火龙果•整理
uml.org.cn

第一章　　绪 论

1.1 中间件

分布对象技术是伴随网络发展起来的一种面向对象的技术。以前的计算机系统多是单机系统，多个用户是通过联机终端来访问的，没有网络的概念。网络出现后，产生了 Client/Server 的计算服务模式，多个客户端可以共享数据库服务器和打印服务器。随着网络的进一步发展，许多软件需要在不同厂家的网络产品、硬件平台、网络协议异构环境下运行，应用的规模也从局域网发展到广域网。在这种情况下，中间件技术逐渐出现。中间件是位于操作系统和应用软件之间的通用服务，**它的主要作用是用来屏蔽网络硬件平台的差异性和操作系统与网络协议的异构性**，使应用软件能够比较平滑地运行于不同平台上。同时中间件在负载平衡、连接管理和调度方面起了很大的作用，使企业级应用的性能得到大幅提升。

中间件的思想其实并不复杂。假设我们有 n 个应用，m 个操作系统，为了使所有的应用在所有的操作系统上都能工作，就可能需要 n*m 个接口。而且，每引入一个新的操作系统，就要重新改写 n 个应用的源代码；每开发一个新的应用，就要考虑实现 m 个不同的版本，以工作于 m 个操作系统上。

中间件使这一切变得简单。开发应用程序时不必再关心底层操作系统的类型，而只需专心于应用的逻辑处理（当然，这只是一种理想状态）。

中间件的引入使原来的网状接口类型变成了沙漏状接口类型，如图 1.1 所示。接口数目从 n*m 降到 n+m。当 n 和 m 都很小的情况下，这种差异并不明显，但随着 n 和 m 不断地增大（正如现实世界正在发生的那样），这种差异将极大地降低开发应用软件的困难，并增加系统整体运行的效率和性能。因此，中间件的出现是分布式系统发展的产物，是软件构架演进的必然。

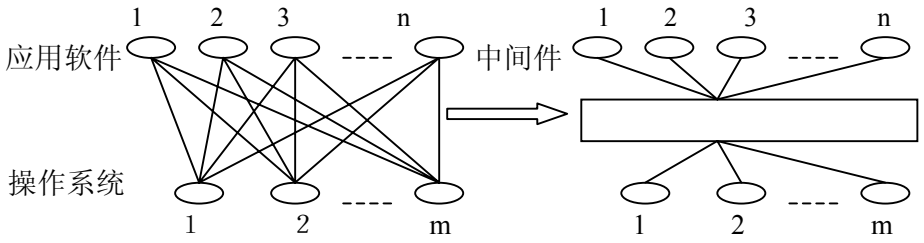


图 1.1 中间件的加入减少了接口数目

火龙果•整理
uml.org.cn

1.2 OMG 组织

对象管理组织英文名是 Object Management Group，简称为 OMG。OMG 是以美国为主体的非盈利性国际组织，其目标是为在计算机网络上独立开发的应用软件建立一个相互之间互操作性的标准。

OMG 成立于 1989 年，作为一个非营利性组织，集中致力于开发在技术上具有先进性、在商业上具有可行性并且独立于厂商的软件互联规范，推广面向对象模型技术，增强软件的可移植性（Portability）、可重用性（Reusability）和互操作性（Interoperability）。

该组织成立之初，成员包括 Unisys、Sun、Cannon、Hewlett-Packard、Philips 等在业界享有声誉的软硬件厂商，目前该组织拥有 800 多家成员。

最初，OMG 在 1990 年制订了对象管理体系结构 OMA。OMA 是比 CORBA 更高一层的概念，它定义了一种体系结构，在 OMA 之上可以用任何方法来实现。CORBA 是其中的一种实现方案。

1.3 对象管理体系结构 OMA

OMA 由对象模型（Object Model）和参考模型（Reference Model）组成。对象模型主要定义了如何描述在异种环境中的分布式对象；参考模型则刻画了对象之间的交互，其组成如下图所示：

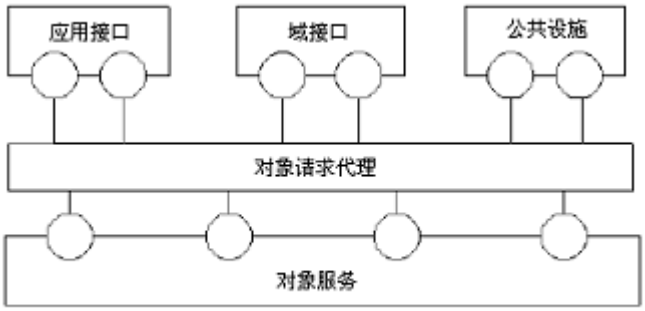


图 1.2 OMA 参考模型

其中对象请求代理作为对象互相通信的软总线，用来联系客户端和服务端间的通信。ORB 是 OMA 参考模型的核心，它保证在分布式异构环境中透明地向对象发送和接收请求，帮助实现应用部件之间的互操作。

在 ORB 之上有 4 个对象接口：

1） 对象服务 Object Services（OS）。定义加入 ORB 的系统级服务，如安全性、命名和事务处理，它们是与应用领域无关的。对象服务是一些最有可能被用来支持分布式对象环境下构造应用的标准化部件。目前通过的对象

火龙果•整理
uml.org.cn

服务包括对象命名服务、事件服务、对象生存期服务、永久对象服务、对象关系服务以及事务服务、并发控制服务等等。

2） 公共设施 Common Facilities（CF）。水平级的服务，定义应用程序级服务，如复合文档等。公共设施是比对象服务粒度更大的可重用的部件块，它主要用来帮助构造跨多个应用域的应用程序。典型的公共设施包括用户接口、信息管理、系统管理和任务管理等。

3） 领域接口 Domain Interfaces（DI）。它面向特定的领域，在 OMA 中所处的位置与对象服务与公共设施相似。

4） 应用接口 Application Interfaces（AI）。面向指定的现实世界应用。是指供应商或用户借助于 ORB、公共对象服务及公共设施而开发的特定产品，它不在 CORBA 体系结构中标准化。

1.4 公共对象请求代理体系结构 CORBA

CORBA 是一种中间件技术，全称是公共对象请求代理体系结构（Common Object Request Broker Architecture），它是由 IBM、HP 等多家公司联合开发的部件软件的体系结构和部件接口标准。支持 CORBA 的对象管理 OMG 已有 800 多家成员。

CORBA 的结构基础是 OMG 的核心对象模型（CORE object model）。部件可作为对象，以客户身份通过封装的接口提出请求，并获取服务对象所提供的服务，形成协同工作的系统。为了达到互操作性，设计了对象请求代理 ORB（Object Request Broker）作为通信的基础设施。ORB 能够帮助对象实现和网络上另一个对象的握手过程。它截获客户的请求，并负责找出实现其请求的服务对象；传送参数及调用方法给服务对象，并负责返回结果。客户不必知道服务对象是位于同一机器或是跨网位于不同机器上；也不要求客户与服务对象是否有同一编程语言、运行于何种平台、使用何种操作系统或有否其它特征。即由 ORB 来提供异步分布环境中不同机器上应用的互操作性。

CORBA 是建立在 OMA 概念之上，是 OMG 所采纳的第一个标准，同时也是 OMA 的一个重要组成部分。CORBA 的 OMA 定义了一个四层模型，并且规定了其 IDL 语言工作于各个层面。四层模型的底层是 ORB；直接建筑在 ORB 层之上的，是对象服务层（Object Service Layer），它负责对象的命名、创建和管理；第三层是公共设施层（Common Facilities Layer），在这一层完成用户对象直接使用的一些功能；而最高层是业务对象（Business Objects），像 Java 和 ActiveX applet 等。

对应 OMA 参考模型，CORBA 也相应的有 4 个对象接口，分别与 OMA 中四个对象接口对应：

1) CORBA 服务（CORBA Services）对应对象服务；
2) 应用对象（Application Object）对应应用接口；

火龙果•整理
uml.org.cn

3) CORBA 领域（CORBA Domains）对应领域接口；
4) CORBA 设施（CORBA Facilities）对应公共设施；

目前 CORBA 已经发展到 CORBA3.0，主要包括以下几个部分：

1) ORB 核心（core）；
2) OMG 接口定义语言 IDL；
3) 语言映射（Language Mapping）；
4) 运行机制；
5) 静态调用：包括桩和实现框架（Stub and Skeleton）；
6) 动态调用：包括动态框架和接口库（Interface Repository）；
7) 对象适配器（Object Adapter）；
8) Inter-ORB 协议；

1.5 CORBA 主要版本发展历程

1） 1990 年 11 月，OMG 发表《对象管理体系指南》，初步阐明了 CORBA 的思想；
2） 1991 年 10 月，OMG 推出 1.0 版，其中定义了接口定义语言（IDL）、对象管理模型以及基于动态请求的 API 和接口仓库等内容；
3） 1991 年 12 月，OMG 推出了 CORBA 1.1 版，在澄清了 1.0 版中存在的二义性的基础上，引入了对象适配器的概念；
4） 1996 年 8 月，OMG 基于以前的升级版本，完成了 2.0 版的开发，该版本中重要的内容是对象请求代理间协议（IIOP，Internet Inter-ORB Protocol）的引入，用以实现不同厂商的 ORB 真正意义上的互通;
5） 1998 年 9 月，OMG 发表了 CORBA 2.3 版，增加了支持 CORBA 对象的异步实时传输、服务质量规范等内容。
6） 2002 年 6 月，OMG 发表了 CORBA 3.0 版。

1.6 CORBA 的主要应用方向及产品介绍

CORBA 规范的推出，重新调整了客户机与服务器之间的关系。客户机可以向服务器提出事务请求，同时也可以为一个请求充当服务器角色。

由于 CORBA 系统引入了中间件的概念，即事务代理，由中间件完成客户机与服务器之间的通信，使得服务器对于客户机的位置相对透明，改变了原有分布式计算模型中客户机、服务器之间的一一对应关系。CORBA 客户机可以在运行时动态获得服务对象的位置，并且可以对多个服务对象提交事务请求，因此，极大推动了分布计算的发展。

分布计算是指网络中两个或两个以上的软件相互共享信息资源。这些软

火龙果•整理
uml.org.cn

件可以位于同一台计算机中，也可以部署在网络节点的任意位置。基于分布式模型的软件系统具有均衡运行系统负载、共享网络资源的技术优势。

另外，CORBA 规范约束采用面向对象的分布式软件的构造方法，以接口定义语言的形式实现对象内部细节的完整封装，从而降低了软件系统的复杂程度，增加了软件功能的可重用性。CORBA 提供到 C/C++、Java、SmallTalk 等高级语言的映射，很大程度上减小了对程序设计语言的依赖性，使软件开发人员可以在较大范围内共享已有成果。

正是以上特点推动了分布式多层软件体系结构的发展。目前，CORBA 技术在银行、电信、保险、电力和电子商务领域都有广泛的应用。

软件市场中能够见到的 CORBA 中间件产品很多，但基于不同公司的产品战略以及研发方向，各个产品在服务性能、对高级语言的支持和所依赖的系统平台方面有很大区别。下表对一些产品在上述几方面进行了分析：

产品	支持语言	采用协议	提供服务类型	支持平台
VisiBroker	IDL、C++、Java、COBOL	IIOP	Naming、Transactions、Event、Security 等	Solaris、Win98/NT、Linux 等
IBM	IDL、C/C++、Java、COBOL	IIOP	Naming、Externalization 等	Win98/NT、OS/2 等
Sun	IDL、C/C++、Java	IIOP	Naming、LifeCycle、Event、Properity	Solaris、Win98/NT
IONA	IDL、C++、Java、COBOL、Ada	IIOP	Naming、Trading、Event、Security 等	Solaris、Win98/NT、Linux 等
BEA	IDL、C++、Java	IIOP、DCE	LifeCycle、Security、Transactions 等	Solaris、Win98/NT、Linux 等
HP	IDL、C++、Java	IIOP、DCE	Naming、Trading、Event、Security 等	Solaris、Win98/NT、Linux 等

表 1.1 主要中间件产品分析

1.7 CORBA 与其它中间件技术比较

当前比较流行的分布式技术有：

1) Microsoft 提出的 COM/DCOM 技术；
2) OMG 提出的 CORBA 技术；
3) Sun 提出的 Java Beans 技术；

这其中 COM/DCOM 技术是微软操作系统的核心，利用该技术可以将由不同语言编写的部件合成到一个应用程序中。现在通用的 ActiveX 控件就是基于 COM/DCOM 技术。CORBA 技术是对象管理组织制定的有关对象技术的技术规范，它实现了分布式的部件技术。通过采用 CORBA 技术，对于异构系统中

火龙果•整理
uml.org.cn

的部件来说，它们之间的通信就好像是在本地一样。Java Beans 是 Sun 公司提出的基于 Java 的部件技术标准，Sun 公司的目标是实现一种与平台无关的部件技术。利用 JavaBeans 编写的部件可以存在于各种环境中，包括 Microsoft 的 Windows 环境，各种 Unix 环境，Macintosh 系统，JavaOS 系统，甚至是各种非典型的计算机系统，比如手机，机顶盒等可能的网络节点中。但是由于 Java 技术本身的发展，JavaBeans 技术还不是十分的成熟。

采用 CORBA 作为支撑环境的优点如下所示：

首先，对于跨平台的特性而言，CORBA 具有着先天的优势。对于任何流行的操作系统而言，用户均可获得相应的 CORBA 产品支持，甚至于可以有多种不同厂商的产品可供选择。对于 COM/DCOM 而言，虽然 Microsoft 声称该技术不局限于 Windows 平台，但是事实上，在 Unix 平台使用 COM/DCOM 技术是相当难的。而对于 JavaBeans，虽然由于 Java 具有跨平台的特性，使得 JavaBeans 技术在跨平台的特性上与 CORBA 不相上下，但是 JavaBeans 技术本身还在迅速发展，其自身还存在不足。

CORBA 简化了分布式应用的集成，对于最终用户而言，它更容易使用，因而在时间和成本上都有节约。利用 CORBA 技术集成历史遗留系统，如果该系统本身是基于 CORBA 技术的，或者该系统本身提供 CORBA 接口（许多大型的软件现在都提供 CORBA 接口，比如大型设计分析软件 Catia），那么与这样的遗留系统的集成本身就是无缝的，不需要额外的工作。若该遗留系统没有以上特点，需要在该系统的基础上作一个包装层，把该系统改进成一个 CORBA 对象。

其次，与原有的基于 RPC 机制的单纯的 C/S 结构相比，采用 CORBA 作为底层支撑结构，更有利于资源的灵活、合理的利用。因为 CORBA 是对等的分布式计算环境，所有对象之间的地位是平等的（并不是客户和服务器的固定角色），其担任的角色是可以转换的：当某一对象产生服务请求时就被称为客户方，而当它接受服务请求时就被称为服务方。绝大多数 CORBA 对象都可以担任客户方和服务方两种角色。

第三，CORBA 是一种标准，因此其核心元素的稳定性是有保证的。同时 CORBA 是面向对象的，这样面向对象的种种方便与强大功能都可以在 CORBA 中得到体现，例如系统的开放性，可重用性以及与原有系统的无缝集成。

本文将探讨 CORBA 平台。

1.8 本文的工作及其意义

本文之所以研究 CORBA 平台的实现，有以下几个原因：

1）CORBA 有着良好的开放性和扩展性。近年来，CORBA 又将其它中间件技术的特色吸收到所定义的服务中，同时它也注重不断融合最新出现的技

火龙果•整理
uml.org.cn

术，如 Java，UML，XML，组件技术和移动代理（mobile agent）等。CORBA 技术是先有标准，后有产品，这与许多的与 CORBA 竞争的中间件产品有很大的区别。CORBA 作为一个标准，在成为产品的过程中，虽然会出现一些性能不够理想或与其它产品沟通的问题，但是随着时间的推移，这些问题终将得到解决，其理论上的优势会更加明显地突现出来。目前，CORBA 技术在银行、电信、保险、电力和电子商务领域都有广泛的应用。

