COM接口与对象

潘爱民

panaimin@icst.pku.edu.cn

内容

- 组件的接口
- COM接口
- COM IDL
- COM对象

从历史看COM

- COM产生的背景
  - 93年因为OLE 2的需要而产生
  - OLE 1的缺陷
- COM又从OLE中脱颖而出
  - COM的优势不限于OLE
  - COM成为Microsoft跟上Internet的一项重要基础技术
- 今天的Windows平台上，COM无处不在

COM基础——几个概念

- COM组件
  - 可独立发布的二进制组件
  - 在Windows平台上为DLL或者EXE
- COM对象
  - 通过COM接口提供服务
  - 符合OO中对象的基本概念
- COM接口
  - 客户与对象之间的协议，对象实现COM接口，客户使用COM接口

如何设计？

- COM组件
  - 为方便起见，只讨论Windows平台上DLL类型的组件
- COM对象
  - 如何标识一个对象？对象以什么形式存在？客户如何创建对象？
  - 对象如何暴露接口？一个或是多个？
- COM接口
  - 要求：跨编译器、跨语言、跨平台

设计COM接口——从C++入手

- C++类：接口与实现的分离
  - 接口：类的public部分

```
class CMyString
{
private:
   char *m_psz;
public:
   CMyString(const char * psz);
   ~CMyString();
   const char*Find(const char *psz);
   int Length();
};
```

设计COM接口——从C++入手(续)

- C++类的实现

```
CMyString::CMyString(const char * psz)
 : m_psz( new char[psz ? strlen(psz)+1 :1]) {
   if ( psz )
      strcpy(m_psz,psz);
   else
      m_psz[0] = 0;
}
CMyString::~CMyString() {
   delete [] m_psz;
}
const char*CMyString::Find(const char *psz) {
   return strstr(m_psz,psz);
}
int CMyString::Length() {
   return strlen(m_psz);
}
```

C++类的链接linking

- 静态链接
  - 许多类库的做法
  - 编译时刻的链接
- 静态链接的缺点
  - 代码重复：多个程序各有自己的代码，需要更多的内存
  - 客户程序占据更多的外存空间
  - 库代码更新需要重新编译所有的客户程序

C++类的链接linking(续)

- 动态链接
  - 运行时刻的链接
- 动态链接形式
  - 编译时刻通过引入库
  - 运行时刻完全动态

```
#ifdef MYSTRINGDLL
#define EXPORTORIMPORT _declspec(dllexport)
#else
#define EXPORTORIMPORT _declspec(dllimport)
#endif

class EXPORTORIMPORT CMyString
{
private:
   char *m_psz;
public:
   CMyString(const char * psz);
   ~CMyString();
   const char*Find(const char *psz);
   int Length();
};
```

C++接口如何走向COM接口

- 动态链接符合COM的需要
- C++中类形式的接口存在的问题
  - 客户看到了什么？
  - 若用Visual C++ 5.0/6.0编译器

??0CMyString@@QAE@PBD@Z

??1CMyString@@QAE@XZ

?Find@CMyString@@QAEPBDPBD@Z

?Length@CMyString@@QAEHXZ

客户眼中的C++类(续)

- 如果是Borland C++编译器(4.02)

@CMyString@$bctr$qpxc

@CMyString@$bdtr$qv

@CMyString@Find$qpxc

@CMyString@Length$qv

- 问题1：名字冲突

命名冲突解决方案1

- 模块定义文件(.def)中给出别名

```
LIBRARY MYSTRING
EXPORTS
   @CMyString@$bctr$qpxc=??0CMyString@@QAE@PBD@Z
   @CMyString@$bdtr$qv=??1CMyString@@QAE@XZ
   @CMyString@Find$qpxc=?Find@CMyString@@QAEPBDPBD@Z
   @CMyString@Length$qv=?Length@CMyString@@QAEHXZ
```

命名冲突解决方案2

- 模块定义文件(.def)中给出序号别名

```
LIBRARY MYSTRING
EXPORTS
   ??0CMyString@@QAE@PBD@Z          @1
   ??1CMyString@@QAE@XZ             @2
   ?Find@CMyString@@QAEPBDPBD@Z     @3
   ?Length@CMyString@@QAEHXZ        @4

LIBRARY MYSTRING
EXPORTS
   @CMyString@$bctr$qpxc            @1
   @CMyString@$bdtr$qv              @2
   @CMyString@Find$qpxc             @3
   @CMyString@Length$qv             @4
```

