VC++ 实战 OLEDB 编程

OLEDB 作为目前最全面，最强大的 Windows 平台下的数据库编程接口，其资料在网上却少之又少，这着实有些让人纳罕。

现在很多的应用软件系统都要和数据库打交道，没有一个好的强大的数据库编程接口作为支撑，这些系统的功能，性能，安全性等等都将是不可想象的事情。当然我们还可以选择 OLEDB 之上的 ADO 接口来作为我们的编程接口，由于 ADO 是基于 OLEDB 的上层封装，ADO 比之 OLEDB 最大的优势就在于方便。使用 ADO 只需要 3-5 行代码的事情，用 OLEDB 却需要将近 200-300 行代码才能完成，这也是很多系统目前使用 ADO 作为数据库编程接口的主要原因。当然这并不表示 OLEDB 在功能上比 ADO 有什么逊色之处，相反，OLEDB 接口不但功能强大，而且由于处于底层的原因，它在灵活性上有更明显的优势。甚至我鼓励 VC++程序员在大型项目中使用 OLEDB 而不是 ADO 来作为数据库编程的接口，因为 OLEDB 的很多东西都是由你完全控制的，比如数据存放的位置，存放的方式等等。

另外 OLEDB 还可以作为数据提供者编程接口，比如你可以自己实现一个 ORACLE 或 MySQL 的 OLEDB 提供程序。

好了，废话就不多说了，下面就开始我们正式的 OLEDB 之旅，注意本文以及以后一系列文章中所有的关于 OLEDB 的介绍都以 OLEDB2.6 以上版本为准，你可以在微软的网站上免费下载到完整的 OLEDB2.6SDK 包，其中有关于 OLEDB，ADO 等的详细帮助以及例子代码。

首先让我们来看看 OLEDB 中两个最基本的概念：数据提供者和数据消费者。在 OLEDB 中，不但要考虑使用数据的一方，还要考虑提供数据的一方，比如各种数据库系统几乎都提供了自己的 OLEDB 接口。从本质上说，OLEDB 其实就是一个标准的数据库与应用系统间的数据标准交换接口，它的好处就是高效，通用和灵活。

如果是数据提供的一方，在 OLEDB 中就称之为数据提供者，如果是使用数据的一方，那么就称之为数据消费者。在此系列文章中将主要的介绍一下数据消费者接口。

OLEDB 顾名思义，它肯定是一组 COM 接口，因此要用好 OLEDB，一个必不可少的先决条件就是必须熟悉 COM 原理，当然这对现在的 VC++程序员来说是必须掌握的技能之一，本文中不对 COM 的基本内容做任何介绍，需要的请自己查阅相关资料。

要完成一个数据库访问任务，第一件要做的事情就是打开一个连接，在 OLEDB2.0 以前的版本中，要做到这点只需要像下面这样调用一段代码就可以创建一个数据库连接：

```
#define COM_NO_WINDOWS_H      //如果已经包含了 Windows.h 或不使用其他 Windows

//库函数时

#define DBINITCONSTANTS

#define INITGUID

#define OLEDBVER 0x0250

#include <oledb.h>

#include <oledberr.h>

IDBInitialize *pIDBInitialize = NULL;
CoCreateInstance(CLSID_MSDASQL, NULL, CLSCTX_INPROC_SERVER,

                    IID_IDBInitialize,(void**)&pIDBInitialize);
```

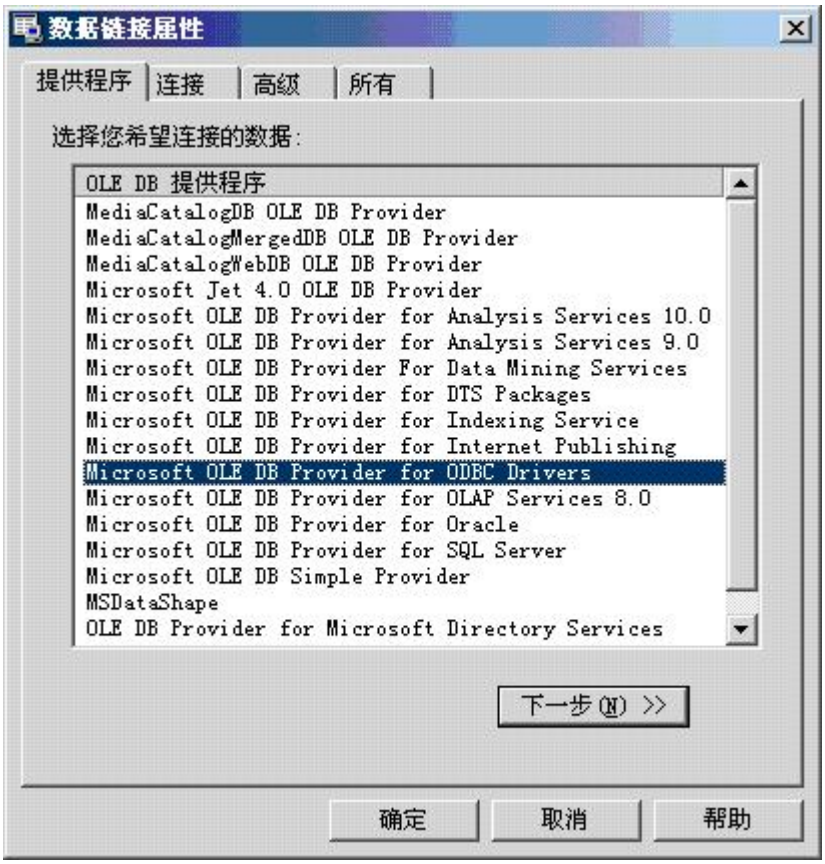
当然如果你使用的 SQL Server 2005 以上的数据库，那么还可以像下面这样来创建一个连接对象：

```
#include <sqlncli.h>

CoCreateInstance( CLSID_SQLNCLI10,NULL,CLSCTX_INPROC_SERVER,

IID_IDBInitialize,(void **) &pIDBInitialize);
```

这样调用的前提是你已经安装了 SQL Server Native Client 支持组件，当然这还需要你包含 SQLNCLI.h 这个头文件，这个在 Windows SDK 组件中找到它。

当然这只是创建了一个数据库连接的 COM 接口，而且要特别注意的是这是 OLEDB2.0 以前推荐的方法，现在不应该再这样使用了，介绍它的目的就是方便你看懂一些例子代码。

同时这个地方要特别注意的是，这种方法初始化出来的 IDBInitialize 接口，在后续的调用中是无法使用一些高级接口的，比如 IRowsetFind、IRowsetIdentity 等等，同时这将导致你后续创建的 IRowset 接口无法直接和 ADO 混用（ADO 和 OLEDB 混用这个话题将在后续的文章中讨论），在 2.0 以后的 OLEDB 接口中，替代的是两个接口：IDataInitialize 和 IDBPromptInitialize。其中 IDBPromptInitialize 将打开那个 Windows 系统标准的数据库连接创建对话框，像下图这样：



而 IDataInitialize 则纯粹的是以代码的方式创建一个数据库连接对象。它们分别用于不同的编程目的。调用这两个接口例子如下：

1、使用 IDataInitialize 接口：

```
IDataInitialize* pIDataInitialize = NULL;

CoCreateInstance(CLSID_MSDAINITIALIZE, NULL, CLSCTX_INPROC_SERVER,

    IID_IDataInitialize,(void**)&pIDataInitialize);

IDBInitialize *pIDBInitialize = NULL;

pIDataInitialize->CreateDBInstance(CLSID_MSDASQL, NULL,

    CLSCTX_INPROC_SERVER, NULL, IID_IDBInitialize,

    (IUnknown**)&pIDBInitialize);
```

2、使用 IDBPromptInitialize 接口：

```
IDBPromptInitialize* pIDBPromptInitialize = NULL;

CoCreateInstance(CLSID_DataLinks, NULL, CLSCTX_INPROC_SERVER,
```

```
    IID_IDBPromptInitialize, (void **)&pIDBPromptInitialize);

IDBInitialize *pIDBInitialize = NULL;

//下面这句将弹出前面所说的对话框

pIDBPromptInitialize->PromptDataSource(NULL, hWndParent,

    DBPROMPTOPTIONS_PROPERTYSHEET, 0, NULL, NULL, IID_IDBInitialize,

    (IUnknown **)&pIDBInitialize);

pIDBInitialize->Initialize();//根据对话框采集的参数连接到指定的数据库
```

上面两种方法中实际最后还是创建了 IDBInitialize 接口，表面上看这没什么区别，其实不然，在这个地方要特别注意的是，一定要使用后面这两种方法创建的 IDBInitialize 的接口，虽然很多例子中都像文中前面提到的那样直接创建了 IDBInitialize 接口，但是在正式的项目代码中一定要使用后面这两种方法之一来创建 IDBInitialize 接口，否则你就会遇到很多奇怪的问题，最重要的就是后面这两种方法打开了所有的数据源 OLEDB 功能支持，而直接创建的 IDBInitialize 只打开了基本 OLEDB 功能支持，这在项目中会导致很多问题，因为很多的接口以及属性都将无法使用。而后两种方法保证你后续所有的 OLEDB 接口和功能都能使用。

在创建了 IDBInitialize 接口之后（pIDBPromptInitialize->PromptDataSource 这种方式创建的除外），我们就需要详细的指定连接数据库的各种参数，在 OLEDB 中这些参数都叫做属性，并且被加以分组，每个分组都叫做一个属性集合。典型的我们需要指定数据库实例名，数据库名，连接数据库的用户名以及密码等等属性。在 SQL Server 中有一种特殊的方式就是集成安全模式，它的属性相对简单些，下面的例子演示了如何使用集成安全性连接到 SQL Server 数据库：

```
DBPROP  InitProperties[4];

DBPROPSET  rgInitPropSet[1];

//初始化属性值变量

for ( i = 0 ; i < 4 ; i++ )

{

        VariantInit(&InitProperties[i].vValue);

}

//指定数据库实例名，这里使用了别名 local，指定本地默认实例

InitProperties[0].dwPropertyID = DBPROP_INIT_DATASOURCE;

InitProperties[0].vValue.vt = VT_BSTR;

InitProperties[0].vValue.bstrVal= SysAllocString(L"(local)");

InitProperties[0].dwOptions = DBPROPOPTIONS_REQUIRED;
```

