C++ Builder 5
高级编程技术

—— COM、CORBA 与 Internet 编程

徐新华　等　编著

人民邮电出版社

内容提要

本书全面深入地介绍了 COM 与 Interface、ActiveX 框架、类型库、COM 客户与 COM 服务器、ActiveX 控件、OLE Automation、Active Server Page、MTS 或 COM+、CORBA、WinSock、连接 Internet、Internet 协议、Web 服务器扩展和 MIDAS Web 应用程序等内容。

C++ Builder 5 是一个完全面向对象的编程工具。众多长期从事编程的人员从实践中体会到，只要真正领会了面向对象的编程思想，即使是很高深的编程领域，诸如 COM、ActiveX、CORBA、MIDAS 都不难掌握。所以，本书的重点是面向对象编程。

本书内容全面而又不失简洁，例子丰富，既可以作为广大读者学习 C++ Builder 5 的入门指导书，也可以作为程序员编程时的参考手册。

C++ Builder 5 高级编程技术

——COM、CORBA 与 Internet 编程

◆ 编　　著　徐新华 等

责任编辑　王晓明

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号

邮编　100061　　电子函件　315@pptph.com.cn

网址　http://www.pptph.com.cn

北京汉魂图文设计有限公司制作

北京顺义向阳胶印厂印刷

新华书店总店北京发行所经销

◆ 开本：787×1092　1/16

印张：24.5

字数：611 千字　　2000 年 12 月第 1 版

印数：1 – 0 000 册　　2000 年 12 月北京第 1 次印刷

ISBN 7-115-09010-6/TP・1986

定价：37.00 元

前　言

C++ Builder 5 是用于电子商务、Internet 应用和数据库编程等开发工作的最佳工具之一。C++ Builder 5 最接近 ISO 的 C++标准；同时支持 COM 与 CORBA 两大分布式计算规范；C++ Builder 5 支持 ADO，从而可以访问更多的数据；C++ Builder 5 增加了数据模块设计器，可以轻松地创建和维护数据模块；通过 InterBase Express(IBX)，C++ Builder 5 集成了对 InterBase 数据库的访问，不再需要借助于 BDE；MIDAS 与新增加的 Internet Express 配合使用，使客户(Web 浏览器)能更方便地与 MIDAS 服务器交互；C++ Builder 5 内建了全球 ORB 分发数量最多的 VisiBroker 4.0，集成了 CORBA IDL 编译器，单一步骤就能生成 CORBA 对象；TeamSource 是一个集成的工作流程管理工具，大大简化了团队开发；运用 Integrated Translation Environment(ITE)，可以方便地进行软件的本地化和国际化。另外，C++ Builder 5 可以很方便地操纵 Microsoft Office 97/2000 的文档和程序，这在进行企业级开发时是很重要的。

为了帮助广大用户全面、准确地掌握 C++ Builder 5 的编程思想和用法，作者专门编写了这套《C++ Builder 5 高级编程技术》。这套丛书与作者以前编写的《C++ Builder 4 高级编程丛书》之间有着继承性，但在内容上，因该软件版本的升级又有很大的区别。这套《C++ Builder 5 高级编程技术》可以使读者在掌握了 C++ Builder 4 编程技术以后，进一步学习使用新版本软件编程的方法和技巧。考虑到很多程序员已经初步掌握了 C++ Builder 5 的基本用法，因此本套丛书内容的重点放在了编程技术的进一步精通和提高上。

本套丛书分为 4 册。第 1 册介绍 IDE 与 OOP 编程，第 2 册介绍 GUI 编程，第 3 册介绍 Database 与 MIDAS 编程，第 4 册介绍 COM、CORBA 与 Internet 编程。

本书是此套丛书的第 4 册，全面深入地介绍了 COM 与 Interface、ActiveX 框架、类型库、COM 客户与 COM 服务器、ActiveX 控件、OLE Automation、Active Server Page、MTS 或 COM+、CORBA、WinSock、连接 Internet、Internet 协议、Web 服务器扩展和 MIDAS Web 应用程序等内容。

