

说明：在 Engine 的学习过程中，困难重重，原因是对好多概念好多思想不明白，于是就有了疯狂的 Google 搜索，看到了不少好的文章，于是便将这些收录下来，再加上自己的一些心得，现在向大家分享，希望对大家有所帮助，文章的出处我尽量给出链接，在很多博文中我将自己的一些观点也加入进去，还望作者原谅。Engine 中涉及到 COM 的一些东西，我就从 COM 整理。

## COM 篇

### (-) Com 速成原理

<http://www.cnblogs.com/smwikipedia/archive/2008/09/19/1259833.html>

#### 一、COM 编程思想--面向组件编程思想(COP)

众所周知，由 C 到 C++，实现了由面向过程编程到面向对象编程的过渡。而 COM 的出现，又引出了面向组件的思想。其实，面向组件思想是面向对象思想的一种延伸和扩展。因此，就让我们先来回忆一下面向对象的思想吧。

面向对象思想是将所有的操作以及所操作的对象都进行归类(由 Class 实现)，而它的目标是要尽量提高代码的可重用性(这也是面向对象相比面向过程最大的优点之一)。比如，有两个程序 A 和 B 都需要对 Class C 的对象进行操作，那么 Class C 的代码就可以重用了(即 A 和 B 都可以使用 Class C 的代码)。但是，对于这一点，面向对象做得并不好。还是举刚才的例子，程序 A 和 B 都要对 Class C 的对象进行操作，那么，程序 A 和 B 的编程人员都必须将 Class C 的代码拷贝过来，然后重新编译一次，这将是多么麻烦的事！况且，如果 Class C 的代码没有公开，那这种重用就根本不可能实现了(除非程序 A 和 B 的编程人员和 Class C 的编程人员是同一个人或者团队，但这样局限性就相当大了)。

由于面向对象的这些局限性，很多程序员就会想，如果我们编程需要重用别人的成果时，不需要重新编译别人的代码那就好了。换句话说，我们要达到的目标是，直接重用别人的成果而不是重用别人的代码。这样说也许很抽象，举个例子大家就会比较明白。比如将 Class C 的代码编译生成一个 dll，那么当其他程序员想要重用 Class C 时，就只需要在自己的程序中加载这个 dll 而不需要重新编译 Class C 的代码了(这也就是组件必须要能动态链接的原因)。正是这种思路引出了面向组件的编程思想。

下面，我就简单介绍一下面向组件的思想。在以前，应用程序总是被编写成一个单独的模块，就是说一个应用程序就是一个单独的二进制文件。后来在引入了面向组件的编程思想后，原本单个的应用程序文件被分隔成多个模块来分别编写，每个模块具有一定的独立性，也应具有一定的与本应用程序的无关性。一般来说，这种模块的划分是以功能作为标准的。比如，一个网上办公管理系统，从功能上说它需要包含网络通信、数据库操作等部分，我们就可以将网络通信和数据库操作的部分分别提出来做成两个独立的模块。那么，原本单个的应用程序就分隔成了三个模块：主控模块、通信模块和数据库模块。而这里的通信模块和数据库模块还可以做得使其具有一定的通用性，那么其他的应用程序也就可以利用这些模块了。这样做的好处有很多，比如当对软件进行升级的时候，只要对需要改动的模块进行升级，然后用重新生成的一个新模块来替换掉原来的旧模块(但必须保持接口不变)，而其他的模块可以完全保持不变。这样，软件升级就变得更加方便，工作量也更小。

说了这么多，总结一下：面向组件编程思想，归结起来就是四个字：模块分隔。这里的“分隔”有

两层含义，第一就是要“分”，也就是要将应用程序(尤其是大型软件)按功能划分成多个模块；第二就是要“隔”，也就是每一个模块要有相当程度的独立性，要尽量与其他模块“隔”开。这四个字是面向组件编程思想的精华所在，也是 COM 的精华所在！理解了这四个字，也就真正理解了面向组件编程的思想。(这里说一点题外话，COM 其实是一套规范或者说一套标准，但是在我看来，COM 的核心还在于它的思想，也就是面向组件编程思想。标准谁都能定，但是思想只有一个！)

## 二、COM 的优点

COM 的优点也就是面向组件编程思想的优点。而面向组件编程思想有很多的优点，上面所说的便于软件升级只是其中之一。对于它的优点，我总结了一下，有下面几条：

- 1、便于重用，使软件开发更快捷
- 2、便于软件升级
- 3、便于软件开发的分工协作
- 4、便于用户定制自己的应用

以上几点，第一和第二点都不用再多说了，前面讲面向组件编程思想的部分里面已经充分展示出了这两点优点。在这里我解释一下第三和第四点。

如今的很多大型软件，都不可能由某一个人单独开发，甚至不会由某一个公司去单独开发。这是因为现在的很多大型软件，综合性太强，涉及的面也太广。而一个人的精力是有限的，不可能学会这么多方面的知识，也不可能掌握到这么多方面的编程技术，即使有可能，这样做的效率也是很低下的。所以，通常的情况是分工协作。仍以前面提到的网上办公管理系统为例，这个系统分为了三个模块：主控模块、通信模块和数据库模块。由于这三个模块具有相当的独立性，那么就可以将现有的所有开发人员分为三组，每一组负责一个模块。而这三组之间，只需要商量好相互间的接口就可以了。这样，对于每一个开发人员来说，就不需要掌握所有的编程技术，甚至不需要了解其他模块的具体实现，而软件仍然能有效的开发成功。这就是所谓的便于软件开发的分工协作了。

除此之外，如果一个大型的软件希望允许用户在一定程度上定制自己的应用，那么 COM 也是最好的选择。比方说一个软件由两个模块组成，模块 A 和模块 B，现在软件的开发商希望给予用户一定的灵活性，希望可以允许用户自己定制模块 B 来实现自己特定的应用，那么就只需要公开模块 B 的所有接口；而用户自己编程实现模块 B 时也只需要实现了所有的这些接口就行了。当然，这里面还有很多问题，比如 COM 组件的注册，这涉及到 COM 标准的一些细节，在这里不作讨论。

## 三、COM 中的几个重要概念

### 1、组件：

其实只要你仔细阅读了前面的部分，组件的概念应该已经很清楚了。这里所说的组件，就是前面反复在讨论的所谓“模块”。现在我只想强调一下组件需要满足的一些条件。首先是封装性，组件必须向外部隐藏其内部的实现细节，使从外部所能看到的只是接口。然后是组件必须能动态链接到一起，而不必像面向对象中的 Class 一样必须重新编译。

### 2、接口：

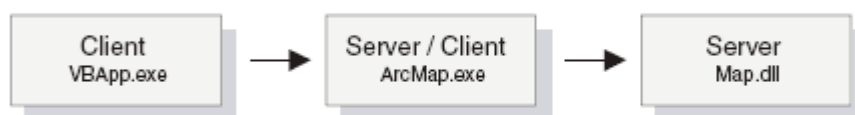
由于组件向外部隐藏了其内部的细节，因此客户要使用组件时必须通过一定的机制，也就是说要通过一定的方法来实现客户与组件之间的通信，这就需要接口。所谓接口就是组件对外暴露的、向外

部客户提供服务的“连接点”。外部的客户见不到组件内部的细节，它所能看到的只是接口，客户也是通过接口来获取组件提供的服务。这有点像 OSI 网络协议分层模型，每一层就像一个组件，它内部的实现细节对于其他层是不可见的；而每一层通过“服务接入点”向其上层提供服务，这就像这里所说的接口。一般来说，接口总是固定的，也是公开的。组件的开发人员要实现这些接口，而客户则通过接口获得服务。正是接口的这种固定和公开，才使得组件和客户能够在不了解对方的情况下达成一致。

### 3、客户：

这里所说的客户不是指使用软件的用户，而是指要使用某一个组件的程序或模块。也就是说，这里的客户是相对组件来说的。我们用《Exploring ArcObjects》（我相信学习 AE 或者 AO 的朋友都看过或者知道这本书，关于这本书的下载,Google 下就可以了，这本书是英文的，因为看起来比较麻烦，曾经有打印这本书的冲动，但是 Money 的问题……）上的一幅图说明下：

COM is a client/server architecture. The server (or object) provides some functionality, and the client uses that functionality. COM facilitates the communication between the client and the object. An object can at the same time be a server to a client and be a client of some other object's services.



The client and its servers can exist in the same process or in a different process space. In-process servers are packaged in Dynamic Link Library (DLL) form, and these DLLs are loaded into the client's address space when the client first accesses the server. Out-of-process servers are packaged in executables (EXE) and run in their own address space. COM makes the differences transparent to the client.

## 四、COM 的实现原理与雏形

COM 编程的一个重要特点就是要模块化，说得具体一些，就是要将客户和组件分隔开来，而客户和组件之间又是通过接口来通信的。下面，我就介绍一下 COM 是怎样将客户与组件分隔开来，又是怎样利用接口来实现客户与组件间的通信的。

首先我要讲讲接口。COM 中的接口实际上是一个函数地址表，当组件实现了这个接口后，这个函数地址表中就填满了组件所实现的那些接口函数的地址。而客户也就是通过这个函数地址表获得组件中那些接口函数的指针，从而获得组件所提供的服务的。从某种意义上说，我们可以把接口理解为 c++ 中的虚拟基类；或者说，在 c++ 中可以用虚拟基类来实现接口！这是因为 COM 中规定的接口的存储结构，和 c++ 中的虚拟基类在内存中的结构是一致的。其存储结构如下图：

## 虚函数表

```

vtbl指针----->Fun1()指针----->
                                Fun2()指针----->
                                Fun3()指针----->
                                .....

```

Vtbl 指针指向一个虚函数表，而这个虚函数表的表项就是指向这些虚函数的指针。

接口有了，那么组件又是怎样实现接口的呢？实际上，如果用虚拟基类来实现接口，那么组件就是对这个虚拟基类的继承。大家知道，当某个类继承于一个虚拟基类的时候，它就要实现这个虚拟基类里声明的虚函数，这就正好与组件实现接口这一点相吻合。举一个例子来说明，有一个接口

InterfaceA，组件 ComponentB 要实现这个接口，那么就可以这样用 c++ 语言来描述：

//接口：

```
Class InterfaceA
```

```
{
virtual void Fun1()=0;
virtual void Fun2()=0;
};
```

//实现了接口 InterfaceA 的组件：

```
Class ComponentB: public InterfaceA
```

```
{
virtual void Fun1()
{
    printf("Fun1\n");
}
virtual void Fun2()
{
    printf("Fun2\n");
}

};
```

而客户只需要得到一个指向 ComponentB 实体的 InterfaceA 指针就可以获得 ComponentB 组件的服务了：

//使用了组件 ComponentB 的客户：

.....

```
ComponentB CB;
```

InterfaceA \*pIA=&CB; //获得指向 ComponentB 实体的 InterfaceA 指针，以下客户就可以只通过接口来获取组件的服务

pIA->Fun1();

pIA->Fun2();

.....

但是我们注意到，这样做组件 ComponentB 和客户还是没有被完全分隔开。因为在客户代码里需要创建 ComponentB 实体，这对于只能看到接口而对组件一无所知的客户来说，是不可以接受的(比如客户不会知道组件的类名叫 ComponentB)。解决问题的方法是在实现组件的动态链接文件(比如 dll 文件)里创建组件的实体，而不是在客户代码里创建组件实体。通常组件都是以 dll 的形式出现的，而在实现组件的 dll 里都会实现一个叫 CreateInstance 的函数，这个函数可以被外部的客户调用。它返回一个接口的指针，当客户调用这个函数后就能够获得指向组件实体的接口指针了。它的实现也很简单：

//在实现组件 ComponentB 的 dll 里：

```
InterfaceA *CreateInstance()
```

```
{
```

```
    ComponentB CB;
```

```
    InterfaceA *pIA=&CB;
```

```
    return pIA;
```

```
}
```

(注：在 DirectX 编程时，很多时候都会用到 CreateXXX 之类的函数，道理就是这样。)

当然，真正的 CreateInstance 函数没有这么简单，我上面的代码只是一个简单的模拟。有个 CreateInstance 函数之后，客户代码就变成了：

//使用了组件 ComponentB 的客户：

.....

InterfaceA \*pIA=CreateInstance(); //获得指向 ComponentB 实体的 InterfaceA 指针，以下客户就可以只通过接口来获取组件的服务

pIA->Fun1();

pIA->Fun2();

.....

这样，组件和客户就完全被分隔开了，而连接它们的只有接口以及一个 CreateInstance 的函数。

以上就是 COM 的基本原理了。当然，我前面也说了，COM 其实是一套规范，它定义了很多标准，比如 COM 规定每个接口都必须继承于一个叫 IUnknown 的接口。我这里基本上没有提及它的这些标准，只是希望能通过对它进行一个简单的模拟来说清楚它的实现原理。下面就给出我模拟 COM 机制实现的一套 COM 的雏形，希望能对大家理解 COM 有帮助。

1、实现了组件 ComponentB 的 ComponentDll.dll：

//Interface.h

```
//接口
```

```
Class InterfaceA
```

```
{  
  
public:  
  
virtual void Fun1()=0;  
virtual void Fun2()=0;  
  
};
```

```
//Component.h
```

```
//组件(实现了接口 InterfaceA)
```

```
Class ComponentB: public InterfaceA
```

```
{  
  
    public:  
  
        virtual void Fun1()  
        {  
            printf("Fun1\n");  
        }  
  
        virtual void Fun2()  
        {  
            printf("Fun2\n");  
        }  
  
};
```

```
//ComponentDll.cpp
```

```
//CreateInstance 函数
```

```
    ComponentB instance;
```

```
extern "C" __declspec(dllexport) InterfaceA *CreateInstance()
```

```
{  
  
    InterfaceA *pIA=&instance;  
    return pIA;  
  
}
```

2、客户 Client.exe:

```
//Client.cpp
```

```
#include "Interface.h"
```

```
#pragma comment(lib,"ComponentDll")
```

```
int main(int argc, char* argv[])
```

```
{
```

```
InterfaceA *pIA=0;

pIA=CreateInstance();

if(pIA!=0)

    pIA->Fun1();

return 0;

}
```

## （二）COM 编程技术基础

<http://blog.csdn.net/jianglike18/archive/2009/09/22/4578041.aspx>

### 第 1 章 组件

- 1、COM，即组件对象模型，是关于如何建立组件以及如何通过组件建构应用程序的一个规范。
- 2、组件的优点：应用程序可随时间的流逝而发展变化；定制应用程序；组件库；分布式组件。
- 3、对组件的需求：组件必须动态连接；必须隐藏其内部实现细节。
- 4、COM 组件是以 Win32 动态链接库（DLLs）或可执行文件（EXEs）的形式发布的可执行代码组成的。遵循 COM 规范编写的组件将能够满足对组件家够的所有需求。COM 组件是动态链接的，COM 使用 DLL 将组件动态链接起来。对于 COM 组件的封装是很容易的。COM 组件按照一种标准的方式来宣布他们的存在。COM 组件是一种给其他应用程序提供面向对象的 API 或服务的极好方法。
- 5、COM 并不是一种计算机语言。
- 6、将 COM 同 DLL 相提并论是不合适的。实际上 COM 使用了 DLL 来给组件提供动态链接的能力。
- 7、COM 并不是像 Win32API 那样的函数集，它更主要的是一种编写能够按面向对象 API 形式提供服务的组件的方法。
- 8、COM 并不是类似于 MFC 这样的 C++类库。COM 给开发人员提供的是一种开发与语言无关的组件库的方法，但 COM 本身并没有提供任何实现。
- 9、COM 具有一个被称作是 COM 库的 API，它提供的是对所有客户及组件都非常有用的组件管理服务

### 第 2 章 接口

- 1、在 COM 中接口就是一切。
  - （1）接口可以保护系统免首外界变化的影响。
  - （2）接口可以使客户用同样的方式来处理不同的组件。
- 2、（1）COM 接口在 C++中是用纯抽象基类实现的。
  - （2）一个 COM 组件可以提供多个接口。
  - （3）一个 C++类可以使用多继承来实现一个可以提供多个接口的组件。
- 3、类并非组件。
- 4、接口并非总是继承的。对接口的继承只不过是一种实现细节而已。除了可以使用一个类来实现几个不同的接口外，还可以用单个的类来实现每一个接口再使用指向这些类的指针。
- 5、组件可以支持任意数目的接口。为支持多重接口，可以使用多重继承。支持多重接口的组件



可以被看作是接口的集合。

#### 6、COM 接口的不变性、多态以及接口继承。

(1) 一旦公布了一个接口，那么它将永远保持不变。当对组件进行升级时，一般不会修改已有的接口，而是加入一些新的接口。

(2) 多态指的是可以按同一种方式来处理不同的对象。

#### 7、虚拟函数表 (vtbl): 包含一组指向虚拟函数实现的指针。

定义一个纯抽象基类也就是定义了相应的内存结构。但此内存只是在派生类中实现此抽象基类时才会被分配。当派生类继承一个抽象基类时，它将继承此内存结构。

8、在 COM 中，对一个组件的访问只能通过函数完成，而绝不能直接通过变量。

9、接口的真正的威力在于继承此接口的所有类均可以被客户按同一方式进行处理。

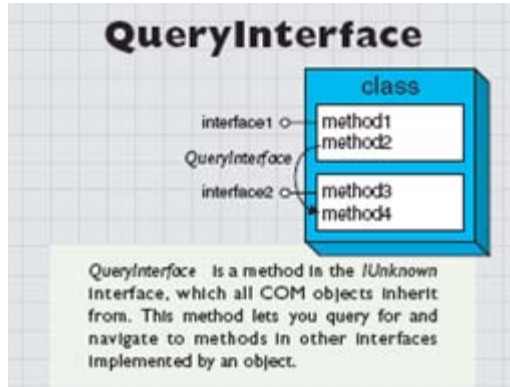
(注接口和类的区别：接口可以看成是一个特殊的类的形式，除了不能实例化一个对象外，能实现类能完成的一切功能)。

### 第 3 章 QueryInterface 函数

#### 1、接口查询：

客户同组件的交互都是通过一个接口完成的。在客户查询组件的其他接口时，也是通过接口完成的。这个接口就是 IUnknown。

(注：为什么会有 QueryInterface，因为一个类可以实现多个接口，这个函数就是为了访问同一个类中的不同接口) 用《Exploring ArcObjects》上的一幅图说明下：



IUnknown 接口的定义包含在 Win32 SDK 中的 UNKNOWN.H 头文件中。

```
interface IUnknown
```

```
{  
  
    virtual HRESULT _stdcall QueryInterface(const IID& iid, void **ppv) = 0;  
  
    virtual ULONG _stdcall AddRef() = 0;  
  
    virtual ULONG _stdcall Release() = 0;  
  
}
```

在 IUnknown 中定义了一个名为 QueryInterface 的函数。客户可以调用 QueryInterface 来决定组件是否支持某个特定的接口。

#### 2、所有的 COM 接口都需要继承 IUnknown。



- 3、由于所有的 COM 接口都继承了 IUnknown,每个接口的 vtbl 中的前三个函数都是 QueryInterface,AddRef 和 Release。若某个接口的 vtbl 中的前三个函数不是这三个,那么它将不是一个 COM 接口。由于所有的接口都是从 IUnknown 继承的,因此所有的接口都支持 QueryInterface.因此组件的任何一个接口都可以被客户用来获取它所支持的其他接口。
- 4、非虚拟继承: 注意 IUnknown 并不是虚拟基类,所以 COM 接口并不能按虚拟方式继承 IUnknown,这是由于会导致与 COM 不兼容的 vtbl。若 COM 接口按虚拟方式继承 IUnknown,那么 COM 接口的 vtbl 中的头三个函数指向的将不是 IUnknown 的三个成员函数。
- 5、一个 QueryInterface 可以用一个简单的 if-then-else 语句实现,但 case 语句是无法用的,因为接口标识符是一个结构而不是一个数。
- 6、多重类型及类型转换
- 7、QueryInterface 的规则
- (1) QueryInterface 返回的总是同一 IUnknown 指针。
  - (2) 若客户曾经获取过某个接口,那么它将总能获取此接口。
  - (3) 客户可以再次获取已经拥有的接口。
  - (4) 客户可以从任何接口返回到起始接口。
  - (5) 若能够从某个借口获取某特定接口,那么可以从任意接口都将可以获取此接口。
- 8、接口的 IID 决定了它的版本。当改变了下列条件中的任何一个时,就应给新接口指定新的 ID:
- (1) 接口中函数的数目。
  - (2) 接口中函数的是顺序。
  - (3) 某个函数的参数。
  - (4) 某个函数参数的顺序。
  - (5) 某个函数参数的类型。
  - (6) 函数可能的返回值。
  - (7) 函数参数的含义。
  - (8) 接口中函数的含义。
- 9、避免违反隐含和约:
- (1) 使接口不论在其成员函数怎么被调用都能正常工作。
  - (2) 强制客户按一定的方式来使用此接口并在文档中将这一点说明清楚。
- (注:我觉得以下几章都没有必要,但是为了保证原博客的完整,就附上)

## 第4章 引用计数

### 1、生命期控制

IUnknown 的另外两个成员函数 AddRef 和 Release 的作用就是给客户提供一种让它指示何时处理完一个接口的手段。

### 2、AddRef 和 Release 实现的是一种名为引用计数的内存管理技术。

引用计数是使组件能够自己将自己删除的最简单同时也是效率最高的方法。

COM 组件将维护一个称做是引用计数的数值。当客户从组件取得一个接口时,此引用计数值将增 1

。当客户使用完某个接口后，组件的引用计数值将减 1。当引用计数值为 0 时，组件即可将自己从内存中删除。

### 3、正确使用引用计数规则：

- (1) 在返回之前调用 `AddRef`。对于那些返回接口指针的函数，在返回之前应用相应的指针调用 `AddRef`。这些函数包括 `QueryInterface` 及 `CreateInstance`。这样当客户从这种函数得到一个接口后，它将无需调用 `AddRef`。
- (2) 在使用完接口之后调用 `Release`。在使用完某个接口之后应调用此接口的 `Release` 函数。
- (3) 在赋值之后调用 `AddRef`。在将一个接口指针赋给另外一个接口指针时，应调用 `AddRef`。换句话说，在建立接口的另外一个引用之后应增加相应组件的引用计数。
- (4) 在客户看来，引用计数是处于接口级上而不是组件级上的。
- (5) 为什么选择为每一个接口单独维护一个引用计数而不是针对整个组件维护引用计数？（1）使程序调试更为方便；（2）支持资源的按需获取。