命名冲突解决方案3

- 使用C++类的vtable
- C++类的vtable不随编译器而变化
- vtable包含了各个函数的原型，顺序固定，每个函数的参数、返回类型等，函数名并不重要
- vtable要求这些接口函数必须是虚函数
- 客户如何得到vtable？
  - 仍然需要有一种办法来创建C++类
  - new/delete？需要对象的二进制结构——问题2

C++对象的二进制结构

- C++的封装是语法上的封装，而不是二进制封装
- new/delete是编译器相关的
  - 编译器不仅要知道public信息，也要知道private信息
- C++对象的二进制结构是编译器相关的
- 即使客户看到的C++类公开接口没有变化，但是C++类的实现改变了，仍然会打破客户与对象之间的连接

C++对象与客户之间的连接问题

- 客户与C++对象之间的连接点越小越好
  - 只有接口部分必要的信息才放入接口
  - 把C++类的实现细节与接口分开
  - 提取出针对所有编译器都不变的因素作为客户与对象共享的接口信息
- 方案1：句柄

句柄方案

- 方案1：句柄

```
class CMyString
{
private:
   char *m_psz;
   int m_nLength;
public:
   CMyString(const char * psz);
   ~CMyString();
   const char*Find(const char *psz);
   int Length();
};
```

```
#ifndef CMyString
class CMyString;
#endif

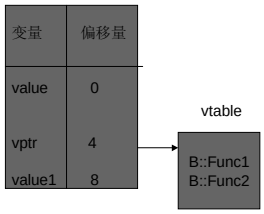
class EXPORTORIMPORT IMyString
{
private:
   CMyString *m_pthis;
public:
   IMyString(const char * psz);
   ~IMyString();
   const char*Find(const char *psz);
   int Length();
};
```

纯虚基类方案

- 前提条件：
  - 在给定平台上所有的编译器都会产生同样的二进制结构
  - 纯虚函数在单继承情况下满足这一条件
- 纯虚基类只包含虚函数，限定每个虚函数的调用习惯
- 对于跨平台的情形，我们肯定要通过中间层，所以暂时可以不考虑

虚函数的继承布局情况

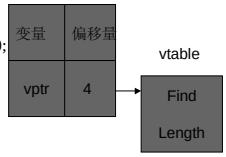
```
class B : pulic A {
    private :
        int value1;
    public:
        virtual void Func1(void)
        virtual void Func2(void)
};
```



纯虚基类方案例子

- 解决了名字冲突
- 解决了C++类的二进制布局不兼容问题
  - 客户只看到vtable，没有看到其他的实现细节
  - 保证不同语言编写的程序可以互操作
  - 在不改变接口的情况下，可以单独升级客户或者对象

```
class IString
{
   virtual const char*Find(const char *psz)=0;
   virtual int Length()=0;
};
```



纯虚接口的使用？

- 假如有一个C++对象实现了IString

```
#include "istring.h"

class CMyString : public IString
{
private:
   char *m_psz;
public:
   CMyString(const char * psz);
   ~CMyString();
   const char*Find(const char *psz);
   int Length();
};
```

- 客户怎么使用？
  - 怎么拿到vtable接口

如何创建对象？

- 不能使用new
- DLL的唯一接口是引出函数
- 可行方案：单独提供一个引出函数供客户调用

```
extern "C" _declspec(dllexport)IString *CreateString(const char *psz);

extern "C" IString *CreateString(const char *psz)
{
   return  new CMyString(psz);
}
```

通过引出函数创建对象

```
extern "C" _declspec(dllimport)
            IString *CreateString(const char *psz);

void main() {
   IString *p;

   p = CreateString("Hello");
   if (p) {
      const char*psz = p->Find("llo");
      int n = p->Length();
   }
};
```

创建对象

```
#include "istring.h"
typedef IString * (*PfnCreateString)(const char *psz);

void main() {
   IString *p;
   HANDLE h = LoadLibrary("c:\\temp\mystring.dll");
   if (NULL!=h) {
      PfnCreateString pfn =
            (PfnCreateString)GetProcAddress(h,"CreateString");
      if (pfn) {
         p = pfn("Hello");
         if (p) {
            const char*psz = p->Find("llo");
            int n = p->Length();
         }
      }
      // Be careful about calling FreeLibrary.
   }
};
```

