COM 组件设计与应用（一）
起源及复合文件

一、前言

公元一九九五年某个夜黑风高的晚上，我的一位老师跟我说：“小杨呀，以后写程序就和搭积木一样啦。你赶快学习一些 OLE 的技术吧.....”，当时我心里就寻思 ：“开什么玩笑？搭积木方式写程序？再过 100 年吧.....”，但作为一名听话的好学生，我开始在书店里“踅摸”（注 1）有关 OLE 的书籍（注 2）。功夫不负有心人，终于买到了我的第一本 COM 书《OLE2 高级编程技术》，这本 800 多页的大布头花费了我 1/5 的月工资呀.....于是开始日夜耕读.....

功夫不负有心人，我坚持读完了全部著作，感想是：这本书，在说什么呐？
功夫不负有心人，我又读完了一遍大布头，感想是：咳~~~，没懂！

功夫不负有心人，我再，我再,我再读 ... 感想是：哦~~~，读懂了一点点啦，哈哈哈。

....

功夫不负有心人，我终于，我终于懂了。

800 页的书对现在的我来说，其实也就 10 几页有用。到这时候才体会出什么叫“书越读越薄”的道理了。到后来，能买到的书也多了，上网也更方便更便宜了.....

为了让 VCKBASE 上的朋友，不再经历我曾经的痛苦、不再重蹈我“无头苍蝇”般探索的艰辛、为了 VCKBASE 的蓬勃发展、为了中国软件事业的腾飞（糟糕，吹的太也高了）.....我打算节约一些在 BBS 上赚分的时间，写个系列论文，就叫“COM 组件设计与应用”吧。今天是第一部分——起源。

二、文件的存储

传说 350 年前，牛顿被苹果砸到了头，于是发现了万有引力。但到了二十一世纪的现在，任何一个技术的发明和发展，已经不再依靠圣人灵光的一闪。技术的进步转而是被社会的需求、商业的利益、竞争的压力、行业的渗透等推动的。微软在 Windows 平台上的组件技术也不例外，它的发明，有其必然因素。什么是这个因素那？答案是——文件的存储。

打开记事本程序，输入了一篇文章后，保存。——这样的文件叫“非结构化文件”；

打开电子表格程序，输入一个班的学生姓名和考试成绩，保存。——这样的文件叫“标准结构化文件”；

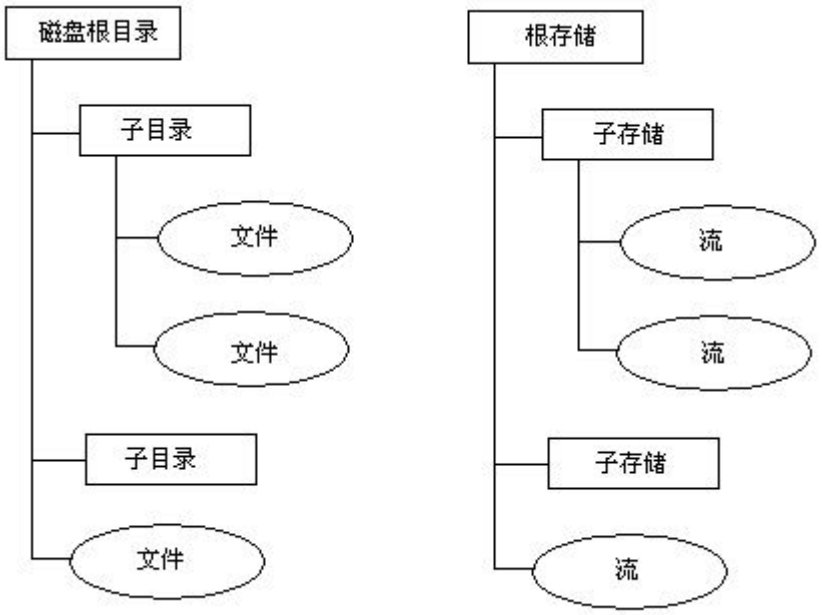
在我们写的程序中，需要把特定的数据按照一定的结构和顺序写到文件中保存。——这样的文件叫“自定义结构化文件”；（比如 *.bmp 文件）

以上三种类型的文件，大家都见的多了。那么文件存储就依靠上述的方式能满足所有的应用需求吗？恩~~~，至少从计算机发明后的 50 多年来，一直是够用的了。嘿嘿，下面看看商业利益的推动作用，对文件 的存储形式产生了什么变化吧。30 岁以上的朋友，我估计以前都使用过以下几个著名的软件：WordStar（独霸 DOS 下的英文编辑软件），WPS（裘伯君写的中文编辑软件，据说当年的市场占有率高达 90%，各种计算机培训班的必修课程），LOTUS-123（莲花公司出品的电子表格软件）.....

微软在成功地推出 Windows 3.1 后，开始垂涎桌面办公自动化软件领域。微软的 OFFICE 开发部门，各小组分别独立地开发了 WORD 和 EXCEL 等软件，并采用“自定义结构”方式，对文件进行存储。在激烈的市场竞争下，为了打败竞争对手，微软自然地产生了一个念头------如果我能在 WORD 程序中嵌入 EXCEL，那么用户在购买了我 WORD 软件的情况下，不就没有必要再买 LOTUS-123 了吗？！“恶毒”（中国微软的同志们看到了这个词，不要激动，我是加了引号的呀）的计划产生后，他们开始了实施工作，这就是 COM 的前身 OLE 的起源（注 3）。但立刻就遇到了一个严重的技术问题：需要把 WORD 产生的 DOC 文件和 EXCEL 产生的 XLS 文件保存在一起。

方案	优点	缺点
建立一个子目录，把 DOC、XLS 存储在这同一个子目录中。	数据隔离性好，WORD 不用了解 EXCEL 的存储结构；容易扩展。	结构太松散，容易造成数据的损坏或丢失。 不易携带。
修改文件存储结构，在 DOC 结构基础上扩展出包容 XLS 的结构。	结构紧密，容易携带和统一管理。	WORD 的开发人员需要通晓 EXCEL 的存储格式；缺少扩展性，总不能新加一个类型就扩展一下结构吧？！

以上两个方案，都有严重的缺陷，怎么解决那？如果能有一个新方案，能够合并前两个方案的优点，消灭缺点，该多好呀.....微软是作磁盘操作系统起家的，于是很自然地他们提出了一个非常完美的设计方案，那就是把磁盘文件的管理方式移植到文件中了------复合文件，俗称“文件中的文件系统”。连微软当年都没有想到，就这么一个简单的想法，居然最后就演变出了 COM 组件程序设计的方法。可以说，复合文件是 COM 的基石。下图是磁盘文件组织方式与复合文件组织方式的类比图：



图一、左侧表示一个磁盘下的文件组织方式，右侧表示一个复合文件内部的数据组织方式。

三、复合文件的特点

1. 复合文件的内部是使用指针构造的一棵树进行管理的。编写程序的时候要注意，由于使用的是单向指针，因此当做定位操作的时候，向后定位比向前定位要快；
2. 复合文件中的“流对象”，是真正保存数据的空间。它的存储单位为 512 字节。也就是说，即使你在流中只保存了一个字节的数据，它也要占据 512 字节的文件空间。啊~~~，这也太浪费了呀？不浪费！因为文件保存在磁盘上，即使一个字节也还要占用一个“簇”的空间那；
3. 不同的进程，或同一个进程的不同线程可以同时访问一个复合文件的不同部分而互不干扰；
4. 大家都有这样的体会，当需要往一个文件中插入一个字节的话，需要对整个文件进行操作，非常烦琐并且效率低下。而复合文件则提供了非常方便的“增量访问”能力；
5. 当频繁地删除文件，复制文件后，磁盘空间会变的很零碎，需要使用磁盘整理工具进行重新整合。和磁盘管理非常相似，复合文件也会产生这个问题，在适当的时候也需要整理，但比较简单，只要调用一个函数就可以完成了。

四、浏览复合文件

VC6.0 附带了一个工具软件“复合文件浏览器”，文件名是“vc 目录\Common\Tools\DFView.exe”。为了方便使用该程序，可以把它加到工具(tools)菜单中。方法是：Tools\Customize...\Tools 卡片中增加新的项目。运行 DFView.exe，就可以打开一个复合文件进行观察了（注 4）。但奇怪的是，在 Microsoft Visual Studio .NET 2003 中，

我反而找不到这个工具程序了,汗！不过这恰好提供给大家一个练习的机会，在你阅读完本篇文章并掌握了编程方法后，自己写一个“复合文件浏览编辑器”程序，又练手了，还有实用的价值。

五、复合文件函数

复合文件的函数和磁盘目录文件的操作非常类似。所有这些函数，被分为 3 种类型：WIN API 全局函数，存储 IStorage 接口函数，流 IStream 接口函数。什么是接口？什么是接口函数？以后的文章中再陆续介绍，这里大家只要把“接口”看成是完成一组相关操作功能的函数集合就可以了。

WIN API 函数	**功能说明**
StgCreateDocfile()	建立一个复合文件，得到根存储对象
StgOpenStorage()	打开一个复合文件，得到根存储对象
StgIsStorageFile()	判断一个文件是否是复合文件
IStorage 函数	**功能说明**
CreateStorage()	在当前存储中建立新存储，得到子存储对象
CreateStream()	在当前存储中建立新流，得到流对象
OpenStorage()	打开子存储，得到子存储对象
OpenStream()	打开流，得到流对象
CopyTo()	复制存储下的所有对象到目标存储中，该函数可以实现“整理
MoveElementTo()	移动对象到目标存储中
DestoryElement()	删除对象
RenameElement()	重命名对象
EnumElements()	枚举当前存储中所有的对象
SetElementTimes()	修改对象的时间
SetClass()	在当前存储中建立一个特殊的流对象，用来保存 CLSID（注 5）
Stat()	取得当前存储中的系统信息
Release()	关闭存储对象
IStream 函数	**功能说明**
Read()	从流中读取数据
Write()	向流中写入数据

Write()	向流中写入数据
Seek()	定位读写位置
SetSize()	设置流尺寸。如果预先知道大小，那么先调用这个函数，可以提高性能
CopyTo()	复制流数据到另一个流对象中
Stat()	取得当前流中的系统信息
Clone()	克隆一个流对象，方便程序中的不同模块操作同一个流对象
Release()	关闭流对象
WIN API 补充函数	**功能说明**
WriteClassStg()	写 CLSID 到存储中，同 IStorage::SetClass()
ReadClassStg()	读出 WriteClassStg() 写入的 CLSID，相当于简化调用 IStorage::Stat()
WriteClassStm()	写 CLSID 到流的开始位置
ReadClassStm()	读出 WriteClassStm()写入的 CLSID
WriteFmtUserTypeStg()	写入用户指定的剪贴板格式和名称到存储中
ReadFmtUserTypeStg()	读出 WriteFmtUserTypeStg()写入的信息。方便应用程序快速判断是否是它需要的格式数据。
CreateStreamOnHGlobal()	内存句柄 HGLOBAL 转换为流对象
GetHGlobalFromStream()	取得 CreateStreamOnHGlobal()调用中使用的内存句柄

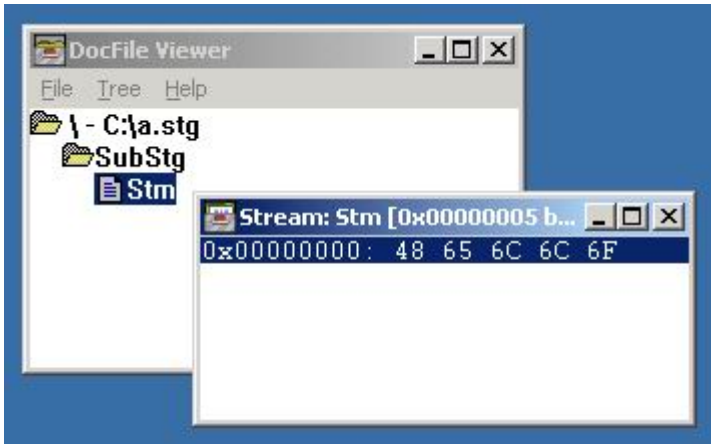
为了让大家快速地浏览和掌握基本方法，上面所列表的函数并不是全部，我省略了“事务”函数和未实现函数部分。更全面的介绍，请阅读 MSDN。

下面程序片段，演示了一些基本函数功能和调用方法。

示例一：建立一个复合文件，并在其下建立一个子存储，在该子存储中再建立一个流，写入数据。

```
void SampleCreateDoc()
{
        ::CoInitialize(NULL);       // COM 初始化
                                    // 如果是 MFC 程序，可以使用 AfxOleInit()替代

        HRESULT hr;                 // 函数执行返回值
        IStorage *pStg = NULL;      // 根存储接口指针
        IStorage *pSub = NULL;      // 子存储接口指针
        IStream *pStm = NULL;       // 流接口指针
```

```
    hr = ::StgCreateDocfile(    // 建立复合文件
            L"c:\\a.stg",       // 文件名称
            STGM_CREATE | STGM_WRITE | STGM_SHARE_EXCLUSIVE,     // 打开方式
            0,          // 保留参数
            &pStg);  // 取得根存储接口指针
    ASSERT( SUCCEEDED(hr) );    // 为了突出重点，简化程序结构，所以使用了断言。
                                // 在实际的程序中则要使用条件判断和异常处理

    hr = pStg->CreateStorage(  // 建立子存储
            L"SubStg",          // 子存储名称
            STGM_CREATE | STGM_WRITE | STGM_SHARE_EXCLUSIVE,
            0,0,
            &pSub);             // 取得子存储接口指针
    ASSERT( SUCCEEDED(hr) );

    hr = pSub->CreateStream(    // 建立流
            L"Stm",             // 流名称
            STGM_CREATE | STGM_WRITE | STGM_SHARE_EXCLUSIVE,
            0,0,
            &pStm);             // 取得流接口指针
    ASSERT( SUCCEEDED(hr) );

    hr = pStm->Write(           // 向流中写入数据
            "Hello",            // 数据地址
            5,                  // 字节长度(注意，没有写入字符串结尾的\0)
            NULL);              // 不需要得到实际写入的字节长度
    ASSERT( SUCCEEDED(hr) );

    if( pStm )          pStm->Release();// 释放流指针
    if( pSub )          pSub->Release();// 释放子存储指针
    if( pStg )          pStg->Release();// 释放根存储指针

    ::CoUninitialize()          // COM 释放
                                // 如果使用 AfxOleInit(),则不调用该函数
}
```



图二、运行示例程序一后，使用 DFView.exe 打开观察复合文件的效果图

示例二：打开一个复合文件，枚举其根存储下的所有对象。

```
#include <atlconv.h>        // ANSI、MBCS、UNICODE 转换

void SampleEnum()
{       // 假设你已经做过 COM 初始化了

        LPCTSTR lpFileName = _T( "c:\\a.stg" );
        HRESULT hr;
        IStorage *pStg = NULL;

        USES_CONVERSION;                                    // （注 6）
        LPCOLESTR lpwFileName = T2COLE( lpFileName );// 转换 T 类型为宽字符
        hr = ::StgIsStorageFile( lpwFileName );       // 是复合文件吗?
        if( FAILED(hr) )  return;

        hr = ::StgOpenStorage(                              // 打开复合文件
                lpwFileName,                  // 文件名称
                NULL,
                STGM_READ | STGM_SHARE_DENY_WRITE,
                0,
                0,
                &pStg);                       // 得到根存储接口指针

        IEnumSTATSTG *pEnum=NULL;  // 枚举器
        hr = pStg->EnumElements( 0, NULL, 0, &pEnum );
        ASSERT( SUCCEEDED(hr) );
```

```
        STATSTG statstg;
        while( NOERROR == pEnum->Next( 1, &statstg, NULL) )
        {
                // statstg.type 保存着对象类型 STGTY_STREAM 或 STGTY_STORAGE
                // statstg.pwcsName 保存着对象名称
                // ..... 还有时间，长度等很多信息。请查看 MSDN

                ::CoTaskMemFree( statstg.pwcsName );// 释放名称所使用的内存（注
6）
        }

        if( pEnum )         pEnum->Release();
        if( pStg )          pStg->Release();
}
```

六、小结

复合文件，结构化存储，是微软组件思想的起源，在此基础上继续发展出了持续性、命名、ActiveX、对象嵌入、现场激活.....一系列的新技术、新概念。因此理解和掌握 复合文件是非常重要的，即使在你的程序中并没有全面使用组件技术，复合文件技术也是可以单独被应用的。祝大家学习快乐，为社会主义软件事业而奋斗:-)

留作业啦.....

作业 1：写个小应用程序，从 MSWORD 的 doc 文件中，提取出附加信息（作者、公司.....）。

作业 2：写个全功能的“复合文件浏览编辑器”。

注 1：踅摸(xuemo)，动词，北方方言，寻找搜索的意思。

注 2：问：为什么不上网查资料学习？

答：开什么国际玩笑！在那遥远的 1995 年代，我的 500 块工资，不吃不喝正好够上 100 小时的 Internet 网。

注 3：OLE，对象的连接与嵌入。

注 4：可以用 DFView.exe 打开 MSWORD 的 DOC 文件进行复合文件的浏览。但是该程序并没有实现国际化，不能打开中文文件名的复合文件，因此需要改名后才能浏览。

注 5：CLSID，在后续的文章中介绍。

注 6：关于 COM 中内存使用的问题，在后续的文章中介绍。

COM 组件设计与应用（二）

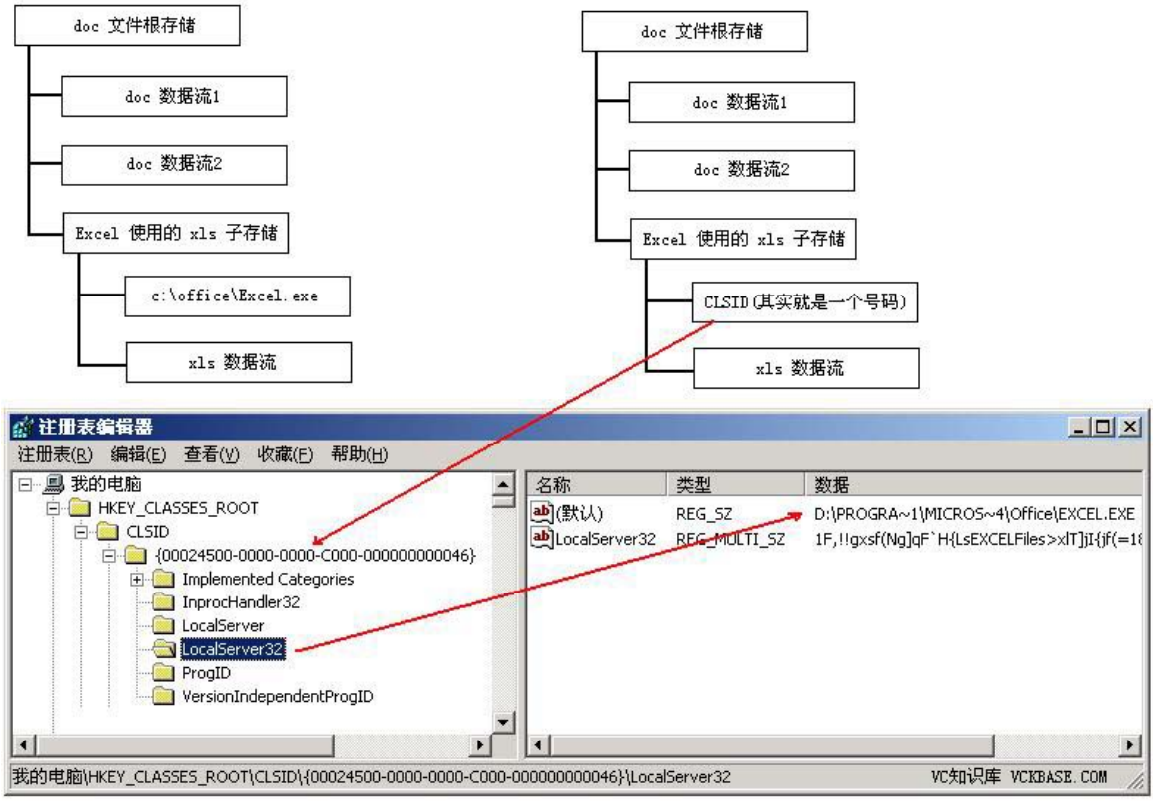
GUID 和 接口

一、前言

书接上回，话说在 doc(Word) 复合文件中，已经解决了保存 xls(Excel) 数据的问题了。那么，接下来又要解决另一个问题：当 WORD 程序读取复合文件，遇到了 xls 数据的时候，它该如何启动 Excel 呢？启动后，又如何让 Excel 自己去读入、解析、显示 xls 数据呢？

二、CLSID 概念

有一个非常简单的解决方案，那就是在对象数据的前面，保存有处理这个数据的程序名。（见下图左上）



图一、CLSID 的概念

这的确是一个简单的方法，但同时问题也很严重。在“张三”的计算机上，Excel 的路径是："c:\office\Excel.exe"，如果把这个 doc 文件复制到“李四”的计算机上使用，而“李四”的 Excel 的路径是：

"d:\Program files\Microsoft Office\Office\Excel.exe"，完蛋了:-(

于是，微软想出了一个解决方案，那就是不使用直接的路径表示方法，而使用一个叫 CLSID（注 1）的方式间接描述这些对象数据的处理程序路径。CLSID 其实就是一个号码，或者说是一个 16 字节的数。观察注册表（上图），在 HKCR\CLSID\{.....}主键下，

LocalServer32（DLL 组件使用 InprocServer32） 中保存着程序路径名称。CLSID 的结构定义如下：

```
typedef struct _GUID {
        DWORD Data1;       // 随机数
        WORD Data2;        // 和时间相关
        WORD Data3;        // 和时间相关
        BYTE Data4[8];     // 和网卡 MAC 相关
} GUID;

typedef GUID CLSID;  // 组件 ID
typedef GUID IID;    // 接口 ID
#define REFCLSID const CLSID &

// 常见的声明和赋值方法
CLSID CLSID_Excel =
{0x00024500,0x0000,0x0000,{0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46}};
struct __declspec(uuid("00024500-0000-0000-C000-000000000046")) CLSID_Excel;
class DECLSPEC_UUID("00024500-0000-0000-C000-000000000046") CLSID_Excel;
// 注册表中的表示方法
{00024500-0000-0000-C000-000000000046}
```

用一个号码间接表示程序名，的确是个 Good idea，实现了组件位置的透明性，并方便地扩展出 DCOM（远程组件）。但，但，但，但.....CLSID 有 16 个字节共 128 位二进制数，干吗用这么长的数字呀？遥想当年.....我还在上幼儿园的时候，人们设计了 socket，用 TCP/IP 协议进行网络通讯。每个参与通讯的计算机都有一个 4 字节的 IP 表示编号地址，范围是 0,0,0,0 ~ 255,255,255,255 共 42 亿个地址。可是没想到啊，没想到，自从 Internet 选择了 TCP/IP 协议后，42 亿个地址就不够全世界的劳动人民分配啦。除了劳动人民，还有冰箱、彩电、电饭锅、手机、手提电脑.....这些都需要连网呀。在办公室通过网络开启电饭锅给我焖饭，下班回家后就能吃现成的啦，多幸福呀？！（注：在我们家老婆是领导，所以是我做饭。咳.....）

由于前车之鉴，微软这次设计 CLSID/IID 就使用了 GUID 概念的 16 个字节，这下好啦，全世界 60 亿人口，每个人每秒钟分配 10 亿个号码，那么需要分配 1800 亿年。反正等到地球没有了都不会使用完的:-)

三、产生 CLSID

1. 如果使用开发环境编写组件程序，则 IDE 会自动帮你产生 CLSID；
2. 你可以手工写 CLSID，但千万不要和人家已经生成的 CLSID 重复呀，所以严重地

不推荐；（可是微软的 CLSID 都是手工写的，这叫“只许州官放火，不许百姓点灯”）；

3. 程序中，可以用函数 CoCreateGuid() 产生 CLSID；
4. 使用工具产生 GUID（注 2）；

vc6.0 版本运行："vc 目录\Common\Tools\GuidGen.exe"程序（你可以参照上回文章中介绍的方法，把这个工具程序加到开发环境中，方便调用）。vc.net 版本，在菜单“工具\创建 GUID”中，就可以执行了。

四、ProgID 概念

每一个 COM 组件都需要指定一个 CLSID，并且不能重名。它之所以使用 16 个字节，就是要从概率上保证重复是“不可能”的。但是，（世界上就怕“但是”二字）微软为了使用方便，也支持另一个字符串名称方式，叫 ProgID(注 3）。见上图注册表的 ProgID 子键内容（注 4）。由于 CLSID 和 ProgID 其实是一个概念的两个不同的表示形式，所以我们在程序中可以随便使用任何一种。（有些人就是讨厌，说话不算数。明明 GUID 的目的就是禁止重复，但居然又允许使用 ProgID？！ProgID 是一个字符串的名字，重复的可能性就太大了呀。赶明儿我也写个程序，我打算这个程序的 ProgID 叫“Excel.Application”，嘿嘿）下面介绍一下 CLSID 和 ProgID 之间的转换方法和相关的函数：

函数	功能说明
CLSIDFromProgID() 、CLSIDFromProgIDEx()	由 ProgID 得到 CLSID。没什么好说的，你自己都可以写，查注册表贝
ProgIDFromCLSID()	由 CLSID 得到 ProgID，调用者使用完成后要释放 ProgID 的内存(注 5)
CoCreateGuid()	随机生成一个 GUID
IsEqualGUID() 、 IsEqualCLSID() 、IsEqualIID()	比较 2 个 ID 是否相等
StringFromCLSID() 、StringFromGUID2() 、StringFromIID()	由 CLSID,IID 得到注册表中 CLSID 样式的字符串，注意释放内存

五、接口（Interface）的来历

到此，我们已经知道了 CLSID 或 ProgID 唯一地表示一个组件服务程序，那么根据这些 ID，就可以加载运行组件，并为客户端程序提供服务了。（启动组件程序的方法，会陆续介绍）。接下来先讨论如何调用组件提供的函数？-----接口。

作为客户端程序员，它希望或者说他要求：我的程序只写一次，然后不做任何修改就可以调用任意一个组件。举例来说：

1. 你可以在 Word 中嵌入 Excel，也可以嵌入 Picture，也可以嵌入任何第三方发表的 ActiveX 文档.....也就是说，连 Word 自己都不知道使用它的人将会在 doc 里面插入什么东东；

2. 你可以在 HTML 文件中插入一个 ActiveX，也可以插入一个程序脚本 Script，..... 你自己写的插件也可以插入到 IE 环境中。为了完成你的功能， 你绝对也不会去让微软修改 IE 吧？！

这个要求实在有点难度，Office 开发停滞了。说来话巧，一天老 O(Office 项目的总工程师)和小 B(VB 项目的总工程师)一起喝酒，老 O 向小 B 倾诉了他的烦恼：

老 O：怎么能让我写的程序 C，可以调用其它人写的程序 S 中的函数？(C 表示客户程序，S 表示提供服务的程序)

小 B：你是不是喝糊涂了？让 S 作成 DLL，你去 LoadLibrary()、GetProcAddress()、...FreeLibrary()？！

老 O：废话！要是这么简单就好了。问题是，连我都不知道这个 S 程序是干什么的？能干什么？我怎么调用呀？

小 B：哦.....这个比较高级，但我现在不能告诉你，因为我怕你印象不深。

老 O：~！ ·#￥%……—*.....

小 B：是这样的，在 VB 中，我们制定了一个标准，这个标准允许任何一个 VB 开发者，把他自己写的某个功能的小程序放在 VB 的工具栏上，这样就好象他扩展了 VB 的功能一样。

老 O：哦？就是那个叫什么 VBX 的滥玩意儿？

小 B：我呸.....别看 VBX 这个东西不起眼儿，的确我也没看上它。但你猜怎么着？现在有成千上万的 VB 程序爱好者把他们写的各式各样功能的 VBX 小程序，放到网上，让大家共享那。

老 O：哦~~~，那你们的这个 VBX 标准是什么？

小 B：嘿嘿.....其实特简单，就是在 VBX 中必须实现 7 个函数，这 7 个函数名称和功能必须是：初始化、释放、显示、消息处理.....，而至于它内部想干什么，我也管不着。我只是在需要的时候调用我需要的这 7 个函数。

老 O：哦~~~，这样呀.....对了，我现有个急事，我先走了。88，你付帐吧.....

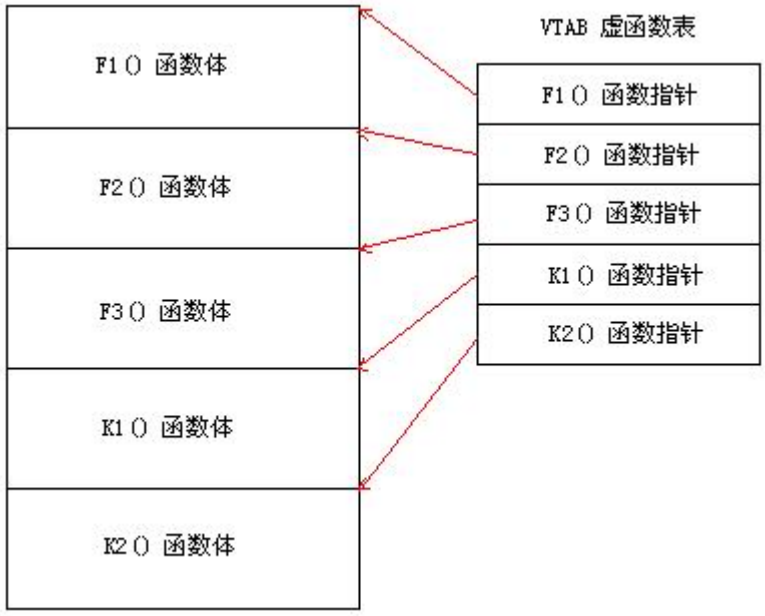
小 B：喂！喂喂..... 走这么急干什么，钱包都掉了:-)

老 O 虽然丢了钱包，仍然兴奋地冲回办公室，他开始了思考.....

1、我的程序 C，要能调用任何人写的程序 B。那么 B 必须要按照我事先的要求，提供我需要的函数 F1(),F2(),F3(),K1(),K2()。

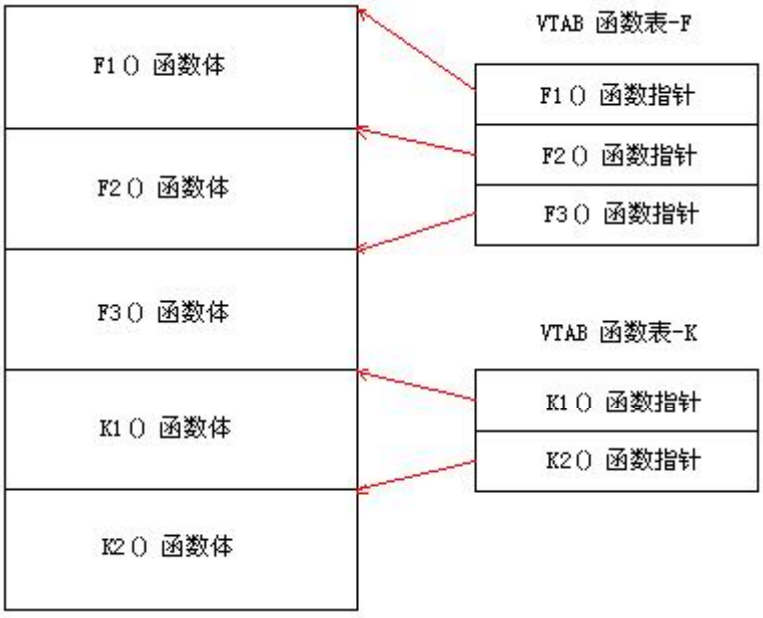
2、BASIC 是解释执行，因此它的函数不用考虑书写顺序，只要给出函数名，解释器就能找到。但我使用的是 C++呀.....