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

//指定数据库名

InitProperties[1].dwPropertyID = DBPROP_INIT_CATALOG;

InitProperties[1].vValue.vt = VT_BSTR;

InitProperties[1].vValue.bstrVal = SysAllocString(L"MyTest");

InitProperties[1].dwOptions = DBPROPOPTIONS_REQUIRED;

InitProperties[1].colid = DB_NULLID;

//指定身份验证方式为集成安全模式“SSPI”

InitProperties[2].dwPropertyID = DBPROP_AUTH_INTEGRATED;

InitProperties[2].vValue.vt = VT_BSTR;

InitProperties[2].vValue.bstrVal = SysAllocString(L"SSPI");

InitProperties[2].dwOptions = DBPROPOPTIONS_REQUIRED;
```

```
InitProperties[2].colid = DB_NULLID;

//创建一个 GUID 为 DBPROPSET_DBINIT 的属性集合，这也是初始化连接时需要的唯一一//个属性集合

rgInitPropSet[0].guidPropertySet = DBPROPSET_DBINIT;

rgInitPropSet[0].cProperties = 4;

rgInitPropSet[0].rgProperties = InitProperties;

//得到数据库初始化的属性接口

IDBProperties* pIDBProperties = NULL;

hr = pIDBInitialize->QueryInterface(IID_IDBProperties, (void **)&pIDBProperties);

if (FAILED(hr))

{//无法得到 IDBProperties 接口，详细的错误信息可以使用 IerrorRecords 接口得到

return FALSE;
```

```
}

hr = pIDBProperties->SetProperties(1, rgInitPropSet);

if (FAILED(hr))

{//设置属性失败

return -1;

}

//属性一但设置完成，相应的接口就可以释放了

pIDBProperties->Release();

//根据指定的属性连接到数据库

pIDBInitialize->Initialize();
```

在上面的代码中，新加入了一个重要的概念就是属性和属性集合，对于 OLEDB 来说，除了接口，就是属性和属性集合最重要了，在 MSDN 或 SDK 自带的帮助文档中都有关于每个属性以及属性集合的详细介绍，尤其是详细的说明了每个属性控制的接口行为，弄清楚这些 OLEDB 就没有什么秘密可言了，还有

一点就是有些 OLEDB 提供者，如 ORACLE 的 OLEDB 提供者还添加了一些特殊的属性，这些属性往往就要看相应的提供者帮助文档才能知道，在 MSDN 中通常是没有这些特殊属性的任何有用信息的。

对于一个属性集合而言，最重要的就是它的标识，标识其实就是一个预定义的 GUID 值，在连接数据库时，最重要的属性集合就是 DBPROPSET_DBINIT，在这个属性集合中有下列的属性：

DBPROP_AUTH_CACHE_AUTHINFO

DBPROP_AUTH_ENCRYPT_PASSWORD

DBPROP_AUTH_INTEGRATED

DBPROP_AUTH_MASK_PASSWORD

DBPROP_AUTH_PASSWORD

DBPROP_AUTH_PERSIST_ENCRYPTED

DBPROP_AUTH_PERSIST_SENSITIVE_AUTHINFO

DBPROP_AUTH_USERID

DBPROP_INIT_ASYNCH

DBPROP_INIT_BINDFLAGS

DBPROP_INIT_GENERALTIMEOUT

DBPROP_INIT_HWND

DBPROP_INIT_IMPERSONATION_LEVEL

DBPROP_INIT_LCID

DBPROP_INIT_LOCATION

DBPROP_INIT_LOCKOWNER

DBPROP_INIT_MODE

DBPROP_INIT_OLEDBSERVICES

DBPROP_INIT_PROMPT

DBPROP_INIT_PROTECTION_LEVEL

DBPROP_INIT_CATALOG　　DBPROP_INIT_PROVIDERSTRING

DBPROP_INIT_DATASOURCE　　DBPROP_INIT_TIMEOUT

这些属性不一一赘述，你可以在 MSDN2008 中找到关于它们的详细介绍，这里只讲几个重要常用的属性：

DBPROP_INIT_DATASOURCE 这个属性指定我们要连接的数据库实例名，它是一个字符串型属性，前面的例子中已经演示了如何设定这个属性，夸张的是这需要 5 行代码（不包括属性定义）。通常对于一个项目来说，最不固定的就是这个参数，因此建议这个参数放置到配置文件中，以方便连接到部署在不同地方的数据库。

DBPROP_INIT_CATALOG 是第二个比较重要的属性，它将指定我们要连接的具体数据库的名称，注意它叫做 CATALOG（目录），而不是数据库什么的，因此这是一个比较不好记住的属性。

如果我们没有指定 DBPROP_AUTH_INTEGRATED 这个属性，那么我们就需要提供连接数据库的用户名和密码，其中用户名使用 DBPROP_AUTH_INTEGRATED 属性指定，这也是一个字符串型的变量，而密码则使用 DBPROP_AUTH_PASSWORD 属性来指定。

在设置属性时要特别注意的就是，一个属性集合中的属性必须是连续存放的，也就是说，它们必须是一个数组，还有一个需要注意的就是那些通过调用 SysAllocString 函数得到字符串变量就不需要释放了，这个由 SetProperties 函数在内部负责释放。而不同属性集合的属性就可以放到不同的数组中。

在连接到一个数据源时我们还有一种方法就是使用连接字符串，这也是很多使用 ADO 的程序员常用的一种方式，在 OLEDB 中这可以通过 IdataInitialize 接口的 GetDataSource 方法来实现，例子如下：

IDataInitialize* pIDataInitialize = NULL;

```
CoCreateInstance(CLSID_MSDAINITIALIZE, NULL, CLSCTX_INPROC_SERVER,

    IID_IDataInitialize,(void**)&pIDataInitialize);

IDBInitialize *pIDBInitialize = NULL;

//使用连接字符串得到一个 IDBInitialize 接口

pIDataInitialize->GetDataSource(NULL,

CLSCTX_INPROC_SERVER,

L"Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data Source=D:\\My Document\\db1.mdb;
Mode=Share Deny None;", __uuidof(IDBInitialize),

IUnknown**)&pIDBInitialize);

    //连接到数据库

    pIDBInitialize ->Initialize();
```

上回书说到（哎哟！谁扔的臭鸡蛋？不好意思忘了我是搞 IT 的不是说书的了。）在前面我们已经介绍了如何创建一个连接对象（记住叫 IDBInitialize，而不是别的什么东西），接下来我们就需要用这个连接对象来创建一个叫做事务的对象了，搞数据库的都知道什么叫事务我就不多说了，这个地方只是强调下一个连接对象可以创建多个事务对象，这一点大家要记清楚，在 ATL 的 OLEDB 封装中将连接对象和事务对象被一对一的封装到了一起，统一被称作数据源对象，在一个大型的复杂的数据库应用系统中，我觉得这显然是不够的，必须要针对具体的业务对事务进行一个划分，并为每一类事务创建一个事务对象。

最常用的一个事务对象接口是 IOpenRowset，顾名思义，这个接口就是用来打开一个一个结果集的，这个接口的名字就比较好记忆了，同时也比较好理解，下面的例子代码显示了如何从一个数据连接对象创建出一个 IOpenRowset 接口，注意我这个地方说的是接口，而没有说是对象，例子之后将详细解释这是为什么。

```
IOpenRowset* pIOpenRowSet = NULL;

IDBCreateSession * pIDBCreateSession = NULL;

//首先从连接对象接口找到等价的 CreateSession 接口

pIDBInitialize->QueryInterface(

        IID_IDBCreateSession, (void**)&pIDBCreateSession));

//创建一个 IOpenRowset 接口
```

```
pIDBCreateSession->CreateSession(

                NULL,

                IID_IOpenRowset,

                (Iunknown**)&pIOpenRowSet));

//与 IDBInitialize 等价的 IDBCreateSession 可以释放了，需要时再 Query 出来就行了

if( pIDBCreateSession )

{

    pIDBCreateSession->Release();

}
```

上面的代码非常简单，其实实际使用中也就这么简单，但是作为一个比较完整的介绍，这个地方我还要告诉大家一些重要的秘密，首先让我们回到例子前的那个问题，就是为什么我说这仅是一个接口，而不说是对象呢？

这是因为，其实一个 Session 对象是有很多的与 IOpenRowset 等价的接口来表现的，懂 COM 的立刻就会知道，这些接口间是可以相互随意 QueryInterface 的，也就是说拿到了一个接口就等于拿到了其他所有接口，而这些接口全体才完整的表现了 Session 对象，所以我只说这是一个接口，而不是对象。

其实前面的连接对象的接口 IDBInitialize，也是有很多等价的接口的，比如我们在例子中看到的 IDB CreateSession 接口就是它的一个等价接口，下面我就把分别表现连接对象和事务对象的等价接口族列出来，供大家方便的查询。

首先来看看表现数据连接对象的等价接口族（这个列表来源于 MSDN）：

```
CoType  TDataSource

{

    [mandatory]     interface  IDBCreateSession;

    [mandatory]     interface  IDBInitialize;

    [mandatory]     interface  IDBProperties;

    [mandatory]     interface  IPersist;

    [optional]      interface  IConnectionPointContainer;

    [optional]      interface  IDBAsynchStatus;

    [optional]      interface  IDBDataSourceAdmin;

    [optional]      interface  IDBInfo;
```

```
        [optional]      interface  IPersistFile;

        [optional]      interface  ISupportErrorInfo;

