.NET 数据访问架构指南

Alex Mackman, Chris Brooks, Steve Busby, 和 Ed Jezierski
微软公司
2001 年 10 月

概述：本文提供了在多层.NET 应用程序中实施基于 ADO.NET 的数据访问层的指导原则。其重点是一组通用数据访问任务和方案，并指导你选择最合适的途径和技术（68 张打印页）。

目录

简介

如果你在为.NET 应用程序设计数据访问层，那么就应该把 Microsoft ADO.NET 用作数据访问模型。ADO.NET 扩展丰富，并且支持结合松散的数据访问需求、多层 Web 应用程序及 Web 服务。通常，它利用许多扩展丰富的对象模型， ADO.NET 提供了多种方法用于解决一个特定问题。

本文将指导你选择最合适的数据访问方法，其做法是详细列出大范围的通用数据访问方案，提供运用技巧，并且建议最优实践。本文还回答了其它经常问到的问题：何处最适合存放数据库链接字符串？应如何实现链接存储池？如何处理事务？如何实现分页以允许用户在许多记录中滚动？

注意本文的重点是 ADO.NET 的使用：利用 SQL Server .NETData Provider--随 ADO.NET 一起提供的两个供应器之一--访问 Microsoft SQL Server 2000。本文在合适的地方，将突出显示在你使用 OLE DB .NET 数据供应器访问其它 OLE DB 敏感数据源时需要注意的所有差别。

对于利用本文所讨论的指导原则和最优实践所开发的数据访问组件的具体实现，见(Data Access Application Block)数据访问应用程序块。注意，本实现的源代码是可以获得的，并且能直接用于你的.NET 应用程序中。

谁应当阅读本文？

本文为希望构建.NET 应用程序的应用程序设计师和企业开发人员提供了指导原则。如果你负责设计并开发多层.NET 应用程序的数据层，那么请阅读本文。

你首先需要知道什么？

要利用本指南构建.NET 应用程序，你必须有利用 ActiveX 数据对象(ADO)和/或 OLE DB 开发数据访问代码的实际经验，及 SQL Server 经验。你也必须明白如何为.NET 平台开发管理代码，并且也必须清楚 ADO.NET 数据访问模型引入的基本变化。有关.NET 开发的更多信息，见 http://msdn.microsoft.com/net 。

ADO.NET 简介

ADO.NET 是.NET 应用程序的数据访问模型。它能用于访问关系型数据库系统，如 SQL Server 2000，及很多其它已经配备了 OLE DB 供应器的数据源。在某种程度上，ADO.NET 代表了最新版本的 ADO 技术。然而，ADO.NET 引入了一些重大变化和革新，它们专门用于结构松散的、本质非链接的 Web 应用程序。关于 ADO 与 ADO.NET 的比较，见 MSDN 中的"用于 ADO 程序员的 ADO.NET"一文。

ADO.NET 引入的一个重要变化是，用 DataTable, DataSet, DataAdapter, 和 DataReader 对象的组合代替了 ADO Recordset 对象。DataTable 表示来自一个表的行集合，在这方面它与 Recordset 类似。DataSet 表示 DataTable 对象的集合，及与其它表绑定在一起的关系和限制。实际上，DataSet 是具有内置的扩展标记语言（XML）支持的内存中的关联结构。

DataSet 的一个主要特点是，它对底层的数据源一无所知，而这些数据源可能用于对其进行填充。这是一个分离的用于表示数据集合的独立实体，并且它可通过多层应用程序的不同层由一个组件传递到另一组件。它也可作为 XML 数据流被序列化，因而非常适合于不同类型平台间的数据传输。ADO.NET 使用 DataAdapter 对象为发送到和来自 DataSet 及底层数据源的数据建立通道。DataAdapter 对象还支持增强的批更新特性，以前这是 Recorder 的相关功能。

图 1 显示了完整的 DataSet 对象模型。

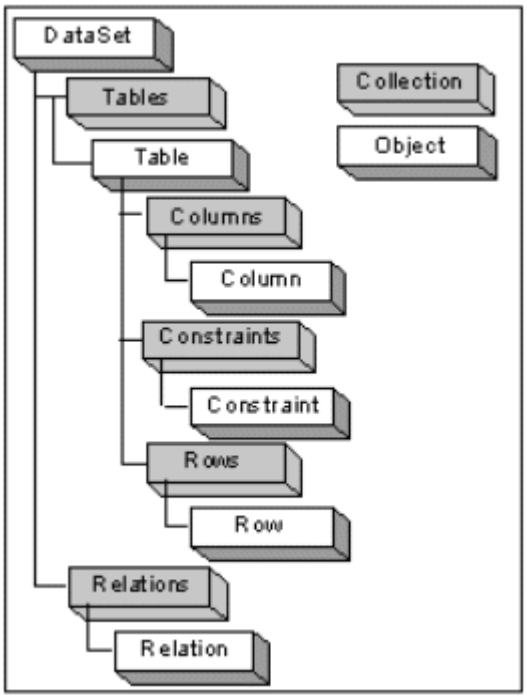

图 1 DataSet 对象模型

.NET 数据供应器

ADO.NET 依靠.NET 数据供应器的服务。 它们提供了对底层数据源的访问，包括四个主要对象（Connection, Command, DataReader,及 DataAdapter），目前，ADO.NET 只发行了两个供应器：

- **SQL Server .NET 数据供应器。这是用于** Microsoft SQL Server 7.0 **及其以后版本数据库的供应器，它优化了对** SQL Server **的访问，并利用** SQL Server **内置的数据转换协议直接与** SQL Server **通信。**
- **当链接到** SQL Server 7.0 **或** SQL Server 2000 **时，总是要使用此供应器。**
- **OLE DB .NET 数据供应器。. 这是一个用于管理** OLE DB **数据源的供应器。它的效率稍低于** SQL Server .NET Data Provider**，因为在与数据库通信时，它需通过** OLE DB **层进行呼叫。注意，此供应器不支持用于开放数据库链接（**ODBC**），**MSDASQL **的** OLE DB **供应器。对于** ODBC **数据源，应使用** ODBC .NET **数据供应器。有关与** ADO.NET **兼容的** OLE DB **供应器列表，见。**

目前测试版中的其它.NET **数据供应器包括：**

- **ODBC .NET 数据供应器。目前** Beta 1.0 **版可供下载。它提供了对** ODBC **驱动器的内置访问，其方式与** OLE DB .NET **数据供应器提供的对本地** OLE DB **供应器的访问方式相同。关于** ODBC .NET **及** Beta **版下载的更多信息见**.
- **用于从 SQL Server 2000 中得到 XML 的管理供应器。用于** SQL Server Web **升级** 2 **版的** XML **还包括了专用于从** SQL Server 2000 **中得到** XML **的管理供应器。关于此升级版本的更多信息，见** .

名称空间组织

与每个.NET **数据供应器相关的类型（类，结构，枚举，等等）位于它们各自的名称空间中：**

- **System.Data.SqlClient**. **包含了** SQL Server .NET **数据供应器类型。**
- **System.Data.OleDb**. **包含了** OLE DB .NET **数据供应器类型。**
- **System.Data.Odbc**. **包含了** ODBC .NET **数据供应器类型。**
- **System.Data**. **包含了独立于供应器的类型，如** DataSet **及** DataTable**。**

在各自关联的名称空间中，每个供应器都提供了 Connection, Command, DataReader, **及** DataAdapter **对象的实现。**SqlClient **实现都有前缀**"Sql" **;而** OleDb **实现前面都有前缀**"OleDb"**。例如，**Connection **对象的** SqlClient **实现是** SqlConnection**。而** OleDb **实现是** OleDbConnection**。类似的，**DataAdapter **对象的两种实现是** SqlDataAdapter **和** OleDbDataAdapter**。**

通用编程

如果你很有可能以不同的数据源为目标，并希望将代码从一种数据源移植到另一数据源，那么可以考虑对 System.Data **名称空间中的** IDbConnection, IDbCommand, IDataReader,**和** IDbDataAdapter **接口进行编程。**Connection, Command, DataReader, **及** DataAdapter **对象的所有实现都必须支持这些接口。**

关于实现.NET **数据供应器的更多信息，见**
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconimplementingnetdataprovider.asp.

图 2 **显示了数据访问堆栈及** ADO.NET **如何与其它数据访问技术，包括** ADO **和** OLE DB**，联系起来。该图还显示了** ADO.NET **模型中的两个管理供应器和主要对象。**

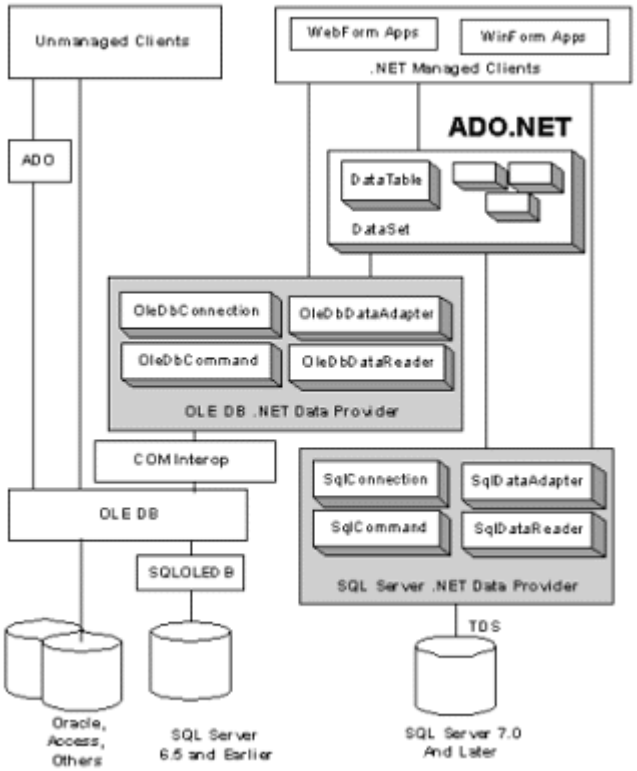

图 2 数据访问堆栈

关于 ADO 到 ADO.NET 的演化，见 MSDN 杂志 2000 年 11 月期的文章“ ADO+简介：用于微软.NET 框架的数据访问服务”。

存储过程与直接 SQL 的比较

在本文剩余部分的大部分代码片段中，都使用了 SqlCommand 对象调用存储过程去执行数据库操作。在一些例子中，你见不到 SqlCommand 对象，因为存储过程名直接传递给了 SqlDataAdapter 对象，但这仍将导致 SqlCommand 对象的创建。

使用存储过程而非 SQL 语句的原因是：

- 存储过程通常会使性能增加，因为数据库可以优化过程使用的数据访问计划，并对其进行缓存以备将来重用。
- 在数据库中，存储过程可分别得到保护。客户可以被给予执行某个存储过程的权限，但无权处理底层的表。
- 存储过程将导致维护简单，因为在一个已部署组件内，修改存储过程通常要比修改硬编码的 SQL 语句简单。
- 存储过程增加了一个从底层的数据库结构中提取出的层。存储过程的客户与存储过程的实现细节及底层结构被隔离开了。
- 存储过程可以降低网络流量，因为 SQL 语句可以以批处理的方式执行，而不是从客户端发送多个请求。

属性与构造函数的比较

可以通过构造函数参数或直接设置属性来为 ADO.NET 对象设置具体的属性值。例如，下面的代码片段在功能上是等同的。

```
// Use constructor arguments to configure command object

SqlCommand cmd = new SqlCommand( "SELECT * FROM PRODUCTS", conn );



// The above line is functionally equivalent to the following

// three lines which set properties explicitly

sqlCommand cmd = new SqlCommand();

cmd.Connection = conn;

cmd.CommandText = "SELECT * FROM PRODUCTS";
```

从性能角度来说，两种方法的差别可以忽略，因为设置或获得.NET 对象的属性比对 COM 对象执行类似操作要有效得多。

所作出的选择只是个人爱好和编码风格而已。然而，明确地设置属性的确使代码易于理解（特别是当你不熟悉 ADO.NET 对象模型时），便于调试。

注意 过去，VB 开发人员被建议避免使用"Dim x As New…!结构创建对象。在 COM 环境中，这些代码将导致 COM 对象创建过程的“短路”，产生一些奇妙的和不怎么奇妙的错误。然而，在.NET 环境中，这已不再是一个问题。

管理数据库链接

数据库链接是一种危险的、昂贵的、有限的资源，特别是在多层 Web 应用程序中。你必须正确管理你的链接，因为你的方法将极大的影响应用程序的整体升级性。还有，必须仔细考虑在哪儿存放链接字符串。你需要一个可配置的、安全的位置。

在管理数据库链接和链接字符串时，你应当努力：

- 通过跨多个客户多路复用一池数据库链接来帮助实现应用程序的扩展性。
- 采用可配置的、高性能的链接池战略。
- 在访问 SQL Server 时使用微软 Windows 操作系统认证。
- 避免中间层的冒充。
- 安全地存储链接字符串。
- 较晚地打开数据库链接，而较早地关闭它们。

本节讨论链接池，并帮你选择合适的链接池战略。其它可选方法也是存在的。本节也将考虑如何管理、存储、控制数据库链接字符串。最后，本节还提供了两个编码方案，使用它们将有助于确保链接已可靠关闭，并返回到链接池中。

链接池

数据库链接池使应用程序能够重用池中的现有链接，而不是重复地建立对数据库的链接。这种技术将极大地增加应用程序的可扩展性，因为有限的数据库链接可以为很多的客户提供服务。此技术也将提高性能，因为能够避免用于建立新链接的巨大时间。

数据访问技术，如 ODBC 和 OLE DB，提供了多种形式的链接池，它们可配置到不同级别上。这两种方式对数据库客户端应用程序来说都是透明的。OLE DB 链接池经常被称为会话或资源池。

关于微软数据访问组件（MDAC）中池的一般讨论，见 http://msdn.microsoft.com/library/en-us/dnmdac/html/pooling2.asp。

ADO.NET 数据供应器提供了透明的链接池，每种链接池的确切机制对每种供应器来说是不同的。本节讨论的链接池是关于:

- SQL Server .NET 数据供应器
- OLE DB .NET 数据供应器

用 SQL Server .NET 数据供应器池化

如果正在使用 SQL Server .NET 数据供应器，那么就可使用该供应器提供的链接池化支持特性。它是由供应器在管理代码内内置实现的对事务敏感的高效机制。每个过程都将创建池，并且直到过程结束，池才被取消。

你可以透明地使用此种链接池，但应当清楚池是如何被管理的，并要知道可以用哪些选项来调整链接池。

如何配置 SQL Server .NET 数据供应器链接池

可以使用一组名称-值对以链接字符串的形式配置链接池。例如，可以配置池是否有效（默认是有效的），池的最大、最小容量，用于打开链接的排队请示被阻断的时间。下面的示例字符串配置了池的最大和最小容量。

```
"Server=(local); Integrated Security=SSPI; Database=Northwind;

Max Pool Size=75; Min Pool Size=5"
```

当链接打开，池被创建时，多个链接增加到池中以使链接数满足所配置的最小值。此后，链接就能增加到池中，直到配置的最大池计数。当达到最大计数时，打开新链接的请求将排队一段可配置的时间。

选择池容量

能建立最大极限对于管理几千用户同时发出请求的大型系统来说是非常重要的。你需要监视链接池及应用程序的性能，以确定系统的最优池容量。最优容量还要依赖于运行 SQL Server 的硬件。

在开发期间，也许需要减小默认的最大池容量（目前是 100）以帮助查找链接泄漏。

如果设立了最小池容量，那么当池最初被填充以达到该值时，会导致一些性能损失，尽管最初链接的几个客户会从中受益。注意，创建新链接的过程被序列化了，这就意味着当池最初被填充时，服务器无法处理同时发生的请求。

关于监视链接池的更多信息，见本文监视链接池一节。关于链接池链接字符串关键字的完整列表，见 http://msdn.microsoft.com/library/en-us/cpguide/html/cpconconnectionpoolingforsqlservernetdataprovider.asp。

更多信息

在使用 SQL Server .NET 数据供应器链接池时，必须清楚：

- 链接是通过对链接字符串精确匹配的法则被池化的。池化机制对名称-值对间的空格敏感。例如，下面的两个链接字符串将生成单独的池，因为第二个字符串包含了一个额外的空字符。

```
SqlConnection conn = new SqlConnection(
            "Integrated  Security=SSPI;Database=Northwind");
conn.Open(); // Pool A is created


SqlConmection conn = new SqlConnection(
            "Integrated  Security=SSPI ; Database=Northwind");
conn.Open(); // Pool B is created (extra spaces in string)
```

- 在.NET 框架 Beta 版中，当在调试器中运行时，链接池化总是失效了。在调试器外，对调试版和发行版，链接池都能正常运作。.NET 框架的最终发行版（RTM）取消了这种限制，链接池在所有情况下都能运行。
- 链接池被划分为了多个特定于事务的池和一个用于目前没有列在事务中的多个链接的池。对于与特定事务上下文相关的线程，将从（包含了与事务建立的链接的）合适的池中返回链接。这使得使用已建立的链接成为透明过程。

用 OLE DB .NET 数据供应器池化

OLE DB .NET 数据供应器利用 OLE DB 资源池化的底层服务将链接存储到池中。很多方法可用于配置资源池化：

- 可以使用链接字符串来配置、使能资源池化或使其使失效。
- 可以使用注册表。
- 可以通过程序来配置资源池化。

为了避开与注册表相关的部署问题，应避免使用注册表配置 OLE DB 资源池化。

关于 OLE DB 资源池化的更多细节，见 MSDN 中“ OLE DB 程序员参考”一书的第 19 章：OLE DB 服务中的资源池化部分。

用池化对象管理链接池化

作为 Windows DNA 开发人员，建议你使 OLE DB 资源池化和/或 ODBC 链接池化失效，并把 COM+对象池化用作将数据库链接存储到池中的技术。这样做主要出于两个原因：

- 池容量和极限可以（在 COM+目录）被明确配置。
- 性能提高了。池化对象的方法可以成倍的胜过固有池化。

然而，由于 SQL Server .NET 数据供应器内置地使用池化，所以（在使用此供应器时）你不再需要开发自己的对象池化机制。这样就可以避免手工事务征募带来的复杂性。

如果正在使用 OLE DB .NET 数据供应器，那么考虑 COM+对象池化以从高级配置和改进的性能中受益。如果你为此目的开发一个池化对象，那么必须使用 OLE DB 资源池化和自动事务征募失效（例如，通过将“OLE DB Services=-4”包含进链接字符串中）。必须在池化对象的实现中处理事务征募。

监视链接池化

要监视应用程序对链接池化的应用情况，可以使用随 SQL Server 发行的 Profiler 工具，或随微软 Windows 2000 发行的性能监视器。

要利用 SQL Server Profiler 监视链接池化，操作如下：

1. 单击开始，指向程序，指向 **Microsoft SQL Server**，然后单击 **Profiler** 运行 Profiler。
2. 在文件菜单中，指向新建，然后单击跟踪。
3. 提供链接内容，然后单击确定。
4. 在跟踪属性对话框中，单击事件标签。
5. 在已选事件类别列表中，确保审核登录和审核登出事件显示在安全审核下面。
6. 单击运行开始跟踪。在链接建立时，将会看到审核登录事件；在链接关闭时看到审核登出事件。

要通过性能监视器监视链接池化，操作如下：

1. 单击开始，指向程序，指向管理工具，然后单击性能运行性能监视器。
2. 在图表背景中右击，然后单击增加计数器。
3. 在性能对象下拉列表框中，单击 SQL Server:通用统计。
4. 在出现的列表中，单击用户链接。
5. 单击增加，然后单击关闭。

注意 .NET 框架的 RTM 版本将另外包含一组 ADO .NET 性能计数器（这些计数器能与性能监视器结合起来使用），这些计数器用于为 SQL Server .NET 数据供应器监视并积累链接池化状态。

管理安全性

尽管数据库链接池化提高了应用程序的整体扩展性，这也意味着你不再能够在数据库端管理安全性。这是因为为了支持链接池化，链接字符串必须是相同的。如果需要跟踪每个用户的数据库操作，那么考虑为每个操作增加一个参数，通过这个参数就可以传递用户身份，手工将用户活动记入数据库。

使用 Windows 认证

在链接到 SQL Server 时，应当使用 Windows 认证，因为它提供了许多优点：

- 安全性易于管理，因为使用了单一(Windows)安全模型而不是分散的 SQL Server 安全模型。
- 避免了在链接字符串中嵌入用户名和密码。
- 用户名和密码不是以明文方式在网络中传输的。
- 通过密码过期期限，最小长度，多次无效登录请求后帐号锁定提高了登录的安全性。

性能

.NETBeta 2 版的性能测试表明，使用 Windows 认证与使用 SQL Server 认证相比，要花费更多的时间才能打开池化的数据库链接。然而，尽管 Windows 认证的成本较高，但与执行一个命令或存储过程所花费的时间相比，其（引起的）性能损失相对来说并不重要。结果，上面所列出的 Windows 认证的优点通常会稍微超过性能损失。

同样，当打开一个池化链接时，在.NET 框架的 RTM 版本中，Windows 认证与 SQL Server 认证的差别有望变得更不明显。

避免在中间层中冒充

Windows 认证需要访问数据库的 Windows 帐号。虽然看上去在中间层中使用冒充更符合逻辑，但必须避免这样做，因为损害链接池化并对应用程序的扩展性产生严重影响。

为了解决这个问题，考虑对有限的 Windows 帐号（而不是被认证的负责人）实施冒充，每个帐号代表一个特定的角色。

例如，可以考虑下面的方法：

- 创建两个 Windows 帐号，一个用于读操作，一个用于写操作（也可以用单独的帐号映射针对特定应用程序的角色。例如，可以为互联网用户使用一个帐号，而为内部操作员和/或管理员使用另外的帐号）。
- 将每个帐号映射到一个 SQL Server 数据库角色，然后为每个角色设置所需的数据库权限。
- 在数据访问层中使用应用程序逻辑确定执行数据库操作时，哪个 Windows 帐号需要冒充。

注意 每个帐号必须是同一域或信任域中在 Internet 信息服务（IIS）和 SQL Server 中存在的域帐号；也可以是在每台计算机上创建（具有相同用户名和密码）的匹配帐号。

为网络库使用 TCP/IP

SQL Server 7.0 及其以后版本支持用于所有网络库的 Windows 认证。使用 TCP/IP 可以获得配置、性能及扩展性优点。关于使用 TCP/IP 的更多信息，见本文通过防火墙建立链接 一节。

存储链接字符串

有多种方法可存储链接字符串，每种方法具有不同程度的灵活性和安全性。尽管在源代码中对字符串进行硬编码提供了最优性能，但文件系统缓存确保了与在文凭系统外部存储字符串相关的性能损失可被忽略。实际上外部链接字符串（允许管理员进行配置）所提供的附加灵活性在任何情况下都是受欢迎的。

选择存储链接字符串的方法时，首先要考虑的两个重要因素是配置的安全性与简易性，其次是性能。

可以选择将数据库链接字符串存储在下列位置：

- 应用程序配置文件 例如用于 ASP.NET Web 应用程序的 Web.config文件。
- 通用数据链接文件(UDL) （只被 OLE DB .NET 数据供应器所支持)
- Windows 注册表
- 定制文件
- COM+ 目录，通过过使用构造字符串(只用于服务组件)

使用 Windows 认证访问 SQL Server，就可以避免在链接字符串存储用户名和密码。如果 安全需求要求更严格的方式，那么就考虑以加密格式存储链接字符串。

对于 ASP.NET Web 应用程序，以加密格式将链接字符串存储在 Web.config文件中是一种安全而可配置的解决方案。

注意，在链接字符串中将 Persist Security Info 命名值设置为假，就可以阻止利用 SqlConnection 或 OleDbConnection 对象的 ConnectionString属性返回对安全敏感的内容，如密码。

下面几个小节讨论了如何用这些方法存储链接字符串，并说明了相对的优点和缺点。这使你能根据特定的应用程序环境作出相应的的选择。

使用 XML 应用程序配置文件

可以使用元素 appSettings 将数据库链接字符串存储在应用程序配置文件的定制设置部分。该元素支持任意关键字-值对，如下面的代码片段所示：

```
<configuration>
 <appSettings>
  <add key="DBConnStr"
      value="server=(local);Integrated Security=SSPI;database=northwind"/>
 </appSettings>
</configuration>
```

注意：appSettings 元素现在在 configuration 元素下面，并且不能直接出现在 system.web 下面。

优点

- **易于部署。通过常规**.NET xcopy **部署，链接字符串随配置文件一起被部署。**
- **通过程序易于访问。**ConfigurationSettings **类的** AppSettings **属性使得在运行时读取数据库链接字符串更为简单。**
- **支持动态更新（仅限于 ASP.NET）。如果管理员更新了** Web.config **文件中的链接字符串，那么下次在字符串被访问时所作出的变化生效，这对一个无状态的组件来说，就象客户再次利用组件作出了数据访问请求一样。**

缺点

安全性。尽管 ASP.NET Internet **服务器应用程序编程接口（**ISAPI**）**DLL **阻止了客户直接访问带**.config **扩展名的文件，并且** NTFS **文件系统权限也用于进一步限制访问，但你可能仍希望避免以明文方式将这些内容存储在前端的** Web **服务器上。要增加安全性，需将链接字符串以加密格式存储在配置文件中。**

更多信息

利用 System.Configuration.ConfigurationSettings **类的** AppSettings **静态属性，可以获取应用程序的定制设置。如下面的代码片段所示，此处假定先前示例的定置关键字为** DBConnStr**。**

```
using System.Configuration;

private string GetDBaseConnectionString()

{

   return  ConfigurationSettings.AppSettings["DBConnStr"];

}
```

关于配置.NET **框架应用程序的更多信息，见**
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconconfiguringnetframeworkapplications.asp.

使用 UDL 文件

OLE DB .NET **数据供应器支持在它的链接字符串中使用统一数据链接（**UDL**）文件名。可以以构建参数的形式将链接字符串传给** OleDbConnection **对象，或利用对象的** ConnectionString **属性设置链接字符串。**

注意 SQL Server .NET **数据供应器不支持在它的链接字符串中使用** UDL **文件。因此，只有使用** OLE DB .NET **数据供应器，此方法才有效。**

对于 OLE DB **供应器，要利用链接字符串引用** UDL **文件，使用**“ File Name=name.udl.”**。**

优点

标准方法。你也许已经在用 UDL **文件进行链接字符串的管理了。**

缺点

- **性能。**每次打开链接时，包含 UDLs 的链接字符串都被读取并被解析。
- **安全性。**UDL 文件以纯文本格式存储。利用 NFTS 文件权限可以确保这些文件的安全性，但这样做将引发与使用.config文件相同的问题。
- **SqlClient 对象不支持 UDL 文件。**此方法不被 SQL Server .NET 数据供应器所支持，而你要用此供应器访问 SQL Server 7.0 及其以后版本。

更多信息

- 必须确保管理员拥有该文件的读/写访问权限以便进行管理，并且还要确保运行应用程序的身份拥有读权限。对于 ASP.NET Web 应用程序，应用程序工作者进程默认是以 SYSTEM 帐号运行的，但利用机器范围的配置文件（Machine.config）中的元素可以将其覆盖掉。利用 Web.config文件中的元素，及一个可选的指定帐号，可以进行冒充。
- 对于 Web 应用程序，要确保没有将 UDL 文件放在虚目录中，因为那样会使该文件可通过网络下载。
- 关于这些及其它与安全性相关的 ASP.NET 特性的更多信息，见 http://msdn.microsoft.com/library/en-us/dnbda/html/authaspdotnet.asp .。

使用 Windows 注册表

可以利用定制关键字将链接字符串存储在 Windows 注册表中，但由于部署问题，建议不要使用。

优点

- **安全性。**利用访问控制列表（ACLs），可以对所选的注册表关键字的访问进行管理。对更高级别的安全性，考虑对数据进行加密。
- **通过程序易于访问。**.NET 类支持从注册表中读取字符串。