3、C++编译后的代码中没有函数名，只有函数地址，因此我必须改进为用 VTAB（虚函数表）表示函数入口：



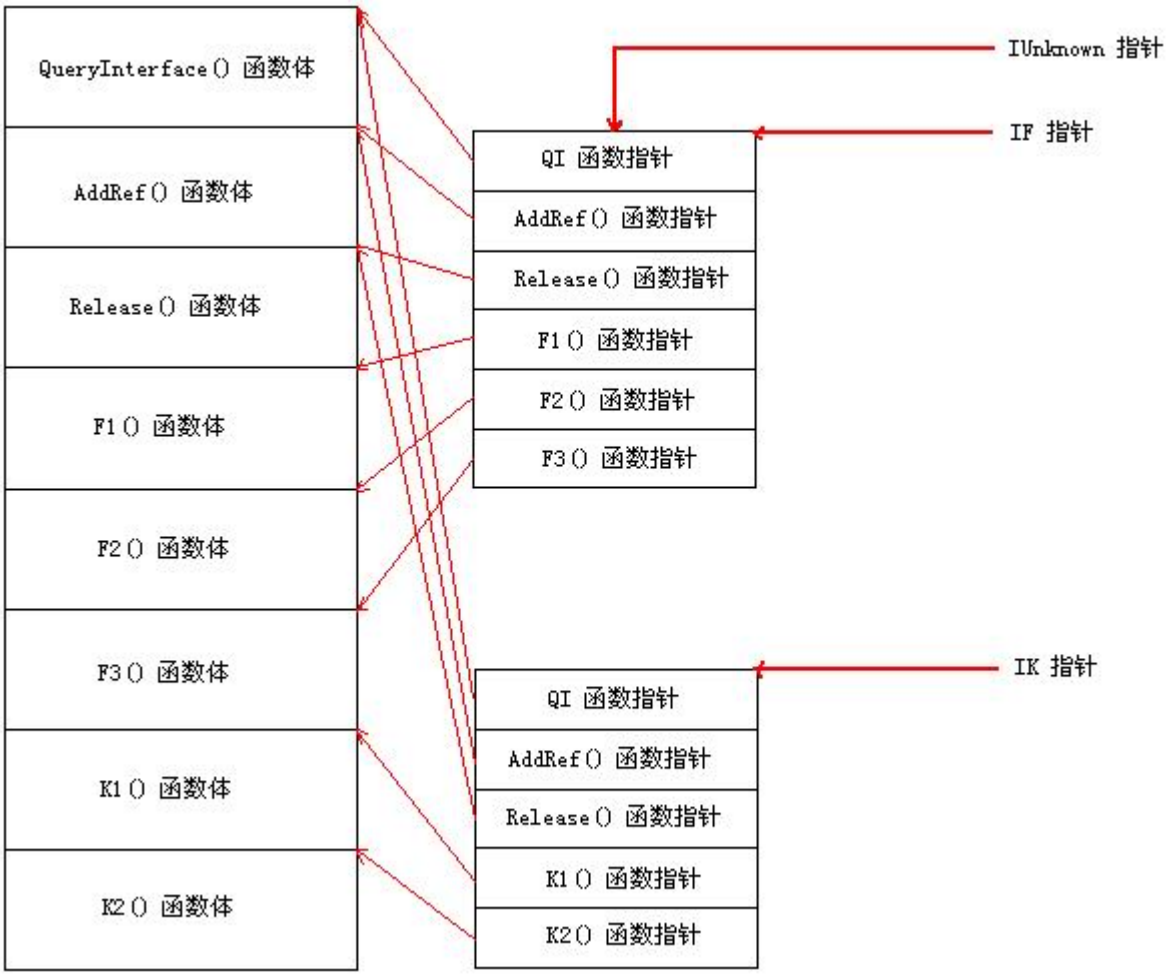
图二、VTAB 的结构

4、还不够好，需要改进一下，因为所有的函数地址都放在一个表中会不灵活、不好修改、不易扩展。恩，有了！按照函数功能的类型进行分类：



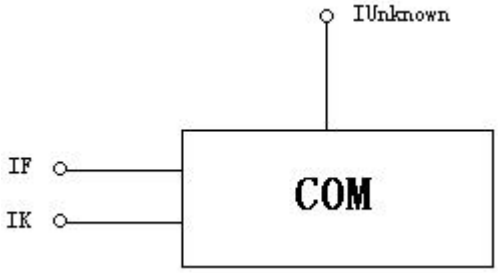
图三、多个 VTAB 的结构

5、问题又来了，现在有 2 个 VTAB 虚函数表，那么怎么能够从一个表找到另一个表那？恩又有办法了，我要求你必须要实现一个函数，并且这个函数地址必须放在所有表的开头（表中的第一个函数指针），这个函数就叫 QueryInterface()吧，完成从一个表查找到另一个表的功能:（除了 QueryInterface()函数,顺便也完成另外两个函数，叫 AddRef() 和 Release()。这两个函数的功能以后再说）



图四、COM 接口结构

6、为了以后描述方便，不再使用上图（图四）的方法了，而使用图五这样简洁的样式：



图五、COM 接口结构的简洁图示

六、接口（Interface）概念

1、函数是通过 VTAB 虚函数表提供其地址， 从另一个角度来看，不管用什么语言开发，编译器产生的代码都能生成这个表。这样就实现了组件的“二进制特性”轻松实现了组件的跨语言要求。

2、假设有一个指针型变量保存着 VTAB 的首地址，则这个变量就叫“接口指针”(注

6)， 变量命名的时候，习惯上加上"I"开头。另外为了区分不同的接口，每个接口 也都要有一个名字，该名字就和 CLSID 一样，使用 GUID 方式，叫 IID。

3、接口一经发表，就不能再修改了。不然就会出现向前兼容的问题。这个性质叫“接口不变性”。

4、组件中必须有 3 个函数，QueryInterface、AddRef、Release，它们 3 个函数也组成一个接口，叫"IUnknown"。(注 7)

5、任何接口，其实都包含了 IUnknown 接口。随着你接触到更多的接口就会了更体会解到接口的另一个性质“继承性”。

6、在任何接口上，调用表中的第一个函数,其实就是调用 QueryInterface()函数，就得到你想要的另外一个接口指针。这个性质叫“接口的传递性”

7、C/C++语言中需要事先对函数声明，那么就 会要求组件也必须提供 C 语言的头文件。不行！为了能使 COM 具有跨语言的能力，决定不再为任何语言提供对应的函数接口声明，而是独立地提供一个叫类型库（TLB）的声明。每个语言的 IDE 环境自己去根据 TLB 生成自己语言需要的包装。这个性质叫“接口声明的独立性”（注 8）

七、客户程序与组件之间的协商调用

回到我们的上一个话题，Word 中嵌入一个组件，那么 Word 是如何协商使用这个组件的那？下面是容器和组件之间的一个模拟对话过程：

	容器 协商部分	组件 应答部分
1	根据 CLSID 启动组件 。 CoCreateInstance()	生成对象，执行构造函数，执行初始化动作。
2	你有 IUnknown 接口吗？	有，给你！
3	恩，太好了，那么你有 IPersistStorage 接口吗？(注 9) IUnknown::QueryInterface(IID_IPersistStorage...)	没有！
4	真差劲，连这个都没有。那你有 IPersistStreamInit 接口吗？(注 10) IUnknown::QueryInterface(IID_IPersistStreamInit...)	哈，这个有，给！
5	好，好，这还差不多。你现在给我初始化吧。 IPersistStreamInit::InitNew()	OK，初始化完成了。
6	完成了？好！现在你读数据去吧。 IPersistStreamInit::Load()	读完啦。我根据数据，已经在窗口中显示出来了。
7	好，现在咱们各自处理用户的鼠标、键盘消息吧.....	.....
8	哎呀！用户要保存退出程序了。你的数据被用户修改了吗？ IPersistStreamInit::IsDirty()	改了，用户已经修改啦。

9	那好，那么用户修改后，你的数据需要多大的存储空间呀？ IPersistStreamInit::GetSizeMax()	恩，我算算呀……好了，总共需要 500KB。
10	晕，你这么个小玩意居然占用这么大空间？！……好了，你可以存了。 IPersistStreamInit::Save()	谢谢，我已经存好了。
11	恩。拜拜了您那。(注 11) IPersistStreamInit::Release()；IUnknown::Release()	执行析构函数，删除对象。
12	我自己也该退出了…… PostQuitMessage()	

容器（或者说客户端）就是这样和组件进行对话，协商调用的。如果组件甲实现了 IA 接口，那么容器就会使用它，如果组件乙没有提供 IA 接口，但是它提供了 IB 接口，那么容器就会调用 IB 接口的函数……如此，容器程序根本就不需要知道组件到底是干什么的，组件到底是用什么语言开发的，组件的磁盘位置到底在哪里，它都可以正常运行。太奇妙了！太精彩了！怎一个“爽”字了得！

八、小结

第二回中，介绍了两个非常重要的概念：CLSID 和 Interface。由于全篇都是概念描述而没有示例程序相配合，可能读者的理解还不太深入、不彻底。别着急，我们马上就要进入到组件程序设计阶段了，到那个时候，你根据具体的程序代码，再回过头来再次阅读本回文章，没读懂？哦……再读！慢慢地您老人家就懂了:-)

留作业啦……

1、IDispatch 接口的 IID 是多少？（哎~~~ 笨笨，在源程序中，用鼠标右键执行 Go to definition 呀）

2、IPicture 接口有几个函数？功能是什么？（别玩了！你多大了？想不想在程序中显示 JPG 图像呀，看 MSDN 去）

想知道为什么 COM 函数总是返回 HRESULT 吗？想知道如何使用 BSTR、VARIANT 吗？想知道 COM 中应该如何使用内存吗？想知道如何使用 UNICODE 吗？……恩~~~，我现在不能告诉你，我现在告诉你，怕你印象不深！且听下回分解……

注 1：CLSID = Class ID 上回书已经介绍了把 CLSID 写入复合文件的函数：WriteClassStg()、IStorage::SetClass()。

注 2：GUID 全局唯一标示符，CLSID/IID 其实是借用了 GUID 的概念。

注 3：ProgID = Program ID，等价于 CLSID, 是用字符串表示的。

注 4：注册表子键 ProgID 和 VersionIndependentProgID 分别表示真正的 ProgID 和

版本无关的 ProgID。比如在我计算机上安装的 Excel，它的 ProgID = "Excel.Application.9"，而 VersionIndependentProgID = "Excel.Application"。

注 5：COM 组件的内存管理，见后续的文章。

注 6：Interface = 接口，以前微软不叫它接口，而叫协议 Protocol。其实我 到认为这个词更贴切一些。

注 7：IUnknown 这个名字起的好，居然叫“我不知道”:-)，它的 IID 叫 IID_IUnknown，如果用注册表样式表示，那么它的值是{00000000-0000-0000-C000-000000000046}。

注 8：TLB 是由一个描述接口的文件 IDL 经过编译产生的。IDL 的说明，见后续的文章吧。

注 9：IPersistStorage 是用复合文件的存储(Storage)功能来保存/读取数据用的一个接口。

注 10：IPersistStreamInit 是用复合文件的流(Stream)功能来保存/读取数据用的一个接口。

注 11：拜拜了您那 = 英语北京话，再见。

COM 组件设计与应用（三）
数据类型

一、前言

上回书介绍了 GUID、CLSID、IID 和接口的概念。本回的重点是介绍 COM 中的数据类型。咋还不介绍组件程序的设计步骤呀？咳.....别着急，别着急！孔子曰：“饭要一口一口地吃”；老子语：“心急吃不了热豆腐”，孙子云：“走一步看一步吧” ..... 先掌握必要的知识，将来写起程序来才会得心应手也:-)

走入正题之前，请大家牢牢记住一条原则：**COM 组件是运行在分布式环境中的**。比如，你写了一个组件程序（DLL 或 EXE），那么使用者可能是在本机的某个进程内加载组件（INPROC_SERVER）；也可能是从另一个进程中调用组件的进程（LOCAL_SERVER）；也可能是在这台计算机上调用地球那边计算机上的组件（REMOTE_SERVER）。所以在理解和设计的时候，要时时刻刻想起这句话。快！拿出小本本，记下来！

二、HRESULT 函数返回值

每个人在做程序设计的时候，都有他们各自的哲学思想。拿函数返回值来说，就有好多种形式。

函数	返回	返回值信息

	值	
double sin(double)	浮点数值	计算正玄值
BOOL DeleteFile(LPCTSTR)	布尔值	文件删除是否成功。如失败，需要GetLastError()才能取得失败原因
void * malloc(size_t)	内存指针	内存申请，如果失败，返回空指针 NULL
LONG RegDeleteKey(HKEY,LPCTSTR)	整数	删除注册表项。0 表示成功，非 0 失败，同时这个值就反映了失败的原因
UINT DragQueryFile(HDROP,UINT,LPTSTR,UINT)	整数	取得拖放文件信息。以不同的参数调用，则返回不同的含义： 一会儿表示文件个数，一会儿表示文件名长度，一会儿表示字符长度
....	...	....

如此纷繁复杂的返回值，如此含义多变的返回值，使得大家在学习和使用的过程中，增加了额外的困难。好了，COM 的设计规范终于对他们进行了统一。组件 API 及接口指针中，除了 IUnknown::AddRef()和 IUnknown::Release()两个函数外，其它所有的函数，都以 HRESULT 作为返回值。大家想象一个组件的接口函数比如叫 Add()，完成 2 个整数的加法运算，在 C 语言中，我们可以如下定义：

```
long Add( long n1, long n2 )
{
    return n1 + n2;
}
```

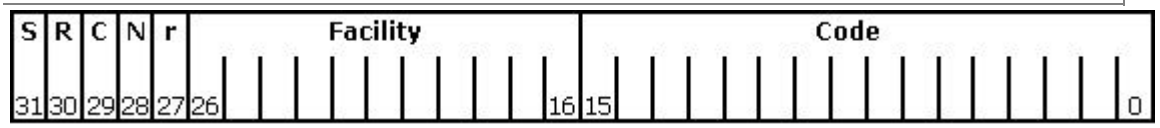
还记得刚才我们说的原则吗？**COM 组件是运行在分布式环境中的**。也就是说，这个函数可能运行在“地球另一边”的计算机上，既然运行在那么遥远的地方，就有可能出现服务器关机、网络掉线、运行超时、对方不在服务区.....等异常。于是，这个加法函数，除了需要返回运算结果以外，还应该返回一个值------函数是否被正常执行了。

```
HRESULT Add( long n1, long n2, long *pSum )
{
    *pSum = n1 + n2;
```

```
    return S_OK;
}
```

如果函数正常执行，则返回 S_OK，同时真正的函数运行结果则通过参数指针返回。如果遇到了异常情况，则 COM 系统经过判断，会返回相应的错误值。常见的返回值有：

HRESULT	值	含义
S_OK	0x00000000	成功
S_FALSE	0x00000001	函数成功执行完成，但返回时出现错误
E_INVALIDARG	0x80070057	参数有错误
E_OUTOFMEMORY	0x8007000E	内存申请错误
E_UNEXPECTED	0x8000FFFF	未知的异常
E_NOTIMPL	0x80004001	未实现功能
E_FAIL	0x80004005	没有详细说明的错误。一般需要取得 Rich Error 错误信息(注 1)
E_POINTER	0x80004003	无效的指针
E_HANDLE	0x80070006	无效的句柄
E_ABORT	0x80004004	终止操作
E_ACCESSDENIED	0x80070005	访问被拒绝
E_NOINTERFACE	0x80004002	不支持接口



图一、HRESULT 的结构

HRESULT 其实是一个双字节的值，其最高位(bit)如果是 0 表示成功，1 表示错误。具体参见 MSDN 之"Structure of COM Error Codes"说明。我们在程序中如果需要判断返回值，则可以使用比较运算符号；switch 开关语句；也可以使用 VC 提供的宏：

```
HRESULT hr = 调用组件函数;
if( SUCCEEDED( hr ) ){...} // 如果成功
......
if( FAILED( hr ) ){...} // 如果失败
......
```

三、UNICODE

计算机发明后，为了在计算机中表示字符，人们制定了一种编码，叫 ASCII 码。ASCII 码由一个字节中的 7 位(bit)表示，范围是 0x00 - 0x7F 共 128 个字符。他们以为这 128 个数字就足够表示 abcd....ABCD....1234 这些字符了。

咳.....说英语的人就是“笨”！后来他们突然发现，如果需要按照表格方式打印这些字符的时候，缺少了“制表符”。于是又扩展了 ASCII 的定义，使用一个字节的全部 8 位(bit)来表示字符了，这就叫扩展 ASCII 码。范围是 0x00 - 0xFF 共 256 个字符。

咳.....说中文的人就是聪明！中国人利用连续 2 个扩展 ASCII 码的扩展区域（0xA0 以后）来表示一个汉字，该方法的标准叫 GB-2312。后来，日文、韩文、阿拉伯文、台湾繁体（BIG-5）.....都使用类似的方法扩展了本地字符集的定义，现在统一称为 MBCS 字符集（多字节字符集）。这个方法是有缺陷的，因为各个国家地区定义的字符集有交集，因此使用 GB-2312 的软件，就不能在 BIG-5 的环境下运行（显示乱码），反之亦然。

咳.....说英语的人终于变“聪明”一些了。为了把全世界人民所有的所有的文字符号都统一进行编码，于是制定了 UNICODE 标准字符集。UNICODE 使用 2 个字节表示一个字符(unsigned shor int、WCHAR、_wchar_t、OLECHAR)。这下终于好啦，全世界任何一个地区的软件，可以不用修改地就能在另一个地区运行了。虽然我用 IE 浏览日本网站，显示出我不认识的日文文字，但至少不会是乱码了。UNICODE 的范围是 0x0000 - 0xFFFF 共 6 万多个字符，其中光汉字就占用了 4 万多个。嘿嘿，中国人赚大发了:0)

在程序中使用各种字符集的方法：

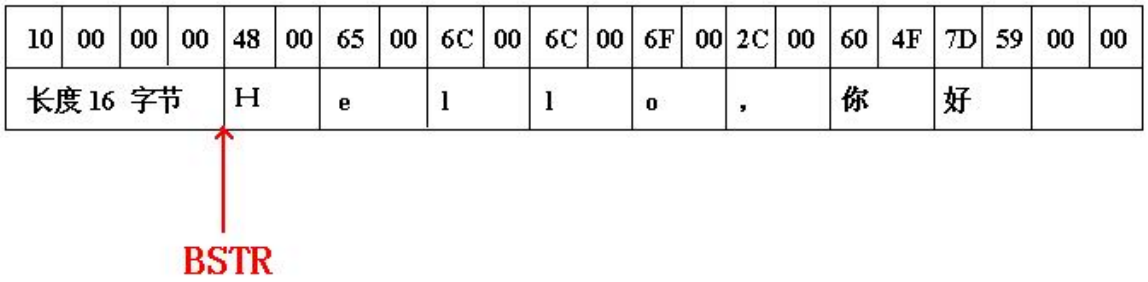
```
const char * p = "Hello"; // 使用 ASCII 字符集
const char * p = "你好"; // 使用 MBCS 字符集，由于 MBCS 完全兼容 ASCII，
                         // 多数情况下，我们并不严格区分他们
LPCSTR p = "Hello,你好"; // 意义同上

const WCHAR * p = L"Hello,你好"; // 使用 UNICODE 字符集
LPCOLESTR p = L"Hello,你好"; // 意义同上

// 如果预定义了_UNICODE，则表示使用 UNICODE 字符集；如果定义了_MBCS,则表示
// 使用 MBCS
const TCHAR * p = _T("Hello,你好");
LPCTSTR p = _T("Hello,你好"); // 意义同上
```

在上面的例子中，T 是非常有意思的一个符号（TCHAR、LPCTSTR、LPTSTR、_T()、_TEXT()...），它表示使用一种中间类型，既不明确表示使用 MBCS，也不明确表示使用 UNICODE。那到底使用哪种字符集那？嘿嘿.....编译的时候决定吧。设置条件编译的方式是：VC6 中，"Project\Settings...\C/C++卡片 Preprocessor definitions" 中添加或修改 _MBCS、_UNICODE；VC.NET 中，"项目\属性\配置属性\常规\字符集"然后用组合窗进行选择。使用 T 类型，是非常好的习惯，严重推荐！

四、BSTR

COM 中除了使用一些简单标准的数据类型外（注 2），字符串类型需要特别重点地说明一下。还记得原则吗？**COM 组件是运行在分布式环境中的**。通俗地说，你不能直接把一个内存指针直接作为参数传递给 COM 函数。你想想，系统需要把这块内存的内容传递到“地

球另一 边”的计算机上，因此，我至少需要知道你这块内存的尺寸吧？不然让我如何传递呀？传递多少字节呀？！而字符串又是非常常用的一种类型，因此 COM 设计者引入了 BASIC 中字符串类型的表示方式---BSTR。BSTR 其实是一个指针类型，它的内存结构是：（输入程序片段 BSTR p = ::SysAllocString(L"Hello,你好");断点执行，然后观察 p 的内存）



图二、BSTR 内存结构

BSTR 是一个指向 UNICODE 字符串的指针，且 BSTR 向前的 4 个字节中，使用 DWORD 保存着这个字符串的字节长度（ 没有含字符串的结束符）。因此系统就能够正确处理并传送这个字符串到“地球另一 边”了。特别需要注意的是，由于 BSTR 的指针就是指向 UNICODE 串，因此 BSTR 和 LPOLESTR 可以在一定程度上混用，但一定要注意：

有函数 fun(LPCOLESTR lp)，则你调用 BSTR p=...; fun(p); 正确

有函数 fun(const BSTR bstr)，则你调用 LPCOLESTR p=...; fun(p); 错误!!!

有关 BSTR 的处理函数：

API 函数	**说明**
SysAllocString()	申请一个 BSTR 指针，并初始化为一个字符串
SysFreeString()	释放 BSTR 内存
SysAllocStringLen()	申请一个指定字符长度的 BSTR 指针，并初始化为一个字符串
SysAllocStringByteLen()	申请一个指定字节长度的 BSTR 指针，并初始化为一个字符串
SysReAllocStringLen()	重新申请 BSTR 指针
CString 函数	**说明**
AllocSysString()	从 CString 得到 BSTR
SetSysString()	重新申请 BSTR 指针，并复制到 CString 中
CComBSTR 函数 ATL 的 BSTR 包装类。在 atlbase.h 中定义	
Append()、AppendBSTR()、AppendBytes()、ArrayToBSTR() 、 BSTRToArray() 、	太多了，但从函数名称不能看出其基本功能。详细资料，查看 MSDN 吧。另外，左侧函

AssignBSTR()、Attach()、Detach()、Copy()、CopyTo()、Empty()、Length()、ByteLength()、ReadFromStream() 、 WriteToStream() 、LoadString()、ToLower()、ToUpper() 运算符重载：!,!=,==,<,>,&,+=,+,=,BSTR	数，有很多是 ATL 7.0 提供的，VC6.0 下所带的 ATL 3.0 不支持。 由于我们将来主要用 ATL 开发组件程序，因此使用 ATL 的 CComBSTR 为主。VC 也提供了其它的包装类 _bstr_t。

五、各种字符串类型之间的转换

1、函数 WideCharToMultiByte()，转换 UNICODE 到 MBCS。使用范例：

```
LPCOLESTR lpw = L"Hello,你好";
size_t wLen = wcslen( lpw ) + 1; // 宽字符字符长度，+1 表示包含字符串结束符

int aLen=WideCharToMultiByte(  // 第一次调用，计算所需 MBCS 字符串字节长度
            CP_ACP,
            0,
            lpw,  // 宽字符串指针
            wLen, // 字符长度
            NULL,
            0,  // 参数 0 表示计算转换后的字符空间
            NULL,
            NULL);

LPSTR lpa = new char [aLen];

WideCharToMultiByte(
            CP_ACP,
            0,
            lpw,
            wLen,
            lpa,  // 转换后的字符串指针
            aLen, // 给出空间大小
            NULL,
            NULL);

// 此时，lpa 中保存着转换后的 MBCS 字符串
... ... ... ...
delete [] lpa;
```

2、函数 MultiByteToWideChar()，转换 MBCS 到 UNICODE。使用范例：

```
LPCSTR lpa = "Hello,你好";
size_t aLen = strlen( lpa ) + 1;

int wLen = MultiByteToWideChar(
            CP_ACP,
            0,
            lpa,
            aLen,
            NULL,
            0);

LPOLESTR lpw = new WCHAR [wLen];
MultiByteToWideChar(
            CP_ACP,
            0,
            lpa,
            aLen,
            lpw,
            wLen);
... ... ... ...
delete [] lpw;
```

3、使用 ATL 提供的转换宏。

A2BSTR	OLE2A	T2A	W2A
A2COLE	OLE2BSTR	T2BSTR	W2BSTR
A2CT	OLE2CA	T2CA	W2CA
A2CW	OLE2CT	T2COLE	W2COLE
A2OLE	OLE2CW	T2CW	W2CT
A2T	OLE2T	T2OLE	W2OLE
A2W	OLE2W	T2W	W2T

上表中的宏函数，其实非常容易记忆：

2	好搞笑的缩写，to 的发音和 2 一样，所以借用来表示“转换为、转换到”的含义。
A	ANSI 字符串，也就是 MBCS。
W、OLE	宽字符串，也就是 UNICODE。

T	中间类型 T。如果定义了 _UNICODE，则 T 表示 W；如果定义了 _MBCS，则 T 表示 A
C	const 的缩写

使用范例：

```
#include <atlconv.h>

void fun()
{
    USES_CONVERSION;  // 只需要调用一次，就可以在函数中进行多次转换

    LPCTSTR lp = OLE2CT( L"Hello,你好") );
    ... ... ... ...
    // 不用显式释放 lp 的内存，因为
    // 由于 ATL 转换宏使用栈作为临时空间，函数结束后会自动释放栈空间。
}
```

使用 ATL 转换宏，由于不用释放临时空间，所以使用起来非常方便。但是考虑到栈空间的尺寸（VC 默认 2M），使用时要注意几点：

1、只适合于进行短字符串的转换；

2、不要试图在一个次数比较多的循环体内进行转换；

3、不要试图对字符型文件内容进行转换，因为文件尺寸一般情况下是比较大的；

4、对情况 2 和 3，要使用 MultiByteToWideChar() 和 WideCharToMultiByte()；

六、VARIANT

C++、BASIC、Java、Pascal、Script.....计算机语言多种多样，而它们各自又都有自己的数据类型，COM 产生目的，其中之一就是要跨语言(注 3)。而 VARIANT 数据类型就具有跨语言的特性，同时它可以表示（存储）任意类型的数据。从 C 语言的角度来讲，VARIANT 其实是一个结构，结构中用一个域(vt)表示------该变量到底表示的是什么类型数据，同时真正的数据则存贮在 union 空间中。结构的定义太长了（虽然长，但其实很简单）大家去看 MSDN 的描述吧，这里给出如何使用的简单示例：

学生：我想用 VARIANT 表示一个 4 字节长的整数，如何做？

老师：VARIANT v; v.vt=VT_I4; v.lVal=100;

学生：我想用 VARIANT 表示布尔值“真”，如何做？

老师：VARIANT v; v.vt=VT_BOOL; v.boolVal=VARIANT_TRUE;

学生：这么麻烦？我能不能 v.boolVal=true; 这样写？

老师：不可以！因为

类型	字节长度	假值	真值
bool	1(char)	0(false)	1(true)
BOOL	4(int)	0(FALSE)	1(TRUE)
VT_BOOL	2(short int)	0(VARIANT_FALSE)	-1(VARIANT_TRUE)

所以如果你 v.boolVal=true 这样赋值，那么将来 if(VARIANT_TRUE==v.boolVal) 的时候会出问题(-1 != 1)。但是你注意观察，任何布尔类型的“假”都是 0，因此作为一个好习惯，在做布尔判断的时候，不要和“真值”相比较，而要与“假值”做比较。

学生：谢谢老师，你太牛了。我对老师的敬仰如滔滔江水，连绵不绝.....

学生：我想用 VARIANT 保存字符串，如何做？

老师：VARIANT v; v.vt=VT_BSTR; v.bstrVal=SysAllocString(L"Hello,你好");

学生：哦.....我明白了。可是这么操作真够麻烦的，有没有简单一些的方法？

老师：有呀，你可以使用现成的包装类 CComVariant、COleVariant、_variant_t。比如上面三个问题就可以这样书写：CComVariant v1(100),v2(true),v3("Hello,你好"); 简单了吧？！(注 4)

学生：老师，我再问最后一个问题，我如何用 VARIANT 保存一个数组？

老师：这个问题很复杂，我现在不能告诉你，我现在告诉你怕你印象不深.....(注 5)

学生：~!@#$%^&*().....晕！

七、小结

以上所介绍的内容，是基本功，必须熟练掌握。先到这里吧，休息一会儿.....更多精彩内容，敬请关注《COM 组件设计与应用(四)》

注 1：在后续的 ISupportErrorInfo 接口中介绍。

注 2：常见的数据类型，请参考 IDL 文件的说明。（别着急，还没写那.....嘿嘿）

注 3：跨语言就是各种语言中都能使用 COM 组件。但啥时候能跨平台呢？

注 4：CComVariant/COlevariant/_variant_t 请参看 MSDN。

注 5：关于安全数组 SafeArray 的使用，在后续的文章中讨论。

COM 组件设计与应用（四）

简单调用组件

一、前言

同志们、朋友们、各位领导，大家好。

VCKBASE 不得了，
网友众多文章好。
组件设计怎么学？
知识库里闷头找！

摘自---杨老师打油集录

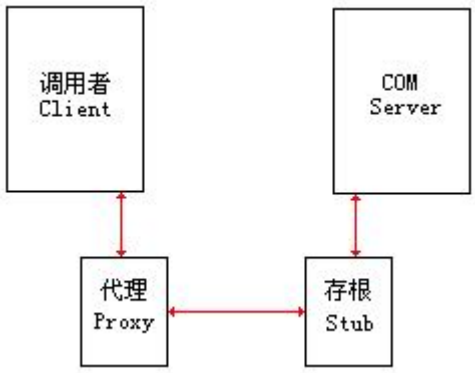
在 VCKBASE 的顶力支持下，在各位网友回帖的鼓励下，我才能顺利完成系列论文的前三回。书到本回，我们终于开始写代码啦。写点啥那？恩，有了！咱们先从如何调用现成的简单的组件开始吧，同时也顺便介绍一些相关的知识。

二、组件的启动和释放

在第三回中，大家用“小本本”记录了一个原则：**COM 组件是运行在分布式环境中的 。**于是，如何启动组件立刻就遇到了严重的问题，大家看这段代码：

```
p = new 对象;
p->对象函数();
delete p;
```

这样的代码再熟悉不过了，在本地进程中运行是不会有问题的。但是你想想，如果这个对象是在“地球另一边”的计算机上，结果会如何？嘿嘿，C++ 在设计 new 的时候，可没有考虑远程的实现呀（计算机语言当然不会，也没必要去设计）。因此启动组件、调用接口的功能，当然就由 COM 系统来实现了。



图一　组件调用机制

由上图可以看出，当调用组件的时候，其实是依靠代理（运行在本地）和存根（运行在远端）之间的通讯完成的。具体来说，当客户程序通过 CoCreateInstance() 函数启动组件，则代理接管该调用，它和存根通讯，存根则它所在的本地（相对于客户程序来说就是远程了）

执行 new 操作加载对象。对于初学者，你可以不用理它，代理和存根对我们来说是透明的。只要大约知道是怎么一回事就一切 OK 了。

问题又来了，这个远程的对象什么时候消灭呢？在第二回介绍接口概念的时候，当时我们特意忽略了两个函数，就是 IUnknown::AddRef()和 IUnknown::Release()，从函数名就能猜到了，一个是对内部引用记数器(Ref)加 1，一个是释放(减 1)，当记数器减为 0 的时候，就是释放的机会啦。看起来很复杂，没办法，因为这是在介绍原理。其实在我们写程序的时候到比较简单，请大家遵守几个原则：

1、启动组件得到一个接口指针(Interface)后，不要调用 AddRef()。因为系统知道你得到了一个指针，所以它已经帮你调用了 AddRef()函数；

2、通过 QueryInterface()得到另一个接口指针后，不要调用 AddRef()。因为.....和上面的道理一样；

3、当你把接口指针赋值给（保存到）另一个变量中的时候，请调用 AddRef()；

4、当不需要再使用接口指针的时候，务必执行 Release()释放；

5、当使用智能指针的时候，可以省略指针的维护工作；（注 1）

三、内存分配和释放

自从学习了 C 语言，老师就教导我们说：对于动态内存的申请和释放，一定要遵守“谁申请，谁释放”的原则。在此原则的指导下，不仅是我、不仅是你，就连特级大师都设计了这样怪怪的函数：

函数	说明	评论
GetWindowText(HWND,LPTSTR,int)	取得窗口标题。需要在参数中给出保存标题所使用的内存指针，和这块内存的尺寸。	晕！我又不知道窗口标题的长度，居然还要我提供尺寸？！没办法，只能估摸着给一个大一些的尺寸吧。
sprintf(char *,const char *,...)	格式化一个字符串。这个函数不用给出缓冲区的长度啦。	恩，虽然不用给出长度了，但你敢给个小尺寸吗？哼！
int CListBox::GetTextLen(int) CListBox::GetText(int,LPTSTR)	取得列表窗中子项目的标题。需要调用两个函数，先取得长度，然后分配内存，再实际取得标题内容。	真烦！

说实在的，不但函数调用者感觉别扭，就连函数设计者心情也不会爽的，而这一切都是为了满足所谓“谁申请，谁释放”的原则。 解决这个问题最好的方式就是：函数内部根据实际需要动态申请内存，而调用者负责释放。这虽然违背了上述原则，但 COM 从方便性

和效率出发，确实是这么设计的。

	C 语言	C++语言	Windows 平台	COM	IMalloc 接口	BSTR
申请	malloc()	new	GlobalAlloc()	CoTaskMemAlloc()	Alloc()	SysAllocString()
重新申请	realloc()		GlobalReAlloc()	CoTaskRealloc()	Realloc()	SysReAllocString()
释放	free()	delete	GlobalFree()	CoTaskMemFree()	Free()	SysFreeString()

以上这些函数必须要按类型配合使用（比如：new 申请的内存，则必须用 delete 释放）。在 COM 内部，当然你可以随便使用任何类型的内存分配释放函数，但组件如果需要与客户进行内存的交互，则必须使用上表中的后三类函数族。

1、BSTR 内存在上回书中，已经有比较丰富的介绍了，不再重复；

2、CoTaskXXX()函数族，其本质上就是调用 C 语言的函数（malloc...）；

3、IMalloc 接口又是对 CoTaskXXX() 函数族的一个包装。包装后，同时增强了一些功能，比如：IMalloc::GetSize()可以取得尺寸，使用 IMallocSpy 可以监视内存的使用；

四、参数传递方向

在 C 语言的函数声明中，尤其当参数为指针的时候，你是看不出它传递方向的。比如：void fun(char * p1, int * p2); 请问，p1、p2 哪个是入参？哪个是出参？甚或都是入参或都是出参？由于牵扯到内存分配和释放等问题，COM 需要明确标注参数方向。以后我们写程序，就类似下面的样子：

```
HRESULT Add([in] long n1, [in] long n2, [out] long *pnSum);  // IDL 文件(注 2)
STDMETHOD(Add)(/*[in]*/ long n1, /*[in]*/ long n2, /*[out]*/ long *pnSum);  // .h 文件
```

如果参数是动态分配的内存指针，那么遵守如下的规定：

方向	申请人	释放人	提示
[in]	调用者	调用者	组件接收指针后，不能重新分配内存
[out]	组件	调用者	组件返回指针后，调用者“爱咋咋地”(注 3)
[in,out]	调用者	调用者	组件可以重新分配内存

五、示例程序

示例一、由 CLSID 得到 ProgID。（程序以 word 为例子。如果运行不正确，嘿嘿，你没有安装 word 吧？）

```
::CoInitialize( NULL );

HRESULT hr;
// {000209FF-0000-0000-C000-000000000046} = word.application.9
CLSID clsid = {0x209ff,0,0,{0xc0,0,0,0,0,0,0,0x46}};
LPOLESTR lpwProgID = NULL;

hr = ::ProgIDFromCLSID( clsid, &lpwProgID );
if ( SUCCEEDED(hr) )
{
        ::MessageBoxW( NULL, lpwProgID, L"ProgID", MB_OK );

        IMalloc * pMalloc = NULL;
        hr = ::CoGetMalloc( 1, &pMalloc );  // 取得 IMalloc
        if ( SUCCEEDED(hr) )
        {
                pMalloc->Free( lpwProgID );  // 释放 ProgID 内存
                pMalloc->Release();          // 释放 IMalloc
        }
}

::CoUninitialize();
```

示例二、如何使用“浏览文件夹”选择对话窗。

```
CString BrowseFolder(HWND hWnd, LPCTSTR lpTitle)
{
    // 调用 SHBrowseForFolder 取得目录(文件夹)名称
    // 参数 hWnd: 父窗口句柄
    // 参数 lpTitle: 窗口标题

    char szPath[MAX_PATH]={0};
    BROWSEINFO m_bi;

    m_bi.ulFlags = BIF_RETURNONLYFSDIRS  | BIF_STATUSTEXT;
    m_bi.hwndOwner = hWnd;
    m_bi.pidlRoot = NULL;
```

```
    m_bi.lpszTitle = lpTitle;
    m_bi.lpfn = NULL;
    m_bi.lParam = NULL;
    m_bi.pszDisplayName = szPath;

    LPITEMIDLIST pidl = ::SHBrowseForFolder( &m_bi );
    if ( pidl )
    {
        if( !::SHGetPathFromIDList ( pidl, szPath ) )  szPath[0]=0;

        IMalloc * pMalloc = NULL;
        if ( SUCCEEDED ( ::SHGetMalloc( &pMalloc ) ) ) // 取得 IMalloc 分配器接口
        {
            pMalloc->Free( pidl );    // 释放内存
            pMalloc->Release();      // 释放接口
        }
    }
    return szPath;
}
```

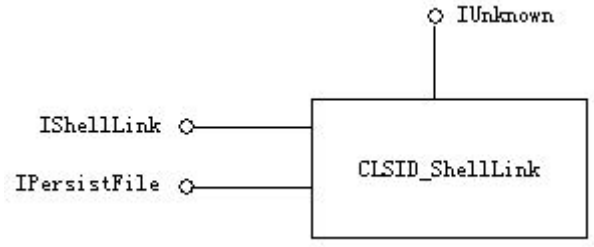
示例三、在窗口中显示一幅 JPG 图象。

```
void CxxxView::OnDraw(CDC* pDC)
{
         ::CoInitialize(NULL);  // COM 初始化
         HRESULT hr;
         CFile file;

         file.Open( "c:\\aa.jpg", CFile::modeRead | CFile::shareDenyNone );  // 读
入文件内容
         DWORD dwSize = file.GetLength();
         HGLOBAL hMem = ::GlobalAlloc( GMEM_MOVEABLE, dwSize );
         LPVOID lpBuf = ::GlobalLock( hMem );
         file.ReadHuge( lpBuf, dwSize );
         file.Close();
         ::GlobalUnlock( hMem );

         IStream * pStream = NULL;
         IPicture * pPicture = NULL;
```

```
        // 由 HGLOBAL 得到 IStream，参数 TRUE 表示释放 IStream 的同时，释放内存
        hr = ::CreateStreamOnHGlobal( hMem, TRUE, &pStream );
        ASSERT ( SUCCEEDED(hr) );

        hr = ::OleLoadPicture( pStream, dwSize, TRUE, IID_IPicture, ( LPVOID
* )&pPicture );
        ASSERT(hr==S_OK);

        long nWidth,nHeight;  // 宽高，MM_HIMETRIC 模式，单位是 0.01 毫米
        pPicture->get_Width( &nWidth );   // 宽
        pPicture->get_Height( &nHeight );  // 高

        ////////原大显示//////
        CSize sz( nWidth, nHeight );
        pDC->HIMETRICtoDP( &sz );  // 转换 MM_HIMETRIC 模式单位为 MM_TEXT 像素单位
        pPicture->Render(pDC->m_hDC,0,0,sz.cx,sz.cy,
                0,nHeight,nWidth,-nHeight,NULL);

        ////////按窗口尺寸显示////////
//      CRect rect;       GetClientRect(&rect);
//      pPicture->Render(pDC->m_hDC,0,0,rect.Width(),rect.Height(),
//              0,nHeight,nWidth,-nHeight,NULL);

        if ( pPicture ) pPicture->Release();// 释放 IPicture 指针
        if ( pStream ) pStream->Release();  // 释放 IStream 指针，同时释放了 hMem

        ::CoUninitialize();
}
```

示例四、在桌面建立快捷方式

在阅读代码之前，先看一下关于“快捷方式”组件的结构示意图。



图二、快捷方式组件的接口结构示意图

从结构图中可以看出，“快捷方式”组件(CLSID_ShellLink)，有 3 个（其实不止）接口，

每个接口完成一组相关功能的函数。IShellLink 接口(IID_IShellLink)提供快捷方式的参数读写功能（见图三），IPersistFile 接口(IID_IPersistFile)提供快捷方式持续性文件的读写功能。对象的持续性(注 5)，是一个非常常用，并且功能强大的接口家族。但今天，我们只要了解其中两函数，就可以了：IPersistFile::Save()和 IPersistFile:Load()。(注 6)

图三、快捷方式中的各种属性

```
#include < atlconv.h >
void CreateShortcut(LPCTSTR lpszExe, LPCTSTR lpszLnk)
{
        // 建立块捷方式
        // 参数 lpszExe: EXE 文件全路径名
        // 参数 lpszLnk: 快捷方式文件全路径名

        ::CoInitialize( NULL );

        IShellLink * psl = NULL;
        IPersistFile * ppf = NULL;

        HRESULT hr = ::CoCreateInstance(  // 启动组件
                CLSID_ShellLink,       // 快捷方式 CLSID
                NULL,                  // 聚合用(注 4)
                CLSCTX_INPROC_SERVER, // 进程内(Shell32.dll)服务
```

```
                IID_IShellLink,         // IShellLink 的 IID
                (LPVOID *)&psl );       // 得到接口指针

        if ( SUCCEEDED(hr) )
        {
                psl->SetPath( lpszExe );  // 全路径程序名
//              psl->SetArguments();      // 命令行参数
//              psl->SetDescription();    // 备注
//              psl->SetHotkey();         // 快捷键
//              psl->SetIconLocation();   // 图标
//              psl->SetShowCmd();        // 窗口尺寸

                // 根据 EXE 的文件名，得到目录名
                TCHAR szWorkPath[ MAX_PATH ];
                ::lstrcpy( szWorkPath, lpszExe );
                LPTSTR lp = szWorkPath;
                while( *lp )    lp++;
                while( ''\\'' != *lp )    lp--;
                *lp=0;

                // 设置 EXE 程序的默认工作目录
                psl->SetWorkingDirectory( szWorkPath );

                hr = psl->QueryInterface(  // 查找持续性文件接口指针
                        IID_IPersistFile,       // 持续性接口 IID
                        (LPVOID *)&ppf );       // 得到接口指针

                if ( SUCCEEDED(hr) )
                {
                        USES_CONVERSION;        // 转换为 UNICODE 字符串
                        ppf->Save( T2COLE( lpszLnk ), TRUE );  // 保存
                }
        }
        if ( ppf )          ppf->Release();
        if ( psl )          psl->Release();

        ::CoUninitialize();
}
```

```
void OnXXX()
{
        CreateShortcut(
                _T("c:\\winnt\\notepad.exe"),  // 记事本程序。注意，你的系统是否也是这个目录？
                _T("c:\\Documents and Settings\\Administrator\\桌面\\我的记事本.lnk")
        );
        // 桌面上建立快捷方式(lnk)文件的全路径名。注意，你的系统是否也是这个目录？
        // 如果用程序实现寻找桌面的路径，则可以查注册表
        // HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
}
```

七、小结

本回介绍的内容比较实用。大家不要只抄袭代码，而一定要理解它。结合 MSDN 的说明去思索代码、理解其含义。好了，想方设法把代码忘掉！三天后（如过你还没有忘记，那就再过三天），你在不参考示例代码，但可以随便翻阅 MSDN 的情况下，自己能独立地再次完成这四个例程，那么恭喜你，你已经入门了:0) 从下回开始，我们要用 ATL 做 COM 的开发工作啦，您老人家准备好了吗？

作业，留作业啦.....