    }
```

Mandatory 这个单词的意思是强制的，强制的意思是说，这样的接口是 OLEDB 提供者必须实现的接口，你调用 QueryInterface 一定可以查到这些接口，从功能上说，这也定义了一个 OLEDB 数据源提供者提供的数据连接对象的基本功能。如果这些强制的基本接口都无法顺利的 Query 到，那么说明这个 OLEDB 提供者的接口是不完整的，或者说是不符合 OLEDB 接口规范的。而目前大多数的 OLEDB 提供程序都已经完整的实现了这些基本的接口。

而相应的 optional 则说明这些接口是可选的接口，或者说是扩展功能的接口，如果 QueryInterface 查询不到，这没什么奇怪的，说明 OLEDB 提供程序没有提供这个扩展的接口。无论怎样在 QueryInterface 互相查找这些接口时最好判断下返回值，以使程序有较强的容错性。

也许你会很奇怪，为什么一个对象会被划分为这么多复杂的接口呢？其实这也是 COM 所要求的基本原则，接口的功能越单一越清晰越好，这样无论是接口的实现者，还是接口的调用者都将保持最大的灵活性（因为将来甚至可以很方便的扩展其它接口而不用修改已有的任何代码），同时在某些情况下还可以保持代码的清晰简洁性。比如在我们最开始几个例子中，我们就只用到了 IDBInitialize 和 IDBProperties 两个接口，而且代码是非常的清晰简单的，这对一个大型的软件项目来说是非常重要的（我始终认为，大型复杂项目并不意味着代码就一定要复杂，相反无论是代码还是结构等等方面都要保持简单性原则，即越简单越好）。

对于以上的每个接口的详细功能和使用方法，我将在以后的提高部分中详细介绍，这里先简单的例举下，让大家有个映像，关键是要理解一个对象是可以有 n 个接口的（n >= 1）,这对最终理解和使用好 OLEDB 是至关重要的。

关于等价接口的深层次含义还有一个要注意的问题就是，我们可以创建这些接口中的任意一个作为我们要创建对象的第一个接口，比如，我们可以直接创建 IDBProperties 接口，再来 QueryInterface 出 ID

BInitialize 接口，这是绝对被允许的，所以这也是我说这些接口等价的深层原因，其实只要你创建一个对象的任何一个接口，那么这个对象其实都已被成功创建了，而该对象所有的接口其实都变成了可用，因此第一次被创建哪个接口其实完全是一个习惯问题，比如通常的我们创建一个数据连接对象都是先创建 IDBInitialize 接口，再 Query 出 IDBProperties 接口，设置一堆连接属性后，再调用 IDBInitialize 的 Initialize 方法连接到数据源，其实这完全可以颠倒过来，我们可以先创建 IDBProperties 接口，设置一堆连接属性后，再 Query 出 IDBInitialize 接口，然后在 Initialize 连接到一个指定的数据源。下面的例子代码片段将加深你对这一概念的理解：

```
IDBProperties * pIDBProperties = NULL;

IDBInitialize * pIDBInitialize = NULL;

//1、直接创建 IDBProperties 接口

CoCreateInstance(CLSID_MSDASQL,

NULL,CLSCTX_INPROC_SERVER,IID_IDBProperties,(void **) &pIDBProperties );

//2、设置连接属性，此处省略了若干步骤，请参看前文的例子

InitProperties[0].dwPropertyID = DBPROP_INIT_DATASOURCE;

InitProperties[0].vValue.vt = VT_BSTR;

InitProperties[0].vValue.bstrVal= SysAllocString(L"(local)");
```

```
InitProperties[0].dwOptions = DBPROPOPTIONS_REQUIRED;

InitProperties[0].colid = DB_NULLID;

//指定数据库名

InitProperties[1].dwPropertyID = DBPROP_INIT_CATALOG;

InitProperties[1].vValue.vt = VT_BSTR;

InitProperties[1].vValue.bstrVal = SysAllocString(L"MyTest");

InitProperties[1].dwOptions = DBPROPOPTIONS_REQUIRED;

InitProperties[1].colid = DB_NULLID;

//指定身份验证方式为集成安全模式“SSPI”

InitProperties[2].dwPropertyID = DBPROP_AUTH_INTEGRATED;

InitProperties[2].vValue.vt = VT_BSTR;

InitProperties[2].vValue.bstrVal = SysAllocString(L"SSPI");
```

```
InitProperties[2].dwOptions = DBPROPOPTIONS_REQUIRED;

InitProperties[2].colid = DB_NULLID;

//创建一个 GUID 为 DBPROPSET_DBINIT 的属性集合，这也是初始化连接时需要的唯一一//个属性集合

rgInitPropSet[0].guidPropertySet = DBPROPSET_DBINIT;

rgInitPropSet[0].cProperties = 3;

rgInitPropSet[0].rgProperties = InitProperties;

//3、设置属性

pIDBProperties->SetProperties( 1, (DBPROPSET*)&rgInitPropSet);

    CHECK_HR( hr, OpenSessionCleanup );

//4、Query 出 IDBInitialize 接口

pIDBProperties->QueryInterface( IID_IDBInitialize,

(void**) &pIDBInitialize );
```

//5、调用 Initialize 连接到数据库

```
pIDBInitialize->Initialize();
```

其次让我们再来看看 Session 对象都有些什么接口吧：

```
CoType  TSession

{

    [mandatory]     interface  IGetDataSource;

    [mandatory]     interface  IOpenRowset;

    [mandatory]     interface  ISessionProperties;

    [optional]      interface  IAlterIndex;

    [optional]      interface  IAlterTable;

    [optional]      interface  IBindResource;
```

```
[optional]      interface  ICreateRow;

[optional]      interface  IDBCreateCommand;

[optional]      interface  IDBSchemaRowset;

[optional]      interface  IIndexDefinition;

[optional]      interface  ISupportErrorInfo;

[optional]      interface  ITableCreation;

[optional]      interface  ITableDefinition;

[optional]      interface  ITableDefinitionWithConstraints;

[optional]      interface  ITransaction;

[optional]      interface  ITransactionJoin;

[optional]      interface  ITransactionLocal;

[optional]      interface  ITransactionObject;
```

}

从表中就可以看出，Session 对象其实只有 3 个强制的基本接口，其他的都是扩展的接口，这里还需要注意的几个接口就是 ITransaction（事务）的一系列接口，这些接口对于大型的分布式应用是非常重要的，尤其是 ITransactionJoin 接口更是分布式事务的关键接口，如果 OLEDB 提供者没有实现这个接口，那么就说明这个数据源是不支持分布式事务的。

在本章开始的创建简单 Session 接口的例子中我们通过创建 IOpenRowset 接口的方法创建了一个 Session 对象，其它的 Session 对象接口就可以通过它 QueryInterface 来得到。

特别要注意的是，Session 对象有个 IGetDataSource 接口，可以通过 Session 对象找到原初的数据连接对象，有些时候为了代码的简洁性我们可以在创建了 IOpenRowset 接口之后丢掉之前创建的所有的接口，甚至是数据连接对象的接口，而通过 IGetDataSource 接口就可以方便的找回数据连接对象的接口，下面的例子代码演示了如何找回数据连接对象的接口：

```
IGetDataSource* pIGetDataSource = NULL;

IDBInitialize * pIDBInitialize = NULL;

pIOpenRowset->QueryInterface( IID_IGetDataSource,

(void**) &pIGetDataSource );

pIGetDataSource-> GetDataSource (

IID_IDBInitialize,
```

(IUnknown**)&pIDBInitialize);

像数据连接对象一样，Session 对象也是有很多的属性的，这可以通过 Query 出 ISessionProperties 再创建一个想要的属性集，然后设置的方法来实现，下面的例子演示了如何设置 Session 对象的属性：

//注意 Session 对象只有一个属性集合的一个属性，就是并发级别

//并发级别是控制数据库并发操作的关键在 OLEDB 中它是一系列值的位或结果

//属性：

SessionProperties[0].dwPropertyID = DBPROP_SESS_AUTOCOMMITISOLEVELS;

SessionProperties[0].vValue.vt = VT_I4;

SessionProperties[0].vValue.lVal= DBPROPVAL_TI_READCOMMITTED;

SessionProperties[0].dwOptions = DBPROPOPTIONS_REQUIRED;

SessionProperties[0].colid = DB_NULLID;

//创建一个 GUID 为 DBPROPSET_SESSION 的属性集合

```
//属性集合：

rgSessionPropSet[0].guidPropertySet = DBPROPSET_SESSION;

rgSessionPropSet[0].cProperties = 1;

rgSessionPropSet[0].rgProperties = SessionProperties;



ISessionProperties* pISessionProperties = NULL;

//找到接口

pIOpenRowset->QueryInterface( IID_ISessionProperties,

(void**) &pISessionProperties );

    //设置属性

pISessionProperties->SetProperties(1,rgSessionPropSet);
```

这样 Session 最关键的属性并发级别就被设定了，这个属性将直接影响后续所有的操作的并发控制级别，当然是指由同一 Session 对象创建出来的其它后续的对象。

在 Session 对象的诸多接口中，如果我们要使用 SQL 语句来操作数据源，那么我们经常使用的一个接口就是 IDBCreateCommand 这个接口，特别注意的是这个接口是一个可选（optional）的接口，而不是强制的，这是因为 OLEDB 不但支持连接到传统的数据库使用 SQL 语句的操作方式，还支持连接到类似网页这样的无结构的数据源的，这也是 OLEDB 接口较其它数据源接口强大的一个方面，OLEDB 的目标就是无论应用系统要使用何种数据（结构化的如数据库，半结构化的如 XML，无结构化的如互联网），它都能轻松胜任，而你无需再去寻找别的接口。因此 IDBCreateCommand 接口就被设置为了一个可选的属性，当然大多数情况下我们都是用 OLEDB 来处理数据库型的数据源，而这些数据源的大多数都提供了这个接口，因此我们不必担心这个接口不可用，从而导致 SQL 语言不能用的问题。

Session 对象的基础知识就介绍到这，下一回我将继续介绍 OLEDB 中我们经常应用的焦点：Command 对象和 Rowset 对象。

接下来我们详细的讨论有关命令（Command）对象的各个接口。通过前面的两篇文章，大家应该已经知道一些基础知识：属性集合，属性，对象，接口，如何打开连接，如何创建事务等等，有了这些基础的概念性的知识，对于理解和应用好 OLEDB 编程接口是非常重要的，对于访问数据源数据的任务来说前面仅仅是开始。对于一个像数据库这类的数据源，操作它最好的方法就是使用 SQL 语句，在 OLEDB 中对执行 SQL 语句提供了完整的支持，实现这一功能最重要的对象就是 Command，首先我们来看看这个对象的接口全貌：

