

.NET 数据访问体系结构指南

摘要： 本文提供有关在基于 .NET 的多层应用程序中实现基于 ADO.NET 的数据访问层的指南。它集中讨论一系列常见的数据访问任务和方案，并介绍了可帮助您选择最适合的方法和技术的指南。

简介

如果您要为基于 .NET 的应用程序设计数据访问层，应该使用 Microsoft®ADO.NET 作为数据访问模型。ADO.NET 功能丰富，支持松耦合的多层 Web 应用程序和 Web 服务的数据访问要求。像其他功能丰富的对象模型一样，ADO.NET 提供了多种方法来解决特定问题。

《.NET 数据访问体系结构指南》提供的信息可帮助您选择最适合的数据访问方法。它通过描述一系列广泛的常见数据访问方案、提供性能提示以及推荐最佳实施策略来达到此目的。本指南还提供了对常见问题的解答，这些问题包括：哪里是存储数据库连接字符串的最佳场所？如何实现连接池？如何处理事务？如何实现分页以使用户能够在大量记录中滚动？

本指南集中讨论如何使用 ADO.NET 并通过 SQL Server .NET 数据提供程序（ADO.NET 随附的两个数据提供程序之一）来访问 Microsoft SQL®Server®2000。在适当的时候，本指南将明确指出您在使用 OLE DB .NET 数据提供程序来访问其他支持 OLE DB 的数据源时应该了解的差异。

有关使用本文中讨论的指南和最佳实施策略开发的数据访问组件的具体实现，请参阅 [Data Access Application Block](#)。Data Access Application Block 包含该实现的源代码，您可以在基于 .NET 的应用程序中直接使用这些代码。

《.NET 数据访问体系结构指南》分为以下几个部分：

ADO.NET 简介

- 管理数据库连接

- 错误处理

- 性能

- 通过防火墙进行连接

- 处理 BLOB

- 通过数据集执行数据库更新

- 使用强类型数据集对象

- 处理空数据字段

- 事务处理

- 数据分页

- 附录

本文档的目标读者

本文档为需要生成基于 .NET 的应用程序的应用程序设计人员和企业开发人员提供指南。如果您负责设计和开发基于 .NET 的多层应用程序的数据层，请阅读本文档。

预备知识

要使用本指南来生成基于 .NET 的应用程序，您必须具有使用 ActiveX?Data Objects (ADO) 和/或 OLE DB 开发数据访问代码以及 SQL Server 方面的经验。您必须了解如何为 .NET 平台开发托管代码，并且必须了解 ADO.NET 数据访问模型引入的根本性的变化。有关 .NET 开发的详细信息，请参阅 <http://msdn.microsoft.com/net>。

新增功能

本文档已经进行了更新，以便包含有关执行数据集更新、使用类型化 DataSet 以及使用空数据字段的部分。

如正文中所指出的那样，本指南中的一些内容特别适用于 Microsoft Visual Studio?2003 开发系统和 .NET 框架 SDK 1.1 版。

下载《.NET 数据访问体系结构指南》

单击可从 [MS.com 下载中心](#) 下载《.NET 数据访问体系结构指南》

ADO.NET 简介

ADO.NET 是基于 .NET 的应用程序的数据访问模型。可以使用它来访问关系数据库系统（如 SQL Server 2000、Oracle）和其他许多具有 OLE DB 或 ODBC 提供程序的数据源。在某种程度上，ADO.NET 代表 ADO 技术的最新进展。不过，ADO.NET 引入了一些重大变化和革新，旨在解决 Web 应用程序的松耦合特性以及在本质上互不关联的特性。有关 ADO 与 ADO.NET 的比较，请参阅 MSDN 文章“ADO.NET for the ADO Programmer”，网址为：

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/adonetprogmsdn.asp>。

ADO.NET 引入的主要变化之一是用 DataTable、DataSet、DataAdapter 和 DataReader 对象的组合取代了 ADO Recordset 对象。DataTable 表示单个表中行的集合，在这一方面类似于 Recordset。DataSet 表示 DataTable 对象的集合，同时包括将各种表绑定在一起的关系和约束。实际上，DataSet 是带有内置 XML 支持的、内存中的关系结构。

DataSet 的主要特性之一是它不了解可能用来填充它的基础数据源。它是一个不连续的、独立的实体，用于表示数据集合，并且可以通过多层应用程序的不同层在组件之间传递。它还可以作为 XML 数据流进行序列化，这使其非常适合于在

不同种类的平台之间进行数据传输。ADO.NET 使用 **DataAdapter** 对象将数据传送到 **DataSet** 和基础数据源，或者从数据源传出。**DataAdapter** 对象还提供以前与 **Recordset** 关联的增强的批量更新功能。

图 1 显示了完整的 **DataSet** 对象模型。

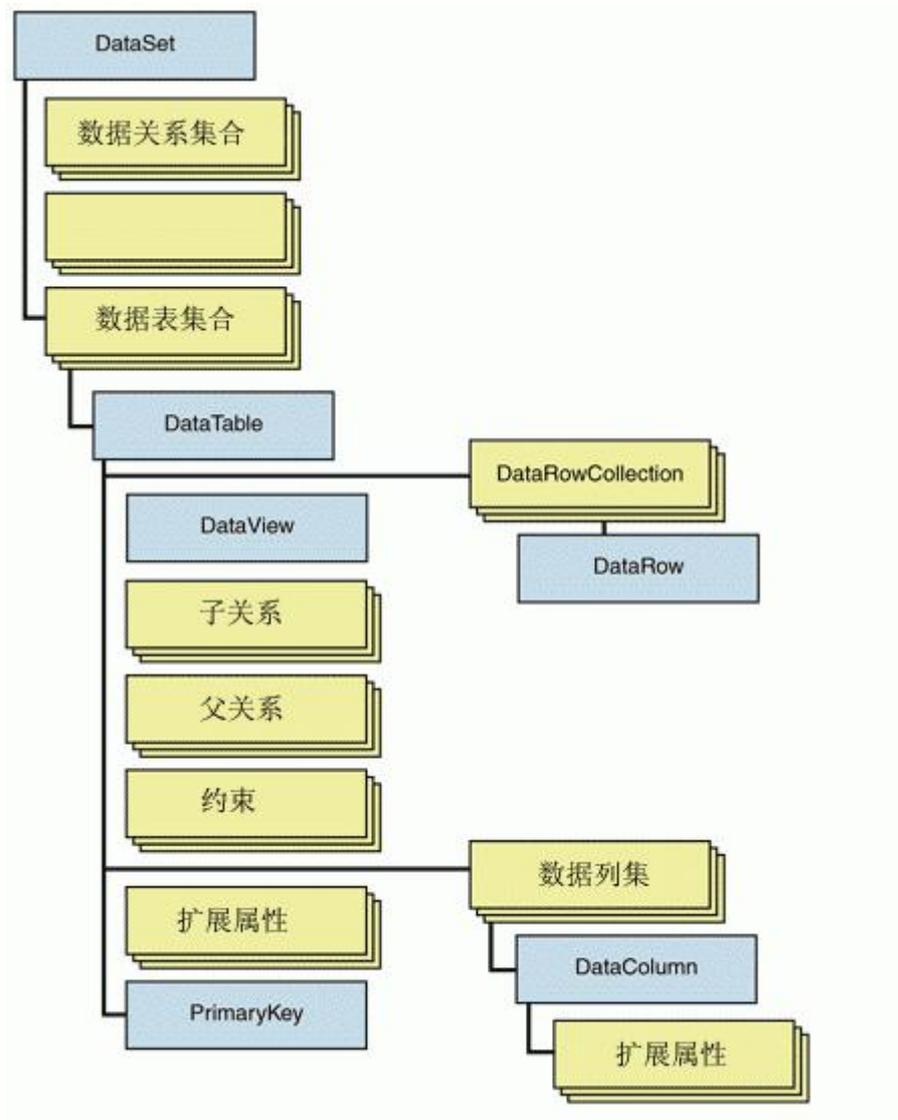


图 1.1.DataSet 对象模型

.NET 数据提供程序

ADO.NET 依赖于 .NET 数据提供程序的服务。这些提供程序提供对基础数据源的访问，并且包括四个主要对象（**Connection**、**Command**、**DataReader** 和 **DataAdapter**）。

目前，ADO.NET 随附了两类提供程序：Bridge 提供程序和 Native 提供程序。通过 Bridge 提供程序（如那些为 OLE DB 和 ODBC 提供的提供程序），可以使

用为以前的数据访问技术设计的数据库。Native 提供程序（如 SQL Server 和 Oracle 提供程序）通常能够提供性能方面的改善，部分原因在于少了一个抽象层。

- **SQL Server .NET 数据提供程序。**这是一个用于 Microsoft SQL Server 7.0 和更高版本数据库的提供程序。它被进行了优化以便访问 SQL Server，并且它通过使用 SQL Server 的本机数据传输协议来直接与 SQL Server 进行通讯。

当您连接到 SQL Server 7.0 或 SQL Server 2000 时，请始终使用该提供程序。

- **Oracle.NET 数据提供程序。**用于 Oracle 的 .NET 框架数据提供程序通过 Oracle 客户端连接软件支持对 Oracle 数据源的数据访问。该数据提供程序支持 Oracle 客户端软件版本 8.1.7 及更高版本。
- **OLE DB .NET 数据提供程序。**这是一个用于 OLE DB 数据源的托管提供程序。它的效率要比 SQL Server .NET 数据提供程序稍微低一些，因为它在与数据库通讯时通过 OLE DB 层进行调用。请注意，该提供程序不支持用于开放式数据库连接 (ODBC) 的 OLE DB 提供程序 MSDASQL。对于 ODBC 数据源，请改为使用 ODBC.NET 数据提供程序（稍后将加以介绍）。有关与 ADO.NET 兼容的 OLE DB 提供程序的列表，请参阅 <http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconadonetproviders.asp>。

其他目前正处于测试阶段的 .NET 数据提供程序包括：

- **ODBC.NET 数据提供程序。**用于 ODBC 的 .NET 框架数据提供程序使用本机 ODBC 驱动程序管理器 (DM) 来支持借助于 COM 互操作性进行的数据访问。
- **用于从 SQL Server 2000 检索 XML 的托管提供程序。**XML for SQL Server Web update 2（目前正处于测试阶段）包含一个托管提供程序，专门用于从 SQL Server 2000 中检索 XML。有关此更新的详细信息，请参阅 <http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28001300>。

有关不同数据提供程序的详细概述，请参阅《.NET 框架开发人员指南》中的“**.NET 框架数据提供程序**”，网址为：

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconadonetproviders.asp>。

命名空间组织结构

与各个 .NET 数据提供程序相关联的类型（类、结构、枚举等）位于其各自的命名空间中：

- **System.Data.SqlClient**。包含 SQL Server .NET 数据提供程序类型。
- **System.Data.OracleClient**。包含 Oracle .NET 数据提供程序。
- **System.Data.OleDb**。包含 OLE DB .NET 数据提供程序类型。
- **System.Data.Odbc**。包含 ODBC .NET 数据提供程序类型。
- **System.Data**。包含独立于提供程序的类型，如 **DataSet** 和 **DataTable**。

在各自的关联命名空间内，每个提供程序都提供了对 **Connection**、**Command**、**DataReader** 和 **DataAdapter** 对象的实现。**SqlConnection** 实现的前缀为“Sql”，而 **OleDb** 实现的前缀为“OleDb”。例如，**Connection** 对象的 **SqlConnection** 实现是 **SqlConnection**，而 **OleDb** 实现则为 **OleDbConnection**。同样，**DataAdapter** 对象的两个实现分别为 **SqlDataAdapter** 和 **OleDbDataAdapter**。

在本指南中，所用示例取自 SQL Server 对象模型。尽管此处未加说明，Oracle/OLEDB 和 ODBC 中提供了类似的功能。

一般编程

如果您可能要面向不同的数据源，并且需要将您的代码从一个数据源移至另一个数据源，请考虑编程以支持 **System.Data** 命名空间中的 **IDbConnection**、**IDbCommand**、**IDbDataReader** 和 **IDbDataAdapter** 接口。**Connection**、**Command**、**DataReader** 和 **DataAdapter** 对象的所有实现都必须支持这些接口。

有关实现 .NET 数据提供程序的详细信息，请参阅 <http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconimplementingnetdataprovider.asp>。

还应该注意，如果应用程序使用单对象模型来访问多个数据库，则 OLE DB 和 ODBC 桥接提供程序都可以使用。在此情况下，需要考虑与应用程序的性能需要相比，要求应用程序具有多大的灵活性，以及在何种程度上需要数据库特有的功能。

图 2 阐明了数据访问栈，并说明了 ADO.NET 与其他数据访问技术（包括 ADO 和 OLE DB）之间的关系。它还说明了 ADO.NET 模型内的两个托管提供程序和主要对象。

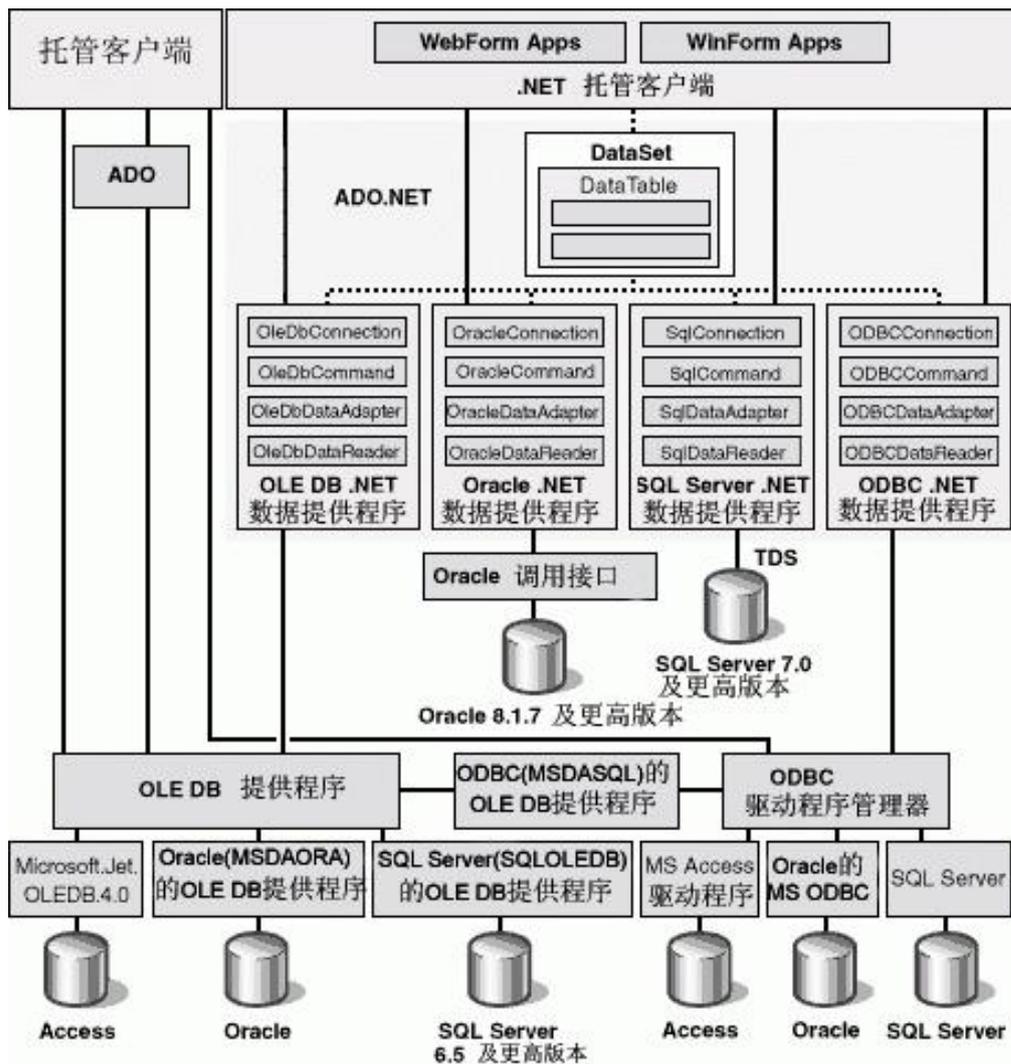


图 1.2.数据访问栈

有关 ADO 向 ADO.NET 的演变的详细信息，请参阅文章“Introducing ADO+:Data Access Services for the Microsoft .NET Framework”（MSDN Magazine, 2000 年 11 月号），网址为：
<http://msdn.microsoft.com/msdnmag/issues/1100/adoplus/default.aspx>。