本书主要由徐新华编写，另外，参加本书编写工作的还有顾洪均、凌晨、张莉、郭平等人。由于我们的水平有限，再加上时间很紧，因此，尽管我们作了比较严格的审核和测试，但书中还是难免会有一些错误，敬请广大读者不吝赐教，我们谨在此表示感谢。

为了帮助广大程序员更好地掌握 C++ Builder 5 这个优秀的开发工具，北京东大阿尔发软件技术有限公司愿意为购买此书的读者提供咨询。

与作者的联系方法如下：

地址：北京市上地信息产业基地上地村路 1 号(100085)
电话：(010)62987260　传真：(010)62985141
网址：http://www.allfa.com.cn　邮件：books@allfa.com.cn

作 者

2000 年 8 月

目　录

第一章　COM 与 Interface

组件对象模型(Component Object Model，简称 COM)是组件对象之间相互接口的规范，凡是遵循 COM 接口规范的对象彼此之间都能相互通信和交互，即使这些对象是由不同的厂商、不同的语言、不同的 Windows 版本甚至是在不同的机器上编写和建立的。

C++ Builder 支持 COM 接口规范，C++语言间接地支持对象接口的语法。用 C++ Builder 创建的 COM 对象还可以工作在 MTS(Microsoft Transaction Server)环境中。本章将详细介绍组件对象模型的体系结构以及虚拟方法表、分派接口、双重接口的基本概念。

1.1　COM 的基本概念

软件重用是业界追求的目标，人们一直希望能够像搭积木一样随意“装配”应用程序，组件对象就充当了积木的角色。所谓组件对象，实际上就是预定义好的、能完成一定功能的服务或接口。问题是，这些组件对象如何与应用程序、如何与其他组件对象共存并相互通信和交互？这就需要制定一个规范，以便让这些组件对象能按统一的标准方式工作。

COM 是个二进制规范，它与源代码无关。这样，即使 COM 对象是由不同的编程语言创建，是运行在不同的进程空间和不同的操作系统平台上，这些对象也能相互通信。COM 既是规范，也是实现，它以 COM 库(OLE32.dll 和 OLEAut32.dll)的形式提供了访问 COM 对象核心功能的标准接口以及一组 API 函数，这些 API 函数用于创建和管理 COM 对象。

COM 本质上仍然是客户/服务器模式。客户请求创建 COM 对象并通过接口操纵 COM 对象，服务器响应客户的请求创建并管理 COM 对象。客户和服务器这两种角色不是绝对的。

组件对象与一般意义上的对象既相似也有区别。一般意义上的对象是一种把数据和操纵数据的方法封装在一起的数据类型的实例，而组件对象则是使用接口(Interface)而不是方法来描述自己并提供服务。所谓接口，其精确定义是“基于对象的一组语义上相关的功能”，实际上是一个纯虚类，真正实现接口的是接口对象(Interface Object)。一个 COM 对象可以只有一个接口，例如 Windows 95/98 外壳扩展；也可以有许多接口，例如 ActiveX 控件一般就有多个接口，客户可以从很多方面来操纵 ActiveX 控件。

接口是客户与服务器通信的唯一途径。如果一个组件对象有多个接口，则通过一个接口不能直接访问其他接口。但是，COM 允许客户调用 COM 库中的 QueryInterface()去查询组件对象所支持的其他接口。从这个意义上讲，组件对象就像接口对象的经纪人。

在调用 QueryInterface()后，如果组件对象正好支持要查询的接口，则 QueryInterface()将返回该接口的指针。如果组件对象不支持该接口，则 QueryInterface()将返回一个出错信息。所以，QueryInterface()是很有用的，它可以动态了解组件对象所支持的接口。

