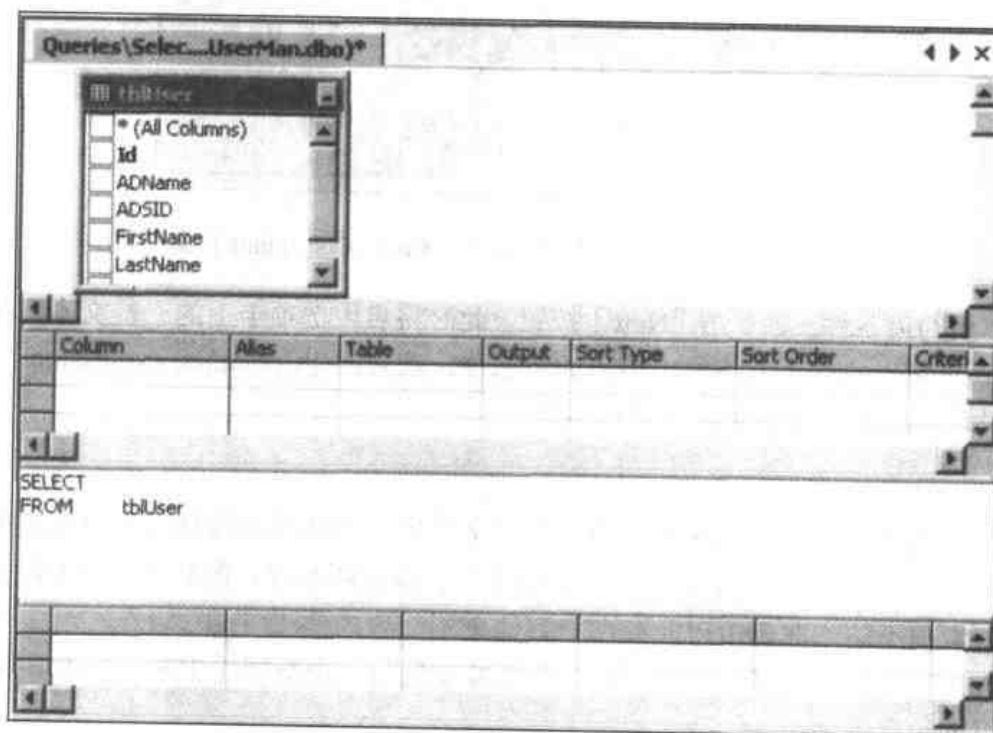


图 4.26 Query Designer

4.5.1 深入观察 Query Designer 的窗格

正如前文所述，在 Query Designer 中有 4 个窗格：

- Diagram 窗格
- Grid 窗格
- SQL 窗格
- Results 窗格

除了 Results 窗格之外，这些窗格都带有执行函数，它们在功能上是重叠的。实际上，Diagram、Grid、和 SQL 窗格不仅仅是功能重叠，它们还执行完全相同的函数。当编辑其中一个窗格时，观察其他两个窗格便可以发现它们是同步的。终究使用哪个窗格完全依据个人的偏好。有关每个窗格的功能，请参阅下面的内容。

Diagram 窗格

Diagram 窗格位于 Query Designer 的顶部，可以在该窗格内添加和显示查询中使用的表。要添加查询，可从服务器资源管理器中拖拽表，或者在 Diagram 窗格的任意位置单击鼠标右键，然后从弹出菜单中选取 [Add Table] 命令。Diagram 窗格中的每个表都可以选择列作为查询的一部分，而选中列名旁的复选框就可完成这项操作。如果要输入所有行，则只需选中选项 [*(All Columns)]。也可在表上单击鼠标右键，再选取 [Select All Columns] 命令，并按下 [Space] 键以选中复选标记。



如果选中了表中的所有列，又选中了选项 [*(ALL Columns)]，那么将两次获得所有行组成的输出。

移走表

在表上单击鼠标右键并从弹出菜单中选取 [Remove] 命令，就可移走 Diagram 窗格中的表。Diagram 窗格所允许容纳的表的数量依赖于查询类型。SELECT、Make Table 和 INSERT Results 查询可以支持任意数量的表，而 UPDATE、DELETE 和 INSERT VALUES 查询只支持一个表。如果 Diagram 窗格中的表是相关联的，或者在任意两个表间存在关系，那么在 Diagram 中将会显示出来。但是，不能在 Query Designer 中改变和/或删除这些关系。实际上，这并不完全正确。如果任意两个表具有内部或外部连接 (join)，而且从 Diagram 窗格中移走了这个连接，那么该连接将变成交叉连接 (cross join)。交叉连接会以所有可能的组合从讨论的表中输出所有选定的列。换句话说，不能用 Diagram 窗格删除两个表间的连接。只能在 Diagram 窗格中改变连接而不能改变数据库本身。如果要改变两个表间的关系，需要使用 Table Designer 或 Database Designer。

改变连接类型

在连接上单击鼠标右键并从弹出菜单中选取 [Property Pages] 命令，就可改变连接类型。在 [Property Pages] 对话框中可以改变连接，使之包含连接表中其中一个（或两个）表的所

有行或仅包含所选列。可用的连接类型是 LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN 和 INNER JOIN。另外，通过将列比较符号改变为下列形式之一，也可以改变表的连接方式：=（等于）、<>（不等于）、<（小于）、<=（小于或等于）、>（大于）或>=（大于或等于）。当然，这里讨论的每个表中的列都是用于连接表的。出现 [Property Pages] 对话框时，可以从连接自身观察到表的连接方式。中间部分是比较符号。如果连接为 OUTER JOIN，那么方框会添加到钻石形状中。图 4.27 是代表连接的钻石和长方形形状的不同组合示例。

图 4.27 中的 4 种连接形状从左到右分别表示下列连接：

- FULL OUTER JOIN，连接不相等的列。
- FULL OUTER JOIN，连接相等的列。
- LEFT OUTER JOIN，连接相等的列。
- INNER JOIN，连接相等的列。



图 4.27 Diagram 窗格中描述的不同连接

如果在 Diagram 窗格中有许多表，那么可以选择只显示表的名称，并为更多的表保存一些真实状态。在表上单击鼠标右键并从弹出菜单中选取 [Name Only] 命令就可完成这项操作。在表上单击鼠标右键并选取 [Column Names] 命令将再次显示整个表。

对输出进行分组和排序

使用 Diagram 窗格，可以用任何普通的 SQL 语句对输出进行排序和分组。在表中的列上单击鼠标右键并选取 [Sort Ascending] 或 [Sort Descending] 就可对列进行排序。当根据指定列对输出排序时，排序符号就紧靠着列名。要取消排序，则可以在表中的列上单击鼠标右键并单击选中的排序顺序。

除了要先在 Query 工具栏上单击 [Group By] 按钮之外，分组方法几乎和排序一样。在图 4.28 中，[Group By] 按钮位于图中从右起的第二个位置。按下 [Group By] 按钮后，就可以选择要分组的列。当列是输出组的一部分时，分组符号和 [Group By] 按钮显示的符号一样都位于列名的最右端。再次按下 [Group By] 按钮则可以取消分组。请注意，分组是按照选择列的顺序进行的。就像任何 SQL 查询一样，至少应保证所有输出列是分组的一部分。



图 4.28 Query 工具栏

如果你对在 SQL 窗格中创建 SQL 语句还有疑问，那么使用 Diagram 窗格是较好的学习方式，因为对应于 Diagram 窗格中所做的改变，从 SQL 窗格中可以看到具有同样效果的 SQL 语句。

Grid 窗格

回想一下，Grid 窗格允许完成与 Diagram 窗格和 SQL 窗格所做的相同改变。但是，不能向 Grid 窗格中添加表。因此，为了向查询添加表，必须使用 Diagram 窗格或 SQL 窗格的功能。Grid 窗格的弹出菜单中没有 [Add Table] 命令，但是可以使用 [Query] 菜单中的 [Add Table] 命令来添加表。



当在不同的列中输入数据时，如果将输入点移动到另一行或列，那么就会验证或校验该数据。

Grid 窗格支持许多命名列（有点奇怪，是吗？），在表 4.2 中可以看到每个命名列的说明。请注意，对于不同查询类型而言，并不是所有的列都是可见的。通过检查该表中的 [有效查询类型] 列可以对特定查询而言该列是否有效。

表 4.2 Grid 窗格列的说明

列名	有效查询类型	说明
Column	所有查询类型	这是在 Table 列引用的表的列名
Alias	SELECT 、 INSERT Results 和 Make Table	如果给列赋予别名，那么输出列将命名为该别名。使用 Alias 列给输出赋予描述性的名称。别名也用于计算列，与 SQL 窗格中的 AS SQL 子句相同
Table	SELECT 、 INSERT Results 、 UPDATE 、 DELETE 和 Make Table	在此处将表的名称选到列所在位置。可以只从下拉列表中选择表。如果需要添加更多的表，使用 [Query] 菜单中的 [Add Table] 命令。如果列经过计算，那么该列应为空
OutPut	SELECT 、 INSERT Results 和 Make Table	该列用于指示列是否作为结果的一部分输出
Sort Type	SELECT 、 INSERT Results 和 Make Table	如果该列为空，则不对输出进行排序。要指定排序顺序，从下拉列表中选择 [Ascending] 或 [Descending]。这与 SQL 窗格中的 ORDER BY 子句相同。ASC 用于升序，而 DESC 用于降序
Sort Order	SELECT 、 INSERT Results 和 Make Table	如果要根据多列进行排序，那么在此处可以指定按照何种顺序排序。首先根据第一列排序，接着根据第二列排序，依此类推。这与 SQL 窗格中 ORDER BY 子句的列构成的列表一样
Group By	SELECT 、 INSERT Results 和 Make Table	与 SQL 窗格中 ORDER BY 子句相同
Criteria	SELECT 、 INSERT Results 、 UPDATE 、 DELETE 和 Make Table	不要指定列名，因为向带有正确表的行中添加了标准。如果要用 Or 操作符添加多个标准，那么应将每个标准都放在单独的 Or... 列中。如果要用 And 操作符添加标准，也应同样操作，如：>1 AND <5。这意味着结果集中包含了列（在 Column Name 列中指定）大于 1 且小于 5 的行
Or...	SELECT 、 INSERT Results 、 UPDATE 、 DELETE 和 Make Table	用于添加一个以上的标准。将每个标准添加到每个 Or... 列中，直到添加完所有标准为止
Append	INSERT Results	将返回行查询的结果追加到已存在的表中。在 Column 列中的结果值会追加到在目标表的 Append 列命名的列中。通常，如果可以断定目标列与源列 (Column) 相匹配，那么由 Query Designer 填写该列
New Value	UPDATE 和 INSERT VALUES	为 Column 列中指定的列赋予新值。新值可以是能求值的表达式，或是常数值

如果对在 SQL 窗格中创建 SQL 语句还有疑问，那么使用 Grid 窗格是较好的学习方式。因为对应于 Grid 窗格中所作的改变，从 SQL 窗格中可以看到具有同样效果的 SQL 语句。真是学无止境啊！

SQL 窗格

SQL 窗格 (pane) 用于输入自由文本格式的查询，这种文本基于所连接数据库的 SQL 标准。现在，多数关系数据库都将 ANSI SQL 标准作为附加功能的基础。本书不讨论 SQL 标准的细节，因为该主题需要用整本书来介绍。如果需要有关 SQL 标准的详细描述，推荐阅读数据库提供的帮助文件和/或文档，也可购买介绍该主题的书籍。

在 SQL 窗格中可以完成在 Grid 和 Diagram 窗格内能完成的任何任务。主要不同点在于：在其他两个窗格根据 SQL 窗格的内容进行更新之前，需要验证 SQL 语法（详情请参阅本章后面的“验证 SQL 语法”部分）。实际上，也可以将光标移动到 Diagram 或 Grid 窗格，从而完成这些窗格的更新，但是不会进行验证。可以在 SELECT 语句中输入无效列的名称，而且这些名称将在 Grid 窗格中显示。因此，当使用 SQL 窗格编辑查询时，应经常使用 Verify SQL Syntax（验证 SQL 语法）函数。这样可以避免许多麻烦。

Results 窗格

Results 窗格与 Query Designer 中的其他三个窗格有很大差异，这因为 Results 窗格并不用于编辑查询。正如其名称所暗示的一样，Results 窗格只是输出窗口。Results 窗格仅用于返回行的查询，如 SELECT 查询和带有统计函数（如 SELECT COUNT(*) FROM TableName）的标量查询。

在返回行的查询中，如果查询只从单个表返回行，那么可以使用 Results 窗格向源表添加新行。如果熟悉 MS Access，可能就会认出 Results 窗格中的栅格 (grid)，而且也知道在栅格最后一行（带有星号标记）中输入列值即可添加新行。如果要编辑查询的返回值，则可以在期望的列中输入新值。一旦输入了列值，只需将光标移到另一行，那么 Query Designer 就会立即更新数据库。如果在更新时发生错误，那么就会显示描述错误详情的消息框。单击 [OK] 按钮后，光标将位于发生错误的栅格的行中。

隐藏和显示不同窗格

所有窗格都可以隐藏，但是在显示 Query Designer 时，至少有一个窗格是可见的。要隐藏窗格，只需在窗格上单击鼠标右键，并从弹出菜单中选取 [Hide Pane] 命令。实际上，使用图 4.28 所示的 [Query] 工具栏（打开 Query Designer 时默认显示该工具栏）也可隐藏窗格。工具栏上的前 4 个按钮用于隐藏和显示 Query Designer 窗格。工具提示有助于了解按钮的作用。要查看工具提示，可将鼠标指针放在按钮上并保持一段时间。如果窗格被隐藏，则必须使用这些按钮来显示隐藏的窗格（与隐藏窗格不同，不能用弹出菜单中的命令来显示窗格）。

4.5.2 验证 SQL 语法

如果要确保查询有效，则可以使用 Verify SQL Syntax 函数。只有在使用 SQL 窗格编辑

查询时才需要这样做,而其他窗格会自动验证所做的修改。要实现这项任务,可以单击[Query]工具栏上的[Verify SQL Syntax]按钮,或者选取[Query]菜单中的[Verify SQL Syntax]命令。另外,可以在SQL窗格上单击鼠标右键并从弹出菜单中选取[Verify SQL Syntax]命令。验证完查询后便会显示一个消息框,该消息框告知用户查询有效,或者需要根据消息框中的说明修改查询。如果查询无效,可单击消息框中的[Help]按钮以获取问题的详细说明。

4.5.3 执行查询

创建完查询后,便可以执行该查询。要执行或运行查询,可以单击[Query]工具栏上的[Run Query]按钮,或选取[Query]菜单上的[Run]命令,或在Diagram窗格的空白处单击鼠标右键并从弹出菜单中选取[Run]命令。



警告 查询操作无法取消,因此,如果要从一个或多个表中删除行,则应确保这些行是无用的。

如果SQL窗格中的SQL语法无效,则该查询不会被执行。如果无法确定查询是否正确,那么在运行查询前尽量验证语法。详情请参阅本章前面的“验证SQL语法”部分。

4.5.4 分析不同查询类型

默认情况下,创建的查询为SELECT查询,但也可以单击[Query]工具栏上的[Change Type]按钮来改变查询类型。使用[Query]菜单中的[Change Type]命令也可改变查询类型。下面将简单说明IDE提供的所有查询类型,并给出示例(以SQL语句查询的格式)。下面是查询列表:

- SELECT
- UPDATE
- DELETE
- Make Table
- INSERT

4.5.5 SELECT 查询

SELECT查询用于从数据库的一个或多个表中返回行。有时,SELECT语句与INSERT查询组合使用,或作为INSERT语句的一部分,从而向已存在的表附加结果(INSERT Results查询)或向新表添加结果(Make Table查询,下面将会进行介绍)。

下面最简单的SELECT查询格式: `SELECT * FROM tblUser`。该查询会返回表tblUser中的所有行,并输出所有列(*)。

UPDATE 查询

UPDATE查询用于更新目标表中指定行的列值。查询 `UPDATE tblUser SET`

FirstName='Peter'将更新表 tblUser 中的所有行，并将 FirstName 列设置为“Peter”。如果只想更新指定行，则需要包括 WHERE 子句，如：

```
UPDATE tblUser SET FirstName='Peter' WHERE FirstName='John'
```

该查询只更新 FirstName 列的值为“John”的行。如果需要用 WHERE 子句更新匹配标准的行中的多个列，则可以用逗号分隔开这些列，如：

```
UPDATE tblUser SET FirstName='Peter',LastName='Johnson'WHERE FirstName='John'
```

每次只能更新一个表中的行。

DELETE 查询

如果要从表中删除一定的行，则最佳选择是 DELETE 查询。DELETE 查询非常简单，例如：DELETE FROM tblUser。该查询将删除表 tblUser 中的所有行。和更新查询一样，可以使用 WHERE 子句选择要删除的行。如：

```
DELETE tblUser WHERE FirstName='John'
```

该查询只删除 FirstName 列的值为“John”的行，但每次只能删除一个表中的行。

Make Table 查询

Make Table 查询实际上比其名称所暗示的要复杂些。虽然使用该查询时要创建表，不过该查询也可选择要插入到新表中的行。下例中的 INTO 键造成了这种差异，该查询创建了新表并将 tblUser 中的所有行都复制到新表 (tblTest) 中。如：

```
SELECT * INTO tblTest FROM tblUser
```

从该示例可以看出：SELECT 语句用于重新得到表 tblUser 中的行，并将其插入到新表 tblTest 中。如果在数据库中已经存在表 tblTest，则会发生错误。如果需要向已存在的表附加结果，则需使用 INSERT Results 查询。

Make Table 查询非常适于从源数据库复制特定行和/或列到临时表中，从而使得在临时表中操作不会干扰“真实”数据。Make Table 查询也用于备份一个或多个表中的数据，然后进行存储。

每次只能创建一个表。

INSERT 查询

有两种 INSERT 查询：一种用于将返回行的查询结果插入到另一个表中，另一种则用于将值插入到表中。

INSERT Results 查询

INSERT Results 查询用于向已存在的表中附加行。如果需要向新表添加行，那么必须使用 Make Table 查询。INSERT Results 查询的结构如下：

```
INSERT INTO tblUser (LoginName,FirstName,LastName>Password) _
SELECT LoginName,FirstName,LastName>Password FROM tblTest
```

在该查询中，从表 tblTest 中取回 LoginName、FirstName 和 LastName 列，并附

加到表 `tblUser` 中。源表 `tblTest` 中的列名不必和目标表 `tblUser` 相同，但是列的顺序则很重要。

只要列的数目和目标表的列数匹配，就可以根据需要从多个表中取回行。但目标表永远只有一个。

INSERT VALUES 查询

INSERT VALUES 查询与 UPDATE 查询在 SQL 语句的结构方面非常相似，如下所示：

```
INSERT INTO tblUser(LoginName,FirstName,LastName>Password)_  
VALUES('peterj','Peter','Johnson','password')
```

该查询将向表 `tblUser` 插入一条记录，该记录的 `LoginName`、`FirstName`、`LastName` 和 `Password` 列的值分别为“peterj”、“Peter”、“Johnson”和“password”。值（VALUES）的顺序必须与列（在第一组括号中指定）的顺序相匹配。如果你指定了所有列的值都要按照这些列在数据库中出现的顺序来排列，那么就可以省略表名后的字段名和括号。



每次只能向目标表中插入一行。

4.6 使用 SQL Editor 编辑脚本

SQL Editor 是个文本编辑器，与编写 VB 代码的编辑器相同。这意味着 SQL Editor 具有和代码编辑器一样的便利功能，如改变代码颜色和编制行号等。但是 SQL Editor 没有智能感应功能！当选取 [Tools] 菜单中的 [Options] 命令，打开 [Options] 对话框后，就可以改变编辑器的默认行为。展开 Text Editor 节点后的 [Options] 对话框如图 4.29 所示。

从图 4.29 可以看出：有几个选项可用来定制不同的 SQL 语言，如 PL/SQL（Oracle）和不同版本的 T-SQL（Microsoft SQL Server）。在 Database Tools 节点下也有一些与 SQL Editor 有关的选项。

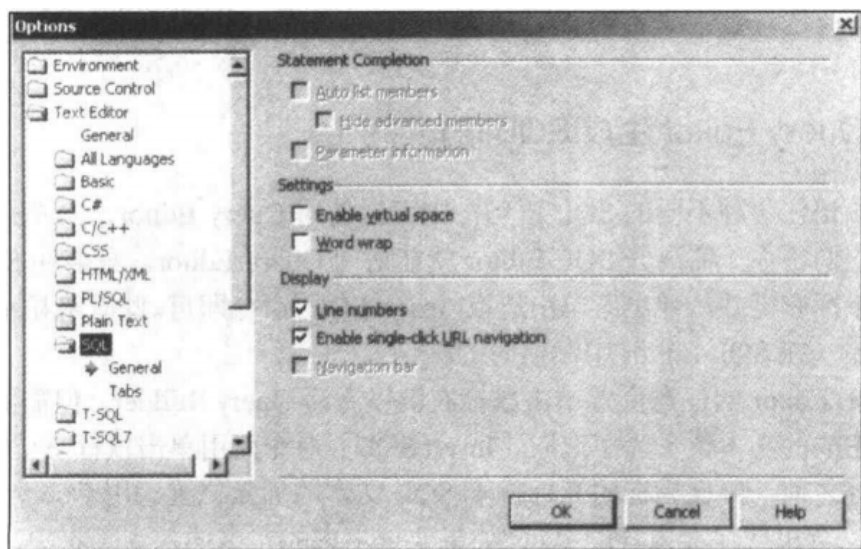


图 4.29 展开 Text Editor 节点后的 [Options] 对话框

练 习

向 Create Scripts 文件夹中添加 Create Table Script (创建表脚本), 命名为 Create SQL Server Table。有关向数据库工程添加新数据库对象的更多信息, 请参阅“添加新的数据库对象”部分。创建后的脚本, 应如图 4.30 所示。

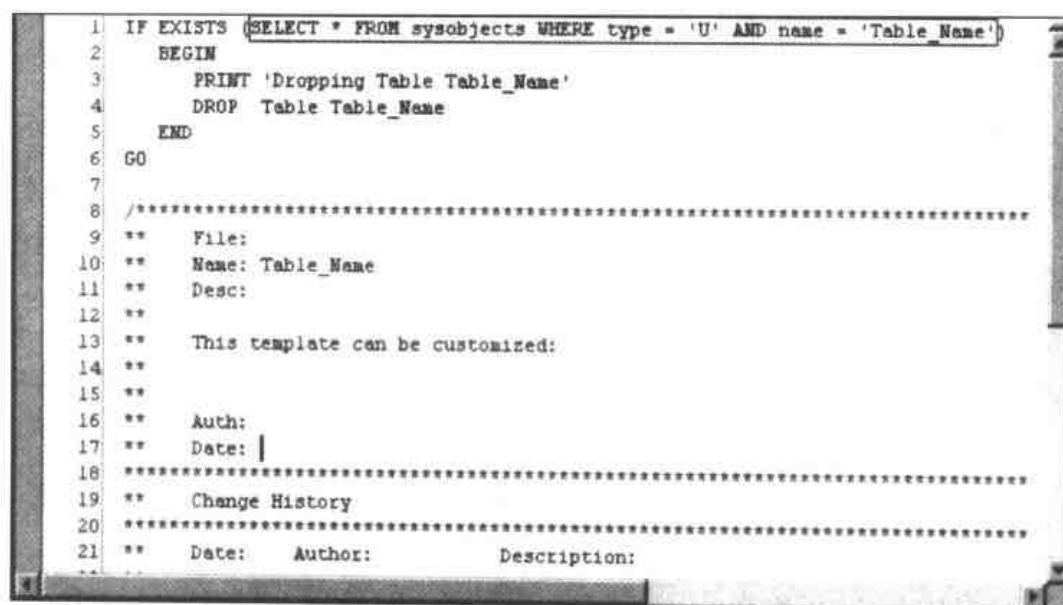


图 4.30 打开了 Create SQL Server Table 脚本的 SQL Editor

练 习

使用 Text Editor 的 Replace 功能 ([Ctrl] + [H]), 将所有出现字符串 Table_Name 的地方都替换为 tblTest。在 CREATE TABLE 语句下面的左括号之后、右括号之前输入下列文本: Id int PRIMARY KEY NOT NULL IDENTITY, Test varchar(50) DEFAULT('Test')。

4.6.1 使用 Query Editor 生成 SQL 语句

如果你无法记住各种不同的 SQL 语句, 则可以使用 Query Editor, 而不必在数据库的帮助文件中查询它们。你无需离开 SQL Editor 就能访问 Query Editor, 只需在 SQL Editor 中的任意位置单击鼠标右键并从弹出菜单中选取 [Insert SQL] 命令即可。此时将显示 Query Editor, 或 Query Builder (在 SQL Editor 中激活时的称谓)。

可以在 SQL Editor 的任意位置单击鼠标右键以激活 Query Builder, 但需要指出的是: 用 Query Builder 生成的文本位于为了选取 [Insert SQL] 命令而用单击鼠标右键的位置。因此, 在单击鼠标右键之前, 确保鼠标的光标位于 SQL 文本的插入位置。用 Query Builder 完成任务后, 关闭窗口, 会出现保存内容的提示。如果要把生成的文本插入到脚本中, 那么单击 [Yes] 按钮, 如果不想保存文本, 则单击 [No] 按钮。如果单击 [No] 按钮, 则将丢弃生成的文本,

而且无法再恢复！有关 Query Editor/Query Builder 的使用方法请参阅本章前面的“使用 Query Designer 设计查询”部分。

4.6.2 保存脚本

编辑完脚本后，或者完成了一些你不想因为不可预见的情况而丢失的操作后，就应该保存脚本。只需按下 [Ctrl] + [S] 便可保存脚本。也可使用菜单完成保存操作。在 [File] 菜单中选取 [Save Create Scripts\Create SQL Server Table.sql] 命令。该菜单命令显然是动态创建的，因此，如果你的脚本名称不同，那么菜单命令也会有所差别。

练习

保存脚本。

4.6.3 编辑和使用脚本模板

在这里选择为 MS SQL Server 创建脚本仅仅是因为默认模板就是 SQL Server 模板。使用任何能保存普通文本的文本编辑器(如 NotePad)进行编辑都能改变模板。模板位于 C:\Program Files\Microsoft Visual Studio.NET\Common7\Tools\Templates\Database Project Items 文件夹中。如果在安装时把 Visual Studio.NET 放在了不同驱动器和/或不同文件夹中，则需要相应改变模板的路径。

4.6.4 运行 SQL 脚本

创建了脚本后，在解决方案资源管理器中用鼠标右键单击该脚本，并从弹出菜单中选取 [Run] 命令就可运行该脚本了。从 [Run On] 对话框中选择期望的数据库连接或引用，并单击 [OK] 按钮。然后，脚本就将针对所选数据库连接而运行。

在老版本的 MS Visual Database Tools 中必须明确地为脚本指定连接，现在的版本则有不少改进。用户可以在运行脚本之前指定连接。需注意的是：并非所有脚本都能与任何数据库连接及引用相兼容。

实际上这并不是运行脚本的惟一途径。如果在 SQL Editor 中打开了脚本，那么在编辑器的任意位置单击鼠标右键，并从弹出菜单中选取 [Run] 命令也可运行脚本。如果没有保存对脚本所做的修改，那么在运行脚本之前会有保存修改的提示。

在脚本运行的时候，所有来自脚本的输出都会打印到输出窗口，默认情况下，输出窗口位于 SQL Editor 的下方。用户应该检查输出以观察脚本的执行是否正确。

4.7 创建类型数据集

如果要使用类型数据集 (typed data set)，则必须手工创建，或者借助于一些工具（有关

类型数据集和非类型数据集的说明请参阅第3B章)。总之,不能从代码中创建类型数据集!

在生成类型数据集的过程中,需要遵循下列这些步骤:

1. 得到或创建模式(schema)。
2. 生成数据集类。
3. 创建新生成的或继承的数据集类的实例。

前面所列的步骤中需解释的是:类型数据集实际上只是封装了数据访问的类,它提供了强类型(strong typing)、智能感应功能(IntelliSense feature)和编译时语法检查(compile time syntax checking),等等。

有三种工具可用来创建类型数据集(实际上是两种,因为 DataSet Designer 和 XML Designer 在本质上是相同的):

- Component Designer
- DataSet Designer
- XML Designer

本书没有介绍在 Component Designer 中创建类型数据集的方法。

4.7.1 使用 XML Designer 创建类型数据集

虽然可以使用 XML Designer 创建模式,不过使用 DataSet Designer 更加容易。DataSet Designer 实际上是带有一些额外功能的 XML Designer。因此,如果喜欢的话,也可使用 DataSet Designer。因为 XML Designer 和 DataSet Designer 的多数功能都是重复的,所以在这里只介绍使用 DataSet Designer 创建类型数据集的方法。

4.7.2 使用 DataSet Designer 创建类型数据集

实际上,可以使用 DataSet Designer 从头开始创建新模式,也可创建新表,甚至新数据库。但本书不会深入讨论这些内容,只集中介绍根据已存在的表创建模式。

练 习

打开文件夹 Chapter 4 里的 Typed DataSet Project。

如果打开的工程不是数据库工程,那么在解决方案资源管理器中的工程上单击鼠标右键并从 [Add] 子菜单中选取 [Add New Item] 命令,就可添加新的数据集。在 [Add New Item] 对话框中,选择 [DataSet] 模板,并在单击 [Open] 按钮之前为数据集命名。

练 习

打开 UserManDataSet.xsd Schema,并确保是在 Schema 视图而非 XML 视图中打开。单击 UserManDataSet.xsd 窗口底部的选项卡就可改变视图,参看图 4.31。

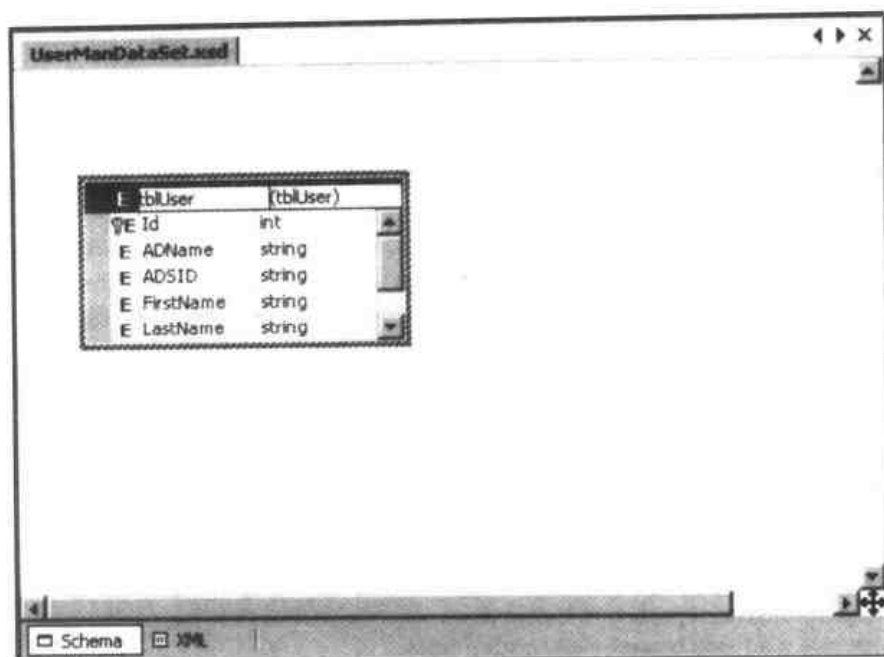


图 4.31 UserManDataSet.xsd 模式文件

练 习

打开服务器资源管理器，展开 UserMan database 节点和 [Tables] 节点，并将表 tblUserRights 拖拽到 DataSet Designer 窗口中。现在，在 DataSet Designer 窗口中有两个表。这时需要为新表命名，建议使用原始名：tblUserRights。单击新表头部的标题 E 旁的文本框，就可修改表名。删除当前文本并输入新的名称。单击 DataSet Designer 其他任意位置就可在内存中保存名称。按下 [Ctrl] + [S] 将把模式文件保存到磁盘。现在需要创建数据集，或者 Visual Basic DataSet 类文件。按下 [Ctrl] + [Shift] + [B] 开始创建过程，就可完成该任务。至此成功创建了类型数据集。



虽然 DataSet 类文件和模式文件一样可以在相同的文件夹中访问，但是在解决方案资源管理器中，DataSet 类文件是不可见的，而且不应该编辑该文件。在下次构建解决方案时，任何修改都将被覆盖。

显然，需要修改 Form1 Windows 格式类文件中的代码，以便使用添加的新类型表。不过，在第 3B 章中对此进行了介绍，因此这里不再讨论。快速浏览一下代码，查看在 button1_Click 事件中使用强类型的方法。

4.8 小结

本章介绍了 IDE 中与数据库相关的所有功能。讨论了保存服务器资源的服务器资源管理

器。此外，还特别讨论了数据连接和消息队列。本章的练习使你熟悉了创建带有数据库引用、脚本、查询和命令文件的数据库工程所要求的各个步骤。

本章还介绍了如何使用 Database Designer 创建数据库图表，以及如何使用 Table Designer 创建带有键、索引、约束和关系的表。并深入讨论了使用 SQL Editor 创建脚本和存储过程的方法。并在最后示范了创建类型数据集的方法。

下一章将介绍在应用程序中处理异常的方法。

第5章

错误处理

任何应用程序都需要处理发生在运行时的错误。要解决所有可能发生的错误并不是一件容易的事情。但是尝试创建代码以解决多数错误，却始终是一个好主意。因此，用户无需为它们担忧。而你应当捕获剩下的那些错误，并向用户显示带有详细信息的出错消息，以告诉用户如何从此处继续执行。错误处理程序的目的正是使得程序可以正常地从出错的地方恢复，这在处理关系数据库时特别重要，由于你要通过技术的多个层次，程序可能会在许多地方中断。尽管本章是关于与数据相关的异常的处理，但也会简要地介绍处理各类异常的基本知识。

在开始之前让我们先明确一件事情：对于在本章中所讨论的内容，错误和异常是相同的。因此，从此处开始，只使用“异常”一词。

常常听到这样的问题：所有程序都需要异常处理吗？回答总是否定的，因为一些程序非常简单，它们不使用可能溢出的变量或相似的量。这些程序可能调用几个非常简单的其他程序。在这种情况下，可以肯定不会发生应用程序异常，因而请不要在这些类型的程序中放入异常处理程序，那是浪费！虽然在何处使用异常处理程序是经验及个人偏好问题，但是如果你存有疑虑，就应在该处放上一个！而一旦确定了某处无需异常处理程序时，则应及时将它删除。

VB.NET有两种处理运行时异常的程序化方法：结构化的异常处理和未结构化的异常处理。

使用两种类型的异常处理

尽管可以同时使用未结构化和结构化两种异常处理，但建议你还是不要这样做，因为这只会使你的代码可读性更差，而且更难于维护和调试。值得注意的是，单个程序不能同时包含未结构化和结构化异常处理，因此即便你打算使用这两种方法，也只能在单个程序中使用其中的一种。

5.1 结构化异常处理

结构化异常处理是这样运行的：它将代码块标记为“已保护”，表示如果块中的任一行代码引发异常，都将被相关的异常处理程序捕获到。已保护的代码块使用 **Try...Catch** 语句标

记，这样已保护的代码必须插入到这些语句之间。这些语句是 **Try...Catch...Finally...End Try** 析构函数的一部分。这为你提供了一种控制结构，此结构可以监控已保护块中的代码，以便在引发异常时捕捉异常，并通过一个所谓的过滤器处理程序对这些异常进行定位。此控制结构也可以分离哪些可能引发异常的代码，即那些可能从“正常”代码中引发异常的代码，我个人很喜欢这种方法，因为它使代码读起来更为容易。

程序清单 5.1 展示了如何 将 **Try...Catch...Finally...End Try** 构造函数放在一起。

程序清单 5.1 Try...Catch...Finally...End Try 构造函数

```
' 下一行的 Try 语句启动异常处理程序
Try
... ' 这是已保护的代码块
Catch
... ' 这是放入异常处理代码的位置，代码可以
    解决异常并从异常中恢复。
Finally
... ' 始终执行此代码块
End Try
```

如果要使用结构化的异常处理，则必须使用 **Try** 和 **End Try** 语句（不读取任何参数）。除了这些语句外，至少还需要使用 **Catch** 或 **Finally** 语句中的一个，以下是不同代码块的不同之处：

- **Try block**: 这是放入要监测的各行代码的位置，因为它们可能引发异常。
- **Catch block(s)**: **Catch** 块是可选的（尽管并不明白去掉它的理由），你可以保留许多 **Catch** 块（如果你愿意），但是过滤器必须不同。本章后面的“过滤异常”部分中将介绍异常过滤。在放入异常处理代码（代码可以解决异常并无需通知可能的用户继续执行）的块中，**Catch** 块也称为失败处理程序。
- **Finally block**: **Finally** 块（可选的）是所有异常处理结束后应当立即运行代码行的位置，你可能会想：“请等一下，为什么不只是将那些代码行放在 **End Try** 语句后面呢？”你可以这样做，但它会使你的代码很难读懂。将它放在 **Finally** 块中，表示与 **Try** 块中已保护代码有某种关系。还有一点，如果你在 **End Try** 语句后放入此代码，而在处理异常的 **Catch** 块中有 **Exit Sub** 或 **Exit Function** 语句，那么它就不会执行。但相同情况下，若是将代码放在 **Finally** 块中，则会运行。因此，确实需要运行代码，不管 **Try** 块中是否会引发异常，就这样处理！**Finally** 块也称为最后处理程序。



在 **Try** 块中放入代码不会影响性能（即使没有引发异常）。

在处理已保护的代码块时，有三种异常处理程序类型：

- **Fault handler**: 在没有指定过滤器时，与 **Catch** 块相同。
- **Filter handler**: 当指定过滤器时，过滤器处理程序与 **Catch** 块相同。实际上有两种过滤器处理程序类型：

- *Type-filter handler*: 此过滤器处理程序用于处理指定类及其子类的异常。
- *User-filter handler*: 当你要指定异常的要求时, 用户过滤器处理程序用来处理具有任何异常类的异常。使用 **When** 键来完成。
- *Finally handler*: 与 **Finally** 块相同。

5.1.1 启用结构化异常处理

这个标题也许有点儿不确切, 因为你不是像处理未结构化异常处理那样真正“启用”结构化异常处理。相反, 它是将认为可能引发异常的代码放入已保护的块(称为 **Try** 块)中。放入此块中的所有代码都将被监控, 引发的所有异常也都被移植至 **Try...Catch...Finally...End Try** 构造函数的默认处理程序中。有关详细信息, 请参阅本章前面“结构化异常处理”一节的开头部分。

5.1.2 从正常代码中分离异常处理程序

这种方法在 VB.NET 中实现, 它可以巧妙地处理结构化异常。不管你如何处理代码, 异常处理程序都会自动从正常代码中分离出来。所有异常处理, 包括启用异常处理、捕获异常、处理异常, 以及最后运行清除代码, 均放在 **Try...Catch...Finally...End Try** 同一析构函数中。与未结构化异常处理相比, 这会使你的代码更容易阅读。

5.1.3 在同一程序中使用多个结构化异常处理程序。

对于未结构化的异常处理, 在相同的程序中, 可以使用多个结构化异常处理程序。你只需将代码封装在 **Try...Catch...Finally...End Try** 构造函数中即可(只要你觉得有必要)。但在多数情况下, 这可能会矫枉过正, 因为它可以过滤各种异常, 确定引发异常的原因, 然后从中恢复, 即使在 **Try** 块中具有许多行代码。我认为这只是个人偏好问题, 它可增强代码的可读性。程序清单 5.2 说明了如何在同一程序中同时使用两个结构化异常处理程序。

程序清单 5.2 在一个程序中的两个结构化处理程序

```
1 Public Sub TwoStructuredExceptionHandler()  
2     Dim lngResult As Long  
3     Dim lngValue As Long = 0  
4  
5     Try ' 第一个异常处理程序  
6         lngResult = 8 / lngValue  
7     Catch objFirst As Exception  
8         MsgBox(objFirst.Message)  
9     End Try  
10  
11    Try ' 第二个异常处理程序  
12        lngResult = 8 / lngValue  
13    Catch objSecond As Exception  
14        MsgBox(objSecond.Source)
```

```

15 End Try
16 End Sub

```

程序清单 5.2 中没有深奥的内容, 仅是用纯代码说明了如何将多个结构化异常处理程序包含在同一程序中。

5.1.4 检验异常类

Exception 类保留了有关上次抛出的异常的信息。这就是当在异常处理程序中捕获到异常时检验该类属性的原因。在表 5.1 中列出了这些属性。

表 5.1 Exception 类属性

名称	说明
HelpLink	该属性返回或设置与异常有关的帮助文件的链接, 其数据类型为 String , 必须用 URL 或 URN 来指定
InnerException	此只读属性返回内部异常的引用。当异常处理程序捕获到系统异常并将信息存储在新异常对象的 InnerException 属性中时, 通常使用此属性。然后新的异常对象用于引发新的异常(对用户友好), 如果捕获新异常的异常处理程序想知道原始异常, 则可以通过检验 InnerException 属性来完成
Message	此只读属性(可以被忽略)的数据类型为 String , 用于返回详细描述异常的出错消息文本
Source	Source 属性返回或设置 String 值, 用来保留应用程序或引发异常的对象名称。如果未设置此属性, 则返回的值就是字符串, 该字符串保留了抛出异常所在的汇编的名称
StackTrace	此只读属性(可以被忽略)返回堆栈跟踪(stack trace), 堆栈跟踪指出代码中引发异常所在的位置(确切的行号), 返回的数据类型为 String 。堆栈跟踪在抛出异常之前可以立即捕获
TargetSite	TargetSite 属性只读且返回数据类型为 MethodBase 的对象, 返回的对象包含引发异常的方法, 但是如果引发异常的方法不可用, 则可以从堆栈跟踪中获得。如果方法不可用且堆栈跟踪为空(Nothing), 则返回空值(Nothing)

Exception 类也具有表 5.2 中描述的非继承方法。

表 5.2 Exception 类方法

名称	说明
GetBaseException()	此方法返回引发原始异常的引用, 返回对象为 Exception 类型
GetObjectData(ByVal vobjInfo As SerializationInfo, ByVal vobjContext As StreamingContext)	GetObjectData 方法将有关引发的异常(必须序列化)的所有信息都设置为 vobjInfo

5.1.5 在异常处理程序中处理异常

一旦启用了异常处理程序，就要创建一些代码，以便在异常抛出时处理异常。如果你可以解决或者从异常中恢复，则可以继续执行，否则，必须通知用户知道并/或记录下此异常。

Catch 块是处理引发异常的位置。在其最简形式中，**Catch** 块也称为默认的处理程序，其最简形式不会指定过滤器，而且只用来捕获异常，如程序清单 5.3 所示。

程序清单 5.3 最简形式的 **Catch** 块，默认的处理程序

```
1 Public Sub SimpleCatchBlock()  
2     Dim lngResult As Long  
3     Dim lngValue As Long = 0  
4  
5     Try  
6         lngResult = 8 / lngValue  
7     Catch  
8         MsgBox("Catch")  
9     End Try  
10End Sub
```

在程序清单 5.3 中，只有一个 **Catch** 块，它将接收 **Try** 块中抛出的所有异常。这种形式的问题是你在没有指定想要 **Exception** 对象的副本，即无法访问抛出的异常。但这不完全正确，因为实际上你可以在 **Catch** 块中捕获它。请注意，将它指定为 **Catch** 语句的一部分非常容易，如程序清单 5.4 所示。

程序清单 5.4 带有默认异常对象的 **Catch** 块

```
1 Public Sub CatchBlockWithDefaultExceptionObject()  
2     Dim lngResult As Long  
3     Dim lngValue As Long = 0  
4     Dim objE As Exception  
5  
6     Try  
7         lngResult = 8 / lngValue  
8     Catch objE  
9         MsgBox(objE.ToString)  
10    End Try  
11 End Sub
```

程序清单 5.4 指定了需要的所有异常以及需要存储在 **objE** 变量中的引发异常的属性值。实际上，你所需做的只是将程序清单 5.4 中的第 8 行替换为：

```
8     Catch objE As Exception
```

这样会将 **objE** 变量作为类型 **Exception** 对象初始化，然后将引发变量的值存储在 **objE** 中。如果你要确切地指定由特殊 **Catch** 块处理的异常类型，则需要过滤异常。有关如何过滤异常的详细信息，请参阅本章后面的“过滤异常”部分。



如果异常处理程序中有多个 **Catch** 块, 则当相应的 **Try** 块中的代码引发异常时, CLR 会依次尝试。所说的尝试, 就是像在 **Select Case** 析构函数中一样, 从头至尾地检查各种 **Catch** 语句。若有 **Catch** 语句与异常相匹配时, 就执行该 **Catch** 块。如果没有 **Catch** 语句与异常相匹配, CLR 将向用户显示标准消息框, 以详细说明此异常。

那么, 当捕获了异常后又该如何处理呢? 为了能够从异常中恢复, 你需要知道引发的异常类型。表 5.3 列出了 CLR 提供的标准异常类型。

表 5.3 标准异常类型

异常类型	基本类型	说明	示例
<i>Exception</i>	Object	这是所有异常的基类。有关详细信息, 请参阅“查看异常类”部分	有关详细信息, 请参阅子类
<i>SystemException</i>	Exception	SystemException 类是由 CLR 引发的所有异常的基类。它是 CLR 引发的异常, 可以使用用户应用程序重新恢复, 是非致命异常 (nonfatal exceptions)	有关详细信息, 请参阅子类
<i>IndexOutOfRangeException</i>	SystemException	当你尝试访问数组中不存在的元素 (元素的索引超出了范围) 时, 此异常将由 CLR 引发。此类无法继承	请参阅程序清单 5.5
<i>NullReferenceException</i>	SystemException	若尝试引用无效 (空) 对象, NullReferenceException 异常就将由 CLR 引发。无效对象的值为 Nothing	请参阅程序清单 5.6
<i>InvalidOperationException</i>	SystemException	如果某方法所属的对象处于无效状态, 此异常使由该方法引发	请参阅程序清单 5.7
<i>ArgumentException</i>	SystemException	此异常类型为所有变量异常的基类。当引发异常 (如果存在) 时, 应当使用子类异常	有关详细信息, 请参阅子类
<i>ArgumentNullException</i>	ArgumentException	为不允许变量为空的变量提供空值时, 此异常由对象的方法引发	请参阅程序清单 5.8
<i>ArgumentOutOfRangeException</i>	ArgumentException	当一个方法中的一个或多个变量超出有效范围时, ArgumentOutOfRangeException 由该方法引发	请参阅程序清单 5.9

在程序清单 5.5 中, 尝试将引用的第四个元素设置给数组 **arrlngException**, 但是在该数组中只能设置三个元素, 因此引发了 **IndexOutOfRangeException** 异常。

程序清单 5.5 引发 **IndexOutOfRangeException** 异常

```

1 Public Sub ThrowIndexOutOfRangeException()
2     Dim arrlngException(2) As Long
3 
```



```
4 Try
5     arrlngException(3) = 5
6 Catch objE As Exception
7     MsgBox(objE.ToString)
8 End Try
9 End Sub
```

在程序清单 5.6 中，尝试引用 objException 对象，但是此对象还没有初始化，因此会引发 **NullReferenceException** 异常。

程序清单 5.6 引发 NullReferenceException 异常

```
1 Public Sub ThrowNullreferenceException()
2     Dim objException As Exception
3
4     Try
5         MsgBox(objException.Message)
6     Catch objE As Exception
7         MsgBox(objE.ToString)
8     End Try
9 End Sub
```

在程序清单 5.7 的第 18 行中，尝试使用 cmmUser 命令执行非行返回 (non-row-returning) 查询，但是此命令要求打开而且已经就绪的连接。因为当数据阅读器打开时，此连接 (cnnUserMan) 正被数据阅读器占用，因此会引发 **InvalidOperationException** 异常。

程序清单 5.7 引发 InvalidOperationException 异常

```
1 Public Sub ThrowInvalidOperationException()
2     Const strConnection As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4     Const strSQLUserSelect As String = "SELECT * FROM tblUser"
5     Dim cnnUserMan As SqlConnection
6     Dim cmmUser As SqlCommand
7     Dim drdUser As SqlDataReader
8
9     Try
10         ' 实例化并打开连接
11         cnnUserMan = New SqlConnection(strConnection)
12         cnnUserMan.Open()
13         ' 实例化命令
14         cmmUser = New SqlCommand(strSQLUserSelect, cnnUserMan)
15         ' 实例化并填充数据阅读器
16         drdUser = cmmUser.ExecuteReader()
17         ' 打开数据阅读器时执行查询
18         cmmUser.ExecuteNonQuery()
19     Catch objE As Exception
20         MsgBox(objE.ToString)
21     End Try
```

22 End Sub

在程序清单 5.8 的第 20 行中, 尝试使用数据适配器来更新数据源, 但是作为惟一变量的数据集未被实例化。由于数据集对象为空 (Nothing), 因而引发了 **ArgumentNullException** 异常。

程序清单 5.8 引发 **ArgumentNullException** 异常

```

1 Public Sub ThrowArgumentNullException()
2     Const strConnection As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4     Const strSQLUserSelect As String = "SELECT * FROM tblUser"
5
6     Dim cnnUserMan As SqlConnection
7     Dim cmdUser As SqlCommand
8     Dim dstUser As DataSet
9     Dim dadUser As SqlDataAdapter
10
11     Try
12         ' 实例化并打开连接
13         cnnUserMan = New SqlConnection(strConnection)
14         cnnUserMan.Open()
15         ' 实例化命令
16         cmdUser = New SqlCommand()
17         ' 实例化数据适配器
18         dadUser = New SqlDataAdapter(cmdUser)
19         ' 更新数据源
20         dadUser.Update(dstUser)
21     Catch objE As Exception
22         MsgBox(objE.ToString)
23     End Try
24 End Sub

```

在程序清单 5.9 的第 7 行中, 尝试显示连接字符串前面的 200 个字符, 但是连接字符串中没有那么多字符, 因而引发了 **ArgumentOutOfRangeException** 异常。