```
CoType  TCommand

{
```

[mandatory] interface IAccessor;

[mandatory] interface IColumnsInfo;

[mandatory] interface ICommand;

[mandatory] interface ICommandProperties;

[mandatory] interface ICommandText;

[mandatory] interface IConvertType;

[optional] interface IColumnsRowset;

[optional] interface ICommandPersist;

[optional] interface ICommandPrepare;

[optional] interface ICommandWithParameters;

[optional] interface ISupportErrorInfo;

[optional] interface ICommandStream;

}

现在只要一看到这张表，应该立刻就清楚其中哪些接口是强制的，哪些是可选的。通常我们使用 IDBCreateCommand 事务接口来创建 Command 对象，当然这个事务对象可以直接从 IDBCreateSession 接口创建得来，下面就让我们看一个相对完整的例子，看看如何从连接数据库开始，直到创建出一个 Command 对象（这是一个非常繁琐的过程，记住每一步是非常重要的）：

```
//1、一大堆头文件和定义，作为前面例子代码的一个相对完整的总结

#define  COM_NO_WINDOWS_H      //如果已经包含了 Windows.h 或不使用其他 Windows

//库函数时

#define  DBINITCONSTANTS

#define  INITGUID

#define  OLEDBVER  0x0250

#include  "oledb.h"              //  OLE  DB  Header

#include  "oledberr.h"          //  OLE  DB  Errors

#include  "msdasc.h"             //  OLE  DB  Service  Component  header
```

```
#include "msdaguid.h"          // OLE DB Root Enumerator

#include "msdasql.h"           // MSDASQL - Default provider

int main()

{

    CoInitialize(NULL);

    //2、创建 IDataInitialize 接口

    IDataInitialize* pIDataInitialize = NULL;

    CoCreateInstance(CLSID_MSDAINITIALIZE, NULL, CLSCTX_INPROC_SERVER,

        IID_IDataInitialize,(void**)&pIDataInitialize);



    IDBProperties *pIDBProperties = NULL;

    IDBInitialize *pIDBInitialize = NULL;
```

```
//2、直接创建 IDBProperties 接口

pIDataInitialize->CreateDBInstance(CLSID_MSDASQL, NULL,

    CLSCTX_INPROC_SERVER, NULL, IID_IDBProperties,

    (IUnknown**)&pIDBProperties);

//3、定义属性集合，属性（略）,设置属性

.....

//4、连接到数据源

pIDBProperties->QueryInterface(IID_IDBInitialize,(void**)&pIDBInitialize);

pIDBInitialize->Initialize();

    //5、得到事务对象

    IDBCreateSession* pIDBCreateSession = NULL;

    pIDBInitialize->QueryInterface(IID_IDBCreateSession,(void**)& pIDBCreateSession);
```

//注意这次直接创建了 IDBCreateCommand 对象作为事务对象，注意一般情况下

//因为这个接口是个可选接口，所以直接创建有可能会失败，所以要检验返回值

//在这一系列文章中我省略这些检验性的操作，在实际的代码中一定要包含丰富的

//错误处理代码，有关错误处理的话题我将在以后的专门文章中详细阐述，这里先//聚焦于我们所关心的问题

```
IDBCreateCommand* pIDBCreateCommand = NULL;

        pIDBCreateSession->CreateSession(NULL,IID_IDBCreateCommand,

(IUnknown**)&pIDBCreateCommand);

        //6、创建 Command 对象，我们直接创建 ICommandText 接口

        ICommandText* pICommandText = NULL;

        pIDBCreateCommand->CreateCommand(NULL,IID_ICommandText,

                (IUnknown**)&pICommandText);

    .....
```

}

上面例子的最后一步我们通过创建一个 ICommandText 接口来创建了一个 Command 对象，在通常的例子中是通过创建 ICommand 接口来创建一个 Command 对象，因为他们是等价的，此处例子中之所以没有这样做，就是为了加深大家对 COM 对象和 COM 接口概念的理解，而这是彻底掌握和理解 OLEDB 接口的基础。

创建 Command 对象的一般目的就是为了能够执行一段 SQL 语句，注意是一段 SQL 语句而不是一句，此处说的一段的含义是，既可以一次执行一句，也可以一次执行多句，甚至是多句 Select 语句。后面的讨论中还将继续深入讨论这个话题，这里我们先来看看 ICommandText 接口，注意在 Command 对象中 ICommandText 接口实际是从 ICommand 接口派生来的，因此 ICommandText 接口实际包含了 ICommand 接口的全部方法：

```
HRESULT SetCommandText(REFGUID rguidDialect,LPCOLESTR pwszCommand);

HRESULT GetCommandText(GUID* pguidDialect,LPOLESTR* ppwszCommand);

//以下的方法实际也是 ICommand 的方法：

HRESULT Cancel();

HRESULT Execute(IUnknown* pUnkOuter,REFIID riid,DBPARAMS* pParams,

    DBROWCOUNT     *pcRowsAffected,IUnknown** ppRowset);

HRESULT GetDBSession(REFIID riid,IUnknown** ppSession);
```

在这些方法中我们实际经常用到的就是 SetCommandText 和 Execute 方法。在 SetCommandText 方法中有一个奇怪的参数 rguidDialect，这参数其实就是为了指定各种 SQL 方言而定的，因为我们知道 SQL Server 的 T-SQL 语句和 ORACLE 的 P-SQL 语句很多细节的语法上是不兼容的，因此不论你怎么使用 OLEDB 接口，都必须了解你所连接的数据库支持的 SQL 语言的一些细节语法（SQL 方言），虽然各大数据库厂商都声称自己的 SQL 语句是如何符合 SQL 语言标准的，但是遗憾的是，它们之间的细节是非常不同的，所以 OLEDB 接口在这里是无能为力的，你必须学好 SQL 语言。而 OLEDB 只不过就是个编程接口而已。

通常情况下，我们不必刻意的去设置一个专有的“SQL 方言”的 GUID，而是直接指定 DBGUID_DEFAULT 常量即可，剩下的语法就交给数据库去处理了。下面的例子显示了如何设置和执行一个 SQL 语句：

```
TCHAR* pSQL = _T("Select * From SomeTable");

pICommandText->SetCommandText(DBGUID_DEFAULT,pSQL);

IRowset* pIRowset = NULL;

pICommandText->Execute(NULL,IID_IRowset,NULL,NULL,(IUnknown**)&pIRowset);
```

上面代码中，通过 Execute 方法我们就得到了一个结果集对象，其初始接口就是 IRowset，关于结果集对象的内容我将放在下一篇文章中，这里我们主要还是聚焦在 Command 对象上。

之前我说过，Command 对象是可以执行一段 SQL 语句的，甚至是多个 Select 语句，也就是说我们可以批量的执行的 SQL 语句。从本质上说执行一条 SQL 语句和执行一批 SQL 语句并没用什么区别，同时很多数据源的 OLEDB 提供者都必须提供批量执行 SQL 语句的功能，这是因为很多复杂的系统中，大多数逻辑的处理都是通过一次执行多条 SQL 语句才能实现的，因此掌握批量执行 SQL 语句的方法是至关重要的。下面的例子代码就演示了如何执行多条 SQL 语句：

```
TCHAR* pSQL = _T("Select * From SomeTable1\nSelect * From SomeTable2");

pICommandText->SetCommandText(DBGUID_DEFAULT,pSQL);

IMultipleResults* pIMultipleResults = NULL;

pICommandText->Execute(NULL,IID_IMultipleResults,NULL,NULL,

(IUnknown**)& pIMultipleResults);
```

上面的例子代码中引入了一个新的对象及其接口就是多结果集（MultipleResults），这个对象比较特殊，它只有这一个接口，通过它可以得到多个结果集对象及其接口（IRowset）。一般情况下，作为自己对 OLEDB 的封装或是应用，都应该使用 IMultipleResults 接口，而不是直接使用 IRowset，这样做的好处就是你既可以一次只执行一条 SQL 语句，也可以一次执行多条 SQL 语句，而不用顾虑其他的问题。下面的代码框架演示了如何利用 IMultipleResults 接口的这一特性：

```
IMultipleResults* pIMultipleResults = NULL;

pICommandText->Execute(NULL,IID_IMultipleResults,NULL,NULL,

(IUnknown**)& pIMultipleResults);

IRowset* pIRowset = NULL;

DBROWCOUNT  cRowsAffected = 0;//注意不要被这个参数迷惑大多数情况下它是没用的
```

```
            //并不能通过它知道结果集中实际包含多少行

//循环处理每一个结果集，当然这需要你起码知道你执行 SQL 语句的顺序

      while( S_OK == pIMultipleResults->GetResult(NULL,

DBRESULTFLAG_DEFAULT,

                        IID_IRowset,

cRowsAffected,

                        (IUnknown**)&pIRowset) )

{

      ......//处理每一个 IRowset

      pIRowset->Release();

      pIRowset = NULL;

}
```

上面的代码就是最一般的一个遍历所有结果集的框架性代码，最好能记住或把它封装的你的代码框架中。

同前面的数据库联接对象和事务对象一样，命令（Command）对象也有自己的属性集，而且这个属性集与前面两个对象的属性集所表示的意义是不同的，前两个对象的属性只影响对象自己，而 Command 对象的属性则会影响因执行 SQL 语句而得到的结果集对象的属性，比如我们最关心的结果集是否是可以修改的，结果集是否是可以插入的，以及结果集是否是可以更新的等等。在这里还需要特别注意的一个问题就是，一个事务对象可以创建 n 个命令对象（n>=1）。下面的例子代码演示了如何设置命令对象（其实是结果集对象）的属性：

