神通数据库

OLEDB/ADO用户手册

版本 7.0

天津神舟通用数据技术有限公司

2010 年 1 月

版权声明

神通数据库是天津神舟通用数据技术有限公司开发的数据库管理系统软件产品。神通数据库的版权归天津神舟通用数据技术有限公司，任何侵犯版权的行为将追究法律责任。

《神通数据库 OLEDB/ADO 用户手册》的版权归天津神舟通用数据技术有限公司所有。

未经天津神舟通用数据技术有限公司的书面准许，不得将本手册的任何部分以任何形式、采用任何手段(电子的或机械的，包括照相复制或录制)、或为任何目的，进行复制或扩散。

(c)Copyright 2010 天津神舟通用数据技术有限公司。版权所有，翻制必究。

天津神舟通用数据技术有限公司不对因为使用该软件、用户手册或由于该软件、用户手册中的缺陷所造成的任何损失负责。

阅读指南

〖阅读对象〗

本手册是为通过 OLEDB/ADO 接口来访问神通数据库的用户编写的。神通数据库的用户在使用 OLEDB/ADO 接口之前应当认真阅读本手册，以便熟悉 OLEDB/ADO 的使用，并最终通过它来访问神通数据库系统。

〖内容简介〗

OLEDB/ADO 是 Microsoft 的 UDA(Universal Data Access，一致数据访问) 技术的一个具体实现，它为关系型或非关系型数据访问提供了一致的访问接口，为企业级 Intranet 应用多层软件结构提供了数据接口标准。其中， OLEDB 是一组符合 COM 规范的接口，ADO 则以 OLEDB 为基础，提供了更高层的软件接口，可以在各种脚本语言或一些宏语言中直接使用。OLE DB 一般使用于对性能要求比较高的开发工具，应用和底层组件。本手册根据神通数据库的特点，阐述了如下问题：

什么是 OLEDB/ADO 接口？

OLEDB/ADO 具有哪些特性？

如何通过 OLEDB/ADO 连接神通数据库？

本手册还提供了使用 OLEDB/ADO 进行编程的典型范例，以便帮助用户更好的理解和使用神通数据库 OLEDB/ADO。

〖手册构成〗

本手册主要由三部分组成：

第 1 章，“介绍”，简要介绍 OLEDB/ADO，包括安装，运行环境要求，基本连接属性等。

第 2 章，“OLEDB 使用说明”，具体介绍 OLEDB 的编程接口。

第 3 章，“神通数据库 ADO 应用参考”，具体介绍 ADO 的编程接口。

〖相关文档〗

使用本手册时可以参考神通数据库的手册集，手册集包含以下文档：

《神通数据库安装手册》

《神通数据库备份恢复工具使用手册》

《神通数据库 DBA 管理工具使用手册》

《神通数据库系统管理员手册》

《神通数据库嵌入式 SQL 语言手册》

《神通数据库交互式 SQL 查询工具使用手册》

《神通数据库 JDBC 开发指南》

《神通数据库过程语言手册》

《神通数据库 OLEDB/ADO 用户手册》

《神通数据库迁移工具使用手册》

《神通数据库 ODBC 程序员开发指南》

《神通数据库审计管理》

《神通数据库审计工具使用手册》

《神通数据库性能监测工具使用手册》

《神通数据库作业调度工具使用手册》

〖手册约定〗

本手册遵循以下约定：

所有标题均使用黑体字。

如果标题后跟有“【条件】”字样，说明该标题下正文所要求的内容只是在一定条件下必须的。

【注意】的意思是请读者注意那些需要注意的事项。

【警告】的意思是请读者千万注意某些事项，否则将造成严重错误。

【提示】的意思是提供给读者一些实用的操作技巧。

对于手册中出现的正文和程序代码，遵循如下约定：

表 0-1　手册正文约定

约定	含义	范例
粗体	表示强调	确保控制文件和数据文件不要驻留在同一个磁盘上。
大写等宽字母	表示由系统提供的元素，如参数、权限、数据类型、关键词、命令、函数名，以及由系统提供的列名、数据库对象和结构等。	利用 BACKUP 命令备份数据库。 在 USER_TABLES 数据字典视图中查询 TABLE_NAME 列
小写等宽字母	表示执行程序、文件名、目录名以及需要由使用者提供的元素，包括计算机名、数据库名、数据库对象和结构、列名、程序单元以及参数等。 【注意】：某些元素要求使用大写或者大小写混合形式。此时，应当根据实际的要求输入。	department_id，department_name 以及 location_id 列在 departments 表中。
小写等宽斜体	表示占位符或者变量	

表 0-2 手册中出现的程序代码的书写约定

约定	含义	范例
[]	表示包含一个或者多个可选项。不需要输入中括号本身。	DECIMAL (digits [, precision])
{ }	表示包含两个以上（含两个）的选项，其中有一个是必须的。不需要输入花括号本身。	{ENABLE | DISABLE}
|	分割中括号或者花括号中的两个或者两个以上的选项。不需要输入“|”本身。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
…	表示省略 表示重复	CREATE TABLE … AS subquery; SELECT col1, col2, … , coln FROM employees;
. . .	表示省略了若干行	
斜体	表示占位符或者需要提供特定值的变量	CONNECT SYSTEM/system_password DB_NAME = database_name
大写	表示系统提供的元素，主要是为了	SELECT last_name, employee_id FROM employees;

	与使用者定义的元素相互区分。除了出现在方括号中的元素外，应当按照顺序逐字输入。当然，有些元素在系统中是大小写不敏感的，因此使用者也可以根据系统说明以小写的方式输入。	SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
小写	表示由使用者提供的元素。例如：表、列和文件的名字。 【注意】：根据某些具体的要求，有些由使用者提供的元素可能要求使用大写或者大小写混合的形式。此时，应当根据实际的要求输入。	SELECT last_name, employee_id FROM employees; CREATE USER tom IDENTIFIED BY a8M9j7

目录

〖阅读对象〗 ii
〖内容简介〗 ii
〖手册构成〗 ii
〖相关文档〗 ii
〖手册约定〗 iii
第 1 章 介绍 1
1.1 神通数据库 OLEDB/ADO 简介 2
1.2 神通数据库 OLEDB/ADO 的安装 3
1.2.1 系统资源要求 3
1.2.2 神通数据库 OLEDB/ADO 文件 3
1.3 神通数据库 OLEDB/ADO 基本连接参数 4
第 2 章 OLEDB 使用说明 5
2.1 概述 6
2.2 神通数据库 OLE DB 应用参考 7
2.2.1 创建基本 OLE DB 应用 7
2.2.2 数据源对象(Data Source Object) 10
2.2.3 Command 对象 17
2.2.4 Rowset 对象 17
2.2.5 数据类型 26
2.2.6 事务 27
2.2.7 神通数据库 OLE DB 枚举器 27
2.3 OLE DB 编程参考 34
第 3 章 神通数据库 ADO 应用参考 35
3.1 概述 36
3.2 神通数据库 ADO 应用参考 37
3.2.1 Connection 对象 37
3.2.2 Command 对象 37
3.2.3 Recordset 对象 38
3.2.4 在多种语言中使用 ADO 39

第1章 介绍

本章内容包括：

1. 神通数据库 OLEDB/ADO 简介
2. 如何安装神通数据库 OLEDB/ADO？
3. 如何通过 OLEDB 来连接神通数据库？

1.1神通数据库 OLEDB/ADO 简介

随着网络技术和数据库技术的不断发展，现在的应用系统对数据集成的要求越来越高，这些数据有可能分布在不同的地方，并且使用不同的格式，例如关系型数据库和操作系统中的文件、电子表格、电子邮件、多媒体数据以及目录服务信息等等。传统的解决方案是使用大型的数据库系统，把所有这些数据都移到数据库系统中，然后按照操作数据库的办法对这些数据进行访问，这样做虽然能够按统一的方式对数据进行各种操作，但这种间接访问方式带来了很多问题，比如数据更新不及时、空间资源的冗余和访问效率低等等。

Microsoft 公司推出的一致数据访问技术则较好地解决了这些问题，它使得应用通过一致的接口来访问各种各样的数据，而不管数据驻留在何处，也不需要进行数据转移或复制、转换，在实现分布式的同时也带来了高效率。并且 UDA 技术在统一数据访问接口的同时，它的多层结构使数据使用方有了更多的选择机会，而它强大的扩展能力也给数据提供方留下了更多的扩展余地，这种开放型的软件结构使它具有极强的生命力，所以，这种技术从一推出便获得了广泛的欢迎，可以说，UDA 技术是继 ODBC 之后的又一数据访问技术的飞跃。

UDA 技术包括 OLE DB 和 ADO 两层标准接口，OLE DB 是系统级的编程接口，它定义了一组 COM 接口，这组接口封装了各种数据系统的访问操作，这组接口为数据使用方和数据提供方建立了标准，OLE DB 还提供了一组标准的服务组件，用于提供查询、缓存、数据更新、事务处理等操作，因此，数据提供方只需实现一些简单的数据操作，在使用方就可以获得全部的数据控制能力。

ADO 是应用层的编程接口，它通过 OLE DB 提供的 COM 接口访问数据，它适合于各种客户机/ 服务器应用系统和基于 Web 的应用，尤其在一些脚本语言中访问数据库操作是 ADO 的主要优势。ADO 是一套用自动化技术建立起来的对象层次结构，它比其他的一些对象模型如 DAO(Data Access Object)、RDO(Remote Data Object) 等具有更好的灵活性，使用更为方便，并且访问数据的效率更高。

神通数据库 OLEDB/ADO 是针对神通数据库开发的符合 Microsoft MDAC2.6 规范的 UDA 接口。

1.2神通数据库 OLEDB/ADO 的安装

1.2.1 系统资源要求

为了快捷的完成安装，需要对系统的软硬件资源进行检查，在安装之前必须确保系统资源满足如下要求：

- 硬件的要求

最低配置：

CPU 166MHz 以上　　内存 64 兆或更多

建议配置：

CPU 700MHz 以上　　内存 256 兆或更多

- 虚拟内存

至少需要 512MB（根据应用实际情况增加）。

- 操作系统的要求

Windows 2000 / WindowsXP

1.2.2 神通数据库 OLEDB/ADO 文件

神通数据库 OLEDB 包括两个文件：OSCARProv.dll， OSCAROdbc.dll。两个文件所在的目录必须放在同一个目录下，在确定两个文件都存在后，运行如下命令注册神通数据库 OLEDB 组件：

```
Regsvr32 OSCARProv.dll
```