1、你已经学会如何建立快捷方式了，那么你知道怎么读取它的属性吗？（如果写不出这个程序，那么你就不用继续学习了。因为.....动点脑筋呀！我还没有见过象你这么笨的学生呢！）

2、示例程序三中使用了 IPicture 接口显示一个 JPG 图象。那么你现在去完成一个功能，把 JPG 文件转换为 BMP 文件。

注 1：智能指针的概念和用法，后续介绍。

注 2：IDL 文件，下回就要介绍啦。

注 3：东北话，想干什么都可以，反正我不管啦。

注 4：聚合，也许在第 30 回中介绍吧:-)

注 5：持续性，IPersistXXXXXX 是一个非常强大的接口家族，后续介绍。

注 6：想知道 IShellLink、IPersistFile 接口的所有函数吗？别愣着，快去看 MSDN 呀.....

COM 组件设计与应用（五）

用 ATL 写第一个组件

下载源代码

一、前言

1、如果你在使用 vc5.0 及以前的版本，请你升级为 vc6.0 或 vc.net 2003；

2、如果你在使用 vc6.0 (ATL 3.0)请阅读本回内容；

3、如果你在使用 vc.net(ATL 7.0)请阅读下回内容；(当然读读本文内容也不错)

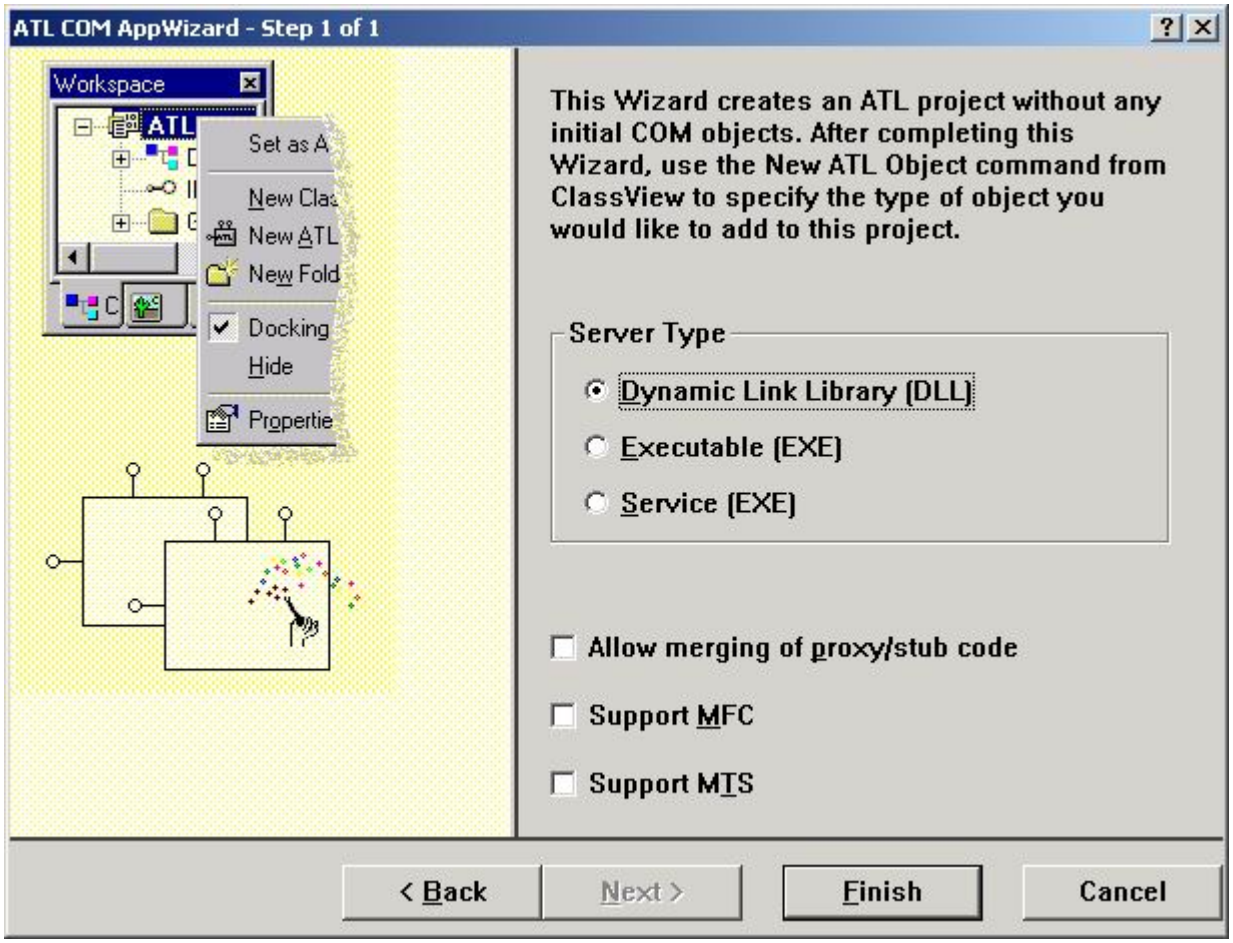
4、这第一个组件，除了所有 COM 组件必须的 IUnknown 接口外，我们再实现一个自己定义的接口 IFun，它有两个函数： Add()完成两个数值的加法，Cat()完成两个字符串的连接。

5、下面.....好好听讲！开始了:-)

二、建立 ATL 工程

步骤 2.1：建立一个工作区(WorkSpace)。

步骤 2.2：在工作区中，建立一个 ATL 工程(Project)。示例程序叫 Simple1，并选择 DLL 方式，见图一。



图一、建立 ATL DLL 工程

Dynamic Link Library(DLL) 表示建立一个 DLL 的组件程序。

Executable(EXE) 表示建立一个 EXE 的组件程序。

Service(EXE) 表示建立一个服务程序，系统启动后就会加载并执行的程序。

Allow merging of proxy/stub code 选择该项表示把“代理/存根”代码合并到组件程序中，否则需要单独编译，单独注册代理存根程序。代理/存根，这个是什么概念？还记得我们在上回书中介绍的吗？当调用者调用进程外或远程组件功能的时候，其实是代理/存根负责数据交换的。关于代理/存根的具体变成和操作，以后再说啦……

Support MFC 除非有特殊的原因，我们写 ATL 程序，最好不要选择该项。你可能会说，如果没有 MFC 的支持，那 CString 怎么办呀？告诉你个秘密吧，一般人我都不告诉他，我后半辈子就靠着这个秘密活着了：

1、你会 STL 吗？可以用 STL 中的 string 代替；

2、自己写个 MyString 类，嘿嘿；

3、悄悄地、秘密地、不要告诉别人（特别是别告诉微软），把 MFC 中的 CString 源码拿过来用；

4、使用 CComBSTR 类，至少也能简化我们字符串操作；

5、直接用 API 操作字符串，反正我们大家学习 C 语言的时候，都是从这里干起的。（等于没说，呵呵）

Support MTS 支持事务处理，也就是是否支持 COM+ 功能。COM+ 也许在第 99 回介绍吧。

三、增加 ATL 对象类

步骤 3.1：菜单 Insert\New ATL Object...（或者用鼠标右键在 ClassView 卡片中弹出菜单）并选择 Object 分类，选中 Simple Object 项目。见图二。

图二、选择建立简单 COM 对象

Category Object 普通组件。其中可以选择的组件对象类型很多，但本质上，就是让向导帮我们默认加上一些接口。比如我们选 "Simple Object"，则向导给我们的组件加上 IUnknown 接口；我们选 "Internet Explorer Object"，则向导除了加上 IUnknown 接口外，再增加一个给 IE 所使用的 IObjectWithSite 接口。当然了，我们完全可以手工增加任何接口。

Category Controls ActiveX 控件。其中可以选择的 ActiveX 类型也很多。我们在后续的专门介绍 ActiveX 编程中再讨论。

Category Miscellaneous 辅助杂类组件。

Categroy Data Access 数据库类组件(我最讨厌数据库编程了，所以我也不会)。

步骤 3.2：增加自定义类 CFun(接口 IFun) ,见图三。

图三、输入类中的各项名称

其实，我们只需要输入短名(Short Name)，其它的项目会自动填写。没什么多说的，只请大家注意一下 ProgID 项，默认的 ProgID 构造方式为“工程名.短名”。

步骤 3.3：填写接口属性，见图四。

图四、接口属性

Threading Model 选择组件支持的线程模型。COM 中的线程，我认为是最讨厌，最复杂的部分。COM 线程和公寓的概念，留待后续介绍。现在吗.....大家都选 Apartment，它代表什么那？简单地说：当在线程中调用组件函数的时候，这些调用会排队进行。因此，这种模式下，我们可以暂时不用考虑同步的问题。(注 1)

Interface 接口基本类型。Dual 表示支持双接口(注 2)，这个非常 非常重要，非常非常常用，但我们今天不讲。Custom 表示自定义借口。**切记！切记！我们的这第一个 COM 程序中，一定要选择它！！！！**（如果你选错了，请删除全部内容，重新来过。）

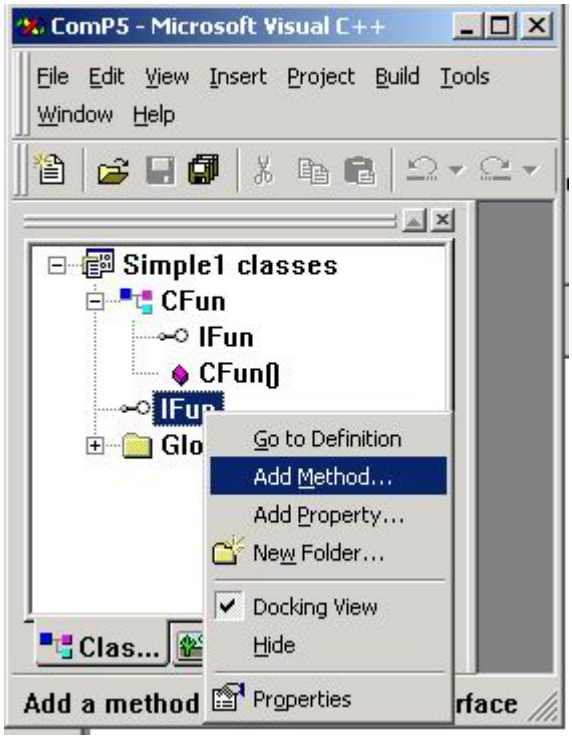
Aggregation 我们写的组件，将来是否允许被别人聚合(注 3)使用。Only 表示必须被聚合才能使用，有点类似 C++ 中的纯虚类，你要是总工程师，只负责设计但不亲自写代码的话，才选择它。

Support ISupportErrorInfo 是否支持丰富信息的错误处理接口。以后就讲。

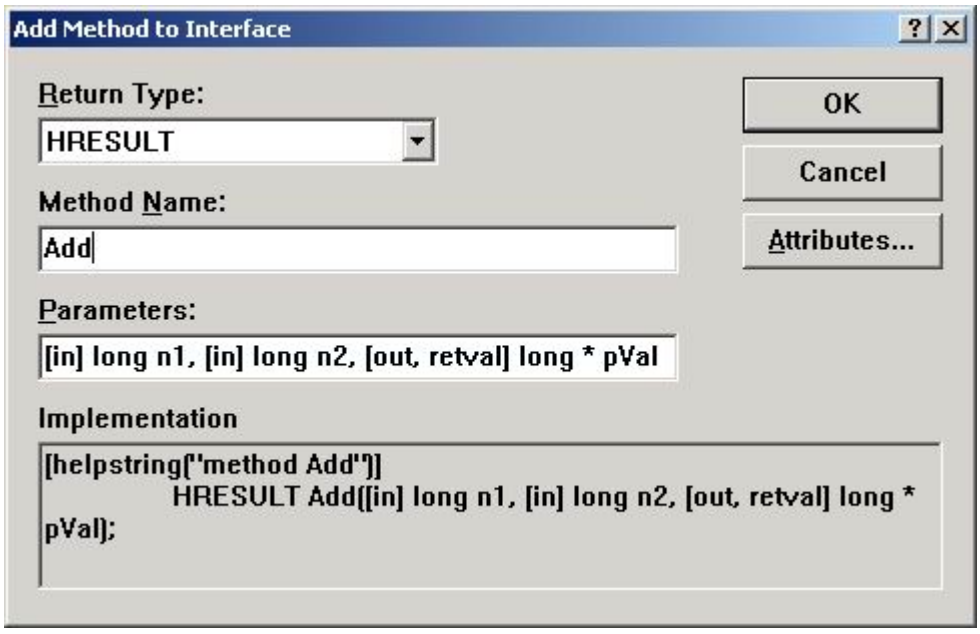
Support Connection Points 是否支持连接点接口（事件、回调）。以后就讲。

Free Threaded Marshaler 以后也不讲，就算打死你，我也不说！(注 4)

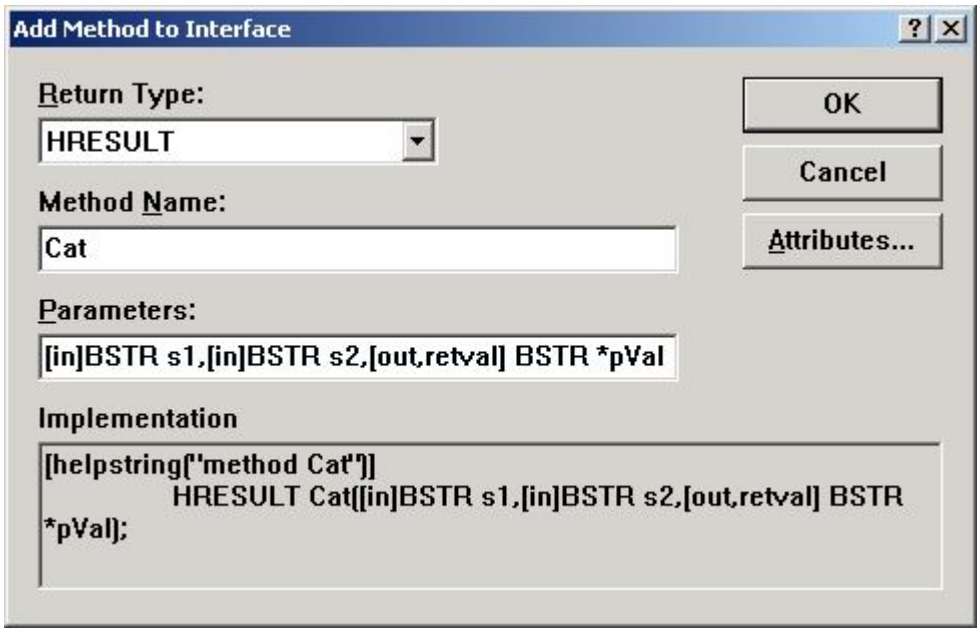
四、添加接口函数



图五、调出增加接口方法的菜单

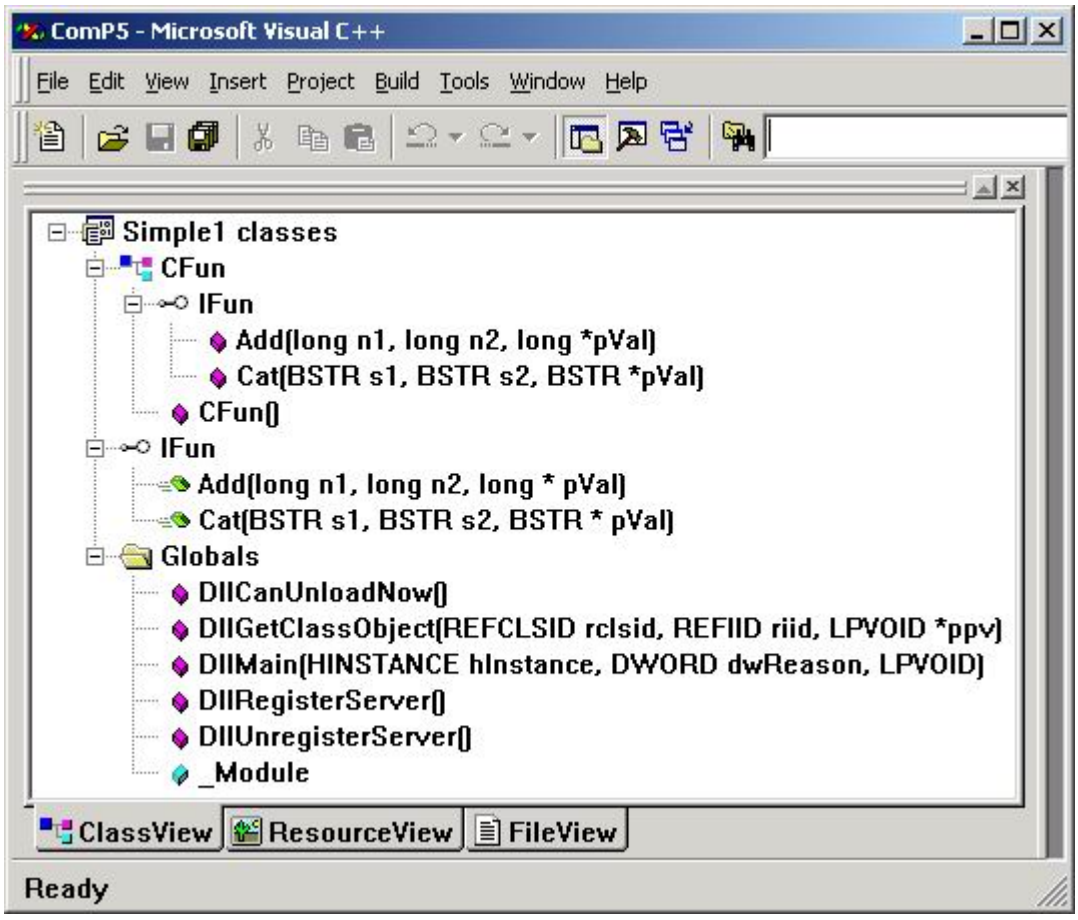


图六、增加接口函数 Add



图七、增加接口函数 Cat

请严格按照图六的方式，增加 Add()函数；由于图七中增加 Cat()函数的参数比较长，我没有适当的输入空格，请大家自己输入的时候注意一下。[in]表示参数方向是输入；[out]表示参数方向是输出；[out,retval]表示参数方向是输出，同时可以作为函数运算结果的返回值。一个函数中，可以有多个[in]、[out]，但[retval]只能有一个，并且要和[out]组合后在最后一个位置。(注 5)



图八、接口函数定义完成后的图示

我们都知道，要想改变 C++ 中的类函数，需要修改两个地方：一是头文件(.h)中类的函数声明，二是函数体(.cpp)文件的实现处。而我们现在用 ATL 写组件程序，则还要修改一个地方，就是接口定义(IDL)文件。别着急 IDL 下次就要讨论啦。

由于 vc6.0 的 BUG，导致大家在增加完接口和接口函数后，可能不会向上图（图八）所表现的样式。解决方法：

1	关闭工程，然后重新打开	该方法常常有效
2	关闭 IDE，然后重新运行	
3	打开 IDL 文件，检查接口函数是否正确，如不正确请修改	
4	打开 IDL 文件，随便修改一下(加一个空格，再删除这个空格)，然后保存	该方法常常有效
5	打开 h/cpp 文件，检查函数是否存在或是否正确，有则改之	无则嘉勉，不说完这个成语心理别扭
6	删除 IDL/H/CPP 中的接口函数，然后	再来一遍
7	重新建立工程、重新安装 vc、重新安装 windows、砸	砸！

	计算机	

五、实现接口函数

鼠标双点图八中 CFun\IFun\Add(...)就可以开始输入函数实现了：

```
STDMETHODIMP CFun::Add(long n1, long n2, long *pVal)
{
	*pVal = n1 + n2;
	return S_OK;
}
```

这个太简单了，不再浪费“口条”。下面我们实现字符串连接的 Cat()函数：

```
STDMETHODIMP CFun::Cat(BSTR s1, BSTR s2, BSTR *pVal)
{
	int nLen1 = ::SysStringLen( s1 );    // s1 的字符长度
	int nLen2 = ::SysStringLen( s2 );    // s2 的字符长度

	*pVal = ::SysAllocStringLen( s1, nLen1 + nLen2 );      // 构造新的 BSTR 同时把 s1 先保存进去
	if( nLen2 )
	{
		::memcpy( *pVal + nLen1, s2, nLen2 * sizeof(WCHAR) ); // 然后把 s2 再连接进去
//		wcscat( *pVal, s2 );
	}

	return S_OK;
}
```

学生：上面的函数实现，完全是调用基本的 API 方式完成的。

老师：是的，说实话，的确比较烦琐。

学生：我们是用 memcpy()完成连接第二个字符串功能的，那么为什么不用函数 wcscat()那？

老师：多数情况下可以，但你需要知道：由于 BSTR 包含有字符串长度，因此实际的 BSTR 字符串内容中是可以存储 L"\0"的，而函数 wcscat() 是以 L"\0"作为复制结束标志，因此可能会丢失数据。明白了吗？

学生：明白，明白。我看过《COM 组件设计与应用(三)之数据类型》后就明白了。那么老师，有没有简单一些的方法那？

老师：有呀，你看.....

```
STDMETHODIMP CFun::Cat(BSTR s1, BSTR s2, BSTR *pVal)
{
        CComBSTR sResult( s1 );
        sResult.AppendBSTR( s2 );

        *pVal = sResult.Copy();
//      *pVal = sResult.Detach();

        return S_OK;
}
```

学生：哈哈，好！使用了 CComBSTR，这个就简单多了。CComBSTR::Copy()和 CComBSTR::Detach()有什么区别？

老师：CComBSTR::Copy() 会制造一个 BSTR 的副本，另外 CComBSTR::CopyTo()也有类似功能。而 CComBSTR::Detach()是使对象与内部的 BSTR 指针剥离，这个函数由于没有复制过程，因此速度稍微快一点点。但要注意，一但剥离后，就不能再使用该对象啦。

学生：老师，您讲的太牛啦，我对您的敬仰如巍巍泰山，直入云霄.....

老师：STOP，STOP！留作业啦.....

1、自己先按照今天讲的内容写出这个组件；

2、不管你懂不懂，一定要去观察 IDL 文件，CPP 文件；

3、编译后，看都产生了些什么文件？如果是文本的文件，就打开看看；

4、下载本文的示例程序(vc6.0 版本)编译运行，看看效果。然后预习一下示例程序中的调用方法；

学生：知道啦，快下课吧，我要上厕所，我都憋的不行了.....

老师：下课！别忘了顶我的帖子呀.....

六、小结

本回介绍第一个 ATL 组件程序的建立步骤，而如何使用该组件，敬请关注《COM 组件设计与应用(七)》。

注 1：Apartment，系统通过隐藏的窗口消息来排队组件调用，因此我们可以暂时不考虑同步问题。注意，是暂时哈。

注 2：双接口表示在一个接口中，同时支持自定义接口和 IDispatch 接口。以后，以后，以后就讲。因为双接口非常重要，我们以后会天天讲、夜夜讲、常常讲------简称“三讲”:)

注 3：组件的重用方法有 2 个，聚合和包容。

注 4：名称的功能很好听，但微软根本就没有实现。

注 5：这些都是 IDL 文件中的概念，以后用到什么，就介绍什么。

COM 组件设计与应用（六）
用 ATL 写第一个组件

下载源代码

一、前言

1、与 《COM 组件设计与应用(五)》的内容基本一致。但本回讲解的是在 vc.net 2003 下的使用方法，即使你不再使用 vc6.0，也请和上一回的内容，参照比对。

2、这第一个组件，除了所有 COM 组件必须的 IUnknown 接口外，我们再实现一个自己定义的接口 IFun，它有两个函数： Add()完成两个数值的加法，Cat()完成两个字符串的连接。

3、下面.....好好听讲！开始了:-)

二、建立 ATL 工程

步骤 2.1：建立一个解决方案。

步骤 2.2：在 该解决方案中，新建一个 vc++ 的 ATL 项目。示例程序叫 Simple2，并选择 DLL 方式，见图一、图二。

图一、新建 ATL 项目

图二、选择非属性化的 DLL 组件类型

属性化 属性化编程，是未来的方向，但我们现在先不要选它。

动态链接库(DLL) 选择它。

可执行文件(EXE) 以后再讲。

服务(EXE) 表示建立一个系统服务组件程序，系统启动后就会加载并执行的程序。

允许合并代理/存根(stub)代码 选择该项表示把“代理/存根”代码合并到组件程序中，否则需要单独编译，单独注册代理存根程序。代理/存根，这个是什么概念？还记得我们在上回书中介绍的吗？当调用者调用进程外或远程组件功能的时候，其实是代理/存根负责数据交换的。关于代理/存根的具体变成和操作，以后再说啦.....

支持 MFC 除非有特殊的原因，我们写 ATL 程序，最好不要选择该项。你可能会说，如果没有 MFC 的支持，那 CString 怎么办呀？告诉你个秘密吧，一般人我都不告诉他，我后半辈子就靠着这个秘密活着了：

1、你会 STL 吗？可以用 STL 中的 string 代替；

2、自己写个 MyString 类，嘿嘿；

3、悄悄地、秘密地、不要告诉别人（特别是别告诉微软），把 MFC 中的 CString 源码拿过来用；

4、使用 CComBSTR 类，至少也能简化我们字符串操作；

5、直接用 API 操作字符串，反正我们大家学习 C 语言的时候，都是从这里干起的。（等于没说，呵呵）

支持 COM+ 1.0 支持事务处理的 COM+ 功能。COM+ 也许在第 99 回介绍吧。

三、添加 ATL 对象类

步骤 3.1：菜单"项目\添加类..."（或者用鼠标右键在 项目中弹出菜单"添加\添加类..."）并选择 ATL 简单对象。见图三。

图三、选择建立 ATL 简单对象

除了简单对象(只实现了 IUnknown 接口)，还可以选择“ATL 控件”(ActiveX，实现了 10 多个接口).....可以选择的组件对象类型很多，但本质上，就是让向导帮我们默认加上一些接口。在以后的文章中，陆续介绍吧。

步骤 3.2：增加自定义类 CFun(接口 IFun) ,见图四。

图四、填写名称

其实，我们只需要输入简称，其它的项目会自动填写。没什么多说的，只请大家注意一下 ProgID 项，默认的 ProgID 构造方式为“项目名.简称名”。

步骤 3.3：填写接口属性选项，见图 五。

图五、接口选项

线程模型 COM 中的线程，我认为是最讨厌，最复杂的部分。COM 线程和公寓的概念，留待后续介绍。现在吗.....大家都选"单元"(Apartment)，它代表什么那？简单地说：当在线程中调用组件函数的时候，这些调用会排队进行。因此，这种模式下，我们可以暂时不用考虑同步的问题。(注 1)

接口。双重(Dual)，这个非常 非常重要，非常非常常用，但我们今天不讲(注 2)。**切记！切记！我们的这第一个 COM 程序中，一定要选择“自定义”！！！！**（如果你选错了，请删除全部内容，重新来过。）

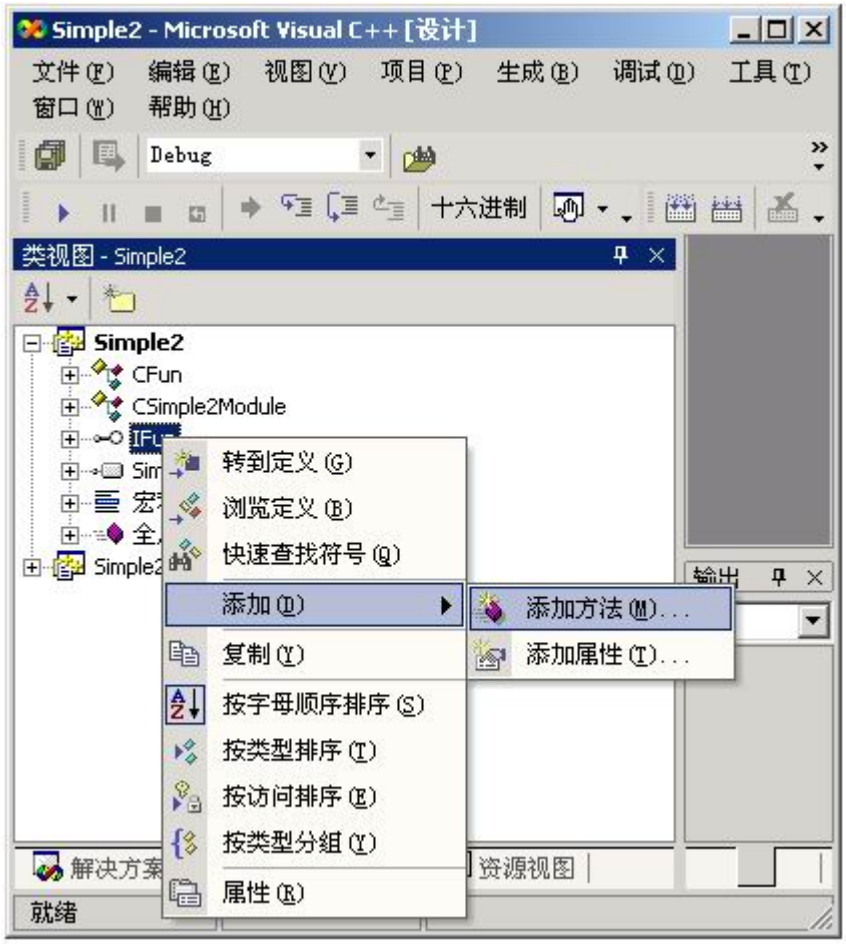
聚合 我们写的组件，将来是否允许被别人聚合(注 3)使用。“只能创建为聚合”，有点类似 C++ 中的纯虚类，你要是总工程师，只负责设计但不亲自写代码的话，才选择它。

ISupportErrorInfo 是否支持丰富信息的错误处理接口。以后就讲。

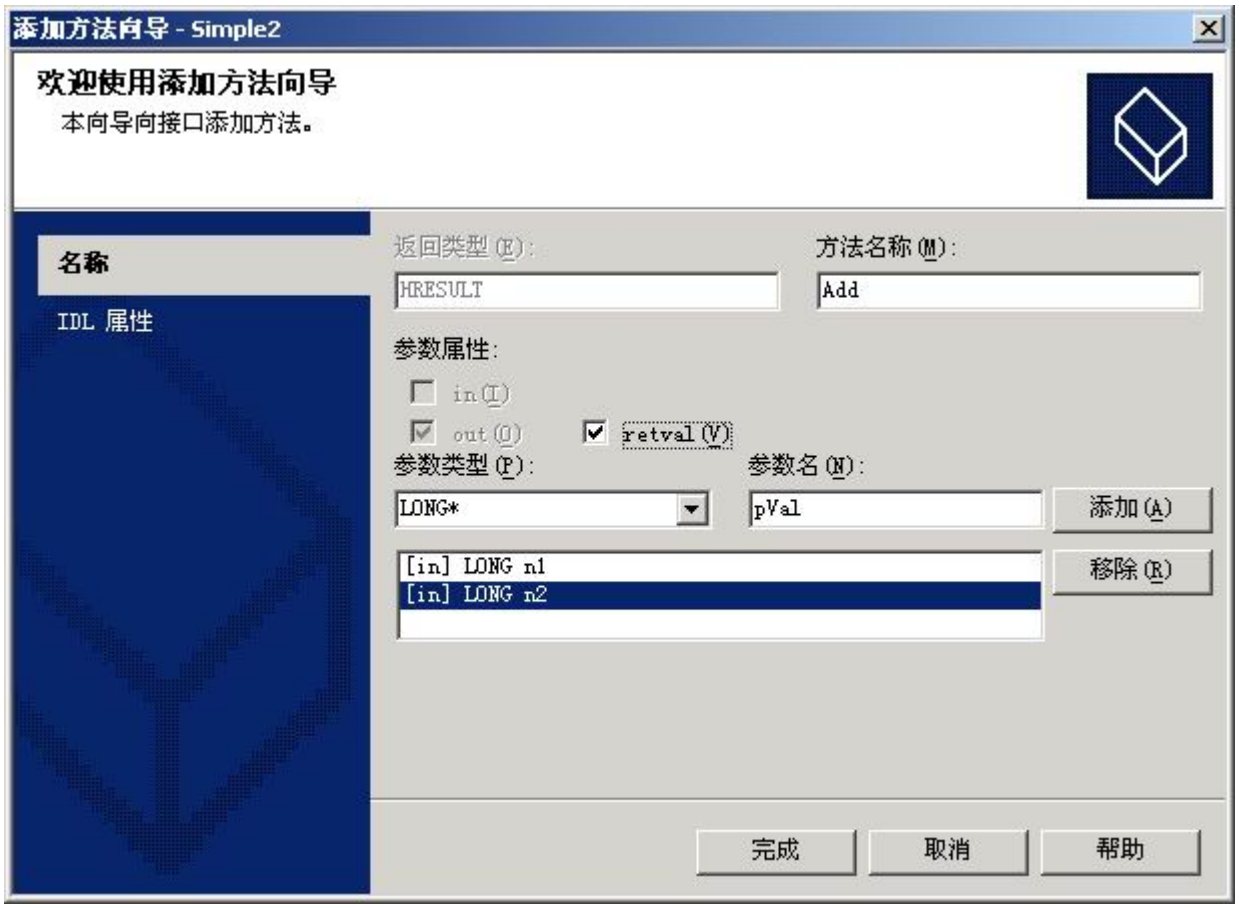
连接点 是否支持连接点接口（事件、回调）。以后就讲。

IObjectWithSite 是否支持 IE 的调用

四、添加接口函数

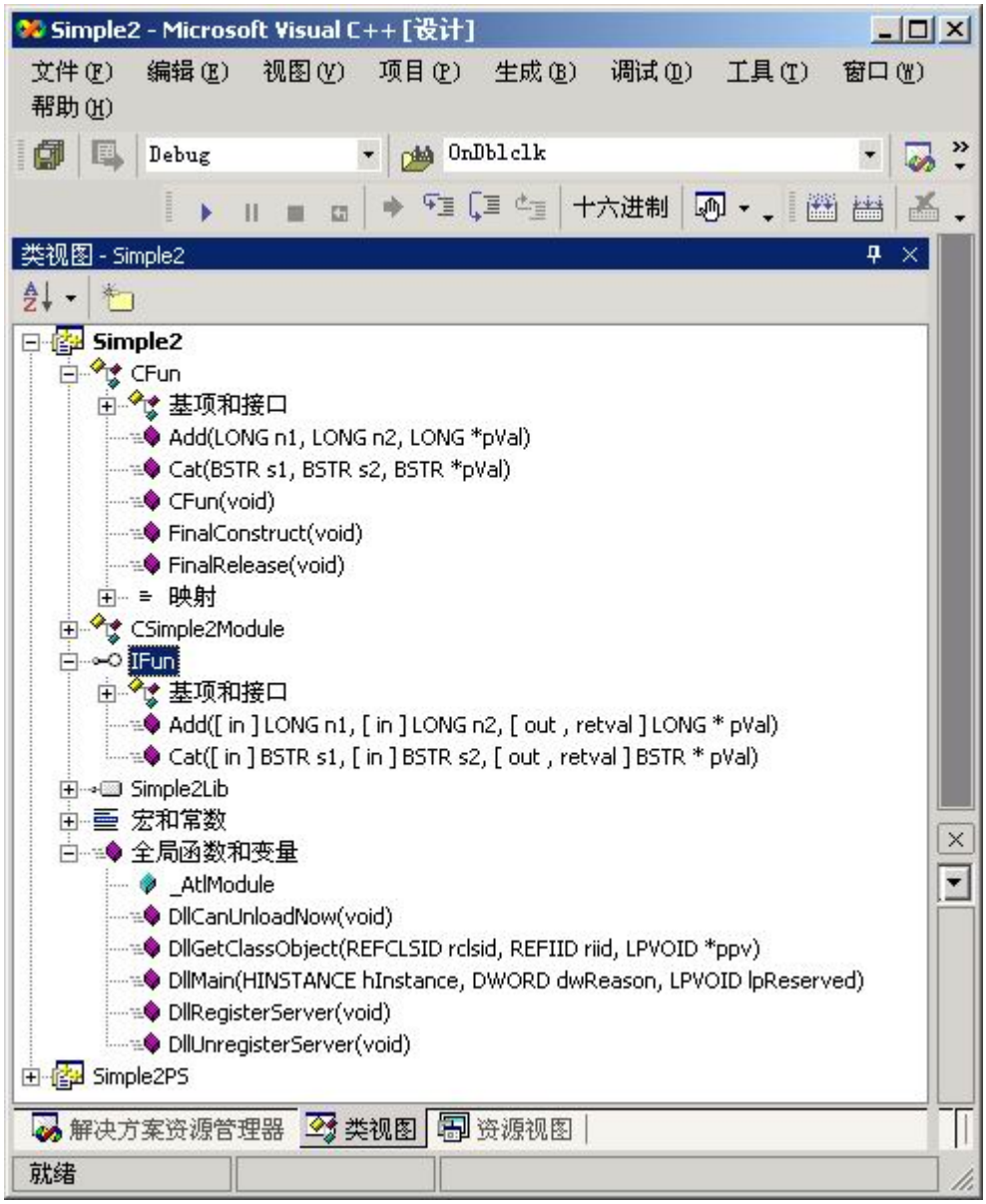


图六、调出增加接口方法的菜单



图七、增加接口函数 Add

请按照图示的方法，增加 Add()函数，增加 Cat()函数 。[in]表示参数方向是输入；[out]表示参数方向是输出；[out,retval]表示参数方向是输出，同时可以作为函数运算结果的返回值。一个函数中，可以有多个[in]、[out]，但[retval]只能有一个，并且要和[out]组合后在最后一个位置。(注 4)



