用 Visual C++开发数据库应用程序

1、 概述

1、1 Visual C++开发数据库技术的特点

Visual C++提供了多种多样的数据库访问技术——ODBC API、MFC ODBC、DAO、OLE DB、ADO 等。这些技术各有自己的特点，它们提供了简单、灵活、访问速度快、可扩展性好的开发技术。

1. 简单性

   Visual C++中提供了 MFC 类库、ATL 模板类以及 AppWizard、ClassWizard 等一系列的 Wizard 工具用于帮助用户快速的建立自己的应用程序，大大简化了应用程序的设计。使用这些技术，可以使开发者编写很少的代码或不需编写代码就可以开发一个数据库应用程序。

2. 灵活性

   Visual C++提供的开发环境可以使开发者根据自己的需要设计应用程序的界面和功能，而且，Visual C++提供了丰富的类库和方法，可以使开发者根据自己的应用特点进行选择。

3. 访问速度快

   为了解决 ODBC 开发的数据库应用程序访问数据库的速度慢的问题，Visual C++提供了新的访问技术——OLE DB 和 ADO，OLE DB 和 ADO 都是基于 COM 接口的技术，使用这种技术可以直接对数据库的驱动程序进行访问，这大大提供了访问速度。

4. 可扩展性

   Visual C++提供了 OLE 技术和 ActiveX 技术，这种技术可以增强应用程序的能力。使用 OLE 技术和 ActiveX 技术可以使开发者利用 Visual C++中提供的各种组件、控件以及第三方开发者提供的组件来创建自己的程序，从而实现应用程序的组件化。使用这种技术可以使应用程序具有良好的可扩展性。

5. 访问不同种类数据源

   传统的 ODBC 技术只能访问关系型数据库，在 Visual C++中，提供了 OLE DB 访问技术，不仅可以访问关系型数据库，还可以访问非关系型数据库。

1、2 Visual C++开发数据库技术

Visual C++提供了多种访问数据库的技术，如下所示：

1. ODBC（Open DataBase Connectivity）
2. MFC ODBC(Microsoft Foundation Classes ODBC)
3. DAO (Data Access Object)
4. OLE DB(Object Link and Embedding DataBase)
5. ADO(ActiveX Data Object)

这些技术各有自己的特点，总结如下：

1. ODBC

   ODBC 是客户应用程序访问关系数据库时提供的一个统一的接口，对于不同的数据库，ODBC 提供了一套统一的 API，使应用程序可以应用所提供的 API 来访问任何提供了 ODBC 驱动程序的数据库。而且，ODBC 已经成为一种标准，所以，目前所有的关系数据库都提供了 ODBC 驱动程序，这使 ODBC 的应用非常广泛，基本上可用于所有的关系数据库。

   但由于 ODBC 只能用于关系数据库，使得利用 ODBC 很难访问对象数据库及其它非关系数据库。

   由于 ODBC 是一种底层的访问技术，因些，ODBC API 可以使客户应用程序能够从底层设置和控制数据库，完成一些高层数据库技术无法完成的功能。

2. MFC ODBC

   由于直接使用 ODBC API 编写应用程序要编制大量代码，在 Visual C++中提供了 MFC ODBC 类，封装了 ODBC API，这使得利用 MFC 来创建 ODBC 的应用程序非常简便。

3. DAO

   DAO 提供了一种通过程序代码创建和操纵数据库的机制。多个 DAO 构成一个体系结构，在这个结构中，各个 DAO 对象协同工作。MFC DAO 是微软提供的用于访问 Microsoft Jet 数据库文件(*.mdb)的强有力的数据库开发工具，它通过 DAO 的封装，向程序员提供了 DAO 丰富的操作数据库手段。

4. OLE DB

   OLE DB 是 Visual C++开发数据库应用中提供的新技术，它基于 COM 接口。因此，OLE DB 对所有的文件系统包括关系数据库和非关系数据库都提供了统一的接口。这些特性使得 OLE DB 技术比传统的数据库访问技术更加优越。

   与 ODBC 技术相似，OLE DB 属于数据库访问技术中的底层接口。

   直接使用 OLE DB 来设计数据库应用程序需要大量的代码。在 VC 中提供了 ATL 模板，用于设计 OLE DB 数据应用程序和数据提供程序。

5. ADO

ADO 技术是基于 OLE DB 的访问接口，它继承了 OLE DB 技术的优点，并且，ADO 对 OLE DB 的接口作了封装，定义了 ADO 对象，使程序开发得到简化，ADO 技术属于数据库访问的高层接口。

2、 使用 ODBC API

Microsoft 开放数据库互连(ODBC,Open DataBase Connectivity)是 Microsoft Windows 开放服务体系（WOSA）的一部分，是一个数据库访问的标准接口。使用这一标准接口，我们可以不关心具体的数据库管理系统（DBMS）的细节，而只要有相应类型数据库的 ODBC 驱动程序，就可以实现对数据库的访问。

ODBC 编程接口为我们提供了极大的灵活性，我们可以通过这一个接口访问不同种类的数据库。而且，通过相应的 ODBC 驱动程序，我们可以方便地实现不同数据类型之间的转换。

2．1 ODBC API 概述

ODBC 是一个应用广泛的数据库访问应用编程接口（API），使用标准的 SQL（结构化查询语言）作为其数据库访问语言。

2．11 体系结构

ODBC 的结构是建立在客户机 / 服务器体系结构之上，它包含如下四个部分：

应用程序（Application ）：

应用程序即用户的应用，它负责用户与用户接口之间的交互操作，以及调用 ODBC 函数以给出 SQL 请求并提取结果以及进行错误处理。

ODBC 驱动程序管理器（Driver Manager）:

ODBC 驱动程序管理器为应用程序加载和调用驱动程序，它可以同时管理多个应用程序和多个驱动程序。它的功能是通过间接调用函数和使用动态链接库（DLL）来实现的，因此它一般包含在扩展名为”DLL”的文件中。

ODBC 驱动程序（Driver）

ODBC 驱动程序执行 ODBC 函数调用，呈送 SQL 请求给指定的数据源，并将结果返回给应用程序。驱动程序也负责与任何访问数据源的必要软件层进行交互作用，这种层包括与底层网络或文件系统接口的软件。

数据源

数据源由数据集和与其相关联的环境组成，包括操作系统、DBMS 和网络（如果存在的话）。ODBC 通过引入“数据源”的概念解决了网络拓扑结构和主机的大范围差异问题，这样，用户看到的是数据源的名称而不必关心其它东西。

2．12 数据类型

ODBC 使用两类数据类型：SQL 数据类型和 C 数据类型。SQL 数据类型用于数据源，C 数据类型用于应用程序代码中。

2．13 句柄

ODBC API 实现数据库操作的手段是语句，这是一个强有力的手段。ODBC 语句除了能执行 SQL 语句和完成查询操作之外，还能实现大多数数据库操作。

在 ODBC 中，使用不同的句柄（HANDLE）来标志环境(ENVIRONMENT)、连接(CONNECTION)、语句（STATEMENT）、描述器（DESCRIPTOR）等。

句柄就是一个应用程序变量，系统用它来存储关于应用程序的上下文信息和应用程序所用到的一些对象。它和 Windows 编程中的概念类似，不过 ODBC 更加完善了句柄的作用。

1、 环境句柄是 ODBC 中整个上下文的句柄，使用 ODBC 的每个程序从创建环境句柄开始，以释放环境句柄结束。所有其它的句柄（这一应用程序所有的联接句柄和语句句柄）都由环境句柄中的上下文来管理。环境句柄在每个应用程序中只能创建一个。

2、联接句柄管理有关联接的所有信息。联接句柄可以分配多个，这不仅合法而且很有用；但不要生成不必要的句柄以免资源的浪费。但是，不同的驱动程序支持的联接情况有所不同，有的驱动程序在一个应用程序中仅支持一个联接句柄，有的驱动程序仅支持一个语句句柄。在应用程序中，可以在任何适当的时候联接或脱离数据源，但不要轻易地建立或脱离联接。

3、语句句柄是 ODBC API 真正发挥重要作用的，它被用来处理 SQL 语句及目录函数，每个语句句柄只与一个联接有关。当驱动程序接收一个来自应用程序的函数调用指令而该指令包含一个语句句柄时，驱动程序管理器将使用存储在语句句柄中的联接句柄来将这一函数调用发送给合适的驱动程序。

4、描述器句柄是元数据的集合，这些元数据描述了 SQL 语句的参数、记录集的列等信息。当有语句被分配内存之后，描述器自动生成，称为自动分配描述器。在程序中，应用程序也可调用 SQLAllocHandle 分配描述器。

当应用程序调用 API 函数 SQLAllocHandle 时，驱动管理器或者 ODBC 驱动程序将为所声明的句柄类型分配内部结构，返回句柄值。

2．14 异常处理

为了在程序开发过程中调试程序，发现程序错误，ODBC API 通过两种方式返回有关 ODBC API 函数执行的的信息：返回码和诊断记录。返回码返回函数执行的返回值，说明函数执行成功与否。诊断记录说明函数执行的详细信息。

1. 返回码（Return Code）

每一个 ODBC API 函数都返回一个代码——返回码，指示函数执行的成功与否。如果函数调用成功，返回码为 SQL_SUCCESS 或 SQL_SUCCESS_WITH_INFO。SQL_SUCCESS 指示可通过诊断记录获取有关操作的详细信息，SQL_SUCCESS_WITH_INFO 指示应用程序执行结果带有警告信息，可通过诊断记录获取详细的信息。如果函数调用失败，返回码为 SQL_ERROR。

下面的一段代码根据函数 SQLFetch()执行的返回码，判断函数执行的成功与否，从而据此进行相应的处理。

```
SQLRETURN rtcode;

SQLHSTMT hstmt;

While(rtcode=SQLFetch(hstmt)!=SQL_NO_DATA)

{

if(rtcode==SQL_SUCCESS_WITH_INFO)

{

//显示警告信息

}

else

{

//显示出错信息

break;

}

//函数调用成功，进行处理
```