Regsvr32 是操作系统附带的注册组件的命令程序。

如果用户使用神通数据库安装程序，则上述过程会自动进行，用户不需再做特别设置。安装好的 OLEDB 文件存放在安装目录里的 OLEDB 目录下。

ADO 是一组 windows 标准组件，由 Microsoft 公司提供，不需要用户特别安装。

1.3神通数据库 OLEDB/ADO 基本连接参数

如果通过 OLEDB 来连接神通数据库，用户必须提供 DBPROP_INIT_DATASOURCE，DBPROP_INIT_LOCATION，DBPROP_AUTH_USERID，DBPROP_AUTH_PASSWORD 四个基本属性，它们的具体含义说明如下：

DBPROP_INIT_DATASOURCE　　数据库名

DBPROP_INIT_LOCATION　　数据库所在位置(主机 IP)

DBPROP_AUTH_USERID　　用户名

DBPROP_AUTH_PASSWORD　　用户密码

例如，通过 Microsoft 公司提供的测试工具 LTM.exe 进行接口测试的话，初始化字串（initializing string）应填写为：

```
DATA SOURCE=OSRDB;LOCATION=192.9.200.99; USERID=SYSDBA; PASSWORD = szoscar55;
```

如果通过 ADO 来连接，则 Provider 名称为 OSCARProv，ADO 中 ConnectionString 的基本格式为：

```
provider= OSCARProv;DATA SOURCE=数据库名;LOCATION=服务器 IP;USERID=用户名;PASSWORD=密码;
```

举例如下：

```
Dim connection As New ADODB.Connection
connection.ConnectionString =
"provider=OSCARProv; DATA SOURCE=OSRDB; LOCATION=192.9.200.78;
USERID=SYSDBA;PASSWORD= szoscar55;"
```

如果用 Microsoft 的测试工具 tabledump.exe 来生成测试脚本的话，tabledump 的命令行参数应该如下：

```
tabledump provider=OSCARProv;tablename=test;providerstring="DATA SOURCE = OSRDB; LOCATION=192.9.200.99; USERID=SYSDBA; PASSWORD = szoscar55;"
```

神通数据库 OLEDB 还提供了 Binder 对象，通过它用户可以使用 URL 来访问指定的数据。神通数据库 ROOTBINDER 名为 OSCARBinder。Binder URL 的格式为:“OSCARBinder://数据库所在位置:账号名:密码/数据库名/表名/行号”;例如通过 Microsoft 公司提供的测试工具 LTM.exe，如果要测试 Binder 接口，则初始化字串（initializing string）应填写为：

```
DATA SOURCE=OSRDB;LOCATION=192.9.200.99; USERID=SYSDBA;PASSWORD= szoscar55;ROOTBINDER=OSCARBinder;
ROOT_URL=OSCARBinder://192.9.200.99:SYSDBA: szoscar55/OSRDB/ICOLUMNS/1;
```

其中 ICOLUMNS 为已经存在的表名。

第2章 OLEDB 使用说明

本章内容包括：

1. 概述
2. 神通数据库 OLE DB 应用参考
3. OLE DB 编程参考

2.1概述

神通数据库 OLE DB 通过神通数据库底层接口来访问数据，为用户提供一组高性能的访问神通数据库的 COM 接口。神通数据库 OLE DB 兼容 OLE DB 2.0 规范。

OLE DB 体系结构中包含二个角色：使用者，提供者。提供者就是 OLE DB，提供一组 COM 组件。所有的数据以虚拟表的形式来组织，对于神通数据库关系数据库而言，这也是一种自然组织方式。提供者内部使用神通数据库 CFCI 对底层数据库进行访问，保证了数据访问的高效。使用者使用基于 COM 的 OLE DB API 和提供者进行交互。

神通数据库 OLE DB 包含 7 个核心组件。这些组件由提供者实现，被使用者使用。这 7 个组件的说明如下。

Enumerator 对象被用来定位数据源（data source）。使用者可以用 Enumerator 来检索系统中可用的数据源。一旦通过 Enumerator 对象选择了一个数据源，data source 对象就会被创建。

Data source 对象用来连接底层的数据存储（OSCAR）。每个 data source 组件都有一个唯一的 CLSID，使用者可以使用这个 CLSID，通过 CoCreateInstance 函数直接创建一个 data source 对象实例；使用者也可以使用 Enumerator 对象获取 data source 对象。

通过 data source 对象可以创建 Session 对象。Session 对象代表对数据源的一个特定连接。Session 对象定义了一个事务操作的边界。它也可以创建 Command 和 Rowset 对象，使用者主要通过这两个对象来访问数据。一个 Data source 对象可以创建多个 Session 对象。

Command 对象负责控制和执行查询命令。一个 Session 对象可以创建多个 Command 对象。尽管在 OLE DB 规范中，Command 对象没有规定查询命令所使用的语言，但对于神通数据库 OLE DB 而言，查询语言就是 SQL 语言。

Rowset 对象可以直接通过 Session 对象得到，也可以通过一个执行查询命令的 Command 对象得到。Rowset 用表格的形式存储数据。使用者可以通过 Rowset 读取或者更新数据。

Error 对象能够被任何其他的 OLE DB 对象生成。相对于 COM 方法的返回值 HRESULT，Error 对象包含更丰富的错误信息。

2.2神通数据库 OLE DB 应用参考

本节将详细讲述如何通过神通数据库 OLE DB 来访问神通数据库。

2.2.1 创建基本 OLE DB 应用

OLEDB 对象通过属性来设置各个对象的具体行为。另外，对 OLEDB 应用的编译应遵循一定的规范，一般说来，创建一个基本 OLE DB 应用包括如下步骤：

1. 建立对指定数据源的连接
2. 执行命令
3. 处理结果集

2.2.1.1 建立连接

使用神通数据库 OLE DB 时，用户（consumer）必须调用 CoCreateInstance 创建一个数据源对象。每个 OLE DB 都有一个唯一的类标示符。对于神通数据库 OLE DB，类标示符是 CLSID_OSCAROLEDB = {}。

数据源对象提供 IDBProperties 接口，通过它用户可以设置一些基本的认证信息，比如数据库服务器地址，数据库名，用户名和密码等。数据源对象还提供 IDBInitialize 接口。属性已经设置后，通过调用 IDBinitialize::Initialize 方法将建立与指定数据库的一个连接。下面以一个代码片断作为例子。

```
void InitializeAndEstablishConnection()
{
    //Initialize the COM library.
    CoInitialize(NULL);
    //Obtain access to the SHENTONG OLE DB provider.
    hr = CoCreateInstance(CLSID_OSCAROLEDB,
                          NULL,
                          CLSCTX_INPROC_SERVER,
                          IID_IDBInitialize,
                          (void **) &pIDBInitialize);

  /*
    Initialize the property values needed
    to establish the connection.
    */
    for(i = 0; i < 4; i++)
        VariantInit(&InitProperties[i].vValue);

    //Server name.
    InitProperties[0].dwPropertyID    = DBPROP_INIT_LOCATION;
    InitProperties[0].vValue.vt       = VT_BSTR;
    InitProperties[0].vValue.bstrVal=
                            SysAllocString(L"Server IP");
    InitProperties[0].dwOptions       = DBPROPOPTIONS_REQUIRED;
```

```
	InitProperties[0].colid                  = DB_NULLID;

	//Database.
	InitProperties[1].dwPropertyID    = DBPROP_INIT_ DATASOURCE;
	InitProperties[1].vValue.vt          = VT_BSTR;
	InitProperties[1].vValue.bstrVal= SysAllocString(L"database");
	InitProperties[1].dwOptions          = DBPROPOPTIONS_REQUIRED;
	InitProperties[1].colid                  = DB_NULLID;

	//Username (login).
	InitProperties[2].dwPropertyID    = DBPROP_AUTH_USERID;
	InitProperties[2].vValue.vt          = VT_BSTR;
	InitProperties[2].vValue.bstrVal= SysAllocString(L"system ");
	InitProperties[2].dwOptions          = DBPROPOPTIONS_REQUIRED;
	InitProperties[2].colid                  = DB_NULLID;

	//Password.
	InitProperties[3].dwPropertyID    = DBPROP_AUTH_PASSWORD;
	InitProperties[3].vValue.vt          = VT_BSTR;
	InitProperties[3].vValue.bstrVal= SysAllocString(L" szoscar55");
	InitProperties[3].dwOptions          = DBPROPOPTIONS_REQUIRED;
	InitProperties[3].colid                  = DB_NULLID;

	/*
	Construct the DBPROPSET structure(rgInitPropSet). The
	DBPROPSET structure is used to pass an array of DBPROP
	structures (InitProperties) to the SetProperties method.
	*/
	rgInitPropSet[0].guidPropertySet = DBPROPSET_DBINIT;
	rgInitPropSet[0].cProperties        = 4;
	rgInitPropSet[0].rgProperties       = InitProperties;

//Set initialization properties.
	hr = pIDBInitialize->QueryInterface(IID_IDBProperties,
	                                        (void **)&pIDBProperties);
	hr = pIDBProperties->SetProperties(1, rgInitPropSet);
	pIDBProperties->Release();
	//Now establish the connection to the data source.
	pIDBInitialize->Initialize()
}
```

如果用户在 ADO 中连接神通数据库，连接字串的形式为：

```
DATA SOURCE = dbname; LOCATION=serverlocation; USERID=id; PASSWORD = xxx;
```

例如：

```
"DATA SOURCE=OSRDB; LOCATION=192.9.200.100;USERID=SYSDBA; PASSWORD
```

```
= szoscar55;"
```

2.2.1.2 执行一条命令

连接建立后，用户调用数据源对象的 IDBCreateSession::CreateSession 方法创建一个 Session 对象。Session 对象是 Command，Rowset 对象的创建者。

