COM 和.NET 组件简明讲义

（V0.99）

前言

COM 是一种 Windows 平台上的现代软件组件系统，目前已经有些过时。在.NET Framework 上，全新的.NET Assembly（程序集 / 配件）取代了 COM 组件，使得 COM 组件在企业应用开发中已经不是十分重要，但是.NET Framework 仍然与 COM 有千丝万缕的联系：.NET Framework 本身就是基于 COM 组件实现的；.NET Framework 中实现了 RCW 和 CCW，通过 DCOM 远程调用.NET Assembly 仍然有一定意义；.NET Remoting 可以看作是 DCOM 的升级版；.NET Enterprise Services 是对 COM+企业组件服务的直接包装等等。了解 COM 对于了解现代软件组件系统及其实现仍然有重要意义。

本文简要介绍 COM，不涉及 COM 的过多细节，并简要介绍.NET 组件，包括.NET Assembly，适合需要了解 COM 的.NET 程序员参考。

本文中 COM 部分的程序使用 Visual C++ 6.0 开发，.NET 部分的程序基于.NET Framework 2.0 和 Visual Studio 2005。

1 现代软件组件系统简介

1.1 现代软件组件

现代软件组件（或者构件，以下简称组件）是现代软件开发，特别是企业应用开发中的重要概念，是构成现代软件的基础。组件具有语言无关性、位置透明性、自描述性、可复用性、安全性等重要特性，可以满足现代软件开发和现代软件工程的基本要求。

COM 组件、.NET Assembly、XML Web Service 等都是典型的组件。

1.2 面向对象的组件系统和非面向对象的组件系统

通常认为组件是面向对象的，因为在调用组件（或者组件对象）时，组件相当于对象，但是各种组件系统对面向对象的支持程度并不是完全相同的，通常可以将组件系统划分为两类：面向对象的组件系统和非面向对象的组件系统。

在面向对象的组件系统中，组件本身相当于类，调用者调用组件，必须先将组件实例化成为组件对象才能调用，组件对象相当于对象。

在非面向对象的组件系统中，组件本身相当于对象，调用者可以直接调用组件。

面向对象的组件系统中，对于调用一个组件，组件可以为每一个调用者提供一个独立的组件对象，独立的组件对象有一定的状态（属性值），每一个调用者都可以通过独立的组件对象保存自己一定的状态，在调用者对组件对象的多次调用中，这种状态是可以保持的，因此面向对象的组件系统中的组件可以简称为有状态组件（Stateful Component）。

非面向对象的组件系统中，对于调用一个组件，所有调用者都调用同一组件，任何一个调用者都不可能通过组件保存自己的状态，所以组件本身通常是只有方法，没有属性，也就是没有状态的，因此非面向对象的组件系统中的组件通常称为无状态组件（Stateless Component）。

COM 组件和.NET Assembly 是典型的有状态组件，而 XML Web Service 是典型的无状态组件。实际上无状态组件可以认为是一个函数(方法)库，例如 XML Web Service 就可以认为是 Internet 上的函数库。

2 COM

2.1 设计 COM 时所要达到的目标

COM 起初是作为 OLE 2 的底层技术推出的，COM 有许多重要的特性，按照 Microsoft 的文档，COM 的特性有：

⑴语言无关性（Language Independence）
⑵位置透明性（Location Transparency）
⑶厂商无关性（Vendor Independence，或者接口独立性／接口固定性）

⑷减少版本问题（Reduced Version Problem）

实际上 COM 的基本特性有：

⑴语言无关性：COM 组件是语言无关的，COM 组件的调用协议是语言无关的二进制协议。
⑵位置透明性：调用者对 COM 组件的调用，与 COM 组件的位置无关，无论 COM 组件位于什么位置，调用者调用 COM 组件的方法都相同。
⑶面向对象的特性：COM 是一种面向对象的组件系统，COM 组件相当于类，COM 组件实例化成 COM 组件对象，COM 组件对象相当于对象，调用者调用 COM 组件对象。

