

# 1 COM组件概述

**COM与组件分开理解**，COM是开发软件组件的一种方法，组件实际上是一些小的二进制可执行程序，它们可以给应用程序，操作系统以及其他组件提供服务。

COM最早的设计意图是，跨语言实现程序组件的重用，比如说VC++开发一个控件，在VB中调用，或者在VB中开发一个组件库，给VC++调用。

COM最广泛的应用是ActiveX控件，时至今日，能在很多网站上看到它，比如淘宝安全登录控件，网银控件，甚至大名鼎鼎的Flash Player控件等，整个Visual Basic都基于COM以及ActiveX控件。

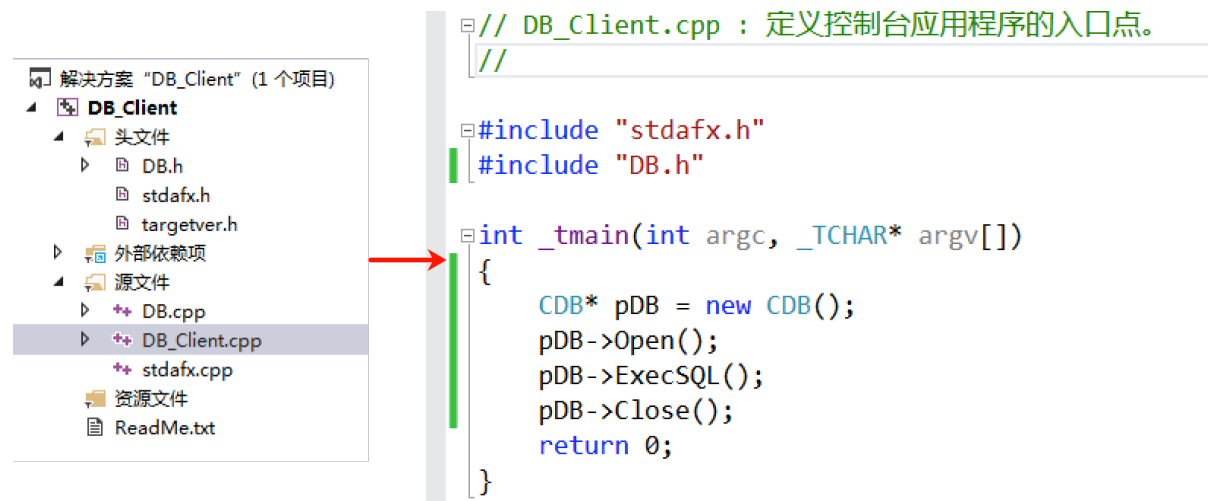
## 2 从C++到COM

### 2.1 C++客户重用C++对象

C++客户重用C++对象，这个方法很简单就是将C++类的接口文件(.h)和实现文件(.cpp)提供给客户即可。



如上所示是一个简单的C++类，如果客户要使用的话，我们需要将这个文件提供给他：



那么这样就会存在一个缺点，就是你的代码都会被客户看见，为了解决这个问题我们可以将C++对象打包到DLL中。

## 2.2 将C++对象打包到DLL中

### 2.2.1 代码改造

将C++对象打包到DLL中，使用的还是之前的代码，不过我们需要改造一下，如下图所示（创建Win32项目的时候选择DLL项目）：

```
#pragma once

#ifdef MACRO_DB_DLL
#define DEF_DLL_PORT _declspec(dllexport)
#else
#define DEF_DLL_PORT _declspec(dllimport)
#endif

typedef long HRESULT;

class CDB {
public:
    HRESULT DEF_DLL_PORT Open();
    HRESULT DEF_DLL_PORT ExecSQL();
    HRESULT DEF_DLL_PORT Close();
private:
    int m_nState;
public:
    CDB(void);
    ~CDB(void);
};
```

```
#include <windows.h>
#include <tchar.h>

#define MACRO_DB_DLL

#include "DB.h"

CDB::CDB(void){
    m_nState = 0;
}

CDB::~CDB(void){
}

HRESULT CDB::Open(){
    m_nState = 1;
    ::MessageBox(NULL, _T("Open Successfully!"), _T("Alert"), MB_OK);
    return NO_ERROR;
}

HRESULT CDB::ExecSQL(){
    m_nState = 1;
    ::MessageBox(NULL, _T("ExecSQL Successfully!"), _T("Alert"), MB_OK);
    return NO_ERROR;
}

HRESULT CDB::Close(){
    m_nState = 0;
    ::MessageBox(NULL, _T("Close Successfully!"), _T("Alert"), MB_OK);
    return NO_ERROR;
}
```

虽然这样我们就完成了改造，但是为了便于客户侧使用，不让客户去操心内存的管理，我们需要在代码中写好申请、释放，这就需要实现一个类工厂来进行内存的管理。

```
1 // 声明类工厂
2 class CDBSrvFactory {
3 public:
4     HRESULT DEF_DLL_PORT CreateDB(CDB** ppObject); // 申请CDB对象
5     ULONG DEF_DLL_PORT Release(); // 释放类工厂对象
6 };
7
8 // 声明返回类工厂对象的引出函数
9 HRESULT DEF_DLL_PORT DllGetClassFactoryObject(CDBSrvFactory** ppObject);
```

代码实现如下：

```
1 HRESULT CDBSrvFactory::CreateDB(CDB** ppObject) {
2     *ppObject = new CDB();
3     return NO_ERROR;
4 }
```

```

5
6     ULONG CDBSrvFactory::Release() {
7         delete this;
8         return 0;
9     }
10
11     HRESULT DllGetClassFactoryObject(CDBSrvFactory** ppObject) {
12         *ppObject = new CDBSrvFactory();
13         return NO_ERROR;
14     }

```

完成了这些之后别忘记还有一个CDB对象本身也需要释放，所以需要增加一个方法：

```

1     class CDB {
2     public:
3         HRESULT DEF_DLL_PORT Open();
4         HRESULT DEF_DLL_PORT ExecSQL();
5         HRESULT DEF_DLL_PORT Close();
6         HRESULT DEF_DLL_PORT Release();
7     private:
8         int m_nState;
9     public:
10         CDB(void);
11         ~CDB(void);
12     };

```

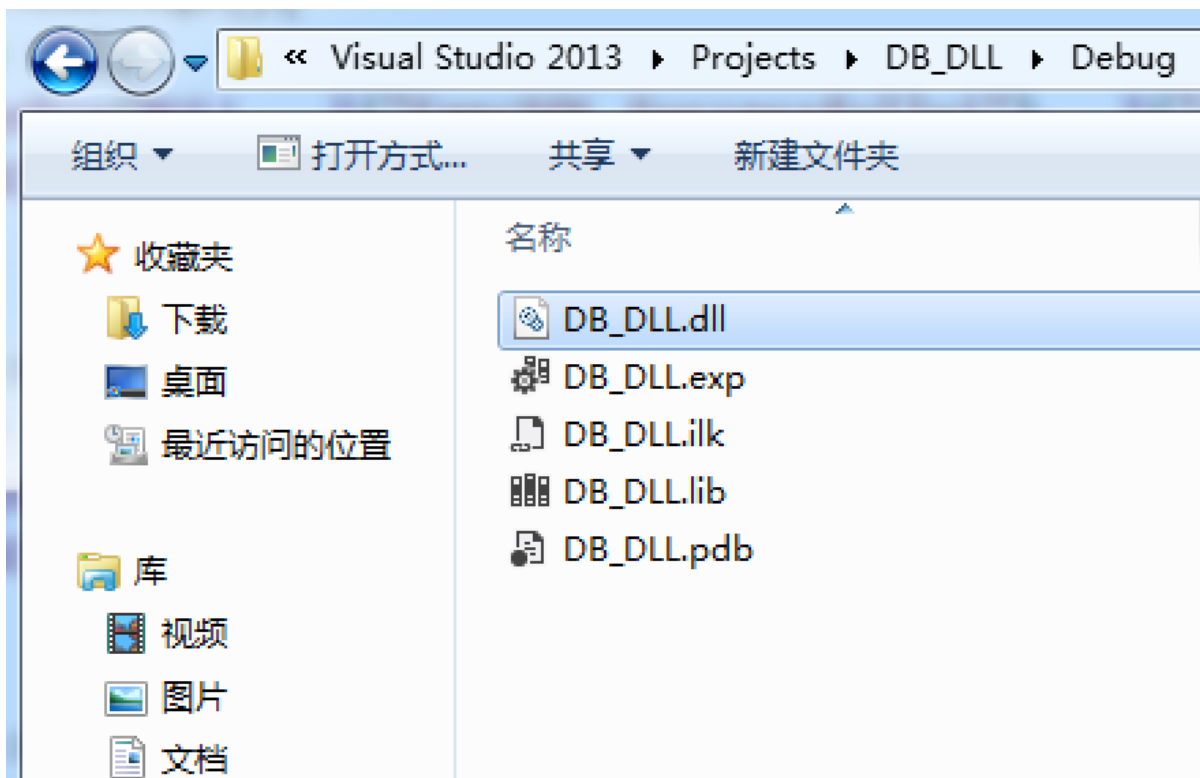
Release方法实现如下所示：

```

1     HRESULT CDB::Release(){
2         delete this;
3         return NO_ERROR;
4     }

```

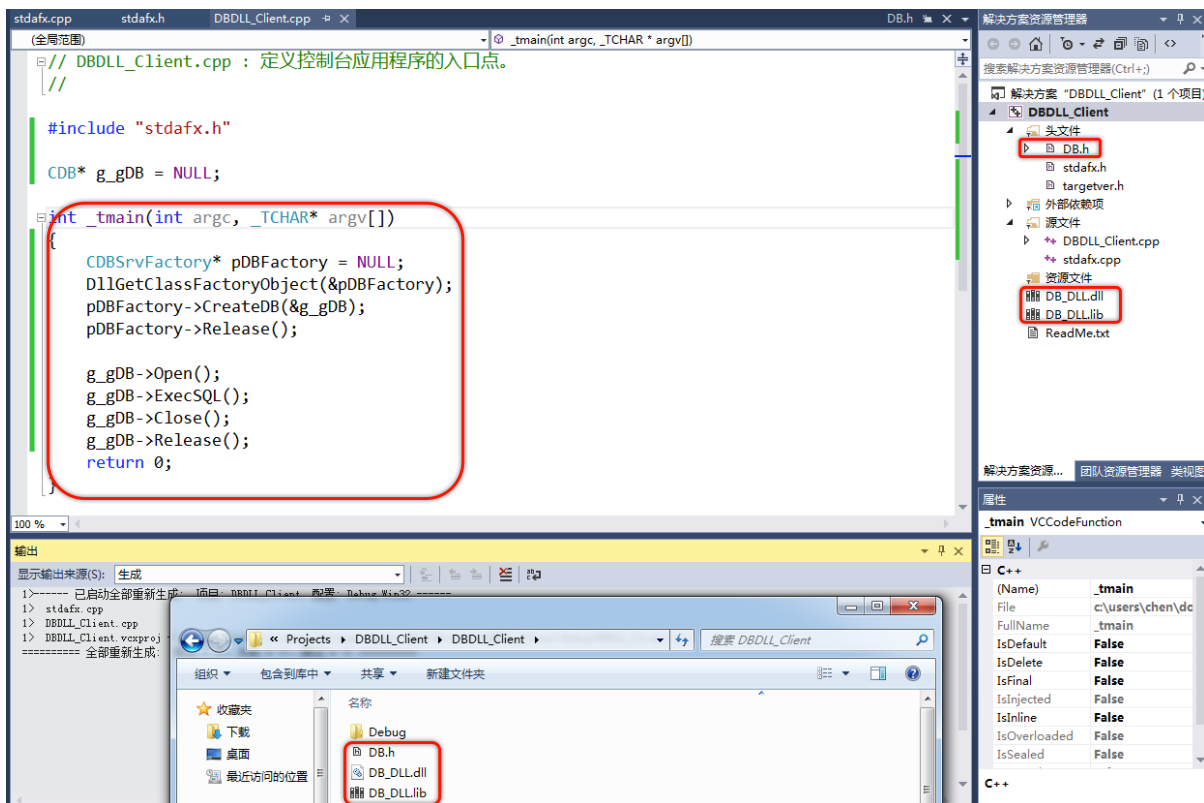
最后我们编译一下即可生成：



## 2.2.2 调用文件

在客户侧去使用，需要注意的是你要给到头文件、lib文件和dll文件：

```
1 #include "DB.h"
```

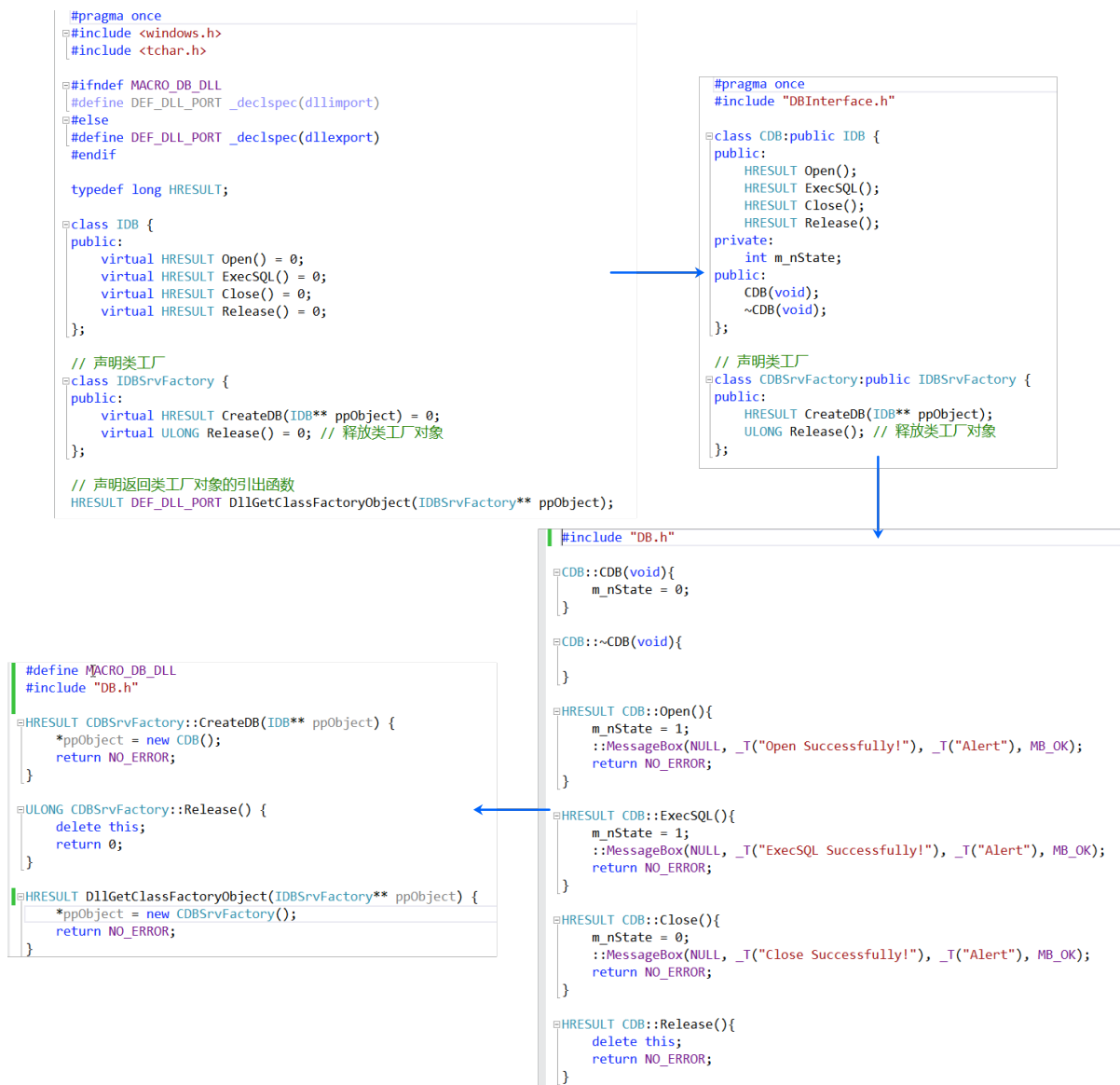


同样这里的缺点就很容易暴露出来了，我们的对象的所有私有成员会被客户看得一清二楚，即便客户不能访问它们，如果改变了数据成员的大小，所有客户程序必须重新编译，同样为了避免这种情况，我们可以使用抽象基类。

