COM特性

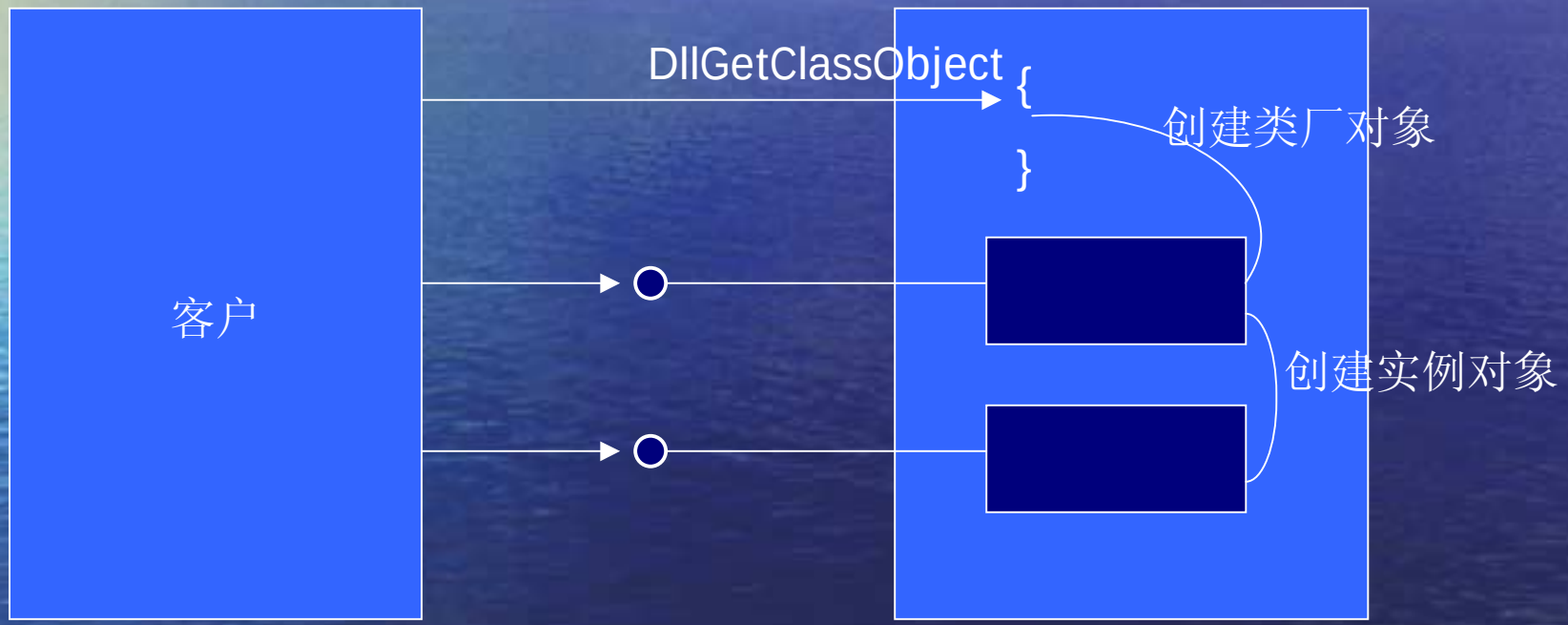
潘爱民

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

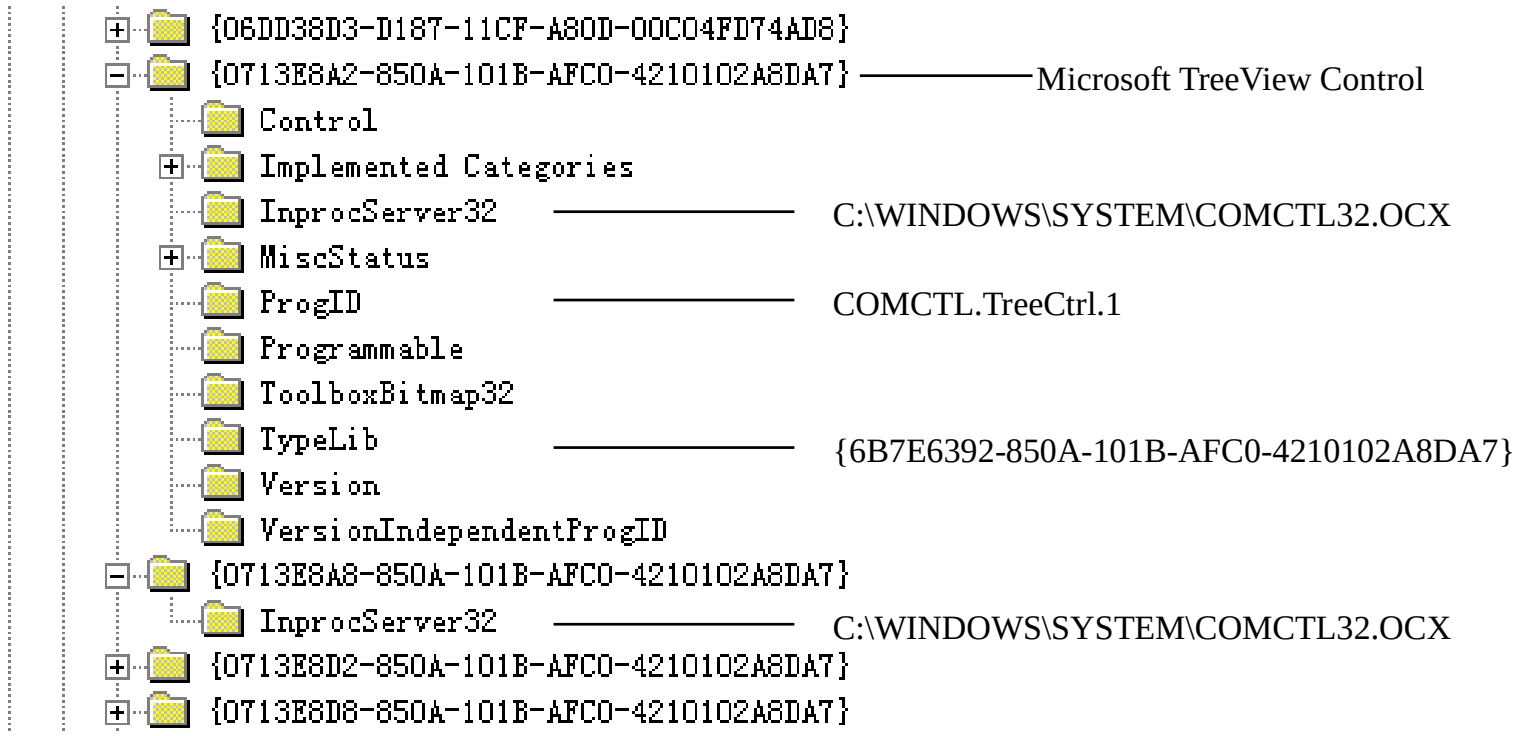
内容

- 复习：COM实现
- COM可重用模型 √
- COM跨进程模型 √
- COM线程模型
- COM安全性

复习：对象创建过程



复习：TreeView组件的注册信息

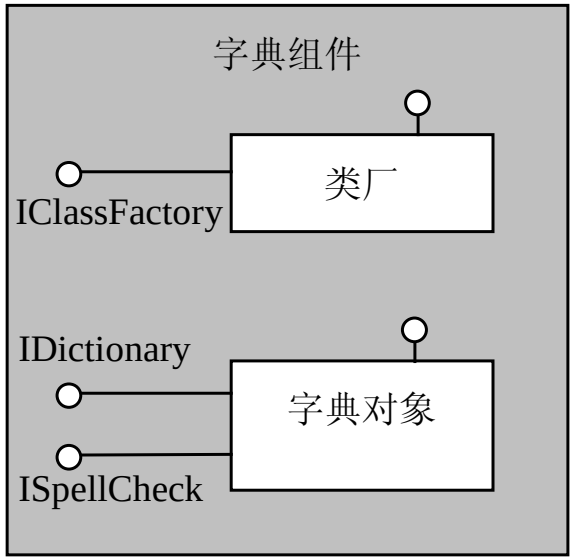


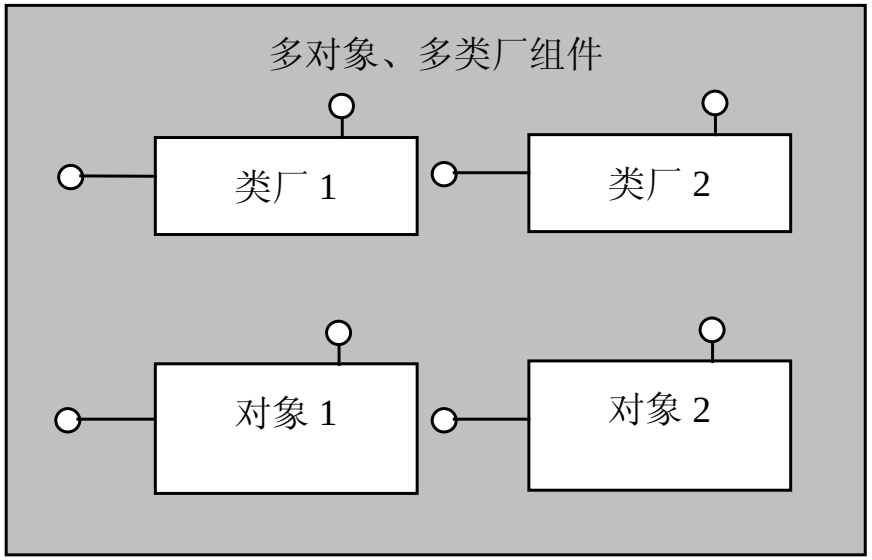
复习：类厂(Class Factory)

- 类厂：用于创建COM对象的COM对象
- 目标：完成COM对象的创建过程，更好地把客户与对象隔离开来。
- 特殊性：
  - 实现一个或多个创建接口，缺省的接口为IClassFactory
  - 类厂本身没有CLSID
  - 但是类厂的引用计数不参与对组件生命周期的控制，如果需要，可以使用锁计数功能

复习：类厂(续)

