COM：可连接对象 & 结构化存储

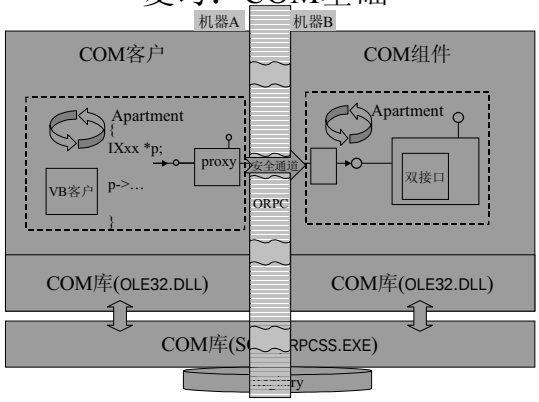
潘爱民

http://www.icst.pku.edu.cn/compcourse

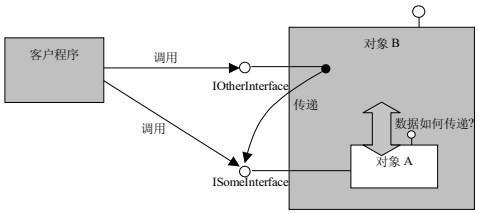
内容

- 复习：COM基础
- 可连接对象
- 结构化存储

复习：COM基础



聚合模型的关键



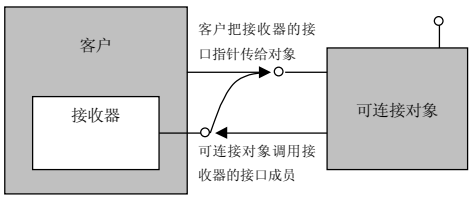
可连接对象(connectable object)

内容：

- 可连接对象结构模型
- 实现可连接对象(源对象)
- 客户-源对象-接收器的协作过程
- 可连接对象的程序实现

双向通信机制

（客户与可连接对象的关系）



两个概念

- 入接口(incoming interface)
  - 组件对象实现入接口，客户通过入接口调用对象提供的功能
  - 客户和组件都需要知道接口的类型信息
- 出接口(outgoing interface)
  - 客户端提供的COM对象实现出接口
  - 组件端的对象通过出接口调用客户提供的功能
  - 组件提供接口类型信息，客户实现该接口
  - 类似于回调(callback)，但是要复杂和灵活得多

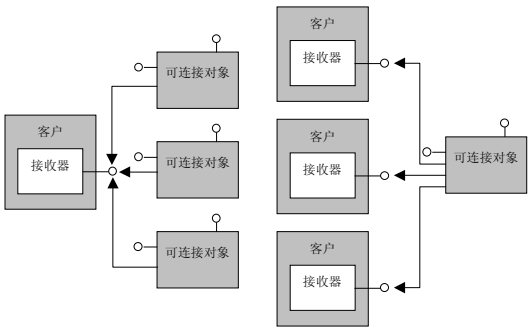
出接口

- 类型信息由组件一方提供
- 客户提供出接口的实现，实现出接口的COM对象被称为接收器对象(sink)
  - sink没有CLSID，也不需要类厂
- 也是一个COM接口，有IID
- 每个成员函数代表了：
  - 事件event
  - 通知notification
  - 请求request

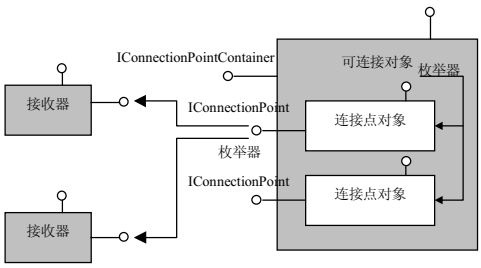
源对象 or 可连接对象

- Connectable object，source
- 普通的COM对象，支持一个或者多个出接口
- 提供出接口的类型信息
  - 通过IProvideClassInfo[2]接口
  - 通过typelib

客户与可连接对象之间的两种结构



可连接对象的基本结构



可连接对象

- 如何管理多个出接口
  - 每个出接口对应一个连接点对象
  - 通过连接点枚举器管理
- 对于每个出接口，如何管理多个客户连接
  - 通过连接枚举器管理多个连接

