# 第2章 CORBA回顾

本书不打算再给出一个对CORBA体系结构的介绍。本书假定读者对CORBA体系结构概念有基本的理解，最好还有使用ORB的实际经验。在这里需要做的是回顾CORBA，并探讨CORBA规范的抽象世界与ORB实现和基于这些ORB应用程序的现实世界之间的差距。通过识别和定义关键元素，可为下一步的讨论设立共同的基础。

首先，我们想回顾CORBA体系结构并提及CORBA激发和CORBA对象的生命周期。处理CORBA激发可能会涉及ORB与应用程序间的复杂交互，我们也想研究在激发生命周期间可能发生的不同事件。当然，CORBA激发的目标总是一个CORBA对象。这意味着激发与对象的生命周期之间有着密切的关系。

在本书的余下部分中读者可以看到，许多设计和实现的决策会受到CORBA激发和对象的生命周期的强烈影响。所以，本章定义的概念都是下面讨论的基础。为反映CORBA体系结构的不断进展，本书从演变的角度来看待激发和对象的生命周期。

## 2.1 CORBA的演变

这里简短地讨论一下CORBA体系结构的演变。问题在于有很多细微的或重大的修正，有时会引起混淆。我们对于研究所有不同的细微修正间的差异不感兴趣，相反，本章打算设立以演变的角度来讨论CORBA体系结构的范围。

对象管理组最早出版的重要文档之一是对象管理体系结构——OMA。OMA定义了分布式对象系统的基本成分。读者都看过这样的图形，ORB作为软件总线连接如公共服务、公共工具等高层次组件和应用程序级对象，CORBA只是这个图形的一部分。但没有了ORB，就不能构建服务、工具和应用程序对象。因此，最初的焦点就放在ORB体系结构本身。这就使得CORBA这个术语变得流行起来，一下子什么都是CORBA：CORBA服务、CORBA工具等。

CORBA体系结构的第一代主要集中在为分布式对象计算定义一个基础。使这个基础独立于技术和编程语言的一个关键成分是接口定义语言——IDL的定义。最初是为C编程语言定义IDL映射。第一代的CORBA伴随有命名、事件和生命周期等一些基本服务。

CORBA体系结构的第二代加入了ORB互操作性，它基于与ORB独立的互操作协议：通用Inter-ORB协议（General Inter-ORB Protocol, GIOP），以及基于TCP/IP的特殊协议，因特网Inter-ORB协议（Internet Inter-ORB Protocol, IIOP）。还增加了另外一些语言的映射，如C++，以及定义了大量的附加服务，如分布式事务处理服务、安全性、对象关系、并发性、查询处理、服务交易和时间。这些服务中的很多对基本ORB增加了重要的价值，使得开发者能够很好地通过ORB来利用高级的系统功能。但是，一些服务也会带来问题：它们中有一些很学术化，不太适用于现实世界；其他的则太复杂，很难实现和使用。第3章中会讨论不同CORBA服务规范的质量。

## 2.2 BOA代和POA代

正如前面所说，我们对于详细讨论不同规模的CORBA体系结构版本不感兴趣。为帮助组

织关于激发和对象生命周期的讨论，我们要识别两代不同的CORBA：BOA（基本对象适配器）代和POA（可移植对象适配器）代。这样区分将允许我们定义在核心ORB体系结构演变过程中两个重要的里程碑。

### 2.2.1 BOA代

CORBA体系结构的BOA代定义对象适配器是“应用程序用来访问ORB函数的主要接口……它包括……生成对象引用的接口，含有一个或多个程序的注册实现和激活实现所需的接口。”[⊖]

要注意，当谈论到注册和激活实现时，BOA着重于CORBA服务器实现，而不是CORBA对象实现。BOA规范的一个主要不足就是它面对这样的问题时的不确定性。对于生成的服务器端框架缺少命名约定也是一个例证，这使得以遵循CORBA的方式来关联框架和对象实现变得不可能。

下面使用IONA Technologies的Orbix 2.x的CORBA实现来作为ORB的BOA代的基准。

### 2.2.2 POA代

可移植对象适配器试图一方面克服最初BOA规范的不确定性，另一方面又为对象生命周期管理提供一整套的高级服务。

对POA本身的深入讨论可在Schmidt和Vinoski的书[⊖]中找到，其中还提供了如下定义：“对象适配器是CORBA组件，负责使CORBA的对象概念能适应编程语言的伺服对象概念。”这个定义强调对实现的注册和激活从过程级转移到了对象级。在本章 2.4节中将会看到CORBA对象和CORBA对象实现之间的区别。

对于对象生命周期，POA本身并不是在CORBA体系结构演变过程中唯一令人感兴趣的变化。其他令人感兴趣的新特征包括拦截器，它允许对激发生命周期，以及服务上下文(service context)的监控。服务上下文支持以标准的CORBA激发来发送额外的信息。尽管最后这两个特征和对象适配器的概念无关，本书仍然把它们纳入“POA代”定义中。这意味着本书对BOA和POA两代的观点节略了CORBA的详尽版本的历史，使用户能够侧重于概念。

## 2.3 激发生命周期

在此要描述在CORBA激发生命周期中的不同步骤。本书叙述在激发生命周期中非常重要的几个不同方面的条目，然后利用这个条目对ORB实现的BOA代和POA代分析激发生命周期。

### 2.3.1 CORBA请求

CORBA体系结构的主要目的是定义一个描述客户机如何能向远程的对象实现发送请求的框架，并潜在地从对象处得到回应。对象接口用与编程语言无关的接口定义语言描述。基本上有两种不同的方式来让客户机和对象实现发送和接收请求：静态方法和动态方法。静态方法要求所有的IDL接口在编译时已知，这样IDL编译器就能生成桩和框架代码，这些都必须链接到实现。动态方法使用户在编译时不用了解不同的IDL接口就能实现用程序来处理任何类型

---

⊖ 对象管理组公共对象请求代理：体系结构和规范，修正本2.1，1997年8月，8-1页

⊖ Schmidt, D.C. and Vinoski.S.,“Object Adapters: Concepts and Terminology”.SIGS C++ Report. Vol 9, No11. November/December 1997。

的请求。处理请求的动态方式要求在客户端使用动态激发接口（Dynamic Invocation Interface, DII）以及在服务器端使用动态框架接口（Dynamic Skeleton Interface, DSI）。DII和DSI通常用来构建如桥接器等一般的系统级组件。

对于正常的应用程序，静态桩和框架的使用更加普遍。静态方法的好处是使用户能很好地使用CORBA对象，就好像它们是编程语言中的普通元素一样。在客户端，这通过使用代理对象来完成。代理是远程目标对象的本地代表。代理包含足够的信息来向远程目标对象发送请求，封装网络地址、端口号等细节。代理对象通过使用客户机编程语言的标准类型，提供了以类型安全的方式来访问目标对象的方法。如果客户机想使用实现了Stock IDL接口的对象，桩代码就会向它提供等价的用特定编程语言编写的Stock接口，例如C++的Stock类。如果客户机想向远程Stock对象实现发送消息，它只需简单地激发本地代理的一个方法。桩代码，即生成的代理实现，负责打包（marshal）请求的参数，这样客户机的ORB运行时模块就能向目标服务器发送消息。服务器的ORB运行时模块读取从网络传来的消息，并把消息传给生成的框架代码，使得框架代码解包请求的参数，这样它就能把这些参数传递到目标对象的实现。生成的框架把请求作为服务器端的正常方法调用来传递，使得客户机和服务器都像对待普通的编程语言对象一样来对待CORBA对象。请求的回答能以同样的方式发送回客户机。

因为静态接口比动态接口使用得更普遍，这里着重讲述静态接口。图 2-1总结了使用静态接口的CORBA远程激发的原理。

请求的CORBA模型假定每个请求都有一个目标、一个操作和一系列参数。目标标识目标对象，操作描述所激发操作的名称，而参数则是需要传递的数据。一个请求必须提供一种激发功能。激发功能可以有不同的语义，例如阻塞和非阻塞的调用，或是单向的语义。在本章的余下部分假定请求可以获取预期的回应，而且调用是阻塞的（即标准的双向远程激发）。