存储过程与直接 SQL

本档中显示的大多数代码片段使用 `SqlCommand` 对象来调用存储过程，以执行数据库操作。在某些情况下，您将不会看到 `SqlCommand` 对象，因为存储过程名被直接传递给 `SqlDataAdapter` 对象。在内部，这仍然会导致创建 `SqlCommand` 对象。

您应该使用存储过程而不是嵌入的 SQL 语句，原因如下：

- 存储过程通常可以改善性能，因为数据库能够优化存储过程使用的数据库访问计划，并且能够缓存该计划以供将来重用。
- 可以在数据库内分别设置各个存储过程的安全保护。客户端不必对基础表拥有访问权限，就可以获得执行存储过程的权限。
- 存储过程可以简化维护工作，因为修改存储过程通常要比更改已部署组件中的硬编码 SQL 语句容易。
- 存储过程为基础数据库架构增加了额外的抽象级别。存储过程的客户端与存储过程的实现细节是彼此隔离的，与基础架构也是彼此隔离的。
- 存储过程可以减少网络流量，因为可以批量执行 SQL 语句，而不是从客户端发送多个请求。

SQL Server 联机文档强烈建议您不要使用“sp_”作为名称前缀来创建任何存储过程，因为此类名称已经被指定给系统存储过程。SQL Server 始终按以下顺序来查找以 sp_ 开头的存储过程：

1. 在主数据库中查找存储过程。
2. 基于所提供的任何限定符（数据库名或所有者）来查找存储过程。
3. 使用 dbo 作为所有者来查找存储过程（如果未指定所有者）。

属性与构造函数参数

可以通过构造函数参数来设置 ADO.NET 对象的特定属性值，也可以直接设置属性值。例如，下面的代码片段在功能上是等效的。

```
// 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";
```

从性能角度看，这两种方法之间的差异是微不足道的，因为针对 .NET 对象设置和获取属性要比针对 COM 对象执行类似操作更为高效。

选择哪种方法取决于个人喜好和编码风格。不过，对属性进行明确设置确实能够使代码更易理解（尤其是当您不熟悉 ADO.NET 对象模型时）和调试。

注 过去，Microsoft Visual Basic? 开发系统的开发人员被告诫避免使用“Dim x As New...”构造来创建对象。在 COM 领域里，上述代码会导致 COM 对象创建过程出现短路现象，从而带来一些微小和重大的错误。然而，在 .NET 领域里，这不再是一个问题。

[返回页首](#)

管理数据库连接

数据库连接代表一种关键的、昂贵的和有限的资源，尤其是在多层 Web 应用程序中。正确地管理连接是十分必要的，因为您采取的方法可能显著影响应用程序的总体可伸缩性。同时，还要认真考虑在何处存储连接字符串。需要使用可配置的且安全的位置。

在管理数据库连接和连接字符串时，应该努力做到：

- 通过在多个客户端中多路复用数据库连接池，帮助实现应用程序的可伸缩性。
- 采用可配置的、高性能的连接池策略。
- 在访问 SQL Server 时使用 Windows 身份验证。
- 在中间层避免模拟。
- 安全地存储连接字符串。
- 尽量晚地打开数据库连接，尽量早地将其关闭。

本节讨论连接池，并且帮助您选择适当的连接池策略。本节还将考虑应该如何管理、存储和操纵数据库连接字符串。最后，本节将给出两种编码模式，可用来帮助确保连接被可靠地关闭，并被返回到连接池。

使用连接池

通过数据库连接池，应用程序可以重用池中现有的连接，而不必反复与数据库建立新的连接。该技术可显著提高应用程序的可伸缩性，因为有限数量的数据库连接可以为数量大得多的客户端提供服务。同时，由于可以节省建立新连接所需的大量时间，该技术还能够改善性能。

像 ODBC 和 OLE DB 这样的数据访问技术提供了多种形式的连接池，可以在不同程度上进行配置。这两种方法对于数据库客户端应用程序而言在很大程度上是透明的。OLE DB 连接池经常被称为会话或资源池。

有关 Microsoft Data Access Components (MDAC) 内部的池机制的一般性讨论，请参阅“Pooling in the Microsoft Data Access Components”，网址为 <http://msdn.microsoft.com/library/en-us/dnmdac/html/pooling2.asp>。

ADO.NET 数据提供程序提供了透明的连接池，这些连接池的确切技术细节对于各个提供程序而言是不同的。本节针对下列提供程序来讨论连接池：

- SQL Server .NET 数据提供程序
- Oracle .NET 数据提供程序

- OLE DB .NET 数据提供程序
- ODBC .NET 数据提供程序

SQL Server .NET 数据提供程序的池机制

如果您使用的是 SQL Server .NET 数据提供程序，请使用该提供程序提供的连接池支持。这是一种由该提供程序在内部实现的支持事务处理并且非常高效的机制，它存在于托管代码中。池是以每个应用程序的域为基础创建的，并且在应用程序域卸载之前不会销毁。

可以透明地使用这种形式的连接池，但应该知道池的管理方式以及可用来微调连接池的各种配置选项。

在许多情况下，对于您的应用程序而言，SQL Server .NET 数据提供程序的默认连接池设置可能已经足够了。在开发和测试基于 .NET 的应用程序的过程中，建议您对规划通信模式进行模拟，以确定是否需要修改连接池大小。

需要生成可伸缩的高性能应用程序的开发人员应该最大限度地减少使用连接的时间，只在检索或更新数据时才使连接保持打开状态。连接关闭时，将被返回到连接池，并可供重用。在此情况下，到数据库的实际连接不会被切断；不过，如果连接池被禁用，则到数据库的实际连接将被关闭。

开发人员应该十分小心，不要依赖垃圾回收器来释放连接，因为当引用离开作用范围时，连接未必能够关闭。这是连接泄漏的一种常见根源，当请求新连接时，这会导致连接异常。

配置 SQL Server .NET 数据提供程序连接池

可以使用一组名称-值对（通过连接字符串提供）来配置连接池。例如，可以配置是否启用连接池（默认情况下启用）、池的最大容量和最小容量，以及要打开连接的排队请求可以阻塞的时间长度。下面是一个示例连接字符串，用于配置池的最大容量和最小容量。

```
"Server=(local); Integrated Security=SSPI; Database=Northwind;  
Max Pool Size=75; Min Pool Size=5"
```

打开连接并创建池以后，会将多个连接添加到池中，以便将连接数量提高到所配置的最小数量。随后，可以继续向该池中添加连接，直至达到所配置的最大池数量。当达到最大数量时，要打开连接的新请求将排队等待一段可配置的时间。

选择池大小

对于管理成千上万个客户端的并发请求的大规模系统而言，能够建立最大值阈值是非常重要的。您需要监控连接池和应用程序的性能，以便确定系统的最佳池大小。最佳大小还取决于用来运行 SQL Server 的硬件。

在部署过程中，您可能希望减小默认的最大池大小（目前为 100）以帮助查找连接泄漏。

如果您设立最小池大小，将会产生一点性能开销，因为在开始时对池进行填充以使其达到该尺寸，尽管头几个进行连接的客户端会因此受益。注意，创建新连接的过程是按顺序执行的，这意味着在最初填充池时，服务器不会被同时到来的大量请求所淹没。

有关监控连接池的详细信息，请参阅本文档中的监控连接池一节。有关连接池连接字符串关键字的完整列表，请参阅《.NET 框架开发人员指南》中的“Connection Pooling for the .NET Framework Data Provider for SQL Server”一节，网址为：

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconnectionpoolingforsqlservernetdataprovider.asp>。

更多信息

当使用 SQL Server .NET 数据提供程序连接池时，请注意以下几个方面：

- 连接是通过连接字符串上的完全匹配算法进行池化的。池机制甚至对名称-值对之间的空格也敏感。例如，下面的两个连接字符串将导致两个独立的池，因为第二个连接字符串包含额外的空格字符。
- ```
SqlConnection conn = new SqlConnection(
 "Integrated Security=SSPI;Database=Northwind");
conn.Open(); // Pool A is created
```
- ```
SqlConnection conn = new SqlConnection(
    "Integrated Security=SSPI ; Database=Northwind");
conn.Open(); // Pool B is created (extra spaces in string)
```
- 连接池被划分为多个事务专用池和一个与当前尚未在事务中登记的对池。对于与特定事务上下文关联的线程，会返回相应池（该池包含在该事务中登记的对池）的连接。这就使得使用已登记的对池成为一个透明的过程。

OLE DB .NET 数据提供程序的池机制

OLE DB .NET 数据提供程序通过使用基础 OLE DB 资源池来池化连接。有多个用于配置资源池的选择：

- 可以使用连接字符串来配置、启用或禁用资源池。
- 可以使用注册表。
- 可以用编程方式配置资源池。

为避免出现与注册表相关的部署问题，请不要使用注册表来配置 OLE DB 资源池。

有关 OLE DB 资源池的详细信息，请参阅《OLE DB 程序员参考》第 19 章“OLE DB Services”中的“Resource Pooling”，网址为：

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/olprcore_chapter19.asp。

使用池对象管理连接池

作为 Windows DNA 开发人员，鼓励您禁用 OLE DB 资源池和/或 ODBC 连接池，并使用 COM+ 对象池作为池化数据库连接的技术。这有两个主要原因：

- 池大小和阈值可以明确的配置（在 COM+ 目录中）。
- 性能得到改善。池对象方法的性能比本机池高 50%。

然而，因为 SQL Server .NET 数据提供程序在内部使用池机制，所以您不再需要开发自己的对象池机制（在使用该提供程序时）。因此，您可以避免执行与手动事务登记相关联的复杂任务。

如果您使用的是 OLE DB .NET 数据提供程序，您可能需要考虑使用 COM+ 对象池，以便充分利用卓越的配置和改善的性能。如果您为此目的开发池对象，则必须禁用 OLE DB 资源池和自动事务登记（例如，通过在连接字符串中包含“OLE DB Services=-4”）。您必须在自己的池对象实现中处理事务登记。

监控连接池

要对应用程序使用连接池的情况进行监控，可以使用 SQL Server 随附的事件探查器工具，或者使用 Microsoft Windows? 2000 操作系统随附的性能监视器工具。

使用 SQL Server 事件探查器监控连接池

1. 单击 **Start**，指向 **Programs**，指向 **Microsoft SQL Server**，然后单击 **Profiler** 以启动事件探查器。
2. 在 **File** 菜单上，指向 **New**，然后单击 **Trace**。
3. 提供连接详细信息，然后单击 **OK**。
4. 在 **Trace Properties** 对话框中，单击 **Events** 选项卡。
5. 在 **Selected event classes** 列表中，确保 **Audit Login** 和 **Audit Logout** 事件显示在 **Security Audit** 下面。要使跟踪变得更为清晰，请从该列表中删除所有其他事件。
6. 单击 **Run** 以启动跟踪。当连接建立时，您将看到 **Audit Login** 事件；当连接关闭时，您将看到 **Audit Logout** 事件。

使用性能监视器监控连接池

1. 单击 **Start**，指向 **Programs**，指向 **Administrative Tools**，然后单击 **Performance** 以启动性能监视器。
2. 右键单击图形背景，然后单击 **Add Counters**。
3. 在 **Performance object** 下拉列表中，单击 **SQL Server:General Statistics**。
4. 在显示的列表中，单击 **User Connections**。
5. 单击 **Add**，然后单击 **Close**。

管理安全性

尽管数据库连接池提高了应用程序的总体可伸缩性，但这意味着您不再能够在数据库级别管理安全性。这是因为，要支持连接池，连接字符串必须完全相同。如果您需要跟踪每个用户的数据库操作，请考虑添加一个参数，以便能够传递用户标识并在数据库中手动记录用户操作。您需要将该参数添加到每个操作中。

使用 Windows 身份验证

在连接到 SQL Server 时，应该使用 Windows 身份验证，因为它提供了许多好处：

1. 安全性更易于管理，因为您使用单一（Windows）安全模型，而不是独立的 SQL Server 安全模型。
2. 可避免将用户名和密码嵌入到连接字符串中。
3. 不会以明文方式通过网络传递用户名和密码。
4. 通过采用密码到期期限、最小长度以及在多次无效登录请求后锁定帐户，改善了登录安全性。

更多信息

在使用 Windows 身份验证来访问 SQL Server 时，请遵循以下指导原则：

- **考虑性能折衷。**性能测试已经表明，在使用 Windows 身份验证打开池数据库连接时，速度要比使用 SQL Server 身份验证慢。.NET 运行库 1.1 版已经降低了 SQL Server 安全性能超过 Windows 身份验证的幅度，但 SQL Server 身份验证仍然要快一些。

然而，尽管 Windows 身份验证的开销仍然比较高，但与执行命令或存储过程所花的时间相比，性能上的降低相对而言是可以忽略的。因此，在大多数情况下，使用 Windows 身份验证在安全方面的好处胜过了性能略有下降所带来的坏处。在做决定之前，请对应用程序的性能要求进行评估。

- **在中间层避免模拟。**Windows 身份验证要求有 Windows 帐户以便进行数据库访问。尽管在中间层使用模拟似乎是合理的，但请避免这样做，因为这样会使连接池失效，并且对应用程序可伸缩性产生严重的影响。

要解决该问题，请考虑模拟有限数量的 Windows 帐户（而不是已验证身份的用户），每个帐户都代表一个特定的角色。

例如，您可以使用以下方法：

1. 创建两个 Windows 帐户，一个用于读操作，一个用于写操作。（或者，您可能需要单独的帐户来镜像应用程序专有的角色。例如，您可能需要将一个帐户用于 Internet 用户，将另一个帐户用于内部操作员和/或管理员。）
2. 将各个帐户映射到 SQL Server 数据库角色，并且为每个角色设立必要的数据库权限。
3. 在数据访问层使用应用程序逻辑，确定在执行数据库操作之前要模拟的 Windows 帐户。

注 每个帐户都必须是域帐户，并且 Internet 信息服务（IIS）和 SQL Server 位于同一域中或位于可信域中。或者，您可以在每台计算机上创建匹配帐户（具有相同的帐户名和密码）。

- **对网络库使用 TCP/IP。**SQL Server 7.0 及更高版本为所有网络库提供 Windows 身份验证支持。使用 TCP/IP 可获得配置、性能和可伸缩性方面的好处。有关使用 TCP/IP 的详细信息，请参阅本文档中的通过防火墙连接一节。

有关开发安全的 ASP.NET 和 Web 应用程序的一般性指导，请参阅下面的 Microsoft *patterns & practices* 指南：

- 卷 I, 《生成安全的 ASP.NET 应用程序: 身份验证、授权和安全通讯》(网址为 <http://www.microsoft.com/practices>)
- 卷 II, 《提高 Web 应用程序安全性: 威胁和对策》(将发布于以下网址: <http://www.microsoft.com/practices>)

存储连接字符串

要存储数据库连接字符串,可以有多种选择,这些选择具有不同级别的灵活性和安全性。尽管在源代码中对连接字符串进行硬编码可提供最佳性能,但文件系统缓存可确保在外部将该字符串存储到文件系统中所带来的性能下降是微不足道的。几乎在所有情况下,人们都首选外部连接字符串所提供的额外的灵活性(它支持管理员配置)。

当您选择连接字符串存储方法时,需要注意的两个最重要的事项是安全性和配置简易性,然后紧跟着的是性能。

可以选择下列位置来存储数据库连接字符串:

- 在应用程序配置文件中;例如,ASP.NET Web 应用程序的 Web.config
- 在通用数据链接(UDL)文件中(仅由 OLE DB .NET 数据提供程序支持)
- 在 Windows 注册表中
- 在自定义文件中
- 在 COM+ 目录中,方法是使用构建字符串(仅适用于服务组件)