程序清单 5.9 引发 **ArgumentOutOfRangeException** 异常

```

1 Public Sub ThrowArgumentOutOfRangeException()
2     Const strConnection As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Try
6         ' 显示连接字符串前面的 200 个字符
7         MsgBox(strConnection.Substring(1, 200))
8     Catch objE As Exception
9         MsgBox(objE.ToString)
10    End Try
11 End Sub

```

现在,你知道了可以捕获的标准异常类型,但应当如何在 **Catch** 块中对它们进行处理呢?这要根据所谈论的具体异常以及捕获的方式来决定。首先需要查看如何过滤异常,以确定相应的 **Catch** 块可以处理相应的异常。有关详细信息,请参阅以下部分。

5.1.6 过滤异常

当设置异常处理程序时,最好知道期望何种类型的异常,尽管这并不总是可能的。但是,根据放入 **Try** 块中的代码,可以很容易地预测出部分可能会引发的异常。如果你调用的方法使用了一个或多个变量且移植的值是变量,就可以确定其中的某个变量超出了范围或设置为空。这将引发标准异常,你可以方便地使用类型过滤器进行过滤。有时要在过滤异常时,应用用户定义的标准,这种情况下则需要应用用户过滤。请注意,这两种类型的过滤可以配合使用。

类型过滤异常

类型过滤是按类型或类对异常进行过滤的方式,它可以确保技术上正确无误。如果在 **Catch** 语句中指定了一个类,则 **Catch** 块将处理此类及其所有子类。有关示例代码,请参阅程序清单 5.10。

程序清单 5.10 类型过滤异常

```
1 Public Sub TypeFilterExceptions()  
2     Dim arrlngException(2) As Long  
3     Dim objException As Exception  
4     Dim lngResult As Long  
5     Dim lngValue As Long = 0  
6  
7     Try  
8         arrlngException(3) = 5  
9         MsgBox(objException.Message)  
10        lngResult = 8 / lngValue  
11    Catch objE As NullReferenceException  
12        MsgBox("NullReferenceException")  
13    Catch objE As IndexOutOfRangeException  
14        MsgBox("IndexOutOfRangeException")  
15    Catch objE As Exception  
16        MsgBox("Exception")  
17    End Try  
18 End Sub
```

在程序清单 5.10 里, **Try** 块中有三个潜在的异常。各行代码中只有一行代码将被执行,因为代码将引发异常,并随之进入异常处理程序。你需要将不希望其引发异常的各行代码排除,然后再运行代码。

但是,程序清单 5.10 说明了如何包含多个 **Catch** 块,且各自执行不同的任务,或者处理完全不同的异常。当执行 **Try** 块中第 1 行代码(第 8 行)时,引发了异常。这表示 CLR 为此代码块查找可用的处理程序。它从第 11 行开始,但在第 8 行引发的 (**IndexOutOfRangeException**)

异常间没有发现匹配的语句，然后查找与其相匹配的第 13 行。结果会显示一个消息框，显示消息“`IndexOutOfRangeException`”。

请注意，在该示例的末尾处创建了一个“一般”异常处理程序，以捕获那些没有被其他处理程序捕获的异常。你不必这样做，而且只创建一个也经常不太合适。但是在某些情况下，当你不确定引发的异常类型时，最好创建此处理程序以捕获所有异常。当它捕获到未经处理的异常时，你可以向用户显示消息、记录异常，或者按自己的想法进行处理。

用户过滤异常

类型过滤是过滤异常的一种好方法，但有时候还不够。假设你要引用相同数据类型的两个对象，且知道它们有时为空（不取值）。则可以对 `NullReferenceException` 异常使用类型过滤器，但是该如何辨别两个对象中哪一个对象导致了异常呢？请参阅程序清单 5.11。

程序清单 5.11 用户过滤异常

```

1 Public Sub UserFilterExceptions()
2     Dim objException1 As New Exception()
3     Dim objException2 As Exception
4
5     Try
6         MsgBox(objException1.Message)
7         MsgBox(objException2.Message)
8     Catch objE As NullReferenceException When objException1 Is Nothing
9         MsgBox("objException NullReferenceException1")
10    Catch objE As NullReferenceException When objException2 Is Nothing
11        MsgBox("objException NullReferenceException2")
12    End Try
13 End Sub

```

程序清单 5.11 包含了两个 **Catch** 块，它们对相同的异常类型（`NullReferenceException`）进行过滤。现在，为了知道引发异常的是哪个对象，在类型过滤之后添加了一些用户过滤。**When** 关键词指定用户定义的标准，在此特殊的 **Catch** 块可以处理异常之前必须符合该标准。

在这种情况下，仅检查对象是否已被实例化。实质上，用户过滤器可以是任何标准。

5.1.7 创建自己的异常

有时，有必要创建自己定制的异常。也许你在创建一个类或组件以引发定制异常。在这种情况下，需要创建一个从 `ApplicationException` 类（它是由用户应用程序引发的异常类）继承的类。请注意，这只适用于引发非致命异常的情况（请参阅程序清单 5.12）。

程序清单 5.12 创建自己定制的异常类

```

1 Public Class UserManException
2     Inherits ApplicationException
3
4     Private prstrSource As String = "UserManException"
5
6     Public Overrides ReadOnly Property Message() As String

```

```
7      Get
8          Message = "This exception was thrown because you..."
9      End Get
10 End Property
11
12 Public Overrides Property Source() As String
13     Get
14         Source = prstrSource
15     End Get
16
17     Set(ByVal Value As String)
18         prstrSource = Value
19     End Set
20 End Property
21 End Class
```

在程序清单 5.12 中可以看到该如何创建自己定制的异常类。在显示的示例代码中替代了两个继承属性，并从基类（**ApplicationException** 类）中获得所有其他属性和方法。这是非常简单的异常类，但它确实说明了创建自己的异常类的基本方法。

5.1.8 引发结构化异常

如果需要引发异常，则可以使用 **Throw** 语句达到此目的。**Throw** 语句常用于检测异常处理程序，所用格式为：

Throw 表达式

要求的变量表达式是要引发的异常。下面的代码将引发新的 **IndexOutOfRangeException** 异常：

```
Throw New IndexOutOfRangeException()
```

现在，引发异常非常简单，这样它就可以用于常规交流。但是，如其名称所暗示的那样，异常适用于特殊情况。因此请不要将它们用于常规交流目的。如果要处理类，只需创建事件即可处理与客户或用户反馈的交流。

5.1.9 退出结构化的异常处理

有时，需要中途退出异常处理程序。这可以使用 **Exit Try** 语句来完成。**Exit Try** 语句的操作方式与 VB.NET 中的其他 **Exit** 语句类似，它们均可以退出部分类型结构。

- **Exit Sub**、**Exit Function** 或 **Exit Property** 退出并结束当前程序。
- **Exit Do**、**Exit For** 和 **Exit While** 退出并结束一个循环。
- **Exit Select** 退出并结束一个 **Select Case** 结构。

现在你应该已经明白了，所有这些 **Exit** 语句都退出某结构，**End...** 语句其后的一行则继续执行。对于 **Exit Try** 语句而言，它将从最后的处理程序（如果存在）那里继续执行，然后再执行紧随 **End Try** 语句其后的第一行代码。**End Try** 可以在所有异常处理程序块中（**Finally**

块除外)使用(请参阅程序清单 5.13)。

程序清单 5.13 退出结构化的异常处理程序

```

1 Public Sub ExitSEH()
2     Dim lngTest As Long = 1
3
4     Try
5         ' 检查以避免被 0 除的异常
6         If lngTest = 0 Then Exit Try
7
8         lngTest = lngTest \ 0
9     Catch objException As Exception
10        MsgBox("This code is executed when an exception is thrown.")
11    Finally
12        MsgBox("This code is ALWAYS executed.")
13    End Try
14
15    MsgBox("This code is executed after the SEH ends.")
16 End Sub

```

在程序清单 5.13 中创建了一些虚构代码,以说明该如何退出结构化异常处理程序。第 6 行检查了 `lngTest` 变量是否为 0。如果为 0,则退出结构化异常处理程序。如果在进入结构化异常处理程序之前,`lngTest` 变量的值没有改变,则执行第 8 行,因为此变量保留了值 1。事实上,第 8 行可能看起来有点愚蠢,但将它放在此处以显式引发被零除的异常,以表示只有 **Catch** 块会被执行。

尝试运行代码以查看显示的消息。然后将变量的初始值更改为 0,再查看得到了什么消息。

回想一下,**Exit Try** 语句可以放在结构化异常处理程序的任意块中,但如果放入 **Finally** 中的话,代码将无法编译。现在考虑一下,当你将 **Exit Sub**、**Exit Function** 或 **Exit Property** 语句放在 **Try** 块或一个 **Catch** 块中时,将出现什么现象?显然,程序会终止执行,而且该执行会传回程序的调用程序。但是 **Finally** 块会怎样呢,它会执行吗?请将程序清单 5.13 中第 6 行上的 **Exit Try** 语句更改为 **Exit Try**,并把 `lngTest` 变量的初始值设置为 0,然后再重新运行代码。你会发现执行了 **Finally** 块。这就是将必须运行的代码放在 **Finally** 块中而不放在结构化异常处理程序结束之后的理由!

5.1.10 处理与数据相关的异常

到目前为止,我们已经介绍了结构化异常处理的基本知识。现在该看看如何使用它处理与数据相关的异常了。如果你按照前面的说明没有在同一保护块中放入过多代码,则能很容易地指出是哪个与数据相关的程序或对象引发了异常。

程序清单 5.14 捕获 `SqlConnection` 类异常

```

1 Public Sub CatchSqlConnectionClassExceptions()

```

```

2  Const strConnection As String = "Data Source=USERMANPC;" & _
3    "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4  Const strSQLUserSelect As String = "SELECT * FROM tblUser"
5
6  Dim cnnUserMan As SqlConnection
7
8  Try
9      ' 实例化连接
10     cnnUserMan = New SqlConnection(strConnection & _
11        ";Connection Timeout=-1")
12  Catch objException As ArgumentException
13      If objException.TargetSite.Name = "SetConnectTimeout" Then
14          MsgBox(objException.StackTrace)
15      End If
16  End Try
17 End Sub

```

在程序清单 5.14 中，我们尝试用一个无效的 **Connection Timeout** 值实例化 **SqlConnection**。这显然会引发可捕获的 **ArgumentException**。因为知道此类异常相当普通，而且其他代码行也可以引发它，所以我们应该查看一下它是否是 **Connection**（第 13 行）**Timeout** 异常。此示例非常短小，但是你肯定能看到用此作为标题的地方。你需要的所有信息都在 **Exception** 对象或子类对象中，因此只需将它取出来即可。

如本章前面所示，这全部都是关于过滤异常的。本章将继续深入并向你说明如何捕获并过滤 ADO.NET 中每个与数据相关的类的各种方法，但为什么不只是坚持单凭经验的方法呢？你要做的工作是查找特殊方法并查看它可以引发什么类型的异常。设置异常处理程序过滤那些异常，然后将一些用户过滤添加到类型过滤中，或者对 **TargetSite.Name** 进行额外检查（如果有必要）。这将确保你使用正确的异常处理程序来处理异常。

5.2 结构化异常的 CLR 处理

如果你已经阅读了本章前面的部分，则应该对 CLR 如何处理异常有了清楚的认识，下面是对其具体操作的一个简介。

首先，CLR 使用保护的代码块和异常对象处理异常。如果引发异常，CLR 会创建异常对象并用有关异常的信息进行填充。

在每个可执行代码中都有一个关于异常的信息表，且可执行的每种方法都在该信息表中有一个相关数组。此数组保留异常处理的有关信息，但它可以为空。数组中的每个元素都描述一个保护的代码块、所有与此代码相关的异常过滤器以及所有异常处理程序。

异常表其实非常有效，因此它不牺牲处理器时间，也不占用内存。当引发异常时，显然要使用更多的资源，但是如果没有引发异常，总的开销和“正常”代码相同。

因此当引发异常时，CLR 启动异常处理过程。这种两步的处理过程从 CLR 搜索数列开始。CLR 查找第一个已保护的代码块，它保护当前执行的代码行，还包含了异常处理程序和与此异常相关的过滤器。

此处理过程的第二步取决于是否找到匹配语句。如果找到了，CLR 会创建描述此异常的异常对象，然后执行引发异常处的代码行间所有最后和/或默认的语句，以及处理异常的语句。你必须意识到异常处理程序的顺序非常重要，因为最里面的异常处理程序总是首先进行求值。

如果未找到匹配语句，则搜索调用的方法，如果在那里也未找到匹配的内容，CLR 就在转储堆栈跟踪后中断应用程序。

5.3 未结构化异常处理

未结构化异常处理正如名称所表明的一样，没有结构化。虽然你可以将它组织得很精细，且不在代码中设置许多跳转，但它仍是未结构化的。多数 Visual Basic 程序员对未结构化异常处理（`unstructured exception handling, UEH`）非常熟悉。随着 Visual Basic.NET 的发布，如今它仅仅是捕获并处理运行时异常的内建机制。建议仅将其作为一种更新旧代码的方法，这是因为未结构化异常处理会导致代码非常难于维护，更不用说调试了。因而在本书中提到的异常处理都是用结构化异常处理构建的。

5.3.1 启用未结构化异常处理

为了启用未结构化异常处理，必须使用 **On Error Goto <Label>** 语句（参阅程序清单 5.15）或者 **On Error Goto <LineNumber>** 语句（参阅程序清单 5.16）。其中，<Label>是当前程序中的定义标签，而<LineNumber>是当前程序中的行号。

程序清单 5.15 使用标签启用未结构化异常处理

```
1 Public Sub EnableUnstructuredExceptionHandling1()
2     ' 启用本地异常处理
3     On Error Goto Err_EnableUnstructuredExceptionHandlingFromLabel
4
5     Exit Sub
6 Err_EnableUnstructuredExceptionHandlingFromLabel:
7 End Sub
```

在程序清单 5.15 中，显示的行号是由 IDE 显示的，而不是由程序员提供的行号。

程序清单 5.16 使用行号启用未结构化异常处理

```
1 Public Sub EnableUnstructuredExceptionHandling2()
2     ' 启用本地异常处理
3     On Error Goto 5
4
5     Exit Sub
6 5:
7 End Sub
```

如果你没有在当前程序中定义标签，或者当前程序中不存在行号，就会发生编译时错误，如程序清单 5.15 和程序清单 5.16 所示。因此无法用 **On Error Goto <Label>** 或 **On Error Goto <LineNumber>** 语句跳转到不同的程序。



在此谈到的行号，不是指 IDE 直观显示的行号，它是文本编辑器提供的一项可选功能。这里所谈论的是用户可以自己添加的行号，就像其他标签一样，通常这种情况下的行号是一个标签。可以从程序清单 5.16 中看到，其中行号 5（显示在该示例的第 6 行）以名称“5”被作为标签键入（名称后面的冒号表示一个标签）。

从程序清单 5.15 和 5.16 中可以看出，程序的第 1 行启用了异常处理程序。虽然不是必须这样处理，但建议你将异常处理程序放在程序的开始部分。如果你经常要编写由成百上千行代码组成的程序，则很可能要在同一程序中包括多个异常处理程序。事实上，如果真要编写那么长的程序，就应该把程序分割成多个较小的程序。

请记住，异常处理程序仅从启用它的所在行被激活，这表示在启用异常处理程序之前，由程序中的代码所引起的任意异常都将被 CLR 捕获，而你则无法回应和/或解决该异常。

5.3.2 从正常代码中分离异常处理程序

当程序中具有未结构化异常处理程序时，有必要将其从正常代码中分离出来，因为如果不进行分离，即使没有引发异常，异常处理程序代码也可能被执行。在程序清单 5.15 和程序清单 5.16 中，异常处理程序被放在程序的末尾，并在其标签/行号前面的一行添加了 **Exit Sub** 语句，从而将其从正常的代码中分离出来。如果没有这么做，异常处理程序代码将和正常的代码一起被执行。

5.3.3 在同一程序中使用多个未结构化异常处理程序

由于可以在代码中使用多个未结构化异常处理程序，因而造成了程序的可读性问题（参阅程序清单 5.17）。

程序清单 5.17 同一程序中的两个未结构化异常处理程序

```

1  Public Sub EnableUnstructuredExceptionHandling3()
2      ' 启用本地异常处理 1
3      On Error Goto Err_EnableUnstructuredExceptionHandlingFromLabel1
4
5      ' 启用本地异常处理 2
6      On Error Goto Err_EnableUnstructuredExceptionHandlingFromLabel2
7
8  Err_EnableUnstructuredExceptionHandlingFromLabel1:
9      Exit Sub
10 Err_EnableUnstructuredExceptionHandlingFromLabel2:
11 End Sub

```

程序清单 5.17 中包含未结构化异常处理程序 `Err_EnableUnstructuredExceptionHandlingFromLabel1` 和 `Err_EnableUnstructuredExceptionHandlingFromLabel2`。当只有其中一个程序时，建议你开始异常处理程序的标签或行号放在“正常”代码（也就

是非异常处理代码)最后一行的后面。这样可以使代码更容易读取,因为它不会妨碍程序也不会干扰正常代码。

在同一程序中使用多个未结构化异常处理程序的问题所在是标签号码,以及需要放置的位置。程序清单 5.17 仅包含了两个处理程序,按照它们在程序中的启用顺序将其放在程序的末尾处。必须在代码中额外放置一个 **Exit Sub** 语句,否则,当执行第一个异常处理程序时,第二个异常处理程序也会同时执行。我在这里想说的重点是,即使使用了标签和 **Exit Sub** 语句,添加的异常处理程序越多,你的代码读起来也就越复杂。这确实是种看法问题,但是不明白为什么有人会喜欢这种方法,认为它比使用结构化异常处理程序更容易进行维护。

需要提到的另一点是,未结构化异常处理程序还会使得代码难于调试。这不是指 CLR,而是对用户要一步一步调试代码而言的。你需要先转到一个标签并跳回来,然后再转到另一个标签再跳回来。这样一步一步调试所有定制的程序确实是非常难的。

有关在代码中使用多个未结构化异常处理程序最值得注意的是:从启用第一个异常处理程序所在的那一行开始,直到启用下一个异常处理程序前面的那一行,这之间的代码都会引发某种异常,而第一个异常处理程序只处理这些异常。在程序清单 5.17 中,这表示在第 4 行引发的异常将导致执行第 9 行。如果在第 7 行引发了异常,则将在第 11 行继续执行。现在,考虑一下如果产生所谓的复式(spaghetti)代码,并因此从第二个异常处理程序跳到位于代码块中的代码中(该代码块由第一个异常处理程序服务)时,又会发生什么情况呢?参阅程序清单 5.18。

程序清单 5.18 从一个异常处理程序的代码中跳到另一个处理程序

```

1  Public Sub EnableUnstructuredExceptionHandling4()
2      Dim intResult As Integer
3      Dim intValue As Integer
4
5      On Error Goto Err_EnableUnstructuredExceptionHandlingFromLabel1
6      Goto SecondExceptionHandlerBlock
7 FirstExceptionHandlerBlock:
8      intValue = 0
9      intResult = 9 / intValue
10
11     On Error Goto Err_EnableUnstructuredExceptionHandlingFromLabel2
12 SecondExceptionHandlerBlock:
13     intValue = 0
14     intResult = 9 / intValue
15
16     Exit Sub
17 Err_EnableUnstructuredExceptionHandlingFromLabel1:
18     MsgBox("Err_EnableUnstructuredExceptionHandlingFromLabel1")
19     Exit Sub
20 Err_EnableUnstructuredExceptionHandlingFromLabel2:
21     Goto FirstExceptionHandlerBlock
22 End Sub

```

在程序清单 5.18 中启用了两个异常处理程序(第 5 行和第 11 行)。在启用 Err_

EnableUnstructuredExceptionHandlingFromLabel1 异常处理程序之后，跳转到 SecondExceptionHandlerBlock 标签（第 12 行），再由 Err_EnableUnstructuredExceptionHandling FromLabel2 异常处理程序服务的代码块中，通过与 0 相除（第 14 行）引发了异常。这表示 CLR 从第二个错误处理程序的第一行代码中（即第 12 行）继续执行。现在此行代码仅跳至 FirstExceptionHandlerBlock 标签，表示将执行第 8 行和第 9 行。在第 9 行通过与 0 相除引发异常后又会发生什么情况呢？第一个或第二个异常处理程序会被激活吗？请尝试运行此示例代码。

创建如此杂乱的示例是为了说明以下几点：

- 不要使用 **GoTo** 语句编写冗长的代码。可以用其他更为自然流畅的代码将其替换这些语句。
- 尽量保持异常处理的简单化。
- 请勿使用未结构化的异常处理，应该始终使用结构化异常处理。

显示的消息框表示 CLR 知道什么样的代码行属于什么样的异常处理程序。因此，即使启用了第二个异常处理程序，执行由第一个异常处理程序服务的代码行也会激活第一个异常处理程序，而不是第二个。

5.3.4 使用父异常处理程序

如果你的程序启用了未结构化异常处理程序，父异常处理程序（parent exception handler）也将被激活。如果在代码块（由异常处理程序服务）中调用的程序引发了异常，就会发生这种情况（参阅程序清单 5.19）。

程序清单 5.19 使用父异常处理程序

```

1  Public Sub UsingParentExceptionHandling()
2      On Error Goto Err_EnableUnstructuredExceptionHandling
3
4      UsesParentExceptionHandling()
5
6      Exit Sub
7 Err_EnableUnstructuredExceptionHandling:
8      MsgBox("Err_EnableUnstructuredExceptionHandling")
9  End Sub
10
11 Public Sub UsesParentExceptionHandling()
12     Dim intResult As Integer
13     Dim intValue As Integer
14
15     intValue = 0
16     intResult = 9 / intValue
17 End Sub

```

程序清单 5.19 在 UsingParentExceptionHandling 程序的第 2 行启用了 Err_EnableUnstructuredExceptionHandling 异常处理程序。在由异常处理程序服务的代码块中，会调用 UsesParentExceptionHandling 程序，而不启用异常处理程序。

这表示在执行程序中的第 18 行时会引发异常。因为该程序没有自己的异常处理程序，所以 CLR 会进行检查以确定调用的程序是否具有一个异常处理程序。它的确有一个，因此在调用该异常时，会显示包含文本“Err_EnableUnstructuredExceptionHandling”的消息框。

5.3.5 禁用未结构化异常处理

如果要在当前程序中禁用异常处理，则可以通过执行 **On Error Goto 0** 语句实现。它会禁用当前程序中的所有异常处理程序。请注意，如果引发异常，此语句不会通知 CLR 跳到行号为 0 的一行。即使程序中确实有行号为 0 的一行，也是如此。如程序清单 5.20 的第 11 行所示。

程序清单 5.20 禁用未结构化的异常处理

```

1  Public Sub DisableUnstructuredExceptionHandling()
2      ' 启用结构化异常处理
3      On Error Goto 5
4
5      ' 禁用结构化异常处理
6      On Error Goto 0
7
8      Exit Sub
9  5:
10     Exit Sub
11  0:
12     MsgBox("Line Number 0 has been reached!")
13 End Sub

```

当在一个程序中禁用结构化异常处理（如程序清单 5.20）时，也将从总体上禁用未结构化异常处理，而不仅是在执行 **On Error Goto 0** 语句的程序位置。这说明在引发的异常被递交至应用程序但还没被 CLR 处理之前，就必须启用异常处理。

5.3.6 禁用非结构化、本地异常

如果禁用程序中引发的异常，则表示该异常将被忽略，可以用 **On Error GoTo -1** 语句实现。它指示 CLR 忽略当前程序中的异常，以及它所调用的没有异常处理程序的程序中的异常。CLR 确实尝试要在异常之后进行清除，但是它不会向应用程序递交异常，也不会显示详细说明异常的消息框。



与 **On Error GoTo 0** 语句的操作相比，如果引发异常，**On Error GoTo -1** 语句确实不会通知 CLR 跳到行号为 -1 的一行。即使程序某行的行号为 -1，也是如此。请注意，这仅对当前程序有效，因为一旦程序结束，就会返回正常的异常处理。

5.3.7 忽略异常并继续执行

有时，仅忽略错误且在中断的情况下继续执行是个不错的主意，**On Error Resume Next**

语句可以帮助你完成此操作。如果在程序中放入了该语句，就从引起异常的语句之后的行开始继续执行（请参阅程序清单 5.21）。

程序清单 5.21 忽略异常

```
1 Public Sub IgnoreExceptions()  
2     Dim intResult As Integer  
3     Dim intValue As Integer  
4  
5     On Error Resume Next  
6  
7     ' 引发异常  
8     intValue = 0  
9     intResult = 9 / intValue  
10  
11     MsgBox("Was an exception thrown?", MsgBoxStyle.Question)  
12 End Sub
```

当程序清单 5.21 的第 9 行引发异常时，程序将在第 10 行继续执行，并显示消息框。只有这样才能确保所有 **On Error Resume Next** 语句都被排除，否则将不会捕获到多个异常！

注意，**On Error Resume Next** 语句仅作用于放置其中的程序。当你调用其他程序时，那些程序不会继承该忽略行为。如果调用的程序启用了异常处理程序，则该处理程序将捕获调用程序中引发的全部异常。但是，如果调用的程序未启用异常处理程序，则会在调用的程序中停止执行，而异常被传回受到忽略的调用程序中。

5.3.8 在异常处理程序中处理异常

一旦启用了异常处理程序，就应当创建一些引发异常时处理异常的代码。如果可以解决异常，就能继续执行，否则必须通知用户知道，并记录下该异常。

如果可以解决异常，则使用 **Resume** 语句从引发异常的代码行继续执行，其结果是该行被重新执行（参阅程序清单 5.22）。

程序清单 5.22 解决异常

```
1 Public Sub ResolveException()  
2     On Error Goto Err_EnableUnstructuredExceptionHandling  
3  
4     Exit Sub  
5 Err_EnableUnstructuredExceptionHandling:  
6     ' 解决异常  
7     ...  
8  
9     ' 通过重新尝试冲突的代码行来继续执行  
10     Resume  
11 End Sub
```

有时无法从引发异常的代码行继续执行，但是你已经异常处理程序中采取了其他动

作。这时便可以使用 **Resume Next** 语句，在不执行冲突代码行的情况下继续执行这些操作。更具体地说，就是在紧随冲突代码行后面的代码行启动执行（请参阅程序清单 5.23）。

程序清单 5.23 围绕异常进行操作

```

1  Public Sub WorkingAroundAnException()
2      On Error Goto Err_EnableUnstructuredExceptionHandling
3
4      Exit Sub
5  Err_EnableUnstructuredExceptionHandling:
6      ' 采取其他操作处理异常
7      ...
8
9      ' 从下面的行开始继续执行
10     Resume Next
11 End Sub

```

关于未结构化异常处理程序非常重要的一点是，如果新异常是由异常处理程序的部分代码（程序清单 5.23 中的第 6 行至第 10 行）引发的，异常处理程序则无法处理该异常，它会被传回调用程序。这可能会导致某些意想不到的结果，因此，在构造异常处理程序时必须消除引发新异常的危险，至少要将其降到最低程度。

5.3.9 检验 Err 对象

Err 对象保留了有关最新发生的异常信息。这就是当异常处理程序捕获到异常时，检验此对象属性的原因。



注意

Err 对象仅用于处理使用 **On Error GoTo <Label>** 语句捕获到的异常。

由于 **Err** 对象在每次引发新异常时都会“溢出”，因此保存该对象的所有信息非常重要。**Err** 对象的属性如表 5.4 所示。

表 5.4 Err 对象属性

名称	说明
<i>Description</i>	此属性返回或设置 String 值，并对引发的异常进行说明
<i>Erl</i>	此属性返回引发异常的行号
<i>HelpContext</i>	HelpContext 属性用来表示 HelpFile 属性所指定的文件中的特定帮助主题。如果 HelpFile 和 HelpContext 两个属性均有效，或者两个帮助文件均存在且帮助文件中都有主题，则显示帮助主题
<i>HelpFile</i>	HelpFile 属性返回或设置有效帮助文件的完整有效路径。在显示出错消息对话框时，用户按下键盘上的 [F1] 键或单击 [Help] 按钮，都会显示此帮助文件。应当将 HelpFile 属性和 HelpContext 属性配合使用，以确保特定的异常号码会指向相应的帮助主题

续表

名称	说明
<i>LastDLLError</i>	此只读属性返回调用 DLL 时引发的系统生成的异常，它仅在从 VB 代码中调用 DLL 时进行设置。系统对引发异常的 DLL 的调用在此属性中没有反映。该属性对在进行设置时，它不会抛出异常。这表示在调用 DLL 后应立即检查此属性，以确保能捕获所有引发的异常
<i>Number</i>	Number 属性是 Err 对象的默认属性，它返回或设置表明引发异常的一个数字值
<i>Source</i>	此属性返回或设置 String 值，表明生成或引发异常的对象

Err 对象也具有表 5.5 中描述的两种非继承方法。

表 5.5 Err 对象方法

名称	说明	示例
<i>Clear()</i>	Clear 方法用于清除 Err 对象，这表示所有属性都将被重新设置。当遇到 Resume 、 Resume Next 、 Resume <LineNumber> 、 Exit Sub 、 Exit Function 、 Exit Property 以及所有 On Error xxx 语句时，该方法会被自动激活	<code>Err.Clear()</code>
<i>Raise(ByVal vintNumber As Integer, Optional ByVal vobjSource As Object = Nothing, Optional ByVal vobjDescription As Object = Nothing, Optional ByVal vobjHelpFile As Object = Nothing, Optional ByVal vobjHelpContext As Object = Nothing)</i>	Raise 方法用于在应用程序中引发异常。当检测异常处理程序，或者要让调用程序知道某处出错时，这点会很有帮助。所有变量（ vintNumber 除外）都是可选的。请查看相应的属性以获取有关说明。请注意，如果你没有设置所有的属性，而且没有设置的属性已经取了值，则这些值将用于引发异常的一部分。请始终使用 Clear 方法手动设置属性，以确保不会引发带有无效值的异常	<code>Err.Raise (vbObject Error + 9)</code>

Err 对象是一个全局对象，无需创建实例即可使用。调用其属性或方法非常简单，如：
`intExceptionNum = Err.Number` 或 `Err.Clear()`。

5.3.10 引发未结构化异常

如果基于某种原因需要引发异常，则可以使用 **Err** 对象的 **Raise** 方法来完成。实际上，**Error** 语句也可以引发异常，但它是比较旧的语句且兼容性不好，因此不要使用。

引发异常有两个主要原因：检测异常处理程序或通知调用程序用户对象中引发了异常。不管是什么原因，引发异常都非常简单。

引发系统异常

如果要模拟系统异常，请使用系统异常（**System exception**）的保留号码。这表示你必须

调用 **Raise** 方法（带有 0~512 之间的一个数字），如程序清单 5.24 所示。

程序清单 5.24 引发系统异常

```
1 Public Sub RaiseSystemException(ByVal vintExceptionNum As Integer)
2     Err.Raise(vintExceptionNum)
3 End Sub
```

如果你调用 `RaiseSystemException` 程序的 `vintExceptionNum` 变量值为 5，将引发可怕的“无效的程序调用或变量”异常。当向程序传递无效的变量值时，通常都会引发此异常。有关其他系统错误号码，请参阅说明文档。

引发有效用户异常

如果你要引发的异常应当识别为用户定义的接收异常，可以将 `vbObjectError` 常量的值添加到错误号码中。此常量的值为 -2147221504，这使它适于在引发异常时模拟用户异常（请参阅程序清单 5.25）。

程序清单 5.25 引发用户定义的异常

```
1 Public Sub RaiseUserException(ByVal vintExceptionNum As Integer)
2     Err.Raise(vbObjectError + vintExceptionNum)
3 End Sub
```

在程序清单 5.25 中，你可以调用带有任意数字的 `RaiseUserException` 程序，而奇数通常与系统错误号码（0~512）重叠。这是因为在引发异常之前，传递的值里添加了 **vbObjectError** 常量。当你提出此类异常时，而且自己的异常处理程序也设置为处理此类异常，这时很重要的一点，就是要确定该异常是否用户定义的异常，或者是否是系统异常。有关如何进行处理的信息，请参阅下一节“确定是否引发用户定义的异常”。

5.3.11 确定是否引发了用户定义的异常

当在异常处理程序中捕获了异常后，就要弄清楚它是系统异常（号码 0~512），还是用户自己定义的。你所需做的是返回到前面，并从 **Err** 对象的 **Number** 属性中抽出 **vbObjectError** 常量（参阅程序清单 5.26）。

程序清单 5.26 确定用户定义的异常

```
1 Public Sub DetermineUserException(ByVal vlngExceptionNum As Long)
2     Dim lngExceptionNum As Long
3
4     lngExceptionNum = vlngExceptionNum - vbObjectError
5
6     ' 确定是否是用户定义的异常号码
7     If lngExceptionNum > 0 And lngExceptionNum < 65535 Then
8         MsgBox("User-defined Exception")
9     Else
```



```

10     MsgBox("Visual Basic Exception")
11     End If
12 End Sub

```

程序清单 5.26 保存了 `vbObjError` 常量的不同之处。在第 7 行通过比较，确定了保存的值是否在 0~65535 范围之内。如果在这个范围之内，那么异常号码便是用户定义的，否则就是 Visual Basic 系统错误。

5.3.12 捕获 DLL 中发生的异常

当使用 **Declare** 语句通过 API 调用来调用 DLL 时，应该检查调用的程序是否出错。因为 DLL 不提供异常信息，所以需要检查返回值。如果该值是一个意外值，则应当检查 **Err** 对象的 **LastDLLError** 属性。此处谈到的 DLL，指的是“无格式”（plain-vanilla）DLL 而非 ActiveX 或 COM DLL。

5.3.13 处理与数据相关的异常

到此为止，我们已经介绍了未结构化异常处理的基本知识，现在该看你如何使用该类型处理来处理与数据相关的异常了。程序清单 5.27 展示了一个示例。

程序清单 5.27 处理与数据相关的异常

```

1  Public Sub CatchOpenConnectionException()
2      Const strConnection As String = "Data Source=USERMANPC1;" & _
3          "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5      Dim cnnUserMan As SqlConnection
6
7      On Error Goto Err_EnableUnstructuredExceptionHandling
8
9      ' 实例化并打开连接
10     cnnUserMan = New SqlConnection(strConnection)
11     cnnUserMan.Open()
12
13     Exit Sub
14 Err_EnableUnstructuredExceptionHandling:
15     MsgBox(Err.Description & vbCrLf & Err.Erl & vbCrLf & Err.Number & _
16         vbCrLf & Err.Source)
17 End Sub

```

运行程序清单 5.27 中的示例代码，你将看到一个显示下列值的消息框：

Timeout expired	(Err.Description)
0	(Err.Erl)
5	(Err.Number)
SQL Server Managed Provider	(Err.Source)

如果进一步查看这些值，便会发现其中没有一个是惟一的。它们每一个都可以有不同的异常复制，因此很难解决该异常。程序清单 5.27 仅实例化并打开了连接，但是在多数程序中，你所要进行的操作并不止这一些。

当 **Err** 对象的属性值不惟一时，该如何确定导致异常的原因呢？解决方法有如下几种：

- 为能够进行逻辑分组（例如对象实例化和打开）的每个代码块启用新的异常处理程序。
- 通过使用 **Err** 属性的值，来获得确切的冲突代码行（假设已经用行号标签设置了代码）。
- 检查 **Number** 属性。

上述方法又有什么问题呢？频繁使用新的异常处理程序会使代码显得杂乱无章，进而影响读者对代码的理解。使用 **Err** 属性则表示在每次添加或删除程序中的某行时，都必须更改异常处理程序，且还必须在每个非空行上保留行号标签。而 **Number** 属性不是惟一的，也就是说多个异常将使 **Number** 的属性变为 5（请参考前面的程序清单 5.25 和解释该程序清单的文本）。

事实上，只要将额外的代码添加到异常处理程序中，就可以确定是什么导致了异常，但是此处的意思是，它只是编写了太多代码而已。因此在这种情况下，可以将 **Source** 和 **Description** 的值组合起来以确定异常的原因，但是，这是为什么呢？

这里有一点要说明：当使用结构化异常处理时，你可以从 **Exception** 对象获得值（要是可以的话）。为什么直到现在才告诉你呢？因为，如果你要使用新的 **Exception** 对象，它在其信息中的描述比 **Err** 对象的更详细，为什么不保持一致，仅使用结构化异常处理呢？因为这样你还可以更好地控制异常处理程序和异常。让自己忘掉未结构化异常处理吧！

5.4 小结

本章向你介绍了如何使用结构化异常处理以及未结构化异常处理。以及为什么不要对 VB6 或其较早的兼容版本的代码使用未结构化异常处理。

本章还讨论了以下几点：

- 当你使用未结构化异常处理时，**Err** 对象会保留有关异常的信息。
- 如果使用结构化异常处理，将会处理 **Exception** 类或其子类。

下一章将向你介绍如何处理服务器上的存储过程、触发器和视图。也就是说，将向你介绍如何代替客户处理服务器上的数据。如果能正确地实施，它将极大地提高系统性能。

第 6 章

存储过程、视图与触发器的使用

在 SQL Server 2000 中使用存储过程、 视图与触发器的方法

服务器端处理（Server-side processing）就是让服务器处理查询或类似事务，你可能听说过这个概念，而它在某种程度上也正是本章的主题。本章将从三个方面讲解如何进行服务器端处理：存储过程、触发器以及视图。服务器端处理的优点在于可以使用服务器的能力和资源来处理查询，这样客户机就可以空闲出来做其他事情。这样做虽然不总是合适，但在很多情况下是有利的。

本章包含一些有关创建存储过程、视图以及触发器的实际练习。请参考在文中出现的练习项目。

本章主要讲解 SQL Server 2000 的特性，有些功能性可能会与 MS Access 和 SQL Server 的早期版本相同。然而，如果你对 MS Access 并不熟悉，则建议你先阅读《From Access to SQL Server》，作者 Russell Sinclair，ISBN：1-893115-240。

6.1 优化因素

当提到应用程序的性能优化时，要考虑很多事情。但在这之前，你先要弄清一件事：这里提到的应用程序是分布式应用程序，而不是在一台单独 PC 上运行的独立应用程序，这种独立应用程序也称为独立绑定程序或单一应用程序。在这里讨论的应用程序使用某种网络来访问数据以及商业服务。

好了，讨论过了基础问题，现在就来集中注意力处理那些可能导致性能下降的因素，以及开始优化处理的时候怎样让你很好地了解这些因素。在设计应用程序时要时刻注意这些障碍。当各种资源（比如网络带宽、处理器能力、可用的 RAM，等等）发生改变时，应考虑你的应用程序是否也需要做相应改变。

表 6.1 列出了所有能够影响应用程序性能的因素，这些内容足可以写一整本书。虽然本书仅简要地介绍一下，但还是希望能够引起你的注意。它们对你设计应用程序时应选择什么服务器端处理资源有着极大的影响。

表 6.1 性能资源优化

资源名称	说明
网络资源	这里所说的网络资源指的是网络的实际带宽。考虑你的网络设置（你是处在局域网中，还是通过类似 Internet 的广域网来访问资源），如果你的带宽比较窄，则要求网络上传输的数据要尽量地少。反过来，如果你有足够的带宽，则可能希望在网络上传输大量的数据。然而，最实际的情况是，无论你有多大带宽，都要求只在网络上传输所需的数据
本地处理资源	如果本地机有可以利用的资源，则大部分数据处理工作都可以在那里完成。但要提醒一下，这完全取决于带宽以及服务器的处理资源
服务器处理资源	如果服务器有足够的资源，则使用服务器端处理就相当可取。如果使用服务器进行数据处理，那么就要考虑服务器的资源是否能够对所有客户服务

表 6.1 仅仅是一个概述。表 6.2 展示了一些不同的应用场景（Scenario）。

表 6.2 不同的应用场景

客户机	服务器	网络	建议
有限的处理资源	大量的处理资源	有限的带宽	在这种情况下，可以使用服务器的资源来处理查询以及类似的工作，并在完成后只返回需要的数据。这样做能够节省网络以及客户机的资源，但会限制应用程序的规模
大量的处理资源	大量的处理资源	有限的带宽	放在客户机或是服务器上处理都是可行的，但这要取决于将要在网络中传输的数据数量。如果数据量小，则在任何一方进行处理都可以，但如果数据量很大，就要让服务器来进行处理。另一个解决方案是将数据存储在本地的之后，再使用副本或批处理来更新服务器
大量的处理资源	有限的处理资源	有限的带宽	在这种情况下，处理应该由客户机来完成，但这实际取决于需要在网络上进行传输的数据量。如果数据量小，就该客户机做处理工作。但如果数据量很大，就要考虑让服务器做些工作
大量的处理资源	有限的处理资源	足够的带宽	在这种情况下，处理应该由客户机来完成

虽然还能向表中添加很多应用场景，但这些已足够让你对各种因素的影响有个大概的了解了。而且，在实际应用中很少会遇到用一个简单的方法就可以符合要求的简单场景。你的工作就是分析所有在设计应用程序时可能出现的潜在问题（Potential issue），并决定该在什么地方处理数据。

在下面的章节中，我们将讲解存储过程、视图以及触发器。现在，让我们开始吧！

6.2 存储过程的使用

存储过程 (stored procedure, SP) 是存储在数据库服务器上预编译好的 SQL 语句的批命令。存储过程在很长时间里一直都是一直让服务器处理数据的好方法。它能显著地减少客户机的工作负担,一旦你了解了它,就会为以前没使用也可以进行管理感到惊奇。

有关存储过程,不仅仅是上面提到的那些内容,但那一点确实是存储过程最重要的功能。想想看:使用它可以将 SQL 语句组合起来并存储在数据库服务器上,而且一次调用就可以执行。由于存储过程是预编译好的,因而省下编译时间并缩短执行时间。此外,存储过程还可以由任意数量的用户来执行,这意味着通过调用存储过程而不是每次都发送整个 SQL 语句可以节省很多带宽。

存储过程中可以包含数据库服务器所能理解的任何 SQL 语句。即你可以使用存储过程来完成各种任务,拿查询来说,既能进行动作查询 (action query),如 DELETE 查询,也能进行返回行查询 (row-returning query),例如 SELECT 语句。

使用存储过程还可以维护数据库。当服务器不那么繁忙的时,可以用存储过程来执行做清洁工作的 SQL 语句,这样就能省下手工操作的时间和精力。本章虽然没有提到维护任务,但它们也很重要,何况,你应该了解能使用存储过程完成的各种任务。如果你曾经或正在为一家没有 DB 管理员的小公司工作,那么就需要由你来负责数据库服务的正常运行。当然,这不是一个理想的情形,但你必须要比一个程序员从更多方面了解 DBMS,而且这一点也不是那么糟糕。

总结一下:存储过程是预编译好的一条令或一组存储在数据库服务器上的 SQL 命令。所有的处理都由服务器完成,结果则返回到客户机。

6.2.1 为什么要使用存储过程?

在下列情况下应使用存储过程 (请注意,其他情况可能也适用,主要取决于你的环境):

- 有规律地执行一条或多条相关的 SQL 语句。
- 因为网络的带宽很有限,返回 SQL 语句的结果。
- 因为客户机的处理资源很有限,将数据处理委托给服务器。
- 许多客户有规律地执行包含 SQL 语句的批命令。

当然,建立存储过程还需要更多的工作,但是以往的经验证实了这些额外的工作将在开始编码和使用应用程序时节省至少十倍的时间。

最后要指出的一点是,如果将很多数据调用建立在存储过程基础之上,那么在之后改变这些数据调用时将非常简单。只需简单地改变 SP 而不是程序本身就可以了,即你不必重新编译商业服务或者客户应用程序,这取决于你设计应用程序的方式。负面的因素是,存储过程总是使用数据库供应商特定的 SQL 扩展而写成的,这意味着这些存储过程很难迁移到不同类型的 RDBMS 中。当然,只在你打算转移到另一个 RDBMS 时,这才会是一个实际性的考虑。

6.2.2 创建存储过程

创建存储过程非常简单。你可能对 SQL Server 带有的 Server Manager 十分熟悉，若是这样，你可能会去检查 VS.NET IDE 的服务器资源管理器功能。同其他方式相比，从文本编辑器中直接执行并测试 SP 是较为简单的。下面列出的就是为数据库示例 UserMan 建立存储过程的方法。

1. 打开服务器资源管理器窗口。
2. 展开数据库服务器上的 UserMan 数据库。
3. 右键单击 [Stored Procedures] 节点并选择 [New Stored Procedure]。

这样就显示出了存储过程的文本编辑器，它与 VB.NET 的代码编辑器很相似。除了语法检查以及其他次要方面外，这两种编辑器完全一样（参见图 6.1）。

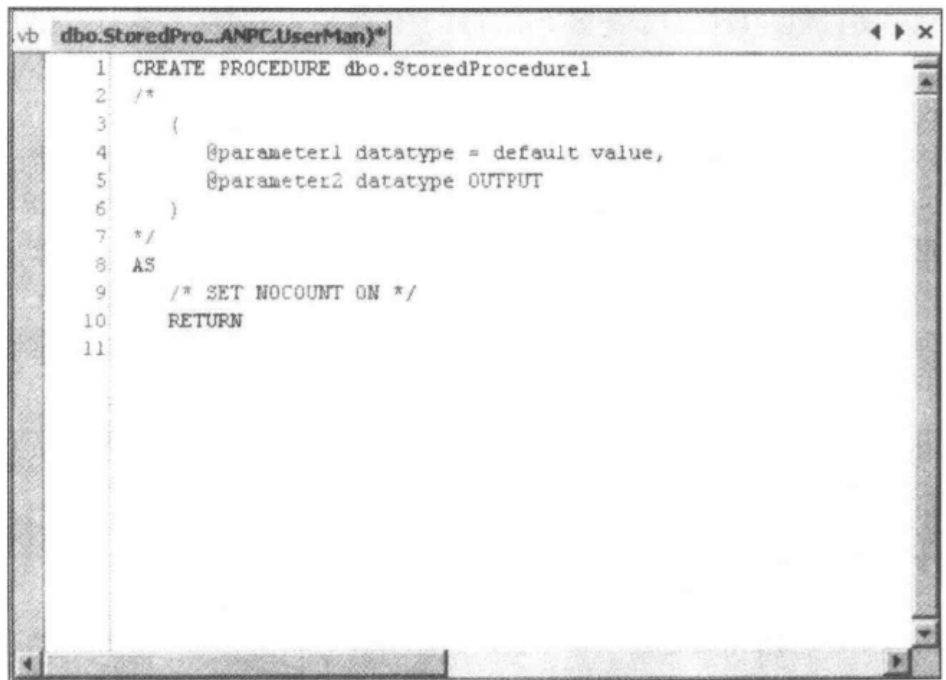


图 6.1 默认模板的存储过程编辑器

创建一个简单的存储过程

一旦创建了存储过程，就需要给它命名。正像你从 SP 编辑器中看到的一样，模板自动将其命名为 StoredProcedure1。你可能会对 dbo 前缀感到奇怪，实际上它仅表示 SP 是为 dbo 用户创建的。在 SQL Server 的术语中，dbo 指的是数据库所有者（DataBase Owner）。如果你曾用过 SQL Server，就可能知道下面这个术语：断所属关系链（*broken ownership chain*）。所属关系链指的是依赖于其他数据库对象（比如表）及其他视图及存储过程的存储过程。

一般来讲，视图或存储过程所依赖的对象都被视图或存储过程的所有者所拥有。在这种情况下没有任何问题，因为 SQL Server 在这种情况下不会检查权限。然而，当一个或多个独立数据库对象的拥有者不同于视图或存储过程的拥有者的时候，所属关系链就被认为是断裂的。这就意味着 SQL Server 必须检查所有属于不同拥有者的独立数据库对象的权限。如果所

有的数据库对象都被同一个用户拥有（比如 dbo），则可以避免这点。我并不是说一定要用这个方法，但它的确是个可以使用的选项。

假定你已经删除了 `StoredProcedure1` 的名称，并将其替换为 `SimpleStoredProcedure`。需要在继续操作之前保存 SP，请按 [Ctrl] + [S]。如果在这个时刻保存存储过程，你将发现不必使用 `Save As` 对话框来对它命名，因为你已经完成了命名。编辑器会保证 SP 以你所输入的名称（这里是 `SimpleStoredProcedure`）保存在数据库服务器上。

一旦保存完毕，SP 的第一行就会改变。SQL 语句 **CREATE PROCEDURE** 将改变为下面的语句：

```
ALTER PROCEDURE dbo.SimpleStoredProcedure
```

这是为什么？你已经保存了一个新创建的 SP，即不能再以同一个名称创建另外一个 SP。将 **CREATE** 改为 **ALTER** 保证了这一点。就是这么简单！如果你想知道当改变 SP 的名称时 SQL 语句是否还是 **ALTER PROCEDURE...** 的话，则可以在这里告诉你，编辑器已经想到了这一点并创建了一个新的过程。自己尝试一下吧！

`SimpleStoredProcedure` 实际上没起什么作用，接下来将为你演示如何改变这种情况。在图 6.1 中，可以看到 SP 的两个部分。第一部分是，而接下来的部分才是 SP 本身。第 12 第 7 行（包含第 7 行）的所有文本组成了。基本上，声明了 SP 的调用方式、包含的参数个数以及参数的类型，等等。因为这是一个非常简单的过程，并不需要任何参数，所以没有改动提示文本。

如果没有改变默认的编辑器设置，那么注释文本或其他你自己加入的文本将以绿色显示。在 SQL Server 的 SP 中，注释使用的是开始标记与结束标记。/* 是注释开始标记，而 */ 是注释结束标记。这比在 VB 代码中插入注释的方式要好，因为不必在每个注释行都设置一个注释开始标记。只需一个起始标记及一个结束标记就可以了。

SP 的第二部分是以第 8 行的 **AS** 子句起始的部分。**AS** 子句意味着下面的文本是 SP 主体，是调用、执行 SP 时完成工作的指令。

练 习

创建一个存储过程，将它命名为 `SimpleStoredProcedure` 并按照前面所讲的进行保存。用下面的文本代替 `SimpleStoredProcedure` 第 10 行的 `RETURN` 语句：

```
SELECT COUNT(*) FROM tblUser
```