图八、接口函数定义完成后的图示

我们都知道，要想改变 C++ 中的类函数，需要修改两个地方：一是头文件(.h)中类的函数声明，二是函数体(.cpp)文件的实现处。而我们现在用 ATL 写组件程序，则还要修改一个地方，就是接口定义(IDL)文件。别着急 IDL 下次就要讨论啦。

五、实现接口函数

鼠标双点图八中 CFun\基项和接口\Add(...)就可以开始输入函数实现了：

```
STDMETHODIMP CFun::Add(long n1, long n2, long *pVal)
{
        *pVal = n1 + n2;
```

```
        return S_OK;
}
```

这个太简单了，不再浪费“口条”。下面我们实现字符串连接的 Cat()函数：

```
STDMETHODIMP CFun::Cat(BSTR s1, BSTR s2, BSTR *pVal)
{
        int nLen1 = ::SysStringLen( s1 );    // s1 的字符长度
        int nLen2 = ::SysStringLen( s2 );    // s2 的字符长度

        *pVal = ::SysAllocStringLen( s1, nLen1 + nLen2 );// 构造新的 BSTR 同时把 s1 先保存进去
        if( nLen2 )
        {
                ::memcpy( *pVal + nLen1, s2, nLen2 * sizeof(WCHAR) ); // 然后把 s2 再连接进去
//              wcscat( *pVal, s2 );
        }

        return S_OK;
}
```

学生：上面的函数实现，完全是调用基本的 API 方式完成的。

老师：是的，说实话，的确比较烦琐。

学生：我们是用 memcpy()完成连接第二个字符串功能的，那么为什么不用函数 wcscat()那？

老师：多数情况下可以，但你需要知道：由于 BSTR 包含有字符串长度，因此实际的 BSTR 字符串内容中是可以存储 L"\0"的，而函数 wcscat() 是以 L"\0"作为复制结束标志，因此可能会丢失数据。明白了吗？

学生：明白，明白。我看过《COM 组件设计与应用(三)之数据类型》后就明白了。那么老师，有没有简单一些的方法那？

老师：有呀，你看.....

```
STDMETHODIMP CFun::Cat(BSTR s1, BSTR s2, BSTR *pVal)
{
        CComBSTR sResult( s1 );
        sResult.AppendBSTR( s2 );

        *pVal = sResult.Copy();
//      *pVal = sResult.Detach();
```

```
        return S_OK;
}
```

学生：哈哈，好！使用了 CComBSTR，这个就简单多了。CComBSTR::Copy()和 CComBSTR::Detach()有什么区别？

老师：CComBSTR::Copy() 会制造一个 BSTR 的副本，另外 CComBSTR::CopyTo()也有类似功能。而 CComBSTR::Detach()是使对象与内部的 BSTR 指针剥离，这个函数由于没有复制过程，因此速度稍微快一点点。但要注意，一但剥离后，就不能再使用该对象啦。

学生：老师，您讲的太牛啦，我对您的敬仰如巍巍泰山，直入云霄……

老师：STOP，STOP！留作业啦……

1、自己先按照今天讲的内容写出这个组件；

2、不管你懂不懂，一定要去观察 IDL 文件，CPP 文件；

3、编译后，看都产生了些什么文件？如果是文本的文件，就打开看看；

4、下载本文的示例程序(vc.net 2003 版本)编译运行，看看效果。然后预习一下示例程序中的调用方法；

学生：知道啦，快下课吧，我要上厕所，我都憋的不行了……

老师：下课！别忘了顶我的帖子呀……

六、小结

本回介绍第一个 ATL 组件程序的建立步骤，而如何使用该组件，敬请关注《COM 组件设计与应用(七)》。

注 1：Apartment，系统通过隐藏的窗口消息来排队组件调用，因此我们可以暂时不考虑同步问题。注意，是暂时哈。

注 2：双接口表示在一个接口中，同时支持自定义接口和 IDispatch 接口。以后，以后，以后就讲。因为双接口非常重要，我们以后会天天讲、夜夜讲、常常讲------简称“三讲”:)

注 3：组件的重用方法有 2 个，聚合和包容。

注 4：这些都是 IDL 文件中的概念，以后用到什么，就介绍什么。

COM 组件设计与应用（七）

编译、注册、调用

一、前言

上两回中，咱们用 ATL 写了第一个 COM 组件程序，这回中，主要介绍编译、注册和调用方法。示例程序你已经下载了吗？如果还没有下载，vc6.0 的用户点这里，vc.net 的用户点这里。

二、关于编译

2-1 最小依赖

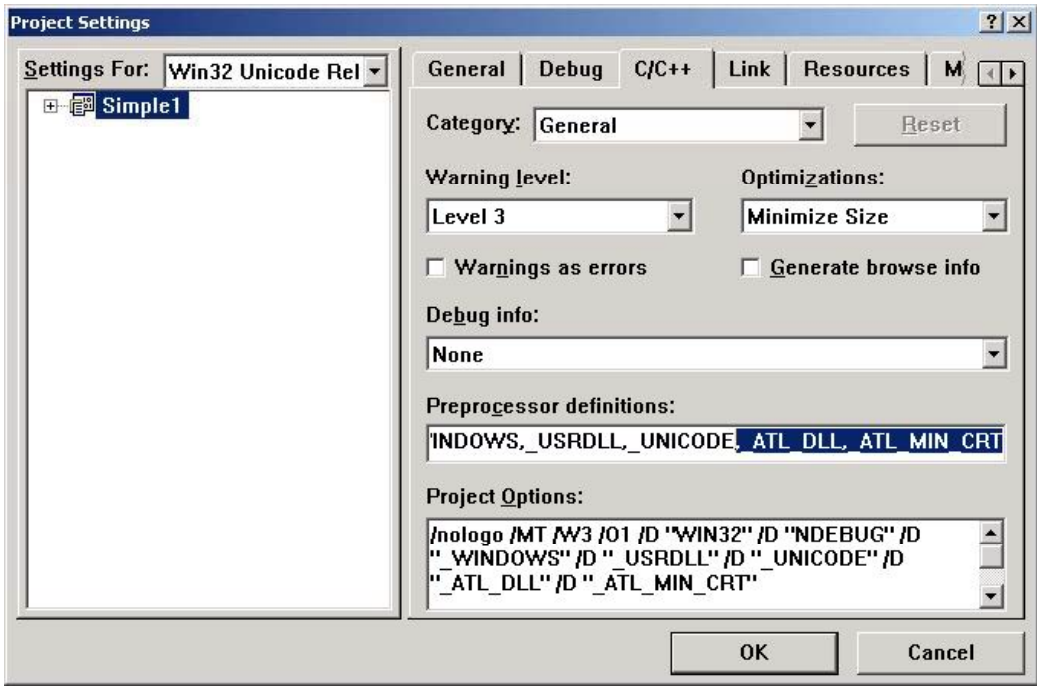
“最小依赖”，表示编译器会把 ATL 中必须使用的一些函数静态连接到目标程序中。

这样目标文件尺寸会稍大，但独立性更强，安装方便；反之系统执行的时候需要有 ATL.DLL 文件的支持。如何选择设置为“最小依赖”呢？答案是：删除预定义宏“_ATL_DLL”，操作方法见图一、图二。

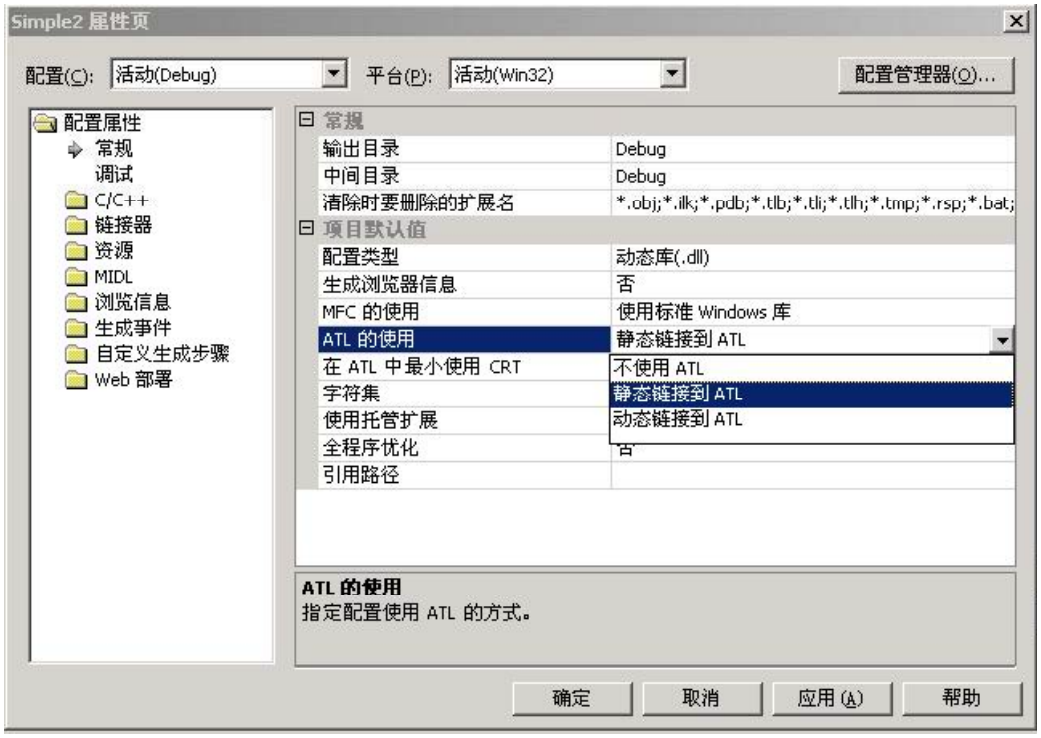
图一、在 vc6.0 中，设置方法

图二、在 vc.net 2003 中，设置方法

2-2 CRT 库



如果在 ATL 组件程序中调用了 CRT 的运行时刻库函数，比如开平方 sqrt()，那么编



译的时候可能会报错“error LNK2001: unresolved external symbol _main”。怎么办？删除预定义宏“_ATL_MIN_CRT”！操作方法也见图一、图二。（vc.net 2003 中的这个项目

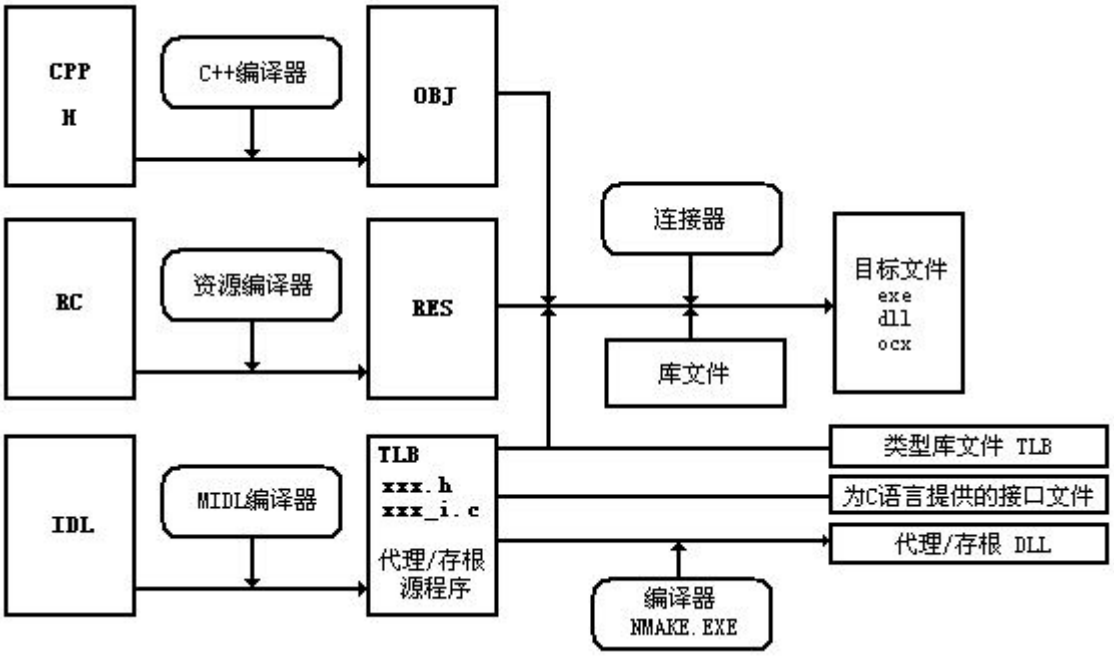
属性叫“在 ATL 中最小使用 CRT”）

2-3 MBCS/UNICODE

这个不多说了，在预定义宏中，分别使用 _MBCS 或 _UNICODE。

2-4 IDL 的编译

COM 在设计初期，就定了一个目标：要能实现跨语言的调用。既然是跨语言的，那么组件的接口描述就必须在任何语言环境中都要能够认识。怎么办？用 .h 文件描述？------ C 语言程序员笑了，真方便！BASIC 程序员哭了:-(因此，微软使用了一个新的文件格式---IDL 文件（接口定义描述语言）。IDL 是一个文本文件，它的语言语法比较简单，很象 C。具体 IDL 文件的讲解，见下一回《COM 组件设计与应用（八）之添加新接口》。IDL 经过编译，生成二进制的等价类型库文件 TLB 提供给其它语言来使用。图三示意了 ATL COM 程序编译的过程：



图三、ATL 组件程序编译过程

说明 1：编译后，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，我们将来在 #import 引入类型库的时候，既可以指定 TLB 文件，也可以指定目标文件；

说明 2：我们作为 C/C++ 的程序员，还算是比较幸福的。因为 IDL 编译后，特意为我们提供了 C 语言形式的接口文件。

说明 3：IDL 编译后生成代理/存根源程序，有：dlldata.c、xxx_p.c、xxxps.def、xxxps.mak，我们可以用 NMAKE.EXE 再次编译来产生真正的代理/存根 DLL 目标文件(注 1)。

三、关于注册

情况 1：当我们使用 ATL 编写组件程序，注册不用我们来负责。编译成功后，IDE 会帮我们自动注册；

情况 2：当我们使用 MFC 编写组件程序，由于编译器不知道你写的是否是 COM 组件，所以它不会帮我们自动注册。这个时候，我们可以执行菜单“Tools\Register Control”来注册。

情况 3：当我们写一个具有 COM 功能的 EXE 程序时，注册的方法就是运行一次这个程序；

情况 4：当我们需要使用第三方提供的组件程序时，可以命令行运行“regsvr32.exe 文件名”来注册。顺便说一句，反注册的方法是“regsvr32.exe /u 文件名”；

情况 5：当我们需要在程序中（比如安装程序）需要执行注册，那么：

```
typedef HRESULT (WINAPI * FREG)();
TCHAR szWorkPath[ MAX_PATH ];

::GetCurrentDirectory( sizeof(szWorkPath), szWorkPath );// 保存当前进程的工作目录
::SetCurrentDirectory( 组件目录 );  // 切换到组件的目录

HMODULE hDLL = ::LoadLibrary( 组件文件名 );  // 动态装载组件
if(hDLL)
{
        FREG lpfunc = (FREG)::GetProcAddress( hDLL, _T("DllRegisterServer") );
        // 取得注册函数指针
        // 如果是反注册，可以取得"DllUnregisterServer"函数指针
        if ( lpfunc )     lpfunc();// 执行注册。这里为了简单，没有判断返回值
        ::FreeLibrary(hDLL);
}

::SetCurrentDirectory(szWorkPath);  // 切换回原先的进程工作目录
```

上面的示例，在多数情况下可以简化掉切换工作目录的代码部分。但是，如果这个组件在装载的时候，它需要同时加载一些必须依赖的 DLL 时，有可能由于它自身程序的 BUG 导致无法正确定位。咳.....还是让我们自己写的程序，来弥补它的错误吧.....谁让咱们是好人呢 ，谁让咱们的水平比他高呢，谁让咱们在 vckbase 上是个“榜眼”呢.....

四、关于组件调用

总的来说，调用组件程序大概有如下方法：

#include 方法	IDL 编译后，为方便 C/C++程序员的使用，会产生 xxx.h 和 xxx_i.c 文件。我们真幸福，直接#include 后就可以使用了
#import 方法	比较通用的方法，vc 会帮我们产生包装类，让我们的调用更方便
加载类型库包装类方法	如果组件提供了 IDispatch 接口，用这个方法调用组件是最简单的啦。不过还没讲 IDispatch，只能看以后的文章啦
加载 ActiveX 包装类 方法	ActiveX 还没介绍呢，以后再说啦

下载示例程序后，请逐项浏览使用方法：

示例	方法	简要说明
1	#include	完全用最基本的 API 方式调用组件，使大家熟悉调用原理
2	#include	大部分使用 API 方式，使用 CComBSTR 简化对字符串的使用
3	#include	展示智能指针 CComPtr<> 的使用方法
4	#include	展示智能指针 CComPtr<> 和 CComQIPtr<> 混合的使用方法
5	#include	展示智能指针 CComQIPtr<> 的使用方法
6	#include	展示智能指针的释放方法
7	#import	vc 包装的智能指针 IxxxPtr、_bstr_t、_variant_t 的使用方法和异常处理
8	#import	import 后的命名空间的使用方法

示例程序中都写有注释，请读者仔细阅读并同时参考 MSDN 的函数说明。这里，我给大家介绍一下“智能指针”：

对于操作原始的接口指针是比较麻烦的，需要我们自己控制引用记数、API 调用、异常处理。于是 ATL 提供了 2 个智能指针的模板包装类，CComPtr<> 和 CComQIPtr<>，这两个类都在 <atlbase.h> 中声明。CComQIPtr<> 包含了 CComPtr<>的所有功能，因此我们可以完全用 CComQIPtr<> 来使用智能接口指针，唯一要说明的一点就是：CComQIPtr<> 由于使用了运算符的重载功能，它会自动帮我们调用 QueryInterface()函数，因此 CComQIPtr<> 唯一的缺点就是不能定义 IUnknown * 指针。

```
// 智能指针 smart pointer，按照匈牙利命名法，一般以 sp 开头来表示变量类型
CComPtr < IUnknown > spUnk;    // 正确
// 假设 IFun 是一个接口类型
CComPtr < IFun > spFun;        // 正确
CComQIPtr < IFun > spFun;      // 正确
CComQIPtr < IFun, &IID_IFun > spFun;    // 正确
CComQIPtr < IUnknown > spUnk; // 错误！CComQIPtr 不能定义 IUnknown 指针
```

给智能指针赋值的方法：

```
CComQIPtr < IFun > spFun;      // 调用构造函数，还没有赋值，被包装的内部接口指针为 NULL
CComQIPtr < IFun > spFun( pOtherInterface ); // 调用构造函数，内部接口指针赋值为
// 通过 pOtherInterface 这个普通接口指针调用 QueryInterface()得到的 IFun 接口指针
CComQIPtr < IFun > spFun( spOtherInterface ); // 调用构造函数，内部接口指针赋值为
// 通过 spOtherInterface 这个只能接口指针调用 QueryInterface()得到的 IFun 接口指针
CComQIPtr < IFun > spFun ( pUnknown ); // 调用构造函数，由 IUnknown 的 QueryInterface()得到 IFun 接口指针

CComQIPtr < IFun > spFun = pOtherInterface;      //= 运算符重载，含义和上面一样
spFun = spOtherInterface;      // 同上
spFun = pUnknown;     // 同上

pUnknown->QueryInterface( IID_IFun, &sp );// 也可以通过 QueryInterface 赋值

// 智能指针赋值后，可以用条件语句判断是否合法有效
if ( spFun ){}                 // 如果指针有效
if ( NULL != spFun ){}         // 如果指针有效

if ( !spFun ){}                // 如果指针无效
if ( NULL == spFun ){}         // 如果指针无效
```

智能指针调用函数的方法：

```
spFun.CoCreateInstance(...);  // 等价与 API 函数::CoCreateInstance(...)
spFun.QueryInterface(...);    // 等价与 API 函数::QueryInterface()

spFun->Add(...);      // 调用内部接口指针的接口函数

// 调用内部接口指针的 QueryInterface()函数，其实效果和 spFun.QueryInterface(...) 一样
spFun->QueryInterface(...);

spFun.Release();      // 释放内部的接口指针，同时内部指针赋值为 NULL
```

```
spFun->Release();      // 错！！！一定不要这么使用。
// 因为这个调用并不把内部指针清空，那么析构的时候会被再次释放（释放了两次）
```

咳.....不说了，不说了，大家多看书，多看 MSND，多看示例程序吧。 写累了:-(

五、小结

敬请关注《COM 组件设计与应用(八)》------如何增加 ATL 组件中的第二个接口

注 1：编译代理/存根，vc6.0 中稍微麻烦，我们在后面介绍“进程外组件”和“远程组件”的时候再介绍。在 vc.net 2003 下则比较简单，因为代理/存根作为单独的一个工程项目会自动加到我们的解决方案中了。

COM 组件设计与应用（八）
实现多接口

下载源代码

一、前言

从第五回开始到第七回，咱们用 ATL 写了一个简单的 COM 组件，之所以说简单，是因为在组件中，只实现了一个自定义(custom)的接口 IFun。当然如果想偷懒的话，我们可以把 200 个函数都加到这一个接口中， 果真如此的话，恐怕就没有人喜欢使用我们这个组件了。一个组件既然可以提供多个接口，那么我们在设计的时候，就应该按照函数的功能进行分类，把不同功能分类的函数用多个接口表现出来。这样可以有如下的一些好处：

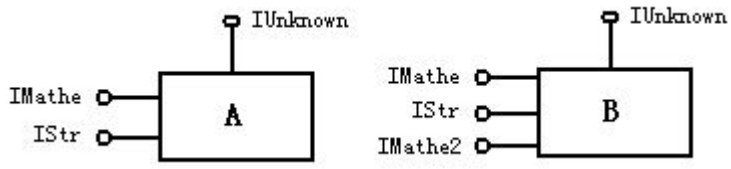
1、一个接口中的函数个数有限、功能集中，使用者容易学习、记忆和调用。一个接口到底提供多少个函数合适那？答案是：如果你是黑猩猩，那么一个接口最多 3 个函数，如果你是人，那么一个接口最好不要超过 7 个函数。(注 1)

2、容易维护。至少你肉眼搜索的时候也方便一些呀。

3、容易升级。当我们给组件增加函数的时候，不要修改已经发表的接口，而是提供一个新的接口来完成功能扩展。(注 2)

本回书着落在------如何实现一个组件，多个接口。

二、接口结构



图一、组件 A 有 2 个自定义接口，组件 B 是 A 的升级

某日，我们设计了组件 A，它有 2 个自定义(custom)接口。IMathe 有函数 Add()完成整数加法，IStr 有函数 Cat()完成字符串连接。忽一日，我们升级组件 A 到 B，欲增加一个函数 Mul() 完成整数的乘法。注意，由于我们已经发表了组件 A，因此我们不能把这个函数安排到老接口 IMathe 中了。解决方法是再定义一个新接口 IMathe2，在新接口中增加 Mul() 函数并依旧保留 Add() 函数。这样，老用户不知道新接口 IMathe2 的存在，他仍然使用旧接口 IMathe；而新用户则可以抛弃 IMathe，直接使用 IMathe2 的新接口功能。看，多平顺的升级方式呀！

三、实现

3-1、首先用 ATL 实现一个自定义(custom)接口 IMathe 的 COM 组件，在接口中完成 Add()整数加法函数。**注意！！！**一定是自定义(custom)的接口（dual 双接口以后再介绍）。如果你不了解这个操作，请重新阅读“第五回”或“第六回”。

3-2、查看 IDL 文件。完成上一个步骤后，打开 IDL 文件，内容如下：(名称及 UUID 会和你程序中的 IDL 有所不同)

```
1 import "oaidl.idl";
2 import "ocidl.idl";
3       [
4               object,
5               uuid(072EA6CA-7D08-4E7E-B2B7-B2FB0B875595),

6               helpstring("IMathe Interface"),
7               pointer_default(unique)
8       ]
9       interface IMathe : IUnknown
10      {
11              [helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2,
[out,retval] long *pnVal);
12      };

13 [
14      uuid(CD7672F7-C0B4-4090-A2F8-234C0062F42C),
15      version(1.0),
16      helpstring("Simple3 1.0 Type Library")
17 ]
18 library SIMPLE3Lib
19 {
20      importlib("stdole32.tlb");
```

```
21        importlib("stdole2.tlb");

22        [
23                uuid(C6F241E2-43F6-4449-A024-B7340553221E),
24                helpstring("Mathe Class")
25        ]
26        coclass Mathe
27        {
28                [default] interface IMathe;
29        };
30 };
```

1-2	引入 IUnknown 和 ATL 已经定义的其它接口描述文件。import 类似与 C 语言中的 #include
3-12	一个接口的完整描述
4	object 表示本块描述的是一个接口。IDL 文件是借用了 PRC 远程数据交换格式的说明方法
5	uuid(.....) 接口的 IID，这个值是 ATL 自动生成的，可以手工修改或用 guidgen.exe 产生(注 3)
6	在某些软件或工具中，能看到这个提示
7	定义接口函数中参数所使用指针的默认属性(注 4)
9	接口叫 IMathe 派生自 IUnknown，于是 IMathe 接口的头三个函数一定就是 QueryInterface,AddRef 和 Release
10-12	接口函数列表
13-30	类型库的完整描述(类型库的概念以后再说)，下面所说明的行，是需要先了解的
18	#import 时候的默认命名空间
23	组件的 CLSID，CoCreateInstance()的第一个参数就是它
27-29	接口列表
28	[default]表示谁提供了 IUnknown 接口

3-3、手工修改 IDL 文件，黑体字部分是手工输入的。完成后保存。

```
import "oaidl.idl";
import "ocidl.idl";
        [
```

```
		object,
		uuid(072EA6CA-7D08-4E7E-B2B7-B2FB0B875595),

		helpstring("IMathe Interface"),
		pointer_default(unique)
	]
	interface IMathe : IUnknown
	{
		[helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2,
[out,retval] long *pnVal);
	};

	[	// 所谓手工输入，其实也是有技巧的：把上面的接口描述（IMathe）复制、
粘贴下来，然后再改更方便哈
		object,
		uuid(072EA6CB-7D08-4E7E-B2B7-B2FB0B875595), // 手工或用工具产生
的 IID

		helpstring("IStr Interface"),
		pointer_default(unique)
	]
	interface IStr : IUnknown
	{
		// 目前还没有任何接口函数
	};

[
	uuid(CD7672F7-C0B4-4090-A2F8-234C0062F42C),
	version(1.0),
	helpstring("Simple3 1.0 Type Library")
]
library SIMPLE3Lib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(C6F241E2-43F6-4449-A024-B7340553221E),
```

```
                helpstring("Mathe Class")
        ]
        coclass Mathe
        {
                [default] interface IMathe;
                interface IStr;   // 别忘了呦，这里还有一个那
        };
};
```

3-4、打开头文件(Mathe.h)，手工增加类的派生关系和接口入口表 ，然后保存。

```
class ATL_NO_VTABLE CMathe :
        public CComObjectRootEx <CComSingleThreadModel>,
        public CComCoClass <CMathe, &CLSID_Mathe>,
        public IMathe,     // 别忘了，这里加一个逗号
        public IStr        // 增加一个基类
{
public:
        CMathe()
        {
        }

DECLARE_REGISTRY_RESOURCEID(IDR_MATHE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CMathe)     // 接口入口表。这里填写的接口，才能被 QueryInterface()找到
        COM_INTERFACE_ENTRY(IMathe)
        COM_INTERFACE_ENTRY(IStr)
END_COM_MAP()
```

3-5、好了，一切就绪。接下来，就可以在 IStr 接口中增加函数了。示例程序中增加一个字符串连接功能的函数：

HRESULT Cat([in] BSTR s1, [in] BSTR s2, [out,retval] BSTR *psVal); 如果你不知道如何做，请重新阅读前三回的内容。

四、接口升级

我们这个组件已经发行了，但忽然一天我们需要在 IMathe 接口上再增加一个函数.....不行！绝对不能在 IMathe 上直接修改！怎么办？解决方法是------再增加一个接口，我们就叫 IMathe2 吧，如果以后还要增加函数，那么我们再增加一个接口叫 IMathe3.....子子孙孙，无穷尽也。

4-1、修改 IDL 文件，其实如果你理解了上面一小节的内容，再增加一个接口是很简单的事情了。

```
import "oaidl.idl";
import "ocidl.idl";
        [
                object,
                uuid(072EA6CA-7D08-4E7E-B2B7-B2FB0B875595),

                helpstring("IMathe Interface"),
                pointer_default(unique)
        ]
        interface IMathe : IUnknown
        {
                [helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2,
[out,retval] long *pnVal);
        };

        [
                object,
                uuid(072EA6CB-7D08-4E7E-B2B7-B2FB0B875595),

                helpstring("IStr Interface"),
                pointer_default(unique)
        ]
        interface IStr : IUnknown
        {
                [helpstring("method Cat")] HRESULT Cat([in] BSTR s1, [in] BSTR s2,
[out,retval] BSTR *psVal);
        };

        [
                object,
                uuid(072EA6CC-7D08-4E7E-B2B7-B2FB0B875595),

                helpstring("IMathe2 Interface"),
                pointer_default(unique)
        ]
        interface IMathe2 : IUnknown
```

```
        {       // 下面这个 Add（）函数，只有 IDL 中的声明，而不用增加任何程序代码，
因为这个函数早在 IMathe 中就已经实现了
                [helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2,
[out,retval] long *pnVal);
        };

[
        uuid(CD7672F7-C0B4-4090-A2F8-234C0062F42C),
        version(1.0),
        helpstring("Simple3 1.0 Type Library")
]
library SIMPLE3Lib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(C6F241E2-43F6-4449-A024-B7340553221E),
                helpstring("Mathe Class")
        ]
        coclass Mathe
        {
                [default] interface IMathe;
                interface IStr;
                interface IMathe2;// 别忘了，这里还有一行呢！
        };
};
```

4-2、打开头文件，增加派生关系和接口入口表

```
class ATL_NO_VTABLE CMathe :
        public CComObjectRootEx <CComSingleThreadModel>,
        public CComCoClass <CMathe, &CLSID_Mathe>,
        public IMathe,
        public IStr,        // 这里增加一个逗号
        public IMathe2
{
public:
        CMathe()
        {
```

```
        }

DECLARE_REGISTRY_RESOURCEID(IDR_MATHE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CMathe)
        COM_INTERFACE_ENTRY(IMathe)
        COM_INTERFACE_ENTRY(IStr)
        COM_INTERFACE_ENTRY(IMathe2)
END_COM_MAP()
```

4-3、示例程序中，增加了一个整数乘法函数：

HRESULT Mul([in] long n1, [in] long n2, [out,retval] long *pnVal); 如果你不知道如何做，那就别学了:-(都讲好几遍了，怎么还不掌握呢？知道狗熊是怎么死的吗？(注 5)

4-4、因为我们的组件升级了，于是也应该修改版本号了（这不是必须的，但最好修改一下）。打开注册表文件(.rgs) 把有关 ProgID 的版本 "Mathe.1" 修改为"Mathe.2"。另外如果你愿意，把 IDL 文件中的 version 和提示文字一并修改一下。这里就不再粘贴文件内容了，因为很简单，大家下载示例程序(注 6)后，自己看吧。

五、小结

为祖国的软件事业而奋斗！

下回书介绍“自动化”--- IDispatch 接口，好玩的很！谢谢关注:-)

注 1： 黑猩猩的瞬时记忆量是 3，人类的瞬时记忆量是 7。科学家做过实验，当着面，把一块糖扣在 3 个碗的其中之一，黑猩猩能立刻准确找到，但如果超过 3 个碗，猩猩就晕了.....如果给你看一串数字，然后立刻让你说出来，一般的人只会记得其中的 7 个。

注 2：组件一经发表，就不要修改已有接口。这样软件的升级才能做到“鲁棒”性。

注 3：guidgen.exe 工具，在《COM 组件设计与应用(二)》中已经介绍。

注 4：组件函数对内存指针的处理，以后有专门的章回讨论。

注 5：笨死的！

注 6：示例程序有两部分，分别是 vc6.0 版本和 vc.net 2003 版本。

COM 组件设计与应用（九）
IDispatch 接口 for vc6.0

下载源代码

一、前言

终于写到了第九回，我也一直期盼着写这回的内容耶，为啥呢？因为自动化(automation)

是非常常用、非常有用、非常精彩的一个 COM 功能。由于 WORD、EXCEL 等 OFFICE 软件提供了“宏”的功能，就连我们使用的 VC 开发环境也提供了“宏”功能，更由于 HTML、ASP、JSP 等都要依靠脚本(Script)的支持，更体现出了自动化接口的重要性。

如果你使用 vc6.0 的开发环境，请继续阅读。

如果你使用 vc.net 2003，请阅读下一回。

二、IDispatch 接口

如果是编译型语言，那么我们可以让编译器在编译的时候装载类型库，也就是装载接口的描述。在第七回文章当中，我们分别使用了 #include 方法和 #import 方法来实现的。装载了类型库后，编译器就知道应该如何编译接口函数的调用了---这叫“前绑定”。但是，如果想在脚本语言中使用组件，问题就大了，因为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，怎么办？自动化接口就为此诞生了---“后绑定”。