2）正如一些专家所说，国内软件领域应该多一些系统软件的研究，才能赶上甚至超过国际水平；CORBA 平台的设计正是系统级软件的设计，而国内 CORBA 方面研究主要还是以应用为主，探讨其实现的很少；

3）由于国内 CORBA 方面研究偏重于应用，以致大家对其内部工作方式缺乏了解，这将影响大家对 CORBA 的使用；

于是本文中打算实现一个微型的 CORBA 平台，完成 CORBA 平台最基本的功能。程序采用了较为灵活的程序结构，便于扩展，结构也很清晰，一方面可以使大家了解 CORBA 的运行机制,用于教学目的，另一方面也便于改动用作嵌入式等领域或完善成完整的 CORBA 平台。

本文把实现的微型 CORBA 平台称为 mincorba。

火龙果•整理
uml.org.cn

第二章 CORBA 规范简介

2.1 CORBA 基本组成与基本概念

本节首先给出 CORBA 的基本组成，然后介绍 CORBA 的基本概念。CORBA 的基本组成见下图：

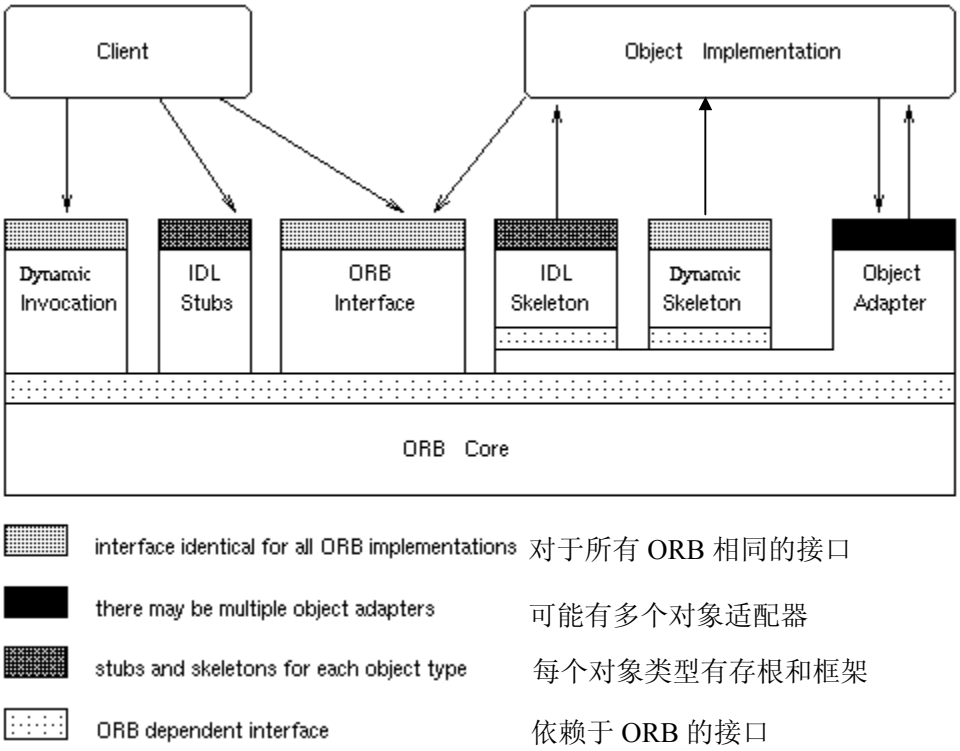


图 2.1 CORBA 的基本组成

2.1.1 CORBA 的工作方式

客户通过发送请求使用对象实现所提供的服务，具体实现过程中，客户可以使用静态调用或动态调用接口来实现，有时为了调用某些特殊功能，用户可能需要与 ORB 直接进行交互。

火龙果•整理
uml.org.cn

2.1.2 客户端码根(Stub)

也称为桩，把对某一对象类型(由接口定义)的 OMG IDL 操作定义映射到实现这些操作的过程化例程的具体语言映射。客户桩把对对象类型(由接口定义)的 OMG IDL 操作定义映射到过程化例程，这些例程会被调用来发送请求。其作用是作为客户应用程序访问 ORB 核心的纽带，进而提供了客户方静态的接口方法调用。

码根实际上是一段程序代码，为接口中的每一个操作（方法）提供一种虚实现。特点是：自动生成，静态的，与 ORB 的具体实现相关的。

2.1.3 服务器端框架（Skeleton）

是 IDL 接口对某种编程语言映射结果的服务器方内容，其作用是作为服务对象实现和 ORB 核心交互的纽带，进而提供了服务方静态的接口方法调用。

从功能上讲，框架实现了对象适配器的框架和方法分派的过程。

2.1.4 IDL(Interface Definition Language)

接口定义语言，用于描述分布对象或过程提供的服务，它规定了接口定义的语法和语义。在 CORBA、DCE、DCOM 等标准中均有各自的 IDL 定义。本文中提到的 IDL，都是 CORBA 的 IDL。它是一种中性语言，不依赖于任何具体的系统、语言和环境。

1）IDL 接口

是 IDL 语言中的具体语义内容，可以定义对象提供的服务，在接口中可以描述对象操作方法、属性及一些类型、常数等。

从语法上说，IDL 语言类似于 C++的语法，但是它不依赖于任何具体的系统、语言和环境。

2）IDL 编译器

是对象中间件产品中非常重要的开发工具，提供了 IDL 语言向具体编程语言映射的能力。由编译器产生的存根框架对各个产品是不同的，需要与用户程序联合编译。

2.1.5 ORB 核心 （ORB Core）

ORB 核心的有以下的作用

1）对象的定位：当客户端程序得到对象引用之后，会调用该对象引用，客

火龙果•整理
uml.org.cn

户端的 ORB 会根据对象引用中的信息来定位对象实现；

2）编码与解组：在客户端发起调用的时候，输入参数格式都与特定平台和特定语言有关，客户端 ORB 负责将它们编码成可以在网络上传送的格式（在线格式），到达服务器端的 ORB 后，服务器端 ORB 负责将这些在线格式还原成本地所使用的特定平台和语言的格式；

3）启动初始服务：在程序初启的时候，需要得到一些通用的对象引用，由于处于程序初启的特殊阶段，这些对象引用很难用常规的方法得到。ORB 可以通过伪对象接口 CORBA::ORB 提供两个操作(list_initial_services()和 resolve_initial_references())来完成这项功能。

4）屏蔽底层网络协议；

5）提供接口库和其它的 API：每一个 ORB 都有一个接口库，接口库负责存储用接口定义语言（IDL）定义的接口的信息，它还支持一些标准的 API，用来遍历或查询系统中的接口信息。

2.1.6 动态调用接口（DII）

有时客户端程序在编译时无法得知对象的详细信息，也没有 IDL 编译后的码根，这时，可以通过 DII 利用标准的 API 调用服务端对象。

2.1.7 动态框架接口（DSI）

是一套标准的 API，用来接收请求。

在 CORBA 中，客户端和服务器端是分开实现的，所以，一个由 DII 发起的请求可能由框架来处理，也可以由 DSI 来处理，同样，一个由码根发起的请求可能由 DSI 来处理，也可以由静态框架来处理。

2.1.8 对象适配器（OA）

对象适配器是对象实现访问 ORB 所提供的服务的主要方式，对象适配器向对象实现提供公共接口，向程序框架提供专用接口。如果 ORB 核心提供这项服务，则适配器只需提供服务接口，如果 ORB 核心不提供这项服务，则适配器必须在 ORB 核心之上实现它。

它有以下作用：

1）对象引用的生成与解释

2）根据对象引用找到它对应的对象实现

3）方法调用

4）交互的安全性

火龙果•整理
uml.org.cn

5）对象与实现的激活与终止激活
6）对象引用向相应的对象实现的映射
7）对象实现的注册

对象适配器可以有多个，对象实现可以选择合适的对象适配器，对象适配器的选择由对象实现所需要的服务而确定。

2.2 CORBA 几个重要接口的定义

概括介绍了 CORBA 的基本概念之后，下面看一看 CORBA 规范中的一些重要的接口都是如何定义的。也就是构建 CORBA 平台要实现的接口。

2.2.1 ORB 接口

该功能接口提供的功能对所有的 ORB 和对象实现都是一致的。接口 ORB 包括那些客户和服务器都能使用的操作，这些操作并不依赖于任何特定的对象适配器或对象引用。

a.对象引用与字符串的相互转换(由于对象引用是不透明的并可能因 ORB 而异，所以对象引用本身并不是一个便于存储在存储器中的值，也不是一个便于采用除调用以外的方式进行对象引用传递的值)

string object_to_string(in Object obj);
说明： 将对象引用转换成客户可以存储在其它媒介中的值
object string_to_object(in string str);
说明：转换成合适的对象引用

b.动态调用相关操作

1. void create_list(in long count,out NVList new_list);
说明： 该操作分配一个列表，在首次使用时清空列表
参数 count 为该列表分配的条目数
参数 new_list 为新生成的列表
2. void create_operation_list(in OperationDef oper,out NVList new_list);
3. void get_default_context(out Context ctx);
4. void send_multiple_requests_oneway(in RequestSeq req);
5. void send_multiple_requests_deferred(in RequestSeq req);
6. Boolean poll_next_response();
7. void get_next_response(out Request req);

c.获取服务信息

1.Boolean get_service_information(
In ServiceType service_type;

火龙果•整理
uml.org.cn

out ServiceInformation service_information);

说明：获取这个 ORB 所支持的 CORBA 设施和服务的信息

in 参数 service_type 传递要获取的信息的服务类型，其值是 CORBA 模块中定义的常数。 如果存在这种类型的服务信息，则在 out 参数 service_information 中返回，同时操作返回 TRUE, 如果不存在这种类型的服务信息，则操作返回 FALSE

2.ObjectIdList list_initial_services();

d.初始引用操作

1.Object resolve_initial_references(in ObjectId identifier)
raises(InvalidName);

e.类型码生成操作

f.线程相关操作

1.Boolean work_pending();

说明：该操作返回一个 ORB 是否需要主线程完成某些工作的指示

备注：非关键性特性，可避免使用，在最小 CORBA 中可以删除

2.void perform_work();

说明：如果被主线程调用，这个操作完成一个由实现定义的工作单元

备注：非关键性特性，可避免使用，在最小 CORBA 中可以删除

3.void run();

说明：该操作给 ORB 提供了执行资源以便于它执行内部功能.

4.void shutdown(in boolean wait_for_completion);

说明：该操作命令 ORB 关闭，停止处理准备析构

如果参数 wait_for_completion 为 TRUE,该操作阻塞直至关闭动作完成，如果该参数为 FALSE,则 shutdown 返回时关闭处理可能还没有完成。

5.void destroy();

说明：该操作删除了 ORB 以便于使应用重新使用这些资源。

g.策略相关操作

1.Policy create_policy(in PolicyType type,in any val)
raises(PolicyError);

2.2.2 OBJECT 接口

OBJECT 代表对象引用，在对象引用上所进行的操作由 ORB 直接实现，而不是传递给对象实现。

1. 查询对象接口

InterfaceDef get_interface();

火龙果•整理
uml.org.cn

说明：返回接口库中的一个对象

备注：最小 CORBA 中可以删除

2. 对象引用的复制

Object duplicate();

3. 对象引用的释放

void release();

4. 空对象引用

Boolean is_nil();

说明：检验当前对象引用是否是这个值

5. 等价检验操作

Boolean is_a(in ReponsitoryId logical_type_id);

说明：此操作用来维护对象引用的类型安全。

备注：非关键性特性，可避免使用，在最小 CORBA 中可以删除

6. 对象存在性探测

Boolean non_existent();

说明：用于检验一个对象是否已经被释放.

备注：非关键性特性，可避免使用，在最小 CORBA 中可以删除.

7. 对象引用标识(目的是为了有效地管理包括大量对象引用的服务的状态)

1）unsigned long hash(in unsigned long maximun);

说明：对象引用与 ORB 内部标识符相关联，应用程序可以使用 hash()操作间接访问这些标识符，maximum 参数规定了 ORB 返回的散列值的上界，散列值的下界为零

2）Boolean is_equivalent(in object other_object);

说明：用于判断两个对象引用是否相等，如果目标对象引用和作为参数传递的另一个对象引用等价，该操作返回 TRUE,否则返回 FALSE

8. 创建请求

```
void create_request(in Context ctx,
                    in Identifier operation,
                    in NVList arg_list,
                    inout NamedValue result,
                    out Request request,
                    in Flags req_flag);
```

9. 与对象关联的策略获取

Policy get_policy(in PolicyType policy_type);

说明：返回指定类型的策略对象

10. 对象引用关联策略的重载

火龙果•整理
uml.org.cn

Object set_policy_overrides(in PolicyList policies,
in SetOverrideType set_add);

说明：该操作返回一个新的对象引用，该对象引用有一个与它相关的新策略参数 policies，是策略对象的一个引用序列 ；参数 set_add 表明这些策略是否应该被加到已存在的对象引用的任何一个重载上，还是应该被加到一个清除重载释放对象引用上。

11. 与对象关联的域管理器的获取

DomainManagersList get_domain_managers();

说明：允许管理服务（和应用）检索域管理器，进而获得适用于该域中各个成员对象的安全以及其它管理策略。

2.2.3 动态调用接口（DII）

DII 允许动态构造和调用对象请求，这一服务是 CORBA 内核的组成部分，在最小 CORBA 中该接口可以删除。

支持这些服务的方法分布在 4 个伪对象接口中：

1. CORBA::Object 定义每个 CORBA 对象都必须支持的方法
2. CORBA::ORB 定义客户端和服务器都能使用的操作，它们是由内核实现的方法
3. CORBA::Request 定义远程对象操作
4. CORBA::List 用于构建参数列表

其中分布在接口 1 和接口 2 中的操作，一部分属于 ORB 接口环境，另一部分属于 DII 环境；分布在接口 3 和接口 4 中操作只属于 DII 环境。

DII 环境	ORB 接口环境
.create_request	.is_nil
.get_interface	.release
	.is_a
	.non_existent
	.is_equivelent
	.hash
	.duplicate

表 2.1 CORBA::Object

火龙果•整理
uml.org.cn

DII 环境	ORB 接口环境
.create_list	.object_to_string
.create_operation_list	.string_to_object
.get_default_context	.list_initial_services
.send_multiple_request_oneway	.resolve_initial_references
.poll_next_response	
.get_next_response	

表 2.2　CORBA::ORB

动态调用接口的分布:

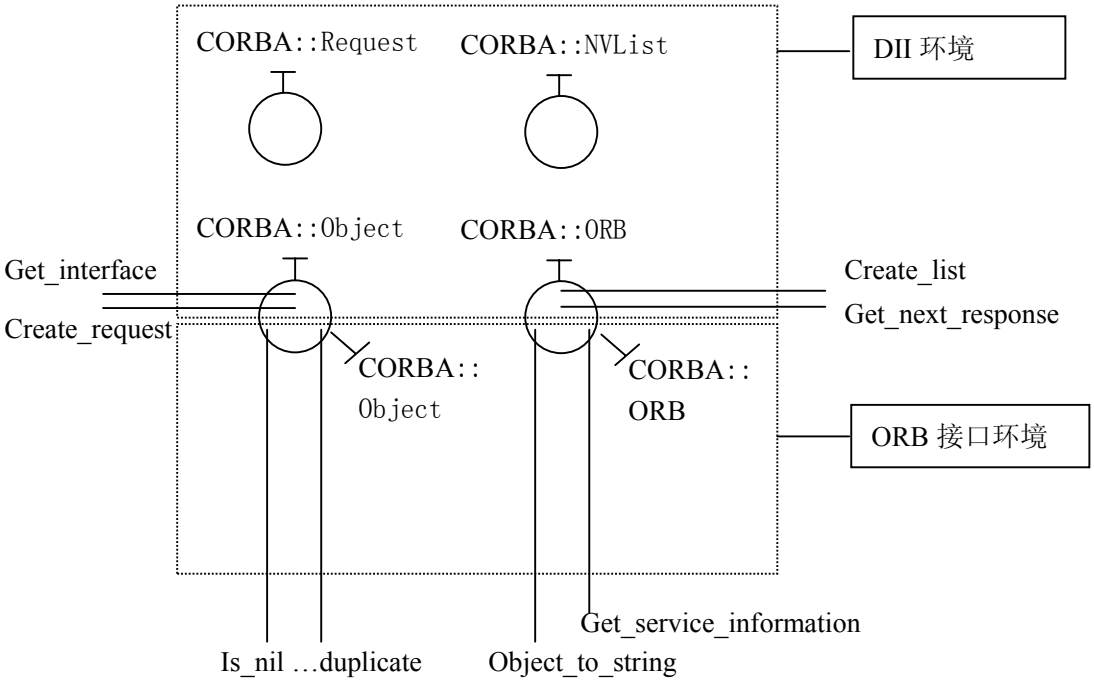


图 2.2　动态调用接口的分布

◇CORBA::Object

1. void create_request(in Context ctx,
   …
   out Request request,
   …
)

