OPCDA 服务器与客户程序开发指南

修订本（第二版）

司纪刚

南大傲拓科技（北京）有限公司
Nanda Automation Technology

序

****

最初接触 OPC 时，是在 2002 年，公司因为工程功能的需要定购了 RockWell 公司的 SLC500 PLC，RockWell 公司提供了 Rslinx 作为 OPC 服务器，即提供 OPC 接口供其它厂家通讯使用。因此为适应公司工程功能的需要，即使用 RockWell 公司提供的 OPC 服务器接口，需要采用 Visual C++6. 0 作为编程语言，通过自定义接口来开发客户端程序。当时由于有关 OPC 的资料非常少，从个人的角度来看，走了很多弯路，主要原因在于采用 C++需要了解很多的 COM 知识，如接口等等，不象采用 VB 这样的语言相对简单一些，在此之前对 COM 基本不了解。

中国工控网（http://www.gongkong.com）在某个时期有了 OPC 的论坛，当时我已经编写调试完 OPC 的客户程序(DA1.0,DA2.0)，也时不时的到论坛上去看其他技术人员的帖子，同时也回复一些帖子，但不是很多。

到 2004 年，个人想尝试开发 OPC 服务器，于是开始了 OPC 服务器的编写，编写过程中最大的困难在于需要对 COM 有很好的理解，编写服务器程序和客户程序的工作量是完全不同的。编程时花时间最多的是连接点的实现，由于国内 ATL 编程的书籍非常少，每一步都需要自己看 MSDN，每一步的调试成功都有很多的乐趣，由此也加深了对 COM 编程的理解，加深了对 OPC 规范的理解。

最初并没有想到写一本书，2004 年的某一天在中国工控网上看到一位网友的留言，说我的回复给了他很大的帮助，我也不知道他是谁，但他的留言让我认为可以把自己的一些经验记录下来，或许可以给大家更

多的帮助。我属于那种韧性不是很强的人，非常害怕自己不能坚持下去。幸运的是终于把书写完了，虽然我通读全稿，并不满意，但毕竟是我第一次写这样的技术书籍，书中的错误在所难免，希望读者能够指正。

本书可以完成得到了很多朋友的帮助，也非常感谢很多编程技术网站。本书的技术支持网站是 http://www.opc-china.com，请各位读者把对本书的意见，建议发到技术支持网站上，以便我对本书修正，从而把更好的版本奉献给大家。

司纪刚

2005 年 5 月于南京

第一次修改版序

****

从 2005 年 5 月到 2006 年 11 月，本书得到了 200 多位朋友的订阅。由于个人工作的原因，并没有完全地对各位朋友的问题进行回复，也有很多位朋友对本书提出了很好的建议，对本书的某些方面也提出了很有见地的建议。

作为国内第一本由国人编写的 OPC 技术专业书籍，此次修改将主要针对 OPC 服务器的介绍，OPC 客户程序的编写，和 DCOM 的配置，仍然以 OPCDA2.0 规范为基础。

由于国家对 BBS 的管理需要，http://www.opc-china.com 已经关闭，谢谢几年来支持的朋友。本次修改版本发表到互联网上，供有兴趣的朋友阅读、参考。

司纪刚

2008 年 1 月

前言

****

OPC是一种最新的工业控制标准，目前已有许多的资料在介绍OPC。但是有关OPC服务器和客户程序开发的资料相对较少，因此为推动OPC技术的普及，本书旨在提供OPC服务器和客户程序的开发指南。

作为工业控制领域的应用技术，OPC服务器实际上涉及到的领域不仅仅是COM接口的实现，还包括如何将实时数据库技术融入OPC服务器等。对于本书而言，主要实现了OPC服务器的常用接口和OPC客户程序。

OPC服务器的编写借鉴了GE DEMO OPC SERVER（OPC1.0规范）的程序结构。

本书为谁编写

本书不是为所有的读者编写的，而是为研究OPC技术的人员编写的。

•本书不适合新程序员和Visual C++，Visual Basic的初学者，本书的论题与OPC开发有关。如果你还不了解COM，你应该先找一本COM技术的书籍，本书仅仅介绍了有关的概念。

•本书是专门为那些希望学习、研究OPC技术的人员所编写的。

•本书内容也适合进行COM应用程序开发的人员。

•本书提供开发 OPC 服务器的一个简单示例，研究开发人员需具有MFC编程基础和COM基础。如果您只想学习客户端程序，可以略过OPC服务器开发的章节。如果您想了解 OPC 服务器与客户端的详细协作过

程，您可以采用调试设置断点的方法来测试协作过程。

本书首先对ATL进行介绍，以便了解如何编写OPC服务器应用程序；其次采用 Visual C++ 6.0 环境详细介绍了 OPC 服务器的实际实现过程；最后采用 Visual C++ 6.0，Visual Basic6.0 环境，详细介绍了 OPC 客户程序的编写方法，并给出了代码的实现。

如何使用本书

本书共分为五个部分

第一部分：OPC 规范概述

本部分包括：OPC 的技术本质，OPCDA204 规范简介，OPC 对象接口定义，OPC 同步异步通讯，OPC 服务器开发方式。

第二部分：ATL 简介

本部分包括：ATL 简介，用 ATL 开发组件应用程序，测试组件应用程序。

第三部分：ATL 开发 OPC 服务器

本部分包括：OPC 服务器的设计思路，OPC 服务器的接口实现。

第四部分：OPC 客户程序开发

本部分包括：基于 Visual C++6.0 的 OPC 客户程序（同步，异步，浏览地址空间，多组）开发；基于 Visual Basic6.0 的 OPC 客户程序（同步，异步，浏览地址空间）开发。

第五部分：OPC 服务器的远程访问

本部分包括：介绍 DCOM 技术；OPC 服务器远程访问的 DCOM 配置；DCOM 的远程连接管理；OPC 客户程序远程访问的实现(Visual C++6.0)。

目录

第 1 章 OPC 概述

****

关键字：COM DCOM OPC DA 通讯 规范 CLIENT SERVER GROUP ITEM 自定义接口 自动化接口 同步 异步 回调

随着计算机科学技术、工业控制等各方面新技术的迅速发展，计算机监控系统由早期的集中式监控向全分布式的方向发展，计算机监控系统软件随着面向对象技术、分布式对象计算、多层次 Client/Server 技术的成熟，也从早期面向功能的系统软件，发展为面向具体现场设备为特征的面向对象的监控系统软件。

同时，计算机监控系统规模越来越大，不同厂家生产的现场设备的种类在不断增加，由于不同厂家所提供的现场设备的通讯机制并不尽相同，计算机监控系统软件需要开发的硬件设备通信驱动程序也就越来越多，造成了硬件通讯驱动程序需要不断开发的现象，而基于 COM/DCOM 技术的 OPC 技术，提供了一个统一的通讯标准，不同厂商只要遵循 OPC 技术标准就可以实现软硬件的互操作性。

OPC（OLE for Process Control，用于过程控制的 OLE）是为过程控制专门设计的 OLE 技术，由一些世界上技术占领先地位的自动化系统和硬件、软件公司与微软公司（Microsoft）紧密合作而建立的，并且成立了专门的 OPC 基金会来管理，OPC 基金会负责 OPC 规范的制定和发布。OPC 提出了一套统一的标准，采用典型的 CLIENT/SERVER 模式，针对硬件设备的驱动程序由硬件厂商或专门的公司完成，提供具有统一 OPC 接口标准的 SERVER 程序，软件厂商只需按照 OPC 标准编写 CLIENT 程序访问（读/写）SERVER 程序，即可实现与硬件设备的通信。

如图 1.1 所示，与传统的通讯开发方式相比，OPC 技术具有以下优势：

• 硬件厂商熟悉自己的硬件设备，因而设备驱动程序性能更可靠、效率更高。

• 软件厂商可以减少复杂的设备驱动程序的开发周期，只需开发一套遵循 OPC 标准的程序就可以实现与硬件设备的通信，因此可以把人力、物力资源投入到系统功能的完善中。

• 可以实现软硬件的互操作性。

• OPC 把软硬件厂商区分开来，使得双方的工作效率有了很大的提高。

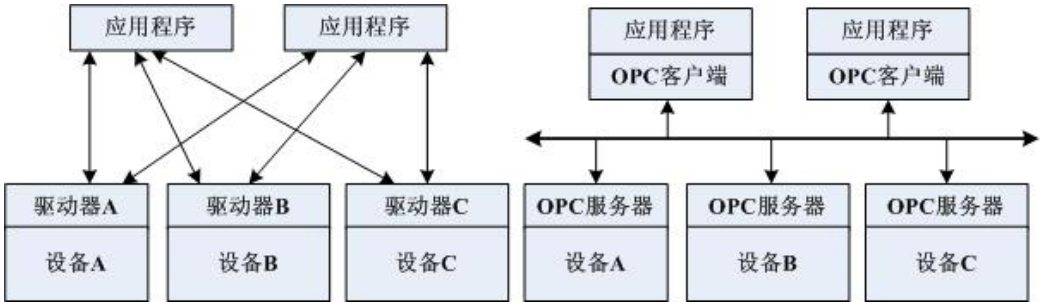


图 1.1 OPC 技术应用前后比较

因此 OPC 技术的出现得到了广大软硬件厂商的支持，并迅速发展。自从 1997 年 9 月发布 OPC DA1.0 规范以来，经过多年的发展，OPC 规范已经被工控领域大多数厂商接受，并成了工控软件的技术标准。目前 OPC 规范主要有 DA（Data Access）规范，AE(alarm and event)规范，HDA(history data access)规范等。而且随着 OPC 技术与企业整体信息系统集成的需求变得日益迫切，对 OPC 技术的跨平台性能和 Internet 特性

提出了更高要求。为此，OPC 基金会开始以 XML 为基础着手制定一系列新的标准。2002 年 3 月 OPC 基金会正式发布了 OPC XML-DA 规范, 并与 2004 年 12 月正式发布了 OPC XML-DA1.01 规范，为 OPC 进一步提高工业控制系统的互操作性揭开了新的篇章。本书仅仅以符合 DA 规范的 OPC 服务器和客户程序为例介绍 OPC 技术，对于其它规范的 OPC 技术，本书未能介绍。

1.1 OPC 技术的本质——COM/DCOM

随着计算机网络技术的发展，计算机监控系统也普遍的采用了分布式结构，因而系统的异构性是一个非常显著的特点。OPC 技术本质是采用了 Microsoft 的 COM/DCOM（组件对象模型/分布式组件对象模型）技术，COM 主要是为了实现软件复用和互操作，并且为基于 WINDOWS 的程序提供了统一的、可扩充的、面向对象的通讯协议，DCOM 是 COM 技术在分布式计算领域的扩展，使 COM 可以支持在局域网、广域网甚至 Internet 上不同计算机上的对象之间的通讯。

COM 是由 Microsoft 提出的组件标准，它不仅定义了组件程序之间进行交互的标准，并且也提供了组件程序运行所需的环境。在 COM 标准中，一个组件程序也被称为一个模块，它可以是一个动态链接库，被称为进程内组件(in-process component)；也可以是一个可执行程序(即 EXE 程序)，被称作进程外组件(out-of-process component)。一个组件程序可以包含一个或多个组件对象，因为 COM 是以对象为基本单元的模型，所以在程序与程序之间进行通信时，通信的双方应该是组件对象，也叫做 COM 对象，而组件程序(或称作 COM 程序)是提供 COM 对象的代码载体。

COM 标准为组件软件和应用程序之间的通信提供了统一的标准，包括规范和实现两部分，规范部分规定了组件间的通信机制。由于 COM 技术的语言无关性，在实现时不需要特定的语言和操作系统，只要按照 COM 规范开发即可。然而由于特定的原因，目前 COM 技术仍然是以 Windows 操作系统为主，在非 Windows 操作系统上开发 OPC，具有很大的难度。

COM 的模型是 C/S（客户/服务器）模型，OPC 技术的提出就是基于 COM 的 C/S 模式，因此 OPC 的开发分为 OPC 服务器开发和 OPC 客户程序开发，对于硬件厂商，一般需要开发适用于硬件通讯的 OPC 服务器，对于组态软件，一般需要开发 OPC 客户程序。

对于 OPC 服务器的开发，由于多种编程语言在实现时都提供了对 COM 的支持，如 Microsoft C/C++,Visual Basic, Borland 公司的 Delphi 等。但是开发 OPC 服务器的语言最好是 C 或者是 C++语言。在本书中选用 Visual C++6.0 为开发语言。

对于 OPC 客户程序的开发，可根据实际需求，选用比较合适的，能够快速开发的语言。

1.2 OPCDA204 规范简述

OPCDA204 规范（OPC Data Access Custom Interface Specification 2.04）是 2000 年 9 月 OPC 基金会发布的 OPCDA 自定义接口规范。该规范制定了 OPC 服务器和 OPC 客户程序的 COM 接口标准，通过制定标准的接口来实现多个厂家的 OPC 服务器和 OPC 客户程序开发。本书附带 OPCDA204 规范的 WORD 文档。

1.2.1 OPC 客户程序和 OPC 服务器

一个 OPC 客户可以连接一个或多个 OPC 服务器，而多个 OPC 客户也可以同时连接同一个 OPC 服务器，如图 1.2 所示。

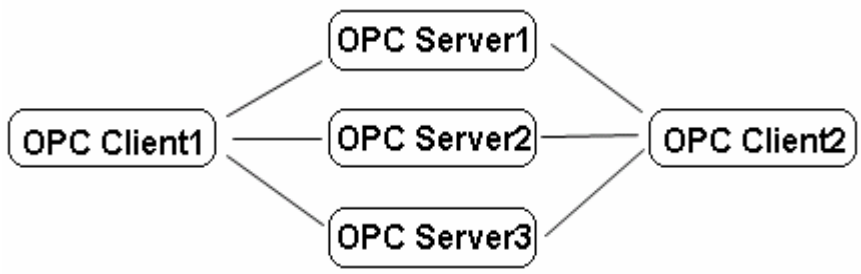


图 1.2 OPC 客户程序/OPC 服务器关系

1.2.2 OPC 服务器的对象组成

一个 OPC 服务器由三个对象组成：服务器(Server)，组(Group)，项(Item)。OPC 服务器对象用来提供关于服务器对象自身的相关信息，并且作为 OPC 组对象的容器。OPC 组对象用来提供关于组对象自身的相关信息，并提供组织和管理项的机制。

OPC 组对象提供了 OPC 客户程序用来组织数据的一种方法。例如一个组对象代表了一个 PLC（可编程控制器）中的需要读写的寄存器组。一个客户程序可以设置组对象的死区，刷新频率，需要组织的项等。OPC 规范定义了 2 种组对象：公共组和私有组。公共组由多个客户共享，局部组只隶属于一 OPC 客户。全局组对所有连接在服务器的应用程序都有效，而私有组只能对建立它的 CLIENT 有效。在一个 SERVER 中，可以有若干个组。

OPC 项代表了 OPC 服务器到数据源的一个物理连接。数据项是读写数据的最小逻辑单位。一个 OPC 项不能被 OPC 客户程序直接访问，因此在 OPC 规范中没有对应于项的 COM 接口，所有与项的访问需要通过包含项的 OPC 组对象来实现。简单的讲，对于一个项而言，一个项可以是 PLC 中的一个寄存器，也可以是 PLC 中的一个寄存器的某一位。在一个组对象中，客户可以加入多个 OPC 数据项。每个数据项包括 3 个变量：值（Value）、品质(Quality)和时间戳（Time Stamp）。数据值是以 VARIANT 形式表示的。

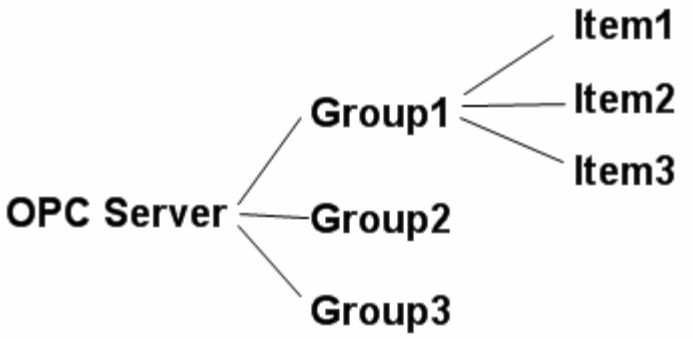


图 1.3 Server/Group/Item 关系

这里最需要注意的是项并不是数据源，项代表了到数据源的连接。例如一个在一个 DCS 系统中的 TAG 不论 OPC 客户程序是否访问都是实际存在的。项应该被认为是到一个地址的数据。大家一定要注意项的概念。不同的组对象里可以拥有相同的项，如组 1 中有对应于一个开关的 ITEMAAA，组 2 中也可以有同样意义对应于一个开关的 ITEMAAA，即同样的项可以出现在不同的组中。

1.2.3 OPC 接口体系

OPC 规范提供两种接口：自定义接口（the OPC Custom Interfaces），自动化接口（the OPC Automation interfaces）。

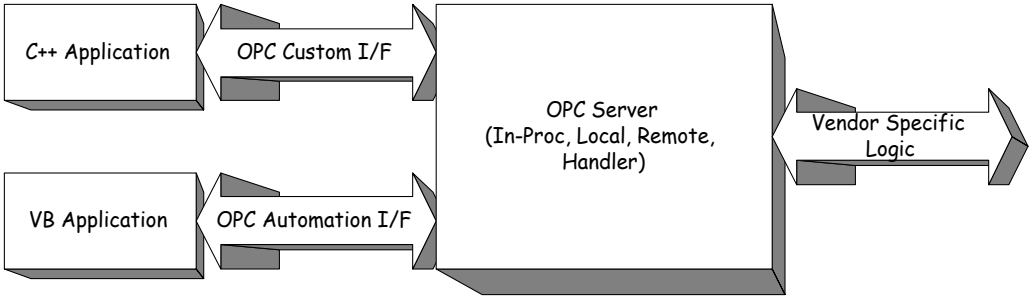


图 1.4 OPC 接口

如前所述，象所有的 COM 结构一样，OPC 是典型的 CLIENT/SERVER 结构，OPC 服务器提供标准的 OPC 接口供 OPC 客户程序访问。OPC 服务器必须提供自定义接口，对于自动化接口，在 OPC 规范定义中是可选的。

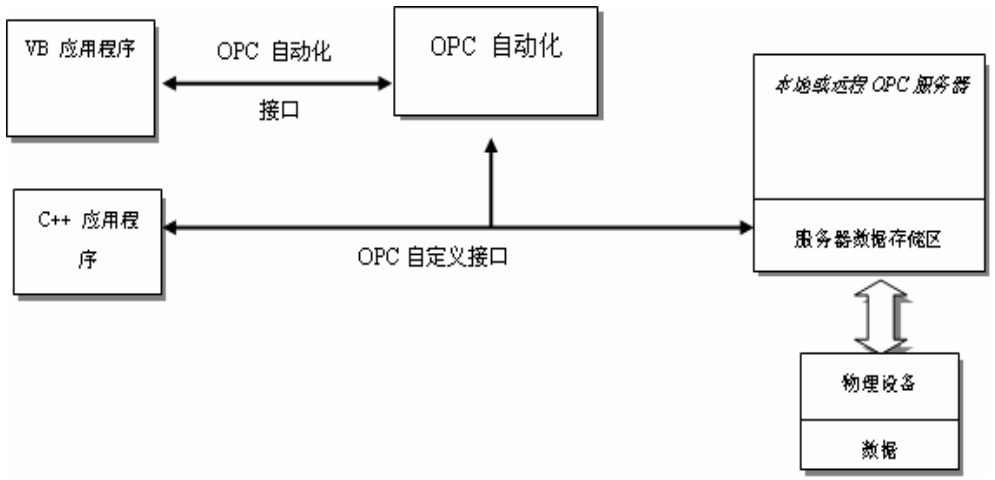


图 1.5 典型 OPC 结构

1.3 OPC 对象接口定义

本节主要对 OPC 服务器对象和 OPC 组对象的接口进行简要的介绍。

OPC 服务器对象提供一些方法去读取或连接一些数据源。OPC 客户程序连接到 OPC 服务器对象，并通过标准接口与 OPC 服务器联系。OPC 服务器对象提供接口（AddGroup）供 OPC 客户程序创建组对象并将需要操作的项添加到组对象中，并且组对象可以被激活，也可以被赋予未激活状态。对于 OPC 客户程序而言，所有 OPC 服务器和 OPC 组对象可见的仅仅是 COM 接口。

下面的两个图例是 OPC 规范中定义的 OPC 服务器对象和 OPC 组对象的 COM 接口，其中任选的接口均以[ ]表示。（注：任选指开发 OPC 服务器时，这些接口可以根据实际情况选择实现还是不实现，除任选项外的接口在开发时必须全部实现。）

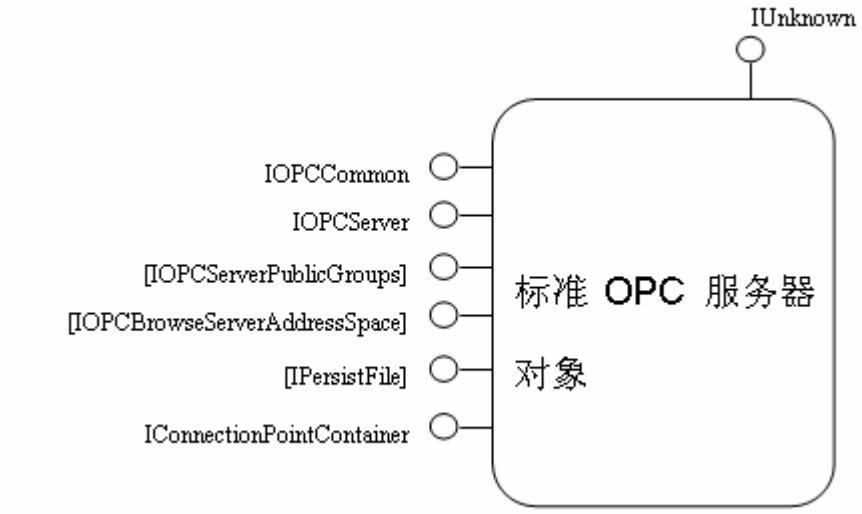


图 1.6 标准 OPC 服务器对象及接口

IOPCServerPublicGroups，IOPCBrowseServerAddressSpace 和

IPersistFile 为任选（optional）接口，OPC 服务器提供商可根据需要选择是否需要实现。其它接口为 OPC 服务器必须实现的接口。其中：IOPCServerPublicGroups 接口用于对公共组进行管理。IPersistFile 接口允许用户装载和保存服务器的设置，这些设置包括服务器通信的波特率、现场设备的地址和名称等，这样用户就可以知道服务器启动和配置的改变而不需要启动其它的程序。

IOPCBrowseServerAddressSpace 允许用户浏览服务器中的有用的组员的数据，为用户提供 OPC 服务器各个组员的定义列表。IOPCCommon 接口是其它 OPC 服务器（例如 OPC 报警与事件服务器）也使用的接口。通过该接口可为某个特定的客户/服务器对话（session）设置和查询本地标识（LocateID）。这样，一个客户程序的操作将不会影响其它客户程序。IConnectionPointContainer 接口服务器（OPC 服务器对象接口）支持可连接点对象，当 OPC 服务器关闭时需要通知所有的客户程序释放 OPC 组对象和其中的 OPC 组员，此时可利用该接口调用客户程序方的 IOPCShutdown 接口实现服务器的正常关闭。

IOPCServer 接口及成员函数主要用于对组对象进行创建、删除、枚举和获取当前状态等操作。是 OPC 服务器对象的主要接口。接口及成员函数定义为：

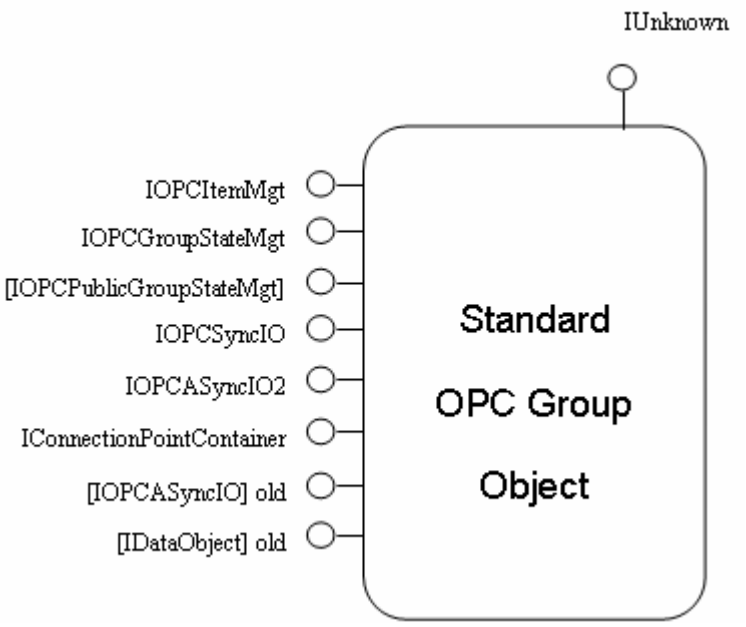


图 1.7 标准 OPC 组对象及接口

其中：IOPCItemMgt 接口及成员函数用于 OPC 客户程序添加、删除和组对象中组员等控制操作。IOPCGroupStateMgt 接口及其成员函数允许 OPC 客户程序操作或获取用户组对象的全部状态（主要是组对象的刷新率和活动状态，刷新率的单位为毫秒）。IOPCPublicGroupStateMgt 为任选接口，用于将私有组对象（private group）转化为公有组对象（public group），这个接口一般不用，在很多商业的 OPC 服务器中，此接口都没有开发。可选接口 IOPCAsyncIO 和 IdataObject 接口用于异步数据传输（在 OPC 数据访问规范 1.0 中定义，现在其功能已经被 IOPCAsyncIO2 和 IConnectionPointContainer 接口取代）。IOPCSyncIO 用于同步数据访问。IOPCAsyncIO2 用于异步数据访问。这两个接口是数据访问规范进行数据访问最重要的接口。

有关 OPC 服务器对象和 OPC 组对象的 COM 接口详细定义请看 OPC

规范定义，除在开发实例中用到的 COM 接口，其它接口本文不再详述。

1.4 OPC 同步异步通讯

OPCDA 规范规定了两种通讯方式：同步通讯和异步通讯。这两种通讯方式与常见的串口同步通讯、异步通讯以及以太网的同步通讯、异步通讯的功能差不多。

同步通讯时，OPC 客户程序对 OPC 服务器进行相关操作时，OPC 客户程序必须等到 OPC 服务器对应的操作全部完成以后才能返回，在此期间 OPC 客户程序一直处于等待状态，如进行读操作，那么必须等待 OPC 服务器完成读后才返回。因此在同步通讯时，如果有大量数据进行操作或者有很多 OPC 客户程序对 OPC 服务器进行读、写操作，必然造成 OPC 客户程序的阻塞现象。因此同步通讯适用于 OPC 客户程序较少，数据量较小时的场合。

异步通讯时，OPC 客户程序对服务器进行相关操作时，OPC 客户程序操作后立刻返回，不用等待 OPC 服务器的操作，可以进行其他操作。当 OPC 服务器完成操作后再通知 OPC 客户程序，如进行读操作，OPC 客户程序通知 OPC 服务器后离开返回，不等待 OPC 服务器的读完成，而 OPC 服务器完成读后，会自动的通知 OPC 客户程序，把读结果传送给 OPC 客户程序。因此相对于同步通讯，异步通讯的效率更高，适用于多客户访问同一 OPC 服务器和大量数据的场合。

OPC 的异步通讯有四种方式：

• 数据订阅，客户端通过订阅方式后，服务器端将变化的数据通过回调传送给客户程序。

• 异步读，返回操作结果和数据值。

- 异步写，返回操作结果，成功、失败。
- 异步刷新，异步读所有 Item 的值

1.5 OPC 服务器开发方式

OPC 服务器本身就是一个可执行程序，该程序以一定的速率不断地同物理设备进行数据交互。服务器内有一个数据缓冲区，其中存有最新的数据值，数据质量戳和时间戳。OPC 数据服务器的设计与实现是一个较为复杂与繁重的任务，设计者既需要熟悉 OPC 规范，同时也必须掌握相应的硬件产品特性。OPC 数据服务器大致可以分解为不同的功能模块。OPC 对象接口管理模块，Item 数据项管理模块以及服务器界面和设置等等。一个设备的 OPC Server 主要有两部组成，一是 OPC 标准接口的实现，二是与硬件设备的通信模块。

虽然 COM 技术本质上具有语言无关性，可以用各种语言开发，但由于最适合 COM 开发的语言仍然是 C++，因此一般都选择采用 Visual C++进行开发。

目前用 Visual C++开发 COM 组件主要有三种方式：使用 COM SDK 直接开发 COM 组件；通过 MFC 提供的 COM 支持实现 COM 组件；通过 ATL 来实现 COM 组件。

此外，目前国内外很多的工控软件厂商也推出了一系列的 OPC 快速开发工具包，使用专门的 OPC 开发工具包，开发者只需具备基本的编程基础即可快速上手，无需掌握 ATL，COM/DCOM，也无需了解 OPC 技术的细节，而且大多数的 OPC 开发工具都支持多种常用编程语言，如 VB,VC 等。

建议学生或有志向的开发人员可以尝试独立开发 OPC 服务器，如果

是公司使用，建议购买 OPC 服务器开发工具。

重点：何为 OPC？OPCDA 有哪些对象？OPCDA 有哪些接口？OPCDA 的通讯方式？OPCDA 的开发方式？

第 2 章 ATL 简介

****

关键字：ATL 类厂 接口 标识符 IDL 组件 聚合 双重接口 自定义接口 自动化接口 连接点 事件 注册 属性 方法 客户程序

Visual C++从 4.0 版本就已经提供全面的 COM 支持，尤其在 5.0 和 6.0 版本中，不仅 MFC 类库提供 COM 应用的支持，而且 Visual C++的集成开发环境 Visual Studio 也为 COM 应用提供了各种向导支持，并且，Visual C++还提供了另一套模板库 ATL 专门用于 COM 应用程序的开发。在上一章中介绍了采用 Visual C++进行 OPC 服务器开发的几种方式，因为 ATL 是专为 COM 应用程序开发，因此本书的 OPC 服务器开发采用了 ATL 的模式，在本章中首先对 ATL 进行简要介绍，并以实际例子介绍如何进行 ATL 编写 COM 组件。

ATL(Active Template Library)是 Visual C++提供的一套基于模板的 C++类库，利用这些模板类，开发人员可以快速的开发 COM 组件程序。所以说 ATL 专门针对于 COM 应用开发的，它内部的模板类实现 COM 的一些基本特征，比如一些基本 COM 接口 IUnknown, IClassFactory, IDispatch 等，也支持 COM 的一些高级特性，如双接口（dual interface）、连接点(connection point)、Activex 控制等。ATL 最初的设计是快速的开发小型的组件，ATL2.0 版本添加了模板库用来支持可视化的控件开发。

ATL 所具有的特点：

•包含所有 C++的功能。

•无需运行库，除非你想使用它。

•引用计数。

•高水平的对象和接口实现方法。

•类厂自动操作，对象创造，接口查询。

ATL 开发应用程序并不像开发 MFC 应用程序那样容易。但 Visual Stduio 提供某种帮助使开发者迅速开发应用程序。可利用 ATL COM Winzard(活动模板库组件向导)和 ATL Object Wizard(活动模板库对象向导)开发 ATL 应用程序。

从目前介绍 ATL 技术的书籍来看，国内的书籍较少，本章主要通过介绍如何创建一个 ATL 应用程序来使大家对 ATL 技术有一定的了解，如果读者了解 ATL，可以略过此章。

2.1 COM 基础

本节主要对 COM 的两个基本概念（接口，组件）做简要介绍。

2.1.1 COM 接口

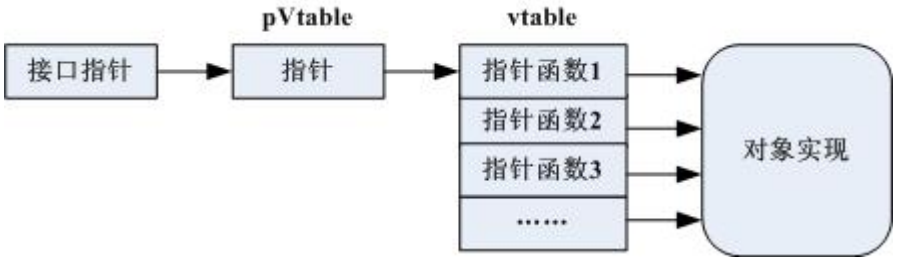


图 2.1 接口结构

从理论上讲，完整的 COM 编程系统是基于接口的。

从技术上讲，接口是包含了一组函数的数据结构，通过这组数据结构，客户代码可以调用组件对象的功能。接口定义了一组成员函数，这组成员函数是组件对象暴露出来的所有信息，客户程序利用这些函数获

得组件对象的服务。需要注意的是在接口成员函数中，字符串变量必须用 Unicode 字符指针，COM 规范要求使用 Unicode 字符，而且 COM 库中提供的 COM API 函数也使用 Unicode 字符。所以如果在组件程序内部使用到了 ANSI 字符的话，则应该进行两种字符表达的转换。当然，在即建立组件程序又建立客户程序的情况下，可以使用自己定义的参数类型，只要它们与 COM 所能识别的参数类型兼容。这里需要特别注意的是 COM 需要使用 Unicode 字符。

COM 接口可以分为以下两类：

- 标准接口
- 自定义接口

标准接口之 IUnknown，是所有接口的基接口。自定义接口也是基于 IUnknown 接口。 所有的 COM 组件都必须以这个接口为基础。IUnknown 提供了两个非常重要的特性，一个用于组件对象的生命周期管理，也可以查询被组件对象使用的其他接口。客户程序只能通过接口与 COM 对象进行通信，虽然客户程序可以不管对象内部的实现细节，但它要控制对象的存在与否。如果客户还要继续对对象进行操作，则它必须保证对象能一直存在于内存中；如果客户对象的操作已经完成，以后不再需要该对象了，则它必须及时地把该对象释放掉，以提高资源的利用率。IUnknown 引入了“引用计数”方法，可以有效地控制对象的生存期。另一方面，如果一个 COM 对象实现了多个接口，在初始时刻，客户程序不太可能得到该对象的所有接口指针，它只会拥有一个接口指针。如果客户程序需要其它的指针，则可利用 IUnknown 的“接口查询”方法来完成接口之间的跳转 。

IUnknown 的 IDL 定义：

interface IUnknown

```
{
    HRESULT QueryInterface([in] REFIID iid, [out] void **ppv);
    ULONG   AddRef(void);
    ULONG   Release(void);
}
```

IUnkown 的 C++ 定义：

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

标准接口之 IDispatch ，此接口用于脚本语言（如 Visual Basic）访问组件。当脚本语言调用组件对象的方法时，此接口查询函数的地址并执行。

标准接口之 IClassFactory（类厂）接口用于创建新的 COM 对象的实例。类厂(class factory)是 COM 对象的生产基地，COM 库通过类厂创建 COM 对象；对应每一个 COM 类，有一个类厂专门用于该 COM 类的对象创建操作。类厂本身也是一个 COM 对象，它支持一个特殊的接口 IClassFactory：

```
class IClassFactory : public IUnknown
{
  virtual HRESULT _stdcall CreateInstance(IUnknown *pUnknownOuter,
  const IID& iid, void **ppv) = 0;
```

```
    virtual HRESULT _stdcall LockServer(BOOL bLock) = 0;
}
```

如果你还没有学过 COM，那么你来创建一个组件的时候很可能会采用 C++的 new 操作符，这样的话会返回一个奇怪的错误。这也是一个非常常见的 ATL 错误，一个初学者不太理解的错误。实际上，创建一个对象，需要调用 CoCreateInstance 来创建一个 COM 对象的实例。在这里要注意的是创建 COM 对象不是用 new 操作符。在后面的章节中同样可以看到释放一个对象时，不能用 delete,而是要通过 Release 接口来释放对象。

自定义接口的目的是提供更多的功能给用户，开发者可以自己定义基于 IUnknown 的接口提供更多的功能。

COM 标识符 UUID/GUID 用来标识组件，通过唯一标识行 UUID 来唯一标识组件，如同身份证的意义，可以在系统中标识 COM 组件。在 COM 中，UUID 是指全局唯一标识符 GUID。GUID 分为 CLSID、IID 和 LIBID 三类。

COM 规范采用了 128 位全局唯一标识符 GUID 来标识对象和接口，这是一个随机数，并不需要专门机构进行分配和管理。因为 GUID 是个随机数，所以并不绝对保证唯一性，但发生标识符相重的可能性非常小。从理论上讲，如果一台机器每秒产生 10000000 个 GUID，则可以保证(概率意义上)的 3240 年不重复。

接口描述语言 IDL。COM 规范在采用 OSF（开放软件基金会） 的 DCE （分布式计算环境）规范描述远程调用接口 IDL (interface description language，接口描述语言)的基础上，进行扩展形成了 COM 接口的描述语言。接口描述语言提供了一种不依赖于任何语言的接口描述方法，因此，它可以成为组件程序和客户程序之间的共同语言。Microsoft Visual C++ 提供了 MIDL 工具，可以把 IDL 接口描述文件编译成

C/C++ 兼容的接口描述头文件(.h)。

按照 COM 规范，按照接口成员函数的参数类型的不同，有三种情况：

In 参数：对于 in 参数，由 OPC 客户程序分配和释放内存；

Out 参数：Out 参数由 OPC 服务器分配内存，由 OPC 客户程序释放内存，采用标准 COM 内存分配器。

In-out 参数：in-out 参数最初由 OPC 客户程序分配内存，然后由 OPC 服务器释放和再分配（如果需要）。和 out 参数一样，OPC 客户程序负责最后返回值的释放。

如不能正确释放内存，将会引起难以发现的内存泄漏（memory leak），造成系统可用的内存资源越来越少直至系统崩溃。因此，编写程序时可以参考 IDL 文件来查找 out 参数，并且针对每种类型的结构编写一段子程序处理内存释放。在每次客户程序调用服务器函数的过程中，不管函数执行正确与否，服务器程序都必须为每个 out 参数定义好返回值，而客户程序则负责释放相应的内存资源。

COM 接口所有方法返回类型都为 HRESULT。

2.1.2 组件

一般而言，组件具有三种类型：进程内组件，进程外组件，远程组件 。

进程内组件是采用动态连接库方式实现的组件。客户程序调用组件时，客户程序会把组件程序装入自己的进程空间，即客户程序和组件程序在同一个进程地址空间内。在客户端和服务端组件间有大量数据转移操作的情况下是理想的。进程内服务器会更快地装载。由于它占用和客

户端应用程序同样的地址空间，它可以与客户端更快的通信。进程内服务器是通过将组件作为动态连接库(DLL)的形式来实现的。DLL 允许特定的一套功能以分离于可执行的、以 DLL 为扩展名的文件进行存储。只有当程序需要它们时，DLL 才将其装入内存中，客户程序将组件程序加载到自己的进程地址空间后再调用组件程序的函数。

本地(即进程外)组件，进程外组件是以 EXE 方式实现的组件，具有独立的进程，因此客户程序和组件程序分别处在不同的进程空间地址中。在 COM 中，采用了本地过程调用（local procedure call,LPC）来进行本地通信。进程外服务器对需要运行于独立的处理空间或作为独立客户端应用程序的线程的组件是理想的。由于数据必须从一个地址空间移到另一个地址空间，因此这些服务器就会慢得多。由于进程外服务器是可执行的，它们运行在自己的线程内。当客户端代码正在执行时，客户端不锁住服务器。进程外服务器对需要表现为独立的应用程序的组件也是理想的。例如，Microsoft Internet Explorer 的应用程序是本地服务器的例子。客户端和服务端的通信是通过进程内的通信协议进行的，这个通信协议是 IPC（进程间通信）。

远程组件， 远程服务器与本地服务器类似，除了远程服务器是运行在通过网络连接的分离的计算机上。这种功能是使用 DCOM 实现的。DCOM 的优点在于它并不要求任何特别的编程来使具有功能。另外服务端和客户端通信是通过 RPC（Remote procedure call,RPC）通信协议进行的。对于这三种不同的服务器组件，客户程序和组件程序交互的内在方式是完全不同的。但是对于功能相同的进程内和进程外组件，从程序编写的角度看，客户程序是以同样的方法来使用组件程序的，客户程序不需要做任何的修改。

2.2 ATL 应用程序向导创建应用程序

应用 ATL 应用程序向导来开发 ATL 程序是快速而且有效的，本节主要以 MSDN 的例子为例来介绍如何应用 ATL 应用程序向导来创建新的 ATL 程序。

创建新的 ATL 应用程序第一步，生成一个新的 ATL 工程，从 Visual Studio 菜单中选择 File—>New，选中 Projects(工程)选项卡。在图 2.2 中选择 ATL COM AppWizard，输入工程名 BeepCtrlMod。