### 6、`AddRef&Release` 的例子

```
ULONG _stdcall AddRef()
{
    return InterlockedIncrement(&m_cRef);
}

ULONG _stdcall Release()
{
    if(InterlockedDecrement(&m_cRef)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

7、当建立一个新组件时，应建立一个对此组件的引用。因此创建组件时，在将指针返回给客户之前，应该增大组件的引用计数值。这使程序员可以不必在调用 `CreateInstance` 或 `QueryInterface` 之后记着去调用 `AddRef`。

### 8、引用计数规则优化：

- (1) 输出参数规则：任何在输出参数中或作返回值返回一个新的接口指针的函数必须对此接口指针调用 `AddRef`。
- (2) 输入参数规则：对传入函数的接口指针，无需调用 `AddRef` 和 `Release`，这是因为函数的生命周期嵌套在调用者的生命周期内。
- (3) 输入-输出函数规则：对于用输入-输出参数传递进来的接口指针，必须在给它赋另外一个接口指针之前调用其 `Release`。在函数返回之前，还必须对输出参数中所保存的接口指针调用

AddRef。如：

```
void ExchangeForCachedPtr( int i, IX **ppIX)
{
    (*ppIX)->Fx(); //Do something with in-parameter.
    (*ppIX)->Release();//Release in parameter.
    *ppIX = g_Cache; //Get cached pointer.
    (*ppIX)->AddRef();//AddRef pointer.
    (*ppIX)->Fx();//Do something with out-parameter.
}
```

(4) 局部变量规则：对于局部复制的接口指针，由于它们只是在函数的生命周期内才存在，因

此无需调用 AddRef 和 Release。

(5) 全局变量规则：对于保存在全局变量中的接口指针，在将其传递给另外一个函数之前，必

须调用其 AddRef。由于此变量是全局的，因此任何函数都可以通过调用其 Release 来终止其生命期。对于保存在成员变量中的接口指针，也应按此种方式进行处理。因为类中的任何成员函数都可以改变次中接口指针的状态。

(6) 不能确定时的规则：对于任何不能确定的情形，都应调用 AddRef 和 Release 对。

## 第 5 章 动态链接

1、从 DLL 中输出函数：用 extern "c" 标记。

2、在使用 VC 时，可以用 DUMPBIN。EXE 来得到某个 DLL 中所输出的符号的清单。如下面的命令：

```
dumpbin -exports Cmpnt1.dll
```

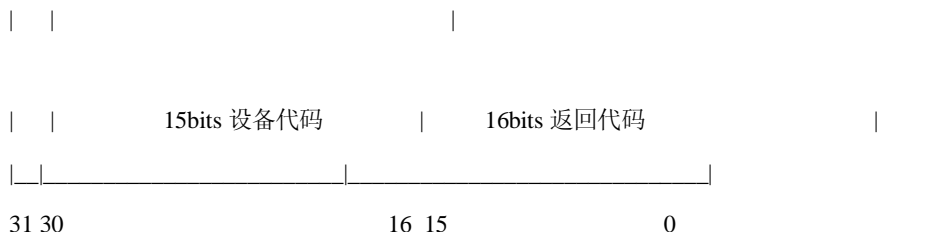
3、装载 DLL：LoadLibrary 以被装载的 DLL 的名称作为参数并返回一个指向所装载的 DLL 的句柄。

win32 的 GetProcAddress 函数可以使用此句柄以及待用的函数的名称，然后返回一个指向次函数的指针。

4、使用 DLL 实现组件的原因：DLL 可以共享它们所链入的应用程序的地址空间。

## 第 6 章 关于 HRESULT、GUID、注册表及其他细节

1、HRESULT 值的结构：



2、常用的 HRESULT 值：

3、一般不能直接将 HRESULT 值同某个成功代码（如 S\_OK）进行比较以决定某个函数是否成功也不

能直接将其同某个失败代码（如 E\_FAIL）进行比较以决定函数调用是否失败。应该使用 SUCCEEDED 和 FAILED 宏。

```
HRESULT hr = CoCreateInstance(...);
```

```
if(FAILED(hr))
```

```
    return ;
```

```
hr = pI->QueryInterface(...);
```

```
if(SUCCEEDED(hr))
```

```
{
```

```
    pIX->Fx();
```

```
    pIX->Release();
```

```
}
```

```
pI->Release();
```

4、当前所定义的设备代码：

---

FACILITY\_WINDOWS 8

FACILITY\_STORAGE 3

FACILITY\_SSPI 9

FACILITY\_RPC 1

FACILITY\_WIN32 7

FACILITY\_CONTROL 10

FACILITY\_NULL 0

FACILITY\_ITF 4

FACILITY\_DISPATCH 2

FACILITY\_CERT 11

---

5、关于定义自己的 HRESULT 的一些一般性规则：

（1）不要将 0X0000 及 0X0FFF 范围内的值作为返回代码。这些值是为 COM 所定义的 FACILITY\_ITF 代码

而保留的。只有遵循这一规则，才不致使用户自己定义的代码同 COM 所定义的代码相混淆。

（2）不要传播 FACILITY\_ITF 错误代码。

（3）尽可能使用通用的 COM 成功及失败代码。

（4）避免定义自己的 HRESULT，而可以在函数中使用一个输出参数。

6、用 MAKE\_HRESULT 宏来定义一个 HRESULT 值，此宏可根据所提供的严重级别、设备代码及返回代码

生成一个 HRESULT 值。如：

```
MAKE_HRESULT (SEVERITY_ERROR, FACILITY_ITF, 100);
```

- 7、GUID 是英文 Globally Unique Identifier(全局唯一标识符)的首字母缩写。IID 是一个 128 比特(16)字节的一个 GUID 结构。
- 8、生成 GUID：UUIDGEN.EXE 和 GUIDGEN.EXE
- 9、GUID 的比较：操作符==;等价函数 IsEqualGUID,IsEqualIID,IsEqualCLSID。
- 10、将 GUID 作为组件标识符
- 11、由于一个 GUID 值占用了 16 个字节，因此一般不用值传递 GUID 参数。而大量使用的是按引用传递。
- 12、COM 只使用了注册表的一个分支：HKEY\_CLASSES\_ROOT。
- 13、注册表 CLSID 是一个具有如下格式的串：  
{\*\*\*\*\*\_\*\*\*\*\_\*\*\*\*\_\*\*\*\*\_\*\*\*\*\*}
- 14、CLSID 关键字的子关键字 InprocServer32 关键字的缺省值是组件所在的 DLL 文件名称。
- 15、一些特殊关键字：
- (1) AppID：此关键字下的子关键字的作用是将某个 APPID（应用程序 ID）隐射成某个远程服务器名称。分布式 COM 将用到此关键字。
  - (2) 组见类别：注册表的这一分支可以将 CATID（组件类别 ID）映射成某个特定的组件类别。
  - (3) Interface：用于将 IID 映射成与某个接口相关的信息。
  - (4) Licenses：保存授权使用 COM 组件的一些许可信息。
  - (5) TypeLib：类型库关键字所保存的是关于接口成员函数所用参数的信息等。
- 16、ProgID 命名约定：  
<Program>.<component>.<version>
- 17、从 ProgID 到 CLSID 的转换：COM 库函数：CLSIDFromProgID 和 ProgIdFromCLSID：  
CLSID clsid;  
CLSIDFromProgID("\*\*\*\*.\*\*\*\*.\*\*\*\*",&clsid);
- 18、自注册：DLL 一定要输出下边两个函数：  
STDAPI DllRegisterServer();  
STDAPI DllUnregisterServer();
- 用户可以使用程序 REGSVR32.EXE 来注册某个组件，它实际上是通过上述函数来完成组件的注册的。
- 19、组件类别使开发人员能够使开发人员无需创建组件实例就能决定它是否特工所需接口。一个组件类别实际上就是一个接口集合，该集合将被分配给一个 GUID，此 GUID 此时被称做是 CATID。对于某个组件而言，若它实现了某个组件类别的所有接口，那么它可以将其注册成该组件类别的一个成员。这样，客户就能够通过从注册表中选择只属于某个特定组件类别的组件而准确找到它所需的组件。
- 20、组件类别的用途：指定某个组件必须实现的接口集合；用于指定组件需要其客户提供的接口集合。
- 22、在使用 COM 库中的其他函数（除 CoBuildVersion 外，此函数将返回 COM 库的版本号）之前，进程必须先调用 CoInitialize 来初始化 COM 库函数。当进程不再需要使用 COM 库函数时，必须调用

CoUninitialize。对每一个进程，COM 库函数只需初始化一次。这并不是说不能多次调用 CoInitialize，但需保证每一个 CoInitialize 都有一个相应的 CoUninitialize 调用。当进程已经调用过 CoInitialize 后，再次调用此函数所得到的返回值将是 S\_FALSE 而不再是 S\_OK。

23、OLE 是建立在 COM 基础之上的，它增加了对类型库、剪贴板、拖放、ActiveX 文档、自动化以及 ActiveX 控件的支持。在 OLE 库中包含对这些特性的额外的支持。在需要使用这些特性时，应调用 OleInitialize 及 OleUninitialize,而不是 CoInitialize 和 CoUninitialize。Ole\*函数将调用 Co\*函数。但若程序中没有用到那些额外的功能，使用 Ole\*将会造成资源的浪费。

24、COM 中分配和释放内存的标准方法：任务内存分配器。使用此分配器，组件可以给客户提供一块可以由客户删除的内存。可在多线程应用程序中使用。

一些方便的函数：

```
void *CoTaskMemAlloc(
    ULONG cb //size in bytes of block to be allocated
);

void CoTaskMemFree(
    void *pv //pointer to memory block to be freed
);
```

25、StringFromGUID2 可以将某个 GUID 转换成一个字符串：

```
wchar_t szCLSID[39];

int r = ::StringFromGUID2(CLSID_Component1,szCLSID,39);
```

传给 StringFromGUID2 的参数是一个 Unicode 串（即一个宽字符 wchar\_t 类型的数组而不是 char 类型的字符数组）。在非 Unicode 的系统中，需要将结果转化为单字节字符(char)。为此，可以使用 ANSI 的 wcstombs 函数如下：

```
#ifndef _UNICODE

char szCLSID_single[39];

wcstombs(szCLSID_single,szCLSID,39);

#endif
```

26、

函数                      用途

StringFromCLSID          将 CLSID 转化成文本串

StringFromIID            将 IID 转化成文本串

StringFromGUID2          将 GUID 转化成文本串。此串将被存放在调用者所分配的缓冲区中

CLSIDFromString      将一个文本串转化成 CLSID

IIDFromString      将一个文本串转化成 IID

来自 CSDN 博客，转载请标明出处：<http://blog.csdn.net/wuyanhuishishi/archive/2005/01/13/251436.aspx>

## Engine 篇

什么是 OMD?

这个应该是我们最开始接触到的，我单独做了一个 ppt，马上合并在这个的后面。

一切都是接口?

在 Engine 中和我们打交道的都是接口，甚至包括返回的类型也是接口，到底什么是借口呢？相信在 oo 的学习中，书上都会这么说“接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为（功能），而这些属性和功能都是对外公开的”，仅仅这样理解行吗？我们知道在插件式开发的时候，开发支持插件功能的应用程序必须解决一个问题：如何在主程序与插件间正确地互相通信。为了在主程序与插件之间能正确地互相通信，应该先制定一套通信标准，这套通信标准就是接口，主程序与插件只能通过制订好的接口进行通信。软件开发中，接口只是定义功能并规定调用功能的形式，而不包含功能的实现。接口实质上是软件模块的调用规范。插件原理就是通过统一的程序接口来调用不同的模块，以实现不同功能的调用。用来扩充主程序的功能。

### The interface

<b>IFeatureClass : IObjectClass</b>
AreaField: IField
FeatureClassID: Long
FeatureDataset: IFeatureDataset
FeatureType: esriFeatureType
LengthField: IField
ShapeFieldName: String
ShapeType: tags.esriGeometryType
CreateFeature: IFeature
CreateFeatureBuffer: IFeatureBuffer
FeatureCount (in QueryFilter: IQueryFilter): Long
GetFeature (in id: Long): IFeature
GetFeatures (in fids: Variant, in Recycling: Boolean): IFeatureCursor
Insert (in useBuffering: Boolean): IFeatureCursor
Search (in Filter: IQueryFilter, in Recycling: Boolean): IFeatureCursor
Select (in QueryFilter: IQueryFilter, in selType: esriSelectionType, in selOption: esriSelectionOption, in selectionContainer: IWorkspace): ISelectionSet
Update (in Filter: IQueryFilter, in Recycling: Boolean): IFeatureCursor

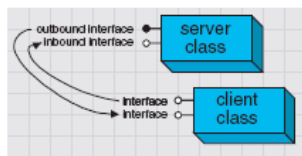
An interface is a specification of properties and methods. Many coclasses can implement the same interface. Interfaces allow a high degree of interoperability and shared behavior among a set of objects.

AreaField is a return property of type IField. FeatureClassID is of type Long.

The CreateFeature method creates an object of type IFeature. FeatureCount takes in a query filter and returns a long.

为什么有两种接口?





*In the diagrams in this book and the ArcObjects object model diagrams, outbound interfaces are depicted with a solid circle on the interface jack.*

or means by calling developers.

## INBOUND AND OUTBOUND INTERFACES

Interfaces can be either inbound or outbound. An inbound interface is the most common kind—the client makes calls to functions within the interface contained on an object. An outbound interface is one where the object makes calls to the client—a technique analogous to the traditional callback mechanism.

There are differences in the way these interfaces are implemented. The implementer of an inbound interface must implement all functions of the interface; failure to do so breaks the contract of COM. This is also true for outbound interfaces. If you use Visual Basic, you don't have to implement all functions present on the interface since it provides stub methods for the methods you don't implement. On the other hand, if you use C++ you must implement all the pure virtual functions to compile the class.

Connection points is a specific methodology for working with outbound COM interfaces. The connection point architecture defines how the communication between objects is set up and taken down. Connection points are not the most efficient way of initializing bidirectional object communication, but they are in common use because many development tools and environments support them.

COM 所建立的是一个软件模块与另一个软件模块之间的链接，当这种链接建立之后，模块之间就可以通过被称之为 Interface “接口” 的机制来进行通信。在绝大部分情况下，客户应用程序与组件的通信过程是单向的，客户创建组件对象，然后客户通过接口调用对象所提供的功能，在适当的时候再把对象释放掉。在这种交互过程中，客户总是主动的，而组件总是处于被动状态，通过自身暴露给客户的接口监听客户的请求，一旦接收到客户的请求便做出反应。这样的接口称为入接口 **incoming interface**，对于一个全面交互过程来说，这样的单向通信往往是不能满足实际的需要，组件对象也要主动与客户进行通信，因此，与入接口相对应，对象也可以提供出接口 **outgoing interface** 也叫回调接口，对象通过这些出接口与客户进行通信。之所以把这些接口称为出接口，其原因在于这些接口并不由 COM 服务器端的对象实现，而是由客户程序来实现，客户实现这些接口，并把接口指针通过一定的手段传给服务器，以后服务器端就利用此接口指针与客户进行通信，服务器端调用此接口的成员函数，即调用了客户自定义的函数，这时组件对象变成了客户端的客户，可见在 com 规范中，com 组件对象提供服务客户调用服务，这种对象与客户之间的关系是相对的。

因为 Com 编程类似客户端和服务端，由服务器处理，但是又有一些代码需要我们去写，尽管 COM 的内部被屏蔽掉了，当我们调用某一个函数的时候，类似于一个服务，从服务器端调用，他是怎么写的，我们不知道，但是我们通过这个方法的介绍却知道这个方法能干什么，但是这个交互似乎不强？所以就有另外的接口 **OutInterface**。

```
Private EngineEditor m_EngineEditor = new EngineEditorClass();

private IEngineEditEvents_Event m_EngineEditEvents;

private void MainForm_Load(object sender, EventArgs e)
{
    m_EngineEditEvents = (IEngineEditEvents_Event)m_EngineEditor;

    m_EngineEditEvents.OnStartEditing += new
    IEngineEditEvents_OnStartEditingEventHandler(OnStartEditingMethod);
}

private void OnStartEditingMethod()
{
    System.Windows.Forms.MessageBox.Show("Editing Started");
}
```

## ArcObjcets 开发技术中的一些关系?

<http://bbs.upgis.com/bbs/viewthread.php?tid=997>

ArcEngine 组件库的每一个组件中定义有不同的类,类下面定义了不同接口,接口中包含不同的属性和方法。类之间有类型继承 (TypeInheritance) 关系,接口之间有互相调用 (QueryInterface) 及相互继承 (InterfaceInheritance) 关系。

### 1.1 类与对象

在面向对象编程中,类和对象是两个非常重要的概念,可以这么说类就是创建对象的蓝本,而对象是指具有属性和动作的实体,它封装了一个客观实体的属性与行为。ArcObjcets 中有三类 class,分别是抽象类 (AbstractClass)、组件类 (CoClass) 和普通类 (Class)。抽象类的主要目的是为它的子类定义公共接口,一个抽象类将它的部分或全部实现延迟到子类中,因此,一个抽象类不能被实例化。一个组件类对象可以被直接创建,普通类对象虽然不能直接创建,但它可以作为其它类的一个属性或者从其它类的实例化来创建。

### 1.2 接口和类

接口定义了一组方法和属性,在 ArcObjects 中接口名称都以 "I" 开始,如 IMap, Ilayer 等。类实现了接口中的方法。一个类可以有多个接口,如 FeatureLayerClass 类有 IFeatureLayer, IFeatureSelection 等不同接口,而一个接口也可被多个类所拥有,如 CadFeatureLayer 类和 FeatureLayer 类都有 IFeatureLayer 接口。接口定义了能做什么,而定义了该怎么做 (The interface defines what an object can do, and the class defines how it is done. 79 页 Exploring ArcObjects V9.0),在 AO 开发的时候,和对象间的通信是通过接口完成的,而不是我们在一些其他面向对象语言如 (Java) 中和对象的通信是通过对象完成的。

### 1.3 接口查询(QueryInterface)

一个类可以有多个接口,声明了接口变量并且指向一个对象的时候,这个变量只能使用该接口内的方法和属性,而不能访问其他接口中的方法和属性,如:

```
Dim pMap as IMap
```

```
Set pMap = New Map
```

```
pMap.Clear '这里会产生错误
```

此时的 pMap 只能使用 IMap 接口中定义的方法和属性,比如获取图层的个数,添加图层等,但是不能清空视图上的内容 (因为这个方法是在 IActiveView 中定义的)

QueryInterface(QI) 很方便的让我们在一个类的不同接口间进行切换:

```
Dim pView as IActiveView
```

```
set pView= pMap 'QI
```

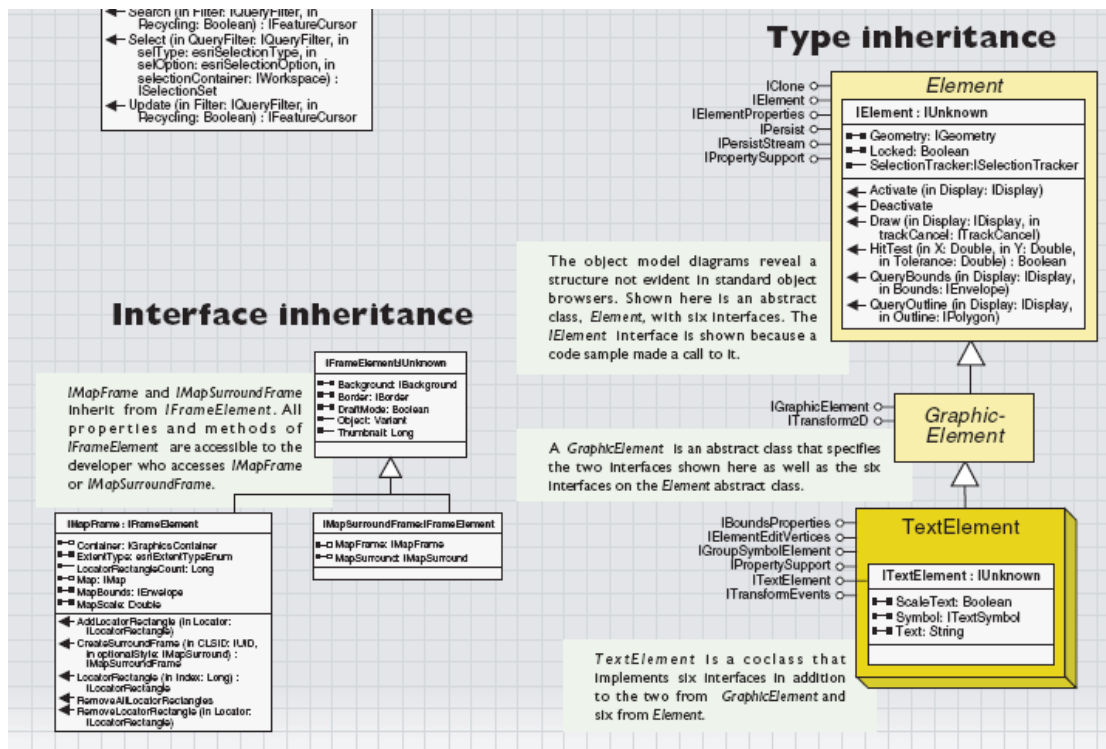
现在 pView 就可以使用 IActiveView 中定义的方法了。

### 1.4 类类型继承

类型继承是指类之间的接口类型的继承,而不是继承其实现。继承过来的接口只是名称相同,具体的实现则不同。比如 ShpfileWorkspaceFactory 和 AccessWorkspaceFactory 都继承 WorkspaceFactory,而他们的打开 (OpenFromFile) 方法却不一样,ShpfileWorkspaceFactory 的 (OpenFromFile) 方法需要一个文件目录位置作为参数,而 AccessWorkspaceFactory 的 (OpenFromFile) 方法需要一个数据库 (mdb) 位置作为参数。

### 1.5 接口继承

如 ImapFrame 接口和 IMapSurroundFrame 接口继承于 IFrameElement 接口,则父类接口 IFrameElement 所具有的方法和属性对派生接口 ImapFrame 和 IMapSurroundFrame 都有效。



### Engine 开发中有没有考虑过冲突？

用 AE 开发，其实就是接口编程，在 Java 和 C# 中都不支持多继承，但是支持多接口继承，我们知道如果类继承了接口，那么这个类就要无条件构造这个接口中所提供的方法，那么如果一个类继承了多个接口，如果这个类继承的多个接口中有方法重复怎么办，我们知道 C++ 中没有接口这个概念，而是有多继承，它为了解决重复，而提出了虚继承，而 C# 中是显式继承，这样就解决了这个重复问题。

(1) C# (以下是在《C#宝典》上看到的)：

在实际的应用中，由于种种原因，经常可能造成名称的冲突问题。对于接口来说，程序可能中存在一个接口 *IShape*，在 *IShape* 接口中定义了一个方法 *Caculate*；同样，C# 中允许存在另外一个接口 *I2DShape* 用于表示 2D 图形，同样也可以存在一个 *Caculate* 方法。但如果有一个类 *Circle* 同时继承了上述两个接口，则实现时会存在名称的冲突。

在 C# 中可以采用显式实现的方法解决名称冲突的问题，下面使用一个实例演示此类问题。

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Example10_10
{
    Class Program
    {
        static void Main(string[] args)
        {
            //创建 Circle 类变量 circle
            Circle circle = new Circle(35);
            //分别调用两个 Caculate 方法
            IShape shape = (IShape)circle;
            shape.Caculate();
            I2DShape shape2d = (I2DShape)circle;
```

```
shape2d.Caculate();
Console.ReadLine();
}
}
/// <summary>
/// IShape 接口
/// </summary>
interface IShape
{
    /// <summary>
    /// Area 属性
    /// </summary>
    int Area
    {
        get;
        set;
    }
    /// <summary>
    /// Caculate 方法
    /// </summary>
    void Caculate();
}
interface I2DShape
{
    /// <summary>
    /// Name 属性
    /// </summary>
    string Name
    {
        get;
        set;
    }
    /// <summary>
    /// Caculate 方法
    /// </summary>
    void Caculate();
}
/// <summary>
/// Circle 类继承 IShape
/// </summary>
Class Circle:IShape, I2DShape
{
    /// <summary>
    /// area 字段
```

```
/// </summary>
int area = 0;
/// <summary>
/// name 字段
/// </summary>
string name = string.Empty;
/// <summary>
/// 构造函数
/// </summary>
/// <param name="m_Area">m_Area 参数</param>
public Circle(int m_Area)
{
    area = m_Area;
}
#region I2DShape 成员
/// <summary>
/// Name 属性
/// </summary>
string I2DShape.Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
/// <summary>
/// Caculate 方法
/// </summary>
void I2DShape.Caculate()
{
    Console.WriteLine("I2DShape 的 Caculate 方法! ");
}
#endregion
#region IShape 成员
/// <summary>
/// Area 属性
/// </summary>
int IShape.Area
{
    get
```

```
{  
return area;  
}  
set  
{  
area = value;  
}  
}  
/// <summary>  
/// Caculate 方法  
/// </summary>  
void IShape.Caculate()  
{  
Console.WriteLine("IShape 的 Caculate 方法! ");  
}  
#endregion  
}  
}
```

程序运行结果如下。

IShape 的 Caculate 方法!