- 类厂与COM对象有一一对应关系





复习：COM创建函数

- CoGetClassObject
- CoCreateInstance
- CoCreateInstanceEx
- 注意：对于DLL组件，创建函数调用DllGetClassObject引出函数创建类厂，再由类厂创建COM对象

复习：COM库

- COM库的初始化
- COM库的内存管理
- 组件程序的装载和卸载
- 常用函数和HRESULT
  - GUID管理
  - 创建函数
  - 初始化函数
  - 内存管理函数
  - 字符串使用OLECHAR类型

复习：实现一个进程内COM组件的步骤

- 定义必要的CLSID和IID
- 实现COM对象
  - 通过QueryInterface暴露其接口
  - 管理引用计数，注意对全局引用计数的维护
- 实现类厂对象
  - 对象的引用计数不记在全局对象引用计数内
  - 维护锁计数
- 实现DllGetClassObject、DllCanUnloadNow
- (可选)实现两个注册函数

可重用性（reuse，复用）

- 重用性：当一个程序单元能够对其他的程序单元提供功能服务时，尽可能地重用原先程序单元的代码，既可以在源代码一级重用，也可以在可执行代码一级重用。
- COM重用性：因为COM是建立在二进制代码基础上的标准，所以其重用性也必然建立于二进制代码上。
- COM重用模型：包容和聚合
- 真正的重用：是实现重用而不是代码重用

C++类的重用模型

- 前提：假设有一个基类COldClass
- 目标：实现一个新类CNewClass，要求重用COldClass中的功能，而不是重新实现
- 做法：两种模型
  - 让CNewClass从COldClass派生，即

    class CNewClass: public COldClass {...};
  - 复合类或者嵌套类，在CNewClass中包含一个数据成员，其类型为COldClass

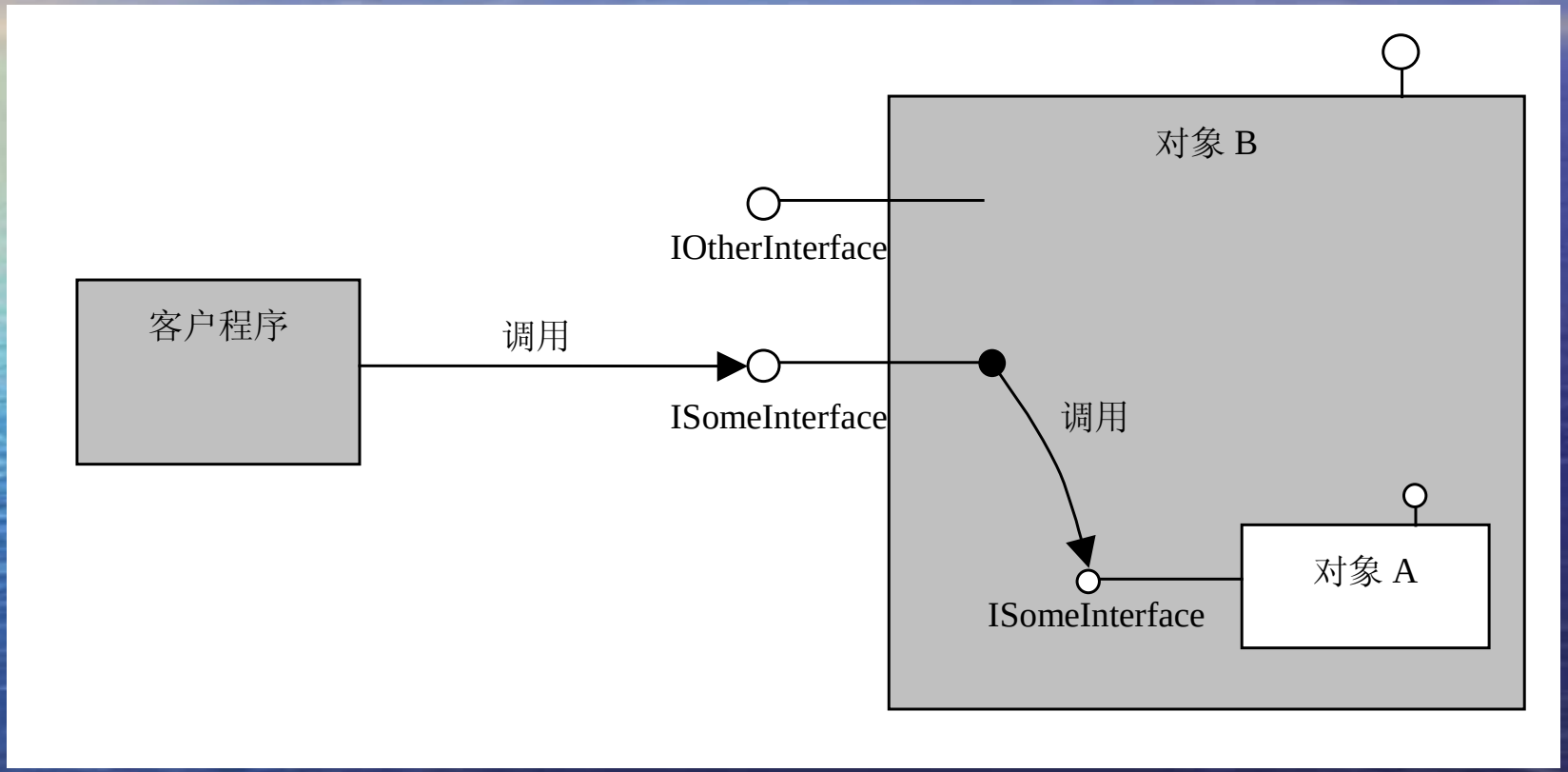
C++类的两种重用模型

- 继承模型，直接继承基类COldClass的所有方法和数据成员，“is-a”关系
  - 客户直接看到基类的接口(public成员)
- 复合模型，把基类的功能包含在内部，“has-a”关系
  - 客户看不到基类的信息

COM重用模型

- 前提：假设有一个COM对象A
- 目标：实现一个新对象B，要求重用对象A的功能，而不是重新实现原来已有的功能
- 做法：两种模型
  - 包容(containment)
  - 聚合(aggregation)

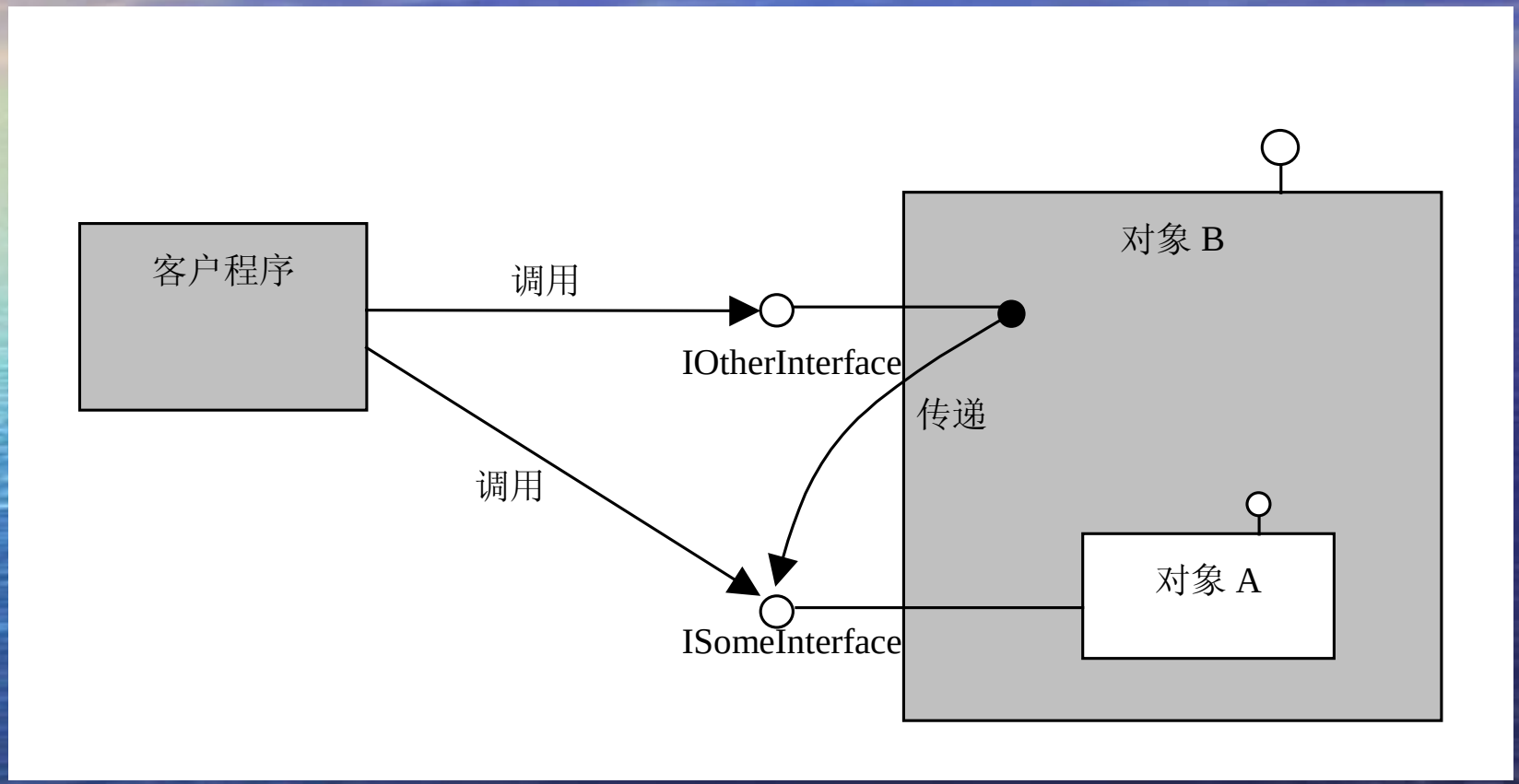
COM包容模型


对象 B
IOtherInterface
客户程序
调用
ISomeInterface
调用
对象 A
ISomeInterface

包容模型说明

- 外部对象包容内部对象的接口
- 外部对象的接口与内部对象的接口可以不同
- 包容模型的实质是客户-组件关系，在包容代码的前后可以插入其他的代码，甚至可以有条件地重用
- 客户看不到内部对象
- 内部对象的生存期包含在外部对象的生存期之内