用户可以通过 IOpenRowset::OpenRowset 获得一个包括指定表的所有行的 Rowset 对象。用户也可以执行 SQL 语句(比如 select * from test)。用户通过 Session 对象的 IDBCreateCommand::CreateCommand 方法创建一个 Command 对象，然后用户调用 Command 对象的 ICommandText::SetCommandText 来设定需要执行的 SQL 语句，最后 Execute 方法被用来执行所设定的语句。任何 SQL 语句都可以被执行。但不是所有的语句都会产生 Rowset 对象。

2.2.1.3 处理结果集

使用 Openrowset 和 Execute 方法得到 Rowset 对象是相同的，用户都可以检索和访问其中的数据。Rowset 对象以一种表格的形式提供数据，一个 Rowset 对象包含一个行的集合，每行又可分为一个或多个列。Rowset 对象具有一组功能接口，比如 IRowset（包含顺序从 rowset 中获取行的方法），IAccessor（包含设置数据邦定的方法），IColumnInfo（获取 rowset 中行的列信息）和 IRowsetInfo(提供 Rowset 的基本信息)。

用户调用 IRowset::GetData 获取 Rowset 的行数据。在调用这个方法之前，用户必须提供一组 DBBINDING 结构的缓存，每个 DBBINDING 描述了 OLE DB 应该将哪些数据以及如何将数据存放在用户的缓存中。用户调用 IAccessor::CreateAccessor 来设置构造好的 DBBINDING 结构数组，这个方法将创建一个 Accessor，这个 Accessor 标示了刚才设置的 DBBDINGDING 结构数据，用户可以使用 Accessor 来获取或者设定数据。

2.2.1.4 编译 OLE DB 应用

OLE DB 应用必须包含 Oledb.h（OLE DB 接口定义文件）和 Oledberr.h (这个文件中定义的错误常量通常会被使用) files，同时必须和 Oledb.lib 连接。OLE DB 应用必须使用 UNICODE 字符集。

2.2.1.5 OLE DB 的属性

用户通过设置和获取属性来了解和设置 OLE DB 的行为。例如，用户通过设置属性来限制 Rowset 对象所能提供的接口；用户通过获取属性了解一个对象（比如，Rowset，Session 或 data source）所支持的功能。

每个属性包含值，类型，描述和读/写信息，对于 Rowset 属性，还有一个标示该属性是否能应用于列的标志。一个属性由一个 GUID 和一个属性 ID(整数)来唯一的标示。属性集(property set)是共享一个 GUID 的属性的集合。另外，每个属性又属于一个或者多个属性组（property group）。属性组是由应用于同一个对象的所有属性组成的。设定属性值包括如下步骤：

1. 明确需要设定的属性
2. 明确所设属性从属的属性集
3. 分配 DBPROPSET 接口数组，数组的一个元素代表一个属性集
4. 为每一个属性集分配一个 DBPROP 结构数组

5. 为每一个属性填写 DBPROP 结构
6. 为每个属性集填写 DBPROPSET 结构
7. 调用接口设定属性，以 DBPROPSET 结构数组和它的元素个数为参数

2.2.2 数据源对象(Data Source Object)

数据源对象用来建立和数据库服务的一个连接。创建一个数据源对象是一个 OLE DB 用户首先做的步骤。成功创建数据源对象并初始化后，用户可以创建 Session 对象。OLE DB Session 对象提供了一组可以进行数据访问和操纵的接口，Session 对象代表了对数据库的一组操作，数据库的事务就作用于 Session 对象上。

2.2.2.1 数据源对象

神通数据库 OLE DB 定义了一个类标示符(CLSID)，用 C/C++表达为：GUID CLSID_OSCAROLEDB = {}。使用这个 CLSID，用户调用 CoCreateInstance 创建一个数据源对象。

神通数据库 OLE DB 是一个进程内服务(IN-PROCESS SERVER)， 神通数据库 OLE DB 的对象必须使用 CLSCTX_INPROC_SERVER 参数来标示它的执行环境。

神通数据库 OLE DB 数据源对象具有 IDBInitialize 接口，通过这个接口的函数，用户能够连接上指定神通数据库服务器。

下面的例子通过 CLSID_OSCAROLEDB 创建数据源对象，然后获取 IDBInitialize 接口执行连接。

```
IDBInitialize*    pIDBInitialize;
HRESULT                hr;

hr = CoCreateInstance(CLSID_OSCAROLEDB, NULL, CLSCTX_INPROC_SERVER,
        IID_IDBInitialize, (void**) &pIDBInitialize);
…
…
if (SUCCEEDED(hr))
{
        //   Perform necessary processing with the interface.
        pIDBInitialize->Initialize();
        pIDBInitialize->Release();
}
else
{
        // Display error from CoCreateInstance.
}
```

2.2.2.2 Session 对象

神通数据库 OLE DB 的 Session 对象标示对数据库的一个单独连接，数据库的事务操作就作用在这个对象上。所有的 Command 对象都由 Session 对象产生。Session 对象由数据源对象的 IDBCreateSession：：CreateSession 方法创建。只要 Session 对象没有被释放，它就一

直保持对数据的一个连接。

下面的例子展示神通数据库 OLE DB 如何连接数据库。

```
int main()
{
    // Interfaces used in the example.
    IDBInitialize*          pIDBInitialize         = NULL;
    IDBCreateSession*       pIDBCreateSession      = NULL;
    IDBCreateCommand*       pICreateCmd1           = NULL;
    IDBCreateCommand*       pICreateCmd2           = NULL;
    IDBCreateCommand*       pICreateCmd3           = NULL;

    // Initialize COM.
    if (FAILED(CoInitialize(NULL)))
    {
        // Display error from CoInitialize.
        return (-1);
    }

    // Get the memory allocator for this task.
    if (FAILED(CoGetMalloc(MEMCTX_TASK, &g_pIMalloc)))
    {
        // Display error from CoGetMalloc.
        goto EXIT;
    }

    // Create an instance of the data source object.
    if (FAILED(CoCreateInstance(CLSID_OSCAROLEDB, NULL,
        CLSCTX_INPROC_SERVER, IID_IDBInitialize, (void**)
        &pIDBInitialize)))
    {
        // Display error from CoCreateInstance.
        goto EXIT;
    }

    // The InitFromPersistedDS function
    // performs IDBInitialize->Initialize() establishing
    // the first application connection to the instance of SHENTONG Server.
    if (FAILED(InitFromPersistedDS(pIDBInitialize, L"MyDataSource",
        NULL, NULL)))
    {
        goto EXIT;
    }

    // The IDBCreateSession interface is implemented on the data source
```

```
    // object. Maintaining the reference received maintains the
    // connection of the data source to the instance of SHENTONG Server.
    if (FAILED(pIDBInitialize->QueryInterface(IID_IDBCreateSession,
        (void**) &pIDBCreateSession)))
    {
        // Display error from pIDBInitialize.
        goto EXIT;
    }

    // Releasing this has no effect on the SHENTONG Server connection
    // of the data source object because of the reference maintained by
    // pIDBCreateSession.
    pIDBInitialize->Release();
    pIDBInitialize = NULL;

    // The session created next receives the SHENTONG Server connection of
    // the data source object. No new connection is established.
    if (FAILED(pIDBCreateSession->CreateSession(NULL,
        IID_IDBCreateCommand, (IUnknown**) &pICreateCmd1)))
    {
        // Display error from pIDBCreateSession.
        goto EXIT;
    }

    // A new connection to the instance of SHENTONG Server is
established to support the
    // next session object created. On successful completion, the
    // application has two active connections on the SHENTONG Server.
    if (FAILED(pIDBCreateSession->CreateSession(NULL,
        IID_IDBCreateCommand, (IUnknown**) &pICreateCmd2)))
    {
        // Display error from pIDBCreateSession.
        goto EXIT;
    }

    // pICreateCmd1 has the data source connection. Because the
    // reference on the IDBCreateSession interface of the data source
    // has not been released, releasing the reference on the session
    // object does not terminate a connection to the instance of
SHENTONG Server.
    // However, the connection of the data source object is now
    // available to another session object. After a successful call to
    // Release, the application still has two active connections to the
    // instance of SHENTONG Server.
```

```
        pICreateCmd1->Release();
        pICreateCmd1 = NULL;

        // The next session created gets the SHENTONG Server connection
        // of the data source object. The application has two active
        // connections to the instance of SHENTONG Server.
        if (FAILED(pIDBCreateSession->CreateSession(NULL,
                IID_IDBCreateCommand, (IUnknown**) &pICreateCmd3)))
        {
                // Display error from pIDBCreateSession.
                goto EXIT;
        }

EXIT:
        // Even on error, this does not terminate a SHENTONG Server connection
        // because pICreateCmd1 has the connection of the data source
        // object.
        if (pICreateCmd1 != NULL)
                pICreateCmd1->Release();

        // Releasing the reference on pICreateCmd2 terminates the SHENTONG
        // Server connection supporting the session object. The application
        // now has only a single active connection on the instance of
SHENTONG Server.
        if (pICreateCmd2 != NULL)
                pICreateCmd2->Release();

        // Even on error, this does not terminate a SHENTONG Server connection
        // because pICreateCmd3 has the connection of the
        // data source object.
        if (pICreateCmd3 != NULL)
                pICreateCmd3->Release();

        // On release of the last reference on a data source interface,
the connection
        // of the data source object to the instance of SHENTONG Server
  is broken.
        // The example application now has no SHENTONG Server
  connections active.
        if (pIDBCreateSession != NULL)
                pIDBCreateSession->Release();

        // Called only if an error occurred while attempting to get a
        // reference on the IDBCreateSession interface of the data source.
```

```
        // If so, the call to IDBInitialize::Uninitialize terminates the
        // connection of the data source object to the instance of
SHENTONG Server.
        if (pIDBInitialize != NULL)
        {
            if (FAILED(pIDBInitialize->Uninitialize()))
            {
                // Uninitialize is not required, but it fails if an
                // interface has not been released. Use it for
                // debugging.
            }
            pIDBInitialize->Release();
        }

        if (g_pIMalloc != NULL)
            g_pIMalloc->Release();
        CoUninitialize();
        return (0);
}
```