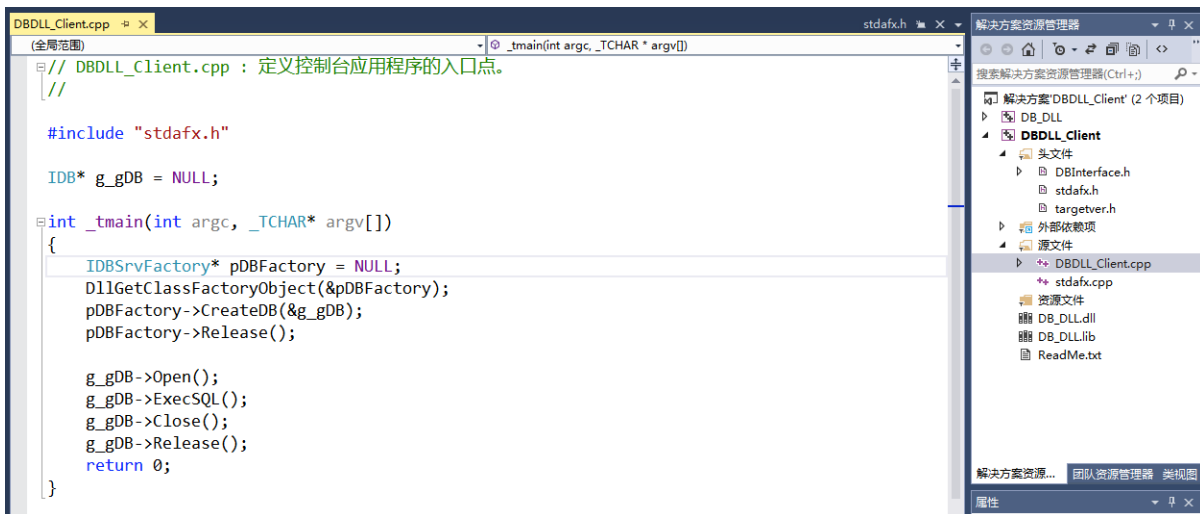
## 2.3 C++对象使用抽象基类

利用C++的抽象基类建立一个只包含所有导出成员函数的地址指针表，客户调用函数时只需简单的查找这个表获取函数地址即可调用。

抽象基类很好理解，就是将成员方法变成纯虚函数即可，然后子类继承这个抽象类，如下图所示：



简单改造一番之后将lib、dll文件和抽象基类头文件打包给客户，这样客户就只能看见抽象基类定义的内容，而无法看见子类的私有成员了。



同样这个也有缺点：

1. 客户与所有的组件交互方式不统一（如果组件进行了更新，就需要改代码）；
2. 会重复编写大量客户与组件通信的基础性代码（比如我们写了一个类就写了一个类工厂和 **DllGetClassFactoryObject函数**）。

而解决这一缺陷的手段就是将C++对象改为COM对象。

## 2.4 将C++对象变成COM对象

将C++对象改为COM对象，需要调用COM库创建对象，实现接口的引用计数，类工厂需要使用标准的 **IClassFactory** 接口来实现DLL的动态卸载和对对象自注册。

需要注意的是在改造过程中所有成员函数都需要添加\_stdcall调用约定，因为COM对象在Win32下采用标准调用约定。

**注：**在这里会引出到很多不了解的概念和意义，不用深究，后续章节会去详细讲解。

### 2.4.1 修改接口文件

1. 将IDB类改为由IUnknown类派生，删除Release成员函数声明（包括CDB类）。

```

1  class IDB : public IUnknown {
2  public:
3      virtual HRESULT _stdcall Open() = 0;
4      virtual HRESULT _stdcall ExecSQL() = 0;
5      virtual HRESULT _stdcall Close() = 0;
6  };

```

2. 删除类工厂IDBSrvFactory声明，因为我们现在要使用标准类工厂接口IClassFactory。
3. 删除DllGetClassFactoryObject函数的声明和定义。



## 2.4.2 修改对象程序

1.将CDBSrvFactory类由IDBSrvFactory类派生改为由IClassFactory类派生，将CreateDB成员函数改为CreateInstance，同时删除该成员函数的定义，并添加一个成员函数LockServer；为CDB类和CDBSrvFactory类都添加一个引用计数变量m\_dwRefCount；为CDB类和CDBSrvFactory类加上QueryInterface、AddRef和Release三个成员函数。

```

1  class CDB : public IDB {
2  public:
3      HRESULT _stdcall Open();
4      HRESULT _stdcall ExecSQL();
5      HRESULT _stdcall Close();
6      HRESULT _stdcall QueryInterface(REFIID riid, void** ppObject);
7      ULONG _stdcall AddRef();
8      ULONG _stdcall Release();
9  private:
10     ULONG m_dwRefCount;
11     int m_nState;
12 public:
13     CDB(void);
14     ~CDB(void);
15 };
16
17 // 声明类工厂
18 class CDBSrvFactory : public IClassFactory {
19 public:
20     HRESULT _stdcall CreateInstance(IUnknown *pUnkOuter, REFIID riid, void*
21 * ppObject);
22     HRESULT _stdcall LockServer(BOOL fLock);
23     HRESULT _stdcall QueryInterface(REFIID riid, void** ppObject);
24     ULONG _stdcall AddRef();
25     ULONG _stdcall Release();
26
27     CDBSrvFactory(void);
28 private:
29     ULONG m_dwRefCount;
30 };

```

在实现中声明一个外部变量g\_dwRefCount：

```

1  extern ULONG g_dwRefCount;

```

2.在CDB构造函数中将m\_dwRefCount初始化为0，实现CDB类的QueryInterface，AddRef，Release三个成员函数。

```

1  CDB::CDB(void) {
2      m_nState = 0;
3      m_dwRefCount = 0;

```

```

4     }
5
6     ...
7
8     HRESULT CDB::QueryInterface(REFIID riid, void** ppObject)
9     {
10         if (riid == IID_IUnknown || riid == IID_IDB)
11         {
12             *ppObject = (IDB*) this;
13         }
14         else
15         {
16             return E_NOINTERFACE;
17         }
18         AddRef();
19         return NO_ERROR;
20     }
21
22     ULONG CDB::AddRef()
23     {
24         g_dwRefCount++;
25         m_dwRefCount++;
26         return m_dwRefCount;
27     }
28
29     ULONG CDB::Release()
30     {
31         g_dwRefCount--;
32         m_dwRefCount--;
33         if (m_dwRefCount == 0)
34         {
35             delete this;
36             return 0;
37         }
38         return m_dwRefCount;
39     }

```

需要注意的是在QueryInterface成员函数中有一个IID\_IDB变量，这个是我们自定义的内容，其表示当前COM接口的ID，这个需要我们去自定义：

```

1     // {30DF3432-0266-11cf-BAA6-00AA003E0EED}
2     static const GUID IID_IDB =
3     { 0x30df3432, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed } };

```

需要注意的是，这个变量实际上是一个结构体GUID，注释就是它更加直观的表现形式。

```

1     typedef struct _GUID {
2         unsigned long Data1;
3         unsigned short Data2;
4         unsigned short Data3;

```

```

5     unsigned char Data4[ 8 ];
6 } GUID;

```

在代码中的IID\_IUnknown是COM库的东西，所以我们需要包含一个头文件：

```

1  #include <ole2.h>

```

3.定义全局变量g\_dwRefCount；在构造函数中将m\_dwRefCount初始化为0；实现CDBSrvFactory类的QueryInterface, AddRef, Release三个成员函数；将CDBSrvFactory类的CreateDB成员函数修改为CreateInstance；实现LockServer成员函数；添加DllGetClassObject、DllCanUnloadNow、DllUnregisterServer、DllRegisterServer四个成员函数。

```

1  ULONG g_dwRefCount = 0;
2
3  // {30DF3430-0266-11cf-BAA6-00AA003E0EED}
4  static const GUID CLSID_DBSAMPLE =
5  { 0x30df3430, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed
6  } };
7
8  HRESULT CDBSrvFactory::CreateInstance(IUnknown *pUnkOuter, REFIID riid,
9  void** ppObject)
10 {
11     if (pUnkOuter != NULL)
12     {
13         return CLASS_E_NOAGGREGATION;
14     }
15     CDB* pDB = new CDB;
16     if (FAILED(pDB->QueryInterface(riid, ppObject)))
17     {
18         delete pDB;
19         *ppObject = NULL;
20         return E_NOINTERFACE;
21     }
22     return NO_ERROR;
23 }
24
25 HRESULT CDBSrvFactory::LockServer(BOOL fLock)
26 {
27     if (fLock)
28     {
29         g_dwRefCount++;
30     }
31     else
32     {
33         g_dwRefCount--;
34     }
35     return NO_ERROR;
36 }
37
38 CDBSrvFactory::CDBSrvFactory()
39 {

```

```

38     m_dwRefCount = 0;
39 }
40
41 HRESULT CDBSrvFactory::QueryInterface(REFIID riid, void** ppObject)
42 {
43     if (riid == IID_IUnknown || riid == IID_IClassFactory)
44     {
45         *ppObject = (IDB*) this;
46     }
47     else
48     {
49         return E_NOINTERFACE;
50     }
51     AddRef();
52     return NO_ERROR;
53 }
54
55 ULONG CDBSrvFactory::AddRef()
56 {
57     g_dwRefCount++;
58     m_dwRefCount++;
59     return m_dwRefCount;
60 }
61
62 ULONG CDBSrvFactory::Release()
63 {
64     g_dwRefCount--;
65     m_dwRefCount--;
66     if (m_dwRefCount == 0)
67     {
68         delete this;
69         return 0;
70     }
71     return m_dwRefCount;
72 }
73
74 STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppObject)
75 {
76     if (rclsid == CLSID_DBSAMPLE)
77     {
78         CDBSrvFactory *pFactory = new CDBSrvFactory;
79         if (FAILED(pFactory->QueryInterface(riid, ppObject)))
80         {
81             delete pFactory;
82             *ppObject = NULL;
83             return E_INVALIDARG;
84         }
85     }
86     else
87     {
88         // here you could check for additional CLSID's you DLL may provide
89         return CLASS_E_CLASSNOTAVAILABLE;
90     }

```

```

91     return NO_ERROR;
92 }
93
94 HRESULT _stdcall DllCanUnloadNow()
95 {
96     if (g_dwRefCount)
97     {
98         return S_FALSE;
99     }
100    else
101    {
102        return S_OK;
103    }
104 }
105
106 STDAPI DllRegisterServer(void)
107 {
108     HKEY hKeyCLSID, hKeyInproc32;
109     DWORD dwDisposition;
110     if (RegCreateKeyEx(HKEY_CLASSES_ROOT, _T("CLSID\\{30DF3430-0266-11cf-
111     BAA6-00AA003E0EED}"), NULL, _T(""), REG_OPTION_NON_VOLATILE,
112     KEY_ALL_ACCESS, NULL, &hKeyCLSID, &dwDisposition) != ERROR_SUCCESS)
113     {
114         return E_UNEXPECTED;
115     }
116     if (RegSetValueEx(hKeyCLSID, _T(""), NULL, REG_SZ, (BYTE*)_T("DB
117     Sample Server"), sizeof(_T("DB Sample Server"))) != ERROR_SUCCESS)
118     {
119         RegCloseKey(hKeyCLSID);
120         return E_UNEXPECTED;
121     }
122     if (RegCreateKeyEx(hKeyCLSID, _T("InprocServer32"), NULL, _T(""),
123     REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hKeyInproc32,
124     &dwDisposition) != ERROR_SUCCESS)
125     {
126         RegCloseKey(hKeyCLSID);
127         return E_UNEXPECTED;
128     }
129     HMODULE hModule = GetModuleHandle(_T("DB_DLL.DLL"));
130     if (!hModule)
131     {
132         RegCloseKey(hKeyInproc32);
133         RegCloseKey(hKeyCLSID);
134         return E_UNEXPECTED;
135     }
136     TCHAR szName[MAX_PATH + 1];
137     if (GetModuleFileName(hModule, szName, sizeof(szName)) == 0)
138     {
139         RegCloseKey(hKeyInproc32);
140         RegCloseKey(hKeyCLSID);
141         return E_UNEXPECTED;
142     }

```

```

138     if (RegSetValueEx(hKeyInproc32, _T(""), NULL, REG_SZ, (BYTE*)szName,
139         sizeof(TCHAR)*(lstrlen(szName) + 1)) != ERROR_SUCCESS)
140     {
141         RegCloseKey(hKeyInproc32);
142         RegCloseKey(hKeyCLSID);
143         return E_UNEXPECTED;
144     }
145     RegCloseKey(hKeyInproc32);
146     RegCloseKey(hKeyCLSID);
147     return NOERROR;
148 }
149
150 STDAPI DllUnregisterServer(void)
151 {
152     if (RegDeleteKey(HKEY_CLASSES_ROOT, _T("CLSID\\{30DF3430-0266-11cf-
153         BAA6-00AA003E0EED}\\InprocServer32")) != ERROR_SUCCESS)
154     {
155         return E_UNEXPECTED;
156     }
157     if (RegDeleteKey(HKEY_CLASSES_ROOT, _T("CLSID\\{30DF3430-0266-11cf-
158         BAA6-00AA003E0EED}")) != ERROR_SUCCESS)
159     {
160         return E_UNEXPECTED;
161     }
162     return NOERROR;
163 }

```

如上代码中的{30DF3430-0266-11cf-BAA6-00AA003E0EED}，其实就是COM对象的ID，这些都是要在注册表中去注册的，其也是一段GUID的格式，也就是变量CLSID\_DBSAMPLE所表示的内容；在代码中的DB\_DLL.DLL，这个为当前COM组件的文件名，自己需要注意修改一下。

### 2.4.3 DEF导出

我们创建DEF文件将接口导出，以便外部引用：

```

1  EXPORTS
2      ;WEP @1 RESIDENTNAME
3          DllGetClassObject
4          DllCanUnloadNow
5          DllRegisterServer
6          DllUnregisterServer

```

### 2.4.4 客户使用

接下来给只需要将DLL文件、接口头文件、以及对应的COM对象和COM接口的GUID：

```

1  // {30DF3430-0266-11cf-BAA6-00AA003E0EED}
2  static const GUID CLSID_DBSAMPLE =

```

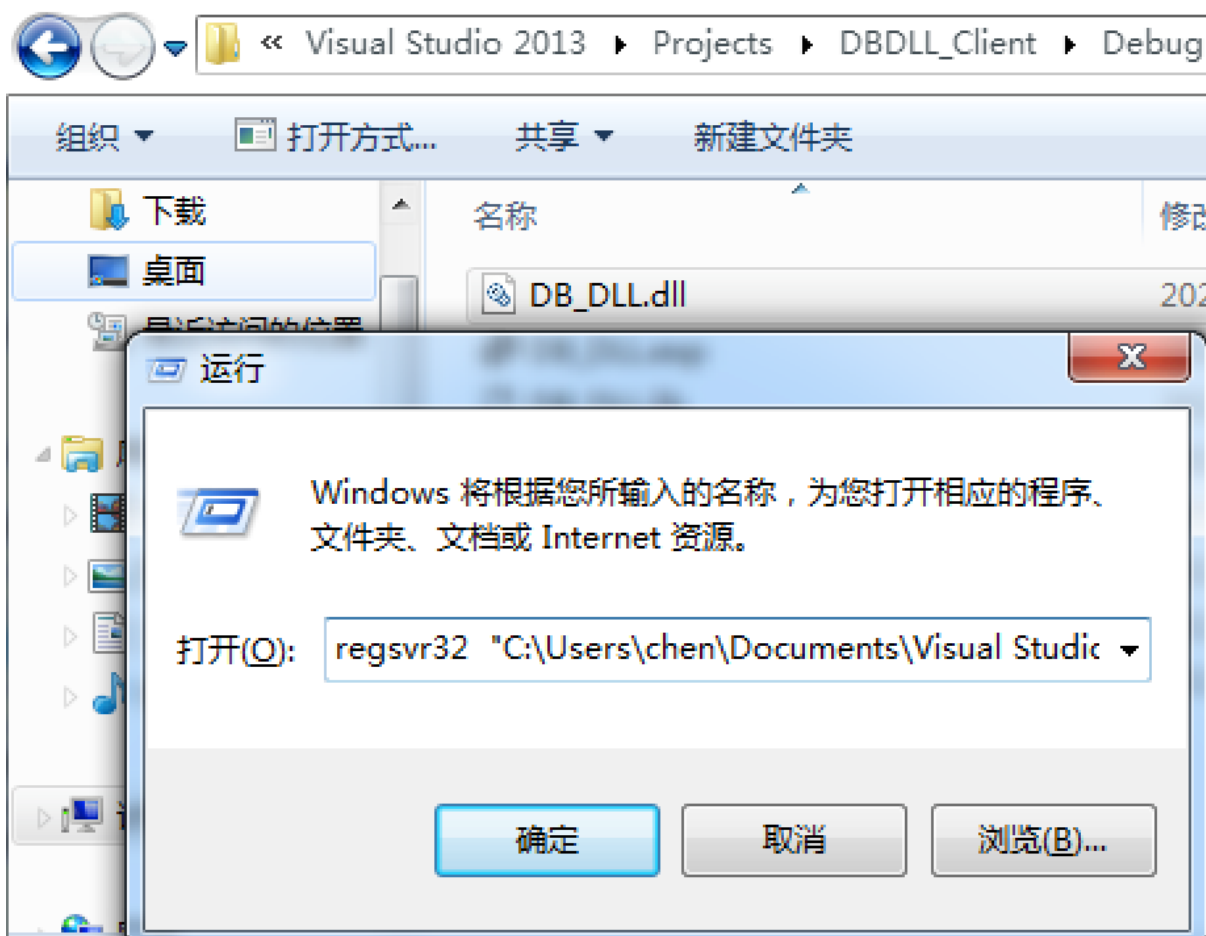
```

3 { 0x30df3430, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed
4 } };
5 // {30DF3432-0266-11cf-BAA6-00AA003E0EED}
6 static const GUID IID_IDB =
7 { 0x30df3432, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed
8 } };

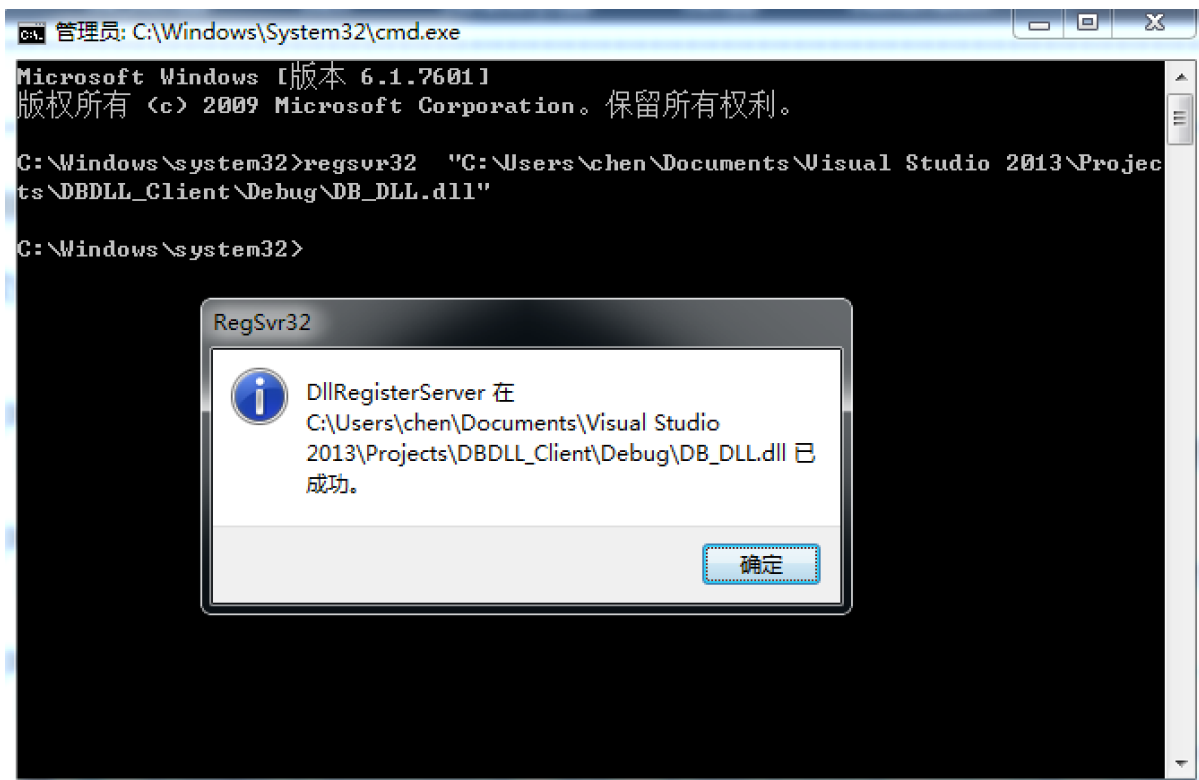
```