聚合模型



聚合模型说明

- 聚合对象和被聚合对象协同工作
- 客户直接与内部对象交互，但它并不知道
- IUnknown唯一性是COM的基本要求
- 重用针对某个接口而言，聚合和包容并不矛盾，在一个对象上可以同时使用
- 聚合和包容的选择策略

包容模型实现

- 定义接口：

```
class ISomeInterface : public IUnknown
{
    public:
        virtual HRESULT __stdcall SomeFunction() = 0;
};

class IOtherInterface : public IUnknown
{
    public:
        virtual HRESULT __stdcall OtherFunction() = 0;
};
```

包容：外部对象定义

```
class CB : public ISomeInterface , public IOtherInterface
{
protected:
    ULONG          m_Ref;
public:
    CB ( );
    ~ CB ();
    HRESULT __stdcall  QueryInterface(const IID& iid, void **ppv);
    ULONG __stdcall  AddRef();
    ULONG __stdcall  Release();
     //ISomeInterface members
     HRESULT __stdcall SomeFunction( ) ;
     //IOtherInterface members
     HRESULT __stdcall OtherFunction( ) ;

      HRESULT Init();
     private :
            ISomeInterface *m_pSomeInterface;
};
```

包容：外部对象的实现

```
CB::CB ( )
{
    m_pSomeInterface = NULL;
    m_Ref = 0;
}
CB::~CB ( )
{
    if (m_pSomeInterface ! = NULL)
            m_pSomeInterface->Release() ;
}
HRESULT CB::Init()
{
    HRESULT result = ::CoCreateInstance(CLSID_ComponentA, NULL,
                    CLSCTX_INPROC_SERVER, IID_ISomeInterface,
                     (void **)&m_pSomeInterface) ;
    if (FAILED(result))       return E_FAIL;
    else                      return S_OK;
}
```

包容：外部对象的实现(续)

```
HRESULT __stdcall CB::SomeFunction( )
{
    return m_pSomeInterface->SomeFunction( );
}

HRESULT __stdcall CB::OtherFunction( )
{
        ……
}
```

包容：外部对象类厂的实现

```
HRESULT CBFactory::CreateInstance(IUnknown *pUnknownOuter,
                                  const IID& iid, void **ppv)
{
    CB  *pObj;      HRESULT hr;       *ppv=NULL;

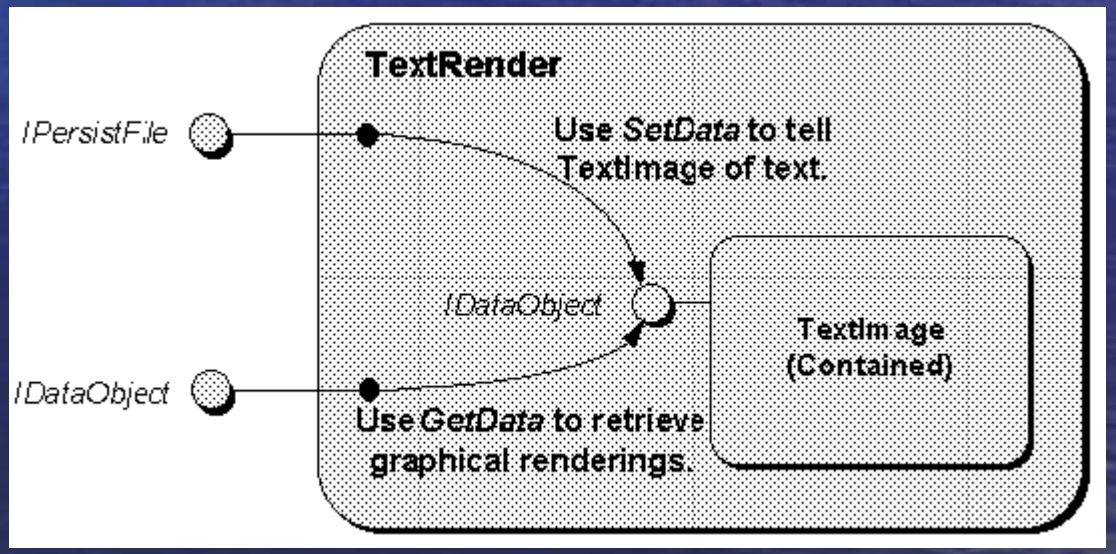
    if (pUnknownOuter != NULL)  return CLASS_E_NOAGGREGATION;
    pObj=new CB ();
    if (pObj == NULL)  return E_OUTOFMEMORY;

    hr = pObj->Init();
    if (FAILED(hr) ) {
            g_ObjectNumber --;
            delete pObj;
    }
    //Obtain the first interface pointer (which does an AddRef)
    hr=pObj->QueryInterface(iid, ppv);
    return hr;
}
```

包容：灵活应用包容模型

```
HRESULT __stdcall CB::SomeFunction( )
{
    if ( ... )
    {
        ......
        HRESULT result = m_pSomeInterface->SomeFunction( );
        ......
        return result;

    } else
    {
        ......
    }
}
```



聚合实现

- 要点：外部对象完全重用内部对象的接口
- 关键在于外部对象的QueryInterface函数
- 内部对象还能够把接口请求传回到外部对象来，所以内部对象必须要支持聚合
- 不管怎么样，我们总是要假定其他的组件是按接口一级实现引用计数的

预备知识：函数指针

- 有两个函数
  - int Func1(int x, int y){return x*y;}
  - int Func2(int x, int y){return x+y;}
- 定义函数指针
  - int (*MyFunc)(int, int);
- 代码段1
  - MyFunc = Func1;
  - int a = MyFunc(10, 20);
- 代码段2
  - MyFunc = Func2;
  - int b = MyFunc(10, 20);

- 要点：
  - 函数名字并不重要，函数指针才决定了函数的功能

预备知识：用vtable调用成员函数

- 有两个二进制结构一样的vtable

```
class Vtable1{
    virtual void __stdcall Method1(int, int) = 0;
    virtual void __stdcall Method2(int) = 0;
};
class Vtable2{
    virtual void __stdcall Operator1(int, int) = 0;
    virtual void __stdcall Operator2(int) = 0;
};
```

预备知识(续)

- 假设某段程序实现了其中之一

```
class CMyObject : public Vtable1{
.....
  virtual void __stdcall Method1(int, int);
  virtual void __stdcall Method2(int);
};
```

