

酷壳

享受编程和技术所带来的快乐

深入浅出 CORBA

一本献给希望了解 CORBA 概念和术语人的简洁说明书

[Ciaran McHale](#)

翻译: [赵锟](#)

译序

从 wiki 上找到此书的链接，初读之下，感觉此文讲解的非常清楚，大量丰富的图例说明，于是就有了将此书翻译成中文的冲动。至于书名本应该是《简单地解释 CORBA 概念》，但是最后还是起了一个比较容易吸引眼球的标题。

本书原文行文非常流畅，用词也相当通俗易懂，建议有英文基础的同行直接阅读原文。

本书第二十三章的内容，涉及到安全的一些概念和术语，翻译也相对比较困难，我虽然给它翻完，但是最后还是不敢发布出来，因为此章需要对安全相关知识要有全面的理解，最后我将此章翻译的内容省略。

在翻译过程，有一些特殊的名词采用意译的方式，比如 IDL 的 `sequence` 类型，被翻译为可变数组，`Servant` 被翻译为服务提供者，而在 IOR 中的 `Contact Detail` 我统一翻译为联系细节等等，请读者在阅读时特别注意。

由于译者的水平有限，书中难免有翻译错误的地方，译者并不会对因为这错误造成的损失负责。

本书原版的版权归 [Ciaran McHale](#) 所有，此版中文版版权归译者和 [酷壳网](#) 所有，任何公司或个人可以任意自由转载，发布，部分发布，出版，部分出版本书，但必须保留如下的版权信息

Copyright © 2009 [赵锐](#)

Copyright © 2009 [酷壳](#)

感谢我的夫人在翻译过程中给我支持，没有她的鼓励，我无法完成此项工作。同时感谢我 9 个月大的小孩，你那可爱和天真无邪的微笑是我工作的动力。

感谢 [酷壳网](#) 的耗子在翻译过程中对我的指导。非常感谢你的帮助。

在阅读过程中，你什么意见和建议欢迎发邮件至 zhaokun.km@gmail.com 和我进行讨论。

有效性和版权

有效性

你可以从 <http://www.CiaranMcHale.com> 上免费获得本书，网站上的下载页 (www.CiaranMcHale.com/download) 提供本书如下形式的下载：

- A5 的 PDF 版本，比薄皮小说要略大一些。每页的尺寸小，并且有嵌入的超链接文本更适合于屏幕阅读。
- 2 页 PDF 文件 (没有任何超链接)，两个 A5 页并排的放置在一起。如果你使用这个以 A4 的纸张打印成书的话，这个版本将会为你节省纸张。如果你用美国信笺纸打印 (比 A4 纸张略短) 那么 Adobe Acrobat 的打印对话框有一个自动旋转和居中的选项，将使你获得更好的打印效果。
- HTML 的文档。如果你使用 windows 阅读工具你可以获得一个 .zip 格式的文件，当然如果你偏爱 unix 的话，你同样可以获得一个 .tar.gz 格式的文件。

如果你想在下载前在线的浏览本书你可以访问 www.CiaranMcHale.com/corba-explained-simply 连接。

版权

Copyright © 2007 Ciaran McHale.

官方地授权给任何人，可以免费，获得本书的复本来拷贝，出版，发布，颁布子许可证，出售本书，但是必须如下的几个前提下：

- 上述的版权信息和授权许可必须在所有本书的所有拷贝和部分拷贝中
- 虽然作者为此书准备的非常仔细，但是作者并不会做出任何承诺，而且不会对书中任何错误或遗漏承担责任。书中所有的观点仅仅是作者自己的观点。

本书这个版本于 2007 年 2 月 27 日写成。

商标

Orbix, Orbacus, INOA, INOA 的图标和 Making Software Work Together 都是 INOA Technologies PLC 或其子公司的注册商标。Java, J2EE, JavaBean 和 Write Once , Run AnyWhere 是 Sun Mircosystem 公司的注册商标。TAO 是华盛顿大学的注册商标。IBM , MQ Series, OS/390 和 AS/400 是国际通用计算机公司的注册商标。COM 和 .NET 是微软的注册商标。Open VMS 是惠普的注册商标。TIBCO Rendezvous 是 TIBCO 的注册商标。Amazon.com 是 Amazon.com 公司的注册商标。Windows 是微软的注册商标。Oracle 是 Oracle 公司的注册商标。Berkeley DB 是 Sleepycat 软件公司的注册商标。Linux 是 Linux Trovalds 的注册商标。Mac OS X 是苹果电脑公司的注册商标。CORBA 是 OMG 组织的注册商标。在美国或其他国家内，这里出现的所有商标都是它们拥有着的私有财产。

献给

我的母亲 Alice，和我的妻子 Bianca

同样也 JS 和 TUF

前言

预期读者

本书提供了 CORBA 相关概念和术语的细节介绍。本书的目标读者是那么具有技术背景并希望清晰了解 CORBA 概念，但不想了解所有底层细节的人。例如：

- 首席技术官(CTO)可能会希望阅读此书，用于了解 CORBA 的相关概念，并借此决定此项技术是否适合他或她的公司
- 项目经理可能会需要阅读此书，这样他或她可以组织他的开发者进行一场关于 CORBA 相关问题的重要讨论。
- 如果你的公司外包了部分 CORBA 应用程序开发给其他公司，那么当你和你外包公司讨论应用程序规范和需求的时候这本书将会对你有很大的帮助
- 刚开始学习 CORBA 的开发员，将会从 CORBA 高层概念的讨论中收益。对于这些概念的清晰了解，将使得他们更容易理解那些面向程序员文档中的底层 API。已经了解了一些 CORBA 基本能力的开发者，从此书中将会得到一个高级 CORBA 功能的概述。除此之外，本书还可以提供给那些具有一定 CORBA 开发经验的程序员关于如何增强他们技能的指导。
- 对于那些必须管理 CORBA 系统的管理员，如果他们阅读了此书将会知道 CORBA 是做什么的，怎么做的后将会发现他们的管理工作变得非常轻松。

本书故意忽略了代码例子，这是因为本书 CORBA 开发者的编程入门。那些对如何开发 CORBA 程序感兴趣的读者可以从 <http://www.amazon.com> 挑选一本被读者点评为优秀的书籍来阅读。在 [26.1 小节](#) 中罗列作者最喜爱的 CORBA 书籍列表。

虽然本书不是一本 CORBA 编程入门手册，但是它是编程入门手册的一个很好的补充。特别是，它简明扼要地介绍了 CORBA 概念，此书为以后读者阅读其他 CORBA 编程书籍提供了一个坚实的基础。

如何阅读本书

本书不需要从头阅读到结尾。本书的信息一章节来组织，每一章的内容都相对独立，而且如果本章概念依赖于其他章节的概念时，相关概念上还有跨章节的引用。这使得读者可以跳跃式的阅读本书，或只读他们感兴趣的章节。唯一的例外是所有的读者都必须阅读第一章，因为它解释了 CORBA 中最基础的概念。

关于作者

Ciaran McHale从Trinity College, Dublin, Ireland获得了计算机科学学士和博士学位。在过去的 11 年中他一直在INOA 技术公司工作。除了做客户的咨询顾问外，他的工作包括培训和开发工作。他居住在England的Reading和他的妻子，Bianca。你可以通过 Ciaran@CiaranMcHale.com和作者联系。

关于贡献的作者

Donal Arundel编写了本书中有关安全的章节([第 23 章](#))。Donal是INOA 技术公司的首席工程师，在这个公司中他是CORBA安全的技术领导，在过去几年中他一直从事分布式对象的安全解决方案的开发工作。之前他在ICL工作，在那里他开发一个安全的基于智能卡的Electronic Money System。他在Dublin City University, Dublin, Ireland获得了理学学士学位。

免责声明

虽然作者为此书准备的非常仔细，但是作者并不会做出任何承诺，而且不会对书中任何错误或遗漏承担责任。对于使用本书中信息造成相关损失不负任何责任。书中所有的观点仅仅是作者自己的观点

致谢

第一，感谢意大利 Vodafone 的 Marco Abbate ，因为你的支持，我开始编写本书。

第二，感谢 Donal Arundel 编写了本书的安全章节

第三，感谢无数的读者给此书的初稿许多反馈意见: Adrian Trenaman, Andres Ortiz, Arne Koschel, Brian Kelly, Francis Byrne, Jan Schaefer, John McHugh, Klaus Hofmann zur Linden, Neil Kenealy, Niall Donnelly, Örjan Petersson, Patrick Donnelly, Paul Taylor, Raffaele Giugliano, Rebecca Bergersen 和 Richard Bonneau

第四，感谢 INOA 中同行对本书给的各种帮助。Klaus Hofmann zur Linden 和 Enda Brennan ，感谢你们的鼓励。Sean Flavin ，感谢你的精神指引；Joe McCarthy, Fintan Bolton 和 John O'Sullivan 感谢你们提的各种个样的建议和帮助。

最后， 在 2001 年我和一个优秀的女人坠入爱河，在写书的过程中我向她提出了求婚的请求。Bianca 感谢你在我问你是否愿意嫁给我的时候回答了愿意，因此我人生发生了巨大的变化。

第一部分 CORBA 介绍

第1章、核心概念和术语

- [对象管理组织 \(OMG\)](#)
- [CORBA](#)
- [客户端和服务端](#)
- [接口定义语言 \(IDL\)](#)
- [交互对象引用 \(IOR\)](#)
- [CORBA服务](#)

1.1 对象管理组织(OMG)

对象管理组织 (OMG) 是一个非盈利的组织，它致力于提升面向对象的使用技术。它定义了CORBA规范和UML标准，OMG网站(www.omg.org)提供这些标准规范文档的PDF文件，并且允许对文档免费下载，OMG还有一些相关的管理工作，比如OMG网站维护和组织协调OMG成员开会等。

定制标准的工作是由 OMG 的成员们来执行的，现在 OMG 大约有 600 个成员。任何组织 (或个人) 只要对 OMG 的工作感兴趣都可以成为 OMG 的成员。这些成员包括大学学院，软件开发商，软件使用者。OMG 的成员可以自愿参与 OMG 小组定义新的 OMG 规范或者修改增强现有的 OMG 规范。通过这样的工作，OMG 规范可以朝其成员关心的方向进行发展。

1.2 CORBA

CORBA 是通用对象请求代理体系结构 (Common ORB Architecture) 的

简写。通用体系结构(common architecture)意味着其是一个技术标准，因此 CORBA 是一种被称为对象请求代理(ORB)的技术标准

ORB 是对象请求代理(Object Request Broker)的简称,它是一个被称为远程过程调用(RPC)的面向对象版本。ORB 或 RPC 是一种调用远程（或不同进程）过程或则远程（或不同进程）对象方法的机制，就程序而言远程调用和本地调用一样方便。

许多人认为 CORBA 是一种中间件或集成软件，这是因为 CORBA 经常是以独立的应用程序和其他程序进行交互。IONA 技术公司曾以一句话“使软件工作在一起”来总结和描述 CORBA 的目标。

当然，CORBA 不仅仅是现存的中间件技术。其他的一些技术包括 JAVA 的远程方法调用(RMI)，IBM 的 MQ series，微软的 COM 和 .NET 技术，SOAP，和 TIBCO Rendezvous，脚本语言——比如 UNIX 脚本，Perl，Python 和 Tcl 也可以被划归为中间件，因为脚本也经常被用于进行程序之间的通讯。一个典型的例子就是 UNIX 脚本的管道操作应用。我们以下面这个例子进行说明：

```
ls -l | grep ^d
```

管道操作发送第一个命令的输出到第二个命令。通过管道，两个应用程序可以进行通讯，而这正是中间件所做的。

CORBA 强大的地方就是它是一个分布式中间件，即，CORBA 允许应用程序间进行通讯，即便是：

- 在不同的机器上，例如跨网络
- 在不同的操作系统至上，CORBA 在许多操作系统上都有其产品，包括 Windows，UNIX，IBM 的主机和一些嵌入式操作系统。
- 在不同的 CPU 上，例如，Intel，SPARC，PowerPC，大编码(big-endian)，小编码(little-endian)，32 位机或则 64 位的 CPU 上。

- 不同的程序语言实现，比如说 C, C++, JAVA, Smalltalk, Ada, COBOL, PL/I, LISP, Python, 和 IDLScripts1

CORBA 同时也是一个面向对象的分布式中间件。这意味着客户端不是要调用服务端进程，而是调用一个对象（这个对象恰巧在服务端的进程中）

1.3 客户端和服务端

在一些计算机术语中，术语客户端和服务端有严格的定义，并且一个应用程序必为其中之一。在 CORBA 中这两个术语的定义并不是这么严格。在 CORBA 术语中，一个服务器进程是指包含了被请求对象的进程，一个客户端进程是指发起对象调用的进程。一个 CORBA 应用程序同时既可以是服务器端进程又可以是客户端进程。

1.4 接口定义语言(IDL)

一个 IDL 文件定义了服务端对象暴露的公共应用程序接口 (API)，CORBA 对象的类型被称为接口 (interface)，接口在概念上和 C++ 的类，JAVA 接口和相似。IDL 的接口支持多重继承。

[图 1.1](#) 是一个 IDL 文件的例子。一个 IDL 的接口 (interface) 含有方法和属性。很多人错误的认为属性 (attributes) 的概念和 C++ 实例变量，JAVA 中的域一样。这样认为是错误的。属性对应着一对 get-和 set-方法，他是 set-和 get-方法另外一种简单地语法表示。属性可以是只读的。在这种情况下，属性只对应 get 方法。

```
module Finance {  
    typedef sequence<string> StringSeq;  
    struct AccountDetails {
```

```

    string      name;
    StringSeq   address;
    long        account_number;
    double      current_balance;
};
exception insufficientFunds { };
interface Account {
    void deposit(in double amount);
    void withdraw(in double amount)
                    raises(insufficientFunds);
    readonly attribute AccountDetails details;
};
};

```

图 1.1: 一个 IDL 文件的例子

方法的参数有指定的方向，参数可以被指定为 `in` (意味着从客户端到服务器端)，`out` (意味着从服务器端到客户端) 或则 `inout` (意味着以两个方向进行传送)。方法也可以有返回值。如果方法出错，方法可以抛出异常，CORBA 共有 30 种预定义的异常类型，这些异常被称为系统异常(system exceptions)，虽然在实际情况中这些异常大都被 CORBA 的运行系统抛出，但是这些异常也可被应用方法抛出。除了系统异常外，应用还可以在 IDL 文件中定义自己的异常。这些异常被称为自定义(user-defined)异常。在方法定义上通过 `raise` 子句指定异常将会被抛出。

方法的参数或返回值可以是内建(built-in)类型例如: `string` , `boolean` 或则 `long` 型。也可以是 IDL 文件中的用户定义类型。用户定义类型可以是如下的几种类型:

- 结构(struct)类型。结构类型于 C/C++ 的结构、JAVA 中仅含公共域的

Class 相类似

- 可变数组(sequence)类型。可变数组是一个集合(collection)类型, 它更像一个可以伸缩的一维数组。
- 数组(array)类型。IDL 数组的维数在 IDL 文件中指定。因此数组类型具有固定长度, 它不能在运行期间伸缩。数组类型很少在 IDL 文件中被使用。大多情况下都是使用更灵活的可变数组(sequence)类型。
- 类型定义(typedef)。类型定义允许为存在的类型定义别名。例如: 如下语句定义了一个 age 类型代表了系统原有的 short 类型。

```
typedef short age;
```

在一般情况下, IDL 的可变数组(sequence)类型或数组类型都是匿名(anonymous)类型, 这是因为可变数组类型和数组类型都没有名字。一种通常而重要的 typedef 使用方法就是在定义可变数组和数组类型的时候给其起一个名字。在图 1.1 中可以看到 StringSeq 就是这样使用的。

- 联合(union)类型。一个联合类型可以运行时保存一个或多个值, 例如:

```
union Foo switch(short) {  
    case 1: boolean  boolVal;  
    case 2: long      longVal;  
    case 3: string    stringVal;  
};
```

一个 Foo 的实例可以保存布尔, 长整形, 或字符串类型, case 标签()指示了当前为活跃的是什么值。和 union 相似的许多结构可以在许多过程语言中找到。不过由于面向对象的多态可以更好完成 IDL 联合类型的功能, 所以联合类型在面向对象语言中很少被使用。

- 枚举(enum)类型。枚举类型从概念上来说很像一个常量声明集合。例如:

```
enum color { red, green, blue };  
enum city { Dublin, London, Paris, Rome };
```

在内部实现上，CORBA 经常使用 integer 来代表不同的枚举类型值。使用枚举声明的好处是，现行的编程语言对枚举有支持，并且通过枚举可以实行强类型检查。比如说当我们把一个 color 加入到 city 中，

- 固定(fixed)类型。不动点(fixed-point)数字值，精度和双精度类型都是存储浮点类型值。浮点类型在很多情况下都适用，但是在少数情况下，浮点类型却会发生一些随机错误。相比而言，不动点类型(fixed-point)保存同样一个浮点类型值需要更多内存空间，但是不动点类型却具有不会发生随机错误的好处。在财务计算和数字信号处理领域，不动点类型被广泛的使用。但是在应用程序应用，不动点类型运算更多被使用为做为实现的细节而不被暴露在 IDL 文件中。由于这些原因，不动点类型很少在 IDL 文件中声明。
- 值(valueType)类型。value的详细讨论在 [9.2 小节](#)中

IDL 的定义的类型可以以模组(Module)的形式被组织起来。这种组织方式和 C++的 namespace，JAVA 中的包类似，模组通过放置一个前缀在类型名字前来解决名字冲突问题。在 IDL 中范围操作符是“::” 例如：
Finance::Account 是定义在 Finance 模组中 Account 类型的全称

1.4.1、 C++预处理器

IDL编译器使用类似于C++预处理器来预处理IDL文件，预处理器将删除idl文件中C++风格的注释代码，同时对预处理指令进行处理。[图 1.2](#)显示了一些预处理指令的例子。

```
#ifndef FOO_IDL  
#define FOO_IDL
```



```
#include "another-file.idl"
#pragma prefix "acme.com"
// This is a one-line comment
/* This is a multi-line
   comment
*/
module Foo {
    ...
};
#endif
```

图 1.2 :例子 Foo.idl 文件

一种比较常见的做法是为每一个模组(Module)定义一个 idl 文件,并且以模组(Module)名字命名 idl 文件名。例如:文件 Foo.idl 包含一个 Foo 模组。
#include 指令告诉预处理器包含指定文件的内容。通过这个指令可以将 IDL 同一个模组文件的定义分布在不同的文件中。

在 [图 1.2](#) 中的 #ifndef ... #define...#endif 指令通常用来保证一个 idl 文件不会被多次包含(#include)

关于指令 #pragma prefix 的我们放在 [9.4](#) 小节来讨论。

1.4.2、 IDL使用习惯

1.4.2.1、 工厂接口(Factory interface)

许多的面向对象语言都使用构造器(constructor)来创建和初始化对象,然而构造器(constructor)只是创建本地对象,即和调用构造器(constructor)的进程在同一个进程空间。因为这样,构造器(constructor)

不能用于创建不同进程空间的对象，这也是不能为 IDL 的接口 (interface) 定义构造器的原因。

在 CORBA 中，客户端进程创建不同进程中对象方法是通过调用服务器上对象的方法来实现。工厂 (factory) 这个术语使用来描述具有创建其他对象能力的对象。创建其他对象的方法名字通常是 `create()` 或则包含 "create" 名字的方法，例如 `create_account()`，但是这只是一个命名习惯并不是要求，IDL 中并没有使用额外的语法来描述工厂方法。因此工厂接口和普通接口并没有区别。

[图 1.3](#) 描述了一个工厂接口的定义。

```
interface Foo {  
    void destroy();  
    ...  
};  
  
interface FooFactory {  
    Foo create(...);  
    ...  
};
```

图 1.3 工厂接口的例子

正如 IDL 没有构造器一样，IDL 也没有销毁器 (destructor)。CORBA 中，对象的销毁一般由服务器端 (server) 来自主决定，并不需要客户端的输入。然而，如果需要客户端来控制对象的销毁通常通过定义一个方法来达到这样的目的。这个方法通常被命名为 `destroy()`，但是这仅只是一个命名习惯并不是一个命名要求。

1.4.2.2、 回调接口(Callback interface)

回调函数或回调对象通常被用于图形用户接口 (GUI)：应用程序开发者注册

一个函数或对象到GUI中，GUI就会在相关的事件发生时回调该方法或对象，比如当鼠标被按下或则键盘被按下。回调对象在CORBA中被广泛的使用，但是做为IDL编译器回调接口(例如 [图 1.4](#) 的FooCallBack)和普通接口并没有区别，即IDL中并没有使用特定的语法来定义回调接口

```
interface FooCallBack {  
    void notify_something_has_happened(...);  
};  
  
interface FooCallbackRegistry {  
    void register_callback(in FooCallback cb_obj);  
    void unregister_callback(in FooCallback cb_obj);  
    ...  
};
```

图 1.4 回调接口

1.4.2.3、 迭代器接口(Iterator Interface)

让我们假设一个接口有一个 `query()` 方法，此方法使用一个可变数组作为查询接口的返回。如果返回结果很大，那么一次返回所有的结果将会是相当不明智的行为，这是因为：

- 返回的结果集合可能会大到占据几兆升至几 G 的空间。虽然服务器端可能运行在有充足内存的机器上，但是客户端程序却不一定能做到这一点。返回这样一大堆的结果数据可能导致客户端的程序内存不足而退出。将结果分批给客户端将会是一种比较好的解决方案。
- 在很多的客户端-服务器端程序中都包含了查询式的操作，即交互用户查询结果并显示在给客户，客户选择其感兴趣的条目进行操作，根据概率，有可能用户感兴趣的条目排在查询结果集的开始，如果是这样那就导致网络带宽和内存的浪费来传送和保存所有结果。为了避免类似的浪费，一种好的做法

是将结果分成小块传送给给客户端，如果用户选取了其感兴趣的项，比如说在第一或第二小块，那么剩下的结果将不需要再从服务器传送到客户端。

```
struct Data { ... };  
typedef sequence<Data> DataSeq;  
interface DataIterator {  
    DataSeq next_n_items(in unsigned long how_many);  
    void destroy();  
};  
interface SearchEngine {  
    DataSeq query(  
        in string          search_condition,  
        in unsigned long  how_many,  
        out DataSeq       results,  
        out DataIterator  iter);  
};
```

图 1.5 一个迭代器例子

[图 1.5](#) 展示了一个迭代器接口的例子。方法 `query()` 返回 `how_many` 个结果在 `results` 中。如果 `how_many` 的取值为 `all`，那么 `iter` 参数将返回一个空 (`nil`) 对象引用，并将结果通过 `results` 返回。否则 `iter` 包含一个 `DataIterator` 对象的引用。通过这个对象可以获得更多的结果内容。当 `iterator` 没有更多的结果返回时，`next_n_item` 返回一个空可变数组对象并且客户端可以通过 `destroy()` 方法释放 `iterator`。

1.4.3、 IDL的局限

对比程序语言的复杂数据结构的定义而言，IDL 的复杂数据结构定义是具有一定的局限的。IDL 数据类型的灵活性局限主要表现为缺乏指针，例如：一个

IDL 的结构(struct)类型不能包含一个指向另外一个结构类型的指针。由于缺乏指针这样的类型,使得 IDL 无法定义出类似图的数据结构,这个局限是 IDL 专门为之的,为什么会有这个局限主要有如下几个原因:

- 如果 IDL 包含指针,那么将 IDL 映射为到不支持指针的程序语言时将会非常困难甚至是不可能为之。
- 如果 IDL 支持指针,那么程序员将可以在远程调用的时候传递非常复杂的结构比如说图做为参数。其实这种的灵活性很少被程序员使用,而且为了满足这样并不会带来多少好处的需求将会使 CORBA 编组(marshaling)引擎变得非常复杂。
- IDL 定义的类型一般都用于被公开的 API 而非实现的 API。公共 API 一般传送的都是非常简单数据类型作为参数,因此在实践中,IDL 的局限并不会造成什么问题。当然,Server 端程序可以在其实现程序中使用指针。

我们应该注意到在新的增加的IDL 传值对象(objects by value (OBV))类型具有类似C++指针的功能。我们将在 [9.2](#) 小节讨论OBV

也许另一个经常被人注意到的 IDL 缺陷是 IDL 的异常无法被继承,即一个异常不能被定义为另外一个异常的子类。虽然这个缺陷对项目开发并不会带来太大的影响,但是这个缺陷却是程序员非常的懊恼。这是因为,如果一个方法它会抛出 10 个不同类型的异常,将这个 10 个异常安排在一个继承体系里面将会是非常自然的想法。但是由于 IDL 不支持异常继承,设计者只能被迫将这个 10 个异常一个个罗列方法定义的在 raise 子句中或则采用一个带有一个 error_code 域标示不同异常的“一般 (generic)”异常来代替。这两种方法都使得客户端程序开发者非常“难受”。采用第一种方法,在 try-catch 块中他将有 10 个不同的 catch 子句围绕着方法调用。采用第二种方法,虽然只有一个 catch 子句,但是却需要 switch 或层叠 if-then-else 来判断具体

的异常种类。

1.4.4、 映射IDL到程序语言

正如 [1.4 节](#)提到的, IDL是用来定义在服务器端通过对象暴露公共API。IDL定义的这些API独立于其他任何特定语言。然而为了使CORBA能够被使用, 必须将IDL映射到特定的语言。例如: IDL到C++语言的映射允许开发者使用C++进行CORBA应用程序开发 IDL到JAVA语言映射允许开发者使用JAVA进行CORBA应用程序开发。

目前的 CORBA 规范定义了如下语言的映射, C, C++, JAVA, Ada, Smalltalk, COBOL, PL/I, LISP, Python, IDLScript。这些官方认可的语言映射可以保证源代码在不同 CORBA 产品间移植(关于移植性我们将在 25 章详细讨论)。还有一些非官方的语言映射, 如果你愿意的话, 这些映射包括 Eiffel, Tcl, 和 Perl。很明显, 你能用非官方的语言开发 CORBA 程序, 但是这些语言的源码移植性将得不到保证。

1.4.5、 IDL编译器

IDL编译器的作用是将IDL文件定义好的类型(结构、联合, 序列等等)翻译成某一种编程语言, 比如说C++, JAVA, Ada, COBOL。此外对于每一个IDL的接口(interface), IDL编译器还提供存根(stub)代码——也称为代理([10.3 小节](#))和框架(Skeleton)代码的生成。这些术语经常会对非英语为母语人们带来一些迷惑, 因此我将逐条的解释他们

- 单词存根(Stub)有很多意思, 字典的一个解释是: “一个较大的东西被使用完后的遗留物”, 举个例子, 比如说笔头(pencil stub)或者烟头(cigarette stub)。此外(在非分布式)程序中存根程序(stub

procedure)是指用来阻止编译时候产生未定义标示错误的一个过程或函数的空实现。在分布式中间件系统中,比如说 CORBA, 远程调用在客户端实现是一个本地调用存根方法和对象的实现。存根用来完成进程间的交互传输请求到服务器进程, 并从服务端接收应答。

- 术语代理(proxy)经常被用来替代存根(Stub)这个词, 词典中对代理的定义是: “一个被其他人授权做某事的人”。例如, 你有可能去参加一个关于某问题的投票, 但你却没有办法到达开会地点。于是你指定某人去帮你投票。如果你这样做你就是“通过代理投票”。在 CORBA 或其他的中间件中代理这个术语比较合适。一个 CORBA 的代理对象简单来说就是扮演着服务器端“真实”对象行为的一个客户端对象。当客户端调用代理对象上的方法, 代理通过进程间通讯机制向服务器端真实对象发送请求, 并等待应答然后将应答传给客户端的应用程序。
- 术语框架(Skeleton)指的是服务器端接收和分发请求到服务器应用对象的代码。框架这个词可能是一个非常奇怪的选择。然而 Skeleton 这词本身并不是只是指的是骨头的意思, 更通常的意思是基础框架的意思。正是因为它提供了基础框架代码支持请求到服务端应用代码的调用, 所以被称为框架代码(Skeleton code)

任何 CORBA 产品必须提供 IDL 编译器, 但是 CORBA 的规范并没有明确规定编译器的名称和命令行参数。这些细节实现对于不同的 CORBA 产品实现都不相同。

1.5 可交互对象引用(Interoperable Object Reference IOR)

对象引用(object reference)是客户端程序和CORBA服务对象交流的“contact details联系细节”, 一些人们将对象引用称为可交互对象引

(interoperable object reference (IOR))或则代理(proxy)。之所以被称为交互是因为IOR主要的功能是完成不同CORBA实现之间的交互。这就意味着一个Orbix的服务器端对象能被不同的CORBA产品客户端使用。比如说Orbacus, Visibroker, TAO, omniORB or JacORB。关于对象引用的更深的讨论我们放到[第十章](#)进行。

1.6 CORBA 服务(CORBA Services)

许多语言都提供了标准的库和类作为核心语言的补充, 这些标准库通常提供集合数据结构(例如: 链表, 集合, 哈希表等等), 文件输入输出和其他的一供应用程序开发的功能。如果你要求一个程序员写一段程序而不使用到这些标准库, 比如说 JAVA, C, 或则 C++, 即将会发现程序员将很难完成你指定的工作。同样、如果只用 CORBA 的核心部分(面向对象 RPC 机制和一些通用在线协议)则 CORBA 从能就非常有限, 就像使用一门编程语言只使用本身而不使用其标准库一样。CORBA 服务(CORBA Services)为开发广泛使用的分布式应用提供非常有用的功能, 这些功能大大增加了 CORBA 的能力。这些 CORBA 服务被定义在 IDL 文件里。你也可以把 CORBA 服务看做成某种标准库, 当然大多数 CORBA 服务都与预建服务程序的形式提供, 而不是以库的连接到你应用程序的形式提供。正因为这样, CORBA 服务是一个真正的分布式, 标准类库。

一些通用的 CORBA 服务就会在本书的其他章节里面讨论:

- 名字服务(Naming Service[第 4 章](#))和Trading Service([第 20 章](#))允许应用程序发布其对象, 使客户端应用程序更容易找到其请求的对象
- 大多数CORBA服务都是同步点对点通讯, 但是一些应用程序要求多对多, 异步通讯, 或则被人们称为的发布者-订阅者 (publish and subscribe) 通讯方式。多样的CORBA服务 (Various CORBA Services [第 22 章](#))

将讲述这种服务。

- 许多程序开发者都熟悉数据库事务。在分布式系统中，一个事务经常跨多个数据库，为了保证所有数据库一致性的更新或不更新操作，对象事务服务 (Object Transaction Service OTS 在 [22 章](#) 讨论) 提供这种能力。

第2章、CORBA的优点

- [成熟的技术](#)
- [开放的标准](#)
- [广泛平台支持](#)
- [广泛语言支持](#)
- [高效](#)
- [扩展性强](#)
- [CORBA的成功案例](#)

在 [1.2](#) 小节，我们曾说过，CORBA是一种中间件，但是这个世界上有很多的中间件，一个很自然的问题是：我们为什么不选用其他中间件而一定要用CORBA呢。原因就是我在本章讨论的内容。CORBA提供了很多的优点，你可能在其他中间产品中能发现这些优点，但是没有一个中间件能提供CORBA所有的优点。

2.1 成熟的技术

CORBA 规范最原始的版本定义在 1991 年，第一个 CORBA 规范定义非常有限，这是 OMG 故意为之。OMG 的哲学是：定义一个非常小的标准，让实现者去获得经验然后并扩展其版本，使其具有更强的能力。这种“慢速但是稳定”的发展方法取得了显著的成效。在 CORBA 规范的定制过程中很少出现新版有旧版本不兼容的修改，新版本大都都是增加了旧版本的功能。今天，CORBA 已经发展

成为了一个提供多语言、多操作系统支持，并且提供了丰富的能力，例如：事务，安全性，名字服务，消息，发布-订阅者服务的中间件。CORBA 提供的这些服务本质上都被应用到企业级应用上。许多新的中间件技术声称他们比 CORBA 更好，但事实是他们只拥有 CORBA 很多年前就拥有功能的一小部分。

2.2 开放的标准

CORBA 是一个开放而非某个公司封闭的技术。这是非常重要的，因为：

第一、用户可以从众多的 CORBA 产品中选择，或则选择免费软件。你可能认为从 CORBA 的一个产品移植到另一个 CORBA 产品上是一件非常困难的事情，但是事实并非如此，如果你按照第 [25 章](#) 建议做的话，你将会发现移植将会变得非常轻松。但是如果你使用封闭的中间件产品的话，你将会发现的你的移植工作将非常具有挑战性。

第二、不同的 CORBA 产商的竞争，降低 CORBA 软件的使用成本。

最后、许多的封闭中间产品都是假设你的应用将完全依赖于他的中间件技术，所以他们只为你提供了有限的移植你应用到其他中间产品的支持。而 CORBA 本身就是设计目标就是能方便的集成其他中间件产品。在 CORBA 的规范中，明确支持集成 TMN, SOAP, Microsoft's (D)COM and DCE(在 CORBA 前的一个中间件规范)。此外，J2EE 的从 CORBA 里面借鉴了许多的概念，这使得 CORBA 集成 J2EE 变得相对简单。一些厂商出售 J2EE 与 CORBA 之间的网关使二者更易集成。一些 CORBA 厂商出售 COM-到-CORBA 和 .NET-到-CORBA 的网关。这为那些希望在 Windows 上用 Visual Basic 写客户端和其他不同机器比如说 unix 或主机做服务器的人提供了非常实用的解决方案。Visual Basic 的被写成 .COM/.NET 客户端组件和服务端进行通讯，而实际上他们是

通过网关转发到 CORBA 的服务器上。

2.3 广泛的操作系统支持

CORBA 在不同的机器上有实现, 包括 IBM OS/390 和富士 GlobServer 主机, 还有一些 UNIX 及其相关的变种操作系统包括: LINUX, Windows, OPEN VMS, Apple's OS X 和其他的一些嵌入式系统。其他的中间件很少能有这样的支持。

2.4 广泛的语言支持

CORBA 标准定义了到多种语言的映射, 比如说 C, C++, Java, Smalltalk, Ada, COBOL, PL/I, LISP, Python 和 IDLScript。一些小公司开始可能只使用一种语言开发他们的应用, 但是随着规模的扩大这些公司有可能开始使用其他的语言。相似的, 有可能公司以前的程序是由一种语言写成, 但是新的程序却是由多种语言写成。因此, 对于中间件来说支持多种语言变得非常的重要; 不幸的是并不是所有的中间件系统都注意到这一点。极端的例子是 J2EE, 其只支持 Java。另一个极端的例子是 SOAP 中间件标准。SOAP 应用程序可以用各种语言编写, 但是一旦开始写程序, SOAP 标准只定义了一种语言映射 (Java)。一些厂商提供了 C++ 的编写 SOAP 应用程序的, 并且这些厂商都提供了其私有的 C++ API。这就意味着对于非 JAVA 的 SOAP 应用而言不存在着源码的移植性。

2.5 高效

CORBA 的联机协议功能模块保证了其服务器到客户端的数据以非常紧凑的形式传送。并且大多数的编组 (marshal) 数据 (就是将程序的数据转换成二进制

的字节流)实现都是非常高效的。其他的一些中间件技术也有采用类似的紧凑格式或高效的编组架构来传输数据。但是这些中间件的解决方案并不完美,正如我们即将讨论的一样。

SOAP 使用 XML 来表示传输数据。XML 的冗余数据表示使得 SOAP 比 CORBA 占用更多的网络宽带。将程序中的数据转换成 XML 和将 XML 转换成程序中的数据使得基于 SOAP 的应用程序势必消耗更多的 CPU 资源。

其他的一些中间技术,比如 IBM 的 MQ Series,统一采用二进制流来传输数据,虽然这样做非常高效,但是这样却要求应用程序开发员自己写程序来完成内存数据到二进制流的转换。相比较而言,CORBA 通过 IDL 编译器来生成编组(marshaling)和解组(unmarshaling)代码使得程序员不需要编写和维护着样底层的代码

2.6 易扩展

灵活性,CORBA 的服务器程序基础功能模块使得开发的 CORBA 服务器能够处理从少数的对象的服务器到无限个对象的服务器,很明显对于不同 CORBA 实现,CORBA 不同产品的扩展性也不尽相同。但是随着时间的推移,CORBA 服务器不仅可以处理大批量的服务端数据,而且也可以处理上千客户端通讯数据。最终,大多数的 CORBA 服务器厂商将会发现,大部分的封闭中间件客户将会因为封闭中间件产品的扩展性不足而试图转向使用 CORBA 中间件。

2.7 CORBA成功案例

由于上述讲了如此多的 CORBA 优点,我们将一点都不奇怪 CORBA 现在被成功的运用到各行各业包括:太空、咨询、教育、电子商务、财务政府、医疗、教育、人力资源、保险,制造业,军队、生化、出版、研究、电信等等行业。

CORBA 被用于计费系统、多媒体、机场跑道灯光控制、太空无线电、电话交换机。全球大多数的电信系统,和一些执行重要任务的大型银行系统都是用 CORBA 来实现的。

讨论世界有多少的使用CORBA而得到益处的项目已经超出了本书的讨论范围,然而越来越多的使用CORBA的成功案例可以在互联网上找到。例如:你可以通过 www.corba.org找到超过 300 个的成功案例。一些CORBA厂家的主页上也会包含一些成功案例说明

第二部分 应用程序介绍

第3章、CORBA应用程序开发

- [开发传统应用程序](#)
- [开发CORBA应用程序](#)
- [CORBA应用程序开发要点](#)
- [其他杂项](#)

本章将用伪代码做一个开发 CORBA 客户端程序简单示例。虽然这些伪代码的形式比较接近 C++或 Java。但是里面的基础概念演示也适用于其他语言的开发。使用伪代码来演示开发 CORBA 应用程序基本概念的主要好处是为了不让读者被不同的语言映射问题干扰。

3.1 开发传统应用程序

[图 3.1](#) 展示了一个传统应用程序(非分布式)，面向对象的应用程序。类声明和类定义放在同一个头文件中。主程序创建一个或多个该类对象并调用其方法。

```
// File: Account.h
class Account {
    void deposit(...) { ... }
    ... // instance variables
};

// File: main.cpp
main(...)
{
    Account obj = new Account(...);
    obj.deposit(...);
}
```

```
}
```

图 3.1 传统应用程序结构

3.2 开发CORBA应用程序

我现在将讨论一个功能和图 3.1 类似的程序结构，但是这个程序是采用 CORBA 的分布式程序。

3.2.1、 IDL文件和代码生成

首先：开发CORBA应用程序的第一步是要使用idl来定义一个类型和其公共接口，具体的参看 [图 3.2](#)。注意看这个Account定义的形式和C++定义类的方法非常相似。除了语法上稍有不同，比如class关键字被interface替换。然而更重要的是这个IDL文件只声明公共方法的接口，并不包含公共方法的实现。

```
// File: Account.idl
interface Account {
    void deposit(...);
};
```

图 3.2 Account.idl

一旦 idl 文件被定义好，就可以使用 idl 编译器来编译它，例如：

```
idl Account.idl
```

注意：CORBA规范中并没有定义IDL编译器的名字和其相关的参数选项。所以具体的命令执行方法在不同的CORBA产品中存在着差别。如果你使用C++的CORBA产品，那么IDL生成的类将对应IDL定义数据类型的C++数据类型代码。如果使用Java或则COBOL的CORBA产品则生成JAVA和COBOL对应类型的代码。在生成的代码中包含一个叫做Account代理类(proxy class)和一个叫做

POA_Account框架类(skeleton-code)。伪码的类容如 [图 3.3](#) 显示

```
// Generated code
class Account {
    void deposit(...)
    {
        marshal request details into a binary buffer
        Send request buffer message to server
        Wait to receive reply from server
        if (reply buffer contains an exception) {
            unmarshal exception and throw it
        } else {
            unmarshal "out" parameters from reply buffer
        }
    }
};

class POA_Account {
    abstract void deposit(...);
    void dispatch(...)
    {
        unmarshal "in" parameters from request buffer
        try {
            deposit(...);
            marshal "out" parameters into reply buffer
        } catch(...) {
            marshal exception into reply buffer
        }
        Send reply buffer to client
    }
};
```

图 3.3 IDL 编译器生成的代码

客户端通过调用本地的代理类对象(local proxy object) 完成远程调用。代理对象执行编组(marshal [11.2 小节](#)) 操作, 对象键值(object key [5.6.1 小节](#)) 唯一的标示了服务器上的目标对象。当方法被调用时, in/inout 参数被转换成二进制流, 并且这些二进制流被传送到包含目标对象的服务器进程中, 然后, 代理类等待从服务器上接收应答信息并且解组(unmarshals) out/inout 参数并返回值, 如果在这个过程中返回信息中包含异常信息, 代理类解组后抛出改异常。

对于每一个 IDL 定义的方法, 生成的框架类(skeleton class) 包含一些抽象方法(在 C++里面是纯虚函数)。这个方法在框架类中不被实现, 而是由开发人员写的子类来实现。框架类中还包含解组请求调用正真的实现方法(在我们这个例子中是 deposit()) 同时解组 in/inout 参数等等这些分发功能的逻辑代码。当方法返回, 框架类编组 out/inout 参数和返回值。并传送回客户端。

开发人员并不需要知道底层代理类和框架类的工作细节, 只需要知道他们是用跨网络转发客户端请求到服务器真正对象方法实现底层框架代码即可。

3.2.2、 服务提供者类 Servant Classes

CORBA 使用服务提供者 Servant 这个术语来表示在宿主语言(C++, Java, Cobol) 实现了 CORBA 对象功能的对象。在 CORBA 中, 服务提供者 Servant 并不是一个 CORBA 对象, 但是他代表了 CORBA 对象。我们将在第五章讨论关于 CORBA 对象和服务提供者对象细节上的差别。

```
// File: AccountImpl.h  
class AccountImpl inherits POA_Account
```

```

{
    void deposit(...) { ... }
    ... // instance variables
};

```

图 3.4 服务提供者类(Servant class)

Servant 类并不是由IDL编译器生成。他是程序开发着手工编写的。当然程序员可以使用任何他喜欢的名字来命名这个类。但是在通常的模式下 Servant 类的名字应该由IDL接口名加一个后缀例如Impl组成。例如 AccountImpl 就是 Account 接口对应的一个 Servant 类。Account 的 Servant 的伪码在 [图 3.4](#) 中。这个Servant类继承了生成的框架(skeleton)类。并且他必须实现所有定义在IDL里面的方法，例如我们例子中的deposit()方法。Servant类可以包含构造方法，实例变量和一些非IDL定义方法来支撑IDL定义方法的实现。

3.2.3、 服务器主程序

[图 3.5](#) 罗列了服务器的主程序，最重要部分用黑体字标明。

```

// File: server_main.cpp
int main(int argc, char* argv[])
{
    exit_status = 0;
    orb = null;
    try {
        orb = CORBA::ORB_init(argc, argv);
        ... // create POAs to contain servants
        sv = new AccountImpl(...);
        ... // activate (insert) sv into a POA
    }
}

```

```

        exportObjRef(..., sv._this(), ...);
        ... // activate POA managers
        orb.run();
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    // Terminate gracefully
    try {
        if (orb != null) { orb.destroy(); }
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    return exit_status;
};

```

图 3.5 服务器主程序伪码

ORB_init()函数创建一个代表 CORBA 运行系统的对象新的 ORB。程序的输入参数同时也传递给此函数，ORB_init 函数解释参数中含-ORB<name> 和 <value>对的参数，并根据参数配置新生成的 ORB 对象。这些识别出来的参数值对通常在不同的 CORBA 产品中不同，但是这些参数值对通常用于一些特殊的信息，比如 CORBA 运行时的诊断信息，CORBA 服务器进程监听端口号，包含其他参数值对文件的文件名等信息。

CORBA 程序结束时，服务器必须调用 orb.destroy 这个函数，这个方法保证了 CORBA 应用程序结束时回收其相应的资源，并优雅的开始进程。

当 CORBA 的 API 函数调用出错时这些函数会抛出异常。因此用 try-catch

包围主 `main` 函数大部分代码将保证当函数调用出错时，应用程序都能调用到 `orb.destroy()` 函数。

虽然 [图 3.5](#) 只是演示了一个Servant被创建，CORBA允许服务器进程创建IDL文件中多个接口对应的多个Servant。当服务器进程实现了多个IDL定义接口，服务器端的开发者可以使不同的Servant具有不同的服务质量(qualities of service (QoS))。例如：如果某一个Servant实现是线程安全的方式，那么CORBA的运行系统应该要保证能并发的派发客户端请求到对应的Servant。相反的，如果一些Servant的实现不允许多线程并发的时候，CORBA运行系统要保证请求以串行的方式派发到对应的Servant上。这种让一种服务质量(QoS)关联到一组Servant，另一种(QoS)关联到另一组Servant的方法，是通过让服务器创建多个POA来实现的。关于POA的细节我们在[第5章](#)讨论，但是本质上说POA就是一组Servant的集合。POA的QoS在POA创建时就确定下来。当某一个Servant在某一个POA里被激活时（插入到某一个POA容器时）时，这个Servant就具有了它所在POA的服务质量(QoS)

当程序调用`ORB_init()`函数后，CORBA会创建一个或多个典型的POA。然后就可以将Servant创建出来并放在这些POA里面激活（或插入到这些POA里）。对于服务器端来说创建一两个工厂对象([1.4.2.1](#)小节)，然后在通过工厂对象创建Servant是非常常见的方式。

在 [3.2.2](#) 小节提到Servant代表一个CORBA对象，Servant的方法 `_this()` (这个方法在Servant的自动生成父类即框架(skeleton)类中被定义)调用后可以获得Servant对应的CORBA对象的引用。服务器通过文件、名字服务(Naming Service)、交易服务(Trading Service)、输出对象引用而

达到向外公告其CORBA对象的目的。 [图 3.5](#) 中exportObjRef()函数不是CORBA的API，这是一个伪码，这个函数的含义是通过某一种方式公告Servant对应CORBA对象的引用。

当服务器初始化完毕，就可以激活器POA管理器(POA Manager 具体讨论在 [5.7 节](#))并调用orb.run()方法进入事件循环。在这个事件循环中，CORBA运行系统从客户端接收连接，读取请求并分发到对应Servant的框架类。除非调用了orb.shutdown()方法，否则orb.run方法不会返回。orb.shutdown一般是在信号处理函数中被调用，或则在某一个IDL定义名字类似为shutdown()或则kill_server()的方法中调用