然后让客户通过regsvr32命令注册组件：

```
1 regsvr32 DLL文件路径
```



如果注册失败了，有可能是非管理员权限的原因，可以使用管理员权限打开一个cmd然后运行命令：



接着客户端按如下示例代码进行编写调用即可：

```

1  #include "DBInterface.h"
2  #include <iostream>
3  using namespace std;
4
5  // {30DF3430-0266-11cf-BAA6-00AA003E0EED}
6  static const GUID CLSID_DBSAMPLE =
7  { 0x30df3430, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed
8  } };
9  // {30DF3432-0266-11cf-BAA6-00AA003E0EED}
10 static const GUID IID_IDB =
11 { 0x30df3432, 0x266, 0x11cf, { 0xba, 0xa6, 0x0, 0xaa, 0x0, 0x3e, 0xe, 0xed
12 } };
13
14 IDB* g_gDB = NULL;
15
16 int _tmain(int argc, _TCHAR* argv[])
17 {
18     IClassFactory *pDBFactory = NULL;
19     HRESULT hRes;
20     // 初始化
21     hRes = ::CoInitialize(NULL);
22     if (FAILED(hRes))
23     {
24         cout << "Error " << hRes << " CoInitialize!" << endl;
25         return FALSE;
26     }
27 }

```



```
24     }
25     // 连接COM组件
26     hRes = CoGetClassObject(CLSID_DBSAMPLE, CLSCTX_SERVER, NULL,
27     IID_IClassFactory, (void**)&pDBFactory);
28     if (FAILED(hRes))
29     {
30         cout << "Error " << hRes << " obtaining class factory for DB
31         Object!" << endl;
32         return FALSE;
33     }
34     // 得到对象ID对应的接口
35     hRes = pDBFactory->CreateInstance(NULL, IID_IDB, (void**)&g_gDB);
36     if (FAILED(hRes))
37     {
38         cout << "Error " << hRes << " creating DB Object!" << endl;
39         return FALSE;
40     }
41     // 释放类工厂
42     pDBFactory->Release();
43     // 调用接口方法
44     g_gDB->Open();
45     g_gDB->ExecSQL();
46     g_gDB->Close();
47     g_gDB->Release();
48     return 0;
49 }
```

如上代码中，当客户调用CoGetClassObject函数加载COM组件后，就会自动去调用DllGetClassObject函数生成类工厂指针，然后通过类工厂指针再调用CreateInstance函数生成类实例，返回接口指针。

## 3 COM基础知识

### 3.1 COM对象

COM对象其实就类似于C++中的对象，也就是说某个类的实例，包含了一组数据和操作。

在COM模型中，COM对象的位置对于客户来说是透明的，即客户代码不需要直接初始化一个COM对象，而是COM库通过一个全局标识码GUID去对其进行初始化工作。

GUID是一个128位的标识符，基本保证了COM对象的唯一性，另外COM接口也是用GUID来标识的。

#### 3.1.1 GUID

GUID，又称之为全局唯一标识符，有16个字节，共128位二进制数，可以保证全球范围内不会重复，标识COM对象的GUID称为：CLSID。

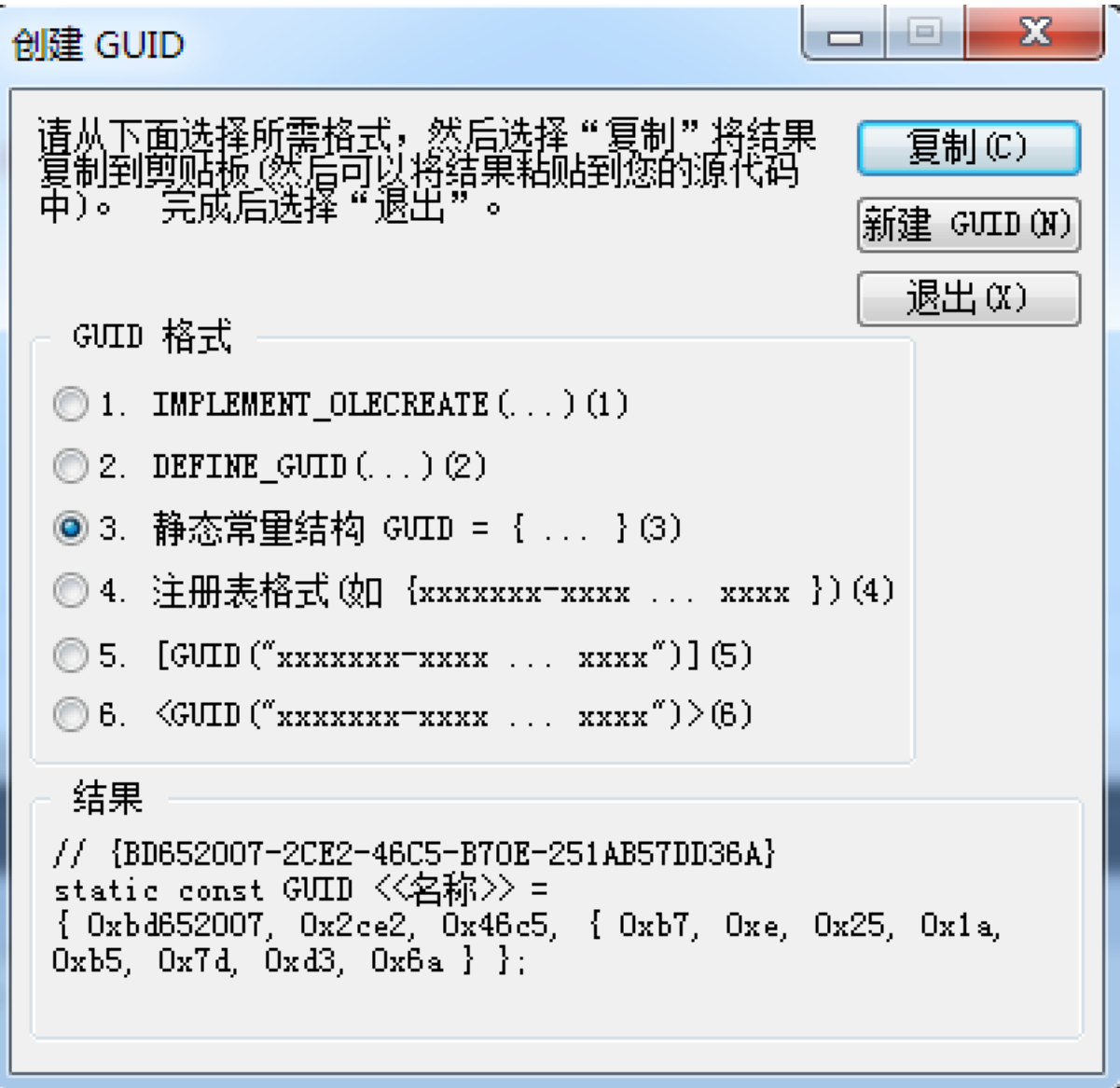
COM组件中，GUID的结构定义如下：

```
1 typedef struct _GUID {  
2     unsigned long Data1; // 随机数  
3     unsigned short Data2; // 和时间有关  
4     unsigned short Data3; // 和时间有关  
5     unsigned char Data4[ 8 ]; // 和网卡MAC有关  
6 } GUID;
```

可以看到这个结构还是挺复杂的，如果我们想要手动生成，这将会是一件非常麻烦的事情，不过我们可以通过VS自带的功能去生成：



一般来说我们选择第三个即可：



初次之外我们害可以借助CoCreateGuid函数来生成这个GUID：

```
1 GUID guid;
2 CoCreateGuid(&guid);
```

从理论上讲，它是不能保证唯一，但重复的可能性非常非常小。有句夸张的说法是：“在每秒钟产生一万亿个GUID的情况下，即使太阳变成白矮星的时候，它仍是唯一的”。

3.2 COM接口

COM接口通常是一组函数的逻辑集合，其命名一般以"I"（大写的i）为前缀，并且继承IUnknown接口；COM对象可以提供多个COM接口，每个接口提供不同的服务，因此COM接口与COM对象一样，都是用GUID来标识的，客户通过GUID来获取接口指针，再通过接口指针获取对应的服务；标识COM接口的GUID称为：IID。

你可以理解为**没有去具体声明和实现AddRef、Release、QueryInterface这三个接口函数的类**就是我们的COM接口，在如上代码中也就是IDB这个类为COM接口。

IUnknown接口是COM的核心，因为所有其他的COM接口都必须从IUnknown继承；它包含三个接口函数：QueryInterface、AddRef和Release，其中**QueryInterface用于接口查询**，从COM对象的一个接口获得另一个接口，一个对象可能实现了多个接口，这样就可以通过QueryInterface在对象多个接口之间跳转从而获得多个接口提供的服务；**AddRef与Release则用于管理COM对象的生命周期**，当COM对象不再使用时需要释放，因此COM使用了引用计数的方法来对对象进行管理，当有一个用户获得接口指针后调用AddRef将引用计数加1，相反，当一个用户用完接口指针后就调用Release来使引用计数减1，这样当引用计数为0时，COM对象就可以从内存中释放；**由于IUnknown提供了接口查询与生命周期控制两个功能，因此COM的每个接口都应该继承于它。**

如下代码是IUnknown的定义，由此我们可以清楚其本质上就是一个含有纯虚函数的抽象类：

```

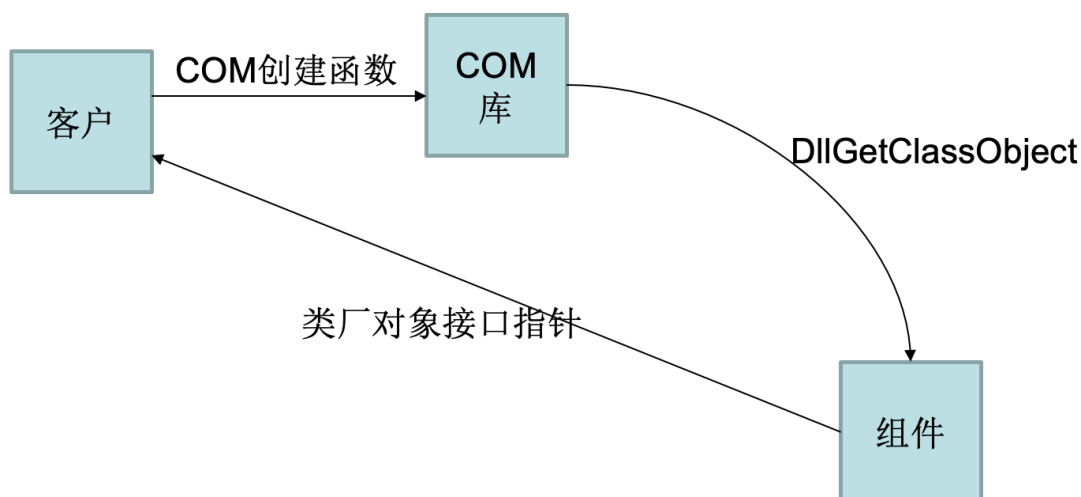
1  class IUnknown
2  {
3  public:
4      virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv)
      = 0;
5      virtual ULONG __stdcall AddRef() = 0;
6      virtual ULONG __stdcall Release() = 0;
7  };

```

### 3.3 COM应用模型

#### 3.3.1 客户/服务模型

如下图就是客户/服务模型：



客户通过COM创建函数获取到COM库，COM库再通过DllGetClassObject获取组件，再通过组件获取类工厂对象接口指针，并用这个指针来调用各个方法。

### 3.3.2 COM库

#### 初始化函数

- 1.CoBuildVersion：获取COM库的版本号
- 2.CoInitialize：初始化COM库
- 3.CoUnInitialize：终止CO服务
- 4.CoFreeUnusedLibraries：释放进程中所有不在使用的组件程序

#### GUID函数

- 1.IsEqualGUID：判断两个GUID是否相等
- 2.IsEqualIID：判断两个IID是否相等
- 3.IsEqualCLSID：判断两个CLSID是否相等
- 4.CLSIDFromProgID：把字符串形式的对像标识转换为CLSID结构形式
- 5.StringFromCLSID：把CLSID结构形式转化为字符串形式
- 6.IIDFromString：把字符串形式的接口标识转换为IID结构形式
- 7.StringFromIID：把IID结构形式转换为字符串形式
- 8.StringFromGUID2：把GUID结构形式转换为字符串形式
- 9.ProgIDFromCLSID：从CLSID获取对象标识

### 3.3.3 对象创建函数

- 1.CoGetClassObject：获取对象的类工厂
- 2.CoCreateInstance：创建COM对象
- 3.CoCreateInstanceEx：创建COM对象，可指定多个接口或远程对象
- 4.CoRegisterClassObject：登记一个对象，以便其它应用程序可以连接到该对象
- 5.CoRevokeClassObject：取消对象的登记操作
- 6.CoDisconnectObject：断开其他应用程序与对象的连接

#### 内存管理函数

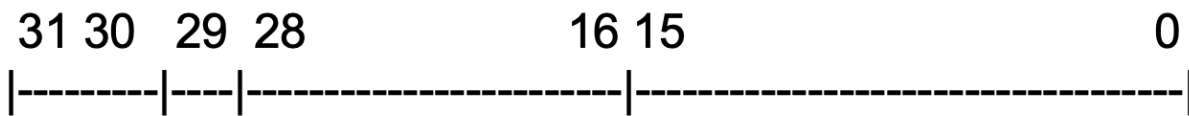
- 1.CoTaskMemAlloc：内存分配函数
- 2.CoTaskMemRealloc：内存重新分配函数
- 3.CoTaskMemFree：内存释放函数
- 4.CoGetMallo：获取COM库的内存管理器接口

### 3.3.4 HRESULT返回值

COM要求所有的方法都会返回一个HRESULT类型的错误号，其就是一个类型定义：

1	<code>typedef LONG HRESULT;</code>
---	------------------------------------

HRESULT类型的返回值反映了函数中的一些情况，其类型定义规范如下：



类别码 (30-31) 反映函数调用结果:

## 00 调用成功

## 01 包含一些信息

## 10 警告

## 11 错误

自定义标记(29位)反映结果是否为自定义标识, 1 为是, 0 则不是; 操作码(16-28位)表示结果操作来源, 在 Windows 平台上, 其定义如下:

```
1 #define FACILITY_WINDOWS      8
2 #define FACILITY_STORAGE      3
3 #define FACILITY_RPC          1
4 #define FACILITY_SSPI         9
5 #define FACILITY_WIN32        7
6 #define FACILITY_CONTROL      10
7 #define FACILITY_NULL         0
8 #define FACILITY_INTERNET     12
9 #define FACILITY_ITF          4
10 #define FACILITY_DISPATCH     2
11 #define FACILITY_CERT         11
```

