《软件构件与中间件技术》

宋胜利
shlsong@xidian.edu.cn

（西安电子科技大学软件学院，710071）

主要内容：

- 概述：中间件与软件构件的动因与基本概念。
- CORBA 中间件：CORBA 的基本原理、CORBA 应用的基本开发过程（CORBA 构件的开发与使用）、CORBA 中构件接口的编写、CORBA 服务端程序的编写。
- Java 企业版中间件：J2EE 的基本概念、EJB 构件的开发与使用、Java 企业版中间件服务的使用。
- Web Service 体系结构：Web Service 体系结构简介、SOAP、WSDL、UDDI。

本书的关注点主要有两个：

- 如何开发与使用基于特定中间件的构件
- 如何使用中间件提供的各种支持

本书中所有例子程序均使用 Borland 公司的 VisiBroker for Java 4.5.1 和 Sun 公司的 Java 企业版参考实现平台开发，这些例子很容易移植到其他开发平台。读者可从西安电子科技大学相关教学网站下载这些例子程序的全部源代码。

声明：本书 CORBA 部分（第二部分）内容来自于从网络途径获得的由李文军、周晓聪、李师贤三位老师编写的中山大学学习资料《分布式软件体系结构》，其中少量地方根据个人观点进行了修改。本书仅用于西安电子科技大学的教学用途。

第一部分　概述

第 1 章　　软件构件与中间件基本概念

本章介绍分布式软件的基本概念、软件构件的基本概念、中间件的动因与基本概念；利用 jdk 的远程方法调用 Java RMI 开发了一个简单的分布式应用程序，通过该例子演示软件构件与中间件技术为软件开发提供的基本支持。

§ 1.1 分布式软件的基本概念

1.1.1　分布式软件与客户机/服务器模型

在计算机硬件技术与网络通信技术的支持下，应用需求驱使计算机软件的规模与复杂度不断增长，软件正变得无处不在，同时软件所面临的挑战也正在日益加剧，软件开发过程中复杂度高、开发周期长、可靠性保证难等问题日益突出。在这种背景下，软件开发人员不得不在软件开发的过程中寻求更多的支持，以帮助其在特定的开发周期内开发出规模更大、更可靠的软件系统。

本书关注在上述背景下大型分布式软件系统的开发支撑。原因主要有两个：

- 随着网络与通信技术的发展，分布式软件的应用越来越广泛，分布式软件在计算机软件应用领域扮演着非常重要的角色。
- 分布式软件一般比集中式软件规模大、复杂，是软件开发复杂性的集中体现。

简单地讲，分布式软件指运行在网络环境中的软件系统，而网络环境是一群通过网络互相连接的处理系统，每个处理节点由处理机硬件、操作系统及基本通信软件等组成。分布式计算有两种典型的应用途径。第一种应用途径是将分布式软件系统看作直接反映了现实世界中的分布性，例如当今许多业务处理流程通常呈现一种分布式运作方式，如某生产制造企业，其负责加工制造的工厂可能位于珠江三角洲一带，而负责销售的部门则可能分别位于北京、上海和广州，这时负责业务流程的软件系统显然也应做相应的分布式处理。第二种应用途径主要用于改进某些应用程序的运行性能，使它们比单进程的集中式实现更具有效率，如利用互联网上的大量计算机实现海量数据的科学计算或分析，此时软件系统的分布性并不是现实世界中分布性的映射，而是为利用额外的计算资源而人为引入的。

分布式软件通常基于客户机/服务器（Client/Server）模型。如果一个系统两个组成部分存在如下关系：其中一方提出对信息或服务的请求（称为客户机），而另一方提供这种信息或服务（称为服务器），那么这种结构即可看作是一种客户机 / 服务器计算模型。互联网的许多应用程序都采用客户机 / 服务器模型，例如 Web 浏览器与 Web 服务器、电子邮件客户程序与服务程序、FTP 客户程序与服务程序等；更一般地，在普通的函数或对象方法调用中，执行调用语句的子程序与实现函数/方法体的子程序或对象可看作一种客户机 / 服务器模型，其中实现方是服务器，调用方是客户机。

分布式软件与传统的集中式软件主要区别在于强调客户端与服务端在地理位置上的分离，这种分离可带来许多好处：

- 更好的支持平台无关性：客户端和服务端可以运行在不同的硬件（PC、工作站、

小型机等）与操作系统（Windows、Unix、Linux 等）平台上；

- 更好的可扩展性：可以在服务端功能不变的前提下对服务端的程序进行改进或扩充，而这种改进或扩充不会影响到客户端的应用程序。

当然客户端与服务端的分离也使得软件系统更复杂：

- 开发人员不得不分别编写客户端和服务端应用程序，并力求保持两者的一致性；
- 软件系统的调试、部署、维护更加困难；
- 需考虑更多的可靠性、安全性、性能等软件质量因素。

1.1.2 分布式软件的三层/多层结构

早期的分布式系统基于图 1-1 所示的两层结构。在两层结构中，简单地将软件系统划分为服务器层和客户层，服务器层又称为数据层。在服务器层，一般放置一个数据库服务器，上面安装一个数据库管理系统，存放系统用到的持久数据。而客户层则实现系统的主要业务功能，实现时需要访问数据库中存放的数据，一般会有多个客户端同时访问数据库服务。这时的系统结构比较简单，就是多个客户端程序共享一个数据库。

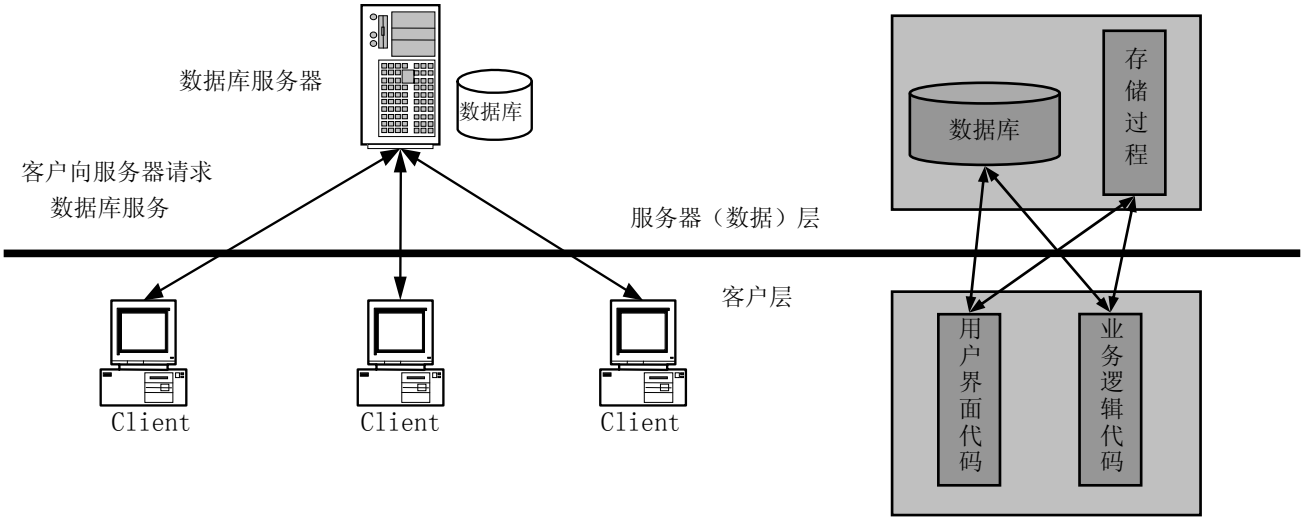


图 1-1 两层结构的分布式系统

两层结构中软件开发的主要工作量在客户层。数据层基本没有什么程序代码，主要就是建好数据库，可能利用存储过程实现一些基本的业务逻辑。开发人员所编写的代码几乎全部都在客户端，一般可以把客户端的代码分为用户界面相关的代码和业务逻辑相关的代码，在客户端的代码中要访问数据库中的数据，可以执行一些 SQL 语句或调用存储过程。

两层结构下，客户程序直接访问数据库，并且用户界面代码和业务逻辑代码交织在一起，这些导致两层结构存在以下重要的缺陷：

第一，客户端的负担比较重。一般认为，客户端程序只要为使用该系统的用户提供一个人机交互的接口就行了，但是在两层结构下，客户端仍然需要进行比较复杂的数据处理，因为客户端从数据库中得到的仅仅是一些原始的数据，必须按照业务逻辑的要求对这些数据进行一定的处理后才能呈现给用户，所以客户端的负担比较重。

第二，客户端的可移植性不好。处理复杂必然牵涉更多的移植性问题，另外在两层结构下，每个客户端上都要安装数据库驱动程序，移植至少需要重新安装数据库驱动。

第三，系统的可维护性不好。因为客户端包含过多的业务逻辑，并且业务逻辑与人机交互界面交织在一起，无论是用户界面需要修改，还是业务逻辑需要修改，都很麻烦。

第四，数据的安全性不好。两层结构下，数据库必须为每一个客户端机器开放直接操作数据库的权限，这时就很难防止一个恶意的用户在某个客户端机器上利用该权限执行其不应该执行的操作。

鉴于以上原因，人们提出了图 1-2 所示的三层结构：

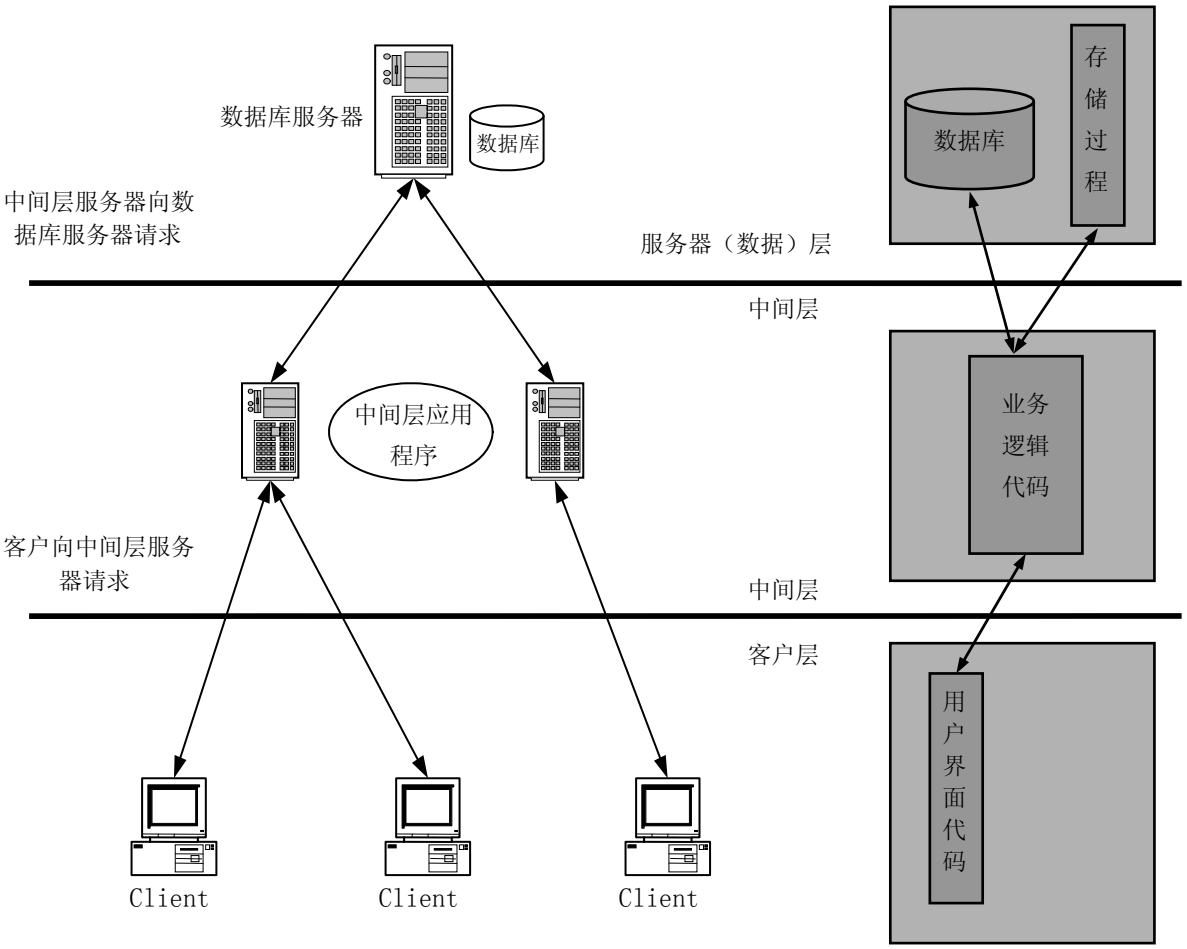


图 1-2　三层结构的分布式系统

在 3 层结构下，在数据层和客户层之间再增加一个中间层，将原来放在客户端的业务逻辑代码移到中间层来。客户程序与数据库的连接被中间层屏蔽，客户程序只能通过中间层间接地访问数据库，即降低了客户端的负担、改善了其可移植性，又提高了系统的数据安全性；同时业务逻辑代码与用户界面代码相对独立，也在很大程度上提高了系统的可维护性，较好地解决了两层结构的上述问题。

三层结构中软件开发的主要工作量在中间层，中间层包括除用户界面代码与持久数据之外的几乎所有系统代码，是整个软件系统的核心。

在 3 层结构中，客户层和数据层已被严格定义，但中间层并未明确定义。中间层可以包括所有与应用程序的界面和持久数据存储无关的处理。假定将中间层划分成许多服务程序是符合逻辑的，那么将每一主要服务都视为独立的层，则 3 层结构就成为了 n 层结构。典型地，可将业务逻辑层分离出实现数据持久化操作的持久层，用于实现对于持久数据操作的封装，从而形成由客户层、业务逻辑层、持久层与数据层构成的四层结构。

§1.2 软件构件的基本概念

本节从分布式对象的角度讨论软件构件的特性。分布式对象是构成分布式系统的软件构件，除了具备一般软件构件的特征外，其重要特征就是分布特性。

1.2.1 对象到构件的发展

按照面向对象的观点，软件系统由若干对象组装而成，将这一观点延伸至分布式系统，分布式系统由若干分布式对象组装而成。面向对象的精髓之一在于对象维护自身的状态并通过消息进行通信，这对于分布式计算环境是非常理想的，但是分布式对象与传统对象相比，有其特殊的特性。

首先，分布式系统由于其规模与分布特性等原因，比集中式系统更容易被拆分给不同的人或团队开发，这就很有可能遇到不同的人或团队习惯使用不同的程序设计语言和环境的情况。但是，这些使用不同语言的人（团队）开发的程序要能够方便的交互，换句话说，就是用不同的语言所写的对象要能够方便的交互。而在传统的集中式面向对象系统中，显然对象之间的交互是局限于某种特定的语言的。从语言这个角度我们可以看到，分布式对象比传统对象面临的环境更复杂，这要求分布式对象比传统对象具备更好的封装特性，比如，要把实现所使用的语言屏蔽起来，对象的使用者看到的是一个跨语言的对象。

更进一步讲，分布式系统要求分布式对象可以在任何粒度上透明的使用，也就是说，无需考虑位置与实现。不需要考虑具体在哪，是与使用者在同一个进程内，还是在不同的进程内，如果不同又在哪个进程内，或是在哪个机器上的哪个进程内，或者是运行在什么样的操作系统之上等等，这些具体的位置信息使用者都不希望去过多的关心。另外也不需要关心对象的具体实现细节：如是用什么语言实现的，是不是用面向对象的语言实现的，还是用一个函数库甚至是一个面向对象数据库实现的，都不用去关心。

传统的面向对象语言中的对象很难满足上面提到的要求。传统对象的关注点是封装和通过继承对实现进行重用，封装提供了一种将对象实现细节与其他对象屏蔽开的严格方法，可以大大缓解在面向过程系统中较突出的维护问题，继承提供了一种重用对象实现的简便方法。而分布式系统要求分布式对象要有更好的可插入性，这个要求仅仅依靠传统面向对象的封装和继承是不可能满足的。首先，要求另一层次上的封装，只需暴露公用接口；其次，从重用的角度来讲，继承局限于程序设计语言，而分布式系统不太关注于直接重用代码，而是要求能够利用远程所实现的服务。这就是说，分布式系统中的分布式对象和传统的对象不一样，实际上是具有良好封装特性的软件构件。

一般地讲，构件指系统中可以明确辨识的构成成分，而软件构件指软件系统中具有一定意义的、相对独立的构成成分，是可以被重用的软件实体。构件提供了软件重用的基本支持，分析传统工业，如机械、建筑等行业，以及计算机硬件产业，其成功的基本模式是符合标准的零部件（构件）生产与基于标准零部件的产品生产（组装），其中复用是必需的手段，而

构件是核心和基础，这种模式是产业工程化、工业化的必由之路，也是软件产业发展的必然途径，这是软件复用与软件构件技术受到高度重视的根本原因。

由于本书关注分布式系统，对应软件构件也关注分布式构件，因此除非特别声明，以后论述中不区分构件与分布式对象这一对概念，一般提到的构件均指分布式构件，即分布式对象，反之亦然。

1.2.2 软件构件的特性与相关概念

与对象相比，构件通常具有如下特性：

- 构件是一个严格定义的可插入单元：类似于硬件模块，一旦开发完成，就可以方便的用来组装系统。构件一般是基于对象实现的，但也可以不作为对象实现。
- 构件将封装运用到了极限：构件通过封装来隐藏构件的实现以达到：
  - 构件的实现语言是未知的：如一个 Java 客户端不会感觉到所使用的构件是由 C++实现的。
  - 构件的物理位置是未知的：一个 VB 客户端不会感觉到所使用的构件是运行在相同的进程内（使用 DLL），还是在同一机器的不同进程内，甚至是位于不同机器上。
- 构件通常在容器中进行管理：按照上面的讨论，构件要屏蔽实现语言、实现方式等很多实现细节，使用一个构件时可以不关心具体实现和位置。这仅是指开发人员不用关心，是不是谁都不用管这些事情，当然不是，如果谁都不去处理不同语言、不同实现方式、不同的物理位置所带来的差异，那用不同语言、不同实现方式实现的、位于不同的物理位置的构件就不可能很好的交互。那具体谁去作这个工作呢？通常的做法是设置容器来为构件一个运行环境，容易是一种特殊的应用程序，构件在容器中运行和管理。当有客户端要访问某个构件时，实际是向容器发起了一个请求，容器再去判断构件的具体实现语言、实现方式以及构件的具体物理位置，然后帮助请求者与构件进行交互。也就是说，我们所使用的、具备很好特性的构件实际上是容器和我们用各种语言、方式所编写的程序共同协作的结果。这就要求构件遵循所处的容器的规则，并按照标准的途径向容器发送事件。
- 构件可以从容器中获得属性或服务：如构件可以使用容器的背景色作为自己的背景色。
- 构件允许对所支持的接口进行动态发现和调用：客户程序可以在运行状态下确定一个构件支持何种功能，然后调用该功能。

既然构件具备极好的封装性和可插入性，那么基于构件的系统从可维护性上来讲就可以达到一种比较理想的状态。如图 1-3 所示，一个构件化的软件系统在进行维护或升级时，在保持构件接口不变的前提下，可单独对系统中若干个构件进行修改，而不会影响构成系统的其它构件。