3.2.4、 客户端主程序

[图 3.6](#) 演示了一个CORBA客户端程序的伪码片段,重要的代码段使用了加粗字体

```
// File: client_main.cpp
int main(int argc, char* argv[])
{
    exit_status = 0;
    orb = null;
    try {
        orb = CORBA::ORB_init(argc, argv);
        obj = importObjRef(...);
        obj.deposit();
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
}
```

```

// Terminate gracefully
try {
    if (orb != null) { orb.destroy(); }
} catch(CORBA::Exception & ex) {
    cout << "Something went wrong: " << ex << endl;
    exit_status = 1;
}
return exit_status;
};

```

图 3.6 客户端主程序

如同 Server 端程序一样,客户端程序也调用 ORB_init 函数初始化 CORBA 运行系统,调用 orb.destroy 来确保在程序结束之前,CORBA 运行系统被优雅地结束。

[图 3.6](#) 中的 importObjRef() 函数不是 CORBA 的 API, 这个是一个伪码函数, 这个函数的功能是通过某一种方式, 比如文件、名字服务、交易服务中倒入 CORBA 对象引用。

一旦客户端程序得到服务器某一个对象引用, 客户端就可以通过这个引用调用其服务器端对象的方法。

3.3 CORBA 应用程序开发要点

[图 3.1](#) 中传统的应用程序伪码比 [图 3.2](#) [3.3](#) [3.4](#) [3.5](#) 的伪码更简洁。开发传统应用程序只需要 9 行代码, 而开发 CORBA 的应用程序代码是传统程序的 5 倍。从这个比较, 很容易得出开发 CORBA 应用程序的困难度是开发传统应用程序的 5 倍。但是这个结论并不正确, 这是因为在 CORBA 应用程序中的额外步骤比如 ORB_init 和 orb.destroy, 创建 POAs, 输出 (export) 或导入 (import) 对

象引用等等，这些步骤不管你开发小CORBA应用或大CORBA应用都是一样的。而当你真正开发一个CORBA应用时，你就会发现这些关于CORBA架构的代码相对于你的应用逻辑而言是非常小的一个部分。

3.4 其他杂项

3.4.1、 `resolve_initail_references()`

在 [3.2.3 小节](#) 我们讨论过，`ORB_init` 函数会创建一个新的 ORB 对象，这个对象表示 CORBA 的运行系统。ORB 对象上定义一个叫 `resolve_initial_references()` 方法。很多人认为这个方法的名字并不太直观，其实这个名字包含很多信息，包括如下的两点：

1、在 CORBA 的术语中，`resolve` 意味着查找(`lookup` 或则 `find`)。所以这个方法意味是用来实现查找功能的。

2、这个方法的名字有一点被误用，更确切的应该被称为 `resolve_initial_reference` 而不是 `resolve_initial_references`，因为这个方法是用来查找一个而非多个对象引用的方法

`resolve_initial_references` 方法是用来查找其他 CORBA 基础对象的 API。它使用一个字符串参数来指明其所希望查找的对象。许多的 CORBA 服务(CORBA Service)都可以通过这个方法获得。例如，你可以通过字符串 "NameService" 为参数来获得名字服务(Naming Service)引用，同样你也可以通过字符串 "NotificationService" 来获得通知服务(NotificationService)的引用。不管那一种情况，这个方法都是返回 Object 类对象的引用，程序员必须通过 `narrow` 这个方法将返回的引用转换成

所查找对象的引用。

`resolve_initial_references`方法不仅可以用于查找CORBA服务，他还可以用于查找CORBA基础对象的引用。例如查找Current对象([13 章](#))，DynAny对象的工厂([15.3 小节](#))，根POA对象([5.5 小节](#))等等。

在 [图 3.5](#)、[图 3.6](#) 中并没有显示的`resolve_initial_references`的调用代码。然而在真正的CORBA应用代码中将会用`resolve_initial_references`来访问根POA，为了导入/导出对象引用，代码也会调用`resolve_initial_references`来访问CORBA的名字服务和交易服务。

3.4.2、 对象引用字符串

ORB 提供了一个 `object_to_string()`方法将对象引用(代理对象)转换成一个字符串。另外一个 `string_to_object()`方法是将字符串转换成对象引用。通过编组(marshaling) `object_to_string()`方法将 IOR 转换成前缀是 IOR:的一个 16 进制表示缓存。

通过`object_to_string()`和`string_to_object()`方法，服务器可以将其对象公告给客户端，例如服务器可以将代表对象的字符串的内容写到一个文件中。客户端应用程序从文件读出内容然后调用将字符串转换成对象引用(代理对象)，其他的公告对象的方法我们将在 [第 4 章](#)和 [第 20 章](#)详细讨论

许多 CORBA 产品都提供了工具来解析和查看代表着 CORBA 对象引用的字符串(IOR 字符串)内容。虽然这个工具是相互通用的，但是 OMG 并没有标准化这些工具，所以对于不同的 CORBA 产品，使用这些工具将有不同操作方式和显示方式。下面罗列了一些常见的这些工具的名称

- Orbix 和 Orbacus 提供的工具恰巧名字都叫做 iordump，但是由于他们不同的公司开发的工具，所以他们的操作和显示方式并不相同。
- omniORB 和 TAO 提供的工具恰巧名字都叫做 catior，但是由于他们不同的公司开发的工具，所以他们的操作和显示方式并不相同。
- JacORB 提供的工具名字叫做 dior，他是“decode IOR”的简写。
- 下面这个连接提供了一个解析表单，你可以将 IOR 字符串拷贝到文本框，然后点击解析按钮，网页将显示你的 IOR 字符串的具体内容。

<http://www.parc.xerox.com/istl/projects/ILU/parseIOR/>

3.4.3、 联系法 (Tie Approach) 实现Servant

[3.2.2 小节](#)提到Servant继承于框架(skeleton)类。事实上CORBA支持两种方式来实现Servant类。一种方式就是[3.2.2 小节](#)的那样Servant类继承框架(skeleton)类。另外一种方式是IDL编译器生成一个类，这个类通常被称为联系(Tie)类，这个类继承了框架(skeleton)类，同时这个类代理了(delegates)一个Servant类(含有一个Servant类，译者注：面向对象的has a 语义)。在这种情况下Servant不直接继承框架(skeleton)类

在支持多重继承的语言中(比如 C++)，一般采用继承方法，在一些不支持多重继承的语言中(比如说 Java)一般就采用联系方法。

第4章、 名字服务 (Naming Service)

- [基本概念](#)
- [IDL中缺点](#)

- [名字服务在实践中的用法](#)

服务器公告器对象的一种方法就是调用 `object_to_string()` 将对象引用字符串化, 然后将字符串内容写在文件中。客户端应用程序从这个文件读出这个字符串的内容, 调用 `string_to_object()` 方法将字符串转换成代理对象(proxy)。

在客户端和服务端都能访问一个共享文件系统的情况下, 这种机制可以很好的工作。这种情况一般发生在客户端和服务端在同一台机器或者客户端和服务端在同一个局域网之内。然而在广域网上共享文件系统将会非常困难。因此, CORBA 经过多年的发展, 提供了更高级的位置无关的访问方法, 这种方法就是本章讨论的名字服务Naming Service, 另一种是将会在 [20 章](#) 讨论的交易服务Trading Service

4.1 基本概念

现实生活中的电话簿提供了一种名字到联系方式(电话和地址)之间的映射方式。同样的, CORBA名字服务Naming Service也提供了一种(可读的)名字到对象“联系方式”(IOR)的映射方式。电话簿里的名字是字母排序的查找, 名字服务中以(类似于UNIX和Windows的文件系统的)层次关系来安排数据。在名字服务中的每一层被称为一个名字上下文naming context (在Windows或unix中称为目录)。名字上下文本身也是一个CORBA对象(它被定义为 `CosNaming::NamingContext`, 如 [图 4.1](#))。由于CORBA对象可以通过位置无关的方式访问, 这意味着名字服务层次能被包含单一个进程中, 这个层次也可跨多个名字服务器进程。这种连接多个名字服务器的方式称为联合federation。许多公司一般都采用单一名字服务器的方式。当然也有一些公司使用联合的名字服务的方式。例如: 一个公司为每一个部门或分支机构都使用一个名字服务进程, 然后通过联合将所有的名字服务组织成一个逻辑单元, 这就是

一个“公司级company-wide”的名字服务

```
#pragma prefix "omg.org"
module CosNaming {

    typedef string Istring;

    struct NameComponent {

        Istring id;

        Istring kind;

    };

    typedef sequence<NameComponent> Name;

    enum BindingType {nobject, ncontext};

    struct Binding {

        Name          binding_name;

        BindingType   binding_type;

    };

    typedef sequence <Binding> BindingList;

    interface BindingIterator {

        boolean next_one(out Binding b);

        boolean next_n(in unsigned long how_many,
                       out BindingList bl);

        void destroy();

    };

    interface NamingContext {

        void bind(in Name n, in Object obj) raises(...);

        void rebind(in Name n, in Object obj) raises(...);

        void bind_context(in Name n, in NamingContext nc)
                           raises(...);

        void rebind_context(in Name n,
```

```

        in NamingContext nc) raises(...);
Object resolve(in Name n) raises(...);
void unbind(in Name n) raises(...);
NamingContext new_context();
NamingContext bind_new_context(in Name n)
        raises(...);
void destroy() raises(...);
void list(in unsigned long    how_many,
        out BindingList      bl,
        out BindingIterator bi);
};
... // interface NamingContextExt omitted for brevity
};

```

图 4.1 名字服务的 IDL

名字的服务功能定义在 IDL 的相应接口 interface 中。其中一些方法可以用来增删改 create/modify/delete 名字上下文的层次。其他的一些方法用来在名字服务绑定 bind 或公告 IOR，当然你也可以通过 resolve 方法来查找名字对应的 IOR。

一些名字服务的实现会提供一些命令行工具或图形化工具来管理名字服务，比如增删改名字上下文层次。这些管理功能通过调用名字服务接口上特定方法来实现。

CORBA 的名字服务规范里面并没有指明关于名字服务要达到的服务质量 QoS。一些名字服务实现只满足在内存中实现名字到 IOR 关系的映射，这样的缺点是当名字服务进程结束后，映射关系也随之消失，这样做的好处是名字服务不需要访问特定的存储设备(对于嵌入式开发非常有利)。一些名字服务提供 IOR 到名字映射关系的持久化。

大多数甚至全部，CORBA 的实现都会包含名字服务的实现，这意味着你不需

要为名字服务增加额外的开销。你使用的 CORBA 配套实现的名字服务的服务质量 QoS 可能无法满足你的需求。如果这种情况发生，你可以使用其他 CORBA 厂商的名字服务实现，甚至你自己也可以实现一个名字服务。在真实情况下，你一般都会满意于你选择 CORBA 实现配套的名字服务实现。这种名字服务(或其他服务)实现的“剔除并替换”的能力，只是在告诉你，只有具有明确规范和实现界限的开放标准(比如说 CORBA)才能拥有如此灵活的能力。

4.2 IDL 中的缺点

虽然名字服务的概念非常简单，但是 OMG 在定义其 IDL 时却做了一些糟糕的选择。这些糟糕的选择经常会引起一些名字服务的开发人员的迷惑。通过讲解这些设计选择的动机，迷惑会快速消失。本节的目的就是解释这些设计的动机，帮助开发人员避免这些不必要的困惑。非开发人员可以直接略过本节的内容。

名字服务的 IDL 定义在 CosNaming 模组中，在这个模组中，NamingContext 接口定义了操作名字上下问的方法。应用程序可以通过以字符串“NameService”做参数调用方法 `resolve_initial_references()` ([3.4.1 小节](#)) 获得名字服务的根名字上下文。第一个让人困惑的地方就是对于严谨的程序员来说这个获得根名字上下文的参数应该是“名字服务 NamingService”而非“命名服务 NameService”

余下大部分的名字服务 API 的困惑来自于名字服务关于层次的表示格式。事后来说，OMG 组织应该采用一个以“/”为分割的字符传来表示名字上下文的层次性。例如，“path/in/naming/service”。然而这样做也有如下几个问题

- 在 OMG 定义层次名字的事后本想采用多字节字符而非单字节字符，但是那个时候多字节字符还未被引入到 IDL 中

- 使用 “/” 做为层次的分隔符对于 unix 的程序员背景来说将会非常熟悉，但是其他操作系统的层次文件系统的分隔符却不相同。因此并没有特别有力的理由选取任意选一种分隔符作为层次的分隔符而不选其他的分隔符。
- 许多文件系统都有名字和扩展名的概念。例如文件名 “foo.txt”，文件名是 foo，扩展名是 txt，“.” 作为二者的分隔符。一些人认为，这样的灵活性也应该被使用到名字服务中

OMG 试图采用如下的定义来解决 string/wstring 的问题

```
typedef string Istring;
```

在名字服务相关的 API 中将会使用 Istring, 如果以后 wstring 被引入到 IDL 中那么以后上面的定义可以改成

```
typedef wstring Istring;
```

无疑, 这看上去似乎是一个比较好的解决方法。但是这个方案修改 Istring 的定义将会导致向后兼容问题。正是因为向后兼容, 这样修改 Istring 定义从来没有使用过。这样带来的负面影响是名字服务中有一个无用的 typedef 定义, 这会使很多的人感到困扰。

为了避免硬编码的使用层次分隔符 (例如 “/”), OMG 决定层次名字使用一个可变数组 (sequence) **Name** 组件来代替。为了满足名字扩展名的方式能被在 CORBA 中被使用, OMG 决定名字组件是一个包含两个域的结构。这个层次名字的最终定义如下:

```
typedef string Istring;  
struct NameComponent {  
    Istring id;    // denotes the "foo" part of "foo.txt"  
    Istring kind; // denotes the "txt" part of "foo.txt"  
};  
typedef sequence<NameComponent> Name;
```

这个定义非常的灵活, 但是它也太过工程化 (over-engineered)。大多数

的人都不需要使用这样的灵活性, 并且使用这样的结构编写代码太过复杂。例如, 一个应用程序从一个配置文件中动态读取一个“path/in/naming/services”形式的字符串。开发人员不得不写一个函数来将字符串转换成 `CosNaming::Name` 格式。这样存在的问题就是, 花费全球大多数公司的开发员的时间和精力来写这样重复的函数。另一个问题是, 大部分开发人员都会选择他们自己使用的 id 和 kind 之间, Name 之间的分隔符。但是时间证明大多数的程序员他们使用的分隔符都是 “.” 和 “/”

```
#pragma prefix "omg.org"
module CosNaming {
    ...
    interface NamingContextExt : NamingContext {
        typedef string StringName;
        StringName to_string(in Name n) raises(...);
        Name to_name(in StringName sn) raises(...);
        Object resolve_str(in StringName sn) raises(...);
        ...
    };
};
```

图 4.2 NamingContextExt 接口

经过多年后, OMG决定通过定义一个新的命名服务来版本简化层次化名字的复杂性。就象我将在 [9.3 小节](#) 讨论的那样, IDL没有包含版本机制, 所以只有通过从NamingContext接口继承新的接口来实现简化。新的类型定义如 [图 4.2](#) (为了简化省略了raise子句中的内容)。to_string()和to_name()实现了CosNaming::Name到“path/in/naming/service”之间转换。客户端可以调用 resolve_str() 直接查找 IOR , 而不需要再将“path/in/naming/service”转换成CosNaming::Name形式后再查找

IOR。然而NamingContextExt并没有定义公告IOR的方法bind和rebind的简化版本。

最后一个让开发人员感到困惑的是list() 方法，[图 4.1](#)。方法的输出参数提供了一种非递归罗列名字上下文的条目方式。由于罗列方式是非递归的方式，Binding结构的bind_name字段应该是一个NameComponet类型，但是类型是Name给开发者误认为list函数提供递归的方式罗列名字。

本节讨论了名字服务 API 中的缺陷，这些缺陷会导致新开发人员迷惑，但是一旦程序了解到这些缺陷，使用这些 API 将会是非常简单的事情。当然对比其他一些系统的 API 缺陷，这些缺陷是比较微小的。

4.3 名字服务在实践中的用法

IDL 定义的名字服务方法用来增加删除修改名字上下文 NamingContext 的层次。然而通过写程序来执行维护名字上下文的操作是非常令人厌烦的事情。幸运的是很多 NamingService 的实现都提供命令行或图形化工具来包装这些操作。这些命令行和图形化界面提供一种可操作的管理名字服务的方式。但是要注意到，这些东西并不属于 CORBA 的规范。所以对于不同 CORBA 实现，这些工具也存在着较大的不同。

一旦名字服务的管理问题被确定，关于名字服务编程方面的问题就只有服务器绑定 bind——向名字服务输出对象引用，和一个客户端如何查找 resovle()，——从名字服务导入对象引用。这些都是非常简单的编程。

```
try {
    instructions1 = "name_service#path/in/Naming/Service";
    instructions2 = "file#/path/to/file.ior";
    obj1 = importObjRef(orb, instructions1);
    exportObjRef(orb, obj2, instructions2);
}
```

```
} catch(ImportExportException ex) {  
    cout << ex << endl;  
}
```

图 4.3 importObjRef 和 exportObjRef

《CORBA Utilities package》[\[McH\]](#)中关于导入和导出对象引用相关的章节提供importObjRef()和exportObjRef()工具函数。根据这两个函数的名字，可以看出这两个函数主要功能是导入和导出对象引用。[图 4.3](#)演示了伪码的例子。例子中这些函数使用instruction参数来指定如何导入或导出一个对象引用。instruction参数是以“name_service#”打头的字符串代表将使用名字服务。当然也支持“file#<filename>”或“exec#<command>”通过文件或外部可执行命令来导入或导出对象引用。当然程序不应该硬编码instruction的内容，而应该从配置文件或命令行输入。像这样做可以提高程序的灵活性。出了灵活性外，这些函数提供更简单使用名字服务的方式。

第5章、服务端程序开发概念

- [对象 Object](#)
- [服务提供者 Servant](#)
- [为什么对象Object和服务提供者Servant是不同的概念](#)
- [对象适配器 Object Adapter](#)
- [可移植对象适配器 Portable Object Adapter \(POA\)](#)
- [不同类型的POA](#)
- [POA管理器 POA Managers](#)

5.1 对象 Object

“公司”往往是一个逻辑概念而非一个物理实体。例如，一个建筑物属于某个公司，但是这个建筑物并不是一个公司。同样的，雇员不是公司（但是雇员能代表公司）。就象公司是逻辑概念一样，CORBA 对象也是一个逻辑概念而非物理实体。

5.2 服务提供者 Servant

雇员代表公司，例如，假设为你家里提供天然气是 EasyGas 公司。你可能对你的朋友有这样的描述，比如“昨天我跟 EasyGas 公司谈了，希望他们修改我的付款计划”，从技术上说，你不能和一个公司交谈，公司只是一个法律上的概念，所以你无法和一个概念交谈。“CORBA 客户端调用账务服务器上一个 CORBA 对象”。技术上说，CORBA 对象只是一个概念，所以没有办法调用一个 CORBA 对象，更精确的说法是：请求被代表 CORBA 对象的服务提供者 Servant 处理了。

Servent 可以是程序语言里实现服务器程序中的结构或对象。比如，一个 Servant 可以是 Java 或 C++ 中的类实例对象。

5.3 为什么对象 Object 和服务提供者 Servant 是不同的概念

一个明显的问题是为什么 CORBA 规范要区分 CORBA 对象和 Servant 这两个概念。我们可以类似雇员和公司的关系来解释这个问题。

首先公司的存在是独立于个人的存在的。例如，当你昨天打电话给 EsayGas 公司时，你可能和一个叫做 John 的员工通话，如果 John 离开公司，那么你明天再次打电话给 EsayGas 时可能就会和另外一个叫 May 的员工通话。同样的当一个 CORBA 客户端调用一个 CORBA 对象时，这个请求也许被一个特定的

Servant (C++ 或 Java Object) 处理。如果服务器进程后来被终止那么那些相关的 Servant 也会被销毁。当服务器重启时，一个代表此 CORBA 对象的新的 Servant 又会被重建，这就意味着所有以后针对此 CORBA 对象的调用都会被不同的 Servant 处理。也就是说，CORBA 对象是“不朽”的，服务器停止和重启并不影响 CORBA 对象的存在（就像公司不管解雇和雇佣新员工，公司也会存在）

5.4 对象适配器OA

对象适配器 (OA) 是 CORBA 运行系统的一部分，OA 适配 CORBA 对象和真实宿主语言以及服务器进程。OA 处理一些底层问题。例如：

- 从 Socket 连接上读取请求，解组请求参数分发请求到相关 Servant 的方法上
- 提供相关的 API 允许对象引用映射到相应的 Servant 上，或则逆过程。
- 按请求创建 Servant 和 Servant 持久化数据库。

CORBA 规范故意模糊了关于对象适配器的功能，这是因为 OMG 认为无法用一个统一的规范来定义对象适配器，而是希望能使用者有不同的适配器。CORBA 的第一版本规范定义了基础对象适配器 (Basic Object Adapter)，并且建议其他的适配器如数据库适配器 (DataBase Object Adapter) 实时适配器 (Realtime Object Adapter) 能够被开发出来。但是这个目标一直没有达到，这是因为 BOA 的定义并不够规范和完整，于是 CORBA 的开发商为了得到一个完整的系统增加了许多自己的功能在 BOA 中。经过这样，CORBA 开发商决定新增功能不再通过定义新的对象适配器来实现，而是不断丰富 BOA 的他们自己的 API。这样就影响了 CORBA 的源代码在不同 CORBA 产品上的移植性。

经过几年后，OMG 决定废弃 BOA，取而代之的是可移植对象适配器

Portable Object Adapter(POA)。POA 在下面几点上比 BOA 更有优势。

- 1、POA 比 BOA 具有更多的内建函数，这样就减少了 CORBA 开发商增加自己的函数的需求。因此这样也减少了开发人员使用非通用函数需求，从而提高了源代码的可移植性
- 2、POA 的体系结构提供了一种可扩充的方式 “open-ended” 方式来增加新功能。通过这种方式给 OMG 提供以向后兼容的方式来增加新功能。他也允许 CORBA 开发商在保证兼容现存 POA API 的方式下新增自己的 API。

5.5 可移植对象适配器 Portable Object Adapter (POA)

POA 是 Servant 的集合。这个集合的概念和数组、链表、哈希表、具有同样的语义。POA 的创建伴随着一套相关的策略 Policy(服务质量 quality of service)。这些 POA 的策略作用于 POA 内的所有 Servant。例如：某个 POA 可以支持多线程，或支持单线程。如果 POA 支持多线程，那么这个 POA 可以以并发的方式派发请求给其包含的 Servant。如果 POA 是一个单线程策略，那么 POA 将以串行独占的方法派发请求到其包含的 Servant。

CORBA 服务器可以包含多个 POA，因为不同 POA 有不同策略，这些策略作用于此 POA 包含的所有 Servant，这就意味着，你可以作用不同到策略到不到的 Servant 集合。例如一些 Servant 以线程安全的方式实现，你就可以将这些 Servant 放入多线程的策略的 POA 中。相反的，如果一些 Servant 以单线程方式实现，则将这些 Servant 放入到单线程策略的 POA 中。

CORBA 规范并没有严格要求一个应用程序中 POA 的数量，也没有要求 POA 里面 Servant 的数量限制，你可以将你所有的 Servant 都放到一个 POA 中，极端情况，你也可以为每一个 Servant 准备一个 POA。实际中的做法是一个 IDL

文件使用一个 POA。这样做就提供了你对不同 IDL 文件使用不同 POA 策略的灵活性。每一个 POA 都可以附上一个任意的名字。如果你为每一个 IDL 文件使用同一个 POA，一个通常的做法是将 POA 的名字命名成 IDL 文件的名字。

5.5.1、 POA的层次

在层次图中，连接各个节点的连线代表的是一种关系。例如在公司组织结构图里面线表示的隶属关系，同样在面向对象中线条也是表示对象间的关系。

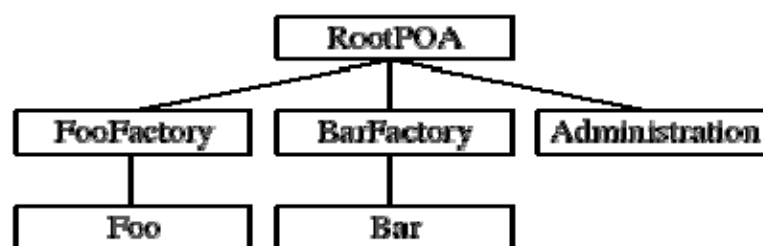


图 5.1 POA 层次的例子

在CORBA中的所有POA都具有层次关系。CORBA层次中的线条表示的是一种POA对象销毁的关系。例如考虑如 [图 5.1](#) 的一个CORBA层次图。CORBA运行系统提供一个叫做RootPOA的内建POA。正如同这个POA的名字所示，这个POA代表的是POA层次关系的根节点。销毁的次序保证了在服务器关闭的时候，这些POA按照如下的顺序进行销毁：

- FooPOA 将在 FooFactory POA 前销毁，同样，Bar POA 将在 BarFactory POA 前销毁
- FooFactory POA , BarFactory POA ,Administration POA 将在 RootPOA 前被销毁。

同级 POA 的销毁次序并不保证。即 Foo POA 和 Boo POA 的销毁顺序并不保证。同样 FooFactory BarFactory Administration 的销毁顺序也不保证。

对象销毁次序的用途是将 Servant 放入 POA 里,当 POA 被销毁的同时 POA 包含的 Servant 也被销毁。本质上说,就是程序员可以使用 POA 层次关系来确定 Servant 的销毁顺序。

POA 的层次关系趋向于简单,这是因为第一、一个 CORBA 服务一般只实现很少的接口。因为一个 POA 定义一个 IDL 是很自然的事情,这就意味着在一个 Server 进程只会存在很少的 POA,第二 CORBA 服务器趋向于比较简单的销毁顺序要求,这反映在 CORBA 的层次上非常平坦的一个结构。

5.6 不同类型的POA

[5.5 小节](#)提到POA是Servant的集合,事实上,一个POA必定是很多类型集合的一种。这就可以让程序员根据Servant的生命周期或Servant和其所代表的对象的关系这两者进行权衡,例如:

- Servant 是否一定是请求到来前就实例化好,还是根据请求进行实例化
- 如果 Servant 被创建,他的生命周期是否和服务器生命周期一样,或则可以被新创建的 Servant 替换,如果采用替换的方法,是否要建立一个缓存来保存最近访问的 Servant。
- 一个 Servant 代表一个 CORBA 对象呢,还是一个 Servant 代表多个 CORBA 对象。

上面罗列的几点允许程序员选用不同的技术来满足不同的性能和扩展性需求。

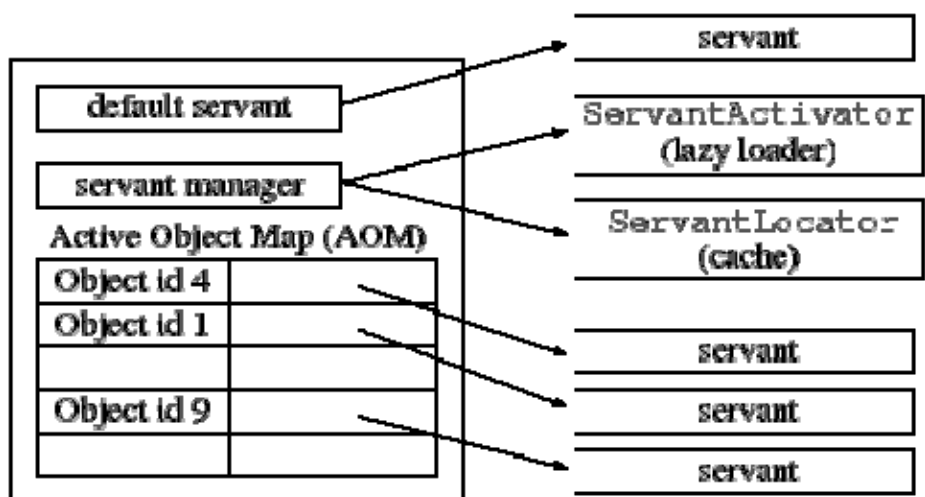


图 5.2 POA 里可能包含的组件

为了满足这种可能性，POA必须是一个灵活的容器。图 5.2 展示POA的逻辑结构。这个图说明了，一个POA可能含有一个默认Servant，一个或两个 Servant 管理器 Servant Manager (ServantActivator 或 ServantLocator)，一个活动对象映射表Active Object Map (AOM)。这些组件的用途将会在后面做说明。POA并不可能同时拥有这些所有的组件，而是POA最多只能同时有两个组件。选择什么样的组件是根据POA所要具有的扩展性和性能要求相关。下面将详细讨论这四常用的POA类型，每一中类型都是这些组件一种联合使用的用法。

5.6.1、 POA类型 1-简单POA类型

简单型POA，就是POA中只包含一个活动对象映射表。图 5.3 展示了这种类型POA，术语活动对象映射表并不是很准确的描述。映射表的含义是一张名字值对的查找表，活动对象含义是CORBA对象是处于“活动状态”，并且有一个 Servant和其关联。

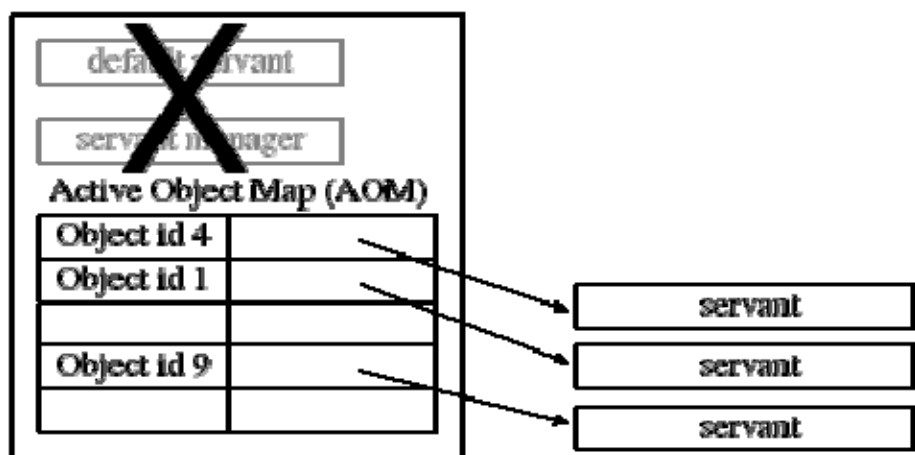


图 5.3 简单 POA 类型

IOR([第 10 章](#)) 含有客户端和服务端 CORBA 对象通讯的“联系细节”。这个联系信息中包含了主机的地址、端口号和唯一的标示服务器进程中的目标对象的对象键值。因为服务器中存在着很多的对象，所以对象键值是必须的。不同的 CORBA 产品有不同的键值表示方式，但是键值必须包含一下的信息：

- 包含 Servant 所代表对象的 POA 的层次全路径。
- 对象标示 Object id 唯一的标示 POA 的 Servant，object id 可以用任意的二进制数据表示，并可以保存在数据库中，例如以数据库的主键来存储 object id。这样就提供了一种非常方便的方式将 CORBA 对象“包装”在数据库中
- 如果 POA 采用了持久 PERSISTENT 策略 ([6.1.3 小节](#)) 并且 CORBA 服务器通过实现库 *Implementation Repository* (IMR) 来部署，那么对象键值必须包含 CORBA 服务在实现库中的标示。具体的原因在 [7.2.2 小节](#) 讨论。
- 如果 POA 采用临时 TRANSIENT 策略 ([6.1.3 小节](#))，那么对象键值必须包含时间戳。

当一个请求到达服务器时，请求的头包含了目标对象的对象键值。服务端的 CORBA 运行系统将解开对象键值中的 POA 名字和对象标示 object id 信息。CORBA 运行系统通过对象标示在该 POA 的活动对象映射表中进行查找。如果对

象标示对应 Servant 存在，那么请求就被派发到该 Servant 上。否则服务器向客户端抛出一个 OBJECT_NOT_EXIST 异常。

满足如下条件你就可以使用简单类型 POA:

- 你必须有足够的内存来同时保存所有的 Servant，一个好的 CORBA 实现中，每一个 Servant 应该占用低于 30 字节的内存，但是用户级的实例变量却会占用更多的内存，一般来说，简单类型的 POA 用来存储 10~100 个 Servant。由于内存的原因，简单 POA 类型不可能用来做百万个以上 Servant 的 POA。
- 你可以在请求到来之前预先创建 CORBA 对象代表的 Servant

5.6.2、 POA类型 2-惰性POA类型

在这种模型中(图 5.4)，服务端程序员创建POA对象，然后联合了一个实现了ServantActivator接口的对象。SevantActivator这个名字并不是很贴切，或许惰性装载更贴切，因为它的目标就是实现惰性装载。ServantActivator这个类必须由程序员来实现。

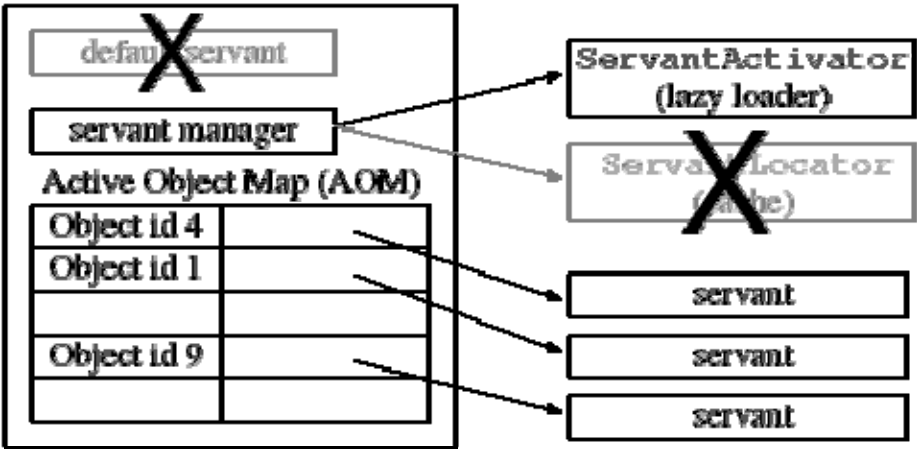


图 5.4 惰性装载 POA

当请求到达服务，CORBA 运行系统从对象键值中解开 POA 名字和对象标示

object id 信息，然后在 POA 中的活动信息映射表找查找 object id 对应的 Servant，如果该 Servant 存在，则请求派发到该 Servant。如果 Servant 不存在，POA 调用惰性装载机 ServantActivator 的 incarnate() 方法创建指定的 Servant，并将该 Servant 加入到活动对象映射表中。如果惰性装载机 ServantActivator 无法创建 Servant，服务器向客户端抛出一个 OBJECT_NOT_EXIST 异常

满足如下条件你就可以使用简单惰性 POA 类型

- 你必须有足够的内存来同时保存所有的 Servant，一个好的 CORBA 实现中，每一个 Servant 应该占用低于 30 字节的内存，但是用户级的实例变量却会占用更多的内存。一般来说，惰性类型的 POA 能用来存储 10~100 个 Servant。由于内存的原因，惰性 POA 类型不可能用来做百万个以上 Servant 的 POA。
- 你不希望在请求到来前就创建 Servant，你更倾向于根据需求来创建 Servant，选择这样做法基于如下的原因
 - 1、每一个 Servant 的初始化可能需要几秒的时间，如果你预创建 10 到 20 个这样的 Servant 将需要几分钟的时间来完成。而你不愿将等待时间花费在服务器的启动上，而更愿将等待时间花费在第一次请求调用上。
 - 2、CORBA 系统中有很多的对象，但是其中只有很少的对象被请求调用到。在这种情况下创建“按需” Servant 能有效的节省服务器的内存。

5.6.3、 POA类型 3-缓存POA类型

这种模型如 [图 5.5](#)，服务器端的程序员创建 POA 联合了一个实现 ServantLocator 接口的对象。ServantLocator 这个名字并不是很贴切；做

为实现了缓存 Servant 的 POA，使用缓存 (cache) 名字更为贴切。

ServantLocator 这个类的具体实现需要程序员来完成。

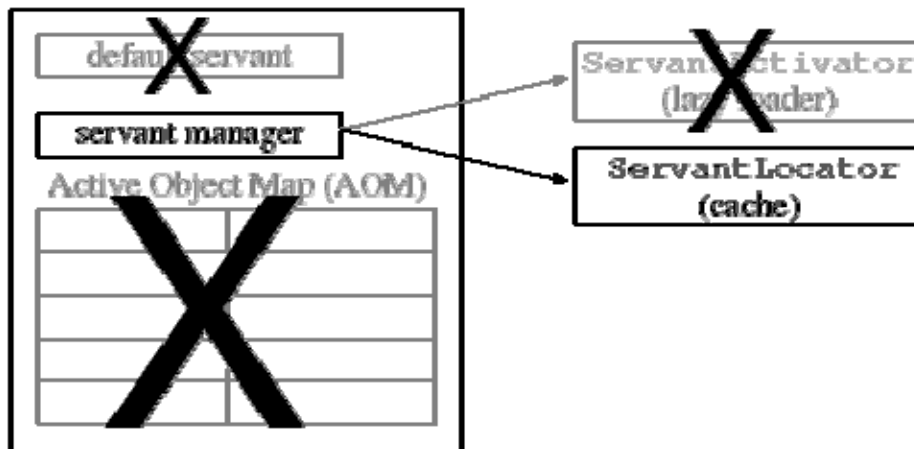


图 5.5 缓存类型 POA

当请求到达服务器，CORBA 将请求中的对象键值 object key 和控制传递给指定的 POA，POA 使用如下的伪码算法来分发请求。

```
sv = cache.preinvoke(object_id,...);
sv.operation(...);
cache.postinvoke(..., sv, ...);
```

这个 POA 并不维护自己活动对象映射表 AOM, 这是因为 CORBA 假设了 cache 的实现者实现了自己的类似 AOM 的数据结构，POA 没有必要来重复这个功能。

如果你有足够的内存存储一些但不是所有的 Servant，你就可以使“缓存”POA。一般 CORBA 服务器是一个数据库的前端，为了优化对数据库的访问，你可以选择将数据库的信息缓存到内存中 Servant 对象。当然将信息缓存到内存也存在着风险。一般而言如果你更新内存中的信息而不立即更新数据库的话，你将会面对两个风险。首先，如果另外一个程序直接访问数据库的话，他看到的信息将和你的 Servant 里面存放的信息不一致。换句话说，你有一个缓存数据一致性的问题。第二，如果你的 CORBA 服务器在将信息回写到数据库中之前意外终止，那么你将丢失你的数据。但是如果将缓存型 POA 用于只读型的数据话，

这类 POA 将会工作的非常好。

5.6.4、 POA类型 4-默认POA类型

在这种 POA 模型 (图 5.6) 中，服务器程序员创建的 POA 只关联一个 Servant。这个 Servant 被称为默认 Servant。当请求到达服务器时的 POA 时，此 POA 分发请求到默认的 Servant。这个默认的 Servant 调用 POACurrent 对象上 `get_object_id()` 方法查找当前请求的目标 CORBA 对象。可以这样说，这个默认 Servant 通过使用 POACurrent 取对象标示符然后去数据库表中提取相关信息。

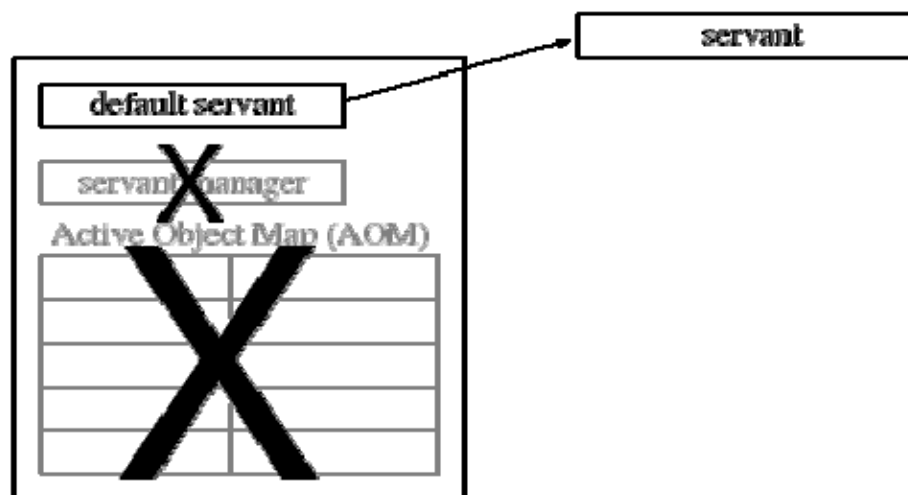


图 5.6 默认 POA 类型

默认 POA 类型最小化了内存的需求，因为这种模型使用了一个 Servant 来服务无限多 (理论上说) 对 CORBA 对象的请求。最小内存消耗是以时间为代价的，因为当每一次请求到达时，默认的 Servant 都会通过 POACurrent 获得对象标示去访问持久化数据，换句话说，默认 POA 类型不会为了快速访问为将 Servant 缓存在内存中。

5.6.5、 其他的POA类型。

从 [5.6.1 小节](#) 到 [5.6.5 小节](#) 罗列了最普通最常用的 4 中 POA 类型。你也可以通过组合特定的 POA Policies 做 `create_POA()` 方法的参数创建某个类型的 POA。通过一些合法的组合 POA 策略，可以得到一些其他类型的 POA，但是这些 POA 类型并不总见得有效。例如一个默认 POA 类型带有一个活动对象映射表。当一个请求到达时，会发生：

- 如果目标对象在 AOM 中那么请求被派发到该 Servant，否则请求被派发到默认 POA
- 程序员很少会考虑这样使用 POA，更清晰的做法是将这个 POA 分为两个 POA，默认的 Servant 类型 POA 和简单类型 POA

5.7 POA 管理器 POA Managers

水龙头的作用是开关水。从概念上来说，水龙头有两种状态：“开”允许水流出，“关”阻止水流出

POA 管理器 POA Manager 的功能非常类似于水龙头，除了他不是控制水流，而是控制请求的接收。POA 有 4 个状态：

HOLDING

这是一个“关”状态。接收到的请求都被队列缓存

DISCARDING

这是一个“关”状态。接收到的请求都被丢弃，并且 `CORBA::TRANSENT` 系统异常被抛出到客户端。

ACTIVE

这是一个“开”状态。接收到的请求被正常的派发到 Servant

INACTIVE

这是一个“关”状态。当系统关闭时，POA Manager 自动进入此状态。接收到请求按 CORBA 厂商自己方式拒绝。

POA 管理器这个名字并不是很贴切，更好的名称应该为：POA 请求阀门 POA request valve。

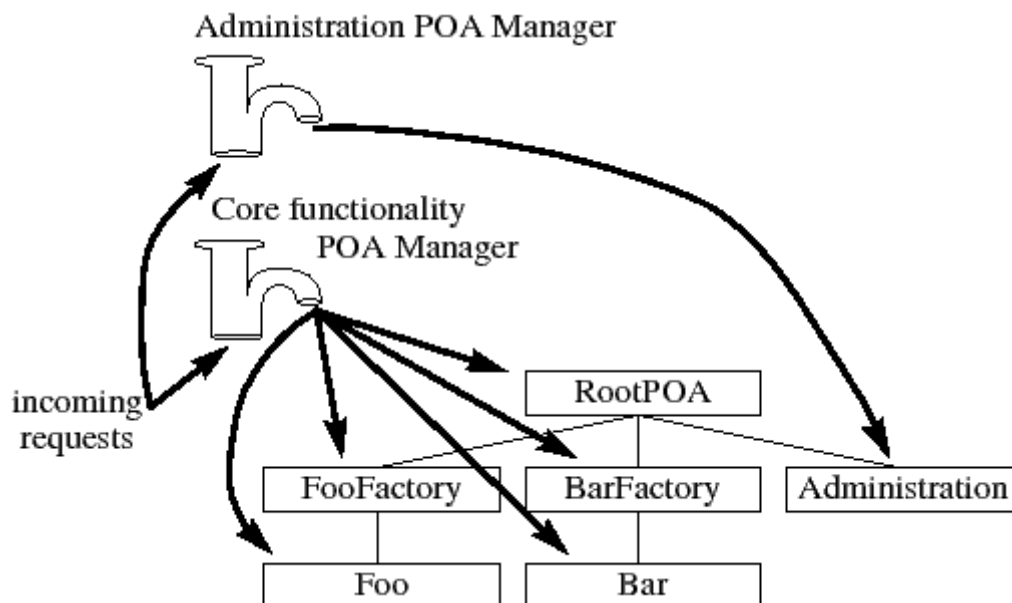


图 5.7 POA 管理器例子

POA 管理器和 POA 的关系是一对多的关系，一个 POA 管理器负责派发到过个 POA 里面所有的 Servant 的控制。一个应用程序如 [图 5.7](#) 那样安排两个 POA 管理器是非常好的做法。一个管理器控制到“管理”功能的请求，一个管理器控制到“核心”功能的请求控制。以这种方式，服务器就可以在一直打开“管理”功能的前提下，启用或禁用核心功能。

```
module PortableServer {  
    ...  
    local interface POAManager {  
        enum State {HOLDING, ACTIVE, DISCARDING,  
                    INACTIVE};  
        State get_state();  
        void activate() raises(...);  
    }  
}
```

```
void hold_requests(...) raises(...);  
void discard_requests(...) raises(...);  
void deactivate(...) raises(...);  
};  
};
```

图 5.8 POA 管理器的接口

POA管理的IDL文件定义如 [图 5.8](#)。你可以通过get_state查询POA管理的状态，也可以通过 activate()，hold_requests()，discard_requests()，activate()等方法切换POA管理的状态。POA管理的用途是在服务器初始化的时候能够阻止请求派发，一旦程序初始化完毕，服务器程序可以通过调用其所有POA管理的activate()方法切换到激活状态，并进入事件循环。

第6章、POA策略(POA Policies)

- [有效的POA策略](#)
- [创建策略对象和POA](#)

[第 5.6 节](#)讨论几种POA的类型，其中包括“简单”，“惰性”，“缓存”，“默认”。然而那一节中并没有讨论怎么创建这些类型的POA。怎么创建这些类型的POA将是本章讨论的主题。

CORBA使用policy来代表服务质量quality of service (QoS)。POA通过方法create_POA创建。这个方法的中的参数就是一个策略对象可变数组。