自动化组件，其实就是实现了 IDispatch 接口的组件。IDispatch 接口有 4 个函数，解释语言的执行器就通过这仅有的 4 个函数来执行组件所提供的功能。IDispatch 接口用 IDL 形式说明如下：(注 1)

```
[
    object,
    uuid(00020400-0000-0000-C000-000000000046),        // IDispatch 接口的 IID = IID_IDispatch
    pointer_default(unique)
]

interface IDispatch : IUnknown
{
    typedef [unique] IDispatch * LPDISPATCH; // 转定义 IDispatch * 为 LPDISPATCH

    HRESULT GetTypeInfoCount([out] UINT * pctinfo);    // 有关类型库的这两个函数,咱们以后再说
    HRESULT GetTypeInfo([in] UINT iTInfo,[in] LCID lcid,[out] ITypeInfo ** ppTInfo);

    HRESULT GetIDsOfNames( // 根据函数名字，取得函数序号(DISPID)
                [in] REFIID riid,
                [in, size_is(cNames)] LPOLESTR * rgszNames,
                [in] UINT cNames,
                [in] LCID lcid,
                [out, size_is(cNames)] DISPID * rgDispId
            );
```

```
    [local]                 // 本地版函数
    HRESULT Invoke(         // 根据函数序号，解释执行函数功能
                [in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS * pDispParams,
                [out] VARIANT * pVarResult,
                [out] EXCEPINFO * pExcepInfo,
                [out] UINT * puArgErr
            );

    [call_as(Invoke)]       // 远程版函数
    HRESULT RemoteInvoke(
                [in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] DWORD dwFlags,
                [in] DISPPARAMS * pDispParams,
                [out] VARIANT * pVarResult,
                [out] EXCEPINFO * pExcepInfo,
                [out] UINT * pArgErr,
                [in] UINT cVarRef,
                [in, size_is(cVarRef)] UINT * rgVarRefIdx,
                [in, out, size_is(cVarRef)] VARIANTARG * rgVarRef
            );
}
```

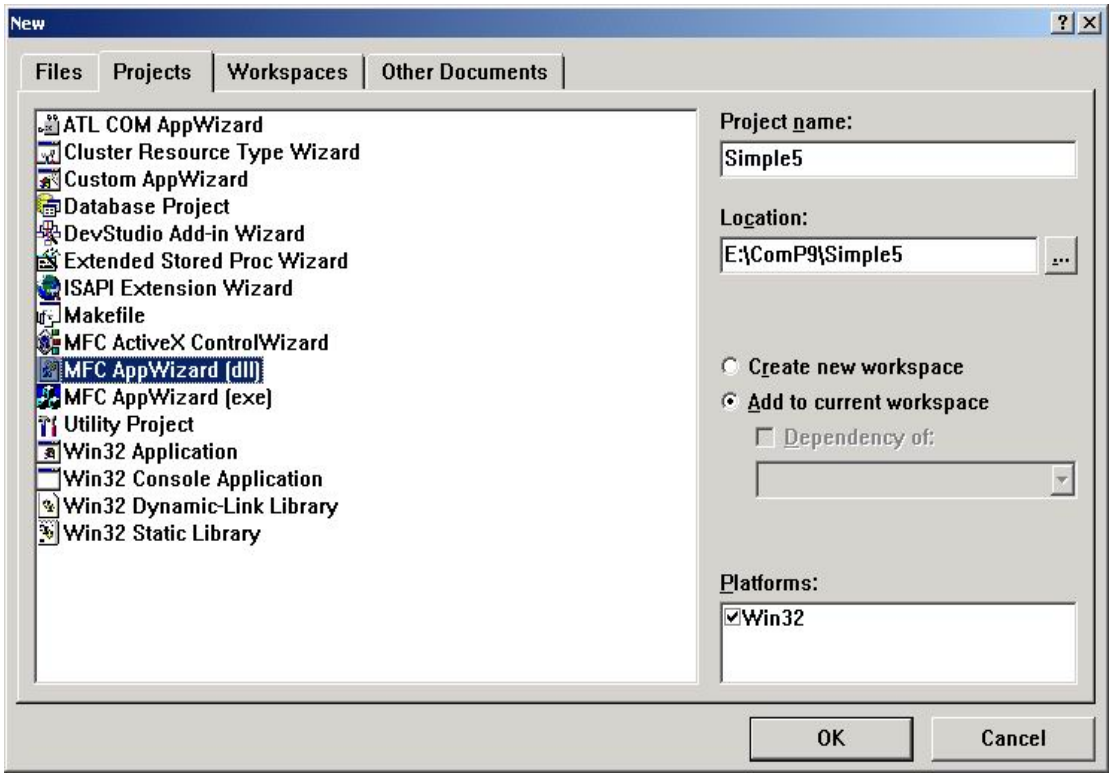
以上 IDispatch 接口函数的讲解，我们留到后回中进行介绍。如何在组件程序中实现这些函数那？还好，还好，就象 IUnknown 一样，MFC 和 ATL 都帮我们已经完成了。本回我们着重介绍组件的编写，下回则介绍组件的调用方法。

三、用 MFC 实现自动化组件

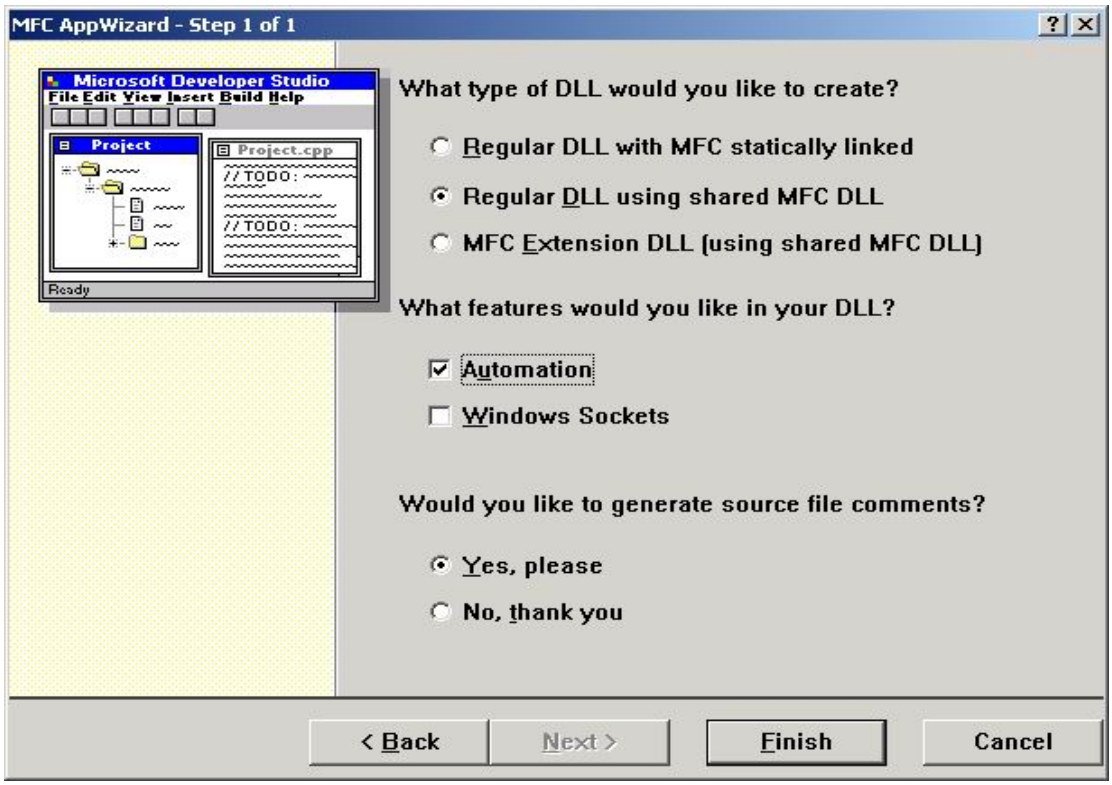
我写的这整个系列文章---《COM 组件设计与应用》，多是用 ATL 写组件程序，但由于自动化非常有用，在后续的文章中，还要给大家介绍组件的“事件”功能，还要介绍如何在 MFC 的程序中象 WORD 一样支持“宏”的功能。这些都要用到 MFC，所以就给读者唠一唠啦:-)

3-1：建立一个工作区(Workspace)

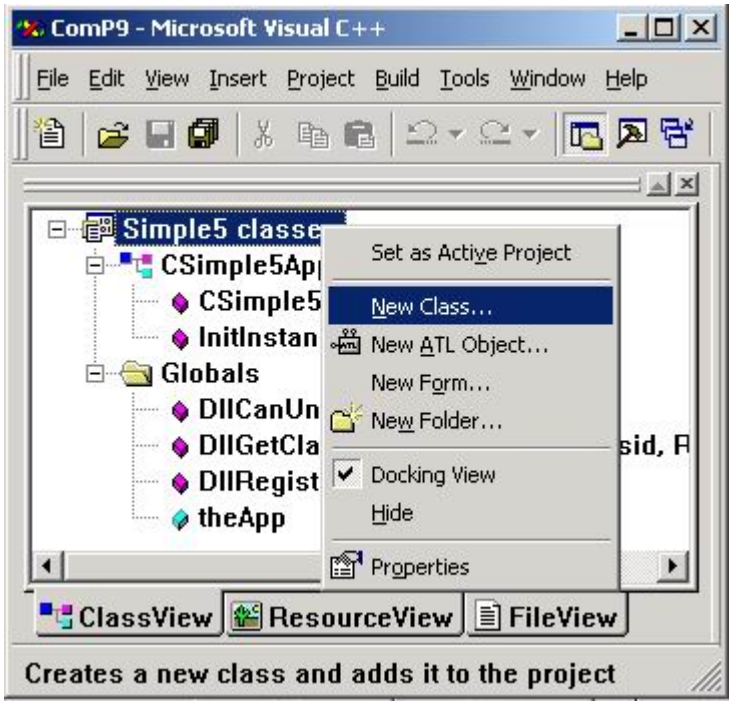
3-2：建立一个 MFC DLL 工程(Project)，工程名称为“Simple5”



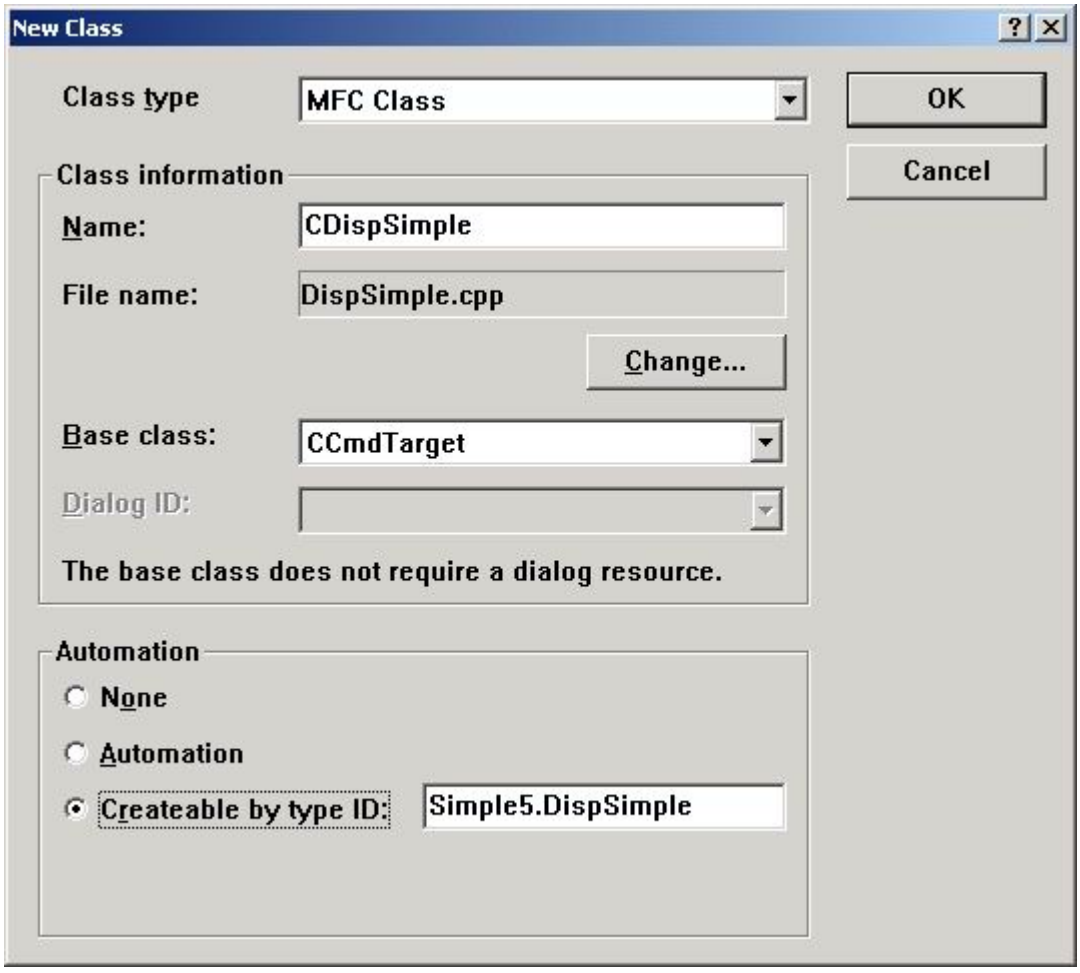
3-3：一定要选择 automation，切记！切记！



3-4：建立新类



3-5：在新建类中支持 automation



Class information - Name 你随便写个类名子啦

Class information - Base class 一定要从 CComTarget 派生呀，只有它才提供了

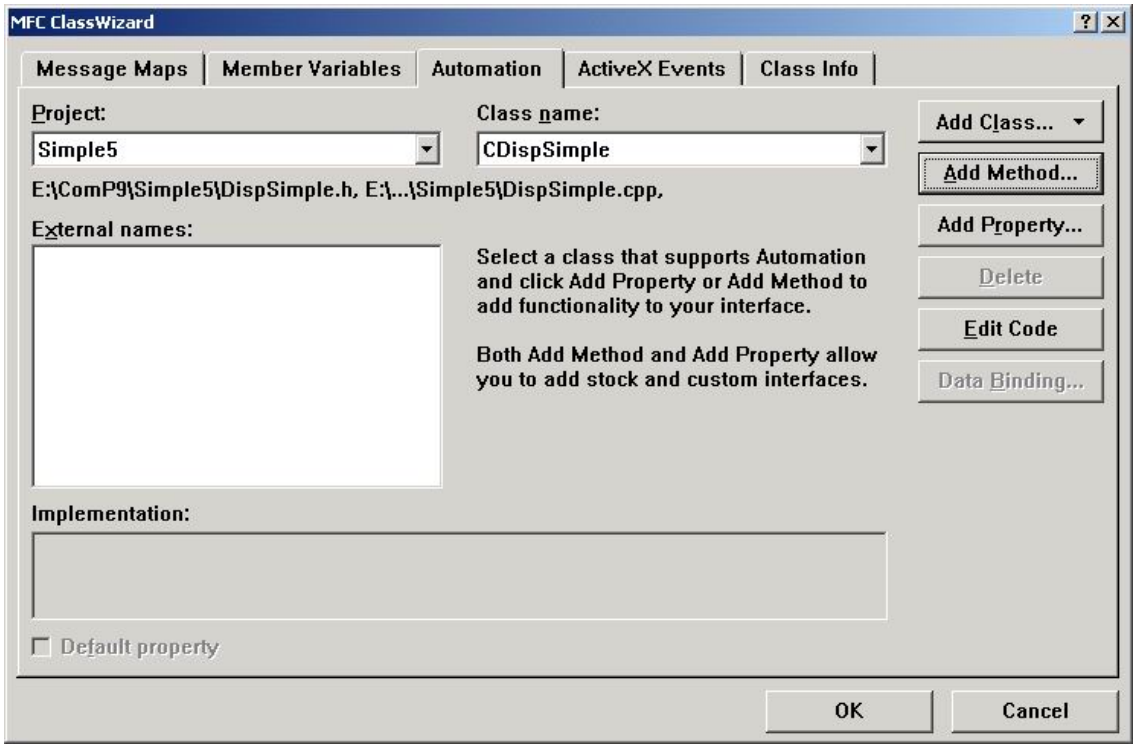
IDispatch 的支持

Automation - None 表示不支持自动化，你要选择了它，那就白干啦

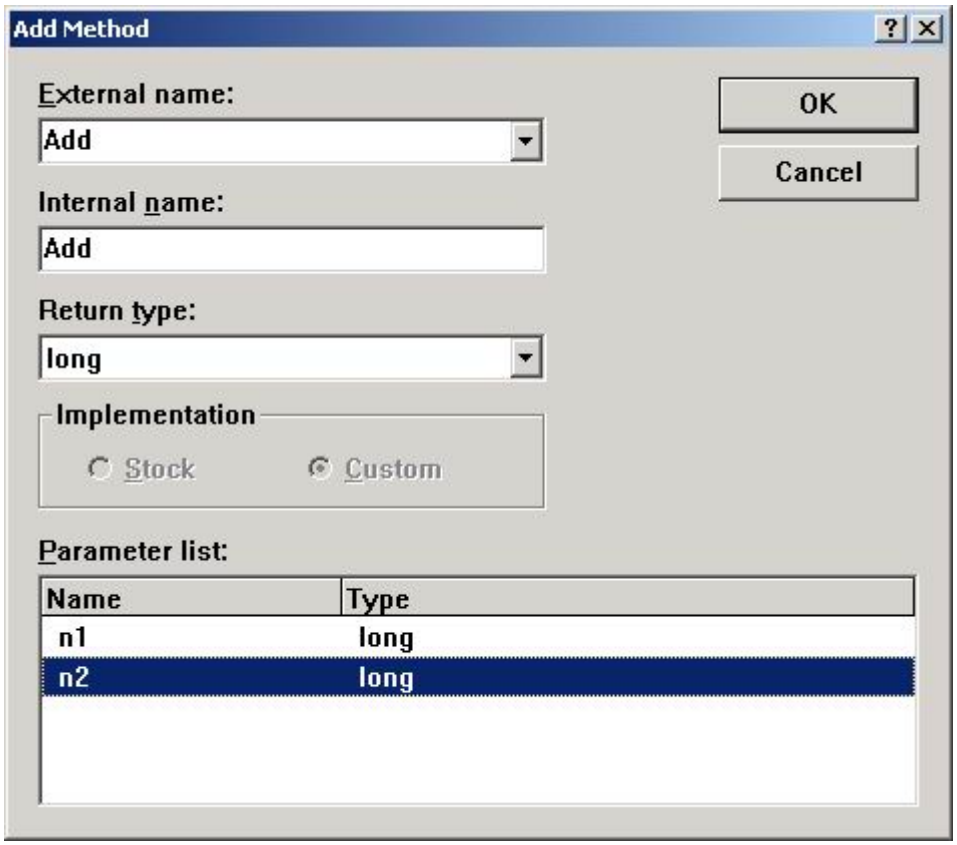
Automation - Automation 支持自动化，但不能被直接实例化。后面在讲解多个 IDispatch 的时候就用到它了，现在先不要着急。

Automation - Createable by type ID 一定要选择这个项目，这样我们在后面的调用中，VB 就能够 CreateObject(),VC 就能够 CreateDispatch()对组件对象实例化了。注意一点，这个 ID 其实就是组件的 ProgID 啦。

3-6：启动 ClassWizard，选择 Automation 卡片，准备建立函数



3-7：添加函数。我们要写一个整数加法函数 Add()。



3-8：再增加一个转换字符串大小写的函数 Upper()。函数返回值是 BSTR，这个没有什么疑问，但参数类型怎么居然是 LPCTSTR？在 COM 中，字符串不是应该使用 BSTR 吗？是的，是应该使用 BSTR，但由于我们是用 MFC 写自动化组件，它帮我们进行 BSTR 和 LPCTSTR 之间的转换了。

3-9：好了，下面开始输入程序代码：

```
long CDispSimple::Add(long n1, long n2)
{
        return n1 + n2;
}

BSTR CDispSimple::Upper(LPCTSTR str)
{
        CString strResult(str);
        strResult.MakeUpper();

        return strResult.AllocSysString();
}
```

3-10：编译注册

如果上面的操作由于疏忽而发生了错误，那么你可以手工进行改正。

其一、步骤<3-6>的对话窗中有“Delete”操作；

其二、你可以打开 ODL 文件(注 2)进行修改，修改时要特别小心函数的声明中，有一个[id(n)]的函数序号，可不要乱了；

其三、同步修改 H/CPP 中的函数声明和函数体；

其四、在 CPP 文件中，根据情况也要修改 BEGIN_DISPATCH_MAP/END_DISPATCH_MAP()

函数影射宏。

正确编译后，MFC 不象 ATL 那样会自动注册。你需要手工执行 regsvr32.exe 进行注册，或者执行菜单“Tools\Register control”

四、用 ATL 实现双接口组件(操作方法和步骤，请参考《COM 组件设计与应用(五)》)

4-1：建立一个 ATL 工程(Project)，工程名称为“Simple6”

4-2：按默认进行。选择 DLL 类型、不合并代理和存根代码、不支持 MFC、不支持 MTS

4-3：New Atl Object... 选择 Simple Object

4-4：输入名称和属性，属性按默认进行，也就是 dual(双接口)方式(注 3)

4-5：增加函数。在 ClassView 卡片中，选择接口、鼠标右键菜单、Add Method...

Add([in] VARIANT v1, [in] VARIANT v2, [out, retval] VARIANT * pVal);

Upper([in] BSTR str, [out,retval] BSTR * pVal);

关于 Add()函数，你依然可以使用 Add([in] long n1, [in] long n2, [out,retval] long * pVal) 方式。但这次我们没有使用 long ，而是使用了 VARIANT 做参数和返回值。这里我先卖个关子，往下看，就知道使用 VARIANT 的精彩之处了。

4-6：完成代码

```
STDMETHODIMP CDispSimple::Add(VARIANT v1, VARIANT v2, VARIANT *pVal)
{
	::VariantInit( pVal );	// 永远初始化返回值是个好习惯

	CComVariant v_1( v1 );
	CComVariant v_2( v2 );

	if((v1.vt & VT_I4) && (v2.vt & VT_I4) )	// 如果都是整数类型
	{	// 这里比较没有使用 == ，而使用了运算符 & ，你知道这是为什么吗？
		v_1.ChangeType( VT_I4 );	// 转换为整数
		v_2.ChangeType( VT_I4 );	// 转换为整数

		pVal->vt = VT_I4;
		pVal->lVal = v_1.lVal + v_2.lVal;	// 加法
	}
	else
	{
		v_1.ChangeType( VT_BSTR ); // 转换为字符串
		v_2.ChangeType( VT_BSTR ); // 转换为字符串

		CComBSTR bstr( v_1.bstrVal );
		bstr.AppendBSTR( v_2.bstrVal );	// 字符串连接

		pVal->vt = VT_BSTR;
		pVal->bstrVal = bstr.Detach();
	}
	return S_OK;
}

STDMETHODIMP CDispSimple::Upper(BSTR str, BSTR *pVal)
{
	*pVal = NULL;	// 永远初始化返回值是个好习惯

	CComBSTR s(str);
	s.ToUpper();	// 转换为大写

	*pVal = s.Copy();
```

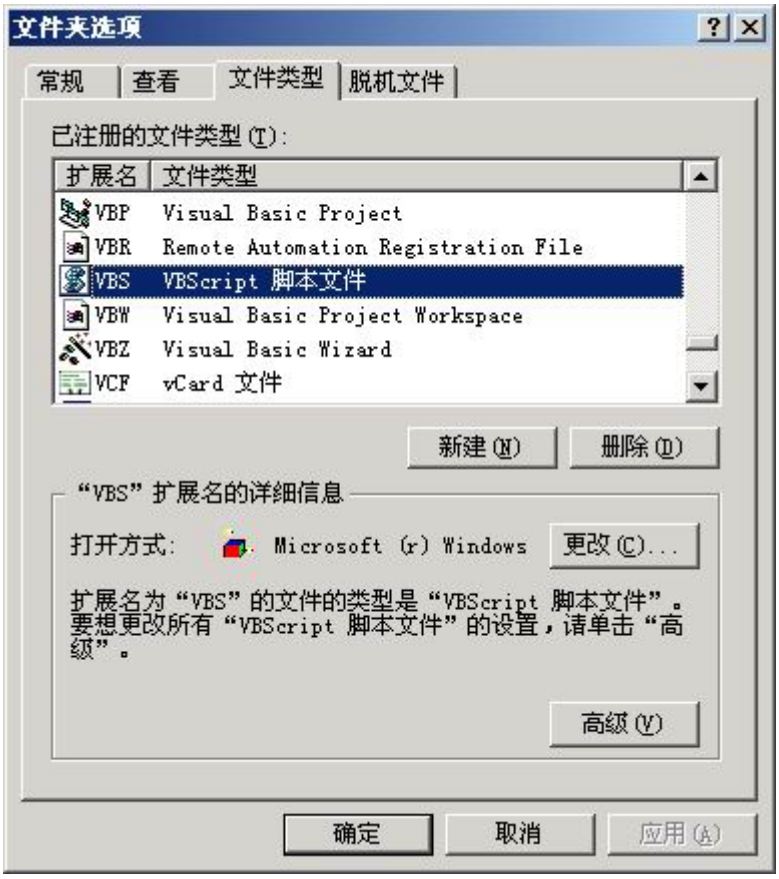
```
        return S_OK;
}
```

刚才卖的关子，现在开始揭密了.....加法函数 Add()不使用 long 类型，而使用 VARIANT 的好处是：函数内部动态判断参数类型，如果是整数则进行整数加法，如果是字符串，则进行字符串加法（字符串加法就是字符串连接哈）。也就是说，如果参数是 VARIANT，那么我们就可以实现函数的可变参数类型呀。怪怪个呢，真爽！

五、脚本中调用举例

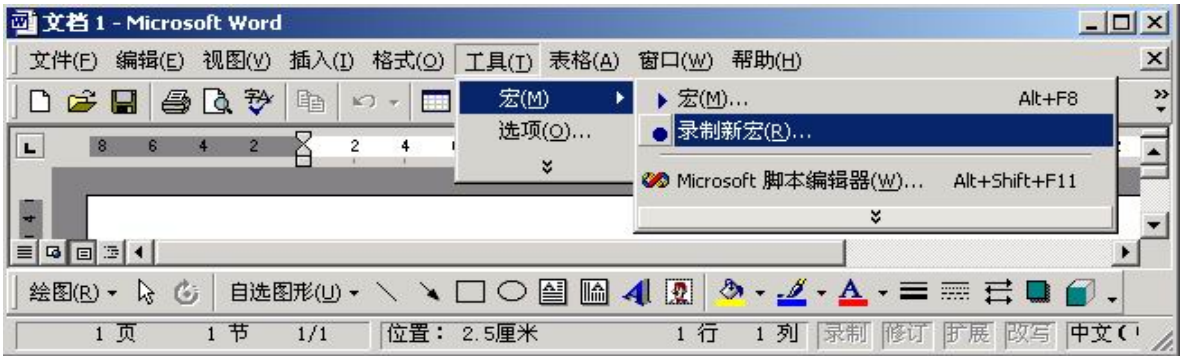
打开“记事本”程序，输入脚本程序，保存为 xxx.vbs 文件。然后在资源管理器里就可以双击运行啦。

如果你有能力，也可以用 JScript 书写上面的程序，然后保存为 xxx.js 文件，同样也可以在资源管理器里运行。另外需要说明的一点是，脚本程序文件的图标(win 2000 下)是  ，如果你不是这样的（有一个软件叫“XX 解霸”。写这款软件的人水平太低，它居然使用 .vbs 的扩展名文件作为它的数据流文件，破坏了系统默认的文件类型影射模式，咳.....），那么需要重新设置，方法是：

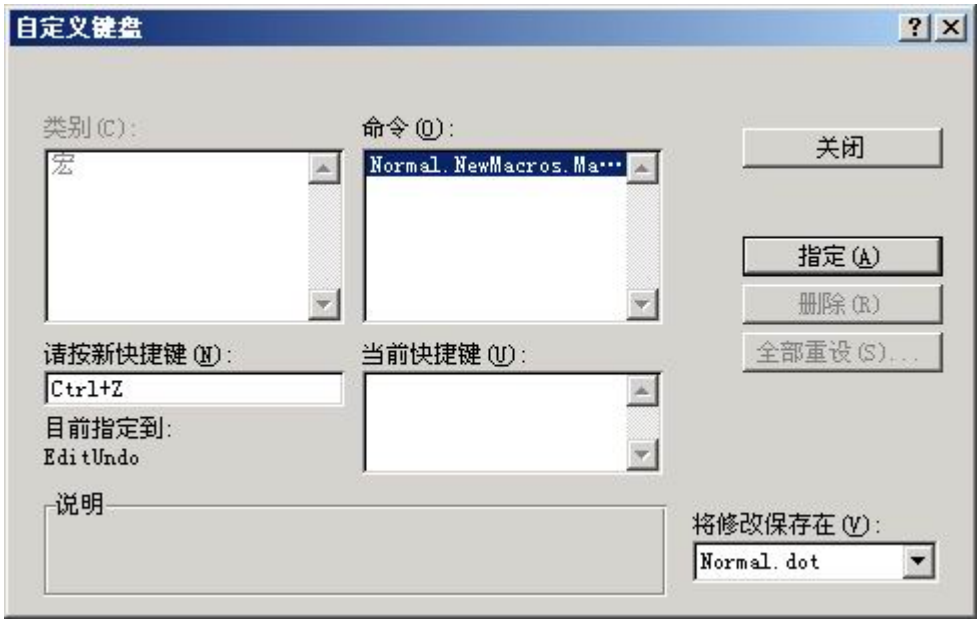


六、WORD 中使用举例

6-1：录制一段宏程序

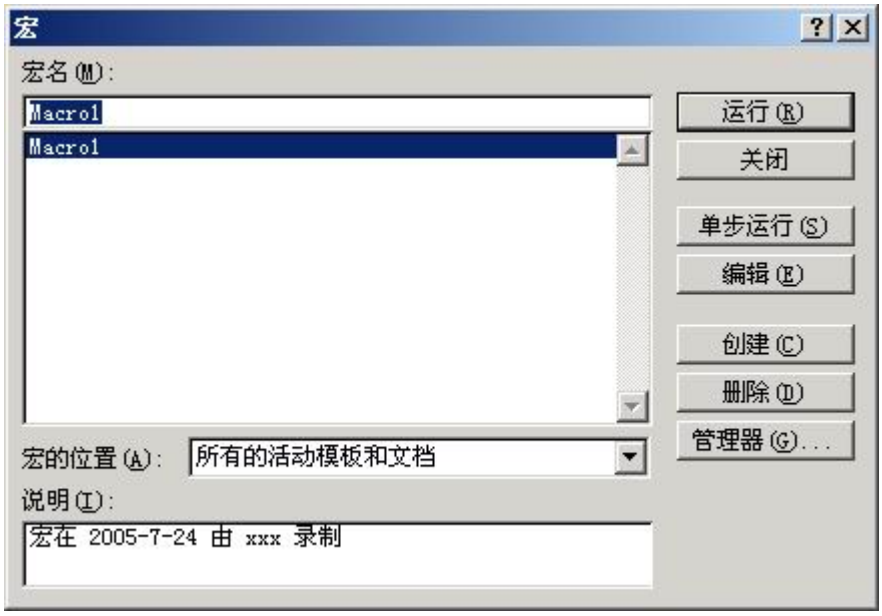


6-2：选择“键盘”，当然你也可以把这个“宏”程序放到“工具栏”上去。这里我们随便指定一个快捷键，比如 Ctrl+Z

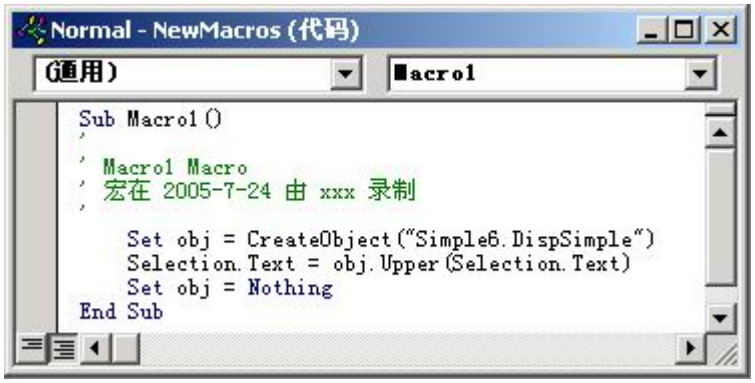


6-3：开始录制了，下面你随便输入点什么东东。然后点“停止”

6-4：接下来，我们执行菜单，选择这个刚刚录制的宏，然后编辑它



6-5：点“编辑”按钮，输入下面的程序：



不做解释了，你如果会一点点 VB ，就能看懂这个东东哈。然后保存关闭 VBA 的编辑器(注4)。

6-6：执行啦，执行啦，看看有什么效果呀.....

然后按快捷键 Ctrl+Z

你已经扩展了 MS WORD 的功能啦，嘿啦啦啦啦，嘿啦啦啦，天空出彩霞呀.....我们只是举了一个简单的例子，其实这个例子并没有什么实际应用的意义，因为人家 WORD 本身就有大小写转换功能。但通过这个小例子，你可以体会出自动化组件的功能了，有够厉害吧？！

七、小结

没小结！嘿嘿.....上当喽:-)

注 1：以后我们描述接口函数，都采用 IDL 的形式了。
注 2：ODL 文件和 IDL 类似，是 MFC 专门为自动化而描述的接口文件
注 3：双接口，是支持 IDispatch 接口的一种特殊接口方式，后面马上就要讲啦
注 4：VBA 是专门开发 Office 的一种语言---Visual Basic for Application

COM 组件设计与应用（十）
IDispatch 接口 for vc.net

下载源代码

一、前言

终于写到了第十回，我也一直期盼着写这回的内容耶，为啥呢？因为自动化(automation)是非常常用、非常有用、非常精彩的一个 COM 功能。由于 WORD、EXCEL 等 OFFICE 软件提供了“宏”的功能，就连我们使用的 VC 开发环境也提供了“宏”功能，更由于 HTML、ASP、JSP 等都要依靠脚本(Script)的支持，更体现出了自动化接口的重要性。

如果你使用 vc6.0 的开发环境，请阅读前一回。

如果你使用 vc.net 2003，请继续.....