I2DShape 的 Caculate 方法!

程序中显式的写出了所继承的接口，如 IShape.Caculate 或 I2DShape.Caculate 进行实现，在调用时又重新进行了包装，如此便能解决接口多重继承时的名称冲突问题。

(2) C++

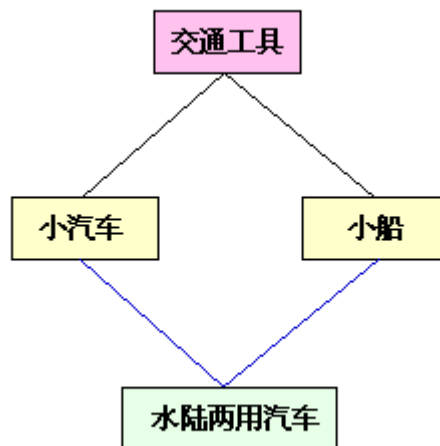
图文例解 C++类的多重继承与虚拟继承

[http://pcedu.pconline.com.cn/empolder/gj/c/0503/579115\\_1.html](http://pcedu.pconline.com.cn/empolder/gj/c/0503/579115_1.html)

在过去的学习中，我们始终接触的单个类的继承，但是在现实生活中，一些新事物往往会拥有两个或者两个以上事物的属性，为了解决这个问题，C++引入了多重继承的概念，**C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。**

举个例子，交通工具类可以派生出汽车和船连个子类，但拥有汽车和船共同特性水陆两用汽车就必须继承来自汽车类与船类的共同属性。

由此我们不难想出如下的图例与代码：



当一个派生类要使用多重继承的时候，必须在派生类名和冒号之后列出所有基类的类名，并用逗号分

隔。

//程序作者:管宁

//站点:[www.cndev-lab.com](http://www.cndev-lab.com)

//所有稿件均有版权, 如要转载, 请务必著名出处和作者

```
#include <iostream>
using namespace std;

class Vehicle
{
public:
    Vehicle(int weight = 0)
    {
        Vehicle::weight = weight;
    }
    void SetWeight(int weight)
    {
        cout<<"重新设置重量"<<endl;
        Vehicle::weight = weight;
    }
    virtual void ShowMe() = 0;
protected:
    int weight;
};

class Car:public Vehicle//汽车
{
public:
    Car(int weight=0, int aird=0):Vehicle(weight)
    {
```



```
        Car::aird = aird;
    }

    void ShowMe ()
    {
        cout<<"我是汽车!"<<endl;
    }

protected:
    int aird;
};

class Boat:public Vehicle//船
{
public:
    Boat(int weight=0, float tonnage=0):Vehicle(weight)
    {
        Boat::tonnage = tonnage;
    }

    void ShowMe ()
    {
        cout<<"我是船!"<<endl;
    }

protected:
    float tonnage;
};

class AmphibianCar:public Car, public Boat//水陆两用汽车, 多重继承的体现
{
public:
    AmphibianCar(int weight, int aird, float tonnage)
        :Vehicle(weight), Car(weight, aird), Boat(weight, tonnage)
        //多重继承要注意调用基类构造函数
    {

    }

    void ShowMe ()
    {
        cout<<"我是水陆两用汽车!"<<endl;
    }

};

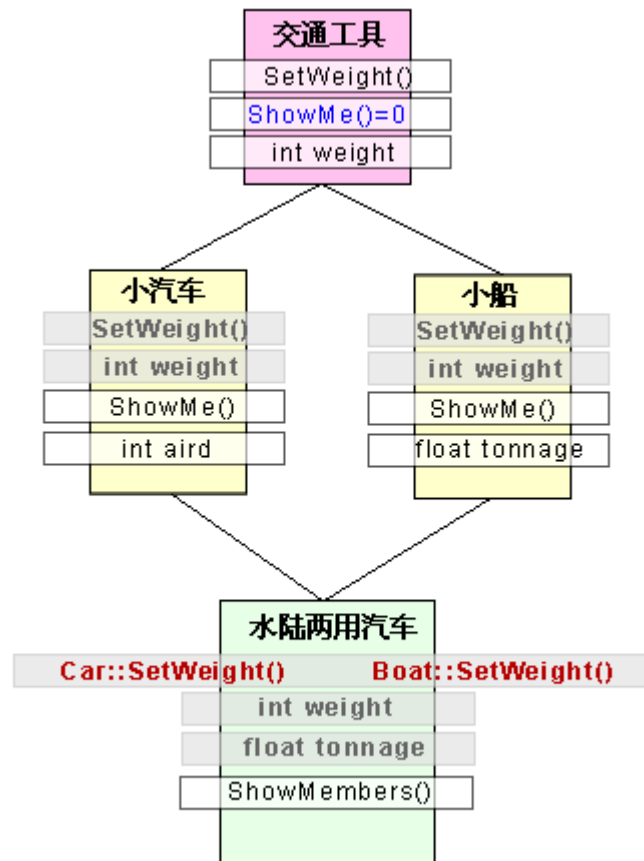
int main()
{
    AmphibianCar a(4, 200, 1.35f); //错误
    a.SetWeight(3); //错误
```

```
system("pause");
}
```

上面的代码从表面看，看不出有明显的语法错误，但是它是不能够通过编译的。这有是为什么呢？

这是由于多重继承带来的**继承的模糊性**带来的问题。

先看如下的图示：



在图中深红色标记出来的地方正是主要问题所在，水陆两用汽车类继承了来自 Car 类与 Boat 类的属性与方法，Car 类与 Boat 类同为 AmphibianCar 类的基类，在内存分配上 AmphibianCar 获得了来自两个类的 SetWeight() 成员函数，当我们调用 a.SetWeight(3) 的时候计算机不知道如何选择分别属于两个基类的被重复拥有了的类成员函数 SetWeight()。

由于这种模糊问题的存在同样也导致了 AmphibianCar a(4, 200, 1.35f); 执行失败，系统会产生“Vehicle”不是基或成员的错误。

以上面的代码为例，我们要想让 AmphibianCar 类既获得一个 Vehicle 的拷贝，而且又同时共享 Car 类与 Boat 类的数据成员与成员函数就必须通过 C++ 所提供的**虚拟继承**技术来实现。

我们在 Car 类和 Boat 类继承 Vehicle 类出，在前面加上 virtual 关键字就可以实现虚拟继承，使用虚拟继承后，**当系统碰到多重继承的时候就会自动先加入一个 Vehicle 的拷贝，当再次请求一个 Vehicle 的拷贝的时候就会被忽略，保证继承类成员函数的唯一性。**

修改后的代码如下，注意观察变化：

```
#include <iostream>
using namespace std;

class Vehicle
```

```
{
    public:
        Vehicle(int weight = 0)
        {
            Vehicle::weight = weight;
            cout<<"载入 Vehicle 类构造函数"<<endl;
        }
        void SetWeight(int weight)
        {
            cout<<"重新设置重量"<<endl;
            Vehicle::weight = weight;
        }
        virtual void ShowMe() = 0;
    protected:
        int weight;
    };
class Car:virtual public Vehicle//汽车, 这里是虚拟继承
{
    public:
        Car(int weight=0, int aird=0):Vehicle(weight)
        {
            Car::aird = aird;
            cout<<"载入 Car 类构造函数"<<endl;
        }
        void ShowMe()
        {
            cout<<"我是汽车! "<<endl;
        }
    protected:
        int aird;
    };

class Boat:virtual public Vehicle//船, 这里是虚拟继承
{
    public:
        Boat(int weight=0, float tonnage=0):Vehicle(weight)
        {
            Boat::tonnage = tonnage;
            cout<<"载入 Boat 类构造函数"<<endl;
        }
        void ShowMe()
        {
            cout<<"我是船! "<<endl;
        }
}
```

```

        protected:
        float tonnage;
    };

class AmphibianCar:public Car,public Boat//水陆两用汽车, 多重继承的体现
{
    public:
        AmphibianCar(int weight, int aird, float tonnage)
        :Vehicle(weight), Car(weight, aird), Boat (weight, tonnage)
        //多重继承要注意调用基类构造函数
        {
            cout<<"载入 AmphibianCar 类构造函数"<<endl;
        }
        void ShowMe ()
        {
            cout<<"我是水陆两用汽车!"<<endl;
        }
        void ShowMembers ()
        {
            cout<<"重量: "<<weight<<"吨, "<<"空气排量: "<<aird<<"CC, "<<"排水量: "<<tonnage<<"
            吨"<<endl;
        }
    };

int main()
{
    AmphibianCar a(4, 200, 1.35f);
    a.ShowMe();
    a.ShowMembers();
    a.SetWeight(3);
    a.ShowMembers();
    system("pause");
}

```

注意观察类构造函数的构造顺序。

虽然说虚拟继承与虚函数有一定相似的地方，但读者务必要记住，他们之间是绝对没有任何联系的！

返回一个接口，这个接口应该是指向一个具体的对象的？

接口对类的内部分区管理，而 **ITopologyContainer::CreateTopology**，返回 ITopology 接口，这个接口就是 Topology 对象所继承的一个，这样就可以跳转既然都是指向 Topology 这个对象，那么我们就可以在同一个对象的不同接口间访问（QI）用 Engine help 上的话说 就是 cast。但是如果一个接口被多个类继承，那么返回的这个接口到底是哪个对象的呢？我们不知道他内部是怎么的，但是我们可以知道，他在内部肯定 new 了一个对象（也不一定是 New, create 等也可以），然后赋给这个要返回的接口，这个接口肯定在这个时候是属于某一个具体的对象的。

QI 之我见？

<http://bbs.upgis.com/bbs/viewthread.php?tid=292>

说起来惭愧，我是大三的时候接触 AE, 里面的 as 语句都要把我弄疯了，因为我一直把里面的接口和这个 as 没有理解，在单纯的 C# 中这个应该是强制转化。当然强制转化的方式有好几种，我只说到了 as，其余的可以参考相关书目。

在 AE 中 as 和接口是很常用的词语，同时给像我这样的初级人员带来了困惑，as 在 AE 中到底充当了设么角色，我个人觉得在 AE 中把 as 翻译成强制转化不太合适，还是 QI 比较好。

简单说一下我对 QI, QI(query interface) 的理解：

在 COM 技术中，对象似乎被屏蔽掉了，和我们打交道的都是 interface，不像 Java 中和我们打交道的都是对象，但是我们知道类是方法和属性的容器，而 interface 相当于一个类的内部管理。我们用接口变量和其他的对象打交道，我觉得把这个理解成指向对象的接口变量比较好理解，因为我们用 new 关键字实例化一个对象，计算机就给这个对象分配空间（内存），我们知道一个类可以继承多个接口，而把这个初始化了的对象赋给一个接口变量，这个接口变量只能使用这个类中所继承的这个接口所定义的方法和属性，而不能使用其他接口中所定义的方法和变量，在 Java 中也有过接口回调这个说法，似乎和这个一样，我记得老师是这样说的 把这个对象赋给接口变量，那么这个接口变量就可以调用这个类中实现的这个接口中的方法。这是 java 课上老师讲的一个例子

```
interface People {  
    void gender();  
}  
  
class Boy implements People {  
    public void gender() {  
        System.out.println("I' m a boy.");  
    }  
}  
  
class Girl implements People {  
    public void gender() {  
        System.out.println("I' m a girl.");  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        People Person; // 声明接口变量  
        Person = new Boy(); // 实例化，接口变量中存放对象的引用  
        Person.gender(); // 接口回调  
        Person = new Girl(); // 实例化，接口变量中存放对象的引用
```

```
        Person.gender(); // 接口回调
    }
}
```

有兴趣的可以把这个代码在自己的机器上运行一下，看看结果和自己想的是不是一样。

但是一个类实现了很多接口，我们要用到其他的接口中的方法怎么办？重新 new 一个对象，我们知道 new 了之后 计算机重新分出一个内存空间。举个例子说明一下： 这是在分类渲染的时候要用到一些东西

```
ITableHistogram pTableHist;//相当于一个统计表
IBasicHistogram pBasicHist;//这个对象有一个很重要的方法
pTable = pGeolayer as ITable;

pBasicHist = new BasicTableHistogramClass();//也可以实例化 pTableHist。
//pTableHist=new BasicTableHistogramClass();
//pBasicHist =pTableHist as IBasicHistogram;QI 的灵活性就在这里。

pTableHist = pBasicHist as ITableHistogram;
pTableHist.Table = pTable;
pTableHist.Field = Field;
object dataavalus;
object Frenquen;

pBasicHist.GetHistogram(out dataavalus,out Frenquen);//获得数据和相应的频数。
```

pBasicHist.GetHistogram 就是对 pTableHist 操作，如果在 new 一个，那么刚才的一些操作(pTableHist)就不起作用了（因为 pBasicHist, pTableHist 就不是指向同一个对象了），因为我们要在同一个对象内操作，所以 QI 就大显身手，实现了在同一个对象见不同接口之间的切换。也可以这样认为 pTableHist 这个接口变量是对 BasicTableHistogramClass 对象里面相关属性的初始化（变量赋值），而 pBasicHist 接口的 GetHistogram（）方法是对上述变量的操作。如果一个庞大的类里面有好多属性和方法，显得杂乱无章，不便于管理，所以接口就应运而生，接口把类里面相关的属性和方法又重新化为一个集合，也可以这么说接口对类又重新划分了一下。有的接口是对类里面一些属性进行赋值，而另外一些接口是对这些赋值了的属性的操作谓之方法。

在 AE 的学习中，我觉得类就像一个公司一样，而类实现的接口相当于公司的办公室，这些办公室的所有功能加起来就是这个公司的职能。而要进入某一个类，那么就要知道这个类继承了那些接口，接口似乎是我们进入类的通道。

再看下面的英文描述：

ArcGIS Developer Help (ESRI.ArcGIS.Geodatabase) FeatureClassDescriptionClass Class

ESRI Feature Class Description Object.

Product AvailabilityAvailable with ArcGIS Engine, ArcGIS Desktop, and ArcGIS Server.

Supported Platforms Windows, Solaris, Linux

InterfacesDescription Provides access to members that control Feature Class Description.

IObjectClassDescription Provides access to members that control Object Class Description.

由上面的可以看出，我们要多了解一些类和一些接口的关系非常那个清楚才行。

## 面向接口编程详解（做 AE 开发应该可以用的到）

<http://bbs.esrichina-bj.cn/ESRI/thread-57806-1-1.html>

### 1. 面向接口编程和面向对象编程是什么关系

首先，面向接口编程和面向对象编程并不是平级的，它并不是比面向对象编程更先进的一种独立的编程思想，而是附属于面向对象思想体系，属于其一部分。或者说，它是面向对象编程体系中的思想精髓之一。

### 2. 接口的本质

接口，在表面上是由几个没有主体代码的方法定义组成的集合体，有唯一的名称，可以被类或其他接口所实现（或者也可以说继承）。它在形式上可能是如下的样子：

```
interface InterfaceName
{
    void Method1();
    void Method2(int para1);
    void Method3(string para2,string para3);
}
```

那么，接口的本质是什么呢？或者说接口存在的意义是什么。我认为可以从以下两个视角考虑：

1）接口是一组规则的集合，它规定了实现本接口的类或接口必须拥有的一组规则。体现了自然界“如果你是.....则必须能.....”的理念。

例如，在自然界中，人都能吃饭，即“如果你是人，则必须能吃饭”。那么模拟到计算机程序中，就应该有一个 IPerson（习惯上，接口名由“I”开头）接口，并有一个方法叫 Eat（），然后我们规定，每一个表示“人”的类，必须实现 IPerson 接口，这就模拟了自然界“如果你是人，则必须能吃饭”这条规则。

从这里，我想各位也能看到些许面向对象思想的东西。面向对象思想的核心之一，就是模拟真实世界，把真实世界中的事物抽象成类，整个程序靠各个类的实例互相通信、互相协作完成系统功能，这非常符合真实世界的运行状况，也是面向对象思想的精髓。

2）接口是在一定粒度视图上同类事物的抽象表示。注意这里我强调了在一定粒度视图上，因为“同类事物”这个概念是相对的，它因为粒度视图不同而不同。

例如，在我的眼里，我是一个人，和一头猪有本质区别，我可以接受我和我同学是同类这个说法，但绝不能接受我和一头猪是同类。但是，如果在一个动物学家眼里，我和猪应该是同类，因为我们都是动物，他可以认为“人”和“猪”都实现了 IAnimal 这个接口，而他在研究动物行为时，不会把我和猪分开对待，而会从“动物”这个较大的粒度上研究，但他会认为我和一棵树有本质区别。

现在换了一个遗传学家，情况又不同了，因为生物都能遗传，所以在他眼里，我不仅和猪没区别，和一只蚊子、一个细菌、一颗树、一个蘑菇乃至一个 SARS 病毒都没什么区别，因为他会认为我们都实现了 IDescendable 这个接口（注：descend vi. 遗传），即我们都是可遗传的东西，他不会分别研究我们，而会将所有生物作为同类进行研究，在他眼里没有人和病毒之分，只有可遗传的物质和不可遗传的物质。但至少，我和一块石头还是有区别的。

可不幸的事情发生了，某日，地球上出现了一位伟大的人，他叫列宁，他在熟读马克思、恩格斯的辩证唯物主义思想巨著后，颇有心得，于是他下了一个著名的定义：所谓物质，就是能被意识所反映的客观



实在。至此，我和一块石头、一丝空气、一条成语和传输手机信号的电磁场已经没什么区别了，因为在列宁的眼里，我们都是可以被意识所反映的客观实在。如果列宁是一名程序员，他会这么说：所谓物质，就是所有同时实现了“`IReflectable`”和“`IEsse`”两个接口的类所生成的实例。（注：`reflect` v. 反映 `esse` n. 客观实在）

也许你会觉得我上面的例子像在瞎掰，但是，这正是接口得以存在的意义。面向对象思想和核心之一叫做多态性，什么叫多态性？说白了就是在某个粒度视图层面上对同类事物不加区别的对待而统一处理。而之所以敢这样做，就是因为有接口的存在。像那个遗传学家，他明白所有生物都实现了 `IDescendable` 接口，那只要是生物，一定有 `Descend()` 这个方法，于是他就可以统一研究，而不至于分别研究每一种生物而最终累死。

可能这里还不能给你一个关于接口本质和作用的直观印象。那么在后文的例子和对几个设计模式的解析中，你将会更直观体验到接口的内涵。

### 3. 面向接口编程综述

通过上文，我想大家对接口和接口的思想内涵有了一个了解，那么什么是面向接口编程呢？我个人的定义是：在系统分析和架构中，分清层次和依赖关系，每个层次不是直接向其上层提供服务（即不是直接实例化在上层中），而是通过定义一组接口，仅向上层暴露其接口功能，上层对于下层仅仅是接口依赖，而不依赖具体类。