   说明：此操作创建了一个 ORB 请求,真正的调用是通过调用 invoke 或者使用 send/get_response 调用引起的。

2. InterfaceDef get_interface();

火龙果•整理
uml.org.cn

说明：获得目标对象所支持的接口

◇CORBA::ORB

1. void create_list(in long count,out NVList new_list);
   说明：该操作分配一个列表，在首次使用时清空列表
   参数 count 为该列表分配的条目数
   参数 new_list 为新生成的列表
2. void create_operation_list(in OperationDef oper,out NVList new_list);
   说明：该操作返回一个以给定操作的参数说明初始化的 NVList
3. void send_multiple_requests_oneway(in RequestSeq seq);
   说明：发送多路请求信息
4. void send_multiple_requests_deferred(in RequestSeq seq);
   说明：发送多路请求/应答信息
5. boolean poll_next_response();
   说明：轮询请求已经完成，返回 TRUE 说明至少有一个已经完成
6. void get_next_response(RequestSeq*&) raise(WrongTransaction)
   说明：返回完成的下一个请求

◇CORBA::Request

1. void add_arg(in Identifier name,
   ……
   );
   说明：向请求逐步添加参数
2. void invoke（in Flags invoke_flags）;
   说明：此操作调用 ORB,ORB 进行函数解析并调用适当的函数。如果这个函数成功返回，它的结果被放在 create_request 指定的 result 参数中
3. void send(in flags);
   说明：send 根据请求中的信息发起一个操作，该操作不需要等待操作结束就把控制返回给调用者
4. boolean poll_reponse(in request req)
   说明：判断请求是否已成功
5. void get_response（）raises (WrongTransaction);
   说明：返回一个请求的结果
6. void delete（）;
   说明：此操作删除请求，与请求相关联的所有内存也被释放

◇CORBA::NVList

1. void add_item(…);
   说明：该操作将一个新条目加入到指定的列表中，该条目被加入到前一个加入的条目的后面

火龙果•整理
uml.org.cn

2．void free();
说明：该操作释放列表结构和任何相关联的内存
3．void free_memory();
说明：该操作释放所有动态分配的与列表相关的输出参数内存，但不释放列表结构本身
4．void get_count (out long count);
说明：该操作返回加入到这个列表的条目的总数
参数 count 返回表中条目数

◇上下文对象

一个上下文对象包含一个特性列表，每个特性有一个名字和与那个名字相关联的一个字符串值组成。上下文特性代表了客户、环境或者不便于作为参数传递请求情况的信息。

1．void get_default_context (out Context ctx);
说明：该操作返回一个缺省进程上下文对象的引用。缺省上下文对象可以被连入其它上下文对象中。
2．void set_one_value(in Identifier prop_name,in string value);
说明：该操作设置一个上下文对象特性。
参数 prop_name 表示要增加的特性名称
参数 value 表示要增加的特性值
3．void set_values(in NVList values);
说明：该操作设置上下文对象中的一个或者多个特性值
参数 values 表示要更改的特性名称
4．void get_values(in Identifier start_scope,
…);
说明：该操作获取指定的上下文特性值
5．void delete_values(in Identifier prop_name);
说明：该操作从上下文对象中删除指定的特性值
参数 prop_name 为要删除的特性名称
6．void create_child(in Identifier ctx_name，out Context child_ctx);
说明：该操作创建一个子上下文对象,返回的上下文对象被连入到它的父上下文中
参数 ctx_name 表示上下文对象名称
参数 child_ctx 表示新生成的上下文对象
7．void delete (in Flags del_flags);
说明：该操作删除指定的上下文对象
参数 del_flags 为标志，控制删除操作

火龙果•整理
uml.org.cn

2.2.4 动态程序框架接口(DSI)

DSI 的主要接口是 ServerRequest 伪对象，不再详述。

2.2.5 接口库（IR）

接口库是一个联机工作的对象定义数据库，其中所有的对象都是 IDL 源文件中的信息的编译文本。CORBA 为 IDL 结构中的每一种数据类型都定义了一个接口定义对象,CORBA 把接口组合成模块，再把信息分组到表示命名空间的模块中。CORBA 规定了一个 Repository 接口，作为所有模块的根,它能提供到接口库的全局访问的接口。由于对象名字在模块中是唯一的，因此可以通过名字来搜索模块。

在接口库中有些接口是用来作为 IR 中对象的基接口（基接口不能实例化），包括 IRObject、Contained、Container、IDLType、TypedefDef。所有的接口库对象都是从 IRObject 接口继承而来。

在基接口中定义一组公共操作，该操作用来在接口库中寻找对象，IRObject 提供了一个可以识别本对象的实际类型的操作，作为包容器的对象从 Container 接口继承了搜索操作，被其它对象包容的对象则从 Contained 接口继承搜索操作。

IDLType 接口被所有代表 IDL 类型的接口库对象继承，包括接口、类型定义等。

TypedefDef 接口被所有的有命名的非接口类型继承。

接口对象的名称和功能：

对象名称	功　　能
ModuleDef	定义接口的逻辑分组、按名字进行该分组的导航
InterfaceDef	定义对象的接口，包括常量定义对象、类型定义对象、异常定义对象等
OperationDef	定义对象接口中的操作，包括参数及操作引起的异常事件
ParameterDef	定义操作的参数
AttributeDef	定义接口的属性
ConstantDef	定义有名常量
ExceptionDef	定义可能有操作引起的异常事件
TypeDef	定义有名类型

表 2.3　接口对象的名称和功能

接口库顶层类分层结构：

火龙果•整理
uml.org.cn

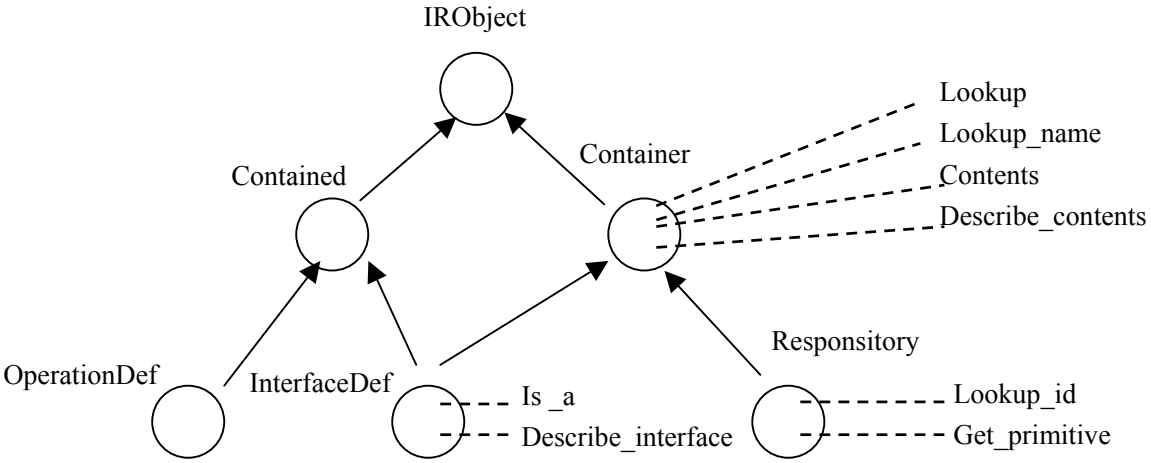


图 2.3　接口库顶层类分层结构：

一种动态调用的流程参考：

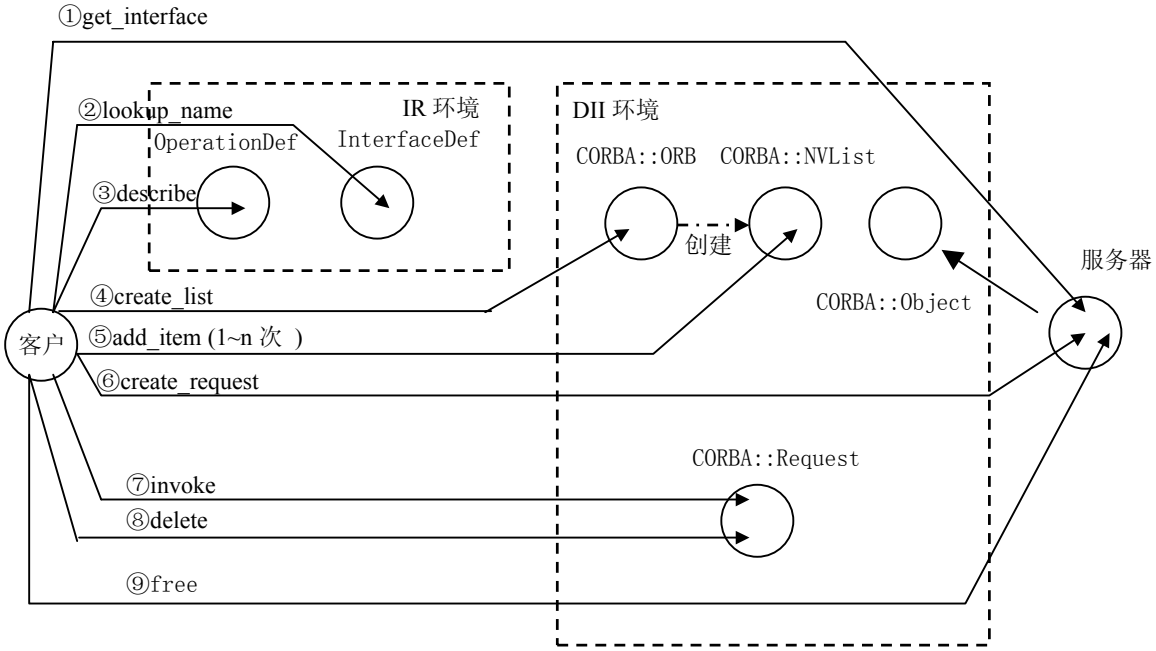


图 2.4 一种动态调用的流程参考

第一步：查询接口定义信息

① 客户调用目标对象中的 get_interface 操作，得到一个存放在接口库中的 InterfaceDef 对象，从而获得目标对象的接口描述信息；

第二步：查找操作的描述信息

② 以 InterfaceDef 对象作为接口库导航的入口点，找出有关的接口对象以及它所支持的操作。（调用 InterfaceDef 对象中

火龙果•整理
uml.org.cn

lookup_name 操作，得到 Operationdef 对象）；

③调用 OperationDef 对象中的 describe 操作，获得需要调用的操作的完整的描述，即操作的全部 IDL 定义；

第三步：准备参数列表

④调用 ORB 接口中用于动态调用的 create_list 方法，创建一个空闲的参数列表，即 NVList 对象；

⑤调用 NVList 对象中的 add_item 操作，将各个请求参数逐一填入参数列表中；

第四步：创建请求对象

⑥调用目标对象上的 create_request 操作，构造一个请求对象；

第五步：调用已建立的请求

⑦调用 ORB::Request 对象中的 invoke 操作，发起对请求的调用；

第六步：善后工作

⑧执行 ORB::Request 对象上的 delete 操作，释放请求对象和相关的内存空间；

⑨调用 CORBA::NVList 对象中的 free 操作，释放列表结构的相关的内存空间；

2.2.6 可移植对象适配器（POA）

1.BOA 适配器存在的问题

1)可移植问题

不同厂商的 ORB 实现对 BOA 的解释也不尽相同，导致服务器端的对象实现不能在不同的 ORB 产品进行移植。

2)对象标识的永久性问题

无法控制对象引用的生成，没有给对象一个永久的标识，给实现跨服务器生命周期的对象（永久对象）带来了困难。

3)对象实现粒度

BOA 缺乏对服务器端对象实现粒度的支持。

服务器端的对象实现粒度是指客户端看到的一批服务器端对象，对象是只用一个实现对象来实现，还是其中每个对象都用单独的实现对象来实现。

BOA 只能向客户提供数量有限的服务器对象。

2.POA 的抽象模型

火龙果•整理
uml.org.cn

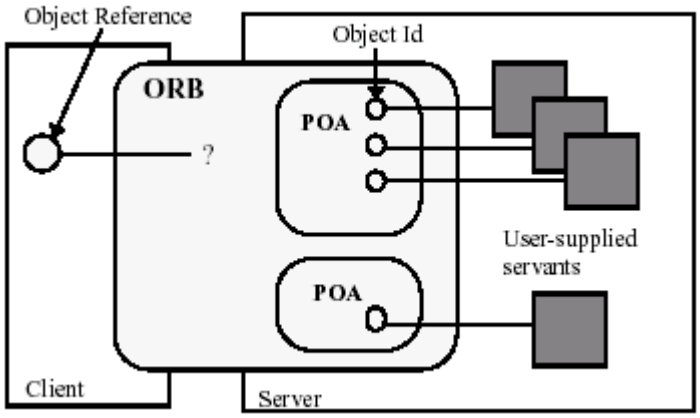


图 2.5　POA 的抽象模型

POA 抽象模型包括以下组件：

1.客户

2.服务器：一个包含对象实现的计算环境，通常一个服务器对应一个进程。

3.对象：是一个具有标识、接口和实现的程序实体。

4.对象标识：用来识别 CORBA 对象的一个值。从客户的角度看，对象的标识封装在对象引用中。从服务器的角度看，对象标识是通过 POA 接口由对象实现直接管理的。

5.对象引用(OR)：（与上同），具体封装了一个 ObjectId 和一个 POA 标识，ORB 用这些信息来识别并寻找服务器和与对象相关的 POA。

6.仆从(Servant)：仆从是实现一个或多个对象请求的编程语言对象或实体，通常存在于服务器进程环境中。ORB 将客户端根据对象引用发出的操作请求转换为对仆从的调用。一个对象可能与多个仆从相联系。

7.策略：它是应用程序于 POA 对象相联系的对象。POA 总共有 7 个策略。

8.POA:它是在服务器环境中的一个可识别的实体。

9.POA 管理器：它是封装了一个或多个 POA 的处理对象。开发人员可以通过 POA 管理器的操作对 POA 的请求进行排队或丢弃。

10.仆从管理器：ORB 利用仆从管理器的操作来激活和终止一个仆从。仆从管理器用来管理以 ObjectId 表征的对象与特定仆从之间的联系，并负责确定一个对象是否存在。