[第 6.1 节](#)将讨论不同策略对象和怎样通过组合这些子类型达到创建不同类型的POA。[6.2 节](#)将简要介绍一下怎样使用创建策略对象的函数

6.1 有效的POA策略

一些POA的策略对象用决定创建什么类型的POA; 这些策略在 [6.1.1 小节](#) 讨论, 其他的一些POA策略决定POA所提供的服务质量(QoS)。这些策略在 [6.1.2](#) - [6.1.6](#) 讨论

6.1.1、 决定POA类型的策略

策略对象是一个关于一些枚举值的封装器。本节中, 我将讨论着这些枚举值, 如何通过这些枚举类型创建这个封装器我将在 [6.2](#) 小节讨论。

下面列出了 3 类枚举类型, 通过将枚举类型组合在一起决定不同的 POA 类型:

```
enum ServantRetentionPolicyValue {RETAIN, NON_RETAIN};
enum IdUniquenessPolicyValue {UNIQUE_ID, MULTIPLE_ID};
enum RequestProcessingPolicyValue
    {USE_ACTIVE_OBJECT_MAP_ONLY,
     USE_DEFAULT_SERVANT,
     USE_SERVANT_MANAGER};
```

第一个枚举类型, 决定是否可以通过活动对象映射表 active object map (AOM)提取对象标示 object id 。许多人都很难记住这个枚举值。一个更直观的名字是 HAS_AN_AOM 和 DOES_NOT_HAVE_AN_AOM 而不应该是 RETAIN 或 NON_RETAIN

第二个枚举类型决定对象标示和 servant 之间是否是一对一的 (UNIQUE_ID)映射或则多对一(MULTIPLE_ID)。MULTIPLE_ID 策略允许一个 Servant 代表多个 CORBA 对象。你必须使用 MULTIPLE_ID 类型创建默认类型 POA 类型。但是你也可以用 MULTIPLE_ID 创建其他类型的 POA, 但是通常使用的更多的 UNIQUE_ID 策略。

第三个枚举值决定是否 POA 中包含 AOM, 是否包含默认 Servant, 是否包

含 Servant 管理器 Servant manager

- `USE_ACTIVE_OBJECT_MAP_ONLY` 枚举值是用来创建简单 POA 类型 ([5.6.1](#) 小节)。你必须联合 `RETAIN` 枚举值，通常还必须联合 `UNIQUE_ID`，但这不是必须的。
- `USE_DEFAULT_SERVANT` 枚举值是用来创建默认 POA 类型 ([5.6.4](#)) 你必须联合 `MUTIPLE_ID` 枚举值。同时你也可以联合 `NON_RETAIN`，但这并不是必须的。
- 如果你联合 `USE_SERVANT_MANAGER` 和 `RETAIN` 枚举值，你将得到一个惰性 POA 类型 ([5.6.2](#) 小节)。如果你联合 `NON_RETAIN` 你将得到一个缓存类型 ([5.6.3](#) 小节)。 `USE_SERVANT_MANGER` 策略一般都要联合 `UNIQUE_ID`，但是着并不是必须

6.1.2、 单线程多线程策略枚举值

下面的枚举值指定怎样让你个 POA 工具化线程分发请求方式

```
enum ThreadPolicyValue {ORB_CTRL_MODEL,
                        SINGLE_THREAD_MODEL,
                        MAIN_THREAD_MODEL};
```

这些策略值将在下面的字小节中讨论:

6.1.2.1、 ORB_CTRL_MODEL 策略值

`ORB_CTRL_MODEL` 策略值指定由 CORBA 的运行系统的实现来控制怎样分发接入的请求，这个策略存在着很多非规范的语义

- 很多的 CORBA 产品使用这个策略值表示多线程分发策略请求，但是多线程分发请求的细节在不同的 CORBA 实现中却各不相同，例如：某一个 CORBA 产品可能会采用 (1) 通过线程池来分发收到的请求 (2) 使用每收到一个请求

就创建线程的方式来分发请求(3)使用线程分离方式,一个 socket 连接一个线程的方式来分发请求(4)一些其他技术策略

- MICO 的 CORBA 中间件原来实现并不支持多线程方式,在 MICO 中 ORB_CTRL_MODEL 被用来使用单线程分发请求,并且这被认为是一种兼容的实现方式。多线程(或多线程分发算法)支持因此被引入至 MICO 中。

ORB_CTRL_MODEL 非规范的语义影响了 CORBA 应用程序的移植性。许多的人们都希望 OMG 能够在未来增加多线程支持的策略值。

6.1.2.2、 SINGLE_THREAD_MODEL和MAIN_THREAD_MODEL 策略值

你也许可能认为 SINGLE_THREAD_MODEL 有非常明确的语义,但是,这个它是含有二义性的。

- 许多 CORBA 产品解释这个策略值为所有请求在分发到进程的主线程中运行,在这种解释下,相当于进程内所有的请求都被串行化。
- 其他的 CORBA 产品解释这个策略值串行化在多个 SINGLE_THREAD_MODEL POA 中,即在 POA 中串行化。

当 OMG 开始意识 SINGLE_THREAD_MODEL 的二义性后,他们决定将 SINGLE_THREAD_MODEL 声明为第二中语义,而新增了 MAIN_THREAD_MODEL 为第一种语义。

读者必须注意到,现在的一些 CORBA 产品错误的使用了 SINGLE_THREAD_MODEL 语义,并且没有及时的增加新 MAIN_THREAD_MODEL 语义支持。因为这个原因,读者在使用 SINGLE_THREAD_MODEL 语义前必须仔细阅读相关 CORBA 产品关于 SINGLE_THREAD_MODEL 语义的说明。

6.1.3、 对象生存周期和名字策略值

下面的几个枚举值用来指定对象的生命周期和他们怎样为对象标示赋值。

```
enum LifespanPolicyValue {TRANSIENT, PERSISTENT};  
enum IdAssignmentPolicyValue {USER_ID, SYSTEM_ID};
```

CORBA 使用术语临时 *transient* 来代表临时易变的意思。然而，TRANSIENT 策略值用来指定对象引用是仅仅在服务器进程中有效。换句话说，对象引用在服务器进程重启后就无效了。CORBA 产品怎样实现这样的功能是一个技术实现细节问题，典型的做法是将时间戳信息放到 IOR 中。CORBA 运行系统能通过时间戳信息来区分在不同进程同时创建的的对象引用。

PERSISTENT 策略指定了 POA 中的对象引用无论服务器进程是否重启过都一样有效。

了解 TRANSIENT 和 PERSISTENT 这两个策略值是决定对象引用的生存周期是非常重要的。这些策略值并不决定与对象相关的数据是在内存中维护还是通过文件、数据库这些持久化介质来维护。下面的例子说明了怎么使用这些策略值：

- 一些 CORBA 对象，比如说用户和账号，这些数据都一般都存储在数据库中，将这些对象放入 PERSISTENT POA 是非常有意义的，因此客户端引用到此对象或数据时，无论服务器进程是否重启，都将有效。
- 一些服务器的对象，例如管理对象，工厂对象，并不和数据有关系，但是这些无状态 *stateless* 对象也是保存在 PERSISTENT POA 中，这样就确保了客户端访问这些对象，无论服务器进程是否重启，都将有效。
- 一些服务进程中的临时性对象，例如名字服务中用于遍历值集合的 `CosNaming::BindIterator`。另一个例子如 `LogSession` 用户保存用户登陆信息的对象。虽然这些对象是有状态的，但他们的状态一般都在内存中而不是数据中维护。因为这些对象临时的特点，所以当服务器重启时，并没有继续使这些对象在客户端中有效的需求。事实上，在服务器重启时这些对象失效是更好的做法。因为这些原因，这些对象应该存储在 TRANSIENT

POA 中。

在 POA 的术语中，程序员被认为是用户。USER_ID 策略值比表明了对象的对象标示 object id 是由程序在对象激活时(插入到 POA)赋值。程序员使用 activate_object_with_id()方法。就如同如下：

```
poa.activate_object_with_id(obj_Id,sv);
```

SYSTEM_ID 策略表明了当对象被激活(插入)到某个 POA 时由 CORBA 运行系统指定对象的对象标示 object id，在这种情况下，程序员使用 activate_object()方法，就如同如下：

```
obj_id = poa.activate_object()
```

从技术上来说，TRANSIENT 和 PERSISTENT 策略独立于 USER_ID 和 SYSTEM_ID 策略，然而由于以下原因，PERSISTENT 总是联合 USER_ID 出现。

- 如果程序员希望在数据库中维护对象的状态，那么他或她会使用数据库表中的主键作为对象标示。这样就为对象和数据库中的数据提供了一种简单的映射方式
- 一个无状态单件(singleton)——例如管理对象和工厂对象的 POA 一般都是 USER_ID 策略，才能保证每一服务器重启都是同一个对象被用到。这种方式下，客户端的引用将在服务器进程重启后有效。

TRANSIENT 策略一般联合 SYSTEM_ID 使用，因为程序员很少为临时的对象赋有意义的键值。

6.1.4、 事务对象策略值

一些C/S系统要求数据库事务具有跨多个客户端到服务器端操作调用，或跨多个数据库的能力。这样的C/S系统要求使用CORBA对象事务服务(OTS)，关于OTS的细节将在 [第 21 章](#)讨论。

这类策略值决定该 POA 内的对象是否参与分布事务。这个策略值可以取如下

的值:

REQUIRES

这个值表明该 POA 内对象的所有方法都是事务的一部分。

FORBIDS

这个值表明该 POA 内对象的所有方法都不是事务的一部分。

ADAPTS

这个值表明 POA 内的对象能适配两类(事务、非事务)的调用。

6.1.5、 明确与隐含激活对象策略值

大多数的策略值都具有影响客户端行为或服务器端高端体系的能力。但是 ImplicitActivationPolicyValue 和这些策略值不同,他只是轻微的影响服务器端编程方式。

```
enum ImplicitActivationPolicyValue
    {IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION};
```

让一个 Servant 代表 CORBA 对象有三种方式:

- 1.创建 Servant
- 2.激活或插入 Servant 至 POA
- 3.调用 _this() 方法获得 Servant 代表 CORBA 对象的应用。

如果你使用 NO_IMPLICIT_ACTIVATION 策略,那么你必须执行这三步,然而如果你使用 IMPLICIT_ACTIVATION,你就可以可选的省略掉 2 步而隐含的激活一个 Servant。所以 IMPLICIT_ACTIVATION 的好处就是可以优化服务器端对象创建的一行代码。这样好处相对来说是非常有限的,并且一些人都非常固执的认为 IMPLICIT_ACTIVATION 是一个好策略,或另一些人也固执认为 IMPLICIT_ACTIVATION 应该避免使用。一方面一些人们非常喜欢在服务器端的代码中很多地方被优化掉一行,而另一方面,一个又一个带带有争议的策略值

带来学习曲线的增加超过了它本身带来的好处。

6.1.6、 私有策略值

上述小节中讨论了 CORBA 系统中用来创建 POA 的兼容的策略值，CORBA 产品被允许定义额外的，私有的策略值。一些 CORBA 产品这样做是为了给程序员提供更多可选择的服务质量(QoS)。因此，你必须仔细阅读相关 CORBA 产品提供的文档是否提供了私有策略。

6.2 创建策略对象和POA

策略对象首次是同 POA 规范一期引入。POA 的规范最初定义了 7 中类型的策略对象和 7 中不同的方法来创建这些策略对象。例如，你能通过如下方法在某个存在的 POA 上创建一个 ThreadPolicy 策略对象

```
ThreadPolicy create_thread_policy(  
    in ThreadPolicyValue value);
```

这个方法的参数是一个枚举值，这个方法创建一个策略值的"封装器 wrapper"。

通过定义 POA 规范，OMG 发现策略值的概念可以应用到 CORBA 对象的其他部分中。然而为每一个新策略在当前接口中创建一个新方法的做法会带来版本的问题。因为这样 OMG 决定定义一个"通用 Generic"的 API 来创建任意类型的策略值。这个通用 API 是 CORBA Message 的一部分。事务相关的策略值创建就是使用这种新的方式。

总的来说，创建 POA 策略值和使用这些策略值创建 POA 的 CORBA API 是相当繁琐的。在《CORBA Utilites》中 POA 层次创建变得简单章节中讨论了一个工具类，这个工具类梦幻般的减少了很多创建 POA 的代码量。

第三部分 应用程序部署

第7章、 实 现 库 Implementation Repository (IMR)

- [IMR介绍](#)
- [IMR概念](#)
- [实现库例子](#)
- [不同的IMR对比](#)

7.1 IMR介绍

OMG 在 CORBA 的规范中简单的描述了 IMR 的概念。但是 IMR 这个术语并不是很直观，因此解释一下 IMR 是有好处的。“实现 *implementation*” 这个术语在 CORBA 规范中的意思是服务端应用，“库 *repository*” 代表持久化存储，比如数据库，文件。根据字面意思 *Implementation Repository* 就是数据库或文件存储服务端应用的意思。这样解释基本正确。IMR 通常上包含了能让 CORBA 应用程序和 IMR 进行通讯的 CORBA 服务器的数据库封装器。

IMR 一般为每一个服务应用程序维护如下的信息：

- 唯一标示 CORBA 服务器的逻辑名称，比如 “BankSvr” 或者 “StockControlSvr”
- 一个启动或重启 CORBA 服务器的命令行参数。
- CORBA 服务器的是否启动的状态信息。如果 CORBA 服务器已经启动，则在 IMR 中记录 CORBA 服务器的机器名和监听端口号。

CORBA 规范只提供了 IMR 的部分定义。OMG 只定义了 IMR 的高层次的功能

需求，并没有明确如何实现 IMR。同样也没有定义了 IMR 该如何进行管理。这样定义的原因是 IMR 的功能和管理功能的实现是具体的平台相关的。这是因为：

- IMR 提供了启动停止 CORBA 服务器的功能，但是不同的操作系统起停进程的操作不同。
- IMR 会记录服务器的细节，例如：如何启动服务器，CORBA 服务器当前的状态是什么。一些 IMR 会将信息记录到数据库中，另一些会将信息记录到文本文件中。还有一些嵌入式设备因为无法访问数据库和文件系统，只能将信息记录到内存(ROM)中。

在主机系统上跑的 IMR 和运行在 PC 上 IMR 不仅实现不一样，而且其管理的方式也不相同。某一个开发商提供的运行在某一个操作系统上的 IMR 使用的“观感”和另外一个开发商在另外一个操作系统上提供的 IMR 的“观感”也不一样。这种 IMR 不同，是 CORBA 规范只定义 IMR 高层次功能需求的原因。

[7.2 小节](#)通过基于不同中的CORBA实现方式的IMR例子。[7.3 小节](#)讲解了三种不同CORBA产品的IMR实现。以这种方式，我将讲解不同产品IMR细节是怎样的不同，虽然不同产品的IMR有细节上的不同，但是IMR基本概念是一样的。

7.2 IMR概念

7.2.1、 在IMR中注册服务器

IMR 实现功能是，通过数据库或文件系统持久化注册在 IMR 中的服务器详细信息封装器。CORBA 的产品都实现了提供命令行工具在 IMR 中注册服务器。这样的工具可能以如下的方式被使用 ('\'是指续行符)

```
reg_srv_with_imr BankSrv \  
-launch "/bin/bank_srv -ORBServerId BankSrv ..."
```

这个工具需要得到一些服务器信息，比如服务器的名字(如上例所示的"BankSvr")，启动命令，与IMR的通讯信息，如 [图 7.1](#) 所示，这些信息都会被持久化到数据库。reg_svr_with_imr工具和IMR的通讯方式是以将reg_svr_with_imr认为是一个CORBA客户端和IMR为服务器的通讯方式来实现的。IMR服务器监听在一个固定的端口号上。(如 [图 7.1](#) 所示的 4000 端口)。然而正如 [7.1 小节](#)提到过的IMR的实现细节不一样的原因，reg_svr_with_imr这个名字，它的命令行参数在不同的CORBA产品的实现上并不相同。一些CORBA产品甚至提供了图形化界面来注册服务器。

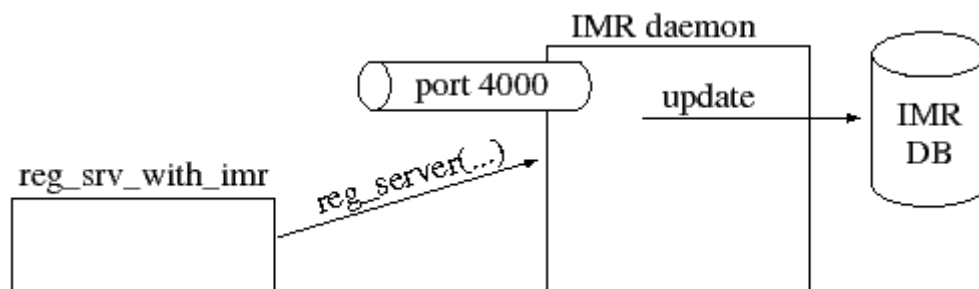


图 7.1 在 IMR 中注册服务器

reg_svr_with_imr中的启动服务器命令中包含一个-ORBServerId <name>的参数对。当IMR在后来启动服务器时，这对参数将被服务器的ORB_init()([3.2.2 小节](#))使用。并且参数也告诉IMR服务进程的唯一标示。需要注意的是，这个参数是CORBA 3.0 中的规范。较早版本的CORBA产品实现必须使用其他的参数形式来实现对服务器的唯一标示的注册。

7.2.2、 手动启动服务器

当注册完服务器后，你可以以如下的方式启动 CORBA 服务器

```
/bin/bank_srv -ORBServerId BankSrv ...
```

需要注意的是，当你手动启动一个服务器时，一旦你在 IMR 注册了服务器，你就要使用注册时的启动命令中-ORBServerId BankSvr 这个命令行参数。

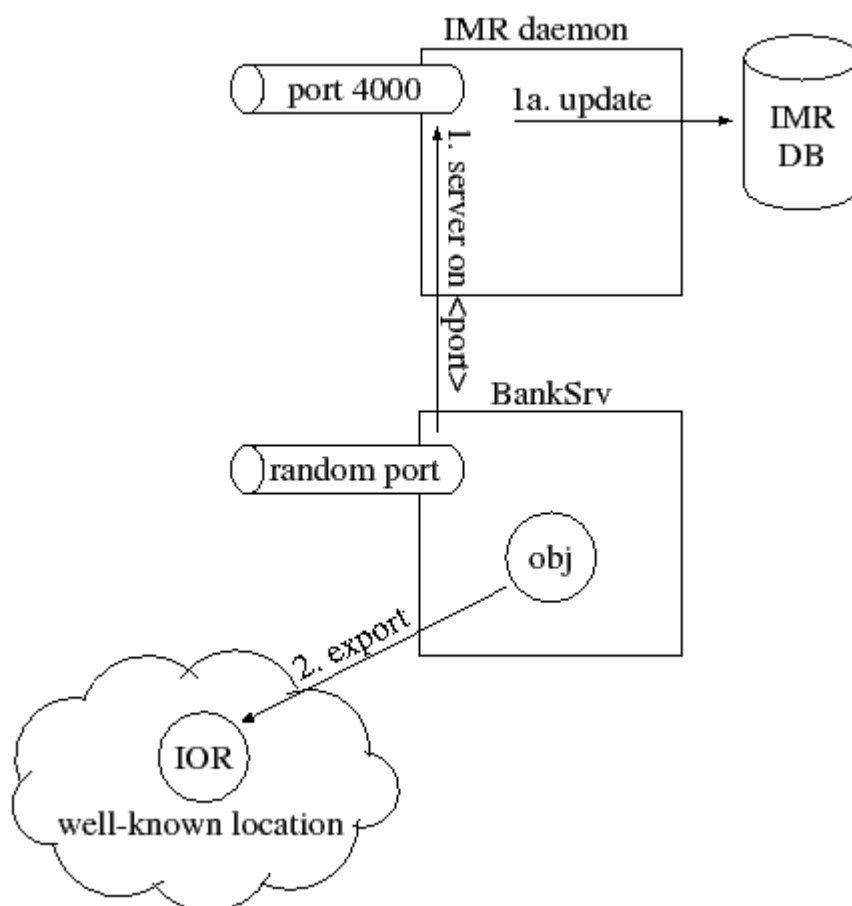


图 7.2 IMR 和手动启动 CORBA 服务器的交互

在默认情况下，CORBA 服务器监听在一个随机的端口上。CORBA 服务器初始化过程中的某一个 CORBA API，一般是 `ORB_init()` 或 `create_POA()`，来会通知 IMR “由 `ORBServerId` 指定的服务器已经启动”，并将其监听的随机端口信息告诉 IMR (上图中的 1 步)。IMR 接着更新数据库相关细节 (图中的 1a 步)。这个通讯过程是初始化 CORBA 服务器中的某一个 API 的实现细节。这就意味着服务器和 IMR 的通讯对程序员来说是透明的。

IMR 通过某一种方式知道服务器何时停止，因此 IMR 可以更新数据库中相关服务器已经停止的信息。CORBA 没有标准化 IMR 诊断服务器停止信息的技术，但是我在这里将会提一些已经被现有 CORBA 产品使用的技术。在 CORBA 产品中有一些未公开的 CORBA 运行系统对象，这些对象周期性的向 CORBA 服务器发送

"ping"信息来检查服务器是否存在。另外一些 CORBA 产品通过打开 IMR 和服务
器之间的 socket 连接或管道来判断，一旦服务器停止，操作系统自动通知
IMR 管道或 socket 连接已经关闭。

作为服务器初始化的一部份，服务器很可能在某一个约定好的地方公告对象
引用。客户端应用程序使用对象引用和服务通讯，所以在客户端试图和服务端通
讯前对于手动启动的服务器(这样就能公告出对象引用)是非常重要的。本章不
讨论对象引用的公告方式，例如以文件、名字服务、交易服务等。

对象引用中包含着对象的“联系细节”。一旦服务器通过 IMR 部署后，服务
器输出 IOR 指定如何通过 IMR 和服务进行通讯。IOR 中的主机，端口，对象
键值等信息可能是如下的形式：

host:<IMR's Host>

port:4000(IMR 的端口)

object key:"BankSvr",<poa name>,<object id>

IOR 中的对象键值唯一的标示了服务器进程中的一个对象。在服务器进程
中，对象被组织到 POA 容器中。服务进程可能包含多个 POA，因此对象键值的
内容由包含该对象 POA 的名字和该 POA 中唯一标示该对象的对象标示 object
id 组成。令人惊奇的是，在一些 CORBA 产品中的 IOR 包含了服务器的逻辑名
称，例如，BankSvr。包含服务器逻辑名称的原因是(下一节将详细讨论到)，
服务器必须知道包含目标对象的服务器是那个，知道了这个服务器才可以将客
户端的请求重新定向到该服务器。将服务器名字嵌入到 IOR 的对象键值中是联系
对象键值和服务器的一种最直接的方式。如果对象键值中不包含服务器名称，那
么 IMR 只能指定对象属于那个 POA。在这种情况下，IMR 要知道对象属于哪个
服务器，只能将所有的服务器的 POA 都注册到 IMR 中，这种方式在管理上就非
常复杂。更重要的是，这样的方式下，无法将具有同样 POA 名字的对象注册到

同一个 IMR 中，因为这样 IMR 就无法知道 Servant 归属于哪一个服务器。

7.2.3、 客户端和IMR及服务器之间的交互

[图 7.3](#) 展示了，客户端怎么和服务中对象建立通讯。客户端从某一个约定好的地方导入一个 IOR (第 1 步)。当客户端准备远程调用服务器端对象时，它首先和在 IOR 中指定的主机端口建立连接。在我们的例子中，IOR 的主机和端口是 IMR 的主机和端口而非 CORBA 服务器的端口，但是客户端进程并不了解这些。然后客户端向主机发送请求 (第 2 步)。每一个请求的头都包含 IOR 中的对象键值信息。当 IMR 收到请求，并且意识到对象的键值不匹配 IMR 中对象，IMR 就从对象键值中取的服务器名字 ("BankSvr")，并且使用这个名字从数据库查找服务器的信息 (第 2a 步)

- 如果数据库中服务器的信息显示这个服务器已经运行，那么 IMR 从数据库中获取服务器的主机和端口号，并基于服务器的主机和端口创建一个新的 IOR，然后发送一个包含此 IOR 的重定向信息给客户端 (第 3 步)。
- 如果数据库服务器信息显示这个 CORBA 服务器还未运行，那么 IMR 从数据库中获取启动服务器的命令行参数，并 (重)启动服务 (第 2b 步)。此时，IMR 将等待服务器完成自身初始化，和临时监听端口监听的通知 (第 2c 步)。IMR 接着更新数据库服务器状态为运行，并更新端口号信息。此后 IMR 将基于此 CORBA 服务器的主机和端口创建一个新的 IOR，最后 IMR 发送一个包含此 IOR 的重定向信息给客户端 (第 3 步)

重定向信息告诉告诉 CORBA 客户端的运行系统 “你所要查找的对象不在这里，但是这里有一个新的 IOR，里面有你需要的目标 CORBA 服务器的通讯 “联系细节” 信息。CORBA 客户端运行系统于是根据这份重定信息向 IOR 包含的主

机和端口建立新连接(第 4 步)。重发的逻辑代码在 CORBA 客户端运行系统里, 对于程序员来说是透明的。

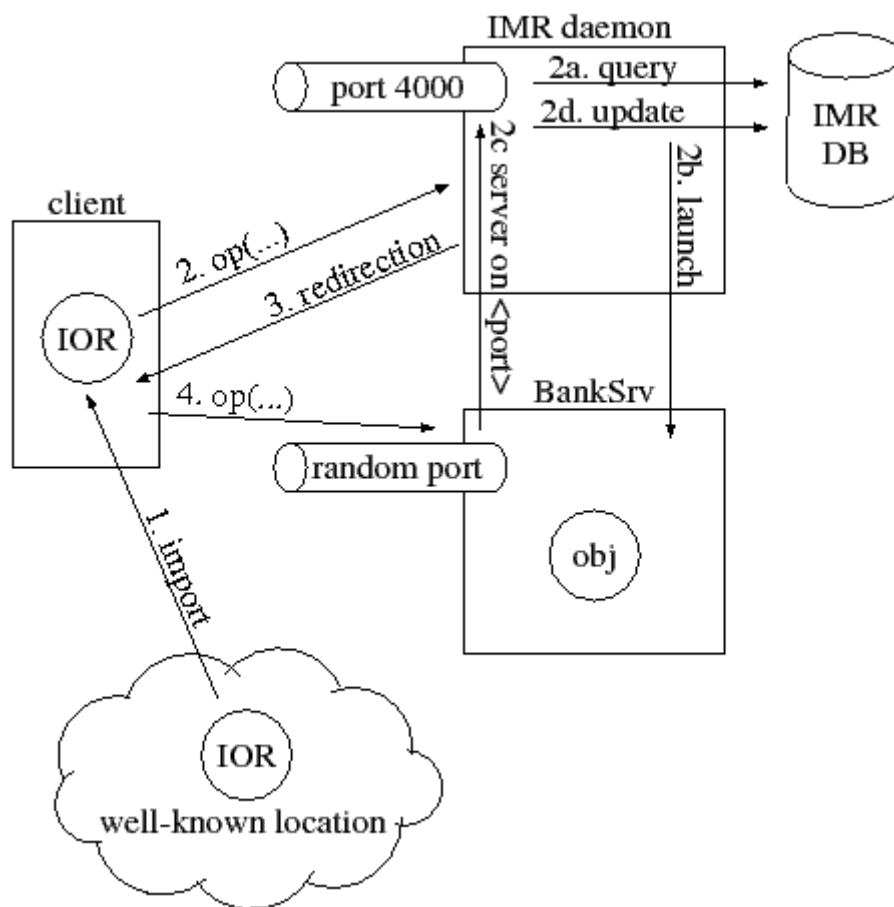


图 7.3 IMR 与客户端通讯和自动启动服务器

关于客户端初始化和 IMR 通讯到后来和真实的目标服务器通信过程中下面几点非常重要:

- 重定向只发生在第一次服务对象调用, 以后的请求将直接发送到真实服务器。正因为如此, 任何和重定向机制有关的信息只在第一次连接建立的时候发生。即使重定向只发生在连接建立的时候, 许多 CORBA 产品都有优化机制避免重复传输大量信息。
- 重定向机制意味着 CORBA 服务器不需要预先启动好。服务器注册到 IMR 中的服务器就可以"按需求"启动。同样只有 IMR 需要监听在一个可配置的固定端口上, 通过 IMR 部署的服务器能监听在随机端口上, 这点非常方便。

- 由于重定向机制是 CORBA 规范的一部分。所以使用 IMR 并不影响不同 CORBA 客户端产品和 CORBA 服务器端产品的交互。但是 IMR 和服务器的交互方式在不同的 CORBA 产品中却不相同,因此,Orbix IMR 只能启动 Orbix 服务器, TAO IMR 只能启动 TAO 服务器。
- 让我假设客户端通过 IMR 对服务器重定向,然后客户端对服务器上的对象进行了几次成功的调用。如果后来服务器终止,这就导致了客户端和服务器的连接丢失,则 CORBA 客户端的运行系统自动使用原有的 IOR 的主机和端口。这意味着客户端将把下一个请求发送到 IMR,这给 IMR 一个重启服务器,并将客户端重定向到新服务器的机会。
- 一些 CORBA 产品允许你在 IMR 中注册重复的服务器,例如,你可以注册 5 个叫 BankSvr 的服务器到 IMR 中。在这样的产品中 IMR 重定向客户端到一个服务器,其他的客户端重定向到另一个服务器等等。这就为每一个客户端提供了负载均衡的机制,而不需要程序员单独的在其客户端程序中增加负载均衡逻辑。当然,如果你是用了重复服务器,那保证服务器缓存数据一致性的工作就是你的责任。
- 服务器也可以独立于 IMR 部署。在这种情况下,输出的 IOR 包含了服务器的主机和端口号,所以客户端不需要通过 IMR 进行重定向。

IMR 启动一个服务器进程调用的是操作系统的 API,例如,Windows 上 `CreateProcess()` 和 unix 上的 `fork()` 和 `exec()`。在多数操作系统上,这些 API 只可以在同一类机器上创建新进程,不能在其它不同他机器上创建新进程。在每一个需要创建新进程的机器上,你都需要部署一个 IMR。这也是早期 CORBA 产品的一个公共特点。因为通常是要管理多个 IMR,这样就导致了这些对大型 CORBA 应用的部署的管理工作非常的棘手。

7.2.4、 分布式实现库

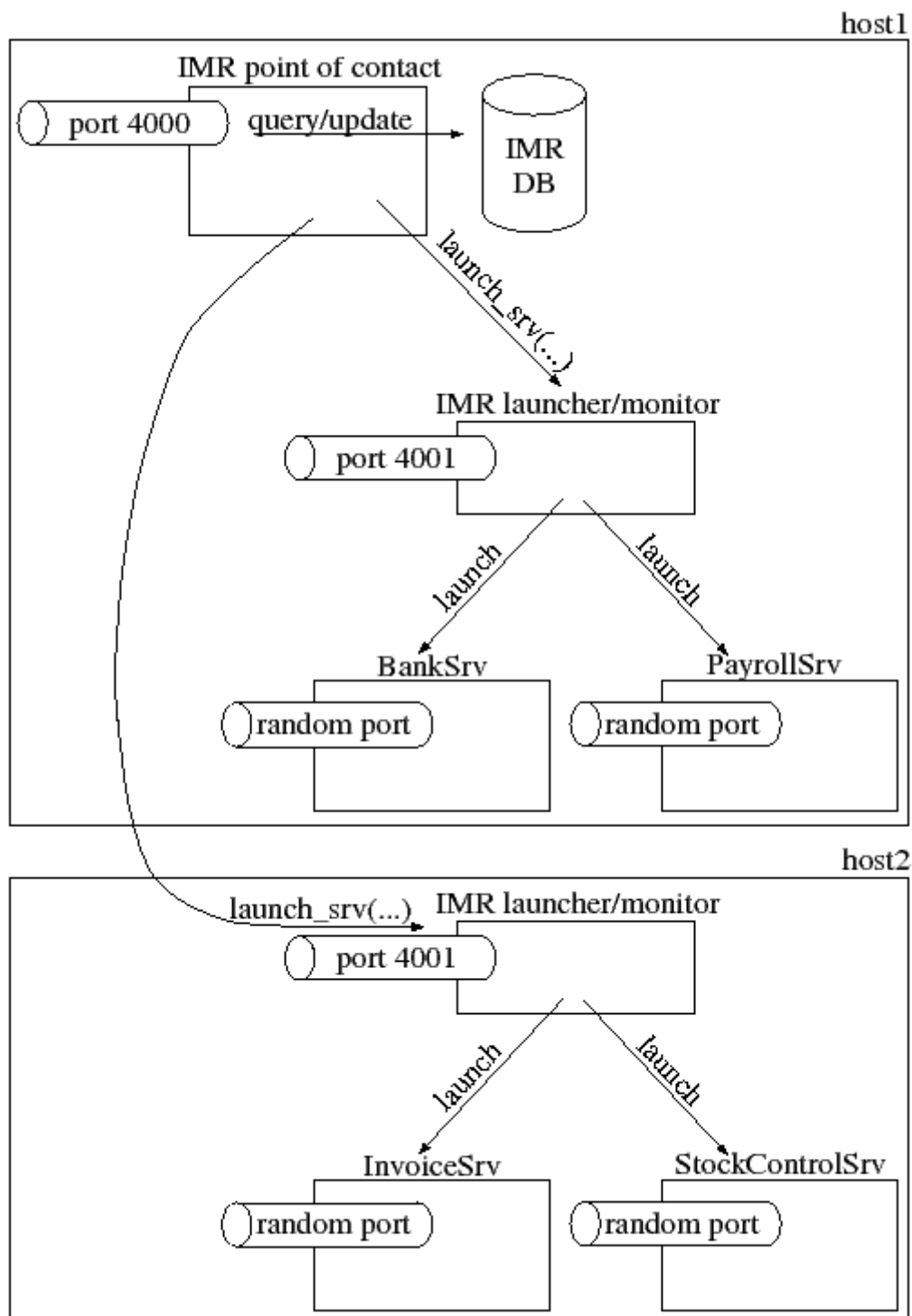


图 7.4 分布式实现库

许多流行的CORBA产品将IMR分成两个部分。如 [图 7.4](#)。进程(图中被称为IMR接入点 "IMR point of contact")监听在一个固定端口上, 例如 4000,

这个端口嵌入在通过IMR部署的服务器CORBA对象的IOR中。除了提供IOR中的“知名”端口外，这个进程还负责执行查询和更新IMR数据库的操作。当这个IMR想启动一个进程时，此IMR会将请求委托到第二个在图 7.4 中叫做IMR启动器和监视器IMR launcher/monitor的进程那里，这个进程监听在另外一个固定端口上。

将 IMR 的功能分成两个部分的目的是因为有可能存在很多个 IMR 启动器和监视器进程——典型情况是每一台机器上都有一个。这样安排就能让 IMR 跨多个机器。这样做就导致工具在 IMR 中注册服务器时需要指定服务器在哪个机器上运行，如同下面例子所示（“\”符号是续行符）

```
reg_srv_with_imr BankSrv -host pizza.acme.com \  
    -launch "/bin/bank_srv -ORBServerId BankSrv ..."
```

使用分布 IMR 的好处时可以集中数据库管理和维护在不同机器上注册 CORBA 服务器的信息。很多公司发现这种方式比每一台机器上有一个独立数据库管理更方便。

另外一个好处是，IOR 中的主机和端口信息统一的是 IMR 启动器和监视器的主机和端口，而不管是哪一个机器是运行服务器进程。例如，BankSvr 运行在一台机器上，但是这台机器需要进行停机维护，于是就可以将 BankSvr 重新注册在另外一台机器上。这样，BankSvr 就可以从一台机器移植到另外一台机器上。这样移植并不需要使之前服务器输出的 IOR 无效，因为 IOR 的主机信息一直都是 IMR 连接点进程的主机信息。

分布式 IMR 体系同样能给 CORBA 的开发商带来好处。在 IMR 中大多数的和平台相关的代码都集中在启动停止服务器进程逻辑上。这些平台相关的代码就可以封装在 IMR 启动器和监视器进程上，这样就增加了 IMR 源代码的维护性。

以上描述分布式在一些流行 CORBA 产品中的存在着差别，这些差别主要表

现在 IMR 数据库的组织方式。例如：

- IMR 连接点进程的数据库可以维护不经常变化的数据库信息，比如服务器注册信息，服务器名字，启动命令行参数。服务器在什么地方运行，服务器监听的端口号在每一个 IMR 启动器和监视器中维护。
- IMR 接入点进程也可以不维护数据库。所有此服务器的信息由运行该服务器上 IMR 启动器和监视器使用的数据库维护。

这些不同分布式 IMR 的差异只是实现上的不同，并不影响 CORBA 产品提供的服务质量。

CORBA 产品通常不对系统中多少个 IMR 或则不同的 IMR 是否放在不同或相同的服务器上做约束。一个公司选择多少个 IMR 一般都是程序上的考虑。例如，每个程序员在每天的日常开发过程中都有他们自己的私有 IMR 是非常常见的事情。一些 IMR 用来做系统测试，另外一些用来部署应用程序。公司可能会觉得有几种 IMR 部署非常方便：也许为公司里面的每一个部门，每一个分支机构使用一个 IMR，或许一个 IMR 用于薪水应用，另一个 IMR 用于股票控制应用。如果一个机器上要跑多个 IMR 的话，这些 IMR 将监听在不同的端口上。通过配置文件、环境变量、命令行参数传递不同的主机的端口给 IMR 来控制其监听的端口。

7.3 实现库例子

以下的几节将简单的讲解一下几个不同 CORBA 产品的 IMR。

7.3.1、 Orbix

Orbix 的 IMR 是一个类似于 [图 7.4](#) 的分布式 IMR

1、Orbix 为 IMR 连接点命名为定位器 “locator daemon(itlocator)” 。Orbix 把这些进程叫做定位器是因为这些进程帮助客户端定位服务器 (中的对象)。Orbix 为 IMR 启动器和监视器命名为节点 “node daemon”。这是因为在每一个通过 IMR 部署的服务器的节点 (机器) 上都有这样的一个进程。

2、itadmin 工具是一个命令行工具用于和定位器 “locator daemon” 通讯并查询和更新 IMR 数据库。itadmin 的工作方式和在本章开头讨论的理论上的 reg_svr_with_imr 工具具有一样的功能。许多 Orbix 的管理命令都是通过这个工具各种各样的子命令完成的。当一个服务器注册到 IMR 时注册信息中必将包含指定启动服务器的节点 “node daemon” 信息。

3、在 Orbix 的以前的版本中, 节点 “node daemon” 周期性的 ping 服务器检查服务是否存在。在现在的版本中, 在 CORBA 服务器进程和 “node daemon” 之间通过 socket 建立一个连接; 当 socket 连接被关闭时, 节点 “node daemon” 就知道服务器进程已经异常退出。

一旦服务器被注册到 IMR 中, 服务器可以被注册为能在很多机器上运行的重复服务器。IMR 中记录着当前运行的重复服务器信息, 并且可以重启已经终止的服务器。IMR 使用轮叫调度 “round-robin” 和随机调度策略对客户端进行调度。在这种方式下, CORBA 不需要在客户端和服务器额外的代码就可以通过 IMR 完成对每一个客户端的负债均衡。

Orbix 中的 IMR (和其他一些重要的基础模块) 都可以重复的部署在多个机器上。这样做就避免了单点故障的问题。

你可以通过运行 itconfigure 工具 (在 Orbix 管理命令手册讨论) 创建一个 Orbix 的 IMR。你应该注意到实现库 (implementation repository) 是

一个 CORBA 规范术语，而且不同的 CORBA 厂商习惯为他们自己的 IMR 起自己的名字。例如 Orbix 将 IMR 命名为定位域 “location domain”。定位域内容是包括了所有注册的服务器信息的数据库的信息，定位器 “locator daemon” 信息，和节点 node daemon 信息。

7.3.2、Orbacus

Orbacus 的 IMR 是一个类似于 [图 7.4](#) 的分布式 IMR

1、IMR 的接入点和 IMR 启动和监听器的功能都集成在可执行文件 imr 中。

在 Orbacus 中，术语 IMR 启动和监听器 “IMR launcher/monitor” 的功能由对象激活器 “object activation daemon(OAD)” 完成。默认情况下 imr 具有 IMR 的接入点和 IMR 启动和监听器的功能；-master 命令行参数指定 imr 是 IMR 接入点，-slave 命令行参数指定 IMR 是 OAD。如果你想你所有的服务器都被唯一的 IMR 控制的话，你需要在每一台机器上都启动 imr，并且通过使用 -master 和 -salve 参数保证你只有一台机器是 IMR 接入器，其他都是 OAD。

2、imradmin 工具是一个命令行工具用于和 IMR 接入点通讯并查询和更新 IMR 数据库。imradmin 的工作方式和在本章开头讨论的理论上的 reg_svr_with_imr 工具具有一样的功能。许多 Orbacus 的管理命令都是通过这个工具各种各样的子命令完成的。

当你第一次运行 IMR 时，imr 创建并初始化数据库。Orbacus 不能有重复的 IMR 服务器，这样导致 Orbacus 具有单点故障，这是许多公司都不能接受的。Orbacus 一直未提供对重复服务器的支持。

7.3.3、 TAO

在 1.2 版本以前, TAO 的IMR只支持单一IMR,直到 1.3 以后TAO开始支持如 [图 7.4](#) 的分布式IMR。这个改造工作还在继续进行。IMR的功能被分为 ImplRepo_Service(IMR接入点)和ImR_Activator(IMR启动和监听器)。然而不论 1.3 还是 1.4 中, 这些IMR进程必须运行在一台机器上。TAO还没有提供从一个ImplRepo_Service委托到另外一台机器的ImR_Activator启动服务器的功能。

tao_imr 工具是一个命令行工具用于和 IMR 接入点通讯并查询和更新 IMR 数据库。tao_imr 的工作方式和在本章开头讨论的理论上的 reg_svr_with_imr 工具具有一样的功能。许多 TAO 的管理命令都是通过这个工具各种各样的子命令完成的

当你第一次运行 IMR 时, tao_imr 创建并初始化数据库。TAO 不能有重复的 IMR 服务器, 这样导致 TAO 具有单点故障, 这是一些公司都不能接受的。TAO 也未提供在 IMR 中注册重复服务器的负载均衡的功能。

7.4 不同的IMR对比

[7.3 小节](#)中讨论了Orbix, Orbacus, TAO三种CORBA产品的IMR的不同点和相同点, 这些是非常值得注意的。因为在其他的CORBA产品中也可能有IMR类似的相同点和不同点。

- 每一个 CORBA 产品都趋向于使用自己术语为 IMR 命名。例如 Orbix 使用 locator daemon, node daemon, Orbacus 使用术语 IMR 和 OAD。这些不同的术语很容易使用户感到迷惑
- 一些早期CORBA使用的都是单一的IMR, 但是如今的CORBA产品都将IMR按

照 [图 7.4](#) 的方式划分成两个应用程序。

- 一些CORBA产品的框架提供了IMR重复部署和应用服务器重复部署的功能。

这种重复的功能使CORBA部署避免了IMR成为单点故障源，并且为客户端提供了负载均衡的能力。关于重复部署机制在 [17 章](#) 和 [18 章](#) 将会讨论。

第8章、部署CORBA应用程序

- [部署CORBA客户端](#)
- [部署CORBA服务端](#)
- [名字服务和IMR](#)

8.1 部署CORBA客户端

在不同的 CORBA 产品中，CORBA 客户端的部署细节存在者一些不同。然而基本概念是非常简单明了的，一般，你需要做如下的事情：

- 安装 CORBA 产品的子集(例如配置文件：`.jar` 格式文件和共享库)在机器上。明显地，你不需要安装 CORBA 产品中有关开发的部分，例如 IDL 编译器。一些 CORBA 产品提供“运行安装包”，“运行安装和开发包”等不同的安装选项。如果没有这些安装选项，那么从开发机上识别出运行包，并拷贝到部署机上也是一件非常容易的事情。
- 在部署机上安装客户端可执行程序，和与 CORBA 框架相关的一些基础文件，例如配置文件、日志目录

当你运行 CORBA 客户端的时候，你可能需要提供一些和命令行相关的参数。

- 你需要通过参数告诉程序，那里可以找到 CORBA 配置文件。这些信息可能以不同方式传递给 JAVA 程序，例如以 JAVA 程序用系统属性 `System`

Property 来传递，非 JAVA 程序使用环境变量。

- 如果配置文件的诊断级别diagnostic level，并不是你想要的诊断级别，你可以通过命令行参数的方式覆盖配置文件中的信息。例如在CORBA的配置文件没有指定连接方法，例如名字服务，那么你可以通过参数-ORBDefaultInitRef或-ORBInitRef参数来指定([12.5 小节](#)详细讨论)
- JAVA 在其标准库中提供了一个 CORBA 的实现，如果你的应用程序要使用其他的实现，那么你必须通过命令行参数来定义系统属性来告知你的应用程序你使用的是什么 CORBA 实现。

如果你通过命令行参数来运行 CORBA 程序，那么记住这些必须要输入的命令参数是非常难得。一种简单的做法是将命令行参数写到 unix 的脚本文件或 windows 的批处理文件中去。

上述关于 CORBA 客户端的部署非常简单，但是正是因为简单，很多 CORBA 提供商都不愿出手册来描述它，讽刺的是：由于缺乏文档的描述，CORBA 的客户端部署看起来比其他的更难。如果你的 CORBA 厂商没有提供部署相关的文档，那么给他们的技术支持部门发邮件将会让你得到有用的帮助指导。

本节讨论的有关客户端的部署不仅适用于客户端，也适用于服务器端。我将在下一节讲解关于服务器端部署的其他细节。

8.2 部署CORBA服务端

服务器的一些概念，例如：POA策略，IMR，对象引用，这些概念互有交互，并影响着CORBA服务器部署。大部分的CORBA程序都部署在TCP/IP的网络上，所以在讨论本节时，我们都假设读者了解TCP网络通讯的基本概念。[8.2.1 节](#)将会对基于TCP的非CORBA服务器进行一个简单的介绍，第 [8.2.2 节](#)我们将通过