这样做的好处是显而易见的，首先对系统灵活性大有好处。当下层需要改变时，只要接口及接口功能不变，则上层不用做任何修改。甚至可以在不改动上层代码时将下层整个替换掉，就像我们将一个 **WD** 的 **60G** 硬盘换成一个希捷的 **160G** 的硬盘，计算机其他地方不用做任何改动，而是把原硬盘拔下来、新硬盘插上就行了，因为计算机其他部分不依赖具体硬盘，而只依赖一个 **IDE** 接口，只要硬盘实现了这个接口，就可以替换上去。从这里看，程序中的接口和现实中的接口极为相似，所以我一直认为，接口（`interface`）这个词用的真是神似！

使用接口的另一个好处就是不同部件或层次的开发人员可以并行开工，就像造硬盘的不用等造 **CPU** 的，也不用等造显示器的，只要接口一致，设计合理，完全可以并行进行开发，从而提高效率。

本篇文章先到这里。最后我想再啰嗦一句：面向对象的精髓是模拟现实，这也可以说是我这篇文章的灵魂。所以，多从现实中思考面向对象的东西，对提高系统设计能力大有裨益。

下篇文章，我将用一个实例来展示接口编程的基本方法。

而第三篇，我将解析经典设计模式中的一些面向接口编程思想，并解析一下 **.NET** 分层架构中的面向接口思想。

对本文的补充：

仔细看了各位的回复，非常高兴能和大家一起讨论技术问题。感谢给出肯定的朋友，也要感谢提出意见和质疑的朋友，这促使我更深入思考一些东西，希望能借此进步。在这里我想补充一些东西，以讨论一些回复中比较集中的问题。

#### 1. 关于“面向接口编程”中的“接口”与具体面向对象语言中“接口”两个词

看到有朋友提出“面向接口编程”中的“接口”二字应该比单纯编程语言中的 `interface` 范围更大。我经过思考，觉得很有道理。这里我写的确实不太合理。我想，面向对象语言中的“接口”是指具体的一种代码结构，例如 **C#** 中用 `interface` 关键字定义的接口。而“面向接口编程”中的“接口”可以说是一种从软件架构的角度、从一个更抽象的层面上指那种用于隐藏具体底层类和实现多态性的结构部件。从这个意义上说，如果定义一个抽象类，并且目的是为了实现在多态，那么我认为把这个抽象类也称为“接口”是合理的。但是用抽象类实现多态合理不合理？在下面第二条讨论。

概括来说，我觉得两个“接口”的概念既相互区别又相互联系。“面向接口编程”中的接口是一种思想层面的用于实现多态性、提高软件灵活性和可维护性的架构部件，而具体语言中的“接口”是将这种思想中的部件具体实施到代码里的手段。

#### 2. 关于抽象类与接口

看到回复中这是讨论的比较激烈的一个问题。很抱歉我考虑不周没有在文章中讨论这个问题。我个人对这个问题的理解如下：

如果单从具体代码来看，对这两个概念很容易模糊，甚至觉得接口就是多余的，因为单从具体功能来看，除多重继承外（C#，Java 中），抽象类似乎完全能取代接口。但是，难道接口的存在是为了实现多重继承？当然不是。我认为，抽象类和接口的区别在于使用动机。使用抽象类是为了代码的复用，而使用接口的动机是为了实现多态性。所以，如果你在为某个地方该使用接口还是抽象类而犹豫不决时，那么可以想想你的动机是什么。

看到有朋友对 IPerson 这个接口的质疑，我个人的理解是，IPerson 这个接口该不该定义，关键看具体应用中是怎么个情况。如果我们的项目中有 Women 和 Man，都继承 Person，而且 Women 和 Man 绝大多数方法都相同，只有一个方法 DoSomethingInWC（）不同（例子比较粗俗，各位见谅），那么当然定义一个 AbstractPerson 抽象类比较合理，因为它可以把其他所有方法都包含进去，子类只定义 DoSomethingInWC（），大大减少了重复代码量。

但是，如果我们程序中的 Women 和 Man 两个类基本没有共同代码，而且有一个 PersonHandle 类需要实例化他们，并且不希望知道他们是男是女，而只需把他们当作人看待，并实现多态，那么定义成接口就有必要了。

总而言之，接口与抽象类的区别主要在于使用的动机，而不在于其本身。而一个东西该定义成抽象类还是接口，要根据具体环境的上下文决定。

再者，我认为接口和抽象类的另一个区别在于，抽象类和它的子类之间应该是一般和特殊的关系，而接口仅仅是它的子类应该实现的一组规则。（当然，有时也可能存在一般与特殊的关系，但我们使用接口的目的不在这里）如，交通工具定义成抽象类，汽车、飞机、轮船定义成子类，是可以接受的，因为汽车、飞机、轮船都是一种特殊的交通工具。再譬如 Icomparable 接口，它只是说，实现这个接口的类必须要可以进行比较，这是一条规则。如果 Car 这个类实现了 Icomparable，只是说，我们的 Car 中有一个方法可以对两个 Car 的实例进行比较，可能是比哪辆车更贵，也可能比哪辆车更大，这都无所谓，但我们不能说“汽车是一种特殊的可以比较”，这在文法上都不通。

问题的提出

---

定义：现在我们要开发一个应用，模拟移动存储设备的读写，即计算机与 U 盘、MP3、移动硬盘等设备进行数据交换。

上下文（环境）：已知要实现 U 盘、MP3 播放器、移动硬盘三种移动存储设备，要求计算机能同这三种设备进行数据交换，并且以后可能会有新的第三方的移动存储设备，所以计算机必须有扩展性，能与目前未知而以后可能会出现存储设备进行数据交换。

各个存储设备间读、写的实现方法不同，U 盘和移动硬盘只有这两个方法，MP3Player 还有一个 PlayMusic 方法。

名词定义：数据交换={读，写}

看到上面的问题，我想各位脑子中一定有了不少想法，这是个很好解决的问题，很多方案都能达到效果。下面，我列举几个典型的方案。

解决方案列举

---

方案一：分别定义 FlashDisk、MP3Player、MobileHardDisk 三个类，实现各自的 Read 和 Write 方法。然后在 Computer 类中实例化上述三个类，为每个类分别写读、写方法。例如，为 FlashDisk 写 ReadFromFlashDisk、WriteToFlashDisk 两个方法。总共六个方法。

方案二：定义抽象类 MobileStorage，在里面写虚方法 Read 和 Write，三个存储设备继承此抽象类，并重写 Read 和 Write 方法。Computer 类中包含一个类型为 MobileStorage 的成员变量，并为其编写 get/set 器，

这样 `Computer` 中只需要两个方法：`ReadData` 和 `WriteData`，并通过多态性实现不同移动设备的读写。

方案三：与方案二基本相同，只是不定义抽象类，而是定义接口 `IMobileStorage`，移动存储器类实现此接口。`Computer` 中通过依赖接口 `IMobileStorage` 实现多态性。

方案四：定义接口 `IReadable` 和 `IWritable`，两个接口分别只包含 `Read` 和 `Write`，然后定义接口 `IMobileStorage` 接口继承自 `IReadable` 和 `IWritable`，剩下的实现与方案三相同。

下面，我们来分析一下以上四种方案：

首先，方案一最直白，实现起来最简单，但是它有一个致命的弱点：可扩展性差。或者说，不符合“开放-关闭原则”（注：意为对扩展开放，对修改关闭）。当将来有了第三方扩展移动存储设备时，必须对 `Computer` 进行修改。这就如在一个真实的计算机上，为每一种移动存储设备实现一个不同的插口、并分别有各自的驱动程序。当有了一种新的移动存储设备后，我们就要将计算机大卸八块，然后增加一个新的插口，在编写一套针对此新设备的驱动程序。这种设计显然不可取。

此方案的另一个缺点在于，冗余代码多。如果有 100 种移动存储，那我们的 `Computer` 中岂不是要至少写 200 个方法，这是不能接受的！

我们再来看方案二和方案三，之所以将这两个方案放在一起讨论，是因为他们基本是一个方案（从思想层面上来说），只不过实现手段不同，一个是使用了抽象类，一个是使用了接口，而且最终达到的目的应该是一样的。

我们先来评价这种方案：首先它解决了代码冗余的问题，因为可以动态替换移动设备，并且都实现了共同的接口，所以不管有多少种移动设备，只要一个 `Read` 方法和一个 `Write` 方法，多态性就帮我们解决问题了。而对第一个问题，由于可以运行时动态替换，而不必将移动存储类硬编码在 `Computer` 中，所以有了新的第三方设备，完全可以替换进去运行。这就是所谓的“依赖接口，而不是依赖与具体类”，不信你看看，`Computer` 类只有一个 `MobileStorage` 类型或 `IMobileStorage` 类型的成员变量，至于这个变量具体是什么类型，它并不知道，这取决于我们在运行时给这个变量的赋值。如此一来，`Computer` 和移动存储器类的耦合度大大下降。

那么这里该选抽象类还是接口呢？还记得第一篇文章我对抽象类和接口选择的建议吗？看动机。这里，我们的动机显然是实现多态性而不是为了代码复用，所以当然要用接口。

最后我们再来看一看方案四，它和方案三很类似，只是将“可读”和“可写”两个规则分别抽象成了接口，然后让 `IMobileStorage` 再继承它们。这样做，显然进一步提高了灵活性，但是，这有没有设计过度的嫌疑呢？我的观点是：这要看具体情况。如果我们的应用中可能会出现一些类，这些类只实现读方法或只实现写方法，如只读光盘，那么这样做也是可以的。如果我们知道以后出现的東西都是能读又能写的，那这两个接口就没有必要了。其实如果将只读设备的 `Write` 方法留空或抛出异常，也可以不要这两个接口。总之一句话：理论是死的，人是活的，一切从现实需要来，防止设计不足，也要防止设计过度。

在这里，我们姑且认为以后的移动存储都是能读又能写的，所以我们选方案三。

实现

下面，我们要将解决方案加以实现。我选择的语言是 `C#`，但是在代码中不会用到 `C#` 特有的性质，所以使用其他语言的朋友一样可以参考。

首先编写 `IMobileStorage` 接口：

```
namespace InterfaceExample
{
    public interface IMobileStorage
    {
        void Read();//从自身读数据
        void Write();//将数据写入自身
    }
}
```

```
}
```

比较简单，只有两个方法，没什么好说的，接下来是三个移动存储设备的具体实现代码：

U 盘

```
namespace InterfaceExample
```

```
{  
    public class FlashDisk : IMobileStorage  
    {  
        public void Read()  
        {  
            Console.WriteLine("Reading from FlashDisk.....");  
            Console.WriteLine("Read finished!");  
        }  
        public void Write()  
        {  
            Console.WriteLine("Writing to FlashDisk.....");  
            Console.WriteLine("Write finished!");  
        }  
    }  
}
```

MP3

```
namespace InterfaceExample
```

```
{  
    public class MP3Player : IMobileStorage  
    {  
        public void Read()  
        {  
            Console.WriteLine("Reading from MP3Player.....");  
            Console.WriteLine("Read finished!");  
        }  
        public void Write()  
        {  
            Console.WriteLine("Writing to MP3Player.....");  
            Console.WriteLine("Write finished!");  
        }  
        public void PlayMusic()  
        {  
            Console.WriteLine("Music is playing.....");  
        }  
    }  
}
```

移动硬盘

```
namespace InterfaceExample
```

```
{  
    public class MobileHardDisk : IMobileStorage
```

```
{
    public void Read()
    {
        Console.WriteLine("Reading from MobileHardDisk.....");
        Console.WriteLine("Read finished!");
    }
    public void Write()
    {
        Console.WriteLine("Writing to MobileHardDisk.....");
        Console.WriteLine("Write finished!");
    }
}
```

可以看到，它们都实现了 `IMobileStorage` 接口，并重写了各自不同的 `Read` 和 `Write` 方法。下面，我们来写 `Computer`：

```
namespace InterfaceExample
{
    public class Computer
    {
        private IMobileStorage _usbDrive;
        public IMobileStorage UsbDrive
        {
            get
            {
                return this._usbDrive;
            }
            set
            {
                this._usbDrive = value;
            }
        }
        public Computer()
        {
        }
        public Computer(IMobileStorage usbDrive)
        {
            this.UsbDrive = usbDrive;
        }

        public void ReadData()
        {
            this._usbDrive.Read();
        }
        public void WriteData()
```

```
{  
    this._usbDrive.Write();  
}  
}  
}
```

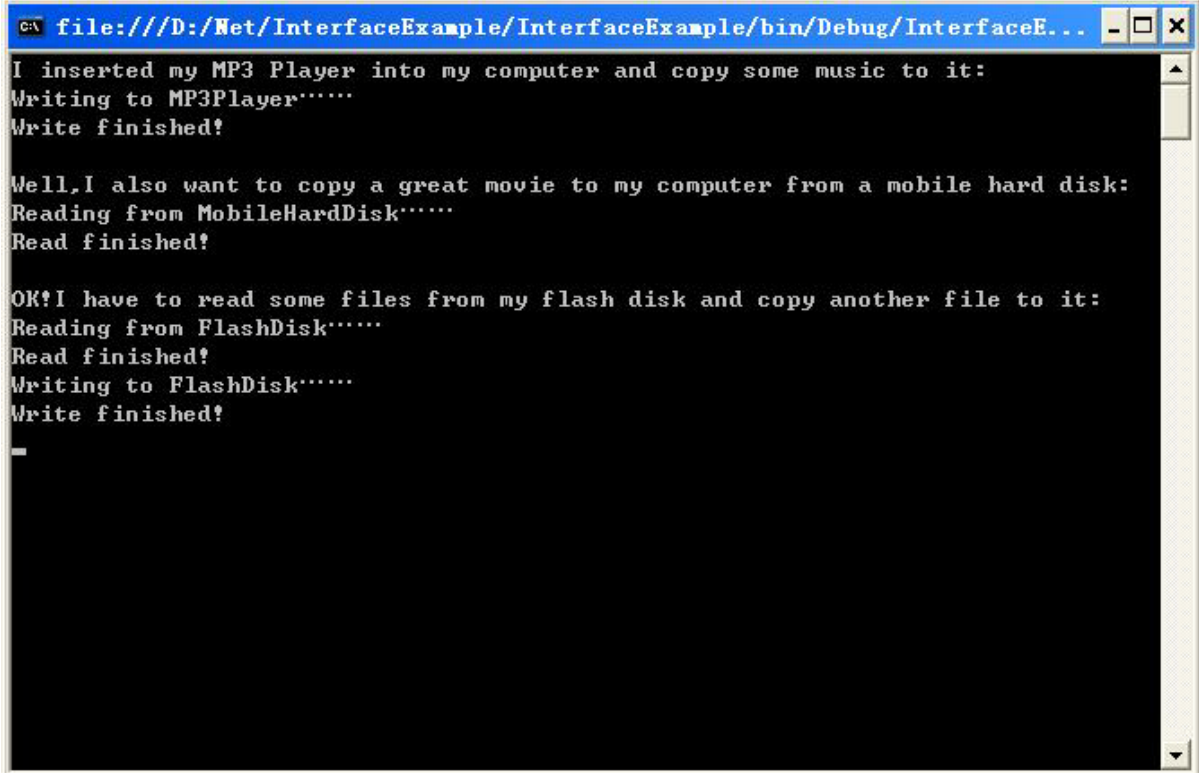
其中的 UsbDrive 就是可替换的移动存储设备，之所以用这个名字，是为了让大家觉得直观，就像我们平常使用电脑上的 USB 插口插拔设备一样。

OK! 下面我们来测试我们的“电脑”和“移动存储设备”是否工作正常。我是用的 C#控制台程序，具体代码如下：

```
namespace InterfaceExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Computer computer = new Computer();  
            IMobileStorage mp3Player = new MP3Player();  
            IMobileStorage flashDisk = new FlashDisk();  
            IMobileStorage mobileHardDisk = new MobileHardDisk();  
            Console.WriteLine("I inserted my MP3 Player into my computer and copy some music to  
it:");  
            computer.UsbDrive = mp3Player;  
            computer.WriteData();  
            Console.WriteLine();  
            Console.WriteLine("Well, I also want to copy a great movie to my computer from a mobile  
hard disk:");  
            computer.UsbDrive = mobileHardDisk;  
            computer.ReadData();  
            Console.WriteLine();  
            Console.WriteLine("OK! I have to read some files from my flash disk and copy another  
file to it:");  
            computer.UsbDrive = flashDisk;  
            computer.ReadData();  
            computer.WriteData();  
            Console.ReadLine();  
        }  
    }  
}
```

}

现在编译、运行程序，如果没有问题，将看到如下运行结果：



```
file:///D:/Net/InterfaceExample/InterfaceExample/bin/Debug/InterfaceE...
I inserted my MP3 Player into my computer and copy some music to it:
Writing to MP3Player.....
Write finished!

Well,I also want to copy a great movie to my computer from a mobile hard disk:
Reading from MobileHardDisk.....
Read finished!

OK!I have to read some files from my flash disk and copy another file to it:
Reading from FlashDisk.....
Read finished!
Writing to FlashDisk.....
Write finished!
-
```

好的，看来我们的系统工作良好。

后来……

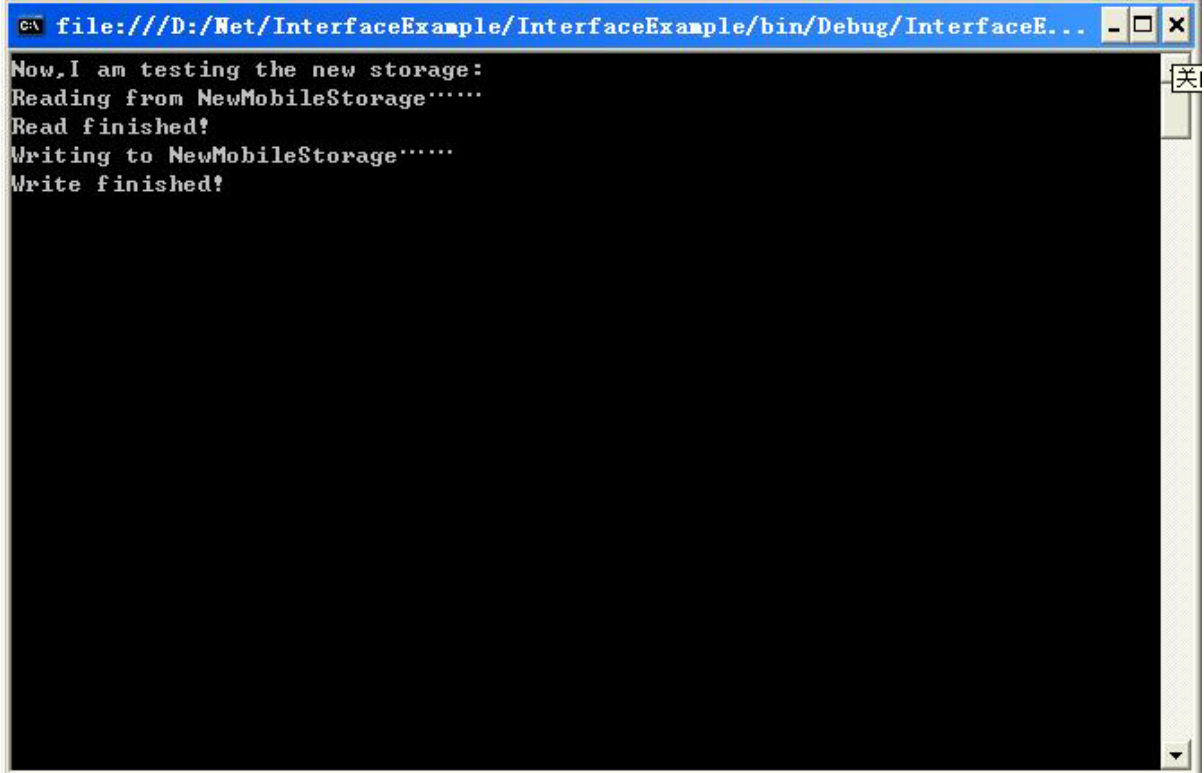
刚过了一个星期，就有人送来了新的移动存储设备 NewMobileStorage，让我测试能不能用，我微微一笑，心想这不是小菜一碟，让我们看看面向接口编程的威力吧！将测试程序修改成如下：

```
namespace InterfaceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Computer computer = new Computer();
            IMobileStorage newMobileStorage = new NewMobileStorage();
            Console.WriteLine("Now,I am testing the new mobile storage:");
            computer.UsbDrive = newMobileStorage;
            computer.ReadData();
        }
    }
}
```



```
computer.WriteData();  
Console.ReadLine();  
}  
}  
}
```

编译、运行、看结果：



哈哈，神奇吧，Computer 一点都不用改动，就可以使新的设备正常运行。这就是所谓“对扩展开放，对修改关闭”。

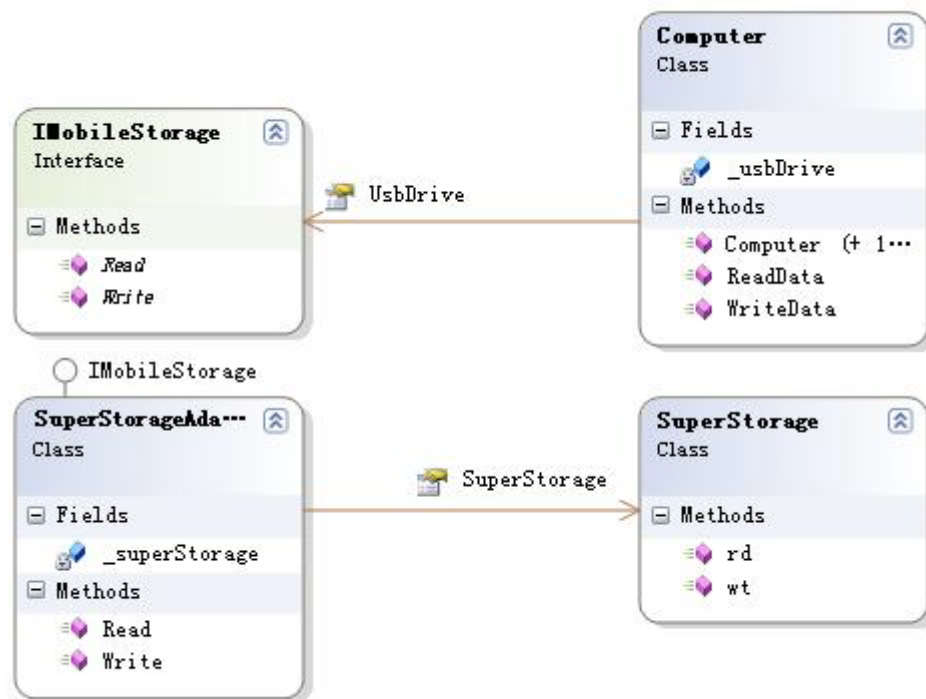
又过了几天，有人通知我说又有一个叫 SuperStorage 的移动设备要接到我们的 Computer 上，我心想来吧，管你是“超级存储”还是“特级存储”，我的“面向接口编程大法”把你们统统搞定。

