返回总目录

目　　录

第 9 章　C++BUILDER 开发 CORBA 程序扼要

第 10 章　CORBA 编程实例解析

第 11 章　CORBA 聊天室

第 1 章　Internet 需要 CORBA

本章内容提要：

- 对象管理组织——*OMG*
- 为什么需要 *CORBA*
- *CORBA* 依赖的技术
- *CORBA* 概述

1.1　对象管理组织 OMG 提出了 CORBA

1990 年 11 月，对象管理组织（Object Management Group）在《对象管理体系指南》一书中首次提出“公共对象请求代理结构”（Common Object Request Broker Architecture），以后被人们简称为 CORBA。

对象管理组织 OMG 的 800 多名成员遍布世界各地，其中 5%来自亚洲、澳大利亚、非洲及南美洲，30%来自欧洲，其余来自美国本土。他们包括 DEC、Microsoft、Netscape、Oracle、Novell、IBM、Inprise、Informix、Iona、Hewlett-Packed、Rogue wave、Sybase 、Sun 等许多著名公司。

不过，OMG 并不开发软件，仅仅制定指标。OMG 成员每年聚会 6 次，提出自己对各种软件标准的建议。这些提议在经过具有“协作资格”成员充分地讨论、修改后，由 OMG 董事会公布为 OMG 官方标准，并由标准倡导者在一年内推出标准的相关实现产品。任何公司决定采用 OMG 标准时，既不需要 OMG 的认可，也不需要交纳任何费用。

CORBA 标准的提出，是软件界的一场革命。

1.2　CORBA 的用途

Internet 使计算机联结起来，CORBA 则使应用软件联结起来。没有操作系统的计算机几乎没有用处，没有应用软件相互集成的 Internet 也很难显示出互联网的终极魅力。在 Internet 上集成应用软件，就是通常所说的“分布式软件开发”。CORBA 正是为满足这种源自 Internet 的需要，实现软件全方位集成而设计的。

分布式软件开发需要解决一系列的兼容问题，包括以下五个跨越：

- 跨平台。未来的软件将分布在各种机型的平台上：大型机、PC 机、笔记本、带程序的电视机、录像机、传感器、报警器、各种 DSP、PDA 等等。
- 跨操作系统。计算机世界的操作系统种类繁多，有称霸 PC 机的 Windows 系列、源远流长的 Unix 及其变种、蓄势待发的 Linux、经典的 Solaris OS 等，分布式软件开发必须正视这一现实问题。
- 跨语言。至今为止，还没有一种通用、万能的计算机编程语言。Java 可谓灵巧，但目前还是不适合于开发用户界面；Delphi、C++Builder 后来居上，却缺乏 Java 跨平台、跨操作系统的兼容性；Visual Basic 虽然开辟了可视化编程的先例，却无法胜任大型软件的开发；Visual C++则神情严肃、过于呆板苛刻，缺乏 C++Builder 的平易近人和灵活便利……未来的世界，允许混合编程，使每种语言都在最擅长的领域一显身手。
- 跨协议。Internet 是一个异质结构的网络，在不同的区域可能具有不同的网络结构、传输协议，为了使软件运行时具有数据、方法共享性，操作透明性，集成软件时

必须考虑协议不同带来的不便。

- 跨版本。用户对软件功能的需求总是在逐步增加，每次变化都会要求软件工程师重新编写程序模块。分布式软件开发会给软件的版本更新带来不便——应该让多少客户进行 Update？又会导致多少软件模块发生连锁性重写？因此，在 Internet 上集成软件必须实现版本的透明性。

到目前为止，CORBA 是用来解决以上五个跨越问题的唯一方案。

作为分布式软件的用户，CORBA 可以使我们得到以下好处：

- 我们可以使用各种信息、数据，无论它们分布在数字世界的哪个角落。
- 我们可以对信息、数据进行各种自动化处理，无论它们需要哪个公司的哪个软件。这种处理是无缝集成的，就好象使用专门编制的软件一样。
- 我们可以随意更改事务处理流程，实行物业动态管理。这种改变、革新对办公室自动化、电子商务的现有设施影响很小。我们可以最大限度的重用已经存在的软件、硬件和数据信息。
- 我们可以使用、控制各种数字化的设施：嵌入程序的智能电视、录像机、报警系统、传感器以及各种几乎类似“职业选手”的“个人数字助理”——PDA 等只要这些带程序的嵌入式数字化设备接入了互联网。

作为分布式软件的开发人员，CORBA 可以使我们获益如下：

- 混合编程。我们可能偏好某种编程语言，嗜好某种开发工具，但我们需要与具有不同癖好的伙伴合作开发大型软件。
- 丰富的编程元素。任何符合 CORBA 的软件模块一旦问世，就会成为可以被重用的资源。无论何时，无论何地。
- 高效率的开发手段。使用 CORBA 方式编程，不但软件模块的重用十分便利，而且软件模块天生具有无缝集成性。第二个 CORBA 软件开发人员就不再是“高楼万丈平地起”，软件开发只需要解决还没有被别人解决过的问题，编写别人还没有编写过的代码。
- 版本无关性。使用 CORBA 方式编程，版本不但具有向下兼容性，而且具有向上兼容性。也就是说，Version2.0 的用户自然能够获得 Version1.0 的功能，同时，Version2.0 的用户也可以调用未来 Version3.0 的新功能。

实际上，CORBA 将在 Internet 上实现软件的即插即用。

1.3 CORBA 采用的技术

CORBA 首先是一种编程技术，是吸收了软件界面向对象技术、分布式计算技术、多层体系结构技术以及接口技术的一种综合技术。

1.3.1 CORBA 采用了面向对象技术

CORBA、Java、COM/DCOM 都采用了面向对象技术，而且它们都已经或正在将面向对象的概念上升到比“类”更加高的一个级别——“组件”（Component）。

面向对象技术涉及许多抽象的定义，对此我们不想过多讨论。但是，面向对象技术是 CORBA 存在、实现的形式，因此简要说明如下：

- 类。类是一个程序集合，它记录了被表达概念或实体的共同特性及共同操作、处理过程、函数。前者形成类的数据，后者成为类的方法。比如，表示客户的类至少包括“客户名称”这一共同特性，称为类的数据。而告诉别人自己的名字以及改变自己名字的函数、过程被认为是“客户类”的方法。
- 对象。对象是类的具体实例，给类中的数据变量赋予确定的取值便得到该类的一个对象。比如，“张老三”、“王老五”就是“客户类”实例化后产生的两个不同对

象。编写程序时，考虑的是类；运行程序时，处理的是对象。不过，在许多场合，人们并不刻意区分类和对象这两个概念。

- 封装。封装是实现类的机制，确保了不同类之间的相对独立性。比如，通过封装，“客户类”和“商品类”相互独立。此时，“客户类”在未授权的情况下不能直接使用“商品类”的数据；“商品类”的修改（增加、更改类中的数据或方法）也不会直接影响“客户类”。
- 继承。继承反映了不同类之间的一种派生关系。比如，电视机首先是一种商品，具有商品的所有特性，其次又含有指定的尺寸、体积、接收装置；遥控电视机也具有电视机的全部特性，同时还增加了遥控设备。通过继承，可以实现代码重用。与以往的子程序模块相比，继承保证了代码重用时的逻辑关系，提高了可移植性和安全性——通过确定继承关系，目前正在编写的代码将与能够重用的代码自动联系起来。类之间还存在多重继承关系。比如，“电视机类”可以从“商品类”和“无线电设备类”共同派生出来，表示电视机是一种作为商品出售的无线电设备。
- 重载。重载体现了类内部及派生类之间的差别，包括函数重载、操作符重载。函数重载可以使类中同名方法具有不同操作。比如，商品类与电视机类中都可以有一个称为 Show 的方法，前者可以在屏幕上显示商品的名称，而后者则在屏幕上直接绘出电视机的外形。客户类还可以有两个带有不同参数的 Name 方法，当参数为计算机屏幕时，在屏幕上显示自己的名字；当无参数时，通过多媒体读出自己的名字。另外，加减乘除等运算符号均可以借助操作符重载在各个类中实现不同的操作。
- 多态。多态反映了类中方法在实际调用过程中呈现的随机性和不确定性。如果发现商品是电视机，我们可以附带赠送一年的电视报；如果发现商品是 DVD，赠送物品则变为一盒影碟不过无论是哪种商品，都需要付款。这时，可以采用多态技术，在运行过程中动态地选择具体操作方法。多态包括虚函数、重置、虚继承等情况。

以上是面向对象技术的一些主要概念。

注意 CORBA2.0 是完全基于面向对象技术的。目前，CORBA3.0 正在朝着基于 Component 的方向发展。Component 可以认为是更高级别上的“类”。不过，真正的类往往用同一种语言实现继承、重载、多态；Component 则可以在不同语言中实现。另外，Component 具有属性、事件。实际上，Component 是经过包装处理的一组类，阅读完本书就会理解这一点。COM/DCOM 是基于 Component 的一种分布式编程技术。本书将不断对比 CORBA 与 COM/DCOM 的差别。

1.3.2 CORBA 采用了分布式计算模型

CORBA、COM/DCOM 都采用了分布式计算模型。IBM 曾经认为，计算机世界是一个“英雄创造历史”的世界，他们仅仅重视大型计算机——处理和控制都高度集中的主机。IBM 的失误造就了 Microsoft 和 Intel 两个巨人。现在，计算机世界都垂青于分布式计算模型——资源、功能、任务、控制等统统分布的系统。

分布式计算模型具有以下特点：

- 分布性。数据、处理、控制并不驻留在某个站点，而是比较均匀的分布在整个互联网上。因此，功能、任务也因而分布开来。
- 并行性。不同任务可以并行执行。既可以在同一站点并行执行，也可以通过数据、计算迁移实现多站点并行执行。
- 透明性。分布式计算隐藏许多了内部实现细节，包括物理位置、并发控制、错误处理。这些隐藏实现的是应用性能上的透明性。

- 共享性。软件、硬件的各种资源高度共享。
- 强健性。数据、处理和控制的分布，使系统故障得以分散。一般情况下，局部故障不会给整个系统造成太大影响。同时，整个系统的整体容错性、可靠性也相应加强。

实际上，CORBA 从一开始就是为分布式软件开发、集成而设计的。

1.3.3 CORBA 采用了多层体系结构

CORBA、COM/DCOM 都采用了多层体系结构。

电子商务软件一般可以划分为三大模块：用户界面、商务规则、信息数据管理。而电子商务软件的开发却已经经历了单层体系结构、双层体系结构、客户/服务器体系结构、三层体系结构、多层体系结构几个阶段。目前，在如何划分多层体系结构上还存在分歧。这些体系结构之间的差异和分歧在于如何定位、调度、集成三大模块。

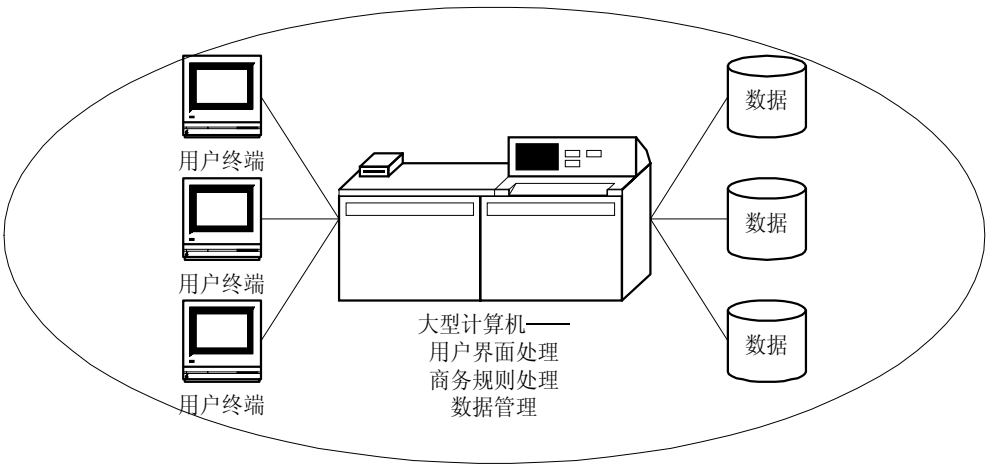

图 1.1　单层体系结构

图 1.1 表示单层体系结构。用户终端仅仅用来显示、接收操作信息，大型计算机将三大模块处理集于一身，数据被存放到指定位置。虽然大型计算机终日忙忙碌碌，可用户还是觉得它行动迟缓；而且，任何用来完善用户界面的代码都会被认为是一种奢侈和浪费。

随着 PC 机的发展，人们将用户界面处理部分的代码移入了用户终端，形成了双层体系结构。后来，PC 机性能更加卓越，人们甚至将一部分商务规则处理代码也移入了用户终端，形成了客户/服务器体系结构，如图 1.2 所示。

客户/服务器体系结构在实际应用中不断受到两个事实的挑战：

- 电子商务规则日益复杂。软件开发人员不可能或者很难再将部分商务规则移入小型工作站、高性能 PC 机。
- 电子商务规则经常变化。软件开发人员需要不断更新电子商务软件算法，而且应该及时完成。

为了有效解决上述问题，人们决定采用三层体系结构，主要包括下列三层：

- 用户界面层，仅处理图形用户界面。
- 商务规则层，仅处理商务规则。
- 数据管理层，仅实现数据管理。

与此同时，面向对象技术、组件（Component）技术相应诞生，类、组件成为人们进行软件开发的主要手段，成为构造三层体系结构软件的核心元素。

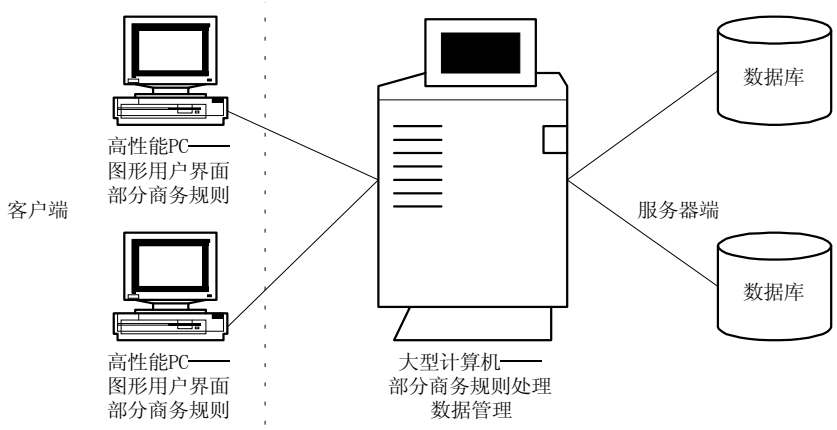

图 1.2　客户/服务器体系结构

接着，人们发现需要位于不同机器上的类、组件能够相互通信，这就导致了中间层——商务规则层的继续分化。它们从一台机器上移到了 Internet 上，相应产生了多层体系结构。也有人认为，多层体系结构就是三层体系结构。这种分歧并不重要，总之电子商务软件要“分布”进行。

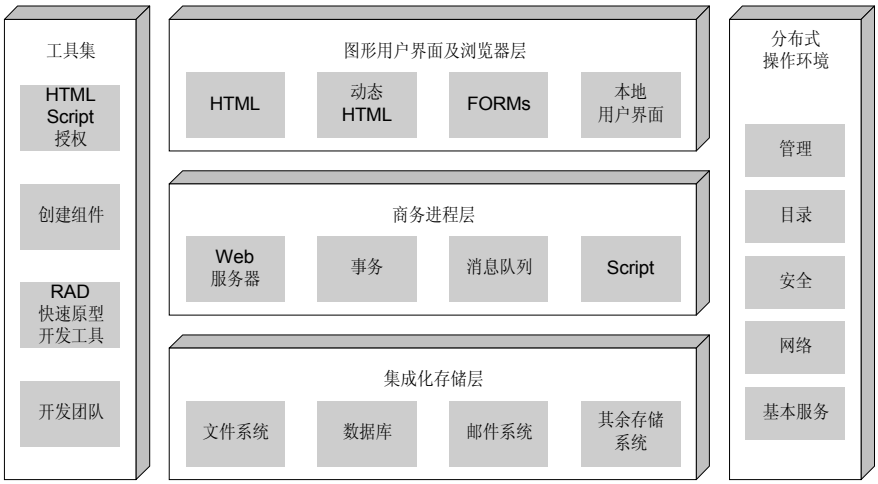

图 1.3　Microsoft 对于多层体系结构的理解

Microsoft 认为三层体系结构就是多层体系结构，包括客户层、组件层和数据管理层。它们配合 Microsoft 的各种产品，形成由 COM/DCOM 统一支配的分布式软件开发结构，如图 1.3 所示（就是现在流行的 ASP）。

CORBA 认为的三层或多层体系结构如图 1.4 所示。而 CORBA 本身是位于中间层的中间件（Middleware），是关于分布式对象、分布式组件的标准。

注意　CORBA 与 Microsoft 对于多层体系结构的理解在后两层产生了分歧。Microsoft 的第三层只有数据，其它服务都归入第二层；CORBA 的中间层更加关心的是对象的互操作性，因此，资源层依旧包括与数据有关的服务。这种分歧与支持这两种模型的软件公司所采取的经营策略密切相关。Microsoft 希望所有的客户都使用自家的产品，因此在第二层里囊括了全部服务，由 Microsoft Transaction Server 实现。而 CORBA 是公司联盟产物，希望各种服务由不同的公司提供。因此，在中间层更加关注产品的兼容、通用性，并认为商务规则可以通过对各种资源的有效集成来实

现。这些资源，可以来自地球上任何一个有智慧的厂家。作者认为，从面向对象技术角度出发，CORBA 似乎更为合理，因为无论是类、组件都符合这样一条规律：数据与数据有关的操作不能从逻辑上分离。也就是说，数据与它们的服务很难被划分到不同层。

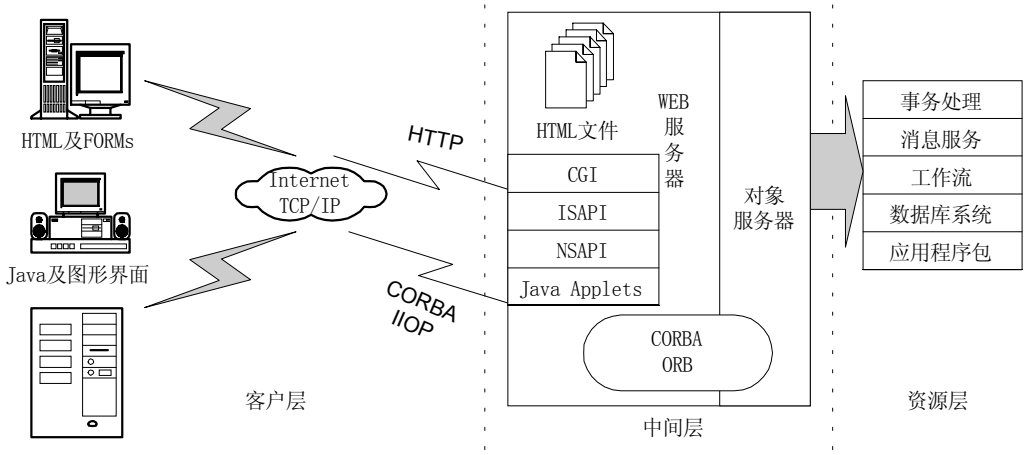

图 1.4　CORBA 对于多层体系结构的理解

1.3.4　CORBA 采用了接口技术

接口技术很容易与面向对象技术混淆，因为它们都有继承的概念。但是，面向对象技术主要从软件重用角度考虑问题；而接口技术主要从软件集成角度考虑问题。

接口技术主要用于解决两个问题：

- 如何提供一种清晰的设计标准，使得软件开发在总体设计、详细设计、具体编码阶段以及维护阶段都能够保持各自的独立性与一致性。人们希望采用接口技术以后，操作的声明与实现可以彻底分开进行，但是又不必为保持它们的一致性而付出过多代价。
- 如何提供一种包装方式，使得软件开发可以在不同程序语言中实现。人们希望采用接口技术以后，不论采用哪种编程语言，操作都可以被成功的激发、调用。其实，Component 就可以被看作是用接口包装的、跨语言的“类”。

实际上，软件中的接口技术与电器行业的接口技术有着非常相似的特点：不管厂家或软件开发人员是谁，都必需达到、完成接口中规定的指标及内容；不管厂家或软件开发人员采用哪种技术，从同一种接口中都应该获得相同的功能。

如果我们把接口当作一种规定和指标，就不难理解接口的以下一些特点：

- 接口本身不能包含数据变量。数据对应着状态，接口本身没有状态，因此也不能含有数据声明。但是，操作中有参数，因此，接口中也包括参数说明。
- 接口本身没有实现自己的能力。接口中所有操作、功能的实现都是通过其它软件模块、其它类来完成的。接口仅仅是一个“交流”的渠道，可以吞吐但无法生产。不过，接口可以强迫软件模块、类在编码实现中符合自身的规定。

1.4　CORBA 概述

CORBA 的最终目的是分布式软件集成。CORBA 既代表了一种软件开发模式、一种软件开发标准，也提供了软件开发必需的服务、可以使用的工具集合。因此，CORBA 整体上是由“对象请求代理”、“CORBA 服务”、“CORBA 工具集”与符合 CORBA 标准的

各种应用程序、对象一同综合形成的。如图 1.5 所示。

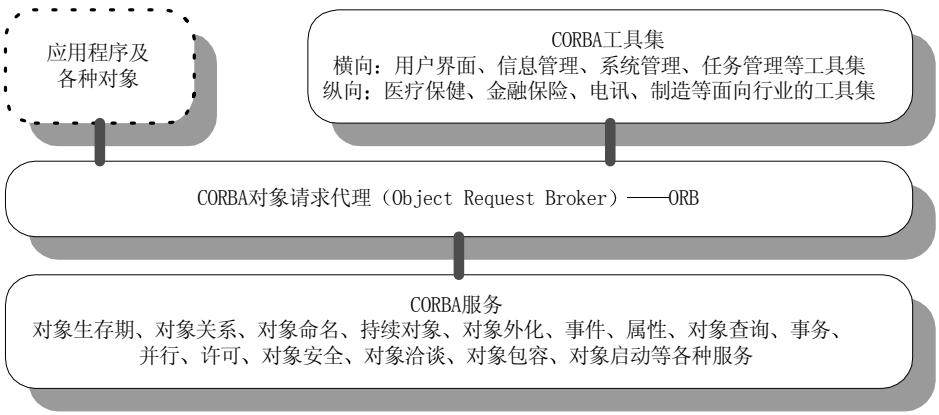

图 1.5　CORBA 的完整体系结构

其中，开发人员仅仅需要完成由虚线框起来的部分，其余均由 CORBA 本身提供。

1.4.1　CORBA 对象请求代理——ORB

在 CORBA 中，对象请求代理（Object Request Broker）被简称为 ORB。ORB 是联结应用程序、各种对象、CORBA 服务、CORBA 工具集的核心。当然，也正是 ORB 将这些构成 CORBA 图景的元素有序地分割开来，而这种分割是 CORBA 实现分布式软件集成和即插即用的全部秘诀。ORB 主要实现了以下分割（联系）：

- 对象方法、服务的“定义”与它们的“实现”之间的分割。通过接口定义语言 OMG IDL，我们可以获得规范、通用的对象方法、服务定义；借助 ORB，这些定义可以在任何编程语言、代码模块中真正实现。这种分割有助于进行具体软件编码互换、编程语言互换以及版本互换。
- 请求“客户”与响应“服务器”之间的分割。客户对其它对象方法、服务的请求并不直接传递给被请求服务器，而是转交给 ORB，由 ORB 监察服务器的位置，状态，决定服务绑定的方式。这种分割有助于对分布式对象进行跨平台、跨协议的“逻辑集成”，是一种“伪客户/服务器”方式。
- CORBA“基础设施开发者”与 CORBA“收益者”的分割。采用 CORBA 进行分布式软件开发的确是一个令人心醉的建议，但是，真正完全实现 CORBA 的所有标准难度很大。作为一般 CORBA 用户，我们都可以借助某种 ORB 产品来简化开发任务。ORB 就是某些软件公司为我们建造好的，进行 CORBA 软件开发的基础设施。

简而言之，ORB 可以帮助 CORBA 对象互相“理解”对方，可以在 CORBA 对象之间传递信息、请求，可以“管理”CORBA 对象之间的分布与集成。

1.4.2　CORBA 基本服务

CORBA 基本服务为基于 CORBA 的分布式软件开发提供了“对象”级别的服务。在 CORBA 中，有 16 个基本服务。

这些基本服务的英文、中文名称及简要说明分别如下：

- Lifecycle Service，对象生存期服务。提供与对象创建、移动、拷贝、释放等操作有关的服务。
- Relationship Service，对象关系服务。提供表达对象之间相互关系的服务。相信软件开发人员都明白，在“对象”世界中“乱搞关系”绝对会引起轩然大波，导致

系统混乱。

- Naming Service，对象命名服务。为对象提供姓名、别名、引用的服务。
- Persistent Object Service，持续对象服务。提供保持对象状态（数据）的服务。
- Externalization Service，对象外化服务。提供一种类似“流”的机制，提取、恢复对象的状态。
- Event Service，事件服务。事件服务允许对象注册、注销自己所关心的事件，并提供传递事件消息的信道。
- Object Query Service，对象查询服务。提供一种操作语言和服务，能够随意查询对象数据、属性、方法、分布等情况。
- Object Properties Service，对象属性服务。为对象提供动态属性的服务。
- Object Transaction Service，事务服务。提供事务处理服务：“要么统统提交，要么统统滚回”。这是电子商务对于“一手交钱，一手交货”的理解。
- Concurrency Service，并行服务。解决对象并行操作中资源冲突、资源共享、同步、死锁等问题的服务。
- Licensing Service，许可服务。许可服务允许用户只有在开发者认可的情况下才能使用对象，获取服务。
- Trader Service，对象洽谈服务。如果你知道你需要的服务，但不知道哪个是最适合的服务器、哪个是最经济的对象，就可以使用对象的洽谈服务。
- Security Service，对象安全服务。提供安全可靠的对象服务。
- Secure Time Service，对象时间服务。时间服务允许对象获取当前时间、确定事件发生顺序、计算时间间隔、实现基于时间的各种业务。
- Object Collection Service，对象包容服务。为对象提供数组、队列、栈、集合等容器。
- Object Startup Service，对象启动服务。决定对象如何启动以及启动时的状态。

以上服务是软件集成时，各种对象可能需要的公共服务。

1.4.3 CORBA 工具集

CORBA 工具集就是 CORBA Facilities。它们为基于 CORBA 的分布式软件开发提供了“应用”级别的服务。比如，在办公室自动化领域，我们需要功能强大的文字处理功能、图形处理功能。软件开发人员可以从 CORBA 工具集中选择一些相关的工具，在此基础上开发应用软件，从而忽略繁杂的文字、图形底层处理细节。

CORBA 工具集没有统一的标准，它们多少可以调用 CORBA 基本服务实现。但是，这些工具集合使基于 CORBA 的分布式软件开发不必总是“从头开始”，不必考虑一些过于原始、基本的问题。实际上，CORBA 工具集是 CORBA 程序库。

CORBA 工具集被分为横向工具集（horizontal facility）和纵向工具集（vertical facility）。

CORBA 横向工具集包括以下四部分内容：

- 用户界面（User Interface Common Facilities）。主要包括将数据转换为显示器、打印机等媒体支持的格式，管理图形用户界面等任务。
- 信息管理（Information Management Common Facilities）。主要包括建立各种信息模型、存储及恢复信息数据、支持多种信息模型的转换、支持不同数据格式互换、数据加解密等任务。
- 系统管理（Systems Management Common Facilities）。涉及内存管理、线程及进程池（Thread and process pooling）等能够降低分布式软件运行开销的任务。可以进一步分为服务质量管理（Quality of Service Management）、测量（Instrumentation）、数据集（Data Collection）、对象集合管理（Collection Management & Instance Management）、规划管理（Scheduling Management）以及系统安全管理（Security

Management）等内容。

- 任务管理（Task Management Common Facilities）。涉及工作流（Workflow）、代理（Agents）、规则管理（Rule Management）、对象自动化等任务。

CORBA 横向工具集涉及每个完整程序都需要考虑的问题：不论你正在开发什么软件，都需要在程序中解决一些类似的问题。比如，我们的应用程序都需要有友好的交互界面，需要通过多种方式显示同一份数据，需要驱动不同介质的存储器、光驱，需要管理特定的工作流、商务代理、商务规则以及实现对象自动化（类似 OLE Automation）等等。这些问题都可以从 CORBA 横向工具集中找到合适的解决方案。

CORBA 纵向工具集主要包括以下五个领域：

- 医疗保健（Healthcare）
- 金融服务（Financial Service）
- 电讯（Telecommunications）
- 电子商务（Business Objects）
- 制造（Manufacturing）。

CORBA 纵向工具集为工业界人士使用基于 CORBA 的软件提供了方便：在每个领域里，都有许多功能强大的应用软件。用户的业务既可以直接采用其中的工具，也可以在它们的基础上有所更新。每个领域中都没有特定的标准，只有可以方便地进行集成的各种服务和保证服务质量的各种许诺。

CORBA 工具集的目的十分清楚：当我们需要开发分布式软件时，可以从横向、纵向工具集中选择便捷的 CORBA 软件“部件”，快速地进行软件无缝集成。而且，任何功能强大、运行便利的 CORBA 对象都可以成为 CORBA 工具集中的侯选对象，甚至是 CORBA 工具集中的名牌对象。

1.5 本章小结

一个真正意义上的跨国组织——OMG 建议采用 CORBA 进行分布式软件开发。

因为 CORBA 可以最终解决跨平台、跨操作系统、跨语言、跨协议、跨版本等兼容问题，彻底地实现分布式软件集成。

CORBA 依赖于面向对象技术、分布式计算技术、多层体系结构技术和接口技术。

对于软件开发人员而言，CORBA 是一种分布式软件的开发方式，一种分布式软件的开发标准，也是一系列分布式对象公用的服务和一系列分布式软件开发的工具。

实际上，如果希望真正理解 CORBA 的奥妙，应该记住：CORBA 的精髓不在于如何开发软件，而在于为何能够这样开发软件。

第 2 章　CORBA 接口及接口定义语言 OMG IDL

本章内容提要：

- CORBA *的伪“客户/服务器”方式*
- CORBA *中的接口*
- *接口存根*（IDL Stub）
- *接口框架*（IDL Skeleton）
- CORBA *接口定义语言* OMG IDL *扼要*
- CORBA *伪对象*
- CORBA *接口定义语言与* Microsoft *接口定义语言比较*

2.1　CORBA 灵活的伪客户/服务器方式归功于 IDL

对象请求代理 ORB 是联结 CORBA 对象、CORBA 应用程序、CORBA 基本服务以及 CORBA 工具集的核心。在以 CORBA 方式开发分布式软件时，上述各个环节并不直接交互或者集成。ORB 将它们有序地分割，但又有机地联系起来。

如果从面向对象分析角度来考虑问题，CORBA 可以简化为如下表述：可集成的分布式对象（程序）和一个能够从逻辑上集成这些对象，被称做 ORB 的对象群，通过请求、响应的方式实现功能互联。如图 2.1 所示。

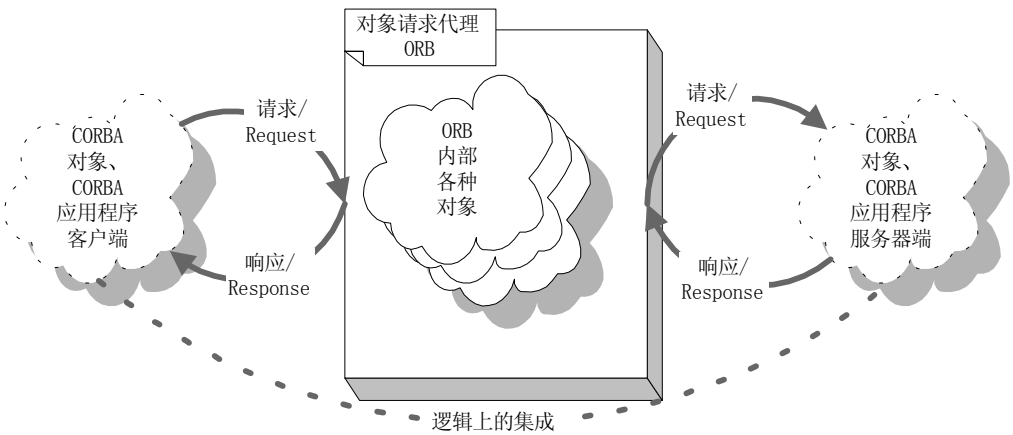

图 2.1　CORBA 的简化表述

分布式对象、程序之间的请求/响应方式就是传统的客户/服务器方式。但是，CORBA 中的请求“客户”和响应“服务器”之间并没有直接进行 “通信”。CORBA 客户向 ORB 发送请求，并从 ORB 接收响应；CORBA 服务器接收来自 ORB 的请求、调度，并向 ORB 发送响应。实际上，这是一种以 ORB 为中间件的伪客户/服务器方式。

CORBA 的这种伪客户/服务器方式具有更大的灵活性，可以满足分布式软件开发的下列现实需求：

- 允许分布式对象、程序具有灵活的角色。一般说来，客户总是提出执行某个任务的请求，服务器则进行相应的处理并发回有关响应。但是，一个对象、程序在现实中可能既是客户又是服务器。比如，全局数据库管理对象被动地接收、转储来自局部数据库

对象的更新数据时，明显是服务器；当全局数据库管理对象主动要求局部数据库对象提交更新数据并加以保存时，又可以看作是客户机。这种角色的歧义性在软件开发中经常发生。在 CORBA 中，只有客户——Client 与“对象实现”——Object Implementation，没有专门指明的服务器对象，就是为了消除这种歧义。

- 本地服务请求与远程服务请求的透明性。客户机与对象实现可以驻留在同一台机器上，可以分布在同一局域网内，也可以分散在互联网上。显然，针对具体情况，每次服务请求的具体操作可能不同。通过伪客户/服务器方式，这种不同由 ORB 监控处理。对客户而言，永远只有一种服务请求方式：CORBA 服务请求。
- 允许客户机与服务器之间保持灵活的对应关系。通过伪客户/服务器方式，客户机与服务器之间不需要保持——对应。当客户工作量较大时，可以请求多个服务器同时提供服务（比如，文件下载时，可以申请多个“对象实现”同时进行）；当“对象实现”吞吐量较大时，可以并行服务于多个客户。虽然 ORB 不能自动实现负载均衡，却为它提供了足够的支持。
- 允许客户机与服务器在运行时构造新型的、动态的服务请求。伪客户/服务器方式不要求客户机/服务器之间直接进行一问一答的请求和响应，因此也不需要在程序编写时绑定客户机/服务器，可以在程序运行时动态提出服务请求。
- 允许服务器以多种形式存在。服务器可以是包含在单个或者多个进程中的对象、程序；可以是需要向别的服务器提出请求的中介对象、程序；甚至可以是一段需要激发的程序代码。ORB 可以获得必要的信息，并采取不同手段激发有关的对象、服务。
- 允许客户/服务器之间以同步、异步方式通信。一问一答的请求/响应方式大多以同步通信方式进行。由于 ORB 的中介，CORBA 伪客户/服务器方式却能够进行异步通信。包括所谓的延迟同步（deferred synchronous）和单向请求（one-way request）。

在我们感慨 ORB 带来如此奇妙的伪客户/服务器方式的时候，应该明白，这些功能的实现，首先是因为 CORBA 客户与 CORBA“对象实现”采用了接口技术，并使用统一的接口定义语言——OMG IDL 来描述各自的数据、结构和行为。本章以下部分就将着重讨论 CORBA 的接口和 CORBA 的接口定义语言。

2.2 CORBA 中的接口

按照面向对象分析的方法，CORBA 中至少应该存在三组对象（或者说“类”）：CORBA 客户对象、ORB 对象、CORBA 对象实现（或者说“CORBA 服务器对象”）。它们的关系如图 2.1 所示。

根据第一章提到的 CORBA 的五个“跨越”目标，我们很容易对这三组对象提出以下要求：

- 我们希望选择不同厂家提供的，或者不同语言编写的 CORBA 客户对象
- 我们希望选择不同厂家提供的对象请求代理 ORB
- 我们希望选择不同厂家提供的，或者不同语言编写的 CORBA 对象实现

为了满足这三个要求，实现 CORBA 所期望的软件即插即用，我们发现，需要在以上三组对象之间引入新的对象，处理由于用户选择不同产品而导致的集成障碍，如图 2.2 所示。

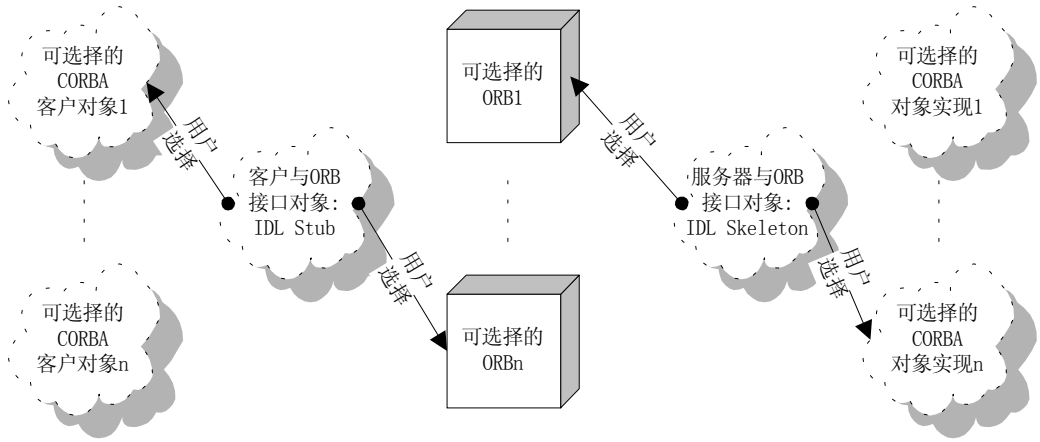

图 2.2　新引入的对象——客户与 ORB 接口对象、服务器与 ORB 接口对象

在 CORBA 标准中，客户与 ORB 接口对象被称为“接口存根”——IDL Stub；服务器与 ORB 接口对象被称为“接口框架”——IDL Skeleton。

这样，CORBA 客户对象与 CORBA 对象实现之间至少被三组对象隔离：接口存根、ORB、接口框架。这种隔离为软件集成带来了巨大的灵活性，但同时需要 CORBA 客户对象、接口存根、ORB、接口框架、CORBA 对象实现各个部分能够相互理解各自的数据、结构和行为。这一要求可以通过接口技术以及 CORBA 接口定义语言（Interface Definition Language）——OMG IDL 来实现。

OMG IDL 不是编程语言，不能用来编写具体的操作算法。但是，它可以用来定义对象能够进行的所有操作，包括全部输入、输出参数以及返回结果；甚至连可能产生的操作错误也不例外。

用 OMG IDL 定义对象就得到了 CORBA 接口。对于 CORBA 客户，这个接口意味着一种承诺：如果客户向 ORB 发送符合接口规定的请求，就一定会得到响应（出错、返回结果都是一种响应）；对于 CORBA 对象实现，这个接口意味着一种义务：软件开发人员必须实现接口规定的方法。

在这两种情况下，都需要 OMG IDL 能够和用户采用的具体编程语言建立映射关系。原因非常明显：在客户端，用户需要从具体编程语言中按照接口规定激发相应的对象实现；在对象实现端，ORB 应该能够根据接口规定激发软件开发人员采用具体编程语言编写的对象方法实现。综合起来考虑，就是“每一个 OMG IDL 句法都应该在具体的 CORBA 编程语言中找到一个对应”。当然，具体编程语言中可以包含不为 OMG IDL 所支持、理解的句法。

CORBA 接口对于接口存根和接口框架也相当重要。

2.2.1　CORBA 接口框架——IDL Skeleton

如图 2.3 所示，接口框架对象连接 ORB 和 CORBA 对象实现。

这里，我们可以进行两次选择：选择某种编程语言编写的对象实现；选择某个厂家提供的对象请求代理 ORB。两种选择都取决于用户的嗜好、侯选产品的性能、侯选方案的可行性。但是，被选中的对象实现和 ORB 只能且必须通过接口框架联系起来。

接口框架可以从 CORBA 接口中自动获得，如图 2.3 所示。

对于任何一个 CORBA 接口，软件开发人员可以借助自动、半自动或者手工的方式，获得从 OMG IDL 映射到自己所使用的具体编程语言中的对象声明，编写代码，完成对象实现。

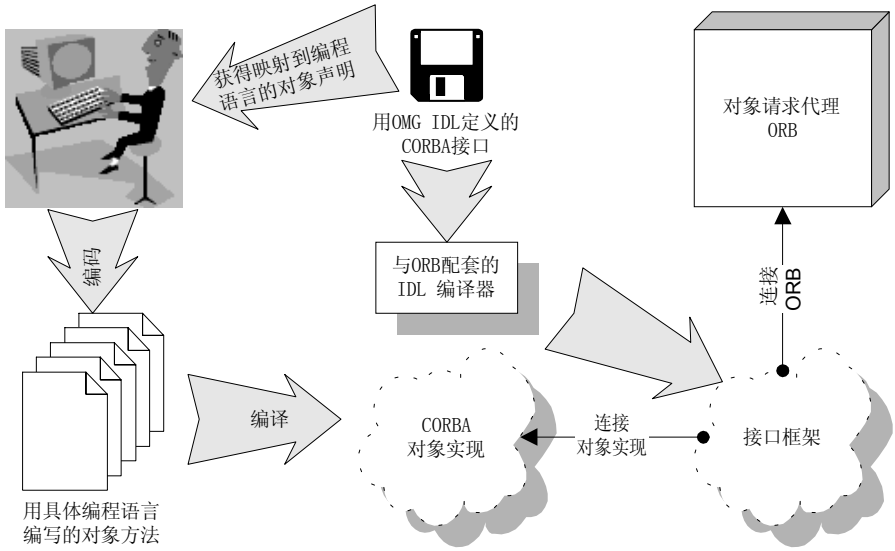

图 2.3　CORBA 接口经过编译自动生成接口 框架

另一方面，我们可以使用 IDL 编译器进行编译，自动获得与 CORBA 接口相应的接口框架。接口框架中既包括与指定 ORB 相连接的函数调用代码，也包括与有关“对象实现”相连接的函数调用代码。这样，接口框架就把“对象实现”与 ORB 连接起来。

OMG IDL 以及 OMG IDL 与 CORBA 编程语言的映射关系都是标准的，如果指定编程语言，不同程序员对于同一个 CORBA 接口将会获得相同的函数声明。因此，我们也希望任何 IDL 编译器在编译同一个 CORBA 接口时，对于同一种语言所产生的与“对象实现”相连接的函数调用代码相同。

这样，不论采用哪个厂家的 ORB，用同一种语言编写同一接口的对象实现时，只要完成具有相同声明的一套函数即可。正是由于这个原因，接口框架如何与“对象实现”连接是标准化的（主要是通过虚拟函数实现）。

接口框架如何与 ORB 连接没有被标准化，因为这样可以提高产品的灵活性。比如，我们在购买收录机时，仅仅要求耳机插孔外部符合通用标准，能够和各种立体声耳机相连；至于耳机插孔内部如何走线，大多数用户并不会关心。

接口框架的这种特性导致用户在选择 ORB 产品时略有限制：一旦选定哪个厂家的 ORB，就必须选择这个厂家的 IDL 编译器。ORB 与 IDL 编译器成对出现，配套使用。

2.2.2　CORBA 接口存根——IDL Stub

接口存根用来连接 CORBA 客户和 ORB。

如图 2.4 所示，当我们用 IDL 编译器编译 CORBA 接口获取“接口框架”时，同时也得到了“接口存根”。不过，“接口框架”中主要是一些函数调用代码，“接口存根”中主要是一些函数声明；“接口存根”告诉客户本 ORB 能够提供什么服务，“接口框架”告诉 ORB 应该请何种性质的“对象实现”来完成指定的服务。

与接口框架类似，接口存根与 ORB 如何连接也没有被标准化。

由于接口存根、接口框架与 ORB 如何连接取决于 ORB 生产厂家，我们可以把接口存根以及接口框架当作 ORB 对象群中两组专门的对象来对待。在以后章节中，我们将经常采用这种方式讨论问题。

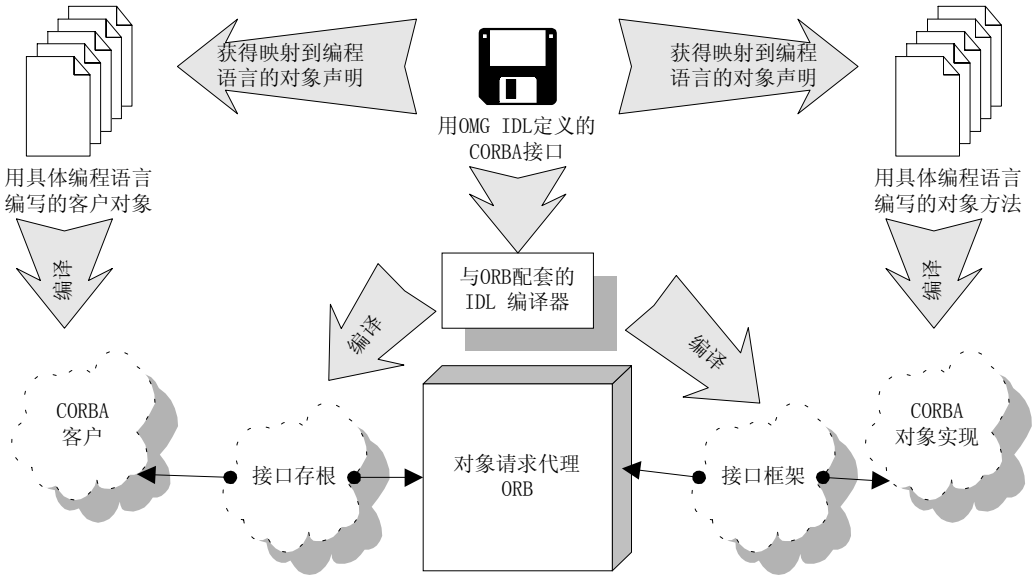

图 2.4　C ORBA 接口自动生成“接口存根”、“接口框架”

注意　在 CORBA 中，接口是用 OMG IDL 定义的。接口不能实现对象的方法，但规定了方法的名称、调用参数、返回值。接口的基本功能有两点：将对象方法标准化，并使得对象声明与实现分离；将真正的对象实现打包，使它能够被不同的语言激发、使用。这两个基本功能是 ORB 得以按伪客户/服务器方式集成软件的根本保障。在 COM/DCOM 中，也存在接口。这个接口是采用 Microsoft IDL 定义的，它的目的也是将对象方法标准化，并使得对象声明与实现分离；将真正的对象实现打包，使它能够被不同的语言激发、使用。CORBA 接口与 COM/DCOM 接口的不同之处在于两者采用了不同的接口定义语言，前者是 OMG IDL，后者是 Microsoft IDL。

2.3　OMG IDL 扼要

OMG IDL 是一种非常类似 C++的语言。

它的词法规则比 C++多了一些表明分布概念的关键词；它的语法规则是 C++的子集，不包括与算法结构和变量声明有关的内容。OMG IDL 是大小写敏感的，并且具有与 C++相似的预处理过程。

我们将通过一些具体实例来逐步说明 OMG IDL。

2.3.1　OMG IDL 基本原则

大的银行一般都设有自动提款机 ATM。ATM 通过磁条解读器读取 ATM 帐号，通过功能键选取操作内容，通过数字键盘键入密码或提款数目，通过专用打印机打印操作记录。

采用 OMG IDL，我们可以得到与 ATM 有关的一系列对象的接口定义如下。为了讨论方便，我们做了一定简化处理，并不太强调对象划分的合理性。

例 2.1　简化后的 ATM 接口定义。

```
/*这个用 OMG IDL 语言编写的接口文件叫做 ATM.IDL*/
module ATM{
        typedef string Accounts;
        typedef unsigned long Password;
```

```
        typedef long Money;

        interface InputMedia{
            typedef unsigned short OperatorCmd;

            void operation_input(in OperatorCmd key);
            void password_input(in Password number);
            void money_amount_input(in Money amount);
        };

        interface OutputMedia{
            boolean output_text(in string string_to_print);
        };

        interface ATMTerminal{
            void exit();
            boolean withdraw(
                        in Accounts NO,
                        in Password number,
                        in Money amount
                        );
            boolean change_password(
                                in Accounts NO,
                                in Password old_number,
                                in Password new_number
                                );
            void
                in Accounts NO,
                in Password number,
                out Money surplus
                );
            void print_operation_log();
            boolean check_card();
        };
        ……
};//结束 ATM 模块定义
```

从例 2.1 中，我们可以得到采用 OMG IDL 编写 CORBA 接口的一些基本原则：

- 注释方法与 C++完全相同。可以在/* */中嵌入注释内容，或者用//表示从下一个字符起到本行末都是注释内容。如例子中首末两行所示。
- 可以把相关的一组对象定义在同一个模块中，这是一个非常有效的编程方式。这样做至少有两个好处：相关对象易于维护，因为它们几乎集中在同一个模块中；参数声明不会冲突，比如，在传呼机模块中，也可以声明一个 Password 参数用来提取个人语音信箱内容，但它不会和 ATM 模块中的同名参数混淆。模块定义方式为 module{模块内容}；。
- 必须为每一个参数指明类型。这样，当参数在异质的网络中传递时，ORB 可以将每一个变量转换为所在平台认可的数据格式。OMG IDL 中的基本数据类型包括 long、short、unsigned long、unsigned short、float、double、char、boolean、octet、any；构造数据类型包括 struct、union、enum、sequence、string。这些数据类型与我们经常使用的数据类

型非常接近，这里不再赘述。其中，any 可以用来和任何一种数据类型匹配，包括构造数据类型以及数组；而且，any 可以在运行中被动态转换为任何一种数据类型。这非常像 C++中的万能数据类型——Variant。

- 任何声明都有一定的作用域。Accounts、Password、Money 的作用域是整个 ATM 模块；OperatorCmd 的作用域仅仅为 InputMedia{}。OMG IDL 作用域的规则与 C++的作用域规则相同。
- 关键字 interface 用来定义一个具体对象及其接口。对象接口中包括一组操作及参数声明。每个 interface 同时也开启一个新的作用域，被它所对应的{}括起来的声明仅仅在对应接口中有效。在 ATM 模块中，有 InputMedia、OutputMedia、ATMTerminal 三个接口对象。
- 操作的声明类似 C++，包括返回类型、函数名称、参数列表三个部分。如果无返回类型，必须指明为 void。
- 必须为操作中的每一个参数指明方向属性。方向属性取值只能为 in、out 或 inout。如果取值为 in，由客户机在运行时声明参数的类型并赋值，服务器只能使用但不能修改其值；如果取值为 out，由客户机在运行时声明参数的类型，服务器在运行中为其赋值；如果取值为 inout，由客户机在运行时声明参数的类型并赋值，服务器可以使用或修改其值。
- 类型声明符号 typedef 的用法与 C++中完全一样。

2.3.2 用 OMG IDL 定义属性及只读属性

CORBA 对象在运行中都有一定的状态，这些状态实际上是 CORBA“对象实现”中数据变量的具体取值。比如，商品对象在运行中具有一定的名称、一定的单价、一定的存量。

在商务运作中，我们经常需要改变对象的状态。比如，每进行一次交易，就需要读取、修改对应商品的库存量。

根据 CORBA 标准，应该为读取、修改 CORBA 对象状态变量声明相应的操作。按照上节的方法，我们可以得到商品对象的接口如下：

例 2.2　用比较原始的方法声明的商品对象接口。

```
//这是一个比较原始的接口声明，仅仅是为了说明问题而编写
module Commodity{
        ……
        interface CommodityItem{
            string _get_name();
            void _set_name(in string name);

            long _get_amount();
            void _set_amount(in long amount);

            double _get_price();
            void _set_price(in double price);
        };
        ……
};
```

这个接口非常繁琐，6 个操作定义仅仅能够读取对象的三个状态变量。而且，6 个定义可以分为 3 组，各组之间有着极大的相似性。

考虑到读取、修改对象状态变量是一个相当普遍的操作，这些操作又极具共性，OMG IDL 引入了属性。属性用关键字 attribute 定义。采用这种方式，我们可以得到商品对象的新的接口定义如下：

例2.3　采用属性定义的商品对象接口。

```
//这是采用属性定义的商品对象接口
module Commodity{
        ……
        interface CommodityItem{
            attribute string name;
            attribute long amount;
            attribute double price;
        };
        ……
};
```

对比以上两个接口定义，我们不难理解属性的来龙去脉。

现在，假设为了便于商务运作，我们希望对每一种商品进行编号。不过，在程序运行过程中，这个编号只能被读取，不能被修改。对于这种情况，我们可以采用只读属性。只读属性用关键字 readonly attribute 定义。

例2.4　具有只读属性的商品对象接口。

```
//本接口中包含只读属性——商品编号
module Commodity{
        ……
        interface CommodityItem{
            attribute string name;
            attribute long amount;
            attribute double price;
            readonly attribute string CommodityNo;
        };
        ……
};
```

在 OMG IDL 中，既可以把基本数据类型声明为属性，也可以把构造数据类型声明为属性。这完全取决于用户需要。

2.3.3　用 OMG IDL 定义构造数据类型

一般情况下，基本数据类型可以用来说明一些简单的参数。比如，long 用来定义 amount、double 用来定义 price。但是，这种方式一次只能声明一个参数。

在现实生活中，许多参数是联袂出现的。比如空间点坐标，通常以（X，Y，Z）的方式表达。为了解决这种问题，OMG IDL 允许使用构造数据类型，主要包括枚举类型、struct、union、sequence 和 string。string 的用法与基本数据类型一致，本节主要说明枚举类型、结构、序列及联合四种构造数据类型。

例2.5　用 OMG IDL 定义枚举类型。

```
//这个例子主要用来说明如何定义枚举类型
module ATM{
        ……
        enum DAY{MON,TUE,WED,THU,FRI,SAT,SUN};

        interface Date{
```

```
        typedef DAY Today;
        boolean is_today_holiday(in Today day);
    };
    ……
};
```

在本例中，我们首先用枚举类型来表达从星期一到星期日这样一个数据类型，它被定义为 DAY，用关键字 enum 说明。

接着，我们声明了一个日期对象接口 Date，它具有一个判断今天是否放假的操作 is_today_holiday，该操作的参数就是一个 DAY 数据类型的变量。也许，你会奇怪：为什么 ATM 也要放假？实际上，我们经常发现有些 ATM 友好地向用户显示“对不起，本提款机暂停服务”。看来，要么是这台机子老出故障，要么是 ATM 有休息日。

例2.6　用 OMG IDL 定义 struct。

```
//这个例子主要用来说明如何定义结构——struct
module ATM{
    ……
    struct ATMLocation {
        string city;
        string blocks;
        unsigned short number;
    };
    ……
};
```

在本例中，我们声明了一个结构 ATMLocation 来表示 ATM 终端的位置。它能够表示终端所在城市、街区和编号。在 ATM 模块中，ATMLocation 成为合法数据类型。

例2.7　用 OMG IDL 定义一维序列。

```
//这个例子主要用来说明如何定义一维序列
module Commodity{
    ……
    struct PurchasingItem{
        string name;
        unsigned short amount;
    };

    interface Purchasing{
        typedef sequence<PurchasingItem> goods_cargo;
        long sum_all_item(in googs_cargo client_buying);
    };
    ……
};
```

在本例中，我们首先定义了一个结构 PurchasingItem，用来记录被购商品的名称和数量。接着，定义了一个一维序列 goods_cargo，用来模拟购物车，装载全部采购物品。在本接口中，查询顾客购物总数的操作 sum_all_item 的输入参数 client_buying 就对应着这种一维序列类型。

goods_cargo 是一个无界一维序列，理论上可以承载无数多个 PurchasingItem。如果希望限定一维序列的上限，可以采用 sequence<数据类型, 上限>的方式来定义。比如，sequence<short, 100>就定义了一个上限不超过 100 的短整型一维序列。

在 OMG IDL 中，一维序列可以嵌套定义。因此，实际上我们也能够定义多维序列。

比如，sequence<sequence<double>>，就定义了一个能够产生无限多个无界双精度一维序列的一维序列。

例2.8　用 OMG IDL 定义联合体。

```
//这个例子主要用来说明如何定义联合体
module Commodity{
        ……
        union PreferentialType switch(short){
            case 1 : float discount_price;
            case 2 : string binding_goods_name;
            case 3 : unsigned short trial_period_time;
            ……
            default : float discount_ratio;
        };
        ……
};
```

在本例中，我们定义了商品的优惠类型 PreferentialType。由于一般情况下只能采取一种优惠策略，我们将 PreferentialType 定义为联合体：要么提供优惠价格；要么提供捆绑销售的物品；要么规定一段免费试用期……默认情况下则根据打折率折算物价。

现在，我们可以为每种商品说明一个 PreferentialType 类型的属性，选择不同的优惠策略来吸引顾客。

2.3.4　用 OMG IDL 定义继承、多重继承及跨模块继承

继承有两种形式：

- 单继承。一个接口从另外一个接口派生出来。
- 多继承。一个接口从两个或者两个以上接口派生出来。

一般说来，被继承的接口应该先于派生接口被定义说明。不过，同 C++一样，OMG IDL 允许使用前置说明，对被继承接口声明但暂时不定义。

例2.9　用 OMG IDL 定义单继承。

```
//这个例子用来说明如何定义单继承
module Commodity{
        ……
        enum SEX{male, female};
        interface Person;//前置说明

        struct PersonData{
            string name;
            SEX sex_type;
            Unsigned short age;
            string address;
        };

        interface Client : Person{
            void give_client_a_number(in string client_number);
        };

        interface Person{
            readonly attribute PersonData data;
```

```
            void change  address(in string new  address);
        };
        ……
};
```

在本例中，我们首先定义了一个结构 PersonData，用来记录每个人的姓名、年龄、性别和住址。

接着，我们采用前置说明的方式，把 Person 声明为一个接口对象，但并没有给出详细的接口定义。从后面的定义中，不难发现接口 Person 实际上声明了两个操作，一个读取 PersonData 的操作和一个改变地址的操作。

接口 Client 从接口 Person 继承而来，并添加了一个分配客户编号的操作。根据 OMG IDL 的定义，接口 Client 和 Person 的关系及操作声明如图 2.5 所示。

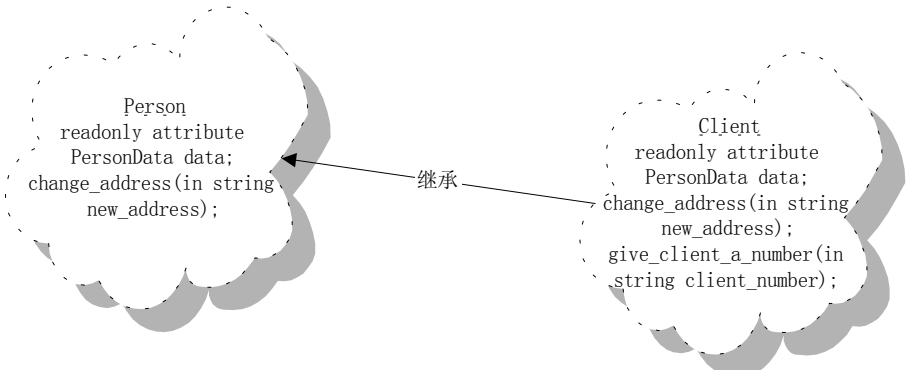

图 2.5　单继承关系图及各接口拥有的操作

以上例子都具有一定的应用背景，是针对具体问题进行的。通过这些例子，我们希望读者能够体会到如何采用 OMG IDL 定义实际问题。

在以后的例子中，我们不再刻意强调它们是否具有现实意义（因为这样太废篇幅），而是更加关心其中的语法现象。

例2.10　用 OMG IDL 定义多重继承。

```
//这个例子用来说明如何定义多重继承
module example{
        interface example1{
            attribute long property1;
            void operation1();
        };

        interface example2{
            attribute short property2;
            void operation2();
        };

        interface example3 : example1, example2{
            attribute string property3;
            void operation3();
        };
};
```

在本例中，接口 example3 从接口 example2、example1 多重继承而来。它们之间的关

系和各自拥有的操作如图 2.6 所示。

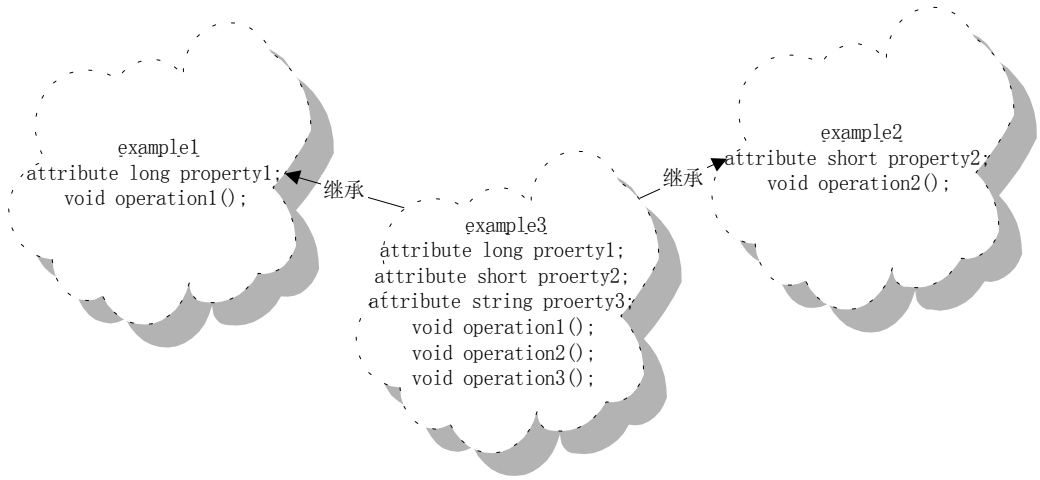

图 2.6　多继承关系图及各接口拥有的操作

显然，我们不能将所有对象接口都放在同一个模块中，因此，有时我们也需要从别的模块中进行继承。在 OMG IDL 中，只需用域操作符**::**说明即可。

例2.11　用 OMG IDL 进行跨模块继承。

```
//这个例子用来说明如何定义跨模块继承
module Module1{
        interface example1{
            attribute long property1;
            void operation1();
        };
};

module Module2{
        interface example1{
            attribute long property2;
            void operation2();
        };

        interface example2 : Module1::example1{
        };
        ……
};
```

在本例中，接口 example2 继承了模块 Module1 中的接口 example1，但与模块 Module2 中的接口 example1 没有继承关系。因此，接口 example2 具有属性 property1 和操作 operation1。

实际上，通过域标志符**::**，我们也可以把操作中的参数说明为其它模块中定义的数据类型。

2.3.5　用 OMG IDL 定义异常

操作在执行过程中往往会出现异常。而用 OMG IDL 定义异常是为了保证当异常发生时，我们的程序能够继续运行。

如果你有使用过盗版光碟的经验，也许会碰到过这样的情景——当我们打开 Setup Wizard 选择好需要安装的软件模块、规定好各自的路径后开始进行安装。由于文件拷贝需要花费二十多分钟，就决定去吃一顿饭。可是，当我们回来后却发现，由于光碟的质量问题，一个号称“找不到某某文件”的对话框出现在屏幕的正中心，而安装进度依然停留在“0%”。

这个体验说明，在分布式软件开发中“异常处理”十分重要：当我们从北京调用一个在上海某服务器上的服务时，绝对不允许与该服务有关的对象实现无可奈何地在上海服务器的屏幕上显示一个“出现异常，下来怎么办”的对话框。因此，电子商务软件中必须为操作定义异常处理。

异常处理包括三个主要环节：

- 定义异常处理。声明可能发生的异常的名称，说明为处理异常提供的数据（如原因）并把异常同具体操作联系起来。
- 引发异常。在客户及对象实现的有关代码中引发异常。
- 捕捉异常、异常处理。在客户及对象实现的有关代码中捕捉异常，并自动处理异常。

对于 CORBA 而言，第一个环节应该在 OMG IDL 中进行；第二、三个环节应该在客户或者对象实现中进行。本节主要说明第一个环节。

由于操作在执行过程中有可能碰到很多类似的异常，CORBA 中定义了一套标准异常。标准异常覆盖范围较广，包括了 ORB 内部错误、存储错误及资源使用错误。标准异常由 CORBA 定义、提供，用户不必在 OMG IDL 中做任何声明、定义，就可以在实现中直接引发、捕捉和处理。当然，标准异常也可以由 ORB 自动引发。

一个标准异常包含三个信息：

- 名称。异常的名称，比如 BAD_OPERATION 表示操作无效。
- 辅助码（minor）。辅助码由 ORB 厂商提供，其取值和对应含义由厂商自行规定。通常，可以把处理异常需要的一些参数或者异常更详细的情况用辅助码表示。
- 完成状态码（completed）。完成状态码用来表示异常引发时，对应操作的执行情况，包括三种取值：CORBA_COMPLETED_YES（表示对象实现在异常引发前已完成处理）、CORBA_COMPLETED_NO（表示对象实现在异常引发前没有完成处理）、CORBA_COMPLETED_MAYBE（表示对象实现的状态在引发时未知）。

CORBA 中的标准异常如表 2.1 所示。

表 2.1　CORBA 中的标准异常

异常名称	异常说明
BAD_CONTEXT	上下文对象出错
BAD_INV_ORDER	例程激发出错
BAD_OPERATION	操作无效
BAD_PARAM	参数无效
BAD_TYPECODE	类型码无效
COMM_FAILURE	通信失败
DATA_CONVERSION	数据转换失败
FREE_MEM	内存无法释放
IMP_LIMIT	违反实现限制
INITIALIZE	ORB 初始化出错
INTERNAL	ORB 内部错误
INTF_REPOS	访问接口仓库出错
INV_FLAG	指定标记位出错
INV_IDENT	指定标识符号无效
INV_OBJREF	对象引用无效
MARSHAL	编组参数或结果发生错误
NO_IMPLEMENT	没有对象实现或实现无法使用
NO_MEMORY	没有足够的动态内存
NO_PERMISSION	没有权限
NO_RESOURCES	没有足够的资源
NO_RESPONSE	没有响应
OBJ_ADAPTER	发生了与对象适配器有关的错误
OBJ_NOT_EXIST	没有此对象引用，自动删除
PERSIST_STORE	发生了持续流或持续对象存储错误
TRANSIENT	发生了瞬时错误；重新请求
UNKNOWN	发生了未知的错误

这些标准异常的定义如下所示：

例2.12　OMG 的标准异常定义：

```
//这是 OMG 定义的标准异常
#define ex_body {unsigned long minor; completion_status completed;}

enum completion_status {COMPLETED_YES,
                        COMPLETED_NO,
                        COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION,
                     USER_EXCEPTION,
                     SYSTEM_EXCEPTION};

exception BAD_CONTEXT ex_body; // error processing context object
exception BAD_OPERATION ex_body; // invalid operation
exception BAD_PARAM ex_body; // an invalid parameter was passed
exception BAD_TYPECODE ex_body; // bad typecode
exception COMM_FAILURE ex_body; // communication failure
exception DATA_CONVERSION ex_body; // data conversion error
exception FREE_MEM ex_body; // cannot free memory
exception IMP_LIMIT ex_body; // violated implementation limit
exception INITIALIZE ex_body; // ORB initialization failure
exception INTERNAL ex_body; // ORB internal error
```

```
exception INTF_REPOS ex_body; // error accessing interface repository
exception INV_FLAG ex_body; // invalid flag was specified
exception INV_IDENT ex_body; // invalid identifier syntax
exception INV_OBJREF ex_body; // invalid object reference
exception MARSHAL ex_body; // error marshalling param/result
exception NO_IMPLEMENT ex_body; //no operation implementation
exception NO_MEMORY ex_body; // dynamic memory allocation failure
exception NO_PERMISSION ex_body; // no permission for attempted op.
exception NO_RESOURCES ex_body; // insufficient resources for req.
exception NO_RESPONSE ex_body; // response to req. not yet available
exception OBJ_ADAPTER ex_body; // failure detected by object adapter
exception OBJECT_NOT_EXIST ex_body; // non-existent object, delete
// reference
exception PERSIST_STORE ex_body; // persistent storage failure
exception BAD_INV_ORDER ex_body; // routine invocations out of order
exception TRANSIENT ex_body; // transient failure - reissue request
exception UNKNOWN ex_body; // the unknown exception
```

除去标准异常以外，CORBA 还允许用户自定义异常。自定义异常有两个主要限制：

- 只能把自定义异常同操作联系起来，不能把自定义异常与属性联系起来。
- 只能由“对象实现”引发自定义异常，不能由 ORB 引发自定义异常。

例2.13　用 OMG IDL 说明自定义异常。

```
//这个例子用来说明如何声明自定义异常
module Module1{
        exception myself_exception{
            string reason;
        };

        interface example{
            void operation() raises(myself_exception);
        };

};
```

在本例中，我们首先定义了一个叫做 myself_exception 的异常，该异常有一个 string 类型的成员 reason，用来供用户记录引发异常的原因。说明自定义异常的关键字为 exception。

接着，我们将操作 operation 与该异常联系起来，表明在 operation 执行过程中有可能产生该异常。联系自定义异常与操作的关键字为 raises。如果一个操作可能引发多种自定义异常，我们可以将所有异常都放在 raises 之后的括号内，并以逗号分隔。

2.3.6　用 OMG IDL 定义上下文对象

在 UNIX 和 PC-DOS 中，有一种帮助用户、程序决定执行优先权、操作偏好的机制：程序模块可以读写一些不是必要参数的信息条目，在考虑这些信息后，既可以改变也可以不改变具体操作、执行优先权。

CORBA 中为这种机制提供了兼容，并称之为上下文对象——context。上下文对象由一系列“名-值对”（name-value pairs）构成。在 OMG IDL 中，规定上下文对象取值必须是 string 类型，并把这种“名-值对”称为特征（property）。

采用上下文对象传递非参数信息时，该信息仅仅在一段时间内有效。而且，客户必须通过 ORB 提供的操作处理上下文对象。服务器则可以根据要求读取这些上下文信息，决

定自己的行为方式。

例2.14　用 OMG IDL 声明上下文对象。

```
//这个例子用来说明如何定义上下文对象
module Module{
        interface example{
            void operation1();
            void operation2() context("info_one","info_two");
        };
};
```

在本例中，我们为接口 example 中的操作 operation2 定义了上下文对象。该上下文对象包括两个特征信息对，分别叫做 info_one、info_two。当客户激发该操作时，可以采用 ORB 提供的函数管理上下文对象并通知服务器考虑上下文信息决定服务方式。

如果对应操作中定义了异常，应该把定义上下文对象的关键字 context 放在 raises 之后。

2.3.7　用 OMG IDL 定义单向请求

在分布式软件开发中，请求/响应的实际执行过程通常采用同步通信方式来实现。但是，CORBA 中确实提供了支持异步通信的机制，单向（oneway）请求就是其中的一种（另外一种机制是延迟同步）。

例2.15　用 OMG IDL 定义单向请求。

```
//这个例子主要用来说明如何定义单向请求
module Module1{
        interface example{
            oneway void SendMessage(in string message);
        };

};
```

在本例中，SendMessage 被定义为单向操作，客户激发该操作后立刻返回，转而执行其它操作、请求。定义单向请求的关键字为 oneway。

被定义为单向请求的操作返回类型必须为 void，同时仅仅允许 in 型参数，且不能同自定义异常建立联系（标准异常可以由自动 ORB 引发）。

2.3.8　CORBA 中的伪对象

在 CORBA 中，有许多伪对象。

其中一个被称做 Object，它的接口中定义了一些几乎在所有对象中都有效的操作。我们可以认为，任何一个 CORBA 接口都自动从 Object 继承并具备这些操作功能。但是，在编写对象实现时却不需要添加任何代码，因为这些操作实际上都由 ORB 执行。也就是说，我们只需考虑使用这些操作，不需考虑它们的实现问题。

之所以称其为伪对象是指它的接口确实是使用 OMG IDL 定义的，但是却与一般对象有显著不同，包括：

- 除系统定义以外，用户不能把伪对象作为操作的参数使用
- 不能使用动态激发接口 DII 调用其操作
- 在接口仓库中没有对应定义。

定义伪对象的 OMG IDL 语言通常又被称做“伪 IDL”——PIDL。

下面是用 PIDL 定义的伪对象 Object 的接口：

```
//这是用 PIDL 定义的伪对象 Object 的接口
module CORBA {
        ……
        interface Object { // PIDL
            ImplementationDef get_implementation ();
            InterfaceDef get_interface ();
            boolean is_nil();
            Object duplicate ();
            void release ();
            boolean is_a (in string logical_type_id);
            boolean non_existent();
            boolean is_equivalent (in Object other_object);
            unsigned long hash(in unsigned long maximum);
            Status create_request (
                                in Context ctx,
                                in Identifieroperation,
                                in NVList arg_list,
                                inout NamedValueresult,
                                out Requestrequest,
                                in Flags req_flags
                                );
        };
        ……
};
```

这些操作的功能我们将在后面有关章节详细说明，不过应该了解，CORBA 中包括许多采用伪接口语言 PIDL 定义的，“放之 CORBA 而皆准”的伪对象。

2.4 OMG IDL 与 Microsoft IDL

COM/DCOM 是 Microsoft 对于分布式软件开发采取的方案，是 CORBA 的主要竞争对手。在前面章节中，我们曾经指出，COM/DCOM 是基于 Component 的。为了实现跨语言编写 Component，Microsoft 也采用了接口技术，并规定了自家的接口定义语言 Microsoft IDL。

例2.16　用 Micosoft IDL 定义的 COM/DCOM 中的两个基本对象接口。

```
#ifndef DO_NO_IMPORTS
import "wtypes.idl";
#endif

[
   local,
   object,
   uuid(00000000-0000-0000-C000-000000000046),
   pointer_default(unique)
]

interface IUnknown
{
     typedef [unique] IUnknown *LPUNKNOWN;
```

```
    HRESULT QueryInterface(
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
}

[
    object,
    uuid(00000001-0000-0000-C000-000000000046),
    pointer_default(unique)
]

interface IClassFactory : IUnknown
{
    typedef [unique] IClassFactory * LPCLASSFACTORY;

    [local]
    HRESULT CreateInstance(
        [in, unique] IUnknown * pUnkOuter,
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject);

    [call_as(CreateInstance)]
    HRESULT RemoteCreateInstance(
        [in] REFIID riid,
        [out, iid_is(riid)] IUnknown ** ppvObject);

    [local]
    HRESULT LockServer(
    [in] BOOL fLock);

    [call_as(LockServer)]
    HRESULT __stdcall RemoteLockServer(
    [in] BOOL fLock);
}
```

这个例子保持了 Microsoft 产品 Visual C++一贯的书写风格。其中定义了 COM/DCOM 中两个著名的对象 IUnknown 和 IClassFactory 的接口。IUnknown 中有三个操作：QueryInterface、AddRef、Release。IClassFactory 从 IUnknown 继承而来，新添加了 CreateInstance、RemoteCreateInstance、LockServer、RemoteLockServer 四个操作。同时，接口中包含两个对象的 uuid——通用统一标志（universally unique id）。