实现可连接对象(源对象)(一)

- 枚举器
  - 内部对象，不需要类厂和CLSID
  - 枚举器接口模板

```
class IEnum<ELT_T> : public IUnknown
{
    virtual HRESULT Next( ULONG celt, ELT_T *rgelt,
                          ULONG *pceltFetched ) = 0;
    virtual HRESULT Skip( ULONG celt ) = 0;
    virtual HRESULT Reset( void ) = 0;
    virtual HRESULT Clone( IEnum<ELT_T>**ppenum ) = 0;
};
```

枚举器的用法

```
class IStringManager : public IUnknown {
    virtual IEnumString* EnumStrings(void) = 0;
};

void SomeFunc(IStringManager * pStringMan)
{
  String  psz;
  IEnumString * penum;
  penum=pStringMan->EnumStrings();
  while (S_OK==penum->Next(1, &psz, NULL))
  {
      … //Do something with the string in psz and free it
  }
  penum->Release();
  return;
}
```

实现可连接对象(源对象)(二)

- IConnectionPointContainer接口

```
class IConnectionPointContainer : public IUnknown
{
    virtual HRESULT EnumConnectionPoints(IEnumConnectionPoints **) = 0;
    virtual HRESULT FindConnectionPoint(const IID *, IConnectionPoint **) = 0;
};
```

- IEnumConnectionPoints接口

```
class IEnumConnectionPoints : public IUnknown
{
    virtual HRESULT Next( ULONG cConnections,  IConnectionPoint **rgpcn,
                          ULONG *pcFetched) = 0;
    virtual HRESULT Skip( ULONG cConnections) = 0;
    virtual HRESULT Reset(void) = 0;
    virtual HRESULT Clone( IEnumConnectionPoints **ppEnum) = 0;
};
```

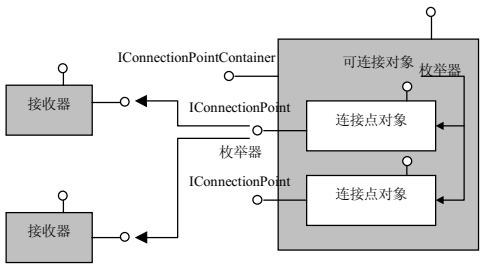
实现可连接对象(源对象)(三)

- 连接点和IConnectionPoint接口

```
class IConnectionPoint : public IUnknown
{
    virtual HRESULT GetConnectionInterface( IID *pIID) = 0;
    virtual HRESULT GetConnectionPointContainer(
                        IConnectionPointContainer **ppCPC) = 0;
    virtual HRESULT Advise( IUnknown *pUnk, DWORD *pdwCookie) = 0;
    virtual HRESULT Unadvise( DWORD dwCookie) = 0;
    virtual HRESULT EnumConnections( IEnumConnections **ppEnum) = 0;
};
```

- 连接枚举器 —— 实现IEnumConnections接口
  - 允许多个客户连接
  - 每个连接用struct CONNECTDATA来描述

回顾：可连接对象的基本结构



客户与源对象建立连接过程

- （1）客户请求IConnectionPointContainer接口；
- （2）客户调用IConnectionPointContainer::FindConnectionPoint找到连接点对象
- （3）客户调用IConnectionPoint::Advise建立与接收器的连接
- （4）最后，客户调用IConnectionPoint::Unadvise取消连接，并释放连接点对象

客户方基本结构

- 客户方实现接收器对象(sink)
  - 支持多个与可连接对象之间的连接
  - 一般只实现专用的出接口（IUnknown除外）
  - 不需要类厂、CLSID
  - 与客户代码紧密连接起来
- 建立连接
  - 1 通过IConnectionPointContainer接口找到连接点对象
  - 2 通过连接点对象建立连接
  - 连接点相当于连接管理器