接口是面向对象编程思想的一种体现，它隐藏了 COM 对象实现服务的细节。COM 对象

可以完全独立于访问它的客户，只要接口本身保持不变。如果需要更新接口，可以重新定义一个新的接口，对于使用老接口的客户来说，代码得到了最大程度的保护。

1997 年，Microsoft 宣布了雄心勃勃的 COM+计划。COM+可以把类型库整合到 C++编译器中，并且增加一个垃圾回收运行期环境，这样就不再需要 IUnknown 接口的 AddRef()和 Release()。此外，COM+还计划把 Microsoft Transaction Server(MTS)整合到 COM 库中。

未来的操作系统越来越像一组服务，这些服务的形式就是 COM 接口。实际上，Windows 2000 已经初步证实了这一点。例如，Word 和 Excel 完全可以通过 COM 来操纵。同样，Internet Information Server、Internet Explorer 以及许多最新的功能诸如 Active Directory、DAO、Microsoft Transaction Server、Microsoft Message Queue 等都可以通过 COM 来访问。

要说明的是，COM 目前只能基于 Windows 平台。不过，Software AG 公司已经在试验把 COM 引入到 UNIX 中，并且发布了第一个商业化的 DCOM for UNIX。

1.2 客户与服务器

COM 本质上仍然是客户/服务器模式。COM 服务器实际上是组件对象的容器，而组件对象用于向 COM 客户提供服务。COM 客户通常是 EXE，也可能是 DLL，甚至就是 Windows 自己。COM 客户一般独立于 COM 服务器，它知道它所获得的服务，但并不知道 COM 服务器实现这些服务的细节，甚至对有没有这样的 COM 服务器都不知道。

当一个客户请求某个 COM 对象的服务时，客户需要传递一个类标识符(CLSID)，请求 Windows 去查找组件对象在哪里。如果 Windows 找到一个组件对象，就把接口的指针传递给客户。Windows 将从注册表中查找 COM 服务器的位置并定位一个合适的 COM 对象。

根据 COM 服务器与 COM 客户是否运行在同一个进程地址空间，COM 服务器分为 3 种，分别是 In-Process 服务器、Out-of-Process 服务器和 Remote 服务器。

In-Process 服务器通常是 DLL，它可以输出 COM 对象，并映射到客户的进程地址空间中运行。例如，一个嵌入在 Web 网页中的 ActiveX 控件是与 Internet Explorer 在同一个进程地址空间中运行的。对于 In-Process 服务器来说，客户可以直接调用 COM 对象的接口。

Out-of-Process 或称 Local 服务器通常是 EXE，它与 COM 客户是在同一个机器但在不同的进程地址空间中运行。例如，一个嵌入到 Word 文档中的 Excel 电子表格就是 Local 服务器。

Remote 服务器可以是 EXE 也可以是 DLL，它与 COM 客户运行在不同的机器上。例如，用 C++ Builder 5 编写的应用服务器与“瘦”客户程序通常不在同一个机器上运行。

图 1.1 是 Out-of-Process 服务器和 Remote 服务器的示意图。

Remote 服务器跨越了机器边界，如果仍然用 COM 来描述这种模型显然是不够的。Microsoft 扩展了 COM 模型，推出了称为 Distributed COM(简称 DCOM)的模型。DCOM 最初是在 Visual Basic 4.0 的专业版中被引入的，目前只有 Windows 98/NT 4.0/2000 才真正实现了 DCOM，Windows 95 需要安装一个特殊的软件才能支持 DCOM。

为什么能够跨进程边界甚至跨机器边界访问组件对象呢？这就是 COM 模型中称为 Marshaling 的策略。Marshaling 实际上是一种打包技术，它先把要跨边界传递的函数调用信息进行必要的转换并打包，然后再传递给另一个进程或者另一个机器，到达目的地后再解包并调用当地的组件对象。不过，这种跨进程边界或跨机器边界的传递是相当费劲的。如果 COM

服务器是 DLL 的话，就用不着 Marshaling，因为 COM 客户和 COM 服务器是在同一个进程地址空间中运行。在这种情况下，COM 客户只要搜索虚拟方法表就可以找到要调用的方法。