缺点

- **部署。**相关的注册表设置必须同应用程序一起部署，从某种程度上抵消了 xcopy 部署的优点。

使用定置文件

可以使用定制文件来存储链接字符串，然而这种技术没有优点，因此并不推荐使用。

优点

- 没有

缺点

- **额外编码。**这种方法需要额外编码，并迫使你明确处理同时发生的问题。
- **部署。**此文件必须同其它 ASP.NET 应用程序文件一起拷贝。避免将此文件放在 ASP.NET 应用程序的目录或子目录中，就可以阻止通过网络对其进行下载。

使用构建参数和 COM+目录

可以将链接字符串存储在 COM+目录中，并利用对象的构造字符串将它自动地传递给对象。COM+在初始化对象，提供配置构造字符串后，将立即调用对象的 Construct 方法。

注意这个方法只用于服务组件。只有管理组件使用了其它服务，如分布式事务处理支持或对象池化时，才考虑使用此方法。

优点

- **管理性**。利用组件服务 MMC 插件，管理员可以很方便地配置链接字符串。

缺点

- **安全性**。COM+目录被认为是一个不安全的存储区（虽然利用 COM+角色你可以限制对它的访问），并因此不能用于以明文维护链接字符串。
- **部署**。COM+目录中的条目必须随.NET 应用程序一同部署。如果使用了其它企业服务，如分布式事务或对象池化，那么将数据库链接字符串存储在目录中不会增加部署的额外开销，因为要支持其它服务，必须部署 COM+目录。
- **必须为组件提供服务**。可以只为所服务的组件使用构造字符串。要使能构造字符串，不能简单地从 ServicedComponent 类中派生所需组件类（这将为组件提供服务）。

更多信息

- 关于如何为对象构造配置.NET 类的更多信息，见附录中的如何为.NET 类使能对象构造 。
- 关于开发服务组件的更多信息，见 http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingservicedcomponents.asp 。

链接使用方式

不管何种.NET 数据供应器，你必须总是：

- 尽可能晚地打开数据库链接。
- 以尽可能短的时间使用该链接。
- 尽可能快地关闭该链接。链接直到通过 Close 或 Dispose 方法关闭后，它才返回到池中。即使发现它处于崩溃状态，也应当关闭它。这样做确保了它能返回池中，并被标记为无效。对象池周期性地扫描池，以查找已被标记为无效的对象。

为确保在方法返回前链接已经关闭，考虑使用下面两个代码片段中演示的方法。第一个示例使用了 finally 块，第二个示例使用了 C# using 声明，此声明确保了对象的 Dispose 方法被调用。

下面的代码确保 finally 块关闭了链接。注意，此方法只用于 Visual Basic .NET 及 C#中，因为 Visual Basic .NET 支持结构化例外处理。

```
public void DoSomeWork()
```

```
{
  SqlConnection conn = new SqlConnection(connectionString);
  SqlCommand cmd = new SqlCommand("CommandProc", conn );
  cmd.CommandType = CommandType.StoredProcedure;


   try
   {
      conn.Open();
      cmd.ExecuteNonQuery();
   }
  catch (Exception e)
   {
     // Handle and log error
   }
   finally
   {
      conn.Close();
   }
}
```

现在的代码显示了另外一种方法，此方法使用了 C# using **声明。注意，**Visual Basic .NET **并不支持** using **声明，或任何功能相同的对应语句。**

```
public void DoSomeWork()
{
  // using guarantees that Dispose is called on conn, which will
  // close the connection.
  using (SqlConnection conn = new SqlConnection(connectionString))
```

```
  {
    SqlCommand cmd = new SqlCommand("CommandProc", conn);
     fcmd.CommandType = CommandType.StoredProcedure;
      conn.Open();
      cmd.ExecuteQuery();
  }
}
```

此方法也适用于其它对象，如 SqlDataReader 或 OleDbDataReader，在其它任何对象对当前链接进行处理前，这些对象必须被关闭。

错误处理

ADO.NET 错误生成后，将由.NET 框架内置的底层结构化异常处理支持所处理。结果，在数据访问代码中的错误处理方式与应用程序中其它地方的错误处理方式完全相同。通过标准的.NET 异常处理语法和技术，异常被检测到并被处理。

本节描述了如何开发强壮的数据访问代码，并解释了如何处理数据访问错误。本节还提供了与 SQL Server .NET 数据供应器相关的异常处理详尽指南。

.NET 异常

.NET 数据供应器将特定的数据库的错误状态转化为标准的异常类型，应当在数据访问代码中对这些异常进行处理。通过相关的异常对象的属性，可以获得特定数据库的错误细节。

所有.NET 异常类型最终是从 System 名称空间的 Exception 基类中派生的。.NET 数据供应器释放特定的供应器异常类型。例如，一旦 SQL Server 返回一个错误状态时，SQL Server .NET 数据供应器释放 SqlException 对象。类似的，OLE DB .NET 数据供应器释放 OleDbException 类型的异常，此对象包含了由底层 OLE DB 供应器暴露的细节。

图 3 显示了.NET 数据供应器异常的层次结构。注意，OleDbException 类是从 ExternalException 类派生的 ExternalException 类是所有 COM 例外的基类。对象的 ErrorCode 属性存储了 OLE DB 生成的 COM HRESULT。

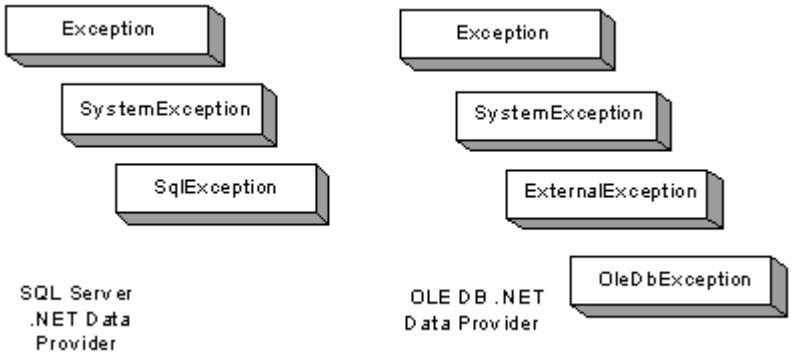

图 3 NET 数据供应器层次结构

缓存并处理.NET 异常

要处理数据访问例外状态，将数据访问代码放在 try 块中，并在 catch 块中利用合适的过滤器捕获生成的任何例外。例如，当利用 SQL Server .NET 数据供应器编写数据访问代码时，应当捕获 SqlException 类型的异常，如下面的代码所示：

```
try
{
  // Data access code
}
catch (SqlException sqlex) // more specific
{
}
catch (Exception ex) // less specific
{
}
```

如果为不止一个 catch 声明提供了不同的过滤标准，记住，按最特殊类型到最不特殊类型的顺序排列它们。通过这种方式，catch 块中最特殊类型将将为任何给定的类型所执行。

SqlException 类所暴露的属性包含了例外状态的细节。其中包括：

- Message 属性，它包含了用于描述错误的文本。
- Number 属性，它包含唯一标识错误类型的错误号。

- State 属性。它包含了关于错误启用状态的附加信息。它经常用于指示特殊错误状态的某个特定事件。例如，如果单一存储过程从不止一行中生成同样的错误，那么本属性将用于标识某个具体的事件。
- Errors 集合。它包含了 SQL Server 生成的错误的详细信息。此集合部是包含至少一个 SqlError 类型的对象。

下面的代码片段演示了如何利用 SQL Server .NET 数据供应器处理 SQL Server 错误状态：

```
using System.Data;

using System.Data.SqlClient;

using System.Diagnostics;


// Method exposed by a Data Access Layer (DAL) Component

public string GetProductName( int ProductID )

{

  SqlConnection conn = new SqlConnection(

            "server=(local);Integrated  Security=SSPI;database=northwind");

  // Enclose all data access code within a try block

   try

   {

      conn.Open();

     SqlCommand cmd = new SqlCommand("LookupProductName", conn );

      cmd.CommandType = CommandType.StoredProcedure;


      cmd.Parameters.Add("@ProductID", ProductID );

      SqlParameter paramPN =

              cmd.Parameters.Add("@ProductName", SqlDbType.VarChar, 40 );

      paramPN.Direction = ParameterDirection.Output;
```

```
      cmd.ExecuteNonQuery();
    // The finally code is executed before the method returns
      return paramPN.Value.ToString();
   }
  catch (SqlException sqlex)
   {
      // Handle data access exception condition
    // Log specific exception details
      LogException(sqlex);
    // Wrap the current exception in a more relevant
    // outer exception and re-throw the new exception
     throw new DALException(
                              "Unknown ProductID: " + ProductID.ToString(), sqlex );
   }
  catch (Exception ex)
   {
    // Handle generic exception condition . . .
      throw ex;
   }
   finally
   {
    conn.Close(); // Ensures connection is closed
   }
}
```

```
// Helper routine that logs SqlException details to the
// Application event log
private void LogException( SqlException sqlex )
{
  EventLog el = new EventLog();
  el.Source = "CustomAppLog";
  string strMessage;
  strMessage = "Exception Number : " + sqlex.Number +
                    "(" + sqlex.Message + ") has occurred";
  el.WriteEntry( strMessage );

  foreach (SqlError sqle in sqlex.Errors)
  {
    strMessage = "Message: " + sqle.Message +
                    " Number: " + sqle.Number +
                    " Procedure: " + sqle.Procedure +
                    " Server: " + sqle.Server +
                    " Source: " + sqle.Source +
                    " State: " + sqle.State +
                    " Severity: " + sqle.Class +
                    " LineNumber: " + sqle.LineNumber;
    el.WriteEntry( strMessage );
  }
}
```

在 SqlException catch 块中，代码最初利用 LogException 帮助函数记录错误状态，此函数利用 foreach 声明枚举了 Errors 集合中特定于供应器的细节，并将错误细节记录到错误日志中。 Catch 块中的代码然后将特

定于 SQL Server 的例外封装在 DALException 类型的对象中，这样做对调用者的 GetProductName 方法更具有意义。例外处理程序使用关键字 throw 将例外传回调用者。

更多信息

- 关于 SqlException 类成员的完整列表，见 http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataSqlClientSqlExceptionMembersTopic.asp。
- 关于定置例外的开发，.NET 例外的记录与封装，返回例外的不同方法的使用的更多信息，见 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp 。

从存储过程中生成错误

T-SQL 提供了一个 RAISERROR（注意拼写）函数。你可用此函数生成定置错误，并将错误返回客户。对于 ADO.NET 客户，SQL Server .NET 数据供应器对这些数据错误进行解释，并把它们转化为 SqlError 对象。

使用 RAISERROR 函数是简单地方法是将消息文本作为第一个参数包括进来，然后指定严重及状态参数，如下面的代码片段所示：

```
RAISERROR( 'Unknown Product ID: %s', 16, 1, @ProductID )
```

在这个例子中，替代参数用于将当前产品 ID 作为错误消息文本的一部分返回，参数 2 是消息的严重性，参数 3 是消息状态。

更多信息

- 为了避免对消息文本进行硬编码，你可以利用 sp_addmessage 系统存储过程或 SQL Server 企业管理器将你自己的消息增加到 sysmessages 表中。然后你就可以使用传递到 RAISERROR 函数的 ID 引用消息了。你所定义的消息 Ids 必须大于 50000，如下代码片段所示：
- RAISERROR(50001, 16, 1, @ProductID)
- 关于 RAISERROR 函数的完整细节，请在 SQL Server 的在线书目中查询 RAISERROR。

正确使用严重性等级

仔细选择错误严重性等级，并要清楚每个级别造成的冲击。错误严重性等级的范围是 0-25，并且它用于指出 SQL Server 2000 所遇到的问题的类型。在客户端代码中，通过在 SqlException 类的 Errors 集合中检查 SqlError 对象的 Class 属性，你可以获得错误的严重性。表 1 指出了不同严重性等级的意义及所造成的冲击。

表 1.错误严重性等级--冲击及意义

严重性等级	链接已关闭	生成 SqlException 对象	意义
10 及其以下	No	No	通知型消息，并不表示犯错误状态。
11-16	No	Yes	可由用户修改的错误，例如，使用修改后的输入数据

			重试操作。
17-19	No	Yes	资源或系统错误。
20-25	Yes	Yes	致命的系统错误（包括硬件错误）。客户链接被终止。

控制自动化事务

SQL Server .NET 数据供应器对它所遇到的任何严重性大于 10 的错误都抛出 SqlException 对象。当作为自动化（COM+）事务一部分的组件检测到 SqlException 对象后，该组件必须确保它能取消事务。这也许是，也许不是自动化过程，并要依赖该方法是否已经对 AutoComplete 属性作出了标记。

关于在自动化事务上下文中处理对象的更多信息，见本文中的确定事务结果一节。

得到通知型消息

10 及其以下严重性等级用于表示通知型消息，并且不会引发 SqlException 对象的抛出。

要获得通知型消息：

- >创建事件处理程序，并提交给 SqlConnection 对象所暴露的 InfoMessage 事件。下面的代码片段显示了事件代理。

```
public delegate void SqlInfoMessageEventHandler( object sender,

SqlInfoMessageEventArgs e );
```

通过传递到你的事件处理处理程序中的 SqlInfoMessageEventArgs 对象，可以得到消息数据。此对象暴露了 Errors 属性，该属性包含一组 SqlError 对象--每个通知消息一个 SqlError 对象。下面的代码片段演示了如何注册用于记录通知型消息的事件处理程序。

```
public string GetProductName( int ProductID )
{
  SqlConnection conn = new SqlConnection(
            "server=(local);Integrated  Security=SSPI;database=northwind");
   try
   {
     // Register a message event handler
       conn.InfoMessage += new
SqlInfoMessageEventHandler( MessageEventHandler );
```

```
      conn.Open();
    // Setup command object and execute it
      . . .
   }
  catch (SqlException sqlex)
   {
     // log and handle exception
      . . .
   }
   finally
   {
      conn.Close();
   }
}
// message event handler
void MessageEventHandler( object sender, SqlInfoMessageEventArgs e )
{
  foreach( SqlError sqle in e.Errors )
   {
     // Log SqlError properties
      . . .
   }
}
```

性能

本节介绍了一些常见的数据访问方案，对每种方案，以 ADO.NET **数据访问代码的形式描述了最优性能和扩展性解决方案。在合适的地方，还对性能，功能及开发最作出了比较。本节考虑了下面的功能方案。**