二、IDispatch 接口

如果是编译型语言，那么我们可以让编译器在编译的时候装载类型库，也就是装载接口

的描述。在第七回文章当中，我们分别使用了 #include 方法和 #import 方法来实现的。装载了类型库后，编译器就知道应该如何编译接口函数的调用了---这叫“前绑定”。但是，如果想在脚本语言中使用组件，问题就大了，因为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，怎么办？自动化接口就为此诞生了---“后绑定”。

自动化组件，其实就是实现了 IDispatch 接口的组件。IDispatch 接口有 4 个函数，解释语言的执行器就通过这仅有的 4 个函数来执行组件所提供的功能。IDispatch 接口用 IDL 形式说明如下：(注 1)

```
[
    object,
    uuid(00020400-0000-0000-C000-000000000046),        // IDispatch 接口的 IID =
IID_IDispatch
    pointer_default(unique)
]

interface IDispatch : IUnknown
{
    typedef [unique] IDispatch * LPDISPATCH; // 转定义 IDispatch * 为 LPDISPATCH

    HRESULT GetTypeInfoCount([out] UINT * pctinfo);    // 有关类型库的这两个函数,咱
们以后再说
    HRESULT GetTypeInfo([in] UINT iTInfo,[in] LCID lcid,[out] ITypeInfo ** ppTInfo);

    HRESULT GetIDsOfNames( // 根据函数名字，取得函数序号(DISPID)
                [in] REFIID riid,
                [in, size_is(cNames)] LPOLESTR * rgszNames,
                [in] UINT cNames,
                [in] LCID lcid,
                [out, size_is(cNames)] DISPID * rgDispId
            );

    [local]                 // 本地版函数
    HRESULT Invoke(         // 根据函数序号，解释执行函数功能
                [in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS * pDispParams,
```

```
            [out] VARIANT * pVarResult,
            [out] EXCEPINFO * pExcepInfo,
            [out] UINT * puArgErr
        );

    [call_as(Invoke)]       // 远程版函数
    HRESULT RemoteInvoke(
            [in] DISPID dispIdMember,
            [in] REFIID riid,
            [in] LCID lcid,
            [in] DWORD dwFlags,
            [in] DISPPARAMS * pDispParams,
            [out] VARIANT * pVarResult,
            [out] EXCEPINFO * pExcepInfo,
            [out] UINT * pArgErr,
            [in] UINT cVarRef,
            [in, size_is(cVarRef)] UINT * rgVarRefIdx,
            [in, out, size_is(cVarRef)] VARIANTARG * rgVarRef
        );
}
```

以上 IDispatch 接口函数的讲解，我们留到后回中进行介绍。如何在组件程序中实现这些函数那？还好，还好，就象 IUnknown 一样，MFC 和 ATL 都帮我们已经完成了。本回我们着重介绍组件的编写，下回则介绍组件的调用方法。

三、用 MFC 实现自动化组件

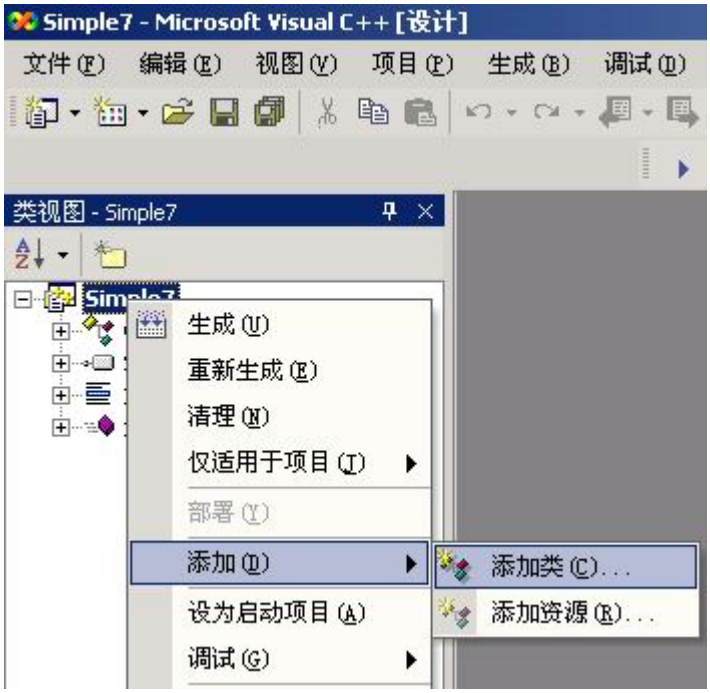
我写的这整个系列文章---《COM 组件设计与应用》，多是用 ATL 写组件程序，但由于自动化非常有用，在后续的文章中，还要给大家介绍组件的“事件”功能，还要介绍如何在 MFC 的程序中象 WORD 一样支持“宏”的功能。这些都要用到 MFC，所以就给读者唠一唠啦:-)

3-1：建立一个解决方案

3-2：建立一个 MFC DLL 项目，项目名称为“Simple7”

3-3：一定要选择附加功能中的“自动化”，切记！切记！

3-4：添加新类



3-5：在新建类中支持自动化

类名 你随便写个类名子啦

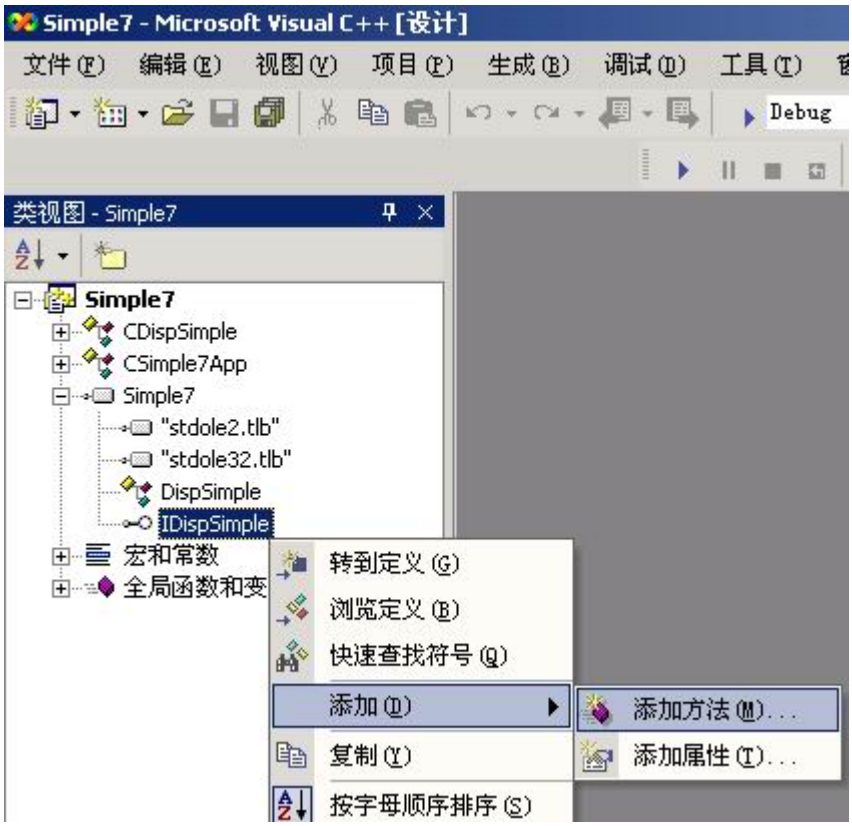
基类 一定要从 CComTarget 派生呀，只有它才提供了 IDispatch 的支持

自动化 - 无 表示不支持自动化，你要选择了它，那就白干啦

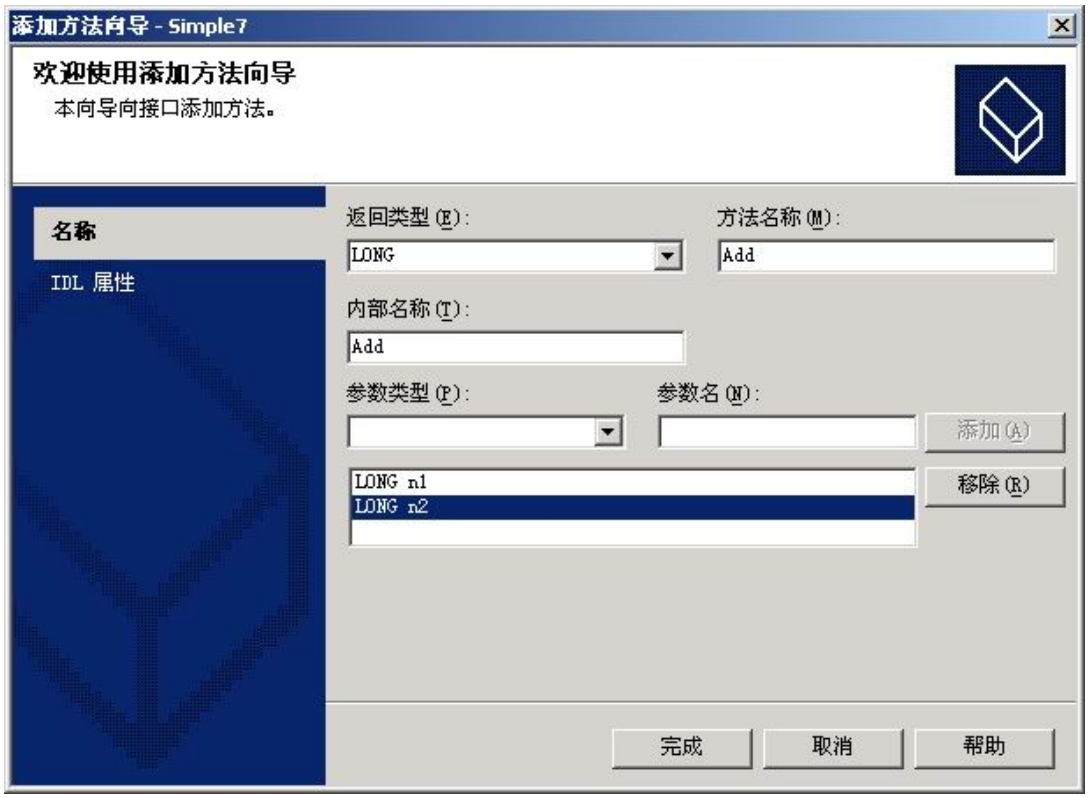
自动化 - 自动化 支持自动化，但不能被直接实例化。后面在讲解多个 IDispatch 的时候就用到它了，现在先不要着急。

自动化 - 可按类型 ID 创建 一定要选择这个项目，这样我们在后面的调用中，VB 就能够 CreateObject(),VC 就能够 CreateDispatch()对组件对象实例化了。注意一点，这个 ID 其实就是组件的 ProgID 啦。

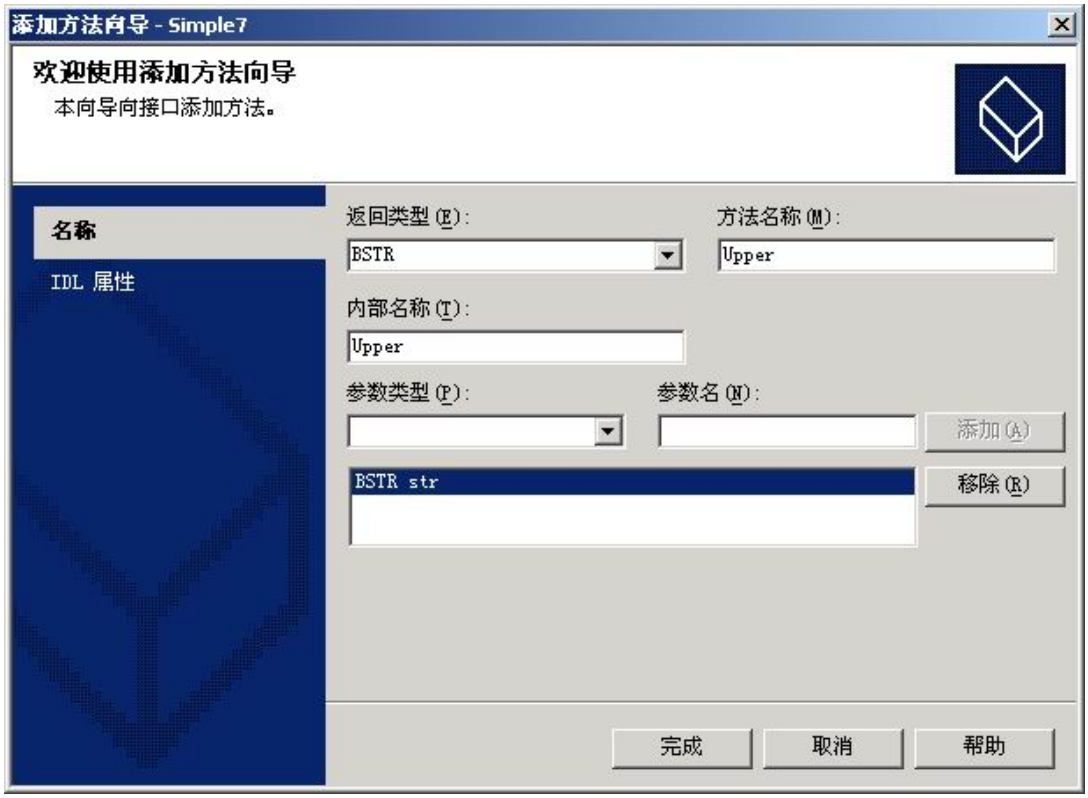
3-6：选择接口，添加函数



3-7：添加函数。我们要写一个整数加法函数 Add()。



3-8：再增加一个转换字符串大小写的函数 Upper()。



3-9：好了，下面开始输入程序代码：

```
LONG CDispSimple::Add(LONG n1, LONG n2)
{
        AFX_MANAGE_STATE(AfxGetStaticModuleState());

        return n1 + n2;
}

BSTR CDispSimple::Upper(LPCTSTR str)
{
        AFX_MANAGE_STATE(AfxGetStaticModuleState());

        CString strResult(str);
        strResult.MakeUpper();

        return strResult.AllocSysString();
}
```

3-10：编译注册

如果上面的操作由于疏忽而发生了错误，那么你可以手工进行改正。

其一、你可以打开 IDL 文件进行修改，修改时要特别小心函数的声明中，有一个[id(n)] 的函数序号，可不要乱了；

其二、同步修改 H/CPP 中的函数声明和函数体；

其三、在 CPP 文件中，根据情况也要修改 BEGIN_DISPATCH_MAP/END_DISPATCH_MAP() 函数影射宏。

正确编译后，vc.net 2003 比 vc6.0 要聪明多了，它会自动注册组件。如果复制到其它计算机上，你也需要手工执行 regsvr32.exe 进行注册。

四、用 ATL 实现双接口组件(操作方法和步骤，请参考《COM 组件设计与应用(六)》)

4-1：建立一个 ATL 项目，项目名称为“Simple8”

4-2：选择 DLL 类型、非属性方式、不要选择任何附加选项

4-3：添加新类，选择 ATL 的简单对象

4-4：输入简称和选项，选项按默认进行，也就是双重接口方式(注 2)

4-5：增加函数。选择接口、鼠标右键菜单、添加方法...

```
Add([in] VARIANT v1, [in] VARIANT v2, [out, retval] VARIANT * pVal);
Upper([in] BSTR str, [out,retval] BSTR * pVal);
```

关于 Add()函数，你依然可以使用 Add([in] long n1, [in] long n2, [out,retval] long * pVal) 方式。但这次我们没有使用 long ，而是使用了 VARIANT 做参数和返回值。这里我先卖个关子，往下看，就知道使用 VARIANT 的精彩之处了。

4-6：完成代码

```
STDMETHODIMP CDispSimple::Add(VARIANT v1, VARIANT v2, VARIANT *pVal)
{
	::VariantInit( pVal );	// 永远初始化返回值是个好习惯

	CComVariant v_1( v1 );
	CComVariant v_2( v2 );

	if((v1.vt & VT_I4) && (v2.vt & VT_I4) )	// 如果都是整数类型
	{	// 这里比较没有使用 == ，而使用了运算符 & ，你知道这是为什么吗？
		v_1.ChangeType( VT_I4 );	// 转换为整数
		v_2.ChangeType( VT_I4 );	// 转换为整数

		pVal->vt = VT_I4;
		pVal->lVal = v_1.lVal + v_2.lVal;	// 加法
	}
	else
	{
		v_1.ChangeType( VT_BSTR ); // 转换为字符串
		v_2.ChangeType( VT_BSTR ); // 转换为字符串

		CComBSTR bstr( v_1.bstrVal );
		bstr.AppendBSTR( v_2.bstrVal );	// 字符串连接

		pVal->vt = VT_BSTR;
		pVal->bstrVal = bstr.Detach();
	}
	return S_OK;
}

STDMETHODIMP CDispSimple::Upper(BSTR str, BSTR *pVal)
{
	*pVal = NULL;	// 永远初始化返回值是个好习惯
```

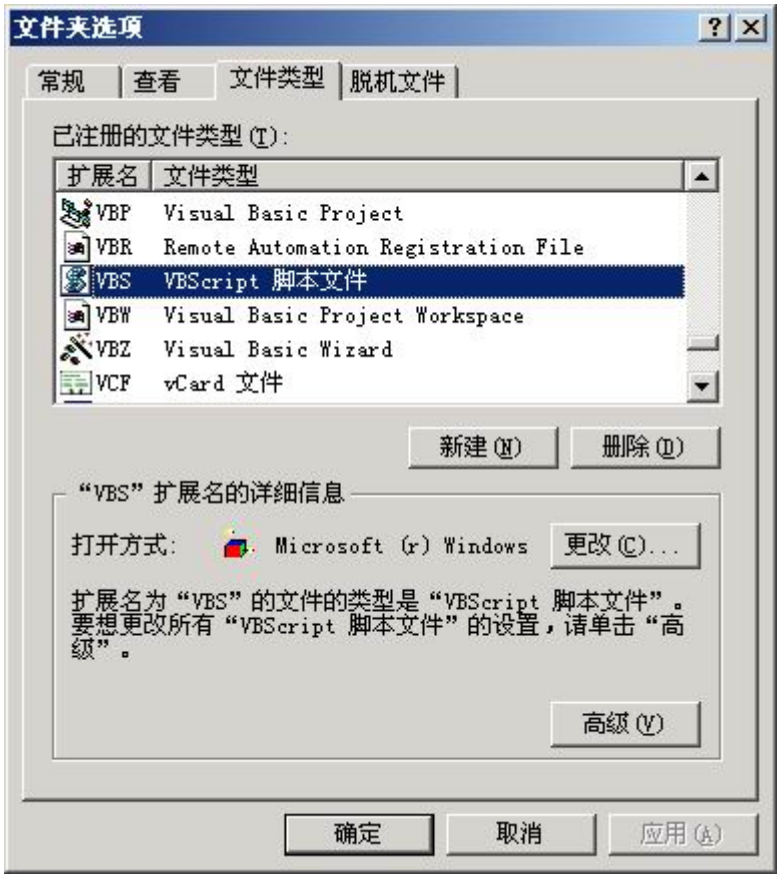
```
        CComBSTR s(str);
        s.ToUpper();        // 转换为大写

        *pVal = s.Copy();

        return S_OK;
}
```

刚才卖的关子，现在开始揭密了.....加法函数 Add()不使用 long 类型，而使用 VARIANT 的好处是：函数内部动态判断参数类型，如果是整数则进行整数加法，如果是字符串，则进行字符串加法（字符串加法就是字符串连接哈）。也就是说，如果参数是 VARIANT，那么我们就可以实现函数的可变参数类型呀。怪怪个咙，真爽！

五、脚本中调用举例

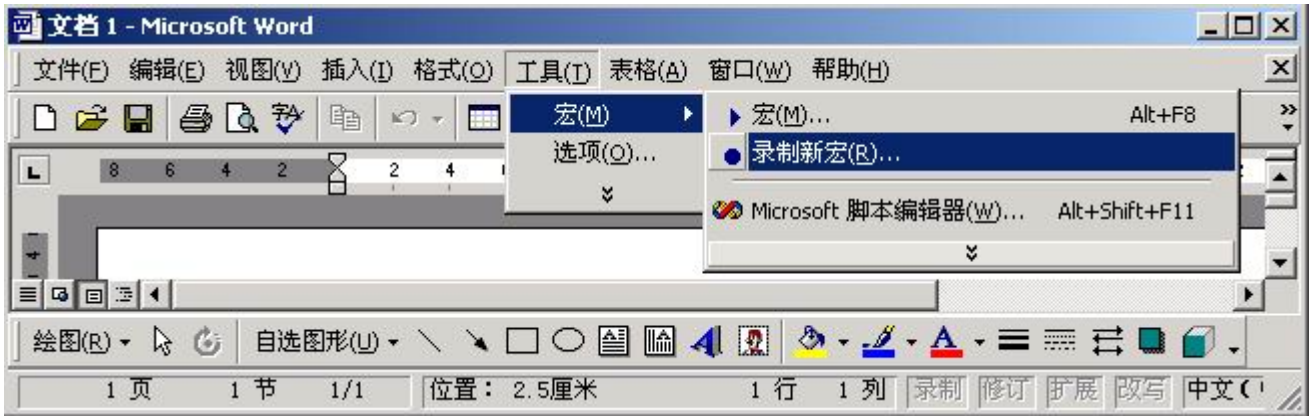
打开“记事本”程序，输入脚本程序，保存为 xxx.vbs 文件。然后在资源管理器里就可以双击运行啦。

如果你有能力，也可以用 JScript 书写上面的程序，然后保存为 xxx.js 文件，同样也可以在资源管理器里运行。另外需要说明的一点是，脚本程序文件的图标(win 2000 下)是  ，如果你不是这样的（有一个软件叫“XX 解霸”。写这款软件的人水平太低，它居然使用 .vbs 的扩展名文件作为它的数据流文件，破坏了系统默认的文件类型影射模式，咳.....），那么需要重新设置，方法是：

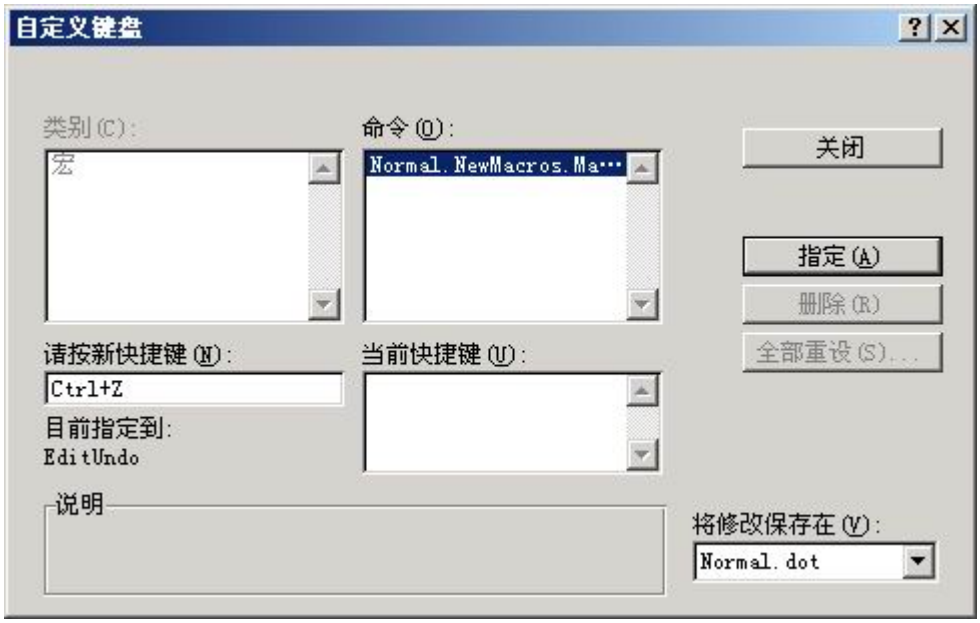


六、WORD 中使用举例

6-1：录制一段宏程序

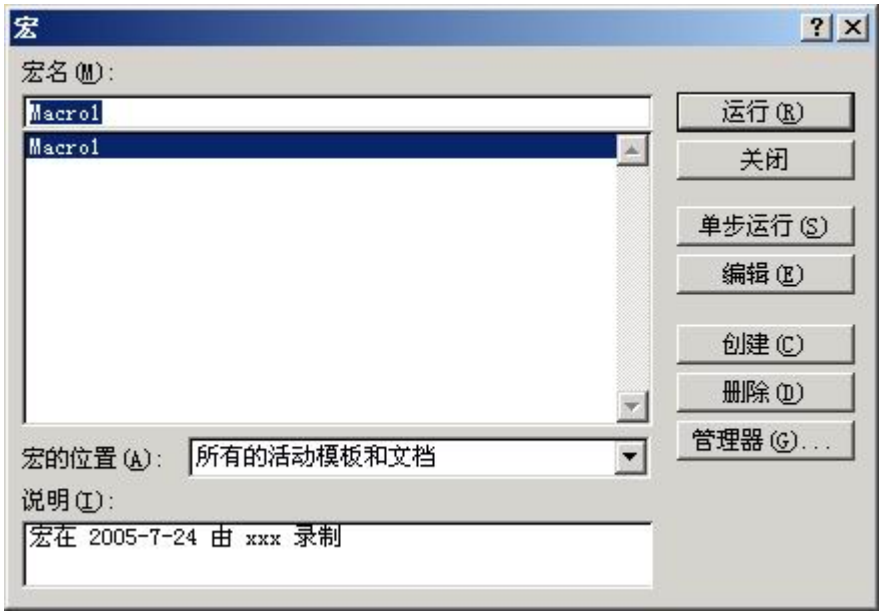


6-2：选择“键盘”，当然你也可以把这个“宏”程序放到“工具栏”上去。这里我们随便指定一个快捷键，比如 Ctrl+Z

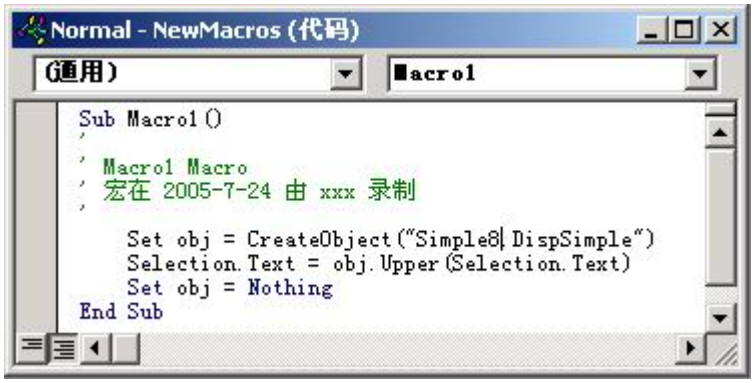


6-3：开始录制了，下面你随便输入点什么东东。然后点“停止”

6-4：接下来，我们执行菜单，选择这个刚刚录制的宏，然后编辑它



6-5：点“编辑”按钮，输入下面的程序：



不做解释了，你如果会一点点 VB ，就能看懂这个东东哈。然后保存关闭 VBA 的编辑器(注4)。

6-6：执行啦，执行啦，看看有什么效果呀.....

然后按快捷键 Ctrl+Z

你已经扩展了 MS WORD 的功能啦，嘿啦啦啦啦，嘿啦啦啦，天空出彩霞呀.....我们只是举了一个简单的例子，其实这个例子并没有什么实际应用的意义，因为人家 WORD 本身就有大小写转换功能。但通过这个小例子，你可以体会出自动化组件的功能了，有够厉害吧？！

七、小结

没小结！嘿嘿.....上当喽:-)

注 1：以后我们描述接口函数，都采用 IDL 的形式了。
注 2：双接口，是支持 IDispatch 接口的一种特殊接口方式，后面马上就要讲啦
注 3：VBA 是专门开发 Office 的一种语言---Visual Basic for Application

COM 组件设计与应用（十一）
IDispatch 及双接口的调用

下载源代码

一、前言

前段时间，由于工作比较忙，没有能及时地写作。其间收到了很多网友的来信询问和鼓励，在此一并表示感谢。咳.....我也需要工作来养家糊口呀.....

上回书介绍了两种方法来写自动化(IDispatch)接口的组件程序，一是用 MFC 方式编写“纯粹”的 IDispatch 接口；二是用 ATL 方式编写“双接口”的组件。

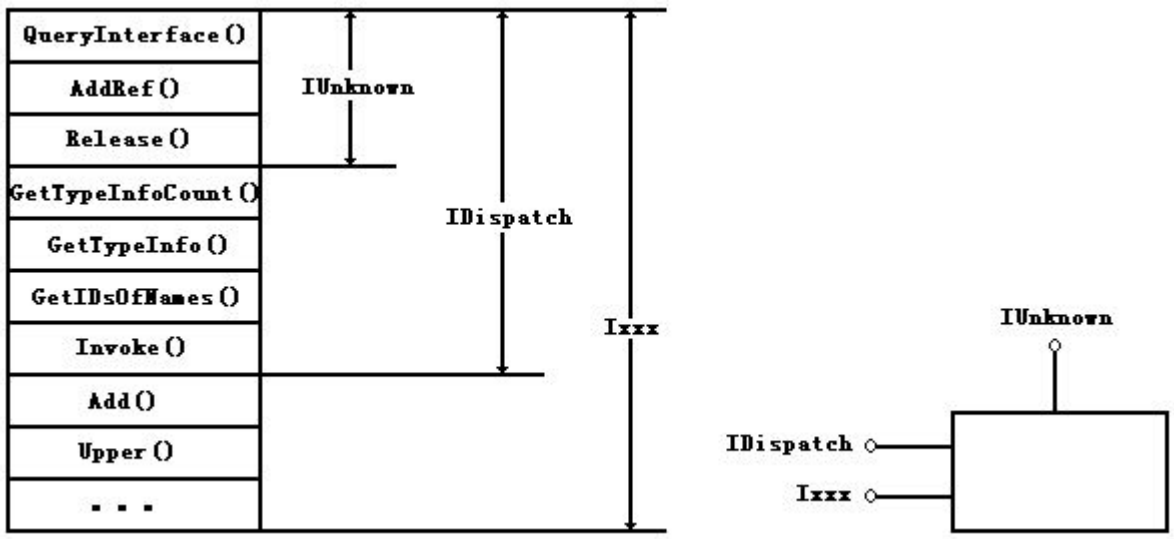
二、IDispatch 接口和双接口

使用者要想调用普通的 COM 组件功能，必须要加载这个组件的类型库(Type library)文件 tlb(比如在 VC 中使用 #import)。然而，在脚本程序中，由于脚本是被解释执行的，所以无法使用加载类型库的方式进行预编译。那么脚本解释器如何使用 COM 组件那？这就是自动化(IDispatch)组件大显身手的地方了。IDispatch 接口需要实现 4 个函数，调用者只通过这 4 个函数，就能实现调用自动化组件中所有的函数。这 4 个函数功能如下：

HRESULT GetTypeInfoCount([out] UINT * pctinfo)	组件中提供几个类型库？当然一般都是一个啦。 但如果你在一个组件中实现了多个 IDispatch 接口，那就不一定啦（注 1）
HRESULT GetTypeInfo([in] UINT iTInfo, [in] LCID lcid, [out] ITypeInfo ** ppTInfo)	调用者通过该函数取得他想要的类型库。 幸好，在 99% 的情况下，我们都不用关心这两个函数的实现，因为 MFC/ATL 都帮我们完成了默认的一个实现，如果是自己完成函数代码，甚至可以直接返回 E_NOTIMPL 表示没有实现。（注 2）
HRESULT GetIDsOfNames([in] REFIID riid, [in,size_is(cNames)] LPOLESTR * rgszNames, [in] UINT cNames, [in] LCID lcid, [out,size_is(cNames)] DISPID * rgDispId)	根据函数名称取得函数序号，为调用 Invoke() 做准备。 所谓函数序号，大家去观察双接口 IDL 文件和 MFC 的 ODL 文件，每一个函数和属性都会有 [id(序号)....] 这样的描述。
HRESULT Invoke([in] DISPID dispIdMember, [in] REFIID riid,	根据序号，执行函数。 使用 MFC/ATL 写的组件程序，我们也不必关心这个函数的实现。如果是自己写代码，则该函数类似

[in] LCID lcid,
[in] WORD wFlags,
[in,out] DISPPARAMS * pDispParams,
[out] VARIANT * pVarResult,
[out] EXCEPINFO * pExcepInfo,
[out] UINT * puArgErr)
	如下实现：
	switch(dispIdMember)
	{
	case 1: .....; break;
	case 2: .....; break;
	....
	}
	其实，就是根据序号进行分支调用啦。(注 3)

从 Invoke() 函数的实现就可以看出，使用 IDispatch 接口的程序，其执行效率是比较低的。ATL 从效率出发，实现了一种叫“双接口(dual)”的接口模式。下面我们来看看，到底什么是双接口：



图一、双接口(dual) 结构示意图

从上图中可以看出，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，叫做双接口）。我们如果从任意一个接口中调用 QueryInterface()得到另外的接口指针的话，其实，得到的指针地址都是同一个。双接口有什么好处那？答：好呀，多好呀，特别好呀.....

使用方式	因为	所以
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	它认识 IDispatch 接口	可以调用，执行效率比较低

件		
编译型语言使用组件	它装载类型库后，就认识了 Ixxx 接口	可以直接调用 Ixxx 函数，效率最高啦
结论	双接口，既满足脚本语言的使用方便性，又满足编译型语言的使用高效性。 于是，我们写的所有的 COM 组件接口，都用双接口实现吗？ 错！否！NO！ 如果不是明确非要支持脚本的调用，则最好不要使用双接口，因为：	
	如果所有函数都放在一个双接口中，那么层次、结构、分类不清	
	如果使用多个双接口，则会产生其它问题（注 4）	
	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喽	
	还有很多弊病呦，不过现在我想不起来喽.....	

三、使用方法

如果你的开发环境是 vc6.0，那么我们使用第九回中的Simple6组件为例，快去下载呀.....

如果你的开发环境是 vc.net 2003，那么用第十回中的 Simple8 组件为例，快去下载呀.....

嘿嘿，其实不下载也没有关系，因为你只要下载本回的示例程序，里面已经包含了所需的组件。但使用前不要忘了去注册呀：regsvr32.exe simple6.dll 或 regsvr32.exe simple8.dll（注意别忘了输入组件的安装目录）。注册成功后，就可以使用了，使用方法有：

示例程序	自动化组件的使用方式	简要说明
示例 0	在脚本中调用	在第九回/第十回中，已经做了介绍
示例 1	使用 API 方式调用	揭示 IDispatch 的调用原理，但傻子才去这么使用那，会累死了
示例 2	使用 CComDispatchDriver 的智能指针包装类	比直接使用 API 方式要简单多啦，这个不错！
示例 3	使用 MFC 装载类型库的包装方式	简单！好用！常用！但它本质上是使用 IDispatch 接口，所以执行效率稍差
示例 4	使用 #import 方式加载类型库方式	#import 方式使用组件，咱们在第七回中讲过啦。常用！对双接口组件，直接调用自定义接口函数，不再经过 IDispatch，因此执行效率最高啦
示例 x	vb 、 java 、 c# 、 bcb 、	反正我不会，自己去请教高人去吧 :-(

	delphi.....	