11.适配器激活器：当 ORB 收到一个对当前不存在的子 POA 的请求时，调用适配器激活器中的操作，创建所需的 POA。

3.POA 的体系结构(每一个 POA 都可以创建新的 POA，通过这种创建关系形成一个 POA 体系接构)。

火龙果•整理
uml.org.cn

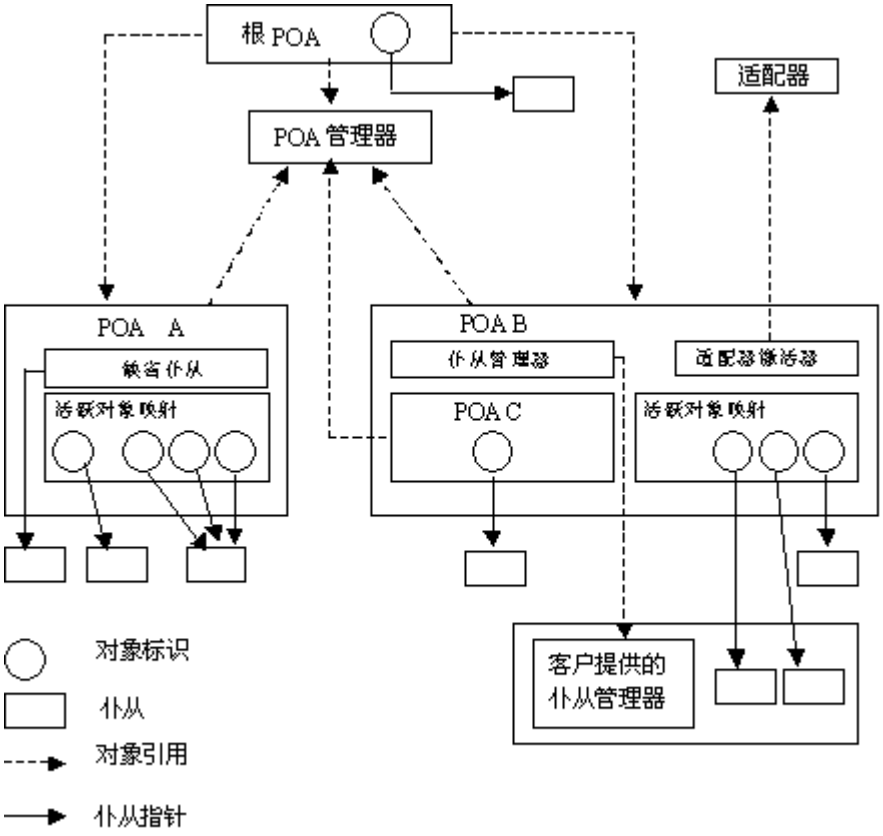


图 2.6　POA 的体系结构

4. 仆从的建立，使用和终止

火龙果•整理
uml.org.cn

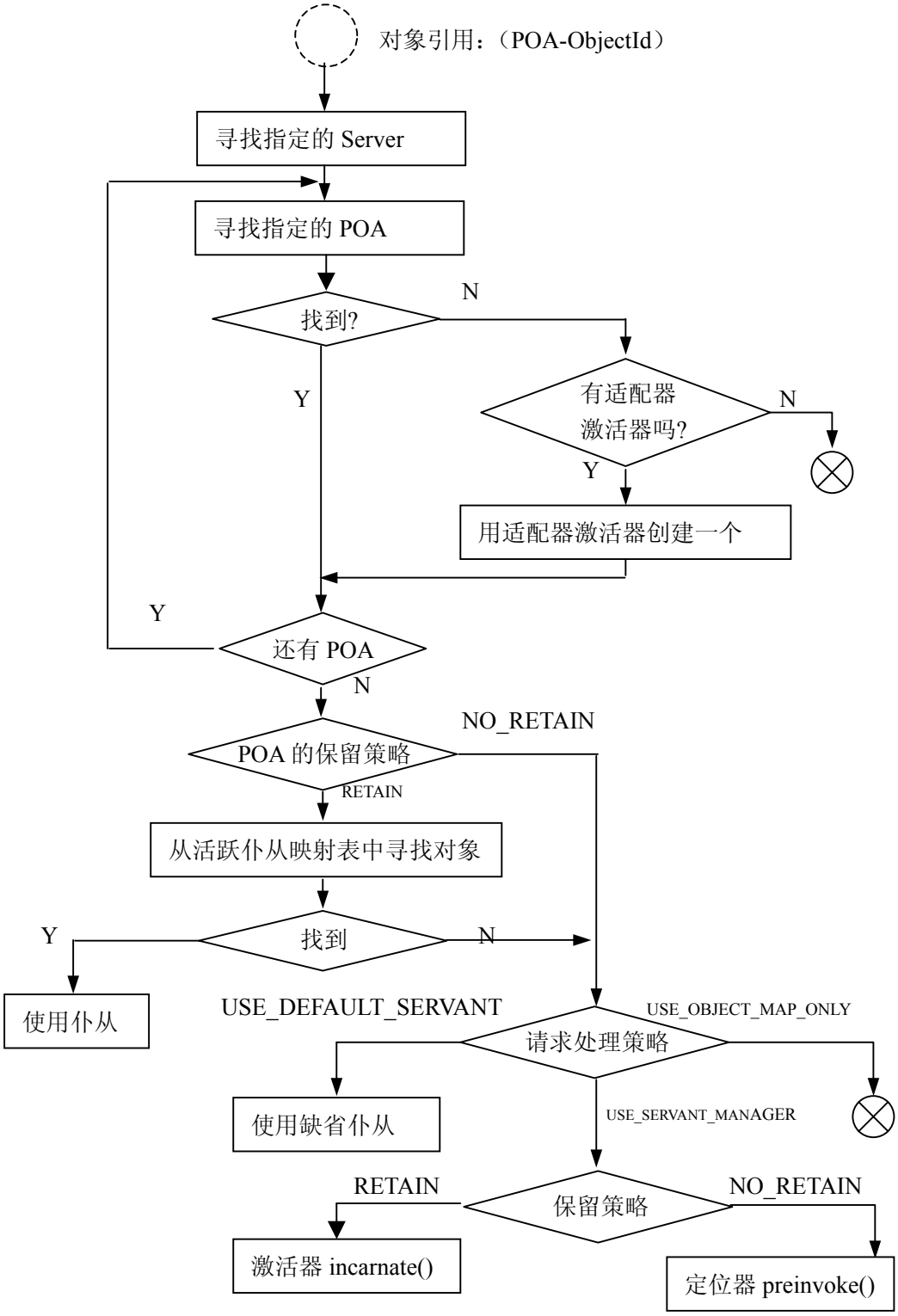


图 2.7　　仆从的建立，使用和终止

火龙果•整理
uml.org.cn

仆从激活器：要求 POA 具有 RETAIN 策略，在仆从激活器中寻找并返回一个仆从后，它会将这个仆从与 ObjectId 的映射关系保存到活跃对象映射表中，这样，当具有相同的 ObjectId 的请求在以后到达时，ORB 可以在活跃对象表中找到这个 ObjectId 以及与之关联的仆从，然后直接使用这个仆从。

仆从定位器：要求 POA 具有 NO_RETAIN 策略，它并不把找到的仆从保存在活跃对象表中。它需要对每一次调用作出相应。

活跃仆从：指已经存在内存中，并通过相关的对象标识提供给 POA 的程序对象。POA 有一个活跃对象映射表，它将对象标识和活跃仆从关联起来，每个关联构成一个活跃对象。

2.2.7 GIOP 协议

GIOP（General Inter-ORB Protocol）协议定义了不同 ORB 之间进行通信的标准的传输语法（数据的低级表示标准）和一套消息格式。该协议定义公共数据表示（Common Data Representation，简称 CDR）；同时，它还定义了可互操作的对象引用（Interoperable Object Reference，简称 IOR），以便可以在不同的支持 CORBA 标准的 ORB 之间进行交换。

GIOP 有三个版本：1.0 、1.1 和 1.2。这就意味着我们的消息根据其符合的 GIOP 版本可能有不同的格式。

1. GIOP 规范的组成：

1)传输假设：传输层是面向联接的、可靠的；

2)公共数据表示（CDR，Common Data Representation）：是 CORBA 调用中将使用的数据类型的正式映射。是用字节流(消息、封装)表示 IDL 数据类型的格式，是二进制的低级表示。

3)消息格式：便于对象请求、定位对象实现和管理通信信道。GIOP 消息类型：

消息类型	发起者	值	GIOP 版本
Request	客户机	0	1.0,1.1,1.2
Replay	服务器	1	1.0,1.1,1.2
CancelRequest	客户机	2	1.0,1.1,1.2
LocalRequest	客户机	3	1.0,1.1,1.2
LocalReplay	服务器	4	1.0,1.1,1.2
CloseConnection	服务器	5	1.0,1.1,1.2
MessageError	两者	6	1.0,1.1,1.2
Fragment	两者	7	1.1 , 1.2

表 2.4　GIOP 的消息类型

GIOP 请求消息分成三部分：GIOP 消息头、GIOP 请求头和请求消息体；

火龙果•整理
uml.org.cn

GIOP 应答消息分成三部分：GIOP 消息头、GIOP 应答头和应答消息体。下面是 IDL 描述的 GIOP 的结构：

```
Module GIOP{
    enum   MsgType{
            Request,Reply,CancelRequest,
            LocateRequest,LocateReply,
            CloseConnention,MessageError
    };
    struct MessageHeader {
            char                        magic[4];
            Version                     GIOP_version;
            boolean                     byte_order;
            octet                           Message_type;
            unsigned long                   messge_size;
    };
    struct  RequestHeader{
            IOP::ServerviceContextList  service_context;
            unsigned    long                request_id;
            boolean                     response_expeceted;
            sequence<octet>         object_key;
            string                          operation;
            Principal                       requesting_principal;
    };
    enum ReplyStatusType{
            NO_EXCEPTION,
            USER_EXCEPTION,
            SYSTEM_EXCEPTION,
            LOCATION_FORWORD
    };
    struct ReplyBeader {
            IOP::ServiceContextList     service_context;
            unsigned long                   request_id;
            ReplyStatusType             reply_status;
    };
    struct  CancelRequestHeader {
```

火龙果•整理
uml.org.cn


```
        unsigned    long                    request_id;
    };
    struct LocateRequestHeader {
        unsigned long                       request_id;
        sequence<octet>             object_key;
    };
…
```

2. IIOP 协议

IIOP（Internet Inter-ORB Protocol）协议则定义了 GIOP 协议是如何通过 TCP/IP 连接进行传输的，是 OMG 组织为支持 CORBA 标准的 ORB 的对象实现之间进行互操作而制订的标准。IIOP 声明了在 Internet 上进行互操作的协议标准，它提供了与所有其它兼容 CORBA 标准的 ORB 产品的透明的互操作协议。

GIOP 与 IIOP 的关系有点象将 OMG 中的 IDL 语言与一种特定程序设计语言的关系。GIOP 协议可以映射到多种不同的传输协议，而 GIOP 协议本身并不能提供完全的互操作，就象 IDL 语言本身不能构造一个完整的程序一样。IIOP 协议和其它将要被映射的传输协议一样，都是抽象 GIOP 协议的具体实现。

3. IOR

在 IIOP 协议的实现中，CORBA 标准中定义了如何将对象引用 IOR 转化为字符串和再将字符串转化为 IOR 的标准。该操作分别由 ORB::object_to_string 和 ORB::string_to_object 实现。IOR 字符串由前缀 “IOR:”加上将按 IDL 定义的对象引用(OR)编码后的一串 16 进制的字符串组成。

2.2.8 小结

上面介绍了 CORBA 的一些重要接口，本文并不打算完全实现这些接口，只实现了其中的一部分，下章将会具体说明。

火龙果•整理
uml.org.cn

第三章 mincorba 总体设计

3.1 mincorba 要实现的接口

实现一个完整的 CORBA 平台十分复杂，本文只计划实现一个微型 CORBA 平台。我把实现的 CORBA 平台称为 mincorba。

mincorba 希望最大限度的保持 CORBA 的可移植性和互操作性，对 CORBA 规范的最大删减是去除了 CORBA 中的动态特性。因为 CORBA 的静态调用(stub 和 skeleton)和动态调用(DII 和 DSI)在功能上可以看作是一对冗余。删除的部分包括：

。Dynamic Skeleton Interface
。Dynamic Invocation Interface
。Dynamic Any
。Interceptors
。Interface Repository
。Advanced POA features
。CORBA/COM interworking

但是：

工作在 mincorba 之上的客户只使用存根方式与工作在 CORBA 上的服务器交互，不过框架服务器则可以使用静态框架方式，也可以使用动态框架方式；

工作在 mincorba 之上的服务器只使用框架方式与工作在 CORBA 上的客户交互，不过客户则可以使用静态存根方式，也可以使用动态存根方式。

下面是最 mincorba 删减功能一览表：

模块	接口	删减的操作和映射	删减原因
ORB Interface	ORB	Create_list()	已不支持 DII
		Create_operation_list()	
		Work_pending()	非关键特性，可避免使用
		Perform_work()	
		Shutdown()	
		Context	已不支持 DII
		Get_default_context()	
	Object	Get_interface()	已不支持接口库
		Get_implementation()	在 CORBA2.2 中已过时
		Is_a()	非关键特性，可避免

火龙果•整理
uml.org.cn

		Non_exsistent()	使用
		Create_request()	已不支持 DII
	ConstructionPolicy	全部	非关键特性，可避免使用
DII	全部	--	已不支持动态特性
DSI	全部	--	已不支持动态特性
DynamicAny	全部	--	已不支持动态特性
接口库	全部(除了RepositoryId 和TypeCode)	--	已不支持动态特性
POA	POA	Create_thread_policy()	只支持策略的缺省值，因此不需要这些操作
		Create_impicit_activation_policy()	
		Create_servant_retention_policy()	
		Create_request_processing_policy()	
		The_activator(属性)	不支持 POA 的动态激活
		Get_servant_manager()	不支持ServantManager
		Set_servant_manager()	
		Get_servant()	不支持USE_DEFAULT_SERVANT 策略值
		Set_servant()	
	接口 ThreadPolicy	全部	只支持原来 CORBA 中这些策略的缺省值，因此这些策略没有存在的必要(仍支持其它的策略对象 ,LifeSpanPolicy,ObjectIdUniquenessPolicy,IdAssignmentPolicy。它们的策略取值也与CORBA 中的相同)
	接口ImplicitActivationPolicy	全部	
	接口ServantRetentionPolicy	全部	
	接口RequestProcessingPolicy	全部	
	接口 POAManager	全部(除了操作 activate 和异常 AdaptorInactive)	接口 POAManager 存在的唯一理由是 create_POA()操作中

火龙果•整理
uml.org.cn

			用到了它。为保证应用在最小CORBA和CORBA之间的可移植性，保留了activation操作
	接口AdapterActivator	全部	不支持POA的动态模式
	接口ServantManagers	全部	
	接口ServantActivator	全部	
	接口ServantLocator	全部	
CORBA互操作	DCE ESIOP	全部	不常用(但保留了除DCE ESIOP之外的几乎所有CORBA互操作特性，以支持最小CORBA与CORBA之间的完全互操作)
COM/CORBA互操作	全部	--	不常用
拦截器	全部	--	不常用
语言映射		那些与前面被删减部分有关的映射部分	--