操作结果码(0-15位)反映操作的状态, WinError.h定义了Win32函数所有可能返回结果。

以下是一些经常用到的返回值和宏定义：

返回值	含义
S_OK	函数执行成功，其值为0(注意，其值与TRUE相反)
S_FALSE	函数执行成功，其值为1
S_FAIL	函数执行失败，失败原因不确定
E_OUTOFMEMORY	函数执行失败，失败原因为内存分配不成功
E_NOTIMPL	函数执行失败，成员函数没有被实现

返回值	含义
E_NOTINTERFACE	函数执行失败，组件没有实现指定的接口

需要注意，我们不能简单地把返回值与S\_OK和S\_FALSE比较，而要用**SUCCEEDED**和**FAILED**宏进行判断。

```
1  #define SUCCEEDED(hr) (((HRESULT)(hr)) >= 0)
2  #define FAILED(hr) (((HRESULT)(hr)) < 0)
```

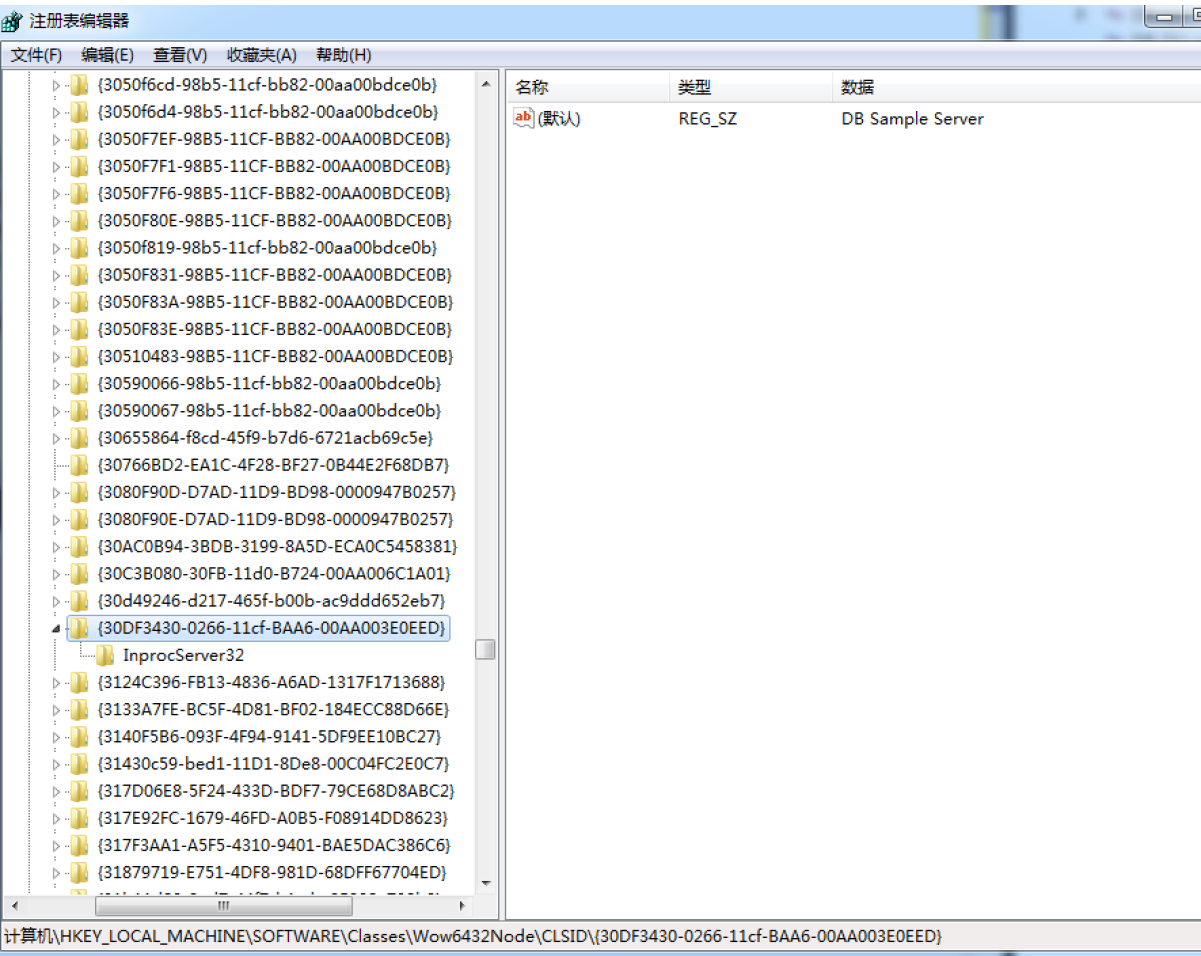
3.3.5 COM与注册表

COM客户和COM组件是相互独立的，COM组件的重要性质之一是位置透明性；当客户程序调用COM对象时不需要考虑COM组件所处的位置，这就是位置无关性；COM位置无关性的实现机制并不深奥，它主要依赖于注册表，这也就是为什么一定要在使用之前对COM组件进行注册的原因；COM库在接到客户程序的请求后，会根据给定的GUID到注册表中检索COM对象的注册条目，并以此来定位COM对象。

COM组件对应二进制文件的存放路径：

```
1  HKEY_CLASSES_ROOT\CLSID\COM组件的CLSID\InprocServer32
2  HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\COM组件的CLSID\InprocServer32
3
4  HKEY_CLASSES_ROOT\Wow6432Node\CLSID\COM组件的CLSID\InprocServer32
5  HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Wow6432Node\CLSID\COM组件的
    CLSID\InprocServer32
```

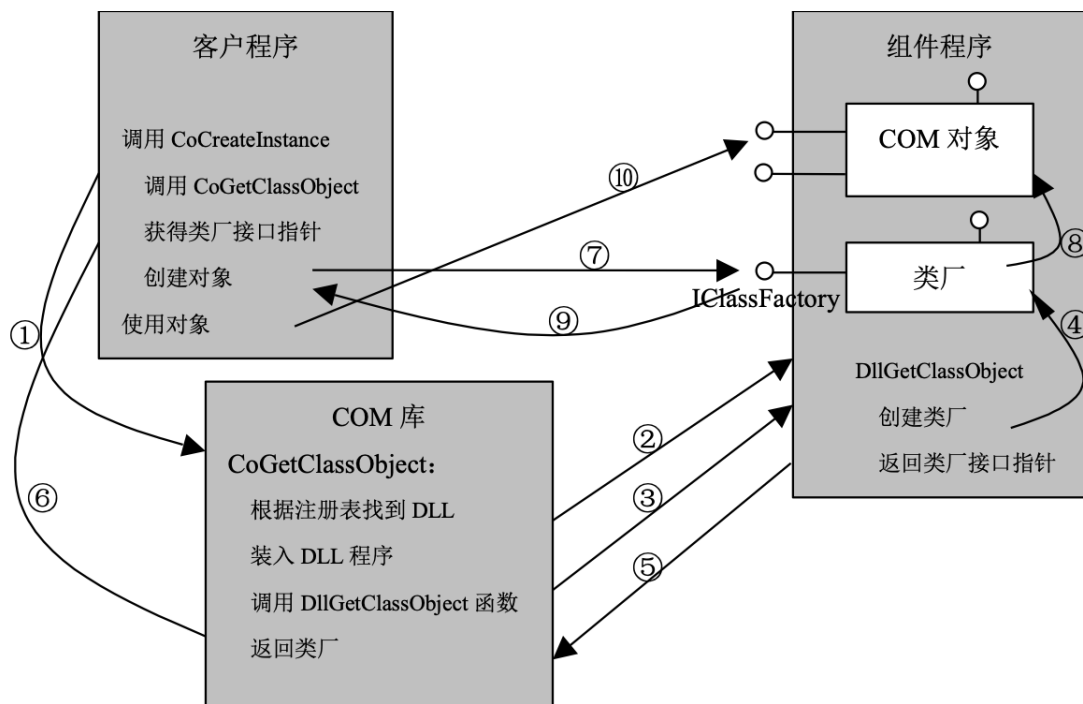




3.4 COM组件

3.4.1 实现类工厂对象

COM对象的创建是通过类工厂来完成的，类工厂是COM对象的生产基地，对应每一个COM类，都有一个类工厂专门用于该COM类的对象创建操作。



类厂本身也是一个COM对象，它支持一个特殊的接口IClassFactory，这个接口的定义如下：

```

1  IClassFactory : public IUnknown
2  {
3  public:
4      virtual /* [local] */ HRESULT STDMETHODCALLTYPE CreateInstance(
5          /* [unique][in] */ IUnknown *pUnkOuter,
6          /* [in] */ REFIID riid,
7          /* [iid_is][out] */ void **ppvObject) = 0;
8      virtual /* [local] */ HRESULT STDMETHODCALLTYPE LockServer(
9          /* [in] */ BOOL fLock) = 0;
10 };
  
```

我们可以看见它继承了IUnknown接口，并且多了两个成员函数：CreateInstance、LockServer。

CreateInstance函数是IClassFactory接口中最重要的函数，它用于创建相应的COM对象，因为每个类厂只针对特定的COM对象，所以CreateInstance成员函数知道该创建什么样的COM对象。

LockServer用于控制组件的生存周期，用于在多客户调用COM时，锁定COM，以免一个客户退出时销毁了COM，那么其他客户的调用将发生错误。

### 3.4.2 实现自动注册

组件程序创建完成之后，必须要通过某种途径把它的信息注册到注册表中，然后客户程序才能根据注册表中的信息对组件程序进行操作。

实现自注册组件必须提供两个导出函数：

```

1  STDAPI DllRegisterServer(void)
2  STDAPI DllUnregisterServer(void)

```

**regsvr32 com组件路径**命令实际是regsvr32.exe调用组件的DllRegisterServer函数，实际注册的操作是在组件的DllRegisterServer函数里完成的，同样执行**regsvr32 /u com组件路径**命令可以进行反注册，实际进行反注册的操作是在组件的DllUnregisterServer函数里完成的。

### 3.4.3 实现自动卸载

只有当组件满足以下两个条件才能被卸载：

1. 组件中对象数为0；
2. 类厂的锁计数为0。

COM中的**CoFreeAllLibraries**函数可以检测当前进程中的所有组件程序，当发现某个组件满足上面两个条件时，就调用FreeLibrary函数把该组件从内存中释放，有两个问题：

1. 谁来调用**CoFreeAllLibraries**函数？是由**客户来调用**，一般在程序空闲的时候调用。
2. CoFreeAllLibraries函数怎么知道满足了上面说的可以卸载的条件？需要**组件导出一个DllCanUnloadNow函数**，如果DllCanUnloadNow函数返回**S\_OK**表示对象可以被卸载。

## 3.5 COM客户

COM客户从只知道CLSID到获取到接口指针必须经过两步：

1. 得到该CLSID的类厂对象；
2. 由类厂对象创建COM对象，返回接口指针给客户。

客户得到对象接口指针后，可以用指针调用接口的成员函数，还可以获得其他接口的指针，从而得到COM对象的所有服务。

## 3.6 COM数据类型

### 3.6.1 COM常见数据类型

COM常见的数据类型如下所示，其中字体加粗部分是我们暂时不了解的：

① CHAR, CHAR\*, BYTE, BYTE\*, SHORT, SHORT\*, USHORT, USHORT\*, INT, INT\*, UINT, UINT\*, LONG, LONG\*, ULONG, ULONG\*, FLOAT, FLOAT\*, DOUBLE, DOUBLE\*, **VARIANT\_BOOL, VARIANT\_BOOL\*, BSTR, BSTR\***, IUnknown\*, IUnknown\*\*, **VARIANT, VARIANT\***

#### VARIANT\_BOOL

数据类型VARIANT\_BOOL的定义如下，其本质就是一个short类型：

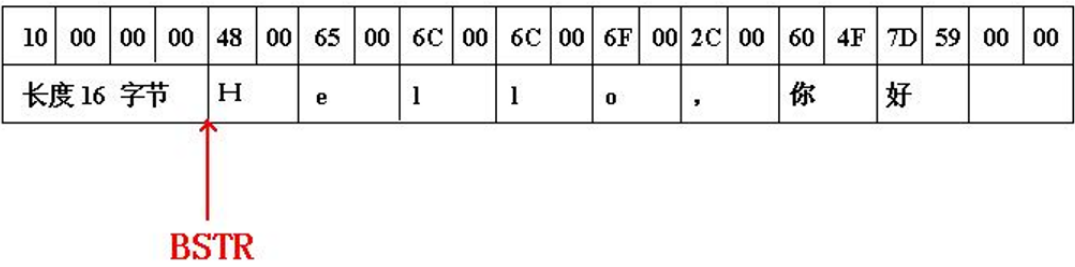
```
1  typedef short VARIANT_BOOL;
```

我们想要使用这个类型的话需要包含一个头文件：

```
1 #include <oidl.h>
```

BSTR

数据类型BSTR是COM中的字符串类型，BSTR\*也就是其的指针；BSTR是指向的是宽字符串的指针，是一个带有字符计数值的字符串，且这个计数值是保存在字符数组的开头的4字节中。



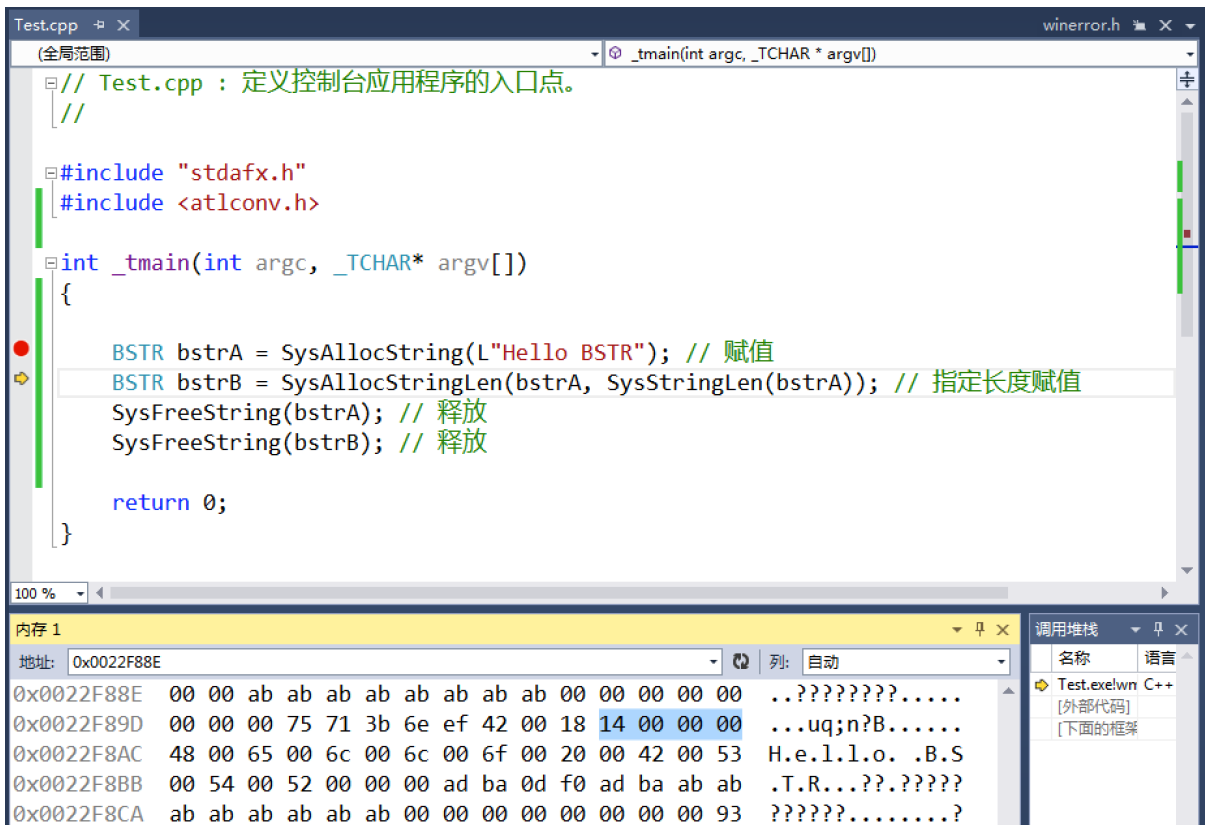
在使用之前需要包含头文件（定义的文件），如果你包含了windows.h或者atlbase.h文件也可以：

```
1 #include <atlconv.h>
```

使用BSTR不能直接赋值，需要借助函数：

```
1 BSTR bstrA = SysAllocString(L"Hello BSTR"); // 赋值
2 BSTR bstrB = SysAllocStringLen(bstrA, SysStringLen(bstrA)); // 指定长度赋值
3 SysFreeString(bstrA); // 释放
4 SysFreeString(bstrB); // 释放
```