示例一、IDispatch 调用原理篇

```
void demo()
{
	::CoInitialize( NULL );				// COM 初始化

	CLSID clsid;					// 通过 ProgID 得到 CLSID
	HRESULT hr = ::CLSIDFromProgID( L"Simple8.DispSimple.1", &clsid );
	ASSERT( SUCCEEDED( hr ) ); // 如果失败，说明没有注册组件

	IDispatch * pDisp = NULL;  // 由 CLSID 启动组件，并得到 IDispatch 指针
	hr = ::CoCreateInstance( clsid, NULL, CLSCTX_ALL, IID_IDispatch, (LPVOID
*)&pDisp );
	ASSERT( SUCCEEDED( hr ) ); // 如果失败，说明没有初始化 COM

	LPOLESTR pwFunName = L"Add";		// 准备取得 Add 函数的序号 DispID
	DISPID dispID;				// 取得的序号，准备保存到这里
	hr = pDisp->GetIDsOfNames(		// 根据函数名，取得序号的函数
		IID_NULL,
		&pwFunName,			// 函数名称的数组
		1,				// 函数名称数组中的元素个数
		LOCALE_SYSTEM_DEFAULT,		// 使用系统默认的语言环境
		&dispID );			// 返回值
	ASSERT( SUCCEEDED( hr ) );		// 如果失败，说明组件根本就没有 ADD 函
数

	VARIANTARG v[2];			// 调用 Add(1,2) 函数所需要的参数
	v[0].vt = VT_I4;  v[0].lVal = 2;	// 第二个参数，整数 2
	v[1].vt = VT_I4;  v[1].lVal = 1;	// 第一个参数，整数 1

	DISPPARAMS dispParams = { v, NULL, 2, 0 };	// 把参数包装在这个结构中
	VARIANT vResult;			// 函数返回的计算结果

	hr = pDisp->Invoke(			// 调用函数
		dispID,				// 函数由 dispID 指定
		IID_NULL,
```

```
                LOCALE_SYSTEM_DEFAULT,      // 使用系统默认的语言环境
                DISPATCH_METHOD,            // 调用的是方法，不是属性
                &dispParams,                // 参数
                &vResult,                   // 返回值
                NULL,                       // 不考虑异常处理
                NULL);                      // 不考虑错误处理
        ASSERT( SUCCEEDED( hr ) ); // 如果失败，说明参数传递错误

        CString str;                        // 显示一下结果
        str.Format("1 + 2 = %d", vResult.lVal );
        AfxMessageBox( str );

        pDisp->Release();           // 释放接口指针
        ::CoUninitialize();         // 释放 COM
}
```

示例二、CComDispatchDriver 智能指针包装类的使用方法

```
void demo()
{
        // 已经进行过了 COM 初始化

        CLSID clsid;                // 通过 ProgID 取得组件的 CLSID
        HRESULT hr = ::CLSIDFromProgID( L"Simple8.DispSimple.1", &clsid );
        ASSERT( SUCCEEDED( hr ) ); // 如果失败，说明没有注册组件

        CComPtr < IUnknown > spUnk;// 由 CLSID 启动组件，并取得 IUnknown 指针
        hr = ::CoCreateInstance( clsid, NULL, CLSCTX_ALL, IID_IUnknown, (LPVOID
*)&spUnk );
        ASSERT( SUCCEEDED( hr ) );

        CComDispatchDriver spDisp( spUnk ); // 构造只能指针
        CComVariant v1(1), v2(2), vResult;  // 参数
        hr = spDisp.Invoke2(        // 调用 2 个参数的函数
                L"Add",             // 函数名是 Add
                &v1,                // 第一个参数，值为整数 1
                &v2,                // 第二个参数，值为整数 2
                &vResult);          // 返回值
        ASSERT( SUCCEEDED( hr ) ); // 如果失败，说明或者没有 ADD 函数，或者参数错
误
```

```
        CString str;                    // 显示一下结果
        str.Format("1 + 2 = %d", vResult.lVal );
        AfxMessageBox( str );
}
```

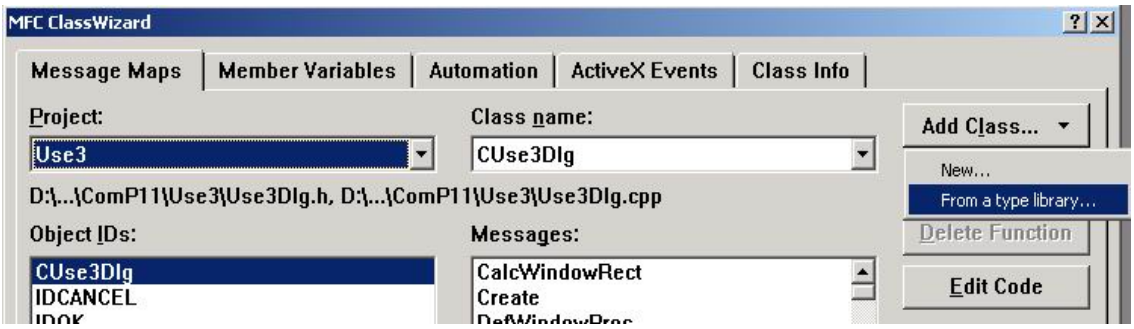
示例程序中使用了 Invoke2()函数，其实你根据不同的函数，还可以使用 Invoke0()、Invoke1()、InvokeN()、PutProperty()、GetProperty()……等等等，的确很方便。

示例三、加载类型库，产生包装类来使用

这个方法使用更简单一些，如果你观察 MFC 帮你产生的包装类的实现，你就会发现，其实它调用的是 IDispatch 接口函数。使用 vc6.0 的朋友，步骤如下：

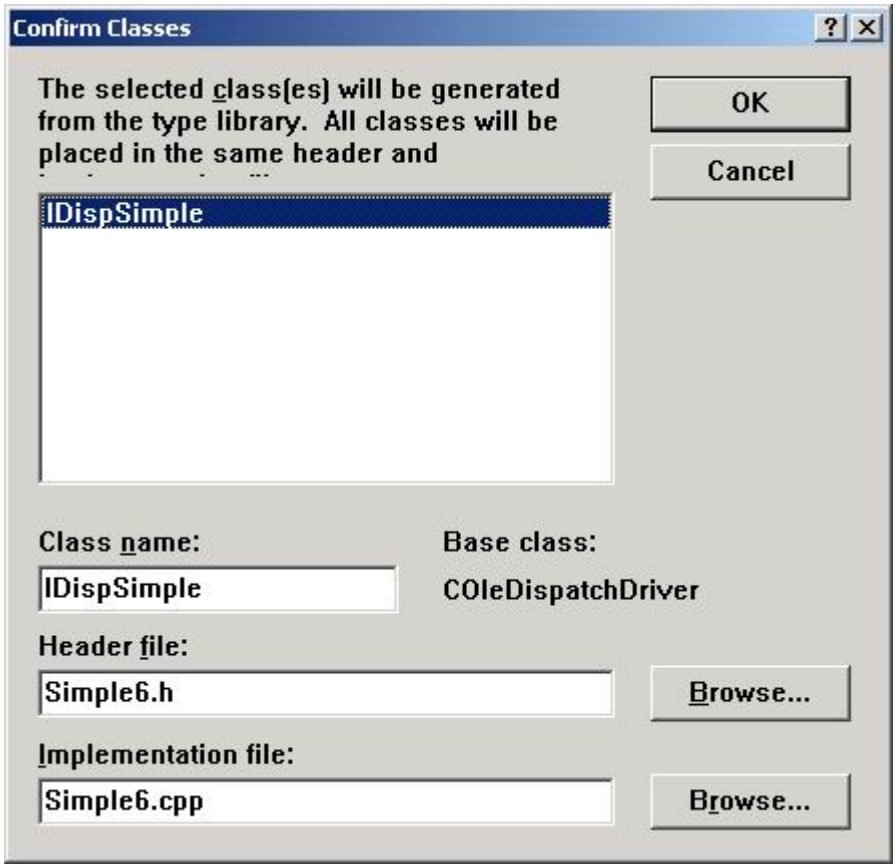
1、建立一个 MFC 的应用程序

2、开启 ClassWizard，执行 Add Class，选择 From a type library



图二、加载类型库

3、然后找到你要使用的组件文件 simple6.dll(tlb 文件也可以)，选择接口后确认



图三、选择类型库中需要包装的接口

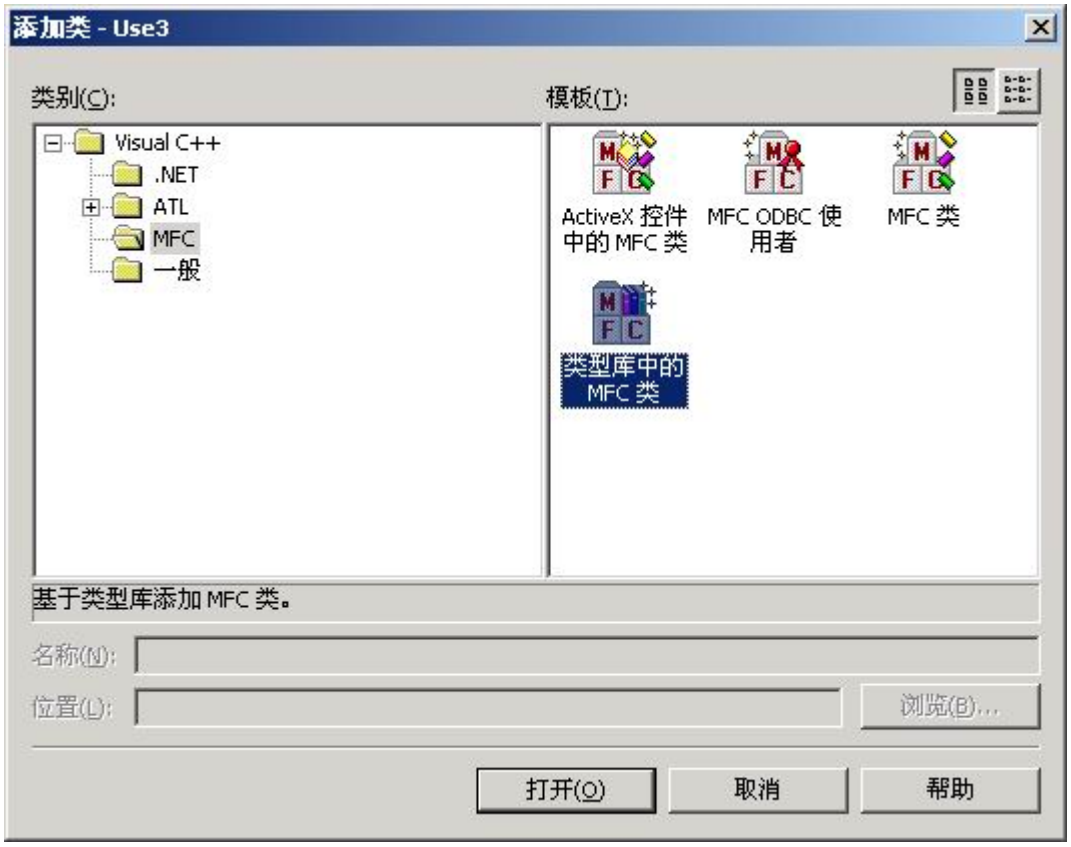
4、在适当的地方输入调用代码

```
#include "simple6.h"        // 包装类的头文件

void demo()
{
        // 已经进行过了 COM 初始化
        IDispSimple spDisp;                    // 包装类的对象
        spDisp.CreateDispatch( _T("Simple6.DispSimple.1") )    //启动组件
        spDisp.xxx(...);  // 调用函数
        spDisp.ReleaseDispatch();  // 释放接口
}
```

使用 vc.net 的朋友，步骤如下：

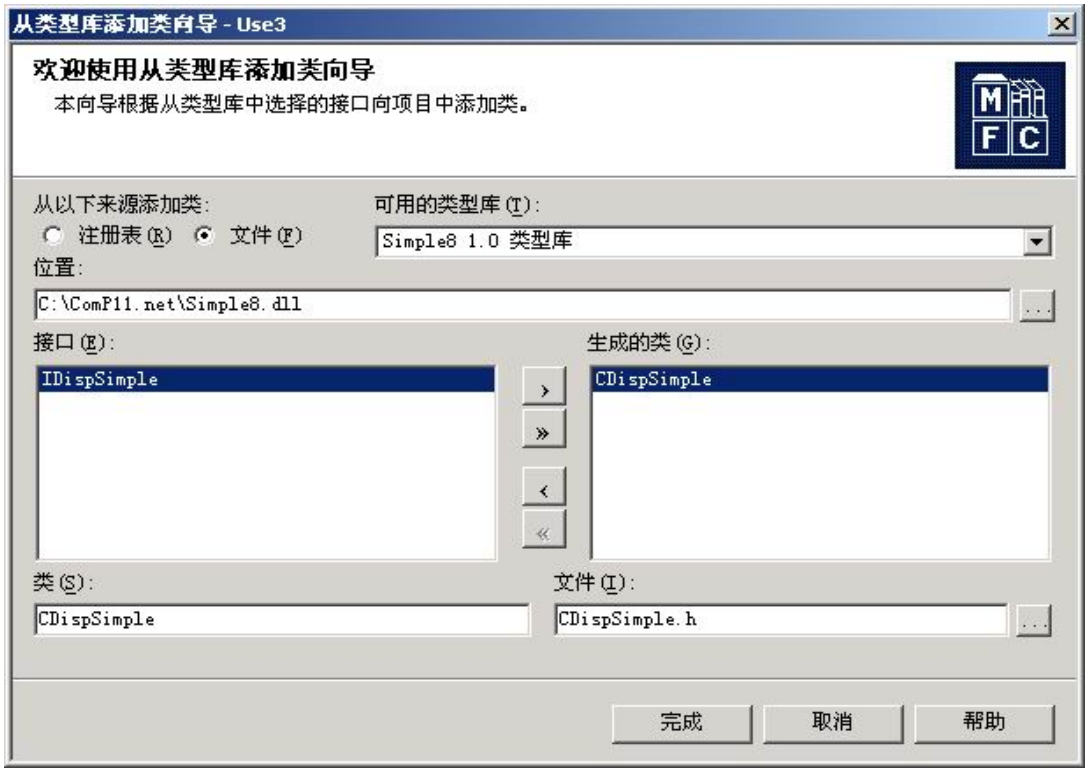
1、建立一个 MFC 的应用程序

2、执行菜单“添加\添加类”，选择 MFC 分类中的“类型库中的 MFC 类”



图四、添加类型库中的 MFC 类

3、选择组件文件 simple8.dll(或 tlb 文件)，并选择需要包装的接口



图五、选择文件和接口

4、在适当的位置输入调用代码

```
#include "CDispSimple.h"    // 包装类的头文件

void demo()
{
        // 已经进行过了 COM 初始化

        CDispSimple spDisp;         // 包装类的对象
        spDisp.CreateDispatch( _T("Simple8.DispSimple.1") )    // 启动组件
        spDisp.xxx(...);  // 调用函数

        spDisp.ReleaseDispatch();  // 释放接口
}
```

示例四、使用 #import 方式调用组件

#import 方式在第七回中已经作过介绍，这里就不多罗嗦了。大家下载本回的示例程序后，自己去看吧。并且一定要掌握这个方法，因为它的运行效率是最快的呀。

四、小结

留作业啦。在我们以前所实现的所有组件程序中，只添加了接口方法（函数），而没有添加接口属性（变量），你自己练习一下吧，很简单的，然后写个程序调用看看。其实对于 VC 来说，调用属性和调用方法没有太大的区别（vc 把属性包装为 GetXXX()/PutXXX()或 getXXX()/putXXX()的函数方式），但在另外一些语言中（比如脚本语言）则更方便，设置属性值是：对象.属性 = 变量或常量，获取属性值是：变量 = 对象.属性。

本回书至此做一了断，更多组件设计和使用的知识，且听下回分解.....

注 1：多个自动化接口的实现方法，我们以后再说。

注 2：将来介绍 ITypeLib::GetTypeInfo() 的时候，大家再回味 IDispatch::GetTypeInfo() 吧。

注 3：在后面介绍“事件”的时候，我们会自己真正去实现一个 IDispatch::Invoke() 函数。

注 4：介绍多个双接口实现的时候，会谈到这个问题。

COM 组件设计与应用（十二）
错误与异常处理

下载源代码

一、前言

程序设计中，错误处理必不可少，而且通常要占用很大的篇幅。本回书着落在 COM 中的错误（异常）的处理方法。

在组件程序中，如果遇到错误，一般有两个方式进行处理。

二、简单返回

对于比较简单的错误，直接返回表示错误原因的 HRESULT。比如下面几个就是常见的错误值：

E_INVALIDARG	0x80070057	参数错误
E_OUTOFMEMORY	0x8007000E	内存错误
E_NOTIMPL	0x80004001	未实现
E_POINTER	0x80004003	无效指针
E_HANDLE	0x80070006	无效句柄
E_ABORT	0x80004004	终止操作
E_ACCESSDENIED	0x80070005	拒绝访问
E_NOINTERFACE	0x80004002	不支持接口

另外，你还可以返回自己构造 HRESULT 错误值。方法是使用宏 MAKE_HRESULT(sev,fac,code)

参数	含义	值（二进制）
sev 严重程度	成功	00
	成功，但有一些报告信息	01
	警告	10
	错误	11
fac 设备信息	FACILITY_AAF	00000010010
	FACILITY_ACS	00000010100
	FACILITY_BACKGROUNDCOPY	00000100000
	FACILITY_CERT	00000001011
	FACILITY_COMPLUS	00000010001
	FACILITY_CONFIGURATION	00000100001
	FACILITY_CONTROL	00000001010
	FACILITY_DISPATCH	00000000010
	FACILITY_DPLAY	00000010101

	FACILITY_HTTP	00000011001
	FACILITY_INTERNET	00000001100
	FACILITY_ITF	00000000100
	FACILITY_MEDIASERVER	00000001101
	FACILITY_MSMQ	00000001110
	FACILITY_NULL	00000000000
	FACILITY_RPC	00000000001
	FACILITY_SCARD	00000010000
	FACILITY_SECURITY	00000001001
	FACILITY_SETUPAPI	00000001111
	FACILITY_SSPI	00000001001
	FACILITY_STORAGE	00000000011
	FACILITY_SXS	00000010111
	FACILITY_UMI	00000010110
	FACILITY_URT	00000010011
	FACILITY_WIN32	00000000111
	FACILITY_WINDOWS	00000001000
	FACILITY_WINDOWS_CE	00000011000
code 唯一错误码	16 位(bit) 你自己定义去吧	

调用者得到返回的 HRESULT 值后，也可以使用宏 HRESULT_SEVERITY()、HRESULT_FACILITY()、HRESULT_CODE() 来取得 sev 错误程度、fac 设备信息和 code 错误代码。

三、错误信息接口

既然 COM 是靠各种各样的接口来提供服务的，于是很自然地就会想到，是否有一个接口能够提供更丰富的错误信息报告那？答案是：ISupportErrorInfo。下面这段代码是使用 ISupportErrorInfo 的一般方法：

```
STDMETHODIMP Cxxx::fun()
{
        ... ... ... ...

        CComQIPtr< ICreateErrorInfo> spCEI;
```

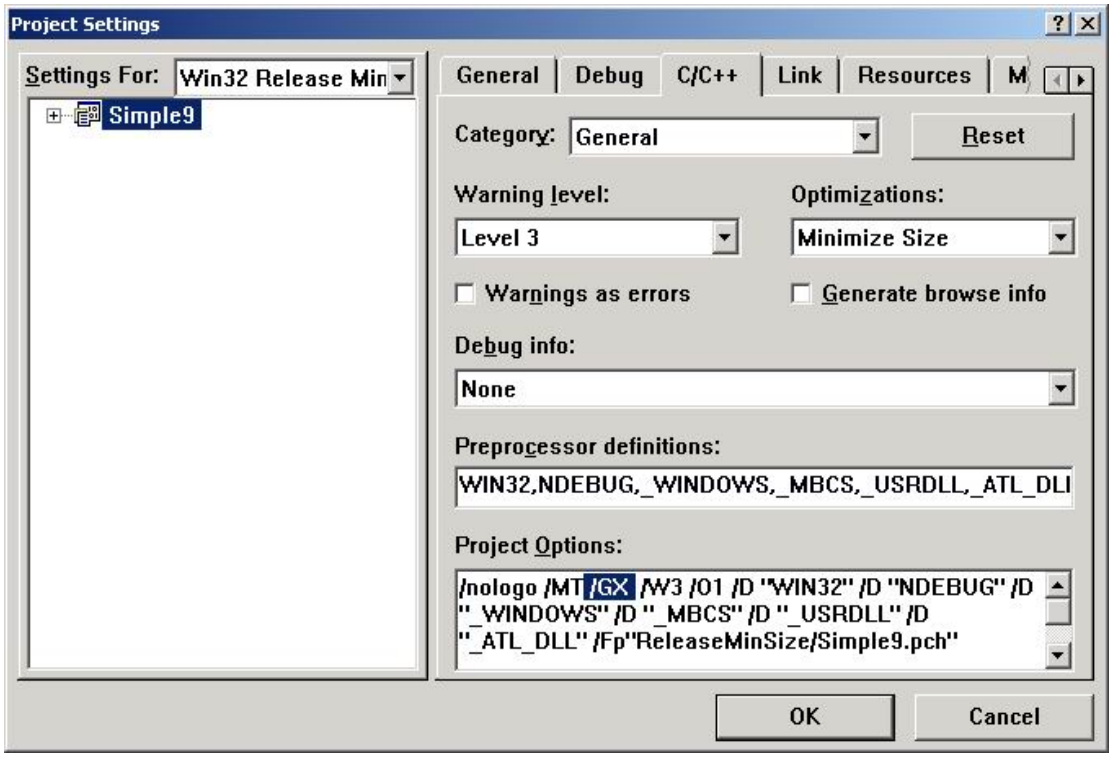
```
        ::CreateErrorInfo( &spCEI );

        spCEI->SetGUID( IID_Ixxx );          // 发生错误的接口 IID

        spCEI->SetSource( L"xxx.xxx" );      // ProgID

        // 如果你的组件同时提供了帮助文件，那么就可以：
        spCEI->SetHelpContext( 0 );          // 设置帮助文件的主题号
        spCEI->SetHelpFile( L"xxx.hlp" );    // 设置帮助文件的文件名

        spCEI->SetDescription( L"错误描述信息" );

        CComQIPtr < IErrorInfo > spErrInfo = spCEI;
        if( spErrInfo )
          ::SetErrorInfo( 0, spErrInfo );    // 这时调用者就可以得到错误信息了

        return E_FAIL;
}
```

上面是原理性代码，在我们写的程序中，不用这么麻烦。因为 ATL 已经把上述的代码给我们包装成 CComCoClass::Error() 的 6 个重载函数了。如此，我们可以非常简单的改写为：

```
STDMETHODIMP Cxxx::fun()
{
        ... ... ... ...

        return Error( L"错误描述信息" );
}
```

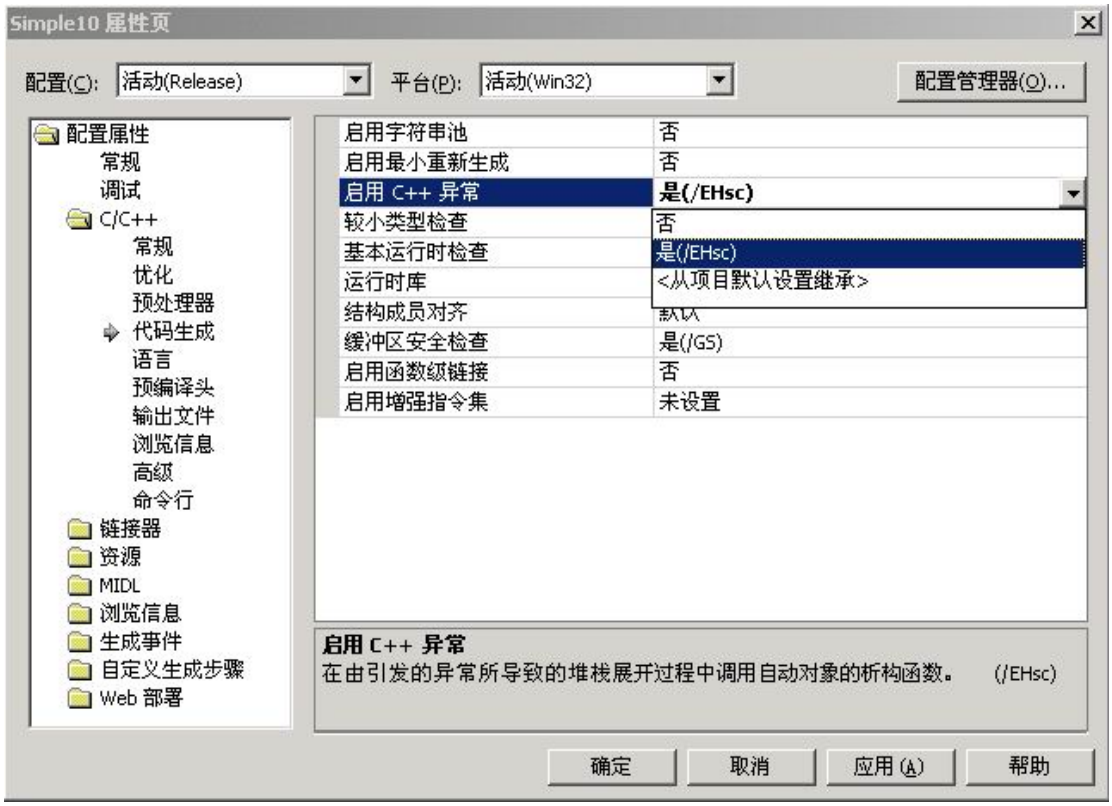
四、关于 try/catch

学习了 C++ 后，很多人都喜欢使用 try/catch 的异常处理结构。如果你使用 vc6.0 的 ATL，编译器默认是不支持异常处理的，编译后会报告“warning C4530: C++ exception handler used, but unwind semantics are not enabled. Specify -GX”，解决方法是手工加上编译开关：



图一、加上编译开关，支持 C++的异常处理结构

在 vc.net 2003 中，编译器默认是支持异常处理结构的，所以不用特别进行设置。如果想减小目标文件的尺寸，你也可以决定不使用 C++ 异常处理，那么在项目属性中



图二、在 vc.net 中修改是否支持 C++异常结构的编译开关

五、客户端接收组件的错误信息

1、如果使用 API 方式调用组件，接收错误的方法是：

```
HRESULT hr = spXXX->fun()  // 调用组件功能
if( FAILED( hr ) )// 如果发生了错误
{
        CComQIPtr <ISupportErrorInfo> spSEI = spXXX;// 组件是否提供了
ISupportErrorInfo 接口?
        if( spSEI )       // 如果支持，那么
        {
                hr = spSEI->InterfaceSupportsErrorInfo( IID_Ixxx );    // 是否支
持 Ixxx 接口的错误处理?
                if( SUCCEEDED( hr ) )
                {       // 支持，太好了。取出错误信息
                        CComQIPtr < IErrorInfo > spErrInfo;          // 声明
IErrorInfo 接口
                        hr = ::GetErrorInfo( 0, &spErrInfo );        // 取得接
口
                        if( SUCCEEDED( hr ) )
                        {
                                CComBSTR bstrDes;
                                spErrInfo->GetDescription( &bstrDes );
        // 取得错误描述
                                .....   // 还可以取得其它的信息
                        }
                }
        }
}
```

2、如果使用 #import 等包装方式调用组件，接收错误的方法是：

```
try
{
        .....   // 调用组件功能
}
catch( _com_error &e )
{
        e.Description();  // 取得错误描述信息
```

```
        ......    // 还可以调用 _com_error 函数取得其它信息
}
```

六、编写支持错误处理的组件程序

非常简单，只要在增加 ATL 组件对象的时候选中 ISupportErrorInfo 即可。

图三、vc6.0 中，选中组件支持错误处理接口

图四、vc.net 2003 中，选中组件支持错误处理接口

七、小结

阅读文章后，请下载本回的示例程序。示例程序中演示了三种错误处理方法和三种接收

错误的方法，同时程序中也有比较详细的注释。

COM 组件设计与应用（十三）
事件和通知(VC6.0)

下载源代码

一、前言

我的 COM 组件运行时产生一个窗口，当用户双击该窗口的时候，我需要通知调用者；

我的 COM 组件用线程方式下载网络上的一个文件，当我完成任务后，需要通知调用者；

我的 COM 组件完成一个钟表的功能，当预定时间到达的时候，我需要通知调用者；

…… …… …… ……

本回书开始话说 COM 的事件、通知、连接点.....这些内容比较多，我分两次（共四回）来介绍。

二、通知的方法

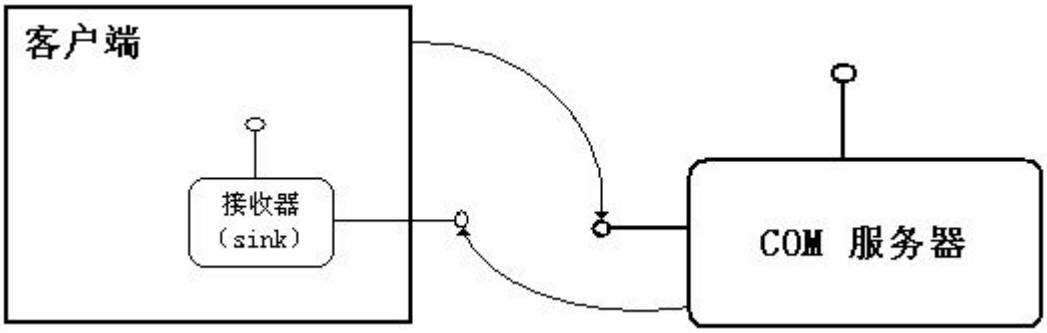
当程序甲方内部发生了某个事件的时候，需要通知乙方，无非使用几个方法：

通知方式		简单说明	评论
直接消息	PostMessage() PostThreadMessage()	向窗口或线程发个消息	你什么时候执行我就不管啦
	SendMessage()	马上执行消息响应函数	不执行完消息处理函数不会返回
	SendMessage(WM_COPYDATA...)	发消息的同时，还可以带过去一些自定义的数据	比较常用，所以单独列了出来
间接消息	InvalidateRect() SetTimer()	被调用的函数会发送相关的一些消息	这样的函数太多了
回调	GetOpenFileName().....	当用户改变文件选择的时候，执行回调函数	嗨！哥们，这是我的电

函数			话，有事就言语一声。

在 COM 的时代，以上这些方法就基本上不能玩转了，因为...您想呀 **COM 组件是运行在分布式环境中的**，地球另一边计算机上运行的组件，怎么可能给你的窗口发消息那？当然不能！（但话又说回来，对于 ActiveX 这样只能在本地运行的组件，当然也可以发送窗口消息的啦。）

回调函数的方式，是设计 COM 通知方法的基础。回调函数，本质上是预先把某一函数的指针告诉我，当我有必要的时候，就直接呼叫该函数了，而这个回调函数做了什么，怎么做的，我是根本不关心的。好了，问你个问题：啥是 COM 的接口？接口其实就是一组相关函数的集合（这个定义不严谨，但你可以这么理解哈）。因此，在 COM 中不使用“回调函数”而是使用“回调接口”（说的再清楚一些，就是使用一大堆包装好的“回调函数”集），回调接口，我们也叫“接收器接口”。



图一、客户端传递接收器接口指针给 COM。当发生事件时，COM 调用接收器接口函数完成通知

本回示例程序完成的功能是：

客户端启动组件(Simple11.IEvent1.1)并得到接口指针 IEvent1 *；

调用接口方法 IEvent1::Advise() 把客户端内部的一个接收器(sink)接口指针(ICallBack *)传递到组件服务器中；

调用 IEvent1::Add() 去计算两个整数的和；

但是计算结果并不通过该函数返回，而是通过 ICallBack::Fire_Result() 返回给客户端；

当客户端不再需要接受事件的时候，调用 IEvent1::Unadvise() 断开和组件的联系。

三、组件实现步骤

1、建立一个工作区(WorkSpace)

2、在工作区中，建立一个 ATL 工程(Project)。示例程序中工程名称叫 Simple11，接受全部默认选项。

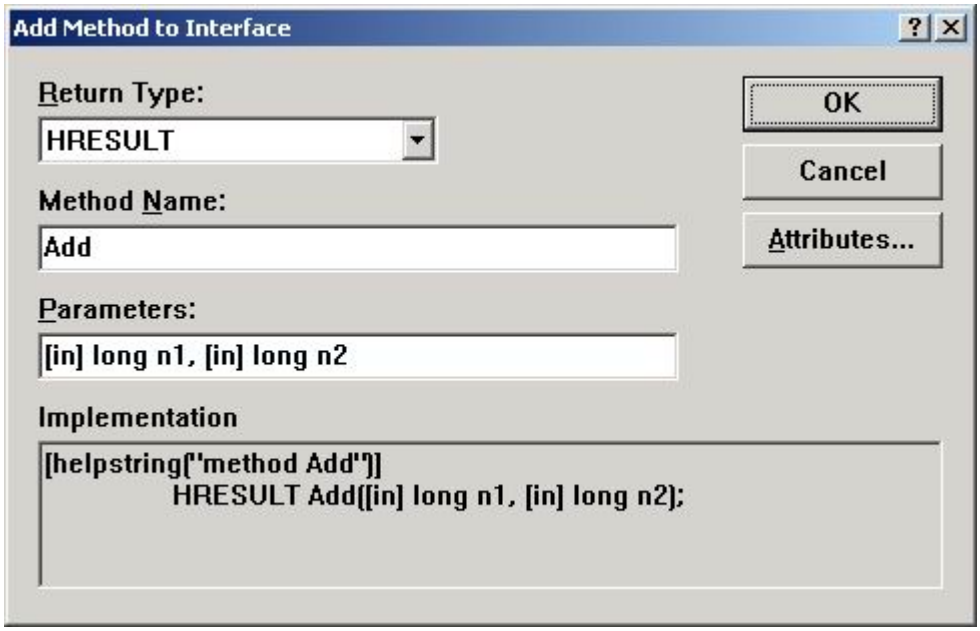
3、ClassView 中，执行鼠标右键菜单命令 New Atl Object...，添加 ALT 类。

3-1、左侧分类 Category 选择 Objects，右侧 Objects 选择 SimpleObject（其实就是默认项目）

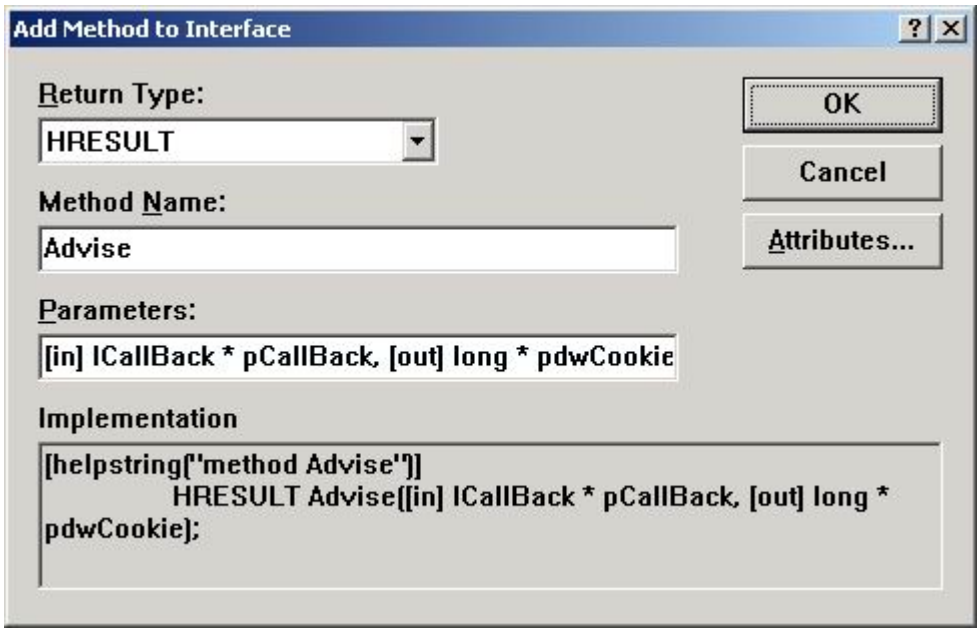
3-2、名称 Name 卡片中，输入组件名称。示例程序中是 Event1（注 1）

3-3、属性 Attributes 卡片中，修改接口类型 Interface 为定制的 Custom（注 2）

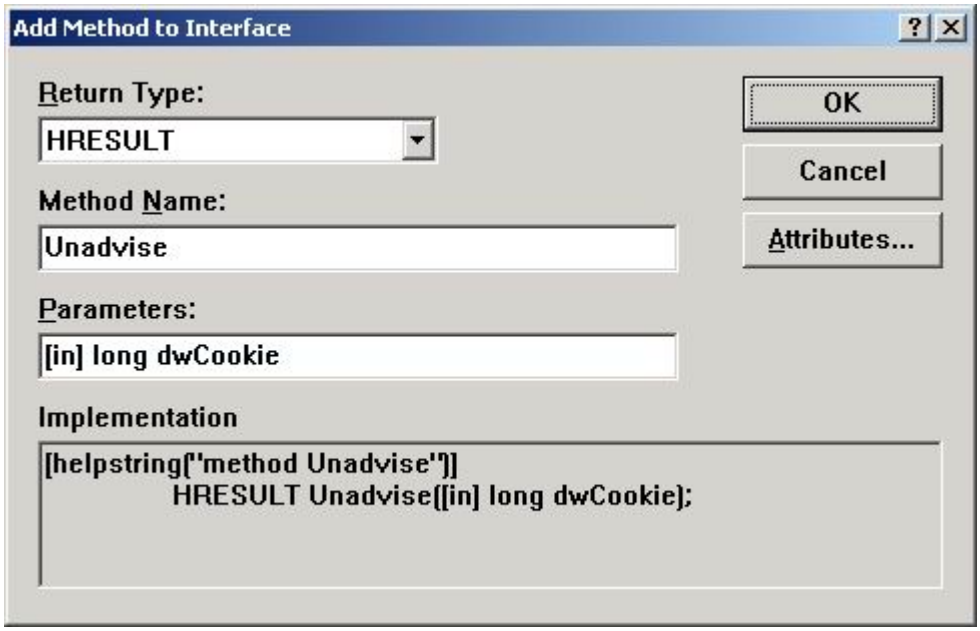
4、ClassView 中，选择接口（IEvent1），鼠标右键菜单添加函数 Add Method...