如果一个应用持续的创建和释放 Session 对象，将对应用性能带来较大影响。应用可以通过更有效的管理和使用 Session 对象来降低这种影响。通过保留 Session 对象的某个接口来保持对该 Session 对象的引用，应用可以保持对数据库的连接。

2.2.2.3 持久数据源对象

神通数据库 OLE DB 数据源对象支持 IPersistFile 接口，通过这个接口，数据源信息可以写入磁盘。下面这个例子展示了一个持久化数据源初始化信息的函数，初始化信息包括服务器地址，数据库，用户名和密码。

```
HRESULT SetAndSaveInitProps
         (
        IDBInitialize* pIDBInitialize,
        WCHAR* pDataSource,
        WCHAR* pCatalog,
        WCHAR* pUID,
        WCHAR* pPWD
        )
{
        const ULONG         nProps = 5;
        ULONG               nSSProps;
        ULONG               nPropSets;
        ULONG               nProp;
        IDBProperties*      pIDBProperties        = NULL;
        IPersistFile*       pIPersistFile         = NULL;
        DBPROP              aInitProps[nProps];
        DBPROP*             aSSInitProps          = NULL;
```

```
    DBPROPSET*          aInitPropSets           = NULL;
    HRESULT             hr;

    nSSProps = 0;
    nPropSets = 1;

    aInitPropSets = new DBPROPSET[nPropSets];

    // Initialize common property options.
    for (nProp = 0; nProp < nProps; nProp++)
    {
        VariantInit(&aInitProps[nProp].vValue);
        aInitProps[nProp].dwOptions =
DBPROPOPTIONS_REQUIRED;
        aInitProps[nProp].colid = DB_NULLID;
    }

    // Level of prompting that will be done to complete the connection
    // process.
    aInitProps[0].dwPropertyID = DBPROP_INIT_PROMPT;
    aInitProps[0].vValue.vt = VT_I2;
    aInitProps[0].vValue.iVal = DBPROMPT_NOPROMPT;

    // Server name.
    aInitProps[1].dwPropertyID = DBPROP_INIT_DATASOURCE;
    aInitProps[1].vValue.vt = VT_BSTR;
    aInitProps[1].vValue.bstrVal = SysAllocString(pDataSource);

    // Database.
    aInitProps[2].dwPropertyID = DBPROP_INIT_CATALOG;
    aInitProps[2].vValue.vt = VT_BSTR;
    aInitProps[2].vValue.bstrVal = SysAllocString(pCatalog);

    // User ID.
    aInitProps[3].dwPropertyID = DBPROP_AUTH_USERID;
    aInitProps[3].vValue.vt = VT_BSTR;
    aInitProps[3].vValue.bstrVal = SysAllocString(pUID);

    // Password.
    aInitProps[4].dwPropertyID = DBPROP_INIT_PASSWORD;
    aInitProps[4].vValue.vt = VT_BSTR;
    aInitProps[4].vValue.bstrVal = SysAllocString(pPWD);

    // Now that properties are set, construct the PropertySet array.
```

```
    aInitPropSets[0].guidPropertySet = DBPROPSET_DBINIT;
    aInitPropSets[0].cProperties = nProps;
    aInitPropSets[0].rgProperties = aInitProps;

    // Set initialization properties
    pIDBInitialize->QueryInterface(IID_IDBProperties,
        (void**) &pIDBProperties);
    hr = pIDBProperties->SetProperties(nPropSets, aInitPropSets);
    if (FAILED(hr))
    {
        // Display error from failed SetProperties.
    }
    pIDBProperties->Release();

    // Free references on OLE known strings.
    for (nProp = 0; nProp < nProps; nProp++)
    {
        if (aInitProps[nProp].vValue.vt == VT_BSTR)
            SysFreeString(aInitProps[nProp].vValue.bstrVal);
    }

    for (nProp = 0; nProp < nSSProps; nProp++)
    {
        if (aSSInitProps[nProp].vValue.vt == VT_BSTR)
            SysFreeString(aInitProps[nProp].vValue.bstrVal);
    }

    // Free dynamically allocated memory.
    delete [] aInitPropSets;
    delete [] aSSInitProps;

    // On success, persist the data source.
    if (SUCCEEDED(hr))
    {
        pIDBInitialize->QueryInterface(IID_IPersistFile,
            (void**) &pIPersistFile);

        hr = pIPersistFile->Save(OLESTR("MyDataSource.dat"),
FALSE);

        if (FAILED(hr))
        {
            // Display errors from IPersistFile interface.
        }
```

```
            pIPersistFile->Release();
        }

        return (hr);
    }
```

IPersistFile::Save 方法可以在 IDBInitialize::Initialize 方法之前或者之后调用。在 Initialize 方法之后调用可以保证持久化的信息是正确的能够成功初始化的信息。

2.2.3 Command 对象

神通数据库 OLE DB 提供 Command 对象，用户可以使用该对象执行 SQL 语句。

2.2.3.1 命令语法

神通数据库 OLE DB 可以接受符合 ODBC SQL 和 SQL-92 规范的 SQL 语句。

2.2.3.2 命令参数

命令的参数由问号字符来标识。例如，下面的 SQL 语句具有一个输入参数：

```
select * from test where testid = ?
```

神通数据库 OLE DB 并不会自动检查由 ICommandWithParameters::SetParameterInfo 方法设定的参数的正确性。在执行的时候可能会引起错误或者导致数据精度丢失等问题。为了避免这个问题，应用程序应该做到：在调用 ICommandWithParameters::SetParameterInfo 的时候，保证 pwszDataSourceType 与数据库中的相应字段类型相同；在使用 Accessor 的时候，保证绑定参数的 DBTYPE 值与数据库中相应字段的类型相同；或者，应用程序调用 ICommandWithParameters::GetParameterInfo 来手动获得参数的数据库类型。

2.2.3.3 产生多结果集的命令

当把一批 SQL 语句当作一条命令执行或者执行一个包含一批 SQL 语句的存储过程时，神通数据库 OLE DB 可能返回一个以上的 Rowset 对象。用户可以使用 IMultipleResults 接口来处理生成的多个结果集。

2.2.4 Rowset 对象

Rowset 对象是一组数据行的集合，其中每一行都包含一个或者多个列信息。Rowset 对象是所有 OLE DB 用来提供表格状数据访问的重要对象。

在用户通过 IDBCreateSession::CreateSession 创建 Session 对象后，用户可以通过 IOpenRowset 或者 IDBCreateCommand 接口来创建 Rowset 对象。神通数据库 OLE DB 对于两种接口都提供支持。

通过 IOpenRowset::OpenRowset 创建 Rowset 对象

这其实是在指定表上创建了一个 Rowset 对象，它包含了指定表的所有行。表由 OpenRowset 的一个参数来指定。

通过 IDBCreateCommand::CreateCommand 创建一个 Command 对象

Command 对象可以执行指定的 SQL 语句。对于神通数据库 OLE DB，用户可以指定任

何符合 SQL-92 标准的 SQL 语句。用 Command 创建 Rowset 对象的步骤如下：

用户调用 Session 对象的 IDBCreateCommand::CreateCommand 方法创建 Command 对象，然后调用这个 Command 对象的 ICommandText::SetCommandText 方法设置所要执行的 SQL 语句（比如一条 select 语句）。

用户调用 ICommand::Execute 来执行刚才设置的语句。如果这条语句具有结果集，那一个 Rowset 对象就会被产生，通过参数返回给用户。

2.2.4.1 用 OpenRowset 创建 Rowset 对象

神通数据库 OLE DB 支持 OpenRowset 方法，如果参数 pTableID 不为 NULL，则必须满足：

DBID eKind 成员的值必须 DBKIND_NAME

DBID uName 成员的值必须是一个内容为已存在的表或者是图的名字的 Unicode 字符串。

2.2.4.2 获取行数据

IRowset 接口是 Rowset 对象的基本接口。这个接口提供了顺序获取和管理行数据的方法。用户使用 IRowset 的方法进行基本的行操作，比如获取和释放行数据，获取列值等等。

当用户创建了 Rowset 对象，通常第一步是通过 IRowsetInfo：：GetProperties 读取 Rowset 的属性，了解 Rowset 的行为和能力。然后用户可以通过 IColumnsInfo 或者 IColumnsRowset 接口获得 Rowset 的列信息，具体内容为：

Rowset 包含的列数

一个 DBCOLUMNINFO 结构数组，每个元素表示一列

一个包含所有字符串的缓存的指针

收集到 Rowset 的列信息后，用户可以根据这些信息和应用的需求，构建一个绑定数组作为参数传递给 IAccessor：：CreateAccessor，这样可以创建一个 Accessor，它代表了用户创建的列信息绑定。用户可以创建多个 Accessor 来获取不同的列数据。只要 Rowset 对象还存在，Accessor 能够随时创建和释放。

用户通过 IRowset::GetNextRows 或者 IRowsetLocate：：GetRowsAt 来获取 Rowset 中的行数据。这些函数把行数据从数据库读取出来存入 OLE DB 的行缓存中。用户不能直接访问 OLE DB 的行缓存。用户必须使用 IRowset：：GetData 将 OLE DB 的行缓存复制到用户的缓存中，同样，用户使用 IRowsetChange：：SetData 来将用户缓存中的要改变的行数据复制到 OLE DB 的行缓存中。

用户调用 GetData，传入一个标识行的句柄，一个标识 Accessor 的句柄，一个用户分配的缓存指针。GetData 根据 Accessor 所规定的列数据绑定，将行标识句柄标识的行的数据写入用户的缓存。用户可以在同一行上使用不同的 Accessor 多次调用 GetData，也就是说，用户可以获取同一份行数据的不同拷贝。

当用户获取或者更新了行数据以后，它调用 ReleaseRows 来释放 OLE DB 的行缓存。当用户对 Rowset 的对象操作完成后，可调用 IAccessor：：ReleaseAccessor 来释放 acceoosr，然后调用 IUnknown：：Release 方法来释放 Rowset 对象本身。当 Rowset 对象被释放后，所有的行句柄信息和 Accessor 都将被释放。