现在，该存储过程应该与图 6.2 中的示例类似，存储过程将返回 `tblUser` 表中的行数。不要忘记使用 [Ctrl] + [S] 来保存所做的改变。

从 IDE 执行简单的存储过程

当然，让一个存储过程摆在那里是没意义的，下面就是执行它的方式。如果已经在 SP 编辑窗口中打开了 SP，就可以用右键单击编辑窗口的任何地方，并在弹出的菜单中选择 [Run Stored Procedure]。如果是处理前面部分的练习中创建的存储过程，处于编辑窗口正下方的输

出窗口中就会显示 SP 的输出，如图 6.3 所示。

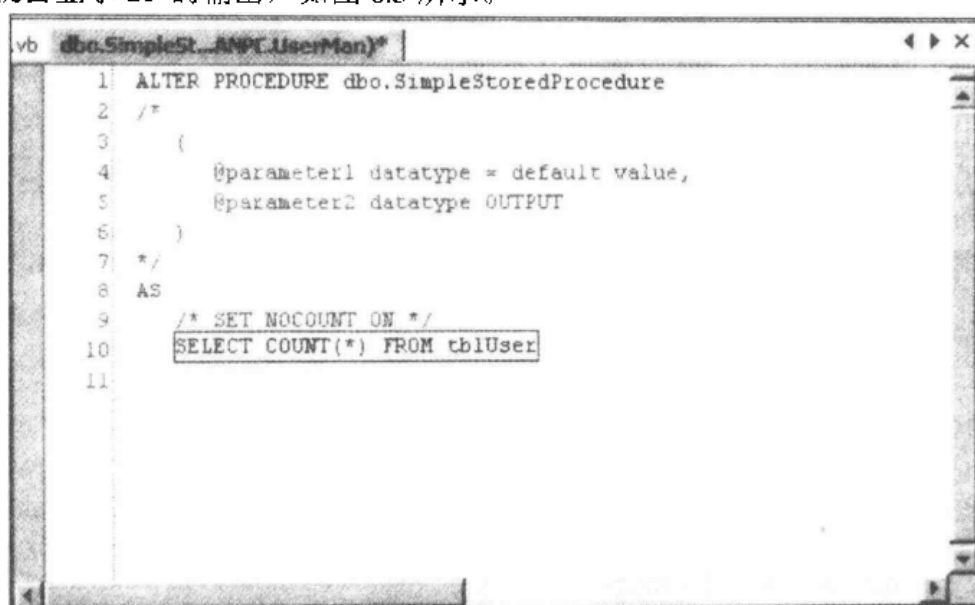


图 6.2 返回 tblUser 表的行数的存储过程

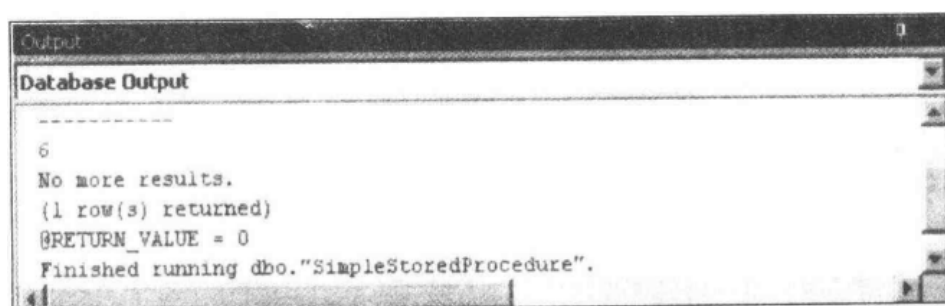


图 6.3 显示 SimpleStoredProcedure 输出的输出窗口

如果关闭了 SP 编辑窗口，则可以从服务器资源管理器中执行 SP。展开数据库节点，右键单击 [Stored Procedures] 节点，并从弹出的菜单中选择 [Run Stored Procedure]。这样，执行存储过程的方式就与在编辑窗口中运行完全相同了。

从代码中运行简单的存储过程

有了一个功能完全的 SP，接下来就可以从代码中运行它了。程序清单 6.1 列出了一些运行 SP 的简单代码。

程序清单 6.1 运行简单的存储过程

```
1 Public Sub ExecuteSimpleSP()
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3       "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Dim cnnUserMan As SqlConnection
6     Dim cmdUser As SqlCommand
7     Dim lngNumUsers As Long
8
```



```

9      ' 实例化并打开连接
10     cnnUserMan = New SqlConnection(STR_CONNECTION)
11     cnnUserMan.Open()
12
13     ' 实例化并初始化命令
14     cmdUser = New SqlCommand("SimpleStoredProcedure", cnnUserMan)
15     cmdUser.CommandType = CommandType.StoredProcedure
16
17     ' 保存结果
18     lngNumUsers = cmdUser.ExecuteScalar()
19 End Sub

```

程序清单 6.1 中的代码检索 SP 的返回值。这个任务并不是真正希望要用存储过程来完成，但它解释了简单的 SP 是怎样的。SP 自身也可以有 `DELETE FROM tblUser WHERE LastName='Johnson'` SQL 这样的语句。如果从代码中执行，就必须注意 SP 是否有返回值。但上例并不是这种情况，因此要使用 `SqlCommand` 类的 `ExecuteNonQuery` 方法。

练 习

创建一个新的存储过程并以 `uspGetUsers` 名称保存。在第 10 行输入下面的文本代替 **RETURN** 语句：

```
SELECT * FROM tblUser
```

该存储过程应该如图 6.4 中所示，它将返回 `tblUser` 表中的所有行。不要忘记使用 `[Ctrl] + [S]` 保存所做的改变。

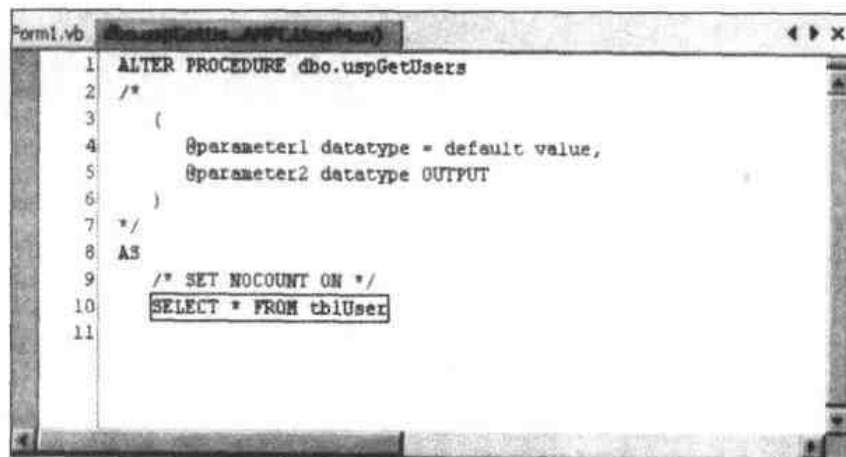


图 6.4 存储过程 `uspGetUsers`

现在你需要做的是检索 SP 中行的代码（参阅程序清单 6.2）。

程序清单 6.2 在存储过程中检索行

```
1 Public Sub ExecuteSimpleRowReturningSP()
```

```

2  Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3    "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5  Dim cnnUserMan As SqlConnection
6  Dim cmdUser As SqlCommand
7  Dim drdUser As SqlDataReader
8
9  ' 实例化并打开连接
10  cnnUserMan = New SqlConnection(STR_CONNECTION)
11  cnnUserMan.Open()
12
13  ' 实例化并初始化命令
14  cmdUser = New SqlCommand("uspGetUsers", cnnUserMan)
15  cmdUser.CommandType = CommandType.StoredProcedure
16
17  ' 检索所有用户行
18  drdUser = cmdUser.ExecuteReader()
19 End Sub

```

程序清单 6.2 中的示例通过 **SqlCommand** 类的 **ExecuteReader** 方法检索到从 SP 返回的行。请注意，这是现在惟一能够从 **command** 类的函数调用结果中检索行的方法。

创建带有参数的存储过程

有时候创建一个带有参数的存储过程要强于拥有多个主要功能相同的 SP。这也带来了更多的灵活性，从而在对应用程序做小的改变时，不必重新编译它的某些部分。这是因为你可以增加参数，并通过为新参数指定默认值的方式来保持现存应用程序的正常运行。

练 习

创建一个新的存储过程并以 **uspGetUsersByLastName** 名称保存。在第 10 行和第 11 行以下文本代替 **RETURN** 语句：

```

SELECT * FROM tblUser
WHERE LastName = @strLastName

```

将第 2 行到第 7 行的注释去掉，并输入下面的文本分别替代原来的第 3 行和第 4 行：

```
@strLastName varchar(50)
```

新的存储过程如图 6.5 所示。它将返回 **tblUser** 表中所有 **LastName** 列与参数 **strLastName** 匹配的所有行。

不要忘记使用 **[Ctrl] + [S]** 保存所做的改变。

存储过程中的参数可以是输入或者输出。如果使用输入参数，在数据类型之后什么都不必指定，但是如果使用输出参数，就需要在数据类型之后指定 **OUTPUT** 键。

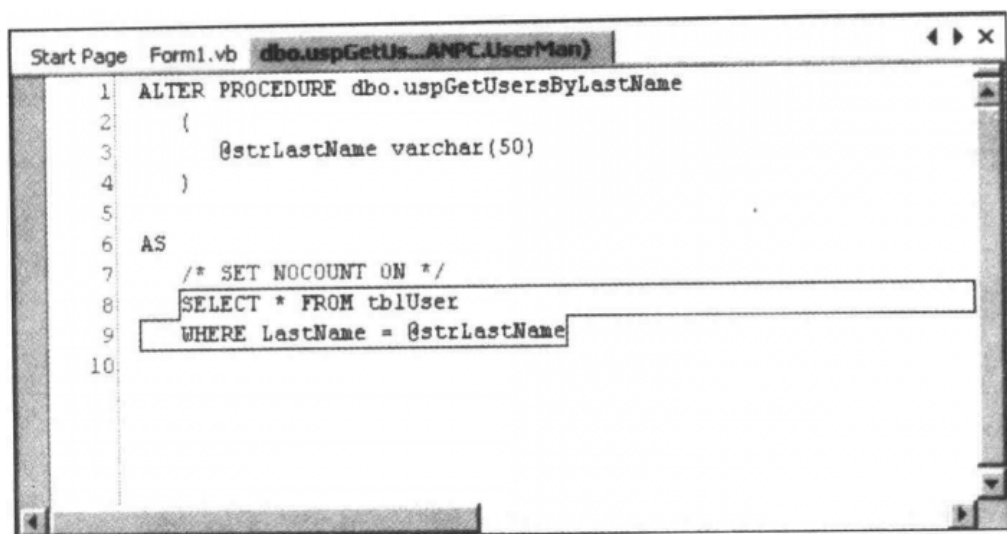


图 6.5 存储过程 uspGetUsersByLastName



本章只介绍了创建存储过程的最基本内容。如果需要了解更多信息，推荐读者在 SQL Server 附带的 Books Online 帮助应用程序中查询 **CREATE PROCEDURE** 语句。

从 IDE 中运行带有参数的存储过程

试着运行在上一个练习中创建的存储过程，观察参数如何影响它的运行方式。从编辑窗口或服务器资源管理器窗口都可以运行 SP。Run 对话框将向你询问 strLastName 参数的值。在文本框中键入 **Doe** 并单击 [OK]。现在，所有姓氏为 Doe 的用户都将作为 SP 的结果被返回。

使用带有参数的存储过程

存储过程 uspGetUsersByLastName 看起来已经可以使用了，试着在代码中运行它。程序清单 6.3 显示了操作步骤。

程序清单 6.3 在带有输入参数的存储过程中检索行

```

1 Public Sub GetUsersByLastName()
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Dim cnnUserMan As SqlConnection
6     Dim cmdUser As SqlCommand
7     Dim drdUser As SqlDataReader
8     Dim prmLastName As SqlParameter
9
10    ' 实例化并打开连接
11    cnnUserMan = New SqlConnection(STR_CONNECTION)
12    cnnUserMan.Open()

```

```

13
14 ' 实例化并初始化命令
15 cmmUser = New SqlCommand("uspGetUsersByLastName", cnnUserMan)
16 cmmUser.CommandType = CommandType.StoredProcedure
17 ' 实例化、初始化命令并为命令增加参数
18 prmLastName = cmmUser.Parameters.Add("@strLastName", _
19     SqlDbType.VarChar, 50)
20 ' 表明这是输入参数
21 prmLastName.Direction = ParameterDirection.Input
22 ' 设置参数的值
23 prmLastName.Value = "Doe"
24
25 ' 返回所有姓氏为 Doe 的用户
26 drdUser = cmmUser.ExecuteReader()
27 End Sub

```

在程序清单 6.3 中，**SqlParameter** 对象指定了存储过程的输入参数。在第 18 行和第 19 行，命令对象创建了一个参数，并将其与 **@strLastName** 参数相关联。通过将该参数的值设为 “Doe”，指明只返回姓氏为 Doe 的行。

就像你所看到的，该示例使用 **ParameterDirection** 枚举将参量指定为输入参数。不要担心参量和参数，它们基本是同一事物。

创建带有参数及返回值的存储过程

至此已经创建了返回单值或结果集（行）的 SP，以及带有一个输入参数的存储过程。在多数情况下这已经足够了，但有时却不是这样。如果你希望同时返回一个单值与一个结果集时会怎么样？实际上，你可能需要数个单值与一个结果集，你一定已经知道该怎么做了。在这种情形下，你可以使用输出参数。

可以在 **AS** 子句后使用大量的 **SELECT** 语句来返回多个不同的值与结果集，但这样做会显得很乱。如果要同时返回行以及一个或多个值，我一般会对值使用输出参数。但这只是个人的偏好问题。

我不会像下面所示代码的顺序返回一个单值、两个结果集以及另一个单值：

```

...
AS
    SELECT 19
    SELECT * FROM tblUser
    SELECT * FROM tblUserRights
    SELECT 21

```

我一般会像下面这样做：

```

...
AS
    SELECT * FROM tblUser
    SELECT * FROM tblUserRights

```

上面的两个返回值会作为输出参数被返回。你完全可以用自己喜欢的方式。

练 习

建立一个新存储过程并以 `uspGetUsersAndRights` 名称保存。这个存储过程将为输出参数 `lngNumRows` 返回值 55 以及表 `tblUser` 和表 `tblUserRights` 中的所有行。

存储过程现在应该如图 6.6 所示。

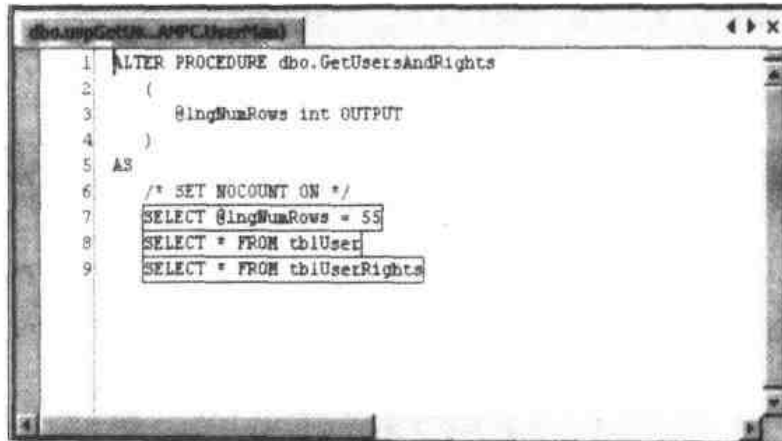


图 6.6 存储过程 `uspGetUsersAndRights`

在 IDE 中运行带有参数及返回值的存储过程

如果已经创建且保存了上一个练习中的存储过程，现在就可以试着运行它。可以试着从编辑窗口或是服务器资源管理器窗口来运行 SP。在编辑窗口正下方的输出窗口，将显示 SP 的输出，如图 6.7 所示。



注意 存储过程的语法检查是在保存的时候完成的，你应该已经遇到过这种情况了。如果还没有，则需要了解一下它是怎么回事：在试图保存 SP 的时候，系统将捕获其中的语法错误。

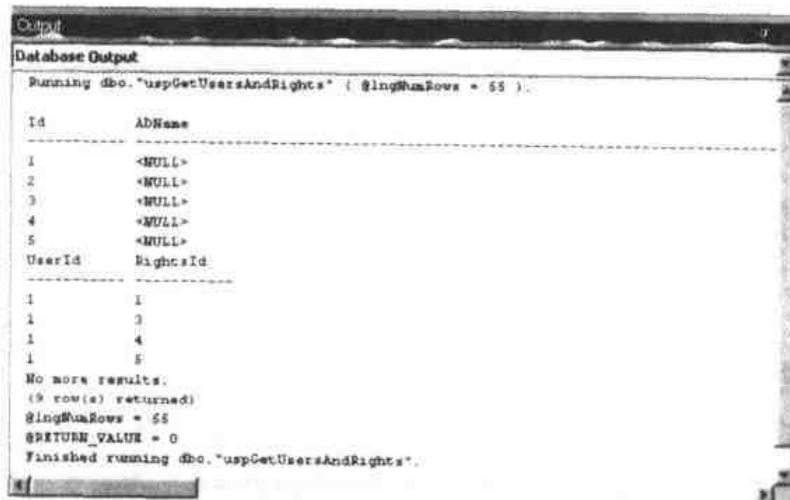


图 6.7 带有 `uspGetUsersAndRights` SP 输出的输出窗口

使用带有参数及返回值的存储过程

程序清单 6.4 列出了执行 `uspGetUsersAndRights` SP 的代码。

程序清单 6.4 在存储过程中检索行与输出值

```

1 Public Sub GetUsersAndRights()
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Dim cnnUserMan As SqlConnection
6     Dim cmdUser As SqlCommand
7     Dim drdUser As SqlDataReader
8     Dim prmNumRows As SqlParameter
9
10    ' 实例化并打开连接
11    cnnUserMan = New SqlConnection(STR_CONNECTION)
12    cnnUserMan.Open()
13
14    ' 实例化并初始化命令
15    cmdUser = New SqlCommand("uspGetUsersAndRights", cnnUserMan)
16    cmdUser.CommandType = CommandType.StoredProcedure
17    ' 实例化、初始化命令并为命令增加参数
18    prmNumRows = cmdUser.Parameters.Add("@lngNumRows", SqlDbType.Int)
19    ' 表明这是一个输出参数
20    prmNumRows.Direction = ParameterDirection.Output
21    ' 取得首批行(用户)
22    drdUser = cmdUser.ExecuteReader()
23
24    ' 显示所有用户行的姓氏
25    Do While drdUser.Read()
26        MsgBox(drdUser("LastName").ToString)
27    Loop
28
29    ' 取得下一批行(用户权限)
30    If drdUser.NextResult() Then
31        ' 显示所有权限的 id
32        Do While drdUser.Read()
33            MsgBox(drdUser("RightsId").ToString)
34        Loop
35    End If
36 End Sub

```

在程序清单 6.4 中返回了两个结果值，因此在第二个结果集的第 30 行使用了数据读入类的 `NextResult` 方法。否则，该存储过程的功能就会像仅带有一个输入参数的存储过程一样了，尽管在第 20 行参量方向已经指定为输出。

6.2.3 用 RETURN 检索特定的值

在存储过程中，可以使用 **RETURN** 语句来返回一个单值。然而，该值不能像 **SELECT** 语句（参阅图 6.2）那样用 **command** 类的 **ExecuteScalar** 方法来检索。当然，可以有其他办法检索该值，下面的练习中就说明了这个方法。

练 习

创建一个新存储过程并以 **uspGetRETURN_VALUE** 名称保存。这个存储过程将值 55 作为 **RETURN_VALUE** 返回。该存储过程如图 6.8 所示。

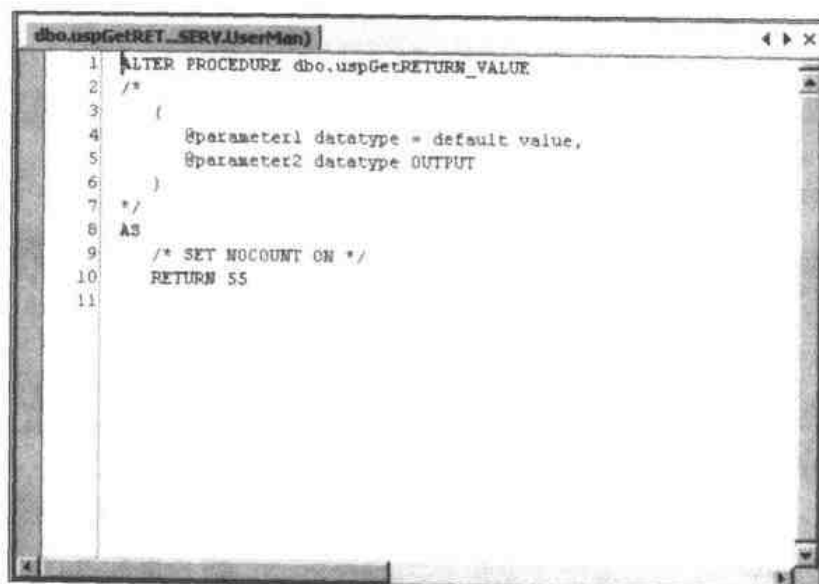


图 6.8 存储过程 **uspGetRETURN_VALUE**

程序清单 6.5 展示了在代码中检索值的方法。

程序清单 6.5 在存储过程中检索 **RETURN_VALUE**

```
1 Public Sub GetRETURN_VALUE()  
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _  
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"  
4  
5     Dim cnnUserMan As SqlConnection  
6     Dim cmdUser As SqlCommand  
7     Dim drdUser As SqlDataReader  
8     Dim prmNumRows As SqlParameter  
9     Dim lngResult As Long  
10  
11     ' 实例化并打开连接
```

```

12  cnnUserMan = New SqlConnection(STR_CONNECTION)
13  cnnUserMan.Open()
14
15  ' 实例化并初始化命令
16  cmdUser = New SqlCommand("uspGetRETURN_VALUE", cnnUserMan)
17  cmdUser.CommandType = CommandType.StoredProcedure
18  ' 实例化、初始化命令并为命令增加参数
19  prmNumRows = cmdUser.Parameters.Add("@RETURN_VALUE", SqlDbType.Int)
20  ' 表明这是一个返回值参数
21  prmNumRows.Direction = ParameterDirection.ReturnValue
22  ' 取得 RETURN_VALUE
23  lngResult = cmdUser.ExecuteScalar()
24 End Sub

```

在程序清单 6.5 里，**ExecuteScalar** 方法在存储过程中取得了返回值。一般情况下，将使用这个方法返回变量 `lngResult` 的值，但在本例中，此变量的值是默认的 0。然而，因为已经使用 **ParameterDirection** enum 的成员 **ReturnValue** 指定了参数 `prmNumRows` 的 **Direction** 属性，因此可以在执行命令后简单地查看该参数的 **Value** 属性。

6.2.4 改变存储过程的名称

如果在编辑窗口中改变了存储过程的名称，那么存储过程就会按照保存时 ([Ctrl]+[S]) 所采用的新名称保存起来。然而，如果你不是使用这种方式复制已经存在的 SP，就要确定旧的 SP 依然存在。当你不再需要旧版本时，应将其删除。

6.3 使用视图

视图 (view)，顾名思义，是数据库中数据的显示。把视图想像为一张虚表也许能帮助你理解它的概念。它可以是表的一个子集或整张表，也可以是多个关联表的一个子集。视图基本上可以表示数据库中数据的任何子集，并且在一个视图中还可以包含其他视图。在视图包含其他视图称为嵌套 (nesting)，这种方法对于分组显示数据来说很有价值，但是嵌套太深可能会导致性能问题，并肯定会使查错过程变得非常复杂。对于视图来说，没有任何法术或者秘密。它只是用来操作数据的一个很好的工具。就像存储过程一样，视图用来在服务器端处理数据，但 SP 主要是基于安全和性能原因使用的，而视图主要是为了保护对数据的访问以及隐藏屏蔽包含很多连接的查询的复杂性。本章的剩余部分将讨论为什么、什么时候、在什么地方，以及怎样使用视图。

6.3.1 视图限制

视图和返回行的查询基本一样，只有很少的例外。下面给出了一些有关视图的具体限制：

- 不能在视图使用 **COMPUTE** 与 **COMPUTE BY** 子句。

- 除非将 **TOP** 子句指定为 **SELECT** 语句的一部分，否则不能在视图中使用 **ORDER BY** 子句。
- 不能使用键 **INTO** 来创建新表。
- 临时表不能作为参照源。

其他限制请你查看 SQL Server 文档或帮助文件。

6.3.2 使用视图的原因

有很多原因会促使你使用视图：

- 安全性：当你不希望用户直接访问到表格时，通过使用视图，可以有效地对用户进行限制，使其只能看到允许其看到的部分。你可以限制为只对指定列和/或行进行访问。
- 加密：可以将视图加密，这样就没人知道后台的数据是从哪里来的。要记住，这是个不可逆的操作！

创建视图还有其他的理由，但上述的两个方面无疑是最常见的。

6.3.3 创建视图

创建视图很简单。如果你惯于使用 SQL Server 的 Server Manager，则可以看看服务器资源管理器为你提供了些什么。下面是使用 UserMan 数据库创建视图的一个示例：

1. 打开服务器资源管理器窗口。
 2. 展开数据库服务器上的 UserMan 数据库。
 3. 右键单击 [Views] 节点，并选择 [New View]。
- 这样就打开了 View Designer，它实际上与 Query Designer 相同。



在第 4 章详细介绍了 Query Designer。虽然 View Designer 与 Query Designer 的外观和感觉一样，但是不能违反前面所述的视图限制（见“视图限制”部分）来建立视图。

在显示 View Designer 的时候，Add Table 对话框也将显示出来。在该对话框中，选择那些需要从其中检索数据的表格并单击 [Add]。当所有表格添加完毕后，单击 [Close]。

当在 Diagram 面板中选择准备在视图运行时输出的列时，SQL 窗格以及 Grid 窗格也会随之改变。当完成输出列的选择后，应该用 [Ctrl] + [S] 对视图进行保存。

练 习

创建一个新视图，使该视图中包含下列表格：tblUser、tblRights 和 tblUserRights。将下列字段作为输出：tblUser.LoginName、tblUser.FirstName、tblUser.LastName 和 tblRights.Name。以 viewUserInfo 名称保存视图。新视图应该如图 6.9 所示。

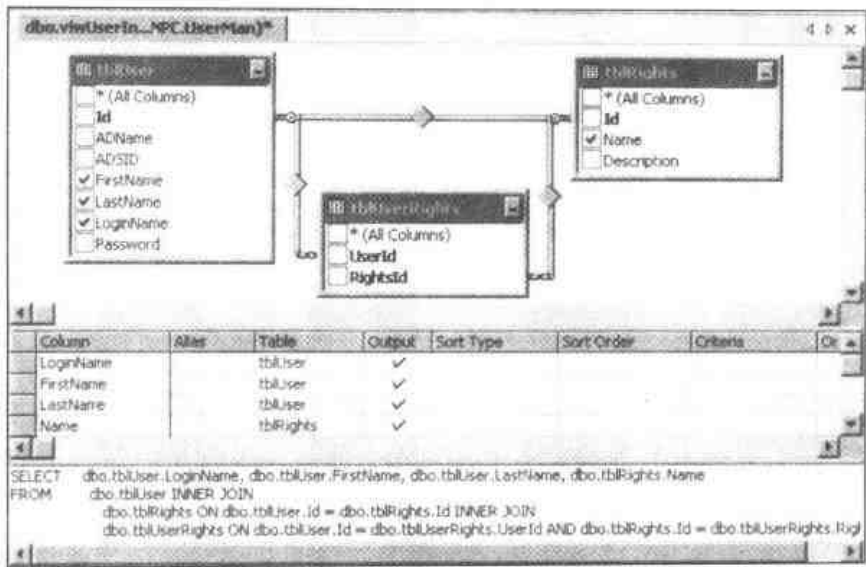


图 6.9 视图 viewUserInfo

6.3.4 在 IDE 中运行视图

“运行视图”可能看起来不是最恰当的表达。但另一方面，视图确实要从其中的参照表格中检索数据，所以“运行视图”也许和这有关。

你可以在 View Designer 中右键单击 View Designer 的空白区域，并从弹出菜单中选择 [Run]。

练习

在 View Designer 中运行视图 viewUserInfo，结果窗格将显示如图 6.10 所示的行。注意，tblRights 表的 Name 字段可能看起来有点乱，这是因为它没有显示出 Name 的真正意思。为了让它更清楚一些，在 Grid 窗格中 Name 行的 Alias 列中加入文本 **RightsName**。重新运行视图，并注意观察 Results 窗格中新的列名是如何显示的。用 [Ctrl] + [S] 来保存视图。

LoginName	FirstName	LastName	Name
UserMan	John	Doe	AddUser

图 6.10 运行 viewUserInfo 视图的结果

6.3.5 在代码中使用视图

实际上，在代码中使用视图非常容易，因为视图在数据库中进行引用的方式与任何标准表一样，这就意味着可以用命令对象或填充数据集的数据适配器等来检索数据。



请参见第 3B 章以获得关于在表格中操作数据的详细信息。

使用代码中的视图检索只读数据

视图最简单的应用就是出于显示的目的，比如在需要显示一个或多个相关表中信息的时候。因为在示例代码中不必考虑更新问题，所以不必进行特别的设置。程序清单 6.6 展示了在视图中检索所有行并填充数据阅读器的方式。

程序清单 6.6 在视图中检索行

```

1 Public Sub RetrieveRowsFromView()
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Dim cnnUserMan As SqlConnection
6     Dim cmdUser As SqlCommand
7     Dim drdUser As SqlDataReader
8
9     ' 实例化并打开连接
10    cnnUserMan = New SqlConnection(STR_CONNECTION)
11    cnnUserMan.Open()
12
13    ' 实例化并初始化命令
14    cmdUser = New SqlCommand("SELECT * FROM viwUserInfo", cnnUserMan)
15    ' 取得行
16    drdUser = cmdUser.ExecuteReader()
17 End Sub

```

除了查询的是视图而不是表格这一点外，程序清单 6.6 与其他返回行查询一样。

在代码中操作视图中的数据

程序清单 6.6 展示了如何从视图中检索数据并将其放入数据阅读器中，这就意味着数据无法更新，因为数据阅读器不允许进行更新。然而，视图中的数据却是有可能更新的。这里的问题是，SQL Server 的不同版本支持对视图更新的不同等级的支持。如果视图中只有一个表格，则不会有什么问题。但是，假如视图中有多个表格，那么只有 SQL Server 2000 才支持多张表的列更新。除了上述问题之外，如果要迁移到不同的 RDBMS 中，你还会遇到更大的麻烦。一般来说，不鼓励在视图中更新数据。

练 习

创建一个包含 tblUser 表的新视图。将 Id、FirstName、LastName 以及 LoginName 作为输出字段。并以名称 viwUser 保存视图。新视图应该如图 6.11 所示。

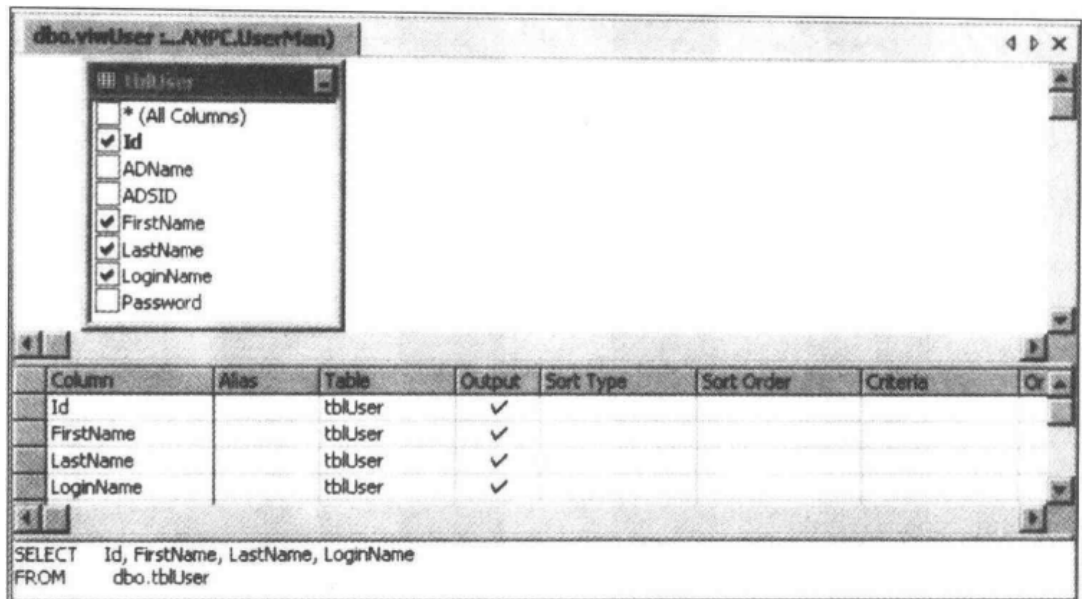


图 6.11 视图 viwUser

本视图既可以放在 SQL Server 7.0 上也可以放在 SQL Server 2000 上，而且可以使用程序清单 6.7 中的代码进行操作。

程序清单 6.7 对基于单个表格的视图中的数据进行操作

```

1 Public Sub ManipulatingDataInAViewBasedOnSingleTable()
2     Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3         "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5     Const STR_SQL_USER_SELECT As String = "SELECT * FROM viwUser"
6     Const STR_SQL_USER_DELETE As String = "DELETE FROM viwUser WHERE Id=@Id"
7     Const STR_SQL_USER_INSERT As String = "INSERT INTO viwUser(FirstName," & _
8         " LastName, LoginName, Logged, Description) VALUES(@FirstName, " & _
9         "@LastName, @LoginName)"
10    Const STR_SQL_USER_UPDATE As String = "UPDATE viwUser SET " & _
11        "FirstName=@FirstName, LastName=@LastName, LoginName=@LoginName " & _
12        "WHERE Id=@Id"
13
14    Dim cnnUserMan As SqlConnection
15    Dim cmdUser As SqlCommand
16    Dim dadUser As SqlDataAdapter
17    Dim dstUser As DataSet
18
19    Dim cmdUserSelect As SqlCommand
20    Dim cmdUserDelete As SqlCommand
21    Dim cmdUserInsert As SqlCommand
22    Dim cmdUserUpdate As SqlCommand
23
24    Dim prmSQLDelete, prmSQLUpdate, prmSQLInsert As SqlParameter
25
26    ' 实例化并打开连接

```

```

27  cnnUserMan = New SqlConnection(STR_CONNECTION)
28  cnnUserMan.Open()
29
30  ' 实例化并初始化命令
31  cmdUser = New SqlCommand("SELECT * FROM viewUser", cnnUserMan)
32  ' 实例化命令
33  cmdUserSelect = New SqlCommand(STR_SQL_USER_SELECT, cnnUserMan)
34  cmdUserDelete = New SqlCommand(STR_SQL_USER_DELETE, cnnUserMan)
35  cmdUserInsert = New SqlCommand(STR_SQL_USER_INSERT, cnnUserMan)
36  cmdUserUpdate = New SqlCommand(STR_SQL_USER_UPDATE, cnnUserMan)
37  ' 实例化命令和数据集
38  cmdUser = New SqlCommand(STR_SQL_USER_SELECT, cnnUserMan)
39  dstUser = New DataSet()
40
41  dadUser = New SqlDataAdapter()
42  dadUser.SelectCommand = cmdUserSelect
43  dadUser.InsertCommand = cmdUserInsert
44  dadUser.DeleteCommand = cmdUserDelete
45  dadUser.UpdateCommand = cmdUserUpdate
46
47  ' 添加参数
48  cmdUserDelete.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
49  "FirstName")
50  cmdUserDelete.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
51  "LastName")
52  cmdUserDelete.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
53  "LoginName")
54  prmSQLDelete = dadUser.DeleteCommand.Parameters.Add("@Id", _
55  SqlDbType.Int, 0, "Id")
56  prmSQLDelete.Direction = ParameterDirection.Input
57  prmSQLDelete.SourceVersion = DataRowVersion.Original
58
59  cmdUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
60  "FirstName")
61  cmdUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
62  "LastName")
63  cmdUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
64  "LoginName")
65  prmSQLUpdate = dadUser.UpdateCommand.Parameters.Add("@Id", _
66  SqlDbType.Int, 0, "Id")
67  prmSQLUpdate.Direction = ParameterDirection.Input
68  prmSQLUpdate.SourceVersion = DataRowVersion.Original
69
70  cmdUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, 50, _
71  "FirstName")
72  cmdUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, 50, _
73  "LastName")
74  cmdUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, 50, _
75  "LoginName")

```

```

76
77   · 填充视图中的数据
78   dadUser.Fill(dstUser, "viwUser")
79
80   · 修改第 2 行中用户的姓氏
81   dstUser.Tables("viwUser").Rows(1)("LastName") = "Thomsen"
82   · 将改变传给数据源
83   dadUser.Update(dstUser, "viwUser")
84 End Sub

```

在程序清单 6.7 中，数据适配器和数据集设置为检索并保存 `viwUser` 视图中的数据。之后，第 2 行的 `LastName` 列按照数据集的改变而更新。这个简单的示例用来解释如何使用基于一个单表的视图进行工作。

6.4 使用触发器

触发器 (trigger) 实际上是当数据发生某种改变时自动激活 (触发) 的存储过程。触发器是本章讨论的最后一种服务器端处理功能。直到 SQL Server 2000 发售之前，触发器一直都是增强参照完整性的关键部分，但随着 SQL Server 2000 的发售，这些功能都变为内置的了。在本章的剩余部分中，我们将讲解触发器的概念以及使用的时机和方式。

触发器对使用 `INSERT`、`UPDATE` 以及 `DELETE` 操做的数据改变作出响应。基本来说，触发器可以让你少写些代码。你可以将公司的业务规定写入触发器，并由此防止出现违反该规定的数据库。

这里有一个绝好的公司业务规定的示例：检查某组织的会员是否交了年费，以此决定其可否订购该组织库中的材料。当会员订购材料时，`INSERT` 触发器会查找会员表并检查该会员是否交了年费。在某些情形下触发器比约束条件更有用是因为触发器可以访问其他表中的列，而约束只能访问当前表或行中的列。

SQL Server 实现了 `AFTER` 触发器，这意味着可以在修改之后将激活触发器。然而，这并不是说改变无法取消了，因为触发器直接访问修改过的列，也因此能够取消任何改变。我们要说明的是，SQL Server 实现了 `AFTER` 触发器，而在 SQL Server 2000 之前，SQL Server 也只支持这种类型的触发器。然而，SQL Server 2000 还支持 `BEFORE` 触发器的概念，这点你或许已从 Oracle RDBMS 中得知了。在 SQL Server 2000 中，它们被称为 `INSTEAD OF` 触发器。

6.4.1 使用触发器的原因

触发器是自动操作的，所以不必将业务逻辑应用在代码中。想想刚才那个组织的某位会员从该组织库中订购材料的示例。如果用代码来处理业务规定，就必须在将订单插入订单表之前先检查会员表中该会员的信息。而使用触发器，这种检查就可以自动完成，如果在插入订购库材料订单时检查出该会员尚未交纳年费，就会抛出一个异常。

为了减少麻烦，最好使用触发器来保持数据合法或符合公司的业务规定。可以把触发器

看作一种附加的保持合法性的工具，同时还要确保建立了参照完整性。



注意

在使用 SQL Server 2000 时，不必使用触发器或是参照完整性（请参阅第 2 章），这是因为可以使用 Database Designer 对其进行设置。关于 Database Designer 请参阅第 4 章。

6.4.2 创建触发器

创建触发器比较简单。可以使用 SQL Server 带有的 Server Manager 完成，但是我将使用服务器资源管理器向你显示创建过程。下面是为示例数据库 UserMan 创建触发器的方法：

1. 打开服务器资源管理器窗口。
2. 展开数据库服务器上的 UserMan 数据库。
3. 展开 [Tables] 节点。
4. 右键单击想要创建触发器的表格，并从弹出菜单中选择 [New Trigger]。

此时将打开触发器文本编辑器，它与 VB.NET 代码的编辑器多少有些相像（参阅图 6.12）。

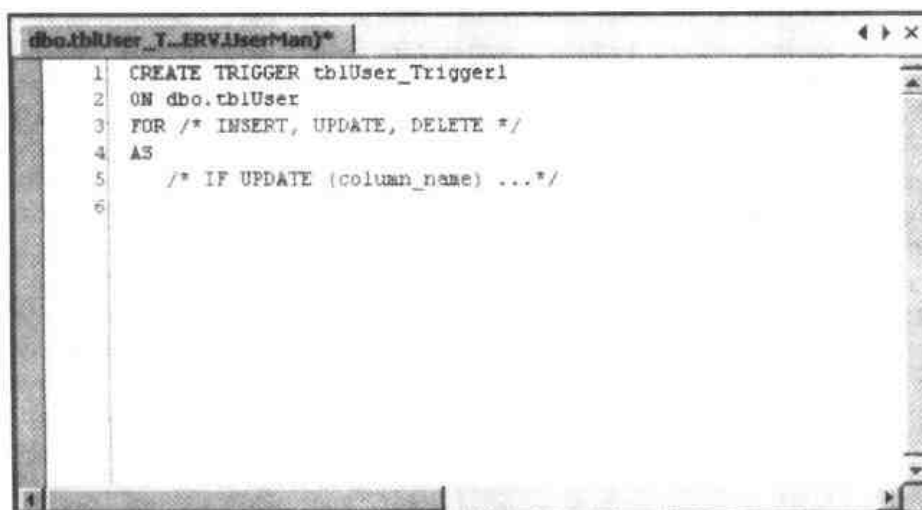


图 6.12 显示默认模板的触发器编辑器

在触发器编辑器中，可以看到模板自动将新触发器命名为带有表格名前缀的 Trigger1。如果已经存在以该名称命名的触发器，则新触发器将以 Trigger2 命名，依此类推。

一旦完成了触发器的编辑，需要按下 [Ctrl] + [S] 来保存。完成保存后，SP 的首行就改变了。SQL 语句 **CREATE TRIGGER** 将变成如下这样：

```
ALTER TRIGGER dbo....
```



注意

在保存触发器时，触发器编辑器会执行语法检查，如果触发器不合法，则不允许将其保存到数据库中。

练 习

为 tblUser 表格创建一个新触发器，并以名称 tblUser_Update 进行保存。这是个更新触发器，所以需要将第 3 行的文本变为 **FOR UPDATE**，并把第 5 行及其以后的部分用下面的语句来替换：

```
DECLARE @strFirstName varchar(50)
/* 为 FirstName 列取得值 */
SELECT @strFirstName = (SELECT FirstName FROM inserted)
/* 查看是否更新了 LastName 列
如果是，确保 FirstName 非空 */
IF UPDATE (LastName) AND @strFirstName IS NULL
BEGIN
    /* 取消更新并提出异常 */
    ROLLBACK TRANSACTION
    RAISERROR ('You must fill in both LastName and FirstName', 11, 1)
END
```

现在存储过程应该如图 6.13 所示，不要忘记用 [Ctrl] + [S] 进行保存。

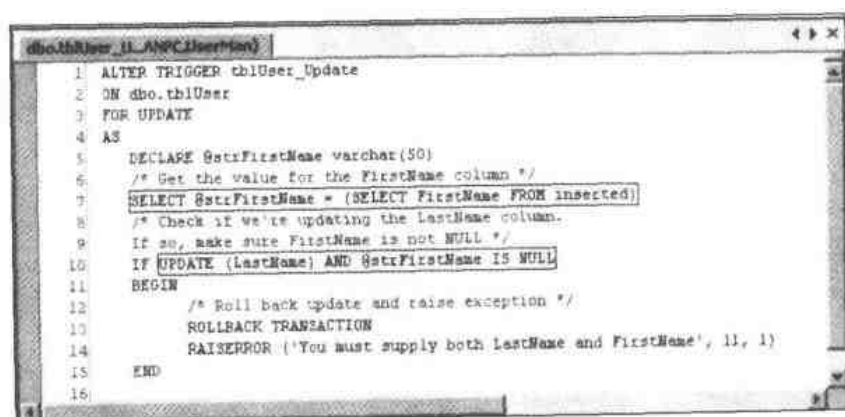


图 6.13 触发器 tblUser_Update

将触发器保存到数据库之后，便可以在服务器资源管理器中其所属的表格下面找到它。

触发器 tblUser_Update 在更新用户表中的行时会被激活。触发器首先检测 LastName 列是否被更新。如果是，那么触发器接下来要检查 FirstName 列是否为空，这是因为如果该列为空，更新就会被取消并提出一个异常。请注意，该触发器一次只能处理一条更新或插入行。如果需要同时插入多行，则需要重新设计触发器来处理这种情况。然而，该触发器仅仅是作为示例。其功能用约束很容易实现，这是因为所做的检查是在同一张表中的。假如这里要查找的是另一张表中的值，那么就只能使用触发器了。

如果想了解更多关于创建触发器的信息，请参阅 SQL Server 文档。程序清单 6.8 展示了执行新触发器的方法，并演示了怎样捕捉代码中的异常。

程序清单 6.8 使用触发器及捕捉抛出的异常

```
1 Public Sub TestUpdateTrigger()
```



```

2  Const STR_CONNECTION As String = "Data Source=USERMANPC;" & _
3    "User ID=UserMan;Password=userman;Initial Catalog=UserMan"
4
5  Const STR_SQL_USER_SELECT As String = "SELECT * FROM tblUser"
6  Const STR_SQL_USER_DELETE As String = _
7    "DELETE FROM tblUser WHERE Id=@Id"
8  Const STR_SQL_USER_INSERT As String = "INSERT INTO tblUser(" & _
9    "FirstName, LastName, LoginName) VALUES(@FirstName, " & _
10   "@LastName, @LoginName)"
11 Const STR_SQL_USER_UPDATE As String = "UPDATE tblUser SET " & _
12   "FirstName=@FirstName, LastName=@LastName, " & _
13   "LoginName=@LoginName WHERE Id=@Id"
14
15 Dim cnnUserMan As SqlConnection
16 Dim cmdUser As SqlCommand
17 Dim dadUser As SqlDataAdapter
18 Dim dstUser As DataSet
19
20 Dim cmdUserSelect As SqlCommand
21 Dim cmdUserDelete As SqlCommand
22 Dim cmdUserInsert As SqlCommand
23 Dim cmdUserUpdate As SqlCommand
24
25 Dim prmSQLDelete, prmSQLUpdate, prmSQLInsert As SqlParameter
26
27 ' 实例化并打开连接
28 cnnUserMan = New SqlConnection(STR_CONNECTION)
29 cnnUserMan.Open()
30
31 ' 实例化并初始化命令
32 cmdUser = New SqlCommand("SELECT * FROM tblUser", cnnUserMan)
33 ' 实例化命令
34 cmdUserSelect = New SqlCommand(STR_SQL_USER_SELECT, cnnUserMan)
35 cmdUserDelete = New SqlCommand(STR_SQL_USER_DELETE, cnnUserMan)
36 cmdUserInsert = New SqlCommand(STR_SQL_USER_INSERT, cnnUserMan)
37 cmdUserUpdate = New SqlCommand(STR_SQL_USER_UPDATE, cnnUserMan)
38 ' 实例化命令与数据集
39 cmdUser = New SqlCommand(STR_SQL_USER_SELECT, cnnUserMan)
40 dstUser = New DataSet()
41
42 dadUser = New SqlDataAdapter()
43 dadUser.SelectCommand = cmdUserSelect
44 dadUser.InsertCommand = cmdUserInsert
45 dadUser.DeleteCommand = cmdUserDelete
46 dadUser.UpdateCommand = cmdUserUpdate
47
48 ' 添加参量
49 cmdUserDelete.Parameters.Add("@FirstName", SqlDbType.VarChar, _
50   50, "FirstName")

```

```

51   cmmUserDelete.Parameters.Add("@LastName", SqlDbType.VarChar, _
52       50, "LastName")
53   cmmUserDelete.Parameters.Add("@LoginName", SqlDbType.VarChar, _
54       50, "LoginName")
55   prmsQLDelete = dadUser.DeleteCommand.Parameters.Add("@Id", _
56       SqlDbType.Int, 0, "Id")
57   prmsQLDelete.Direction = ParameterDirection.Input
58   prmsQLDelete.SourceVersion = DataRowVersion.Original
59
60   cmmUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, _
61       50, "FirstName")
62   cmmUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, _
63       50, "LastName")
64   cmmUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, _
65       50, "LoginName")
66   prmsQLUpdate = dadUser.UpdateCommand.Parameters.Add("@Id", _
67       SqlDbType.Int, 0, "Id")
68   prmsQLUpdate.Direction = ParameterDirection.Input
69   prmsQLUpdate.SourceVersion = DataRowVersion.Original
70
71   cmmUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, _
72       50, "FirstName")
73   cmmUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, _
74       50, "LastName")
75   cmmUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, _
76       50, "LoginName")
77
78   ' 填充视图中的数据
79   dadUser.Fill(dstUser, "tblUser")
80
81   ' 修改第二行中的用户名
82   dstUser.Tables("tblUser").Rows(1)("LastName") = "Thomsen"
83   dstUser.Tables("tblUser").Rows(1)("FirstName") = Nothing
84
85   Try
86       ' 将改变传给数据源
87       dadUser.Update(dstUser, "tblUser")
88   Catch objE As Exception
89       MsgBox(objE.Message)
90   End Try
91 End Sub

```

在程序清单 6.7 中第 2 列被更新了，LastName 列被设为“Thomsen”，FirstName 列被设为 NULL 值。这将激活更新触发器，而触发器将抛出一个异常，该异常会在代码中被捕获并显示出错消息。

这将让你感受到使用触发器的感觉，实际上它们并不难用。只要保证设计良好的数据库没有在使用其他手段（如参照完整性）可以轻易达到的目的上使用触发器就可以了。

6.5 小结

本章讨论了为服务器端数据处理创建各种服务器端对象的方法。讲解了存储过程、视图以及触发器的概念。展示了创建存储过程、在代码中运行和执行存储过程的方法，以及创建、运行和在代码中使用（包括更新）视图的方法。最后介绍了创建触发器的方法。

本章对存储过程、视图以及触发器的细节涉及的不太多，如果你需要更多的信息和样例代码，推荐你读一下 Apress 出的《Code Centric: T-SQL Programming with Stored Procedures and Triggers》，作者 Garth Wells，ISBN: 1-893115-836。

下一章将介绍分层数据库（hierarchical database）。其中将讨论访问 LDAP 协议以及访问网络目录数据库（如 Active Directory）的方法。