通过使用 Windows 身份验证来访问 SQL Server,可以避免将用户名和密码存储在连接字符串中。如果您的安全要求需要采取更严格的措施,请考虑以加密格式存储连接字符串。

对于 ASP.NET Web 应用程序而言,在 Web.config 文件内以加密格式存储连接字符串,代表着一种安全的、可配置的解决方案。

注 可以在连接字符串中将 **Persist Security Info** 命名值设置为 **false**,以禁止通过 **SqlConnection** 或 **OleDbConnection** 对象的 **ConnectionString** 属性返回对安全敏感的细节(如密码)。

下面几小节讨论了如何使用各种选择来存储连接字符串,并介绍了各种方法的相对优点和缺点。这些内容有助于您根据自己特定的应用程序方案做出明智的选择。

注 通过“配置应用程序管理”块,可以管理从数据库连接到复杂层次结构数据的各种配置设置。有关详细信息,请参阅 <http://msdn.microsoft.com/practices>。

使用 XML 应用程序配置文件

可以使用 `<appSettings>` 元素在应用程序配置文件的自定义设置节中存储数据库连接字符串。该元素支持任意的密钥-值对，如以下代码片段所示：

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

注 `<appSettings>` 元素出现在 `<configuration>` 元素下面，并且不是紧跟在 `<system.web>` 的后面。

优点

- **易于部署。**连接字符串是通过定期 `.NET xcopy` 部署与配置文件一起部署的。
- **易于以编程方式访问。**通过 `ConfigurationSettings` 类的 `AppSettings` 属性，可以在运行时方便地读取已配置的数据库连接字符串。
- **支持动态更新（仅限于 ASP.NET）。**如果管理员在 `Web.config` 文件中更新连接字符串，当下一次访问该字符串时（对于无状态组件而言，这可能是客户端下一次使用该组件进行数据访问请求），所做更改将生效。

缺点

- **安全性。**尽管 ASP.NET Internet 服务器应用程序编程接口 (ISAPI) 动态链接库 (DLL) 禁止客户端直接访问带有 `.config` 文件扩展名的文件，并且可以使用 NTFS 权限进一步限制访问，您可能仍然希望避免以明文形式在前端 Web 服务器上存储这些详细信息。为获得额外的安全性，请以加密格式在配置文件中存储连接字符串。

更多信息

- 可以使用 `System.Configuration.ConfigurationSettings` 类的静态 `AppSettings` 属性来检索自定义应用程序设置。以下代码片段对此进行了说明，该代码片段采用了前面例举的名为 `DBConnStr` 的自定义密钥：

```
using System.Configuration;
private string GetDBaseConnectionString()
```

- {
- return ConfigurationSettings.AppSettings["DBConnStr"];
- }
- 有关配置 .NET 框架应用程序的详细信息，请参阅
<http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconconfiguringnetframeworkapplications.asp>。

使用 UDL 文件

OLE DB .NET 数据提供程序在其连接字符串中支持通用数据链接 (UDL) 文件名。您可以通过使用 **OleDbConnection** 对象的构建参数来传递连接字符串，或者通过使用该对象的 **ConnectionString** 属性来设置连接字符串。

注 SQL Server .NET 数据提供程序在其连接字符串中不支持 UDL 文件。因此，只有在您使用 OLE DB .NET 数据提供程序时，才可使用该方法。

对于 OLE DB 提供程序，要通过连接字符串引用 UDL 文件，请使用“File Name=name.udl”。

优点

- **标准方法**。您可能已在使用 UDL 文件进行连接字符串管理。

缺点

- **性能**。每次打开连接时，都要读取和分析含有 UDL 的连接字符串。
- **安全性**。UDL 文件以纯文本形式存储。您可以通过使用 NTFS 文件权限来确保这些文件的安全，但这样做可能带来与 .config 文件相同的问题。
- **SqlClient 不支持 UDL 文件**。SQL Server .NET 数据提供程序不支持该方法。该提供程序用于访问 SQL Server 7.0 及更高版本。

更多信息

- 要支持管理，请确保管理员对 UDL 文件具有读/写访问权限，并且用于运行应用程序的身份具有读访问权限。对于 ASP.NET Web 应用程序，应用程序辅助进程在默认情况下通过使用 SYSTEM 帐户来运行，尽管您可以通过使用计算机全局配置文件 (Machine.config) 的 **<processModel>** 元素来覆盖这一配置。您还可以通过使用 Web.config 文件的 **<identity>** 元素进行模拟（可随意使用指定帐户）。
- 对于 Web 应用程序，请确保不要将 UDL 文件放在虚拟目录中，否则会使该文件可通过 Web 下载。

- 有关上述功能及其他与安全相关的 ASP.NET 功能的详细信息，请参阅“Authentication in ASP.NET: .NET Security Guidance”，网址为 <http://msdn.microsoft.com/library/en-us/dnbda/html/authaspdotnet.asp>。

使用 Windows 注册表

您还可以在 Windows 注册表中使用自定义键来存储连接字符串，尽管由于部署问题不鼓励这样做。

优点

- **安全性。**可以通过使用访问控制列表（ACL）来管理对选定注册表键的访问。要获得更高级别的安全性，请考虑将数据加密。
- **易于以编程方式访问。**可以使用 .NET 类来支持从注册表中读取字符串。

缺点

部署。必须与应用程序一起部署相关注册表设置，这在一定程度上抵消了 `xcopy` 部署的优点。

使用自定义文件

可使用自定义文件来存储连接字符串。然而，该技术没有任何优点，建议不要使用该技术。

优点

- 无。

缺点

- **额外的编码。**该方法需要完成额外的编码工作，并迫使您显式处理并发问题。
- **部署。**该文件必须与其他 ASP.NET 应用程序文件一起复制。应避免将该文件放在 ASP.NET 应用程序目录或子目录中，以防止它通过 Web 被下载。

使用构建参数和 COM+ 目录

可以在 COM+ 目录中存储数据库连接字符串,并通过对象构建字符串将其自动传递给您的对象。COM+ 将在实例化该对象后立即调用该对象的 **Construct** 方法,同时提供所配置的构建字符串。

注 该方法仅对服务组件有效。仅当您的托管组件使用其他服务(如分布式事务处理支持或对象池)时,才应考虑该方法。

优点

- **管理。**管理员可以通过使用“组件服务”MMC 管理单元方便地配置连接字符串。

缺点

- **安全性。**COM+ 目录被视为不安全的存储区域(尽管您可以通过 COM+ 角色来限制访问权限),因而不得用来以明文形式保存连接字符串。
- **部署。**COM+ 目录中的项必须与基于 .NET 的应用程序一起部署。如果您正在使用其他企业服务,如分布式事务处理或对象池,则在该目录中存储数据库连接字符串不会带来任何额外的部署开销,因为必须部署 COM+ 目录以支持那些其他服务。
- **组件必须接受服务。**您只能对服务组件使用构建字符串。不应简单地从 **ServicedComponent** 中派生组件的类(使您的组件接受服务)来启用构建字符串。

重要说明: 确保连接字符串的安全性至关重要。对于 SQL 身份验证,连接字符串包含用户名和密码。如果攻击者利用 Web 服务器上的源代码漏洞,并且获取了访问配置存储的权限,则数据库将容易受到攻击。要避免这一问题,应该将连接字符串加密。有关可用于加密明文连接字符串的不同方法的说明,请参阅“Improving Web Application Security:Threats and Countermeasures”(该文章将发布在 <http://www.microsoft.com/practices>)。

更多信息

- 有关如何配置 .NET 类以便构建对象的详细信息,请参阅附录中的如何启用 .NET 类的对象构造。
- 有关开发服务组件的详细信息,请参阅 <http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconwriting servicedcomponents.asp>。

- 有关开发安全的 ASP.NET 和 Web 应用程序的一般性指导, 请参阅下面的 Microsoft *patterns & practices* 指南:
 - 卷 I, 《生成安全的 ASP.NET 应用程序: 身份验证、授权和安全通讯》(网址为 <http://www.microsoft.com/practices>)
 - 卷 II, 《提高 Web 应用程序安全性: 威胁和对策》(将发布于以下网址: <http://www.microsoft.com/practices>)

连接使用模式

无论您使用哪种 .NET 数据提供程序, 您都必须始终遵循下列原则:

- 尽可能晚地打开数据库连接。
- 以尽可能短的时间使用连接。
- 尽可能早地关闭连接。在通过 **Close** 或 **Dispose** 方法关闭连接之前, 不会将连接返回到池中。即使您检测到连接已经进入断开状态, 也应该关闭该连接。这可以确保将该连接返回到池中, 并将其标记为无效连接。对象池程序定期扫描池, 查找那些已被标记为无效的对象。

要保证在方法返回之前关闭连接, 请考虑使用下面的两个代码示例中阐明的方法之一。第一个方法使用 **finally** 块。第二个方法使用 C# **using** 语句, 它确保调用对象的 **Dispose** 方法。

以下代码确保 **finally** 块关闭连接。注意, 该方法对 Visual Basic .NET 和 C# 都有效, 因为 Visual Basic .NET 支持结构化异常处理。

```
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();
    }
}
```

```
}
```

以下代码显示了一个备选方法，该方法使用了 C# **using** 语句。注意，Visual Basic .NET 不提供 **using** 语句或任何等效功能。

```
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);
        cmd.CommandType = CommandType.StoredProcedure;
        conn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

您还可以将该方法应用于其他必须关闭才能使用当前连接执行其他任何任务的对象，例如 **SqlDataReader** 或 **OleDbDataReader**。

[↑返回顶部](#)

错误处理

ADO.NET 错误是通过 .NET 框架所固有的基础结构化异常处理支持来产生和处理的。因此，在数据访问代码内部处理错误的方法与在应用程序的其他地方一样。可以通过标准的 .NET 异常处理语法和技术来检测和处理异常。

本节向您说明如何开发健壮的数据访问代码，并解释如何处理数据访问错误。同时，本节还提供了与 SQL Server .NET 数据提供程序相关的具体异常处理指导。

.NET 异常

.NET 数据提供程序将数据库特有的错误状况转换为标准异常类型，供您在自己的数据访问代码中进行处理。数据库特有的错误细节是通过相关异常对象的属性提供给您的。

所有 .NET 异常类型归根结底都是从 **System** 命名空间中的 **Exception** 类派生出来的。.NET 数据提供程序引发提供程序特有的异常类型。例如，每当 SQL Server 返回错误状况时，SQL Server .NET 数据提供程序都会引发

SqlException 对象。类似地，OLE DB.NET 数据提供程序引发 OleDbException 类型的异常，它含有基础 OLE DB 提供程序所公开的细节。

图 3 显示了 .NET 数据提供程序异常层次。注意，OleDbException 类是从 ExternalException（它是所有 COM Interop 异常的基类）派生的。该对象的 ErrorCode 属性存储了由 OLE DB 产生的 COM HRESULT。

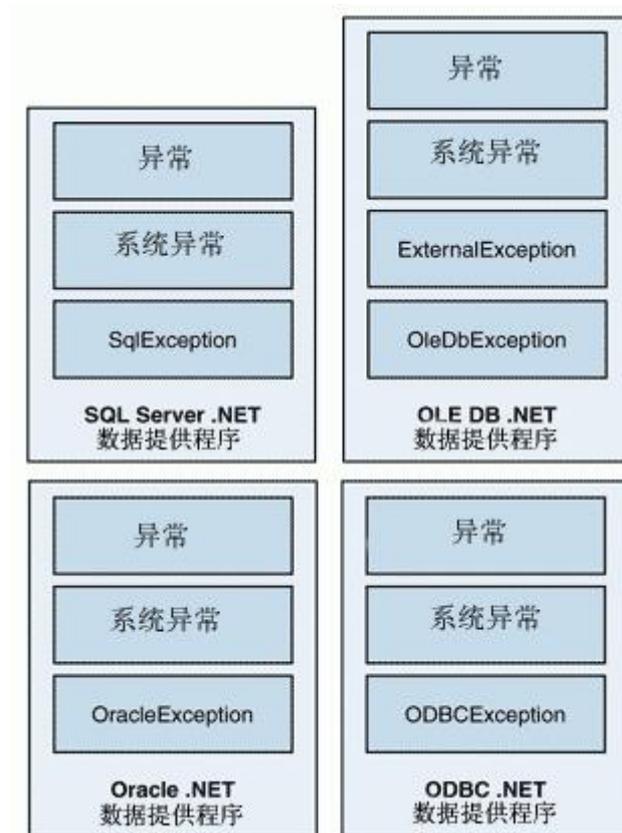


图 1.3. .NET 数据提供程序异常层次

捕获和处理 .NET 异常

要处理数据访问异常状况，请将数据访问代码放在 try 块内部，并使用 catch 块通过适当的筛选器来捕获产生的任何异常。例如，当使用 SQL Server .NET 数据提供程序来编写数据访问代码时，应该捕获 SqlException 类型的异常，如下代码所示：

```
try
{
    // Data access code
}
catch (SqlException sqllex) // more specific
{
```

```
}  
catch (Exception ex) // less specific  
{  
}  
}
```

如果您提供多个具有不同筛选条件的 **catch** 语句，请记住将这些语句按其类型的特殊程度从高到低排序。这样，对于任何给定的异常类型，都会执行最特殊类型的 **catch** 块。

该 **SQLException** 类公开了含有异常状况细节的属性。这些属性包括：

- **Message** 属性，它包含描述错误的文本。
- **Number** 属性，它包含唯一标识错误类型的错误号。
- **State** 属性，它包含有关错误的调用状态的附加信息。它通常用于指示特定错误状况的具体出现位置。例如，如果单个存储过程能够在多行产生同一错误，则应该使用状态来标识错误的具体出现位置。
- **Errors** 集合，它包含有关 SQL Server 产生的错误的详细错误信息。**Errors** 集合将始终至少包含一个 **SqlError** 类型的对象。

以下代码片段阐明了如何使用 SQL Server .NET 数据提供程序来处理 SQL Server 错误状况：

```
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 = null;  
    // Enclose all data access code within a try block  
    try  
    {  
        conn = new SqlConnection(  
            "server=(local);Integrated  
Security=SSPI;database=northwind");  
        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 sqllex)
    {
        // Handle data access exception condition
        // Log specific exception details
        LogException(sqllex);
        // Wrap the current exception in a more relevant
        // outer exception and re-throw the new exception
        throw new DALEXception(
            "Unknown ProductID: " + ProductID.ToString(), sqllex );
    }
    catch (Exception ex)
    {
        // Handle generic exception condition . . .
        throw ex;
    }
    finally
    {
        if(conn != null) conn.Close(); // Ensures connection is closed
    }
}

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

    foreach (SqlError sqle in sqllex.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` 块内部，代码首先使用 `LogException` 辅助函数记录异常细节。该函数使用 `foreach` 语句来枚举 `Errors` 集合内部的特定于提供程序的细节，并将错误细节记录到错误日志中。`catch` 块中的代码随后将 SQL Server 特有的异常包装在 `DALError` 类型的异常内部，后者对于 `GetProductName` 方法的调用方更有意义。异常处理程序使用 `throw` 关键字将该异常传播回调用方。

更多信息

- 有关 `SqlException` 类的成员的完整列表，请参阅 <http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataSqlClientSqlExceptionMembersTopic.asp>。
- 有关开发自定义异常、记录和包装 .NET 异常以及使用各种方法来传播异常的详细指导，请参阅 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp>。

从存储过程中产生错误

Transact-SQL (T-SQL) 提供了一个 `RAISERROR` 函数（请注意拼写），用来产生自定义错误并将其返回到客户端。对于 ADO.NET 客户端，SQL Server .NET 数据提供程序会截获这些数据库错误，并将其转换为 `SqlError` 对象。