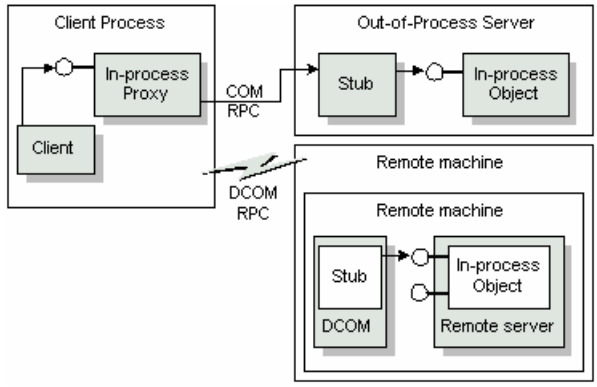

图 1.1　跨进程边界甚至跨机器边界

基于标准的远程过程调用(RPC)，IDispatch 接口提供了通用的 Marshaling 策略。COM 对象也可以自定义 Marshaling 策略。不过，自定义的 Marshaling 策略不具有通用性。

1.3　认识 GUID、CLSID、IID

在一个复杂的系统中，可能充斥着大量的组件对象，每个组件对象可能又有大量的接口。为了保证这些接口彼此不会冲突，Microsoft 规定用 GUID 来标识组件对象和接口。

GUID 是 Globally Unique Identifier 的缩写，意为全局唯一标识符。GUID 可以标识组件对象的类，这时候 GUID 也称为 CLSID(Class Identifier 的缩写)。GUID 也可以标识组件对象的接口，这时候 GUID 也称为 IID(Interface Identifier 的缩写)。

GUID 是一个 128 位的整数(16 字节)，绝对能保证每一个组件对象都具有唯一的标识符。

当使用向导创建一个组件对象或者接口时，向导会自动生成组件对象或接口的 GUID。实际上，GUID 是通过一个叫 CoCreateGUID()的 API 来产生的。CoCreateGUID()会综合考虑当前的日期、时间、CPU 时钟序列、网卡编号以及 Bill Gates 的银行帐号上的余额来生成一个 GUID。如果计算机中安装了网卡，所生成的 GUID 肯定是唯一的，因为每块网卡的编号都是唯一的。如果机器中没有安装网卡，CoCreateGUID()会考虑其他硬件信息。

要创建自己的 GUID，首先需要调用 Windows 的 CoInitialize()来初始化 COM，然后再调用 CoCreateGuid()来得到一个 GUID。也可以用 GUIDGEN 程序来创建一个新的 GUID。C++ Builder 附带的 MFC 示范程序中有 GUIDGEN 程序的源代码。

在 C++ Builder 5 的代码编辑器中，按 Ctrl+Shift+G 键可以生成一个新的 GUID。

1.4 引用计数

引用计数是一种机制，使组件对象具有一定的“智能性”。它的工作原理是这样的：当接口对象第一次创建时，引用计数的初始值为 1。当有一个客户请求获得接口对象的指针时，就调用 AddRef()使该计数加 1。当一个客户不再需要组件对象的服务时，它应当调用 Release()。注意，Release()并不真正释放接口对象，因为可能还有其他客户正在使用接口。Release()只是使引用计数减 1。只有当引用计数正好减为零时，接口对象才被删除。

下面举例说明引用计数的作用。假设客户 A 向服务器请求 IMalloc 接口，服务器收到请求后，首先看该接口对象是否存在。如果没有，就创建一个接口对象，并调用 AddRef()使引用计数变为 1，同时把该接口对象的指针传递给客户 A。假设这时候客户 B 也加入进来，并且也是请求 IMalloc 接口。由于此时 IMalloc 接口对象已存在，服务器只是简单地返回一个指针，并且调用 AddRef()使引用计数变为 2。当客户 A 不再需要 IMalloc 接口时，它就调用 Release()试图释放这个接口。显然，这时候不能删除 IMalloc 接口对象，因为客户 B 还在用着呢。可见，引用计数这种机制使服务器知道如何管理自己的接口。