- 客户的用法

实例化CMyObject：

```
Vtable1 *p1 = CreateObject(...);
```

代码片断1：

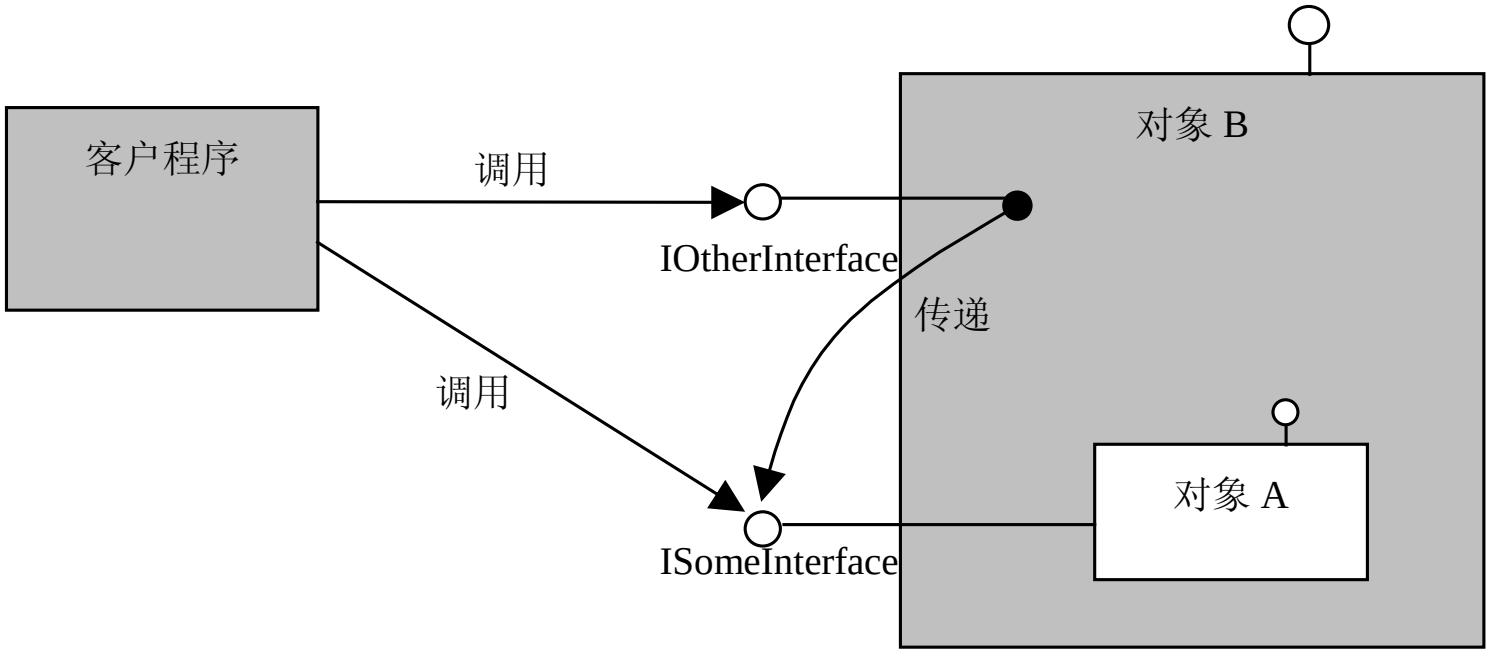
```
p1->Method1(a, b);
```

代码片断1：

```
Vtable2 *p2 = (Vtable2 *)p1;
p2->Operator1(a, b);
```

- 要点：
  - 指针类型并不重要，vtable才决定了内部方法的功能

回顾：聚合模型结构



外部对象的定义

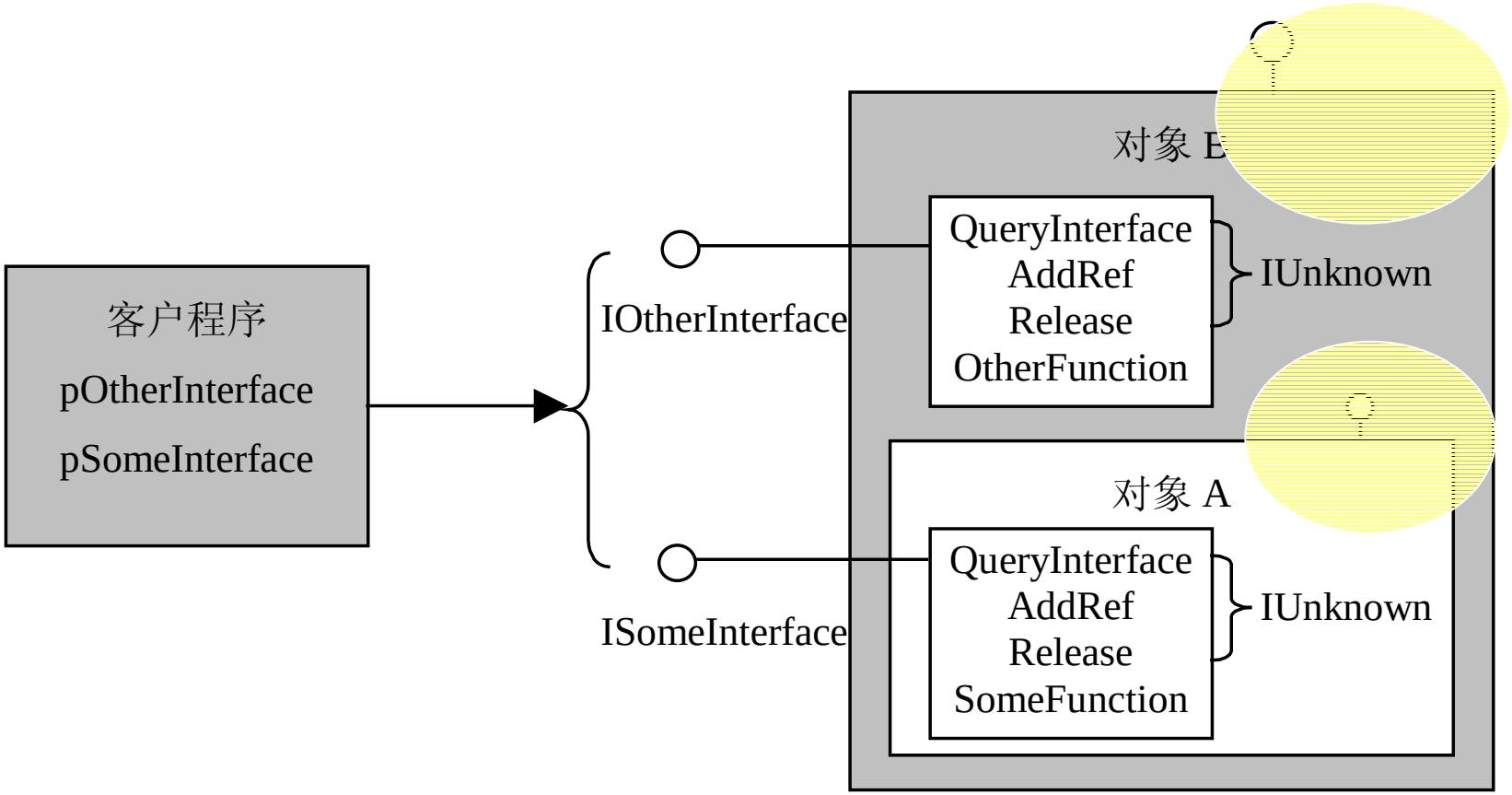
```
class CB : public IOtherInterface
{
protected:
    ULONG          m_Ref;
public:
    CB ( );
    ~ CB ();
    //IUnknown members
    HRESULT __stdcall  QueryInterface(const IID& iid, void **ppv);
    ULONG __stdcall  AddRef();
    ULONG __stdcall  Release();
     //IOtherInterface members
     HRESULT __stdcall OtherFunction( ) ;

    HRESULT Init();
private :
     IUnknown  *m_pUnknownInner;  // pointer to A's IUnknown
};
```

聚合：外部对象的QueryInterface

```
HRESULT CB::QueryInterface(const IID& iid, void **ppv)
{
    if ( iid == IID_IUnknown ) {
            *ppv = (IUnknown *) this ;
            ((IUnknown *)(*ppv))->AddRef() ;
    } else if ( iid == IID_OtherInterface )  {
            *ppv = (IOtherInterface *) this ;
            ((IOtherInterface *)(*ppv))->AddRef() ;
    } else if ( iid == IID_SomeInterface )  {
            return m_pUnknownInner->QueryInterface(iid, ppv) ;
    } else  {
            *ppv = NULL;
            return E_NOINTERFACE ;
    }
    return S_OK;
}
```

聚合：客户看到的接口示意图



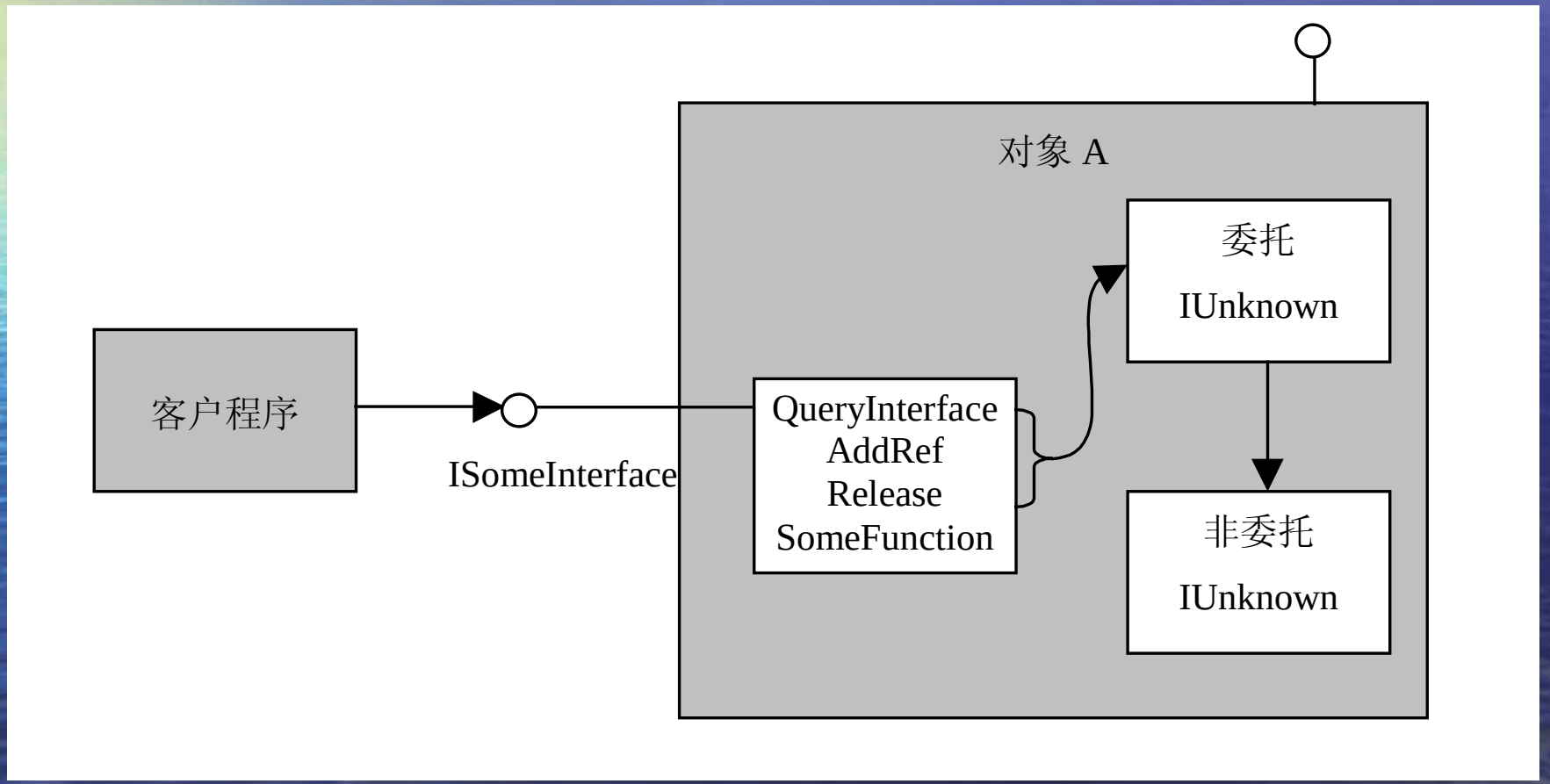
聚合模型中的内部对象

- 客户眼里的内部对象
  - 内部对象的AddRef和Release必须要作用在外部对象的引用计数上
  - 内部对象的QueryInterface必须要能够返回外部对象的接口
  - 解决的办法是，当内部对象知道自己被聚合之后，把IUnknown的方法调用委托给外部对象的IUnknown
- 外部对象眼里的内部对象
  - 外部对象必须有办法控制内部对象的生命周期，以及查询请求它的接口
  - 解决的办法是，内部对象有一个专门的IUnknown版本供外部对象使用，完成基本的生命周期控制功能和接口查询功能