GetNextRows 方法会顺序的读取 Rowset 中的所有行数据，不过 GetNextRows 可以使用 skip 参数来忽略掉某些行。另外，FindNextRow，Seek，RestartPosition 都可以改变 GetNextRows 的顺序读取方式。

下面举例说明如何从 Rowset 中提取行数据。

```
/*
    Example shows how to fetch rows from a result set.
*/
void InitializeAndEstablishConnection();
void ProcessResultSet();

#define UNICODE
#define _UNICODE
#define DBINITCONSTANTS
#define INITGUID

#include <stdio.h>
#include <tchar.h>
#include <stddef.h>
#include <windows.h>
#include <iostream.h>
#include <oledb.h>
#include <SQLOLEDB.h>

IDBInitialize*          pIDBInitialize          = NULL;
IDBProperties*          pIDBProperties          = NULL;
IDBCreateSession*       pIDBCreateSession       = NULL;
IDBCreateCommand*       pIDBCreateCommand       = NULL;
ICommandText*           pICommandText           = NULL;
IRowset*                pIRowset                = NULL;
IColumnsInfo*           pIColumnsInfo           = NULL;
DBCOLUMNINFO*           pDBColumnInfo           = NULL;
IAccessor*              pIAccessor              =   NULL;
DBPROP                  InitProperties[4];
DBPROPSET               rgInitPropSet[1];
ULONG                   i, j;
HRESULT                 hr;
LONG                    cNumRows = 0;
ULONG                   lNumCols;
WCHAR*                  pStringsBuffer;
DBBINDING*              pBindings;
ULONG                   ConsumerBufColOffset = 0;
HACCESSOR               hAccessor;
ULONG                   lNumRowsRetrieved;
HROW                    hRows[10];
```

```
HROW*                    pRows = &hRows[0];
BYTE*                    pBuffer;

void main()
{
    //Here is the command to execute.
    WCHAR* wCmdString
        = OLESTR(" SELECT title, price FROM titles WHERE
royalty > 14");
    // Call a function to initialize and establish connection.
    InitializeAndEstablishConnection();

    //Create a session object.
    if(FAILED(pIDBInitialize->QueryInterface(
                                IID_IDBCreateSession,
                                (void**) &pIDBCreateSession)))
    {
        cout << "Failed to obtain IDBCreateSession interface.\n";
    }

    if(FAILED(pIDBCreateSession->CreateSession(
                                    NULL,
                                    IID_IDBCreateCommand,
                                    (IUnknown**)
&pIDBCreateCommand)))
    {
        cout << "pIDBCreateSession->CreateSession failed.\n";
    }

    //Access the ICommandText interface.
    if(FAILED(pIDBCreateCommand->CreateCommand(
                                    NULL,
                                    IID_ICommandText,
                                    (IUnknown**)
&pICommandText)))
    {
        cout << "Failed to access ICommand interface.\n";
    }

    //Use SetCommandText() to specify the command text.
    if(FAILED(pICommandText->SetCommandText(DBGUID_DBSQL,
wCmdString)))
    {
        cout << "Failed to set command text.\n";
```

```
	}

	//Execute the command.
	if(FAILED(hr = pICommandText->Execute(NULL,
							IID_IRowset,
							NULL,
							&cNumRows,
							(IUnknown **) &pIRowset)))
	{
		cout << "Failed to execute command.\n";
	}

	//Process the result set.
	ProcessResultSet();

	pIRowset->Release();

	//Free up memory.
	pICommandText->Release();
	pIDBCreateCommand->Release();
	pIDBCreateSession->Release();

	if(FAILED(pIDBInitialize->Uninitialize()))
	{
		/*Uninitialize is not required, but it fails if an interface
		has not been released.    This can be used for debugging.
		cout << "Problem uninitializing.\n"; */
	} //endif.
	pIDBInitialize->Release();

	//Release the COM library.
	CoUninitialize();
};
//------------------------------------------------------------------
void InitializeAndEstablishConnection()
{
	//Initialize the COM library.
	CoInitialize(NULL);

	//Obtain access to the SHENTONGOLEDB provider.
	hr = CoCreateInstance(CLSID_OSCAROLEDB,
					NULL,
					CLSCTX_INPROC_SERVER,
					IID_IDBInitialize,
```

```
                                    (void **) &pIDBInitialize);
    if(FAILED(hr))
    {
        printf("Failed to get IDBInitialize interface.\n");
    } //end if

    /*
    Initialize the property values needed
    to establish the connection.
    */
    for(i = 0; i < 4; i++)
        VariantInit(&InitProperties[i].vValue);


    //Server name.
    InitProperties[0].dwPropertyID    = DBPROP_INIT_DATASOURCE;
    InitProperties[0].vValue.vt       = VT_BSTR;
    InitProperties[0].vValue.bstrVal=
                                    SysAllocString(L"testdb");
    InitProperties[0].dwOptions       = DBPROPOPTIONS_REQUIRED;
    InitProperties[0].colid           = DB_NULLID;

//Database.
    InitProperties[1].dwPropertyID    = DBPROP_INIT_LOCATION;
    InitProperties[1].vValue.vt       = VT_BSTR;
    InitProperties[1].vValue.bstrVal= SysAllocString(L"192.9.200.10");
    InitProperties[1].dwOptions       = DBPROPOPTIONS_REQUIRED;
    InitProperties[1].colid           = DB_NULLID;

//Username (login).
    InitProperties[2].dwPropertyID    = DBPROP_AUTH_USERID;
    InitProperties[2].vValue.vt       = VT_BSTR;
    InitProperties[2].vValue.bstrVal= SysAllocString(L"login");
    InitProperties[2].dwOptions       = DBPROPOPTIONS_REQUIRED;
    InitProperties[2].colid           = DB_NULLID;

//Password.
    InitProperties[3].dwPropertyID    = DBPROP_AUTH_PASSWORD;
    InitProperties[3].vValue.vt       = VT_BSTR;
    InitProperties[3].vValue.bstrVal= SysAllocString(L"Password");
    InitProperties[3].dwOptions       = DBPROPOPTIONS_REQUIRED;
    InitProperties[3].colid           = DB_NULLID;

    /*
```

```
        Now that the properties are set, construct the DBPROPSET structure
        (rgInitPropSet). The DBPROPSET structure is used to pass an array
        of DBPROP structures (InitProperties) to the SetProperties method.
        */
        rgInitPropSet[0].guidPropertySet = DBPROPSET_DBINIT;
        rgInitPropSet[0].cProperties     = 4;
        rgInitPropSet[0].rgProperties    = InitProperties;

        //Set initialization properties.
        hr = pIDBInitialize->QueryInterface(IID_IDBProperties,
                                            (void **)&pIDBProperties);
        if(FAILED(hr))
        {
            cout << "Failed to get IDBProperties interface.\n";
        }

        hr = pIDBProperties->SetProperties(1, rgInitPropSet);
        if(FAILED(hr))
        {
            cout << "Failed to set initialization properties.\n";
        } //end if

        pIDBProperties->Release();

        //Now establish the connection to the data source.
        if(FAILED(pIDBInitialize->Initialize()))
        {
            cout << "Problem in establishing connection to the data
            source.\n";
        }
} //end of InitializeAndEstablishConnection.
//-------------------------------------------------------------------
//Retrieve and display data resulting from a query.
void ProcessResultSet()
{
        //Obtain access to the IColumnInfo interface, from the Rowset
        object.
        hr = pIRowset->QueryInterface(IID_IColumnsInfo,
                                      (void **)&pIColumnsInfo);
        if(FAILED(hr))
        {
            cout << "Failed to get IColumnsInfo interface.\n";
        } //end if
```

```
        //Retrieve the column information.
        pIColumnsInfo->GetColumnInfo(&lNumCols,
                                     &pDBColumnInfo,
                                     &pStringsBuffer);

        //Free the column information interface.
        pIColumnsInfo->Release();

        //Create a DBBINDING array.
        pBindings = new DBBINDING[lNumCols];

        //Using the ColumnInfo structure, fill out the pBindings array.
        for(j=0; j<lNumCols; j++) {
                pBindings[j].iOrdinal = j+1;
                pBindings[j].obValue = ConsumerBufColOffset;
                pBindings[j].pTypeInfo = NULL;
                pBindings[j].pObject = NULL;
                pBindings[j].pBindExt = NULL;
                pBindings[j].dwPart = DBPART_VALUE;
                pBindings[j].dwMemOwner =
DBMEMOWNER_CLIENTOWNED;
                pBindings[j].eParamIO = DBPARAMIO_NOTPARAM;
                pBindings[j].cbMaxLen = pDBColumnInfo[j].ulColumnSize;
                pBindings[j].dwFlags = 0;
                pBindings[j].wType = pDBColumnInfo[j].wType;
                pBindings[j].bPrecision = pDBColumnInfo[j].bPrecision;
                pBindings[j].bScale = pDBColumnInfo[j].bScale;

                //Compute the next buffer offset.
                ConsumerBufColOffset = ConsumerBufColOffset +
                                     pDBColumnInfo[j].ulColumnSize;
        };
        //Get the IAccessor interface.
        hr = pIRowset->QueryInterface(IID_IAccessor, (void **)
&pIAccessor);
        if(FAILED(hr))
        {
                cout << "Failed to obtain IAccessor interface.\n";
        }
//Create an accessor from the set of bindings (pBindings).
        pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,
                                   lNumCols,
                                   pBindings,
                                   0,
```

```
                                        &hAccessor,
                                        NULL);

    //Print column names.
    for(j=0; j<lNumCols; j++) {
        printf("%-40S", pDBColumnInfo[j].pwszName);
    };
    //Get a set of 10 rows.
    pIRowset->GetNextRows(
                                NULL,
                                0,
                                10,
                                &lNumRowsRetrieved,
                                &pRows);

    //Allocate space for the row buffer.
    pBuffer = new BYTE[ConsumerBufColOffset];

    //Display the rows.
    while(lNumRowsRetrieved > 0) {
        //For each row, print the column data.
        for(j=0; j<lNumRowsRetrieved; j++) {
            //Clear the buffer.
            memset(pBuffer, 0, ConsumerBufColOffset);
            //Get the row data values.
            pIRowset->GetData(hRows[j], hAccessor, pBuffer);
            //Print title and price values.
            printf("%-40s%f\n", &pBuffer[pBindings[0].obValue],
              (FLOAT) Buffer[pBindings[0].obValue]);

        }; //for.

        //Release the rows retrieved.
        pIRowset->ReleaseRows(lNumRowsRetrieved,
                                    hRows,
                                    NULL,
                                    NULL,
                                    NULL);
        //Get the next set of 10 rows.
        pIRowset->GetNextRows(NULL,
                                    0,
                                    10,
                                    &lNumRowsRetrieved,
                                    &pRows);
```

```
        };    //while lNumRowsRetrieved > 0.

        //Free up all allocated memory.
        delete [] pBuffer;
        pIAccessor->ReleaseAccessor(hAccessor, NULL);
        pIAccessor->Release();
        delete [] pBindings;
    } //ProcessResultSet.
```