和普通TCP 服务器类似的部署来讲解CORBA服务器的部署方式。

8.2.1、 TCP介绍

TCP 服务器监听在一个固定的端口上，并接收来自客户端的连接。TCP 服务器监听的端口具有两个选项。

- 1、服务器能告诉操作系统，它想监听在一个指定(固定)端口上
- 2、服务器告诉操作系统，他想监听在端口“0”上。操作系统以此知道，服务器进程其实是希望监听在一个任意的端口之上。这个任意端口我们称为随机端口或临时端口。

服务器程序在固定在端口上监听，意味着无论服务器重启了多少次，它都将监听在一个固定端口之上。相反的，如果服务器监听在随机端口上，那么每当服务器重启的时候服务器都将监听在不同的端口之上。

通常，一些基于TCP的应用程序监听在知名，固定的端口之上。例如WEB服务器的默认端口就是 80。于是当你在你的浏览器里键入 www.amazon.com，你的浏览器将会连到特定主机的 80 端口。正是因为这样，你只要记住WEB服务器运行的主机名，而不需要记住主机名和端口号，这使得人们很容易的记住URLs。同样的原因，其他一些知名的TCP应用程序，比如POP3 邮件服务器(端口 110)，FTP服务器(端口 21)，TELNET 服务器（端口 23）等等也有固定端口保留供这些应用程序使用。

TCP 服务器监听在随机的端口上，那么它必须通过某种方式将其监听的端口告诉客户端，以便客户端能连接它。一个公告的做法就是将端口写在一个文本文件里面，然后客户端来读取这个文件。当然，这种做法必须服务器和客户端能够访问同一个共享存储，所以这是一中扩展性较差的选择。更复杂，更具有扩展性的方案是采用公告服务器，公告服务器监听在一个知名的固定端口之上，并且维护一个 CORBA 服务器名->到主机端口的映射列表。当一个 CORBA 服务器启动

并监听在一个随机端口之上后,他向公告服务器注册其端口和服务名。通过这样,客户端的 CORBA 应用程序就可以连接目标服务器相关的公告服务器来,并获得相关的端口信息和目标服务器进行通讯。

这种公告服务器的好处是,你只需要分配一个固定的端口,这个固定端口就是公告服务器的端口。其他的服务器通过公告服务器公告自己的随机端口。这样就减少了从可用端口中筛选保留端口的管理工作。

8.2.2、 CORBA服务器的部署模型

大多数的 CORBA 应用程序都是基于 TCP 网络,所以 CORBA 服务器就像 TCP 服务器一样监听在一个端口之上。CORBA 的实现库 “implements repository” 和名字服务 “Naming Service” 就类似于 TCP 服务的公告服务。名字服务提供名字到 IOR 的映射,实现库提供 IOR 到 IOR 的映射。当你了解了映射的细节是:由原始 IOR 到变化后的 IOR 映射是 IOR 中嵌入主机和端口的映射时,IMR 的功能看起来就不那么奇怪了。正因为这样,IMR 提供了包含知名端口的 IOR 到包含目标服务器随机端口的映射。

CORBA 服务器可以通过配置(或者通过私有 API 编程),以下面四种方式部署服务器:

- 1、**服务器监听在随机端口上并且不使用 IMR。**这种部署方式,只适用于临时对象,也就是说,服务器的所有对象都包含于临时策略的 POA 中。这是因为如果持久化对象的 IOR 中嵌入的端口是随机端口的话,服务器就有可能在每次重启后监听不同端口。这种部署方式将得到不可以预期的负面影响。
- 2、**服务器监听在随机端口上,并且通过通过 IMR 部署。**这种情况下,服务在 IOR 中嵌入的是临时对象的自己的主机和随机端口,但是对于持久对象,

IOR 中嵌入的是 IMR 固定端口和主机名。当客户端第一发送他的请求到持久化对象时, 如图 7.2, 请求将被发到 IMR, IMR 使用请求头信息中的对象键值得到目的服务器。如果服务器进程不存在或停止, IMR 重启动服务器, 并将客户端请求重定向到服务器。这种重定向只发生在客户端向服务器请求的第一次, 以后客户端将直接向服务器发请求。

3、服务器监听在固定端口上并且不通过 IMR 部署。这种情况下, 服务器在临时或持久的 IOR 中嵌入自己的服务和端口号。在持久化 IOR 中嵌入固定端口 (相对于临时端口而言), 保证了当服务器重启时 IOR 任然有效。

4、服务器监听在固定端口上, 并通过IMR进行部署。在这种情况下, 服务器将自己的主机名和固定端口号嵌入到临时对象中, 但是服务器嵌入IMR的主机名和端口号到持久对象中。当客户端第一发送他的请求到持久化对象时, 如 [图 7.2](#), 请求将被发到IMR, IMR使用请求头信息中的对象键值得到目的服务器。如果服务器进程不存在或停止, IMR重启动服务器, 并将客户端请求重定向到服务器。 这种重定向只发生在客户端向服务器请求的第一次, 以后客户端将直接向服务器发请求

关于上面部署的讨论, 有几点值得我们注意。

第一、服务器使用自己的主机和 (固定或随机) 端口号, 一般都用于临时 IOR 中。这意味着临时 IOR 重来不通过 IMR 来部署。很多人错误的认为, 只要 IMR 运行, 服务器就会自动使用它。然而, 如果服务器只包含临时对象的话, 服务器将忽略 IMR 的存在。当然并没有人能阻止 CORBA 的厂商设计的产品中临时 IOR 也包含 IMR 的端口和主机, 但是这样做不会带来任何好处。到目前为止, 笔者从来没有看到过有厂商这么做。

第二、服务器使用固定的端口号有如下几点潜在的好处 (上述列表中的 3, 4, 部署方法)

- 一些公司使用硬件路由做负载均衡客户端和运行在多台机器上的重复的 CORBA 服务器的连接。这种方式下，服务器必须使用固定端口
- 使用固定端口，对防火墙是友好的。(固定端口方便防火墙设置)
- 通常情况，一个系统组件越少就越可靠。因为这个原因，一些厂商更愿意使用没有 IMR 的部署方式。通过使用固定端口，可以达到部署持久对象而不需要 IMR。(上述部署模型中的第三点)
- 一个相反的观点是，如果 IMR 非常稳定，那么可以通过 IMR 可以提高系统的可靠性，这是因为 IMR 可以自动重启服务器。同时通过 IMR 一些系统实现了容错的机制。容错可以通过部署重复服务器或 IMR 来实现，通过这样的部署就去除了计算机系统的单点故障问题。

第三、一些 CORBA 产品很容易使用上述讨论的四中模型。其他的一些 CORBA 产品提供了对 CORBA 部署模型子集合易用支持，但是需要使用一些配置文件和私有的 API。这是非常不幸的，因为这样导致了在应用程序中硬编码了服务器的部署逻辑。将部署选择问题放在部署处理而不是在程序设计期间来处理部署问题是更好的一种做法。此外私有 API 的使用，必然导致程序的可移植性降低。CORBA Utilites 包中《使创建 POA 层次结构简单》的章节中讨论了代码移植性和相关的部署问题。一些 CORBA 产品，只支持上述部署模型的子集。例如，omniORB 就没有包含 IMR 的实现，所以它无法支持上述的 2，4 部署模型。

最后、就象前面提到过那样，一些 CORBA 产品提供了建立在 IMR 上的容错机制。简单来说，当客户端第一次通过 IMR 向持久 IOR 发送请求时，IMR 能重定向到多个重复服务器的某一个服务器上。这种容错机制，只支持持久对象。因为临时对象不通过 IMR 进行通讯。

8.3 名字服务和IMR

在 [8.2.2 小节](#),我提过名字服务提供了名字到IOR的映射,同时IMR提供IOR到IOR的映射。在名字服务和IMR中都包含映射功能这点上,经常困惑着CORBA的新手。然而名字服务提供的映射和IMR提供映射是为不同的需求服务的。

- 字符串化的对象引用([3.4.2 小节](#))是“IOR:<16 进制码>”形式,因此它可以以比较方便的形式存储在文本文件和数据库。但是它却不容易让人记住。名字服务提供了一种方便让人记住IOR形式。这种方式容易让人记住一堆的IOR。当然,开发者也可以将很多字符串化IOR存储在文件中,每一个IOR放入一个文件。如果这些文件名容易被记住的话,这种技术基本上和名字服务就具有相似的功能了,除了它不支持一个支持位置上扩展。因为很少有文件系统能在很大的范围内被共享。
- 无论客户端程序是通过文件或名字服务获得 IOR, IOR 嵌入的主机和端口有可能是目标服务器或目标服务的 IMR。如果主机和端口是 IMR,那么 IMR 重新定向客户端到真实的服务器进程。重定向有如下的好处: (1)IMR 总是监听在固定端口,但是 Server 通过 IMR 部署可以监听在随机端口上,那么为服务器分配固定端口的管理工作可以减少; (2)IMR 能(重)启没有运行的服务器进程; (3)一些 IMR 内建了一些负载均衡和容错的功能。

另一个关于 IMR 和名字服务让人迷惑的地方是,一些产品的名字服务本身是通过 IMR 来部署的。当客户端需要访问名字服务,并通过名字服务访问一个通过 IMR 部署的持久对象时,将会出现 IMR 重复重定向的情况。

- 当客户端调用resolve_init_references()([3.4.1 小节](#)),将得到一个名字服务的IOR(这个IOR是从配置文件中得到)。当客户端调用这个IOR

和名字服务通讯的时候，客户端的第一次请求将发送到IMR，并且IMR将请求重定向到客户端到名字服务。

- 当客户端从名字服务中获得一个 IOR，并且通过这个 IOR 请求目标 CORBA 对象时，客户端有可能将请求通过 IMR 重定向到目标服务器进程。

第四部分、CORBA 基础功能部件

第9章、 IDL的更多细节

- [伪IDL,本地local和 native 类型](#)
- [传值对象Object By Value \(OBV\)](#)
- [版本信息](#)
- [库标示 Respository IDs](#)
- [其他 新关键字 Miscellaneous New Keywords](#)

IDL的基础概念在 [1.4 节](#) 讨论过。本节将讨论关于IDL中最近增加的信息和一些容易被忽略的细节,并且我们还将讨论在没有版本机制下的CORBA是如何工作的。

9.1 伪IDL,本地local和 native 类型

在一般情况下,任何系统都无法做到用系统本身的术语来定义自己,CORBA同样不例外。OMG很自然的使用了IDL来定义自己的CORBA中的大部分API,但是其中的一部分API无法通过IDL来定义。例如,所有的CORBA接口都隐式的从基础类型Object继承。由于Object是保留字,所以不可能通过IDL语法来描述Object的API。为了解决这个问题,OMG使用了一种非正式的称为伪IDL “pseduo-IDL”的注解方式来定义内建类型,例如Object的API。伪API的写法非常接近真实的API,但是通过注释//PIDL指示这个API的定义并不是一个IDL语法有效的API,因此它不能被IDL编译器编译,伪IDL用来扩展早期的CORBA规范。

随着CORBA的更新,两个新的关键字local和native也被引入到IDL

中，并使得 IDL 可以定义更大范围 CORBA API。这两个关键字的引入可以减少 (但不能完全去除) 对伪 IDL 的使用

local 关键字能出现在接口定义前。这样定义的效果是这个接口只能以本地的方式访问，应用级程序员一般不使用这个关键字。这个关键字的目的是允许很多本地访问的 API 在 IDL 中定义。例如：DynAny([15.3 小节](#))，Current([13 章](#))，可移植拦截器 “portable interceptor” ([14 章](#))，策略([16.1 小节](#))，ORB 本身和许多在用于服务端实现的类型—— POA，POAManager，ServantManager，策略，等等([第 5 章](#))都定义为 local interface。

native 关键字用来表示指定的类型非 IDL 类型，而是宿主语言类型，例如 C++/Java/Cobol 或其他程序用于实现 CORBA 应用程序的语言的类型。native 只能用于 local interface 的参数。native 类型是为了允许让部分 CORBA 和宿主语言交互。例如 CORBA 使用 Servant 表示代表 CORBA 对象的宿主语言对象；相应的声明如下：

```
native servant
```

POA 的基本部件定义了几种以 Servant 做参数的本地接口 local interface。

9.2 传值对象 Object By Value (OBV)

CORBA 在 J2EE 流行前就流行了很多年。当 J2EE 技术宣布的时候，公认的 CORBA 和 J2EE 是互补的关系，但是另一方面 CORBA 和 J2EE 在又是竞争对手。如今有很多争论，这两种技术是否会一种技术击败另一种技术。在 JAVA 中有一种 CORBA 中没有的功能，许多 OMG 的人认为 CORBA 应该增强并提供相似的功能。这个技术被称之为传值对象 object by value (OBV)。驱动 OBV 力量

并不是因为 OBV 是一个很好的技术创新，而是为在市场和政治上抵御来自于 J2EE 的潜在威胁。可以预见到，由于这项技术的技术界限不清晰和容易让人误用，OBV 规范具有很大争议。

9.2.1、 OBV的java等价类

在我们以技术观点开始讨论 OBV 之前，我们将先讨论 OBV 想模仿的 JAVA 的相应技术。JAVA 具有持序列化 serializing 对象的内建功能，即，将内存中的对象转换为字节流或将字节流转换成内存中的 JAVA 对象。因为这种序列化的能力，JAVA 能方便将其对象持久化到介质，例如将序列化后的对象存储到文件或数据库中。同样以这种方式，JAVA 可以将对象转换成二进制流然后通过 socket 传送到另一个进程中，接收到这个二进制流的进程可以重新在自己的内存空间中创建这个对象。实际上，JAVA 对象可以以传值的方式从一个进程传到另外一个进程。事实上，直到现在所讨论的机制仅涉及到序列化和传送对象的状态(实例变量或域)。对象——包含着状态和维护其状态的方法。然而 JAVA 同样具有将实现对象方法的字节码“byte code”通过网络传输的能力。以这种方式，JAVA 同时就具有了传送方法和状态的能力。传送对象的字节码是很重要的，因为 Java 的接收进程有可能没有权限访问正确的字节码。例如，JAVA 的接收进程可能希望一个 Graphic 类型的对象，但是实际上它却接收到 Graphic 的子类型 Circle，因为这样它不能访问到正确的代码

关于 JAVA 这种能力的一个非常明显的问题就是：这种能力是否真的有用？一个典型的用法发生在如下的客户端和服务器的交互过程中。

- 1、客户端调用服务器上的方法。这个方法返回的是一个对象(状态，如果需要，字节码)

2、客户端调用这个对象本地拷贝上许多方法。

3、当客户端完成对本地对象的更新时，客户端远程调用服务器，并将更新的对象做为参数传向服务器端。

这种使用场景提供的好处是如下：

- 在进程之间传对象(值方式)能提供一个重要的优化。上述过程中的第二步，客户端调用本地对象上的方法相较于对远程对象上方法调用要快。这是因为远程调用一般都要 包含几毫秒的网络传输，本地调用却不需要这样的消耗。
- 同样优化可以通过传数据的方式达到，例如，在服务器和客户端传递结构 struct，和序列 sequence，但是这会给客户端直接暴露底层的数据。

如果这个方法的能够传输字节码，那么将这些底层的数据封装在一个方法里面是一种更好的做法。

9.2.2、 CORBA中的传值对象

CORBA接口interface 只有方法没有状态变量。相对于接口interface，CORBA的结构struct 只有状态变量和域但是没有方法。一个新的类型 valueType, 被引入到了IDL中。valueType看起来介于interface和struct之间，因为valueType既有方法又有状态变量。一个valueType的例子如 [图 9.1](#)

```
valuetype Date {  
    short    year;  
    short    month;  
    short    date;  
    void     next_day();  
    void     previous_day();  
};  
  
valuetype OptionalString string;
```

图 9.1 IDL valueType 定义的例子

当 valueType 作为参数被传递，他的状态变量将被传递。在 valueType 的方法调用将总是本地调用。通常，方法的实现代码无法通过网络传送，因为客户端程序和服务端程序有可能用不同的语言写成，并且运行在不同的 CPU 上。

正因为这样，客户端和服务端必须维护各自的 valueType 方法实现。这样的要求存在很大的问题：因为没有人能保证服务器端和客户端的同样的方法实现具有同样的语义。当开发第一个服务和客户端版本的时候，开发人员将花费很大的精力去保证客户端和服务端端的语义一致。然而随着应用的程序的维护进程，服务器的方法很有可能发生语义上的修改(可能是 bug-fix 可能是性能优化引起)，但是有可能相同的修改并没有在客户端进行。在项目的生存周期内，可能服务器采用一种语言实现，比如 C++，几个客户端采用不同的语言实现，比如 Ada，Java，Cobol。维护 valueType 方法多种语言实现的语义一致性，将会变成非常繁重的负担。

valueType 的反对者指出，不含 valueType 的分布式应用程序成功的开发和部署经历了几十年。因为这样，valueType 不是 CORBA 的本质特点，并且是可以被忽略掉的。

讨论了 valueType 主要的缺点后，下面我们来讨论一下 valueType 提供的额外的功能。

```
valuetype Base {  
    long    some_data;  
};  
  
valuetype Derived : Base {  
    long    more_data;  
};
```

图 9.2 valueType 继承定义

第一、如果你定义的值type只包含状态变量但不包含方法，那么

valueType的语义和struct相同，但是valueType有额外的好处，你可以对valueType使用单继承。如 [图 9.2](#)，你也可以将valueType看成一个可以继承的struct。

第二、valueType总是以类似于C++指针(Java中的reference)的方式传送。例如，如果valueType的域是另一个valueType，那么这个域指向了一个嵌入valueType。在valueType被使用的地方可以用空指针。这样就通过valueType把指针的语义引入到了IDL，同时valueType还允许建立类似环状图的数据结构。如果你的valueType只有一个域，例如，叫做string，这样，就允许你传一个“正常”的字符串或或则空指针做为参数。在效果上，提供一种非常方便的“可选值”做为参数的方法。OBV的设计者认为这种valueType的可选值的使用方式将会非常有用，并且他们为这种用法发明了一种简便的语法定义方式(“语法糖”)。这种语法糖的演示如 [图 9.1](#) 中的例子。这种形式定义通常被称为valueBox。

9.3 版本信息

CORBA 的 IDL 定义中不含有版本机制。不幸的是，这一点让很多人感到了困惑。困惑的原因是在 CORBA 1 中定义了版本的语法形式。这个语法是：

#pragma version。 它以如下的例子被使用：

```
#pragma version "1.2"
```

1.2 的意思是在后面的定义的 IDL 部件的版本号。

作为版本支持，并不是只要定义语法形式就可以，还需要基础框架的额外支持。然而，OMG 从来没有定义有效基础功能模块来支持#pragma version 的使用。由于#pragma version 仅仅是因为历史原因遗留，所以它已经被 ORB 忽略。然而，不幸的是，这个语法形式继续存在导致很多人不正确的假设 CORBA

有版本机制，并且使得他们使用了很多时间和精力来使用他。

弄清了 CORBA 没有版本机制，接下来的问题是是否有存在其他机制来模拟版本机制，两个并不完美的建议在下面讨论：

一种方法是 c 使用继承机制来模拟版本机制。例如，让我假设你有一个存在 IDL 接口叫做 Account 并且你想创建一个新的版本，并增加新的功能。你可以定义一个新接口 Account2，这个接口继承于 Account 并增加了新的方法。这种方法适用于新版本只增加了新方法；对于删除原有方法或修改原有方法函数标示并不适用。这种方法，在你使用多个版本的情况下，将会引起深层次的继承。

<pre>module Finance { ... interface Account { ... }; };</pre>	<pre>module Finance2 { ... interface Account { ... }; };</pre>
--	---

图 9.3 拷贝-修改方式定义版本

另外一种模拟版本机制的方式是定义一个新的无关的接口，如 [图 9.3](#)。原始应用程序定义的 IDL 类型在 module Finance 中（图中左边的方框）。当应用程序定义了新的版本被开发，将原有 IDL 拷贝成一个新的文件并将 Finance 重命名为 Finance2。于是 Finance2 将不会随便修改而不受到约束。就像人们关注的，Finance2::Account 和 Finance::Account 比较接近，因此他们认为这两个接口是同一个接口的不同版本。然而这只是人认为是不同版本，对于 CORBA 来说，这两个接口完全是语义无关的两个接口。

通常情况下，将代码定义在模组中是非常好的编码实践。这样做减少了名字污染。使用模组提供版本的另一种好处：将版本号放在模组名中比将版本号放在模组中定义的类型名上要更方便。同样，随着应用程序源代码的版本从版本 1 升级到版本 2，一个简单在源代码中的全局查找替换就可以达到目的。

应该注意到，并不是只有 CORBA 缺乏版本机制，大多数的中间件产品都缺乏版本机制，同样大多数编程语言也一样。

9.4 库标示 Repository IDs

库标示是 IDL 文件中定义带全路径名的实体一种变形形式。例如：
Finance::Account 的库标示是 IDL:Finance/Account:1.0。通常在全路径中所有的“::”都会被替换为“/”。替换后的字符串再加上“IDL:”前缀和“1.0”后缀。

IDL 允许在 IDL 定义 #pragma prefix "... "的指令。例子如下：

```
#pragma prefix "acme.com"
module Finance {
    interface Account { ... };
};
```

如果 #pragma prefix 指令被用于 IDL 文件中 (例子中 acme.com)，那么这个指定的内容将会出现在 IDL 中所有类型的库标示中。例如类型 Finance::Account 的库标示是 IDL:acme.com/Finance/Account:1.0

库标示是运行类型信息。大部分 CORBA 依赖编译时的类型检查，所以库标示很少被用到。但是，程序员却会经常遇到库标示，因此了解什么是库标示，库标示有什么用处是非常有用的。下面不完整的罗列了库标示是如何使用的：

- 在 1.5 小节提到过，一个 IOR 包含着对象“联系细节”，然而 IOR 也可能包含对象类型的库标示。IOR 中包含库标示对调试是非常有帮助。例如，很多的 CORBA 产品都提供了命令行工具打印 IOR 中包含的库标示和“联系细节”。人们通常都使用这些工具来诊断开发中或部署中的问题。问题出现经常是因为客户端得到错误的 IOR 类型，例如，一个 IOR 本是 Employee 对

象，结果却是 `Finance::Account` 对象。通过查看 IOR 中的库标示可以很轻松帮助人们解决这类问题。

- 当应用服务器的方法抛出一个异常，这个异常的库标示首先被编组(序列化)，然后跟着是异常的域。通过这种方式，CORBA的运行系统的客户端能使用库标示来决定异常的类型；这些告诉了CORBA客户端运行系统如何在解组(反序列化)抛出异常的字段。关于编组的讨论我们将放在[11.2小节](#)讨论。
- 库标示可以作为元信息([第 15 章](#))在程序中被使用

许多人对`#pragma prefix`这个指令的目的非常疑惑。这个问题的答案可以通过一个例子来解释。让我们假设美国银行 Bank of America 定义了一个模组叫做 Finance 包含 Account 接口。如果没有`#pragma prefix`指令，这个类型的标示库是“IDL:Finance/Account:1.0”。问题是美国银行不是唯一的定义了 `Finance::Account` 接口的公司。如果另外一个公司也定义了同样一个名字的(假设含有不同的方法)，客户端程序原来和美国银行的 `Finance::Account` IOR 交互，结果意外的取到其他公司的 `Finance::Account` 接口，当这个问题出现时，将是非常难诊断的。为了避免这个问题的出现，CORBA 鼓励开发人员将`#pragma prefix`指令放在 IDL 文件中。这个前缀字符串将唯一的表示开发者公司。典型的是使用 internet 的域名，如 URL。例如，美国银行的开发人员可能包含如下的指令

```
#pragma prefix "bankofamerica.com"
```

现在，定义在这个文件的 `Finance::Account` 接口的 IOR 将是：

```
IDL:bankofamerica.com/Finance/Account:1.0
```

这样如果客户端取道包含 `IDL:bankofamerica.com/Finance/Account:1.0` 的 IOR 或则包含 `IDL:bankofamerica.co.uk/Finance/Account:1.0` 的 IOR 时，客户端

的运行系统将会抛出一个异常信息。

9.5 其他新关键字Miscellaneous New Keywords

最新几年如下的几个关键字被加入到 IDL 中

`typeprefix` 关键字和 `#pragma prefix` 指令的用途相似，一个使用的例子如下：

```
module CosNaming {  
    typeprefix CosNaming "omg.org";  
    ...  
};
```

在这个例子中，`typeprefix` 命令将引起 "omg.org" 前缀嵌入到 `CosNaming` 模组中所有定义的类型库标示中。

`import` 关键字的作用和 `#include` 非常相似，一个使用例子如下：

```
import CosNaming
```

如例子所示，`import` 后包含一个模组名字，这效果和 `include` 该模组的 IDL 文件一样。

一个方法可以包含抛出 `raise` 子句，这意味方法可以抛出用户定义的方法。

相反的，属性 `attribute` (本质上，属性是一系列 `get`-和 `set`-方法的语法糖定义) 却不能有 `raise` 子句，所以属性不能抛出异常。新的 `getraises` 和 `setraises` 子句用于指定属性对应的 `get`-或 `set`-方法能抛出的异常：

```
exception X { ... };  
exception Y { ... };  
exception Z { ... };  
interface Foo {  
    attribute string name getraises(X, Y) setraises(Y, Z);  
};
```

第10章、 可交互对象引用 (IOR)

- [介绍](#)
- [IOR的IDL定义](#)
- [代理 Proxy](#)

10.1 介绍

可交互对象引用 “interoperator object reference” (IOR), 是客户端和 CORBA 对象交互的“联系细节”。因为 IOR 总是工作在不同的 CORBA 实现中, 所以我们将 IOR 称之为可交互的。例如一个 Orbix 实现的服务器可以和客户端是 Orbacus, Visibroker, TAO, omniORB 或则 JacORB 交互。

很多人都满足于知道 IOR 客户端和服务端 CORBA 对象是“联系细节”。并且他们不再查找额外的关于对象引用的信息。如果你是这类人, 你就可以略过本章的内容。然而, CORBA 的 IOR 是非常灵活的, 并且这些灵活是 CORBA 实现其他组件的基础, 包括实现库 ([第 7 章](#)), 消息 ([16 章](#)), 事务 ([21 章](#))。正因为这样, 了解对象引用 IOR 的内部信息有利于理解 CORBA 的其他部件。

10.2 IOR 的IDL定义

CORBA 使用 IDL 定义其系统的底层 API, 所以, CORBA 使用 IDL 定义 IOR 的信息也不会让人觉得意外。在 IDL 中 IOR 被定义为如 [图 10.1](#) 的一个结构。这个结构的内容可以用类似的名片来解释, 就包含“联系细节”这一点上, 名片和 IOR 是等价的。一个简单名片的例子如 [图 10.2](#)

```
module IOR {  
    typedef unsigned long ProfileId;
```

```
const ProfileId TAG_INTERNET_IOP = 0;

struct TaggedProfile {
    ProfileId      tag;

    sequence<octet> profile_data;
};

struct IOR {
    string          type_id;

    sequence<TaggedProfile> profiles;
};

...

};
```

图 10.1 IOR 的 IDL 定义

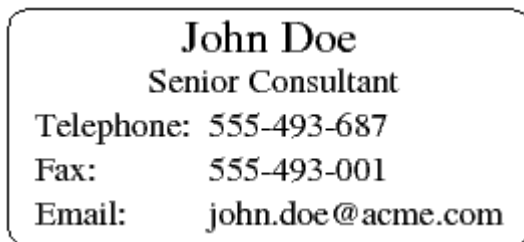


图 10.2 名片例子

结构中的type_id代表的是对象的库标示([9.4 小节](#)), 对应名片中的工作职位, 如: Senior Consultant, Sales person, director, software engineer 等。

IOR的另一个域是一个TaggedProfile 的可变数组(sequence)。TaggedProfile这个名字并不是很直观, 所以需要一些解释。profile_data这个字段是一个有二进制缓存区(sequence<octet>), 用来表示联系细节。并且tag字段告诉我们如何使用解释这个二进制缓存。一个profile_data在名片对应的是一个号码555-493-687。要理解这个号码的意义, 通过查看tag值, 你可以知道这个号码是一个电话号码而非是传真号码。OMG定义了tag值为

TAG_INTERNET_IOP指定二进制的值表示IIOP的“联系细节”，另外的IDL定义IIOP的联系细节内容（我们将在[10.2.2](#)讨论）。这种tagged加profile指定联系细节的方式是经得起未来扩展验证的。例如，如果未来TCP/IP被新的通讯协议所替代，那么OMG将会定义新的tagged和profile_data格式来支持新的传输协议。此外，CORBA的开发商，还可以定义自己的私有的tagged和profile_data。例如，CORBA如今还未标准化使用共享内存和无线网络的通讯方式，但是几家CORBA厂商已经提供了对这些新通讯的方式tagged和profile_data私有定义。

IOR 包含一个 **TaggedProfile** 的可变数组。这意味着一个 IOR 中可以多个联系细节。例如，也许其中的一个联系细节是关于私有共享内存协议的，另外一个则是 CORBA 兼容的 IIOP 协议的。当客户端是建立在同一个 CORBA 产品上并使用 IOR，那么它将使用其中的一个 profile(也许是先发现先用)。而如果客户端是另一个 CORBA 产品，并且试图使用 IOR，那么客户端将忽略共享内存协议，因为客户端不能识别这个私有 tag 值，而使用 IIOP 的 profile。一个 IOR 可以使用多个 profile 类似于名片中的使用的多个联系方式，例如：电话，传真，Email 和邮编等等。

一个 IOR 也可能包含几个 **TaggedProfile**，并且其底层通讯方式相同。例如：一个 IOR 能包含 3 或 4 个不同的联系方式。这种用法用来提供负载均衡或容错。注意到 CORBA 允许这种灵活性，但是 CORBA 并没有要求 CORBA 厂商利用这些灵活性，这是非常重要的。许多 CORBA 产品只有一个在 IOR 中联系细节。有些厂商提供了基于在 IOR 提供多个联系细节来做负载和容错。服务器输出单个或多个 IOR 并不影响客户端和服务器的交互性。

10.2.1、 空间优化Space Optimization

IOR和名片以非常有趣的方式存在一个相似的缺点。名片中都包含人名——换句话说，是他或她的唯一标示。CORBA对象也有标示，唯一标示服务器中的对象。这个标示称为对象键值*object key*。对象键值信息嵌入在**TaggedProfile**信息中。因为这样，如果一个IOR包含多个**TaggedProfile**，那么，对象键值将在**TaggedProfile**中重复多次，然而 TaggedComponent机制([10.2.3 小节](#)讨论)，提出一种可选的空间优化方法，允许一个profile里面包含一个对象键值和多个(主机、端口)对

10.2.2、 IIOP联系细节Contract Details

上面还没有讨论的一个问题是：IOR中的具体联系细节是什么样的？实际情况中，IOR中只包含一个IIOP协议**TaggedProfile**的profiles。[图 10.3](#)展示了一个这个**TaggedProfile**中的profiles的存储信息。GIOP编组规则([11.2 小节](#))被用于编组IIOP profile成一个二进制的缓存。然后这个缓存被存储在**TaggedProfile**中的profiles(sequence<octet>)中。

```
module IOP {
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId      tag;
        sequence<octet>  component_data;
    };
    ...
};

module IIOP {
```

```

struct Version {
    octet  major;
    octet  minor;
};

struct ProfileBody_1_1 { // also used for 1.2 and 1.3
    Version                iiop_version;
    string                 host;
    unsigned short         port;
    sequence<octet>        object_key;
    // added in 1.1; unchanged for 1.2 and 1.3
    sequence<IOP::TaggedComponent> components;
};

...
};

```

图 10.3 IDL 定义的 IIOP IOR 中的 profile

可以看到，IIOP **TaggedProfile** 中的 profiles 中包含一个用于连接服务器进程的主机名和端口号。服务器端可能包含多个对象，所以 object_key 是用于唯一的标示服务器中的对象。TaggedComponent 出现的目的是我们将在 [10.2.3 小节](#) 讨论。

当客户端使用这个 IOR 进行一个远程调用时，客户端和 IOR 指定的主机和端口建立一条连接，然后通过其发送请求。请求包的头部包含有对象键值和希望被调用的方法名。应用服务器的 CORBA 运行系统通过这个头查找相应的服务提供者 Servant ([5.2 小节](#))，并调用上面相应的方法。

10.2.3、在 IOR 中使用 TaggedComponent

IIOP 的 profile 中包含一个 sequence<TaggedComponent>。

TaggedComponent 中包含一个二进制数据(sequence<octet>), 和一个 tag, 这个 tag 告诉我们如何解释这个二进制数据。OMG 组织定义了 30 中不同的 TaggedComponnet 类型, 此外, 还指定了如下关于对象引用的信息。例如:

- 一个可以用来和对象通讯的变长数组(主机、端口)。这是用于空间优化, 当 IOR 中 profile 含有多个主机、端口号, 这种比用多个 profile 更节省空间
- 用来发送 char/string 或者 wchar/wstring 参数指定的字符集。
- 实现安全策略的信息。
- 对象是否参与分布式事务。
- 用于时间独立time-independent调用([16.3 小节](#))的消息路由器信息。

10.3 代理Proxy

IOR结构([图 10.1](#))包含CORBA对象的联系细节, 但是它并没有包含被调用方法的说明(signature)。无论是C++/JAVA还是其他的语言, 代理proxy是一个对IOR的一个封装器, 并且他提供了IDL定义的远程调用接口的方法说明(method signature)。当客户端调用代理上一个方法, 方法编组 Marshal([11.2 小节](#))in/inout参数到一个二进制缓存, 并且将通过IOR的联系细节将请求发送发送出去。代理等待并接受应答信息, 解组UnMarshal out/inout参数和返回值给调用代码。

前面的章节解释了 proxy 是一个 IOR 的封装器。然而区别 IOR 和 IOR 封装器之间的差异并不是那么重要, 所以术语代理 proxy 经常被用做 IOR 的同义词。

第11章、 联 机 协 议 (On-the-wire Protocols)

- [GIOP, IIOP和协议栈](#)
- [编组IDL类型](#)
- [GIOP消息类型](#)
- [GIOP重定向](#)
- [活动连接管理Active Connection Management \(ACM\)](#)
- [服务上下文](#)
- [编码约定](#)
- [双向GIOP和IIOP Bidirectional GIOP/IIOP](#)

11.1 GIOP,IIOP和协议栈

很多人知道 CORBA 客户端是和 CORBA 服务器端通讯，但是他们并不关心通讯是怎么进行的。然而，有一些人则更关心 CORBA 通讯架构的底层细节。如下的这几个原因比较让人感兴趣，例如：

- 一些人们想了解 CORBA 通讯的效率，通过了解底层通讯协议可以和其他的中间件技术如 IBM 的 MQ，SOAP 等进行对比。
- 大多数公司选择 TCP/IP 网络进行通讯，然而一些公司使用其他的传输协议。
这个公司的人们对 CORBA 通讯是否能足够灵活到可以定制自己的网络通讯协议而感到好奇。

OMG通过设计了一个多层协议栈提供了一个高效，灵活的应用程序间的通讯机制，如 [图 11.1](#)。所有CORBA的联机协议的设计都以一个通用原则为指导。这

使得CORBA可以方便的在不同协议之间建立桥。这些通用原则包括对象“联系细节”的IOR([10章](#))的定义, 而且IOR是与底层联机协议无关。

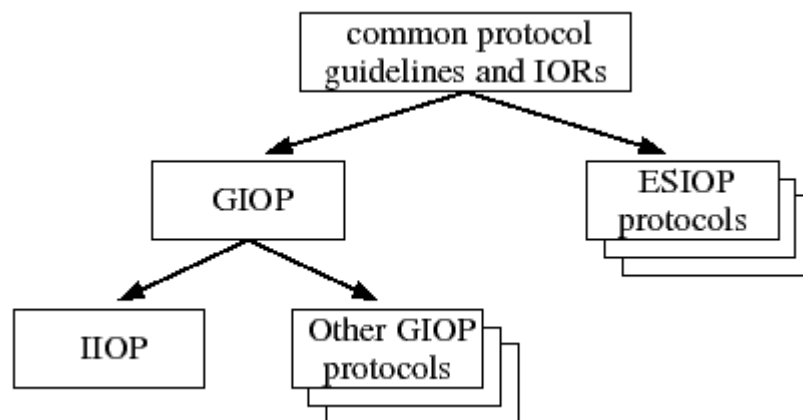


图 11.1 CORBA 协议层次

特定环境 ORB 间协议 *Environment-Specific Inter-ORB Protocol*(ESIOP)是一个针对特定应用环境提出的通讯协议。例如:

- 在 CORBA 早期, 有很多公司使用 DCE 的 RPC(一种老中间件技术)。一个称为 ESIOP for DCE 被定义来用于和 DCE 的应用程序进行交互。
- IONA 公司的 Orbix 已经移植到 IBM 的大机上。在大机的环境中, Orbix 通过使用私有的 ESIOP, 和大机上的本地组件进行高效的交互。

在 ESIOP 旁边, 所有的 CORBA 协议都基于普通 ORB 间协议 *General Inter-ORB Protocol*(GIOP)。这个协议定义了不同的消息类型(例如, 请求和应答消息类型)并且可以通过将 IDL 的数据类型(boolean, long, string, enum, struct, sequence, union, 等等)转换成特定的二进制格式表示并在客户端和服务端进行数据交换。

GIOP 中并没有指定用于在客户端和服务端传送的网络协议。例如 GIOP 并没有指定是否消息应该用 TCP/IP, X25, ATM 或则其他协议进行传送。而传输协议的选择在 GIOP 的特列化中。最著名的 GIOP 特列化就是因特网 ORB 间协议

Internet Inter-ORB Protocol(IIOP), 这个协议在 TCP/IP 网络上。所有的 CORBA 产品都必须支持 IIOP, 他们也可以选择性的支持基于 GIOP 的其他协议或 ESIOP 协议。IOR 中包含客户端和服务端通讯协议所有联系细节。

11.2 编组IDL类型

将 IDL 的类型转换成二进制内存块(为网络传输准备)的行为称为编组 *marshaling*。相反的从二进制内存块中解出为 IDL 类型的行为称为解组 *unmarshaling*。CORBA 使用的编组的规则称为通用数据表示 *Common Data Representation*(CDR)。CDR 包含着一些我即将讨论的规则。

一个 CDR 规则是数据被编组使用的编码 *endian* 规则是发送消息机器的编码规则。在 GIOP 的报文头中包含一个标志指明消息是以大编码 *big-endian* 或小编码 *little-endian* 格式。如果接收的机器编码 *endian* 格式和发送机器的编码格式一致, 那么数据将被直接解组; 否则在解组数值类型的数据时字节交换将会发生。这条 CDR 编码规则为客户端和服务端具有同一种编码规则的机器提供优化的传输方式。

另外一条 CDR 的规则是基础数据类型在内存中表示为多少字节和他们的内存对齐需求。

- `char`, `octet`, `boolean` 占用一个内存字节, 并且没有对齐要求。
- `short`, 和 `unsigned short`, 占用 2 个字节, 并且按照 2 字节对齐
- `long`, `unsigned long`, 和 `float` 占用 4 个字节, 并且以 4 字节对齐
- `long long` 和 `unsigned long long` 和 `double` 占用 8 内存个字节, 并且以 8 个字节对齐
- `long double` 占用 16 个内存字节, 并且以 8 个内存字节对齐

CDR 规则的内存对齐选择和一些 CPU 的内存对齐需求相似。设计者希望 CDR 这种内存对齐的设计能提高 CORBA 传输层的效率。但是, 经验显示, 内存对齐的需求给 CORBA 实现者主要带来的是编程的繁琐。但是这对以后应用将带来好处。

最后, 下面是一些根据基础类型对复合类型编组的规则

- enum 按照 unsigned long 进行编组
- string 按照 string 的长度(按照 unsigned long 编组), 紧跟着一个 string 的字符, 然后接着一个 null 终结符进行编组
- struct 是每一个域进行编组
- exception 编组是一个 string 类型的异常库标示, 紧接着是各个域编组
- union 编组是选择子和其后激活的分支编组。
- sequence 编组是 sequence 中包含元素的个数(按照 unsigned long 编组), 随后是每个元素编组
- array 按数组中每个元素进行编组
- any 编组是 TypeCode([15.3 小节](#)) 编组, 后跟其嵌入值
- object reference 被编组为 IOR, IOR 的结构如 [图 10.1](#)

上述讨论并不完整, 上述并没有讨论怎么编组 TypeCode 和 ValueType。

然而上述讨论提供 CDR 工作的典型例子。内存对齐需求规则意味着 GIOP 消息存在着一些浪费网络带宽的空白字节。然而在实践中, 空白字节只占 GIOP 消息的非常低的百分比, 所以 CDR 对网络带宽的浪费是非常有限的。

11.3 GIOP消息类型

GIOP 定义了 8 种不同的在客户端和服务端传递消息类型。理论上说, 应

用开发者不需要了解这些消息类型。但是，实践中，了解这八种消息类型对 CORBA 应用调试非常有帮助，特别是当 CORBA 厂商关于提供底层发送和接收数据调试信息的时候。

每一条 GIOP 的头 4 个字节是 G-I-O-P 这 4 个字母，表示 magic number 用于标明此消息是 GIOP 类型消息。

GIOP 消息类型是：请求 *Request*，应答 *Reply*，片段 *Fragment*，取消请求 *CancelRequest*，关闭连接 *CloseConnection*，消息错误 *MessageError*，定位请求 *LocateRequest*，定位应答 *LocateReply*，下面的小节将进行分别的讨论

11.3.1、 请求和应答消息 **Request and Reply Message**

请求和应答消息就是用于从客户端向服务器端发送请求的消息和从客户端接收应答回客户端的消息

11.3.2、 定位请求和定位应答消息 **LocateRequest and LocateReply Message**

定位请求消息如同一个 ping 消息，表示“你那里有对象吗？”。这个消息的应答是定位应答消息。CORBA 引入这个消息使得 GIOP 的重定位更有效率。关于 GIOP 的重定向在 [11.4 小节](#) 讨论。

11.3.3、 片段消息 **Fragment Message**

如果请求和应答的消息非常大，那么 CORBA 的运行系统将会以分片而不是整个消息方式的来传输。在这种情况下，第一个消息是正常的请求或应答消息，但是消息头中有一个标志位指示随后还有更多的消息。剩下的消息就以片段消息

的形式进行传输。

CORBA 产品并不强迫把大消息分隔成小消息。但是这样做是可选的。如果一个 CORBA 产品有能力进行消息分片传输，那么运行时的配置文件就要指明最大的不分片消息字节数。无论 CORBA 产品是否能发分片消息，CORBA 产品都必须能接收分片消息。

11.3.4、 取消请求消息CancelRequest Message

CORBA 客户端可以指定远程调用的超时时间，如果客户端在规定时间内接收不到应答，那么 CORBA 运行系统将会向客户端调用代码抛出一个超时异常并放弃对应答消息的等待。客户端的运行系统也可以发送一个取消请求消息(非必须)告诉服务器客户端将忽略服务器发送的应答消息。

CORBA 服务器端的运行系统能使用 CancelRequest 消息作为一个提示放弃之前接收到请求。然而，服务端也可以忽略它，事实上，是否忽略是由服务器是否已经分发这个消息来决定的。这是因为，服务器端 CORBA 运行系统不知道如何取消一个正在执行的应用级代码。

11.3.5、 关闭连接消息CloseConnection Message

CORBA 允许将空闲的连接关闭。如果客户端到服务器端的连接被关闭，那么，如果有客户端到服务器的请求发生，新的连接将被透明的建立。关闭空闲客户端的策略可以使得 CORBA 服务器具有处理成千上万的交互客户端的扩展性。

CORBA 需要监控如下的几种可能。客户端和服务器端的连接空闲了一段时间，正当服务器决定断开连接的时候，此时客户端发送了一个请求到服务器端。如果客户端发送请求和服务器断链同时发生，那么客户端将会认为服务器崩溃，为了避免发生这种情况，服务器端在断开连接前，将会向客户端发送一个 CloseConnection 消息到客户端。当客户端接收到此消息后，它将默认服务

器端忽略所有客户端端已经发送但未接收到应答的请求。因为这样，客户端将会(透明地)和服务端建立新的连接，并重发请求。

11.3.6、 消息错误消息 **MessageError Message**

如果 CORBA 应用服务器收到一条非 GIOP 的消息，它将返回一条 **MessageError** 消息告诉客户端“你向我发送一条垃圾信息!”如果是 CORBA 的客户端和服务端交互(除非使用的产品出现严重的错误)一般不会出现这样的情况。

你能通过非 CORBA 应用程序向 CORBA 服务器发送一个消息，强制服务器发送消息错误消息。例如：如果你知道 CORBA 服务器监听的端口和主机，你就能通过 telnet 连接那个服务器，当你在 telnet 中键入一个回车时，一个非 G-I-O-P 消息将会发送到 CORBA 服务器，并且 CORBA 服务器会返回一个消息错误消息(并有可能会关闭连接)。通过 telnet，能将会看见字母 G-I-O-P 后跟着一些不可见字符(这些就是错误消息消息的内容)。

11.4 GIOP重定向

当客户端，发送一条请求消息到服务器时，他将接收到一条应答消息。应答消息头包含如下的信息：

- 1、“正常应答信息”。这种情况下，应答消息体包含有 inout/out 参数的消息和如果存在返回值的话，返回值
- 2、“异常应答”。这种情况下，应答消息体包含一个 CORBA 的异常。当客户端的 CORBA 运行系统接收到这样的应答时，客户端将解组异常，并将其抛出到客户端的代码
- 3、“重定向信息”(实际在 CORBA 术语中是 `LOCATION_FORWARD`)。

这种消息告诉客户端目标对象在服务器进程中不存在,但是可以在其他的服务器中找到。这个消息的消息体包含告诉客户端对象真实存在地的 IOR。

CORBA 客户端的运行系统将透明的根据新 IOR 重发请求消息。此外客户端的所有未来的请求都会根据新 IOR 来发送。重定向只会在第一次请求过程中发生,以后的调用都直接和目标对象通讯

重定向应答消息被用于CORBA厂商的实现库(IMR),实现库我们在[第7章](#)做过详细讨论。IMR为CORBA提供了不少的灵活性,但是在初始化连接的时候使用IMR有可能需要付出一些代价或额外的成本。我在这里将讨论这个些额外的成本和减少这种成本的优化方案。让我们假设客户端的第一次请求中包含一个sequence<octet>参数,这个参数可能包含几兆字节(这个数据有可能是图像文件或声音文件)。当客户端收到一个重定向应答的时候,客户端将用新的IOR重传这个几兆的请求。明显地,传输如此大的数据两次,必然导致网络带宽的浪费。因为这个原因,CORBA定义了LocateRequest和LocateReply消息(在[11.3.2小节](#)讨论)。LocateRequest消息一个ping风格的消息,他向服务器询问“对象是否在这里”。回返的应答消息是LocateReply消息。这样目的是,CORBA客户端的运行系统将透明的发送一个LocateRequest在第一次真正发送请求消息前。这样就保证了在(潜在的巨大)请求消息被发送前CORBA的重定向就完成。