COM 的基本特性，也就是设计 COM 这种组件系统时所要达到的目标。简而言之：设计 COM，相当于设计一个语言无关和位置透明的面向对象系统，其基础是设计一个语言无关的面向对象系统，再考虑实现位置透明。

COM 中，完整的 COM 组件由组件本身和组件对应的类厂组件构成，实现完整 COM 组件的 DLL 或者 EXE 称为 COM 组件服务器（COM Component Server）。因为调用者／组件结构也是一种 C／S 结构，所以有时也不严格区分组件和组件服务器。

2.2 COM 的三大核心技术——接口、引用计数和类厂

设计 COM 的基础是设计一个语言无关的面向对象系统，这就要求对象的创建、调用和删除都是二进制协议。

面向对象系统的实现中，对象的属性和方法可以分离，对象可以设计成包含对象属性的数据结构，方法可以设计成针对对象属性的过程（函数），将包含对象属性的数据结构的引用（指针）作为方法过程的参数，让方法过程针对特定对象的属性进行操作，这样包含对象属性的数据结构和方法过程在逻辑上就构成通常意义上的对象。例如 C++、Object Pascal 等面向对象编程语言就是这样实现对象的。

例如，下列 C++程序和 C 程序是等价的：

C++程序（ObjectModelSample1.cpp）：

```
#include <stdio.h>
```

```
#include <tchar.h>

class CA
{
public:
	int a;
	int b;
	int c;

	void Foo();
};

void CA::Foo()
{
	_tprintf(_T("a=%d b=%d c=%d\n"),a,b,c);
};

void _tmain()
{
	CA objA;

	objA.a=1;
	objA.b=2;
	objA.c=3;

	objA.Foo();

	_tprintf(_T("The size of object objA is %d.\n"),sizeof(objA));
}
```

运行结果：

```
a=1 b=2 c=3
The size of object objA is 12.
```

C 程序（ObjectModelSample2.c）：

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

typedef struct tagCA
{
	int a;
	int b;
```

```
    int c;
}
CA;

void CA_Foo(CA *This)
{
    _tprintf(_T("a=%d b=%d c=%d\n"),This->a,This->b,This->c);
}

void _tmain()
{
    CA objA;

    objA.a=1;
    objA.b=2;
    objA.c=3;

    CA_Foo(&objA);