接收器的实现

```
class CSomeEventSet : public ISomeEventSet
{
  private:
    ULONG      m_cRef;    // Reference count
        ......   // other private data members
  public:
    DWORD      m_dwCookie;  // Connection key
  public:
    CSomeEventSet ();
    ~CSomeEventSet(void);

    //IUnknown members
    STDMETHODIMP  QueryInterface(REFIID, PPVOID);
    STDMETHODIMP_(DWORD) AddRef(void);
    STDMETHODIMP_(DWORD) Release(void);

    STDMETHODIMP  SomeEventFunction ( ... );
    ......
};
```

接收器的用法

```
ISomeEventSet *gpSomeEventSet;
.......
// Initialize
CSomeEventSet *pSink = new CSomeEventSet;
pSink->QueryInterface(IID_ISomeEventSet, pSomeEventSet ); // Reference count is 1
.......

// connections the sink object to the connectable object we have
hr=pConnectionPoint->Advise(pSomeEventSet , & pSomeEventSet->m_dwCookie);
.....

// disconnections the sink object from the connectable object we have
hr=pConnectionPoint->Unadvise( pSomeEventSet->m_dwCookie);
.......
// Uninitialize
pSink->Release( ); // Reference count is 0
```

事件的激发和处理

```
BOOL CSourceObject::FireSomeEvent(IConnctionPoint *pConnectionPoint)
{
  IEnumConnections   *pEnum;
  CONNECTDATA     connectionData;

  if (FAILED(pConnectionPoint->EnumConnections(&pEnum)))
        return FALSE;

  while (pEnum->Next(1, & connectionData, NULL) == NOERROR)     {
        ISomeEventSet  *pSomeEventSet;
        if (SUCCEEDED(connectionData.pUnk->QueryInterface(IID_ISomeEventSet,
                  (PPVOID)& pSomeEventSet)))     {
          pSomeEventSet->SomeEventFunction();   // Trigger event or request
          pSomeEventSet->Release();
        }
  }
  pEnum->Release();    return TRUE;
}
```

与出接口有关的类型信息

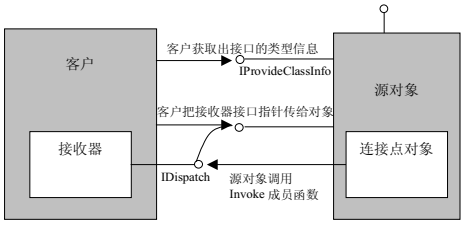
- 客户如何知道出接口？运行时刻？编译时刻？
- 动态构造接收器对象？动态构造vtable？支持部分成员？
- 类型信息的协商
- 能否用标准的接口作为出接口？

用IDispatch接口作为出接口(一)

- IDispatch接口

```
class IDispatch : public IUnknown
{
  public:
    virtual HRESULT GetTypeInfoCount( UINT *pctinfo) = 0;
    virtual HRESULT GetTypeInfo( UINT iTInfo, LCID lcid, ITypeInfo **ppTInfo) = 0;
    virtual HRESULT GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames,
        UINT cNames, LCID lcid, DISPID *rgDispId) = 0;
    virtual HRESULT Invoke( DISPID dispIdMember, REFIID riid, LCID lcid,
        WORD wFlags, DISPPARAMS *pDispParams,
        VARIANT *pVarResult, EXCEPINFO *pExcepInfo, UINT *puArgErr) = 0;
};
```

用IDispatch接口作为出接口(二)



IDispatch出接口的事件激发函数

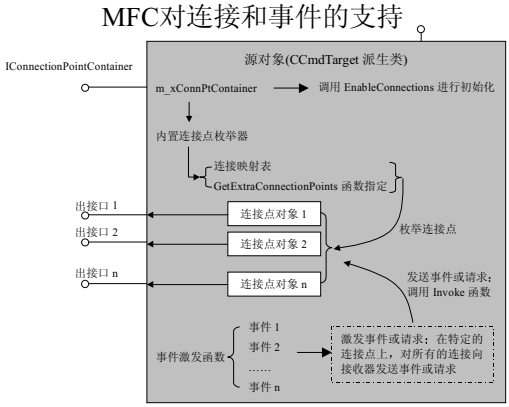
```
void CMySourceObj::FireMyMethod (short nInt)
{
    COleDispatchDriver driver;

    POSITION pos = m_xMyEventSet.GetStartPosition();
    LPDISPATCH pDispatch;
    while (pos != NULL) {
        pDispatch = (LPDISPATCH) m_xMyEventSet.GetNextConnection(pos);
        ASSERT(pDispatch != NULL);
        driver.AttachDispatch(pDispatch, FALSE);
        TRY
            driver.InvokeHelper(eventidMyMethod, DISPATCH_METHOD,
                                VT_EMPTY, NULL,
                                (BYTE *) (VTS_I2), nInt);
        END_TRY
        driver.DetachDispatch();
    }
}
```