}

如果程序执行错误，返回码为 SQL_INVALID_HANDLE，程序无法执行，而其它的返回码都带有程序执行的信息。

2. 诊断记录(Diagnostic Records)

每个 ODBC API 函数都能够产生一系列的反映操作信息的诊断记录。这些诊断记录放在相关连的 ODBC 句柄中，直到下一个使用同一个句柄的函数调用，该诊断记录一直存在。诊断记录的大小没有限制。
诊断记录有两类：头记录（Head Record）和状态记录（Status Record）。头记录是第一版权法记录(Record 0)，后面的记录为状态记录。诊断记录有许多的域组成，这些域在头记录和状态记录中是不同的。
可以用 SQLGetDiagField 函数获取诊断记录中的特定的域，另外，可以使用 SQLGetDiagRec()获取诊断记录中一些常用的域，如 SQLSTATE、原始错误号等。

1. 头记录

头记录的各个域中包含了一个函数执行的通用信息，无论函数执行成功与否，只要不返回 SQL_INVALID_HANDLE,都会生成头记录。

2. 状态记录

状态记录中的每个域包含了驱动管理器、ODBC 驱动程序或数据源返回的特定的错误或警告信息，包括 SQLSTATE、原始错误码、诊断信息、列号和行号等。只有函数执行返回 SQL_ERROR,SQL_STILL_EXEUTING、SQL_SUCCESS_WITH_INFO、SQL_NEED_DATA 或 SQL_NO_DATA 时，才会生成诊断记录。

3. 使用 SQLGetDiagRec 和 SQLGetDiagField

应用程序可以调用函数 SQLGetDiagRec 或 SQLGetDiagField 获取诊断信息。对于给定的句柄，这两个函数返回最近使用该句柄的函数的诊断信息。当有使用该句柄的函数执行时，句柄记录所记录的原有的诊断信息被覆盖。如果函数执行后产生多个状态记录，程序必须多次调用这两个函数以获取信息。

2．2 应用 ODBC API 建立应用程序

虽然直接应用 ODBC API 编制应用程序相对来说较为繁琐，但是，由于直接使用 ODBC API 编写的程序相对要简洁、高效。所以，我们有必要学习直接使用 ODBC API 编程。

一般地，编写 ODBC 程序主要有以下几个步骤：

1. 分配 ODBC 环境

2. 分配连接句柄
3. 连接数据源
4. 构造和执行 SQL 语句
5. 取得执行结果
6. 断开同数据源的连接
7. 释放 ODBC 环境

2．21 分配 ODBC 环境

对于任何 ODBC 应用程序来说，第一步的工作是装载驱动程序管理器，然后初始化 ODBC 环境，分配环境句柄。

首先，程序中声明一个 SQLHENV 类型的变量，然后调用函数 SQLAllocHandle，向其中传递分配的上述 SQLHENV 类型的变量地址和 SQL_HANDLE_ENV 选项。如下代码所示：

```
SQLHENV henv;
SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&henv);
```

执行该调用语句后，驱动程序分配一个结构，该结构中存放环境信息，然后返回对应于该环境的环境句柄。

2．22 分配连接句柄

分配环境句柄后，在建立至数据源的连接之前，我们必须分配一个连接句柄，每一个到数据源的连接对应于一个连接句柄。

首先，程序定义了一个 SQLHDBC 类型的变量，用于存放连接句柄，然后调用 SQLAllocHandle 函数分配句柄。如下代码所示：

```
SQLHDBC hdbc;
SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
```

henv 为环境句柄。

2．23 连接数据源

当连接句柄分配完成后，我们可以设置连接属性，所有的连接属性都有缺省值，但是我们可以通过调用函数 SQLSetConnectAttr()来设置连接属性。用函数 SQLGetConnectAttr()获取这些连接属性。

函数格式如下：

```
SQLRETURN SQLSetConnectAttr(SQLHDBC ConnectionHandle,SQLINTEGER
Attribute,SQLPOINTER ValuePtr,SQLINTEGER StringLength);

SQLRETURN SQLGetConnectAttr(SQLHDBC ConnectionHandle,SQLINTEGER
Attribute,SQLPOINTER ValuePtr,SQLINTEGER StringLength);
```

应用程序可以根据自己的需要设置不同的连接属性。

完成对连接属性的设置之后，就可以建立到数据源的连接了。对于不同的程序和用户接口，可以用不同的函数建立连接：SQLConnect、SQLDriverConnect、SQLBrowseConnect。

SQLConnect

该函数提供了最为直接的程序控制方式，我们只要提供数据源名称、用户 ID 和口令，就可以进行连接了。

函数格式:

SQLRETURN SQLConnect(SQLHDBC ConnectionHandle,SQLCHAR ServerName,SQLSMALLINT NameLength1,SQLCHAR UserName,SQLSMALLINT NameLength2,SQLCHAR *Authentication,SQLSMALLINT NameLength3);

参数:

ConnectionHandle 连接句柄

ServerName 数据源名称

NameLength1 数据源名称长度

UserName 用户 ID

NameLength2 用户 ID 长度

Authentication 用户口令

NameLength3 用户口令长度

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

下面的代码演示了如何使用 ODBC API 的 SQLConnect 函数建立同数据源 SQLServer 的连接。

#include “sqlext.h”

SQLHENV henv;;

SQLHDBC hdbc;

```
SQLHSTMT hstmt;

SQLRETURN retcode;

/*Allocate environment handle */

retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Set the ODBC version environment attribute */

retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3,
0);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Allocate connection handle */

retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Set login timeout to 5 seconds. */

SQLSetConnectAttr(hdbc, (void*)SQL_LOGIN_TIMEOUT, 5, 0);

/* Connect to data source */

retcode = SQLConnect(hdbc, (SQLCHAR*) "Sales", SQL_NTS,

(SQLCHAR*) "JohnS", SQL_NTS,

(SQLCHAR*) "Sesame", SQL_NTS);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){

/* Allocate statement handle */

retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Process data */;

SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
```

}

SQLDisconnect(hdbc);

}

SQLFreeHandle(SQL_HANDLE_DBC, hdbc);

}

}

SQLFreeHandle(SQL_HANDLE_ENV, henv);

SQLDriveConnect

函数 SQLDriveConnect 用一个连接字符串建立至数据源的连接。它可以提供比 SQLConnect 函数的三个参数更多的信息，可以让用户输入必要的连接信息。

如果连接建立，该函数返回完整的字符串，应用程序可使用该连接字符串建立另外的连接。

函数格式:

SQLRETURN SQLDriverConnect(SQLHDBC ConnectionHandle,SQLHWND WindowHandle,SQLCHAR InConnectionString,SQLSMALLINT StringLength1,SQLCHAR OutConnetionString,SQLSMALLINT BufferLength,SQLSMALLINT *StringLength2Ptr,SQLSMALLINT DriverCompletion);

参数：

ConnectionHandle 连接句柄

WindowHandle 窗口句柄，应用程序可以用父窗口的句柄，或用 NULL 指针

InConnectionString 连接字符串长度

OutConnectionString 一个指向连接字符中的指针

BufferLength 存放连接字符串的缓冲区的长度

StringLength2Ptr 返回的连接字符串中的字符数

DriverCompletion 额外连接信息,可能取值有：SQL_DRIVER_PROMPT,

SQL_DRIVER_COMPLETE,

SQL_DRIVER_COMPLETE_REQUIRED, or

SQL_DRIVER_NOPROMPT.

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

SQLBrowseConnect

函数 SQLBrowseConnect 支持以一种迭代的方式获取到数据源的连接，直到最后建立连接。它是基于客房机/服务器的体系结构，因此，本地数据库不支持该函数。

一般，我们提供部分连接信息，如果足以建立到数据源的连接，则成功建立连接，否则返回 SQL__NEED__DATA，并在 OutConnectionString 参数中返回所需要的信息。

函数格式:

SQLRETURN SQLBrowseConnect(SQLHDBC ConnectionHandle,SQLCHAR* InConnectionString,SQLSAMLLINT StringLength1,SQLCHAR* OutConnectionString,SQLSMALLINT BufferLength,SQLSMALLINT *StringLength2Ptr);

参数：

ConnectionHandle 连接句柄

InConnectionString 指向输出字符串的指针

StringLength1 输出字符串的指针长度

OutConnectionString 指向输出字符串的指针

BufferLength 用于存放输出字符串的缓冲区的长度

StringLength2Ptr 实际返回的字符串的长度

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or

SQL_INVALID_HANDLE.