- 获取多行. 获取一个结果集，并在得到的行中重复。
- 获取一行. 获取具有指定关键字的一行。
- 获取一项. 从指定的行中得到一项。
- 确定某项数据的存在性. 检查具有特定关键字的一行是否存在。这是单项查找方案的一种变体，这里返回一个简单的布尔值就足够了。

获取多行

在这个方案中，你要获取一组表格化数据，并在得到的行中重复执行某个操作。例如你得到了一组数据，并以非链接的方式处理，然后（可能通过 Web 服务）将它作为 XML 文档传递给客户应用程序。可选的，你也可以以 HTML 表的形式将这些数据显示出来。

为了帮助确定最合适的数据访问方法，考虑你是否需要（非链接）DataSet 对象的附加灵活性，还是只需要 SqlDataReader 对象提供的原有性能，这些性能非常适合于 B2C Web 应用程序的数据表示。图 4 显示了这两种基本场景。

注意用于填充 DataSet 的 SqlDataAdapter 利用 SqlDataReader 方法数据。

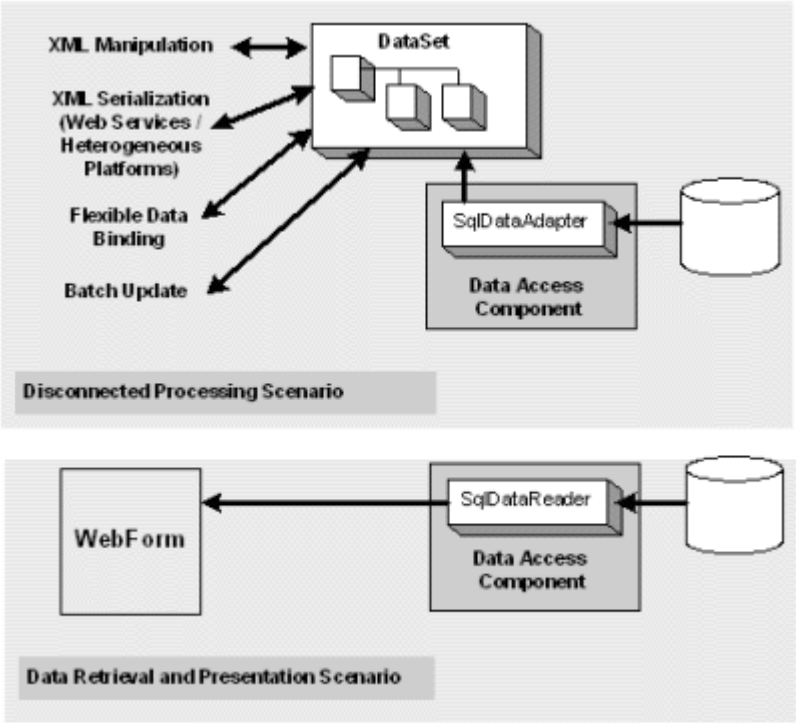

图 4 多行数据访问方案

方法比较

当从数据源中获取多行时，你可以使用下面的方法：

- 使用 SqlDataAdapter 对象生成 DataSet 或 DataTabl 对象。
- 利用 SqlDataReader 对象提供只读的只向前的数据流。

- 利用 XmlReader 对象提供只读的只向前的 XML 数据流。

SqlDataReader 与 DataSet/DataTable 间的选择本质上是性能与功能间的选择。SqlDataReader 提供了最优性能，而 DataSet 提供了额外的功能与灵活性。

数据绑定

所有这三个对象都可以作为数据绑定控件的数据源。而 DataSet 和 DataTable 可作为更广范围控件的数据源。这是因为 DataSet 和 DataTable 实现了（生成 Ilist 接口）IlistSource 接口，而 SqlDataReader 实现了 Ienumerable 接口。许多能进行数据绑定的 WinForm 控件需要实现了 Ilist 接口的数据源。

这种不同是因为为每种对象类型设计的场景类型不同。DataSet (它包含 DataTable)是一个丰富的、非链接结构，它适合于 Web 和桌面（WinForm）应用程序。另一方面，数据阅读器已经为 Web 应用程序进行了优化，这种应用程序需要优化的、只能向前的数据访问。

检查将要绑定到的特定控件类型的数据源需求。

在应用程序层间传递数据

DataSet 提供了可作为 XML 被任意操纵数据的关系图，并允许数据的非链接缓存拷贝在应用程序层与组件间传递。然而，SqlDataReader 提供了更优化的性能，因为它避免了与创建 DataSet 相关的性能及内存开销。记住，DataSet 对象的创建将导致多个子对象--包括 DataTable, DataRow 和 DataColumn--及作为这些子对象容器的集合对象的创建。

使用 DataSet

使用 SqlDataAdapter 填充的 DataSet 对象，当：

- 你需要非链接的驻留内存的缓存数据，以便你能将它传递到其它组件或应用程序中的其它层。
- 你需要内存中的数据关系图以执行 XML 或非 XML 操作。
- 你正在使用的数据来自多个数据源，如多个数据库、表或文件。
- 你希望更新获得的一些或所有行，并希望利用 SqlDataAdapter 的批更新功能。
- 你要对控件绑定数据，而此控件需要支持 IList 接口的数据源。

更多信息

如果使用 SqlDataAdapter 生成 DataSet 或 DataTable，需注意：

- 不必明确打开或关闭数据库链接。SqlDataAdapter Fill 方法打开数据库链接，并在此方法返回前关闭该链接。如果链接原来已经打开，那么此方法仍使链接处于打开状态。
- 如果出于其它目的需要链接，那么考虑在调用 Fill 方法前打开链接。这样你就可以避免不必要的打开/关闭操作，提高性能。
- 尽管能重复使用同一 SqlCommand 对象多执行同样的命令，但不要重复使用此对象执行不同的命令。
- 关于如何利用 SqlDataAdapter 对象填充 DataSet 或 DataTable 对象的代码示例，见附录中的如何利用 SqlDataAdapter 对象获得多行。

使用 SqlDataReader

些劣情况，可以使用通过调用 SqlCommand **对象的** ExecuteReader **方法得到的** SqlDataReader **对象：**

- **正在处理大量数据时--太多了而不能在单个缓冲区内维护。**
- **希望减少应用程序在内存中的印迹。**
- **希望避免与** DataSet **对象创建相关的开销。**
- **希望对某控件执行数据绑定操作，而此控件支持实现了** IEnumerable **接口的数据源。**
- **希望流水线化数据访问，并对其优化。**
- **正在读取包含二进制大对象（**BLOB**）列的行。你可以使用** SqlDataReader **对象以可管理的大块为单位从数据库中将** BLOB **数据拉出来，而不是一次性地将所有数据提取出来。关于处理** BLOB **数据的更多细节，见本文处理** BLOBs **一节。**

更多信息

如果使用 SqlDataReader **对象，请注意：**

- **在数据阅读器活动期间，底层的数据库链接保持打开，并不能用于其它任何目的。尽可能早地对** SqlDataReader **对象调用** Close **方法。**
- **每个链接只能有一个数据阅读器。**
- **通过向** ExecuteReader **方法传递** CommandBehavior.CloseConnection **枚举值，可以在使用完数据阅读器后，明确地关闭链接；或者，将链接生命周期绑定到** SqlDataReader **对象。这预示着当** SqlDataReader **对象关闭时，链接也将关闭。**
- **在利用阅读器访问数据时，如果你知道列的底层数据类型，那么就应使用类型化存取器方法(如** GetInt32 **和** GetString)**，这是因为在读取列数据时，这些方法减少了读取列数据所需的类型转换量。**
- **为避免将不必要的数据从服务器发送到客户端，如果你要关闭阅读器并抛弃所有保留的结果，那么在对阅读器调用** Close **方法前调用命令对象的** Cancel **方法。**Cancel **方法确保了服务器的结果被抛弃，而不会被发送到客户端。相反，对数据阅读器调用** Close **方法会使阅读器不必要地提取出保留的结果，以清空数据流。**
- **如果要得到从存储过程返回的输出值或返回值，并且你在利用** SqlCommand **对象的** ExecuteReader **方法，那么在得到输出或返回值前，必须对阅读器调用** Close **方法。**
- **关于演示如何利用** SqlDataReader **对象的代码示例，附录中的如何利用** SqlDataReader **对象获取多行数据。**

使用 XmlReader

下列情况下，使用通过调用 SqlCommand **对象的** ExecuteXmlReader **方法得到的** XmlReader **对象：**

- **希望将得到的数据作为** XML **处理，但不希望引发因创建** DataSet **对象而造成的额外性能开销，并且不需要数据的非链接缓存。**
- **希望利用** SQL Server FOR XML **语法的功能，这种语法允许以灵活的方式从数据库中得到** XML **片段（即，不带根元素的** XML **文档）。例如，这种方法使你能够精确指定元素名，是使用元素还是使用以属性为核心的图解，图解是否随** XML **数据一起被返回，等等。**

更多信息

如果使用 XmlReader**，请注意**:

- **在从** XmlReader **对象中读取数据时，链接必须保持打开。** SqlCommand **对象的** ExecuteXmlReader **方法目前不支持** CommandBehavior.CloseConnection **枚举值，因此在使用完阅读器后必须明确关闭链接。**
- **对于如何使用** XmlReader **对象的代码示例，见附录中的如何利用** XmlReader **获取多行数据。**

获取单行数据

在这种场景中，将从数据源中获取包含一组指定列的单行数据。例如，你得到一个客户 ID**，并希望查找与客户相关的细节；或得到一个产品** ID**，并希望得到产品信息。**

方法比较

如果要对从数据源中得到的一行数据执行绑定操作，可以用 SqlDataAdapter **对象填充** DataSet **或** DataTable **对象，其方式与在先前讨论过的获取多行数据及重复场景中描述的方式相同。然而，除非特别需要** DataSet **或** DataTable **对象的功能，否则应当避免创建这些对象。**

如果需要获取单行数据，那么请使用下面的一种方法：

- **使用存储过程输出参数**.
- **使用** SqlDataReader **对象**.

这两种方法都避免了在服务器端创建结果集，在客户端创建 DataSet **对象的不必要额外开销。每种方法的相对性能要依赖于强度等级及数据库链接池化是否被使能。当数据库链接池化使能时，性能测试表明存储过程方法在高强度环境下（同时存在** 200 **多链接）其性能比** SqlDataReader **方法高近** 30%**。**

使用存储过程输出参数

如下情况中使用存储过程输出参数：

- **要从链接池化使能的多层** Web **应用程序中获得一行数据。**

更多信息

- **关于演示如何使用存储过程输出参数的代码示例，见附录中的使用存储过程输出参数获取一行数据。**

使用 SqlDataReader 对象

下列情况，需使用 SqlDataReader **对象：**

- **除了数据值，还需要元数据时。可以利用数据阅读器的** GetSchemaTable **方法获取列元数据。**

- 未使用链接池化时。在链接池化无效时，SqlDataReader 对象在所有强度环境下都是好方式；性能测试表明，在 200 浏览器链接时，此方法比存储过程方法在性能上要高约 20%。

更多信息

- 如果知道查询结果只需返回一行，那么在调用 SqlCommand 对象的 ExecuteReader 方法时，使用 CommandBehavior.SingleRow 枚举值。一些供应器，如 OLE DB .NET 数据供应器，用此技巧来优化性能。例如，供应器使用 IRow 接口（如果此接口存在）而不是代价更高的 IRowset 接口。这个参数对 SQL Server .NET 数据供应器没有影响。
- 在使用 SqlDataReader 对象时，总是应当通过 SqlDataReader 对象的类型化存取器方法，如 GetString 和 GetDecimal，获得输出参数。这样做就避免了不必要的类型转换。
- 关于如何使用 SqlDataReader 对象获取单行数据的代码示例，见附录中的如何使用 SqlDataReader 对象获取单行数据。

获取单项数据

在本场景中，要获取单项数据。例如，提供了产品 ID 后，希望查询单一的产品名；或，给出了客户名后，希望查询客户的信用等级。在这种场景中，为得到单项数据，通常不希望引发创建 DataSet 对象或甚至是 DataTable 对象的额外开销。

也许只希望检查数据库中是否存在特定的行。例如，当新用户在网站注册时，需要检查所选用户名是否已经存在。这是单项数据查询中很特殊的例子，但在此例子中，返回一个简单的布尔返回值就足够了。

方法比较

当从数据源获取单项数据时，考虑下面的方法：

- 同存储过程一起使用 SqlCommand 对象的 ExecuteScalar 方法。
- 使用存储过程输出或返回参数。
- 使用 SqlDataReader 对象。

ExecuteScalar 方法直接返回数据项，因为它是为只返回单个值的查询设计的，与存储过程输出参数和 SqlDataReader 方法相比，它需要更少的代码。

从性能方面来说，应当使用存储过程输出或返回参数，因为测试结果表明，存储过程方法在从低强度到高强度环境中（从同时不到 100 浏览器链接到 200 浏览器链接）提供了一致的性能。

更多信息

- 如果通过 ExecuteQuery 方法所执行的查询返回多列和/或行，那么此方法只返回第一行的第一列。
- 关于演示如何使用 ExecuteScalar 方法的代码片段，见附录中的如何使用 ExecuteScalar 获取单项数据。
- 关于演示如何利用存储过程输出或返回参数获取单项数据的代码示例，见附录中的如何利用存储过程输出或返回参数获取单项数据
- 关于演示如何使用 SqlDataReader 对象获取单项数据的代码示例，见附录中的如何使用 SqlDataReader 对象获取单项数据。

通过防火墙建立链接

需要经常配置互联网应用程序以使它能够通过防火墙链接到 SQL Server。例如，许多 Web 应用程序及防火墙的主要结构组件是周边网络（也被称为 DMZ 或非军事化区），它们用于隔离高端 Web 服务器与内部网络。

通过防火墙链接到 SQL Server 时，需要对防火墙，客户和服务器进行明确配置。SQL Server 提供了客户网络应用程序和服务器网络应用程序以帮助进行配置。