如何删除对象？

- 删除对象发生在客户与对象建立联系之后，所以比较好办
- 但是不能用delete
- 可以让对象自己把自己删除
- 在IString中增加一个方法

```
class IString
{
   virtual void Delete()=0;
   virtual const char*Find(const char *psz)=0;
   virtual int Length()=0;
};
```

删除对象自身

```
#include "istring.h"

void main() {
   IString *p;

   p = CreateString("Hello");
   if (p) {
      const char*psz = p->Find("llo");
      int n = p->Length();
      p->Delete();
   }
};
```

```
#include "istring.h"

class CMyString : public IString
{
private:
   char *m_psz;
public:
   CMyString(const char * psz);
   virtual ~CMyString();
   void Delete();
   const char*Find(const char *psz);
   int Length();
};

void CMyString::Delete() {
   delete this;
}
```

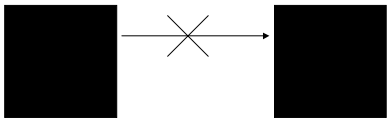
小结

- 我们已经建立起对象与客户之间的基本通信方式
- 更高的要求：
  - 接口的升级
  - 增加新的功能
  - 生命周期管理
    - 什么时候该删除对象
    - 多个客户共享同一个对象，如何管理？

对象的进化

- 在原有接口的基础上增加新的功能，例如

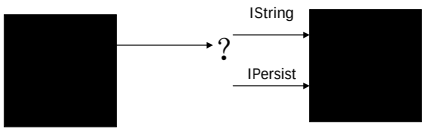
```
class IString
{
   virtual void Delete();
   virtual const char*Find(const char *psz);
   virtual int Length();
   virtual char FindAt(int index);
};
```



对象的进化（续）

- 完全增加新的功能

```
class  IPersist
{
   virtual void Delete();
   virtual void Save(const char *pszFile);
   virtual void Load(const char *pszFile);
};
```



接口的进化

- 对象的接口不能发生变化
  - 如果接口中需要增加新的方法，可以派生出新的接口来

```
class IString2 : public IString
{
        virtual char FindAt(int index);
};
```

- 新对象实现两个接口：IString2和IString
  - 不打断新的客户与老的对象之间的关系
  - 但是客户必须明确地知道对象是否实现了自己感兴趣的接口

对象实现多个接口

- 假如对象实现了两个接口IString和IPersist
  - 客户需要在runtime中明确地知道接口的类型信息，包括通过创建函数得到的初始接口类型
  - at runtime，客户可以灵活地从一个接口变换到另一个接口，如果对象不支持某个接口，客户有办法知道
  - 回忆RTTI(Runtime type identification)
    - dynamic_cast
    - RTTI依赖于编译器的实现
    - RTTI只能用于类的继承层次中

对象实现多个接口(续一)

- 所以我们需要自己构造一套类型机制，要求：
  - 每个接口都要提供类型转换机制，能转换到同一对象上实现的其他接口中
  - 客户只要得到了一个接口就可以得到其他的接口，所以创建函数可以返回任一个接口
  - 如果对象不支持某个接口，客户必须能明确地知道，而不是发生异常 —— robust
  - Dynamic_cast

对象实现多个接口(续二)

- 一个对象实现IString2和IString接口：

```
class IString
{
   virtual void Delete()=0;
   virtual void *Dynamic_cast(const char *psz)=0;
   virtual const char*Find(const char *psz)=0;
   virtual int Length()=0;
};

class  IString2 : public IString
{
   virtual char FindAt(int index)=0;
};
```

对象实现多个接口(续三)

- 实现Dynamic_cast

```
class CMyString : public IString2{...}

void *CMyString::Dynamic_cast(const char *psz)
{
   if (strcmp(psz,"IString")==0)
      return static_cast<IString *>(this);
   else
   if (strcmp(psz,"IString2")==0)
      return static_cast<IString2 *>(this);
   return NULL;
}
```

对象实现多个接口(续四)

```
#include "istring.h"

void main() {
   IString *p = CreateString("Hello");
   if (p) {
      IString2 *p2;

      const char*psz = p->Find("llo");
      int n = p->Length();
      if ((p2=(IString2 *)p->Dynamic_cast("IString2")))
         char c = p2->FindAt(3);
      p->Delete();
   }
};
```