表3.1 mincorba的删减

3.2 CORBA的实现方式

CORBA的实现方式常见的有以下几种：

1）基于服务的ORB；

2）基于操作系统的ORB；

3）基于程序库的ORB；

本文采用的是基于Java类库的ORB。即ORB的实现是以JDK类库形式存在的。mincorba由Java写成。

JDK包中包含有CORBA包，里面是Java描述的CORBA接口框架，不能运行，需要实现这些接口之后才可以执行。实现CORBA就是要实

火龙果•整理
uml.org.cn

现一些类来完成这些接口定义的操作。

在设计好的 CORBA 平台运行过程中，作者设计的类与 JDK 已有的 CORBA 类会互相调用。

此外，由于工作量的原因，本文利用的是 JDK 自带的 idlj 编译器，并没有自己设计一个 IDL 编译器。

3.3 mincorba 各个类与接口说明

下面将分模块描述 mincorba 中的各个类。我使用了 JBuilder7 绘制这些 UML 图。

图中图标的意义如下：

protect 属性；

public 属性；

private 属性；

构造函数；

public 方法；

protect 方法；

get 某属性的方法；

加下划线的是静态属性或方法；

3.3.1 ORB 类

火龙果•整理
uml.org.cn

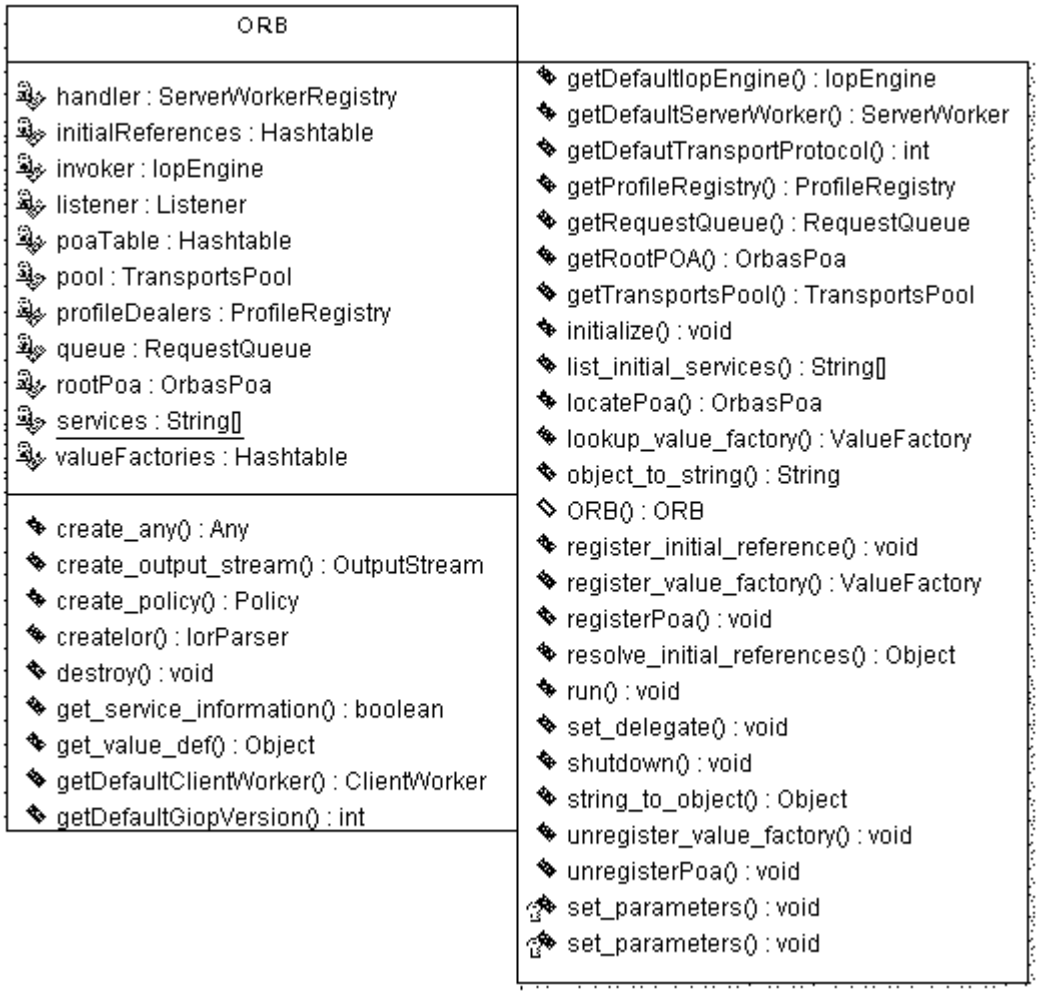


图 3.1　ORB 类

ORB 模块是实现 CORBA 规范中定义的 ORB 接口的功能模块。此外，ORB 模块还要管理其它所有的部分，它是整个 mincorba 的中心。后面将阐述它和其它部分的关系。

火龙果•整理
uml.org.cn

3.3.2 IOP 引擎

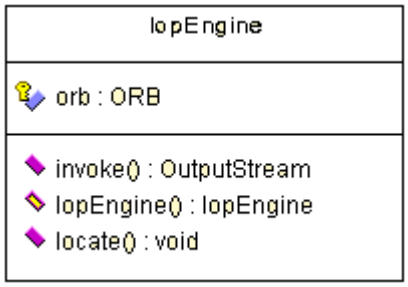


图 3.2 IopEngine 类

本文中 GiopServerWorker 等类将利用 IOP 引擎找到 ORB 和 POA，最终找到对象实现。

3.3.3 ORB 与 IOP 引擎之间的接口

Worker 对象、ClientWorker 对象和 ServerWorker 对象共同构成 ORB 对象与 IOP 消息引擎之间的接口。

1．Worker 类：

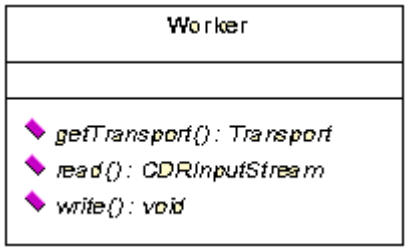


图 3.3 Worker 类

Worker 对象有 3 个成员：

1）public abstract Transport getTransport()，返回当前进行所有网络访问的缺省 Transport 对象。这是一个虚函数。它是 IOP 引擎与底层传输协议之间的接口之一。

2）public abstract CDRInputStream read()：从网络中读取一条消息，并构造 org.com.mincorba.cdr.CDRInputStream 对象返回。

3）abstract void write(org.omg.CORBA.portable.OutputStream out)：将参数所表示的流输出到网络中。

2．ClientWorker 接口：

火龙果•整理
uml.org.cn

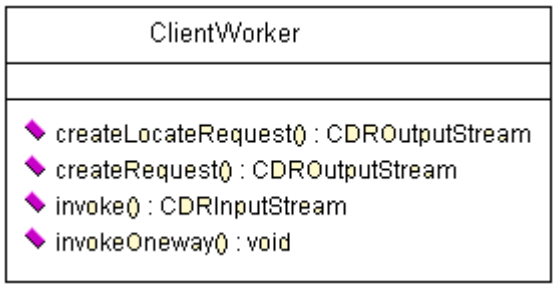


图 3.4 ClientWorker 接口

ClientWorker 接口完成一个 IOP 客户端的功能，它的功能包括创建各种 IOP 消息的 CDR 编码流以及引发远程对象的操作（invoke 和 invokeOneway 操作）。

3．ServerWorker 接口

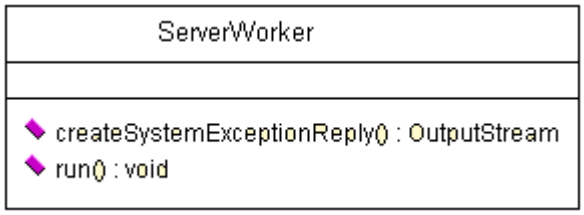


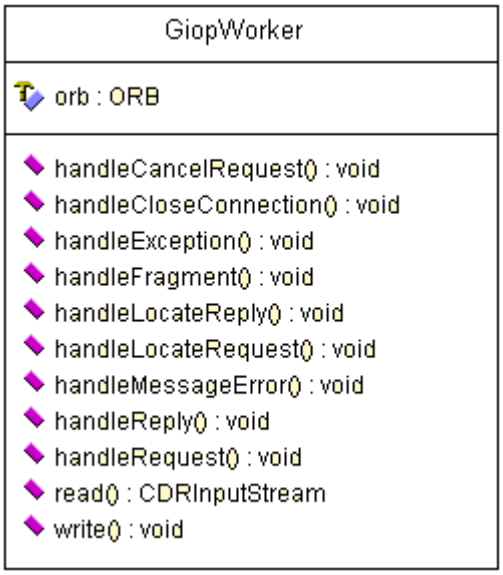
图 3.5　ServerWorker 接口

ServerWorker 接口实现了 IOP 服务器端的功能。

3.3.4 GIOP 类

GIOP 模块是 ORB 与 IOP 引擎接口组的一个具体的实现。

1．GiopWorker 类



火龙果•整理
uml.org.cn

图 3.6 GiopWorker 接口

GiopWorker 类是 org.com.mincorba.orb.Worker 类的子类，它仍然是一个抽象类。它实现了 Worker 的 read 和 write 操作。

2．GiopClientWorker 类

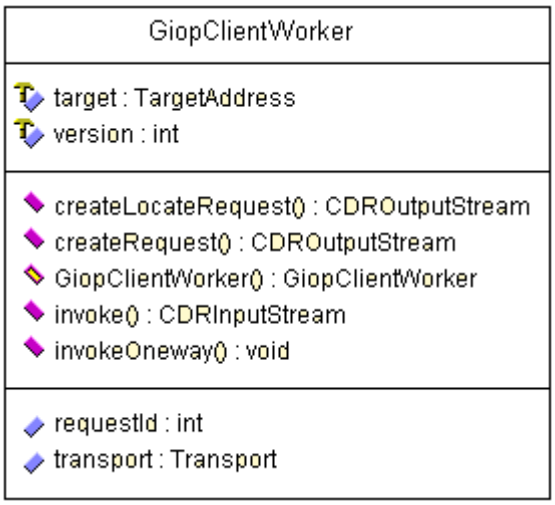


图 3.7　GiopClientWorker 类

GiopClientWorker类继承了GiopWorker抽象类并实现org.com.mincorba.orb.ClientWorker接口。它还实现了org.com.mincorba.orb.Worker类 的getTransport（）方法。

3．GiopServerWorker 类

火龙果•整理
uml.org.cn

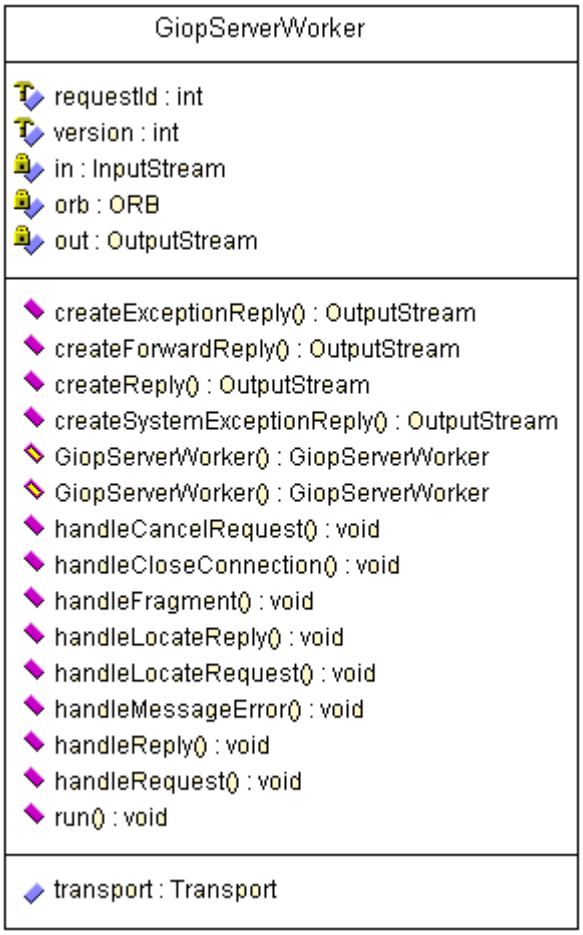


图 3.8 GiopServerWorker 类

GiopServerWorker 类继承了 GiopWorker 抽象类并实现了 org.com.mincorba.orb. ServerWorker 接口。它还实现了 org.com.mincorba.orb.Worker 类 的 getTransport（）方法。

从 ORB 的 getDefaultClientWorker 就可以取得缺省的 IOP 引擎的客户对象。ObjRefDelegate 使用它来创建请求，与其它的 CORBA server 通讯。

3.3.5 pluggable protocols framework 对象说明

1．Channel 类

火龙果•整理
uml.org.cn

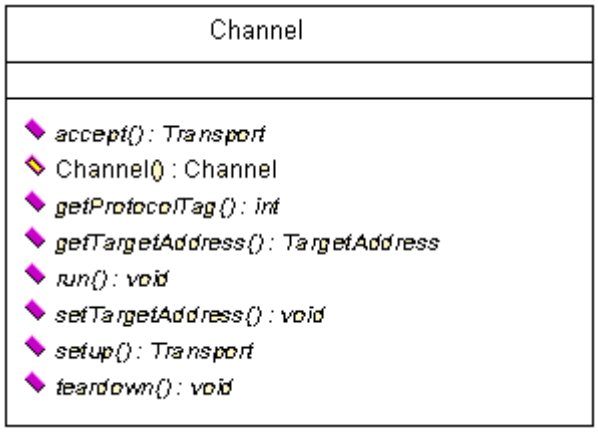


图 3.9　Channel 类

Channel 类是一个抽象类，它代表了一条可能的网络连接（或半连接，即具有连接能力的客户端或等待连接的服务器端），它的主要方法有：

1）SetTargetAddress（）：设定 Channel 类的 target 成员。

2）Setup（）：它向指定的 TargetAddress 发起一个连接，并返回具有传送能力的 Transport 类型的对象。

3）Accept（）：在某端口监听网络上的连接请求。这是一个阻塞的方法，它直到有了连接请求并成功的建立了一个连接之后才会返回。

4）public abstract void teardown()：销毁该连接；

2．TargetAddress 类

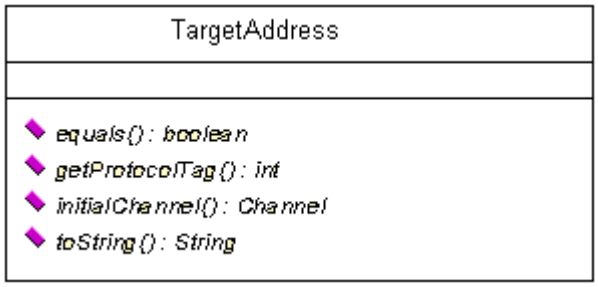


图 3.10　TargetAddress 类

它标识一个目标端点。它也是一个抽象类。它的主要方法有：

1）equal（）：比较两个 TargetAddress 类型的对象是否指向的同一个服务器端点。

2）InitialChannel（）：向本对象代表的目标端点发起连接，并返回 Transport 类型的对象。

3．ProfileExtractor 类

火龙果•整理
uml.org.cn

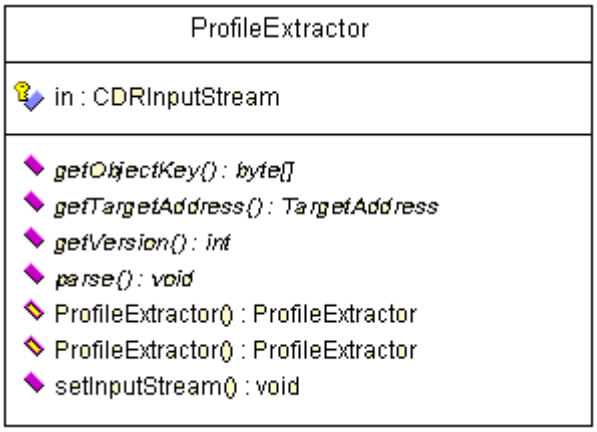


图 3.11　ProfileExtractor 类

ProfileExtractor 类似一个抽象类。它可以分析 org.omg.IOP.TaggedProfile 对象，并返回需要的值。与协议相关的具体操作需要继承并实现这个抽象类。

1）in：org.com.mincorba.orb.cdr.CDRInputStream，ProfileExtractor 从这个输入流中读取数据并分析出其中的结构。

2）public void setInputStream(CDRInputStream in)：设置输入流；

3）public abstract void parse()：进行具体分析的动作；

4）public abstract TargetAddress getTargetAddress()：当对 Profile 的分析完成以后，调用此操作可以返回该 Profile 标识的 TargetAddress 对象，协议相关的实现必须重载该方法；