对比 OMG IDL 与 Microsoft IDL，它们之间有以下主要共同点：

- 都可以定义对象的操作接口
- 都可以定义对象的继承关系（COM/DCOM 仅支持单重继承）
- 都可以声明操作需要的参数及参数的方向
- 都无法声明变量

- 都不能用来编写对象具体的实现代码
- 都把面向对象技术提升到了高于“类”的层次
- 都可以采用多种编程语言混合实现

当然，OMG IDL 与 Microsoft IDL 之间的差异更加显著，主要包括：

- 具体语法不同。这一点非常明显，包括数据类型、关键字格式等不同。
- 采用策略不同。OMG IDL 仅仅定义了接口，Microsoft IDL 不但定义了接口，还在系统注册表中登录了对象。
- 使用期限不同。OMG IDL 一经颁布，就不会改变（当然可以修改）；而 Microsoft 本身承认，有可能有一天将会彻底改变 Microsoft IDL 的面貌。

在 CORBA2.0 版本中，就专门定义了 OMG IDL 与 Microsoft IDL 的映射，关于这些内容本书不再赘述。

2.5 本章小结

对象请求代理 ORB 对整个 CORBA 体系结构做了有序地分割，使得基于 CORBA 的分布式软件开发建立在伪客户/服务器方式之上。为了保证软件的即插即用，需要使被分割的各个 CORBA 子部分遵循相同的定义。这个定义采用 CORBA 接口定义语言 OMG IDL 说明。

为了能够在客户、对象实现、ORB 之间也保持互换兼容性，CORBA 引入了“接口存根”及“接口框架”两个对象，它们与 ORB 的关系似乎更加紧密，经常被归入 ORB 一同考虑。它们可以用与 ORB 配套使用的 IDL 编译器自动编译获得。

通过一些具体实例，我们不难理解 OMG IDL 的基本原则以及如何定义属性、结构参数、枚举类型参数、序列参数、联合参数、继承关系、异常、上下文对象和单向请求。而 PIDL 主要用来定义 CORBA 伪对象。

说起 CORBA 与 COM/DCOM 的分歧，恐怕在 OMG IDL 与 Microsoft IDL 中就已经开始显露了。

第 3 章　OMG IDL 在 C 及 C++中的映射

本章内容提要：

- OMG IDL *在 C 中的映射扼要*
- *对象引用*（Object reference）
- *伪对象* Environment
- *异常处理例程*
- *伪对象* Context *及其处理例程*
- OMG IDL *在* C++*中的映射扼要*

3.1　讨论 OMG IDL 在 C 及 C++中的映射目的

虽然 OMG IDL 为 CORBA 提供了跨越语言障碍、统一描述分布式对象之间各种操作的可能，却不能够真正实现这些对象操作。因为对象实现中的任何代码都是通过支持 CORBA 编程的具体开发语言编写的。

任何一种编程语言，只要能够辨别和理解 OMG IDL 的语法，或者说，OMG IDL 在这种语言中建立了映射，就可以用来进行 CORBA 软件开发。在上一章中，我们已经说明，CORBA 编程语言中的语法、功能可以比 OMG IDL 中提供的更多。

目前，有许多语言都支持 CORBA 软件开发，或者说，OMG IDL 与它们已经建立了映射。其中包括 Java、C++、C、Smalltalk、Ada、COBOL 等等。每种编程语言都有它自身的优缺点。这一点我们不再多加评论：因为 CORBA 的目的之一就是让每个程序员自由使用得心应手的开发工具。

研究 OMG IDL 在具体语言中的映射，实际上就是研究如何用具体语言编写、调度 CORBA 分布式对象。根据前言中提出的“万变不离其宗”的思路，我们需要选择合适的语言映射来理解 CORBA 编程及分布式软件开发的实 质。

Java、C++、C 是大家比较偏好的编程语言。但是，Java 比 C++、C 语言的抽象级别更高，用户编程时比较方便，对于理解分布式软件开发的核心思想却并没有帮助（因为被隐藏起来了）。这就如同采用 C++Builder 编程时，简便可行；但对于理解 Windows 编程的核心机制，就没有 Visual C++或 Borland C++诠释的透彻。

我们重点说明 OMG IDL 在 C 及 C++中的映射，就是为了能够对 CORBA 以及分布式软件开发“融会贯通”，了解它们的一些底层问题和内幕。

我们相信，了解 OMG IDL 在 C 及 C++中的映射后，不论采用 Java、C++Builder、Delphi 还是其它工具进行 CORBA 软件开发，一定都会游刃有余。

3.2　OMG IDL 在 C 中的映射

C 语言不是面向对象类型的语言，但是它支持指针、结构、动态内存申请，因此非常容易和 OMG IDL 建立映射。

任何语言与 OMG IDL 建立映射时，都应该解决几个基本问题：

- 建立两种语言的基本数据类型、构造数据类型、常量以及对象之间的映射
- 建立两种语言操作调用、参数传递之间的映射
- 建立两种语言读取、修改属性值之间的映射

- 建立两种语言引发异常、处理异常之间的映射
- 建立两种语言内存申请、释放、管理之间的映射。

3.2.1 OMG IDL 数据类型在 C 中映射

OMG IDL 中的数据类型定义非常严格，考虑到不同 C 编译器在解释相同名称的数据类型时有可能有所区别，映射中没有把某种 OMG IDL 数据类型直接映射为 C 语言中的某种数据类型，而是贯以前缀，表示为一种规范。

OMG IDL 数据类型在 C 语言中的映射如表 3.1 所示：

表 3.1 OMG IDL 数据类型在 C 语言中的映射

OMG IDL 数据类型	C 语言中映射的数据类型
short	CORBA_short
long	CORBA_long
unsigned short	CORBA_unsigned_short
unsigned long	CORBA_unsigned_long
float	CORBA_float
double	CORBA_double
char	CORBA_char
boolean	CORBA_boolean
octet	CORBA_octet
enum	CORBA_enum
any	typedef struct CORBA_any { CORBA_TypeCode _type; void *_value;} CORBA_any;

通过这种映射方式，CORBA_long 总对应于一个 32-bit 的整数类型，CORBA_float 总对应于一个 IEEE32-bit 浮点数类型。无论采用哪种 C 编译器，都应该遵循这些规范。

3.2.2 接口及操作在 C 中的映射

现在，假设有这样一个非常简单的接口：

例3.1 拥有一个带参数的操作的简单接口。

```
//这是在 OMG IDL 中定义的接口
interface example1{
    long operation1(in long parameter);
};
```

这个接口在 C 语言中对应于如下声明：

例3.2 前一个例子在 C 语言中的映射。

```
//这是上一个例子在 C 语言中的映射
typedef CORBA_Object example1;

extern CORBA_long example1_operation1(
                                    example1 o,
                                    CORBA_long parameter,
                                    CORBA_Enviroment *ev
                                        );
```

从这个简单接口的 C 语言映射中，我们可以得到以下规律：

- OMG IDL 接口在 C 语言中被映射为一系列函数声明、参数声明。

- 映射后，每个 OMG IDL 接口拥有一个与接口名称相同的对象引用（Object reference）。本例中，这个对象引用是 example1，被定义为 CORBA_Object 类型。CORBA_Object 是伪对象 Object 在 C 语言中的映射，表示被定义的参数为对象引用。用户在编码时，可以把对象引用作为服务的请求目标、请求参数、请求返回值传入传出。
- 映射后，操作的名称由模块、接口及操作名称三部分用“_”连接而成，具体形式为：模块名称_接口名称_操作名称。比如，在本例中，映射后的操作名称为 example1_operation1。为了讨论、书写方便，我们在举例中暂时不考虑模块名称。
- 操作中的各个参数在映射前后不改变名称。
- 操作中的方向属性没有被直接映射，需要程序员在编码时酌情处理。比如，对于 in 方向属性，如果参数是基本数据类型、枚举数据类型或者对象引用，直接传递参数值；如果参数是数组，传递对应数组变量第一个元素的地址；如果参数是 struct 构造数据类型，传递参数对应变量的地址；如果参数是 string，传递为 char*。详细情况如表 3.2 所示，其中的数组切片稍后再解释。
- 映射后的操作中增加了两个参数：对象引用参数 o 被添加在原来所有参数之前；环境参数指针 ev 被添加在所有参数之后。环境参数 CORBA_Environment 是一个 struct 数据类型（对应于环境伪对象 Environment），可以用来进行异常处理，它的详细情况在异常处理映射部分讨论。
- 映射后的操作均被声明为 extern 类型，表示实际代码位于外模块中。

表 3.2　不同方向属性的数据类型在 C 语言中的映射

数据类型	in	inout	out	return
short	short	short*	short*	short
long	long	long*	long*	long
unsigned short	unsigned_short	unsigned_short*	unsigned_short*	unsigned_short
unsigned long	unsigned_long	unsigned_long*	unsigned_long*	unsigned_long
float	float	float*	float*	float
double	double	double*	double*	double
boolean	boolean	boolean*	boolean*	boolean
char	char	char*	char*	char
octet	octet	octet*	octet*	octet
enum	enum	enum*	enum*	enum
对象引用 ptr	objref_ptr	objref_ptr*	objref_ptr*	objref_ptr
定长度 struct	struct*	struct*	struct*	struct
变长度 struct	struct*	struct*	struct**	struct*
定长度 union	union*	union*	union*	union
变长度 union	union*	union*	union**	union*
string	char*	char**	char**	char*
sequence	sequence*	sequence*	sequence**	sequence*
定长度 array	array	array	array	array slice*
变长度 array	array	array	array slice**	array slice*
any	any*	any*	any**	any*

在将 OMG IDL 定义的对象接口映射为 C 语言时，引入了两个重要数据类型：CORBA_Object 及 CORBA_Environment。

这两个数据类型实际上是 CORBA 伪对象 Object 及 Environment 在 C 语言中的映射，它

们很好地揭示出 CORBA 的实现内幕：前者实际上是被请求的每个对象的“句柄”——Handle；后者为异常自动处理提供了可能。

☎ **技巧** CORBA 接口 C 语言映射中的操作名称是由模块名称、接口名称、操作名称与“_”直接连接而成的。因此，我们在用 OMG IDL 定义接口操作时，如果碰巧声明了一个由接口名称、“_”及已经存在的操作名称直接连接构成的新操作，用 IDL 编译器映射为 C 语言时就很容易导致混淆。为了避免这种倾向，我们应该谨慎使用下划线“_”，或者全面禁止使用下划线“_”。本书中为了方便读者理解函数定义，没有限制使用下划线“_”。但是，如果在实际开发软件时，最好避免。当然，这种情况在面向对象类型的语言，如 C++、Smalltalk 中就不会发生。

3.2.3 对象引用——Object reference

在 C 语言中，用 CORBA_Object 类型定义的参数就是对象引用。它可以用来引用有关对象实现的具体实例（或者说服务器对象）。

例3.3 假设有能够返回另一个接口的 CORBA 操作，如下所示。

```
//这是一个返回另一接口的操作
#include “example1.idl”

interface example2{
    example1 operation2();
};
```

在本例中，接口 example2 含有一个不带参数的操作 operation2。该操作返回类型为 example1 所定义的接口。而接口 example1 在另外一个 IDL 文件中定义。

在这个例子中，我们还使用了关键字#include。实际上，OMG IDL 有很多类似的预处理（preprocessor）符号，使用方法与 C++的同类符号相似，这里不再赘述。

用 IDL 编译器编译这个接口定义，可以获得接口在 C 语言中的映射如下：

例3.4 前一个接口在 C 语言中的映射。

```
//这是前一个 OMG IDL 接口在 C 语言中的映射
#include “example1.h”

typedef CORBA_Object example2;

extern example1 example2_operation2(
                                    example2 o;
                                    CORBA_Environment * ev
                                    );
```

作为客户端程序员，我们可以在程序中调用对象实现中的这个函数，如下所示：

例3.5 在客户端调用接口对象实现中的函数。

```
//这个例子用来说明如何使用对象引用
#include “example2.h”

example1 ObjectRef1;
example2 ObjectRef2;
example2 ObjectRef3;

CORBA_Environment ev;
```

```
        ……
ObjectRef2 = function_to_get_example2_from_ORB();
ObjectRef3 = function_to_get_example2_from_Program();

ObjectRef1 = example2_operation2(ObjectRef2,&ev);
        ……
```

在这个例子中，我们可以体会到对象引用的以下作用：

- 对象引用可以用来表示对象实现的具体实例，也就是服务器对象。服务器对象可以由 ORB 直接提供，也可以由用户自己在程序中生成。
- 对象引用可以作为参数传递。
- 对象引用可以作为函数返回值。

3.2.4 继承在 C 语言中的映射

由于 C 语言本身不支持继承，继承在 C 语言中的映射显得比较繁琐，需要为派生的接口添加许多操作声明。

例3.6　带继承关系的简单接口

```
//这是一个带有继承关系的简单接□
interface example1{
    void operation1();
};

interface example2 : example1{
    void operation2();
};
```

接口 example1、 example2 的 C 语言映射如下：

例3.7　继承接口的 C 语言映射。

```
//这是带有继承关系的简单接□在 C 语言中的映射
Typedef CORBA_Object example1;

extern void example1_operation1(
                                    example1 o,
                                    CORBA_Environment * ev
                                    );

Typedef CORBA_Object example2;

extern void example2_operation1(
                                    example2 o,
                                    CORBA_Environment * ev
                                    );
extern void example2_operation2(
                                    example2 o,
                                    CORBA_Environment * ev
                                    );
```

由于继承关系的存在，example2 中的操作声明增加了。

在实际应用中，用户通过 example1_operation1 激发的是接口 example1 中的操作

operation1；通过 example2_operation1 激发的是接口 example2 中的操作 operation1。而且，在激发 example1_operation1 时，既可以使用 example1 类型的对象引用作参数，也可以使用 example2 类型的对象引用作参数。实际上，它们都是 CORBA_Object 类型。

3.2.5 属性在 C 语言中的映射

在上一章中，我们曾指出，引入属性的目的是为了定义接口的方便性。即使没有属性，有关操作依然可以被定义。

把属性映射为 C 语言时，实际上就是把属性还原为_set_和_get_函数对，再对这些函数对进行映射。

例3.8　带有属性的简单接口。

```
//这是带有属性的简单接口
interface example{
    attribute float property1;
    readonly attribute double property2;
};
```

这个带有属性的接口可以被还原为下列操作：

例3.9　带有属性的接口被还原为普通操作后的接口。

```
//这是带有属性接口被还原为普通操作后的接口
interface example{
    float _get_property1();
    void _set_property1(in float p1);

    double _get_property2();
};
```

最后，该接口在 C 语言中映射如下：

例3.10　带有属性的接口在 C 语言中的映射。

```
//这是带有属性的接口在 C 语言中的映射
typedef CORBA_Object example;

extern CORBA_float example__get_property1(
                                        example o,
                                        CORBA_Environment * ev
                                        );
extern void example__set_property1(
                                    example o,
                                    CORBA_float p1,
                                    CORBA_Environment * ev
                                    );
extern CORBA_double example__get_property2(
                                    example o,
                                    CORBA_Environment *ev
                                    );
```

值得注意的是，在映射后的操作名称中，接口名与 get_xxxxxxxx、set_xxxxxxxx 之间由双下划线“__”连接而成。

3.2.6 异常处理在 C 语言中的映射

CORBA_Environment 是 C 语言中实现异常处理的一个重要机制。

在 C 语言中，CORBA_Environment 实际上是 CORBA 伪对象 Environment 的映射，采用结构方式定义，如下所示：

例3.11 CORBA_Environment 在 C 语言中映射的一部分。

```
//这个例子是 CORBA_Environment 在 C 语言映射中必不可少的一部分
tydedef struct CORBA_Environment{
    CORBA_execption_type _major;
    ……
} CORBA_Environment;
```

CORBA_Environment 中至少含有一个成员_major，用来指示某个操作在执行过程中是否发生了异常；如果发生了异常又属于哪种主要类型。

_major 是一个枚举类型数据，包括三种取值：

- CORBA_NO_EXCEPTION。说明操作在执行过程中没有发生异常，被请求的操作成功完成。
- CORBA_SYSTEM_EXCEPTION。说明操作在执行过程中发生了异常，而且是由 CORBA 本身定义的标准异常。
- CORBA_USER_EXCEPTION。说明操作在执行过程中发生了异常，而且是由用户自己定义的自定义异常。

同时，在 CORBA 的 C 语言映射中，还提供了一系列函数例程，用来读取异常信息、定位异常结构的位置、释放异常结构。这些例程如下所示：

- CORBA_exception_id。本例程返回一个字符串指针，指向被引发的异常所对应的字符串。该字符串将在标准异常、自定义异常的映射中再详细说明。本例程的 C 语言映射为：

```
// CORBA_exception_id 的 C 语言映射
CORBA_char * CORBA_exception_id(CORBA_Environment * ev);
```

- CORBA_exception_value。本例程返回指向与异常信息结构相关的指针。该结构中通常含有处理异常所需要的详细信息。本例程的 C 语言映射为：

```
// CORBA_exception_value 的 C 语言映射
void * CORBA_exception_value(CORBA_Environment * ev);
```

- CORBA_exception_free。本例程释放与异常相关的结构所对应的存储区。本例程的 C 语言映射为：

```
// CORBA_exception_free 的 C 语言映射
void CORBA_exception_free(CORBA_Environment * ev);
```

借助这些例程和结构，在 C 语言中处理异常的一般方式如下所示：

- 首先，通过环境对象 CORBA_Environment 中_major 参数的取值捕捉异常，判断被引发异常属于标准异常还是自定义异常。
- 其次，调用 CORBA_exception_id、CORBA_exception_value 两个函数获得异常的详细信息，进行异常处理。
- 最后，通过 CORBA_exception_free 释放该异常占用的所有资源。

如果希望获取异常的详细信息，需要了解标准异常以及自定义异常在 C 语言中的映射

格式。

CORBA 中的标准异常拥有相似的结构：一个异常名称对应一个由辅助码 minor 和完成状态码 completed 组成的结构，如下所示：

例3.12　CORBA 中用 OMG IDL 定义的一个标准异常。

```
//这个例子说明 CORBA 中标准异常的定义格式
module CORBA{
        enum completion_status{COMPLETED_YES,
                               COMPLETED_NO,
                               COMPLETED_MAYBE
                               };

        exception UNKNOWN{
            unsigned long minor;
            completion_status completed;
        };
        ……
};
```

这个标准异常的名称为 UNKNOWN，与其它标准异常一样，它实际上是一个具有两个成员的结构。

该标准异常在 C 语言中的映射如下：

例3.13　标准异常 UNKNOWN 在 C 语言中的映射。

```
//这个例子说明标准异常在 C 语言中映射的格式
typedef unsigned long CORBA_enum;
typedef unsigned long CORBA_completion_status;

#define CORBA_COMPLETED_YES 0;
#define CORBA_COMPLETED_NO 1;
#define CORBA_COMPLETED_MAYBE 2;

#define ex_CORBA_UNKNOWN “1001233”;

typedef struct CORBA_UNKNOWN{
        CORBA_unsigned_long minor;
        CORBA_completion_status completed;
) CORBA_UNKNOWN;
```

在上面这个映射中，与标准异常 CORBA_UNKNOWN 对应的字符串被定义为“1001233”。实际上，该字符串是表明异常类型的唯一标志号。通常，可以把 ex_CORBA_UNKNOWN 作为该唯一标志号的索引名称使用，以增加程序代码的可读性、互换性。

对照标准异常的上述映射格式，用户应该能够想象捕捉到异常后的处理方式：

- 通过 CORBA_exception_id 获得异常的唯一标志字符串。
- 通过 CORBA_exception_value 获得 CORBA_UNKNOWN 结构的存储地址，读取 minor 中附加的详细信息、根据 completed 取值判断引发异常的操作的执行状态（可参看第二章中 completed 取值说明）并处理异常。
- 通过 CORBA_exception_free 最终释放异常对应的内存。

与标准异常不同，自定义异常在 C 语言中的映射方式取决于用户自身的定义。

例3.14　一个简单的用户自定义异常。

```
//这是一个简单自定义异常接口
exception MY_EXCEPTION1{
        string reason;
};

exception MY_EXCEPTION2{
        string reason;
        string time;
};

interface example{
    void operation1() raises(MY_EXCEPTION1,MY_EXCEPTION2);
};
```

上述自定义异常被映射为以下 C 代码。

例3.15　简单自定义异常在 C 语言中的映射。

```
//这是自定义异常在 C 语言中的格式
#define ex_MY_EXCEPTION1 “100001”;
#define ex_MY_EXCEPTION2 “100002”;

typedef struct MY_EXCEPTION1{
        CORBA_string reason;
} MY_EXCEPTION1;

typedef struct MY_EXCEPTION2{
        CORBA_string reason;
        CORBA_string time;
} MY_EXCEPTION2;
```

采用类似处理标准异常的步骤，用户也可以获得自定义异常 my_exception1 中的 reason 信息以及自定义异常 my_exception2 中的 reason、time 信息，进行异常处理。

3.2.7　上下文对象在 C 语言中的映射

上下文对象在 C 语言中的映射实际上非常简单：如果在 OMG IDL 定义的某个操作声明中包括上下文对象关键字 context，就把一个类型为 CORBA_Context 的参数添加在 CORBA_Object 类型参数之后， 使它成为该操作的第二个参数。

比如，假设我们有如下所示的一个接口：

例3.16　使用上下文对象的接口。

```
//这个例子使用了上下文对象
interface example{
        void operation(long param1) context(“info_one”);
};
```

这个接口在 C 语言中的映射为：

例3.17　使用上下文对象的接口在 C 语言中的映射。

```
//本例中使用了上下文对象
typedef CORBA_Object example;

extern void example_operation(
                        example o,
```

```
                    CORBA_Context cxt_obj,
                    CORBA_long param1,
                    CORBA_Environment * ev
                    );
```

同 CORBA_Environment 类似，CORBA_Context 也是 CORBA 伪对象在 C 语言中的映射。这个伪对象是 Context，OMG 对其如下定义。

例3.18　OMG 对伪对象 Context 的定义。

```
module CORBA {
        interface Context { // PIDL
        Status set_one_value (
                in Identifier prop_name, // property name to add
                in string value // property value to add
                );
        Status set_values (
                in NVList values // property values to be changed
                );
        Status get_values (
                in Identifier start_scope, // search scope
                in Flags op_flags, // operation flags
                in Identifier prop_name, // name of property(s) to retrieve
                out NVList values // requested property(s)
                );
        Status delete_values (
                in Identifier prop_name // name of property(s) to delete
                );
        Status create_child (
                in Identifier ctx_name, // name of context object
                out Context child_ctx // newly created context object
                );
        Status delete (
                in Flags del_flags // flags controlling deletion
                );
        };
        ……
};
```

上下文对象中定义了一组用来处理上下文信息的函数：

- get_values。获得上下文对象的特征值。
- set_one_value。设置上下文对象的某个特征值。
- set_values。设置上下文对象的特征值。
- delete_values。删除上下文对象的特征值。
- delete。删除上下文对象的某个特征值。
- create_child。为上下文对象创建一个子对象。

另外，在 ORB 中还定义了一个函数，用来获取默认的上下文对象：

```
Status get_default_context (out Context ctx );
```

3.2.8 内存的申请与释放

内存的申请和释放与用户采用的 C 编译器以及参数的方向属性密切相关。

如果参数的方向属性为 in，处理起来相对简单，用户全权负责为相应参数申请内存、释放内存。如果参数的方向属性为 out 或 inout，处理就十分复杂，详细情况如表 3.3、表 3.4 所示。

表 3.3 不同数据类型的内存管理方式

Type	Inout	Out	Return
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1
对象引用 ptr	2	2	2
定长度 struct	1	1	1
变长度 struct	1	3	3
定长度 union	1	1	1
变长度 union	1	3	3
string	4	3	3
sequence	5	3	3
定长度 array	1	1	6
变长度 array	1	6	6
any	5	3	3

6 个数字所代表的具体方式如表 3.4 所示。

表 3.4　不同数字所代表的内存管理方式的具体内容

1	一般由调用者负责内存的申请与释放。如果方向属性为 inout，调用者负责初始化参数取值，被调用方可以修改、使用参数取值；如果方向属性为 out，调用者无需初始化参数，被调用方负责为参数赋值；如果是函数返回类型，则按值传递即可。
2	调用者负责为对象引用申请内存。如果方向属性为 inout，调用者负责初始化对象引用取值，被调用方在修改对象引用取值之前，应该首先调用 CORBA_Object_release 释放该对象引用对应的存储空间。因此，如果调用方在传递方向属性为 inout 的对象引用之后还需要使用该参数的原来取值，应该首先调用 CORBA_Object_duplicate 复制该对象引用。一般情况下，调用方需要负责释放对象引用对应的内存，除非该对象引用内嵌在其它 struct 类型参数的内部。这时，调用 CORBA_free 可以自动释放有关内存。
3	如果方向属性为 out，调用者负责申请一个指针并把它传递给被调用者，被调用者负责将指针指向一个有效的参数实例（内存空间）；如果是函数返回类型，被调用者也返回一个类似的指针。在这两种情况下，都不允许被调用者返回空指针，调用者负责释放有关参数的内存。
4	如果是 inout 方向的 string，调用者负责为传入字符串及指向它的 char*指针申请内存，被调用者可以释放传入字符串的内存并为 char*重新赋值，使它指向一个与传出字符串相适应的内存空间。这样，传出字符串的大小不受传入字符串大小的限制。当然，被调用者不允许返回空指针，调用者负责释放内存。
5	对于 inout 方向的 sequence 及 any，赋值或修改操作都有可能释放参数先前对应的内存，具体情况与释放参数时的 boolean 取值有关。
6	对于 out 方向的参数，调用者负责申请一个指向数组切片#的指针，并将该指针传递给被调用者。被调用者负责将指针指向一个有效的数组实例；如果是函数返回类型，被调用者也返回一个类似的指针。在这两种情况下，都不允许被调用者返回空指针，调用者负责释放有关参数的内存。

3.3　OMG IDL 在 C++中的映射

C++是面向对象类型的语言，本身就支持类的定义、对象的构造和析构等面向对象的机制。借助这些语言优势，OMG IDL 在映射为 C++时比映射为 C 语言时更加直观、便捷。

不过，C++的标准本身还在不断发展变化。因此，真正的映射与 C++编译器有关。也正是由于这个原因，在 OMG IDL 的 C 映射中，我们强调映射后的格式；但是在 C++中，我们更强调如何使用映射后的操作。

3.3.1　OMG IDL 数据类型在 C++中的映射

OMG IDL 中的数据类型都可以映射为 C++数据类型。考虑到不同 C++编译器在解释相同名称的数据类型时可能有所区别，映射中没有把某种 OMG IDL 数据类型直接映射为 C++语言中的某种数据类型，而是贯以前缀，表示为一种规范。OMG IDL 数据类型在 C++中的映射如表 3.5 所示：

表 3.5　OMG IDL 数据类型在 C++中的映射

OMG IDL 数据类型	在 C++中映射的数据类型
short	CORBA::Short
long	CORBA::Long
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
float	CORBA::Float
double	CORBA::Double
char	CORBA::Char
boolean	CORBA::Boolean
Octet	CORBA::Octet

通过这种映射方式，CORBA::Long 总对应于一个 32-bit 的整数类型，CORBA::Float 总对应于一个 IEEE32-bit 浮点数类型。无论采用哪种 C++编译器，都应该遵循这些规范。至于其它数据类型的映射情况，在每个 IDL 编译器的文档中一定能够找到。

3.3.2　C++中的内存自动管理数据类型——T_var

C++映射中提供了一种内存自动管理数据的类型，T_var。这种数据类型通过在现有数据类型之后直接添加_var 构成。其标准定义如下所示：

例3.19　C++映射中内存自动管理数据类型的标准定义。

```
// C++
class T_var{
        public:
        T_var();
        T_var(T *);
        T_var(const T_var &);
        ~T_var();
        T_var &operator=(T *);
        T_var &operator=(const T_var &);
        T *operator-> const ();
        // other conversion operators to support
        // parameter passing
};
```

T_var 类型参数能够自动在堆栈中申请内存，当变量超出了作用域后，又会自动析构，释放在堆栈中的内存。显然，这种数据类型对于传递变长度的结构体提供了方便。为了保持编程风格的一致性，定长度结构体也允许被定义为 T_var。

在 C 语言映射中，我们曾经提到过方向属性为 inout 的 string 在传递中的处理方式。在 C++映射中，T_var 为解决这种问题提供了方便：如果将这些参数定义为 T_var，客户、ORB 以及“对象实现”在调用刚开始时，就自动释放了这些参数原来占用的内存，并可以根据实际需要为它们重新分配足够大小的内存空间。如果没有 T_var，这些工作都需要软件开发人员来负责处理。

因此，当 OMG IDL 中定义的参数的方向属性为 out 或 inout 时，我们可以在 C++中把它映射为 T_var。

如果一个结构类型参数被声明为内存自动管理数据类型时，则应该使用->获取其成员项。这种情况可以参看如下所示的例子：

例3.20　一个带 out 参数的简单操作。

```
//这个例子用 OMG IDL 定义
struct my_struct{
        string name;
        float age;
};

void operation(out my_struct s);
```

在 C++中，我们可以如此调用该操作：

例3.21　在 C++中使用 T_var。

```
//这个例子说明在 C++中使用 T_var 的方式
        ……
my_struct a;
my_struct_var b;

a = b;
operation(b);
cout<<"a 的数据成员 name 的取值为：”<<a.name<<endl;
cout<<"b 的数据成员 name 的取值为：”<<b->name<<endl;
```

本例中，a 为一个 my_struct 类型实例；b 为一个 my_struct_var（自动管理内存类型）实例。operation 将 b 作为参数传递。为了获取数据成员，两个变量采用了不同的操作运算符“.”及“->”。最后，a.name 为调用 operation 前的值；b->name 为调用 operation 后的值。

在第九章，我们可以看到 C++Builder 为我们自动生成的 T_var 数据类型定义。

3.3.3　操作在 C++中的映射

现在，假设有这样一个非常简单的接口：

例3.22　拥有一个带参数的操作的简单接口。

```
//这是在 OMG IDL 中定义的接口
interface example1{
        long operation1(in long parameter);
};
```

与 C 语言比较起来，这个接口在 C++中使用时就显得非常的“面向对象”。

例3.23　前一个接口在 C++语言中的调用方式。

```
//这是上一个例子在 C++语言中的调用方式
        ……
example1_var ObjRef;
CORBA::long result, param;

ObjRef = function_to_get_example1_from_ORB();
result = ObjRef->operation1(param);
        ……
```

在 C++语言中，我们可以用两种方式获得对象引用：一种是通过对象指针（T_ptr）得到；另一种是通过内存自动管理类型（T_var）获得。前者在应用中由用户显式地申请、释放内存；后者则在退出作用域后自动析构。本例中，采用了后一种方式。更详细的内容可以参见第九章。

☎ **技巧**　在解决实际问题时，应该有选择地使用 C++语言提供的两种获得对象引用

的方式。如果服务对象构造起来比较简单，或者该对象来回传递经过的不同作用域比较少，或者对服务时间要求不严，均可以采用 T_var 方式。服务器对象将自动释放内存，降低程序员的编程难度。相反，如果服务对象构造起来比较麻烦，或者对象来回传递经过极多的不同作用域，或者服务时间要求严格，就可以采用 T_ptr 方式。服务器对象不会频繁被构造及析构，这样就减少了不必要的互操作损耗。但是，程序员必须全权负责对象内存的申请、释放。

至于不同方向属性的参数在 C++中的映射情况如表 3.6、表 3.7 所示。

表 3.6　不同方向属性的数据类型在 C++中的映射

数据类型	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
对象引用 ptr	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
定长度 struct	const struct&	struct&	struct&	struct
变长度 struct	const struct&	struct&	struct*&	struct*
定长度 union	const union&	union&	union&	union
变长度 union	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
定长度 array	const array	array	array	array slice*
变长度 array	const array	array	Array slice*&	array slice*
any	const any&	any&	Any*&	any*

不同方向属性的_var 类型的参数在 C++中的映射情况如表 3.7 所示：

表 3.7　不同方向属性的_var 类型参数在 C++中的映射

Data Type	In	Inout	Out	Return
对象引用 var	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	Sequence_var&	Sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

在 C++中，除了 T_var 类型参数可以自动管理内存以外，其余参数的内存管理方式与 C 语言中的基本一样。

3.3.4　接口在 C++中的映射

每个接口在 C++中被映射为一组接口类，包括以下一些内容：

- 与接口同名的一个类。这个类就是接口存根对象。

- 在接口名称后直接添加_var 的类。这个类能够自动管理内存。
- 在接口名称或前或后添加一标志，用来说明是接口框架的类。

有的编译器还可能添加一些别的辅助类。比如，在接口名称后直接添加**_out** 的类，这个类可以当作方向属性为 **out** 的参数使用。

另外，在有的 C++编译器中，模块也被映射为一组类，用来限制名称有效作用域。

有关接口映射的详细情况可以参看第九章内容。

3.3.5 在 C++中使用对象引用

例3.24 返回另一个接口 CORBA 操作。

```
//这是一个返回另一接口 CORBA 的操作
#include “example1.idl”

interface example2{
        example1 operation2();
};
```

在本例中，接口 example2 含有一个不带参数的操作 operation2。该操作返回类型为 example1 所定义的接口。而接口 example1 在另外一个 IDL 文件中定义。

在 C++语言中，我们可以如下使用其中的操作及对象引用：

例3.25 用 C++使用操作及对象引用的方式。

```
//这个例子用来说明在 C++中如何使用对象引用
        ……
example1_var ObjectRef1;
example2_var ObjectRef2;
example2_var ObjectRef3;

ObjectRef2 = function_to_get_example2_from_ORB();
ObjectRef3 = function_to_get_example2_from_Program();

ObjectRef1 = ObjectRef2->operation2();
        ……
```

3.3.6 继承在 C++中的映射

由于 C++语言本身支持继承，继承在 C++中的使用显得比较简单。

例3.26 带继承关系的简单接口

```
//这是一个带有继承关系的简单接口
interface example1{
        void operation1();
};

interface example2 : example1{
        void operation2();
};
```

在 C++中，我们有多种方式调用继承接口中的操作。

例3.27 在 C++调用继承接口中的操作。

```
//这是带有继承关系的简单接口在 C++中的使用方式
    ……
```

```
example1  var ObjRef1;
example2_var ObjRef2;

ObjRef2 = function_to_get_example2_from_ORB();
ObjRef1 = function_to_get_example2_from_ORB();

ObjRef2->operation1();
ObjRef2->operation2();

ObjRef1->operation1();
        ……
```

在本例中，用户首先应该注意，由于继承关系的存在，对象引用 ObjRef1 可以引用 example2 类型的对象实例。同时，用户通过 ObjRef2->operation1()激发的是接口 example2 中的操作 operation1；通过 ObjRef2->operation2()激发的是接口 example2 中的操作 operation2；通过 ObjRef1->operation1()激发的是接口 example1 中的操作 operation1。

3.3.7 属性在 C++中的映射

把属性映射为 C++时，实际上是把属性映射为一对同名重载函数，一个无参数，另一个有一个与属性数据类型相匹配的参数。前者读取属性值，后者设置属性值。

例3.28 带有属性的简单接口。

```
//这是带有属性的简单接口
interface example{
        attribute float property1;
        readonly attribute double property2;
};
```

在 C++中，可以如下处理有关属性：

例3.29 在 C++中使用属性的方式。

```
//这个例子说明在 C++中使用属性的方式
        ……
example_var ObjRef;
CORBA::float result1;
CORBA::double result2;

ObjRef = function_to_get_example_from_ORB();
result1 = ObjRef->property1();
result1 *= 100.0;
ObjRef->property1(result1);

result2 = ObjRef->property2();
        ……
```

在本例中，首先把属性 property1 的值放在 result1 中，乘以 100 后重新赋予属性 property1。result2 中最后保存了只读属性 property2 的取值。

如果属性为只读属性，被重载的函数对实际上退化为一个无参数函数。

3.3.8 异常处理在 C++中的映射

OMG 规定，在 C++中，CORBA 标准异常以及自定义异常都必须从 Exception 类中派生。而标准异常又都从 SystemException 中派生，如下所示：

例3.30　CORBA 标准异常在 C++中的定义。

```
enum CompletionStatus {
        COMPLETED_YES,
        COMPLETED_NO,
        COMPLETED_MAYBE
};

class SystemException : public Exception{
        public:
        SystemException();
        SystemException(const SystemException &);
        SystemException(ULong minor, CompletionStatus status);
        ~SystemException();
        SystemException &operator=(const SystemException &);
        ULong minor() const;
        void minor(ULong);
        CompletionStatus completed() const;
        void completed(CompletionStatus);
};

class UNKNOWN : public SystemException { ... };
class BAD_PARAM : public SystemException { ... };
        ……
```

从上面代码可以看出，在 SystemException 内定义了读取、设置 minor 及 completed 的函数。

不难推测，用户自定义异常应该从 UserException 中派生。

经过这种映射后，在 C++中处理 CORBA 异常保持了 C++异常处理的一贯风格，主要由引发异常、捕捉异常及异常处理三部分组成。

不论引发标准异常还是自定义异常，都使用关键字 throw 进行，如下所示：

例3.31　在 C++中引发一个标准异常。

```
//这个例子说明在 C++中如何引发标准异常
        ……
if (execute_function_is_ok)
        cout<<"本函数执行成功了"<<endl;
else
        throw(CORBA::UNKNOWN(NULL, COMPLETED_MAYBE));
        ……
```

被引发的标准异常的名称为 UNKNOWN，在 C 语言映射中，我们已经详细说明了它的定义格式。

本例中，首先检查 execute_function_is_ok 是否为真：如果是真，说明某函数调用成功了；如果为否，就引发该异常。异常的第一个参数为 NULL，因为没有附加信息；第二个参数为 COMPLETED_MAYBE，因为不能确定操作的完成情况。

用同样的方式，我们可以引发自定义异常。

例3.32　一个简单的用户自定义异常。

```
//这是一个简单自定义异常接□
exception MY_EXCEPTION1{
        string reason;
```

```
};

exception MY_EXCEPTION2{
        string reason;
        string time;
};

interface example{
        void operation1() raises(MY_EXCEPTION1,MY_EXCEPTION2);
};
```

在 C++中，上述自定义异常可以如下被引发：

例3.33　在 C++中引发自定义异常。

```
//这是自定义异常在 C++中的引发方式
        ……
if (execute_function1_is_ok)
        cout<<"函数 1 执行成功了"<<endl;
else
        throw(MY_EXCEPTION1("执行函数 1 时发生了错误"));
if (execute_function2_is_ok)
        cout<<"函数 2 执行成功了"<<endl;
else
        throw(MY_EXCEPTION1("执行函数 2 时发生了错误", "19:30PM"));
        ……
```

使用异常时，除了在合适的地方引发异常外，还需要在“事故多发点”或“重要地界”捕捉异常，然后分析异常信息，再进行异常自动处理。

捕捉异常的关键字就是 try、catch。

比如，我们可以在需要的代码处按照下列方式捕捉、处理异常：

例3.34　在 C++中捕捉异常、处理异常。

```
//这是在 C++中的捕捉、处理异常的方式
        ……
try{
        ……
}
catch(const CORBA::UNKNOWN & e){
        cout<<"发生了未知错误，此时执行状态为：”<<e.completed<<endl;
}
    ……
try{
    ……
}
catch(const MY_EXCEPTION1& e){
        cout<<"自定义 1 号异常，原因为"<<e.reason<<endl;
}
        ……
try{
        ……
}
catch(const MY_EXCEPTION2& e){
```

```
    cout<<"自定义 2 号异常，原因为"<<e.reason<<"时间：”<<e.time<<endl;
}
        ……
```

在本例中，我们只是简单地处理异常——把异常信息打印出来。实际上，用户可以用 try 包裹任意代码，用 catch 包裹各种异常处理例程。

3.4 本章小结

OMG IDL 只能定义具有统一标准的接口，却不能用来编写对象实现或者客户的代码。任何能够用来进行 CORBA 编程的语言都必须和 OMG IDL 建立映射。这种映射关系既揭示了接口在程序员眼中的存在方式，也揭示了程序员使用接口的方式。

Java、C++及 C 恐怕是编写 CORBA 软件首选的几种优秀语言。但是，在揭示 CORBA 实现内幕上，C++及 C 可能略胜一筹。本章重点说明了 OMG IDL 在 C 及 C++中的映射。由于 C 语言已经严格标准化，而 C++语言还在不断更新发展，在说明 OMG IDL 与这两种语言的映射时，我们采取了不同策略。

在 OMG IDL 与 C 的映射中，强调数据类型、接口、操作、对象引用、继承、属性、异常、上下文对象在具体语言中的存在形式，以便使大家体会 CORBA 对象如何存在；在 OMG IDL 与 C++的映射中，强调接口、操作、对象引用、继承、属性、异常在具体语言中的使用方式，以便使大家体会如何调用 CORBA 对象。

在 C 语言映射中，本章还说明了 CORBA_Environment、CORBA_Context 在 CORBA 中的作用。它们分别是 CORBA 伪对象 Environment、Context 在 C 语言中的映射。实际上，这两个伪对象在 C++中也依然存在，不过是以类的形式出现。那些与它们有关的操作也以类成员函数的形式出现。

在 C++中，每个 OMG IDL 接口被映射为一组类。其中包括接口存根、接口框架以及其它有关的类。

第 4 章　通过 ORB 动态激发请求

本章内容提要：

- *ORB* 客户端透视
- 动态激发的步骤
- 动态激发接口 *DII*
- “值-名体”(*NamedValue*)
- 伪对象 *NVList*
- 伪对象 *Request*
- 伪对象 *TypeCode*
- 同步(*synchronous*)通信
- 延迟同步(*deferred synchronous*)通信
- 单向(*oneway*)通信
- 接口仓库（*Interface Repository*）
- 请求及函数的签名（*signature*）
- 对象引用初始化及其与字符串的相互转换

4.1　ORB 客户端透视——ORB 中有什么

对象请求代理 ORB 通过接口存根（IDL Stub）、接口框架（IDL Skeleton）、环境伪对象（Environment）、上下文伪对象（Context）等对象成功地分离了 CORBA 客户及对象实现，为实现 CORBA 程序的即插即用提供了可能。

在实现有序分离之后，我们必须把这些看似支离破碎的对象重新有机地联系起来。而这种联系，是借助请求/响应进行的。

请求实际上就是函数、过程或者操作调用。OMG IDL 接口中的每个操作都被映射为一个合法的“请求”。客户通过接口存根，非常容易了解请求的格式、返回内容并激发请求。在这种请求激发方式中，客户在编码时必须首先通过接口存根获得对象实现的有关信息。

但是，我们能否向一个目前还不存在的对象实现发出请求？能否调用某个对象实现中目前还不存在的服务？如果采用前面所说的通过接口存根的激发方式，这两个希望注定会破灭。不过，CORBA 却能够完成，因为 ORB 提供了动态激发机制。

动态激发请求同样依赖于请求。因此，我们不妨考察一下请求的构成要素：

- 被请求操作的名称。
- 被请求对象的对象引用。在 C 语言中，对象引用是每个操作的第一个参数；在 C++中，对象引用与每个成员函数所对应的具体的 **this** 指针有关。
- 环境伪对象（Environment）。在 C 语言中，环境伪对象是每个操作的最后一个参数，可以用来处理异常；在 C++中，环境伪对象是一个全局变量，用户在必要时可以使用它（不过，异常处理时不必如此繁琐）。
- 上下文伪对象（Context）。在 C 语言中，上下文伪对象可能会作为第二个参数传递，并通过一系列例程来处理；在 C++中，上下文对象也可以是一个全局变量，用户在必要时通过其成员函数来处理。
- 被请求操作的零个或多个参数。

显然，即使采用动态激发，上述构成请求的若干要素并不能缺少。因此，为了实现动态激发，CORBA 必须提供两个必要的机制：

- 让客户发出动态请求的机制
- 让客户获取未知操作名称、参数、返回值的机制

实际上，在 ORB 的客户端，确实引入了实现这两个机制的对象群及有关操作：就是动态激发接口（Dynamic Invocation Interface）及接口仓库（Interface Repository）。当然，引入接口仓库后，为我们使用、发布接口信息也提供了便利。

如果我们再次考察 ORB 的客户端，理所当然得到如下所示的“透视”图。

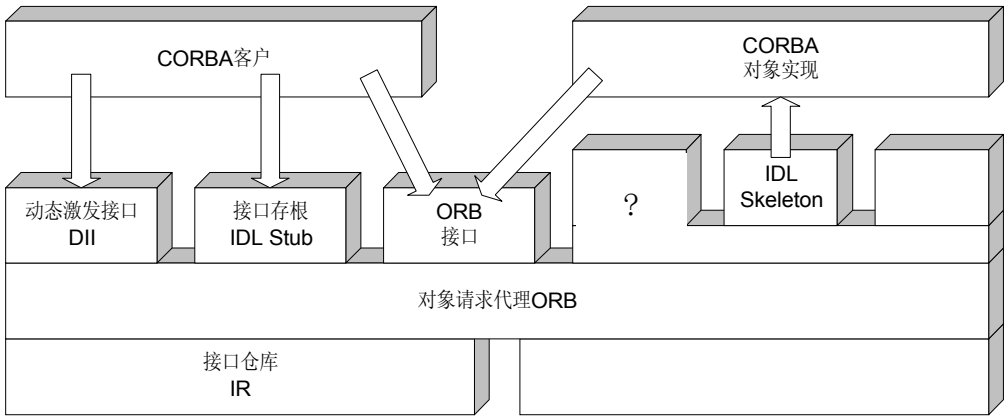

图 4.1　ORB 客户端“透视”

从该图可以看出，对于 CORBA 客户而言，ORB 由四部分组成：

- 接口存根 IDL Stub，用于静态激发各种请求
- 动态激发接口 DII，用于动态激发各种请求
- ORB 接口，用于调用与 CORBA 伪对象有关的操作
- 接口仓库 IR，从中可以获得 IDL 接口的详细信息

4.2　CORBA 的动态激发

我们知道，一个 CORBA 客户可以根据 OMG IDL 接口定义，通过接口存根向 ORB 发出伪客户/服务器请求并获得服务。这种方法的每个请求在编译前已经构造完毕，运行时不能改变，被称为静态激发。

动态激发则允许 CORBA 客户调用 ORB 相关函数，在程序执行过程中构造一些临时的伪客户/服务器请求，经过 ORB 处理后获得服务。这种方法甚至可以用来向软件开发时尚且未知的对象或服务发出请求，是“跨版本”使用 CORBA 对象的根本保证。

动态激发是通过动态激发接口 DII 实现的；而静态激发是通过接口存根实现的。为了讨论方便，接口存根也被称为静态激发接口（Static Invocation Interface）——SII。

SII 与 DII 有很大区别，最主要的一点是，SII 在编译时进行对象类型匹配和操作选择，DII 则把对象类型匹配和操作选择推迟到了执行时刻。这就是所谓的静态类型匹配及动态类型匹配。但是，不论是 SII 还是 DII，都是在执行时才确定对哪一个具体的对象实例进行引用（也就是动态绑定技术）。

显然，在 SII 方式中，编译时有可能发现类型不匹配或者参数错误；而在 DII 方式中则没有这种可能。因此，使用 DII 方式时，应该格外注意异常处理。

采用 SII 请求服务时，除非被调用的函数被声明为 oneway，一般情况下采取的是同步通信方式。在这种通信方式中，客户调用函数后被阻塞，直到有结果返回或者发生异常。

采用 DII 请求服务时，客户不一定需要被阻塞，因为 DII 提供了同步通信、异步通信两种方式。异步通信方式通常又称为延迟同步通信方式。

如果使用 SII 方式请求服务，速度会比直接使用 DII 方式请求同一服务时略快。因此，动态激发可以说是牺牲时间换取灵活的一种请求方式。

注意 在 COM/DCOM 中，也有可以实现静态请求或动态请求的机制。采用 IDispatch 方式激发 COM/DCOM 的方法时，不需要客户在编译时确定操作对象、操作名称及操作参数，实际上就是动态激发。然而，不得不承认，在 Visual Basic、Delphi 中使用 IDispatch 方式，动态激发的味道没有被正确显示出来：因为它们在调用方法时往往采用“name.operation(params)”的方式，而用户程序中都将对象名称、操作名称、参数赋予了具体的取值并进行了编译。但是，如果你使用过 C++ Builder，就会发现，在 Inprise 的这个开发工具中，调用 COM/DCOM 的方法是采用“Variant.OleFunction (“name”,param)”的形式，我们可以在执行中决定 Variant 代表的对象，函数的名称及参数项。这绝对是动态请求的极好证明。另外，在 Delphi 及 C++ Builder 中，我们还可以通过类型库、虚拟方法表 VMT 来激发 COM/DCOM 的方法，这种情况下，实际上就是静态激发。这时，处理速度会提高许多。在第九章中，我们将举例说明 CORBA 的动态激发过程。

4.3 动态激发接口 DII

动态激发接口由以下三部分例程构成：

- 请求管理例程。包括要求 ORB 动态建立一个服务请求，给服务请求添加参数，删除无用的服务请求等例程。
- 各种异步通信例程。包括同时发送一个、多个请求，接收响应判断请求是否完成，确定下一个响应等例程。
- 列表管理例程。为参数列表申请内存、进行初始化、释放内存的例程。

通过这些例程，我们可以按照如下步骤建立动态请求：

- 确定需要采用动态请求去实现的服务及服务提供对象
- 动态获取对象的接口信息
- 构造请求伪对象
- 激发请求伪对象、接收响应

4.3.1 定位服务及服务提供对象

在分布式软件中，程序模块分散在整个网络之上，用户必须通过一定的机制来查询、获得每个对象实现能够提供的服务以及提供这些服务的对象实例。这种机制是 CORBA 基本服务中的一个专题，被称为对象洽谈服务（Trader Service）。

注意 对象实现是用户编写的分布式程序，对象实例是在正在运行中的对象实现。

我们可以从这种服务的名称中想象它的构成形式：在某些集中的“网络空间”，罗列一系列对象实现的名称，同时标明各个对象能够完成的服务、实现服务的品质及价位、对象实例的分布站点……用户可以在各个对象实现之间比较、选择。如果决定“成交”，用户可以从对象洽谈服务中获得一个能够提供相应服务的对象实例的对象引用。

对象洽谈服务可以采用专门的浏览器交互进行；也可以通过编程接口直接实现。反正，

它是分布式对象实现之间召开“贸易洽谈会”的一种机制。

确定具体的对象引用之后，我们更加关心如何获得这个对象的接口信息，以便能够有效地向对象发出请求、激发操作、完成服务。

4.3.2 提取对象实现的接口信息

对于任何一个对象实例而言，它并不关心客户发出的请求是通过接口存根静态激发的还是通过 DII 动态激发的，只要符合 OMG IDL 的接口定义，都会执行服务并返回响应。在静态激发时，接口定义可以从接口存根中获得；在动态激发时，接口定义应该从接口仓库（Interface Repository）中获得。

接口仓库 IR 中记录了许多分布式对象的接口定义，根据已经确定好的对象引用，用户可以获得相应对象的接口。在提取接口时，首先应该调用伪对象 Object 中定义的操作 get_interface，获得 IR 中“接口定义对象”InterfaceDef 的对象引用；然后通过一系列接口仓库例程，获得操作的名称、参数及返回类型等等信息。具体内容在接口仓库中再详细介绍。

> **注意** 也许有人会质疑对象洽谈服务与接口仓库为什么会同时存在？的确，通过洽谈服务，我们可以获得各个操作的功能，每个参数的类型、取值范围、运行性能以及使用指南等各种有用信息。但是，这种服务是“面向人类”的。它不需要一定的格式，不需要一定的标准，是一种自由风格的交流模式。相比之下，接口仓库提供了一种“面向程序”的交流方式。对象实现与对象实现之间、程序模块与程序模块之间通过接口仓库中被标准化的接口彼此沟通，以便完成逻辑上的集成。

4.3.3 参数列表 NVList 及“值-名体”NamedValue

参数是请求中不可缺少的成分，动态构造请求同样要为激发服务提供足够的参数。而且，动态构造请求很大程度上就是在设法建立参数传递表。在 CORBA 中，操作参数表可以通过三种方式建立：

- 自动建立参数表。调用伪对象 ORB 的函数 create_operation_list，自动建立一个参数列表。这个列表一次建立成功，包括参数的名称、类型及方向属性。
- 逐步建立参数列表。调用伪对象 Request 的函数 add_arg，每次添加一个参数，包括参数的名称、类型、方向属性及具体取值。
- 手动建立参数列表。直接调用伪对象 ORB 的列表操作函数 create_list，手控生成参数列表，包括参数的名称、类型及方向属性。

无论采用哪种方式，参数表都被记录在一个被定义为伪对象 NVList 类型的列表中。该列表中的每个元素是一个“值-名体”（NamedValue），表示与操作中某个参数相关的各种信息。如果采用 OMG IDL，每个“值-名体”的定义如下所示：

例 4.1 “值-名体” ——NamedValue 的定义。

```
//这个例子是“值-名体”——NamedValue 的定义
typedef unsigned long Flags;

struct NamedValue {
        Identifier name; // argument name
        any argument; // argument
        long len; // length/count of argument value
        Flags arg_modes;// argument mode flags
};
```

其中，name 表示参数的名称，argument 表示参数值，len 表示参数的长度或数目（数

目是对于数组而言的），arg_modes 表示参数的方向属性，取值可以是 ARG_IN、ARG_OUT 或 ARG_INOUT。

至于伪对象 NVList，稍后再详细说明。

对于建立参数表的三种不同方式，用户构造请求的步骤也有所区别：

- 如果是自动建立参数列表方式，用户还需要为每个参数赋予具体取值，然后再构造请求伪对象；
- 如果是逐步建立参数列表方式，用户首先应该构造好一个不含任何参数项的请求伪对象，再逐步添加参数直到符合接口规定；
- 如果是手动建立参数列表的方式，用户也需要为每个参数赋予具体取值，然后再构造请求伪对象。

这几种方式的详细情况分别说明如下。

4.3.4 自动建立参数列表构造请求

在 CORBA 中，有一个采用 PIDL 定义的伪对象 ORB，含有如下一个操作：

例4.2 伪对象 ORB 中自动建立参数列表的函数。

```
//这个例子说明 ORB 中自动建立参数列表的函数
interface ORB{
        ……
        Status create_operation_list(
                                in OperationDef oper,
                                out NVList new_list
                                );
        ……
};
```

在这个函数中，oper 是一个 OperationDef 类型的对象引用，取值与具体接口以及具体操作有关，记录了特定操作的各种信息。该“操作定义对象”OperationDef 从接口仓库中获得。

根据输入参数 oper，create_operation_list 自动生成一个参数列表 new_list。new_list 中的每个元素是一个上节所示的“值-名体”，记录了被请求操作的参数信息。

随后，客户程序可以为每个参数赋值。如果方向属性为 **in**，一般情况下直接赋值；如果方向属性为 **out**，一般情况下为参数建立接收缓冲区；如果方向属性为 **inout**，一般情况下为参数建立接收缓冲区并进行初始化（参看上章内容）。

最后，我们可以调用伪对象 Object 中的请求构造函数 create_request 构造出可以被动态激发的请求伪对象。

例4.3 伪对象 Object 中构造请求的函数。

```
//这个例子说明 Object 中构造请求的函数
interface Object{
        ……
Status create_request(
                        in Context ctx,
                        in Identifier operation,
                        in NVList arg_list,
                        inout NamedValue result,
                        out Request request,
                        in Flags req_flags,
                        );
```

```
        ……
};
```

其中，ctx 用来表示可能需要使用的上下文伪对象的对象引用，operation 代表被请求操作名称的字符串，arg_list 是由伪对象 ORB 中自动建立参数列表函数 create_operation_list 输出的参数列表，result 也是一个“值-名体”类型的参数，用来记录请求的返回结果（如果返回类型为 void，可以用 NULL 表示），request 是被构造的请求伪对象的对象引用，用户可以根据需要在合适的时候采用合适的方法激发这个请求伪对象，req_flags 目前可以取 0 值或 OUT_LIST_MEMORY，用来表示在本操作中如何管理与参数列表有关的存储区。

如果不发生异常，调用该函数后可以获得一个动态构造的服务请求伪对象 request。

采用自动建立参数列表方式构造请求的步骤可以概括为：获得服务提供者的对象引用→获得 IR 中“操作定义对象”的对象引用→自动建立参数列表→赋值→构造请求。

4.3.5 逐步建立参数列表构造请求

采用自动建立参数列表的方式构造请求是比较方便的。但是有时侯，用户可能希望逐步建立参数列表。

比如，某个站点的接口仓库突然发生故障或根本就没有启动，但是，由于某种原因，我们必须立即构造好一个动态请求，否则就会引起死锁。

再比如，假设对象实现中有两个拥有相同名称，但是参数数目不同的多态函数 operation1(prarm1,prarm2,param3)及 operation1(prarm1,prarm2)，用户需要根据参数的具体取值选择被请求的多态服务，动态激发请求……

在这些情况下，自动建立参数列表构造请求的方式就不太适合。为此 CORBA 中也允许用户采取逐步建立参数列表的方式构造请求。

采用逐步建立参数列表的方式构造请求，用户首先需要构造出无参数的请求。

构造无参数的请求实际上也是通过调用伪对象 ORB 中的函数 create_request 来实现的。不过，在这种情况下，NVList 类型的参数 arg_list 被 NULL 代替，表示目前参数数目为零，参数列表为空。

在获得没有参数列表的请求伪对象之后，用户可以按照自己的速度准备参数、为每个参数赋值。同样，如果参数的方向属性为 in，一般情况下直接赋值；如果方向属性为 out，一般情况下为参数建立接收缓冲区；如果方向属性为 inout，一般情况下为参数建立接收缓冲区并进行初始化。

不同之处是，每当准备好一个参数，用户可以调用请求伪对象 Request 中的操作 add_arg 为请求添加参数。该操作的定义如下。

例4.4　请求伪对象 Request 中的 add_arg 及 delete 函数。

```
//这个例子用来说明请求伪对象中的 add_arg 及 delete 函数
interface Request{
        ……
        Status add_arg(
                in Identifier name,
                in TypeCode arg_type,
                in void * value,
                in long len,
                in Flags arg_flags
                );
        Status delete();
        ……
};
```

在函数 add_arg 中，name 表示待添加参数的名称，arg_type 表示参数对应的数据类型码，value 表示待添加参数值，len 表示待添加参数的长度或数目（数目是对于数组而言的），arg_flags 表示待添加参数的方向属性，取值可以是 ARG_IN、ARG_OUT、或 ARG_INOUT。

通过请求伪对象中的 add_arg，我们可以逐步为请求添加足够的参数。

有时，我们还可以在必要的时候，调用请求伪对象中的 delete 函数，释放这个请求。当然，通过自动建立参数列表方式获得的请求对象也可以采用相同的方式被释放。

采用逐步建立参数列表方式构造请求的步骤可以概括为：获得服务提供者的对象引用→构造请求→赋值→逐步建立参数列表。这种方式可以不依赖接口仓库 IR。

4.3.6 伪对象 TypeCode

通过伪对象 Request 中的 add_arg 为请求添加参数时，需要确定参数的数据类型码。

参数的数据类型码由 CORBA 伪对象 TypeCode 表示。它代表了被激发操作中有关参数的类型和属性的类型。可以是基本的数据类型，也可以是自定义的构造数据类型。通常，TypeCode 可以从接口仓库 IR 中获得或者从 OMG IDL 编译器中产生。

在 CORBA 中，TypeCode 具有以下功能：

- 在动态激发时，用来代表参数的数据类型
- 在接口仓库中，用来规范表达操作声明中的数据类型
- 对于 any 类型的参数，用来说明最终是哪种数据类型

为了实现这些功能，伪对象 TypeCode 中也定义了不少的操作

例4.5　伪对象 TypeCode 的定义。

```
module CORBA {
        enum TCKind {
            tk_null, tk_void,
            tk_short, tk_long, tk_ushort, tk_ulong,
            tk_float, tk_double, tk_boolean, tk_char,
            tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
            tk_struct, tk_union, tk_enum, tk_string,
            tk_sequence, tk_array, tk_alias, tk_except
            };
        interface TypeCode {
            exception Bounds {};
            exception BadKind {};
            boolean equal (in TypeCode tc);
            TCKind kind ();
            RepositoryId id () raises (BadKind);
            Identifier name () raises (BadKind);
            unsigned long member_count () raises (BadKind);
            Identifier member_name (in unsigned long index)
                    raises (BadKind, Bounds);
            TypeCode member_type (in unsigned long index)
                    raises (BadKind, Bounds);
            any member_label (in unsigned long index)
                    raises (BadKind, Bounds);
            TypeCode discriminator_type () raises (BadKind);
            long default_index () raises (BadKind);
            unsigned long length () raises (BadKind);
            TypeCode content_type () raises (BadKind);
```

```
            long param  count ();
            any parameter (in long index) raises (Bounds);
        };
};
```

其中，一些主要的函数分别可以实现以下功能：

- equal，用来判断两个 TypeCode 伪对象是否相等
- kind，确定 TypeCode 伪对象所代表的数据类型
- id，获取 tk_objref、tk_struct、tk_union、tk_enum、tk_alias 以及 tk_except 类型数据的接口仓库标志号
- name，获取 tk_objref、tk_struct、tk_union、tk_enum、tk_alias 以及 tk_except 类型数据的名称
- param_count，构造数据类型需要使用一定的参数才能够明确表达。比如，sequence<double,100>应该和 sequence<double,99>不同，这就需要用一定的参数来说明伪对象 TypeCode 实际代表的数据类型的不同。这些参数通常放在一个与 TypeCode 有关的参数表中，而该参数表中参数的数目可以通过本函数获得。具体情况可以参看表 4.1
- parameter，确定伪对象 TypeCode 有关参数表中对应参数的取值

表 4.1　不同数据类型在 TypeCode 中对应的参数列表

数据类型	参数列表
tk_null	*NONE*
tk_void	*NONE*
tk_short	*NONE*
tk_long	*NONE*
tk_ushort	*NONE*
tk_ulong	*NONE*
tk_float	*NONE*
tk_double	*NONE*
tk_boolean	*NONE*
tk_char	*NONE*
tk_octet	*NONE*
tk_any	*NONE*
tk_TypeCode	*NONE*
tk_Principal	*NONE*
tk_objref	{interface id}
tk_struct	{struct name, member name, TypeCode, ... (repeat pairs) }
tk_union	{union name, discriminator TypeCode, label value, member name,TypeCode, ... (repeat triples) }
tk_enum	{enum name, enumerator name, ... }
tk_string	{maxlen integer}
tk_sequence	{TypeCode, maxlen integer }
tk_array	{TypeCode, length integer }
tk_alias	{alias name, TypeCode }
tk_except	{except name, member name, TypeCode, ... (repeat pairs) }

4.3.7 手动建立参数列表构造请求

CORBA 中也允许用户手动建立参数列表，动态构造请求。

在伪对象 ORB 中，还有一个手动建立参数列表的函数：

例4.6　伪对象 ORB 中手动建立参数列表的函数 create_list。

```
//这是伪对象 ORB 中手动建立参数列表的函数 create_list 的定义
interface ORB{
        ……
        Status create_list(
                in long count;
                out NVList new_list
                );
        ……
};
```

调用该函数时，用户给 count 赋予具体的取值，说明需要产生拥有 count 个“值-名体”的 NVList 类型参数列表。

随后，客户程序可以为每个参数赋值。如果方向属性为 in，一般情况下直接赋值；如果方向属性为 out，一般情况下为参数建立接收缓冲区；如果方向属性为 inout，一般情况下为参数建立接收缓冲区并进行初始化。

与此同时，用户应该手动为 new_list 中的每个“值-名体”赋值。

最后，用户可以调用伪对象 Object 中的请求构造函数 create_request 构造出可以被动态激发的请求伪对象。这次，函数的第三个参数 arg_list 对应于我们手动建立起来的参数列表 new_list。

如果不发生异常，用户也会获得一个动态建立的请求伪对象。

采用手动建立参数列表方式构造请求的步骤可以概括为：获得服务提供者的对象引用→手动建立参数列表→赋值→构造请求。这种方式，也不必依赖接口仓库 IR。

另外，伪对象 NVList 的定义如下所示：

例4.7　伪对象 NVList 的定义。

```
interface NVList { // PIDL
        Status add_item (
        in Identifier item_name, // name of item
        in TypeCode item_type, // item datatype
        in void *value, // item value
        in long value_len, // length of item value
        in Flags item_flags // item flags
        );
        Status free ( );
        Status free_memory ( );
        Status get_count (
                        out long count // number of entries in the list
                        );
};
```

NVList 中的这些函数主要用来管理参数列表：

- add_item，为对应的参数列表添加一个“值-名体”
- get_count，获得相应参数列表中“值-名体”的个数
- free，释放整个参数列表对应的存储空间
- free_memory，释放参数列表中方向属性为 out 的有关参数的存储空间。不过这时，

create_request 中的最后一个参数应该取 OUT_LIST_MEMORY，表示本请求对应的参数列表可以使用 free_memory 函数。

不论采用自动建立参数列表方式、逐步建立参数列表方式还是手动建立参数列表方式，最终都是为了获得动态建立的请求伪对象。对于请求伪对象，我们又可以采用同步通信、异步通信或单向通信三种方式激发。

4.3.8 同步激发动态请求

在请求伪对象 Request 中，有一个 invoke 函数，定义如下：

例4.8 请求伪对象 Request 中的 invoke 函数。

```
//这个例子说明 Request 中 invoke 函数的定义
interface Request{
        ……
Status invoke(
        In Flags invoke_flags
        );
        ……
};
```

如果请求伪对象涉及的操作本身没有被定义为 oneway，调用这个函数后，对应的请求/响应将以同步通信方式实现。这时，执行效果与从 SII 直接调用对应操作的效果一致，都采用阻塞的方式进行。

所谓阻塞方式是指用户激发请求后，程序控制权也被移交给 ORB，在请求完成或发生异常之前，用户不能再作任何别的处理。

Windows 编程中模态对话框是理解阻塞方式一个非常好的比拟：用户一旦激活这种对话框，在按“确定”或“取消”键之前，无法进行该对话框以外的任何操作。

采用 invoke 激发的请求收到响应以后，程序控制权又交还给客户。这时，参数列表中所有 out 及 inout 类型参数的值都已经经过必要的修改。而且，如果被请求的操作有返回值，应该被记录在 result 中。

当然，采用 invoke 激发的请求也有可能以发生异常而告终。

注意 只有当请求伪对象涉及的操作没有被定义为 oneway 时，调用 invoke 才可以同步激发这个动态请求。同样，只有当操作没有被定义为 oneway 时，采用 SII 直接调用才可以按同步通信方式进行。

4.3.9 异步激发动态请求

CORBA 中的异步通信又称作延迟同步通信（deferred synchronous）。任何从 SII 中激发的操作都不可能采用异步通信方式（CORBA3.0 正在做这方面的修改）。

在请求伪对象 Request 中，send 函数用来异步激发动态请求，其定义如下：

例4.9 请求伪对象 Request 中的 send 函数。

```
//这是伪对象 Request 中 send 函数的定义
interface Request{
        ……
Status send(
        in Flags invoke_flags
        );
        ……
};
```

调用该函数激发请求伪对象后，控制权立刻交还给客户。现在，客户没有被阻塞，可以去完成任何别的处理。

这时，我们一定会提出一个问题：怎样才能知道被激发的请求已经完成或者发生了异常？这个问题，可以通过请求伪对象 Request 中的 get_response 函数解决。

例4.10　请求伪对象 Request 中的 get_response 函数。

```
//这是伪对象 Request 中 get_response 函数的定义
interface Request{
        ……
Status get_response(
        in Flags response_flags
        );
        ……
};
```

在合适的时候，用户可以调用这个函数，检查对应的请求是否完成或发生了异常。函数中的 response_flags 有两个取值：RESP_NO_WAIT 及 NULL。

如果是 RESP_NO_WAIT，该函数在执行后立刻返回。用户可以通过其返回值 Status 查看请求的执行状况。假如该请求仍旧没有完成，也没有发生异常，可以在以后继续调用本函数进行查询。这种方式实际上是轮询（poll）方式。

如果是 NULL，函数调用后被阻塞，直到对应的请求执行完毕或发生异常。

注意　使用 get_response 并不能及时检测请求伪对象的执行状况。比如，在某一时刻，用户通过该函数发现请求已经执行完毕，这并不说明该请求就是在这一时刻完成的，只能说明在这个时刻以前的某个时刻，请求已经完成。同样，在某一时刻，如果用户通过该函数发现请求发生了异常，也无法直接判断这个异常发生的时间（除非异常中有相关信息）。当然，如果将 response_flags 取为 NULL，情况会有所好转，但也没有彻底改变这种情况。

由于异步激发动态请求时，客户程序没有被阻塞，用户很有可能在激发一个请求后又去激发别的请求。这样，这些被依次激发的请求将并行地在网络环境中执行。

在 CORBA 中，为了便于客户在运行时依次激发多个请求，特地定义了一个函数：send_multiple_requests。这个函数没有也不可能被归属于请求伪对象。在 C 语言中，其映射形式如下所示：

例4.11　send_multiple_requests 在 C 语言中的映射形式。

```
//这个例子说明 send_multiple_requests 在 C 语言中的映射形式
Status send_multiple_requests(
                        CORBA_Request requests[],
                        CORBA_Environment * ev,
                        CORBA_long count,
                        Flags invoke_falgs
                        );
```

需要并行激发的请求伪对象放在 requests 数组中，count 标明这批请求的数目。调用该函数后，这批请求伪对象会自动地被并行激发。

同样，并行激发请求伪对象后，也需要在合适的时候检查请求的执行情况。这时，可以采用 get_next_response 函数。该函数也没有归属于请求伪对象，其 C 语言映射形式如下所示：

例4.12　get_next_response 在 C 语言中的映射形式。

```
//这个例子说明 get_next_response 在 C 语言中的映射形式
```

```
Status get_next_response(
                        CORBA_Environment * ev,
                        Flags response_falgs,
                        CORBA_Request * request
                        );
```

与 get_response 相似，该函数的 response_falgs 有两个取值：RESP_NO_WAIT 及 NULL。

如果是 RESP_NO_WAIT，该函数在执行后立刻返回。用户可以通过其返回值 Status 查看是否有请求执行完毕或发生异常，假如所有请求都没有完成也没有出错，可以在以后继续调用该函数进行查询。否则，用户可以通过 request 参数获得执行完毕或发生异常的请求。

如果是 NULL，函数调用后被阻塞，直到至少有一个请求执行完毕或发生异常后才返回。用户可以通过 request 参数获得执行完毕或发生异常的请求。

send_multiple_requests 及 get_next_response 函数在 C++中被映射到 ORB 类中，这里不再赘述。

注意 在 COM/DCOM 中，客户/服务器之间没有直接采用异步通信方式的可能。以前，网络函数的调用主要是采用 RPC 方式进行，实际上都是同步通信方式。可是在现实生活中，有许多现象都可以采用异步的方式进行：e-mail、信息发布、通知、广告、聊天等等，都并不要求接收方立刻回音。采用异步通信方式激发动态请求，使我们在软件开发时可以更加接近现实世界的需要。COM/DCOM 为了弥补自身不具备异步通信方式的不足，正在计划引入消息队列（Message Queuing）服务，提供一系列的对象接口、API 及服务。他们将这项计划命名为 MSMQ，昵称为 Falcon，并解释为：这是软件之间的 e-mail。

4.3.10 单向激发动态请求

同步通信方式或异步通信方式都需要接收方作出响应。不过一个是一直守侯到接收方发出响应；一个是不定期的查询接收方的响应。单向通信方式则根本不要求接收方作出响应。

如果在 OMG IDL 的定义中，操作被声明为 oneway，采用 SII 静态激发该操作时，就是采取的单向通信方式。客户通过接口存根“通知”ORB 请求该操作，然后立刻可以去处理别的事物。而该操作执行完后也没有反馈信息。

如果操作被声明为 oneway，采用 invoke 函数动态激发时，也会采取单向通信方式进行。客户立刻可以去处理别的事物，同时不会直接接收到任何与该操作有关的信息（标准异常除外）。这是单向激发动态请求的一种情况。

但是，如果在 OMG IDL 的定义中，操作没有被声明为 oneway，我们也可以在动态激发时将它强迫为单向请求。

在请求伪对象 Request 的 send 函数以及不属于任何伪对象的 send_multiple_requests 函数中，都有一个参数 invoke_flags。如果将它的取值设为 INV_NO_RESPONSE，意味着客户不需要查询这个请求或这组请求的响应；同时，也意味着这个请求或这组请求中的方向属性为 out 或 inout 的参数不需要修改。这时，这个请求或这组请求对应的操作就是以单向通信方式激发的。

注意 如果把 send 函数中的 invoke_flags 取值设为 INV_NO_RESPONSE，只能将对应请求伪对象所涉及的操作强迫为单向方式；而把 send_multiple_requests 函数中的 invoke_flags 取值设为 INV_NO_RESPONSE 时，则将把由它激发的整组请求伪对象所涉及的全部操作均强迫为单向方式。

send_mul-

tiple_requests 的 invoke_flags 还可以设置为 INV_TERM_ON_ERR，表示如果有一个请求发生错误，所有剩下的请求都将被停止激发。

4.4 接口仓库 IR

顾名思义，接口仓库（Interface Repository）就是 CORBA 分布式对象接口定义的集中存储“区域”，用来辅助用户使用、发布、管理其中记录的对象接口。当然，这种集中存储仅仅是逻辑上的集中，并不要求所有接口定义都保存在同一个文件、同一个数据库中，也并不要求所有接口都驻留在相同的站点。

对于 ORB，接口仓库 IR 主要提供以下功能：

- 为不同 ORB 之间的互操作提供实现途径。关于这一点，将在 CORBA 互操作中详细说明。
- 供 ORB 用来检查请求的签名（signature）。所谓请求的签名就是指请求涉及函数的参数名称、参数数据类型、参数方向属性及函数返回类型。无论是通过 SII，还是 DII，ORB 在通过接口框架激发对象实现中的操作时，都需要检查请求的签名。
- 供 ORB 用来检查接口之间的继承关系。在 OMG IDL 的 C 语言映射中，我们曾说明，具有继承关系的接口在映射后，其对象引用参数有可能“多样化”，别的语言也存在类似的情况。ORB 则坚信一条原则：多样化并非为所欲为，多态应该符合继承关系。

通过接口仓库自带的各种工具，用户还可以实现以下功能：

- 在整个互联网范围内安装、分布所有 CORBA 对象。分布式软件不需要程序模块集中驻留，但是，我们必须提供足够的信息让用户了解对象的存在和使用途径。这些服务于软件集成的信息就在接口仓库中。也正是这些信息，才使得 CORBA 对象能够一处安装，到处使用。
- 以更加直观的方式检索接口的定义、接口的继承关系图。通过一些工具，用户可以直接得到接口层次图和接口定义。使用过 Borland C++的用户一定记得功能强大的类浏览器；使用过 Visual C++的用户一定也记得学习 MFC 类库时类层次图的重要性。
- 直接从接口仓库中获得相关对象的接口存根及接口框架。在软件开发时，用户也许曾经碰到过“未找到 XXX 模块，无法链接”的错误。如果没有接口仓库，我们在继承或使用现存的 CORBA 对象时，就需要在编译前指明被包括的接口定义文件、对象实现编码或库的位置，否则，一定会使编译器找不到相关文件。通过使用接口仓库，这些细节问题就会被隐蔽起来，用户可以直接下载接口定义并生成 CORBA 对象的接口存根及接口框架。