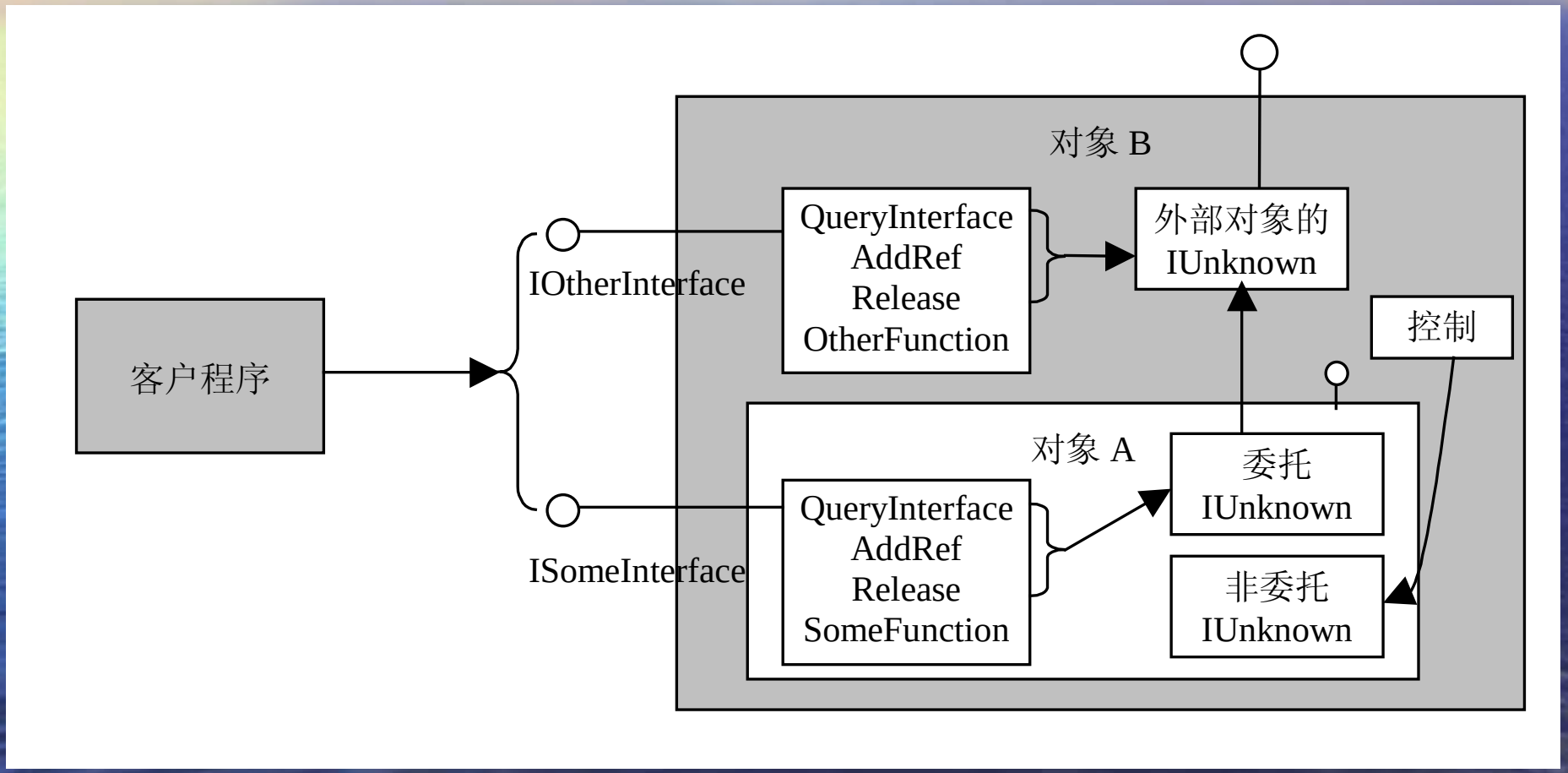
聚合：内部对象的实现方案

- 解决办法：内部对象实现两个IUnknown接口
  - 1. 非委托IUnknown接口用于正常处理引用计数和QI；
  - 2. 委托IUnknown接口按情况处理：
    - (1) 当对象被聚合时，调用外部对象的IUnknown接口；
    - (2) 当对象未被聚合时，调用非委托IUnknown接口

聚合：支持聚合的对象在非聚合方式下的接口示意图



聚合：支持聚合的对象在聚合方式下的接口示意图



聚合模型的要点

- 这些要点指导我们如何实现聚合对象
  - 创建内部对象的时候，外部对象必须把自己的IUnknown接口指针传给内部对象，被称为*controlling unknown*
  - 内部对象类厂的CreateInstance必须检查pUnkOuter参数，如果非NULL，则表明聚合，内部对象把指针保存起来，但不调用AddRef。若内部对象不支持聚合，则返回CLASS_E_NOAGGREGATION
  - 如果pUnkOuter 参数非NULL，并且外部对象请求IUnknown接口，则内部对象把自己的非委托版本的IUnknown传给外部对象

聚合模型的要点(续)

- 这些要点指导我们如何实现聚合对象
  - 如果内部对象本身又聚合了其他的对象，那么它必须把同样的pUnkOuter参数传递给它的内部对象
  - 当外部对象接到对于聚合接口的请求时，它必须调用非委托版本的IUnknown的QueryInterface函数，并把结果返回给客户
  - 对于除了非委托版本的IUnknown之外的接口，它的三个IUnknown调用必须全部委托给外部对象的pUnkOuter

聚合：内部对象如何获得外部对象的IUnknown接口

```
HRESULT CoCreateInstance(const CLSID& clsid,
    IUnknown *pUnknownOuter,
    DWORD dwClsContext,
    const IID& iid,
    (void **)ppv
);
HRESULT IClassFactory::CreateInstance(
                            IUnknown *pUnknownOuter,
                            const IID& iid,
                            void **ppv
);
```

聚合：定义非聚合IUnknown接口

- 因为C++类不能同时继承实现两个IUnknown，所以为非委托接口定义一个新的类

```
class INondelegationUnknown
{
    public:
        virtual HRESULT  __stdcall  NondelegationQueryInterface(
                                    const IID& iid, void **ppv) = 0 ;
        virtual ULONG    __stdcall  NondelegationAddRef() = 0;
        virtual ULONG    __stdcall  NondelegationRelease() = 0;
};
```

内部对象的定义

```
class CA : public ISomeInterface, public INondelegationUnknown
{
protected:
    ULONG          m_Ref;
public:
    CA ( IUnknown *pUnknownOuter);
    ~ CA ( );
public :
    virtual HRESULT __stdcall  QueryInterface(const IID& iid, void **ppv);
    virtual ULONG   __stdcall  AddRef() ;
    virtual ULONG   __stdcall  Release() ;

    virtual HRESULT  __stdcall  NondelegationQueryInterface(
                                        const IID& iid, void **ppv) ;
    virtual ULONG   __stdcall  NondelegationAddRef() ;
    virtual ULONG   __stdcall  NondelegationRelease() ;
    .......

    private :
          IUnknown  *m_pUnknownOuter;  // pointer to outer IUnknown
};
```

聚合：非委托IUnknown接口的实现

- AddRef和Release与正常情况相同

```
HRESULT CA:: NondelegationQueryInterface(const IID& iid, void **ppv)
{
    if ( iid == IID_IUnknown ) {
            *ppv = (INondelegatingUnknown *) this ;
            ((IUnknown *)(*ppv))->AddRef() ;
    } else if ( iid == IID_SomeInterface )  {
            *ppv = (ISomeInterface *) this ;
            ((ISomeInterface *)(*ppv))->AddRef() ;
    } else {
            *ppv = NULL;
            return E_NOINTERFACE ;
    }
    return S_OK;
}
```

聚合：委托IUnknown接口的实现

```
ULONG CA:: AddRef ()
{
    if  ( m_pUnknownOuter != NULL )
            return m_pUnknownOuter->AddRef();
    else
            return NondelegatingAddRef();
}
ULONG CA:: Release ()
{
    ……
}
HRESULT CA:: QueryInterface(const IID& iid, void **ppv)
{
    if  ( m_pUnknownOuter != NULL )
            return m_pUnknownOuter->QueryInterface(iid, ppv);
    else
            return NondelegatingQueryInterface(iid, ppv);
}
```

聚合：外部对象的创建

- 外部对象类厂在构造了CB之后，调用Init函数，类厂的CreateInstance函数与包容模型相同。但CB::Init函数不同：

```
HRESULT CB::Init()
{
    IUnknown *pUnknownOuter = (IUnknown *)this;
    HRESULT result = ::CoCreateInstance(CLSID_ComponentA,
                    pUnknownOuter,  CLSCTX_INPROC_SERVER,
                    IID_IUnknown, (void **)& m_pUnknownInner) ;
    if (FAILED(result))
            return E_FAIL;
    else
            return S_OK;
}
```

聚合：外部对象的析构

```
CB::~CB ( )
{
        if (m_pUnknownInner ! = NULL)
                m_pUnknownInner->Release() ;
}
```

内部对象的创建

```
HRESULT CAFactory::CreateInstance(IUnknown *pUnknownOuter,
                                  const IID& iid, void **ppv)
{
    //  iid must be IID_IUnknown for aggregating
    if ( ( pUnknownOuter != NULL ) && ( iid != IID_IUnknown ) )
    {
        return CLASS_E_NOAGGREGATION;
    }
    *ppv=NULL;

    //Create the object passing function to notify on destruction.
    CA *pObj=new CA (pUnknownOuter);
    if (pObj == NULL) return E_OUTOFMEMORY;

    //Obtain the first interface pointer (which does an AddRef)
    HRESULT hr = pObj->NondelegatingQueryInterface(iid, ppv);
    return hr;
}
```

聚合：内部对象的构造函数

```
CA::CA (IUnknown *pUnknownOuter)
{
    m_pUnknownOuter = pUnknownOuter;
}
```

聚合：外部对象的创建（修订）

```
HRESULT CB::Init()
{
    IUnknown *pUnknownOuter = (IUnknown *)this;
    HRESULT result = ::CoCreateInstance(CLSID_CompA, pUnknownOuter,
                                        CLSCTX_INPROC_SERVER,
                                        IID_IUnknown, (void **)& m_pUnknownInner) ;
    if (FAILED(result)) return E_FAIL;

    result = m_pUnknownInner->QueryInterface(IID_ISomeInterface,
                        (void **)&m_pSomeInterface);
    if (FAILED(result)) {
            m_pUnknownInner->Release();
            return E_FAIL;
    }

    pUnknownOuter->Release();
    return S_OK;
}
```