第 7 章

分层数据库

通过 LDAP 访问 Active Directory

本章将讨论 LDAP (Lightweight Directory Access Protocol, 轻量级目录访问协议) 和使用 LDAP 连接 Active Directory (Microsoft 的网络目录服务) 的方法。Active Directory 是一种数据库, 在第 2 章已对此进行了全面描述。

7.1 LDAP 一览

因为在示例应用程序中将访问 Active Directory, 所以先简要地介绍一些有关 Active Directory 和 LDAP 的背景知识。与域名系统 (DNS, Domain Name System) 和 Novell 目录服务 (NDS, Novell Directory Services) 一样, Active Directory 也是一种网络目录服务。网络目录服务 (network directory service) 处理下类对象: 用户、打印机、客户以及服务器, 等等。简单来说, 网络目录服务处理用于管理或访问网络的信息。本章包含的示例代码将带着你接触 Active Directory 的各个方面。

Microsoft Windows 2000 操作系统首先引入了 Active Directory。Active Directory 是基于开放式标准协议 (如 LDAP) 且与 X.500 相兼容的网络目录。Active Directory 与 X.500 兼容表示 Active Directory 是带有树状结构的分层数据库。在开发 X.500 目录规范时, 由于需要制订基于 X.500 规范访问网络目录的访问协议, 因而创建了 DAP (目录访问协议, Directory Access Protocol)。但是, DAP 的规范太过庞大而且管理费用很高, 因此, 只有很少的客户和/或应用程序可以连接到 DAP。如果需要有关 X.500 目录标准 (由国际标准组织 ISO 和国际电信联盟 ITU 制订) 的更多信息, 请访问 <http://www.nexor.com/x500frame.htm> 站点。

Michigam 大学的一组研究人员也碰到了这个难题。他们认识到: 削减 DAP 的管理费用可以更快地检索相同的目录信息, 而且客户会变得更小。因此他们创建了新的协议规范, 即 LDAP。



有关构造分层数据库方法, 请参阅第 2 章。

LDAP 直接在 TCP/IP 堆栈上运行。如果要使用 LDAP 访问目录（虽然这意味着 LDAP 是基于客户机-服务器的），那么网络目录服务需要提供 LDAP 服务。

现在，LDAP 成为客户访问目录信息的实际标准。对于工作在 Internet 和 Intranet 中的客户都是如此，标准的 LAN 目录访问现在也转向了 LDAP。

LDAP 协议标准中最重要的一个特征是：LDAP 提供了与平台无关的通用 API（应用程序编程接口）。因而，开发访问目录服务的应用程序，如 MS Active Directory 或 Novell Directory Services（NDS）就变得更加简单且便宜。



注意。

LDAP 目录服务协议虽然提供了访问已存在目录的方法，但是不能创建新

7.1.1 探索 Active Directory

目前，Active Directory 只能在 Windows 2000 中使用。因此，为了理解这里进行的讨论，至少需要安装了 AD 的 Windows 2000 Server。

AD 不仅可被访问，而且可以扩展，这点非常重要。如果要把某对象暴露到网络中，那么可以扩展 AD 的模式，使可以访问 AD 实现的任何用户都可访问该对象。AD 中的模式定义了可以存储在目录中的所有对象类和属性。这并不意味着在模式中定义的对象实际上都在 AD 中，但是，可以说模式允许创建这些对象。对于每个对象类，通过指定类的有效或合法父类，模式都能定义对象在目录中的创建位置。而且，通过类必须或可能包含的属性列表，可以定义类的内容。这意味着 AD 对象是命名的属性集。该属性集是独特的，例如，用于描述用户或打印机。网络中任何具体的事物都可指定为 AD 中的对象。

这里不再讨论 AD 的更多细节。如果你想知道更多细节，可以在下面的列出的站点中查找有关信息：<http://www.microsoft.com/windows2000/technologies/directory/default.asp>。

总而言之，Active Directory 是一种网络目录服务，基于 X.500 规范，并可以使用基于客户机-服务器的 LDAP 目录服务协议进行访问。

7.2 以编程方式访问 AD

当然，Active Directory 可以从 VB.NET 用编程方式进行访问（和 VB6 一样）。它与 VB6 的区别是：VB.NET 有一些内建功能，使你无需通过 Windows API 就可访问 Active Directory。**System.DirectoryServices** 名称空间（namespace）包含了用于该用途的所有功能。

7.2.1 System.DirectoryServices 名称空间

System.DirectoryServices 名称空间包含了许多可以访问 Active Directory 的类。表 7.1 列举了该名称空间中比较重要的类。

表 7.1 System.DirectoryServices 类

类名	说明
DirectoryEntries	该集合类包含了 Active Directory 项的子项 (DirectoryEntry.Children 属性)。请注意, 该集合只包含直接子类
DirectoryEntry	DirectoryEntry 类封装了 Active Directory 数据库分层结构中的一个对象或节点
DirectorySearcher	该类用于执行对 Active Directory 的查询
PropertyCollection	该集合包含了单个 DirectoryEntry 类的所有属性 (DirectoryEntry.Properties 属性)
PropertyValueCollection	PropertyValueCollection 集合包含了多值属性的值 (DirectoryEntry.Properties.Values 属性)
ResultPropertyCollection	该集合包含了 SearchResult 对象的属性 (SearchResult.Collection.Item(0).Properties 属性)
ResultPropertyValueCollection	ResultPropertyValueCollection 集合包含用于多值属性 “()” 的值
SchemaNameCollection	该集合包含了用于 DirectoryEntries 类 SchemaFilter 属性的模式名称列表
SearchResult	该类封装了 Active Directory 数据库分层结构中的节点。该节点作为搜索 (该搜索由 DirectorySearcher 类的实例执行) 的结果返回。利用 DirectorySearcher 类的 FindOne 方法来使用 SearchResult 类
SearchResultCollection	该集合包含 SearchResult 类的实例。当使用 DirectorySearcher.FindAll 方法查询 Active Directory 分层结构时, 返回该集合
SortOption	就该类用定指定查询的排序方式

为使用这些类, 需要注意下列限制条件:

- 必须在计算机中安装了 ADSI SDK (Active Directory Services Interface Software Development Kit) 或 ADSI。如果使用的是 Windows 2000, 则默认安装了这些内容。如果使用的是 Windows 的早期版本, 可以安装从 <http://www.microsoft.com/windows2000/techinfo/howitworks/activedirectory/adsilinks.asp> 下载的 SDK。
- 必须在计算机中安装目录服务提供程序, 如 Active Directory 或 LDAP。

我并不打算在本节讲解 **System.DirectoryServices** 名称空间的所有类, 但由于 **DirectoryEntry** 类非常重要, 所以接下来单独讨论一下。

7.2.2 DirectoryEntry 类

DirectoryEntry 类封装了 Active Directory 数据库分层结构中的对象。可以使用该类来绑

定到 AD 中的对象或操作对象属性。该类和辅助类可用于 IIS、LDAP、NDS 和 WinNT 提供程序。未来可能会有更多的提供程序出现，但本章只示范 LDAP 访问 AD 的用法。

表 7.2 显示了 **DirectoryEntry** 类中非继承的公共属性。

表 7.2 DirectoryEntry 类的属性

属性名	说明
AuthenticationType	该属性返回或设置所用的验证类型，其值必须是 AuthenticationTypes 枚举的成员，且默认值是 None 。下面是 AuthenticationTypes 枚举的其他值： Anonymous 、 Delegation 、 Encryption 、 FastBind 、 ReadonlyServer 、 Sealing 、 Secure 、 SecureSocketsLayer 、 ServerBind 和 Signing
Children	此只读属性返回 DirectoryEntries 类（集合），该类包含 Active Directory 数据库分层结构中节点的子项，且只返回直接子类
Guid	Guid 属性返回 DirectoryEntry 类的全局唯一标识符（Globally Unique Identifier）。如果绑定到 AD 中的对象，则应使用 NativeGuid 属性。此属性为只读属性
Name	此只读属性返回对象的名称。返回值是出现在基础目录服务中的 DirectoryEntry 对象的名称。注意，与此属性一起的 SchemaClassName 属性使目录项唯一化，换句话说，这两个名称可以把一个目录从它的兄弟目录中分离出来
NativeGuid	NativeGuid 属性用于返回 DirectoryEntry 类的全局唯一标识符（就像从提供程序返回那样）。此属性为只读属性
NativeObject	此只读属性返回本地 ADSI 对象。当用户采用的是 COM 接口时，可以使用此属性
Parent	Parent 属性返回 Active Directory 数据库分层结构中 DirectoryEntry 对象的父对象。此属性为只读属性
Password	此属性返回或设置在验证客户时使用的密码。设置了 Password 和 Username 属性后，会自动创建从当前实例得到的 DirectoryEntry 类的所有其他实例，而 Password 和 Username 属性采用相同的设置值
Path	Path 属性返回或设置 DirectoryEntry 类的路径。默认值为空的 String
Properties	此属性返回 PropertyCollection 类（该类包含为 DirectoryEntry 对象设置的属性）
SchemaClassName	SchemaClassName 属性返回用于 DirectoryEntry 对象的模式名称
SchemaEntry	此属性返回 DirectoryEntry 对象（包含用于当前 DirectoryEntry 对象的模式信息）。 DirectoryEntry 类的 SchemaClassName 属性会确定对 DirectoryEntry 实例有效的属性
UsePropertyCache	此属性返回或设置一个 Boolean 值，该值指示在每次操作后是否提交到高速缓存。默认值为 True
Username	Username 属性返回或设置用于验证客户的用户名。设置了 Password 和 Username 属性后，会自动创建从当前实例得到的 DirectoryEntry 类的所有其他实例，而 Password 和 Username 属性采用相同的设置值

要了解 **DirectoryEntry** 所包含的内容，请查阅该类的方法。表 7.3 显示了 **DirectoryEntry** 类的非继承公共方法。

表 7.3 DirectoryEntry 类的方法

方法名	说明	示例
Close()	该方法会关闭 DirectoryEntry 对象。这意味着此处可以释放 DirectoryEntry 使用的任何系统资源	objAD.Close()
CommitChanges()	CommitChanges 方法把对 DirectoryEntry 对象所做的改变保存到 Active Directory 数据库中	objAD.CommitChanges()
CopyTo() As DirectoryEntry	该重载方法用 (或不用) 新名称 (strNewName) 创建 DirectoryEntry 对象的复本, 作为 objADParent 的子对象	objADCopy = objAD. CopyTo(objADParent) 或 objADCopy = objAD. CopyTo(objADParent, strNewName)
DeleteTree()	正如其名称所暗示的一样, DeleteTree 方法从 Active Directory 数据库分层结构中删除 DirectoryEntry 对象和整个子树	objAD.DeleteTree()
Invoke(ByVal vstrMethodName As String, ByVal ParamArray varrArgs() As Object) As Object	该方法用 varrArgs 参数来调用本地 Active Directory 上的 strMethodName 方法	
MoveTo()	这个重载方法用 (或不用) 新的名称 (strNewName) 将 DirectoryEntry 对象移动到 objADParent 父对象中	objAD.MoveTo(objADParent) 或 objAD.MoveTo (objADParent, strNewName)
RefreshCache()	这个重载方法把 DirectoryEntry 对象的属性值载入属性高速缓存中。并载入所有属性值, 或是用 arrstrProperties 参数指定要载入的属性	objAD.RefreshCache() 或 objAD.RefreshCache (arrstrProperties)
Rename(ByVal vstrNewName As String)	Rename 方法将 DirectoryEntry 的名称重命名或改为 vstrNewName	objAD.Rename(strNewName)

LDAP 语法概述

要绑定到 AD 或搜索 AD, 就需要理解用于绑定和查询的 LDAP 语法。因此现在深入讨论 LDAP 语法。

在使用 LDAP 访问时, AD 中的每个对象都由两个名称进行标识。分别是 RDN (相对可分辨的名称, Relative Distinguished Name) 和 DN (可分辨的名称, Distinguished Name)。DN 实际上由 RDN 及其所有父名称及组成。请看下面的示例:

- UserMan 对象的 RDN 为 CN=UserMan。
- 在我的 AD 中, UserMan 对象的 DN 为:

\C=DK\O=UserMan\OU=developers\CN=UserMan。

分层结构中的每个节点都由反斜线 (\) 分隔。在目录服务中, DN 是惟一的。这意味着对每个节点而言, 对象名都是惟一的。如果考虑 DN, 那么在开发者的节点中通用名 (CN, Common Name) UserMan 必须是惟一的。在 UserMan 节点中, 组织单元 (OU, Organizational Unit) developers 必须是惟一的, 依此类推。O=UserMan 中的 O 代表组织, C=DK 中的 C 代表国家, 此例中为丹麦 (Denmark)。如果是美国 (United States), 则使用 C=US。

这些前缀称为 (moniker), 用于标识对象的类别。前面介绍的 CN 代表 Common Name, 是最常用的。分层结构中, 组织单元节点下的多数对象都使用绰号。表 7.4 列举了一些绰号。

表 7.4 常用绰号

绰号名称	说明	示例
Common Name (CN)	所有绰号中最常用的逻辑上称为 Common ? Name。用于 O 节点 OU 节点或对象下的多数对象	CN=UserMan
Country (C)	该绰号用于描述顶级节点	C=DK 或 C=US
Domain Component (DC)	绰号 DC 用于描述域。因为不允许在 RDN 中使用句点 (.), 所以需要使用两个 DC 绰号来描述域	DC=userman, DC=com
Organization (O)	绰号 O 用于描述组织, 通常是公司的名称	O=UserMan
Organizational Unit(OU)	如果组织中有多个单元, 可用绰号 OU 进行描述	OU=developers

下面的示例显示了我的系统中 UserMan 用户完整的 LDAP 路径:

LDAP://CN=UserMan,CN=Users,DC=userman,DC=com

虽然改变了域名, 但是你仍可以得到相似的内容。该示例首先指出其使用的是 LDAP 协议 (LDAP://), 接着是用逗号分开、组成 DN 的 RDN 列表。将该查询翻译成普通语言为: 这是寻找用户 UserMan 的路径, 该用户属于 userman.com 域中的 Users 组。

有时, 使用逗号作为分隔符, 有时则用斜线 (正斜线或反斜线) 作为分隔符。实际上, 这两种分隔符都可以使用。但必须注意, 随着使用不同的分隔符, 放置 DN 的顺序会有所改变。例如, 若用逗号作为分隔符, 那么 DN 就以最后的对象, 或者以分层结构中最低级的对象开头。然后横穿树节点, 直到添加了顶级节点为止。而使用斜线作为分隔符, 顺序则刚好相反即:

LDAP://DC=com/DC=userman/CN=Users/CN=UserMan

多数情况下, 使用哪种分隔符取决于你自己的偏爱。

绑定到 Active Directory 中的对象

要使用 **System.DirectoryServices** 名称空间中的类来完成 AD 中的操作, 必须先绑定对象。绑定对象并不困难, 但是, 如果你对 LDAP 尤其是 LDAP 语法比较陌生, 就会感到困惑。在开始绑定到 AD 前, 确保已经阅读了前面的“LDAP 语法概述”部分。程序清单 7.1 是绑定

到 AD 中特定用户的简单示例。

程序清单 7.1 绑定到 AD 中的对象

```
1 Dim objEntry As DirectoryEntry
2
3 ' 绑定到 userman 对象
4 objEntry = New DirectoryEntry("LDAP://CN=UserMan,CN=Users,DC=userman," & _
5 "DC=com", "Administrator", "adminpwd")
```

在程序清单 7.1 中,用示例说明了 **DirectoryEntry** 对象,并将其绑定到 AD 中的 **UserMan** 用户对象。此外,还指定了用户 **Administrator** 及其账户的密码。在你的系统(至少对 **Active Directory** 具有读权限)中,应改变对这些账户的认证。现在你了解了,只要知道绑定到什么对象,操作就很简单。

7.2.3 搜索 AD 中的对象

要搜索 **Active Directory** 中的特定对象,可以使用 **DirectorySearcher** 类的实例,如程序清单 7.2 所示。

清单 7.2 搜索 AD 中的特定对象

```
1 Public Sub SearchForSpecificObjectInAD()
2     Dim objEntry As DirectoryEntry
3     Dim objSearcher As DirectorySearcher
4     Dim objSearchResult As SearchResult
5
6     ' 实例化并绑定到 AD 中的 Users 节点
7     objEntry = New DirectoryEntry("LDAP://CN=Users,DC=userman,DC=com", _
8     "UserMan", "userman")
9
10    ' 设置对 Users 节点上 UserMan 的搜索
11    objSearcher = New DirectorySearcher(objEntry, _
12    "(&(objectClass=user)(objectCategory=person)" & _
13    "(userPrincipalName=userman@userman.com))")
14
15    ' 找到用户
16    objSearchResult = objSearcher.FindOne()
17
18    ' 检查是否找到了用户
19    如果没有找到, objSearchResult 为空
20    ' 显示用户的路径
21    MsgBox("Users Path: " & objSearchResult.Path)
22    Else
23        MsgBox("User not found!")
24    End If
25 End Sub
```

在程序清单 7.2 中,包括了在 Users 节点上对 UserMan 用户进行定位的 **DirectorySearcher** 对象的实例。使用 **FindOne** 方法刚好返回一个与 **userPrincipalName** 匹配的结果。有关 **userPrincipalName** 的更多信息,请参阅前面的表 7.4。如果在搜索时找到了多个对象,那么只返回第一个对象。如果要返回多个对象,则需要使用 **FindAll** 方法。程序清单 7.3 显示了该示例。

程序清单 7.3 搜索 AD 中特定类的所有对象

```

1 Public Sub SearchForAllUserObjectsInAD()
2   Dim objEntry As DirectoryEntry
3   Dim objSearcher As DirectorySearcher
4   Dim objSearchResult As SearchResult
5   Dim objSearchResults As SearchResultCollection
6
7   ' 实例化并绑定到 AD 中的根域
8   objEntry = New DirectoryEntry("LDAP://DC=userman,DC=com", _
9     "UserMan", "userman")
10
11   ' 设置对 Users 节点上 UserMan 的搜索
12   objSearcher = New DirectorySearcher(objEntry, _
13     "(&(objectClass=user)(objectCategory=person))")
14
15   ' 找到了类用户的所有对象
16   objSearchResults = objSearcher.FindAll()
17
18   ' 检查是否找到了用户
19   如果没有找到,则 objSearchResults 为空
20   ' 通过所有返回的用户循环
21   对于 objSearchResults 中每个 objSearchResult 来说
22     ' 显示用户的路径
23     MsgBox("Users Path: " & objSearchResult.Path)
24   Next
25   Else
26     MsgBox("No users were found!")
27   End If
28 End Sub

```

在清单 7.3 中,返回了类用户在 AD 中的所有对象和排序个人信息。但是,从示例代码中无法看出没有返回对象的所有属性。默认情况下,只返回 **adsPath** 和 **Name** 属性。如果要返回其他属性,则需要指定相应的属性,如程序清单 7.4 所示。

程序清单 7.4 从 AD 节点或对象返回非默认属性

```

1 Public Sub ReturnNonDefaultNodeProperties()
2   Dim objEntry As DirectoryEntry
3   Dim objSearcher As DirectorySearcher
4   Dim objSearchResult As SearchResult
5   Dim objProperty As DictionaryEntry
6

```

```

7   ' 实例化并绑定到 AD 中的 Users
8   objEntry = New DirectoryEntry("LDAP://CN=Users,DC=vb-joker,DC=com", _
9       "UserMan", "userman")
10
11   ' 设置对 Users 节点上 UserMan 的搜索, 并
12   ' 返回非默认属性
13   objSearcher = New DirectorySearcher(objEntry, "&{(objectClass=user)" & _
14       (objectCategory=person)(userPrincipalName=userman@vb-joker.com))", _
15       New String(2) {"sn", "telephoneNumber", "givenName"})
16
17   ' 找到用户
18   objSearchResult = objSearcher.FindOne()
19
20   ' 检查是否找到了用户
21   如果没有找到, objSearchResult 为空
22       ' 显示用户属性
23       For Each objProperty In objSearchResult.Properties
24           MsgBox(objProperty.Key.ToString)
25       Next
26   Else
27       MsgBox("User not found!")
28   End If
29 End Sub

```

如果需要 LDAP 显示名 (代表 AD 中的用户) 的列表, 请访问下面的链接: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netdir/w2k/DN_person.asp。

在程序清单 7.4 中指定了和找到的对象一起返回的额外属性。这由第 15 行的代码完成, 其中添加了由三个属性组成的 **String** 数组, 以作为 **DirectorySearcher** 构造函数的参数。也可使用 **PropertiesToLoad** 属性的 **Add** 和 **AddRange** 方法来添加对象, 如下所示:

```

' 逐一添加属性
objSearcher.PropertiesToLoad.Add("sn")
objSearcher.PropertiesToLoad.Add("telephoneName")
objSearcher.PropertiesToLoad.Add("givenName")
' 批量添加属性
objSearcher.PropertiesToLoad.AddRange(New String(2) _
{"sn", "telephoneNumber", "givenName"})

```

下面的网页提供了构造 LDAP 查询过滤器的更多信息:

- http://msdn.microsoft.com/library/psdk/adsis/glquery_0j3m.htm
- http://msdn.microsoft.com/library/psdk/adsis/ds2prggd_0hfc.htm
- http://msdn.microsoft.com/library/psdk/adsis/ds2prggd_56lw.htm

7.2.4 处理对象属性值

通常, 只需读取从 Active Directory 返回的数据就够了。但有时可能还想处理这些数据。

实际上，一旦绑定了对象，就可以编辑、删除或添加该对象的属性值了。

检查属性是否存在

首先需要学习检查特定属性是否随对象一起被返回。使用 **Properties** 集合的 **Contains** 方法就可实现这个任务。该方法和集合是 **DirectoryEntry** 和 **SearchResult** 类的一部分。下面的表达式用于检查是否返回了 **telePhoneNumber** 属性的方法：

```
If objEntry.Properties.Contains("telePhoneNumber") Then
If objSearchResult.Properties.Contains("telePhoneNumber") Then
```

如果在访问属性之前没有检查该属性是否在 **Properties** 集合中，就可能抛出异常！

对属性使用本地高速缓存

当使用 **DirectoryEntry** 类的实例时，属性和属性值默认处于本地高速缓存（local Cache）中。因为所有改变只应用到本地高速缓存而不会提交给 Active Directory 数据库，所以对 **DirectoryEntry** 对象的访问速度会更快。在第一次读取属性时，该属性就被放到高速缓存中了。

可以用 **UsePropertyCache** 属性改变这种默认行为。其默认值为 **True**，即在本地对属性进行高速缓存。如果将该属性设置为 **False**，那么在每次操作之后，对高速缓存的所有改变都会提交到 Active Directory 数据库中。

如果对 Active Directory 数据库做了改动，则可以调用 **RefreshCache** 方法来更新高速缓存。如果对高速缓存的内容做了改动，那么就应该在调用 **RefreshCache** 之前调用 **CommitChanges** 方法。否则，会覆盖高速缓存中未提交的改变内容。

编辑已存在的属性

编辑 AD 中某对象的已存在属性非常容易。请参看程序清单 7.5 中的示例。

程序清单 7.5 编辑已存在的用户属性

```
1 Public Sub EditUserProperty()
2     Dim objEntry As DirectoryEntry
3     Dim objPropertyValues As PropertyValueCollection
4
5     ' 绑定到 UserMan 用户对象
6     objEntry = New DirectoryEntry("LDAP://CN=UserMan,CN=Users," & _
7         "DC=userman,DC=com", "UserMan", "userman")
8
9     ' 改变用户的电子邮件地址
10    objEntry.Properties("mail")(0) = "userman@userman.com"
11    ' 向 AD 数据库提交改变
12    objEntry.CommitChanges()
13 End Sub
```

在程序清单 7.5 的示例代码中，第 6、7 行绑定到 UserMan AD 对象，第 10 行改变了该用户的电子邮件地址，所做的改变则是在第 12 行提交的。

添加新属性

如果要向 AD 中已存在的对象添加新属性,可以用 **DirectoryEntry** 类的 **Add** 方法来进行。请参看程序清单 7.6 中的示例。

程序清单 7.6 添加新的用户属性

```

1 Public Sub AddNewUserProperty()
2     Dim objEntry As DirectoryEntry
3     Dim objPropertyValues As PropertyValueCollection
4
5     ' 绑定到 UserMan 用户对象
6     objEntry = New DirectoryEntry("LDAP://CN=UserMan,CN=Users," & _
7         "DC=userman,DC=com", "UserMan", "userman")
8
9     ' 添加新的电子邮件地址
10    objEntry.Properties("mail").Add("userman@userman.com")
11    ' Commit the changes to the AD database
12    objEntry.CommitChanges()
13 End Sub

```

在程序清单 7.6 中,通过 **objEntry** 对象的 **Add** 方法向用户 **UserMan** 添加了电子邮件地址,该项操作在第 10 行进行。但是,在此之前必须先绑定到 AD 中的用户对象 **UserMan** (该项操作在第 6、7 行进行)。最后,在第 12 行使用 **CommitChanges** 方法将改变提交给 AD 数据库。

更新 Active Directory 数据库

如果将 **DirectoryEntry** 对象的 **UsePropertyCache** 属性设置为 **False**,则不必再向 Active Directory 数据库提交改变了(该项操作会自动完成)。但是,如果将 **UsePropertyCache** 属性设置为 **True**(默认值),就必须手工提交改变。可以调用 **DirectoryEntry** 类的 **CommitChanges** 方法来提交所做的改变。

7.3 使用 OLE DB .NET 数据提供程序访问 Active Directory

虽然利用 **System.DirectoryServices** 名称空间中的类可以完成绝大多数与 Active Directory 相关的工作。但是,因为 OLE DB .NET 数据提供程序允许用户实现标准的 SQL 语法以提取所需数据,所以使用 OLE DB .NET 数据提供程序来实现能够更为简单。请参看程序清单 7.7 中的示例。

程序清单 7.7 使用 OLE DB .NET 访问 AD

```

1 Dim cnnAD As OleDbConnection
2 Dim cmdAD As OleDbCommand
3 Dim drdAD As OleDbDataReader
4
5 ' 实例化并打开连接

```

```

6 cnnAD = New OleDbConnection("Provider=ADsDSOObject;" & _
7 "User Id=UserMan;Password=userman")
8 cnnAD.Open()
9 ' 实例化命令
10 cmmAD = New OleDbCommand("SELECT cn, adsPath FROM " & _
11 "'LDAP://userman.com' WHERE objectCategory='person' AND " & _
12 "objectClass='user' AND cn='UserMan'", cnnAD)
13 ' 数据阅读器中的检索行
14 drdAD = cmmAD.ExecuteReader()

```

程序清单 7.7 展示了为 userman.com 域的用户 UserMan 从 AD 中检索所有行的方法。需要改变第 7 行中的用户 ID 和密码，以匹配你的 AD 中具有读权限的用户。在第 11 行中，需要改变域名以匹配你所连接的域名。

在第 11 和 12 行中的 **objectCategory** 和 **objectClass** 引用是标准的 LDAP 类型查询调用，它们确保只对用户对象进行搜索。

7.3.1 为连接指定 OLEDB 提供程序

和 OLE DB .NET 数据提供程序一起用于访问 AD 的提供程序只有一个，即 **ADsDSOObject**。在程序清单 7.7 的第 6 行可以找到 **ADsDSOObject**。

一般而言，可以使用下面的代码打开与 AD 的连接：

```

cnnAD = New OleDbConnection("Provider=ADsDSOObject")
cnnAD.Open()

```

显然，必须用对 AD 有读权限的账户才能登录到使用 AD 的系统，但这实际上非常简单！即使你没有登录，或是没有对 AD 的读权限，也可以使用程序清单 7.7 第 6 行和第 7 行中指定的语法。

7.3.2 用 LDAP 协议指定要访问的域

打开 **OleDbConnection** 后，如果要访问 AD，则需指定协议和要访问的 AD 域名。这就像在 **SELECT** 语句中指定表名一样，如下所示：

```
FROM 'LDAP://userman.com'
```

可以看到，协议名和域名放在单引号中。必须这样做，否则会抛出异常。指定协议的过程和使用浏览器时指定 **HTTP** 或 **FIP** 一样，协议名后紧跟冒号和两个斜线 (://)。

程序清单 7.7 是如何构造从 AD 检索行的 **SELECT** 语句的完整示例。

7.3.3 指定要从 AD 获取的信息

与其他 **SELECT** 语句一样，可以告知该命令返回哪些“列”。

表 7.5 显示了可以从 AD 获取的信息列表。请注意，该列表并不全面，它只是学习如何获取用户信息的起点。“等价的用户属性对话框”列提到了可以使用 **Active Directory Users**

and Computers (Active Directory 用户和计算机) MMC 插件可以找到的信息。如果要查看某用户的信息,可以双击该用户,然后从弹出的用户对话框中查看该用户信息。

表 7.5 可以从 Active Directory 获取的用户信息

名称	说明	等价的用户属性对话框
adsPath	对象的完整路径,包括协议报头	无
mail	用户的电子邮件账户。请注意,该账户与在 Exchange Server 中创建的电子邮件账户无关	[General] 选项卡, [E-mail] 文本框
objectSid	表示用户的安全标识符 (SID, Security Identifier)。请注意,该值以字节数组形式返回。如果要以人工可读格式 (S-1-5-21-...) 显示,则必须转换 SID	无
samAccountName	由安全账户管理器 (SAM, Security Account Manager) 使用的名称	[Account] 选项卡, [User logon name(pre-Windows 2000)] 文本框
userPrincipalName	用户主体名称 (UPN, user principal name) 是用户的 Internet 型登录名。UPN 是基于 Internet 标准 RFC 822 的,且通常与“mail”相同	[Account] 选项卡, [User logon name] 文本框和下拉列表。UPN 来自于访问所有服务的登录名的概念,因此 UPN 通常与用户的电子邮件地址相同

请注意,不能像标准 SQL 语句那样用星号 (*) 指定返回所有“列”。如果用了星号,那么查询只返回 **adsPath**。请在以下网页查阅有关 SQL Dialect 的信息: http://msdn.microsoft.com/library/psdk/adsi/ds2prgdd_55pw.htm。

如果要获取有关模式、模式的对象以及对象属性等方面的更多信息,可以访问下面的地址,其中包含了大量有关 Active Directory 模式的信息: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netdir/hh/adsi/glschema3_868x.asp

7.3.4 更新 Active Directory 对象

目前,用于 ADSI 的 OLE DB 提供程序是随 MDAC 2.6 和 2.7 一起提供且是只读的,即不能用来发布 UPDATE 语句。Microsoft 表示会进行修改,但是,是否在 Windows 2002 中对 ADSI 做修改? 或者对 Windows 2000 中的 ADSI 也进行修改? 我还没有得到任何详细的信息。请读者朋友留意!

7.3.5 检索用户 SID

你可能已经注意到,在 UserMan 数据库的 tblUser 表中有名为 ADNmae 和 ADSID 的两列。它们保存了用户的 SID (objectSid) 和 SAM Account Name (samAccountName) (有关

objectSid 和 samAccountName 的描述请参看表 7.5)。考虑程序清单 7.8, 观察检索用户 UserMan 的 SID 和 SAM Account Name 的方法。显然, 要完成这项工作, 必须在自己网络的 Active Directory 中添加 UserMan 用户。

程序清单 7.8 从 AD 检索 SAM Account Name 和 SID

```

1 Public Sub GetSIDAndSAMAccountNameFromAD()
2     Dim cnnAD As OleDbConnection
3     Dim cmdAD As OleDbCommand
4     Dim drdAD As OleDbDataReader
5     Dim strSAMAccountName As String
6     Dim strSID As String
7
8     ' 实例化并打开连接
9     cnnAD = New OleDbConnection("Provider=ADsDSOObject;" & _
10         "User Id=UserMan;Password=userman")
11     cnnAD.Open()
12     ' 实例化命令
13     cmdAD = New OleDbCommand("SELECT objectSid, samAccountName " & _
14         "FROM 'LDAP://vb-joker.com' WHERE objectCategory='person' " & _
15         "AND objectClass='user' AND cn='UserMan'", cnnAD)
16
17     ' 数据阅读器中的检索行
18     drdAD = cmdAD.ExecuteReader()
19     ' Go to first row
20     drdAD.Read()
21     ' 得到 SAM Account 名称
22     strSAMAccountName = drdAD("samAccountName").ToString
23     ' Get SID
24     strSID = drdAD("objectSid").ToString
25 End Sub

```

在程序清单 7.8 中, 使用 Microsoft Directory Services 的 OLE DB 提供程序从 AD 检索到了 SID 和 SAM Account Name。得到返回值后, 将该值保存在局部变量中。我们在第 11 章中将进一步讨论该示例, 并演示将值保存到 UserMan 数据库中的方法。

7.4 小结

本章包括了有关 LDAP 和 Active Directory 的一些简要技术背景, 并介绍了将它们接合起来使用的方法。此外, 还讨论了 System.DirectoryServices 名称空间及其所包括的类, 重点介绍了 **DirectoryEntry** 类和该类的属性与方法。

文中还说明了使用 **System.DirectoryServices** 名称空间中的类及使用 OLE DB .Net 数据提供程序, 并示例了用编程方式建立连接的方法。

虽然 Active Directory 是存储某些数据的独特方法,但是 Active Directory 不会代替关系数据库的地位。Active Directory 用于存储长期的数据对象和极少变化的对象。

在编写本章的时候, .NET Framework Class Library (.NET Framework 类库) 中 Active Directory 对象的功能还未完全实现, 因此本章中的某些代码可能无法运行。虽然我已尽力使代码可以运行, 也愿意包含更多详细的示例, 但是时间和程序错误都不允许这么做。不过, 在本书的其余部分会更新本章的示例代码, 在 Apress 的 Web 站点也可找到更新的示例代码。

下一章将介绍消息队列, 并讲述使用消息队列进行无连接编程 (Connectionless Programming) 的方法。

第 8 章

消息队列

在无连接的程序设计中使用时消息队列

如果你不使用固定连接，或者使用不稳定的连接，那么消息队列对你来说无疑是很好的选择。如果只想简单地将输入进行转储，随后就返回到应用程序，而不等待来自对输入进行处理的应用程序或者服务器的输出，使用消息队列也会很方便。可以将消息队列作为自动防故障应用程序的基础，随后在常规数据库发生故障时就可以使用它们，使用事务型队列可以确保不丢失数据，缩短或者消除停机时间。很明显，并不是所有应用程序都可以从使用消息队列中获益，但是用在存储或者接收顾客输入方面它无疑是一个很棒的解决方案。要记住，有些情况下使用消息队列没有好处，比如需要实时处理的银行交易就是一个例子。

Microsoft 的消息传递技术称作 *Message Queuing*。就像 Microsoft 一向惯用的市场宣传手段一样，该技术在引入到 Windows 2000 中去之前用的是别的名字。然而，Windows 2000 中包含的消息队列版本是建立在该技术的旧版本基础上的，其中加入了活动目录（Active Directory，AD）集成方案。Active Directory 集成方案非常好用，利用它可以更快地定位你所在域的公共消息队列，并能更方便地进行企业版范围的备份工作。

MSMQ 已经被废弃了吗？

Microsoft 为它的消息排队技术 1.0 版取名为 Microsoft 消息队列服务器（Microsoft Message Queue Server，MSMQ）。该技术首先被引入到 Windows NT 4 企业版中，其 2.0 版变为了 Windows NT 4 所有服务器版本的附件，是 Windows NT 可选项安装包中的一部分。Windows NT 可选项安装包可以从 Microsoft 的 Web 站点上免费下载。之后，消息队列技术被集成到操作系统中，比如 Windows 2000 以及即将发布的 Windows 2002 以及 Windows XP 中都集成了该技术。MSMQ 的改变不仅仅是集成到操作系统中，它的名称也变成了 *Message Queuing*，并在很多方面得到了扩展。扩展之一就是它与 Active Directory 结合起来，而不再基于 SQL 服务器。

现在我们来回答这个问题：“MSMQ 废弃了吗？”，答案只有“是”或者“否”。从某

个角度来说，答案是“是”，因为它不再是一个孤立的产品。但从另外一个角度来看，答案又是“否”，因为原来那个 MSMQ 的特性已经作为一种服务加入到操作系统中了，称作 Message Queuing。

8.1 无连接程序设计

无连接程序设计 (connectionless programming) 涉及的内容是使用队列将输入保留起来，以供其他应用程序或者服务器使用。在这种意义下，无连接意味着不需要拥有与应用程序或者服务器之间的固定连接，而是将输入记录在队列中。这样，其他应用程序或者服务器就可以从该队列中取出该输入，对其进行处理，再在随后某个时间将结果放回同一个队列或者另一个队列中。

与无连接相反的是面向连接 (connection oriented)，当今所有的标准数据库系统基本上都可以归为后者。你也许听说过与 TCP/IP (传输控制协议/网际协议, Transmission Control Protocol/ Internet Protocol) 有关的两种形式，协议中的 UDP (用户数据报协议, User Datagram Protocol) 部分是无连接的，而 TCP 部分是面向连接的。然而，消息队列和 UDP 协议之间存在的一个主要区别是，UDP 协议不能保证交付，使用 UDP 传输协议沿网络移植的包可能会丢失，而消息队列不会造成丢包。消息队列被归为无连接是因为客户端不与服务器直接“对话”，不像标准数据库那样存在直接的连接。

到目前为止，你应该对无连接程序设计已经有了初步了解。现在再介绍一下 MessageQueue 类。

8.2 MessageQueue 类概览

MessageQueue 类是 System.Messaging 名称空间的一个组成部分，这个名称空间中还包含有 Message 类，该类用来与 MessageQueue 类配合使用。System.Messaging 名称空间中还包含多个专门用于访问以及操纵消息队列的类。

MessageQueue 类是对 Message Queuing 的包装，Message Queuing 是 Windows 2000 以及 Windows NT 中的一个可选组件。对于 Windows NT 来说，该组件是可选项安装包的一个组成部分。在尝试运行本章中的示例之前，必须先确认你已经在网络中的服务器上安装了消息队列。

8.3 何时使用消息队列

如果你需要完成下列任务，就可以考虑在应用程序中使用消息队列：

- 当 DBMS (数据库管理系统, Database Management System) 忙于为顾客提供服务时，会将不太重要的信息存储到消息队列中。然后在 DBMS 有空闲时再对消息队列中的信息进行批处理。一种比较好的做法是，对所有实时请求进行处理，同时将其

他请求全都放入消息队列中待稍后再处理。

- 通过广域网（比如 Internet）连接到 DBMS，但由于某种原因该连接并不十分稳定，不能保持可用。在设计应用程序时可以考虑两种做法。一种做法是让应用程序以常规方式访问 DBMS，如果连接超时或者发生其他连接故障，则将请求存储到消息队列中，随后再启动一个组件来检查 DBMS 是否可用，如果可用就将存储在消息队列中的请求转发给 DBMS，一旦有结果就将它返回给应用程序。此时应用程序就可以恢复正常运行，并且关闭消息队列组件。另一种可选的方案是将应用程序设计成脱机执行，也就是说将所有对 DBMS 的请求都存储到本地消息队列中，然后再由消息队列组件负责将这些请求转发给 DBMS。
- 与大型机系统（比如 SAP）交换数据时可以使用消息队列来保持 SAP 的 IDocs，并由队列管理器或者队列组件负责将 IDocs 发送给 SAP，或者从 SAP 接收这些 IDocs。曾经使用过 SAP 的人都知道，与 SAP 进行实时通信常会遇到困难，因为服务器在每天的某些时段通常都会超负荷运行。因此，应该使用消息队列来存储和转发。要知道 SAP 原来是大型机上的系统，但是当今 SAP 常常运行在中型机甚至 PC 上。R/3 上 5 种最流行的平台依次是 Windows NT、Sun Solaris、HP UX、AS/400 以及 AIX。

根据你自己当前或者以前的工作经验，肯定能够提出几种从消息队列技术获益的应用程序。那么它对销售人员有没有用呢？如果他们能够将某些信息先存储在消息队列中，等回到办公室之后再对这些信息进行成批处理，是不是很理想呢？

为什么要使用消息队列而不是数据库表？

许多场合下消息队列都很好用，但是它与数据库表相比究竟有什么不同呢？当你调用 DBMS 时，操作一般是同步完成的，但是如果用的是消息队列的话操作就是异步完成的。这意味着访问消息队列通常比访问数据库表速度要快。如果客户端应用程序使用同步访问，它必须等待服务器的响应。而如果使用异步访问，客户端应用程序只需把查询之类的操作发送给服务器即可，随后就可以继续自己的工作。当服务器完成对该查询的处理之后，它将通知应用程序，并将结果发送给应用程序。

如果使用数据库表，在加入数据时必须遵守极为严格的格式规范。某些字段是必须提供的，而且很可能无法添加额外的信息。虽然消息队列也同样要求提供某些字段，但是它却允许添加额外的信息，而这种信息是难以在数据库表中进行存储的。

对于 Windows 表单和 Web 表单来说这种方案是可行的，但是它同时也取决于应用程序类型（application type）。分布式应用程序、位于本地网上的标准客户端服务器应用程序以及单机应用程序构成了不同的应用程序类型。假如你的应用程序不是分布式的，并且要访问位于本地服务器或者客户机上的标准 DBMS，就没有理由使用消息队列。实际上，一个可能的原因是你的 DBMS 系统整日处于繁忙状态，因而你希望将请求和消息存储在消息队列中，以供 DBMS 在稍有空闲时能对这些请求和消息进行批处理。

简而言之，应用程序的基础构造以及商务方面的需求通常决定了是否应该采用消息队列技术。而且，如果借助其他工具的话，可以更好地了解在某种场合下是否应该采用这种技术。如果你对消息队列技术还不熟悉，一定要尝试运行一下本章后面的示例代码。

消息队列的运行机制是“先进先出”（First In, First Out）原则，也就是说，如果使用 **MessageQueue** 类中的 **Receive** 或者 **Peek** 方法，将会取出队列中的首条消息。如果想要了解关于这些方法的详细信息，请参阅本章后面的“检索消息”以及“取出消息”。

由于在队列中放置消息时所采用的是先进先出原则，所以新到达的消息将会放置在消息队列的尾部。这很正确，但是仅仅是消息到达的时间并不能决定它在消息队列中的位置。每个消息还包含有一个优先级，由它决定消息在队列中的放置位置。优先考虑的是消息优先级而不是消息的到达时间，因此可以把指定的某个消息放在消息队列的头部，即使消息队列中已经有其他消息也可以做到这点。下面的 **SELECT** 语句可以用来描述消息插入队列的次序以及从队列中取出的次序：

```
SELECT * FROM MessageQueue ORDER BY Priority, ArrivalTime
```

很明显，该语句只是一个示例而已，但是它说明了消息的排序方式：首先根据优先级排序，在优先级相同的情况下则根据到达时间排序。并不能简单地设定优先级，因而大部分消息具有同一个优先级，但是有时“插队”是确实必要的。如果想要详细了解如何定制消息的优先级，请参阅本章后面的“定制消息优先级”。

8.4 如何使用消息队列

当然，在将消息队列应用到应用程序中之前，必须先建立消息队列。也许你已经拥有一个可用的消息队列了，但是还是有必要先说明一下建立消息队列的过程。要注意，如果你处于工作组环境下，则无法查看或者管理公有队列。公有队列只存在于域环境下。请参阅你的 Windows 2000 或者 Windows NT 文档查找 Message Queuing 的安装步骤。假如在工作组模式下运行，则无法使用服务器资源管理器查看队列，就连私有队列也不行。下一节将讨论私有队列和公有队列。

8.4.1 私有队列与公有队列

在创建这两种队列之前，你必须先了解它们之间的区别。表 8.1 对这两种消息队列的某些特性进行了比较。

表 8.1 私有队列与公有队列

私有消息队列	公有消息队列
不在消息队列信息服务（Message Queue Information Service, MQIS）数据库中发布	在消息队列信息服务（Message Queue Information Service, MQIS）数据库中发布
只能在本机创建	必须在目录服务中注册
可以脱机创建或者删除	必须联机创建或者删除

续表

私有消息队列	公有消息队列
其他消息队列应用程序无法定位该队列，除非得到消息队列的完整路径	其他消息队列应用程序可以通过消息队列信息服务（Message Queue Information Service, MQIS）数据库来定位该队列
是固定的，虽然可以进行备份工作但却很困难	是固定的，可以通过 Active Directory 在企业版级别上进行备份

8.4.2 通过编程创建队列

在第 4 章介绍了关于如何使用服务器资源管理器创建队列方面的内容。但是如果想要通过编程来创建队列，请继续阅读以下内容。

通过编程来创建消息队列非常方便。程序清单 8.1 和程序清单 8.2 展示了在本地机器 USERMANPC 上创建队列的两种不同方法。

程序清单 8.1 使用机器名创建私有消息队列

```

1 Public Sub CreatePrivateQueueWithName()
2     Dim queUserMan As New MessageQueue()
3
4     queUserMan.Create( "USERMANPC\Private$\UserMan")
5 End Sub

```

程序清单 8.2 使用默认名创建私有消息队列

```

1 Public Sub CreatePrivateQueue()
2     Dim queUserMan As New MessageQueue()
3
4     queUserMan.Create(".\Private$\UserMan")
5 End Sub

```

在程序清单 8.1 中指定了机器名作为路径参数的一部分，而在程序清单 8.2 中只是简单地使用了一个句点。句点起的是缩略符的作用，它通知 **MessageQueue** 对象在本机上创建队列。还需要注意的是在队列名前加上了前缀 **Private\$**，这是为了指明队列是私有队列。如果不使用该前缀，就说明要创建的是一个公有队列。路径中的各部分彼此之间用反斜杠分开，这与标准 DOS 文件路径的做法是一样的。

MachineName\Private\$\QueueName

公有队列的创建方法与私有队列大同小异，只是前者在路径中没有 **Private\$** 这个部分。也可以使用句点来指明要将队列创建在本机上，如程序清单 8.3 所示。

程序清单 8.3 使用默认名创建公共消息队列

```
1 Public Sub CreatePublicQueue()
2     Dim queUserMan As New MessageQueue()
3
4     queUserMan.Create(".\UserMan")
5 End Sub
```

如前所述，句点可以用本机名来代替。如果要求在某一台机器上创建消息队列，则可以将句点替换为该机器的机器名。



如果试图创建一个已经存在的消息队列，将会抛出一个 **QueExists** 异常。

MessageQueue 类中的 **Create** 方法存在两个重载版本，刚才看到的是其中较为简单的一个。另一个版本有另外一个参数，它是一个 **Boolean** 值，指明消息队列是否是事务型的。我们将在本章后面的“创建事务型消息队列”一节中介绍消息队列事务处理，现在先看一下程序清单 8.4，它说明了如何创建事务型的消息队列。

程序清单 8.4 创建事务型私有消息队列

```
1 Public Sub CreateTransactionalPrivateQueue()
2     Dim queUserMan As New MessageQueue()
3
4     queUserMan.Create(".\Private$\UserMan", True)
5 End Sub
```

在程序清单 8.4 中，利用事务处理支持在本机上创建了一个名为 **UserMan** 的私有消息队列。如果运行了程序清单 8.4 中的示例，之后最好将其删除，否则它可能会影响到本章后面其他示例的运行。

8.4.3 显示或更改消息队列的属性

消息队列的某些属性可以在创建后进行更改。在 Windows 2000 时，其管理工具（Administrative Tools）中的 Computer Management MMC 插件即可完成这项工作。在打开 Computer Management 插件之后，请按照以下步骤执行：

1. 展开 [Services and Applications] 节点。
2. 展开 [Message Queuing node] 节点。
3. 展开带有所请求队列的节点，并选中该队列。当前 Computer Management MMC 插件的外观应该如图 8.1 所示。
4. 用鼠标右键单击该队列，从弹出的菜单中选取 [Properties]，系统将会弹出 [queue Properties] 对话框（如图 8.2 所示）。

如图 8.2 所示，队列是否为事务型是队列创建后无法更改的选项之一。因此在创建队列前明确是否需要事务处理支持就更加重要了。

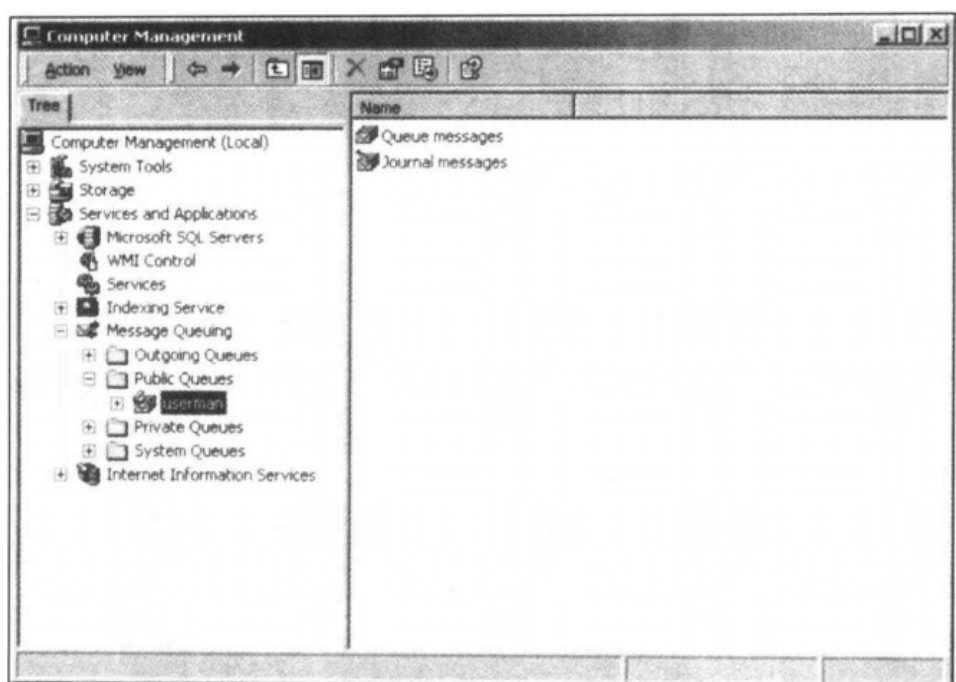


图 8.1 选中消息队列之后的 Computer Management MMC 插件

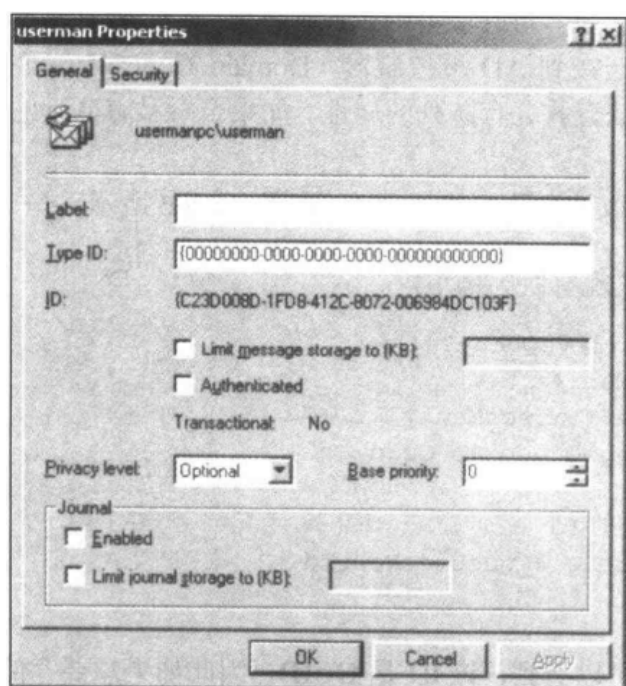


图 8.2 消息队列的 [Properties] 对话框

为消息队列分配标签

如果想要使用标签绑定到消息队列，则必须先为队列指定一个标签。看一下图 8.2，请注意，可以在 [Label] 文本框中输入文本。有一点必须记住，并不要求消息队列一定要使用与其他消息队列不同的标签，但是如果使用了重复的标签，则在绑定到队列或向队列发送消息时就会碰到麻烦。因此，在这里要强调的一点就是，必须保证标签是惟一的！