2.2.4.3 Row 对象

神通数据库 OLE DB 支持 Row 对象。Row 对象允许用户直接访问行数据。如果用户事先知道所执行的查询只会产生包含一行的结果集，那它可以使用 Row 对象来检索行数据。但如果查询结果有多行，仍然使用 Row 对象的话，Row 对象只能取到数据集中的第一条。用户使用 IRow：：GetColumns 获取数据。

2.2.4.4 更新 Rowset 中的数据

神通数据库 OLE DB 支持 IRowsetChange 和 IRowsetUpdate 接口，通过这两个接口，用户可以更新 Rowset 中的行数据并写入数据库。神通数据库 OLE DB 也支持 IRowsetResynch 和 IRowsetUpdate 接口，用户可以使用这两个接口从服务器读取刚更新过的数据。

对于用户对数据的更新，神通数据库 OLE DB 支持两种更新模式：立即更新和延迟更新。对于立即更新模式， IRowsetChange：：SetData 的更新结果马上会被写入数据库；对于延迟更新，用户必须手动调用 IRowsetUpdate：：Update，更新结果才会写入数据库。当调用 IRowsetUpdate：：Update 时，神通数据库 OLE DB 会处理用户指定的所有行，如果某一行的更新因为数据，长度或者状态不正确而失败，OLE DB 不会停止其他行的处理。如果调用的返回值是 DB_S_ERRORSOCCURED，那用户应该检查 prgRowStatus 数组。

用户不应该对神通数据库 OLE DB 进行行数据更新的顺序有任何假设。如果应用需要按某种顺序更新数据，那这个顺序应该由应用来维护。

2.2.5 数据类型

神通数据库所支持的类型和神通数据库 OLEDB 数据类型的对应关系如下表：

表 2-1 数据类型的对应关系

神通数据库类型	OLEDB 类型
TINYINT	DBTYPE_I1
SMALLINT	DBTYPE_I2
INT	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
FLOAT	DBTYPE_R8
DOUBLE PRECISION	DBTYPE_R8
DECIMAL(p, s)	DBTYPE_NUMERIC
NUMERIC(p, s)	DBTYPE_NUMERIC
BIT	DBTYPE_BOOL

BOOL	DBTYPE_BOOL
CHAR(n)	DBTYPE_STR
VARCHAR(n)	DBTYPE_STR
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
BINARY(n)	DBTYPE_BYTES
VARBINARY(n)	DBTYPE_BYTES
BLOB	DBTYPE_BYTES
CLOB	DBTYPE_STR
TEXT	DBTYPE_STR

目前，神通数据库 OLEDB 还不支持神通数据库的 INTERVAL 类型。

2.2.6 事务

神通数据库 OLE DB 支持本地事务。对于分布式事务，用户可以通过 Microsoft Distributed Transaction Coordinator (MS DTC)来实现。

缺省情况下，神通数据库 OLE DB 使用自动提交模式，用户在通过 Session 执行的每个对数据库的访问动作都包含在一个完整的单独的事务中。用户可以通过 ITransactionLocal 接口来改变 Session 的提交模式。神通数据库 OLE DB 不支持嵌套事务。

2.2.7 神通数据库 OLE DB 枚举器

神通数据库 OLE DB 提供一个枚举器组件，用户可以使用它得到网络中所有可用神通数据库 OLE DB 连接上的数据源列表。枚举器组件由 GUID CLSID_OSCAROLEDB_ENUMERATOR ＝ {}来标识。

下面举例说明。枚举 SHENTONG OLE DB 数据源的步骤如下：

通过调用 ISourceRowset::GetSourcesRowset 检索源行集。

通过调用 GetColumnInfo::IcolumnInfo 查找对枚举符行集的描述。

根据列信息创建绑定结构。

通过调用 IAccessor::CreateAccessor 创建行集存取程序。

通过调用 IRowset::GetNextRows 提取行。

通过调用 IRowset::GetData 从行集的行复本中检索数据并对数据进行处理。

```
//How to use the enumerator object to list
//the data sources available.

#define UNICODE
#define _UNICODE
#define DBINITCONSTANTS
#define INITGUID

#include <windows.h>
```

```
#include <stddef.h>
#include <oledb.h>
#include <oledberr.h>
#include <SQLOLEDB.h>
#include <stdio.h>

#define NUMROWS_CHUNK    5

//AdjustLen supports binding on four-byte boundaries.
_inline ULONG AdjustLen(ULONG cb)
{
        return ((cb + 3) & ~3);
}

// Get the characteristics of the rowset (the IColumnsInfo interface).
HRESULT GetColumnInfo
        (
        IRowset*                                pIRowset,
        UINT*                                   pnCols,
        DBCOLUMNINFO**                          ppColumnsInfo,
        OLECHAR**                               ppColumnStrings
        )
        {
        IColumnsInfo*     pIColumnsInfo;
        HRESULT           hr;

        *pnCols = 0;
        if (FAILED(pIRowset->QueryInterface(IID_IColumnsInfo,
                                                (void**) &pIColumnsInfo)))
                {
                return (E_FAIL);
                }

        hr = pIColumnsInfo->GetColumnInfo((ULONG*) pnCols,
                                                ppColumnsInfo,
                                                ppColumnStrings);
        if (FAILED(hr))
                {
                //Process error.
                }
        pIColumnsInfo->Release();

        return (hr);
        }
```

```
// Create binding structures from column information. Binding structures
// will be used to create an accessor that allows row value retrieval.
void CreateDBBindings
    (
    UINT                nCols,
    DBCOLUMNINFO*       pColumnsInfo,
    DBBINDING**         ppDBBindings,
    BYTE**              ppRowValues
    )
    {
    ULONG               nCol;
    ULONG               cbRow = 0;
    ULONG               cbCol;
    DBBINDING*          pDBBindings;
    BYTE*               pRowValues;

    pDBBindings = new DBBINDING[nCols];

    for (nCol = 0; nCol < nCols; nCol++)
        {
        pDBBindings[nCol].iOrdinal = nCol+1;
        pDBBindings[nCol].pTypeInfo = NULL;
        pDBBindings[nCol].pObject = NULL;
        pDBBindings[nCol].pBindExt = NULL;
        pDBBindings[nCol].dwPart = DBPART_VALUE;
        pDBBindings[nCol].dwMemOwner =
DBMEMOWNER_CLIENTOWNED;
        pDBBindings[nCol].eParamIO = DBPARAMIO_NOTPARAM;
        pDBBindings[nCol].dwFlags = 0;
        pDBBindings[nCol].wType = pColumnsInfo[nCol].wType;
        pDBBindings[nCol].bPrecision = pColumnsInfo[nCol].bPrecision;
        pDBBindings[nCol].bScale = pColumnsInfo[nCol].bScale;

        cbCol = pColumnsInfo[nCol].ulColumnSize;

        switch (pColumnsInfo[nCol].wType)
            {
            case DBTYPE_STR:
                {
                cbCol += 1;
                break;
                }
```

```
                case DBTYPE_WSTR:
                    {
                    cbCol = (cbCol + 1) * sizeof(WCHAR);
                    break;
                    }

                default:
                    break;
                }

            pDBBindings[nCol].obValue = cbRow;

            pDBBindings[nCol].cbMaxLen = cbCol;
            cbRow += AdjustLen(cbCol);
        }

    pRowValues = new BYTE[cbRow];

    *ppDBBindings = pDBBindings;
    *ppRowValues = pRowValues;

    return;
    }

int main()
{
    ISourcesRowset*         pISourceRowset = NULL;
    IRowset*                pIRowset = NULL;
    IAccessor*              pIAccessor = NULL;
    DBBINDING*              pDBBindings = NULL;
    HROW*                   pRows = new HROW[500];
    BYTE*                   pData = NULL;
    HACCESSOR               hAccessorRetrieve = NULL;
    ULONG                   cRows = 0;
    ULONG                   DSSeqNumber = 0;
    HRESULT                 hr;
    UINT                    nCols;
    DBCOLUMNINFO*           pColumnsInfo = NULL;
    OLECHAR*                pColumnStrings = NULL;
    DBBINDSTATUS*           pDBBindStatus = NULL;
    BYTE*                   pRowValues = NULL;
    ULONG                   cRowsObtained;
    ULONG                   iRow;
    char*                   pMultiByte = NULL;
```

```
    short*                          psSourceType = NULL;
    BYTE*                           pDatasource = NULL;

    //Initialize COM library.
    CoInitialize(NULL);

    //Initialize the enumerator.
    if(FAILED(CoCreateInstance(CLSID_OSCAROLEDB_ENUMERATOR,
                                     NULL,
                                     CLSCTX_INPROC_SERVER,
                                     IID_ISourcesRowset,
                                     (void**)&pISourceRowset)))
    {
        //Process error.
        return TRUE;
    }

    //Retrieve the source rowset.
    hr = pISourceRowset->GetSourcesRowset(NULL,
                                               IID_IRowset,
                                               0,
                                               NULL,
                                               (IUnknown**)&pIRowset);
    pISourceRowset->Release();
    if(FAILED(hr))
    {
        //Process error.
        return TRUE;
    }
    //Get the description of the enumerator's rowset.
    if(FAILED(hr = GetColumnInfo(pIRowset,
                                      &nCols,
                                      &pColumnsInfo,
                                      &pColumnStrings)))
    {
        //Process error.
        goto SAFE_EXIT;
    }

    //Create the binding structures.
    CreateDBBindings(nCols,
                     pColumnsInfo,
                     &pDBBindings,
                     &pRowValues);
```

```
        pDBBindStatus = new DBBINDSTATUS[nCols];

        if (sizeof(TCHAR) != sizeof(WCHAR))
        {
                pMultiByte = new char[pDBBindings[0].cbMaxLen];
        }
        if(FAILED(pIRowset->QueryInterface(IID_IAccessor,
(void**)&pIAccessor)))
        {
                //Process error.
                goto SAFE_EXIT;
        }
        //Create the rowset accessor.
        if(FAILED(hr = pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,
                                                  nCols,
                                                  pDBBindings,
                                                  0,
                                                  &hAccessorRetrieve,
                                                  pDBBindStatus)))
        {
                //Process error.
                goto SAFE_EXIT;
        }

        //Process all the rows, NUMROWS_CHUNK rows at a time.
        while (SUCCEEDED(hr))
        {
        hr=pIRowset->GetNextRows(NULL,
                                 0,
                                 NUMROWS_CHUNK,
                                 &cRowsObtained,
                                 &pRows);
        if(FAILED(hr))
        {
                //process error
        }
        if(cRowsObtained == 0 || FAILED(hr))
                break;

                for(iRow = 0; iRow < cRowsObtained; iRow++)
                {
                        //Get the rowset data.
                        if(SUCCEEDED(hr = pIRowset->GetData(pRows[iRow],
                                                            hAccessorRetrieve,
```

```
                                                pRowValues)))
            {
            psSourceType = (short *)(pRowValues +
                                    pDBBindings[3].obValue);

            if (*psSourceType == DBSOURCETYPE_DATASOURCE)
            {
                DSSeqNumber = DSSeqNumber + 1;
 //Data source counter.
                pDatasource = (pRowValues +
pDBBindings[0].obValue);

                if(sizeof(TCHAR) != sizeof(WCHAR))
                {
                    WideCharToMultiByte(CP_ACP, 0,
                        (WCHAR*)pDatasource, -1, pMultiByte,
                        pDBBindings[0].cbMaxLen, NULL, NULL);

                    printf( "DataSource# %d\tName: %S\n",
                              DSSeqNumber, (WCHAR *) MultiByte );
                }
                else
                {
                    printf( "DataSource# %d\tName: %S\n",
                      DSSeqNumber, (WCHAR *) pDatasource );
                } //if
            }   //if
        } //if

    } //for
        pIRowset->ReleaseRows(cRowsObtained, pRows, NULL,
NULL, NULL);
    } //while
    //Release COM library.
    CoUninitialize();

    return(0);
SAFE_EXIT:
//Do the clean-up.
    return TRUE;
};
```