    _tprintf(_T("The size of structure objA is %d.\n"),sizeof(objA));
}
```

运行结果：

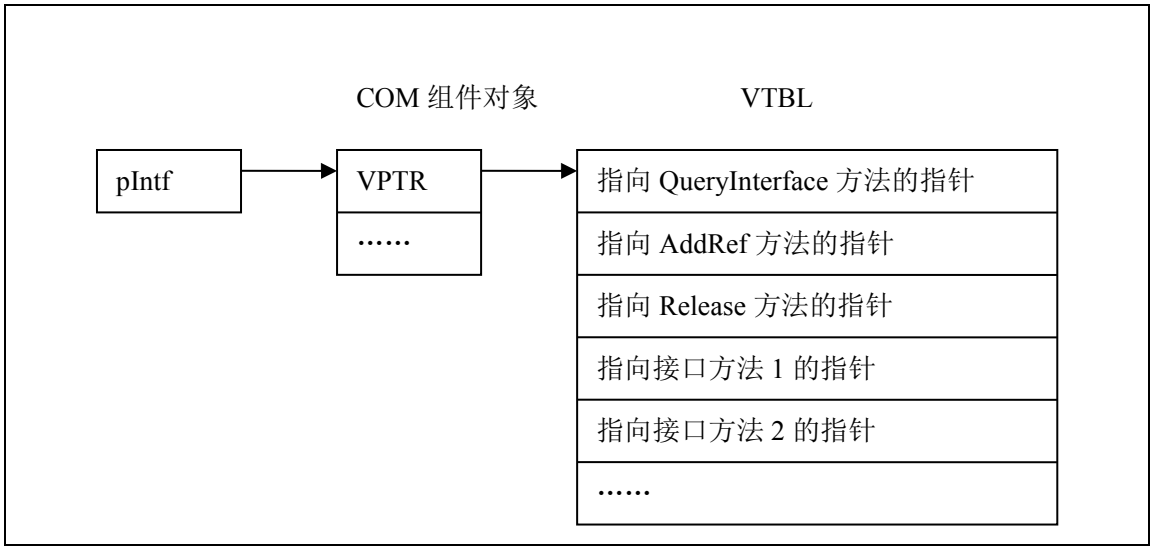
```
a=1 b=2 c=3
The size of structure objA is 12.
```

可见上述 C++程序和 C 程序的运行结果是等价的。尽管 C 程序中的“对象”objA 实际上是结构变量，只包含“对象”的“属性”，CA_Foo 函数是针对结构 CA 的结构变量的函数，CA_Foo 函数将指向结构 CA 的结构变量的指针 This 作为参数，这样 CA_Foo 函数就可以针对“对象”的“属性”进行操作，相当于“对象”的“方法”，这样结构变量 objA 和函数 CA_Foo 在逻辑上仍然相当于 C++程序中的对象 objA。

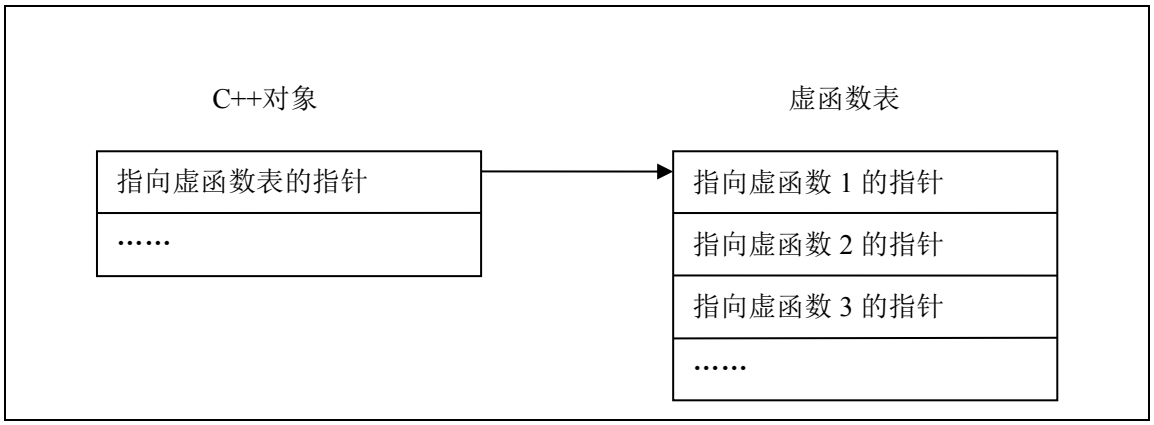
还可以注意到，C++程序中的对象 objA 的长度和 C 程序中的结构变量 objA 的长度是相同的，都是 12 字节，这说明 C++程序中的对象本质上也是只包含对象属性的数据结构，这更进一步说明了面向对象系统的实现中，对象的属性和方法可以分离。

特别注意：COM 与 C++无关，C++类只能使用 C++调用，是语言相关的，虽然 COM 组件可以用 C++描述和实现，但这不等于 COM 与 C++有必然联系，理论上说 COM 组件可以用任何编程语言实现，包括 C++、Object Pascal 等，甚至包括 C 语言和汇编语言。

接口的本质是 VPTR（指向 VTBL 的指针）和 VTBL（虚表），COM 组件实现几个接口，就有几个 VTBL，VPTR 包含在 COM 组件对象中，实际上 COM 组件对象也是包含组件对象属性的数据结构，VPTR 通常在数据结构起始处。COM 组件对象引出的指向接口的指针（通常用 pIntf 表示）指向 VPTR，VPTR 指向 VTBL，显然指向接口的指针也引用 COM 组件对象。



VPTR 和 VTBL 的格式与 C++指向虚函数表的指针和虚函数表的格式相同，因此 COM 组件的接口和组件可以使用 C++描述和实现。



显然，在运行时通过 COM 组件对象引出的指向接口的指针，可以直接定位组件对象接口方法的地址（指针），进而直接调用接口方法，实现了到组件对象接口方法的运行时绑定（后期绑定，动态链接），这样就实现了语言无关地调用 COM 组件对象。

引用计数的本质是实现语言无关地删除 COM 组件对象的方法。创建 COM 组件对象时组件对象的引用计数为 0；当使用一个组件对象引出的指向接口的指针，添加组件对象的一个引用时，应该调用接口的 AddRef 方法将引用计数加 1；当不再使用一个指向接口的指针，减少组件对象的一个引用时，应该调用接口的 Release 方法将引用计数减 1，引用计数减到 0 时，方法自删除组件对象。因为

COM 组件对象是自删除的，删除 COM 组件对象自然是语言无关的。

类厂的本质是实现语言无关地创建 COM 组件对象的方法。完整的 COM 组件由组件本身和组件对应的类厂组件构成，COM 组件服务器初始化时，必须创建类厂（类厂组件对象），然后将类厂引出的指向 IClassFactory 接口的指针暴露给 COM 环境。调用者通过类厂引出的指向 IClassFactory 接口的指针调用 IClassFactory 接口的 CreateInstance 方法创建类厂对应的组件的组件对象，这种模式相当于设计模式中创建型模式中的工厂方法模式。类厂是 COM 组件服务器自创建的，通过类厂引出的指向 IClassFactory 接口的指针调用 IClassFactory 接口的 CreateInstance 方法是语言无关的调用，创建 COM 组件对象自然是语言无关的。

2.3 COM 组件实例

下列 C++程序是实现一个完整的“HelloWorld”COM 组件的 C++程序的主程序部分，包括组件本身和组件对应的类厂组件的实现：

C++头文件（HelloWorld.h）：