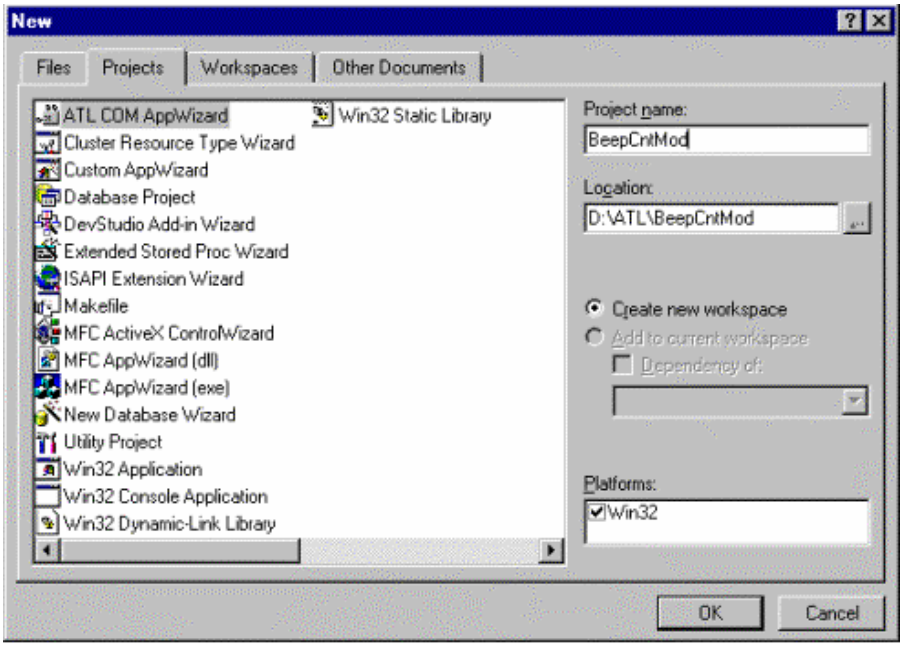


图 2.2Visual Studio 的 ATL COM AppWizard

单击 New 对话框的 OK，打开 ATL COM AppWizard 对话框。MFC AppWizard 由许多步骤组成，而 ATL COM Wizard 仅需要一步，即必须选定开发应用程序类型。由图 2.3 可以看出，一共有三种类型的组件，

动态连接库，可执行，服务三种类型，分别对应着进程内组件，进程外组件，服务。

选择动态连接库（进程内组件）。单击 Finish，打开 New Project Information 对话框，然后单击 OK 完成后，一个新的组件的框架已经建立。

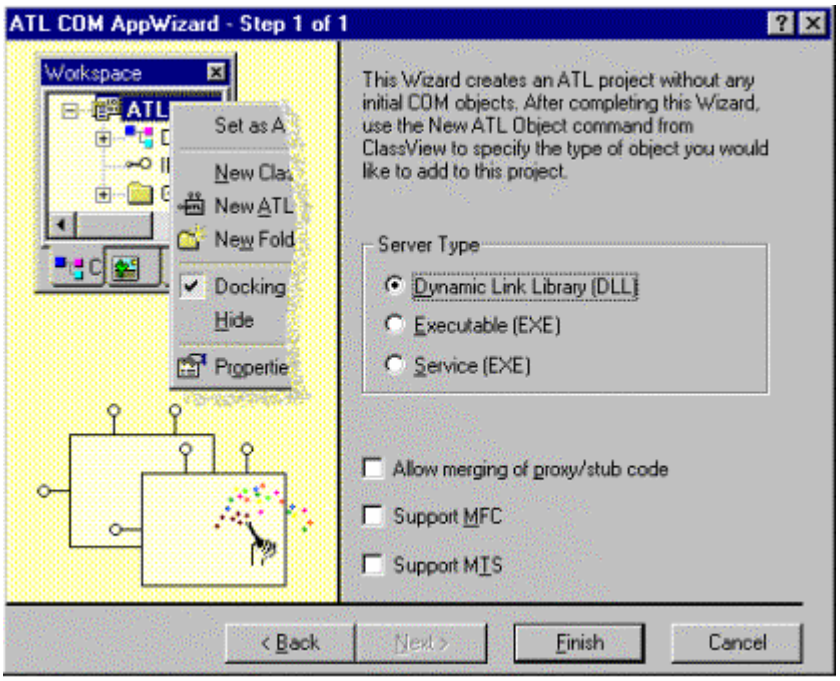


图 2.3 ATL COM AppWizard 选择组件类型

经过 AppWizard 创建得到的应用仅仅是一个程序框架，可以看到包含一个全局成员：

CComModule _Module;

•一个包含着最初的类型库的说明的.idl 文件

•一个.def 文件。

•一个.rc 资源文件

•一个包含着资源 ID 定义的头文件

•stdafx.h 和 stdafx.cpp 文件

•其它的.cpp 文件，用来实现全局功能应用的源文件。

可以看出与 MFC 应用程序不同的一点是多了一个后缀为.idl 的文件。每一个 ATL 工程都有一个与工程同名的 IDL 文件，此 IDL 文件记录了该工程中所用到 COM 接口或 COM 对象的定义。ATL AppWizard 或 ATL Object Wizard 会自动维护此 IDL 文件，也可以手动的修改此文件加入需要的 COM 接口的 IDL 定义。

2.3 源文件说明

本节对 BeepCntMod.cpp 文件做简要说明，从而了解主文件的程序结构。

```
#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "BeepCntMod.h"
#include "BeepCntMod_i.c"
CComModule _Module;
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

Initguid.h 文件是一个标准 OLE 系统文件，需要在工程中引用，用来定义 GUID（the globally unique identifier）结构。

如果现在编译和连接一下应用程序，会发现多了一个与工程的 IDL 文件同名的.h 头文件（BeepCntMod.h）和一个._i.c（BeepCntMod_i.c）

文件。这是集成环境调用 MIDL 编译器对工程的 IDL 文件进行编译生成的。在进行第一次编译之前，看不到这两个文件，必须编译以后才可以看这两个文件。

BeepCntMod.h 包含了接口和组件的定义。BeepCntMod_i.c 包含着 GUID 的定义。

_Module 是一个全局变量，定义了类厂以及类厂所有的初始化代码功能，如类厂的注册等，如果不用 ATL 进行开发，而是采用 MFC 开发，类厂的注册等都需要手动加入。可以通过 MSDN 来查看 **CComModule** 的详细文档。

最后一部分，是一个空的对象映射，对象向导会将需要的映射加入进去。每一个元素的对象映射都有一个 CLSID 和一个类名。**_Module** 对象读取这些映射根据 CLSID 来创建对象。

接下来看一下 DllMain 函数，它简单的调用了**_Module** 对象的 Init 和 Term 函数。

```
// DLL Entry Point
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
LPVOID /*lpReserved*/)
{
    if (dwReason == DLL_PROCESS_ATTACH)
   {
       _Module.Init(ObjectMap, hInstance,
          &LIBID_BEEPCNTMODLib);
       DisableThreadLibraryCalls(hInstance);
    }
     else if (dwReason == DLL_PROCESS_DETACH)
       _Module.Term();
 return TRUE;       // ok
}
```

最后的四个函数如下：

```
STDAPI DllCanUnloadNow(void)
{
    return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID*
ppv)
{
 return _Module.GetClassObject(rclsid, riid, ppv);
}
STDAPI DllRegisterServer(void)
{
   // registers object, typelib and all interfaces in typelib
   return _Module.RegisterServer(TRUE);
}
STDAPI DllUnregisterServer(void)
{
      return _Module.UnregisterServer(TRUE);
}
```

这些函数除非调用_Module 对象的合适的函数，什么也不做。有关这四个函数的详细说明请参考有关 COM 书籍和 MSDN。

2.4 添加组件对象

通过上面的介绍，开始对 ATL 有了一定的认识，下面为上面的应用程序添加一个 COM 组件，本节的主要目的是熟悉如何通过 ATL 添加组件对象。

最简单的添加组件的方法是通过 ATL Object Wizard。从 Insert 菜单下选择 New ATL Object...,你将看到图 2.4 所示的对话框。

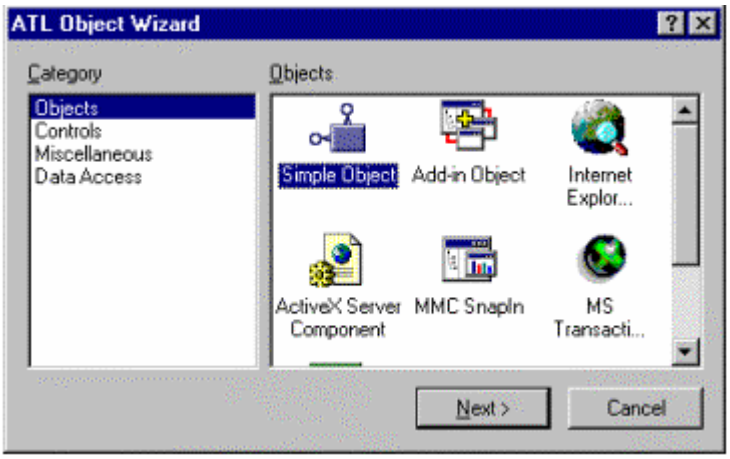


图 2.4 ATL Object Wizard

这里选择 Simple Object,单击 Next,图 2.5 所示的属性页显示出来，在这个属性页可以输入组件的名称和组件的属性。在 Short Name 编辑框里输入“BeepCnt”，也可以输入其它名字。输入“BeepCnt”其它七个编辑框的内容会自动生成。

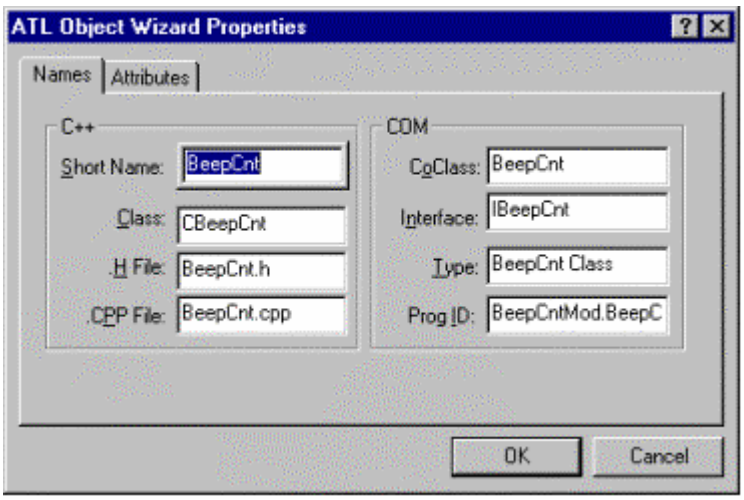


图 2.5 ATL Object Wizard Properties

单击属性页上的 Attributes,出现图 2.6 所示的属性框，在这个框中可

以选择组件相应的属性，本例中选择默认。单击 OK。

图 2.6 ATL Object Wizard Properties

Apartment Threading Model 的意思是指这个对象可以被一个或者多个线程所调用。实际上每个 COM 对象的实例都是单线程的，需要注意的是假设 COM 对象有多个线程在调用它，如果在程序的方法中引用了全局变量，必须保证程序多个线程访问的安全问题。不管如何，只要引用了全局变量，需要考虑程序多线程访问时的安全问题，或者在此选择 Single Threading Model.

在这里，选择 Dual Interface,双重接口的意思是指创建的组件支持自定义接口，也支持自动化接口。自动化接口是用来支持脚本语言访问的接口，考虑到客户程序自动化的实现比自定义的实现简单的多，所以选择了双接口的方式。

Aggregation，汉语意思为聚合。包容和聚合是 COM 的两种重要类型。聚合是指组件对象一种特殊的包容方法，在一个组件对象里面可以拥有另外的组件对象。一般而言，采用 COM 自己实现聚合的的方式是

复杂的，而 ATL 可以帮助我们操作聚合的实现。如果组件支持聚合，那么组件将是更加灵活的，因此我们选择支持聚合。

ISupportErrorInfo，创建 ISupportErrorInfo 接口支持，以便对象可将错误信息返回到客户端。我们不选择它。

Connection Points，连接点用来支持事件，通过使对象的类从 IConnectionPointContainerImpl 导出来启用对象的连接点，连接点用来支持事件，所谓事件是指 COM 对象中当某个属性发生改变时，对象产生一个事件，通知到客户程序，客户程序可以处理这些事件。本例中，不选择连接点。

Free-Threaded Marshaller，自由线程封送拆收器 ，创建自由线程封送拆收器对象，以有效地在同一进程中的两个线程之间封送接口指针。对指定“两者”或“自由”作为线程模型的对象可用。用于多线程控制的特定类型。本例中，不选择它。

创建完成后的组件对象由于支持双接口（自动化接口，自定义接口），聚合，多线程访问，因此它是一个简单但是易于扩展的组件。

现在工程中又多了两个 ATL 对象向导创建的文件：beepcnt.cpp 和 beepcnt.h。beepcnt.cpp 基本上是空的，因为没有为组件对象添加任何的属性。所有将来添加的对象方法和属性都将在这个文件中实现。

beepcnt.h 包含了对象的类的定义。

```
class ATL_NO_VTABLE CBeepCnt :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CBeepCnt, &CLSID_BeepCnt>,
public IDispatchImpl<IBeepCnt, &IID_IBeepCnt,
 &LIBID_BEEPCNTMODLib>
{
public:
    CBeepCnt()
    {
```

```
    }
DECLARE_REGISTRY_RESOURCEID(IDR_BEEPCNT)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CBeepCnt)
    COM_INTERFACE_ENTRY(IBeepCnt)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
// IBeepCnt
public:
};
```

可以看出，CBeepCnt 类基于三个模板类。CComObjectRootEx 操作了组件的引用计数；CComCoClass 用来定义该对象的默认类工厂和聚合模型；IDispatchImpl，提供了一个双重接口 IBeepCnt（接口 ID 是 IID_IbeepCnt，支持自动化接口和自定义接口）。

CBeepCnt 类同样提供了一组宏，ATL_NO_VTABLE 会告诉编译器不要再为 CBeepCnt 类生成虚函数，因为 ATL_NO_VTABLE 必须只能与一个无法直接创建的基类一起使用。在项目中一定不能将 declspec(novtable) 与基本上是导出的类一起使用，因为该类（通常是 CComObject、CComAggObject 或 CComPolyObject）为项目初始化 vtable 指针。一定不能从任何使用 declspec(novtable) 的对象的构造函数调用虚函数。应该将那些调用移动到 FinalConstruct（大家要记着这个方法） 方法。需要我们切记的是，我们可以尝试着添加一个虚函数，但是编译器会编译错误，切记 ATL_NO_VTABLE 意味着没有虚函数，不要为基于 ATL_NO_VTABLE 的类添加虚函数。

当对象创建时，对象实际上是 CComObject<CBeepCnt>。换言之，用的是把对象类型作为一个参数的 CComObject 模板类。一个派生的图如下图所示。

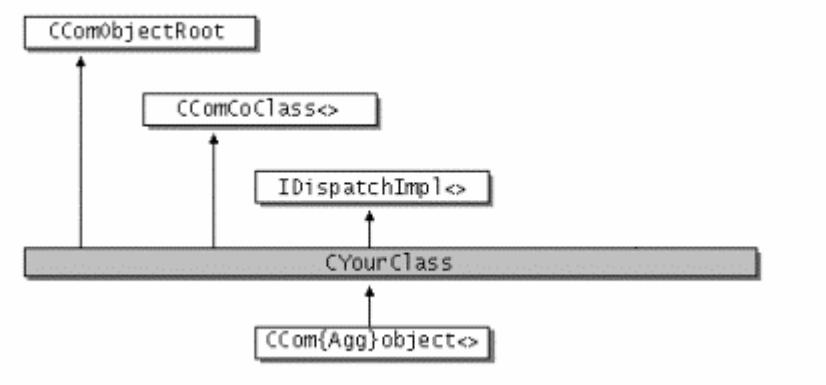


图 2.7 CcomObject 派生图

CComObject 的主要工作是提供 IUnknown 方法的实现。这些必须由最低层的派生类提供，以便所有由 IUnknown 派生的类可以共享 IUnknown 方法的实现。CComObject 的方法仅仅调用 CComObjectRootEx 中的实现。

另外需要注意的是 COM 接口映射，COM 接口映射包含着我们实现自定义接口和 **IDispatch** 的宏。COM 接口映射中包含的接口是可以通过 QueryInterface 返回的接口指针的接口。

这里还有一对宏。一个用来根据资源 ID 定义注册入口，一个用来改变对象创建的方法来保证对象不会被意外的删除。

与非采用 ATL 方式的 COM 相比，可以看到采用 ATL 开发的组件中没有引用计数的代码，是因为引用计数的工作由 CComObjectRootEx 类操作，而不需要我们来操作了。

接着讨论的新文件，beepcnt.rgs,包含了为 ATL 处理的代码的源脚本。大部分直接反映了为了 COM 运行时能够找到组件的注册入口。文件如下：

```
HKCR
{
    BeepCntMod.BeepCnt.1 = s 'BeepCnt Class'
```

```
        {
            CLSID = s '{AE73F2F8-4E95-11D2-A2E1-00C04F8EE2AF}'
        }
        BeepCntMod.BeepCnt = s 'BeepCnt Class'
        {
            CLSID = s '{AE73F2F8-4E95-11D2-A2E1-00C04F8EE2AF}'
            CurVer = s 'BeepCntMod.BeepCnt.1'
        }
        NoRemove CLSID
        {
          ForceRemove {AE73F2F8-4E95-11D2-A2E1-00C04F8EE2AF} =
          s 'BeepCnt Class'
          {
            ProgID = s 'BeepCntMod.BeepCnt.1'
            VersionIndependentProgID = s 'BeepCntMod.BeepCnt'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
           'TypeLib' = s '{170BBD8D-4DE8-11D2-A2E0-00C04F8EE2AF}'
          }
        }
    }
```

这段脚本用来注册和注销组件。默认注册时，所有的键都会添加到注册表，不管这些键是否在注册表中。当注销时，默认的方式是删除所有在脚本中列出的键。一般情况下，你不需要编辑这个文件，当你添加对象和接口时，向导会自动的实现它。

最后讨论的文件是“BeepCntMod_i.c”，在此文件中，定义了组件的 CLSID。CLSID 是唯一地标识类或组件的 GUID，传统地，CLSID 的一般形式为 CLSID_<unique identifier>。TypeLibID 是标识系统上的类型库。按照惯例，TypeLibID 的一般形式为 LIBID_<组件工程名>Lib。

```
#ifdef __cplusplus
```

```
extern "C"{
#endif
#ifndef __IID_DEFINED__
#define __IID_DEFINED__
typedef struct _IID
{
    unsigned long x;
    unsigned short s1;
    unsigned short s2;
    unsigned char    c[8];
} IID;
#endif // __IID_DEFINED__
#ifndef CLSID_DEFINED
#define CLSID_DEFINED
typedef IID CLSID;
#endif // CLSID_DEFINED
const IID LIBID_BEEPCNTMODLib =
{0x170BBD8D,0x4DE8,0x11D2,{0xA2,0xE0,0x00,0xC0,0x4F,0x8E,0x
E2,0xAF}};
#ifdef __cplusplus
}
#endif
```

与添加组件对象前相比，已经存在的文件也有一些小的变化。

•BeepCntMod.cpp 现在添加了#includes BeepCnt.h，并且为 CbeepCnt 添加了一个对象映射入口，每一个 COM 对象都有一个对象映射入口：

```
BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_BeepCnt, CBeepCnt)
END_OBJECT_MAP()
```

•.idl 文件里现在有了 IDispatch 派生的 IBeepCnt 接口入口，一个为 BeepCnt 类的在类型库的 Coclass 入口。

当重新编译程序时，MIDL 生成了一个新的反映新建组件对象的 BeepCntMod.h 文件。编译后就生成了组件的 DLL 文件，在编译的同时

完成组件的注册。

2.5 添加组件对象的属性和方法（函数）

上面已经介绍了如何用 ATL 组件。然而生成的对象是个空的对象，属于这个对象的属性和方法都是空的。如何添加组件对象的属性和方法是本节要讨论的内容。

从 Class View 中右击 IbeepCnt，选择“Add Method…” 如图 2 所示的对话框出现。在 Method Name 编辑框中，选择返回值类型为 HRESULT，方法名输入 Beep，参数一项为空，单击 OK，完成 Beep 方法的添加，向导会把所有 Beep 的定义的代码自动添加到 BeepCnt.cpp 和 BeepCnt.h 文件中。

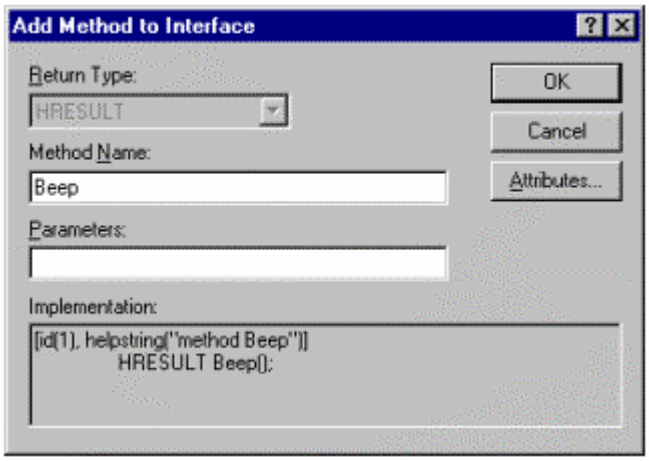


图 2.8 向接口添加方法

添加一个属性也是类似的。右击接口，选择“Add Property…”，如图 2.9 所示的对话框出现。选择合适的类型，输入属性的名称，单击 OK 完成添加。

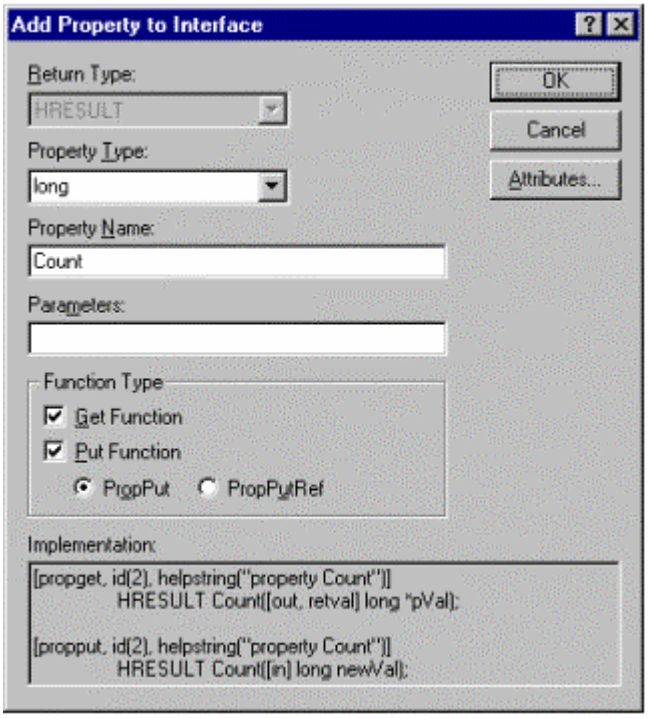


图 2.9 向接口添加属性

完成了方法和属性的添加后，向导向.idl 文件添加了方法和属性的定义。

```
interface IBeepCnt : IDispatch
{
    [id(1), helpstring("method Beep")] HRESULT Beep();
    [propget, id(0), helpstring("property Count")]
    HRESULT Count([out, retval] long *pVal);
    [propput, id(0), helpstring("property Count")]
    HRESULT Count([in] long newVal);
};
```

BeepCnt.h 包含了三个新函数的定义，BeepCnt.cpp 文件包含了这些函数的框架。

```
STDMETHODIMP CBeepCnt::Beep()
{
```

```
    // TODO: Add your implementation code here
    return S_OK;
}
STDMETHODIMP CBeepCnt::get_Count(long *pVal)
{
    // TODO: Add your implementation code here
    return S_OK;
}
STDMETHODIMP CBeepCnt::put_Count(long newVal)
{
    // TODO: Add your implementation code here
    return S_OK;
}
```

为了让创建的组件可以做一些我们能感觉到的事情，需要添加一些代码，首先，为 CBeepCnt 类添加一个计数器，并且在 CBeepCnt()构造函数中把它初始化为 1。

```
long cBeeps;
CBeepCnt() : CBeeps(1) { }
```

接着编写一部分代码在 CBeepCnt 的方法里。注意属性有两个功能，一个是设置计数器（put_Count），一个是获得计数器（get_Count）。

```
STDMETHODIMP CBeepCnt::Beep()
{
    for (int i = 0; i < cBeeps; i++)
    {
        MessageBeep((UINT) -1);
        Sleep(1000);
    }
    return S_OK;
}
STDMETHODIMP CBeepCnt::get_Count(long *pVal)
{
    *pVal = cBeeps;
    return S_OK;
}
STDMETHODIMP CBeepCnt::put_Count(long newVal)
```

```
{
    cBeeps = newVal;
    return S_OK;
}
```

现在一个简单功能的组件诞生了，它可以用来发出嘟嘟声。

2.6 测试组件

如何来测试这个组件呢？一般而言，采用 Visual Basic 是个不错的选择，可以快速的编写客户程序来实现测试。可以编写一个图 2.10 所示的对话框程序来测试它。

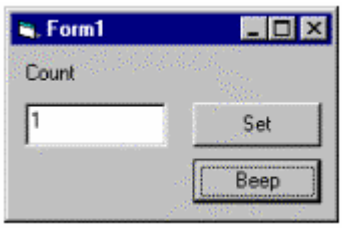


图 2.10 测试界面

采用 Visual Basic 6.0 生成的代码如下：

```
Dim BeeperCnt As BeepCnt
Private Sub Beep_Click()
    Text1 = BeeperCnt
    BeeperCnt.Beep
End Sub
Private Sub Set_Click()
    BeeperCnt = Val(Text1)
    Text1 = BeeperCnt
End Sub
Private Sub Form_Load()
    Set BeeperCnt = New BeepCnt
    Text1 = BeeperCnt
End Sub
```

需要注意的是需要引用 BeepCntMod 1.0 Type Library。

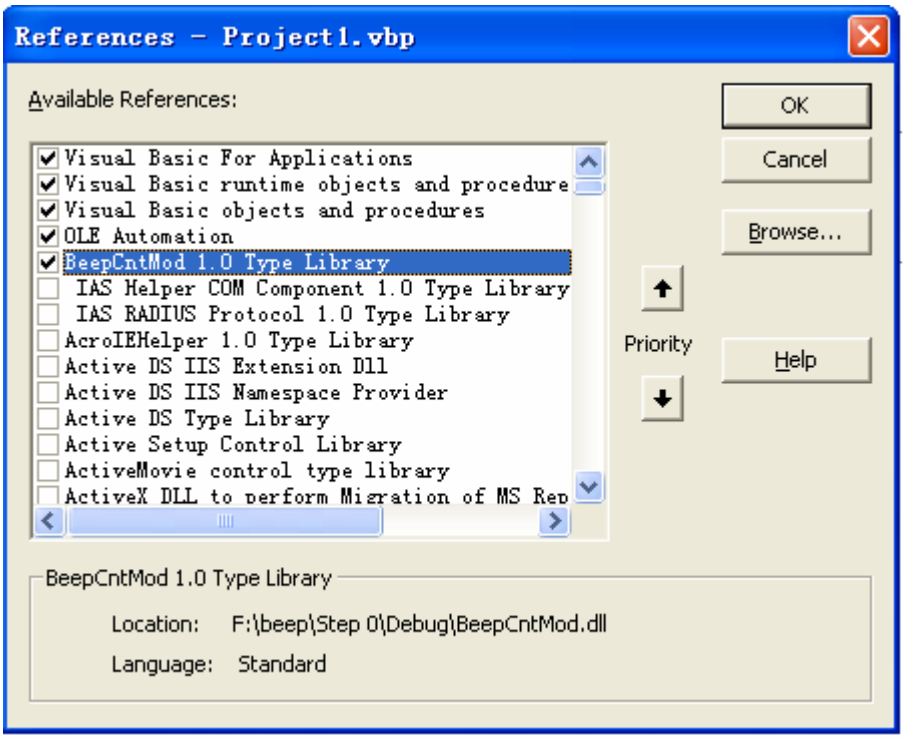


图 2.11 引用库

运行这个程序，是不是听到了嘟嘟声，而且可以设置嘟嘟声的次数，类似的，你可以添加更多的属性和方法，来增强这个组件的功能，是不是很简单呢。

重点：何为 ATL？何为类厂？COM 对象如何创建？如何测试 COM 组件功能？

第 3 章 ATL 开发 OPC 服务器

****

关键字：包容 聚合 回调 结构 服务器对象 组对象

前面已经介绍了 OPC 服务器的结构，接口以及如何利用 ATL 开发组件。本章主要以前面的知识作为基础，来讨论如何实现 OPC 服务器的开发。

OPC 服务器有两个对象 Server 和 Group，从两个对象的关系来看，Server 包容着所有 Group，也就是说客户程序创建 Server 对象后，通过调用 Server 的接口来创建 Group 对象。对于一个实际的 OPC 服务器而言，一个成功 OPC 服务器不只是两个对象的问题，还涉及到各个接口之间的协调，Server 对象对 Group 对象的管理，Group 对象对 Item 的管理等。尤其是在异步通讯中，回调函数的实现。

在第一章中简要介绍了 Server 对象和 Group 对象及两个对象的接口。

OPC 服务器的开发必须以 OPC 规范为基础，实现各个对象及其接口。本章从实用化的角度及工程化的角度，设计服务器实现的功能如下：

- 进程外服务器。
- 支持异步通讯。
- 每个客户可以最多建立 10 个 Group 对象。
- 支持 Group 对象的复制，删除，可以设置 Group 的名称，状态。
- 支持 Item 的添加，删除。
- 支持浏览服务器的地址空间。
- 通过 DCOM，可以远程访问。

本章首先介绍 Server 对象和 Group 对象以及两个对象的主要接口和方法，然后主要介绍用 ATL 如何实现 OPC 服务器的功能。

3.1 OPC Server 对象定义

OPC 规范定义 OPC Server 对象接口如下：

OPCServer
- IOPCServer
- IOPCItemProperties (new 2.0)
- IConnectionPointContainer (new 2.0)
- IOPCCommon (new 2.0)

IOPCCommon ：

```
HRESULT     SetLocaleID ( dwLcid )
HRESULT     GetLocaleID ( pdwLcid )
HRESULT     QueryAvailableLocaleIDs ( pdwCount, pdwLcid )
HRESULT     GetErrorString ( dwError, ppString)
HRESULT     SetClientName (szName)
```

此接口被使用于各种 OPC 规范服务器（OPCDA,OPC ALARM,OPC HDA 等）。它可以用来建设或查询服务器的区域位置（LocaleID），SetLocaleID 用来设置服务器的区域位置，GetLocaleID 用来获得服务器的区域位置，QueryAvailableLocaleIDs 用来查询可用的服务器的区域位置，GetErrorString 用来返回错误的信息，SetClientName 注册客户端名称。

IOPCServer

```
HRESULT     AddGroup(szName, bActive,
                     dwRequestedUpdateRate, hClientGroup, pTimeBias,
                     pPercentDeadband, dwLCID, phServerGroup,
                     pRevisedUpdateRate, riid, ppUnk)
HRESULT     GetErrorString(dwError, dwLocale, ppString)
HRESULT     GetGroupByName(szName, riid, ppUnk)
HRESULT     GetStatus(ppServerStatus)
HRESULT     RemoveGroup(hServerGroup, bForce)
HRESULT     CreateGroupEnumerator(dwScope, riid, ppUnk)
```

对于此接口的方法，AddGroup 用来对 Server 对象添加组对象；GetErrorString 用来返回错误的信息；GetGroupByName 通过 Group 名称来查询需要的接口；GetStatus 获得 Server 的状态；RemoveGroup 删除 Group 对象；CreateGroupEnumerator 用来创建 OPCGroup 枚举器。下面对 AddGroup 和 RemoveGroup 方法进行详述。其它方法略。

```
IOPCServer::AddGroup
HRESULT AddGroup(
    [in, string] LPCWSTR szName,
    [in] BOOL bActive,
    [in] DWORD dwRequestedUpdateRate,
    [in] OPCHANDLE hClientGroup,
    [unique, in] LONG *pTimeBias,
    [in] FLOAT * pPercentDeadband,
    [in] DWORD dwLCID,
    [out] OPCHANDLE * phServerGroup,
    [out] DWORD *pRevisedUpdateRate,
    [in] REFIID riid,
    [out, iid_is(riid)] LPUNKNOWN * ppUnk
);
```

功能:向一个 Server 添加一个 Group 对象。

参数	描述
SzName	组名。组名必须为 UNICODE 码，不能是 ASC 码，而且不能和同一个客户程序创建的其它组的名字相同。如果没有定义组名，服务器会给它定义一个独有的名字。
bActive	如果值为 FALSE，那么组被定义为不活动的。如果为 TRUE，组被定义为活动的。
dwRequestedUpdateRate	客户程序定义的组最快的刷新速率，单位为毫秒，主要在 OnDataChage 中使用。同样说明了要求 CACHED DATA 的期望速率。如果此值为 0，那么服务器必须以最快的实际速率刷新。
HClientGroup	客户程序提供的组句柄。

PTimeBias	时间戳指针。如果指针为空表示你希望使用系统默认时间戳。
pPercentDeadband	定义了死区参数。这个参数仅仅对本组内的模拟量有效。当模拟量的值变化超出死区，必须产生一个回调给客户程序。NULL 值代表 0.0，即没有死区。
DwLCID	本组字符串操作时，服务器使用的语言。
phServerGroup	服务器为新的组产生的句柄。客户程序将采用服务器提供的句柄来要求服务器对组进行其它的操作。
pRevisedUpdateRate	服务器返回的实际刷新速率，这个值可能与客户程序要求的刷新速率不一样。注意：这个值可能也比服务器实际从设备获取数据和刷新 CACHE 的速率要慢。
Riid	期望接口类型 (如 IID_IOPCItemMgt)
PpUnk	返回接口指针，如果操作失败，返回 NULL。

返回码

返回码	**描述**
S_OK	操作成功。
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存空间。
E_INVALIDARG	功能非法。
OPC_E_DUPLICATENAME	Duplicate name not allowed.
OPC_S_UNSUPPORTEDRATE	服务器不支持指定的刷新速率，服务器返回实际支持的刷新速率(pRevisedUpdateRate)。
E_NOINTERFACE	服务器不支持请求的接口。

说明：

一个 Group 是为客户组织操作数据项的逻辑容器。

服务器创建一个组对象，并且返回一个客户要求的接口指针。如果客户要求一个任选的接口，而服务器并不支持这个接口，服务器会返回一个接口不支持的错误码。

被请求的刷新速率和修正后的刷新速率必须确定在服务器和客户的对话间。客户创建的组对象，按照修正后的刷新速率刷新，而不依赖于项的多少，项的添加，删除也没有影响。

注释：

对象的生存周期按以下定义。即使所有的接口都被释放，组对象仍然不会被删除，直到 RemoveGroup 被调用。在所有的界扩没有被释放前，客户不应该调用 RemoveGroup 来删除所有的私有组。

由于服务器被认为组的容器，因此服务器可以在服务器的接口被释放时强制删除所有存在的组对象。

DwLCID 完全由服务器指定。服务器可以忽略掉这个参数，如果不支持动态的语言定义。

IOPCServer::RemoveGroup

```
HRESULT RemoveGroup(
    [in] OPCHANDLE hServerGroup,
    [in] BOOL bForce
);
```

功能：删除组对象

参数	描述
HserverGroup	被删除组的句柄。
Bforce	强制删除标志，即使组对象仍然在引用中。

返回码

返回码	**描述**
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存。
E_INVALIDARG	功能非法。
S_OK	操作成功。
OPC_S_INUSE	组没有被删除，因为引用存在。组将被标记为删除，当组的所有引用释放后，服务器会自动删除组。

注释：

如前所述，一个组不会被删除，当所有的客户接口被释放后。客户程序可以调用 GetGroupByName 在所有的接口被释放后。RemoveGroup() 可以使服务器释放对组最后的引用，从而真正的删除组。

一般而言，一个好的客户程序只在释放所有接口后才调用此功能。如果接口还存在，Remove group 将把组标记为删除。任何这个组的接口调用将会返回 E_FAIL。当所有的接口被释放后，组将会被自动删除。如果 bForce 的值为 TRUE,那么组将被无条件的删除，即使引用仍然存在，那些接口的使用将会导致一个非法存取。这个功能不能被公用的组调用。

3.2 OPC Group 对象定义

OPC 规范定义 OPC Group 对象接口如下：

IOPCGroupStateMgt

HRESULT	GetState(pUpdateRate, pActive, ppName, pTimeBias, pPercentDeadband, pLCID, phClientGroup, phServerGroup)
HRESULT	SetState(pRequestedUpdateRate, pRevisedUpdateRate, pActive, pTimeBias, pPercentDeadband, pLCID, phClientGroup)
HRESULT	SetName(szName);
HRESULT	CloneGroup(szName, riid, ppUnk);

IOPCPublicGroupStateMgt (optional)

HRESULT	GetState(pPublic);
HRESULT	MoveToPublic(void);

IOPCSyncIO

HRESULT	Read(dwSource, dwCount, phServer, ppItemValues, ppErrors)
HRESULT	Write(dwCount, phServer, pItemValues, ppErrors)

IOPCAsyncIO2

HRESULT Read(dwCount, phServer, dwTransactionID,
, ppErrors,)
HRESULT Write(dwCount, phServer, pItemValues,
dwTransactionID, pdwCancelID, ppErrors);
HRESULT Cancel2 (dwCancelID);
HRESULT Refresh2(dwSource, dwTransactionID, pdwCancelID);
HRESULT SetEnable(bEnable);
HRESULT GetEnable(pbEnable);

IOPCItemMgt

HRESULT AddItems(dwCount, pItemArray, ppAddResults,
ppErrors)
HRESULT ValidateItems(dwCount, pItemArray,
bBlobUpdate, ppValidationResults, ppErrors)
HRESULT RemoveItems(dwCount, phServer, ppErrors)
HRESULT SetActiveState(dwCount, phServer, bActive,
ppErrors)
HRESULT SetClientHandles(dwCount, phServer, phClient,
ppErrors)
HRESULT SetDatatypes(dwCount, phServer,
pRequestedDatatypes, ppErrors)
HRESULT CreateEnumerator(riid, ppUnk)

IOPCItemMgt::AddItems

```
HRESULT AddItems(
    [in] DWORD dwCount,
    [in, size_is(dwCount)] OPCITEMDEF * pItemArray,
    [out, size_is(,dwCount)] OPCITEMRESULT ** ppAddResults,
    [out, size_is(,dwCount)] HRESULT ** ppErrors
    );
```

功能：添加一个或多个项到一个组

参数	**描述**
DwCount	要添加的项的数目。
PitemArray	OPCITEMDEF 数组，包含着项的存取路径, 定义和被请求的数据类型。

PpAddResults	OPCITEMRESULT 数组，服务器用来告诉客户关于项的附加的信息（项句柄和规范的数据类型）
PpErrors	HRESULT 数组，服务器用来告诉客户每个项是否被成功添加，如果添加失败，服务器提供一个原因。

HRESULT 返回码

Return Code	**Description**
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存。
E_INVALIDARG	非法参数。 (例如 dwCount=0).
S_OK	操作成功。
S_FALSE	操作完成，但是有一个或者多个错误。
OPC_E_PUBLIC	不能向一个公共组添加项。

ppErrors 返回码

Return Code	**Description**
S_OK	对这个项的操作成功。
OPC_E_INVALIDITEMID	ItemID 不合法。
OPC_E_UNKNOWNITEMID	ItemID 不在服务器的地址空间。
OPC_E_BADTYPE	被请求的数据类型不能为这个项返回。
E_FAIL	操作不成功。
OPC_E_UNKNOWNPATH	项的存取路径不存在。

注释：

允许向组添加同样的项。也就是说一个项可以被多次添加。

所有失败的 ppErrors 码表示相应的项没有被添加到组，相应的 OPCITEMRESULT 没有有用的信息。

服务器提供的项句柄在这个组内是唯一的。但在所有组内不一定是唯一的，服务器允许重新使用被删除项的句柄。

项不可以被添加到公共组。

客户需要释放所有的 OPCITEMRESULT 内存。

注意，如果 Advise 是活动的，客户将要接收活动项的回调。这可能非常快，可能比客户来处理那些返回的结果更快，客户必须要处理好这些问题，一个解决的方式是在调用 AddItems 之前置组的状态为非活动态，在调用完成后再恢复组的状态为活动态。

IOPCItemMgt::RemoveItems