5）public abstract byte[] getObjectKey()：返回 Profile 中的 objectkey；

4．ProfileCreator 类

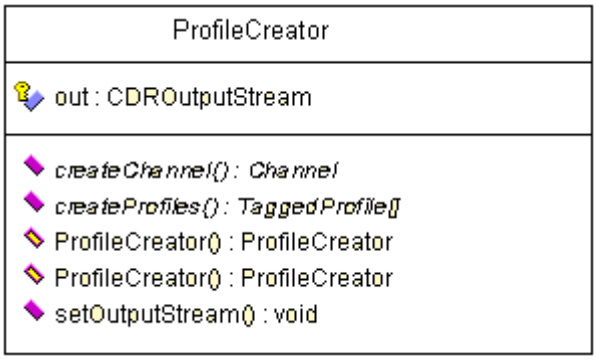


图 3.12　ProfileCreator 类

ProfileCreator 是一个抽象类。它用于创建协议相关的 org.omg.IOP.TaggedProfile 对象。成员：

1）CDROutputStream　out：Profile 的输出流，最后生成的 Profile 就放在 out 中；

火龙果•整理
uml.org.cn

2）public void setOutputStream(CDROutputStream out)：为 ProfileCreator 设置输出流；

5．ProfileRegistry 类

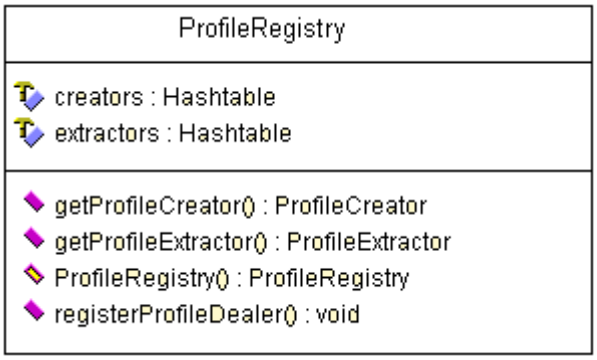


图 3.13 ProfileRegistry 类

这是所有的 ProfileCreator 和 ProfileExtractor 的协议相关的实现对象的注册表，所有协议相关的 ProfileCreator 和 ProfileExtractor 对象都必须在 ProfileRegistry 对象中注册。ProfileRegistry 对象可以根据协议的枚举值返回相应的 ProfileCreator 和 ProfileExtractor 对象。

1）public void registerProfileDealer(int tag, ProfileCreator creator, ProfileExtractor extractor)：注册协议相关的 ProfileCreator 和 ProfileExtractor 对象，tag 表明了它们能处理的协议类型；

2）public ProfileExtractor getProfileExtractor(int tag)：根据协议类型取得具有处理该类型 Profile 能力的 ProfileExtractor 对象；

3）public ProfileCreator getProfileCreator (int tag)：根据协议类型取得具有处理该类型 Profile 能力的 ProfileCreator 对象；

6．Transport 类

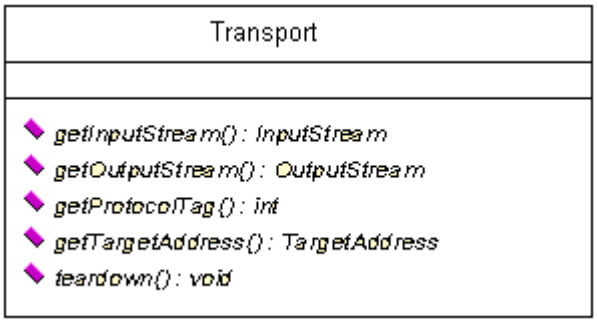


图 3.14 Transport 类

具有传送能力，它可以将指定的字节流传送到网络的另一端。它的传送能力依赖于不同网络协议的具体实现。

火龙果•整理
uml.org.cn

1）public abstract TargetAddress getTargetAddress()：取得 Transport 的另一端的 TargetAddress 对象；
2）public abstract java.io.InputStream getInputStream()：取得输入流对象，用于从网络读取字节流；
3）public abstract java.io.OutputStream getOutputStream()：取得输出流对象，用于向网络输出字节流；
4）public abstract void teardown(boolean wait)：销毁该对象表示的连接；
以上的几个方法都是抽象方法，当映射到具体的传输协议时，这些方法都需要实现。

7．TransportsPool 类

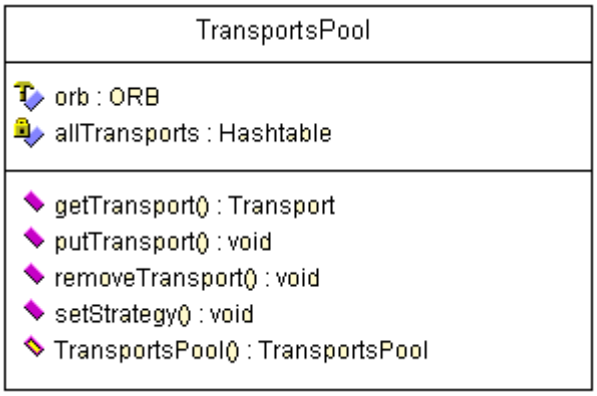


图 3.15　TransportsPool 类

TransportsPool 类根据不同的 TargetAddress 索引了所有的 Transport 对象。它具有一定程度上的连接管理功能。
1）public void setStrategy(int stra)　设置策略；
2）public Transport getTransport(TargetAddress a)　取得 Transport 对象；
3）public void putTransport(Transport t)　向索引中加入 Transport 对象；
4）public void removeTransport(Transport t)　从索引中删除 Transport 对象

8．ProtocolInitializer 类

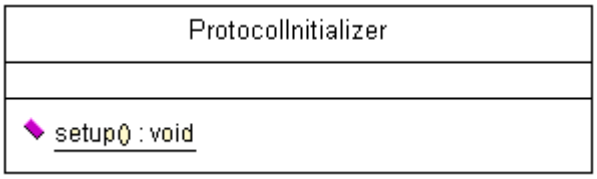


图 3.16　ProtocolInitializer 类

如果要使用某个网络协议作为底层传输协议，它必须继承 ProtocolInitializer 并重载它的 setup 方法，setup 方法是 ProtocolInitializer 唯一

火龙果•整理
uml.org.cn

的方法，使得每个协议实现有机会做一些准备工作（主要是注册一些对象，比如 ProfileCreator 和 ProfileExtractor 的协议相关实现对象）。

9．Listener 类

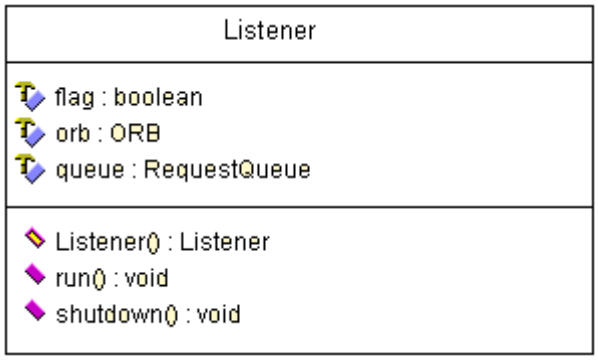


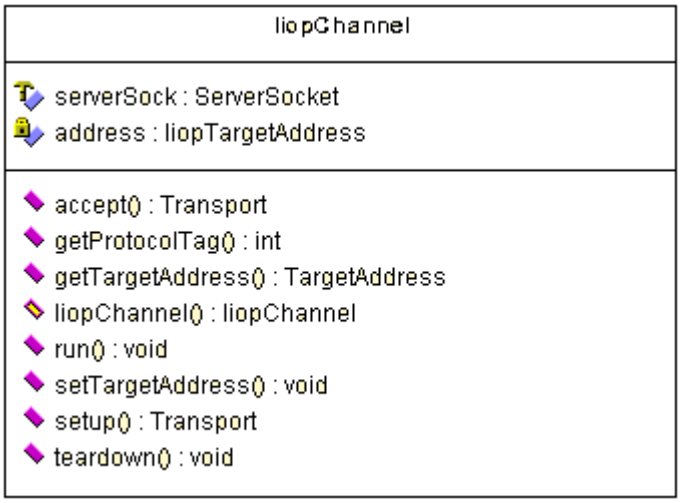
图 3.17　Listener 类

Listener 对象在 ORB 中一般只有一个，它负责监听网络请求，它是一个单独的线程；如果监听到了请求，Listener 对象会接受连接，并建立一个 ServerWorker 对象，ServerWorker 对象也是一个线程，用来处理请求。

3.3.6 IIOP 模块说明

IIOP 模块就是 mincorba 使用的缺省传输协议，它也是一个 pluggable protocols framework 到 tcp 传输协议映射实现的一个例子。此模块中的大多数类都与对应 pluggable protocols framework 中的类，下面只列出它们的结构，具体方法的功能类似于 pluggable protocols framework 中的类。

IiopChannel 类



火龙果•整理
uml.org.cn

图 3.18 IiopChannel 类

IiopProfileCreator 类

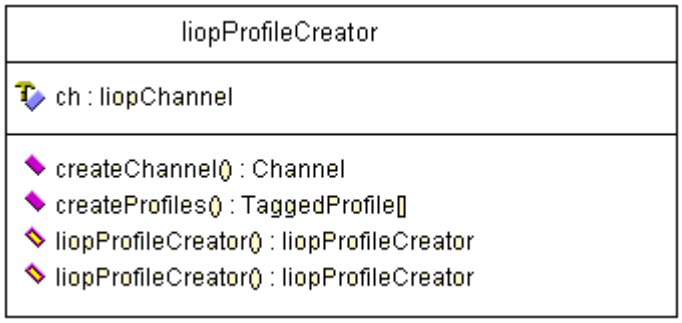


图 3.19 IiopProfileCreator 类

IiopProfileExtractor 类

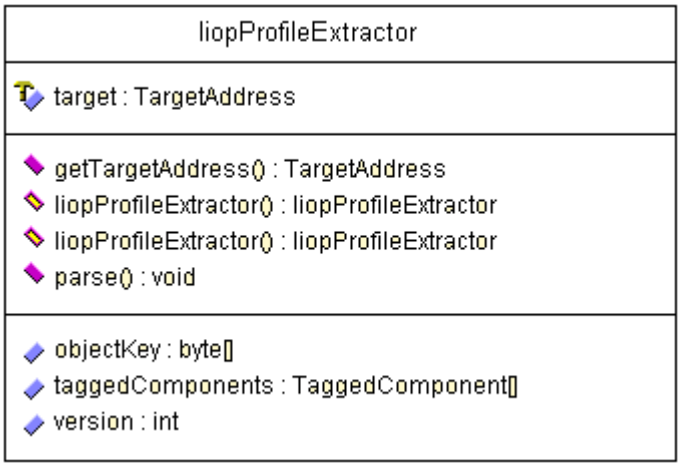


图 3.20 IiopProfileExtractor 类

IiopProtocolInitializer 类

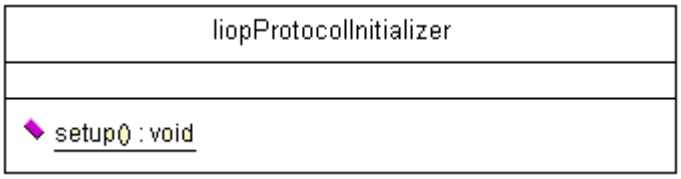


图 3.21 IiopProtocolInitializer 类

IiopTargetAddress 类

火龙果•整理
uml.org.cn

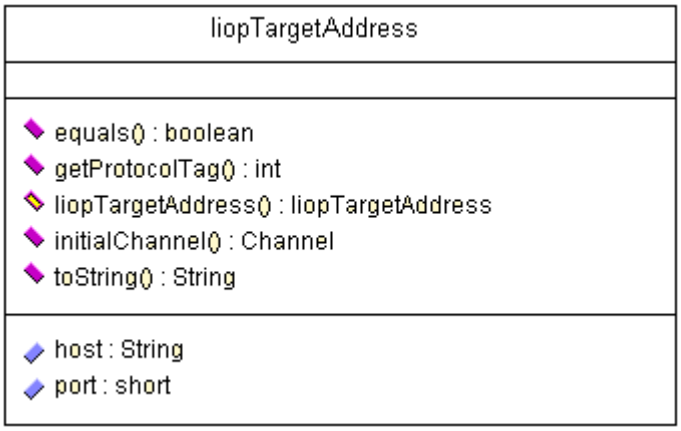


图 3.22　IiopTargetAddress 类

IiopTransport 类

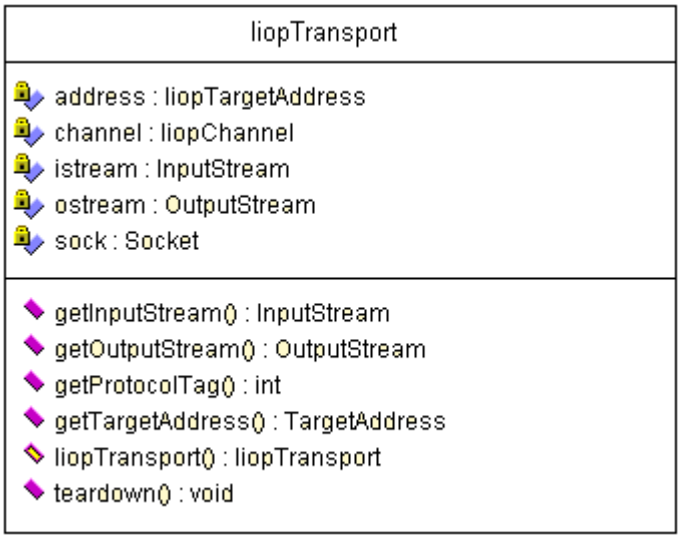


图 3.23　IiopTransport 类

火龙果•整理
uml.org.cn

3.3.7 IorParser 类

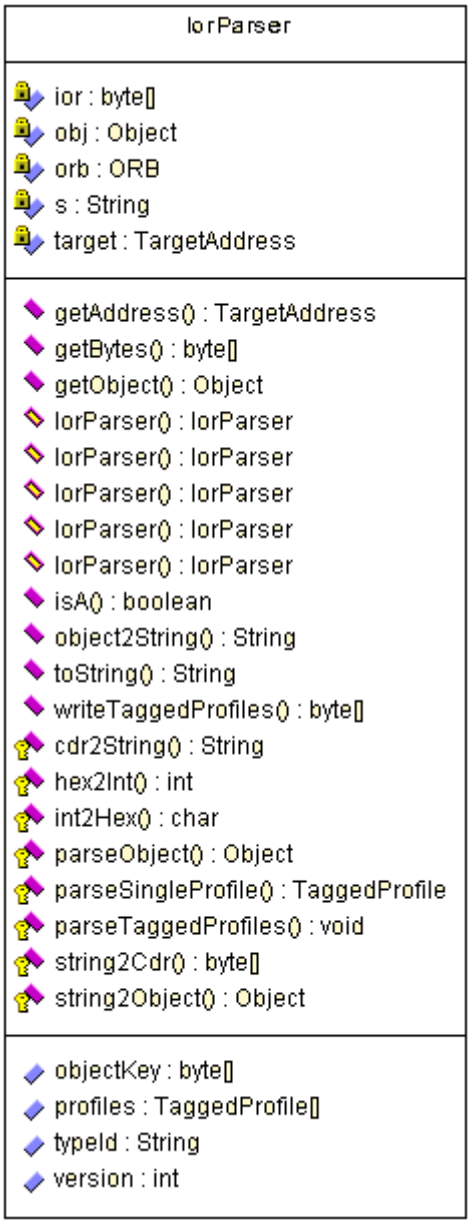


图 3.24 IorParser 类

IorParser 可以形成和分析 ior。

火龙果•整理
uml.org.cn

3.3.8 线程模型

ORB 的线程模型可以分为每客户一个线程和线程池模型。在每客户一个线程模型中，每当有用户连接，ORB 都产生一个新的 ServerWorker 线程对象，来处理它对应的客户的请求；在线程池模型中，每当有客户发起连接时，ORB 接受这个连接，它会查询在 ServerWorkerRegistry 中是否有可用的 ServerWorker 对象，如果没有，客户需要等待，当 ServerWorkerRegistry 中有了可用的 ServerWorker 对象以后，客户的请求才得以处理。本文采用前者。

3.3.9 ORB 与 POA 的接口

Objectkey 用于标识一个分布对象；它实际上标识了实现（incarnate）这个分布对象的对象实现在进程中的位置信息。它由一组 poa 的名字和一个 objectId 组成。根据 objectkey 中所包含的 poa 的名字序列，可以从 rootPOA 找到管理对象实现的 poa；而该 poa 根据 objectid 从 ActiveObjectMap 中能找到 objectkey 标识的对象实现的引用。