引用计数这种机制也带来一个问题，就是在调用 AddRef()和 Release()时不能出现混乱。一旦出现混乱，可能导致接口对象永远不被删除或者过早地被删除。

1.5 虚拟方法表

COM 是个二进制规范，任何开发环境只要遵守这个规范，都可以生产出 COM 对象。COM 采用一种称为虚拟方法表(virtual method table)的语法来解决方法调用。不过，COM 接口与 C++的类还是有一些区别的。COM 接口中凡是要表露给客户的方法必须声明为纯虚的，客户得到的只是指向虚拟方法表的指针，具体实现接口的是接口对象。

如果建立了同一个 COM 对象的多个实例，虚拟方法表就是共享的，但每个实例的数据是私有的。图 1.2 是虚拟方法表的示意。

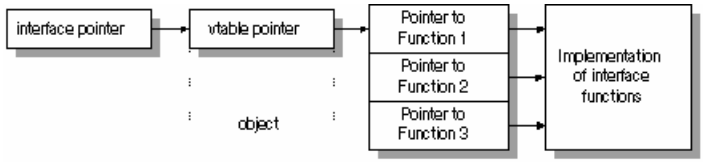

图 1.2 虚拟方法表的示意

对于 COM 对象来说，不管是用 Delphi、C++ Builder 还是用其他工具创建的，虚拟方法在内存中的布局都是一致的，这就是为什么 COM 在二进制上与语言无关的原因。

1.6 接口的语法

C++ Builder 5 的接口能自动与 COM(Component Object Model)和 CORBA(Common Object Request Broker Architecture)规范兼容。由于 COM 具有语言无关性，因此，用 C++ Builder 5 创建的 COM 对象可以与 Delphi、Java、Visual Basic 的对象彼此交互。

在声明接口时，可以指定一个祖先接口(interface heritage)。如果没有指定祖先接口，则默认的祖先接口是 IUnknown。

任何一个接口，除了它自身的成员外，还继承了它的祖先接口中的成员，当然肯定包括 IUnknown 接口中的成员。

在声明接口时，可以给出接口的标识符(interface identification)，即接口的 GUID。

接口标识符是可选的。如果有的话，接口标识符应当出现在接口的任何其他成员之前，并且应当遵循这样的格式：

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

可以看出，接口标识符实际上是一个用方括号括起来的字符串常量。其中，每个 x 是一个十六进制数字(0~9 和 A~F)。每个接口的标识符都应当是唯一的。

接口可以有成员，包括特性(property)和方法(method)。

由于接口可能要在不同语言的模块之间共享，因此，接口中的方法最好采用 stdcall 方式。在双重接口和 CORBA 接口中可以采用 safecall 方式。

下面列出了一个典型的接口，叫 Imalloc：

```
IMalloc = interface(IUnknown)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(cb: Longint): Pointer; stdcall;
    function Realloc(pv: Pointer; cb: Longint): Pointer; stdcall;
    procedure Free(pv: Pointer); stdcall;
    function GetSize(pv: Pointer): Longint; stdcall;
    function DidAlloc(pv: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
end;
```

类名一般是以 T 打头，而接口的名称一般是以 I 打头。例如，上述接口的名称是 IMalloc，其祖先接口是 IUnknown。第二行是一串数字，是这个接口的 GUID。接着又声明了若干个过程和函数，它们的调用约定方式都是 stdcall。IMalloc 接口没有特性。

在声明了接口后，还要具体实现接口，才能被外部代码访问。要实现接口，需要先声明一个特殊的类，即所谓的实现类。实现类的祖先一般是 TInterfacedObject。

一个实现类也可以以另一个实现类为祖先。所有的接口都是从 IUnknown 派生的，因此，任何一个实现类都要实现 AddRef()、Release()和 QueryInterface()。

要实现接口中的方法，必须把接口中的方法映射到一个实现类中，然后在这个实现类中实现这些方法。在默认情况下，可以把接口的方法映射成同名的方法。

1.7 IUnknown 接口

Addref()、Release()和 QueryInterface()是 COM 对象提供的三个核心服务，这三个方法构成了 IUnknown 接口。在 System 单元中，IUnknown 接口是这样声明的：

```
IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;out Obj):HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
end;
```