接口仓库提供的功能及工具主要依赖于 CORBA 供应商。在第九章，我们将具体介绍 VisiBroker 的接口仓库。

注意 CORBA 遇到的上述问题在 COM/DCOM 中也会遇到。不过，COM/DCOM 与 CORBA 采取了截然不同的策略：通过注册，COM/DCOM 分布式对象的信息被登录在每台机器的系统注册信息表中。当然，只有 Windows 才有符合要求的注册信息表。虽然一些别的公司已经在致力于开发 Unix 下的 COM/DCOM，但 Mi-

crosoft 设计 COM/DCOM 的本意就是强迫这些分布式对象必须运行在 Windows 平台中。另外，如果用户希望继承已有的 COM/DCOM 对象，需要从类型库*.TLB 中获取有关对象的接口定义信息并链接该库。通过对比，我们可以认为，CORBA 中的接口仓库就是一个动态的注册信息表和一个动态的

类型库。

4.4.1 使用接口仓库的两种途径

用户可以通过两种基本途径来提取、使用接口仓库中的接口信息：

- 通过 ORB 附带的工具。这些工具非常像类浏览器、Oleview 等程序开发的附件，借助可视化的手段，供用户查看、修改、导入（Import）、导出（Export）、注册、卸载接口定义信息。
- 通过 OMG 规定的 IR 接口。这种方式允许用户直接通过一些底层函数来控制接口仓库，不过，必需进行编程。大多数用户不会喜欢这种方式，也不必采用这种方式。

由于接口仓库主要由 CORBA 厂商提供，不同产品之间的性能恐怕有较大差异。OMG 仅仅规定了接口仓库必需的功能，但绝没有限制接口仓库可以具备的功能。因此，选择哪种接口仓库是一个由用户自己去选择、对比和决定的领域。

4.4.2 如何组织接口仓库

Windows 注册信息表的组织方式非常简单：一个平台对应一个信息注册表。接口仓库却采取了更加灵活多样的组织方式：

- 一个 ORB 至少对应一个接口仓库
- 一个接口仓库可以被多个 ORB 共享
- 一个 ORB 可以使用多个接口仓库

接口仓库这种灵活的组织方式，使我们可以按照多样化的形式构筑分布式软件世界：如果我们是软件开发人员，可能需要维护一个 Release 版本的接口仓库及一个 Beta 版本的接口仓库，用来区别正式公布的分布式对象和正在测试的分布式对象；如果我们是软件销售人员，可能需要维护一个 Free 类型的接口仓库及一个 Charge 类型的接口仓库，用来区别免费使用的分布式对象和收费使用的分布式对象……

我们还可以通过接口仓库标志号 RepositoryID 来获取接口仓库中不同接口的信息。在局域网及单机范围内，接口仓库标志号可以采用字符串名称方式定义；在更大范围的互联网上，我们可以采用基于通用统一标志 uuid（universally unique id）的格式。

目前，OMG 规定接口标志号由以下三部分组成：

- 前导标志。该标志被指定为字符串“IDL”。
- 接口信息名称。这部分由用“/”分割的字符串构成，分级表达了模块名称、接口名称、操作名称等一系列信息。比如，“Module1/Interface1/Operation2”。
- 版本号。由主版本号、“.”及小版本号构成。比如，“1.0”。

以上三部分字符串用“:”连接后就是一个完整的接口仓库标志号。

比如，“IDL:Module1/Interface1/Operation2:1.0”就是一个完整的接口仓库标志号。

接口仓库标志号是接口信息的“索引关键字”。在实际应用中，ORB 总是假设具有相同接口标志号的接口定义相同。因此，在组织接口仓库时，我们应该格外注意这个要求，否则，很可能导致 CORBA 程序异常或运行结果紊乱。

CORBA 允许用户为接口信息指定接口仓库标志号，有关方法不再赘述。

4.4.3 接口仓库的存储

接口仓库是接口信息的集中存放区域。因此，任何厂家的接口仓库都需要解决两个基本问题：

- 如何存储这些接口信息
- 如何将这些接口信息结构化，以便管理

在具体存储机制的选择上，接口仓库没有作任何规定。用户可以使用与操作系统及所在平台最合适的方法。比如，可以采用一般格式的数据文件，可以采用注册信息表的样式，也可以采用类似初始化文件的“名称-取值对”，还可以构筑在数据库系统甚至是网络数据库系统之上。相对而言，采用数据库系统或分布式数据库系统是一种较好的存储方式。

不论如何存储数据信息，接口仓库都必须满足一点要求：接口仓库中的对象应该是具有某种形式的持续流对象。持续流对象是一种可以永久保持数据信息的对象，这样，即使关闭接口仓库，它所存储的接口信息也不会丢失。更详细的内容我们将在以后专门讨论。

4.4.4 接口仓库的结构

在面向对象技术中，我们希望把各种复杂的信息划分为具有一定层次结构的“类”来表达。同样，接口仓库也为存储、管理接口信息提供了类层次结构。这些类与 OMG IDL 语言中的语法规则密切相关（为了方便讨论，我们称之为“语法描述”对象），主要有以下类型：

- Repository，用来表示接口仓库本身。每个接口仓库对象都拥有自己的 ID，其它“语法描述”对象都可以包含在特定的接口仓库对象之中。
- ModuleDef，用来表示模块定义信息。每个模块定义对象都拥有自己的名字，同时也可以包含一些合适的“语法描述”对象。
- InterfaceDef，用来表示接口信息。每个接口定义对象都拥有自己的名字、继承关系，可以包含定义接口的各种语法所对应的“语法描述”对象。
- AttributeDef，用来表示属性信息。每个属性定义对象都拥有自己的名字、类型及属性的模式（表示属性是否具有只读性）。
- OperationDef，用来表示操作信息。每个操作定义对象都具有自己的名字、返回类型和有关参数等信息。
- ConstantDef，用来表示常数信息。每个常数定义对象都拥有自己的名字、数据类型及对应取值。
- ExceptionDef，用来表示异常信息。每个异常定义对象都拥有自己的名字及构成成员等信息。
- StructDef，用来表示结构信息。每个结构定义对象都拥有自己的名字和构成成员等信息。
- UnionDef，用来表示联合信息。每个联合定义对象都拥有自己的名字、判别信息以及构成成员等信息。
- EnumDef，用来表示枚举信息。每个枚举定义对象都拥有自己的名字以及对应的取值信息。
- AliasDef，用来表示 Typedef 中定义的信息。每个别名定义对象都拥有自己的名字及所对应的数据类型信息。
- PrimitiveDef，用来表示基本数据信息。基本数据类型定义对象可以表示 **long**、**short** 等基本数据类型，通常被预先放置在接口仓库中。
- StringDef，用来表示字符串信息。字符串定义对象并不需要拥有自己的名字，通常记录了它的长度（0 表示无界字符串）。
- SequenceDef，用来表示序列信息。序列定义对象也不需要拥有自己的名字，通常记录了它的长度（0 表示无界序列）和元素类型等信息。
- ArrayDef，用来表示数组信息。数组定义对象不需要拥有自己的名字，通常记录了它

的长度和元素类型等信息。不过，通过嵌套定义的方式，也允许用多个 ArrayDef 来表示多维数组对象。

这些对象之间存在着复杂的包含与被包含关系。简而言之，由于它们各自对应于特定的语法规则，它们的包含关系也类似 OMG IDL 中各个语法规则的相互包含关系。比如，模块中可以直接包含接口定义、别名定义、数组定义，ModuleDef 就可以直接包含 InterfaceDef、AliasDef、ArrayDef 及 SequenceDef。

为了便于构造接口仓库，便于表达“语法描述”对象之间的包含与被包含关系，接口仓库中还引入了以下一些对象，都是一些纯虚父类：

- IRObject，接口仓库中所有对象的基类。
- IDLType，用来定义各种数据类型的语法所对应的所有“语法描述”对象的共同基类。
- TypedefDef，在定义构造数据类型时，我们经常使用 typedef，来规定这个新被定义的数据类型的名称。比如：typedef struct my_struct{……} my_struct;。这时，这个新数据类型的名称就是 my_struct。TypedefDef 就是与此相似的那些语法所对应的所有“语法描述”对象的基类。
- Contained，所有可以被包含的“语法描述”对象的基类。
- Container，所有可以包含其它对象的“语法描述”对象的基类。

这些对象的关系如图 4.2 所示：

用 OMG IDL 定义的接口信息可以原原本本地记录在接口仓库中。不过，接口仓库中并不一定需要直接记录 OMG IDL 定义。

接口仓库中如何实现这些对象，如何还原和抽取 OMG IDL 接口定义，都是不需要一般用户去操心的。

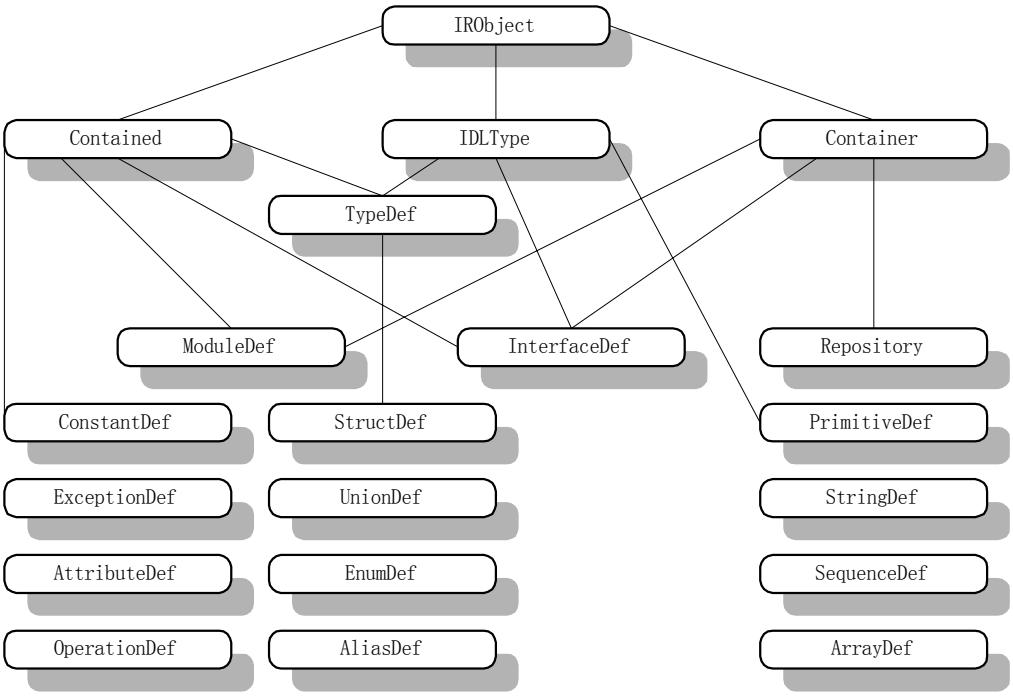

图 4.2　接口仓库中的对象关系

4.4.5 接口仓库中有关的接口定义扼要

在这里，我们将列举并注释接口仓库中一部分重要的接口定义。

例4.13 枚举类型 DefinitionKind 的定义。接口仓库中的所有对象都需要用到这个枚举类型数据。

```
module CORBA {
        typedef string Identifier;
        typedef string ScopedName;
        typedef string RepositoryId;
        enum DefinitionKind {
            dk_none, dk_all,
            dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
            dk_Module, dk_Operation, dk_Typedef,
            dk_Alias, dk_Struct, dk_Union, dk_Enum,
            dk_Primitive, dk_String, dk_Sequence, dk_Array,
            dk_Repository
        };
};
```

例4.14 IRObject 的定义，这是接口仓库中所有对象的基类。

```
interface IRObject {
        readonly attribute DefinitionKind def_kind;//说明本对象类型
        void destroy ();//本操作可以释放所有被包容的对象
};
```

例4.15 Contained 的定义，这是接口仓库中所有可以被包含的对象的基类。

```
typedef string VersionSpec;
interface Contained : IRObject {
        attribute RepositoryId id;//接口仓库标志号
        attribute Identifier name;//名称
        attribute VersionSpec version;//版本号
        readonly attribute Container defined_in;//说明被包容于哪个对象
        readonly attribute ScopedName absolute_name;//作用域名称
        //说明对象最终被包含在哪个接口仓库中
        readonly attribute Repository containing_repository;
        struct Description {//该结构用来表达本对象的信息
            DefinitionKind kind;
            any value;
        };
        Description describe ();//本操作返回描述本对象的信息
        void move (//本操作可以将对象移到另一个包容对象中
                in Container new_container,
                in Identifier new_name,
                in VersionSpec new_version
                );
        };
```

例4.16 Container 的定义，这是接口仓库中所有能够包含其它对象的对象的基类。

```
typedef sequence <Contained> ContainedSeq;//定义被包含对象无界序列
interface Container : IRObject {
```

```
Contained lookup (in ScopedName search  name);//按名查询
ContainedSeq contents (//返回所有直接被包含的对象
                in DefinitionKind limit_type, //限制要返回结果的类型
                in boolean exclude_inherited//是否排除继承关系
                );
ContainedSeq lookup_name (//按名查询
                in Identifier search_name,//名称
                in long levels_to_search,//搜索层次，-1 为不限制
                in DefinitionKind limit_type,//限制要返回结果的类型
                in boolean exclude_inherited//是否排除继承关系
                );
struct Description {//描述结果的结构，与 Contained 中的不同
    Contained contained_object;
    DefinitionKind kind;
    any value;
};
typedef sequence<Description> DescriptionSeq;
DescriptionSeq describe_contents (//返回直接被包含对象的信息
                in DefinitionKind limit_type, //限制要返回结果的类型
                in boolean exclude_inherited, //是否排除继承关系
                in long max_returned_objs//限制最大返回数目
                );
ModuleDef create_module (//产生并包含一个 ModuleDef 对象
                in RepositoryId id,//接口仓库标志号
                in Identifier name,//名称
                in VersionSpec version//版本号
                );
//产生并包含一个 ConstantDef 对象
ConstantDef create_constant (
                in RepositoryId id,//接口仓库标志号
                in Identifier name, //名称
                in VersionSpec version, //版本号
                in IDLType type,//类型
                in any value//取值
                );
StructDef create_struct (//产生并包含一个 StructDef 对象
                in RepositoryId id, //接口仓库标志号
                in Identifier name, //名称
                in VersionSpec version, //版本号
                in StructMemberSeq members//具体成员表
                );
UnionDef create_union (//产生并包含一个 UnionDef 对象
                in RepositoryId id, //接口仓库标志号
                in Identifier name, //名称
                in VersionSpec version, //版本号
                in IDLType discriminator_type,//判别数据的类型
                in UnionMemberSeq members//具体取值情况
                );
```

```
        EnumDef create  enum (//产生并包含一个 EnumDef 对象
                        in RepositoryId id,
                        in Identifier name,
                        in VersionSpec version,
                        in EnumMemberSeq members
                        );
        AliasDef create_alias (//产生并包含一个 AliasDef 对象
                        in RepositoryId id,
                        in Identifier name,
                        in VersionSpec version,
                        in IDLType original_type
                        );
        InterfaceDef create_interface (//产生并包含一个 InterfaceDef 对象
                        in RepositoryId id,
                        in Identifier name,
                        in VersionSpec version,
                        in InterfaceDefSeq base_interfaces
                        );
};
```

例4.17　IDLType 的定义，接口仓库中许多对象使用了该对象。

```
interface IDLType : IRObject {
        readonly attribute TypeCode type;
};
```

例4.18　Repository 的定义，它代表一个具体的接口仓库。

```
interface Repository : Container {
        //用接口仓库标志号查询接口信息
        Contained lookup_id (in RepositoryId search_id);
        //获得基本数据类型定义对象
        PrimitiveDef get_primitive (in PrimitiveKind kind);
        //获得字符串定义对象
        StringDef create_string (in unsigned long bound);
        SequenceDef create_sequence (//产生并包含一个 SequenceDef
                                in unsigned long bound,
                                in IDLType element_type
                                );
        ArrayDef create_array (//产生并包含一个 ArrayDef
                            in unsigned long length,
                            in IDLType element_type
                            );
};
```

例4.19　ModuleDef 的定义，这个对象也经常被使用。

```
interface ModuleDef : Container, Contained {
};
struct ModuleDescription {//定义了描述一个 ModuleDef 所需要的信息
        Identifier name;//名字
        RepositoryId id;//本身的接口仓库标志号
        RepositoryId defined_in;//被该接口仓库标志号的某个对象直接包含
```

```
        VersionSpec version;//版本号
};
```

例4.20　InterfaceDef 的定义，这个对象代表一个接口定义对象。

```
interface InterfaceDef;//前置说明
typedef sequence <InterfaceDef> InterfaceDefSeq;
typedef sequence <RepositoryId> RepositoryIdSeq;
typedef sequence <OperationDescription> OpDescriptionSeq;
typedef sequence <AttributeDescription> AttrDescriptionSeq;
interface InterfaceDef : Container, Contained, IDLType {
        attribute InterfaceDefSeq base_interfaces;
        //判断是否是指定接口仓库标志号的 InterfaceDef
        boolean is_a (in RepositoryId interface_id);
        struct FullInterfaceDescription {//用于完整描述 InterfaceDef 信息
                Identifier name;//名字
                RepositoryId id;//接口仓库标志号
                //被该接口仓库标志号的某个对象直接包含
                RepositoryId defined_in;
                VersionSpec version;//版本号
                OpDescriptionSeq operations;//所有操作的信息
                AttrDescriptionSeq attributes;//所有属性的信息
                //所有基类的接口仓库标志号
                RepositoryIdSeq base_interfaces;
                TypeCode type;//数据类型码
        };
        FullInterfaceDescription describe_interface();//返回全部信息
        AttributeDef create_attribute (//产生并包含一个 AttributeDef
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in IDLType type,
                in AttributeMode mode
                );
        OperationDef create_operation (//产生并包含一个 OperationDef
                in RepositoryId id,
                in Identifier name,
                in VersionSpec version,
                in IDLType result,
                in OperationMode mode,
                in ParDescriptionSeq params,
                in ExceptionDefSeq exceptions,
                in ContextIdSeq contexts
                );
};
struct InterfaceDescription {//接口定义对象的描述信息
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        RepositoryIdSeq base_interfaces;
```

```
};
```

例4.21　OperationDef 的定义，在自动创建参数列表时需要使用该对象。

```
enum OperationMode {OP_NORMAL, OP_ONEWAY};//定义操作执行方式
//定义参数方向属性
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
struct ParameterDescription {//用来描述参数信息
        Identifier name;//参数名称
        TypeCode type;//参数的数据类型码
        IDLType type_def;//参数的 IDLType 码
        ParameterMode mode;//参数的方向属性
};
typedef sequence <ParameterDescription> ParDescriptionSeq;//参数序列
typedef Identifier ContextIdentifier;//上下文参数标志
typedef sequence <ContextIdentifier> ContextIdSeq;//上下文参数序列
typedef sequence <ExceptionDef> ExceptionDefSeq;//异常信息序列
typedef sequence <ExceptionDescription> ExcDescriptionSeq;//异常描述
interface OperationDef : Contained {
        readonly attribute TypeCode result;//返回结果数据类型码
        attribute IDLType result_def;//返回结果 IDLType 码
        attribute ParDescriptionSeq params;//操作参数表
        attribute OperationMode mode;//操作执行方式
        attribute ContextIdSeq contexts;//上下文参数表
        attribute ExceptionDefSeq exceptions;//异常信息表
};
struct OperationDescription {//操作定义对象的描述信息
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode result;
        OperationMode mode;
        ContextIdSeq contexts;
        ParDescriptionSeq parameters;
        ExcDescriptionSeq exceptions;
};
```

接口仓库中还定义了许多其它的对象，这些定义的形式大同小异，这里不再赘述。

4.5　对象引用初始化

客户程序在通过 SII 或 DII 激发请求时，都必须指定给请求提供服务的对象实例的对象引用。这些对象引用可以从 CORBA 对象命名服务、对象洽谈服务中获得；而被激发的请求涉及的接口定义可以从接口存根或者接口仓库对象中获得。不过，在各种情况下，我们都应该首先解决一个共同问题：当客户程序启动时，如何获得与 ORB、接口仓库以及对象命名服务相关的 CORBA 对象的对象引用。通常，我们把这个问题简称为“对象引用初始化”问题。

对象引用初始化问题首先应该给客户提供对 ORB 伪对象的对象引用。

4.5.1 ORB 伪对象的初始化

CORBA 程序启动时，都需要调用 ORB 初始化函数获得 ORB 伪对象的对象引用，否则，各种操作都无法进行。

ORB 初始化函数采用 PIDL 在 CORBA 模块中定义，如下所示：

例4.22 ORB 初始化函数的定义。

```
//这是 ORB 初始化函数的定义
module CORBA{
        ……
        typedef string ORBid;
        typedef sequence<string> arg_list;

        ORB ORB_init(
                inout arg_list argv,
                in ORBid orb_identifier
                );
        ……
};
```

ORB 初始化函数的参数 orb_identifier 是 ORBid 类型数据，argv 是 arg_list 类型数据。ORBid 实际上是标志 ORB 名称的字符串，OMG 没有限制如何命名 ORB。arg_list 实际上是字符串序列，可以用来传递与特定环境有关的参数，也可以用来传递 ORBid。

与文件名称类似，ORB 名称可以随意选择。不过，当 ORB 系统服务的范围较为广阔时，最好协调所有相关 ORB 的名称，以避免不必要的命名冲突。命名字符串可以通过 orb_identifier 直接传递给初始化函数。

如果在调用初始化函数时，orb_identifier 被设置为 NULL，用户则可以在字符串序列 argv 中传递 ORB 名称。方法是在字符串序列 argv 中加入一个规定格式的“值-名对”，形式为：-ORBid “对应名称”，比如：-ORBid “VisiBroker”。

用户还可以在字符串序列 argv 中传递与 ORB 执行有关的其它各种数据，比如主机名称、服务端口号等。argv 中的参数是一些符合规定格式的“值-名对”，每个“值-名对”的形式为：-ORB<后缀><参数值>。实际上，通过 argv 传递 ORB 名称就是其中的一个特例。

ORB 初始化函数可以被用户多次调用。如果调用参数相同，应该返回相同 ORB 伪对象的对象引用。这样，用户可以在不同线程（进程）中访问相同的 ORB，也可以使 ORB 成为众多 CORBA 对象共享的资源。

由于在调用 ORB 初始化函数时，CORBA 系统中不存在任何对象引用，因此，该函数的映射也与传统的映射方式不太一致。

在 C 语言中，ORB 初始化函数的映射如下所示：

例4.23 ORB 初始化函数在 C 语言中的映射。

```
//这是 ORB 初始化函数在 C 语言中的映射
typedef CORBA_string CORBA_ORBid;

extern CORBA_ORB CORBA_ORB_init(
                                        CORBA_Environment *env,
                                        int * argc,
                                        char ** argv,
                                        CORBA_ORBid orb_identifier
                                        );
```

应该注意，环境参数实际上也是一个伪对象，所以它也可以在初始化时使用。

在 C++中，ORB 初始化函数的映射如下所示：

例4.24　ORB 初始化函数在 C++中的映射。

```
//这是 ORB 初始化函数在 C++中的映射
CORBA{//在 CORBA 名称作用域内
    typedef string ORBid;

ORB _ptr ORB_init(
                    int & argc,
                    char ** argv,
                    ORBid orb_identifier
                    );
        ……
};
```

由于该映射函数直接在 CORBA 命名作用域内定义，所以，无需在各种定义前加上 CORBA::来限定。

4.5.2　关键对象的对象引用初始化

获得 ORB 伪对象的引用后，我们仍然没有给请求涉及的操作指定对象引用。这时，用户可以进一步通过 CORBA 命名服务、对象洽谈服务获得这些对象引用。

不难想象，CORBA 命名服务、对象洽谈服务的“业务量”将随着 CORBA 分布式对象的增多而激增。在初始化时，用户即使是为了获得一个简单的对象引用，可能也需要等待 CORBA 命名服务、对象洽谈服务检索许多信息。

为了解决这个问题，CORBA 在 ORB 伪对象中提供了两个函数：list_initial_services、resolve_initial_references，用来辅助用户获得少量关键的对象引用。

通过这两个函数，实际上在 ORB 中构建了一个小规模、专门用途的“内部微型 CORBA 命名服务”。用户首先可以通过它获得一些重要的对象引用，然后在这些对象引用的基础上继续寻找更具一般性的对象引用。

由于 CORBA 命名服务、接口仓库都是通过一系列对象实现的，因此，关于“CORBA 命名服务对象”、“接口仓库对象”的对象引用就是一些关键的对象引用，需要（可以）通过这两个函数从 ORB 中直接获取。

另一方面，关键对象引用的数目毕竟有限，ORB 无需在启动之初就维护大量的对象引用。这就较好地解决了初始化效率与服务对象广泛性之间的矛盾。

在伪对象 ORB 中，这两个函数的定义如下所示：

例4.25　在初始化时获取关键对象引用的函数定义。

```
//这个例子说明在初始化时获取关键对象引用的函数定义
interface ORB{
        ……
        typedef string ObjectId;
        typedef sequence<ObjectId> ObjectIdList;

        exception InvalidName;

        ObjectIdList list_initial_services();
        Object resolve_initial_reference(
                                    in OnbjectId identifier
                                    ) raises (InvalidName);
        ……
};
```

首先，用户通过 list_initial_services 获得 ORB 内建服务的列表，接着，可以从列表中选择一项服务的名称，将其作为参数来调用 resolve_initial_reference，获得对应的关键对象引用。目前，从这两个函数中能够获得的关键对象引用仅仅有两个，一个是 InterfaceRepository，对应于接口仓库对象的对象引用；一个是 NameService，对应于 CORBA 命名服务对象的对象引用。

这两个函数的 C 语言映射如下所示：

例4.26　对象引用初始化函数的 C 语言映射。

```
//这是对象引用初始化函数的 C 语言映射
typedef CORBA_string CORBA_ORB_ObjectId;
typedef CORBA_sequence_CORBA_ORB_ObjectId
                                        CORBA_ORB_ObjectIdList;

typedef struct CORBA_ORB_InvalidName
                                        CORBA_ORB_InvalidName;

extern CORBA_ORB_ObjectIdList
                        CORBA_ORB_list_initial_services(
                                        CORBA_ORB orb,
                                        CORBA_Environment *ev
                                        );
extern CORBA_Object
                        CORBA_ORB_resolve_initial_reference(
                                        CORBA_ORB orb,
                                        CORBA_Environment *ev,
                                        CORBA_ORB_ObjectId identifier
                                        );
        ……
```

可以发现，在 C 语言映射中，这两个函数本身的对象引用都是 ORB 伪对象。

这两个函数的 C++语言映射如下所示：

例4.27　对象引用初始化函数的 C++映射。

```
//这是对象引用初始化函数的 C++映射
class ORB{
        public:
        typedef string ObjectId;

        class InvalidName{……};

        ObjectIdList_ptr list_intial_services();
        Object_ptr resolve_initial_references(
                                        ObjectId identifier
                                        );
    ……
};
```

由于这些类及函数本身就定义在 CORBA 模块中，各个语法定义前不需要加域标志符 CORBA。

4.5.3　对象引用与字符串之间的转换

有时候，用户可能需要在应用中多次使用相同的一个对象引用，但是，由于某种原因，

并不能在一个连续的时间段内进行。这时，用户可能需要首先存储这个对象引用的有关信息，以便随时载入这些信息获得同一个对象引用。为此，在 CORBA 的伪对象 ORB 中，引入了两个函数：object_to_string、string_to_object，用来将对象引用转换为字符串，并将这些字符串还原为对象引用，其定义如下所示：

例4.28　保存及读取字符串格式的对象引用的函数。

```
//这个例子说明保存及读取字符串格式的对象引用的函数
module CORBA{
        ……
        interface ORB{
        ……
        string object_to_string(in Object obj);
        Object string_to_object(in string str);
        ……
        };
        ……
};
```

当然，用户可以始终从 CORBA 命名服务及对象洽谈服务中获取对象引用，也可以进一步编写自己的函数，用来保存、读取对象引用。

4.6　本章小结

CORBA 用户有两种方式向 ORB 发出请求调用对象实现提供的服务：通过接口存根（SII）的静态方式以及通过 DII 的动态方式。后一种方式需要借助动态激发接口、接口仓库和 ORB 接口。这三部分也是 CORBA 客户程序对 ORB 构成的基本要求。

动态激发接口中包括请求管理例程、列表管理例程和多种异步通信例程。可以用来自动建立参数列表、逐步建立参数列表或手动建立参数列表，然后以同步、异步（延迟同步）或单向的方式激发动态构建的请求。

接口仓库可以被认为是 CORBA 分布式对象的“信息注册机制”以及“类型库”。是对象接口信息存储、管理、发布和使用的集中“区域”。不过，这种集中也仅仅是建立在逻辑集中的概念上。实际应用中，并不要求接口仓库驻留在同一个站点，对应于一个 ORB。

最后，ORB 还为 CORBA 客户获得初始化的对象引用提供了必要的“设施”，包括 ORB 初始化函数，CORBA 服务对象、接口仓库对象的对象引用初始化函数以及对象引用与字符串之间进行转换的函数。

第 5 章　通过 ORB 调度对象实现

本章内容提要：

- ORB *对象实现端透视*
- *基本对象适配器* Basic Object Adapter
- *共享服务器策略* Shared server policy
- *持续服务器策略* Persistent server policy
- *非共享服务器策略* Unshared server policy
- *单个方法服务器策略* Server-per-method policy
- *库对象适配器* Library Object Adapter
- *面向对象数据库适配器* Object-Oriented Database Adapter
- BOA *的未来替身* POA
- *实现仓库* Implementation Repository
- *接口与对象实现的三种对应关系*
- *实现扩散* Implementation spreading
- *动态框架接口* Dynamic Skeleton Interface

5.1　ORB 对象实现端透视

对于 CORBA 客户而言，CORBA 分布式对象就是接口存根（IDL Stub）、动态激发接口 DII、接口仓库以及 ORB 接口。但是，对于 CORBA 对象实现而言，情况就复杂的多。CORBA 客户使用越方便、越透明，CORBA 对象实现需要完成的义务也越多。

从 ORB 角度观察，每个 CORBA“对象实现”的实例都是一个服务提供对象。ORB 向这些对象实例直接发出请求，并直接接收来自它们的响应。一般情况下，我们可以将“对象实现”具体实例所驻留的进程理解为 CORBA 服务器。为了能够通过 ORB 调度对象实例，必须解决以下几个问题：

- 定位对象实现的实例。对象引用只是程序在编译、运行过程中的一个标志，ORB 在进一步处理客户请求时，必须解析这个对象引用到底和哪个“对象实现”的实例相联系，并找到被引用的对象。
- 管理“对象实现”具体实例的运行状态。在任何时候，CORBA 客户都可以通过对象引用建立 CORBA 服务请求。但是，这并不意味着这些服务提供对象始终保持在运行状态：它们可以是永久运行的对象，也可以是已经存在于某个站点、某个进程中的对象实例，还可以是尚未激发的可执行文件或动态链接库（DLL），甚至可以是需要某些机制进一步解释、执行的脚本（Script）……因此，ORB 应该能够按照用户需要控制这些对象的运行状态。
- 将请求涉及的操作与对象实现中具体的方法对应起来。

为了解决这些问题，ORB 中又引入了新的对象组，用来实现相关功能。这些对象组包括对象适配器（Object Adapter）、实现仓库（Implementation Repository）、动态框架接口（Dynamic Skeleton Interface）以及前面介绍过的接口框架（IDL Skeleton）。

现在，如果我们继续考察 ORB 的构成，可以得到如下所示的“透视全图”。

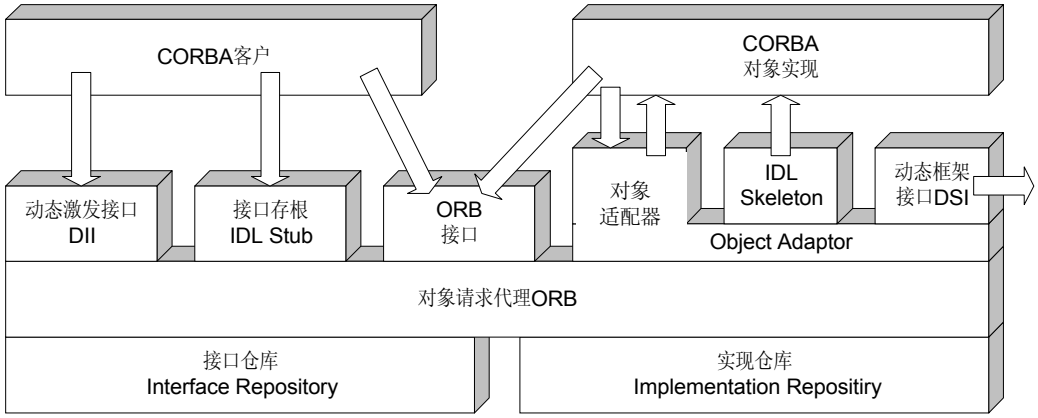

图 5.1　ORB 构成示意图

通过查询实现仓库中的信息，ORB 可以获得一个对象实例的运行状态。借助动态框架接口 DSI，ORB 可以激发远程对象实现中的方法。在软件开发过程中，程序员可以调用对象适配器中的函数，要求 ORB 以最合适的方案激活服务提供对象。而接口框架将把请求中涉及的操作与服务对象中具体的方法最终绑定。

5.2　对象适配器

对象适配器（Object Adapter）是 ORB 中管理对象引用与对象实现的核心机制。尽管 CORBA 中提供了多种类型的对象适配器，但它们都应该具备以下功能：

- 产生、解释对象引用
- 注册对象实现的实例
- 建立对象引用与具体对象实例的映射
- 激活、冻结对象实现的实例
- 通过接口框架或动态框架接口 DSI 激发对象实现中的方法
- 为 CORBA 分布式对象交互操作提供安全保障

目前，CORBA 中提供的对象适配器主要有三种，包括以下类型：

- Basic Object Adapter——基本对象适配器
- Library Object Adapter——库对象适配器
- Object-Oriented Database Adapter——面向对象数据库适配器

OMG 仅仅为基本对象适配器 BOA 定制了标准，其余类型对象适配器均由开发商自由提供。当然，这些标准也仅仅规定了基本对象适配器 BOA 必须具备的服务，并没有限制开发商如何对其进行扩展。

5.2.1　基本对象适配器 BOA

基本对象适配器 BOA 实际上也是一个 CORBA 伪对象，其中定义了一些用来管理对象引用、对象实现的操作，如下所示：

例5.1　　BOA 伪对象中定义的操作。

```
//这是 BOA 伪对象中定义的操作
module CORBA {
        interface InterfaceDef;
        interface ImplementationDef;
```

```
        interface Object;
        interface Principal;
        typedef sequence <octet, 1024> ReferenceData;
        interface BOA {
            Object create (
                        in ReferenceData id,
                        in InterfaceDef intf,
                        in ImplementationDef impl
                        );
            void dispose (in Object obj);
            ReferenceData get_id (in Object obj);
            void change_implementation (
                                    in Object obj,
                                    in ImplementationDef impl
                                    );
            Principal get_principal (
                                    in Object obj,
                                    in Environment ev
                                    );
            void set_exception (
                                in exception_type major,
                                in string userid, // exception type id
                                in void *param // pointer to associated data
                                );
            void impl_is_ready (in ImplementationDef impl);
            void deactivate_impl (in ImplementationDef impl);
            void obj_is_ready (in Object obj, in ImplementationDef impl);
            void deactivate_obj (in Object obj);
        };
};
```

BOA 中这些操作具有的功能在此简要说明如下：

- create，产生一个新的对象引用，该操作用到接口仓库以及实现仓库中的信息。
- dispose，注销一个对象引用。
- get_id，检索在 ORB 范围内唯一标志一个对象引用的标识。
- change_implementation，切换对象引用涉及的对象实现。
- get_principal，返回与对象引用及客户身份鉴别有关的各种信息。
- impl_is_ready，某些对象实现的实例需要调用本函数告知 BOA“自己”所驻留的服务器已经准备就绪，可以开始接收请求、提供服务。
- deactivate_impl，某些对象实现的实例需要调用本函数告知 BOA“自己” 所驻留的服务器将被关闭，不能继续接收请求、提供服务。
- object_is_ready，另外一些对象实现的实例需要调用本函数告知 BOA“自己”以及所驻留的服务器已经准备就绪，可以开始接收请求、提供服务。
- deactivate_obj，另外一些对象实现的实例需要调用本函数告知 BOA“自己”以及所驻留的服务器将被冻结或关闭，不能继续接收请求、提供服务。

之所以提供两套向 ORB 发送“准备就绪”以及“将被冻结或关闭”信息的函数，是为了采用多种方式激活服务提供对象的实例。在 CORBA 中，为了适应用户的各种需求，共设置了四种激活对象实例的方式：

- Shared server policy——共享服务器策略。在本策略中，一个对象实现的多个实例可以共享同一个服务器。
- Persistent server policy——持续服务器策略。与共享服务器策略类似，但是，在本策略中，服务器本身不需要被初始化，好象在持续运行一样。
- Unshared server policy——非共享服务器策略。在本策略中，需要为对象实现的每个实例创建一个服务器。
- Server-per-method policy——单个方法服务器策略。在本策略中，每调用一次对象实现中的操作，就需要为请求创建一个服务器，该请求调用结束后，服务器也被注销。

这四种激活策略的详细情况分别说明如下。

5.2.2 共享服务器策略

共享服务器策略是最常用的一种激活策略。采用这种策略，服务器将支持所有用户发出的请求，服务器中也可以包含相同对象实现的多个实例。当然，软件开发人员应该保证服务器能够同时处理多个对象。

在现实应用中，很多问题都可以采用这种方式进行。比如，服务器可以是一个视频点播服务器，其中的对象实例是为不同客户提供类似服务的“点播机对象”。

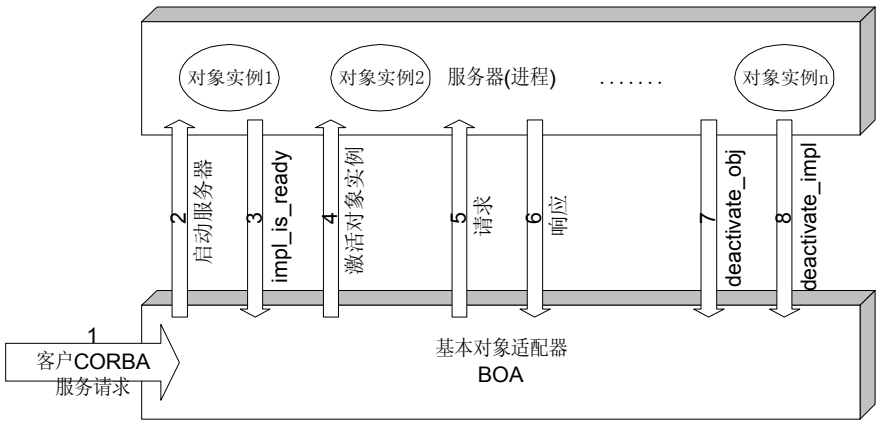

图 5.2 共享服务器策略示意图

共享服务器策略的示意图如图 5.2 所示，可以通过以下步骤实现：

- ORB 接收到客户发出的 CORBA 服务请求，通过检查实现仓库，了解对象引用涉及的服务器及对象实例是否已经激活。
- 如果服务器尚未激活，ORB 首先启动服务器。具体激活方式与系统及对象实现存在的形式有关（DLL、EXE 或者 Script）。同时，一些与 BOA 有关的信息也被传递给服务器。此时，客户的请求暂时“寄居”在 ORB 中。
- 服务器完成自身初始化。然后，可以通过 impl_is_ready 通知 BOA：服务器准备完毕，可以进一步激活对象实例。
- BOA 激活对象引用关联的对象实例。既可以通过服务器中提供的操作产生新的对象实例，也可以通过 get_id 获得先前已经被激活的对象实例。
- 根据客户提出的 CORBA 服务请求，BOA 将通过接口框架调用对象实例中的方法。这些方法调用是通过 ORB 与对象实现实例之间的请求/响应方式进行的。当然，它们之

间的请求/响应可以多次进行。

- 冻结对象实例。冻结对象实例既可以是来自客户主动发出的要求，也可以是服务器自身管理的需要。通过调用 deactivate_obj，可以通知 BOA 某个对象实例已经被冻结。这时，通常需要存储对象的状态（详见持续对象服务）。
- 如果希望最终关闭服务器，应该调用 deactive_impl，通知 BOA。

这种服务器中可以包含相同对象实现的不同实例。

5.2.3 持续服务器策略

持续服务器策略与共享服务器策略非常相似。但是，采用这种策略，服务器不需要由 BOA 启动。对于 BOA 而言，服务器应该永远处于激活状态。

在现实应用中，持续服务器并不一定需要永远保持运行状态。不过，它们不应该从 BOA 启动，而是通过手动方式或者系统方式启动。比如，数据库服务器、Web 服务器就可以采用手动方式加载，也可以在系统启动时自动加载。这种策略非常适合那些需要始终运行的服务器或者需要特殊控制的服务器。

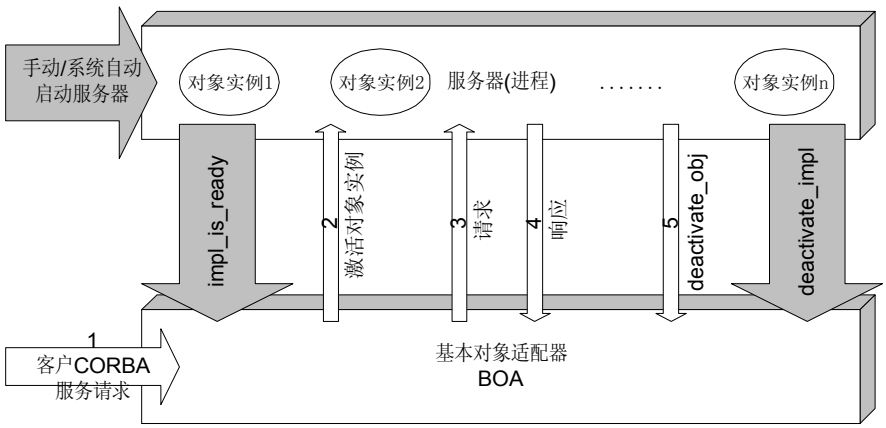

图 5.3 持续服务器策略示意图

持续服务器策略的示意图如图 5.3 所示，可以通过以下步骤激活：

- 以手动方式或者系统方式启动服务器。服务器完成初始化后，也应该通过 impl_is_ready 通知 BOA：服务器已经被激活。
- ORB 接收到客户发出的 CORBA 服务请求，通过检查实现仓库，可以得知对象引用涉及的服务器属于持续服务器类型，同时，也可以了解对象引用涉及的对象实例是否被激活。
- BOA 激活对象引用关联的对象实例。此时，既可以通过服务器中的操作产生新的对象实例，也可以通过 get_id 获得先前已经被激活的对象实例。
- 根据客户提出的 CORBA 服务请求，BOA 将通过接口框架调用对象实例中的方法。这些方法调用是通过 ORB 与对象实现实例之间的请求/响应方式进行的。当然，它们之间的请求/响应可以多次进行。
- 冻结对象实例。冻结对象实例既可以是来自客户主动发出的要求，也可以是服务器自身管理的需要。通过调用 deactivate_obj，可以通知 BOA 对象实例已经被冻结。这时，通常需要存储对象的状态（详见持续对象服务）。

- 如果服务器被最终关闭，应该调用 deactive_impl，通知 BOA。但是，关闭服务器的请求不能由 BOA 发出，必须从外界执行。

持续服务器策略除了不能由 BOA 启动、关闭服务器以外，其它特点与共享服务器策略完全一致。

5.2.4 非共享服务器策略

非共享服务器策略非常容易让人产生误会，以为这种方式中的对象实例不能被多个客户共享。实际上，应该将非共享服务器策略理解为：相同“对象实现”的不同对象实例之间不能够驻留在同一个服务器进程中。也就是说，服务器进程不能被相同“对象实现”的不同对象实例所共享。为了便于理解，用户还可以把这种策略想象为“每个对象一个服务器”。

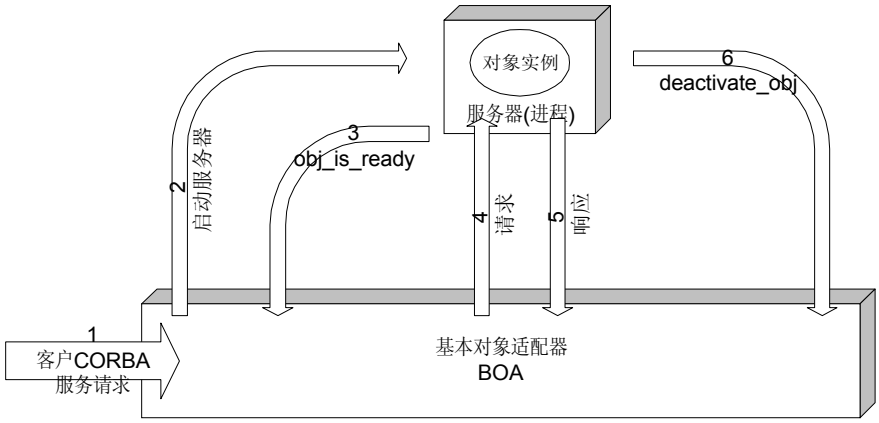

图 5.4 非共享服务器策略示意图

在现实应用中，很多问题都需要采用这种方式进行。比如，服务器可以对应于一个网络打印机，其中的对象实例就是“待打印作业对象”。对于独占类型的资源，一般都可以采用这种策略来实现它们对应的对象。

非共享服务器策略的示意图如图 5.4 所示，可以通过以下步骤实现：

- ORB 接收到客户发出的 CORBA 服务请求，通过检查实现仓库，了解对象引用涉及的对象实例是否已经激活。
- 如果对象实例尚未激活，ORB 首先启动服务器。具体激活方式与系统及对象实现存在的形式有关（DLL、EXE 或者 Script）。同时，一些与 BOA 有关的信息也被传递给服务器。此时，客户的请求暂时“寄居”在 ORB 中。
- 服务器完成自身初始化后即产生一个对象实例。然后，可以通过 obj_is_ready 通知 BOA：对象实例（此时，也就是服务器）准备完毕。
- 根据客户提出的 CORBA 服务请求，BOA 将通过接口框架调用对象实例中的方法。这些方法调用是通过 ORB 与对象实现实例之间的请求/响应方式进行的。当然，它们之间的请求/响应可以多次进行。
- 冻结对象实例。此时，冻结对象实例就是关闭该对象实例驻留的服务器。这既可以是来自客户主动发出的要求，也可以是服务器自身管理的需要。通过调用 deactivate_obj，可以通知 BOA 对象实例已经被冻结，对应的服务器也被关闭。这时，通常需要存储对象的状态（详见持续对象服务）。

- 如果希望获得相同对象实现不同的对象实例，必须重新建立相应的服务器。

☎ **技巧** 采用非共享服务器策略时，CORBA 客户依然可以通过使用相同的对象引用来共享这个对象实例，或者说共享这个对象实例驻留的服务器。因此，我们认为非共享服务器策略这个“称谓”非常容易误导 CORBA 客户。

5.2.5 单个方法服务器策略

单个方法服务器策略是一种很少使用的激活策略。采用这种策略，每次请求都需要启动一个新的服务器，服务器中包含某个对象实现的一个实例。不过，请求一旦结束，对象实例就会被注销，服务器也会被关闭。

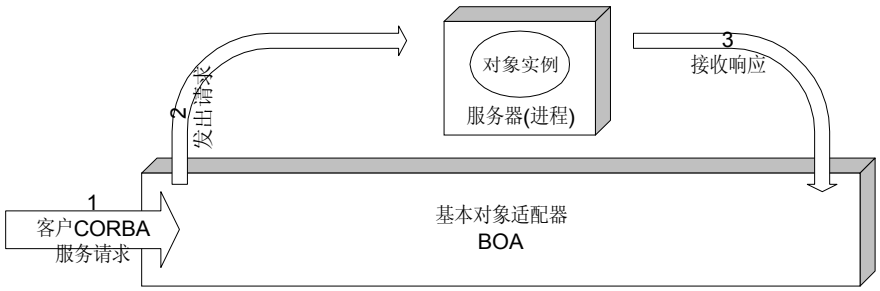

图 5.5 单个方法服务器策略示意图

这种策略非常适合于脚本（Script）语言类型的对象实现，每调用一次对象实现中的方法，就启动脚本解释机制，构成相应的对象实现服务器，完成方法调用后立刻释放有关的资源。有时，一些仅仅需要激发一次的请求也可以采用这种策略。比如，批处理文件、编译一个文件或执行一个遗留程序等等情况。

单个方法服务器策略的示意图如图 5.5 所示，可以通过以下步骤实现：

- ORB 接收到客户发出的 CORBA 服务请求，根据该请求，BOA 直接通过接口框架调用对象实例中的方法。这个方法调用将自动启动一个新的服务器，建立一个对象实现的实例。
- BOA 接收响应。本方法调用一旦结束，对象实例自动被注销，它驻留的服务器也自行关闭。

即便用户希望再次调用同一个对象实现的相同的方法，也需要重新建立服务器。

5.2.6 库对象适配器

如果对象实例与那些发出请求的客户程序驻留在相同进程中，就非常适合采用库对象适配器（Library Object Adapter）。

库对象实例不需要激活、授权就可以被用户直接使用。库对象实例的运行代价非常小，效率也相对较高。不过，如何操作库对象实例并没有被标准化，完全依赖于 CORBA 供应商。

5.2.7 面向对象数据库适配器

面向对象数据库适配器（Object-Oriented Database Adapter）为用户使用面向对象数据库存储对象状态提供各种必要的连接。

由于面向对象数据库中通常会提供存储、操作各种持续对象的函数，所以，对象激活时不需要“显性”注册，状态也无需专门处理。同样，面向对象数据库适配器也没有被标准化，完全依赖于 CORBA 供应商。

注意 在 COM/DCOM 中，对象（组件）的产生、激活等工作主要由类工厂 Iclas-

sFactory 及其派生接口对应类中相应的方法来完成。因此，在某种程度上，可以把 CORBA 中的对象适配器与 COM/DCOM 中的类工厂 IClassFactory 对照起来理解。在 CORBA 中，采用的策略是：借助依附于 ORB 的对象适配器，完成产生对象实例、激活对象实例的任务。而在 COM/DCOM 中，用户必须为每个 COM/DCOM 对象对应的类工厂编码，实现其中规定的方法来产生对象实例。可以认为，在 COM/DCOM 中，“对象适配器” 实际上被分配到每个分布式对象内部去实现。此时，我们可以更加清楚的认识到，CORBA 将许多分布式软件必需的功能集中于中间件中实现；而 COM/DCOM 仅仅是一些拥有多种共同接口的复合对象。当然，如果用户采用 Delphi 或者 C++ Builder 来开发 COM/DCOM 程序，类工厂中的方法可以由编译器自动生成。

5.2.8 BOA 未来的替身 POA

在 1998 年 2 月颁布的 CORBA2.2 规范中，BOA 已经被 POA 取代。POA 是 Portable Object Adaptor 的缩写。顾名思义，POA 希望提供一种新型的对象适配器，使得对象实现代码可以在不改动或者很少改动的情况下就可以在不同厂家的 ORB 系统上使用。

POA 希望至少达到以下一些目标：

- 允许用户开发可以在不同的厂商 ORB 系统中移植的对象实现
- 为对象提供永久标志，更确切的说，POA 允许一个对象实例的寿命超过服务器（进程）的寿命。
- 允许以最简便的方式创立瞬态对象实例，也就是寿命不会超过服务器（进程）寿命的一般对象实例。
- 允许透明的激活一个对象实例
- 允许对象实例最大限度的控制自己的行为，如规定自己的标志号、确定不同标志号所对应的状态、控制自己的“上班时间”。
- 允许提供一种可扩展机制，记录与使用对象实例策略有关的信息

由于 POA 的商业产品还很不普及，我们对此不再详细介绍。

5.3 实现仓库

实现仓库（Implementation Repository）存储供 ORB（主要是 BOA）定位、激活对象实现所需要的各种信息。CORBA 使用这些信息了解分布式对象的实际运行状态，并进一步建立对象引用与对象实现实例之间的关联。

实现仓库中的信息一般存放在伪对象 ImplementationDef 中，可以包括以下内容：

- 服务器注册、活动状态、执行及控制信息。主要用来说明对象实现驻留的服务器是否被初始化、是否被注销、处于哪种运行状态。
- 对象实例活动状态、执行及控制信息。主要用来说明对象实现的实例是否被激活、是否被冻结、处于何种状态。
- 各种附加信息，如调试信息、管理信息、资源分配信息、安全信息。

实现仓库采取哪种结构、如何访问、提供哪些服务，都取决于 CORBA 供应商。在 CORBA 中，用户一般很少直接与实现仓库交互信息。CORBA 规范中也并未对此作出具体规定。

5.4 接口框架

接口框架（IDL Skeleton）为把对象接口中定义的操作与对象实现中的具体方法连接起来提供必要的信息。

接口定义本身仅仅是编程应该满足的一种规范，并没有如何实现这些操作的具体编码。因此，每个接口中的操作至少必须与一个对象实现中的方法相连才能够使用。而接口框架可以看作是“对象实现”编码中编写方法的“花名册”，通过其中的信息，ORB 可以找到相关的方法代码，激发或使用对应的方法。

接口框架在 ORB 中主要完成以下功能：

- 查找请求涉及的操作在对象实现中对应的具体方法
- 对请求涉及的操作中的参数进行编组（Marshal），把它们从客户端传输过来的格式转化为编写对象实现的语言所支持的形式。
- 对响应中的参数以及请求中 inout、out 类型的参数进行编组，把它们从对象实现端传输过来的格式转化为编写客户端程序的语言所支持的形式。

注意 在 COM/DCOM 中，IDispatch 接口对应的实现提供类似 CORBA 中接口框架的功能，可以找到客户希望激发的方法并完成调用。另外，一般情况下，Windows 会自动为需要来回传递的数据进行编组，除非用户通过 IMarshal 接口对应的实现提供明确的支持。

通过 IDL 编译器，我们可以从 OMG IDL 定义的接口中获得接口框架。该接口框架将接口中的每个操作都和某个对象实现中的特定方法建立了连接。实际上，可以认为：接口框架中规定了软件开发人员必须实现的一些函数。这是 CORBA 为保证所有接口操作都可以被激发而强迫给软件开发人员的编程义务。但是，CORBA 并没有规定，这个接口中的操作只能由上述对象实现中的方法完成。

也就是说，虽然对象实现与接口框架之间存在一一对应的关系，但是接口与接口框架之间并不存在一一对应的关系。因此，一个接口既可以对应于一个对象实现，也可以对应于多个对象实现中的一个，甚至可以分别对应于不同的对象实现。

在 BOA 中，为实现上述三种对应方式提供了必要的函数：创建对象引用时，可以在 create 中指定用户希望的对象实现，创建对象引用后，可以通过 change_implementation 切换用户希望的对象实现。有关函数的详细定义可在前面的相关章节中找到。

上述三种对应情况分别说明如下。

5.4.1 一个接口对应一个对象实现

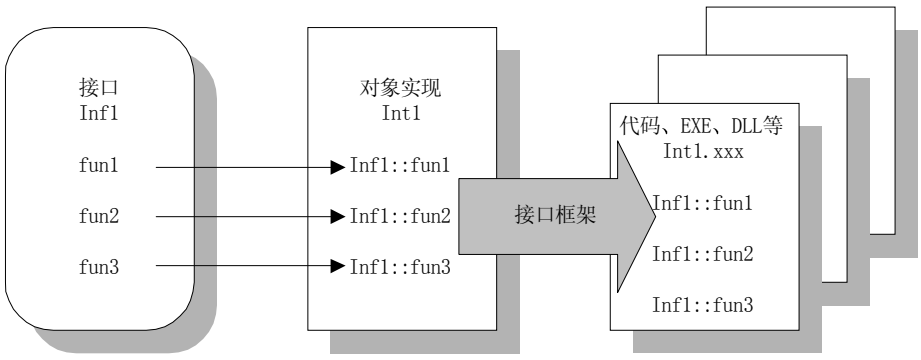

图 5.6 一个接口对应一个对象实现

一个接口对应一个对象实现是最简单的一种情况。这时，一个接口中的所有操作全部都能在指定的（默认的）对象实现中找到对应的方法，如图 5.6 所示。

在这种情况下，接口 Inf1 中的操作 fun1、fun2、fun3 分别由对象实现 Inf1 中的操作 fun1、fun2、fun3 实现；而 ORB 可以通过对象实现 Inf1 的接口框架调用软件开发人员编写的方法 fun1、fun2、fun3。这些方法可以以可执行文件、DLL、Script 等多种形式存在。

这种方法虽然简单，但缺乏灵活性。接口与对象实现之间的联系显得过分紧密。

5.4.2 一个接口对应多个对象实现中的一个

一个接口对应多个对象实现中的一个是较灵活的一种情况。这时，一个接口中的所有操作全部都能在侯选对象实现中的某一个对象实现中找到对应的方法，而且，侯选的对象实现都可以单独与接口建立对应关系。如图 5.7 所示。

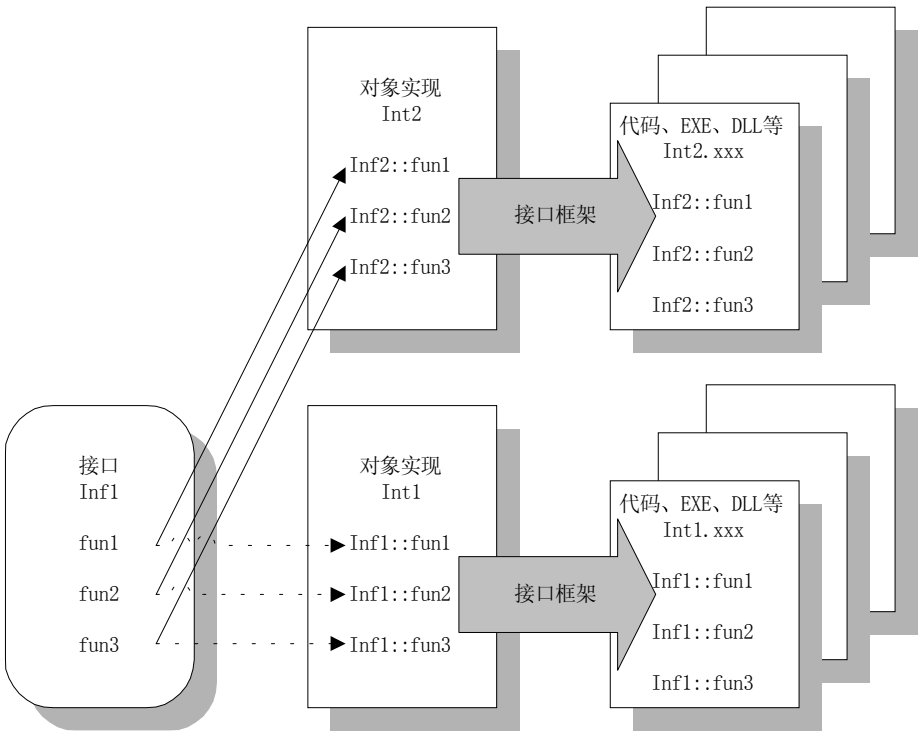

图 5.7 一个接口对应多个对象实现中的一个

在这种情况下，接口 Inf1 中的操作 fun1、fun2、fun3 可以由对象实现 Inf1 中的操作 fun1、fun2、fun3 实现；也可以由对象实现 Inf2 中的操作 fun1、fun2、fun3 实现。当然，在这种情况下，用户应该首先通过一定的机制来选择自己需要的对象实现，比如命名服务、对象洽谈服务以及上下文对象等等。

而 ORB 既可以通过对象实现 Inf1 的接口框架调用软件开发人员在模块 Inf1.*中编写的方法 fun1、fun2、fun3；也可以通过对象实现 Inf2 的接口框架调用软件开发人员在模块 Inf2.*中编写的方法 fun1、fun2、fun3。到底调用哪些函数最终取决于接口与哪个对象实现建立了对应关系。

在实际应用中，供选择的对象实现可以是驻留在不同站点的对象，比如，本地对象实现及远程对象实现，可以是针对不同任务而设计的、具有不同执行过程的对象，比如，轻载荷对象实现及重载荷对象实现，可以是相同对象实现的不同版本……

这种方式使得接口也具备了“多态性”。

5.4.3 一个接口对应多个对象实现

一个接口对应多个对象实现是最为灵活的一种情况。这时，一个接口中的操作可以分别对应于多个对象实现中的方法，如图 5.8 所示。

在这种情况下，接口 Inf1 中的操作 fun1、fun2 可以由对象实现 Inf1 中的操作 fun1、fun2 实现；而操作 fun3 可以由对象实现 Inf2 中的操作 fun3 实现。同样，接口 Inf3 中的操作 fun1、fun2 也可以通过激发对象实现 Inf2 中的操作 fun1、fun2 来实现，而且，接口 Inf3 对操作 fun3 完全未知。

这种方式被称为实现扩散（Implementation spreading）。实现扩散允许客户随心所欲的通过接口激发任意对象实现中符合定义的操作，非常易于代码重用。不过，实现扩散强调的是接口的中操作必须被某个对象实现中的方法支持，并不强调接口与对象实现之间存在何种关系。

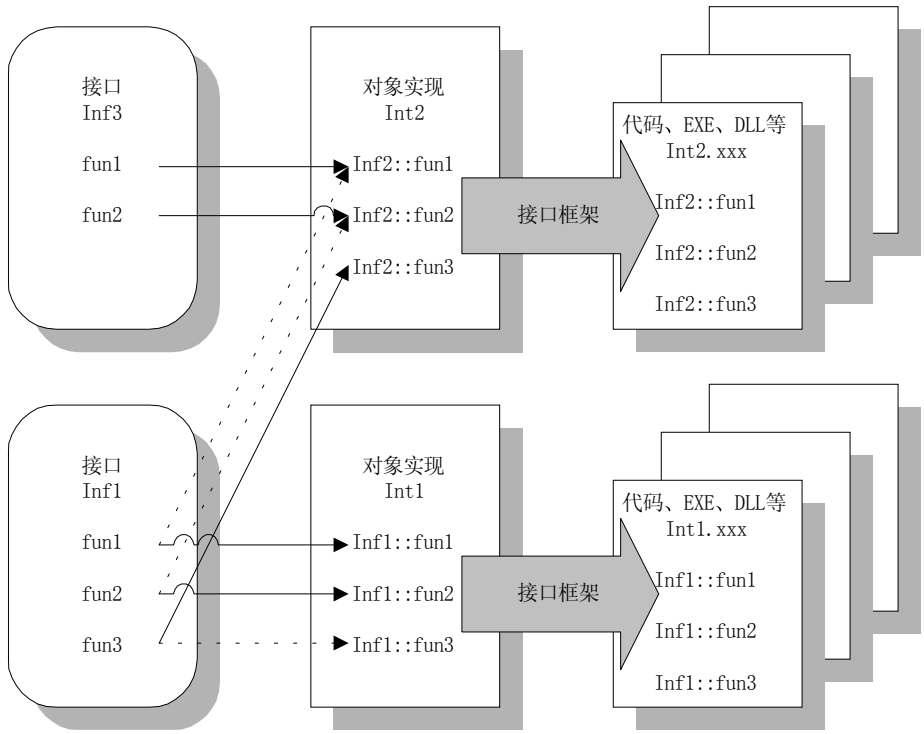

图 5.8　一个接口对应多个对象实现

这种方式极大丰富了接口所具备的“多态性”。

5.5　动态框架接口 DSI

显然，CORBA 分布式对象不能在每个 ORB 中都拥有与自己相应的接口框架，否则，每个 ORB 都有可能消耗趋于无限大的存储资源。如果客户向 ORB 发出的请求不能在该 ORB 中找到需要的接口框架，就必须通过其它机制获得需要被激发的操作的信息。在 CORBA 中，该机制被称为动态框架接口（Dynamic Skeleton Interface），由一系列伪对象及动态实现例程 DIR（Dynamic Implementation Routine）构成。

动态框架接口也是 CORBA 实现互操作的核心机制。它能够沟通不同的 ORB，使得 ORB 之间可以通过远程调度互相激发任意分布的对象。关于 CORBA 互操作的内容，我

们在下一章中详细说明。

动态框架接口的核心思想就是，通过使用 DIR，有关 ORB 可以发出一个标准的 DSI 请求，该请求的目标实际上是一个代理对象（但是对于该 ORB 而言，与使用接口框架一样，无需作出任何调整），收到 DSI 请求的 ORB 如果可以真正激发其中涉及的操作，就完成相应的激发并把结果通过 DIR 返回，否则仅作为请求级别的“网桥”传递 DSI 请求。

为了使 CORBA 客户能够使用动态框架接口，必须编码实现一个或者多个具有 DynamicImplementation 的接口对象，并调用相关的函数通知对象适配器采用了动态框架。DynamicImplementation 伪对象以及 BOA 中登记动态框架的操作定义如下所示：

例5.2　DynamicImplementation 伪对象及有关 BOA 登记函数。

```
//这是 DynamicImplementation 伪对象及有关 BOA 登记函数的定义
module CORBA{
        ……
        interface ServerRequest;

        interface DynamicImplementation{
            void invoke(inout ServerRequest request);
        };

        interface BOA{
            ……
            void setImpl(
                    in Implementation implDef,
                    in DynamicImplementation impl
                    );
            ……
        };
        ……
};
```

对象实现可以调用 BOA 中的函数 setImpl 通知对象适配器，与 implDef 有关的对象实现应该通过动态框架接口与对象实现 impl 对应。因此，凡是与 implDef 对象实例有关的请求，都应该使用动态框架接口去激发。当然，调用这个函数并不影响使用 implDef 对象实现的 CORBA 客户，对他们而言，无论使用接口框架还是使用动态框架接口，能够完成的功能是一致的（也许时间上有些差异）。

随后，当 BOA 激发与对象实现 implDef 有关的请求（既可以是静态激发也可以是动态激发）时，并不通过本 ORB 的接口框架去调用对应的方法，而是调用 impl 中的函数 invoke 去其它远程 ORB 中执行。

invoke 中的参数属于 ServerRequest 类型，是一个类似 DII 方式中请求伪对象 Request 的伪对象，完整记录了客户请求的全部内容。ServerRequest 伪对象的接口定义如下：

例5.3　ServerRequest 伪对象的接口定义。

```
//这是 ServerRequest 伪对象的接口定义
module CORBA{
        ……
        interface ServerRequest{
            Identifier op_name();
            Context ctx();
            void params(inout NVList parms);
            any result();
        };
```

```
        ……
};
```

在 ServerRequest 中，op_name 返回实际上被调用的对象实例中操作的名称，ctx 返回可能采用的上下文对象，params 来回传递调用函数所必须的参数列表，result 返回请求完成后的最终结果。

所有对象适配器都可以支持动态框架接口 DSI，但是，OMG 仅仅规定了基本对象适配器 BOA 中的格式。其它对象适配器如何使用 DSI 完全取决于 CORBA 供应商。

实际上，客户也不会直接使用动态框架接口 DSI。

5.6 对象实现编程扼要

对于 CORBA 对象实现而言，ORB 意味着对象适配器（主要是 BOA）、实现仓库、接口框架 IDL Skeleton、动态接口框架 DSI 以及一些相关的 ORB 接口。软件人员在编写对象实现的代码时，可能需要和 ORB 的以上各部分进行必要的交互。下面，将对象实现编程中一些具有共性的问题扼要说明如下。

5.6.1 编写对象实现的一般步骤

软件开发人员在编写对象实现代码时，通常应该考虑解决以下问题：

- 决定对象实现所采取的激活策略。在前面章节中，我们已经详细说明过四种激活策略及它们适用的场合。软件开发人员应该首先决定激活对象实现实例的策略，并按照 CORBA 供应商提供的途径将有关决定“告知”ORB。
- 获得接口框架。接口框架可以通过编译 OMG IDL 接口定义文件获得，也可以从接口仓库中获得。这些接口框架中通常有一些需要软件开发人员必须实现的函数定义、必须派生并实现的纯虚父类定义。
- 初始化对象实例所驻留的服务器。初始化工作根据用户采用的 ORB 以及激活策略而异。但通常都包括启动服务器、通知 ORB 服务器已经成功启动两部分内容。
- 激活对象实现的实例。既可以要求服务器启动后主动创建对象实现的实例，也可以要求服务器在接收到有关命令或者在需要的时候创建对象实现的实例。同样，对象实现应该负责通知 ORB 对象实例已经被激活。ORB 在没有接收该通知之前，将阻止任何与该对象实例有关的请求。
- 编写消息接收循环以及事件响应代码。在大多数情况下，对象实例所驻留的服务器进程就是一个消息接收、解析、分发机制。因此，大多数对象实现的编码中都有消息分发循环，而且，消息分发循环有可能就是对象实现程序代码的主体框架。
- 编写方法代码。方法代码是各种任务得以完成的基本点。编写方法代码时，既可以向其它 CORBA 对象发出请求，也可以使用 CORBA 基本服务、CORBA 工具集，还可以调用适当的 ORB 函数。当然，在 CORBA 分布式对象中，仍然可以使用其它的库函数、程序模块或者类；另外，一些 ORB 系统或者 CORBA 软件开发工具也允许用户同时使用 COM/DCOM 技术。
- 冻结对象实现的实例。可以根据客户的请求或者服务器管理的需要，在合适的时刻冻结对象实例。软件开发人员需要编写代码通知 BOA，还可能需要编写代码存储对象的状态、释放资源。
- 关闭服务器。关闭服务器代码与启动服务器代码互相对应，需要通知 ORB，并且可能

需要保存一些信息，释放许多资源。

5.6.2 服务器初始化

为了启动服务器、初始化服务器，通常需要完成以下任务：

- 编写服务器全局变量、系统数据的初始化代码。这些任务与 CORBA 没有任何关系。
- 编写与 CORBA 相关的对象初始化代码。比如，用户需要调用 ORB_init 来获得 ORB 伪对象的引用，还需要调用 BOA_init 来获得基本对象适配器 BOA 伪对象的引用。BOA_init 是伪对象 ORB 中的一个函数，其定义与 ORB_init 非常相似，使用方法也非常类似。
- 编写由对象适配器（BOA）启动服务器的代码。除了持续服务器策略以外，服务器通常都由 BOA 启动。因此，必须提供由 BOA 启动服务器的接口，可以是数据文件、注册信息等方式。最终取决于 CORBA 供应商。
- 编写代码通知 BOA 服务器已经激活。对于共享服务器策略、持续服务器策略，可以调用 impl_is_ready；对于非共享服务器策略，可以调用 obj_is_ready，不过在此之前，应该激活对象实例。

5.6.3 终止服务器

显然，终止服务器与启动服务器时进行的操作相反，主要包括以下任务：

- 编写代码通知 BOA。如果采取的是共享服务器策略或者持续服务器策略，可以通过调用 deactivate_impl 进行；如果采取的是非共享服务器策略，可以通过调用 deactivate_obj 进行。
- 编写由对象适配器（BOA）终止服务器的代码。因为终止服务器的请求通常由对象适配器提出，所以 BOA 应该知道如何去终止一个服务器。
- 编写代码注销服务器涉及的 CORBA 对象引用，释放服务器占用的与 CORBA 有关的资源。这一部分操作通常也需要对象适配器的参与。
- 编写代码注销服务器涉及的与 CORBA 无关的对象实例，关闭、释放服务器占用的与 CORBA 无关的资源。

编写 CORBA 软件的方式最终取决于用户使用的 ORB 系统及所使用的开发工具。详细情况可以参看第九章。

5.7 本章小结

CORBA 用户通过 ORB 调度对象实现的实例时，必须确定对象引用所针对的实例、获取并控制对象实例的运行状态、决定接口操作能够激发的对象实现中的方法。为了解决这些问题，ORB 中引入了对象适配器、实现仓库、接口框架、动态框架接口以及一些新的 ORB 接口函数。

实现仓库是记录对象实例运行状态以及它所驻留的服务器进程的运行状态的 ORB 内部数据库。接口框架是将接口操作映射到具体对象实现方法的“花名册”。对象适配器通过实现仓库、接口框架中的信息将对象引用、客户请求、对象实现联系起来，并调用合适的方法完成 CORBA 客户转交给 ORB 的请求。

CORBA 中共有三种类型的对象适配器：基本对象适配器、库对象适配器以及面向对象数据库适配器。其中，OMG 仅仅对基本对象适配器 BOA 作出了某些规定。

基本对象适配器允许客户以四种方式激活对象实现对应的实例：共享服务器策略、持

续服务器策略、非共享服务器策略以及单个方法服务器策略。四种策略可以用来满足不同的实际需要，解决不同的实际问题。

对象适配器也允许客户在接口与对象实现之间建立多种映射关系，包括一个接口对应一个对象实现、一个接口对应多个对象实现中的一个、一个接口对应多个对象实现。这些灵活的对应方式给接口也带来了“多态性”。

由于任何一个 ORB 都不可能拥有所有分布式对象的接口框架，对象适配器可以通过动态框架接口 DSI，在 ORB 之间传递请求，并最终调用合适的对象方法。

开发 CORBA 对象实现时，编码的难易程度取决于用户采用的 ORB 及开发工具。

第 6 章　CORBA 互操作

本章内容提要：

- ORB-ORB *之间的通信*
- CORBA *域(*Domain*)*
- *直接桥接* Immediate Bridging
- *间接桥接* Mediated Bridging
- *请求级别的桥接* Request_Level Bridging
- *内联桥接* in-line Bridging
- *互操作对象引用* Interoperable Object Reference
- *通用* ORB *互操作协议* GIOP
- *因特网* ORB *互操作协议* IIOP
- *特定环境* ORB *互操作协议* ESIOP、DCE-CIOP
- *互操作的层次结构*

6.1　CORBA 互操作

任何对象请求代理 ORB 能够记录的接口框架及接口存根信息都是有限的。如果 CORBA 客户希望随意使用集成分布在整个互联网上的 CORBA 对象，各个对象请求代理 ORB 之间必须可以进行必要的“交流”，能够相互调度。这就是所谓的 CORBA 互操作。

CORBA 互操作建立在 ORB 与 ORB 之间的通信基础之上。而 ORB 的各个构成部分，如接口存根、接口仓库、实现仓库、接口框架、对象适配器以及动态框架接口 DSI 也都为 CORBA 互操作提供了必要的支持。

在整个互联网上，CORBA 客户、ORB 与对象实现的布局可能如图 6.1 所示：

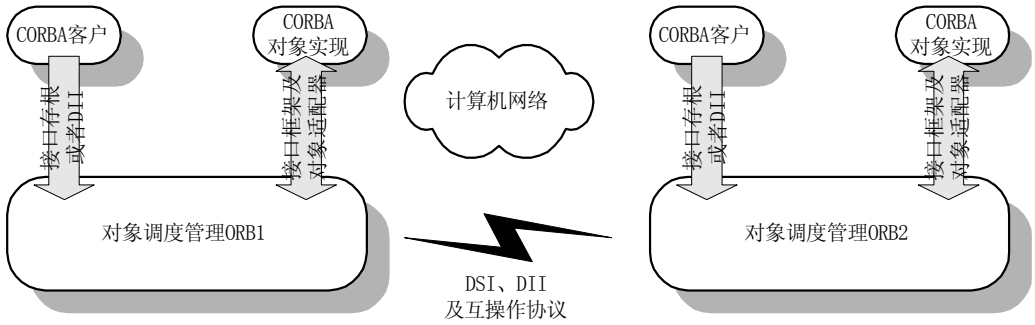

图 6.1　互联网上的 CORBA 布局

客户可以通过静态激发或者动态激发向 ORB1 发出 CORBA 服务请求。ORB1 根据请求涉及的对象引用检查自身的实现仓库及接口框架，如果在“本地”发现相关的对象实现，ORB1 立刻调度合适的方法，完成服务请求；如果相关的对象实现在远程对象请求管理 ORB2 上，ORB1 通过动态框架接口 DSI、动态激发接口 DII 以及互操作协议，将请求“路由”到对象请求代理 ORB2，ORB2 则通过“本地”的接口框架调度合适的方法，完成服务请求后再将响应“路由”到 ORB1。