你也可以通过编程来更改消息队列的标签。请参阅程序清单 8.5 中的示例代码。

程序清单 8.5 更改现有队列的标签

```
1 Public Sub ChangingQueueLabel()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3
4     queUserMan.Label = "USERMAN1"
5 End Sub
```

程序清单 8.5 中，在本机上是使用友好名（friendly name）绑定到现有的私有队列上。友好名是更具可读性并且更容易解释的名称。在程序清单 8.5 的示例代码中，友好名告诉你队列是在本机上创建的（通过“.”来说明），该队列是私有队列（通过 Private\$来说明），名为 UserMan。如果想要通过编程来更改某个消息队列的标签，可以使用该友好名绑定到该队列！第 4 代码行对标签做了实际改动，通过设置 label 属性也可做到这一点。

检索消息队列的 ID

如果想利用公有消息队列的 ID 来绑定到该队列，则可以通过 Computer Management MMC 插件来获取 ID。请参阅本章后面的“显示或更改消息队列的属性”一节，获取关于如何显示消息队列属性的信息。再看一下图 8.2，你会发现 ID 就在 [Type ID] 文本框下边！有必要提醒一下，只有在连接到 AD 域控制器（Domain Controller）的情况下，才能够看到该 ID。如果未连接，就无法查看公有队列的属性。如果查看私有队列的 ID，你会发现它是空白的。

可以通过编程来获取消息队列的 ID，如程序清单 8.6 所示。

程序清单 8.6 获取现有队列的 ID

```
1 Public Sub RetrieveQueueId()
2     Dim queUserMan As New _
3     MessageQueue(".\UserMan")
4     Dim uidMessageQueue As Guid
5
6     ' 保存消息队列的 Id
7     uidMessageQueue = queUserMan.Id
8 End Sub
```

从程序清单 8.6 中可以看到，该 ID 是 GUID，因此如果想要存储它，则必须将变量的数据类型声明为 Guid。必须连接到 AD 域控制器以获取消息队列的 ID，否则将导致异常的出现。如果试图检索私有消息队列的 ID，系统则会返回一个“空的”GUID，即 00000000-0000-0000-0000-000000000000。可以使用 Guid 类中的 ToString 方法来显示 GUID。

8.4.4 绑定到现有的消息队列

创建好消息队列之后，或者如果想要访问现有队列，可以绑定到该队列，如以下各节中所述。

使用友好名绑定

程序清单 8.7 中展示了绑定到现有队列的三种不同方法：使用 `New` 析构函数，使用所谓的友好名（`.\Private$\UserMan`），或是指定消息队列路径以及创建该队列时指派的名称。

程序清单 8.7 绑定到现有队列

```
1 Public Sub BindToExistingQueue()
2     Dim queUserManNoArguments As New MessageQueue()
3     Dim queUserManPath As New MessageQueue(".\Private$\UserMan")
4     Dim queUserManPathAndAccess As New MessageQueue(".\Private$\UserMan", _
5         True)
6
7     ' 初始化队列
8     queUserManNoArguments.Path = ".\Private$\UserMan"
9 End Sub
```

第 2 行所使用的是一种最简单的方法，但是它需要与额外的一行代码相配合，因为使用该方法时并未告知消息队列，对象到底绑定到哪一个队列。第 8 行负责使用 **Path** 属性来指定队列。第 4 行在指定队列的同时还指定了对读访问的限制。如果第二个参数像程序清单 8.7 中一样设为 **True**，那么访问该队列的第一个应用程序就可以确保获得惟一的读访问权。这意味着无法从队列中读出其他 **MessageQueue** 类的实例，因此在使用该选项时必须慎重小心。

使用 Format Name 绑定

由于无法在处于工作组模式下的机器上创建队列，也无法绑定到种机器上的队列，因此要访问队列所在的机器必须是 **Active Directory** 的一部分。然而，当想要访问的消息队列服务器无法访问主域控制器时，就会出现问题。由于这时将使用 **Active Directory** 来解析路径，所以无法使用程序清单 8.6 中的语法来绑定到消息队列。

幸好有办法可以绕过这个限制。不使用前面程序清单中的友好名格式，使用格式名或者标签。程序清单 8.8 说明了如何使用格式名来绑定到现有队列。

程序清单 8.8 使用格式名来绑定到现有队列

```
1 Public Sub BindToExistingQueueUsingFormat()
2     Dim queUserManFormatTCP As New _
3         MessageQueue("FormatName:DIRECT=TCP:10.8.1.15\Private$\UserMan")
4     Dim queUserManFormatOS As New _
5         MessageQueue("FormatName:DIRECT=OS:USERMANPC\UserMan")
6     Dim queUserManFormatID As New _
7         MessageQueue("FormatName:Public=AB6B9EF6-B167-43A4-8116-5B72D5C1F81C")
8 End Sub
```

在程序清单 8.8 中，使用 **TCP** 协议绑定到机器上名为 **UserMan** 的私有队列，该队列位于 IP 地址为 10.8.1.15 的机器上，如第 2 行和第 3 行的代码所示。第 4 行和第 5 行代码则将名为 **UserMan** 的公有队列绑定到机器名为 **USERMANPC** 的机器上。当然，也可以用 **SPX** 协议进行绑定。如果要使用 **SPX** 网络协议，则必须使用下列语法：

FormatName: DIRECT= SPX: NetworkNumber; HostNumberQueueName。

可以选择“任意格式名”选项用来连接到公有队列和私有队列，因此在程序清单 8.8 的示例代码中，公有队列和私有队列绑定可以在三个不同格式名中的任意两个之间进行交换。程序清单 8.8 显示的最后一个格式名在第 6 行和第 7 行，使用了消息队列的 ID 作为队列的标识符。注意，示例代码中使用的 ID 是虚构的，与网络上的 ID 不同。该 ID 的类型是 GUID，是在创建时由 MQIS 生成的。如果想要了解获得消息队列 ID 的详细信息，请参阅“检索消息队列的 ID”一节。

使用标签绑定

现在介绍最后一种绑定到现有消息队列的方法，也就是使用标签语法。当处于脱机状态时无法使用该语法，只有在连接到 Active Directory 域控制器时才能够使用。程序清单 8.9 说明了如何使用 UserMan 标签来连接到消息队列。如果想要了解关于设置消息队列标签的更多信息，请参阅本章前面的“为消息队列指定标签”一节

程序清单 8.9 使用标签绑定到现有队列

```
1 Public Sub BindToExistingQueueUsingLabel()
2     Dim queUserManLabel As New MessageQueue(" Label:UserMan")
3 End Sub
```

至此，我们已经讲述了如何绑定到消息队列。接下来将演示一下如何发送消息、检索消息以及如何进行其他与消息有关的操作。

8.4.5 发送消息

发送消息无疑是消息队列最重要的特征之一。如果不能发送消息，那还要消息队列干什么呢？我们先来看一下向消息队列发送消息的最简单方式。**MessageQueue** 类中的 **Send** 方法就是用来完成这项工作的，程序清单 8.10 说明了这一点。

程序清单 8.10 向消息队列发送一条简单消息

```
1 Public Sub SendSimpleMessage()
2     Dim queUserMan As New _
3     MessageQueue(".\Private$\UserMan")
4
5     ' 向队列发送简单消息
6     queUserMan.Send("Test")
7 End Sub
```

在绑定到本机上的私有队列之后，向该队列发送一个包含有文本“Test”的 **String** 对象。虽然这并不实用，但可以用来测试是否向队列发送了数据。就像你猜到的那样，**Send** 方法被重载了，在示例中使用的那个版本只有一个参数，且该参数是一个对象。

执行程序清单 8.10 中的代码后，可以用服务器资源管理器或者 **Computer Management** MMC 插件查看结果消息。展开私有消息队列 **UserMan**，选定 [Queue Messages] 节点。现在

Computer Management 插件应该呈现为图 8.3 中的样子，如果使用的是服务器资源管理器，则应呈现为图 8.4 中的样子。请注意，在服务器资源管理器中，只有在展开 [Queue Messages] 节点后才能看到队列中的消息，就像图 8.4 中那样。

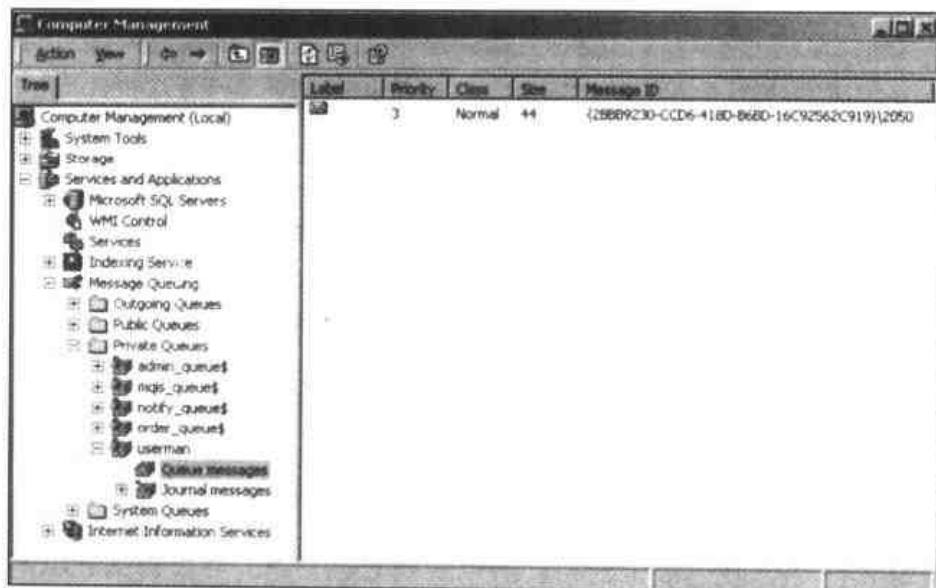


图 8.3 选定 [Queue Messages] 节点时的 Computer Management

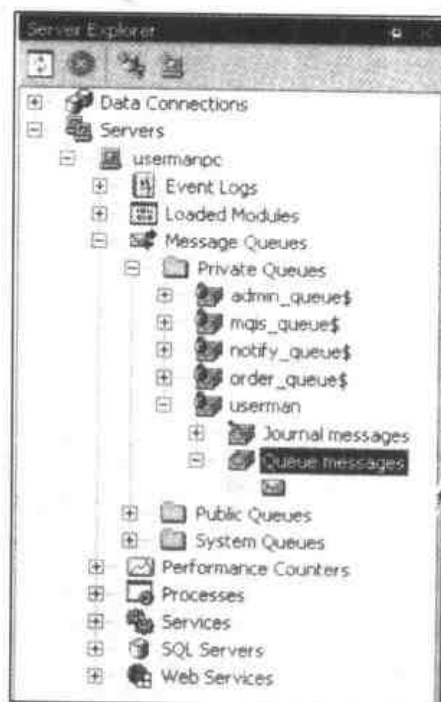


图 8.4 选定 [Queue Messages] 节点时的服务器资源管理器



注意 Computer Management 与服务器资源管理器的区别在于，前者可以在不连接到 Active Directory 域控制器的情况下使用，而服务器资源管理器却不可能的，即使你查看的是私有消息队列也一样！

8.4.6 获取消息

很明显，能够检索投递到消息队列中的消息是很必要的，否则消息就会堆满消息队列服务器，这绝对没有好处。可以通过许多种方法来从消息队列中检索消息，但是我们先讨论检索程序清单 8.10 中所发送消息的方法。请参阅程序清单 8.11，其中的代码检索到消息队列中的首条消息。

程序清单 8.11 检索消息队列中的首条消息

```
1 Public Sub RetrieveSimpleMessage()  
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")  
3     Dim msgUserMan As Message  
4  
5     ' 检索队列中的首条消息  
6     msgUserMan = queUserMan.Receive()  
7 End Sub
```

在程序清单 8.11 中，我们绑定到私有消息队列 **UserMan**，并检索队列中的首条消息。在消息队列中检索到该消息之后，它就从消息队列中删除了。如果既想从队列中读取消息，同时又不希望从队列中删除该消息，那么请参阅本章后面的“抽取消息”以获取更多有关信息。

虽然使用程序清单 8.11 中的示例代码可以检索消息队列中的首条消息，但是它无法完成更多的工作。如果试图访问 **msgUserMan** 的任何属性，就会导致抛出异常，因为尚未建立一个所谓的格式符（**formatter**）来读取该消息。消息可能以多种形式存在，因此在从队列中获取消息之前一定要先建立格式符。请参阅下一节“建立消息格式符”以获取这方面的详细信息。

接收与抽取

MessageQueue 类中的 **Receive** 和 **Peek** 方法都是同步的，也就是说这两个方法会在收到消息前阻塞其他所有操作。如果消息队列中没有消息存在，则 **Receive** 和 **Peek** 方法会等待消息到达队列。如果需要采用异步访问，则必须使用 **BeginReceive** 方法。实际上，在调用 **Receive** 和 **Peek** 方法时可以指定一个超时值，以避免无限期地等待下去。这两个方法都有重载版本，它们使用一个 **TimeSpan** 参数，如果在队列中找到了消息，或者经过了 **TimeSpan** 指定的时间之后仍未等到消息，它们都将返回。如果是由于超时而返回，则将抛出一个异常。

建立消息格式符

回想一下，必须建立一个格式符以便读取从消息队列中检索到的消息。这项工作很容易，但是也很重要。**MessageQueue** 类的 **Formatter** 属性正是用于此用途的。在实例化一个 **MessageQueue** 类对象的时候，就创建了一个默认的格式符，但是这个格式符不能用来从队

列中读取消息，只能用来向队列中写消息或者发送消息。这意味着要么修改默认的格式符，要么建立一个新的格式符。

请参阅程序清单 8.12 中的示例代码，它可以检索并读出程序清单 8.10 中发送的消息。需要提醒的是，如果已经检索到来自队列的消息，队列就变成空的，在下一次尝试从队列中检索消息之前，必须再次发送消息到该队列中去。如果消息队列是空的，**Receive** 方法就会等候新消息到达队列。

程序清单 8.12 建立新格式符并检索队列中的首条消息

```

1 Public Sub RetrieveMessage()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As Message
4     Dim strBody As String
5
6     ' 建立格式符
7     queUserMan.Formatter = New XmlMessageFormatter( New Type() _
8         {GetType(String)})
9
10    ' 检索来自队列的首条消息
11    msgUserMan = queUserMan.Receive
12    ' 保存消息主体
13    strBody = msgUserMan.Body.ToString
14 End Sub

```

在程序清单 8.12 中指出了格式符的类型为 **XmlMessageFormatter**，这是消息队列使用的默认格式符，用来串行化或非串行化使用 XML 格式的消息主体。此外，还指定消息格式符接受数据类型为 **String** 的消息主体。在第 7 行和第 8 行代码完成了这项工作。如果不这样做，就无法访问消息主体。

在发送消息和接收消息时一般使用同一格式符，但上面的示例是非常简单的，在向消息队列发送消息时没有用到格式符。下面，我们来看看，在什么情况下必须用到 **Formatter** 属性。我们在程序清单 8.13 中建立了一个格式符，它可以读取不同数据类型的消息主体：数据类型为 **String** 的主体和数据类型为 **Integer** 的主体。通过向析构函数中的惟一一个参数（**Type** 数组）添加这两种数据类型便可以做到这一点。

程序清单 8.13 发送和检索来自消息队列的不同类型的消息

```

1 Public Sub SendDifferentMessages()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 建立格式符
6     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
7         GetType(String), GetType(Integer))
8
9     ' 创建文本消息主体
10    msgUserMan.Body = "Test"

```

```

11    ' 向队列发送消息
12    queUserMan.Send(msgUserMan)
13    ' 创建 interger 型主体
14    msgUserMan.Body = 12
15    ' 向队列发送消息
16    queUserMan.Send(msgUserMan)
17 End Sub
18
19 Public Sub RetrieveDifferentMessages()
20     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
21     Dim msgUserMan As Message
22     Dim strBody As String
23     Dim intBody As Integer
24
25     ' 建立格式符
26     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
27         GetType(String), GetType(Integer))
28
29     ' 检索来自队列的首条消息
30     msgUserMan = queUserMan.Receive
31     strBody = msgUserMan.Body.ToString
32     ' 检索来自队列的下一条消息
33     msgUserMan = queUserMan.Receive
34     intBody = msgUserMan.Body
35 End Sub

```

我们在程序清单 8.13 中先用 `SendDifferentMessages` 过程向队列发送了两条消息：一个数据类型为 `String` 的消息主体，以及一个数据类型为 `Integer` 的消息主体。接着，再在 `SendDifferentMessages` 中检索这两条消息，检索到的次序与发送次序相同。“可以发送主体类型不同的消息”这一事实说明，在要交换的数据结构差异甚大的情况下，消息队列比数据库表更适于处理数据交换。

在上述例子中，我们用 `String` 和 `Integer` 数据类型作为消息主体格式。在从队列中读出消息时可以非串行化这些类型，但是事实上可以使用任何基本数据类型、结构和受控对象。如果需要传递老式的 COM/ActiveX 对象，则应使用 `ActiveXMessageFormatter` 类。也可以使用 `BinaryMessageFormatter` 类以二进制格式串行化或非串行化消息，这与 `XmlMessageFormatter` 类使用的 XML 格式形成了鲜明的对照。

8.4.7 抽取消息

有时只想看一看消息内容，而并不想从队列中删除该消息，所以不能使用 `MessageQueue` 类的 `Receive` 方法。可以使用 `MessageQueue` 类中的 `Peek` 方法实现上述操作。该方法的功能与 `Receive` 方法一样，只是不会从队列中删除取出的消息。另外，若重复调用 `Peek` 方法，则会得到同样的消息，除非有一个具有更高优先级的消息插入到队列之中。如果了解这方面的详细信息，请参阅本章后面的“定制消息优先级”。

程序清单 8.14 实际上是程序清单 8.12 的近乎完全相同的副本，只是在第 11 行调用了 `Peek`

方法。所以，如果只查看队列中的首消息，则 **Peek** 方法和 **Receive** 方法的效果是完全一样的，它们只是在是否删除消息上有所区别。如果要了解关于删除消息的详细信息，请参阅“清除队列中的消息”。

程序清单 8.14 抽取队列中的消息

```

1  Public Sub PeekMessage()
2      Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3      Dim msgUserMan As Message
4      Dim strBody As String
5
6      ' 建立格式符
7      queUserMan.Formatter = New XmlMessageFormatter(New Type() _
8          {GetType(String)})
9
10     ' 抽取队列中的第一条消息
11     msgUserMan = queUserMan.Peek
12     ' 保存消息主体
13     strBody = msgUserMan.Body.ToString
14 End Sub

```

8.4.8 从队列中选取特定消息

消息队列的结构类似于堆栈的，它不具有任何查找功能，也就是说消息总是由栈顶取出的，或者说由消息队列的顶部取出。然而，要从队列中选取特定的消息是可以做到的。可以用 **ReceiveById** 方法达到这个目的，它使用的惟一个参数是消息 ID。消息 ID 由 **Message Queuing** 生成，并在将 **Message** 类实例发送到队列中时，为该实例指派一个消息 ID。程序清单 8.15 中显示了如何从消息中检索其 ID，之后再使用该 ID 来检索消息，即使消息不在消息队列顶部也可以检索到它。

程序清单 8.15 依据消息 ID 从队列中检索消息

```

1  Public Sub RetrieveMessageById()
2      Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3      Dim msgUserMan As New Message()
4      Dim strId As String
5
6      ' 建立格式符
7      queUserMan.Formatter = New XmlMessageFormatter(New Type() _
8          {GetType(String)})
9
10     ' 创建文本消息主体
11     msgUserMan.Body = "Test 1"
12     ' 向队列发送消息
13     queUserMan.Send(msgUserMan)
14
15     ' 创建文本消息主体

```

```

16 msgUserMan.Body = "Test 2"
17
18 ' 向队列发送消息
19 queUserMan.Send(msgUserMan)
20 strId = msgUserMan.Id
21 ' 创建文本消息主体
22 msgUserMan.Body = "Test 3"
23 ' 向队列发送消息
24 queUserMan.Send(msgUserMan)
25
26 msgUserMan = queUserMan.ReceiveById( strId)
27 MsgBox(" Saved Id=" & strId & vbCrLf & "Retrieved Id=" & msgUserMan.Id)
28 End Sub

```

在程序清单 8.15 中，首先绑定到现有的私有队列 **UserMan**，然后建立了可接受 **String** 数据类型的格式符，并创建了三个消息并将它们发送到队列中去。在把第二个消息发送到队列后，将该消息的 **ID** 储存下来，随后使用该 **ID** 检索与之相关联的消息。该消息位于消息队列中的第二位。第 27 行代码只是简单地显示储存下来的消息 **ID** 以及检索到的消息，它们是相同的。

如果使用 **ReceiveById** 方法（实际上是 **PeekById** 方法），则必须注意队列中是否存在消息，若没有，则将抛出一个 **InvalidOperationException** 异常。所以不论什么时候，只要使用了这两种方法之一，就必须使用结构化错误处理程序，程序清单 8.16 说明的正是这一点。

程序清单 8.16 以安全的方式来根据消息 **ID** 检索消息

```

1 Public Function RetrieveMessageByIdSafe( ByVal vstrId As String) As Message
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 建立格式符
6     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
7         {GetType(String)})
8
9     Try
10        msgUserMan = queUserMan.ReceiveById(vstrId)
11        Catch objE As InvalidOperationException
12            ' 消息没有找到，返回空值
13            RetrieveMessageByIdSafe = Nothing
14
15            Exit Function
16        End Try
17
18    ' 返回消息
19    RetrieveMessageByIdSafe = msgUserMan
20 End Function

```

在程序清单 8.16 中，如果企图根据错误的 **ID** 来查找一个不存在的消息，那么该企图会

被捕获，而且仅会向调用方返回一个空值。如果在队列中可以找到相符的消息（其 ID 与传递过来的消息 ID 相同），就会向调用方返回检索到的消息。

消息 ID 的数据类型是 **String**，但是在内部它的类型是 **GUID**，就像在程序清单 8.15 中看到的那样。无论如何，消息的 **Id** 属性都是只读的，所以不能自己指定消息的 ID。这可能是为了确保消息惟一性所采取的措施之一。惟一的办法是保存某个特定消息的 ID，随后使用保存下来的 ID 来访问该消息。



注意 **PeekById** 方法与 **ReceiveById** 方法几乎完全相同，区别只在于前者不会从消息队列中删除消息。

8.4.9 异步发送消息和接收消息

有时需要以异步的方式来发送和接收消息，从而使应用程序不必等待消息发送和接收完毕就可以继续执行。为了在消息队列中使用异步通信，需要设定一个事件处理程序，由它来处理异步操作结果。添加处理程序的过程可以用 **AddHandler** 语句来完成。程序清单 8.17 中列出了有关示例。

程序清单 8.17 使用 **AddHandler** 来以异步方式接收消息

```

1 Public Sub MessageReceiveCompleteEvent(ByVal vobjSource As Object, _
2     ByVal vobjEventArgs As ReceiveCompletedEventArgs)
3     Dim msgUserMan As New Message()
4     Dim queUserMan As New MessageQueue()
5
6     ' 确保绑定到正确的消息队列
7     queUserMan = CType(vobjSource, MessageQueue)
8
9     ' 结束异步检索
10    msgUserMan = queUserMan.EndReceive(vobjEventArgs.AsyncResult)
11 End Sub
12
13 Public Sub RetrieveMessagesAsync()
14     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
15     Dim msgUserMan As New Message()
16
17     ' 建立格式符
18     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
19         {GetType(String)})
20
21     ' 添加事件处理器
22     AddHandler queUserMan.ReceiveCompleted, _
23         AddressOf MessageReceiveCompleteEvent
24
25     queUserMan.BeginReceive(New TimeSpan(0, 0, 10))
26 End Sub

```

在程序清单 8.17 中，首先建立一个过程用来接收 **Receive Complete** 事件。该过程负责绑定到传递给该过程的队列，而该队列是用来启动消息检索操作的，随后异步检索操作通过调用 **EndReceive** 方法而结束。该方法同时返回从队列中得到的消息。

在 **RetrieveMessagesAsync** 过程中建立了消息队列，在第 22 行添加了事件处理器，以便调用 **MessageReceiveCompleteEvent** 过程，完成消息检索。最后，通过第 25 行代码中的 **BeginReceive** 方法启动异步消息检索操作。其中已经指定，由于使用了 **TimeSpan** 类的一个新实例，调用会在 10 秒钟之后超时。

在程序清单 8.17 中使用了 **BeginReceive** 以及 **EndReceive** 方法。如果需从队列中抽取消息而不是接收消息，则只需对示例代码做少许改动（请参阅程序清单 8.18）。

程序清单 8.18 使用 **AddHandler** 异步抽取消息

```

1 Public Sub MessagePeekCompleteEvent(ByVal vobjSource As Object, _
2   ByVal vobjEventArgs As PeekCompletedEventArgs)
3     Dim msgUserMan As New Message()
4     Dim queUserMan As New MessageQueue()
5
6     ' 确保我们绑定到正确的消息队列
7     queUserMan = CType(vobjSource, MessageQueue)
8
9     ' 结束异步抽取
10    msgUserMan = queUserMan.EndPeek(vobjEventArgs.AsyncResult)
11 End Sub
12
13 Public Sub PeekMessagesAsync()
14     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
15     Dim msgUserMan As New Message()
16
17     ' 建立格式符
18     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
19       {GetType(String)})
20
21     ' 添加事件处理器
22     AddHandler queUserMan.PeekCompleted, _
23       AddressOf MessagePeekCompleteEvent
24
25     queUserMan.BeginPeek(New TimeSpan(0, 0, 10))
26 End Sub

```

8.4.10 从队列中清除消息

有两种方法可以从队列中删除消息：逐个删除消息或者一次清空整个队列。

删除队列中的单个消息

只能通过编程删除队列中的单个消息，服务器资源管理器或者 **Computer Management** 都无法完成。

可以用 **Receive** 方法从队列中删除消息。该方法一般是用来从队列中检索消息，但是它也可以用于删除消息。请参阅本章前面的“检索消息”以获得有关的详细信息。

也可以使用 **ReceiveById** 方法来删除消息。先用 **PeekById** 方法找出特定消息的 ID，而后再用 **ReceiveById** 方法删除该消息。请参阅本章前面的“从队列中选取特定消息”来获得关于使用 **ReceiveById** 和 **PeekById** 方法的详细信息。

删除队列中的所有消息

如果要删除队列中的所有消息，可以使用**分层数据库**或者 **Computer Management** 来手工完成，也可以通过编程来完成。



清除队列中所有消息的操作是不可撤销的，一旦确认删除，消息将会永远丢失。因此在进行此项操作时，一定要非常谨慎！

手工删除所有消息

可以用服务器资源管理器或者 **Computer Management** 来手工删除队列中的所有消息。选定 [Queue Messages] 节点（请参阅前面的图 8.3 和图 8.4），右键单击该节点，随后选择弹出菜单中的 [All Tasks/ Purge]（对 **Computer Management** 来说）或者 [Clear Messages]（对服务器资源管理器来说）。然后在提示确认的对话框中单击 [Yes] 或者 [OK] 以清除队列中的全部消息。

以编程方式删除所有消息

如果希望以编程方式清除队列中的所有消息，则可以用 **Purge** 方法来进行。程序清单 8.19 说明了如何绑定到消息队列并清除其中的全部消息。

程序清单 8.19 清除队列中的所有消息

```
1 Public Sub ClearMessageQueue()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As Message
4
5     ' 清除队列中的所有消息
6     queUserMan.Purge()
7 End Sub
```

在程序清单 8.19 中，用 **MessageQueue** 类中的 **Purge** 方法清除了队列中的所有消息。无论队列中有多少消息都可以使用该方法清空，也就是说即使调用该方法时队列是空的也没有问题。

8.4.11 定制消息优先级

特定的消息是以 **ASAP** 的方式读取的，这一点很重要。一般在发送消息时，它会被放到消息队列末尾，这是因为消息是根据到达的时间排序的。但是，在根据时间排序前消息要先

根据优先级排序，因此如果想确保发送的消息能够放到消息队列的头部，则应为该消息指定一个较高的优先级。

可以用 **Message** 类中的 **Priority** 方法设定消息的优先级。请参阅程序清单 8.20 中的示例代码，该代码向队列发送了两条消息，其中一条具有普通的优先级，而另一条则拥有最高优先级。

程序清单 8.20 发送拥有不同优先级的消息

```

1 Public Sub SendPriorityMessages()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 建立格式符
6     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
7         GetType(String), GetType(Integer))
8
9     ' 创建第一个主体
10    msgUserMan.Body = "First Message"
11    ' 向队列发送消息
12    queUserMan.Send(msgUserMan)
13
14    ' 创建第二个主体
15    msgUserMan.Body = "Second Message"
16    ' 将优先级设为最高
17    msgUserMan.Priority = MessagePriority.Highest
18    ' 向队列发送消息
19    queUserMan.Send(msgUserMan)
20 End Sub
21
22 Public Sub RetrievePriorityMessage()
23     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
24     Dim msgUserMan As Message
25
26     ' 建立格式符
27     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
28         GetType(String))
29
30     ' 检索队列中的首条消息
31     msgUserMan = queUserMan.Receive
32     ' 显示消息主体
33     MsgBox(msgUserMan.Body.ToString)
34 End Sub

```

运行程序清单 8.20 中的代码将会发现，如果把第二个消息的优先级设定为最高，就能够确保系统将该消息放置在队列头部，消息框中将会显示文本“Second Message”。实际上，只在消息队列不是事务型的情况下才会出现前面所述的情况。如果消息队列是事务型的，消息框中的文本将是“First Message”。

在第 17 行中设定了第二个消息的优先级。在设定 **Message** 对象的 **Priority** 属性时，必须将其设为 **MessagePriority** 所列举的一个成员，默认值为 **Normal**。

8.4.12 定位消息队列

有时并不知道某个特定消息队列所在的路径，或者想要检查某个队列是否还存在。我们先来解决其中较为容易的一个：检查某个特定队列是否存在（参阅程序清单 8.21）。

程序清单 8.21 检查某个消息队列是否存在

```
1 Public Function CheckQueueExists(ByVal vstrPath As String) As Boolean
2     CheckQueueExists = MessageQueue.Exists(vstrPath)
3 End Function
```

在程序清单 8.21 中，我们使用了 **MessageQueue** 类的 **Exists** 方法。由于这个方法是一个公有的共享函数，所以不必实例化消息队列对象就可以使用该方法。

CheckQueueExists 过程将会检查传递过来的参数是否与现存的某个消息队列相匹配，如果找到匹配项则返回 **True**，没找到则返回 **False**。

接下来的事情就很简单了，因为你已经得到了路径。然而，有时并不知道路径，知道的是安置该消息队列的机器名。请参阅程序清单 8.22 中的示例代码，该代码可以检索在特定机器上的私有队列的列表。

程序清单 8.22 检索某台机器上的所有私有队列

```
1 Public Sub BrowsePrivateQueues()
2     Dim arrquePrivate() As MessageQueue = _
3     MessageQueue.GetPrivateQueuesByMachine("USERMANPC")
4     Dim queUserMan As MessageQueue
5
6     ' 显示机器上所有私有队列的名称
7     For Each queUserMan In arrquePrivate
8         MsgBox(queUserMan.Label)
9     Next
10 End Sub
```

在程序清单 8.22 中，检索了位于名为 **USERMANPC** 机器上的所有消息队列，并显示了它们的标签。就像从第 2 行代码中可以看到的那样，全部队列都存储在一个 **MessageQueue** 类实例数组中。如果要检索的是公有队列，则只需用 **GetPublicQueuesByMachine** 方法替换 **GetPrivateQueuesByMachine** 方法即可。

我们不但可以在某一台机器上定位公有队列，还可以在整個网络上定位。有三种可用来定位网络上公有队列的方法。

- **GetPublicQueues**
- **GetPublicQueuesByCategory**
- **GetPublicQueuesByLabel**

请参阅程序清单 8.23、程序清单 8.24 和程序清单 8.25 的代码，它们说明了如何使用上述三种方法。

程序清单 8.23 检索某网络上的所有公有队列

```

1 Public Sub BrowsePublicQueuesNetworkWide()
2     Dim arrquePublic() As MessageQueue = _
3     MessageQueue.GetPublicQueues()
4     Dim queUserMan As MessageQueue
5
6     ' 显示网络上所有公有队列的名称
7     For Each queUserMan In arrquePublic
8         MsgBox(queUserMan.QueueName)
9     Next
10 End Sub

```

在程序清单 8.23 中使用 **GetPublicQueues** 方法来检索某网络上的所有公有队列，随后再显示每个队列的名称。

在程序清单 8.24 中使用 **GetPublicQueuesByCategory** 方法来排序检索某网络上所有的公有队列。对于每个返回的类别为“00000000 0000 0000 0000 000000000001”的消息队列，都显示该消息队列的名称。这里所指的排序与图 8.2 中的 **TypeID** 以及 **MessageQueue** 类的 **Category** 属性是一样的。这是一种用来对消息队列进行分组的方法，特别适于管理工作。

程序清单 8.24 排序检索某网络上的所有公有队列

```

1 Public Sub BrowsePublicQueuesByCategoryNetworkwide()
2     Dim arrquePublic() As MessageQueue = _
3     MessageQueue.GetPublicQueuesByCategory( _
4     New Guid("00000000 0000 0000 0000 000000000001"))
5     Dim queUserMan As MessageQueue
6
7     ' 显示所有公有队列的名称
8     ' 在专门类别的网络上
9     For Each queUserMan In arrquePublic
10        MsgBox(queUserMan.QueueName)
11    Next
12 End Sub

```

程序清单 8.25 中的代码说明了如何找到某网络上所有代有“userman”标签的公有队列，并逐个显示该消息队列所在的机器名称。

程序清单 8.25 检索某个网络上带有某标签的所有公有队列

```

1 Public Sub BrowsePublicQueuesByLabelNetworkWide()
2     Dim arrquePublic() As MessageQueue = _
3     MessageQueue.GetPublicQueuesByLabel("userman")
4     Dim queUserMan As MessageQueue
5
6     ' 显示所有公有队列的名称
7     ' 在带有特殊标签的网络上
8     For Each queUserMan In arrquePublic

```



```
9   MsgBox(queUserMan.MachineName)
10  Next
11 End Sub
```

8.4.13 删除消息队列

有多种办法可删除机器上的消息队列。既可以通过服务器资源管理器或 Computer Management 手工删除消息队列，也可以通过编程来删除。



删除消息队列的操作是不可撤销的，一旦确认删除，该队列以及其中的所有消息就会永远丢失！

手工删除消息队列

如果要手工删除消息队列，可以使用服务器资源管理器或者 Computer Management。先选定该队列对应的节点（请参阅图 8.1 和图 8.5），然后再用右键单击该节点，并选择弹出菜单中的 [Delete]，最后单击提示确认对话框中的 [Yes]。这样就把该消息队列以及其中的所有消息一并删除了。

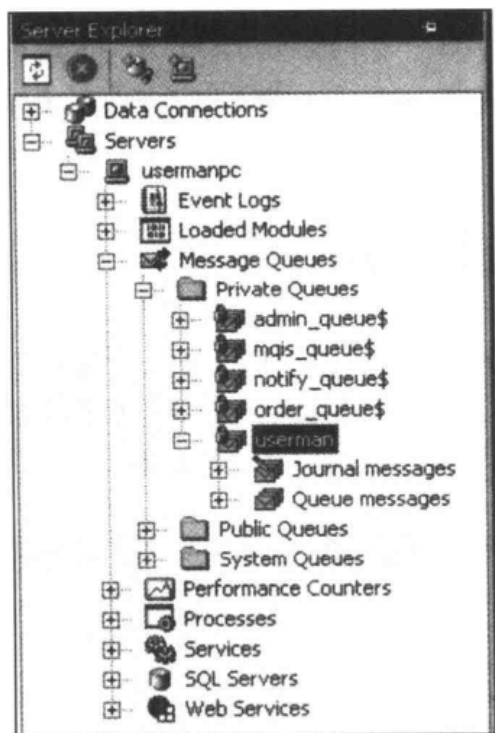


图 8.5 选定消息队列之后的服务器资源管理器

通过编程来删除消息队列

如果要使用编程方式来清空队列中的所有消息，可以使用 **Purge** 方法。程序清单 8.26 中的代码说明了如何绑定到消息队列并清空其中的所有消息。

程序清单 8.26 删除消息队列

```
1 Public Sub RemoveMessageQueue(ByVal vstrPath As String)
2     MessageQueue.Delete(vstrPath)
3 End Sub
```

程序清单 8.26 的代码中使用了 **MessageQueue** 类的 **Delete** 方法来删除路径位于 **vstrPath** 的队列。要牢记，如果该队列不存在，就会抛出一个异常，因此不推荐使用程序清单 8.26 中的示例代码，除非确认要删除的队列确实存在。请参阅本章后面的“定位消息队列”一节以获取关于使用 **Exists** 方法的详细信息。另外，也可以像程序清单 8.27 中的代码那样建立一个结构化错误处理程序。

程序清单 8.27 以安全方式删除消息队列

```
1 Public Sub RemoveMessageQueueSafely(ByVal vstrPath As String)
2     Try
3         MessageQueue.Delete(vstrPath)
4     Catch objE As Exception
5         MsgBox(objE.Message)
6     End Try
7 End Sub
```

在程序清单 8.27 中，尝试删除的消息队列的路径由 **vstrPath** 参数指定，而代码则用来捕获删除队列时可能抛出的异常。示例中只显示了出错消息，你可以用自己的错误处理操作来替代它。

8.5 创建事务型消息队列

对于一般的数据库访问来说，可以将消息队列转变为事务型的，这意味着可以将若干消息作为单个事务处理一同发送，随后再根据这些消息的传输结果来决定是提交还是回滚这些消息。消息队列是否是事务型的是在队列创建时就决定的，一旦队列创建完成，就无法再进行更改。

对于消息队列来说存在两种类型的事务处理：内部的和外部的。下面的几节对这两种类型的事务处理做了简要介绍。

8.5.1 内部事务处理

内部事务处理是指手工或者显式管理的事务处理，它只与在客户端和消息队列之间传递消息相关。也就是说，内部事务处理并不把其他诸如数据库操纵之类的资源包括为自身的一部分，并且必须显式地启动事务处理、提交事务处理或者回滚事务处理。内部事务处理是由 **Message Queuing** 的 **Transaction Coordinator** 来处理的。

内部事务处理比外部事务处理执行更快，随后我们将解释这一点。

8.5.2 外部事务处理

顾名思义，外部事务处理肯定与外部有关。除了消息队列资源外，它还使用其他资源作为自身的一部分，这些资源包括数据库访问、Active Directory 访问，等等。更进一步说，外部事务处理不是由 Message Queuing 的 Transaction Coordinator 来处理的，而是由 Microsoft Distributed Transaction Coordinator (MS DTC) 之类的协调器来处理的。

外部事务处理比内部事务处理执行得慢。本章只涉及了内部事务处理，因为外部事务处理已经超出了本书的范围，它可以单独作为一本书的主题。

8.5.3 创建事务型消息队列

如果要通过编程创建消息队列，请参阅程序清单 8.4 中的示例代码，它说明了如何生成事务型消息队列。如果想要使用服务器资源管理器来创建事务型消息队列，则可以在第 4 章中找到有关的详细信息。当然也可以使用 Computer Management MMC 插件。打开该插件，并且执行以下操作：

1. 展开 [Services and Applications] 节点。
2. 展开 [Message Queuing] 节点。
3. 选定 [Public Queues] 节点或者 [Private Queues] 节点。
4. 右键单击该节点，选择弹出菜单中的 [New/ Public Queue] 或者 [New/ Private Queue] 选项。系统将弹出 [Queue Name] 对话框，如图 8.6 所示。

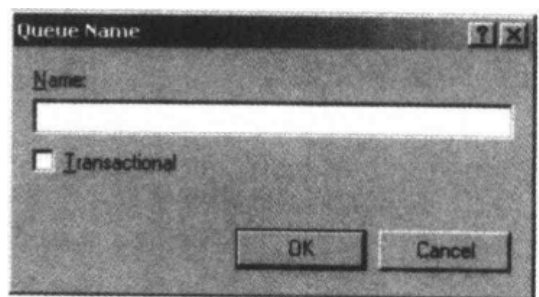


图 8.6 [Queue Name] 对话框

随后给消息队列取一个名字，输入到 [Name] 文本框中去，并且确认勾选了 [Transactional] 复选框，最后单击 [OK] 以完成消息队列的创建过程。

8.5.4 启动事务处理

由于内部事务处理是显式的或者说是手工的，所以必须先要在代码中启动事务处理，然后才可以开始发送和接收消息，这些消息属于事务处理的一部分。事实上，首先要做的是检查消息队列是否是事务型的，通过检查其 **Transactional** 属性可以做到这点。该属性是个 **Boolean** 值，如果为 **TRUE** 就说明队列是事务型的，反之则不是。如下所示：

```
If queUserMan.Transactional Then
```

随后必须创建 **MessageQueueTransaction** 类的实例。由于该类的析构函数并未重载，所以创建实例的代码应该如下所示：

```
Dim qtrUserMan As New MessageQueueTransaction()
```

接下来，通过使用 **MessageQueueTransaction** 类中的 **Begin** 方法启动事务处理，如下所

示:

```
qtrUserMan.Begin()
```

可能令你感到疑惑的是, 应该如何从消息队列对象中引用事务处理对象。你的困惑可以理解, 但是整个过程实际上并不深奥。只要在发送或检索消息时传递该事务处理对象就可以了, 如下所示:

```
msgUserMan = queUserMan.Receive(qtrUserMan)
```

或是

```
queUserMan.Send(msgUserMan, qtrUserMan)
```

从这些短小的示例中可以看出, 它们与通常不使用事务处理的代码并没有什么不同, 只是在需要执行属于事务处理部分的操作时必须传递事务处理对象。这样做与直接将事务处理与消息队列关联起来的做法相比有下列几个优点: 你可以决定将哪些操作要作为事务处理的一部分, 因此可以将某些消息作为事务处理的一部分加以发送和接收, 而另外一些消息则不作为事务处理的一部分来发送和接收。此外, 还可以将事务处理用在多个消息队列上。

8.5.5 终止事务处理

在启动事务处理之后, 必须在适当时候终止它。如果要问为什么, 只能说这是理所当然的。然而, 应该如何终止事务处理并不是那么显而易见。如果在执行事务处理部分的操作时没有遇到问题, 将会顺利地把事务处理提交给消息队列, 或将改动应用到消息队列中。如果在执行事务处理操作时遇到问题, 则要在发生问题之后尽快中断该事务处理。这里所说的问題不仅仅指 **MessageQueue** 对象抛出的异常, 还包括消息队列操作之外可能导致事务处理中断的其他操作。

提交事务处理

提交事务处理的操作十分简明, 可以通过 **MessageQueue** 类的 **Commit** 方法来实现, 如下所示:

```
qtrUserMan.Commit()
```

Commit 方法没有被重载, 也没有任何参数。然而, 如果试图提交一个未启动 (未使用 **Begin** 方法启动) 的事务处理, 就将导致 **MessageQueueException** 异常的抛出。

中断事务处理

如果某种情况下不得不中断事务处理, 则可以使用 **MessageQueue** 类中的 **Abort** 方法来实现, 如下所示:

```
qtrUserMan.Abort()
```

Abort 方法未被重载, 也没有任何参数。然而, 如果试图中断一个未启动 (未使用 **Begin** 方法启动) 的事务处理, 就将导致抛出 **MessageQueueException** 异常, 这点与 **Commit** 方法

相类似。

8.5.6 使用 MessageQueueTransaction 类

前面已经讨论了与事务管理有关的所有重要方法，但是有一个属性还没有提到过，那就是 **Status** 属性。该属性是只读的，且返回一个 **MessageQueueTransactionStatus** 枚举值，该枚举值指出了事务处理的状态。从创建 **MessageQueueTransaction** 类实例开始到该实例对象被销毁的整个过程中，都可以读取 **Status** 属性的值。请参阅表 8.2 中的 **MessageQueueTransactionStatus** 枚举成员值。

表 8.2 MessageQueueTransactionStatus 枚举成员值

成员名	说明
<i>Aborted</i>	事务处理已中断。这可能是用户调用 Abort 方法导致的结果
<i>Committed</i>	事务处理已提交。这可能是使用 Commit 方法导致的结果
<i>Initialized</i>	事务处理对象已被实例化，但是事务处理尚未启动。在这种状态下，不能将该事务处理对象传递给消息队列方法
<i>Pending</i>	事务处理已启动，正在进行 Abort 或者 Commit 调用。在事务处理挂起时，可以将该事务处理对象传递给消息队列方法

程序清单 8.28 中的示例代码说明了如何将事务处理与结构化错误处理程序配合使用。必须先在本机上创建一个名为 **UserMan** 的事务型私有消息队列，然后才能运行该示例代码。

程序清单 8.28 使用消息队列事务处理

```

1 Public Sub UseMQTransactions()
2     Dim qtrUserMan As New MessageQueueTransaction()
3     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
4     Dim msgUserMan As New Message()
5
6     ' 建立队列格式符
7     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
8         GetType(String))
9
10    ' Clear the message queue
11    queUserMan.Purge()
12    ' 启动事物处理
13    qtrUserMan.Begin()
14
15    Try
16        ' 创建消息主体
17        msgUserMan.Body = "First Message"
18        ' 向队列发送消息

```

```

19      queUserMan. Send(msgUserMan, qtrUserMan)
20
21      ' 创建消息主体
22      msgUserMan. Body = "Second Message"
23      ' 向队列发送消息
24      queUserMan. Send( msgUserMan)
25
26      ' Retrieve message from queue
27      msgUserMan = queUserMan.Receive()
28      ' 显示消息主体
29      MsgBox( msgUserMan.Body)
30
31      ' 提交事务处理
32      qtrUserMan.Commit()
33
34      ' 检索队列中的消息
35      msgUserMan = queUserMan.Receive()
36      ' 显示消息主体
37      MsgBox( msgUserMan.Body)
38      Catch objE As Exception
39          ' 异常中断事务处理
40          qtrUserMan.Abort()
41      End Try
42 End Sub

```

在程序清单 8.28 中，我们说明了如何将事务型消息和非事务型消息发送到同一个队列。在运行代码时，将会看到先显示“Second Message”文本，然后再显示“First Message”文本。这是因为当第 27 行代码从队列中检索到第一条消息时，该消息尚未提交给队列。稍后的代码才会提交事务处理，并把第一条消息发送到队列。

8.6 由系统生成的队列

到此为止，我们讨论的都是由用户创建的队列，但是还有一类队列需要加以重视，那就是由系统生成的队列。这些队列由 **Message Queuing** 加以维护。以下列出的队列都是系统队列：

- 死信消息
- 日志消息
- 事务型死信消息

在图 8.7 中的服务器资源管理器的 [System Queues] 节点能够看到上述的两种死信消息队列，它们专门用来存储无法传递的消息。其中一个队列用于存储非事务型消息，另一个则用于存储事务型消息。在下一节我们将对日志消息队列进行说明。

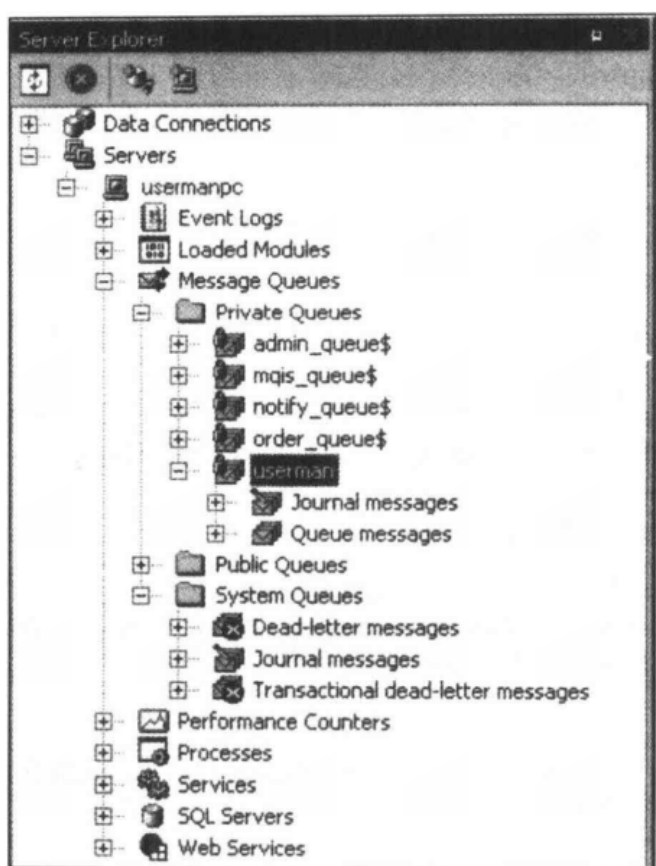


图 8.7 系统队列以及日志队列

8.6.1 使用日志存储

日志存储是一种强大的工具，它可以用来保存发送到队列的消息或者从队列中删除的消息副本。如果由于某种原因必须重发某条消息，日志存储会非常有用。消息传递可能会失败，在这种情况下将会收到一个表示拒绝的确认消息。由于确认消息并不包括原消息的主体，因此无法利用它来重新发送原始消息，但是可以利用它从日志消息队列中找到原始消息的副本。