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

下面的代码讲述了如何使用 ODBC API 的 SQLBrowseConnect 函数建立同数据源的连接。

```
#define BRWS_LEN 100SQLHENV

henv;SQLHDBC hdbc;

SQLHSTMT hstmt;

SQLRETURN retcode;

SQLCHAR szConnStrIn[BRWS_LEN], szConnStrOut[BRWS_LEN];

SQLSMALLINT cbConnStrOut;/* Allocate the environment handle. */

retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Set the version environment attribute. */

retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, SQL_OV_ODBC3, 0);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Allocate the connection handle. */

retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Call SQLBrowseConnect until it returns a value other than */

/* SQL_NEED_DATA (pass the data source name the first time). */

/* If SQL_NEED_DATA is returned, call GetUserInput (not */

/* shown) to build a dialog from the values in szConnStrOut. */
```

```
/* The user-supplied values are returned in szConnStrIn, */

/* which is passed in the next call to SQLBrowseConnect. */

lstrcpy(szConnStrIn, "DSN=Sales"); do {

retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,

szConnStrOut, BRWS_LEN, &cbConnStrOut);

if (retcode == SQL_NEED_DATA)

GetUserInput(szConnStrOut, szConnStrIn);

} while (retcode == SQL_NEED_DATA);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){

/* Allocate the statement handle. */

retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Process data after successful connection */ ...;

SQLFreeHandle(SQL_HANDLE_STMT, hstmt); }

SQLDisconnect(hdbc); } }

SQLFreeHandle(SQL_HANDLE_DBC, hdbc); }}

SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

2．24 SQL 操作

构造和执行 SQL 语句

构造 SQL 语句

可以通过三种方式构造 SQL 语句：在程序开发阶段确定、在运行时确定或由用户输入 SQL 语句。

在程序开发时确定的 SQL 语句，具有易于实现、且可在程序编码时进行测试的优势。

在程序运行时确定 SQL 语句提供了极大灵活性，但给程序高度带来了困难，且需更多的处理时间。由用户输入的 SQL 语句，极大的增强了程序的功能，但是，程序必须提供友好的人机界面，且对用户输入的语句执行一定程序的语法检查，能够报告用户错误。

执行 SQL 语句

应用程序的绝大部分数据库访问工作都是通过执行 SQL 语句完成的，在执行 SQL 语句之前，必须要先分配一个语句句柄，然后设置相应语句的语句属性，再执行 SQL 语句。当一个语句句柄使用完成后，调用函数 SQLFreeHandle()释放该句柄。

1. SQLExecute()

   SQLExecute 用于执行一个准备好的语然，当语句中有参数时，用当前绑定的参数变量的值。

   函数格式：

   SQLRETURN SQLExecute(SQLHSTMT StatementHandle);

   参数:

   StatementHandle 标识执行 SQL 语句的句柄，可以用 SQLAllocHandle()来获得。

   返回值:

   SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE

   成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

2. SQLExecDiret()

   SQLExecDirect 直接执行 SQL 语句，对于只执行一次的操作来说，该函数是速度最快的方法。

   函数格式：

   SQLRETURN SQLExecDirect(SQLHSTMT StatementHandle,SQLCHAR *StatementText,SQLINTEGER TextLength);

参数:

StatementHandle 语句句柄

StatementText 要执行的 SQL 语然

StatementText SQL 语句的长度。

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

3. SQLPripare()

对于需多次执行的 SQL 语句来说，除了使用 SQLExecDirect 函数之外，我们也可以在执行 SQL 语句之前，先准备 SQL 语句的执行。对于使用参数的语句，这可大提高程序执行速度。

函数格式:

SQLRETURN SQLPrepare(SQLHSTMT StatementHandle,SQLCHAR* StatementText,SQLINTEGER TextLength);

参数:

StatementHandle 语句句柄

StatementText 要执行的 SQL 语然

StatementText SQL 语句的长度。

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

使用参数

使用参数可以使一条 SQL 语句多次执行，得到不同结果

SQLBindParameter

函数 SQLBindParameter 负责为参数定义变量，实现参数值的传递。

函数格式如下：

SQLRETURNSQLBindParameter(SQLHSTMT StatementHandle,SQLUSMALLINT ParameterNumber,SQLSMALLINT InputOutputType,SQLSMALLINT ValueType,SQLSMALLINT ParameterType,SQLUINTEGER ColumnSize,SQLSMALLINT DecimalDigits,SQLPOINTER ParameterValuePtr,SQLINTEGER BufferLength,SQLINTEGER

*StrLen_or_IndPtr);
参数：
StatementHandle 语句句柄
ParameterNumber 绑定的参数在 SQL 语句中的序号，在 SQL 中，所有参数从左到右顺序编号，从 1 开始。SQL 语句执行之前，应该为每个参数调用函数 SQLBindParameter 绑定到某个程序变量。
InputOutputType 参数类型，可为 SQL_PARA_INPUT, SQL_PARAM_INPUT_OUTPUT, SQL_PARAM_OUTPUT。
ParameterType 参数数据类型
ColumnSIze 参数大小
DecimalDigits 参数精度
ParameterValutePtr 指向程序中存放参数值的缓冲区的指针
BufferLength 程序中存放参数值的缓冲区的字节数
StrLen_or_IndPtr 指向存放参数 ParameterValuePtr 的缓冲区指针
返回值：
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。
执行时传递参数

对于某些文本文档或位图文件，要占用大量的存储空间。因此，在数据源传递这些数据时，可以分开传递。有两个函数可完成这个工作。

函数格式：
SQLRETURN SQLPutData(SQLHSTMT StatementHandle, SQLPOINTER DataPtr,SQLINTEGER StrLen_or_Ind);
参数：
StatementHandle 参数句柄
DataPtr 指向包含实际数据的缓冲区指针。
StrLen_or_Lnd 缓冲区长度
返回值:
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。
函数格式：
SQLRETURNSQLParamData(SQLHSTMT StatementHandle,SQLPOINTER *ValuePtrPtr);
参数：
StatementHandle 参数句柄
ValuePtrPtr 指向缓冲区地址的指针
返回值：

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。
下面的代码展示如何使用这两个函数

```
#define MAX_DATA_LEN 1024

SQLINTEGER cbPartID = 0, cbPhotoParam, cbData;

SQLUINTEGER sPartID; szPhotoFile;

SQLPOINTER pToken, InitValue;

SQLCHAR Data[MAX_DATA_LEN];

SQLRETURN retcode;

SQLHSTMT hstmt;

retcode = SQLPrepare(hstmt, "INSERT INTO PICTURES (PARTID, PICTURE) VALUES

(?, ?)", SQL_NTS);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Bind the parameters. For parameter 2, pass */

/* the parameter number in ParameterValuePtr instead of a buffer */

/* address. */ SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG,

SQL_INTEGER, 0, 0, &sPartID, 0, &cbPartID);

SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT,

SQL_C_BINARY, SQL_LONGVARBINARY,

0, 0, (SQLPOINTER) 2, 0, &cbPhotoParam);

/* Set values so data for parameter 2 will be */

/* passed at execution. Note that the length parameter in */

/* the macro SQL_LEN_DATA_AT_EXEC is 0. This assumes that */
```

```
/* the driver returns "N" for the SQL_NEED_LONG_DATA_LEN */

/* information type in SQLGetInfo. */

cbPhotoParam = SQL_LEN_DATA_AT_EXEC(0);

sPartID = GetNextID(); /* Get next available employee ID */

/* number. */ retcode = SQLExecute(hstmt);

/* For data-at-execution parameters, call SQLParamData to */

/* get the parameter number set by SQLBindParameter. */

/* Call InitUserData. Call GetUserData and SQLPutData */

/* repeatedly to get and put all data for the parameter. */

/* Call SQLParamData to finish processing this parameter */

while (retcode == SQL_NEED_DATA) {

retcode = SQLParamData(hstmt, &pToken);

if (retcode == SQL_NEED_DATA) {

InitUserData((SQLSMALLINT)pToken, InitValue);

while (GetUserData(InitValue, (SQLSMALLINT)pToken, Data,

&cbData))

SQLPutData(hstmt, Data, cbData); } }}

VOID InitUserData(sParam, InitValue)SQLPOINTER InitValue;{

SQLCHAR szPhotoFile[MAX_FILE_NAME_LEN];

/* Prompt user for bitmap file containing employee */

/* photo. OpenPhotoFile opens the file and returns the */ /* file handle. */

PromptPhotoFileName(szPhotoFile);

OpenPhotoFile(szPhotoFile, (FILE *)InitValue); break; }
```

BOOL GetUserData(InitValue, sParam, Data, cbData)SQLPOINTER InitValue;

SQLCHAR *Data;SQLINTEGER *cbData;BOOL Done;{

/* GetNextPhotoData returns the next piece of photo */

/* data and the number of bytes of data returned */

/* (up to MAX_DATA_LEN). */ Done = GetNextPhotoData((FILE *)InitValue, Data,

MAX_DATA_LEN, &cbData); if (Done) {

ClosePhotoFile((FILE *)InitValue);

return (TRUE); }

return (FALSE); }

记录的添加、删除和更新

应用程序对数据源的数据更新可以通过三种方式实现：使用相应的 SQL 语句在数据源上执行；调用函数 SQLSetPos 实现记录集的定义更新；调用函数 SQLBulkOperations 实现数据的更新。

直接在数据源上执行 SQL 语句的方式，可以适用于任何的 ODBC 数据源，但是，对于后两种更新方式，有的数据源并不支持，应用程序可以调用函数 SQLGetInfo 确定数据源是否支持这两种方式。

1. 定位更新和删除

   要使用定位更新和删除功能，要按如下顺序：

   1)取回记录集：

   2)定位到要进行更新或删除操作的行

   3)执行更新或删除操作

   参考如下的代码：

   #define ROWS 20#define STATUS_LEN 6

   SQLCHAR szStatus[ROWS][STATUS_LEN], szReply[3];

```
SQLINTEGER cbStatus[ROWS], cbOrderID;

SQLUSMALLINT rgfRowStatus[ROWS];

SQLUINTEGER sOrderID, crow = ROWS, irow;

SQLHSTMT hstmtS, hstmtU;

SQLSetStmtAttr(hstmtS, SQL_ATTR_CONCURRENCY, (SQLPOINTER)
SQL_CONCUR_ROWVER, 0);

SQLSetStmtAttr(hstmtS, SQL_ATTR_CURSOR_TYPE, (SQLPOINTER)
SQL_CURSOR_KEYSET_DRIVEN, 0);

SQLSetStmtAttr(hstmtS, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER) ROWS, 0);

SQLSetStmtAttr(hstmtS, SQL_ATTR_ROW_STATUS_PTR, (SQLPOINTER) rgfRowStatus,
0);

SQLSetCursorName(hstmtS, "C1", SQL_NTS);

SQLExecDirect(hstmtS, "SELECT ORDERID, STATUS FROM ORDERS ", SQL_NTS);

SQLBindCol(hstmtS, 1, SQL_C_ULONG, &sOrderID, 0, &cbOrderID);