当请求从当前工作的 ServerWorker 对象流到 invokeEngine 对象时，invokeEngine 从请求流中得到 objectkey，然后调用 org.com.mincorba.poa.ObjectKeyExtractor 对象对 objectkey 进行解码，得到 poa 的名字序列和 objectId；然后根据 poa 的名字序列找到我们所需要的 poa，在使用 objectId 从 poa 返回 org.omg.PortableServer.Servant 对象，然后构造 InvokeHandler 的_invoke 方法的参数并调用之；得到返回值后，把它们发送到网络中。

org.com.mincorba.poa.POA_impl 类继承 mincorbaPoa 接口。本接口提供了一个 invoke 方法，mincorbaPoa 是实现 pluggable poa 的基础。原型如下：

```
public org.omg.CORBA.portable.OutputStream
invoke(org.omg.PortableServer.ObjectId oid,
                    String method,
                    org.omg.CORBA.portable.InputStream is,
                    ResponseHandler handler)
          throws org.omg.CORBA.OBJECT_NOT_EXIST;
```

ORB 提供给 POA 的接口：ORB 提供给 POA 的接口有 createIor 方法：createIor 方法用于 POA 创建对象引用。它的原型如下：

```
public org.omg.CORBA..portable.InputStream createIor(byte[] objectKey, String
typeId);
```

POA 首先自己生成 objectkey，然后调用 createIor 得到一个 CDRInputStream 流；然后使用如下过程生成对象引用：

```
IorParser parser = new IorParser(orb, inputStream);
```

火龙果•整理
uml.org.cn

Org.omg.CORBA.Object obj = parser.getObject();

3.3.10 小结

上面按模块说明了 mincorba 的大致结构，下一章将描述最小 CORBA 的动态执行情况，期望让读者对此实现有更深的了解。

火龙果•整理
uml.org.cn

第四章　mincorba 算法简介

4.1 示例程序

本章打算以例子程序执行过程来阐明 mincorba 的主要算法。你可以通过本章了解 CORBA 程序的编写方式以及 mincorba 的主要算法。

首先要安装 JDK1.4.0,因为 mincorba 是以 JDK 程序包的形式使用的；mincorba 要装在%JAVAHOME%\lib\下（或装在其它目录下，加到 classpath 中）。

本章的示例程序是一个分布式的 hello 程序。分布式的 hello 与以前学习编程语言的 hello 程序都不同，它是一个分布式的 hello 程序，它有一个客户端和一个服务器端。客户端将一个字符串传送到服务器端，服务器端将显示这个字符串，然后服务器端自己生成一个字符串，将它传送回客户端，最后客户端显示这个字符串。

我们首先要定义接口 hello.idl：

```
module demo {
module hello {
        exception HelloException { };
        interface HelloObj {
                string getHelloString(in string helloString)
                        raises(HelloException);
        };
};
};
```

然后使用 JDK 自带的 idlj 编译该文件：

idlj -fall hello.idl

这时 idlj 会在 demo\hello\hello 目录下生成一些文件，它们是：

HelloException.java
HelloExceptionHelper.java
HelloExceptionHolder.java
HelloObj.java
HelloObjHelper.java
HelloObjHolder.java
HelloObjOperations.java
HelloObjPOA.java
_HelloObjStub.java

1）然后编写客户端 Client.java，它的代码如下：

火龙果•整理
uml.org.cn


```
package demo.hello;
import java.io.*;
import org.omg.CORBA.*;

public class Client {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.out.println("Usage: demo.hello.Client <ior_file>");
            System.exit(1);
        }
        try {
            String fileName = new String(args[0]);
            java.util.Properties props = new java.util.Properties();
            props.setProperty("org.omg.CORBA.ORBClass",
                              "org.com.mincorba.orb.ORB");
            props.setProperty("org.omg.CORBA.ORBSingletonClass",
                              "org.com.mincorba.orb.ORBSingleton");
            ORB orb = ORB.init(args, props);
            FileInputStream f = new FileInputStream(fileName);
            byte[] buf = new byte[f.available()];
            f.read(buf, 0, f.available());
            String ior = new String(buf);
            org.omg.CORBA.Object obj = orb.string_to_object(ior);
            HelloObj helloObject = HelloObjHelper.narrow(obj);
            for(;;) {
                System.out.println(helloObject.getHelloString("say hello"));
                }
          }
        catch (Exception ex) {
            System.err.println(ex);
            ex.printStackTrace();
        }
    }
}
```

这样客户端就完成了。客户端由下列文件构成：

Client.java
_HelloObjStub.java
HelloException.java

火龙果•整理
uml.org.cn

HelloHolder.java
HelloHelper.java
HelloExceptionHoler.java
HelloExceptionHelper.java
HelloObjOperations.java

2）然后是服务器端，服务器端分两部分，一部分是服务器端的主程序，一部分是服务对象的实现。

主程序 Server.java 的代码如下，它加载服务对象的实现，并激活它：

```
package demo.hello;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class Server {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java demo.hello.Server <ior_file>");
            System.exit(1);
        }
        try {
            java.util.Properties props = new java.util.Properties();
            props.setProperty("org.omg.CORBA.ORBClass",
                              "org.com.mincorba.orb.ORB");
            props.setProperty("org.omg.CORBA.ORBSingletonClass",
                              "org.com.mincorba.orb.ORBSingleton");
            ORB orb = ORB.init(args, props);
            POA poa =
              POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            poa.the_POAManager().activate();
            HelloObjImpl impl = new HelloObjImpl(poa);
            org.omg.CORBA.Object obj = poa.servant_to_reference(impl);
            PrintWriter pw = new PrintWriter(new FileWriter(args[0]));
            pw.println(orb.object_to_string(obj));
            pw.flush();
            pw.close();
            orb.run();
        }
        catch (Exception e) {
            System.out.println(e);
```

火龙果•整理
uml.org.cn


```
                e.printStackTrace();
            }
        }
}
```

接下来是服务对象的实现 HelloObjImpl.java，它打印传入的字符串，返回一个字符串“hello”：

```
package demo.hello;
public class HelloObjImpl extends HelloObjPOA {
    public HelloObjImpl(org.omg.PortableServer.POA poa) {
    }
    public String getHelloString(String helloString) {
        System.out.println(helloString);
        return "hello";
    }
}
```

整个服务器端的程序有如下的文件组成：

HelloException.java
HelloExceptionHelper.java
HelloExceptionHolder.java
HelloObj.java
HelloObjHelper.java
HelloObjHolder.java
HelloObjImpl.java
HelloObjOperations.java
HelloObjPOA.java
Server.java

3）编译后即可运行，此程序可以在两台不同操作系统机器间执行。

首先在 demo 的父目录中运行服务器端：

```
java demo.hello.Server hello.ref
```

然后把在当前目录生成的 hello.ref 文件拷贝到 Client 的运行目录中，

```
java demo.hello.Client hello.ref
```

如果 Client 和 Server 放在相同的目录中，就不需要拷贝 hello.ref 文件了。运行结果是:

客户端不断向服务器发字符串“say hello” ，服务器端显示；

服务器端返回 hello,客户端显示。

火龙果•整理
uml.org.cn

4.2 程序执行过程分析

下面就以此程序的执行过程为脉络，介绍一下 mincorba 的一些主要算法。

4.2.1 服务器端的程序 Server.java

1 .ORB orb = ORB.init(args, props)

Server.java 中首先是 orb 初始化。

ORB orb = ORB.init(args, props);

ORB 类在 org.omg.CORBA 包的 ORB.java 文件中

```
public static ORB init(String[] args, java.util.Properties props)
{
        org.omg.CORBA.ORB orb = initOrb();
        orb.set_parameters( args, props );
        return orb;
}
```

a. 上面 org.omg.CORBA.ORB orb = initOrb()执行情况

```
private static ORB initOrb()
{ …
    return (ORB) Class.forName(className).newInstance();
  …
}
```

我改动了 JDK 中 ORB.java 的 defaultORB:

private static final String defaultORB = "org.com.mincorba.orb.ORB";

于是最终返回的是 mincorba 的 orb.这里利用了 Java 的反射机制。

b. orb.set_parameters(args, props) 执行情况

实际调用的是

```
public void initialize()
{
        profileDealers = new ProfileRegistry();
        pool = new TransportsPool(this);
        org.com.mincorba.orb.protocols.iiop.
            IiopProtocolInitializer.setup(this);
        invoker = new IopEngine(this);
        queue = new RequestQueue(this);
        handler = new ServerWorkerRegistry(this);
        listener = new Listener(this);
```

火龙果•整理
uml.org.cn


```
        poaTable = new java.util.Hashtable();
        rootPoa = null;
}
```

上述代码初始化一些对象。

1) ProfileRegistry 构造函数中建立 Hash 表，用来存储 Profile 生成器 ProfileCreator 与解析器 ProfileExtrator;
2) TransportsPool 构造函数中建立 allTransport(Hash 表), 用来存储 transport 对象;
3) IiopProtocolInitializer.setup(),向 ProfileRegistry 中存储 iiopPrifileCreator 和 iiopProfileExtractor，前者构造函数中又建立 iiopChannel，iiopChannel 构造函数建立 ServerSocket；
4) 后面几句建立的几个对象构造函数几乎是空操作。

2. Server.java 下一句建立 POA

orb.resolve_initial_references("RootPOA")

在 JDK 的 ORB.java 中：

```
{   …
        if (object_name.equals("RootPOA")) {
                return (org.omg.CORBA.Object)getRootPOA();
   …
}
```

调用 getRootPOA()

```
{…
    if( rootPoa == null ) {
            rootPoa=
             (mincorbaPoa)(Class.forName("org.com.mincorba.poa.
                                        POA_impl").newInstance())
                        }
…
}
```

新建一个 POA_impl 对象,也就是我们自己写的 POA 的类,它实现了 OMG 的 POA 接口。值得注意的是 POA_impl 对象的构造函数：

POA_impl.java 文件中：

```
public POA_impl()
{//create default policies
                …
    this.parent = this;
    this.name = "RootPOA";
    this.childs = new java.util.Hashtable();
```

火龙果•整理
uml.org.cn


```
        this.activeObjectMap = new ActiveObjectMap();
        if( manager != null) {
                                this.manager = manager;
                                }
        else {
                                this.manager = new POAManager_impl();
                                }
            this.id = UniqueIdGenerator.generateId().getBytes();
    }
```

它完成的功能有：

1)创建 POA 的缺省策略；
2)形成活动对象表，以便管理对象；
3)如果没有 POA 管理器，就生成一个；
4)为 POA 编号。

3. 激活 POAManager

```
poa.the_POAManager().activate()
```

4.下面重点分析一下服务端的代理情况

Server.java 中再向下一句是建立 HelloObjImpl 对象：

```
HelloObjImpl impl = new HelloObjImpl(poa);
```

再向下一句是：

```
org.omg.CORBA.Object obj = poa.servant_to_reference(impl);
```

这一句执行的是 POA_impl.java 文件中的方法：

```
public org.omg.CORBA.Object
servant_to_reference(org.omg.PortableServer.Servant p_servant)
{
    synchronized(this) { …
        String poaId = new String(id);
        String oid = UniqueIdGenerator.generateId();
        String stringKey = poaId + "/";
        stringKey = stringKey + oid;
        String[] ids = p_servant._all_interfaces(this, null);
        IorParser parser =orb.createIor(stringKey.getBytes(), ids[0]);
        org.omg.CORBA.Object obj = parser.getObject();
        …
        Object ret = activeObjectMap.put(oid, p_servant);
        return obj;
    }
}
```

火龙果•整理
uml.org.cn

1) 首先产生对象的 id,再加上 poaid 形成 stringKey；
2) 取得传入 sevant 的 ids,此例中为"IDL:demo/hello/HelloObj:1.0"
3) 然后利用上述两个参数调用 orb.createIor，如下：

```
public IorParser createIor(byte[] objectKey, String typeId)
 {
    org.omg.IOP.TaggedProfile[] profiles =
            profileDealers.getProfileCreator(getDefautTransportProtocol()).cr
            eateProfiles(objectKey,
            getDefaultGiopVersion());
     IorParser parser = new IorParser(this, profiles, typeId);
     return parser;
}
```

上述代码执行的过程是：

a.形成 profiles

profileDealers.getProfileCreator 取得一个 iiopProfileCreator 对象后，再调用 iiopProfileCreator 对象的 createProfiles,最终 host+port+objectkey(poaid+oid)+version+…共同构成了 profile.

b. 创建 Iorparser

orb.createIor.java 中下一行是：IorParser parser = new IorParser(this, profiles, typeId)

执行的代码很简单：

```
public IorParser(org.com.mincorba.orb.ORB orb, TaggedProfile[] profiles,
String typeId) {
            this.orb = orb;
            this.profiles = profiles;
            this.typeId = typeId;
 }
```

4）让我们回到所讨论的 POA_impl.java 文件：

org.omg.CORBA.Object obj = parser.getObject();

此函数执行情况如下：

```
 {
  ObjRefDelegate delegate = new ObjRefDelegate(this.orb, this);
  ObjectImpl objectImpl = new ObjectImpl();
  objectImpl._set_delegate(delegate);
  return objectImpl;
  }
```

于是生成了下图所示的关系：obj 即

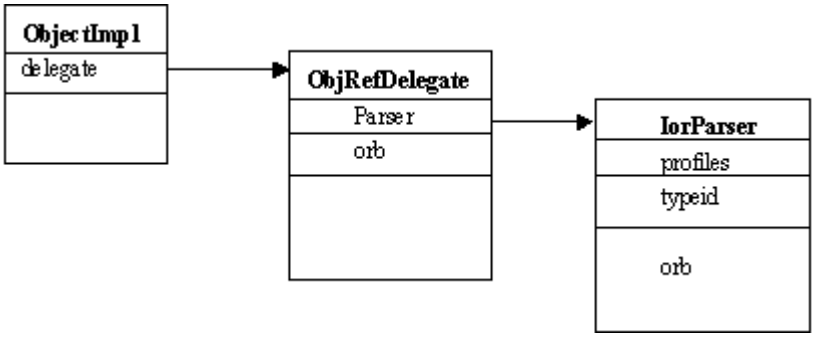


图 4.1　servant_to_reference 结果图

5)activeObjectMap.put(oid, p_servant) 把此对象实现放入到活动对象表中。

5. Server.java 下一句形成 IOR 字符串 orb.object_to_string(obj)

执行情况如下：

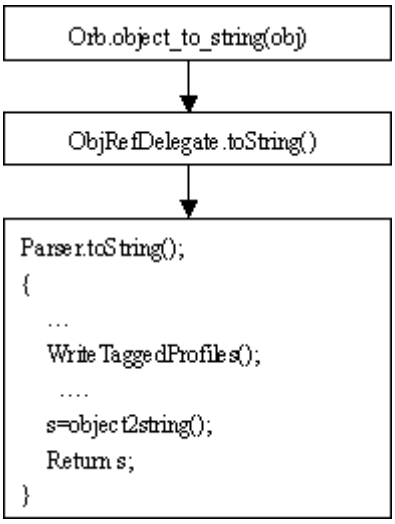


图 4.2　object_to_string 执行调用图

WriteTaggedProfiles()创建 ior

Byte[] ior=false+typeid+profile.length()+profile;

Object2string()根据 ior 生成一个十六进制字符串并写入文件 hello.ref;

6．Orb 开始监听 Orb.run();

```
{
  java.lang.Thread handlerThread = new java.lang.Thread(handler);
  handlerThread.start();
  listener.run();
}
```

开两个线程，listener 和 ServerWorkerRegistry：

listener 线程利用 ProfileCreator 创建 Channel,接收消息，返回 Transport；

handlerThread.start()调用 ServerWorkerRegistry 的 run(),执行情况如下：

火龙果•整理
uml.org.cn

{
从监听队列中取一个 transport；
新建 GiopServerworker 对象；
调用 GiopServerworker 对象的 run 方法；
}
GiopServerworker 的 run():分析消息头的格式，并分类处理。