用连接点机制实现回调的讨论

- 比传统的回调函数
  - 功能强大，灵活
  - 可以跨进程、跨机器
- Tightly coupled *vs* loosely coupled (COM+)
  - 要求客户与组件同步
  - 没有第三方的参与，所以双方必须保持共识

MFC对连接和事件的支持



用MFC实现源对象

- 创建工程——支持COM
- 定义出接口——编辑.odl文件
- 利用MFC宏加入连接点声明以及连接点对象的定义
- 在对象构造函数中调用EnableConnections();
- 在接口映射表中加入接口IConnectionPointContainer的表项，再加入连接映射表
- 定义连接点类的虚函数(至少为GetIID)
- 加入事件激发函数

用MFC在客户程序中实现接收器

- 初始化 —— AfxOleInit
- 定义出接口成员类
- 实现出接口成员类
- 创建源对象
- 建立连接和取消连接
- 完成可触发事件的动作

用MFC实现的例子



ATL实现可连接对象

- 在IDL中
  - 定义一个用作出接口的automation接口
  - 在coclass中加入出接口，含source属性
- 增加IConnectionPointContainer接口
  - 在基类列表中增加
  - IConnectionPointConntainerImpl<CMyClass>
  - 在COM MAP中加入
  - COM_INTERFACE_ENTRY(IConnectionPointConnta iner)

模板类IConnectionPointImpl

- CMyClass继承IConnectionPointImpl一次或多次
  - IConnectionPointImpl实现了独立的引用计数
  - 用法：在基类列表中增加
  - IConnectionPointImpl<CMyClass, &DIID__IEventSet>
- 加入connection point map，如下

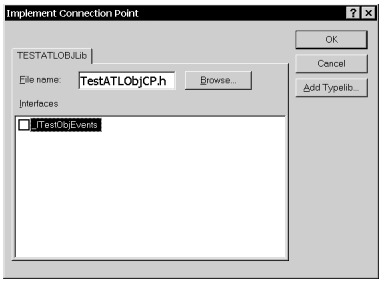
```
BEGIN_CONNECTION_POINT_MAP(CMyClass)
   CONNECTION_POINT_ENTRY(DIID__IEventSet)
END_CONNECTION_POINT_MAP()
```

激发事件辅助函数

- 手工激发事件
  - IConnectionPointImpl包含一个m_vec成员，内含所有已经建立的接收器连接
  - 遍历m_vec数组，逐一调用Invoke函数
- 利用VC IDE提供的源码产生工具
  - ATL连接点代理生成器，启动对话框Implement Connection Point
  - 产生名为CProxy<SinkInterfaceName>的模板类
    - 例如CProxy_IEventSet，它从IConnectionPointImpl派生
    - 对于每一个事件或者请求，都有一个对应的Fire_Xxx成员函数
  - 用模板类代替IConnectionPointImpl基类

Implement Connection Point对话框

- 创建对象时选择Connection Point
- ClassView中，在对象类上右键点击选择此项功能



ATL实现连接点：最后的工作

- 在需要激发事件的地方
  - 调用CProxy<Xxxx>提供的辅助函数
- 增加对IProvideClassInfo2接口的支持
  - 需要typelib的支持
  - 加入基类IProvideClassInfo2Impl
  - 在COM MAP中加入：
  - COM_INTERFACE_ENTRY(IProvideClassInfo2)
  - COM_INTERFACE_ENTRY(IProvideClassInfo)