使用 `RAISERROR` 函数的最简单方法是将消息正文包含为第一个参数，然后指定严重度和状态参数，如以下代码片段所示。

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

在该示例中，使用了一个替代参数将当前产品 ID 作为错误消息正文的一部分返回。第二个参数是消息严重度，第三个参数是消息状态。

更多信息

- 要避免对消息正文进行硬编码，可以使用 `sp_addmessage` 系统存储过程或使用 SQL Server 企业管理器，将您自己的消息添加到 `sysmessages`

表中。然后，您可以使用传递给 RAISERROR 函数的 ID 引用该消息。您定义的消息 ID 必须大于 50,000，如以下代码片段所示。

- RAISERROR(50001, 16, 1, @ProductID)
- 有关 RAISERROR 函数的详细信息，请在 SQL Server 联机图书索引中查找 RAISERROR。

适当地使用严重度级别

请认真选择错误严重度级别，并注意各个级别的影响。错误严重度级别的范围为 0 到 25，用于表示 SQL Server 2000 已经遇到的问题类型。在客户端代码中，可以通过检查 `SqlError` 对象的 `Class` 属性获得错误的严重度，该对象位于 `SqlException` 类的 `Errors` 集合中。表 1 指出了各种严重度级别的影响和含义。

表 1. 错误严重度级别 — 影响和含义

严重度级别	是否关闭连接	产生 <code>SqlException</code>	含义
10 及更低	否	否	指示性消息，不一定代表错误状况。
11 - 16	否	是	可由用户纠正的错误，例如，通过修正的输入数据重试操作。
17 - 19	否	是	资源或系统错误。
20 - 25	是	是	致命系统错误（包括硬件错误）。客户端的连接被终止。

控制自动事务处理

对于所遇到的任何严重度超过 10 的错误，SQL Server .NET 数据提供程序都会引发 `SqlException`。当自动 (COM+) 事务处理中的某个组件检测到 `SqlException` 时，该组件必须确保它赞成中止该事务处理。这可能是也可能不是自动过程，具体取决于是否用 `AutoComplete` 属性对该方法进行了标记。

有关在自动事务处理的上下文中处理 `SqlException` 的详细信息，请参阅本文档中的确定事务处理结果一节。

检索指示性消息

严重度级别 10 及更低级别用于表示指示性消息，并且不会导致引发 `SqlException`。

检索指示性消息:

- 创建一个事件处理程序，并预订由 `SqlConnection` 对象公开的 `InfoMessage` 事件。该事件的委托如以下代码片段中所示。
- `public delegate void SqlInfoMessageEventHandler(object sender,`
- `SqlInfoMessageEventArgs e);`

消息数据可通过传递给事件处理程序的 `SqlInfoMessageEventArgs` 对象获得。该对象公开了一个 `Errors` 属性，该属性包含一组 `SqlError` 对象 — 每条指示性消息对应一个对象。以下代码片段阐明了如何注册一个用于记录指示性消息的事件处理程序。

```
public string GetProductName( int ProductID )
{
    SqlConnection conn = null;
    try
    {
        conn = new SqlConnection(
            "server=(local);Integrated
Security=SSPI;database=northwind");
        // Register a message event handler
        conn.InfoMessage += new
SqlInfoMessageEventHandler( MessageEventHandler );
        conn.Open();
        // Setup command object and execute it
        . . .
    }
    catch (SqlException sqllex)
    {
        // log and handle exception
        . . .
    }
    finally
    {
        if(conn != null) conn.Close();
    }
}
// message event handler
void MessageEventHandler( object sender, SqlInfoMessageEventArgs e )
{
    foreach( SqlError sqle in e.Errors )
    {
        // Log SqlError properties
        . . .
    }
}
```

```
}  
}
```

[返回页首](#)

性能

本节介绍一些常见的数据访问方案，对于每种方案，都提供了就 ADO.NET 数据访问代码而言性能最高、可伸缩性最强的解决方案的有关详细信息。在适当的地方，对性能、功能和开发工作进行了比较。本节考虑了下列功能方案：

- 检索结果集并对检索到的行进行迭代处理。
- 检索具有指定主键的单个行。
- 从指定行检索单个项。
- 检查是否存在具有特定主键的行。这是单项查找方案的变种，此时返回一个简单的 Boolean 类型值就足够了。

检索多行

在该方案中，您希望检索一组表格形式的数据，并对检索到的行进行迭代以执行操作。例如，您可能希望检索一组数据，以不连续的方式处理这些数据，并将其作为 XML 文档传递给客户端应用程序（可能是通过 Web 服务传递）。或者，您可能还希望以 HTML 表的形式显示这些数据。

要帮助确定最适当的数据访问方法，请考虑您是需要（不连续的）**DataSet** 对象所提供的额外的灵活性，还是需要 **SqlDataReader** 对象（它非常适合于企业级用户（B2C）Web 应用程序中的数据展示）所提供的原始性能。图 4 显示了这两种基本方案。

注 用于在内部填充 **DataSet** 的 **SqlDataAdapter** 使用 **SqlDataReader** 来访问数据。

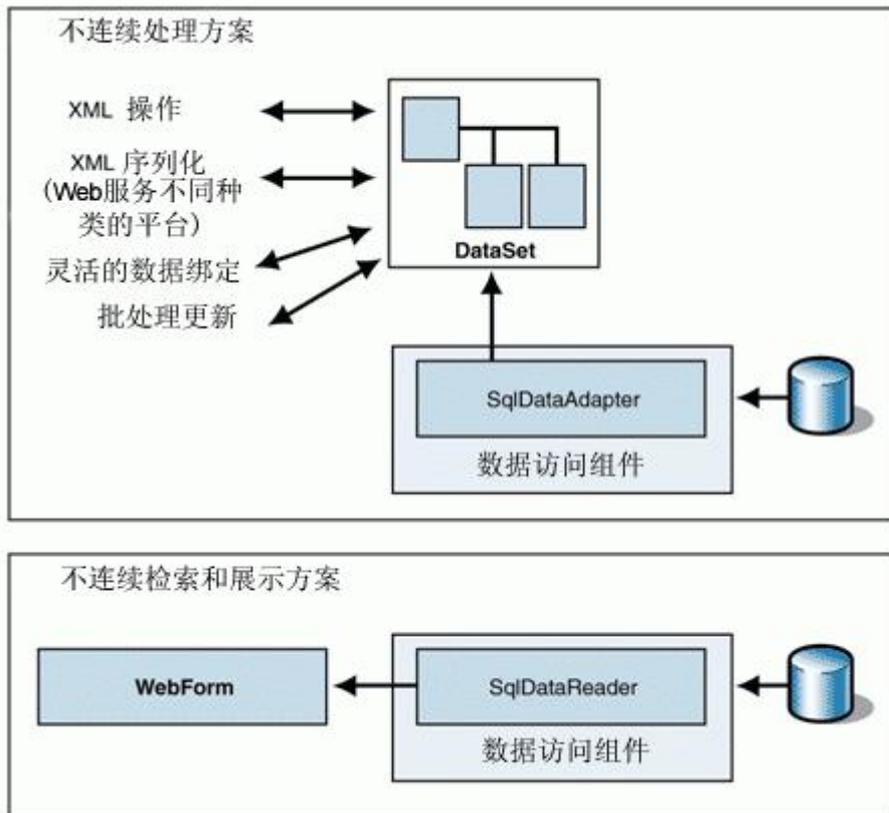


图 1.4.多行数据访问方案

比较几种选择

当从数据源中检索多个行时，您具有下列选择：

- 使用 **SqlDataAdapter** 对象生成 **DataSet** 或 **DataTable**。
- 使用 **SqlDataReader** 提供只读的、只进的数据流。
- 使用 **XmlReader** 提供 XML 数据的只读、只进数据流。

选择 **SqlDataReader** 还是 **DataSet/DataTable**，从根本上说是一个注重性能还是注重功能的问题。**SqlDataReader** 可提供最佳性能；**DataSet** 可提供额外的功能和灵活性。

数据绑定

上述所有三个对象都可以充当数据绑定控件的数据源，尽管 **DataSet** 和 **DataTable** 可以充当比 **SqlDataReader** 更多种控件的数据源。这是因为 **DataSet** 和 **DataTable** 实现了 **IListSource** (产生 **IList**)，而 **SqlDataReader** 则实现了 **IEnumerable**。一些支持数据绑定的 **WinForm** 控件需要实现了 **IList** 的数据源。

这一差异的原因在于为其设计各种对象类型的方案的类型。**DataSet**（它包含**DataTable**）是一种丰富的、不连续的结构，同时适用于 Web 和桌面（WinForm）这两种方案。另一方面，数据读取器对需要执行优化的只进数据访问的 Web 应用程序进行了优化。

请检查您希望绑定到的特定控件类型的数据源要求。

在应用程序层之间传递数据

DataSet 为可以根据需要作为 XML 进行操作的数据提供了关系视图，并且使您可以在应用程序层和组件之间传递不连续的缓存数据副本。然而，**SqlDataReader** 提供了最佳性能，因为它避免了与 **DataSet** 的创建关联的性能和内存开销。请记住，**DataSet** 对象的创建可导致多个子对象（包括 **DataTable**、**DataRow** 和 **DataColumn** 对象）以及用作这些子对象的容器的集合对象的创建。

使用数据集

在下列情况下，请使用由 **SqlDataAdapter** 对象填充的 **DataSet**：

- 您需要不连续的内存驻留型数据缓存，以便可以将其传递给应用程序内的其他组件或层。
- 您需要数据在内存中的关系视图，以便进行 XML 或非 XML 操作。
- 您要处理从多个数据源（如多个数据库、表或文件）中检索到的数据。
- 您希望更新检索到的全部或部分行，并且使用 **SqlDataAdapter** 的批量更新功能。
- 您希望对其执行数据绑定的控件需要支持 **IList** 的数据源。

注 有关详细信息，请参阅 MSDN 网站上的“Designing Data Tier Components and Passing Data Through Tiers”，网址为 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp>。

更多信息

如果您使用 **SqlDataAdapter** 来生成 **DataSet** 或 **DataTable**，请注意下列事项：

- 不需要显式打开或关闭数据库连接。**SqlDataAdapter Fill** 方法可打开数据库连接，然后在其返回之前关闭该连接。如果连接已经打开，**Fill** 将使该连接保持打开。
- 如果您需要将该连接用于其他目的，请考虑在调用 **Fill** 方法之前打开它。这样，您可以避免不必要的打开/关闭操作，从而提高性能。

- 尽管您可以反复使用同一 `SqlCommand` 对象来多次执行同一命令, 请不要重用同一 `SqlCommand` 对象来执行不同的命令。
- 有关说明如何使用 `SqlDataAdapter` 来填充 `DataSet` 或 `DataTable` 的代码示例, 请参阅附录中的如何使用 `SqlDataAdapter` 来检索多个行。

使用 `SqlDataReader`

在下列情况下, 请使用通过调用 `SqlCommand` 对象的 `ExecuteReader` 方法得到的 `SqlDataReader`:

- 您要处理大量的数据 — 多得难以在单个缓存中进行维护。
- 您希望减少应用程序的内存使用量。
- 您希望避免与 `DataSet` 相关联的对象创建开销。
- 您希望用对实现了 `IEnumerable` 的数据源提供支持的控件执行数据绑定。
- 您希望简化和优化数据访问。
- 您要读取的行包含二进制大对象 (BLOB) 列。您可以使用 `SqlDataReader` 在可管理的块区中将 BLOB 数据从数据库中提取出来, 而不是一次将其全部提取出来。有关处理 BLOB 数据的详细信息, 请参阅本文档中的处理 BLOB 一节。

更多信息

如果您使用 `SqlDataReader`, 请注意以下事项:

- 当数据读取器处于活动状态时, 到数据库的基础连接将保持打开状态, 并且无法用于任何其他目的。尽可能早地调用 `SqlDataReader` 上的 `Close`。
- 每个连接只能有一个数据读取器。
- 您可以在使用完数据读取器后显式关闭连接, 或者通过将 `CommandBehavior.CloseConnection` 枚举值传递给 `ExecuteReader` 方法, 将连接的生存期与 `SqlDataReader` 对象联系起来。这表示在关闭 `SqlDataReader` 后, 应该关闭连接。
- 在使用读取器访问数据时, 如果知道列的基础数据类型, 应使用类型化的访问器方法 (如 `GetInt32` 和 `GetString`), 这是因为它们可减少读取列数据时需要执行的类型转换的数量。
- 要避免将不需要的数据从服务器提取到客户端, 如果您希望关闭读取器并丢弃其余任何结果, 应在调用读取器上的 `Close` 之前, 调用命令对象的 `Cancel` 方法。`Cancel` 确保在服务器上丢弃结果, 而不会将结果无谓地提取到客户端上。否则, 调用数据读取器上的 `Close` 会导致读取器无谓地提取其余结果以清空数据流。
- 如果您希望获得输出或者从存储过程返回的返回值, 并且您使用的是 `SqlCommand` 对象的 `ExecuteReader` 方法, 则必须在输出和返回值可用之前调用读取器上的 `Close` 方法。

- 有关说明如何使用 `SqlDataReader` 的代码示例，请参阅附录中的如何使用 `SqlDataReader` 来检索多个行。

使用 `XmlReader`

在以下情况下，请使用通过调用 `SqlCommand` 对象的 `ExecuteXmlReader` 方法得到的 `XmlReader`：

- 您希望将检索到的数据作为 XML 处理，但您不希望创建 `DataSet` 带来性能开销，并且不需要不连续的数据缓存。
- 您希望利用 SQL Server 2000 **FOR XML** 子句的功能，该子句可用来以灵活的方式从数据库中检索 XML 片段（即不带根元素的 XML 文档）。例如，使用该方法可以指定准确的元素名称，还可以指定是否应该使用以元素或属性为中心的架构，是否应该通过 XML 数据返回架构，等等。

更多信息

如果您使用 `XmlReader`，请注意以下事项：

- 在您从 `XmlReader` 中读取数据时，连接必须保持打开。`SqlCommand` 对象的 `ExecuteXmlReader` 方法目前不支持 `CommandBehavior.CloseConnection` 枚举值，因此您在使用完读取器后必须显式关闭连接。
- 有关说明如何使用 `XmlReader` 的代码示例，请参阅附录中的如何使用 `XmlReader` 来检索多个行。

检索单个行

在该方案中，您希望从数据源中检索包含指定列集的单行数据。例如，您有一个客户 ID 并希望查找相关的客户详细信息，或者您有一个产品 ID 并希望检索产品信息。

比较几种选择

如果您希望使用从数据源中检索到的单行执行数据绑定，可以使用 `SqlDataAdapter`，以与前面讨论的“多行检索和迭代”方案中介绍的方式来填充 `DataSet` 或 `DataTable`。然而，除非您特别需要 `DataSet/DataTable` 功能，否则应该避免创建这些对象。

如果您需要检索单个行，请使用下列选择之一：

- 使用存储过程输出参数。
- 使用 `SqlDataReader` 对象。

这两个选择都可以避免由于在服务器上创建结果集和在客户端上创建 `DataSet` 而带来的不必要的开销。各个方法的相对性能取决于压力级别以及是否启用了数据库连接池。当启用了数据库连接池时，性能测试表明，存储过程方法在高压力条件（同时存在 200 个以上的连接）下的性能比 `SqlDataReader` 方法高将近 30%。

使用存储过程输出参数

当您希望从已经启用连接池的多层 Web 应用程序中检索单个行时，请使用存储过程输出参数。

更多信息

有关说明如何使用存储过程输出参数的代码示例，请参阅附录中的如何使用存储过程输出参数来检索单个行。

使用 `SqlDataReader`

在以下情况下，请使用 `SqlDataReader`：

- 除了数据值以外，您还需要元数据。您可以使用数据读取器的 `GetSchemaTable` 方法获得列元数据。
- 您未使用连接池。如果禁用了连接池，`SqlDataReader` 在所有压力条件下都是一个不错的选择；性能测试已经表明，在存在 200 个浏览器连接的情况下，它的性能比存储过程方法高 20% 左右。