但是，当设备真的送来，我傻眼了，开发这个新设备的团队没有拿到我们的 IMobileStorage 接口，自然也没有遵照这个约定。这个设备的读、写方法不叫 Read 和 Write，而是叫 rd 和 wt，这下完了……不符合接口啊，插不上。但是，不要着急，我们回到现实来寻找解决的办法。我们一起想想：如果你的 Computer 上只有 USB 接口，而有人拿来一个 PS/2 的鼠标要插上用，你该怎么办？想起来了，是不是有一种叫“PS/2-USB”转换器的东西？也叫适配器，可以进行不同接口的转换。对了！程序中也有转换器。

这里，我要引入一个设计模式，叫“Adapter”。它的作用就如现实中的适配器一样，把接口不一致的两个插件接合起来。由于本篇不是讲设计模式的，而且 Adapter 设计模式很好理解，所以我就不细讲了，



先来看我设计的类图吧：



如图所示，虽然 SuperStorage 没有实现 IMobileStorage，但我们定义了一个实现 IMobileStorage 的 SuperStorageAdapter，它聚合了一个 SuperStorage，并将 rd 和 wt 适配为 Read 和 Write，

SuperStorageAdapter 的具体代码如下：

```

namespace InterfaceExample
{
    public class SuperStorageAdapter : IMobileStorage
    {
        private SuperStorage _superStorage;

        public SuperStorage SuperStorage
        {
            get
            {
                return this._superStorage;
            }
            set
            {
                this._superStorage = value;
            }
        }
    }
}

```

```
    }

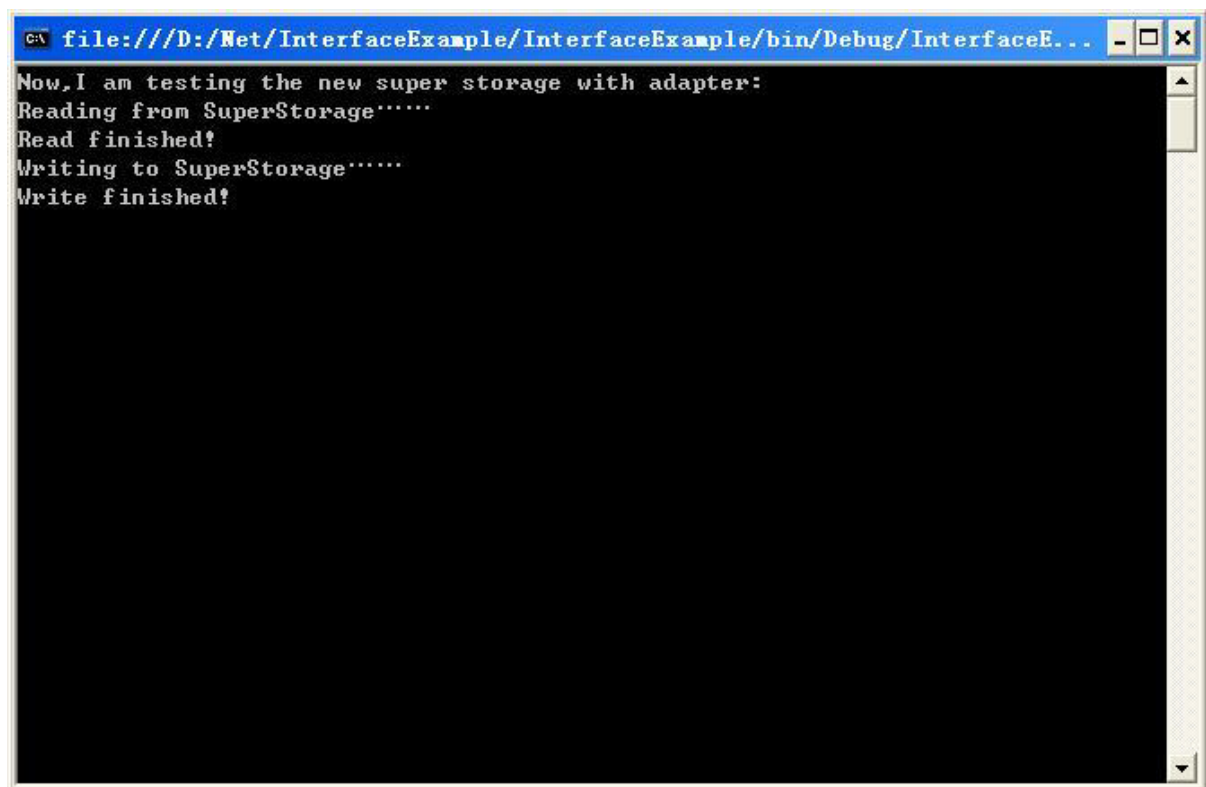
    public void Read()
    {
        this._superStorage.rd();
    }

    public void Write()
    {
        this._superStorage.wt();
    }
}
```

好，现在我们来测试适配过的新设备，测试代码如下：

```
namespace InterfaceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Computer computer = new Computer();
            SuperStorageAdapter superStorageAdapter = new SuperStorageAdapter();
            SuperStorage superStorage = new SuperStorage();
            superStorageAdapter.SuperStorage = superStorage;
            Console.WriteLine("Now,I am testing the new super storage with adapter:");
            computer.UsbDrive = superStorageAdapter;
            computer.ReadData();
            computer.WriteData();
            Console.ReadLine();
        }
    }
}
```

运行后会得到如下结果：



```
file:///D:/Net/InterfaceExample/InterfaceExample/bin/Debug/InterfaceE...
Now, I am testing the new super storage with adapter:
Reading from SuperStorage.....
Read finished!
Writing to SuperStorage.....
Write finished!
```

OK! 虽然遇到了一些困难，不过在设计模式的帮助下，我们还是在没有修改 Computer 任何代码的情况下实现了新设备的运行。

### 1. 从 MVC 开始

MVC 简介：

本文不打算详细解释 MVC 架构，而是把重点放在其中的面向接口思想上。所以在这里，只对 MVC 做一个简略的介绍。

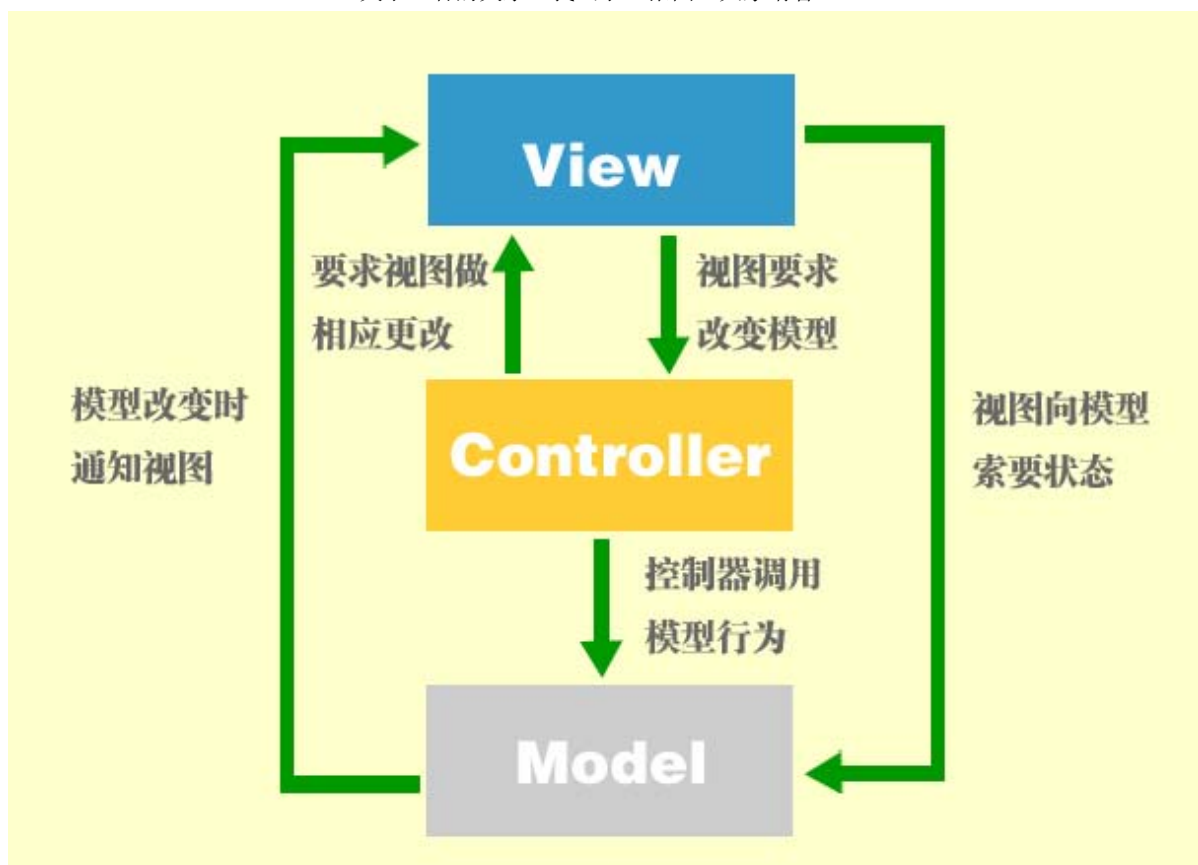
MVC 是一种用于表示层设计的复合设计模式。M、V、C 分别表示模型(Model)、View(视图)、Controller(控制器)。它们的职责如下：

模型：用于存储应用中的数据及运行逻辑，是应用的实体。

视图：负责可视部分，用于与用户交互及呈现数据。视图只负责显示，不负责将用户的操作行为解释给模型。

控制器：负责将用户的行为解释给模型。根据指定的策略和用户的操作，调用模型的逻辑。

关于三者的关系，我画了一张图，大家请看：



它们之间的交互有以下几种：

1. 当用户在视图上做任何需要调用模型的操作时，它的请求将被控制器截获。
2. 控制器按照自身指定的策略，将用户行为翻译成模型操作，调用模型相应逻辑实现。
3. 控制器可能会在接到视图操作时，指定视图做某些改变。
4. 当模型的状态发生改变时，将通过某种方式通知视图。
5. 视图可以从模型获取状态，从而改变自己的显示。

MVC 介绍完了，那么可能会有人问，我们的主题呢？面向接口思想呢？其实，MVC 中处处都存在面向接口的影子。下面，我对其中几个侧面进行解释。

1. 首先我们可以看到，视图和模型是有直接交互的，也就是上面的 4、5 两点。但是有一点可能会让你吃惊：它们两个谁也不“认识”谁，即它们相互并不知道对方是做什么的、有什么属性、有什么方法，但是它们能交互。这是怎么做到的呢？因为它们个各知道对方实现了某一个接口。

此乃面向接口思想一大作用：使相互不认识的类进行交互。这样做是很有好处的，首先它们之间的耦合度大大降低，其次双方都可以进行替换，只要实现了相同的接口，就没有问题。

打个不太恰当的比喻。我们都知道 120 这个电话号码，是急救电话。其实 120 就是个接口，因为当你拨打这个电话时，你不知道那边是哪所医院，甚至不知道那边是不是医院，你只知道电话那头的地方可以救人，也可以说实现了 IHelp 接口。这样，你通过一个号码可以说同全部的救人机构联系起来了，当有紧急事件，接线控制那边会将你的请求接到最近可用的机构，你就可以最快的得到帮助。

现在我们假设没有使用面向接口思想，来看看会发生什么恐怖的事情：首先，我家的 120 号码是绑定在本市第一人民医院的，即当我拨打 120 时，只能拨通第一人民医院。如果有一天我食物中毒了，急忙拨通了 120，但是电话那边告诉我他们医院的救护车都派出去了，我问那怎么接通别家医院的电话，那边

的 MM 很温柔的告诉我，让我打电话给网通公司，然后重新为我布线。于是我吐血而亡……

言归正传。这里，我要引入一个设计模式，叫观察着（Observer）模式。这个模式大约是这样的：整个模式中有两种实体：观察者和被观察者，它们分别实现一个接口，这里我们姑且叫做 IObserver 与 IObserverSubject。IObserver 只有一个方法，例如叫 Update，当被观察者状态改变时，调用这个方法，用来通知观察者。IObserverSubject 接口有两个方法，都是供观察者调用。一个用来将观察者注册为此被观察者的观察对象，另一个用于将观察者移除。

一般情况下，一个被观察者对应多个观察者。

在 MVC 中，视图是观察者，模型是被观察者，当模型状态改变时，调用所有观察者的 Update 方法，通知视图模型有变，视图在 Update 方法里写下响应代码，完成操作。通过这个方法，视图和模型就可以在仅依赖接口的情形下进行交互，而不必强耦合，而且在模型不变的情况下，视图可以随意替换。（只要实现了 IObserver）

2. 在 MVC 中另一个使用接口的地方就是控制器，这里我要首先引入一个设计模式：策略模式（Strategy）。在 MVC 中，控制器就使用了这个模式。

刚才我说过，视图负责与用户交互，但是，它只负责界面显示部分，至于当用户做了某个操作（如单击某个按钮）后系统应该怎么反应，视图并不负责，它只是将这个动作交给控制器，控制器根据内置的策略，将用户操作翻译成模型的逻辑。这就是说，同一个视图、同一种操作，模型可以做出不同的反应，这取决于控制器的内置策略。所以，我们的系统中可以有多个控制器，它们有不同的策略，当视图希望改变策略时，它可以更换控制器。怎么实现呢？这就需要视图不能和具体控制器耦合，而是要仅依赖一个控制器接口（如 IController），并聚合一个 IController 的实例。当希望更改策略时，可以在系统运行时动态更换 Controller，这就是策略模式的实现。

关于 MVC 的接口思想就先介绍到这里。其实 MVC 中还有很多地方用到面向接口，由于本文不是专门介绍 MVC 或设计模式的，所以对用到的模式没有做详解，而是把重点放在其中的面向接口思想上。如果没有设计模式的基础，读上文可能会有些困难，希望各位见谅！我打算在以后专门写文章来解析 MVC。

## 2. .NET 平台下分层架构的面向接口思想

我们知道，在做大一点的系统应用时（特别是 B/S 架构），比较好的方法是分层架构。所谓分层架构，是指将系统从职责上分成若干层，每层各司其职，上层依赖下层完成操作。

在 .NET 平台上，比较经典的分层架构是三层架构，从下到上依次是：数据访问层、业务逻辑层、表示层。各层职责如下：

数据访问层：负责与数据源交互，完成数据访问等一系列操作。

业务逻辑层：完成与系统业务有关的逻辑操作。

表示层：负责与用户交互、呈现数据等一切与系统表示有关的操作。

刚才我们说过，分层架构下是向下依赖的（不考虑依赖倒置），也就是业务逻辑层要调用数据访问层完成与数据源有关的操作，而表示层调用业务逻辑层完成业务逻辑工作。但是，表示层对数据访问层是没有依赖的。

在这个架构中，每一层都不是一个类，而是一个类族，例如，在一个 CMS 系统中，数据访问层可能会有一系列的类，分别负责用户、文章、评论等业务实体的数据访问操作，而业务逻辑层也一样。如果我们直接依赖，即业务逻辑层实例化数据访问层的类，表示层再实例化业务逻辑层的类，会造成强耦合。如果我想把数据库从 SQLServer 换成 MySQL，则要改变整个业务逻辑层代码，这是个不好的设计。（还记得“开放-关闭”原则吗）所以，一般的做法是，为数据访问层和业务逻辑层分别定义一族接口，业务逻辑层不依赖具体的数据访问层，而是仅依赖数据访问层的接口族，表示层也一样，依赖业务逻辑层的接口族。如此一来，当要更换数据库时，我们就不必改写整个业务逻辑层，因为业务逻辑层里根本没有任何数据访问层中的具体类，而全是通过接口实现的。在 .NET 中，只要配合配置文件和反射机制，再运用 Abstract Factory 设计模式，就可以实现“依赖注入”，即在不改动代码的情况下根据配置选择相应的层次组件。这样，我们就可以为不同数据库分别实现数据访问层，也可以编写 ORM 的数据访问层，甚至是基于 XML 的，

只要实现了数据访问层接口族，就可以和业务逻辑层无缝连接，从而极大提高了软件的灵活性和可维护性。当然要更改业务逻辑层也是一样。

如果说，前面的例子都是从微观视角讨论接口，那么，这个例子则从宏观视角展现了面向接口编程的内涵和优势。很抱歉在这里不能对这个架构深入讲解，有兴趣的朋友可以参考微软的官方示例.NET PetShop4。（但是请注意，这个示例中业务逻辑层没有定义接口族，而是强耦合于表示层中，这可能是考虑到在这个系统中业务逻辑没有更改的可能。另外由于是个示例，不是真正的 B2C 系统，所以业务逻辑层很简单。

说明：由于我的电脑出现了点问题。以前收录的 Word 打不开了，因此只有将这篇放到 Engine 篇后面，很抱歉！

### （三）COM 编程技术基础

<http://blog.csdn.net/jianglike18/archive/2009/09/22/4578041.aspx>

```
// 抽象基类 IX 的实现  
virtual void Func1() {cout<<"Func1"<<endl;};  
virtual void Func2() {cout<<"Func2"<<endl;};  
  
// 抽象基类 IY 的实现  
virtual void Func3() {cout<<"Func3"<<endl;};  
virtual void Func4() {cout<<"Func4"<<endl;};  
  
};
```

对于接口，通常是采用抽象基类来定义，并利用类的多重继承来实现该组件。例如，在上面这段代码中，IX 和 IY 是用于实现接口的抽象基类。所谓的抽象基类是只包含一个或多个虚函数声明而未包括虚函数的具体实现的类。抽象基类不能被实例化，而只能用作基类使用，并要求其派生类完成其所有虚函数的实现。在上面这段代码中，CObjectA 组件即继承了 IX 和 IY 这两个抽象基类，并实现了其所定义的虚函数。图 2 为此组件具有的这两个接口的模型展示：

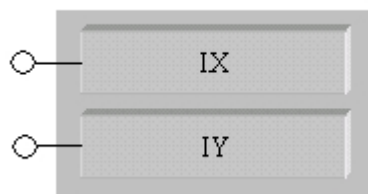


图 2 接口模型

抽象基类本身由于没有实体函数与变量，所以并不分配内存。通常只是用来为派生类指定内存结构。只有在派生类实现此抽象基类时，指定的内存才会被分配。图 3 为此内存结构的示意：

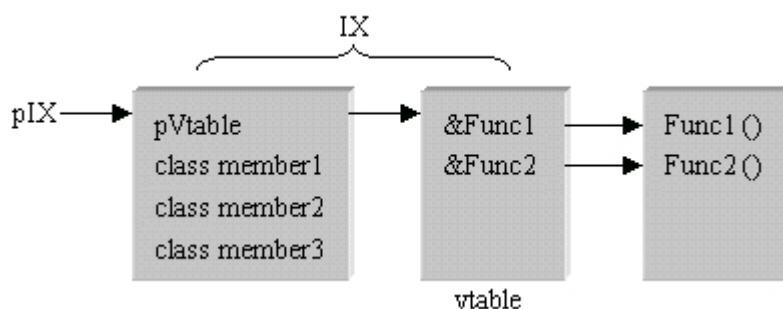


图 3 抽象基类定义的内存结构示意

图中 `vtable` 为虚拟函数表，能够为实例数据的提供一个方便保存的位置，并能够在同一类的多个实例间共享。在每个实例的内存映射中均包含一个指向该类的 `vtable` 表的指针 `pVtable`。`pVtable` 指针存放于所有数据成员之前，由于每个虚函数在 `vtable` 表中有唯一的索引，编译器只需根据索引从 `vtable` 表中找到函数地址即可。也就是说，客户只要获取到了接口指针，就可以使用此 COM 对象的实际功能。

由抽象基类指定的内存结构是符合 COM 规范的，因此抽象基类 `IX` 可以认为是一个 COM 接口，但这还不是一个严格意义上的 COM 接口。对于一个真正意义上的 COM 接口，在设计时应遵循以下几个规则：

- 1) 接口必须直接或间接地从 `IUnknown` 继承。
- 2) 接口必须具有唯一的标识 (IID)。
- 3) 一旦分配和公布了 IID，有关接口定义的任何因素都不能被改变。
- 4) 接口成员函数应具有 `HRESULT` 类型的返回值。
- 5) 接口成员函数的字符串参数应采用 `Unicode` 类型。

这几条规则中，最基本的是第一条，如果一个对象没有至少实现一个最小程度为 `IUnknown` 的接口，那么该对象也就不是一个严格的 COM 对象。`IUnknown` 接口是 COM 的核心接口，从上述规则可以得知，任何一个 COM 接口都必须从 `IUnknown` 接口继承。客户在组件之间的通信是通过接口来实现的。组件可以不提供其他接口，但是必须提供 `IUnknown` 接口以使客户能够对组件其他接口进行查询。

`IUnknown` 接口提供有成员函数 `QueryInterface()`、`AddRef()` 和 `Release()`，分别用于查询组件中的其他接口和进行生存期控制。由于任何 COM 接口都是从 `IUnknown` 接口派生，因此在所有 COM 接口虚拟函数表中保存的前三个成员函数指针一定是指向 `QueryInterface()`、`AddRef()` 和 `Release()` 的指针。这样，任何一个 COM 接口都可以被当作 `IUnknown` 接口来处理。在创建组件时，客户可以通过 `CreateInstance()` 函数得到 `IUnknown` 接口指针。