对于客户而言，几乎无法知道被发出的请求是通过本地对象实现还是远程对象实现完

成的；对于对象实现而言，也无法知道请求是由本地客户发出的还是远程客户发出的、是通过静态接口还是动态接口激发的。因此，CORBA 互操作是客户得以透明使用、集成 CORBA 分布式对象的核心机制。而且，CORBA 互操作的主体仅仅是分布在网络环境中的对象请求代理 ORB，CORBA 客户以及 CORBA 对象实现并不直接参与。

显然，ORB 与 ORB 之间的通信可以采用如下所示的两种方式进行：

- 所有 ORB 都采用一种相同的通信协议，这样，各个 ORB 之间就可以直接相互通信，实现调度。
- 允许 ORB 使用不同的通信协议，然后，安装必要的“网桥”，在不同 ORB 之间翻译它们所采用的协议。

目前，CORBA 同时支持这两种方式。所有 ORB 采用相同协议的方式比较容易被我们接受，因为这种方式会给 CORBA 互操作带来管理简便、使用简洁、高效率等特点。不过，之所以允许 ORB 使用不同的协议，也存在许多理由：

- 不同用途、不同位置的 ORB 可能需要满足不同安全性指标的通信方式。显然，对不同级别的通信消息采用统一的方式传递，是一个吃力不讨好的差事。因此，我们希望提供多种通信协议，实现不同安全级别的通信。
- 在结构异质的网络中，不同区域的 ORB 更适合采用与自身网络结构相适应的通信协议。因特网是一个异质型的网络，在每个不同的区域中都有可能采用不同的网络协议。而 ORB 必须“生存”于特定的网络空间，最好还是具备入乡随俗的适应能力。
- 不同系统的 ORB 可能具有不同的硬件环境，因此可能需要采用不同的通信方式。随着网络社会的完善，一些 ORB 可能主要用来管理嵌入程序的智能仪器，比如电子开关、电子温度计、各种寻呼机等等。这些 ORB 通常处于内存有限、低带宽的硬件中，因此，这些 ORB 可能需要采用专有的、精练通信方式。
- 解决遗留的问题。在 CORBA1.x 的版本中，并没有规定 ORB 之间通信所采用的协议，不同厂商可能因此而采取了不同的通信格式。在 CORBA2.0 以上的版本中，特地增加了 GIOP 协议。当然，也允许 ORB 继续采用不同格式进行通信。

显然，ORB 与 ORB 之间的通信协议是应用层的协议。

对于 CORBA 客户以及 CORBA 对象实现而言，CORBA 互操作是一种透明的机制，在软件开发过程中并不需要为此付出额外的代价。但是，如果希望了解 CORBA 的内幕以及各种不断导致软件界技术更新的现实问题，我们不妨尝试一下 CORBA 互操作。

6.2 CORBA 的域

所谓域（Domain）就是指满足一定特征的网络区域、网络计算机集合。我们可以从不同角度去划分域。比如，可以根据网络区域、网络计算机组成的技术、网络计算机的使用人员或机构、网络计算机的管理措施以及它的安全级别去划分域…。

当然，在同一个域内的网络计算机应该符合所有定义该域的特征，同时，不同域可以交叉定义，一个网络计算机可能被归属到不同的域。

CORBA 的域可以是任何技术域、管理域或者策略域。一般情况下，任何域都存在以下某些共性：

- 每个域都有一个确实存在的边界。域的边界可以存在于地域上、技术上或者逻辑上。域的管理者通常非常注意这些边界。
- 在同一个域内，通常以相同的方式使用应用程序、数据及各种策略。因此，我们可以把应用程序、数据、策略设计成随同域变化而变化的模式。

- 如果两个域中的特征并不相同，但是，这些特征在两个域中可以成对的对应起来，我们就能够在两个域的边界构筑一定的“桥梁”，在不同域中翻译各自的特征，将这两个域“桥接”起来。
- 虽然域的边界可以以多种方式存在，但是，在“桥接”不同的域时，却有可能可以集中进行，同时处理各种类型的边界。比如，ORB 就可以是解决 CORBA 互操作问题，实现各个层次的桥接的集中“地界”。
- 如果两个不同的域可以被桥接起来，那么，同样意味着可以在桥接机制中加入条件选择，根据不同的情况或者不同的需要，在这两个域之间建立选择性桥接。

由于 CORBA 域可以是技术域、管理域或者策略域，ORB 在桥接不同域时相应可以分为技术桥接（Technological Bridging）、策略桥接（Policy-Mediated Bridging）。技术桥接主要解决不同域之间传输协议、数据格式方面的差异，为 CORBA 互操作提供基本机制，策略桥接则主要取决于 ORB 管理人员。实际上，用 ORB 作为不同域之间的策略桥接时，ORB 就是一个基于面向对象技术的防火墙。CORBA 的技术桥接是实现 CORBA 互操作的核心，策略桥接也建立在技术桥接基础之上。

CORBA 提供了两种技术桥接形式：直接桥接、间接桥接。

6.3 CORBA 桥接

桥接就是一种在不同域之间解释、转换、控制各个域中相关信息元素的机制。显然，桥接需要能够穿越它所作用的不同域的边界，因此，桥接往往驻留在作用域的公共边界上。

6.3.1 直接桥接

直接桥接（Immediate Bridging）实际上是一个仅仅能建立在两个域上的桥接方式。直接桥接将各种有关的信息直接从一个域中的格式转化为另一个域中的格式，沟通其所作用的两个不同域，如图 6.2 所示：

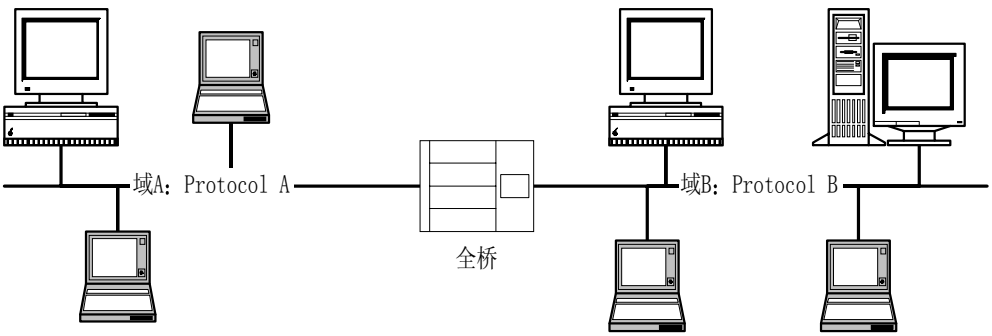

图 6.2　直接桥接

直接桥接运行效率比较高。但是，直接桥接必须针对所作用的两个域专门设计。当需要桥接的域增多时，所需的全桥迅速增长，如图 6.3 所示。

当连接四个不同的域时，共需要六个全桥；如果有 n 个需要连接的域，则共需要（n^2-n）/2 个专门设计的全桥。

直接桥接非常适合于策略桥接。

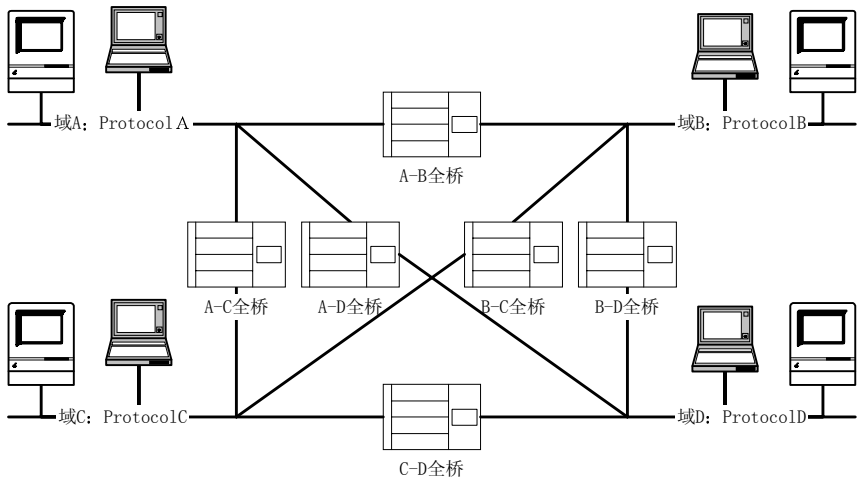

图 6.3　用来联系四个域的直接桥接

6.3.2　间接桥接

在间接桥接（Mediated Bridging）中，所有域的协议都被解释为一个公共协议。当某个域的信息需要穿越边界进入另一个域时，首先被转换为公共协议支持的格式，然后再由公共协议格式转换为目的域的格式，如图 6.4 所示：

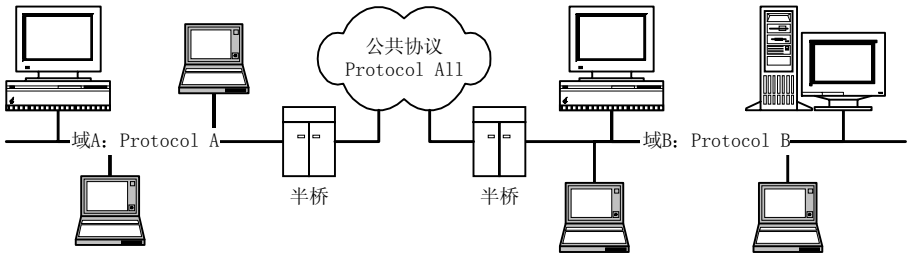

图 6.4　间接桥接

采用间接桥接连接多个域时，所需的半桥数目绝对不会超过域的数目。如果我们将大多数用户使用的域的协议设置为公共协议，则数目会更少，如图 6.5 所示。

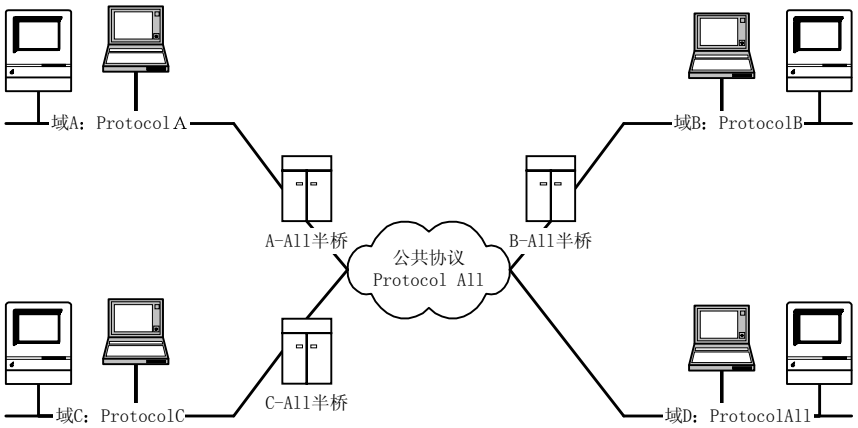

图 6.5　用来联系四个域的间接桥接

当然，采用间接桥接时，传递效率相对较低。因为每个使用非公共协议的域希望将信息传递到另一个使用非公共协议的域时，首先需要将源域的信息转化为公共协议支持的格式，然后，再将公共协议支持的格式转化为目的域协议支持的格式进行传递。

如果从实现途径上划分，桥接机制可以分为请求级别的桥接以及内联桥接。

6.3.3　请求级别的桥接

当桥接使用各种公共的 ORB APIs 来实现时，就是请求级别的桥接（Request-Level Bridging）。这些公共的 ORB APIs 主要来自以下几个 ORB 构成部分：

- 动态激发接口 DII。请求级别的桥接可以调用这组函数获得其它域的对象实现、动态构造请求、激发请求。
- 动态框架接口 DSI。如果在本地 ORB 上没有对象实现的接口框架，需要调用动态框架接口生成接口框架代理，调度远程的对象实现。
- 接口仓库。这组函数为请求级别的桥接提供对象的有关信息以及各种操作的签名。
- 对象适配器。通过对象适配器，请求级别的桥接可以提供引用其它域的对象实现的对象引用。
- CORBA 对象引用以及互操作对象引用。只有通过对象引用，才能正确激发服务提供对象。关于互操作对象引用，我们稍后再作解释。

一般情况下，请求级别的桥接如图 6.6 所示：

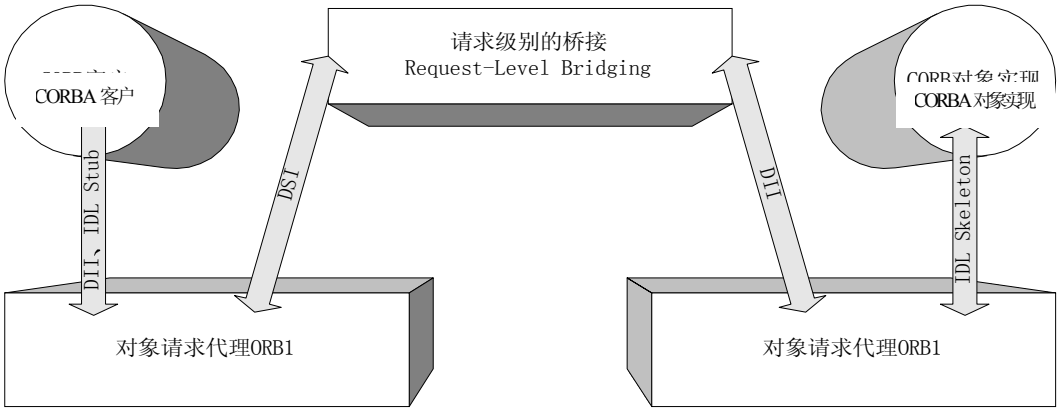

图 6.6　请求级别的桥接

请求级别的桥接有两种不同的有效作用范围：指定接口范围内有效、通用接口范围内有效。其区别如下所示：

- 指定接口范围内有效。这时，桥接仅适用于某些特定的接口。被支持的接口需要将 IDL Stub、IDL Skeleton 等信息预先装入桥接机制。
- 通用接口范围内有效。借助动态激发接口 DII、动态框架接口 DSI、接口仓库等现有功能，我们也可以构筑适用于任何接口的桥接机制。

显然，指定接口范围内有效的桥接不如通用接口范围内有效的桥接灵活，当用户修改、添加接口时，都必须更新桥接中的接口存根及接口框架。但是，指定接口范围内有效的桥接效率会高于通用接口范围内有效的桥接，这也许是一种补偿。

6.3.4　内联桥接

如果构建桥接时并没有使用公共的 APIs，就得到内联桥接（in-line Bridging）。在内联桥接中，连接是通过 ORB 内部各种信息表示格式直接转化形成的。由于信息格式直接转化，效率显著提高。内联桥接的示意图如图 6.7 所示。

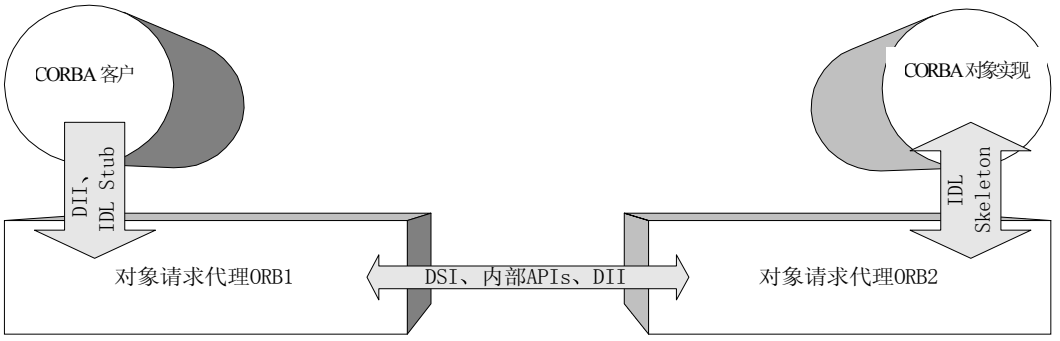

图 6.7　内联桥接

由于内联桥接无需调用过多的公共 APIs，直接转换信息格式，因此，桥接速度相当快捷。目前，大多数厂家都使用 CORBA 规定的公共协议 GIOP/IIOP 实现不同 ORB 之间的通信，为构建内联桥接提供便利。

6.4 互操作对象引用

对象引用是实现 CORBA 服务请求的关键参数。因此，当我们在网络上互相传递各种互操作信息时，也必须来回传递对象引用。不过，在实际应用中，对象引用不仅仅是一个表示对象实例的参数，还是一组相关参数构成的结构。因此，在各种 CORBA 桥接机制中，应该使用互操作对象（Interoperable Object Reference）引用来传递对象引用。互操作对象引用可以用缩写 IOR 代替。

IOR 由下列信息构成：

- 被引用对象的类型信息。在网络中传递对象引用时，需要在各个域中的 ORB 内始终保持对象类型一致。
- ORB 采用的协议信息。桥接可以根据 ORB 需要，转化各种信息的传递格式。但是，我们必须提供适当的参数，让桥接决定把信息从哪种协议支持的格式转化为另外一种协议支持的格式。
- 请求中采用的 CORBA 服务的信息。客户请求中可能需要激发各种 CORBA 服务，比如，事物处理 Transaction、安全服务等等。在 IOR 中包含这些信息，可以减少上下文对象中需要传递的信息。
- 对象引用是否为空。如果对象引用为空，桥接的工作负担会显著降低，避免许多不必要的工作。

根据 OMG 规定，IOR 由类型信息及带有一个或多个轮廓标签（TaggedProfile）的轮廓文件（profiles）组成。与 IOR 有关的定义如下所示：

例6.1　与 IOR 有关的定义。

```
module IOP{//IOR 在 IOP 模块中定义
        typedef unsigned long ProfileId;
        const ProfileId TAG_INTERNET_IOP = 0;
        const ProfileId TAG_MULTIPLE_COMPONENTS = 1;
        struct TaggedProfile {//用来表示轮廓标签的结构
            ProfileId tag;
            sequence <octet> profile_data;
        };
        struct IOR {//互操作对象引用 IOR 的定义
            string type_id;//类型信息
            //轮廓文件的定义，可以带有一个或多个轮廓标签
            sequence <TaggedProfile> profiles;
        };
        typedef unsigned long ComponentId;
        struct TaggedComponent {//标签元素的定义
            ComponentId tag;//标签号码
            sequence <octet> component_data;//元素内容
        };
        //带有多个标签元素的轮廓文件
        typedef sequence <TaggedComponent>
                                        MultipleComponentProfile;
};
```

除类型信息外，其余用来辨别对象引用的信息都记录在轮廓文件中。至于如何定义和使用轮廓文件，这里不再赘述。

现在，我们举例说明 IOR 如何穿越域边界，在使用不同协议的 ORB 之间传递。假设有两个不同的域，一个构筑在使用 TCP/IP 协议的网络之上，另一个构筑在使用 Novell IPX 协议的网络之上。两个域内都有各自的 ORB。TCP/IP 域内的 ORB 采用 DCE CIOP 互操作协议；Novell IPX 域内的 ORB 采用 GIOP 互操作协议。如图 6.8 所示：

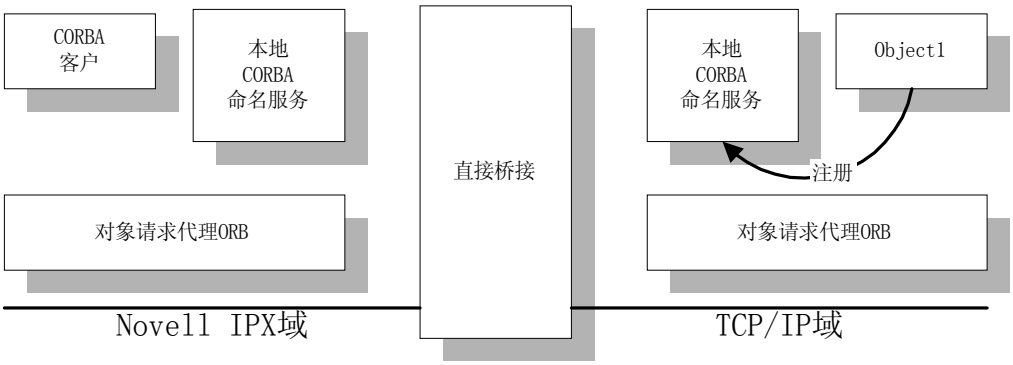

图 6.8　一个对象实现注册后可以在不同域中被引用

两个域中都拥有自己的本地 CORBA 命名服务，通过桥接，两个命名服务中都可以包含其它域的信息。也就是说，一旦对象实现 Object1 在 TCP/IP 域中注册成功，IPX 域中也可以得到相关的信息。假设我们可以将上述对象实现的信息表示为 Domain/host/uObject/TCPIP/DCE-CIOP/ApplicationOne/Object1，用来说明该对象所处的域、主机、使用协议、应用程序分类等多种信息。当然，如何表示这些信息最终取决于 CORBA 命名服务。

如果 IPX 域中的客户希望获得对象实现 Object1 的引用，就可以向本地命名服务发出请求（或者调用其中已有的某些操作）。本地命名服务则根据一些形式类似 Domain/host/uObject/TCPIP/DCE-CIOP/ApplicationOne/Object1 的信息，通过桥接向 TCP/IP 域的命名服务转发要求获得对象引用的请求，如图 6.9 所示：

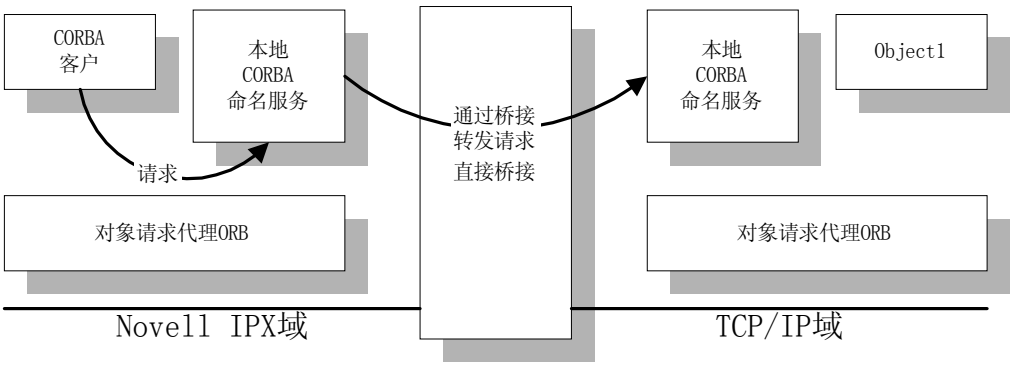

图 6.9　本地命名服务向其它域转发有关请求

TCP/IP 域中的命名服务获得本地对象实现的对象引用后，将通过桥接把它返回 Novell IPX 域。由于 Novell IPX 域中的 ORB 无法通过本地接口框架激发有关 Object1 的方法，桥接将通过动态框架接口 DSI 构造一个代理框架，并把该代理框架的引用添加到互操作对象引用 IOR 记录的信息中，与由 TCP/IP 域中命名服务传递过来的对象引用一同经过协议转换后返回给客户，如图 6.10 所示：

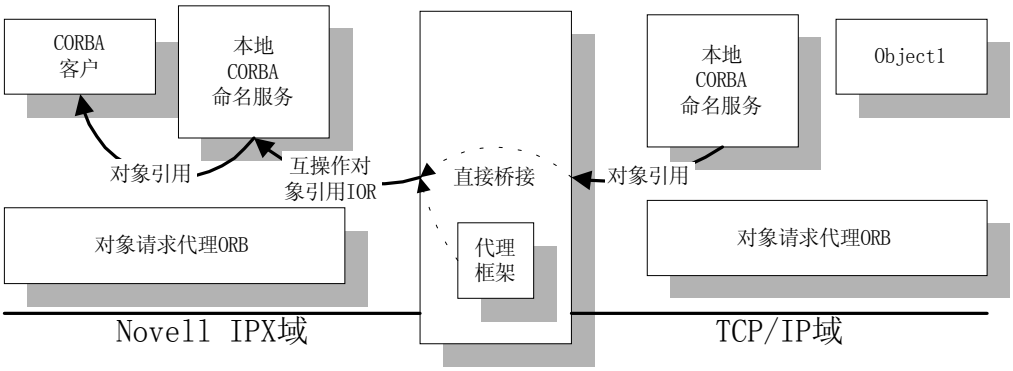

图 6.10　桥接构造互操作对象引用 IOR

最初提出要求的客户对这些幕后工作并不了解，他会获得一个对象引用，并不知道这实际上是一个互操作对象引用 IOR。当然，本地 ORB 知道这是一个互操作对象引用 IOR，但并不关心 IOR 中还包括了一个代理框架的引用，也不关心对象实现所在域采用的协议。

如果客户通过先前获得的对象引用激发某些请求，本地 ORB 将通过 IOR 与桥接使用 GIOP 协议通信，而桥接会调用代理框架对象，将客户的请求转换为 DCE CIOP 格式，并激发真正的对象实例，完成服务。

6.5　通用 ORB 互操作协议 GIOP 及其实现

为了实现 ORB 与 ORB 之间的通信，OMG 规定了两种互操作协议：通用 ORB 互操作协议 GIOP 以及特定环境 ORB 互操作协议 ESIOP。与 OMG IDL 相似，这两种通信协议只是一些抽象的规范，用户必须采用不同网络协议去实现这两个抽象协议。当我们用 TCP/IP 实现 GIOP 时，通常称为因特网 ORB 互操作协议 IIOP；而 ESIOP 抽象协议的一个实现例子是 DCE-CIOP。

通用 ORB 互操作协议 GIOP 由以下三部分构成：

- 公共数据表示 CDR（Common Data Representation）
- GIOP 消息格式
- GIOP 信息传递。

下面分别说明这三个部分。

6.5.1　公共数据表示 CDR

公共数据表示 CDR（Common Data Representation）定义了表达 OMG IDL 数据类型的标准格式，在 OMG IDL 数据类型与二进制流之间建立起必要的映射，以便网络传输。公共数据表示 CDR 具有以下一些特点：

- 允许改变字节顺序。每个 GIOP 消息均带有相应的字节顺序标志，当通信双方字节顺序相同时，不必进行字节顺序转换。否则，由消息发出方决定字节顺序，接收方可以根据字节顺序标志调整字节顺序，以适应本地需要。
- 基本数据类型边界对齐（Aligned primitive types）。OMG IDL 中的基本数据类型应该根据其自然边界（natural boundaries）对齐。
- 与 OMG IDL 中的数据类型建立完整的映射。

总之，公共数据表示 CDR 采取的策略是“接收方根据需要调整”，综合考虑了灵活性与传输效率问题。

一般情况下，用户不会接触到公共数据表示 CDR。

6.5.2 GIOP 消息格式

GIOP 中为 ORB 与 ORB 之间的通信定义了七种消息，用来解决以下三类任务：

- 传递请求
- 定位对象实现
- 管理通信信道

所有 GIOP 消息都以一个统一的 GIOP 头开始，这个消息头采用 OMG IDL 定义，格式如下：

例6.2　GIOP 消息头的定义。

```
//这个例子说明 GIOP 消息头的格式
module GIOP {
        enum MsgType {
            Request, Reply, CancelRequest,
            LocateRequest, LocateReply,
            CloseConnection, MessageError
        };
        struct MessageHeader {
            char magic [4];
            Version GIOP_version;
            boolean byte_order;
            octet message_type;
            unsigned long message_size;
        };
};
```

其中，GIOP_version 用来记录本消息采用的 GIOP 协议的版本号，byte_order 说明字节顺序，message_type 说明本消息是七个 GIOP 消息中的哪一个，message_size 说明本消息的全部字节数。

七种 GIOP 消息的简要说明如下：

- Request——请求消息。本消息由 GIOP 消息头、请求头、请求体构成。请求头记录了服务内容，包括请求标志号、对象引用、操作名称及其它信息。请求体中包括所有方向属性为 in 或 inout 的参数。
- Reply——响应消息。本消息由 GIOP 消息头、响应头、响应体构成。响应头记录了服务的内容，包括与原请求消息对应的请求标志号以及服务执行状态码。如果状态码为 NO_EXCEPTION，那么，响应体中将包括对应请求的返回值，以及对应请求中所有方向属性为 out 或 inout 的参数。如果状态码为 USER_EXCEPTION 或者 SYSTEM_EXCEPTION，那么响应体中将包括执行服务操作时发生的异常信息。如果状态码为 LOCATION_FORWARD，则表示被引用的对象实例已经迁移到别的站点，响应体中将包含一个新的互操作对象引用 IOR，以便有关的 ORB 重新定向先前发出的请求。
- CancelRequest——取消请求消息。本消息由 GIOP 消息头、取消请求头构成。取消请求头中包括需要被取消的请求的请求标志号。本消息可以通知有关服务器，客户希望取消先前发出的某个请求。
- LocateRequest——定位请求消息。客户可以向服务器发送该消息，确定某个对象引用是否有效，或者目前对应的服务器是否能够完成某项服务。本消息由 GIOP 消息头以

及定位请求头构成。定位请求头包括需要定位的对象引用及请求标志号信息。

- LocateReply——定位响应消息。服务器通过本消息来响应定位请求消息 LocateRequest。本消息由 GIOP 消息头、定位响应头、定位响应体构成。定位响应头包括被响应请求的请求标志号以及一个内容被重新定义的执行状态码。如果状态码为 UNKNOWN_OBJECT，表示定位失败，没有该对象引用。如果状态码为 OBJECT_HERE，表示定位成功。如果状态码为 OBJECT_FORWARD，表示对象引用已经移动到别的站点，此时定位响应体才有意义，包括一个新的互操作对象引用 IOR。
- CloseConnection——关闭连接消息。本消息由服务器发送给 ORB，表明服务器将关闭与 ORB 的连接，不再继续提供响应。本消息仅仅在服务器中还有从某个 ORB 发出的、没有来得及处理的服务请求时才能发出。它通知相关的 ORB，由于某种原因，客户发出的请求不会再被响应；如果 ORB 愿意重新启动一个请求，一定是安全的，不会收到任何干扰响应消息。此时，ORB 将向客户引发通信异常 COMM_FAILURE，并把执行状态设置为 COMPLETED_MAYBE。本消息仅由 GIOP 消息头构成。
- MessageError——消息错误消息。如果接收方不能理解 GIOP 消息，就可以发出本消息作为响应。本消息仅由 GIOP 消息头构成。

6.5.3 GIOP 消息传递

GIOP 消息传递主要是指如何在网络上传递 GIOP 数据及消息。用来进行 GIOP 消息传递的网络应该具备以下一些特点：

- 应该提供面向连接的服务
- 传输可靠，应该保持传输字节的顺序、提供足够的分发确认
- 连接失序时，应该及时被通知
- 传输内容可以被统一当作字节流处理，不需要限制长度、对齐方式
- 连接初始化应该能够达到 TCP/IP 模型要求的性能

而 CORBA 采用的 GIOP 协议本身具有以下特点：

- 非对称连接。CORBA 客户与 CORBA 服务提供对象的角色不同，客户方通常发起连接，发送请求消息，服务器接收连接，发送响应。
- 不同请求可以共享相同连接。由于连接的发起者是 ORB，因此，使用相同 ORB 的不同客户有可能共享与某些远程 ORB 建立的连接。而请求消息本身包含区别客户、对象引用的信息。
- 允许请求/响应交叉。由于连接者是 ORB，请求消息/响应消息可以通过异步方式传递。ORB 会根据请求标志号自动匹配请求/响应，无需客户操心。

GIOP 协议不能直接使用，必须通过已有的网络协议实现。比如，TCP/IP、Novell IPX 以及各种 OSI 协议都可以用来实现 GIOP。

6.5.4 因特网 ORB 互操作协议 IIOP

因特网 ORB 互操作协议 IIOP（Internet Inter-ORB Protocol）是通用 ORB 互操作协议 GIOP 在 TCP/IP 上的实现。IIOP 协议中关于数据表示和消息格式等规定都与 GIOP 完全相同，只是对于消息传递部分进行具体化，规定了如何采用 TCP/IP 来连接 ORB，交换 GIOP 消息。

IIOP 规定“服务器方”将提供一个 IP 地址及服务端口号，同时应该侦听传入的连接请求；“客户方”则应该具备发起连接、初始化连接的功能。“服务器方”接收到连接请求

时，必须明确建立连接或者说明不予连接的理由。连接建立后，“客户方”可以发送Request、LocateRequest 或 CancelRequest 消息，“服务器方”则可以发送 Reply、LocateReply、CloseConnection 消息，双方都可以发出 MessageError 消息。发出或者收到 CloseConnection 消息后，双方都需要关闭已经建立的 TCP/IP 连接。

实际上，任何一个 ORB 既是服务器，又是客户机。这是 CORBA 互操作得以实现的根本要求。

6.6 特定环境 ORB 互操作协议 ESIOP 及其实现

为了给特殊环境中的 ORB 提供互操作，CORBA 定义了一种非 GIOP 的互操作协议——特定环境 ORB 互操作协议 ESIOP（Environment-Specific Inter-ORB Protocol）。比如，当用户需要具有实时响应的 ORB 系统时，就可以采用 ESIOP。

分布式环境公共 ORB 互操作协议 DCE-CIOP（DCE Common Inter-ORB Protocol）是 ESIOP 的一个实现。

DCE-CIOP 的数据表示格式 CDR 与 GIOP 一致。七个 GIOP 消息在 DCE-CIOP 中被简化为两个远程调用：locate 以及 invoke。而且，GIOP 的消息传递机制主要是互联网网络传输，DCE-CIOP 却依赖于远程方法调用 DCE RPC 中提供的服务来执行 locate 以及 invoke。locate、invoke 函数都把请求信息及响应信息作为参数传递，采用同步方式执行。

在 DCE-CIOP 中，互操作对象引用 IOR 由类型信息及带有多个标签的轮廓文件构成。每个标签代表一个不同参数成分，共有以下六个：

- 用一个 DCE 字符串表示的服务器进程信息
- 为 DCE 命名服务辨别服务器进程的信息
- 被引用对象实例的信息
- 确保 DCE-CIOP 客户辨认具有相同结束点（endpoint）的对象的信息
- 供用户决定发送定位（locate）消息还是请求（invoke）消息的信息
- 供用户确定何时不具备基于管道（pipe-based）的接口的信息

6.7 CORBA 互操作层次结构

综上所述，CORBA 互操作的层次结构如图 6.11 所示。

对于 CORBA 用户而言，CORBA 互操作就是指各种分布式对象或应用程序，不论它们是本地还是远程对象，都能被无差别的透明使用。

CORBA 互操作的第二个层次就是互操作基础设施和相关的机制，包括各种桥接、ORB 以及接口定义语言。

CORBA 互操作的第三个层次就是 ORB 与 ORB 之间、桥接与桥接之间的通信协议。这些协议本身都位于网络应用层协议之上。主要有 GIOP、ESIOP。如果将位于应用层的互操作协议加以实现，就可以获得 IIOP、DCE-CIOP 等应用类型的互操作协议。

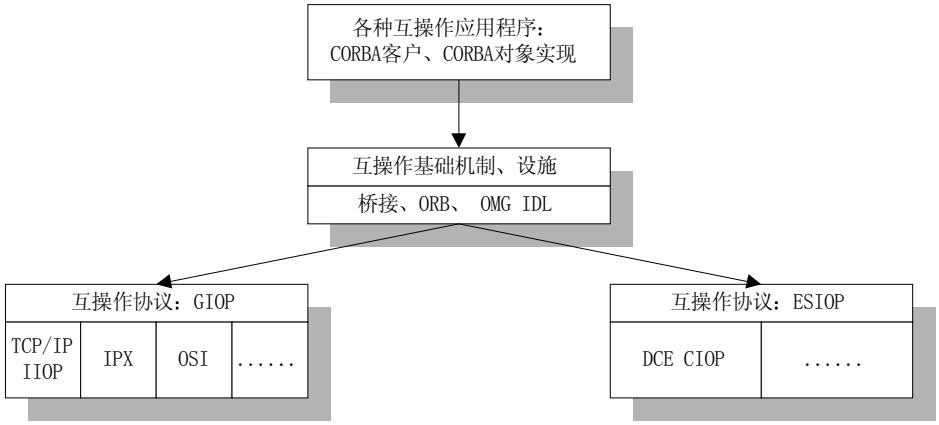

图 6.11　CORBA 互操作的层次结构

6.8　本章小结

CORBA 互操作是 CORBA 客户得以透明使用、集成 CORBA 分布式对象的核心机制。互操作的核心内容就是 ORB 与 ORB 之间的通信。

ORB 可以位于不同的域中。域就是满足一定特征的网络区域、网络计算机集合。CORBA 的域可以是任何技术域、管理域或者策略域。不同的域可以通过桥接连接起来。

与划分域的标准相对应，桥接可以划分为技术桥接、策略桥接。它们既可以是直接桥接，也可以是间接桥接；既可以建立在请求级别之上，也可以是内联桥接。

对象引用在桥接中通过互操作对象引用 IOR 传递。实际上，互操作对象引用 IOR 由类型信息及带有一个或多个轮廓标签的轮廓文件构成，记录了被引用对象的类型信息、ORB 采用的协议信息、请求中采用的 CORBA 服务的信息、对象引用是否为空的信息以及对应的代理框架对象引用信息。

CORBA 有两种基本互操作协议：通用 ORB 互操作协议 GIOP、特定环境 ORB 互操作协议 ESIOP。

将这两个抽象协议用某些网络协议实现之后，我们可以得到因特网 ORB 互操作协议 IIOP 以及分布式环境公共 ORB 互操作协议 DCE-CIOP。

对于 CORBA 而言，CORBA 互操作实际上是一个三层体系结构：可互操作的分布式 CORBA 对象，实现互操作的设施，如桥接或 ORB，各种互操作赖以通信的互操作协议及其具体实现。

第 7 章　程序员眼中的电子商务——分布式软件

本章内容提要：

- *电子商务是什么*
- *电子商务的组成部分*
- *电子商务的分类*
- *电子商务活动的主要流程*
- *电子商务的交易模式*
- *电子商务中的支付手段*
- *电子支付安全协议*
- *电子商务安全*
- *电子商务软件的若干要求*

7.1　电子商务是什么

现在，我们先暂时停止对 CORBA 内幕的继续探讨，来看看一场影响深远的革命：电子商务。无疑，我们已经领会 CORBA 是一种分布式软件开发方式、一种分布式软件开发标准。但是，我们还应该了解一点：电子商务软件是分布式软件的最佳实例，分布式电子商务软件是电子商务得以实现的重要技术支撑。

因此，本章将首先遍历一下电子商务中涉及的方方面面，说明软件开发人员对于电子商务的切入点就是分布式软件开发，并在此基础上考察一般电子商务软件系统应该具备的共性。这些共性问题恰好是“CORBA 应该具备哪些基本服务”的解答。

如果说 CORBA 是分布式软件开发的利器，电子商务则是必争的法宝。

7.1.1　电子商务的组成部分

时下，电子商务正如火如荼。可什么是电子商务呢？电子商务就是商务活动电子化，人们可以通过各种电子手段实现商务活动的方方面面，是商业活动的现代化。

电子商务之所以能够具有强大的号召力、生命力，主要由于它是一个社会、技术的综合产物，囊括了社会各方面机构、服务以及各种高新技术。电子商务通常由如图 7.1 所示几部分组成。

如果电子商务是商务活动电子化，那么，我们大多数人都已经在进行着电子商务活动的创造，事实的确如此。网络工程师正在构筑、改进和维护的因特网是电子商务重要的运作环境，软件工程师设计的网站、服务器、软件是电子商务重要的功能载体，管理人员提倡的 B2B（企业与企业）、B2C（企业与客户）、C2C（客户与客户）是电子商务新型的运行模式，货物速递是电子商务新型的交货途径……

作为现代化的商务活动，电子商务需要各行各业人员的参与，也必将服务于各行各业人员。因此，对于程序员而言，我们可以肯定地说：“我们已经涉足电子商务了”。

图 7.1　电子商务的组成部分

7.1.2　电子商务的分类

目前，电子商务可以根据交易对象、活动内容以及使用的网络类型作出不同划分。

按照交易对象，电子商务可以分为以下四种类型：

- 企业与消费者之间的电子商务。这种类型的电子商务类似于在线商品买卖，消费者利用计算机网络、电视购物、电话购物等形式直接参与经济消费，是比较流行的电子化零售。目前，因特网上已经遍布各种类型的“商业中心”，提供从鲜花、书籍到计算机、汽车等各种商品；一些电视台推出了各具特色的电视购物，提供化妆品、健身器材、服饰、医药等多种商品及服务；航空公司、铁路部门也推出了灵活方便的电话订票业务它们都可以看作本类型电子商务的实例。本类型电子商务通常被简称为 B2C。
- 非特定企业间的电子商务。这种类型的电子商务不关心企业之间的持续交易，希望通过开放的信息网络对每笔交易都寻找最佳伙伴，并与其进行从订购到结算的全部交易活动。企业之间仅存在简单交易行为。
- 特定企业间的电子商务。企业之间过去一直保持交易关系或者今后需要继续保持交易关系。因此，企业之间具有相同的经济利益，需要在某种程度上共同运营，需要高度共享的信息网络。通常，企业与企业之间的电子商务被简称为 B2B，既需要面向非特定企业又需要面向特定企业。
- 政府与企业之间的电子商务。这种类型的电子商务覆盖政府与企业之间的各项事物。比如市场调控、企业税收、财务检查等等。

按照商务活动的内容，电子商务可以分为以下两种类型：

- 间接电子商务，也就是有形商品的交易。这种类型的电子商务仍然需要利用一些传统的非电子渠道，比如邮政服务或者货物速递等才能最终得以完成。
- 直接电子商务，也就是无形商品或者服务的交易。这种类型的电子商务往往涉及电子杂志、游戏软件、教育软件、在线娱乐、在线咨询等服务，可以通过电子手段直接实现。

由于计算机网络是信息、数据流通的最佳途径，我们也可以按照实现电子商务活动时采用的网络类划分来进行电子商务活动：

- 专用的电子数据交换网。电子数据交换 EDI（Electronic Data Interchange）就是按照国际标准组织定义的格式，将商业文件标准化，并利用计算机网络在贸易伙伴之间进行数据交换和自动处理。私有网络上的 EDI 最早开始于 60 年代，而这些私有网络也由企业自己建立、维护和管理，拥有可靠的信用保证和安全措施。采用这种网络的电子商务主要应用于企业与企业之间、企业与批发商、批发商与零售商之间的批发业务。
- 因特网 Internet。因特网允许不同主机之间采用 TCP/IP 协议进行通信，是电子商务得以普及、繁荣的重要基础设施。甚至在许多人心目中，电子商务就是因特网上的贸易。
- 企业内部互联网 Intranet。Intranet 是在互联网基础上发展起来的。通过在原有局域网上附加一些特定的软件，将局域网与因特网互联，并形成企业内部的虚拟网络。该虚拟网络受到防火墙的保护，仅允许内部、外部人员授权使用其中的信息。企业内部各部门之间的商务活动多采用 Intranet 进行。
- 企业外部互联网 Extranet。在多个 Intranet 之间构筑共享数据库并通过 TCP/IP 协议将它们互联，就得到企业与业务伙伴或者客户之间共享信息的企业外部互联网。

如何划分电子商务可能有很多标准，但是，电子商务的本质内容不会因此而改变：它是信息社会中人们进行各种交易的方式。

7.1.3 电子商务活动的主要流程

一般情况下，电子商务活动由如图 7.2 所示的五个流程构成：包括信息共享、洽谈订购、支付、交付以及售后服务等过程。

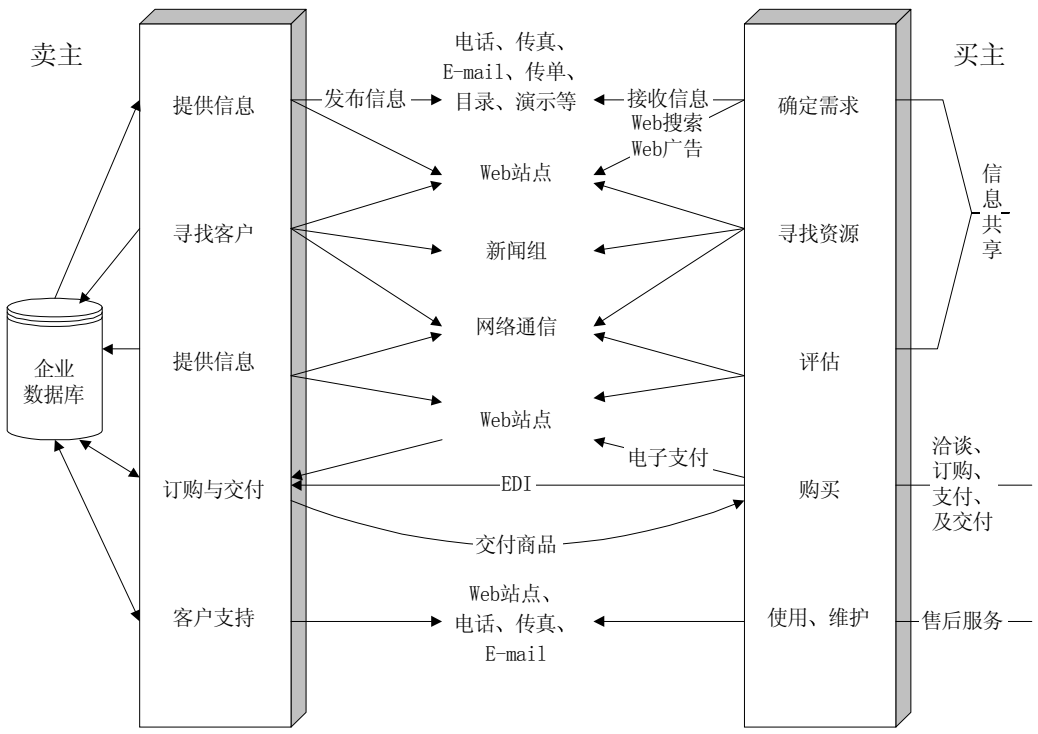

图 7.2 电子商务活动的主要流程

在信息共享阶段，买卖双方或者交易双方通过信息社会提供的各种手段进行必要的信息“交流”。买方根据自己需要的商品，从电视广告、网站广告、搜索引擎、聊天室、中介机构等多种渠道筛选出一些侯选产品、商家；卖方则将商品的信息通过广告、网络新闻、常见问题 FAQ、新闻发布会、产品演示会等多种形式公布。

随后，买方可以与一些侯选卖方进行实质性洽谈、订购，磋商双方在交易中的权利、义务、购买商品或服务的种类、数量、价格、交付地点、交付期限、交付方式、违约及索赔合同条款。磋商过程广泛依赖各种电子手段，如传真、E-mail、Web 定单、数字签名等等，磋商结果通常以电子文件形式保存。

因为买卖双方通常无需直接会晤，“一手交钱一手交货”在电子商务中表现出新的特点：支付过程主要借助于信用卡、电子支票、数字现钞或微钞实现。对于不同商品或服务，卖方会根据其本身特点，采用不同的交付方式。

交易完成后，买卖双方的关系并没有由此终结。买方可以随时向卖方提出各种服务请求，包括常见问题 FAQ 求解、维护指南、维修等等；卖方则通常将买方数据录入数据库中，并通过定期跟踪调查来维护已经建立的市场、开拓新的市场、寻求更新产品或服务品质的“动力”。

不同类型的电子商务通常都包括上述五个流程。但是，在交易过程中，买卖双方并不一定就是商品或服务的最终客户以及商品或服务的生产厂家，因而，电子商务又演化为商品直销模式以及商品中介交易模式。

7.1.4 电子商务活动的交易模式

商品直销模式是指商品或服务的生产厂家与商品或服务的最终客户直接交易的商务形式，其过程如图 7.3 所示。

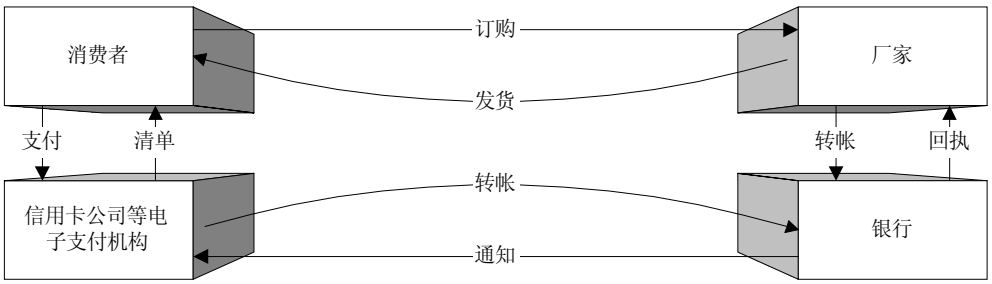

图 7.3 电子商务中的商品直销模式

这种模式的交易通过以下步骤完成：

- 消费者、厂家通过电子手段进行信息共享
- 消费者与厂家通过电子手段直接洽谈、订购
- 消费者选择某种形式的电子支付
- 厂家检查消费者的电子支付，及时向被认可的客户交付产品或服务
- 电子支付机构、银行进行必要的转帐，并通知交易各方

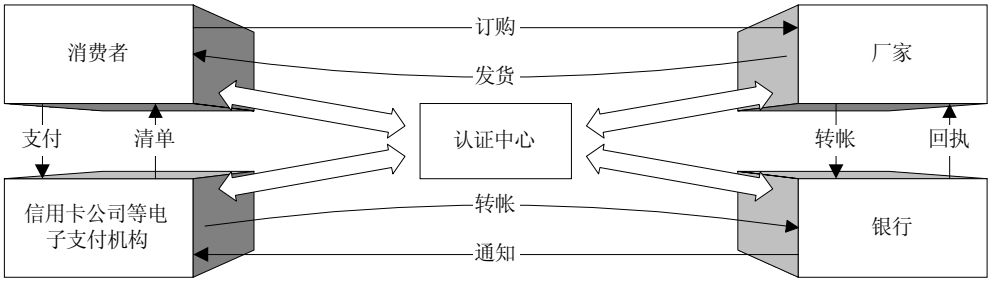

图 7.4　借助认证机构的商品直销模式

有时，为了确保交易的安全，需要一些特定的认证机构对交易双方、电子支付机构、银行进行认证，以确定其身份真实性，这时，商品直销模式演化为图 7.4 所示。

目前，认证机构均支持由 MasterCard、Visa、IBM、Microsoft、Netscape、Sun、Oracle 等公司共同开发的 SET 协议。

商品中介交易模式指商品或服务的生产厂家与商品或服务的最终客户通过一些类似虚拟商场的中介机构进行交易的商务形式。其过程如图 7.5 所示。

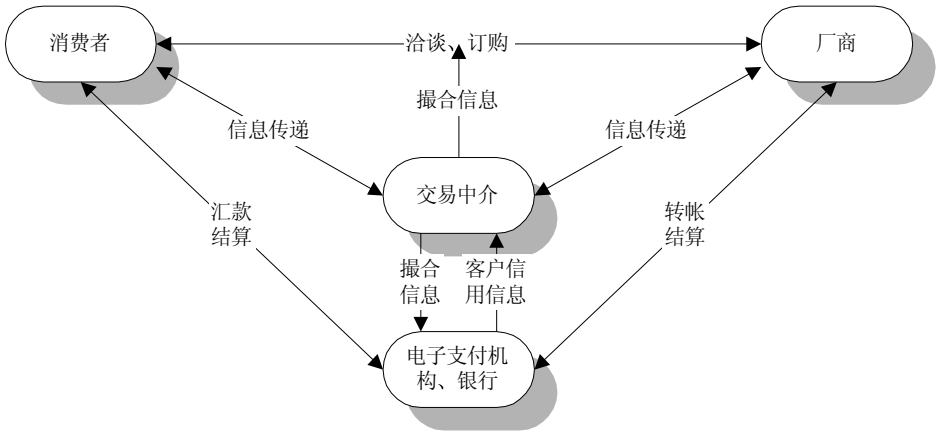

图 7.5　商品中介交易模式

商品中介交易模式通常通过以下步骤完成：

- 消费者、厂商通过电子手段将需求信息、供给信息传递到交易中介机构；中介机构通过电子手段向所有参与者提供各种交易数据、市场信息。
- 中介机构根据自己掌握的信息撮合消费者与厂商成为交易伙伴。
- 消费者在中介机构指定的电子支付机构、银行支付
- 中介机构通知厂商交付商品或服务
- 消费者提取商品或服务
- 中介机构将消费者货款转交给厂商

当然，商品直销模式与商品中介交易模式各有长短、互相补充。商品直销模式环节少、速度快、费用低，但是无法确保消费者能够不同不良厂商进行交易；商品中介交易模式便于消费者集中挑选商品，便于厂商准确估计市场潜力及动向，也可以避免不良消费者“拿货不给钱”，避免不良厂商兜售劣质商品或者“拿钱不给货”。

了解电子商务的本质、组成、分类、电子商务活动的主要流程以及交易模式后，我们应该明白一点：如何从电子商务活动中赢利取决于经营电子商务的策略，主要是一个管理

问题，但是，不管采用哪种商务策略，必须通过一定的技术得以实现。这些技术包括网络技术、电子支付技术、安全技术、无线通信技术、移动通信技术、Web 技术以及分布式软件技术。

显然，作为程序员，我们对电子商务的切入点应该是通过 Web 技术以及分布式软件技术实现各种商务策略。不过，为了全面了解电子商务，我们不妨继续探讨一下电子支付及电子商务安全方面的有关内容。

7.2 电子支付

在电子商务中，支付问题必须通过电子支付解决。所谓电子支付是指交易当事人，包括消费者、厂商、商品中介、金融机构使用安全电子手段通过计算机网络进行货币支付或资金流转。

电子支付具有以下特点：

- 电子支付系统完全是数字化的，货币或资金的流动表现为数字信息的流转
- 电子支付的工作环境是计算机网络
- 电子支付方便、快捷，（理论上）不受时间、空间条件的限制

目前，电子支付方式可以分为三种类型：

- 电子货币类，包括电子现钞、电子钱包等
- 电子信用卡类，包括智能卡、借记卡等
- 电子支票类，包括电子支票、电子汇款、电子拨款等

这些电子支付方式都有自己的特点及运作模式，适用于不同的交易过程。

7.2.1 智能卡（Smart Card or IC）

智能卡最早于法国问世，属于嵌入式芯片存储装置，包括以下三个部分构成：

- 智能卡程序编制器。程序编制器主要在智能卡开发过程种使用，可以初始化智能卡、创建智能卡所记录的个人数据。
- 智能卡处理系统。包括智能卡操作系统及部分智能卡应用程序接口。智能卡处理系统应该具有移植性，以便集成到芯片阅读机、PC 机或其它服务装置上。
- 智能卡管理系统。主要是提供应用程序控制、管理智能卡的应用程序接口。

智能卡通过内部的嵌入式微型可控芯片存储、处理数据，其数据受用户的个人身份号码（PIN）保护。由于智能卡拥有独立的芯片及内置操作系统，因而非常易于进行功能扩充。

使用智能卡进行支付的过程如下：

- 用户通过电子设备进入电子商务网络。这些电子设备可以是计算机、POS、终端电话、商务通等等工具。
- 用户在电子设备附带安装的芯片阅读机中插入智能卡，智能卡根据记录数据自动登录到有关银行服务器内。
- 用户通过电子设备把智能卡上的现金转入卖方帐户或者把银行帐号内的现金下载到智能卡内。

目前，智能卡存在一些缺点，包括制作成本过高、不能一卡多功能或一卡多用、不同智能卡及芯片读写机之间缺乏互操作性等。不过，如同 CD-ROM 一样，一旦价格下降，人们会趋之若鹜。

在因特网上，还有一种简化的智能卡：用户通过在计算机上键入自己的智能卡号、密码后，就可以登录到银行服务器内进行支付或转帐。

7.2.2 电子现钞（E-Cash）

电子现钞很有可能成为电子商务中的下一个冲击波。通俗地说，电子现钞就是以数字形式发行的钞票。电子现钞将现金数值转换为一系列加密数字序列，通过这些数字序列来表示现实生活中不同面额的钞票。

与现实钞票一样，电子现钞由银行负责“发行”及“回收”。采用电子现钞进行支付通常按下列步骤完成：

- 用户在可以发行电子现钞的银行开设电子现钞帐户，并用一定的现金购买电子现钞使用证书。
- 用户通过各种电子设备，输入个人身份号码（PIN）、密码，登录到银行的电子现钞服务器，从中购买一定数量、一定面额的电子现钞，保存到自己的存储介质上。每个电子现钞都是被加密的数字序列，对应于一定面额的现钞。
- 用户与同意接收电子现钞的卖方洽谈、订购，并使用电子现钞支付所需商品或服务。
- 卖方与电子现钞发放银行进行结算。银行通过检查被加密的数字序列，验证电子现钞的真假、是否已经被回收，并将款项转给卖方或通知卖方是伪电子现钞。

电子现钞具有以下一些特点：

- 买卖双方都需要与银行建立授权关系
- 买卖双方、银行都需要使用相同的电子现钞软件、遵循共同的运作协议
- 由银行最终负责进行资金转移
- 身份验证由电子现钞直接完成。银行在发行电子现钞时使用了数字签名。交易过程中，电子现钞的真假有效性由银行负责检查。
- 买卖双方可以匿名使用。
- 具有现钞的实际特点，可以存、取、转让，适于小交易。

当然，电子现钞也存在以下若干问题：

- 不同银行发行的电子现钞如何兑换、不同国家发行的电子现钞如何兑换？显然，这需要电子现钞软件之间具有较强的兼容性、互操作性。
- 应该禁止还是应该允许非金融机构发行电子现钞？如果允许非金融机构发行电子现钞，如何解决买卖双方之一就是电子现钞的发行者的问题？
- 如何解决电子现钞丢失（比如用户存储介质损坏）以及电子现钞流动中的安全问题？

电子现钞对于一些软件商品、在线服务非常具有吸引力。比如，股市行情、天气预报、音乐库、图像库、网上游戏、法律咨询、医药咨询等等都可以按照信息量、时间进行计费，用户只需支付小面额的电子现钞即可。

7.2.3 电子支票（E-Check）

电子支票就是数字形式的支票。其中，以数字签名、个人身份号码（PIN）等方式代替了传统支票的手写签名，主要支付过程如下所示：

- 买卖双方达成购销协议并选择电子支票形式进行支付
- 买方通过电子商务网络向卖方发出电子支票，同时向有关银行发出同意付款通知
- 卖方通过有关验证机构检验电子支票，确认无误后向有关银行兑现支票
- 银行进行必要的验证后兑现支票或转帐

采用电子支票进行支付时，买卖双方以及有关银行均需要进行身份验证。

由于安全问题的影响，目前，人们对在线支票的兑现仍持谨慎态度。电子支票的普及恐怕还需要一个过程。

7.2.4 电子钱包（E-Purse）

我们在上街购物时，通常带有一个钱包，里面可能装有一定量的现金，用来应付小额交易，还可能装有 ATM 卡，以应付大额交易。电子钱包就是对现实钱包的模拟，用来装载电子现钞、电子信用卡等电子支付物品。

电子钱包通常以软件模块形式出现，可以安装在各种电子商务服务器中。当用户需要支付时，只需点击相应项目，即可完成。“携带”电子钱包进行电子商务交易，主要包括以下步骤：

- 买卖双方达成购销协议后，买方既可以选择使用与自己银行帐号相关的电子商务服务器上的电子钱包，也可以利用其它可靠的电子商务服务器上的电子钱包进行支付。在输入用户标志及密码，证明自己是该电子钱包的主人后，可以选择其中的某张电子信用卡或某些电子现钞进行支付。
- 与电子钱包有关的服务器接收到来自买卖双方共同确认的帐单后，向有关信用卡机构、银行发送各种验证、转帐、结算请求，处理有关事务。
- 如果支付顺利，卖方可以交付商品或服务，买方获得财务结算清单；如果支付过程中出现资金不够、透支等现象，与电子钱包有关的服务器会要求买方从电子钱包中选择其它电子信用卡或电子现钞进行弥补。

采用电子钱包进行支付时，还有一个非常重要的优点：即卖方始终不知道买方使用的是哪种支付手段，也无法窃取用户的电子信用卡号码及密码。

电子钱包实际上可以包容各种电子支付手段，是一种电子支付代理机制。

7.2.5 电子支付的安全协议

保证电子支付过程中的安全是一个必须考虑的问题。目前，电子支付的安全问题并没有形成一个公认的、成熟的解决办法。而 SSL 安全协议以及 SET 安全协议是两种主要的电子支付安全协议。

SSL 安全协议由 Netscape 公司率先设计，称为安全套接层协议。SSL 安全协议提供三方面服务：

- 认证用户及服务器，确保数据被发送到正确的客户机及服务器
- 加密被传递的数据
- 维护数据完整性，确保数据在传递中不被改变

但是，SSL 安全协议运行的基点是卖方被默认为是可以信赖的，卖方必须承担对买方信息的保密。因此，SSL 安全协议有利于卖方而不利于买方。正是由于这种缺陷，SSL 安全协议逐渐被 SET 安全协议取代。

SET 安全协议依赖于认证机构，希望达到以下目标：

- 确保信息在因特网上安全传输，防止数据被黑客或内部人员窃取。
- 确保电子商务活动参与者之间信息相互隔离。买方的信息被加密或打包后经由卖方到达银行、信用卡机构，卖方无法获得买方私人信息。
- 多方认证，对买卖双方、信用卡机构、银行均进行认证。
- 确保交易的实时性，支付过程在线完成。
- 效仿 EDI 交易形式，具有规范的协议及消息格式，具有较好的兼容性及互操作性。

目前，SET 安全协议也存在一些缺陷，比如：

- 协议没有说明银行在向卖方转帐前，是否必须收到买方的货物接收证书。否则，当卖方商品或服务不符合合同时，无法限制卖方。
- 卖方不能证明订单一定是由签署证书的买方发出的。

- 没有规定在支付事务完成后，如何安全地保存或销毁有关数据。

人们正在设法改进 SET 安全协议的不足。

7.3 电子商务的安全管理

电子商务的安全管理是电子商务另外一个比较重要的内容。

电子商务的不安全因素可能来自信息风险、信用风险、管理风险以及法律风险。

信息风险主要包括以下可能：

- 信息泄露。黑客可以通过 IP 欺骗冒名获取商务秘密、资源或信息；也可以通过线路窃听截获各种传递信息，威胁电子商务的安全。
- 篡改数据。病毒、黑客都可以非法删除、修改、伪造各种信息，破坏数据完整性，误导商务活动。
- 信息丢失。通信线路故障、安全措施不得力、自然灾害以及软硬件不兼容或损害都有可能导致信息丢失。

信用风险主要来自三个方面：

- 来自买方的信用风险。个人消费者可能通过恶意透支，使用伪信用卡、电子假钞等欺诈卖方，集团消费者可能故意拖延货款。
- 来自卖方的信用风险。卖方可能不能按照规定的时间、地点、数量、质量交付商品或服务，也可能拒不履行售后服务。
- 来自买卖双方的合伙欺诈。买卖双方还有可能勾结起来，欺骗中介机构、信用卡机构、银行。

而且，由于电子商务出现时间不长，当然存在众多的管理、法律纰漏。

从技术角度出发，电子商务安全问题主要依靠认证、防范等手段进行。

7.3.1 客户认证

客户认证 CA（Client Authentication）是指通过用户客户端主机 IP 地址进行认证的一种机制。客户认证主要包括身份认证以及信息认证。前者鉴别用户身份，后者确保信息通信的不可抵赖、完整及真实性。

身份认证可以通过三种基本方式或其组合方式来实现：

- 用户所知道的某个秘密信息，比如口令
- 用户所持的某种硬件秘密信息，比如智能卡
- 用户所具有的某些生物特征，比如指纹、声音、DNA、视网膜扫描等。当然，这种方式造价极高，仅适合于极度保密的场合。

根据认证采用因素的多少，可以分为单因素认证、双因素认证以及多因素认证。

信息认证必须实现以下要求：

- 对重要信息应该加密，即使被别人截获也无法得到其内容
- 保证信息完整，防止信息被篡改损害
- 对信息的来源进行验证，确保信息来源可靠。

现在，普遍采用私钥加密、公钥加密或者两者的组合方式来进行信息安全认证。

私钥加密体系加解密使用相同的密钥，被称为对称加密算法。由于通信双方共享密钥，因此通信方可以确定信息是否由相关人员发出，这是一种简单的信息来源认证方式。

但是，私钥加密体系也存在若干问题：

- 要求提供一个安全的渠道来传递密钥

- 密钥数目将随通信方增多而快速增长，难于管理
- 一般不能提供信息完整性鉴别。

公钥加密体系具有两个密钥，公开密钥及私有密钥。如果用公开密钥加密，则必须用对应的私有密钥解密；如果用私有密钥加密，则必须用对应的公开密钥解密，所以被称为非对称加密算法。由于这种算法较慢，仅适合于对少量信息加密。

在公钥加密体系中，信息加密传递过程可以采用如下步骤进行：

- 发送方生成一个“信息”密钥，并用接收方的公开密钥加密后传递给接收方。
- 发送方将信息用“信息”密钥加密后传递给接收方
- 接收方用自己的私有密钥解密发送方的“信息”密钥
- 接收方用解密后的“信息”密钥解密加密后的信息

这时，信息加密过程包括两个内容：密钥的加密传递、信息的加密传递。

如果希望确保信息完整性、验证发信人身份，就需要借助哈希函数（Hashing Function）来进行签名。哈希函数的输入是需要签名的信息或文件，输出是一组定长的代码，叫做数字签名。数字签名代表了信息或文件的特征，如果信息或文件发生改变，数字签名也会发生变化。不同信息或文件的数字签名不同。

加入签名和验证的信息传递过程通常如下所示：

- 发送方使用哈希函数获得待发信息的数字签名，然后用发送方的私有密钥将数字签名加密，并将加密后的数字签名附于要发送的信息之后
- 发送方生成一个“信息”密钥，加密并发送信息（包含已加密的数字签名）
- 发送方用接收方的公开密钥加密“信息”密钥，并传递给接收方
- 接收方用自己的私有密钥解密“信息”密钥
- 接收方用解密后的“信息”密钥解密加密的信息，并得到加密的数字签名
- 接收方用发送方的公开密钥解密加密的数字签名
- 接收方用哈希函数计算信息或文件的数字签名，并与解密后的数字签名进行对比。

显然，上述过程可以满足信息认证的三个要求：重要信息加密、确保信息完整及验证信息来源。

但是，还有一个问题需要解决：如何验证公开密钥与持有者的关系？因为公共密钥在传递过程中也可能被篡改、替换，所以，需要有一个机构来验证公开密钥与持有者之间的关系。这些机构就是电子商务中的认证机构。

在电子商务交易中，买卖双方应该互相提交一个由认证机构 CA 签发的证书来证明自己的身份。证书通常包含有用户的名字及公开密钥。证书主要有持卡人证书、商家证书、支付网关证书、银行证书以及发卡机构证书。

7.3.2 防范

电子商务中的防范措施非常多，包括使用网络安全检测设备、访问控制设备、安全浏览器/服务器软件、证书、防火墙等，也包括良好的安全管理制度。这些话题显然超越了本书的主题。

7.4 电子商务软件的要求

在概要了解电子商务的方方面面之后，我们应该明白，对于程序员而言，电子商务就是具有商务功能的分布式软件。

Web、Java 是目前流行的网络编程技术，不过，Web、Java 却绝对不能用来实现全部的分布式软件，也无法最终集成所有的分布式软件。因为它们仅仅是一些编程技术、工具

而已。相反，作为分布式软件的开发标准和体系结构，CORBA、COM/DCOM 正在构筑着分布式软件的整体蓝图，为分布式软件的最终集成铺平道路。我们相信，未来的分布式软件世界，是符合 CORBA、COM/DCOM 等结构，采用 Web、Java 等技术的精彩世界。

为了使分布式软件能够服务于电子商务，这些软件必须满足以下要求：

- 随时可用（Availability）。无论何时，只要买卖双方希望进行交易，分布式商务软件都应该能够响应。显然，当用户企图抛售自己的股票时，绝对不愿意发现自己的股票交易系统拒绝提供服务。
- 性能良好（Performance）。在电子商务时代，所有客户都显得缺乏耐性，如果响应时间频繁超过“2 秒”，恐怕就会有人抱怨商务软件性能过于糟糕了。
- 记忆持久（Durability）。在电子商务时代，的确可以进入无纸办公。但是软件系统必须有“经久不忘”的记忆力。当系统接收一份订单后，在相当长的时间内应该“能够回想起有关内容”，当完成一笔交易后，恐怕需要“铭记终生”。
- 具有预见能力（Predictability）。商务软件的执行结果应该符合商务逻辑，可以预见。显然，没有客户愿意在支付货款后被告知因为软件失误厂家没有收到订单；也没有厂家希望在交付商品后发现因为软件失误客户没有付款。
- 安全可靠（Security）。我们相信，软件系统、买卖双方、所有有关机构都希望确保自己正在进行的活动、操作安全可靠。
- 容易扩充、升级（Scalability）。随着商务活动的扩展，我们可能需要扩充软件系统的负载能力、功能。因此，分布式软件开发过程中就应该考虑这些未来问题。
- 适应性强（Adaptability）。商场如战场，瞬息万变。电子商务中真正获取利润的是商务经营策略。商务软件是商务经营策略的技术支持，因此也必须适合经营策略的千遍万化。如果每改变一下商务经营策略，就需要重新开发商务软件，显然会捉襟见肘。
- 灵活性强（Flexibility）。不可否认，分布式商务软件开发者的预见是有限的，但是，商机是无限的。如果客户希望同一些未预见的商务系统进行交易，那么，有关商务系统之间应该能够灵活集成、灵活交互，否则，必然造成“商机有界”。

显然，分布式商务软件的上述要求不可能采用一种技术或语言完成，必须制订一定的软件标准，规定一定的软件体系结构得以实现。本书前面章节已经说明，CORBA 就是为此目的设计的分布式软件开发方式及分布式软件开发标准。本书的后续章节将继续说明，CORBA 基本服务恰好能够用来实现分布式商务软件的上述要求。

7.5 本章小结

CORBA 与电子商务紧密相关。

电子商务就是商务活动的电子化，是人类社会进入到信息时代，进行商务活动的一种必然选择。虽然电子商务是一个牵扯社会各行各业的革命，但是，对软件开发人员的要求恐怕更加强烈：如果没有计算机网络、没有分布式软件，电子商务活动根本无法展开。

从不同的角度出发，电子商务可以划分为不同种类：B2C、B2B、C2C、间接电子商务、直接电子商务、Intranet 电子商务、Extranet 电子商务就是一些含义可能重叠的分类方式。

不管如何分类，电子商务的基本流程是相似的：买卖双方通过电子商务网络信息共享，通过电子商务网络洽谈订购，通过电子支付结算货款并交付商品，通过电子商务网络进行售后联系。如果买卖双方恰好是最终消费者及厂商，就形成了商品直销模式，否则，是商品中介交易模式。这两种模式都可以给企业带来勃勃生机。

在电子商务中，支付手段有电子货币、电子支票、电子信用卡等形式。而电子商务的安全问题则是一个非常引人注目的话题。

也许，你发现，目前的商务软件似乎都热衷于建设网站，开发 Web 页面、编制 Java 程序。实际上，Web 页面仅仅是分布式软件的“人机交互界面”， Java 仅仅是编制分布式软件的优秀语言。而电子商务软件也不会永久停留在“人机交互界面”阶段，必将朝着互联网软件集成的方向发展。如何自动生成 Web 页面，如何从纷繁的网络中自动提取、归纳、整理商务信息；如何进行分布式软件的快速集成；如何使软件功能能够适应电子商务的瞬息万变……一系列分布式软件问题迟早会迎面而来。

这些问题也引出了分布式商务软件需要满足的若干要求。当我们试图满足这些要求时，就会发现，我们应该制订一种分布式软件开发标准，形成一套分布式软件开发方式。CORBA 就是为此目的而诞生的。CORBA 基本服务可以实现分布式商务软件的上述要求。

对于程序员而言，电子商务就是分布式商务软件。因此，请不要忽略 CORBA。

第八章　CORBA 基本服务

本章内容提要：

- 厂对象 *Factory*、厂定位对象 *FactoryFinder*
- 对象的拷贝、移动及删除
- 角色对象 *Role*、关系对象 *Relationship*、节点对象 *Node*、角色工厂 *RoleFactory*、关系工厂 *RelationshipFactory*、节点工厂 *NodeFactory*
- 混合对象的生存期
- 持续对象服务（*Persistent Object Service*）的结构及有关对象
- 对象外化服务（*Externalization Service*）
- 名称对象 Name、命名上下文对象 *NamingContext* 及绑定迭代器 *BindingIterator*
- 对象洽谈服务（*Trader Service*）
- Push 模型及 Pull 模型、通用事件信道及类型事件信道
- 事务服务（*Transaction Service*）
- 并行服务（*Concurrency Service*）
- 属性服务（*Property Service*）
- 对象查询服务（*Query Service*）
- 对象包容服务（*Collection Service*）
- 安全服务（*Security Service*）
- 时间服务（*Time Service*）
- 许可服务（*Licensing Service*）

8.1　对象生存期服务

CORBA 基本服务也是由一些对象（类）实现的。这些对象用 OMG IDL 语言定义，提供 ORB 的厂家可以编程实现其中一些具有共性的对象及操作，而一般的对象实现开发人员具有义务编程实现相关接口。

具备 CORBA 基本服务的对象可以分为两类：

- 专用对象（Specific objects）。这些对象主要用来实现特定的 CORBA 基本服务，它们存在的目的就是提供 CORBA 基本服务。
- 一般对象（Generic objects）。这些对象通常由用户定义、实现，不过，它们也继承或具备 CORBA 基本服务接口。

显然，在所有基本服务中，首先应该考虑的是对象生存期服务：任何 CORBA 对象，都需要在合适的时候被创建、释放；在必要的时候进行拷贝、移动或删除。

8.1.1　厂对象 Factory

我们知道，对于 CORBA 客户或对象实现而言，最先获得的对象应该是 ORB 伪对象，随后，我们可以要求 ORB 产生我们需要的对象实例。一般情况下，产生一个对象实例需要解决以下问题：

确定对象实例的位置，比如，决定对象是本地实例还是远程实例

获取资源，比如，为对象申请内存、打开对应的数据流文件、封锁特定的硬件资源（打印机、芯片阅读机）。

向 BOA 注册以获得合法的对象引用

通知与对象命名服务、洽谈服务等有关对象，使 CORBA 客户可以通过以上服务引用

对象实例。

对象实例有关数据初始化

当然，创建不同对象实例时，需要满足的要求可能不同；用户在创建特殊对象时，还可能有特殊的要求。因此，我们无法用统一的方式产生所有对象实例。在 CORBA 中，专门用来产生别的对象实例的对象被称为厂对象 Factory。厂对象本身也是对象，它的作用是专门产生对应类型的对象实例。