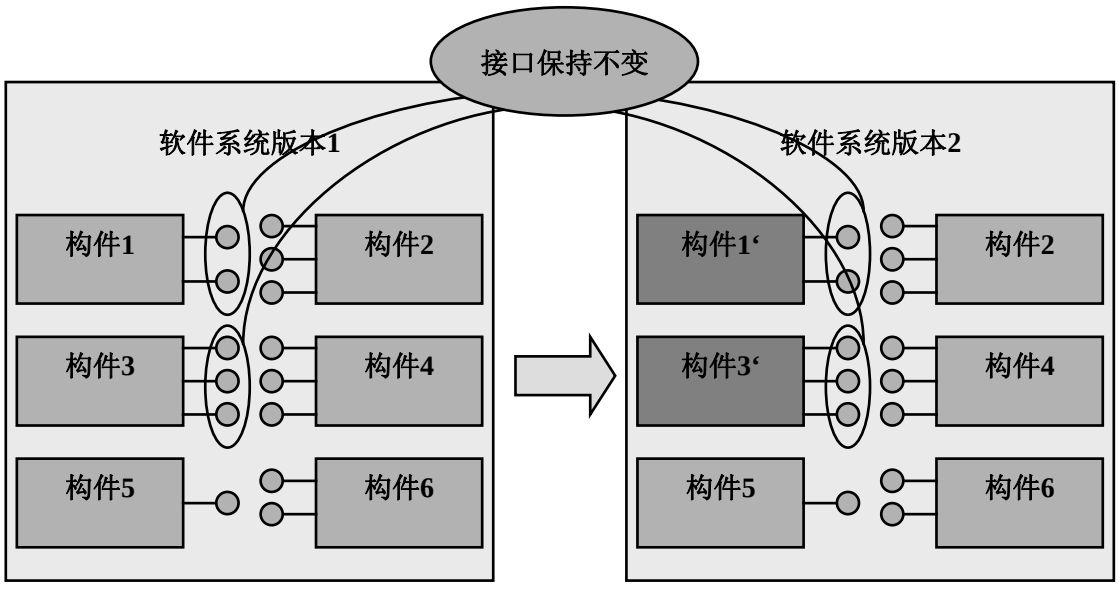


图 1-3　构件化软件系统的升级方式

下面介绍几个构件相关的概念。

- **接口：**接口是系统中用来定义分布式对象能力的约定。由于构件将封装运用到了极限，构件之间相互看到的就只有接口。这里的接口和程序设计语言，如 Java 中的 interface 所起的作用基本一致，但构件的接口通常是跨语言的。
- **数据类型：**分布式对象之间不可避免的要进行交互，而交互的最基本支撑就是要定义在分布式对象之间传输的数据类型。一般说来，现有程序设计语言中已有的数据类型不能直接在分布式系统中使用，因为分布式系统通常涉及多种程序设计语言与多个平台，而不同语言或平台的数据类型往往不能兼容，如整型数在不同的语言或平台上可能字节数或字节顺序不尽相同，因此一般需要一种独立于语言和平台的数据类型系统。
- **编组与解组（Marshaling and Unmarshaling）：**分布式对象之间交互时，由于一般至少跨越了进程边界，因此交换的数据通常要在网络上进行传输，而网上传输的数据只能是串行化的流数据，所以需要把程序员熟悉的有类型的数据转化成便于网络传输的流数据发送出去，并且需要把网络上接收到的流数据转化成程序员容易处理的有类型数据，这就是编组与解组的过程。编组过程将数据封装成标准的格式以便于传输，解组过程则负责打开传输来的数据。
- **对象句柄（Object Handle）：**对象句柄是在客户程序的编程语言或脚本环境内用来引用分布式对象的实例。从功能角度上讲，对象句柄类似于面向对象语言中的对象指针或对象引用，但是具体实现上和对象指针或对象引用却存在本质的差别，由于对象句柄是对远端分布式对象的引用，因此不可能像对象指针或对象引用那样指向对象在内存中的首地址。
- **对象创建（Object Creation）：**和面向对象语言中使用一个对象要先 new 类似，分布式对象系统必须为创建一个新的分布式对象实例提供一种机制，但是这也不同于简单的 new，要涉及更多的其它工作，分布式系统中经常使用工厂（factory）来完成对象创建工作。工厂是一种特殊的分布式对象类型，常用来创建其它的分布式对象。
- **对象调用（Object Invocation）：**分布式对象系统必须为分布式对象的调用提供一种机制。通常情况下开发人员不希望自己写代码编组与解组这些复杂的底层工作，而是希望像使用面向对象语言中的对象那样实现分布式对象的调用，通过后面的讨论我们可以发现，有了中间件的支持就可以实现。但是分布式系统的开发人员一定要随时牢记，分布式对象的调用一般会涉及一个多方参与的跨越网络通信的过程，忽略这种区别会给开发的系统带来很大的危害。

§ 1.3 中间件的基本概念

1.3.1 中间件的动因

尽管有了构件技术的支持，但是随着软件系统规模与复杂度的不断提高，软件开发过程中复杂度高、开发周期长、可靠性保证难等突出问题并没有得到根本缓解；而分布式软件面临更大的挑战，分布式软件所运行的网络环境具有明显的分布性、开放性、演化性、异构性、并发性等特征，因此分布式软件必须解决互操作、数据交换、分布性、可行性等一系列更复杂的问题。

究其本质原因，在于人们控制复杂性的能力相对稳定，但面临的问题却越来越多。在现

实生活中，如果遇到一件很复杂的事情要完成，我们往往会寻求工具的支持，很多工具的作用是帮人们完成重复性的、每次手工做起来又很费力费时的工作，在软件开发时解决问题的思路是一致的。基本的解决思路就是抽取软件的共性成分，抽取的共性成分由系统级软件完成，向开发人员屏蔽系统低层的复杂度，从而在高层保持整体复杂度的相对稳定。在软件领域，这种解决思路往往导致新型系统软件的产生。操作系统与数据库管理系统的产生就是经历了类似这样的过程。

在操作系统出现之前，计算机的初始组成就是“硬件＋程序”，即程序直接运行在裸机硬件之上。此时，应用程序直接控制硬件的各种运行细节，应用程序中存在大量的代码用于管理各种物理器件，以访问数据为例，程序必须控制怎样连接磁盘，如何读取其中的数据，如何处理各种异常情况等。这使得程序代码十分庞大，而且正确性难以保证。随着计算机应用的日益广泛，程序的规模不断增大，软件开发变得越来越困难。在这种背景下，人们进行了软件共性的第一次抽取，即抽取出了程序的共性（稳定）成分——计算机资源管理，此次共性的抽取导致了操作系统的产生，分离出了应用程序。初期的操作系统被称为管理程序或监督程序，提供大量的与硬件相关的代码（系统调用）来完成上层应用程序的各种请求，隐藏了与硬件相关的程序执行过程的复杂性，从而简化了应用程序的开发。

操作系统形成之后，计算机的组成变成了“硬件＋操作系统＋应用程序”。此时，应用程序中访问的数据和应用程序一样以简单文件的方式存储，应用程序的开发人员需要了解数据的具体组织格式，并且需要自己编写程序解决完整性等相关问题。随着应用程序处理的数据规模越来越大，应用程序中数据管理这一共性也越来越明显，即应用程序中普遍存在大量代码实现数据管理功能。于是人们进行了软件共性的第二次抽取，即抽取出了程序的共性（稳定）成分——数据管理，此次共性的抽取导致了数据库管理系统的产生，分离出了应用软件。数据库管理系统对数据进行统一的管理和控制，并保证数据库的安全性和完整性，为用户屏蔽系统关于数据存储和维护等的细节，从而再次简化了应用程序的开发。

类似的工作仍在继续，在软件系统规模与复杂度不断提高的同时，人们不断从应用软件中提取共性、降低高层复杂性，最终导致了中间件的产生。与操作系统、数据库管理系统类似，中间件是在操作系统（数据库管理系统）与应用系统之间的一层软件，通常为分布式应用的开发、部署、运行与管理提供支持。

1.3.2　中间件提供的支撑

在中间件应用的早期，人们依据所抽取出的应用软件中的不同共性设计与实现了多种类型的中间件，一般一种类型的中间件实现一种共性功能，为应用软件提供一种开发支撑。由于所属的具体领域不同，面临的问题差异很大，因此不同开发组织分离、开发出的中间件也不尽相同。以下是几种常见的中间件以及其提供的支持：

终端仿真/屏幕转换中间件：用以实现客户端图形用户接口与已有的字符接口方式的服务器应用程序的互操作。在该种中间件支持下，可以很容易地为原有字符界面的应用程序提供图形用户界面。

数据访问中间件：在数据库管理系统的基础上，对异构环境下的数据库实现联接或文件系统实现联接的中间件，为应用程序访问数据库提供开发支撑。

远程过程/方法调用中间件：用以实现远程过程或方法调用的中间件。向应用程序提供远程调用时的底层通信支持，帮助应用程序完成编组与解组等工作，程序员方便地编写客户端应用程序，像调用本地过程或对象那样方便调用位于远端服务器上的过程或对象方法。

消息中间件：为应用程序提供发送和接收异步消息支持的中间件。基于消息的交互方式提供了基本的异步编程模式，即客户端可以通过发送消息来请求某种服务，在服务端处理请求期间，客户端不必等待对方完成，可以执行其它操作，服务端完成后会以消息的形式通知

客户端。在消息中间件的支持下，应用程序可以很容易地实现消息的发送和接收，而不必关心消息交换过程中的具体细节。

事务（交易）中间件：提供事务控制机制的中间件。事务（交易）管理支持可靠的服务端计算，这在很多关键系统中都是必需的。事务的基本特征是维护一系列操作的原子性，如银行业务系统中转账功能所包含的扣除源账户余额与增加目标账户余额两个操作，这两个操作从业务逻辑上讲应该是原子的，即要么全部都完成，要么全部都不做。基于事务中间件，应用程序可以很方便地实现事务控制，而不必关心具体事务控制的细节。

构件中间件：提供构件化支持的中间件。在分布、异构的网络计算环境中将各种分布对象有机地结合在一起，完成系统的快速集成，实现对象重用。

有了各种中间件的支撑，在应用软件中用到中间件对应的功能时，不需要开发人员自己实现，可直接利用中间件将其已实现好的功能快速集成到应用软件中。

随着中间件应用越来越广泛，又出现了一个新问题：中间件越来越多，开发时需要安装的支撑环境越来越复杂，开发人员不得不花费越来越多的时间安装与配置需要的各种中间件。因此自然地出现一种中间件集成的强烈需求，在中间件研究的基础上，人们开始考虑将各种中间件的功能集成在一起，现有中间件多以集成中间件的形式出现，集成中间件也称为应用服务器。

现有的集成中间件典型地为三层/多层结构的分布式软件系统提供各种开发支撑，因为三层结构的分布式软件的核心为中间层，因此支撑主要集中在对中间层开发的支撑上。目前应用最广泛的集成中间件有三类：

- 基于 OMG（Object Management Group，对象管理组织）CORBA 规范的集成中间件
- 基于 Sun JEE（Java Enterprise Edition，Java 企业版）规范的集成中间件
- 基于微软.NET 架构的集成中间件

其中前两种所基于的规范均为工业标准，这两种标准得到了产业界众多厂商的广泛支持，因此可供选择的具体中间件产品较多，也是本书主要内容的关注点。第三种基于微软公司的私有技术，因此具体的中间件产品基本局限于微软公司的平台，有兴趣的读者可参考其它相关书籍或资料。为便于论述，在不引起混淆的情况下下文中用“中间件”一词代表“集成中间件”。

现有中间件为分布式软件系统提供的基本支持与分布式软件所运行的网络环境密切相关，具体可归为**提供构件运行环境**、**提供互操作机制**与**提供公共服务**三个方面。

提供构件运行环境：

现有中间件均提供构件化的基本支持，支持方便开发与使用符合特定规范的构件（分布式对象）。中间件一般通过构件容器为构件提供基本的运行环境，具体功能一般包括管理构件的实例及其生命周期、管理构件的元信息等。

提供互操作机制：

因为分布式软件跨越了多台计算机，所以需要一种像 TCP/IP 或者 IPX 这样的网络基础设施来连接应用程序的各节点。现有操作系统（如 Unix、Linux、Windows）或高级程序设计语言（如 Java、C++）均提供了像套接字（Socket）这样的开发接口支持编写跨越网络交互的代码；但是有相关开发经验的读者不难发现，基于这些开发接口实现需要进行比较复杂的开发与调试；而跨越网络的交互是每个分布式系统必须解决的首要问题，因此现有集成中间件均集成了早期远程过程/方法调用中间件的功能，提供了很强的高层通信协议以屏蔽节点的物理特性以及各节点在处理器、操作系统等方面的异构性。基于中间件的互操作支持，开发人员在开发与调用分布式对象时，均不需自己编写处理底层通信的代码。

广泛使用的这种高层通信协议包括以下几种：

远程过程调用（Remote Procedure Call，RPC）：RPC 是第一个得到广泛应用的高层通信协议，使用 RPC，客户应用程序可以像调用本地过程那样调用在远程计算机上执行的 C 语言函数。由于是结构化的，因此目前已经基本被面向对象的通信协议取代。

IIOP(Internet Inter-ORB Protocol)：IIOP 是 CORBA 中使用的一种通信协议，有了它，运行在不同平台上的两个对象可以很方便的进行交互。

DCOM 通信协议：微软在 RPC 基础上实现的分布式 COM 构件间使用的通信协议。

JRMP (Java Remote Messaging Protocol)：特定于 Java 语言，支持用 Java 语言编写的对象之间进行远程交互（Java Remote Method Invocation, Java RMI）的通信协议。

RMI/IIOP：Java 企业版中访问 EJB 使用的通信协议，基于 IIOP 实现。

以上各种协议的共同特征就是帮助应用程序完成编组与解组等跨越网络通信的底层工作，实现远程过程/方法调用中间件的功能。

提供公共服务：

除了互操作的支持外，现有集成中间件将早期各种中间件中针对分布式软件的通用支持集成于一身，以公共服务的形式提供给应用程序。公共服务又称为系统级服务，指由中间件（应用服务器）实现的、应用程序使用的软件系统中共性程度高的功能成分。公共服务有两个基本特征：

- 由中间件而非应用程序实现
- 应用程序中通常会调用其实现的共性功能

与应用程序中开发人员开发的构件实现的功能不同，公共服务通常不实现应用系统中具体业务逻辑，而是为具体业务逻辑的实现提供共性的支持，而开发人员开发的构件则实现具体的业务逻辑。

显然，一个中间件平台所提供的公共服务越多，开发者就越容易在更短的时间内开发出高质量的分布式系统。有了中间件提供的公共服务，开发者可以将主要精力集中于系统的具体业务逻辑。以下是几种常见的公共服务：

事务服务（Transactions Service）：提供支持事务处理的机制，以保证系统状态与数据的一致性与完整性，支持可靠的服务端计算。

安全服务（Security Service）：为系统提供在分布式环境下的安全机制，以防止未授权用户对系统的非法访问。

命名服务（Naming Service）：在分布式系统中，命名服务提供了一种定位分布式对象与其它系统资源的机制。

持久性服务（Persistence Service）：持久性服务使得分布式对象可以通过持久的数据存储来保存、更新和恢复他的状态。

消息服务（Messaging Service）：消息处理服务提供异步编程模式，异步模式在很多应用中都需要。

分布式垃圾回收服务（Distributed Garbage Collection Service）：当一个程序不再使用分布式对象时，分布式垃圾回收服务会自动释放分布式对象所占用的存储单元。

资源管理服务（Resource Management Service）：一般来说，资源管理器按照使可伸缩性最大化的方式来管理分布式对象，即支持大量的客户程序同分布式对象在短时间内进行交互的能力。

§ 1.4 互操作的基本原理与实例

广义的互操作包括中间层应用构件与数据库、客户层构件与中间层应用构件、中间层应用构件与公共服务构件、中间层应用构件之间的互操作等多个层次，本书主要关注应用构件

之间的互操作，即软件系统开发人员编写的程序之间的互操作。

1.4.1 互操作的基本原理

上节提到的 RPC、IIOP、DCOM 通信协议、JRMP、RMI/IIOP 等高层通信协议均可以帮助应用程序完成编组与解组等跨越网络通信的底层工作，实现传统远程过程/方法调用中间件的功能。

这些高层通信协议尽管具体的实现细节不尽相同，但是在实现方式上与开发模式上均采用了 RPC 中相同的通信模型与类似的开发模式，它们采用的通信模型称为 Stub/Skeleton 结构，如图 1-4 所示。

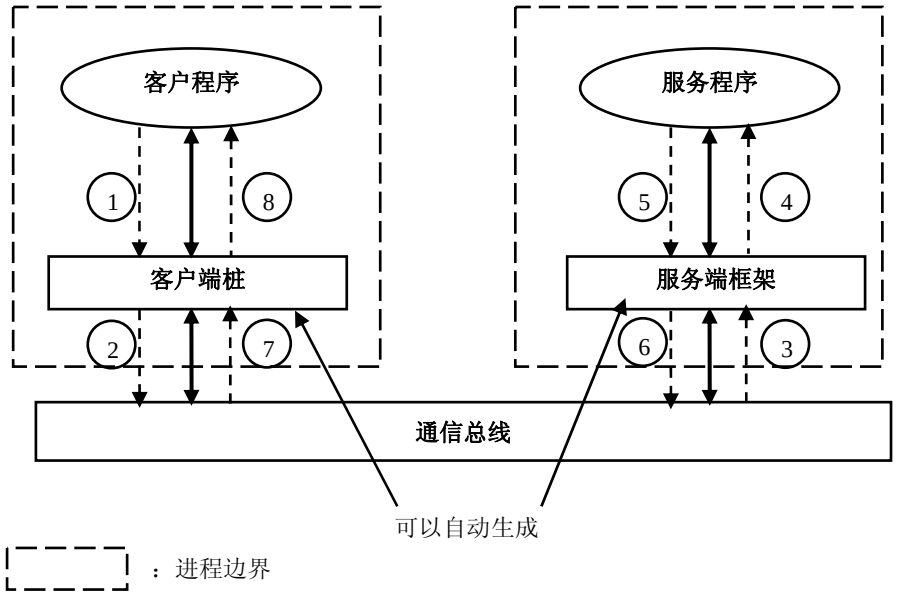


图 1-4 Stub/Skeleton 结构

在 Stub/Skeleton 结构中，由客户端桩（Stub）替客户端完成与服务端程序交互的具体底层通信工作，客户程序中的远程对象引用实际上是对本地桩的引用；而服务端框架（Skeleton）负责替服务端完成与客户端交互的具体底层通信工作。由于客户端桩与服务端框架分别位于客户端与服务端程序的进程内，因此开发人员开发客户端与服务端程序时只需分别与本进程内的桩与框架构件交互即可实现与远端的交互，而负责底层通信的客户端桩与服务端框架在开发过程中自动生成而非由开发人员编写，从而为开发人员省去底层通信相关的开发工作。

在 Stub/Skeleton 结构的支撑下，客户程序与服务程序按照图中所示的 8 个步骤完成一次服务的调用：

①：客户程序将调用请求发送给客户端桩，对于客户程序来说，桩就是服务程序在客户端的代理。

②：客户端桩负责将远程调用请求进行编组并通过通信总线发送给服务端。

③：调用请求经通信总线传送到服务端框架。

④：服务端框架将调用请求解组并分派给真正的远程对象实现（服务程序）。

⑤：服务程序完成客户端的调用请求，将结果返回给服务端框架。

⑥：服务端框架将调用结果编组并通过通信总线发送给客户端桩。

⑦：客户端桩将调用结果解组并返回给客户程序。

⑧：客户程序得到调用结果。

Stub/Skeleton 结构的支持使得开发人员从繁杂的底层通信工作中解脱出来，开发人员可以不用编写底层通信代码即可实现客户端与服务端的远程交互，从而将主要精力集中到业务逻辑的实现上。

1.4.2 互操作实例

下面通过一个具体实例来演示互操作的基本原理。该例子实现远程查询通话记录的功能，客户端远程调用分布式对象实现的查询通话记录功能，分布式对象则通过查询数据库完成具体的查询功能。

该例子程序采用纯 Java 语言编写，利用 JDBC/ODBC 访问关系数据库，并采用远程方法调用 RMI 实现客户程序与服务程序之间的交互。客户程序只能通过服务程序间接地访问关系数据库，因而该例子程序属于一种典型的三层设计模型，该例子程序的三层结构与每一层实现基本功能如图 1-5 所示。