选择网络库

当通过防火墙建立链接时，使用 SQL Server TCP/IP 网络库来简化配置，这是 SQL Server2000 安装的默认选项。如果使用先前版本的 SQL Server，那么分别利用客户端网络应用程序和服务器端网络应用程序检查 TCP/IP 是否在客户和服务器端已经被配置为默认的网络库。

除了配置优点，使用 TCP/IP 库还意味着：

- 受益于大宗数据的改进性能和增加的扩展性。
- 避免与指定管道相关的附加安全信息。

必须在客户和服务器计算机上配置 TCP/IP，因为大多数防火墙限制了流量通过的端口，所以必须仔细考虑 SQL Server 所使用的端口号。

配置服务器

SQL Server 的默认实例监听 1433 端口。然而，SQL Server 2000 的指定实例在它们首次开启时，动态地分配端口号。网络管理员有希望在防火墙打开一定范围的端口；因此，当随防火墙使用 SQL Server 的指定实例时，利用服务网络应用程序对实例进行配置，使它监听特定的端口。然后管理员对防火墙进行配置，以使防火墙允许流量到达特定的 IP 地址及服务器实例所监听的端口。

注意，客户端网络库所使用的源端口号在 1024-5000 间动态分配。这是 TCP/IP 客户端应用程序的标准作法，但这意味着防火墙必须允许途经此范围的任何端口流量能够通过。关于 SQL Server 所使用的端口的更多信息，在微软产品支持服务网站上，参见 INF: P 通过防火墙对 SQL Server 进行通讯所需的 TCP 端口 。。

动态查找指定实例

如果改变了 SQL Server 所监听的默认端口，那么就要对客户端进行配置，以使它链接到此端口。更多细节，见本文中的配置客户端 一节。

如果改变了 SQL Server 2000 默认实例的端口号，那么不修改客户端将导致链接错误。如果存在多个 SQL Server 实例，最新版本的 MDAC 数据访问堆栈（2.6）将进行动态查找，并利用用户数据报协议（UDP）协商（通过 UDP 端口 1434）对指定实例进行定位。尽管这种方法在开发环境下也许有效，但在现在环境中却不大可能正常工作，因为典型发问下防火墙阻止 UDP 协商流量的通过。

为了避开这种情况，总是将客户端配置为链接到已配置好的目的端口号。

配置客户端

应当对客户端进行配置以利用 TCP/IP 网络库链接到 SQL Server，并且也应当确保客户端库使用了正确的目的端口号。

使用 TCP/IP 网络库

利用 SQL Server 客户端网络库，可以对客户端进行配置。在某些安装版本中，可能没有将这个应用程序安装到客户端（如 Web 服务器）。在这种情况下，可以按如下方式之一解决：

- 利用通过链接字符串提供的“ Network Library=dbmssocn”名称-值对指定网络库。字符串 dbmssocn 用于标识 TCP/IP（套接字）库。

注意 在使用 SQL Server .NET 数据供应器时，网络库的默认设置是使用“ dbmssocn”。

- 在客户端机器上修改注册表，把 TCP/IP 设置为默认库。关于配置 SQL Server 网络库的更多信息，参见 HOWTO: 不使用客户端网络应用程序而修改 SQL Server 默认网络库(Q250550)。

指定端口

如果 SQL Server 的实例被配置为监听默认的 1433 以外的其它端口，那么通过以下操作，就能指定链接到的端口号：

- 使用客户端网络应用程序
- 利用提供给链接字符串的“ Server”或“ Data Source”名称-值对来指定端口号。要按下面的格式使用字符串：
- "Data Source=ServerName,PortNumber"

注意 ServerName 可以是 IP 地址，或域名系统（DNS）名，为了优化性能，可以使用 IP 地址以避免 DNS 查询。

分布式事务处理

如果开发了使用 COM+分布式事务处理和微软分布式事务处理协调器（DTC）服务的服务组件，那么就需要对防火墙进行配置，以允许 DTC 流在不同 DTC 实例间及 DTC 与资源管理器（例如 SQL Server）间流动。

有关为 DTC 开放端口的更多信息，见 INFO:为通过防火墙工作，配置微软分布式事务处理协调器 (DTC)。

处理 BLOBs

目前，很多应用程序除了处理许多传统的字符串和数字型数据外，还要处理象图形或声音--甚至复杂的数据格式，如视频格式的数据。图形、声音与视频的数据格式类型不一。然而从存储角度来说，它们都可被视为二进制数据块，通常将其称为 BLOBs（二进制大对象）。

SQL Server 提供了 binary, varbinary, 和 image 数据格式来存储 BLOBs。不考虑名称，BLOB 数据也可被称为基于文件的数据。例如，你可能要存储与特定行相关的二进制长注释字段。SQL Server 为此目的提供了 ntext 和 text 数据类型。

通常，对于小于 8KB 的二进制数据，使用 varbinary 数据类型。对于超过此大小的二进制数据，使用 image 。表 2 汇集了每个数据类型的主要特性。

表 2 数据类型特性

数据类型	大小	描述
binary	范围从 1-8KB。存储大小是指定大小加 4 字节。	固定长度的二进制数据
varbinary	范围从 1-8KB。存储大小是所提供数据的实际大小加 4 字节。	可变长度的二进制数据
image	从 0-2GB 大小的可变长度二进制数据	大容量可变长度二进制数据
text	从 0-2GB 大小的可变长度数据	字符型数据
ntext	从 0-2GB 大小的可变长度数据	宽字节字符数据

何处存储 BLOB 数据

SQL Server 7.0 及其以后版本已经提高了存储在数据库中的 BLOB 数据的使用性能。这种情况的一个原因是数据库页面大小已经增加到了 8KB。结果，小于 8KB 的文本或图象数据不必再存储在页面单独的二进制树结构中，而是能被存储在单行中。这意味着读取和写入 text, ntext, 或 image 数据能象读取或写入字符或二进制字符串那样快。超出 8KB 后，将在行中建立一个指针，数据本身存储在独立数据页面的二进制树结构中，这不可避免会对性能产生冲击。

关于迫使 text, ntext, 和 image 数据存储在单行中的更多信息，见 SQL Server 在线图书中的使用 text 和 image 数据主题。

一个经常使用的处理 BLOB 数据的可选方法是，将 BLOB 数据存储在文件系统中，并在数据库列中存储一个指针（通常是一个统一资源定位器--URL 链接）以引用正确的文件。对于 SQL Server 7.0 以前的版本，将 BLOB 数据存储在数据库外的文件系统中，可以提高性能。

然而，SQL Server 2000 改进了 BLOB 支持，以及 ADO.NET 对读取和写入 BLOB 数据的支持，使在数据库中存储 BLOB 数据成为一种可行的方法。

在数据库中存储 BLOB 数据的优点

将 BLOB 数据存储在数据库中，带来了很多优点：

- 易于保持 BLOB 数据与行中其它项数据的同步。
- BLOB 数据由数据库所支持，拥有单一的存储流，易于管理。
- 通过 SQL Server 2000 所支持的 XML 可以访问 BLOB 数据，这将在 XML 流中返回 64 位编码描述的数据。

- 对包含了固定或可变长度的字符（包括宽字符）数据的列可以执行 SQL Server 全文本搜索（FTS）操作。也可以对包含在 image 字段中的已格式化的基于文本的数据--Word 或 Excel 文档--执行 FTS 操作。

将 BLOB数据写入到数据库中

下面的代码演示了如何利用 ADO.NET 将从某个文件获得的二进制数据写入 SQL Server image 字段中。

```
public void StorePicture( string filename )

{

  // Read the file into a byte array

  FileStream fs = new FileStream( filename, FileMode.Open,
FileAccess.Read );

  byte[] imageData = new Byte[fs.Length];

  fs.Read( imageData, 0, (int)fs.Length );

   fs.Close();



  SqlConnection conn = new SqlConnection("");

  SqlCommand cmd = new SqlCommand("StorePicture", conn);

  cmd.CommandType = CommandType.StoredProcedure;

  cmd.Parameters.Add("@filename", filename );

  cmd.Parameters["@filename"].Direction = ParameterDirection.Input;

  cmd.Parameters.Add("@blobdata", SqlDbType.Image);

   cmd.Parameters["@blobdata"].Direction = ParameterDirection.Input;

  // Store the byte array within the image field

  cmd.Parameters["@blobdata"].Value = imageData;

   try

   {

      conn.Open();
```

```
        cmd.ExecuteNonQuery();
    }
    catch
    {
        throw;
    }
    finally
    {
        conn.Close();
    }
}
```

从数据库中读取 BLOB 数据

在通过 ExecuteReader **方法创建** SqlDataReader **对象以读取包含** BLOB **数据的行时，需使用** CommandBehavior.SequentialAccess **枚举值。如果没有此枚举值，阅读器一次只从服务器中向客户端发送一行数据。如果行包含了** BOLB **数据，这预示着要占用大量内存。通过利用枚举值，就获得了更好的控制权，因为** BLOB **数据只在被引用时才被发出（例如，利用** GetBytes **方法，可以控制读取的字节数）。这在下面的代码片段中进行了演示。**

```
// Assume previously established command and connection
// The command SELECTs the IMAGE column from the table
conn.Open();
SqlDataReader reader =
cmd.ExecuteReader(CommandBehavior.SequentialAccess);
reader.Read();
// Get size of image data - pass null as the byte array parameter
long bytesize = reader.GetBytes(0, 0, null, 0, 0);
// Allocate byte array to hold image data
byte[] imageData = new byte[bytesize];
```

```
long bytesread = 0;

int curpos = 0;

while (bytesread < bytesize)

{

  // chunkSize is an arbitrary application defined value

  bytesread += reader.GetBytes(0, curpos, imageData, curpos, chunkSize);

   curpos += chunkSize;

}

// byte array 'imageData' now contains BLOB from database
```

注意使用 CommandBehavior.SequentialAccess 需要以严格的顺序访问列数据。例如，如果 BLOB 数据存在于第 3 列，并且还需要从第 1，2 列中读取数据，那么在读取第 3 列前必须先读取第 1，2 列。

事务处理

实际上所有用于更新数据源的面向商业的应用程序都需要事务处理支持。通过提供四个基本担保，即众所周知的首字缩写 ACID：可分性，一致性，分离性，和耐久性，事务处理将用于确保包含在一个或多个数据源中的系统的完整性。

例如，考虑一个基于 Web 的零售应用程序，它用于处理购买订单。每个订单需要 3 个完全不同操作，这些操作涉及到 3 个数据库更新：

- 库存水准必须减少所订购的数量。
- 所购买的量必须记入客户的信用等级。
- 新订单必须增加到数据库中。

这三个不同的操作作为一个单元并自动执行是至关重要的。三个操作必须全部成功，或都不成功--任何一个操作出现误差都将破坏数据完整性。事务处理提供了这种完整性及其它保证。

要进一步了解事务处理过程的基本原则，见 http://msdn.microsoft.com/library/en-us/cpguide/html/cpcontransactionprocessingfundamentals.asp。

可以采用很多方法将事务管理合并到数据访问代码中。每种方法适合下面两种基本编程模型之一。

- **手工事务处理**。可以直接在组件代码或存储过程中分别编写利用 ADO.NET 或 Transact-SQL 事务处理支持特性的代码。
- **自动化（COM+）事务处理**。可以向.NET 类中增加声明在运行时指定对象事务处理需要的属性。这种模型使你能方便地配置多个组件以使它们在同一事务处理内运行。

尽管自动化事务处理模型极大地简化了分布式事务处理过程，但两种模型都用于执行本地事务处理（即对单个资源管理器如 SQL Server 2000 执行的事务处理）或分布式事务处理（即，对位于远程计算机上的多个资源管理执行的事务处理）。

你也许会试图利用自动化（COM+）事务处理来从易于编程的模型中获益。在有多个组件执行数据库更新的系统中，这种优点更明显。然而，在很多情况下，应当避免这种事务处理模型所带来的额外开销和性能损失。

本节将指导你根据特定的应用程序环境选择最合适的模型。

选择事务处理模型

在选择事务处理模型前，首先应当考虑是否真正需要事务处理。事务处理是服务器应用程序使用的最昂贵的资源，在不必要使用的地方，它们降低了扩展性。考虑下面用于管理事务处理使用的准则：

- 只在需要跨一组操作获取锁并需要加强 ACID 规则时才执行事务处理。
- 尽可能短地保持事务处理，以最小化维持数据库锁的时间。
- 永远不要将客户放到事务处理生命周期的控制之中。
- 不要为单个 SQL 语句使用事务处理。SQL Server 自动把每个语句作为单个事务处理执行。

自动化事务处理与手工事务处理的对比

尽管编程模型已经对自动化事务处理进行了简化，特别是在多个组件执行数据库更新时，但本地事务处理总是相当快，因为它们不需要与微软 DTC 交互。即使你对单个本地资源管理器（如 SQL Server）使用自动化事务处理，也是这种情况（尽管性能损失减少了），因为手式本地事务处理避免了所有不必要的与 DTC 的进程间通信。

对于下面的情况，需使用手工事务处理：

- 对单个数据库执行事务处理。

对于下列情况，则宜使用自动事务处理：

- 需要将单个事务处理扩展到多个远程数据库时。
- 需要单个事务处理拥有多个资源管理器（如数据库和 Windows 2000 消息队列（被称为 MSMQ）资源管理器）时。

注意 避免混用事务处理模型。最好只使用其中一个。

在性能足够好的应用程序环境中，（甚至对于单个数据库）选择自动化事务处理以简化编程模型，这种做法是合理的。自动化事务处理使多个组件能很容易地执行现一事务处理中的多个操作。

使用手工事务处理

对于手工事务处理，可以直接在组件代码或存储过程中分别编写使用 ADO.NET 或 Transact-SQL 事务处理支持特性的代码。多数情况下，应选择在存储过程中控制事务处理，因为这种方法提供了更高的封装性，并且在性能方面，此方法与利用 ADO.NET 代码执行事务处理兼容。

利用 ADO.NET 执行手工事务处理

ADO.NET 支持事务处理对象，利用此对象可以开始新事务处理过程，并明确控制事务处理是否执行还是回滚。事务处理对象与单个数据库链接相关，可以通过链接对象的 BeginTransaction 方法获得。调用此方法并不是暗示，接下来的命令是在事务处理上下文中发出的。必须通过设置命令的 Transaction 属性，明确地将每个命令与事务处理关联起来。可以将多个命令对象与事务处理对象关联，因此在单个事务处理中就针对单个数据库把多个操作进行分组。