每个消息队列客户端（机器）都拥有一个名为系统日志的日志消息队列。系统日志负责存储从该消息队列客户端发送的所有消息，而且不论消息发送是否成功都对其进行存储。

除了系统日志以外，所有消息队列都拥有自身的日志队列。而每个消息队列都有与之相关联的日志队列，用它来存储从该消息队列中删除的所有消息，但只在该消息队列的日志功能已启用的情况下才会进行这种日志存储。如果了解关于如何启用与消息队列相关的日志队列的详细信息，请参阅“启用消息队列日志”一节。

最后一种使用日志存储的方法是通过编程来启用针对每条消息的日志，启用该日志之后，从系统中发送的所有消息的副本都会被储存在本机的系统日志中。

启用消息队列日志

可以手工启用消息队列日志，也可以通过编程来启用。如果需要手工进行，则可以通过 Computer Management MMC 插件完成。在图 8.2 中可以看到，某现有队列或将要创建的队列

的 [Properties] 对话框。如果想要启用日志，则必须勾选 [Journal group] 中的 [Enabled] 复选框，然后单击 [Apply]。

也可以通过编程来启用针对现有队列的日志，只要把现有消息队列的 **UseJournalQueue** 属性设为 **True** 就可以了，如下所示：

```
Dim queUserMan As New MessageQueue(".\Private$\UserMan")
queUserMan.UseJournalQueue = True
```

启用针对每条消息的日志

可以通过编程来使用针对每条消息的日志，先把现有消息的 **UseJournalQueue** 属性设置 **True**，随后再将其发送到队列中，如程序清单 8.29 中的代码所示。

程序清单 8.29 启用消息日志

```
1 Public Sub EnableMessageJournaling()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 建立格式符
6     queUserMan.Formatter = New XmlMessageFormatter(New Type() _
7         {GetType( String)})
8
9     ' 创建消息主体
10    msgUserMan.Body = "Message"
11    ' 启用消息日志
12    msgUserMan.UseJournalQueue = True
13    ' 向队列发送消息
14    queUserMan.Send(msgUserMan)
15 End Sub
```

在程序清单 8.29 的代码中，第 12 行代码启用了消息日志，随后的代码将会把消息发送到队列，此时系统会把该消息的副本储存在本机的系统日志中，实际上是一直等到消息发送到队列之后才进行存储的（第 14 行代码）。

从日志存储中检索消息

现在，如何启用日志这个问题已经很清楚了，但是究竟如何在需要的时候从日志中检索消息呢？可以通过指定队列所在的正确路径来访问日志队列。比如说，下面的代码就实现了对日志存储的访问，该日志对应的是本机上名为 **UserMan** 的私有队列。

```
Dim queUserMan As New MessageQueue(".\Private$\UserMan\Journal$")
```

8.7 保护消息队列的安全

到目前为止，本章已经介绍了如何创建私有队列和公有队列、事务型队列和非事务型队

列、如何向队列发送消息以及从队列中检索消息，以及如何从队列中抽取消息而不将消息从队列中删除。然而，还没有涉及消息队列安全，本节我们将涉及这个话题。

消息队列使用 Windows 操作系统的内建特性来保护消息安全。下面列出了这些特性：

- 验证
- 加密
- 访问控制
- 审核

8.7.1 使用验证

验证 (authentication) 是一种进程，该进程可以确保消息的完整性，以及检验消息的发送方。通过将消息队列的 **Authenticate** 属性设为 **True** 可以实现这点。该属性的默认值是 **False**，这意味着不需要验证。如果将该属性设为 **True**，队列就只接受通过验证的消息，而未通过验证的消息将被拒收。也就是说，是服务器上的消息队列要求消息通过验证，而不仅仅是设置了 **Authenticate** 属性的消息队列对象要求这样做。这意味着对该属性的设置会影响同一个消息队列上所有其他的消息队列对象。可以通过编程手段来启用验证，程序清单 8.30 中的代码就是此类示例。

程序清单 8.30 启用消息队列验证

```
1 Public Sub EnableQueueAuthentication()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 启用队列验证
6     queUserMan.Authenticate = True
7 End Sub
```

也可以用 Computer Management MMC 插件来设置这项属性，以达到要求消息队列实行验证的目的。可以参阅图 8.2 以及“显示或更改消息队列的属性”一节，得到关于如何设置现有消息队列属性的详细信息。在图 8.2 中可以看到 [Authenticated] 复选框，必须勾选该复选框，随后单击 [OK]，这样系统就会要求该队列上的所有消息都必须通过验证。

如果某个消息未获验证，它将会被拒收并被丢弃。然而，可以指定将拒收的消息放置在死信队列中，如程序清单 8.31 所示。

程序清单 8.31 拒收未通过验证的消息

```
1 Public Sub RejectNonauthenticatedMessage()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 启用队列验证
6     queUserMan.Authenticate = True
7     ' 建立队列格式符
8     queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
```

```

9
10 ' 创建消息主体
11 msgUserMan.Body = "Message Body"
12 ' 确保拒收消息被放在死信队列中
13 msgUserMan.UseDeadLetterQueue = True
14
15 ' 向队列发送消息
16 queUserMan.Send( msgUserMan)
17 End Sub

```

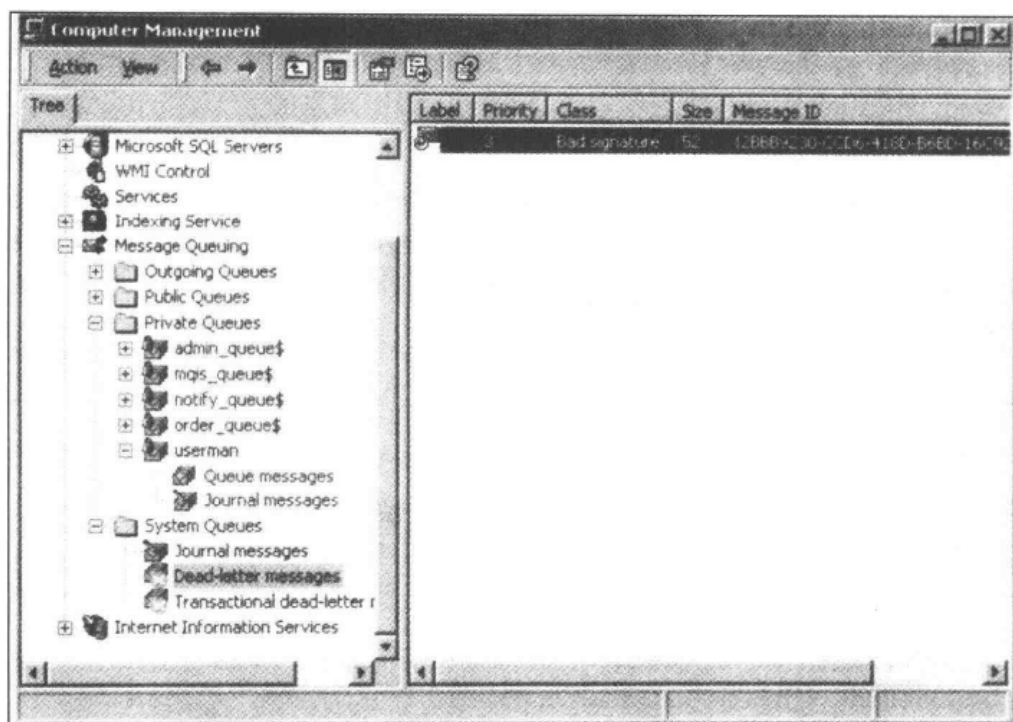


图 8.8 位于死信队列中的某个未通过验证的拒收消息

在程序清单 8.31 的代码中，将 **UseDeadLetterQueue** 属性设为 **True** 意味着如果有消息被拒收，系统就会将它放置在死信队列中（参阅图 8.8）。

从图 8.8 中可以看到，死信队列中的消息是由于不合适的签名而遭到拒收的。另一种处理拒收消息的方法，是要求在拒收消息时发出通知。要实现这些操作必须对 **AcknowledgeType** 属性和 **AdministrationQueue** 属性进行设置（请参阅程序清单 8.32）。

程序清单 8.32 在管理队列中接收拒收通知

```

1 Public Sub PlaceNonauthenticatedMessageInAdminQueue()
2     Dim queUserManAdmin As New MessageQueue(".\Private$\UserManAdmin")
3     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
4     Dim msgUserMan As New Message()
5
6     ' 启用队列验证
7     queUserMan.Authenticate = True
8     ' 建立队列格式符

```

```

9     queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
10
11     ' 创建消息主体
12     msgUserMan.Body = "Message Body"
13     ' 确保拒收消息被放在管理队列中
14     msgUserMan.AdministrationQueue = queUserManAdmin
15     ' 拒收消息的类型
16     msgUserMan.AcknowledgeType = AcknowledgeTypes.NotAcknowledgeReachQueue
17
18     ' 向队列发送消息
19     queUserMan.Send(msgUserMan)
20 End Sub

```

程序清单 8.32 中的代码建立了一个名为 **UserManAdmin** 的私有队列来接收拒收通知。请注意，在运行上面的示例代码之前，必须先创建该私有队列，并将其属性设为非事务型。

就像程序清单 8.31 中的代码一样，程序清单 8.32 中的代码将拒收你发出的消息。这看起来是合理的，因为是你要求所有消息都必须通过验证，而恰恰是你自己“忘记”了为消息加上验证用标识。

将 **AttachSenderId** 属性设为 **True** 可以为消息添加验证标识，因为这样的设置会让 **Message Queuing** 对 **SenderId** 属性进行设定。然而，仅仅这样做是不够的，还必须将 **UseAuthentication** 属性设为 **True**，这样才能确保在消息发送给消息队列之前为其加上了数字签名。同样，在收到消息时，由 **Message Queuing** 为该消息加上的数字签名会用来进行验证。程序清单 8.33 说明了如何发送加上验证标识的信息。

程序清单 8.33 发送加上验证标识的消息

```

1 Public Sub AcceptAuthenticatedMessage()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 允许队列验证
6     queUserMan.Authenticate = True
7     ' 建立队列格式符
8     queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
9
10    ' 确保拒收的信息放置在死信队列中
11    msgUserMan.UseDeadLetterQueue = True
12    ' 确保消息队列附属发送者 id，并且
13    ' 在发送前加上数字签名
14    msgUserMan.UseAuthentication = True
15    msgUserMan.AttachSenderId = True
16    ' 创建消息主体
17    msgUserMan.Body = "Message Body"
18
19    ' 向队列发送消息
20    queUserMan.Send(msgUserMan)

```

21 End Sub

在程序清单 8.33 中, 将 **Message** 对象的 **UseAuthentication** 属性以及 **AttachSenderId** 属性都设置为 **True**, 这样可以确保 **Message Queuing** 为该消息加上了数字签名, 并且确保该数字签名可以通过 **Message Queuing** 的验证。至此, 虽然消息队列仍然只能接受通过验证的消息, 但我们发出的消息不会再像程序清单 8.31 以及程序清单 8.32 中的示例那样被消息队列拒收。如果已经设置了消息队列的验证措施, 则可以省略第 5 行和第 6 行代码。

对每条消息是否经过验证都进行判定当然很好, 但实际上不必这么做。如果设置消息队列的验证措施, 则队列中的所有消息在到达队列之前就都已经通过验证了, 这样就可以信任从队列中接收到的任何消息。

8.7.2 使用加密措施

加密 (encryption) 是另外一种保护消息安全的方法, 可以用来保护在计算机之间使用消息队列传递的消息。通过对消息加密, 任何试图监视使用消息队列的计算机之间的网络通信的人将只能接收到加密过的消息。或许某些人能够解密这些消息, 但使用加密手段还是会增加窃取敏感信息的难度。

虽然在发送端要对发送消息进行加密以及在接收端对收到的消息进行解密, 会带来一定的开销。但是如果网络是公用的且需要移植敏感信息, 则还是应该考虑对消息进行加密。

就像验证一样, 加密要求可以对整个队列生效, 也就是说所有未加密的消息都会被队列拒收, 将 **MessageQueue** 对象的 **EncryptionRequired** 属性设置为 **EncryptionRequired.Body** 可以实现这点, 如程序清单 8.34 所示。

程序清单 8.34 确保队列拒收未加密消息

```
1 Public Sub EnableRequireBodyEncryption()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 允许主体加密要求
6     queUserMan.EncryptionRequired = EncryptionRequired.Body
7 End Sub
```

在程序清单 8.34 的代码中, 将 **MessageQueue** 对象的 **EncryptionRequired** 属性设置为 **EncryptionRequired.Body**。这样做的结果是, 发送到队列的任何消息主体都必须经过加密, 否则就会被拒收。消息队列加密也称为私密(privacy), 指的是私密消息(private message), 也就是经过加密的消息, 不要将它同私有消息队列相混淆。除了通过编程手段来设置消息队列的加密属性之外, 也可以在 **Computer Management MMC** 插件中进行设置 (请参阅图 8.9)。

在图 8.9 中, 从 [Privacy level] 下拉菜单中选定 [Body], 这与运行程序清单 8.34 中代码的效果相同。请注意, 在本例中未选取 [Authentication] 复选框, 这并不是说加密和验证不能同时使用, 只是为了简化下面几个程序清单中的代码。

与验证不同, 即使消息队列并没有强制要求进行加密, 也可以使用加密, 但是只有在私密等级设为 **Optional** 的情况下才能这样做。这可以在 **Computer Management MMC** 插件中设

定，也可以通过编程手段实现，比如下面的代码就可以：

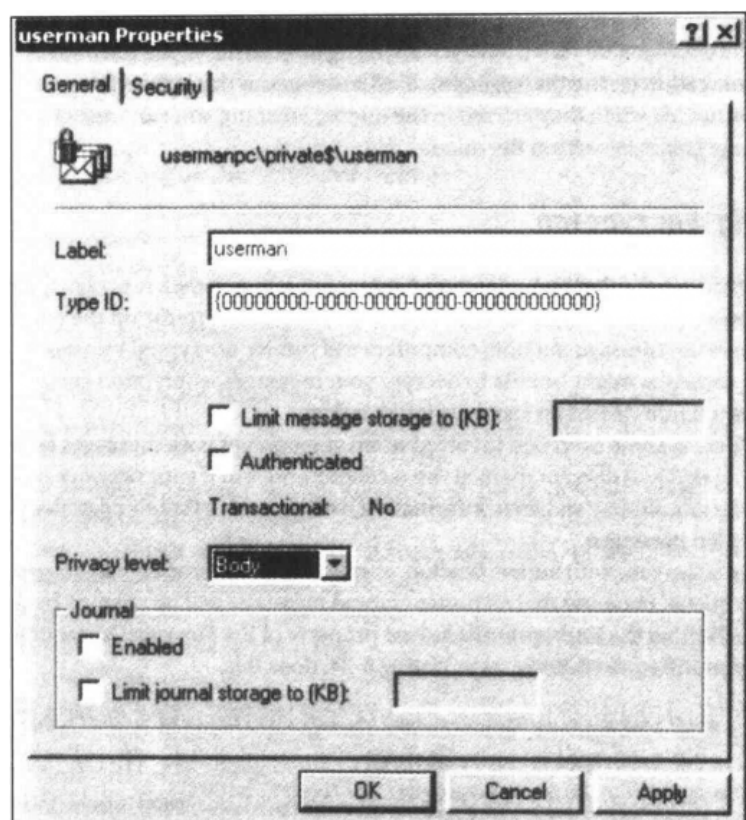


图 8.9 Body 的私密等级

```
Set message body encryption to optional
queUserMan.EncryptionRequired = EncryptionRequired.Optional
```

如果将消息队列的私密等级指定为 **Optional**，那么无论消息是否经过加密都可以发送给该消息队列。然而，如果将私密等级设为 **None**，则只能向队列发送未经加密的消息。如果向私密等级为 **None** 的队列发送加密过的消息，它将拒收该消息。

程序清单 8.35 中的代码说明了如何发送和接收加密消息。

程序清单 8.35 发送和接收加密消息

```
1 Public Sub SendAndReceiveEncryptedMessage()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4
5     ' 要求消息主体加密
6     queUserMan.EncryptionRequired = EncryptionRequired.Body
7     ' 建立队列格式符
8     queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
9
10    ' 确定消息在发送之前被加密
11    msgUserMan.UseEncryption = True
12    ' Create message body
```

```

13 msgUserMan.Body = "Message Body"
14
15 ' 将消息发送到队列
16 queUserMan.Send(msgUserMan)
17
18 ' 从队列检索消息
19 msgUserMan = queUserMan.Receive()
20 ' 显示解密消息主体
21 MsgBox(msgUserMan.Body.ToString)
22 End Sub

```

程序清单 8.35 中的代码对消息进行了加密，并将加密后的消息发送给队列，随后又从队列检索该消息。在消息传输的过程中，无论消息是输入消息队列还是由消息队列输出，消息体都是加密的，但由负责进行接收的 **Message Queue** 对象自动对消息进行解密。这就是说，对加密过的消息进行自动解密的工作总是在接收消息的时候进行。

8.7.3 使用访问控制

对消息队列进行访问控制也许是保护消息安全的最佳方案。就像大多数 Windows 操作（比如创建新用户）一样，对消息的读写也可以进行访问控制。在用户尝试执行某项操作时，比如从消息队列中读取消息，访问控制将会生效。Windows 2000 或者 Windows NT 下的每位用户都对应有一份 ACL（Access Control List，访问控制列表），其中包含了允许该用户执行的所有操作。在用户尝试读取消息的时候，访问控制就对该用户的 ACL 进行检查，如果该用户拥有读权限，他就可以成功地从队列中读出数据。而如果不允许用户从队列中读消息，读操作就将被禁止。

访问控制可以应用到消息队列级别上或者消息级别上，也能应用到 **Message Queuing** 级别上。如果是后者，那么 Active Directory 中的所有消息队列都必须遵守设定的许可权。图 8.10 展示了用户 User Man 在 UserMan 域上的 ACL，列出的许可权限对应于 UserMan 私有消息队列。

启动 Computer Management MMC 插件，选定名为 UserMan 的私有队列，右键单击该队列，随后选择弹出菜单中的 [Properties]，系统将会显示 [Properties] 对话框，在该对话框中单击 [Security] 就会看到 ACL（请参见图 8.10）。

就像在图 8.10 中看到的一样，消息队列的所有操作，比如写、读、抽取消息等，几乎都有相应的许可权与其对应。在每项操作后面都有两个复选框，选定其中一个将允许此项操作，选定另一个则意味着拒绝此项操作。如果你希望设定某些用户或组的许可权限，只要在对对话框顶部的列表框中进行选取即可。如果想设置的用户或组未列出，可以单击 [Add] 按钮，然后将它们添加到列表框中。如果想要删除某些用户或组，则只需单击 [Delete] 按钮。

在设定组的许可权限时需要特别小心，因为这里的 ACL 与 Windows 的 ACL 的惯例一样的，限定性越高的许可权限优先级越高。举个例子来说，如果允许用户 User Man 拥有删除 (Delete) 操作许可权限，随后又将 Everyone 组的同一项权限设为拒绝 (deny)，用户 UserMan

将不具有删除消息队列的权限。域中的所有用户都是 **Everyone** 组的一部分，所以在设定该组的权限时要特别小心。设定组 and 用户许可权限方面的内容已经超出了本书的范围，如果想要了解关于 **ACL** 的详细信息以及为用户和组设定许可权限的详细资料，请参阅 **Windows** 操作系统附带的文档。

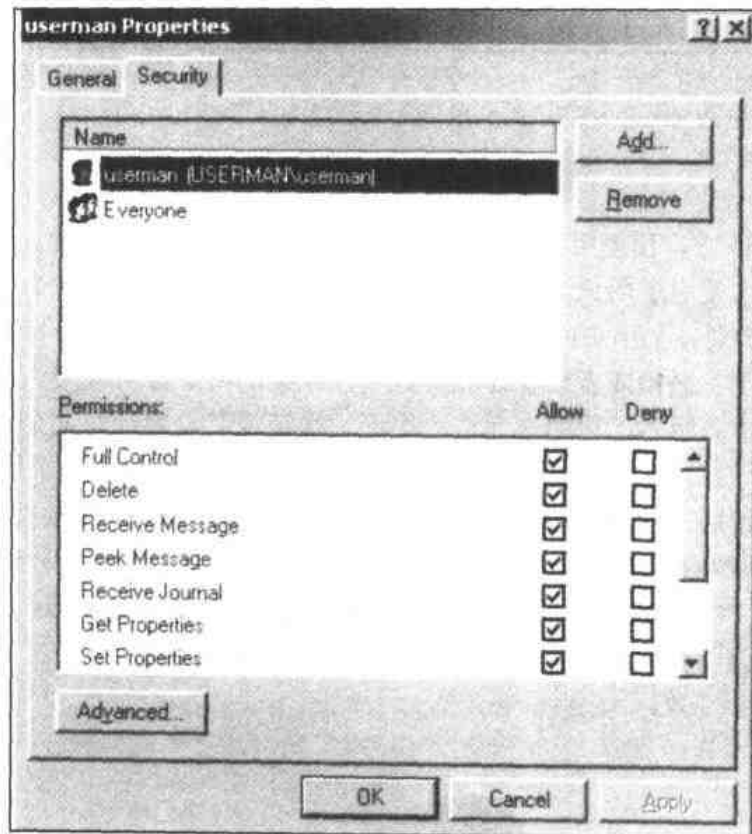


图 8.10 选定 [Security] 选项卡后的 [user Properties] 对话框

使用 SetPermissions 方法

除了在 **Computer Management** 中设定用户许可权限之外，也可以通过编程手段来实现。这需要使用 **MessageQueue** 对象的 **SetPermissions** 方法，程序清单 8.36 中的代码就是具体的说明。

程序清单 8.36 通过编程手段设定用户许可权限

```

1 Public Sub SetUserPermissions()
2     Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3     Dim msgUserMan As New Message()
4     Dim aclUserMan As New AccessControlList()
5
6     ' 赋予 UserMan 用户对 UserMan 私有队列的完全控制权
7     queUserMan.SetPermissions(" userman", MessageQueueAccessRights.FullControl)
8     ' 赋予 UserMan 用户对 UserMan 私有队列的完全控制权
9     queUserMan.SetPermissions(New MessageQueueAccessControlEntry( _

```

```

10     New Trustee("userman"), MessageQueueAccessRights.FullControl))
11     ' 拒绝 UserMan 删除 UserMan 私有队列
12     queUserMan. SetPermissions("userman", MessageQueueAccessRights.DeleteQueue, _
13     AccessControlEntryType.Deny)
14     ' 拒绝 UserMan 对 UserMan 私有队列的全部访问权限
15     aclUserMan.Add( New AccessControlEntry(New Trustee("userman"), _
16     GenericAccessRights.All, StandardAccessRights.All, AccessControlEntryType.Deny))
17     queUserMan.SetPermissions(aclUserMan)
18 End Sub

```

在程序清单 8.36 中，我使用了 **SetPermissions** 方法的四种重载版本。头两种（第 7 行和第 9 行）所用的版本使用了用户名或组名以及赋予该用户或组的权限作为参数。这两种方法基本相同，只能用于向一个用户或组指派权限，而不能撤销权限。第 12 行代码中的重载版本可以用来允许、拒绝、撤销或者设置针对特定用户或组的特殊许可权限。而第 15 到 17 行代码使用的最后一种重载版本，可用来允许、拒绝、撤销或者设置针对某个用户或组的通用访问权限以及标准访问权限。实际上，后面两种重载版本可以用来更改多个用户和/或组的许可权限，做法是：在使用 ACL 设定许可权限之前（第 17 行代码），先要按照需要向 ACL 添加多个访问控制项（Access Control Entries, ACE）（第 15 行代码）。

程序清单 8.36 中的示例代码只是用来说明如何使用 **SetPermissions** 方法的四种重载版本，但实际上不应该同时使用它们。如果在指派或撤销的权限之间发生冲突，则最后所做的指派或者撤销操作会生效。

8.7.4 使用审核

从本章的角度来说，审核（Auditing）用来记录对消息队列的访问。也就是说在启用该功能后，就可以通过查看事件日志（Event Log）以确定有哪些用户访问过消息队列，甚至哪些人试图访问消息队列而被拒绝。审核超出了本书的内容，它是一种通用工具，用于记录 Windows 操作系统中任何服务或对象所受到的访问。

可以在 Computer Management MMC 插件中建立审核。选定以消息队列的名称命名的节点，以此选定希望进行审核的消息队列（实际上是希望审核对应于该消息队列的所有消息队列）。右键单击选定的节点，选择弹出菜单中的 [Properties] 选项。系统将显示 [Properties] 对话框，先从中选择 [Security] 选项卡，随后单击 [Advanced] 按钮，系统将会显示 [Access Control Settings] 对话框，选择 [Auditing] 选项卡，将会看到如图 8.11 所示的对话框。一般说来列表中并没有审计项，但是图 8.11 中加入了一项，当用户（属于 Everyone 组）向消息队列中写入消息的操作失败时将被其记录。

有两种类型的审核：成功审核以及失败审核。执行消息队列操作来使审核生效，同时察看事件日志（应用程序日志）中的新增项。可以用事件查看器（Event Viewer）来查看事件日志。

如果想要进一步了解关于审核的详细信息，或者想要了解一般意义上的 Message Queuing security 安全，请阅读 Windows 2000 文档。

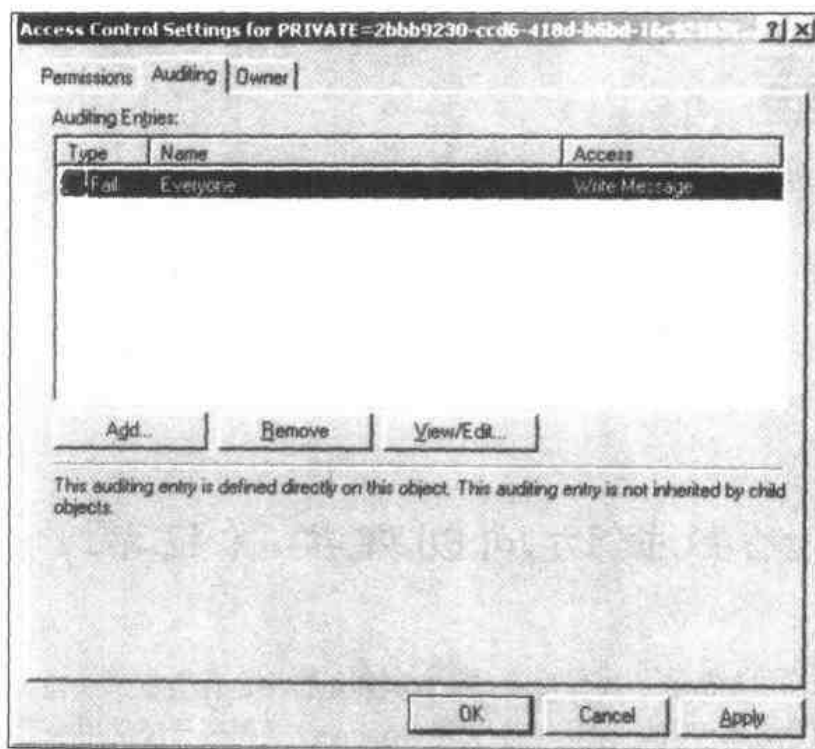


图 8.11 [Access Control Settings] 对话框

8.8 小结

本章引入了使用消息队列进行无连接编程的概念。说明了在服务器资源管理器中或者 Computer Management MMC 插件中创建消息队列的过程, 以及通过编程手段来创建消息队列的方法。

此外还讨论了如何在网络上定位消息队列, 消息队列如何与事务处理相配合, 为什么应该使用消息队列来替代数据库表, 消息在消息队列中是如何排序的, 以及如何使用 **MessageQueue** 类以及 **Message** 类中繁多的属性和方法。

下一章我们将讨论如何将数据访问功能包装到类中, 其中将涉及 OOP 方面的基本内容 (比如多态), 并讨论如何将这些内容应用到 UserMan 应用程序中。

第 9 章

数 据 包 装

为数据访问创建类（包装）

本章将讲述如何把数据访问打包成封装到类和组件中。你会看到一些代码示例，它们涵盖了 OOP（面向对象编程）的一些原则。包装（wrapper）是程序员经常用来形容术语类（class）的一个词，虽然后者更为准确一些。但在本书中，外壳就是指类。本章包含了一些将在第 11 章中用来完成 UserMan 应用程序示例的重要信息。此外，你还会遇到一些实用的练习，可以通过它们创建一个数据库项目，并增加脚本、查询以及命令文件。

9.1 为什么要使用数据包装？

总的来说，使用数据外壳和类或组件的原因有很多。将数据访问打包成一个类可以避免直接访问你的数据。你可以为数据建立访问权限，这样就只有管理员或者是合法成员才能访问数据，从而更好地控制对数据的访问方式，并避免数据的潜在篡改危险。

对一个组件里的数据打包可以将该组件放到网络里的最佳位置。所谓最佳位置，即服务器的某个位置，客户可以很容易地访问组件，组件也很容易更新，且由于客户使用而导致的工作负荷也可以得到有效的处理。这显然包括了发布应用程序的方法以及使用的是何种应用程序模型（客户—服务器、多层结构，等等）。

9.2 关于面向对象编程

本节讲述 OOP 的一些基本原则。这只是一个简短的介绍，如果你对于 OOP 完全陌生，则建议你多阅读一些资料，以便能够最大限度地发挥 Visual Basic.NET 的潜力。建议阅读以下书目：

- Moving to VB NET: strategies, and Cocol, 作者 Dan Appleman。ISBN 1-893-115-97-6。2001 年 6 月出版。

- **Doing Web Development: Client Side Technologies**, 作者 Dan Appleman。ISBN 1-893-115-97-6。2001 年 6 月出版。
- **Programming VB.NET: A Guide for Experienced Programmers**, 作者 Gary Cornell 和 Jonathan Morrison。ISBN 1-893-115-99-2。2001 年 7 月出版。该书第 4 章和第 6 章讨论了 OOP。

我将对 OOP 和普通编程的指导方针做一比较,并告诉你怎样将 OOP 应用到 UserMan 实例中去。所以,本章为第 11 章作了一个预备性的简单介绍,而本章所讨论的原则将会贯穿该实例的始终。

下面我们来谈谈 OOP 的基本原则:多态性、继承和封装。

9.2.1 多态性

多态性 (polymorphism) 也称为多种形态 (many form), 是对某种对象调用特殊方法的能力, 该对象实例化自基类派生的任何类, 并且无需知道该对象属于哪个类。

下面用框架基类中的标准对象类来说明这点。框架类中的其他类都是由这个类派生出来的, 而且都暴露 **ToString** 方法。该方法只返回实例化后对象的字符串表达。也就是说, 你可以在任何时候对一个对象调用 **ToString** 方法来返回一个字符串表达, 而无需知道该对象属于哪个子类。

这就意味着继承自其他类的类应该和基类暴露同样的属性和方法, 也就使得程序员可以很方便地对所有对象使用标准的方法和属性。

如果可以覆盖, 你可以对继承的方法和属性进行扩展, 这样, 你就不用暴露基类的函数特性 (下一节将讨论继承性方面的知识)。

我给出的解释很简单, 但从本质上讲, 这就是多态性。如果要知道更多关于多态性在 VB.NET 中实现的信息, 请参阅 “与 OOP 相关的 Visual Basic.NET 键” 一节。

9.2.2 继承

继承 (inheritance) 是指一个类从其他类派生出来的能力, 或者顾名思义, 指继承其他类的特点的能力。你会遇到诸如超类 (superclass) 或者基类 (base class) 这些描述被继承类的词汇, 以及子类 (subclass) 或者派生类 (derived class) 这些描述从其他类派生出来的类的词汇。

VB.NET 支持两种继承类型: 接口继承 (interface inheritance) 和实现继承 (implementation inheritance)。

接口继承和实现继承

在早期的 Visual Basic 版本中, 只支持用 **Implements** 关键词进行的接口继承, 但是现在有了实现继承。接口继承也称 “具有 (has a...)” 继承, 而实现继承通常称为 “是 (is a...)” 继承。

接口继承

接口是一套相关的公用方法、属性、事件和索引器, 它们都是派生类所必须实现的。接

口就是一种契约，因此，如果在你的类中实现了某种接口，你就受到了这种接口所带来的契约限制。即你必须实现所有方法以及接口所指定的那些方法。

接口不包括实现，这就意味着你不能直接实例化接口。相反，接口必须由类来实现，而你只能实例化这个类。接口是一种抽象的数据类型，它在接口本身和实现它的类之间提供一种松散的耦合（Ccoupling）。程序清单 9.1 示例了如何实现接口继承。

程序清单 9.1 创建和实现一个接口

```

1 Public Interface IUserMan
2     Property TestProperty() As String
3     Sub TestMethod()
4 End Interface
5
6 Public Class CUserMan
7     Implements IUserMan
8
9     Private prstrUserName As String
10
11     Public Property UserName() As String Implements IUserMan.TestProperty
12         Get
13             UserName = prstrUserName
14         End Get
15
16         Set(ByVal vstrUserName As String)
17             prstrUserName = vstrUserName
18         End Set
19     End Property
20
21     Public Sub CalculateOnlineTime() Implements IUserMan.TestMethod
22     End Sub
23 End Class
24
25 Public Sub CreateUserManObject()
26     Dim objUserMan As New CUserMan()
27 End Sub

```

在程序清单 9.1 中创建了 IUserMan 接口，它定义了 TestProperty 属性和 TestMethod 方法。在 CUserMan 类（第 6~23 行）中，在第 7 行通过 **Implements** 语句实现了 IUserMan 接口。正如你在代码中所看到的，在实现接口的类中引用公用特征（方法、属性、事件和索引器）是很必要的，但并不要求你保持这些特征的名称。注意第 11 行和第 21 行，我在 **Implements** 键后做了属性和方法的声明，指出这些属性和方法都实现了接口中的哪些特征。

然后创建了一个过程，并通过它创建了一个实现该接口的类的实例（第 25~27 行）。可以说，这是实例化接口的惟一方法。

实现继承

当你可以用“**Is a...**”短语指代一个子类或者派生类时，就应该使用实现继承。也就是说

子类属于超类或者子类和超类是同一类型。例如，假设有一个叫做 `CBird` 的超类，以及叫做 `CHedgeHawk` 和 `CPigeon` 的子类。显然，`hedge hawk`（一种鹰）和 `pigeon`（鸽子）都是 `bird`（鸟）类。这就是“**Is a...**”短语的含义。

我们可以为基类赋予很多功能属性，而通过实现继承我们就可以将这些功能赋予基类所派生的类，而不需要对它们进行扩展。在这点上，实现继承的优点体现得非常突出。这不是说不能对基类代码进行扩展，但是既然实现继承能满足你的需求，与其扩展基类的大部分代码，实现继承岂不是更好？实际上，做出这样的选择并非容易的事。当你考虑使用实现继承的时候，要牢记“**Is a...**”短语。这有助你做出决策。

实现继承较之接口继承最明显的一个优点，就是你可以避免将所有的标准代码都放入超类中而可能产生的复制代码。

实现继承可以使超类和子类间形成紧密的结合。对于实现继承的具体使用方法请参阅本章后面的，程序清单 9.4 和程序清单 9.5。

9.2.3 封装

封装（**encapsulation**）这个词已经使用了多年。对于这个 OOP 术语，不同的人可能有截然不同的定义。我会尽量对封装及其在 VB.NET 中的使用 and 实现方法做一个简要地解释。

封装，也称作信息隐藏（**information hiding**），是指将程序源代码和实现方式对类用户隐藏的能力。这就意味着你在创建类的时候，暴露了一些公用属性和方法，但是你的代码决定了如何操控你的专用变量以及诸如此类的成分。如果你看到下文中的程序清单 9.5，就会发现在第 2 行定义了专用字符串变量 `prstrTest`。由于该变量被定义为专用变量，所以类用户不能直接访问它。相反，还建立了 `Test` 属性，它可以提供对变量的访问。在设置属性的时候，实际上并没实现对传递值的任何检查，但通常这就是封装的概念：不能直接访问专用成员，只有通过公开的方法或者是属性才能验证输入。具体例子请见程序清单 9.2 中属性 `Set` 过程中对所传递的参数的验证。

程序清单 9.2 验证属性 `Set` 过程的参数

```
1 Set(ByVal vstrTest As String)
2     ' 检查字符串是否包含非法字符
3     If InStr(vstrTest, "@") = 0 Then
4         prstrTest = vstrTest
5     End If
6 End Set
```

9.2.4 与 OOP 有关的 Visual Basic.NET 键

在 Visual Basic.NET 中，有一些键是和 OOP 直接或间接相关的。这些键如下：

- **MustInherit**: 任何有 **MustInherit** 标注的类都只能作为基类，而且不能被实例化。如果你试图通过 **New** 操作符来实例化这样的类，在编译的时候就会出错。**MustInherit** 不能用于声明类的方法和属性。其错误使用方法如程序清单 9.3 所示，正确使用方法则如程序清单 9.4 所示。

- **Overridable**: 有此键标注的属性或者方法在派生类中可以被覆盖。但也并非一定要覆盖它。如果你希望确保其在子类中会被覆盖,就要使用 **MustOverride** 键来做出声明。怎样在派生类中覆盖属性请参阅程序清单 9.5。
- **MustOverride**: 有此键标注的属性或方法在派生类中一定会被覆盖。如果你只是想让编程者自主选择是否覆盖,则必须用 **Overridable** 键声明。
- **NotOverridable**: 如果你用此键对属性或方法进行标注,即显式表示派生类中不能覆盖该方法或属性。你不必说明这点,因为它是默认行为。但是,这样做会使你的程序更为易读。
- **Overrides**: 标有此键的方法或属性会覆盖掉基类中同名的方法或属性。如果你要覆盖在基类中声明的方法或属性,就一定要使用这个键。如何在派生类中覆盖属性请参阅程序清单 9.5。
- **Overloads**: **Overloads** 语句是为声明同类中同名的多个程序而设置的。这些程序惟一的的不同之处是其参数表。在调用这些程序的时候,编译器会根据所提供的变量进行选择。
- **Inherits**: 这个键用来从其他类(也就是超类或者基类)进行继承。请参阅程序清单 9.4 的示例。
- **Implements**: **Implements** 键用来指定一个类、方法、属性、事件或索引器会实现接口中对应的特性。使用 **Implements** 键的例子见程序清单 9.1。
- **Interface**: **Interface** 键用来指定某个接口。使用 **Interface** 键的例子见程序清单 9.1。

程序清单 9.3 不能实例化的 MustInherit 类

```

1 Public MustInherit Class CMustInherit
2     Private prstrTest As String
3
4     Public Property Test() As String
5         Get
6             Test = prstrTest
7         End Get
8
9         Set(ByVal vstrTest As String)
10             prstrTest = vstrTest
11         End Set
12     End Property
13 End Class
14
15 Public Sub InstantiateMustInheritClass()
16     Dim objMustInherit As New CMustInherit()
17 End Sub

```

在程序清单 9.3 中, CMustInherit 类被键 **MustInherit** 标注,也就是说在实例化类的实例的时候,只能由另外的类来继承。这也就意味着第 16 行会在编译时由于语法错误而抛出异常。

但是，如果你像程序清单 9.4 中那样将 `CMustInherit` 打包到 `CWrapper` 类中，就可以像第 7 行所示继承和实例化 `CWrapper` 类了。

程序清单 9.4 可以实例化的 `MustInherit` 类

```
1 Public Class CWrapper
2     Inherits CMustInherit
3     ' Place your other code here 将其他代码放在这里
4 End Class
5
6 Public Sub InstantiateMustInheritClassWrapper()
7     Dim objMustInherit As New CWrapper()
8
9 End Sub
```

程序清单 9.4 中有错误，会在编译时抛出异常。程序清单 9.5 向你展示了正确的方法。

程序清单 9.5 在派生类中覆盖属性

```
1 Public Class COverridable
2     Private prstrTest As String
3
4     Public Overridable Property Test() As String
5         Get
6             Test = prstrTest
7         End Get
8
9         Set(ByVal vstrTest As String)
10             prstrTest = vstrTest
11         End Set
12     End Property
13 End Class
14
15 Public Class CWrapper
16     Inherits COverridable
17
18     Private prstrTest As String
19
20     Public Overrides Property Test() As String
21         Get
22             Test = prstrTest & "Overridden"
23         End Get
24
25         Set(ByVal vstrTest As String)
26             prstrTest = vstrTest
27         End Set
28     End Property
29 End Class
```

在程序清单 9.5 中，声明了两个类 `COverridable` 和 `CWrapper`。`COverridable` 是一个基类，`CWrapper` 由它继承而来。在第 4 行 `Test` 属性被声明为可覆盖，而在派生类中的第 20 行，该属性的确被 **Overrides** 键语句所覆盖了。当你在覆盖方法或属性时，**Overrides** 键是很必要的，而无论这种方法或者属性是用 **MustOverride** 还是 **Overridable** 键声明的。所以，如果你将第 20 行中的 **Overrides** 键删除，在编译时就会报错。

9.3 将数据库打包

我们已经清楚了一些初步的东西，下面来看看可以打包的内容，以及对数据库打包的情况。如果你对第 2 章中关于数据库的概念已经比较熟悉了，那么当进入分析和设计阶段的时候，也就不会再有什么困难了。基本上，对数据库的访问进行打包就像是在相关的表格间对数据进行分组一样。实际上，由于你已经将数据分配到了各个表格，所以至少找到了对这些类和外壳定义的一种方法。

为数据库中的每个表格都建立一个类，从维护的观点来看这通常是一个最让人满意的解决方案，同时也考虑到了这些类和组件的使用者。现在提到了组件，所以我必须声明类不一定要包含组件。组件中通常含有多个类，但具体情况要根据你发布商业和数据服务的方法来判断。

那么在这里到底需要做什么呢？首先要创建类，或者可能是含有类的组件。由于本书只限于讲述数据的操作，因而不会提及商业服务，即只关注数据服务。如果你看过 `UserMan` 数据库的模式或架构，就会看到如下几个表：`tblLog`、`tblRights`、`tblUser` 和 `tblUserRights`。

根据上述所讲的，我们需要为这些表分别创建一个类。为了说明做什么以及怎么做，接下来先创建一个 `CUser` 类。

9.3.1 创建 `CUser` 类

`tblUser` 表有以下几列：`Id`、`ADName`、`ADSID`、`FirstName`、`LastName`、`LoginName` 和 `Password`。

现在要做的是找出需要在类中列出哪几列，同时，如果列出某一列，那么要判断它是可读写的、只读，还是只写。列、数据类型和访问类型请参阅表 9.1。

表 9.1 `tblUser` 的各列

列名	数据类型	可空否	访问类型	说明
<code>Id</code>	<code>int</code>	不可空	只读	这是在相关表中查找值的 <code>Id</code> ，是一个身份栏，所以你不需要为它赋值
<code>ADName</code>	<code>varchar</code>	可空	可读写	这是用户的 Active Directory (AD) 名，如果他或她是 AD 成员
<code>ADSID</code>	<code>uniqueidentifier</code>	可空	可读写	这是用户的 Active Directory 安全标识(SID)，如果他或她是 AD 成员

续表

列名	数据类型	可空否	访问类型	说明
FirstName	varchar	可空	可读写	这是用户的名字
LastName	varchar	可空	可读写	这是用户的姓氏
LoginName	varchar	不可空	可读写	这是用户的登录名，当他或她登录的时候，必须和密码一起使用
Password	varchar	不可空	可读写	这是用户密码，当他或她登录系统的时候，必须和用户名一起使用

现在，如果你看看表 9.1，就会发现一些列的访问类型需要更深入地讨论。ADName 和 ADSID 可以根据连接的 AD 不同而变化，但是，为什么希望类的使用者可以进行这样的操作呢？首先，需要确定这些值是可以设置的。然后，通过应用程序监控那些可以进行操作的人。

接下来，需要创建一个新的类库工程，然后向其中添加一个包括 CUser 类的方法、属性等等的新类。

练 习

创建一个新的类库工程，名为 UserMan，将已经存在的类（Class1）更名为 CUser。打开 CUser 类并将类名改为 CUser。

现在是加入可以操作数据库表访问的专用变量、方法和属性的时候了。最好对所有的列都创建属性，因其所有操作都是读出或者写入一个值。在创建属性之前，最好建立专用变量，这样就可以避免在读取变量值的时候每次都要访问数据库。创建含有用户数据的专用变量的代码示例请参见程序清单 9.6。

练 习

在 CUser 类中，为 tblUser 表中的每一列都添加一个专用变量。请见程序清单 9.6 的示例。

程序清单 9.6 创建包含用户值的专用变量

· 用户表列的值

```
Private prlngId As Long
Private prstrADName As String
Private prguiADSID As Guid
Private prstrFirstName As String
Private prstrLastName As String
Private prstrLoginName As String
Private prstrPassword As String
```

程序清单 9.6 仅仅是与 `tblUser` 表中各列直接对应的专用变量列表。现在需要加入可以设置和返回变量值的属性。`Id` 属性的示例请参见程序清单 9.7。

程序清单 9.7 只读的 `Id` 属性

```
Public ReadOnly Property Id() As Long
    Get
        Id = prlngId
    End Get
End Property
```

练 习

在 `CUser` 类中，为 `tblUser` 表的各列都添加属性。请参见程序清单 9.7 和程序清单 9.8 的示例。

程序清单 9.8 可读写的 `ADName` 属性

```
Public Property ADName() As String
    Get
        ADName = prstrADName
    End Get
    Set(ByVal vstrADName As String)
        prstrADName = vstrADName
    End Set
End Property
```

程序清单 9.8 说明了怎样通过属性访问数据库表中的 `ADName` 列。这么说不完全正确，因为它只访问了一个专用变量。但是第 11 章会加入访问数据库表和 Active Directory 的代码。

数据类型 **String** 的所有属性都可以设置在数据库中的最大长度，所以，最好检查所有通过的字符串长度是否小于或等于这个长度。这也意味着你必须将这个最大值赋给专用变量，就像程序清单 9.9 所示的那样。

练 习

在 `CUser` 类中，为 `tblUser` 表的每一列都添加专用变量，并设置最大长度。请参见程序清单 9.9 的示例。

程序清单 9.9 创建包含表列最大长度的专用变量

```
· 用户表列的最大长度
Private prshtADNameMaxLen As Short
Private prshtFirstNameMaxLen As Short
Private prshtLastNameMaxLen As Short
```

```
Private prshtLoginNameMaxLen As Short  
Private prshtPasswordMaxLen As Short
```

现在，需要对属性 **Set** 过程创建长度检查。如程序清单 9.10 所示。

练 习

对属性 **Set** 过程中所传递的参数添加长度检查。记住，只对那些对应于数据库表中某列的变量设置最大长度。参见程序清单 9.10 的示例。

程序清单 9.10 为属性 **Set** 参数创建最大长度检查

```
Set(ByVal vstrADName As String)  
    If Len(vstrADName) <= prshtADNameMaxLen Then  
        prstrADName = vstrADName  
    Else  
        prstrADName = Left(vstrADName, prshtADNameMaxLen)  
    End If  
End Set
```

在程序清单 9.10 中，通过比较 **vstrADName** 参数的长度和专用变量 **prshtADNameMaxLen** 的值，可以判断 **vstrADName** 参数长度是否小于或等于表列的最大长度限制。如果是，那么就存储数据，如果不是，则存储 **vstrADName** 数据串的前 **prshtADNameMaxLen** 位数字。我想你可能对此提出异议，但是，移植过长数据是不好的。我设置这种检查的原因是，如果我试着向数据库移植一个过长的数据，那么就会出错。诚然，可以通过不同的方式来进行检查，而你也一定知道一到两种其他你所常用的方式，但是，这仅仅表明你可以通过什么方式建立这些属性。

CUser 类还远没有完成（还需要加入很多代码，例如方法，等等），但至此你应该已经知道要怎样为自己的数据访问建立外壳了。第 11 章会通过完成 **UserMan** 实例给你更多的概念和提示。

9.4 小结

本章对 **OOP** 和怎样对你的数据访问代码使用 **OOP** 进行了简要介绍。我们介绍了有关 **OOP** 的三个主要概念：多态性、继承和封装。关于继承，还简要描述了接口继承和实现继承的不同。

在 **UserMan** 应用程序示例中，通过为 **tblUser** 数据库表的访问建立类讲述了怎样将数据访问代码打包到类。这项工作将会在第 11 章中完成。

下一章讲述的是如何创建和使用数据捆绑控件。

第 10 章

数据绑定型控件

创建和使用数据绑定型控件

本章将介绍数据绑定型控件，并讨论为何有的开发者非常喜欢使用它，而有的开发者则不然。我们将研究如何使用在 Visual Basic.NET 中出现的各种数据绑定型控件。此外，还将为你介绍如何创建自己的数据绑定型控件。请注意本章中的实践练习。

10.1 数据绑定型控件与手工数据引入

数据绑定型控件 (data-bound control) 是与数据源挂钩的控件，例如列表框、网格或文本框。这意味着，如果用户的输入改变了控件中的数据，开发者在主要数据发生变化或者更新数据源时并不需要更新控件。数据绑定型控件使开发者的工作更为简单。

手工数据引入是数据绑定型控件的一个可用选择。手工数据引入描述了开发者从数据源中检索值，并在一个或多个 UI 控件中显示结果的过程。但是，当 Visual Basic 开始用数据绑定型控件来传输数据时，开发者突然间有了一个特别的工具来显示和处理数据。