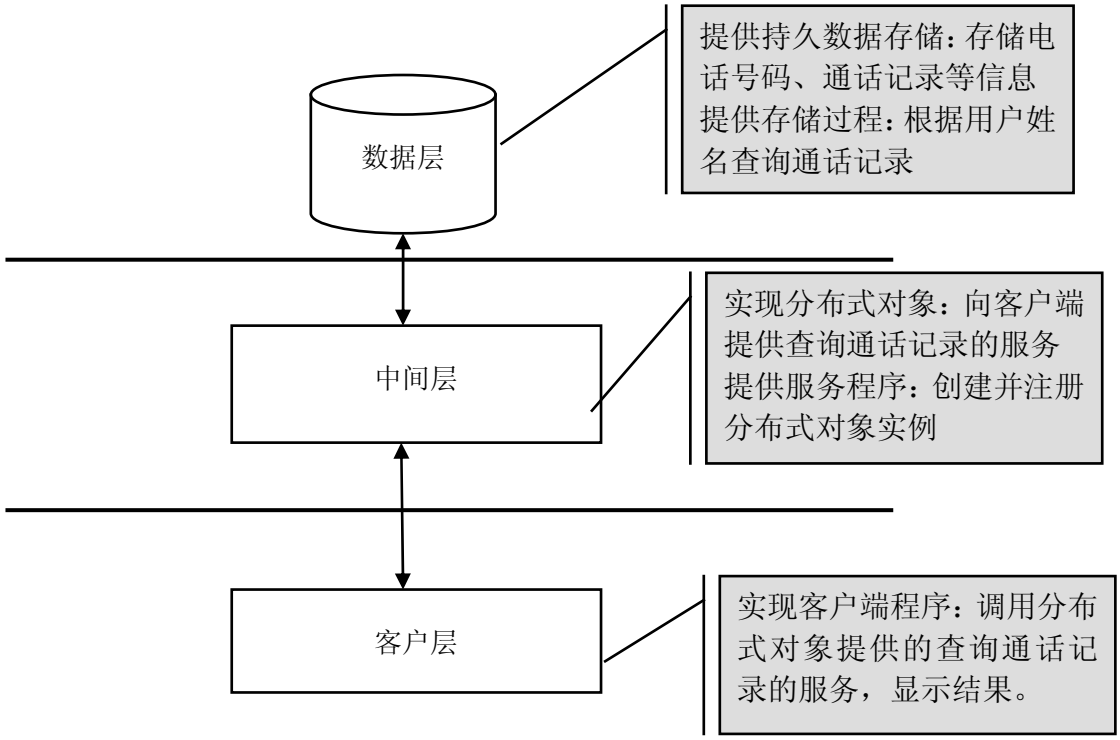


图 1-5　话费查询实例的系统结构

本例子关注的重点是客户端与中间层分布式对象之间的交互，交互基于 Java RMI 实现，Java RMI 采用 JRMP 协议支持客户程序访问远端分布式对象。

1.4.2.1 数据库设计

数据库中存放本例中使用的持久数据，即电话用户的通话记录，数据库中的表结构如图 1-6 所示。其中，表 TelephoneDirectory 记录了所有用户的电话号码，以 number 为主键；表 CallHistory 记录了所有电话的通话历史，其中 number 是外键，建立该属性的允许重复的索引。表 CallHistory 通过外键 number 与表 TelephoneDirectory 相关联。

本例中利用 JDBC/ODBC 访问数据库，因而支持使用多种不同的数据库管理系统，只要这些数据库管理系统提供了 ODBC 接口，如 Microsoft Access、Microsoft SQL Server、Sybase、Oracle 等。

数据库Telephony

表TelephoneDirectory

number *文本类型：电话号码*
subscriber *文本类型：电话用户姓名*

表CallHistory

number *文本类型：电话号码(索引,可重复)*
startTime *日期/时间类型：起始通话时间*
endTime *日期/时间类型：终止通话时间*

图 1-6　话费查询实例的数据库表结构

使用 ODBC 访问数据库之前，必须将数据库配置为 ODBC 的一个数据源。以采用 Microsoft Access 数据库为例，主要的步骤为：在 Windows 的控制面板中打开“数据源(ODBC)”，单击“添加”按钮后选择“Microsoft Access Driver (*.mdb)”驱动程序，然后为数据源命名并选择相关联的 Access 数据库文件，如图 1-7 所示。

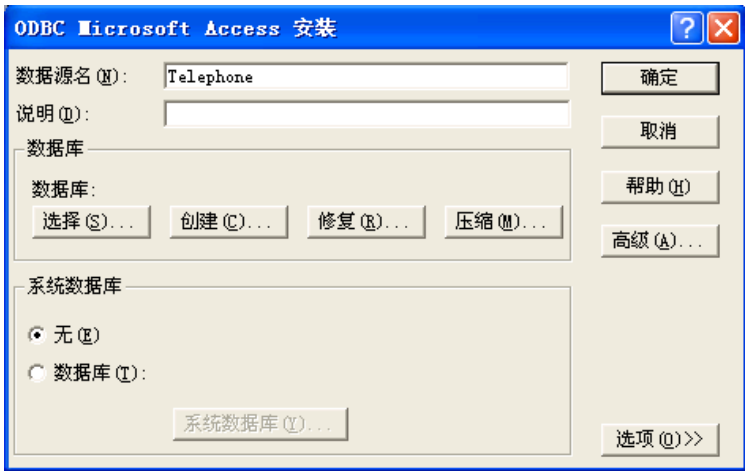


图 1-7　配置 ODBC 数据源

为进一步提高程序的数据独立性，还可以利用许多关系型数据库管理系统支持的“存储过程”来完成数据查询与更新操作。数据独立性使得在数据库中表的属性或表之间的关联发生某些变化时，程序员无需对应用程序作任何修改。使用存储过程访问数据库还可带来其他好处，例如提高了 SQL 语句查询与更新数据库的效率，在网络环境下加强了数据库的安全性等等。在不同的数据库管理系统中，存储过程可能有不同的名字，如保留过程、触发器、查询等。

图 1-8 展示了本例中使用的存储过程（即 Microsoft Access 中的查询），它根据指定电话用户名字的参数（Telephone Subscriber）查询该用户登记的所有电话的通话记录。本例程序中利用该存储过程实现对数据的查询。

查询**QueryCallHistoryWithSubscriber**

```
SELECT TelephoneDirectory.number,CallHistory.startTime, CallHistory.endTime
    FROM   TelephoneDirectory, CallHistory
    WHERE (TelephoneDirectory.number = CallHistory.number)   AND
    (TelephoneDirectory.subscriber = [Telephone Subscriber])
    ORDER   BY   startTime;
```

图 1-8　存储过程

1.4.2.2 接口定义

除了数据库中数据持久数据外，本例中需要编写的程序代码包括以下几个主要部分：

- 中间层（服务端程序）：
  - 分布式对象实现，实现中间层的核心功能，即具体查询数据库完成查询通话记录功能；
  - 服务程序：使用 Java RMI 实现分布式对象时，需要开发人员编写一个服务程序，该服务程序完成真正提供服务的分布式对象的创建与注册，服务程序中的真正提供服务的对象实例通常又称为伺服对象（servant）；
- 客户程序：利用服务程序中伺服对象提供的服务完成通话记录查询的功能。

在开发服务端程序与客户程序之前，接口是需要首先考虑的问题，因为接口是客户程序与服务端分布式对象之间的约定。在 Java RMI 中分布式对象的接口用 Java 语言的 interface 定义，并且要求所有分布式对象的远程接口必须继承 java.rmi.Remote 接口，还要求其中的每一个方法必须声明抛出 java.rmi.RemoteException 异常，因为网络通信或服务程序等原因均可能导致远程调用失败。程序 1-1 给出了例子程序中通话记录管理器的远程接口定义。

定义的接口 CallManagerInterface 约定，服务端的分布式对象提供一个可供远程调用的操作 getCallHistory，该操作需要一个字符串类型的参数（电话用户的姓名），返回一个 DatabaseTableModel 类型的对象（该用户的通话记录）。

程序 1-1　CallManagerInterface.java

```
// 分布式对象通话记录管理器 CallManager 的远程接口
package Telephone;

public interface CallManagerInterface
   extends java.rmi.Remote
{
   // 根据电话用户名字查询通话记录。
   // 参数: subscriber - 电话用户的名字
   public Database.DatabaseTableModel getCallHistory(String subscriber)
      throws java.rmi.RemoteException;
}
```

1.4.2.3 服务端程序

程序 1-2 定义的类 CallManager 就是服务端分布式对象的实现，该类实现了远程接口 CallManagerInterface 中约定的 getCallHistory 方法。为防止多个客户程序并发地调用数据库查询操作，方法 getCallHistory()被定义为同步方法。

在 CallManager 类的实现中，由于 Java RMI 中 Stub/Skeleton 结构的支持，因此在实现 getCallHistory 方法时，并没有因为该方法是被远程调用的方法而添加任何代码处理底层通

信；在下面的实现代码中，除了方法接口处声明抛出 RemoteException 外，该方法的实现与一个普通 Java 类中方法的实现没有区别。

程序 1-2　CallManager.java

```
// 通话记录管理器（即远程接口 CallManagerInterface 的实现）
package Telephone;

public class CallManager
    extends java.rmi.server.UnicastRemoteObject
    implements CallManagerInterface
{
    // 属性定义
    protected Database.DatabaseAccess database;

    // 缺省构造方法，必须抛出 RemoteException 异常。
    public CallManager()
        throws java.rmi.RemoteException
    {
        database = new Database.DatabaseAccess();
    }

    // 根据电话用户名字 subscriber 查询通话记录，实现远程接口指定的方法。
    public synchronized Database.DatabaseTableModel getCallHistory(String subscriber)
        throws java.rmi.RemoteException
    {
        String sql = "";                                // SQL 查询语句
        Database.DatabaseTableModel table = null;    // 返回的二维表模型

        System.out.println("Respond to client request: " + subscriber);
        try {
            sql = "QueryCallHistoryWithSubscriber('" + subscriber + "')";
            java.sql.ResultSet rs = database.callQuery(sql);
            table = new Database.DatabaseTableModel(rs);
            rs.close();
        } catch(java.sql.SQLException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
        return table;
    }
}
```

程序 1-3 所示 ServerApplication 是服务程序的主程序，它的核心功能是完成真正提供服务的分布式对象的创建与注册。

程序 1-3　ServerApplication.java

```
// 服务程序的主程序
public class ServerApplication
{
    final static String JDBC_DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";

    public static void main(String args[])
    {
        // 为 RMI 设置安全管理器
        System.setSecurityManager(new java.rmi.RMISecurityManager());
        // 加载 JDBC 驱动程序
        try {
            Class.forName(JDBC_DRIVER);
        } catch(ClassNotFoundException exc) {
            System.out.println(exc.getMessage());
```

```
            System.exit(1);
        }
        // 创建并注册伺服对象
        try {
            // 创建伺服对象
            Telephone.CallManager callManager = new Telephone.CallManager();
            // 用名字"CallManagerServant001"注册伺服对象
            java.rmi.Naming.rebind("CallManagerServant001", callManager);
        } catch(java.rmi.RemoteException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        } catch(java.net.MalformedURLException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
        // 提示服务程序就绪
        System.out.println("Call manager in the server is ready ...");
    }
}
```

程序 1-4 定义的 DatabaseAccess 类与程序 1-5 定义的 DatabaseTableModel 类均用于抽象 JDBC 访问数据库的行为，在实际应用中我们通常会设计更完善、更个性化的数据库访问程序包来包装 JDBC 的 API。由于本例主要关注客户程序与分布式对象之间的交互，因此对于 Java 程序中访问数据库不熟悉或不感兴趣的读者可略过这两个类的具体实现代码。

DatabaseAccess 主要用于管理服务程序与数据库的连接，并完成数据库的查询与更新操作。DatabaseTableModel 负责将数据库查询结果转换为二维表数据模型的形式，方便利用 swing 的二维表控件显示查询结果。在基于关系数据库的应用中，二维表控件是最常用的数据表达方式，对于某些具有层次结构的数据则以树控件表达会更加自然。

程序 1-4　DatabaseAccess.java

```
// 实现 JDBC 与数据库的连接
package Database;

import java.sql.*;

public class DatabaseAccess
{
    // 常量定义
    protected final String DATABASE_NAME = "jdbc:odbc:Telephone";

    // 属性定义
    protected Connection connection;            // 为数据库建立的连接
    protected Statement statement;              // 将执行的 SQL 语句
    protected CallableStatement callable;       // 将调用的 SQL 存储过程语句

    // 行为定义

    // 构造方法，建立与数据库的连接。
    public DatabaseAccess()
    {
        try {
            // 建立与指定数据库的连接
            connection = DriverManager.getConnection(DATABASE_NAME);
            // 如果连接成功则检测是否有警告信息
            SQLWarning warn = connection.getWarnings();
            while (warn != null) {
                System.out.println(warn.getMessage());
                warn = warn.getNextWarning();
```

```
            }
            // 创建一个用于执行 SQL 的语句
            statement = connection.createStatement();
            callable = null;
        } catch(SQLException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
    }

    // 析构方法，撤销与数据库的连接。
    public synchronized void finalize()
    {
        try {
            connection.close();
        } catch(SQLException exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
    }

    // 利用存储过程执行数据库查询操作。
    // 参数: procedure - 存储过程名字
    // 返回: 查询结果集
    public synchronized ResultSet callQuery(String procedure)
        throws SQLException
    {
        callable = connection.prepareCall("{call " + procedure + "}");
        ResultSet rs = callable.executeQuery();
        return rs;
    }
}
```

程序 1-5　DatabaseTableModel.java

```
// 根据数据库查询结果构造供 JTable 控件使用的二维表数据模型
package Database;

import javax.swing.*;
import java.sql.*;
import java.util.Vector;

public class DatabaseTableModel
    extends javax.swing.table.AbstractTableModel
{
    // 属性定义
    protected String[] titles;      // 列标题
    protected int[] types;          // 各列的数据类型
    protected Vector data;          // 二维表的数据

    // 构造方法，根据 SQL 查询结果集 rs 构造二维表。
    public DatabaseTableModel(ResultSet rs)
    {
        try {
            // 取得结果集的元数据
            ResultSetMetaData meta = rs.getMetaData();
            int columnCount = meta.getColumnCount();
            // 取所有列标题与类型名字（注意 JDBC 元数据的下标从 1 开始计数）
            titles = new String[columnCount];
            types = new int[columnCount];
            for (int index = 1; index <= columnCount; index++) {
```

```
            titles[index - 1] = meta.getColumnName(index);
            types[index - 1] = meta.getColumnType(index);
         }
         // 逐行取结果集中的数据
         data = new Vector(1000, 100);
         while (rs.next()) {
            Vector row = new Vector(30);
            for (int index = 1; index <= columnCount; index++)
               row.addElement(rs.getObject(index));
            row.trimToSize();
            data.addElement(row);
         }
         data.trimToSize();
      } catch(SQLException exc) {
         System.out.println(exc.getMessage());
         System.exit(1);
      }
   }

   // 实现 AbstractTableModel 遗留的抽象方法。
   public int getRowCount() {
      return data.size();
   }
   public int getColumnCount() {
      return titles.length;
   }
   public String getColumnName(int col) {
      return titles[col];
   }
   public Object getValueAt(int row, int col) {
      return ((Vector) data.elementAt(row)).elementAt(col);
   }
   public void printData() {
      int i,j;
      String ss = "";
      for(j=0; j<getColumnCount(); j++)
         ss = ss + getColumnName(j) + "\t";
      System.out.println(ss);
      for(i=0; i<getRowCount(); i++){
         ss = "";
         for(j=0; j<getColumnCount(); j++)
            ss = ss + getValueAt(i,j).toString() + "\t";
         System.out.println(ss);
      }
   }
}
```

1.4.2.4 客户端程序

本例客户程序完成的功能为调用中间层的分布式对象，并将结果显示在客户端界面上。在 Java RMI 中，客户程序通过 Java RMI 的命名服务查找希望使用的分布式对象。程序 1-6 给出的查询程序 Client.java 中，用于解析远程对象的对象标识为“CallManagerServant001”，它必须与服务程序注册远程对象时采用的名字完全相同，同时构造远程对象的 URL 时必须包含主机名。

与实现服务端分布式对象时类似，由于 Java RMI 中 Stub/Skeleton 结构的支持，因此在实现客户端程序时同样不需编写

注意程序中远程对象 callManager 必须声明为远程对象接口 CallManagerInterface 的实例，而不是远程对象实现类 CallManager 的实例，因为客户程序只能看到远端分布式对象（构件）的接口。

程序 1-6　Client.java

```
// 客户端程序
package Telephone;

public class Client
{
    // 实现 main 方法，根据参数调用分布式对象的 getCallHistory 完成通话记录的查询
    public static void main(String[] args)
    {
        String name = args.length > 0? args[0]:"songshengli";

        Database.DatabaseTableModel result = null;
        try {
            // 从 Applet 取主机名
            String host = "//localhost/";
            // 远程对象的标识必须与服务程序注册时使用的对象标识完全相同
            String objectId = "CallManagerServant001";
            // 根据主机名与对象标识解析远程对象
            CallManagerInterface callManager = (CallManagerInterface)
                  java.rmi.Naming.lookup(host + objectId);
            // 调用远程对象方法查询用户的通话记录
            result = callManager.getCallHistory(subscriber);
            System.out.println(result.toString());
        } catch(Exception exc) {
            System.out.println(exc.getMessage());
            System.exit(1);
        }
    }
}
```

1.4.2.5 编译并运行应用程序

Java 语言要求程序包与子目录相对应，因而在编写 Java 程序之前就必须决定程序包与子目录的名字。表 1-1 总结了本小节例子程序完整的目录组织与文件清单。

表 1-1　目录组织与文件清单

```
\TelephoneExample – 整个例子程序所处的子目录
    Client.java – 客户程序
    Client.class – 由 Client.java 编译得到的字节码
    ServerApplication.java – 服务程序
    ServerApplication.class - 由 ServerApplication.java 编译得到的字节码
\TelephoneExample\Database – 数据库访问程序包
    DatabaseAccess.java – 提供对数据库的连接与访问服务
    DatabaseAccess.class - 由 DatabaseAccess.java 编译得到的字节码
    DatabaseTableModel.java – 将数据库查询结果转换为二维数据表模型
    DatabaseTableModel.class - 由 DatabaseTableModel.java 编译得到的字节码
\TelephoneExample\Telephone – 电话计费查询程序包
    CallManager.java – 通话记录管理器的实现
    CallManager.class - 由 CallManager.java 编译得到的字节码
    CallManagerInterface.java – 通话记录管理器的接口
    CallManagerInterface.class - 由 CallManagerInterface.java 编译得到的字节码
    CallManager_Stub.class - 由 rmic 根据 CallManager.java 生成的客户端的桩
    CallManager_Skel.class - 由 rmic 根据 CallManager.java 生成的服务端的框架
```

表 1-1 中的 Java 字节码由 JDK 提供的 javac 编译器生成，而远程对象在客户端的桩与服务端的框架必须另由 rmic 编译器生成，rmic 的输入是 Java 字节码文件而不是 Java 源程序文件。具体的编译过程如下：

----------*编译 DatabaseAccess 类与 DatabaseTableModel 类*

D:\TelephoneExample> javac Database*.java

----------*编译 CallManagerInterface 接口与 CallManager 类*

D:\TelephoneExample> javac Telephone*.java

----------*生成客户端桩 CallManager_Stub 与服务端框架 CallManager_Skel*

D:\TelephoneExample> rmic Telephone.CallManager

----------*编译 ServerApplication 类与 Client 类*

D:\TelephoneExample> javac *.java

如果开发者不是直接使用 JDK，而是使用某种集成化开发环境（如 NetBean、Borland JBuilder、Eclipse 等），则利用项目（Project）机制可减轻许多编译工作的负担，开发环境会根据项目内容自动调用相应的编译器生成字节码文件以及客户程序桩与服务程序框架文件。

完成例子程序的编译与布置后，就可以开始运行例子程序了。首先启动 RMI 远程对象注册表，然后启动服务程序。例如在 Windows 中可在 MS-DOS 方式键入以下命令：