我们可以在VS中调试一下观察内存，可以很清楚的看见在Hello前面记录了字符串的长度0x14，也就是20：



如上示例中也直接将BSTR有关的函数给列出来了：

1	BSTR SysAllocString(const OLECHAR *); // 是COM中申请BSTR字符串的方法。
2	BSTR SysAllocStringLen(const OLECHAR *, UINT); // 根据字符串指针与字符个数构造BSTR字符串。
3	UINT SysStringLen(BSTR); // 获取字符串前面的计数值。
4	void SysFreeString(BSTR); // 释放字符串，当COM中的字符串（BSTR）不再使用时，调用该函数。

VARIANT

计算机语言多种多样，COM产生的目的之一就是要跨语言，而VARIANT数据类型就具有了跨语言的特性，同时它可能存储任何的数据类型，说夸张一点，它是万能数据类型。

为实现万能类型的功能，在C++中，VARIANT是一个结构体，该结构体内部又有联合体（联合了多种基本的数据类型），又有变量类型标志VARTYPE vt，可见VARIANT被设置得多么巧妙、合理。

1	typedef /* [wire_marshal] */ struct tagVARIANT VARIANT;
2	
3	struct tagVARIANT
4	{
5	union
6	{
7	struct __tagVARIANT

```
8      {
9      VARTYPE vt;
10     WORD wReserved1;
11     WORD wReserved2;
12     WORD wReserved3;
13     union
14     {
15         LONGLONG llVal;
16         LONG lVal;
17         BYTE bVal;
18         SHORT iVal;
19         FLOAT fltVal;
20         DOUBLE dblVal;
21         VARIANT_BOOL boolVal;
22         _VARIANT_BOOL bool;
23         SCODE scode;
24         CY cyVal;
25         DATE date;
26         BSTR bstrVal;
27         IUnknown *punkVal;
28         IDispatch *pdispVal;
29         SAFEARRAY *parray;
30         BYTE *pbVal;
31         SHORT *piVal;
32         LONG *plVal;
33         LONGLONG *pllVal;
34         FLOAT *pfltVal;
35         DOUBLE *pdblVal;
36         VARIANT_BOOL *pboolVal;
37         _VARIANT_BOOL *pbool;
38         SCODE *pscode;
39         CY *pcyVal;
40         DATE *pdate;
41         BSTR *pbstrVal;
42         IUnknown **ppunkVal;
43         IDispatch **ppdispVal;
44         SAFEARRAY **pparray;
45         VARIANT *pvarVal;
46         PVOID byref;
47         CHAR cVal;
48         USHORT uiVal;
49         ULONG ulVal;
50         ULONGLONG ullVal;
51         INT intVal;
52         UINT uintVal;
53         DECIMAL *pdecVal;
54         CHAR *pcVal;
55         USHORT *puiVal;
56         ULONG *pulVal;
57         ULONGLONG *pullVal;
58         INT *pintVal;
59         UINT *puintVal;
60     struct __tagBRECORD
```

```

61         {
62             PVOID pvRecord;
63             IRecordInfo *pRecInfo;
64         } __VARIANT_NAME_4;
65     } __VARIANT_NAME_3;
66 } __VARIANT_NAME_2;
67     DECIMAL decVal;
68 } __VARIANT_NAME_1;
69 } ;

```

变量类型标志VARTYPE vt，其有对应的值：

```

1  /* VARIANT STRUCTURE
2  *
3  *  VARTYPE vt;
4  *  WORD wReserved1;
5  *  WORD wReserved2;
6  *  WORD wReserved3;
7  *  union {
8  *      LONGLONG      VT_I8
9  *      LONG          VT_I4
10 *      BYTE          VT_UI1
11 *      SHORT         VT_I2
12 *      FLOAT         VT_R4
13 *      DOUBLE        VT_R8
14 *      VARIANT_BOOL  VT_BOOL
15 *      SCODE         VT_ERROR
16 *      CY            VT_CY
17 *      DATE          VT_DATE
18 *      BSTR          VT_BSTR
19 *      IUnknown *    VT_UNKNOWN
20 *      IDispatch *   VT_DISPATCH
21 *      SAFEARRAY *   VT_ARRAY
22 *      BYTE *        VT_BYREF|VT_UI1
23 *      SHORT *       VT_BYREF|VT_I2
24 *      LONG *        VT_BYREF|VT_I4
25 *      LONGLONG *    VT_BYREF|VT_I8
26 *      FLOAT *       VT_BYREF|VT_R4
27 *      DOUBLE *      VT_BYREF|VT_R8
28 *      VARIANT_BOOL * VT_BYREF|VT_BOOL
29 *      SCODE *       VT_BYREF|VT_ERROR
30 *      CY *          VT_BYREF|VT_CY
31 *      DATE *        VT_BYREF|VT_DATE
32 *      BSTR *        VT_BYREF|VT_BSTR
33 *      IUnknown **   VT_BYREF|VT_UNKNOWN
34 *      IDispatch **  VT_BYREF|VT_DISPATCH
35 *      SAFEARRAY **  VT_BYREF|VT_ARRAY
36 *      VARIANT *     VT_BYREF|VT_VARIANT
37 *      PVOID         VT_BYREF (Generic ByRef)
38 *      CHAR          VT_I1
39 *      USHORT        VT_UI2
40 *      ULONG         VT_UI4

```

```

41 *      ULONGLONG      VT_UI8
42 *      INT             VT_INT
43 *      UINT            VT_UINT
44 *      DECIMAL *       VT_BYREF|VT_DECIMAL
45 *      CHAR *          VT_BYREF|VT_I1
46 *      USHORT *        VT_BYREF|VT_UI2
47 *      ULONG *         VT_BYREF|VT_UI4
48 *      ULONGLONG *     VT_BYREF|VT_UI8
49 *      INT *           VT_BYREF|VT_INT
50 *      UINT *          VT_BYREF|VT_UINT
51 *  }
52 */

```

当我们想用VARIANT来保存LONG类型的时候可以这样写：

```

1  VARIANT var;
2  VariantInit(&var); // 初始化
3  var.vt = VT_I4; // 变量类型标志，如上所示VT_I4对应LONG类型
4  var.lVal = 100; // 这里的成员lVal就表示着LONG，这个可以在VARIANT结构体的定义中找到
5
6  VariantClear(&var); // 清空

```

存储学会了，接下来就要从VARIANT中读取存储的值了：

```

1  VARIANT var;
2  VariantInit(&var); // 初始化
3  var.vt = VT_I4; // 变量类型标志，如上所示VT_I4对应LONG类型
4  var.lVal = 100; // 这里的成员lVal就表示着LONG，这个可以在VARIANT结构体的定义中找到
5
6  if (var.vt == VT_I4) // 判断
7  {
8      LONG lVar = var.lVal; // 获取
9  }
10
11 VariantClear(&var); // 清空

```

如果你想要将VARIANT进行数据类型的转换可以使用函数VariantChangeType：

```

1  WINOLEAUTAPI VariantChangeType(
2      _Inout_ VARIANTARG * pvargDest, // 目标VARIANT的指针
3      _In_ const VARIANTARG * pvarSrc, // 源VARIANT的指针
4      _In_ USHORT wFlags, // 强制转化的控制符
5      _In_ VARTYPE vt // 需要强制转化的类型
6  );

```

例如我现在想将LONG类型转为FLOAT类型：

```

1  VARIANT var;

```



```
2 VariantInit(&var); // 初始化
3 var.vt = VT_I4; // 变量类型标志, 如上所示VT_I4对应LONG类型
4 var.lVal = 100; // 这里的成员lVal就表示着LONG, 这个可以在VARIANT结构体的定义中找到
5
6 VariantChangeType(&var, &var, 0, VT_R4);
7
8 VariantClear(&var); // 清空
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    VARIANT var;
    VariantInit(&var); // 初始化
    var.vt = VT_I4; // 变量类型标志, 如上所示VT_I4对应LONG类型
    var.lVal = 100; // 这里的成员lVal就表示着LONG, 这个可以在VARIANT结构体的定义中找到

    VariantChangeType(&var, &var, 0, VT_R4);

    VariantClear(&var); // 清空

    return 0;
}
```

监视 1			调用堆栈
名称	值	类型	名称
var.lVal	100.000000	float	Test.exe!wn (外部代码)

## 4 ATL

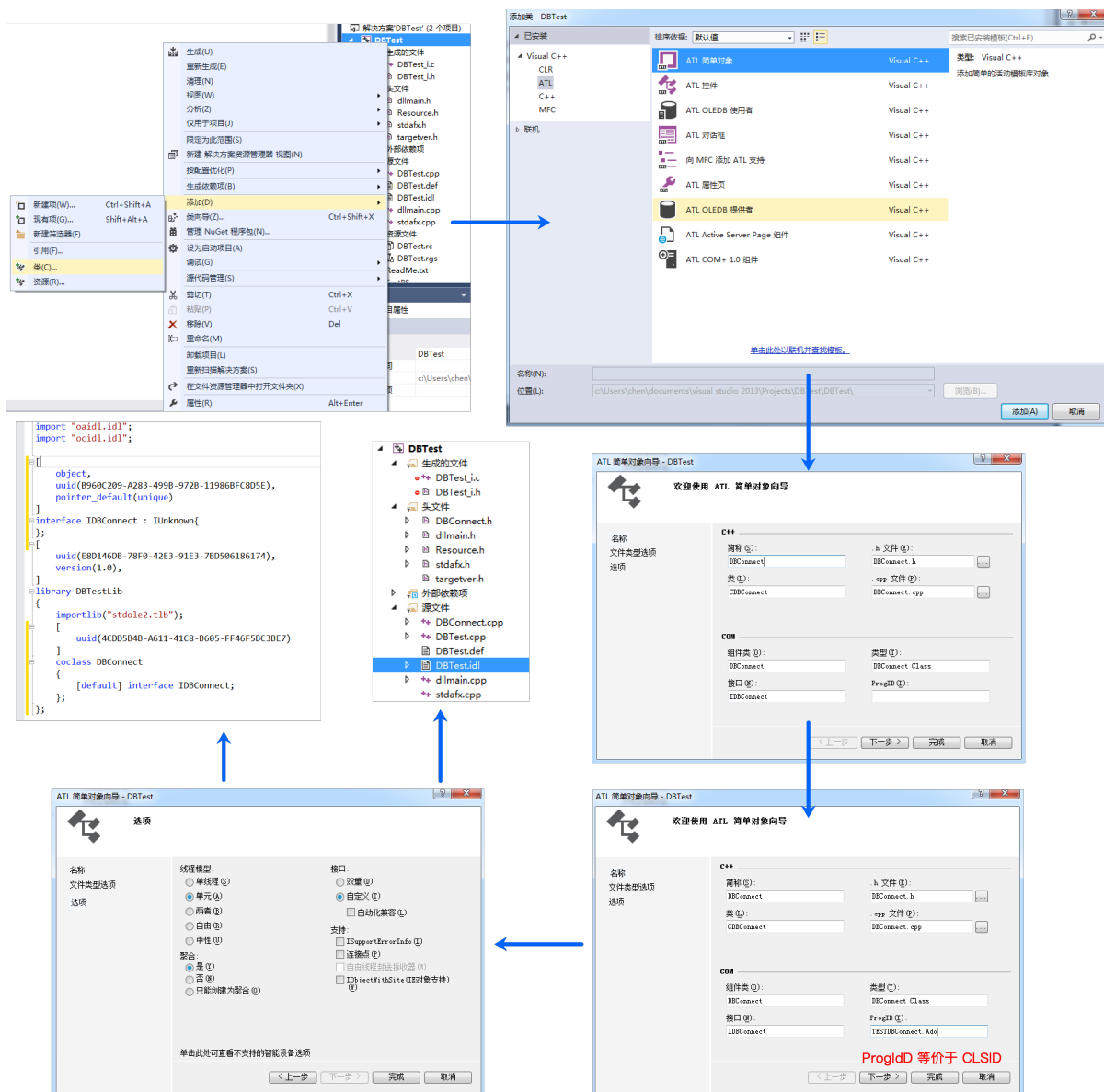
### 4.1 简介

ATL是活动(动态)模板库(ActiveX Template Library)的缩写，它是一套C++模版；ATL的基本目标：**使COM开发尽可能自动化**，这个基本目标决定了ATL只面向COM开发提供支持。

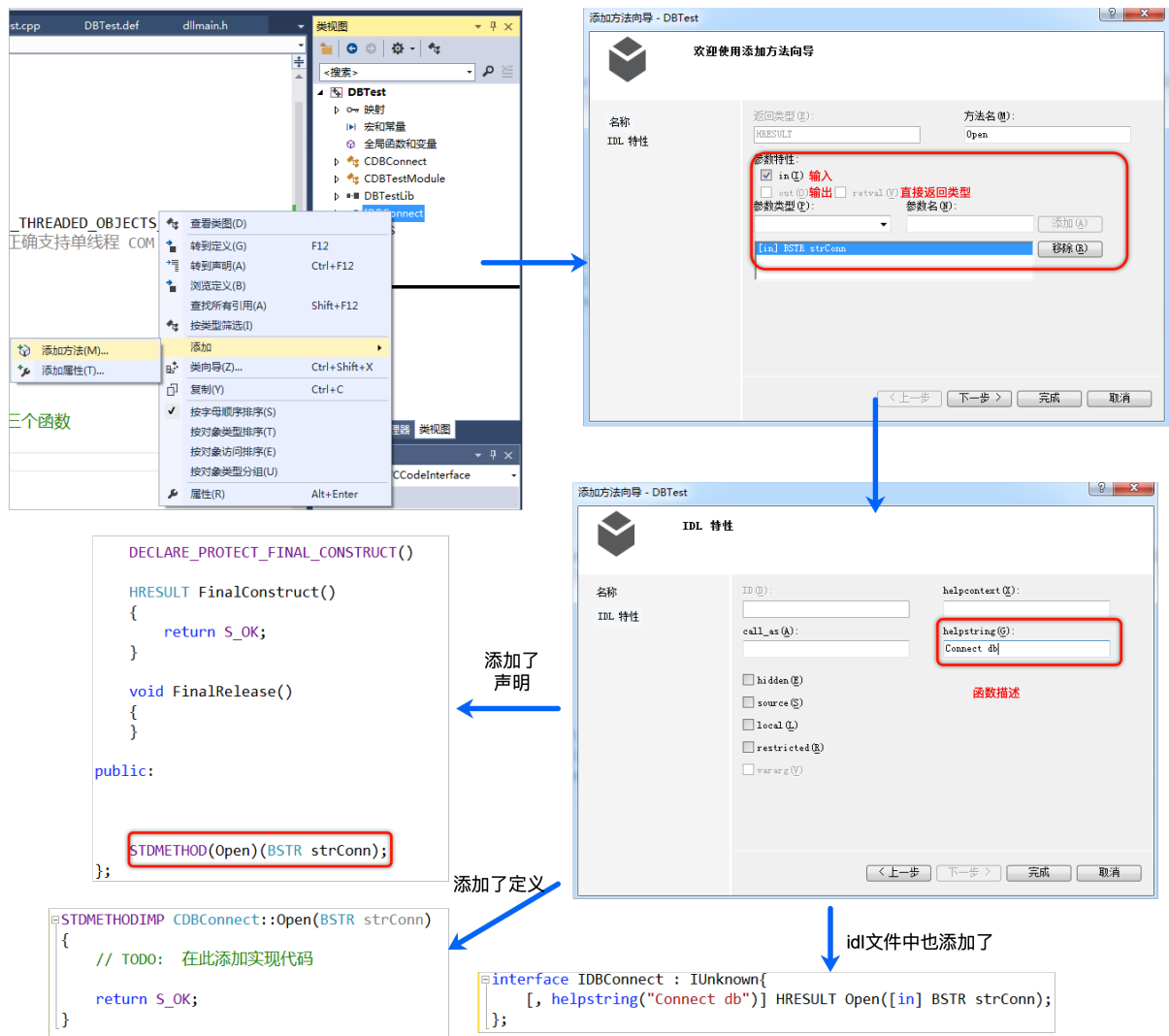
### 4.2 用ATL制作简单对象

创建ATL工程：直接使用VS创建默认的即可。

添加简单对象：通过可视化操作直接去创建，相当于给你创建了头文件（h）和实现文件（cpp），以及在idl文件中添加了内容。



添加接口：在这里创建完成之后在头文件中加入了接口的声明，在定义文件中添加了接口的定义，并且在idl文件中也添加了对应的接口信息。



在添加接口的时候可以选择接口的参数类型，根据这个接口的作用去选择。

至此整个步骤了解之后就会产生的一些疑问：

1. ProgID是什么：**ProgID**等价于**CLSID**
2. 找不到COM类的AddRef、Release、QueryInterface函数的实现
3. 找不到类厂的定义和实现
  - a. 问题2、3在ATL中实际上是换了一个形式出现了，可以在头文件中找到，见如下图注释：

```

class ATL_NO_VTABLE CDBConnect :
public CComObjectRootEx<CComSingleThreadModel>, // 这里实现了IUnknow的三个函数
public CComCoClass<CDBConnect, &CLSID_DBConnect>, // 这里生成了类工厂
public IDBConnect
{
public:
    CDBConnect()
    {
    }
};

DECLARE_REGISTRY_RESOURCEID(IDR_DBCONNECT)