聚合：外部对象的创建（修订）

```
HRESULT CBFactory::CreateInstance(IUnknown *pUnknownOuter,
                                  const IID& iid, void **ppv)
{
    CB  *pObj;      HRESULT hr;   *ppv=NULL;

    if (NULL != pUnknownOuter)      return CLASS_E_NOAGGREGATION;

    pObj=new CB ();
    if (pObj == NULL)        return E_OUTOFMEMORY;

    pObj->AddRef();  // The Reference count of pObj is 1
    hr = pObj->Init();
    if (FAILED(hr) ) {        g_CompBNumber --;  delete pObj;
                              return E_FAIL;     }
    hr=pObj->QueryInterface(iid, ppv);
    pObj->Release();          // The Reference count of pObj is 1
    return hr;
}
```

聚合：外部对象的析构（修订）

```
CB::~CB ( )
{
    m_Ref = 1;

    IUnknown *pUnknownOuter = this;
    pUnknownOuter->AddRef ( );

    if (m_pSomeInterface != NULL)
        m_pSomeInterface->Release();

    if (m_pUnknownInner != NULL)
        m_pUnknownInner->Release() ;
}
```

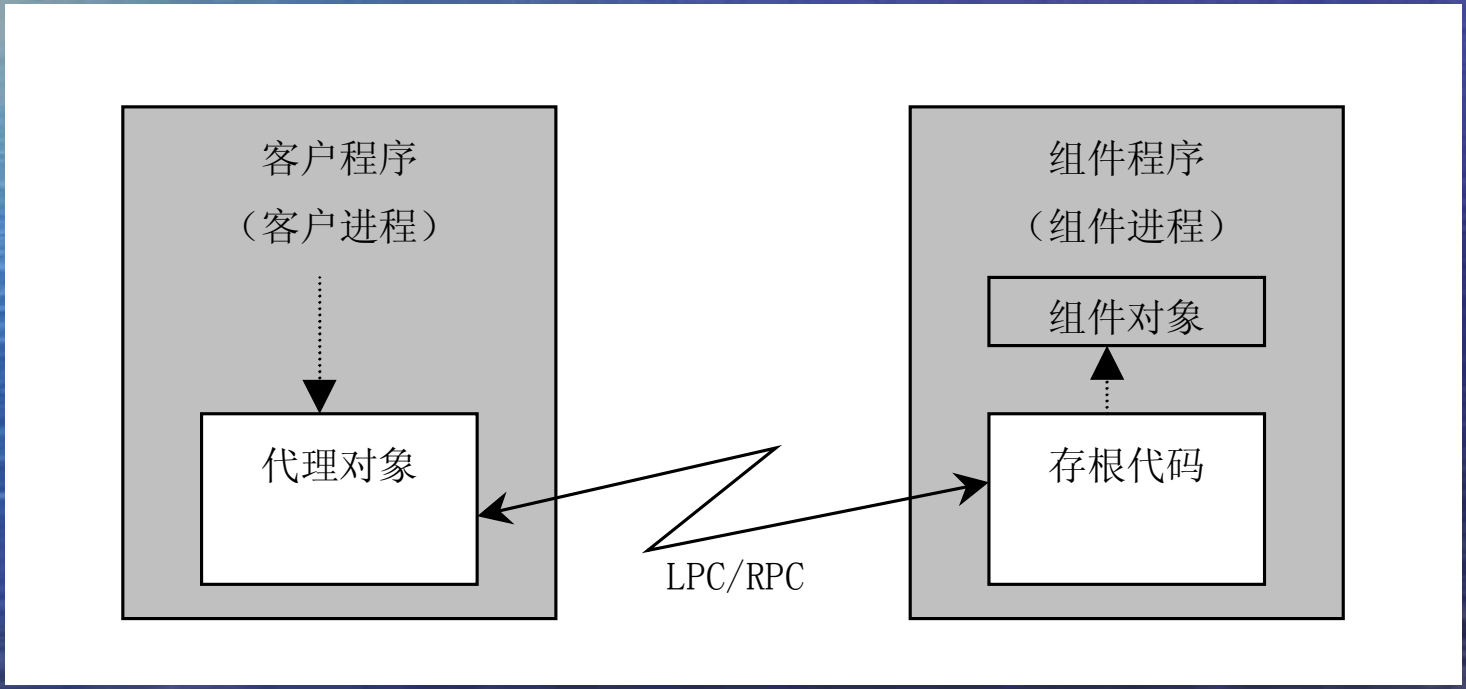
聚合例子

进程透明性

- 基本的机制：列集(marshaling)

marshal的字典含义：编组、调度、引导、安排

整顿、配置、汇集、排列、集合



列集(marshaling)

- 定义：是指客户进程可以透明地调用另一进程中的对象成员函数的一种参数处理机制。
- 连接：客户程序的一个有效接口指针，连接是在函数调用的过程中产生的
- 客户与进程外组件的协作包括：
  （1）如何建立一个连接
  （2）如何使用连接进行跨进程调用

进程透明性：两个阶段

- 客户访问对象时，对于客户和对象来说都是透明的
- 客户创建对象时，COM库介入其中，保证了客户端的透明效果；服务器端不完全透明

两种列集方式

- （1）自定义列集法(custom marshaling)或基本列集法(basic marshaling architecture)
- （2）标准列集法(standard marshaling)
- 标准列集法是自定义列集法的一个特例
- 两者粒度不同，标准列集法以接口为基础，自定义列集法以对象为基础

跨进程建立一个连接

- 列集：找到代理对象的CLSID，或者标准代理对象；列集数据包，包括跨进程信息。这些信息被称为对象引用(marshaled object reference)
- 传输：把代理对象的CLSID和列集数据包传输到客户进程中。
- 散集：客户进程中，根据传输过来的CLSID创建代理对象，并且把列集数据包传给代理对象。代理对象向客户返回一个接口指针。

列集和散集

```
HRESULT CoMarshalInterface( IStream *pStm,
        REFIID riid,
        IUnknown *pUnk,
        void * dwDestContext,
        unsigned long pvDestContext,
        unsigned long mshlflags
);

HRESULT CoUnmarshalInterface(IStream *pStm,
    REFIID riid
    void * * ppvObj
);
```

IMarshal接口

- IMarshal接口是使用自定义列集或标准列集的标志

```
class IMarshal : public IUnknown
{
    HRESULT GetUnmarshalClass( ...) = 0;
    HRESULT GetMarshalSizeMax(...) = 0;
    HRESULT MarshalInterface( ...) = 0;
    HRESULT UnmarshalInterface(...) = 0;
    HRESULT DisconnectObject(...) = 0;
    HRESULT ReleaseMarshalData(...) = 0;
};
```

列集的过程(CoMarshalInterface)

- 说明：列集过程发生在对象进程中
- 首先向对象查询是否实现了IMarshal接口，如果实现了，则调用其GetUnmarshalClass成员函数获取代理对象的CLSID(如果对象没有实现IMarshal接口，则指定使用COM提供的缺省代理对象，其CLSID为CLSID_StdMarshal)。
- 调用GetMarshalSizeMax函数确定列集数据包最大可能的大小值，并分配一定的空间。
- 调用MarshalInterface成员函数建立列集数据包。

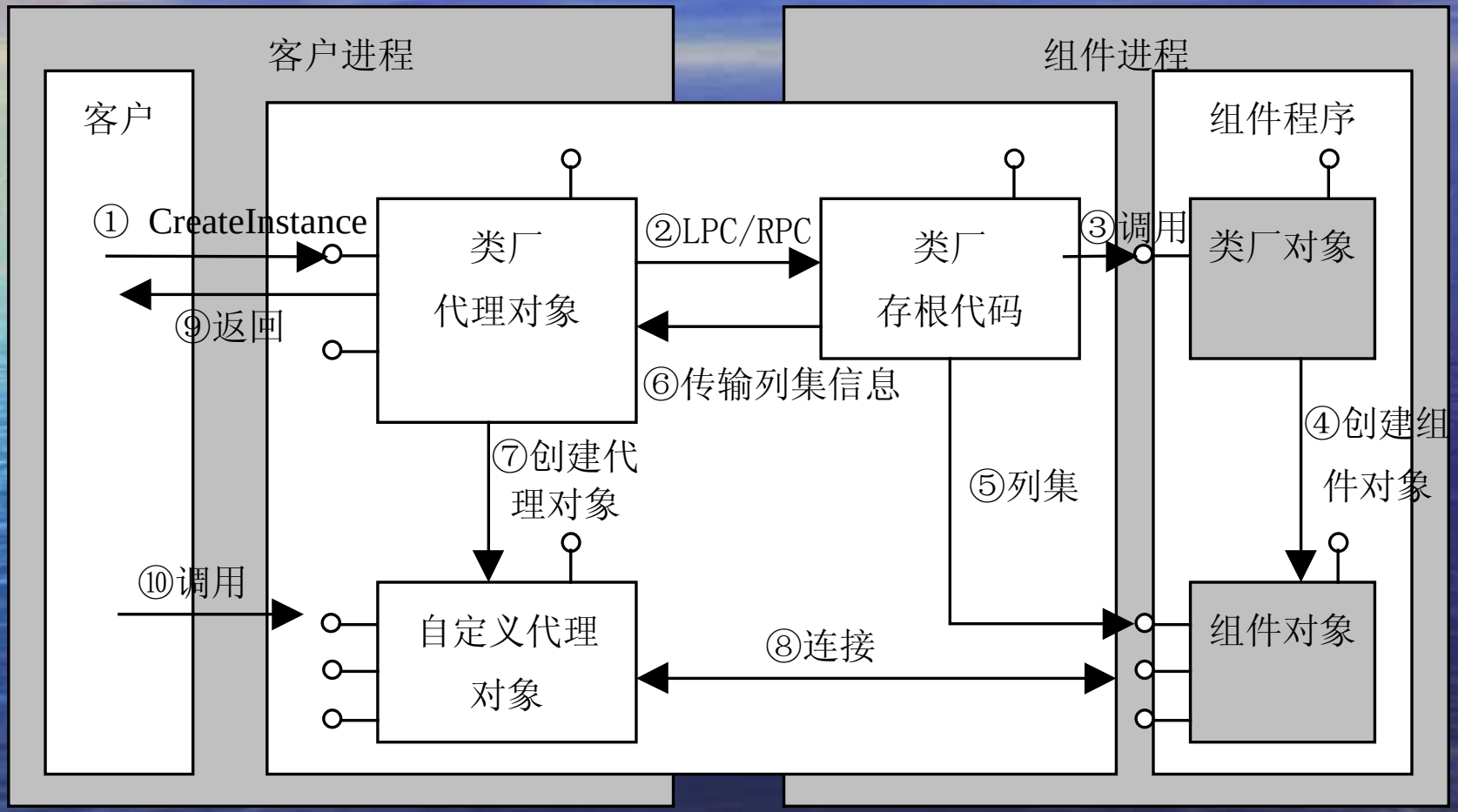
散集过程(CoUnmarshalInterface)