```
#ifndef __HelloWorld_H__
#define __HelloWorld_H__

//IHelloWorld  接口定义

class IHelloWorld : public IUnknown
{
public:

    virtual HRESULT __stdcall HelloWorld()=0;

};

//IHelloWorld  接口的 GUID（标识）

// {B9DF3F71-A4A3-4767-B536-B7552B794CEF}
static const GUID IID_IHelloWorld =
{ 0xb9df3f71, 0xa4a3, 0x4767, { 0xb5, 0x36, 0xb7, 0x55, 0x2b, 0x79, 0x4c, 0xef } };

//HelloWorld COM 组件（组件类）的 GUID（标识）

// {B331555E-AC7E-4608-A1D6-FDF51373BE77}
static const GUID CLSID_HelloWorld =
```

```
{ 0xb331555e, 0xac7e, 0x4608, { 0xa1, 0xd6, 0xfd, 0xf5, 0x13, 0x73, 0xbe, 0x77 } };

#endif
```

C++程序（HelloWorld.cpp）：

```
//HelloWorld COM 组件（服务器）主程序

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

#include "HelloWorld.h"
#include "registry.h"

static HMODULE hModule1=NULL;

static long ObjectCount=0;
static long ServerLockCount=0;

const char szName[]=_T("HelloWorld");
const char szVerIndProgID[]=_T("HelloWorld.HelloWorld");
const char szProgID[]=_T("HelloWorld.HelloWorld.1");

//HelloWorld COM 组件定义，对应的 C++ 类名为 CHelloWorld。

class CHelloWorld : public IHelloWorld
{
private:

    long m_ref;

public:

    CHelloWorld();
    ~CHelloWorld();

    virtual HRESULT __stdcall QueryInterface(const IID& iid,void** ppv);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    virtual HRESULT __stdcall HelloWorld();