----------*启动 RMI 远程对象注册表*

D:\TelephoneExample> start rmiregistry

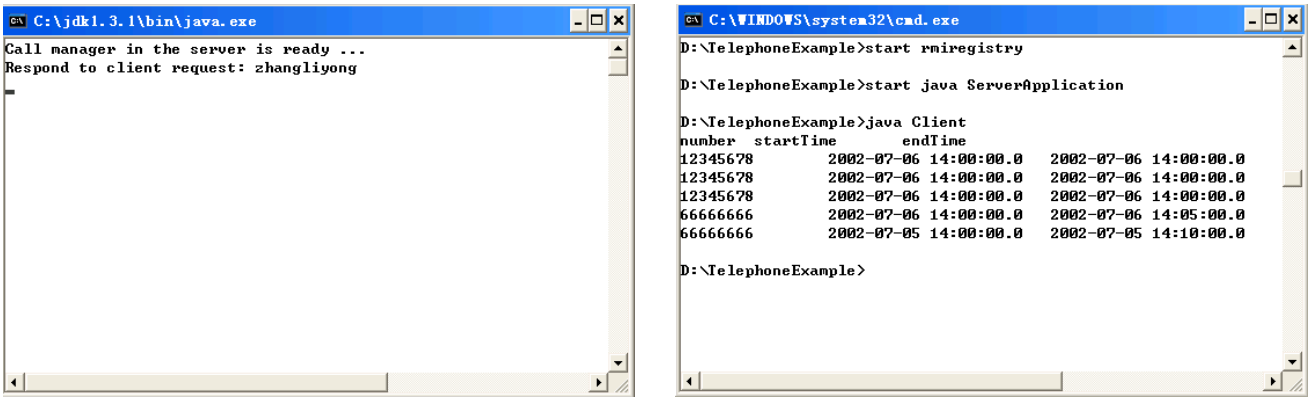
----------*启动服务程序*

D:\TelephoneExample> start java ServerApplication

----------*运行客户端程序*

D:\TelephoneExample> java Client

客户端与服务端程序的输出界面如图 1-9 所示：



a）服务端输出　　b）客户端输出

图 1-9　执行结果

1.4.2.6 分布运行

此例子程序可以很容易地将客户端与服务端分布到两个不同的物理机器上运行，由于 Java 良好的可移植性，除了 Windows 平台，客户端程序还可以运行在安装 JDK 的 LINUX、UNIX 等其它操作系统之上，由于服务端使用 JDBC/ODBC 访问数据库，因此可让其仍运行在 Windows 操作系统之上。

跨机器运行，我们需要修改客户端程序中查找分布式对象的代码，使其查找运行在服务端机器上的分布式对象，而不是本机上的分布式对象。具体的修改就是将调用命名服务的 lookup 方法时将对应的主机名改为服务程序运行的机器的主机名或 IP 地址，如程序 1-7 所示中的黑体部分所示：

程序 1-7　Client.java

```
……
   try {
      // 从 Applet 取主机名
      String host = "//serverhost/";   //改为服务程序运行机器的主机名或 IP 地址
      // 远程对象的标识必须与服务程序注册时使用的对象标识完全相同
      String objectId = "CallManagerServant001";
      // 根据主机名与对象标识解析远程对象
      CallManagerInterface callManager = (CallManagerInterface)
            java.rmi.Naming.lookup(host + objectId);
      // 调用远程对象方法查询用户的通话记录
      result = callManager.getCallHistory(subscriber);
      System.out.println(result.toString());
   } catch(Exception exc) {
      System.out.println(exc.getMessage());
      System.exit(1);
   }
```

1.4.2.7 小结

在本例中，我们实现了一个简单的三层结构的分布式应用：

数据层：存放系统的持久数据——通话记录，并提供一个存储过程以提高数据独立性。

中间层：提供分布式对象实现系统的核心功能——通话记录查询，并编写了一个服务程序将分布式对象准备好（创建并注册）。

客户层：调用分布式对象的功能，将结果呈现给用户。

在上述三层结构中，希望读者关注的重点是中间层分布式对象与客户端程序通信的实现，即基于 Java RMI 的 Stub/Skeleton 结构实现的远程交互。在上述开发过程中，rmic 命令的执行生成了负责客户端底层通信的 CallManager_Stub 与负责分布式对象底层通信的 CallManager_Skel[①]，所以尽管我们没有编写任何代码处理跨越机器的通信，我们的程序可以很容易分布到两个物理机器上运行。

在实际的复杂分布式系统中，除了分布式对象与其使用者之间的基本通信问题外，特别是当要解决的问题变得更庞大、更复杂的时候，我们往往需要寻求中间件提供的更多支持，典型的主要包括如下两个方面：

① 更复杂的可互操作性

除了跨越机器与操作系统之外，复杂的分布式系统中基于网络交互的双方经常出现跨越其它异构环境的进行交互的需求，如交互双方采用不同的程序设计语言实现等。而 Java RMI 的实现约束程序员只能使用 Java 语言，如果需要实现更复杂的互操作，我们需要寻求其它中间件的支持。

② 公共服务

大规模的分布式系统经常需要公共服务中共性的功能，如系统中分布式对象数量很多时，我们需要更有效的命名服务来查找定位分布式对象，需要更复杂的伺服对象管理机制来管理大量的分布式对象，需要资源管理机制以使得关键的系统资源能够在大量的对象中方便、高效的共享；当分布式对象的操作可能修改系统的核心业务数据时，我们需要事务控制机制以保证系统状态的一致性；当应用在网络中发布时，我们需要有效的安全控制来保证只有合法的授权用户才能执行特定的操作等等。JDK 并没有对公共服务提供更多的支持，如果需要我们也要寻求其它中间件的支持。

本书的后续内容以 CORBA 与 Java 企业版中间件为核心，重点讨论如何开发与使用符

①：在新版本的 JDK 中，读者可能会发现 rmic 只会生成一个负责客户端底层通信的 Stub 类，而不会生成负责服务端底层通信的 Skeleton 类，这主要是因为随着研究的进展，技术上已经实现通用的 Skeleton，即一份代码即可为实现所有的分布式对象实现底层通信的功能，而由于每个分布式对象对客户端提供的接口不尽相同，为保持客户端编程的简单性，每个分布式对象仍需一个 Stub 类。

合对应规范的分布式对象（构件）、如何获得更好的可互操作性以及如何使用中间件提供的公共服务等内容。

思考与练习

1-1　试描述三层分布式软件体协结构，并简要分析三层结构相比两层结构的特点和优势。

1-2　试描述分布式对象与客户端交互时采用的 Stub/Skeleton 结构，说明完成一次远程调用的基本过程。

1-3　现有集成中间件为分布式系统开发提供的基本支持有哪些？

1-4* 参照§1.4 中例子实现一个提供银行帐户操作的分布式对象和一个使用该对象的简单的客户端程序，具体要求如下：

1）该分布式对象至少提供一个查询余额的功能

2）可以将帐户信息存放在数据库中，也可以不存放在数据库中

3）客户端可以仅提供一个简单字符界面

4）**尝试跨机器和操作系统运行

第二部分　CORBA 规范与 CORBA 中间件

第 2 章　　CORBA 基本原理

本章简单 CORBA 应用程序的基本结构——对象管理体系结构、CORBA 程序通信总线 ORB 的体系结构、CORBA 对于可互操作性的支持以及 CORBA 规范与基于 CORBA 的中间件平台等内容。

§ 2.1 对象管理体系结构

2.1.1　对象管理组织与其主要规范

在学习 CORBA 之前，我们首先了解一下负责制定和发布 CORBA 规范的组织 OMG。OMG 是对象管理组织（Object Management Group）的缩写，他是 1989 年成立的非盈利性联盟。从其名字我们可以看到，该组织所做的工作是和面向对象技术密切相关的，他的主要目标是促进分布式系统开发中面向对象技术理论与实践的发展。OMG 现在有成员 800 多个，包括 IBM、Sun、HP、西门子、爱立信等大型信息产品供应商；也包括像 Oracle、BEA、Borland 等知名软件开发商；同时还包括波音、花旗等最终用户和全球众多的高校与研究机构。

OMG 的技术规范主要用来支持分布式、异类环境的软件开发项目。所谓异类是指我们所开发的软件所运行的环境可能包含不同的硬件平台（如不同厂商的 PC 机、工作站、小型机甚至 Framework）、操作系统；开发时可能使用不同的程序设计语言、可能使用不同的网络通信协议进行交互。异类环境下分布式对象与其客户端之间的可互操作性一直是分布式系统关注的一个重点。一种中间件或规范要支持如此异类的环境是很不容易的，CORBA 规范也是经过了多次更新与扩充才达到了一个比较理想的程度。

OMG 所制定和发布的规范覆盖了从分析、设计到编码、部署、运行和管理的整个软件开发过程。这些规范是一种工业或行业标准。什么意思呢？就是这些规范不是某一个或几个厂商根据自己的产品所提出的，而是在整个行业中大家共同遵循的标准。比如，我们要基于 CORBA 开发一套应用系统，我们可以自由选择不同厂商的 CORBA 产品，我们开发好的系统也可以比较容易的从一个厂商的平台上迁移到其他厂商的平台上。OMG 发布的最有影响力的两套规范，一个是 UML（Unified Modeling Language），统一建模语言；另一个就是 CORBA（Common Object Request Broker Architecture），通用对象请求代理体系结构。

UML 是一套面向对象分析与设计阶段的表示技术规范。它为我们提供了一种可视化的方式来描述一个系统的分析与设计结果。 Rational 公司的 Rational Rose 是目前较为流行的 UML 软件工具，另外还有 Borland 收购的 TogetherSoft 公司的 Together。Rational 公司和 Rose 工具相关的有一整套软件开发规范 RUP（Rational Unified Process）。

CORBA 是一套面向分布式系统的中间件规范，基于 CORBA 的中间件为应用系统提供构件管理、构件互操作、公共服务等典型支持。具体的讲，CORBA 又包含一系列单独的规范，比如核心的 ORB 体系结构、接口定义语言 IDL、网络通信协议 GIOP 和 IIOP、可移植对象适配器 POA、CORBA 组件模型 CCM 等。每一个规范都对我们开发一个分布式系统的一个侧面作了标准化。比如 IDL 就规定了我们应该如何定义分布式对象的接口。

2.1.2 对象管理体系结构

CORBA 所基于的概念框架是对象管理体系结构（Object Management Architecture，OMA），OMA 描述了一个基于 CORBA 的应用系统的基本结构与构成系统的构件的特性。由 OMG 发布的《*Object Management Architecture Guide*》是关于 OMA 的正式规范，该指南描述了 OMG 的技术目标与相关术语，并为所有 CORBA 规范提供了概念性的基础设施。指南的核心内容是对象模型与参考模型，其中对象模型定义了对象外部可见特征的、独立于具体实现的语义，参考模型则标识与刻划了组成 OMA 的组件、接口与协议。

2.1.2.1 OMA 参考模型

OMA 参考模型如图 2-1 所示，它描述了一个基于 CORBA 的应用系统的基本结构。从图中我们可以直观地看到，软件系统有很多个构件（对象）构成，这些对象都挂接到了一个类似总线的东西上；这些对象又被划分到不同的组中。图中方框加半圆的表示被封装成对象的非面向对象实现。

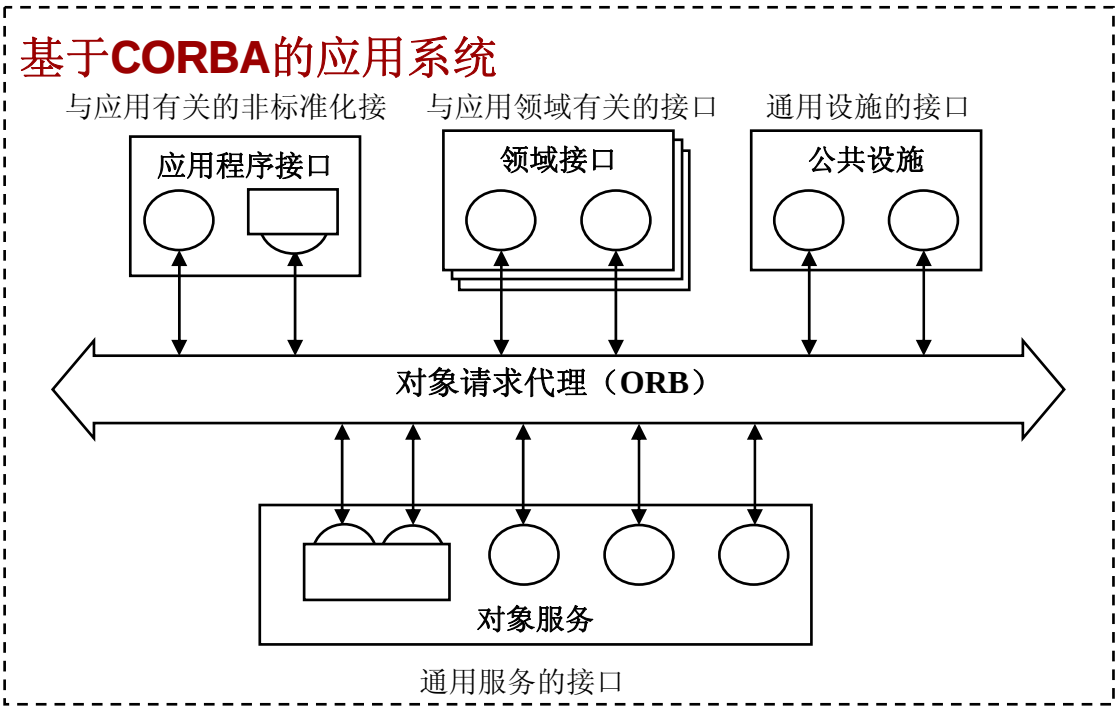


图 2-1 OMA 参考模型

在 OMA 参考模型中，ORB 为构成系统的各个构件提供了通信总线，为其提供互操作的基本支持；同时，构成系统的构件又被划分到了不同的组中，这是因为在实际的系统中除了最基本的通信设施外，构成系统的对象还有很多共性可以加以提取并重用。比如，在几乎每一个系统中都会有一些对象来实现命名服务，系统其他的对象利用该对象提供的服务来查找和定位对象；另外在很多系统中还会有一些对象实现安全性控制、一些对象实现事务控制等功能；再有在很多系统中都会有一些对象来提供打印功能、电子邮件功能等面向大多数领域的工具类的功能以及一些特定领域内通用的功能。CORBA 规范把这些对象的共性按照其基础性分别抽象并标准化为对象服务、通用设施和领域接口，从而导致了对象组的划分。

下面对 OMA 参考模型中各构成部分进行解释：

1. **对象请求代理**

对象请求代理是OMA参考模型的核心，它提供了分布式对象之间透明地发送请求或接

收响应的基本机制，独立于实现对象的特定平台与技术。客户程序无需知道如何与对象通讯、如何激活对象、对象如何实现、如何查找对象等。ORB是基于分布式对象构建应用程序的基础，并保证了在异类网络中对象的可移植性与可互操作性。

OMG的接口定义语言IDL（Interface Definition Language）为定义CORBA对象的接口提供了一种统一标准，用IDL定义的对象接口是对象实现与客户程序之间的合约。IDL是一种强类型的说明性语言，独立于任何程序设计语言。IDL到程序设计语言的映射支持开发者选择自己的程序设计语言来实现对象和发送请求。

OMG发布的*CORBA: Common Object Request Broker Architecture and Specification*是关于ORB体系结构的规范，定义了ORB组件的程序设计接口。OMG发布的*CORBA languages*是一系列独立的语言映射规范，包括Java、C++、Smalltalk、Ada、C、COBOL等语言。

2. 对象服务

对象服务是CORBA中通用性最强的一组公共服务，这些服务要么是利用分布式对象开发基于CORBA的应用程序的基础，要么为应用程序的可互操作性提供与具体应用领域无关的基础。

对象服务将覆盖对象整个生存期的对象管理任务标准化，例如对象服务提供的功能包括了创建对象、对象访问控制、查找对象、维持对象间关系等。这种标准化可导致不同应用程序的一致性，并提高软件开发者的生产率。

OMG的对象服务统称CORBA Services，OMG发布的*CORBA Services: Common Object Services Specifications*（简称COSS）是关于对象服务的一组规范，其中包括对象命名、事件、生存期、持久对象、事务、并发控制、关系、外表化、许可机制、查询、属性、安全性、时间、对象收集、交易对象等服务的规范。

3. 公共设施

公共设施的通用性比对象服务稍低，是可用于大多数应用领域的、面向终端用户的设施，包括分布式文档设施、打印设施、数据库设施、电子邮件设施等。公共设施提供的一系列通用的应用程序功能可配置为特定的应用需求，公共设施的标准化使得通用操作具有统一性，并且终端用户可方便地选择自己的配置。

OMG的公共设施统称CORBA facilities，OMG发布的*CORBA Facilities: Common Facilities Architectures*是描述公共设施体系结构的规范。与对象服务规范一样，公共设施规范包括了一系列用OMG IDL表达的接口定义。

4. 领域接口

领域接口是CORBA中通用性最低的一组公共服务，仅在特定的应用领域具备一定的通用性，是与应用领域有关的接口，例如金融、医疗、制造业、电信、电子商务、运输等应用领域，领域内通用的功能如电信领域内的电话控制功能。图2-1中的领域接口表示为一组领域的接口，暗示了开发者可按照不同的应用领域来组织领域接口。

OMG正是按不同应用领域组织与发布一系列领域接口规范，例如OMG已发布了制造、医疗、金融、电信等行业的规范集*CORBA Manufacturing*、*CORBA Med*、*CORBA Finance*、*CORBAtelecoms*等。OMG正在进一步完善并将陆续推出新的领域接口规范。

5. 应用程序接口

应用程序对象为终端用户执行该系统特定的任务，因为这部分不具有通用性，所以它不是OMG标准化的内容，而是构成整个OMA参考模型的最上层元素。一个典型的应用程序由大量基本的对象类构建而成的，其中部分对象与具体应用有关，部分对象则来自领域接口、公共设施与对象服务。应用程序对象可通过继承机制重用现有的对象。

应用程序只需支持或使用与OMG一致的接口即可加入到OMA中，这些程序本身未必要用面向对象风格来实现。图2-1的对象服务与应用程序接口展示了现有的非面向对象软件可

以嵌入在一些对象包装器中，从而融入OMA体系结构。

上述的对象服务、公共设施与领域接口构成了CORBA中的公共服务集合，OMG对于这些公共服务的接口均进行了标准化，并且以IDL的形式定义出来。厂商实现一个CORBA中间件平台的时候按照接口的约定去实现命名服务、安全性控制、事务控制、打印、电子邮件、电话控制等功能，可以采用任意的实现机制。程序员在程序中也按照接口的约定来使用这些服务（服务是由一个一个的对象提供的），而不需要关心使用的那个厂商的平台，因为它们所实现的服务都提供了相同的接口。

2.1.2.2 OMA 对象模型

OMG 在 *Object Management Architecture Guide* 中定义的对象模型描述了对象外部特征的标准语义，CORBA 系统中对象就是本书第一章描述的分布式对象——构件。

在对象模型中，对象、类型、操作、属性、对象实现等语义与 Java、C++等面向对象程序设计语言十分相近。在该模型中，客户程序与对象实现之间的界面是对象的接口定义。对象接口采用接口定义语言 IDL（Interface Definition Language）定义。

请求是一个在特定时刻发生的事件，它携带的信息包括操作、提供服务的目标对象引用、0 个或多个实际参数以及一个可选的请求上下文（request context）等。对象引用是可以有效地指称一个对象的对象名字。请求上下文提供了可能影响请求执行的额外信息，这些信息通常与操作有关。请求表（request form）用于发送请求，可多次求值与执行。请求表由 IDL 与特定语言的绑定来定义，另一种形式的请求表通过调用动态调用接口 DII 创建一个调用结构，往调用结构中添加参数后可发出调用。

在 OMG 的对象模型中，对象可以被创建或撤销。但从客户程序的角度看，没有专门机制用于创建或撤销对象，对象创建与撤销只是发出请求的结果，分布式对象的生命周期管理工作全部收缩到服务端。这样的好处是客户端的负担轻，客户端无需关心服务端分布式对象的生命周期特性，只是根据需要使用分布式对象。