- 从stream中读出proxy的CLSID
- 根据CLSID创建一个proxy
- 获取proxy的IMarshal接口指针
- 调用IMarshal::UnmarshalInterface，把stream中的数据传给proxy，proxy根据这些数据建立起它与对象之间的连接，并返回客户请求的接口指针

Custom marshaling举例

- 假定客户已经建立了它与类厂之间的连接，也就是说它通过CoGetClassObject获得了类厂的接口指针
- 客户要通过类厂创建另一个COM对象，而这个对象使用custom marshaling
- 客户调用IClassFactory::CreateInstance创建对象，并返回对象的接口指针

通过类厂建立代理对象和组件对象自定义列集过程



自定义列集的要点

- 对象必须实现IMarshal接口
- 代理对象也必须实现IMarshal接口，并且代理对象与进程外对象之间协作
- 代理对象必须负责所有接口的跨进程操作
- 典型用途：
  - 提高跨进程调用的效率，使用缓存状态等优化技术
  - marshal-by-value

标准列集

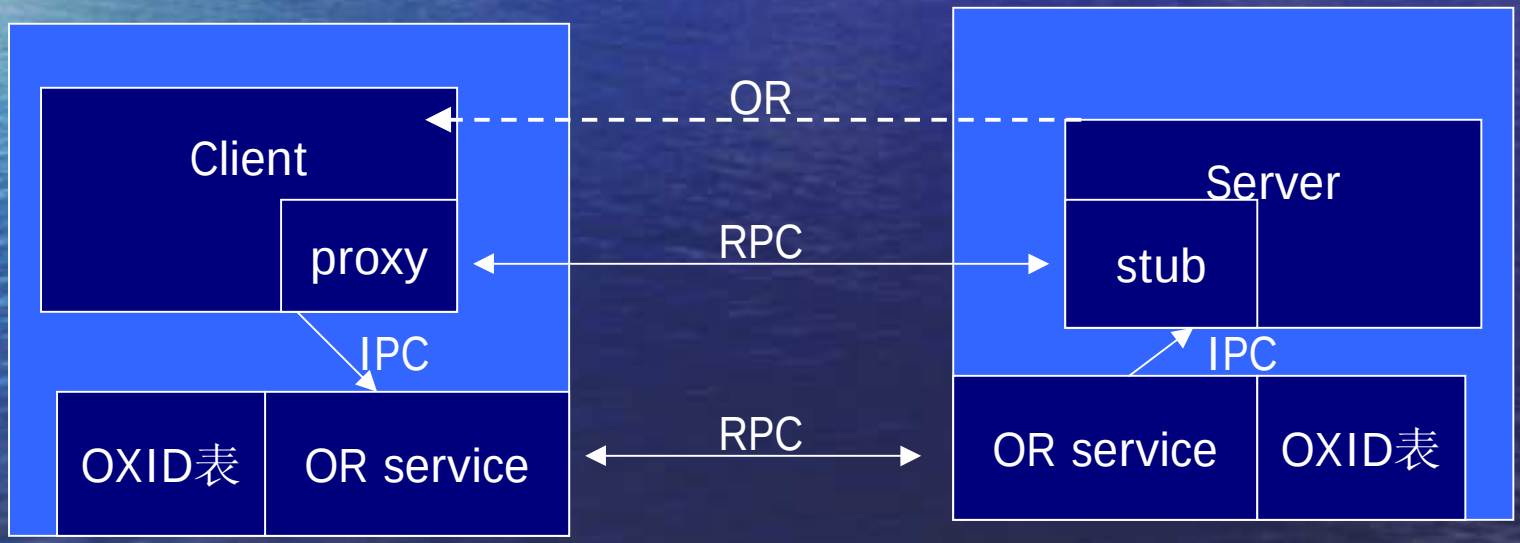
- 对象不需要实现IMarshal，COM提供代理对象，其CLSID为CLSID_StdMarshal
- 细粒度的控制能力，可以控制每个接口的marshaling
- 跨进程通信被抽象成RPC通道，RPC通道也是一个COM对象
- 代理对象按照聚合模型实现每一个被列集的接口
- 标准列集方式下的对象引用(OR)

标准列集方式下的对象引用(OR)

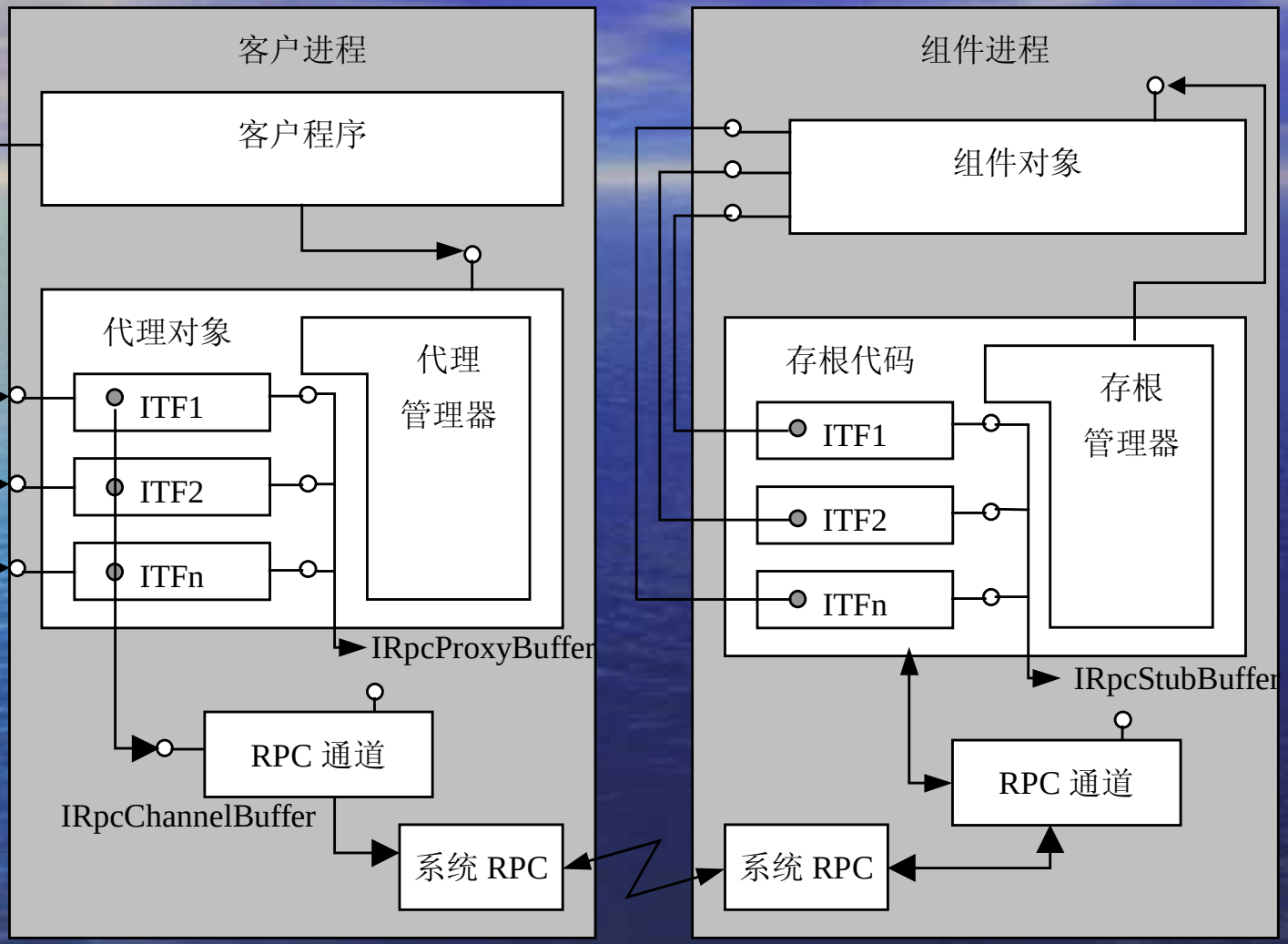


COM的标准远程结构

- COM使用ORPC协议实现跨进程(套间)通信
- COM ORPC建立在MS RPC基础上
- MS RPC有很好的扩展性



标准列集的proxy和stub结构



RPC通道

```
class IRpcChannelBuffer : public IUnknown
{
        HRESULT   GetBuffer(RPCOLEMESSAGE *pMessage,
                                        REFIID riid) = 0;
        HRESULT   SendReceive(RPCOLEMESSAGE pMessage,
                                        ULONG *pStatus) = 0;
        HRESULT   FreeBuffer(RPCOLEMESSAGE pMessage) = 0;
        HRESULT   GetDestCtx(DWORD *pdwDestCtx,
                                        void **ppvDestCtx) = 0;
        HRESULT   IsConnected() = 0;
};
```

IRpcProxyBuffer

```
class IRpcProxyBuffer : public IUnknown
{
    HRESULT   Connect(IRpcChannelBuffer *pRpcChannelBuffer) = 0;
    void      Disconnect() = 0;
};
```