使用 LocateRequest 消息是可选的优化方案。一些老的 CORBA 产品,重来不发送 LocateRequest 消息。一些其他的 CORBA 产品总是在第一次真正发送请求前发送 LocateRequest 消息。可以想象,某个 CORBA 产品可以这样实现,在它第一此发送请求信息前,如果请求消息太小,他就去掉优化,否则它就发送 LocateRequest 消息。但是并不是所有的 CORBA 产品都以这种优化方式实现,因为这种优化掉 LocateRequest 消息的优化方案得到的性能对于真实

世界的部署并不重要。

11.5 活动连接管理 Active Connection Management (ACM)

CORBA的GIOP协议框架允许服务器和客户端的空闲的连接被关闭，并且如果客户端需要发消息，连接可以透明的被打开。这样就允许CORBA的实现优化网络的资源。CloseConnection消息([11.3.5 小节](#)讨论)提供了底层支持。CORBA标准并没有定义关闭空闲连接这种能力的术语。因为缺乏CORBA术语定义。一些厂商使用自己的名字。Orbix和Orbacus使用术语活动链接管理 *active connection management* (ACM)来指自动关闭空闲socket连接。其他的CORBA产品使用不同的术语来表示同一个概念。

CORBA 并不要求产品实现 ACM; 这是一个可选功能, 具有高质量、流行 CORBA 都应该实现它。CORBA 也并没有指定应该使用什么样关闭连接的算法, 在 Orbacus 中配置文件的某个条目用来指定应该被关闭连接的最大空闲时间。Orbix 使用不同的机制, 配置文件的条目来指定能打开的连接数, 一旦达到限制数, Orbix 将关闭 idle 时间最大的连接。其他的 CORBA 产品有可能使用其他的算法来关闭空闲连接。

11.6 服务上下文

服务上下文的概念是很容易理解的。然而, 这个名字并不是很直观的。服务上下文 “Service Context” 术语中的上下文 “context” 出现的原因是因为服务上下文是用来传请求消息和应答消息中的额外的上下文信息。服务上下文这个名词中的服务 Service 出现的原因是因为服务上下文一般都用于 CORBA

服务的实现，例如安全服务，对象事务服务。已经被明确被 OMG 公开的 API 中指明服务上下文也可以用于应用程序员。

```
module IOP {  
    ...  
    typedef unsigned long ContextId;  
    struct ServiceContext {  
        ContextId      id;  
        sequence<octet> data;  
    };  
    ...  
};
```

图 11.2 服务上下文的 IDL 定义

IDL 定义的服务上下文如 [图 11.2](#)。可以看出，服务上下文是一个简单结构，包含一个二进制数据缓存和一个整形值。整型值是表示二进制数据应该如何被解释。例如：某一个整型值标明对应的二进制数据应该OTS使用。另外一个整型值指定二进制数据包含安全信息；等等。公司可以联系OMG公司，要求他们唯一的整型值，用于指定他们特殊的需要。

每个请求消息和应答消息的头包含一个ServiceContext数组。除非是应用程序会使用，比如OTS，大多数情况请求和应答消息将包含一个空的ServiceContext数组。应用程序可以使用可移植拦截器 portable interceptor ([14 章](#)) 将服务上下文增加到待发送消息中，并且也可以检查接收消息中是否含包含指定的服务上下文。以这种方式，如果应用程序接收到服务上下文不是所期望的服务上下文，则该上下文将被忽略。

11.7 编码集约定 Codeset Negotiation

编码集 Codeset 是编码字符集 *coded character set* 的简称。编码集

可以是, 例如, Ascii 或者 Unicode。在早期的 CORBA 版本中, ISO Latin1 (US ASCII 扩展字符集)被硬编码用于传输 char 和 string 参数。然而, 如今 CORBA 已经可以支持宽字符 wide characters (IDL 的类型是 wchar 和 wstring)。CORBA 已经发展成不再需要硬编码来传输 wchar 或 wstring, 而是在连接建立的时候和客户端约定好传输时的编码方式。这个过程被称为编码集约定 Codeset Negotiation。编码集约定过程主要包含如下的几个步骤:

- 1、从服务器端输出的对象的引用 ([10 章](#)) 包含“联系细节”(例如端口号, 主机名, 对象键值), 但是同时也包含用于和对象通讯的编码集信息。对象引用使用指定服务器本地 native 加上一个编码集数组来传输指定的 char 和 string, 另一个编码集数组用来传输 wchar 和 wstring 类型。codeset 数组称为转换编码集 conversation codeset, 因为 CORBA 服务器端的运行系统可以通过本地编码集转换成这个数组中的编码集。
- 2、当应用程序导入一个对象引用, 他将对比在对象引用中的编码集和其 CORBA 运行系统理解的编码集。如果客户端的 CORBA 运行系统没有发现对象引用的编码集和自己的编码集有重合, 客户端运行系统将向外抛出不能和服务端对象通讯的异常。假设他能发现客户端和服务端有共同的编码集, 那么 CORBA 客户端运行系统决定选用什么编码集进行通讯。
- 3、当客户端发送其第一个请求消息到服务器时, 他使用服务上下文 ([11.6 小节](#)) 告诉服务器哪一个编码集将被选取。这个编码集称为传输 *transmission* 编码集

11.8 双向GIOP和IIOP Bidirectional GIOP/IIOP

GIOP/IIOP 的规范宣称, 他们是单向协议。客户端在通道 (通道是 GIOP 术

语，对于IIOP就是socket连接)上传输消息给服务器，服务器从同样的通道返回应答。这听起来好像是一个双向的通讯，因为消息是以两个方向进行传送的。然而为什么将其称为单向的原因是因为请求只能以一个方向进行传输。GIOP不允许服务器方在客户端用来传输请求的通道上传输请求到客户端。如 [图 11.3](#)。客户端打为了调用obj1 开一个通道到服务器，并且客户端传输一个回调对象引用作为参数供服务器稍后调用。如果服务器想调用回调对象上的方法，那么GIOP不允许服务器在已经打开的通道上传输请求，而必须打开一条新的通道来和客户端通讯。

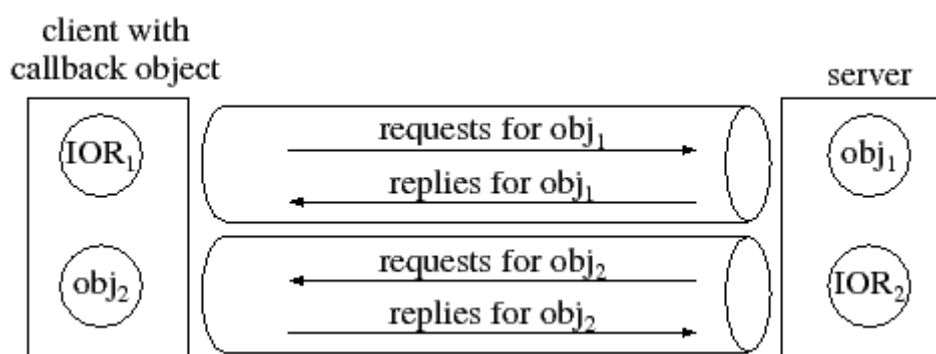


图 11.3 GIOP 的单向通道用法

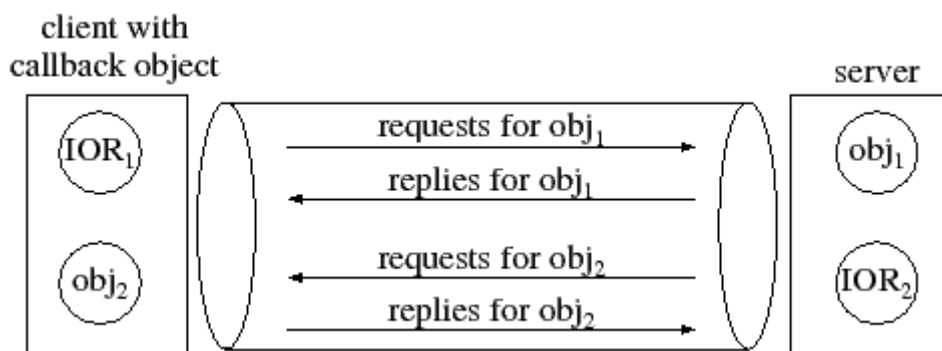


图 11.4 GIOP 双向通道用法

在定义 GIOP 规范的时候，在 OMG 中存在者一些争论，是否 GIOP 既可以支持单向通道，又可以支持双向通道。最后 OMG 决定 GIOP 使用单向通道，但是这是一个让人感到后悔的决定。在使用回调对象的情况下单向协议通道有如下两个缺点：

1、打开第二条通道浪费网络资源

2、在一些真实的部署环境中，在服务器和客户端中存在着一些防火墙。虽然客户端有可能很容易的通过服务器连接到服务器端，但是有些情况却无法建立从服务器端到客户端的连接

为了解决这两个问题，双向通道增强 GIOP/IIOP 的规范被引入到 CORBA2.4 规范中。

第12章、 corbaloc和corbaname URL

- [介绍](#)
- [corbaloc URL](#)
- [corbaname URL](#)
- [corbaloc的支持架构](#)
- [启动的交互性的问题](#)

12.1 介绍

URL 是基于万维网 world wide web(www)，以协议名字为开头，紧跟着“:”例如：“http:”，“ftp:”，“file:”。字符串化的对象引用以“IOR:”开头，所以和 URL 非常相似。

在CORBA早期的版本中，能作为参数传递给string_to_object()方法 ([3.4.2 小节](#))是字符串化的对象引用。CORBA如今已经发展成允许其他类似URL的字符串作为参数传递给string_to_object()方法。CORBA可以可选的支持“http:” “ftp:” “file:” 格式。这些字符串的语义提供了如何取得字符串化IOR的信息。(或则递归的调用其他URL最终获得字符串化的IOR)。

虽然“http:” “ftp:” “file:” 都是可选的，但是所有 CORBA 产品都

必须支持“corbaloc:” “corbaname:”，这两个是由 OMG 提供的 URL。这两个 URL 的功能：通过提供以具有可读性/可编辑性的方式来指定 IOR 可以被获取位置。

12.2 corbaloc URL

一些 corbaloc URL 例子如下：

```
corbaloc:iiop:1.2@host1:3075/NameService
corbaloc:iiop:host1:3075,iiop:host2:3075/NameService
```

第一个 URL 通过使用 1.2 版本的 IIOP 协议向主机 host1, 端口 3075 发送一个带有 NameService 参数的 LocateRequest 消息获得指定 IOR。

第二个 URL 和第一个 URL 有两点不同。第一，省略 1.2@ 这个符号的含义是使用 1.0 版本的 IIOP 协议。第二，这个 URL 中包含了两个 <host>:<port> 地址。一般来说可以指定任意个地址，用冒号进行分隔。这种形式提供了一种容错机制：LocateRequest 消息将发向地址列表中某一个地址，如果该地址无效，LocateRequest 将试图发向地址列表中的其他的地址。

corbaloc URL 中的各个部分都有默认值：

- 默认协议是 iiop
- 如果协议是 iiop 那么默认的版本号就是 1.0。在 corbaloc 指定最近客户端和服务端都能处理的 IIOP 的版本号是明智的。因为越新的版本号，功能就强大，处理客户端和服务器的交互效率就越高。
- 默认的端口号是 2809。这个端口号是由因特网编号分配机构 Internet Assign Number Authority (www.iana.org) 指定给 corbaloc 使用的。

CORBA 规范中指定了能被用于 corbaloc URL 中的两个协议。一个协议已经讨论过的 iiop，另外一个称为 rir 的协议，当你知道他是“定位初始引用”

resolve initial reference 的同义词的时候，你就不会这个名字感到陌生，这个协议指明对象引用应该通过调用 resolve_initial_reference() 方法，并根据此方法的参数名字指定获得对象引用。例如，如下的 corbaloc URL 指定一个 IOR 通过调用 resolve_initial_reference("NameService") 获得：

```
corbaloc:rir:/NameService
```

rir 的一个好处是，它使得 string_to_object() 具有了 resolve_initial_reference() 的功能。例如：应用程序员不用硬编码通过调用 resolve_initial_reference() 查找一个名字服务，而是通过硬编码从将配置文件内容或命令行参数带入到 string_to_object() 获得名字服务。如果这个参数是“corbaloc:rir:/NameService”那么将会默认程序员使用的是 resolve_initial_reference()，但是现在可以有指定参数是字符串 IOR 或包含 iiop 的 corbaloc URL 的灵活性。以这种方式，应用程序有额外的灵活性来查找 CORBA 服务。

rir 并不经常用于 corbaloc URL 中，它一般用于 corbaname URL 中。

12.3 corbaname URL

corbaname URL 是一种 corbaloc 指定 NameService 的一种特例。在 NameService 后紧跟着#和一个名字。一个例子如下：

```
corbaname::foo.bar.com:2809/NameService#x/y  
corbaname::, :host1, :host2, :host3/NameService#x/y  
corbaname:rir:/NameService#x/y
```

以上述字符串做为参数调用 string_to_object() 可以定位名字服务，并且在名字服务调用 resolve_str() 方法获取 x/y 的 IOR。如上例子所示，corbaname 能使用 iiop 和 rir 协议为参数定位名字服务。

12.4 corbaloc的支持架构

12.4.1、 corbaloc客户端支持

`string_to_object()`方法提供 corbaloc 和 corbaname URL 的内建支持:

- 如果传递给 `string_to_object()` 参数, 以 “IOR:” 为起始 , 那么这个方法参数是一个字符串化的对象引用, 并建立相应的代理 proxy(或 stub) 对象。
- 如果参数是以 “corbaloc:rir” 起头, 那么方法将调用 `resolve_object_reference()`, 并把 URL 中指定的名字作为此方法的参数。
- 如果参数是 corbaloc URL 并且使用 iiop 协议, 那么函数将打开指定主机端口的 socket 连接, 并通过该连接发送一个 LocateRequest 消息 ([11.3.2 小节](#)), 并以指定的名字作为消息头的对象键值 ([10.2.1 小节](#))
- 如果 `string_to_object()` 方法的参数是 corbaname URL, 那么嵌入在其中的 corbaloc 信息用来定位 NameService。 `string_to_object()` 方法将 URL 中 # 后的字符串作为参数调用名字服务的 `resolve_str()` 方法。
`resovle_str()` 方法 (译者注: `resovle_str()` 方法的参看 [4.2 小节](#)) 返回的 IOR 作为 `string_to_object()` 的返回值。

12.4.2、 corbaloc服务器端支持

CORBA 并没有标准化服务器端对 corbaloc URL 的支持, 甚至连术语都没有定义。这意味着, CORBA 产品要提供自己的私有机制, 通常也定义自己的私

有术语。例如:

- Orbix 的实现库以内建的形式支持 orbaloc URL, 实现主要以名字键值的方式 *named key*。named key 是一个 corbaloc 中名字组件到字符串化 IOR 的映射。itadmin 的 name_key 子命令用来创建, 显示, 列表和删除名字键值。默认情况下, Orbix 的实现库监听在 3705, 因此 corbaloc 应该是如下的形式:

```
corbaloc:<host-of-IMR>:3075/<name>
```

当 itconfigure 用来配置 Orbix 的域的时候, CORBA 服务对应的名字键值被自动的加入到域中。例如: 如果域包含有名字服务, 那么 NameService 的名字键值将被创建。

Orbix 很久以来, Orbix 没有暴露服务器端 corbaloc 对普通应用程序支持的 API。Orbix6.1 的 Service Pack1 是第一个暴露此 API 的版本。

- Orbacus 提供了一些私有的 API (在 BootManager 接口中), 这些 API 可以用来在服务器端开发对 corbaloc 的支持。这些 API 被用于 Orbacus 的实现库, 功能是实现名字到字符串 IOR 的映射。
- TAO 提供和另外一套的私有 API, 但是其功能和 Orbacus 相似
- omniORB 对 corbaloc 的支持依赖于将对象放置于预先定义好的 POA 中。

omniORB 同时也提供预写好的称为 omniMapper 服务器程序, 他监听在固定端口上完成配置文件中的名字和字符串化 IOR 的映射。

由上可知, 每一个 CORBA 产品对于服务器端 corbaloc 的支持都有不同的“感官”。因为这样, 通过在 CORBA 服务器里使用 corbaloc URL 来公告它所包含的对象的方式, 是难于被移植的。程序在考虑开发出可移植的 CORBA 程序使, 对于 corbaloc 服务器支持主要考虑的应该是以 CORBA 服务方式, 例如,

名字服务，通知服务，交易服务等等。

12.5 启动的交互性的问题

一个关于不同 CORBA 产品交互的问题的明确要求是：他们必须通过同样联机协议 (IIOP) 进行交互。然而光这样还不够，一个隐含的需求是，一个 CORBA 产品要能查找另一个产品提供的服务，如：名字服务，交易服务等 CORBA 服务。例如，一个 Orbix 的客户端如何连接 Orbacus 安装的名字服务。我们经常将这个问题称为启动问题 bootstrapping problem。corbaloc 和 corbaname URL 就是用来解决此类问题的。正如我们将讨论的一样。

CORBA 应用程序通过传如指定服务名为参数并调用 `resolve_initail_reference()` 方法连接指定 CORBA 服务, 例如名字服务, 事务服务, 通知服务, 等等。CORBA 的规范中并没有说明 `resolve_initial_reference()` 的实现细节。但是大多数 CORBA 实现都会去配置文件查找 CORBA 服务名到 CORBA 服务的字符串 IOR 之间映射, 然后将此 IOR 传递给 `string_to_object()` 方法。这个配置文件通常在安装和配置 CORBA 产品的时候设置。比如如果 Orbix 要使用 Orbacus 的名字服务主要要做的事情就是获取 Orbacus 名字服务的字符串化的 IOR 并将这个字符串保存在 Orbix 的配置文件中。下次当 Orbix 要使用到名字服务的时候调用 `resolve_initial_reference("NameService")`, 客户端将直接连接 Orbacus 的名字服务。这个机制能很好的工作, 但是这种机制存在一些问题, 因为字符化的 IOR 对于维护人员来说是不可读的。如果将 corbaloc URL 引入进来, 这个方法将更加简单。现在不是拷贝 Orbacus 名字服务的 IOR 到 Orbix 的配置文件, 而是拷贝 corbaloc URL 到配置文件。正是因为 corbaloc 是易

读的，所以使用 corbaloc 更容易将不同的 CORBA 产品组织在一起协同工作。

在某些特定环境下，更新配置文件中的 IOR 和 corbaloc URL 并不是很方便，比如，将名字服务更新为另一个 CORBA 产品的名字服务。为了解决这个问题。OMG 定义了标准 CORBA 产品都必须的实现两个命令行参数。

第一命令行参数如下：

```
-ORBInitRef <name>=<value>
```

例子如：

```
-ORBInitRef NameService=corbaloc::host1:3075/NameService
```

如果<name>为参数调用 `resovle_initial_reference()` 方法，那么 `resovle_initial_reference()` 方法将使用 `<name>=<value>` 中 `<value>` 来查找。命令行参数指定的信息将先于配置文件中的信息。你必须在每次运行应用程序的时候指定命令行参数，每次都这样使用，将显得比较累赘。然而命令行参数依然非常有用，例如，因为文件权限的约束，导致你无法修改 CORBA 应用程序的配置文件，那么你使用命令行参数将是非常有效的方式。

第二个命令行参数采用如下的形式：

```
-ORBDefaultInitRef <URL-up-to-but-not-including-final-"/">
```

例子如下：

```
-ORBDefaultInitRef corbaloc:iiop:1.2@host1:3075
```

```
-ORBDefaultInitRef corbaname::host1/NameService#x/y
```

调用 `resolve_reference_object("<name>")`，的结果是，`"/<name>"` 将会被加到 `-ORBDefaultInitRef` 参数字符串后面，并将拼接好的字符串以参数传递给 `string_to_object()` 方法调用。

`-ORBDefaultInitRef` 参数的目的是，建立集中的名字-IOR 映射存储，于是应用程序就可以以一个单一的 `-ORBDefaultInitRef` 命令行参数指向那个集中点。这样使用就比每一个应用启动时都带着几个 `-ORBInitRef` 方便。

和 `-ORBInitRef` 一样，每次启动应有都带 `-ORBDefaultInitRef` 参数是

比较累赘繁琐的，更方便的方式是将参数的信息在 CORBA 安装时写到相关的配置文件中。

如果 `-ORBDefaultInitRef` 和 `-ORBInitRef` 同时使用，`-ORBInitRef` 参数将覆盖 `-ORBDefaultInitRef` 参数。

第13章、 当前对象 Current Object

- [线程本地存储的概念](#)
- [Current Object 提供线程本地数据](#)

13.1 线程本地存储的概念

unix 的函数 `getpid()` 函数返回一个整型值的进程 id 唯一的标示系统内的进程。类似的，线程的函数库提供了相应的原语，有可能是 `get_thread_id()`，这个方法返回一个唯一的整型值标示当前线程。在不同的线程包中这个函数名称不同。一种被用于多线程编程的技术是提供一个查找表，并提供线程标示和其相关数据之间的映射。这种技术允许程序员为每一个线程关联一块相关数据内存区。这种技术被称为线程本地存储 *thread-local data* 或 *thread-specific data*。

13.2 Current Object 提供线程本地存储

CORBA 使用术语“当前 *Current*”来表示线程本地存储。事实上，CORBA 定义了一些不同类型的“当前对象 *Current Object*”。例如，有为安全的，有为交易的，还有其他为请求分发的。所有的当前接口 *current interface* 都定义在 IDL 文件中，他们都继承于一个空的基类型 `CORBA::Current`

```
module CORBA {
```

```

    local interface Current { };
    ...
};
module PortableServer {
    ...
    typedef sequence<octet> ObjectId;
    local interface Current : CORBA::Current {
        exception NoContext { };
        POA get_POA() raises(NoContext);
        ObjectId get_object_id() raises(NoContext);
    };
    ...
};

```

图 13.1 Current 类型的 IDL 定义

[图 13.1](#) 中是 `PortableServer::Current` 从 `CORBA::Current` 继承，并用于访问分发到当前线程的目标对象的相关信息。

当前对象通过指定一个相关的名字作为参数调用 `resolve_initial_reference()` 方法访问，这个方法在 [3.4.1 小节](#) 讨论过。例如：`PortableServer::Current` 单件被称为 `POACurrent`，因为你可以传递“`POACurrent`”为参数，调用 `resolve_initial_reference()` 方法获得。默认 POA 类型 ([5.6.4 小节](#)) 模型必须使用 `POACurrent` 对象，对于其他的 POA 类型模型，`POACurrent` 对象是可选的。

第14章、 可移植拦截器 Portable Interceptors

- [IOR拦截器](#)
- [请求拦截器](#)
- [PICurrent 对象](#)

早期的 CORBA 产品是单一的：当你买了一个 CORBA 产品，你可以使用该产品提供的所有功能，但是你（或则第三方公司）并没有机会去扩展产品功能。CORBA 现今已经发展到“插件 plug-in”的架构，即允许人们在 CORBA 产品中增加新的代码。第一个插件架构版本在 CORBA2.2 中定义为拦截器 *interceptor*。拦截器命名的原因是插件可以通过拦截一些 ORB 的功能，修改 ORB 的行为。但是不幸的是，CORBA2.2 的拦截器并没有规范化，这样导致了拦截器使用大量的不可以移植的 API。CORBA2.4 提供了一个更完整的定义，在 2.4 中拦截器被称为可移植拦截器 *portable interceptor*。

如今有两类可移植拦截器：IOR 拦截器和请求拦截器 request interceptor。在下面的章节中我将对这两类拦截器做一个简单的说明。更深层次的拦截器介绍在《Pure CORBA book》[[Bo101](#)]可以找到

14.1 IOR拦截器

当 IOR 被创建时，IOR 拦截器被调用。IOR 拦截器可以查找出对象的 POA 中那些策略被使用，并可以通过这个信息决定是否需要在 IOR 嵌入额外的 TaggedComponent ([10.2.3 小节](#))。例如，一个对象是否在分布事务中是由该对象是否使用了特定 POA 策略来决定的。如果这些 POA 策略被使用，那么作为分

布式服务的IOR拦截器将在IOR中加入相应的TaggedComponent。

14.2 请求拦截器

CORBA 中有两类请求拦截器 *request interceptor*: 一类是在客户端拦截客户端发送的请求和应答, 另一类是在服务器端拦截接收的请求和应答。明显地, 如果客户端和服务器端在同一个进程, 那么进程内就可以同时使用两类拦截器。服务器端和客户端的拦截器 API 具有类似观感(look and feel)。

一个请求拦截器在传输请求/应答/异常消息过程中的多个点被调用。请求拦截器完成如下的工作:

- 检查正在传输的请求/应答/异常消息的参数。拦截器检查消息, 并使用 DynAny API ([15.3 小节](#)) 将所有接收或发送的消息的诊断信息写入到日志文件
- 服务器端的拦截器, 能查找目标对象的对象标示和相应的 POA。拦截器通过这些信息记录什么对象什么时候被客户端访问过, 并垃圾回收未被访问的对象, 例如, 20 分钟未被访问的。
- 在向外发送的信息中增加服务上下 *service context* ([11.6 小节](#)), 在接收的消息中解包服务上下文。
- 检查目标对象的 TaggedComponent。例如, 被用于事务服务客户端拦截器就有可能检查目标对象是否是事务中的一个部分, 如果在事务之中, 拦截器, 将在向外发送的请求报文中增加事务服务上下文。

14.3 PICurrent 对象

可移植拦截器规范定义了一个称为PICurrent的当前Current对象 ([13 章](#)), 之所以称为PICurrent是因为必须通过传递一个“PICurrent”参数给

`resolve_initial_reference()`([3.4.1 小节](#)) 才能访问它。提供 `PICurrent` 的原因是，为 CORBA 提供一种服务上下文([11.6 小节](#))与可移植拦截器和应用级代码通讯的功能。

第15章、元 信 息 编 程

Meta-information Programing

- [什么是元信息编程](#)
- [类型码和接口库TypeCodes and Interface Respoitory](#)
- [Any和DynAny类型](#)
- [动态调用接口Dynamic Invocation Interface\(DII\)](#)
- [动态服务接口Dynamic Server Interface\(DSI\)](#)

15.1 什么是元信息编程

在许多的计算机编程语言中，当你编写程序，在编译期你知道你将维护什么数据类型并且编译器执行强类型检查来保证你能正确的访问那些数据类型。一些程序语言推迟类型检查到运行期。在这些语言中，对象关联着一些元信息 *meta information* (也被称为运行期信息)；语言的运行期系统使用这些信息来检查程序是否以合法的方式访问对象。一些语言使用运行类型检查甚至允许程序员通过使用对对象的元信息查询达到查找对象包含那些方法和什么名字和类型的实例变量。这种能力通常被称为自省 *introspection* 或反射 *reflection*。

Smalltalk 语言以运行期系统检查而著名。JAVA 提供了编译时期检查，同时也提供了 `java.lang.Class` 的 API 来允许运行期的检查。相反的，C 语言并不提供元信息，C++通过 *dynamic cast*，提供了对运行期检查非常有限的

支持。

15.1.1.1、元信息编程的使用

大多数的 CORBA 程序是由程序员将要维护的 IDL 数据类型和程序员将要实现调用的 IDL 接口的编译期 *compile-time* 信息写成。然而 CORBA 同样维护对象和数据类型的元信息，同时允许程序员写程序时使用这些元信息。CORBA 的这种能力很少被使用，但是使用元信息能写出一类非常重要的应用程序，例如：

- 一家公司想编写新的基于中间件技术应用程序，并且要和公司现存的基于老的其他中间技术的程序进行通讯。这种情况是非常常见的事情。这些连接不同中间件(或连接不同联机协议)的软件程序通常被称为桥 *bridge* 或者网关 *gateway*。

网关的基本概念使用某类技术的联机协议接收请求，执行参数的数据类型转换后再使用其他技术的协议来发送请求。为特定的 API 集合编写网关程序，通常是繁琐和高重复性的工作，而且如果这些 API 发生了变化，我们将不得不重新修改这个网关程序。然而如果被桥接 *bridged* 的中间件技术能提供元信息的话，那么程序员就可以实现一种通用的网关程序来接收该技术协议上任何请求，通过使用元信息可以确定请求方法的参数个数和类型，可以将该参数转换为第二种技术协议的格式，并通过使用第二种技术的协议将请求发送出去。

这种通用的网关程序通常比特定的网关程序要慢，但是你只要实现了这个网关程序，他就可以用于所连接系统任何 API 的网关。一些公司开发了 CORBA 到 DCE 的网关(一种老的中间件技术)，另一些 CORBA 厂家开发了 COM 到 CORBA 的网关程序，.NET 到 CORBA 的网关程序。这些网关程序都是通过高效的使用元信息实现的。

- 传统语言的调试器使用嵌入在可执行程序中符号表(元信息的一种)将变量

显示成可读的而非二进制数据块的格式。如果某家中间件厂商提供了元信息，那么对于公司来说将就能开发出无缝的调试的工具来满足中间件的需求，也可以通过元信息开发出用于软件开发过程中的其他工具。例如，当编写一个 CORBA 服务器时，有一个模拟客户端对服务器做测试是非常有效的，对于客户端开发也是一样。一些公司出售一些使用元信息，通过很少部分编码就能生成模拟客户端/服务器的工具。

CORBA 对元信息编程提供了几类不同的，但功能丰富的 API，这些 API 在下面的章节中讨论。

15.2 类型码和接口库 TypeCodes and Interface Respository

CORBA 使用类型码 `CORBA::TypeCodes` 来代表元信息。`TypeCodes` 的 `kind()` 方法返回一个枚举值来指定 `TypeCodes` 代表的是一个内建类型 (`long`, `boolean` 等等)还是用户自定义类型 (`struct`, `union`, `interface` 等等)。如果 `TypeCode` 代表用户自定义类型，还需要调用更多的方法来决定结构或联合等等所包含的成员的名字。

`TypeCode` 提供 ORB 运行系统通过这些元信息来决定作为复合数据结构中的域或组件占用原始内存块大小的字节数。然而，就更普遍元信息编程来说，光 `TypeCode` 提供的信息是不够的。例如: `TypeCode` 没有包含接口方法的返回值和参数信息。同时 `TypeCode` 也不提供模块 `Module` 中包含所有类型 (递归子模块) 的列表。为此，CORBA 对 `TypeCode` 辅于使用接口库 `interface respository (IFR)` 来提供更多的元信息。

你可以将 IFR 认为是一个关于 IDL 类型的元信息的数据库。这个数据库使用一个 CORBA 服务器对其进行封装；这个封装器就叫做 IFR。IFR 的 IDL 定义

提供了在 IFR 中查询和存储元信息的方法。IFR 组织元信息的方法和编译器组织语法树的方法类似，因为这样，在 IFR 中查询元信息类似于编译器遍历整个语法树。

虽然 CORBA 规范了 IFR 的 IDL 接口，CORBA 并没有规范如何管理 IFR。典型的情况是，某个 CORBA 产品会提供私有的命令行工具来解析 IDL 文件，并调用 IFR 的方法在 IFR 中存储元信息。这些工具根据 CORBA 产品的不同而不同。在一些 CORBA 产品，这个工具是一个独立的程序，另外一些产品则把这个工具作为 IDL 的一个命令选项。

IFR 提供的元信息包含 TypeCode 提供的元信息。然而，由于每一个 IFR 调用都是远程调用，所以 IFR 需要一定的网络成本。相反地，通过 TypeCode 查找元信息仅只是一个本地调用，所以 TypeCode 要更快。组织元信息程序结构的通常方式是使用 IFR 的 API 来导向特定类型在语法树中的叶子节点，然后获得相关的 TypeCode，只需要通过本地调用，就可以高效地做相关的类型检查

15.3 Any和DynAny类型

IDL 有一种被称为 any 的内建类型。它与 C/C++ 的 void *，java 的 java.lang.Object 具有类似功能：是一种传递不知道编译时期类型信息的数据的方式。当然，这样做的时候，在运行期你需要通过某种方式获得数据是什么类型的信息。在内部，any 包含着原始数据和一个表示数据类型的 CORBA::TypeCode

any 类型不象 C/C++ 的 void * 类型，java 中 java.lang.Object 那样被频繁的使用。换句话说，any 被使用于完成特殊编程任务，而非那些一般的普通的应用。

在IDL中，any经常用于定义一个存储名字和任意数据的类型，如 [图 15.1](#)

```
struct NameValuePair {  
    string    name;  
    any      value;  
};
```

图 15.1 使用类型 any 定义名字和值

IDL编译器会生成any类型和特定IDL类型的三个方法，这三个方法是：

1) 插入insert：即将特定的IDL类型转换成any类型，2) 查询：查询any中的数据，3) 解出extract：将any类型安全转换成any所代表的特定IDL类型，解出extract和插入insert是互逆的两个方法。这些方法被编译生成stub代码 ([1.4.5 小节](#)) 供应用程序使用。然而这些用来插入到，解出于，某个any的API只能用于由该any嵌入的类型生成的代理代码编译成的应用程序。

(译者注：例如在 Orbix 中，有一个如下的自定义结构 AStruct，

```
struct AStruct {  
    string str;  
    float number;  
};
```

如果要将 AStruct 插入到 any 中，那么必须使用 idl 编译器的 -A 参数。通过此参数，一个如下方法就被生成

```
// C++  
void operator<< = (CORBA::Any& a, const AStruct& t);
```

这样在程序里面就可以以如下的步骤插入一个 AStruct 到 any

```
// C++  
CORBA::Any a;  
AStruct s;  
// Initialise s.  
a << = s;  
)
```

如果应用程序想不通过使用生成的代理代码API而将某个数据类型插入到

any中，就必须先将any转换成DynAny，DynAny是一个本地local接口([9.1 小节](#))，它处于继承层次的最顶层，是一个基类。对于每一种IDL构件，DynAny都有不同的子类，例如，DynStruct，DynUnion，DynSequence等子类。

DynAny 上的方法允许程序员递归的钻取 DynAny 所包含的复合数据结构，按照这样执行，就可以将复合类型解组成一个个独立内建类型。DynAny 上的方法也可以用于递归的由内建类型构成一个复合类型。

DynAny 可以转换成一个 any 类型，any 也可以转换成 DynAny。这点是非常重要的，因为大多数的 CORBA API 都只操作 any 类型而非 DynAny 类型。

15.4 动态调用接口Dynamic Invocation Interface(DII)

动态调用接口 *Dynamic Invocation Interface* (DII)是一套 API，这套 API 允许客户端应用程序直接调用对象引用，而无需客户端程序与相关的 IDL 接口和方法参数生成的代理代码编译在一起。一个基于 DII 的客户端程序完成如下的几件典型的事情：

- 客户端从某处获得对象引用
- 客户端调用对象引用上的get_interface()方法访问IFR([15.2 小节](#))中的元信息，并且找到对象的相关要调用方法的参数和返回值信息。
- 客户端使用 DynAny 来创建这些参数，然后将 DynAny 转换成 any 类型
- 目标对象引用，被调用的方法的名字，所有参数的值和方向等信息被加入到一个 CORBA::Request 伪对象中。然后这个调用对象上 invoke()方法
- 客户端使用 DynAny 检查 inout/out 参数和返回值。

DII API用于没有客户端需要调用的IDL接口的信息情况，这意味着，客户端应用程序没有太多的硬编码那些将会被用于远程调用的参数，或则甚至哪个将会被调用方法的“业务逻辑”，这些基于DII的远程调用是由外部元信息来驱动

的。使用这种DII应用程序的例子是在 [15.1.1](#) 中简单讨论过的网关和测试程序客户端。我现在将讨论的关于基于DII的测试客户端的更多细节。

在编写一个客户端时，有一个能执行特定测试的模拟客户端是非常有用的。可以用 DII 来做一个图形界面的通用测试客户端，通常这个客户端以如下的方式工作：

- 用户指定测试客户端将要使用的对象引用。这个对象引用可以从名字服务(4章)或则从一个文件中获得
- GUI程序从对象引用中提取库标示`repository id`(译者注：[9.4 小节](#))信息。并通过库标示从IFR中获取对象的提供的方法信息。GUI工具的菜单显示了有可调用方法的菜单。
- 用户从菜单上选择要调用的方法，GUI 工具接着显示对话框提示用户输入 `in` 和 `inout` 参数的值。如果参数是一个符合类型的参数那么 GUI 客户端使用 `TypeCode` 递归下钻的到每一类型上并为每一个复合类型显示对话框。复合类型的值通过 `DynAyn` 的 API 建立。
- 当用户提供完所有的参数值，GUI 客户端创建一个 `CORBA::Request` 对象，并调用对象上 `invoke()`，当方法返回，GUI 显示所有的 `out`，`inout` 参数和返回值

利用这样的 GUI 测试工具来记录下用户输入参数值，开发商可以用这些记录生成脱离于 GUI 交互的回归测试 *regression testing* 程序。

15.5 动态服务接口Dynamic Server Interface(DSI)

动态服务调用 `Dynamic Server Interface(DSI)`经常被描述为 DII 的服务端的等价物。DSI 是一套允许服务端应用程序处理没有相关框架代码

skeleton code 处理请求的一套 API。类似于 DII，DSI 可以用建立网关和测试应用程序。例如基于 DSI 的测试服务器可以接收请求并给 inout 和 out 参数赋上随机的值或从配置文件或数据库读取值。

第16章、 CORBA 消 息 CORBA Messaging

- [服务质量Quality of Service\(Policy Objects\)](#)
- [异步消息接口Asynchronous Messaging Interface](#)
- [时间无关调用Time Independent Invocation \(DII\)](#)
- [参考读物](#)

CORBA 的消息规范分为三个独立但互为补充的主题，每一个主题将在以下的章节中进行讨论。

16.1 服务质量Quality of Service(Policy Objects)

CORBA 的第一个版本中没有为程序员想要在其应用中达到什么样的服务质量定义任何可移植方法。而是由各个 CORBA 厂商各自提供私有 API 来达到服务质量的目的。

经过一段时间的发展，CORBA 已经发展到为程序员提供在其应用程序中指定服务质量的机制。在 CORBA 的术语中服务质量的值被称为策略对象 *policy object*。策略对象首先和 POA 的规范 ([5.5 小节](#)) 一起被引入。POA 规范中定义 7 种类型的策略对象和分别创建这 7 种类型的方法。OMG 意识到：以后每新增策略对象就相应增加一个新的方法到现存的接口中的这种方式来增加策略对象是非常不实际的做法，因为这样会带来版本噩梦 *versioning nightmare*。消息

规范Messaging specification(在POA规范之后被引入)定义了一种更通用的机制来引入新的策略对象到CORBA中。这个机制包含下面几点:

- 策略对象的基类型 `CORBA::Policy` 是一个本地接口 `local interface`([9.1 小节](#)), 所有策略对象都继承于该接口
- ORB 包含一个叫做 `create_policy` 的方法, 此方法可以用来创建 `CORBA::Policy` 的子类型, 这个方法指定两个参数, 一个参数是类型 `type` 另外一个参数策略对象的值 `value`。例如, 类型是 `RELATIVE_RT_TIMEOUT_POLICY_TYPE`(此值是一个整型常量代表 `CORBA::Policy` 的子类型 `RelativeRoundtripTimeoutPolicy`), 那么值 `value` 就应该是超时的时间。类型 `type` 参数总是一个 `typedef` 的整型常量, 并且总有着一个该整型常量到一个 `CORBA::Policy` 子类型的一一对应的映射关系。然而 `value` 参数总是根据不同的子类而不同。因此, `value` 参数应该是 `any` 类型; 这就为 `value` 提供了嵌入整型, 字符串型或其他任意 IDL 数据结构灵活性。在 CORBA 内部, `create_policy()` 使用它的参数来创建 `CORBA::Policy` 对应子类型实例。

这样创建策略对象的程序步骤具有繁琐的缺点, 但是同时也带来了两个好处。

第一个好处是这种创建机制是一种更普遍的创建方法。这意味着 `create_policy` 可以处理更多未来有可能被引入的 `policy` 类型(不管是 CORBA 兼容的 `policy` 类型还是厂家私有的 `policy` 类型)

第二个好处是, 一些 `policy` 类型的使用将会对 IOR(10 章)中嵌入的信息或服务上下文(11.6 小节)产生影响, 由于每一个策略对象的类型都有一个与之对应的整型常量, 这就意味着, 这个值可以嵌入到 IOR 或服务上下文中, 因为

这样，所以使用策略对象并不会产生很大的网络成本。

除了提供更普遍的机制来定义策略类型外，CORBA 消息规范同样也定义一些可用来控制远程调用服务质量的特定策略，这些策略包括：

- CORBA 允许空闲 socket 连接被关闭(节省网络资源)和重建已被关闭的连接。RebindPolicy 类型用来指定客户端是否可以允许透明的重建已被关闭的连接
- IDL 定义的方法前可以使用 oneway 关键字，这个关键字的意思是，这个方法是否可以被异步的调用，然而，早期的 CORBA 版本并没有明确定义清楚 oneway 调用的语义，所以在不同的 CORBA 产品中，oneway 具有不同的含义。SyncScopePolicy 类型如今用来精确的表示 oneway 调用的含义。
- CORBA 定义了两类优先级策略，一类用于指定发送请求的优先级，一类用于指定接收应答的优先级。
- CORBA 中存在多种启动策略用来阻止请求/应答信息在特殊时间端前被发送。CORBA 中也有各种各样的时间策略用来取消发送那些在特定时间内不能被发送的请求/应答消息。
- CORBA 还有许多策略用于连接请求/应答路由器 ([16.3 小节](#))

消息规范同样定义了一个服务上下文([11.6 小节](#))用来在客户端和服务端保存域请求或应答相关的策略信息。客户端如果指定了远程调用超时时间，并且 CORBA 服务器端运行系统只有在请求还在队列中而未在服务器内分发的时候才能取消请求；服务器运行系统不能取消已经被分发到目标对象的请求。

16.2 异步消息接口 Asynchronous Messaging Interface

通常，IDL 方法提供阻塞式的请求-应答语义，即，当客户端远程调用服务器上的对象，客户端将等待到服务端返回应答。这种语义适合很多应用程序，但

是一些程序能从非阻塞的请求-应答语义中获益。

长时间以来，开发人员有两种选择实现非阻塞的请求-应答

1、客户端可以创建一个新线程，并让那个线程做阻塞调用。以这种方式，客户端的主线程就无需阻塞。这种方法不能用于单线程客户端。同样，甚至客户端是多线程，由于这种方法线程创建的成本，所以本方法不适用存在多个非阻塞调用场景下。

2、客户端使用DII([15.4 小节](#))的send_deferred() API。然而，DII的使用非常繁琐的，所以很少有程序员使用这种方法。即使使用这种方法，客户端不得不采取周期性轮询检查应答是否到来。轮询需要浪费很多CPU时间。另一方面，不经常轮询又会造成应答延迟。

CORBA 经过多年的发展，如今已经可以提供不同的非阻塞请求-应答机制，这种机制被称为异步消息接口 Asynchronous Messaging Interface (AMI)

AMI 是 CORBA 的可选部分，并不是所有的 CORBA 产品都支持它。同样，AMI 要求在 IDL 编译器生成的代理类中增加额外生成的方法。不幸的是这样做会打破 JAVA CORBA 产品二进制代码的可移植性。由于这个原因，至少非 JAVA 的 CORBA 产品可以支持 AMI，直到 IDL 到 JAVA 的映射能支持 AMI 的映射后，JAVA 才开始支持 AMI。因为这些原因，如果你考虑在项目中使用 AMI 那么检查你的产品是否支持 AMI 是非常重要的。

如果 CORBA 产品支持 AMI，那么它如下的方式工作：

- 对于每一个 IDL 方法，foo()，IDL 编译器生成(1)一个阻塞调用方法 foo()，(2)一个非阻塞调用方法 sendc_foo() (3)另一个非阻塞调用方法 sendp_foo()。第一个方法所有的 IDL 都会生成，第二、第三个方法是支持 AMI，注意，对于 IDL 来说，AMI 默认是被禁用的，因此你必须传

递相应的命令行参数给 IDL 编译器告诉其生成 AMI 方法。你的 CORBA 产品的厂商提供的文档将会告诉你使用什么命令行参数。

- sendc 版本的方法需要 in 和 inout 参数，他也可以再加一个回调对象 ([1.4.2.2 小节](#)) 作为参数。回调对象的方法将在稍后被带着 out/inout 参数和返回值去调用，客户端程序员有责任将这个回调对象实现成一个 Servant 并在 POA 中激活他。注意到回调对象同时也是一个 CORBA 对象是非常重要的，因为这个对象可以在其他的应用程序中执行。实际上，一个应用程序能发送请求到服务器，而应答处理则有可能在同一个客户端或其他的应用程序中。
- sendp 版本需要 in 和 inout 参数，这个方法返回一个轮询器 poller 对象。客户端能周期性调用轮询器对象上的方法来判断应答是否返回，如果应答返回，应用程序就得到 out/inout 参数和返回值。轮询器对象的实现是有 IDL 编译器生成的。

回调或轮询的机制都是完全的在客户端的运行系统中实现的，服务器端的应用程序，并不会意识到，受其采用什么方式处理请求影响，开发人员注意到这一点是非常重要的。

大多数的实现 AMI 的 CORBA 厂商都支持回调模型，而不支持轮询模型模型。这是因为多数开发者的观点认为，回调模型要比轮询模型更为高级，所以 CORBA 的实现者很少有动力是实现轮询模型。开发人员注意到这点也是重要的。

16.3 时间无关调用 Time Independent Invocation (DII)

GIOP ([11 章](#)) 是一个同步协议。这意味着异步的 CORBA 消息架构应该被安置于 GIOP 协议栈的最顶层。例如：回调和轮询模型 ([16.2 小节](#)) 通过 CORBA 客户端运行系统置于 GIOP 的最顶层。另一个 CORBA 消息规范中值得关注的方面是时间无关调用 *Time Independent Invocation* (TII)，TII 的内容将在下面进