};
```

```
CHelloWorld::CHelloWorld()
{
    m_ref=0;

    InterlockedIncrement(&ObjectCount);
}

CHelloWorld::~CHelloWorld()
{
    InterlockedDecrement(&ObjectCount);
}

//IUnknown 接口的 QueryInterface 方法的实现

//通过接口 GUID（标识）查询获取组件对象引出的指向接口的指针的接口方法

HRESULT CHelloWorld::QueryInterface(const IID& iid,void** ppv)
{
    if((iid==IID_IUnknown)||(iid==IID_IHelloWorld))
    {
        *ppv=(IHelloWorld*)this;

        ((IUnknown*)(*ppv))->AddRef();

        return S_OK;
    }

    *ppv=NULL;

    return E_NOINTERFACE;
}

//IUnknown 接口的 AddRef 方法的实现

//添加引用计数的接口方法

ULONG CHelloWorld::AddRef()
{
    return InterlockedIncrement(&m_ref);
}

//IUnknown 接口的 Release  方法的实现
```

```
//减少引用计数的接口方法，当引用计数减到 0 时，自删除组件对象。

ULONG CHelloWorld::Release()
{
    long lResult1;

    lResult1=InterlockedDecrement(&m_ref);

    if(lResult1==0)
    {
        delete this;

        return 0;
    }

    return lResult1;
}

//IHelloWorld 接口的 HelloWorld 方法的实现

HRESULT CHelloWorld::HelloWorld()
{
    MessageBox(NULL,_T("Hello,
world!"),_T("Information"),MB_ICONINFORMATION|MB_OK);

    return S_OK;
}

//HelloWorld COM 组件对应的类厂组件定义，对应的 C++ 类名为
CHelloWorldClassFactory。

class CHelloWorldClassFactory : public IClassFactory
{
private:

    long m_ref;

public:

    CHelloWorldClassFactory();
    ~CHelloWorldClassFactory();

    virtual HRESULT __stdcall QueryInterface(const IID& iid,void** ppv);
    virtual ULONG __stdcall AddRef();
```

```
    virtual ULONG __stdcall Release();

    virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,const
IID& iid,void** ppv);
    virtual HRESULT __stdcall LockServer(BOOL bLock);

};

CHelloWorldClassFactory::CHelloWorldClassFactory()
{
    m_ref=0;
}

CHelloWorldClassFactory::~CHelloWorldClassFactory()
{
}

HRESULT CHelloWorldClassFactory::QueryInterface(const IID& iid,void** ppv)
{
    if((iid==IID_IUnknown)||(iid==IID_IClassFactory))
    {
        *ppv=(IClassFactory*)this;

        ((IUnknown*)(*ppv))->AddRef();

        return S_OK;
    }

    *ppv=NULL;

    return E_NOINTERFACE;
}

ULONG CHelloWorldClassFactory::AddRef()
{
    return InterlockedIncrement(&m_ref);
}

ULONG CHelloWorldClassFactory::Release()
{
    long lResult1;

    lResult1=InterlockedIncrement(&m_ref);
```

```
    if(lResult1==0)
    {
        delete this;

        return 0;
    }

    return lResult1;
}

HRESULT CHelloWorldClassFactory::CreateInstance(IUnknown*
pUnknownOuter,const IID& iid,void** ppv)
{
    if(pUnknownOuter!=NULL)
    {
        return CLASS_E_NOAGGREGATION;
    }

    CHelloWorld* pHelloWorld=new CHelloWorld;

    if(pHelloWorld==NULL)
    {
        return E_OUTOFMEMORY;
    }

    HRESULT hResult=pHelloWorld->QueryInterface(iid,ppv);

    if(FAILED(hResult))
    {
        delete pHelloWorld;
    }

    return hResult;
}

HRESULT CHelloWorldClassFactory::LockServer(BOOL bLock)
{
    if(bLock)
    {
        InterlockedIncrement(&ServerLockCount);
    }
    else
    {
        InterlockedDecrement(&ServerLockCount);
```