COM 规范允许使用多接口，`QueryInterface()` 成员函数可以用来查询组件是否支持某个特定的接口。如果支持，`QueryInterface()` 将返回此接口的指针。其第一个参数为一个 IID 结构，指出了客户所要查询的接口，查询到的接口指针将存放在 `ppv` 所指向的变量中。函数的成功执行与否将返回 `S_OK` 或 `E_NOINTERFACE`。但是，在使用时不能简单的将 `QueryInterface()` 返回值与其进行比较，而应使用 `SUCCEEDED` 或 `FAILED` 宏。例如：

```
IUnknown* pI = CreateInstance();
IX* pIX = NULL;
HRESULT hResult = pI->QueryInterface(IID_IX, (void**)&pIX);
if (SUCCEEDED(hResult))
    pIX->Func1();
```

由于 `QueryInterface()` 过于灵活，为避免由此引发的冲突在 COM 规范中定义了 `QueryInterface()` 所有实现都必须遵循的一些规则：

- 1) 过同一对象各个接口指针所查询得到的 `IUnknown` 接口指针必须是指向同一个 `IUnknown` 接口的。即，`IUnknown` 接口的唯一性。
- 2) 如果某接口曾经被成功查询过，那么此后任何时间对该接口的查询也必定会成功。即，接口与查



询时间的无关性。

- 3) 对于已经获取到的接口仍可对其进行再次查询，并且必定会成功。即，接口的自反性。
- 4) 客户能够从任何接口查询到另外一个接口，而且能够返回到起始接口。即，接口的对称性。
- 5) 如果能够从某接口获取到某特定接口，那么从任意接口都可以得到此接口。即，接口的传递性。

IUnknown 接口的另两个成员函数 `AddRef()` 和 `Release()` 对对象的生存期进行了控制。每个 COM 对象都记录有一个引用计数，该引用计数表示了当前引用了此 COM 对象的有效指针的个数。`AddRef()` 和 `Release()` 实现的即是这种引用计数的内存管理技术：引用计数初始为 0，客户每得到一个指向此对象的接口指针即通过 `AddRef()` 将引用计数加 1；在每用完此接口指针后，调用 `Release()` 函数将引用计数减 1。如果引用计数减到 0，则从内存卸载掉此 COM 对象。关于引用计数的使用，在 COM 规范中也设置了以下几条简单的规则：

1) 任何能够返回接口指针的函数（如 `QueryInterface()`、`CreateInstance()` 等）在返回接口指针之前，必须用相应的指针调用 `AddRef()` 函数。

2) 在使用完任何一个接口后，应及时调用该接口的 `Release()` 函数。

3) 在进行接口指针赋值操作后，应调用 `AddRef()` 函数。

COM 组件的创建可以通过 `CoCreateInstance()` 函数来完成，函数原型为：

```
HRESULT __stdcall CoCreateInstance(  
    const CLSID& clsid,  
    IUnknown* pIUnknownOuter,  
    DWORD dwClsContext,  
    const IID& iid,  
    void** ppv  
);
```

函数参数 `clsid` 是要创建组件的 CLSID，`pIUnknownOuter` 用于聚合组件，如果不使用可以设置为 NULL。参数 `dwClsContext` 则限定了所创建组件的执行上下文。最后两个参数 `iid` 和 `ppv` 则分别为要使用接口的 IID 和返回得到的接口指针。在使用时只需将 CLSID、IID 等作为参数传入即可创建相应的组件并从输出参数 `ppv` 得到所请求接口的指针。如果函数是直接创建组件的，那么在函数返回时组件将创建完毕，这样客户将无法对组件的创建过程进行任何干预，灵活性太差。因此，`CoCreateInstance()` 在函数内部实现中通过调用 `CoGetClassObject()` 函数先创建一种专门用来创建组件的组件来解决此问题。这种用途的组件被称为类厂（class factory）。

类厂所支持的用以创建组件的接口是 `IClassFactory`，该接口从 `IUnknown` 派生，并具有两个自己的接口成员函数 `CreateInstance()` 和 `LockServer()`。这两个成员函数分别用于创建 COM 组件对象和控制组件的生存期。下面先给出 `CreateInstance()` 的函数声明：

```
HRESULT __stdcall CreateInstance(IUnknown* pIUnknownOuter, const IID& iid, void** ppv);
```

可以看出，这个用于创建组件对象的 `CreateInstance()` 函数并未包含一个用来接受 CLSID 的参数，显然该函数将只能创建同某个 CLSID 相应的组件。对于一个类厂，由于只能通过 `CreateInstance()` 函数去创

建组件，因此只能创建与某个特定 CLSID 相应的组件。

创建类厂的 CoGetObject ( ) 函数将接收一个 CLSID 作为参数并返回指向类厂对象 IClassFactory 接口的指针。客户将可以通过此指针来创建所需要的组件并返回某接口的指针。通过此指针，客户将可以直接调用新创建的 COM 对象接口的成员函数，从而获得 COM 对象的所有服务。

在用 CoGetObject ( ) 创建类厂对象时，如果 COM 对象是进程内组件（组件与客户处于同一进程地址空间，通常多以 DLL 形式存在），CoGetObject ( ) 将调用 DLL 模块的 DllGetObject ( ) 引出函数并把 clsid、iid 和 ppv 等参数传递进去以创建类厂，并返回类厂对象的接口指针。

如果 COM 对象是进程外组件（拥有独立的进程地址空间，通常多以 EXE 形式存在），则 CoGetObject ( ) 将要首先启动组件进程，并一直等待到组件进程通过 CoRegisterClassObject ( ) 函数将类厂注册到 COM 后，才会返回 COM 中相应的类厂信息。一旦组件进程退出，此注册的类厂对象也就不再有效，需调用 CoRevokeClassObject ( ) 函数予以通知。图 4 展示了通过类厂创建组件的过程：

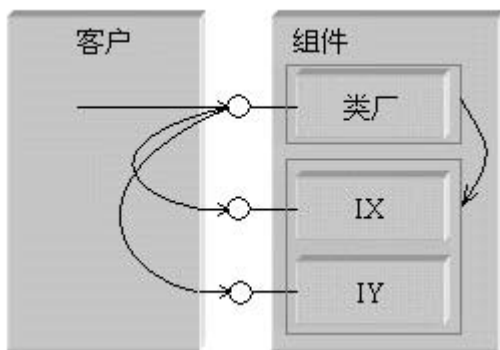


图 4 组件的创建过程

客户程序对 COM 组件的调用主要分对进程内组件调用和进程外调用两种情况。在具体过程上却并没有什么太大的区别。为了能够使用 COM 库提供的 API 函数，首先要用 CoInitialize ( ) 初始化 COM 库。

虽然通过 CLSID 和 ProgID 都可以标识一个组件，但 ProgID 显然要比 CLSID 更易于理解和使用，因此通常很少直接使用 CLSID，而是通过使用 CLSIDFromProgID ( )，根据 ProgID 得到组件的 CLSID。进而以此返回的 CLSID 作为参数去调用 CoGetObject ( ) 以创建类厂对象并返回类厂接口指针。通过该指针调用类厂对象的 CreateInstance ( ) 接口成员函数，执行结果将创建与 CLSID 相应的组件对象并返回 IUnknown 接口指针。通过此接口的 QueryInterface ( ) 成员函数将能够进一步获过程将是隐含进行的，使用更为简单。

取组件的其他接口指针，从而使用组件提供的各种服务。

最后，通过 Release ( ) 函数释放接口指针。如果使用的进程内组件，在调用 CoUninitialize ( ) 函数释放 COM 库资源之前，应首先调用 CoFreeUnusedLibraries ( ) 将其从内存卸载。由于在 CoCreateInstance ( ) 函数内部实现了对 CoGetObject ( ) 的调用并一直完成了类厂对象接口函数对组件的创建和类厂对象的释放。

COM 的东西很多很多，也感谢那些将自己的心得写在博客上的人，我们是二次开发，所以我只将部分内容收录，也因为只是的缘故，我多 COM 也只是了之甚少，如果有兴趣的朋友可以在 [GOOGLE](https://www.google.com) 上搜索，我在这些文章中经常提到 [GOOGLE](https://www.google.com)，我不是为它打广告，推荐一个网

址: [http://tech.ddvip.com/2006-04/11446295404705\\_6.html](http://tech.ddvip.com/2006-04/11446295404705_6.html)

当然文章中的链接都可以访问,自己也可以在搜索些感兴趣的比如 COM 对象的产生过程之类的话题。

题外话: (来自: [http://tech.ddvip.com/2006-04/11446295404705\\_6.html](http://tech.ddvip.com/2006-04/11446295404705_6.html))

**题外话:** 如果这个 COM 对象实现一个以上的接口 (不包括 IUnknown), 你就必须用 QueryInterface() 方法来获得任何你需要的附加的接口指针。QueryInterface() 的原型如下:

```
HRESULT IUnknown::QueryInterface (  
    REFIID iid,  
    void** ppv );
```

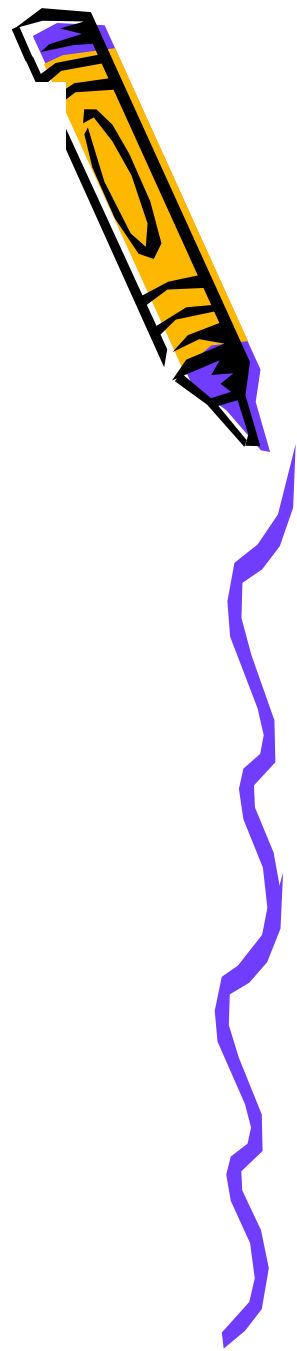
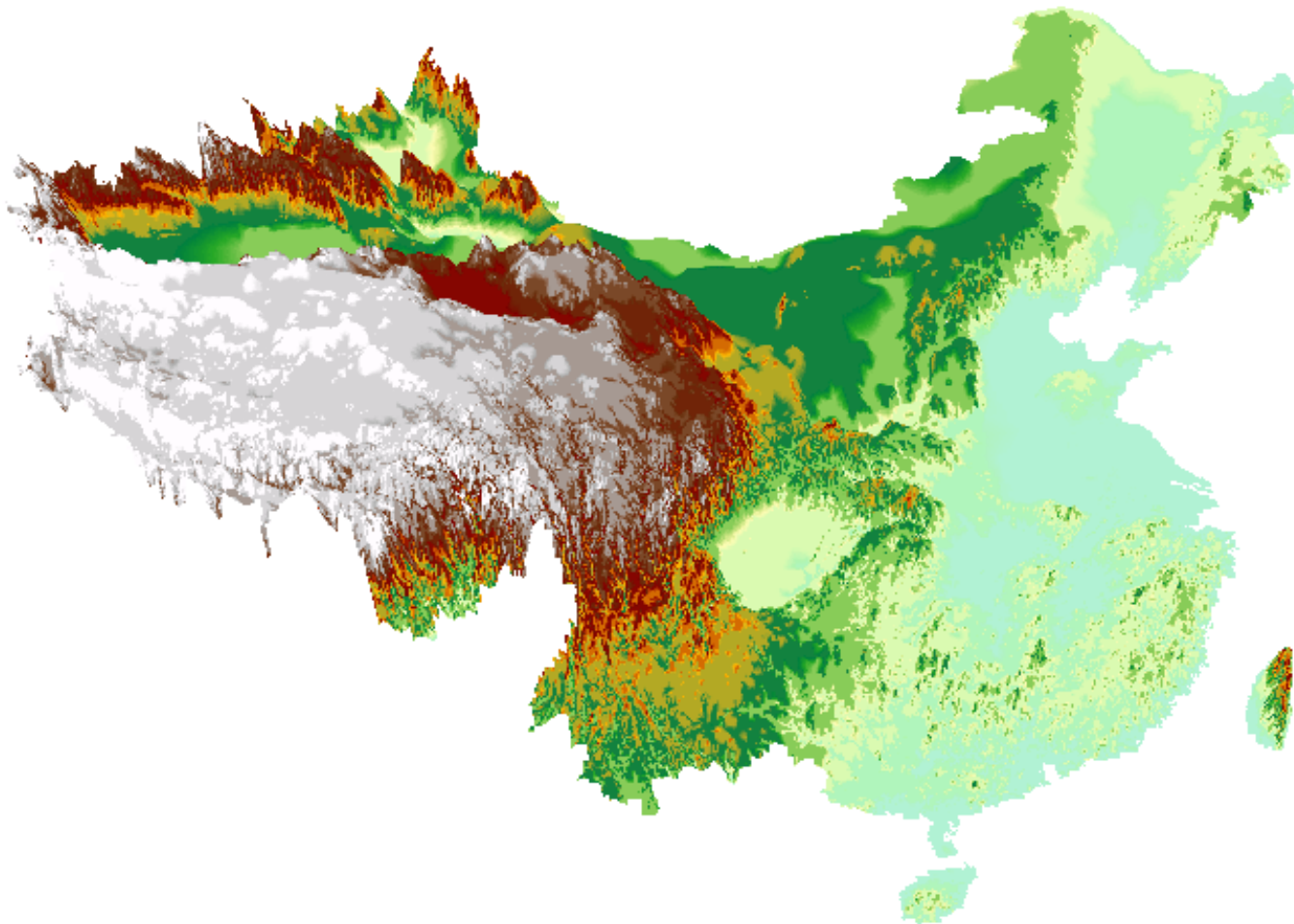
以下是参数解释:

iid

所请求的接口的 IID。

ppv

接口指针的地址, QueryInterface() 通过这个参数在成功时返回这个接口。



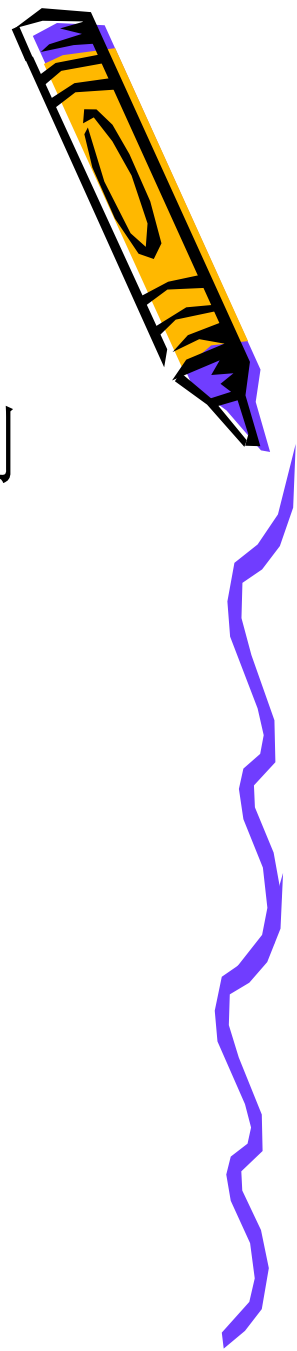
QQ:271718002 E-mail: sishui198@163.com

在学习**AE**或者关于**AO**的那些东西的时候，  
一头的雾水不知从何开始.....

对于一个地学出身的我，对计算机的知识缺乏太多，仅仅凭借自己当时在**Java**课堂中的一点东西和借助搜索引擎上路，通过一段时间的摸索，自己也有了一点体会，在此和大家分享.....



Omd这个东西应该是一开始就会接触到的，那我就从这里开始.....



补充几个名词概念:

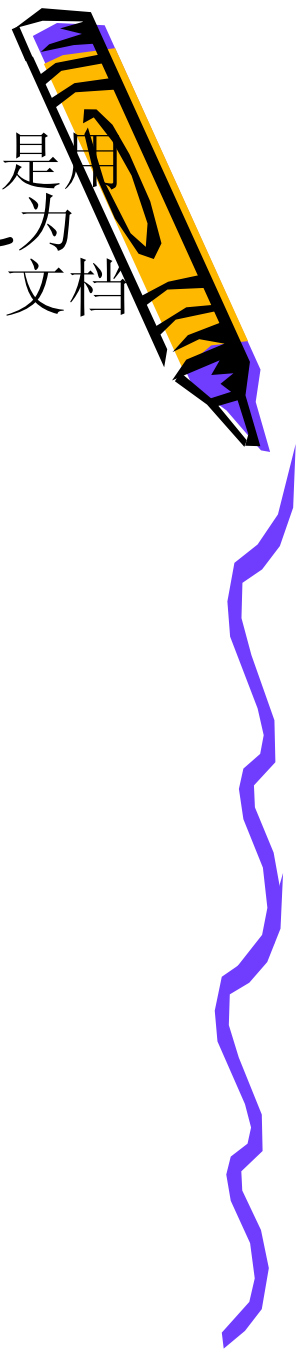
**UML: Unified Modeling Language** 统一建模语言，是用来对软件密集系统进行可视化建模的一种语言。UML为面向对象开发系统的产品进行说明、可视化、和编制文档的一种标准语言。

**OMD: Object model diagrams** 对象模型图表。

首先来看看OMD能帮我们做什么？

1. 该类支持哪些接口；
2. 完成任务需要哪些对象；
3. 如何使用该类的对象；
4. 是否可以直接实例化类；
5. 接口有哪些方法和属性；
6. 是否有其它类也支持该接口；
7. 对象间的关系

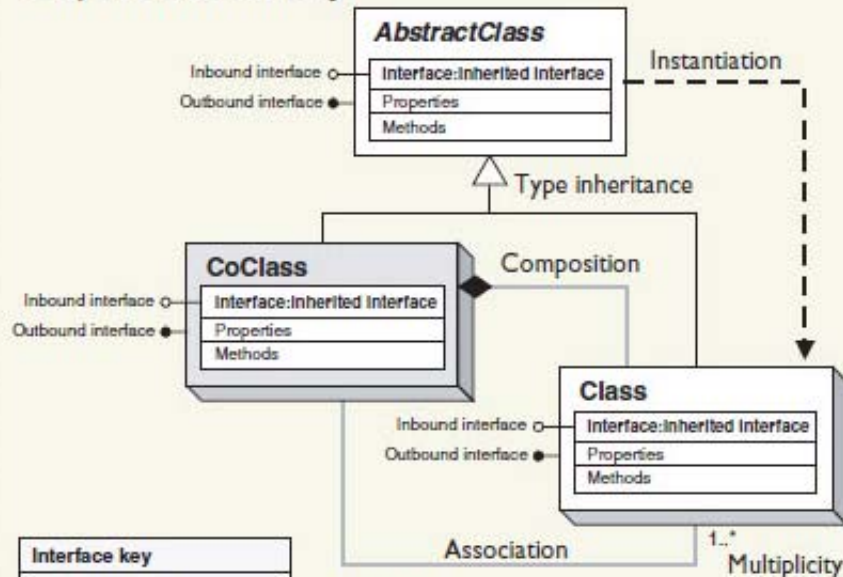
下面图示中，便是贯穿本书的对象模型图的钥匙。





# Reading the object model diagrams

## Object model key



Interface key	
■	Property Get
■	Property Put
■	Property Get/Put
□	Property Put by Reference
←	Function
←	Event function

### Special Interfaces

(Optional) represents interfaces that are inherited by some subclasses but not all. The subclasses list the optional interfaces they implement.

(Instance) represents interfaces that are only on specific instances of the class.

(<classname>) indicates the name of the helper class required to support this event interface in Visual Basic.

### Types of Classes

An **abstract class** cannot be used to create new objects, it is a specification for instances of subclasses (through type inheritance.)

A **coclass** can directly create objects by declaring a new object.

A **class** cannot directly create objects, but objects of a class can be created as a property of another class or instantiated by objects from another class.

### Types of Relationships

**Associations** represent relationships between classes. They have defined multiplicities at both ends.

**Type inheritance** defines specialized classes of objects that share properties and methods with the superclass and have additional properties and methods. Note that interfaces in superclasses are not duplicated in subclasses.

**Instantiation** specifies that one object from one class has a method with which it creates an object from another class.

**Composition** is a relationship in which objects from the "whole" class control the lifetime of objects from the "part" class.

An **N-ary association** specifies that more than two classes are associated. A diamond is placed at the intersection of the association branches.