关于使用 ADO.NET 事务处理代码的示例，见附录中如何编码 ADO.NET 手工事务处理。

更多信息

- ADO.NET 手工事务处理的默认分离级别是读联锁，这意味着在读取数据时，数据库控制共享锁，但在事务处理结束前，数据可以被修改。这种情况潜在地会产生不可重复的读取或虚数据。通过将事务处理对象的 IsolationLevel 属性设置为 IsolationLevel 枚举类型所定义的一个枚举值，就可改变分离级别。
- 必须仔细为事务处理选择合适的分离级别。其折衷是数据一致性与性能的比例。最高的分离等级（被序列化了）提供了绝对的数据一致性，但是以系统整体吞吐量为代价。较低的分离等级会使应用程序更易于扩展，但同时增加了因数据不一致而导致出错的可能性。对多数时间读取数据、极少写入数据的系统来说，较低的分离等级是合适的。
- 关于选择恰当事务处理级别极有价值的信息，见微软出版社名为 Inside SQL Server 2000 的书，作者 Kalen Delaney。

利用存储过程执行手工事务处理

也可以在存储过程中使用 Transact-SQL 语句直接控制手工事务处理。例如，可以利用包含了 Transact-SQL 事务处理语句（如 BEGIN TRANSACTION、END TRANSACTION 及 ROLLBACK TRANSACTION）的存储过程执行事务处理。

更多信息

- 如果需要，可以在存储过程中使用 SET TRANSACTION ISOLATION LEVEL 语句控制事务处理的分离等级。读联锁是 SQL Server 的默认设置。关于 SQL Server 分离级别的更多信息，见 SQL Server 在线书目“访问和修改关系数据”一节中的分离级别部分。
- 关于演示如何利用 Transact-SQL 事务处理语句执行事务更新的代码示例，见附录中的如何利用 Transact-SQL 执行事务处理。

使用自动化事务

自动化事务简化了编程模型，因为它们不需要明确地开始新事务处理过程，或明确执行或取消事务。然而，自动化事务的最大优点是它们能与 DTC 结合起来，这就使单个事务可以扩展到多个分布式数据源中。在大

型分布式应用程序中，这个优点是很重要的。尽管通过手工对 DTC 直接编程来控制分布式事务是可能的，但自动化事务处理极大的简化了工作量，并且它是为基于组件的系统而设计的。例如，可以方便地以说明方式配置多个组件以执行包含了单个事务处理的任务。

自动化事务依赖于 COM+提供的分布式事务处理支持特性。结果，只有服务组件（即从 ServicedComponent 类中派生的组件）能够使用自动化事务。

要为自动化事务处理配置类，操作如下：

- 从位于 EnterpriseServices 名称空间的 ServicedComponent 类中派生新类。
- 通过 Transaction 属性定义类的事务处理需求。来自 TransactionOption 的枚举值决定了如何在 COM+类中配置类。可与此属性一同设置的其它属性包括事务处理分离等级和超时上限。
- 为了避免必须明确选出事务处理结果，可以用 AutoComplete 属性对方法进行注释。如果这些方法释放异常，事务将自动取消。注意，如果需要，仍可以直接挑选事务处理结果。更多详情，见本文稍后确定事务处理结果的节。

更多信息

- 关于 COM+自动化事务的更多信息，可在平台 SDK 文档中搜索“通过 COM+的自动化事务”获取。
- 关于.NE T 事务处理类的示例，见附录中的如何编码.NET 事务处理。

配置事务处理分离级别

用于 COM+1.0 版--即运行在 Windows 2000 中的 COM+--的事务处理分离级别被序列化了。这样做提供了最高的分离等级，却是以性能为代价的。系统的整体吞吐量被降低了。因为所涉及到的资源管理器（典型地是数据库）在事务处理期间必须保持读和写锁。在此期间，其它所有事务处理都被阻断了，这种情况将对应用程序的扩展能力产生极大冲击。

随微软 Windows .NET 发行的 COM+ 1.5 版允许有 COM+目录中按组件配置事务处理分离等级。与事务中根组件相关的设置决定了事务处理的分离等级。另外，同一事务流中的内部子组件拥有的事务处理等级必须不能高于要组件所定义的等级。如果不是这样，当子组件实例化时，将导致错误。

对.NET 管理类，Transaction 属性支持所有的公有 Isolation 属性。你可以用此属性陈述式地指定一特殊分离等级，如下面的代码所示：

```
[Transaction(TransactionOption.Supported,
Isolation=TransactionIsolationLevel.ReadCommitted)]

public class Account : ServicedComponent

{

  . . .

}
```

更多信息

关于配置事务处理分离等级及其它 Windows .NET COM+增强特性的更多信息，见 MSDN 杂志 2001 年 8 月期的“ Windows XP：利用 COM+ 1.5 的增强特性使你的组件更强壮”一文。

确定事务处理结果

在单个事务流的所有事务处理组件上下文中，自动化事务处理结果由事务取消标志和一致性标志的状态决定。当事务流中的根组件成为非活动状态（并且控制权返回调用者）时，确定事务处理结果。这种情况在图 5 中得到了演示，此图显示的是一个典型的银行基金传送事务。

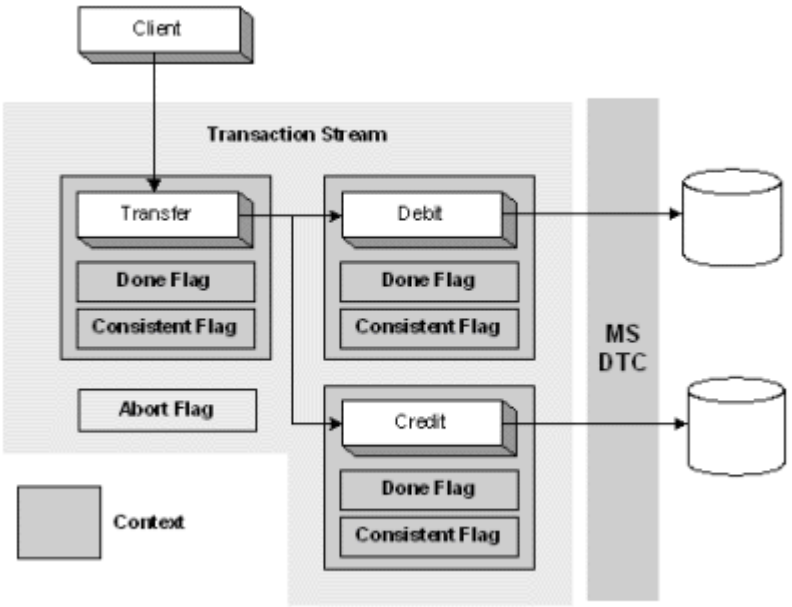

图 5 事务流上下文

当根对象（在本例中是对象）变为非活动状态，并且客户的方法调用返回时，确定事务处理结果。在任何上下文中的任何一致性标志被设为假，或如果事务处理取消标志设为真，那么底层的物理 DTC 事务将被取消。

可以以下面两种方式之一从.NET 对象中控制事务处理结果：

- 可以用 AutoComplete 属性对方法进行注释，并让.NET 自动存放将决定事务处理结果投票。如果方法释放异常，利用此属性，一致性标志自动地被设为假（此值最终使事务取消）。如果方法返回而没有释放异常，那么一致性标志将设为真，此值指出组件乐于执行事务。这并没有得到保证，因为它依赖于同一事务流中其它对象的投票。
- 可以调用 ContextUtil 类的静态方法 SetComplete 或 SetAbort，这些方法分别将一致性标志设为真或假。

严重性大于 10 的 SQL Server 错误将导致管理数据供应器释放 SqlException 类型的异常。如果方法缓存并处理异常，就要确保或者通过手工取消了事务，或者方法被标记了[AutoComplete]，以保证异常能传递回调用者。

AutoComplete 方法

对于标记了属性的方法，执行下面操作：

- **将** SqlException **传递加调用堆栈。**
- **将** SqlException **封装在外部例外中，并传递回调用者。也可以将异常封装在对调用者更有意义的异常类型中。**

异常如果不能传递，将导致对象不会提出取消事务，从而忽视数据库错误。这意味着共享同一事务流的其它对象的成功操作将被提交。

下面的代码缓存了 SqlException**，然后将它直接传递回调用者。事务处理最终将被取消，因为对象的一致性标志在对象变为非活动状态时自动被设为假。**

```
[AutoComplete]
void SomeMethod()
{
   try
   {
    // Open the connection, and perform database operation
     . . .
   }
  catch (SqlException sqlex )
   {
    LogException( sqlex ); // Log the exception details
     throw;                    // Rethrow the exception, causing the consistent
                               // flag to be set to false.
   }
   finally
   {
    // Close the database connection
     . . .
```

```
  }

}
```

Non-AutoComlete 方法

对于没有 AutoComplete **的属性的方法，必须：**

- **在** catch **块内调用** ContextUtil.SetAbort **以终止事务处理。这就将相容标志设置为假。**
- **如果没有发生异常事件，调用** ContextUtil.SetComplete**，以提交事务，这就将相容标志设置为真（缺省状态）。**

代码说明了这种方法。

```
void SomeOtherMethod()

{

   try

   {

    // Open the connection, and perform database operation

     . . .

      ContextUtil.SetComplete(); // Manually vote to commit the transaction

   }

  catch (SqlException sqlex)

   {

     LogException( sqlex );    // Log the exception details

     ContextUtil.SetAbort();   // Manually vote to abort the transaction

    // Exception is handled at this point and is not propagated to the caller

   }

   finally

   {

    // Close the database connection

     . . .
```

```
  }

}
```

注意 如果有多个 catch 块，在方法开始的时候调用 ContextVtil.SetAbort，以及在 try 块的末尾调用 ContextUtil.SetComplete 都会变得容易。用这种方法，就不需要在每个 catch 块中重复调用 ContextUtil.SetAbort。通过这种方法确定的相容标志的设置只在方法返回时有效。

对于异常事件（或循环异常），必须把它传递到调用堆栈中，因为这使得调用代码认为事务处理失败。它允许调用代码做出优化选择。比如，在银行资金转账中，如果债务操作失败，则转帐分支可以决定不执行债务操作。

如果把相容标志设置为假并且在返回时没有出现异常事件，则调用代码就没有办法知道事务处理是否一定失败。虽然可以返回 Boolean 值或设置 Boolean 输出参数，但还是应该前后一致，通过显示异常事件以表明有错误发生。这样代码就有一种标准的错误处理方法，因此更简明、更具有相容性。

数据分页

在分布式应用程序中利用数据进行分页是一项普遍的要求。比如，用户可能得到书的列表而该列表又不能够一次完全显示，用户就需要在数据上执行一些熟悉的操作，比如浏览下一页或上一页的数据，或者跳到列表的第一页或最后一页。

这部分内容将讨论实现这种功能的选项，以及每种选项在性能和缩放性上的效果。

选项比较

数据分页的选项有：

- 利用 SqlDataAdapter 的 Fill 方法，将来自查询处的结果填充到 DataSet 中。
- 通过 COM 的可相互操作性使用 ADO，并利用服务器光标。
- 利用存储的过程手工实现数据分页。

对数据进行分页的最优选项依赖于下列因素：

- 扩展性要求
- 性能要求
- 网络带宽
- 数据库服务器的存储器和功率
- 中级服务器的存储器和功率
- 由分页查询所返回的行数
- 数据总页数的大小

性能测试表明利用存储过程的手工方法在很大的应力水平范围上都提供了最佳性能。然而，由于手工方法在服务器上执行工作，如果大部分站点功能都依赖数据分页功能，那么服务器性能就会成一个关键要素。为确保这种方法能适合特殊环境，应该测试各种特殊要求的选项。

下面将讨论各种不同的选项。

使用 SqlDataAdapter

如前面所讨论的，SqlDataAdapter 是用来把来自数据库的数据填充到 DataSet 中，过载的 Fill 方法中的任一个都需要两个整数索引值（如下列代码所示）：

```
public int Fill(
    DataSet dataSet,
    int startRecord,
    int maxRecords,
    string srcTable
);
```

StartRecord 值标示从零开始的记录起始索引值。MaxRecord 值表示从 startRecord 开始的记录数，并将拷贝到新的 DataSet 中。

SqlDataAdapter 在内部利用 SqlDataReader 执行查询并返回结果。SqlDataAdapter 读取结果并创建基于来自 SalDataReader 的数据的 Dataset。SqlDataAdapter 通过 startRecord 和 maxRecords 把所有结果都拷贝到新生成的 DataSet 中，并丢弃不需要的数据。这意味着许多不必要的数据将潜在的通过网络进入数据访问客户--这是这种方法的主要缺陷。

比如，如果有 1000 个记录，而需要的是第 900 到 950 个记录，那么前面的 899 个记录将仍然穿越网络然后被丢弃。对于小数量的记录，这种开销可能是比较小的，但如果针对大量数据的分页，则这种开销就会非常巨大。

使用 ADO

实现分页的另一个选项是利用基于 COM 的 ADO 进行分页。这种方法的目标是获得访问服务器光标。服务器光标通过 ADO Recordset 对象显示。可以把 Recordset 光标的位置设置到 adUseServer 中。如果你的 OLE DB 供应器支持这种设置（如 SQLOLEDB 那样），就可以使用服务器光标。这样就可以利用光标直接导航到起始记录，而不需要将所有数据传过网络进入访问数据的用户代码中。

这种方法有下面两个缺点：

- 在大多数情况下，可能需要将返回到 Recordset 对象中的记录翻译成 DataSet 中的内容，以便在客户管理的代码中使用。虽然 OleDbDataAdapter 确实在获取 ADO Recordset 对象并把它翻译成 Dataset 时过载了 Fill 方法，但是并没有利用特殊记录进行开始与结束操作的功能。唯一现实的选项是把开始记录移动到 Recordset 对象中，循环每个记录，然后手工拷贝数据到手工生成的新 Dataset 中。这种操作，尤其是利用 COM Interop 调用，其优点可能不仅仅是不需要在网络上传输多余的数据，尤其对于小的 DataSet 更明显。

- 从服务器输出所需数据时，将保持连接和服务器光标开放。在数据库服务器上，光标的开放与维护需要昂贵的资源。虽然该选项提高了性能，但是由于为延长的时间两消耗服务器资源，从而也有可能降低可扩展性。

提供手工实现

在本部分中讨论的数据分页的最后一个选项是利用存储过程手工实现应用程序的分页功能。对于包含唯一关键字的表格，实现存储过程相对容易一些。而对于没有唯一关键字的表格（也不应该有许多关键字），该过程会相对复杂一些。

带有唯一关键字的表格的分页

如果表格包含一个唯一关键字，就可以利用 WHERE 条款中的关键字创建从某个特殊行起始的结果设置。这种方法，与用来限制结果设置大小的 SET ROWCOUNT 状态是相匹配的，提供了一种有效的分页原理。这一方法将在下面存储的代码中说明：

```
CREATE PROCEDURE GetProductsPaged

@lastProductID int,

@pageSize int

AS

SET ROWCOUNT @pageSize

SELECT *

FROM Products

WHERE [standard search criteria]

AND ProductID > @lastProductID

ORDER BY [Criteria that leaves ProductID monotonically increasing]

GO
```

这个存储过程的调用程序仅仅维护 LastProductID 的值，并通过所选的连续调用之间的页的大小增加或减小该值。

不带有唯一关键字的表格的分页

如果需要分页的表格没有唯一关键字，可以考虑添加一个--比如利用标识栏。这样就可以实现上面讨论的分页方案了。

只要能够通过结合结果记录中的两个或更多区域来产生唯一性，就仍然有可能实现无唯一关键字表格的有效分页方案。

比如，考察下列表格：

Col1	Col2	Col3	Other columns…
A	1	W	…
A	1	X	.
A	1	Y	.
A	1	Z	.
A	2	W	.
A	2	X	.
B	1	W	…
B	1	X	.