并非所有对象都需要对应的厂对象。如果对象在应用中具有下列特性，就可以为其提供相应的厂对象：

对象可能具有多个实例；

经常需要拷贝移动对象实例；

创建对象实例时具有特殊要求；

通用厂对象的 OMG IDL 接口定义如下所示：

例 8.1 通用厂对象 Factory 的接口定义。

```
//这是通用厂对象 Factory 的接口定义
interface Factory{
        ……
        Object create();
        ……
}
```

通用厂对象主要用来创建一些通用、标准的对象实例；大多数对象都有自己对应的厂对象。关于这一点，后面将会举例说明。

注意 在 COM/DCOM 中，也有与厂对象 Factory 相似的概念，就是类工厂 IclassFactory。所有的类都拥有自己相应的类工厂，类工厂负责创建对象实例。各个对象的类工厂通常应该由程序员负责实现。不过，由于其代码的相似性，往往可以简化任务，甚至自动完成有关编码。

8.1.2 厂定位对象 FactoryFinder

在我们眼中，对象实例的位置可能只有两种情况：远程或本地。但是，对 CORBA 而言，位置的含义就广泛了：不同位置可能拥有完全不同的平台、操作系统。比如，某个位置可能仅仅具备存储介质（数据存储公司的电子仓库），另外某个位置可能仅仅有图象、声音显示装置（电视机、简单的商务手机），还有某个位置却具有一些专用的附属设备（商业 POS 机、医疗设备）。当需要在这些位置创建同一对象实现的实例时，显然需要不同的厂对象。

在 CORBA 中，为了解决上述问题，特地引入了厂定位对象 FactoryFinder。厂定位对象 FactoryFinder 由 CORBA 系统管理员配置，表明在不同的位置创建对象实例时，应该使用哪种对应的厂对象。

采用厂定位对象 FactoryFinder 的最大好处有两个：

对象接口定义可以不随对象实例的位置不同而改变

需要将对象引入新的位置（工作环境）时，只需修改厂定位对象 FactoryFinder 的配置，添加新的厂对象，无需变动以前的厂对象。

当然，如果，我们没有涉及到如此复杂的工作环境，仅仅为厂定位对象 FactoryFinder 配置一个厂对象也是完全可以的。

在厂定位对象 FactoryFinder 中，有一个 find_factories 函数，可以用来获取某个或某些厂对象，其定义如下：

例 8.2 厂定位对象 FactoryFinder 中的 find_factories 函数。

```
//这是厂定位对象 FactoryFinder 中的 find_factories 函数
interface FactoryFinder{
        ……
        Factories find_factories(
                        in Key factory_key
                        )
                        raises(no_factory);
        ……
};
```

其中，factory_key 可以从对象命名服务中获得。调用本函数后，可能返回与 factory_key 相符合的一个或若干厂对象，再通过这些厂对象创建所需的对象实例，也可能引发异常，说明找不到有关厂对象。

8.1.3 对象生存期服务的若干接口定义

对象生存期服务是由 OMG IDL 定义的一组操作。如同前面所指出的，用户自己定义的对象即可以从 CORBA 规定对象中继承这些标准定义，也可以自己定义与标准兼容的接口。CORBA 中有关的接口在对象 LifeCycleObject 中定义，如下所示：

例 8.3 对象生存期服务定义。

```
//这是对象生存期服务定义
interface LifeCycleObject{
        ……
        LifeCycleObject copy(
                        in FactoryFinder there,
                        in Criteria the_critera
                        )
                        raises(NoFactory,NotCopyable,
                                InvalidCriteria,CannotMeetCriteria);
        void remove() raises(NotRemovable);
        void move(
            in FactoryFinder there,
            in Criteria the_critera
            )
            raises(NoFactory,NotCopyable,
                        InvalidCriteria,CannotMeetCriteria);
        ……
};
```

在拷贝对象时，copy 函数首先激发厂定位对象 FactoryFinder 中的 find_factories 函数，获得相关对象的厂对象，然后激发厂对象中的 create 函数，在目的位置创建一个对象实例并返回其对象引用，拷贝策略由 the_critera 决定，稍后再详细说明。

当删除对象时，仅仅需要激发对象的 remove 即可。

至于移动对象，往往可以看作先拷贝，再删除源对象。

在接口中，同时定义了操作执行中可能被引发的各种异常。

如果用户希望某些对象具备 CORBA 生存期服务，就可以继承 LifeCycleObject 接口，也可以在对象接口中直接定义有关操作，不过，编码实现这些接口是软件开发人员的义务。

8.2 对象关系服务

与现实世界相似，对象之间也存在复杂的关系，比如：

一个顾客对象可能“拥有”一个或多个带有维修编号的商品对象

一个顾客对象可能“隶属”于某个消费组织对象

一个顾客对象可能需要定期从某个公司销售快讯服务对象中“获取”某类商品对象的有关信息。

某个公司对象中可能“包括”一个或多个部门对象

某个部门对象中可能“引用”了某些服务对象，以提供不同服务

恐怕没有人能够穷尽对象之间所有的关系。在 CORBA 中，为了给千遍万化的对象关系提供一些统一的处理方式，将“关系”本身也用对象来表示，定义了包括角色对象 Role、关系对象 Relationship、节点对象 Node 及其厂对象等一系列对象，形成 CORBA 对象关系服务。

不过，在讨论这些对象之前，让我们先探讨一下在 CORBA 中如何判断辨别一个对象实例。

8.2.1 辨别对象实例

我们曾经说明，CORBA 客户通过对象引用来调度不同的对象实例，不过，在不同 CORBA 域之中，同一个对象实现可能表现为对象引用及互操作对象引用 IOR。那么，这些对象引用及互操作对象引用是否等价？

如何回答这个问题取决于用户关心什么。如果用户仅仅需要通过对象引用或者互操作对象引用激发对象实例提供的各种操作，这些对象引用及互操作对象引用是等价的；如果用户是网络管理人员，需要考虑如何配置 Intranet、Extranet、桥接及防火墙，则对象引用与互操作对象引用是来源不同的对象，需要分别处理，不能等价观之。

为了妥善解决上述辨别对象实例问题，CORBA 中定义了 IdentifiableObject 公共对象接口，继承并实现该接口的对象可以按照统一的方式辨别对象实例等价问题。

IdentifiableObject 接口包含两个元素：

只读属性 Object_Identifier。本属性返回对象的标志号，可以用来与另外一些 IdentifiableObject 对象标志号进行比较。如果不同 IdentifiableObject 对象的标志号不同，则这些对象绝对不等价；如果不同 IdentifiableObject 对象的标志号相同，则这些对象可能等价，最终由用户根据需要决定。

函数 is_identical。本函数以某个对象的对象引用为参数，检查本对象引用与指定为参数的对象引用是否等价，如果等价则返回 true，否则返回 false。当然，如何判断对象引用是否等价由用户具体编程实现。

如果在对象运行过程中需要辨别对象实例是否等价，可以让有关对象继承 IdentifiableObject 接口。但是，有关代码应该由对象开发人员编写实现。

8.2.2 角色对象 Role 及关系对象 Relationship

在面向对象领域，每一种关系都由若干角色构成，每个角色都由一个或多个对象承担。因此，当我们抽象一种关系时，需要注意以下问题：

关系的类型（Type）。构成关系的对象及关系对象本身都有类型。比如，“拥有”、“隶属”、“包括”、“引用”都对应于不同的关系类型。

关系所涉及的角色（Role）。在关系中，对象实例都承担一定的角色，角色也可以由相同类型的不同对象实例承担，相同对象实例还可以承担不同关系中的不同角色。比如，“隶属”关系中有“所有者”、“所有物”两个角色，可以由多种对象实例承担；一个顾客对象实例既可以承担“拥有”关系中的“拥有者”角色，也可以承担“隶属”关系中的“所有物”角色。

关系的度（Degree）。关系的度就是关系中涉及的角色数目。“拥有”关系的度通常为 2，对应于“拥有者”、“拥有物”两个角色；“获取”关系的度通常为 3，对应于

“提出要求者”、“所需物品或服务”、“查找范围”三个角色。

关系的基（Cardinality）。关系的基就是一个角色所能够被卷入的不同关系对象的最大值。比如，在“拥有”关系中，同一个“拥有者”角色可以拥有多个物品，因而也可以被卷入多个“拥有”关系对象。

关系的语义（Semantics）。关系的语义定义了与关系有关的属性、操作。比如，当顾客希望从某个公司的销售快讯服务中定期获取扫描仪的信息时，还需要指定传递这些信息的渠道，是邮寄、E-mail、传真还是电话通知？而这个信息传递渠道属性最好归属于“获取”关系对象，以便于扩展。

最简单的 CORBA 对象关系服务由角色对象 Role、关系对象 Relationship 及其对应的厂对象实现。

角色对象 Role 由角色厂对象 RoleFactory 创建，RoleFactory 定义如下。

例 8.4 RoleFactory 对象的定义。

```
//这是 RoleFactory 对象的定义
interface RoleFactory{
        exception……;

        readonly attribute InterfaceDef role_type;
        ……

        Role create_role(
                in RelatedObject related_object
                )
                raises(NoRelatedObject,RelatedObjectError);
        ……
};
```

每个角色厂对象仅仅能创建一种类型的角色，只读属性 role_type 为进一步创建关系对象提供了方便。在创建角色对象时，应该为每个角色分配相应的对象实例，以便角色厂对象检查对象实例是否是能够承担对应角色的类型。

角色对象 Role 的接口定义如下所示：

例 8.5 角色对象 Role 的接口定义。

```
//这是角色对象 Role 的接口定义
interface Role{
        exception ……;

        readonly attribute RelatedObject related_object;

        RelatedObject get_other_related_object(
                        in RelationshipHandle rel,
                        in RoleName target_name
                        )
                        raises(……);
        Role get_other_role(
                        in RelationshipHandle rel,
                        in RoleName target_name
                        )
                        raises(……);
        void get_relationships(
                        in unsigned long how_many,
                        out RelationshipHandles rels,
```

```
                                out RelationshipIterator iterator);
        void link(
            in RelationshipHandle rel,
            in NamedRoles named_roles,
            )
            raises(……);
        void unlink(in RelationshipHandle rel) raises(……);
        ……
};
```

在创建角色对象时，角色厂对象会将有关的对象实例分配给 RelatedObject，通过该只读属性，用户可以了解承担角色的对象实例。

如果希望获得某个关系对象中涉及的角色对象或者角色承担对象实例，可以调用 get_other_role、get_other_related_object。由于一个角色有可能出现在多个关系对象中，我们还可以通过调用 get_relationships 获得角色被卷入的所有关系。

而 link 及 unlink 主要用来建立、解除角色对象与关系对象之间的连接。

关系厂对象 RelationshipFactory 的接口定义如下所示：

例 8.6 关系厂对象 RelationshipFactory 的接口定义。

```
//这是关系厂对象 RelationshipFactory 的接口定义
struct NamedRole{
        RoleName name;
        Role aRole;
};

typedef sequence<NamedRole> NamedRoles;

interface RelationshipFactory{
        ……
        struct NamedRoleType{
                RoleName name;
                InterfaceDef named_role_type;
        };

        typedef sequence<NamedRoleType> NamedRoleTypes;

        exception……;

        readonly attribute InterfaceDef relationship_type;
        readonly attribute unsigned short degree;
        readonly attribute NamedRoleTypes named_role_types;

        Relationship create(
                        in NamedRoles named_roles
                        )
                        raises(RoleTypeError,
                        MaxCardinalityExceeded,
                        DegreeError,
                        DuplicateRoleName,
                        UnknownRoleName
                        );
        ……
```

```
};
```

在创建完有关角色对象后，才可以创建关系对象，因为关系厂对象需要将所有角色对象作为输入参数使用。在创建关系对象过程中，关系厂对象应该调用每个角色对象的 link 函数，将这些角色对象“卷入”新创建的关系对象。

关系对象 Relationship 本身的定义如下所示：

例 8.7 关系对象 Relationship 的定义。

```
//这是关系对象 Relationship 的定义
interface Relationship : IdentifiableObject{
        ……
        readonly attribute NamedRoles named_roles;

        void destroy() raises (CannotUnlink);
};
```

关系厂对象创建关系对象时输入的参数记录在 named_roles。当用户释放关系对象时，可以调用 destroy 函数，在该函数中，应该调用有关角色对象中的 unlink 函数。

采用角色对象、关系对象表示对象之间的关系，具有下列优点：

在开发对象时，无需考虑对象之间的关系，简化对象开发

在处理对象之间的关系时，可以从关系、角色对象直接入手，无需考虑作为角色承担者的对象实例，加速关系处理过程。

当然，这种方式也有一定的缺点，比如：

我们可以从关系对象中查询有关的角色对象及角色承担者，也可以从角色对象中查询角色承担者及涉及的关系对象；但是，我们无法从对象实例中查询它所承担的角色，被卷入的关系。

当一个对象实例承担了多个关系中的不同角色时，无法遍历所有关系。比如，一个顾客实例可以“拥有”多个商品，同时，这个顾客实例又可以“隶属”于一个消费团体。在这种方式中，我们无法从顾客与商品的“拥有”关系中进入顾客与消费团体的“隶属”关系。

为了解决第二个缺点，我们需要引入新的对象，节点对象 Node。

8.2.3 节点对象 Node

如图 8.1 所示，通过节点对象 Node，可以形成对象的关系图。

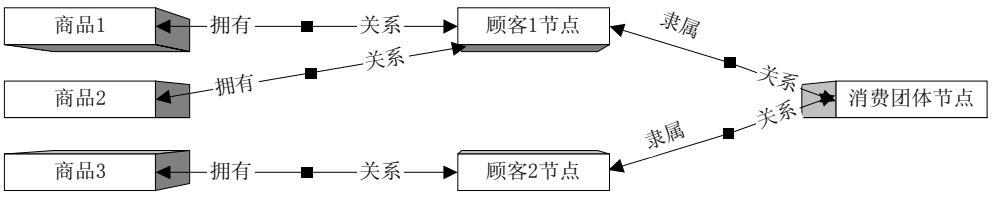

图 8.1 顾客、消费团体、商品之间的关系图

每个节点对象 Node 可以记录某个对象实例所卷入的所有角色。比如，在上图中，顾客对象、消费团体对象都用单独的节点对象表示；有两个顾客对象“隶属”于消费团体对象，其中一个顾客“拥有”两样商品，另一个“拥有”一样商品。

我们还可以通过 CORBA 中规定的 Traversal 接口遍历由节点对象 Node 衔接而成的关系图。比如，在上述的关系图中，我们可以从消费者对象实例获得商品 1、商品 2 及商品 3；也可以从商品 3 获得顾客对象实例 1。

同样，节点对象 Node 也是由其对应的节点厂对象 NodeFactory 创建，而 NodeFactory

的接口定义如下所示：

例 8.8 节点厂对象 NodeFactory 的接口定义。

```
//这是节点厂对象 NodeFactory 的接口定义
interface NodeFactory{
        Node create_node(
                        in Object related_object
                        );
};
```

在创建节点对象时

，也需要为节点分配对应的对象实例。

节点对象 Node 的接口定义如下所示：

例 8.9 节点对象 Node 的接口定义。

```
//这是节点对象 Node 的接口定义
interface Node : IdentifiableObject{
        ……
        typedef sequence<Role> Roles;

        exception……;

        readonly attribute RelatedObject related_object;
        readonly attribute Roles roles_of_node;

        Roles roles_of_type(in InterfaceDef role_type);
        void add_role(in Role a_role) raises(DuplicateRoleType);
        void remove_role(
                        in InterfaceDef role_type
                        )
                        raises(NoSuchRole);
        ……
};
```

用户有两种方式使用节点对象 Node：

让有关对象继承节点对象 Node 的接口定义，这时，每个对象实例本身就是一个节点，可以调用 add_role、remove_role 增加或删减对象实例被卷入的角色、关系，调用 roles_of_type 查询有关角色。

为每个对象实例创建一个节点对象，作为其代理，负责查询、增加或删减对象实例被卷入的角色、关系。

通过节点对象 Node 及 Traversal 接口，我们可以设计不同的策略，用来拷贝、移动或删除某个对象实例及与它具备某些关系的对象实例。

8.2.4 混合对象的处理策略

对象实例往往并不单独存在，通过本身具备的节点对象特性或者专门的节点对象 Node，它们构筑成一个通过关系衔接的图，这时，我们可以把有关对象称为混合对象。当我们希望拷贝、移动或删除混合对象之中的某个实例时，如何处理与它具有关系的对象？为了满足不同需要，CORBA 通过 Criteria 参数从下列四种方案中选择进行：

deep：说明此次操作对相关的关系对象、角色对象以及所有角色承担者的对象实例都有效。

shallow：说明此次操作仅对关系对象、指定角色对象及对象实例有效。比如，我们采

用本策略拷贝对象时，源对象将被拷贝，与其相关的关系对象也会被拷贝，不过，新的关系对象中仅有一个角色由源对象的新拷贝承担，其余角色都由原来的对象实例承担。

none：说明此次操作仅对指定的对象实例有效，不必考虑它所卷入的关系、角色。

inhibit：说明此次操作不考虑与对象实例有关的节点对象。比如，当我们试图释放一个对象实例时，就可以通过本参数确保节点对象不被释放。

选定参数后，遍历过程将根据操作有效范围选择一系列的对象实例、节点对象、角色对象、关系对象，在清除重复引用后，按照一定的顺序进行有关操作。至于遍历算法，由 Traversal 接口定义、实现，这里不再赘述。

在 CORBA 中，还规定了一些常用的关系对象，包括“包容”关系及“引用”关系，它们分别由 Containment 及 Reference 对象描述。

8.3 持续对象服务

对象在运行过程中的有关取值就是对象的状态。一般情况下，对象的状态有两种情况：

动态状态，主要保存在内存中，一旦程序结束或者机器关闭，就会消失

持续状态，主要保存在硬盘等存储介质中，状态永久有效。

持续对象服务（Persistent Object Service）用来管理、存储、恢复对象的持续状态，主要由下列对象构成：

持续对象 PO（Persistent Object）。一个对象可以通过继承、实现 PO 接口定义成为持续对象。不论何时，持续对象都能够保持、恢复自己的状态。而 PO 接口中定义了 connect、disconnect、store、restore、delete 五个操作，通过调用这些函数，客户可以控制持续对象与其持续数据之间的关系。

持续标志号 PID（Persistent Identifier）。持续标志号仅仅在表示持续对象被存储的状态数据时有效。PID 也可以被转化为字符串形式，以便存储及日后使用。但是，与对象引用不同，PID 不能用来激发对象实例。我们可以把 PID 想象为持续对象的“档案号码”。

持续对象管理器 POM（Persistent Object Manger）。持续对象管理器将持续对象与具体的对象状态数据存储服务连接起来。一个持续对象对应一个持续对象管理器对象。持续对象管理器对象将持续对象与其状态存储方式隔离，其接口中定义也定义了 connect、disconnect、store、restore、delete 五个操作，持续对象可以使用这些函数与持续数据服务对象 PDS 通信。

持续数据服务 PDS（Persistent Data Service）。PDS 负责在特定数据存储器、对象之间传递数据。PDS 接口中也定义了 connect、disconnect、store、restore、delete 五个操作，用来与特定存储器交互。另外，PDS 还必须支持某种协议，以便 POM 从存储器中提取对象的状态数据。目前，有三种协议可以选择：直接访问协议 DA（Direct Access）、ODMG-93 协议以及动态数据对象协议 DDO（Dynamic Data Object）。DA 协议通过对象属性的方式直接读取对象的各个状态数据；ODMG-93 协议支持 OODB；DDO 协议允许使用 PID 进行任何方式的存储，如数据库等。

数据存储器（Datastore）。数据存储器可以是流文件、任何格式的数据文件，也可以是数据库系统。数据存储器的接口与 PDS 使用的协议有关。

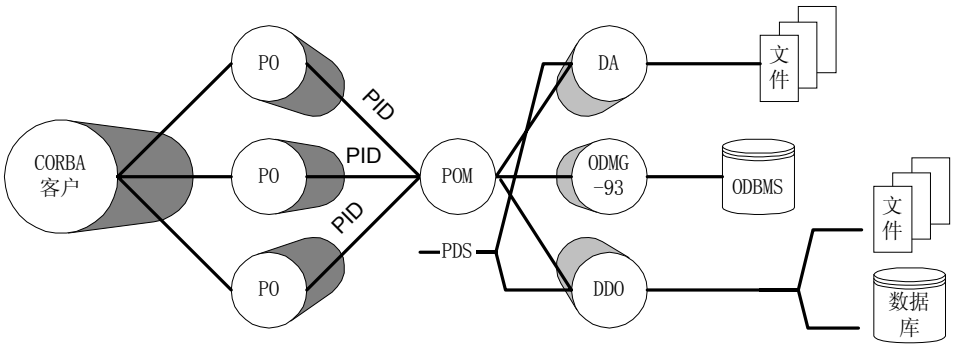

图 8.2　持续对象服务的结构

通过以上对象，持续对象服务形成了如图 8.2 所示的结构。

使用持续对象服务时，可以首先调用 connection 函数，将对象的动态状态与持续状态数据联系起来。当上述联系建立后，对象的动态状态与持续状态始终保持一致，直到用 disconnect 结束联系。

当然，用户也可以调用 restore 函数，输入 PID 及对象引用参数，恢复对象的状态。而这个状态数据往往又是先前调用 store 函数保存的。store 函数的参数也是 PID 及有关对象引用。

8.4　对象外化服务

对象外化服务（Externalization Service）是和对象内化服务（Internalization Service）成对出现的。它们的目的就是提供一种称为“流”的机制，将对象的状态提取出来，外化为一个“流”；用户可以传播、保存这个“流”对象，然后，在合适的时候，再创建一个对象实例，并从相关的“流”中将对象的状态数据恢复，内化一个对象实例。

能够进行外化、内化的对象首先必须是可流对象。可流对象的接口定义如下所示：

例 8.10 可流对象 Streamable 的接口定义。

```
//这是可流对象 Streamable 的接口定义
interface Streamable :: IdentifiableObject{
        ……
        readonly attribute Key exteranl_form_id;

        void externalize_to_stream(in StreamIO targetStreamIO);
        void internalize_from_stream(
                                in StreamIO SourceStreamIO,
                                in FactoryFinder there
                                )
                                raises(……);
        ……
};
```

当外化一个可流对象时，需要指明用来保存可流对象状态的流对象 targetStreamIO；当需要内化一个可流对象时，除了需要指明存储着状态数据的流对象 SourceStreamIO 外，还应该提供与可流对象有关的厂定位对象 FactoryFinder。

至于流对象，可以由流厂对象 StreamFactory 或文件流厂对象 FileStreamFactory 创建。StreamFactory 的接口定义如下所示：

例 8.11 流厂对象 StreamFactory 的接口定义。

```
//这是流厂对象 StreamFactory 的接口定义
interface StreamFactory{
        Stream create();
};
```

文件流厂对象 FileStreamFactory 的接口定义如下所示：

例 8.12 文件流厂对象 FileStreamFactory 的接口定义。

```
//这是文件流厂对象 FileStreamFactory 的接口定义
interface FileStreamFactory{
        Stream create(in string theFileName)
                Raises(InvalidFileNameError);
};
```

上述厂对象可以创建相应的流对象；其中，文件流厂对象 FileStreamFactory 允许流对象以文件形式存在。无论如何，流对象支持 externalize、internalize、begin_context、end_context、flush 五个函数，并且支持 StreamIO 对象中规定的 write_xxxx、read_xxxx 操作。其中，xxxx 表示 graph、object、string、char 等数据类型。

显然，用户也可以通过对象外化服务来拷贝、移动对象实例。

8.5 对象命名服务

对象命名服务（Naming Service）就是给对象实例提供一个名称，以便用户通过这些名称获取对象实例。一般情况下，用户还可以规定对象的命名原则，借以简单表示厂家、所在主机、功能等等一系列信息。

用来命名对象的原则千变万化。比如，我们可以采用将国家、机构、部门、开发者、对象组件名称等信息依次组合的方式来命名一个对象实例；也可以采用将域名、主机名、工程名、对象组件名称等信息依次组合的方式来命名一个对象实例。在区别各个层次的“子名称”时，我们可以使用“\”分割，也可以采用“;”分割。

为了有效解决这些问题，CORBA 命名服务通过名称对象 Name 来表示对象实例的名称，名称对象 Name 的定义如下所示：

例 8.13 名称对象 Name 的定义。

```
//这是名称对象 Name 的定义
typedef string Istring;

struct NamedComponent{
        Istring id;
        Istring kind;
};

typedef sequence<NamedComponent> Name;
```

名称对象 Name 实际上是一个由名称构件 NamedComponent 组成的序列。名称构件 NamedComponent 本身包括两个元素：一个 id，用来表示不同层次的子名称；一个 kind，用来表示本层次子名称的类别，而 id 及 kind 都是字符串。

通过名称对象 Name，用户可以随意命名对象实例，只需要将子名称及子名称的类别附于名称构件 NamedComponent 即可。

例 8.14 一个名称对象实例。

```
//这是一个名称对象实例
```

```
        ……
CosNaming::Name name;
Name.length(4);
Name[0].id = “中国”;
Name[0].kind = “国家名称”;
Name[1].id = “WinMagic”;
Name[1].kind = “公司名称”;
Name[2].id = “E-CommerceBase”;
Name[2].kind = “工程名称”;
Name[3].id = “TClient”;
Name[3].kind = “组件名称”;
        ……
```

CosNaming 是定义命名服务有关对象的模块，其中，还包括命名上下文对象 NamingContext、绑定迭代器 BindingIterator。

我们可以调用命名上下文对象 NamingContext 中的有关操作将命名对象 Name 与具体对象实例绑定，这些操作定义如下：

例 8.15NamingContext 的绑定操作。

```
//这是 NamingContext 的绑定操作定义
interface NamingContext{
        ……
        void bind(
            in Name n,
            in Object obj
            )raises(NotFound,CannotProceed,
                InvalidName,AlreadyBound);
        void rebind(
            in Name n,
            in Object obj
            )raises(NotFound,CannotProceed,InvalidName);
        void bind_context(
            in Name n,
            in NamingContext nc
            )raises(NotFound,CannotProceed,
                    InvalidName,AlreadyBound);
        void rebind_context(
            in Name n,
            in NamingContext nc
            )raises(NotFound,CannotProceed,InvalidName);
        ……
};
```

当名称对象 Name 由一个名称构件 NameComponent 组成时，调用 bind 或 rebind 函数后，名称对象 Name 直接与指定的对象实例绑定；当名称对象 Name 由多个名称构件 NameComponent 组成时（被称为混合名称），调用 bind 或 rebind 函数后，仅有最后一个名称构件 NameComponent 与指定对象实例绑定，其余名称构件都自动与其他命名上下文对象 NamingContext 绑定。因为在碰到 ctx->bind(<c_1;c_2;c_3…c_n>,obj)类的语法时，NamingContext 会自动递归理解为(ctx->resolve(<c_1;c_2;c_3…c_{n-1}>))->bind(<c_n >,obj)。

bind_context 与 rebind_context 可以将一个名称对象 Name 与一个名称上下文对象绑定，这样，我们可以对原有命名进行必要的改造。比如，当我们希望在“中国”子名称前再加上

一个“亚洲”子名称时，只需将定义为“亚洲”的名称对象与原来的上下文对象绑定即可。

通过命名上下文对象中的解析函数，我们可以获得一个名称对象所对应的对象实例。该解析函数的定义如下：

例 8.16 命名上下文对象中解析函数的定义。

```
//这是命名上下文对象中解析函数的定义
interface NamingContext{
        ……
        Object resolve(in Name n)
                Raises(NotFound,CannotProceed,InvalidName);
        ……
};
```

该函数返回 Object 类型的对象引用，用户可以进一步调用 narrow 操作将其转化为适当类型的对象实例引用。当输入参数为混合名称对象时，函数也以递归方式进行处理，也就是说，ctx->resolve(<c_1;c_2…c_n>)等于 ctx->resolve(<c_1;c_2…c_{n-1}>)-> resolve(<c_n>)。

在命名上下文对象中，还定义了一些其它功能的操作。

例 8.17 命名上下文对象中其它操作的定义。

```
//这是命名上下文对象中其它操作的定义
interface NamingContext{
        ……
        void unbind(in Name n)
                Raises(NotFound,CannotProceed,InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context(in Name)
                        Raises(NotFound,AlreadyBound,
                                CannotProceed,InvalidName);
        void destory() raises(NotEmpty);

        enum BindingType{object,ncontext};
        struct Binding{
                Name binding_name;
                BindingType binding_type;
        };
        typedef sequence<Binding> BindingList;
        void list(
             in unsigned long how_many,
             out BindingList bl,
             out BindingIterator bi
             );
        ……
};
```

其中，unbind 用来解除已经定义的绑定；new_context 创建一个还未绑定的命名上下文对象；bind_new_context 创建并绑定一个命名上下文对象；destory 删除一个无用的命名上下文对象；list 用来列表与本命名上下文对象有关的绑定，列表的数目由 how_many 决定，而指定数目的绑定将记录在 bl 中，其余部分通过 bi 输出（可能为空）。

列表函数使用的 bi 参数为绑定迭代器 BindingIterator 类型，定义如下：

例 8.18 绑定迭代器 BindingIterator 的定义。

```
//这是绑定迭代器 BindingIterator 的定义
interface BindingIterator{
```

```
        boolean next_one(out Binding b);
        boolean next_n(
                in unsigned long how_many,
                out BindingList bl
                );
        void destory();
};
```

用户可以调用 next_one 返回下一个绑定，也可以调用 next_n，返回下 n 个绑定，具体数目由 how_many 决定。最后，还可以调用 destory 释放无用的绑定迭代器。

8.6 对象洽谈服务

实际上，我们可以把对象洽谈服务（Trader Service）看作一个供 CORBA 对象开发者与 CORBA 客户“买卖”CORBA 对象的 “电子商务平台”。对象开发者通过对象洽谈服务提供的程序接口或工具注册、说明、促销自己开发的对象；客户通过对象洽谈服务提供的工具查询、搜索、获取并使用对象。

一般情况下，对象开发者/客户可以通过对象洽谈服务提供/获取以下内容：

对象引用。洽谈成功的结果就表现在客户同意引用对象开发者极力推荐的对象产品的对象实例。

服务类型说明。服务类型说明可以包括服务涉及的对象接口、操作说明，操作、函数中有关参数和返回结果说明，服务实现算法、操作过程说明等等。由于洽谈服务主要是与“人”交流，因此可以不采用 OMG IDL。

服务特性说明。服务特性说明允许对象开发者以“值-名体”来描述服务的特性。有些服务特性是对象开发者必须实现的，有些服务特性可以由对象开发者额外提供。服务特性说明的方式也非常简单，用特性名称与特性值填充一个“值-名体”即可。

客户可以使用服务查询对象 Lookup、供给迭代器 OfferIterator 从洽谈服务中获取有关的信息、对象引用。查询对象 Lookup 中定义了一个 Query 操作，允许用户输入服务类型作为查询参数，并可以设置希望满足的查询策略、约束条件及优先权。供给迭代器 OfferIterator 作为查询对象 Lookup 的返回结果之一，允许用户调用 next_n 返回下 n 个服务提供者。

另一方面，对象开发者可以调用服务注册对象 Register 发布各种信息。将对象引用、服务类型说明、服务特性说明通过 export 注册后，每个对象引用获得一个唯一的供给者标志号 OfferID。如果希望删除、修改服务供给者对应的对象实例，可以调用 withdraw、withdraw_using_constrait、modify 进行；如果希望获得服务类型说明、服务特性说明，可以调用 describe 完成；如果希望得到服务供给者对应的对象实例的引用，可以调用 resolve 进行。

显然，对象开发者不能在所有的洽谈服务器上注册自己的对象产品，但是却可能经常需要修改已经注册的对象信息（比如，提供当日特色服务）。因此，洽谈服务中特地定义了连接对象 Link、洽谈代理对象 Proxy 以及动态属性辩识对象 DynamicPropEval。连接对象 Link 用来共享不同洽谈服务器上的信息，必要的时候，可以通过洽谈代理对象 Proxy 最终获得不在本洽谈服务器上注册的服务供给者的对象引用。用户还可以使用动态属性辩识对象 DynamicPropEval 中的 evalDP 函数，获取与服务供给对象有关的动态属性信息。

洽谈服务器将服务类型说明存储在服务类型仓库 ServiceTypeRepository 中，并通过统一的管理接口 Admin 进行管理。至于如何保存对象引用及服务属性说明，CORBA 并未作出具体规定。

在实际引用中，对象开发者及客户往往通过各种具有图形功能的工具使用对象洽谈服务，就如同我们通过浏览器浏览各种黄页、订购商品一样。另外，通过动态激发接口 DII，

我们可以从对象洽谈服务中获取最具价值的对象，快速集成一个商务软件，完全可以适应电子商务的瞬息万变。

8.7 事件服务

分布式对象非常了解“有所为有所不为”的处事原则，它们并不关心所有的事件。但是，我们必须允许对象声明它们关心什么事件；也应该允许对象撤消对某些事件的关心。

比如，当我们实例化一个商品对象时，规定了该商品的零售价、批发价，在经营过程中，可能经常需要改变这些价格。该商品的价格变化可以被一些电子商务平台中的促销对象关心，也可以被一些电子商务平台中的管理决策对象关心，还可以被一些电子商务平台中的税收对象关心……当我们开发商品对象时，并不知道它的调价事件会被哪些对象关心。某个促销对象也可能因为“改行”而不再关心某些商品的调价事件。为了处理上述的类似情况，CORBA 引进了事件服务（Event Service）。

一个对象可以通过两种方式向“关心”它的其它对象发送“事件被激发的消息”：

一旦事件发生，主动向有关对象发送消息。在 CORBA 中，本方式被称为推（Push）方式。

发生事件后，在被有关对象询问时再发送消息。在 CORBA 中，本方式被称为拉（Pull）方式。

与之相似，对象也可以通过两种方式获取它所关心的其它对象的有关“事件消息”：

被自动告知所关心的事件已经发生，在 CORBA 中，这是推（Push）方式的作用结果。

主动向有关对象查询所关心的事件，在 CORBA 中，这是拉（Pull）方式的作用结果。

当用户希望对象之间采用上述方式进行“事件消息”传递时，可以让 CORBA 对象继承并实现一些规定的接口，这些接口包括推型产生者 PushSupplier、推型接收者 PushConsumer、拉型产生者 PullSupplier 以及拉型接收者 PullConsumer。

推型产生者 PushSupplier 的接口如下所示：

例 8.19 推型产生者 PushSupplier 的接口定义。

```
//这是推型产生者 PushSupplier 的接口定义
interface PushSupplier{
        void disconnect_push_supplier();
};
```

该接口中定义了一个函数 disconnect_push_supplier，用来结束与本事件生产对象有关的“事件消息”传递，它将释放本对象中用来传递“事件被激发消息”的所有资源。

推型接收者 PushConsumer 的接口如下所示：

例 8.20 推型接收者 PushConsumer 的接口定义。

```
//这是推型接收者 PushConsumer 的接口定义
interface PushConsumer{
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
};
```

该接口中，push 函数用来接收有关的“事件消息”；disconnect_push_consumer 用来结束与本事件接收对象有关的“事件消息”传递，它将释放本对象中用来传递“事件被激发消息”的所有资源。

拉型产生者 PullSupplier 的接口如下所示：

例 8.21 拉型产生者 PullSupplier 的接口定义。

```
//这是拉型产生者 PullSupplier 的接口定义
```

```
interface PullSupplier{
        any pull() raises(Disconnected);
        any try_pull(out boolean has_event) raises(Disconnected);
        void disconnect_pull_supplier();
};
```

该接口中，pull 函数将以阻塞的方式进行，直到“事件消息”准备就绪或者发生异常为止；try_pull 函数将以非阻塞方式进行，如果“事件消息”已经准备好，has_event 返回 true，“事件消息”也作为调用结果返回，否则，has_event 返回 false；disconnect_pull_supplier 函数则用来结束与本事件生产对象有关的“事件消息”传递。

拉型接收者 PullConsumer 的接口如下所示：

例 8.22 拉型接收者 PullConsumer 的接口定义。

```
//这是拉型接收者 PullConsumer 的接口定义
interface PullConsumer{
        void disconnect_pull_consumer();
};
```

该接口中定义了一个函数 disconnect_push_consumer 用来结束与本事件接收对象有关的“事件消息”传递。

显然，对象开发者不会愿意去维护一个随时变动的“关注者”名单；对象使用者也不会在意其它对象对自己关心的对象的关心程度。因此，事件服务必需由独立于事件产生者、事件接收者对象实现之间的其它对象进行，这些对象包括事件信道对象 EventChannel、产生者管理对象 SupplierAdmin、接收者管理对象 ConsumerAdmin 以及各种代理对象。

事件信道 EventChannel 可以以多种方式实现事件产生者与事件接收者之间的通信，包括以下各种形式：

单一的“Push”类型通信。事件产生者将事件信息“Push”给事件信道对象，事件信道对象将事件信息继而“Push”给事件接收者。

单一的“Pull”类型通信。事件接收者向事件信道对象查询有关事件信息，事件信道对象继而向事件产生者查询有关事件信息，既可以阻塞进行，也可以非阻塞进行。

混合类型的通信。事件信道对象与事件产生者及事件接收者以不同的方式进行通信。比如，事件产生者可以把事件信息“Push”给事件信道对象，事件信道对象继而等着事件接收者前来查询、“Pull”有关事件信息。

多对多通信。事件信道并不被一组事件产生者及事件接收者单独占用，相反，一个事件产生者可以通过事件信道通知一组事件接收者；一个事件接收者也可以通过事件信道接收多种事件数据。

事件信道 EventChannel 既允许以 any 来传递通用的事件信息，也允许传递以 OMG IDL 定义的类型化事件信息。有关内容不再赘述。

8.8 事务服务

“一手交钱，一手交货”是交易的基本原则，它的等价命题是既不能提了货却不给钱，又不能给了钱却没提货。在现实生活中，有许多对象之间的操作存在这种类似关系：要么这些操作全都进行，要么这些操作全都不进行，这种关系的操作就构成了事务（Transaction）。

8.8.1 ACID——事务的特点

事务是构筑稳定、强健的分布式程序的根本保证。事务处理涉及的一系列操作总能满足四个特点，这四个特点就是原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、

持久性（Durability）。通过提取每个特点的第一个字符，它们被简写为 ACID。ACID 的具体含义如下所示：

原子性（Atomicity）。所有的变化要么同时发生——committed，要么都不发生——rolled back。

一致性（Consistency）。相关对象的数据变化协调一致，符合设定的关系。比如，如果是转帐事务，则资金转入方获得的金额应该与资金转出方支付的金额一致（可能需要考虑手续费）。

隔离性（Isolation）。其余事务只能读取到某事务发生前以及某事务发生后的数据；在事务处理过程中，其余事务不关心也不能获取中间状态的数据。

持久性（Durability）。事务处理结果被永久保存在硬盘等存储介质上，一般不会丢失。

CORBA 中的事务服务实际上就是对象事务服务 OTS。

8.8.2 OTS 的目标

CORBA 的对象事务服务 OTS 希望到达以下目标：

支持多种事务模型。事务模型主要有单层（Flat）事务、嵌套（Nested）事务以及链式（Chained）事务。OTS 必需支持单层事务，可以有选择地支持嵌套事务。

允许 ORB 和非 ORB 应用程序、资源共同参与同一个事务服务。

与 X/Open 分布式事务处理模型（DTP）兼容，使 OTS 与 DTP 具备互操作性。

允许非面向对象的程序、资源参与事务服务。

支持涉及多个 ORB 的事务服务，包括单一事务单一 ORB、多个事务单一 ORB、单一事务多个 ORB 以及多个事务多个 ORB 四种情形。

灵活的事务传播控制途径。允许客户控制一个操作是否参与事务；允许用户继承实现具有事务服务功能的对象接口，也允许用户直接定义实现自己的事务服务接口。

支持事务服务监控器（TP Monitors）。事务监控器主要用来在多用户环境下规划事务，以利于资源共享及负载均衡。事务监控器具有并行处理多个事务以及为客户、对象实现、事务服务创建不同进程的能力。

8.8.3 可恢复型对象及事务型对象

并非所有对象都可以参与事务服务，只有具备可恢复型对象（Recoverable Object）或事务型对象（Transactional Object）特性的对象才可以参与事务服务。

可恢复型对象通常具备“失败-恢复”机制。比如，可恢复型对象往往将自己的状态数据存储在硬盘等介质上，如果在事务执行过程中发生异常，该对象可以在重新启动的进程中发现上一次事务执行失败的记录，进而从硬盘等介质中读取存储的数据，恢复执行有关事务前的稳定状态。另外，可恢复型对象还应该能够理解事务管理对象发出的 commit 或 roll back 消息，执行相应的操作。实际上，可恢复型对象本身的状态取决于事务完成情况。

事务型对象涉及的范围比可恢复型对象广，通常指在事务作用范围内有效操作所对应的所有对象。这个对象可能控制着某些可恢复型对象、使用了某些可恢复型对象的存储数据、调用了支持 commit 或 roll back 的数据库操作……同样，事务型对象也应该能够理解事务管理对象发出的 commit 或 roll back 消息。不过，它们的状态并不取决于事务的完成情况。

事务服务通常采用两段提交（two-phase commit）方式进行：客户指明事务的开始、结束位置，事务管理器向该事务作用范围内的所有操作（对象）发出询问：是否可以提交（commit）？如果所有操作（对象）都同意提交，整个事务就被提交；如果至少有一个操作（对象）要求滚回（roll back），整个事务就滚回。

8.8.4 事务上下文

在事务服务中，用户指明事务的开始、结束位置后，在事务作用范围内的每个操作都被默认为是该事务的一部分，并不需要用户特地强调。

但是，由于 CORBA 中并不直接与对象实例进行交互，而是通过 ORB 代理进行，所以，ORB 在激发对象操作时，必需明白、记住并传达这个操作是否是某个事务的一部分。也就是说，ORB 应该对“事务敏感”（transaction-aware）。

当 ORB 激发一个事务中的操作时，会自动创建一个事务上下文对象（transaction context），并把它作为参数传递给相关的操作。同时，可恢复型对象及事务型对象会主动查询事务上下文对象参数，检查本次操作是否被包括在某个事务服务中，如果是，又是哪个事务服务。

在上述方式中，事务中的操作被 ORB 自动记录、传递给对象实现，通常被称为隐含事务传播（implicit transaction propagation）。与之相对应，用户也可以主动在操作的参数中定义事务上下文对象，并通过激发这一类操作实现事务服务。这时，我们称之为显性事务传播（explicit transaction propagation）。

不过，无论采用隐含事务传播还是显性事务传播，对象实现都是通过激发操作时是否带有事务上下文对象作为判别事务作用范围的标志。

8.8.5 单层事务及嵌套事务

目前，CORBA 要求 OTS 供应商必须支持单层事务（flat transactions），但可以有选择地支持嵌套事务（nested transactions）。

单层事务非常简单，用户用规定的方式说明事务的开始，然后可以调度一系列操作，激发一系列可恢复型对象或事务型对象，最后，再以规定的方式说明事务的结束。一切都显得直来直去。

相比之下，嵌套事务就显得复杂多了：在一个事务内部，可以继续创建一个或多个子事务；而子事务中还可以包括更低级别的子事务。我们规定，直接或间接包含一个子事务的所有事务都是该子事务的父类事务（ancestors）；直接或间接包含于一个父类事务的所有子事务都是该父类事务的子类事务（descendants）。

嵌套事务的顶层（top-level）是没有任何父类事务的事务。实际上，单层事务就是一个不含任何子类事务的顶层事务。我们规定，顶层事务及其所有子类事务构成了事务系（transaction family）。

在事务系中，子事务并不具有持久性。比如，当子事务提交后，子事务中对象的状态依然“悬而未决”，直到该子事务的所有父类事务都提交（或滚回）后方可。因为，当某个子事务滚回后，它的所有子类事务都必须滚回。然而，子事务的滚回并不一定会影响到它的父类事务，具体情况视用户设置的失败粒度（fail granularity）而定。

8.8.6 OTS 的组成

图 8.3 说明了对象事务服务的各个组成部分及实现过程。

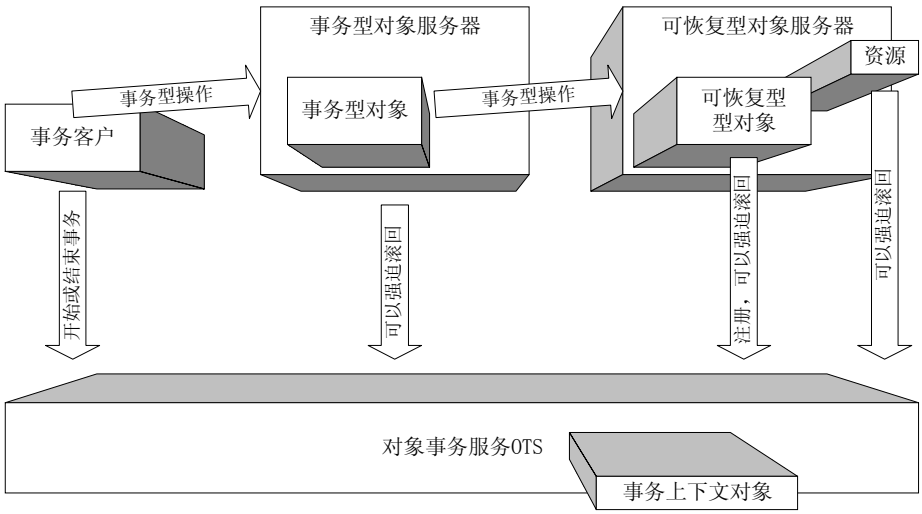

图 8.3　对象事务服务 OTS 的组成部分及实现过程

CORBA 客户首先通过规定的格式通知 OTS 开始或结束一个事务，然后就可以在事务作用范围内激发各种事务型对象及可恢复型对象的操作。OTS 会要求 ORB 建立、传播有关的事务上下文对象，并与事务型对象服务器及可恢复型对象服务器一同协作，完成事务服务。

事务型服务器是事务型对象的集合，这些对象本身没有独立可恢复资源或状态，而是通过引用、使用、控制别的可恢复型对象参与事务。事务型对象的最终状态与它所参与事务无关，但它可以强迫相关的事务滚回。

可恢复型对象服务器是可恢复型对象的集合，这些对象往往拥有一些需要保护的资源对象，如数据文件、队列或数据库等，本身也具备独立的可恢复状态。可恢复型对象通过向 OTS 服务器注册资源参与事务完成，它们及它们拥有的资源的最终状态与事务执行情况有关。当然，可恢复型对象、资源对象也可以强迫事务滚回。

OTS 的各个组成部分也是由对象构成的。

8.8.7　对象事务服务的接口

OTS 中定义的接口主要包括当前事务伪对象 Current、事务协调对象 Coordinator、资源对象 Resource、子事务敏感资源对象 SubtransactionAwareResource 等。这些接口的具体功能如下所示：

当前事务伪对象 Current。该伪对象用于客户启动、结束、控制一个事务服务。如果希望开始一个事务，可以调用伪对象的 begin 函数；如果希望结束一个事务，可以调用 commit 提交或 rollback 滚回；在事务执行过程中，还可以调用 suspend 挂起当前事务，再调用 resume 重新启动当前事务。另外，本伪对象中定义了 get_status、get_transaction_name、get_control、set_timeout 等其它控制函数。

事务协调对象 Coordinator。可恢复型对象、嵌套事务都需要利用事务协调对象 Coordinator 来完成事务服务。比如，register_resource 用来注册可恢复型对象需要保护的资源；register_subtran_aware、create_subtransaction 用来注册子事务资源、创建子事务；hash_transaction、hash_top_level_transaction 可以返回有关级别事务的句柄；get_transaction_name、get_status、get_parent_status、get_top_level_status 可以用来查询获取指定级别的事务的状态或名称；而 is_ancestor_transaction、is_descendant_transaction 、 is_top_level_transaction 、 is_same_transaction 、 is_related_transaction 函数都具备与其名称相称的功能。

资源对象 Resource。该接口由参与两段提交方式的可恢复型对象实现。在确认的第一阶段，事务协调对象将调用所有参与本事务的资源对象上的 prepare 函数，询问每个资源对象对最终提交事务的意见，返回值可以是 vote_commit、vote_rollback 或 vote_readonly；Coordinator 将根据情况决定调用资源对象的 commit 或 rollback 函数，最终提交或滚回事务。如果事务协调对象 Coordinator 中仅仅有一个注册资源，两段提交方式明显多余，这时，Coordinator 会激发资源对象的 commit_one_phase 函数，直接提交事务。

子事务敏感资源对象 SubtransactionAwareResource。本接口从资源对象派生而来，应该由具备嵌套事务处理能力的可恢复型对象实现。新增加的方法 commit_subtransaction、rollback_subtransaction 分别用来提交、滚回子事务。这时，子事务中涉及的资源应该通过 register_subtran_aware 向事务协调对象 Coordinator 注册。

事务基对象 TransactionObject。该对象中没有定义任何操作，不过，所有具有事务服务能力的对象都应该继承该接口，表明自己是一个支持事务服务的对象。

最后，ORB 还会在需要的时候建立、传播事务上下文对象 TransactionContext。

8.9 并行服务

并行服务（Concurrency Service）的目标非常简单：为分布式对象或分布式事务提供一种合理的机制，以便它们能够成功读取、修改共享资源，不发生读脏数据、丢失修改等并发类型的错误。

解决并发类型错误的一个方案就是锁定共享资源：当分布式对象或分布式事务希望使用某个共享资源时，首先需要申请对该资源“加锁”；如果加锁成功，可以继续执行任务，在整个处理过程中，将保持对该资源的锁定，直到任务完成后解锁为止。

并行服务中锁的类型主要有共享锁、排它锁。根据不同的使用环境，可以采用不同类型的锁来锁定共享资源。为了避免造成死锁，可以对共享资源排序，并规定申请加锁只能以升序进行；有时，还可以要求分布式对象或分布式事务对需要加锁的资源一次申请，要么同时加锁，要么一个也不加锁……所有这些改进，本质上都建立在共享锁及排它锁之上。

在并行服务中，被锁定资源的范围、大小被定义为封锁粒度（granularity）。由于实际应用中的分布式对象或分布式事务千变万化，很难制订统一的封锁粒度。因此，CORBA 并行服务中引入了锁定集对象 Lockset 及事务锁定集对象 TransactionalLockset，用来关联需要同时被锁定的对象、事务及资源，藉以控制封锁粒度。

为了对嵌套事务进行锁定，并行服务中引入了事务锁协调对象 LockCoordinator。一般情况下，如果两个不同的事务拥有一些共同的作用对象，就不能同时申请对事务锁定。但是，在嵌套事务的同一个事务系中，这种限制可以适当放宽。因为父类事务与子类事务多少具有一些相关性，所以，子事务可以继续申请向已经被父类事务占用的资源加锁，当子事务结束后，父类事务不会丢失对这些资源的封锁权。

实现并行服务的对象主要有锁定集厂对象 LocksetFactory、锁定集对象 Lockset、事务锁定集对象 TransactionalLockset 以及事务锁协调对象 LockCoordinator，其功能分别如下所示：

锁定集厂对象 LocksetFactory。调用 create，本对象可以用来创建锁定集对象 Lockset；调用 create_transactional，本对象可以用来创建事务锁定集对象 TransactionalLockset；调用 create_related 或 create_transactional_related，可以将新创建的锁定集对象或事务锁定集对象与已有的锁定集对象、事务锁定集对象相关联。

锁定集对象 Lockset。调用 lock，本对象以阻塞的方式申请锁定相关资源；调用 try_lock，本对象尝试申请锁定相关资源，不论成功与否，都立即返回，不以阻塞方式运行；

调用 unlock，本对象将释放对相关资源的锁定；如果希望改变锁定方式，可以调用 change_lock；而调用 get_coordinator，本对象可以返回一个事务锁协调对象 LockCoordinator。

事务锁定集对象 TransactionalLockset。本对象中的操作与锁定集对象 Lockset 的操作一致，不过，必须传递一定的参数，用以区别不同事务。当用户以显性传播方式激发事务时才需要使用本对象；否则锁定集对象 Lockset 既支持分布式对象，也支持以隐含传播方式激发的事务。

事务锁协调对象 LockCoordinator。本对象仅有一个操作 drop_locks，将解除某个事务拥有的锁定。

8.10 对象属性服务

OMG IDL 可以用来定义属性。但是，这种属性一经编译后就几乎不能被修改，除非用户去修改定义它们的 OMG IDL 接口。

在现实应用中，有时需要为对象添加动态属性。比如，一个商品可以作为样品被商场用来促销，这时，它通常具备了一些与销售服务有关的新属性：表面上可以粘贴、图覆一些广告信息，允许客户试用，某些部位被“解剖”，以便客户观察内部信息……这些属性几乎不能由商品厂家出厂时定制，只能在使用过程中动态决定。对象属性服务（Property Service）可以用来提供这种服务，在运行过程中为对象动态添加属性。

动态属性由“值-名体”构成，定义如下。

例 8.23 动态属性的接口定义。

```
//这是动态属性的接口定义
struct property{
        string property_name;
        any property_value;
};

typedef sequence<property> properties;
```

property_name 表示属性的名称，property_value 表示属性的取值，甚至可以是别的结构等数据。而 properties 是一系列属性构成的序列，用来表达两个以上的属性。

有时，我们还需要规定属性的使用模式，比如，正常模式（normal）、只读模式（readonly）、免删模式（fixed_normal）、免删只读模式（fixed_readonly）。这时，可以用 PropertyDef 表示属性，定义如下：

例 8.24PropertyDef 的接口定义。

```
//这是 PropertyDef 的接口定义
enum PropertyMode{normal, readonly, fixed_normal,
                        fixed_readonly, undefined};

struct PropertyDef{
        string property_name;
        any property_value;
        PropertyMode property_mode;
};

typedef sequence<PropertyDef> PropertyDefs;
```

其中，normal 表示正常模式，用户可以读写这种属性，也可以在运行时删除该属性；

readonly 表示只读模式，用户只能读取创建该属性时设定的值，不能修改，但可以删除该属性；fixed_normal 表示免删模式，这种属性在对象整个运行过程中不能被删除，但可读可写；fixed_readonly 表示免删只读模式，结合了免删模式与只读模式；undefined 仅仅作为某些函数，如 get_mode 的返回值出现，表示属性模式未详。用户不能直接使用该值。

如果用户希望在运行过程中为对象添加动态属性，就应该让对象继承并实现动态属性集对象接口 PropertySet，该接口定义如下：

例 8.25 动态属性集对象 PropertySet 的接口定义。

```
//这是动态属性集对象 PropertySet 的接口定义
interface PropertySet{
        ……
        void define_property(
                        in PropertyName property_name,
                        in any property_value
                        ) raises(…);
        void define_properties(in properties nproperties) raises(…);
        unsigned long get_number_of_properties();
        void get_all_property_names(
                                in unsigned long how_many,
                                out PropertyNames property_names,
                                out PropertyNamesIterator rest
                                );
        any get_property_value(
                                in PropertyName property_name
                                )raises(…);
        boolean get_properties(
                                in PropertyNames property_names,
                                out properties nProperties
                                );
        void get_all_properties(
                                in unsigned long how_many,
                                out properties nproperties,
                                out PropertiesIterator rest
                                );
        void delete_property(in PropertyName property_name)raises(…);
        void delete_properties(
                                in PropertyNames property_names
                                )raises(…);
        void delete_all_properties();
        boolean is_property_defined(
                                in PropertyName property_name
                                )raises(…);
            ……
};
```

如果希望修改属性的取值，可以调用 define_property、define_properties；如果希望了解动态属性的个数，可以调用 get_number_of_properties；如果希望列表或获取指定属性的名称、取值，可以调用 get_all_property_names、get_property_value、get_properties、get_all_properties；如果希望删除属性，可以调用 delete_property 、delete_properties、delete_all_properties；如果希望查询是否定义了某个属性，可以调用 is_property_defined。

如果用户希望为动态对象设置模式，可以用模式属性集接口 PropertySetDef 代替动态

属性集接口 PropertySet。

PropertySetDef 是从 PropertySet 派生的接口，新添加了修改属性模式、获取属性模式等操作，包括 get_allowed_property_types、get_allowd_properties、get_property_mode、get_property_modes 、 define_property_with_mode 、 define_properties_with_modes 、set_property_mode、set_property_modes 八个。

8.11 对象查询服务

对象查询服务的目标是使用户可以采用一种通用的查询语言来查找任何 CORBA 对象，不论它们处于持续状态还是动态状态，也不管它们是本地对象还是远程对象……

显然，对象查询服务首先必须应该提供一些通用查询语言。目前，有两种语言可以用来设计对象查询任务：

有对象扩展的结构化查询语言 SQL3

ODMG-93 的对象查询语言 OQL

不过，OMG 声称，随着 CORBA 技术的不断进步，最终将只有一种查询语言。

8.11.1 对象查询的形式

对象查询可以采用如下所示多种形式进行：

在编译查询任务后直接进行查询

通过一些具备查询管理功能的机制分解、优化查询任务，然后并行执行各个子查询任务

将查询任务授权给一些具备评估功能的机制，利用最适合对象驻留、存储方式的局部搜索引擎查询，然后由评估机制综合评定、返回查询结果。

CORBA 对象查询服务定义了多种接口对象，通过不同组合方式，可以实现上述各种形式的对象查询。

另外，当用户执行一个查询任务后，往往不会恰好得到一个符合条件的对象，而是一组符合条件的对象集合。为了能够方便的操纵、处理这个结果集合，CORBA 中还定义了一组集合操作对象。

8.11.2 集合操作对象

集合操作对象包括集对象 Collection、集厂对象 CollectionFactory 和集迭代器对象 Iterator，其功能分别说明如下：

集对象 Collection。本对象是各种 CORBA 对象的一个包容集，可以增加、更换、删除、检索被包容的对象。如果希望增加对象，可以调用 add_all_elements 或 add_element，也可以调用 insert_element_at 将对象添加在指定位置；如果希望更换、删除、获取指定位置的对象，可以分别调用 replace_element_at、remove_element_at、retrieve_element_at。如果希望为包容集创建一个集合迭代器，可以调用 create_iterator。

集厂对象 CollectionFactory。与其它厂对象一样，可以调用 create 来创建集对象 Collection。

集迭代器对象 Iterator。本对象为查看包容集提供了必要的操作。如果希望定位到第一个对象，可以调用 reset；如果希望定位到下一个对象，可以调用 next；如果希望还有后续的对象，可以调用 more。

通过这些集合操作对象，我们可以非常方便地控制查询结果。

8.11.3 查询服务的有关对象接口

CORBA 对象查询服务共提供了 5 个接口，用于实现不同形式的查询方式，包括查询评估对象 QueryEvaluator、查询管理对象 QueryManager、查询对象 Query 以及可查询集对象 QueryableCollection，功能分别如下所示：

查询对象 Query。本对象用来表示一个查询任务。Prepare 用来编译查询，为执行查询做准备；Execute 用来执行一个编译成功的查询任务；Get_status 用来检查目前任务的状态；Get_result 返回查询结果。

查询评估对象 QueryEvaluator。本对象含有 evaluate 操作，可以根据需要评估一个查询。

查询管理对象 QueryManger。本对象从 QueryEvaluator 派生而来，添加了一个 create 操作，用来创建查询对象 Query。

可查询集对象 QueryableCollection。本对象继承了 QueryEvaluator 以及集对象 Collection 的接口，没有添加新的操作。可以用来处理嵌套子查询。

8.12 对象包容服务

对象包容服务（Collection Service）为对象提供了各种各样的容器，包括队列、堆栈、数组、树、集合、包等等。与 ANSI 的 C++标准模板（STL）以及 Rouge 公司的 Wave 类似，这些容器不仅可以包容对象，而且也定义了与数据结构相关的操作。比如，集合中就定义了子、交、并、补等运算。

包容类的容器通常由三部分接口组成，容器对象接口、容器厂对象接口以及容器迭代器接口。上节中，我们曾经说明过集对象 Collection、集厂对象 CollectionFactory、集迭代器对象 Itertaor。实际上，集对象 Collection 以及集迭代器对象 Itertaor 恰好是所有包容容器、包容容器迭代器的基类。

一般情况下，可以从以下四个方面区分包容容器的固有属性：

是否有序。有的容器中的对象本身就具有顺序，属于显式有序，如数组、队列；有的容器中的对象本身没有顺序，但用户可以要求按照某种方式排序，属于隐式有序，如排序包（SortedBag）、排序集合（SortedSet）；有的容器中的对象根本没有任何顺序，如包（Bag）、集合（Set）。

是否加密。有的容器要求用户通过密钥使用其中的对象，如加密包（KeyBag）、加密排序集合（KeySortedSet）。

是否能够测试元素等同性。有的容器能够测试元素的等同性，如集合（Set）。

元素取值是否唯一。有的容器要求每个元素只能有一个具体值。

组合以上几个属性，我们可以得到各具特色的容器对象以及它们相应的容器厂对象和容器迭代器。这里不再赘述。

8.13 对象安全服务

安全性是分布式软件、电子商务的重要问题，几乎涉及因特网程序开发、使用、维护、管理的每个方面。而且，分布式对象的安全问题比以往的客户机/服务器系统面临更多的挑战，具体如下所示：

分布式对象之间不能轻易相信对方。在客户机/服务器系统中，服务器通常可以被客户信赖，服务器却不能轻信客户；在 CORBA 中，不能明显区别客户与服务器，各个对象都有可能承担客户或服务器角色，因此，谁也不能轻信对方。

分布式对象属于开放式体系结构，不断变化、发展。在客户机/服务器系统中，对象

数目、功能相对稳定，比较容易预测运行图景，往往只需要针对一些固定的问题、服务实施安全措施即可；在 CORBA 中，分布式对象的数目、功能可以不断变化，如何组合、运用这些对象难以预料，安全问题复杂多变。

分布式对象之间的交互作用难以预测。在客户机/服务器系统中，交互作用一般仅仅局限于客户、服务器对象；在分布式对象中，由于一个或多个 ORB 的介入，到底如何交互很难预测。从一定程度上说，越“即插即用”的软件系统就越易于被破坏。

分布式对象使用者对使用对象缺乏了解、控制。分布式对象用接口封装了所有内幕；ORB 又封装了许多对对象的控制功能。对于 CORBA 客户而言，各种幕后工作虽然带来了使用上的透明性，却也容易陷入“宰你没商量”的危险处境。

分布式对象的生存环境具有动态特性，因此安全措施也应该适合于随时变化的动态环境。

分布式对象的数目必然会逐渐增多，如何对这些数目惊人的对象进行访问权限的控制本身就是一个难题。

在 CORBA 中，将安全服务移入了 ORB，使问题变的容易解决。

8.13.1 CORBA 安全服务的功能

为了解决分布式对象之间的安全问题，CORBA 安全服务设计了以下功能：

主体鉴别（Identification and authentication of principals）。主体就是使用人或者对象。经过 ORB 鉴别的主体将会获得一组证书（Credentials），其中包括主体涉及的角色、访问权限等内容。在用户、对象激发各种操作时，ORB 会自动检查、传递主体的有关证书。

特权委托（Privilege Delegation）。在分布式系统中，被客户调用的对象又可以（不断）调用别的对象，并最终形成一个调用链。特权委托规定了在调用链上传递、使用证书的方式：无委托、简单委托或组合委托方式。详细情况稍后说明。

访问控制（Access Control）。通过设置不同的组、角色、访问策略控制用户对于系统、程序、对象、方法的使用权限及使用方式。

安全审核（Security auditing）。通过日志等形式，记录监控的事件。

不可否认性（Non_Repudiation）。ORB 必须能够为某些行为提供不可反驳的证据，证明主体和被调用对象之间发生了某些事件。

安全通信（Secure communication）。通过加密、数字签名等方式，在对象之间构筑一条安全的信息通道。

由于 CORBA 安全服务内置于 ORB，应用程序可以完全忽略 CORBA 安全服务的存在。但是，如果应用程序本身需要较强的安全措施（如各种分布式商务软件），可以直接调用相关安全服务对象接口，实现自己需要的安全控制。

如果用户希望了解自己使用的 ORB 具备哪些安全服务功能，可以调用 ORB 中一个叫做 get_service_information 的函数，它将返回本 ORB 支持的所有安全性质。当然，这个函数也可以返回其它类型的 CORBA 服务，如对象事件服务、持续对象服务、事务服务等信息。

8.13.2 与主体鉴别有关的对象

主体鉴别涉及的对象如下所示：

当前事务对象 Current。我们在事务服务中曾经介绍过该伪对象，通过调用 ORB 中的 get_current 函数，可以获得本伪对象。为了进行主体鉴别，本对象特地定义了两个新操作和一个属性：如果希望将一个安全证书与本对象关联起来，可以调用 get_credentials；如果希望获取本对象关联的安全证书，可以读取属性 received_credentials；如果希望修改安全证书，可以调用 set_credentials。

安全证书对象 Credentials。调用 set_security_features 可以取消或激活证书中的各种安全特征；调用 get_security 可以获得证书中当前有效的安全特征；调用 get_attributes 可以获取安全特征和其它属性的有关取值；调用 set_privileges 可以改变证书的委托特权；调用 isvalid 检查本证书是否有效；调用 refresh、copy 可以刷新、拷贝一个证书对象。

主体鉴别对象 PrincipalAuthenticator。调用 authenticate，可以获得主体一组经过 ORB 鉴别的安全证书对象，包括鉴别机制、主体标志、访问权限等内容；如果希望添加自己的鉴别信息，可以调用 continue_authentication。

对象引用 Object。调用 get_active_credentials 可以获得对象中当前处于活动状态的安全证书对象；调用 override_default_credentials 可以设置对象默认的安全证书对象；调用 get_sercurity_features 可以获得对象支持的安全特征；调用 override_default_QOP 可以改变对象的保护质量；调用 get_security_mechanisms 可以获得对象支持的安全机制；调用 override_default_mechanisms 可以改变对象默认的安全机制；调用 get_security_names 可以获得对象中一些被命名的安全机制。

主体鉴别证书可以通过当前事务对象 Current 自动传递。

8.13.3 特权委托的方式

对象调用链上的特权委托有以下几种方式：

无委托方式。客户不授权中间对象使用自己的安全证书。中间对象根据调用链上前一个对象的安全证书决定自己的访问控制，然后根据结果，利用自己的安全证书去调用位于调用链中的下一个对象。

简单委托方式。客户授权中间对象使用自己的安全证书。中间对象始终模仿客户，将客户的安全证书传递给调用链上的下一个对象。

组合委托方式。客户授权中间对象使用自己的安全证书，并同意中间对象加入自己的安全证书。中间对象将自己的安全证书与调用链中上一个对象的安全证书加以组合，传递给调用链上的下一个对象。

客户还可以设置特权委托的有效时间，规定授权调用链的最大长度，得以进一步控制特权委托。

特权委托可以通过主体鉴别中说明的对象，如当前事务对象 Current 等直接实现。

8.13.4 与安全审核有关的对象

与安全审核直接有关的对象有两个，如下所示：

审核信道对象 AuditChannel。该对象只有一个叫做 audit_write 的函数，用来记录需要审核的行为，包括事件类型、行为主体、事件发生时间、日期等内容。通常用该对象审核与 ORB 无关的活动。

审核决策对象 AuditDecision。本对象只有一个叫做 audit_needed 的操作，用来判断某个行为是否应该由 AuditChannel 审核、记录。

8.13.5 与不可否认性有关的对象

在 CORBA 中，不可否认性仅仅表现在提供、证实不可否认性证据，至于如何传递、保存这些证据并没有规定。因此，也仅有一个对象与不可否认性直接相关：不可否认证书对象 NRCredentials，它具有以下几个操作：

set_NR_features。设置不可否认性服务的特征。

get_NR_features。获取当前不可否认性服务的特征。

generate_token。产生一个令牌，作为不可否认证据的核心内容。

form_complete_evidence。添加诸如时间印（Timestamp）等信息，构造完整的不可否

认证据。

verify_evdience。验证不可否认证据。

get_token_details。获取不可否认证据的完整信息。

8.13.6 各种安全策略对象

由于安全服务的方方面面都有自己的安全策略，CORBA 中的安全策略对象种类颇为繁多，具体情况如下所示：

访问策略基对象 AccessPolicy。本对象是其它安全策略对象的基类，定义了一个 get_effective_rights 函数，用以获取有效的访问权限。

域访问策略对象 DomainAccessPolicy。可以利用 grant_rights、revoke_rights、replace_rights、get_rights 函数分别去授予、撤消、更换、获取域的访问权限。

审核策略对象 AuditPolicy。可以调用 set_audit_selectors、clear_audit_selectors、replace_audit_selectors、get_audit_selectors 函数分别去设置、清除、更换、获取需要审核的行为；也可以调用 set_audit_channel 来指定需要使用的审核信道对象 AuditChannel，决定记录审核信息的渠道。

安全激发策略对象 SecureInvocationPolicy。本对象规定 ORB 激发对象时所采用的安全策略。可以调用 set_association_options、get_association_options 函数设置、获取这些安全策略。

委托策略对象 DelegationPolicy。可以调用本对象中的 set_delegation_mode、get_delegation_mode 函数设置、获取特权委托方式，决定在调用链上如何使用证书。

不可否认策略对象 NRPolicy。可以调用本对象中的 set_NR_policy_info、get_NR_policy_info 函数设置、获取产生或验证不可否认证据的规则。

另外，以下一些对象与安全策略对象密切相关：

域管理对象 DomainManager。本对象中仅定义了一个函数 get_domain_policy，用来获取安全域的有关策略。

使用权对象 RequiredRights。本对象负责封装每个方法的访问策略信息数据库。可以调用 set_required_rights、get_required_rights 函数去设置、获取一个方法的使用权。

8.14 对象时间服务

在分布式系统中，如何保持时间的一致性是一个重要的任务。而时间服务的主要目的就是使得整个分布式系统都遵守一个共同的“软件格林威治时间”，以便完成以下任务：

获取当前时间以及有关的误差

确定事件的发生顺序

提供计时器，产生基于时间的各种事件

计算事件的间隔

保持不同系统的时钟同步

对象时间服务采用的是通用时间协调 UTC 定义的时间，即以 1582 年 10 月 5 日以来流逝的，以 100 纳秒为单位的数字。UTC 是工业界最通用的时间标准。而 CORBA 规定的 UTC 时间总是相对于格林威治零时区时间定义的。

为了提供对象时间服务，CORBA 中定义了以下一些对象接口：

通用时刻对象 UTO。本对象代表一个 UTC 时间数值及其不准确因子。调用 absloute_time，可以获取本 UTO 对象所代表的时刻；调用 interval，可以获取本 UTO 对象的不准确因子（时刻是测不准的）。

时间间隔对象 TIO。本对象表示一段时间间隔，并可以将它与具有不准确因子的时刻

进行对比，求取本时间间隔与指定时刻的相关程度。调用 spans，可以获取本时间间隔在某个指定带有不准确因子的时刻上的跨度；调用 overlaps，可以获取某个指定带有不准确因子的时刻在本时间间隔上的跨度；调用 time，可以将本时间间隔转换为一个 UTO 对象，其不准确因子恰好等于本时间间隔大小。

时间服务器对象 TimeService。本对象可以生成各种通用时刻对象 UTO 及时间间隔对象 TIO。调用 universal_time、secure_universal_time 可以创建 UTO 对象及安全的 UTO 对象；调用 new_universal_time 可以规定某个事件激发的时间；调用 uto_form_utc 可以将一个标准的 UTC 时间转化为一个 UTO 对象；调用 new_interval 可以创建一个时间间隔对象 TIO。

时控事件对象 TimeEventHandler。本对象可以表示某个时控事件。调用 set_timer，可以设置事件触发的时刻（可以是周期性的，也可以是指定时刻）；调用 cancel_timer，可以取消尚未触发的事件；调用 set_data 可以修改由 push 方式传递的事件信息；调用 time_set，可以检查事件是否已经被触发。

时控事件服务器对象 TimeEventService。本对象可以产生时控事件对象 TimeEventHandler，并把它们向有关的事件服务对象注册。本对象以推方式传递事件信息，并定义了 register、unregister 以及 event_time 三个函数。

8.15 对象许可服务

对象许可服务（License Service）恐怕是将分布式软件商业化的一个绝妙服务。它允许对象开发者为对象发放使用证书，只有获得许可的用户才可以使用对象提供的服务。并且，用户可以规定许可的有效事件，监控对象的使用情况。

当然，将分布式软件商业化的服务本身也非常容易用来进行电子商务软件编程：我们既然能为使用对象发放许可证书，同样也能为使用这些对象所蕴涵的服务发放许可证书。

在 CORBA 中，对象许可服务由以下对象接口实现：

许可服务管理对象 LicenceServiceManager。本对象仅有一个操作，名字十分冗长：obtain_producer_specific_license_service，调用本操作，可以为使用者返回被对象开发者许可的对象（也就是服务）的引用。

许可服务对象 ProducerSpecificLicenseService。本对象代表被对象开发者许可使用的对象或服务。调用 start_use，用户可以开始使用本对象或服务；调用 end_use，用户可以结束使用本对象或服务；调用 check_use，本对象或服务可以获取自己是否被使用，被谁使用等信息。

8.16 本章小结

本章主要说明了各式各样的 CORBA 服务。

对象生存期服务提供了与创建、移动、拷贝、释放对象等操作有关的服务，在这里，我们还可以体会到厂对象、厂定位对象的作用。

通过引入角色对象、关系对象、节点对象等接口，对象关系服务几乎可以描述现实世界对象之间的任何关系；而且，随着对象关系的引入，CORBA 为对象生存期服务设计了不同的策略，如 deep、shallow、none、inhibit 等方式。

持续对象服务实际上是由持续对象、持续对象管理器、持续数据服务、数据存储器等对象构成的一个对象状态数据保存体系，每个持续对象拥有一个持续标志号，并通过 connect/disconnect、store/restore 等操作将自己的状态持续化。

对象外化服务则通过一系列基于“流”的对象，提取、恢复对象的状态。

对象命名服务用命名对象 Name 表示对象的名称，并采用递归的方式将名称与命名上

下文对象及有关对象引用绑定、解析，以便用户通过名称获取对象引用。如果用户希望在引用对象之前获取更详细的资料，了解哪个是最适合的服务器、哪个是最经济的对象，就可以毫不犹豫地使用对象洽谈服务。

事件服务就好象对象之间的 E-mail，不过，对象可以根据需要在事件信道上对各种事件消息“推推拉拉”（push/pull）。

事务的特点就是 ACID，为了处理单层事务及嵌套事务，CORBA 中定义了当前事务对象、事务协调对象、资源对象、子事务敏感资源对象等接口；OTS 还要求 ORB 能够以隐含事务传播、显性事务传播方式传递事务上下文对象。

并行服务以封锁资源的方式解决对象并行操作中资源冲突、资源共享等问题。如果希望进一步控制封锁粒度，可以求助于锁定集对象或者事务锁定集对象。

对象属性服务允许用户为对象添加动态属性，这些动态属性可以是只读、免删或两者的组合模式。

对象查询服务希望通过一种操作语言就能够随意查询对象数据、属性、方法、分布等情况。为了分解查询任务，得益于不同系统的本地搜索引擎，对象查询还定义了方式查询评估对象、查询管理对象，查询的结果则可以放入集对象，并通过集迭代器对象操作、控制。