更多信息

如果您使用 `SqlDataReader`，请注意以下事项：

- 如果您知道查询仅返回单个行，请在调用 `SqlCommand` 对象的 `ExecuteReader` 方法时使用 `CommandBehavior.SingleRow` 枚举值。一些提供程序（如 OLE DB .NET 数据提供程序）使用这一提示来优化性能。例如，该提供程序通过使用 `IRow` 接口（如果该接口可用）而不是开销更大的 `IRowset` 来执行绑定。该方法对 SQL Server .NET 数据提供程序没有影响。
- 如果您的 SQL Server 命令包含输出参数或返回值，在关闭 `DataReader` 之前它们将不可用。

- 在使用 `SqlDataReader` 对象时,请始终通过 `SqlDataReader` 对象的类型化访问器方法(例如 `GetString` 和 `GetDecimal`)来检索输出参数。这可避免不必要的类型转换。
- .NET 框架 1.1 版包含一个名为 `HasRows` 的附加 `DataReader` 属性,使用该属性,可以在从 `DataReader` 中读取之前确定它是否已经返回结果。
- 有关说明如何使用 `SqlDataReader` 对象检索单个行的代码示例,请参阅附录中的如何使用 `SqlDataReader` 来检索单个行。

检索单个项

在该方案中,您希望检索单个数据项。例如,您可能希望查找单个产品名(给定了产品 ID)或单个客户信用等级(给定了客户名称)。在此类方案中,您通常不希望在检索单个项时产生 `DataSet` 的开销,甚至不希望产生 `DataTable` 的开销。

您可能还希望简单地检查一下数据库中是否存在特定行。例如,当新用户在网上注册时,您需要检查选择的用户名是否已存在。这是单项查找的特殊情况,但在此情况下,简单的 `Boolean` 类型返回值就足够了。

比较几种选择

当您从数据源中检索单个数据项时,请考虑下列选择:

- 通过存储过程使用 `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,因为它专用于只返回单个值的查询。它所需要的代码比存储过程输出参数和 `SqlDataReader` 方法需要的代码都要少。

从性能角度而言,应该使用存储过程输出或返回参数,因为测试表明,存储过程方法在低压力条件和高压力条件下(从同时存在的浏览器连接少于 100 个到同时存在 200 个浏览器连接)都能够提供一致的性能。

更多信息

在检索单个项时,请注意以下事项:

- 如果查询通常返回多个列和/或行,则通过 `ExecuteQuery` 执行该查询将仅返回第一行的第一列。
- 有关说明如何使用 `ExecuteScalar` 的代码示例,请参阅附录中的如何使用 `ExecuteScalar` 来检索单个项。

- 有关说明如何使用存储过程输出或返回参数来检索单个项的代码示例，请参阅附录中的如何使用存储过程输出或返回参数来检索单个项。
- 有关说明如何使用 `SqlDataReader` 对象检索单个项的代码示例，请参阅附录中的如何使用 `SqlDataReader` 来检索单个项。

[↑返回顶部](#)

通过防火墙进行连接

您将经常需要配置 Internet 应用程序以通过防火墙连接到 SQL Server。例如，许多 Web 应用程序及其防火墙的关键体系结构组件是外围网络（也称为 DMZ，即非管制区），它用于将前端 Web 服务器与内部网络隔离。

通过防火墙连接到 SQL Server 时，要求对防火墙、客户端和服务端进行特殊的配置。SQL Server 提供了客户端网络实用工具和服务器网络实用工具程序来帮助进行配置。

选择网络库

在通过防火墙进行连接时，可使用 SQL Server TCP/IP 网络库来简化配置。这是 SQL Server 2000 安装的默认配置。如果您使用的是较低版本的 SQL Server，请确保已经分别通过使用客户端网络实用工具和服务器网络实用工具，在客户端和服务端上将 TCP/IP 配置为默认网络库。

除了配置方面的好处以外，使用 TCP/IP 库结果还意味着您可以：

- 由于在处理大量数据时的性能改善以及可伸缩性得到提高而受益。
- 避免其他与命名管道关联的安全问题。

您必须将客户端计算机和服务端计算机配置为使用 TCP/IP。因为大多数防火墙都对允许通信的端口集进行限制，所以您还必须对 SQL Server 使用的端口号给予认真的考虑。

配置服务器

SQL Server 的默认实例在端口 1433 上侦听。同时，还使用了 UDP 端口 1434，以使 SQL 客户端能够定位其网络中的其他 SQL 服务器。然而，SQL Server 2000 的命名实例在首次启动时动态分配端口号。网络管理员不希望在防火墙上打开一系列端口号；因此，在使用带防火墙的 SQL Server 命名实例时，请使用服务器网络实用工具来配置该实例，以使其在特定端口号上侦听。然后，您的管理员可以配置防火墙，以允许通信到达该服务器实例正在侦听的特定 IP 地址和端口号。

注 客户端网络库使用的源端口是在范围 1024 - 5000 中动态分配的。对于 TCP/IP 客户端应用程序，这是标准做法，但这意味着您的防火墙必须允许来自该范围内任意端口的通信。有关 SQL Server 使用的端口的详细信息，请参阅 Microsoft 知识库文章 287932 “[INF:TCP Ports Needed for Communication to SQL Server Through a Firewall](#)”。

动态发现命名实例

如果您更改 SQL Server 所侦听的默认端口号，请配置客户端以连接到此端口。有关详细信息，请参阅本文档中的配置客户端一节。

在为 SQL Server 2000 的默认实例更改端口号后，如果未能修改客户端，将导致连接错误。如果有多个 SQL Server 实例，最新版本的 MDAC 数据访问栈 (2.6) 将使用动态发现，并且使用用户数据文报协议 (UDP) 协商 (通过 UDP 端口 1434) 来查找命名实例。尽管这可能在开发环境中有效，但却不太可能在实际应用环境中起作用，因为防火墙通常会阻塞 UDP 协商通信。

要避免这一问题，请始终配置客户端以连接到已配置的目标端口号。

配置客户端

应该配置客户端以使用 TCP/IP 网络库连接到 SQL Server，并且应确保客户端库使用正确的目标端口号。

使用 TCP/IP 网络库

您可以使用 SQL Server 客户端网络实用工具来配置客户端。在某些安装中，客户端 (例如，您的 Web 服务器) 上可能尚未安装该实用工具。在此情况下，您可以执行下列操作之一：

- 通过使用由连接字符串提供的名称-值对 “Network Library=dbmssocn” 指定网络库。字符串 “dbmssocn” 用于标识 TCP/IP (套接字) 库。

注 在使用 SQL Server .NET 数据提供程序时，网络库设置在默认情况下使用 “dbmssocn”。

- 修改客户端计算机上的注册表，以便将 TCP/IP 设置为默认库。有关配置 SQL Server 网络库的详细信息，请参阅 [HOWTO:Change SQL Server Default Network Library Without Using Using Client Network Utility \(Q250550\)](#)。

指定端口

如果您的 SQL Server 实例被配置为在除默认的 1433 以外的端口上侦听，可以通过以下方法指定要连接到端口号：

- 使用客户端网络实用工具。
- 指定端口号，并将“Server”或“Data Source”名称-值对提供给连接字符串。使用具有以下格式的字符串：
- “Data Source=ServerName, PortNumber”

注 *ServerName* 可能是 IP 地址或域名系统 (DNS) 名称。要获得最佳性能，请使用 IP 地址以避免执行 DNS 查找。

分布式事务处理

如果您已经开发了使用 COM+ 分布式事务处理以及 Microsoft 分布式事务处理协调器 (DTC) 的服务的服务组件，您可能还需要配置您的防火墙以允许 DTC 通信在独立的 DTC 实例之间流动，以及在 DTC 和资源管理器（如 SQL Server）之间流动。

有关为 DTC 打开端口的详细信息，请参阅 [INFO:Configuring Microsoft Distributed Transaction Coordinator \(DTC\) to Work Through a Firewall](#)。

[↑返回页首](#)

处理 BLOB

现在，许多应用程序除了处理较为传统的字符和数值数据以外，还需要处理诸如图形和声音之类的数据格式，甚至需要处理更为复杂的数据格式，如视频。存在许多不同类型的图形、声音和视频格式。不过，从存储角度而言，它们都可以视为二进制数据块，通常称为二进制大对象，即 BLOB。

SQL Server 提供了 **binary**、**varbinary** 和 **image** 数据类型来存储 BLOB。尽管具有现在的名称，BLOB 数据还可以指基于文本的数据。例如，您可能希望存储可以与特定行关联的任意长的注释字段。为此，SQL Server 提供了 **ntext** 和 **text** 数据类型。

通常，对于小于 8 KB 的二进制数据，请使用 **varbinary** 数据类型。对于超过此大小的二进制数据，请使用 **image**。表 2 介绍了每个数据类型的主要功能。

表 2. 数据类型功能

数据类型	大小	说明
binary	从 1 到 8,000 字节。存储大小为指定长度加 4 字节。	固定长度二进制数据
varbinary	从 1 到 8,000 字节。存储大小为所提供数据的实际长度加 4 字节。	变长二进制数据
image	大小介于 0 到 2 GB 之间的变长二进制数据。	大型、变长二进制数据
text	大小介于 0 到 2 GB 之间的变长数据。	字符数据
ntext	大小介于 0 到 2 GB 之间的变长数据。	Unicode 字符数据

注 Microsoft? SQL Server 2000 能够在数据行中存储小型到中等大小的 **text**、**ntext** 和 **image** 值。该功能对于具有以下特征的表最为适用：表的 **text**、**ntext** 和 **image** 列中的数据通常作为一个单元读/写，并且大多数引用该表的语句都使用 **text**、**ntext** 和 **image** 数据。有关详细信息，请参阅 SQL Server 联机图书中的“text in row”主题。

将 BLOB 数据存储在哪里

SQL Server 7.0 及更高版本已经改善了使用数据库中存储的 BLOB 数据时的性能。原因之一是数据库页大小已经增加到 8 KB。因此，小于 8 KB 的文本或图像数据不再需要存储在单独的树页结构中，而可以存储在一行中。这意味着读/写 **text**、**ntext** 或 **image** 数据可以像读/写字符和二进制字符串一样快。当数据大于 8 KB 时，将在行内维护一个指针，而将数据本身保存在单独数据页的树结构中 — 从而不可避免地影响性能。

有关将 **text**、**ntext** 和 **image** 数据强行存储在一行中的详细信息，请参阅 SQL Server 联机图书中的“Using Text and Image Data”主题。

处理 BLOB 数据的常用备选方法是将 BLOB 数据存储在文件系统中，并且在数据库列中存储一个指针（最好是统一资源定位符 [URL] 链接）以引用相应的文件。对于低于 SQL?Server 7.0 的版本而言，在数据库外部将 BLOB 数据存储到文件系统中可以改善性能。

然而，由于在 SQL Server 2000 中改善了 BLOB 支持，再加上 ADO.NET 对读/写 BLOB 数据的支持，因此在数据库中存储 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

在数据库中存储 BLOB 数据可提供许多优点：

- 更易于使 BLOB 数据与行中的其余项保持同步。
- BLOB 数据通过数据库进行备份。拥有单个存储系统可以简化管理。
- 可以通过 SQL Server 2000 中的 XML 支持访问 BLOB 数据，从而在 XML 流中返回 Base64 编码形式的数据。
- 对于包含固定长度或可变长度字符（包含 Unicode）数据的列，可以执行 SQL Server 全文检索（FTS）操作。还可以针对 **image** 字段中包含的基于带格式文本的数据（例如，Microsoft Word 或 Microsoft Excel 文档）执行 FTS 操作。

在数据库中存储 BLOB 数据的缺点

请认真考虑哪些资源存储在文件系统中可能比存储在数据库中更好。通常通过 HTTP HREF 引用的图像就是很好的例子。这是因为：

- 从数据库中检索图像会导致比使用文件系统更大的开销。
- 数据库 SAN 上的磁盘存储通常比 Web 服务器场中使用的磁盘上的存储更为昂贵。

注 通过精心设计元数据策略，可以消除在数据库中存储图像、电影甚至 Microsoft Office 文档之类资源的需要。元数据可以编入索引，并可以包含指向文件系统中存储的资源的指针。

将 BLOB 数据写入数据库

以下代码说明了如何使用 ADO.NET 将从文件中得到的二进制数据写入 SQL Server 中的 image 字段。

```
public void StorePicture( string filename )
{
    // Read the file into a byte array
    using(FileStream fs = new FileStream(filename, FileMode.Open,
    FileAccess.Read))
    {
        byte[] imageData = new Byte[fs.Length];
        fs.Read( imageData, 0, (int)fs.Length );
    }

    using( SqlConnection conn = new SqlConnection(connectionString) )
    {
        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;
conn.Open();
cmd.ExecuteNonQuery();
}
}
```