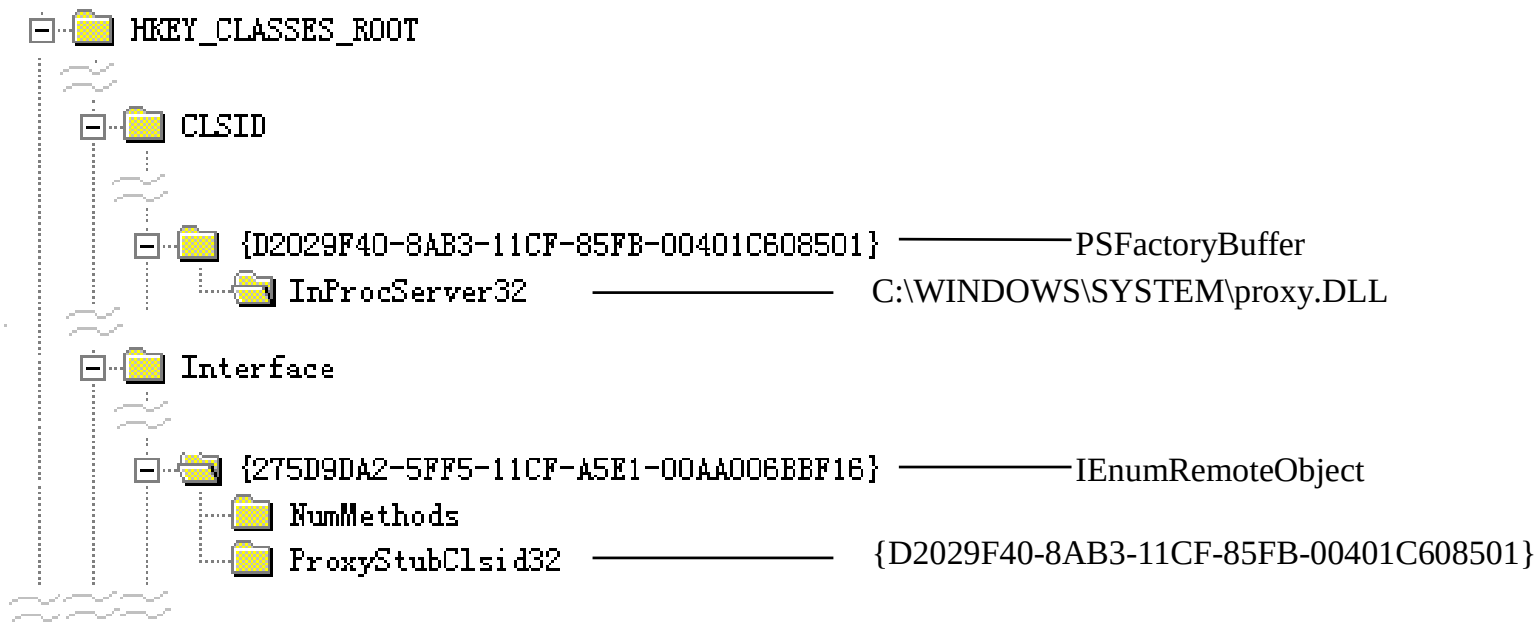
- 每个接口代理都必须实现IRpcProxyBuffer接口，并且是非委托IUnknown
- 代理管理器通过这个接口把接口代理与RPC通道连接起来，Connect方法把RPC通道保存起来
- 接口代理接到方法请求后，通过IRpcChannelBuffer接口的GetBuffer和SendReceive方法处理远程方法调用

IRpcStubBuffer

```
class IRpcStubBuffer : public IUnknown
{
    HRESULT             Connect(IUnknown *pUnkServer) = 0;
    void                Disconnect() = 0;
    HRESULT             Invoke(RPCOLEMESSAGE *pMessage,
                        IRpcChannelBuffer *pChannel) = 0;
    IRPCStubBuffer*     IsIIDSupported(REFIID iid) = 0;
    ULONG               CountRefs() = 0;
    HRESULT             DebugServerQueryInterface(void **ppv) = 0;
    void                DebugServerRelease(void *pv) = 0;
};
```

- 存根管理器通过Connect方法把接口存根与目标对象联系起来
- Invoke函数处理来自接口代理的方法请求，也要利用RPC通道把处理结果发送回去

代理/存根对象的注册信息



接口代理和接口存根的创建

```
clsid = LookUpInRegistry(iid);
CoGetClassObject(clsid, CLSCTX_SERVER,
        NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateProxy(pUnkOuter, riid, &pProxy,
   &piid);

clsid = LookUpInRegistry(iid);
CoGetClassObject(clsid, CLSCTX_SERVER,
        NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateStub(iid, pUnkServer, &pStub);
```

跨进程访问

- 调用：客户-〉对象，处理[in]或[in,out]参数
  - 列集：把数据marshal到一个buffer中
    - 如果是普通数据，如整数、字符串等
    - 如果是接口指针，调用CoMarshalInterface
  - 传输：通过RPC通道
  - 散集：从buffer中unmarshal出数据
    - 如果是普通数据，直接恢复
    - 如果是接口指针，则调用CoUnmarshalInterface
- 调用返回：对象-〉客户，处理[out]、[in,out]参数及返回值
  - 同样分marshal、传输、unmarshal三个步骤

创建进程外COM对象

- 客户调用CoGetClassObject创建类厂对象
- 在CoGetClassObject函数内部，它找到EXE组件的程序位置后，启动组件进程，等待...
- 组件进程启动后，调用CoInitialize初始化，创建所有的类厂，调用CoRegisterClassObject把类厂注册到COM中
- CoRegisterClassObject对于类厂的接口指针marshaling，采用标准列集方法得到列集数据包
- 客户进程中的CoGetClassObject正在等待这些列集数据包（待续）

创建进程外COM对象(续)

- CoGetClassObject对类厂unmarshaling，得到类厂的代理对象
- 客户调用类厂代理对象的CreateInstance
- 通过标准列集法，类厂代理对象把创建请求通过RPC通道传给类厂存根
- 类厂存根调用类厂CreateInstance创建目标对象
- 类厂存根对目标对象的接口指针进行marshaling，并通过RPC通道传回给客户
- 类厂代理unmarshal得到目标对象的代理对象并返回给客户代码

CoRegisterClassObject

```
STDAPI CoRegisterClassObject(
  REFCLSID rclsid,            //Class identifier (CLSID) to be registered
  IUnknown * pUnk,            //Pointer to the class object
  DWORD dwClsContext, //Context for running executable code
  DWORD flags,                //How to connect to the class object
  LPDWORD  lpdwRegister
);
HRESULT CoRevokeClassObject( DWORD dwRegister );
```

```
typedef enum tagCLSCTX
{
    CLSCTX_INPROC_SERVER    = 1,
    CLSCTX_LOCAL_SERVER     = 4
    CLSCTX_REMOTE_SERVER    = 16
    ...
} CLSCTX;
```

```
typedef enum tagREGCLS
{
    REGCLS_SINGLEUSE       = 0,
    REGCLS_MULTIPLEUSE     = 1,
     ...
} REGCLS;
```

问题解释

- 为什么要用类厂作为对象创建机制？
  - 类厂与对象使用同一套远程机制，无需另建一套专门用来创建对象的跨进程机制
- 为什么类厂的引用计数不计入组件总计数内？
- COM库的介入保证了进程透明性

标准列集的实现

- COM已经提供了缺省的代理对象、存根管理器以及RPC通道
- 我们只需实现每一个接口的代理/存根组件。参数和返回值的数据类型是关键。
- 只有对非标准接口才需要实现代理/存根组件

代理/存根实现：IDL描述

```
import "unknwn.idl";
#define MaxWordLength 32
[
   object,
   uuid(54BF6568-1007-11D1-B0AA-444553540000),
   pointer_default(unique)
]
interface IDictionary : IUnknown
{
    HRESULT Initialize();
    HRESULT LoadLibrary([in, string] WCHAR *pFilename);
    HRESULT InsertWord([in, string] WCHAR *pWord,
                       [in, string] WCHAR *pWordUsingOtherLang);
    HRESULT DeleteWord([in, string] WCHAR *pWord);
    HRESULT LookupWord([in, string] WCHAR *pWord,
                       [out] WCHAR pWordOut[MaxWordLength]);
    HRESULT RestoreLibrary([in, string] WCHAR *pFilename);
    HRESULT FreeLibrary();
};
```

IDL描述要点

- 不要使用int数据类型，而改用short或者long数据类型。因为int类型与平台相关，而short和long与平台无关。
- 指针类型不要使用void *，接口指针类型可使用IUnknown *或其他明确定义的接口指针类型。
- COM标准中的字符串使用双字节宽度，对应在IDL中也使用双字节宽度。
- 函数的每一个参数必须明确指明是输入参数、输出参数或者输入-输出参数。
- IDL中也可以定义结构类型，并以结构类型作为参数类型。
- 数组类型必须明确指定大小，如果需要动态指定大小或只用到数组的一部分，可使用size_is、length_is和max_is等修饰符。
- IDL文件中的import指令与C/C++中的include类似。

实用工具：MIDL

- 命令行：
  - midl dictionary.idl
- 则产生下面的文件：
  - dictionary.h —— 包含接口说明的头文件，可用于C或者C++语言；
  - dictionary_p.c —— 该C文件实现了接口IDictionary的代理和存根；
  - dictionary_i.c —— 该C文件定义了IDL文件中用到的所有全局描述符GUID，包括接口描述符；
  - dlldata.c —— 该C文件包含代理/存根程序的入口函数以及代理类厂所需要的数据结构等。

创建代理/存根

- 编写一个DEF文件。引出四个标准函数，再加上GetProxyDllInfo。
- 编写一个MAKE文件。
- 编译选项中加入EGISTER_PROXY_DLL
- 连接选项中加入rpcrt4.lib、uuid.lib
- 接口注册：regsvr32 dictionary.dll

进程外组件注意事项

- 自注册方式的变化
  - 命令行参数/RegServer和/UnregServer
- 注册类厂
- 何时被卸载
- 调用CoInitialize和CoUninitialize
- 实现自定义接口的代理/存根组件

自由作业

- 聚合例子
  - 内部对象的实现细节
- 进程外组件例子
  - 增加了一个代理/存根工程
  - 组件的注册方式以及调试方式的变化
  - CoRegisterClassObject函数对类厂的用法
  - 客户代码和组件对象代码与以前例子基本相同——体现了进程透明性