```
HRESULT RemoveItems(
    [in] DWORD dwCount,
    [in, size_is(dwCount)] OPCHANDLE * phServer,
    [out, size_is(,dwCount)] HRESULT ** ppErrors
    );
```

功能：从一个组删除项，与 AddItems 相对应。

参数	**描述**
dwCount	被删除的项的数目。
phServer	项句柄的数组。
ppErrors	HRESULT 数组，说明项是不是被成功的删除。

HRESULT 返回码

返回码	**描述**
S_OK	操作成功。
S_FALSE	表明有一个或者多个失败。
E_FAIL	操作失败。
E_INVALIDARG	非法参数。 (例如 dwCount=0).
OPC_E_PUBLIC	不能从一个公共组删除项。

ppErrors 返回码

Return Code	**Description**
S_OK	相应的项被删除。
OPC_E_INVALIDHANDLE	相应的项句柄非法。

注释：

从组中添加或删除项并不影响服务器的地址空间或者实际的物理设备。仅仅表明客户对那些项感兴趣。

项不是实际的对象，在自定义接口中，项没有自己的接口，项也没有引用计数的概念。

3.3 用于客户端的回调定义

```
IOPCDataCallback
HRESULT     OnReadComplete(dwTransid, hGroup,
            hrMasterquality, hrMastererror, dwCount,
            phClientItems, pvValues,
            pwQualities,   pftTimeStamps, pErrors,);
HRESULT     OnWriteComplete(dwTransid,  hGroup,  hrMastererr,
            dwCount, phClientItems,pErrors);
HRESULT     OnCancelComplete(dwTransid, hGroup);
HRESULT     OnDataChange(dwTransid, hGroup, hrMasterquality,
            hrMastererror, dwCount,   phClientItems, pvValues,
            pwQualities,   pftTimeStamps, pErrors,);
IOPCDataCallback::OnDataChange
HRESULT OnDataChange(
    [in] DWORD    dwTransid,
    [in] OPCHANDLE hGroup,
    [in] HRESULT    hrMasterquality,
    [in] HRESULT    hrMastererror,
    [in] DWORD    dwCount,
    [in, sizeis(dwCount)] OPCHANDLE * phClientItems,
    [in, sizeis(dwCount)] VARIANT * pvValues,
    [in, sizeis(dwCount)] WORD    * pwQualities,
    [in, sizeis(dwCount)] FILETIME    * pftTimeStamps,
    [in, sizeis(dwCount)] HRESULT *pErrors
    );
```

功能:这个方法由客户提供用来操作组的数据变化和刷新。

参数	描述
dwTransid	事务标识符，如果是一般的回调，此值为 0。
hGroup	组的客户句柄。
hrMasterquality	S_OK ，如果 OPC_QUALITY_MASK 是 OPC_QUALITY_GOOD,否则为 S_FALSE 。
hrMastererror	如果无错误，返回 S_OK，否则返回 S_FALSE.。
dwCount	读取的在客户句柄表里的项数目。
phClientItems	数据发生变化的项的客户句柄表。.
pvValues	数据发生变化的项的 VARIANTS 类型数据表。
pwQualities	读取的项的品质值的表。
pftTimeStamps	读取的项的时间戳表。
pErrors	项的 HRESULTS 表。如果数据项的品质变为 UNCERTAIN 或者 BAD，这里返回附加的服务器提供的更有用的错误信息。

HRESULT 返回码

返回码	描述
S_OK	客户总是返回 S_OK。

ppErrors 返回码

返回码	描述
S_OK	项的数据的品质是好的（OPC_QUALITY_GOOD）。
E_FAIL	项的操作失败。
OPC_E_BADRIGHTS	操作的项是不可读的。
OPC_E_UNKNOWNITEMID	项在服务器的地址空间是不可用的。
S_xxx, E_xxx	S_xxx – 卖方特殊信息。 E_xxx – 卖方指定的特殊错误。

注释：

对于所有的 S_xxx 错误码，客户需要假设相应的值，品质和时间戳都已经定义好了，不管品质是 UNCERTAIN 或者 BAD，建议服务器卖主提供关于 UNCRTAIN 或 BAD 项的附加的信息。

回调发生在以下情况：

一个或者多个的数据变化事件。事件在活动 Group 中的活动项的值或者品质发生变化时发生。回调不会以超过刷新速率的速度发生。一般而言，除非值或者品质发生变化，否则回调不会发生。transaction ID 为 0。

通过 AsyncIO2 接口的刷新请求。一旦请求发生，在活动 Group 中的活动项都会刷新。Transaction ID 不为 0。

3.4 OPC 服务器的设计及初步实现

根据工程化，实用化的角度，为了使读者尽快熟悉 OPC 服务器的体系和结构。本书中对于 OPC 服务器的实现做了一定的简化，主要实现以下接口及功能。

Server：

IOPCServer

IOPCBrowseServerAddressSpace

Group:

IOPCItemMgt

IOPCGroupStateMgt

IOPCASyncIO2

IconnectionPointContainer

图 3.1 是 OPC 服务器的整体结构图。

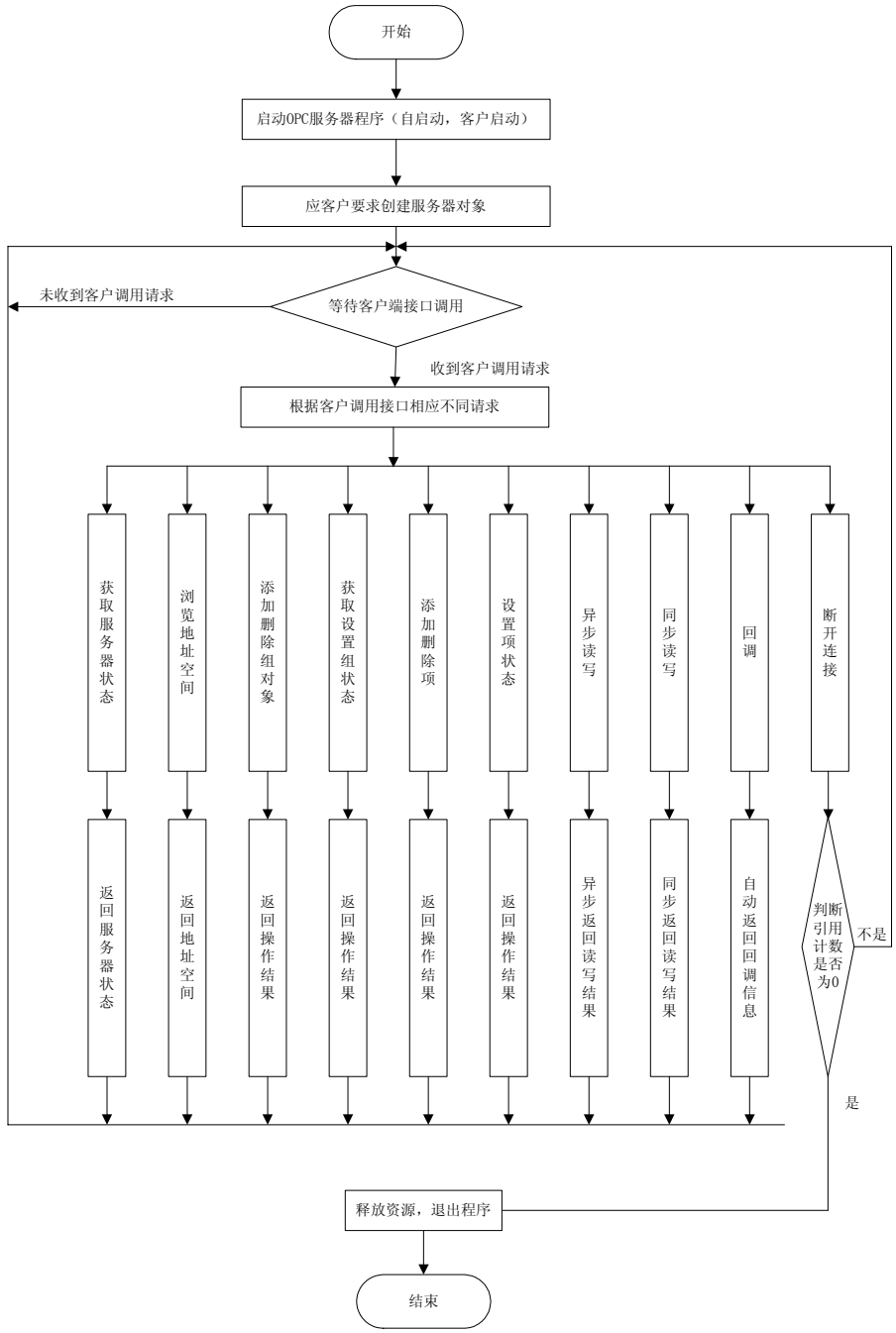


图 3.1OPC 服务器的整体结构图

这样设计 OPC 服务器主要实现本章开始时定义的功能。

OPC 访问的实际流程：

1．OPC 服务器首先要建立类厂，通过类厂，客户程序可以创建 OPC 服务器。

2．客户程序通过 OPC 服务器的名字找到 OPC 服务器在注册表中的信息。然后调用 CoCreateInstance()在服务器中创建一个 OPCServer，这个函数主要完成两个工作：创建类厂，创建 OPCServer。

3．通过访问 OPCServer 的接口来实现一系列的功能，如创建 GROUP，获得 SERVER STATUS，浏览地址等等。

4．创建 GROUP 对象后，通过 GROUP 的接口来实现一系列的功能，如添加 ITEM，删除 ITEM 等。

OPC 的数据传输：

OPC 的数据传输是 OPC 最终要实现的功能，也是 OPC 的一个主要内容。从 2.0 规范看，支持同步，异步，为简化，本章的例子中只实现了异步读。

3.5 OPC 服务器的编程实现

OPCDA.IDL 的使用：

首先需要将 OPCDA.IDL 引用，以此来生成 OPCDA.h 和 OPCDA_i.c 文件。

假设 OPCDA.IDL 在 D:根目录下, 步骤如下：

进入 DOS 模式：

进入 D:目录：

键入命令 MIDL OPCDA.IDL

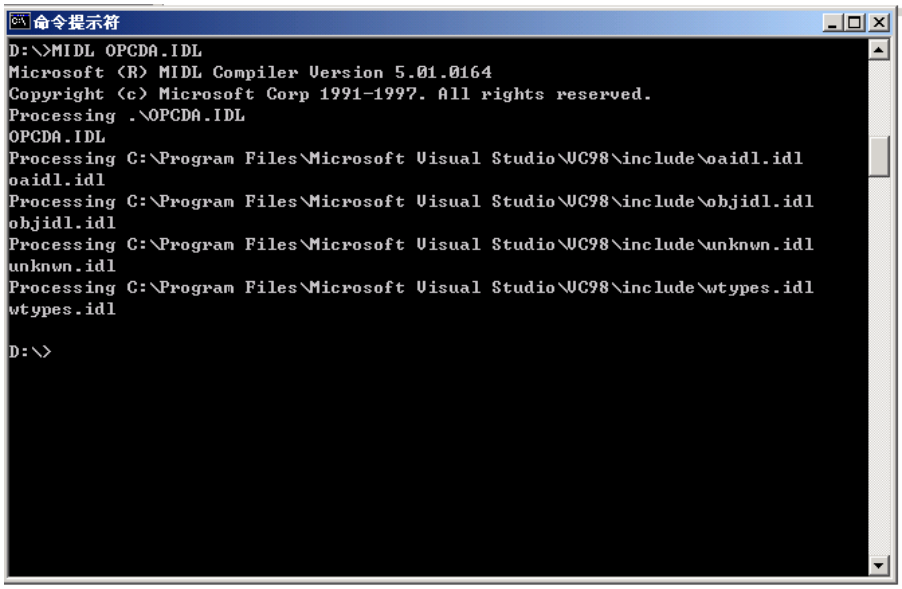


图 3.2MIDL 编译.IDL 文件

实际上不需要这样来生成头文件，在下面会介绍。

打开 VC++6.0，选择 ATL COM AppWizard，目录选择 D:\OPCDA，项目名称为 OPCDA。

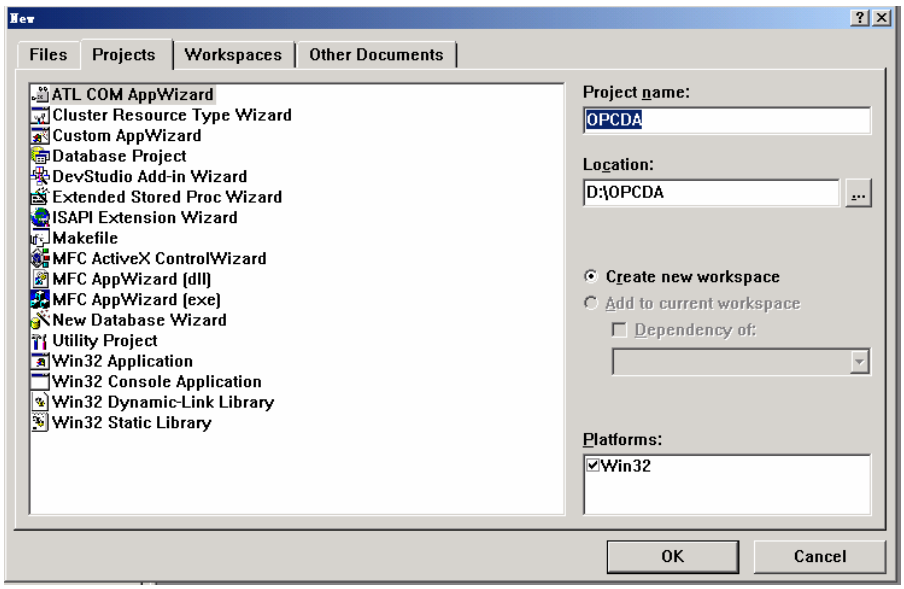


图 3.2 输入项目名称

点击 OK。

选择 Executable(EXE)方式。

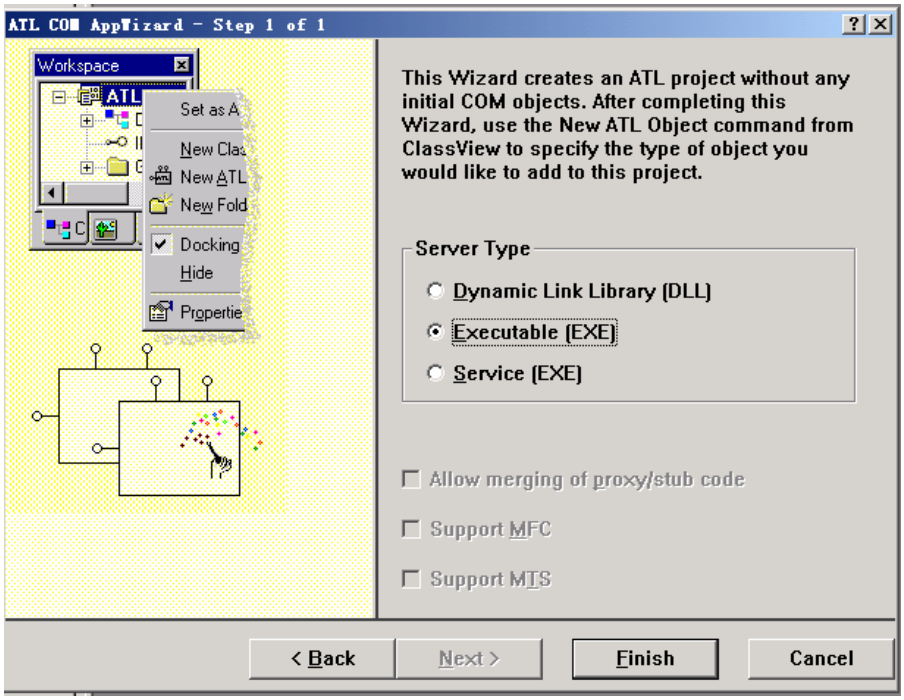


图 3.3 选择组件类型

点击 Finish。

由此生成的文件等可以参照第二章。

下面要做的工作是将 OPC 规范定义的 IDL 文件加入到我们的项目中。

现在的 IDL 文件如下：

```
// OPCDA.idl : IDL source for OPCDA.dll
//
// This file will be processed by the MIDL tool to
// produce the type library (OPCDA.tlb) and marshalling code.
import "oaidl.idl";
import "ocidl.idl";
[
    uuid(9DB24EAC-C452-477B-8B70-871F51F3330D),
    version(1.0),
    helpstring("OPCDA 1.0 Type Library")
```

```
]
library OPCDALib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
};
```

打开工程的 OPCDA.IDL 文件，将 650 行前的内容复制到工程中的 IDL 文件的前九行，把前九行替换为我们复制的内容。

然后在工程中的最后处的修改 library OPCDALib 中如下：

```
library OPCDALib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    interface IOPCServer ;
    interface IOPCServerPublicGroups ;
    interface IOPCBrowseServerAddressSpace;
    interface IOPCGroupStateMgt ;
    interface IOPCPublicGroupStateMgt ;
    interface IOPCSyncIO ;
    interface IOPCAsyncIO ;
    interface IOPCItemMgt;
    interface IEnumOPCItemAttributes ;
    interface IOPCDataCallback ;
    interface IOPCAsyncIO2 ;
    interface IOPCItemProperties ;
};
```

然后进行工程的第一次编译。可以看到 ClassView 中多了很多接口定义。双击每一个接口都可以看到每一个接口下的方法。

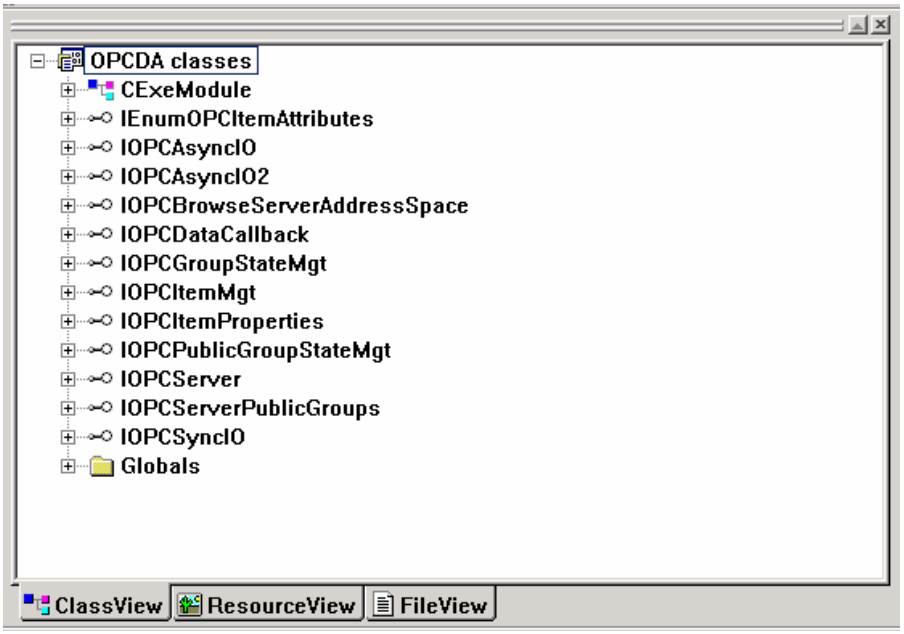


图 3.4OPCDA ClassView

通过这样的方式便将 OPC 规范定义的接口全部引入到现在的工程中，以后不需要对这些接口和这些接口的方法进行修改。

如第二章所示，完成一个简单的组件并不复杂。然而对于 OPC 服务器的开发，仅仅接口这一块的工作就比较大。

从 OPC 规范可以看出，OPC 规范定义了服务器，组，以及项，服务器相当于组的容器，组相当于项的容器，我们的服务器需要支持多个客户访问，每个服务器对象又需要支持多个组对象的创建，每个组对象可以管理多个项。因此除了实现服务器对象和组对象的接口外，我们需要对服务器，组以及项进行管理。

在此，工程需要建立三个管理类：

XXXServer

XXXGroup

XXXItem

假设工程的服务器对象类为 TestServer，组对象类为 TestGroup。

XXXServer 管理 TestServer，并对数据的刷新进行管理。

考虑到服务器设计功能最大支持 10 个组，因此在 Resource.h 文件中定义如下宏：

#define MAXGROUPNUM 10

下面来建立两个对象（服务器，客户）。

Server 对象的建立：

1. 在 ATL Object Wizard 中选择 Simple Object，Names 属性页如下：

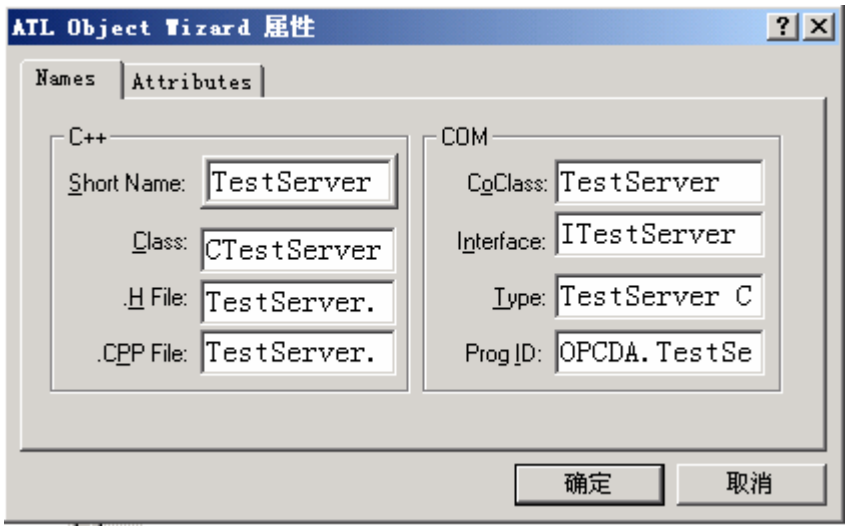


图 3.5

Attributes 属性如下：

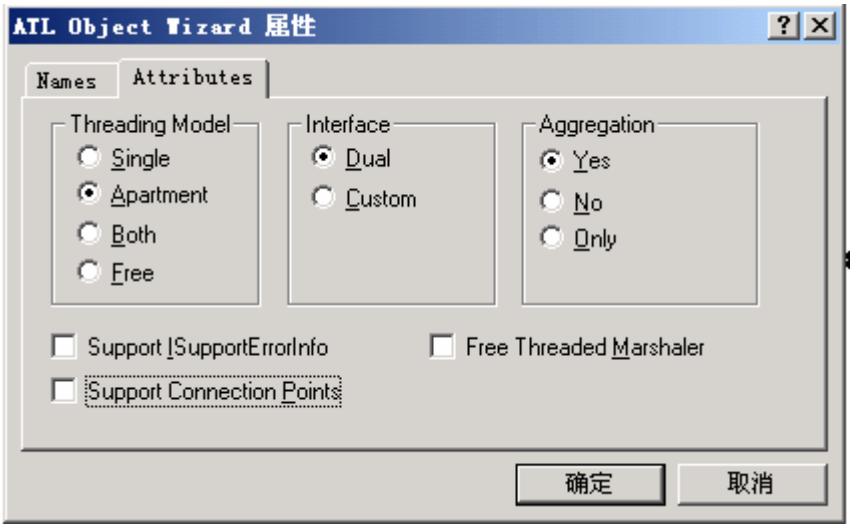


图 3.6

在此要注意的是对象的聚合模型（Aggregate），由于 COM 规范不允许对象的实现继承，但是可以通过聚合方式重用其它的 COM 对象。ATL 对象属性设置中的聚合模型可以指定待创建的 COM 对象是否支持聚合模型。本章的服务器程序中需要支持聚合。。

现在我们来编译一下，出现如下错误：

```
D:\OPCDA\OPCDA.idl(651) : error MIDL2025 : syntax error : expecting ] (
D:\OPCDA\OPCDA.idl(661) : error MIDL2025 : syntax error : expecting an
D:\OPCDA\OPCDA.idl(661) : error MIDL2026 : cannot recover from earlier
```

图 3.7

这是因为这个对象的接口没有定义造成的，为适应于 OPC，需要在 IDL 文件中做如下修改：

首先注释掉：

```
/*  [
        object,
        uuid(8213CDA9-B76B-43FA-9396-3F6B792CFA4E),
        dual,
        helpstring("ITestServer Interface"),
        pointer_default(unique)
    ]
```

```
    interface ITestServer : IDispatch
    {
    };
*/
```

然后将倒数第三行的 ItestServer 改为 IOPCServer。

重新编译，上述错误消除，但是又产生了 6 个新的错误如下：

```
d:\opcda\testserver.h(13) : error C2065: 'ITestServer' : undeclared identifier
d:\opcda\testserver.h(13) : error C2065: 'IID_ITestServer' : undeclared identifier
d:\opcda\testserver.h(14) : fatal error C1903: unable to recover from previous error(s);
TestServer.cpp
d:\opcda\testserver.h(13) : error C2065: 'ITestServer' : undeclared identifier
d:\opcda\testserver.h(13) : error C2065: 'IID_ITestServer' : undeclared identifier
d:\opcda\testserver.h(14) : fatal error C1903: unable to recover from previous error(s);
```

图 3.8

这是由于 ITestServer 的定义有错，ITestServer,IID_ITestServer 没有定义造成的。将所有的 ITestServer 替换为 IOPCServer，同时在类定义中做如下修改：// COM_INTERFACE_ENTRY(IDispatch)（注释掉这一行）

因为 CTestServer 类拥有 IOPCServer 接口，所以需要对接口的方法定义。

将下列方法按照第二章的方法添加：

```
STDMETHODIMP            GetStatus(
    OPCSERVERSTATUS** ppServerStatus);
STDMETHODIMP            GetErrorString(
    HRESULT hr,
    LCID locale,
    LPWSTR *ppstring);
STDMETHODIMP            AddGroup(
    LPCWSTR szName,
    BOOL bActive,
    DWORD dwRequestedUpdateRate,
    OPCHANDLE hClientGroup,
    LONG *pTimeBias,
```

```
        FLOAT *pPercentDeadband,
        DWORD dwLCID,
        OPCHANDLE *phServerGroup,
        DWORD *pRevisedUpdateRate,
        REFIID riid,
        LPUNKNOWN *ppUnk
        );
    STDMETHODIMP GetGroupByName(
        LPCWSTR szGroupName,
        REFIID riid, LPUNKNOWN *ppUnk);
    STDMETHODIMP RemoveGroup(
        OPCHANDLE groupHandleID,
        BOOL bForce);
    STDMETHODIMP CreateGroupEnumerator(
        OPCENUMSCOPE dwScope,
        REFIID riid,
        LPUNKNOWN *ppUnk
        );
```

在每一个函数实现的地方加上 return S_OK;

CTestServer 类定义如下：

```
// CTestServer
class ATL_NO_VTABLE CTestServer :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CTestServer, &CLSID_TestServer>,
    public IDispatchImpl<IOPCServer, &IID_IOPCServer,
&LIBID_OPCDALib>
{
public:
    CTestServer()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_TESTSERVER)

DECLARE_PROTECT_FINAL_CONSTRUCT()
```

```
    BEGIN_COM_MAP(CTestServer)
        COM_INTERFACE_ENTRY(IOPCServer)
    //    COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    // IOPCServer
    public:
        STDMETHOD(CreateGroupEnumerator)(OPCENUMSCOPE
    dwScope,REFIID riid,LPUNKNOWN *ppUnk);
        STDMETHOD(RemoveGroup)(OPCHANDLE
    groupHandleID,BOOL bForce);
        STDMETHOD(GetGroupByName)( LPCWSTR szGroupName,
    REFIID riid, LPUNKNOWN *ppUnk);
        STDMETHOD(AddGroup)(LPCWSTR szName,BOOL
    Active,DWORD dwRequestedUpdateRate,OPCHANDLE
    hClientGroup,LONG *pTimeBias,FLOAT *pPercentDeadband,DWORD
    dwLCID,OPCHANDLE *phServerGroup,DWORD
    *pRevisedUpdateRate,REFIID riid,LPUNKNOWN *ppUnk);
        STDMETHOD(GetErrorString)(HRESULT hr,LCID
    locale,LPWSTR *ppstring);
        STDMETHODIMP GetStatus( OPCSERVERSTATUS**
    ppServerStatus);
};
```

到现在为止完成了 OPC 服务器对象接口 IOPCServer 的方法实现，但只是个空的接口。

下面介绍如何实现 OPC 组对象以及它的一些接口。

Names 属性框如下

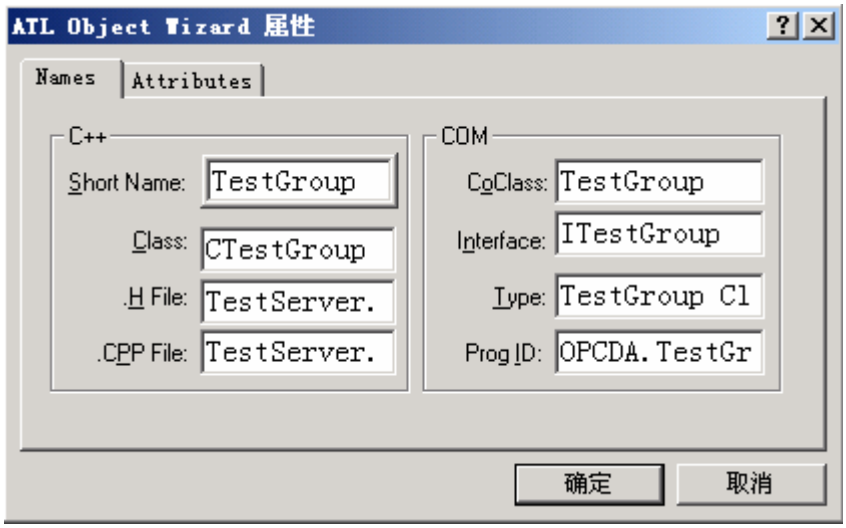


图 3.9

Attributies 属性框如下：注意要选择 Support Connection Points 选项。

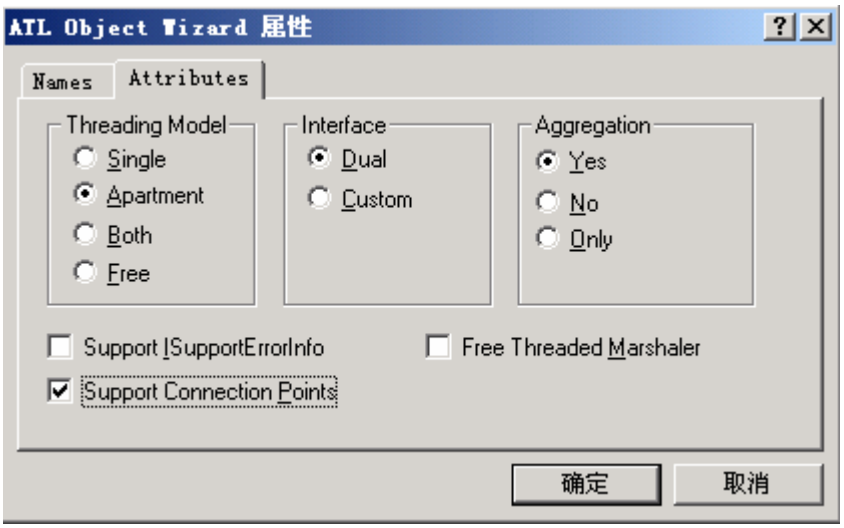


图 3.10

编译出现三个错误：

```
D:\OPCDA\OPCDA.idl(651) : error MIDL2025 : syntax error : expecting ] or , near "["
D:\OPCDA\OPCDA.idl(671) : error MIDL2025 : syntax error : expecting an interface name or Dispatch
D:\OPCDA\OPCDA.idl(671) : error MIDL2026 : cannot recover from earlier syntax errors; aborting c
```

图 3.11

类似于 Server 对象的修改：

IDL 文件修改如下

```
/*  [
        object,
        uuid(11BCB085-D4E2-4B69-B259-DFB2982A12EE),
        dual,
        helpstring("ITestGroup Interface"),
        pointer_default(unique)
    ]
    interface ITestGroup : IDispatch
    {
    };*/
/*  [
        uuid(FBE0721B-8705-4287-9758-D1EE7A046CF2),
        helpstring("_ITestGroupEvents Interface")
    ]
    dispinterface _ITestGroupEvents
    {
        properties:
        methods:
    };

    [
        uuid(5CAD073F-3421-42F5-9479-BC536F142BEE),
        helpstring("TestGroup Class")
    ]
    Coclass TestGroup
    {
        [default] interface ITestGroup;
        [default, source] dispinterface _ITestGroupEvents;
    };*/
```

CTestGroup 定义处修改

```
// CTestGroup
class ATL_NO_VTABLE CTestGroup :
    public CComObjectRootEx<CComSingleThreadModel>,
//  public CComCoClass<CTestGroup, &CLSID_TestGroup>,
    public IConnectionPointContainerImpl<CTestGroup>
//  public IDispatchImpl<ITestGroup, &IID_ITestGroup,
```

```
&LIBID_OPCDALib>
{
public:
    CTestGroup()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_TESTGROUP)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CTestGroup)
//    COM_INTERFACE_ENTRY(ITestGroup)
//    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()
BEGIN_CONNECTION_POINT_MAP(CTestGroup)
END_CONNECTION_POINT_MAP()
// ITestGroup
public:
};
```

OPCDA.CPP 修改如下

```
BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_TestServer, CTestServer)
//OBJECT_ENTRY(CLSID_TestGroup, CTestGroup)
END_OBJECT_MAP()
```

重新编译，错误消除。

下面来说一下做如上修改的原因。

因为服务器程序启动时是根据一个 CLSID_TestServer 来实现的，而且 ATL 生成了 TestServer 类型库，而 Server 对象对 Group 对象的创建是在程序内部创建，不是通过类型库来创建的，为了避免 Group 对象同样生成类型库，因此需要对以上文件做修改才可以，使服务器程序编译时只能生成 TestServer 的类型库。

下面为 CTestGroup 添加 IOPCItemMgt，及 IOPCItemMgt 的方法，我们采用手动添加的方式。

在 CTestGroup 定义处添加代码如下：

```
public IDispatchImpl<IOPCItemMgt, &IID_IOPCItemMgt,
&LIBID_OPCDALib>
BEGIN_COM_MAP(CTestGroup)处添加代码如下：
COM_INTERFACE_ENTRY(IOPCItemMgt)
```

添加 IOPCItemMgt 的方法后，CTestGroup 定义如下：

```
// CTestGroup
class ATL_NO_VTABLE CTestGroup :
    public CComObjectRootEx<CComSingleThreadModel>,
//  public CComCoClass<CTestGroup, &CLSID_TestGroup>,
    public IConnectionPointContainerImpl<CTestGroup>,
        public IDispatchImpl<IOPCItemMgt, &IID_IOPCItemMgt,
&LIBID_OPCDALib>
//  public IDispatchImpl<ITestGroup, &IID_ITestGroup,
&LIBID_OPCDALib>
{
public:
    CTestGroup()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_TESTGROUP)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CTestGroup)
//  COM_INTERFACE_ENTRY(ITestGroup)
//  COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
    COM_INTERFACE_ENTRY(IOPCItemMgt)
END_COM_MAP()
BEGIN_CONNECTION_POINT_MAP(CTestGroup)
```

```
END_CONNECTION_POINT_MAP()


// ITestGroup// IOPCItemMgt
public:
    STDMETHOD(SetDatatypes)(DWORD
 dwNumItems,OPCHANDLE    * phServer,VARTYPE       *
 pRequestedDatatypes,HRESULT      ** ppErrors);
    STDMETHOD(SetClientHandles)(DWORD
 dwNumItems,OPCHANDLE    * phServer,OPCHANDLE    *
 phClient,HRESULT      ** ppErrors);
    STDMETHOD(SetActiveState)(DWORD dwNumItems,
 OPCHANDLE    * phServer,BOOL    bActive,HRESULT      **
ppErrors);
    STDMETHOD(RemoveItems)(DWORD
 dwNumItems,OPCHANDLE    * phServer,HRESULT      ** ppErrors);
    STDMETHODIMP ValidateItems(DWORD
 dwNumItems,OPCITEMDEF * pItemArray,BOOL
 bBlobUpdate,OPCITEMRESULT    ** ppValidationResults,HRESULT
 ** ppErrors);
    STDMETHODIMP AddItems(DWORD
 dwNumItems,OPCITEMDEF    * pItemArray,OPCITEMRESULT **
 ppAddResults,HRESULT ** ppErrors);

};

////////////////////////////////////////////////////////////////////////////
// CTestGroup

STDMETHODIMP CTestGroup::AddItems(DWORD dwNumItems,
OPCITEMDEF *pItemArray, OPCITEMRESULT **ppAddResults,
HRESULT **ppErrors)
{
    return S_OK;
}
```

testserver.cpp 中 CtestGroup 实现如下：

```
STDMETHODIMP CTestGroup::ValidateItems(DWORD dwNumItems,
```

```
OPCITEMDEF *pItemArray, BOOL bBlobUpdate, OPCITEMRESULT
**ppValidationResults, HRESULT **ppErrors)
{
    return S_OK;
}

STDMETHODIMP CTestGroup::RemoveItems(DWORD dwNumItems,
OPCHANDLE *phServer, HRESULT **ppErrors)
{
    return S_OK;
}

STDMETHODIMP CTestGroup::SetActiveState(DWORD dwNumItems,
OPCHANDLE *phServer, BOOL bActive, HRESULT **ppErrors)
{
    return S_OK;
}

STDMETHODIMP CTestGroup::SetClientHandles(DWORD
dwNumItems, OPCHANDLE *phServer, OPCHANDLE *phClient,
HRESULT **ppErrors)
{
    return S_OK;
}