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，请使用 `CommandBehavior.SequentialAccess` 枚举值。如果不使用该枚举值，读取器会每次一行地将数据从服务器提取到客户端。如果行中包含 BLOB 列，可能需要占用大量内存。通过使用枚举值，可以进行更精确的控制，因为仅在 BLOB 数据被引用（例如，通过 `GetBytes` 方法，该方法可用于控制所读取的字节数）时才会进行提取。以下代码片段对此进行了说明。

```
// Assume previously established command and connection
// The command SELECTs the IMAGE column from the table
conn.Open();
using(SqlDataReader reader =
cmd.ExecuteReader(CommandBehavior.SequentialAccess))
{
    reader.Read();
    // Get size of image data
    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 使用 `CommandBehavior.SequentialAccess` 时，要求按照严格的顺序次序访问列数据。例如，如果 BLOB 数据位于第 3 列，并且您还需要第 1 列和第 2 列中的数据，则必须在读取第 3 列之前读取第 1 列和第 2 列。

[返回页首](#)

通过数据集执行数据库更新

在引入 ADO.NET 以后，执行数据库更新的体系结构已经显著改变。ADO.NET 的目标是使得开发能够适应大型数据库和大量客户端的多层应用程序变得更加容易。这已经产生了一些重要的结果，尤其是：

- ADO.NET 应用程序通常将客户端上的应用程序逻辑与中间层和数据库层上的业务和数据完整性计算分开。实际上，这意味着典型的应用程序将具有较多的批处理或事务处理性质，而在客户端应用程序和数据库之间具有较少的（但较大的）交互。
- ADO.NET 应用程序对于究竟如何处理更新具有更多的控制（与 ADO 及其前身比较）。
- ADO.NET 允许应用程序通过存储在后端数据库中的存储过程来传播更改，而不是直接操纵数据库表中的行。这是推荐的实施策略。

更新使用模式

使用 ADO.NET 从 `DataSet` 中更新数据的过程可以按如下方式加以概述：

1. 创建一个 `DataAdapter` 对象，并且使用数据库查询的结果填充 `DataSet` 对象。数据将在本地缓存。
2. 对本地 `DataSet` 对象进行更改。这些更改可以包括对本地缓存的 `DataSet` 中的一个或多个表执行更新、删除和插入操作。
3. 初始化 `DataAdapter` 的与更新相关的属性。此步骤配置处理更新、删除或插入的确切方式。因为有多种处理该方法的方法，下面的“初始化 `DataAdapter` 以进行更新”部分对推荐方法进行了讨论。
4. 调用 `DataAdapter Update` 方法以提交挂起的更改。本地缓存的 `DataSet` 中的每个已更改的记录都将被处理。（未更改的记录将被 `Update` 方法自动忽略。）
5. 处理由 `DataAdapter Update` 方法引发的异常。当无法在数据库中进行请求的更改时，将引发异常。

（还有另外一种执行更新的方法。可以使用 `ExecuteNonQuery` 方法直接执行 SQL 更新查询。当您希望以编程方式更新特定的行并且不使用 `DataSet` 对象时，适合使用该技术。）

初始化 `DataAdapter` 以进行更新

在 ADO.NET 中，必须添加您自己的代码来向 `DataAdapter` 对象提交数据库更新。有三种完成该任务的方法：

- 可以提供您自己的更新逻辑。
- 可以使用“数据适配器配置向导”生成更新逻辑。
- 可以使用 **CommandBuilder** 对象生成更新逻辑。

建议您提供自己的更新逻辑。要节省时间，您可以使用“数据适配器配置向导”，但如果您确实使用了该向导，请尽量不要在运行时生成更新逻辑。除非迫不得已，否则请不要依赖于 **CommandBuilder** 对象，因为它会导致性能下降，并且您无法控制该对象生成的更新逻辑。此外，**CommandBuilder** 不会帮助您使用存储过程来提交更新。

对于动态生成数据访问逻辑的应用程序，如报告工具或数据提取工具，您可以使用 **CommandBuilder**。使用 **CommandBuilder** 可免除这些工具编写其自身的代码生成模块的需要。

使用存储过程

通过使用存储过程来进行更新，可以使数据库管理员实现比使用动态 SQL 时粒度更低的安全性，以及更完善的数据完整性检查。例如，存储过程除了执行请求的更新以外，还可以向审核日志中插入一项。因为数据库内部对存储过程执行脱机查询优化，所以存储过程还能够提供最佳性能。最后，因为存储过程在数据库结构和应用程序之间提供了隔离，所以更加易于维护。

因为使用存储过程的 ADO.NET 应用程序提供了许多好处，并且不比那些直接对数据库进行更改的应用程序更难实现，所以建议在几乎所有情况下都使用该方法。例外情况是当您必须使用多个后端或不支持存储过程的数据库（如 Microsoft Access）时。在这些情况下，请使用基于查询的更新。

管理并发

DataSet 对象的目的是鼓励对长期运行的活动（例如，当您远程处理数据时以及当用户与数据进行交互时）使用开放式并发。在从 **DataSet** 向数据库服务器提交更新时，有四种管理开放式并发的主要方法：

- 仅包括主键列
- 包括 WHERE 子句中的所有列
- 包括唯一键列和时间戳列
- 包括唯一键列和已修改的列

注意，后三种方法维护了数据完整性；第一种方法则没有。

仅包括主键列

该选项造成了这样一种情况，即最后的更新覆盖了所有以前的更改。

`CommandBuilder` 不支持此选项，而“数据适配器配置向导”却支持。要使用此选项，请转至 `Advanced Options` 选项卡并清除 `Use Concurrency` 复选框。

该方法不是推荐的实施策略，因为它允许用户无意中改写其他用户的更改。损害其他用户更新的完整性永远是不可取的。（该技术仅适用于单用户数据库。）

包括 WHERE 子句中的所有列

使用该选项，可防止您改写由其他用户在您的代码取出行和您的代码提交该行中挂起的更改之间这段时间内进行的更改。该选项是“数据适配器配置向导”与 `SqlCommandBuilder` 生成的 SQL 代码二者的默认行为。

由于以下原因，该方法不是推荐的实施策略：

- 如果向表中添加了额外列，查询将需要修改。
- 通常，数据库不让您比较两个 BLOB 值，因为它们太大，以至于这样的比较效率很低。（像 `CommandBuilder` 与“数据适配器配置向导”这样的工具不应该在 WHERE 子句中包含 BLOB 列。）
- 将表中的所有列与已更新行中的所有列进行比较时，会产生额外的开销。

包括唯一键列和时间戳列

使用该选项，数据库会在每次更新行后，将时间戳列更新为唯一值。（您必须在表中提供时间戳列。）目前，`CommandBuilder` 与“数据适配器配置向导”都不支持该选项。

包括唯一键列和已修改的列

通常，不推荐使用该选项，因为如果您的应用程序逻辑依赖于过期的数据字段甚至它不更新的字段，则可能会产生错误。例如，如果用户甲更改了订单数量，而用户乙更改了单价，则可能计算出错误的订单总值（数量乘以价格）。

正确地更新空字段

当数据库中的字段不包含数据值时，通常可以方便地将这些空字段视为包含特殊的空值。然而，这一心理却可能是编程错误的根源，因为数据库标准要求对空值进行特殊处理。

空字段的核⼼问题在于：当两个操作数都为空值或其中一个为空值时，普通的 SQL = 运算符总是返回 **false**。在 SQL 查询中，运算符 **IS NULL** 是检查是否存在空字段的唯一正确方法。

如果应用程序通过指定 **WHERE** 子句，使用上面介绍的技术来管理并发，则您必须在字段可能为空的任何地方包括显式的 **IS NULL** 表达式。例如，如果 **OldLastName** 为空，下面的查询将总是失败：

```
SET LastName = @NewLastName WHERE StudentID = @StudentID AND
                                   LastName = @OldLastName
```

应该按如下方式重写该查询：

```
SET LastName = @NewLastName WHERE (StudentID = @StudentID) AND
                                   ((LastName = @OldLastName) OR
                                   (OldLastName IS NULL AND LastName IS
NULL))
```

要了解如何编写上述种类的更新逻辑，一种好方法是阅读 **CommandBuilder** 工具生成的输出。

更多信息

有关数据库更新的完整论述，请参阅 David Sceppa 的《Microsoft ADO.NET》(Microsoft Press, 2002) 第 11 和 12 章。

[↑返回页首](#)

使用强类型数据集对象

强类型 **DataSet** 对象将数据库表和列呈现为对象和属性。访问是按名称执行的，而不是通过对集合进行索引来执行的。这意味着您可以使用访问字段的方法来识别强类型和非类型化 **DataSet** 对象之间的区别：

```
string n1 = myDataSet.Tables["Students"].Rows[0]["StudentName"]; //
untyped
string n2 = myDataSet.Students[0].StudentName; // strongly
typed
```

使用强类型 **DataSet** 对象有以下几点好处：

- 访问字段所需的代码可读性更高、更加简洁。
- Visual Studio .NET 代码编辑器中的智能感知功能可以在您键入时自动完成代码行。

- 编译器可以捕获强类型 **DataSet** 类型不匹配错误。在编译时检测类型错误要比在运行时检测更好。

何时使用强类型数据集

强类型 **DataSet** 很有用，因为它们使应用程序开发变得更加容易并且更少出错。对于多层应用程序的客户端而言尤其如此，在此类客户端上，重点在于需要进行多字段访问操作的图形用户界面和数据验证。

不过，如果数据库结构改变(例如，当字段名和表名被修改时)，则强类型 **DataSet** 可能会很麻烦。在此情况下，必须重新生成类型化 **DataSet** 类，并且必须修改所有相关类。

可以在同一应用程序中使用强类型方法和非类型化方法。例如，一些开发人员在客户端使用强类型 **DataSet**，在服务器上使用非类型化记录。强类型 **DataSet** 的 **.Merge** 方法可用来从非类型化 **DataSet** 中导入数据。

生成 DataSet 类

.NET 框架 SDK 和 Visual Studio.NET 都提供了实用工具，帮助您生成必要的 **DataSet** 子类。.NET 框架 SDK 涉及到使用命令行工具和编写代码。很显然，Visual Studio .NET 方法依赖于 Visual Studio .NET 开发环境，并且不要求您打开命令窗口。

无论如何生成 **DataSet** 类，都必须将新类部署到所有引用该类型化 **DataSet** 的层上。(这种情形不太常见，但如果通过使用远程处理技术在多个层中传递类型化 **DataSet**，则需要考虑这种情况。)

使用 .NET 框架实用工具

.NET 框架 SDK 包含一个称为 XML 架构定义工具的命令行实用工具，可帮助您基于 XML 架构 (.xsd) 文件生成类文件。请将该实用工具与 **DataSet** 对象的 **WriteXmlSchema** 方法结合使用，以将非类型化 **DataSet** 转化为强类型 **DataSet**。

以下命令阐明了如何从 XML 架构文件生成类文件。打开一个命令窗口，并键入以下内容：

```
C:\>xsd MyNewClass.xsd /d
```

该命令中的第一个参数是 XML 架构文件的路径。第二个参数表示要创建的类派生于 **DataSet** 类。默认情况下，该工具会生成 Visual C# .NET 类文件，但它

还可以通过添加适当的选项来生成 Visual Basic .NET 类文件。要列出该工具的可用选项，请键入以下内容：

```
xsd /?
```

在创建了新的类文件以后，请将其添加到项目中。现在，可以创建强类型 **DataSet** 类的实例，如下面的 Visual C# .NET 代码片段所示：

```
MyNewClass ds = new MyNewClass();
```

使用 Visual Studio .NET

要在 Visual Studio .NET 中生成强类型 **DataSet**，请右键单击窗体设计器窗口，然后单击 **Generate Dataset**。这将创建一个 .xsd (XML 架构定义) 文件以及一个类文件，然后将其添加到项目中。在执行此操作之前，请确保已经将一个或多个 **DataAdapter** 添加到 Windows 窗体中。注意，类文件是隐藏的。要查看该文件，请单击位于解决方案资源管理器窗口工具栏中的 **Show All Files** 按钮。该类文件与 .xsd 文件相关联。

要向强类型 **DataSet** 添加关系，请通过双击解决方案资源管理器窗口中的架构文件打开 XML 架构设计器，然后右键单击您要向其添加约束的表。在快捷菜单上，单击 **Add New Relation**。

在 Visual Studio .NET 中生成强类型 **DataSet** 的另一种方法是，右键单击项目资源管理器中的项目，选择 **Add Files**，然后选择 **dataset**。将创建一个新的 .xsd 文件。此时您可以使用服务器资源管理器连接到数据库，并将表拖到 xsd 文件上。

[返回页首](#)

处理空数据字段

以下是几点帮助您在 .NET 数据体系结构中正确使用空字段值的提示：

- 始终使用 **System.DBNull** 类设置空字段的值。不要使用由 C# 或 Visual Basic .NET 提供的空值。例如：

```
rowStudents["Nickname"] = DBNull.Value // correct!
```
- 强类型 **DataSet** 包含两个针对 **DataRow** 执行操作的附加方法：一个用于检查列是否含有空值，另一个用于将列值设置为空。以下代码片段显示了这两个方法：

```
If (tds.rowStudent[0].IsPhoneNoNull()) {a€_.}  
tds.rowStudent[0].SetPhoneNoNull()
```

- 请始终使用 `DataRow` 类（或在上一个项目符号中给出的强类型等效类）的 `IsNull` 方法来测试数据库中的空值。该方法是测试数据库空值的唯一受支持的方式。
- 如果数据字段可能包含空值，请确保在需要非空值的上下文中使用该值之前，对其进行测试（通过 `IsNull` 方法）。这方面的一个典型示例是可能为空的 `Integer` 值数据字段。注意，.NET 运行时 `Integer` 数据类型不包含空值。以下为一个示例：
 - ```
int i = rowStudent["ZipCode"]; // throws exception if null!
```
- 使用强类型 `DataSet` .xsd 文件的 `nullValue` 批注来配置如何映射数据库中的空值。默认情况下会引发异常；然而，为了获得更高粒度的控制，可以将该类配置为使用指定的值（如 `String.Empty`）来替换空值。

[返回页首](#)

## 事务处理

几乎所有更新数据源的、面向商业的应用程序都需要事务处理支持。事务处理用来确保一个或多个数据源中包含的系统状态的完整性，其方法是提供以下四项由著名的缩写词 ACID 表示的基本保证：原子性、一致性、隔离性和持久性。

例如，考虑一个处理采购订单的基于 Web 的零售应用程序。每个订单都需要三种截然不同的操作，这些操作涉及三种数据库更新：

- 必须按订购数量降低存货水平。
- 必须按采购数量借记客户的信用级别。
- 必须将新订单添加到订单数据库中。

将上述三种截然不同的操作作为一个单位以原子方式执行是至关重要的。它们必须或者全部成功，或者全部失败：任何其他情况都将损害数据完整性。事务处理能够提供这一保证以及其他一些保证。

有关事务处理基本原理的详细信息，请参阅

<http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpcontransactionprocessingfundamentals.asp>。

有许多可以用来将事务管理集成到数据访问代码中的方法。每种方法都适用于两种基本编程模型中的一种：

- **手动事务处理**。直接在组件代码或存储过程中分别编写使用 ADO.NET 或 Transact-SQL 事务处理支持功能的代码。
- **原子 (COM+) 事务**。向 .NET 类中添加声明性属性，从而指定对象在运行时的事务处理要求。该模型使您可以方便地将多个组件配置为在同一事务中执行工作。

这两种技术都可以用来执行本地事务（即针对单个资源管理器如 SQL Server 2000 执行的事务）或分布式事务（即针对远程计算机上的多个资源管理器执行的事务），尽管原子事务处理模型大大简化了分布式事务处理。

您可能倾向于使用原子（COM+）事务处理，以便从更容易的编程模型中受益。在有很多执行数据库更新的组件的系统中，这种编程模型的好处尤其明显。但是，在许多方案中，您应该避免这种事务处理模式所带来的额外开销和性能损失。

本节将提供一些指导，帮助您基于特定的应用程序方案来选择最合适的模型。

## 选择事务处理模型

在选择事务处理模型之前，您应该考虑是否真正需要事务处理。事务是服务器应用程序所使用的最昂贵的单一资源，如果不必要地加以使用，还会降低可伸缩性。考虑下列对事务的使用进行控制的准则：

- 仅当需要在一组操作中获得锁并且需要贯彻 ACID 规则时，才执行事务处理。
- 使事务处理时间尽可能地短，以最大限度减少占有数据库锁的时间。
- 永远不要让客户端控制事务的生存期。
- 不要对单个 SQL 语句使用事务。SQL Server 自动将每个语句作为单个事务运行。

## 原子事务处理与手动事务处理

尽管自动事务处理的编程模型在某种程度上得到了简化，尤其是在多个组件都执行数据库更新时，但手动执行本地事务处理总是要快很多，因为它们不需要与 Microsoft DTC 进行交互。即使您要针对单个本地资源管理器（如 SQL Server）使用原子事务，也是这样的（尽管可以减少性能损失），原因是手动的本地事务处理能够避免与 DTC 进行任何不必要的进程间通讯（IPC）。

在以下情况下，请使用手动事务处理：

- 针对单个数据库执行事务处理。

在以下情况下，请使用自动事务处理：

- 要求单个事务跨越多个远程数据库。
- 您要求单个事务包括多个资源管理器，例如，一个数据库和一个 Windows 2000 消息队列（以前称为 MSMQ）资源管理器。

**注** 请避免混用事务处理模型。二者只能择其一。

在认为性能可以满足要求的应用程序场合中，选择使用自动事务处理（即使针对单个数据库）以简化编程模型是合理的。自动事务处理使得多个组件可以容易地执行属于同一事务的操作。

## 使用手动事务处理

使用手动事务处理时，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 事务处理支持功能的代码。在大多数情况下，应该选择在存储过程中控制事务处理，因为该方法提供了优良的封装，并且从性能角度看也与使用 ADO.NET 代码执行事务处理相当。

## 通过 ADO.NET 执行手动事务处理

ADO.NET 支持一种事务对象，可以使用该对象来开始新的事务，然后对应该提交还是回滚该事务进行显式控制。该事务对象与单个数据库连接相关联，并且通过该连接对象的 **BeginTransaction** 方法获得。调用该方法并不暗示着随后的命令是在该事务的上下文中发出的。必须将各个命令与事务显式关联，方法是设置相应命令的 **Transaction** 属性。可以将多个命令对象与事务对象相关联，从而将针对单个数据库的多个操作组合到单个事务中。

有关使用 ADO.NET 事务处理代码的示例，请参阅附录中的如何编写 ADO.NET 手动事务处理代码。

## 更多信息

- ADO.NET 手动事务处理的默认隔离级别是“提交读”，这意味着在读取数据时数据库拥有共享锁，但数据在事务处理结束之前可以更改。这有可能导致不可重复的读取，或者导致幻像数据。可以更改隔离级别，方法是将事务对象的 **IsolationLevel** 属性设置为 **IsolationLevel** 枚举类型所定义的枚举值之一。
- 必须在经过认真考虑后，为事务选择适当的隔离级别。这涉及到在数据一致性和性能之间进行权衡。最高隔离级别（Serialized）提供绝对的数据一致性，但代价是降低系统总吞吐量。较低的隔离级别可使应用程序具有更好的可伸缩性，但同时会增加因为数据不一致导致出错的可能性。对于在大多数时间读取数据而很少写数据的系统而言，使用较低的隔离级别可能是适当的。
- 有关选择适当事务隔离级别的有价值信息，请参阅 Microsoft Press? 书籍《Inside SQL Server 2000》（作者：Kalen Delaney）。

## 通过存储过程执行手动事务处理

还可以通过在存储过程中使用 Transact-SQL 语句来直接控制手动事务处理。例如,可以通过使用单个存储过程(该存储过程使用 Transact-SQL 事务处理语句,如 **BEGIN TRANSACTION**、**END TRANSACTION** 和 **ROLLBACK TRANSACTION**) 来执行事务性操作。

## 更多信息

- 如果需要,可以通过在存储过程中使用 **SET TRANSACTION ISOLATION LEVEL** 语句来控制事务隔离级别。Read Committed 是 SQL Server 的默认隔离级别。有关 SQL Server 隔离级别的详细信息,请参阅 SQL Server 联机图书“Accessing and Changing Relation Data”部分中的“Isolation Levels”。
- 有关说明如何使用 Transact-SQL 事务处理语句来执行事务性更新的代码示例,请参阅附录中的如何使用 Transact-SQL 执行事务处理。

## 使用自动事务处理

自动事务处理简化了编程模型,因为它们不要求显式开始新事务或者显式提交或中止事务。然而,自动事务处理的最大优势是它们与 DTC 协同工作,这使得单个事务可以跨越多个分布式数据源。在大型分布式应用程序中,这一优势可能很有意义。尽管可以通过直接编写 DTC 来手动控制分布式事务,但自动事务处理大幅度地简化了该任务,并且非常适合于基于组件的系统。例如,很容易以声明方式配置多个组件来执行组成单个事务的工作。

自动事务处理依赖于 COM+ 提供的分布式事务处理支持功能,因此,只有服务组件(即从 **ServicedComponent** 类派生的组件)可以使用自动事务处理。

配置类以执行自动事务处理:

- 从位于 **System.EnterpriseServices** 命名空间中的 **ServicedComponent** 类派生该类。
- 通过使用 **Transaction** 属性来定义该类的事务处理要求。**TransactionOption** 枚举类型中提供的值确定了将如何在 COM+ 目录中配置该类。其他可以使用该属性建立的属性包括事务隔离级别和超时。
- 要明确避免必须在事务处理结果中进行表决,可以使用 **AutoComplete** 属性来给方法加上批注。如果这些方法引发异常,事务将自动中止。注意,如果需要,您仍然可以直接对事务处理结果进行表决。有关详细信息,请参阅本文下面的确定事务处理结果一节。

## 更多信息

- 有关 COM+ 自动事务处理的详细信息，请在 Platform SDK 文档中搜索“Automatic Transactions Through COM+”（通过 COM+ 执行自动事务处理）。
- 有关事务性 .NET 类的示例，请参阅附录中的如何编写事务性 .NET 类。

## 配置事务隔离级别

COM+ 1.0 版（即在 Windows 2000 上运行的 COM+）的事务隔离级别是 Serialized。尽管这提供了最高程度的隔离，这样的保护是以牺牲性能为代价的。系统的总吞吐量会降低，因为涉及到的资源管理器（通常是数据库）必须在事务处理期间同时拥有读锁和写锁。在此过程中，所有其他事务都将被阻塞，这可能会对应用程序的伸缩能力产生重大影响。

COM+ 1.5 版（它随附在 Microsoft Windows .NET 中）允许在 COM+ 目录中以组件为单位对事务隔离级别进行配置。与事务中的根组件相关联的设置确定了该事务的隔离级别。此外，属于同一事务流的内部子组件所具有的事务级别不得高于由根组件所定义的事务级别。如果不满足此项要求，当相关子组件被实例化时，将产生错误。

对于 .NET 托管类，**Transaction** 属性支持公共 **Isolation** 属性。可以使用该属性以声明方式指定特定的隔离级别，如下面的代码所示。