```
DBPROPSET  ps[1];

DBPROP  prop[2];

prop[0].dwPropertyID  =  DBPROP_UPDATABILITY;

prop[0].vValue.vt  =  VT_I4;

prop[0].vValue.lVal=DBPROPVAL_UP_CHANGE        //打开 Update 属性

|DBPROPVAL_UP_DELETE                  //打开 Delete 属性

|DBPROPVAL_UP_INSERT;                //打开 Insert 属性

prop[0].dwOptions  =  DBPROPOPTIONS_REQUIRED;
```

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

ps[0].guidPropertySet = DBPROPSET_ROWSET;          //注意属性集合的名称

ps[0].cProperties = 1;

ps[0].rgProperties = prop;

ICommandProperties * pICommandProperties = NULL;

pICommandText->QueryInterface(IID_ICommandProperties,

(void**)& pICommandProperties);

    pICommandProperties->SetProperties(1,ps);//注意必须在 Execute 前设定属性

    IRowset* pIRowset = NULL;

    pICommandText->Execute(NULL,IID_IRowset,NULL,NULL,(IUnknown**)&pIRowset)
```

上面的例子演示了如何打开一个结果集带有更新、插入、删除的属性，这个属性打开后，我们可以直接利用结果集对象的相关方法来修改数据并提交，同时可以绕过使用等价的 SQL 语句（UPDATE、DELETE、INSERT 等），而是使用纯粹代码的方式修改数据并提交到数据库，这是非常棒的一个特性，同时这种方式要比直接使用 SQL 语句的方式高效，具体如何利用这种方式的结果集，我将在后续的提高内容中详细阐述。

作为命令对象的基础性介绍就到这里，下次我将详细介绍结果集对象，以及如何真正拿到我们关心的数据。

在开始新的结果集对象（Rowset）之旅之前，我们再来补充一个关于 Command 对象的用法，在有些情况下，我们执行的 SQL 语句只是一个 Update、Insert 或 Delete 等操作，有些时候我们可能直接执行就是一个存储过程而已，存储过程可能产生也可能不产生一个结果集，这类没有结果集的 SQL 语句，我们可以像下面这样来执行：

```
pICommandText->Execute(NULL,IID_NULL,NULL,NULL,NULL);
```

这样就不用关心 SQL 语句的结果集。当然对于返回结果集的查询也可以这样来执行，即直接丢弃结果集，当然这通常是没有意义的。

下面依照惯例，在开始正式的 Rowset 对象之旅前，让我们首先来认识下 Rowset 对象究竟有哪些接口：

```
CoType  TRowset

{

    [mandatory]     interface  IAccessor;
```

[mandatory] interface IColumnsInfo;

[mandatory] interface IConvertType;

[mandatory] interface IRowset;

[mandatory] interface IRowsetInfo;

[optional] interface IChapteredRowset;

[optional] interface IColumnsInfo2;

[optional] interface IColumnsRowset;

[optional] interface IConnectionPointContainer;

[optional] interface IDBAsynchStatus;

[optional] interface IGetRow;

[optional] interface IRowsetChange;

[optional] interface IRowsetChapterMember;

```
    [optional]      interface  IRowsetCurrentIndex;

    [optional]      interface  IRowsetFind;

    [optional]      interface  IRowsetIdentity;

    [optional]      interface  IRowsetIndex;

    [optional]      interface  IRowsetLocate;

    [optional]      interface  IRowsetRefresh;

    [optional]      interface  IRowsetScroll;

    [optional]      interface  IRowsetUpdate;

    [optional]      interface  IRowsetView;

    [optional]      interface  ISupportErrorInfo;

    [optional]      interface  IRowsetBookmark;

}
```

作为补充，我们同时一看看上一会提到的多结果集对象的接口：

```
CoType  TMultipleResults

{

    [mandatory]     interface  IMultipleResults;

    [optional]      interface  ISupportErrorInfo;

}
```

从 Rowset 对象的接口表中，我们可以知道这个对象都有什么接口，以及那些接口是强制的，那些接口是可选的。在 Rowset 对象中，我们经常用的接口就是 IRowset、IColumnInfo 和 IAccessor，通过这几个接口，才可以访问到我们查询得到的数据结果。同时要注意的就是，最终这个过程并不像前面的接口及方法那么简单，得到可访问的数据就是 OLEDB 中最复杂的一个过程，如果这个过程彻底理解和掌握了，才能说是掌握了 OLEDB 编程的基础，当然不要丧气，在这里我将详细、细致且有条理的介绍这个得到数据的过程：

一、得到结果集：

（请参看（三）文中关于得到结果集的方法示例，此处只简单放一个示意性代码）

```
pICommandText->Execute(NULL,IID_IRowset,NULL,NULL,(IUnknown**)&pIRowset)
```

二、得到列信息：

得到列信息是一个非常重要的过程，通过列信息我们可以知道结果集的完整数据结构，这为后续的创建访问器，准备数据缓冲奠定了基础。很多时候，在 SQL 的查询语句中是没有关于数据结构的信息的，这个信息是隐含在数据库中的，而我们查询得到结果集时就必须要知道这个数据结构的信息，否则对数据的访问将无从谈起。要想得到一个结果集确切的数据结构信息，就要使用 IColumnsInfo 接口，下面的例子演示了如何得到一个结果集完整的列信息，已经如何安全的释放这个列信息：

```
IColumnsInfo * pIColumnsInfo = NULL;

ULONG cColumns = 0;

DBCOLUMNINFO * rgColumnInfo = NULL;

LPWSTR pStringBuffer = NULL;



HRESULT hr = pIRowset->QueryInterface(IID_IColumnsInfo,

(void**)&pIColumnsInfo));



hr = pIColumnsInfo->GetColumnInfo(&cColumns,

&rgColumnInfo,
```

```
                    &pStringBuffer

                    ));

//如果成功了，那么 rgColumnInfo 中已经包含了一个关于列信息数据结构的数组，

//数组元素的个数即 cColumns  也就是最终的列数

//使用完毕后释放所有的资源及接口

CoTaskMemFree(rgColumnInfo);

CoTaskMemFree(pStringBuffer);

if( NULL != pIColumnsInfo )

{

        pIColumnsInfo->Release();

}
```

上面的实例代码显示了如何从一个结果集查询出 IColumnsInfo 接口，以及如何得到列信息的完整过程。在这个过程中特别需要注意的就是最后对 rgColumnInfo 数组和 pStringBuffer 内存的释放，这里使用了 CoTaskMemFree 这个 COM 库函数，通常这样就足够了 ，而在 OLEDB 的帮助文档中则说要使用 IMalloc 接口的 Free 方法来释放，其实二者是等价的。一般的我们使用 CoTaskMemFree 函数即可，特别说明的是调用这个函数释放内存的时候不需要检查被释放的指针是否为空，它内部有检查是否为空的机制，所以直接调用皆可，这又为我们节约了 2-3 行代码。

下面让我们来认识下 DBCOLUMNINFO 这个结构，它的原型如下：

```
typedef  struct  tagDBCOLUMNINFO

{

    LPOLESTR            pwszName;

    ITypeInfo          *pTypeInfo;

    DBORDINAL           iOrdinal;

    DBCOLUMNFLAGS       dwFlags;

    DBLENGTH            ulColumnSize;

    DBTYPE              wType;

    BYTE                bPrecision;
```

```
    BYTE            bScale;

    DBID            columnid;

}

DBCOLUMNINFO;
```

pwszName 字段即为字段的名称，注意是个 UNICODE 字符串。如果查询中没有明确为列指定名称时，这个字段即为空，也就是没有列名。

pTypeInfo 是一个保留待将来使用的接口，直到 MSDAC2.6 版为止这个字段都没有被使用，它永远是 NULL，我们忽略它即可。

iOrdinal 就是字段在结果集中的序号，也就是表示这个列是结果集的第几列，这个字段非常重要。特别注意的是，如果是结果集中的列，那么序号是从 1 开始的。而序号 0 是为一些特殊用途保留的，比如我们打开了结果集的书签功能时，那么序号 0 的列将是结果集的行号。关于书签的内容将在后续的提高部分中介绍。这里大家只要知道 1 对应第一列，n 对应第 n 列即可。

dwFlags 字段描述了列的状态，最重要的状态就是描述该列是否为一个 BLOB 型字段，或者该列是否可为空等。它的类型为 DBCOLUMNFLAGS 是个枚举类型，在 MSDN 中有关于这个枚举类型的详细描述，本文就不在赘述，只在用到时特意说明。

ulColumnSize 字段描述了列的大小，单位为字节。需要注意得是，这个大小仅对字符型列（即 wType = DBTYPE_STR 和 wType = DBTYPE_WSTR）有意义，包括预定义的，或由查询得到的字符串等，而对于其他类型的列该字段则设置为一个~0 的值，即所有 bit 位都为一的值。

wType 则说明了结果集列的数据类型，OLEDB 中定义了完整的被支持的数据类型，此处将所有类型含义及其值列表出来，方便大家查询：

```
enum DBTYPEENUM

{

// The following values exactly match VARENUM

// in Automation and may be used in VARIANT.

    DBTYPE_EMPTY = 0,

    DBTYPE_NULL = 1,

    DBTYPE_I2 = 2,

    DBTYPE_I4 = 3,

    DBTYPE_R4 = 4,

    DBTYPE_R8 = 5,

    DBTYPE_CY = 6,
```