对于该表，结合 Col 、Col2 **和** Col3 **就可能产生一种唯一性。这样，就可以利用下面存储过程中的方法实现分布原理：**

```
CREATE PROCEDURE RetrieveDataPaged

@lastKey char(40),

@pageSize int

AS

SET ROWCOUNT @pageSize

SELECT

Col1, Col2, Col3, Col4, Col1+Col2+Col3 As KeyField

FROM SampleTable

WHERE [Standard search criteria]

AND Col1+Col2+Col3 > @lastKey

ORDER BY Col1 ASC, Col2 ASC, Col3 ASC

GO
```

客户保持存储过程返回的 keyField **栏的最后值，然后又插入回到存储过程中以控制表的分页。**

虽然手工实现增加了数据库服务器上的应变，但它避免了在网络上传输不必要的数据。性能测试表明在整个应变水平中这种方法都工作良好。然而，根据站点工作所涉及的数据分页功能的多少，在服务器上进行手工分页可能影响应用程序的可扩展性。应该在所在环境中运行性能测试，为应用程序找到最合适的方法。

附录

如何为一个.NET 类启用对象结构

要利用 Enterprise (COM+)Services **为对象结构启用**.NET **管理的类，需要执行下列步骤：**

- **从位于** System. Enterprise Services **名字空间中的** Serviced Component **中导出所需类。**

```
using System.EnterpriseServices;

public class DataAccessComponent : ServicedComponent
```

- **为该类添加** Construction Enabled **属性，并合理地指定缺省结构字符串，该缺省值保存在** COM+**目录中，管理员可以利用组件服务微软管理控制台（**MNC**）的** snap-in **来维护该缺省值。**

```
[ConstructionEnabled(Default="default DSN")]

public class DataAccessComponent : ServicedComponent
```

- **提供虚拟** Construct **方法的替换实现方案。该方法在对象语言构造程序之后调用。在** COM **目录中保存的结构字符串是该方法的唯一字符串。**

```
public override void Construct( string constructString )
{
  // Construct method is called next after constructor.
   // The configured DSN is supplied as the single argument
}
```

- **通过** Assembly key **文件或** Assembly key Name **属性为该汇编提供一个强名字。任何用** COM+**服务注册的汇编必须有一个强名字。关于带有强名字汇编的更多信息，参考：** http://msdn.microsoft.com/library/en-us/cpguide/html/cpconworkingwithstrongly-namedassemblies.Asp。

```
[assembly: AssemblyKeyFile("DataServices.snk")]
```

- **为支持动态注册，可以利用汇编层上的属性** ApplicationName **和** Application Action **分别指定用于保持汇编元素和应用程序动作类型的** COM+**应用程序的名字。关于汇编注册的更多信息，参考：** http://msdn.microsoft.com/library/en-us/cpguide/html/cpconregisteringserviced components.asp 。

```
// the ApplicationName attribute specifies the name of the
// COM+ Application which will hold assembly components
[assembly : ApplicationName("DataServices")]
```

```
// the ApplicationActivation.ActivationOption attribute specifies

// where assembly components are loaded on activation

// Library : components run in the creator's process

// Server : components run in a system process, dllhost.exe

[assembly: ApplicationActivation(ActivationOption.Library)]
```

下列代码段是一个叫做 DataAccessComponent **的服务组件，它利用** COM+**结构字符串来获得数据库连接字符串。**

```
using System;

using System.EnterpriseServices;


// the ApplicationName attribute specifies the name of the

// COM+ Application which will hold assembly components

[assembly : ApplicationName("DataServices")]


// the ApplicationActivation.ActivationOption attribute specifies

// where assembly components are loaded on activation

// Library : components run in the creator's process

// Server : components run in a system process, dllhost.exe

[assembly: ApplicationActivation(ActivationOption.Library)]


// Sign the assembly. The snk key file is created using the

// sn.exe utility

[assembly: AssemblyKeyFile("DataServices.snk")]


[ConstructionEnabled(Default="Default DSN")]

public class DataAccessComponent : ServicedComponent
```

```
{
    private string connectionString;
    public DataAccessComponent()
    {
     // constructor is called on instance creation
    }
   public override void Construct( string constructString )
    {
     // Construct method is called next after constructor.
     // The configured DSN is supplied as the single argument
       this.connectionString = constructString;
    }
}
```

如何利用 SqlDataAdapter 来检索多个行

下面的代码说明如何利用 SqlDataAdapter **对象发出一个生成** Data Set **或** Datatable **的命令。它从** SQL Server Northwind **数据库中检索一系列产品目录。**

```
using System.Data;
using System.Data.SqlClient;


public DataTable RetrieveRowsWithDataTable()
{
  using ( SqlConnection conn = new SqlConnection(connectionString) )
   {
    SqlCommand cmd = new SqlCommand("DATRetrieveProducts", conn);
    cmd.CommandType = CommandType.StoredProcedure;
```

```
    SqlDataAdapter da = new SqlDataAdapter( cmd );
    DataTable dt = new DataTable("Products");
     da.Fill(dt);
     return dt;
   }
}
```

按下列步骤利用 SqlAdapter 生成 DataSet 或 DataTable：

- 创建 SqlCommand 对象启用存储过程，并把它与 SqlConnection 对象（显示的）或连接字符串（未显示）相联系。
- 创建一个新的 SqlDataAdapter 对象，并把它 SqlCommand 对象相联系。
- 创建 DataTable(或者 DataSet)对象。利用构造程序自变量命名 DataTable.
- 调用 SqlData Adapter 对象的 Fill 方法，把检索的行转移到 DataSet 或 Datatable 中。

如何利用 SqlDataReader 检索多个行

下列代码说明了如何利用 SqlDataReader 方法检索多行：

```
using System.IO;
using System.Data;
using System.Data.SqlClient;

public SqlDataReader RetrieveRowsWithDataReader()
{
  SqlConnection conn = new SqlConnection(
            "server=(local);Integrated Security=SSPI;database=northwind");
  SqlCommand cmd = new SqlCommand("DATRetrieveProducts", conn );
  cmd.CommandType = CommandType.StoredProcedure;
   try
   {
     conn.Open();
```

```
    // Generate the reader. CommandBehavior.CloseConnection causes the
    // the connection to be closed when the reader object is closed
    return( cmd.ExecuteReader( CommandBehavior.CloseConnection ) );
  }
  catch
  {
     conn.Close();
     throw;
  }
}

// Display the product list using the console
private void DisplayProducts()
{
  SqlDataReader reader = RetrieveRowsWithDataReader();
  while (reader.Read())
  {
    Console.WriteLine("{0} {1} {2}",
                      reader.GetInt32(0).ToString(),
                      reader.GetString(1) );
  }
  reader.Close(); // Also closes the connection due to the
                  // CommandBehavior enum used when generating the reader
}
```

按下列步骤利用 SqlDataReader **检索多行：**

- 创建用于执行存储的过程的 SqlCommand 对象，并把它与 SqlConnection 对象相联系。
- 打开链接。
- 通过调用 SqlCommand 对象的 Excute Reader 方法生成 SqlDataReader 对象。
- 从流中读取数据，调用 SqlDataReader 对象的 Read 方法来检索行，并利用分类的存取程序方法（如 GetIut 32 和 Get String 方法）检索列的值。
- 完成读取后，调用 Close 方法。

如何利用 XmlReader 检索多个行

可以利用 SqlCommand 对象生成 XmlReader 对象，它提供对 XML 数据的基于流的前向访问。该命令（通常是一个存储的过程）必须生成一个基于 XML 的结果设置，它对于 SQL Server2000 通常是由带有有效条款 FOR XML 的 SELECT 状态组成。下列代码段说明了这种方法：

```
public void RetrieveAndDisplayRowsWithXmlReader()

{

   SqlConnection conn = new SqlConnection(connectionString);

  SqlCommand cmd = new SqlCommand("DATRetrieveProductsXML",
conn );

  cmd.CommandType = CommandType.StoredProcedure;

   try

   {

      conn.Open();

     XmlTextReader xreader = (XmlTextReader)cmd.ExecuteXmlReader();

     while ( xreader.Read() )

      {

       if ( xreader.Name == "PRODUCTS" )

         {

           string strOutput = xreader.GetAttribute("ProductID");

           strOutput += " ";

            strOutput += xreader.GetAttribute("ProductName");

             Console.WriteLine( strOutput );

         }
```

```
        }
        xreader.Close();
    }
    catch
    {
        throw;
    }
    finally
    {
        conn.Close();
    }
}
```

上述代码使用了下列存储过程：

```
CREATE PROCEDURE DATRetrieveProductsXML
AS
SELECT * FROM PRODUCTS
FOR XML AUTO
GO
```

按下列步骤检索 XML **数据：**

- **创建** SqlCommand **对象启用生成** XML **结果设置的过程。(比如,利用** SELECT **状态中的** FOR XML **条款）。把** SqlCommand **对象与一个链接相联系。**
- **调用** SqlCommand **对象的** ExecuteXmlReader **方法，并把结果分配给前向对象** XmlTextReader**。当不需要任何返回数据的基于** XML **的验证时，这是应该使用的最快类型的** XmlReader **对象。**
- **利用** XmlTextReader **对象的** Read **方法读取数据。**

如何利用存储过程输出参数检索单个行

可以调用一个存储过程，它通过一种称做输出参数的方式可以在单个行中返回检索数据项。下列代码段利用存储的过程检索产品的名称和单价，该产品包含在 Northwind **数据库中。**

```
void GetProductDetails( int ProductID,
                                    out string ProductName, out decimal UnitPrice )
{
  SqlConnection conn = new SqlConnection(
            "server=(local);Integrated Security=SSPI;database=Northwind");


  // Set up the command object used to execute the stored proc
  SqlCommand cmd = new SqlCommand( "DATGetProductDetailsSPOutput",
conn );
  cmd.CommandType = CommandType.StoredProcedure;
  // Establish stored proc parameters.
   //   @ProductID int INPUT
   //   @ProductName nvarchar(40) OUTPUT
   //   @UnitPrice money OUTPUT


  // Must explicitly set the direction of output parameters
  SqlParameter paramProdID =
            cmd.Parameters.Add( "@ProductID", ProductID );
   paramProdID.Direction = ParameterDirection.Input;
  SqlParameter paramProdName =
            cmd.Parameters.Add( "@ProductName", SqlDbType.VarChar, 40 );
  paramProdName.Direction = ParameterDirection.Output;
  SqlParameter paramUnitPrice =
             cmd.Parameters.Add( "@UnitPrice", SqlDbType.Money );
  paramUnitPrice.Direction = ParameterDirection.Output;
   try
```

```
  {
     conn.Open();
   // Use ExecuteNonQuery to run the command.
   // Although no rows are returned any mapped output parameters
    // (and potentially return values) are populated
     cmd.ExecuteNonQuery( );
   // Return output parameters from stored proc
    ProductName = paramProdName.Value.ToString();
    UnitPrice = (decimal)paramUnitPrice.Value;
  }
  catch
  {
     throw;
  }
  finally
  {
     conn.Close();
  }
}
```

按下列步骤利用存储的过程输出参数检索单个行：

- **创建一个** SqlCommand **对象，并把它与** SqlConnection **对象相联系。**
- **通过调用** SqlCommand’s Parameters **集合的** Add **方法设置存储过程参数。缺省情况下，参数假定为输出参数，所以必须明确设置任何输出参数的方向。**

注意 明确设置所有参数的方向是一次很好的练习，包括输入参数。

- **打开连接。**
- **调用** Sqlcommand **对象的** ExecuteNonQuery **方法。它在输出参数（并潜在地带有一个返回值）中。**
- **利用** Value **属性从合适的** SqlParameter **对象中检索输出参数。**

- **关闭连接。**

上述代码段启用了下列存储过程。

```
CREATE PROCEDURE DATGetProductDetailsSPOutput

@ProductID int,

@ProductName nvarchar(40) OUTPUT,

@UnitPrice money OUTPUT

AS

SELECT @ProductName = ProductName,

        @UnitPrice  =  UnitPrice

FROM Products

WHERE ProductID = @ProductID

GO
```

如何利用 SqlDataReader检索单个行

可以利用 SqlDataReader **对象检索单个行，以及来自返回数据流的所需栏的值。这由下列代码说明：**

```
void GetProductDetailsUsingReader( int ProductID,

                                   out string ProductName, out decimal UnitPrice )

{

   SqlConnection  conn = new SqlConnection(

              "server=(local);Integrated  Security=SSPI;database=Northwind");



  // Set up the command object used to execute the stored proc

  SqlCommand cmd = new SqlCommand( "DATGetProductDetailsReader",
conn );

   cmd.CommandType = CommandType.StoredProcedure;

  // Establish stored proc parameters.
```

```
  //   @ProductID int INPUT



  SqlParameter paramProdID = cmd.Parameters.Add( "@ProductID",
ProductID );

  paramProdID.Direction = ParameterDirection.Input;

   try

   {

       conn.Open();

       SqlDataReader reader = cmd.ExecuteReader();

    reader.Read(); // Advance to the one and only row



    // Return output parameters from returned data stream

      ProductName = reader.GetString(0);

      UnitPrice = reader.GetDecimal(1);

       reader.Close();

   }

   catch

   {

       throw;

   }

   finally

   {

       conn.Close();

   }

}
```

按下列步骤返回带有 SqlDataReader **对象：**

- 建立 SqlCommand 对象。
- 打开连接。
- 调用 SqlDReader 对象的 ExecuteReader 对象。
- 利用 SqlDataReader 对象的分类的存取程序方法检索输出参数--在这里是 GetString 和 GetDecimal.