对象实现多个接口(续五)

- 一个对象实现两个没有继承关系的接口

```
class  IPersist
{
   virtual void Delete()=0;
   virtual void *Dynamic_cast(const char *psz)=0;
   virtual void Save(const char *pszFile)=0;
   virtual void Load(const char *pszFile)=0;
};
```

对象实现多个接口(续六)

```
#include "istring.h"

class CMyString : public IString2, public IPersist {...}

void *CMyString::Dynamic_cast(const char *psz) {
   if (strcmp(psz,"IString")==0)
      return static_cast<IString *>(this);
   else
   if (strcmp(psz,"IString2")==0)
      return static_cast<IString2 *>(this);
   else
   if (strcmp(psz,"IPersist")==0)
      return static_cast<IPersist *>(this);
   return NULL;
}
```

对象实现多个接口(续七)

```
void main() {
   IString *p = CreateString("Hello");
   if (p) {
      IString2 *p2;
      IPersist *p3;

      const char*psz = p->Find("llo");
      int n = p->Length();
      if ((p2=(IString2 *)p->Dynamic_cast("IString2")))
         char c = p2->FindAt(3);
      if ((p3=(IPersist *)p->Dynamic_cast("IPersist")))
         p3->Save("c:\\temp\\str.txt");
      p->Delete();
   }
};
```

接口的转换

- 每个接口提供一个用于接口转换的函数
- 对象实现接口的时候，可以使用C++编译器本身提供的类型转换功能
- 每个接口的Dynamic_cast函数决定了客户可以访问其他哪些接口

对象的生命周期管理

- 对象只需要被删除一次。每个对象都有Delete函数？还是只有一个对象都有？进一步，什么时候删除对象？
- 客户可能拥有多个指向对象的引用，每个引用各有自己的lifetime
- 每个引用从被有效赋值开始，一直到生命周期结束，这期间被称为：outstanding reference 未完结引用
- 客户管理每个引用的lifetime，也就是说它要显式地告诉对象引用无效了

对象的生命周期管理(续)

- 每个对象要管理一个被称为引用计数(reference count)的整数值。
- 为了有效地管理对象的生命周期，它应该提供一些规则和操作，供客户遵守和使用：
- 规则：保持引用计数的确切含义，也就是记录当前outstanding reference的数目。引用计数从0开始，首次把接口递交给客户时为1，以后由客户管理，当引用计数回到0时，删除自己。
- 当客户通过复制获得新的接口指针时，引用计数加一，当某个接口不用时，减一

引用计数的两个操作

- 我们用引用计数的两个管理操作代替原来简单的Delete函数

```
class  IString
{
   virtual void DestroyPointer()=0;
   virtual void *Dynamic_cast(const char *psz)=0;
   virtual void DuplicatePointer()=0;
   virtual const char*Find(const char *psz)=0;
   virtual int Length()=0;
};

class  IPersist
{
   virtual void DestroyPointer()=0;
   virtual void *Dynamic_cast(const char *psz)=0;
   virtual void DuplicatePointer()=0;
   virtual void Save(const char *pszFile)=0;
   virtual void Load(const char *pszFile)=0;
};
```

实现引用计数的两个操作

```
#include "istring.h"

class CMyString : public IString2,
                  public IPersist
{
private:
   char *m_psz;
   long m_refcount;
public:
   CMyString(const char * psz);
   ~CMyString();
   void DuplicatePointer();
   void DestroyPointer();
   void *Dynamic_cast(const char *);
   const char*Find(const char *psz);
   int Length();
   char FindAt(int index);
   void Save(const char *pszFile);
   void Load(const char *pszFile);
};
```

```
CMystring::CMyString(const char * psz)
 : m_psz( new char[psz ? strlen(psz)+1 :1]),
   m_refcount(0) {
   if ( psz )
      strcpy(m_psz,psz);
   else
      m_psz[0] = 0;
}
void CMyString::DestroyPointer() {
   if (0<m_refcount)
      m_refcount--;
   if (0==m_refcount)
      delete this;
}
void CMyString::DuplicatePointer() {
   m_refcount++;
}
```