行解释。我们假设客户端应用程序发送一个请求到服务器，但是客户端在其未收到请求前就被终止。TII就意味着应答消息被以某种形式持久化到介质中直到客户端程序重启，应答消息将被重新发回客户端。

CORBA 的 Messaging 规范通过引入另外一个称为消息路由器 message router 的部件来提供对 TII 支持。消息路由器 message router 是一个服务器的代理人角色，即，服务器接收到消息后，将该消息再转发到另外一台服务器上。

IOR ([10 章](#)) 包含某一 CORBA 对象的联系细节。然而，在 IOR 中可以有一个 TaggedComponent 包含一系列嵌入路由器 IOR。这一系列的路由器 IOR 反映的是一个代理链，从第一个路由器到下一个路由器，直到最后到目标服务器。如果客户端使用了这样的 IOR，但是客户端没有使用相关的路由策略，那么请求以默认的方法，直接发向服务器。然而如果客户端使用不同的路由策略的话，那么客户端的运行系统使用不同的机制来发送请求：客户端将请求发向路由器 1，回调对象的 IOR 也跟随请求发送到路由器 1。每一个路由器代理请求到下一个路由器，最终最后一个路由器将请求发送给目标服务器。对于服务而言，最后一个路由器，它就认为是他的客户端，因此服务器将其应答消息返回给此路由器，此路由器按照路由器链返回应答信息。最靠近客户端的路由器发送应答信息，并调用原客户端提供回调对象的相应方法。

路由器所提供的“增值”服务就是每一个路由器可以存储请求和应答信息到持久化存储器，例如数据库，如果某一个进程，比如说，客户端，某一个路由器或服务器在发送请求和应答过程中终止，那么这些持久化信息在这些进程重启后会被重发。

如下几种策略被用于消息发送的 QoS 控制：

- RoutingPolicy 策略，通过检查在 IOR 中是否嵌入路由 TaggedComponent 以决定是否以正常或者通过路由器发送请求。

ROUTE_NONE 值意味着请求直接发向目标对象。ROUTE_FORWARD 值意味着

请求将经过路由器。ROUTE_STORE_AND_FORWARD 意味着路由器将通过持久化信息提供 TII 功能。

- MaxHopsPolicy 指定发生在消息传送过程中的“跳数 hops (从一个代理路由器到另外一个)”的上限。
- QueueOrderPolicy 告诉路由器按照什么顺序转发消息。例如，他们可以转发消息以先进先出 (FIFO)，以每个消息优先级，或则以“最后期限 deadline” (将要超时的消息) 来转发消息

16.4 参考读物

关于AMI的细节可以在一些期刊杂志上有更多描述[[SV99a](#) [SV99b](#)]

第17章、 私有容错技术

- [关于容错的基本问题](#)
- [实例产品](#)
- [其他问题](#)

CORBA 规范的一些底层的功能模块定义使得 CORBA 具有一定的容错能力。

然而，长时间以来，CORBA 规范并没有明确的提出应该提供什么容错机制和怎么管理容错机制。这样就导致了许多 CORBA 产品自己提供了私有的易以使用但功能有限的容错机制。最近，OMG 定义了一个(可选的)容错规范，这个规范提供更标准化、功能更强大(但更复杂的)的容错机制架构。

本章讨论一些CORBA实现提供的私有容错机制。[17.1 小节](#)介绍了一些在讨论容错机制之前必须被阐明的问题。[17.2 小节](#)对几种CORBA产品的私有容错机制做了概览。最后，[17.3 小节](#)讨论一些CORBA容错使用过程中需要注意的问题。

CORBA中关于新的容错规范在下章讨论。

17.1 关于容错的基本问题

17.1.1、 复制粒度Replication Granularity

关于容错的一个重要方面就是复制的要求。因为 CORBA 是一个面向对象的中间件技术，所以在 CORBA 中的容错机制就是复制对象的需求。虽然这么说也对，但是对象是存在于进程(POA 中)，所以在实践中，复制对象就包含了对服务器的复制。而且大多数的 CORBA 产品支持的复制的粒度是服务器的复制，而非单个对象的复制。这是因为粗粒度的赋值大量减少了 CORBA 运行系统需要维护的复制单元的信息，并且增加了复制对象应用程序的扩展性(译者注：我们将复制后的对象称为副本对象)。

17.1.2、 副本对象的联系细节Contact Details for a Replicated Object

如果客户端连接某一个对象失败，客户端将试图去连接对象的另外一个副本对象。为了做到这样，客户端需要能访问到对象的所有副本对象的“联系细节”。正如我将讨论的那样，IOR 和 IMR 的灵活性为副本对象的访问提供基础的支持。

没有 IMR 情况下的副本服务器部署

就像在 [10 章](#)讨论的那样，IOR中可以包含很多的联系细节。这就使得——一个IOR可以包含隔离的多个副本对象的联系细节。客户端会试图用IOR中某一个联系细节和副本对象进行通讯。如果失败客户端会切换到IOR中另外一个联系细节。CORBA并没有指定那一个联系细节应该第一个被尝试。通常都是按照在IOR中的出现顺序

在 IMR 下部署复制服务

就像在 [7.2.3 小节](#) 讨论的那样，一些CORBA允许副本服务器在IMR中注册，如果按照这样做，那么IOR中就包含IMR的端口号和主机名；当客户端发送第一个请求到IMR，IMR会重定向客户端到目标对象的某一个副本服务器。客户端的第二请求也会发向此服务器。如果客户端和服务服务器的通讯失败，客户端将用原来IMR的端口和主机信息，IMR将重新定位到目标对象的另外一个副本服务器。

重复 IMR

为了阻止 IMR 成为新的单点故障源，IMR 需要有 IMR 服务器副本。通过重复 IMR 部署的 CORBA 对象的 IOR 中包含着所有重复 IMR 的主机和端口号细节。因为这样，客户端的第一个请求会送到副本 IMR 的某一台服务器上。如果客户端不能和副本 IMR 通讯，那么客户端 CORBA 运行系统将切换到另一个副本 IMR 服务器。这个 IMR 服务器将重定向客户端到目标对象服务器上。

17.1.3、 使用持久化POA

持久化PERSISTENT和临时TRANSIENT的策略已经在 [6.1.3 小节](#) 介绍过。简单回顾一下，POA([5.5 小节](#))是Servant(就是在宿主语言中代表CORBA对象的对象)的容器。创建POA的策略将应用在POA的所有Servant中。如果POA有PERSISTENT策略，那么此POA内的所有对象/Servant的IOR无论在服务器进程在停止时或启动后都总是有效。相反地，如果POA有TRANSIENT策略，那么POA内对象/Servant的IOR将只在服务器进程运行期有效。一旦服务器停止，这些IOR将自动无效。

17.2 实例产品

17.2.1、OmniORB

omniORB 没有提供实现库 Implemenet Repository 的实现，所以我们不需要讨论通过 IMR 来部署重复服务器。而是，本节将讨论 omniORB 在没有 IMR 情况部署副本服务器。

omniORB 存在两个配置变量，一旦这两个变量联合使用，就可以部署重复服务器。

- `endPoint = giop:tcp:<host>:<port>`

等号右边 `giop:tcp:` 部分指定 IIOP 协议被使用 (omniORB 同时支持其他协议); 在后面是指定监听的主机和端口。一些读者认为明确为服务器指定运行主机的信息是多余。但是对于服务器运行在多主机的机器上是有用的，所谓多主机即：此服务器机器是可能同时存在多个主机名或者多个 IP 地址的情况。服务器监听在特定的主机，端口上，同时主机:端口信息也会嵌入在服务器对象的 IOR 信息中。

- `endPointNoListen = giop:tcp:<host>:<port>`

这个参数的值格式和 `endPoint` 值的格式相同。但是服务器并不会监听在这个指定的主机:端口上，但是这个信息也会被嵌入到服务器对象的 IOR 信息中。

让我们假设，你想要三个副本的服务器：一个副本运行在 `host1.foo.com` 端口 5000，另外一个副本在 `host2.foo.com` 端口 6000，第三个副本将运行在 `host3.foo.com` 端口 7000。那么 omniORB 配置文件中 `endPoint` 和 `endPointListen` 的在第一台机器上的配置应该是：

```
#副本 1 下 omniORB 配置文件部分
```



```
endPoint = giop:tcp:host1.foo.com:5000
endPointNoListen = giop:tcp:host2.foo.com:6000
                  = giop:tcp:host3.foo.com:7000
```

为此服务器副本 1 监听在 host1:5000 端口上, 但是其输出的 IOR 将会包含 host1:5000, host2:6000, host3:7000 (所有都在 foo.com 域中)。在第二服务器副本中的配置文件信息如下

```
# Extract from omniORB configuration file for replica 2
endPoint = giop:tcp:host2.foo.com:6000
endPointNoListen = giop:tcp:host1.foo.com:5000
                  = giop:tcp:host3.foo.com:7000
```

相似的服务副本 3 配置文件信息如下:

```
# Extract from omniORB configuration file for replica 3
endPoint = giop:tcp:host3.foo.com:7000
endPointNoListen = giop:tcp:host1.foo.com:5000
                  = giop:tcp:host2.foo.com:6000
```

总的效果就是每个服务器监听在自己的端口上, 但是输出的 IOR 中却包含所有服务器副本的信息。

要注意到 omniORB 是通过 OMNIORB_CONFIG 环境变量指定配置文件所在位置的。如果你想在同一台机器上运行几个服务器副本那么你需要几个配置文件 (每个服务副本一个), 并且在启动时分别指定不同的 OMNIORB_CONFIG 环境变量。如果你不愿使用多个配置文件, 那么你可以通过将 endPoint 和 endPointNoListen 信息以启动参数的形式传递给服务器副本。服务将命令行参数作为参数传递给 ORB_init(), 正如我们在 [3.2.3 小节](#) 讨论的一样。

17.2.2、Orbix

orbix 允许使用 IMR 或不使用 IMR 部署副本服务器, 因此我将讨论这两种部署形式

17.2.2.1、 通过IMR部署重复服务器

Orbix的管理是通过实用工具itadmin的子命令完成的。每一个子命令只执行一部分工作，所以如果你要完成有效的工作，必须执行几次itadmin命令，例如，用IMR注册一个Server。然而itadmin内建了一个解释器用来解释一种叫做Tcl的开源脚本语言(发Tickle的音)。因此就可以通过编写一个Tcl脚本完成一系列itadmin的可能。CORBA 工具包CORBA Utilities Package [\[Mch\]](#) 中的《使Orbix管理变得简单》的章节介绍几个有用的itadmin的Tcl脚本。其中的一个脚本，orbix_srv_admin，简化了包括注册服务器的工作。

当使用 orbix_srv_admin 注册服务器，你能指定哪一个主机或哪一些主机来运行服务。如果你指明了一系列主机，那么你注册的服务器就是重复服务器。

如同 [17.1.2](#) 讨论的。副本服务器的对象的IOR包含了IMR的端口和主机名；当客户端第一次发送请求到IMR，IMR重定向客户端到目标对象的一个副本。客户端以后将发向同一个副本。如果客户端和服务器副本通讯失败，客户端的CORBA运行系统自动切回IMR的主机和端口，IMR可以重新定位客户端到另外一个目标对象的服务器副本。

itconfig工具用于建立IMR的初始化配置，其中的一个选项可以方便创建IMR副本(这样IMR就不会成为单点故障源)。如果你这样做，如同 [17.1.2 小节](#) 讨论的那样，通过重复IMR部署的服务器的IOR中就包含所有IMR副本的通讯信息。因为这样，客户端第一次是请求就会发向IMR的副本之一。如果客户端不能和IMR进行通讯，客户端的CORBA运行系统会切换使用另外一个IMR副本。于是IMR会重定向客户端到指定的服务器(也可能是服务器副本)。

orbix 保留着服务器当前的状态信息，并且它可以自动地重启已死的服务副

本。当客户端第一发送请求到 IMR, IMR 重定向客户端到正在运行的服务器副本。

IMR 可以使用轮叫调度 *round-robin* 和随机 *random* 策略选择服务器副本。

通过这样, 容错的机制同样也会为一个客户端提供负载均衡策略

17.2.2.2、 不使用IMR, 部署重复服务器

默认情况下, 带持久化策略的orbix服务器通过orbix的IMR来部署, 如果你想不通过IMR部署持久化的orbix服务器, 就必须使用某些orbix的私有API。PoaUtility类 (CORBA 工具包 CORBA Utilities Package [\[MCH\]](#) 的 *Creation of POA Hierarchies Made Simple* 章节讨论)封装了这些私有API的功能, 并且使得选择不同的部署选项更简单, 例如通过命令行参数。

当不使用IMR部署持久化POA服务器, 服务器就需要监听在固定的端口之上。固定端口可以在orbix运行配置文件中指定, 如同 [17.1 小节](#)讨论的。这个配置文件有如下几点需要注意:

- orbix 的配置被放在某一个 (有可能潜在是嵌入的) 范围 scope。例如, BankSvr 是一个范围, BankSvr.replica_1 是嵌入在 BankSvr 范围中三个子范围之一。当 orbix 应用程序启动, -ORBName <scope> 命令行参数用于指定范围, 应用程序通过这个参数获取其配置文件
- orbix允许不同的POA在一个服务器监听同样或不同的端口。私有的API用来实现这个灵活性的功能, 但是PoaUtility类 [\[MCH, Ch.5\]](#)封装了这些API的功能, 并且允许每个POA 管理器 ([5.7 小节](#)) 使用不同的端口。[图 17.1](#) 中的例子假设服务使用了PoaUtility类, 并且有两个POAManager, 一个用标记core来标示, 另外一个用标记admin来标示
- 具有<lable>:iiop:addr_list 形式的变量指定了一个 host:port 列

表字符串，其中可以以“+”作为可选的前缀。服务器将监听在任何一个没
“+”前缀的地址上。然后所有的 host:port 都会嵌入在服务的
(<table>POA 管理器控制的 POA 中的)对象的 IOR 中

如果服务器副本用 -ORBName BankSvr.replica_1 命令行启动那么服务
监听在 host1:5000 和 host2:5000 端口上(每个 POA 管理器一个端口)，但
是服务器也会嵌入其他的 host:port 地址在输出的 IOR 中。启动第二副本以
-ORBName BankSvr.replica_2 参数和启动第三个副本以 -ORBName
BankSvr.replica_3 参数,总的效果就是每个服务器监听在自己地端口之上，
但是输出的 IOR 包含所有服务器副本的 host:port 信息

```
BankSrv {
  replica_1 {
    core:iiop:addr_list = [ "host1:5000",
                             "+host2:6000",
                             "+host3:7000"];

    admin:iiop:addr_list = [ "host1:5001",
                              "+host2:6001",
                              "+host3:7001"];
  };
  replica_2 {
    core:iiop:addr_list = [ "+host1:5000",
                             "host2:6000",
                             "+host3:7000"];

    admin:iiop:addr_list = [ "+host1:5001",
                              "host2:6001",
```

```

        "+host3:7001"];
};
replica_3 {
    core:iiop:addr_list = ["+host1:5000",
        "+host2:6000",
        "host3:7000"];

    admin:iiop:addr_list = ["+host1:5001",
        "+host2:6001",
        "host3:7001"];
};
};

```

图 17.1 orbix 重复服务器的容错配置文件

orbix 的机制和 omniORB 的机制很相似，但是有以下几点不同：

- 某一个 omniORB 服务器中的所有对象通过一个端口来访问，orbix 允许服务器一些对象以一个端口来访问，其他对象用另一个端口来访问。orbix 提供这点好处主要是在一个端口对于客户端可以在通过防火墙来访问，而另外一个端口对客户端来却有约束不能通过防火墙来访问。这样，就提供了一个简单的安全机制。然而这中安全机制对某些公司来说只是非常基础的需求。CORBA Security Service ([23 章](#)) 提供更灵活的方法，具有更广泛的应用性。
- 如果几个 omniORB 应用程序跑在同样的机器上，因为不同的 omniORB 监听在不同的端口上，因此必须为不同的 omniORB 应用程序准备不同的配置文件。相比而言，分离不同的范围 scope 的功能允许为不同的 orbix 应用共享同一个配置文件，这样就减少了配置文件数目，并简化配置文件的管理。orbix 允许配置信息存在在文本文件或配置库 configuration

repository 中，配置库是一个通过 CORBA 服务器封装访问的数据库。使用配置库，可以将管理信息集中在某一个地方。集中化管理信息，简化了配置管理工作。

上述对比只是不使用 IMR 部署重复服务器 orbix 和 omniORB 的微小差别，orbix 和 omniORB 的容错的主要差别是，orbix 支持通过 IMR 部署重复服务器（正如在 [17.2.2.1 小节](#) 讨论的那样）

17.2.3、 orbacus

orbacus 在其运行系统中不提供任何容错能力，orbacus 提供的称为 iormerge 的命令行工具。一个 iormerge 使用的例子如下：

```
iormerge -f replica1.ior replica2.ior replica3.ior >new.ior
```

通常，iormerge 解释他的命令行参数的字符串化的 IOR，然而，如上所示，如果你使用 -f 参数，那么命令行输入的就是含有字符串化 IOR 的文件名。iormerge 从这些 IOR 中读取详细的“联系细节”信息，并将包含所有“联系细节”的 IOR 信息输出到标准输出。如果新创建的 IOR 对客户端有效的话，那么客户端可以和任何服务器副本进行通讯。

使用 iormerge 对于只含有单个对象的服务器副本是足够的。但是，iormerge 不适合服务器包含有能创建新的对象的工厂对象，原因是新创建的 IOR 只包含一套联系细节（创建对象的服务器副本的联系细节）而不是包含所有服务器副本的信息。

17.2.4、 服务器端的corbaloc URL支持

正如在 [12.4.2 小节](#) 讨论的那样，CORBA 并没有标准化服务器端对 corbaloc URL 的支持。而是大多数的 CORBA 产品提供用于通过 corbaloc URL

访问的私有API。让我假设你通过使用私有API使得你的对象可以通过foo这个名字能被访问。如果你的服务器副本分别监听在host1:5000, host2:6000, host3:7000 那么如下的corbaloc URL可以用于客户端和foo对象的服务器副本进行通讯。

```
corbaloc::host1:5000,:host2:6000,:host3:7000/foo
```

使用这样的 corbaloc 对于只含有单独的对象的服务副本是足够的，然而，对于包含能创建新对象的对象工厂的服务器却不适合。原因是，创建出来的对象的 IOR 只包含创建其对象的服务副本的联系细节信息而不包含其他服务器副本的信息。

17.2.5、 总结

就像你看到的，不同 CORBA 服务器的私有容错机制各自不同，无论是观感还是提供的能力都有所不同，然而，他们依然有一些共有特点：

- 除了 corbaloc URL 外，没有任何机制需要在服务器端进行编码，只要通过配置文件就可以实现
- 私有的容错机制单一地关注都是将服务器副本信息放入 IOR 中。这种机制不能帮助你或也不会影响你的多个服务器副本状态的一致性。
- 虽然这些机制是私有的机制，但是这些机制并不会影响你和其他 CORBA 客户端实现的产品交互。这是因为这些私有机制都是单一地关注如何嵌入服务器副本的联系细节到 IOR 中。并且 IOR 中有几套的联系细节是 CORBA 兼容的方式，用任何 CORBA 产品实现的客户端都可以和使用私有容错机制的服务器进行交互。

17.3 其他问题

17.3.1.1、容错不是负载均衡

需要注意的是，虽然容错和负载均衡都依赖于重复服务器的使用，容错结构并不需要隐式地实现负载均衡。本章所讨论的大多数的容错机制都不提供负载均衡机制。而是，很可能(但并不保证)大多数/所有的只和一个服务器副本通讯；只有在那个服务副本终止后客户端才试图寻找其他服务器副本进行通讯。这种情况下的一个例外是通过 Orbix IMR 部署的重复服务器。在这种情况下，IMR 执行每客户端负载均衡，即，IMR 重定向一些客户端到某一个服务器副本，其他客户端到另外一个副本，等等。

虽然 Orbix IMR 提供了可用的负载均衡机制，这种以单一的一种方法为客户端提供跨多个服务器副本的负载是可行的。例如，让我们假设 10 个客户端，通过 Orbix IMR 负载到两个服务器，这样就 5 个客户端连接一个服务器。如果每个客户端都发送相同的请求数到服务器，那么这种就实现了这两个服务副本的负载均衡，至少，在初始化的时候是这样的。然而，让我们假设，几分钟后，连接其中一个服务器副本的 3 个客户端终止。现在我们 2 个客户端和一个服务器副本通讯，而 5 个客户端和另外一个服务器副本通讯。Orbix IMR 没有其他的方法重新负载几个客户端到两个服务器副本。同样，如果第三个副本现在启动，IMR 也没有任何办法重新负载客户端到新增加的服务器副本。

术语适应性负载均衡 *adaptive load balancing* 是经常用于指可以周期性重新执行负载均衡策略的机制。CORBA 并没有标准化负载均衡机制，关于负载均衡额外的讨论也超出了本书讨论的范围。很多 CORBA 产品提供了私有的私有的负载均衡机制。感兴趣的读者可以去参考 CORBA 产品文档，或使用 internet 搜索引擎搜索，例如“CORBA adaptive load balancing”。

另外一种负载均衡的方法是投资硬件负载均衡器。硬件负载均衡器扮演了代理服务器的角色。他从客户端机器上接收请求并根据均衡器的配置信息将请求委托到服务器机器上。一些硬件负载均衡器使用这样机制完成负载均衡功能。

17.3.1.2、容错中的超时值

一个实际的问题在建立容错系统中，你需要调整客户端的超时时间值。如果服务器没有运行的话，客户端试图连接服务器将很快失败(典型的只需要几毫秒)并且一些 CORBA 客户端的运行系统将很快使用 IOR 中其他的联系细节信息。然而如果服务器计算机被关闭，或者不能从网络物理连接，那么客户端连接服务器连接服务器将要花掉相对较长的时间(也许几秒)

第18章、CORBA容错技术

- [术语和基础架构](#)
- [编写CORBA-FT服务器](#)
- [CORBA-FT客户端支持](#)
- [日志和恢复架构Logging and Recovery Infrastructure](#)
- [故障提醒](#)
- [总结](#)

前面章节讨论了一些 CORBA 实现的私有容错技术。本章将讨论 CORBA-FT，这是在 2000 年才加入 CORBA 规范的 CORBA 容错技术规范简称。

18.1 术语和基础架构

CORBA-FT 类型定在在模块 FT 中。大多数的容错基础功能集中在 FT::RelocationManager 接口中。这个接口定义了两个方法，但是多数的功

能从 `ObjectGroupManager`, `GenericFactory` 和 `PropertyManager` 继承而来, 我将依次讨论这些定义在 FT 模块中的每一个接口。

18.1.1.1、对象组管理器 `ObjectGroupManager` 接口

CORBA-FT 用术语对象组 *object group* 表示重复服务器。一个独立的副本被称为对象组中的一个成员 *a member of an object group*。可交互对象组引用 *Interoperable Object Group Reference (IORG)* 是对象组的 IOR。IORG 中包含多个档案 *profile* (多套联系细节): 典型的, 每个运行中的成员一个 *profile*。IORG 有一个嵌入的 `TaggedComponent` ([10.2.3 小节](#)) 记录了 IORG 的版本号。一旦对象组的成员集合发生变化 (例如, 一个成员终止或重启), IORG 的版本号就要发生变化。

`FT::ObjectGroupManager` 接口定义维护对象组中当前活动状成员的信息, 例如, 增加或者删除对象组中的成员。

18.1.1.2、泛型工厂接口 `GenericFactory Interface`

`FT::GenericFactory` 是一个工厂接口 ([1.4.2 小节](#))。在其名字中包含一个泛型 `Generic` 这个单词的原因是它可以创建任意类型的对象。

```
module FT {
    struct Property { ... }; // name-value pair
    typedef sequence<Property> Properties;
    typedef Properties Criteria;
    typedef CORBA::RepositoryId TypeId;
    ...
    interface GenericFactory
    {
```

```

typedef any FactoryCreationId;

Object create_object(
    in TypeId          type_id,
    in Criteria        the_criteria,
    out FactoryCreationId factory_creation_id)
    raises (...);

void delete_object(
    in FactoryCreationId factory_creation_id)
    raises (...);

};
};

```

图 18.1 FT::GenericFactory 接口

`create_object()` 方法的第一个参数是库标示([9.4 小节](#))。这个参数指定要创建对象的类型。第二个参数由一组名字-值对组成，用于应用程序的一些特殊处理，例如，提供对象初始化的参数值等等。这个方法返回新创建对象的方法，但是也输出一个对象标识符用于唯一标识工厂的对象。在某些情况下，这个标识符也用作`delete_object()`方法的参数。

18.1.3、 属性管理器PropertyManager Interface

“一项技术解决所有问题 one size fits all”的概念并不能适用于容错。容错可以使用很多技术来解决，在某些情况下，每一个技术都是适合的，但是这些技术却不是总能解决所有的情况。因为这样，开发人员使用属性(名字-值对)来告诉 CORBA-FT 在一个应用程序中可以使用什么样的技术。

FT::PropertyManager 属性管理器接口定义了用于管理容错属性的方法。属性可以分成默认属性，特定类型的属性(属性和 IDL 的定义相关)或者对象组属性(用于指定对象组的属性)。对象组属性高于特定类型属性，特定类型

属性高于默认属性。

CORBA-FT 定义的容错的属性将在下面做仔细的讨论:

工厂属性 `Factories` .

开发人员不得不实现 `GenericFactory` 接口, 并且创建每一个 CORBA-FT 服务器的副本实例。当 CORBA-FT 基础功能模块想创建对象组中的成员, 就需要调用服务器 `GenericFactory` 的 `create_object()` 方法。

CORBA是面向对象而非面向进程技术规范。因为这样, CORBA-FT基础功能模块并不知道在哪一个服务器进程中有特定的`GenericFacotry`对象驻留。为了解决这个问题, CORBA-FT定义了一个`FT::Location`类型, 它是`CosNaming::Name`([图 4.1](#))的typedef, 就是说, 是一个复合字符串。从效果上来说, `Location`就是一个逻辑名字, CORBA-FT能以该名字识别这个服务器进程。CORBA-FT指定了这个复合字符串的应该具有的格式。即, CORBA-FT将运行服务器上主机名进行编码, 并给这个服务器进程起了一个逻辑上的名字。

工厂属性 `factories property` 的值是一个结构可变数组, 这个结构包含一个 `GenericFactory` 对象, 一个 `Location` 和一些能被传递给 `create_object()` 方法的 `criteria` (类似于属性, 是一个名字值对的可变数组)

工厂属性是唯一不能设置为默认属性的属性。它必须为特定的接口类型所指定, 或为特定的对象组指定。通过类型特定的工厂属性, 使程序员可以在单个 `location` (服务器进程中) 存在多个类型特定的工厂。另一种情况, 如果开发人员想用在一个工厂在服务器内创建几种对象, 那么这个工厂可以被注册多次: 每一个 IDL 类型一次。

副本属性 `Replication style`.

副本属性可以设置为如下几个值的其中之一：STATELESS，COLD_PASSIVE，WARM_PASSIVE，或 ACTIVE。另有一个值是 ACTIVE_WITH_VOTING，这个值很有可能在未来被支持。所有这些值以常量的形式被定义在 FT 模块中。

STATELESS 值表示，这个对象的行为和该对象被调用的历史无关。这个副本属性用于只读地访问数据库的对象。

COLD_PASSIVE 和 WARM_PASSIVE 这两个副本属性，在对象组中只有一个被称为主成员的副本执行对象组上的调用。对象组的其他成员被称为备份成员。CORBA-FT 对主成员的检查点 checkpoint (状态快照) 执行周期性地保存。此外 CORBA-FT 基础模块将每一个主成员的请求日志记录下来。但是当新的主成员的检查点 checkpoint 被保存时，日志将会被截断。当主成员意外终止，最近的检查点加上日志中的应答被用来提升一个备份成员成为新的主成员。

COLD_PASSIVE 和 WARM_PASSIVE 的区别在于更新备份成员的技术。用 COLD_PASSIVE 的属性，原主成员的最近的检查点被加载在新的主成员上，并且日志中最新的请求在新的主成员上被重新调用一次。用 WARM_PASSIVE 属性，一旦检查点被保存，它就自动被装载到备份成员，因此故障的恢复也要更快。

使用 ACTIVE 和 ACTIVE_WITH_VOTING 两个属性，IORG 包含的联系细节不是组中成员的信息，而是代理服务器网关的信息。当网关从客户端接收到一个请求，它将请求转发给所有的组成员 (这使用的对服务器和客户端透明的是私有协议)。每一个组成员独立地处理请求，并将应答返回的网关。

在 ACTIVE 属性中，网关选取一条应答，并将它返回给客户端，同时丢弃其他应答。

以 ACTIVE_WITH_VOTING 属性，网关收集所有应答并对比他们，他们应该全部一致，但是如果某处发生错误，一个或多个应答就有可能不对。假设主要的

应答是可以识别的，网关将发送识别后的应答给客户端，并丢弃其他应答。

ACTIVE_WITH_VOTING 现在还不能被 CORBA-FT 规范支持，但是有可能会在将来被支持。

初始化和最小副本数属性 Initial and minimum number of replicas

这个整型值用于指定初始化的将要被创建的对象副本的数量。另一个整型值用于指定保证容错机制的对象所需要的最小副本数。如果太多副本终止以至于副本的数量小于最小副本数量的话，更多的服务器副本将被创建以达到最小的副本数量。

成员属性 membership style

成员属性值能被设置成 MEMB_INF_CTRL 或 MEMB_APP_CTRL

如果 MEMB_INF_CTRL 属性被使用，那么服务器通过调用副本管理器 replication manager 的 create_object() 方法来创建对象组，副本管理器是 CORBA-FT 的基础功能模块之一。副本管理器于是通过调用服务器进程中的工厂对象的 create_object() 方法来创建对象副本。

如果 MEMB_APP_CTRL 成员属性被指定，服务器进程通过调用副本服务器的 create_object() 方法创建一个初始化空对象组 *initial empty object group*。服务器必须做如下的步骤添加对象组中的成员。

- 服务器反复调用副本管理器的 create_member()，每一次调用指定不同 location 创建对象。副本管理器在每一个特定的 location 上调用工厂的 create_object()。
- 另外一种服务器在不同的 location 上调用工厂 create_object 方法，然后对每一个创建的副本再调用副本管理器 add_member() 方法。

一致性和检查点间隔属性 Consistency style and checkpoint interval

一致性属性可以是 CONS_INF_CTRL 或是 CONS_APP_CTRL。

如果 CONS_INF_CTRL 被使用，那么 CORBA-FT 基础模块自动执行检查点 checkpoint，记录请求日志，当主成员终止时执行故障恢复。当 CONS_INF_CTRL 属性被使用，一个作为补偿的策略值用于指定执行以 checkpoint 的频率。

如果 CONS_APP_CTRL 属性被使用，服务器应用程序代码必须执行检查点 checkpoint，记录请求日志，主成员终止调用故障恢复。

故障监控，间隔，超时，和粒度属性 Fault monitoring, interval & timeout ,and granularity

故障监控属性可以是 PULL，PUSH 或 NOT_MONITORED。PUSH 属性现在还未支持，但是可以预见到未来将会支持。

PULL 故障监控属性用于 CORBA-FT 周期性执行类似 ping 的操作，调用 is_alive() 检查对象成员是否还在运行。

故障监控间隔和超时属性是一个有两个域的结构，一个域用于指定 ping 请求的频率，另外一个域用于指定检查对象是否正常的请求的应答的超时时间

故障监控的粒度属性是对 PULL 属性的一个补充。粒度属性可以是 MEMB，LOC，或则 LOC_AND_TYPE 其中之一。MEMB(member 的短写) 值表示 CORBA-FT 基础功能模块必须单独地 ping 每一个成员。如果对象组中的成员数非常大，或则监控间隔时间非常短，那么这个属性值将会造成一定的网络成本。网路成本可以通过指定 LOC 属性来减少。一个 location，就是一个服务器进程。一个单独的 ping 消息发送给同一个 location(即同一个进程内)上的所有对象成员。如果被 ping 的对象是故障的，那么 CORBA-FT 将会假定所有的成员都是故障的。LOC_AND_TYPE 和 LOC 很相似，但是 LOC_AND_TYPE 是 ping 同一个 location 上的同一个 IDL 类型的所有对象成员。

如果 NOT_MONITORED 属性被使用，那么 CORBA-FT 并不会周期性去检查对象成员的状态。而是由开发者自己实现自己的故障监控机制，并且将故障告诉给 CORBA-FT 的故障提醒器 *fault notifier*，将在 18.5 小节详细讨论。

18.1.4、副本管理器接口 ReplicationManager Interface

正如在前面声明的，副本管理器接口继承于 ObjectGroupManager，GenericFactory 和 PropertyManager。且副本管理器定义两个方法 `registry_fault_operation()` 和 `get_fault_operation()`。故障提醒器 (FaultNotifier 接口) 是一个用于报告故障的对象。故障讨论器将 [18.5 小节](#) 讨论。

CORBA-FT 的实现提供了包含了副本服务器的基础进程，应用程序可以通过传递 “ ReplicationManager ” 字符串作为参数调用 `resolve_init_reference()` 方法连接副本服务器。副本管理器对象以单个对象的形式出现在应用程序中。然而，在真实情况下，副本管理器是有副本的，这是为了避免单点故障。CORBA-FT 规范要求 CORBA-FT 的实现中不能存在单点故障的情况。

18.2 编写CORBA-FT服务器

18.2.1、 IDL接口的修改

编写容错服务器的第一步，是保证服务端实现对象的 IDL 接口继承于 CORBA-FT 中定义的相应接口。作为一个例子，让我假设服务器实现了接口 Foo 和称为 FooFactory 的工厂接口 ([1.4.2 小节](#))。IDL 文件可能定义如下的内容

[图 18.2](#)

```
interface Foo
: FT::PullMonitorable , FT::Checkpointable
{
    ...
    void destroy();
};
interface FooFactory
: FT::PullMonitorable , FT::Checkpointable
{
    Foo create(...);
};
```

图 18.2 服务器使用 CORBA FT 的 IDL 定义例子

PullMonitor 接口定义了 `is_alive()` 方法，这是用于被 CORBA-FT 周期性调用的类似 ping-的方法检查对象副本的状态。一个 Servant 可以以全返回 `true` 来表示这个对象状态正常的做法来实现这个方法。相应的，Servant 实现这个方法也可以执行健康检查来确保服务器进程(或相关的硬件资源)在一个一致的状态，并且仅在此状态下返回真。

Checkpointable 接口定了 `get_state()` 和 `set_state()` 两个方法，这两个方法用于存储对象的状态(例如对象中的实例变量)。状态以二进制数据的形式代表(以 `sequence<octet>`)，将二进制数据转换成对象状态和将对象状态转换成二进制数据是服务器端程序责任。

IDL 接口可以可选的继承 Updateable 接口。他是 Checkpointable 的子类。Updateable 接口定义了两个方法 `get_update()` 和 `set_update()`。update 方法是最近一次 checkpoint 的对象状态的一种增量变化。

IDL 接口是否一定要从 Checkpointable 或 PullMonitorable 接口依赖于对象类型的容错属性的影响。如果对象使用了 `CONS_INF_CTRL` 策略和 `COLD_PASSIVE` 策略或 `WARM_PASSIVE` 策略那么对象就需要继承

Checkpointable 接口。否则，如果对象设置了 PULL 属性的话，对应的 IDL 接口必须继承 PullMonitorable 接口。

18.2.2、 创建和销毁一个对象副本

实现一个CORBA-FT服务器，开发人员必须实现了我们例子中的服务器的IDL Foo 和 FooFactory 接口的 Servant，并且也必须写一个实现 GenericFactory的Servant类（在 [图 18.1](#) 中显示的那样）。这就意味着，GenericFactory这个类不仅副本管理基础进程要实现，服务器进程也需要实现。

当服务器想创建一个副本对象，例如，在 FooFactory::create 方法体内，他并不是在本地创建对象，而是根据他所使用的成员属性来确定。

如果 MEMB_INF_CTRL 成员属性被使用，服务器调用副本管理继承 GenericFactory 的接口的 create_object()方法。副本管理器接着调用某些服务器副本上 GenericFactory 对象的 create_object()方法。服务器副本 create_object()方法的实现创建正常的 CORBA 对象，这个 CORBA 对象是对象组中成员。副本管理器通过联合各个成员联系细节信息来构建 IORG 信息。这个 IORG 就是 FooFactory::create()发回给客户端的信息。

如果 MEMB_APP_CTRL 成员属性被指定，服务器进程通过调用副本服务器的 create_object()方法创建一个初始化空对象组 *initial empty object group*。服务器必须做如下的步骤添加对象组中的成员

- 服务器反复调用副本管理器的 create_member()，每一次调用指定不同 location 创建对象。副本管理器在每一个特定的 location 上调用工厂的 create_object()。

- 另外一种服务器在不同的 location 上调用工厂 `create_object` 方法，然后对每一个创建的副本再调用副本管理器上 `add_member()` 方法。

`create_object()` 方法有一个 `out` 参数标示符用于关联新创建的对象。这个标示符在工厂创建的对象中唯一。调用者有责任保存这个标示符的值，并在将来将这个值传递给 `delete_object()` 方法。

当服务器想销毁一个 CORBA 对象，例如，在 `Foo::destroy()` 方法体，它调用副本管理器中 `GenericFactory` 对象的 `delete_object()` 方法。接着副本管理器调用服务器副本上的 `GenericFactory` 对象的 `delete_object()`，这样对象组中的每一个成员都能被销毁。

18.2.3、在 CORBA-FT 中注册服务器副本

CORBA-FT 服务器的主要思路是创建一个或多个 `GenericFactory` 对象输出它们的对象引用，例如到文件或者名字服务。

开发人员需要写一个工具(或者使用 CORBA-FT 实现提供的工具)，在副本服务器中为容错客户端注册属性。一些属性是工厂属性的信息，另外一些是每一个 IDL 类型的位置信息。注册工厂的原因是每一个 CORBA-FT 服务器都要求能输出他们的自己的工厂 IOR。

通过为副本注册系统属性，工具可以调用副本服务器的 `create_object()` 来创建一个对象组(这是在假设应用程序使用了 `MEMB_INF_CTRL` 属性的情况下)客户端可以通过文件或名字服务获得对象组 IORG。

18.3 CORBA-FT 客户端支持

IORG 中包含嵌入的 `TaggedComponent` ([10.2.3 小节](#))，

TaggedComponent用于说明这个对象引用是对象组的对象引用而非“通常”的对象引用。当客户端的运行系统遇到这个TaggedComponent，它将启用客户端的CORBA-FT功能，并将增强端到端的系统容错功能。如果客户端是用非CORBA-FT实现生成，那么客户端将忽略这个TaggedComponent。这个客户端仍能 and CORBA-FT的服务器通讯，但是它不能享受CORBA-FT所提供容错功能的任何好处。

18.3.1、 保持随时最新的IORG Keeping IORGs Up to Date

IORG中嵌入的TaggedComponent记录着IORG的版本信息。只要对象组中的成员发生变化(例如，某个成员终止或重启)，副本管理器将更新IORG的版本号，并通知CORBA-FT的服务器基础架构更新版本号。每当CORBA-FT客户端进行一次远程调用，一个服务上下文([11.6 小节](#))就用于在请求中传输IORG版本号。服务端CORBA-FT 架构对比服务上下文中包含的版本号和本身的版本号。它将：

- 如果客户端版本号匹配服务器端的版本号，这意味着客户端有最新的 IORG 版本号，于是请求被正常的派发。
- 如果客户端的版本比服务器的版本号低，这意味着客户端有一个过期的 IORG，在这种情况下，服务端 CORBA-FT 的基础模块将不分发请求。而是它将向客户端发送一个重定向消息到客户端，并提供给客户端一个最新的 IORG。CORBA 客户端的运行系统将使用新的 IORG 重发该请求。
- 如果客户端的版本号比服务器端的版本号高，这意味着服务端的版本号过期，在这种情况下，服务器端 CORBA-FT 的基础模块在分发请求前重新向副本管理器获得最新的 IORG。

版本号协议的目的是为了增加客户端获得的 IORG 对象组中当前状态正常成员的联系细节信息，忽略成员组中已终止成员的联系细节信息，并减少客户端和已终止成员通讯的可能性。

18.3.2、 确保客户端在主成员上调用

如果对象组使用 COLD_PASSIVE 或 WARM_PASSIVE 的副本属性，那么在 IORG 中的其中一个 profile(一套联系细节)将包含一个用于指明对象组主成员的 TaggedComponent ([10.2.3 小节](#))。理想情况下，CORBA 客户端的运行系统应该使用这些 TaggedComponent 作为其选择对象组中应该发送请求的目标的提示。然而，如果客户端不是通过 CORBA-FT 实现的产品生成的话，它将忽略此信息。甚至如果客户端是通过 CORBA-FT 实现的产品生成，这个提示也可能是过期的提示，因为有可能由于错误恢复，导致另一个不同的成员成为新的主成员。

如果请求被送到对象组的后备成员，在后备服务器的 CORBA-FT 的基础模块将会使用重定向信息 ([11.4 小节](#)) 重定向客户端到对象组主成员。

18.3.3、 透明的失败重调

正如将在 [18.4 小节](#) 讨论的，服务器端的 CORBA-FT 基础模块将记录所有的请求和应答信息在日志中。CORBA-FT 客户端将会在每一个请求中带上 FT_REQUEST 上下文。这个上下文有一个唯一标示符来标示，这个标示符唯一的标示了这个客户端中请求和一个过期时间。一旦服务器收到包含有 FT_REQUEST 的请求，CORBA-FT 基础模块会检查日志中是否有标示上下文请求信息。如果日志中存在，那么这个请求将不被执行，而是将日志中相应的应答返回给客户端。这个目的是允许为了客户端的运行系统执行自动失败重发机制，并且保证最多只成功调用一次的语义。

18.3.4、心跳消息Heartbeat Messages

TCP/IP (IIOP 基于此协议之上), 并不能很好的处理某几种类型的网络崩溃。例如, 如果客户端进程通过 TCP 正在连接一个服务器进程, 如果服务终止, 那么客户端进程将马上得到通知, 如果只是服务失败, 或者网络中断了, 那么客户端进程将不能即时的得到通知。在这种情况下, 客户端将会等待很长时间 (也许会永远) 从服务器接收应答。这个问题可以通过设置超时时间来解决。然而, 就算你知道特定的调用大概需要多长时间, 为每一个请求设置超时时间是比较繁琐的工作。

CORBA-FT 提供了另外一种机制以时间性的方法来检测网络故障。这个机制使用 CORBA-FT 客户端的基础模块周期性的发送类似 "ping" 的消息给 CORBA-FT 服务器。当 CORBA-FT 接收这个信息 (在 CORBA-FT 中称为心跳消息) 的某一笔时, CORBA-FT 基础模块不会分发此消息到对象。而是 CORBA-FT 基础模块自己发送一个应答信息到客户端。这是因为心跳消息只是用来检测网络连接是否异常, 而不是用来检测对象状态是否正常的。

心跳消息是否被发送, 是由客户端的策略 ([6.1 小节](#)) 来决定的。这个策略同时也指定了心跳消息发送的间隔频率和心跳应答接收的超时时间。

18.4 日志和恢复模块 Logging and Recovery Infrastructure

每一个 CORBA-FT 服务器都包含内建的称为日志机制和恢复机制的模块。这些模块是由 CORBA-FT 的实现者提供, 但是 CORBA-FT 并没有定义他们的 IDL 接口, 因为这些模块不是被应用程序代码直接调用的。

日志机制负责将到达对象的请求消息持久化日志, 并将调用完成的应答信息也持久化。日志机制也周期性地记录对象组中主成员检查点 (或则更新)。日志

可以以分布式的方式访问。如何达到这个目的是实现者的细节问题，一种可行的方式是日志被写入可复制的数据库；另外一种可行的技术是每一个主机单独的将日志维护在存储器里，并通过使用可信赖的，完全守序的广播协议来发送日志更新消息到其他的服务器。

如过你使用 COLD_PASSIVE 或 WARM_PASSIVE 复制属性，那么如果对象组中的主成员终止，那么备选成员将提升为主成员。当这个发生时，恢复机制用于将备选主成员的状态更新成最新。当新成员被加入到对象组时，恢复机制也用于 COLD_PASSIVE 或 ACTIVE 副本属性。

当恢复机制被使用，它将分析日志，并调用相关对象成员上的 `set_state()` 方法初始化对象状态到最近一个检查点状态。接着，恢复机制可能调用 `set_update()` 方法，装载最新的更新到对象。最后，它将调用比最后一个检查点/更新还近的请求消息将对象更新为最新状态。

为了节省空间，日志机制将周期性的对日志进行压缩。特别地，当对象的一个新的检查点被取得，前一个检查点和请求与应答信息将会被从日志中移走。同样，一但通过 `get_update()` 获得的新的更新，那么就的更新和请求应答消息也通常被从日志中移走。唯一的例外是，如果请求中 FT_REQUEST 服务上下文中的过期时间还未到，那么这些请求和应答信息就不能被移走。这是为了支持在

[18.3.3 小节](#) 讨论的透明的失败重调用机制

18.5 故障提醒

一个 CORBA-FT 的实现，包含一些称为故障监控器 *fault monitor* 和故障侦测器 *fault detector* 基础模块，这两个名词在 CORBA-FT 规范中可以进行互换。故障监控器的职责是监测故障。CORBA-FT 并没有为故障监控器定义 IDL 接口，因为故障监控器不会被用户直接调用。在 CORBA-FT 的部署中可以有几个故障监控器：典型的方法是每一个 CORBA-FT 服务器都有一个故障监控

器。故障监控器可以调用成员对象上的 `is_alive()` 方法来检查对象是否故障。

当故障监控器发现一个故障，故障监控器需要向副本服务器报告并且有可能向用户写的程序报告，例如，分析报告，或则用图形方式显示它们的信息。OMG 定义了使用故障报告机制应该基于通告服务 `Notification Service` 中概念 ([22.3 小节](#))。通告服务自身被认为是太过复杂以至于它无法以容错的方式实现它自身的所有功能。而仅仅是通告服务功能的一个非常小的子集被 `CORBA-FT` 提取并重新组织成 `FT::FaultNotifier` 接口，如 [图 18.3](#) 小节所示。

```
module FT {
    ...
    interface FaultNotifier
    {
        typedef unsigned long long ConsumerId;
        void push_structured_fault(
            in CosNotification::StructuredEvent event);
        void push_sequence_fault(
            in CosNotification::EventBatch event);
        ConsumerId connect_structured_fault_consumer(
            in CosNotifyComm::StructuredPushConsumer
                push_consumer);
        ConsumerId connect_sequence_fault_consumer(
            in CosNotifyComm::SequencePushConsumer
                push_consumer);
        void disconnect_consumer(
            in ConsumerId connection) raises(...);
        void replace_constraint(
            in ConsumerId connection,
            in CosNotification::EventTypeSeq event_types,
```



```

        in string                                constr_expr);
    };
};