上述代码段启用了下列存储过程：

```
CREATE PROCEDURE DATGetProductDetailsReader

@ProductID int

AS

SELECT ProductName, UnitPrice FROM Products

WHERE ProductID = @ProductID

GO
```

如何利用 ExecuteScalar 单个项

ExecuteScalar 方法是设计成用于返回单个值的访问。在返回多列或多行的访问事件中，ExecuteScalar 只返回第一行的第一例。

下列代码说明如何查询某个产品 ID 的产品名称：

```
void GetProductNameExecuteScalar( int ProductID, out string ProductName )

{

  SqlConnection conn = new SqlConnection(

              "server=(local);Integrated  Security=SSPI;database=northwind");

  SqlCommand cmd = new SqlCommand("LookupProductNameScalar",
conn );

  cmd.CommandType = CommandType.StoredProcedure;



  cmd.Parameters.Add("@ProductID", ProductID );

   try

   {

      conn.Open();
```

```
    ProductName = (string)cmd.ExecuteScalar();
  }
  catch
  {
    throw;
  }
  finally
  {
    conn.Close();
  }
}
```

按下列步骤利用 Execute Scalar 检索单个项：

- **建立调用存储过程的** SqlCommand **对象。**
- **打开链接。**
- **调用** ExecuteScalar **方法，注意该方法返回对象类型。它包含检索的第一列的值，并且必须设计成合适的类型。**
- **关闭链接。**

上述代码启用了下列存储过程：

```
CREATE PROCEDURE LookupProductNameScalar
@ProductID int
AS
SELECT TOP 1 ProductName
FROM Products
WHERE ProductID = @ProductID
GO
```

如何利用存储过程输出或返回的参数检索单个项

利用存储过程输出或返回的参数可以查询单个值，下列代码说明了输出参数的使用：

```
void GetProductNameUsingSPOutput( int ProductID, out string ProductName )

{

  SqlConnection conn = new SqlConnection(

            "server=(local);Integrated  Security=SSPI;database=northwind");

  SqlCommand cmd = new SqlCommand("LookupProductNameSPOutput",
conn );

  cmd.CommandType = CommandType.StoredProcedure;



  SqlParameter paramProdID = cmd.Parameters.Add("@ProductID",
ProductID );

  ParamProdID.Direction = ParameterDirection.Input;

  SqlParameter paramPN =

              cmd.Parameters.Add("@ProductName", SqlDbType.VarChar, 40 );

  paramPN.Direction = ParameterDirection.Output;

   try

   {

      conn.Open();

      cmd.ExecuteNonQuery();

     ProductName = paramPN.Value.ToString();

   }

   catch

   {

      throw;

   }

   finally

   {
```

```
      conn.Close();

   }

}
```

按下列步骤利用存储过程的输出参数检索单个值：

- 创建调用存储过程的 SqlCommand 对象。
- 通过把 SqlParmeters 添加到 SqlCommand’s Parameters 集合中设置任何输入参数和单个输出参数。
- 打开链接。
- 调用 SqlCommand 对象的 Execute NonQuery 方法。
- 关闭链接。
- 利用输出 SqlParameter 的 Value 属性检索输出值。

上述代码使用了下列存储过程：

```
CREATE PROCEDURE LookupProductNameSPOutput

@ProductID int,

@ProductName nvarchar(40) OUTPUT

AS

SELECT @ProductName = ProductName

FROM Products

WHERE ProductID = @ProductID

GO
```

下列代码说明如何利用返回值确定是否存在特殊行。从编码的角度看，这与使用存储过程输出参数相类似，除了需要明确设置到 ParameterDirection.ReturnValue 的 SqlParameter 方向。

```
bool CheckProduct( int ProductID )

{

  SqlConnection conn = new SqlConnection(

          "server=(local);Integrated  Security=SSPI;database=northwind");

  SqlCommand cmd = new SqlCommand("CheckProductSP", conn );

  cmd.CommandType = CommandType.StoredProcedure;
```

```
 cmd.Parameters.Add("@ProductID", ProductID );
SqlParameter paramRet =
            cmd.Parameters.Add("@ProductExists", SqlDbType.Int );
paramRet.Direction = ParameterDirection.ReturnValue;
 try
 {
    conn.Open();
    cmd.ExecuteNonQuery();
 }
 catch
 {
    throw;
 }
 finally
 {
    conn.Close();
 }
return (int)paramRet.Value == 1;
}
```

按下列步骤，可以利用存储过程返回值检查是否存在特殊行：

- **建立调用存储过程的** SqlCommand **对象。**
- **设置包含需要访问的行的主要关键字的输入参数。**
- **设置单个返回值参数。把** SqlParameter **对象添加到** SqlCommand’s Parameter **集合中，并设置它到** ParameterDireetion.ReturnValue **的方面。**
- **打开链接。**
- **调用** SqlCommand **对象的** ExecuteNonQuery **的方法**.
- **关闭链接。**

- **利用返回值** SqlParameter **的** Value **属性检索返回值。**

上述代码使用了下列存储过程：

```
CREATE PROCEDURE CheckProductSP

@ProductID int

AS

IF EXISTS( SELECT ProductID

                FROM  Products

               WHERE  ProductID  =  @ProductID  )

   return 1

ELSE

   return  0

GO
```

如何利用 SqlDataReader检索单个项。

通过调用命令对象的 ExecuteReader **方法，可以利用** SqlDataReader **对象获得单个输出值。这需要稍微多一些的代码，因为** SqlDataReader Read **方法必须调用，然后所需值通过读者存取程序方法得到检索。** SqlDataReader **对象的使用在下列代码中说明：**

```
bool CheckProductWithReader( int ProductID )

{

   SqlConnection  conn = new SqlConnection(

              "server=(local);Integrated  Security=SSPI;database=northwind");

  SqlCommand cmd = new SqlCommand("CheckProductExistsWithCount",
conn );

   cmd.CommandType = CommandType.StoredProcedure;



   cmd.Parameters.Add("@ProductID",  ProductID );

   cmd.Parameters["@ProductID"].Direction = ParameterDirection.Input;
```

```
    try
    {
        conn.Open();
      SqlDataReader reader = cmd.ExecuteReader(
                                                CommandBehavior.SingleResult );
        reader.Read();


       bool bRecordExists = reader.GetInt32(0) > 0;
        reader.Close();
       return bRecordExists;
    }
    catch
    {
        throw;
    }
    finally
    {
       conn.Close();
    }


}
```

上述代码使用了下列存储过程：

```
CREATE PROCEDURE CheckProductExistsWithCount

@ProductID int

AS
```

```
SELECT COUNT(*) FROM Products

WHERE ProductID = @ProductID

GO
```

如何编码 ADO.NET 手工事务

下列代码说明如何利用 SQL Server. NET **数据供应器提供的事务支持来保护事务的支金转帐操作。该操作在位于同一数据库中的两个帐户之间转移支金。**

```
public void TransferMoney( string toAccount, string fromAccount, decimal
amount )

{

  using ( SqlConnection conn = new SqlConnection(

                  "server=(local);Integrated
Security=SSPI;database=SimpleBank" ) )

   {

    SqlCommand cmdCredit = new SqlCommand("Credit", conn );

     cmdCredit.CommandType = CommandType.StoredProcedure;

     cmdCredit.Parameters.Add( new SqlParameter("@AccountNo",
toAccount) );

    cmdCredit.Parameters.Add( new SqlParameter("@Amount", amount ));



    SqlCommand cmdDebit = new SqlCommand("Debit", conn );

     cmdDebit.CommandType = CommandType.StoredProcedure;

     cmdDebit.Parameters.Add( new SqlParameter("@AccountNo",
fromAccount) );

    cmdDebit.Parameters.Add( new SqlParameter("@Amount", amount ));



      conn.Open();

    // Start a new transaction
```

```
    using ( SqlTransaction trans = conn.BeginTransaction() )
     {
      // Associate the two command objects with the same transaction
       cmdCredit.Transaction = trans;
       cmdDebit.Transaction = trans;
        try
        {
           cmdCredit.ExecuteNonQuery();
           cmdDebit.ExecuteNonQuery();
        // Both commands (credit and debit) were successful
           trans.Commit();
        }
       catch( Exception ex )
        {
          // transaction failed
           trans.Rollback();
        // log exception details . . .
          throw ex;
        }
     }
  }
}
```

如何利用 Transact-SQL 执行事务

下列存储过程说明了如何在 Transact-SQL **过程内执行事务的支金转移操作。**

```
CREATE PROCEDURE MoneyTransfer
```

```
@FromAccount char(20),

@ToAccount char(20),

@Amount money

AS

BEGIN TRANSACTION

-- PERFORM DEBIT OPERATION

UPDATE Accounts

SET Balance = Balance - @Amount

WHERE AccountNumber = @FromAccount

IF @@RowCount = 0

BEGIN

   RAISERROR('Invalid From Account Number', 11, 1)

  GOTO  ABORT

END

DECLARE @Balance money

SELECT @Balance = Balance FROM ACCOUNTS

WHERE AccountNumber = @FromAccount

IF @BALANCE < 0

BEGIN

  RAISERROR('Insufficient funds', 11, 1)

  GOTO  ABORT

END

-- PERFORM CREDIT OPERATION

UPDATE Accounts

SET Balance = Balance + @Amount
```

```
WHERE AccountNumber = @ToAccount

IF @@RowCount = 0

BEGIN

  RAISERROR('Invalid To Account Number', 11, 1)

   GOTO  ABORT

END

COMMIT TRANSACTION

RETURN 0

ABORT:

   ROLLBACK  TRANSACTION

GO
```

该存储过程使用 BEGIN TRANSACTION， COMMIT TRANSACTION，**和** ROLLBACK TRANSACTION **状态手工控制事务。**

如何编码事务性的.NET 类

下述例子是三种服务性的 NET **类，它们配置或用于自动事务。每个类都带有** Transaction **属性，它的值将决定是否启动新事务流或者对象是否共享即时调用程序的数据流。这些元素一起工作来执行银行支金转移。** Transfer **类配置有** RequiresNew **事务属性，而** Debit **和** Credit **类配置有** Required **属性。这样，在运行的时候三个对象共享同一个事务。**

```
using System;

using System.EnterpriseServices;



[Transaction(TransactionOption.RequiresNew)]

public class Transfer : ServicedComponent

{

    [AutoComplete]

   public void Transfer( string toAccount,

                                        string fromAccount, decimal amount )
```

```
  {
     try
     {
      // Perform the debit operation
      Debit debit = new Debit();
       debit.DebitAccount( fromAccount, amount );
      // Perform the credit operation
      Credit credit = new Credit();
       credit.CreditAccount( toAccount, amount );
     }
    catch( SqlException sqlex )
     {
      // Handle and log exception details
      // Wrap and propagate the exception
       throw new TransferException( "Transfer Failure", sqlex );
     }
  }
}
[Transaction(TransactionOption.Required)]
public class Credit : ServicedComponent
{
   [AutoComplete]
  public void CreditAccount( string account, decimal amount )
   {
    SqlConnection conn = new SqlConnection(
```

```
                "Server=(local); Integrated Security=SSPI";
database="SimpleBank");

      SqlCommand cmd = new SqlCommand("Credit", conn );

     cmd.CommandType = CommandType.StoredProcedure;

     cmd.Parameters.Add( new SqlParameter("@AccountNo", account) );

     cmd.Parameters.Add( new SqlParameter("@Amount", amount ));

      try

      {

         conn.Open();

         cmd.ExecuteNonQuery();

      }

     catch (SqlException sqlex)

      {

        // Log exception details here

        throw; // Propagate exception

      }

   }

}

[Transaction(TransactionOption.Required)]

public class Debit : ServicedComponent

{

   public void DebitAccount( string account, decimal amount )

   {

     SqlConnection conn = new SqlConnection(

                "Server=(local); Integrated Security=SSPI";
database="SimpleBank");
```

```
    SqlCommand cmd = new SqlCommand("Debit", conn );
     cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add( new SqlParameter("@AccountNo", account) );
    cmd.Parameters.Add( new SqlParameter("@Amount", amount ));
      try
      {
         conn.Open();
         cmd.ExecuteNonQuery();
      }
     catch (SqlException sqlex)
      {
        // Log exception details here
       throw; // Propagate exception back to caller
      }
   }
}
```

合作者

非常感谢下列撰稿者和审校者：

Bill Vaughn, Mike Pizzo, Doug Rothaus, Kevin White, Blaine Dokter, David Schleifer, Graeme Malcolm**（内容专家）**, Bernard Chen**（西班牙人）**, Matt Drucke**（协调）和** Steve kirk.

读者有什么样的问题、评论和建议？关于本文的反馈信息，请发 E-mail **至** devfdbck®microsoft.com。

你希望学习并利用.NET **的强大功能吗？与微软技术中心的技术专家一起工作，学习开发最佳方案。详细信息请访问：** http://www.micrsoft.com/business/services/mtc.asp。