SQLBindCol(hstmtS, 2, SQL_C_CHAR, szStatus, STATUS_LEN, &cbStatus);

while ((retcode == SQLFetchScroll(hstmtS, SQL_FETCH_NEXT, 0)) != SQL_ERROR) {

if (retcode == SQL_NO_DATA_FOUND) break;

for (irow = 0; irow < crow; irow++) {

if (rgfRowStatus[irow] != SQL_ROW_DELETED)

printf("%2d %5d %*s\n", irow+1, sOrderID, NAME_LEN-1, szStatus[irow]);

} while (TRUE) { printf("\nRow number to update?");

gets(szReply); irow = atoi(szReply);

if (irow > 0 && irow <= crow) { printf("\nNew status?");

gets(szStatus[irow-1]);
```

SQLSetPos(hstmtS, irow, SQL_POSITION, SQL_LOCK_NO_CHANGE);

SQLPrepare(hstmtU,

"UPDATE ORDERS SET STATUS=? WHERE CURRENT OF C1", SQL_NTS);

SQLBindParameter(hstmtU, 1, SQL_PARAM_INPUT,

SQL_C_CHAR, SQL_CHAR,

STATUS_LEN, 0, szStatus[irow], 0, NULL);

SQLExecute(hstmtU); } else if (irow == 0) { break; }

}

}

2. 用 SQLBulkOperations()更新数据

   函数 SQLBulkOperations 的操作是基于当前行集的，在调用函数 SQLBulkOperations 之前，必须先调用函数 SQLFetch 或 SQLFetchScroll 获取行集，然后，再执行修改或删除操作。
   函数格式：

SQLRETURN SQLBulkOperations(SQLHSTMT StatementHandle,

SQLUSMALLINT Operation);

参数：

StatementHandle 参数句柄

Operation 标识执行的操作类型，可以是以下几种之一：

SQL_ADD

SQL_UPDATE_BY_BOOKMARK

SQL_DELETE_BY_BOOKMARK

SQL_FETCH_BY_BOOKMARK

返回值：

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

取回查询结果

绑定列

从数据源取回的数据存放在应用程序定义的变量中，因此，我们必须首先分配与记录集中字段相对应的变量,然后通过函数 SQLBindCol 将记录字段同程序变量绑定在一起。对于长记录字段，可以通过调用函数 SQLGetData()直接取回数据。

绑定字段可以根据自己的需要，全部绑定，也可以绑定其中的某几个字段。

记录集中的字段可以在任何时候绑定，但是，新的绑定只有当下一次从数据源中取数据时执行的操作才有郊，而不会对已经取回的数据产生影响。

函数格式:

SQLRETURN SQLBindCol(SQLHSTMT StatementHandle,

SQLUSMALLINTColumnNumber, SQLSMALLINT TargetType,

SQLPOINTERTargetValuePtr,SQLINTEGERBufferLength,

SQLINTEGER *StrLen_or_IndPtr);

参数：

StatementHandle 语句句柄

ColumnNumber 标识要绑定的列号。数据列号从 0 开始升序排列，其中第 0 列用作书签。如果没有使用书签，则列号从 1 开始。

TargetType 数据类型

TargetValuePtr 绑定到数据字段的缓冲区的地址

BufferLength 缓冲区长度

StrLen_or_IndPtr 指向绑定数据列使用的长度的指针.

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

SQLFetch()

函数 SQLFetch 用于将记录集中的下一行变为当前行，并把所有捆绑过的数据字段的数据拷贝到相应的缓冲区。

函数格式:

SQLRETURN SQLFetch(SQLHSTMT StatementHandle)

参数:

StatementHandle 语句句柄

返回值:

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

成功返回 SQL_SUCCESS，如果返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，可以用函数 SQLGetDiagRec 获取相应 SQLSTATE 的值。

光标

应用程序获取数据是通过光标(Cursor)来实现的，在 ODBC 中，主要有三种类型的光标:

1. 单向光标：

   单向光标只能向前移动，要返回记录集的开始位置，程序必须先关闭光标，再打开光标，它对于只需要浏览一次的应用非常有用，而且效率很高。对于需要光标前后移动的，单向光村不适用。

2. 可滚动光标

   可滚动光标通常基于图形用户界面的程序，用户通过屏幕向前或向后滚动，浏览记录集中的数据。

3. 块光标

   所谓块光标，可以理解为执行多行的光标，它所指向的行称为行集。对于网络环境下的应用，使用块光标可以在一定程序上减轻网络负载。

   要使用块光标，应完成以下工作:

   1）设定行集大小

   2）绑定行集缓冲区

   3）设定语句句柄属性

   4）取行行集结果

   由于块光标返回多行记录，应用程序必须把这多行的数据绑定到某些数据组中，这些数据称为行集缓冲区。绑定方式有两种:列方式和行方式。

4. ODBC 光标库

   有些应用程序不支持可滚动光标和块光标，ODBC SDK 提供了一个光标库(ODBCCR32.DLL),在应用程序中可通过设置连接属性(SQL_STTR_ODBC_CURSORS)激活光标库。

参考如下代码：

```
#define NAME_LEN 50

#define PHONE_LEN 10

SQLCHAR szName[NAME_LEN], szPhone[PHONE_LEN];

SQLINTEGER sCustID, cbName, cbCustID, cbPhone;
```

```
SQLHSTMT hstmt;

SQLRETURN retcode;retcode = SQLExecDirect(hstmt,

"SELECT CUSTID, NAME, PHONE FROM CUSTOMERS ORDER BY 2, 1, 3",

SQL_NTS);

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

/* Bind columns 1, 2, and 3 */

SQLBindCol(hstmt, 1, SQL_C_ULONG, &sCustID, 0, &cbCustID);

SQLBindCol(hstmt, 2, SQL_C_CHAR, szName, NAME_LEN, &cbName);

SQLBindCol(hstmt, 3, SQL_C_CHAR, szPhone, PHONE_LEN, &cbPhone);

/* Fetch and print each row of data. On */

/* an error, display a message and exit. */ while (TRUE) {

retcode = SQLFetch(hstmt);

if (retcode == SQL_ERROR || retcode == SQL_SUCCESS_WITH_INFO) {

show_error(); }

if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){

fprintf(out, "%-*s %-5d %*s", NAME_LEN-1, szName,

sCustID, PHONE_LEN-1, szPhone); } else { break; }

}

}
```

2．25 断开同数据源的连接

当完成对数据库操作后，就可以调用 SQLDisconnect 函数关闭同数据源连接。当该句柄的事务操作还未完成时，应用程序调用 SQLDisconnect,这时，驱动器返回 SQLSTATE 25000，表明此次事务没有变化且连接还打开。

在应用程序调用 SQLDisconnect 之前没有释放与之相连的描述时，当成功的与数据源断开连接后，驱动器自动释放与之相连的参数、描述器等。但当与之相连的某一个参数还在执行异步操作时，SQLDisConnect 返回 SQL_ERROR，且 SQLSTATE 的值置为 HY010.

2．26 释放 ODBC 环境

最后一步就是释放 ODBC 环境参数了。

2．3 ODBC API 编程总结

关系数据库的研究与应用是当今计算机界最活跃的领域之一，各种数据库产品行行色色，各有千秋；这种情况一方面给用户带来了好处，另一方面又给应用程序的移植带来了困难。尤其是在客户机 / 服务器体系结构中，当用户要从客户机端访问不同的服务器，而这些服务器的数据库系统又各不相同，数据库之间的互连访问就成为一个难题，因此，微软公司提出了 ODBC。由于 ODBC 思想上的先进性及其微软公司的开放策略，ODBC 现在已经成为事实上的工业标准，它是目前数据库应用方面很多问题强有力的解决方案，正逐步成为 Windows 平台上的标准接口。

ODBC 是一种用来在相关或不相关的数据库管理系统（DBMS）中存取数据的标准应用程序设计接口（API）。它的基本思想是为用户提供简单、标准、透明、统一的数据库联接的公共编程接口，在各个厂家的支持下能为用户提供一致的应用开发界面，使应用程序独立于数据库产品，实现各种数据库之间的通信。

开发厂商根据 ODBC 的标准去实现底层的驱动程序，它对用户是透明的。

作为一种数据库联接的标准技术，ODBC 有以下几个主要特点：

1 • ODBC 是一种使用 SQL 的程序设计接口；

2 • ODBC 的设计是建立在客户机 / 服务器体系结构基础之上的；

3 • ODBC 使应用程序开发者避免了与数据源联接的复杂性；

4 • ODBC 的结构允许多个应用程序访问多个数据源，即应用程序与数据源的关系是多对多的关系。

3、 使用 MFC 访问 ODBC 数据源

3．1 概述

VisualC++的 MFC 类库定义了几个数据库类。在利用 ODBC 编程时，经常要使用到 CDatabase(数据库类)，CRecordSet(记录集类)和 CRecordView(可视记录集类)。 其中：

CDatabase 类对象提供了对数据源的连接，通过它你可以对数据源进行操作。

CRecordView 类对象能以控制的形式 显示数据库记录。这个视图是直接连到一个 CRecordSet 对象的表视图。

CRecordSet 类对象提供了从数据源 中提取出的记录集。CRecordSet 对象通常用于两种形式： 动态行集（dynasets）和快照集（snapshots）。动态行集能保 持与其他用户所做的更改保持同步。快照集则是数据的一个静态视图。每一种形式在记录集被打开时都提供一组记录，所不同的是，当你在一个动态行集里滚 动到一条记录时，由其他用户或是你应用程序中的其他记录集对该记录所做的更改会相应地显示出来。

Visual C++提供了几种记录集，可以用来定制应用程序的工作方式。查看这些不同选项的最快方式要兼顾速度和特征。你会发现，在很多情况下，如果想添加

特征，就必须付出程序执行速度降低的代价。下面告诉你一些可以自由支配的记录集选项。更重要的是，要告诉你从这个选项可以获得更快的速度还是更多的特征。

1、Snapshot（快照） 这个选项要 Visual C++在一次快照中下载整个查询。换言之，及时快速地给数据库内容拍照，并把它作为未来工作的基础。这种方法有三个缺点。第一，你看不到别人在网络上做的更新，这可能意味着你的决定是建立在老信息的基础上。第二，一次就下载所有这些记录，这意味着在下载期间给网络增加了沉重的负担。第三，记录下载时用户会结束等待，这意味着网络的呼叫性能变得更低。然而这种方法也有两个优点。第一，记录一旦被下载，该工作站所需的网络活动几乎就没有了棗这为其它请求释放了带宽。总之，你会看到网络的吞吐量增大了。第二，因为所有被申请的记录都在用户的机器上，所以用户实际上会得到应用程序更佳的总体性能。你可能想把快照的方法限制在较小的数据库上使用，原因在于快照适用于用户请求信息而不适用于数据编辑会话。

2、Dynaset（动态集） 使用这个选项时，Visual C++创建指向所请求的每个记录的实际指针。另外，只有填充屏幕时实际需要的记录是从服务器上下载来的。这种方法的好处很明显。几乎马上就能在屏幕上看到记录。而且还会看到其它用户对数据库所做的更改。最后，其它用户也会看到你做的更改，因为动态集在你更改记录时被上载到服务器上。很明显，这种方法要求对服务器的实时访问，它减小了网络总吞吐量并降低了应用程序的性能。这个选项适合于创建用户要花费很多时间来编辑数据的应用程序。同时，它也是大型数据库的最佳选择，原因在于只需下载用户实际需要的信息。

3．2 应用 ODBC 编程

可以应用 AppWizard 来建立一个 ODBC 的应用程序框架，也可以直接使用 ODBC 来进行数据库编程，这时，应包括头文件 afxdb.h。

应用 ODBC 编程两个最重要的类是 CDatabase 和 CRecordSet，但在应用程序中，不应直接使用 CRecordSet 类，而必须从 CRecordSet 类产生一个导出类，并添加相应于数据库表中字段的成员变量。随后，重载 CRecordset 类的成员函数 DoFieldExchange，该函数通过使用 RFX 函数完成数据库字段与记录集域数据成员变量的数据交换，RFX 函数同对话框数据交换（DDX）机制相类似，负责完成数据库与成员变量间的数据交换。

下面举例说明在 VisualC++环境中 ODBC 的编程技巧：

3．21 数据库连接

在 CRecordSet 类中定义了一个成员变 量 m_pDatabase:

```
CDatabase *m_pDatabase;
```

它是指向对象数据库类的指针。如果在 CRecordSet 类对象调用 Open()函数之前，将一个已经打开的 CDatabase 类对象指针传给 m_pDatabase，就能共享相同 的 CDatabase 类对象。如：

```
CDatabase m_db;

CRecordSet m_set1,m_set2;

m_db.Open(_T("Super_ES")); // 建 立 ODBC 连 接

m_set1.m_pDatabase=&m_db; //m_set1 复 用 m_db 对 象

m_set2.m_pDatabse=&m_db; // m_set2 复 用 m_db 对 象
```

或如下：

```
Cdatabase db;

db.Open(“Database”); //建立 ODBC 连接

CrecordSet m_set(&db); //构造记录集对象,使数据库指向 db
```

3．22 查询记录

查询记录使用 CRecordSet::Open()和 CRecordSet::Requery()成员函数。在使用 CRecordSet 类对象之前，必须使用 CRecordSet::Open()函数来获得有效的记录集。一旦已经使用过 CRecordSet::Open() 函数，再次查询时就可以应用 CRecordSet::Requery()函数。在调 用 CRecordSet::Open()函数时，如果已经将一个已经打开的 CDatabase 对象指针传给 CRecordSet 类对象的 m_pDatabase 成员变量，则使 用该数据库对象建立 ODBC 连接；否则如果 m_pDatabase 为空指 针，就新建一个 CDatabase 类对象并使其与缺省的数据源 相连，然后进行 CRecordSet 类对象的初始化。缺省数据源 由 GetDefaultConnect()函数获得。你也可以提供你所需要的 SQL 语句，并以它来调用 CRecordSet::Open()函数，例如：

```
m_Set.Open(AFX_DATABASE_USE_DEFAULT,strSQL);
```

如果没有指定参数，程序则使 用缺省的 SQL 语句，即对在 GetDefaultSQL()函数中指定的 SQL 语 句进行操作：

```
CString CTestRecordSet::GetDefaultSQL()

{return _T("[BasicData],[MainSize]");}
```

对于 GetDefaultSQL()函数返回的表名， 对应的缺省操作是 SELECT 语句，即：

SELECT * FROM BasicData,MainSize

查询过程中也可以利用 CRecordSet 的 成员变量 m_strFilter 和 m_strSort 来执行条件查询和结果排序。m_strFilter 为过滤字符串，存放着 SQL 语句中 WHERE 后的条件串；m_strSort 为排序字符串，存放着 SQL 语句中 ORDERBY 后的字符串。 如：

m_Set.m_strFilter="TYPE='电动机'";

m_Set.m_strSort="VOLTAGE";

m_Set.Requery();

对应的 SQL 语句为：

SELECT * FROM BasicData,MainSize

WHERE TYPE='电动机'

ORDER BY VOLTAGE

除了直接赋值给 m_strFilter 以外，还 可以使用参数化。利用参数化可以更直观，更方便地 完成条件查询任务。使用参数化的步骤如下：

(1)．声明参变量：

Cstring p1;

Float p2;

(2)．在构造函数中初始化参变量

p1=_T("");

p2=0.0f;

m_nParams=2;

(3)．将参变量与对应列绑定

pFX->SetFieldType(CFieldExchange::param)

RFX_Text(pFX,_T("P1"),p1);

RFX_Single(pFX,_T("P2"),p2);

完成以上步骤之后就可以利用参变量进行条件查询了：

m_pSet->m_strFilter="TYPE=?ANDVOLTAGE=?";

m_pSet->p1="电动机";

m_pSet->p2=60.0;

m_pSet->Requery();

参变量的值按绑定的顺序替换 查询字串中的“?”适配符。

如果查询的结果是多条记录的 话，可以用 CRecordSet 类的函数 Move()，MoveNext()，MovePrev()，MoveFirst() 和 MoveLast()来移动光标。

3．23 增加记录

增加记录使用 AddNew()函数，要求数据库必须是以允许增加的方式打开：

m_pSet->AddNew(); //在表的末尾增加新记录

m_pSet->SetFieldNull(&(m_pSet->m_type),FALSE);

m_pSet->m_type="电动机";

... //输入新的字段值

m_pSet-> Update(); //将新记录存入数据库

m_pSet->Requery(); //重建记录集

3．24 删除记录

直接使用 Delete()函数，并且在调用 Delete() 函数之后不需调用 Update()函数：

m_pSet->Delete();

if(!m_pSet->IsEOF())

m_pSet->MoveNext();

else

m_pSet->MoveLast();

3．25 修改记录

修改记录使用 Edit()函数：

```
m_pSet->Edit(); //修改当前记录

m_pSet->m_type="发电机"; //修改当前记录字段值

...

m_pSet->Update(); //将修改结果存入数据库

m_pSet->Requery();
```

3．26 统计记录

统计记录用来统计记录集的总数。可以先声明一个 CRecordset 对象 m_pSet。再绑定一个变量 m_lCount，用来统计记录总数。执行如下语句：

```
m_pSet->Open(“Select Count(*) from 表名 where 限定条件”);

RecordCount=m_pSet->m_lCount;

m_pSet->Close();
```

RecordCount 即为要统计的记录数。

或如下：

```
CRecordset m_Set(&db); //db 为 CDatabase 对象

CString strValue;

m_Set.Open(Select count(*) from 表名 where 限定条件”);

m_pSet.GetFieldValue((int)0,strValue);

long count=atol(strValue);

m_set.Close();
```

count 为记录总数。

3．27 执行 SQL 语句

虽然通过 CRecordSet 类，我们可以完成 大多数的查询操作，而且在 CRecordSet::Open()函数中也可以 提供 SQL 语句，但是有的时候我们还想进行一些其他操 作，例如建立新表，删除表，建立新的字段等等，这 时就需要使用到 CDatabase 类的直接执行 SQL 语句的机制。通 过调用 CDatabase::ExecuteSQL()函数来完成 SQL 语句的直接执行：

如下代码所示

```
BOOL CDB::ExecuteSQLAndReportFailure(const CString& strSQL)

{

TRY

{

m_pdb->ExecuteSQL(strSQL); //直接执行 SQL 语句

}

CATCH (CDBException,e)

{

CString strMsg;

strMsg.LoadString(IDS_EXECUTE_SQL_FAILED);

strMsg+=strSQL;

return FALSE;

}

END_CATCH

return TRUE;

}
```

应当指出的是，由于不同 DBMS 提 供的数据操作语句不尽相同，直接执行 SQL 语句可能会破坏软件的 DBMS 无关性，因此在应用中应当慎用此类操作。

3．28 注意

从 CRecordSet 导出的类中如果包含 DataTime 类型的数据，在 VC 中是用 CTime 类型来替代的，这时，构造函数没有赋于缺省值。这时，我们应当手工赋值。如下所示:

```
CTime m_time;

m_time=NULL;
```

3．3 总结

VisualC++中的 ODBC 类库可以帮助程序员完成绝大多数的数据库操作。利用 ODBC 技术使得程序员从具体的 DBMS 中解脱出来，从而极大的减少了软件开发的工作量，缩短开发周期，提高了效率和软件的可靠性。

4、使用 DAO

4．1 概述

Visual C++提供了对 DAO 的封装，MFC DAO 类封装了 DAO（数据库访问对象）的大部分功能，从面 Visual C++程序就可以使用 Visual C++提供的 MFC DAO 类方便的访问 Microsoft Jet 数据库，编制简洁、有 Visaul C++特色的数据库应用程序。

数据库访问对象（DAO）提供了一种通过程序代码创建和操纵数据库的机制。多个 DAO 对象构成一个体系结构，在这个结构里，各个 DAO 对象协同工作。DAO 支持以下四个数据库选项：

1. 打开访问数据库（MDB 文件）——MDB 文件是一个自包含的数据库，它包括查询定义、安全信息、索引、关系，当然还有实际的数据表。用户只须指定 MDB 文件的路径名。
2. 直接打开 ODBC 数据源——这里有一个很重要的限制。不能找开以 Jet 引擎作为驱动程序的 ODBC 数据源；只可以使用具有自己的 ODBC 驱动程序 DLL 的数据源。
3. 用 Jet 引擎找开 ISAM 型（索引顺序访问方法）数据源（包括 dBase,FoxPro,Paradox,Btrieve,Excel 或文本文件）——即使已经设置了 ODBC 数据源，要用 Jet 引擎来访问这些文件类型中的一种，也必须以 ISAM 型数据源的方式来找开文件，而不是以 ODBC 数据源的方式。
4. 给 ACCESS 数据库附加外部表——这实际上是用 DAO 访问 ODBC 数据源的首选方法。首先使用 ACCESS 把 ODBC 表添加到一个 MDB 文件上，然后依照第一选项中介绍的方法用 DAO 找开这个 MDB 文件就可以了。用户也可以用 ACCESS 把 IASM 文件附加到一个 MDB 文件上。

4．2 应用 DAO 编程

4．21 打开数据库

CDaoWorkspace 对象代表一个 DAO Workspace 对象，在 MFC DAO 体系结构中处于最高处，定义了一个用户的同数据库的会话，并包含打开的数据库，负责完成数据库的事务处理。我们可以使用隐含的 workspace 对象。
CDaoDatabase 对象代表了一个到数据库的连接，在 MFC 中，是通过 CDaoDatabase 封装的。
在构造 CDaoDatabase 对象时，有如下两种方法:

1. 创建一个 CDaoDatabase 对象，并向其传递一个指向一个已经找开的 CdaoWorkspace 对象的指针。
2. 创建一个 CDaoDatabase 对象，而不明确地指定使用的 workspace,此时，MFC 将创建一个新的临时的 CDaoWorkspace 对象。

如下代码所示：

```
CDaoDatabase db;
db.Open(“test.mdb”,FALSE,FALSE,_T(“”);
```

其中参数一包括要打开的文件的全路径名。

4．22 查询记录

一个 DAO recordset 对象，代表一个数据记录的集合，该集合是一个库表或者是一个查询的运行结果中的全部记录。CDaoRecorset 对象有三种类型：表、动态集、快照。
通常情况下，我们在应用程序中可以使用 CDaoRecordset 的导出类，这一般是通过 ClassWizard 或 AppWizard 来生成的。但我们也可以直接使用 CDaoRecordset 类生成的对象。此时，我们可以动态地绑定 recordset 对象的数据成员。

如下代码所示：

```
COleVariant var;

long id;

CString str;

CDaoRecordset m_Set(&db);

m_Set.Open(“查询的 SQL 语句”);

while(!m_Set.IsEOF())

{

/*

处理

m_Set.GetFieldValue(“ID”,var);
```

```
id=V_I4(var);

m_Set.GetFieldValue(“Name”,var);

str=var.pbVal;

*/

m_Set.MoveNext();

}

m_Set.Close();
```

4．23 添加记录

添加记录用 AddNew 函数，此时用 SetFieldValue 来进行赋值。

如下代码所示:

```
m_pDaoRecordset->AddNew ();

sprintf(strValue,"%s",>m_UserName );

m_pDaoRecordset->SetFieldValue ("UserName",strValue);

sprintf(strValue,"%d",m_PointId );

m_pDaoRecordset->SetFieldValue ("PointId",strValue);

dataSrc.SetDateTime
(m_UpdateTime .GetYear ),m_UpdateTime .GetMonth ),m_UpdateTime .GetDay (),

m_UpdateTime .GetHour (),m_UpdateTime .GetMinute (),m_UpdateTime .GetSecond ());

valValue=dataSrc;

m_pDaoRecordset->SetFieldValue ("UpdateTime",valValue);

sprintf(strValue,"%f",m_pRecordset->m_OldValue );

m_pDaoRecordset->SetFieldValue ("OldValue",strValue);

sprintf(strValue,"%f",m_pRecordset->m_NewValue );

m_pDaoRecordset->SetFieldValue ("NewValue",strValue);

m_pDaoRecordset->Update ();
```

此时，要注意，日期时间型数据要用 SetDataTime 函数来赋值,这里面要用到 COleVariant 类型数据，具体用法可以参考有关帮助。

4．24 修改记录

修改记录用 Edit()函数，把记录定位到要修改的位置，调用 Edit 函数，修改完成后，调用 Update 函数。

如下代码所示：

```
m_Set.Edit();

m_Set.SetFieldValue(“列名”,”字符串”);

m_Set.Update();
```

4．25 删除记录

删除记录用 Delete()函数，使用后不需调用 Update()函数。

4．26 统计记录

可以使用如下代码来统计记录数：

```
COleVariant varValue;

CDaoRecordset m_Set(&db);

m_Set.Open(dbOpenDynaset,”SQL 语句”);

varValue=m_Set.GetFieldValue(0);

m_lMaxCount=V_I4(&varValue);

m_Set.Close();
```

如果是统计一张表中总记录，可以使用 CDaoTableDef 对象，如下代码所示：

```
CDaoTableDef m_Set(&gUseDB);

Count=m_Set.GetRecordCount();

m_Set.Close();
```

不能用 CDaoRecordset 对象的 GetRecordCount()来取得记录数。

4．3 总结

使用 DAO 技术可以便我们方便的访问 Microsoft Jet 引擎数据库，由于 Microsoft Jet 不支持多线程，因此，必须限制调用到应用程序主线程的所有 DAO。

5 使用 OLE DB

5．1 概述

OLE DB 的存在为用户提供了一种统一的方法来访问所有不同种类的数据源。OLE DB 可以在不同的数据源中进行转换。利用 OLE DB，客户端的开发人员在进行数据访问时只需把精力集中在很少的一些细节上，而不必弄懂大量不同数据库的访问协议。

OLE DB 是一套通过 COM 接口访问数据的 ActiveX 接口。这个 OLE DB 接口相当通用，足以提供一种访问数据的统一手段，而不管存储数据所使用的方法如何。同时，OLE DB 还允许开发人员继续利用基础数据库技术的优点，而不必为了利用这些优点而把数据移出来。

5．2 使用 ATL 使用 OLE DB 数据使用程序

由于直接使用 OLE DB 的对象和接口设计数据库应用程序需要书写大量的代码。为了简化程序设计，Visual C++提供了 ATL 模板用于设计 OLE DB 数据应用程序和数据提供程序。

利用 ATL 模板可以很容易地将 OLE DB 与 MFC 结合起来，使数据库的参数查询等复杂的编程得到简化。MFC 提供的数据库类使 OLE DB 的编程更具有面向对象的特性。Viual C++所提供用于 OLE DB 的 ATL 模板可分为数据提供程序的模板和数据使用程序的模板。

使用 ATL 模板创建数据应用程序一般有以下几步骤：

1. 创建应用框架
2. 加入 ATL 产生的模板类
3. 在应用中使用产生的数据访问对象

3. 不用 ATL 使用 OLE DB 数据使用程序

利用 ATL 模板产生数据使用程序较为简单，但适用性不广，不能动态适应数据库的变化。下面我们介绍直接使用 MFC OLE DB 类来生成数据使用程序。

模板的使用

OLE DB 数据使用者模板是由一些模板组成的，包括如下一些模板，下面对一些常用类作一些介绍。

1. 会话类

CDataSource 类

CDataSource 类与 OLE DB 的数据源对象相对应。这个类代表了 OLE DB 数据提供程序和数据源之间的连接。只有当数据源的连接被建立之后，才能产生会话对象，可以调用 Open 来打开数据源的连接。

CSession 类

CSession 所创建的对象代表了一个单独的数据库访问的会话。一个用 CDataSource 类产生的数据源对象可以创建一个或者多个会话，要在数据源对象上产生一个会话对象，需要调用函数 Open()来打开。同时，会话对象还可用于创建事务操作。

CEnumeratorAccessor 类

CEnumeratorAccessor 类是用来访问枚举器查询后所产生的行集中可用数据提供程序的信息的访问器，可提供当前可用的数据提供程序和可见的访问器。

2. 访问器类

CAcessor 类

CAccessor 类代表与访问器的类型。当用户知道数据库的类型和结构时，可以使用此类。它支持对一个行集采用多个访问器，并且，存放数据的缓冲区是由用户分配的。

CDynamicAccessor 类

CDynamicAccessor 类用来在程序运行时动态的创建访问器。当系统运行时，可以动态地从行集中获得列的信息，可根据此信息动态地创建访问器。

CManualAccessor 类

CManualAccessor 类中以在程序运行时将列与变量绑定或者是将参数与变量捆定。

3. 行集类

CRowSet 类

CRowSet 类封装了行集对象和相应的接口，并且提供了一些方法用于查询、设置数据等。可以用 Move（）等函数进行记录移动，用 GetData()函数读取数据，用 Insert()、Delete()、SetData()来更新数据。

CBulkRowset 类

CBulkRowset 类用于在一次调用中取回多个行句柄或者对多个行进行操作。

CArrayRowset 类

CArrayRowset 类提供用数组下标进行数据访问。

4. 命令类

CTable 类

CTable 类用于对数据库的简单访问，用数据源的名称得到行集，从而得到数据。

CCommand 类

CCommand 类用于支持命令的数据源。可以用 Open()函数来执行 SQL 命令，也可以 Prepare（）函数先对命令进行准备，对于支持命令的数据源，可以提高程序的灵活性和健壮性。

在 stdafx.h 头文件里，加入如下代码。

#include

extern CComModule _Module;

#include

#include

#include // if you are using schema templates

在 stdafx.cpp 文件里，加入如下代码。

#include

CComModule _Module;
决定使用何种类型的存取程序和行集。
获取数据

在打开数据源，会话，行集对象后就可以获取数据了。所获取的数据类型取决于所用的存取程序，可能需要绑定列。按以下步骤。

1. 用正确的命令打开行集对象。
2. 如果使用 CManualAccessor，在使用之前与相应列进行绑定。要绑定列，可以用函数 GetColumnInfo,如下所示：

   // Get the column information

   ULONG ulColumns = 0;

   DBCOLUMNINFO* pColumnInfo = NULL;

   LPOLESTR pStrings = NULL;

   if (rs.GetColumnInfo(&ulColumns, &pColumnInfo, &pStrings) != S_OK)

   AfxThrowOLEDBException(rs.m_pRowset, IID_IColumnsInfo);

   struct MYBIND* pBind = new MYBIND[ulColumns];

   rs.CreateAccessor(ulColumns, &pBind[0], sizeof(MYBIND)*ulColumns);

   for (ULONG l=0; l< P>

   rs.AddBindEntry(l+1, DBTYPE_STR, sizeof(TCHAR)*40, &pBind[l].szValue, NULL, &pBind[l].dwStatus);

   rs.Bind();

3. 用 while 循环来取数据。在循环中，调用 MoveNext 来测试光标的返回值是否为 S_OK，如下所示：

   while (rs.MoveNext() == S_OK)

   {

   // Add code to fetch data here

   // If you are not using an auto accessor, call rs.GetData()

   }

4. 在 while 循环内，可以通过不同的存取程序获取数据。

1. 如果使用的是 CAccessor 类，可以通过使用它们的数据成员进行直接访问。如下所示：
2. 如果使用的是 CDynamicAccessor 或 CDynamicParameterAccessor 类，可以通过 GetValue 或 GetColumn 函数来获取数据。可以用 GetType 来获取所用数据类型。如下所示：

   while (rs.MoveNext() == S_OK)

   {

   // Use the dynamic accessor functions to retrieve your

   // data

   ULONG ulColumns = rs.GetColumnCount();

   for (ULONG i=0; i<>

   {

   rs.GetValue(i);

   }

   }

3. 如果使用的是 CManualAccessor,可以指定自己的数据成员，绑定它们。就可以直接存取。如下所示：

while (rs.MoveNext() == S_OK)