A **Multiplicity** is a constraint on the number of objects that can be associated with another object. Association and composition relationships have multiplicities on both sides. This is the notation for multiplicities:

1 - One and only one (if none shown, one is implied)

0..1 - Zero or one

M..N - From M to N (positive integers)

\* or 0..\* - From zero to any positive integer

1..\* - From one to any positive integer



其中

OMD中的符号：

一个三角形符号 表示继承

菱形的黑色小块 表示组成

虚线前头带个箭头 表示用来创建

\*表示对应关系 1：N

一条直线表示 联合

类的类别：

抽象类：不能创建或实例化，从来没有一个抽象类的实例

用于定义子类的公共接口，子类继承其定义的接口。

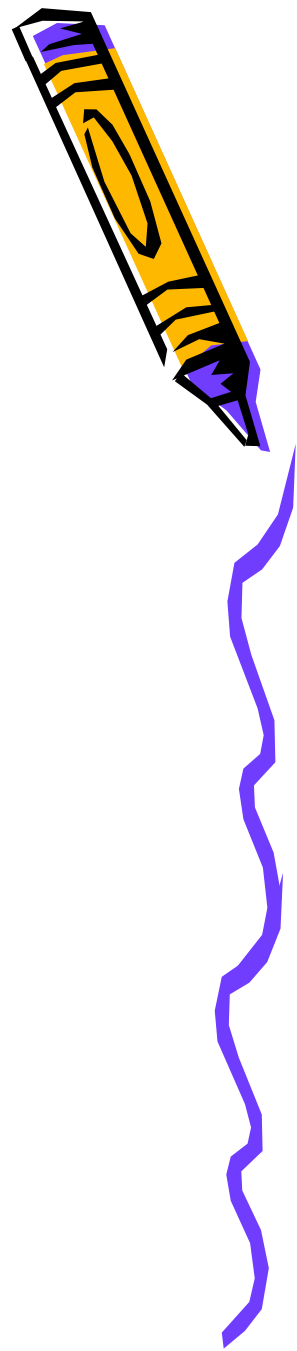
OMD符号为：二维的内部有阴影的矩形。

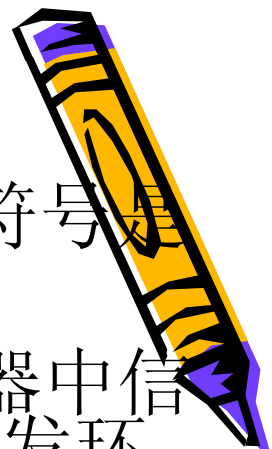
实例化类：不能创建，从别的对象获得实例。

OMD符号为：3D矩形内部没有阴影。

可创建的类：用New关键字创建对象或者从别的对象获得运行实例。

OMD符号为：带阴影的3D矩形符号。





这些符号是基于UML 画图工具创建的，UML 符号是面向对象分析和设计的工业图样标准。

对象模型图中提供的信息非常多，是对象浏览器中信息的重要补充。**Visual Basic**，或者其它的开发环境，都会列出所有的类和成员，但不会指明这些类之间的关系。所以，对象模型图是非常有利于读者对**ArcInfo** 组件的理解的！

本书使用UML 来描述**ArcInfo** 组件，即**ArcObjects**，并描述你能够创建的数据模型。

以下详细说明。



# 1. 类和对象

在UML 图中有三种类型的类：抽象类（**abstract class**）  
可创建类（**createable class**）与可实例化类  
（**instantiable class**）。

抽象类不能用以创建新对象，但可以用来指定子类。举个例子，“**line**”（线）是“**primaryline**”（干线）和“**secondary line**”（副干线）的抽象类。

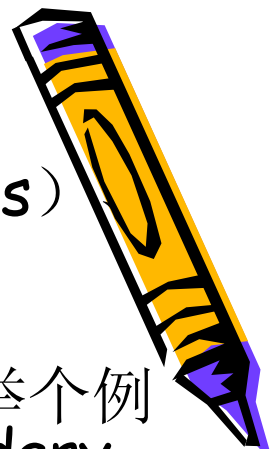
可创建类指的是那些你能够直接使用开发环境中的对象定义语法来创建对象。比如在**Visual Basic** 中是这样书写：**Dim As New <object>** 或者**CreateObject <object>**。

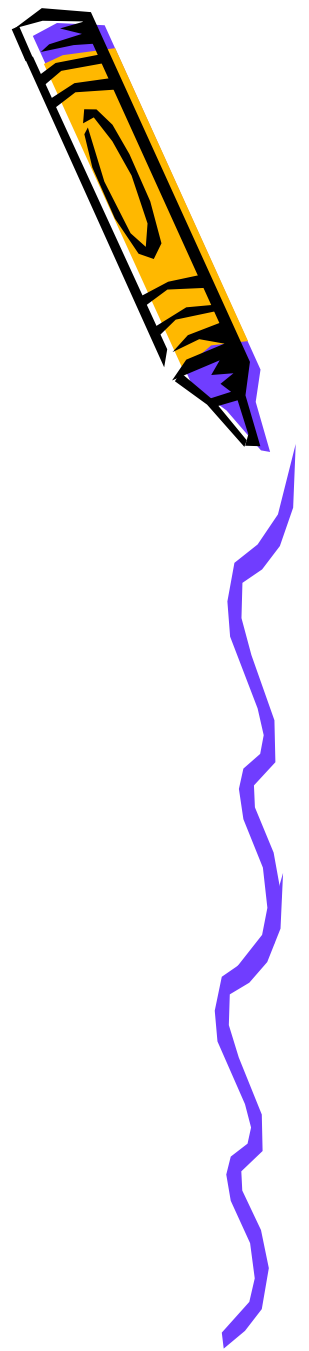
可实例化类不能够直接创建新对象，但是这种类的对象能够作为其它对象的属性被创建或是从其它类的方法中创建。-----比较难理解。

## 2. 关联

在抽象类、可创建类和可实例化类之间，有几种存在的关联（或称关系）。

关联（**association**）便描述了类之间的关联。在两端的类中可以定义多重性（**Multiplicity**）关联。

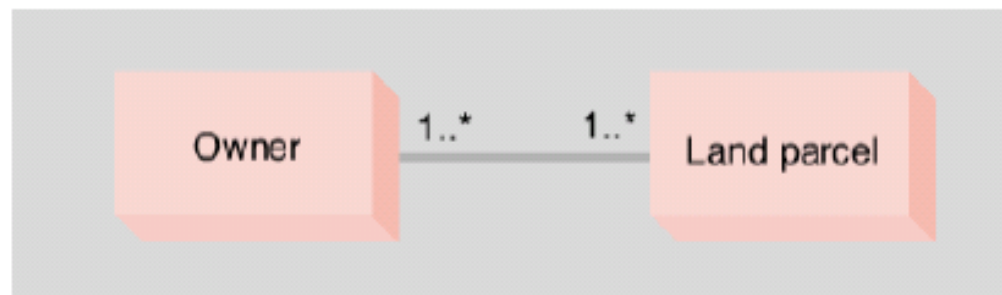




## 2. 关联

在抽象类、可创建类和可实例化类之间，有几种存在的关联（或称关系）。

**联系（association）**便描述了类之间的关联。在两端的类中可以定义多重性（ Multiplicity）关联。



在这张图上，一个业主能有拥有一块或多块宗地；同样地，一块宗地可以被一个或多个业主所共有。

多重性关联就是限制对象类与其它对象关联的数目关系。以下是用于多重性关联的符号：

1 —— 一个并且只有一个，这种多样性是可选的；如果不标明，则默认为“1”

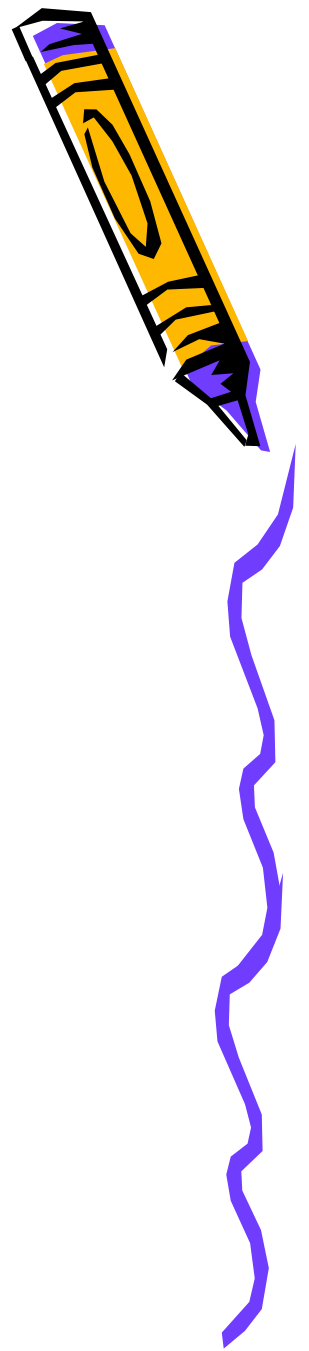
0..1 —— 零个或一个

M..N —— 从M到N（正整数）

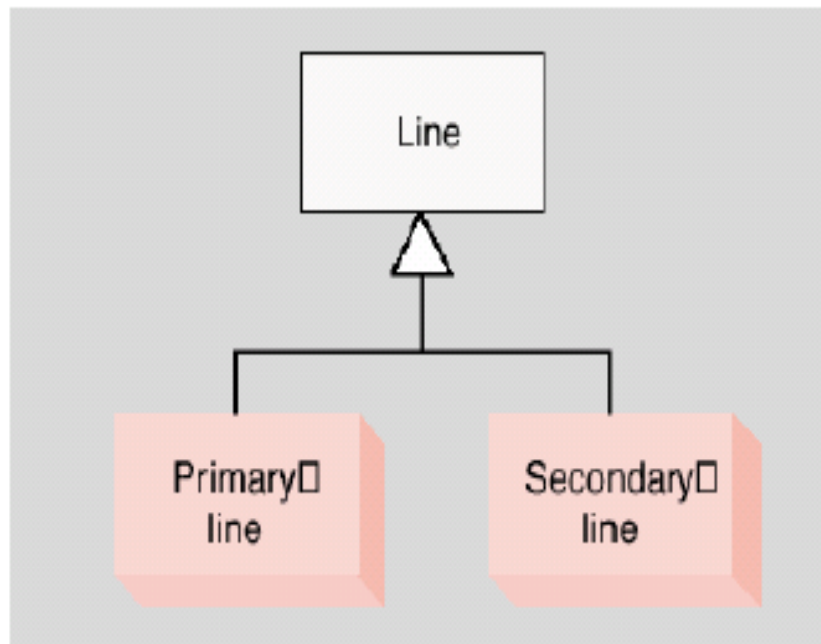
\*或者0... \* —— 从零到任意正整数

4 \* 11 正整数





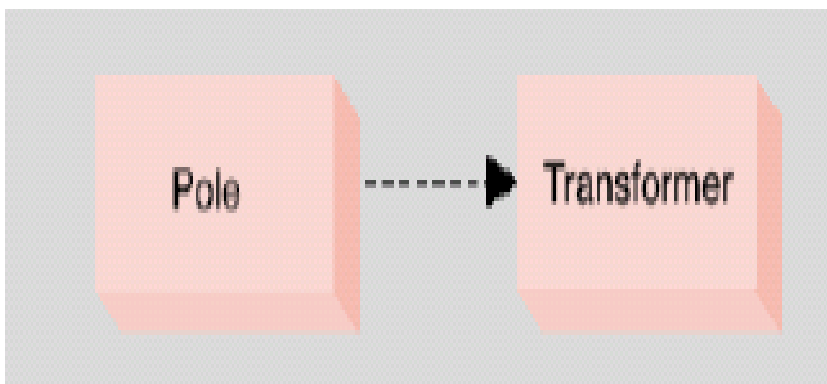
类继承 (type inheritance) 定义了专门的类，它们拥有超类的属性和方法，并且同时也有自身的属性和方法。



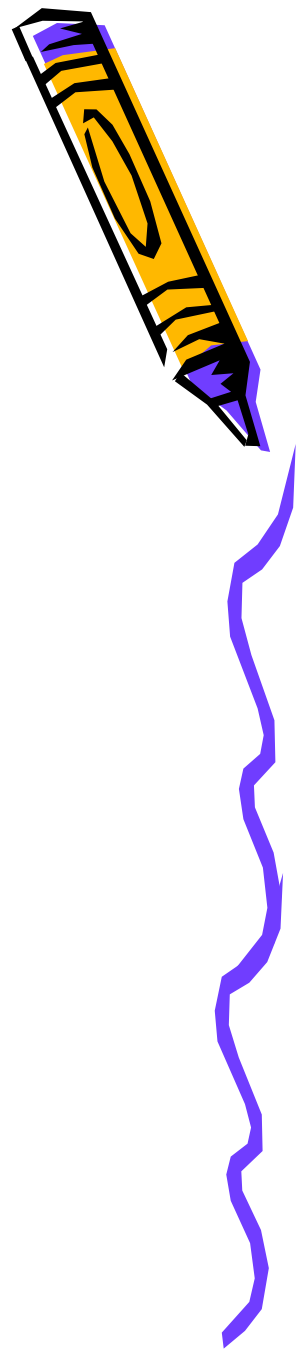
上图说明primary line 和secondary line 是line 的一种类型。



**实例化 (Instantiation)** 指定一个类的对象有这样的方法，它能够创建另外一个类的对象。



pole 对象有一个方法能够创建transformer 对象。





**聚合 (Aggregation)** 是一种不对称的关联方式，在这种方式下一个类的对象被认为是一个“整体”，而另一个类的对象被认为是“部件”。

#### 4. 聚合关系 (Aggregation)

聚合关系是一种特殊形式的关联。聚合表示类之间的关系是整体与部分的关系。一辆轿车包含四个车轮、一个方向盘、一个发动机和一个底盘，这是聚合的一个例子。在需求分析中，“包含”、“组成”、“分为……部分”等经常设计成聚合关系。如图 2.6 所示。

在图 2.6 中，一个 Transformer Bank 正好有 3 个 Transformer。在这个图中，Transformer 能和一个 Transformer Bank 相关联，但当 Transformer Bank 移除以后，Transformer 依然能够存在。

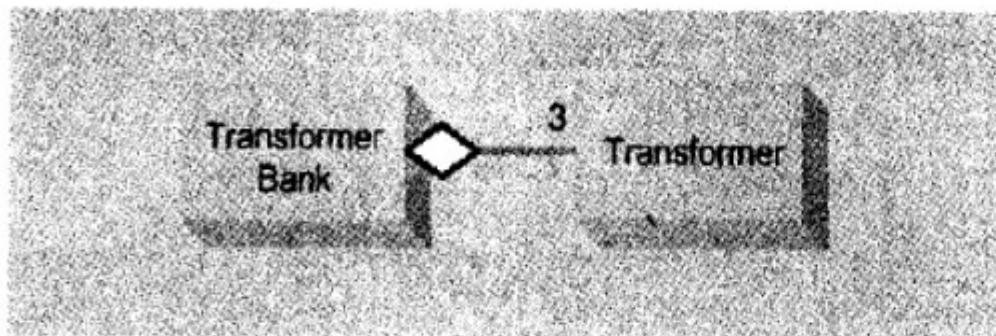


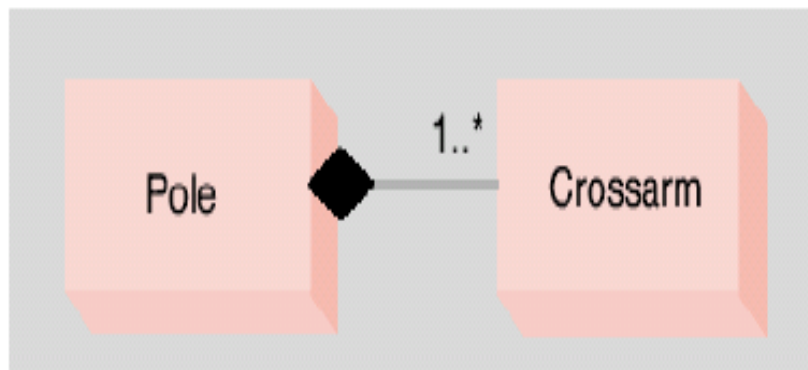
图 2.6 聚合关系示例

一个 transformer bank 正好有 3 个 transformer。在这个图中 transformer 能和一个 transformer bank 相关联，但当 transformer bank 移除以后，transformer 依然能够存在。





**组成 (Composition)** 是一种更为强壮的聚合方式，此种方式下，“整体”对象控制着“部分”对象的生存时间。



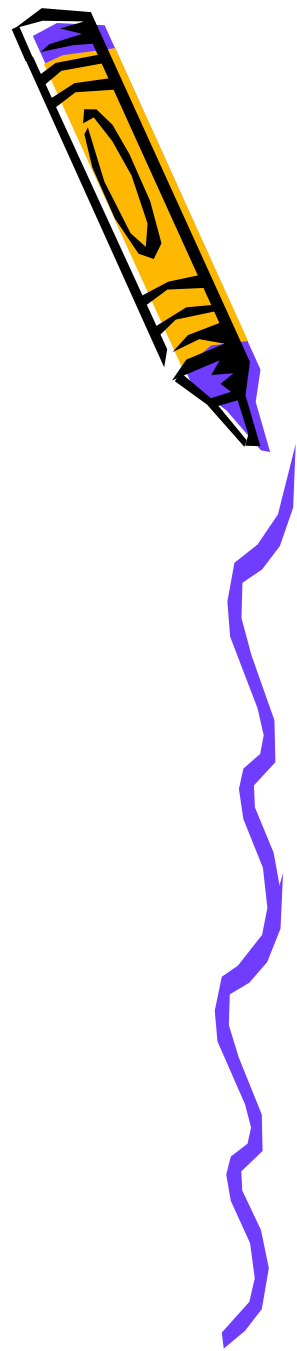
一个pole 包含一个或多个crossarm。在这个图中当pole 被移除后，crossarm 就不能再使用了。因为pole 控制着crossarm 的生存时间。

在安装完AE后,我们就可以在相关目录中找到AE的OMD图,

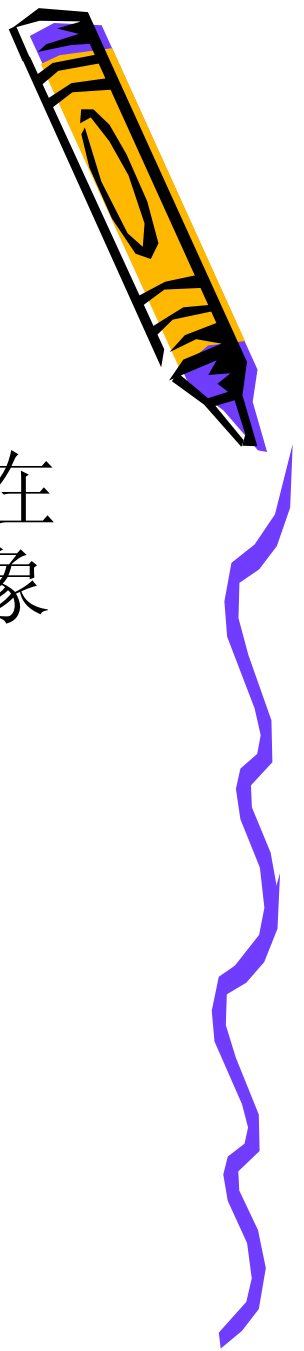




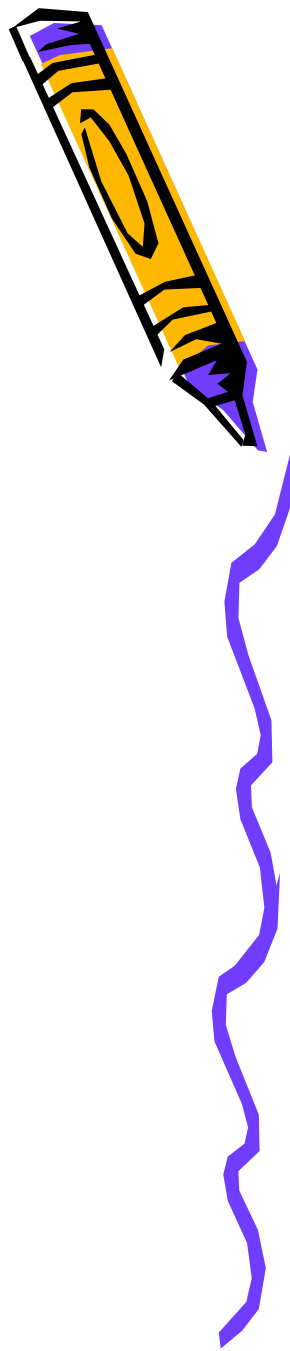
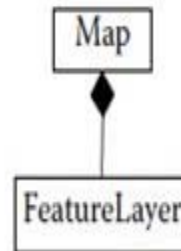
到处都是接口

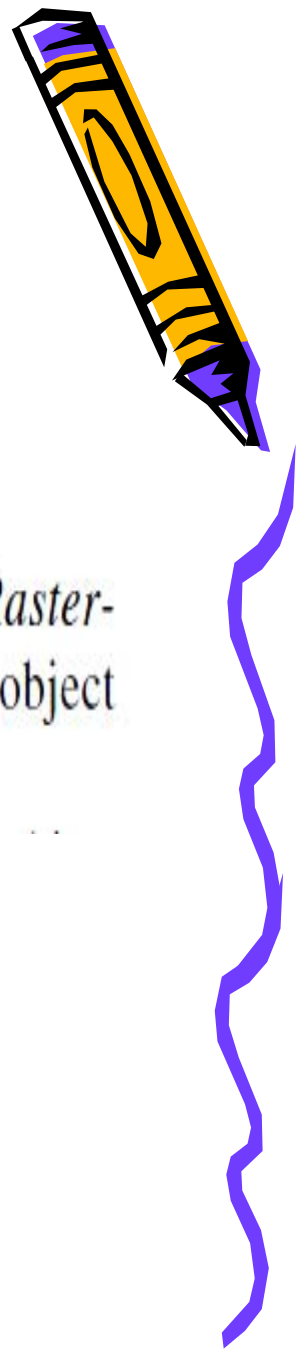


很多资料都会提到在**AO**开发的时候，  
和我们打交道的是接口，而不是我们在  
一些面向对象语言如（**java**）中的对象  
接口将方法和属性暴露给我们。



When programming with objects in ArcObjects, one would never work with the object directly but, instead, would access the object via one of its interfaces. An *interface* represents a set of externally visible operations. For example, a *RasterReclassOp* object implements *IRasterAnalysisEnvironment* and *IRasterReclassOp*





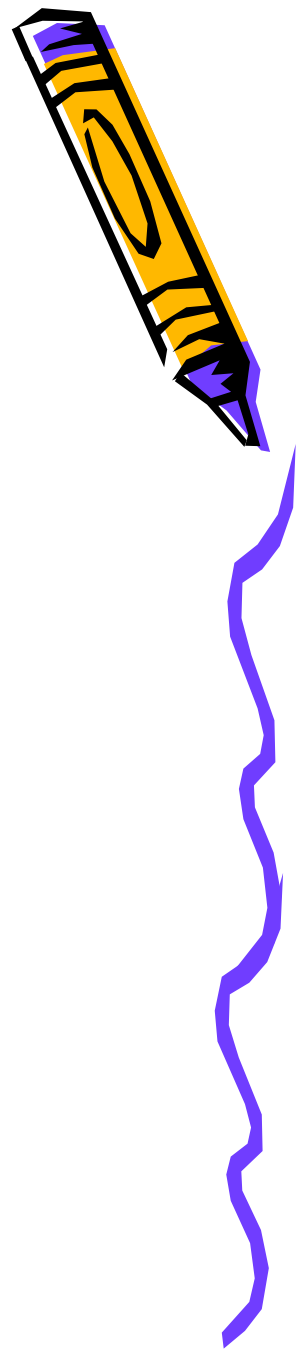
(Figure 1.6). We can access a *RasterReclassOp* object via either the *IRasterAnalysisEnvironment* interface or the *IReclassOp* interface, but not the object itself.



## 接口和类的关系

interfaces are often referred to as the “What” and “How” of COM. The interface defines what an object can do, and the class defines how it is done.