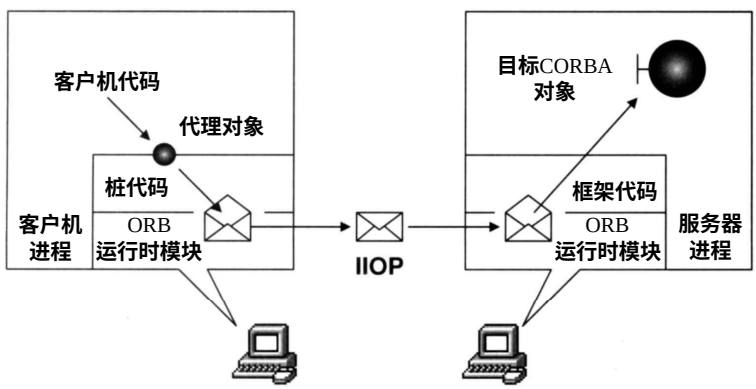


图2-1 使用静态接口的CORBA远程激发的原理

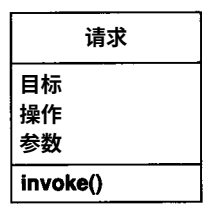


图2-2 CORBA请求模型

### 2.3.2 激发生命周期：评估标准

下面我们定义在讨论CORBA激发生命周期时受关注的评估标准的条目。这些标准可以应用到BOA代ORB，也可以应用到POA代ORB。接着会探讨激发监控、上下文信息传递、对象激活、服务器端的线程分配、配置以及请求的重定向。

- 激发监控　在激发的生命周期中，有很多要监控的点是服务库或应用程序在允许ORB继续处理激发之前或甚至是在请求ORB终止激发之前想要进入的。通常，想要监控的点在客户端是输出请求和输入回答，而在服务器端则是输入请求和输出回答。激发可以在两个不同的层次上进行监控：请求对象和消息缓冲区。请求对象让用户得到访问请求属性的机会，例如目标、操作名称和参数。消息缓冲区含有请求的打好包后的版本，即所有请求参数是平展的。监控激发有很多不同的原因，例如记日志、跟踪、加密或访问控制。另外，监控点有时候可用来作为钩子，以触发其他重要的激发生命周期事件，例如服务器端的线程分配。
- 请求上下文　在很多场合中，用户想与每个请求一起传递额外的信息。例如，一个事务服务可能想和请求一起传送关于事务的信息，这样事务服务就能保证在服务器端代表该特定事务的所有数据都已被更新。或者是安全服务实现想和每个请求一起传送安全标志，以便接收者用来实现访问控制逻辑。我们想以通用的形式来提供这些服务，即要独立于应用程序特定的IDL。如果应用程序不得不在传送安全标志时向每个IDL操作加进额外的参数，那么这就是一个糟糕的主意。传送隐含的请求上下文的机制是用来实现像事务和安全性这样的服务，以使服务以独立于应用程序级上使用的IDL的方式来进行。
- 对象激活　当请求到达服务器端时，服务器的ORB运行时模块试图定位能处理该请求的对象实现。如果找不到这样的对象，ORB通常生成一个异常，以向客户机指出该对象不存在于服务器端。但是，用户希望ORB能提供一种机制，以让服务器实现用命令来激活对象。这意味着如果ORB不能定位目标对象，它就会请求应用程序向它提供一个能用来处理请求的对象。对象激活的这一方面实际上和对象生命周期密切相关，所以对象激活的更详细处理会推迟到后面的对CORBA对象生命周期的讨论中。
- 线程分配　在服务器端，我们希望能够使用线程来并发处理多个请求。问题是在请求生命周期中哪一点能分配线程。可以想到很多不同的线程分配策略，例如每个对象的线程，每个客户机的线程，或每个请求的线程。一个重要的问题是，到底是ORB还是应用程序来负责分配线程和实现某种线程策略。
- 配置　配置的思想是把客户机和服务器逻辑链接到相同的可执行程序，并由客户机激发本地的对象实现。出于性能的考虑，配置的调用明显不能在网络上发送。一个值得注意的问题是配置的调用能否绕过激发的监控点。
- 定制的代理　在很多场合，扩展代理的标准功能是有意义的，即使用代理来处理打包和解包(unmarshal)以及这两者之外的请求。例如，一定制的代理可用来实现客户端的负载平衡机制。定制代理的另一个例子可以是存储对象属性的代理。在使用高级的代码生成工具时，定制的代理实现就特别有意义。在第19章“自动化工程过程”中会有代码生成的更详细讨论。
- 客户机重定向　最后，我们要求客户机具有当对象重定位时重定向请求的能力。在这里一个重要的方面是客户机的透明性——即如果由于重定向而必须重新发送请求时，激发

代理的客户机是否应被通知。

### 2.3.3 激发生命周期：评估BOA代

首先，我们根据上面所描述的激发生命周期不同的评估标准来考察 CORBA ORB的BOA代。问题在于 BOA规范十分含糊，而且对大多数方面没有进行论述。因此要用 IONA Technologies Orbix的2.x代来作为BOA代的参照实现。由于BOA规范对于想讨论的大多数方面都很模糊，其他ORB可能就会提供这些方面的其他实现，甚至根本就没有实现。但是，本书不打算提供不同的ORB实现如何处理BOA规范的模糊性的讨论，只是要给出 BOA代ORB如何实现激发生命周期的一个例子。

1. 激发监控

BOA规范没有论述激发监控的问题。 Orbix ORB提供了能用于监控激发的两个钩子。

- 过滤器 应用程序可以实现一个过滤器对象，并把它注册到 ORB运行时模块。然后ORB运行时模块将在激发生命周期中的某些点激发注册的过滤器对象，给过滤器一个机会来以应用程序指定的方式对事件作出反应。 Orbix会向过滤器传递一请求对象，代表当前的请求。但是请求参数不可访问。
- 请求转换器 类似过滤器，应用程序可以把请求转换器注册到 ORB运行时模块。对于请求和回答，ORB都会让进出消息的缓冲区对应用程序可用。

2. 请求上下文

请求上下文和对象适配器并不是真正地紧密相关，它更多是和 GIOP/IIOP规范相关。在对BOA代的讨论中读者可以看到，这个问题实际是在 CORBA较后版本中的协议层讨论的。这个问题的初始解决方法是引入附加（或隐含）参数的概念。如果一个 IDL操作定义为含有a1到an的参数，那么上下文可作为附加的参数 an+1传送。例如，在CORBA OTS 1.0规范中就用到了这个方法。

Orbix ORB允许应用程序使用类似于“ 搭载 ”（piggy backing）的方式发送隐含的请求上下文。Orbix过滤器如上所述，可以用来向请求加入附加参数，也可以从请求中取出附加参数。问题在于应用程序无法决定一个请求是否含有上下文。因此，这个方法要求在客户机和服务器的程序之间有很多约束和协调。

3. 对象激活

虽然CORBA的BOA代对对象和服务器激活作出了明确的区分，但它并没有清楚地定义后期绑定如何实现，即对象如何通过命令来激活。

Orbix ORB提供了“ 装载器 ”机制：应用程序可以实现一装载器，并把它的一个实例注册到服务器ORB运行时模块。如果ORB不能定位请求的目标对象，它会激发装载器上的 load()方法，给应用程序一个机会来创建和返回对象实现，该实现再被 ORB用来处理请求。

4. 线程分配

同样，CORBA规范的BOA代没有描述服务器端ORB运行时模块如何处理线程的分配。

Orbix ORB允许应用程序安装一个称为线程过滤器的特殊种类过滤器。这种线程过滤器的实现可以分配线程，并告诉 ORB使用该特定线程继续处理请求。由于线程分配完全是实现的职责，所以任何种类的线程策略都能实现。在第 14章“ 服务器资源管理 ”中会讨论不同的可能线程策略。