对象包容服务为对象提供了数组、队列、栈、集合等容器，并允许用户决定这些容器是否能够排序、需要排序，是否需要密钥，每个元素取值是否唯一，是否需要测试元素等同性。实际上，对象包容服务提供了大多数常用的数据结构。

对象安全服务提供了主体鉴别、特权委托、访问控制、安全审核、不可否认性、安全通信等功能，并把大多数操作内置于 ORB，一般的用户不必为安全问题过于牵肠挂肚；不过，如果用户希望自己的对象能够自主控制某些安全特性时，可以方便地调用有关对象接口。

对象时间服务允许对象获取当前时间、确定事件发生顺序、计算时间间隔、实现基于时间的各种业务。

许可服务要求用户只有在得到开发者认可的情况下才能使用对象、获取服务。同时，开发者可以监控、跟踪对象使用情况。也许，软件开发人员可以从这种服务中“实现自我”，也可以及时放弃对那些无人问津的对象的维护。

由于 CORBA 基本服务的内容十分庞杂，我们只能采取蜻蜓点水的方式浏览其中希望解决的问题及解决的途径。目前，市场上还没有完全实现上述基本服务的 CORBA 产品。因为这些服务几乎概括了分布式软件中需要解决的所有棘手问题。但是，这并不表明，我们不能采用 CORBA 技术开发软件，理由非常简单：CORBA 几乎是最先进、最严密、最完善的分布式体系结构，它没有解决或实现的问题，别的技术也不能解决。

第 9 章　C++Builder 开发 CORBA 程序扼要

本章内容提要：

- C++Builder *与* Inprise *的* CORBA *产品* VisiBroker *紧密集成*
- *用* C++Builder *自动编译* OMG IDL *接口文件*
- VisiBroker *的* ORB *扩展* Smart Agent
- *配置* ORB *域*
- VisiBroker *的接口仓库* Interface Repository
- C++Builder *的各种* CORBA *编程向导*（Wizard）
- CORBA *静态激发实例*
- VisiBroker *的* ORB *扩展* Object Activation Daemon
- CORBA *服务器自动启动实例*
- CORBA *动态激发实例*

9.1　Inprise 的 CORBA 产品 VisiBroker

有许多公司都已经按照 OMG 的标准开发出形式、功能各具特色的 CORBA 中间件系统，如 IONA 公司的 Orbix（C++版）、OrbixWeb（Java 版），Digital 公司的 ObjectBroker；IBM 公司的 Component Broker；Sun Microsystems 公司的 NEO（Network Environment Object）、JOE（Java Object Environment），SunSoft 公司的 DOE（Distributed Object Everywhere）等等。

由于 Microsoft 公司对于分布式软件体系结构的理解是 COM/DCOM，许多公司的 CORBA 产品也考虑了 CORBA 与 COM/DCOM 的兼容性。采用这些公司的 CORBA 中间件，我们可以得益于 CORBA 与 COM/DCOM 两种分布式软件结构。这些产品有 HP 公司的 ORBplus，INOA 公司的 OrbixCOMet，Expersoft 的 CORBA+ ActiveX Bridge，DEC 公司的 DTC（ObjectBroker Desktop Connection）等等。

我们下面主要介绍一下 Inprise 公司的 VisiBroker。

9.1.1　VisiBrokert 简介

VisiBroker 最初被称作 Black Window，由 PostModern 公司研制。这是世界上第一个同时支持客户、对象实现的 ORB（即客户、对象实现端均采用相同的中间件），采用 Java 语言编写。1996 年初，PostModern 被 Visigenic 公司收购，1996 年 7 月，Black Window 被正式命名为 VisiBroker for Java，而 PostModern 的另一个产品 C++ Orbeline 被正式命名为 VisiBroker for C++。

1996 年 7 月底，Netscape 公司宣布他们的 Netscape ONE(Open Network Environment) 将集成 VisiBroker for Java 以支持 CORBA 分布式软件结构；1997 年 2 月，Oracle 公司宣布在他们的 NCA（Network Computing Architecture）体系结构中，将以 VisiBroker 作为请求对象代理 ORB。

1998 年 2 月，Visigenic 被 Inprise 收购，而 Inprise 的前身就是著名的 Borland 公司。Borland 公司的创业产品是 Turbo pascal 编译器，他们的 Borland C++、Turbo C++曾经成为我国程序开发工具中的垄断产品。在 Microsoft 推出 Visual Basic、Visual C++后，Borland C++霸主地位受到冲击。不过，在相当长的时间内，Borland C++ 4.x~5.x 继续与 Visual C++ 1.x~6.x 抗衡。

Delphi 的出现为 Borland 带来了转机。目前，Delphi 已经成为可视化编程工具的经典“作品”，具有强大的数据库、网络、多媒体软件开发功能，用户、第三方支持遍布世界各地。如果说，“希腊美女”Delphi（因为 Delphi 的标志是一个希腊美女）的出现使所有程序员哑口无言的话，那么，她的“妹妹”C++Builder 则更加风姿卓越。

C++Builder 具备 Delphi 的所有功能，直接使用源自 Delphi 的可视化组件库 VCL，并可以直接使用、编译 Delphi 模块；另外，C++Builder 具备许多向导（Wizard），可以将原来已经用 Borland C++或 Visual C++开发的软件自动转换为现有格式，同时支持 OWL 以及 MFC 类库。

更为重要的是，现在，C++Builder 已经集成了他们自己公司（也就是 Inprise）的 CORBA 产品——VisiBroker。

所以，如果我们希望采用 C++开发 CORBA 程序，C++Builder 必然成为一个优秀的工具。当然，如果我们希望采用 Java 开发 CORBA 程序，那么，不妨试试集成了 VisiBroker 的 Java 程序开发工具，Inprise 的 JBuilder。

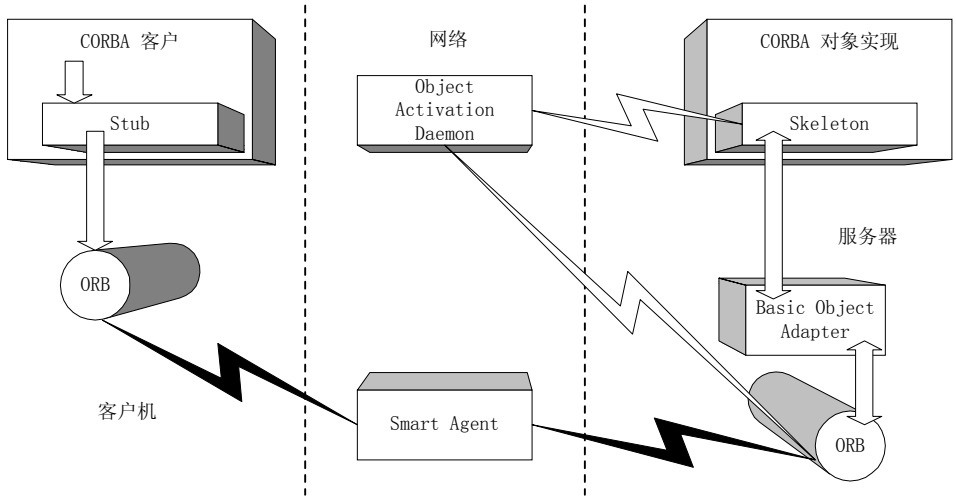

图 9.1　VisiBroker 构成的 CORBA 程序结构

9.1.2　VisiBroker 的特点

目前，VisiBroker 有 C++、Java、Delphi 三个版本，均符合 CORBA2.0 规范，支持跨平台、支持多种操作系统、内置 IIOP 引擎。同时，VisiBroker 还实现了事务服务、命名服务、事件服务等 CORBA 基本服务；能够完成基于 SSL 协议的 IIOP 加密传输。

通常情况下，借助 VisiBroker 构建的 CORBA 分布式软件如图 9.1 所示：

- 接口存根对象 Stub 被 CORBA 客户当作 CORBA 服务的提供者直接使用；而接口存根对象 Stub 却仅仅是一个代理，真正的对象实现既可以与客户程序驻留在相同进程中，也可以与客户程序驻留在相同机器的不同进程中，还可以驻留在别的机器中。
- 接口存根对象 Stub 会把客户的请求转发给与客户程序驻留在相同机器上的 ORB；ORB 通过运行在局域网上的 Smart Agent 定位对象实现的实例。Smart Agent 能够在动态变化的分布式环境下定位一个可以响应客户请求的服务对象，而且，在必要的时候，Smart Agent 能够自动进行负载均衡，当服务器突然崩溃时，Smart Agent 能够重新定位一个服务对象或重新启动服务对象。当然，在客户机器所在的局域网内，至少需要运行一个 Smart Agent。
- 对象实现所在的机器可以被认为是服务器。当客户发出请求时，服务器上的本地 ORB

会把请求传递给接口框架对象 Skeleton，激发有关操作，获取服务。接口框架对象 Skeleton 还可以通过基本对象适配器 BOA 与本地 ORB 通信。比如，当对象实现启动、激活后，接口框架对象 Skeleton 会通过 BOA 向 Smart Agent 注册有关信息。

- 通常情况下，对象实现的实例应该手动启动、关闭。但是，如果对象实现向 OAD（Object Activation Daemon）注册过，则可以在需要的时候由 OAD 自动启动、激活对象实现的实例。向 OAD 注册过的对象实现信息被保存在对象实现仓库中。当服务器的本地 ORB 在接收到有关请求后，会自动与 OAD 通信，如果有必要，OAD 将负责启动、激活一个对象实例。当然，如果希望实现上述功能，在服务器对象实现所在的局域网内，至少需要运行一个 OAD。

另外，VisiBroker 还提供了接口仓库管理工具，用户可以把用 OMG IDL 定义的对象接口信息存储在自由选定的接口仓库中，并通过各种图形界面查询、获取这些信息。这样，CORBA 开发人员就可以十分方便的发布、维护对象接口信息。当用户向 OAD 注册对象实现时，也可以从接口仓库中获益。

随着 Inprise 的兼并，VisiBroker 具备了许多额外的工具，如紧密集成了 VisiBroker 的开发工具 C++Builder、JBuilder、Delphi；用来监控 CORBA 网络运行环境的 VisiBroker Manager；用来远程调试 CORBA 程序模块的 Remote Debugger 等等。

下面，我们将扼要说明如何采用 C++Builder 开发 CORBA 程序。

9.2 编译 IDL 文件自动生成 Stub 及 Skeleton

开发 CORBA 程序时，我们首先需要把用 OMG IDL 编写的接口文件编译为接口存根对象 Stub 及接口框架对象 Skeleton。VisiBroker 为 C++Builder 提供的 IDL 编译器是 IDL2CPP.EXE（为 JBuilder 提供的 IDL 编译器是 IDL2JAVA），如果我们拥有 C++Builder 的 C/S 或企业版（4.0 以上版本），并且已经安装了 VisiBroker，应该可以在 VisiBroker 安装目录的 Bin 子目录下找到该文件。

9.2.1 编译 IDL 文件的步骤

当然，我们完全可以借助 C++Builder 的可视化环境 IDE 完成这些工作。

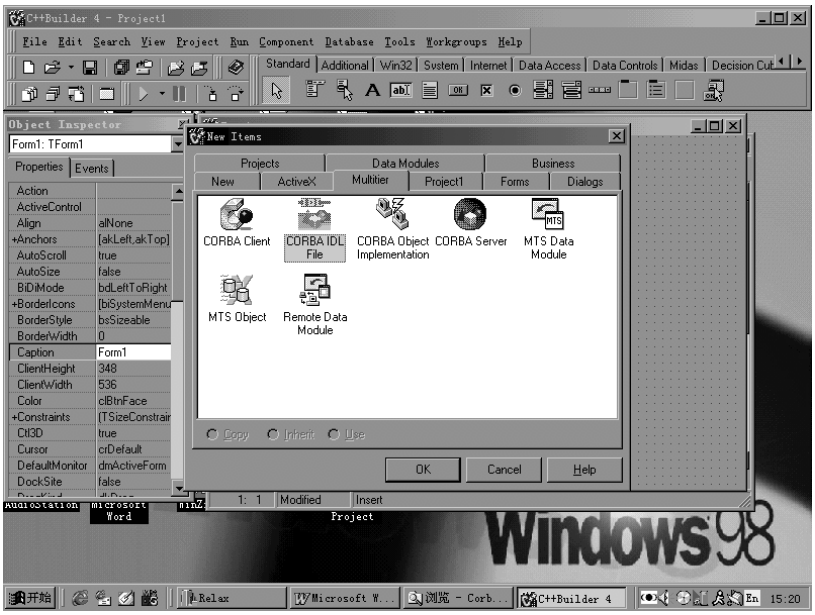

图 9.2　打开 C++Builder 的 OMG IDL 编辑器

采用 OMG IDL 语言编写的接口定义文件可以用任何文本编辑工具书写，只要它的文件扩展名是“.idl”即可。不过，C++Builder 的 OMG IDL 文件编辑器能够以不同颜色显示语法关键字。如图 9.2 所示，执行 File|New 菜单项，在 New Items 对话框中选择 Multitier 页，点取 CORBA IDL File 图标，即可打开该编辑器。

现在，我们在编辑器中编辑如下所示的 IDL 文件：

```
module CorbaTest1{
        interface example1{
                long operation1();
                void operation2(
                        in short param1,
                        out string param2,
                        inout double param3
                        );

                attribute long property1;
                readonly attribute string property2;

                exception myself_exception{
                string reason;
                string name;
                };

        void operation3() raises(myself_exception);
        void operation4() context("info_one","info_two");
        };
};
```

整个文件中的关键字会自动用较深的颜色表示，如图 9.3 所示。如果用户希望自己配置这些颜色，可以执行 Tools|Environment Options 菜单项。

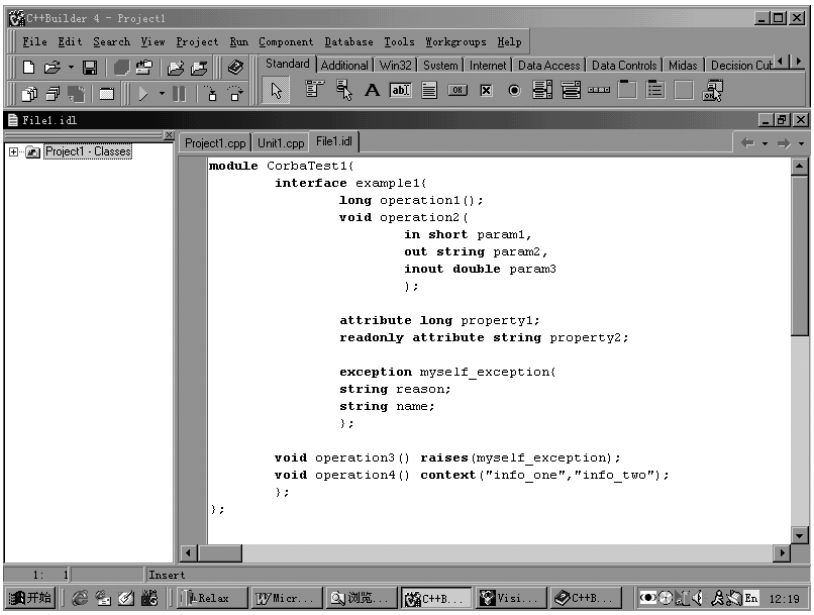

图 9.3　用 C++Builder 编辑 OMG IDL 文件

C++Builder 允许用户以多种方式编译 OMG IDL 文件，生成相应的接口存根对象 Stub 及接口框架对象 Skeleton。相关选项可以通过执行 Project|Options 菜单项，并在如图 9.4 所示的 CORBA 页中设置，主要包括以下内容：

- Object Wrappers：是否需要采用对象包装方式
- Tie：是否需要采用代理捆绑方式
- Virtual Impl. Inheritance：对象实现是否可以虚拟继承
- Typecode Information：是否需要生成类型码
- Include Files Code：是否需要为被 Include 的其它 IDL 文件生成接口框架对象及接口存根对象
- Stream Operators：是否需要支持流操作
- Header Extension：指定接口存根对象及接口框架对象头文件的扩展名
- Source Extension：指定接口存根对象及接口框架对象代码文件的扩展名
- Add Server Unit to Project：是否需要从 IDL 文件中编译出接口框架对象
- Add Client Unit to Project：是否需要从 IDL 文件中编译出接口存根对象
- Include IR：程序中是否需要与接口仓库直接通信
- Include DSI：程序中是否需要使用动态框架接口 DSI

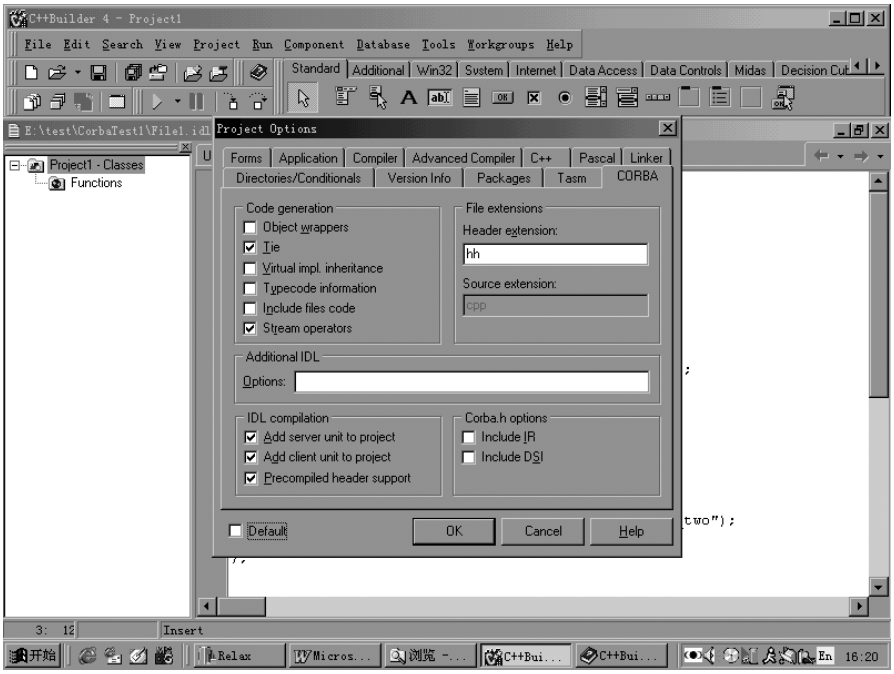

图 9.4　编译 OMG IDL 文件的选项

为了讨论方便，我们将上述选项中与 Code generation 有关的参数全部清除，这样，我们可以获得最简单、最本质的接口框架对象 Skeleton 及接口存根对象 Stub。

可以用扩展名为“.idl”的任何名称保存这个编辑完毕的 OMG IDL 文件，并执行 Project|Add to Project 菜单项，将该文件加入一个工程项目。这时，如果执行 View|Project Manager 菜单项激活工程管理器，应该可以在项目列表中看见这个文件。

确保需要编译的 OMG IDL 文件在 C++Builder 多页编辑器 Code Editor 中处于激活状态，执行 Project|Compile Unit 菜单项，我们可以获得相应的接口存根对象 Stub 及接口框架对象 Skeleton。而且，有两个文件为自动出现在多页编辑器中，如图 9.5 所示：

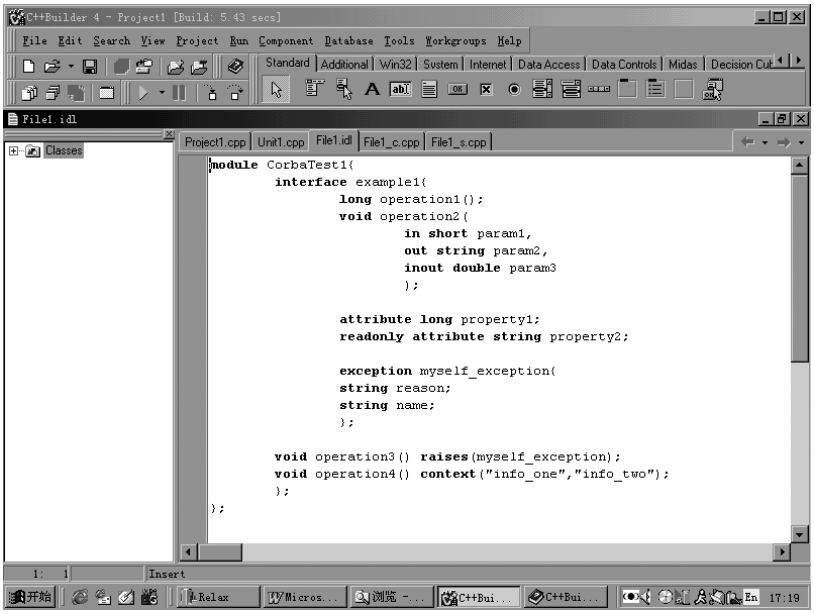

图 9.5　包含接口存根对象及接口框架对象的文件自动出现

9.2.2　Stub 及 Skeleton 代码解析

默认情况下，接口存根对象 Stub 的定义在 xxx_c.hh 中。该文件全部由 C++Builer 的 IDL2CPP 编译生成。下面，我们对其进行简单注释、说明，以便读者可以连续阅读相关内容。

```
#ifndef _File1_c_hh
#define _File1_c_hh

#include "corba.h"
#include "vpre.h"

class CorbaTest1 {//定义模块
public:

#ifndef _CorbaTest1_example1_var_
#define _CorbaTest1_example1_var_

   class example1;//前置说明
   typedef example1* example1_ptr;//定义了 example1_ptr
   typedef example1_ptr example1Ref;

   //为 example1_ptr 类型对象定义了流操作算子 “>>” 及 “<<”
   friend VISistream& operator>>(VISistream&, example1_ptr&);
   friend VISostream& operator<<(VISostream&, const example1_ptr);

   class example1_out;//example1_out 前置说明
   class example1_var: public CORBA::_var {//定义 example1_var
```

```
    friend class example1  out;

  private:
    example1_ptr _ptr;// example1_out 中包含一个 example1_ptr 指针

  public:
    //为 example1_var 定义赋值算子 “=”
    void operat or=(const example1_var&_v) {
      if ( _ptr ) _release(_ptr);
      if ( _v._ptr )
        _ptr = _duplicate(_v._ptr);
      else
        _ptr = (example1_ptr)NULL;
    }

    static example1_ptr _duplicate(example1_ptr);//定义复制函数
    static void _release(example1_ptr);//定义释放函数

    //example1_var 的三种构造函数
    example1_var();
    example1_var(example1_ptr);
example1_var(const example1_var&);
// example1_var 的析构函数
~example1_var();
// example1_var 与 example1_ptr 之间也可以使用赋值算子 “=”
    example1_var& operator=(example1_ptr);
    operator example1_ptr() const { return _ptr; }//有两种方式获取对象指针
    example1_ptr operator->() const { return _ptr; }
    //对不同的方向属性使用不同的数据类型
    example1_ptr in() const { return _ptr; }//方向属性为 in 时为指针
    example1_ptr& inout() { return _ptr; }//方向属性为 inout 指针引用
example1_ptr& out();//方向属性为 out 指针引用
    example1_ptr _retn() {//作为函数返回值指针
      example1_ptr _tmp_ptr;
      _tmp_ptr = _ptr;
      _ptr = (CorbaTest1::example1_ptr)NULL;//空指针的处理
      return _tmp_ptr;
}
    //为 example1_var 类型对象定义了流操作算子 “>>” 及 “<<”
    friend VISistream& operator>>(VISistream&, example1_var&);
    friend VISostream& operator<<(VISostream&, const example1_var&);
  };

  class example1_out {//当 example1 作为方向属性为 out 的参数时使用
  private:
    example1_ptr & _ptr;//这个类中记录的是指针引用
    static example1* _nil() { return (example1*)NULL; }
    void operator=(const example1_out&);//两个赋值操作符的定义
    void operator=(const example1_var&);
```

```
  public://构造函数及析构函数
    example1_out(const example1_out& _o) : _ptr(_o._ptr) {}
    example1_out(example1_ptr & _p) : _ptr(_p)
      _ptr = _nil();
    }
    example1_out(example1_var& _v) : _ptr(_v._ptr) {
        example1_var::_release(_ptr); _ptr = _nil();
    }
    ~example1_out() {}
    example1_out& operator=(example1_ptr _p) {//又定义了一个赋值操作
        _ptr = _p; return *this;
    }
    operator example1_ptr& () { return _ptr; }//也有两种方式获得指针引用
    example1_ptr& ptr() { return _ptr; }
    example1_ptr operator->() { return _ptr; }
  };


#endif

  //定义 example1 对象，从 CORBA_Object 继承而来
  class example1 : public virtual CORBA_Object {
  private:
    static const CORBA::TypeInfo _class_info;//静态成员：类信息说明
    //example1 的构造函数
example1(const example1&){ ___root = this; }
//example1 的赋值算子 "="
    void operator=(const example1&){}

  protected:
    //与 example1 有关的对象被记录在___root 变量中
    example1_ptr ___root;
    //修改 example1 关联对象的函数
    void set_root(example1_ptr root) {
      ___root = root;
    }

  public:
    //定义用户自定义异常，从 CORBA_UserException 继承而来
    class myself_exception : public CORBA_UserException {
    public:

#if defined(MSVCNEWDLL_BUG)
      void *operator new(size_t ts);
      void *operator new(size_t ts, char*, int) {return operator new(ts);}
      void operator delete(void *p);

#endif
      //静态成员：异常描述
      static const CORBA_Exception::Description _description;
      //用户自定义异常 myself_exception 中的两个参数
```

```
        CORBA::String_var reason;//是自动管理内存的字符串
        CORBA::String_var name;

        //自定义异常 myself_exception 的两个构造函数
        myself_exception() {}
        myself_exception(
              const char * _reason,
              const char * _name) {
           reason = _reason;
           name = _name;
        }
        //用来产生自定义异常的厂函数
        static CORBA::Exception *_factory()
              { return new myself_exception(); }
        //自定义异常析构函数
        ~myself_exception() {}
        virtual const CORBA_Exception::Description& _desc() const;
        //_narrow 函数，将一个异常细化为自定义异常
        static myself_exception *_narrow(CORBA::Exception *_exc);
        CORBA::Exception *_deep_copy() const//深拷贝函数
                                  { return new myself_exception(*this); }
        //引发自定义异常的函数
        void _throw() const { throw *this; }

        void _write(VISostream&) const;//将异常信息写到流对象中的函数
        void _copy(VISistream&);//将异常信息拷贝到流对象中的函数
        void _pretty_print(VISostream&) const;//将异常信息打印到一个流中
        //流算子>>
        inline friend VISistream& operator>>
              (VISistream& _strm, myself_exception& _e) {
           CORBA::String_var _exp_name;
           _strm >> _exp_name;
           _e._copy(_strm);
           return _strm;
           }
     };

     static    const CORBA::TypeInfo *_desc();//返回类说明信息
     virtual const CORBA::TypeInfo *_type_info() const;
     virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
     static CORBA::Object *_factory();

     example1_ptr _this();
   protected:

     //example1 的另外一个构造函数
example1(const char *obj_name = NULL):
CORBA_Object(obj_name, 1) { ___root = this; }
   public:
     //example1 的析构函数
```

```
    virtual ~example1() {}

    static example1_ptr _duplicate(example1_ptr _obj) {
      if ( _obj ) _obj->_ref();
      return _obj;
    }
    static example1_ptr _nil() { return (example1_ptr)NULL; }
    static example1_ptr _narrow(CORBA::Object *_obj);
    static example1_ptr _clone(example1_ptr _obj) {
      CORBA::Object_var _obj_var(__clone(_obj));

#if defined(_HPCC_BUG)
      return _narrow(_obj_var.operator CORBA::Object_ptr());

#else
      return _narrow(_obj_var);

#endif
    }
    //example1 的绑定函数
    static example1_ptr _bind(
        const char *_object_name = NULL,
        const char *_host_name = NULL,
        const CORBA::BindOptions* _opt = NULL,
        CORBA::ORB_ptr _orb = NULL);

    //example1 接口中的只读属性 property2
    virtual char* property2();

    //example1 接口中的属性 property1
    virtual CORBA::Long property1();
virtual void property1(CORBA::Long _val);
    //example1 接口中的 4 个函数

    //本函数中自动添加了上下文对象类型的参数
    virtual void operation4(
        CORBA::Context_ptr _context);

    //虽然本函数被声明为能够引发异常，接口存根中并无特殊地方
    virtual void operation3(
        );

    //本函数中对于不同方向属性的参数映射情况不同
    virtual void operation2(
        CORBA::Short _param1,
        char*& _param2,
        CORBA::Double& _param3
        );

    //本函数也对应于 operation2，但 String 参数的映射情况有所不同
    void operation2(
```

```
            CORBA::Short   param1,
            CORBA::String_out _param2,
            CORBA::Double& _param3
            );

      virtual CORBA::Long operation1(
            );

friend VISistream& operator>>
(VISistream& _strm, example1_ptr& _obj);
friend VISostream& operator<<
(VISostream& _strm, const example1_ptr _obj);
   };

};

#include "vpost.h"

#endif
```

从上述的接口存根对象 Stub 文件中，我们可以获得以下一些重要信息：

- 接口存根对象 Stub 实际上是一组相关对象的声明及实现，包括 xxx_var、xxx_ptr、xxx_out 等类（对象）。
- OMG IDL 中的模块被映射为一个类。
- OMG IDL 中的自定义异常被映射为一个类。
- 与上下文对象有关的操作会自动添加一个_ptr 类型的上下文对象参数。
- OMG IDL 中的 String 可以被映射为多种形式。
- 接口存根对象中除去用户定义的属性、操作、结构外，还包括许多其它函数，如构造函数、赋值算子“=”、流算子“<<”及“>>”、析构函数、_bind 等等。
- 对于不同方向属性的参数，其映射结果不同。

接口存根对象的实现代码在 xxx_c.cpp 中，因为是自动生成代码，我们不再罗列。另外，我们还可以选择不同的编译条件，考察接口存根对象的变化，这里也不再赘述。

默认情况下，接口框架对象 Skeleton 的定义在 xxx_s.hh，内容如下所示：

```
#ifndef _File1_s_hh
#define _File1_s_hh

#include "File1_c.hh"//导入接口存根对象文件
#include "vpre.h"

class _sk_CorbaTest1 {//定义模块的接口框架对象
public:

  //定义 example1 的接口框架对象
  class   _sk_example1 : public   CorbaTest1::example1 {
  protected:

    //定义 example1 接口框架对象的两个构造函数
    _sk_example1(const char *_obj_name = (const char *)NULL);
```

```
        sk_example1(
            const char *_service_name,
            const CORBA::ReferenceData& _data);
        //定义 example1 接口框架对象的析构函数
        virtual ~_sk_example1() {}

    public:
        //定义接口框架对象的静态数据成员
        static const CORBA::TypeInfo _skel_info;

        //定义静态成员函数__noop
        // No op function to force base skeletons to be linked in
static void ___noop();
//以下函数必须由开发人员实现
        // The following operations need to be implemented

        //以下几个函数均是纯虚函数，将与对象实现中的代码自动连接
        virtual char* property2() = 0;
        virtual CORBA::Long property1() = 0;
        virtual void property1(CORBA::Long _val) = 0;
        virtual void operation4(CORBA::Context_ptr _context) = 0;
        virtual void operation3() = 0;
        virtual void operation2(
            CORBA::Short param1,
            char*& param2,
            CORBA::Double& param3
            ) = 0;
        virtual CORBA::Long operation1() = 0;

        //以下静态成员函数由编译器自动实现，将与 ORB 自动连接
        // Skeleton Operations implemented automatically
        static void _get_property2(
            void *_obj,
            CORBA::MarshalInBuffer &_istrm,
            CORBA::Principal_ptr _principal,
            const char *_oper,
            void *_priv_data);

        static void _get_property1(
            void *_obj,
            CORBA::MarshalInBuffer &_istrm,
            CORBA::Principal_ptr _principal,
            const char *_oper,
            void *_priv_data);

        static void _set_property1(
            void *_obj,
            CORBA::MarshalInBuffer &_istrm,
            CORBA::Principal_ptr _principal,
            const char *_oper,
            void *_priv_data);
```

```
    static void   operation4(
          void *_obj,
          CORBA::MarshalInBuffer &_istrm,
          CORBA::Principal_ptr _principal,
          const char *_oper,
          void *_priv_data);

    static void _operation3(
          void *_obj,
          CORBA::MarshalInBuffer &_istrm,
          CORBA::Principal_ptr _principal,
          const char *_oper,
          void *_priv_data);

    static void _operation2(
          void *_obj,
          CORBA::MarshalInBuffer &_istrm,
          CORBA::Principal_ptr _principal,
          const char *_oper,
          void *_priv_data);

    static void _operation1(
          void *_obj,
          CORBA::MarshalInBuffer &_istrm,
          CORBA::Principal_ptr _principal,
          const char *_oper,
          void *_priv_data);

  };

};


#include "vpost.h"

#endif
```

相对而言，接口框架对象 Skeleton 显得更为简洁一些，包括以下特点：

- 接口框架对象必须#include 接口存根对象文件，这也是它显得简洁的真正原因所在。
- 接口框架对象的名称通过在 OMG IDL 文件的 module 或 interface 对象名称前添加_sk_构成，而接口对应的框架对象是一个纯虚父类。
- 接口框架对象中定义了静态数据成员_skel_info，属于 const CORBA::TypeInfo 类型，用来表达与接口对象有关的信息，这些信息的详细内容可以在 xxx_s.cpp 文件中找到。
- 接口框架对象中定义了静态成员函数___noop()，编译器解释说，这个函数将强迫接口框架对象被链接到工程中（参见代码的英文注释）。
- 接口框架对象中定义了一批必须由开发人员实现的纯虚函数，这些函数恰好与 OMG IDL 接口中定义的函数、属性相对应。
- 接口框架对象中为每一个纯虚函数定义了一个由编译器自动实现的静态成员函数，它们的名称互相关联，而所有这些静态成员函数的五个参数都相同。

如果打开接口框架对象的实现代码 xxx_s.cpp，我们可以发现接口框架对象的一些内部秘密：

```
#include <corbapch.h>

#pragma hdrstop
#include "File1_s.hh"

//定义了一个将方法名称与方法指针相关联的数组
static CORBA::MethodDescription __sk_CorbaTest1_example1_methods[] = {
   {"_get_property2", &_sk_CorbaTest1::_sk_example1::_get_property2},
   {"_get_property1", &_sk_CorbaTest1::_sk_example1::_get_property1},
   {"_set_property1", &_sk_CorbaTest1::_sk_example1::_set_property1},
   {"operation4", &_sk_CorbaTest1::_sk_example1::_operation4},
   {"operation3", &_sk_CorbaTest1::_sk_example1::_operation3},
   {"operation2", &_sk_CorbaTest1::_sk_example1::_operation2},
   {"operation1", &_sk_CorbaTest1::_sk_example1::_operation1}
};

/*定义 example1 接口框架对象的类说明信息
  包括名称、所含方法数目（7）、方法名称与方法指针关联数组*/
const CORBA::TypeInfo _sk_CorbaTest1::_sk_example1::_skel_info(
   "CorbaTest1::example1",
   (CORBA::ULong)7,
   __sk_CorbaTest1_example1_methods);

//example1 接口框架对象构造函数的实现
_sk_CorbaTest1::_sk_example1::_sk_example1(const char *_obj_name) {
   _object_name(_obj_name);
}

_sk_CorbaTest1::_sk_example1::_sk_example1(
      const char *_serv_name,
      const CORBA::ReferenceData& _id) {
   _service(_serv_name, _id);
}

//___noop 操作实际上不做任何处理
void _sk_CorbaTest1::_sk_example1::___noop() {}
//下面是几个静态成员函数的实现代码，它们的形式基本一致
void _sk_CorbaTest1::_sk_example1::_get_property2(
      void *_obj,//对象引用
      CORBA::MarshalInBuffer &_istrm,//可以自动编组传递的输入流
      CORBA::Principal_ptr ,//主体鉴别对象为空，这是安全服务的内容
      const char *,
      void *_priv_data)//私有数据（结构）
{
   //重新建立一个对象指针，因为_obj 会自动被释放
   CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

   //初始化可以自动编组传递的输出流，使用了私有数据（结构）
```

```
   VISostream&   ostrm = *(VISostream *)
(CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);

   //调用由开发人员编码实现的操作
   CORBA::String_var _ret = _impl->property2();
   //将调用结果“流”入可以自动编组传递的输出流
   _ostrm << _ret;
}

void _sk_CorbaTest1::_sk_example1::_get_property1(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr ,
      const char *,
      void *_priv_data)
{
   CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

   VISostream& _ostrm = *(VISostream *)
      (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
   CORBA::Long _ret = _impl->property1();
   _ostrm << _ret;
}

void _sk_CorbaTest1::_sk_example1::_set_property1(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr ,
      const char *,
      void *_priv_data)
{
   /*如果有输入参数，必须重新定义一个局部可自动编组传递的输入流，
     因为_istrm 会自动被释放*/
   VISistream& _vistrm = _istrm;
   CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

   CORBA::Long _val;
   /*设置属性的操作不需要返回数据，所以无需自动编组
   传递的输出流，操作顺序也略有不同*/
   _vistrm >> _val;
   _impl->property1(_val);
   _impl->_prepare_reply(_priv_data);

}
void _sk_CorbaTest1::_sk_example1::_operation4(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr _principal,
      const char *_oper,
      void *_priv_data) {
   VISistream& _vistrm = _istrm;
   CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;
```

```
  //定义上下文对象，并使用自动编组传递的输入流进行初始化
  CORBA::Context_var _context;
  _vistrm >> _context;
  _impl->operation4(
        _context);

  VISostream& _ostrm = *(VISostream *)
    (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
}

void _sk_CorbaTest1::_sk_example1::_operation3(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr _principal,
      const char *_oper,
      void *_priv_data) {
  VISistream& _vistrm = _istrm;
  CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

  _impl->operation3(
        );

  VISostream& _ostrm = *(VISostream *)
    (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
}

void _sk_CorbaTest1::_sk_example1::_operation2(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr _principal,
      const char *_oper,
      void *_priv_data) {
  VISistream& _vistrm = _istrm;
  CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

  //从可以自动编组传递的输入流中提取函数的参数;
  CORBA::Short param1;
  CORBA::String_var param2;
  CORBA::Double param3;
  _vistrm >> param1;
  _vistrm >> param3;
  _impl->operation2(
        param1,
        param2.out(),//注意对于方向属性为 out 的参数的一种使用方法
        param3
        );

  VISostream& _ostrm = *(VISostream *)
    (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
  _ostrm << param2;//方向属性为 out 的参数需要返回
  _ostrm << param3;//方向属性为 inout 的参数也需要返回
```

```
}

void _sk_CorbaTest1::_sk_example1::_operation1(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr _principal,
      const char *_oper,
      void *_priv_data) {
   VISistream& _vistrm = _istrm;
   CorbaTest1::example1 *_impl = (CorbaTest1::example1 *)_obj;

   CORBA::Long _ret = _impl->operation1(
         );

   VISostream& _ostrm = *(VISostream *)
     (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
   _ostrm << _ret;//返回值不是 void 类型，该参数也需要返回
}
```

从以上自动编译生成的代码中，我们可以发现，与纯虚函数相关的静态成员函数以一种相似的方法调用了某个纯虚函数，而这些纯虚函数由用户自己编码实现。因此，不难想象，接口框架对象中的静态成员函数是供 ORB、BOA 使用的；纯虚函数就是用户必须自己编码实现的接口操作。

接口存根对象 Stub 及接口框架对象 Skeleton 主要通过可自动编组传递的 I/O 流进行通信和数据交换。当客户调用某个对象实现的方法时，所有参数被压入堆栈，并激发有关的接口存根对象。接口存根对象继而将参数写入可自动编组传递的输入流对象 MarshalInBuffer，并通过 ORB、BOA 将与函数调用有关的数据结构信息传递给接口框架对象，接口框架对象自动解释有关的数据结构信息，激发对象实现上的具体方法，并将结果写入可自动编组传递的输出流对象 MarshalOutBuffer 返回。

上述 OMG IDL 接口文件是专门设计的，几乎囊括了所有常见的问题。如果我们能够读懂由它编译出的接口存根对象及结构框架对象，则任何 OMG IDL 接口的接口存根对象 Stub 及接口框架对象 Skeleton 都不在话下。

9.3 VisiBroker 的 Smart Agent

如果我们希望调试、运行 CORBA 程序，在本机或本机所处的局域网内必须至少运行一个 VisiBroker 的 Smart Agent。Smart Agent 是 ORB 的扩展部分，能够在动态变化的分布式环境中定位一个对象实现的实例，完成客户要求的服务。在 VisiBroker 中，ORB 功能由 orb_b.dll、orb_br.dll、orb_r.dll 等动态链接库实现；Smart Agent 功能由 osagent.exe 实现，这些文件均位于 VisiBroker 安装目录的 Bin 子目录下。

9.3.1 Smart Agent 的特点

执行 Tools|VisiBroker SmartAgent 菜单项是启动 Smart Agent 的最简便方法，这时，一个标题为 VisiBroker SmartAgent 的空窗体会展现出来，标志着 SmartAgent 已经成功运行。

Smart Agent 具有以下一些功能、特点：

- 当客户向 CORBA 对象发出请求时，如果 Smart Agent 仅仅能够定位一个对象实现的实例，就直接向该实例传递请求；如果 Smart Agent 能够定位多个对象实现的实例（也就是发现多个可用服务器），就会自动进行负载均衡。

- 当某个对象实现的实例因网络连接故障等原因出现运行错误时，Smart Agent 可以重定位一个对象实例；如果设置正确，Smart Agent 还可以重启动服务器。
- ORB 通过向所在局域网广播的方式与 Smart Agent 建立通信连接。如果局域网内有多个 Smart Agent 运行，ORB 则与最先响应的 Smart Agent 建立通信关系。一旦与 Smart Agent 建立起通信连接，ORB 自动采用点到点的 UDP（User data protocol）通信协议与 Smart Agent 通信，以便减少网络运行开销。
- 当局域网上有多个 Smart Agent 运行时，每个 Smart Agent 仅仅需要注册一部分对象实现实例的定位信息；如果某个 Smart Agent 发现不能直接定位的对象实例，会自动与其它 Smart Agent 通信（互操作）。
- 当局域网上有多个 Smart Agent 运行时，如果某个 Smart Agent 发生故障，它所记录的对象实例定位信息会自动注册到其它 Smart Agent 上。
- 用户可以通过设置 Smart Agent 的侦听端口，将位于同一个局域网上的多个 Smart Agent 归属于不同的 ORB 域。

9.3.2 配置 Smart Agent、ORB 域

Smart Agent 的上述某些功能必须通过设置不同的启动参数获得。Smart Agent 共有三个参数，详细情况如表 9.1 所示：

表 9.1 Smart Agent 的启动参数

参数	说明	实例
-v	运行过程中将会详细记录各种信息。这些信息存放在 osagent.log 文件中，默认情况下，可以在 VisiBroker 安装目录的子目录 adm\log 中找到。	osagent -v
-p<n>	指定 Smart Agent 侦听局域网广播信息的 UDP 端口，默认情况下，该端口为 14000。在同一个局域网范围内，端口相同的 Smart Agent 属于同一个 ORB 域。	osagent -vp11000 osagent -p6000
-C	在 Windows NT 中，如果 Smart Agent 已经被作为 NT 服务器加载，此参数允许以 DOS 方式显示运行中的 Smart Agent。	osagent -C

确定启动参数后，既可以在 Dos 窗口内运行类似实例中的命令行；也可以先执行任务栏的“运行”菜单项，然后在标题为“运行”的对话框中运行命令行。Smart Agent 会按照用户的意图启动（这时无需运行 C++Builder）。

实际上，在局域网内运行多个 Smart Agent 是一个明智的方案，至少包括两个好处：可以提高可靠性，即使部分 Smart Agent 出现运行故障，整个 CORBA 系统也不会瘫痪。还可以根据用户需要，将局域网划分为多个 ORB 域。图 9.6 就是一个包含两个 ORB 域的局域网实例。

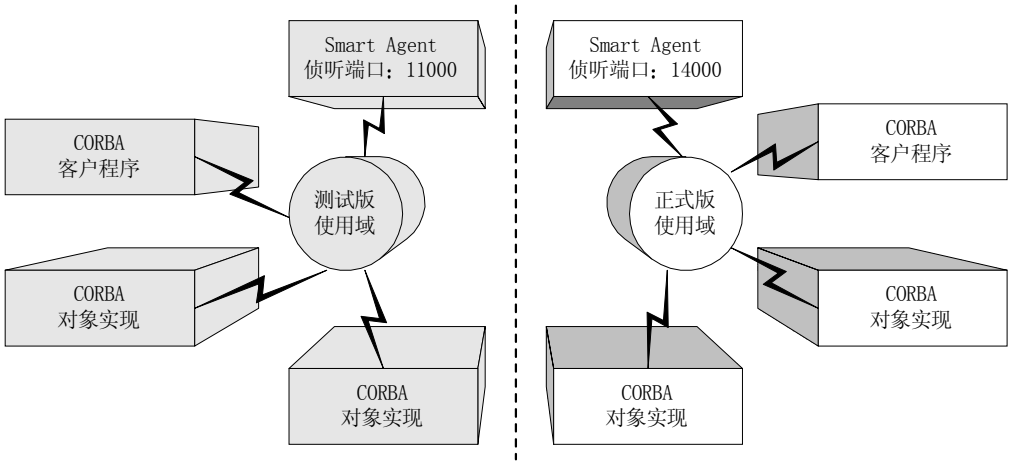

图 9.6　一个局域网中的多个 ORB 域

在测试版使用域中，可以启动一个或多个 Smart Agent，并将它们的 UDP 侦听端口设置为 11000。在正式版使用域中，也可以启动一个或多个 Smart Agent，并将他们的 UDP 侦听端口设置为 14000。当然，同一个域中 Smart Agent 的侦听端口应该一致。而且，我们还需要适当配置各个 ORB 域中的 CORBA 客户程序、对象实现，指定它们能够使用的 Smart Agent。

限定 CORBA 客户程序、对象实现能够使用的 Smart Agent 等 CORBA 设施可以通过表 9.2 所示的环境变量来完成。

表 9.2　与 CORBA 有关的环境变量

变量名	说明
PATH	说明 ORB 库文件所在路径
VBROKER_ADM	指定接口仓库、OAD、Smart Agent 配置文件、log 文件所在目录
OSAGENT_ADDR	指定能够被使用的 Smart Agent 的 IP 地址，如果没有设置本变量，则使用最先响应的 Smart Agent
OSAGENT_PORT	指定能够被使用的 Smart Agent 的 UDP 侦听端口
OSAGENT_ADDR_FILE	说明 agentaddr 文件所在路径名称，该文件中记录着位于其它局域网上的 Smart Agent 的 IP 地址，可以用来进行跨局域网通信
OSAGENT_LOCAL_FILE	说明 localaddr 文件所在路径名称，如果 Smart Agent 运行在网关、桥接等具有多个 IP 地址的主机上，则该文件记录了这些 IP 地址的详细情况
VBROKER_IMPL_PATH	说明被 OAD 使用的对象实现仓库文件的路径名称
VBROKER_IMPL_NAME	说明被 OAD 使用的对象实现仓库文件的名称

以上环境变量都记录在全局变量_environ 中，该变量是一个“char **”类型的字符串数组，其中每个元素都是一个类似“变量名 = 变量值”的字符串。用户可以通过 C++库函数 getenv、_wgetenv 获取某个环境变量的当前取值。如果有必要，还可以通过 C++库函数 putenv、_wputenv 设置某个环境变量的取值，所做改变都只影响到目前程序。

不过，我们非常愿意介绍大家使用 VisiBroker 的另外一个工具：vregedit.exe。该工具位于 VisiBroker 安装目录的 Bin 子目录下，启动后如图 9.7 所示：

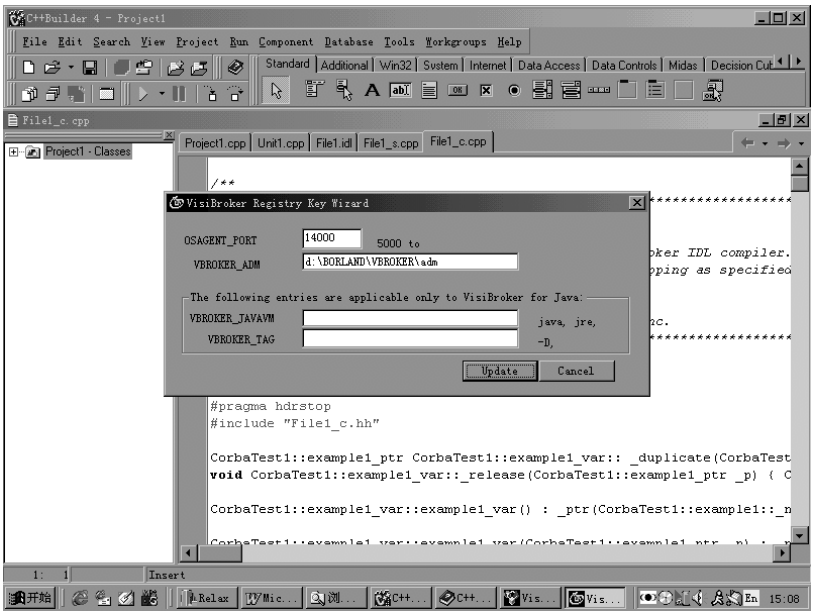

图 9.7　通过 vregedit 配置 ORB 域

该工具把 OSAGENT_PORT 及 VBROKER_ADM 两个变量写入 Windows 注册表。这样，除非用户特别指明，在某机器上启动的 Smart Agent、CORBA 客户程序、CORBA 对象实现会自动使用注册表中的有关设置。无疑，使用该工具配置 ORB 域十分便捷。

另外，不同局域网上的 Smart Agent 之间也可以相互通信，主要通过 agentaddr 或 localaddr 文件实现，这里不再赘述。

9.4　VisiBroker 的接口仓库

我们可以将各种 OMG IDL 接口文件注册到我们认为合适的接口仓库（Interface Repository）中，这样，不论是对象开发人员发布对象接口，还是客户使用对象接口，都会比较方便。

VisiBroker 中与接口仓库有关的文件是 irep.exe 及 idl2ir.exe，前者在局域网内启动一个接口仓库，后者将指定的 IDL 文件注册到指定的接口仓库。这两个执行文件也有很多启动参数，在 Dos 窗口下直接运行 irep 或 idl2ir 会获得相关参数的说明。

执行 Tools|IDL Rspository 菜单项是启动接口仓库的最快方式，但是，我们首先必须保证局域网内已经至少有一个 Smart Agent 在运行，而且其 UDP 侦听端口与本机所在 ORB 域的侦听端口参数吻合。

如果在局域网内还没有启动任何接口仓库，C++Builder 会通知这一情况，随后，我们将得到如图 9.8 所示的接口仓库启动设置窗口。

在该对话框中，用户可以通过 Repository Name 设置接口仓库的名称，如“FirstIR”；还可以通过 Storage File 规定该接口仓库与哪个 IDL 文件相关联（也可以不设置任何值）。另外，最好将 Console 选项清除，否则会以 Dos 方式启动一个接口仓库管理器。

接着，我们会得到如图 9.9 所示的一个接口仓库管理器。现在，我们可以通过这个图形化管理器的菜单来注册（也就是 Load）、下载（也就是 Save、Save As、Copy）我们感兴趣的 IDL 接口文件。

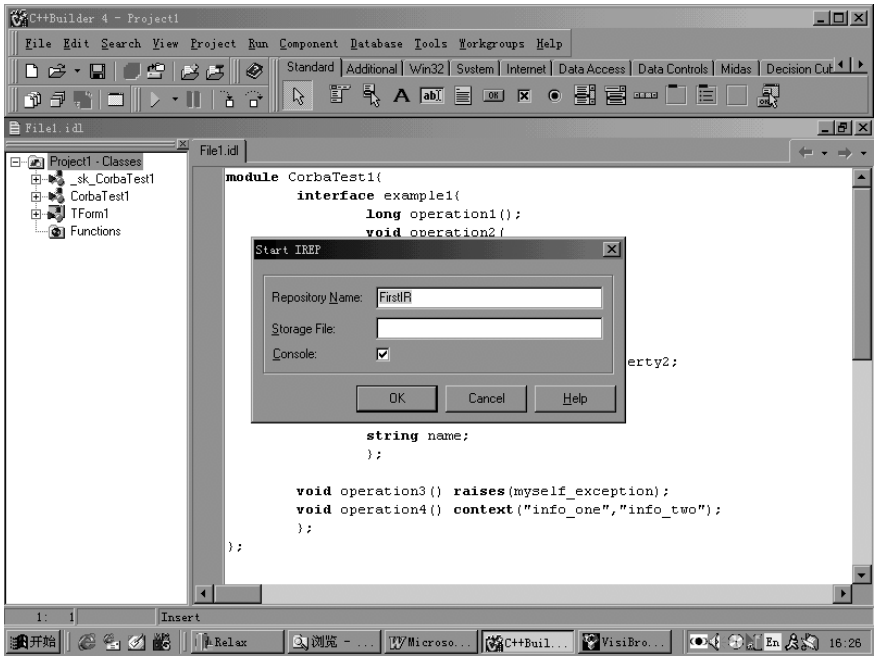

图 9.8　设置接口仓库启动模式

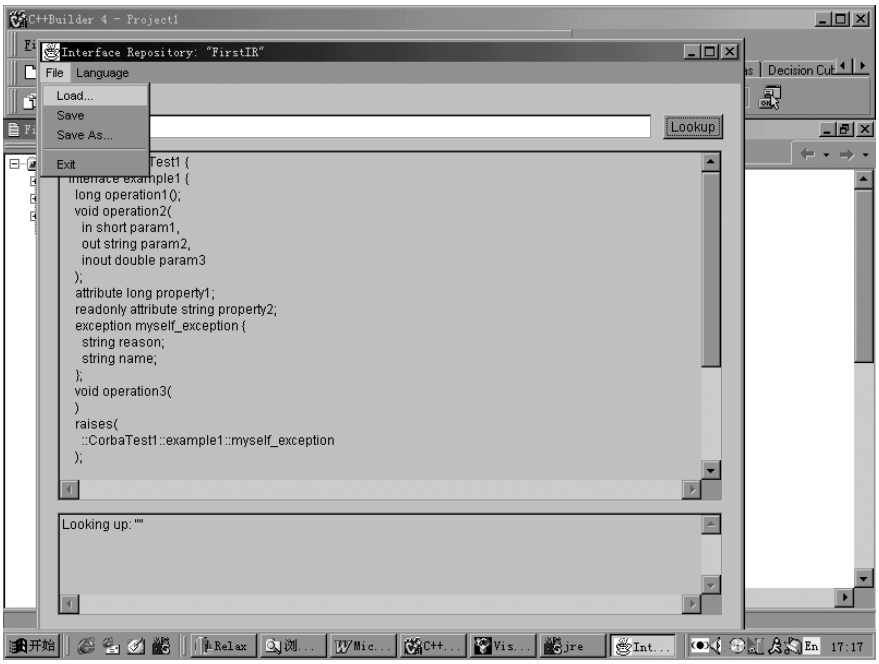

图 9.9　一个图形化接口仓库管理器

如果我们希望在某个接口仓库中查询一个 IDL 接口（可以具体到操作、属性），只需在 Name 编辑框中输入接口名称，如“CorbaTest1::example1”，然后点击 Lookup 按钮即可。这时，我们应该得到有关接口的名称、绝对名称、接口仓库标志号、版本以及接口定义文件清单，如图 9.10 所示。

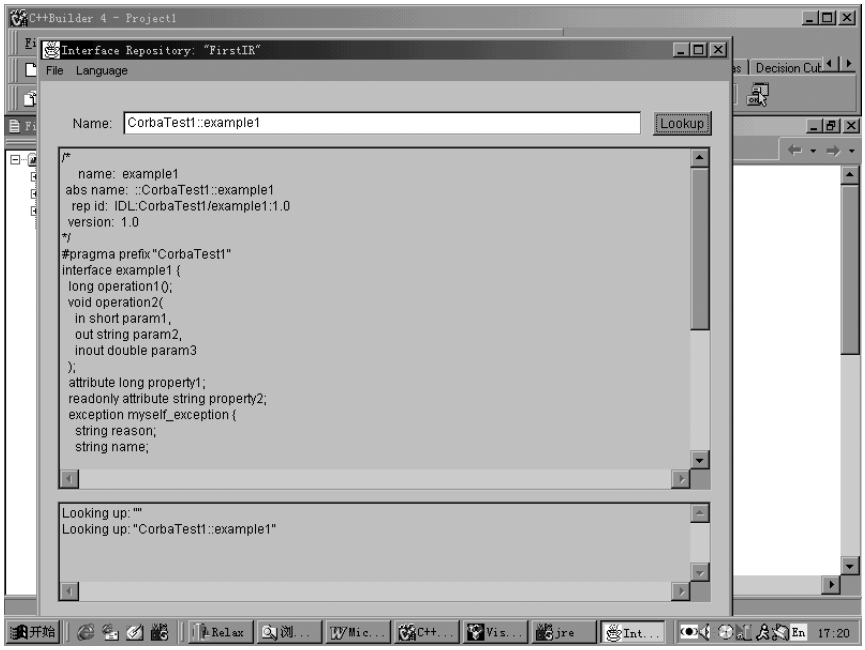

图 9.10　查询 IDL 接口文件

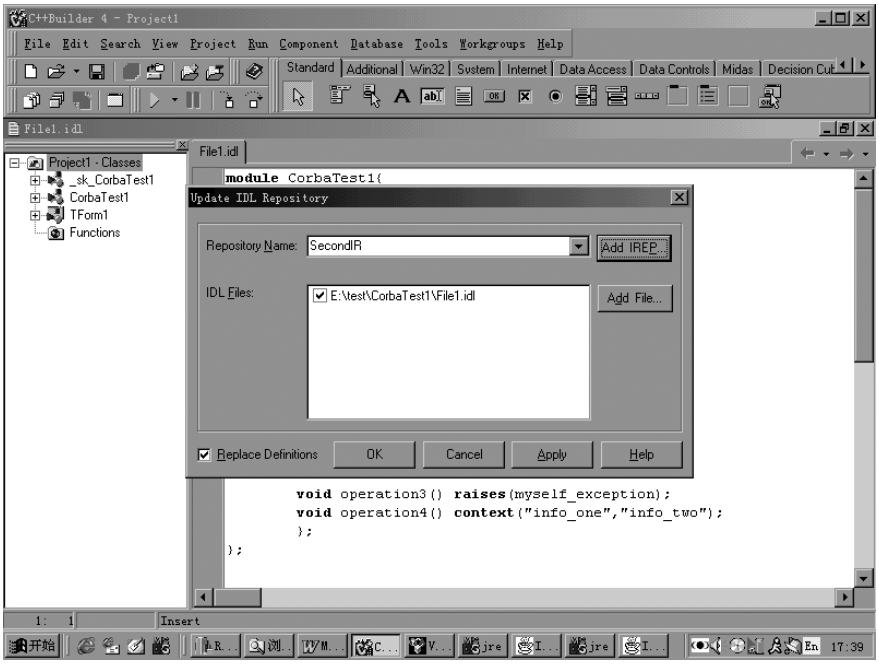

图 9.11　仓库管理器

上述接口仓库管理器非常便于我们集中管理 OMG IDL 接口文件，而 C++Builder 还为我们提供了一个仓库管理器，可以用来同时管理两个以上的接口仓库，如图 9.11 所示。

如果局域网上已经有某个接口仓库处于运行状态，再次执行 Tools|IDL Repository 就会得到这个仓库管理器。点击 Add IREP 按钮，我们可以继续启动一个或多个接口仓库。从 Repository Name 下拉框中，我们可以选择需要进行管理的接口仓库；在 IDL Files 复选列表框中，会显示可以用来注册的 IDL 文件；Add File 按钮用来添加新的 IDL 文件；Replace

Defines 选项用来说明，当发现同名接口定义时，是否需要更新原有定义；Apply 可以在不退出仓库管理器的情况下执行对接口仓库的修改。

当然，我们也可以通过行命令的方式，在 C++Builder 以外启动接口仓库。比如，“irep FirstIR”或“start irep FirstIR”均可。

9.5 开发 CORBA 对象实现

C++Builder 中包含许多向导（Wizard），可以辅助用户开发 CORBA 分布式软件。如果希望编写 CORBA 对象实现，最好从 CORBA Server Wizard 入手。

9.5.1 创建 CORBA 服务器程序

执行 File|New 菜单项，在类似图 9.2 所示的对话框中选择标为 CORBA Server 的图标，就可以激活 CORBA 服务器程序向导——CORBA Server Wizard。实际上，这个向导就是如图 9.12 所示的一个对话框。

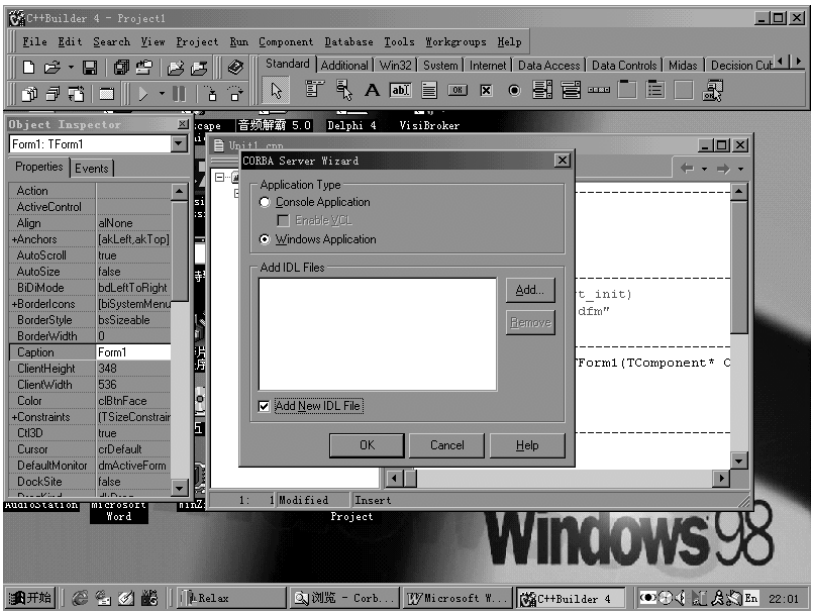

图 9.12 CORBA 服务器程序向导

在这个标题为“CORBA Server Wizard”的对话框中，可以完成两件重要事项：

- 通过 Application Type 单选按钮选择服务器程序类型。如果选择的是 Console Application，则类似于 DOS 程序。但是，如果同时选定了 VCL 选项，在程序中依然可以使用 C++Builder 的 VCL 库。如果选择的是 Windows Application，则为标准的 Windows 程序，自然需要使用 VCL 库。
- 将有关的 IDL 文件加入工程项目。如果希望使用已经存在的 IDL 文件，可以点击 Add 按钮添加；如果希望使用新编写的 IDL 文件，选定 Add New IDL File 选项即可。

如果将服务器程序向导的参数设置为图 9.12 所示，我们可以得到一个由向导自动创建的工程项目，主要包括一个工程文件 Project1.cpp、一个单元文件 Unit1.cpp、一个没有内容的 IDL 文件 File1.idl 及一些 C++Builder 的其它常规文件。

与一般的工程文件相比，这个 Project1.cpp 中多了两行初始化 ORB 及 BOA 的代码，如下所示：

```
    // Initialize the ORB and BOA
    CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
```

Project1.cpp 整个文件内容如下所示：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <corba.h>//说明需要包括一个与 CORBA 有关的头文件
USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
USEIDL("File1.idl");//说明需要使用用户规定的 IDL 文件
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        //以下三行代码是一般工程文件所没有的
        // Initialize the ORB and BOA
        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
        CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

这些代码说明，服务器程序向导为我们完成了下列工作：

- 包括了一些必要的头文件，如 CORBA.h
- 使用了用户规定的 IDL 文件，如 File1.idl
- 生成了 ORB、BOA 初始化代码。但是，在实际应用中，我们有可能需要将这些代码移到其它地方。

另外，还有一点值得注意：在创建服务器程序的主窗口之前，我们就已经完成 ORB、BOA 的初始化工作了。

现在，我们在 File.idl 键入一个新的 IDL 接口，内容如下所示：

```
interface MyFirstCorbaObj{
    string TheResponse(
        in string ClientName,
        in string ServiceName
        );
};
```

这个接口非常简单。但是，我们相信，复杂的程序总是由一系列简单的程序模块有序

构成的。不管怎样，麻雀虽小，五脏俱全。

9.5.2 CORBA 对象实现向导

C++Builder 也为编写 CORBA 对象实现提供了向导：CORBA Object Implementation Wizard。在启动对象实现向导之前，我们应该在如图 9.4 所示的对话框中设置编译 IDL 文件的选项，使 C++Builder 能够生成具有我们所希望功能的接口存根对象 Stub 及接口框架对象 Skeleton。

执行 File|New 菜单项，在 New Items 多页对话框中选择 Multitier 页面，双击标有 CORBA Object Implementation 的图标，C++Builder 会自动编译 IDL 文件，并得到如图 9.13 所示的对象实现向导。在本例中，我们只要求 C++Builder 为对象实现工程自动编译出最简单的接口存根对象 Stub 及接口框架对象 Skeleton 文件。

在这个标题为 CORBA Object Implementation Wizard 的对话框中，我们可以完成以下选择、操作：

- 在 IDL File 下拉框中选择需要实现的对象的接口文件，如果有必要，还可以引入新的 IDL 接口文件。在本例中，我们选择的是 File1.idl。
- 在 Interface Name 下拉框中选择需要实现的接口名称。在本例中，我们选择的是 MyFirstCorbaObj。

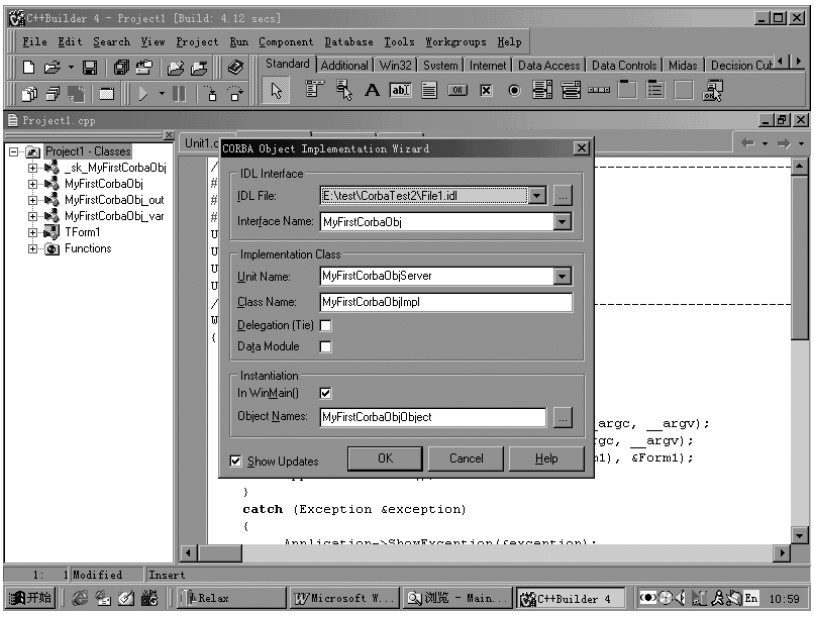

图 9.13　对象实现向导

- 在 Unit Name 下拉框中选择保存该对象实现代码的单元文件名称。如有必要，用户也可以自行定义这个名称。在本例中，我们选择的是默认文件名 MyFirstCorbaObjServer。
- 在 Class Name 编辑框中键入实现接口的类的名称。在本例中，我们选择的是默认类（对象）名称 MyFirstCorbaObjImpl。
- 设置 Delegation（Tie）选项决定是否使用代理捆绑模式。如果使用 Tie 模式，用户既可以将以前的遗留程序打包，借以快速实现接口对象；也可以用非标准方式实现接口对象。在本例中，我们不使用 Tie 模式。
- 设置 DataModule 选项决定是否需要使用数据模块。如果使用数据模块，可以将 VCL

中的 Table、Query、Database 等数据库控件及各种 PageProducer、WebDispatcher 等 Web 控件放在 DataModule 内，构造多层分布式软件。此时，必须使用代理捆绑模式。在本例中，不使用数据模块。

- 设置 In WinMain 选项决定对象是否需要 Startup 服务。如果需要 Startup 服务，C++Builder 会自动编写代码，在程序启动时创建对象实例；否则，用户需要自己编码创建对象实例。在本例中，我们采用了 Startup 服务。
- 在 Object Names 编辑框中说明对象实现一个或多个实例的有关名称。在本例中，我们使用的是默认的实例名称 MyFirstCorbaObjObject。
- 设置 Show Updates 选项决定是否需要预览由对象实现向导自动创建的程序代码。在本例中，我们采用了预览功能，这样，就可以在向导生成代码之前进行必要的预览、修改。

对象实现向导的代码预览器如图 9.14 所示，它以可视化方式有机地显示了向导将对工程项目所做的各种修改。

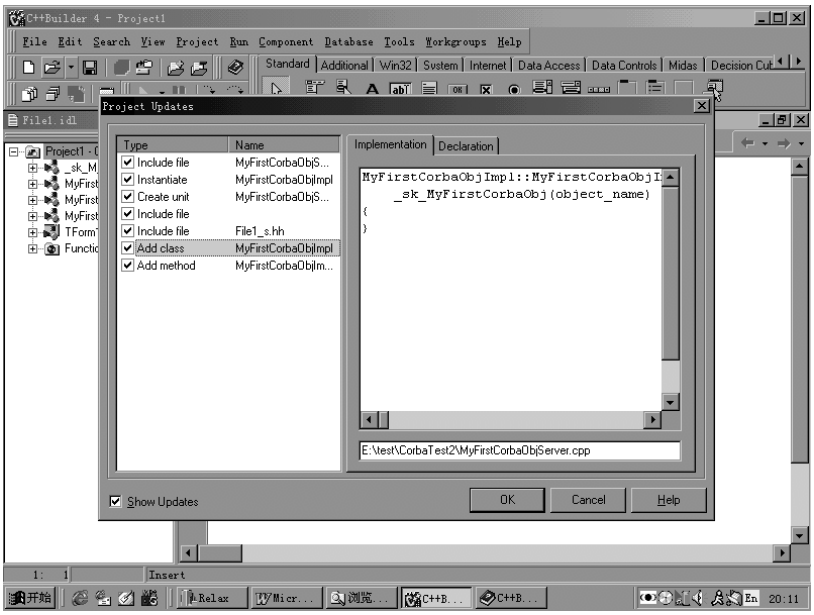

图 9.14　预览对象实现向导生成的代码

在预览器的左侧，列举了将要被添加、修改的代码的类型及名称；在预览器的右侧，可以看见这些代码的内容，如果需要，我们可以直接修改；在预览器的右下侧，说明了这些代码将被添加哪个模块。

如果我们希望实现多个接口对象，就需要多次激活 CORBA 对象实现向导。不过通过设置文件名称，我们可以将所有对象实现代码集中于一个或多个文件。

9.5.3 编写 CORBA 对象实现

对象实现向导在添加代码之前，会自动编译 IDL 文件，生成有关接口存根对象及接口框架对象的文件。为了保持阅读的连续性，我们将由 File1.idl 生成的 Skeleton、Stub 文件列举如下，首先是头文件 File1_s.hh：

```
#ifndef _File1_s_hh
#define _File1_s_hh
```

```
#include "File1_c.hh"
#include "vpre.h"

class    _sk_MyFirstCorbaObj : public    MyFirstCorbaObj {
protected:

   _sk_MyFirstCorbaObj(const char *_obj_name = (const char *)NULL);
   _sk_MyFirstCorbaObj(
         const char *_service_name,
         const CORBA::ReferenceData& _data);
   virtual ~_sk_MyFirstCorbaObj() {}

public:
   static const CORBA::TypeInfo _skel_info;

   // No op function to force base skeletons to be linked in
   static void ___noop();
   // The following operations need to be implemented

   virtual char* TheResponse(
         const char* ClientName,
         const char* ServiceName
         ) = 0;

   // Skeleton Operations implemented automatically

   static void _TheResponse(
         void *_obj,
         CORBA::MarshalInBuffer &_istrm,
         CORBA::Principal_ptr _principal,
         const char *_oper,
         void *_priv_data);
};
#include "vpost.h"

#endif
```

File1_s.cpp 文件如下所示：

```
#include <corbapch.h>

#pragma hdrstop
#include "File1_s.hh"

static CORBA::MethodDescription __sk_MyFirstCorbaObj_methods[] = {
   {"TheResponse", &_sk_MyFirstCorbaObj::_TheResponse}
};

const CORBA::TypeInfo _sk_MyFirstCorbaObj::_skel_info(
   "MyFirstCorbaObj",
   (CORBA::ULong)1,
   __sk_MyFirstCorbaObj_methods);
```

```
_sk_MyFirstCorbaObj::_sk_MyFirstCorbaObj(const char *_obj_name) {
   _object_name(_obj_name);
}

_sk_MyFirstCorbaObj::_sk_MyFirstCorbaObj(
      const char *_serv_name,
      const CORBA::ReferenceData& _id) {
   _service(_serv_name, _id);
}

void _sk_MyFirstCorbaObj::___noop() {}
void _sk_MyFirstCorbaObj::_TheResponse(
      void *_obj,
      CORBA::MarshalInBuffer &_istrm,
      CORBA::Principal_ptr _principal,
      const char *_oper,
      void *_priv_data) {
   VISistream& _vistrm = _istrm;
   MyFirstCorbaObj *_impl = (MyFirstCorbaObj *)_obj;

   CORBA::String_var ClientName;
   CORBA::String_var ServiceName;
   _vistrm >> ClientName;
   _vistrm >> ServiceName;
   CORBA::String_var _ret = _impl->TheResponse(
         ClientName.in(),
         ServiceName.in()
         );

   VISostream& _ostrm = *(VISostream *)
      (CORBA::MarshalOutBuffer*)_impl->_prepare_reply(_priv_data);
   _ostrm << _ret;
}
```