正如 TObject 是所有类的祖先一样，IUnknown 是所有接口的祖先。这样，凡是取得了接口对象指针的客户总是能访问 COM 对象的核心服务，诸如 AddRef()、Release()和 QueryInterface()，这三个核心服务管理着接口对象的生存期。

_AddRef()和_Release()比较简单，都没有参数。而 QueryInterface()则比较复杂，它有两个参数：一个是 IID 参数，用于指定要查询的接口；另一个是 Obj 参数，用于返回找到的接口对象的指针。如果 COM 对象不支持所查询的接口，则 Obj 参数将返回 nil。

AddRef()和 Release()前均加了一个下划线前缀，这是为了更加醒目。过去，COM 对象必须自己维护引用计数，也就是说，必须调用 AddRef()和 Release()来把引用计数加 1 或减 1。COM 的另一个核心服务 QueryInterface()也是不可缺少的，客户只有调用 QueryInterface()才能申请到一个接口指针。不过，从 C++ Builder 3 开始，由于采用了 ActiveX 框架，引用计数是由 TComObject 对象自动维护的，所以应用程序不再需要直接与 IUnknown 接口打交道。

这里顺便介绍一下 COM 模型中称为 Interface Aggregation 的概念。面向对象的编程思想允许通过继承(Inheritance)来实现软件重用。在 COM 模型中没有继承的概念，而是通过 Interface Aggregation 技术把多个接口聚合起来，共同完成某一复杂的功能。

1.8 分派接口

OLE Automation 对象实际上就是实现分派接口的对象。与一般的接口相比，声明分派接口的文法稍有不同。

分派接口是用保留字 dispinterface 来声明的，并且不需要也不能指定祖先接口。在声明分派接口时可以指定接口标识符，即 GUID。

分派接口的成员可以是特性，也可以是方法。在声明分派接口的成员时，可以用 dispid 指示字再跟一个整数常量来指定成员的识别号。

分派接口中的特性不能有 read 和 write 子句，但可以用 readonly 或 writeonly 指示字来表示特性是只读的或是只写的，也可以用 default 指示字来表示该特性是接口的默认特性。

下面列出了一个典型的分派接口，叫 IStringsDisp：

```
IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
```

```
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
end;
```

与一般的接口不同的是，分派接口不需要也不能用一个类去实现。换句话说，在声明一个实现类时，分派接口不能出现在它的祖先列表中。这是因为，分派接口的方法仅仅充当了 IDispatch **接口中** Invoke()**的原型，并不需要实现。**

由于分派接口的方法实际上并不被调用，因而不需要指定调用约定方式(诸如 stdcall **等)。另外，方法的参数类型和返回类型(如果有的话)必须与** OLE Automation **兼容。**

dispid **指示字后面必须跟一个整数常量，用于给分派接口中的成员指定一个识别号。如果不用** dispid **指示字，则编译器会自动给成员指定一个识别号，并且保证每个成员的识别号是唯一的。当外部代码中出现成员的名称时，**IDispatch **接口就用成员的识别号来表示。实际上，**IDispatch **接口是通过调用** GetIDsOfNames()**来把名称转换为识别号的。**

一旦获得了成员的识别号，外部代码就可以调用 IDispatch **接口中的** Invoke()**。**Invoke() **实际上是裹在成员身上的外套，它的第一个参数是实际要访问的成员的识别号，这就是为什么要调用** GetIDsOfNames()**来获取识别号的原因。**

既然 Invoke()**只是成员的外套，而成员的参数又是不固定的，那么，如何传递成员的参数和返回值呢？不用担心，传递给** Invoke()**的参数个数和数据类型可以是不固定的。**