接口转换时刻相当于接口复制

```
void *CMyString::Dynamic_cast(const char *psz) {
   void *p = NULL;
   if (strcmp(psz,"IString")==0)
      p = static_cast<IString *>(this);
   else
   if (strcmp(psz,"IString2")==0)
      p = static_cast<IString2 *>(this);
   else
   if (strcmp(psz,"IPersist")==0)
      p = static_cast<IPersist *>(this);
   if (NULL!=p)
      m_refcount++;
   return p;
}
```

客户管理对象的生命周期

```
void main() {
   IString *p = CreateString("Hello");
   if (p) {
      IString2 *p2;
      IPersist *p3;

      const char*psz = p->Find("llo");
      int n = p->Length();
      if ((p2=(IString2 *)p->Dynamic_cast("IString2"))) {
         char c = p2->FindAt(3);
         p2->DestroyPointer();
      }
      if ((p3=(IPersist *)p->Dynamic_cast("IPersist"))) {
         p3->Save("c:\\temp\\str.txt");
         p3->DestroyPointer();
      }
      p->DestroyPointer();
   }
};
```

修改创建函数

- 让创建函数也正确地维护引用计数

```
extern "C" void *CreateString(const char *psz,
                              const char *pszinterface)
{
   void *pret = NULL;
   CMyString *p = new CMyString(psz);
   if (NULL!=p) {
      pret= p->Dynamic_cast(pszinterface);
      if (NULL==pret)
         delete p;
   }
   return pret;}
```

接口整理

- 每个接口都需要下面的三个函数
  - Dynamic_cast
  - DuplicatePointer
  - DeletePointer
- 把三个函数放到一个基接口中，所有的接口都从这个基接口派生

```
class  IAnyInterface
{
virtual void *DynamicCast(const char *psz)=0;
virtual void DuplicatePointer()=0;
virtual void DestroyPointer()=0;
};
```

其他尚待考虑的问题

- Dll什么时候被卸载？
- 如何标识一个接口？字符串？
- 线程安全？
- 如何标识一个对象？对象的身份？
- 跨进程？跨机器？对象环境？
- ……

COM接口

- 概念：函数集，以二进制的形式给出了从一方到另一方的调用规范
- 接口标识——IID
- IUnknown
- COM接口二进制结构

COM接口的标识——IID

- 是GUID的一种用法
- GUID是一个128位的长整数
- 产生规则保证了唯一性
- 例子：{54BF6567-1007-11D1-B0AA-444553540000}
- C语言结构和定义：

```
typedef struct _GUID {
    DWORD    Data1;
    WORD     Data2;      extern "C" const GUID CLSID_MYSPELLCHECKER =
    WORD     Data3;              { 0x54bf6567, 0x1007, 0x11d1,
    BYTE     Data4[8];           { 0xb0, 0xaa, 0x44, 0x45, 0x53,
} GUID;                                      0x54, 0x00, 0x00} } ;
```

IUnknown接口

- 所有的COM接口都从IUnknown派生
- C++定义：

```
class IUnknown
{
public:
  virtual HRESULT__stdcall  QueryInterface(
                const IID& iid, void **ppv) = 0 ;
  virtual ULONG        __stdcall  AddRef() = 0;
  virtual ULONG        __stdcall  Release() = 0;
};
```

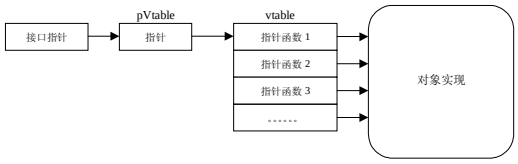
IUnknown接口(续)

- C定义：

```
typedef struct IUnknownVtbl
  {
  HRESULT ( STDMETHODCALLTYPE __RPC_FAR *QueryInterface )(
        IUnknown __RPC_FAR * This,          /* [in] */ REFIID riid,
        /* [iid_is][out] */ void __RPC_FAR *__RPC_FAR *ppvObject);
    ULONG ( STDMETHODCALLTYPE __RPC_FAR *AddRef )(
        IUnknown __RPC_FAR * This);
    ULONG ( STDMETHODCALLTYPE __RPC_FAR *Release )(
        IUnknown __RPC_FAR * This);
} IUnknownVtbl;

interface IUnknown
  {
    CONST_VTBL struct IUnknownVtbl __RPC_FAR *lpVtbl;
  };
```