5. 配置

CORBA体系结构设计用来以最透明的方式支持对 CORBA对象的远程和配置的激发。例如，IDL到C++映射的内存管理准则的设计方式是让远程和配置场合在客户机和服务器的角度看来是相同的。

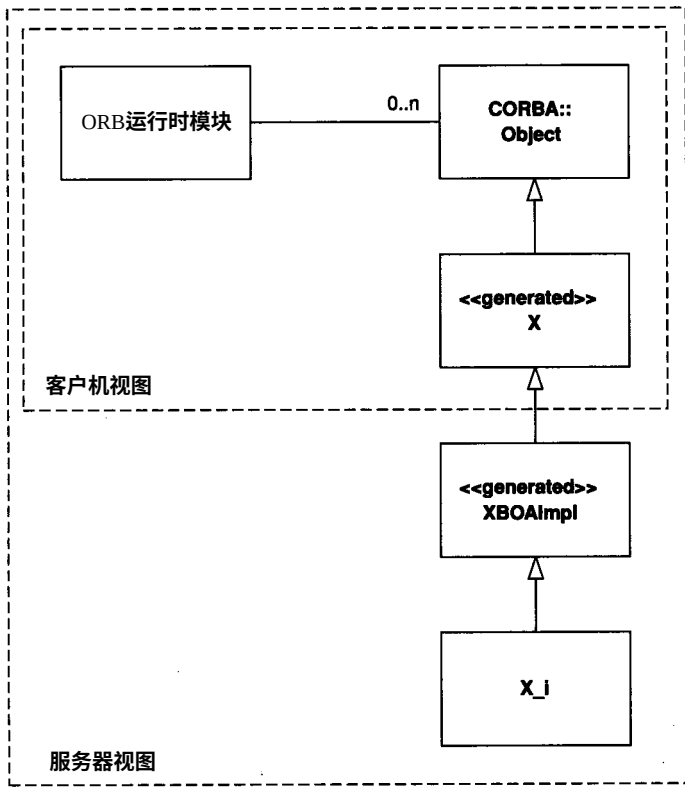


图2-3 用于接口X的Orbix桩/框架类层次

如在绪论中所说的那样，值得注意的问题是扩展哪一方面以使这种透明性能保证激发生命周期的其他方面，如激发监控。同样， BOA规范在这里也很模糊。 Orbix使用特定的桩/框架层次来处理配置。

图2-3描述了以客户机和服务器角度所看到的 Orbix C++桩/框架层次。客户机看到的总是接口类，在图中是“ X ”，而不管它是代理还是真的实现。这意味着在配置的场合中，客户机直接激发对象实现。一方面，这是最有效的实现，因为它意味着配置的调用是一单独的虚拟功能调用。另一方面，这意味着 ORB不能截住激发，所以没有人能调用任何的监控点。

那么，这种方法的缺陷有多大呢？对于请求转换器，这不成问题，因为它们通常用于如加密之类的事情，这些对于配置调用都是不需要的。对于请求级过滤器，这可能就是个问题。下面看一下对象事务服务的实现，它使用过滤器把事务上下文信息从客户机透明地传送到服务器（关于OTS的更详细解释参看第 12章“ 分布式事务处理 ”）。客户机的线程和事务上下文相关联。这样，此处看上去就没有问题，因为对象使用同一个线程来执行调用。任何初始化调用都提前到达 DBMS。但在每个对象基础上使用服务器端过滤器来实现访问控制机制的安全性服务又如何呢？这个访问控制机制在配置场合中本来是可以绕过的。根据安全性需求，

这可能是可接受的。如果不能，则要为配置的调用找出一种特殊的解决方案。

6. 定制代理

当解包一个对象引用时，ORB创建一代理对象以使客户机能使用该对象引用。为支持定制代理，ORB必须提供一个钩子以使应用程序能向ORB提供定制代理的实例。CORBA规范的BOA代没有定义这样的钩子。

许多ORB实现专有的机制来支持定制代理。例如，VisiBroker有智能桩的概念；Orbix把它们称为智能代理，Orbix ORB使用代理工厂来创建代理。应用程序可以安装特定的代理工厂，它们能根据命令向ORB提供定制代理。

7. 客户机重定向

BOA代的GIOP规范定义了一个特殊的回答状态类型，称为LOCATION_FORWARD。一个转发定位的回答可包含一个新的对象引用，这个引用又可被客户机用来透明地向新目标对象重新发送请求（在请求之后）。不幸的是，BOA代的GIOP规范没有定义公共的API，这样转发定位机制只能被ORB内部使用，或是要通过专有的API使用。

8. 小结

图2-4描述了BOA代ORB中的激发生命周期，在这里用的是Orbix ORB。在客户端，应用程序激发一个代理。使用客户端过滤器，输出的请求可作为请求对象被截住。作为选择，消息缓冲区可用请求转换器来访问。在服务器端，到来的消息同样由服务器端的请求转换器来访问。Orbix现在首先要完成的是在内部表中定位对象。如果目标对象不能定位，Orbix会尝试已注册的装载器。如果目标对象能定位，下一步就是分配线程，用线程来继续处理请求。这可以是ORB使用的缺省线程，或是由应用程序在线程过滤器中提供的线程。接着，在调用目标对象之前，ORB先调用请求中的过滤器点。当目标对象完成请求的处理后，回答被发送回客户机。在请求的生命周期中可能会有异常抛出。在这种情况下，ORB不会继续处理激发。但是Orbix提供了故障过滤器点，它们在有异常的场合中会被调用。

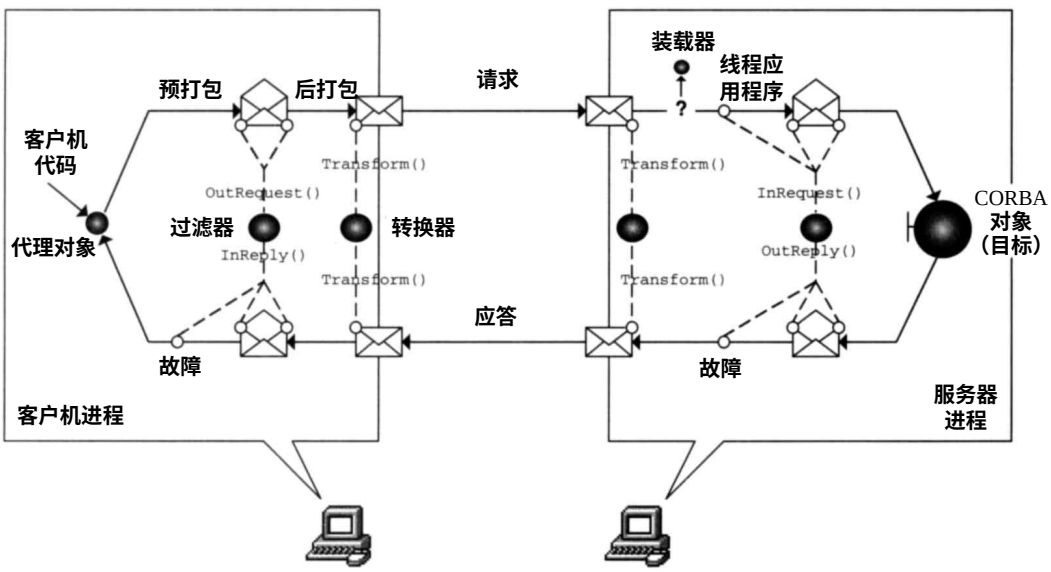


图2-4 Orbix激发生命周期概貌

### 2.3.4 激发生命周期：评估POA代

从BOA的角度已经讨论过激发生命周期的问题，现在我们想以 POA的角度来探讨相同的问题。POA规范在CORBA 2.2版本中引入。归类在“ POA代 ”这个通用术语下的所有其他成分是在CORBA 2.x规范的某一个版本中引进的。在写本书期间，大多数 ORB供应商已开始发售POA代的ORB，或是宣布了这样做的计划。因为移植性被认为是 POA的主要优点，所以本书的注意力不放在特定的实现上——这里讨论的概念可以应用到所有 POA代CORBA规范的可靠实现中。