§ 2.2 对象请求代理结构

对象请求代理（Object Request Broker，ORB）是 OMA 的核心基础设施，CORBA 规范规定了 ORB 的标准体系结构。ORB 负责完成查找请求的对象实现、让对象实现准备好接收请求、传递构成请求的数据等完成远程调用时底层通信任务所需的全部机制。

客户程序所看到的对象接口完全独立于对象所处的位置、实现对象的程序设计语言以及对象接口中未反映的其他特性。为调用远程对象实现的一个实例，客户程序必须首先获取一个对象引用（以后我们将知道有多种方式可获取对象引用）。客户程序发出远程调用的方式与本地调用相似，只不过调用的是远程对象实例的对象引用。ORB 检查对象引用，如何发现目标对象是远程的，就将参数打包并通过网络传递给远程对象所在的 ORB。

ORB提供的最基本功能是从客户程序向对象实现传递请求。在逻辑上ORB可理解为一个由ORB接口定义的服务集合，但在物理上ORB通常不必实现为一个单独的组件（例如进程或程序库）。ORB内核（ORB Core）是ORB最关键的部分，负责请求的通信设施，每一个ORB产品供应商都有一个自己特有的ORB内核。图2-2展示了ORB体系结构的主要组成部分以及它们之间的关系。

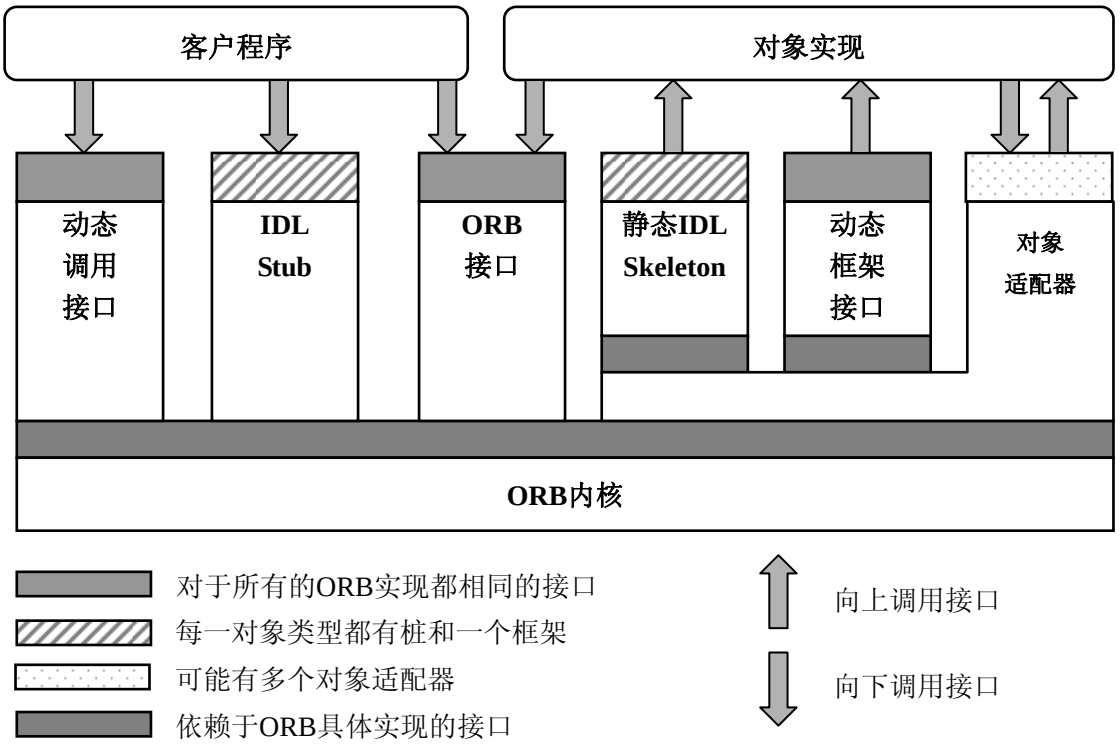


图 2-2 ORB 体系结构

从图2-2可以看到ORB采用了Stub/Skeleton结构来支持客户端与分布式对象的交互，其中客户程序桩和对象实现框架与RMI中的作用相同。但ORB的体系结构显然比RMI更复杂，因为它不仅提供了动态调用的方式，还支持用不同的程序设计语言实现对象。

1. 对象接口

客户程序所看到的对象接口完全独立于对象所处的位置、实现对象的程序设计语言以及对象接口中未反映的其他特性，这种独立性是通过 ORB 来保证的。IDL 根据可执行的操作以及操作的参数来定义对象的类型，并可映射到特定的编程语言或对象系统。为能在运行时充分利用对象接口定义的有关信息，还可将对象接口定义添加到接口库（Interface Repository）服务中，该服务允许运行时动态访问接口信息。IDL 与接口库具有同等的表达能力。

客户程序只能通过对象的接口定义掌握对象的逻辑结构，并通过发送请求来影响对象的行为与状态。客户程序不必了解对象实现的具体实现方式，也不必知道该对象实现采用哪个对象适配器以及需要用哪个 ORB 访问该对象实现。CORBA 通过对象接口进一步延伸了传统程序设计语言的封装与信息隐藏概念。

对象实现可以用多种方式实现，例如独立的服务程序、程序库、每个方法的程序代码、被封装的应用程序、面向对象数据库等。通过使用附加的对象适配器，ORB 事实上可支持所有风格的对象实现。通常对象实现不依赖于 ORB 或客户程序调用对象的方式，如果对象实现需要使用依赖于 ORB 的服务，则需要通过选择合适的对象适配器获得服务的接口。

2. ORB 中的 Stub/Skeleton 结构

客户程序通过发送请求调用由对象实现提供的服务，ORB支持客户端与分布式对象通信的最基本方式是Stub/Skeleton结构。

通过Stub/Skeleton结构完成调用的方式称为静态方式，对应客户端利用Stub发起调用的方式称为静态调用方式，服务端采用Skeleton分发请求的方式静态请求分派方式。采用静态方式时，ORB产品会提供一个IDL编译器，在编译IDL文件时会创建客户程序桩与服务端框

架。如果IDL文件进行了修改，则必须重新用IDL编译器创建新的桩与框架。由于桩与框架在编译时创建并且在运行时不再改变，因而这些接口又称作静态调用接口SII（Static Invocation Interface）。IDL桩负责客户程序的实现语言与ORB内核之间的映射，IDL框架负责服务程序的实现语言与ORB内核之间的映射，只要ORB支持某种语言的映射，客户程序或服务程序就可选择该语言作为实现语言，并且客户程序与服务程序可以使用不同的语言。使用静态接口的程序开发者必须在程序编译之前就知道操作的名字和所有参数与返回值的类型，实际的操作名字、参数值和返回值是编写在应用程序的源代码中。

3. ORB 中的动态调用方式

除了上面提到的静态方式外，ORB还支持客户端与分布式对象之间采用动态方式交互。客户端可以通过动态调用方式将请求发送给ORB内核，对应地，ORB也可以通过动态请求分派方式将请求传递给分布式对象。

客户端的动态调用方式使用动态调用接口DII（Dynamic Invocation Interface）完成。从发送请求的功能上看，动态调用方式与静态调用方式具有完全相同的能力（即两者的调用语义相同），对象实现不能知道也无需知道请求从客户端是如何发出的。DII允许客户程序调用在编译客户程序时尚未确定对象接口的对象实现。客户程序使用DII时必须生成一个请求，其中包括对象引用、操作以及参数表。DII的这种动态特性使得DII在某些应用场合更优于SII，例如编写CORBA服务的浏览器、应用程序浏览器、转换协议的桥接、访问大量不同接口、应用程序的监控、通用对象测试程序等等。使用DII的应用程序访问对象实现提供的服务时，不必包含由IDL编译器生成的桩，只需在运行时访问ANY对象。当然，使用DII会比SII更麻烦，程序员必须用DII接口指定操作和每个参数的类型与值，并且由程序员自己利用CORBA定义的类型码（typecode）作类型检查。DII类似于Java语言的自省（introspection）机制。DII还提供了一种延迟同步调用，使得客户程序提交请求后不必等候答复。延迟同步调用与单向（one-way）操作相似，但延迟同步调用支持返回值和输出型参数（这时必须进行轮询）。CORBA 3.0通过异步消息服务（Asynchronous Messaging Service）同时支持静态的和动态的延迟同步调用。

ORB将请求分派给对象实现也有两种方式：静态方式通过由IDL生成的框架，动态方式使用动态框架接口DSI（Dynamic Skeleton Interface）。除了通过静态框架外，ORB还可通过DSI查找合适的实现代码、传送参数，并将控制传给对象实现，对象实现执行请求，请求完成后控制与结果返回给客户程序。虽然DSI比较复杂且性能不高，但DSI的动态特性使得DSI在某些应用场合更优于静态分派请求，例如开发服务端的桥接（协议转换器）、监控应用程序、解释性服务或由脚本驱动的服务等，下节我们还将看到DSI在实现对象的可互操作性时发挥的作用。使用DSI的应用程序提供服务时不必包含由IDL编译创建的框架，因而这些服务支持更通用的请求，但需要更多的手工编程并且必须检查类型的安全性。

ORB屏蔽了客户端发送请求与服务端接收请求的不同方式。从客户程序的角度看，使用DSI的对象实现与使用IDL框架的对象实现行为相同，客户程序不必提供特别的处理去与使用DSI的对象实现通信；同样对于服务端来说，客户端发起请求的方式也是透明的。对象实现框架的存在并不意味着一定要有客户程序桩，客户程序也可通过DII发送请求。

4. 对象适配器

在 ORB 中，服务端对象实现的框架并没有直接与 ORB 内核交互，而是由对象适配器（Object Adapter）充当了中介，在 CORBA 服务端程序中，对象适配器负责管理服务端的分布式对象。在 ORB 结构中，分布式对象与 ORB 内核之间的通信由对象适配器完成，对象适配器负责对象引用的生成与解释、方法调用、交互的安全性、对象实现的激活与冻结、将对象引用映射到相应的对象实现、对象实现的注册等。为满足特定系统的需要，供应商会提供不同的专用对象适配器。对象实现可选择使用哪种对象适配器，这取决于对象实现所需的

服务。

§ 2.3 CORBA 中的可互操作性

对可互操作性的良好支持是 CORBA 解决的重点问题之一，也是 CORBA 的主要优势所在。除了同一平台上的可互操作性之外，当前有许多商品化和免费的 ORB 产品，每一种产品都试图满足它们的操作环境的特定需求，因而产生了不同 ORB 之间可互操作的需要。此外，有些分布式应用系统并不是 CORBA 兼容的，例如 DCE、DCOM 等，这些系统与 CORBA 的可互操作需求也在不断增长。

影响对象之间可互操作性的因素不仅仅是实现方面的差异，还包括安全性方面的严格限制或需要提供正在开发产品的受保护测试环境等原因。为了提供一个完全可互操作的环境，必须将这些差别都考虑进去。

2.3.1 ORB 域和桥接

在 CORBA 中支持互操作的基本方式是通过划分域并通过域（domain）之间的桥接来实现。域支持开发人员根据自己的实现因素或管理原因将对象划分为不同的集合，不同域的对象之间需要桥接机制才可彼此交互。用于实现互操作的桥接机制必须足够灵活，既可适用于协议翻译量少而传输性能比较重要的情况（例如位于同一个 ORB 上两个不同的域），也可适用于性能不太重要但需要访问不同 ORB 的情况。

实现可互操作性的方法通常可分为直接桥接和间接桥接两种方式。采用直接桥接方式时，需交互的元素直接转换为两个域的内部表示，这种桥接方式具有较快的传输速度，但在分布式计算中缺乏通用性。

采用间接桥接方式时，需交互的元素在域的内部表示形式与各个域一致认可的另一种表示形式之间相互转换，这种一致认可的中间表示形式要么是一种标准（例如 OMG 的 IIOP），要么是双方的私下协议。如果中间表示是某一运行环境的内部表示（如 TCP/IP），则称为全桥（full bridge）；否则如果一个 ORB 运行环境不同于公共协议，则称该 ORB 为半桥（half bridge）。

桥接既可在一个 ORB 的内部实现（例如只是连接两个管理领域的边界），也可在更高层次实现。在 ORB 内部实现的桥接称为内联桥（in-line bridge），否则称为请求层桥（request-level bridge）。实现内联桥时，既可要求 ORB 提供某种附加的服务，也可引入额外的桩和框架代码。

请求层桥的工作方式大致如下：客户程序的 ORB 将桥与服务程序的 ORB 看作对象实现的一部分，并通过 DSI 向该对象发送请求（注意 DSI 无需在编译时知道对象的规格说明）；DSI 与桥协作，将请求转换为服务程序的 ORB 能够理解的形式，并通过服务程序的 ORB 的 DII 调用被转换的结果；如果请求有返回结果，也是通过类似的路径返回。实际上桥为了完成其功能不得不了解对象的有关信息，因而要么必须访问接口库，要么只能是一个与特定接口有关的桥。

2.3.2 GIOP、IIOP 与 ESIOP

为了桥接正常工作，必须制订传输请求的统一标准，规定传输底层的数据表示方法与消息格式，由 OMG 定义的通用 ORB 间协议 GIOP（General Inter-ORB Protocol）负责完成这一功能。GIOP 专门用于满足 ORB 与 ORB 之间交互的需要，并设计成可在任何传输协议的

上层工作，只要这些协议满足最小的假设集。当然用不同传输协议实现的 GIOP 版本不必直接兼容，但它们能够交互将更加有效。

除了定义通用的传输语法外，OMG 还规定了如何以 TCP/IP 协议为基础实现 GIOP 协议，这一更具体的标准称为因特网 ORB 间协议 IIOP（Internet Inter-ORB Protocol）。GIOP 与 IIOP 之间的关系就好象 IDL 与 Java、C++等具体语言之间的映射关系。由于 TCP/IP 是独立于供应商的最流行传输协议，IIOP 为 ORB 提供了开放式的（OMG 术语为 out of the box）可互操作性。此外，IIOP 还可作为半桥的中间层，除了提供可互操作性的功能之外，供应商还可用于 ORB 间的消息传递。

IIOP 是 ORB 之间的通信协议，虽然也可用于实现 ORB 内部的消息传递，但并不是 CORBA 的硬性规定。一个特写的 ORB 产品可能支持多种通信协议，但声称与 CORBA 2.0 兼容的 ORB 产品至少必须支持 IIOP。

CORBA 规范还提供了一套特定环境 ORB 间协议 ESIOP（Environment-Specific Inter-ORB Protocols），这些协议可用于特定环境（诸如 DCE、DCOM、无线网络等系统）的可互操作性。与 IIOP 相比，ESIOP 可针对于特定环境进行优化。

2.3.3 CORBA 对可互操作性的支持

CORBA 的目标是支持多个层次的可互操作性，CORBA 规范经过多次改进与发展才达到这一目标。CORBA 支持在可互操作性主要包括如下几个层次：

1. 不同平台（如操作系统）与语言之间的可互操作性：这是早期的 CORBA 版本强调解决的主要问题，解决方法包括制定 IDL 标准以及 IDL 到程序设计语言的映射。这使得使用同一供应商的 ORB 产品开发的客户程序与服务程序之间可以交互，但使用不同供应商的 ORB 产品开发的客户程序与服务程序则未必是可互操作的。
2. 不同厂商 ORB 产品之间的可互操作性：CORBA 2.0 版引入了 GIOP 和 IIOP，从而实现了不同供应商的 ORB 产品之间的可互操作性，所有供应商的 ORB 产品如果与 CORBA 2.0 兼容则彼此之间可互操作。
3. 不同体系结构之间的可互操作性：更完善的可互操作性还应包括不同体系结构之间的可互操作，例如一个 CORBA 对象可通过协议桥接操作一个 DCOM 对象，OMG 通过引入 ESIOP 来解决这一问题。但 ESIOP 只能解决 CORBA 与特定体系结构（如 DCOM）之间的互操作，并不能通过一套 ESIOP 解决所有的问题。

§ 2.4 CORBA 规范与 CORBA 产品

OMG 本身不开发或发布任何软件或实现任何规范，它只是将 OMG 成员的信息需求（RFI）与建议需求（RFP）汇集为规范。CORBA 规范是一套开放式的规范，OMG 的成员或非成员公司均可免费实现符合 CORBA 规范的 ORB 产品。

2.4.1 CORBA 规范

CORBA 这一名词既用于专指关于 ORB 体系结构的规范，也泛指 OMG 基于 OMA 参考模型发布的一系列规范集。由于 OMG 需要不断改进与完善 CORBA 体系结构，因而为 CORBA 规范编制了完整的版本号。仅当对体系结构作重大改变时，OMG 才增加 CORBA 规范的主版本号，所以有时也以主版本号代指 CORBA 的新特征，例如 CORBA 2 指 ORB 之间的可互操作性和基于 TCP/IP 协议的 IIOP，而正式发布最新版本 CORBA 3 则指 CORBA 的组件模型。

为更好掌握 CORBA 的改进历程与发展趋势，并理解 ORB 产品供应商对 CORBA 规范各种新特性的支持，我们有必要了解 CORBA 规范各主要版本的发布时间和主要改进内容。

1. 1991 年 12 月正式发布 CORBA 1.1

定义了接口定义语言 IDL 的标准以及 ORB 的应用程序设计接口（API），使客户程序与对象实现可在 ORB 的具体实现中彼此交互。

2. 1995 年 7 月正式发布 CORBA 2.0

定义了不同供应商的 ORB 之间的可互操作性。1997 年 9 月修订的 CORBA 2.1 增加了 CORBA 与 Microsoft 的分布式计算模型 COM 的可互操作性。1998 年 2 月修订的 CORBA 2.2 引入可移植对象适配器 POA 取代原有的基本对象适配器 BOA，并增加了 IDL 到 Java 语言的映射标准。OMA 参考模型中的领域接口也是从该版本开始引入，表明 CORBA 规范已从 ORB 内部运行方式扩展到 CORBA 技术应用。1998 年 12 月修订了 CORBA 2.3，本书采用的实验环境 VisiBroker for Java 4.0 完全遵循了该版本。

2000 年 11 月修订的 CORBA 2.4.1 是 OMG 正式发布 CORBA 规范的最新版本。CORBA 2.4 新增了三个规范：CORBA 消息规范（CORBA Messaging）由服务质量（QoS）、异步方法调用和可互操作的路由接口组成，异步方法调用使 CORBA 的消息传递方式包括了同步、延迟同步、单向和异步四种方式，服务质量可用于根据应用需求管理与选择不同的底层传输方式；极小化 CORBA 规范（MinimumCORBA）将 CORBA 裁剪为适合仅有有限资源的系统；实时 CORBA 规范（Real-Time CORBA）对 CORBA 进行扩充，将 ORB 作为实时系统的一个部件。此外，该版本还对可互操作的命名服务、通知服务等规范进行了修改。

3. 2002 年 8 月正式发布 CORBA 3.0

鉴于组件技术在面向对象技术中扮演着越来越重要的角色，3.0 版本的最大改进是为 CORBA 引入了 CORBA 组件模型（CORBA Component Mode,CCM）。CORBA 组件模型参照 Sun Microsystems 的 EJB（Enterprise JavaBeans），为开发即插即用的 CORBA 对象提供了基本架构，程序员可用 CORBA 脚本语言 CSL（CORBA Scripting Language）合成 CORBA 组件。CORBA 组件模型将会给客户端和服务端的可伸缩性带来有力支持。此外，CORBA 3.0 更好地集成了 Java、因特网和 DCE 遗留系统，允许在 IIOP 上使用 RMI，并支持 IIOP 穿越防火墙。

2.4.2 CORBA 产品

尽管 OMG 不断改进与完善 CORBA 规范，但每一版本保持了较好的向后兼容性，因而 CORBA 规范相当成熟与稳定，并且拥有大量产品，在企业计算与因特网计算领域拥有庞大的市场。基于 CORBA 的软件适用于因特网应用与企业计算，特殊版本的 CORBA 还可运行在实时系统、嵌入式系统与容错系统。