2.3OLE DB 编程参考

神通数据库 OLE DB 是一个符合 OLE DB 2.0 的 OLE DB 提供者(OLE DB Provider)。这里不对 OLE DB 的相关标准和规范进行重复描述。编程参考以及完整的 OLE DB API 可以参看 Microsoft OLE DB Software Development Kit (SDK)。 OLE DB SDK is 是 Microsoft Developer Network (MSDN®)的一个部分，可以从微软的官方网站获得。

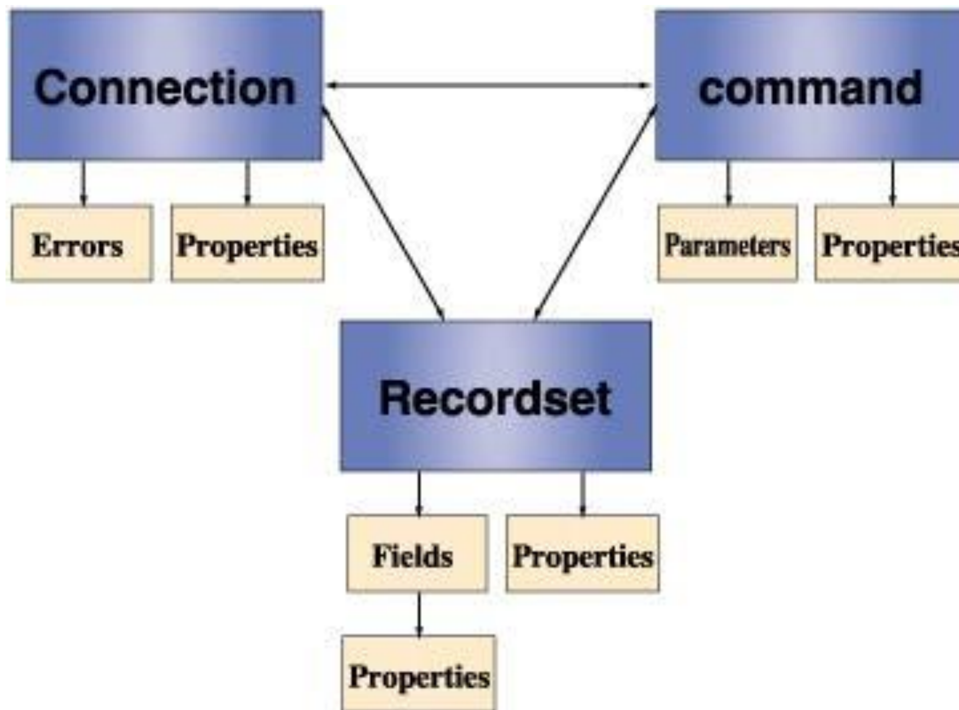
第3章 神通数据库 ADO 应用参考

本章内容包括：

1. ADO 对象模型的概述
2. 如何应用 ADO 三个主体对象 Connection、Command 和 Recordset
3. 如何在不同的语言环境中使用 ADO 对象

3.1概述

ADO 对象模型定义了一组可编程的自动化对象，可用于 Visual Basic、Visual C++、Java 以及其他各种支持自动化特性的脚本语言。ADO 最早被用于 Microsoft Internet Information Server 中访问数据库的接口，与一般的数据库接口相比，ADO 可更好地用于网络环境，通过优化技术，它尽可能地降低网络流量；ADO 的另一个特性是使用简单，不仅因为它是一个面向高级用户的数据库接口，更因为它使用了一组简化的接口用以处理各种数据源。这两个特性使得 ADO 必将取代 RDO 和 DAO，成为最终的应用层数据接口标准。



从上图我们也看到了 ADO 实际上是 OLE DB 的应用层接口，这种结构也为一致的数据访问接口提供了很好的扩展性，而不再局限于特定的数据源，因此，ADO 可以处理各种 OLE DB 支持的数据源。

在 ADO 模型中，主体对象只有 3 个：Connection、Command 和 Recordset，其他 4 个集合对象 Errors、Properties、Parameters 和 Fields 分别对应 Error、Property、Parameter 和 Field 对象，整个 ADO 对象模型由这些对象组成。

一个典型的 ADO 应用使用 Connection 对象建立与数据源的连接，然后用一个 Command 对象给出对数据库操作的命令，比如查询或者更新数据等，而 Recordset 用于对结果集数据进行维护或者浏览等操作。Command 命令所使用的命令语言与底层所对应的 OLE DB 数据源有关，不同的数据源可以使用不同的命令语言，对于关系型数据库，通常使用 SQL 作为命令语言。

在 Connection、Command 和 Recordset 3 个对象中，Command 对象是个可选对象，它是否有效取决于 OLE DB 数据提供者是否实现了 ICommand 接口。由于 OLE DB 可提供关系型数据源也可以提供非关系型数据源，所以在非关系型数据源上使用传统的 SQL 命令查询数据有可能无效，甚至 Command 命令对象也不能使用。

3.2神通数据库 ADO 应用参考

3.2.1 Connection 对象

Connection 对象代表与数据源之间的一个连接，ADO 的 Connection 对象封装了 OLE DB 的数据源对象和会话对象。根据 OLE DB 提供者的不同性能，Connection 对象的特性也有所不同，所以 Connection 对象的方法和属性不一定都可以使用。利用 Connection 对象，我们可以完成以下一些基本设置操作。

a. 通过 ConnectionString、ConnectionTimeOut 和 Mode 属性设置连接串、超时信息、访问模式。

b. 还可以设置 CursorLocation 属性以便指定使用客户端游标，以便在客户程序中使用批处理修改方式。

c. 设置连接的缺省数据库属性 DefaultDatabase。

d. 设置 OLE DB 提供者的属性 Provider。

e. 通过 Open 和 Close 控制 Connection 对象与物理数据源的连接。

f. 通过 Execute 方法执行命令。

g. 提供事务机制，通过 BeginTrans、CommitTrans 和 RollbackTrans 方法实现事务控制。

h. 通过 Errors 集合属性检查数据源的错误信息。

i. 通过 OpenSchema 方法获取数据库的表信息。

Connection 对象是 ADO 的基本对象之一，它独立于所有其他的对象。如果我们要对数据库进行查询操作，既可以使用 Execute 方法，也可以使用 Command 对象。使用 Execute 方法比较简便，但用 Command 对象可以保存命令的信息，以便多次查询。

3.2.2 Command 对象

Command 对象代表一个命令，可以通过其方法执行针对数据源的有关操作，比如查询、修改等。Command 对象的用法如下：

a. 通过 CommandText 属性设置命令串。

b. 通过 Parameters 集合属性和 Parameter 对象定义参数化查询或存储过程的参数。

c. 通过 Execute 方法执行命令，可能的话，返回 Recordset 对象。

d. 在执行命令之前，可通过设置 CommandType 属性以便优化性能。

e. 可以通过 Prepared 属性指示底层的提供者为当前命令准备一个编译过的版本，以后再执行时，速度会大大加快。

f. 通过 CommandTimeOut 属性设置命令执行的超时值(以秒为单位)。

g. 可以设置 ActiveConnection 属性，为命令指定连接串，Command 对象将在内部创建 Connection 对象。

h. 可以设置 Name 属性，这样以后可以在相应的 Connection 对象上按 Name 属性指定的方法名执行。

Command 对象执行时，既可以通过 ActiveConnection 属性指定相连的 Connection 对象，也可以独立于 Connection 对象，直接指定连接串，即使连接串与 Connection 对象的连接串相同，Command 对象仍然使用其内部的数据源连接。