1. 激发监控

CORBA 2.2规范引入了拦截器的概念，它可以在激发生命周期中作为钩子使用。拦截器可以在客户端和服务器端用来监控激发生命周期，允许用户监控请求和回答。在客户机和服务器端都可安装多个拦截器，这些拦截器将轮流进行处理。规范定义了两类不同的拦截器。

- 请求级拦截器 ORB调用请求拦截器，并让结构化的请求对它可用。
- 消息级拦截器 ORB调用消息级拦截器，并让目标对象和非结构化消息缓冲区对它可用。

要注意，一些人批评说当前拦截器的结构规定不够具体而且很难实现。 OMG关于可移植拦截器的计划正开始致力于解决这些问题。

2. 请求上下文

CORBA 2.0规范定义了服务上下文的概念。这些服务上下文可以与请求和回答一起发送，象事务或安全性服务这些 CORBA服务都需要上下文信息。一个服务上下文有一个 ID和一些数据，以8位位组的序列形式组织。任何数目的服务上下文都可与请求和回答一起发送。发送器和接收器是非耦合的。接收器可以忽略它不能理解的上下文，也可以处理不包含任何期望上下文的请求。

把服务上下文包括为 CORBA请求的一部分，并定义这些服务上下文如何打包消息，可以解决前面在 BOA代的请求上下文讨论中所描述的搭载方法问题。附加的数据现在也可以以标准和类型安全的方式发送。

3. 对象激活

POA规范明确规定了 ORB如何处理后期绑定，这点它做得很好。有了 POA，一些对象激活的不同措施就成为可能。基于 POA的服务器通常使用一系列不同的 POA实例。这些POA实例的每一个都和一系列的对象实现（即伺服对象）相关联。 POA和一些策略相关联，这些策略控制POA的行为。 POA如何处理对象激活取决于和该特定 POA相关的策略。例如，应用程序可以提供一伺服对象激活器（ Secvant Activator），它在对象故障时由 POA激发。在本章后面有对POA体系结构更详细的讨论。

4. 线程分配

POA规范定义了两种不同的策略来决定 POA如何处理向请求分配线程。第一种策略指出 POA只能使用一单独线程，第二种策略决定 ORB可以控制线程分配策略。显然，这还不够明确，因为应用程序经常需要控制 ORB使用的线程策略类型，就如同 ORB如何处理最大负载、阈值和线程重用等事情。其他未回答的问题包括应用程序如何能控制被 ORB使用的策略，以及应用程序如何实现它本身的线程策略。希望 POA规范的未来修正版会解释这些问题。

5. 配置

CORBA的POA代要求ORB即使在配置的场合下也可支持请求级拦截器。像前面提过的那样，如果桩和框架层次被设计为客户机持有一直接指向所配置实现对象的指针，这是不可能的。这就是为什么CORBA的POA代允许ORB实现拥有分离的，且用于客户端接口类和服务器端对象实现的继承层次的一个原因。这使得 ORB易于实现某种委托机制，在机制中由客户机线程激发某个代理，该代理本身又把该调用委托给的真正实现对象的。这一额外的委托步骤的优点在于ORB在激发目标对象前可先调用拦截器。但这明显有一个小缺点：这个额外的委托步骤（用户在此处理抽象的请求对象）比大多数 BOA代ORB在配置场合中所使用的单独虚拟功能调用的代价更高。基于 POA的ORB类层次如图2-5所示。

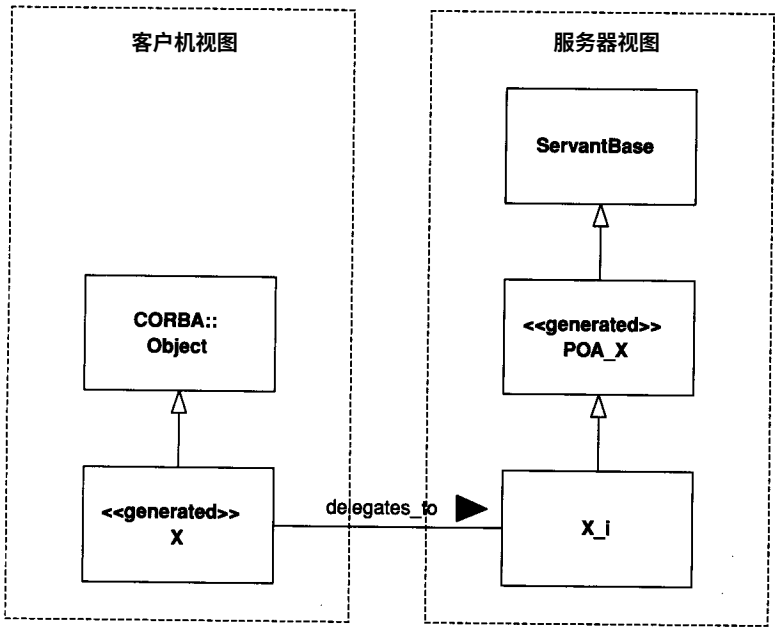


图2-5 用于接口X的POA桩/框架类层次

6. 定制代理

CORBA规范的POA代没有提到ORB如何能为应用程序控制代理的创建提供钩子。这意味着目前还没有标准的方法让ORB供应商以遵循CORBA的方式来实现这一点。

7. 客户机重定向

如同前面讨论过的BOA代的ORB，CORBA规范定义了特殊的回答状态类型LOCATION_FORWARD。它用于把客户机重定向到新的对象实现。 POA规范使用称为ForwardRequest的异常来对这个重定向特征定义一公共的 API，这个异常含有一类型为 Object的单独成员。这个异常可由伺服对象管理器中的应用程序生成，该管理器使用异常机制来把客户机重定向到新的对象实现，例如在不同的服务器中。规范陈述了从此时起 ORB要把所有的请求发送到新对象。新对象的引用作为提出请求的结果返回。

图2-6显示了一合作流程图，描述了客户机重定向的一个例子。在主机 A中的一个客户机向对象发送请求，这个对象假定是位于主机B的服务器中。因为该对象在这个服务器上不存在，所以此请求不能被处理。服务器中伺服对象管理器抛出一含有对象引用的 ForwardRequest异常。这个引用指向主机C中的实现。客户机ORB的职责是把请求透明地重新发向新地点。同样，

以后所有的请求必须都发向主机C的对象。在第15章“负载平衡”和第16章“容错性”中会对这种机制的应用程序进行更详细的探讨。

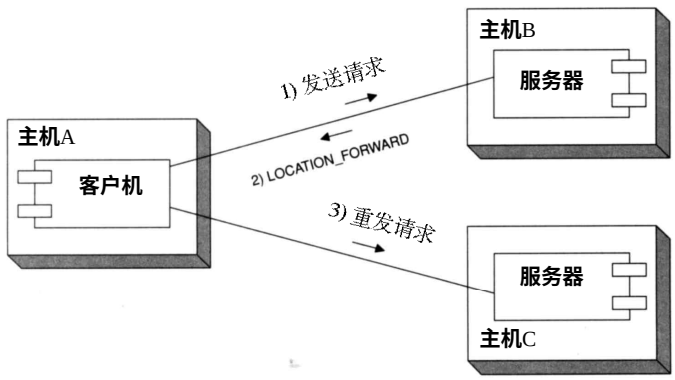


图2-6 LOCATION_FORWARD示例

8. 小结

图2-7显示了POA代ORB的激发生命周期，其中使用了类似于图 2-4中对BOA代ORB描述的图解。同样，客户机激发一个代理。在激发生命周期中第一个值得注意的事件是客户端的请求拦截器：ORB在第一个安装的请求拦截器上调用client_invoke()。注意这个调用是阻塞的，并且只有当目标处理完请求后才能返回。请求拦截器使用 DII函数invoke（）（图中没有显示）来继续请求的生命周期。如果没有其他的请求级拦截器， ORB通过使用send_message()来调用第一个消息拦截器以继续下去，并把消息（或消息的一部分）传递到拦截器。拦截器通过调用send()（图中没有显示）来继续请求的生命周期， send()通常在请求没有完成时就返回。ORB负责把消息发送到目标服务器。