```

- b.
4. 注册表脚本文件有什么作用：注册表脚本文件就是资源文件中的rgs后缀名文件，这里其实是要写入注册表里的信息，当调用注册函数的时候就会使用这里的信息写入到注册表中

```

HKCR
{
    TESTDBConnect.Ado.1 = s 'DBConnect Class'
    {
        CLSID = s '{4CDD5B4B-A611-41C8-B605-FF46F5BC3BE7}'
    }
    TESTDBConnect.Ado = s 'DBConnect Class'
    {
        CurVer = s 'TESTDBConnect.Ado.1'
    }
    NoRemove CLSID
    {
        ForceRemove {4CDD5B4B-A611-41C8-B605-FF46F5BC3BE7} = s 'DBConnect Class'
        {
            ProgID = s 'TESTDBConnect.Ado.1'
            VersionIndependentProgID = s 'TESTDBConnect.Ado'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
            TypeLib = s '{E8D146DB-78F0-42E3-91E3-7BD506186174}'
            Version = s '1.0'
        }
    }
}

```

- a.
5. 后缀为idl的文件是什么文件：IDL全称Interface Description Language，中文为接口描述语言；IDL的主要作用是用来以一种与语言无关的方法来定义一个组件的接口（它的方法和参数），使组件的接口描述在任何语言环境中都认识；IDL是一个文本文件，它的语言语法比较简单，很像C语言；在ATL中IDL文件由MIDL编译，编译后生成TLB文件，类型库以TLB文件形式单独存在，同时也保存在目标文件的资源中，因此我们在引入类型库的时候，即可以指定TLB文件，也可以指定目标文件。
- a. 我们可以看一下这个idl文件，其具体含义可以见如下图注释：

```

// 引入IUnknown和ATL已经定义的其他接口描述文件
import "oaidl.idl";
import "ocidl.idl";

// 一个接口的完整描述
[
    object, // object表示本块描述的是一个接口
    uuid(B960C209-A283-499B-972B-11986BFC8D5E), // 接口的IID, ATL自动生成
    pointer_default(unique) // 定义接口函数中参数所使用指针的默认属性
]
interface IDBConnect : IUnknown{ // 接口叫IDBConnect, 派生自IUnknown
    // 接口函数列表, helpstring里的字符串, 有的工具可以看到这个提示
    [, helpstring("Connect db")] HRESULT Open([in] BSTR strConn);
};

// 类型库的完整描述
[
    uuid(E8D146DB-78F0-42E3-91E3-7BD506186174), // 类型库的IID, ATL自动生成
    version(1.0), // 类型库的版本
]

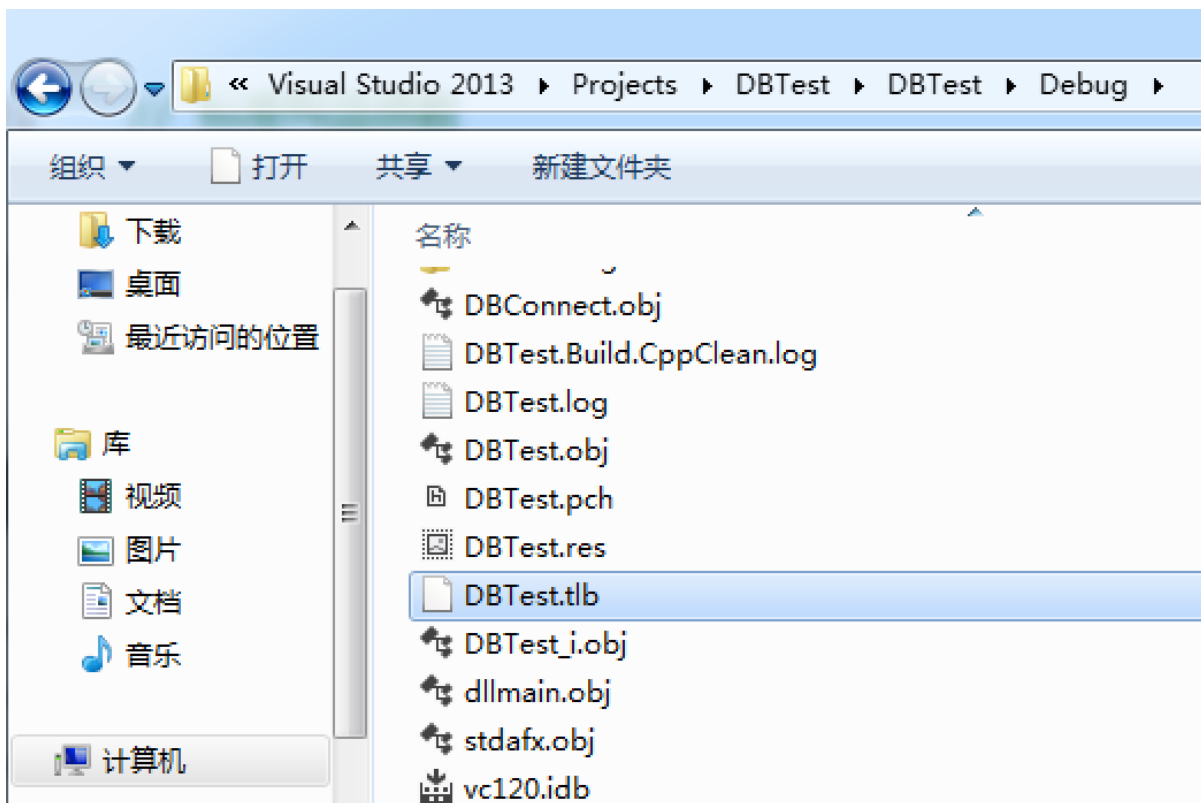
library DBTestLib // 类型库的名字DBTestLib
{
    importlib("stdole2.tlb"); // 引入类型库时的默认命名空间
    [
        // 组件的CLSID, CoCreateInstance函数的第一个参数就是它
        uuid(4CDD5B4B-A611-41C8-B605-FF46F5BC3BE7)
    ]
    coclass DBConnect // COM类DBConnect
    {
        // 接口列表
        [default] interface IDBConnect;
    };
}

```

b. };

6. 组件的CLSID在哪里：如上图注释所示。

最后你只需要编译一下，在项目Debug目录可以找到TLB文件：



### 4.3 MFC调用COM组件

首先你需要创建一个MFC项目，选择基于对话框Dialog创建即可，而后将其他的控件删除留一个按钮便于后续调用，接着你需要引入类型库：

```
DB_ClientDlg.h  DB_ClientDlg.cpp  DB_Client.rc - ID...T_DIALOG - Dialog*  DBTest_i.h  Resource.h  DBTest.rgs  DBConnect.cpp  DBConnect.h
(全局范围)
// DB_ClientDlg.h : 头文件
//

#pragma once

#import "C:\\Users\\chen\\Documents\\Visual Studio 2013\\Projects\\DBTest\\DBTest\\Debug\\DBTest.tlb" no_namespace

// CDB_ClientDlg 对话框
class CDB_ClientDlg : public CDialogEx
{
// 构造
public:
    CDB_ClientDlg(CWnd* pParent = NULL);    // 标准构造函数

// 对话框数据
enum { IDD = IDD_DB_CLIENT_DIALOG };
}
```

1	<code>#import "C:\\Users\\chen\\Documents\\Visual Studio 2013\\Projects\\DBTest\\DBTest\\Debug\\DBTest.tlb" no_namespace</code>
---	---

然后，你需要在头文件中定义好这个接口指针：

```

class CDB_ClientDlg : public CDialogEx
{
// 构造
public:
    CDB_ClientDlg(CWnd* pParent = NULL);    // 标准构造函数

// 对话框数据
    enum { IDD = IDD_DB_CLIENT_DIALOG };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持
public:
    IDBConnectPtr m_pDB;

```

接着在OnInitDialog初始化的时候去初始化COM，根据ProgId创建COM对象：

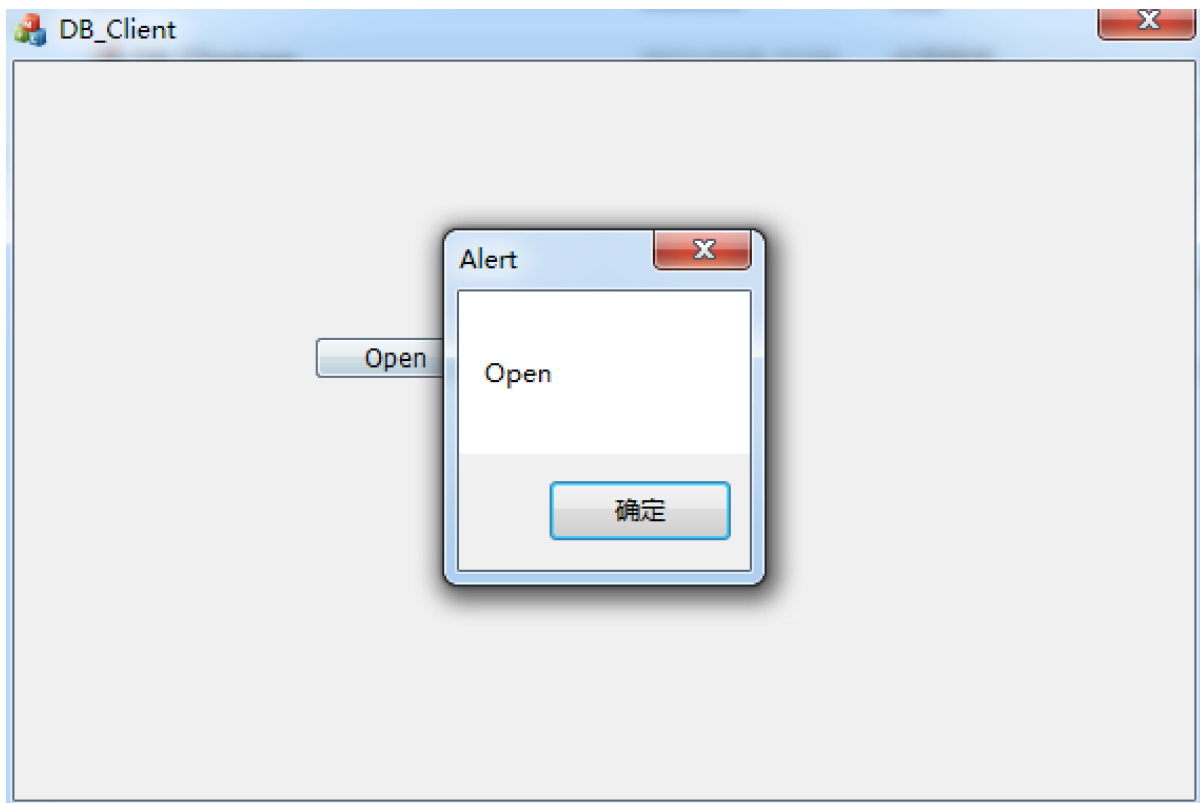
```

1  HRESULT hRes;
2  CoInitialize(NULL);
3  hRes = m_pDB.CreateInstance(L"TESTDBConnect.Ado.1");
4  if (FAILED(hRes))
5  {
6      ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
7  }

```

最后，我们在按钮左键单击事件中去调用接口函数即可：

<pre> void CDB_ClientDlg::OnBnClickedButton1() {     // TODO: 在此添加控件通知处理程序代码     m_pDB-&gt;Open(L""); } </pre>	→	<pre> STDMETHODIMP CDBConnect::Open(BSTR strConn) {     // TODO: 在此添加实现代码     ::MessageBox(NULL, _T("Open"), _T("Alert"), MB_OK);     return S_OK; } </pre>
--	---	---



在这里之所以能成功，是因为我们在编译ATL项目的时候就自动注册了这个COM组件（如果你编译项目失败，可以尝试使用管理员权限打开vs）。



## 5 拓展操作

### 5.1 加属性

#### 5.1.1 添加组件属性

我们现在所模拟的是一个DB的COM组件，而现在我们要添加一个DB连接的状态，这时候就需要通过ATL来添加属性了，如下图所示操作，当你添加一个属性之后在头文件中就会多出两个成员函数的声明以及在实现文件中多出对应的代码实现，最后在idl文件中也可以看见接口函数列表也多出了这些内容。

The image illustrates the process of adding a property to a COM component using Visual Studio's ATL wizard. The wizard is shown in two states: the initial 'Add Property' dialog and the 'IDL Property' configuration window. The resulting code is shown in three parts: the header file (IDL特性), the implementation file (实现文件), and the IDL file (idl文件).

**Header File (IDL特性):**

```
public:
    STDMETHOD(Open)(BSTR strConn);
    STDMETHOD(get_State)(LONG* pVal);
    STDMETHOD(put_State)(LONG newVal);
};

STDMETHODIMP CDBConnect::get_State(LONG* pVal)
{
    // TODO: 在此添加实现代码
    return S_OK;
}

STDMETHODIMP CDBConnect::put_State(LONG newVal)
{
    // TODO: 在此添加实现代码
    return S_OK;
}
```

**Implementation File (实现文件):**

```
STDMETHODIMP CDBConnect::get_State(LONG* pVal)
{
    // TODO: 在此添加实现代码
    return S_OK;
}

STDMETHODIMP CDBConnect::put_State(LONG newVal)
{
    // TODO: 在此添加实现代码
    return S_OK;
}
```

**IDL File (idl文件):**

```
interface IDBConnect : IUnknown { // 接口叫IDBConnect, 派生自IUnknown
    // 接口函数列表, helpstring里的字符串, 有的工具可以看到这个提示
    [helpstring("Connect db")] HRESULT Open([in] BSTR strConn);
    [propget, helpstring("DB State")] HRESULT State([out, retval] LONG* pVal);
    [propput, helpstring("DB State")] HRESULT State([in] LONG newVal);
};
```

通过多出的代码我们可以知道，原来这里通过ATL添加的State属性需要使用get\_State来获取值，以及通过put\_State来修改值，但是这里的最关键的值我们却没有看见，所以我们可以选择在类中去声明一个成员变量：

public:

```
    STDMETHOD(Open)(BSTR strConn);  
    STDMETHOD(get_State)(LONG* pVal);  
    STDMETHOD(put_State)(LONG newVal);  
  
    LONG m_iState;  
};
```

接着我们去实现这两个方法即可：

```
1  STDMETHODIMP CDBConnect::get_State(LONG* pVal)  
2  {  
3      // TODO: 在此添加实现代码  
4      *pVal = m_iState;  
5      return S_OK;  
6  }  
7  
8  STDMETHODIMP CDBConnect::put_State(LONG newVal)  
9  {  
10     // TODO: 在此添加实现代码  
11     m_iState = newVal;  
12     return S_OK;  
13 }
```

接着要在对象创建的时候去初始化这个值，在ATL中初始化的话，你可以在头文件中找到**FinalConstruct**方法，并在该方法体中去初始化：

```
BEGIN_COM_MAP(CDBConnect)
    COM_INTERFACE_ENTRY(IDBConnect)
END_COM_MAP()
```

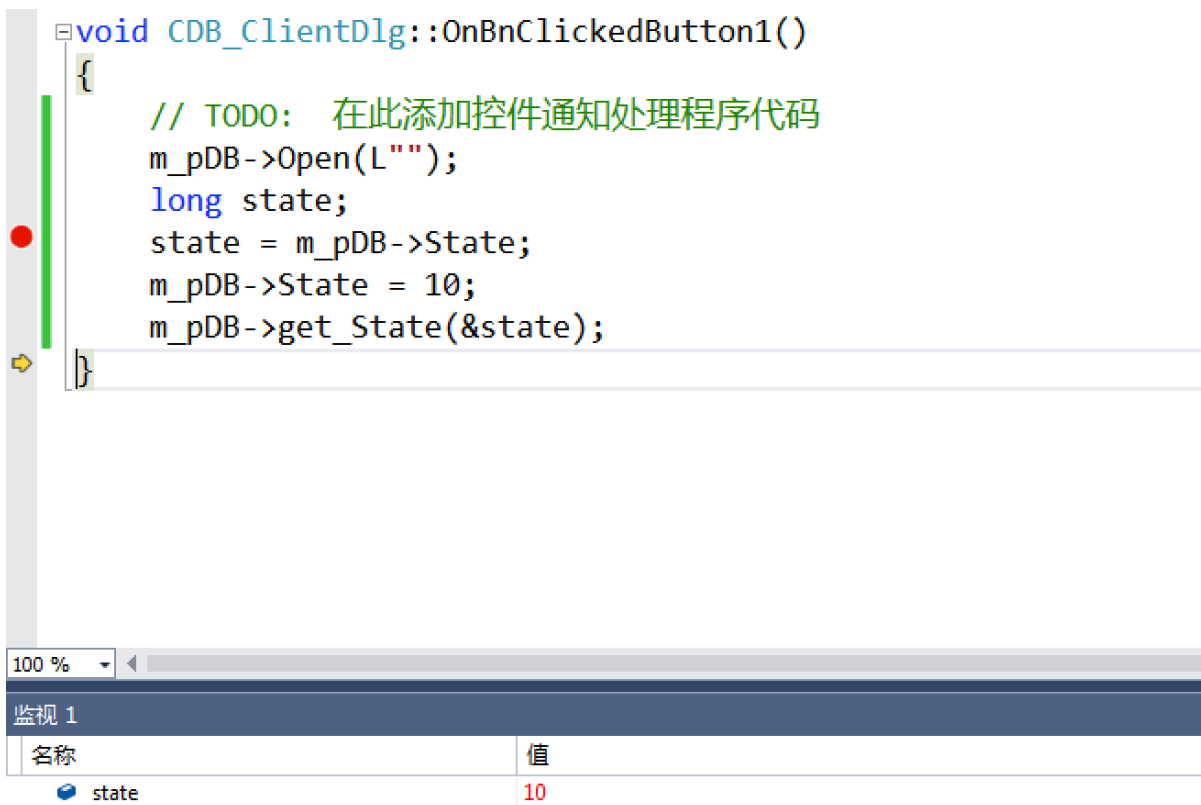
```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