ATL实现接收器sink

- IDispEventSimpleImpl
  - 轻量，不需要typelib的支持
- IDispEventImpl
  - 需要typelib的支持
- Event Sink Map

```
BEGIN_SINK_MAP(CMyCLass)
   SINK_ENTRY_EX(...)// 适合用于non-UI object
   SINK_ENTRY(...)      // 适合用于UI object
END_SINK_MAP
```

ATL：建立sink和source之间的连接

- IDispEventSimpleImpl成员
  - DispEventAdvise
  - DispEventUnadvise
- AtlAdviseSinkMap
  - 建立sink与source缺省源接口的连接

结构化存储(structured storage)

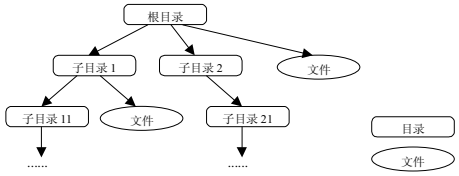
内容：

- 结构化存储模型
- 复合文档
- 永久对象

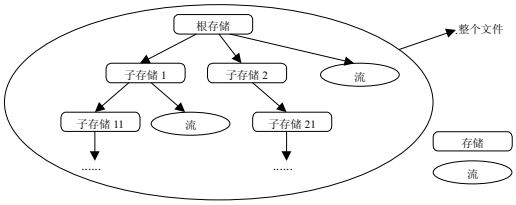
问题的由来

- 文件系统的诞生
  - 多个应用程序共享同一个存储设备
  - 文件服务功能的抽象
- 进展到结构化存储
  - 多个组件共享同一个文件
  - 组件软件存储功能的基本要求
  - OLE的需求
  - 组件共享句柄方案，如何定位？避免冲突？

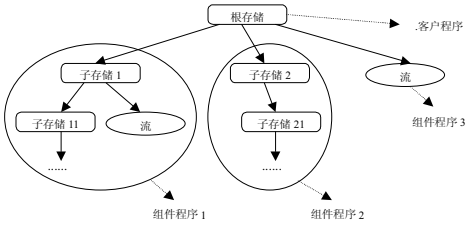
文件系统结构



结构化存储



多个组件程序共享一个复合文件



复合文件

- 文件内部的文件系统
- 只有两种对象：存储对象和流对象
- 实现了部分访问和增量访问的功能

流对象

- COM库提供实现，实现了IStream接口

```
class IStream : public IUnknown
{
public :
    virtual HRESULT Read (void *pv, unsigned long cb, unsigned long *pcbRead) = 0;
    virtual HRESULT Write (void *pv, unsigned long cb, unsigned long *pcbWritten) = 0;
    virtual HRESULT Seek (LARGE_INTEGER dlibMove, unsigned long dwOrigin,
                          ULARGE_INTEGER *plibNewPosition) = 0;
    virtual HRESULT SetSize (ULARGE_INTEGER libNewSize) = 0;
    virtual HRESULT CopyTo (LPSTREAM pStm, ULARGE_INTEGER cb,
                          ULARGE_INTEGER *pcbRead, ULARGE_INTEGER *pcbWritten) = 0;
    virtual HRESULT Commit (unsigned long dwCommitFlags) = 0;
    virtual HRESULT Revert ()= 0;
    virtual HRESULT LockRegion (ULARGE_INTEGER libOffset, ULARGE_INTEGER cb,
                          unsigned long dwLockType) = 0;
    virtual HRESULT UnlockRegion (ULARGE_INTEGER libOffset, ULARGE_INTEGER cb,
                          unsigned long dwLockType) = 0;
    virtual HRESULT Stat (STATSTG *pStatStg, unsigned long grfStatFlag) = 0;
    virtual HRESULT Clone(LPSTREAM * ppStm) = 0;
};
```