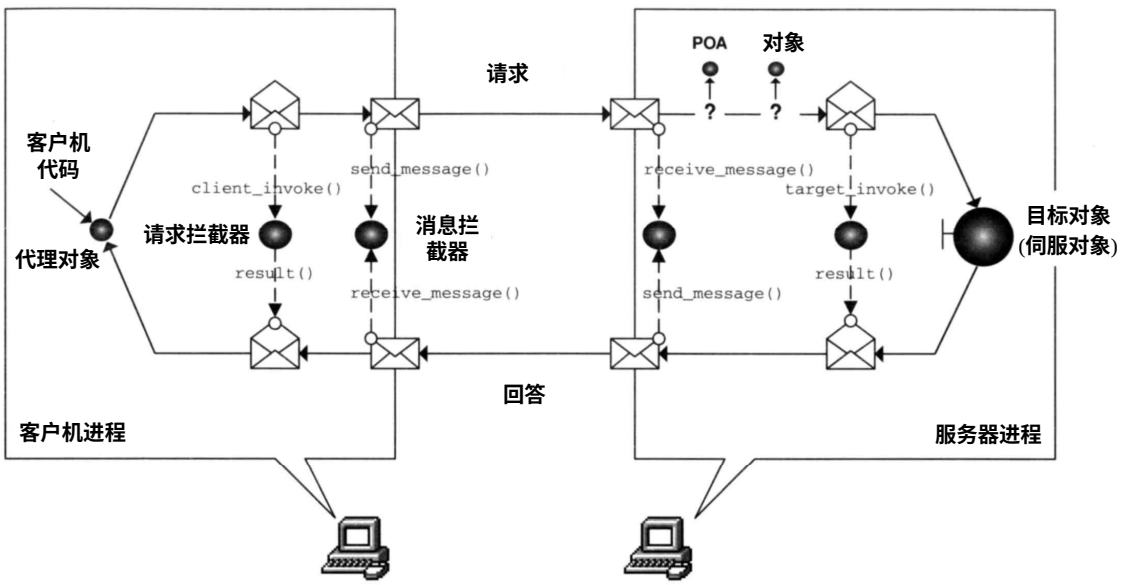


图2-7 具有POA代ORB的激发生命周期

在服务器端，服务器的ORB在运行时从网络读取请求，并通过调用在第一个安装的消息拦截器上的receive_message()开始处理请求。ORB用对象关键词以标识目标必须含有POA的名字，通过POA才能到达该对象。找到正确的POA后，下一步是寻找对象本身。这个工作如何完成取决于为对象的POA定义的策略，本章的后面对此有更详细的描述。如果对象能够定位，ORB通过调用在第一个安装的请求拦截器上的target_invoke()来继续处理请求，拦截器则使用DII函数invoke()来依次继续处理请求，这在客户端中已经讨论过。这里假设只安装了一个请求拦截器，请求最后会被分派到目标对象的实现。

当对象完成处理请求后，消息拦截器的invoke()调用返回，拦截器现在就有机会在返回到ORB运行时模块之前通过激发请求对象上的result()来检验操作的结果。服务器端的最后一件事是ORB对服务器消息拦截器上的send_message()的激发。在客户端，拦截器以类似的方式来处理：首先ORB调用消息拦截器上的receive_message()方法，然后请求拦截器对invoke()的阻塞调用返回，并使这些调用能检验请求的结果。最后，由客户机激发的在代理上的方法返回，向客户机提供结果。

POA规范第一版的问题在于很难决定生命周期中的事件的确切顺序，特别是在服务器端。例如，规范没有明确定义在服务器端的的哪个时段为特定的请求分配线程——很多线程策略要求在请求对象中提供的信息。另外，规范没有清楚规定POA和与请求拦截器有关的对象查找的顺序，所以在流程图中，这个顺序是基于它们的逻辑关系。

## 2.4 对象生命周期

在本节要讨论CORBA对象的生命周期及其实现，从总体上看一下对象生命周期，然后再探讨BOA和POA两代ORB的细节。

本书主要以应用程序而不是ORB的角度来看待对象生命周期，所以要首先看一下应用程序通常要求什么来有效地管理CORBA对象的生命周期。

### 2.4.1 CORBA对象

首先研究CORBA对象的本质：什么是真正的CORBA对象？假设用户有一客户机应用程序，通过和应用服务器中的股票对象的交互来监控股票的价格。股票对象实现了IDL接口Stock，并有一对象关键字“IONAY”。包含股票对象的CORBA服务器以COBOL实现，运行在MVS主机上。股票实现使用VSAM数据库来持久地存储它的状态。

现在，为了实现再工程，金融信息服务的供应者决定把应用程序以C++语言实现在Unix系统上，并使用RDBMS来存储与股票相关的信息。在晚上，关掉主机，并用Unix系统来代替。第二天早上，股票经纪来工作，愉快地使用自己的价格监控系统。对他（和他使用的客户机应用程序）来说，这种整体的改变是完全透明的。

股票对象的新实现提供相同的接口，并有相同的标识，但其实现和持久数据表示的结构有了很大的变化。新对象实现看上去有相同的行为，至少在股票经纪看来是没有区别的。问题在于，股票经纪仍然访问相同的CORBA对象吗？

1. CORBA对象

POA规范定义CORBA对象为具有标识、接口和实现的抽象实体。上面的讨论已经说明了

为什么CORBA对象是抽象的——它和用来实现它的机制并没有直接的结合。

从客户机的角度来看，对象表示为对象引用，对象引用封装了对象接口类型和标识，并包含足够的信息来定位对象的实现。但从服务器的角度来看又怎样呢？

2. 伺服对象

POA规范引入了伺服对象（servant）的概念，使抽象的CORBA对象能和实现该对象功能的具体编程语言实体彻底分离。这样从服务器的角度来看，CORBA对象是作为伺服对象实现的。要记住CORBA是与编程语言独立的体系结构。伺服对象可实现为C++或Java类，也可以实现为一系列的COBOL段或C函数。伺服对象的概念也有助于反映这种与编程语言的独立性。

那么ORB如何为一到来的请求找出正确的伺服对象呢？在CORBA对象关键字和伺服对象之间必然有某种绑定。下面将探讨这种绑定并讨论应用程序如何能控制这种绑定。

3. 对象适配器

CORBA体系结构定义了对象适配器的概念，用来处理应用程序和ORB如何交互来管理伺服对象和CORBA对象生命周期的问题。对象适配器的一个很好的定义来自Schmidt and Vinoski（1997）[⊖]："对象适配器是一CORBA组件，负责把CORBA的对象概念适配为编程语言的伺服对象概念。"前面已讨论过CORBA对象是抽象的，而伺服对象是具体的，那么这个定义就充分表达了这一点。

### 2.4.2 对象生命周期事件

对于CORBA对象，以下两个生命周期事件是很重要的：

- 创建 CORBA对象的生命周期从创建事件开始。CORBA对象通常通过工厂对象创建，即由对象提供操作来创建新对象。回忆一下，在IDL级，CORBA并没有像构造器这样的静态函数概念。
- 删除 CORBA对象的生命周期伴随着删除事件而结束。通常，CORBA对象可在它们的IDL接口中定义某种delete()操作来删除。有时候，可通过其他对象来删除CORBA对象，例如，通过创建这些对象的工厂对象。

由于前面说过CORBA对象实际是由伺服对象来实现的，所以也要看一下伺服对象的生命周期。本书为伺服对象定义了下面两个生命周期事件：

- 激活 激活事件使伺服对象能处理为特定CORBA对象[⊜]而进入的请求。这就提示了抽象的CORBA对象和具体伺服对象之间的绑定是通过对象ID来创建的。通常（但不是必须），激活事件也要包括伺服对象的实例化。
- 冻结 冻结事件的结果是使伺服对象从CORBA对象中取消绑定。通常（但不是必须），冻结事件也包括伺服对象的销毁。