3.2.3 Recordset 对象

Recordset 对象代表一个表的记录集或者命令执行的结果，在记录集中，总是有一个当前的记录。记录集是 ADO 管理数据的基本对象，所有的 Recordset 对象都按照行列方式的表状结构进行管理，每一行对应一个记录(Record)，每一列对应一个域(Field)。Recordset 对象也通过游标对记录进行访问，在 ADO 中，游标分为以下 4 种：

静态游标提供对数据集的一个静态拷贝，允许各种移动操作，包括前移、后移等等，但其他用户所做的操作反映不出来。动态游标允许各种移动操作，包括前移、后移等等，并且其他用户所做的操作也可以直接反映出来。

动态游标允许各种前向移动操作，不能向后移动，并且其他用户所做的操作也可以直接反映出来。

键集(keyset) 游标类似于动态游标，也能够看到其他用户所做的数据修改，但不能看到其他用户新加的记录，也不能访问其他用户删除的记录。

Recordset 对象的用法如下：

a. 利用 CursorType 属性设置游标类型。

b. 通过 Open 方法打开记录集数据，既可以在 Open 之前对 ActiveConnection 属性赋值，指定 Recordset 对象使用连接对象，也可以直接在 Open 方法中指定连接串参数，ADO 将创建一个内部连接，即使连接串与外部的连接对象相同，它也使用新的连接对象。

c. Recordset 对象刚打开时，当前记录被定位在首条记录，并且 BOF 和 EOF 标志属性为 False，如果当前记录集为空记录集，则 BOF 和 EOF 标志属性为 True。

d. 通过 MoveFirst、MoveLast、MoveNext 和 MovePrevious 方法可以对记录集的游标进行移动操作。如果 OLE DB 提供者支持相关功能的话，可以使用 AbsolutePosition、AbsolutePage 和 Filter 属性对当前记录重新定位。

e. ADO 提供了两种记录修改方式：立即修改和批修改。在立即修改方式下，一旦调用 Update 方法，则所有对数据的修改立即被写到底层的数据源。在批修改方式下，可以对多条记录进行修改，然后调用 UpdateBatch 方法把所有的修改递交到底层数据源。递交之后，可以用 Status 属性检查数据冲突。

Recordset 对象是 ADO 数据操作的核心，它既可以作为 Connection 对象或 Command 对象执行特定方法的结果数据集，也可以独立于这两个对象而使用，由此可以看出 ADO 对象在使用上的灵活性。

上面 3 个对象都包含一个 Property 对象集合的属性，通过 Property 对象可使 ADO 动态暴露出底层 OLE DB 提供者的性能。由于并不是所有的底层提供者都有同样的性能，所以 ADO 允许用户动态访问底层提供者的能力。这样既使得 ADO 很灵活，又提供了很好的扩展性。ADO 的其他集合对象及其元素对象，都用在特定的上下文环境中，比如 Parameter 对象一定要与某个 Command 对象相联系后，才能真正起作用。而另外三个对象 Field、Error 和 Property 对象只能依附于其父对象，不能单独创建这些对象。

3.2.4 在多种语言中使用 ADO

以上介绍了 ADO 的对象模型，现在我们来讨论如何在不同的语言环境中使用 ADO 对象。因为 ADO 是作为自动化组件程序实现的，所以我们可以在任何支持 COM 和自动化特性的语言环境中使用 ADO，比如 Visual Basic、Visual C++、ASP 和 Java 等等，下面分别加以介绍。

1. 在 Visual Basic 应用中使用 ADO

Visual Basic 应用在设计模式和运行模式下都可以创建和使用自动化对象，在设计模式下，像 ADO 这样的对象库可以作为内部对象来使用，我们只需在"Project" 菜单下的 "References" 命令弹出的对话框中选中 ADO 对象库"Microsoft ActiveX Data Objects Library"，于是我们就可以在程序中直接声明或新建 ADO 对象，举例如下：

```
Dim cn as New ADODB.Connection
Dim cmd as New ADODB.Command
Dim rs as New ADODB.Recordset
```

可以看出，在设计时使用 ADO 对象非常方便，而且 Visual Basic 设计环境中提供的对象浏览器(Object Browser) 功能允许用户很方便地查看 ADO 对象的属性和方法。

我们也可以在运行时创建自动化对象，使用 Visual Basic 的 CreateObject 函数可以创建任意的自动化对象，由于 ADO 中只有 Connection 对象、Command 对象和 Recordset 对象可以被独立创建，所以我们也只能创建这 3 种对象，举例如下：

```
Dim cn
Set rs=CreateObject("ADODB.Connection ")
Dim cmd
Set rs=CreateObject("ADODB.Command")
Dim rs
Set rs=CreateObject("ADODB.Recordset")
```

不管是设计模式还是运行模式，调用 ADO 对象的属性和方法都非常简单，直接调用即可。

2. 在 ASP 的 VBScript 中使用 ADO

ADO 对象也可以用于 HTML 和 Active Server Page 的 VBScript 脚本代码，VBScript 脚本代码与 Visual Basic 的代码很类似，它们内嵌在 HTML 或 ASP 文件的特定标记对内部。但 VBScript 引擎比 Visual Basic 的设计环境或运行库在功能上还是要弱一些，首先，在 VBScript 代码中，没有与设计环境类似的用法，VBScript 引擎不能装入 ADO 类型库，所以不能使用 New 操作符创建 ADO 对象，但可以使用 CreateObject 函数创建对象；其次，ADO 对象库中用到的常量只能通过包含文件引入，随 ADO 一起提供的 Adovbs.inc 文件包含所有 ADO 常量的定义，我们可在脚本代码中直接包含此文件。

因此，为了在 VBScript 代码中使用 ADO，首先要包含 Adovbs.inc 文件，然后使用 CreateObject 函数创建 ADO 对象，以后就可以调用这些对象的属性或方法了。下面的例子显示了在 ASP 文件中用 ADO 列出数据表中所有作者的姓名和职称，代码如下：

```
< % @ LANGUAGE = VBScript % >
< HTML >
< TITLE >Using ADO in a Visual Basic Script Web Page
< /TITLE >< /HEAD >
```

```
< LANGUAGE="VBS" >
< !--#include file="adovbs.inc"-- >
< CENTER >
< H1 >< font size=4 >Using ADO in a Visual Basic Script Web Page
< /H1 >< /font >< br >< br >
< %set myConnection = CreateObject("ADODB.Connection")
myConnection.Open "DSN=MySamples;UID=sa"
SQLQuery = "select AuthorName, Title from AuthorDB"
set rs = myConnection.Execute(SQLQuery)% >
< TABLE align=center COLSPAN=8 CELLPADDING=5 BORDER=0 WIDTH=200 >
< !-- BEGIN column header row -- >
< TR >
< TD VALIGN=TOP BGCOLOR="#800000" >
< FONT STYLE="ARIAL NARROW" COLOR="#ffffff" SIZE=1 >
Title ID< /FONT >
< /TD >
< TD ALIGN=CENTER BGCOLOR="#800000" >
< FONT STYLE="ARIAL NARROW" COLOR="#ffffff" SIZE=1 > Title< /FONT ><
/TD >< /TR >
< !-- Get Data -- >< % do while not rs.EOF % >
< TR >
< TD BGcolor ="f7efde" align=center >
< font style ="arial narrow" size=1 >
< %=rs("AuthorName")% >< /font >
< /TD >
< TD BGcolor ="f7efde" align=center >< font style ="arial narrow" size=1 >
< %=rs("Title") % >
< /font >< /TD >< /TR >
< % rs.MoveNext% >< %loop % >< !-- Next Row -- >
< /TABLE >< /center >< /BODY >< /HTML >
```

3. 在 Visual C++ 中使用 ADO

在 Visual C++ 中使用 ADO 有多种方法，第一种方法是我们使用 CoCreateInstance 函数创建 ADO 对象，并得到对象的 IDispatch 接口指针，然后调用其 Invoke 函数，用这种方法需要我们自己处理参数和返回值，ADO 提供了 Adoid.h 和 Adoint.h 头文件分别定义了 ADO 对象的 CLSID 和接口 ID；第二种方法是利用#import 编译指示符(在 Visual C++ 5.0 及以后的版本中可以使用)，可以方便地使用 ADO 对象；第三种方法是利用 MFC(Microsoft Foundation Class) 库提供的 IDispatch 接口封装类 COleDispatchDriver 创建和调用 ADO 对象。

下面的代码显示了在 Visual C++ 创建数据源连接的过程：

```
GUID connectionCLSID;
HRESULT hResult = ::CLSIDFromProgID(L"ADODB.Connection", &connectionCLSID);
if (FAILED(hResult))
{
......
```

```
  }
IDispatch *pDispatch = NULL;
hResult = CoCreateInstance(connectionCLSID, NULL, CLSCTX_SERVER,
IID_IDispatch, (void **)&pDispatch);
if (FAILED(hResult))
{
......
}
COleDispatchDriver driver;
driver.AttachDispatch(pDispatch, FALSE);
TRY
{
   BYTE parms[] =VTS_BSTR;
  driver.InvokeHelper(0xa,DISPATCH_METHOD,VT_EMPTY,&hResult,
parms,L"Provider=SQLOLEDB; User ID=sa;Password=;"
L"Initial Catalog=LEAVES;Data Source=NetTestServer");
}
END_TRY
driver.DetachDispatch();
```

4. 在 Java 中使用 ADO

在 Java 程序中可以引入 ADODB 类，然后声明 ADO 变量，也可以使用 new 操作符创建 ADO 变量。下面的代码说明了如何在 Java 中打开与数据源连接：

```
import msado10.*;
_Connection m_conn = null;
_Recordset       m_rs = null;
_Command        m_cmd = null;
void OpenConnection()
{
        String                  s;
        Properties        properties;
Try
{
                properties = m_conn.getProperties();
m_conn.Open("dsn=MySamples", "sa", "");
properties = null;
}
catch (Exception e)
{
System.out.println("\nUnable to make a connection \n");
}
}
```