存储对象

- COM库提供实现，实现了IStorage接口

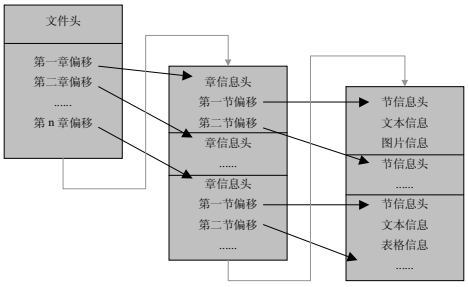
```
class IStorage : public IUnknown
{
    virtual HRESULT CreateStream (const WCHAR * , unsigned long , LPSTREAM * ) = 0;
    virtual HRESULT OpenStream (const WCHAR * , unsigned long , LPSTREAM * ) = 0;
    virtual HRESULT CreateStorage (const WCHAR * , unsigned long ,LPSTORAGE * ) = 0;
    virtual HRESULT OpenStorage (const WCHAR* , LPSTORAGE  *,
                          unsigned long , SNB , unsigned long , LPSTORAGE * ) = 0;
    virtual HRESULT CopyTo (unsigned long , IID const *, SNB snbExclude, LPSTORAGE * pStgDest) = 0;
    virtual HRESULT MoveElementTo (const WCHAR * , LPSTORAGE  *,char const * , unsigned long ) = 0;
    virtual HRESULT Commit (unsigned long ) = 0;
    virtual HRESULT Revert ()= 0;
    virtual HRESULT EnumElements (unsigned long , void *,unsigned long , LPENUMSTATSTG * ) = 0;
    virtual HRESULT DestroyElement (const WCHAR * pwcsName) = 0;
    virtual HRESULT RenameElement (const WCHAR * pwcsOldName,  const WCHAR * pwcsNewName) = 0;
    virtual HRESULT SetElementTimes(const WCHAR *,FILETIME const *,FILETIME const*,
                          FILETIME const *) = 0;
    virtual HRESULT SetClass (REFCLSID rclsid) = 0;
    virtual HRESULT SetStateBits (unsigned long grfStateBits, unsigned long grfMask) = 0;
    virtual HRESULT Stat (STATSTG *pStatStg, unsigned long grfStatFlag) = 0;
};
```

客户如何获取存储对象和流对象

- 如何得到指向根存储对象的接口指针？
- CreateStorage和OpenStorage成员函数得到一个子存储对象，是唯一的途径
- CreateStream和OpenStream成员函数得到一个流对象，也是唯一的途径

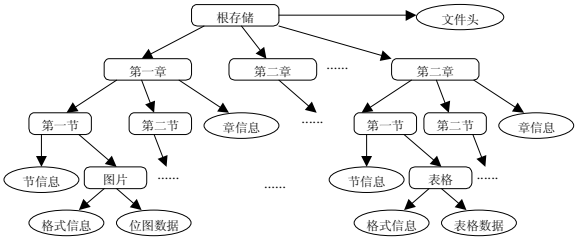
用结构化存储设计应用(一)

- 用普通文件组织的文档结构



用结构化存储设计应用(二)

- 复合文件格式的文档结构



结构化存储特性——访问模式

- STGM_CREATE
- STGM_CONVERT
- STGM_FAILIFTHERE
- STGM_DELETEONRELEASE
- STGM_DIRECT
- STGM_TRANSACTED
- STGM_PRIORITY
- STGM_READ
- STGM_WRITE
- STGM_READWRITE
- STGM_SHARE_DENY_READ
- STGM_SHARE_DENY_WRITE
- STGM_SHARE_EXCLUSIVE
- STGM_SHARE_DENY_NONE

结构化存储特性——事务机制

- 数据一致性和完整性
- 操作：Commit、Revert
- 事务嵌套：以STGM_TRANSACTED标志为基础
- 事务机制需要消耗较多系统资源
- Commit参数：
  - STGC_DEFAULT
  - STGC_OVERWRITE
  - STGC_ONLYIFCURRENT
  - STGC_DANGEROUSLYCOMMITMERELYTODISKCACHE

结构化存储特性——命名规则

- 根存储对象的名字遵守文件系统的命名约定
- 长度不超过32个字符。
- 首字符使用大于32的字符，小于32的字符作为首字符有特殊意义
- 不能使用字符“\”、“/”、“:”和“!”
- 名字“.”和“..”被保留
- 名字保留大小写，但比较操作大小写无关