数据绑定型控件是一个非常好的想法，因为它可以免除开发大量代码的麻烦，如果你使用手工数据引入，则不得不开发大量代码。手工数据引入可以对数据处理进行完全的控制，但是数据绑定型控件在显示数据或者将数据写回到数据源之前，通常对数据的干预和处理具有严格的限制。这一度曾是数据绑定型控件的一项令人厌烦的“特别功能”，肯定有人曾出于这个原因告诫你不要使用数据绑定型控件。我则主要为显示目的或者只对只读数据才使用数据绑定型控件，因为一旦用户修改了显示的数据，就不必再为更新数据源的问题而烦恼。

习惯做法是，先将所有数据绑定型控件都绑定到一个单独的数据控件上，然后再将这个数据控件绑定到数据源中，如图 10.1 中所示。

你可以在图 10.1 中看到该数据控件是如何处理数据源与数据绑定型控件之间的通信。这就有力地说明了程序员对于从数据源传输到绑定型控件以及回传的数据没有控制权。数据控件为你封装了所有操作，但是你需要对数据源如何更新做一些设置。

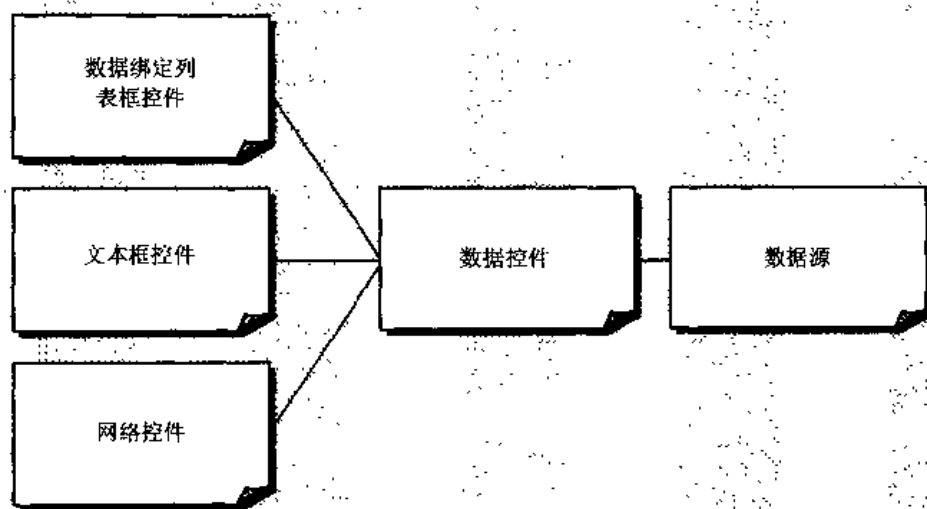


图 10.1 传统数据绑定型控件

10.2 用于不同 UI 的不同控件

Visual Studio.NET 支持多种 UI，这些 UI 各自需要不同的控件。Web 和 Windows 就是 VS.NET 所支持的两种 UI。例如，如果你查看 IDE 中的工具箱，就可以看到不同用途的控件选项卡。图 10.2 中显示了带有 Windows Forms 扩展选项卡的工具箱。如果你在 IDE 中显示工具箱并查看 Windows Forms 选项卡和 Web Forms 选项卡，那么你会发现许多控件的命名都相同。这并不意味着它们是相同的控件，相反，它们是不同的控件。但是，它们的用途很相似。继续学习本章，你会了解到关于这些重要差别的更多内容。

10.2.1 使用带有 Windows 表单的数据绑定型控件

Windows 表单中的数据绑定需要数据提供者和数据使用者。数据提供者并不是传统意义上的数据源（如数据库表）。在 Windows 表单的数据绑定中，数据提供者的概念要广泛得多，它可以包括集合、数组，或者来自 ADO.NET 的任何数据结构。这些数据结构包括 **DataReader** 和 **DataSet** 类。本节将介绍带有 ADO.NET 数据结构的数据绑定。



关于 **DataReader** 和 **DataSet** 数据结构的更多内容，请参阅第 3A 章和第 3B 章。

所有的 Windows 表单都具有一个 **BindingContext** 对象。当你将一个 Windows 表单的控件绑定到一个数据源时，该控件将具有一个相关的 **CurrencyManager** 对象。**CurrencyManager** 对象处理与数据源的通信，并负责保持数据绑定型控件的同步。所有绑定型控件同时显示来自同一行的数据，一个 Windows 表单可以具有多个流通管理器（**CurrencyManager**）。当 Windows 表单具有多个数据源时，就会出现多个流通管理器，因为

表单为与之相关的每个数据源都保留一个流通管理器。

关于流通管理器的另一个重要内容就是其知道数据源中当前位置的功能。你可以使用 **CurrencyManager** 类的 **Position** 属性读取位置。这对于 ADO.NET 数据结构特别有用（例如 **DataTable** 类），因为它不提供指针功能，也就是说你不能检索到当前行的指针或者指示器。但是，通过使用流通管理器(Currency manager)可以做到。在本章后面“查看由数据表单向导创建的代码”一节中，你将学习到如何使用 **CurrencyManager** 对象。

检查绑定上下文

绑定上下文跟踪 Windows 表单的所有流通管理器。即使没有流通管理器和数据源，Windows 表单也会始终具有一个与之相关的 **BindingContext** 对象。事实上，这对于源自 **Control** 类（它也是 **System.Windows.Forms** 名称空间的一部分）的所有类都适用。

数据绑定型控件借助绑定上下文来进行通信。绑定上下文先与流通管理器进行通信，然后流通管理器再与数据源进行通信，如图 10.3 所示。本章后面“查看由数据表单向导创建的代码”一节中将介绍如何在代码中使用 **BindingContext** 对象。



图 10.2 带有 Windows Forms 扩展选项卡的工具箱

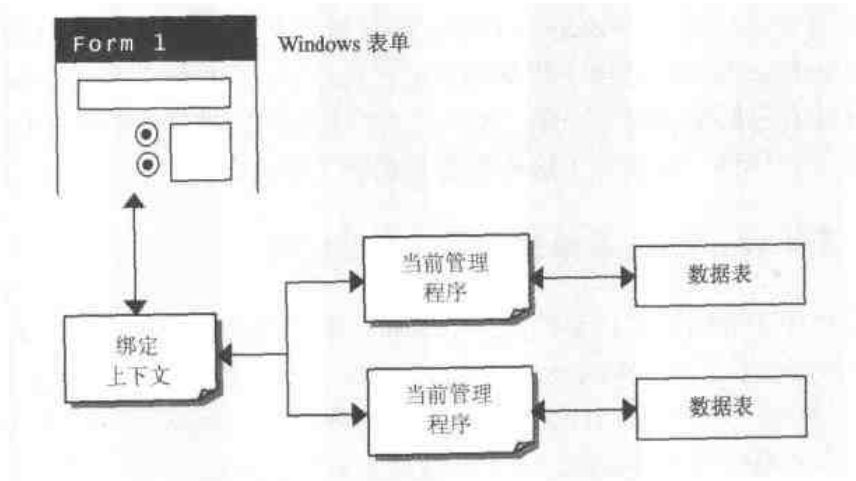


图 10.3 数据绑定型控件如何与数据源进行通信

使用数据表单向导创建表单

你可以用数据表单向导来创建带有一个或多个数据绑定型控件的表单，从而避免亲自做大量繁琐的工作。下面将为你介绍这一过程，因为它能帮助你理解控件如何绑定到 Windows 表单中的数据。以下是创建一个新表单的步骤：

1. 将新项目添加到工程中，并选择 [Data Form Wizard] 模板。
2. 为新表单命名。然后会出现数据表单向导。单击 [Next] 按钮。

3. 出现 [Choose the dataset you want to use] 对话框（见图 10.4）。如果要创建一个新的数据集，那么请选择 [Create a new dataset named] 选项，并在文本框中输入数据集的名称。否则请选择 [Use the following dataset] 选项，并从下拉列表框中选择一个现有的数据集（该选项仅在工程中已经具有数据集时才可用）。然后单击 [Next] 按钮。

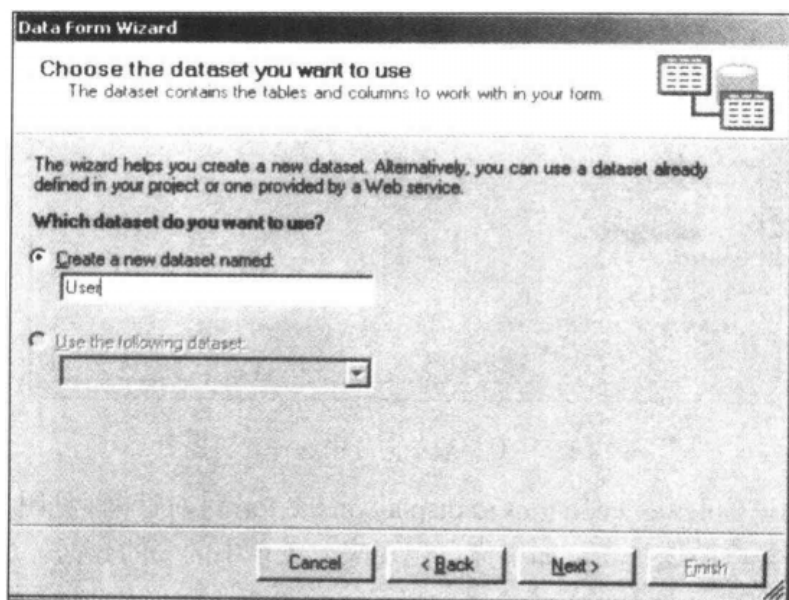


图 10.4 在数据表单向导中选择数据集

4. 出现 [Choose a data connection] 对话框（见图 10.5）。在 [Which connection should the wizard use?] 下拉列表框中选择所需的数据库连接，接着单击 [Next] 按钮。

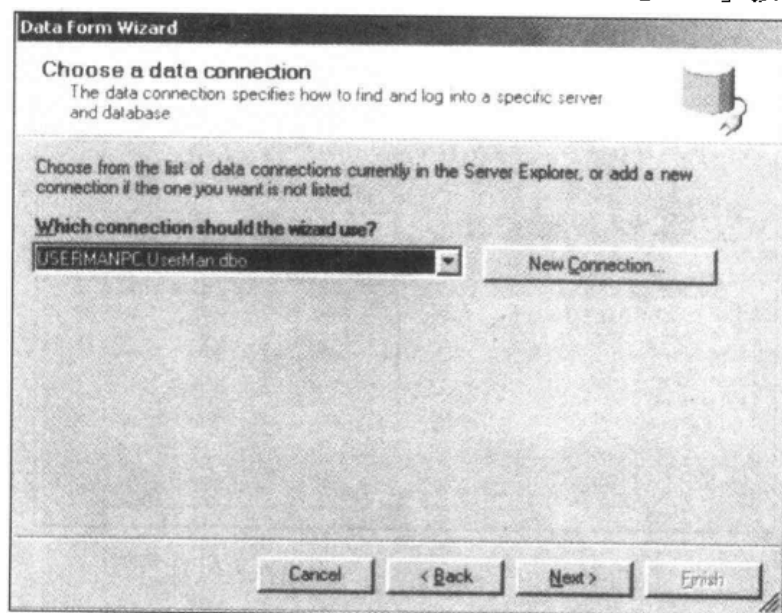


图 10.5 在数据表单向导中选择连接

5. 出现 [Choose tables or views] 对话框（见图 10.6）。从 Available item (s) 树状视图中选择所需表或视图，并从 Selected item (s) 树状视图中选择数据表单。单击向右键头按钮进行选择，然后单击 [Next] 按钮。

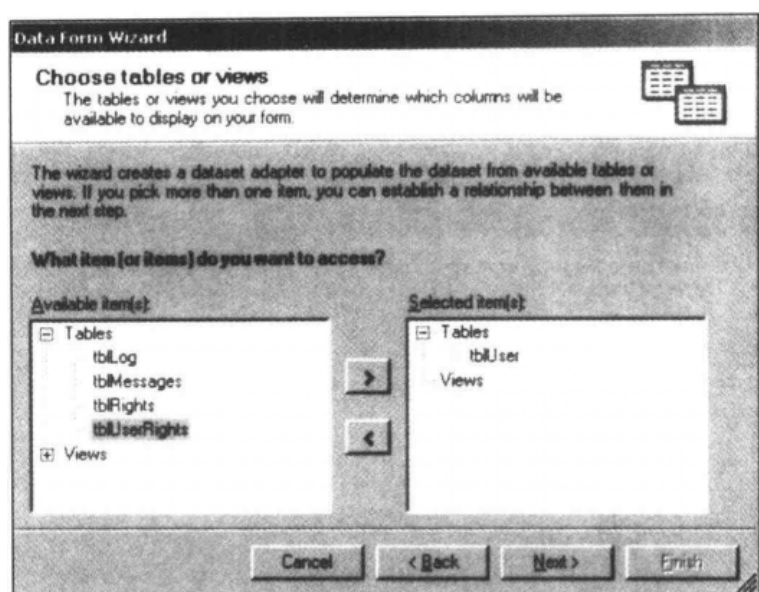


图 10.6 在数据表单向导中选择表和视图

6. 出现 [Choose tables and columns to display on the form] 对话框 (见图 10.7)。从 [Master or single table] 下拉框中选择表。如果你只想要来自表单中的表的数据, 那么做到这一步就足够了。如果你要创建一个主要细节表单, 那么还必须在 [Detail table] 下拉框中选择表。

7. 在 [Columns] 列表框中选中在表单中所要显示的列 (右边的 [Columns] 列表框仅在从 [Detail table] 下拉列表中选择表时才被激活), 接着单击 [Next] 按钮。

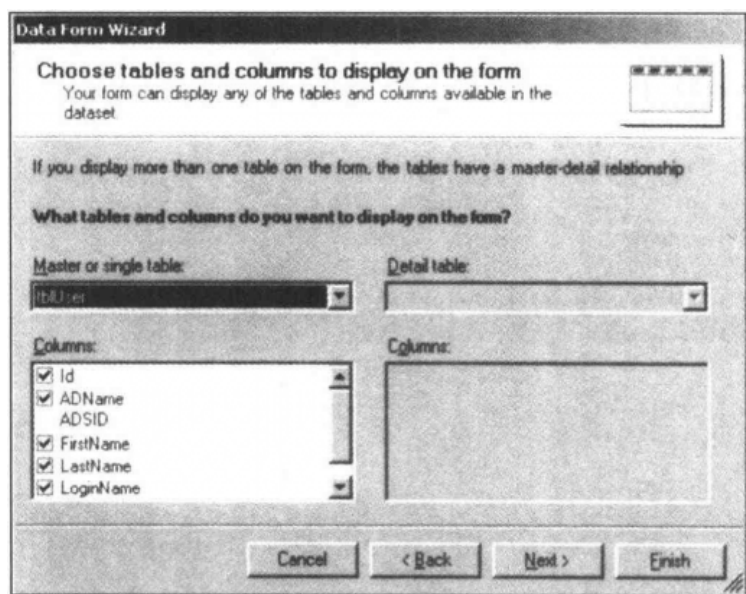


图 10.7 在数据表单向导中选择表和列

8. 出现 [Choose the display style] 对话框 (见图 10.8)。如果你想要在网格中显示数据, 那么请选择 [All records in a grid] 选项。如果希望能够定位和处理数据, 那么请选择 [Single record in individual controls] 选项。根据是否要将数据设为只读的要求, 可以选中和取消 [Cancel All] 复选框。

9. 最后单击 [Finish] 按钮。

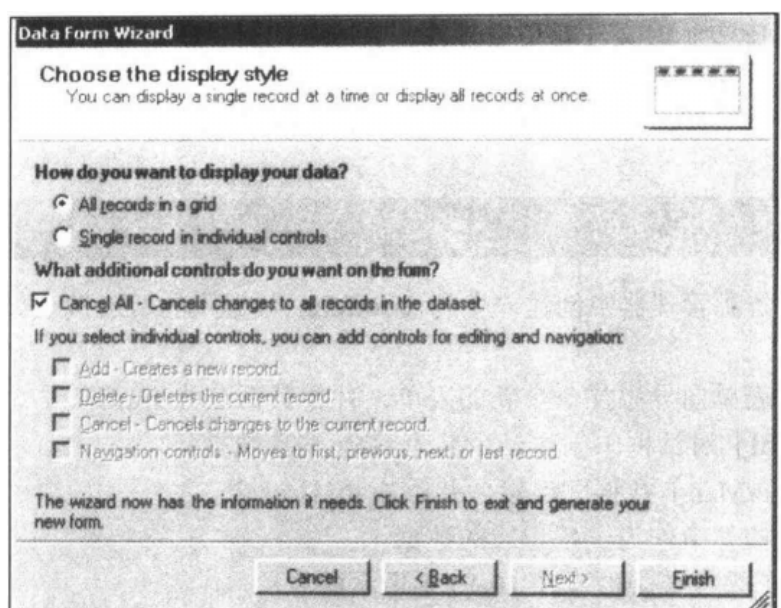


图 10.8 选择在数据表单向导中显示数据的方式

现在，数据表单便已创建完了并显示在 IDE 中。正如你在图 10.9 底部所看到的，向导创建了 objUser 数据集、OleDbconnection1 连接和 OleDbDataAdapter1 适配器。这些对象仅供新表单使用。如果你打开文件（.vb）的代码，则将看到向导按照你的要求所编写的所有代码。你可以按照自己的需要来修改这些代码，因为它们全部都在这里。这与 Visual Basic 以前的版本有着显著区别。你可以看到向导如何创建所有的 OLEDB.NET Data Provider 数据对象，例如插入、更新、删除及选择命令、数据适配器和连接对象。



关于数据适配器的讨论，请参阅第 3A 章。

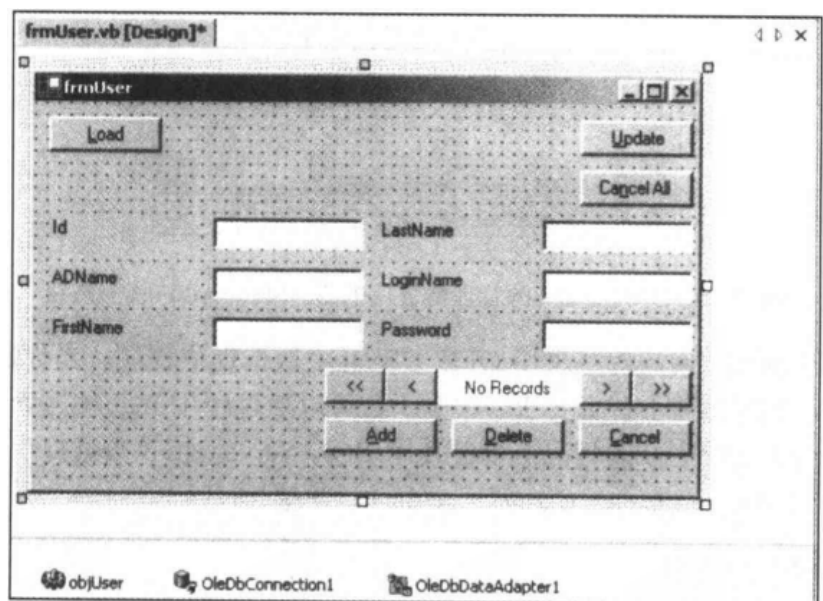


图 10.9 由数据表单向导创建的表单

向导还创建了一个强类型的 XML 数据集文件（.xsd）。另一个由向导创建但通常不在解

决方案资源管理器中显示的文件就是类文件，它压缩或封装了数据访问。这个文件的名称与数据集文件（.xsd）相同，但是带有 .vb 扩展名。你可以在存储工程的文件夹中看到这个文件，或者可以单击解决方案资源管理器中的 [Show All Files] 按钮。

练 习

1. 从服务器资源管理器中创建一个到 UserMan 数据库的新连接（如果目前还没有连接的话）。
 2. 使用数据表单向导添加一个新的表单，并将其命名为 frmUser。
 3. 在 [wizard] 对话框中创建一个名为 User 的新数据集。
 4. 选择 [UserMan] 数据库连接，并添加 tblUser 表。
 5. 显示单独控件中作为一条记录的数据。
- 新的表单看起来应该类似于图 10.10 中所示的表单。

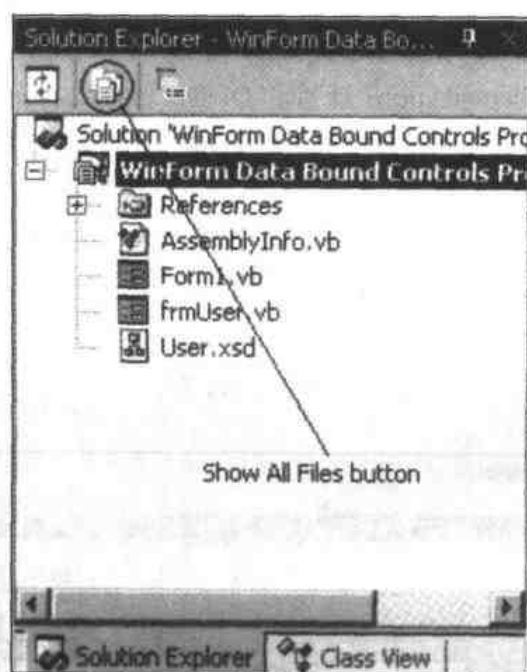


图 10.10 解决方案资源管理器窗口中的 [Show All Files] 按钮

你可能已经注意到，向导始终使用 OLE DB.NET 数据提供程序。当你连接到 SQL 服务器上时也是如此，这显然并不符合要求。但是，这可以通过使用 IDE 的搜索和替换功能来修改。我曾经定制过由向导生成的代码，以便使用 SQL Server.NET 数据提供程序，你可以在 Apress Web 站点 (<http://www.apress.com>) 上找到该代码。

由数据表单向导创建的代码

接下来将为你介绍由数据表单向导创建的代码，如果你想继续学习，请你打开包含在前一个练习中创建的 frmUser 表单的工程。如果你没有做那个练习，那么现在最好做一下，因

为下面将介绍如何在代码中使用 **CurrencyManager** 和 **BindingContext** 对象。

如果你是在 **Code Editor** 中而不是在 **Design View** 中打开 **frmUser** 表单，那么会看到几百行的代码。先用一分钟考虑一下：如果你自己键入这些代码需要多长时间？在这里想说的是，即使是由数据表单向导生成的代码也需要轻微地调整，它比起你自己键入所有的代码来说，节省了大量的时间。

为了更好地学习 **Code Editor** 中的代码，请展开 **Windows Forms Designer generated code** 区域，方法是单击文字“**Windows Forms Designer generated code**”行号旁边的加号。这类似于在 **Windows Explorer** 中搜索文件时所做的操作。

接着向下滚动代码，一直到达 **New** 构造函数 (**Sub**) 和 **Dispose** 方法。请注意，在这些方法的后面，向导放置了表单使用的所有对象的声明。其中包括数据对象，例如数据适配器、连接和命令对象，以及表单上所显示的所有按钮和标签。所有这些对象都使用 **Friend** 访问修改器来声明，这意味着只有表单类自身和同一工程中的其他类才可以访问这些对象。它们也可以使用 **WithEvents** 语句来声明，这意味着所有事件都会在代码中显示，这样你就可以通过编程来响应这些对象产生的事件。

请仔细学习这个声明，它是 **objUser** 对象。格式为 **<ProjectName>**，其中 **ProjectName** 是工程名，**User** 是类。你无法想起如何创建该类吗？不必担心，因为这是由向导来完成的。右键单击该类并选择 **[Go To Definition]**。而后，会在 **Code Editor** 窗口中出现 **User** 类。这个类是源自 **System.Data** 名称空间中的 **DataSet** 类，它主要说明该类是你所要绑定的数据源的非连接部分。**User** 类位于 **User.vb** 文件中，其中还包括以下类：

- **tblUserDataTable** 类：源自 **System.Data.DataTable**。其公共属性为 **Count**，默认属性为 **Item**。该类还包括如计数器等程序，以及在表中处理行的程序。
- **tblUserRow** 类：源自 **System.Data.DataRow**。可以为获得和设置 **tblUser** 表中行的列值、析构造函数 (**New**)、检查列中 **Null** 值的布尔函数，以及将列值设为 **Null** 的功能来实现属性。
- **tblUserRowChangeEvent** 类：源自 **System.EventArgs**。该类是事件数据的基本类，这意味着该类用于控制某些事件，这些事件通过 **User** 类与 **tblUser** 表里的数据操作相连接。

在 **InitializeComponent Sub** 程序中，实例化并初始化所有的对象。

至此，你拥有了所有专用 **Sub** 程序，专用于表单上按钮的 **Click** 事件，用于更新当前行位置标签的 **Sub** 程序，用于更新、装载或填充 **DataSet** 的 **Sub** 程序，以及用于更新数据源的 **Sub** 程序。

你的确应该花很长时间仔细学习这些文件和类，因为其中含有大量知识。如果你在 **frmUser.vb** 文件中搜索 **BindingContext** 对象，那么你将会看到它在处理数据中所执行的实际操作。

将 Windows 表单控件绑定到数据源

任何源自 **Control** 类（它是 **System.Windows.Forms** 名称空间中的一员）中的 **Windows** 表单控件，都可以绑定到对象的属性和数据源上。如果查看本章前面部分由数据表单向导生成的代码，你可以了解到，表单上所有按钮的 **DataBindings** 属性是如何绑定到在前一节中所

提及的 `objUser` 对象上的。`DataBindings` 属性返回 `ControlBindingsCollection`，这个集合具有一个 `Add` 方法，用于添加 `Binding` 对象。该 `Binding` 对象是数据绑定的真正手段，它是由 `ControlBindingsCollection` 的 `Add` 方法创建的，指定了特定的控件属性如何绑定到一定的数据源成员上。以下示例代码是取自数据表单向导生成的代码：

```
Me.mfp_editId.DataBindings.Add(_
    New System.Windows.Forms.Binding("Text", Me.objUser, "tblUser.Id"))
```

`mfp_editId` 是一个 `TextBox` 控件，但是如前面所介绍的一样，你可以将数据绑定到源自 `Control` 类的任何控件上。在本例中，文本框绑定到 `objUser` 对象（数据源）中 `tblUser` 表（数据部分）的 `Id` 列。第一个“Text”参数指定了 `mfp_editId` `TextBox` 控件的 `Text` 属性绑定到 `Id` 列。

使用 `BindingContext` 对象

`BindingContext` 对象可以与 Windows 表单相关联，此时它是一个全局对象，因此你无需对它进行实例化。如果进行搜索，你会在代码中看到许多 `Me.BindingContext` 的实例，这些实例中指出了 `BindingContext` 对象是与当前 Windows 表单对象相关联还是与该对象的一部分相关联。`BindingContext` 类的非继承、公共属性如表 10.1 所示，非继承、公共方法如表 10.2 所示。

表 10.1 `BindingContext` 类的属性

属性名称	说明
<code>IsReadOnly</code>	这个只读属性返回一个布尔值，表明 <code>BindingContext</code> 是否为只读
<code>Item</code>	这个过载、只读属性返回一个特殊的 <code>BindingManagerBase</code> 。该属性是默认的，也就是说你不必指定它。你可以使用 <code>BindingContext.Item()</code> 或者 <code>BindingContext()</code> 。该属性有两种形式（过载）：一种是采用数据源对象，另一种是采用数据源对象和数据成员字符串

表 10.2 `BindingContext` 类的方法

方法名称	说明
<code>Contains()</code>	这个过载方法返回一个布尔值，表明 <code>BindingContext</code> 对象是否包含指定的 <code>BindingManagerBase</code>

正如你在表 10.1 和表 10.2 中所看到的那样，可使用的公共属性和方法并不多，但是这并不成问题。记住，`BindingContext` 对象为表单上的每个数据源都提供了一个 `CurrencyManager` 对象。这样你根本不必处理 `BindingContext` 对象，但是需要使用该对象的 `Item` 属性来访问 `CurrencyManager` 对象和 `Contains` 方法，以检查特定的 `CurrencyManager` 是否存在。你明白这部分内容了么？如果我没有为同一事物使用两种术语（`Currency Manager` 和 `BindingManagerBase`），你或许就明白了。

`BindingManagerBase` 类是抽象的，也就是说它在使用之前必须是派生或继承的。正如

你所猜到的：**CurrencyManager** 类的确源自 **BindingManagerBase** 类。因此，为了讨论 Windows 表单上的数据绑定型控件，这两个类或多或少都是一样的。事实上，它们具有相同的属性，只有一个方法能将它们区别开，即 **CurrencyManager** 类的 **Refresh** 方法。

在继续学习 **CurrencyManager** 类之前，先介绍一下如何使用 **BindingContext** 对象来得到它。在由数据表单向导创建的源代码中，使用 **Me.BindingContext** 来引用 **frmUser** 表单的 **BindingContext** 对象。程序清单 10.1 和程序清单 10.2 显示的是访问流通管理器的两种不同方法。

程序清单 10.1 保存到流通管理器的引用

```
Dim objCurrencyManager As CurrencyManager
objCurrencyManager = Me.BindingContext(objUser, "tblUser")
```

程序清单 10.2 通过绑定上下文访问流通管理器的 Count 属性

```
Me.BindingContext(objUser, "tblUser").Count
```

正如你在程序清单 10.1 和程序清单 10.2 中所看到的那样，可以通过 **BindingContext** 对象（如程序清单 10.2 中所示）来引用流通管理器，或者将引用保存在其中而直接使用它（如程序清单 10.1 中所示）。两种方法没有什么区别，究竟选用那个只是个人喜好问题。关于 **CurrencyManager** 类要注意的一点是它没有构造函数（**New Sub** 程序），因此你只能使用 **BindingContext** 对象的 **Item** 属性来实例化数据类型为 **CurrencyManager** 的对象。尽管程序清单 10.1 中的实例并没有遵守这个习惯，但是要记住，**Item** 属性是默认的，因此不必对其进行指定。所以，程序清单 10.3 中的代码同程序清单 10.1 中代码的作用是一样的。

程序清单 10.3 保存对指定 **Item** 属性的流通管理器的引用

```
Dim objCurrencyManager As CurrencyManager
objCurrencyManager = Me.BindingContext.Item(objUser, "tblUser")
```

借此机会重申一下我所做的事情，以及操作的方式。我不喜欢使用 Visual Basic 以前版本中的默认属性，但是出于某种原因 **Item** 属性始终例外，在 VB 的新版本中也是这样。因此，在本书的示例代码中没有出现 **Item** 属性。

关于 **Item** 属性，需要讨论的最后一点就是其调用方法。它是可重载的，但是在由数据表单向导生成的代码中，你仅能看到该属性的一种重载形式。这种形式是采用数据源对象和数据部分字符串的，而其他形式的属性仅将数据源对象作为一个参数。如果数据源是一个 **DataTable** 对象，那么就应该采用这种形式，因为这样你就不必再像数据源是数据集时那样指定表名称了。

使用 **CurrencyManager** 对象

现在我们来学习 **CurrencyManager** 类，它是一个使用非常广泛的类，当你创建数据绑定型控件时，会在代码中大量使用该类的所有属性和方法。表 10.3 列出了流通管理器的所有非继承、公共属性，表 10.4 列出的是所有非继承、公共方法。

表 10.3 CurrencyManager 类的属性

属性名称	说明
Bindings	这个只读属性返回一个 BindingsCollection 对象，它包括由 CurrencyManager 管理的所有 Bindings 对象
Count	Count 属性是只读的，它返回一个 Integer ，表明由 CurrencyManager 管理的行数
Current	该属性返回当前对象。由于返回的对象是数据类型 Object ，因此你在使用它之前需要使之具有与数据源所包含的数据类型相同的数据类型
Position	Position 属性检索或设置数据源中的当前位置。该值基于 0，数据类型是 Integer

表 10.4 CurrencyManager 类的方法

方法名称	说明	示例
AddNew()	AddNew 方法将一个新的项目添加到底层的列表中。在 DataTable 中，新的项目就是 DataRow	Me.BindingContext (objUser, "tblUser").AddNew() (取自数据表单向导生成的代码)
CancelCurrentEdit()	该方法取消当前的编辑操作。数据并不被存储到其当前状态中，而是数据回到你开始编辑操作的状态下。如果你要结束编辑操作并存储所有的改变，则应该使用 EndCurrentEdit 方法	Me.BindingContext (objUser, "tblUser"). CancelCurrentEdit() (取自数据表单向导生成的代码)
EndCurrentEdit()	EndCurrentEdit 方法用于结束当前的编辑操作。数据存储其当前的状态，因此它并不取消编辑。可以用 CancelCurrentEdit 方法取消编辑	Me.BindingContext (objUser, "tblUser"). EndCurrentEdit() (取自数据表单向导生成的代码)
GetItemProperties()	这个可重载方法用于检索或设置绑定的属性描述符的集合。返回值是数据类型 PropertyDescriptorCollection	Dim objPropertyDescriptor Collection As PropertyDescriptor Collection - Me.BindingContext (objUser "tblUser"). GetItemProperties()
Refresh()	Refresh 方法用于刷新或重新填充绑定型控件。该方法用于不支持修改提示的数据源，例如数组。数据集和数据表格支持修改提示，因此对这些数据类型的对象无需使用该方法	Me.BindingContext (objUser, "tblUser").Refresh()

续表

方法名称	说明	示例
RemoveAt(ByVal vintIndex As Integer)	该方法用于在指定索引位置删除项目	<code>Me.BindingContext(objUser, "tblUser").RemoveAt(0)</code>
ResumeBinding()	该方法和 SuspendBinding 方法用于临时暂停数据绑定。如果你要让用户执行一些编辑, 而又不想在恢复绑定之前就使之生效, 则应该同时使用这两种方法。该方法会恢复暂停的绑定	<code>Me.BindingContext(objUser, "tblUser").ResumeBinding()</code>
SuspendBinding()	该方法和 ResumeBinding 方法用于临时暂停数据绑定。如果你要让用户执行一些编辑, 而又不想在恢复绑定之前使之生效, 则应该同时使用这两种方法。该方法会暂停绑定	<code>Me.BindingContext(objUser, "tblUser").SuspendBinding()</code>

正如你在表 10.3 和表 10.4 中所看到的, **CurrencyManager** 类具有可以执行数据源上任何动作的方法和属性。

检索数据源中的行数

有时候为了显示的需要, 要检索数据源中控件所绑定的行数, 例如显示当前行是实际行数的数字 *n*。这可以通过使用 **CurrencyManager** 类的 **Count** 属性轻易做到, 方法如下:

```
MsgBox(" This is row number " & CStr( Me.BindingContext(objUser,& _
" tblUser").Position + 1) & _
" of " & Me.BindingContext(objUser, "tblUser").Count.ToString & "rows.")
```

该代码显然使用了 **Position** 和 **Count** 属性来显示当前行位置和行数。由于 **Position** 属性基于 0, 因此在显示的时候应为该属性赋值 1。这样就可以确保当前位置在 **Count** 属性是基于 1 的。

检索数据源的当前行

有时候, 你需要处理数据源中当前行的内容。这可以通过使用 **CurrencyManager** 类的 **Current** 属性来完成。但是, 该属性返回数据 **Object** 的值, 这样你就需要将该值设为与数据源所用相同的数据类型。当你绑定到 **DataSet**、**DataTable** 或 **DataViewManager** 时, 或者当数据源设置为这些数据类型中的一种对象时, 都要绑定到 **DataView** 对象。在这种情况下, 你需要将返回对象设为 **DataRowView** 对象。如下所示:

```
Dim drwCurrent As DataRowView = CType( Me. BindingContext ( objUser,&_"
tblUser").Current, DataRowView)
```

在该示例代码中, **drwCurrent** 保存了数据源中的当前行。

检索和设置数据源中的当前位置

可以使用 **CurrencyManager** 类的 **Position** 属性来检索和设置数据源中的当前位置。请参阅本章前面的“检索数据源中的行数”一节，学习如何读取 **Position** 属性的示例代码。

但是，检索数据源中的当前位置并不是 **Position** 属性所能完成的惟一任务，你还可以通过其设置位置。可以使用 **Position** 属性在数据源中来回移动，其最大的优点就是操作非常简单，如下例所示：

```
Me.BindingContext(objUser, "tblUser").Position = 3
```

该代码将第 4 行设为当前行（**Position** 属性是基于 0 的）。表 10.5 中显示了如何将特殊行设为当前行。

表 10.5 导航至绑定型控件的数据源中的特殊行

位置	示例
第一行	<code>Me.BindingContext(objUser, "tblUser").Position = 0</code>
最后一行	<code>Me.BindingContext(objUser, "tblUser").Position = Me. BindingContext(objUser, "tblUser").Count - 1</code>
下一行	<code>Me.BindingContext(objUser, "tblUser").Position += 1</code>
前一行	<code>Me.BindingContext(objUser, "tblUser").Position -= 1</code>

移动数据源中的连接后需要在实际操作之前检查移动的有效性。当移动到第一行或者最后一行时并不需要检查，因为这是始终有效的，但是在移动到前一行或者下一行时，如果目标行不存在，就会引起异常。

因此，如果你要移动到前一行，则先要确定目前不在位置 0（即第一行）上。检查方法如下所示：

```
If Not Me.BindingContext(objUser, "tblUser").Position = 0 Then
```

如果你要移动到下一行，那么要检查目前是否在最后一行，方法如下：

```
If Not Me.BindingContext(objUser, "tblUser").Position = _  
Me.BindingContext(objUser, "tblUser").Count - 1 Then
```

控制编辑操作的验证

当你在数据源中编辑数据时，会触发 **CurrentChanged** 和 **ItemChanged** 事件。这或许正是你需要的，通过响应这些事件在该位置上执行你所需的操作。但是，有时候并不需要出现事件。如果你正在进行大量的更新操作，那么肯定希望在更新完成之后再响应这些事件。这时，就可以使用 **SuspendBinding** 和 **ResumeBinding** 方法来完成该设置。这些方法用于临时暂停数据绑定。请参阅程序清单 10.4，学习如何暂停和恢复数据绑定的示例。

程序清单 10.4 暂停和恢复数据绑定

1. ' Suspend data binding (暂停数据绑定)


```
2 Me.BindingContext(objUser, "tblUser").SuspendBinding()  
3 ' Perform bulk edits (执行批编辑)  
4 . . .  
5 ' Resume data binding (恢复数据绑定)  
6 Me.BindingContext(objUser, "tblUser").ResumeBinding()
```

程序清单 10.4 显示了如何在执行大量编辑操作之前暂停数据绑定,并在完成编辑后再恢复数据绑定。在第 2 行和第 6 行之间没有触发事件。

10.2.2 创建自己的数据绑定型 Windows 表单控件

正如前面介绍过的,创建数据绑定型 Windows 表单控件非常简单。如果你要创建类似表单 UI 的控件,那么建议你始终使用数据表单向导来进行创建。即使在创建后需要修改一些由向导生成的代码,使用该方法也更为快捷和简单。

显然,当你使用表单之类的方法时,要创建一个具有 UI 的组件,而在某些情况中却不用这样做,即当你需要创建 Windows 用户控件时。你可以在任何 Windows 工程中这样做,步骤如下所示:

1. 将新项目添加到工程中。当显示 [Add New Item] 对话框时,选择 [User Control] 模板,为用户控件命名,并单击 [OK] 按钮。
2. 在设计模式中,从工具框中将所需的 Windows 表单控件拖拽到 Windows 用户控件上。
3. 创建一个类似于由向导生成的代码中的 objUser 类的类,即数据源。
4. 将你放置到 Windows 用户控件上的每个 Windows 表单控件的一个或多个属性绑定到数据源类的属性。

至此,所有的操作就完成了!

练习

创建一个新的用户控件,并命名为 UserManId。将 **TextBox** 控件添加到用户控件上,然后将其绑定到 tblUser 表的 Id 列。你可以在 Apress Web 站点的下载区中找完成该操作的实例。

10.2.3 使用带有 Web 表单的数据绑定型控件

绑定到 Web 控件的数据与 Windows 表单的数据在代码实现上有所区别,但是并不复杂。一些与 ASP.NET.Server 相关的控件具有 **DataSource** 属性(关于 ASP.NET 的更多内容,请参阅第 1 章)。**DataSoure** 属性用于绑定 **DataSet** 对象。



关于 **DataSet** 对象的更多内容,请参阅第 3B 章。

在继续介绍之前,先演示一下如何创建带有数据绑定型控件的 Web 表单,就像前面讲解

Windows 表单时所做的那样。

使用数据表单向导创建表单

如果你不想亲自完成所有繁琐的操作，可以像处理 Windows 表单那样，使用数据表单向导来创建带有一个或多个数据绑定型控件的 Web 表单。该向导可以为 Web 表单完成与处理 Windows 表单多少有些相似的工作，但是它只能创建只读的 Web 表单。这样，你就要亲自来创建可更新的页面了。建议你在 Web 应用程序工程中运行数据表单向导，执行与在 Windows 表单中相同的步骤（参阅本章前面“使用数据表单向导创建表单”一节）。请注意，你需要跳过步骤 10～步骤 12，因为它们无法应用到 Web 表单上。除了将 Web 表单限制为只读以外，数据表单向导的另一个缺点是它只允许数据网格作为 Web 表单上的控件。

总之，运行向导并在向导完成工作之后打开工程，观察它是如何在数据网格中以只读方式精致地显示数据的。



我在 Apress Web 站点上已经创建了一个新的工程，它是向导生成代码的一个扩展，你可以使用它来编辑和更新你的数据源。

维护状态

与 ASP.NET 中数据绑定有关的问题是维护状态。Web 表单，以及最终的 ASP.NET 页面框架，与客户端（例如浏览器）使用超文本传输协议（HTTP）进行通信。关于与 HTTP 协议绑定的数据的问题是其不能保持状态。这意味着每次刷新页面或 Web 表单时，都会丢失信息，包括控件所绑定的数据。当客户端的用户向服务器请求新信息或更多的信息时（例如当用户在填充其个人详细资料之后，单击浏览器的 [发送] 按钮时），就会刷新 Web 表单。

这种情况中出现的问题是，用户可能会输入一些无效数据，或者遗漏了某些必需的数据。然后 Web 表单需要通知用户出错，并且同时重新显示带有已输入信息的表单。因为 Web 表单在刷新时会丢失信息，所以你需要自行处理，因此必须在别处存储信息。

如何存储信息并不困难，但是选择正确的方法来完成却是一件非常棘手的事情，因为你可以有多种选择：

- 在 Web 表单的 **ViewState** 属性中存储信息。
- 如果你使用的是 **DataGrid** 控件（**System.Web.UI.WebControls** 名称空间的一部分），则可以通过将数据网格的 **EnableViewState** 属性设置为 **True**，启用网格的视图状态。
- 在 Web 表单的状态包中存储信息。状态包作为条件暗示了存储表单状态的位置。它是由页面（Web 表单）为存储服务器之间往返值进行维护的数据结构。
- 在 **Session** 和 **Application** 对象中存储信息。如果你曾经使用 Visual InterDev 开发过 Web 站点，那么这些对象对你来说可能非常熟悉。本书并不打算介绍这些对象，但是你可以在 Visual Studio.NET 中找到关于它们的大量信息。

你应该根据自己的环境进行选择，最简单的方法就是在出现性能问题时，实验各种选择。在这里，测试是一个关键词。

前面说过 Web 表单在更新时会丢失其状态，但是这句话并不十分准确。页面（Web 表

单) 的当前属性、页面自身和服务端控件的基本属性, 以及它们的状态会在往返之间自动存储。你所需要注意的是那些来提及的控件和属性部分, 例如专用变量和标准 HTML 元素的内容。

创建自己的数据绑定型 Web 表单控件

因为数据表单向导为 Web 表单始终创建带有 **DataGrid** 控件的只读表单, 所以使之成为可更新的表单, 或者使用分离的控件来代替数据网格都需要做大量的工作, 但这是建立的初始代码, 而且扩展向导所生成的代码比手工完成更为容易。

当你使用前一节中推荐的类似表单的方法时, 需要创建一个页面 UI, 但在有些情况中并不是这样。有时, 你需要一个可以从工具框拖拽到 Web 表单上的控件。在这种情况下, 就需要创建一个 Web 用户控件。你可以将新项目添加到工程中, 并以此在 Web 工程中创建一个 Web 用户控件。当出现 [Add New Item] 对话框时, 选择 [Web User Control] 模板, 为用户控件命名, 然后单击 [OK] 按钮。下面列出的是你需要做的其他操作:

1. 在设计模式下, 从工具箱中将所需的 Web 表单控件拖拽到 Web 用户控件上。
2. 创建一个绑定到控件的数据集。具体方法是将你要作为数据集的表从服务器资源管理器拖拽到 Web 用户控件上。这样就创建了一个新的连接和新的数据适配器对象, 如图 10.11 所示。这些对象被设置成指向从服务器资源管理器中拖拽的数据源, 因而不必对它们进行初始化。

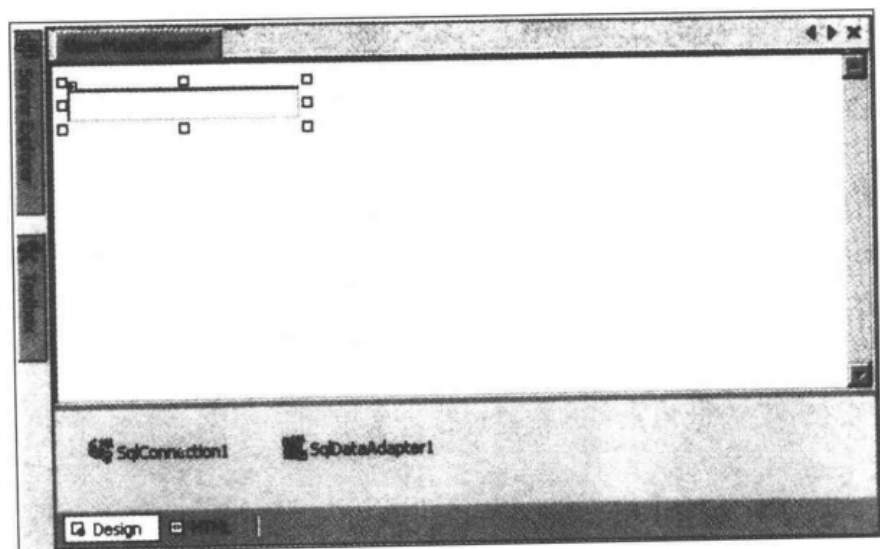


图 10.11 设计模式中带有数据对象的 Web 用户控件

3. 用数据适配器对象创建一个数据集。选中 **SqlDataAdapter1** 对象, 并用右键单击, 然后从弹出的菜单中选择 [Generate Dataset]。
4. 在 [Generate Dataset] 对话框中 (如图 10.12 所示), 选择 [New] 选项, 然后在 [New] 选项旁边的文本字段中为数据集命名, 并以此指定要创建的新数据集。
5. 将数据集 (本实例中的 **User1**) 直观地添加到数据适配器对象旁边的数据设计视图中。
6. 将你所放置在 Web 用户控件上的每个 Web 表单控件的一个或多个属性绑定到数据源类的属性上。可以通过以下步骤来完成此操作:

- 1) 在设计模式中选择 Web 用户控件。

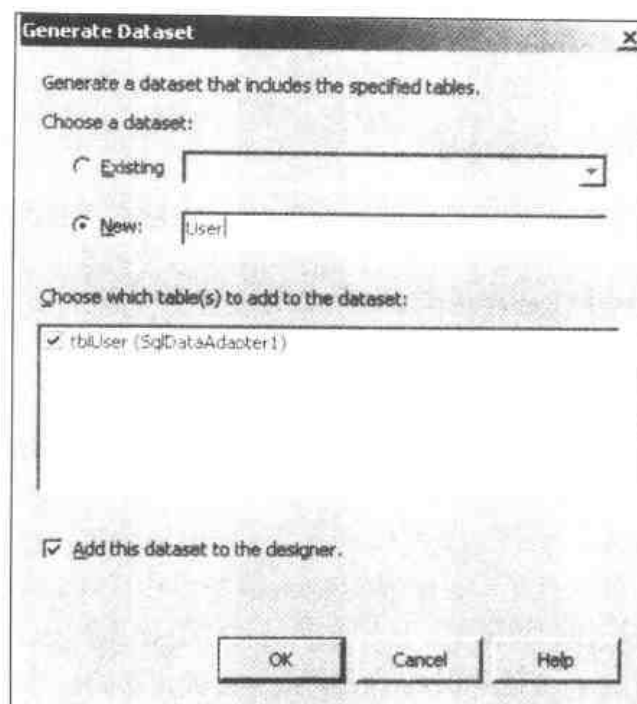


图 10.12 [Generate Dataset] 对话框

- 2) 选择 [Properties] 窗口，将指针放在 [DataBindings] 文本框中。
- 3) 单击文本框旁边带有三个圆点 (...) 的按钮，而后便会出现 [DataBindings] 对话框。
- 4) 在 Bindable Properties 树状视图中选择 [Text] 属性。
- 5) 选中 [Simple binding] 单选按钮，在 [Simple binding] 单选按钮下面的树状视图中扩展 User1 数据集。
- 6) 扩展节点，以便能够从数据集中选择 tblUser 表的 Id 列，如图 10.13 所示。然后单击 [OK] 按钮。
- 7) 将程序清单 10.5 中的代码放置在 Web 用户控件代码的 Page_Load Sub 程序中。

程序清单 10.5 在 Web 用户控件中完成数据绑定

```

' 打开连接
SqlConnection1.Open()
' 填充数据集
SqlDataAdapter1.Fill( User1, " tblUser")
' 将控件绑定于数据源
txtUserId.DataBind()

```

程序清单 10.5 中的实例非常简单，但是这确实就是你执行其他步骤之后所需做的全部工作。你现在有了一个自己的数据绑定型用户控件，它可以在任何 Web 页面上使用。

现在的问题是你是否真正想要采用这种方法。你可能还有其他方面的考虑，比如是否真的要为一个控件打开连接，何时关闭这个连接，等等。实例代码仅仅显示了它如何可以轻易地完成。到底该如何完成用户控件完全取决于你的喜好。