4.2.2 客户端的程序 Client.java

1．客户端 orb 的初始化与服务端基本相同；此后读入 ior 字符串；
2．下一句是 org.omg.CORBA.Object obj = orb.string_to_object(ior)

```
public org.omg.CORBA.Object string_to_object(String str)
  {
      IorParser ior = new IorParser(this, str);
      return ior.getObject();
  }
```

IorParser 的构造函数如下：

```
public IorParser(org.com.mincorba.orb.ORB orb, String s)
{
      this.orb = orb;
      this.s = s;
      this.obj = string2Object(s);
}
```

string2Object(s)又调用 parseTaggedProfiles()，根据 ior 生成 profiles，typeid 等；而后又新建 objectImpl 与 objRefDelegate 对象，形成下图：

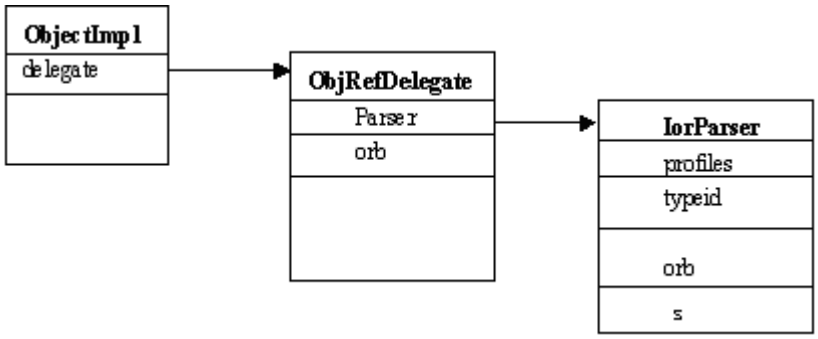


图 4.3　string_to_object 结果图

于是 ior.getObject()最终返回的 obj 与服务端的基本相同。

3. **Client.java 中下一句** HelloObj helloObject =
HelloObjHelper.narrow(obj)

火龙果•整理
uml.org.cn

执行的代码是：

```
{   org.omg.CORBA.portable.Delegate delegate =
                    ((org.omg.CORBA.portable.ObjectImpl)
                      obj)._get_delegate();
    demo.hello._HelloObjStub stub = new demo.hello._HelloObjStub();
    stub._set_delegate(delegate);
    return stub;
}
```

即新建 demo.hello._HelloObjStub，再设置其 delegate 为上图的 objRefDelegate; 形成下图：

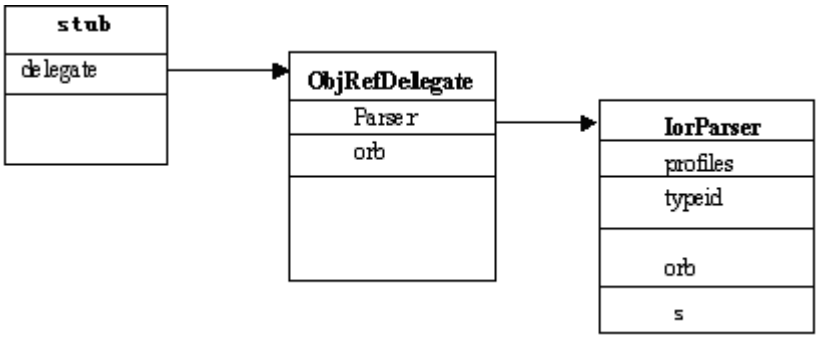


图 4.4　客户桩结果图

4. Client.java 中下一句 helloObject.getHelloString("say hello")

执行代码如下：

```
_HelloObjstub.getHelloString("say hello")
{   org.omg.CORBA.portable.OutputStream _output = null;
    org.omg.CORBA.portable.InputStream _input = null;
    try {
        _output = this._request("getHelloString", true);
        _output.write_string(helloString);
        _input = this._invoke(_output);
        return _input.read_string();
      }
      ….
}
```

1) this._request("getHelloString", true)

_HelloObjstub 继承自 objectImpl,所以调用

```
…OutputStream    objectImpl._request(("getHelloString", true)
{
return _get_delegate().request(this, operation, responseExpected);
```

火龙果•整理
uml.org.cn

}

据 4.4 图，上述函数调用 objRefDelegate 的 request(this,operation,ResponseExpected)：

```
request(this,operation, responseExpected)
```

↓

```
request(this,operation, responseExpected,12)
{
org.omg.GIOP.TargetAddress targetObject= new
                    org.omg.GIOP.TargetAddress();
targetObject.object_key(parser.getObjectKey());
if( worker == null )
{
                    if( version == 0 ) {
                    version = orb.getDefaultGiopVersion();
                                        }
     worker=orb.getDefaultClientWorker(version);
}
          return
worker.createRequest(parser.getAddress(),targetObject,the_
operation,responseExpected);
```

图 4.5　stub.request 调用情况图

host+port+objectkey(poaid+oid)+version+…共同构成了 profile，此 profile 在建立 IorParser 时已经解析；上图中重要的几句是：

a.targetObject.object_key(parser.getObjectKey())将 Parser 的 objectkey 赋值给 TargetAddress;

b.而 orb.getDefaultClientWorker 执行情况见下图：

```
worker= orb.getDefaultClientWorker(version)
{return new ..giop.GiopClientWorker(this,
version);
}
```

↓

```
GiopClientWorker(this, version);
     {     requestId = 0;
           this.orb = orb;
           this.version = version;
     }
```

图 4.6　GiopClientWorker 的建立

火龙果•整理
uml.org.cn

c.worker.createRequest 执行情况见下图：

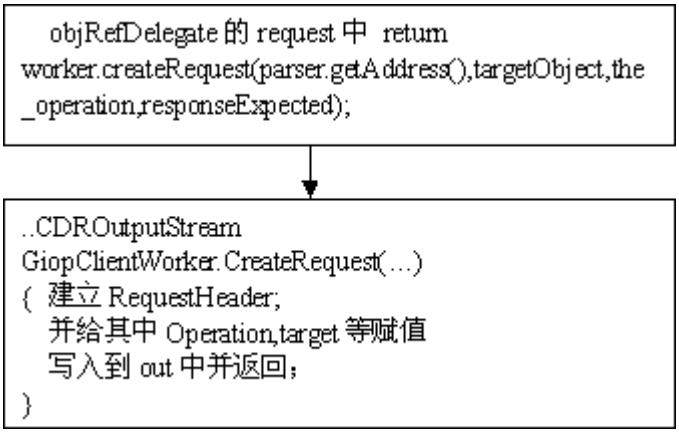


图 4.7　CreateRequest

至此形成了 out 流；

2）最终返回到_HelloObjStub 中；执行_output.write_string(helloString)后，随后调用

_input = this._invoke(_output)，执行情况如下：

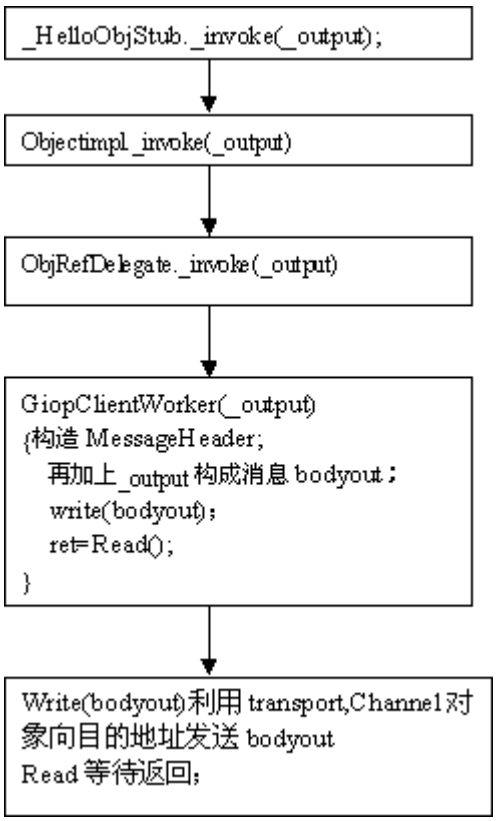


图 4.8　stub.invoke 调用情况图

火龙果•整理
uml.org.cn

把字符串发出并监听；
服务端 Listener 监听到后，交给 GiopServerWorker，GiopServerWorker 利用 IopEngine 找到 ORB 与 POA，再由 POA 找到合适的对象处理。

4.3 小结

综上：
1）客户端 _HelloObjStub 利用 GiopClientWorker 创建和发送请求；
2）服务端 Listener 线程读取请求，GiopServerWorker 线程负责处理请求，调用 IopEngine，IopEngine 又调用 ORB 和 POA 来找到合适的对象实现；
3）客户端与服务端通信利用 IIOP 协议；
4）这里对象引用 IOR 是通过 hello.ref 传递的。

火龙果•整理
uml.org.cn

第五章　总结与展望

本文详细介绍了 CORBA 规范，并介绍了 mincorba 的总体设计与具体算法。mincorba 系统实现的功能有：

1）借助于 JDK 的 idlj 编译器生成的客户桩和服务端框架，mincorba 能够实现客户端与服务器端调用，完成了 CORBA 平台的最基本的功能；
2）可以与其它的 CORBA 产品通信；
3）可以实现在不同平台不同操作系统间通信；
4）采用了较为灵活的程序结构，便于扩展以及改进；

当然，CORBA 平台的构建如同操作系统的构建一样，涉及到的问题很多，决不是一个人在短期内可以完成的，再加上本人知识面有限，对系统软件的设计缺乏经验，只能做一些尝试性工作，其完整的实现还需要进一步的努力。本人认为要实现完整的 CORBA 平台还有以下工作要做：

1）首先要完善 ORB 线程模型；
2）实现 IDL 编译器，完成 IDL 到多种语言的映射；
3）要加入动态调用：
　a 动态框架接口 Dynamic Skeleton Interface
　b 动态调用接口 Dynamic Invocation Interface
　c 动态 Any Dynamic Any
　d 接口库 Interface Repository
　因为虽然与静态调用冗余，毕竟动态调用更加灵活；
4）加入与动态调用相关的 POA 特性；
5)为了与其它分布式对象技术交互，需要加入 CORBA/COM 等桥接技术；
3）实现各种服务，如：
　a 对象命名服务（Naming Service）
　b 对象安全性服务（Security Service）
　c 并发控制服务（Cocurrency Control Service）
　d 对象生命期服务（LifeCycle Service）等

火龙果•整理
uml.org.cn

参考文献

[1] OMG. The Common Object Request Broker: Architecture and Specification. See: http://www.omg.org/corba

[2] OMG. The Unified Modeling Language Specfication, November 1999: See:http://www.omg.org/.

[3] OMG. C++Language mapping Specification .June1999

[4] OMG. IDL to Java Language Mapping Specification. June1999

[5] OMG. CORBA Service : Common Object Services Specification1998

[6] OMG. Naming Service Specification.April 2000

[7] OMG. Transaction Service Specification.May 2000

[8] OMG. Trading Service Specification.May 2000

[9] OMG. Event Service Specification.June 2000

[10] OMG. CORBA Messaging. Joint Revised Submission. Document orbos, 1998;(5):05

[11] OMG. Object by Value. Document orbos, 1998;(1):18

[12] OMG. CORBA Components. Document orbos, 1998;(10):18

[13] Robert Orfali, DamHarkey, Jeri Edwads ,Instanst CORBA; 1996

[14] Vinoski,S., New Features for CORBA 3.0, Communications of The ACM, 1998;41(10):44-52

[15] Krishnan Seetharman. The CORBA Connection. Communications of The ACM, 1998;41(10):34-43

[16] James Rumbaugh,Ivar Jacobson,Grady Booch: The Unified Modeling Language Reference Manual, Addison Wesley Longman,Inc. 1999

[17] Desmond Francis D'Souza, Alan Cameron Wills: Objects, Components, and Frameworks with UML:The Catalysis Approach, Addison Wesley Longman, Inc. 1998

[18] Inprise Corporation .Reference of VisiBroker For C++ .2000

[19] Inprise Corporation .Reference of VisiBroker For Java .2000

[20] Inprise Corporation .Programmer's Guide of VisiBroker For C++ .2000

[21] Inprise Corporation.Programmer's Guide of VisiBroker For Java .2000

火龙果•整理
uml.org.cn

[22] Jamie Jaworski 《Java 1.2 unleashed》 1998,Macmillan Computer publishing

[23] CORBA successful stories.Object Management Group ,see http://www.corba.org

[24] Douglas C. Schmidt and Steve Vinoski, Object Interconnections: Standard C++ and the OMG C++ Mapping,(CUJ Experts, January, 2001).

[25] Mohamed Fayad and Douglas C. Schmidt, Object-Oriented Application Frameworks.,(http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html).

[26] 朱其亮 郑斌 《CORBA 原理及应用》 北京邮电大学出版社，2001.9

[27] 汪芸 《CORBA 技术及其应用》，东南大学出版社 1999.4

[28] Joseph L. Weber 《Java2 编程详解》，电子工业出版社

[29] 苏 森 面向对象的互操作技术.电子科技大学博士论文，1998

[30] 苏 森 刘锦德.用于开放式分布处理的中间件的标准和规范，计算机应用，1998；18(9):5-8

[31] 刘锦德 苏森， CORBA 技术的新进展 <<计算机应用>> 第十九卷第五期

[32] [美] Michi Henning Steve Vinoski .《基于 C++ CORBA 高级编程》，清华大学出版社 2000.7

[33] Robert O Dan H 等.《智能 CORBA》，北京:电子工业出版社,1999

[34] 宋雨 史光莉 赵文清 CORBA 技术综述 ，电力情报 2001 年第一期

[35] 甘卫东，张开德 CORBA 实时性研究初探 计算机工程与应用 2001 年第一期

[36] 李卫华 傅晓东编著 《Java 语言基础及其应用》， 广西科学技术出版社 1998 年 7 月第 1 版

[37] Thomas 等编 熊伟华等译，《Java for Internet 编程技术》，北京：电子工业出版社，1997.11

[38] 张卫民等 《Java 语言及应用》，北京：清华大学出版社，1996.10

[39] 京京工作室译 《Java 编程思想》，机械工业出版社，1999

[40] 姜刚等著 《Java 语言程序设计》 1998，人民邮电出版社

火龙果•整理
uml.org.cn

[41] 王克宏（主编） 《Java2 程序设计》，清华大学出版社 2000 年 9 月

[42] Stephen R. Schach 著 袁兆山等译 《软件工程 Java 语言实现》,机械工业出版社，1999.9

[43] Erich Gamma 等著 李应军等译《设计模式》——可复用面向对象软件的基础,机械工业出版社, 2000.9

[44] Dirk Slama 等著 李师贤等译 《CORBA 企业解决方案》, 机械工业出版社, January, 2001

[45] 罗武军等 分布式面向对象系统设计模型框架研究 (第七界联合国际计算机会议论文集, October,2000).

[46] 周明天 汪文勇 《TCP/IP 网络原理与技术》 清华大学出版社

[47] 智少游 中间件、系统集成与 CORBA ，通信技术 1997.3

[48] 兑继英 沈卓炜 顾冠群 ,COM/CORBA 互操作的研究与实现 ,东南大学学报,1999.5

[49] 刘晖等 《用 CORBA 创建电子商务系统》，北京希望电子出版社 2002.12

[50] 汪芸 顾冠群 CORBA 规范及其实现， 东南大学学报 1997.3

[51] 谭晖，基于 CORBA 的分布式工作流执行服务的设计与实现，吉林大学硕士学位论文，2002.3

[52] 袁泉，工作流模型定义工具的设计与实现，吉林大学硕士生论文，吉林大学硕士学位论文，2002.3

火龙果•整理
uml.org.cn

致谢

本论文的研究工作是在导师苑森淼教授的严格要求和悉心指导下完成的，导师严谨的学风、求实的科学态度、创新的治学精神和诲人不倦的指导作风，尤其是刻苦专研的精神使我终生难忘。在此向老师致以最衷心的感谢！

特别感谢网友 mep 以及郑洪波同学，他们分别在 CORBA 和 Java 方面给了我很大的帮助。

感谢实验室各位朋友平时对我的帮助。

感谢所有关心、帮助过我的人们。