DBTYPE_DATE = 7,

DBTYPE_BSTR = 8,

DBTYPE_IDISPATCH = 9,

DBTYPE_ERROR = 10,

DBTYPE_BOOL = 11,

DBTYPE_VARIANT = 12,

DBTYPE_IUNKNOWN = 13,

DBTYPE_DECIMAL = 14,

DBTYPE_UI1 = 17,

DBTYPE_ARRAY = 0x2000,

DBTYPE_BYREF = 0x4000,

DBTYPE_I1 = 16,

```
    DBTYPE_UI2 = 18,

    DBTYPE_UI4 = 19,

// The following values exactly match VARENUM

// in Automation but cannot be used in VARIANT.

    DBTYPE_I8 = 20,

    DBTYPE_UI8 = 21,

    DBTYPE_GUID = 72,

    DBTYPE_VECTOR = 0x1000,

    DBTYPE_FILETIME = 64,

    DBTYPE_RESERVED = 0x8000,

// The following values are not in VARENUM in OLE.

    DBTYPE_BYTES = 128,
```

```
    DBTYPE_STR = 129,

    DBTYPE_WSTR = 130,

    DBTYPE_NUMERIC = 131,

    DBTYPE_UDT = 132,

    DBTYPE_DBDATE = 133,

    DBTYPE_DBTIME = 134,

    DBTYPE_DBTIMESTAMP = 135

    DBTYPE_HCHAPTER = 136

    DBTYPE_PROPVARIANT = 138,

    DBTYPE_VARNUMERIC = 139

};
```

需要特别注意的是，这里的数据类型不是数据库支持的数据类型，或者说并不是所有的数据库都支持这些所有的数据类型，这仅是一个所有可能数据类型的全部概括，当然也有一些数据库中的类型并不在这个列表中，这时往往数据库对应的 OLEDB 接口提供程序都做了很好的转换，已经转换成了这个列表中所具有的类型，因此不用担心会碰到不支持的数据类型。

bPrecision 和 bScale 字段就不用介绍了，一个是精度，一个是小数位数，他们对于数值型字段是非常重要的。

columnid 字段是该列在数据库系统字典表中的 id 号，这个 id 号也很重要，当我们需要一些关于列的其他更多信息时往往就需要这个 id 号。

至此关于列的信息以及列信息的结构体就介绍完了，也许你会很奇怪，为什么这个东西我会啰嗦这么多，这是因为这对我们后续的操作是非常非常重要的。看了接下来的操作你就知道为什么这个会这么重要了。

三、创建一个绑定：

绑定是 OLEDB 中最重要最核心的概念之一，它是我们访问数据的关键操作，也是比较难理解的一个操作。其实对绑定最简单的理解就是：安排得到数据的内存摆放方式，并将这一方式告诉数据提供者，让它按要求将数据摆放到我们指定的内存中。为了这个目的，我们就需要自己创建一个被称作 DBBINDING 的数组，首先我们来看下这个结构体的样子：

```
typedef  struct  tagDBBINDING

{

    DBORDINAL          iOrdinal;
```

```
DBBYTEOFFSET      obValue;

DBBYTEOFFSET      obLength;

DBBYTEOFFSET      obStatus;

ITypeInfo         *pTypeInfo;

DBOBJECT          *pObject;

DBBINDEXT         *pBindExt;

DBPART            dwPart;

DBMEMOWNER        dwMemOwner;

DBPARAMIO         eParamIO;

DBLENGTH          cbMaxLen;

DWORD             dwFlags;

DBTYPE            wType;
```

```
    BYTE            bPrecision;

    BYTE            bScale;

} DBBINDING;
```

乍一看这个结构和前面的 DBCOLUMNINFO 结构很相似，其实他们的字段大多数确实是一致的，甚至可以直接使用 DBCOLUMNINFO 对 DBBINDING 进行赋值，但是二者字段的含义确是完全不同的，只有理解这二者的差别，才能真正玩转 OLEDB。首先 DBCOLUMNINFO 是数据提供者给你的信息，它是固定的，对相同的查询来说，列总是相同的，因此数据提供者返回的 DBCOLUMNINFO 数组也是固定的。其次 DBBINDING 是你作为数据消费者创建之后给数据提供者的一个结构数组，它的内容则由你来完全控制，通过这个结构我们可以指定数据提供者最终将数据摆放成我们指定的格式，或者进行指定的数据类型转换。

下面的例子展示了如何完成这个过程：

```
ULONG                   cColumns;

DBCOLUMNINFO *          rgColumnInfo            = NULL;

LPWSTR                  pStringBuffer           = NULL;

IColumnsInfo *          pIColumnsInfo           = NULL;

ULONG                   iCol;

ULONG                   dwOffset                = 0;
```

```
DBBINDING *                 rgBindings                      = NULL;

pIRowset->QueryInterface(IID_IColumnsInfo,(void**)&pIColumnsInfo));

pIColumnsInfo->GetColumnInfo(&cColumns, &rgColumnInfo,

                        &pStringBuffer));

rgBindings = (DBBINDING*)HeapAlloc(GetProcessHeap(),

            HEAP_ZERO_MEMORY,

cColumns * sizeof(DBBINDING));




for( iCol = 0; iCol < cColumns; iCol++ )

{

      rgBindings[iCol].iOrdinal     = rgColumnInfo[iCol].iOrdinal;

      rgBindings[iCol].dwPart       = DBPART_VALUE|DBPART_LENGTH|DBPART_STATUS;
```

```
        rgBindings[iCol].obStatus     = dwOffset;

        rgBindings[iCol].obLength     = dwOffset + sizeof(DBSTATUS);

        rgBindings[iCol].obValue      = dwOffset+sizeof(DBSTATUS)+sizeof(ULONG);


rgBindings[iCol].dwMemOwner = DBMEMOWNER_CLIENTOWNED;


            rgBindings[iCol].eParamIO     = DBPARAMIO_NOTPARAM;


rgBindings[iCol].bPrecision = rgColumnInfo[iCol].bPrecision;

        rgBindings[iCol].bScale       = rgColumnInfo[iCol].bScale;
```

```
        rgBindings[iCol].wType          = rgColumnInfo[iCol].wType;

            rgBindings[iCol].cbMaxLen    = rgColumnInfo[iCol].ulColumnSize;

        dwOffset = rgBindings[iCol].cbMaxLen + rgBindings[iCol].obValue;

        dwOffset = ROUNDUP(dwOffset);

    }

    CoTaskMemFree(rgColumnInfo);

    CoTaskMemFree(pStringBuffer);

    if( pIColumnsInfo )

    {

        pIColumnsInfo->Release();

    }
```

在上面的例子代码中，主体循环中演示了如何遍历得到的列信息数组，以及如何使用这个列信息创建一个 DBBINDING 数组。

在代码中，要特别留意的就是下面这四行：

```
rgBindings[iCol].dwPart = DBPART_VALUE|DBPART_LENGTH|DBPART_STATUS;

rgBindings[iCol].obStatus = dwOffset;

rgBindings[iCol].obLength = dwOffset + sizeof(DBSTATUS);

rgBindings[iCol].obValue = dwOffset+sizeof(DBSTATUS)+sizeof(ULONG);
```

其中第一句指明数据提供者最终提交数据时必须包含的信息，这里指定了列值（DBPART_VALUE），长度（DBPART_LENGTH）和状态（DBPART_STATUS）共 3 个信息，列值不用说就是要放字段的最终结果值，长度就是这个字段占用的字节长度，列状态中将存放的是值的状态比如是否为空等。

后面 3 句则明确的指出了以上三个信息在内存中摆放的偏移位置，通常按习惯，我们要求数据提供者先摆放状态，再摆放长度，最后摆放数据。

循环结束后 dwOffset 的值就是一行记录数据所需要的内存的大小。

至此一个简单的 DBBINDING 就创建好了。当然实际中创建这个结构数组时可不是这么简单就行了，还需要考虑很多问题，比如 BLOB 型的字段如何处理等等。作为基础的了解上面的例子已经足够了，关于 DBBINDING 更高级的内容我将放在后续的高级内容中。这里先快速给大家打个基础。

四、创建访问器：

再有了绑定结构之后，接下来要做的工作就是通知数据提供者按我们的要求将数据进行一个“格式化”，这个过程就是创建一个访问器。创建访问器就要使用 IAccessor 接口，同样这个接口也是从 IRowset 查询的来，代表访问器的标志则是一个类型为 HACCESSOR 的句柄。

下面的例子代码演示了如何创建一个访问器：

```
HACCESSOR *                 phAccessor,

IAccessor *                 pIAccessor                  = NULL;

pIRowset->QueryInterface(IID_IAccessor,(void**)&pIAccessor));

pIAccessor->CreateAccessor( DBACCESSOR_ROWDATA,

                    cColumns,rgBindings,0,phAccessor,NULL));

    if( pIAccessor )

{

    pIAccessor->Release();
```

}

上面的例子代码中直接使用了前一个例子中创建的绑定结构，这段例子代码中唯一需要注意的就是最后我们直接释放了 IAccessor 接口，这也表示创建了访问器后这个接口就没有用了，如果需要我们其实也可以随时从同一 IRowset 接口再查询出这个接口即可，因为这两个接口表示的结果集对象是同一个。

至此数据提供者也知道了我们要以什么具体的内存格式来得到数据，接下去就是真正的得到数据了。

五、得到数据：

有了前面 4 步的基础工作，最后我们终于可以正式的得到我们的数据了，为了描述的连贯性这里先将例子代码写出来：

```
void *          pData           = NULL;

ULONG           cRowsObtained;

HROW *          rghRows         = NULL;

ULONG           iRow;

LONG            cRows           = 10;//一次读取 10 行

void *          pCurData;
```

```
//分配 cRows 行数据的缓冲，然后反复读取 cRows 行到这里，然后逐行处理之

pData = HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY, dwOffset * cRows);



while( S_OK == pIRowset->GetNextRows(DB_NULL_HCHAPTER,

                    0,cRows,&cRowsObtained, &rghRows)) )

{//循环读取数据，每次循环默认读取 cRows 行，实际读取到 cRowsObtained 行

    for( iRow = 0; iRow < cRowsObtained; iRow++ )

    {

        pCurData      = (BYTE*)pData + (dwOffset * iRow);

        pIRowset->GetData(   rghRows[iRow],hAccessor,pCurData));

        //pCurData 中已经包含了结果数据，显示或者进行处理
```

```
        ......

    }

    if( cRowsObtained )

    {//释放行句柄数组

pIRowset->ReleaseRows(cRowsObtained,rghRows,NULL,

                          NULL,NULL));

    }

        CoTaskMemFree(rghRows);

    rghRows = NULL;

}
```

```
HeapFree(GetProcessHeap(),0,pData);

if( pIRowset )

{

    pIRowset->Release();

}
```