要注意，CORBA对象的创建与删除和它们的激活和冻结之间有根本的区别。对很多CORBA对象的类型，CORBA对象生命周期会包括多个伺服对象的激活和冻结，如图2-8所示。

---

⊖ Schmidt, D.C. and Vinoski, S.,"Object Adapters: Concepts and Terminology".SIGS C++ Report. Vol 9, No11. November/December 1997.由SIGS Publications授权重印。

⊜ 要注意POA规范区分对象激活和对象化身之间的差别。我们觉得这个差别对本书讨论的层次是过于精细的。关于POA的优秀论述可在"Object Adapters: Concepts and Terminology", Schmidt and Vinoski, 1997中找到。

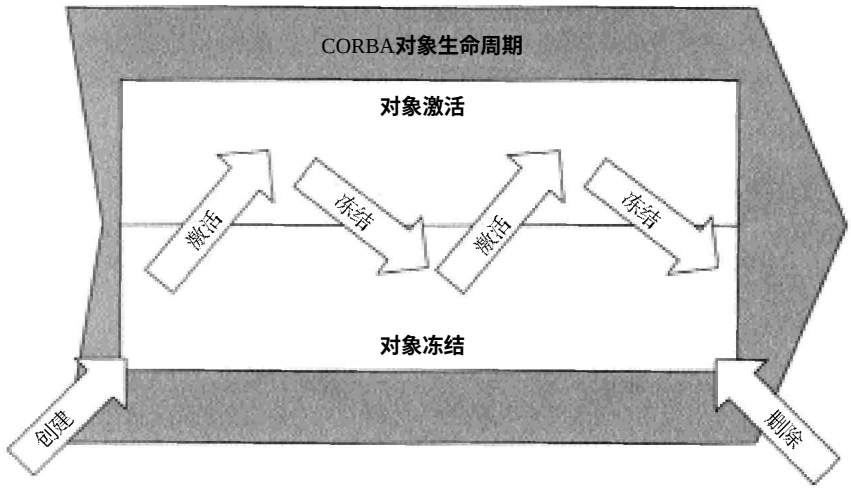


图2-8 CORBA对象的生命周期

### 2.4.3 早期绑定和后期绑定

在伺服对象生命周期事件的定义中，已经说过伺服对象通过 CORBA对象的ID依附于CORBA对象。下面会为BOA和POA代的ORB而分别讨论CORBA对象ID的概念，然而，伺服对象和CORBA对象之间的绑定的思想还是很重要的。对象适配器必须提供必需的接口，使我们能执行这种绑定。问题在于，是什么触发了真正的绑定：是某种标准应用程序逻辑还是来到的请求？下面首先介绍早期绑定，然后讨论后期绑定，或者说是通过命令的绑定。

1. 早期绑定

早期绑定通常描述的是某种标准应用程序逻辑通过对象适配器来执行绑定的情况。例如，一工厂对象可为创建新对象提供操作。这个操作的实现可创建一新的 CORBA对象，并在返回新创建对象的引用之前激活一个伺服对象。因为 CORBA对象现在是激活的 —— 即存在对伺服对象的绑定 —— 客户机现在可使用引用和激发新创建的 CORBA对象。

2. 后期绑定

后期绑定意指绑定只能通过命令创建，即是在对象故障发生的情况下。如果在目标服务器的ORB运行时模块中所请求的目标对象和伺服对象之间不存在绑定，就会发生对象故障，即ORB不能为目标对象找出实现。

在对象故障的情况下， ORB可以要求应用程序提供一个伺服对象，以把这个伺服对象和目标对象绑定。如果应用程序不能完成这个请求， ORB就会引发异常以通知客户机被请求对象不存在。

创建后期绑定有两个选择：绑定只在请求期间被创建，或是绑定可以比请求生存得更久。在第一种情况中，下一个相同对象的请求会导致另一个对象故障，而第二种情况中绑定仍然存在（同时不需要任何动作来冻结伺服对象）。

图2-9显示了早期绑定和后期绑定的关系：基本上，早期绑定意味着应用程序采取主动。后期绑定意味着对象适配器请求应用程序提供一个伺服对象，以使绑定可以创建。这通常是通过伺服对象管理器来完成。伺服对象管理器是一个本地回调对象，由应用程序实现并注册

到ORB，这样ORB在对象故障的情况下就可以激发它。

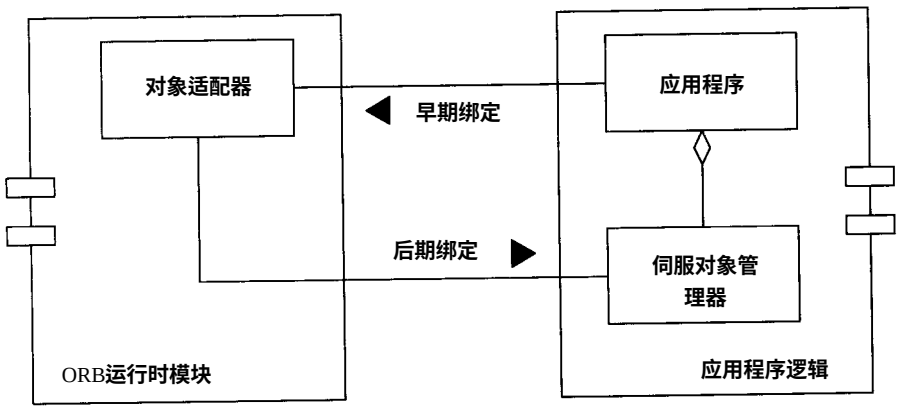


图2-9 早期绑定和后期绑定

### 2.4.4 CORBA对象实现的分类

已经讨论过伺服对象和CORBA对象的不同生命周期事件以及伺服对象和CORBA对象之间绑定创建的不同方式，现在从应用程序的角度来研究CORBA对象实现的分类。这会帮助读者在下面的对BOA代和POA代ORB对象生命周期的讨论中能把重点放在应用问题上。

1. 伺服对象和状态

第一个分类和伺服对象的状态相关。基本上可以划分两类完全不同的伺服对象：无状态伺服对象和有状态伺服对象。在这里简单介绍一下它们，因为在下面的讨论中需要弄清这两者间的区别。对状态管理的深入讨论在第10章“数据库集成”中可找到。

2. 无状态伺服对象

无状态伺服对象与内存中任一特定应用程序的状态没有关联。这并不是指由伺服对象实现的CORBA对象必须是无状态的。例如，CORBA对象的状态可驻留在数据库中，由伺服对象访问以执行请求。在这种情况中，伺服对象像一个瞬态的胶囊（capsule），它通过执行SQL语句而把到来的CORBA请求委托给数据库服务器。

3. 有状态伺服对象

有状态伺服对象与某个特定应用程序的状态有关联，可由伺服对象实现的IDL操作来访问这些状态。

### 2.4.5 CORBA对象实现和内存管理

根据伺服对象的状态把它们进行分类后，现在基于内存管理来定义CORBA对象的分类。显然，这个讨论和伺服对象的生命周期密切相关。这里侧重于分类，所以只是简单地讨论一下内存管理问题，并在下面引入一通用管理模式。对该模式的改进以及对内存管理策略的更详细讨论可参考第14章“服务器资源管理”。

1. 静态CORBA对象

静态对象是在系统整个生命时期中存在的CORBA对象。通常，这些对象是组件入口点（对组件入口点的更详细讨论可参考第18章“工程化过程的重要性”）。例如，CORBA命名服

务必须提供一根命名上下文，用来创建新的命名层次。这个根命名上下文可归类为静态的，因为它始终存在。

从内存管理的角度来看，静态对象的实现是很简单的。通常，静态对象可通过在服务器主线中实例化伺服对象来实现，并把它直接绑定到相关的 CORBA对象（即早期绑定）。

2. 瞬态CORBA对象

瞬态CORBA对象并不和任何持久的状态相关联——它确实是瞬态的。通常，瞬态CORBA对象绑定到有状态伺服对象，即对象的状态仅由伺服对象包含。不幸的是，这意味着瞬态CORBA对象的生命周期紧密绑定到伺服对象的生命周期：瞬态 CORBA对象的创建必然导致伺服对象的立即激活，以实现对象。另一方面，伺服对象的销毁会立即导致相关 CORBA对象的删除，因为所有的状态都随着伺服对象而消失。