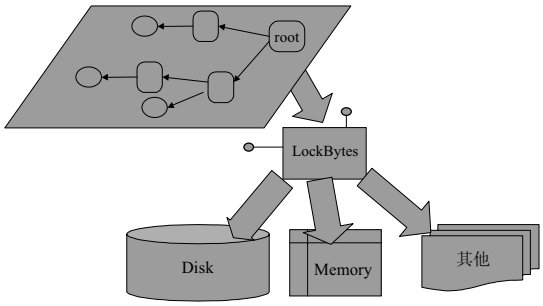
结构化存储特性——增量访问

- 减少保存和打开文件的时间
- 降低了应用程序对系统资源的要求
- 问题：
  - 通过根存储逐层找到目标对象
  - 空间回收

复合文档

- 结构化存储的具体实现
- 底层机制：LockBytes对象
  - 把存储介质描述成一般化的字节序列
- 复合文档API函数
- 零内存保存特性

复合文档模型



LockBytes对象

- ILockBytes接口

```
class ILockBytes : public IUnknown
{
public :
    virtual HRESULT ReadAt (ULARGE_INTEGER , VOID *pv, unsigned long ,
                unsigned long *) = 0;
    virtual HRESULT WriteAt (ULARGE_INTEGER , VOID *pv, unsigned long ,
                unsigned long *) = 0;
    virtual HRESULT Flush ()= 0;
    virtual HRESULT SetSize (ULARGE_INTEGER cb) = 0;
    virtual HRESULT LockRegion (ULARGE_INTEGER , ULARGE_INTEGER ,
                unsigned long ) = 0;
    virtual HRESULT UnlockRegion (ULARGE_INTEGER , ULARGE_INTEGER ,
                unsigned long ) = 0;
    virtual HRESULT Stat (STATSTG *, unsigned long ) = 0;
};
```

复合文档API函数

- 创建复合文档的API函数
  - StgCreateDocfile、StgCreateDocfileOnILockBytes
- 打开复合文档的API函数
  - StgOpenStorage、StgOpenStorageOnILockBytes
- 与内存句柄有关的一组操作函数
  - CreateILockBytesOnHGlobal、GetHGlobalFromILockBytes
  - CreateStreamOnHGlobal、GetHGlobalFromStream
- 其他

零内存保存特性

- 意义：资源耗尽之后，保留修改信息
- 资源预留，对于所有的流对象和存储对象
- “Save”操作，只要调用Commit函数即可
- “Save As”操作，利用根存储对象上的IRootStorage接口，调用SwitchToFile成员函数，再调用Commit函数即可。

与CLSID的联系

- IStorage::SetClass函数把存储对象与CLSID联系起来
- GetClassFile函数，从文件到CLSID：

  复合文件，直接得到根存储的CLSID

  非复合文件：

  (1) 文件扩展名-〉ProgID-〉CLSID

  (2) HKEY_CLASSES_ROOT\FileType键提供了匹配规则：

```
HKEY_CLASSES_ROOT
  FileType
    {<clsid >}
      <type id> = <offset>,<cb>,<mask>,<value>
      <type id> = <offset>,<cb>,<mask>,<value>
```

复合文档与COM的关系

- 复合文档技术以COM为基础
- 应用程序在处理复合文档时
  - 把storage或stream直接交给COM组件来处理
  - COM组件接受storage或stream作为数据存储
  - 多个组件协同处理同一个文件
- ->永久对象

永久对象

- 永久对象
  - 实现了IPersist*XXX*接口的COM对象
- 永久接口：
  - class IPersist : public IUnknown
  - class IPersistStream : public IPersist
  - class IPersistStreamInit : public IPersist
  - class IPersistFile : public IPersist
  - class IPersistStorage : public Ipersist
- 永久接口的成员函数：
  - GetClassID、IsDirty、Load和Save，……
- 永久对象可以实现多个永久接口，但使用时要保持一致性

永久对象用法

- 永久对象与结构化存储模型结合
- 永久对象例子
  - 用MFC实现的COM对象
  - 功能：永久状态为一段文本，使用永久接口对文本维护
  - 实现了IPersistStream和一个自动化接口

复合文档例子



复合文档查看工具