由于 CORBA 规范独立于软件供应商，在不同供应商的 ORB 产品上运行的程序之间具有较好的可互操作性。当前市场上有许多 CORBA 产品可供用户选择，其中一些是商品化的，一些是免费的，许多商品化商品也提供了免费的试用期。但这些产品在对 CORBA 规范的支持程度、对 CORBA 对象服务的支持程度、其他附加特征等方面有很大差异。

1. Orbix

IONA 技术公司的 Orbix 是一个可靠的商品化 CORBA 产品，其主要版本是 Orbix 6，分为标准版（Orbix Standard）、企业版（Orbix Enterprise）和大型主机版（Orbix Mainframe）三个版本，早期广泛使用的版本包括 Orbix 3 等。Orbix 完全遵循 CORBA 规范，支持 Java、

C++、COBOL、PL/I 等多种语言，可运行在 Windows、Linux、Solaris 和 HP-UX 等多种平台，实现了 CORBA 与 COM 的桥接，并提供命名、安全性、事务、交易对象、事件、通知等多种对象服务。

全球的相当一部分电话呼叫都是通过 IONA 的企业 CORBA 产品处理的。全球最大的金融机构的三个总部都在其最关键的 IT 系统中使用了 Orbix。Orbix 为全球规模最大要求最高的面向服务的体系结构 (SOA) 提供了基础结构。

Orbix 是 IONA 利用基于标准的解决方案解决高端企业集成问题的主要手段之一，在一些全球最大规模的集成系统中，在考虑到极高的性能、可用性、安全性和系统管理以及一流的 24x7 技术支持时，很多企业选择了基于 Orbix 实现。

2. VisiBroker

Borland 公司是当前商品化 ORB 产品的主要供应商之一，其产品 VisiBroker 目前支持支持 Java、C++以及 C#等语言，可在 Windows、Linux、Solaris、HP-UX 和 AIX 等平台运行，目前最新版本为 7.0。新版本 VisiBroker 完全遵循 CORBA 2.6 规范，同时支持实时 CORBA 规范，支持 SOA 架构与.NET 应用的互操作。作为一个 ORB 平台，VisiBroker 支持可移植对象适配器、值类型、RMI/IIOP、具有集群和容错特性的命名服务、属性管理、服务质量、拦截器等，并扩充了位置服务、对象包装器等功能。

VisiBroker 的应用非常广泛，除了作为独立的 ORB 平台外，VisiBroker 还其短小精悍而被嵌入在其他产品之中，例如 Netscape Communicator 浏览器嵌入了 VisiBroker 产品后，Applet 可向 CORBA 对象发出请求而不必须下载相关的 ORB 类。

3. TAO

TAO 是由美国华盛顿大学分布式对象计算研究小组开发的著名免费 ORB 产品，采用代码公开开发模式。TAO 支持 C++语言，提供命名、事件、交易对象、生存期、属性、并发性等对象服务。TAO 目前支持 CORBA2.6，其特色是包括了支持实时 CORBA 的显式绑定与可移植同步器。

4. OmniORB

OmniORB 是由 AT&T 剑桥实验室开发的一个免费 ORB 产品，该产品的第 4 版完全遵循 CORBA 2.6 规范，支持 C++，提供命名、属性等对象服务，主要特色是具有较高的性能。OmniORB 自 1997 年开始成为 GNU 公开许可证（GNU Public Licence）的免费软件。

5. ORBit

ORBit 是一个遵循 CORBA 2.4 规范的 ORB 产品，支持 C 语言、C++、Lisp、Python、Ruby 等语言绑定。运行 POA、DII、SII 等特性。

关于这些产品的详细信息可浏览表 2-1 提供的站点。

表 2-1 部分商品化 CORBA 产品及其站点

IONA Orbix	http://www.iona.com/products/orbix/
Borland VisiBroker	http://www.borland.com/us/products/visibroker/
TAO	http://www.cs.wustl.edu/~schmidt/TAO.html
omniORB	http://omniorb.sourceforge.net
ORBit	http://orbit-resource.sourceforge.net

2.4.3 CORBA 小结

1. CORBA 的优势

CORBA 提供了一个工业标准而不是一个软件产品，不同供应商的竞争导致市场上存在大量高质量的、完全遵循 CORBA 的产品。共同遵循的 CORBA 标准为应用系统提供了较好的可移植性（注意应用程序在不同 CORBA 产品之间并不是 100%可移植）。

CORBA 的最大特点在于提供了在异类分布式环境中对象之间高度的可互操作性，这种高度可互操作性使得运行于异类环境的分布式对象之间可以方便地通信，异类环境包括不同供应商的不同硬件平台、操作系统、网络系统、程序设计语言或其他特性。CORBA 应用程序客户可运行在小至手持无线设备或嵌入式系统，大至大型计算机的平台。CORBA 支持多种现有的程序设计语言，并且支持在单个分布式应用程序中混合使用这些语言。这一特性导致了 CORBA 在诸如电信领域等典型领域内得到了非常广泛的应用，电信领域内部的异构问题非常突出，而 CORBA 对于可互操作性提供了良好的支持，因此 CORBA 技术在电信智能网、电信网管系统中应用很广泛。

CORBA 的另一优势是提供了灵活的服务端模型（将在第 5 章详细讨论）与丰富的系统级服务，这一特性的典型应用是开发有效管理大量服务端对象的大型系统，CORBA 的可伸缩性与容错性使得许多大型网站后台都应用了 CORBA 技术。

由于 CORBA 是相对早期提出的规范，因此新的规范在提出时借鉴了很多 CORBA 规范的成功之处，我们可以在 Java 企业版等中间件规范中看到 CORBA 成功技术的影子。

2. CORBA 的问题

作为一种支持分布式系统开发的中间件规范，由于规范制定与修定的周期较长（再加上厂商实现的周期情况会更糟）等诸多因素，CORBA 在一些现代分布式系统新需求方面并没有及时提供足够的支持，尤其是现在基于 Web 的分布式系统开发需要的人机界面（如动态页面技术等）支持、持久化支持、更强大的构件自动化管理支持（CORBA3.0 中提出的 CORBA 组件模型 CCM 具备这类特性，但提出较晚，因此厂商实现的进度很难满足实际应用的需求）等，这使得基于 CORBA 开发分布式系统时有时不能获得全面的支持，尤其在开发典型的以数据库为核心（database-backened）的 Web 信息系统时，这种情况尤为突出，这在某种程度是限制了 CORBA 的进一步应用，相比之下，Java 企业版与.NET 对于这类系统的支持更为全面。

综上所述，按照作者的观点，CORBA 目前比较适合对于互操作要求较高、基于异类环境的分布式系统开发，这类系统的核心特征是系统核心**业务逻辑构件之间**（而不是界面构件与业务逻辑构件之间或业务逻辑构件与数据库之间）的频繁跨越网络交互是系统需要解决的主要问题之一；另外 CORBA 灵活的服务端模型与丰富的系统级服务也使得 CORBA 中间件成为支撑分布式系统中间层开发的有力候选者之一。相比之下，由于在人机界面支持、持久化支持等方面的欠缺，CORBA 中间件在支撑以数据库为核心的 Web 信息系统开发时显得支撑不够全面，而 Java 企业版或.NET 中间件的优势更为明显。

思考与练习

2-1 结合 OMA 参考模型说明 CORBA 规范如何向应用系统提供互操作与公共服务的支持？

2-2 请说明 CORBA 规范对可互操作性支持具体情况，并针对可互操作性的每个侧面简要解释 CORBA 规范是如何做到的。

2-3* 结合 CORBA 规范的主要特点，说明 CORBA 中间件适合什么类型的分布式系统开发。

第 3 章　基于 CORBA 的开发过程

本章通过一个基于 Borland VisiBroker（4.5.1）的简单例子说明基于 CORBA 的基本开发过程，读者可以从网上免费获得其带试用期限的试用版。

§ 3.1 设计相关的若干问题

首先，我们简单看一下在系统设计阶段应注意的问题。由于篇幅的限制，此外并不想去考察软件系统分析设计阶段的普遍问题，而是重点讨论几个和基于 CORBA 的分布式系统相关的问题。

1. 运行平台

设计者必须在设计初期必须决定待开发的分布式系统要运行在哪类硬件和软件平台之上。由于不同平台（计算机硬件和操作系统）之间的差异，为一个平台开发的软件系统通常不能直接运行在另一个平台之上（读者应注意尽管 CORBA 提供了异类环境中良好的可互操作性，但这与系统的可移植性是截然不同的两个问题）。一般来说，设计者总是在软件系统的性能与通用性的矛盾之间作一个折衷的选择：要使所开发的软件系统具备良好的通用性，能够方便的在不同平台上迁移，就需要尽可能的避免使用特定平台（比如某特厂商的平台或某个操作系统）相关的机制，而较好的性能很多情况下都要借助与特定平台的特性才能获得。所以设计者要根据自己特定项目或系统的特点与需求作出一个折衷的选择。。

2. 调用方式

在 CORBA 中，分布式对象提供的服务的调用方式可有三种：

- 同步方式：调用时调用者会阻塞直到被调用的服务完成并返回。
- 异步方式：调用者发起调用后不会阻塞，等待服务完成期间可以执行其它操作，调用者通过轮询方式或服务者发送的事件检测调用完成，服务完成后调用者检查并处理结果。异步方式通常依靠异步消息来实现。
- 单向方式：调用者只是发出调用请求，并不关心调用什么时候完成（以及完成的结果）。

不同的调用方式适合不同的应用场合。客户程序的请求所引起的服务程序操作只需要很短时间即可完成，例如查询某一帐户的当前余额，这时应选用同步通信方式。如果客户程序请求服务程序格式化并打印几个大型文档，服务程序需要较长时间才可完成该请求，这时应选用异步通信方式，从而在服务程序格式化并打印文档期间，客户程序可以做一些其他事情。如果客户程序无需获知请求已完成的确认信息，例如向系统日志模块登记系统执行了某一操作，则应选用单向通信方式。

3. 资源优化

在分布式环境下，跨网络的通信开销是相当可观的，通常是影响系统整体效率的瓶颈，而在 Stub/Skeleton 机制的支撑下，开发者已经不需要自己编程处理底层通信，分布式对象在开发时并没有表现出很强的公布特性，这更容易使设计者忽略跨网络的通信对系统的影响。在一个集中式软件系统中，程序员可在同一进程中随意地连续多次调用一个例程，因为这些调用的系统资源开销微不足道。但在分布式环境下，同样的调用如果发生在跨网络的进程之间，这些调用占用的系统资源是相当可观的，因此在设计阶段，特别是在接口详细设计阶段，应考虑尽量提高网络通信资源的利用率，避免频繁的跨网络（尤其是广域网）通信，而不应只从功能实现的方面去考虑。

4. 其他决策问题

分布式系统通常要比集中式软件考虑更多的安全性、可靠性、事务处理、并发控制等问题。另外需要考虑更多的错误处理，例如客户程序发出请求但服务程序未就绪，甚至找不到服务程序或无权限访问服务程序时，应如何处理这种情况。

§ 3.2 CORBA 应用程序开发过程

虽然 OMG 为 CORBA 制订了统一的规范，但规范中也赋予了软件供应商实现 ORB 产品时自由选择各自不同的实现途径的权利，例如 ORB 可以是一个独立运行的守护进程，也可以嵌入到客户程序和对象实现中，有些 ORB 产品则选择了几种实现方式的组合。所以不同供应商提供的 ORB 产品在具体使用方法上可能存在较大差异，本书 CORBA 部分以 Borland 公司的 VisiBroker for Java 4.5.1 为例，介绍一个 CORBA 应用程序的具体开发步骤，使用其他 ORB 产品时可参照类似做法。

尽管使用不同 ORB 产品的具体操作差异较大，但程序员开发一个 CORBA 应用程序通常会遵循一定的框架，即首先通过面向对象分析与设计过程认定应用程序所需的对象，包括对象的属性、行为与约束等特性，然后遵循本节所述的几个开发步骤完成应用程序的开发、部署与运行，如图 3-1 所示，图中的箭头表示了任务之间的先后次序。

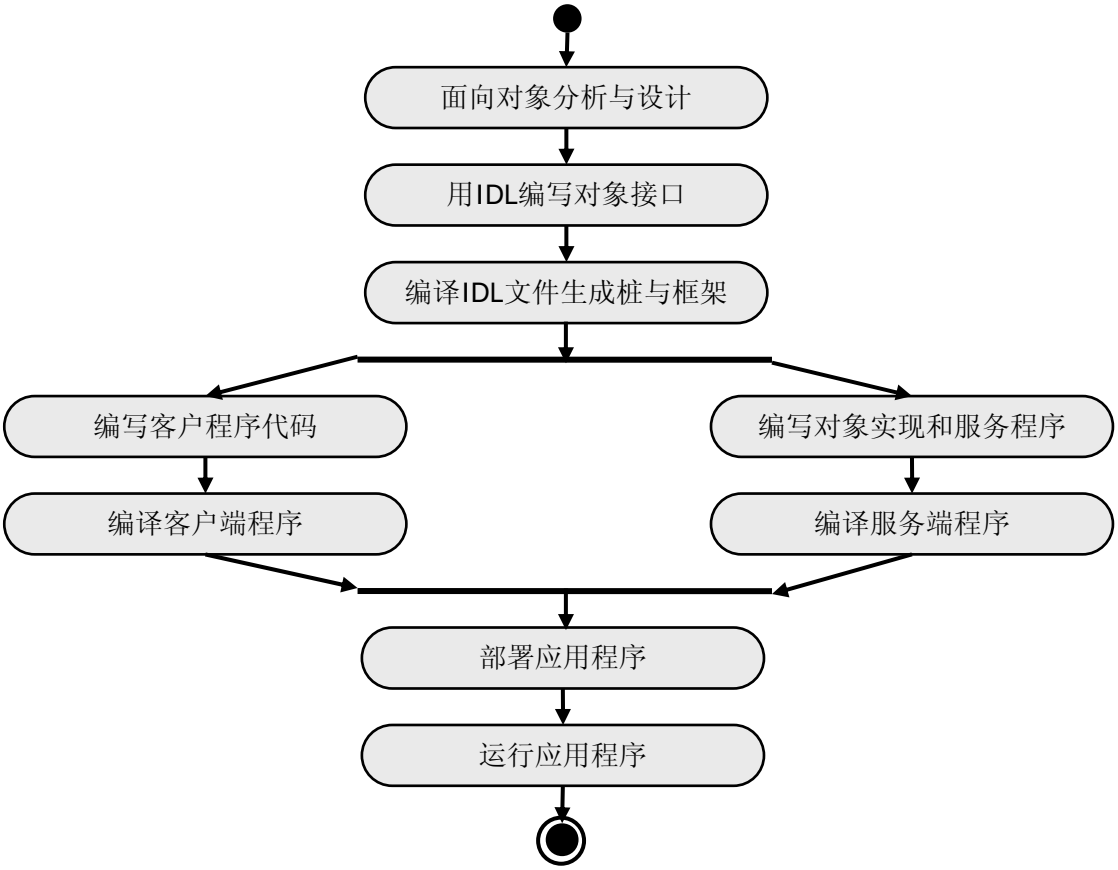


图 3-1 CORBA 应用的典型开发过程

下面简要解释开发的每一步骤的主要工作：

1. 编写对象接口

对象接口是分布式对象对外提供的服务的规格说明。CORBA 中分布式对象的接口定义

中包括以下内容：

- 提供或使用的服务的名字：即客户端可以调用的方法名；
- 每个方法的参数列表与返回值；
- 方法可能会引发的异常；
- 可选的上下文环境：调用相关的上下文环境，上下文环境起和参数类似的作用，包含调用相关的若干信息，只不过它并不写死在参数列表中，有更强的灵活性。

可以看出，CORBA 中对象接口中定义的核心内容和 RMI 例子中用 Java interface 定义的接口类似，但它们有一点明显区别，那就是 CORBA 中对象接口是由 OMG IDL 定义的，而这种 IDL 是独立于程序设计语言的，正是有了这种中性的接口约定，才使得分布式对象和客户端的跨语言得以实现。本书第四章详细介绍了 IDL 的语法与语义。

2. 编译 IDL 文件

IDL 是一种独立于具体程序设计语言的说明性语言，IDL 编译器的作用是将 IDL 映射到具体程序设计语言，产生客户程序使用的桩代码以及编写对象实现所需的框架代码。由 OMG 制订的语言映射规范允许将 IDL 语言映射到 Java、C++、Ada、C、COBOL 等多种程序设计语言，这通常是由软件供应商提供的不同编译器分别完成的。

一般厂商实现 CORBA 平台时都会提供专门的 IDL 编译器来完成 IDL 接口的编译工作。厂商实现 IDL 编译器时应参照 OMG 制订的语言的规范，编程人员只要选择使用合适的编译器就可以了，IDL 编译器的工作原理如图 3-2 所示。

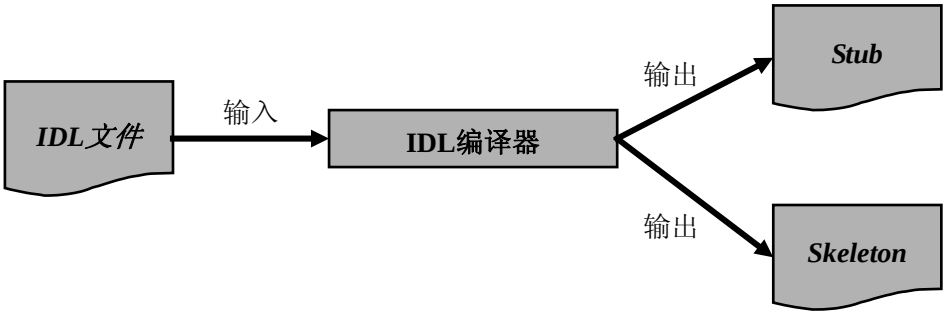


图 3-2 IDL 编译器的工作原理

VisiBroker for Java 提供的 idl2java 编译器将 IDL 映射到 Java 语言，生成 Java 语言的客户端桩代码以及服务端框架代码，桩和框架可看作分别是服务对象在客户端和服务端的代理。IDL 文件严格地定义了客户程序与对象实现之间的接口，因而客户端的桩可以与服务端的框架协调工作，即使这两端是用不同的程序设计语言编译，或运行在不同供应商的 ORB 产品之上。

3. 编写客户程序

在 CORBA 中，客户程序的流程较为简单，如图 3-3 所示，首先初始化 ORB，然后绑定到要使用的服务对象，然后调用服务对象提供的服务。和 Java RMI 相比，CORBA 客户端程序多了一步初始化 ORB 的操作。

在 CORBA 中，无论是客户程序还是服务程序，都必须在利用 ORB 进行通信之前初始化 ORB。初始化 ORB 的作用有两个，一个让 ORB 了解有新的成员加入，以便后继为其提供服务；另一个作用就是获取 ORB 伪对象的引用，以备将来调用 ORB 内核提供的操作。所谓伪对象专指在 CORBA 基础设施中的一个对象，比如 ORB 本身可以看作一个伪对象，伪对象的这个“伪”字主要针对在程序中远程访问的 CORBA 分布式对象而言，因为伪对象是本地的，我们通过伪对象调用 CORBA 基础设施提供的操作。

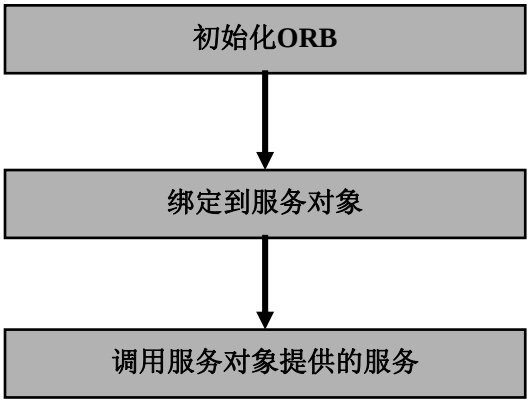


图 3-3 客户程序的工作流程

ORB 内核提供了一些不依赖于任何对象适配器的操作，这些操作可由客户程序或对象实现调用，包括获取初始引用的操作、动态调用相关的操作、生成类型码的操作、线程和策略相关的操作等，初始化 ORB 即是其中的一个操作。程序员在程序中通过 ORB 伪对象来调用这些操作，也就是说，在程序中通过使用“<orb 伪对象名>.<方法名>”的方式来调用 ORB 内核提供的操作。