上面的例子代码中也直接使用了第三步和第四步中的一些变量，大家要注意每个变量的含义，dwOffset 实际在这里已经表示一行数据需要的内存大小。

在这里又多了一个类型 HROW，这个类型表示行的句柄，这里使用的是 HROW 的数组，读出多少行，数组中就有多少行的句柄，然后在使用句柄调用 GetData 将数据读到指定的内存位置。最后每次都释放了这个数组，因为数据已经读进了我们分配的缓存中，这个句柄数组也就变得没有意义了。最终数据读取完毕，我们也就释放了 IRowset 接口。

至此进过 5 个复杂而难于理解的步骤，我们终于读到了我们想要的数据，每一行数据都在那个 pCurData 中，并且严格按照我们指定的 DBBINDING 结构中的几个 Offset 偏移地址摆放，处理行数据内存时，也就按照这几个偏移就的得了具体的每一行每一个列的值。

大体上如何最终得到行数据的基础性内容也就介绍完了，整个的 OLEDB 基础性的知识也就介绍完了，之后的后续文章中我将继续介绍一些更加细节性的问题，或者是更加深入和专题化的问题，也就是提高部分的内容，在继续后续内容之前，也希望大家对之前的所有的内容都有所了解和掌握。

不管怎么说 OLEDB 接口虽然功能强大，但其复杂性也是让人望而却步的。通过前面 4 篇文章的介绍，也只是很基础的内容，同时限于本人的知识能力水平，错漏之处也在所难免，在此也恳请大家对这一系列文章提出宝贵的意见或问题，以便我之后改进提高。最终的目的无非就是让大家都真正的掌握 OLEDB 这个“神兵利器”，做到“攻无不克，战无不胜！”。

前面的系列文章只是使用 OLEDB 的基础，这一章中我们将重点讨论对于大二进制数据类型的访问，比如 SQL Server 中的 TEXT 型、Image 型还有 XML 类型等。

现在将各种文件图片甚至音像视频文件存入数据库中，并统一进行管理已经不是什么稀奇的事情了，各种开发工具环境都提供了对这些大二进制对象存取的完备支持，在 OLEDB 中也有对这些数据的完整支持，当然理解起来和使用起来不是那么方便，但它绝对是最底层的接口，执行效率依然是我们选择 OLEDB 的首要原因。

在 OLEDB 中对这类数据有专门的称呼 BLOB（Binary Large Objects），其他大多数工具中也这样称呼这些数据。实际上如果理解了数据库存放此类数据的具体方式，那么理解对它的访问也比较简单，BLOB 型数据与其他的数据类型唯一不同的地方就是绑定和获取数据方式的不同。

首先在绑定时，需要判定结果集中对应的列是否是 BLOB 型，这可以通过判定 DBCOLUMNINFO 中的 wType 或 dwFlags 标志来判定，具体的可以写如下的判定：

```
pColInfo[iCol].wType == DBTYPE_IUNKNOWN ||

(pColInfo[iCol].dwFlags & DBCOLUMNFLAGS_ISLONG)
```

当这里的两个具体条件之一成立时，我们就可以认为当前这个列是 BLOB 型的。

其次对于此类列数据进行访问时，需要使用一个称为 ISequentialStream 的接口，这里要注意的是这不是一个 OLEDB 特有接口，而是一个 COM 规范的接口，专门用于流式数据访问的接口，它非常的简单，只有两个方法：

```
HRESULT Read(void      *pv,
```

ULONG cb,

ULONG *pcbRead);

HRESULT Write(void const * pv,

ULONG cb,

ULONG * pcbWritten);

这两个方法都非常简单，一看他们的参数类型已经可以猜到如何调用它们了。

当然说到这里你肯定很奇怪，其它数据类型不就是按照指定的长度准备一块内存，然后让 OLEDB 接口把数据放到我们指定的内存处就 ok 了吗？为什么 BLOB 型非要这些奇怪的 Read 和 Write 方法才能读取和存储呢？其实 BLOB 型的数据在实际的数据库（DBMS）系统中存储的方式与其它的简单类型是非常的不同的，最大的不同就是它们不是被直接放入到“行记录”中，通常对它们的存储使用的是一种相对存储技术，典型的就是在真正的行记录中对应的 BLOB 字段处只放一个“指针”，而真正的 BLOB 数据放在表数据的后边，或者专门的磁盘扇区处。注意这里的指针不是内存指针，而是硬盘扇区的指针，通常它是一个 64 bit 值或更长的一个值，因为当前几百个 G 的硬盘已经不是什么稀奇的东西了，索引它上面的地址必须要用更多的 bit。这样存放 BLOB 数据的根本原因其实是因为此类数据通常是长度不固定的，而且行与行之间的同列数据的尺寸差异非常之大，也可能为空，也可能只有几 K，还有可能是几个 G，因此为它们安排统一大小的行记录位置显然是非常的不明智的，所以就采用了上面的这种策略。

对于 OLEDB 来说，对这类数据的访问因为这种存放方式的原因，也需要进行一些特殊的处理。明显的就是无法准确的知道这个数据的实际长度。在使用时，这就需要你具有非常棒的动态内存管理技巧，不然很容易在这各地方造成内存访问异常或者可怕的内存泄露问题。

那么先来看看绑定时我们究竟要做什么不同的工作，示例代码如下：

pBindings[iCol].pBindExt = NULL;

pBindings[iCol].dwFlags = 0;

pBindings[iCol].bPrecision = 0;

pBindings[iCol].bScale = 0;

pBindings[iCol].wType = DBTYPE_IUNKNOWN;

pBindings[iCol].cbMaxLen = 0;

pBindings[iCol].pObject = （DBOBJECT*）HeapAlloc（GetProcessHeap（）,0,sizeof(DBOBJECT));

pBindings[iCol].pObject->iid = IID_ISequentialStream;

pBindings[iCol].pObject->dwFlags = STGM_READ;

在上面的例子中有几个明显不同的地方，首先就是 wType 也就是我们指定的数据类型，是一个 IUNKNOWN 接口类型，其实这是一个暗示方式，就是告诉 OLEDB 的提供者我们将对 BLOB 列采用 COM 流方式来访问，你直接给我一个 COM 接口。

接下来，我们为列的大小显式指定了 0 值，这很重要，不要忽略了，大小指定 0 并不影响我们后续对数据的访问，因为这里我们使用的是“接口”。

紧接着我们设定了 pObject 成员，以及此成员的子成员，尤其注意对 pObject 子成员的赋值。这里对 pObject 使用了一个堆变量，而不是一个栈变量，这样做是因为很多时候我们使用 OLEDB 时绑定操作和实际的访问操作，以及最后的释放操作都不是在同一个函数中，因此需要使用堆变量。

对 pObject 的赋值使用了一个结构叫 DBOBJECT，此结构非常简单原型如下：

typedef struct tagDBOBJECT

{

DWORD dwFlags;

IID iid;

} DBOBJECT;

其中的 dwFlags 成员根据之后的 IID 变量类型指定一组标志，用于控制实际创建的最终的 BLOB 数据访问 COM 接口的行为，目前我们关注的就是 3 个值：STGM_READ，STGM_WRITE，STGM_READWRITE。这几个值一看已经明白它们的具体含义了，我就不再啰嗦了。

iid 成员变量最终指定由 OLEDB 提供者暴露给我们的访问 BLOB 数据的 COM 接口的实际类型，就是我们前面提到的 IID_ISequentialStream 类型。至此绑定也搞定了。

在绑定之后，OLEDB 提供者最终会将一个 COM 接口的指针放在行集中，因为我们已经指定了类型，那么这个指针实际上就是一个 ISequentialStream 的指针。下面的例子代码演示了如何访问这个接口：

//pData 为指向整个行集开始地址的指针

dwStatus = *(DBSTATUS *)((BYTE*)pData + pBindings[iCol].obStatus);

ulLength = *(ULONG *)((BYTE *)pData + pBindings[iCol].obLength);

pvValue = (BYTE *)pData + pBindings[iCol].obValue;

if(DBTYPE_IUNKNOWN == pBindings[iCol].wType)

{

```
ISequentialStream* pISeqStream = *(ISequentialStream**)pvValue;
if(NULL != pISeqStream)
{
const ULONG cBytes = 16 * 4 * 1024; //一次访问 64k
ULONG nBufLen = 0;
ULONG cbRead = 0;
BYTE* pBuffer = (BYTE*)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY,
cBytes);
do
{
hr = pISeqStream->Read(pBuffer + nBufLen,cBytes,&cbRead);
if( 0 < cbRead && SUCCEEDED(hr) )
{
nBufLen += cbRead;
pBuffer = (BYTE*)HeapReAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY,
pBuffer,
nBufLen + cBytes);
}
}while (S_OK == hr);
pISeqStream->Release();
}
```

最终通过上面的访问代码，pBuffer 中就放着对应行的 BLOB 列的数据了。这段代码中使用了 Windows 的堆函数进行了内存重分配操作，省却了一个内存拷贝动作，因为 HeapReAlloc 函数已经帮我们隐含的做了这一切。最后我们立即释放了我们得到的 IsequentialStream 接口的指针，这个是一定要记住的操作，因为根据 OLEDB 的规定，这个接口的生命期完全由消费者控制，因此，即使我们释放了对应的 IRowset 接口或者 IAccess 接口，OLEDB 提供者也不会回收这个 ISequentialStream 接口，这会造成严重的内存泄露问题，最终导致程序崩溃，因此这个步骤非常的重要，一定要像洗脸刷牙一样的记住它，在我们通过 ISequentialStream 接口得到我们想要的数据之后，我们就应该释放这个接口了，当然不一定总是立即释放，你可以在合适的地方释放它。