STDMETHODIMP CTestGroup::SetDatatypes(DWORD dwNumItems,
OPCHANDLE *phServer, VARTYPE *pRequestedDatatypes,
HRESULT **ppErrors)
{
    return S_OK;
}
```

现在完成了两个对象的创建，除了现在定义的接口，仍然需要添加其它的接口，方法同上，在此不多论述。接口如下：

Server:

IOPCBrowseServerAddressSpace

Group:

IOPCAsyncIO2

IOPCGroupStateMgt

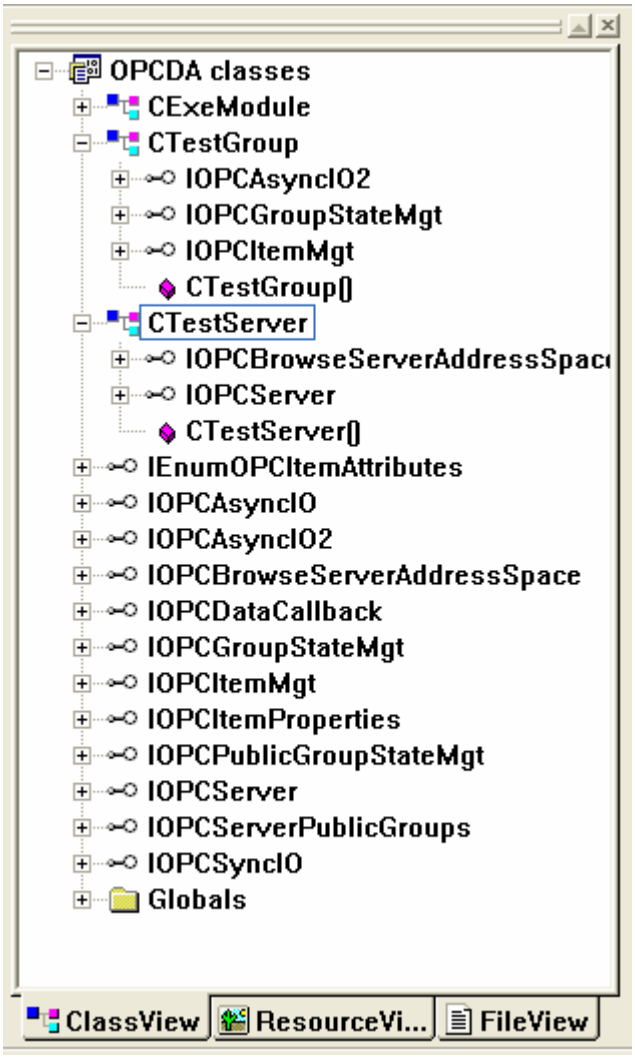


图 3.12

双击每个接口，可以看到接口下面的方法。如 IOPCServer

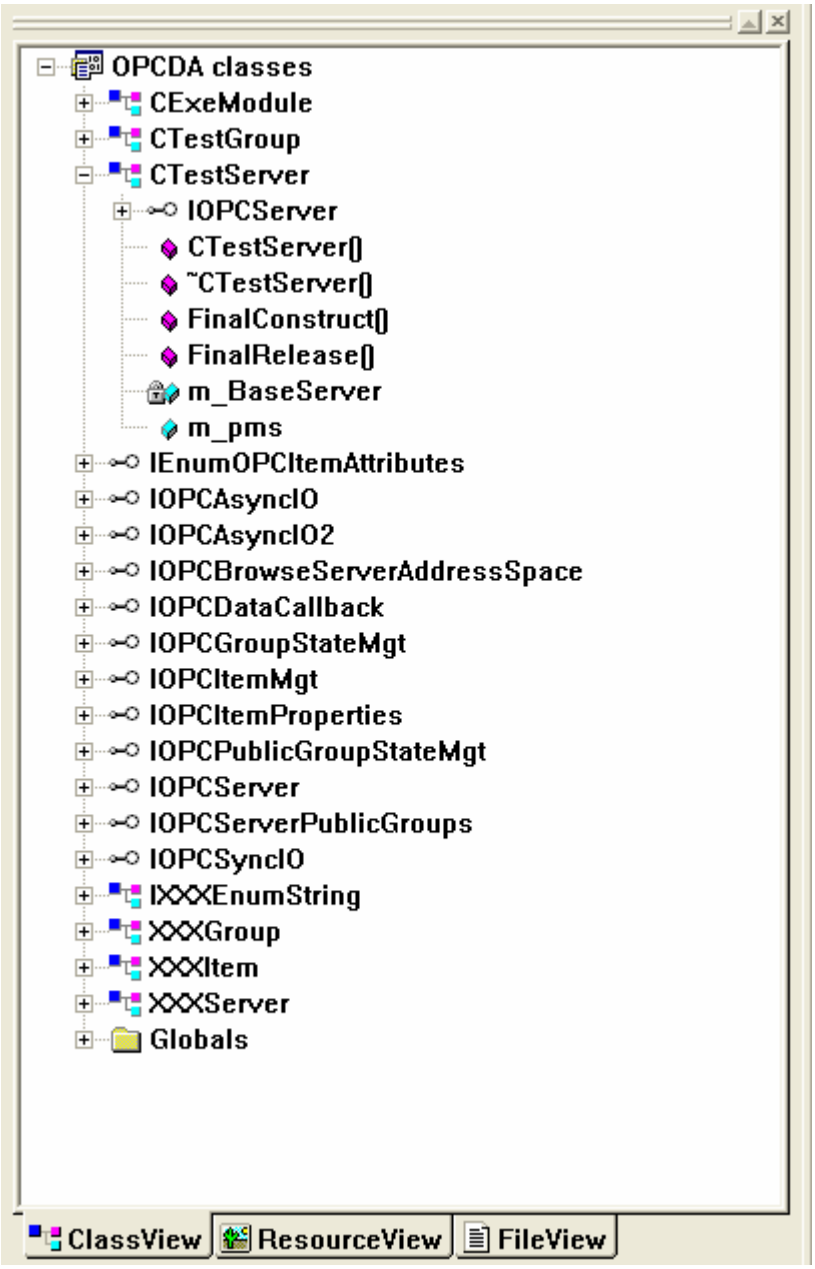


图 3.13

现在来重新编译一下工程，没有错误出现。

在此之前的示例程序为 STEP0。

3.6 OPC 服务器的类实现

现在工程中有了 Server 对象和 Group 对象的空壳，实际上现在什么功能也没有。那么如何实现 Server 和 Group 接口的访问，如何实现对 Server 对象和 Group 对象的管理呢？首先需要来讨论在一个服务器程序中实现两个对象接口的访问问题。

从下面的内容开始，你需要参照示例程序才可以进行，因为 OPC 服务器的设计是个比较大的工程，如果不是针对性的学习，可能大家理解的比较慢，所以为加快读者的了解，本书提供源程序，请参考示例程序。

在服务器程序中，大家可以看到， 只有一个 CLSID 定义，也就是说客户程序在访问服务器时只需要创建一个 Server 对象，Group 对象的接口是通过对 Server 对象的创建来实现的。

COM 的特性的可重用性包括包容和聚合两种。这两种类型在实现方法上有所不同。由于在前面的新建工程的选项时选择了支持聚合，因此本章在本程序中以聚合方式来实现对 Group 对象接口的支持。Server 对象对 Group 对象的聚合如下图所示：

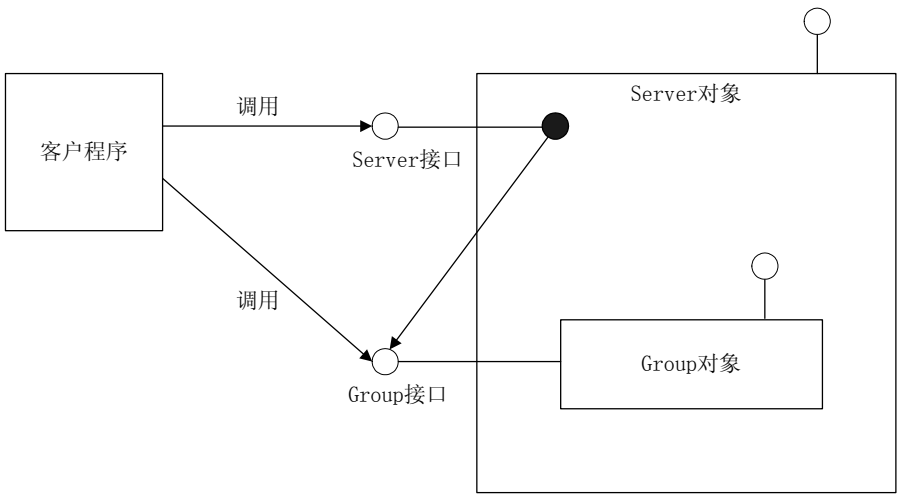


图 3.14 Server 对象对 Group 对象的聚合示意图

在 CTestServer 定义处可以看到

```
BEGIN_COM_MAP(CTestServer)
    COM_INTERFACE_ENTRY(IOPCServer)
//  COM_INTERFACE_ENTRY(IOPCBrowseServerAddressSpace)
    COM_INTERFACE_ENTRY_AGGREGATE(IID_IOPCItemMgt,
m_pms)
    COM_INTERFACE_ENTRY_AGGREGATE(IID_IOPCAsyncIO2,
m_pms)
COM_INTERFACE_ENTRY_AGGREGATE(IID_IOPCGroupStateMgt,
m_pms)
END_COM_MAP()
CComObject<CTestGroup>* m_pms[10];
HRESULT FinalConstruct()
{
    HRESULT hr;
    for(int j=0;j<10;j++)
    {
    hr=CComObject<CTestGroup>::CreateInstance(&m_pms[j]);
    if(FAILED(hr))return hr;
    m_pms[j]->g_seqnum=j;
    m_pms[j]->m_Parent=this;
    }
    return hr;
}
HRESULT   FinalRelease()
{
    for(int j=0;j<10;j++)
    m_pms[j]->Release() ;
    return S_OK;
}
```

上面的这段代码定义了 Server 对象聚合的 Group 接口，以及如何在创建 Server 对象时，创建 Group 对象。下面说明如下：

COM_INTERFACE_ENTRY_AGGREGATE(iid, punk) ，用来定义要聚合的对象的接口，iid 代表 GUID，punk 代表 IUnknown 指针。这个

是 ATL 中一个非常有用的宏定义，如果用 MFC 来实现聚合的对象，则要麻烦的多，而通过 ATL 只需要一句语句就可以实现聚合。

CComObject<CTestGroup>* m_pms[10];//定义了 10 个 Group 对象指针。

在 FinalConstruct() 函数中来实现创建聚合对象，这个函数在 CTestServer 类初始化时调用，并通过调用 CreateInstance，来创建聚合的对象 Group，由于定义了最大十个 Group 组对象，所以我们程序中循环的次数为 10。

在 FinalRelease()函数中来实现聚合对象的释放，释放 IUnknown 指针，本函数在 CTestServer 析构时调用，在此函数中释放了通过 FinalConstruct（）创建的 Group 对象。

这里一定要理解两个 Final 函数的功能，作用，大家也可以参考 MSDN 的文档和微软网站的技术论文。

按照本章前面的设计功能，以及 OPC 规范，需要建立几个类来管理 Server,Group,Item。设计思路如下：

XXXServer 类作为 Server 对象的容器，来管理 Server 对象，并负责各个 Server 对象数据的刷新。

XXXGroup 类作为 Group 对象的容器，来管理 Group 对象，并负责各个 Group 对象数据的刷新，XXXItem 的管理，检查异步读写通知，检查数据刷新，检查是否回调等。

XXXItem 类作为 Item 的容器，来管理 Item，Item 的仿真，ITEM 值的仿真实现等。

XXXServer 类的定义：

```
class XXXServer
{
public:
```

```
        void GetAddressList( OPCBROWSETYPE
        dwBrowseFilterType,LPCWSTR szFilterCriteria,VARTYPE
        vtDataTypeFilter,DWORD dwAccessRightsFilter,
        LPOLESTR**AddressList,int *AddressCount);
        XXXServer();//构造函数
        virtual ~XXXServer();//析构函数
        //定义下列嵌套类为其友类
        friend class XXXGroup;
        friend class CTestServer;
        friend class CTestGroup;
        FILETIME        mLastUpdate;
        void      UpdateData(DWORD tics);
        struct {
        int         inuse;
        XXXGroup  *pGroup;
        CTestGroup *pG;
        } m_groups[N_GRPS];
        CTestServer *pServer;
    private:
        int   m_Slot;
    };
```

GetAddressList 主要为 IOPCBrowseServerAddressSpace 接口服务，用来获得地址空间。

UpdateData 用来刷新数据。

```
struct {
  int       inuse;
  XXXGroup*pGroup;
  CTestGroup *pG;
 } m_groups[N_GRPS];
```

用来定义一个 Server 对象包容的 Group 对象结构。

XXXGroup 类定义：

```
class XXXGroup
{
public:
```

```
    //定义下列嵌套类为其友类
    friend class XXXServer;
    friend class CTestServer;
    friend class CTestGroup;
    friend class XXXItem;
    XXXGroup(XXXServer *Server);//构造函数
    virtual ~XXXGroup();//析构函数
    void        DataCheck(int i,DWORD tics );
    // Item List Management Functions
    BOOL   ItemValid(OPCHANDLE h);  //检查 Item 句柄是否合法
    int   ItemHandles(void);   // Item 句柄
    HRESULT   ItemAlloc(OPCHANDLE *h); //  分配句柄
    HRESULT   ItemReAlloc(OPCHANDLE h);   //  分配指定的句柄
    void ItemSet(OPCHANDLE h, XXXItem * p); //设置一个 SLOT 中
    //的 Item
    XXXItem    *ItemPtr(OPCHANDLE h);   //通过句柄，返回指针
    void ItemFree(OPCHANDLE h);
    void CheckDOAsyncRead( int flag );
    void CheckDOAsyncWrite( int flag );
    void SendStream(int Count, OPCHANDLE *ItemHandleList,
    DWORD tid,int flag,int m_Fire);
    void Scan( void );
    void CheckDOOnDataChange( int flag );
private:
    XXXServer  *m_ParentServer;     //GROUP  父 SERVER 指针
    DWORD     m_dwRevisedRate;   //Update Rate
    FLOAT     m_Deadband;        //Deadband
    DWORD     m_LCID;//Server uses english language to for text
    values
    LONG      m_TimeBias;        //TimeBias
    BOOL      m_bActive;         //TRUE,Active,FALSE,InActive
    WCHAR          *m_szName;
    OPCHANDLE   m_ServerGroupHandle;   //服务器句柄
    OPCHANDLE   m_ClientGroupHandle;   //客户句柄
    int test;
    BOOL      m_AsyncWriteActive;
    BOOL      m_AsyncWriteCancel;
```

```
    DWORD          m_AsyncWriteTID;
    DWORD          m_AsyncReadTID;
    BOOL           m_AsyncReadActive;
    OPCHANDLE   *m_AsyncReadList;
    BOOL           m_AsyncReadCancel;
    OPCHANDLE   *m_AsyncWriteList;
    HRESULT hrMasterquality;
    HRESULT hrMastererror;
    VARIANT * pvValues;
    WORD * pwQualities;
    long           m_scan;
    struct {
    int       inuse;
    XXXItem    *pItem;
    } m_items[N_ITEMS];
};//结束
```

XXXGroup 类定义说明：

```
BOOL       ItemValid(OPCHANDLE h);判断 ITEM 句柄是否合法。
int        ItemHandles(void);返回最大的句柄数。
HRESULT        ItemAlloc(OPCHANDLE *h);分配 ITEM 句柄。
HRESULT        ItemReAlloc(OPCHANDLE h);重新分配 ITEM 句柄。
void       ItemSet(OPCHANDLE h, XXXItem * p);设置一个句柄。
XXXItem        *ItemPtr(OPCHANDLE h);返回句柄指针。
void       ItemFree(OPCHANDLE h);释放 ITEM 句柄。
void       CheckDOAsyncRead( int flag );检查是否有异步读标志。
void       CheckDOAsyncWrite( int flag );检查是否有异步写标志。
void       SendStream(int Count, OPCHANDLE *ItemHandleList,
DWORD tid,int flag,int m_Fire);回调
void       Scan( void );扫描并仿真数据。
void       CheckDOOnDataChange( int flag );检查是否有数据变化
//事件发生。
XXXServer  *m_ParentServer;     //GROUP 父 SERVER 指针
DWORD          m_dwRevisedRate;        //刷新速率
FLOAT      m_Deadband;                 //死区
DWORD          m_LCID;                     //语言标识
LONG       m_TimeBias;                 //时间
BOOL       m_bActive;              //TRUE,活动,FALSE,非活动
```

```
WCHAR          *m_szName;                    //名字
OPCHANDLE   m_ServerGroupHandle;   //Server 句柄
OPCHANDLE   m_ClientGroupHandle;   //Client  句柄
BOOL        m_AsyncWriteActive;//异步写标志
BOOL        m_AsyncWriteCancel;//异步写取消标志
DWORD           m_AsyncWriteTID;//
DWORD           m_AsyncReadTID;//
BOOL        m_AsyncReadActive;//异步读标志
OPCHANDLE   *m_AsyncReadList;//异步读句柄列表
BOOL            m_AsyncReadCancel;//异步读取消标志
OPCHANDLE   *m_AsyncWriteList;//异步写句柄列表
HRESULT hrMasterquality;//品质码
HRESULT hrMastererror;//错误码
VARIANT * pvValues; //值
WORD * pwQualities; //品质
long            m_scan;//
struct {
       int      inuse;
       XXXItem   *pItem;
    } m_items[N_ITEMS];//定义最大 ITEM 数目。
```

XXXItem 类的定义：

```
class XXXItem
{
public:
    friend class XXXGroup;
    XXXItem(XXXGroup *group);
    virtual ~XXXItem();
    HRESULT   Init(int j, OPCITEMDEF *def, OPCITEMRESULT
*ir);
    HRESULT        SetDatatype(VARTYPE v);
    void           MarkAsChanged( WORD flg );
        XXXItem    * Clone(XXXGroup * newparent, OPCHANDLE
    newserveritemhandle);
    void           SetHandle(OPCHANDLE h);
    void           SetActive(BOOL a);
    void           ClearChanged( WORD flg );
    BOOL Changed( WORD flg);
```

```
    void        Simulate( void );
    void        QueDeviceWrite( VARIANT * v);
    HRESULT     SetValue(VARIANT * v);
private:
    OPCHANDLE   m_hServerItem;
    VARTYPE     m_vtCanonical;
    OPCHANDLE   m_hClientItem;
    WCHAR       * m_szItemID;
    WCHAR       * m_szAccessPath;
    VARTYPE     m_vtRequested;
    BOOL        m_bActive;
    FLOAT       m_SimValue;
    FLOAT       m_Value;
    WORD        m_Quality;
    FILETIME    m_TimeStamp;
    HRESULT     m_LastWriteError;
    BOOL        m_GenData;
    WORD        m_AsyncMask;
    XXXGroup    *m_ParentGroup;
};//结束
HRESULT   Init(int j, OPCITEMDEF *def, OPCITEMRESULT
*ir);//ITEM 部分参数的初始化。
HRESULT     SetDatatype(VARTYPE v);//设定数值类型
XXXItem   * Clone(XXXGroup * newparent, OPCHANDLE
newserveritemhandle);//ITEM 的复制。
void        SetHandle(OPCHANDLE h);//设定句柄。
void        SetActive(BOOL a);//设定活动/非活动
void        Simulate( void );//仿真数据
OPCHANDLE   m_hServerItem;
VARTYPE     m_vtCanonical;
OPCHANDLE   m_hClientItem;
WCHAR       * m_szItemID;
WCHAR       * m_szAccessPath;
VARTYPE     m_vtRequested;
BOOL        m_bActive;

// 仿真数据的变量定义
```

```
FLOAT       m_SimValue;
FLOAT       m_Value;
WORD        m_Quality;
FILETIME    m_TimeStamp;

HRESULT         m_LastWriteError;    // 异步写结果
BOOL        m_GenData;
WORD        m_AsyncMask;
XXXGroup  *m_ParentGroup;
```

同时需要对 CTestServer，CTestGroup 类的定义函数添加实现过程。详看源程序 STEP1。大家注意到把 IOPCBrowseServerAddressSpace 接口及其方法都注释掉了，IOPCBrowseServerAddressSpace 的实现方法将在后面的章节中详细介绍。通过这些类的定义，大家应该对整个的程序结构有了比较深的理解。

服务器程序应客户程序调用接口添加 Group，添加 Item 的结构如下：

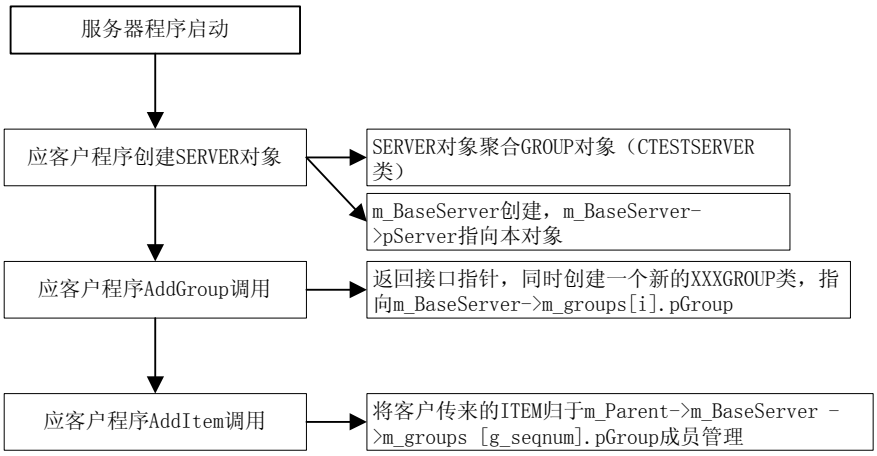


图 3.15

在 STEP1 示例程序中，大家注意到与 STEP0 中除了这些类的实现外，还有一些代码是不一样的。

在 OPCDA.cpp 文件的 WinMain 主程序人口的实现方法中，多了创建主窗口和应用程序初始化，以及消息分发循环等。

对于进程内 OPC 服务器，由于是以动态连接库形式存在，一般而言并不存在主窗口的创建问题。而对于以 EXE 形式存在的进程外 OPC 服务器，即本章要实现的服务器，可以通过创建一个用来显示程序运行的主窗口。对于一个窗口而言最主要的元素是窗口句柄（handle）和窗口过程（window procedure），窗口句柄是一个系统全局唯一标识，对窗口的各种操作都以句柄为基础；窗口过程是一个回调（CallBack）函数，它负责处理窗口的各种消息和事件，当系统接收到发给此窗口的消息或者在此窗口内发生事件时，系统就会调用此窗口过程。WNDCLASS 结构包含了 Windows 窗口类的属性，其参数指定了窗口的风格、窗口过程、窗口的背景色、窗口的菜单、图标和光标等。可以通过实现 WNDCLASS 结构的一个对象并用 RegisterClass（）函数对这个对象进行注册，然后调用 CreateWindow（）函数创建窗口并用 ShowWindow（）函数显示此程序运行主窗口，当窗口创建成功后，返回窗口句柄，以后就可以利用此句柄进行各种窗口操作，包括显示和隐藏窗口、调整窗口的位置和大小等。

具体代码如下：

```
/*WinMain 的实现：*/
extern "C" int WINAPI _tWinMain(HINSTANCE hInstance,
    HINSTANCE /*hPrevInstance*/, LPTSTR lpCmdLine, int
/*nShowCmd*/)
{
    lpCmdLine = GetCommandLine(); //this line necessary for
_ATL_MIN_CRT

#if _WIN32_WINNT >= 0x0400 & defined(_ATL_FREE_THREADED)
    HRESULT hRes = CoInitializeEx(NULL,
COINIT_MULTITHREADED);
#else
    HRESULT hRes = CoInitialize(NULL);
```

```
#endif
    _ASSERTE(SUCCEEDED(hRes));
    _Module.Init(ObjectMap, hInstance, &LIBID_OPCDALib);
    _Module.dwThreadID = GetCurrentThreadId();
    TCHAR szTokens[] = _T("-/");

    int nRet = 0;
    BOOL bRun = TRUE;
    LPCTSTR lpszToken = FindOneOf(lpCmdLine, szTokens);
    while (lpszToken != NULL)
    {
        if (lstrcmpi(lpszToken, _T("UnregServer"))==0)
        {

            _Module.UpdateRegistryFromResource(IDR_OPCDA,
            FALSE);
            nRet = _Module.UnregisterServer(TRUE);
            bRun = FALSE;
            break;
        }
        if (lstrcmpi(lpszToken, _T("RegServer"))==0)
        {

            _Module.UpdateRegistryFromResource(IDR_OPCDA,
            TRUE);
            nRet = _Module.RegisterServer(TRUE);
            bRun = FALSE;
            break;
        }
        lpszToken = FindOneOf(lpszToken, szTokens);
    }
    CoGetMalloc(MEMCTX_TASK, &pIMalloc);//added
    BOOL success = TRUE;//added
    if (bRun)
    {
        _Module.StartMonitor();
#if _WIN32_WINNT >= 0x0400 & defined(_ATL_FREE_THREADED)
```

```
                hRes =
_Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER,
                    REGCLS_MULTIPLEUSE | REGCLS_SUSPENDED);
                _ASSERTE(SUCCEEDED(hRes));
                hRes = CoResumeClassObjects();
    #else
                hRes =
_Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER,
                 REGCLS_MULTIPLEUSE);
    #endif
                 _ASSERTE(SUCCEEDED(hRes));
               serverInstance=hInstance;
               WNDCLASS wc;
               wc.style = CS_HREDRAW | CS_VREDRAW;
               wc.lpfnWndProc = OPCWndProc;
               wc.cbClsExtra = 0;
               wc.cbWndExtra = 0;
               wc.hInstance = serverInstance;
               wc.hIcon = 0;
               wc.hCursor = LoadCursor( NULL, IDC_ARROW);
               wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
               wc.lpszMenuName = 0;
               wc.lpszClassName = "OPC Sample";
               success = (RegisterClass( &wc) != 0);
               serverWindow = CreateWindow( "OPC Sample", "DEMO OPC
Server(OPC2.0 规范),作者 EMAIL:41063473@QQ.COM,QQ:41063473",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, serverInstance,
NULL) ;
               ShowWindow( serverWindow, SW_MINIMIZE);
               UpdateWindow( serverWindow);
                CoFileTimeNow( &serverStartTime);

               InitServerSlots();
               TimerID = SetTimer(serverWindow, 0, 500, 0);
               MSG msg;
                msg.wParam = 0;
```

```
            while (GetMessage(&msg, 0, 0, 0))
            {
                TranslateMessage( &msg);
                DispatchMessage( &msg);
            }
            _Module.RevokeClassObjects();
            //Sleep(dwPause); //wait for any threads to finish
        }
        pIMalloc->Release();
        _Module.Term();
        CoUninitialize();
        return nRet;
}
```

Windows 是消息驱动的系统。系统与应用程序之间、程序与程序之间、程序与各种输入设备之间以及窗口和窗口之间都是通过消息传递进行交互。这样一个应用程序就不需要周期性的检查鼠标和键盘的状态以获得用户的各种输入信息，只需要调用利用三个 API 函数和一个 While 结构建立一个消息分发循环。则系统会把所有传给此应用程序的输入信息都发过来。然后，利用 GetMessage（）函数从调用线程的消息队列中获取消息，然后调用程序的分发循环， TranslateMessage（）函数将虚拟按键（virtual-key）消息翻译为字符消息，然后将消息发送到调用线程的消息队列，DispatchMessage（）函数将消息发送给响应消息的窗口过程。如果 GetMessage（）函数从消息队列中得到一个 WM_QUIT 消息时，就退出消息循环。

```
/*消息分发循环的实现*/
LRESULT CALLBACK OPCWndProc( HWND hWnd, UINT iMsg,
WPARAM wParam, LPARAM lParam)
{
   switch ( iMsg) {
      case WM_DESTROY:
         KillTimer(serverWindow, TimerID);
         PostQuitMessage( 0);
```

```
            break;
        case WM_TIMER:
        //UpdateServers(500);//此处注释掉了 Server 对象的数据刷新功能
             break;
        case WM_CLOSE:
            KillTimer(serverWindow, TimerID);
            PostQuitMessage( 0);
         default:
            return DefWindowProc( hWnd, iMsg, wParam, lParam);
      }
      return 0;
}
```

最后需要说明的几个函数：

```
void _ InitServerSlots (void);
BOOL FindServerSlot(int *slot, XXXServer * pServer);
void ClearServerSlot(int i);
void UpdateServers(DWORD tics);
```

大家可能注意到了_tWinMain 函数中有 InitServerSlots 函数的调用。上面的这四个函数为服务器程序用来管理总的 Server 对象。下面对代码做简要的解释。

```
#define N_SERVERS       10    //最大十个 Server 对象的创建。
struct {
     BOOL   inuse;
     XXXServer  *pServer;
}slist[N_SERVERS];
```

slist 结构来管理 Server 对象。

```
void InitServerSlots(void)
{
     int i;

     for(i=0; i<N_SERVERS; i++)
     {
          slist[i].inuse = 0;
          slist[i].pServer = NULL;
     }
```

```
}
初始化 slist[N_SERVERS]为空
//////////////////////////////////////
BOOL FindServerSlot(int *slot, XXXServer * pServer)
{
    int i;
    // By convention, start with 1
    // so that 0 can indicate NO SLOT
    //
    *slot = 0;
    for(i=1; i<N_SERVERS; i++)
    {
        if(!slist[i].inuse)
        {
            slist[i].inuse = 1;
            slist[i].pServer = pServer;
            *slot = i;
            return TRUE;
        }
    }
    return FALSE;
}
```

本函数在 XXXServer 类的结构函数中调用，如果 slist[i]没有被引用，则 slist[i].inuse 置为 1，代表已经被引用，slist[i].pServer 指向 pServer,返回 TRUE，否则返回 FALSE。这样做的目的是为了实现对最大数目为 N_SERVERS 的 Server 对象的管理，即仅对前 N_SERVERS 个 Server 对象进行异步数据刷新处理。

```
void ClearServerSlot(int i)
{

    if(slist[i].inuse)
    {
        slist[i].inuse = 0;
        slist[i].pServer = NULL;
```

```
        }
    }
```

在 Server 对象被释放时调用，用来释放所隶属的 slist 的指针和引用标志。

```
    void UpdateServers(DWORD tics)
    {
        int i;
        for(i=0; i<N_SERVERS; i++)
        {
            if(slist[i].inuse)
            {
                if(slist[i].pServer)
                {
                    slist[i].pServer->UpdateData(tics);
                }
            }
        }
    }
```

在消息处理循环中调用，用来处理异步通讯的实现，根据异步通讯的标志或者数据刷新变化来处理异步通讯。

3.7 OPC 服务器的异步通讯实现

对于一个全面交互过程来说，同步通讯往往不能满足要求，在异步读取数据时，OPC 服务器主动和客户程序通信，此时，OPC 服务器提供出接口（outgoing interface），这些出接口是由客户程序实现，并将接口指针告诉 OPC 服务器对象，然后 OPC 服务器对象就利用此接口指针与客户程序进行通信。客户程序方实现这些接口的对象被称为接收器（sink，对于 OPC 客户程序来说，其接收器应该至少实现 IUnknown 和 IOPCDataCallBack 接口）。对客户程序来说，可以通过常规方式调用 OPC 服务器的接口，也可以通过接收器接受 OPC 服务

器发送的通知或事件，对 OPC 服务器来说，它的入接口和出接口分别承担了这两个通信过程，这样就实现了客户/服务器的双向通信。

异步通讯就是通过 COM 机制中的连接点实现的。可连接点对象通过 IConnectionPointContainer 接口管理所有的出接口。对应于每一个出接口，可连接点对象又管理了一个连接点（connection point）对象，每一个连接点对象实现了 IConnectionPoint 接口，客户程序通过连接点对象建立接收器与可连接对象的连接。

为了使用连接点（IConnectionPointContainer 和 IConnectionPoint 接口），客户程序必须创建一个对象支持 Iunknown 和 IConnectionPoint 接口，客户程序会传递一个指针给 IUnknown 接口去激活服务器的连接点。可连接对象的基本结构如下。

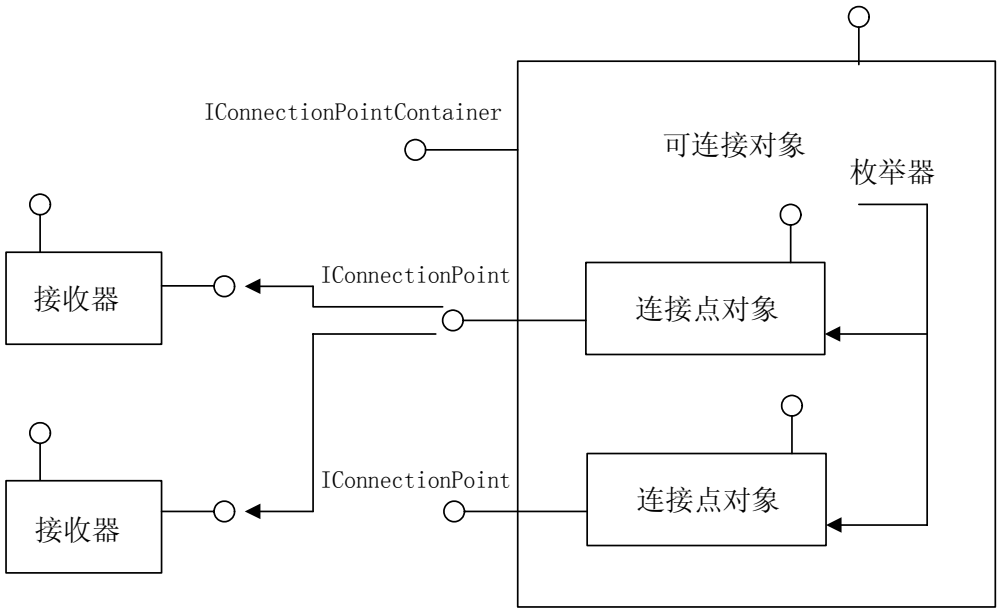


图 3.16

首先来看一下 IConnectionPoint 接口的方法。重要的两个方法 Advise 和 Unadvise，客户（接收器）通过调用连接点对象的 IConnectionPoint::Advise 来建立连接。Advise 的参数是一个指向接收器

的执行事件接口的 IUnknown 指针，Advise 返回一个连接标识 dwCookie，用于唯一标志此连接的 32 位整数，dwCookie 必须非零才能标识一个有效连接。通过 Advise 建立连接后，源对象（在此对应于 OPC 组对象）就可以激发事件或者向客户发出请求，事件或请求是源对象的内部程序逻辑，对应于 OPC 组对象来说，一般而言在客户调用异步读写操作或者组对象项成员的数据发生变化时，组对象对客户程序发出事件（OnDataChange 等）。当客户需要取消连接时，调用带有同样连接标识的 Unadvise 来取消连接。对于 IConnectionPoint 另外几个方法，我们在此不作讨论，请大家参考有关书籍或 MSDN。（注意，看 MSDN，此处要修改）。

在 2.0 规范中定义了义 IOPCAsyncIO2、IConncetionPointerContainer 和 IOPCDataCallBack 接口（用于客户端）支持对 OPC 服务器数据进行异步读写操作。

为了实现异步通讯功能，工程中需要修改 CTestGroup 的声明如下：

```
BEGIN_COM_MAP(CTestGroup)
    COM_INTERFACE_ENTRY(IOPCItemMgt)
    COM_INTERFACE_ENTRY(IOPCGroupStateMgt)
    COM_INTERFACE_ENTRY(IOPCAsyncIO2)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
 END_COM_MAP()
BEGIN_CONNECTION_POINT_MAP(CTestGroup)
    CONNECTION_POINT_ENTRY(IID_IOPCDataCallback)
 END_CONNECTION_POINT_MAP()
```

后面的一段宏用来告诉 IConnectionPointContainer，只有一个连接点 IID_IOPCDataCallback，换句话说是 IOPCDataCallback 接口。如果在工程中需要更多的连接点接口，工程中需要根据实际的接口需要添加更多的 CONNECTION_POINT_ENTRY 宏。

现在 IConnectionPoint 的问题解决了，但是我们怎么来实现实际的事

件（OnDataChange 等）功能呢。首先需要在工程中添加 IOPCDataCallback 方法定义。在 CTestGroup 类定义修改如下：

```
public IConnectionPointContainerImpl<CTestGroup>,
public IConnectionPointImpl<CTestGroup,
&IID_IOPCDataCallback, CComDynamicUnkArray>,
public IDispatchImpl<IOPCItemMgt, &IID_IOPCItemMgt,
&LIBID_OPCDALib>,
```

注意到在工程中新添加了一个 IConnectionPointImpl<CTestGroup, &IID_IOPCDataCallback, CComDynamicUnkArray>,这个语句对于创建一个连接点是必需的，CComDynamicUnkArray 拥有 IUnknown 指针的动态分配表，并有一个连接点的接口，并且可以作为 IconnectionPointImpl 模板类的一个参数。

加入这样一句是不能实现需要的功能的。需要添加下面的宏和函数定义。

```
BEGIN_CONNECTION_POINT_MAP(CTestGroup)
    CONNECTION_POINT_ENTRY(IID_IOPCDataCallback)
 END_CONNECTION_POINT_MAP()
```

函数定义：

```
STDMETHODIMP OnDataChange(
    DWORD dwTransid,
    OPCHANDLE hGroup,
    HRESULT hrMasterquality,
    HRESULT hrMastererror,
    DWORD dwCount,
    OPCHANDLE * phClientItems,
    VARIANT * pvValues,
    WORD * pwQualities,
    FILETIME * pftTimeStamps,
    HRESULT * pErrors
    );
STDMETHODIMP OnReadComplete(
    DWORD dwTransid,
```

```
        OPCHANDLE hGroup,
        HRESULT hrMasterquality,
        HRESULT hrMastererror,
        DWORD dwCount,
        OPCHANDLE * phClientItems,
        VARIANT * pvValues,
        WORD * pwQualities,
        FILETIME * pftTimeStamps,
        HRESULT * pErrors
        );
    STDMETHODIMP OnWriteComplete(
        DWORD dwTransid,
        OPCHANDLE hGroup,
        HRESULT hrMastererr,
        DWORD dwCount,
        OPCHANDLE * pClienthandles,
        HRESULT * pErrors
        );
    STDMETHODIMP OnCancelComplete(
        DWORD dwTransid,
        OPCHANDLE hGroup
        );
```

然后添加函数的实现过程，本章在此详细介绍 OnDataChange 函数的实现。

此函数的实现如下：

```
STDMETHODIMP CTestGroup::OnDataChange(
    DWORD dwTransid,
    OPCHANDLE hGroup,
    HRESULT hrMasterquality,
    HRESULT hrMastererror,
    DWORD dwCount,
    OPCHANDLE * phClientItems,
    VARIANT * pvValues,
    WORD * pwQualities,
    FILETIME * pftTimeStamps,
```

```
        HRESULT * pErrors
        )
        {
            IConnectionPointImpl<CTestGroup, &IID_IOPCDataCallback,
            CComDynamicUnkArray>* p = this;
            Lock();
            HRESULT hr = S_OK;
            IUnknown** pp = p->m_vec.begin();
            while (pp < p->m_vec.end() && hr == S_OK)
            {
                if (*pp != NULL)
                {
                    IOPCDataCallback* pIOPCEvent =
                    (IOPCDataCallback*)*pp;
                    hr =
                    pIOPCEvent->OnDataChange(dwTransid,hGroup,hrM
                    asterquality,hrMastererror,dwCount,phClientItems,pvV
                    alues,pwQualities,pftTimeStamps,pErrors);
                }
            pp++;
            }
        return hr;
    }
```

通过 pIOPCEvent->OnDataChange，就可以把 OPC 服务器的数据传送到客户端的 OnDataChange 函数中。

OPC 服务器程序调用这个函数的过程如下：

OPC 服务器周期性的刷新数据，在 UpdateData 函数中调用 Group 对象的 DataCheck 函数，Group 在 DataCheck 函数中调用 CheckDOOnDataChange 来检查是否有变化的数据，如果有变化数据，则传送数据到客户端。

3.8 OPC 服务器的浏览地址空间实现

OPC 服务器的浏览地址空间功能主要供客户程序来查看 OPC 服务器可用的 ITEM 信息。IOPCBrowseServerAddressSpace 接口实现 OPC 服务器的这个功能。

IOPCBrowseServerAddressSpace **(**optional**)**
HRESULT QueryOrganization(pNameSpaceType);
HRESULT ChangeBrowsePosition(dwBrowseDirection, szString);
HRESULT BrowseOPCItemIDs(dwBrowseFilterType, szFilterCriteria, vtDataTypeFilter, dwAccessRightsFilter, ppIEnumString);
HRESULT GetItemID(szItemDataID, szItemID);
HRESULT BrowseAccessPaths(szItemID, ppIEnumString);

IOPCBrowseServerAddressSpace:: QueryOrganization

```
HRESULT QueryOrganization(
        [out] OPCNAMESPACETYPE    * pNameSpaceType
          );
```

功能：

提供给客户程序一个方法来查询是 FLAT 型还是分等级的。

参数	描述
pNameSpaceType	查询 OPCNAMESPACE 的结果（OPC_NS_HIERARCHIAL 或 OPC_NS_FLAT）。

返回码

返回码	描述
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存空间。
E_INVALIDARG	非法的功能。
S_OK	操作成功。

注释：

FLAT 和分等级的地址空间操作不一样。如果结果是 FLAT 型的，客户不要再调用 BrowseOPCItemIDs 或 ChangeBrowsePosition 。

IOPCBrowseServerAddressSpace:: ChangeBrowsePosition

```
HRESULT ChangeBrowsePosition(
    [in]    OPCBROWSEDIRECTION dwBrowseDirection,
    [in, string] LPCWSTR    szString
    );
```

功能：

提供一个方法对一个分等级的地址空间进行‘上’、‘下’、‘到’操作。

参数	描述
dwBrowseDirection	OPC_BROWSE_UP 或 OPC_BROWSE_DOWN 或 OPC_BROWSE_TO.
szString	‘下’，分支的名字。如 REACTOR10 ‘上’，此参数为空。 ‘到’，一个合适的名字（如通过 GetItemID 获得的名字），或者一个空的字符串（根目录下）。如 AREA1.REACTOR10.TIC1001

返回码

返回码	描述
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存。
E_INVALIDARG	操作非法。
S_OK	操作成功。

注释：

在 FLAT 地址空间下这个函数将返回 E_FAIL。

如果 szString 没有代表分支，一个错误会返回。

从根目录下进行‘上’操作会返回 E_FAIL。

IOPCBrowseServerAddressSpace:: BrowseOPCItemIDs