瞬态对象的一个恰当例子是迭代器，它使用户能反复查看查询的结果集合（参看第 14章“服务器资源管理”的StockPrice迭代器例子）。迭代器对象并不和任何持久的状态绑定，因为查询结果通常是瞬态的。为瞬态 CORBA对象找出好的内存管理策略会是很困难的。第 14章“服务器资源管理”也会讨论这个问题。

3. 持久CORBA对象

最后，持久CORBA对象和其他持久状态相关联，并由数据管理系统来维护。这使用户可以为这些对象的实现应用十分灵活的内存管理策略，因为我们可以使用后期绑定来动态激活和冻结伺服对象。

4. 伺服对象池模式

对CORBA对象实现的分类，说明了从内存管理的角度来看，不同的对象有不同的需求。伺服对象池模式为伺服对象管理定义了一个通用的框架。它的基本思想是包含一个池管理器，来管理激活伺服对象所在的池。每个伺服对象和一驱逐策略相关联。该策略描述了伺服对象何时被逐出。池管理器有两个角色：保持器和驱逐器。保持器保证对象在需要时存在。例如，瞬态对象不能重新创建，所以它必须保持到客户机对它的请求完成。驱逐器必须保证伺服对象是经常被逐出的，以避免不必要的资源消耗。对伺服对象池模式的详细讨论可在第 14章“服务器资源管理”中找到。

### 2.4.6 对象生命周期：评估准则

迄今为止，已一般地考察了 CORBA对象的生命周期，包括生命周期事件，对早期和后期绑定的讨论，以及CORBA对象实现的一般分类。显然，用户希望 ORB提供的应用程序能支持所有这些CORBA对象生命周期不同方面的有效实现。 ORB通过对象适配器（OA）来提供这种支持。下面定义了一系列的评估准则，通过这些准则可对对象适配器进行涉及 CORBA对象生命周期有效支持的分析。然后用户采纳这些不同的评估准则，并把它们应用到 BOA和POA代的对象适配器中。

- 适配器结构　最为重要的方面是适配器的一般结构。
- 对象标识　CORBA系统中的对象标识不是小问题，必须仔细检验特定的 ORB代如何为CORBA对象支持对象标识的概念。
- 早期绑定　用户需要检验不同ORB代支持早期绑定的方式。这里值得注意的是绑定和伺服对象创建相互间有多大程度的关联。

- 后期绑定 不同ORB为后期绑定提供的机制必须要检验，特别是关于用来支持持久对象的应用程序的有用性。
- 无状态伺服对象 用户想要检验ORB代为实现无状态伺服对象而提供的支持。思路是对于无状态伺服对象，用户并不真正需要每个 CORBA对象代表一个伺服对象实例——一个单独的伺服对象可以作为多个 CORBA对象的瞬态胶囊，并在每个请求的基础上设定一个特别CORBA对象的标识。
- 有状态伺服对象 正如前面讨论过的，对于有状态伺服对象，确保用户不必为每个请求重新激活这些伺服对象，并且激活伺服对象未超过某一阈值，这通常是很重要的。

### 2.4.7 对象生命周期：评估BOA代

下面想在上述定义的每个与对象生命周期相关的评估准则下探讨基本对象适配器。下一节将对可移植对象适配器完成同样的工作。

1. BOA 体系结构

前面提过，BOA体系结构在很多方面是极其模糊的。应用程序要求用来有效管理对象生命周期的很多特征都不够具体。 BOA定义了一些激活CORBA服务器和CORBA对象实现的函数。不幸的是，BOA侧重于服务器激活，而不是像后期绑定和动态对象激活这些重要的问题。BOA还为对象引用的生成和解释定义了一些函数。 BOA隐含了伺服对象和CORBA对象间的一对一关系，即对于特定服务器支持的每个 CORBA对象要求有一专用的伺服对象实例。

因为BOA规范过于模糊，所以本书使用 IONA Technologies Orbix 2.x ORB作为参考实现。同样，本书的目的不是要给出不同 BOA代ORB的全面比较，而是使用 ORB作为例子来讨论一般的概念。

2. 对象标识

BOA代的ORB把对象引用和创建时的引用数据相关联。引用数据是 8位位组序列，并由ORB控制。这就使在遵循CORBA的方式中提供应用程序定义的对象 ID变得困难。

Orbix ORB提供了 _marker() API来为CORBA对象指明应用程序定义的引用数据。 marker是一字符串，由应用程序提供，并作为引用数据的一部分由 ORB存储。

3. 早期绑定

如前所述，大多数 BOA代ORB在客户端桩类层次和服务器端框架类层次之间提供了紧密的耦合。这样做的结果是伺服对象的创建通常会间接导致对象的激活，因为伺服对象继承了构造方法，这个构造方法能间接在伺服对象和 ORB运行时模块之间创建绑定。这就暗示，如果用户想把对象引用返回给客户机，也就要创建和激活伺服对象实例，即使用早期绑定。

4. 后期绑定

BOA结构并没有定义ORB和应用程序之间如何交互以支持后期绑定，或是通过命令进行对象激活。因此，本书把Orbix ORB作为BOA ORB如何论述这个问题的例子。

Orbix ORB使用图2-10所示的装载器机制来支持后期绑定。如前所述，装载器扮演伺服对象管理器的角色。基本思想是应用程序把装载器实例注册到 ORB运行时模块。 ORB运行时模块在对象故障的情况下依次激活装载器上的 load()方法。这就给了装载器一个机会来激活请求对象，这样ORB就可以分派它。

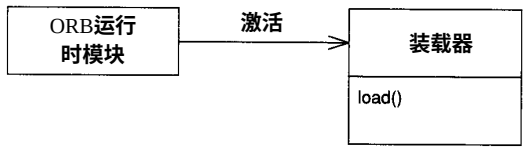


图2-10 Orbix装载器机制

应用程序的职责是决定伺服对象是只为特定的请求服务还是为后面的请求继续保持激活。Orbix装载器特征可被应用程序用来实现为对象伺服对象池模式，而这些对象可用命令，即持久对象激活。

5. 无状态伺服对象

BOA代ORB实现并没有为无状态伺服对象提供特别的支持。基本上， CORBA对象和伺服对象之间一对一的关系暗示着必须为每个 CORBA对象激活一个伺服对象。根据实例化和激活一无状态伺服对象的花费，用户要决定它是否值得实现为池模式，以减少对象激活 /冻结的次数。这包括调查由 ORB强加的伺服对象激活和冻结的开销，以及和应用程序相关的花费。问题在于从性能的角度来看是在每个请求的基础上激活伺服对象昂贵，还是维护一对象池昂贵。

6. 有状态伺服对象

前面讨论过，对于有状态的伺服对象，伺服对象的激活 /冻结是很昂贵的，特别是如果它包括了数据库的查找。在这种情况下，把后期绑定和对象池结合起来就很有意义。因为 BOA规范在这里不是很明确，所以必须为后期绑定而依赖于专有的 ORB支持，象Orbix装载器。

### 2.4.8 对象生命周期：评估POA代

现在把评估准则应用到可移植对象适配器。

1. POA体系结构

POA体系结构在开始时可算是占绝对优势的，因为它为管理伺服对象和 CORBA对象提供了很多先进的特性。幸运的是，以很简单的方式来开始使用 POA是可能的。但是，不同对象类型的不同需求确实需要 POA的高级支持机制。

图2-11给出了POA结构顶层元素的概述。 ORB通过POA来管理伺服对象。每个 ORB和一个POA关联。通过调用 POA上的create_POA()并传递新POA的名字，POA可用来创建嵌套的 POA，这是创建新POA的唯一方法，意味着应用程序不能提供它自己的 POA实现。相反，应用程序通过把新 POA和一系列策略关联来定义新 POA的具体行为。这些策略根据如伺服对象激活、伺服对象生命期限、 ID管理和线程分配等事项来定义 POA的行为。