COM接口结构



C语言描述示例——IDictionary

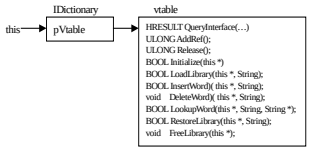
```
struct IDictionaryVtbl;

struct IDictionary
{
    IDictionaryVtbl *  pVtbl;
};

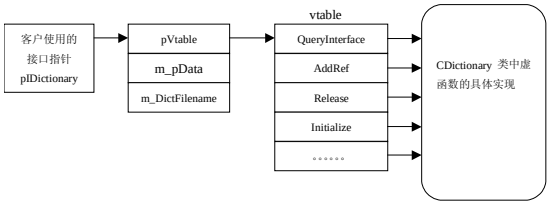
struct IDictionaryVtbl
{
    /* ... QueryInterface, AddRef, Release */
    BOOL (*Initialize)( IDictionary * this);
    BOOL (*LoadLibrary)( IDictionary * this, String);
    BOOL (*InsertWord)( IDictionary * this, String, String);
    void  (*DeleteWord)( IDictionary * this, String);
    BOOL (*LookupWord)( IDictionary * this, String, String *);
    BOOL (*RestoreLibrary )( IDictionary * this, String);
    void  (*FreeLibrary)( IDictionary * this);
};
```

C++语言描述示例—IDictionary

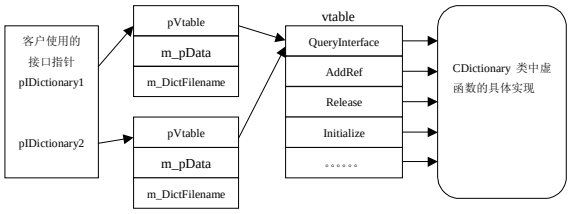
```
class IDictionary : public IUnknown
{
    virtual BOOL Initialize() = 0;
    virtual BOOL LoadLibrary(String) = 0;
    virtual BOOL InsertWord(String, String) = 0;
    virtual void  DeleteWord(String) = 0;
    virtual BOOL LookupWord(String, String *) = 0;
    virtual BOOL RestoreLibrary(String) = 0;
    virtual void  FreeLibrary() = 0;
};
```



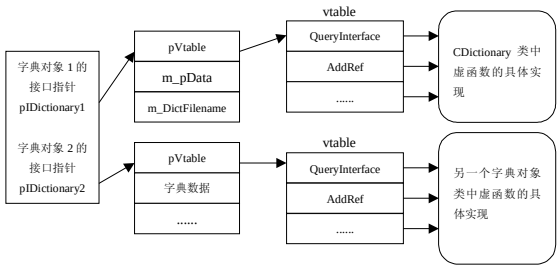
COM接口的内存模型



COM接口的内存模型(续一)



COM接口的内存模型(续二)



接口查询

- 目的：按照COM规范，一个COM对象可以实现多个接口。从一个接口到另一个接口的访问途径
- 函数QueryInterface(iid, ppv)
- 用法：
  - 初始得到了一个接口指针之后，调用它的QueryInterface函数，获得另一个接口指针
- 返回值说明了对象对接口的支持情况
- S_OK、E_NOINTERFACE、E_UNEXPECTED

接口查询用法示例

```
// load the dictionary
retValue = pIDictionary->LoadLibrary("eng_ch.dict");
if (retValue == FALSE)
{   pIDictionary->Release(); return; }
......
ISpellCheck *pISpellCheck;
HRESULT result
 = pIDictionary->QueryInterface(IID_SpellCheck,
          (void **)&pISpellCheck);
if (result != S_OK)
{ pIDictionary->Release(); return; }

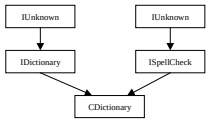
// ...... use interface pISpellCheck
//  finally, release dictionary object
pIDictionary->Release();
pISpellCheck ->Release ( );
......
```

QueryInterface实现

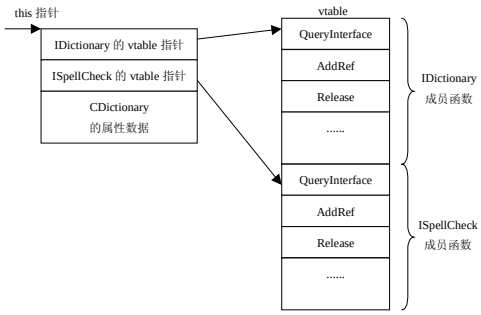
- 与对象的定义有关
- IUnknown必须是个静态接口指针，其他接口指针可以是动态的
- QueryInterface在返回接口之前，必须调用新接口的AddRef函数