```
{

// Use the data members you specified in the calls to

// AddBindEntry.

wsprintf("%s", szFoo);

}
```

决定行集的数据类型

在运行时决定数据类型，要用动态或手工的存取程序。如果用的是手工存取程序，可以用 GetColumnInfo 函数得到行集的列信息。从这里可以得到数据类型。

5．4 总结

由于现在有多种数据源，，想要对这些数据进行访问管理的唯一途径就是通过一些同类机制来实现，如 OLE DB。高级 OLE DB 结构分成两部分：客户和提供者。客户使用由提供者生成的数据。

就像其它基于 COM 的多数结构一样，OLE DB 的开发人员需要实现很多的接口，其中大部分是模板文件。

当生成一个客户对象时，可以通过 ATL 对象向导指向一个数据源而创建一个简单的客户。ATL 对象向导将会检查数据源并创建数据库的客户端代理。从那里，可以通过 OLE DB 客户模板使用标准的浏览函数。

当生成一个提供者时，向导提供了一个很好的开端，它们仅仅是生成了一个简单的提供者来列举某一目录下的文件。然后，提供者模板包含了 OLE DB 支持的完全补充内容。在这种支持下，用户可以创建 OLE DB 提供者，来实现行集定位策略、数据的读写以及建立书签

6、 使用 ADO

6．1 概述

ADO 是 ActiveX 数据对象（ActiveX Data Object），这是 Microsoft 开发数据库应用程序的面向对象的新接口。ADO 访问数据库是通过访问 OLE DB 数据提供程序来进行的，提供了一种对 OLE DB 数据提供程序的简单高层访问接口。

ADO 技术简化了 OLE DB 的操作，OLE DB 的程序中使用了大量的 COM 接口，而 ADO 封装了这些接口。所以，ADO 是一种高层的访问技术。

ADO 技术基于通用对象模型（COM），它提供了多种语言的访问技术，同时，由于 ADO 提供了访问自动化接口，所以，ADO 可以用描述的脚本语言来访问 VBScript,VCScript 等。

6．2 在 VC 中使用 ADO

可以使用 VC6 提供的 ActiveX 控件开发应用程序，还可以用 ADO 对象开发应用程序。使用 ADO 对象开发应用程序可以使程序开发者更容易地控制对数据库的访问，从而产生符合用户需求的数据库访问程序。

使用 ADO 对象开发应用程序也类似其它技术，需产生与数据源的连接，创建记录等步骤，但与其它访问技术不同的是，ADO 技术对对象之间的层次和顺序关系要求不是太严格。在程序开发过程中，不必选建立连接，然后才能产生记录对象等。可以在使用记录的地方直接使用记录对象，在创建记录对象的同时，程序自动建立了与数据源的连接。这种模型有力的简化了程序设计，增强了程序的灵活性。下面讲述使用 ADO 对象进行程序设计的方法。

6．21 引入 ADO 库文件

使用 ADO 前必须在工程的 stdafx.h 文件里用直接引入符号＃import 引入 ADO 库文件，以使编译器能正确编译。代码如下所示：

```
#define INITGUID

#import "c:\program files\common files\system\ado\msado15.dll" no_namespace
rename("EOF","EndOfFile")

#include "icrsint.h"
```

这行语句声明在工程中使用 ADO，但不使用 ADO 的名字空间，并且为了避免冲突，将 EOF 改名为 EndOfFile。

6．22 初始化 ADO 环境

在使用 ADO 对象之前必须先初始化 COM 环境。初始化 COM 环境可以用以下代码完成：

```
::CoInitialize(NULL);
```

在初始化 COM 环境后，就可以使用 ADO 对象了，如果在程序前面没有添加此代码，将会产生 COM 错误。

在使用完 ADO 对象后，需要用以下的代码将初始化的对象释放：

```
::CoUninitialize();
```

此函数清除了为 ADO 对象准备的 COM 环境。

6．23 接口简介

ADO 库包含三个基本接口:

__ConnectionPtr 接口、

__CommandPtr 接口、

__RecordsetPtr 接口,

__ConnectionPtr 接口返回一个记录集或一个空指针。通常使用它来创建一个数据连接或执行一条不返回任何结果的 SQL 语句，如一个存储过程。用__ConnectionPtr 接口返回一个记录集不是一个好的使用方法。通常同 CDatabase 一样，使用它创建一个数据连接，然后使用其它对象执行数据输入输出操作。

__CommandPtr 接口返回一个记录集。它提供了一种简单的方法来执行返回记录集的存储过程和 SQL 语句。在使用__CommandPtr 接口时，可以利用全局__ConnectionPtr 接口，也可以在__CommandPtr 接口里直接使用连接串。如果只执行一次或几次数据访问操作，后者是比较好的选择。但如果要频繁访问数据库，并要返回很多记录集，那么，应该使用全局__ConnectionPtr 接口创建一个数据连接，然后使用__CommandPtr 接口执行存储过程和 SQL 语句。

__RecordsetPtr 是一个记录集对象。与以上两种对象相比，它对记录集提供了更多的控制功能，如记录锁定，游标控制等。同__CommandPtr 接口一样，它不一定要使用一个已经创建的数据连接，可以用一个连接串代替连接指针赋给__RecordsetPtr 的 connection 成员变量，让它自己创建数据连接。如果要使用多个记录集，最好的方法是同 Command 对象一样使用已经创建了数据连接的全局—ConnectionPtr 接口，然后使用__Recordse7tPtr 执行存储过程和 SQL 语句。

6、24 使用 ADO 访问数据库

__ConnectionPtr 是一个连接接口，首先创建一个__ConnectionPtr 接口实例，接着指向并打开一个 ODBC 数据源或 OLE DB 数据提供者(Provider)。以下代码分别创建一个基于 DSN 和非 DSN 的数据连接。

```
//使用__ConnectionPtr(基于 DSN)

__ConnectionPtr MyDb;

MyDb.CreateInstance(__uuidof(Connection));

MyDb－〉Open("DSN=samp;UID=admin;PWD=admin","","",－1);

//使用—ConnectionPtr (基于非 DSN)

__ConnectionPtr MyDb;
```

```
MyDb.CreateInstance(__uuidof(Connection));

MyDb.Open("Provider=SQLOLEDB;SERVER=server;DATABASE=samp;UID=admin;PWD=admin","","",
－1);

//使用__RecordsetPtr 执行 SQL 语句

__RecordsetPtr MySet;

MySet.CreateInstance(__uuidof(Recordset));

MySet－〉Open("SELECT *     FROM some__table",
MyDb.GetInterfacePtr(),adOpenDynamic,adLockOptimistic,adCmdText);
```

现在我们已经有了一个数据连接和一个记录集，接下来就可以使用数据了。从以下代码可以看到，使用 ADO 的__RecordsetPtr 接口，就不需要像 DAO 那样频繁地使用大而复杂的数据结构 VARIANT，并强制转换各种数据类型了，这也是 ADO 的优点之一。假定程序有一个名称为 m__List 的 ListBox 控件，下面代码我们用__RecordsetPtr 接口获取记录集数据并填充这个 ListBox 控件：

```
__variant__t Holder

try{while(!MySet－〉adoEOF)

{ Holder = MySet－〉GetCollect("FIELD__1");

if(Holder.vt!=VT__NULL)

m__List.AddString((char     )__bstr__t(Holder));

MySet－〉MoveNext();} }

catch(__com__error       e)

{ CString Error = e－〉ErrorMessage();

  AfxMessageBox(e－〉ErrorMessage());

} catch(...)