接口存根对象 Stub 的头文件 File1_c.hh 如下所示：

```
#ifndef _File1_c_hh
#define _File1_c_hh

#include "corba.h"
#include "vpre.h"

#ifndef _MyFirstCorbaObj_var_
#define _MyFirstCorbaObj_var_

class MyFirstCorbaObj;
typedef MyFirstCorbaObj* MyFirstCorbaObj_ptr;
typedef MyFirstCorbaObj_ptr MyFirstCorbaObjRef;

  VISistream& operator>>(VISistream&, MyFirstCorbaObj_ptr&);
  VISostream& operator<<(VISostream&, const MyFirstCorbaObj_ptr);
```

```
class MyFirstCorbaObj  out;
class MyFirstCorbaObj_var: public CORBA::_var {
   friend class MyFirstCorbaObj_out;

private:
   MyFirstCorbaObj_ptr _ptr;

public:
   void operator=(const MyFirstCorbaObj_var&_v) {
      if ( _ptr ) _release(_ptr);
      if ( _v._ptr )
         _ptr = _duplicate(_v._ptr);
      else
         _ptr = (MyFirstCorbaObj_ptr)NULL;
   }

   static MyFirstCorbaObj_ptr _duplicate(MyFirstCorbaObj_ptr);
   static void _release(MyFirstCorbaObj_ptr);

   MyFirstCorbaObj_var();
   MyFirstCorbaObj_var(MyFirstCorbaObj_ptr);
   MyFirstCorbaObj_var(const MyFirstCorbaObj_var&);
   ~MyFirstCorbaObj_var();
   MyFirstCorbaObj_var& operator=(MyFirstCorbaObj_ptr);
   operator MyFirstCorbaObj_ptr() const { return _ptr; }
   MyFirstCorbaObj_ptr operator->() const { return _ptr; }
   MyFirstCorbaObj_ptr in() const { return _ptr; }
   MyFirstCorbaObj_ptr& inout() { return _ptr; }
   MyFirstCorbaObj_ptr& out();
   MyFirstCorbaObj_ptr _retn() {
      MyFirstCorbaObj_ptr _tmp_ptr;
      _tmp_ptr = _ptr;
      _ptr = (MyFirstCorbaObj_ptr)NULL;
      return _tmp_ptr;
   }
   friend VISistream& operator>>(VISistream&, MyFirstCorbaObj_var&);
   friend VISostream& operator<<
            (VISostream&, const MyFirstCorbaObj_var&);
};

class MyFirstCorbaObj_out {
private:
   MyFirstCorbaObj_ptr & _ptr;
   static MyFirstCorbaObj* _nil() { return (MyFirstCorbaObj*)NULL; }
   void operator=(const MyFirstCorbaObj_out&);
   void operator=(const MyFirstCorbaObj_var&);

public:
   MyFirstCorbaObj_out(const MyFirstCorbaObj_out& _o) : _ptr(_o._ptr) {}
   MyFirstCorbaObj_out(MyFirstCorbaObj_ptr & _p) : _ptr(_p)
      _ptr = _nil();
   }
```

```
    MyFirstCorbaObj_out(MyFirstCorbaObj_var&  v) :  ptr(  v.  ptr) {
        MyFirstCorbaObj_var::_release(_ptr); _ptr = _nil();
    }
    ~MyFirstCorbaObj_out() {}
    MyFirstCorbaObj_out& operator=(MyFirstCorbaObj_ptr _p) {
        _ptr = _p; return *this;
    }
    operator MyFirstCorbaObj_ptr& () { return _ptr; }
    MyFirstCorbaObj_ptr& ptr() { return _ptr; }
    MyFirstCorbaObj_ptr operator->() { return _ptr; }
};

#endif

class MyFirstCorbaObj : public virtual CORBA_Object {
private:
    static const CORBA::TypeInfo _class_info;
    MyFirstCorbaObj(const MyFirstCorbaObj&){ ___root = this; }
    void operator=(const MyFirstCorbaObj&){}

protected:
    MyFirstCorbaObj_ptr ___root;
    void set_root(MyFirstCorbaObj_ptr root) {
      ___root = root;
    }

public:

    static    const CORBA::TypeInfo *_desc();
    virtual const CORBA::TypeInfo *_type_info() const;
    virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
    static CORBA::Object *_factory();

    MyFirstCorbaObj_ptr _this();
protected:

    MyFirstCorbaObj(const char *obj_name = NULL):
                        CORBA_Object(obj_name, 1) { ___root = this; }
public:
    virtual ~MyFirstCorbaObj() {}

    static MyFirstCorbaObj_ptr _duplicate(MyFirstCorbaObj_ptr _obj) {
      if ( _obj ) _obj->_ref();
      return _obj;
    }
    static MyFirstCorbaObj_ptr _nil() { return (MyFirstCorbaObj_ptr)NULL; }
    static MyFirstCorbaObj_ptr _narrow(CORBA::Object *_obj);
    static MyFirstCorbaObj_ptr _clone(MyFirstCorbaObj_ptr _obj) {
      CORBA::Object_var _obj_var(__clone(_obj));

#if defined(_HPCC_BUG)
      return _narrow(_obj_var.operator CORBA::Object_ptr());
```

```
#else
      return _narrow(_obj_var);

#endif
   }

   static MyFirstCorbaObj_ptr _bind(
         const char *_object_name = NULL,
         const char *_host_name = NULL,
         const CORBA::BindOptions* _opt = NULL,
         CORBA::ORB_ptr _orb = NULL);

   virtual char* TheResponse(
         const char* _ClientName,
         const char* _ServiceName
         );

   friend VISistream& operator>>
          (VISistream& _strm, MyFirstCorbaObj_ptr& _obj);
   friend VISostream& operator<<
           (VISostream& _strm, const MyFirstCorbaObj_ptr _obj);
};

#include "vpost.h"

#endif
```

接口存根对象的 File1_c.cpp 文件如下所示：

```
#include "File1_c.hh"

MyFirstCorbaObj_ptr MyFirstCorbaObj_var::
                           _duplicate(MyFirstCorbaObj_ptr _p)
                           { return MyFirstCorbaObj::_duplicate(_p); }
void MyFirstCorbaObj_var::_release(MyFirstCorbaObj_ptr _p)
                                 { CORBA::release(_p); }

MyFirstCorbaObj_var::MyFirstCorbaObj_var() :
                            _ptr(MyFirstCorbaObj::_nil()){}

MyFirstCorbaObj_var::MyFirstCorbaObj_var(MyFirstCorbaObj_ptr _p) :
                            _ptr(_p) {}

MyFirstCorbaObj_var::MyFirstCorbaObj_var(
                            const MyFirstCorbaObj_var& _var) :
    _ptr(MyFirstCorbaObj::_duplicate((MyFirstCorbaObj_ptr)_var)) {}

MyFirstCorbaObj_var::~MyFirstCorbaObj_var() { CORBA::release(_ptr); }

MyFirstCorbaObj_var&   MyFirstCorbaObj_var::operator=(MyFirstCorbaObj_ptr
_p) {
   CORBA::release(_ptr);
```

```
    ptr =   p;
  return *this;
}

MyFirstCorbaObj_ptr& MyFirstCorbaObj_var::out() {
  CORBA::release(_ptr);
  _ptr = (MyFirstCorbaObj_ptr)NULL;
  return _ptr;
}

VISistream& operator>>(VISistream& _strm,
                       MyFirstCorbaObj_var& _var) {
                       _strm >> _var._ptr;
                       return _strm;
                       }

VISostream& operator<<(VISostream& _strm,
                       const MyFirstCorbaObj_var& _var) {
                       _strm << _var._ptr;
                       return _strm;
                       }

const CORBA::TypeInfo MyFirstCorbaObj::_class_info(
  "MyFirstCorbaObj",//这是 MyFirstCorbaObj 的类说明信息
  "IDL:MyFirstCorbaObj:1.0",
  NULL,
  &MyFirstCorbaObj::_factory,
  NULL, 0,
  NULL, 0,
  CORBA::Object::_desc(),
  0);

VISistream& operator>>(VISistream& _strm, MyFirstCorbaObj_ptr& _obj) {
  CORBA::Object_var _var_obj(_obj);
  _var_obj = CORBA::Object::_read(_strm, MyFirstCorbaObj::_desc());
  _obj = MyFirstCorbaObj::_narrow(_var_obj);
  return _strm;
}

VISostream& operator<<(VISostream& _strm, const MyFirstCorbaObj_ptr
_obj) {
  _strm << (CORBA_Object_ptr)_obj;
  return _strm;
}

const CORBA::TypeInfo *MyFirstCorbaObj::_desc() { return &_class_info; }

const CORBA::TypeInfo *MyFirstCorbaObj::_type_info() const
                       { return &_class_info; }

void *MyFirstCorbaObj::_safe_narrow(
 const CORBA::TypeInfo& _info) const {
```

```
   if (   info ==   class  info)
      return (void *)this;
   return CORBA_Object::_safe_narrow(_info);
}

CORBA::Object *MyFirstCorbaObj::_factory() {
   return new MyFirstCorbaObj;
}

MyFirstCorbaObj_ptr MyFirstCorbaObj::_this() {
   return MyFirstCorbaObj::_duplicate(___root);
}

MyFirstCorbaObj_ptr MyFirstCorbaObj::_narrow(CORBA::Object *_obj) {
   if ( _obj == CORBA::Object::_nil() )
      return MyFirstCorbaObj::_nil();
   else
return MyFirstCorbaObj::_duplicate(
       (MyFirstCorbaObj_ptr)_obj->_safe_narrow(_class_info));
}

MyFirstCorbaObj *MyFirstCorbaObj::_bind(
      const char *_object_name,
      const char *_host_name,
      const CORBA::BindOptions *_opt,
      CORBA::ORB_ptr _orb) {
   CORBA::Object_var _obj= CORBA::Object::_bind_to_object(
         "IDL:MyFirstCorbaObj:1.0", _object_name, _host_name, _opt, _orb);
   return MyFirstCorbaObj::_narrow(_obj);
}

char* MyFirstCorbaObj::TheResponse(
      const char* _ClientName,
      const char* _ServiceName
      ) {

   char* _ret = (char*)0;
   CORBA_MarshalInBuffer_var _ibuf;
   CORBA::MarshalOutBuffer_var _obuf;

   while( 1 )
      _obuf = ___root->_create_request(
            "TheResponse",
            1,
            122467);
VISostream& _ostrm = *(VISostream *)
                           (CORBA::MarshalOutBuffer*)_obuf;
      _ostrm << _ClientName;
      _ostrm << _ServiceName;


      try
```

```
            ibuf =       root->  invoke(  obuf);
        } catch (const CORBA::TRANSIENT& ) {
           continue;
        }
        break;
    }
    VISistream& _vistrm = *(CORBA::MarshalInBuffer *)_ibuf;
    _vistrm >> _ret;
    return _ret;
}
```

由于接口存根对象及接口框架对象文件内容总是大同小异，我们不再详细解释。

更重要的是，对象实现向导为接口框架对象自动生成了一个代码框架，其头文件名称为 MyFirstCorbaObjServer.h，内容如下所示：

```
//---------------------------------------------------------------------------
#ifndef MyFirstCorbaObjServerH
#define MyFirstCorbaObjServerH
#include "File1_s.hh"
//---------------------------------------------------------------------------
class MyFirstCorbaObjImpl: public _sk_MyFirstCorbaObj
{
protected:
public:
      MyFirstCorbaObjImpl(const char *object_name=NULL);
char* TheResponse(const char* ClientName,
                          const char* ServiceName);
};
#endif
```

从这个文件可以看出，接口框架对象_sk_MyFirstCorbaObj 由 MyFirstCorbaObjImpl 类实现，其代码文件名为 MyFirstCorbaObjServer.cpp，内容如下所示：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>
#include "MyFirstCorbaObjServer.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)


MyFirstCorbaObjImpl::MyFirstCorbaObjImpl(const char *object_name):
      _sk_MyFirstCorbaObj(object_name)
{
}

char* MyFirstCorbaObjImpl::TheResponse(const char* ClientName,
   const char* ServiceName)
```

```
{
}
```

该文件仅仅是对象实现的一个代码框架，用户必须自己编写方法、操作以最终完成对象实现。为此，我们在 MyFirstCorbaObjImpl::TheResponse 函数中加入以下代码：

```
    //下面这些代码由用户添加，完成本操作的功能

    //编写服务日志
    AnsiString Log = "客户名称：";
    Log += ClientName;
    Log += "；服务内容：";
    Log += ServiceName;
    Log += "；时间：";
    Log +=   FormatDateTime("yyyy'年'm'月'd'日'h'点'm'分's'秒'",Now());
    Form1->Memo1->Lines->Add(Log);

    //准备响应数据
    AnsiString Result = "您好，";
    Result += ClientName;
    Result += "！我是 MyFirstCorbaObj，您要的服务是：";
    Result += ServiceName;
    Result += "。您愿意为此项服务付款吗？";
    Result +=   FormatDateTime("yyyy'年'm'月'd'日'h'点'm'分's'秒'",Now());

    //返回响应数据
return Result.c_str();
```

上述代码完成了两件任务：编写服务日志及响应客户请求。由于服务日志被记录在程序主窗体 From1 上的 Memo1 中，我们还必须为该文件添加如下一行：

```
#include "Unit1.h"//本文件使用了程序主窗体，所以必须包括其头文件
```

当然，我们还应该编辑本程序主窗体的标题，添加一个名为 Memo1 的 TMemo 控件，令其对齐方式为 alClient 并使用两个方向的滚动条。到此，我们已经成功的编写了我们的第一个 CORBA 对象实现。

9.5.4 启动对象实例的代码

由于我们选择了 In Winmain 选项，对象实现向导会为我们自动生成用来启动对象实例的代码，内容非常简单，如下所示：

```
MyFirstCorbaObjImpl myFirstCorbaObj_MyFirstCorbaObjObject(
                                                    "MyFirstCorbaObjObject");
boa->obj_is_ready(&myFirstCorbaObj_MyFirstCorbaObjObject);
```

这些代码完成了两项任务：声明一个名为 MyFirstCorbaObjObject 的实例；通知 BOA，对象实例可以接收服务请求。对象实现向导将这些代码放在 Project1.cpp 文件中初始化 ORB 及 BOA 对象的代码之下，不过，用户可以将这些代码移到最合适的地方。

☎ **技巧** 在对象实现向导自动创建的 CORBA 对象实现程序中，与 CORBA 有关的初始化、创建实例等任务代码被放在固定的位置。但是，用户可以将这些代码随意移动。比如，我们可以添加一些按钮，只有在点击这些按钮后才完

成以上这些任务。我们还可以创建不同的子线程，将 CORBA 初始化、实例化代码移入线程执行。实际上，这些任务仅仅是有一定格式和执行顺序的“普通任务”。

9.5.5 修改接口定义

如果用户希望修改 IDL 文件中定义的接口，这些变化也必须反映在对象实现程序代码中。这绝对是一项繁琐的任务！不过，如果我们执行 Edit|CORBA Refresh 菜单项，事情就变的简洁了。这时，将要修改的新代码以及被修改的旧代码都会在代码预览器上显示，以便用户直接控制。

编译整个工程，我们的第一个 CORBA 对象实现程序可以运行了，它的“朴素”的脸孔如图 9.15 所示。

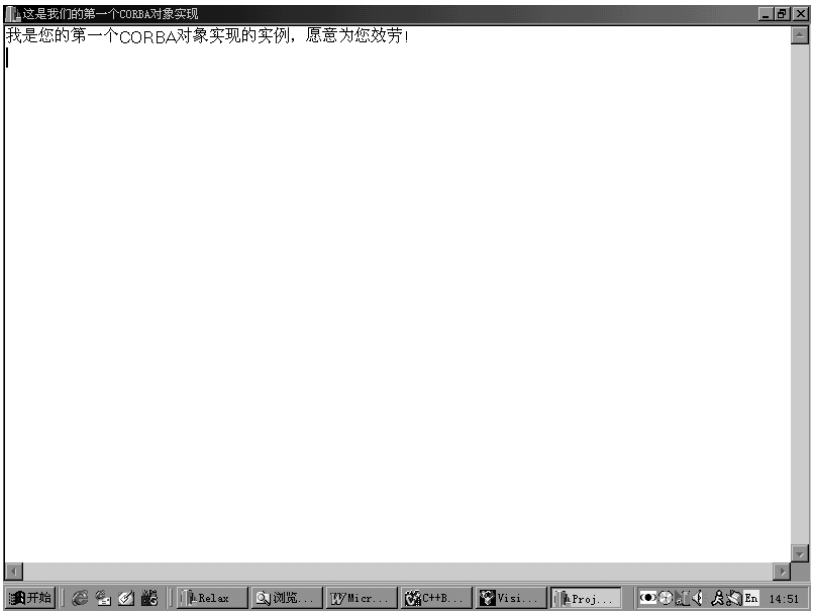

图 9.15　运行中的 CORBA 对象实现程序

9.6 开发 CORBA 客户程序

开发 CORBA 客户程序也不必从头开始，我们可以从接口仓库中获得 IDL 接口定义，然后通过客户程序向导 CORBA Client Wizard 逐步入手。

9.6.1 客户程序向导

执行 File|New 菜单项，在 New Items 多页对话框中选择 Mulitier 页面，点击标有 CORBA Client 的图标，就可以启动如图 9.16 所示的 CORBA 客户程序向导 CORBA Client Wizard。

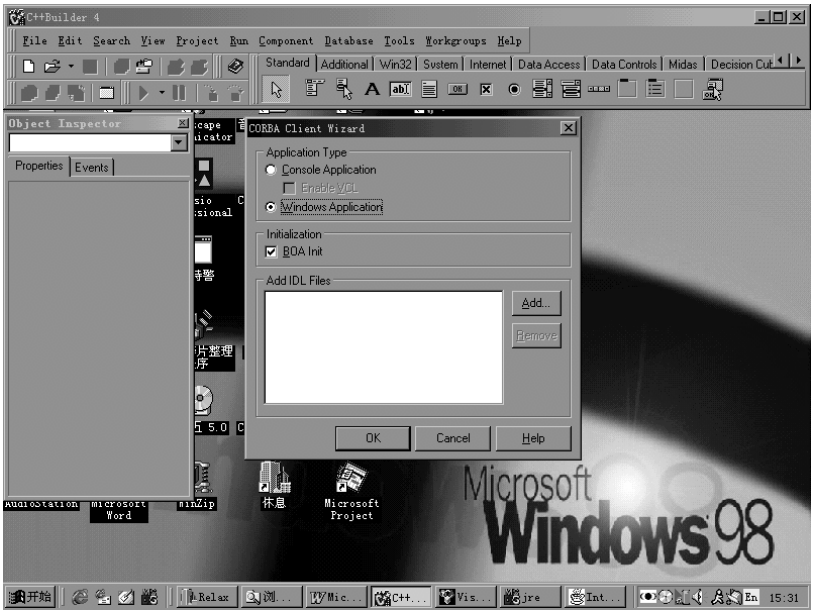

图 9.16　CORBA 客户程序向导

客户程序向导与对象实现向导非常相似，不过，用户可以选择是否需要添加 BOA 初始化代码。在本例中，我们选择了该选项。

客户程序向导创建的代码与对象实现向导创建的代码一模一样，这里不再赘述。

这次，我们还特地从接口仓库中下载有关的 IDL 接口文件，并故意将其命名为 File2.idl 后再加入工程项目。当然，这一步骤并不是必须的，我们只是为了证实接口仓库的功能。

9.6.2　编写静态激发客户程序

在继续编写 CORBA 客户程序之前，我们可以设置 IDL 文件编译选项，使 C++Builder 不再产生接口框架对象 Skeleton 的有关代码，因为这些代码对 CORBA 客户毫无用处。

执行 Edit|Use CORBA Object 菜单项，一个如图 9.17 所示的对象使用向导会展现在我们眼前。在这个标题为 Use CORBA Object Wizard 的对象使用向导中，我们可以完成以下选择、操作：

- 在 IDL File 下拉框中选择需要使用对象所在的 IDL 接口定义文件。在本例中，该文件是我们从接口仓库中 Save As 获得的 File2.idl。
- 在 Interface Name 下拉框中选择需要使用对象所对应的 IDL 接口。在本例中，该接口当然是 MyFirstCorbaObj。
- 在 Object Name 编辑框中说明需要使用对象实例的名称。大家应该没有忘记，在编写对象实现时，对象实现向导曾经要求我们说明一个或多个对象实例名称；CORBA 客户可以要求仅仅同具有某个名称的对象实例通信。在本例中，我们没有规定这个名称，表示“只要接口一致，哪个对象实例都行”。
- 在 Use in Form 对话页面中，允许我们为某个 Form 对象添加一个属性，该属性值就是我们需要使用的 CORBA 对象实例的对象引用。这样，通过该属性，我们可以随时使用这个对象实例。Form 的名称可以在 Form Name 下拉框中选择；属性的名称可以在 New Property Name 编辑框中说明。在本例中，我们采用的是 Form1 及 myFirstCorbaObj。

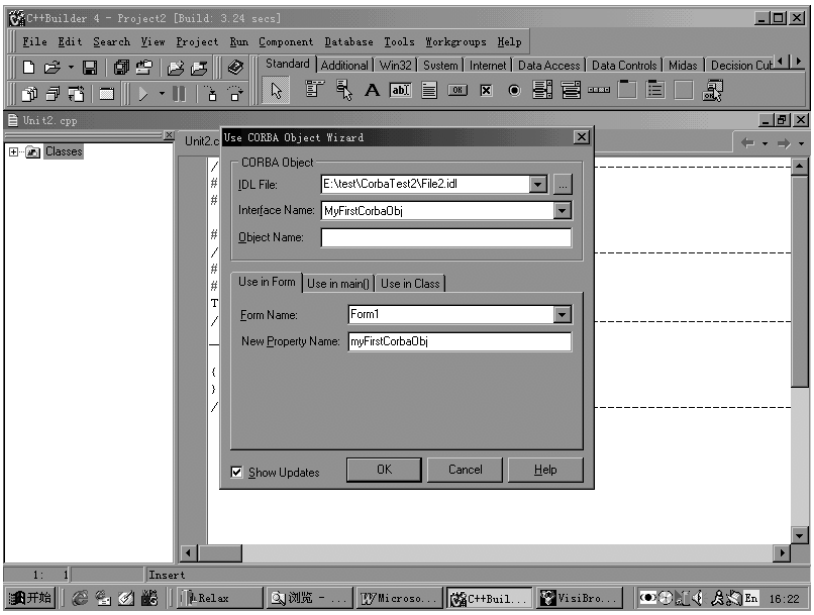

图 9.17　对象使用向导

- 在 Use in Main（）对话页面中，允许我们指定一个变量名称，用来和需要使用的对象实例的对象引用绑定。在本例中，我们没有采用这种方式。
- 在 Use in Class 对话页面中，允许我们在新创建的或者已经存在的类中添加一个变量，用来和需要使用的对象实例的对象引用绑定；如果需要，还可以将这个变量作为类的属性对待。在本例中，我们没有采用这种方式。
- 在 Show Updates 选项中决定是否需要预览自动生成的代码。在本例中，我们需要预览功能。

注意　实际上，对象使用向导所做的最本质的一件事就是用一个用户规定的变量，同需要使用的对象实例的对象引用绑定。不过，对象使用向导为我们在 Form、类及主函数中绑定对象引用、使用对象引用提供了极大便利。

代码预览器表明，对象使用向导为我们在 Unit2.h 中添加了如下所示的声明：@@

```
    MyFirstCorbaObj_ptr __fastcall GetmyFirstCorbaObj();
    MyFirstCorbaObj_var FmyFirstCorbaObj;
void __fastcall SetmyFirstCorbaObj(MyFirstCorbaObj_ptr _ptr);
```

这些声明位于 Private 作用区，定义了两个私有函数、一个私有变量。私有变量用来记录对象引用，两个函数用来辅助实现 Unit2.h 下面一行新添的属性说明：

```
__property MyFirstCorbaObj_ptr myFirstCorbaObj =
            //我们在这里将一行说明分行表示，否则书写不下
            {read=GetmyFirstCorbaObj, write=SetmyFirstCorbaObj};
```

在 Unit2.cpp 中，以下函数代码说明了属性与对象引用绑定的实际过程：

```
MyFirstCorbaObj_ptr __fastcall TForm1::GetmyFirstCorbaObj(void)
{
    if (FmyFirstCorbaObj == NULL)
    {
        FmyFirstCorbaObj = MyFirstCorbaObj::_bind();
```

```
    }
    return FmyFirstCorbaObj;
}

void __fastcall TForm1::SetmyFirstCorbaObj(MyFirstCorbaObj_ptr _ptr)
{
    FmyFirstCorbaObj = _ptr;
}
```

经过进一步加工，整个 Unit2.h 文件如下所示：

```
//---------------------------------------------------------------------------
#ifndef Unit2H
#define Unit2H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "File2_c.hh"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:// IDE-managed Components
    TButton *Button1;//发出请求按钮
    TLabel *Label1;
    TLabel *Label2;
    TEdit *Edit1; //客户名称编辑框
    TEdit *Edit2;//服务名称编辑框
    TMemo *Memo1;//记录运行日志
    void __fastcall Button1Click(TObject *Sender);//点击按钮事件函数
private:// User declarations
    //以下黑体部分代码由对象使用向导生成
    MyFirstCorbaObj_ptr __fastcall GetmyFirstCorbaObj();
    MyFirstCorbaObj_var FmyFirstCorbaObj;
    void __fastcall SetmyFirstCorbaObj(MyFirstCorbaObj_ptr _ptr);
public:         // User declarations
    __fastcall TForm1(TComponent* Owner);
__property MyFirstCorbaObj_ptr myFirstCorbaObj =
  {read=GetmyFirstCorbaObj, write=SetmyFirstCorbaObj};
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

整个 Unit2.cpp 文件如下所示：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>//由对象使用向导添加
```

```
#include "Unit2.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------

//黑体部分由对象使用向导添加
MyFirstCorbaObj_ptr __fastcall TForm1::GetmyFirstCorbaObj(void)
{
    if (FmyFirstCorbaObj == NULL)
    {
        FmyFirstCorbaObj = MyFirstCorbaObj::_bind();
    }
    return FmyFirstCorbaObj;
}

void __fastcall TForm1::SetmyFirstCorbaObj(
                                MyFirstCorbaObj_ptr _ptr)
{
    FmyFirstCorbaObj = _ptr;
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //本函数由用户编写，发送客户名称及服务名称
    Memo1->Lines->Add(
        myFirstCorbaObj->TheResponse(
                        Edit1->Text.c_str(),
                        Edit2->Text.c_str()
                        )
        );
}
//---------------------------------------------------------------------------
```

现在，我们已经全部完成了以静态激发方式运作的 CORBA 客户程序。

9.6.3 运行 CORBA 程序

一般情况下，运行 CORBA 程序首先需要在局域网内至少启动一个 Smart Agent，然后启动 CORBA 服务器，也就是 CORBA 对象实现程序；最后，我们可以在局域网的任何角落启动 CORBA 客户。如果希望在网络的任何角落都能启动 CORBA 客户程序，还需要对网络及 Smart Agent 做一定的设置工作。

图 9.18 是上述最简单的 CORBA 程序的运行情况。

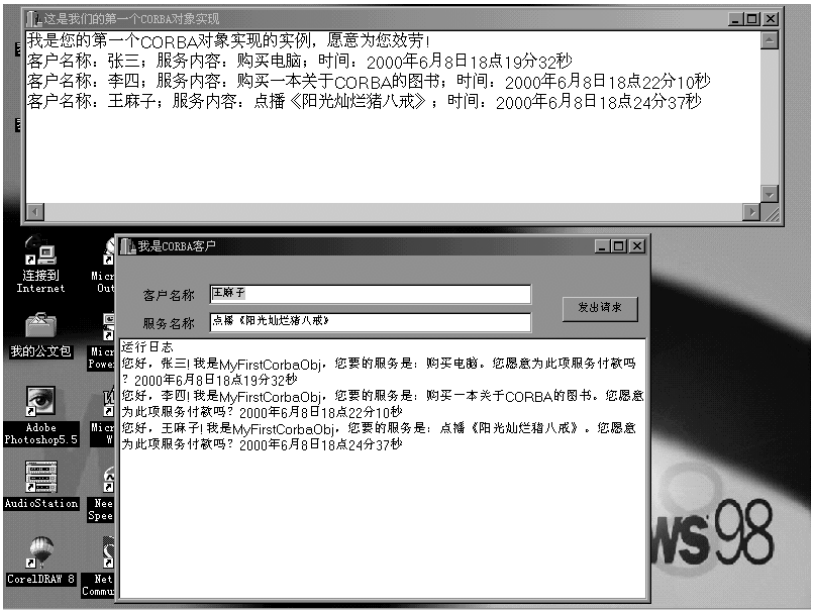

图 9.18　运行一个简单 CORBA 程序

注意　图中显示的仅仅是一个最简单的 CORBA 客户程序及 CORBA 对象实现，如果我们有兴趣，或者有人愿意投资，完全可以用 CORBA 开发功能复杂的电子商务系统。比如，CORBA 客户可以进一步提出服务质量、可接受价格、服务时间、服务地点等请求；CORBA 对象实现则可以一边悄悄更新客户信息数据库，一边根据客户要求去调度、激发、管理更多的 CORBA 对象。

9.6.4　通过 OAD 自动启动 CORBA 服务器程序

在运行上述 CORBA 程序时，用户需要人为启动 CORBA 对象实现程序（也就是 CORBA 服务器），这明显有许多不便之处：我们不可能让所有 CORBA 服务器都处于无限等待状态。

VisiBroker 的 ORB 扩展 Object Activation Daemon 就是用来解决这个问题的。如果用户将 CORBA 服务器程序信息注册到局域网中的某个 OAD 之中，我们就无需手工启动 CORBA 服务器。当 CORBA 客户发出请求时，Smart Agent 一旦发现没有可用的对象实例，就会通知 OAD，而 OAD 将根据用户的设置启动 CORBA 服务器，创建、激活对象实例。

Object Activation Daemon 由 oad.exe 及 oadutil.exe 两个文件组成，前者就是运行在网络环境中的 OAD，后者负责维护、管理运行中的 OAD。

oad.exe 的执行参数如表 9.3 所示。

表 9.3　oad.exe 的执行参数

参数	说明	实例
-v	运行过程中需要详细记录各种运行信息。该日志文件可以在 VBROKER_ADM 规定目录的 Log 子目录下找到，名字为 oad.log	oad -v
-f	允许用户在同一台主机上启动多个 OAD。如果无此参数，在同一个主机上启动多个 OAD 会导致错误警告。	oad -vf oad -f
-t<n>	设定一个以秒为单位的时间参数。如果在规定时间没有成功创建、激活用户需要的对象实例，OAD 就引发一个异常，并负责关闭已经启动的 CORBA 服务器程序。当秒数为 0 时，OAD 将无休止地等待下去。默认情况下，该值为 20。	oad -vft30 oad -t60
-C	在 Windows NT 中，如果 OAD 已经被作为 NT 服务器加载，本参数允许用户以 Dos 方式启动 OAD。	oad -C
-k	如果一个进程中所有的对象实现都从 OAD 中卸载了，就应该杀死这个进程。	oad -k
-?	列举上述各参数说明	oad -?

与 VisiBroker 的一贯风格一致，OAD 也是一个“黑窗”。现在，我们可以通过 oadutil.exe 向 OAD 注册对象实现的有关信息。当用 oadutil.exe 注册对象实现时，其执行参数如表 9.4 所示：

表 9.4　oadutil.exe 注册对象实现时的执行参数

参数	说明
-i<interface name>	指定接口名称。注册对象实现时，本参数与-r<repository id>必选其一。
-r<repository id>	通过接口仓库标志号指定接口名称。 注册对象实现时，本参数与-i<interface name>必选其一。
-o<object name>	指定为对象实现创建实例时所使用的对象名称。本参数为必选内容。
-cpp<file name>	指定创建对象实例的 CORBA 服务器可执行程序的名称。本参数为必选内容。
-host<host name>	说明 OAD 所驻留远程主机的名称。本参数为可选内容。
-verbose	在运行时需要记录详细的信息。本参数为可选内容。
-d<reference data>	当激活 CORBA 服务器程序时所需的引用数据。本参数为可选内容。
-a arg1	输入启动 CORBA 服务器程序时所需的参数。如果有多个启动参数，就应该多次使用。本参数为可选内容。
-e env1	为 CORBA 服务器程序输入需要重新说明的环境变量。如果有多个环境变量，就应该多次使用。本参数为可选内容。
-p {shared |unshared |per-method}	决定 CORBA 服务器程序的启动策略，三个参数分别对应于共享服务器策略、非共享服务器策略以及单个方法服务器策略。 默认情况下为共享服务器策略。

使用 oadutil.exe 注册对象实现及 CORBA 服务器程序时，在说明上述各参数前，还应该加上关键字 reg。下面这个命令就是用来注册我们第一个 CORBA 对象实现的一个例子：

```
oadutil reg –i MyFirstCorbaObj –o MyFirstCorbaObjObject
        -cpp E:\test\CorbaTest2\Project1.exe
```

执行该命令后，在 Dos 窗口下应该显示以下一些信息：

```
Completed registration of repository_id      =    IDL:MyFirstCorbaObj:1.0
object_name          =    MyFirstCorbaObjObject
reference data       =
path_name            =    e:\test\CorbaTest2\project1.exe
activation_policy =    SHARED_SERVER
args                 =    NONE
```

```
env                    =    NONE
for OAD on host 127.0.0.1
```

这些信息说明我们已经把对象实现及 CORBA 服务器程序注册成功。但是，如果用户是在任务栏的运行窗口下执行该命令，有关信息会一闪而逝。

现在，我们可以不必事先启动 CORBA 服务器程序就直接运行 CORBA 客户程序，OAD 将在我们需要使用对象实例时自动激活 CORBA 服务器程序。

注意 用以上命令注册的 CORBA 服务器程序采用的是共享服务器策略。如果我们在局域网不同站点上启动多个 CORBA 客户，每个客户都可以与同一个 CORBA 服务器单独对话；但是，服务器程序会记录所有对象的发送的请求。另外，在这种情况下，用户需要自己负责关闭 CORBA 服务器程序：可以编程实现，也可以手工实现。为了保证服务器程序安全退出，我们应该编码通知 BOA、ORB，对于这一点，这里不再赘述。

如果我们执行下面这个命令，CORBA 服务器程序也会在需要时自动启动，不过，此时，CORBA 服务器程序将不能被共享。每个 CORBA 客户程序都会自动重新启动一个新的 CORBA 服务器程序。

```
oadutil reg -r IDL:MyFirstCorbaObj:1.0 -o MyFirstCorbaObjObject
  -cpp e:\test\CorbaTest2\Project1.exe -p unshared
```

在这个命令中，我们改用接口仓库标志号来说明接口名称。还有一点需要说明，在重新注册之前，我们应该首先取消前一次的注册，有关命令稍后再详细列举。

不难猜测，如果执行下面这个命令行，CORBA 服务器程序会采用单个方法服务器策略：CORBA 客户每发送一次请求，就会启动一个服务器程序。

```
oadutil reg -r IDL:MyFirstCorbaObj:1.0 -o MyFirstCorbaObjObject
  -cpp e:\test\CorbaTest2\Project1.exe -p per-method
```

表 9.5 oadutil.exe 注销对象实现时的执行参数

参数	说明
-i<interface name>	指定接口名称，本参数与-r<repository id>必选其一。
-r<repository id>	通过接口仓库标志号指定接口名称，本参数与-i<interface name>必选其一。
-o<object name>	指定对象名称，本参数为必选内容。
-host<host name>	说明 OAD 所驻留远程主机的名称，本参数为可选内容。
-verbose	在运行时需要记录详细的信息，本参数为可选内容。
-version	显示 oadutil 的版本号

即使在关闭 OAD 后，各种服务器注册信息也不会丢失。因此，当我们重新注册一些服务器程序信息之前，应该首先取消原来的注册。取消已有注册信息也通过 oadutil. exe 进行，这时操作关键字被改为 unreg，有关参数如表 9.5 所示。

假设我们需要取消对 MyFirstCorbaObj 的注册，执行以下命令即可：

```
oadutil unreg -r IDL:MyFirstCorbaObj:1.0 -o MyFirstCorbaObjObject
```

如果希望查看 OAD 中已经注册的内容，只需执行如下所示的命令，就会一目了然。

```
oadutil list
```

9.6.5 编写动态激发客户程序

CORBA 客户程序也完全可以采用 DII，以动态方式激发 CORBA 服务器程序。现在，我们改写前一个 CORBA 客户程序。这次，在整个工程中并没有使用任何接口存根文件。

工程项目文件 Project2.cpp 如下所示，并没有任何与 CORBA 有关的代码：

```
#include <vcl.h>
#pragma hdrstop
USERES("Project2.res");
USEFORM("Unit2.cpp", Form1);
USELIB("D:\Borland\VBROKER\lib\orb_br.lib");//注意此处添加了一些库文件
USELIB("D:\Borland\VBROKER\lib\event_br.lib");
USELIB("D:\Borland\VBROKER\lib\name_b.lib");
USELIB("D:\Borland\VBROKER\lib\name_br.lib");
USELIB("D:\Borland\VBROKER\lib\orb_b.lib");
USELIB("D:\Borland\VBROKER\lib\evchn_br.lib");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TForm1), &Form1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

客户程序的头文件 Unit2.h 也没有任何特殊代码，如下所示：

```
//---------------------------------------------------------------------------
#ifndef Unit2H
#define Unit2H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published: // IDE-managed Components
    TButton *Button1;
    TLabel *Label1;
    TLabel *Label2;
    TEdit *Edit1;
    TEdit *Edit2;
    TMemo *Memo1;
    void __fastcall Button1Click(TObject *Sender);
private:  // User declarations
public:         // User declarations
    __fastcall TForm1(TComponent* Owner);
};
```

```
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

动态激发操作的有关代码全部集中在 Unit2.cpp，这些带有解释的内容必须由用户自己编写，其中包括了通过 DII 使用 CORBA 对象的基本流程。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>//必须包括的头文件
#include "Unit2.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    //首先初始化 ORB
    CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);

    //声明返回值
    AnsiString result;

    //声明一个动态对象
    CORBA::Object_var diiObj;
    try
    {//将动态对象与规定接口绑定
    diiObj = orb->bind("IDL:MyFirstCorbaObj:1.0");

    //声明一个请求对象
    CORBA::Request_var req = diiObj->_request("TheResponse");

    //声明两个参数
    CORBA::Any ClientName_arg;
    CORBA::Any ServiceName_arg;
    //为参数赋值
    ClientName_arg <<= (const char*)Edit1->Text.c_str();
    ServiceName_arg <<= (const char*)Edit2->Text.c_str();

    //声明参数链
    CORBA::NVList_ptr args = req->arguments();
    //将参数放入参数链
```

```
    args->add_value("ClientName",ClientName_arg,CORBA::ARG_IN);
    args->add_value("ServiceName",ServiceName_arg,CORBA::ARG_IN);

     //设置返回结果类型
     req->set_return_type(CORBA::_tc_string);

          try
          {//激发操作
          req->invoke();
          //处理返回情况
          if (req->env()->exception())//动态请求中不能用常规方式获得异常
               ShowMessage("执行动态激发时发生错误");
          else
               {
               //抽取返回值
               CORBA::Any& return_value = req->return_value();
               char* val;
               return_value >>= val;
               result = val;
               }
          }
          catch(const CORBA::Exception& E)
          {
          ShowMessage("激发对象操作时发生错误");
          }
     }
     catch(const CORBA::Exception& E)
     {
     ShowMessage("获取动态对象时发生错误");
     }

     //显示结果
     Memo1->Lines->Add(result);
}
//---------------------------------------------------------------------------
```

这些代码的详细功能可以参考代码注释内容。

编译这些程序之后，我们就可以采用动态方式激发 CORBA 对象实现了。

注意 如果将动态激发、接口仓库查询、OAD 等内容结合起来，我们有可能构造一个灵活的程序，首先自动查询对象的接口、操作及参数，然后根据用户的可视化指令搭建一个临时的软件模块，执行由用户动态指定的任务。这一点，对于经常需要变化的电子商务行业应该具有诱人的前景。下面，我们将介绍 C++Builder 附带的一个 CORBA 对象万能测试工具。这个软件已经实现了自动查询接口、参数，用可视化方式搭建、测试动态任务的功能。

9.7 CORBA 对象万能测试工具

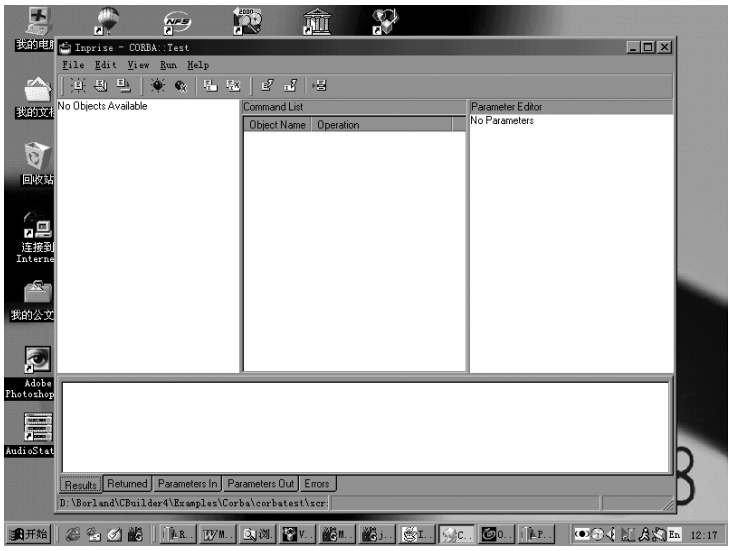

图 9.19 corbatest 的运行界面

CORBA 对象万能测试工具是 C++Builder 附带的一个工具，可以在例子程序中找到该工具的源文件，整个工程项目的名称为 corbatest.bpr，编译该工程文件后可以得到执行文件 corbatest.exe，其运行界面如图 9.19 所示。

在运行该程序之前，应该保证局域网内至少有一个 Smart Agent、接口仓库运行，如果用户不愿意手工启动 CORBA 服务器程序，还应该先运行 OAD。

执行 Edit|New Object 菜单项，可以弹出如图 9.20 所示的对话框，用来选择需要测试、使用的接口仓库标志号、接口名称以及对象引用的名称。

随后，在程序的 ObjectList 列表中会显示有关对象实例的操作、属性。如果将其中某个操作拖入 CommondList 列表框，有关参数编辑器会自动显示参数的名称，并允许用户赋值，如图 9.21 所示。

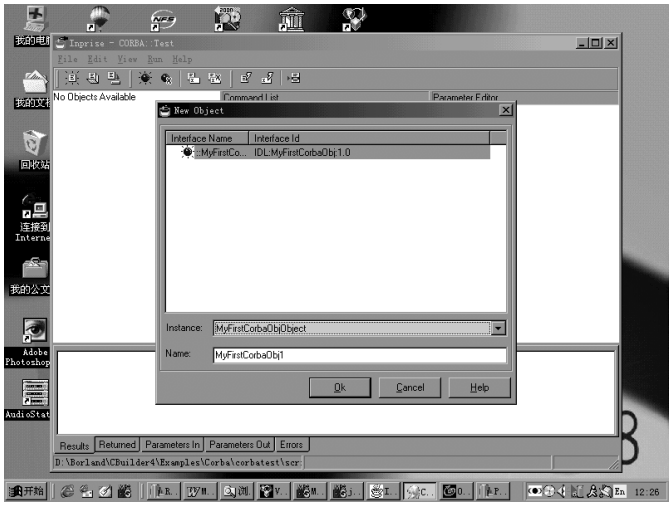

图 9.20 选择测试对象

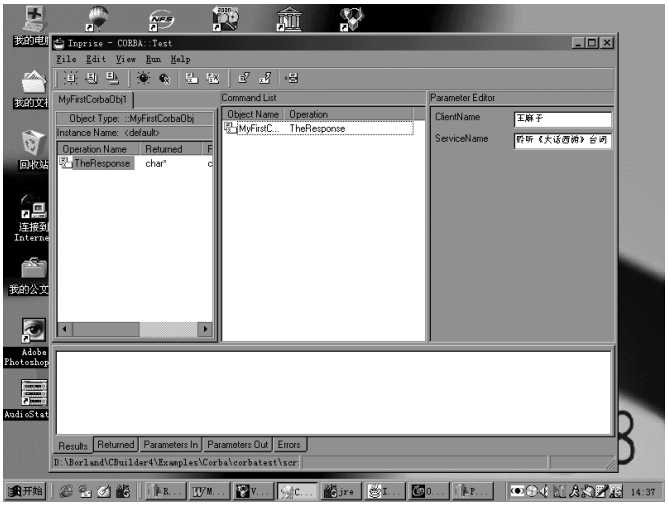

图 9.21　添加测试命令，给参数赋值

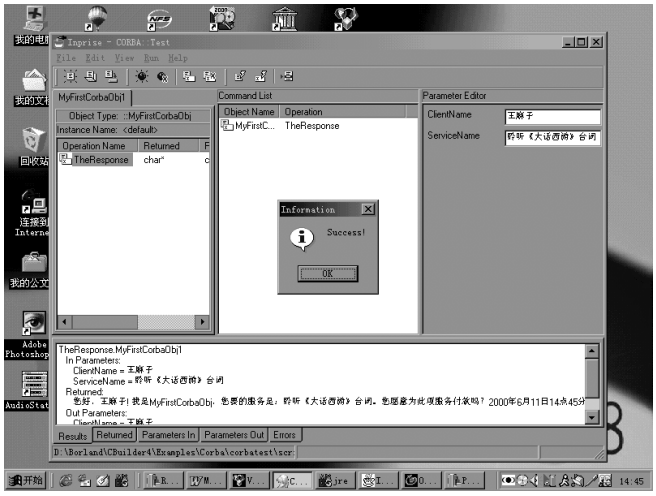

图 9.22　运行结果成功，有关信息正确返回

现在，如果我们执行 Run 下面的任何一个菜单项，都可以用这个临时搭建的指令激发 CORBA 服务器程序，其运行结果、输入参数、输出参数以及有可能发生的错误都将在程序下方的结果栏中显示，如图 9.22 所示。

由于我们成功激发了 CORBA 服务器程序，服务器程序也会改变自己的运行状态。

☎ **技巧**　上面仅仅简单说明了 corbatest 的使用方法，通过该工具，我们完全可以临时搭建一个程序模块，向我们新发现的 CORBA 对象发出请求。这个程序本身也说明，用 CORBA 技术开发的电子商务软件将完全能够适应于电子商务运行方式的千遍万化。

9.8　CORBA 管理器

VisiBroker 还带有一个内置的 CORBA 管理器，可以查看网络上 Smart Agent、接口仓库、OAD 的运行情况，获取接口仓库标志号等有关信息。这个管理器的执行文件是 osfind.exe，它的详细运行参数可以通过以下命令获得：

```
osfind  -?
```

如果仅仅键入文件名，就会以默认方式获得局域网内与 CORBA 有关的信息，下面就是一部分结果清单：

```
osfind: Found one agent at port 14000
    HOST: 127.0.0.1

osfind: Found 1 OADs in your domain
    HOST: 127.0.0.1

osfind: Following are the list of Implementations registered with OADs.
    HOST: 127.0.0.1

    REPOSITORY ID: IDL:MyFirstCorbaObj:1.0
            OBJECT NAME: MyFirstCorbaObjObject

osfind: Following are the list of Implementations started manually.
    HOST: 127.0.0.1

    REPOSITORY ID: IDL:visigenic.com/tools/ir/RepositoryManager:1.0
            OBJECT NAME: FirstIR

    REPOSITORY ID: IDL:visigenic.com/irtx/Repository:1.0
            OBJECT NAME: FirstIR

    REPOSITORY ID: IDL:visigenic.com/Activation/OAD:1.0
            OBJECT NAME: 127.0.0.1
```

另外，在 Inprise 的站点上，还可以下载到一个 osfind 的可视化版本，就是 VisiBroker Manager。

9.9 本章小结

这是本书最为冗长的一个章节，但是，它所囊括的内容却仅仅是用 C++Builder 开发基于 CORBA 的分布式软件的一些基本内容。

在 C++Builder 中，接口存根对象 Stub 及接口框架对象 Skeleton 均可以由 IDL2CPP 自动产生，其中已经封装了 CORBA 的所有实现内幕，用户仅需要按照一般 C++类来使用、理解它们即可。

我们还说明了 Smart Agent 的使用与配置、ORB 域的配置、接口仓库的使用与管理、OAD 的使用与配置。这些对象构成了 VisiBroker 所提供的 CORBA 中间件产品，而 C++Builder 提供的各种向导，无疑使得开发 CORBA 程序变的既轻松又愉快。

至于如何用 C++Builder 开发真正的基于 CORBA 的电子商务软件，恐怕至少还需要一本书的篇幅才能够说清。

第 10 章　CORBA 编程实例解析

本章内容提要：

- ATM Demo *实例解析*
- CORBA Midas *实例解析*

10.1　一个经典的电子商务演示系统——ATM demo

下面，我们首先介绍一个经典的电子商务演示系统——ATM demo。在 C++Builder 中，有一个相似的演示例子，但是，我们这里解析的例子主要借助 CORBA 向导编写，与原例有显著不同。

整个演示系统涉及到三个角色：银行、ATM 机以及客户，是一个非常接近生活的电子商务实例。

系统首先假设共有两家银行，用户可以在其中开设一定的帐户，而每个银行可以根据自己的时间上下班。

整个系统可以有一台或多台 ATM 机器，我们可以为每台 ATM 机命名。每台 ATM 机可以归属于任何一个指定的银行，但它们能够处理所有银行的交易。我们还可以根据需要关闭或启动每一台 ATM 机。

每个合法客户都有一个名字和开户银行规定的个人身份标志号 PIN，用户可以选定自己希望使用的 ATM 机名称，通过 ATM 机，客户可以对自己的帐户进行查询、支出或存款，也可以对自己的支票进行查询、提款或拨款。

银行、ATM 机将检查每一笔交易的可行性，并忠实记录每一笔成功的交易。

这个电子商务演示系统由 4 个工程项目（exe）组成，共同归属于 atmdemo 工程组。工程组可以通过 C++Builder 的 New 命令创建，选择“Project Group”图标项即可。

10.1.1　编写 IDL 接口

与所有 CORBA 程序一样，在分析完这个电子商务系统以后，我们应该开始编写有关的 IDL 接口。本系统中共定义了两个接口文件，分别如下所示：

例10.1　xaction.idl 接口定义文件及其说明。

```
enum EnumAction {//定义三种常见操作
   balance,//查询
   withdraw,//支出
   deposit//存款
};

enum EnumStatus {//定义操作执行状态
   OK,//成功
   invalid,//无效
   complete,//完成
   incomplete//未完成
};

struct xaction {//定义一个结构体，记录一笔交易的有关内容
```

```
    long UserID;//用户 ID
    long PIN;//用户个人身份标志号
    long account;//帐号
    double amount;//操作金额
    ::EnumAction action;//操作类型，根据定义，共有三种可能
    double balance;//现有余额
    ::EnumStatus status;//操作执行状态
};

interface Server {//定义 ATM 机服务接口
    ::xaction HandleTransaction(//定义交易处理函数，返回类型为交易结构体
        inout ::xaction Transaction//参数为交易结构体，方向属性为 inout
    );
};
```

在这个接口文件中定义的操作枚举类型、操作执行状态枚举类型、交易结构体以及 Server 接口实际上是对 ATM 机的一种抽象表示。

例10.2　banking.idl 接口定义文件及其说明。

```
#include "xaction.idl"//引用 xaction.idl 中定义的内容

interface BankServer {//银行中与 ATM 机相关的服务接口
    ::xaction HandleTransaction(//定义交易处理函数，返回类型为交易结构体
        inout ::xaction Transaction//参数为交易结构体，方向属性为 inout
    );

    long BankID();//返回所在银行 ID
};
```

这个接口文件主要用来抽象表达“银行”。当然，这里只关心银行中与 ATM 机相关的服务。

10.1.2　第一个银行的 CORBA 实现

我们用 BankServer.bpr 来完成第一个银行的 CORBA 实现。在 atmdemo 工程组中执行 New，创建一个叫做 BankServer 的项目，并将有关的 cpp 文件保存为 bankunit.cpp。接着，我们可以把 banking.idl 文件加入该项目，将主窗体更名为 BankForm。在陆续添加一系列控件后，我们应该得到如图 10.1 中小窗体所示的界面。

除引用了接口定义文件外，BankServer.cpp 文件与常规的项目执行文件几乎一致。

例10.3　BankServer.cpp 文件内容。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("BankServer.res");
USEFORM("bankunit.cpp", BankForm);
USEIDL("banking.idl");
USEIDL("xaction.idl");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
```

```
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TBankForm), &BankForm);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

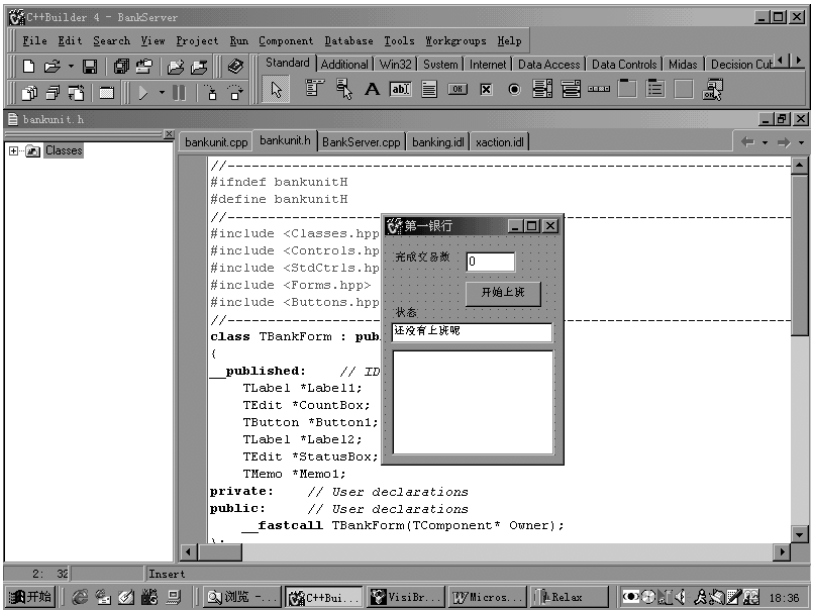

图 10.1　设计中的第一个银行的界面

在编译 IDL 文件之前，我们清除了 CORBA Generation 中的所有选项，这样可以得到比较简洁的接口框架文件。由于接口框架文件内容大同小异，本章不再列举。

如第 9 章所述，我们可以采用 CORBA 对象实现向导来简化开发任务。在这里，我们首先实现 baking.idl 中的 BankServer 接口，并将有关内容仍旧存放在 bankunit.cpp 中，对象的名称被定为 BankOne。与往常一样，“向导”会为我们创建一些必要的代码。

在 bankunit.h 头文件中，CORBA 对象实现向导为我们添加了下面一些代码。

例10.4　bankunit.h 头文件中新添加的代码。

```
class BankServerImpl: public _sk_BankServer
{
protected:
public:
    BankServerImpl(const char *object_name=NULL);
    CORBA::Long BankID();
    xaction HandleTransaction(xaction& Transaction);
};
```

在 bankunit.cpp 文件中，CORBA 对象实现向导为我们添加了下面一些代码。

例10.5　bankunit.cpp 文件中新添加的代码。

```
#pragma hdrstop

#include <corba.h>
……

BankServerImpl::BankServerImpl(const char *object_name):
    _sk_BankServer(object_name)
{
}

CORBA::Long BankServerImpl::BankID()
{
}

xaction BankServerImpl::HandleTransaction(xaction& Transaction)
{
}
```

另外，在 BankServer.cpp 中，CORBA 对象实现向导为我们添加了下面一些“激活”代码。在实际应用中，我们需要将这些代码移动到更加合适的地方。关于这一点，稍后再详细说明。

例10.6　BankServer.cpp 中新添加的“激活”代码。

```
#include <corba.h>
……
        // Initialize the ORB and BOA
        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
        CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
        BankServerImpl bankServer_BankOne("BankOne");
        boa->obj_is_ready(&bankServer_BankOne);
```

显然，对于本程序而言，编写 CORBA 对象实现的主要任务是完成 bankunit.cpp 中列出的函数框架。为了处理有关任务，我们在 bankunit.h 头文件的 BankServerImpl 类中进一步引入了以下一些变量及函数声明：

例10.7　在 BankServerImpl 类中引入的新变量、函数及其说明。

```
private:
    long BankNumber;//银行 ID
    int count;//交易数目
    int UserCount;//用户数目
    struct _user{//用户结构体
      long UserID;//用户 ID
      double AccountChecking;//支票存额
      double AccountSavings;//资金存额
    };

_user Customers[3];//预留的 3 个用户

public:
```

```
void __fastcall UpCount(int i = 0);//更新界面的函数
```

这些新引入的变量、函数以及接口框架函数的编码如下所示。

例10.8　接口框架函数及有关函数的代码和说明。

```
BankServerImpl::BankServerImpl(const char *object_name):
      _sk_BankServer(object_name)
{     //对象实例构造函数
      BankNumber = 900;//设置银行 ID
      count = 0;//初始化交易数目
      UserCount = 2;//初始化用户数目

      Customers[0].UserID = 0;//用户 John
      Customers[0].AccountChecking =    2000.00;
      Customers[0].AccountSavings =     90.0;

      Customers[1].UserID = 1;//用户 Bill
      Customers[1].AccountChecking = 100.00;
      Customers[1].AccountSavings =    500.00;

      Customers[2].UserID = 2;//用户 Jim
      Customers[2].AccountChecking = 1000.00;
      Customers[2].AccountSavings =    5000.00;
}

CORBA::Long BankServerImpl::BankID()
{
      return BankNumber;//返回银行 ID
}

xaction BankServerImpl::HandleTransaction(xaction& Transaction)
{
      bool PostChange = false;//用来记录资金或支票额是否变化的变量
      double workingBalance = 0;//工作变量
      AnsiString type[]={"查询","提款","存款"};
AnsiString kind[]={"支票","资金"};

      UpCount();//更新界面
      Transaction.status = incomplete;//交易尚未完成
BankForm->Memo1->Lines->Add(
                  "用户 ID " + String(Transaction.UserID));
BankForm->Memo1->Lines->Add(
              "交易类型  " + type[Transaction.action]);
BankForm->Memo1->Lines->Add(
              "交易对象  " + kind[Transaction.account]);
      for (int i = 0; i < UserCount; i++)
            {
            if(Transaction.UserID == Customers[i].UserID)//身份验证
                  {
                  if (Transaction.account == 0)//交易对象为现金还是支票
```

```
                workingBalance = Customers[i].AccountChecking;
            else
                workingBalance = Customers[i].AccountSavings;

            switch(Transaction.action)//选择交易类型
            {
            case balance://查询
            Transaction.balance = workingBalance;//更新余额
            Transaction.status = complete;//交易结束
            break;

            case withdraw://提款
            Transaction.balance = workingBalance;//暂存余额
            workingBalance -= Transaction.amount;//尝试结帐
            if (workingBalance < 0)//检查交易是否合理
                {
                Transaction.status = incomplete;//不合理则交易未结束
                }
            else
                {
                Transaction.balance = workingBalance;//更新余额
                Transaction.status = complete;//交易结束
                PostChange = true;//金额发生变化
                }
            break;

            case deposit://存款
            workingBalance += Transaction.amount;//结帐
            Transaction.balance = workingBalance;//更新余额
            Transaction.status = complete;//交易结束
            PostChange = true;//金额发生变化
            break;
            }

            if (PostChange)//更新帐户
                {
                if (Transaction.account == 0)
                    Customers[i].AccountChecking = workingBalance;
                else
                    Customers[i].AccountSavings    = workingBalance;
                }
            }
        }
    return Transaction;
}

void __fastcall BankServerImpl::UpCount(int i)
{
    //根据用户要求，更新界面
    if (i ==1)
```

```
            BankForm->StatusBox->Text = "第一银行正在上班";
        else
            BankForm->CountBox->Text = AnsiString(++count);
}
```

其实，到这里为止，第一银行的 CORBA 对象实现已经完成了。

在现实生活中，每个银行都有自己的工作时间，因此，我们在界面中特地设置了一个“开始上班”按钮，以便用户在需要的时候才激活有关的对象实例。当然，这时需要移动由 CORBA 对象实现向导自动添加的实例激活代码。在本程序中，我们将这些代码移入了一个子线程对象，具体情况如下所示。

这个子线程对象可以通过执行 New 命令获得，名称为 BankThread。

例10.9　子线程 BankThread 的头文件。

```
//---------------------------------------------------------------------------
#ifndef bankthreadH
#define bankthreadH
//---------------------------------------------------------------------------
#include <Classes.hpp>
//---------------------------------------------------------------------------
class BankThread : public TThread
{
private:
protected:
    void __fastcall Execute();
public:
    __fastcall BankThread(bool CreateSuspended);
};
//---------------------------------------------------------------------------
#endif
```

由 CORBA 对象实现向导生成的实例激活代码也被我们移到了子线程 BankThread 的执行函数 Execute 中。

例10.10　子线程 BankThread 的执行函数及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "bankthread.h"
#include <corba.h>//引用了 CORBA.h 头文件
#include "bankunit.h"//引用了 bankunit.h 头文件
#pragma package(smart_init)
//---------------------------------------------------------------------------
__fastcall BankThread::BankThread(bool CreateSuspended)
    : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall BankThread::Execute()
{
    //---- Place thread code here ----
    try
```

```
    {
    // Initialize the ORB and BOA
    CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
    CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
    BankServerImpl bankServer_BankOne("BankOne");
boa->obj_is_ready(&bankServer_BankOne);
bankServer_BankOne.UpCount(1);//修改界面
boa->impl_is_ready();//通知 BOA 服务器已经激活
    }
    catch(const CORBA::Exception& e)
    {
    ShowMessage("激活第一银行 CORBA 服务时出错");
    }
}
//---------------------------------------------------------------------------
```

最后，将子线程 BankThread 的头文件包括在 bankunit.cpp 中，并给点击“开始上班”按钮添加如下所示两行代码，则第一银行的代码全部编写完毕。

例10.11　点击“开始上班”按钮的事件函数。

```
void __fastcall TBankForm::Button1Click(TObject *Sender)
{
    if (StatusBox->Text == "还没有上班呢")
        BankThread * work = new BankThread(false);
}
```

10.1.3　第二个银行的 CORBA 实现

第二个银行的 CORBA 实现与第一个银行的 CORBA 实现几乎一致，主要差别在于对象实例的名称以及银行客户不同。第二个银行 CORBA 实现的工程名称为 BankServer2.bpr，其余文件也是在第一银行 CORBA 对象实现有关文件后加上“2”形成，如 bankunit2.cpp、bankthread2.cpp 等。这些文件的详细内容如下所示：

例10.12　BankServer2.cpp 的全部内容。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "bankunit2.h"

USERES("BankServer2.res");
USEFORM("bankunit2.cpp", Bank2Form); /* banking.idl: CORBAIdlFile */
USEIDL("banking.idl");//使用的 IDL 文件相同
USEIDL("xaction.idl");
USEUNIT("banking_c.cpp");//在 CORBA 对象实现中会自动包括有关的
USEUNIT("banking_s.cpp");//接口存根、接口框架文件
USEUNIT("xaction_c.cpp");
USEUNIT("xaction_s.cpp");
USEUNIT("bankthread2.cpp");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
```

```
    {
        Application->Initialize();
        Application->CreateForm(__classid(TBank2Form), &Bank2Form);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

例10.13　bankunit2.h 头文件的全部内容。

```
//---------------------------------------------------------------------------
#ifndef bankunit2H
#define bankunit2H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "banking_s.hh"
//---------------------------------------------------------------------------
class TBank2Form : public TForm
{
__published: // IDE-managed Components
    TLabel *Label1;
    TEdit *CountBox;
    TButton *Button1;
    TLabel *Label2;
    TEdit *StatusBox;
    TMemo *Memo1;
    void __fastcall Button1Click(TObject *Sender);
private:  // User declarations
public:         // User declarations
    __fastcall TBank2Form(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TBank2Form *Bank2Form;
//---------------------------------------------------------------------------
class BankServerImpl: public _sk_BankServer
{//也使用 BankServerImpl 类来实现接□框架
private:
    long BankNumber;
    int count;
    int UserCount;
    struct _user{
       long UserID;
       double AccountChecking;
       double AccountSavings;
    };
```

```
    _user Customers[3];
protected:
public:
    BankServerImpl(const char *object_name=NULL);
    CORBA::Long BankID();
xaction HandleTransaction(xaction& Transaction);
    void __fastcall UpCount(int i = 0);
};
#endif
```

注意　在相同工程组的不同工程项目中，我们可以采用相同名称的类来实现同一个接口框架；相反，在同一个工程项目中，我们也可以采用不同名称的类来完成同一个接口框架的不同实现。

例10.14　bankunit2.cpp 文件的全部内容。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>
#include "bankunit2.h"
#include "bankthread2.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TBank2Form *Bank2Form;
//---------------------------------------------------------------------------
__fastcall TBank2Form::TBank2Form(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------

BankServerImpl::BankServerImpl(const char *object_name):
    _sk_BankServer(object_name)
{
    BankNumber = 800;//设置银行 ID
    count = 0;
    UserCount = 2;

    Customers[0].UserID = 3;//用户 Gene
    Customers[0].AccountChecking =    92000.00;
    Customers[0].AccountSavings =     990.0;

    Customers[1].UserID = 4;//用户 Wally
    Customers[1].AccountChecking = 9100.00;
    Customers[1].AccountSavings =    9500.00;

    Customers[2].UserID = 5;//用户 Frank
    Customers[2].AccountChecking = 91000.00;
    Customers[2].AccountSavings =    95000.00;
```

```
}

CORBA::Long BankServerImpl::BankID()
{
      return BankNumber;
}

xaction BankServerImpl::HandleTransaction(xaction& Transaction)
{
      bool PostChange = false;
      double workingBalance = 0;
      AnsiString type[]={"查询","提款","存款"};
AnsiString kind[]={"支票","资金"};

      UpCount();
      Transaction.status = incomplete;
Bank2Form->Memo1->Lines->Add(
                              "用户 ID " + String(Transaction.UserID));
Bank2Form->Memo1->Lines->Add(
                              "交易类型  " + type[Transaction.action]);
Bank2Form->Memo1->Lines->Add(
                              "交易对象  " + kind[Transaction.account]);
      for (int i = 0; i < UserCount; i++)
            {
            if(Transaction.UserID == Customers[i].UserID)
                  {
                  if (Transaction.account == 0)
                        workingBalance = Customers[i].AccountChecking;
                  else
                        workingBalance = Customers[i].AccountSavings;

                  switch(Transaction.action)
                  {
                  case balance:
                  Transaction.balance = workingBalance;
                  Transaction.status = complete;
                  break;

                  case withdraw:
                  Transaction.balance = workingBalance;
                  workingBalance -= Transaction.amount;
                  if (workingBalance < 0)
                        {
                        Transaction.status = incomplete;
                        }
                  else
                        {
                        Transaction.balance = workingBalance;
                        Transaction.status = complete;
                        PostChange = true;
                        }
                  break;
```

```
                case deposit:
                workingBalance += Transaction.amount;
                Transaction.balance = workingBalance;
                Transaction.status = complete;
                PostChange = true;
                break;
                }

                if (PostChange)
                    {
                    if (Transaction.account == 0)
                        Customers[i].AccountChecking = workingBalance;
                    else
                        Customers[i].AccountSavings    = workingBalance;
                    }
                }
            }
    return Transaction;
}

void __fastcall BankServerImpl::UpCount(int i)
{
    //TODO: Add your source code here
    if (i == 1)
        Bank2Form->StatusBox->Text = "第二银行正在上班";
    else
        Bank2Form->CountBox->Text = AnsiString(++count);
}

void __fastcall TBank2Form::Button1Click(TObject *Sender)
{
    if (StatusBox->Text == "还没有上班呢")
        BankThread2 * work = new BankThread2(false);
}
//---------------------------------------------------------------------------
```

例10.15　bankhread2.h 头文件的全部内容。

```
//---------------------------------------------------------------------------
#ifndef bankthread2H
#define bankthread2H
//---------------------------------------------------------------------------
#include <Classes.hpp>
//---------------------------------------------------------------------------
class BankThread2 : public TThread
{
private:
protected:
    void __fastcall Execute();
public:
    __fastcall BankThread2(bool CreateSuspended);
};
```

```
//---------------------------------------------------------------------------
#endif
```

例10.16　bankhread2.cpp 文件的全部内容。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>
#include "bankunit2.h"
#include "bankthread2.h"
#pragma package(smart_init)
__fastcall BankThread2::BankThread2(bool CreateSuspended)
    : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall BankThread2::Execute()
{
    //---- Place thread code here ----
    try
    {
    // Initialize the ORB and BOA
    CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
    CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
    BankServerImpl bankServer_BankTwo("BankTwo");
boa->obj_is_ready(&bankServer_BankTwo);
bankServer_BankTwo.UpCount(1);
boa->impl_is_ready();
    }
    catch(const CORBA::Exception& e)
    {
    ShowMessage("激活第二银行 CORBA 服务时出错");
    }
}
//---------------------------------------------------------------------------
```

当然，第二银行 CORBA 实现的建立步骤与第一银行 CORBA 实现的建立步骤是完全一致的。

10.1.4　ATM 服务器

ATM 服务器既是 ATM 接口的 CORBA 实现，也是银行接口的 CORBA 客户。作为 ATM 接口的 CORBA 实现，它应该包括 ATM 接口框架 Skeleton 及 ATM 接口存根 Stub；作为银行接口的 CORBA 客户，它应该包括银行接口存根 Stub。因此，我们在工程组 atmdemo 中创建 ATM 服务器项目 AtmServer.bpr 时，也需要加入两个接口文件。

ATM 服务器的界面如图 10.2 中小窗体所示。用户可以在名称编辑框中设置本 ATM 机的名称；在所属银行下拉框中选择本 ATM 机归属的银行；通过“ID”按钮获取本 ATM 机所属银行的 ID 号；通过“启动 ATM 服务”以及“关闭 ATM 服务”可以开启、关闭本 ATM 机。

在使用 CORBA 对象实现向导时，将类名称设置为 AtmServerImpl，不要求向导自动

创建对象实例，可以获得以下一些文件。

例10.17　atmunit.h 头文件中自动添加的代码。

```
#include "xaction_s.hh"
……
class AtmServerImpl: public _sk_Server
{
protected:
public:
AtmServerImpl(const char *object_name=NULL);
      xaction HandleTransaction(xaction& Transaction);
};
```

例10.18　atmunit.cpp 文件中自动添加的代码。

```
#include <corba.h>
……

AtmServerImpl::AtmServerImpl(const char *object_name):
      _sk_Server(object_name)
{
}

xaction AtmServerImpl::HandleTransaction(xaction& Transaction)
{
}
```

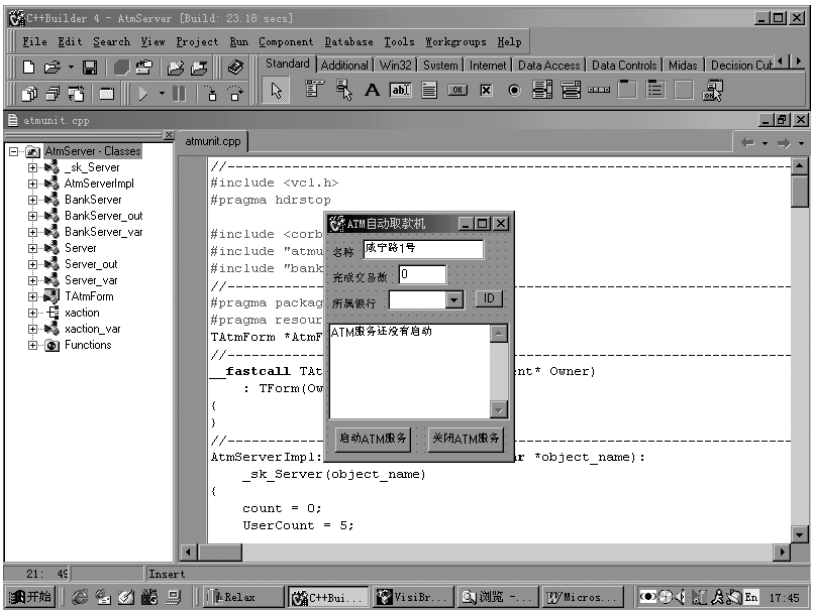

图 10.2　设计中的 ATM 服务器界面

例10.19　AtmServer.cpp 文件中新添加的代码。

```
#include <corba.h>

            // Initialize the ORB and BOA
```

```
CORBA::ORB_var orb = CORBA::ORB_init(    argc,    argv);
CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
```

为了实现接口函数，我们在 atmunit.h 头文件中引入了以下一些变量及函数声明。

例10.20　AtmServerImpl 中新添的变量声明及函数声明。

```
private:
    int count;//交易数
    struct _user{//持卡用户
        long UserID;//持卡用户 ID
        long Bank;//开户银行 ID
        long PIN;//用户 IC 卡 ID
    };

   _user CardHolders[6];//用来记录持卡者的数组
   int UserCount;//有效用户数目

public:
void __fastcall UpCount(int i = 0);//更新 ATM 机服务器界面
……
```

例10.21　TAtmForm 中新添的变量声明及事件声明。

```
__published: // IDE-managed Components
    void __fastcall Button1Click(TObject *Sender);//启动按钮点击事件
    void __fastcall Button2Click(TObject *Sender);//关闭按钮点击事件
void __fastcall Button3Click(TObject *Sender);//银行 ID 按钮点击事件
//关闭主窗体事件
void __fastcall FormClose(TObject *Sender, TCloseAction &Action);

public:    // User declarations

    CORBA::ORB_var orb;//ATM 机中的 ORB 对象
    CORBA::BOA_var boa;//ATM 机中的 BOA 对象
    CORBA::Object_ptr atm;//被 ATM 机引用的 CORBA 对象
```

有关函数的详细编码如下所示。

例10.22　AtmServerImpl 的函数编码及其说明。

```
AtmServerImpl::AtmServerImpl(const char *object_name):
    _sk_Server(object_name)
{
    count = 0;//初始化交易数目
    UserCount = 5;//初始化用户数目

    AtmForm->BankBox->Items->Clear();//设置所属银行选项
    AtmForm->BankBox->Items->Add("第一银行");
    AtmForm->BankBox->Items->Add("第二银行");
AtmForm->BankBox->Text = "第一银行";

    CardHolders[0].UserID = 0; //持卡用户 John
```

```
    CardHolders[0].PIN = 123;
    CardHolders[0].Bank = 900;

    CardHolders[1].UserID = 1;//持卡用户 Bill
    CardHolders[1].PIN = 456;
    CardHolders[1].Bank = 900;

    CardHolders[2].UserID = 2; //持卡用户 Jim
    CardHolders[2].PIN = 789;
    CardHolders[2].Bank = 900;

    CardHolders[3].UserID = 3; //持卡用户 John
    CardHolders[3].PIN = 123;
    CardHolders[3].Bank = 800;

    CardHolders[4].UserID = 4;//持卡用户 Bill
    CardHolders[4].PIN = 456;
    CardHolders[4].Bank = 800;

    CardHolders[5].UserID = 5; //持卡用户 Jim
    CardHolders[5].PIN = 789;
    CardHolders[5].Bank = 800;
}

xaction AtmServerImpl::HandleTransaction(xaction& Transaction)
{
    UpCount();

    Transaction.status = invalid;
    for (int i = 0; i < UserCount; i++)
        if(Transaction.UserID == CardHolders[i].UserID)
            if (Transaction.PIN == CardHolders[i].PIN)//鉴别用户
            {
                Transaction.status = OK;
                try
                {
                BankServer_var BankServer;//声明一个临时银行引用
                AnsiString BankName;//声明一个临时字符串

                if (CardHolders[i].Bank == 800)//与有关 ID 的银行绑定
                    {
                    BankServer = BankServer::_bind("BankTwo");
                    BankName = "第二银行";
                    }

                if (CardHolders[i].Bank == 900) //与有关 ID 的银行绑定
                    {
                    BankServer = BankServer::_bind("BankOne");
                    BankName = "第一银行";
                    }
```