```
HRESULT BrowseOPCItemIDs(
    [in] OPCBROWSETYPE     dwBrowseFilterType,
    [in, string] LPCWSTR    szFilterCriteria,
    [in] VARTYPE      vtDataTypeFilter,
    [in] DWORD        dwAccessRightsFilter,
    [out] LPENUMSTRING    * ppIEnumString
    );
```

功能：

返回一个含有 ItemID 表的 IENUMString。浏览的地址可以经过 ChangeBrowsePosition 来设定。

参数	描述
dwBrowseFilterType	OPC_BRANCH –返回含有子分支的项。 OPC_LEAF – 返回不含有子分支的项。 OPC_FLAT – 返回所有的地址空间。
szFilterCriteria	过滤字符串，用途如同 SQL 中的 LIKE 功能，一个空的字符串代表没有过滤。
vtDataTypeFilter	过滤数据类型，VT_EMPTY 代表没有过滤。

dwAccessRightsFilter	过滤存取权限(OPC_READABLE 或 OPC_WRITEABLE)。如果此值为 0，则在过滤条件中忽视权限过滤。
ppIEnumString	存放返回的接口指针。如果 HRESULT 为 S_OK 或 S_FALSE，那么 ppIEnumString 为 NULL。

返回码

返回码	**描述**
S_OK	操作成功。
S_FALSE	枚举为空。一个空的枚举器必须要返回并且需要释放。
E_FAIL	操作失败。
E_OUTOFMEMORY	没有足够的内存空间。
E_INVALIDARG	非法的操作。
OPC_E_INVALIDFILTER	无效的过滤字符串。

注释：

返回的枚举器可能什么都没有，如果所有的 ITEMID 不符合过滤的条件。返回的字符串代表当前根下的 BRANCH 和 LEAF。服务器尽可能的返回那些可以通过 AddItems 添加的 ITEM 的字符串。然而有时服务器会返回一个暗示的字符串。例如一个 PLC 含有 32000 个寄存器，但是在浏览时并不返回 32000 个单独的字符串，而是返回一个字符串代表“0－31999”，在这种情况下客户需要自己通过手动添加 ITEMID 来测试服务器的地址空间。最新的 GE OPC SERVER（ETHERNET，VERSAMAX，90-30，90-70）就是这样的。

客户必须释放每个枚举器。

IOPCBrowseServerAddressSpace:: GetItemID

```
HRESULT GetItemID(
    [in] LPCWSTR    szItemDataID,
    [out, string] LPWSTR * szItemID
```

);

功能：

获取地址空间中的 ITEM ID。

参数	描述
szItemDataID	当前根下的 BRANCH 或 LEAF 的名字，或者一个空的字符串。一个空的字符串将返回当前位置下的名字。
szItemID	返回的 ItemID。

返回码

返回码	描述
E_FAIL	操作失败。
E_INVALIDARG	操作非法。
E_OUTOFMEMORY	没有足够的内存空间。
S_OK	操作成功。

IOPCBrowseServerAddressSpace:: BrowseAccessPaths

```
HRESULT BrowseAccessPaths(
    [in, string] LPCWSTR      szItemID,
    [out] LPENUMSTRING * ppIEnumString
    );
```

功能：

提供一个方法去浏览一个 ITEM ID 的可用 AccessPaths。

参数	描述
szItemID	合法的 ITEMID。
ppIEnumString	存放返回的字符串枚举器。如果 HRESULT 不是 S_OK 或 S_FALSE，返回 NULL。

返回码

Return Code	描述
E_FAIL	操作失败。

E_INVALIDARG	操作非法。
S_FALSE	枚举为空。一个空的枚举器必须要返回并且需要释放。
E_OUTOFMEMORY	没有足够的内存空间。
E_NOTIMPL	服务器不支持 access paths。
S_OK	操作成功。

在本书的服务器设计中，需要设计浏览类型为 FLAT 型。

在浏览地址空间功能的实现中，需要了解一个新的接口 IEnumString，此接口用于存放返回的字符串枚举器。IEnumString 是一个标准的 COM 接口，在 COM 中被定义为枚举字符串，LPWSTR 是表示宽字符，UNICODE 字符的数据类型，IEnumString 拥有同所有枚举接口一样的方法:Next，Skip，Reset，Clone。

然而 IEnumString 仅仅是一个接口，所以定义了一个基于 IEnumString 接口的类 IXXXEnumString，定义如下：

```
class IXXXEnumString : public IEnumString
{
 public:
    IXXXEnumString(LPUNKNOWN, ULONG, LPOLESTR*,
    IMalloc*);
     ~IXXXEnumString( void);
    // the IUnknown Functions
    STDMETHODIMP    QueryInterface( REFIID iid,
    LPVOID* ppInterface);
    STDMETHODIMP_(ULONG) AddRef( void);
    STDMETHODIMP_(ULONG) Release( void);
    // the IEnumString Functions
    STDMETHODIMP Next (
        ULONG celt,LPOLESTR *rgelt,ULONG *pceltFetched
        );
    STDMETHODIMP Skip (
        ULONG celt
        );
```

```
        STDMETHODIMP Reset(
            void
            );
        STDMETHODIMP Clone(
            IEnumString **ppenum
            );
      // 成员变量
    private:
      ULONG              m_cRef;       //对象引用计数
      LPUNKNOWN          m_pUnkRef;    //IUnknown
      ULONG              m_iCur;       //当前元素
      ULONG              m_cstr;       //字符串数目
      LPOLESTR           *m_prgstr;    //字符串的复制
      IMalloc          *m_pmem;  // 内存
};
```

如上所示，QueryInterface，AddRef，Release 为 IUnknown 方法，IUnknown 接口为所有 COM 接口的基础接口，即所有的 COM 接口均含有 IUnknown 的方法，而 Next，Skip，Reset，Clone 为 IEnumString 方法。

实现代码及解释如下。

```
IXXXEnumString::IXXXEnumString(LPUNKNOWN          pUnkRef,
ULONG cstr, LPOLESTR *prgstr, IMalloc * pmem)
{
    UINT        i;
    m_cRef = 0;
    m_pUnkRef = pUnkRef;
    m_iCur = 0;
    m_cstr = cstr;
    m_prgstr = new LPOLESTR[cstr];
    m_pmem = pmem;
    if (NULL != m_prgstr)
    {
        // 复制所有的字符串
        for (i=0; i < m_cstr; i++)
        {
            m_prgstr[i] = new WCHAR[wcslen(prgstr[i]) + 1];
```

```
                wcscpy(m_prgstr[i], prgstr[i]);
            }
        }
        return;
    }
```

在本构造函数中，第一个参数为 IUnknown 接口指针，第二个参数为字符串的数目，第三个参数为传递的字符串指针，第四个参数为内存指针，通过这四个参数复制所有字符串到 m_prgstr，并将引用计数 m_cRef 置 0，获得最大的字符串数目 m_cstr，获得内存指针 m_pmem。

```
    IXXXEnumString::~IXXXEnumString(void)
    {
        unsigned int i;
        if (NULL!=m_prgstr)
        {
            // 删除所有字符串的复制
            for (i=0; i < m_cstr; i++)
            {
                delete [] m_prgstr[i];
            }
            delete [] m_prgstr;
        }
        return;
    }
```

在析构函数中，如果字符串 m_prgstr 不为空，则删除所有的字符串。

```
    STDMETHODIMP IXXXEnumString::QueryInterface(REFIID riid
        , LPVOID *ppv)
    {
        *ppv=NULL;

        if (IID_IUnknown==riid || IID_IEnumString==riid)
            *ppv=(LPVOID)this;

        if (NULL!=*ppv)
        {
            ((LPUNKNOWN)*ppv)->AddRef();
```

```
            return S_OK;
        }

        return E_NOINTERFACE;
    }
```

在 QueryInterface 函数中判断 riid 是否为 IID_Iunknown 或者 IID_IEnumString，如果是则将 this 指针赋予*ppv 参数。如果*ppv 不为空，则增加引用计数，并返回 S_OK；如果为空，则不增加引用计数并回 E_NOINTERFACE。

```
    STDMETHODIMP_(ULONG) IXXXEnumString::AddRef(void)
    {
        ++m_cRef;
        if(m_pUnkRef != NULL)
            m_pUnkRef->AddRef();
        return m_cRef;
    }
```

AddRef 用来增加引用计数。

```
    STDMETHODIMP_(ULONG) IXXXEnumString::Release(void)
    {
        if(m_pUnkRef != NULL)
            m_pUnkRef->Release();

        if (0L!=--m_cRef)
            return m_cRef;

        delete this;
        return 0;
    }
```

Release 用来减少引用计数。

```
    STDMETHODIMP IXXXEnumString::Next(ULONG cstr, LPOLESTR
    *pstr, ULONG *pulstr)
    {
        ULONG      cReturn = 0L;
        ULONG      maxcount = cstr;
```

```
        *pulstr = 0L;// 初始化为 0
        *pstr = NULL;   // 初始化为 NULL,
        // 如果枚举器为空，返回失败信息
        if (NULL == m_prgstr)
            return S_FALSE;
        // If user passed null for count of items returned
        // Then he is only allowed to ask for 1 item
        //
        if (NULL == pulstr)
        {
            if (1L != cstr)
                return E_POINTER;// E_POINTER 代表非法指针。
        }

        // If we are at end of list return FALSE
        //
        if (m_iCur >= m_cstr)
            return S_FALSE;

        // Return as many as we have left in list up to request count
        //
        while (m_iCur < m_cstr && cstr > 0)
        {
            int   size;

            // Compute WCHARs in string (will be at least 1 for nul).
            //
            size = (wcslen(m_prgstr[m_iCur])+1);

            pstr[cReturn] = (WCHAR*)m_pmem->Alloc(size *
sizeof(WCHAR));
            if(pstr[cReturn])
            {
                wcscpy(pstr[cReturn], m_prgstr[m_iCur]);
            }

            // And move on to the next one
```

```
            //
            m_iCur++;
            cReturn++;
            cstr--;
        }

        if (NULL != pulstr)
            *pulstr = cReturn;

        if (cReturn == maxcount) return S_OK;
        return S_FALSE;
}
```

Next 用来返回枚举中的下一个元素。

```
STDMETHODIMP IXXXEnumString::Skip(ULONG cSkip)
{
    if (((m_iCur+cSkip) >= m_cstr) || NULL==m_prgstr)
        return S_FALSE;

    m_iCur+=cSkip;
    return S_OK;
}
```

Skip 用来跳过枚举中的下 N 个元素。

```
STDMETHODIMP IXXXEnumString::Reset(void)
{
    m_iCur=0;
    return S_OK;
}
```

Reset 用来设置枚举的当前查询索引为 0。

```
STDMETHODIMP IXXXEnumString::Clone(LPENUMSTRING
*ppEnum)
{
    IXXXEnumString     *pNew;

    *ppEnum=NULL;

    //Create the clone
```

```
        //
        pNew=new IXXXEnumString(m_pUnkRef, m_cstr, m_prgstr,
        m_pmem);
        if (NULL==pNew)
            return E_OUTOFMEMORY;
        pNew->AddRef();
        // Set the 'state' of the clone to match the state if this
        //
        pNew->m_iCur=m_iCur;
        *ppEnum=pNew;
        return S_OK;
    }
```

Clone 用来复制一个于当前枚举状态一样的枚举。

3.9 OPC 服务器的注册

根据 OPC 服务器支持的规范的版本，OPC 服务器在注册表中的信息也不一样。

OPC 基金会制定：

OPC DA1.0 规范 {63D5F430-CFE4-11D1-B2C8-0060083BA1FB}

OPC DA2.0 规范 {63D5F432-CFE4-11D1-B2C8-0060083BA1FB}

OPC DA3.0 规范 {CC603642-66D7-48F1-B69A-B625E73652D7}

在开发 OPC DA 服务器时需要将相应的类别信息注册到系统中。

本章的示例程序的注册如下：

[HKEY_CLASSES_ROOT\CLSID\{7C13259A-74FD-4064-818F-C639E4B5811B}\Implemented Categories\

{63D5F432-CFE4-11D1-B2C8-0060083BA1FB}]

图 3.17CLSID 下的注册信息

如图 3.17 所示。详见源程序下的注册.reg 文件。

iFIX3.5 作为 OPC 服务器（Intellution.OPCiFIX），其 OPC 类别注册信息如图 3.18 所示。

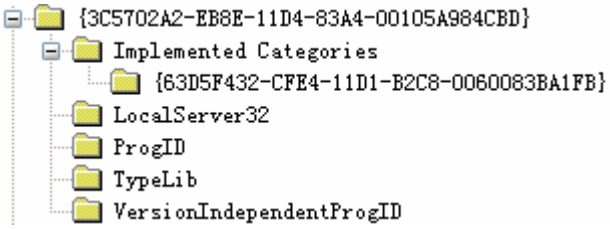


图 3.18iFIX3.5CLSID 下的注册信息。

重点：何为包容？何为聚会？OPC 服务器的结构？OPC 服务器，组，项的管理。

第 4 章 OPC 客户程序实例

****

关键字：同步 异步 VC VB

4.1 OPC 客户程序开发环境

无论开发者还是最终使用者都必须安装 OPC 代理/存根(Proxy/Stub)DLL，并进行环境设置。这些文件（opc_aeps.dll，opccomn_ps.dll，opchda_ps.dll，opcproxy.dll，aprxdist.exe，opcenum.exe）可以从 OPC 基金会网站下载，也可以在 http://www.opc-china.com 下载。所有文件必须安装在客户端机器和服务器端机器上。

安装步骤如下：

1．复制所有的文件到你的 Windows 系统路径，如：

copy opcproxy.dll c:\winnt\system32

copy opccomn_ps.dll c:\winnt\system32

copy opc_aeps.dll c:\winnt\system32

copy opchda_ps.dll c:\winnt\system32

copy aprxdist.exe c:\winnt\system32

copy opcenum.exe c:\winnt\system32

2．安装代理 DLL。

REGSVR32 opcproxy.dll

REGSVR32 opccomn_ps.dll

REGSVR32 opc_aeps.dll

REGSVR32 opchda_ps.dll

3．如果 aprxdist.dll 不存在，可以运行 aprxdist.exe 生成 aprxdist.dll。

4．安装 opcenum.exe

opcenum /regserver

在本书的 VB 客户程序中引用了 OPC Automation2.0 库，库文件为 opcdaauto.dll。可以在本书附送的源码中找到，注册方式为：REGSVR32 opcdaauto.dll。

4.2 OPC 客户程序(VC++同步篇)

OPC 客户程序访问 OPC 服务器，实际上就是一个典型的客户访问进程外组件的过程。OPC 客户程序与 OPC 服务器的协作过程(同步)如下所示。

OPC 客户程序	COM 库	OPC 服务器（EXE）
CLSID clsid;		
IclassFactory *pcf;		
IUnknown* pUnknown;		
CoInitialize(NULL);		
CLSIDFromProgID(“”,&clsid)		
	COM 库在注册表中查找 OPC 服务器的 CLSID	
CoGetClassObject(clsid, CLSCTX_LOCAL_SERVER , NULL, IID_IclassFactory, (void**)&pCF);		
	COM 库在内存中查找 clsid	

	If(OPC 服务器还没有启动
	||COM 需要另外一个实例)
	{
	从注册表中获取 OPC 服
	务器名创建 OPC 服
	务器
	进程
	}
	等待	

		调用 CoInitialize 创建 OPC 服务器支持的各种类 厂对象 调用 COM 函数注册所有类厂对 象：CoRegisterClassObject
	OPC 服务器注册完成后， COM 库把 IclassFactory 接口 返回给 OPC 客户	
利用类工厂创建 OPC Server 对象 PCF->CreateInstance(NULL, IID_IUnknown, (void**)&pUnknown);		
		类工厂对象的

	CreateInstance 函数被调用（通过进程中代理对象被间接调用）
	构造 OPC 服务器对象
	返回 Iunknown 接口指针
指向 OPC 服务器对象的 IUnknown 接口指针	
IUnknown * pOPC	
通过 IOPCServer 接口，添加组对象	
POPC->AddGroup(L"MyGroup1",	IOPCServer 接口的成员函数
TRUE, 0, 0, 0, &DeadBand1,0,	AddGroup()被调用
&hServerGroup1,	
&RevisedRate1,IID_IUnknown,	
(LPUNKNOWN*)&pGRP1U);	
通过 IOPCItemMgt 接口添加 item	
IOPCItemMgt * pIM	IOPCItemMgt 接口的成员函数
PIM->AddItems(1000, id, &ir,	AddItems()被调用
&ih);	
通过 IOPCSyncIO 接口实现数据访问	
IOPCSyncIO *pSIO	
//写操作	IOPCSyncIO 接口的成员函

		数
PSIO->Write(1000, sh, v, &hr)		Write()被调用
//读操作		IOPCSyncIO 接口的成员函数
PSIO->Read(OPC_DS_CACHE, 1000, sh, &is, &hr);		Read()被调用
操作完成，释放接口		服务器对象 Release 函数被调用
PSIO->Release();		If(m_Ref==0)
PIM->Release();		{delete this;return 0;}
POPC->Release();		如果满足条件，则正常退出
CoFreeUnusedLibrary()		
	COM 库对所有该客户没有成功释放的对象调用 Release 函数	
		OPC 服务器程序退出
	COM 库释放资源	
OPC 客户程序退出		

几个函数介绍如下：

在 COM 库中，有三个 API 函数可以用于对象的创建，它们分别是 CoGetClassObject、CoCreateInstnace 和 CoCreateInstanceEx。通常情况下，客户程序调用其中之一完成对象的创建，并返回对象的初始接口指针。COM 库与类厂也通过这三个函数进行交互。

HRESULT CoGetClassObject(const CLSID& clsid, DWORD dwClsContext,COSERVERINFO *pServerInfo, const IID& iid, (void

**)ppv);

CoGetClassObject 函数先找到由 clsid 指定的 COM 类的类厂，然后连接到类厂对象，如果需要的话，CoGetClassObject 函数装入组件代码。如果是进程内组件对象，则 CoGetClassObject 调用 DLL 模块的 DllGetClassObject 引出函数，把参数 clsid、iid 和 ppv 传给 DllGetClassObject 函数，并返回类厂对象的接口指针。通常情况下 iid 为 IClassFactory 的标识符 IID_IClassFactory。如果类厂对象还支持其它可用于创建操作的接口，也可以使用其它的接口标识符。例如，可请求 IClassFactory2 接口，以便在创建时，验证用户的许可证情况。IClassFactory2 接口是对 IClassFactory 的扩展，它加强了组件创建的安全性。

参数 dwClsContext 指定组件类别，可以指定为进程内组件、进程外组件或者进程内控制对象(类似于进程外组件的代理对象，主要用于 OLE 技术)。参数 iid 和 ppv 分别对应于 DllGetClassObject 的参数，用于指定接口 IID 和存放类对象的接口指针。参数 pServerInfo 用于创建远程对象时指定服务器信息，在创建进程内组件对象或者本地进程外组件时，设置 NULL。

如果 CoGetClassObject 函数创建的类厂对象位于进程外组件，则情形要复杂得多。首先 CoGetClassObject 函数启动组件进程，然后一直等待，直到组件进程把它支持的 COM 类对象的类厂注册到 COM 中。于是 CoGetClassObject 函数把 COM 中相应的类厂信息返回。因此，组件外进程被 COM 库启动时(带命令行参数“/Embedding”)，它必须把所支持的 COM 类的类厂对象通过 CoRegisterClassObject 函数注册到 COM 中，以便 COM 库创建 COM 对象使用。当进程退出时，必须调用 CoRevokeClassObject 函数以便通知 COM 它所注册的类厂对象不

再有效。组件程序调用 CoRegisterClassObject 函数和 CoRevokeClassObject 函数必须配对，以保证 COM 信息的一致性。

HRESULT CoCreateInstance(const CLSID& clsid, IUnknown *pUnknownOuter,

DWORD dwClsContext, const IID& iid, (void **)ppv);

CoCreateInstance 是一个被包装过的辅助函数，在它的内部实际调用 CoGetClassObject 函数。CoCreateInstance 的参数 clsid 和 dwClsContext 的含义与 CoGetClassObject 相应的参数一致，(CoCreateInstance 的 iid 和 ppv 参数与 CoGetClassObject 不同，一个是表示对象的接口信息，一个是表示类厂的接口信息)。参数 pUnknownOuter 与类厂接口的 CreateInstance 中对应的参数一致，主要用于对象被聚合的情况。CoCreateInstance 函数把通过类厂创建对象的过程封装起来，客户程序只要指定对象类的 CLSID 和待输出的接口指针及接口 ID，客户程序可以不与类厂打交道。

CoCreateInstance 可以用下面的代码实现：

```
HRESULT CoCreateInstance(const CLSID& clsid, IUnknown
*pUnknownOuter, DWORD dwClsContext, const IID& iid, void *ppv)
{
    IClassFactory *pCF;
    HRESULT hr;
    hr = CoGetClassObject(clsid, dwClsContext, NULL,
    IID_IClassFactory, (void *) pCF);
    if (FAILED(hr)) return hr;
    hr = pCF->CreateInstance(pUnknownOuter, iid, (void *)ppv);
    pFC->Release();
```

```
    return hr;
}
```

从这段代码中可以看出，CoCreateInstance 函数在最开始的时候利用了 CoGetClassObject 函数创建类厂对象，然后用得到的类厂对象的接口指针创建真正的 COM 对象，最后把类厂对象释放掉并返回，这样就把类厂屏蔽起来。

但是，用 CoCreateInstance 并不能创建远程机器上的对象，因为在调用 CoGetClassObject 时，把第三个用于指定服务器信息的参数设置为 NULL。如果要创建远程对象，可以使用 CoCreateInstance 的扩展函数 CoCreateInstanceEx，CoCreateInstanceEx 用于远程组件的创建，我们在下一章会做介绍。

调用 COM 库的函数之前，为了使函数有效，必须调用 COM 库的初始化函数：

```
HRESULT CoInitialize(IMalloc *pMalloc);
```

pMalloc 用于指定一个内存分配器，可由应用程序指定内存分配原则。一般情况下，我们直接把参数设为 NULL，则 COM 库将使用缺省提供的内存分配器。

返回值：S_OK 表示初始化成功

返回值：S_FALSE 表示初始化成功，但这次调用不是本进程中首次调用初始化函数

返回值：S_UNEXPECTED 表示初始化过程中发生了错误，应用程序不能使用 COM 库

通常，一个进程对 COM 库只进行一次初始化，而且，在同一个模块单元中对 COM 库进行多次初始化并没有意义。

COM 程序在用完 COM 库服务之后，通常是在程序退出之前，一

定要调用终止 COM 库服务函数，以便释放 COM 库所维护的资源：

void CoUninitialize(void);

注意：凡是调用 CoInitialize 函数返回 S_OK 的进程或程序模块一定要有对应的 CoUninitialize 函数调用，以保证 COM 库有效地利用资源。

下面本章给出 VC++客户程序实现过程。

在 VC++6.0 中新建工程，名称为 SynOpc，选择 MFC AppWizard 如图 4.1 所示。点击 OK。

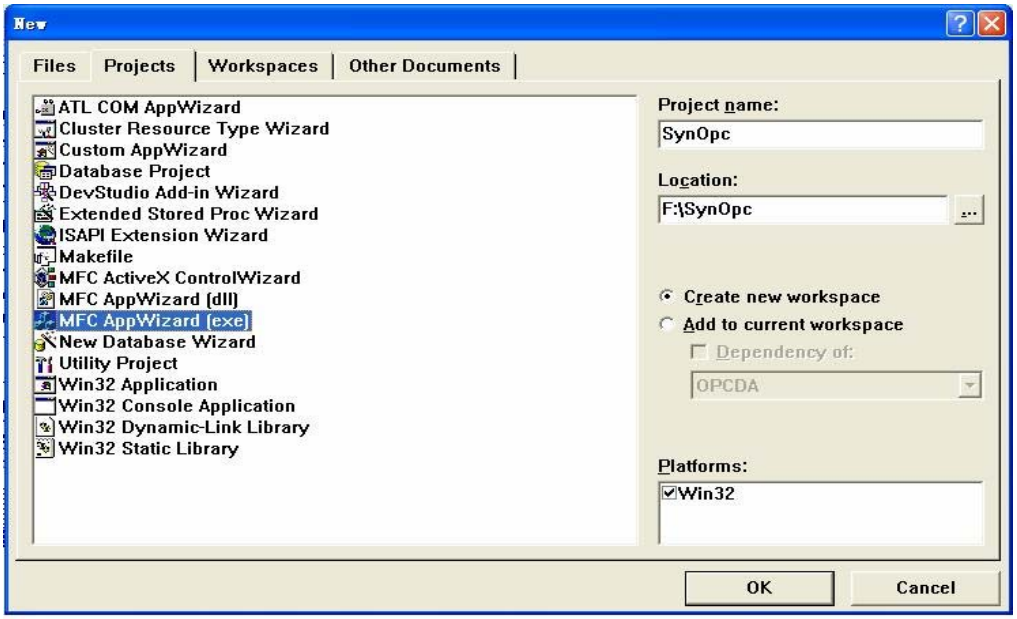


图 4.1

选择为 Dialog based，点击 Finish。在随后出来的对话框中点击 OK，完成创建。

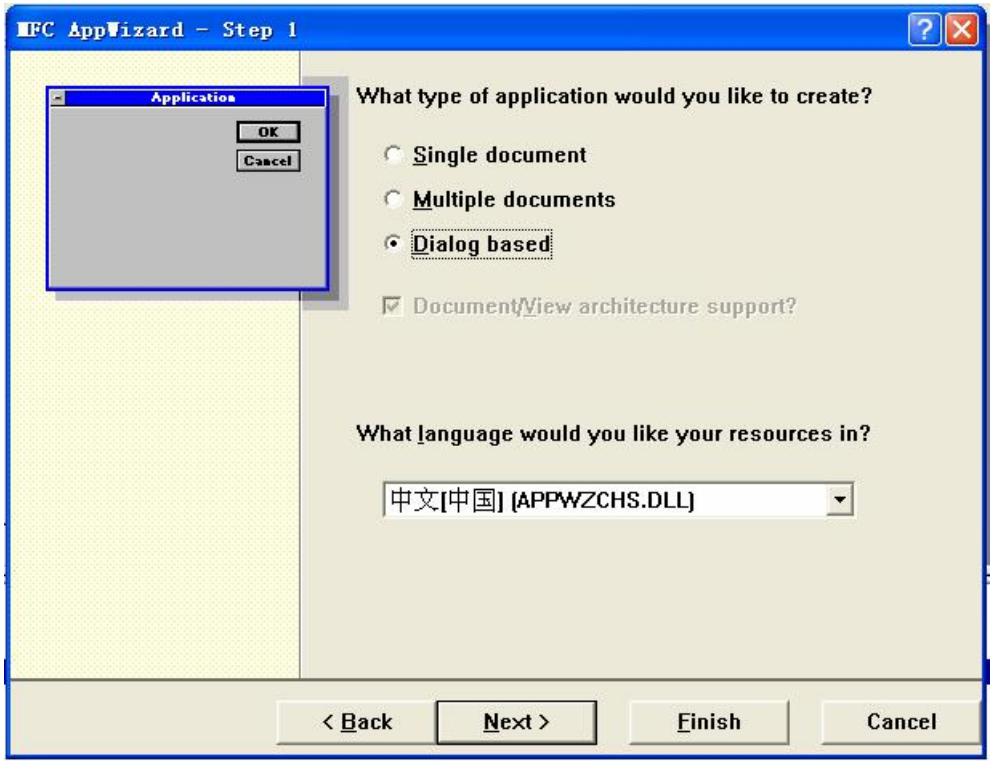


图 4.2

将 OPCDA.IDL 生成的两个文件 opcda.h，opcda_i.c 拷入工程所在的文件夹,，将 OpcError.h 拷入工程所在的文件夹。在 SynOpcDlg.h 文件的开始处加入

```
#include "OpcError.h"
#include "opcda.h"
#define LOCALE_ID       0x409   // Code 0x409 = ENGLISH
//// ItemID used in this sample
const LPWSTR   szItemID = L"XXX";
```

在 class CsynOpcDlg 类定义处添加保护型变量如下

```
IOPCServer       *m_IOPCServer;
IOPCItemMgt          *m_IOPCItemMgt;
IOPCSyncIO           *m_IOPCSyncIO;
OPCITEMDEF           m_Items[1];
OPCITEMRESULT   *m_ItemResult;
OPCHANDLE            m_GrpSrvHandle;
HRESULT              *m_pErrors;
```

在 SynOpc.cpp 文件处添加如下代码

```
#include "opcda.h"
#include "opcda_i.c"
```

在对话框上添加控件如下：IDC_START，IDC_READ，IDC_WRITE，IDC_STOP 四个按纽控件。IDC_READVALUE，IDC_QUALITY，IDC_TIMESTAMP，IDC_WRITEVALUE，IDC_WRITERESULT 五个 TEXTBOX 控件。

图 4.3OPC 同步例子界面（VC）

通过双击四个按纽定义如下函数

```
/*启动 OPC 服务器，添加组对象，添加项*/
void CSynOpcDlg::OnStart()
{
    // TODO: Add your control notification handler code here
}
/*同步读，将结果显示在对话框*/
void CSynOpcDlg::OnRead()
{
```

```
    // TODO: Add your control notification handler code here
}
/*同步写，将写结果显示在对话框*/
void CSynOpcDlg::OnWrite()
{
    // TODO: Add your control notification handler code here
}
/*停止服务器*/
void CSynOpcDlg::OnStop()
{
    // TODO: Add your control notification handler code here
}
```

对五个 TEXTBOX 定义变量如下图所示。

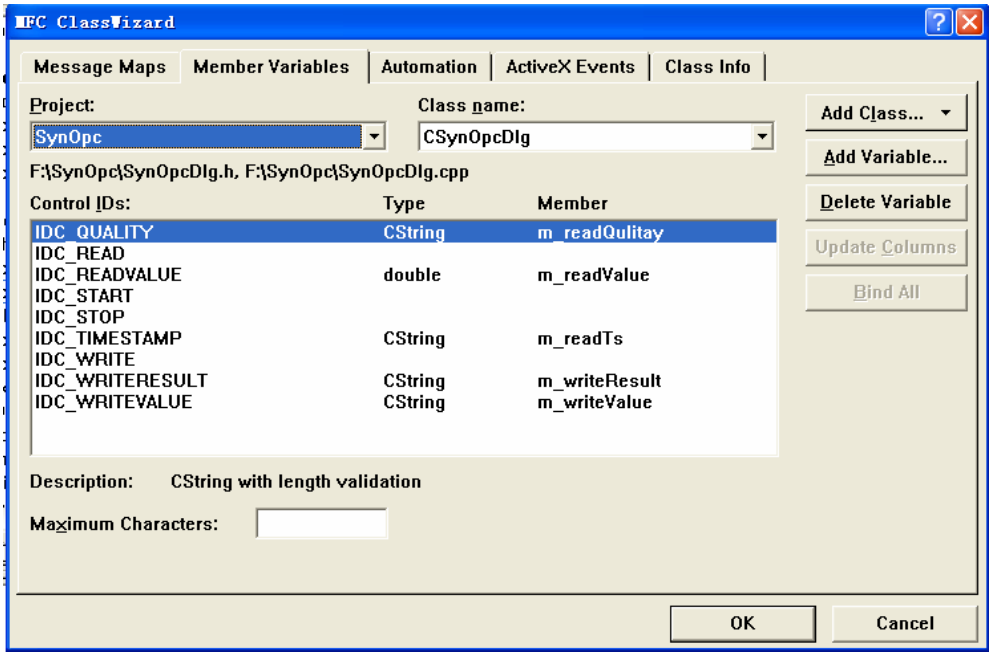


图 4.4

根据前面的 OPC 客户程序与 OPC 服务器的协作过程(同步)，我们在 CSynOpcDlg::OnStart()，OnRead()，OnWrite()，OnStop()添加代码如下：

```
/*启动 OPC 服务器，添加组对象，添加项*/
void CSynOpcDlg::OnStart()
{
    // TODO: Add your control notification handler code here
```

```
    HRESULT         r1;
    CLSID       clsid;
    LONG        TimeBias = 0;
    FLOAT       PercentDeadband = 0.0;
    DWORD           RevisedUpdateRate;
    CString     szErrorText;
    m_ItemResult = NULL;
    //   初始化 COM 库
    r1 = CoInitialize(NULL);
    if (r1 != S_OK)
    {
        if (r1 == S_FALSE)
        {
        AfxMessageBox("COM 库已经初始化");
    }
    else
    {
        AfxMessageBox("COM 库初始化失败");
        return;
    }
}
//   通过 ProgID,查找注册表中的相关 CLSID
r1 = CLSIDFromProgID(L"Intellution.OPCiFIX", &clsid);
if (r1 != S_OK)
{
    AfxMessageBox("获取 CLSID 失败");
    CoUninitialize();
    return;
}
//创建 OPC 服务器对象，并查询对象的 IID_IOPCServer 接口
r1 = CoCreateInstance (clsid, NULL,
CLSCTX_LOCAL_SERVER ,IID_IOPCServer,
(void**)&m_IOPCServer);
if (r1 != S_OK)
{
    AfxMessageBox("创建 OPC 服务器对象失败");
    m_IOPCServer = NULL;
```

```
        CoUninitialize();
        return;
    }
    //添加一个 group 对象，并查询 IOPCItemMgt 接口
    r1=m_IOPCServer->AddGroup(L"grp1",  //  [in]组名字
    TRUE,                   //  [in]是否活动状态
    500,                    //  [in] 刷新率（毫秒）
    1,                      //  [in] 客户句柄
    &TimeBias,              //  [in]
    &PercentDeadband,       //  [in] 死区参数
    LOCALE_ID,              //  [in]语言
    &m_GrpSrvHandle,        //  [out] 服务器句柄
    &RevisedUpdateRate,     //  [out] 服务器返回的刷新率
    IID_IOPCItemMgt,        //  [in] 需要的接口指针
    (LPUNKNOWN*)&m_IOPCItemMgt);   //  [out] 返回的需要的接口指针
    if (r1 == OPC_S_UNSUPPORTEDRATE)
    {
        AfxMessageBox("请求的刷新速率与实际的刷新速率不一致");
    }
    else
        if (FAILED(r1))
        {
            AfxMessageBox("不能为服务器添加 group 对象");
            m_IOPCServer->Release();
            m_IOPCServer = NULL;
            CoUninitialize();
            return;
        }
    //  为 AddItem 定义 item 表的参数
    m_Items[0].szAccessPath  = L"";          //  不需要 Accesspath
    m_Items[0].szItemID      = szItemID;     //  ItemID
    m_Items[0].bActive       = TRUE;
    m_Items[0].hClient       = 1;
    m_Items[0].dwBlobSize     = 0;
    m_Items[0].pBlob          = NULL;
    m_Items[0].vtRequestedDataType = 0;      //  数据类型
```

```
r1 = m_IOPCItemMgt->AddItems(1,        //    [in] 添加 1 个 item
m_Items,    //    [in] 前面定义
&m_ItemResult,//[out] 结果信息指针
&m_pErrors);//  [out] 错误码
//    为没有添加成功的 item 返回信息
if ( (r1 != S_OK) && (r1 != S_FALSE) )
{
    AfxMessageBox("AddItems 失败");
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = NULL;
    m_GrpSrvHandle = NULL;
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    CoUninitialize();
    return;
}
else if(r1==S_OK)
{
    AfxMessageBox("AddItems()成功");
}
//    检测 Item 的可读写性
if (m_ItemResult[0].dwAccessRights != (OPC_READABLE +
OPC_WRITEABLE))
{
    AfxMessageBox("Item 不可读，也不可写,请检查服务器配置");
}
//查询 group 对象的同步接口
r1 = m_IOPCItemMgt->QueryInterface(IID_IOPCSyncIO,
(void**)&m_IOPCSyncIO);
if (r1 < 0)
{
    AfxMessageBox("IOPCSyncIO 没有发现，错误的查询!");
    CoTaskMemFree(m_ItemResult);
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = NULL;
    m_GrpSrvHandle = NULL;
    m_IOPCServer->Release();
```

```
        m_IOPCServer = NULL;
        CoUninitialize();
        return;
    }
}
/*同步读，将结果显示在对话框*/
void CSynOpcDlg::OnRead()
{
    // TODO: Add your control notification handler code here
    OPCHANDLE       *phServer;
    OPCITEMSTATE    *pItemValue;
    HRESULT         *pErrors;
    HRESULT         r1;
    UINT            qnr;
    if (m_pErrors[0] != S_OK)       //Item 不可用
    {
        AfxMessageBox("OPC Item  不可用，不能用同步读功能!");
        return;
    }
    //内存分配
    phServer = new OPCHANDLE[1];
    //通过 server 句柄选择 item（由 AddItem 得到的）
    phServer[0] = m_ItemResult[0].hServer;
    r1 = m_IOPCSyncIO->Read(OPC_DS_DEVICE, 1, phServer,
    &pItemValue, &pErrors);
    //释放内存
    delete [] phServer;
    if (r1 == S_OK) {
        m_readValue = pItemValue[0].vDataValue.fltVal;
        qnr = pItemValue[0].wQuality;
        switch(qnr)
        {
        case OPC_QUALITY_BAD:
            m_readQulitay = "BAD";
            break;
        case OPC_QUALITY_UNCERTAIN:
            m_readQulitay= "UNCERTAIN";
```

```
        break;
    case OPC_QUALITY_GOOD:
         m_readQulitay = "GOOD";
        break;
    case OPC_QUALITY_NOT_CONNECTED:
         m_readQulitay = "NOT_CONNECTED";
         break;
    case OPC_QUALITY_DEVICE_FAILURE:
         m_readQulitay = "DEVICE_FAILURE";
         break;
    case OPC_QUALITY_SENSOR_FAILURE:
         m_readQulitay = "SENSOR_FAILURE";
        break;
    case OPC_QUALITY_LAST_KNOWN:
         m_readQulitay = "LAST_KNOWN";
        break;
    case OPC_QUALITY_COMM_FAILURE:
         m_readQulitay = "COMM_FAILURE";
        break;
    case OPC_QUALITY_OUT_OF_SERVICE:
         m_readQulitay = "OUT_OF_SERVICE";
        break;
    case OPC_QUALITY_LAST_USABLE:
         m_readQulitay = "LAST_USABLE";
        break;
    case OPC_QUALITY_SENSOR_CAL:
         m_readQulitay = "SENSOR_CAL";
        break;
    case OPC_QUALITY_EGU_EXCEEDED:
        m_readQulitay = "EGU_EXCEEDED";
        break;
    case OPC_QUALITY_SUB_NORMAL:
         m_readQulitay = "SUB_NORMAL";
        break;
    case OPC_QUALITY_LOCAL_OVERRIDE:
         m_readQulitay = "LOCAL_OVERRIDE";
        break;
```

```
        default:
            m_readQulitay = "UNKNOWN ERROR";
        }
        m_readTs = COleDateTime( pItemValue[0].ftTimeStamp ).Format();
        UpdateData(FALSE);
        }
        if (r1 == S_FALSE)
        {
            AfxMessageBox("Read()错误");
        }
        if (FAILED(r1))
        {
            AfxMessageBox("同步读失败!");
        }
        else
        {
            //释放内存，如果操作成功
            CoTaskMemFree(pErrors);
            CoTaskMemFree(pItemValue);
        }
}
/*同步写，将写结果显示在对话框*/
void CSynOpcDlg::OnWrite()
{
    // TODO: Add your control notification handler code here
    OPCHANDLE       *phServer;
    HRESULT         *pErrors;
    VARIANT         values[1];
    HRESULT         r1;
    LPWSTR          ErrorStr;
    CString         szOut;

    if (m_pErrors[0] != S_OK)          //Item not available
    {
        AfxMessageBox("OPC Item 不可用，不能用同步读功能!");
        return;
    }
```

```
    //通过 server 句柄选择 item（由 AddItem 得到的）
    phServer = new OPCHANDLE[1];
    phServer[0] = m_ItemResult[0].hServer;

    //   从对话框获得数据
    UpdateData(TRUE);

    //   设置 Variant 变量的数据类型和数值
    values[0].vt = VT_R4;
    values[0].fltVal = m_writeValue;
    r1 = m_IOPCSyncIO->Write(1, phServer, values, &pErrors);
    delete[] phServer;

    if (FAILED(r1))
    {
        AfxMessageBox("同步写 Item 错误");
    }
    else
    {
        m_IOPCServer->GetErrorString(pErrors[0], LOCALE_ID,
        &ErrorStr);
        m_writeResult = ErrorStr;
        m_writeResult.Remove('\r');
        m_writeResult.Remove('\n');
        UpdateData(FALSE);
        CoTaskMemFree(pErrors);
    }
}
/*停止服务器*/
void CSynOpcDlg::OnStop()
{
    // TODO: Add your control notification handler code here
    HRESULT         r1;
    OPCHANDLE   *phServer;
    HRESULT         *pErrors;
```