```
[Transaction(TransactionOption.Supported,
Isolation=TransactionIsolationLevel.ReadCommitted)]
public class Account : ServicedComponent
{
 . . .
}
```

## 更多信息

- 有关可配置的事务隔离级别和其他 Windows .NET COM+ 增强功能的详细信息，请参阅 MSDN Magazine 文章“Windows XP:Make Your Components More Robust with COM+ 1.5 Innovations”，网址为 <http://msdn.microsoft.com/msdnmag/issues/01/08/ComXP/default.aspx>。

## 确定事务结果

自动事务处理的结果由事务中止标志以及一致性标志（它们位于单事务流中的所有事务性组件的上下文中）的状态控制。当事务流中的根组件被停用（并且控制被返回到调用方）时，将确定事务结果。图 5 对此进行了说明，该图显示了一个传统的银行资金转帐事务。

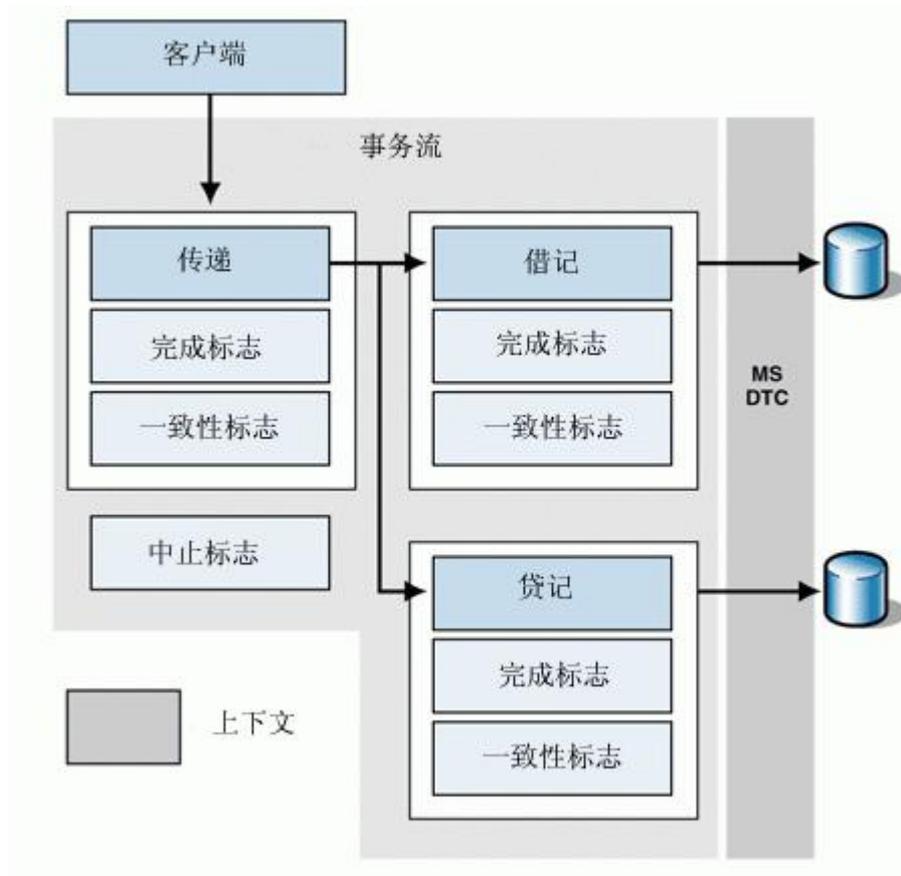


图 1.5.事务流和上下文

事务的结果是在根对象（在该示例中，为 **Transfer** 对象）被停用并且客户端的方法调用返回时确定的。如果任意上下文中的任意一致性标志被设置为 `false`，或者如果事务中止标志被设置为 `true`，则底层的物理 DTC 事务被中止。

可以用下列两种方式之一从 .NET 对象中控制事务结果：

- 可以使用 **AutoComplete** 属性给方法加上批注，并且让 .NET 自动发出您的表决以控制事务的结果。使用该属性时，如果方法引发异常，一致性标志将被自动设置为 `false`（最终会导致事务中止）。如果方法返回且未引发异常，则一致性标志被设置为 `true`，这表示该组件同意提交事务。这一点不能保证，因为它依赖于同一事务流中其他对象的表决。

- 可以调用 `ContextUtil` 类的静态 `SetComplete` 或 `SetAbort` 方法，这两个方法分别将一致性标志设置为 `true` 或 `false`。

严重度大于 10 的 SQL Server 错误会导致托管数据提供程序引发 `SQLException` 类型的异常。如果您的方法捕获并处理了该异常，请确保手动表决以中止该事务，或者（对于被标记为 `[AutoComplete]` 的方法）确保将该异常传播给调用方。

## [AutoComplete] 方法

对于用 `AutoComplete` 属性标记的方法，请执行以下任何一种操作：

- 将 `SQLException` 沿调用堆栈向上传播。
- 将 `SQLException` 包装到外部异常中，并将后者传播给调用方。您可能希望将该异常包装在对调用方更有意义的异常类型中。

如果不传播该异常，将导致对象不会表决以中止该事务（尽管发生了数据库错误）。这意味着：由共享同一事务流的其他对象所进行的其他成功操作可能被提交。

下面的代码捕获了 `SQLException`，然后将其直接传播到调用方。该事务最终将中止，因为该对象的一致性标志将在其被停用时自动设置为 `false`。

```
[AutoComplete]
void SomeMethod()
{
 try
 {
 // Open the connection, and perform database operation

 }
 catch (SQLException sqllex)
 {
 LogException(sqllex); // Log the exception details
 throw; // Rethrow the exception, causing the
consistent // flag to be set to false.
 }
 finally
 {
 // Close the database connection

 }
}
```

## 非 [AutoComplete] 方法

对于没有用 `AutoComplete` 属性进行标记的方法，必须执行以下操作：

- 在 `catch` 块中调用 `ContextUtil.SetAbort`，从而表决中止该事务。这会将一致性标志设置为 `false`。
- 如果未发生异常，则调用 `ContextUtil.SetComplete` 以表决提交该事务。这会将一致性标志设置为 `true`（其默认状态）。

以下代码阐明了该方法。

```
void SomeOtherMethod()
{
 try
 {
 // Open the connection, and perform database operation
 . . .
 ContextUtil.SetComplete(); // Manually vote to commit the
transaction
 }
 catch (SQLException sqllex)
 {
 LogException(sqllex); // 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** 如果有多个 `catch` 块，则在方法开头调用一次 `ContextUtil.SetAbort`，并且在 `try` 块结尾调用 `ContextUtil.SetComplete` 将会更加容易。这样，就不需要在每个 `catch` 块中重复调用 `ContextUtil.SetAbort`。这些方法所确定的一致性标志设置仅当方法返回时才有意义。

必须始终将异常（或被包装的异常）沿调用堆栈向上传播，因为这样可使调用代码知道该事务将失败。这使得调用代码可以进行优化。例如，在银行资金转帐方案中，如果借记操作已经失败，则转帐组件可以决定不执行贷记操作。

如果您将一致性标志设置为 `false`，然后返回并且不引发异常，则调用代码无法知道事务将要失败。尽管可以返回一个 `Boolean` 值或者设置一个 `Boolean` 输出参数，您应该始终如一并引发异常以表明发生了错误。这样会产生更加清晰、更加一致的代码，并且具有标准的错误处理方法。

[^返回页首](#)

## 数据分页

对数据进行分页是分布式应用程序中的共同要求。例如，可能要向用户展示一个书籍清单，此时，同时显示整个清单可能被禁止。用户将希望对数据执行熟悉的操作，如查看下一页或上一页数据，或者跳到该清单的开头或结尾。

本节将讨论用于实现此功能的选项，以及每个选项对可伸缩性和性能的影响。

## 比较几种选择

数据分页的选择包括：

- 通过 `SqlDataAdapter` 的 `Fill` 方法，用查询的一系列结果填充 `DataSet`
- 通过 COM 互操作性使用 ADO，并且使用服务器端游标
- 使用存储过程来手动实现数据分页

数据分页的最佳选择取决于下面列出的因素：

- 可伸缩性要求
- 性能要求
- 网络带宽
- 数据库服务器内存和功能
- 中间层服务器内存和功能
- 希望进行分页的、由查询返回的行数
- 数据页的大小

性能测试表明，使用存储过程的手动方法在广泛的压力级别下提供了最佳性能。不过，因为手动方法在服务器上完成其工作，所以如果网站的大部分功能都依赖于数据分页功能，则服务器压力级别可能成为一个重要问题。要确保该方法适合您的特定环境，应该针对您的特定要求测试所有选择。

下面讨论了各种选择。

## 使用 SqlDataAdapter 的 Fill 方法

如上所述, 可通过 `SqlDataAdapter` 用数据库中的数据来填充 `DataSet`。重载的 `Fill` 方法之一 (如以下代码所示) 采用两个整数索引值作为参数。

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

`startRecord` 值表示起始记录的从零开始的索引。`maxRecords` 值表示要复制到新 `DataSet` 中的记录数 (从 `startRecord` 开始)。

`SqlDataAdapter` 在内部使用 `SqlDataReader` 来执行查询并返回结果。`SqlDataAdapter` 读取结果并基于从 `SqlDataReader` 中读取的数据创建一个 `DataSet`。`SqlDataAdapter` 通过 `startRecord` 和 `maxRecords` 将所有结果复制到刚生成的 `DataSet` 中, 并且丢弃它不需要的结果。这意味着可能会将大量不需要的数据通过网络提取到数据访问客户端, 而这正是该方法的主要缺点。

例如, 如果有 1,000 条记录, 并且只需要记录 900 到 950, 则仍然会将前面的 899 条记录通过网络提取到客户端并丢弃。这一开销对于小型结果集而言可能很小, 但当您对较大的数据集进行分页时, 这一开销可能会很大。

## 使用 ADO

另一种实现分页的选择是使用基于 COM 的 ADO 进行分页。该选项背后的主要动机是获取对服务器端游标的访问, 该游标通过 ADO `Recordset` 对象公开。可以将 `Recordset` 游标位置设置为 `adUseServer`。如果您的 OLE DB 提供程序支持它 (SQLOLEDB 就支持), 这将导致对服务器端游标的使用。然后, 您可以使用该游标直接导航到起始记录, 而无须将所有记录通过网络提取到数据访问客户端代码。

该方法有以下两个主要缺点:

- 大多数情况下, 您需要将在 `Recordset` 对象中返回的记录转换到 `DataSet` 中, 以便在客户端托管代码中使用。尽管 `OleDbDataAdapter` 的确重载了 `Fill` 方法, 采用 ADO `Recordset` 对象作为参数并将其转换到 `DataSet` 中, 但没有用来指定起始和结束记录的工具。唯一现实的选择是移动到 `Recordset` 对象中的起始记录, 依次处理各个记录, 并将数据手动复制到手动生成的新 `DataSet` 中。这样做 (特别是通过 COM Interop

调用的开销)有可能完全抵消不通过网络提取额外数据的优点,尤其是对小型 **DataSet** 而言。

- 在从服务器提取所需数据花费的这段时间内,您需要使连接和服务器端游标保持打开状态。在数据库服务器上打开和维护时,游标通常是一种昂贵的资源。尽管该选择可以提高性能,它还可能由于在服务器上长期消耗宝贵的资源而降低可伸缩性。

## 使用手动实现

本节讨论的最后一个对数据进行分页的选择是,通过使用存储过程为应用程序手动实现分页功能。对于包含唯一键的表而言,可以相对容易地实现存储过程。对于不包含唯一键的表(您不应该具有许多这样的表)而言,该过程更复杂一些。

### 对包含唯一键的表进行分页

如果表包含唯一键,可以在 **WHERE** 子句中使用该键来创建从特定行开始的结果集。将这种方法与用于限制结果集大小的 **SET ROWCOUNT** 语句或 SQL Server **TOP** 语句结合起来,可以提供一种有效的分页机制。下面的存储过程代码阐明了这一方法:

```
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
```