一个类可以继承个接口，而一个接口也可以被多个类继承





## 属性和方法：

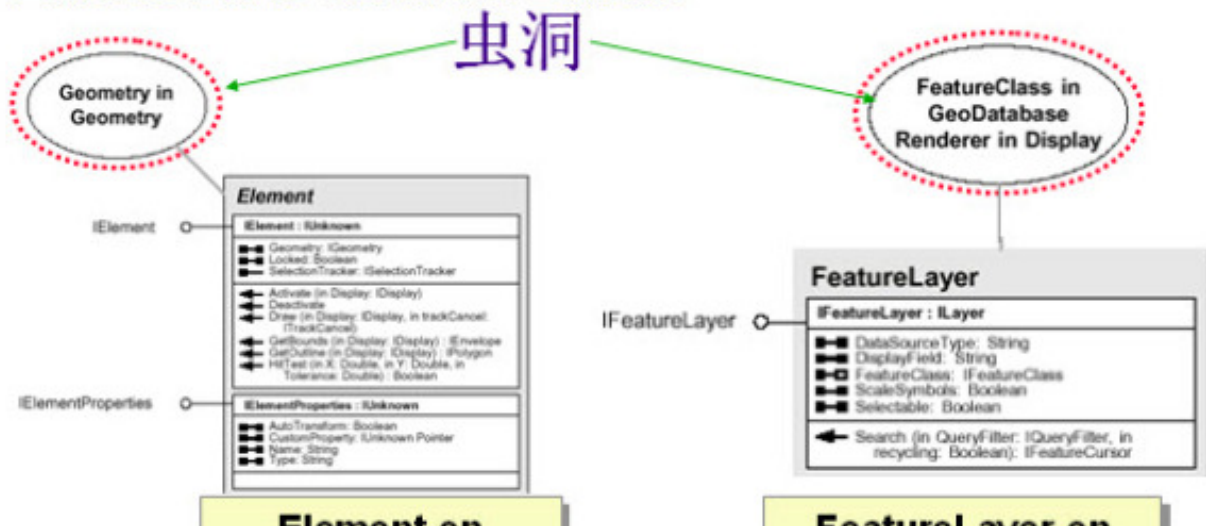
属性：哑铃状的图标，Read（左侧的哑铃）和write（右侧的哑铃）属性可分为两种类型（值和引用）见下图

方法：指向左侧的箭头

接口：棒棒糖类型的图标

图表之间的连接：虫洞

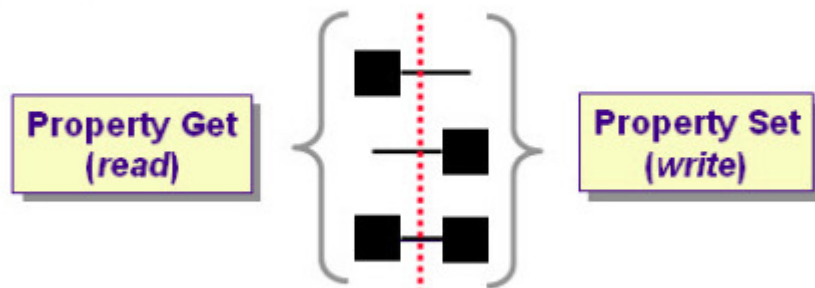
- ◆ 概念上,这是一个对象模型
- ◆ 事实上,被分别在几张图表里存放
- ◆ 虫洞表现了图表与图表间的连接关系



## Property and method symbols

◆ Property 

◆ 哑铃形状的图标



◆ Method 

---

## ◆ Property Put: Most ArcObjects properties

- ◆ Property holds a value or a *copy* of an object
- ◆ Do not use Set keyword

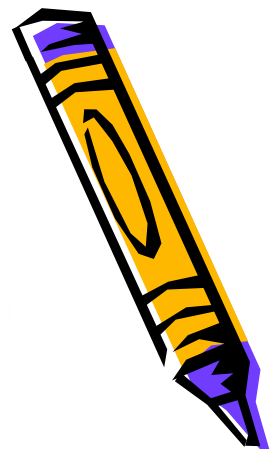
```
pLayer.Name = "Port Moresby" 'No Set keyword
```

## ◆ Property Put by Reference: Some ArcObjects properties

- ◆ Property holds a *reference* to an object
- ◆ Must use the Set keyword

```
Set pLayer.FeatureClass = pMoresbyData 'Must use Set!
```

- ◆ 如果引用对象发生了变化，对象的属性将同步受到影像





## Setting properties

There are two ways in which a value may be assigned to an object property, *by value* or *by reference*. Most object properties are set by value, which means a copy of the value is stored as the property value. Below is an example of setting a property by value:

```
valueY = 43
someObjectVariable.PropertyX = valueY
valueY = 100
MsgBox someObjectVariable.PropertyX
```

In the example above, even though the value of valueY changes later in the program, someObjectVariable's PropertyX value is unchanged. The value that appears in the message box above would be 43.

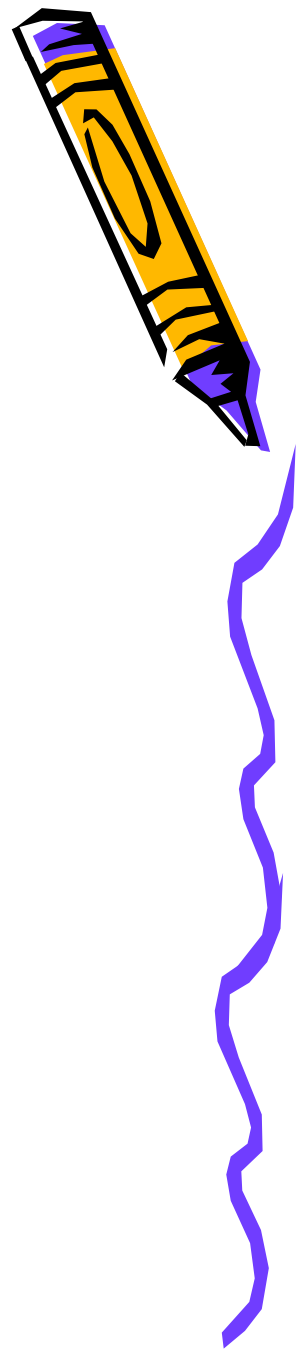
Some object properties, however, are set by reference, which means the property is not assigned using a copy of a value, but rather with a reference to some existing object. Below is an example of setting a property by reference:

```
valueA = 43
Set anotherObjectVariable.PropertyB = valueA
valueA = 100
MsgBox anotherObjectVariable.PropertyB
```

When a property is assigned with a reference, there is a dynamic association between the object property and the object that was used to set it. In the example above, the value 100 would be displayed in the message box because as the value of valueA changes, so does the value of anotherObjectVariable's PropertyB property.

When to use set





个人感觉在OMD图中比较难理解的就是可实例化类CLASS。我做了以下例子去理解它：

ArcEngine中 workspace 就是一个可实例化类，

Workspace不能用new创建，但可以通过WorkspaceFactory.OpenFromFile()方法来创建。如下：

```
IWorkspaceFactory pWorkspaceFactory = new AccessWorkspaceFactoryClass();
```

```
IWorkspace pWorkspace = pWorkspaceFactory.OpenFromFile(ConnectionString, 0);
```

但当我们尝试用New去创建一个Workspace类时，如下：

```
IWorkspace pWorkspace = new WorkspaceClass();
```

就会出现以下错误：

“类型” ESRI.ArcGIS.Geodatabase.WorkspaceClass未定义构造函数；

所以，Class和CoClass的主要区别就是CoClass是带构造函数的Class。以此它可以用New直接创建对象。

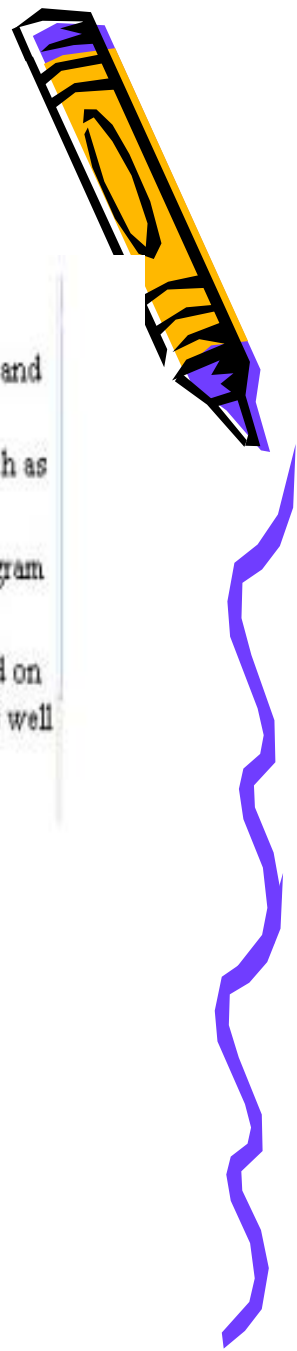


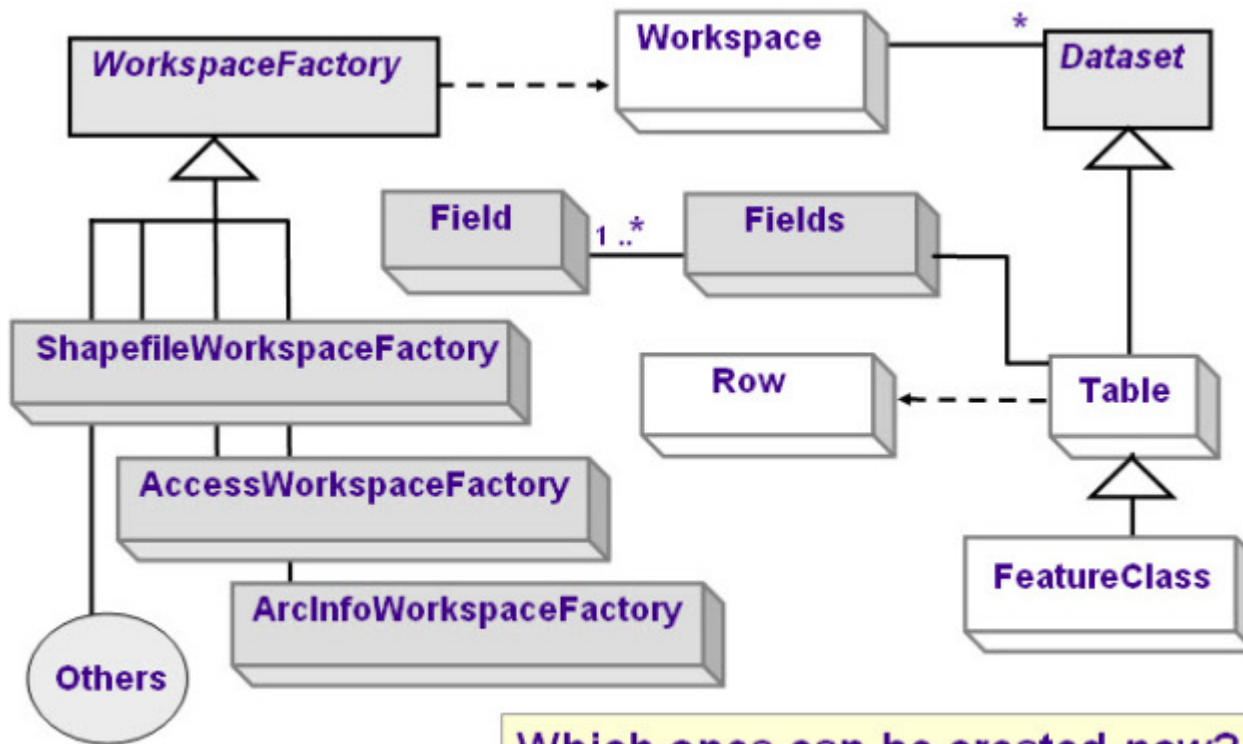
## Wormholes

Remember that there is only one ArcObject object model. All of the ArcObject classes are defined in a single library (*esnCore.olb*), and all of these classes are somehow related to other ArcObject classes. To make your life (at least a little) simpler, the object model diagrams have been organized into logical groups of classes. At the moment, there are over 20 different object model diagrams, such as ArcMap, Geodatabase, Raster, Display, and Geometry.

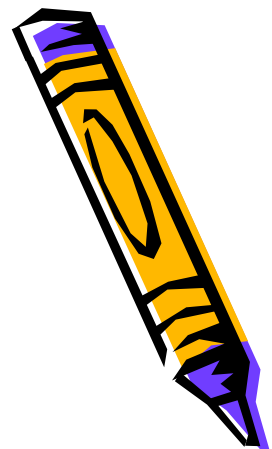
When a class on one diagram is related to a class on another diagram, a *Wormhole* is used to indicate the related class and the diagram on which it appears. As a programmer, this is your bridge that connects each of the object model diagrams.

In the example above, the wormhole on Element indicates that the Element class is related to the Geometry class, which is described on the Geometry diagram. The FeatureLayer class is associated with the FeatureClass class (described in the Geodatabase diagram) as well as the Renderer class (described on the Display diagram).





Which ones can be created *new*?

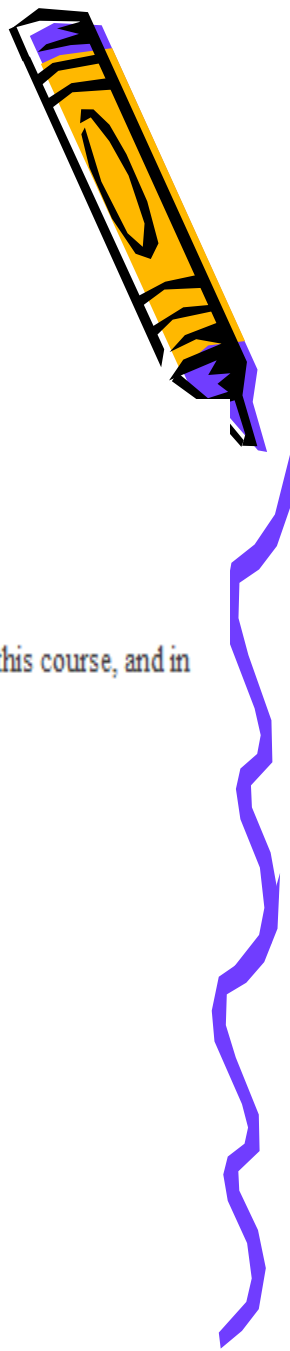


## 关于QI

### QueryInterface

The term QueryInterface refers to the process of using additional interfaces on the same object. You will hear this term used often throughout this course, and in the code you will often see it abbreviated as 'QI'.

Here is the basic form for QueryInterface:



'Create a new RaceCar with the IDrive interface

```
Dim pCar As IDrive
Set pCar = New RaceCar
pCar.Accelerate
```

'Switch interfaces

```
Dim pRace As IRace
Set pRace = pCar
pRace.PitStop
pCar.Accelerate
```

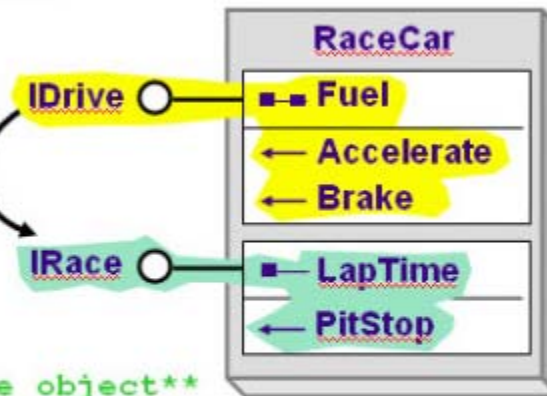
'\*\*pCar and pRace point to the same object\*\*

```
Dim pArea As IArea
```

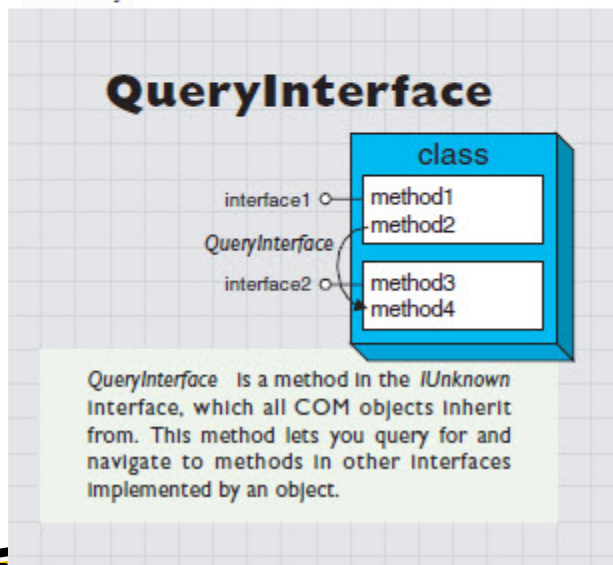
```
Dim pPt As IPoint
```

```
Set pArea = pPolygon ' QI for IArea on pPolygon
```

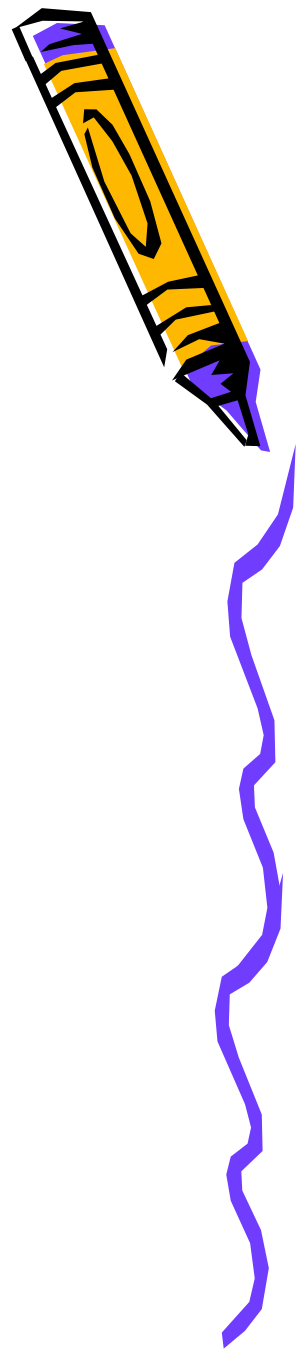
```
Set pPt = pArea.Center
```



When requested for a particular interface, the *QueryInterface* method can return an already assigned piece of memory for that requested interface, or it can allocate a new piece of memory and return that. The only case when the same piece of memory must be returned is when the *IUnknown* interface is requested. When comparing two interface pointers to see if they point to the same object, it is important that a simple comparison not be performed. To correctly compare two interface pointers to see if they are for the same object, they both must be queried for their *IUnknown*, and the comparison must be performed on the *IUnknown* pointers. In this way, the *IUnknown* interface is said to define a COM object's identity.

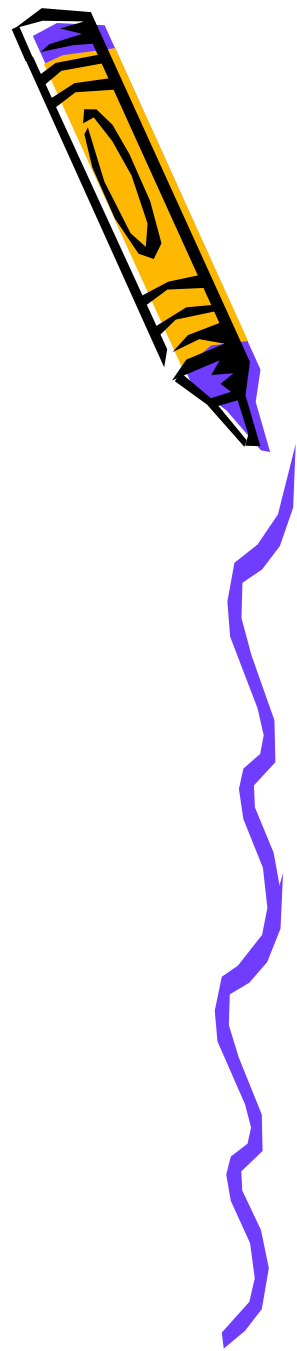


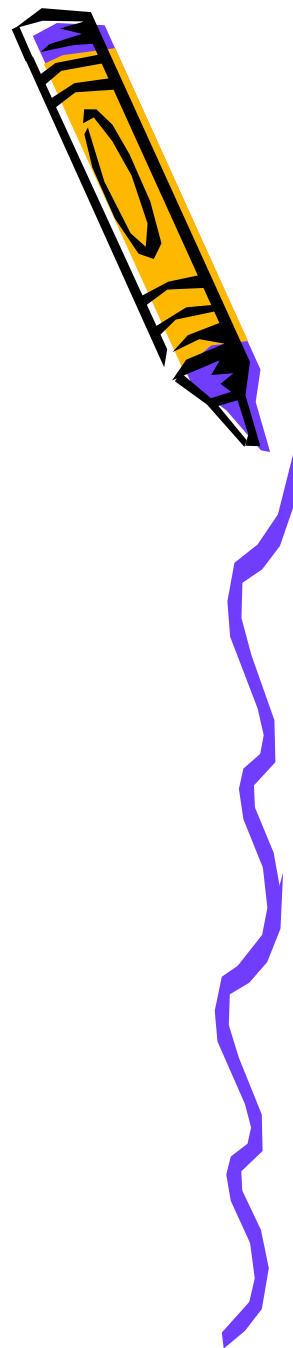
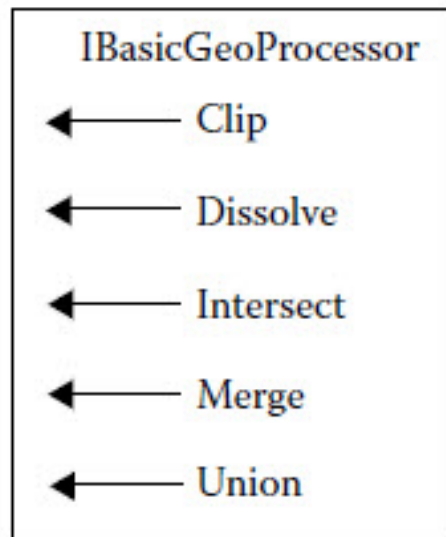
关于更多QI见我的一个word





常用的几个





## PROGRAMMING ARCOBJECTS WITH VBA: A TASK-ORIENTED APPROACH