```
//    删除 Item
phServer = new OPCHANDLE[1];
phServer[0] = m_ItemResult[0].hServer;
r1 = m_IOPCItemMgt->RemoveItems(1, // [in]  删除 1 个 item
phServer,     //    [in]  服务器句柄
&pErrors);// [out]  服务器返回的错误码
if ( (r1 != S_OK) && (r1 != S_FALSE) )
{
    AfxMessageBox("RemoveItems 失败!");
}
else if(r1==S_OK)
{
    AfxMessageBox("RemoveItems()成功");
}
delete[] phServer;
CoTaskMemFree(pErrors);
CoTaskMemFree(m_ItemResult);
m_ItemResult=NULL;
CoTaskMemFree(m_pErrors);
m_pErrors = NULL;
//    释放同步接口
m_IOPCSyncIO->Release();
m_IOPCSyncIO = NULL;
//    释放 item 管理接口
m_IOPCItemMgt->Release();
m_IOPCItemMgt = NULL;
//    删除 group 对象
r1=m_IOPCServer->RemoveGroup(m_GrpSrvHandle, TRUE);
if (r1 != S_OK)
{
    AfxMessageBox("RemoveGroup 失败!");
}
else
{
    AfxMessageBox("RemoveGroup 成功!");
}
m_GrpSrvHandle = NULL;
```

```
    //   释放 OPC 服务器
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    //关闭 COM 库
    CoUninitialize();
}
```

本例中以 IFIX3.5 为 OPC 服务器，并在 IFIX3.5 的数据库中建立 tag 名称为”XXX”的模拟量点，ItemID 为 L"FIX.XXX.F_CV"，并且此点为可读可写。在本例中大家需要注意的是所有的接口返回的指针都需要释放，否则会造成内存溢出。一般而言 CoTaskMemFree，XXX->Release()，CoUninitialize()都需要在程序结束时调用。获得指针的 Release 函数一定要注意必须调用，用来释放服务器的引用计数。

归结起来，VC++环境下访问 OPC 服务器同步读写的步骤如下：

1．初始化 COM 库，CoInitialize。

2．通过 OPC 服务器的 ProgID 来查询 CLISD，CLSIDFromProgID。

3．创建 OPC 服务器对象，并查询对象的 IID_IOPCServer 接口，CoCreateInstance。

4．添加一个 group 对象，并查询 IOPCItemMgt 接口，AddGroup。

5．为 group 对象添加 item，AddItems。

6．同步读，Read。

7．同步写，Write。

8．程序退出时或者停止服务器时，依次删除 item(RemoveItems)，删除 group(RemoveGroup)，释放资源。

同步读写结果如下图所示

图 4.5 OPC 同步例子操作结果

4.3 OPC 客户程序(VC++异步篇)

OPC 客户程序与 OPC 服务器的协作过程(异步)如下所示。

OPC 客户程序	OPC 服务器程序
接口定义	利用嵌套类定义连接点对象
IConnectionPointerContainer* pCPC;	IXXXConncetionPointContainer:
IOPCAsyncIO2* pASIO2;	IConncetionPointContainer{… …};
IOPCDataCallBack* pDCB;	IXXXConncetionPointContainer* myCPC;
IConnectionPoint* pCP;	
定义接收器接口	
IOPCDataCallBack* myDCB;	
查询 OPC 服务器是否支持可连接点，	

对象如果调用 失败，则说明 OPC 服务器不支持出接口	
Punk->QueryInterface(IID_IConnectionPointContainer, &pCPC);	IUnknown 接口的 QueryInterface 函数被调用
查询 OPC 服务器是否支持接收器对象 PCPC->FindConnectionPoint(IID_IOPCDataCallBack, &pCP);	IConnectiontPointContainer 接口的成员函数 FindConnectionPoint 被调用
如果支持接收器对象，在 OPC 客户/服务器之间建立 连接，并返回连接点对象生成的唯一标志此连接的 32 位整数 m_dwcookie pCP->Advise(&pDCB,&m_dwcookie);	IConnectionPoint 的成员函数 Advise 被调用 生成的唯一标志此连接的 32 位整数 dwcookie 发送给客户
设置刷新速率 IOPCGroupStateMgt* pGSP; PGRP->SetState(&RequestedRate, &Rate, &Active, 0, 0,0,0);	IOPCGroupStateMgt 的成员函数 SetState 被调用
异步读数据，传递事务 ID 给 OnReadComplete 函数 pASIO2->Read(1000,&m_phserver,m_dwtranscationID,	

&m_pdwcancelID,&hr);	OPC 服务器按指定的刷新速率从数据源读取数据，如果数据发生变化，则调用接收器接口 成员函数 OnDataChange 将数据送给客户 MyCPC->IOPCDataCallBack->OnDataChange (0,m_hgrGroup, m_hrmasterquality, m_hrmastererror,1000, &m_phClientItems, &m_ pvValues, &m_ pwQualities, &m_pftTimeStamps, &hr); 数据读取完成，OPC 服务器调用接收器接口成员函数 OnReadComplete 通知客户读数据完 成 MyCPC->IOPCDataCallBack-> OnReadComplete (m_dwtranscationID,m_hgrGroup, m_hrmasterquality,m_hrmastererror, 1000, &m_phClientItems, &m_pvValues, &m_ pwQualities, &m_pftTimeStamps, &hr);
异步写数据，传递事务 ID 给 OnWriteComplete 函数 pASIO2->Write(1000,&m_phserver,&m_pitemvalues m_dwtransactionID,m_pdwcancelID,&hr);	OPC 服务器调用接收器接口成员函数 OnWriteComplete 通知客户写数据完成

	MyCPC->IOPCDataCallBack->OnWriteComplete(m_dwtransactionID, m_hgroup,m_hrmastererror,m_dwcount, &m_phclientitems,&hr);
异步读写结束，调用 m_dwcookie,取消连接 PCP->Unadvise(m_dwcookie);	

释放接口
pASIO2->Release();
PGRP->Release();
pCP-> Release();
PCPC-> Release();
	IUnknown 接口的 Release 函数被调用

……	……

简单的示意图如下：

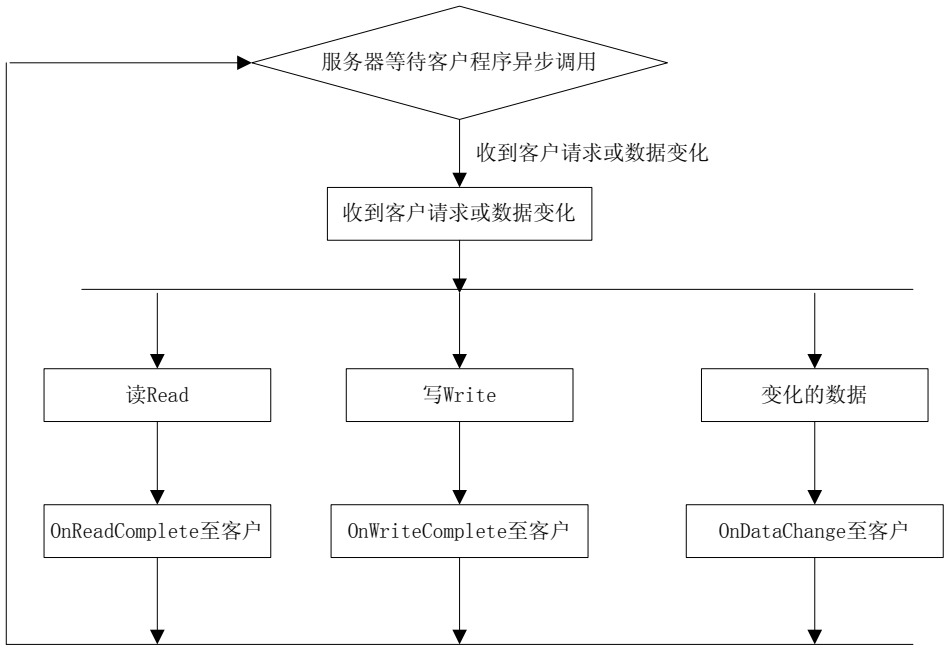
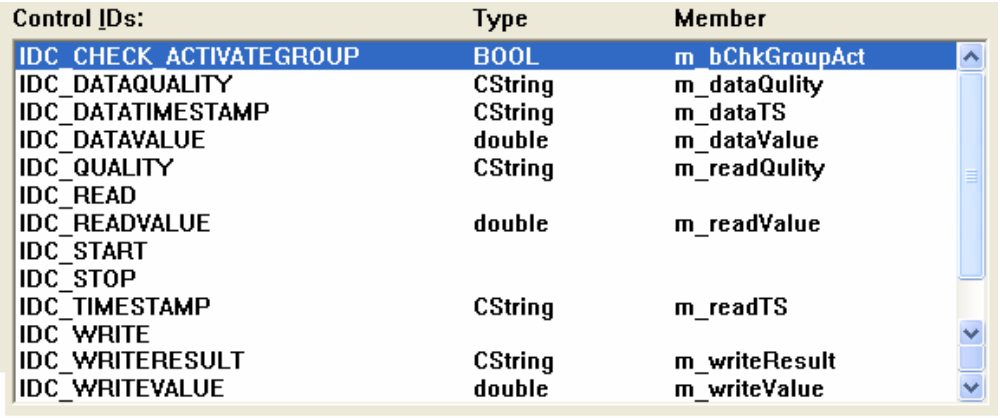


图 4.6OPC 异步通讯结构示意图

在 VC++6.0 下新建工程，命名为 AsynOpc,步骤如同步示例程序。对话框如下图所示。

图 4.7

成员变量如下所示。



Control IDs:	Type	Member
IDC_CHECK_ACTIVATEGROUP	BOOL	m_bChkGroupAct
IDC_DATAQUALITY	CString	m_dataQulity
IDC_DATATIMESTAMP	CString	m_dataTS
IDC_DATAVALUE	double	m_dataValue
IDC_QUALITY	CString	m_readQulity
IDC_READ		
IDC_READVALUE	double	m_readValue
IDC_START		
IDC_STOP		
IDC_TIMESTAMP	CString	m_readTS
IDC_WRITE		
IDC_WRITERESULT	CString	m_writeResult
IDC_WRITEVALUE	double	m_writeValue

图 4.8

双击按扭控件，双击复选框控件生成相应的函数。

```
/*启动 OPC 服务器，添加组对象，添加项*/
void CAsynOpcDlg::OnStart()
{
    // TODO: Add your control notification handler code here

}
/*异步读，将结果显示在对话框*/
void CAsynOpcDlg::OnRead()
{
    // TODO: Add your control notification handler code here

}
/*异步写，将写结果显示在对话框*/
void CAsynOpcDlg::OnWrite()
{
    // TODO: Add your control notification handler code here

}

void CAsynOpcDlg::OnCheckActivategroup()
{
    // TODO: Add your control notification handler code here
```

```
}
/*停止服务器*/
void CAsynOpcDlg::OnStop()
{
    // TODO: Add your control notification handler code here

}
```

异步程序与同步程序最大的不同在于异步可以通过回调来由服务器通知客户变化的数据。为此在异步程序中我们添加了一个基于 IOPCDataCallback 的类 COPCDataCallback 作为回调的入口。

类声明如下：

```
class COPCDataCallback :
public CComObjectRoot,public IOPCDataCallback
{
public:
    COPCDataCallback() {};
    BEGIN_COM_MAP(COPCDataCallback)
        COM_INTERFACE_ENTRY(IOPCDataCallback)
    END_COM_MAP()
        // IOPCDataCallback
        STDMETHODIMP    OnDataChange(
          /* [in] */ DWORD dwTransid,
          /* [in] */ OPCHANDLE hGroup,
          /* [in] */ HRESULT hrMasterquality,
          /* [in] */ HRESULT hrMastererror,
          /* [in] */ DWORD dwCount,
          /* [size_is][in] */ OPCHANDLE __RPC_FAR *phClientItems,
          /* [size_is][in] */ VARIANT __RPC_FAR *pvValues,
          /* [size_is][in] */ WORD __RPC_FAR *pwQualities,
          /* [size_is][in] */ FILETIME __RPC_FAR *pftTimeStamps,
          /* [size_is][in] */ HRESULT __RPC_FAR *pErrors);
        STDMETHODIMP    OnReadComplete(
          /* [in] */ DWORD dwTransid,
          /* [in] */ OPCHANDLE hGroup,
```

```
            /* [in] */ HRESULT hrMasterquality,
            /* [in] */ HRESULT hrMastererror,
            /* [in] */ DWORD dwCount,
            /* [size_is][in] */ OPCHANDLE __RPC_FAR *phClientItems,
            /* [size_is][in] */ VARIANT __RPC_FAR *pvValues,
            /* [size_is][in] */ WORD __RPC_FAR *pwQualities,
            /* [size_is][in] */ FILETIME __RPC_FAR *pftTimeStamps,
            /* [size_is][in] */ HRESULT __RPC_FAR *pErrors);
        STDMETHODIMP    OnWriteComplete(
            /* [in] */ DWORD dwTransid,
            /* [in] */ OPCHANDLE hGroup,
            /* [in] */ HRESULT hrMastererr,
            /* [in] */ DWORD dwCount,
            /*[size_is][in]*/ OPCHANDLE __RPC_FAR *pClienthandles,
            /* [size_is][in] */ HRESULT __RPC_FAR *pErrors);
        STDMETHODIMP    OnCancelComplete(
            /* [in] */ DWORD dwTransid,
            /* [in] */ OPCHANDLE hGroup)
        {
            return S_OK;
        };
};
```

现在来修改 CAsynOpcDlg::OnStart()如下：

```
/*启动 OPC 服务器，添加组对象，添加项,设置回调*/
void CAsynOpcDlg::OnStart()
{
    // TODO: Add your control notification handler code here
    HRESULT         r1;
    CLSID       clsid;
    LONG        TimeBias = 0;
    FLOAT       PercentDeadband = 0.0;
    DWORD           RevisedUpdateRate;
    CString     szErrorText;
    m_ItemResult = NULL;
    //    初始化 COM 库
    r1 = CoInitialize(NULL);
    if (r1 != S_OK)
```

```
{
    if (r1 == S_FALSE)
    {
        AfxMessageBox("COM 库已经初始化");
    }
    else
    {
        AfxMessageBox("COM 库初始化失败");
        return;
    }
}
//   通过 ProgID,查找注册表中的相关 CLSID
r1 = CLSIDFromProgID(L"Intellution.OPCiFIX", &clsid);
if (r1 != S_OK)
{
    AfxMessageBox("获取 CLSID 失败");
    CoUninitialize();
    return;
}
//创建 OPC 服务器对象，并查询对象的 IID_IOPCServer 接口
r1 = CoCreateInstance (clsid, NULL,
CLSCTX_LOCAL_SERVER ,IID_IOPCServer,
(void**)&m_IOPCServer);
if (r1 != S_OK)
{
    AfxMessageBox("创建 OPC 服务器对象失败");
    m_IOPCServer = NULL;
    CoUninitialize();
    return;
}
//添加一个 group 对象，并查询 IOPCItemMgt 接口
r1=m_IOPCServer->AddGroup(L"grp1",
        TRUE,
        500,
        1,
        &TimeBias,
        &PercentDeadband,
```

```
        LOCALE_ID,
        &m_GrpSrvHandle,
        &RevisedUpdateRate,
         IID_IOPCItemMgt,
        (LPUNKNOWN*)&m_IOPCItemMgt);
    if (r1 == OPC_S_UNSUPPORTEDRATE)
    {
        AfxMessageBox("请求的刷新速率与实际的刷新速率不
        一致");
    }
    else
    if (FAILED(r1))
    {
        AfxMessageBox("不能为服务器添加 group 对象");
        m_IOPCServer->Release();
        m_IOPCServer = NULL;
        CoUninitialize();
        return;
    }

    //    为 AddItem 定义 item 表的参数
    m_Items[0].szAccessPath = L"";
    m_Items[0].szItemID = szItemID;
    m_Items[0].bActive              = TRUE;
    m_Items[0].hClient              = 1;
    m_Items[0].dwBlobSize           = 0;
    m_Items[0].pBlob                = NULL;
    m_Items[0].vtRequestedDataType = 0;
    r1 = m_IOPCItemMgt->AddItems(1,
        m_Items,
        &m_ItemResult,
        &m_pErrors);

    if ( (r1 != S_OK) && (r1 != S_FALSE) )
    {
        AfxMessageBox("AddItems 失败");
        m_IOPCItemMgt->Release();
```

```
            m_IOPCItemMgt = NULL;
            m_GrpSrvHandle = NULL;
            m_IOPCServer->Release();
            m_IOPCServer = NULL;
            CoUninitialize();
            return;
        }
        else if(r1==S_OK)
        {
            AfxMessageBox("AddItems()成功");
        }
        //   检测 Item 的可读写性
        if (m_ItemResult[0].dwAccessRights != (OPC_READABLE +
        OPC_WRITEABLE))
        {
            AfxMessageBox("Item 不可读，也不可写,请检查服务器
            配置");
        }

    //查询 group 对象的异步接口
        r1 = m_IOPCItemMgt->QueryInterface(IID_IOPCAsyncIO2,
        (void**)&m_IOPCAsyncIO2);
        if (r1 < 0)
        {
            AfxMessageBox("IOPCAsyncIO2 没有发现，错误的查
            询!");
            CoTaskMemFree(m_ItemResult);
            m_IOPCItemMgt->Release();
            m_IOPCItemMgt = NULL;
            m_GrpSrvHandle = NULL;
            m_IOPCServer->Release();
            m_IOPCServer = NULL;
            CoUninitialize();
            return;
        }
        //获得 IOPCGroupStateMgt 接口
        r1=m_IOPCItemMgt->QueryInterface(
```

```
    IID_IOPCGroupStateMgt,(void**) &m_IOPCGroupStateMgt);
    if (r1 != S_OK)
    {
        AfxMessageBox("IOPCGroupStateMgt 接口没有找到");
        CoTaskMemFree(m_ItemResult);
        m_IOPCItemMgt->Release();
        m_IOPCItemMgt = 0;
        m_GrpSrvHandle = 0;
        m_IOPCServer->Release();
        m_IOPCServer = NULL;
        CoUninitialize();
        return;
    }
    // 建立异步回调
    CComObject<COPCDataCallback>* pCOPCDataCallback;  //
回调对象的指针

    //通过 ATL 模板创建回调对象的实例

    CComObject<COPCDataCallback>::CreateInstance(&pCOPCD
    ataCallback);

    // 查询 IUnknown 接口
    LPUNKNOWN pCbUnk;
    pCbUnk = pCOPCDataCallback->GetUnknown();

    // 建立一个服务器的连接点与客户程序接收器之间的连接
    HRESULT hRes = AtlAdvise( m_IOPCGroupStateMgt, // [in]
    //连接点的 IUnknown  接口
    pCbUnk, // [in]  回调对象的 IUnknown  接口
    IID_IOPCDataCallback,// [in]  连接点  ID
    &m_dwAdvise                 // [out]  唯一的标识符
    );

    if (hRes != S_OK)
    {
        AfxMessageBox("Advise 失败!");
```

```
            CoTaskMemFree(m_ItemResult);
            m_IOPCItemMgt->Release();
            m_IOPCItemMgt = 0;
            m_GrpSrvHandle = 0;
            m_IOPCServer->Release();
            m_IOPCServer = NULL;
            CoUninitialize();
        return;
}
```

现在来修改 CAsynOpcDlg:: OnRead ()如下：

```
/*异步读*/
void CAsynOpcDlg::OnRead()
{
    // TODO: Add your control notification handler code here
    OPCHANDLE       *phServer;
    DWORD           dwCancelID;
    HRESULT         *pErrors;
    HRESULT         r1;
    CString         szOut;

     if (m_pErrors[0] != S_OK) // //Item 不可用

    {
        AfxMessageBox("OPC Item  不可用，不能用异步读功能!");
        return;
    }

     //内存分配
    phServer = new OPCHANDLE[1];
    //通过 server 句柄选择 item（由 AddItem 得到的）
    phServer[0] = m_ItemResult[0].hServer;

    r1 = m_IOPCAsyncIO2->Read(1,            // [in]读  1 个  Item
                    phServer,       // [in]  定义的 Item
                    1,              // [out]  客户 Transaction ID
                    &dwCancelID,    // [out]  服务器 Cancel ID
```

```
                    &pErrors        // [out] 服务器返回的错误码
                    );

    delete[] phServer;

    if (r1 == S_FALSE)
    {
        AfxMessageBox("Read()错误");
     }

    if (FAILED(r1))
    {
        AfxMessageBox("异步读失败!");
    }
    else
    {
        //释放内存，如果操作成功
        CoTaskMemFree(pErrors);
    }
}
```

现在来修改 CAsynOpcDlg:: OnWrite ()如下：

```
/*异步写，将写结果显示在对话框*/
void CAsynOpcDlg::OnWrite()
{
    // TODO: Add your control notification handler code here
    OPCHANDLE       *phServer;
    DWORD           dwCancelID;
    VARIANT         values[1];
    HRESULT         *pErrors;
    HRESULT         r1;
    LPWSTR          ErrorStr;
    CString         szOut;

    if (m_pErrors[0] != S_OK) // Item not available
    {
        AfxMessageBox("OPC Item 不可用，不能用同步读功能!");
        return;
```

```
}

//通过 server 句柄选择 item（由 AddItem 得到的）
phServer = new OPCHANDLE[1];
phServer[0] = m_ItemResult[0].hServer;

//    从对话框获得数据
UpdateData(TRUE);
//    设置 Variant 变量的数据类型和数值
values[0].vt = VT_R8;
values[0].dblVal = m_writeValue;


r1 = m_IOPCAsyncIO2->Write(    1,    //    [in] 写 1 Item
                        phServer,     // [in] 定义的 Item
                        values,       // [in] 定义的值
                        2,            // [in] 客户 transaction ID
                        &dwCancelID,// [out] 服务器 Cancel ID
                        &pErrors      // [out] 服务器返回的错误码
                        );
delete[] phServer;

if (r1 == S_FALSE)
{
    m_IOPCServer->GetErrorString(pErrors[0], LOCALE_ID,
    &ErrorStr);
    m_writeResult = ErrorStr;
    UpdateData(FALSE);
 }

if (FAILED(r1))
{
    AfxMessageBox("异步写 Item 错误");
}
else
{    // release [out] parameter in case of not failed
    CoTaskMemFree(pErrors);
```

```
        }
    }
```

从异步与同步的代码比较中，可以看出异步的读和写的结果显示不在 OnRead 和 OnWrite 中，这就是异步和同步的不同之处，异步不需要等待，而同步需要等待，异步调用后的结果显示需要在回调类 COPCDataCallback 实现。为了在对话框中显示操作的结果，在程序中使用了定时器来定时刷新对话框，实时显示数据，并采用了几个变量，changeFlag,readFlag,writeFlag 来区别回调来自数据变化，读，写。

异步程序执行的结果如下图所示。

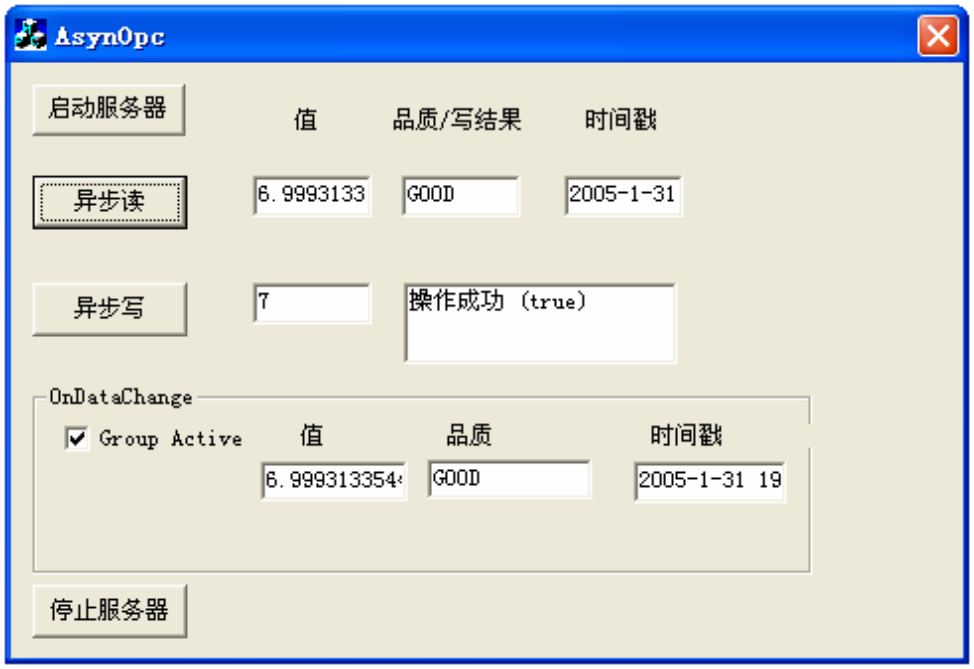


图 4.9

4.4 OPC 客户程序(VB 基础篇)

采用自动化接口的客户程序比采用自定义接口的客户程序要简单，

本节及下两节主要给出基于 VB 环境下的客户程序。对于 VB 环境下的客户程序编写，首先要了解 VB 编程，最基本的 VB 对象、VB 方法、VB 事件的概念。

VB 环境下的 OPC 对象定义如下:

OPCServer 对象，用来连接 OPC 服务器；OPCGroup 对象，用来添加 OPC 组对象；OPCItem 对象，用来添加 Item。

OPCServer 对象的几个关键方法：

1．Connect：连接 OPC 服务器。

Connect(ProgID As String,Optional Node as Variant)

ProgID：OPC 服务器的标识符。

Node：OPC 服务器所在计算机的名称或 IP 地址，默认不写时为本地 OPC 服务器。

2．Disconnect：断开 OPC 服务器的连接，与 Connect 成对使用。

DisConnect()。

OPCGroup 对象的几个关键方法：

1．Syncread 同步读取 OPC 组对象内单个或多个 ITEM 的数据值，质量戳，时间戳等。

SyncRead(Source As Integer,NumItems As Long,ServerHandles() As long,Byref Values() As Variant,Byref Errors() As long,Optional Byref Qualitys As Variant,Optional Byref TimeStamps As Variant)

Source：数据源，可以为 OPCCache，也可以为 OPCDevice。

NumItems：要读取的 Item 数目。

ServerHandles：要读取的 Item 的服务器句柄数组。

Values：返回的读取的 Item 的值数组。

Errors：返回的读取的 Item 的对应错误码数组。

Qualities：返回的读取的 Item 的品质（质量戳）数组，可选参数。

TimeStamps：返回的读取的 Item 的时间戳数组，可选参数。

2．SyncWrite 同步写 OPC 组对象内单个或多个 ITEM 的数据值。

SyncWrite(Numitems As long, ServerHandles()As long,Values()As Variant, ByRef Errors() As long)

Numitems：要写入的 ITEM 的数目。

ServerHandles：要写入的 Item 的服务器句柄数组。

Values：写入的 Item 的值数组。

Errors：返回的写入的 Item 的错误码数组。

3．Asyncread 异步读取 OPC 组对象内单个或多个 ITEM 的数据值，质量戳，时间戳等。读取结果由 AsyncReadComplete 事件返回。

AsyncRead(NumItems As Long,ServerHandles() As Long,Byref Errors() As long,TransactionID As Long,ByRef CancelID as Long)

NumItems：要读取的 Item 数目。

ServerHandles：要读取的 Item 的服务器句柄数组。

Errors：返回的读取的 Item 的对应错误码数组。

TransactionID：由 OPC 客户程序发送的事务标识符，

CancelID：由服务器发出的取消标识符，OPC 客户程序可以使用这个标识符来取消正在进行中的异步数据访问。

4．Asyncwrite 异步写入 OPC 组对象那单个或多个 ITEM 的数据值。写入结果由 AsyncwriteComplete 事件返回。

Asyncwrite(NumItems As Long,ServerHandles() As Long,Valuse() As Variant,Byref Errors() As long,TransactionID As Long,ByRef CancelID as Long)

NumItems：要读取的 Item 数目。

ServerHandles：要读取的 Item 的服务器句柄数组。

Values：要写入的 Item 的值数组。

Errors：返回的读取的 Item 的对应错误码数组。

TransactionID：由 OPC 客户程序发送的事务标识符。

CancelID：由服务器发出的取消标识符，OPC 客户程序可以使用这个标识符来取消正在进行中的异步数据访问。

5．事件 Datachange，在 OPC 组对象内的 ITEM 值或品质发生变化触发的事件，由服务器发给客户。

DataChange(ByVal TransactionID As Long, ByVal NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date)

TransactionID：由 OPC 客户程序发送的事务标识符。

NumItems：要读取的数据或品质发生变化的 Item 数目。

ClientHandles：OPC ITEM 的客户句柄数组。

ItemValues：返回的数值数组。

Qualities：返回的质量戳数组。

TimeStamps：返回的时间戳数组。

6．事件 AsyncReadComplete，在异步读取完成时发生的事件，由服务器通知客户。

AsyncReadComplete(ByVal TransactionID As Long, ByVal NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date, Errors() As Long)

TransactionID：由 OPC 客户程序发送的事务标识符。

NumItems：要读取的 Item 数目。

ClientHandles：OPC ITEM 的客户句柄数组。

ItemValues：返回的数值数组。

Qualities：返回的质量戳数组。

TimeStamps：返回的时间戳数组。

Errors：返回的错误码数组。

7．AsyncWriteComplete，在异步写入完成时发生的事件，由服务器通知客户。

AsyncWriteComplete(ByVal TransactionID As Long, ByVal NumItems As Long, ClientHandles() As Long, Errors() As Long)

TransactionID：由 OPC 客户程序发送的事务标识符。

NumItems：要读取的 Item 数目。

ClientHandles：OPC ITEM 的客户句柄数组。

Errors：返回的错误码数组。

4.5 OPC 客户程序(VB 同步篇)

在 VB6.0 中新建工程，建立如下窗体：

图 4.10

引用如下：

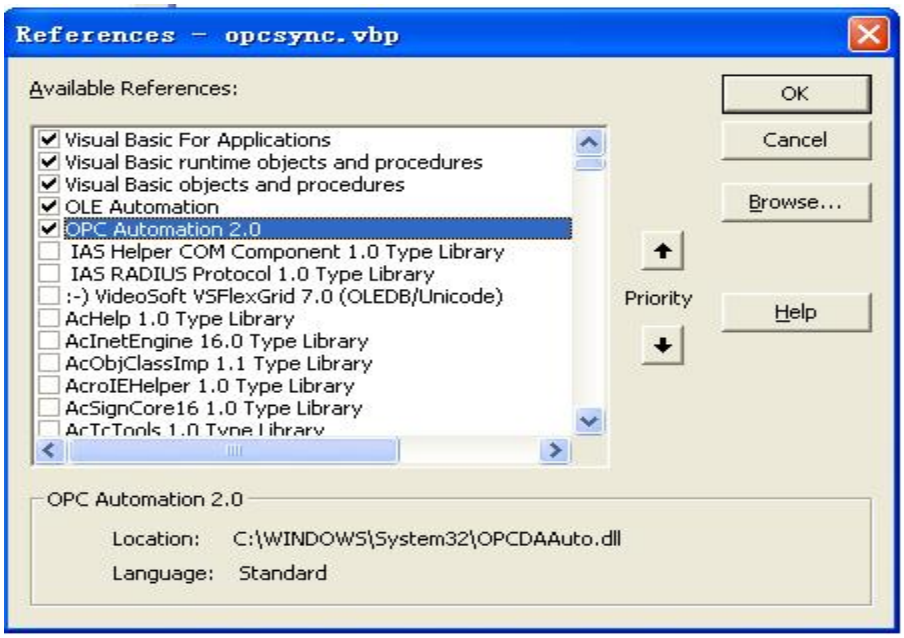


图 4.11

代码如下：

```
Option Explicit
    ‘OPC 对象的声明
    Dim WithEvents ServerObj As OPCServer
    Dim WithEvents GroupObj As OPCGroup
    Dim ItemObj As OPCItem

Private Sub Command_Start_Click()
    Dim OutText As String

    On Error GoTo ErrorHandler

    Command_Start.Enabled = False
    Command_Read.Enabled = True
    Command_Write.Enabled = True
```

```
    Command_Exit.Enabled = True

    OutText = "连接 OPC 服务器"

    ‘创建一个 OPC 服务器对象
    Set ServerObj = New OPCServer

    ‘连接一个 OPC 服务器
    ServerObj.Connect ("XXXSERVER")'XXXSERVER 为某 OPC 服
务器名称

    OutText = "添加组"

    ‘添加一个 OPC 组对象
    Set GroupObj = ServerObj.OPCGroups.Add("Group")

   OutText = "为组添加 Item"

    ‘向组对象添加 Item
    Set ItemObj = GroupObj.OPCItems.AddItem("XXXITEM", 1)
    'XXXITEM 为添加的 ITEM 名称

    Exit Sub

ErrorHandler:              '如果出现异常，则报出错误。
    MsgBox Err.Description + Chr(13) + _
    OutText, vbCritical, "ERROR"

End Sub

Private Sub Command_Read_Click()'同步读

    Dim OutText As String
    Dim myValue As Variant
    Dim myQuality As Variant
    Dim myTimeStamp As Variant

    On Error GoTo ErrorHandler
```

```
        OutText = "读 ITEM 值"

        ‘同步读
        ItemObj.Read OPCDevice, myValue, myQuality, myTimeStamp
        Edit_ReadVal = myValue
        Edit_ReadQu = GetQualityText(myQuality)
        Edit_ReadTS = myTimeStamp

        Exit Sub

    ErrorHandler:
        MsgBox Err.Description + Chr(13) + _
            OutText, vbCritical, "ERROR"

    End Sub

Private Sub Command_Write_Click()'同步写


    Dim OutText As String
    Dim Serverhandles(1) As Long
    Dim MyValues(1) As Variant
    Dim MyErrors() As Long

    OutText = "写值"
    On Error GoTo ErrorHandler

    Serverhandles(1) = ItemObj.ServerHandle
    MyValues(1) = Edit_WriteVal

    ‘同步写
    GroupObj.SyncWrite 1, Serverhandles, MyValues, MyErrors

    Edit_WriteRes = ServerObj.GetErrorString(MyErrors(1))

Exit Sub
ErrorHandler:
```

```
    MsgBox Err.Description + Chr(13) + _
        OutText, vbCritical, "ERROR"

End Sub


Private Sub Command_Exit_Click()'停止，删除 ITEM,删除 GROUP，删除 SERVER。

  Dim OutText As String

  On Error GoTo ErrorHandler

  Command_Start.Enabled = True
  Command_Read.Enabled = False
  Command_Write.Enabled = False
  Command_Exit.Enabled = False

  OutText = "删除对象"
  Set ItemObj = Nothing
  ServerObj.OPCGroups.RemoveAll
  Set GroupObj = Nothing

  '断开与 OPC 服务器的连接
  ServerObj.Disconnect
  Set ServerObj = Nothing

  Exit Sub

  ErrorHandler:
     MsgBox Err.Description + Chr(13) + _
         OutText, vbCritical, "ERROR"

  End Sub


  Private Function GetQualityText(Quality) As String
```

```
Select Case Quality
    Case 0:     GetQualityText = "BAD"
    Case 64:    GetQualityText = "UNCERTAIN"
    Case 192:   GetQualityText = "GOOD"
    Case 8:     GetQualityText = "NOT_CONNECTED"
    Case 13:    GetQualityText = "DEVICE_FAILURE"
    Case 16:    GetQualityText = "SENSOR_FAILURE"
    Case 20:    GetQualityText = "LAST_KNOWN"
    Case 24:    GetQualityText = "COMM_FAILURE"
    Case 28:    GetQualityText = "OUT_OF_SERVICE"
    Case 132:   GetQualityText = "LAST_USABLE"
    Case 144:   GetQualityText = "SENSOR_CAL"
    Case 148:   GetQualityText = "EGU_EXCEEDED"
    Case 152:   GetQualityText = "SUB_NORMAL"
    Case 216:   GetQualityText = "LOCAL_OVERRIDE"

    Case Else: GetQualityText = "UNKNOWN ERROR"
 End Select

End Function
```

可以看出采用自动化接口的 VB 客户程序首先要定义对象的声明，接着调用“New”方法生成服务器对象，通过 OPCServer 对象的 Connect 方法来连接 OPC 服务器，Connect 方法有两个参数，第一个参数是 OPC 服务器的 ProgID，第二个参数是节点名称，如果第二个参数为空，则标识为本地 OPC 服务器，如果需要连接远程 OPC 服务器，那么需要把远程机器的名称或 IP 地址作为第二个参数；连接 OPC 服务器后，需要通过调用 OPCServer.OPCGroups.Add 方法为 OPCServer 对象添加 OPCGroup 对象，参数为 OPCGroup 名称；接着调用 OPCItems.AddItem 方法添加 Item。至此完成了 OPC 服务器对象的创建，OPC 组对象的添加，和 OPC Item 的添加。如果一切返回都是成功的，便可以通过调用同步读和写的方法对 Item 进行读写操作了。在程序退出前同样需要释放掉

程序中引用的资源，断开 OPC 服务器的连接详见 Command_Exit_Click()。

与 VC 下的客户程序编写相比，VB 下的代码简单的多。从 VC 下对自定义接口的操作和 VB 下自动化接口的操作来看，两种环境下的步骤是类似的，至于读者采用何种语言编写 OPC 客户程序，需要看读者的实际需要。

4.6 OPC 客户程序(VB 异步篇)

上一节介绍了在 VB6.0 的环境下如何编写对 OPC 服务器的同步访问，本节主要介绍如何在 VB6.0 的环境下实现对 OPC 服务器的异步访问。

在 VB6.0 中新建工程，建立如下窗体：

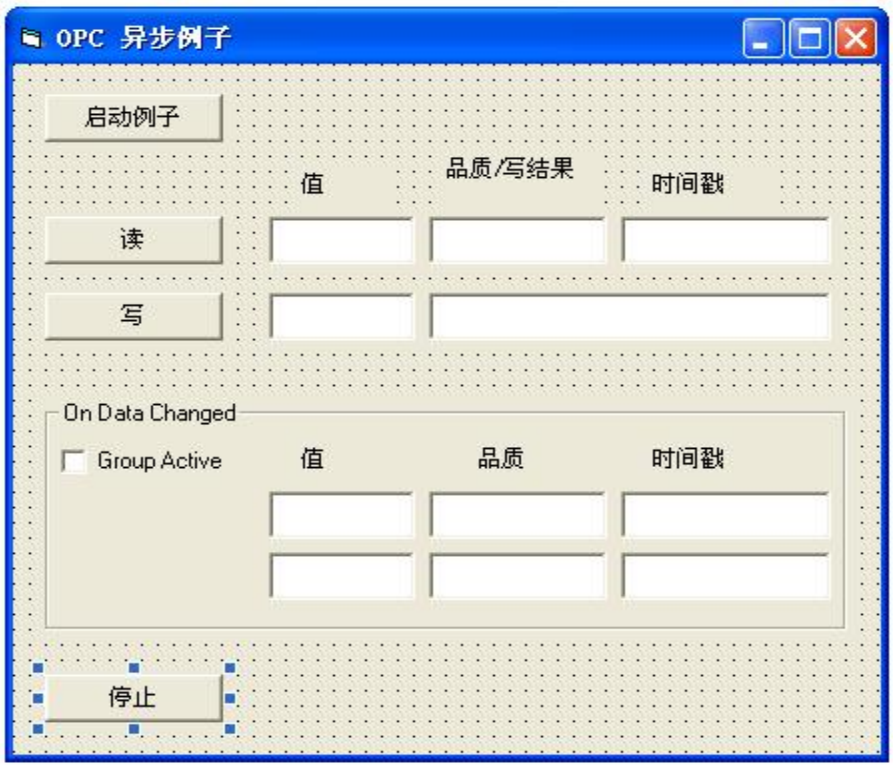


图 4.12

引用如下：

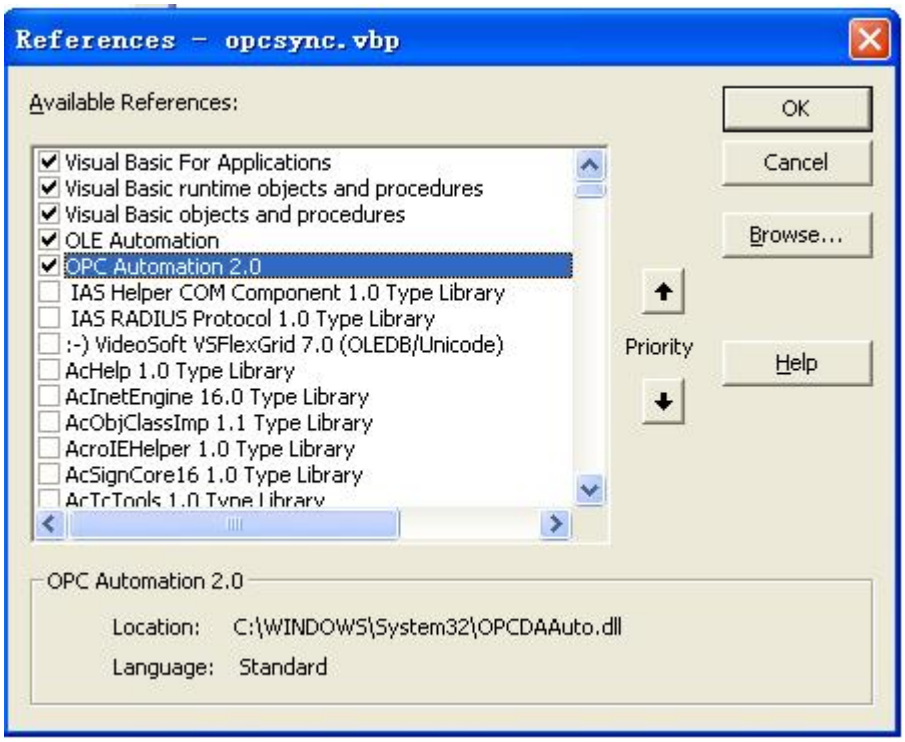


图 4.13

代码如下：