该存储过程的调用方简单地维护 **lastProductID** 值,并且在连续调用之间将其递增或递减所选的页面大小。

### 对不包含唯一键的表进行分页

如果您要对其进行分页的表不包含唯一键,可以考虑通过某种方法(例如,使用标识列)添加一个唯一键。这将使您能够实现前面讨论的分页解决方案。

对于不包含唯一键的表,只要您能够通过将属于结果集的其他两个或更多个字段组合起来生成唯一性,则仍然能够实现有效的分页解决方案。

例如，请考虑下表：

| Col1 | Col2 | Col3 | 其他列 § |
|------|------|------|-------|
| 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，可以通过组合 **Col1**、**Col2** 和 **Col3** 来生成唯一性。因此，可以通过使用以下存储过程中阐明的方法来实现分页机制。

```
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
```

客户端保持由存储过程返回的 **KeyField** 列的最后值，并将其重新插入到存储过程中以控制表分页。

尽管手动实现增加了数据库服务器的负担，但它能够避免通过网络传递不必要的数据。性能测试表明，该方法能够在广泛的压力级别下正常工作。然而，根据与数据分页功能有关的网站工作量的不同，在服务器上执行手动分页可能会影响应用程序的可伸缩性。您应该在自己的环境中运行性能测试，以便找到适合于特定应用程序方案的最佳方法。

[返回页首](#)

## 附录

## 如何启用 .NET 类的对象构建

可以使用企业 (COM+) 服务来启用 .NET 托管类, 以便进行对象构建。

### 启用 .NET 托管类

1. 从位于 `System.EnterpriseServices` 命名空间中的 `ServicedComponent` 类派生类。
2. `using System.EnterpriseServices;`
3. `public class DataAccessComponent : ServicedComponent`
4. 用 `ConstructionEnabled` 属性修饰该类, 并根据需要指定默认的构建字符串。该默认值存放在 COM+ 目录中。管理员可以使用“组件服务” Microsoft 管理控制台 (MMC) 单元来维护该值。
5. `[ConstructionEnabled(Default="default DSN")]`
6. `public class DataAccessComponent : ServicedComponent`
7. 提供虚拟 `Construct` 方法的重写实现。该方法在对象的语言特定构造函数之后调用。COM+ 目录中维护的构建字符串作为该方法的唯一参数提供。
8. `public override void Construct( string constructString )`
9. `{`
10. `// Construct method is called next after constructor.`
11. `// The configured DSN is supplied as the single argument`
12. `}`
13. 通过 `AssemblyKey` 文件或 `AssemblyKeyName` 属性对程序集进行签名, 以便为其提供强名称。向 COM+ 服务注册的程序集必须具有强名称。有关强名称程序集的详细信息, 请参阅 <http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconworkingwithstrongly-namedassemblies.asp>。
14. `[assembly:AssemblyKeyFile("DataServices.snk")]`
15. 要支持动态 (惰性) 注册, 请使用程序集级别属性 `ApplicationName` 和 `ApplicationActivation`, 分别指定用于容纳程序集组件的 COM+ 应用程序的名称以及该应用程序的激活类型。有关程序集注册的详细信息, 请参阅 <http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconregisteringservicedcomponents.asp>。
16. `// the ApplicationName attribute specifies the name of the`
17. `// COM+ Application which will hold assembly components`
18. `[assembly : ApplicationName("DataServices")]`
- 19.
20. `// the ApplicationActivation.ActivationOption attribute specifies`
21. `// where assembly components are loaded on activation`
22. `// Library : components run in the creator's process`

23. // Server : components run in a system process, dllhost.exe
24. [assembly: ApplicationActivation(ActivationOption.Library)]

以下代码片段显示了一个名为 **DataAccessComponent** 的服务组件,它使用 COM+ 构建字符串来获取数据库连接字符串。

```
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;
 }
}
```

## 如何使用 **SqlDataAdapter** 来检索多个行

以下代码阐明了如何使用 **SqlDataAdapter** 对象发出可生成 **DataSet** 或 **DataTable** 的命令。它从 SQL Server Northwind 数据库中检索一组产品类别。

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

public DataTable RetrieveRowsWithDataTable()
{
 using (SqlConnection conn = new SqlConnection(connectionString))
 {
 conn.Open();
 SqlCommand cmd = new SqlCommand("DATRetrieveProducts", conn);
 cmd.CommandType = CommandType.StoredProcedure;
 SqlDataAdapter adapter = new SqlDataAdapter(cmd);
 DataTable dataTable = new DataTable("Products");
 adapter.Fill(dataTable);
 return dataTable;
 }
}
```

## 使用 SqlDataAdapter 生成 DataSet 或 DataTable

1. 创建一个 SqlCommand 对象以调用该存储过程，并将其与一个 SqlConnection 对象（显示）或连接字符串（不显示）相关联。
2. 创建一个新的 SqlDataAdapter 对象并将其与 SqlCommand 对象相关联。
3. 创建一个 DataTable（也可以创建一个 DataSet）对象。使用构造函数参数来命名 DataTable。
4. 调用 SqlDataAdapter 对象的 Fill 方法，用检索到的行填充 DataSet 或 DataTable。

## 如何使用 SqlDataReader 来检索多个行

以下代码片段阐明了可检索多个行的 SqlDataReader 方法。

```
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();
 try
 {
 while (reader.Read())
 {
 Console.WriteLine("{0} {1} {2}",
 reader.GetInt32(0).ToString(),
 reader.GetString(1));
 }
 }
 finally
 {
 reader.Close(); // Also closes the connection due to the
 // CommandBehavior enum used when generating the
reader
 }
}
```

## 使用 SqlDataReader 检索行

1. 创建一个用来执行存储过程的 `SqlCommand` 对象，并将其与一个 `SqlConnection` 对象相关联。
2. 打开连接。
3. 通过调用 `SqlCommand` 对象的 `ExecuteReader` 方法创建一个 `SqlDataReader` 对象。

4. 要从流中读取数据,请调用 `SqlDataReader` 对象的 `Read` 方法来检索行,并使用类型化访问器方法(如 `GetInt32` 和 `GetString` 方法)来检索列值。
5. 使用完读取器后,请调用其 `Close` 方法。

## 如何使用 `XmlReader` 检索多个行

可以使用 `SqlCommand` 对象来生成 `XmlReader` 对象,后者可提供对 XML 数据的基于流的只进访问。命令(通常为存储过程)必须产生基于 XML 的结果集,对于 SQL Server 2000 而言,该结果集通常包含一个带有有效 **FOR XML** 子句的 **SELECT** 语句。以下代码片段阐明了该方法:

```
public void RetrieveAndDisplayRowsWithXmlReader()
{
 using(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(); // XmlTextReader does not support IDisposable so
it can't be
 // used within a using keyword
 }
}
```

上述代码使用了以下存储过程:

```
CREATE PROCEDURE DATRetrieveProductsXML
AS
SELECT * FROM PRODUCTS
```

```
FOR XML AUTO
GO
```

## 使用 XmlReader 检索 XML 数据

1. 创建一个 `SqlCommand` 对象来调用可生成 XML 结果集的存储过程(例如, 在 `SELECT` 语句中使用 `FOR XML` 子句)。将该 `SqlCommand` 对象与某个连接相关联。
2. 调用 `SqlCommand` 对象的 `ExecuteXmlReader` 方法, 并且将结果分配给只进 `XmlTextReader` 对象。当您不需要对返回的数据进行任何基于 XML 的验证时, 这是应该使用的最快类型的 `XmlReader` 对象。
3. 使用 `XmlTextReader` 对象的 `Read` 方法来读取数据。

## 如何使用存储过程输出参数来检索单个行

借助于命名的输出参数, 可以调用在单个行内返回检索到的数据项的存储过程。以下代码片段使用存储过程来检索 Northwind 数据库的 Products 表中包含的特定产品的产品名称和单价。

```
void GetProductDetails(int ProductID,
 out string ProductName, out decimal UnitPrice)
{
 using(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;

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;
}
}
```

## 使用存储过程输出参数来检索单个行

1. 创建一个 `SqlCommand` 对象并将其与一个 `SqlConnection` 对象相关联。
2. 通过调用 `SqlCommand` 的 `Parameters` 集合的 `Add` 方法来设置存储过程参数。默认情况下，参数都被假设为输入参数，因此必须显式设置任何输出参数的方向。

**注** 一种良好的习惯做法是显式设置所有参数（包括输入参数）的方向。

3. 打开连接。
4. 调用 `SqlCommand` 对象的 `ExecuteNonQuery` 方法。这将填充输出参数（并可能填充返回值）。
5. 通过使用 `Value` 属性，从适当的 `SqlParameter` 对象中检索输出参数。
6. 关闭连接。

上述代码片段调用了以下存储过程。

```
CREATE PROCEDURE DATGetProductDetailsSPOutput
@ProductID int,
@ProductName nvarchar(40) OUTPUT,
@UnitPrice money OUTPUT
AS
SELECT @ProductName = ProductName,
 @UnitPrice = UnitPrice
FROM Products
WHERE ProductID = @ProductID
GO
```

## 如何使用 SqlDataReader 来检索单个行

可以使用 `SqlDataReader` 对象来检索单个行，尤其是可以从返回的数据流中检索需要的列值。以下代码片段对此进行了说明。

```
void GetProductDetailsUsingReader(int ProductID,
 out string ProductName, out decimal UnitPrice)
{
 using(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;
 conn.Open();
 using(SqlDataReader reader = cmd.ExecuteReader())
 {
 if(reader.Read()) // Advance to the one and only row
 {
 // Return output parameters from returned data stream
 ProductName = reader.GetString(0);
 UnitPrice = reader.GetDecimal(1);
 }
 }
 }
}
```

## 使用 SqlDataReader 对象来返回单个行

1. 建立 `SqlCommand` 对象。
2. 打开连接。
3. 调用 `SqlDataReader` 对象的 `ExecuteReader` 方法。
4. 通过 `SqlDataReader` 对象的类型化访问器方法（在这里，为 `GetString` 和 `GetDecimal`）来检索输出参数。

上述代码片段调用了以下存储过程。

```
CREATE PROCEDURE DATGetProductDetailsReader
@ProductID int
AS
SELECT ProductName, UnitPrice FROM Products
WHERE ProductID = @ProductID
GO
```

## 如何使用 **ExecuteScalar** 来检索单个项

**ExecuteScalar** 方法专门适用于仅返回单个值的查询。如果查询返回多个列和/或行，**ExecuteScalar** 将只返回第一行的第一列。

以下代码说明了如何查找与特定产品 ID 相对应的产品名称：

```
void GetProductNameExecuteScalar(int ProductID, out string
ProductName)
{
 using(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);
 conn.Open();
 ProductName = (string)cmd.ExecuteScalar();
 }
}
```

## 使用 **ExecuteScalar** 来检索单个项

1. 建立一个 **SqlCommand** 对象来调用存储过程。
2. 打开连接。
3. 调用 **ExecuteScalar** 方法。注意，该方法返回一个对象类型。它包含检索到的第一列的值，并且必须转化为适当的类型。
4. 关闭连接。

上述代码使用了以下存储过程：

```
CREATE PROCEDURE LookupProductNameScalar
```

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

## 如何使用存储过程输出或返回参数来检索单个项

可以使用存储过程输出或返回参数来查找单个值。以下代码阐明了输出参数的用法：

```
void GetProductNameUsingSPOutput(int ProductID, out string
ProductName)
{
 using(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;

 conn.Open();
 cmd.ExecuteNonQuery();
 ProductName = paramPN.Value.ToString();
 }
}
```

## 使用存储过程输出参数来检索单个值

1. 建立一个 `SqlCommand` 对象来调用存储过程。
2. 通过将 `SqlParameter` 添加到 `SqlCommand` 的 `Parameters` 集合中，设置任何输入参数和单个输出参数。
3. 打开连接。
4. 调用 `SqlCommand` 对象的 `ExecuteNonQuery` 方法。
5. 关闭连接。

6. 通过使用输出 `SqlParameter` 的 `Value` 属性来检索输出值。

上述代码使用了以下存储过程。

```
CREATE PROCEDURE LookupProductNameSPOutput
@ProductID int,
@ProductName nvarchar(40) OUTPUT
AS
SELECT @ProductName = ProductName
FROM Products
WHERE ProductID = @ProductID
GO
```

以下代码阐明了如何使用返回值来指明是否存在特定行。从编码角度来看，这类类似于使用存储过程输出参数，不同之处在于必须将 `SqlParameter` 方向显式设置为 `ParameterDirection.ReturnValue`。

```
bool CheckProduct(int ProductID)
{
 using(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;
 conn.Open();
 cmd.ExecuteNonQuery();
 }
 return (int)paramRet.Value == 1;
}
```

## 通过使用存储过程返回值来检查是否存在特定行

1. 建立一个 `SqlCommand` 对象来调用存储过程。
2. 设置一个输入参数，该参数含有要访问的行的主键值。
3. 设置单个返回值参数。将一个 `SqlParameter` 对象添加到 `SqlCommand` 的 `Parameters` 集合中，并将其方向设置为 `ParameterDirection.ReturnValue`。
4. 打开连接。
5. 调用 `SqlCommand` 对象的 `ExecuteNonQuery` 方法。

6. 关闭连接。
7. 通过使用返回值 `SqlParameter` 的 `Value` 属性来检索返回值。

上述代码使用了以下存储过程。

```
CREATE PROCEDURE CheckProductSP
@ProductID int
AS
IF EXISTS(SELECT ProductID
 FROM Products
 WHERE ProductID = @ProductID)
 return 1
ELSE
 return 0
GO
```

## 如何使用 `SqlDataReader` 来检索单个项

可以使用 `SqlDataReader` 对象并通过调用命令对象的 `ExecuteReader` 方法来获取单个输出值。这要求编写稍微多一点的代码，因为必须调用 `SqlDataReader` `Read` 方法，然后通过该读取器的访问器方法之一来检索需要的值。以下代码阐明了 `SqlDataReader` 对象的用法。

```
bool CheckProductWithReader(int ProductID)
{
 using(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;
 conn.Open();
 using(SqlDataReader reader = cmd.ExecuteReader(
 CommandBehavior.SingleResult))
 {
 if(reader.Read())
 {
 return (reader.GetInt32(0) > 0);
 }
 }
 return false;
 }
}
```

```
 }
}
```

上述代码采用了以下存储过程。

```
CREATE PROCEDURE CheckProductExistsWithCount
@ProductID int
AS
SELECT COUNT(*) FROM Products
WHERE ProductID = @ProductID
GO
```

## 如何编写 ADO.NET 手动事务处理代码

以下代码显示了如何充分利用 SQL Server .NET 数据提供程序所提供的事务处理支持，通过事务来保护资金转帐操作。该操作在同一数据库中的两个帐户之间转移资金。

```
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;
}
}
}
```

## 如何使用 Transact-SQL 执行事务处理

以下存储过程阐明了如何在 Transact-SQL 存储过程内部执行事务性资金转帐操作。

```
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** 语句来手动控制该事务。

## 如何编写事务性 .NET 类

以下示例代码显示了三个服务性 .NET 托管类，这些类经过配置以执行自动事务处理。每个类都使用 **Transaction** 属性进行了批注，该属性的值确定是否应该启动新的事务流，或者该对象是否应该共享其直接调用方的事务流。这些组件协同工作来执行银行资金转帐任务。**Transfer** 类被使用 **RequiresNew** 事务属性进行了配置，而 **Debit** 和 **Credit** 被使用 **Required** 进行了配置。结果，所有这三个对象在运行时都将共享同一事务。

```
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 sqllex)
 {
 // Handle and log exception details
 // Wrap and propagate the exception
 throw new TransferException("Transfer Failure", sqllex);
 }
}
}
[Transaction(TransactionOption.Required)]
public class Credit : ServicedComponent
{
 [AutoComplete]
 public void CreditAccount(string account, decimal amount)
 {
 try
 {
 using(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));
 conn.Open();
 cmd.ExecuteNonQuery();
 }
 }
 catch(SQLException sqllex){
 // Log exception details here
 throw; // Propagate exception
 }
 }
}
[Transaction(TransactionOption.Required)]
public class Debit : ServicedComponent
{
 public void DebitAccount(string account, decimal amount)
 {
 try
 {
```

```
using(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));
 conn.Open();
 cmd.ExecuteNonQuery();
}
}
catch (SqlException sqllex)
{
 // Log exception details here
 throw; // Propagate exception back to caller
}
}
```