```
    }

    return S_OK;
}

//将类厂引出的指向 IClassFactory 接口的指针暴露给 COM 环境的 DLL 引出函数

STDAPI DllGetClassObject(const CLSID& clsid,const IID& iid,void** ppv)
{
    if(clsid!=CLSID_HelloWorld)
    {
        return CLASS_E_CLASSNOTAVAILABLE;
    }

    CHelloWorldClassFactory* pHelloWorldClassFactory=new CHelloWorldClassFactory;

    if(pHelloWorldClassFactory==NULL)
    {
        return E_OUTOFMEMORY;
    }

    HRESULT hResult=pHelloWorldClassFactory->QueryInterface(iid,ppv);

    if(FAILED(hResult))
    {
        delete pHelloWorldClassFactory;
    }

    return hResult;
}

//确定 COM 组件服务器能否从内存中卸载的 DLL 引出函数

STDAPI DllCanUnloadNow()
{
    if((ObjectCount==0)&&(ServerLockCount==0))
    {
        return S_OK;
    }
    else
    {
        return S_FALSE;
```

```
    }
}

//自注册 COM 组件服务器的 DLL 引出函数

STDAPI DllRegisterServer()
{
    //RegisterServer 函数在 REGISTRY.H 和 REGISTRY.CPP 文件中定义和实现

    //REGISTRY.H 和 REGISTRY.CPP 文件来自《Inside COM》（第二版）配套光盘

    return RegisterServer(hModule1,CLSID_HelloWorld,szName,szVerIndProgID,szProgID);
}

//自反注册 COM 组件服务器的 DLL 引出函数

STDAPI DllUnregisterServer()
{
    //UnregisterServer 函数在 REGISTRY.H 和 REGISTRY.CPP 文件中定义和实现

    //REGISTRY.H 和 REGISTRY.CPP 文件来自《Inside COM》（第二版）配套光盘

    return UnregisterServer(CLSID_HelloWorld,szVerIndProgID,szProgID);
}

BOOL APIENTRY DllMain(HANDLE hModule,DWORD ul_reason_for_call,LPVOID lpReserved)
{
    if(ul_reason_for_call==DLL_PROCESS_ATTACH)
    {
        hModule1=(HMODULE)hModule;
    }

     return TRUE;
}
```

完整的“HelloWorld”COM 组件是使用 DLL 实现的，换而言之，“HelloWorld”COM 组件服务器是 DLL 形式。因为 DLL 形式的 COM 组件服务器可以直接被调用者加载到调用者进程地址空间中，所以 DLL 形式的 COM 组件服务器通常被称为“进程内组件服务器”，相对 EXE 形式的 COM 组件服务器被称为“进程外组件服务器”。

如果 COM 组件需要远程调用，可以使用 DCOM（分布式 COM）完成，相应的 COM 组件服务器被称为“远程组件服务器”，可以是 EXE 形式或者 DLL 形式。关于进程内 COM 组件服务器、进程外 COM 组件服务器和远程 COM 组件服务器的详细内容，本文不再详述，感兴趣的读者可以查阅相关资料。

DLL 形式的 COM 组件服务器通常有 4 个引出函数：

DllGetClassObject：将类厂引出的指向 IClassFactory 接口的指针暴露给 COM 环境。
DllCanUnloadNow：确定 COM 组件服务器能否从内存中卸载。
DllRegisterServer：用于自注册 COM 组件服务器，当使用 regsvr32 <COM 组件服务器 DLL 文件名>命令注册 COM 组件服务器到注册表中时，调用该函数。
DllUnregisterServer：用于自反注册 COM 组件服务器，当使用 regsvr32 /u <COM 组件服务器 DLL 文件名>命令从注册表中反注册 COM 组件服务器时，调用该函数。

判断一个 DLL 是否实现了 COM 组件服务器，只要用 dumpbin、tdump 等 PE 文件分析工具查看一下 DLL 是否有这 4 个引出函数就可以了。

“HelloWorld”COM 组件编译连接成为 HelloWorld.dll 后，使用 regsvr32 HelloWorld.dll 命令注册，即可被调用者调用。

下列 C++ 程序实现一个“HelloWorld”COM 组件的简单调用者（HelloWorldClient.cpp）：