QueryInterface实现举例

```
class CDictionary : public IDictionary ,  public ISpellCheck
{
public :
    CDictionary();
    ~CDictionary();
public :
    // IUnknown member function
    virtual HRESULT  QueryInterface(const IID& iid, void **ppv) ;
    virtual ULONG            AddRef() ;
    virtual ULONG            Release() ;
......
private :
    int  m_Ref ;
......
};
```



QueryInterface实现举例（续一）



QueryInterface实现举例（续二）

```
HRESULT CDictionary::QueryInterface(const IID& iid,
                                    void **ppv)
{
  if ( iid == IID_IUnknown ) {
      *ppv = static_cast<IDictionary *>this ;
      ((IDictionary *)(*ppv))->AddRef() ;
  } else if ( iid == IID_Dictionary )  {
      *ppv = static_cast<IDictionary *>this ;
      ((IDictionary *)(*ppv))->AddRef() ;
  } else if ( iid == IID_SpellCheck ) {
      *ppv = static_cast<ISpellCheck * >this ;
      ((ISpellCheck *)(*ppv))->AddRef() ;
  } else {
      *ppv = NULL;
      return E_NOINTERFACE ;
  }
  return S_OK;
}
```

COM对象的接口原则

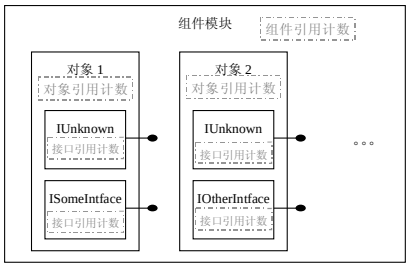
- IUnknown接口一致性
- 接口对称性
- 接口自反性
- 接口传递性
- 接口查询时间无关性

引用计数

- 目的：是为了控制对象的生命周期
- 多个客户可以独立地控制对象的生存
- 引用计数反映了被客户引用的个数
  - Outstanding references
- 引用计数是个整数，从0开始
- 两个操作：增一和减一
- 当引用计数为0时，表示没有客户在使用对象或者接口——删除

实现引用计数

- 层次，或者粒度



实现引用计数三种方案

- 组件级
  - 计数分辨率太粗
- 对象级
- 接口级
  - 计数分辨率太细

引用计数实现示例

```
class CDictionary : public IDictionary
{
public :
    CDictionary();
    ~CDictionary();
public :
virtual HRESULT QueryInterface(const IID& iid,
                  void **ppv) ;
virtual ULONG     AddRef() ;
virtual ULONG     Release() ;

virtual BOOL Initialize();
virtual BOOL LoadLibrary(String);
// ……
private :
    struct DictWord *m_pData;
    char *m_DictFilename[128];
    int  m_Ref ;
};
```

引用计数实现示例(续)

```
CDictionary::CDictionary ()
{
    m_Ref  = 0;
    // ... initialize
}

ULONG CDictionary::AddRef ()
{
    m_Ref ++;
    return  (ULONG) m_Ref;
}

ULONG CDictionary::Release ()
{
    m_Ref --;
    return  (ULONG) m_Ref;
}
```

引用计数用法

- 客户要引用接口时——增一操作
  - 调用IUnknown::AddRef()
- 客户用完接口时——减一操作
  - 调用IUnknown::Release()
- 当对象的引用计数为0时，释放
- 当组件中所有对象的引用计数为0时，卸载
- AddRef和Release的返回值不可靠

引用计数使用示例

```
// load the dictionary
BOOL retValue = pIDictionary->LoadLibrary("eng_ch.dict");
if (retValue == FALSE)
{
    pIDictionary->Release();
    return;
}
......
IDictionary *pIDictionaryForWord = pIDictionary;
pIDictionaryForWord ->AddRef();

// Insert or delete some word
pIDictionaryForWord ->InsertWord("...","...");
pIDictionaryForWord ->DeleteWord("...");
pIDictionaryForWord ->Release ( );
......

// finally, release dictionary object
pIDictionary->Release ( );
```