```
                    BankServer->HandleTransaction(Transaction);//激发服务
                    AtmForm->Memo1->Lines->Add("参与交易的为"
                                                + BankName);
                    }
                    catch(CORBA::Exception &e)
                    {
                    AtmForm->Memo1->Lines->Add("**有关银行没上班**");
                    return Transaction;
                    }
                }
    return Transaction;
}

void __fastcall AtmServerImpl::UpCount(int i)
{
    //TODO: Add your source code here
    if (i == 1)//根据用户要求，更新有关界面
        AtmForm->Memo1->Lines->Add("**本 ATM 机正在服务**");
    else
        AtmForm->Count->Text = String(++count);
}
```

由 CORBA 对象实现向导自动编写的 ORB、BOA 初始化代码也被移到了启动按钮事件函数中。

例10.23　TAtmForm 中有关事件函数代码。

```
void __fastcall TAtmForm::Button1Click(TObject *Sender)
{    //启动 ATM 按钮点击事件函数
    try
    {
    orb = CORBA::ORB_init(__argc, __argv);//将向导生成的代码移动至此
    boa = orb->BOA_init(__argc, __argv);
    if (atm)//判断 ATM 对象是否为空
        {
        ServerNameEdit->Text = atm->_object_name();//返回当前名称
        }
    else
        {
        atm = new AtmServerImpl(ServerNameEdit->Text.c_str());
        boa->obj_is_ready(atm);//创建、注册指定名称的对象实例
        }
        Memo1->Lines->Add("**ATM 机已经启动**");//更新界面
    }
    catch(const CORBA::Exception& e)
    {
    ShowMessage("启动 ATM 服务时出错");
    }
}
//---------------------------------------------------------------------------
void __fastcall TAtmForm::Button2Click(TObject *Sender)
{    //关闭 ATM 服务按钮点击事件函数
```

```
    try
    {
    if (atm)//判断 ATM 对象是否为空
        {
        boa->deactivate_obj(atm);//冻结 ATM 对象
        atm->_release();//释放资源
        atm = atm->_nil();//清空对象指针
        }
    Memo1->Lines->Add("**ATM 机已关闭**");//更新界面
    }
    catch(CORBA::Exception &e)
    {
    ShowMessage("关闭 ATM 服务时出错");
    }

}
//---------------------------------------------------------------------------
void __fastcall TAtmForm::Button3Click(TObject *Sender)
{
    try
    {
    BankServer_ptr BankServer;//声明对银行对象的临时引用
    if   (BankBox->Text == "第一银行")//与有关 ID 的银行对象绑定
        BankServer = BankServer::_bind("BankOne");
    else
        BankServer = BankServer::_bind("BankTwo");

Memo1->Lines->Add("本机银行 ID 为"
            + String(BankServer->BankID()));//激发银行对象有关服务
    }
    catch(CORBA::Exception &e)
    {
    ShowMessage(e._name());
    }
}
//---------------------------------------------------------------------------
void __fastcall TAtmForm::FormClose(TObject *Sender, TCloseAction &Action)
{
    Button2Click(Sender);//首先关闭 ATM 服务
}
//---------------------------------------------------------------------------
```

在应用中，我们可以启动多个 ATM 服务器，并为每台 ATM 机设置不同的名称。

10.1.5　ATM 客户程序

ATM 客户程序是这个电子商务演示系统的一个重要组成部分。通过 ATM 客户程序的界面，我们可以选择需要使用的 ATM 机，执行有关的业务，获取 ATM 服务。

ATM 客户程序通过 atmdemo 工程组中的 AtmClient.bpr 工程实现。ATM 客户程序是一个 ATM 机 CORBA 对象的客户，因此，仅仅需要使用有关接口存根即可。整个 ATM 客户程序的界面如图 10.3 所示。

用户可以通过“用户清单”按钮获得演示系统中有效用户信息，通过“选择 ATM 机”按钮选取一台指定的 ATM 机完成有关服务；通过“用户名称”、“用户身份标志号”、“交易额”编辑框输入服务信息；通过“操作类型”、“帐户类型”确定具体需要的 ATM 服务；通过“执行”按钮获得有关的 ATM 服务。

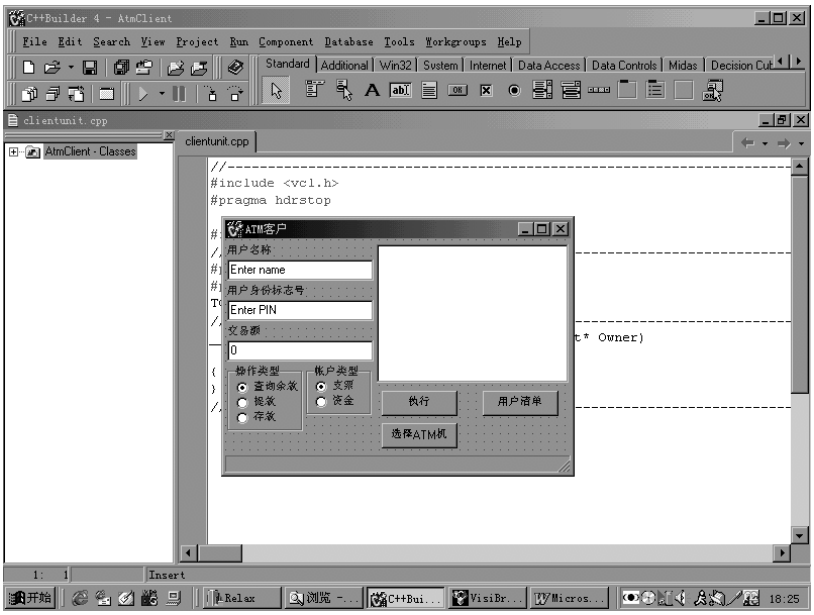

图 10.3　设计中的 ATM 客户界面

☎ **技巧**　在继续编写 ATM 客户程序之前，我们最好首先执行 Edit|Use CORBA Object 菜单项。当然，在更新项目文件时，我们可以仅仅要求 CORBA 客户向导为我们添加上适当的“头文件”即可。如果不这样做，我们手工编写的 CORBA 客户程序可能会出现链接错误。万一出现了链接错误，应该将 VisiBroker 安装目录 Lib 子目录下的所有库文件加入工程项目后再进行编译。实际上，在使用各种 CORBA 向导时，C++Builder 会自动将有关库文件信息写在项目的编译文件 makefile 中。

在 ATM 客户程序主窗体 TClientForm 中，我们添加了以下一些变量及函数声明。

例10.24　TClientForm 类中新添加的变量、函数声明及其说明。

```
//---------------------------------------------------------------------------
#ifndef clientunitH
#define clientunitH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
//---------------------------------------------------------------------------
class TClientForm : public TForm
{
__published: // IDE-managed Components
    TLabel *Label1;
```

```
    TEdit *NameBox;//用于输入客户名称
    TLabel *Label2;
    TEdit *PINBox;//用于输入客户 PIN
    TLabel *Label3;
    TEdit *AmountBox;//用于输入交易额
    TRadioGroup *Account;//用于选择帐户类型
    TRadioGroup *Action;//用于选择操作类型
    TMemo *Memo1;
    TButton *ExecuteButton;//执行按钮
    TButton *ListButton;//用户清单按钮
    TButton *AtmButton;//选择 ATM 机按钮
    TStatusBar *StatusBar;//状态条
    void __fastcall ListButtonClick(TObject *Sender);//列举清单
    void __fastcall ExecuteButtonClick(TObject *Sender);//执行服务
    void __fastcall AtmButtonClick(TObject *Sender);//选择 ATM 机
private:
    AnsiString AtmName;//当前选择的 ATM 机
    long __fastcall UserID(void);//获取用户 ID
    long __fastcall PIN(void);//获取 PIN
    double __fastcall Amount(void);//获取交易额
    long __fastcall AccountType(void);//获取帐户类型
    long __fastcall ActionType(void);//获取操作类型
public:         // User declarations
    __fastcall TClientForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TClientForm *ClientForm;
//---------------------------------------------------------------------------
#endif
```

例10.25　TClientForm 中各函数及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop//由 CORBA 向导自动添加
#include <corba.h>//由 CORBA 向导自动添加
#include "xaction_c.hh"//由向导自动添加，从 clientunit.h 中移至此
#include "clientunit.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TClientForm *ClientForm;
//---------------------------------------------------------------------------
__fastcall TClientForm::TClientForm(TComponent* Owner)
    : TForm(Owner)
{
    AtmName = "咸宁路 1 号";//设置默认的 ATM 机名称
}
//---------------------------------------------------------------------------
```

```
void     fastcall TClientForm::ListButtonClick(TObject *Sender)
{
      Memo1->Lines->Clear();//列举用户清单
      Memo1->Lines->Add("User: John PIN:123");
      Memo1->Lines->Add("User: Bill PIN:456");
      Memo1->Lines->Add("User: Jim PIN:789");
      Memo1->Lines->Add("User: Gene PIN:123");
      Memo1->Lines->Add("User: Wally PIN:456");
      Memo1->Lines->Add("User: Frank PIN:789");
}
//---------------------------------------------------------------------------
void __fastcall TClientForm::ExecuteButtonClick(TObject *Sender)
{
    xaction_var Task = new xaction;//创建一个交易任务

    //Poplulate transaction object
    Task->UserID    = UserID();//初始化有关交易任务中的各个参数
    Task->PIN    = PIN();
    Task->amount    = Amount();
    Task->action    = (EnumAction)ActionType();
    Task->account = AccountType();
    Task->balance = 0.00;
    Task->status = incomplete;

    try
    {
    // Initialize the ORB and BOA
    //下面这行代码可以由向导添加移动到此，也可以手工添加
    CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
    StatusBar->SimpleText = "**初始化 ORB**";

    //与指定名称的 ATM 服务机绑定
    Server_ptr AtmServer = Server::_bind(AtmName.c_str());
    StatusBar->SimpleText = "**已经与 ATM 机连接成功**";

    //激发 ATM 服务
    AtmServer->HandleTransaction(Task);

    switch(Task->status)//判断执行结果
     {
     case    complete://交易完成
     Memo1->Lines->Add("余额为: $" + CurrToStr(Task->balance));
     Memo1->Lines->Add("本次交易完成");
     break;

     case incomplete://交易没有完成
     Memo1->Lines->Add("本次交易没有完成");
     Memo1->Lines->Add("余额为: $" + CurrToStr(Task->balance));
     break;
```

```
      case invalid://无法进行交易
      Memo1->Lines->Clear();
      Memo1->Lines->Add("用户 ID 与 PIN 不匹配");
      break;
      }
    }
    catch(const CORBA::Exception& e)
    {
    ShowMessage(e._name());
    StatusBar->SimpleText = "**ATM 机可能已关闭**";
    }
}
//---------------------------------------------------------------------------

void __fastcall TClientForm::AtmButtonClick(TObject *Sender)
{
      //修改选定的 ATM 机的名称
      AtmName = InputBox("选择需要使用的 ATM 机",
                         "请输入 ATM 服务器的名称","咸宁路 1 号");
}
//---------------------------------------------------------------------------

long __fastcall TClientForm::UserID(void)
{
      //TODO: Add your source code here
      if (NameBox->Text == "John")//返回有关用户 ID
            return 0;
      if (NameBox->Text == "Bill")
            return 1;
      if (NameBox->Text == "Jim")
            return 2;
      if (NameBox->Text == "Gene")
            return 3;
      if (NameBox->Text == "Wally")
            return 4;
      if (NameBox->Text == "Frank")
            return 5;
      return -1;
}

long __fastcall TClientForm::PIN(void)
{
      //TODO: Add your source code here
      return (long)PINBox->Text.ToIntDef(-1);//返回用户 PIN
}

double __fastcall TClientForm::Amount(void)
{
      //TODO: Add your source code here
      try
      {
```

```
    return AmountBox->Text.ToDouble();//返回用户交易额
    }
    catch (Exception &e)
    {
    ShowMessage("交易额必须为有效双精度数");
    return 0.0;
    }
}

long __fastcall TClientForm::AccountType(void)
{
    //TODO: Add your source code here
    return (long)Account->ItemIndex; //返回帐户类型
}

long __fastcall TClientForm::ActionType(void)
{
    //TODO: Add your source code here
    return (long)Action->ItemIndex;//返回操作类型
}
```

现在，整个 ATM 电子商务演示系统已经编写完毕。

注意　在使用 C++Builder 开发 CORBA 客户程序时，不一定需要获得 BOA 对象。本程序就说明了这一点。

10.1.6　执行 ATM 电子商务演示系统

图 10.4 是运行中的 ATM 电子商务演示系统。与所有 CORBA 程序一样，在局域网内应该至少运行一个 Smart Agent。整个局域网内银行最多可以有两个，第一银行或第二银行；ATM 机可以有无数多个，分别属于某一个银行；客户也可以有无数多个，并可以自由选择希望使用的 ATM 机。本图中为两个银行、两个 ATM 机、一个客户。

在上述演示过程中，被称为“咸宁路 1 号”的 ATM 机隶属于第一银行；“咸宁路 2 号”ATM 机则隶属于第二银行；客户既可以选择使用“咸宁路 1 号”ATM 机获取服务，也可以选择使用“咸宁路 2 号”ATM 机获取服务；任何一台 ATM 机均可以对两个银行的客户提供服务。

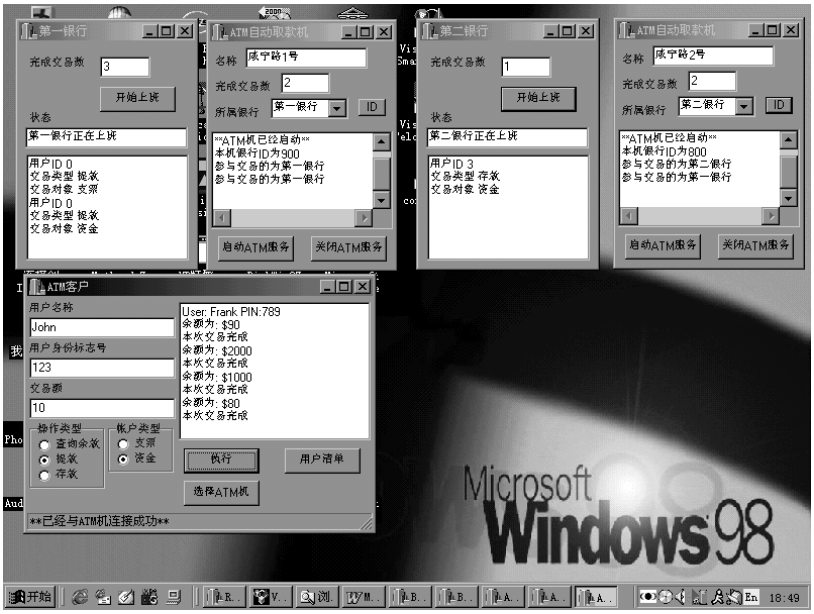

图 10.4　运行中的 ATM 电子商务演示系统

另外，我们将所有程序放在同一台机器上运行只是为了便于说明问题。

10.2　带有数据库的电子商务演示系统——CORBA Midas

电子商务软件通常需要使用数据库系统作为多层体系结构的支撑。下面这个“网上售书演示系统”就采用了数据库技术。服务器端必须拥有数据库支持，带有两个相关的“主从”表和一个订单表。“主”表记录了有关书籍的 ISBN 号、书名、作者、关键字、出版社名称、价格等内容；“从”表是一个销售明细表，包括书籍的 ISBN 号、现有库存量、各种批量购买价格等信息。客户端不需要数据库支持，用户可以从服务器上下载“相关表格”，在主表中选择一种书籍，获取其相应的销售明细表，还可以填写一个简易的订单，发给服务器保存。

为了讨论方便，我们假设数据库系统与服务器位于同一个站点。但是，在实际应用中，数据库也可以位于其它远程站点。

10.2.1　编写 IDL 接口

“网上售书演示系统”仅仅需要一个 CORBA 接口，内容如下所示。

例10.26　BookAgent.idl 文件内容及其说明。

```
interface BookAgent{//接口的名称
        exception BookAgentException{//一个自定义的异常
            string s;
        };
        //获取书籍表格内容
        any GetBooks(in boolean metadata);
        //订单初始化
        any InitOrders(in boolean metadata);
        //返回销售明细表格内容
        any GetDetails(in boolean metadata, in string bookISBN);
        //提交用户订单
```

```
        any ApplyOrders(in any Delta, out long ErrorCount);
};
```

BookAgent 接口中首先声明了一个自定义异常，通过字符串 s 表示错误信息；获取书籍表格内容的 GetBooks 以是否需要传递元数据为参数，返回数据为书籍数据表格的有关内容；订单初始化函数 InitOrders 以是否需要传递元数据为参数，返回一份空订单；获取销售明细表格内容的 GetDetails 以是否需要传递元数据及有关书籍 ISBN 号为参数，返回数据为整个明细表的有关内容；提交用户订单的函数 ApplyOrders 以有关订单为参数，返回可能发生的错误个数及错误内容。

在上述接口声明中，我们用 CORBA 中的 any 类型来表示数据库表格内容。不过，在 C++Builder 中，数据库表格内容本身采用与 Microsoft 兼容的 OleVariant 类型表示。因此，我们需要确保 CORBA any 类型与 OleVariant 类型之间能够互相转换，这些工作由 any2variant.*中的有关函数完成。

10.2.2 any 与 OleVariant 类型之间的转换

any 与 OleVariant 之间的转换牵扯到一些低层问题，对于这些内容大多数程序员可以不必操心。我们只需在需要上述转换的工程项目中加入下列 any2variant.*即可。

例10.27 any2variant.h 头文件及其说明。

```
#ifndef Any2VariantH
#define Any2VariantH
#include <windows.h>
#include <corba.h>
#include <any.h>

//将 VARIANT 转换为 any 指针
//如果是 VT_ARRAY/VT_UI1 就转换为 tk_sequence/tk_octet
//如果是 VT_VOID 就转换为 tk_void
//如果是其它类型就转换为 tk_null
CORBA::Any * SAFEARRAYToAny(const VARIANT &va);

//将 any 转换为 VARIANT 指针类型或引用类型
//如果是 tk_sequence/tk_octet 就转换为 VT_ARRAY/VT_UI1
//如果是 tk_void 就转换为 VT_VOID
//如果是其它类型就转换为 VT_EMPTY
VARIANT * AnyToSAFEARRAY(const CORBA::Any& any);
BOOL AnyToSAFEARRAYVar(const CORBA::Any& any, VARIANT &var);

#endif
```

例10.28 any2variant.cpp 文件及其说明。

```
#include <windows.h>
#include <corbapch.h>
#pragma hdrstop
#include <corba.h>
#include <any.h>

#include "Any2Variant.h"
//---------------------------------------------------------------------------
//将 Variant 类型的一维 BYTE 数组转换为 any 序列
```

```
BOOL SAFEARRAYToSequence(const VARIANT &va, CORBA::Any &any)
{
  if (va.vt & VT_ARRAY)//确保在处理数组类型 VARIANT
  {
    //获取 VARIANT 具体类型
    VARTYPE varType = VARTYPE(va.vt & ~(VT_ARRAY|VT_BYREF));
    if (varType == VT_UI1)//判断是否为 byte 类型
    {
    //获取数组的引用
   SAFEARRAY &sa = (va.vt & VT_BYREF) ? (**va.pparray) : (*va.parray);

      UINT dim = ::SafeArrayGetDim(&sa);//获取数组维数

      if (!(dim == 1))//只能处理一维数组
        return 0;

      DWORD cElements = sa.rgsabound[0].cElements;//获取元素数

      //为转换数组申请内存
      CORBA::ULong value_len = cElements + sizeof(CORBA::ULong);
      char *value = new char[value_len];
      void *ptr = value;

      //把元素总数 cElements 拷贝到第一个 ULong 单元中
      *(CORBA::ULong *)ptr = cElements;
      ((CORBA::ULong *)ptr)++;//移动指针，偏移量为 ULong 对应字节

      ::SafeArrayLock(&sa);//封锁有关 VARIANT
      memcpy(ptr, sa.pvData, cElements);//数据拷贝
      ::SafeArrayUnlock(&sa);//解除封锁

      //将内存块 value 中的数据转换为 tk_octet/ tk_sequence 类型的 any
      CORBA_TypeCode_var etc = new
                                  CORBA_TypeCode(CORBA::tk_octet, 0);
      CORBA_TypeCode_var tc = new CORBA::TypeCode(
            CORBA::tk_sequence,
            0,
            etc, 0);
      CORBA_MarshalInBuffer ibuf(value, value_len);
      any.read_value(ibuf, tc);

      delete[] value;//释放有关内存
      return TRUE;//转换成功
    }
  }

  return FALSE;//转换失败
}
```

```
CORBA::Any * SAFEARRAYToAny(const VARIANT &va)
{
   //创建一个 tk_null 实例
   CORBA::Any_var any = new CORBA::Any;
   if (va.vt == VT_VOID)//处理 VT_VOID 类型
   {
CORBA::TypeCode_var ptc = new
                                    CORBA::TypeCode(CORBA::tk_void, 0);
      any->replace(CORBA::_tc_TypeCode, ptc);
   }
   else//其它类型一律转换为 any 序列
      SAFEARRAYToSequence(va, *any);

   return CORBA::Any::_duplicate(any);
}

//将 any 序列转换为存储在 Variant 中的 SAFEARRAY
BOOL SequenceToSAFEARRAY(const CORBA::Any &any, VARIANT &var)
{
   void *ptr = (void *)any.value();//指向 any 取值

   //从第一个 ULong 单元中获取元素总数目
   DWORD count = *(CORBA::ULong *)ptr;
   ((CORBA::ULong *)ptr)++;//移动指针，偏移量为 ULong 对应字节

    SAFEARRAYBOUND bounds[1];//创建一个一维 SAFEARRAY 定界对象
    bounds[0].lLbound = 0;//起始下界为 0
    bounds[0].cElements= count;//上界为 count

   //创建规定起始界限的一维数组
   SAFEARRAY *psa = ::SafeArrayCreate(VT_UI1, 1, bounds);

   if (psa)
   {
      ::SafeArrayLock(psa);//封锁 SAFEARRAY
      memcpy(psa->pvData, ptr, count);//数据拷贝
      ::SafeArrayUnlock(psa);//解除封锁

      var.vt = VT_ARRAY|VT_UI1;//设置有关标志位
      V_ARRAY(&var) = psa;//将 psa 放入 Variant

      return TRUE;//转换成功
   }

   return FALSE;//转换失败
}

//将 any 内部数据转换为相关类型
```

```
BOOL InternalAnyToSAFEARRAY(const CORBA::Any& any,
                            CORBA::TypeCode_ptr tc, VARIANT &var)
{
   VARIANT *pv = &var;

   //数据拷贝
   switch (tc->kind())
   {
      case CORBA::tk_null:
            ::VariantClear(pv);
            pv->vt = VT_NULL;
            return TRUE;

      case CORBA::tk_void:
            ::VariantClear(pv);
            pv->vt = VT_VOID;
            return TRUE;

      case CORBA::tk_sequence:
            ::VariantClear(pv);
            return SequenceToSAFEARRAY(any, *pv);

      case CORBA::tk_alias:
      {
          CORBA_TypeCode_var ctc = tc->content_type();
          while (ctc->kind() == CORBA::tk_alias)
              ctc = ctc->content_type();
          return InternalAnyToSAFEARRAY(any, ctc, var);
      }

      case CORBA::tk_any:
      {
          CORBA_Any_ptr a = (CORBA_Any_ptr) any.value();
          CORBA_TypeCode_var ctc = a->type();
          return InternalAnyToSAFEARRAY(*a, ctc, var);
      }
   }
   return FALSE;
}

BOOL AnyToSAFEARRAYVar(const CORBA::Any& any, VARIANT &var)
{
   CORBA::TypeCode_var tc = any.type();
   return InternalAnyToSAFEARRAY(any, tc, var);
}

VARIANT * AnyToSAFEARRAY(const CORBA::Any& any)
{
   VARIANT *pv= new VARIANT;
   ::VariantInit(pv);
```

```
    AnyToSAFEARRAYVar(any, *pv);

    return pv;
}
```

10.2.3 建立有关数据表格

网上售书演示系统中需要三个表格。“主”表为书籍表，用来记录各种书籍信息；“从”表为销售信息表，用来记录各种销售信息；订单表与上述两个表独立，主要用来记录用户订购信息。为了讨论方便，我们将上述表按照本地 Paradox 表处理。实际应用中，上述表可以是 Oracle、Sybase、SQL、InterBase、DB2 等任何一种数据表，也不一定位于本地数据库中。

书籍信息表名称为 book.db，详细定义及内容如表 10.1 所示。

表 10.1 book.db 定义及内容

ISBN	书名	作者	出版社	定价	关键字	出版日期
7-116-14326-1	PHP 编程宝典	东方胜	IT 在线出版社	￥110.00	PHP	00-7-10
7-123-23465-4	CORBA/Java 编程	刘琪、董华	希望出版社	￥45.00	Jbuilder、CORBA	00-8-12
7-143-13245-9	C++Builder 5 突飞猛进	安澜、林泰	绝对出版社	￥110.00	C++Builder5	00-7-15
7-234-02345-1	Delphi 5 高级编程技术	罗岱、齐瑞	喜闻乐见出版社	￥99.00	Delphi5	00-6-30
7-302-02232-2	COM/DCOM 技术内幕	杨屏藩	计算机工业出版社	￥45.00	COM、DCOM	00-3-10
7-732-18436-8	DSP 即查即用	原萌	时代出版社	￥75.00	DSP、TI	00-8-10

销售信息表名称为 sales.db，详细定义及内容如表 10.2 所示。

表 10.2 sales.db 定义及内容

ISBN	现有库存	5-10 本批价	10-50 本批价	50-100 本批价	100 本以上批价
7-116-14326-1	10000	￥100.00	￥95.00	￥90.00	￥60.00
7-123-23465-4	10000	￥45.00	￥40.00	￥35.00	￥30.00
7-143-13245-9	30000	￥95.00	￥85.00	￥80.00	￥60.00
7-234-02345-1	50000	￥90.00	￥80.00	￥60.00	￥45.00
7-302-02232-2	10000	￥45.00	￥40.00	￥35.00	￥30.00
7-732-18436-8	20000	￥70.00	￥65.00	￥60.00	￥40.00

订单表的名称为 orders.db，详细定义如表 10.3 所示。

表 10.3 orders.db 定义

名称	地址	订购内容

现在，我们可以使用上述表格开始构筑网上售书演示系统了。

10.2.4　网上售书演示系统的 CORBA 实现

首先，我们采用 CORBA 服务器向导创建一个 CORBA 服务器程序。该程序对应的工程项目为 BookServer.bpr，主窗体所在的单元名称为 serverunit.*。

在第九章中，我们曾说明，可以采用 Tie 的代理方式编写 CORBA 对象实现。在本演示系统中，我们就采用了上述方式编写 CORBA 服务器程序。在激活 CORBA 对象实现向导后，选中 Delegation（Tie）、Data Module 两项，将文件名称设置为 ServerImpl，我们会得到一个数据模块，该数据模块也是实现 CORBA 接口定义函数的类。

现在，BookServer.cpp 的内容如下所示。

例10.29　BookServer.cpp 内容。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "ServerImpl.h"

#include <corba.h>
USERES("BookServer.res");
USEFORM("serverunit.cpp", ServerForm);
USEIDL("BookAgent.idl");
USEUNIT("any2variant.cpp");
USEUNIT("BookAgent_c.cpp");
USEUNIT("BookAgent_s.cpp");
USEFORM("ServerImpl.cpp", TBookAgentImpl);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        // Initialize the ORB and BOA
        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
        CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
        TTBookAgentImpl* bookAgent_BookAgentObject = new
                                        TTBookAgentImpl(NULL);
        _tie_BookAgent<TTBookAgentImpl>
        tie_bookAgent_BookAgentObject(*bookAgent_BookAgentObject,
                                        "BookAgentObject", true);
        boa->obj_is_ready(&tie_bookAgent_BookAgentObject);
        Application->CreateForm(__classid(TServerForm), &ServerForm);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

与标准的数据库程序类似，我们将有关表格、SQL 查询放在自动生成的数据模块

TBookAgentImpl 中。该数据模块共包括三个 TTable 对象，分别对应于上节中定义的三个数据表格，名称也各自匹配。为了建立主从关系，在数据模块中还特地引入了一个 TDataSource 控件，将其与 Book 表相连，并作为 Sales 的 MasterSource。

在主窗体上放置一些必要的控件，我们可以得到如图 10.5 所示的服务器界面。

由于售书服务器有可能同时支持多个用户，我们还引入了一个用于互斥用户的信号量 VISMutex。这样，整个对象实现的头文件如下所示。

例10.30　ServerImpl.h 头文件及其说明。

```
//---------------------------------------------------------------------------
#ifndef ServerImplH
#define ServerImplH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "BookAgent_s.hh"
#include <Db.hpp>
#include <DBTables.hpp>
//---------------------------------------------------------------------------
class TTBookAgentImpl : public TDataModule
{
__published: // IDE-managed Components
      TTable *Book;//数据表格
      TTable *Sales;
      TTable *Orders;
TDataSource *BookSource;//数据源控件，主要用于表示主从关系
private:
      VISMutex Mutex; // User declarations//用户互斥信号量
public:          // User declarations
      __fastcall TTBookAgentImpl(TComponent* Owner);
      CORBA::Any* __fastcall ApplyOrders(const CORBA::Any& Delta,
         CORBA::Long& ErrorCount);
      CORBA::Any* __fastcall GetBooks(CORBA::Boolean metadata);
      CORBA::Any* __fastcall GetDetails(CORBA::Boolean metadata,
         const char* bookISBN);
      CORBA::Any* __fastcall InitOrders(CORBA::Boolean metadata);
};
//---------------------------------------------------------------------------
extern PACKAGE TTBookAgentImpl *TBookAgentImpl;
//---------------------------------------------------------------------------
#endif
```

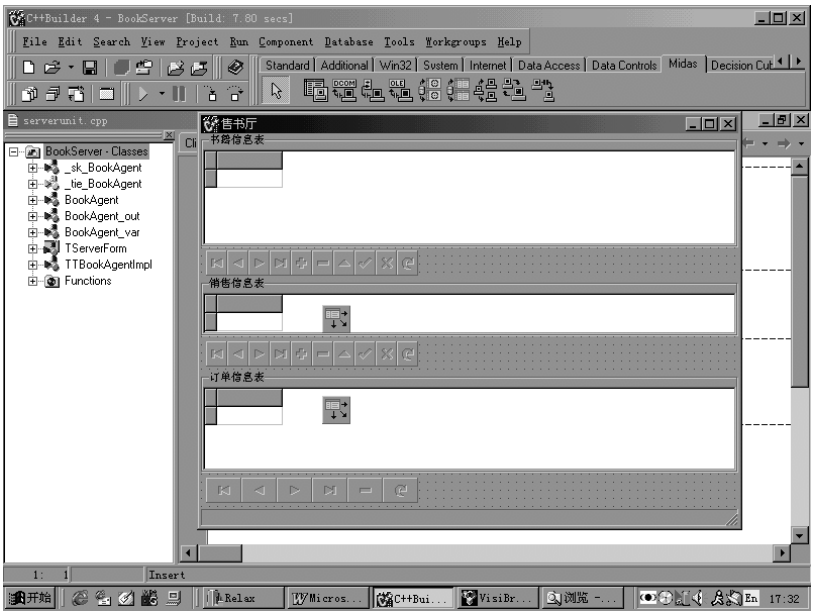

图 10.5　设计中的网络售书服务器主窗体

这些函数的实现代码如下所示。

例10.31　ServerImpl.cpp 文件及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include <corba.h>
#include "ServerImpl.h"
#include "serverunit.h"
#include "any2variant.h"//需要引入 any2variant.h
#include <BdeProv.hpp>//该文件与 TProvider 控件有关
#include <Provider.hpp>//该文件也与 TProvider 控件有关
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TTBookAgentImpl *TBookAgentImpl;
//---------------------------------------------------------------------------
__fastcall TTBookAgentImpl::TTBookAgentImpl(TComponent* Owner)
    : TDataModule(Owner)
{
    AnsiString path = ExtractFilePath(Application->ExeName);//提取路径
    //初始化有关数据表格
    Book->TableName =   path + "book.db";
    Book->Open();
    Sales->TableName = path + "sales.db";
    Sales->MasterFields =   "ISBN";//设置主从表关联字段
    Sales->Open();
    Orders->TableName = path + "orders.db";
    Orders->Open();
```

```
}
//---------------------------------------------------------------------------

CORBA::Any* __fastcall TTBookAgentImpl::ApplyOrders(
          const CORBA::Any& Delta, CORBA::Long& ErrorCount)
{
     VISMutex_var lock(Mutex);//封锁互斥量，超出作用域后自动释放
     ServerForm->StatusBar->SimpleText = "某用户发回一个订单";
     try
     {
     CoInitialize(NULL);//调用 dbclient.dll
     TProvider * Provider = new TProvider(this);//创建一个 TProvider 对象
     Provider->DataSet = Orders;//将 Orders 数据传递给 Provider
     OleVariant varDelta = AnyToSAFEARRAY(Delta);//数据转换
     int Count;//修改表格时错误个数记录变量
     OleVariant varErrors = Provider->ApplyUpdates(varDelta, -1, Count);
     CORBA::Any_var anyErrors = SAFEARRAYToAny(varErrors);
     ErrorCount = Count;//转换错误数及错误数据
     delete Provider;//释放 Provider
CoUninitialize();//释放 dbclient.dll
Orders->Refresh();//刷新订单记录
     return CORBA::Any::_duplicate(anyErrors);//返回错误数据
     }
     catch (::Exception & e)
     {
     throw BookAgent::BookAgentException(e.Message.c_str());//传递异常
     }
}

CORBA::Any*     __fastcall     TTBookAgentImpl::GetBooks(CORBA::Boolean
metadata)
{
     VISMutex_var lock(Mutex); //封锁互斥量，超出作用域后自动释放
     ServerForm->StatusBar->SimpleText = "某用户请求下载书籍信息";
     try
     {
     CoInitialize(NULL);
     TProvider * Provider = new TProvider(this);

     Book->Refresh();//刷新 Book
     Provider->DataSet = Book;
     Provider->Reset(metadata);//提取 Book 数据
     CORBA::Any_var any = SAFEARRAYToAny(Provider->Data);

     delete Provider;
     CoUninitialize();
     return CORBA::Any::_duplicate(any);//返回数据
     }
     catch (::Exception & e)
     {
```

```
    throw BookAgent::BookAgentException(e.Message.c_str());
    }
}

CORBA::Any*  __fastcall  TTBookAgentImpl::GetDetails(CORBA::Boolean
metadata,
  const char* bookISBN)
{
    VISMutex_var lock(Mutex); //封锁互斥量，超出作用域后自动释放
    ServerForm->StatusBar->SimpleText = "某用户请求下载指定销售信息";
    try
    {
    CoInitialize(NULL);
    TProvider * Provider = new TProvider(this);

    Book->SetKey();//为使用 FindKey 做准备
    Book->FindKey(ARRAYOFCONST((bookISBN)));//定位 bookISBN

    Provider->DataSet = Sales;//将与主表相关联的 Sales 赋予 Provider
    Provider->Reset(metadata);//提取从表 Sales 中的数据
    CORBA::Any_var any = SAFEARRAYToAny(Provider->Data);

    delete Provider;
    CoUninitialize();
    return CORBA::Any::_duplicate(any);
    }
    catch (::Exception & e)
    {
    throw BookAgent::BookAgentException(e.Message.c_str());
    }
}

CORBA::Any*  __fastcall  TTBookAgentImpl::InitOrders(CORBA::Boolean
metadata)
{
    VISMutex_var lock(Mutex); //封锁互斥量，超出作用域后自动释放
    try
    {
    CoInitialize(NULL);
    TProvider * Provider = new TProvider(this);

    Orders->Refresh();//关闭现有查询
```

```
    Provider->DataSet = Orders;//将 Orders 取值赋予 Provider
OleVariant MetaData =
                Provider->GetMetaData();//提取 Orders 中定义的字段
    CORBA::Any_var any = SAFEARRAYToAny(MetaData);

    delete Provider;
    CoUninitialize();
    return CORBA::Any::_duplicate(any);
    }
    catch (::Exception & e)
    {
    throw BookAgent::BookAgentException(e.Message.c_str());
    }
}
```

将上述工程编译后，我们就得到了网上售书演示系统的 CORBA 服务器程序。

10.2.5 网上售书演示系统的 CORBA 客户程序

网上售书演示系统的 CORBA 客户程序实际上采用了“公文包”模型。用户在需要的时候，可以向“售书厅”申请服务，获取最新的书籍信息。书籍信息下载完毕后，用户可以脱机浏览数据。如果希望进一步了解某种书籍的销售信息，可以要求“售书厅”仅仅发送一条有关记录；用户的订单也在脱机情况下填写，最后根据用户要求提交给“售书厅”，并由“售书厅”保存到自身数据库中。

由于 CORBA 客户程序所在站点不一定都带有数据库支持，因此，本演示系统中也不要求客户程序直接使用数据库功能。但是，通过使用 C++Builder 有关控件，客户端程序使用起来就如同支持数据库操作一样。

与所有 CORBA 客户程序一样，我们首先可以使用 CORBA 客户向导为我们创建一个 CORBA 客户工程项目，其名称为 BookClient.bpr，主窗体为 ClientForm，存放在 clientunit.* 单元中。窗体界面如图 10.6 所示。

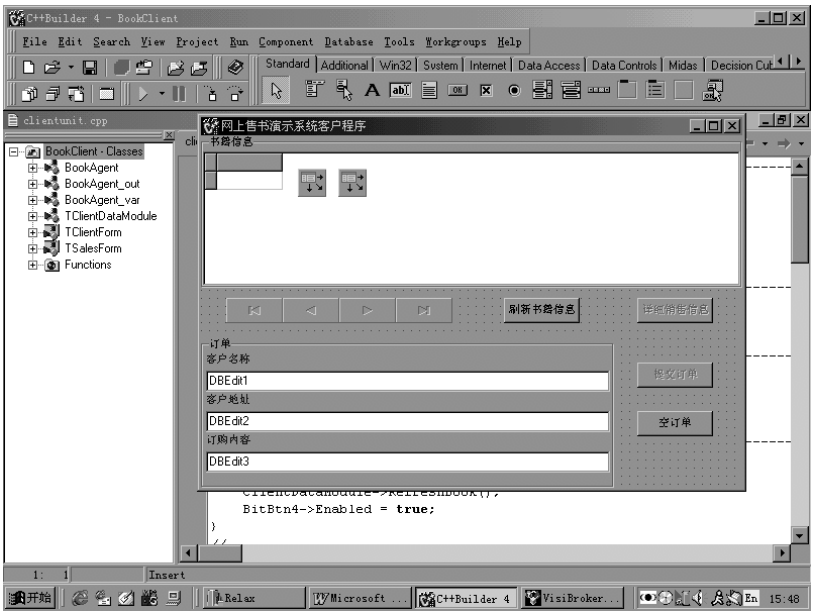

图 10.6　设计中的 CORBA 客户程序主界面

在服务器程序中，书籍信息表与销售信息表构成主从关系，销售信息表内容自动随书籍信息表当前记录改变而改变。在客户程序中，这样会迫使用户在浏览书籍信息时也必须与服务器保持连接，不利于服务器共享。因此，我们特地设置了一个“详细销售信息”按钮，只有在点击这个按钮后，客户程序才会向服务器发出请求，获取当前书籍记录的详细销售信息，并显示在如图 10.7 所示的 SalesForm 中。

例10.32　BookAgent.cpp 文件及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include <corba.h>
USERES("BookClient.res");
USEFORM("clientunit.cpp", ClientForm);
USEIDL("BookAgent.idl");
USEUNIT("any2variant.cpp");
USEUNIT("BookAgent_c.cpp");
USEFORM("ClientData.cpp", ClientDataModule);
USEFORM("salesunit.cpp", SalesForm);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        // Initialize the ORB and BOA
        CORBA::ORB_var orb = CORBA::ORB_init(__argc, __argv);
        CORBA::BOA_var boa = orb->BOA_init(__argc, __argv);
        Application->CreateForm(__classid(TClientForm), &ClientForm);
        Application->CreateForm(__classid(TClientDataModule),
                                                    &ClientDataModule);
        Application->Run();
```

```
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

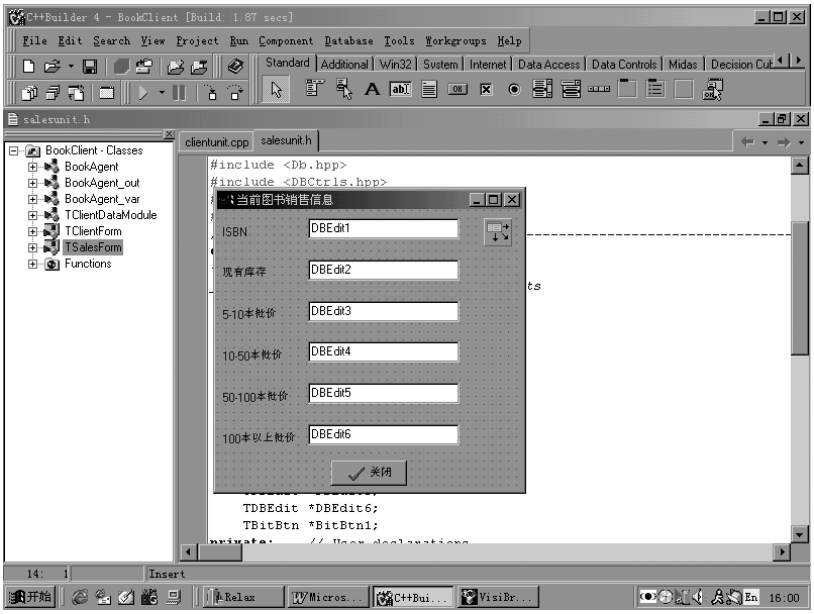

图 10.7　设计中的 SalesForm

例10.33　clientunit.h 头文件及其说明。

```
#include <Db.hpp>
//---------------------------------------------------------------------------
class TClientForm : public TForm
{
__published: // IDE-managed Components
    TGroupBox *GroupBox1;
    TDBGrid *DBGrid1;//显示书籍信息
    TDBNavigator *DBNavigator1;//书籍信息浏览导航器
    TGroupBox *GroupBox2;
    TDBEdit *DBEdit1;//记录显示客户名称
    TDBEdit *DBEdit2; //记录显示客户地址
    TDBEdit *DBEdit3; //记录显示订购内容
    TLabel *Label1;
    TLabel *Label2;
    TLabel *Label3;
    TBitBtn *BitBtn1;//刷新书籍信息按钮
    TBitBtn *BitBtn2;//提交订单按钮
    TBitBtn *BitBtn3;//空订单按钮
    TDataSource *Book;//书籍信息源
    TDataSource *Orders;//订单信息源
```

```
    TBitBtn *BitBtn4;//获取当前书籍记录的销售信息按钮
    void __fastcall BitBtn1Click(TObject *Sender);
    void __fastcall BitBtn2Click(TObject *Sender);
    void __fastcall BitBtn3Click(TObject *Sender);
    void __fastcall BitBtn4Click(TObject *Sender);
private:  // User declarations
public:         // User declarations
    __fastcall TClientForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TClientForm *ClientForm;
//---------------------------------------------------------------------------
#endif
```

例10.34　clientunit.cpp 文件及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "clientunit.h"
#include "ClientData.h"
#include "salesunit.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TClientForm *ClientForm;
//---------------------------------------------------------------------------
__fastcall TClientForm::TClientForm(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TClientForm::BitBtn1Click(TObject *Sender)
{
    ClientDataModule->RefreshBook();//刷新书籍记录
    BitBtn4->Enabled = true;//允许获取当前书籍记录的销售信息
}
//---------------------------------------------------------------------------
void __fastcall TClientForm::BitBtn2Click(TObject *Sender)
{
    ClientDataModule->PostOrders();//提交订单
    BitBtn2->Enabled = false;//强迫用户必须清空订单
}
//---------------------------------------------------------------------------
void __fastcall TClientForm::BitBtn3Click(TObject *Sender)
{
    ClientDataModule->RefreshOrders();//获取空订单
    DBEdit1->DataField = "名称";//将数据编辑控件与有关字段关联
    DBEdit2->DataField = "地址";
    DBEdit3->DataField = "订购内容";
    BitBtn2->Enabled = true;//允许提交订单
```

```
}
//---------------------------------------------------------------------------
void __fastcall TClientForm::BitBtn4Click(TObject *Sender)
{
    ClientDataModule->GetSales();//获取销售信息
    SalesForm = new TSalesForm(this);//用指定的窗体显示销售信息
    SalesForm->ShowModal();
    delete SalesForm;
}
//---------------------------------------------------------------------------
```

例10.35　ClientData.h 头文件及其说明。

```
//---------------------------------------------------------------------------
#ifndef ClientDataH
#define ClientDataH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Db.hpp>
#include <DBClient.hpp>
#include "BookAgent_c.hh"
//---------------------------------------------------------------------------
class TClientDataModule : public TDataModule
{
__published: // IDE-managed Components
    TClientDataSet *Book;//用户数据集 Book
    TClientDataSet *Sales;//用户数据集 Sales
    TClientDataSet *Orders;//用户数据集 Orders
private:  // User declarations
    BookAgent_ptr __fastcall GetBookServer();
    BookAgent_var FBookServer;//对 BookAgent 对象的引用
    void __fastcall SetBookServer(BookAgent_ptr _ptr);
public:         // User declarations
    __fastcall TClientDataModule(TComponent* Owner);
    void __fastcall RefreshBook(void);//刷新书籍信息
    void __fastcall RefreshOrders(void);//获取空订单
    void __fastcall PostOrders(void);//提交订单
    void __fastcall GetSales(void);//获取销售信息
    __property BookAgent_ptr BookServer = {read=GetBookServer,
write=SetBookServer};
};
//---------------------------------------------------------------------------
extern PACKAGE TClientDataModule *ClientDataModule;
//---------------------------------------------------------------------------
#endif
```

例10.36　ClientData.cpp 文件及其说明。

```
//---------------------------------------------------------------------------
```

```
#include <vcl.h>
#pragma hdrstop

#include <corba.h>
#include "ClientData.h"
#include "any2variant.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TClientDataModule *ClientDataModule;
//---------------------------------------------------------------------------
__fastcall TClientDataModule::TClientDataModule(TComponent* Owner)
    : TDataModule(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TClientDataModule::RefreshBook(void)
{
    //TODO: Add your source code here
    try
    {
    CORBA::Any_var anyData = BookServer->GetBooks(!Book->Active);
    OleVariant varData = AnyToSAFEARRAY(anyData);
    Book->Data = varData;//激发服务并转换数据
    }
    catch (const BookAgent::BookAgentException &e)
    {
    throw Sysutils::Exception(AnsiString(e.s));//转发自定义异常
    }
    catch (const CORBA_Exception & e)
    {
    throw Sysutils::Exception(e._name());//转发系统异常
    }
}

void __fastcall TClientDataModule::RefreshOrders(void)
{
    //TODO: Add your source code here
    try
    {
    CORBA::Any_var anyData = BookServer->InitOrders(!Orders->Active);
    OleVariant varData = AnyToSAFEARRAY(anyData);
    Orders->Data = varData;//激发服务并转换数据
    }
    catch (const BookAgent::BookAgentException &e)
    {
    throw Sysutils::Exception(AnsiString(e.s));//转发自定义异常
    }
    catch (const CORBA_Exception & e)
    {
    throw Sysutils::Exception(e._name());//转发系统异常
    }
```

```
}

void __fastcall TClientDataModule::PostOrders(void)
{
    //TODO: Add your source code here
    try
    {
    long ErrorCount;//初始化参数

    CORBA::Any_var anyDelta = SAFEARRAYToAny(Orders->Delta);
    CORBA::Any_var anyErrors = BookServer->ApplyOrders(anyDelta,
ErrorCount);
    OleVariant varErrors = AnyToSAFEARRAY(anyErrors);
    Orders->Reconcile(varErrors);//转换数据并激发服务
    }
    catch (const BookAgent::BookAgentException &e)
    {
    throw Sysutils::Exception(AnsiString(e.s));//转发自定义异常
    }
    catch (const CORBA_Exception & e)
    {
    throw Sysutils::Exception(e._name());//转发系统异常
    }
}

void __fastcall TClientDataModule::GetSales(void)
{
    //TODO: Add your source code here
    try
    {
    CORBA::Any_var anyData = BookServer->GetDetails(
        !Sales->Active,Book->FieldByName("ISBN")->AsString.c_str()
        );//传递参数激发服务
    OleVariant varData = AnyToSAFEARRAY(anyData);
    Sales->Data = varData; //转换数据
    }
    catch (const BookAgent::BookAgentException &e)
    {
    throw Sysutils::Exception(AnsiString(e.s));//转发自定义异常
    }
    catch (const CORBA_Exception & e)
    {
    throw Sysutils::Exception(e._name());//转发系统异常
    }
}

BookAgent_ptr __fastcall TClientDataModule::GetBookServer(void)
{
    if (FBookServer == NULL)
    {
        FBookServer = BookAgent::_bind();
```

```
    }
    return FBookServer;
}

void __fastcall TClientDataModule::SetBookServer(BookAgent_ptr _ptr)
{
    FBookServer = _ptr;
}
```

例10.37　salesunit.cpp 文件及其说明。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "salesunit.h"
#include "ClientData.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TSalesForm *SalesForm;
//---------------------------------------------------------------------------
__fastcall TSalesForm::TSalesForm(TComponent* Owner)
    : TForm(Owner)
{
    DBEdit1->DataField = "ISBN";//把信息显示控件与有关字段联系起来
    DBEdit2->DataField = "现有库存";
    DBEdit3->DataField = "5-10 本批价";
    DBEdit4->DataField = "10-50 本批价";
    DBEdit5->DataField = "50-100 本批价";
    DBEdit6->DataField = "100 本以上批价";
}
```

至于 salesunit.h 头文件，都是一些变量声明，没有详细列举的必要。
编译上述所有文件，我们就可以得到网上售书演示系统的 CORBA 客户程序。

10.2.6　执行网上售书演示系统

确保在局域网内至少运行一个 Smart Agent 后，我们就可以首先启动网上售书系统的

图 10.8　运行中的售书服务器程序

服务器程序——“售书厅”。这个服务器程序本身允许系统人员添加、删除、修改、浏览各种书籍信息及有关的销售信息，也允许系统人员浏览、删除用户订单，如图 10.8 所示。实际上，这个服务器程序本身就是一个数据库管理系统。

现在，可以在局域网内任何一台机器上启动售书演示系统客户程序，也可以同时启动多个售书演示系统客户程序。当然，如果对 Smart Agent 做适当的配置，CORBA 客户程序可以位于因特网的任何一个站点上。

运行中的售书客户程序如图 10.9 所示。

图 10.9　浏览书籍信息

如果客户希望进一步了解某种书籍的销售信息，仅仅需要将游标停留在有关记录上，然后点击“详细销售信息”按钮即可。具体情况如图 10.10 所示。

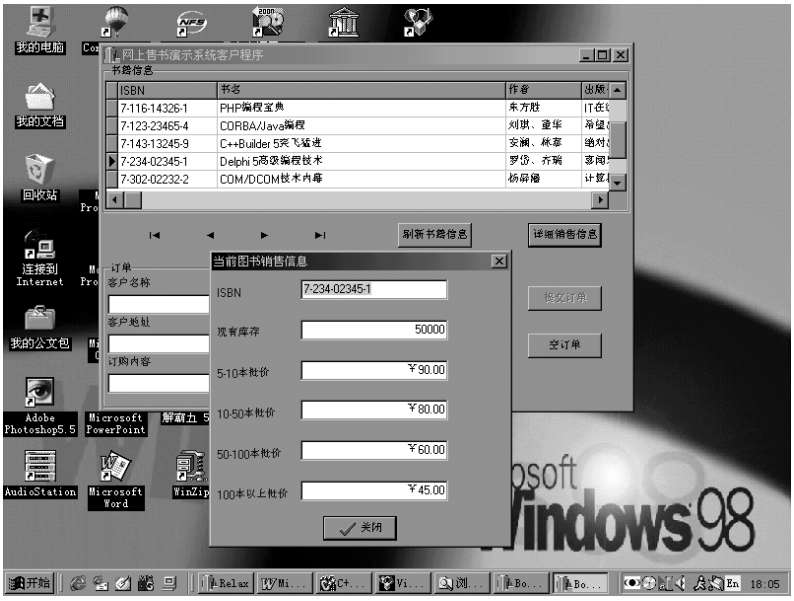

图 10.10　进一步查询某种图书的详细销售信息

在点击“空订单”按钮后，我们可以填写自己的订单，如图 10.11 所示。

图 10.11　填写订单

填写完订单后，点击“提交订单”按钮就可以把订单发送给售书服务器。如果不发生异常，售书服务器将把订单保存到数据库中，以便有关人员查阅。如果发现“提交订单”不能使用，可以首先点击“空订单”按钮。

如果滚动售书客户程序书籍信息的显示表格，我们会发现，在书籍信息的最后，新添加了一个名为“Sales”的字段，内容全部为“DATASET”。显示字段中还多了一个浮动按钮，如果点击这个浮动按钮，相关书籍记录的详细销售信息会自动显示出来，如图 10.12 所示。

图 10.12　新添加的“Sales”字段、浮动按钮及点击结果

以上变化是因为 TProvider 对象在传递数据信息过程中，发现有关表格带有一个“从”表，就将有关信息一并嵌套传递过来，以便用户查询。这一点显示了 C++Builder 数据库功能的强大威力。

由于我们已经将详细销售信息用自己的方式显示出来，这个自动功能显得多余，为此，将 TClientForm::BitBtn1Click 函数修改为下列情况即可取消上述现象。

例10.38　修改后的 TClientForm::BitBtn1Click 函数。

```
void __fastcall TClientForm::BitBtn1Click(TObject *Sender)
{
    ClientDataModule->RefreshBook();
    DBGrid1->Columns->Items[DBGrid1->FieldCount-1]->Visible = false;
    BitBtn4->Enabled = true;
}
```

现在重新执行网上售书客户程序，自动添加的 Sales 字段不再出现。不过，这只是一个表面现象，因为销售信息表的内容已经被嵌套传递到客户程序端了。显然，当“主”表、“从”表信息增多时，会导致数据传递开销过大的问题。为了彻底解决上述问题，应该将 CORBA 对象实现中的 GetBooks 函数修改如下。

例10.39　对 GetBooks 函数的修改。

```
CORBA::Any*  __fastcall  TTBookAgentImpl::GetBooks(CORBA::Boolean
metadata)
{
    VISMutex_var lock(Mutex);
    ServerForm->StatusBar->SimpleText = "某用户请求下载书籍信息";
    try
    {
    CoInitialize(NULL);
    TProvider * Provider = new TProvider(this);
    Provider->Options<<poFetchDetailsOnDemand;//新添加代码
```

```
    Book->Refresh();
    Provider->DataSet = Book;
    Provider->Reset(metadata);
    CORBA::Any_var any = SAFEARRAYToAny(Provider->Data);

    delete Provider;
    CoUninitialize();
    return CORBA::Any::_duplicate(any);
    }
    catch (::Exception & e)
    {
    throw BookAgent::BookAgentException(e.Message.c_str());
    }
}
```

经过上述修改，自动嵌套传递“从”表数据的问题得以彻底解决。

新添加的那行代码要求 TProvider 控件根据用户要求传递“从”表数据。在所附光盘中，以上修改代码用注释方法书写。用户可以自行调整，领悟其中奥妙。

10.3 本章小结

本章详细解析了两个电子商务演示系统。

在 ATM 电子商务演示系统中，我们解析了如何与指定名称的 CORBA 实例绑定，获取有关服务；说明了如何移动 CORBA 向导自动生成的代码，使它符合我们的编程需要。对于同一个 IDL 接口，用户还可以用同名、异名的类完成多个对象实现，在应用中通过指定实例具体名称选择需要的服务。

在网上售书电子商务演示系统中，我们提供了一个标准的多层体系结构、含有数据库功能的 CORBA 程序。服务器程序带有用户互斥机制，便于不同用户共享，支持数据库操作，数据表格之间带有“主从”关系。所有这些特点都与实际应用中的数据库要求一致。客户程序采用了公文包模型，能够有效减轻服务器负载，同时，客户程序端不需要数据库支持，却能够达到使用数据库的同样效果。

我们相信，任何复杂的电子商务系统总是由一些小的、简单的电子商务子系统有机构筑起来的。

第 11 章　CORBA 聊天室

本章内容提要：

- CORBA 与 COM/DCOM
- CORBA 与 Java
- CORBA 与 Web
- CORBA *的数据库能力*
- CORBA3.0 *的新动向*

11.1　CORBA 与 COM/DCOM

CORBA 并不是开发分布式软件的唯一体系结构，Microsoft 独家支持的 COM/DCOM 就是其最有力的竞争对手之一。但是，连 COM/DCOM 的开发人员也承认，这种技术本身并不能单独实现真正的分布式编程。如果希望通过 COM/DCOM 构筑分布式软件，用户就必须同时采用 Microsoft Transaction Server、Falcon（Microsoft 对消息服务的昵称）以及 Wolfpack（Microsoft 对 Clustering 的昵称）等技术（这些技术本身也在不断更新着）。在 Microsoft 眼中，分布式软件体系结构是一个如图 11.1 所示的“洋葱”。所以，软件界很多权威人士纷纷指责 Microsoft 的分布式软件体系结构是一种参了水分的 COM/DCOM。

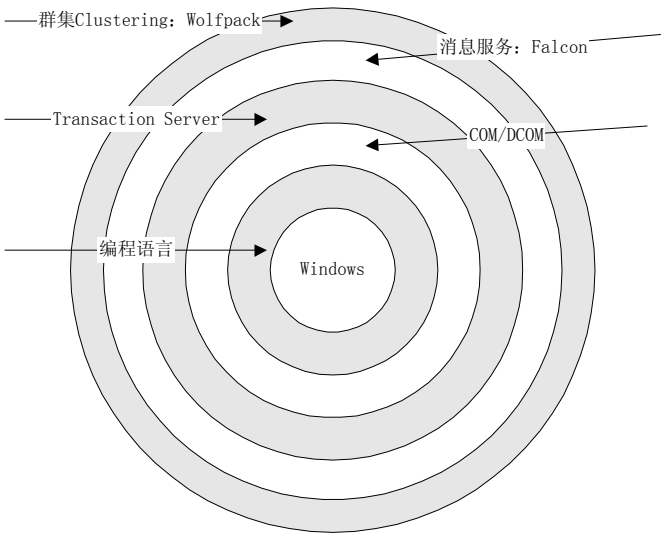

图 11.1　Microsoft 的分布式体系结构

指责归指责，COM/DCOM 在分布式软件中的成就不可否认：COM/DCOM 的产品比 CORBA 的产品问世的早，目前比 CORBA 占有更大的市场份额。但是，随着用户对于分布式软件要求的深入，COM/DCOM 也暴露出一系列问题，Microsoft 公司为此正致力于推出 COM+，以弥补“由于 COM/DCOM 不成熟所带来的不足”。

在 CORBA 与 COM/DCOM 激烈竞争的同时，OMG 及软件行业也为这两种技术的融合、兼容做了必要的工作。OMG 从 1996 年颁布的 CORBA2.0 指标中就开始制订了

CORBA/COM、CORBA/Automation、COM/CORBA、Automation/CORBA 等一系列映射关系，希望 CORBA 用户可以同时透明使用 CORBA 及 COM/DCOM 对象。软件行业中，也有不少厂商着手开发了同时适用于 CORBA 及 COM/DCOM 的中间件，如 HP 公司的 ORBplus，INOA 公司的 OrbixCOMet，Expersoft 的 CORBA+ ActiveX Bridge，DEC 公司的 DTC（ObjectBroker Desktop Connection）等等。所以，大多数软件界人士认为，CORBA 与 COM/DCOM 竞争还只是刚刚开始，在未来的一段时间内，基于 CORBA 的分布式对象与基于 COM/DCOM 的分布式对象将共同存在。

☎ **技巧** 如果使用 VisiBroker，能否同时使用 CORBA 及 COM/DCOM 对象？回答是肯定的。虽然 VisiBroker 目前还没有直接实现 CORBA 与 COM/DCOM 的透明调度，但是，紧密集成了 VisiBroker 的 C++Builder、Delphi、JBuilder 都能够同时开发、使用 CORBA 对象及 COM/DCOM 对象。也就是说，Inprise 的这些开发工具从编程角度为用户提供了 CORBA 与 COM/DCOM 的互操作性，我们可以在同一个程序模块中随意使用这两种分布式对象。

在 CORBA 与 COM/DCOM 的竞争史中，有一些值得深思的现象。CORBA 与 COM/DCOM 的概念都是在 90 年代初期提出的，但是，它们的发展经历、发展思路却显著不同。

OMG 是一个非赢利性的组织，致力于制订标准，所以希望颁布非常合理、结构严谨的分布式体系结构规范。由于规范的制订牵扯到软件行业的各个厂家，因此这个过程漫长而且艰辛。而且，即使规范制订出来后，也不能马上获得满足所有规范的产品。因此，CORBA 在发展过程中曾经一度受到冷落，一些 CORBA 研究者被 Microsoft 重金挖走，转而研究 COM/DCOM。经过十年发展，CORBA3.0 规范已经制订成功，大约比 COM/DCOM 领先两年，不少厂商也开发出了相应的 CORBA 产品，而且，人们对分布式软件的要求越来越高，这些因素带来了 CORBA 的再次复兴。目前，CORBA 面临的问题就是进一步扩大市场，使整个软件界获益于这种精心设计的“软件总线结构”。

Microsoft 公司对于分布式对象系统的理解则是“首先占据市场份额，再详细考虑设计合理性问题”，他们的分布式对象一直在变化着：最初是 DDE（动态数据交换），接着否认、放弃了 DDE，推出 OLE（对象链接与嵌入），后来把 OLE 的概念加以推广，提出了 OCX 控件、ActiveX 技术。现在，又声称有可能重新定义 Microsoft IDL 接口编写格式，推出新一代 COM/DCOM（或者 COM+）。

在这样一种策略引导下，用户很容易产生一种错觉：Microsoft 的技术更新十分快。但是，如果从另一个客观的角度来看：他们自我否定的是否太频繁了？因此，有人曾经指出，如果用户停下来认真思考一下，就不会盲目追随 Microsoft。然而，COM/DCOM 毕竟已经占据了庞大的市场，实际上成为一种不容忽视的标准。目前，COM/DCOM 需要解决的问题是进一步考虑设计合理性、设计全面性问题。在强大资金的保证下，谁又能够断言 Microsoft 的 COM+不会成功？这种先给客户甜头，在用客户的投资继续研究开发产品的思路目前已经被多数软件公司采纳。

在 CORBA 与 COM/DCOM 的竞争中，对于两者，我们都应该给予足够的关注。无论如何，它们要解决的基本问题是一致的，只是采用了不同的方式而已。而这个基本问题就是“如何开发高质量的分布式软件”，这也正是软件界不断革新的技术动力所在。

我们认为，CORBA 与 COM/DCOM 都是软件界集体智慧的结晶。

11.2 CORBA 与 Java

CORBA 与 Java 是相互补充的。CORBA 在最初设计时就希望跨越语言带来的障碍，因此，Java 以及日后可能出现的“超 Java”都能够自动与 CORBA 体系结构紧密衔接。目前看来，Java 的出现为 CORBA 带来了以下一些新的生机：

- Java 使 CORBA 对象代码可以在因特网上随意移动，客户/服务器程序模块都能够在互联网上动态传输。当然，如果缺乏一定的体系结构，即便是采用同一种语言在同一台机器上编写的程序代码之间的集成也是颇费时间的，而 CORBA 恰好提供了这种体系结构。
- Java 扩展并简化了 CORBA 的对象生存期服务以及对象外化服务。Java 的“下载、运行”方式连对象的运行代码都可以轻松获得，那么控制对象的生存状态肯定会简单的多。另一方面，通过 Java 语言，ORB 可以在传送 CORBA 对象行为的基础上进一步传送 CORBA 对象状态。
- Java 降低了开发 ORB 的难度。在没有 Java 以前，CORBA 必须为每种语言、每种操作系统都开发出各自的 ORB 设施。借助 Java 语言本身“一次编写，到处运行”的特点，CORBA 只需为不同语言开发 ORB 系统即可。这不，Inprise 已经推出适用于 Linux 的 VisiBroker for Java 版以及 VisiBroker for C++版了。
- Java 降低了 CORBA 系统的管理、维护成本。如果开发商希望更新 CORBA 对象的 Java 代码，可以在一个集中的服务器上发布；然后用户可以在自己方便的时候、方便的地点获得这些更新。如果与 CORBA 的事件服务、异步通信等方式相结合，这种更新甚至可以自动完成。
- Java 补充了 CORBA 的代理结构。现在，用户可以开发漫游型 CORBA 对象（也称为移动代理）。这些对象携带了自己的状态、行为、漫游路线、目的等信息，能够通过不同站点的 ORB 与其它局部、本地 CORBA 对象交互，并适时改变自己的行为。当然，单独的 Java 小程序（Applet）本身不具有状态，无法与所到站点的对象交流，也不能改变自己的行为。
- Java 为 CORBA 提供了移动式存储容器。如果说 HTML 及 Web 是信息的可视化、移动式存储容器，Java 则是对象及组件的可视化、移动式存储容器。
- Java 语言非常适合开发网络对象，或者说分布式对象。Java 语言提供的内部多线程、垃圾收集及差错管理为开发各种分布式商务对象提供了便利。
- Java 与 CORBA 是软件界抗衡 Microsoft 旗下 Windows、COM/DCOM 的坚强联盟。如果 Windows 真的垄断了操作系统市场，Java 就没有存在的必要了，而 CORBA 也可以完全让位于 COM/DCOM。因此，Java 与 CORBA 可以说是唇亡齿寒的关系。

也正是由于以上这些原因，OMG 在 CORBA2.2 规范中就颁布了 OMG IDL 与 Java 的映射。而且，大多数厂商都已经将自己的 CORBA 产品升级到 Java 平台之上，其中，VisiBroker 就是因为其为 Java 开发的 ORB 而举世闻名的。

眼下，Sun 公司正在研究、完善一种叫做 Java RMI（Remote Method Invocation）的机制，这种 Java 远程方法调用机制可以广义的看作是仅仅适用于 Java 语言的专用 CORBA。Sun 公司已经宣布，RMI 的通信协议也是 IIOP，而且，RMI 也将把 CORBA 的某些基本服务作为自己的对象服务标准。RMI 希望给用纯 Java 语言开发分布式软件提供必要的机制，是 Java 自身发展的需要。不过，当考虑到遗留程序、多种语言编程时，Java RMI 就显得无能为力了。

11.3 CORBA 与 Web

仔细分析 Web 面临的问题，我们不难发现，为 Web 添加交互能力、结构化 Web 内容都只不过是 Web 面向对象化进程中必不可少的步骤。而真正的 Object Web 必须将分布式

对象纳入其中。这就是如图 11.2 所示的多层体系结构。

这幅插图在第一章中也曾出现过，可以看作近几年以及今后几年内 CORBA 与 Web 的相处关系图。

在客户端，结构化的 Web 既可以直接使用 Java Applets 来获取一定的动态行为特性，也可以通过 CGI、ISAPI、NSAPI 将自己的行为与服务器上各种语言编写的分布式对象的方法“挂钩”（也就是在 Web 页面中声明<FROM ACTION = “……” METHOD = POST|GET>）。

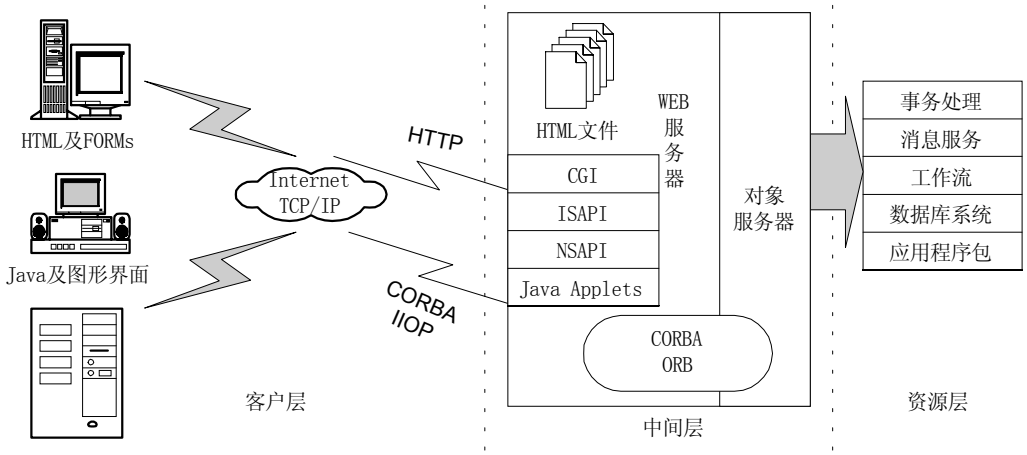

图 11.2　Object Web 模型

在中间层的 Web 服务器、CORBA 对象服务器上，Java Applets 以及各种语言编写的 CORBA 分布式对象接受来自客户的调度，并转而激发资源层更多的 CORBA 对象以完成相应的服务请求，并将结果以 Web 页面方式返回给客户。

在资源层，各种 CORBA 对象将提供包括数据库、文本编辑、事务处理、电子邮件等在内的几乎所有各种应用服务。

11.4　CORBA 的数据库能力

目前，电子商务软件的前台一般是 Web 页面，后台则一般是各种数据库支持。Microsoft 的 Object Web 是以 COM/DCOM 为核心设计的，由于 Microsoft 先后推出了 ODBC 以及 ADO，后台数据库的使用与开发变得更为简便了。那么，CORBA 是否具有配套服务呢？回答是肯定的。

在 Microsoft 推出 ODBC 用来建立、维护与各种数据库系统的连接后，Inprise 也在 Delphi、C++Builder 中推出了 BDE（Borland Database Engine）；而且，Inprise 类似 ADO 模型的产品比 Microsoft 的同类产品问世的更早，就是非常容易使用的 Data Access 以及 Data control 系列组件。这些组件的威力使得 Delphi、C++Builder、JBuilder 能够与 PowerBuilder 相媲美。

在上一章中，我们曾经详细解析了一个网上售书电子商务演示系统，就是 CORBA 数据库能力的一个实际说明。

11.5　CORBA3.0 的新动向

对一项技术进行前瞻恐怕是一件吃力不讨好的事，因此，我们仅仅提供 CORBA3.0 的一些基本信息。该规范已经在 1999 年 8 月 27 日获得 OMG 表决，目前有关产家正在开

发实验产品，以检验这些指标的可行性、细化一些重要的技术细节，估计至少要到 2000 年下半年才会有眉目。

新增加的规范可以大致分为以下内容：

- Internet 集成
- 服务质量控制
- CORBA 组件结构

11.5.1 CORBA3.0 中的 Internet 集成

新添加的第一个 Internet 集成规范围绕防火墙展开。

CORBA3.0 定义了传输层的防火墙、应用层的防火墙以及用于回叫（callbacks）或者事件通知的双向 GIOP 连接协议。

传输层的防火墙在 TCP 层上工作。通过为 IIOP 设置 683 端口号，为基于 SSL 安全协议的 IIOP 设置 684 端口号，将允许管理员配置通过 IIOP 协议传输的 CORBA 通信。规范同时也考虑了使用 Socks 的 CORBA 对象的兼容性问题。

在 CORBA 应用中，对象实例经常需要回叫或者通知激发它们的有关客户，此时，这些对象实例承担客户角色，并且需要反向激发。由于标准的 CORBA 激发只能单向进行，典型的回叫过程一般需要建立第二个 TCP 连接，指向回叫目的地。这种方式通常会被防火墙禁止。因此，在新的 CORBA 规范中，将允许在不威胁端安全的情况下，通过 IIOP 进行反向激发。

另外一个 Internet 集成指标与互操作命名服务（Interoperable Name Service）有关。

对象引用被人们称为 CORBA 体系结构的里程碑。因为没有对象引用，我们就没有办法使用一个对象实例，即使它正在我们知道的某个站点上运行着。在网络上，对象引用是通过互操作对象引用 IOR 来传递的。

互操作命名服务定义了 URL 格式的对象引用。用户既可以通过在程序中键入 iioploc 来获取本地或远程的，包括命名服务在内的某种服务，也可以通过在程序中键入 iiopname 激发用户在 URL 中规定的远程命名服务，并进一步获得需要的互操作对象引用 IOR。

比如，通过互操作命名服务，我们可以使用 iioploc://www.omg.org/NameService 来获得运行在 IP 地址与 www.omg.org 对应的机器上的命名服务。

11.5.2 CORBA3.0 中的服务质量控制

CORBA3.0 中的服务质量控制实际上包括以下内容：

- 异步消息处理（Asynchronous Messaging）。异步消息处理为静态激发及动态激发同时定义了一系列新的异步激发模式、时间相关激发模式。新的异步激发模式允许客户通过轮询方式（polling）或者回叫方式接收响应。时间相关激发模式允许用户通过服务质量控制策略设置激发操作的开始时间、结束时间或者有效时间。
- 服务质量控制策略。服务质量控制策略允许用户规定激发操作的优先权、最迟响应时间（Deadlines）、时间相关参数、路由选择参数，以控制服务质量。
- 极小化 CORBA（Minimum CORBA）。极小化 CORBA 主要用于嵌入式系统（embedded systems）。嵌入式系统的程序被烧制在芯片内，通常不会再改变。因此，即便是为了和其它网络程序集成，包括接口仓库、DII 等在内的许多 CORBA 机制都显得多余。
- 实时 CORBA（Real-time CORBA）。实时 CORBA 将根据用户激发各种操作的优先级别，预测包括线程、协议、连接等在内的各种资源冲突，并提供一种合理控制资源的激发方案。
- CORBA 容错性（Fault-tolerance for CORBA）。CORBA 容错性可以通过实体冗余（entity

redundancy)、错误管理控制来提高 CORBA 系统的健壮性。

11.5.3 CORBA3.0 中的组件结构

CORBA3.0 正在引入 CORBA 组件群（CORBAcomponents）及 CORBA 脚本语言（CORBAscripting）。

CORBA 组件群将拥有以下几个主要特点：

- CORBA 组件群本身就是一个环境容器（container environment）。环境容器将持续对象服务、事务服务、安全服务预先打包，提供一个比 CORBA 服务级别更高的抽象。CORBA 组件群允许用户在开发电子商务软件时自动获取相关服务特性。
- CORBA 组件群自动维护本身使用的各种事件类型，提供传递事件的信道。
- CORBA 组件群自动维护所有被包括在内的组件的接口信息。
- CORBA 组件群支持多重接口，提供在多个接口中导航（navigation）的机制。
- CORBA 组件群能够与 EJBs（Enterprise JavaBeans）集成，允许将 EJBs 组件安装在内。而 EJBs 就是支持 RMI 的 Java。

在 CORBA3.0 中，还将定义分布式软件的格式，包括一个软件安装器（installer）和一个基于 XML 的配置工具。另外，CORBA3.0 还将引入 CORBA 脚本语言，并同 VBScript 以及 JavaScript 等脚本语言建立映射关系。

11.6 本章小结

本章从非技术角度讨论了 CORBA、Java 与 Web 的发展趋势。我们认为，这对于正确把握软件界的发展脉络应该是有益的。

由于 CORBA、COM/DCOM、Java RMI、Object Web 等技术都还在不断发展变化，我们已经可以停止等待，用目前已经具备的技术去构造现实中需要的电子商务软件了。不要忘记，Microsoft 成功的经验告诉我们：用户并没有要求我们去编写一个终极版软件，但是，如果用户喜欢我们的 Version1.0 版本，我们就一定会投资开发 Version2.0、Version3.0 的。