```
#include <stdio.h>
#include <tchar.h>
#include <objbase.h>

#include "HelloWorld.h"

void _tmain()
{
    CoInitialize(NULL);

    IClassFactory* pCF;
    IHelloWorld* pHelloWorld;
    HRESULT hResult1,hResult2;

    //获取类厂引出的指向 IClassFactory 接口的指针

    hResult1=CoGetClassObject(CLSID_HelloWorld,CLSCTX_INPROC_SERVER,0,IID
_IClassFactory,(void**)&pCF);
```

```
    if(SUCCEEDED(hResult1))
    {
        //调用 IClassFactory  接口的 CreateInstance 方法创建类厂对应的组件的组件对
象

        //同时获取组件对象引出的指向接口的指针

        hResult2=pCF->CreateInstance(NULL,IID_IHelloWorld,(void**)&pHelloWorld);

        if(SUCCEEDED(hResult2))
        {
            //调用 IHelloWorld  接口的 HelloWorld 方法

            pHelloWorld->HelloWorld();

            pHelloWorld->Release();
        }

        pCF->Release();
    }

    CoUninitialize();
}
```

3 .NET 组件

3.1 .NET Framework

.NET Framework 包含两个主要部件：.NET CLR（公共语言运行时，Common Language Runtime）和.NET 框架类库（.NET Framework Class Library）。

.NET CLR 是.NET Framework 的核心，.NET CLR 的本质是类似于 Java 虚拟机（Java VM）的虚拟机，相当于.NET 虚拟机，.NET CLR 的核心称为 VES（虚拟执行系统，Virtual Execution System）。任何.NET 编程语言的源程序最终都编译成相同的中间代码——MSIL（微软中间语言，Microsoft Intermediate Language）代码，MSIL 又称为 CIL（公共中间语言，Common Intermediate Language），MSIL 代码相当于虚拟机代码，在.NET CLR（虚拟机）上运行。

.NET 可执行文件（EXE／DLL）完全由类型构成，.NET CLR 的核心 VES 执行.NET 可执行文件，实际上是执行类型。类型遵循.NET CTS（公共类型系统，Common Type System），.NET CTS 定义了一组.NET CLI 在定义、使用和管理类型时遵循的规则，.NET CTS 确定了实现跨语言集成、类型安全和高性能代码执行的框架。

类型可分为值类型（Value Types）和引用类型（Reference Types）。通常所说的.NET 编程语言中的类（Class）属于引用类型，结构（Structure）属于值类型，.NET 编程语言的源程序通常完全由类（结构）构成，源程序编译成为可执行文件后，类型定义编译成为元数据（Metadata），类型中方法的实现通常编译成为 MSIL 代码，这样.NET 可执行文件就仍然完全由类型构成。VES 执行类型时，在元数据——类型定义的驱动下执行类型中方法的实现，这种执行方式称为托管执行（受控执行，Managed Execution）。

任何.NET 编程语言的源程序中的类型，编译后生成的元数据和 MSIL 代码都是相同格式的，与.NET 编程语言无关。显然在.NET CLR 中，创建、调用和删除对象是语言无关的，任何.NET 编程语言开发的类库，理论上可以被任何.NET 编程语言调用，自然.NET 类库可以取代 COM 组件。

.NET CLS（公共语言规范，Common Language Specification）是.NET 编程语言设计者和框架（例如类库）设计者之间的协定，CLS 指定了 CTS 的子集和用法约定。例如：类库暴露出的接口如果只使用 CLS 指定的 CTS 子集中的类型，并遵循 CLS 协定，那么类库就可以被任何.NET 编程语言调用。制定 CLS 的目的是考虑到跨语言集成，因为各种.NET 编程语言虽然有其共性，但也有其特性，例如 C#支持运算符重载而 VB.NET 不支持，同样 VB.NET 支持后期绑定语法而 C#不支持，跨语言调用的类型必须是各种.NET 编程语言都能支持的类型，所以要定义各种.NET 编程语言都支持的 CTS 子集和用法约定。

3.2 .NET Assembly

.NET Assembly 是发布成为一个单元的一个或者多个文件的集合，一个 Assembly 总是包含一个清单（Manifest），清单可以认为是 Assembly 的自描述。模块（Module）是一个文件格式中包含可执行内容的文件，如果模块包含清单，那么包含清单的模块也等价于 Assembly。

Assembly 通常包含一个或者多个模块，可以包含资源文件，还包含一个清单，可以认为.NET Assembly=Modules+Resources+Manifest。

Assembly 是可执行内容的发布单元，严格地说，任何.NET 可执行文件，包括 EXE 和 DLL，都是 Assembly，不过通常 Assembly 是指 DLL 形式的.NET 类库，作为.NET 组件出现，可以被调用者调用。

下列 C#程序实现一个完整的“HelloWorld”Assembly：

C#主程序（HelloWorld.cs）：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    public class HelloWorld
    {
        public void Hello()
        {
            Console.WriteLine("Hello, world!");
        }
    }
}
```