4. 编写对象实现和服务程序代码

在 CORBA 服务端，开发的主要工作包括编写对象实现与服务程序。

IDL 文件只定义了服务对象的规格说明，程序员必须另外编写服务对象的具体实现。CORBA 规定所有对象接口定义必须统一用 IDL 书写，但对象实现则有很多选择的余地，例如可使用 Java、C++、C、Smalltalk 等程序设计语言，并且选择这些语言与客户程序所选用的语言无关，只要 ORB 产品供应商支持 IDL 到这些语言的映射即可。

选用任何一门程序设计语言的程序员应该熟悉 IDL 到该语言的映射规则，因为通常情况下，IDL 编译器除了生成 Stub 与 Skeleton 之外，还会生成一些开发时需要使用辅助代码。例如 VisiBroker for Java 的 IDL 编译器将自动生成一些对象适配器的 Java 类和各种辅助性的 Java 类，编写对象实现的代码时必须继承其中的一些类或使用某些类提供的方法。

与 Java RMI 例子中类似，在 CORBA 服务端，除了编写对象实现代码后，还要编写一个服务程序来将分布式对象准备好。服务程序利用可移植对象适配器（POA）激活伺服对象供客户程序使用。服务程序通常是一个循环执行的进程，不断监听客户程序请求并为之服务。

5. 创建并部署应用程序

编写完代码以后，就可以编译生成目标应用程序了。

创建客户程序时，应将程序员编写的客户程序代码与 IDL 编译器自动生成的客户程序桩代码一起编译；创建服务程序时，应将程序员编写的对象实现代码与 IDL 编译器自动生成的服务程序框架代码一起编译。一些 ORB 产品提供了专门的编译器以简化这一过程，例如 VisiBroker for Java 提供的编译器 vbjc 会自动调用 JDK 中的 Java 编译器 javac，指示 javac 在编译客户程序的同时编译相关的客户程序桩文件，在编译服务程序的同时编译相关的服务程序框架文件。

程序员创建的客户程序和服务应用程序已通过测试并准备投入运行后，进入应用程序的部署（deployment）阶段。分布式系统的布署工作通常远比集中式软件的安装复杂，在该阶段由系统管理员规划如何在终端用户的桌面系统安装客户程序，或在服务器一类的机器上安装服务程序。由于其复杂性，布署工作经常由单独的角色来承担。

6. 运行应用程序

运行 CORBA 应用程序时，必须首先启动服务程序，然后才可运行客户程序。其他步骤

可能与具体 ORB 产品有关，例如 VisiBroker for Java 的 ORB 内核是一个名为 osagent 的独立运行进程（又称智能代理，smart agent），可以在启动服务程序之后才启动 osagent，但必须在运行客户程序之前让 osagent 启动完毕。

读者应注意，上述过程一个典型的 CORBA 应用程序开发过程，具体实施时各个步骤会由于不同项目的各自特点而有所区别。例如利用 CORBA 集成企业原有系统时，就是一种先有对象实现，而后有对象接口规格说明的过程。

§ 3.3 CORBA 开发实例

本节以一个银行帐户管理的简单例子演示 CORBA 应用程序的典型开发过程，使读者对 CORBA 应用程序的开发、部署与运行有一个初步的感性认识。

这是一个简单的银行帐户管理程序，服务端管理大量银行顾客的账户，向远程客户端提供基本的开户、存款、取款、查询余额的功能。

3.3.1 认定分布式对象

按照面向对象的设计理念，我们很容易认定出系统中应包含图 3-4 所示的两类对象，Account 是一个银行帐户的实体模型，它有一个属性 balance 表示当前的余额，另有三个行为分别为存款 deposit、取款 withdraw 和查询余额 getBalance，每一银行帐户在生存期的任何时刻都满足帐户余额不小于 0 这一约束。由于例子程序不是仅仅管理某一位顾客的帐户，而是涉及到大量的帐户需要处理，所以我们还建立了“帐户管理员”这一实体模型，它负责对每一个帐户的开设、撤销和访问等，在实现世界中该实体对应着银行中的储蓄员。帐户管理员 AccountManager 有一个属性记录当前已开设的所有帐户，并且有一个行为表示根据帐户标识查找某一帐户，如果该标识的帐户不存在则创建一个新帐户，我们将该行为命名为 open。这其实只是帐户管理员的最简化模型，在一个实际应用系统中会赋予帐户管理员这一实体更多的职能。

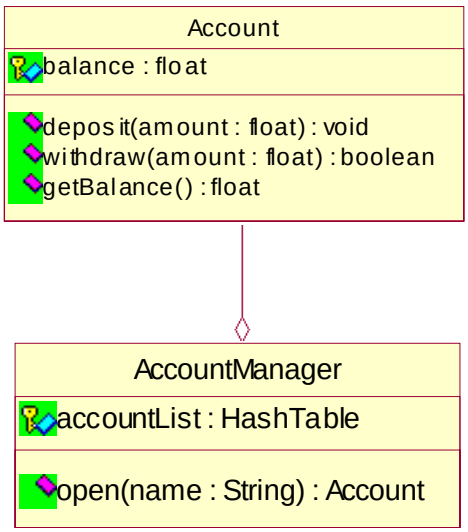


图 3-4 系统对象

读者应注意，上述两类对象是分布式对象，因为对象上的 open、deposit、withdraw 与 getBalance 操作都是要被客户端远程调用的。

3.3.2 编写分布式对象的接口

按照图 3-1 所示的开发过程，对于每一个分布式对象，首先要做得就是定义其接口。我们利用 OMG 的接口定义语言 IDL 编写对象 Account 和 AccountManager 的规格说明，规格说明存放在一个文本文件中，称之为 IDL 文件。从程序 3-1 所示 IDL 文件中的对象接口定义可看出，IDL 具有与 Java 中的 interface 具有 类似的语法。本书第四章将详细介绍如何使用 IDL 定义模块、接口、数据结构等。

程序 3-1 Bank.idl 文件中定义的对象接口

```
// 银行帐户管理系统的对象接口定义
module Bank {
    // 帐户
    interface Account {
        // 存款
        void deposit(in float amount);
        // 取款
        boolean withdraw(in float amount);
        // 查询余额
        float getBalance();
    };

    // 帐户管理员
    interface AccountManager {
        // 查询指定名字的帐户，查无则新开帐户
        Account open(in string name);
    };
};
```

3.3.3 编译 IDL 文件生成桩与框架

完成对象接口的规格说明后，下一步工作是利用 VisiBroker for Java 提供的 idl2java 编译器根据 IDL 文件生成客户程序的桩代码以及对象实现的框架代码。客户程序用这些 Java 桩代码调用所有的远程方法，框架代码则与程序员编写的代码一起创建对象实现。

上述 Bank.idl 文件无需作特殊处理，它可用以下命令编译：

prompt> idl2java Bank.idl

由于 Java 语言规定每一个文件只能定义一个公有的接口或类，因此 IDL 编译器的输出会生成多个.java 文件。这些文件存储在一个新建的子目录 Bank 中，Bank 是 IDL 文件中指定的模块名字，也是根据该模块中所有 IDL 接口自动生成的所有 Java 类所属的程序包。

IDL 编译器为 IDL 文件中定义的每一个接口自动生成 7 个.java 文件，故上述 IDL 文件的编译结果会生成 14 个.java 文件，下面以 Account 接口对应的 7 个文件为例介绍 IDL 编译器生成的代码，IDL 编译器为接口 Account 生成的文件包括：

- AccountOperations.java
- Account.java
- _AccountStub.java
- AccountPOA.java
- AccountPOATie.java
- AccountHelper.java
- AccountHolder.java

在这些文件中，Account.java 和 AccountOperations.java 定义了 IDL 接口 Account 的完整基调（signature，包括操作的名字和参数表，可用于唯一地表示一个操作的类型）。

AccountOperations.java 是 IDL 文件中由 Account 接口定义的所有方法和常量的基调声

明，如程序 3-2 所示。由 IDL 编译器自动生成的对象实现框架代码将实现该接口（参见程序 3-5），该接口还与 AccountPOATie 类一起提供纽带机制（参见程序 3-6）。

程序 3-2　IDL 编译器生成的 AccountOperations.java 文件内容

```
package Bank;

public interface AccountOperations
{
    public void deposit(float amount);
    public boolean withdraw(float amount);
    public float getBalance();
}
```

IDL 编译器为每一个 IDL 接口生成一个最基本的 Java 接口，例如 Account.java 包含了 Account 接口的声明，如程序 3-3 所示。该接口继承了 AccountOperations 接口。在程序员编写的客户程序中，会使用该接口来调用远程账户对象上的操作。

程序 3-3　IDL 编译器生成的 Account.java 文件内容

```
package Bank;

public interface Account
    extends com.inprise.vbroker.CORBA.Object,
            Bank.AccountOperations,
            org.omg.CORBA.portable.IDLEntity
{}
```

IDL 编译器还为每一个接口创建一个桩类，_AccountStub.java 是 Account 对象在客户端的桩代码，它实现了 Account 接口，如程序 3-4 所示。读者应注意此类中 deposit、withdraw、getBanlance 等操作并没有真正实现账户对象的业务逻辑，只是替客户端完成服务端真正业务逻辑实现的调用。该类负责客户端调用账户对象时的底层通信工作，从客户程序的代码上（程序 3-9）看，客户端通过 Account 接口来调用账户对象上的操作，但实际上客户端调用的是该类上的操作，由该类替客户端完成远程调用。

程序 3-4　IDL 编译器生成的_AccountStub.java 文件内容

```
package Bank;

public class _AccountStub
    extends com.inprise.vbroker.CORBA.portable.ObjectImpl
    implements Account
{
    final public static java.lang.Class _opsClass = Bank.AccountOperations.class;
    private static java.lang.String[] __ids = {"IDL:Bank/Account:1.0"};

    public java.lang.String[] _ids()
    {
        return __ids;
    }

    public void deposit(float amount)
    {
        while (true) {
            if (!_is_local()) {
                org.omg.CORBA.portable.OutputStream _output = null;
                org.omg.CORBA.portable.InputStream  _input  = null;
                try {
                    _output = this._request("deposit", true);
```

```
                _output.write_float((float) amount);
                _input = this._invoke(_output);
            } catch(org.omg.CORBA.portable.ApplicationException _exception) {
                final org.omg.CORBA.portable.InputStream in =
                    _exception.getInputStream();
                java.lang.String _exception_id = _exception.getId();
                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception: "
                    + _exception_id);
            } catch(org.omg.CORBA.portable.RemarshalException _exception) {
                continue;
            } finally {
                this._releaseReply(_input);
            }
        } else {
            final org.omg.CORBA.portable.ServantObject _so =
                _servant_preinvoke("deposit", _opsClass);
            if (_so == null) {
                continue;
            }
            final Bank.AccountOperations _self =
                (Bank.AccountOperations)_so.servant;
            try {
                _self.deposit(amount);
            } finally {
                _servant_postinvoke(_so);
            }
        }
        break;
    }
}
public boolean withdraw(float amount)
{
    while (true) {
        if (!_is_local()) {
            org.omg.CORBA.portable.OutputStream _output = null;
            org.omg.CORBA.portable.InputStream  _input  = null;
            boolean _result;
            try {
                _output = this._request("withdraw", true);
                _output.write_float((float)amount);
                _input = this._invoke(_output);
                _result = _input.read_boolean();
                return _result;
            } catch(org.omg.CORBA.portable.ApplicationException _exception) {
                final org.omg.CORBA.portable.InputStream in =
                    _exception.getInputStream();
                java.lang.String _exception_id = _exception.getId();
                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception: "
                    + _exception_id);
            } catch(org.omg.CORBA.portable.RemarshalException _exception) {
                continue;
            } finally {
                this._releaseReply(_input);
            }
        } else {
            final org.omg.CORBA.portable.ServantObject _so =
                _servant_preinvoke("withdraw", _opsClass);
            if (_so == null) {
                continue;
            }
            final Bank.AccountOperations _self =
                (Bank.AccountOperations)_so.servant;
```

```
            try {
                return _self.withdraw(amount);
            } finally {
                _servant_postinvoke(_so);
            }
        }
    }
}

public float getBalance()
{
    while (true) {
        if (!_is_local()) {
            org.omg.CORBA.portable.OutputStream _output = null;
            org.omg.CORBA.portable.InputStream  _input  = null;
            float _result;
            try {
                _output = this._request("getBalance", true);
                _input = this._invoke(_output);
                _result = _input.read_float();
                return _result;
            } catch(org.omg.CORBA.portable.ApplicationException _exception) {
                final org.omg.CORBA.portable.InputStream in =
                    _exception.getInputStream();
                java.lang.String _exception_id = _exception.getId();
                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception: "
                    + _exception_id);
            } catch(org.omg.CORBA.portable.RemarshalException _exception) {
                continue;
            } finally {
                this._releaseReply(_input);
            }
        } else {
            final org.omg.CORBA.portable.ServantObject _so =
                _servant_preinvoke("getBalance", _opsClass);
            if (_so == null) {
                continue;
            }
            final Bank.AccountOperations _self =
                (Bank.AccountOperations)_so.servant;
            try {
                return _self.getBalance();
            } finally {
                _servant_postinvoke(_so);
            }
        }
    }
}
}
```

AccountPOA.java 是 Account 对象的服务端框架代码，如程序 3-5 所示。该类一方面负责解包 in 类型的参数并将参数传递给对象实现，另一方面负责打包返回值与所有 out 类型的参数。所谓打包（marshal）是指将特定程序设计语言描述的数据类型转换为 CORBA 的 IIOP 流格式，经过网络传输到目标地点后再通过解包（unmarshal）从 IIOP 流格式转换为依赖于具体程序设计语言的数据结构。

编写对象实现的最简单途径是继承这些 POA 类，即把它们作为对象实现的基类（参见程序 3-12）。

程序 3-5　IDL 编译器生成的 AccountPOA.java 文件内容

```
package Bank;

public abstract class AccountPOA
    extends org.omg.PortableServer.Servant
    implements org.omg.CORBA.portable.InvokeHandler, Bank.AccountOperations
{
    public Bank.Account _this()
    {
        return Bank.AccountHelper.narrow(super._this_object());
    }

    public Bank.Account _this(org.omg.CORBA.ORB orb)
    {
        return Bank.AccountHelper.narrow(super._this_object(orb));
    }

    public java.lang.String[] _all_interfaces(
        final org.omg.PortableServer.POA poa, final byte[] objectId)
    {
        return __ids;
    }

    private static java.lang.String[] __ids = {"IDL:Bank/Account:1.0"};
    private static java.util.Dictionary _methods = new java.util.Hashtable();
    static {
        _methods.put("deposit",
            new com.inprise.vbroker.CORBA.portable.MethodPointer(0, 0));
        _methods.put("withdraw",
            new com.inprise.vbroker.CORBA.portable.MethodPointer(0, 1));
        _methods.put("getBalance",
            new com.inprise.vbroker.CORBA.portable.MethodPointer(0, 2));
    }

    public org.omg.CORBA.portable.OutputStream _invoke(java.lang.String opName,
        org.omg.CORBA.portable.InputStream _input,
        org.omg.CORBA.portable.ResponseHandler handler)
    {
        com.inprise.vbroker.CORBA.portable.MethodPointer method =
            (com.inprise.vbroker.CORBA.portable.MethodPointer) _methods.get(opName);
        if (method == null) {
            throw new org.omg.CORBA.BAD_OPERATION();
        }
        switch (method.interface_id) {
            case 0: {
                return Bank.AccountPOA._invoke(this, method.method_id, _input,
                    handler);
            }
        }
        throw new org.omg.CORBA.BAD_OPERATION();
    }

    public static org.omg.CORBA.portable.OutputStream _invoke(
        Bank.AccountOperations _self, int _method_id,
        org.omg.CORBA.portable.InputStream _input,
        org.omg.CORBA.portable.ResponseHandler _handler)
    {
        org.omg.CORBA.portable.OutputStream _output = null;
        {
            switch (_method_id) {
                case 0: {
                    float amount;
                    amount = _input.read_float();
```

```
                _self.deposit(amount);
                _output = _handler.createReply();
                return _output;
            }
            case 1: {
                float amount;
                amount = _input.read_float();
                boolean _result = _self.withdraw(amount);
                _output = _handler.createReply();
                _output.write_boolean((boolean)_result);
                return _output;
            }
            case 2: {
                float _result = _self.getBalance();
                _output = _handler.createReply();
                _output.write_float((float)_result);
                return _output;
            }
        }
        throw new org.omg.CORBA.BAD_OPERATION();
    }
}
```

AccountPOATie.java 用于采用纽带机制实现服务端的 Account 对象，其内容如程序 3-6 所示。由于 Java 语言中类只支持单重继承，因此有时对象实现可能由于需要继承其它类而导致其无法继承 POA 类，这里需要利用该纽带机制类来共同支持客户端的远程调用。本书第五章将详细介绍纽带机制。

程序 3-6　IDL 编译器生成的 AccountPOATie.java 文件内容

```
package Bank;

public class AccountPOATie
    extends AccountPOA
{
    private Bank.AccountOperations _delegate;
    private org.omg.PortableServer.POA _poa;

    public AccountPOATie(final Bank.AccountOperations _delegate)
    {
        this._delegate = _delegate;
    }

    public AccountPOATie(final Bank.AccountOperations _delegate,
        final org.omg.PortableServer.POA _poa)
    {
        this._delegate = _delegate;
        this._poa = _poa;
    }

    public Bank.AccountOperations _delegate()
    {
        return this._delegate;
    }

    public void _delegate(final Bank.AccountOperations delegate)
    {
        this._delegate = delegate;
    }
```

```
    public org.omg.PortableServer.POA _default_POA()
    {
        if (_poa != null) {
            return _poa;
        } else {
            return super._default_POA();
        }
    }

    public void deposit(float amount)
    {
        this._delegate.deposit(amount);
    }

    public boolean withdraw(float amount)
    {
        return this._delegate.withdraw(amount);
    }

    public float getBalance()
    {
        return this._delegate.getBalance();
    }
}
```

IDL 编译器为每一个用户自定义类型还生成一个辅助工具类。AccountHelper.java 定义了 AccountHelper 类，该类为 Account 接口定义了许多实用功能和支持功能的静态方法（又称类方法），如程序 3-7 所示。

这些工具性的方法为用户自定义类型的对象提供了以下主要操作：从 Any 对象提取或向 Any 对象插入对象（extract 和 insert 方法）；从输入 / 输出流读写对象（read 和 write 方法）；获取对象的库标识和类型码（id 和 type 方法）；绑定对象与类型转换操作（bind 和 narrow 方法）等等。

由 IDL 编译器自动生成的 Java 类以及程序员编写的客户程序与对象实现代码都直接使用了 AccountHelper 类提供的类方法，例如几种重载形式的 bind 方法用于将客户程序绑定到指定对象标识与主机的 Account 类型对象，narrow 方法则将通用类型对象窄化为 Account 类型。