```
Option Explicit
Option Base 1


Const WRITEASYNC_ID = 1
Const READASYNC_ID = 2
Const REFRESHASYNC_ID = 3

'--------------------------------------------------------------------------
' 接口对象
```

```
'--------------------------------------------------------------------------
Public WithEvents ServerObj As OPCServer
Public WithEvents GroupObj As OPCGroup

Dim ItemObj1 As OPCItem
Dim ItemObj2 As OPCItem

Dim Serverhandle(2) As Long

Private Sub chkGroupActive_Click()

    If chkGroupActive = 1 Then
        GroupObj.IsActive = 1
    Else
        GroupObj.IsActive = 0
    End If
End Sub

Private Sub Command_Start_Click()

    Dim OutText As String

    On Error GoTo ErrorHandler

    Command_Start.Enabled = False
    Command_Read.Enabled = True
    Command_Write.Enabled = True
    Command_Exit.Enabled = True
    chkGroupActive.Enabled = True

    OutText = "连接 OPC 服务器"
    Set ServerObj = New OPCServer
    ServerObj.Connect ("XXXSERVER")

    OutText = "添加组"
    Set GroupObj = ServerObj.OPCGroups.Add("Group")

    ‘ 启用回调
```

```
    GroupObj.IsSubscribed = True

    chkGroupActive_Click

    OutText = "添加 ITEM"
    Set ItemObj1 = GroupObj.OPCItems.AddItem("XXXITEM1", 1)
    Set ItemObj2 = GroupObj.OPCItems.AddItem("XXXITEM2", 2)

    Serverhandle(1) = ItemObj1.Serverhandle
    Serverhandle(2) = ItemObj2.Serverhandle

    Exit Sub


ErrorHandler:
   MsgBox Err.Description + Chr(13) + _
      OutText, vbCritical, "ERROR"


End Sub

Private Sub Command_Read_Click() '异步读

    Dim OutText As String
    Dim myValue As Variant
    Dim myQuality As Variant
    Dim myTimeStamp As Variant
    Dim ClientID As Long
    Dim ServerID As Long
    Dim ErrorNr() As Long
    Dim ErrorString As String

    On Error GoTo ErrorHandler
    OutText = "读值"

    ClientID = READASYNC_ID

    ‘异步读
```

```
        GroupObj.AsyncRead 1, Serverhandle, ErrorNr, ClientID, ServerID
        If ErrorNr(1) <> 0 Then
            ErrorString = ServerObj.GetErrorString(ErrorNr(1))
            MsgBox ErrorString, vbCritical, "Error AsyncRead()"
         End If

         Erase ErrorNr
         Exit Sub

        ErrorHandler:
           MsgBox Err.Description + Chr(13) + _
              OutText, vbCritical, "ERROR"

End Sub

Private Sub Command_Write_Click() '异步写

        Dim OutText As String
        Dim Serverhandles(1) As Long
        Dim MyValues(1) As Variant
        Dim ErrorNr() As Long
        Dim ErrorString As String
        Dim Cancel_id As Long

        OutText = "写值"
        On Error GoTo ErrorHandler


        MyValues(1) = Edit_WriteVal
        ‘异步写
        GroupObj.AsyncWrite 1, Serverhandle, MyValues, ErrorNr,

        WRITEASYNC_ID, Cancel_id

        If ErrorNr(1) <> 0 Then
            ErrorString = ServerObj.GetErrorString(ErrorNr(1))
            MsgBox ErrorString, vbCritical, "Error AsyncRead()"
```

```
    End If

    Erase ErrorNr
    Exit Sub

  ErrorHandler:
       MsgBox Err.Description + Chr(13) + _
      OutText, vbCritical, "ERROR"

End Sub


Private Sub Command_Exit_Click() '停止
    Dim OutText As String

    On Error GoTo ErrorHandler

    Command_Start.Enabled = True
    Command_Read.Enabled = False
    Command_Write.Enabled = False
    Command_Exit.Enabled = False
    chkGroupActive.Enabled = False

    OutText = "删除对象"
    Set ItemObj1 = Nothing
    Set ItemObj2 = Nothing
    ServerObj.OPCGroups.RemoveAll
    Set GroupObj = Nothing

    '断开与 OPC 服务器的连接
    ServerObj.Disconnect
    Set ServerObj = Nothing

    Exit Sub

ErrorHandler:
  MsgBox Err.Description + Chr(13) + _
```

```
        OutText, vbCritical, "ERROR"

End Sub

'异步读回调
Private Sub GroupObj_AsyncReadComplete(ByVal TransactionID As
Long, ByVal NumItems As Long, ClientHandles() As Long,
ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date,
Errors() As Long)
    Dim ErrorString As String

    If (TransactionID = READASYNC_ID) Then
        If Errors(1) = 0 Then
            Edit_ReadVal = ItemValues(1)
            Edit_ReadQu = GetQualityText(Qualities(1))
            Edit_ReadTS = TimeStamps(1)
        Else
            ErrorString = ServerObj.GetErrorString(Errors(1))
            MsgBox ErrorString, vbCritical, "Error AsyncReadComplete()"
        End If
    End If
End Sub

'异步写回调
Private Sub GroupObj_AsyncWriteComplete(ByVal TransactionID As
Long, ByVal NumItems As Long, ClientHandles() As Long, Errors() As
Long)
    Dim ErrorString As String

    If (TransactionID = WRITEASYNC_ID) Then
        If Errors(1) = 0 Then
            Edit_WriteRes = ServerObj.GetErrorString(Errors(1))
        Else
            ErrorString = ServerObj.GetErrorString(Errors(1))
            MsgBox ErrorString, vbCritical, "Error AsyncWriteComplete()"
        End If
    End If
```

```
End Sub
'回调
Private Sub GroupObj_DataChange(ByVal TransactionID As Long,
ByVal NumItems As Long, ClientHandles() As Long, ItemValues() As
Variant, Qualities() As Long, TimeStamps() As Date)

    Dim i As Long

    For i = 1 To NumItems
        Edit_OnDataVal(i - 1) = ItemValues(i)
        Edit_OnDataQu(i - 1) = GetQualityText(Qualities(i))
        Edit_OnDataTS(i - 1) = TimeStamps(i)

    Next i

End Sub


Private Function GetQualityText(Quality) As String

    Select Case Quality
        Case 0:     GetQualityText = "BAD"
        Case 64:    GetQualityText = "UNCERTAIN"
        Case 192:   GetQualityText = "GOOD"
        Case 8:     GetQualityText = "NOT_CONNECTED"
        Case 13:    GetQualityText = "DEVICE_FAILURE"
        Case 16:    GetQualityText = "SENSOR_FAILURE"
        Case 20:    GetQualityText = "LAST_KNOWN"
        Case 24:    GetQualityText = "COMM_FAILURE"
        Case 28:    GetQualityText = "OUT_OF_SERVICE"
        Case 132:   GetQualityText = "LAST_USABLE"
        Case 144:   GetQualityText = "SENSOR_CAL"
        Case 148:   GetQualityText = "EGU_EXCEEDED"
        Case 152:   GetQualityText = "SUB_NORMAL"
        Case 216:   GetQualityText = "LOCAL_OVERRIDE"
        Case Else: GetQualityText = "UNKNOWN ERROR"
    End Select
```

End Function

可以看到异步与同步不同之处在于异步操作后直接返回，操作结果在回调函数中实现。异步读回调在 GroupObj_AsyncReadComplete 中实现，异步写回调在 GroupObj_AsyncWriteComplete 中实现，数据变化回调在 GroupObj_DataChange 中实现。所有回调的实现均采用事件的方式来操作，在 VB 中采用 Dim WithEvents ServerObj As OPCServer 这样的方式来使声明的对象支持事件。

除了上述的不同外，需要注意的是在进行异步通讯前，必须把这个代码： GroupObj.IsSubscribed = True 加上，这句代码是用来启用回调的，与 VC 程序中的 Advise 相对应。

4.7 OPC 客户程序(VC 多个组篇)

在前面几节中介绍了基于 Visual C++6.0 和 Visual Basic6.0 的 OPC 客户程序的编写，读者通过示例的编写，应该能够完成两种语言的客户程序的编写，本节主要介绍如何创建多个组对象，并介绍如何在多个组对象之间进行切换接口指针，通过切换组对象来实现读写操作等。为什么要介绍这样的一个例子呢，因为很可能遇到在一个应用程序中添加多个组对象的情况，而由于介绍 OPC 技术的书籍很少，介绍这种情况的例子更少，为了给大家一个较为实用的例子，所以特别在本节介绍 Visual C++6.0 环境下编写两个组对象的一个例子。

我们把本章 OPC 客户程序(VC++ 异步篇)的程序复制，粘贴后目录 AsynOpc 重命名为 AsynOpc_groups，然后我们在此基础对程序进行部分改写，以满足我们需要的功能。在对话框上添加两个单选框控件，命名为 IDC_RADIO_GROUP1，IDC_RADIO_GROUP2，如图 4.14 所示。

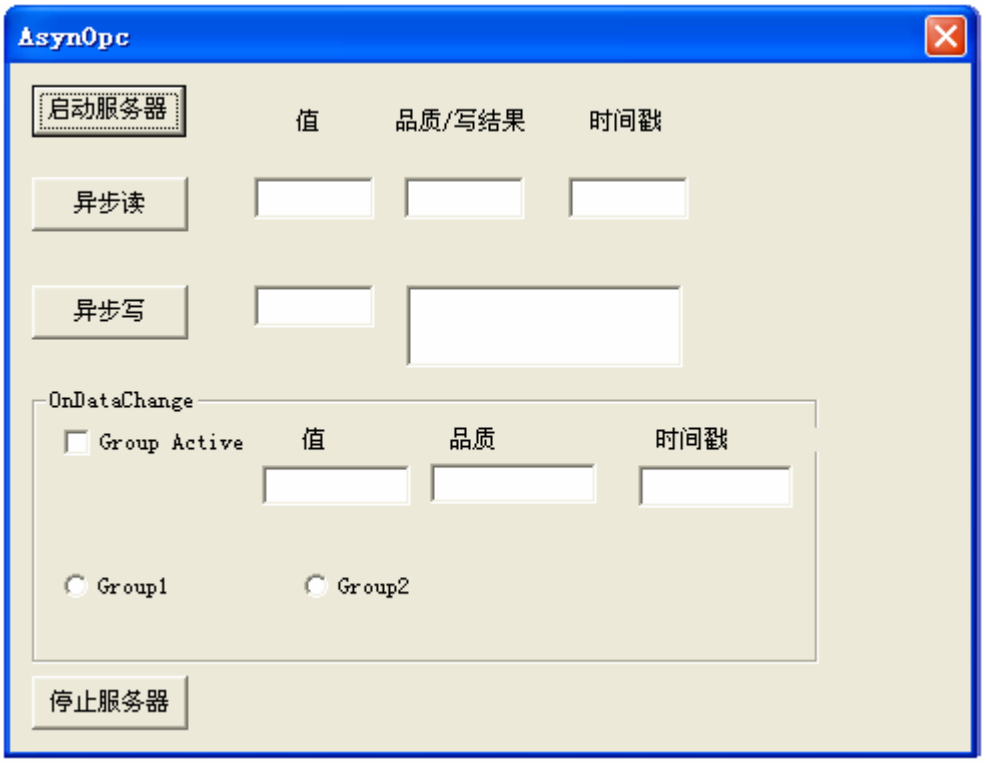


图 4.14 多组对象的客户程序界面

在例子中，创建两个组对象，一个名字为 grp1，刷新速率为 500 毫秒，另一个为 grp2，刷新速率为 1000 毫秒。OPC 服务器仍然选择 IFIX3.5, ProgID 为"Intellution.OPCiFIX"，第一个组添加 1 个 ITEM，第二个组添加 1 个 ITEM。本例中切换 Group 的原则是只有 1 个 Group 的状态为活动态，因此第一个 Group 在创建时状态为非活动态，即 AddGroup 的第二个参数为 FALSE。

为了切换两个 Group，我们为 CAsynOpcDlg 类添加了一个函数 BOOL ChangeGroup(LPWSTR szName);如图 4.15 所示。

图 4.15 添加 ChangeGroup 函数

为 CAsynOpcDlg 类添加了一个函数 int CallBack();如图 4.16 所示。

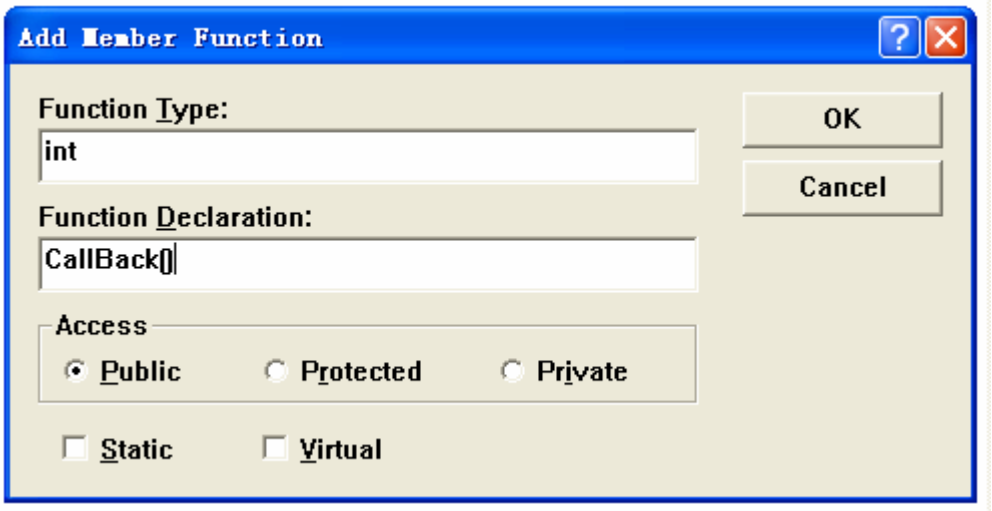


图 4.16 添加 CallBack 函数

双击两个单选按钮控件，并添加代码如下：

```
void CAsynOpcDlg::OnRadioGroup1()
 {
// TODO: Add your control notification handler code here
```

```
ChangeGroup(L"grp1");
 }

 void CAsynOpcDlg::OnRadioGroup2()
 {
     // TODO: Add your control notification handler code here
     ChangeGroup(L"grp2");
    }
```

ChangeGroup 实现如下：

```
BOOL CAsynOpcDlg::ChangeGroup(LPWSTR szName)
 {
DWORD dwRevUpdateRate;BOOL
m_bActivateGroup=FALSE;HRESULT r1;
if(m_dwAdvise)
{
r1 = AtlUnadvise(m_IOPCGroupStateMgt, IID_IOPCDataCallback,
m_dwAdvise);
if(r1!=S_OK)
{
     return FALSE;
}

r1= m_IOPCGroupStateMgt->SetState(NULL, // [in]
     RequestedUpdateRate,
     &dwRevUpdateRate, // [out]  返回的刷新率
     &m_bActivateGroup,//[in]  活动的组的标志
     NULL,                  // [in] TimeBias
     NULL,                  // [in]  死区参数
     NULL,                  // [in] LCID
     NULL);
if (FAILED(r1))
```

```
{    ;
     return FALSE;
}
m_IOPCGroupStateMgt->Release();
}

r1=m_IOPCServer->GetGroupByName(szName,IID_IOPCItemMgt,(LP
UNKNOWN*)&m_IOPCItemMgt);
r1=m_IOPCItemMgt->QueryInterface(IID_IOPCGroupStateMgt,
(void**) &m_IOPCGroupStateMgt);
m_bActivateGroup=TRUE;
r1= m_IOPCGroupStateMgt->SetState(NULL, // [in]
     RequestedUpdateRate,
     &dwRevUpdateRate,
     &m_bActivateGroup,
     NULL,                  // [in] TimeBias
     NULL,
     NULL,                  // [in] LCID
     NULL);
if (FAILED(r1))
{    ;
     return FALSE;
}
if (r1 != S_OK)
{
     CoTaskMemFree(m_ItemResult);
     m_IOPCItemMgt->Release();
     m_IOPCItemMgt = 0;
     m_GrpSrvHandle = 0;
     return FALSE;
}
return TRUE;
}

int CAsynOpcDlg::CallBack()

{
```

```
HRESULT             r1;
r1=m_IOPCItemMgt->QueryInterface(IID_IOPCGroupStateMgt,
(void**) &m_IOPCGroupStateMgt);
if (r1 != S_OK)
{
    CoTaskMemFree(m_ItemResult);
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = 0;
    m_GrpSrvHandle = 0;
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    CoUninitialize();
    return -1;
}
r1 = m_IOPCItemMgt->QueryInterface(IID_IOPCAsyncIO2,
(void**)&m_IOPCAsyncIO2);
if (r1 < 0)
{
    CoTaskMemFree(m_ItemResult);
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = 0;
    m_GrpSrvHandle = 0;
    m_IOPCServer->Release();
```

```
        m_IOPCServer = NULL;
        CoUninitialize();
        return -1;
    }
    // Establish Callback for all async operations
    CComObject<COPCDataCallback>* pCOPCDataCallback;  // Pointer to
Callback Object

    // Create Instance of Callback Object using an ATL template

    CComObject<COPCDataCallback>::CreateInstance(&pCOPCDataCallb
ack);
    LPUNKNOWN pCbUnk;
     pCbUnk = pCOPCDataCallback->GetUnknown();
     // Creates a connection between the OPC servers's connection point and
    // this client's sink (the callback object).
    HRESULT hRes = AtlAdvise(       m_IOPCGroupStateMgt,              //
[in] IUnknown Interface of the Connection Point
    pCbUnk,                         // [in] IUnknown Interface of the Callback
object
                                    IID_IOPCDataCallback,// [in] Connection
Point ID: The OPC Data Callback
                                    &m_dwAdvise                 // [out] Cookie
that that uniquely identifies the connection
                                );
```

```
    if (hRes != S_OK)
    {
        CoTaskMemFree(m_ItemResult);
        m_IOPCItemMgt->Release();
        m_IOPCItemMgt = 0;
        return -1;
    }
    return 0;

}
```

可以从程序代码中看出，在切换中起重要作用的一个接口函数是 GetGroupByName，这个函数通过 Group 名字来获得组的接口，因此我们在切换组时，按照我们两个组的切换原则，首先把一个组的回调取消，然后通过 GetGroupByName 来获得另一个组的接口，再通过调用 AtlAdvise 来建立另一个组的回调。这种设计在工控系统中较为符合双网冗余系统的通讯驱动，当一个网正常时，另一个只保持连接，并不通讯，当一个网中断时，切换到另一个网通讯。

4.8 OPC 客户程序(VB 浏览地址空间篇)

大家在用组态软件客户端程序或者 OPC 测试软件时，会发现可以浏览 OPC 服务器的地址空间，在第三章介绍了 OPC 服务器端地址空间的组织实现，本节介绍 VB6.0 环境下的浏览地址空间的程序。与异步、同步程序有所不同，浏览地址空间除了用到了 OPCServer 对象，主要用到

了 OPCBrowser 对象。OPCBrowser 对象的主要作用就是实现对地址空间的浏览，OPCBrowser 对象的属性如下：

Organization，只读，地址空间类型，FLAT 或 BRANCH（平民或树型）。

Filter：可以读写，使用 ShowBranches 或 ShowLeafs 时的类型过滤变量，通过类型变量，可以缩小被浏览地址空间的范围。

DataType：可以读写，使用 ShowLeafs 时，希望浏览的 ITEM 的数据类型。

AccessRights：可以读写，使用 ShowLeafs 时，希望浏览的 ITEM 的访问权限。

CurrentPosition：只读，返回地址空间中的现在位置。

Count：只读，浏览结果中的 ITEM 数目。

OPCBrowser 对象的方法如下：

Item，返回浏览地址空间结果中按照集合索引指定的对象。

ShowLeafs：将现在位置下的所有符合过滤条件的叶（Leaf）加入到浏览结果中。

ShowBranches：将现在位置下的所有符合过滤条件的枝（Branch）加入到浏览结果中。

MoveUP：向所在位置的上面一层移动。

MoveDown：向所在位置的下一层移动。

MoveToRoot：向地址空间的最上层移动。

MoveTo：向指定的位置移动。

GetItemID：返回 OPC ITEM 的名字。

下面给出具体的代码实现。

新建工程，窗体如图 4.17 所示。一个 TreeView 控件，用来显示浏览

到的地址空间，对于 TreeView 控件的具体用法，可以参考 VB 的编程书籍。

图 4.17

```
Option Explicit

'--------------------------------------------------------------------------
'全局变量
'--------------------------------------------------------------------------
Dim WithEvents g_Server As OPCServer
Dim g_Browser As OPCBrowser

Dim BrowseFilter As Long

Dim Vt_Filter As Integer
```

```
Dim nodX As Node
Dim Value As Variant
Dim FullName As String
Dim Relative As String
Dim i As Integer
'--------------------------------------------------------------------------
'选择读写过滤条件
'--------------------------------------------------------------------------
Private Sub cmbAccessRights_Click()
    Select Case cmbAccessRights.Text
    Case "OPC_ACCESS_WRITE"
    g_Browser.AccessRights = OPCWritable
    Case Else
    g_Browser.AccessRights = OPCReadable
  End Select
End Sub
'--------------------------------------------------------------------------
'选择浏览地址空间的方式（树状，平面）
'--------------------------------------------------------------------------
Private Sub cmbBrowseFilter_Click()
    Select Case cmbBrowseFilter.Text
        Case "OPC_FLAT"
        BrowseFilter = OPC_FLAT
        Case Else
        BrowseFilter = OPC_BRANCH
    End Select
End Sub
```

```
'--------------------------------------------------------------------------
'选择数据类型的过滤条件
'--------------------------------------------------------------------------
Private Sub cmbVtFilter_Click()
    Select Case cmbVtFilter.Text
        Case "VT_EMPTY"
        g_Browser.DataType = VT_EMPTY
        Case "UINT1"
          g_Browser.DataType = VT_UI1
        Case "UINT2"
        g_Browser.DataType = VT_UI2
        Case "TEXT"
        g_Browser.DataType = VT_BSTR
        Case "BOOLEAN"
     g_Browser.DataType = VT_BOOL
    End Select
End Sub
'--------------------------------------------------------------------------
' Form_Load()，窗口初始化
'在窗体初始化加载时，通过调用”New”创建 OPCServer 对象，然后通过'OPCServer
'对象的 Connect 方法连接指定的 OPC 服务器，创建成功后，'再调用 OPCServer 对'象的 CreateBrowser 方法创建 OPCBrowser 对象。并'设置 OPCBrowser 对象的过滤
'条件: DataType 为 VT_EMPTY，'AccessRights 为 OPCReadable。
'--------------------------------------------------------------------------
Private Sub Form_Load()
```

```
    Set g_Server = New OPCServer
    g_Server.Connect ("Matrikon.OPC.Simulation") '连接 OPC 服务器
    Set g_Browser = g_Server.CreateBrowser   '创建 OPCBrowser 对象
    cmbVtFilter.Text = "VT_EMPTY"
    cmbVtFilter.AddItem "UINT1"
    cmbVtFilter.AddItem "UINT2"
    cmbVtFilter.AddItem "TEXT"
    cmbVtFilter.AddItem "BOOLEAN"
    cmbVtFilter.AddItem "VT_EMPTY"
    BrowseFilter = OPC_BRANCH
    g_Browser.AccessRights = OPCReadable
    g_Browser.DataType = VT_EMPTY
    TreeView.LineStyle = tvwRootLines
End Sub
'------------------------------------------------------------------------
'浏览地址空间，根据选定的浏览方式（Branch 或 Flat）来调用相应的函"数
'------------------------------------------------------------------------
Private Sub cmdBrowsing_Click()

    Dim i As Integer
    Dim Key As String
    MousePointer = 11
    cmdBrowsing.Enabled = False

    Set nodX = Nothing
```

```
    TreeView.Nodes.Clear
    Select Case BrowseFilter
        Case OPC_FLAT
        BuildFlat
        Case OPC_BRANCH
        BuildTree
    End Select

    MousePointer = 0
    cmdBrowsing.Enabled = True

End Sub
'--------------------------------------------------------------------------
'程序退出，退出前释放资源
'--------------------------------------------------------------------------
Private Sub cmdExit_Click()
    Set g_Server = Nothing
    Set g_Browser = Nothing
End Sub

'--------------------------------------------------------------------------
' Sub BuildFlat()
' 目的: 显示所有 Item
'--------------------------------------------------------------------------
Public Sub BuildFlat()
```

```
    Dim Key As String
    Dim Leaf As Variant

    g_Browser.MoveToRoot
    g_Browser.ShowLeafs True
    For Each Leaf In g_Browser
        i = i + 1
        Key = "#" & i
        Set nodX = TreeView.Nodes.Add(Null, tvwNext, Key,
        g_Browser.GetItemID(Leaf))
        DoEvents
    Next Leaf
End Sub

'--------------------------------------------------------------------------
' Sub BuildTree()
' 目的: 通过树状方式显示地址空间
'--------------------------------------------------------------------------
Public Sub BuildTree(Optional Relative As Variant)
    Dim Branch, Leaf As Variant
    Dim Key As String

    g_Browser.ShowBranches
    For Each Branch In g_Browser
        i = i + 1
```

```
            Key = "#" & i
            Set nodX = TreeView.Nodes.Add(Relative, tvwChild, Key,
            Branch)
            '向下一个分支
            g_Browser.MoveDown (Branch)
            BuildTree (Key)
            '向上一个分支
            g_Browser.MoveUp
            DoEvents
        Next Branch

        g_Browser.ShowLeafs False
        For Each Leaf In g_Browser
            i = i + 1
            Key = "#" & i
            Set nodX = TreeView.Nodes.Add(Relative, tvwChild, Key,
            g_Browser.GetItemID(Leaf))
        Next Leaf