分派接口的最大优势在于，它内建了对 Marshaling **的支持，从而使跨进程边界和跨机器边界的通信成为可能。更重要的是，**Marshaling **是自动的。也就是说，不管是** Automation **服务器，还是** Automation **客户，程序员都可以不管** Invoke()**的细节。**

对于 Automation **对象来说，并不需要事先知道它的类型库，因为** IDispatch **接口的** GetIDsOfNames()**能够在运行期把名称转换为识别号。同样，在编译期，编译器也不进行类型匹配检查。不过，这样一来，就有可能出现调用失败。要说明的是，凡是用** C++ Builder 5 **提供的向导所创建的** Automation **对象总是有类型库的，这是为了避免过度的盲目性。**

如果 Automation **对象有类型库的话，则成员的识别号在编译期就是确定的。这时候，可以直接在程序中读取或指定识别号，这就是所谓的“**ID Binding**”。由于这时候只需要调用** Invoke()**，而不需要调用** GetIDsOfNames()**，所以这使得程序的速度加快。**

“ID Binding**”的另一个好处是，编译器可以进行类型匹配检查。凡是有错误的代码在编译期就被报告出来，而不是在运行期才发现。**

不过，“ID Binding**”方式也有一个缺点，就是直接用识别号来调用成员不太直观，而且无法检查识别号和名称是否对应。**

1.9 双重接口

在 OLE Automation 操作中，有的 Automation 客户希望通过虚拟方法表来访问 Automation 对象，而有的 Automation 客户则希望通过 IDispatch 接口来访问 Automation 对象。为了同时支持上述两种方式，从 C++ Builder 3 开始便引入了双重接口的概念。

双重接口能够以 OLE Automation 方式同时支持编译期虚拟方法表绑定和运行期动态绑定。也就是说，双重接口是同时以虚拟方法表和 IDispatch 接口两种方式来表露它的方法。

虚拟方法表实际上是一块内存区域，它是由编译器建立的，存放了 Automation 对象的成员地址。虚拟方法表的头三个域是 IUnknown 接口的三个核心服务，即 AddRef()、Release() 和 QueryInterface()。如果 Automation 对象支持 IDispatch 接口，则接下来的域就是 IDispatch 接口的四个方法，然后才是 Automation 对象本身的方法。图 1.3 显示了虚拟方法表的结构。

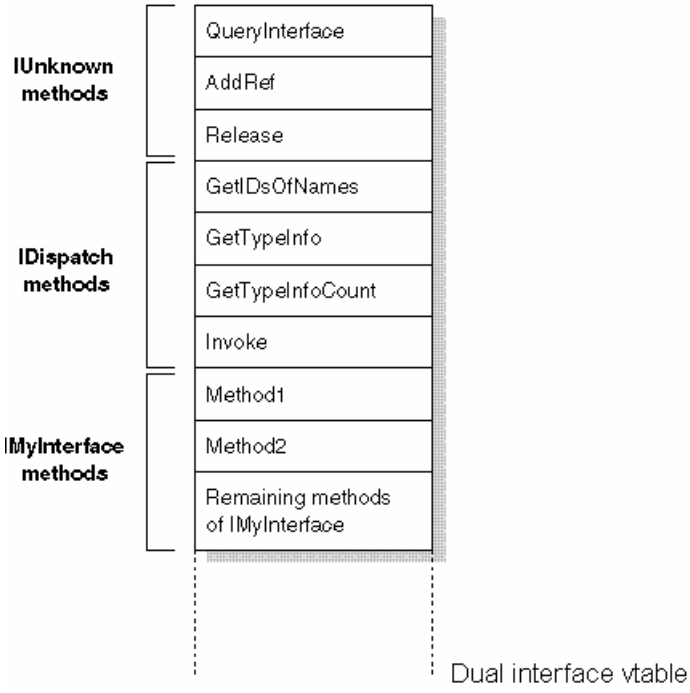

图 1.3 虚拟方法表的结构

双重接口必须直接或间接继承于 IDispatch 接口。在 System 单元中，IDispatch 接口是这样声明的：

```
IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
        NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer; Flags: Word;
```