```
HRESULT FinalConstruct()
{
    m_iState = 1;
    return S_OK;
}
```

```
void FinalRelease()
{
}
```

### 5.1.2 客户使用属性

在客户侧去使用这些属性十分简单，直接访问其对应的函数，或直接访问这个属性即可。



## 5.2 多接口

### 5.2.1 实现多接口

我们之前写的COM组件，只有一个类并且只有一个接口，但是我们完全可以让一个COM类提供多个接口。

实现多接口需要手动去做一些改造，首先我们需要在idl文件中加入接口相关的信息，例如这里我加入一个检测的接口，并且接口函数只有一个检测功能：

```
// 一个接口的完整描述
[
    object, // object表示本块描述的是一个接口
    uuid(B960C209-A283-499B-972B-11986BFC8D5E), // 接口的IID, ATL自动生成
    pointer_default(unique) // 定义接口函数中参数所使用指针的默认属性
]
interface IDBConnect : IUnknown{ // 接口叫IDBConnect, 派生自IUnknown
    // 接口函数列表, helpstring里的字符串, 有的工具可以看到这个提示
    [, helpstring("Connect db")] HRESULT Open([in] BSTR strConn);
    [propget, helpstring("DB State")] HRESULT State([out, retval] LONG* pVal);
    [propput, helpstring("DB State")] HRESULT State([in] LONG newVal);
};

// 一个接口的完整描述
/*
// {10546A72-FE0E-4D52-B646-6E24FEE748D7}
static const GUID <<名称>> =
{ 0x10546a72, 0xfe0e, 0x4d52, { 0xb6, 0x46, 0x6e, 0x24, 0xfe, 0xe7, 0x48, 0xd7 } };
*/
object, // object表示本块描述的是一个接口
uuid(10546A72-FE0E-4D52-B646-6E24FEE748D7), // 接口的IID, ATL自动生成(GUID)
pointer_default(unique) // 定义接口函数中参数所使用指针的默认属性

interface IDBConnectChk : IUnknown{ // 接口叫IDBConnectChk, 派生自IUnknown
    // 接口函数列表, helpstring里的字符串, 有的工具可以看到这个提示
    [, helpstring("Connect db")] HRESULT CheckSQL([in] BSTR sql);
};
```

不要忘记在接口列表中也把接口添加上去：

```
library DBTestLib // 类型库的名字DBTestLib
{
    importlib("stdole2.tlb"); // 引入类型库时的默认命名空间
    [
        // 组件的CLSID, CoCreateInstance函数的第一个参数就是它
        uuid(4CDD5B4B-A611-41C8-B605-FF46F5BC3BE7)
    ]
    coclass DBConnect // COM类DBConnect
    {
        // 接口列表
        [default] interface IDBConnect;
        interface IDBConnectChk;
    };
};
```

接着我们需要在DBConnect头文件中声明接口和接口函数：

```

1  class ATL_NO_VTABLE CDBConnect :
2      public CComObjectRootEx<CComSingleThreadModel>, // 这里实现了IUnknown
   的三个函数
3      public CComCoClass<CDBConnect, &CLSID_DBConnect>, // 这里生成了类工厂
4      public IDBConnect,
5      public IDBConnectChk
6  {
7      public:
8
9      ...
10
11     BEGIN_COM_MAP(CDBConnect)
12         COM_INTERFACE_ENTRY(IDBConnect)
13         COM_INTERFACE_ENTRY(IDBConnectChk)
14     END_COM_MAP()
15
16     ...
17
18     public:
19     ...
20         STDMETHODCALLTYPE(CheckSQL)(BSTR sql);
21 };

```

然后还要去实现这个接口函数：

```

1  STDMETHODCALLTYPE CDBConnect::CheckSQL(BSTR sql)
2  {
3      // TODO: 在此添加实现代码
4      ::MessageBox(NULL, _T("CheckSQL"), _T("Alert"), MB_OK);
5      return S_OK;
6  }

```

如果你想让自己的工程看起来规范化，可以替换一下ProgID、Version，当前是1.0版本，可以变成2.0版本，ProgID的最后那个“.1”就表示版本号也可以替换一下。

查找和替换

在文件中查找

在文件中替换

查找内容(N):

TESTDBConnect.Ado.1

(a)+

替换为(P):

TESTDBConnect.Ado.2

(a)+

查找范围(L):

整个解决方案

...

☒ 包括子文件夹(B)

+ 查找选项(O)

+ 结果选项(S)

☐ 全部替换后保持已修改的文件的打开状态(M)

查找下一个(F)

替换(R)

跳过文件(I)

全部替换(A)

## IDL文件

// 类型库的完整描述

```
[
    uuid(E8D146DB-78F0-42E3-91E3-7BD506186174), // 类型库的IID, ATL自动生成
    version(2.0), // 类型库的版本
]
```


## RGS文件

```
ForceRemove {4CDD5B4B-A611-41C8-B605-FF46F5BC3BE7} = s 'DBConnect Class'
{
    ProgID = s 'TESTDBConnect.Ado.2'
    VersionIndependentProgID = s 'TESTDBConnect.Ado'
    InprocServer32 = s '%MODULE%'
    {
        val ThreadingModel = s 'Apartment'
    }
    TypeLib = s '{E8D146DB-78F0-42E3-91E3-7BD506186174}'
    Version = s '2.0'
}
```

## 5.2.2 客户使用多接口

在客户侧去使用多接口也很简单，首先引入类型文件，其次定义接口指针：

```
#import "C:\\Users\\chen\\Documents\\Visual Studio 2013\\Projects\\DBTest\\DBTest\\Debug\\I
```



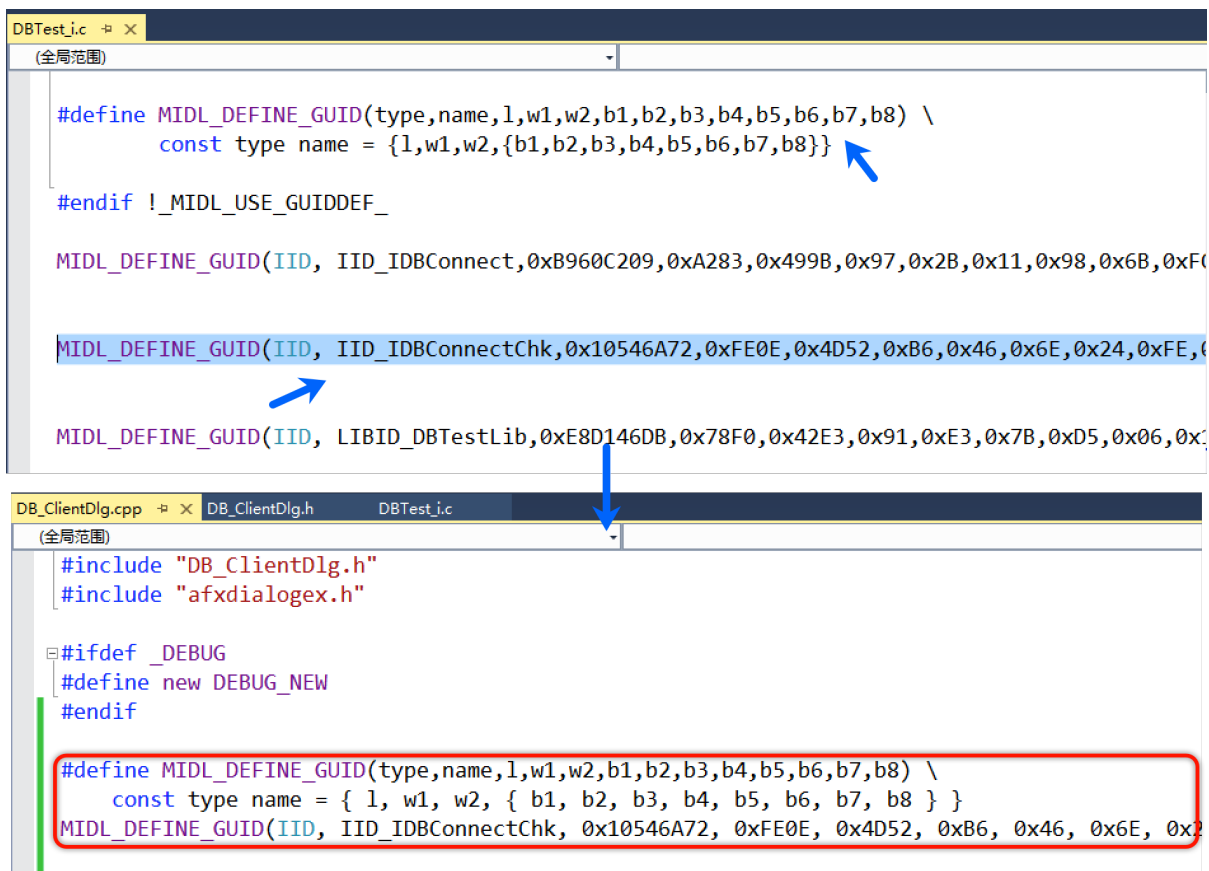
```
public:
    CDB_ClientDlg(CWnd* pParent = NULL); // 标准构造函数

    // 对话框数据
    enum { IDD = IDD_DB_CLIENT_DIALOG };

    protected:
        virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV 支持
    public:
        IDBConnectPtr m_pDB;
        IDBConnectChkPtr m_pDBChk;
```

接着你需要在OnInitDialog函数内去切换接口，这需要使用到QueryInterface方法，你需要传递两个参数，一个是接口的IID，一个是接口指针的地址；接口的IID可以通过COM组件项目的“xxx.i.c”文件找到，这是idl生成的，所以需要编译之后才有内容，接着将MIDL\_DEFINE\_GUID的宏定义以及对应接口的宏调用复制过来，IID\_IDBConnectChk就是最终的接口对应的IID：





再接着我们对应填充参数即可：

```
1 hRes = m_pDB.QueryInterface(IID_IDBConnectChk, &m_pDBChk);
```

// TODO: 在此添加额外的初始化代码

```
HRESULT hRes;
CoInitialize(NULL);
hRes = m_pDB.CreateInstance(L"TESTDBConnect.Ado.2");
hRes = m_pDB.QueryInterface(IID_IDBConnectChk, &m_pDBChk)
if (FAILED(hRes))
{
    ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
}
return TRUE; // 除非将焦点设置到控件，否则返回 TRUE
```

最后我们只需要调用接口函数即可：

```
1 void CDB_ClientDlg::OnBnClickedButton1()
2 {
```

```

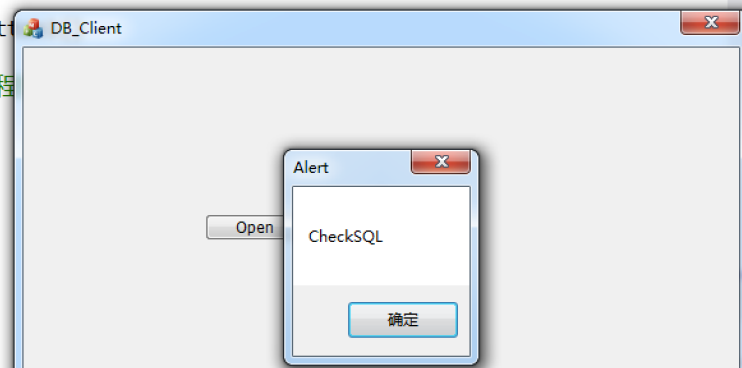
3      // TODO: 在此添加控件通知处理程序代码
4      m_pDBChk->CheckSQL(L"");
5  }

```