C# Assembly 信息程序（AssemblyInfo.cs，用于生成 Manifest）：

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// 有关程序集的常规信息通过下列属性集
// 控制。更改这些属性值可修改
// 与程序集关联的信息。
[assembly: AssemblyTitle("HelloWorld")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("TBsoft Software Studio")]
[assembly: AssemblyProduct("HelloWorld")]
[assembly: AssemblyCopyright("版权所有 (C) TBsoft Software Studio 2006")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// 将 ComVisible 设置为 false 使此程序集中的类型
// 对 COM 组件不可见。如果需要从 COM 访问此程序集中的类型，
// 则将该类型上的 ComVisible 属性设置为 true。
[assembly: ComVisible(false)]

// 如果此项目向 COM 公开，则下列 GUID 用于类型库的 ID
```

```
[assembly: Guid("68b1a0d5-cfeb-459b-a43c-7c0f4cc7587f")]

// 程序集的版本信息由下面四个值组成:
//
//      主版本
//      次版本
//      内部版本号
//      修订号
//
// 可以指定所有这些值，也可以使用“修订号”和“内部版本号”的默认值，
// 方法是按如下所示使用“*”:
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

使用 IDA Pro 4.8 反汇编编译后生成的 HelloWorld.dll 文件，可以清楚地看到对 Assembly 和模块的定义，还可以清楚地看到类型：

……

```
.assembly HelloWorld
{
  .hash  algorithm 0x00008004
  .ver 1:0:0:0
}
.assembly extern mscorlib
{
  .originator =    (
    B7 7A 5C 56 19 34 E0      89)
  .ver 2:0:0:0
}
.module HelloWorld.dll // GUID {B2A46385-E16A-4024-B7AC-1DA6D259C66D}


// Segment type: Pure code
.namespace HelloWorld
{
.class public auto ansi       HelloWorld extends [mscorlib]System.Object
{


  .method public hidebysig void   Hello()
  {
    ldstr "Hello, world!"
    call void [mscorlib]System.Console::WriteLine(class     System.String)
```

```
        ret
    }

    .method public hidebysig specialname void .ctor()
    {
        ldarg.0
        call void [mscorlib]System.Object::.ctor()
        ret
    }
}
}
```

……

上述反汇编结果中，使用一种虚拟的汇编语言表示 MSIL 代码的指令，称为 IL 汇编语言。

.NET Framework 中，.NET Assembly 取代了 COM 组件，某种意义上，可以认为.NET Framework 是一个更好的 COM（早期.NET Framework 就称为 COM+ 2.0）。

MEMO