    End Sub
```

4.9 OPC 客户程序(VC 浏览地址空间篇)

上一节介绍了如何用 VB 来实现对 OPC 服务器地址空间的浏览，本节主要介绍如何采用 VC++来实现对 OPC 服务器地址空间的浏览。实际上用 VC＋＋和用 VB 来实现 OPC 服务器地址空间浏览的流程是一样的，

只不过由于 VB 是采用自动化接口，而 VC++采用自定义接口，所以采用 VB 编写的浏览地址空间的程序看起来比 VC++简单的多，就如同前面介绍的 OPC 异步、同步程序一样。在 VC＋＋6.0 环境下，新建基于对话框的 MFC 工程，命名为 Browser，对话框如图 4.18 所示，主要包含一个 Tree Control 控件和三个组合框控件，两个按钮控件和三个标签控件组成。详细代码见本书源程序。有关控件的使用方法请参考 MSDN 和相关的技术书籍。

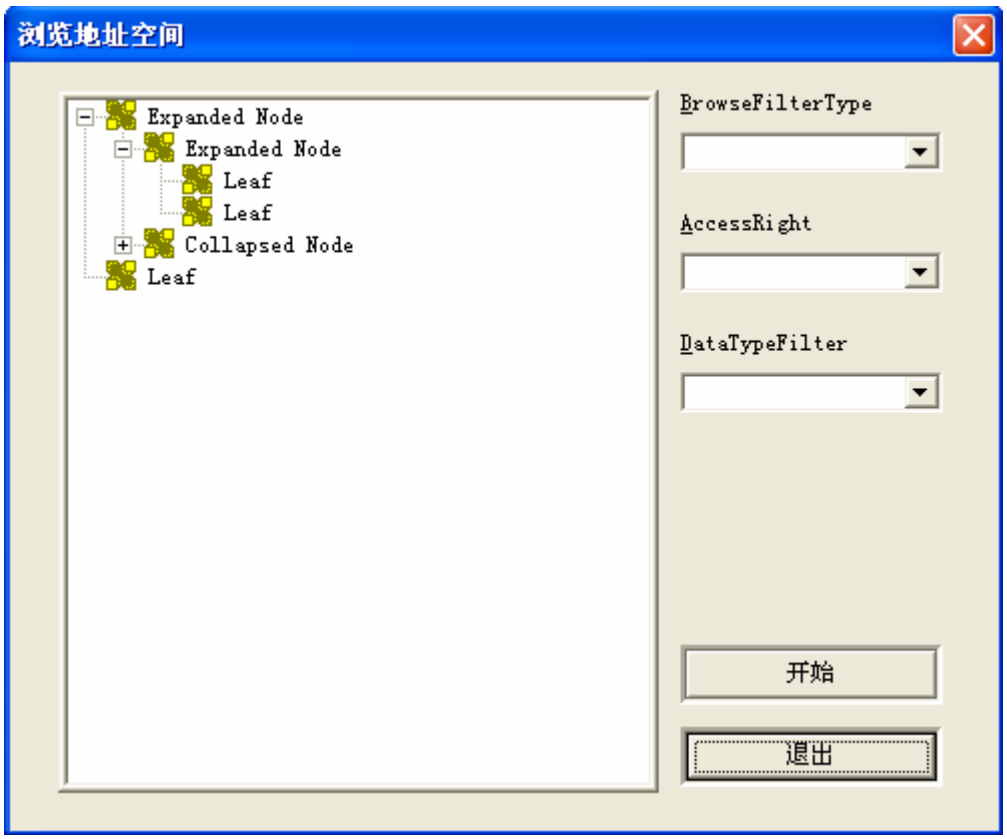


图 4.18

如同 VB 环境下的程序，VC++下仍然采用 CoCreateInstance 来创建 OPC 服务器对象，并通过查询获得 IOPCBrowseServerAddressSpace 接口，详见 OnStartButton()和 CreateOPCBrowser。如同编写 VC++下的异步、

同步程序，浏览地址空间的程序仍然需要在程序初始化时初始化 COM 库，在结束时退出 COM 库。

获得接口后，就可以按照设定的过滤条件（读，写，数据类型等）来浏览 OPC 服务器的地址空间了，主要通过 BrowseLevel(HTREEITEM gParent,CString FullName,IOPCBrowseServerAddressSpace *pIOPCBrowse) 来实现，这个函数中调用了 IOPCBrowseServerAddressSpace 接口下的多个方法，如 BrowseOPCItemIDs，ChangeBrowsePosition，并对 BrowseLevel 进行了递归调用。在此过程中需要注意两个地方，一个是 IEnumString 的使用，在第三章 OPC 服务器浏览地址空间的实现中，已经使用了 IEnumString，在客户程序中，同样需要 IEnumString 来实现对地址空间的操作；另一个需要注意的地方是通过 IEnumString 的方法获得是 UNICODE 编码的字符串，需要把 UNICODE 编码转换为 CHAR 型编码，因此在程序中通过调用_com_util::ConvertBSTRToString 来实现两种类型的转换，为了调用这个函数需要加上 comutil.h 和 comsupp.lib。最后一个需要注意的地方，如同编写 OPC 异步、同步程序一样，需要对申请的资源，在程序退出时释放。相对于 OPC 异步、同步程序而言，编写 OPC 浏览地址空间的程序简单些。

重点：同步过程，异步过程，浏览地址空间过程。

第 5 章 OPC 服务器的远程访问

关键字：DCOM 远程 配置 防火墙

The Distributed Component Object Model (DCOM) 是为了支持在局域网或者广域网或者 INTERNET 上的组件对象通讯。因为 DCOM 技术基于 COM 技术，是 COM 的无缝延续，一种领先的组件技术，可以利用 COM 组件的优势来进行分布式的访问，因为 DCOM，你不需要了解底层的网络协议。通过 DCOM 技术，我们的 OPC 客户程序可以访问不在同一台机器上的 OPC 服务器。DCOM 一般而言在 Microsoft Windows NT® 4.0 及以上操作系统上运行。实际上 DCOM 可以应用在 UNIX 的平台上（http://www.sagus.com）。

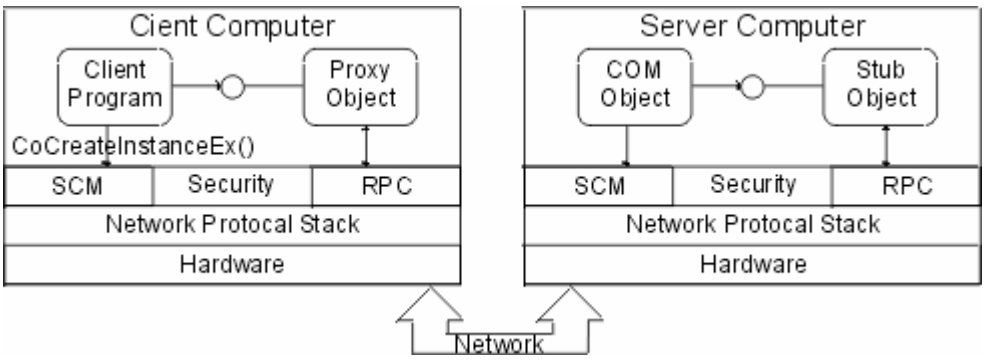


图 5.1 远程创建 COM 组件示意图

一旦创建了远程的 COM 服务器，所有的调用将通过 proxy 和 stub 对象配置。proxy 和 stub 使用 RPC（Remote Procedure Calls，远程过程调用）进行通信，RPC 处理所有网络交互。在服务器端，stub 对象负责配置，而客户端则由 proxy 负责。

跨网络的数据传送由 RPC 负责。实际上，DCOM 使用一个扩展类型

的 RPC，称为对象 RPC（Object RPC）或者 ORPC。RPC 可以运行在多种不同的协议上，包括有 TCP/IP，UDP，NetBEUI，NETBIOS 和命名管道。标准的 RPC 协议是 UDP（用户数据报协议）。UDP 是一个无连接的协议，看来与 DCOM 这种面向连接的系统配合并不是一个好主意。不过这并不是一个问题，DCOM 自动负责管理连接。

对于 COM 组件而言，进程外组件不需要作任何修改，只需要把 DCOM 配置一下便可以供客户程序远程访问。本章主要介绍如何配置 DCOM 来访问远程 OPC 服务器，以及 DCOM 的连接管理，最后给出 Visual C++6.0 下的远程访问 OPC 服务器的客户程序实例。

5.1 OPC 服务器远程访问的 DCOM 配置(WINNT,WIN2000)

下面以 Windows NT 4.0(SP6)和 Windows 2000 为例来说明远程访问 OPC 服务器时服务器端及客户端需要的配置。

一、在装有 OPC 服务器的机器上 DCOM 配置如下

1．运行服务器上的 dcomcnfg 程序，进行 DCOM 配置。

2. 进入 DCOM 的总体默认属性页面，将“在这台计算机上启用分布式 COM”打上勾，将默认身份级别改为“无”。

3. 进入 DCOM 的总体默认安全机制页面，确认默认访问权限和默认

启动权限中的默认值无 EveryOne，

如果不去掉 EveryOne，应用服务器不能正常启动。

4.在常规页面中，双击你的应用服务器，打开你的应用服务器 DCOM 属性设置。

5.将常规页面中的身份验证级别改为“无”。

6.位置页面中选上“在这台计算机上运行应用程序”。

7.将安全性页面设置中，均选择“使用自定义访问权限”，编辑每一个权限，将 EveryOne 加入用户列表中。

8.身份标识页面中，选择“交互式用户”。

注意 NT 的 GUEST 用户不能禁用。

二、在客户端机器上 DCOM 配置如下：

1．点”开始”->”运行”,输入”dcomcnfg”,然后回车，启动 dcom 配置。

2． 常规页面中，双击你的应用服务器，打开你的应用服务器 DCOM 属性设置。

3． 将常规页面中的身份验证级别改为“无”。

4． 身份标识页面中，选择“交互式用户”。

5． 位置页面中,选择“在这台计算机上运行应用程序”。

进入 DCOM 的总体默认属性页面，将“在这台计算机上启用分布式 COM”打上勾，将默认身份级别改为“无”。

两端配置好后，客户端机器就可以访问远方机器的 OPC Server 了。注意：这样的配置需要两端的用户名和密码一致。

可以在服务器端和客户端添加一个专门的用户用来作为 OPC 连接时使用，如 OPCUSER，两端密码一致。然后在配置时需要修改如下：

在服务器端和客户端把身份标识设置为指定用户，用户名为

OPCUSER，输入密码。

5.2 Windows XP (SP2)下 OPC DCOM 的配置

Windows XP SP2 的主要目标就是减少 Windows XP 易受的一般恶意攻击，服务包从以下四个方面的改进可减少最常见的攻击影响：

1.Windows XP 在网络屏蔽方面的改进

a.RPC 和 DCOM 通信的改进

b.内部 Windows 防火墙的改进

2.改善的内存保护

3.更安全的电子邮件处理

4.提高的 Internet 浏览器安全

多数 OPC 客户和服务都是通过 DCOM 进行网络通信，因此由于 SP2 的改变，网络通信将会受到影响，当按照默认设置安装了 SP2 服务包，OPC 通信将不能正常工作。

本节介绍了 Windows XP SP2 情况下恢复 OPC 通信的一些必要设置。SP2 具有许多更新以及安全方面的改进，其中的两个就直接影响着经由 DCOM 的 OPC 通信。第一是增加了新的 DCOM 限制设置，第二是 XP 的软件防火墙有很大的改进，并且默认为开启。

由于 OPC 使用的回调机制是在进行回调时把 OPC Client 看作一个 DCOM 服务而把 OPC Server 看作一个 DCOM 客户，因此在任何通过 DCOM 进行通信的 OPC Servers 和 OPC Clients 机器上都需要进行下述的设置。

注意：在 XP SP2 下，同一台计算机上的 OPC 客户和 OPC 服务器通信时，不需要进行下述的配置 OPC 通信（使用 COM，而不是 DCOM）

将继续工作。

Windows 防火墙

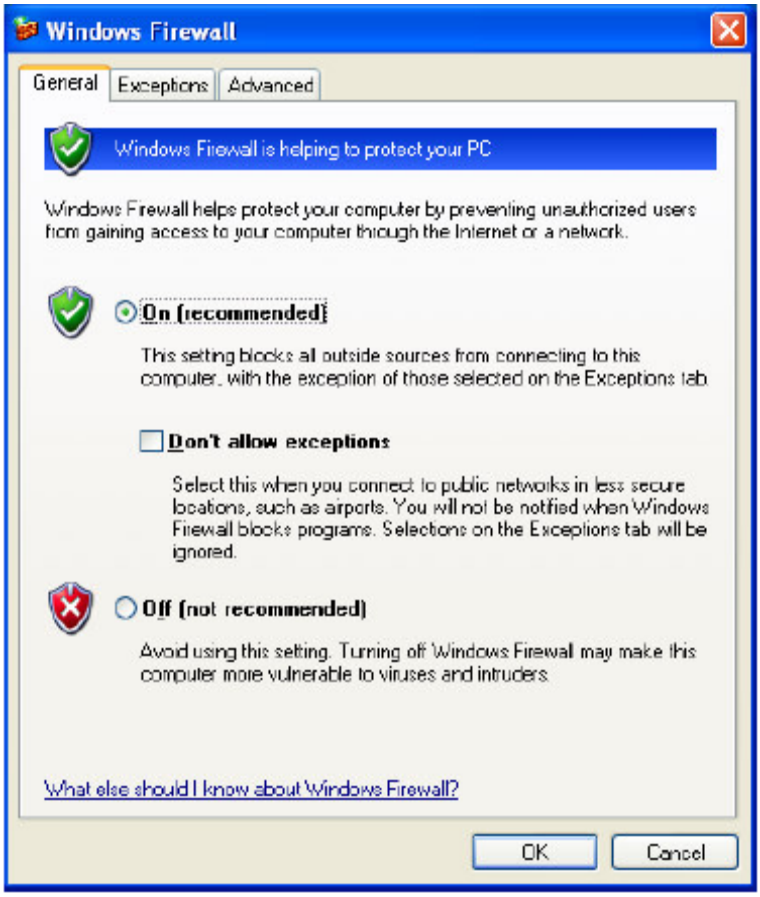
本地初始化时，Windows 防火墙允许通过网络接口进行通信，但是默认情况下阻止任何“未被请求”的通信。然而，此防火墙基于“例外”，也就是说管理员能指定不遵循此规则的应用和端口并且能对未被恳求的请求作出响应。

防火墙的例外可以从以下两个主要级别进行设定，应用级别以及端口和协议级别。应用级别就是你指定哪些应用能够对未被请求的请求作出响应，协议和端口级别就是使防火墙允许或禁止某个特定的 TCP 或 UDP 端口的通信。为了使任何 OPC 客户/服务应用通过 DCOM 进行通信，两个级别都需要进行修改。

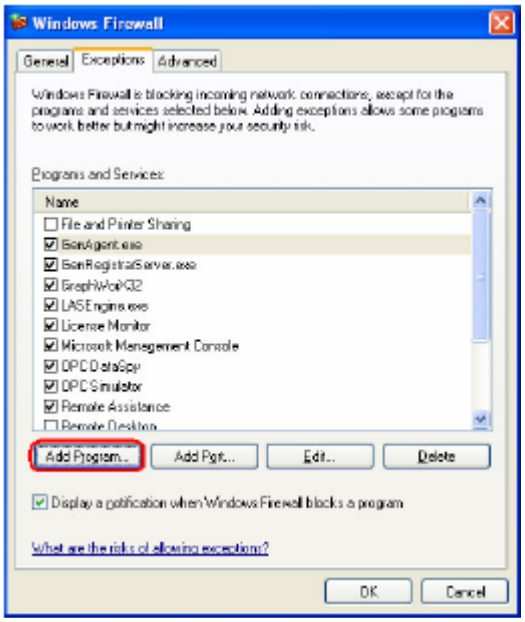
配置防火墙

1.默认情况下，Windows 防火墙设置为“On”。这是微软和 OPC 推荐的设置，可以给你的机器最高保护。进行调试时可以通过暂时关掉防火墙来判断防火墙的配置是否正确。

注意：如果计算机在公司防火墙的保护下，也可以永久地关闭 Windows 防火墙。关掉 Windows 防火墙后，无需进行本文提到的个人防火墙设置 OPC 就可以通信。

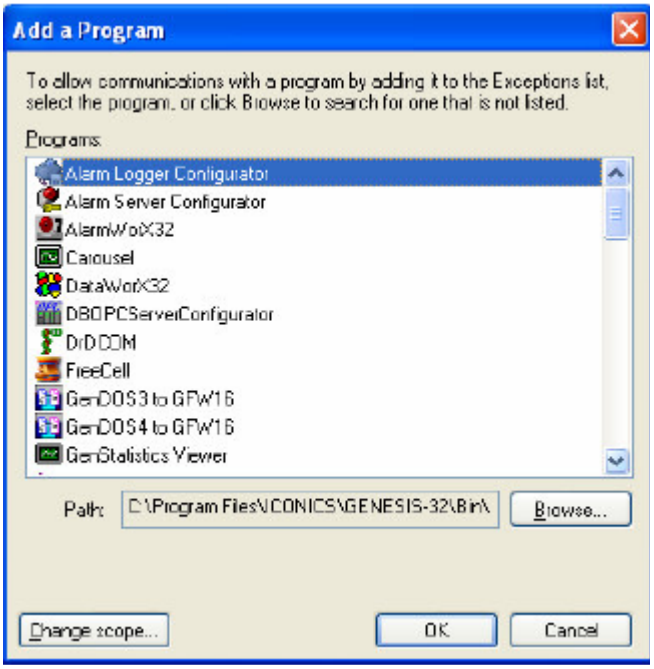


2.选择“Exception”选项并且把所有的 OPC Clients 和 Servers 添加到例外列表。还要添加 Microsoft Management Console(下一部分的 DCOM 配置会用到)和在 Windows\System32 目录下的 OPC 应用 OPCEnum.exe

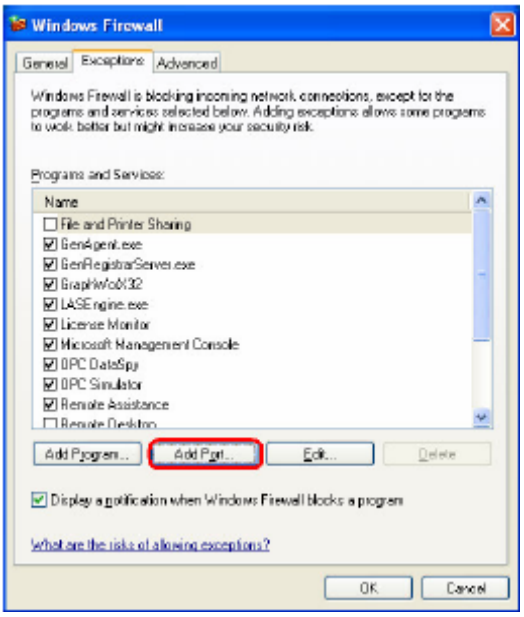


在Add a Program对话框中，有机器上大部分应用的列表，但不是所有应用都显示在此列表中。可以通过“Browse”按钮来查找安装在本机上的其它可执行程序。

注意：只有 EXE 文件可以添加到例外列表中。对于处理中的（in-process）OPC Servers 和 Clients（DLLS 和 OCXS），你须要将访问它们的应用添加到这个列表中。



3.添加程序初始化需要的 TCP 端口为 135，并且允许进来的回调请求。在 Windows 防火墙的例外选项中，点击添加端口。

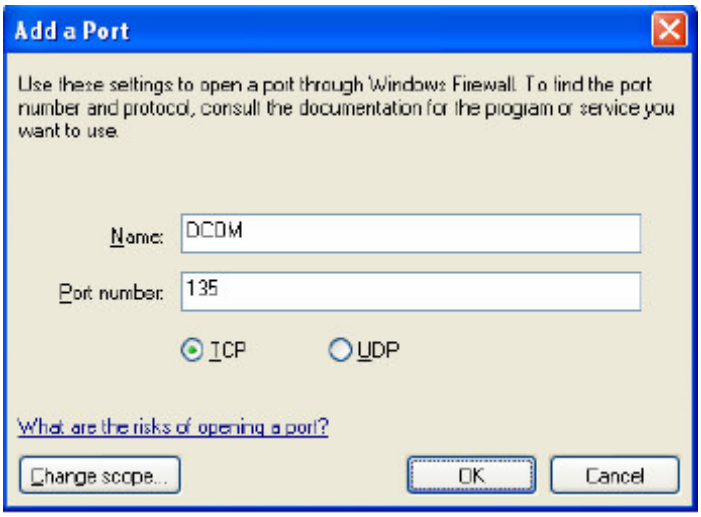


在 Add a Port 对话框中，填写以下部分：

Name：DCOM

Port number：135

选择 TCP 通信选项按钮



DCOM 的改进

Windows XP 的 SP2 在 DCOM 安全方面的改进；当在网络中使用 OPC 时有两个特别需要考虑的方面：首先，默认启动和访问权限对话框允许用户设置使用 DCOM 的应用程序的权限。其次，对在启动和访问权限中定义的每一个用户，都必须明确定义本地访问和远程访问。

关于 DCOM 的默认启动和访问权限的简短背景：启动权限定义通过网络或本地启动 COM 应用的用户（例如一个 OPC Server）。访问权限定义访问已启动的应用程序的用户。应用程序可以从以下三个位置中的其中一个获得它们的启动和访问权限：使用为它们的应用明确定义的设置，使用默认权限或通过可编程设置它们自己的权限。因为一个应用程序能可编程地设置它自己的权限，虽然明确定义或默认的设置正确，但也许不使用并且因此用户不能明确地控制这些设置。为克服这一安全缺陷，微软对启动权限和访问权限中的 DCOM 安全机制增加了“限制”，来限制应用程序可使用的权限。这些限制阻止应用程序使用 DCOM 配置

设置没有指定的权限。默认情况下 SP2 的设制不允许 OPC 通过网络进行通信。

除了新的权限限制之外，现在还必须指定已定义的用户或组是否具有本地或远程权限（或两者都可以）。为了使 OPC 应用在网络上通过 DCOM 进行工作，必须对权限进行设置，使用户可以启动和访问远程机器上的 OPC Servers 和 clients。

配置 DCOM

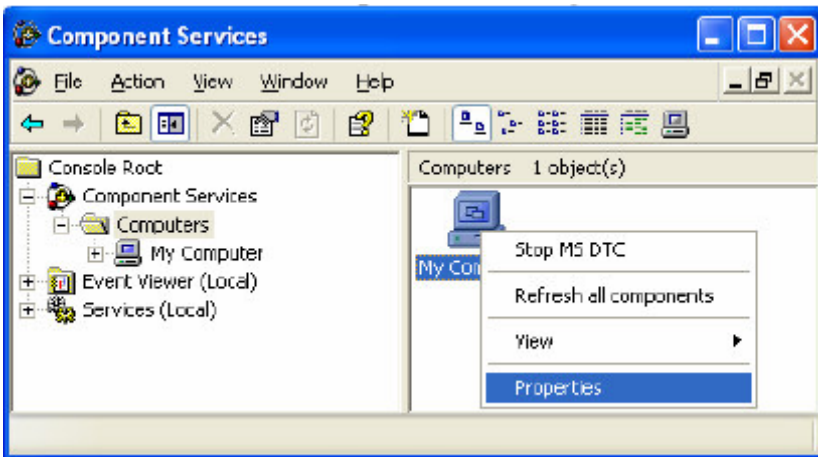
Windows XP SP2 下 OPC 通信的 DCOM 配置步骤如下：

1.开始-〉运行并且输入 DCOMCnfg，点击 OK。

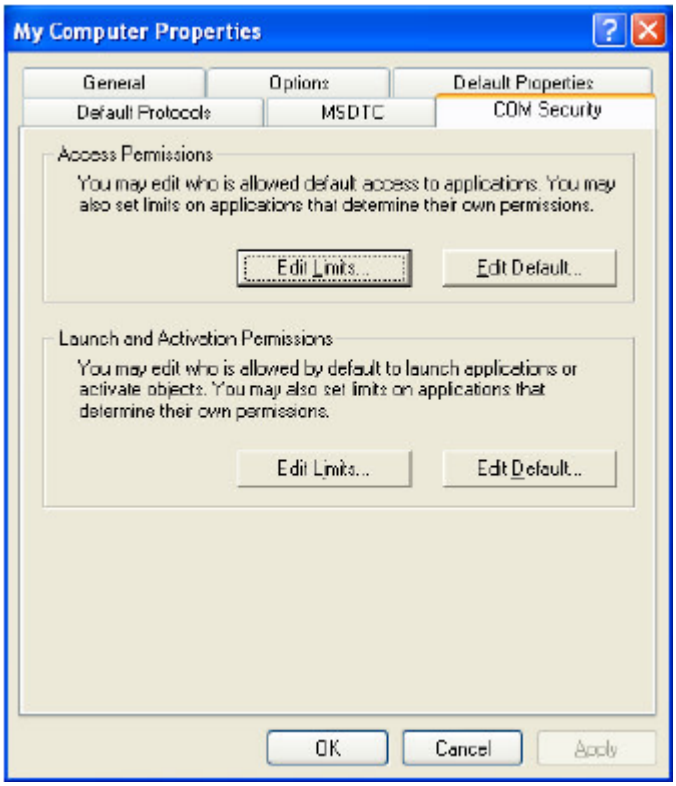
2.点击 Console Root 下的 Component Services。

3.点击 Component Services 下的 Computers 打开它。

4.在右边面板上的 My Computer 上点击鼠标右键并且选择 Properties



5.在 COM Security 选项卡中，这四个权限配置都必须进行设置：



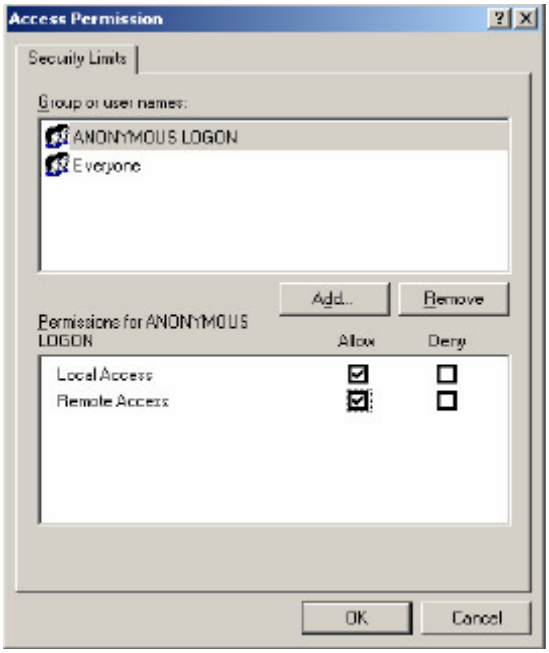
6.编辑访问和启动权限

a.访问权限-Edit Limits…

选中对话框中 ANONYMOUS LOGIN 用户的 Remote Access 复选框

注意：此设置对起作用的 OPCEnum.exe 和一些为了允许匿名连接将

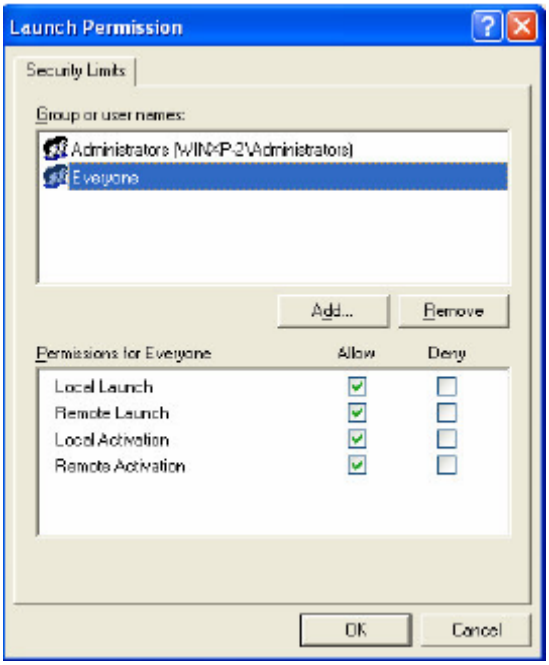
它们的 DCOM 的“Authentication Level”设置成”None”的 OPC Servers 和 Clients 是必须的。如果不使用 OPCEnum，就不需要使匿名用户能进行远程访问。



b.启动和激活权限-Edit Limits…

检查对话框中 Everyone 用户的 Remote 复选框。

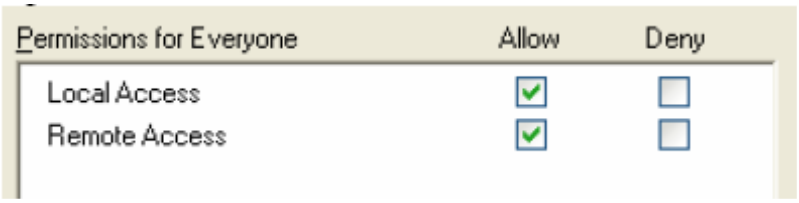
注意：由于 Everyone 包括所有的认证用户，因此把这些权限添加到一个较小的用户子系统是比较合适的。为达到这一目的建议创建一个名为”OPC Users”的组，并且将所有运行 OPC Server 或 Client 的用户帐号添加到这个组中。那么在 Everyone 显示在这些配置对话框中的任何地方代替“OPC Users”



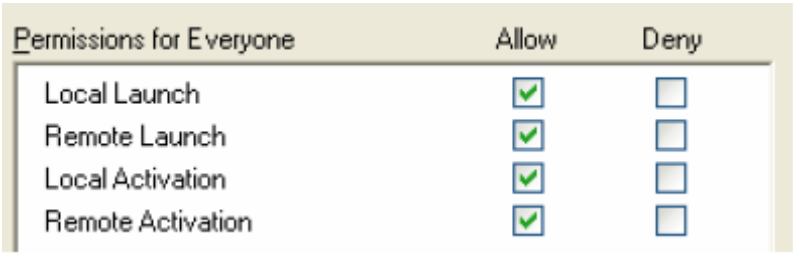
7.编辑默认的访问和启动权限

对每一个进行 OPC 通信的用户（或组）（例如：“OPC Users”）来说,要确保 Local Allow 和 Remote Allow 的复选框都被选中。

每一个用户的访问权限：



每一个用户的启动和激活权限：



5.3 DCOM 的远程连接管理

对于分布式组件而言，通过 DCOM 通讯，我们需要了解 DCOM 的远程连接管理，因为 DCOM 是网络连接，对于连接而言，网络连接显然比同一机器上的连接更加脆弱。当网络中断或者硬件出现故障时，尤其需要注意网络的连接问题。对于大部分 OPC 技术初学者，远程 DCOM 的连接管理相对而言不是很了解，其实 DCOM 按照一种简单的机制来处理远程管理。

DCOM 是通过给每个组件保持一个索引计数来管理对组件的连接，大家知道组件可能仅仅连接到一个客户上，也可能被多个客户共享。当一个远程客户与一个组件建立连接时，DCOM 相应的增加组件的索引计数，相反的当一个远程客户释放连接时 DCOM 相应的减少组件的索引计数，如果索引计数为 0，组件就可以被释放。对于远程组件而言，DCOM 使用有效的地址合法性检查协议来检查客户进程释放仍然是活跃的，客户机周期性的发送消息，当大于等于三次 PING 周期而组件没有收到 PING 消息时，DCOM 就认为这个连接中断，同时相应的减少索引计数，当索引计数为 0 时就释放组件。从组件的这一点来看，无论是客户进程自己中断连接还是网络中断这种情况，都被同一种索引计数机制处理。

因此通过 DCOM 远程访问组件，在网络中断时，不会引起组件崩溃和异常的问题。虽然如此，但是个人建议大家在做网络通讯时，最好不通过 DCOM 来做，而且客户程序一定要做好，在退出时要把所有的资源释放。

5.4 远程访问 OPC 服务器的客户程序实现（VC++）

本节我们主要以 VC++环境下的客户程序访问远程 OPC 服务器为例。远程访问 OPC 服务器是以 DCOM 为基础，所调用的函数与本地客户相比较，创建 OPC 服务器对象的函数不同，其它基本相同。

对于 VB 环境下的远程访问，只需要把 connect 方法的 node 参数输入即可。

我们把第四章 OPC 客户程序(VC++ 异步篇)的程序复制，粘贴后目录 AsynOpc 重命名为 AsynOpc_remote。

打开 AsynOpc.dsw，我们需要将在 CasynOpcDlg 类定义处添加保护型变量：

```
MULTI_QI m_arrMultiQI [6];
```

同时在 void CAsynOpcDlg::OnStart()函数体内代码修改如下：

```
/*启动 OPC 服务器，添加组对象，添加项,设置回调*/
void CAsynOpcDlg::OnStart()
{
    // TODO: Add your control notification handler code here
    HRESULT         r1;
    CLSID       clsid;
    LONG        TimeBias = 0;
    FLOAT       PercentDeadband = 0.0;
    DWORD           RevisedUpdateRate;
    CString     szErrorText;
    m_ItemResult = NULL;
    //    初始化 COM 库
    r1 = CoInitializeEx(NULL,COINIT_MULTITHREADED);
    if (r1 != S_OK)
    {    if (r1 == S_FALSE)
        {
            AfxMessageBox("COM 库已经初始化");
        }
```

```
    else
    {
        AfxMessageBox("COM 库初始化失败");
        return;
    }
}
//  通过 ProgID,查找注册表中的相关 CLSID
r1 = CLSIDFromProgID(L"OPCDA.Opc.1", &clsid);
if (r1 != S_OK)
{
    AfxMessageBox("获取 CLSID 失败");
    CoUninitialize();
    return;
}

memset(m_arrMultiQI,0,sizeof(m_arrMultiQI));
m_arrMultiQI [0].pIID       = &IID_IOPCServer;
m_arrMultiQI [1].pIID       = &IID_IConnectionPointContainer;
m_arrMultiQI [2].pIID       = &IID_IOPCItemProperties;
m_arrMultiQI [3].pIID       =
&IID_IOPCBrowseServerAddressSpace;
m_arrMultiQI [4].pIID       = &IID_IOPCServerPublicGroups;
m_arrMultiQI [5].pIID       = &IID_IPersistFile;
COSERVERINFO tCoServerInfo;
ZeroMemory (&tCoServerInfo, sizeof (tCoServerInfo));

CString m_strRemoteMachine="127.0.0.1";
int nSize = m_strRemoteMachine.GetLength () * sizeof (WCHAR);
tCoServerInfo.pwszName = new WCHAR [nSize];
mbstowcs (tCoServerInfo.pwszName, m_strRemoteMachine,
nSize);
r1 = CoCreateInstanceEx (
        clsid,              // CLSID
        NULL,               // No aggregation
        CLSCTX_REMOTE_SERVER,
        // connect to local, inproc and remote servers
        &tCoServerInfo,                          //  remote
```

```
machine name
                sizeof (m_arrMultiQI) / sizeof (MULTI_QI),    //    number
of IIDS to query
                m_arrMultiQI);                                //  array  of
IID pointers to query

        // COM requires us to free memory allocated for [out] and [in/out]
        // arguments (i.e. name string).
        delete [] tCoServerInfo.pwszName;
        if (r1 == S_FALSE)
        {
            AfxMessageBox("创建远程 OPC 服务器对象失败");
            m_IOPCServer = NULL;
            CoUninitialize();
            return;
        }
        if (SUCCEEDED (m_arrMultiQI [0].hr))
        {
            m_IOPCServer = (IOPCServer *)m_arrMultiQI [0].pItf;
        }

        //添加一个 group 对象，并查询 IOPCItemMgt 接口
        r1=m_IOPCServer->AddGroup(L"grp1",
            TRUE,
            500,
            1,
            &TimeBias,
            &PercentDeadband,
            LOCALE_ID,
            &m_GrpSrvHandle,
            &RevisedUpdateRate,
             IID_IOPCItemMgt,
            (LPUNKNOWN*)&m_IOPCItemMgt);
        if (r1 == OPC_S_UNSUPPORTEDRATE)
        {
            AfxMessageBox("请求的刷新速率与实际的刷新速率不一致
            ");
```

```
}
else
if (FAILED(r1))
{
    AfxMessageBox("不能为服务器添加 group 对象");
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    CoUninitialize();
    return;
}

//  为 AddItem 定义 item 表的参数
m_Items[0].szAccessPath         = L"";
m_Items[0].szItemID             = szItemID;
m_Items[0].bActive              = TRUE;
m_Items[0].hClient              = 1;
m_Items[0].dwBlobSize           = 0;
m_Items[0].pBlob                = NULL;
m_Items[0].vtRequestedDataType = 0;
r1 = m_IOPCItemMgt->AddItems(1,
        m_Items,
        &m_ItemResult,
        &m_pErrors);
if ( (r1 != S_OK) && (r1 != S_FALSE) )
{
    AfxMessageBox("AddItems 失败");
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = NULL;
    m_GrpSrvHandle = NULL;
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    CoUninitialize();
    return;
}
else if(r1==S_OK)
{
    AfxMessageBox("AddItems()成功");
```

```
}

//   检测 Item 的可读写性
if (m_ItemResult[0].dwAccessRights != (OPC_READABLE +
OPC_WRITEABLE))
{
AfxMessageBox("Item 不可读，也不可写,请检查服务器配置");
}

//查询 group 对象的异步接口
r1     =     m_IOPCItemMgt->QueryInterface(IID_IOPCAsyncIO2,
(void**)&m_IOPCAsyncIO2);
if (r1 < 0)
{
    AfxMessageBox("IOPCAsyncIO2 没有发现，错误的查询!");
    CoTaskMemFree(m_ItemResult);
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = NULL;
    m_GrpSrvHandle = NULL;
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
    CoUninitialize();
    return;
}
//获得 IOPCGroupStateMgt 接口
r1=m_IOPCItemMgt->QueryInterface(IID_IOPCGroupStateMgt,
(void**) &m_IOPCGroupStateMgt);

if (r1 != S_OK)
{
    AfxMessageBox("IOPCGroupStateMgt 接口没有找到");
    CoTaskMemFree(m_ItemResult);
    m_IOPCItemMgt->Release();
    m_IOPCItemMgt = 0;
    m_GrpSrvHandle = 0;
    m_IOPCServer->Release();
    m_IOPCServer = NULL;
```

```
            CoUninitialize();
            return;
        }
        OnCheckActivategroup();
        // 建立异步回调
        CComObject<COPCDataCallback>* pCOPCDataCallback; // 回调
        对象的指针

        //通过 ATL 模板创建回调对象的实例

        CComObject<COPCDataCallback>::CreateInstance(&pCOPCData
        Callback);

        // 查询 IUnknown 接口
        LPUNKNOWN pCbUnk;
        pCbUnk = pCOPCDataCallback->GetUnknown();

        // 建立一个服务器的连接点与客户程序接收器之间的连接
        HRESULT hRes = AtlAdvise( m_IOPCGroupStateMgt,     //     [in]
IUnknown Interface of the Connection Point
                                  pCbUnk,                  //          [in]
IUnknown Interface of the Callback object
                                  IID_IOPCDataCallback,// [in] Connection
Point ID: The OPC Data Callback
                                  &m_dwAdvise              // [out] Cookie
that that uniquely identifies the connection
                                );

    if (hRes != S_OK)
    {
        AfxMessageBox("Advise 失败!");
        CoTaskMemFree(m_ItemResult);
        m_IOPCItemMgt->Release();
        m_IOPCItemMgt = 0;
        m_GrpSrvHandle = 0;
        m_IOPCServer->Release();
        m_IOPCServer = NULL;
```

```
        CoUninitialize();
         return;
        }
}
```

我们对上述代码与本地客户程序代码的的不同之处做解释。

• 指定服务器计算机名字的方法。它将载入到 COSERVERINFO 结构体。

• 通过 MULTI_QI 指定接口。

• 调用 CoCreateInstanceEx()代替 CoCreateInstance()。这将包括有一些不同的参数和一个称为 MULTI_QI 的结构体。

通过 COSERVERINFO 指定服务器，进行远程 DCOM 连接时，你必须指定服务器计算机的名字。计算机的名字可以是一个标准的 UNC 计算机名字或者是一个 TCP/IP 地址。在这里我们以“127.0.0.1”为远程机器 IP 地址。

```
CString m_strRemoteMachine="127.0.0.1";
int nSize = m_strRemoteMachine.GetLength () * sizeof (WCHAR);
tCoServerInfo.pwszName = new WCHAR [nSize];
mbstowcs (tCoServerInfo.pwszName, m_strRemoteMachine, nSize);
```

这个地址或者服务器名字被载入到 COSERVERINFO 结构体中，这个结构体需要一个指向宽字符（wide-character）的指针以得到服务器的名字，我们这里首先获得了 strRemoteMachine 的长度，然后通过 mbstowcs 转换为需要的宽字符。

本地客户程序通过调用 CoCreateInstance 得到一个接口指针。对于 DCOM 来说，我们需要使用扩展的版本 CoCreateInstanceEx。这个扩展的函数对于本地的 COM 服务器调用也是适用的。CoCreateInstanceEx 有几个重要的区别。首先，它可让你指定服务器的名字；第二（将第二改为“其次”），通过一次调用，它可让你得到超过一个的接口。在这里需

要注意的是 CoCreateInstanceEx 可一次返回超过一个接口。它通过传送 MULTI_QI 结构体的一个数组来做到这一点。数组的每个元素指定了一个单一的接口。CoCreateInstanceEx 将会填入到数据请求中。另一个要注意的是第三个参数 CLSCTX_REMOTE_SERVER，代表远程服务器。

MULTI_QI 结构体包含有三部分的信息：一个到 IID 的指针，返回接口的指针，一个 HRESULT。MULTI_QI 结构体定义如下：

```
typedef struct tagMULTI_QI
{
   // pass this one in
   const IID *pIID;
   // get these out (must set NULL before calling)
   IUnknown *pItf;
   HRESULT hr;
} MULTI_QI;
```

在例子中，我们在 CasynOpcDlg 定义中添加了保护变量：

```
MULTI_QI m_arrMultiQI [6];
```

在 void CAsynOpcDlg::OnStart()中，

```
memset(m_arrMultiQI,0,sizeof(m_arrMultiQI));
m_arrMultiQI [0].pIID          = &IID_IOPCServer;
m_arrMultiQI [1].pIID          = &IID_IConnectionPointContainer;
m_arrMultiQI [2].pIID          = &IID_IOPCItemProperties;
m_arrMultiQI [3].pIID          =
&IID_IOPCBrowseServerAddressSpace;
m_arrMultiQI [4].pIID          = &IID_IOPCServerPublicGroups;
m_arrMultiQI [5].pIID          = &IID_IPersistFile;
```

我们通过调用 memset()，将整个数组设置为 0。 接着我们就会将一个个指针填入到 pIID 元素中，该指针指向我们选用接口的接口 GUID（ IID ）。 如 m_arrMultiQI [0].pIID 指向 IID_IOPCServer 。CoCreateInstanceEx 的第 5 个参数定义了数组的大小。在这里定义了 6 个数组，与 6 个接口相对应。

调用 CoCreateInstanceEx 将会计算 MULTI_QI 数组的大小。与大多数

的 COM API 函数一样，CoCreateInstanceEx 返回一个 HRESULT。该函数可返回以下 HRESULT 中的任意一个。

S_OK：所有接口均被返回；

CO_S_NOTALLINTERFACES：至少有一个接口被返回，但有一些失败了；

E_NOINTERFACE：没有接口被返回；

为了知道哪个接口失败了，你可以检测 qi 数组中的 HRESULT。

```
if (SUCCEEDED(hr))
 if (SUCCEEDED(arrMultiQI [0].hr)) {
 // pItf pointer is OK.
```

为了获得 m_IOPCServer，即 IID_IOPCServer 接口指针，需要采用如下代码：

```
if (SUCCEEDED (m_arrMultiQI [0].hr))
{
    m_IOPCServer = (IOPCServer *)m_arrMultiQI [0].pItf;
}
```

首先检查是否获得接口成功，然后获得 IID_IOPCServer 指针。然后在程序中可以使用了。

大家可以看到，以后的代码与本地客户的完全一样，没有什么特殊之处。对于跨网络的DCOM来说，错误码的识别是非常重要的。HRESULT中包含有许多你需要的信息，这些信息对于跟踪网络错误是特别有用的。如果你第一次编写 DCOM 远程访问程序，你可能会遇到很多问题，慢慢的处理，直到解决它们。

附录：编程常用函数及名词简介，

****

COM 库常用函数

初始化函数

CoBuildVersion 获得 COM 库的版本号

CoInitialize COM 库初始化

CoUninitialize COM 库功能服务终止

CoFreeUnusedLibraries 释放进程中所有不再使用的组件程序

GUID 相关函数

IsEqualGUID 判断两个 GUID 是否相等

IsEqualIID 判断两个 IID 是否相等

IsEqualCLSID 判断两个 CLSID 是否相等

CLSIDFromProgID 字符串组件标识转换为 CLSID 形式

StringFromCLSID CLSID 形式标识转化为字符串形式

IIDFromString 字符串转换为 IID 形式

StringFromIID IID 形式转换为字符串

StringFromGUID2 GUID 形式转换为字符串

对象创建函数

CoGetClassObject 获取类厂对象

CoCreateInstance 创建 COM 对象

CoCreateInstanceEx 创建 COM 对象，可指定多个接口或远程对象

CoRegisterClassObject 登记一个对象，使其它应用程序可以连接到它

CoRevokeClassObject 取消对象的登记

CoDisconnectObject 断开其它应用与对象的连接

内存管理函数

CoTaskMemAlloc 内存分配函数

CoTaskMemRealloc 内存重新分配函数

CoTaskMemFree 内存释放函数

CoGetMalloc 获取 COM 库内存管理器接口

与 COM 接口有关的一些宏

DECLARE_INTERFACE(iface)

声明接口 iface，它不从其他的接口派生

DECLARE_INTERFACE_(iface, baseiface)

声明接口 iface，它从接口 baseiface 派生

STDMETHOD(method)

声明接口成员函数 method，函数返回类型为 HRESULT

STDMETHOD_(type, method)

声明接口成员函数 method，函数返回类型为 type

OPC 名词

Data Access	数据获取
History Data Access	历史数据获取
AE	报警和事件

Asynchronous Access	异步数据访问
Automation Interface	自动化接口
Branch	枝
Cache	缓存
Client	客户
Handle	句柄
Custom Interface	自定义接口
DeadBand	死区
Flat	平面
Hierarchical	树型
Leav	叶
Locale ID	区域标识符
Browser	浏览
Group	组
Item	项
Server	服务器
Wrapper DLL	包装 DLL
Proxy-Stub DLL	代理－存根 DLL
Public Group	公用组
Quality	品质
Refresh	刷新
Subscription	回调
Synchronous Access	同步数据访问
TimeBias	时间偏差
TimeStamp	时间戳，采样时间

Transaction ID	事务标识符
XML	可扩展标记语言
OPCReadable	可读
OPCWritable	可写
OPCRunning	OPC 服务器正在运行
OPCFailed	OPC 服务器由于异常停止
OPCNoConfig	OPC 服务器正在运行，但没有配置
OPCSuspended	OPC 服务器正处在暂时停止状态
OPCTest	OPC 服务器正处于试验模式下运转
OPCDisconnected	服务器对象没有连接任何实际的 OPC 服务器

OPC 数据类型

数据类型	字节数	说明
VT_I1	1	带符号 1 字节整数
VT_I2	2	带符号 2 字节整数
VT_I4	4	带符号 4 字节整数
VT_R4	4	4 字节实数
VT_R8	8	8 字节实数
VT_CY	8	8 字节货币
VT_DATE	8	日期
VT_BSTR	可变	字符串
VT_BOOL	1	0:FALSE,-1:TRUE
VT_UI1	1	无符号 1 字节整数

VT_UI2	2	无符号 2 字节整数
VT_UI4	4	无符号 4 字节整数
VT_EMPTY		不指定类型

OPC 品质类型

类型	值	说明
OPC_QUALITY_MASK	0xC0	
OPC_STATUS_MASK	0xFC	
OPC_LIMIT_MASK	0x03	
OPC_QUALITY_BAD	0x00	坏
OPC_QUALITY_UNCERTAIN	0x40	不确定
OPC_QUALITY_GOOD	0xC0	好
OPC_QUALITY_CONFIG_ERROR	0x04	配置错误
OPC_QUALITY_NOT_CONNECTED	0x08	没有连接
OPC_QUALITY_DEVICE_FAILURE	0x0c	设备错误
OPC_QUALITY_SENSOR_FAILURE	0x10	
OPC_QUALITY_LAST_KNOWN	0x14	
OPC_QUALITY_COMM_FAILURE	0x18	通讯错误
OPC_QUALITY_OUT_OF_SERVICE	0x1C	
OPC_QUALITY_LAST_USABLE	0x44	
OPC_QUALITY_SENSOR_CAL	0x50	
OPC_QUALITY_EGU_EXCEEDED	0x54	
OPC_QUALITY_SUB_NORMAL	0x58	
OPC_QUALITY_LOCAL_OVERRIDE	0xD8	
OPC_LIMIT_OK	0x00	

OPC_LIMIT_LOW	0x01
OPC_LIMIT_HIGH	0x02
OPC_LIMIT_CONST	0x03

参考文献

****

1．OPC DA2.05 规范

2．GE OPC SERVER 源程序（OPC 规范 1.0）

3．http://www.csdn.net 的 COM 文章

4．《COM 原理与应用》学习笔记（网上一位仁兄的大作）

5．《COM 原理与应用》

6．MSDN

7．《VC++技术内幕》

8．《OPC 应用程序入门》

NA 系列可编程控制器简介

****

1. 南大傲拓科技（北京）有限公司简介

南大傲拓科技（北京）有限公司（Nanda Automation Technology Beijing Co., Ltd.）是一家致力于工业自动化领域的产品研发、生产销售及服务的高新技术企业。

南大傲拓以客户需求为第一要素，高度重视创新性设计，研发和生产性能可靠、品质精良、技术先进的前沿工控产品，并提供贴近用户需求的行业自动化解决方案。同时，公司具有丰厚的应用软件开发经验，积极与科研院所和行业用户紧密合作，联手开发基于行业自动化解决方案的企业管理信息系统。

南大傲拓立志高举国产 PLC 大旗，打造自主民族品牌，为振兴装备制造业、提高工业自动化和企业信息化的整体水平而努力奋斗！

2. NA 系列可编程控制器

NA 系列智能可编程控制器（简称 NA-PLC）由南大傲拓科技（北京）有限公司自主设计与研发，汲取了国际主流 PLC 的成功经验、改进了其不足之处、瞄准了当今 PLC 的最新发展方向，采用了计算机、通信、电子和自动控制等方面的国际先进技术，在 CPU 操作系统、I/O 信号处理、网络通讯、软件开发及生产工艺等方面具有优良性能，整合了 DCS 及 PLC 优点于一身，是适合离散过程控制的智能可编程控制器。NA-PLC 是对传统 PLC 功能的极大提升，其组网的灵活性、系统平台的开放性、编程软件的标准性以及智能性可使复杂的控制项目得以完

美地实现。

NA-PLC 包括 NA200、NA400、NA600 三大系列的产品，以适应不同的工业控制领域，可以很方便地通过以太网与其它工业控制监控软件构成控制系统。

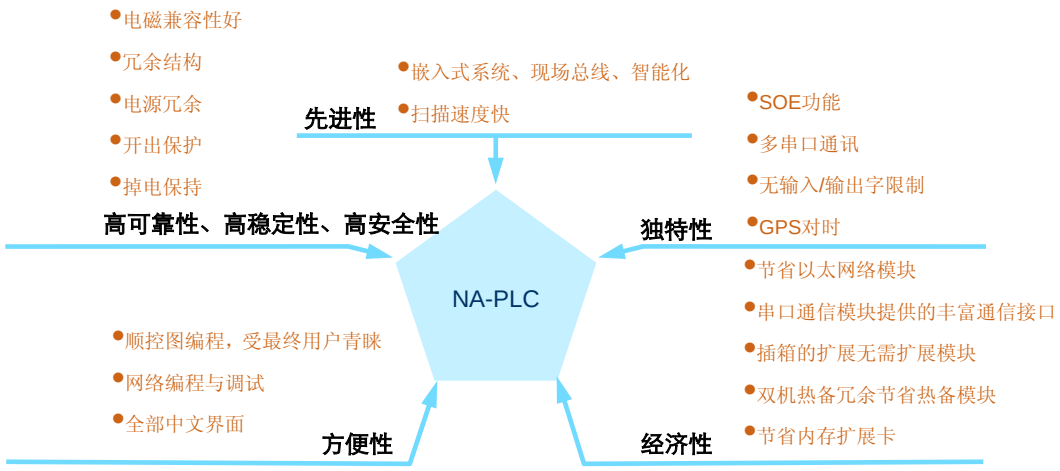
产品定位：

NA200——SIEMENS S7200

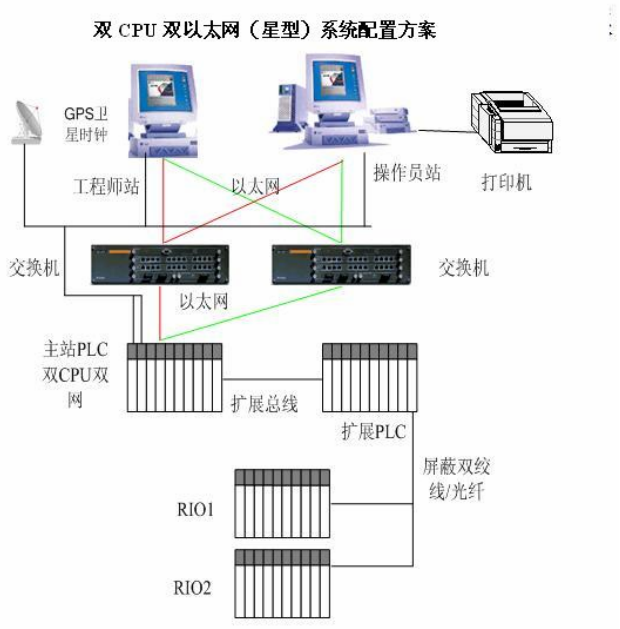
NA400——SIEMENS S7300

NA400——SIEMENS S7400

产品特点：



NA200：微型一体化 PLC，CPU 集成 IO，模块类型丰富，集成电源可作为分布式 IO。



3. 联系方式

网址：http://www.nandaauto.com

EMAIL：

技术支持：support@nandaauto.com

市场合作：sales@nandaauto.com

诚征代理商、合作伙伴、工程服务商、电气成套商！

联系人:司纪刚

南大傲拓科技（北京）有限公司

QQ：41063473

手机：13814097755

MSN：sijigang@hotmail.com

EMAIL: sijg@nandaauto.com