```

图 18.3 FT::Notifier 接口

应用程序可以通过调用 `resovle_initial_reference("ReplicationManager")` 来连接副本服务器，然后调用副本服务器 `get_fault_notifier()` 获得故障通知器的引用 `fault_notifier`。

故障监控器并不需要明确地在故障通知器中注册。而是，一旦故障监控器有一个故障通知器的引用，故障监控器可以调用 `push_structured_fault()` 来发送一个单个的故障事件。另一种方式是，故障监控器可以调用 `push_sequence_fault()` 来发送一个故障事件的数组给故障通知器。

应用程序想要处理故障事件，就需在故障通知器中注册消费者，如果应用程序想接收单个的故障事件，应用程序可以调用 `connect_structured_fault_consumer()` 方法注册消费者。如果应用程序以一个批的形式接收故障，应用程序调用 `connect_sequence_fault_consumer()` 方法注册消费者。这两个方法都返回一个 `ConsumerId`，这个 `Id` 可以在将来作为参数调用 `disconnect_comsumer()` 方法断开故障通知器和消费者的连接。

默认情况下，故障事件的消费者会接收所有故障事件。然而，一旦连接，消费者可以调用 `replace_constraint()` 方法来指定用故障通知器过滤不想要的故障事件 ([22.3.4.1 小节](#)) 的约束条件。

CORBA-FT 规范精确的指定了故障事件所包含的信息。这些信息包括，
location, IDL 接口类型, 故障对象的对象组 id

副本管理器可以将自身作为一个消费者注册在故障通知器中。开发者也可以

开发自己的应用程序，并且在故障通知器中注册为故障事件的消费者。这样的程序可以分析故障(基于 location, 频率, 等等)或则以图形形式统计故障。故障管理器并不知道有多少消费者连接到故障通知器,故障管理器只是简单的将故障事件推向故障通知器,换句话说,故障管理器将故障事件推向所有注册的故障事件消费者。

18.6 总结

CORBA-FT定义了许多基础模块,因为这样,CORBA-FT的第一印象非常复杂。然而大部分CORBA模块都由CORBA-FT产品来实现,并且它是在场景之后工作。只有很少数量基础模块是可见的,或者说,必须被开发者自己实现的。这意味着,CORBA-FT并不像想象中的那么复杂。然而无疑CORBA-FT是比 [17 章](#) 讨论的私有CORBA容错机制更复杂也更强大的容错机制。

CORBA-FT 的使用影响应用程序的设计和编码。因为这样,最好在程序一开始设计就应该考虑 CORBA-FT 的设计,而不应该在一个可用的应用程序中来设计。以此相反的是,大多数私有容错机制通常通过配置文件而非编码的机制实现。这意味着,私有容错机制可以经常并简易地加入到已存在的应用程序中。

也许,CORBA-FT 最大的缺点是它是 CORBA 的一个可选规范,并且不幸地是,大多数 CORBA 产品都选择不实现它。本文作者也尽知道只有一个 CORBA 实现,TAO,如今实现了 CORBA-FT;当然也有可能存在笔者不知道的 CORBA-FT 实现的产品。

第19章、 其他的CORBA基础部件

- [实时CORBA Real-time CORBA](#)
- [嵌入系统CORBA CORBA for Embedded System](#)

- [CORBA组件模型 CORBA Component Model \(CCM\)](#)

本章简要的讲解一下本书还未讨论到的CORBA基础模块的其他的可选部分。

感兴趣的读者可以从CORBA规范 (在 www.omg.org可以找到) 或者从支持这些功能产品的产品手册中获得更多的细节。

19.1 实时CORBA Real-time CORBA

正如名字那样, 实时 CORBA 规范是使 CORBA 适合于在实时应用程序中被使用的 CORBA 扩展。笔者目前还不清楚有关讲述 CORBA 实时编程的书, 当然, 实时规范的文档可以从 OMG 的官方网站获得, 并且任何实现 CORBA 规范的产品都有可能提供在其手册中提供相关的信息。

19.2 CORBA嵌入系统CORBA for Embedded System

CORBA 系统的核心部分拥有很多的功能, 以至于某些 CORBA 的实现很有可能占用几兆的内存空间。现在的桌面系统有足够的内存空间来运行基于 CORBA 的应用程序, 而很多的嵌入系统的内存空间更少。为了裁剪 CORBA 满足嵌入式系统内存的约束, 几家 CORBA 厂家提供 CORBA 功能子集的能力来满足只使用这部分 CORBA 功能的这部分应用程序, 这样的应用程序就可以部分地连接更小的 CORBA 库。一些厂家将有用的 CORBA 子集压缩到少于 100k 那么大。

最小 *Minimum* CORBA 规范 是 OMG 规范的用于嵌入系统的 CORBA 规范子集。定义标准化的 CORBA 功能子集规范的目的是提供 CORBA 子集规范源代码应用程序移植的能力。然而, 最小 CORBA 规范却受到了批评, 因为这个最小规范对于嵌入系统来说还是过大了。由于这个原因, 提供嵌入式应用的 CORBA 厂家提供了他们自己私有的 CORBA 规范子集以满足最小化内存的需求。

19.3 CORBA 组件模型 CORBA Component Model(CCM)

最近几年，有一种逐渐流行的认识，就是应用程序不仅包含“业务逻辑”代码，也同时包含着很多执行如安全，交易，数据持久化等的“基础模块代码”。事实上在应用程序中，基础功能代码比业务逻辑代码更多。不用说，因为写大量基础功能代码增加了程序开发的成本

J2EE(Java 2 Enterprise Edition)中一个重要的能力就是它是一个应用服务器：特点是提供了可以被用于业务逻辑代码基础功能服务。应用服务器的目的是让开发人员可以集中精力在业务逻辑开发上，而无需担心基础功能的支持。业务逻辑代码以 JavaBean 的形式存在，并可以通过 J2EE 服务器来部署。部署是通过 J2EE 应用服务器读去部署描述子 *deployment descriptor*，这是一个指定 JavaBean 将被如何装载的 XML 文件，并且指定那些基础服务功能(交易，安全等)将被应用于 JavaBean 上。

CORBA组件模型是一个J2EE应用服务器的泛化概念。之所以是泛化，因为CCM的目标是为任何一种语言编写的业务逻辑提供应用服务器的功能，而不仅仅是Java。CCM是最新的CORBA规范(3.0)中的一部分。现在有一些CCM的实现，一些是CORBA厂家实现的，另外一些是第三方的公司实现的——但是现在就预言是否CCM会流行，是否CCM会成有特定产品市场都为时过早。更多的关于CCM细节的描述可以在*Pure CORBA Book* [\[Bo101\]](#)中找到。

第四部分、CORBA 服务

第20章、 交易服务Trading Service

- [SerivceTypeRepository接口](#)
- [Register接口](#)
- [Lookup接口](#)
- [交易服务的其他功能](#)
- [使用交易服务](#)
- [服务质量](#)

CORBA提供了为服务器公告其对象引用的几种方式，一种是 [3.4.2 小节](#) 讨论的，将字符串化IOR写到文件中。另一种方式是以名字服务([4 章](#))的方式公告服务器对象。本章讨论第三种方式——交易服务Trading Service。

正如名字服务通常被比作白皮电话号码本一样，交易服务也通常被比作黄页电话簿。黄页包含组织成不同的类型(例如，大厦，维修厂，餐馆)公告，同样交易服务包含几种根据服务资助类型 *service offer types* 组织的服务资助者 *service offers*。在黄页中的每一个公告包括公司的联系细节(地址，电话)，公司的描述(例如 24 小时营业，全镇最便宜等)。相似的，每个交易服务中提供的服务资助者都提供联系细节(IOR)，同样也提供对象所代表服务资助者的描述。

20.1 SerivceTypeRepository接口

[图 20.1](#) 显示了ServiceTypeRepository IDL接口定义的的简化版本。

在本章的后面显示接口，都有一些接口的简化规则：

- `raise` 子句被省略

- 一些 typedef 子句被去掉。例如，add_type 的 name 参数被显示为 string 类型。在真实的 IDL 定义中，name 是一个 typedef string 的定义。
- 一些方法的参数，和结构的域被显示为匿名的可变数组类型 anonymous sequence，例如 sequence<string>。使用匿名可变数组参数是非法的，并且也不赞成在结构中使用这样的类型。使用匿名类型的目的是为了减少对 typedef 的使用，并使得 IDL 接口定义显示出来更简洁。

```
module CosTradingRepos {  
    interface ServiceTypeRepository {  
        enum PropertyMode {  
            PROP_NORMAL, PROP_READONLY,  
            PROP_MANDATORY, PROP_MANDATORY_READONLY};  
        struct PropStruct {  
            string          name;  
            CORBA::TypeCode value_type;  
            PropertyMode    mode;  
        };  
        typedef sequence<PropStruct> PropStructSeq;  
        struct IncarnationNumber {  
            unsigned long high;  
            unsigned long low;  
        };  
        struct TypeStruct {  
            string          name;  
            PropStructSeq   props;  
            sequence<string> super_types;  
            boolean         masked;  
            IncarnationNumber incarnation;  
        };  
    };  
};
```

```

};

...
readonly attribute IncarnationNumber incarnation;

IncarnationNumber add_type(
    in string          name,
    in string          if_name,
    in PropStructSeq   props,
    sequence<string>   super_types)
    raises(...);

void remove_type(in string name) raises(...);
void mask_type(in string name) raises(...);
void unmask_type(in string name) raises(...);
...
};
};

```

图 20.1 ServiceTypeRepository 接口的部分伪定义

正如之前提到过的，电话簿黄页根据公司的类型，例如电器公司和修理厂，来进行组织。在交易服务中等价于“公司类型”的是 `CosTradingRepos::ServiceTypeRepository::TypeStruct`，通常这个在交易服务中被称为服务资助者类型 `service offer type`。 `ServiceTypeRepository` 被用于定义服务资助者类型。

20.1.1.1、 `add_type()` 和 `remove_type()` 方法

`add_type` 用于定义新的服务资助者类型。 `name` 参数指定类型的名称，例如“打印机”。 `if_name` 参数指定一个与IDL接口关联的服务资助者的库标示 `repository id` ([9.4 小节](#))。一些读者可能对为什么不将服务资助者的类型名字硬编码为服务资助者的接口库标示，即资助者类型名等于资助者库标示名，感

到疑惑。这个问题的答案是，名字比库标示更易读。例如“Printer”就比“IDL:acme.com/Equipment/Printer:1.0”更易读。在这里，继承机制也是可以允许的。例如，上一个句子中的库标示定义了一个称为Equipment::Printer的接口，该类型的对象，或子类型都能用于作为“Printer”的服务资助者。

props 参数指定一个关联服务资助者类型可变的属性数组。这是电话簿黄页与名字服务的不同点。在黄页中的一些广告声称公司具有例如“24 小时服务”，“所有工作都有一年的保证”，“本镇最低价”的广告词。然而，黄页中并没有规定广告应该有什么广告词。相反的是，交易服务的服务资助者类型必须指定每一个服务资助者类型(广告)具有什么样的属性(广告词)。每一个属性(由 PropStruct 指定)由一个名字，一个类型(例如 long, boolean, string)和模式 mode 组成。模式 mode 域是一个枚举 enum，指定属性是否是必须的，和是否是只读的。

ServiceTypeRepository 并不对属性的类型有约束。但是坚持使用整型、浮点型、布尔型、字符型、字符串型、这些类型的可变数组类型是有点好处的。这是因为用于支持交易服务的语言提供了对所有这些类型的支持，而对其他的类型只提供了最小支持。

super_type 参数罗列服务资助者的父服务名字列表。一个服务资助者继承其父的所有属性，并且允许在其属性上引入更多的要求。比如，一个可选的属性可以变成必要属性，一个正常属性可以变成只读属性。

add_type() 方法的返回值是 IncarnationNumber，这个值是一个 64 位值。在交易服务被定义的时候，unsigned long long 并不是 IDL 的类型，所以包含两个 unsigned long 的结构用于表示 IncarnationNumber。IncarnationNumber 概念上是一个服务资助者类型什么时候被定义的时间

戳。一个类似list的方法(在 [图 20.1](#) 中未显示)用于罗列自特定 IncarnationNumber 之后的定义服务资助者类型。

remove_type()方法用于移出一个服务资助者类型。

20.1.2、 mask_type()和unmask_type()方法

make_type()方法用于掩住 mask(隐藏 hide)一个服务资助者类型。它有两种用法。一种用法用于不支持的服务者类型,这样交易服务就不在接收类型被 mask 的类型的服务资助者(广告)。另外一种用法是指明服务资助者类型是一个抽象基类型,不能被直接实例化,但是允许其他服务资助者类型继承。unmask_type()方法是用于反掩 unmask 被调用 mask_type()掩住的类型。

20.2 Register接口

当一个服务资助者类型被定义,那么对应用程序来说就可以在交易服务中输出(公告)他们的对象引用。这是通过Register接口([图 20.2](#))来完成的。

```
module CosTrading {
    struct Property {
        string name;
        any value;
    };
    typedef sequence<Property> PropertySeq;
    struct Offer {
        Object reference;
        PropertySeq properties;
    };
    typedef sequence<Offer> OfferSeq;
    interface Register
```

```

        : TraderComponents, SupportAttributes
    {
        string export(
            in Object      reference,
            in string      type,
            in PropertySeq properties) raises(...);
        void withdraw(in string offer_id) raises(...);
        void modify(
            in string      offer_id,
            in sequence<string> del_list,
            in PropertySeq modify_list
        ) raises(...);

        ...
    };
};

```

图 20.2 Register 接口的部分伪 IDL 定义

20.2.1、 **export()**和**withdraw()**方法

export()方法用于创建一个服务资助者，即，对象的公告。这个方法参数指定一个对象引用，匹配的服务资助者类型的名称(参数 `String type`)，及其属性。这里提供的属性必须匹配服务资助者指定的属性。**export()**方法的返回值是一个唯一的字符串，用于标示资助者的标示 `offer id`。这个 `offer id` 可以用于移出 **withdraw** 公告或者修改公告一些非只读属性。

withdraw()方法用于移出(即删除)一个服务资助者。

20.2.2、 `modify()` 方法

`modify()` 方法用于删除或修改某些和服务资助者关联的属性。如果删除一个必要属性将会抛出异常，同样，修改一个只读属性也会抛出异常。

20.3 Lookup接口

Lookup接口 ([图 20.3](#)) 只有一个叫做 `query()` 的方法，这个方法用于从交易服务中匹配满足特定约束条件的服务资助者(公告)的信息。

```
module CosTrading {
    struct Property {
        string name;
        any value;
    };
    typedef sequence<Property> PropertySeq;
    struct Offer {
        Object reference;
        PropertySeq properties;
    };
    typedef sequence<Offer> OfferSeq;
    interface OfferIterator {
        boolean next_n(in unsigned long n,
                      out OfferSeq ids) raises(...);
        ...
    };
    interface Lookup
        : TraderComponents, SupportAttributes,
        ImportAttributes
    {
```

```

enum HowManyProps {none, some, all};
union SpecifiedProps switch (HowManyProps) {
    case some: sequence<string> prop_names;
};
...
void query(
    in string          service_type_name,
    in string          constraint,
    in string          preference,
    ...
    in SpecifiedProps desired_props,
    in unsigned long   how_many,
    out OfferSeq       offers,
    out OfferIterator  offer_iter,
    ...) raises(...);
};
};

```

图 20.3 Lookup 接口的部分伪 IDL 定义

约束 `constraint` 参数是一个作用于服务资助者属性的布尔表达式。例如，让我们假设打印机服务资助者类型定义了一个整型属性叫做分辨率 `resolution` 还有一个 `sequence<string>` 属性叫做语言。如下的一个 `constraint` 可以用于获得分辨率在 600 以上，并且支持 PostScript 打印语言的打印机

```
resolution >= 600 && "PostScript" in languages
```

`in` 这个操作符测试是否左边的值 ("PostScript") 是在右边指定的 `sequence` 之中。Constraint 的语法是 Trade Constraint Language (TCL) 语法，并且这个语法被定义为 CORBA 交易服务规范的一部分。

可能有几种服务提供者类型匹配指定的约束，这些匹配的细节通过几个输出参数返回。例如 `offers` 参数提供了前 `how_many` 个匹配的服务资助者细节。如果有更多的匹配，可以通过调用返回的 `offer_iter` 迭代器 ([1.4.2.3 小节](#)) 上的方法访问。应用程序执行查询 `query` 操作的目的在于查看匹配的服务提供者一些属性。`desired_props` 参数用于指定那么属性可以通过 `offers` 和 `offer_iter` 访问。

`preference` 参数用于指定获得服务资助者的顺序。例如，“最大分辨率”就是按打印机提供分辨率的顺序，随机就是按打印机分辨率的随机顺序，这个参数可以用于让交易服务来做多个客户端到几个服务器端的负载均衡

20.4 交易服务的其他功能

交易服务还有其他几个重要的功能，我在这里简单的罗列一下。

可以通过连接交易服务器，使得查询模式能从一个交易服务器传递到另外一个另外一个交易服务器。交易服务器规范将这个称为连接 *linking* 和联合 *federating* 几个交易服务器。

服务资助者提供的大多数属性都是静态（即，不变）的值。例如，打印机的分辨率 `resolution` 属性一般都不会变化。然而，交易服务允许属性是动态的。动态属性通过一个回调对象 ([1.4.2.2 小节](#)) 实现。当执行查询时，交易服务通过调用回调对象获得指定属性的当前值。动态属性可以是，例如，打印机的队列长度这样属性上。

更多的关于上述提到的规范的讨论超出了本书的讨论范围，感兴趣的读者可以在其他书本 [[HV99](#), [BVD01](#)] 中找到更多的细节

20.5 使用交易服务

交易服务的功能跨了多个 IDL 接口的定义。这可能导致你认为在使用交易服务的过程中有许多编程工作。然而，事实并不是这样的，正如我将讨论的那样。

虽然这并不是交易服务规范的要求，但是几乎每一个厂商都提供了命令行工具和/或图形界面程序用于封装这些 IDL 定义的功能。这些命令行工具和/或图形界面程序使得大多数的交易服务管理工作可以通过不编码实现成为可能。例如：

- 命令行工具和/或图形界面程序可以用于增加，删除，掩住 mask，反掩 unmask 和浏览服务资助者类型。这样你就不需要编写和 ServiceTypeRepository 接口交互的程序
- 你能硬编码一个应用程 `export()` 一个服务资助者到交易服务。另外一种更少的编码的做法是将服务器的对象引用字符串化后写入到文件中。你接着可以使用命令行工具和/或图形界面程序来导入这个 IOR，并从 IOR 中构建一个服务资助者，并将这个服务资助者输出到交易服务。

这样就减少了你的编码负担，而应用程序只需要调用 Lookup 接口的 `query()` 方法。如果应用程序有一个交互的用户，那么交互用户可能希望能看到返回所有的服务资助者的属性，并且手工的选一个。如果使用交互选择返回服务资助者的客户端不可用的话，那么客户端应用程序可以随机的选择一个返回的服务资助者。事实上，命令行工具可以用于执行这样的查询(由特定的命令行参数指定)并随机的选择一个返回的服务资助者的 IOR。如果客户端通过 CORBA Utility Package [[Mch](#), Ch 2] 提供的 `importObjectRef()` 工具函数(4.3 小节)导入一个对象引用，那么客户端就不需要硬编码调用交易服务的 API。而是客户端可以传递一个 `instruction` 参数到 `importObjectRef()` 告诉函数执行命令行工具，并将标准输出解释为一个字符串化的对象引用。

应用程序可以通过调用 `resolve_initial_reference("TradingService")` 来连接交易服务的 Lookup 接口。交易服务的所有功能都继承一些基础接口。这些基础接口提供了只读属性来访问其他交易服务接口。例如，一旦应用程序通过使用 `resolve_initial_reference()` 连接到 Lookup 接口，应用程序可以接着调用接口上的属性访问 Register 和 ServiceTypeRepository 接口。

20.6 服务质量

交易服务规范并没有要求如何实现存储服务资助者类型和服务资助者信息的细节。

一些交易服务实现仅仅将这些信息存储在 RAM 中。这种实现一般用于没有持久化存储的嵌入式系统。但是这种做法在有存储介质的电脑上不值得推荐，因为在每次交易服务终止并重启的时候，你需要重新维护服务资助者的信息。

一些交易服务实现使用平台的文本文件来存储这些信息。这对小的应用来说比较方便，但是这并不利益部署的扩展。并且有可能存在当交易服务正在更新文件中信息时进程被杀死，导致文件信息被破坏的情况。

一些交易服务的实现将信息存储在数据库中。这样比文本文件提供了更高的可靠性，但也增加了额外的管理成本。

交易服务规范的功能被分在多个 IDL 文件定义中，并且交易服务规范并不要求所有的接口都被实现。且一些交易服务的功能是可选的。这就使得交易服务的实现厂家在交易服务实现的功能和交易服务消耗的硬件资源(内存，磁盘空间，CPU 速度等等)做出权衡。

如果你希望使用交易服务，问清楚你的 CORBA 实现厂家，它的交易服务存储信息的方式，是否实现了所有功能或实现了那部分功能对你来说是非常重要的。确定你所选的交易服务提供的服务质量满足你的需求。

第21章、对象事务服务 Object Transaction Service

- [关联CORBA对象和数据库记录](#)
- [每方法事务 Per-operation Transactions](#)
- [分布事务概述](#)
- [CORBA对象事务服务CORBA Object Transaction Service\(OTS\)](#)
- [OTS的原始API](#)
- [如何在其他CORBA部件上建立OTS](#)

21.1 关联CORBA对象和数据库记录

当你设计一个数据库交互的CORBA服务器，你可能会使用数据库的数据库表的每一条记录一个对象。这样做，你就需要关联CORBA对象到相应的数据库中的记录。这可以通过将数据库记录的主键存储在CORBA对象的object id([5.6.1 小节](#))来完成。

可以为每一条记录对应一个Servant。如果你这样做你就需要对象的标示object id存储在Servant的实例变量中。但是将数据的信息缓存在实例变量中并没有太大的用处。因为这样会存在缓存信息冒着过期的风险，因为数据库中的信息有可能被直接更新而不是通过CORBA服务器来更新。通过以每一个Servant对应一条数据库记录，Servant中唯一的实例变量就是object id(数据库表中的主键)。这种方式是比较浪费内存的。一种更节省的方式是使用default servant ([5.6.4 小节](#))来代表所有的CORBA对象。default servant可以调用POACurrent([13 章](#))上的get_object_id()来查找那一

个CORBA对象来处理当前的请求。Default Servant于是使用object id做为数据库的主键访问数据库。

21.2 每方法事务 Per-operation Transactions

在很多的客户-服务器系统中，IDL 方法的方法体经常是如下的形式：

```
void some_operation(...)
{
    begin_transaction();
    ... // query or update a record in the database
    commit_transaction();
}
```

注意整个事务包含在单个的 IDL 方法的方法体中。这种情况经常被称为每方法事务 *per-operation transaction*。如果你的打算写一个 CORBA 服务器，他的所有事务都严格的是每方法事务的话，那么你和数据库的交互就完全的独立于你对使用 CORBA。例如，你可以使用任何你想要的数据库，并且你可以和数据库通过你想要的任何技术，例如，嵌入 SQL，ORACLE OCI，ODBC，JDBC 等进行交互。

对于很多应用程序而言，每方法数据是足够的。然而，一些客户端服务器系统要求事务具有跨多个客户端到服务器端方法调用的能力。这种交互就需要使用 CORBA 对象事务服务 CORBA Object Transaction Service(OTS)。其他一些其他的客户端-服务器交互包括访问多个数据库。这类事务通常称为分布数据事务 *distribute transaction*，同样也需要使用 OTS

21.3 分布事务概述

分布事务的概念早于CORBA的概念，并且它对立任何类型的中间件。让我们假设客户端希望一个交易跨查询和更新两个数据库。为了做到这样，客户端使用交易管理器 *transaction manager(TM)*，如 [图 21.1](#) 所示

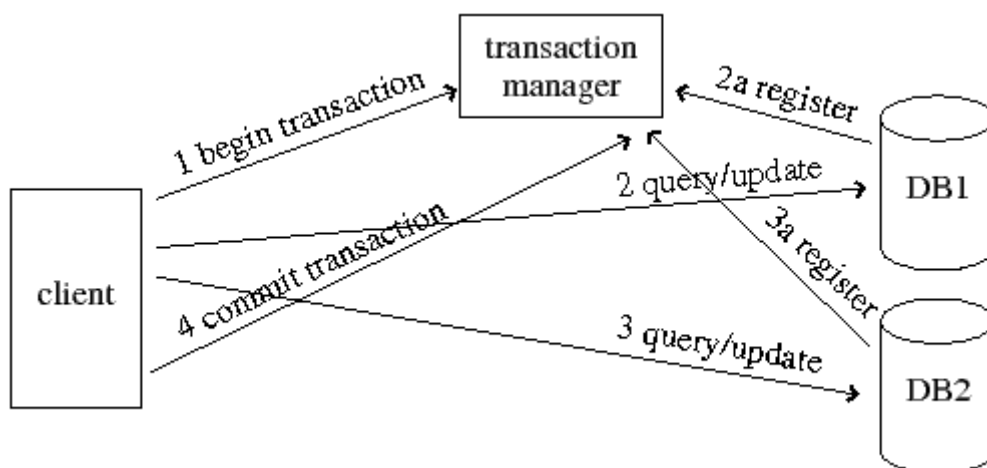


图 21.1 分布式事务

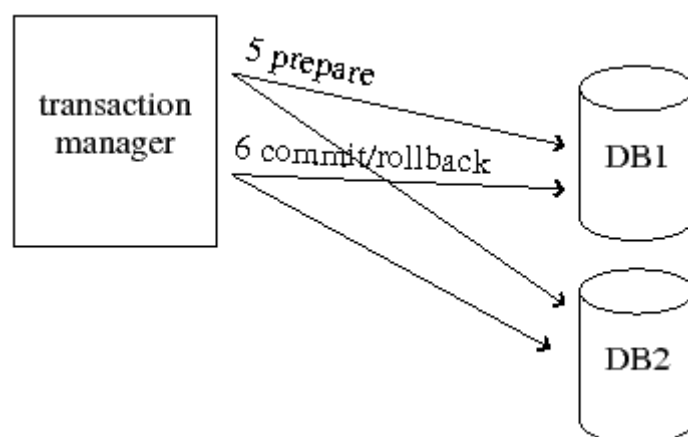


图 21.2 两段提交

当客户端打算开始事务，它将发送一个请求到TM(第一步)。TM返回一个事务标示符唯一的标示了一个新事务。客户端接着发送查询或者更新请求到数据库，或则类似于CORBA的中间件系统，封装了数据库的服务器进程；如图中第二步和第三步。事务标示将会作为请求的一部分进行传输。一旦数据库(或者封装了数据的服务器进程)被客户端访问，数据库(服务器)使用接收到的事务标示符告诉TM，一个数据库(服务器)正在加入到事务中(第2a步，第3a步)。TM使用一个持久化的存储区(例如文件或自己的数据库)来记录下哪么数据库(服务器)参与了事务。最后客户端通知TM提交事务(4步)。为了避免在图太混乱不整洁，剩下的部分在 [图 21.2](#) 中显示。TM开启和所有数据库(服务器)的两段提交对话

two-phase commit dialog.

两段提交协议中的第一段，TM 请求每个数据库/服务器开始准备提交。每个数据库/服务器通过持久任何更新到磁盘做自身的准备，但是这时数据库和服务器的必须一种方式能撤销更新（例如，数据库不仅应该持久化更新信息，还需要持久化怎样撤销更新的信息）。准备完成后，数据库/服务器于是给 TM 应答，告诉 TM 它将选择提交或回滚事务。

在两端提交协议中的第二段，TM 分析所有来自数据库/服务器的选择。如果所有都选择提交那么 TM 向每一个数据库/服务器发指令提交修改。如果有一个数据库或服务器选择了回滚，TM 将通知所有的数据库/服务器撤销在准备阶段做的修改。最后 TM 将在自己的持久化存储里面删除刚刚结束的事务信息。

如果系统曾经终止，那么当系统重启时，TM 将检查自己的持久化存储中关于正在事务进程中的信息。使用这些信息，TM 通知所有在 TM 终止之前还未到提交阶段数据库/服务器回滚。TM 可以重新执行当 TM 意外终止发生时正处于提交阶段事务的后阶段的指令。

开发组织 Open Group (www.opengroup.org)，有时又被称为 X/Open，定义了分布事务的标准。其中的一个标准被 XA，是一个基于 C 语言的 API，用于 TM 和资源管理器 *resource manager*（通常就是数据库）交互实现两段提交。这个标准的重要性是，该标准不仅可以允许协调事务跨多数据库，而且可以跨多个不同厂商的数据库。[21.4 小节](#)我们将讨论，CORBA 如何影响 XA 标准，使得 XA 标准允许 CORBA 应用程序参与分布式事务。但是，在那讨论之前，有两点必须要引起我们的注意。

第一，在设计一个客户端-服务器系统的时候，提前了解你将使用的中间件，确定你有使用分布式事务的需求，并确保你的中间件能用于分布式事务是事半功倍

的。然而，设计你的系统只使用每方法事务的事务方式而不是分布事务也许是更好的做法。一个原因就是，当程序突然终止时，本地事务的恢复比分布式事务的恢复要简单高效。另外一个原因本地事务的生存时间短，所以数据库上锁的时间也较短。事务的短生存时间可以增加数据库访问的并发性，并提升系统的性能的扩展性。相反的，分布式事务是一个长生存周期，尤其如果设计用户输入的话，这样就会限制系统的并发访问和扩展性。最后一个原因是分布式事务比本地事务占用更多的成本(例如 记录有关两段提交协议额外需要的通讯信息等)。

第二，Enterprise CORBA [\[SGR99\]](#)这本书花近 100 页来讨论在客户端-服务器系统中的数据库使用。如果你计划在客户端-服务器系统中使用数据库，那么推荐你读一读此书，因为这本书中提供了很多有用的设计建议。

21.4 CORBA对象事务服务CORBA Object Transaction Service(OTS)

对象事务服务 Object Transaction Service(OTS)是一个 CORBA 服务，此服务使得 CORBA 应用程序中可以使用分布，两段提交事务。OTS 由(1)一些 IDL 定义(大部分的定义在叫做 CosTransactions 的模块) (2)一些额外的库文件需要链接在服务器和客户端的应用程序中 (3)事务管理器。在第一印象中，OTS 规范可以以一个非常复杂的形式出现。这是因为如下的两个原因。

第一个 OTS 表现复杂的原因是，OTS 的规范不仅为普通开发者定义了接口；它同时也定义了用于开发商实现 OTS 的底层基础 API。OTS 规范定义底层基础 API 的原因是为了确保不同的 OTS 实现的交互性。这意味着用某个 CORBA 产品生成的 OTS 客户端参与另外的一个 CORBA 产品实现的 OTS 服务器的分布式事务。

第二个 OTS 表现复杂的原因是，OTS 的规范非常灵活，它甚至允许事务应用

程序以几种不同的方式实现。大多数开发人员可以使用简单的 API 实现他们的业务逻辑，剩下的 API 留给 OTS 自动的执行一些基础的工作。然而 OTS 也允许开发者更多的自己动手手工处理基础任务。这些手工 API 集合可以让开发者集成 OTS 到非 XA 兼容的数据库或者实现一个从 OTS 到非 CORBA 分布事务系统的桥。

```
interface TransactionFactory {
    Control create(in unsigned long time_out);
    ...
};

interface Control {
    Terminator get_terminator();
    Coordinator get_coordinator();
};

interface Terminator {
    void commit(...);
    void rollback();
};

interface Coordinator {
    RecoveryCoordinator register_resource(in Resource r);
    ...
};

interface RecoveryCoordinator {
    Status replay_completion(in Resource r);
};

interface Resource {
    Vote prepare();
    void rollback();
    void commit();
};
```

```

...
};
local interface Current : CORBA::Current {
    void begin();
    void commit();
    void rollback();
    void set_timeout(in unsigned long seconds);
    unsigned long get_timeout();
    Control get_control();
    Control suspend();
    void resume(in Control which);
};

```

图 21.3 OTS API 的子集

大部分的OTS API在 [图 21.3](#) 中显示。本章讨论范围外的细节在图中被省略掉。同样,方法的raise子句也被省略。[21.5 小节](#)将讨论这些OTS的原始API。[21.6 小节](#)讨论OTS如何通过建立在其他的CORBA部件之上提供一个简化API。

21.5 OTS的原始API

Resource 接口是资源(数据库)封装器。定义在这个接口上的方法和基于 C 的 XA 的 API 很相似。OTS 的实现者必须提供 Resource 的实现,主要完成一些繁琐底层 XA C 基础的 API 调用代理。这就意味着服务器开发者将要做在 OTS 和 XA 兼容数据库的集成。如果服务器开发人员使用了 XA 不兼容的数据库,那么他们将不得不自己为那个数据库实现 resource 接口。

OTS 的实现提供事务管理器(TM)。规范并没有明确这个 TM 应该是何种形式发布,例如一个服务器进程,或可以链接在其他程序的库。通常情况下, TM 是

一个独立的应用程序。不管 TM 是什么样的发布方式，TM 包含了一个一些接口 TransactionFactory, Control, Terminator, Coordinator and RecoveryCoordinator 的预实现。

CORBA规范并明确OTS客户端如何实现和TM中的TransactionFactory的连接，所以这些机制根据CORBA产品的不同而不同。然而，这样的连接很有可能通过调用resolve_initial_reference()。这个函数的讨论在[3.4.1 小节](#)。OTS的客户端调用TransactionFactory::create()来开始一个事务。这个方法返回Control对象的对象引用。

客户端在调用OTS的对象时必须有某种传输Control对象引用的方法。这可以通过将Control引用作为这个对象的参数来实现。然而更常用的一种做法是将Control引用(和其他的一些信息)嵌入服务上下文中([11.6 小节](#))传输请求。并且OTS为这样的传输方式定义一个服务上下文结构。

当客户端希望结束一个交易，它调用 Control::get_terminator()方法获得 Terminator 对象的引用，然后调用此对象上的 commit()和 rollback()方法来结束事务。

如果OTS服务器要访问XA兼容的数据库，那么服务器将调用传递Resource数据库封装器的OTS方法(没有在[图 21.3](#)中罗列)。如果服务器使用XA不兼容的数据库，那么服务器开发人员必须实现资源接口，以使得它所使用的数据库能加入到两段提交的事务中去。

原始的OTS规范中，一个对象显示它具有参加OTS事务能力，是通过它实现一个从CosTransactions::TransactionalObject继承的IDL接口来做到的。_is_a()方法(由基类Object类型提供)可用于提供给客户端应用程序判断这个对象是否一个事务相关对象。然而OMG觉得这种方法是不适合的，因为这样

会增加接口定义的数量。最终OMG决定一个对象是否具有参加OTS事务的能力，是由这个对象的服务质量来决定的。在现代的OTS版本中，这个是通过定义一个新的策略类型来([6.1.4 小节](#))完成，即，如果使用了这个策略类型，那么在POA内的对象就是事务性的。IOR的拦截器([14.1 小节](#))检测出有这个POA策略，就嵌入一个OTS TaggedComponent([10.2.3 小节](#))到原IOR中。客户端可以检查IOR中是否有这个TaggedComponent来判断该对象引用是否是事务性的。

当OTS的服务器上的一个方法接收到一个Control对象时，这个方法可以调用 `get_coordinator()` 方法获得访问事务的Coordinator对象。Coordinator接口封装了实现两段提交协议的协同逻辑。它的目的是和OTS服务器中Resource对象交互。服务器调用Coordinator上的 `register_resource()` 注册自己的资源(每个事务中只注册一次)。这个注册通知TM，服务器的资源加入到事务中，当事务提交时，它将被加入到两段提交中去。这个方法返回一个事务RecoveryCoordinator对象的引用。服务器将这个对象存储到持久化存储区以便服务器在两段提交中重启时通过RecoveryCoordinator判断事务是否应该提交或回滚。

在两段提交中，TM调用所有加入到事务中的Resource对象上的 `prepare()` 方法。这个方法返回值Vote用来判断事务是否应该提交或回滚。

21.6 如何在其他CORBA部件上建立OTS

本节简要的讨论OTS提供的API的一个简单子集。这个简单的子集被大多数的开发者使用。本节讨论的目的不是给开发者一个入门手册，而是向开发者展示在CORBA中用于建立更强大模块的其他方面的概念(例如，Current Object，可移植拦截器，和服务上下文)

OTS定义了一个当前对象Current Object(13 章)。这个对象通过调用 `resovle_initial_reference("TranscationCurrent")` 方法获得。OTS 的当前对象([图 21.3](#) 中定义)让客户端和服务端端的线程知道它所关联的事务。

OTS客户端使用Current Object上的`begin()`, `commit()`和`rollback()`方法控制事务的生命周期。内部当前对象调用事务管理器定义相应接口的方法。当客户端调用一个对象上的方法,一个OTS提供的可移植请求拦截器([14.2 小节](#))从当前对象中获得交易上下文信息嵌入到一个即将发送目标对象请求的服务上下文中。在服务器端相应的请求拦截器解出服务上下文中的事务上下文信息并在目标对象方法体内初始化服务器的当前对象。这就意味着服务器方法体将在事务上下文中执行。因为这样,这个方法不需要开始-提交或者继续-挂起事务。而是相关的细节由可移植拦截器处理,于是,方法体可以把精力集中在使用嵌入式SQL或JDBC完成对数据库的查询。

上述讨论的提供给开发者的简单 API 子集机制对于大部分程序已经足够强大。然而开发者可以如果他们愿意,可以不使用当前对象,可移植拦截器,而自己手工的使用他们自己的 OTS 基础模块代码。虽然这很复杂,但是这提供给开发者一个非 XA 兼容数据库和 OTS 继承的道路。

第22章、 发布与订阅服务 Publish Subscribe Service

- [什么是发布与订阅](#)
- [事件服务](#)
- [通知服务](#)
- [电信日志服务Telecom Log Service](#)

22.1 什么是发布与订阅

CORBA 中的默认通讯方式是从某个客户端调用服务器中的某个对象。这是一对一, 点对点的通讯方式。相比而言, 发布和订阅 *publish and subscribe* (通常简称 *pub-sub*) 通讯方式是一个应用程序“发布 *publishes*”(即发送) 一个消息到某一个特定主题 *topic*, 所有的其他“订阅 *subscribe*”这个主题的应用程序接收消息。这是一个一对多的通讯方式, 本质上是一种异步的通讯方式, 因为发布消息的应用程序并不需要等待接收消息的应用程序返回信息。

计算机系统的中邮件系统和 *pub-sub* 通讯方式很相似。例如, 让我假设, ACME 公司有一个不同部门的邮件列表 `eng-staff@acme.com` 是到工程部门, `sale-staff@acme.com` 是到销售部门, 等等, 以下的几点值得注意。

- 如果你发送消息到 `eng-staff@acme.com` 那么你(“发布者”)仅仅发送了一条消息, 但是它被很多人(工程部门中所有“订阅”了这个邮件列表的人)接收。这就意味着, 邮件列表提供了一种一对多的通讯方式
- 当你发送了一封邮件, 你不需要等待邮件被所有订阅了邮件列表的人接收。而是, 由 EMAIL 服务器在后台发送信息(这个发送过程如果遇到网络问题的话有可能几小时或几天)。这就意味着发送邮件消息到邮件列表是一个异步的通讯方式。
- 当你发送一封邮件, 你不用等待应答。这意味着邮件是一种单向的通讯方式。因此, 一个收到你邮件的人, 可能发送你的另外的一封邮件地址做为应答。这就说明, 邮件有可能通过两条单向的通讯方式来模拟一个双向通讯, 但是在邮件系统底层本质上是单向的。
- 在 ACME 公司中存在着多个邮件列表。一个人可能发送一封邮件到 `eng-staff@acme.com` 和 `sales-staff@acme.com` 邮件列表。一些 ACME 的员工有可能只订阅了一个邮件列表, 其他员工订阅了超过一个的邮

件列表。相似的，pub-sub 系统中可能有应用程序可以发送到或应用程序可以订阅的多个“主题 topic”。

- 当你发送邮件到邮件列表，你的消息最初是到了邮件列表计算机，这台机器转发消息 N 次——邮件列表中每个订阅者一次。一些 pub-sub 以同样的方式工作，但是一些 pub-sub 使用多播或广播的协议这样它们消息只需要以此传输（而非 N+1 次），并且同时被所有的订阅者接收。

22.1.1、 模拟不同的通讯方式

一些中间件系统，例如 TIBCO Rendezvous 就是基于发布-订阅通讯方式。一些其他的中间件，例如 IBM 的 MQ series 是基于异步点对点的通讯。还有一些中间件，例如 CORBA 是基于同步，点对点通讯方式

上述任何其中的一个中间件都可以模拟其他中间件的通讯方式。例如，CORBA 可以通过 oneway 调用或者 CORBA 消息 ([16 章](#)) 提供点对点，异步通讯方式。CORBA 也可以通过本章讨论的各种服务提供 pub-sub 的通讯方式。

22.1.2、 CORBA 的发布订阅服务

CORBA 的事件服务 *Event Service* ([22.2 小节](#)) 提供了 pub-sub 通讯的基础形式。通知服务 *Notification Service* ([22.3 小节](#)) 提供更丰富的 pub-sub 通讯方式，并扩展了事件服务。最后电信日志服务 *Telecom Log Service* ([22.4 小节](#)) 通过提供了反复使用的永久日志消息扩展了通知服务。

CORBA 使用的术语和我们经常讨论的术语不同。CORBA 没有使用发布者 *publisher* 和订阅者 *subscriber*，而是使用术语生产者 *supplier* 和消费者 *consumer*。CORBA 没有使用主题 *topic* 和邮件列表 *maillist* 术语而是

使用术语事件通道 *event channel*。

22.2 事件服务

CORBA的事件服务提供 推push和拉pull两种通讯模型。push模型和我讨论的过pub-sub系统比较相似。所以我将在 [22.2.1 小节](#)先讨论push模型，然后在 [22.2.2 小节](#)讨论pull模型。

22.2.1、 推模型Push Model

[图 22.1](#) 和推push模型相关的IDL接口定义。为了简化，方法的raise子句被省略。[图 22.2](#) 显示了各种接口之间是如何交互的。

```
module CosEventComm {  
    interface PushConsumer {  
        void push(in any data);  
        void disconnect_push_consumer();  
    };  
    interface PushSupplier {  
        void disconnect_push_supplier();  
    };  
};  
  
module CosEventChannelAdmin {  
    interface ProxyPushConsumer  
        : CosEventComm::PushConsumer  
    {  
        void connect_push_supplier(  
            in CosEventComm::PushSupplier push_supplier);  
    };  
    interface ProxyPushSupplier
```

```

    : CosEventComm::PushSupplier
{
    void connect_push_consumer(
        in CosEventComm::PushConsumer push_consumer);
};

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();
};
};

```

图 22.1 事件服务推模型的 IDL 定义

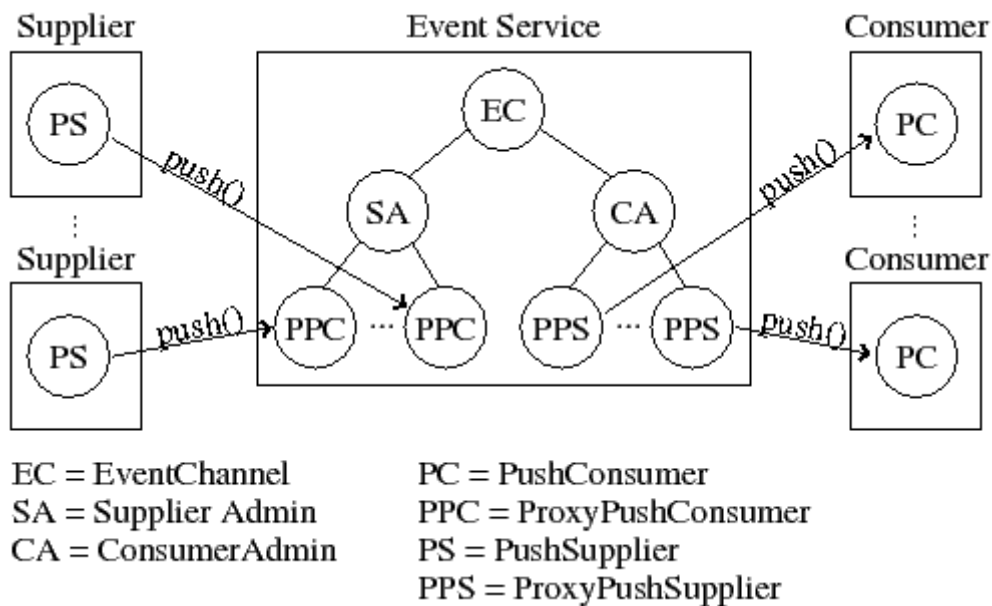


图 22.2: 事件服务的推模型

消费者consumer应用程序必须实现PushComsumer接口，他有一个push()方法，被调用时需要传递一个包含事件相关的任何数据的any类型(15.3)。PushComsumer接口中同样有一个disconnect_push_consumer()的方法，当事件服务决定断开自己和消费者consumer的连接时调用。例如，当EventChannel被destory的时候。

生产者supplier应用程序必须实现PushSupplier接口。这实际上是一个回调接口(1.4.2.2 小节)。事件服务如果想断开和生产者supplier的连接时就会调用disconnect_push_supplier()方法，例如，当EventChannel被destory的时候。

EventChannel是和事件服务的初始连接点。这个接口将其功能分散在SupplierAdmin和ComsumerAdmin两个对象中。并且提供了访问这两个对象的方法。

SupplierAdmin 是一个工厂接口(1.4.2.1 小节)。obtain_push_consumer()(译者注: 此处原文是obtain_push_supplier