特别要注意的是以上的方法只是简单的从数据库中读出 BLOB 型数据，而写入这类数据还需要其它的操作，主要用到的接口是 IRowsetChange，关于这个接口我将在后面的文章中介绍，此处先不做讲解，重点还是放在 BLOB 型数据的其他特性上。

通过以上方法我们可以访问一列 BLOB 数据，通常我们还需要访问多列的 BLOB 数据，在一个结果集中，但并不是所有的 OLEDB 提供者都能支持在一行的一个访问器中返回两个以上的 BLOB 型数据（注意说的是同一个访问器，多个访问器是可以访问多个 BLOB 的，如果你还没有晕，就请接着往下看，呵呵呵），具体的只需要我们通过查询一个叫做 DBPROP_MULTIPLESTORAGEOBJECTS 的 Rowset 属性来判定，代码例子如下：

```
IRowsetInfo *          pIRowsetInfo             = NULL;

DBPROPSET *            rgPropSets               = NULL;

DBPROPID               rgPropertyIDs[1];

DBPROPIDSET            rgPropertyIDSets[1];

ULONG                  cPropSets                = 0;


rgPropertyIDs[0] = DBPROP_MULTIPLESTORAGEOBJECTS;

rgPropertyIDSets[0].rgPropertyIDs   =   rgPropertyIDs;

rgPropertyIDSets[0].cPropertyIDs    =   1;

rgPropertyIDSets[0].guidPropertySet = DBPROPSET_ROWSET;


hr = pIRowSet->QueryInterface(IID_IRowsetInfo,

(void**)&pIRowsetInfo);
```

```
hr = pIRowsetInfo->GetProperties(
1,                  //cPropertyIDSets
rgPropertyIDSets,   //rgPropertyIDSets
&cPropSets,         //pcPropSets
&rgPropSets         //prgPropSets
);
if( V_VT(&rgPropSets[0].rgProperties[0].vValue) == VT_BOOL
&& V_BOOL(&rgPropSets[0].rgProperties[0].vValue) == VARIANT_TRUE )
{//支持多个 BLOB 数据
}
```

通过上面的代码最后一个判断我们就可以知道结果集支不支持多个 BLOB 列。目前遗憾的是 SQL Server 系列 DBMS（包括最新的 2008）都不支持多个 BLOB 型数据，这需要我们使用一定的编程策略来绕过这一特性。通常可以使用多访问器 HACCESSOR 多绑定法来实现这样的访问，这是一个非常有用和重要的 OLEDB 实战技巧，因为很多的 OLEDB 提供者都会有不支持多个 BLOB 的情况，因此掌握这个技巧就为访问多个 BLOB 数据提供了方便。下面的示例代码说明了如何访问多个 BLOB 数据在一行中：

```
//定义一个行记录的结构，方便我们后面对数据的访问
struct ROWDATA
{
DBSTATUS dwStatus;
DBLENGTH dwLength;
ISequentialStream* pISeqStream;
};
ROWDATA BLOBGetData0;
ROWDATA BLOBGetData1;
const ULONG cBindings = 1;
//注意我们定义了两个绑定结构
```

```
DBBINDING rgBindings0[cBindings];

DBBINDING rgBindings1[cBindings];

HRESULT hr = S_OK;

IAccessor* pIAccessor = NULL;

ICommandText* pICommandText = NULL;

IRowset* pIRowset = NULL;

DBCOUNTITEM cRowsObtained = 0;

//两个访问器定义

HACCESSOR hAccessor0 = DB_NULL_HACCESSOR;

HACCESSOR hAccessor1 = DB_NULL_HACCESSOR;

//两个绑定对应的两个绑定状态

DBBINDSTATUS rgBindStatus0[cBindings];

DBBINDSTATUS rgBindStatus1[cBindings];

HROW* rghRows = NULL;

const ULONG cPropSets = 1;

DBPROPSET rgPropSets[cPropSets];

const ULONG cProperties = 1;

DBPROP rgProperties[cProperties];

const ULONG cBytes = 10;

//结果数据缓存

BYTE pBuffer[cBytes];

ULONG cBytesRead = 0;

//表 T_Blob 有三列，第一列为 int 型，第二列为 Text 型，第三列为 image 类型

hr = pICommandText->SetCommandText(DBGUID_DBSQL, L"SELECT * FROM T_Blob");
```

```
hr = pICommandText->Execute(NULL,

IID_IRowset,

NULL,

NULL,

(IUnknown**)&pIRowset);

//设置两个绑定结构，注意 iOrdinal 字段序号，一个绑到第 2 列，一个绑到第 3 列

rgBindings0[0].iOrdinal = 2;

rgBindings0[0].obValue = offsetof(BLOBDATA, pISeqStream);

rgBindings0[0].obLength = offsetof(BLOBDATA, dwLength);

rgBindings0[0].obStatus = offsetof(BLOBDATA, dwStatus);

rgBindings0[0].pTypeInfo = NULL;

rgBindings0[0].pObject = &ObjectStruct;

rgBindings0[0].pBindExt = NULL;

rgBindings0[0].dwPart = DBPART_VALUE | DBPART_STATUS | DBPART_LENGTH;

rgBindings0[0].dwMemOwner = DBMEMOWNER_CLIENTOWNED;

rgBindings0[0].eParamIO = DBPARAMIO_NOTPARAM;

rgBindings0[0].cbMaxLen = 0;

rgBindings0[0].dwFlags = 0;

rgBindings0[0].wType = DBTYPE_IUNKNOWN;

rgBindings0[0].bPrecision = 0;

rgBindings0[0].bScale = 0;

//第 3 列

rgBindings1[0].iOrdinal = 3;

rgBindings1[0].obValue = offsetof(BLOBDATA, pISeqStream);
```

```
rgBindings1[0].obLength = offsetof(BLOBDATA, dwLength);

rgBindings1[0].obStatus = offsetof(BLOBDATA, dwStatus);

rgBindings1[0].pTypeInfo = NULL;

rgBindings1[0].pObject = &ObjectStruct;

rgBindings1[0].pBindExt = NULL;

rgBindings1[0].dwPart = DBPART_VALUE | DBPART_STATUS | DBPART_LENGTH;

rgBindings1[0].dwMemOwner = DBMEMOWNER_CLIENTOWNED;

rgBindings1[0].eParamIO = DBPARAMIO_NOTPARAM;

rgBindings1[0].cbMaxLen = 0;

rgBindings1[0].dwFlags = 0;

rgBindings1[0].wType = DBTYPE_IUNKNOWN;

rgBindings1[0].bPrecision = 0;

rgBindings1[0].bScale = 0;


hr = pIRowsetChange->QueryInterface(IID_IAccessor, (void**)&pIAccessor);

//创建第一个访问器

hr = pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,

cBindings,

rgBindings0,

sizeof(BLOBDATA),

&hAccessor0,

rgBindStatus0);

//创建第二个访问器

hr = pIAccessor->CreateAccessor(DBACCESSOR_ROWDATA,
```

```
cBindings,
rgBindings1,
sizeof(BLOBDATA),
&hAccessor1,
rgBindStatus1);
//取得行
hr = pIRowset->GetNextRows(NULL, 0, 1, &cRowsObtained, &rghRows);
//用第一个访问器得到第二列数据
hr = pIRowset->GetData(rghRows[0], hAccessor0, &BLOBGetData0);
if (BLOBGetData0.dwStatus != DBSTATUS_S_ISNULL
&& BLOBGetData0.dwStatus == DBSTATUS_S_OK)
{
BLOBGetData0.pISeqStream->Read( pBuffer, cBytes, &cBytesRead);
//pBuffer 已经有第二列的数据，可以使用前例中的方法得到全部数据
SAFE_RELEASE(BLOBGetData0.pISeqStream);
}
//用第二个访问器得到第三列数据
hr = pIRowset->GetData(rghRows[0], hAccessor1, &BLOBGetData1);
if (BLOBGetData1.dwStatus != DBSTATUS_S_ISNULL
&& BLOBGetData1.dwStatus == DBSTATUS_S_OK )
{
BLOBGetData1.pISeqStream->Read( pBuffer, cBytes, &cBytesRead);
//pBuffer 已经有第三列的数据，可以使用前例中的方法得到全部数据
SAFE_RELEASE(BLOBGetData1.pISeqStream);
```

}

上面的例子代码很好的演示了如何创建多个绑定结构（注意为了简洁明了，没有加入对 hr 的错误判定处理部分，实际编码时注意错误处理），多个访问器，以及如何用多个访问器访问同一行记录中的不同字段，上面例子中尤其要理解和掌握的技巧就是多个 DBBINDING 、HACCESSOR、CreateAccessor 以及多个 GetData 调用的一对一的关系，同时我们对一行记录只调用了一次 GetNextRows，这样就最终实现了一行数据中有多个 BLOB 字段时的数据访问问题。当然在 ADO 当中已经对这个问题进行了很好的封装，使用 ADO 的程序员根本就不知道有这么个限制的存在，看起来用 ADO 的程序员还是很幸运的。

使用下面的 T-SQL 脚本创建例子中的表：

```
CREATE TABLE [T_Blob](
    [K_ID] [int] NOT NULL,
    [BLOB1] [image] NULL,
    [BLOB2] [text] NULL,
CONSTRAINT [PK_T_Blob] PRIMARY KEY CLUSTERED
(
    [K_ID] ASC
)WITH (PAD_INDEX = OFF,

STATISTICS_NORECOMPUTE = OFF,

IGNORE_DUP_KEY = OFF,

ALLOW_ROW_LOCKS = ON,

ALLOW_PAGE_LOCKS = ON)

ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```