```

void CDB_ClientDlg::OnBnClickedButt
{
    // TODO: 在此添加控件通知处理程
    m_pDBChk->CheckSQL(L"");
}

```



最后需要注意的是idl生成的两个文件可以直接被引用，这样你就可以不用去引入类型库文件（tlb文件）了：

📁 DBTest

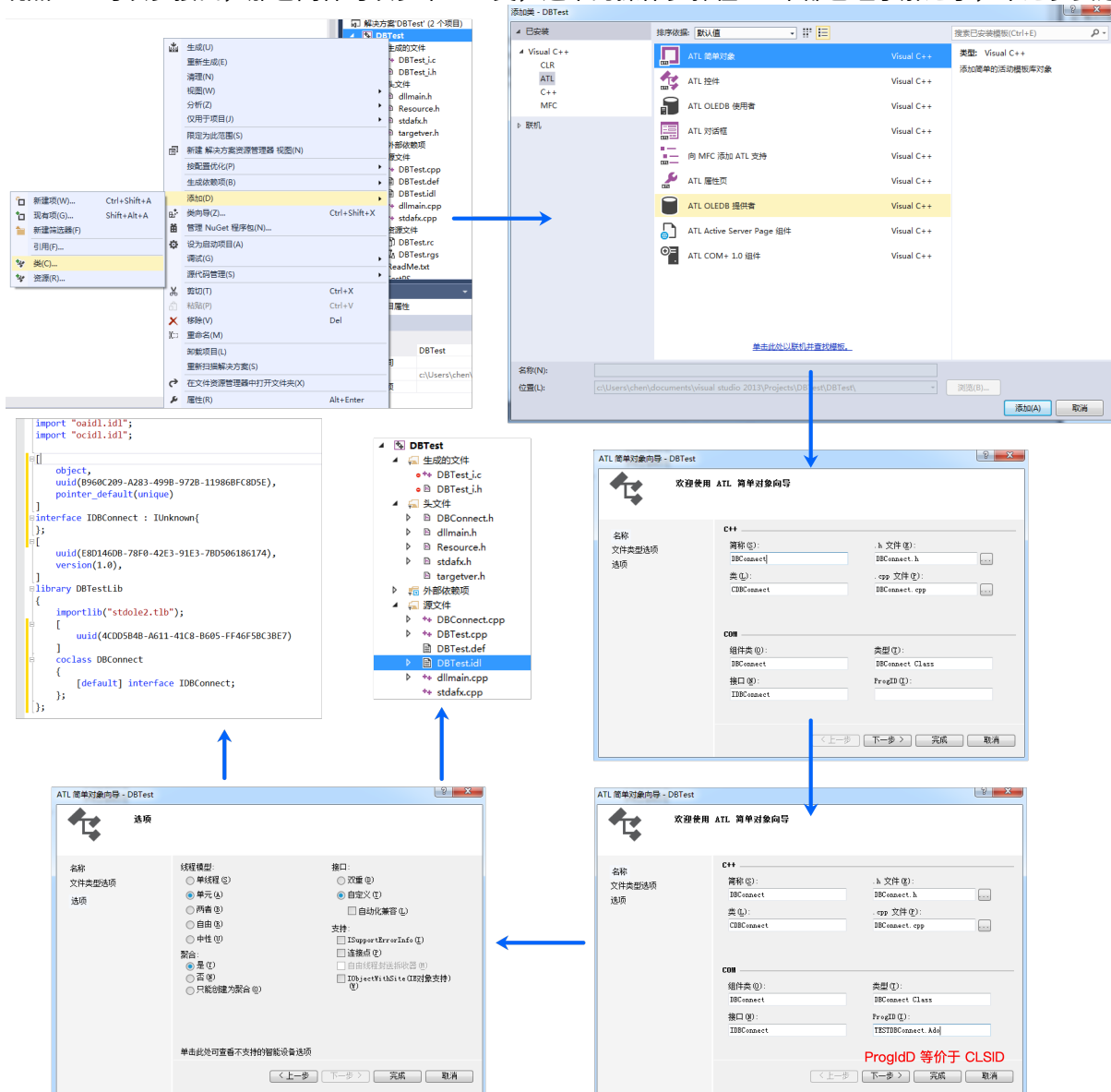
📁 生成的文件

▶ 📄 DBTest\_i.c

▶ 📄 DBTest\_i.h

## 5.3 多COM类

既然COM可以多接口，那也同样可以多个COM类，这个的操作步骤在ATL中都已经了解过了，不过多赘述。



## 6 自动化

### 6.1 简介

编译型语言在编译之前引入类型库(tlb文件)，编译器编译的时候就知道如何编译接口函数的调用了，这种方式我们称为前绑定；而脚本语言是解释执行的，它执行的时候不会知道具体的函数地址，自动化为此诞生了后绑定；**自动化组件其实就是实现了IDispatch（自动化）接口的组件**；解释性语言跟宏语言，要调用COM组件的自定义接口时，都是通过自动化控制程序把自定义接口中的函数名称的字符串跟函数参数传递给IDispatch，让IDispatch间接地去执行自定义接口中的函数；所以本节课所需要学习的就是**IDispatch（自动化）接口**。

### 6.2 IDispatch（自动化）接口

IDispatch接口要实现四个函数：

函数名	作用
HRESULT GetTypeInfoCount()	获取组件中提供几个类型库
HRESULT GetTypeInfo()	调用者通过该函数取得他想要的类型库
HRESULT GetIDsOfNames()	根据函数名称取得函数序号，为调用 Invoke() 做准备
HRESULT Invoke()	根据序号，执行函数

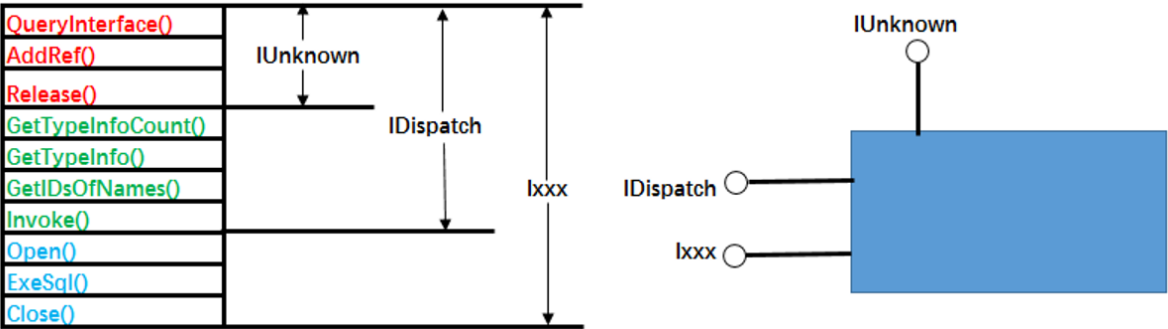
```
interface ITGBRecordset : IDispatch{ GetIDsOfName
    [id(1)] HRESULT Open [in] BSTR bstrCmd, [in] IUnknown* ActiveConnection);
    [propget, id(2)] HRESULT Fields([out, retval] ITGBFields** pVal);
    [propget, id(2)] HRESULT Fields([in] ITGBFields* newVal);
    [propget, id(3)] HRESULT BEOF([out, retval] VARIANT_BOOL* pVal);
    [propget, id(4)] HRESULT BOF();
    [id(5)] HRESULT MoveNext();
};
```

如上图所示受限我们需要使用IDispatch接口的GetIDsOfNames函数去获取函数序号，接着再使用Invoke函数根据序号获取函数的地址然后执行，这样下来执行效率是比较低的，所以ATL从效率出发，实现了一种叫**双接口 (dual)**的接口模式，我们来了解一下。

### 6.3 双接口

#### 6.3.1 示意图

如下是双接口示意图，我们可以看见他有三个部分，分别是IUnknow、IDispatch、自定义接口，所谓双接口，其实是在一个VTAB的虚函数表中容纳了三个接口，因为任何接口都是从IUnknown派生的，所以就不强调IUnknown了，就称之为双接口。



### 6.3.2 场景与优点

如下表所示是在不同场景下使用双接口、原因与使用结果：

使用方式	原因	结果
脚本语言使用组件	解释器只认识IDispatch接口	可以调用，但执行效率最低
编译型语言使用组件	它认识IDispatch接口	可以调用，执行效率比较低
编译型语言使用组件	它装载类型库就认识了Ixxx接口	可以直接调用Ixxx函数，效率最高

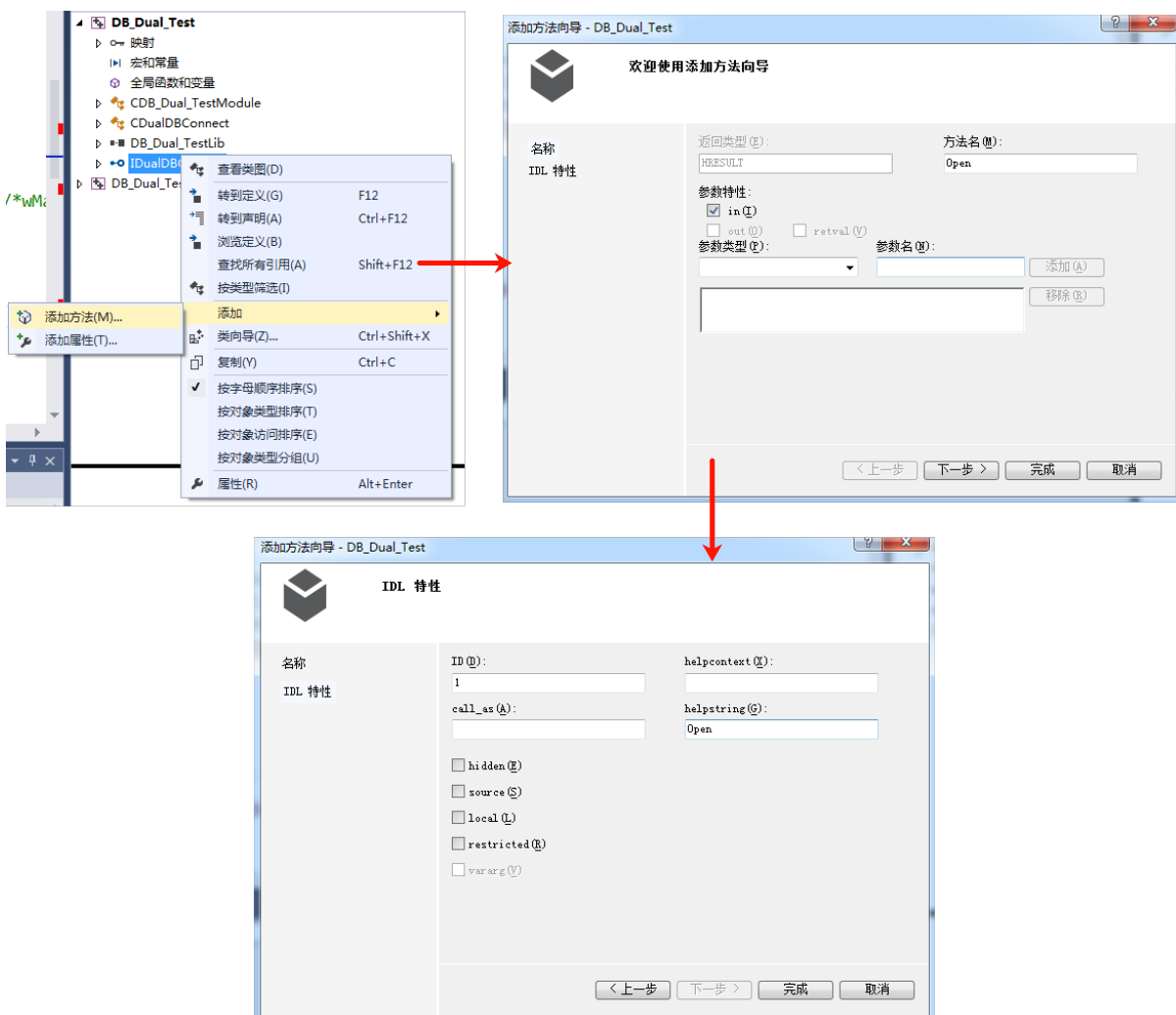
最终我们得到一个结论，那就是双接口即满足脚本语言的使用方便，又满足编译语言的使用高效性，这就是它的优点。

**注：**如果不需要支持脚本语言最好不要用双接口，因为双接口、IDispatch接口只支持自动化的参数类型，使用受到限制，某些情况下很不方便。

### 6.4 使用自动化组件

使用自动化组件并不难，跟之前一样创建ATL项目以及简单对象，只不过在创建对象的时候需要选择接口类型为双重：





当你添加的时候你就会发现，这个方法名会自动赋予ID，这个你可以自定义，也可以不用管它。实现好接口函数之后你只需要编译即可：

```

□ STDMETHODCALLTYPE CDualDBConnect::Open()
{
    // TODO: 在此添加实现代码
    ::MessageBox(NULL, _T("Open"), _T("Alert"), MB_OK);
    return S_OK;
}

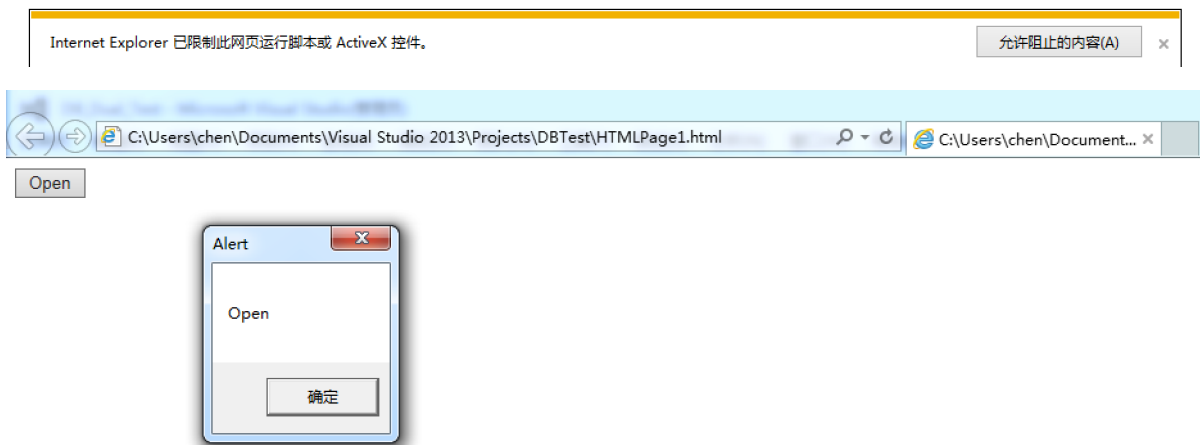
```

### 6.4.1 JavaScript调用

接着我们想要在JavaScript中去调用这个COM组件，编写代码：

```
1 <script type="text/javascript">
2     var myCom = new ActiveXObject("DualDBConnect.Ado.1"); // ProgID
3     function OpenTest() {
4         myCom.Open();
5     }
6 </script>
7 <input type="button" value="Open" onclick="OpenTest()" />
```

接着在IE浏览器中打开这个HTML页面，并允许阻止的内容，单击即可调用接口函数：





## 7 智能类型

之前我们所了解过VARIANT类型，该类型的优点我们都知道是其**可以兼容所有的数据类型**，但同时，我们了解VARIANT类型初始化赋值十分繁琐。

为了使用VARIANT类型更加方便，ATL对VARIANT数据类型做了封装，封装了一个CComVariant类型，它内部维护了一个VARIANT的数据结构，所以我们可以很方便的在大多数情况下**用CComVariant来替代VARIANT**。

### 7.1 CComVariant

CComVariant的数据结构定义可以自行去查看，由于代码过长这里不过多展示，从定义中我们可以看出，其提供了几乎所有数据类型的构造，这就意味着我们初始化CComVariant对象时可以简单的采用：

```
1 CComVariant comVar("ABC");
```

并其提供了所有类型赋值运算符的重载，可以采用下面方式直接赋值：

```
1 comVar = "XYZ";
```

由于它内部维护了一个VARIANT的数据结构，我们也可以使用VARIANT类型的方式：

```
1 CComVariant comVar(123, VT_I4);
```

其他的使用方法都如出一辙，同样，也不过多赘述了。

### 7.2 CComPtr、CComQIPtr

#### 7.2.1 简介

C++在调用COM接口指针时是很危险的，因为使用过程需要每一个使用都严格并且正确地调用AddRef与Release方法。一旦出现问题，就会造成对象不能被正常释放或者对象被重复删除。

CComPtr、CComQIPtr是ATL为了解决COM引用计数问题提供的一个类模版，因为它的使用和行为上类似与一个接口指针，所以有一个通俗易懂的名字：智能指针。

这两个模版类都继承自CComPtrBase，不同之处在于CComQIPtr能在必要的时候自动的对所需接口进行查询（如：对与此智能指针参数化类型不同的指针赋值时，会自动查询是否有所需的接口）。

**CComPtrBase类封装了CComPtr和CComQIPtr中公共的大多数函数**，从而实现代码的复用。

#### 7.2.2 比较

智能指针与接口指针的比较如下：

1. **CComPtr<IDBConnect> spDBConn;**创建了一个智能指针，其实它是一个类对象，对象内部有一个IDBConnect\*的指针变量，而IDBConnect\* pDBConn就是一个指针；
2. 二者操作的用法和意义一样；

3. 智能指针不能执行AddRef与Release操作，因为智能指针封装了COM接口指针的AddRef与Release操作，会智能判断何时内部调用COM接口的AddRef，何时调用COM接口的Release。

### 7.2.3 注意

智能指针使用的注意点：

1. 智能指针已经保证了AddRef和Release的正确调用，所以不需要，也不能够再调用AddRef和Release；
2. 如果要释放一个智能指针，直接给它赋NULL,这样内部才COM接口指针会自动执行Release操作，来减少应用计数；
3. 当对智能指针取地址时(&运算符操作)，要确保智能指针为NULL，因为&是要返回内部的COM接口指针的，如果不为NULL，则旧的COM接口指针将没有执行Release而直接赋值了一个旧的COM接口指针。

### 7.2.4 使用

在使用智能指针需要包含一个头文件：

```
1  #include <atlbase.h>
```

```
#import "C:\\Users\\chen\\Documents\\Visual Studio 2013\\Projects\\DB_Dual_Test\\DB_Dual_
#include <atlbase.h>
```

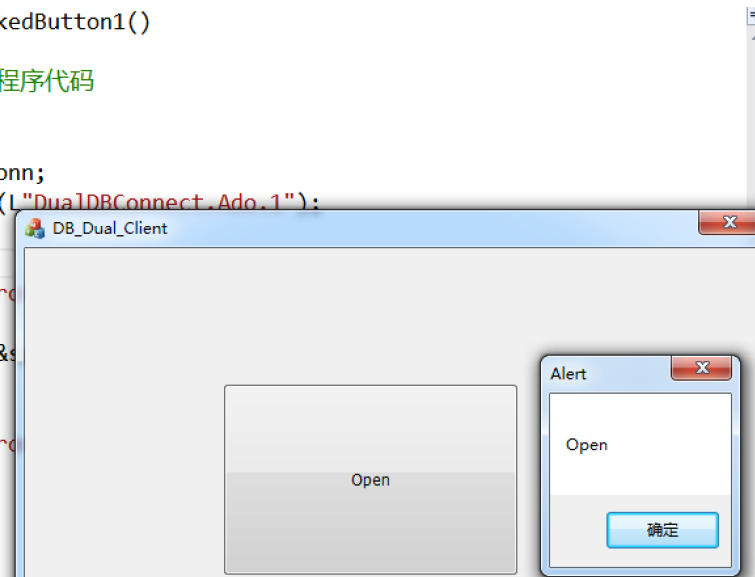
接着调用即可：

```
1  HRESULT hRes;
2  CComPtr<IUnknown> spUnk;
3  CComPtr<IDualDBConnect> spDBConn;
4  hRes = spUnk.CoCreateInstance(L"DualDBConnect.Ado.1"); // 创建IUnknown
5  if (FAILED(hRes))
6  {
7      ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
8  }
9  hRes = spUnk->QueryInterface(&spDBConn); // 寻找IDualDBConnect
10 if (FAILED(hRes))
11 {
12     ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
13 }
14 spDBConn->Open(); // 调用
```

```

void CDB_Dual_ClientDlg::OnBnClickedButton1()
{
    // TODO: 在此添加控件通知处理程序代码
    HRESULT hRes;
    CComPtr<IUnknown> spUnk;
    CComPtr<IDualDBConnect> spDBConn;
    hRes = spUnk.CoCreateInstance(L"DualDBConnect.Ado.1");
    if (FAILED(hRes))
    {
        ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
    }
    hRes = spUnk->QueryInterface(&IID_IDualDBConnect, (void**)&spDBConn);
    if (FAILED(hRes))
    {
        ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
    }
    spDBConn->Open();
}

```



我们也可以使用CComQIPtr，这个智能指针可以直接赋值而不需要使用QueryInterface方法：

```

1  HRESULT hRes;
2  CComPtr<IUnknown> spUnk;
3  CComQIPtr<IDualDBConnect> spDBConn;
4  hRes = spUnk.CoCreateInstance(L"DualDBConnect.Ado.1");
5  if (FAILED(hRes))
6  {
7      ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
8  }
9  spDBConn = spUnk;
10 if (!spDBConn)
11 {
12     ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);
13 }
14 spDBConn->Open();

```

需要注意CComQIPtr不能定义IUnknown指针，所以只需要修改spDBConn为CComQIPtr即可。

同样你可以简化这种写法，直接使用接口的IID即可：

```

1  // MIDL_DEFINE_GUID(IID,
2  // IID_IDualDBConnect, 0x8D7C23E2, 0xEDE3, 0x4E10, 0x96, 0x6F, 0x2D, 0x56, 0x2C, 0x93,
3  // 0xD1, 0xFA);
4  // 将这个转为IID格式即可
5  IID IID_IDualDBConnect = { 0x8D7C23E2, 0xEDE3, 0x4E10, { 0x96, 0x6F, 0x2D,
6  0x56, 0x2C, 0x93, 0xD1, 0xFA } };
7
8  HRESULT hRes;
9  CComQIPtr<IDualDBConnect, &IID_IDualDBConnect> spDBConn;
10 hRes = spDBConn.CoCreateInstance(L"DualDBConnect.Ado.1");
11 if (FAILED(hRes))
12 {

```

```
10         ::MessageBox(NULL, _T("Error"), _T("Title"), MB_OK);  
11     }  
12     spDBConn->Open();
```