使用引用计数规则

- 函数参数
  - 输入参数
  - 输出参数
  - 输入-输出参数
- 局部接口指针变量
- 全局接口指针变量
- 类成员变量

使用引用计数规则（续）

- 一般性规则
  - 在顺序执行过程中，如果要对一个接口指针变量赋值，则对赋值后的接口指针变量调用AddRef，并且，如果赋值前的接口指针变量还没有结束，则赋值前必须对它调用Release以便先结束它的使用。
  - 如果要结束使用一个接口指针变量，以后不再用到它了，则调用Release函数。
- 优化

引用计数问题

- 在整个生存期内，AddRef与Release一定要配对，否则：
  - 漏掉AddRef，程序出错
  - 漏掉Release，对象永不释放

引用计数实现举例

```
ULONG CDictionary::AddRef ()
{
  m_Ref ++;
  return  (ULONG) m_Ref;
}

ULONG CDictionary::Release ()
{
  m_Ref --;
  if (m_Ref == 0 )
  {
        delete this;
        return 0;
  }
  return  (ULONG) m_Ref;
}
```

总结：COM接口特点

- 二进制特性
- 接口不变性
- 继承性(扩展性)
- 多态性——运行过程中的多态性

接口描述语言：IDL

```
interface IDictionary : IUnknown
{
  HRESULT Initialize();
  HRESULT LoadLibrary([in] string);
  HRESULT InsertWord([in] string, [in] string);
  HRESULT DeleteWord([in] string);
  HRESULT LookupWord([in] string, [out] string *);
  HRESULT RestoreLibrary([in] string);
  HRESULT FreeLibrary();
};
```

- MIDL可以由IDL文件生成C/C++接口描述

IDL简介

- 以OSF IDL为基础
- 基本数据类型
  - 与C语言非常接近，包括结构、联合、枚举、typedef等
- interface
- coclass
- library
  - 可以产生类型库

IUnknown接口的IDL描述

- IDL接口定义：

```
[
  local, object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
    typedef [unique] IUnknown *LPUNKNOWN;
    HRESULT QueryInterface( [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
}
```

IDL中类的描述

```
[
    uuid(1e196b20-1f3c-1069-996b-00dd010fe676),
    version(1.0),
    helpstring("A class"),
    helpcontext(2481), appobject
]
coclass myapp
{
    [source] interface IMydocfuncs : IUnknown;
    dispinterface DMydocfuncs;
};
```

IDL中库的描述

```
[
    uuid(12345678-1234-1234-1234-123456789ABC),
    helpstring("Hello 2.0 Type Library"),
    lcid(0x0409),
    version(2.0)
]
library Hello
{
    /* Library definition statements */
};
```

IDL中library示例

```
[
    object, uuid(. . .), <other interface
     attributes>
]
interface IKnown : IUnknown
{
    import "unknwn.idl";
    <declarations, etc. for IKnown
     interface go here>
};

[
    <library attributes>
]
```

```
library KnownLibrary
{

    //reference interface IKnown:
    interface IKnown;

    //or create a new class:
    [
        <coclass attributes>
    ]
    coclass KnowMore
    {
        interface IKnown;
    };
};
```

IDL的意义

- IDL语言无关
  - 跨语言的中间语言
- MIDL.exe产生C++头文件定义
  - 相当于C++定义
- MIDL.exe产生TLB类型库
  - COM本身提供了一套基础设施来解释类型库
- 所有的标准接口都可以在SDK中找到IDL描述

COM对象

- 客户的交互实体
- 包括属性和方法，或者状态和操作
- 能够提供服务——通过COM接口
- 对象的实现由组件完全包装起来

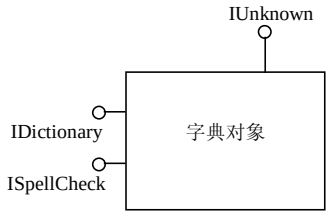
COM对象的标识——CLSID

- 是GUID的一种用法
- 创建对象的时候必须要提供CLSID
- COM对象的身份
  - 身份是否一致的可判断性

COM对象与C++对象的比较

- 层次差异
- 封装特性
- 可重用性
- 多态性的表现形式不同

COM对象和接口图示



今天讲课内容的参考资料

- “COM原理及应用”第二章
- “COM技术内幕”前四章

二选一

- “COM本质论”第一、二章
- “COM Tutorial”

二选一

自由作业

- 读程序
  - “COM原理与应用”中
    - 第二章的“字典组件和字典客户”