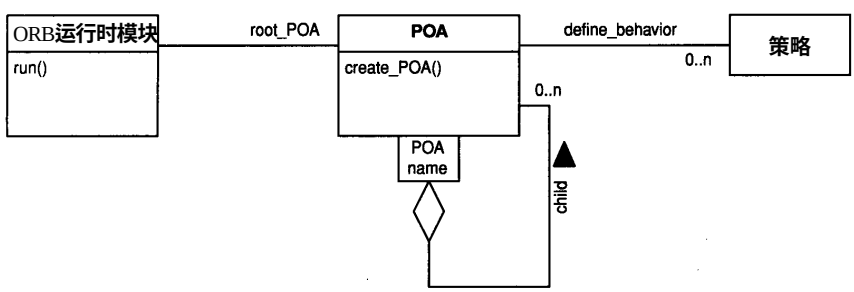


图2-11 POA概貌

图2-12给出了POA和伺服对象之间关系的概述。POA规范采用术语“化身”(incarnation)来指出伺服对象和CORBA对象的关联。在类图中，可以看到这个化身关系，它由对象ID来完成，即POA可通过相关联的对象ID来定位伺服对象。显然，每个POA可以和多个伺服对象相关联。令人惊奇的是一个单独的伺服对象可代表多个CORBA对象！事实上，为多个对象注册一个伺服对象不仅是可能的，甚至注册一个缺省的伺服对象，用来获取没有通过ID而和伺服对象明确相关联的对象的所有请求也是可能的。本书会在下面讨论这个特征的有用性。POA规范还引入了术语“神化（etherealize）”，它是化身的反义。这就是把伺服对象与CORBA对象间的关联取消的操作。

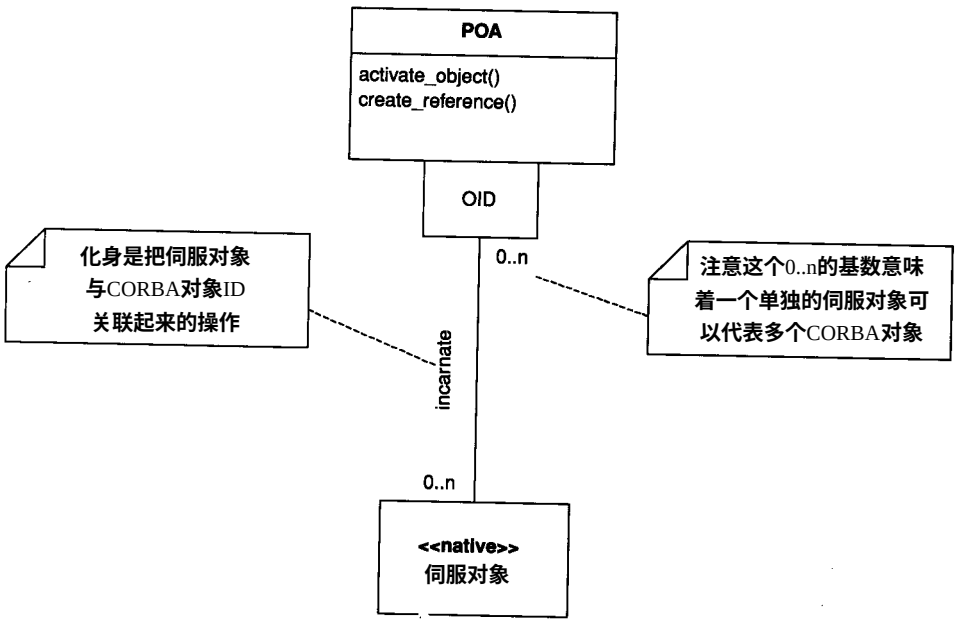


图2-12 POA和伺服对象

2. 对象标识

CORBA的POA代对术语“对象关键字”和对象ID有明确的区分。对象关键字在通信结束点标识对象。例如，对象关键字可含有POA的名字，通过这个名字可到达目标对象实现。另外，对象关键字还可包含一对象ID。对象ID标识和具体POA相关的对象。根据特定的POA策略，新创建对象的ID可由POA或应用程序定义。

3. 早期绑定

为了使用POA的早期绑定，要以明确的激活来使用称为伺服对象保留的策略（用activate_object()或activate_object_with_id()）。

要注意，POA规范把伺服对象的激活和对象引用的创建相隔离。现在把对象引用返回给客户机并不一定意味着服务器必须激活对象；即用户没有执行早期绑定也可以返回对象引用。这意味着用户可以更多地使用后期绑定。当使用无状态服务器和代表所有无状态伺服对象的单独伺服对象的结合时，就会有好处；在这种情况下，用户只需实例化一个单独的伺服对象，并把它作为缺省的伺服对象注册。缺省的伺服对象会为所有对象处理即将到来的请求，我们只要简单地创建和输出这些对象的引用就能使它们变为可用。

4. 后期绑定

POA支持伺服对象管理器的概念作为后期绑定的基础机制，如图 2-13所描述。在对象故障的情况下，POA会激发注册的伺服对象管理器，试图获取请求对象的化身。这意味着伺服对象和CORBA对象间的绑定只为特定的请求而创建（当使用伺服对象定位器时），或者绑定会比请求期存在得更久（当使用伺服对象激活器时）。

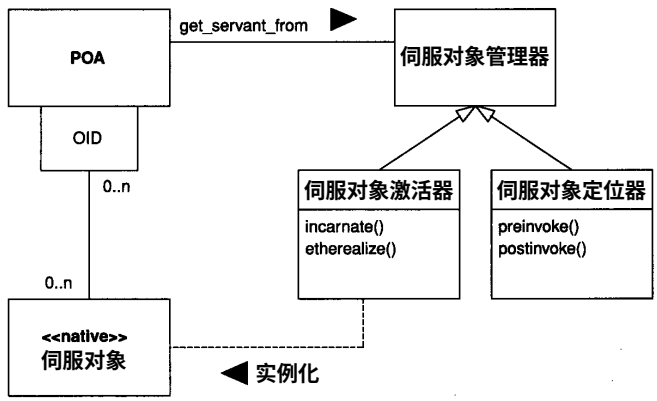


图2-13 POA中的后期绑定支持

5. 无状态伺服对象

POA规范以两种方式提供对无状态伺服对象的支持。首先，应用程序可注册一个缺省的伺服对象，它可以为所有通过特定 POA能访问到的对象处理到来的请求。其次，可以使用伺服对象定位器：伺服对象定位器使伺服对象只在请求期间对 POA可用。这意味着伺服对象会在请求期间采用一个特定的ID。POA提供一个API，使伺服对象在执行请求时能找出它自己当前的识别。

6. 有状态伺服对象

如前面所讨论的，对于有状态伺服对象，用户通常想维护激活伺服对象的池来减少花费巨大的激活的次数。第 14章“服务器资源管理”中将讨论伺服对象池模式。一个重要的问题是谁来实际维护CORBA对象到伺服对象的映射。图 2-12指出了伺服对象通过它的ID和POA关联。但是情况并不总是如此。有了伺服对象定位器机制，用户可以在每个请求的基础上创建对象/伺服对象关联。

注意伺服对象定位器通常在 ORB的控制之外来管理自己的对象 ID/伺服对象映射。这是实现有效的驱逐机制的最简易方法。

## 2.5 小结

本章中讨论了远程 CORBA激发的生命周期和 CORBA对象的生命周期。在这里引入的多个概念是本书余下部分所涉及主题的重要基础。本书为 BOA和POA代的ORB实现考察了激发和对象生命周期事件。正如读者能够看到的， BOA规范经常是很模糊的，并没有表达很多要求用来有效管理远程激发和 CORBA对象生命周期的特性。 POA规范初看上去过于复杂，但另一方面，POA规范覆盖了很多BOA代ORB以专有方式实现的特性，而这些特性并没有被 BOA规范所覆盖。最后，开发者可用简单的方式来使用 POA，以后根据需要可利用更多的特性。