{ MessageBox("ADO 发生错误!");}
```

必须始终在代码中用 try 和 catch 来捕获 ADO 错误,否则 ADO 错误会使你的应用程序崩溃。当 ADO 发生运行错误时(如数据库不存在)，OLE DB 数据提供者将自动创建一个__com__error 对象，并将有关错误信息填充到这个对象的成员变量。

6．25 类型转换

由于 COM 对象是跨平台的，它使用了一种通用的方法来处理各种类型的数据，因此 CString 类和 COM 对象是不兼容的，我们需要一组 API 来转换 COM 对象和 C＋＋类型的数据。__vatiant__t 和__bstr__t 就是这样两种对象。它们提供了通用的方法转换 COM 对象和 C＋＋类型的数据。

6．3 在 VB 中使用 ADO

ADO 提供执行以下操作的方式：

1、连接到数据源。同时，可确定对数据源的所有更改是否已成功或没有发生。

2、指定访问数据源的命令，同时可带变量参数，或优化执行。

3、执行命令。

3、如果这个命令使数据按表中的行的形式返回，则将这些行存储在易于检查、操作或更改的缓存中。

4、适当情况下，可使用缓存行的更改内容来更新数据源。

5、提供常规方法检测错误（通常由建立连接或执行命令造成）。

在典型情况下，需要在编程模型中采用所有这些步骤。但是，由于 ADO 有很强的灵活性，所以最后只需执行部分模块就能做一些有用的工作。

以下元素是 ADO 编程模型中的关键部分：

6．31 连接

通过“连接”可从应用程序访问数据源，连接是交换数据所必需的环境。对象模型使用 Connection 对象使连接概念得以具体化。

“事务”用于界定在连接过程中发生的一系列数据访问操作的开始和结束。ADO 可明确事务中的操作造成的对数据源的更改或者成功发生，或者根本没有发生。如果取消事务或它的一个操作失败，则最终的结果将仿佛是事务中的操作均未发生，数据源将会保持事务开始以前的状态。对象模型无法清楚地体现出事务的概念，而是用一组 Connection 对象方法来表示。ADO 从 OLE DB 提供者访问数据和服务。Connection 对象用于指定专门的提供者和任意参数。

6．32 命令

通过已建立的连接发出的“命令”可以某种方式来操作数据源。一般情况下，命令可以在数据源中添加、删除或更新数据，或者在表中以行的格式检索数据。

对象模型用 Command 对象来体现命令概念。使用 Command 对象可使 ADO 优化命令的执行。

1. 参数

通常，命令需要的变量部分即“参数”可以在命令发布之前进行更改。例如，可重复发出相同的数据检索命令，但每一次均可更改指定的检索信息。
参数对与函数活动相同的可执行命令非常有用，这样就可知道命令是做什么的，但不必知道它如何工作。例如，可发出一项银行过户命令，从一方借出贷给另一方。可将要过户的款额设置为参数。
对象模型用 Parameter 对象来体现参数概念。

6．33 记录集

如果命令是在表中按信息行返回数据的查询（行返回查询），则这些行将会存储在本地。
对象模型将该存储体现为 Recordset 对象。但是，不存在仅代表单独一个 Recordset 行的对象。
记录集是在行中检查和修改数据最主要的方法。

6．34 字段

一个记录集行包含一个或多个“字段”。如果将记录集看作二维网格，字段将排列构成“列”。每一字段（列）都分别包含有名称、数据类型和值的属性，正是在该值中包含了来自数据源的真实数据。
对象模型以 Field 对象体现字段。
要修改数据源中的数据，可在记录集行中修改 Field 对象的值，对记录集的更改最终被传送给数据源。作为选项，Connection 对象的事务管理方法能够可靠地保证更改要么全部成功，要么全部失败。

6．35 错误

错误随时可在应用程序中发生，通常是由于无法建立连接、执行命令或对某些状态（例如，试图使用没有初始化的记录集）的对象进行操作。
对象模型以 Error 对象体现错误。
任意给定的错误都会产生一个或多个 Error 对象，随后产生的错误将会放弃先前的 Error 对象组。

6．36 属性

每个 ADO 对象都有一组唯一的“属性”来描述或控制对象的行为。
属性有内置和动态两种类型。内置属性是 ADO 对象的一部分并且随时可用。动态属性则由特别的数据提供者添加到 ADO 对象的属性集合中，仅在提供者被使用时才能存在。
对象模型以 Property 对象体现属性。

6．37 集合

ADO 提供“集合”，这是一种可方便地包含其他特殊类型对象的对象类型。使用集合方法可按名称（文本字符串）或序号（整型数）对集合中的对象进行检索。
ADO 提供四种类型的集合：
Connection 对象具有 Errors 集合，包含为响应与数据源有关的单一错误而创建的所有 Error 对象。

Command 对象具有 Parameters 集合，包含应用于 Command 对象的所有 Parameter 对象。

Recordset 对象具有 Fields 集合，包含所有定义 Recordset 对象列的 Field 对象。

此外，Connection、Command、Recordset 和 Field 对象都具有 Properties 集合。它包含所有属于各个包含对象的 Property 对象。

ADO 对象拥有可在其上使用的诸如“整型”、“字符型”或“布尔型”这样的普通数据类型来设置或检索值的属性。然而，有必要将某些属性看成是数据类型“COLLECTION OBJECT”的返回值。相应的，集合对象具有存储和检索适合该集合的其他对象的方法。

6．38 事件

ADO 2.0 支持事件，事件是对某些操作将要或已经发生的通知。

有两类事件：ConnectionEvent 和 RecordsetEvent。Connection 对象产生 ConnectionEvent 事件，而 Recordset 对象则产生 RecordsetEvent 事件。

事件由事件处理程序例程处理，该例程在某个操作开始之前或结束之后被调用。

某些事件是成对出现的。开始操作前调用的事件名格式为 WillEvent（Will 事件），而操作结束后调用的事件名格式为 EventComplete（Complete 事件）。

其余的不成对事件只在操作结束后发生。（其名称没有任何固定模式。）事件处理程序由状态参数控制。附加信息由错误和对象参数提供。

可以请求事件处理程序不接受第一次通知以后的任何通知。例如，可以选择只接收 Will 事件或 Complete 事件。

下面的代码显示了一个使用 ADO 的例子。

首先加入 Microsoft ActiveX Data Object 2.0 Library 引用。

```
Dim db As Connection

Set db = New Connection

db.CursorLocation = adUseClient

db.Open "PROVIDER=MSDASQL;DSN=TestDatabase", "sa", "", -1

Dim i As Long

Dim id As Long

Dim value As Single

Dim rst As New Recordset

Set rst = New Recordset

rst.Open "select * from  模拟量变化历史表", db, adOpenDynamic, adLockOptimistic

rst.MoveFirst
```

```
For i = 0 To rst.RecordCount - 1

id = rst.Fields("ID")

value=rst.Fields(“VALUE”)

rst.MoveNext

Next i

rst.Close

Set rst = Nothing

db.Close
```

6．4 总结

ADO 技术是访问数据库的新技术，具有易于使用、访问灵活、应用广泛的特点。用 ADO 访问数据源的特点可总结如下：

1. 易于使用

   这是 ADO 技术的最重要的一个特征。由于 ADO 是高层应用，所以相对于 OLE DB 或者 ODBC 来说，它具有面向对象的特性。同时，在 ADO 的对象结构中，其对象之间的层次关系并不明显。相对于 DAO 等访问技术来讲，又不必关心对象的构造顺序和构造层次。对于要用的对象，不必选建立连接、会话等对象，只需直接构造即可，方便了应用程序的编制。

2. 高速访问数据源

   由于 ADO 技术基于 OLE DB，所以，它也继承了 OLE DB 访问数据库的高速性。

3. 可以访问不同数据源

   ADO 技术可以访问包括关系数据库和非关系数据库的所有文件系统。此特点使应用程序有很多的灵活性和通用性。

4. 可以用于 Microsoft ActiveX 页

   ADO 技术可以以 ActiveX 控件的形式出现，所以，可以被用于 Microsoft ActiveX 页，此特征可简化 WEB 页的编程。

5. 程序占用内存少

   由于 ADO 是基于组件对象模型（COM）的访问技术，所以，用 ADO 产生的应用程序占用内存少。

7、总结

要在访问数据时判断出应该使用哪一种技术，这并不容易。可能需要公用实用程序来处理多个数据库类型；部分数据可能出现在本地硬盘驱动器上，部分在网络上，还有一部分在主机上。甚至客户安装在设备上的产品也会使这种选择更加困难。例如，你所期待的 ODBC 支持级别也许依赖于所安装的 Microsoft Office 的版本，因为这个产品不提供 ODBC 支持。你还会发现，ADO 类提供的对象和方法要比 ODBC 类多。ADO 可以提供程序中绝对必须具有的一些特性棗例如，你会发现 OLE-DB 和 ADO 两者都支持 DFX_Currency，但在 ODBC 中没有对应的功能，但你要想掌握它们也必须付出一定的努力。

选择 OLE-DB 或 ODBC 时，有几条一般的规则。因为 ADO 实际上只是 OLE-DB 的包装，所以这些规则也适用于它。下面提供一些基本的原则，可以用来帮助你决定选择 OLE-DB 还是 ODBC。

非 OLE 环境 如果要访问支持 ODBC 的数据库，而该数据库又在不支持 OLE 的服务器上，那么 ODBC 是最好的选择。

非 SQL 环境 ODBC 在处理 SQL 时非常出众。处理非 SQL 数据库时，OLE-DB 则具有非常明显的优势。

OLE 环境 对支持 OLE 的服务器来说，选择 OLE-DB 还是 ODBC 也许是希望各半。如果有 ODBC 驱动程序可供利用，那么使用 ODBC 是一个好主意；否则，就只有选择 OLE-DB 了。

所需的互操作性 如果需要可互操作的数据库部件，那么只有选择 OLE-DB。