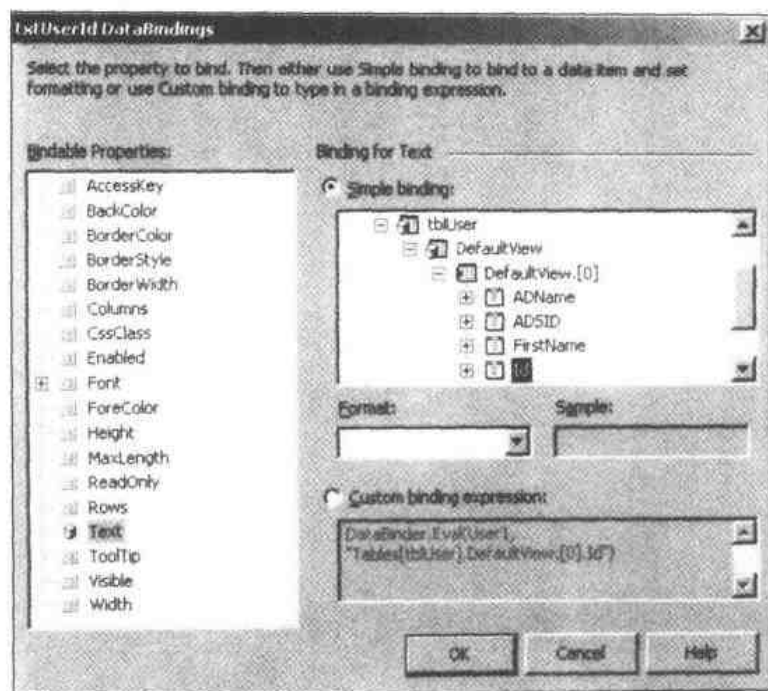


图 10.13 [DataBindings] 对话框

练 习

创建一个新的用户控件，并命名为 `UserManId`。将 `TextBox` 控件添加到用户控件中，并将其绑定到 `tblUser` 表的 `Id` 列。你可以在 Apress 的 Web 站点 (<http://www.apress.com>) 上找到如何完成该操作的示例。

10.3 小结

本章解释了什么是数据绑定型控件，如何在 Windows 表单和 Web 表单中使用它们，以及它们与 Visual Basic 以前版本中的数据绑定型控件有何区别。介绍了关于 Windows 表单中数据绑定的 `CurrencyManager` 和 `BindingContext` 类，并说明了如何将它们用于数据绑定。此外，还讨论了源自 `Control` 类的 Windows 表单控件的 `DataBindings` 属性，并介绍了如何将 Windows 表单手工绑定到数据源。

下一章将完成贯穿全书的 `UserMan` 应用程序示例，如果你想要以 `UserMan` 应用程序为基础来构建自己的应用程序，我还会就如何对该示例做进一步的改进给你一些建议和提示。

第 3 部分

应用程序示例

第 11 章

UserMan

完成应用程序示例

本章将完成 UserMan 应用程序示例，该程序的构建贯穿了本书的全部内容，假定你已经跟着学习了前面的知识并且完成了所要求的练习，如果你觉得该应用程序可以作为自己.NET 应用程序所需的构件块（building block），我还将给你一些更为深入的建议。

这个贯穿全书内容创建的应用程序示例是一个用户管理系统的程序，它可以处理来自 Active Directory（Windows 2000 中新的目录系统）的用户信息。

11.1 标识 UserMan 信息

下面先描述一下 UserMan 应用程序所要完成的任务，以及如何完成等方面的细节。UserMan 是一个用户管理系统，它执行以下任务：

- 登录系统和从系统中注销（检查 tblUser 表）
- 添加、编辑和删除用户（tblUser 表）
- 更新和检查用户的权限（tblUserRights 表）
- 记录所有用户的行为（tblLog 表）
- 添加、更新和删除用户权利和权限（tblRights 表）

这些就是应用程序 UserMan 的功能，但对于任何组织中任何类型的部署，它都需要更多的功能。

11.2 发现对象

这一节包括了应用程序 UserMan 中可以作为实际对象进行标识的内容。

11.2.1 数据库对象

数据库已经编写完成，它包括以下表格：

- tblUser
- tblUserRights
- tblRights
- tblLog

第 2 章的结尾部分介绍了该数据库的架构（图 2.10）。注意遵守第 9 章中的原则，在开发该应用程序过程中，你下一步的任务是确定如何在一个或者多个类中封装对这些表格的访问。这一步非常容易，因为通常都是考虑将每个表格封装在一个类中。虽然我曾经不得不在同一类中封装多个表格，但是在大多数情况下，都是在每个类中封装一个表格。UserMan 数据库中的表格都是明确定义的，尽管它们之间相互关联，但应该都是单独访问以添加、更新和删除行。因此，你需要在关系数据库中包括以下 4 个类：

- CUser
- CUserRights
- CRights
- CLog

11.2.2 完成 CUser 类

我们从第 9 章中开始创建 CUser 类，现在来完成这个特殊的类。类的构架看起来非常相像，可以在 Apress 站点 (<http://www.apress.com>) 的下载区中找到它们。首先，需要向其中添加一些数据库的连接性。在程序清单 11.1 中显示的是所需的专用常量和变量。

程序清单 11.1 CUser 类的专用数据库常量和变量

```

' 数据库常量
Private Const PR_STR_CONNECTION As String = "Data Source=USERMANPC;" &_
"User ID=UserMan;Password=userman;Initial Catalog=UserMan "
Private Const PR_STR_SQL_USER_SELECT As String = "SELECT *FROM tblUser "
Private Const PR_STR_SQL_USER_DELETE As String =_
"DELETE FROM tblUser WHERE Id=@Id "
Private Const PR_STR_SQL_USER_INSERT As String =_
"INSERT INTO tblUser(FirstName,LastName,LoginName>Password) " &_
"VALUES(@FirstName,@LastName,@LoginName,@Password) "
Private Const PR_STR_SQL_USER_UPDATE As String =_
"UPDATE tblUser SET FirstName=@FirstName,LastName=@LastName, " &_
"LoginName=@LoginName>Password=@Password WHERE Id=@Id "

' 数据库常量
Private Shared prshcnnUserMan As SqlConnection
Private prdadUserMan As SqlDataAdapter
Private prdstUserMan As DataSet

' 指引用户表格处理
Private prcmmUser As SqlCommand
' 数据集处理的命令对象
Private prcmmUserSelect As SqlCommand
Private prcmmUserDelete As SqlCommand

```

```
Private prcmmUserInsert As SqlCommand
Private prcmmUserUpdate As SqlCommand
' 数据集操作的参数对象
Private prprmSQLDelete, prprmSQLUpdate, prprmSQLInsert As SqlParameter
```

在程序清单 11.1 中有一些常量，一个用于连接字符串（PR_STR_CONNECTION）以及用于从用户表格（PR_STR_SQL_USER_）中选取、插入、更新和删除行的命令字符串。还有一些数据库对象，例如连接（prshcnnUserMan）、数据适配器（prdadUserMan）和数据集（prdstUserMan）等。



注意 如果需要关于这些对象的更多信息，建议你阅读第 3A 章和第 3B 章。这些章节中介绍了 ADO.NET 中的大多数类。

使连接对象共享

如果看一下连接对象，就会发现我已经用 **Shared** 访问限定语符对其进行了声明。这样设置后，连接对象在 CUser 类中的所有实例之间都可以共享，也就是说它们都将使用同一个连接。同样，这也会使连接对象始终保持激活，直至类中的最后一个实例终止。这样做的效果是好还是坏？你自然会得出结论。当你决定要修改声明或保留这样的声明时，一定要考虑到诸如可伸缩性、性能和用户总数等方面的问题。



注意 编程是一门独立的艺术，如何充分利用应用程序完全取决于你自己。对某个应用程序的好方法对另外一个应用程序来说可能就是非常糟糕的。如果你从这本书中没有学到太多知识，那么希望你至少学会了在设计和实现应用程序时该思考些什么问题。

如果你已经将这些专用变量和常量添加到 CUser 类中，则会发现 **SqlConnection**、**SqlCommand** 和其他此类的数据类型未识别出来。如果你要修正这一点，则需要将以下内容作为代码的第一行添加到 CUser 类中以导入 **System.Data.SqlClient** 名称空间：

```
Imports System.Data.SqlClient
```

事实上，这可能并不是类中的第一行代码，因为如果你的类中有 **Option...** 语句（例如 **Option Strict On**），那么这些语句就必须出现在 **Imports** 语句之前。

打开数据库连接

如果你已经添加了在 SQL Server 2000 上访问 UserMan 数据库所需的全部数据库变量，那么下一步就应该设置建立数据库连接的代码，如程序清单 11.2 所示。

程序清单 11.2 连接到数据库

```
1 Private Sub OpenDatabaseConnection()
2     Try
```

```

3      ' 检查连接是否已经实例化
4      If prshcnnUserMan Is Nothing Then
5          ' 实例化连接
6          prshcnnUserMan = New SqlConnection( PR_ STR_ CONNECTION)
7          ' 检查连接是否关闭
8          If CBool( prshcnnUserManState And ConnectionStateClosed) Then
9              ' 打开连接
10             prshcnnUserMan.Open()
11         End If
12     End If
13 Catch objSqlException As SqlException
14     ' 抛出一个连接低级异常向该类的调
15     ' 用程序添加注释并重新抛出异常
16     Throw New Exception("The connection to the UserMan database could" & _
17         " not be established, due to a connection low- level error", _
18         objSqlException)
19 Catch objE As Exception
20     ' 这里处理抛出的其他异常
21     Throw New Exception("The connection to the UserMan database " & _
22         "could not be established.", objE)
23 End Try
24 End Sub

```

在程序清单 11.2 中打开了专用 **Sub** 程序 **OpenDatabaseConnection** 中的连接。之所以将该程序设为专用，是为了让客户端无需再执行此项操作。当你实例化此类时，应该处理这个程序（参阅下一节中的程序清单 11.3）。可以肯定，在这样做之前，并没有实例化该连接，而在试图打开它之前，它会关闭。如果打开时它还没有关闭，那么就会出现一个异常。如果出现异常，则会建立一个带有如何处理的详细信息的新异常，然后再将当前的异常添加到新异常的 **InnerException** 属性中。这样，该类的调用者就可以看到最初的异常以及错误消息。

实例化类

变量的任何初始化和初始化程序所未处理的内容都应该放置到 **New Sub** 构造函数中，如程序清单 11.3 所示。

程序清单 11.3 在类实例化上打开数据库连接

```

1 Public Sub New()
2     ' Perform standard inherited instantiation (执行标准检查实例化)
3     MyBaseNew()
4     ' Open the connection to the database (打开对数据库的连接)
5     OpenDatabaseConnection()
6 End Sub

```

程序清单 11.3 展示给你的是如何确定数据库连接是打开的，并将其准备好在客户端初始化类之后使用。将一个对自定义 **OpenDatabaseConnection** 程序的调用放置在 **New Sub** 构造函数中便可完成这项操作。我已经将一个调用以 **MyBaseNew()** 形式添加到基类的构造函数中，这样就确保执行了该基类的实例化。该调用必须是你自定义析构函数中的第一行代码。

关闭数据库连接

正如你需要打开数据库连接，它也必须再度关闭，以保存连接资源。程序清单 11.4 显示了如何来完成这项操作。

程序清单 11.4 关闭数据库连接

```
1 Private Sub CloseDatabaseConnection()  
2     Try  
3         ' 关闭连接  
4         prshcnmUserManClose()  
5     Catch objE As Exception  
6         Throw New Exception("The connection to the UserMan database " & _  
7             " could not be closed properly.", objE)  
8     End Try  
9 End Sub
```

程序清单 11.4 使用了连接类的 **Close** 方法来关闭数据库连接。该方法并不确定是否会出现异常，因此异常处理器可能会“滥杀无辜”。你是最终的决定者，如果这个方法给你带去来不便，那么请将它去掉。

正如你所看到的，该程序还被设为是专用的（与 **OpenDatabaseConnection** 程序中的情况一样）。这样做的目的是：客户端不能调用该方法，它会在处理类时由代码自动来完成。

处理类

在处理类时，你必须确保连接是关闭的。将执行关闭连接的程序放置到类的 **Finalize** 方法中便可以完成此项任务（参阅程序清单 11.5）。

程序清单 11.5 类的处理

```
1 Protected Overrides Sub Finalize()  
2     ' 关闭数据库连接  
3     CloseDatabaseConnection()  
4     MyBaseFinalize()  
5 End Sub
```

在程序清单 11.5 中，从类的 **Finalize** 方法中关闭了数据库连接。当处理一个对象时（例如当客户端将对象设置为 **Nothing** 时），会调用 **Finalize** 方法。你需要明确地关闭数据库连接，否则它将会保持打开状态，且不会返回到池中。



在第 3A 章中有关于连接和连接池的更多内容。

实例化命令对象

用于处理数据源和一般命令对象的数据适配器所使用的命令对象需要实例化，如程序清单 11.6 中所示。

程序清单 11.6 实例化命令对象

```

1 Private Sub InstantiateCommands()
2     ' 实例化适配器数据适配器命令对象
3     prcmUserSelect = New SqlCommand( PR_STR_SQL_USER_SELECT, prshcnnUserMan)
4     prcmUserDelete = New SqlCommand( PR_STR_SQL_USER_DELETE, prshcnnUserMan)
5     prcmUserInsert = New SqlCommand( PR_STR_SQL_USER_INSERT, prshcnnUserMan)
6     prcmUserUpdate = New SqlCommand( PR_STR_SQL_USER_UPDATE, _
7         prshcnnUserMan)
8     ' 实例化并初始化一般命令对象
9     prcmUser = New SqlCommand()
10    prcmUserConnection = prshcnnUserMan
11 End Sub

```

在程序清单 11.6 中，对数据适配器（用于选择、添加、更新和删除数据源中的行）所使用的命令对象进行了实例化。然后，对可以直接用于数据源的一般命令对象也进行了实例化。例如，可以使用命令对象的 **ExecuteReader** 方法，从数据阅读程序中仅检索表格的某些特定列。这里并没有指定命令文本，因为执行命令时会完成这步操作，但是将共享连接指定为命令的连接，因为该命令中的所有查询都将使用该连接。进行打开数据库连接的调用之后，应该将 **InstantiateCommands** 程序添加到程序清单 11.3 中的 **New** 构造函数中。

DataSet 对象的实例化

如同大多数对象一样，**DataSet** 对象在使用之前需要实例化。程序清单 11.7 显示了如何完成该操作。

程序清单 11.7 Data Set 的实例化

```

1 Private Sub InstantiateDataSet()
2     prdstUserMan = New DataSet()
3 End Sub

```

正如你在程序清单 11.7 中所见到的那样，**InstantiateDataSet** 程序中的代码并不多。你并不需要更多的代码，同样可以考虑从程序中取出一行代码，并将它放置到你通常放置程序调用的位置。无论如何，实例化代码都应该放置在 **New** 构造函数中。

实例化和初始化数据适配器

上面向你展示的是实例化数据集，但是数据集需要使用数据适配器来从数据源填充。因此，接下来继续实例化和初始化数据适配器，如程序清单 11.8 所示。

程序清单 11.8 实例化和初始化数据适配器

```

1 Private Sub InstantiateAndInitializeDataAdapter()
2     prdadUserMan = New SqlDataAdapter()
3     prdadUserMan.SelectCommand = prcmUserSelect
4     prdadUserMan.InsertCommand = prcmUserInsert
5     prdadUserMan.DeleteCommand = prcmUserDelete
6     prdadUserMan.UpdateCommand = prcmUserUpdate

```

```
7 End Sub
```

在程序清单 11.8 中, 通过将数据适配器和命令属性设置到已实例化的命令对象, 可以实例化数据适配器并初始化命令属性。

添加命令对象参数

第 3A 章和第 3B 章讲解了如何将参数添加到命令对象中, 以使得数据适配器确切知道如何查询数据源。程序清单 11.9 显示的是如何对 UserMan 执行该任务。

程序清单 11.9 添加命令对象参数

```
1 Private Sub AddCommandObjectParameters()  
2     ' 添加删除命令参数  
3     prcmmUserDeleteParametersAdd("@ FirstName", SqlDbTypeVarChar, 50, _  
4         " FirstName")  
5     prcmmUserDeleteParametersAdd("@ LastName", SqlDbTypeVarChar, 50, _  
6         " LastName")  
7     prcmmUserDeleteParametersAdd("@ LoginName", SqlDbTypeVarChar, 50, _  
8         " LoginName")  
9     prcmmUserDeleteParametersAdd("@ Password", SqlDbTypeVarChar, 50, _  
10        " Password")  
11    prprmSQLDelete = prdadUserManDeleteCommandParametersAdd("@ Id", _  
12        SqlDbTypeInt, 0, " Id")  
13    prprmSQLDeleteDirection = ParameterDirectionInput  
14    prprmSQLDeleteSourceVersion = DataRowVersionOriginal  
15  
16    ' 添加更新命令参数  
17    prcmmUserUpdateParametersAdd("@ FirstName", SqlDbTypeVarChar, 50, _  
18        " FirstName")  
19    prcmmUserUpdateParametersAdd("@ LastName", SqlDbTypeVarChar, 50, _  
20        " LastName")  
21    prcmmUserUpdateParametersAdd("@ LoginName", SqlDbTypeVarChar, 50, _  
22        " LoginName")  
23    prcmmUserUpdateParametersAdd("@ Password", SqlDbTypeVarChar, 50, _  
24        " Password")  
25    prprmSQLUpdate = prdadUserManUpdateCommandParametersAdd("@ Id", _  
26        SqlDbTypeInt, 0, " Id")  
27    prprmSQLUpdateDirection = ParameterDirectionInput  
28    prprmSQLUpdateSourceVersion = DataRowVersionOriginal  
29  
30    ' 添加插入命令参数  
31    prcmmUserInsertParametersAdd("@ FirstName", SqlDbTypeVarChar, 50, _  
32        " FirstName")  
33    prcmmUserInsertParametersAdd("@ LastName", SqlDbTypeVarChar, 50, _  
34        " LastName")  
35    prcmmUserInsertParametersAdd("@ LoginName", SqlDbTypeVarChar, 50, _  
36        " LoginName")  
37    prcmmUserInsertParametersAdd("@ Password", SqlDbTypeVarChar, 50, _  
38        " Password")
```

```
39 End Sub
```

在程序清单 11.9 中添加了处理数据源到命令对象的数据适配器所需的参数。应将该调用添加到 **New** 构造函数中。

填充数据集

现在万事俱备，你需要做的事情就是用数据源的数据来填充数据集。完成此项任务的实例代码请参阅程序清单 11.10。

程序清单 11.10 用数据源的数据来填充数据集

```
1 Private Sub PopulateDataSet()
2   Try
3     prdadUserManFill( prdstUserMan, " tblUser")
4     Catch objSystemException As SystemException
5       Throw New Exception("The dataset could not be populated, " & _
6         " because the source table was invalid.", objSystemException)
7     Catch objE As Exception
8       Throw New Exception(" The dataset could not be populated.", objE)
9   End Try
10 End Sub
```

程序清单 11.10 非常简单，这里尝试用数据源中 **tblUser** 表里的数据来填充数据集。如果出现异常，将用原始异常来重新显示异常。该方法也将添加到 **New** 构造函数中。

引入数据集的公共属性

既然数据集已经填充完，现在公共属性可以引入到数据集中了。所有属性读取并设置专用变量，因此你需要从填充的数据集中读取值，并在相应的专用变量中存储它们，如程序清单 11.11 所示。

程序清单 11.11 保存数据集的值

```
1 Private Sub SaveDataSetValues()
2   ' 保存用户 id
3   prlngId = CType( prdstUserManTables(" tbluser")Rows( 0)Item(" Id"), Long)
4   ' 检查 ADName 是否为 Null
5   If prdstUserManTables(" tbluser")Rows( 0)IsNull(" ADName") Then
6     prstrADName = ""
7   Else
8     prstrADName = _
9       CType( prdstUserManTables(" tbluser")Rows( 0)Item(" ADName"), _
10         String)
11   End If
12   ' 检查 ADSID 是否为 Null
13   If Not prdstUserManTables(" tbluser")Rows( 0)IsNull(" ADSID") Then
14     prguiADSID = _
15       CType( prdstUserManTables(" tbluser")Rows( 0)Item(" ADSID"), Guid)
```

```

16 End If
17 ' 检查名是否为 Null
18 If prdstUserManTables(" tbluser")Rows( 0)IsNull(" FirstName") Then
19     prstrFirstName = ""
20 Else
21     prstrFirstName = _
22     CType( prdstUserManTables("tbluser")Rows( 0)Item("FirstName"), _
23     String)
24 End If
25     检查姓是否为 Null
26 If prdstUserManTables(" tbluser")Rows( 0)IsNull(" LastName") Then
27     prstrLastName = ""
28 Else
29     prstrLastName = CType( prdstUserManTables(" tbluser")Rows( 0)Item( _
30     " LastName"), String)
31 End If
32 ' 检查登陆名是否为 Null
33 If prdstUserManTables(" tbluser")Rows( 0)IsNull(" LoginName") Then
34     prstrLoginName = ""
35 Else
36     prstrLoginName = CType( prdstUserManTables(" tbluser")Rows( 0)Item( _
37     " LoginName"), String)
38 End If
39 ' 检查密码是否为 Null
40 If prdstUserManTables(" tbluser")Rows( 0)IsNull(" Password") Then
41     prstrPassword = ""
42 Else
43     prstrPassword = CType( prdstUserManTables(" tbluser")Rows( 0)Item( _
44     " Password"), String)
45 End If
46 End Sub

```

在程序清单 11.11 中，仅从数据集中读取值，并把它们存储到相应的专用变量中。请注意，是从数据集中的第一行读取的。这样做的目的是为了便于对以后所添加的数据进行排序和过滤，而以前添加的对数据库中列的空值检查使这个任务得以完成。

指定父类

你是否想知道自己的 CUser 类衍生于哪个类？如果你一直都是跟随示例做的，那么至此应该还没有加入 **Inherits** 语句（指定类从何处继承）。对于应用程序 UserMan，你不必这样做，因为你应该从 **Object** 类继承，而这是隐式完成的。这意味着你不必将 **Inherits Object** 语句添加到类中。但是，如果你想从其他的类继承，则需要将 **Inherits** 语句添加到类中，并作为类定义（**Public Class CUser**）后面的第一行代码。



在第 9 章开始创建 CUser 类的地方讨论了这些话题。

还需要什么操作？

显然，CUser 类还没有完成，尽管你已经向其中添加了许多代码。现在所需要做的是添加允许对数据集中的行进行过滤和排序的代码，更为重要的是，添加用于更新数据源的代码。



我已经将该代码和其他程序添加到 CUser 类中，这些程序和代码可以在 Apress 的 Web 站点 (<http://www.apress.com>) 的下载区中找到。

11.2.3 Active Directory 对象

从 tblUser 表格的 ADSID 和 ADName 列中读取值需要用过对象。程序清单 11.12 显示的是用于这项任务的示例代码。



第 7 章介绍了关于 Active Directory 访问的更多内容。

程序清单 11.12 Active Directory 类

```

1 Imports SystemDataOleDb
2
3 Public Class CActiveDirectory
4     ' 数据库对象
5     Private prcnnAD As OleDbConnection
6     Private prcmmAD As OleDbCommand
7     Private prdrdAD As OleDbDataReader
8
9     Private Sub OpenConnection()
10    ' 实例化并打开连接
11    prcnnAD = New OleDbConnection(" Provider= ADsDSOObject;" & _
12    " User Id= UserMan; Password= userman")
13    prcnnAD.Open()
14    End Sub
15
16    Private Sub RetrieveUserInformation()
17    ' 实例化命令
18    prcmmAD = New OleDbCommand(" SELECT cn, adsPath FROM " & _
19    "'LDAP:// usermancom' WHERE objectCategory='person' AND "& _
20    " objectClass=' user' AND cn=' UserMan'", prcnnAD)
21    ' 在数据阅读器中检索 UserMan 信息
22    prdrdAD = prcmmAD.ExecuteReader()
23    End Sub
24 End Class

```

正如你从程序清单 11.12 中看到的那样，我使用 OLE DB NET 数据提供程序从 Active Directory 中检索用户信息。这个数据提供程序是只读的，但由于我们只检索信息，因此可以使用这个程序。这个类远未完成，还需要检索正确的值 (ADName 和 ADSID) 并在专用变量

中存储检索值。此外，你还需要设置只读属性，只有如此，使用该类的客户才可以阅读这些值。客户端可以是 CUser 类。

此外，你还需要创建可以提取要检索信息的用户名字的属性和/或方法，而不是对用户的名字进行硬编码。完成了上述工作后，就可以读取任何用户的信息了。



在 Apress 的 Web 站点 (<http://www.apress.com>) 的下载区中可以找到完整的 CActiveDirectory 类。

11.2.4 其他对象

因为仍没有将所有的数据访问封装起来，所以还需要创建更多的类。比如：

- 消息排队
- 数据绑定型控件

在 Apress 的 Web 站点 (<http://www.apress.com>) 的下载区中可以找到这些类的实现。

作为组件来封装类

当你创建完所有类时，需要决定其中的一个或多个类是否要封装为一个组件，以满足在服务器上部署的需要。这涉及组件构建的内容，已经超出了本书所讲的内容。为了满足你的需要，我已经将所有的表格类封装到一个组件中，你可以在 Apress 的 Web 站点 (<http://www.apress.com>) 的下载区中找到它。在大多数情况中，应用程序的体系结构决定了是否将类封装到一个组件中。

11.3 创建客户端

现在，数据库已经连接成一个整体，你也了解了如何为本书中涉及的一些数据相关访问创建封装类。因此，下面将介绍如何创建客户端应用程序，包括如何创建基于 Windows 形式的 Windows 客户端和基于 Web 形式的 Web 客户端，因为它们都是在本书中所讲到的 UI。

11.3.1 创建 Windows 客户端

在创建访问应用程序 UserMan 中的数据 Windows 客户端方面，并没有太多要介绍的内容。因此，本章中省略了 Windows 客户端的示例代码。你可以在 Apress 的 Web 站点的下载区中找到 UserMan Windows 客户端的完整示例。代码中带有大量注释，可以帮助你理解该示例。

11.3.2 创建 Web 客户端

从 Web 客户端的现状看，本章还无法介绍关于 Web 客户端的示例代码，但是可以在

Apress 的 Web 站点 (<http://www.apress.com>) 的下载区中找到 UserMan Web 客户端的完整版本。代码借助注释进行了清楚地归档。

11.4 提示和建议

本节将就进一步开发 UserMan 应用程序给你一些提示和建议,以便你日后可以自如地使用它。下面将这些建议进行了分组,从而使你就可以轻松地查阅。

11.4.1 数据库建议

你将在本节找到所有与改进和/或增强数据库方面有关的建议。

User 表格中的 Password 列

你或许已经注意到, `tblUser` 表格中的 `Password` 列是 `varchar(50)` 列,具有读取访问权限的人都可以读取它。这就意味着具有读取访问权限的用户可以读取任何用户的密码,包括管理员 (UserMan) 的密码。这显然是不正常的。一些第三方供应商在生产加密组件,但在编写本书时,还没有见到过用于 NET 框架的第三方加密组件。不过,在你学习本书时市面上也许就出现这种组件了,或者你可以通过 COM Interop 来使用 COM Component (第 3B 章中曾经介绍过 COM Interop)。

另一个选择是创建自己的密码加密方案。如果你觉得网络访问非常安全,那么随便编写一个小程序就足够用了。因为我在这方面并不擅长,所以无法为你实现一个加密程序或告诉你应如何使用它。

记录事件日志

你可以在 Windows NT 事件日志中记录事件,而不必在 UserMan 数据库的 `tblLog` 表格中记录。这样,所有能够读取网络上事件日志的用户便都可以读取这些事件了。查看 `SystemDiagnostics` 名称空间中的 `EventLog` 类,你可以获取关于如何读取和写入事件日志的更多信息。

将连接对象传递到各个类

你应该考虑将一个开放连接传递到所有类上,以供这些类使用,而不是为每个类创建一个连接。这样,你就可以在类之间共享连接。这取决于你如何创建的应用程序、应用程序如何进行部署,以及在一条连接上你所期望的通信流量。

如果你要实现上述的这种增强效果,那么所要考虑的最后一件事情就是,是否要包括一个数据阅读器,当其打开时要单独使用连接。因此,要使用阅读器的话,显然就不要选择传递连接对象了。

为访问数据库表创建存储程序

一旦类实现后,就可以直接访问数据。但是,你可以改变类的实现来使用 SP (存储程序)。在没有注意性能如何之前,你可能并不想这样做,但是日后你或许要通过使用 SP 来提高性

能。对应用程序中的数据库访问使用类的一点好处是，只要你保持接口完整，就可以改变类的实现，而客户端应用程序不会检测到任何改变。

设置触发器以实施商务规则

在第 6 章中，曾经介绍过如何实现简单的触发器，例如强制包含用户的姓和名，如果你提供了任一者的话。这就是可以由触发器来处理的许多简单任务之一，有了触发器你就不必再将这个功能放置在类中，甚至放置在客户端代码中。

设置数据库安全性

在实现数据库级别的安全性方面，我没有做任何设置，这意味着对你的网络和 SQL 服务器有访问权限的任何用户都可以访问 UserMan 数据库。如果你还没有做任何安全性设置的话，那么建议你首先要添加一些安全性。如果要获得关于如何在数据库级别上实现安全性的更多信息，请向系统管理员咨询或者查阅 SQL Server 的联机帮助。

对表格和列的名称使用常量

用常量替换所有硬性编码的表格和列名称（例如 `tblUser` 和 `Id`）。这样将使你的代码更容易更新和维护，而且我个人认为这样可以使你的代码更容易阅读。

使用本地事务处理

对数据库写入操作使用事务处理是一个好方法。这在 UserMan 的类中并没有实现，强烈建议你考虑添加本地事务处理支持。



注意。

第 3A 章讨论了本地事务处理，也就是使用连接对象显式开启和关闭的事务

11.4.2 常规建议

本节包含的意见和建议在别处可能并不适用。

让 Active Directory 来验证用户

为什么要用自己的检查程序检查，而不让 Active Directory 来检查呢？当然，这要求应用程序的所有用户都存储在 Active Directory 中。在这个前提下将 Active Directory 类扩展允许写入和读取即可。

使用 Web 服务来完成一些功能

你可以通过 Web 服务来完成某些应用程序功能。本书并不详细介绍 Web 服务，但你主要是将 Web 服务作为 Web 服务器程序的方法来使用，利用 HTTP 上的对象访问协议 (SOAP) 通过 Internet、intranet 或者任何网络来完成一定的功能。这保证你可以通过防火墙使用该功能。

异常处理

两个客户端应用程序只有很少的异常处理，这显然是不够的。查阅代码，在你认为需要的地方放置异常处理程序。根据自己喜好，可以使用结构化异常处理或非结构化异常处理。查看各个类的位置后再添加异常处理是一个不错的方法！

使用自动事务处理

所有的 .NET Framework 类都可以是自动事务处理的一部分。这意味着你可以返回（roll back）或者提交类中所做的修改，就像你可以对数据库所做的操作一样。你所做的一切操作都是为了使类进行事务处理。关于如何操作的更多内容，请查阅归档文件。

11.5 小结

本章完成了一个应用程序示例，而本书的其他章节又都是构建在此示例之上的。在这一章介绍了如何使用与数据相关的工具以及/或前面章节中所介绍的方法（例如 Active Directory 访问和 SQL Server 访问）来构建并完成该应用程序示例。

随后，就如何进一步使用 UserMan 应用程序和定制自己的程序给出了一些建议和提示。

本节是本书最后一章的最后一部分内容。衷心希望你喜欢阅读本书，就像本人喜欢编写它一样。如果你对本书有什么问题或者建议，欢迎你来信。你可以通过 dbpwvbnnet@vb-joker.com 与我联系。如果你对这个应用程序示例有什么改进，且愿意与本书的其他读者一起分享，也非常欢迎来信进行探讨。

目 录

序 言	
内容简介	
作者简介	
技术评审简介	
致 谢	

第 1 部分 入 门

第 1 章 VB.NET 快速入门	3
1.1 回顾编程概念	3
1.2 获得合适的.NET 集成开发环境	9
1.3 小结	18

第 2 部分 数据库编程

第 2 章 与数据库对话	21
2.1 数据库究竟是什么?	21
2.2 为什么使用数据库?	21
2.3 数据库管理系统	22
2.4 行和记录	23
2.5 列和字段	23
2.6 关系型和层次型	23
2.7 UserMan 数据库架构	33
2.8 小结	34

第 3A 章 ADO.NET 介绍: 连接层	35
3A.1 数据关联的名称空间	36
3A.2 提供程序	37

3A.3 使用命令对象	69
3A.4 使用 DataReader 类	79
3A.5 DataAdapter 说明	92
3A.6 小结	99

第 3B 章 ADO.NET 介绍: 非连接层	100
3B.1 使用 DataSet 类	100
3B.2 使用 DataTable 类	118
3B.3 使用 DataView 类	127
3B.4 使用 DataRow 类	130
3B.5 指针	133
3B.6 COM Interop	135
3B.7 小结	137

第 4 章 以数据库观点介绍 IDE	138
4.1 使用服务器资源管理器	138
4.2 数据库工程一览	146
4.3 使用 Database Designer 设计 数据库	153
4.4 使用 Table Designer	158
4.5 使用 Query Designer 设计查询	162
4.6 使用 SQL Editor 编辑脚本	169
4.7 创建类型数据集市	171
4.8 小结	173

第 5 章 错误处理	175
5.1 结构化异常处理	175

5.2 结构化异常的 CLR 处理	187	8.6 由系统生成的队列	266
5.3 未结构化异常处理	188	8.7 保护消息队列的安全	268
5.4 小结	198	8.8 小结	277
第 6 章 存储过程、视图与触发器的使用	199	第 9 章 数据包装	278
6.1 优化因素	199	9.1 为什么要使用数据包装?	278
6.2 存储过程的使用	201	9.2 关于面向对象编程	278
6.3 使用视图	212	9.3 将数据库打包	284
6.4 使用触发器	218	9.4 小结	287
6.5 小结	223	第 10 章 数据绑定型控件	288
第 7 章 分层数据库	224	10.1 数据绑定型控件与手工 数据引入	288
7.1 LDAP 一览	224	10.2 用于不同 UI 的不同控件	289
7.2 以编程方式访问 AD	225	10.3 小结	305
7.3 使用 OLE DB .NET 数据提供程序 访问 Active Directory	234		
7.4 小结	237		
第 8 章 消息队列	239	第 3 部分 应用程序示例	
8.1 无连接程序设计	240	第 11 章 UserMan	309
8.2 MessageQueue 类概览	240	11.1 标识 UserMan 信息	309
8.3 何时使用消息队列	240	11.2 发现对象	309
8.4 如何使用消息队列	242	11.3 创建客户端	319
8.5 创建事务型消息队列	262	11.4 提示和建议	320
		11.5 小结	322

.NET 开发系列丛书

Database Programming with Visual Basic.NET

VB.NET

数据库编程

[美] Carsten Thomsen 著
常晓波 译

a!
Apress®



中国电力出版社

责任编辑 / 朱恩从

封面设计 / 肖 广

等

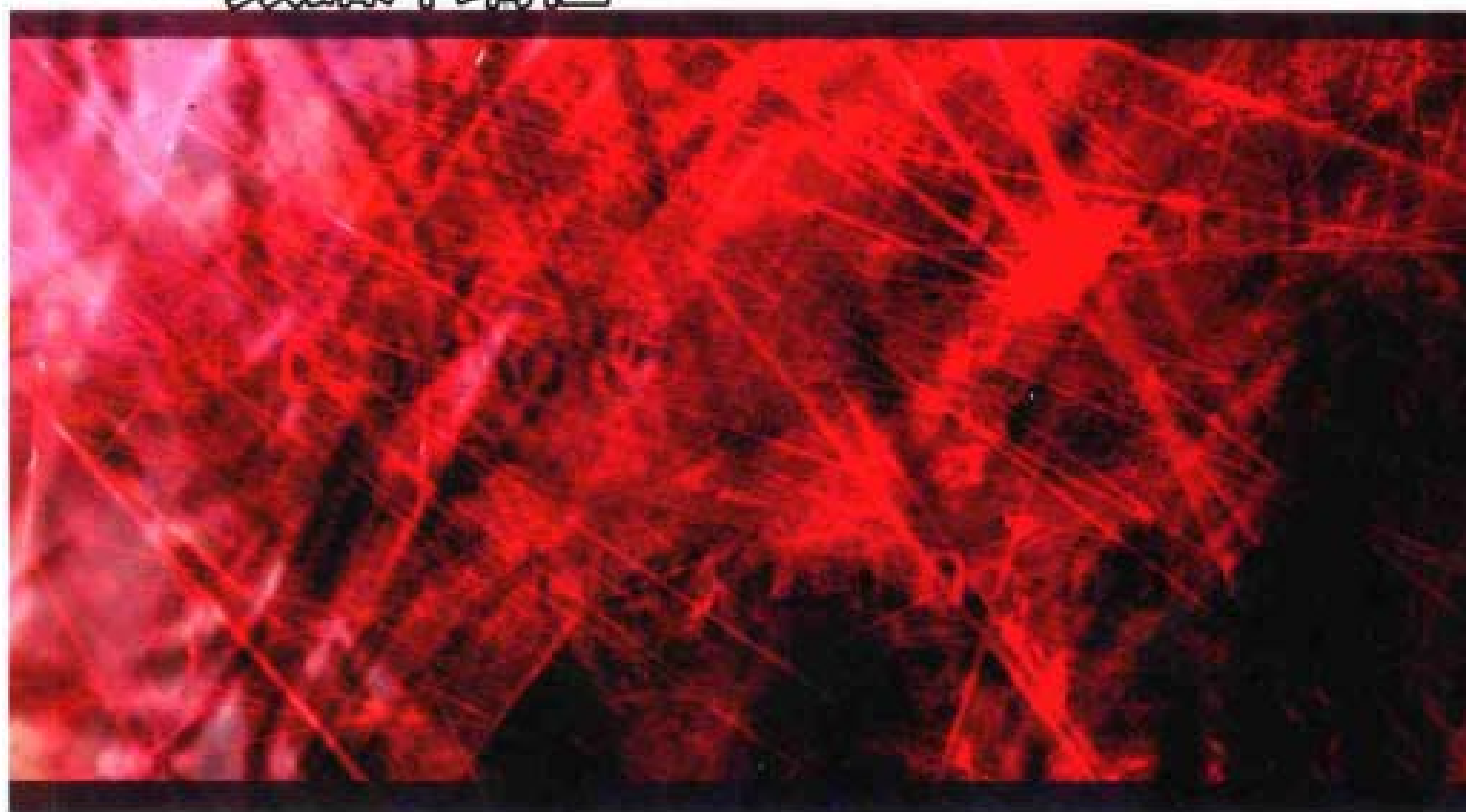
— 74 —

VB.NET

数据库编程

本书全面介绍了ADO.NET, 包括活动目录和LDAP的访问, 及数百个代码清单和表格。所有代码均已测试通过, 确保了与V.NET最终版本的完全兼容性。本书由Microsoft VB的MVP编写, 由Mark Dunn进行了严格的技术审查。

在本书中, Microsoft VB的MVP作者Carsten Thomsen使用V.NET中的范例代码向开发人员展示了Visual Studio.NET中有关数据库访问的方方面面的内容。此外, 他还介绍了如何创建不同的数据库, 如关系数据库、表格、约束条件和类型数据集, 并解释了Visual Studio.NET的集成开发环境(IDE)中使用ADO.NET的方和时机, 而不仅仅是单纯地介绍编程方法。

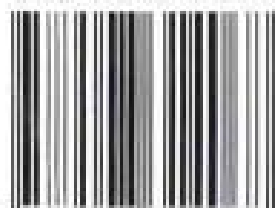


本书介绍:

- 数据库基础知识的介绍
- 完整用户管理系统的构建
- 结构化异常处理的分析
- 不同版本VB中的非结构化异常处理的实现

Carsten Thomsen 拥有 MCSE 和 MCSD 的 Microsoft 认证。此外, 他还是 VB 的 Microsoft MVP, 这是他在 1999 年 1 月得到的褒奖。Carsten 从 VB3.0 发行后就一直使用 VB 进行编程, 且对各种数据存储的访问都有专门的研究。

ISBN 7-5083-1406-9



9 787508 314068 >

ISBN 7-5083-1406-9

定价: 32.00 元

.NET 开发系列丛书

Database Programming with Visual Basic.NET

VB.NET

数据库编程

[美]Carsten Thomsen 著
常晓波 译

中国电力出版社

内 容 提 要

全书共分为三个部分,依次讲述了 VB.NET 和 ADO.NET 的工作原理讲起,介绍了有关数据库访问、错误处理、存储过程、视图、触发器、消息队列、数据外壳,以及数据绑定控件的知识,并完整地实现了一个名为 UserMan 的应用程序。

通过阅读本书,你将学会如何使用由 ADO.NET 产生的类访问数据库,如何使用存储过程、视图和触发器,如何得到 Active Directory 中的信息以及如何在自己的应用程序中使用 Message Queuing。

本书适合于那些对 Visual Studio.NET 和 Visual Basic 有一定了解的中级用户阅读。

本书英文版原名: Database Programming with Visual Basic.NET

Published by arrangement with APRESS

本书由 APRESS 公司授权出版。

北京市版权局著作权合同登记号 图字: 01-2002-6208 号

图书在版编目(CIP)数据

VB.NET 数据库编程 / (美)卡斯頓著;常晓波译. —北京:中国电力出版社,2003

ISBN 7-5083-1406-9

I.V... II.①卡...②常... III.BASIC 语言—程序

中国版本图书馆 CIP 数据核字(2003)第 004553 号

责任编辑:朱恩从

书 名:VB.NET 数据库编程

编 著:(美)卡斯頓

翻 译:常晓波

出版发行:中国电力出版社

地址:北京市三里河路6号 邮政编码:100044

电话:(010) 88515918 传真:(010) 88518169

印 刷:汇鑫印务有限公司

开 本:787×1092 1/16

印 张:21

字 数:475千字

版 次:2003年5月北京第一版

印 次:2003年5月第一次印刷

定 价:32.00 元

序 言

听说最新的消息了吗？范例发生了转变！随着 Microsoft 的 Visual Studio.NET 和 .NET 框架的发行，Microsoft 彻底改变了我们编写基于 Web 和多层（n-tier）应用程序的方式！

.NET 框架展示了许多超前的概念（专为编写数据库应用程序的开发者而用）。不用再使用古老的基于 COM 的 ADO Recordset 类了，使用 ADO.NET 的非连接 DataSet 和 DataTable 类（此处“非连接”为关键词）即可完成任务。

在为本书做序之前，我希望该书能够达到 Microsoft 帮助文件的标准。实际上，它并没有令我失望！该书集中讲述了 ADO.NET 的技术内核，以及它们与 ADO 不同的地方。

这本书不仅可以用作 ADO.NET 程序员的指导手册，还完全可以作为 VB.NET 数据库访问的语言参考。

Carsten，你做得真棒！

Carl Prothman

Issaquah，华盛顿

<http://www.able-consulting.com>

内 容 简 介

本书讲述的是如何访问各种类型的数据库，如 Active Directory、SQL Server 2000 和 Message Queuing。本书的意图是让读者了解 ADO.NET 的工作原理，学习如何使用 ADO.NET 产生的类访问数据库，如何使用存储过程、视图和触发器，如何得到 Active Directory 中的信息以及如何在自己的应用程序中使用 Message Queuing 等。作者最关心的是本书是否简单易读，虽然书中的某些段落不像期望的那么易读，但作者已经尽了自己的最大努力。

本书对象是中级用户，也就是那些对 Visual Studio.NET 和 Visual Basic 先前版本有一定了解的用户。另外，读者还应具备 Object Oriented Programming (面向对象的程序设计，OOP)、ADO 和数据库设计的一些基本知识。本书中部分章节适合初学者阅读，而另外一些章节则适合水平更高的读者阅读。适合初学者水平的内容都放在适当的地方，以使读者能真正掌握。

作者使用的是 Visual Studio.NET 企业版中的某些功能，而读者则需要两个企业版本以进行所有的练习。但是，专业版本就可用于大多数示例代码，如果读者在学习 ADO.NET，而且只是想看一下其中的操作都是如何完成的，此版本也足够了。这说明企业版在数据库访问方面惟一多做的事情，就是提供了一套额外的数据库工具，它可以从 IDE 内部使用。

本书结构

本书分为三部分：

- 第 1 部分是对 Visual Studio.NET 和 .NET 框架的概述。
- 第 2 部分的内容非常丰富，在这部分中将介绍如何连接到关系数据库和分级数据库，如何将数据库访问封装为类，以及如何掌控异常处理等。第二部分以介绍如何设计关系型数据库作为开始。
- 第 3 部分具体完成了 UserMan 应用程序示例。

代码示例

本书使用的所有代码示例都可以在 Apress 网站(<http://www.apress.com>)的下载区中找到。

数据源

本书中示例的数据源都运行在 SQL Server 2000 上。在购买 Visual Studio.NET 时你会得到一个 SQL Server 2000 的试用版本，所以，即使你当前没有 SQL Server 2000，也应该能够

运行这些示例。而且，这些代码示例也很容易修改，可以运行于 MS Access 或 SQL Server 7.0 上。你需要注意的只有一点：SQL Server 2000 能够以 XML 的形式返回行集，而 MS Access 或 SQL Server 7.0 则无法做到这一点。SQL Server 7.0 不支持参考完整性(referential integrity)，这意味着其必须使用触发器才能实现。

反馈

读者可以通过 dbpwvbnnet@vb-joker.com 与作者联系，作者将尽力回答读者关于此书的各种问题。

作者为 UserMan 应用程序示例建立了一个网站，读者可以发表和检索关于如何进一步开发该示例的想法。该网站的地址是 <http://www.userman.dk>。

作者简介

Carsten Thomsen: 丹麦人, 拥有 MCSE 和 MCSD 的 Microsoft 认证。他还是 Visual Basic 的 Microsoft MVP, 这是他在 1999 年 8 月得到的褒奖。Carsten 从 VB 3.0 发行后就一直用 VB 编程, (尽管他也试过用其他语言编程, 但最终选择的还是 VB), 并专攻对各种数据存储的访问。Carsten 与其女友 Mia 以及他们两岁的女儿 Caroline 都住在丹麦西海岸的埃斯比约 (Esbjerg)。Carsten 的另一个女儿 Nicole 则与她的母亲一起住在都柏林。Carsten 承认在计算机上花费了太多时间, 但是也很喜欢和 Mia 与其女儿一起共渡的闲暇时光。“只要有可能, 我们就会在南欧或北非呆上一星期度假”。

Carsten 的信箱地址为 carstent@vb-joker.com。

技术评审简介

Mark Dunn: MCT, MCSD, MCDBA, 他是 Extreme Logic 公司 (<http://www.extremelogic.com>) 的高级顾问和技术讲师。他拥有数学和物理学位, 居住在佐治亚州的亚特兰大城。

Mark 在转为培训之前做了很多年的顾问, 他承认目前自己 90% 的时间都用来教授课程 (主要是 VB 和 SQL Server)。过去他教过 Delphi、Java、JavaScript、Oracle 和 Power Builder。他使用 VB 的历史可以追溯到 VB 1.0 版本以及之前 Microsoft's 的 BASIC PDS。他为 Tapscan 程序编写了主要模块 (该软件现在还用于为各州电台收听率分析); 此外, 他还为 Tapscan 编写了报告生成引擎和地理映射模块。在当顾问的那些年里, 他做了大量客户/服务器应用程序咨询, 主要解决了 VB 前端连接到 Oracle 后端的问题。

Mark 也承认自己“为计算机消得人憔悴”, 并尽力平衡了工作和家庭之间的矛盾。Mark 的婚姻已有 15 年历史, 有一个 8 岁的儿子 (Mark Jr.) 和一个 6 岁的女儿 (Erin)。Mark Jr. 对棒球非常着迷, 而 Erin 很喜欢体操。

Mark 的信箱地址为 markdunn@mindspring.com。

致 谢

这本书对于我而言可以说是梦想成真，我终于自己编写了一本书，没有任何合著者（对于我来说这可能有些难堪，因为本人“仅仅”33岁）。我真的为自己感到骄傲，但是编写它确实需要付出代价。在测试阶段，我花费了很多时间去研究相关技术，尽可能地对它做各种各样的变化。在着手写这本书的时候，我知道会发生很多种变化，但是从不奢求 Microsoft 在 Beta 1 到 Beta 2 也会做这么多变化。不管怎样，编写此书仅仅“花费”了额外的几个月时间。而且，Microsoft 中有些人也在尽力减轻我的痛苦。Steve Ling——他负责处理.NET 归档以及耐心地解答我那些古怪的问题。Steve，我真的很感谢你，感谢你快速、精确和坦率的回答。

我还想感谢相信我（一个未发表过作品的作者）能够编写这本书的出版商。此书本来在几个月之前就可以出版了，我也可以据此责怪其他人，但是我还是很安心，因为即使出书的计划一直在“延续”，Apress 还是在一直支持着我。非常感谢你们为我提供了写书的方法、正确的信息，而且不要求我准时。我知道有些出版商会认为这很成问题，所以我想应该庆幸自己选择了 Apress。

我不想为出版商做鉴定，但也确实有很多关于 Apress 的好处要说。在你们提出之前，我不会去寻找新合同，因为我已经有了，它就在那里！

出版商，尤其是参与本书工作的人们我都需要特别的感谢，因为他们使我或多或少地以自己的方式写出了此书。我还从技术审稿人 Mark Dunn 那里得到了很多帮助，他不但发现了即使最棒的作者在写书时也会犯的致命错误，而且还提出了很多好想法，为更好地安排此书和添加必要的内容提供了帮助。在 Mark 无法帮助我时，我只好打扰以前的一个同事，Michael Thomhav，他对此书的部分章节做了很好的审阅工作。Michael，同样也要感谢你，感谢你那么快就投入了工作。Karen Watterson，是本书的编辑。与 Karen 合作非常愉快，我们通过一些很有趣的电子邮件进行联系，所以我想说的是：感谢你所有的投入，也感谢你使这次合作很有意思。

在编写此书时，我在自己的网站上发表了一些没有编辑过的章节。我这样做是想为自己的站点多吸引一些访问者，同时也想得到一些反馈。所以，非常感谢所有阅读过我的作品并提供了一些很好的反馈意见的人们，尤其是下面这些人：Greg Beamer、Per Jørgensen、Jens Allenbæk、John Fasly、Thomas Moore、Peter Wang 和 Michael Christensen。谢谢你们！