应为作者笔误)方法创建一个ProxyPushConsumer。在名字前使用“代理Proxy”和IDL编译器生成代码无关。而是ProxyPushConsumer在事件服务中是一个委托对象:一个生产者supplier调用ProxyPushConsumer上的push()方法,它将调用(或者安排事件服务的其他对象来调用)在消费者应用程序上的PushConsumer的push方法

生产者应用程序的初始化包括如下的两步: 第一步连接 EventChannel。然后调用 for_supplier()得到一个 SupplierAdmin 对象,并接着调用 obtain_push_consumer()创建 ProxyPushConsumer 对象。最后生产者 supplier 调用 connect_push_supplier()来注册自己的 PushSupplier 对象。到这里,生产者已经完全连接到事件服务上,并可以调用 ProxyPushConsumer 的 push 方法。

消费者应用程序初始化是生产者初始化过程的一个镜像。第一步连接 EventChannel。然后调用 for_consumer()得到一个 ConsumerAdmin 对象,并接着调用 obtain_push_supplier()创建 ProxyPushSupplier 对象。最后消费者 Consumer 调用 connect_push_consumer()来注册自己的 PushConsumer 对象。到这里,生产者已经完全连接到事件服务上,一旦事件发生,它的 push 方法就会被回调。

22.2.2、拉模型Pull Model

push 模型被命名的原因是因为, push 模型会将数据推向(proxy)消费者。相反地, pull 模型命名的原因是它是从(proxy)生产者拉数据。拉模型 IDL 的其他部分定义如图 22.3

```
module CosEventComm {
```

```

...
interface PullSupplier {
    any pull();
    any try_pull(out boolean has_event);
    void disconnect_pull_supplier();
};
interface PullConsumer {
    void disconnect_pull_consumer();
};
};
module CosEventChannelAdmin {
    ...
    interface ProxyPullConsumer
        : CosEventComm::PullConsumer
    {
        void connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier);
    };
    interface ProxyPullSupplier
        : CosEventComm::PullSupplier
    {
        void connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer);
    };
};
};

```

图 22.3 事件服务的拉模型 IDL

PullSupplier 上的 pull() 方法是一个阻塞操作。try_pull() 是一个非阻塞操作，此方法立即返回，has_event 输出参数指定返回的 any 类型是否

是空值。

事件服务规范定义了额外的接口，可以以强类型的 API 来传递数据，而不用将数据打包在 any 类型中。然而，带类型的事件服务 Typed Event Service 是厂家非常难实现的(由于当前 CORBA 中一些不成熟的规范)，因此只有很少的厂家实现了它。因为这样，我将不在这里讨论带类型的事件服务。

八二法则 [\[Koc00\]](#)适用于软件产品：“80%的人只使用软件中 20%的功能”。当然，这个比例不总是 80 和 20，但是大多数人只用到软件产品中很少功能的原则依然是正确的。这个法则适用于事件服务。大多数人使用push模型，很少有人使用pull模型和带类型push/pull模型。

22.2.3、事件服务的局限

事件服务有如下的几个缺陷，我将在本节中详细讨论。

一个局限就是事件服务没有为事件通道定义factory接口([1.4.2.1 小节](#))。这就意味者事件服务的实现者只能有一个事件通道。概念上来说，和一个公司他只使用一个内部邮件列表非常相似。这样结果是订阅了所有的消息，甚至订阅者对该消息并不感兴趣。许多事件服务的实现者通过提供他们自己的私有 API 来克服这个缺陷。然而使用这些私有 API 明显会影响程序的可移植性。

另外一个局限就是事件服务规范并没有定义实现需要提供的服务质量。例如：

- 事件服务的实现者是否应该只把连接的生产者和消费者记录在内存中，在这种情况下，当事件服务器被杀掉重启时，所有的连接将都会丢失，或是否这些信息应该被存储到持久化缓存区，这样即时事件服务器重启，连接信息仍然可以被维护。
- 是否事件服务的实现者应该仅仅将还未发送的消息存储在内存中，在这种情况下，在事件服务器被杀掉重启时，消息将会丢失，或是应该将未发送消息存储到文件或数据库中，这样事件服务器重启的时候消息信息不会丢失。

- 是否当事件服务发送消息到消费者发生困难的时候，事件服务应该在第一次发送失败放弃发送消息？或事件服务应该重新尝试发送多次？如果事件服务应该尝试重发几次，那么再每次重发之间应该等待多长时间？还有它是应该在重发 n 次失败后放弃，还是经过特定时间后放弃？

事件服务规范故意没有定义它本应该提供的服务质量规范。这样做的原因是鼓励不同的厂家相互竞争，并提供不同的服务质量。然而这样策略因为如下的两个原因并没有达到原本的目的。

第一，大多数事件服务的实现者宁愿将所有消息和连接的生产者消费者信息保存在内存数据库中，而不愿将信息保存在文件或数据库中。这就意味着这里并不存在提供不同服务质量的竞争。

第二，一个应用程序在同一时刻可能会希望多种服务质量。例如，假设厂家提供了 `EventChannel` 工厂作为一个私有增强，而生产者应用程序可能想通过一个带有一些服务质量的 `EventChannel` 发送一些信息，而带有另外一些服务质量的另外一个 `EventChannel` 发送另外的一些信息。然而大部分的厂家却提供的是一个服务质量必须应用在所有的 `EventChannel` 中。

由于上述提到的缺陷非常不幸地导致了 `EventChannel` 并不能满足许多应用程序的需求。

22.3 通知服务Notification Service

通知服务Notification Service有时被称为“增强事件服务”或“事件服务++”。正如我将如下讨论的，通知服务去除了事件服务中 [22.2.3 小节](#)讨论的所有的局限，并增加额外的功能，这样就使得发布-订阅系统变得更灵活，更强大。

22.3.1、 IDL接口

通知服务 Notification Service 向后兼容事件服务。向后兼容是通过通知服务的 IDL 接口都是从事件服务的接口中继承。此外，通知服务的名字命名规范和事件服务的名字命名规范几乎一样。向后兼容和相似的命名规范提供如下的两个好处：

- 向后兼容能力，使得开发人员在通知服务中重用那些基于事件服务的应用程序，并可以慢慢进行移植而达到可以使用通知服务提供的额外功能。
- 向后兼容和相似的命名规范使得开发者可以通过使用事件服务为踏脚石来学习通知服务提供的丰富的 API。

通知服务定义了 38 个 IDL 接口，这比通知服务继承的事件服务定义的 11 接口还多的多。总共这些接口加起来就有 49 个接口。整个接口数多的令人吃惊，但是庆幸的是，大多数的程序员只使用其中一小部分接口。其他大多数接口提供的是管理类功能，并且多数的厂家提供了命令行和图形界面工具来和这些管理类接口进行交互，因此，开发人员不需要写任何代码做这事。

22.3.2、 StructureEvent

通知服务允许事件数据传送any类型，这是为了满足向后兼容的需求。然而通知服务提供另外一种格式的称为StructureEvent的事件数据，如 [图 22.4](#)。事实上这个图是一个简化的IDL定义，为了显示更简洁，一些typedef被移走。例如，EventHeader 的 variable_header 类型真实是一个 typedef sequence的typedef 。

```
module CosNotification {  
    struct Property {  
        string name;  
        any    value;  
    }  
}
```

```

};

struct EventType {
    string domain_name;
    string type_name;
};

struct FixedEventHeader {
    EventType event_type;
    string event_name;
};

struct EventHeader {
    FixedEventHeader fixed_header;
    sequence<Property> variable_header;
};

struct StructuredEvent {
    EventHeader header;
    sequence<Property> filterable_data;
    any remainder_of_body;
};

typedef sequence<StructuredEvent> EventBatch;

...
};

```

图 22.4 通知服务的事件数据伪 IDL 定义

StructureHeader 的 EventType 中有两个字符串域。domain_name 应该被设定为标明一个特定的纵向工业 vertical industry 名，例如，“电信行业 Telecoms”，type_name 的值唯一标示在此域中一个事件的类型，例如，CommunicationAlarm。EventHeader 的剩下部分是一个 Property 可变数组类型，Property 是一个名字值对。程序员可以放置任何他们需要的名字值对

在这个可变数组中，但是更多的情况下，Property 是被用于表达成用于控制消息发送服务质量(QoS)，例如，消息的优先级和超时时间等。通知服务可以通过在 EventHeader 中使用强类型域指定特定的 QoS。然而，使用弱类型的名字值对有两个好处：一个好处是，它降低了 EventHeader 的大小，多数情况下生产者更喜欢使用默认的服务质量。另外一个好处是，在以后，通过新增名字-值对可以提供一种向后兼容的方式来修订通知服务。同样，名字值对允许厂商提供自己私有的服务质量。

在事件数据头之后，StructureEvent的filterable_part提供了另外一个可变数组的名字值对。这个可变数组中用户可以用来放置他们的(大部分或全部)事件数据。使用名字值对代表事件数据是：选择难使用单一any类型(事件服务提供的方式)和没有扩展性但是编译时期类型安全的带固定域的struct，这两种方式的一种折中做法。OMG的目的是在不同的域中通过定义那个名字值对来代表所指定的事件类型。例如，来自不同电信公司的代表将会一起为不同的事件类型定义某个名字值对来代表它，例如CommunicationAlarm，并以它的行业相关。这些名字-值对被称为filterable_data因为他能通过过滤器Filter([22.3.4 小节](#))来访问。

StructureEvent 的最后一个域是 any 类型。这个域允许你存储任何数据值，并且这些数据不会被用于过滤器 filter，也许因为数据是一个大的二进制数据类型，比如说，文件的内容。

22.3.3、 EventBatch

通知服务可以用于发送和接收EventBatch([图 22.4](#))，这是一个 StructureEvent的可变数组。这允许应用程序用块的形式来高吞吐量的发送

接收事件。例如，让我假设生产者应用程序每秒生产了 1000 个事件。生产者应用程序不是每秒一个个的发送着 1000 个事件，而是它先生成 1000 个事件的数组，然后将这个数据发送出去。

消费者通过EventBatch来接收事件，而不是单独的一个个的接收事件。正如我将在 [22.3.7 小节](#) 讨论的那样，应用程序可以指定各种各样和通知服务交互的服务质量。消费者程序可以通过服务质量指定它希望的批事件中的事件个数 *batch size* 和步长的间隔 *pacing interval*。步长间隔是两个批事件发送的最大间隔。例如，让我假设，消费者指定批事件的个数为 1000 个，步长间隔为 10 秒。经过 10 秒后，虽然事件通道中只有 300 个事件，通知服务仍然会将这个 300 个事件的批发向消费者，而不是等待直到 1000 个事件全部收到后再将他们发送给消费者。

关于应用程序如何指定以一个StructureEvent或EventBatch中单独的any类型形式消费/生产事件，我将推迟在 [22.3.5 小节](#) 讨论。

22.3.4、过滤器Filters

过滤器 *filter* 是一个约束(条件)集合的封装器。过滤器应用于通过通知服务中的消息。从语法上说，存在两类过滤器：Filter 和 MappingFilter。过滤器工厂 FilterFactory 用于创建这两类过滤器。

```
module CosNotifyFilter {
    interface FilterFactory {
        Filter      create_filter(...) raises(...);
        MappingFilter create_mapping_filter(...)
                                raises(...);
    };
};
```

22.3.4.1、 移除消息过滤器Filter to Remove Message

这类过滤器的功能是在传输到消费者之前删除消息。这样做就省去了消费者应用程序检查其接收到的消息是否和他有关。同样这样做也节省了网络流量，因为消息被传送之前就被删除了。

过滤器中的约束以扩展交易约束语言 Extended Trade Constraint Language (ETCL) 表示，根据名字可以看出，这是用于交易服务的交易约束语言 Trade Constraint Language (TCL) 增强性语言。一个约束是用 StructEvent ([22.3.2 小节](#)) 中名字值对写成的一个布尔表达式。约束同样可以是事件的 domain_name 或 type_name。

正如前面提到的，过滤器封装了一个约束的集合(用一个可变数组表示)。如果任何在 Filter 中的约束的评估值为 true 那么消息就允许通过。换句话说，仅当所有约束的评估值为 false 的时候，消息被丢弃。

过滤器可以附加在所有 push/pull/consumer/supplier 的代理对象的组合上。然而通常，过滤器只附加在代理消费者 consumer 对象而不是代理代理生产者上，这是因为，用消费者而非生产者来过滤事件这是非常符合情理的。

过滤器也可以附加在 SupplierAdmin 或 ConsumerAdmin 上，这样做的好处在 [22.3.5 小节](#) 讨论。

你可以附加多个过滤器在代理和管理对象上。如果这样，消息仅在所有过滤器的所有条件都为 false 时才过滤掉事件。

22.3.4.2、 消息超时和优先级过滤器 Filter for Message Timeout and Priorities

StructureEvent 的头可以包含指定消息优先级和最后发送期限的名字值对。该名字值对是专为生产者指定。MappingFilter 允许消费者，实际上是 ConsumerAdmin 和生产代理对象扮演消费者的行为来覆盖消息中的优先级和发送最后期限。名字 MappingFilter 并不直观，OverrideFilter 是更直接的名字。

消费者管理对象或生产者代理对象能有两个 MappingFilter 对象和它关联。一个用于覆盖消息的优先级，另外一个用于覆盖消息发送的最后时间期限。

[22.3.4.1 小节](#)提到的 Filter 是一组约束集合的封装器。相反地，MappingFilter 是一个约束-值 *constraint-value* 对封装器。如果约束评估值为 true 那么 MappingFilter 中值 *value* 将覆盖消息优先级或者发送最后期限值。因为表示优先级和发送最后期限的 IDL 定义不同，所以值 *value* 用 any 类型来表示。

22.3.5、 ConsumerAdmin 和 SupplierAdmin

[图 22.5](#) 罗列了 SupplierAdmin 和 ConsumerAdmin 接口中 create-风格的方法。这些方法中都带一个 ClientType 参数，该参数表示生产者和消费者是否以 any, StructureEvent, EventBatch 处理事件。create-风格的方法将创建一个指定所希望类型 (any, StructureEvent, EventBatch 三类之一) 的代理对象。这些代理对象是 ProxySupplier 和 ProxyConsumer 的子类。

```
module CosNotifyChannelAdmin {  
    ...  
}
```



```

enum ClientType {ANY_EVENT, STRUCTURED_EVENT,
                  SEQUENCE_EVENT};

interface ConsumerAdmin
    : CosNotifyFilter::FilterAdmin,
    // rest of inheritance clause omitted
{
    attribute CosNotifyFilter::MappingFilter
        priority_filter;
    attribute CosNotifyFilter::MappingFilter
        lifetime_filter;
    ProxySupplier obtain_notification_pull_supplier(
        in ClientType ctype, ...) raises (...);
    ProxySupplier obtain_notification_push_supplier(
        in ClientType ctype, ...) raises (...);
    ...
};

interface SupplierAdmin
    : CosNotifyFilter::FilterAdmin,
    // rest of inheritance clause omitted
{
    ProxyConsumer obtain_notification_pull_consumer(
        in ClientType ctype, ...) raises (...);
    ProxyConsumer obtain_notification_push_consumer(
        in ClientType ctype, ...) raises (...);
    ...
};
};

```

图 22.5 通知服务的管理对象 IDL 定义

ConsumerAdmin 和 SupplierAdmin 都是从 FilterAdmin 继承，这些

对象提供关联 Filter 到 Admin 对象上的方法。此外，ConsumerAdmin 还有两个 MappingFilter 属性，用于覆盖消息优先级信息和发送最后期限信息。

通过使 Admin 对象关联到过滤器对象上，可以使过滤器过滤一组而非单个的消费者。不仅从管理上变得更便利，而且在效率上也得到很大的提高，因为只需要评估一次就能作用于整个组，而非重复的执行约束条件评估。为 SupplierAdmin 提供过滤器并没有多大用处(这是因为过滤器主要是在消费者端)，但是提供这个功能的原因仅仅是为了完成功能上对称。

22.3.6、 EventChannel

事件服务的一个局限是它指定了一个事件通道；没有定义可以用于创建多个事件通道对象的事件通道工厂对象([1.4.2.1 小节](#))。通知服务突破了这个限制，[图 22.6](#) 提供了事件通道和事件通道工厂对象的定义。

```
module CosNotifyChannelAdmin {  
    ...  
    interface EventChannel  
        : // inheritance clause omitted  
    {  
        readonly attribute ConsumerAdmin  
            default_consumer_admin;  
        readonly attribute SupplierAdmin  
            default_supplier_admin;  
        readonly attribute CosNotifyFilter::FilterFactory  
            default_filter_factory;  
        ConsumerAdmin new_for_consumers(...);  
        SupplierAdmin new_for_suppliers(...);  
        ...  
    };  
};
```

```

interface EventChannelFactory
{
    EventChannel create_channel(...) raises(...);

    ...
};
};

```

图 22.6 通知服务的事件通道和事件通道工厂 IDL 定义

工厂接口的 `create_channel()` 方法用于创建 `EventChannel` 的方法。传递给这个工厂的参数(在图 22.6 中被缺省)用于指定新创建事件通道的服务质量。[22.3.7 小节](#)将会提供一个关于服务质量的概述。

每一个 `EventChannel` 提供了一套预创建好的 `ConsumerAdmin` 和 `SupplierAdmin`，和 `FilterFactory` 对象。这些对象可以通过 `default_<...>` 属性来访问。然而 `new_for_consumer()` 和 `new_for_supplier()` 方法允许创建新的对象。通过为不同的消费者/生产者组创建不同的管理对象，可以使得通知服务为不同的消费者/生产者组指定不同的服务质量。同样，这样的做法可以让在 `ConsumerAdmin` 过滤消息的行为作用在一组消费者对象上。这样组级的过滤比单个代理对象粒度过滤更高效(同时也更具有扩展性)。

22.3.7、 服务质量(QoS)

通知服务允许用户为传输事件选择不同的服务质量。下面几点简要的罗列一些通知服务中重要的服务质量：

- 在通知服务中，你可以创建很多通道，每个通道可用于传输不同类型的事件。这和一个公司为了使讨论不同的主题更容易而有多多个邮件列表概念上非常相似，通道可以是持久化 *persistent* 或高效的 *best effort* (这是通知

服务中非持久化的术语)。如果通道是持久化的,那么所有未发送消息将会被通道的基础功能模块持久化到文件或数据库中,因此当通知服务终止重启,这些消息将依然存在。如果通道是高效的 best effort,那么未发送信息只会存储在内存中,这就意味着对于高效 best effort 通道,当通知服务进程终止重启,所有的未发送消息也相应的不再继续存在。

- 高效 best-effort 通道可以仅用于传输高效 best-effort 事件,相反持久通道即可以传输持久事件也可以传输高效事件,如果是高效 best-effort 事件,服务器重启所有的还未发送的高效事件将被丢弃。
- 当通知服务从生产者收到一个事件,通知服务可能不能马上将事件传给所有的消费者。例如,消费者应用程序没有在运行状态,那么通知服务将周期性进行重发事件到消费者,这有相应的几种策略来控制通知服务在其丢弃事件前,怎样重试发送事件到消费。
 - 还未发送的到消费者的事件可以如下的次序被发送:先进先出的次序,优先级次序(基于事件头中的可选优先级),最后期限次序(基于事件头中的发送最后期限)或任何次序(任何一种通知服务想要的次序方式)
 - 可以在保存未发送到消费着消息事件的队列上设置一个队列上限。
 - 如果通知服务将要超出了消费者消息还未发送的队列上限,那么它将丢弃事件,丢弃事件的策略可以和发送数据的策略相同。例如:先进先出,优先级,最后发送期限等等。
 - 如果试图发送事件失败,重发该事件需要等待多长的时间。
 - 重发最大次数,多少此重发事件失败后,通知服务丢弃该事件
 - 发送超时时间,发送事件的方法的最大超时时间。
- 能连接到通知服务的最大消费者/生产者数。

通道的服务质量通过名字-值对指定,QOS可以在创建通道的时候指定([图 22.6](#))。这些服务质量被ConsumerAdmin和SupplierAdmin对象级别或单独的代理对象上覆盖。

22.4 电信日志服务Telecom Log Service

在通知服务中 ([22.3 小节](#))，事件通道创建持久的QoS存储每个事件在持久化存储中 (例如，文件或数据库)，直到事件被发送到消费者。在这时事件被从持久化存储中删除。一些组织希望能永久性的记录事件。他们通过电信日志服务 Telecom Log Service来做到这点，之所以这样命名，是因为它是由电信行业的公司定义的，但是这个功能同样适用于其他的行业。

电信日志服务中最重要的接口是NotifyLogFactory，这是一个工厂对象 ([1.4.2.1 小节](#))，这个工厂用于创建一个NotifyLog对象，这个接口继承于通知服务的EventChannel。

因为电信日志服务借用了通知服务的基础功能模块，NotifyLog同样可以关联一个过滤器 ([22.3.4.1 小节](#)) 对象。这就意味着电信日志服务可以有选择的来存储记录。

NotifyLog 有一个方法允许迭代存储在持久化区的所有事件。你可以查询 `query()` 特定约束的事件。你也可以提取 `retrieve()` 特定时间后发生的事件。

电信日志服务给你许多控制持久化存储 NotifyLog 对象，例如：

- 你可以指定 NotifyLog 对象需要为事件分配在存储去的空间。
- 你可以指定当存储区满后，NotifyLog 的行为。两个可选的策略，一个是 `wrap` 策略，允许你覆盖较老的事件，或使用 `halt` 策略，定制继续记录新事件。
- 你可以指定当电信日志服务的存储使用率达到一定百分比时分产生一个事件。这允许日志在到快满的时候用户采取一定的行动。例如创建一个新日志来存储旧日志内容。
- 你可以从日志中删除单个事件，或匹配某条件的事件。
- 你可以指定当日志中事件存储一段时间后自动删除日志中的一些事件。

- 你可以指定日志中只存储特定时间的事件。

第23章、 安全 Security

(略)

第24章、 本书未讨论的服务

- [持久化状态服务Persistent State Service\(PSS\)](#)
- [其他CORBA服务和域规范](#)

本章简要讨论一下本书未讨论的其他 CORBA 服务。感兴趣的读者可以从其他书中找到这些主题相关讨论，相关的文章和其他的一些 internet 资源都罗列在 26 章。

24.1 持久化状态服务Persistent State Service(PSS)

许多真实世界的应用程序都被要求具有在持久化存储维护数据的能力。在这样的应用程序中，底层 API 用于访问设备，例如文件或数据库的程序经常和业务逻辑混在一起。此外用于访问不同种持久化存储的 API 也是千差万别。这样就会导致将程序使用的持久化介质从一种移植到另外一种持久化介质需要花非常多的时间。比如，让我假设，你需要实现维护一些数据到持久化存储 CORBA 服务器的任务。开发或维护这个程序，需要做到如下的几个步骤：

应用程序的数据存储的需求并不复杂，所以你决定持久化数据在文件中，你

1、主程序和你的 IDL 方法的主体中很可能包含有“业务逻辑”和打开，关闭，读写文件的代码。当你完成了任务，并把它部署好后，程序工作的非常好。

2、在第二年，你装载数据越来越多，最终在文件型数据库持久化机制中越来越多的局限被显示了出来。所以你决定使用一个对象型数据库来替换文件型数据库。替换的需求是将代码中文件访问的代码替换成特定的访问对象数据库的 API。这种修改带来了几周甚至几月的辛苦劳动，但是一旦这个工作完成，你将会非常高兴的发现你的程序能很容易的提供对大数据处理的能力。

3、后来的一年，你公司合并了另外一个公司。在新合并的公司中，定下了一个在公司中使用第三方软件的使用范围的扩展决定。由于这样，你被告知你的 CORBA 服务器中的面向对象数据库访问代码就被替换成，例如，ORACLE，这样的关系型数据库。这样你又需要花上几周甚至几个月的时间来做必要的修改。

持久化状态服务的 Persistent State Service PSS 的目的是，减少因为从一种持久存储移植到另外一种持久化存储的需求。PSS 是一个抽象层，他隔离了开发人员和底层持久化设备技术和 API。PSS 只处理持久化数据。他并不提供事务和查询能力。因为这样，他并不是银弹，但是简单化使用持久化机制能满足很多应用程序的需求。

持久化状态服务 PSS 提供了持久化状态定义语言 Persistent State Definition Language(PSDL)，他是 IDL 的超集。PSDL 用于定义持久化的数据类型。PSDL 编译器将 PSDL 定义类型转换成程序语言(比如 C++或 Java)。PSDL 编译同样会生成这些类型持久化到某一种存储的代码，例如文件或数据库。

PSS 为不同的持久化设备，例如文件，对象数据库，和关系数据库实现是 PSS 的长远目标。使用 PSS 的应用程序开发者将能很容易从基于文件的 PSS 实现切换到基于数据库的 PSS 实现，或者反向。但是长远目标并没有完全的实现，一些 CORBA 的厂家提供了 PSS 的实现，但是这些厂家通常的做法是实现了 PSS

到一种存储介质的映射。你当然可能期望，某个 CORBA 厂家能够实现 PSS 对多种品牌数据库或多种存储技术的支持，但是这需要足够多客户对这些支持感兴趣。这很容易成为一个鸡生蛋还是蛋生鸡的问题：客户很可能避免使用 PSS，除非它已经支持多种存储技术，厂家不愿持久多种多种存储技术除非它有足够多 PSS 用户以值得它对多种存储技术支持。在决定使用 PSS 的时候有一点非常重要，你需要检查你的 CORBA 厂家(或第三方厂家)是否能提供你所需要的对所有存储技术的 PSS 实现

最后还有两点需要注意。第一，一个好的 PSS 技术概述可以在 *Java Programing with CORBA* 书[BVD01]中找到。第二，基于 PSS 的持久化，是 [19 章](#) 讨论的 CORBA 组件模型 (CCM) 提供的一个基础功能模块之一。

24.2 其他CORBA服务和域规范

OMG 的网站 <http://www.omg.org> 提供了免费下载的各种 CORBA 相关规范的 PDF 文档。通过浏览这个网站去看 CORBA 定义的扩展服务的范围是非常值得做的事情。这些服务只是 CORBA 规范的可选部分，所以你的厂家很可能只卖了这些服务子集的一部分。

需要注意的，CORBA 服务这个术语用于指在很多不同类型公司中对应用程序有用的可选的功能。名字，事件，事务是这些服务的一个例子。OMG 认为为特定目的的功能定义标准是非常有用的。这些功能被称为 OMG 域规范 OMG Domain Specification。OMG 网站根据规范的覆盖面例如，交通控制，音频/视频流，计算机辅助设计，电信，财务，账务，制造业，分布模拟，生命科学研究等来罗列域规范。CORBA 服务和 CORBA 规范的分界线是主观的，在 OMG 网站上，一些规范在所有的种类中都有。

如果你的厂家并没有实现你要求的特定的 CORBA 服务或 CORBA 域规范，那么

你可以从第三方公司买你需求功能的相关实现。你可以使用Internet搜索引擎
(例如: www.google.com) 搜索出售特定CORBA服务和域规范实现的公司。

第五部分、最后的问题

第25章、 CORBA应用程序的移植性

- [CORBA移植性问题](#)
- [C++的非CORBA移植性问题](#)

大部分的 CORBA 源代码都可以在不同的 CORBA 产品间移植。事实上, CORBA 的移植性是非常好的以至于在缺乏移植性的领域讨论这问题也是比较容易的。二八法则同样可以运用于移植, 应用程序最难移植的部分归根结底只是几个问题。通过对这个移植问题的预警, 开发组可以预先做一些工作避免移植过程中最容易出现的陷阱并使得移植变得容易。本章集中 C++和 Java 的移植性问题, 因为这些语言是作者用于开发 CORBA 的语言。

25.1 CORBA移植性问题

CORBA 有将 IDL 映射成不同程序语言的标准, 例如 C++, Java, Cobol, Ada 等等。这意味着程序维护一个 struct 或 union, 使用代理或 Servant 类的接口实现进行一个远程调用在所有的 C++ CORBA 产品中将是完全一样的, 在 Java 的 CORBA 产品中也是完全一样的, 等等, 然而, 然而仍然有一些 CORBA 相关的移植问题需要被注意, 正如下面几节讨论的那样。

25.1.1、 Makefile问题

CORBA 并没有标准化 IDL 编译器的名字和相关的命令行参数。CORBA 也没有标准化程序中需要连接的库名称。潜在的影响就是 Makefile(或等价的文件, 如 Microsoft Visual Studio Project 文件)在你从一个 CORBA 产品移植

到另外一个 CORBA 产品时都需要进行修改。尽管是一些繁琐的工作，但是这些修改通常都是直接的。

25.1.2、CORBA相关的C++头文件名

IDL 到 C++映射定义了生成数据类型的 API。不幸的是，这个映射并没有标准化生成源代码文件的名字。例如，如果 `foo.idl` Orbix/C++ IDL 编译器会生成 `foo.hh`, `fooS.hh`, `fooC.cxx`, `fooS.cxx`。其他 CORBA 产品的 IDL 编译器提供可能使用不同的生成文件的前缀名。

改变Makefile中的源代码文件的名字并不是很大的问题，更大的麻烦是在源文件中修改许多的#include指令，因为项目中的一个或多个源文件都会包含一个或多个#include指令。CORBA工具包 [\[McH\]](#) 的C++ CORBA程序的移植性章节深度的讨论这个问题，并解释了如何在项目开始时花上几个小时对移植性问题做一些处理，可以生成可移植的#include指令，这样在项目需要移植的时候可以为你节省几天或几周的时间。

25.1.3、配置和日志API

CORBA 没有一些功能提供标准化的 API，这些功能包括(1)获得运行期配置(2)记录诊断信息在文件。然而大多数的 CORBA 厂家的私有 API 提供了这些功能。这是因为 CORBA 产品的内部需要这样的能力，而 CORBA 厂家决定经常暴露这些内部需求功能给应用程序开发人员。如果你避免使用私有 API，那么你移植达到其他 CORBA 产品将变的比较容易。如果你必须使用这些 API，那么你不应该直接使用这些 API，而应该写一些移植封装器 API 来封装这些 API。这样在移植的时候，你就最小化了你的移植修改。

25.1.4、 实现库Implementation Repositories

正如第 7 章讨论的，CORBA 并没有标准化实现库 Implementation Repository(IMS)的“观感 look and feel”，因为这个原因，必须考虑不同产品中实现库的不同。并且在跨 CORBA 产品的管理上，并没有可移植性。

一些服务器端的部署模型([8.2.2 小节](#))要求使用一些私有的API。直接的使用这些API将影响应用程序的移植性。CORBA工具包 [\[MCH\]](#) 的使POA层次创建更简单的章节中讨论了一个C++或Java类对这些私有API的封装，并同时也简化了应用程序的部署。

25.1.5、 多线程

[6.1.2.1 小节](#)解释ORB_CTRL_MODEL POA策略有用户自定义语义。大多数的产品要不就是为这个策略实现了线程池方式，要不就是提供了一个可供配置的线程池。因为这个原因，一般都假定这个策略的实现方式是线程池的方式。然而，一个例外是TAO，这是一个C++免费CORBA实现。TAO虽然支持ORB_CTRL_MODEL的线程池语义。然而，TAO要求应用程序员将所有线程都建立在线程池中。并且所有的这些线程都必须调用orb->run()方法，这样便使得应用级别的线程成为线程池的一部分

```
int main(int argc, char* argv[])
{
    thread_pool_size = ...;
    exit_status = 0;
    orb = CORBA::ORB::_nil();
    try {
        orb = CORBA::ORB_init(argc, argv);
        ... // create POAs to contain servants
    }
```

```

    ... // create servants and activate into POAs
    ... // export object references
    ... // activate POA managers
1   create_tao_thread_pool(orb, thread_pool_size - 1);
2   orb->run();
    } catch(const CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
3   wait_for_tao_thread_pool_to_terminate();
    // Terminate gracefully
    try {
        if (!CORBA::is_nil(orb)) { orb->destroy(); }
    } catch(CORBA::Exception & ex) {
        cout << "Something went wrong: " << ex << endl;
        exit_status = 1;
    }
    return exit_status;
};

```

图 25.1 为 TAO 线程池移植性使用可移植性抽象的服务器主程序

[图 25.1](#) 中的伪码显示了如何为 ORB_CTRL_MODEL POA 策略写一个使用线程池的可移植 C++ 服务器的伪码。其中最重要的是两个可移植性的创建和销毁线程的“封装”函数。这个函数用法在图中 1 行和 3 行。除了一个线程的其他所有线程池中的线程都在第一行创建；线程池中的最后一个线程是直接调用 `orb->run` 的主线程（第二行）。

可移植的封装函数在 [图 25.2](#) 中，这个实现使用了现存的 `P_USE_TAO` 预处理器符（这个宏的详细说明可以在 [[Mch](#), Ch. 4] 找到）来选择使用为 TAO 创建和

销毁线程代码，或为其他大多数CORBA实现选择空实现。

```
#ifndef P_USE_TAO
#include "ace/Task.h"
class Worker : public ACE_Task_Base {
    CORBA::ORB_var    m_orb;
public:
    Worker(CORBA::ORB_ptr orb) {
        m_orb = CORBA::ORB::_duplicate(orb);
    }
    virtual int svc(void) {
        try {
            m_orb->run();
        } catch(...) { }
        return 0;
    }
};
static Worker * w = 0;
void create_tao_thread_pool(CORBA::ORB_ptr orb, int count)
    throw(std::string)
{
    w = new Worker(orb);
    if (w->activate(THR_NEW_LWP | THR_JOINABLE, count)!=0)
    {
        delete w;
        w = 0;
        throw std::string("Cannot create thread pool");
    }
}
void wait_for_tao_thread_pool_to_terminate() {
```

```

        if (w != 0) {
            w->thr_mgr()->wait();
            delete w;
            w = 0;
        }
    }
}

#else // dummy implementation for other CORBA products
void create_tao_thread_pool(CORBA::ORB_ptr orb, int count)
    throw(std::string)
{ }

void wait_for_tao_thread_pool_to_terminate() { }

#endif

```

图 25.2 为 TAO 线程池可移植性封装

25.2 C++的非CORBA移植性问题

当从一个 CORBA 产品移植到另外一个 CORBA 产品，开发需要需要将编译器从一种移植到另外一种，或将操作系统从一种移植到另外一种。这样的移植对 Java 程序影响并不大，因为 Java(或被认为)提供了可以跨平台和 Java 编译器的“虚拟机”。事实上，当 Java 第一次发布，它就承诺它将是“一旦写成，处处运行 write once, run anywhere”的语言。当开发者注意到基于 Java 的程序都没有 C 或 C++的应用程序跑的快的时候，一些批评者认为 Java 是“一旦写成，处处慢爬 write once, crawl anywhere”语言。一些开发人员遭遇 Java 虚拟机在不同操作系统上的不同行为影响，于是认为 Java 是“一旦写成，处处调试 write once, debug anywhere”语言。然而性能问题和不同 JVM 的问题随着时间的推移而越来越少，如今，Java 程序越来越趋向具有非常好的移植性。

切换编译器或操作系统的比起 Java 来说基于 C++ 的程序具有更多的问题。

下面几节中讨论了你将在 C++ 程序中注意的容易出问题的地方。

25.2.1、跨平台可移植性

由于一些原因，在 Windows 上开发的可移植程序经常会不小心使用到 Windows 的私有 API。这些通常只有在程序移植到 unix 平台时才会发现。在那个时候，这些使用 Windows 私有 API 的程序在程序分布的如此广泛以至于移植这些 API 成为了主要的工作。例如，Windows 的私有 CString 类经常不小心在被移植的程序中使用。一些公司发现在他们要被移植的应用程序中 CString 的代码，并且他们发现使用逆向工程这个 CString 类，并实现一个 unix 版本要比从程序中移出这些代码更快。这些公司最后逆向了 Microsoft's MFC 类库的大部分，以至于他们可以写一个 unix 版本的 MFC，而仅仅为了移植一个 windows 应用程序到 unix 上。

为了避免在未来移植程序的过程中出现这样的问题，在团队中有一个具有 unix 经验的开发人员执行在 Windows 上开发初始化可移植程序任务。

如果你的 CORBA 应用程序需要一个 GUI，那么你可以考虑使用跨平台的 GUI 工具包。例如：mxWindows(www.mxwindows.org)

25.2.2、iostream 库

虽然 C++ 早在上世纪 80 年代初就出现了，但是这个语言直到上世纪 90 年代中期才得到规范化。在没有标准化之前，C++ 的头文件都有一个 .h 的后缀名。标准化委员会做了两个重要的更改来标准化头文件：

后缀名 .h 被移出，例如 <iostream.h> 编程 <iostream>

定义在这些头文件中的类型和变量被定义在 `std` 名字空间中，而不是定义在全局空间，例如 `std::cout`

在许多的编译器中，新标准的编译器类型并不兼容标准前的编译器类型。许多项目不得不花几周甚至几个月的时间来将使用标准前的类型移植到新标准类型。在将CORBA从一个CORBA产品或操作系统移植到另外一个产品或操作系统的时候，出现这种令人头疼的移植性问题是非常常见的。CORBA工具 [\[McH\]](#) 的C++ CORBA应用程序的移植性章节中深度的讨论这个问题，并解释了如何通过在工作开始之前花上几个小时就可以让你的代码在标准编译器或标准前编译中都能被正确编译。那些文档中的建议可以惊人地减少你移植应用程序的时间。

25.2.3、C++应用程序中的同步

C++语言并没有定义标准的同步 API。而操作系统的同步 API 在不同的操作系统中非常的不同。许多的 unix 都支持 POSIX 线程标准，但是有一些 unix 仍然可以使用私有的同步。因为这样，在不同操作系统间，移植多线程程序，如果可能代码中使用了原平台的私有同步 API 的话，那么移植工作将包含巨大的繁琐的工作。

许多公司，试图编写他们自己的围绕底层操作系统的同步 API 可移植封装器。然而，这样做充满了困难。例如，在 Windows 上正确地或高效地模拟 POSIX 的条件变量就是非常困难的一个工作。

许多 CORBA 产品能在很多的操作系统上工作，而且这些 CORBA 产品提供了他们自己的同步 API 的封装器。使用这些库就提供跨不同操作系统的能力，但是却使得从一个 CORBA 产品移植到另外一个 CORBA 产品非常的困难。

通用同步策略 (GSP) 类库 [\[McH, Ch. 7\]](#) 提供了一个和 CORBA 产品无关的跨平台的同步支持。GSP 提供了额外优点就是 GSP 并不是去试图模仿操作系统提供

的底层API，而是提供一个用于能简单化编写多线程的高层API。

第26章、 其他CORBA资源

- [书和文章](#)
- [CORBA工具包 CORBA Utility Package](#)
- [Internet资源](#)
- [咨询和培训课程](#)

26.1 书和文章

大多数的CORBA产品虽然都提供了手册，正如你想到的，这些文档的质量也是千差万别。通常你从免费CORBA软件那里获得的文档要比从商业CORBA软件那里获得的文档要少。无论你选择什么CORBA产品，你都会希望有额外书和文档。一个好的方式是通过www.amazon.com和搜索引擎来帮助你浏览CORBA的相关书籍。读者的点评将帮助你选择好书。本书作者最喜欢的CORBA书籍罗列如下：

- *Pure CORBA* [\[Bo101\]](#) 的目标读者是CORBA的初学者。它通过给读者讲述概念和提供了许多C++和Java有用的代码的例子。以C++或Java方式提供例子意味着这本书对使用C++或Java的开发者有关。此外，提供双语言例子的另外一个好处。当人们通过某种特定语言学习CORBA的时候，人们经常很难分辨那些是CORBA的普遍概念，那些是CORBA和特定语言相关的概念。*Pure CORBA*通过两种语言讲解CORBA的方法帮助读者通用CORBA概念和语言相关CORBA问题。
- *Advance CORBA Programing with C++* [\[HV99\]](#) 不是一本介绍CORBA的书，而是一本帮助已经熟悉CORBA的读者提高他们技能的优秀书籍。虽然

它的例子全部都是以C++代码写成，但是这本书对于使用其他语言的CORBA开发人员也同样重要。

- *IIOP Complete* [RHK99]，如果你需要了解GIOP或IIOP的底层细节，那么这本书就非常重要。这本书提供了对这些概念非常清楚的解释。虽然这本书有一点过时，它讨论的GIOP和IIOP版本是 1.0 和 1.1 版本，而现在的版本是 1.3。然而，虽然在不同的版本的协议中有一些细节上的变化，其基础概念仍然一样。因此，如果你希望熟悉GIOP1.2 或 1.3 的一个好方法，就是阅读此书，并从OMG的网站上下载最新的GIOP规范。

Douglas Schmidt，他是 TAO 项目的领导人，Steve Vinoski 是项目 Orbix 的领导人，他们都发布了很多关于 CORBA 非常有趣的文章。你可以从他们的网站上下到这些文章的电子版。

www.ionacom/hyplan/vinoski/

www.cs.wustl.edu/~schmidt/

26.2 CORBA工具包 CORBA Utility Package

CORBA 工具包是一个使得 CORBA 开发简化的带文档的软件包。这个软件包你可以从下面的连接中免费下载

www.ionacom/devcenter/corba/utilities.htm (译者注：此链接已经

失效，新的连接地址是：

<http://www.ciaranmchale.com/corba-utilities/>)

这个包对 C++和 Java 都有效，并且对 Orbix, Orbacus, TAO 和 omniORB 是开盒即用的。因为这些包中关注了很多移植性的问题，所以它非常容易用于其他的 CORBA 产品，甚至如果你不决定在你项目使用这个包，但是这个包中文档是非常值得一读的，因为里面提供了许多非常有用的建议。

26.3 Internet资源

OMG的网站(<http://www.omg.org>和 <http://www.corba.org>) 提供很多免费的CORBA信息。你可以从这个网站上免费的下载CORBA相关的规范。这个网站同时提供了其他Internet资源的连接。

下面的一些新闻组在讨论 CORBA

```
comp.object.corba  
comp.lang.java.corba
```

这些新闻组的贡献者(包括了许多 CORBA 厂家的员工)非常乐意提供问题的解答和有用的建议。一些厂家还有自己的新闻组和自己的邮件列表支持其自己的产品。你应该以 2.7 讨论的方式, 向你的 CORBA 厂家要求更多的细节, 你也可以在 OMG 的网站和 CORBA 厂家的网站上找到很多 CORBA 的成功案例

26.4 咨询和培训课程

一些 CORBA 厂家, 同时也是些独立公司, 提供了对 CORBA 的培训和咨询。你可以联系这些公司, 并查看他们提供了那些服务。你也可以通过搜索引擎查找提供 CORBA 培训和咨询的公司。

- 当你开始一个基于 CORBA 的项目时, 有一个咨询顾问加入你项目, 并给你的项目的基础体系架构做一个“体检”。有经验的咨询顾问经常能支持你架构中的不足之处, 如果体系中存在不足之处, 将会对以后的项目中的性能和扩展性造成重大影响。
- 后来, 在做项目中初始化编码工作时, 同一个咨询顾问能返回项目组并提醒开发者采用好的编程范式以避免他们犯下通常的编码错误。
- 咨询顾问能在每个月的最后几天对项目进程做“体检”。以这种方式, 项目中任何好的 CORBA 编程实践将会被指出, 因此这样避免了项目以后出现重大的问题。

许多公司经常犯下将开发人员送去参加培训课程学习道很久以后才会用到新技能。当真正需要用到这些技能的时候，开发人员已经忘记了大部分。如果能做到，在项目开始之初就开始相关的 CORBA 的培训会收到更好的效果。这样开发人员就能在项目中实践他们学到的新技能。同时，因为项目和课程同时开始，开发人员就可以问他们的授课者如何在工作中使用他们所学的技能。

一个培训课程的授课老师如果是一个已经熟悉你项目的咨询顾问将变得更有效率。在这种情况下，授课老师就能结合特定项目指出相关的 CORBA 概念和适合的编程范式。

本书的大部分信息来自于 INOA 公司提供的培训材料。如果你觉得本书提供了实用的 CORBA 概念的概述，那么你可能会考虑加入 INOA 的培训课程，获得相关的开发 CORBA 应用程序的技能。

第27章、 参考

[Bla99]

Bob Blakely. *CORBA Security: An Introduction to Safe Computing With Objects*. Addison-Wesley, 1999. ISBN 0201325659.

[Bol01]

Fintan Bolton. *Pure CORBA*. Sams, 2001. 921 pages. ISBN 0672318121.

[BVD01]

Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA, third edition*. Wiley Computer Publishing, 2001. 710 pages. ISBN 0471376817.

[HV99]

Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999. 1120 pages.

[Koc00]

Richard Koch. *The 80/20 Principle: The Secret of Achieving More With Less*. Nicholas Breealey Publishing Ltd., May 2000. ISBN: 187881680. 312 pages.

[KPS02]

Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2002. ISBN 0130460192.

[MBB00]

H. X. Mel, Doris Burnett, and Doris M. Baker. *Cryptography Decrypted*. Addison Wesley, 2000. 256 pages. ISBN 0201616475.

[McH]

Ciaran McHale. CORBA Utilities. Available at www.CiaranMcHale.com/download/.

[OMGa]

OMG. Common Security Interoperability (CSIV2). Available for download from www.omg.org both as a stand-alone document and as a chapter in the CORBA Specification.

[OMGb]

OMG. CORBA Security Specification (version 1.8). Available for download from www.omg.org.

[RHK99]

William Ruh, Thomas Herron, and Paul Klinker. *IIOP Complete: Middleware Interoperability and Distributed Object Standards*. Addison-Wesley, 1999. 288 pages. ISBN 0201379252.

[Sch95]

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons Inc,

1995. ISBN 0471128457.

[SGR99]

Dirk Slama, Jason Garbis, and Perry Russell. *Enterprise CORBA*. Prentice Hall PTR, 1999. 366 pages.

[SV99a]

Douglas C. Schmidt and Steve Vinoski. Programming Asynchronous Method Invocations with CORBA Messasing (Object Connections, column 16). *SIGS C++ Report*, 11, February 1999. Available from Steve Vinoski's web page at www.iona.com/hyplan/vinoski/ or from Doug Schmidt's web page at www.cs.wustl.edu/~schmidt/.

[SV99b]

Douglas C. Schmidt and Steve Vinoski. Time-Independent Invocation Interoperable Routing (Object Connections, column 16). *SIGS C++ Report*, 11, April 1999. Available from Steve Vinoski's web page at www.iona.com/hyplan/vinoski/ or from Doug Schmidt's web page at www.cs.wustl.edu/~schmidt/.