图二、增加接口函数 Add([in] long n1,[in] long n2)



图三、增加接口函数 Advise([in] ICallBack *pCallBack,[out] long *pdwCookie)



图四、增加接口函数 Unadvise([in] long dwCookie)

你应该注意到了，在 Add()函数中，并没有[out]、[retval] 这样的 IDL 属性，嘿嘿，因为我们本来就不打算通过 Add() 函数直接得到计算结果。不然怎么演示回调接口呀:-) 另外，在函数 Advise()中，需要返回一个整数 dwCookie，这是干什么？道理很简单，因为我们的组件想同时支持多个对象的回调连接。因此当客户端传递一个接口给我们组件的时候，我返回给它唯一的一个 cookie 号码来表示身份，将来断开连接的时候 Unadvise()，它需要把这个 cookie 身份号再给我，这样我就知道是谁想断开了。

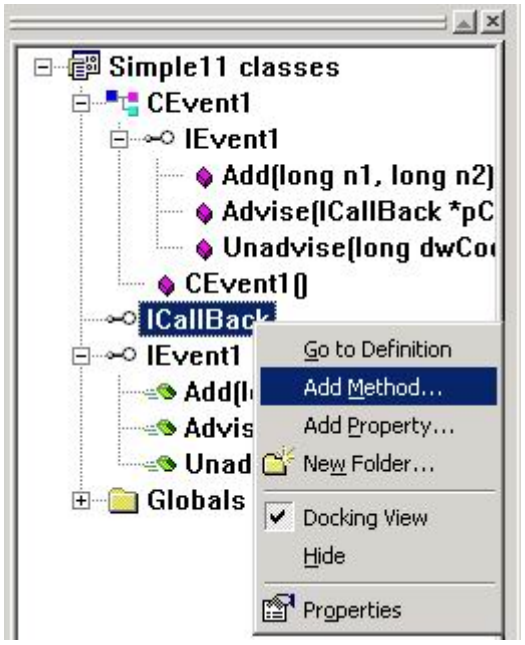
5、增加回调接口 ICallBack 的 IDL 定义。打开 IDL 文件并手工输入（黑体字部分为手工输入的），然后保存：

```
import "oaidl.idl";
import "ocidl.idl";
[
        object,
        uuid(7E659BB1-FB79-4188-9661-65CA22B6A3E6),  // 这个 IID 可以用
GUDIGEN.EXE 产生

        helpstring("ICallBack Interface"),
        pointer_default(unique)
]
interface ICallBack : IUnknown
```

```
{

};

[
        object,    // 以下内容同示例程序,当然如果是你自己生成的程序就肯定有差别的啦
        uuid(7E659BB0-FB79-4188-9661-65CA22B6A3E6),

        helpstring("IEvent1 Interface"),
        pointer_default(unique)
]
interface IEvent1 : IUnknown
{
        [helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2);
       [helpstring("method Advise")] HRESULT Advise([in] ICallBack * pCallBack, [out]
long * pdwCookie);
        [helpstring("method Unadvise")] HRESULT Unadvise([in] long dwCookie);
};

[
        uuid(695C9BB2-2AE9-4232-8225-17AB8BD3BABC),
        version(1.0),
        helpstring("Simple11 1.0 Type Library")
]
library SIMPLE11Lib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(6FCF997C-C811-49DB-9D16-46FAF8D24822),
                helpstring("Event1 Class")
        ]
        coclass Event1
        {
                [default] interface IEvent1;
                // 需要手工输入，据说 VB 使用的话，不能有 [source,default] 属性
                [source, default] interface ICallBack;
```

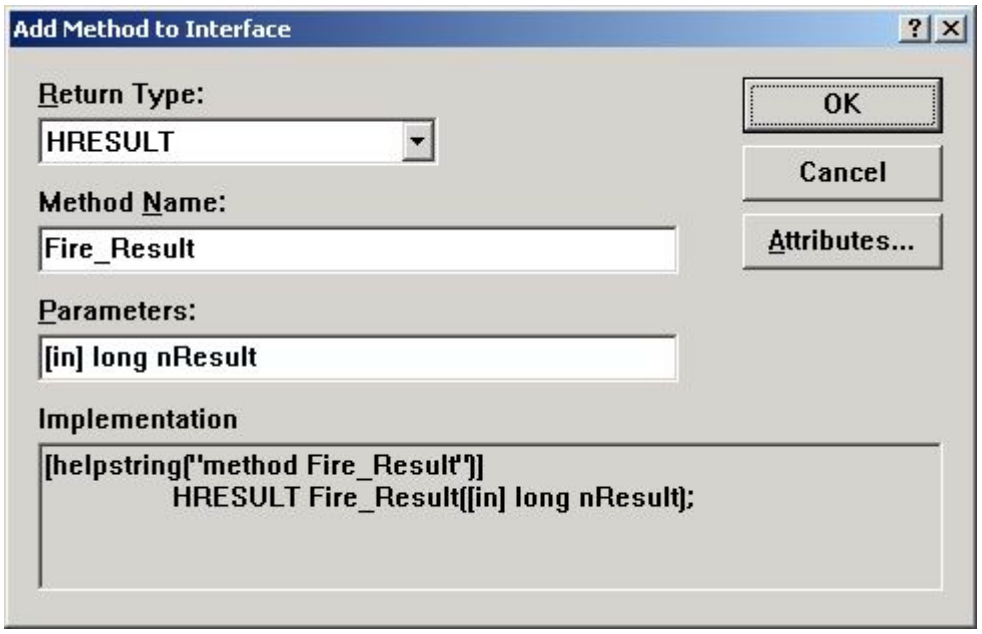
```
        };
};
```

6、增加回调接口函数



图五、增加回调接口函数

其实和以前的方法一样，只要注意别选错了接口就好。



图六、增加接口函数 Fire_Result([in] long nResult)

我们计算整数和，得到结果后，就是要靠这个回调接口函数去反馈给客户端呀。

7、添加组件内部保存回调接口指针的数组

刚才已经说过，我们这个组件打算支持多个对象的回调连接，因此我们要使用一个数组来保存。在 ClassView 中，选择 CEvent1 类，增加成员变量 Add Member Variable...

图七、增加保存 ICallBack * 的数组

当然，保存一个数组可以有多种方式。示例程序比较简单，定义了一个 10 个元素空间的成员数组变量。如果你已经学会了使用 STL，那么你也可以用 vector 等容器来实现。**注意！注意！注意！在构造函数中别忘了初始化数组元素为 NULL**。

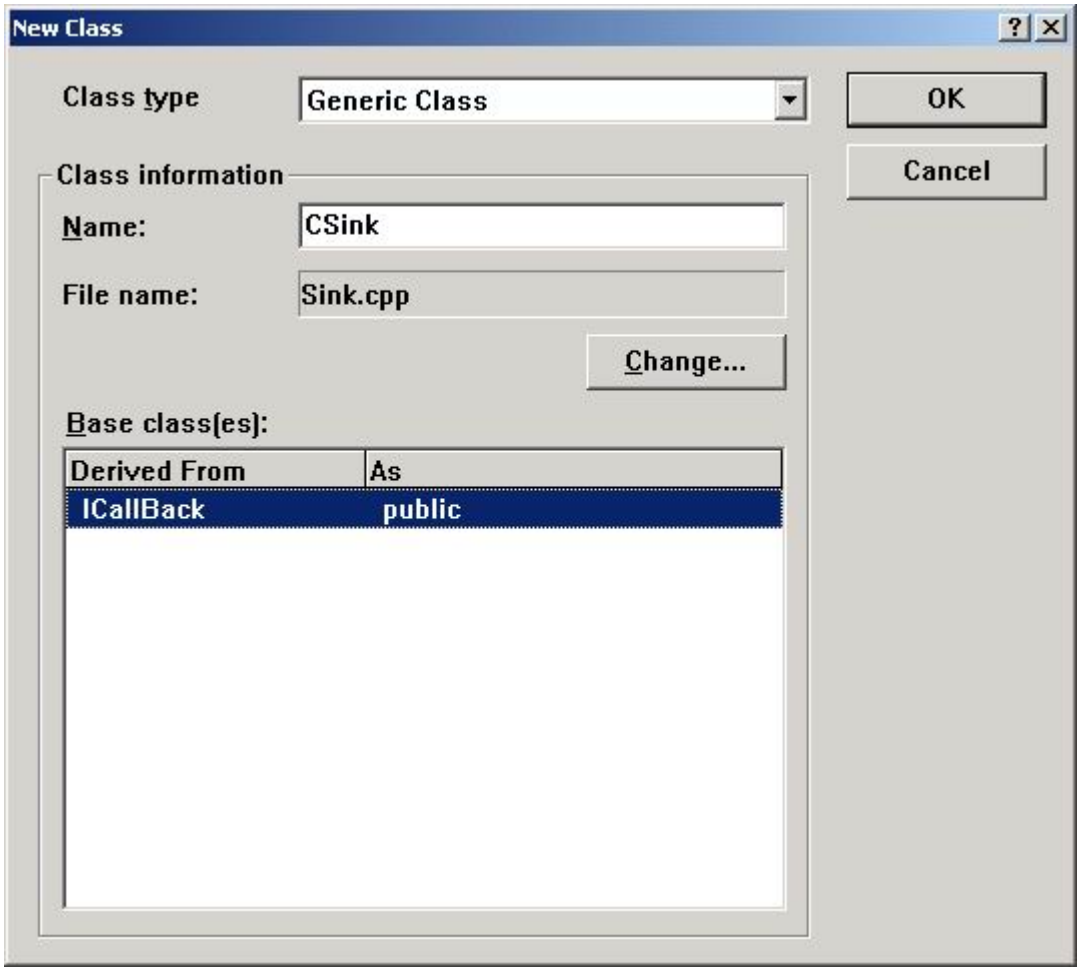
8、好了，下面开始完成所有代码

```
STDMETHODIMP CEvent1::Add(long n1, long n2)
{
	long nResult = n1 + n2;
	for( int i=0; i<10; i++)
	{
		if( m_pCallBack[i] )	// 如果回调接口有效
			m_pCallBack[i]->Fire_Result( nResult );// 则发出事件/通知
	}

	return S_OK;
}

STDMETHODIMP CEvent1::Advise(ICallBack *pCallBack, long *pdwCookie)
{
	if( NULL == pCallBack )	// 居然给我一个空指针？！
```

```
                return E_INVALIDARG;

        for( int i=0; i<10; i++)    // 寻找一个保存该接口指针的位置
        {
                if( NULL == m_pCallBack[i] )        // 找到了
                {
                        m_pCallBack[i] = pCallBack;// 保存到数组中
                        m_pCallBack[i]->AddRef();  // 指针计数器 +1

                        *pdwCookie = i + 1;        // cookie 就是数组下标
                // +1 的目的是避免使用 0，因为 0 表示无效

                        return S_OK;
                }
        }
        return E_OUTOFMEMORY;       // 超过 10 个连接，内存不够用啦
}

STDMETHODIMP CEvent1::Unadvise(long dwCookie)
{
        if( dwCookie<1 || dwCookie>10 )      // 这是谁干的呀？乱给参数
                return E_INVALIDARG;

        if( NULL == m_pCallBack[dwCookie - 1 ])// 参数错误，或该接口指针已经无效了
                return E_INVALIDARG;

        m_pCallBack[ dwCookie -1 ]->Release();        // 指针计数器 -1
        m_pCallBack[ dwCookie -1 ] = NULL;            // 空出该下标的数组元素

        return S_OK;
}
```

四、客户端实现步骤

大家下载示例程序后，去浏览客户端的实现程序吧。这里我只说明一下关于接收器是如何构造的：



图八、从 ICallBack 派生接收器类 CSink

从 ICallBack 派生一个类 CSink。确认后 IDE 会有一个警告，说它找不到 ICallBack 的头文件，不用理它，因为只有当编译的时候，#import 才会为我们生成 xxxx.tlh、xxxx.tli 文件，这些文件就有 ICallBack 的声明啦。

这里 ICallBack 是 COM 接口，因此 CSink 是不能事例化的，如果你去编译，会得到一坨一坨（注 3）的错误，报告说你没有实现 virtual 函数。然后，我们可以按照错误报告，去实现所有的虚函数：

```
// STDMETHODIMP 是宏，等价于 long __stdcall
STDMETHODIMP CSink::QueryInterface(const struct _GUID &iid,void ** ppv)
{
        *ppv=this;          // 不管想得到什么接口，其实都是对象本身
        return S_OK;
}

ULONG __stdcall CSink::AddRef(void)
{       return 1;}// 做个假的就可以，因为反正这个对象在程序结束前是不会退出的
```

```
ULONG __stdcall CSink::Release(void)
{       return 0;}// 做个假的就可以，因为反正这个对象在程序结束前是不会退出的

STDMETHODIMP CSink::raw_Fire_Result(long nResult)
{
        ... ...  // 把计算结果显示在窗口中
        return S_OK;
}
```

五、小结

COM 组件实现事件、通知这样的功能有两个基本方法。今天介绍的回调接口方式非常好，速度快、结构清晰、实现也不复杂；下回书介绍连接点方式(Support Connection Points)，连接点方法其实并不太好，速度慢（如果是远程 DCOM 方式，要谨慎选择它）、结构复杂、唯一的好处就是 ATL 对它进行了包装，所以实现起来反而比较简单。不介绍又不行，因为微软绝大数支持事件的组件都是用连接点实现的，咳.....讨厌的微软(注 4)。

注 1：本来设想多举几个例子，因此第一个叫 Event1，可写完后，感觉程序已经比较复杂了，就没继续再做了。
注 2：当然，你选择使用双接口 Dual 也没有问题。但要注意到在下面的步骤，增加回调接口修改 IDL 文件的时候,我们是要使用 Custom(从 IUnknown 派生，而不是从 IDispatch 派生)的。
注 3：一坨一坨经常用来形容一堆一堆的狗屎。
注 4：微软的同志们，玩笑话不要当真呀！我还靠着你来吃饭那。

COM 组件设计与应用（十四）
事件和通知(vc.net)

下载源代码

一、前言

我的 COM 组件运行时产生一个窗口，当用户双击该窗口的时候，我需要通知调用者；

我的 COM 组件用线程方式下载网络上的一个文件，当我完成任务后，需要通知调用者；

我的 COM 组件完成一个钟表的功能，当预定时间到达的时候，我需要通知调用者；

... ...

本回书开始话说 COM 的事件、通知、连接点.....这些内容比较多，我分两

次（共四回）来介绍。

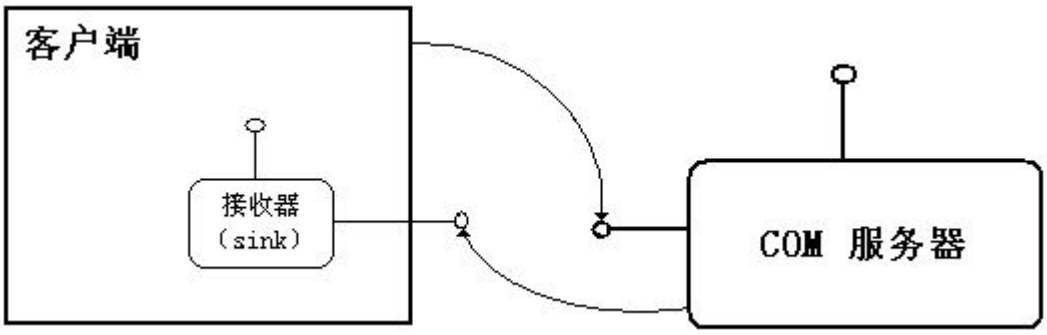
二、通知的方法

当程序甲方内部发生了某个事件的时候，需要通知乙方，无非使用几个方法：

通知方式		简单说明	评论
直接消息	PostMessage() PostThreadMessage()	向窗口或线程发个消息	你什么时候执行我就不管啦
	SendMessage()	马上执行消息响应函数	不执行完消息处理函数不会返回
	SendMessage(WM_COPYDATA...)	发消息的同时，还可以带过去一些自定义的数据	比较常用，所以单独列了出来
间接消息	InvalidateRect() SetTimer()	被调用的函数会发送相关的一些消息	这样的函数太多了
回调函数	GetOpenFileName().....	当用户改变文件选择的时候，执行回调函数	嗨！哥们，这是我的电话，有事就言语一声。

在 COM 的时代，以上这些方法就基本上不能玩转了，因为...您想呀 **COM 组件是运行在分布式环境中的**，地球另一边计算机上运行的组件，怎么可能给你的窗口发消息那？当然不能！（但话又说回来，对于 ActiveX 这样只能在本地运行的组件，当然也可以发送窗口消息的啦。）

回调函数的方式，是设计 COM 通知方法的基础。回调函数，本质上是预先把某一函数的指针告诉我，当我有必要的时候，就直接呼叫该函数了，而这个回调函数做了什么，怎么做的，我是根本不关心的。好了，问你个问题：啥是 COM 的接口？接口其实就是一组相关函数的集合（这个定义不严谨，但你可以这么理解哈）。因此，在 COM 中不使用“回调函数”而是使用“回调接口”（说的再清楚一些，就是使用一大堆包装好的“回调函数”集），回调接口，我们也叫“接收器接口”。



图一、客户端传递接收器接口指针给 COM。当发生事件时，COM 调用接收器接口函数完成通知

本回示例程序完成的功能是：

客户端启动组件(Simple11.IEvent1.1)并得到接口指针 IEvent1 *；

调用接口方法 IEvent1::Advise() 把客户端内部的一个接收器(sink)接口指针(ICallBack *)传递到组件服务器中；

调用 IEvent1::Add() 去计算两个整数的和；

但是计算结果并不通过该函数返回，而是通过 ICallBack::Fire_Result() 返回给客户端；

当客户端不再需要接受事件的时候，调用 IEvent1::Unadvise() 断开和组件的联系。

三、组件实现步骤

1、建立一个解决方案

2、在解决方案中，建立一个 ATL 项目。示例程序中项目名称叫 Simple12，取消“属性化”，其它接受默认选项。

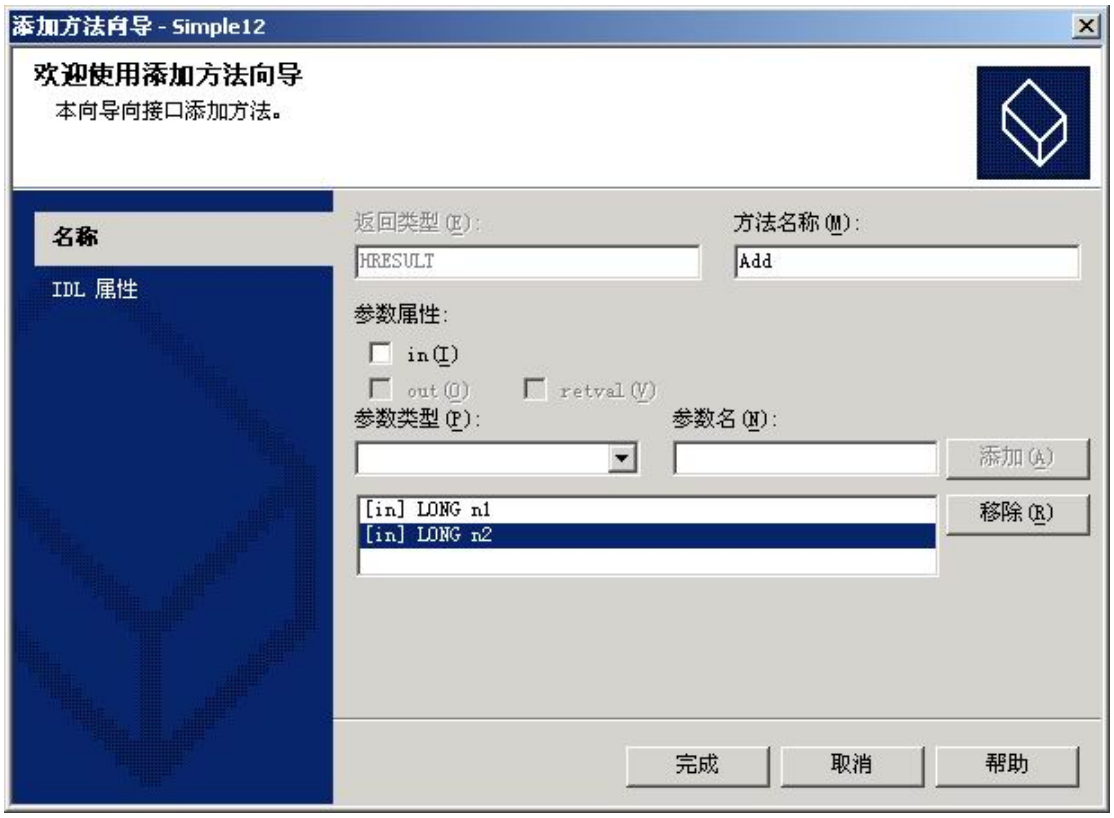
3、选择项目，执行鼠标右键菜单命令“添加\添加类”。

3-1、左侧分类选择 ATL，右侧模板选择 Atl 简单对象

3-2、名称卡片中，输入组件名称。示例程序中是 Event1（注 1）

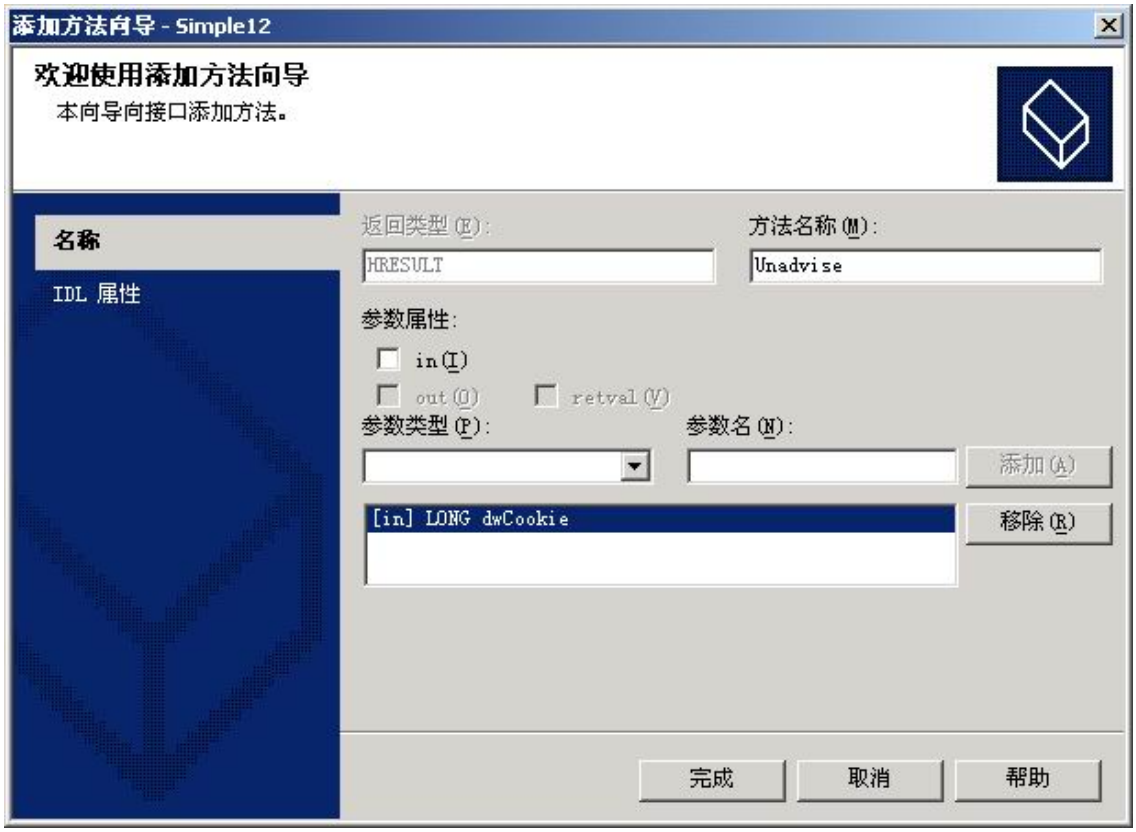
3-3、选项卡片中，修改接口类型“自定义”（注 2）

4、选择 IEnvent1 接口，鼠标右键菜单“添加\添加方法”



图二、增加接口函数 Add([in] long n1,[in] long n2)

图三、增加接口函数 Advise([in] ICallBack *pCallBack,[out] long *pdwCookie)



图四、增加接口函数 Unadvise([in] long dwCookie)

你应该注意到了，在 Add()函数中，并没有[out]、[retval] 这样的 IDL 属性，嘿嘿，因为我们本来就不打算通过 Add() 函数直接得到计算结果。不然怎么演示回调接口呀:-) 另外，在函数 Advise()中，需要返回一个整数 dwCookie，这是干什么？道理很简单，因为我们的组件想同时支持多个对象的回调连接。因此当客户端传递一个接口给我们组件的时候，我返回给它唯一的一个 cookie 号码来表示身份，将来断开连接的时候 Unadvise()，它需要把这个 cookie 身份号再给我，这样我就知道是谁想断开了。

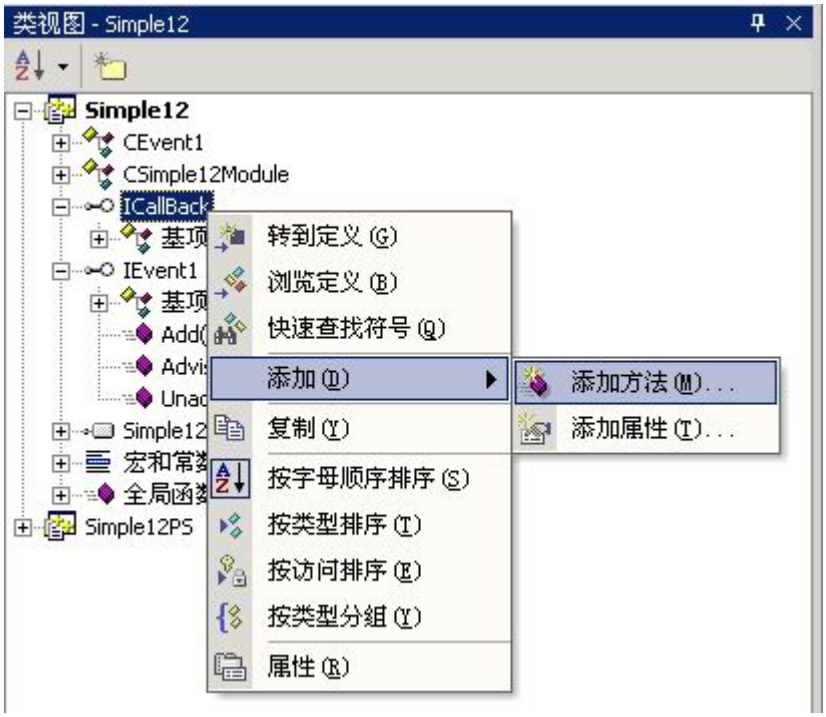
5、增加回调接口 ICallBack 的 IDL 定义。打开 IDL 文件并手工输入（黑体字部分为手工输入的），然后保存：

```
import "oaidl.idl";
import "ocidl.idl";
[
        object,
        uuid(DB72DF86-70E9-4ABC-B2F8-5E04062D3B2E),  // 这个 IID 可以用
GUDIGEN.EXE 产生
        helpstring("ICallBack 接口"),
        pointer_default(unique)
```

```
]
interface ICallBack : IUnknown
{

};

[
        object,    // 以下内容同示例程序,当然如果是你自己生成的程序就肯定有差别的啦
        uuid(DB72DF85-70E9-4ABC-B2F8-5E04062D3B2E),

        helpstring("IEvent1 Interface"),
        pointer_default(unique)
]
interface IEvent1 : IUnknown
{
        [helpstring("method Add")] HRESULT Add([in] long n1, [in] long n2);
       [helpstring("method Advise")] HRESULT Advise([in] ICallBack * pCallBack, [out]
long * pdwCookie);
        [helpstring("method Unadvise")] HRESULT Unadvise([in] long dwCookie);
};

[
        uuid(FBA1E0F0-49CD-4B77-B9B1-4DC066AF8A8E),
        version(1.0),
        helpstring("Simple12 1.0 类型库")
]
library SIMPLE11Lib
{
        importlib("stdole32.tlb");
        importlib("stdole2.tlb");

        [
                uuid(53E00126-B1A0-4510-B9BC-75ED87CE2DB7),
                helpstring("Event1 Class")
        ]
        coclass Event1
        {
                [default] interface IEvent1;
```

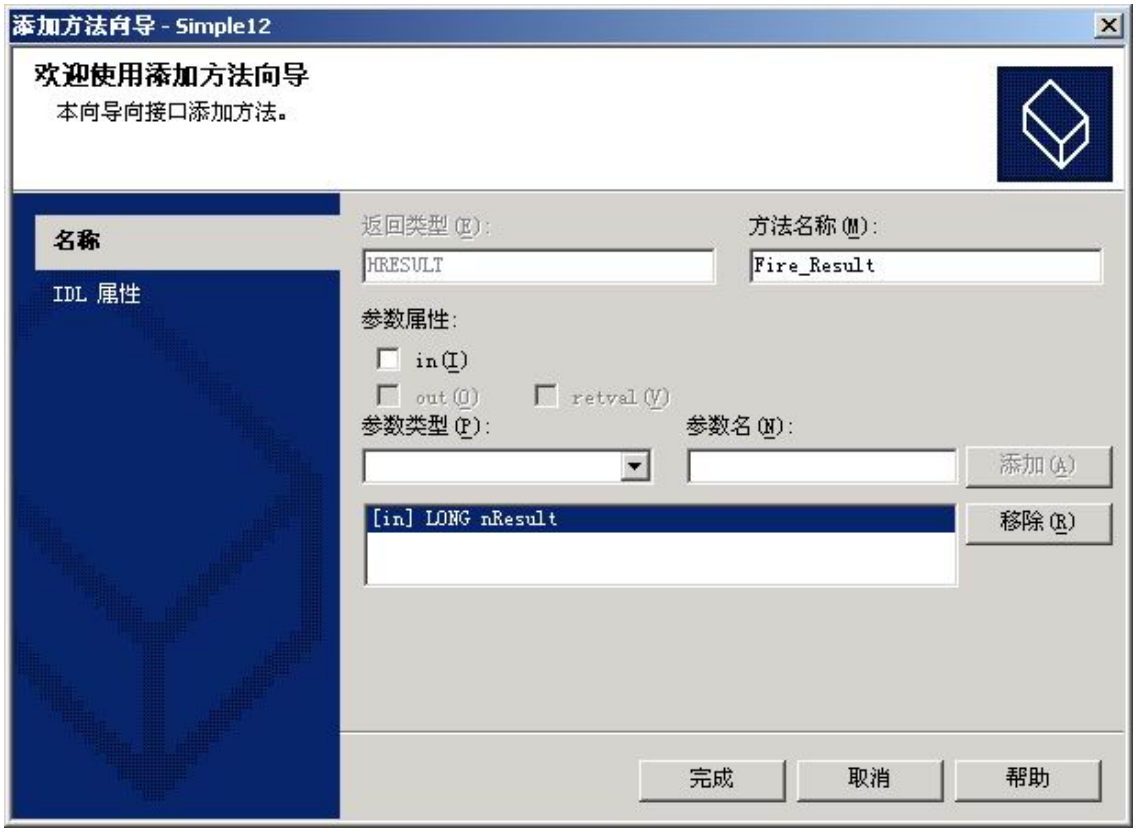
```
                // 需要手工输入，据说 VB 使用的话，不能有 [source,default] 属性
                [source, default] interface ICallBack;
        };
};
```

6、增加回调接口函数



图五、增加回调接口函数

其实和以前的方法一样，只要注意别选错了接口就好。



图六、增加接口函数 Fire_Result([in] long nResult)

我们计算整数和，得到结果后，就是要靠这个回调接口函数去反馈给客户端呀。

7、添加组件内部保存回调接口指针的数组

刚才已经说过，我们这个组件打算支持多个对象的回调连接，因此我们要使用一个数组来保存。由于 vc.net 无法用向导来添加数组形式的成员变量，我们还是打开 CEvent1 类的头文件，手工输入吧：

```
......
private:
        ICallBack * m_pCallBack[10];
......
```

保存一个数组可以有多种方式。示例程序比较简单，定义了一个 10 个元素空间的成员数组变量。如果你已经学会了使用 STL，那么你也可以用 vector 等容器来实现。**注意！注意！注意！在构造函数中别忘了初始化数组元素为 NULL**。

8、好了，下面开始完成所有代码

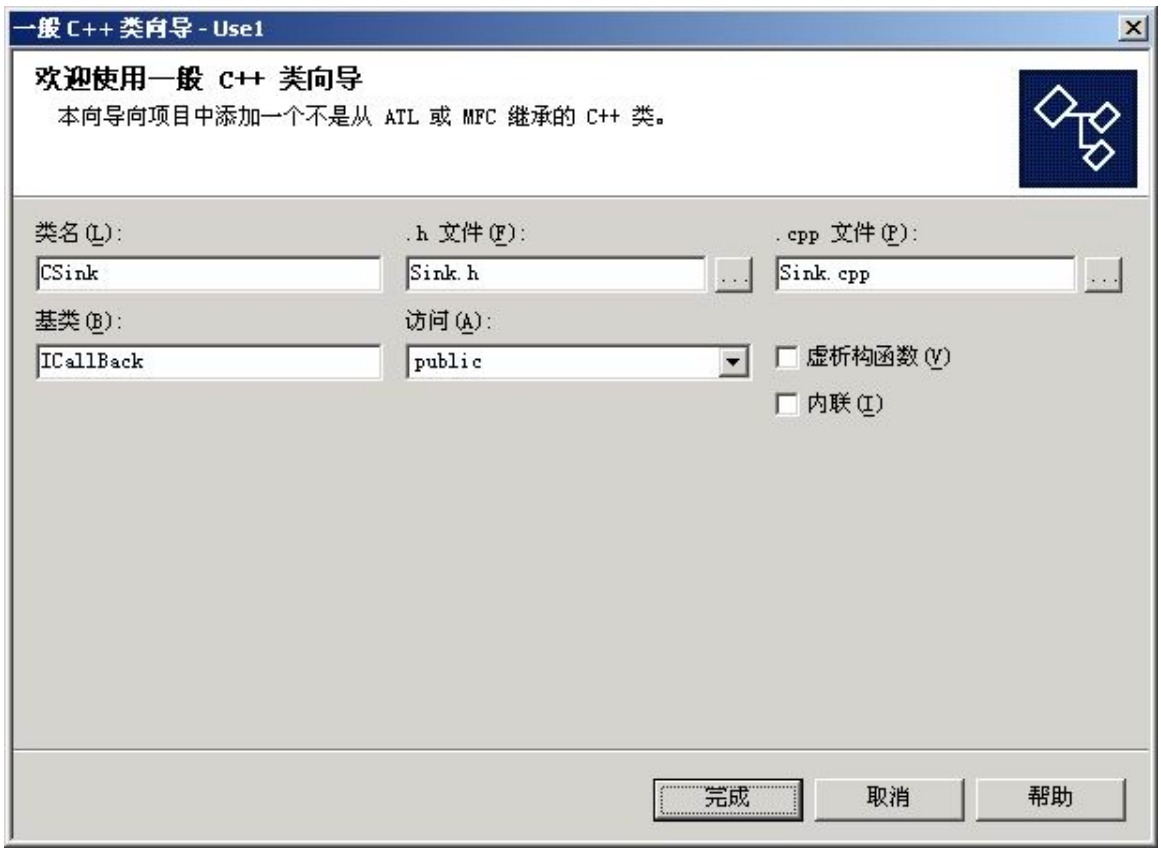
```
STDMETHODIMP CEvent1::Add(long n1, long n2)
{
```

```
        long nResult = n1 + n2;
        for( int i=0; i<10; i++)
        {
                if( m_pCallBack[i] )    // 如果回调接口有效
                        m_pCallBack[i]->Fire_Result( nResult );      // 则发出
事件/通知
        }

        return S_OK;
}

STDMETHODIMP CEvent1::Advise(ICallBack *pCallBack, long *pdwCookie)
{
        if( NULL == pCallBack )    // 居然给我一个空指针？！
                return E_INVALIDARG;

        for( int i=0; i<10; i++)   // 寻找一个保存该接口指针的位置
        {
                if( NULL == m_pCallBack[i] )        // 找到了
                {
                        m_pCallBack[i] = pCallBack;// 保存到数组中
                        m_pCallBack[i]->AddRef();  // 指针计数器 +1

                        *pdwCookie = i + 1;        // cookie 就是数组下标
                // +1 的目的是避免使用 0，因为 0 表示无效

                        return S_OK;
                }
        }
        return E_OUTOFMEMORY;      // 超过 10 个连接，内存不够用啦
}

STDMETHODIMP CEvent1::Unadvise(long dwCookie)
{
        if( dwCookie<1 || dwCookie>10 )     // 这是谁干的呀？乱给参数
                return E_INVALIDARG;
```

```
        if( NULL == m_pCallBack[ dwCookie - 1 ] )     // 参数错误,或该接口指针已经
无效了
                return E_INVALIDARG;

        m_pCallBack[ dwCookie -1 ]->Release();        // 指针计数器 -1
        m_pCallBack[ dwCookie -1 ] = NULL;            // 空出该下标的数组元素

        return S_OK;
}
```

四、客户端实现步骤

大家下载示例程序后，去浏览客户端的实现程序吧。这里我只说明一下关于接收器是如何构造的：



图七、从 ICallBack 派生接收器类 CSink

这里 ICallBack 是 COM 接口，因此 CSink 是不能事例化的，如果你去编译，会得到一坨一坨（注 3）的错误，报告说你没有实现 virtual 函数。然后，我们可以按照错误报告，去实现所有的虚函数：

```
// STDMETHODIMP 是宏，等价于 long __stdcall
```

```
STDMETHODIMP CSink::QueryInterface(const struct _GUID &iid,void ** ppv)
{
        *ppv=this;          // 不管想得到什么接口，其实都是对象本身
        return S_OK;
}

ULONG __stdcall CSink::AddRef(void)
{       return 1;}// 做个假的就可以，因为反正这个对象在程序结束前是不会退出的

ULONG __stdcall CSink::Release(void)
{       return 0;}// 做个假的就可以，因为反正这个对象在程序结束前是不会退出的

STDMETHODIMP CSink::raw_Fire_Result(long nResult)
{
        ... ...  // 把计算结果显示在窗口中
        return S_OK;
}
```

五、小结

COM 组件实现事件、通知这样的功能有两个基本方法。今天介绍的回调接口方式非常好，速度快、结构清晰、实现也不复杂；下回书介绍连接点方式(Support Connection Points)，连接点方法其实并不太好，速度慢（如果是远程 DCOM 方式，要谨慎选择它）、结构复杂、唯一的好处就是 ATL 对它进行了包装，所以实现起来反而比较简单。不介绍又不行，因为微软绝大数支持事件的组件都是用连接点实现的，咳.....讨厌的微软(注 4)。

注 1：本来设想多举几个例子，因此第一个叫 Event1，可写完后，感觉程序已经比较复杂了，就没继续再做了。

注 2：当然，你选择使用双接口 Dual 也没有问题。但要注意到在下面的步骤，增加回调接口修改 IDL 文件的时候,我们是要使用 Custom(从 IUnknown 派生，而不是从 IDispatch 派生)的。

注 3：一坨一坨经常用来形容一堆一堆的狗屎。

注 4：微软的同志们，玩笑话不要当真呀！我还靠着你来吃饭那。