程序 3-7　IDL 编译器生成的 AccountHelper.java 文件内容

```
package Bank;

public final class AccountHelper
{
    public static Bank.Account narrow(final org.omg.CORBA.Object obj)
    {
        return narrow(obj, false);
    }

    public static Bank.Account unchecked_narrow(org.omg.CORBA.Object obj)
    {
        return narrow(obj, true);
    }

    private static Bank.Account narrow(final org.omg.CORBA.Object obj,
        final boolean is_a)
    {
        if (obj == null) {
            return null;
```

```
        }
        if (obj instanceof Bank.Account) {
            return (Bank.Account)obj;
        }
        if (is_a || obj._is_a(id())) {
            final org.omg.CORBA.portable.ObjectImpl _obj =
                (org.omg.CORBA.portable.ObjectImpl)obj;
            Bank._AccountStub result = new Bank._AccountStub();
            final org.omg.CORBA.portable.Delegate _delegate =
                _obj._get_delegate();
            result._set_delegate(_delegate);
            return result;
        }
        throw new org.omg.CORBA.BAD_PARAM();
    }

    public static Bank.Account bind(org.omg.CORBA.ORB orb)
    {
        return bind(orb, null, null, null);
    }

    public static Bank.Account bind(org.omg.CORBA.ORB orb, java.lang.String name)
    {
        return bind(orb, name, null, null);
    }

    public static Bank.Account bind(org.omg.CORBA.ORB orb, java.lang.String name,
        java.lang.String host, com.inprise.vbroker.CORBA.BindOptions _options)
    {
        if (!(orb instanceof com.inprise.vbroker.CORBA.ORB)) {
            throw new org.omg.CORBA.BAD_PARAM();
        }
        return narrow(((com.inprise.vbroker.CORBA.ORB) orb).bind(
            id(), name, host, _options), true);
    }

    public static Bank.Account bind(org.omg.CORBA.ORB orb,
        java.lang.String fullPoaName, byte[] oid)
    {
        return bind(orb, fullPoaName, oid, null, null);
    }

    public static Bank.Account bind(org.omg.CORBA.ORB orb,
        java.lang.String fullPoaName, byte[] oid, java.lang.String host,
        com.inprise.vbroker.CORBA.BindOptions _options)
    {
        if (!(orb instanceof com.inprise.vbroker.CORBA.ORB)) {
            throw new org.omg.CORBA.BAD_PARAM();
        }
        return narrow(((com.inprise.vbroker.CORBA.ORB) orb).bind(
            fullPoaName, oid, host, _options), true);
    }

    public java.lang.Object read_Object(
        final org.omg.CORBA.portable.InputStream istream)
    {
        return read(istream);
    }

    public void write_Object(final org.omg.CORBA.portable.OutputStream ostream,
        final java.lang.Object obj)
    {
```

```
        if (!(obj instanceof Bank.Account)) {
            throw new org.omg.CORBA.BAD_PARAM();
        }
        write(ostream, (Bank.Account)obj);
    }

    public java.lang.String get_id()
    {
        return id();
    }

    public org.omg.CORBA.TypeCode get_type()
    {
        return type();
    }

    private static org.omg.CORBA.TypeCode _type;
    private static boolean _initializing;

    private static org.omg.CORBA.ORB _orb()
    {
        return org.omg.CORBA.ORB.init();
    }

    public static Bank.Account read(
        final org.omg.CORBA.portable.InputStream _input)
    {
        return narrow(_input.read_Object(_AccountStub.class), true);
    }

    public static void write(final org.omg.CORBA.portable.OutputStream _output,
        final Bank.Account _vis_value)
    {
        if (!(_output instanceof org.omg.CORBA_2_3.portable.OutputStream)) {
            throw new org.omg.CORBA.BAD_PARAM();
        }
        if (_vis_value != null &&
             !(_vis_value instanceof org.omg.CORBA.portable.ObjectImpl))
        {
            throw new org.omg.CORBA.BAD_PARAM();
        }
        _output.write_Object((org.omg.CORBA.Object)_vis_value);
    }

    public static void insert(final org.omg.CORBA.Any any,
        final Bank.Account _vis_value)
    {
        any.insert_Object((org.omg.CORBA.Object)_vis_value,
            Bank.AccountHelper.type());
    }

    public static Bank.Account extract(final org.omg.CORBA.Any any)
    {
        Bank.Account _vis_value;
        final org.omg.CORBA.Object _obj = any.extract_Object();
        _vis_value = Bank.AccountHelper.narrow(_obj);
        return _vis_value;
    }

    public static org.omg.CORBA.TypeCode type()
    {
        if (_type == null) {
```

```
            synchronized (org.omg.CORBA.TypeCode.class) {
                _type = _orb().create_interface_tc(id(), "Account");
            }
        }
        return _type;
    }

    public static java.lang.String id()
    {
        return "IDL:Bank/Account:1.0";
    }
}
```

AccountHolder.java 声明的 AccountHolder 类为传递 Account 对象提供支持，其内容如程序 3-8 所示。在下一章我们将了解到 IDL 有三种参数传递方式：in、out 和 inout，调用方法时 in 类型的参数以及返回结果与 Java 的参数传递方式与结果返回方式完全相同，但 out 和 inout 两种类型的参数允许参数具有返回结果的能力，无法直接映射到 Java 语言的参数传递机制，这时 AccountHolder 类为传递 out 和 inout 参数提供了一个托架（holder）。

程序 3-8　IDL 编译器生成的 AccountHolder.java 文件内容

```
package Bank;

public final class AccountHolder
    implements org.omg.CORBA.portable.Streamable
{
    public Bank.Account value;

    public AccountHolder() {}

    public AccountHolder(final Bank.Account _vis_value)
    {
        this.value = _vis_value;
    }

    public void _read(final org.omg.CORBA.portable.InputStream input)
    {
        value = Bank.AccountHelper.read(input);
    }

    public void _write(final org.omg.CORBA.portable.OutputStream output)
    {
        Bank.AccountHelper.write(output, value);
    }

    public org.omg.CORBA.TypeCode _type()
    {
        return Bank.AccountHelper.type();
    }
}
```

3.3.4　编写对象实现

对象实现代码所在的类名字可由程序员自由掌握，只要不与 IDL 编译器自动产生的 Java 类产生名字冲突即可，例如我们为例子程序中的两个对象实现分别命名为 AccountImpl 和 AccountManagerImpl。客户程序也无须了解对象实现是由哪一个 Java 类完成的。

CORBA 应用程序的对象实现最常用、最简单的实现方式是使用继承，即由程序员编写的对象实现 AccountImpl 类直接继承了由 IDL 编译器生成的 AccountPOA 类。但是 Java 语

言仅支持类的单继承，如果对象实现 AccountImpl 继承了 AccountPOA 类就无法再继承其他类，当对象实现需要利用继承机制达到其他目的时，就必须改用上面提到的 CORBA 对象实现的另一种实现方式 —— 纽带机制（tie mechanism）。

我们的例子程序采用简单的继承方式编写对象实现。帐户的对象实现如程序 3-9 所示，AccountImpl 类继承抽象类 AccountPOA，并实现了 AccountPOA 从 AccountOperations 接口遗留的 3 个方法，成为一个可创建对象实例的具体类。

程序 3-9　对象实现 AccountImpl.java

```
// 帐户的具体实现

public class AccountImpl
    extends Bank.AccountPOA
{
    // 属性定义
    protected float balance;

    // 构造方法，按指定余额创建新的帐户
    public AccountImpl(float bal)
    {
        balance = bal;
    }

    // 往帐户中存款
    public void deposit(float amount)
    {
        balance += amount;
    }

    // 从帐户中取款，不足余额则返回 false
    public boolean withdraw(float amount)
    {
        if (balance < amount)  return false;
        else {
            balance -= amount;
            return true;
        }
    }

    // 查询帐户余额
    public float getBalance()
    {
        return balance;
    }
}
```

帐户管理员的对象实现如程序 3-10 所示，AccountManagerImpl 类继承了抽象类 AccountManagerPOA，并实现了 AccountManagerPOA 的所有遗留方法。由于在本例中账户管理员用来管理另一类分布式对象——账户，账户管理员的核心功能是将新创建的账户分布式对象准备好（代码中 open 方法的黑体部分），因此其功能与后面的服务程序非常类似（服务程序的主要作用是把账户管理员分布式对象准备好），读者可以首先学习程序 3-11 中的服务端程序，然后再对比学习 AccountManagerImpl 类的实现代码。

程序 3-10　对象实现 AccountManagerImpl.java

```
// 帐户管理员的具体实现
```

```
import java.util.*;
import org.omg.PortableServer.*;

public class AccountManagerImpl
    extends Bank.AccountManagerPOA
{
    // 属性的定义
    protected Hashtable accountList;       // 该帐户管理员所负责的帐户清单

    // 构造方法，管理员开始时管理的帐户清单为空
    public AccountManagerImpl()
    {
        accountList = new Hashtable();
    }

    // 查找指定名字的帐户，找不到则以该名字新开一个帐户
    public synchronized Bank.Account open(String name)
    {
        // 在帐户清单中查找指定名字的帐户
        Bank.Account account = (Bank.Account) accountList.get(name);
        // 如果不存在则新创建一个
        if (account == null) {
            // 随机虚构帐户的初始余额，金额在 0 至 1000 之间
            Random random = new Random();
            float balance = Math.abs(random.nextInt()) % 100000 / 100f;
            // 按指定余额创建帐户的伺服对象
            AccountImpl accountServant = new AccountImpl(balance);
            try {
                // 用缺省的 POA 激活伺服对象，这里缺省的 POA 就是根 POA
                org.omg.CORBA.Object obj =
                    _default_POA().servant_to_reference(accountServant);
                // 将对象引用收窄为帐户类型
                account = Bank.AccountHelper.narrow(obj);
            } catch(Exception exc) {
                exc.printStackTrace();
            }
            // 将帐户保存到帐户清单中
            accountList.put(name, account);
            // 在服务端控制台打印已创建新帐户的提示信息
            System.out.println("新开帐户: " + name);
        }
        // 返回找到的帐户或新开设的帐户
        return account;
    }
}
```

3.3.5 编写服务程序

通常程序员都会编写一个名为 Server.java 的服务程序，如程序 3-11 所示。典型的服务程序通常执行以下步骤：

(1) 初始化对象请求代理（ORB）。

(2) 用所需策略创建一个可移植对象适配器（POA）。程序中的 BankPOA 是我们为 POA 起的名字，客户程序必须使用相同的 POA 名字。

(3) 创建一个提供服务的伺服对象。伺服对象是服务端的本地对象，当伺服对象通过对象适配器注册到 ORB 后，就成为一个可供远程调用的 CORBA 对象。

(4) 激活新创建的伺服对象，即利用 POA 将伺服对象以一个字符串表示的标识注册到 ORB 上。

(5) 激活 POA 管理器。

(6) 等待客户程序发来请求。

关于 POA、POA 策略以及 POA 管理器，我们将在第五章作详细介绍。

程序 3-11　服务程序 Server.java

```
// 服务端的主程序
import org.omg.PortableServer.*;

public class Server
{
   public static void main(String[] args)
   {
      try {
         // 初始化 ORB
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // 取根 POA 的引用
         POA rootPOA = POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
         // 创建持久 POA 的策略
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };
         // 用新定义的策略创建 myPOA
         POA myPOA = rootPOA.create_POA("BankPOA",
            rootPOA.the_POAManager(), policies);
         // 创建伺服对象
         AccountManagerImpl managerServant = new AccountManagerImpl();
         // 在 myPOA 上用标识"BankManager"激活伺服对象
         myPOA.activate_object_with_id(
            "BankManager".getBytes(), managerServant);
         // 激活 POA 管理器
         rootPOA.the_POAManager().activate();
         // 等待处理客户程序的请求
         System.out.println("帐户管理员 BankManager 已就绪 ...\n");
         orb.run();
      } catch(Exception exc) {
         exc.printStackTrace();
      }
   }
}
```

3.3.6　编写客户程序

客户程序 Client.java 演示如何查询一个银行帐户的当前余额，然后进行存款和取款操作后，查看帐户的最新余额，如程序 3-9 所示，

程序 3-9　客户程序 Client.java

```
// 客户端的主程序

public class Client
{
   public static void main(String[] args)
   {
      // 初始化 ORB
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      // 利用 POA 全称与对象标识"BankManager"查找帐户管理员
      Bank.AccountManager manager = Bank.AccountManagerHelper.bind(
         orb, "/BankPOA", "BankManager".getBytes());
      // 如果命令行参数未指帐户名字则使用缺省名字
      String name = args.length > 0 ? args[0] : "David Zeng";
      // 请求帐户管理员找出一个指定名字的帐户，无此帐户则新开一个
```

```
        Bank.Account account = manager.open(name);
        // 取该帐户的余额
        System.out.println(name + "的帐户余额为" + account.getBalance() + "元");
        // 往帐户中存款 200 元后，重新查看余额
        account.deposit(200);
        System.out.println("存款 200 元后，余额为" + account.getBalance() + "元");
        // 从帐户中取款 600 元后，重新查看余额
        if (account.withdraw(600)) {
            System.out.println("取款 600 元后，余额为" + account.getBalance() + "元");
        } else {
            System.out.println("余额不足 600 元，取款失败，余额保持不变");
        }
    }
}
```

如前所述，IDL 编译器根据 Bank.idl 文件自动生成的许多 Java 类存放在子目录 Bank 中，客户程序会直接利用其中的某些类或接口。例子程序中的客户程序主要执行了以下几个步骤：

(1) 初始化 ORB。利用 org.omg.CORBA.ORB 提供的 init 方法可获取 ORB 伪对象的一个对象引用。

(2) 绑定到一个 AccountManager 对象。从这一例子程序我们可看到了两种典型的对象引用获取方式。第一种是在应用程序刚启动时，没有任何服务对象的引用，这时通常利用 IDL 编译器生成的 Helper 类中提供的类方法 bind，将对象标识解析为对象引用。bind 方法请求 ORB 查找对象实现并与之建立连接，如果成功找到对象实现并建立了连接，则创建一个 AccountManagerPOA 对象作为对象实现的代理，然后将 AccountManager 对象的对象引用返回给客户程序。关于 Helper 类的 bind 方法可参阅程序 3-7。

(3) 通过调用 AccountManager 对象上的 open 方法获取一个 Account 对象。这是获取对象引用更常见的方式，如果已存在一个远程 CORBA 对象的引用，则可以利用该对象引用调用某一方法返回新的对象引用。例如 AccountManager 的 open 方法可根据指定的顾客名字获取一个 Account 对象的引用。

(4) 通过调用 Account 对象上的方法完成指定帐户的存款、取款或查询余额操作。只要客户程序获取了一个 Account 对象的引用，就可调用 deposit、withdraw 和 getBalance 方法对该帐户对象进行存款、取款和查询余额操作。注意客户端的这三个方法实际上是由 idl2java 编译器生成的桩代码，它们收集一个请求所需要的所有数据，并将请求通过 ORB 发送到真正执行操作的对象实现。

3.3.7 编译应用程序

综上所述，我们共开发了 1 个 IDL 文件和 4 个 Java 类，如表 3-1 所示。经过 IDL 编译器 idl2java 处理 Bank.idl 文件后，会创建一个名为 Bank 的子目录，其中含有 14 个.java 文件。

表 3-1 例子程序的源文件清单

```
\BankExample – 整个例子程序所处的子目录
    Bank.idl – 帐户和帐户管理员的对象接口定义
    AccountImpl.java – 帐户的对象实现
    AccountManagerImpl.java – 帐户管理员的对象实现
    Client.java – 客户端主程序
    Server.java – 服务端主程序
```

下一步是调用 Java 语言的编译器生成可运行的客户程序与服务程序的字节码，我们利用 VisiBroker for Java 提供的编译器 vbjc 完成这一工作：

```
prompt> vbjc Server.java
```

prompt> vbjc Client.java

vbjc 实际上封装了 JDK 提供的 Java 编译器，它调用 javac 编译源文件并将与源文件有关的其他文件也编译成字节码，从而避免了我们手工逐个地编译。注意由 IDL 编译器生成的 Holder 类和 POATie 类需要指定额外的参数才会生成，因而上述编译过程完成后 Bank 子目录中只会新增 10 个.class 文件。

3.3.8 运行应用程序

完成上述编译过程后，就准备好了运行我们的第一个 CORBA 应用程序。

1. 启动智能代理

运行 CORBA 应用程序之前，网络中必须至少有一台主机上启动了智能代理 osagent。这是 VisiBroker 特有的分布式位置服务（location service）守护进程，网络中多个智能代理可协作以查找合适的对象实现。启动智能代理的基本命令如下：

prompt> osagent

如果运行平台是 Windows NT，还可将智能代理作为 NT 服务来启动。这需要在安装 VisiBroker for Java 时将 ORB 服务注册为 NT 服务，注册服务后就可通过服务控制面板将智能代理作为一个 NT 服务启动。

2. 启动服务程序

打开一个 MS DOS 方式的命令行窗口，使用以下命令启动服务程序：

prompt> start vbj Server

vbj 实际上封装了 Java 虚拟机 java，它设置一些属性与类路径后，启动 java 运行目标程序。为接着利用当前命令行窗口启动客户程序，我们用 start 命令衍生一个新进程运行服务程序。

3. 运行客户程序

在命令行状态使用以下命令启动客户程序：

prompt> vbj Client

如果我们连续两次启动客户程序且不带命令行参数，则访问的是同一个缺省顾客名字的帐户。第一次运行时将开设一个新的帐户，并且帐户的初始余额是随机设置的。每开设一个新帐户时，服务程序的控制台会给出一行提示信息，例如：

```
帐户管理员 BankManager 已就绪 ...

新开帐户: David Zeng
```

第二次运行客户程序时，由于该顾客的帐户已存在，所以查询的初始余额是上次运行时的最终余额。连接两次不带命令行参数运行客户程序的提示信息类似如下画面：

```
// 第一次运行时输出 ...
David Zeng 的帐户余额为 685.38 元。
存款 200 元后，余额变为 885.38 元。
取款 600 元后，余额变为 285.38 元。
```

```
// 第二次运行时输出 ...
David Zeng 的帐户余额为 285.38 元。
存款 200 元后，余额变为 485.38 元。
余额不足 600 元，取款失败，余额保持不变。
```

如果运行客户程序之前智能代理 osagent 未就绪，或者服务程序进程未启动完毕，客户程序执行 bind 方法时引发一个 CORBA 异常，屏幕上会出现类似如下的出错信息：

```
Exception in thread "main" org.omg.CORBA.OBJECT_NOT_EXIST:
  Could not locate the following POA:
    poa name : /bank_agent_poa
  minor code: 0 completed: No
    at com.inprise.vbroker.orb.LocatorBidder.getBid(LocatorBidder.java:32)
    at com.inprise.vbroker.ProtocolEngine.ManagerImpl.startBidding(Compiled Code)
    at com.inprise.vbroker.ProtocolEngine.ManagerImpl.getConnector(ManagerImpl.java:83)
    at com.inprise.vbroker.orb.DelegateImpl._bind(Compiled Code)
    at com.inprise.vbroker.orb.DelegateImpl.bind(DelegateImpl.java:143)
    at com.inprise.vbroker.CORBA.portable.ObjectImpl._bind(ObjectImpl.java:19)
    at com.inprise.vbroker.orb.ORB.bind(ORB.java:1048)
    at Bank.AccountManagerHelper.bind(AccountManagerHelper.java:78)
    at Bank.AccountManagerHelper.bind(AccountManagerHelper.java:67)
    at Client.main(Client.java:10)
```

3.3.9 互操作尝试

1. 跨越机器与操作系统

与 1.4.2 中 RMI 例子类似，本例也可以将服务端和客户端分布到不同的机器或操作系统之上。如需在 Linux 等操作系统上编译运行客户端或服务端程序，读者需要在对应操作系统上安装 VisiBroker for Java，由于本例客户端与服务端均没有使用特定操作系统相关的机制，因此客户端与服务端都可以方便移植到 Linux 等操作系统。

跨越机器运行时，需要修改客户端程序中绑定到账户管理员对象的代码，使其绑定到远端机器上的账户管理员对象，如程序 3-10 所示：

程序 3-10 绑定到远端机器对象的客户程序 Client.java

```
// 客户端的主程序

public class Client
{
    public static void main(String[] args)
    {
        // 初始化 ORB
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        // 利用 POA 全称与对象标识"BankManager"查找帐户管理员
        Bank.AccountManager manager = Bank.AccountManagerHelper.bind(
            orb, "/BankPOA", "BankManager".getBytes(),
            " serverhost", new com.inprise.vbroker.CORBA.BindOptions()););
        // 如果命令行参数未指帐户名字则使用缺省名字
        String name = args.length > 0 ? args[0] : "David Zeng";
        // 请求帐户管理员找出一个指定名字的帐户，无此帐户则新开一个
        Bank.Account account = manager.open(name);
        // 取该帐户的余额
        //……
    }
}
```

2. 跨语言

由于 CORBA 提供了跨语言的基本支持，因此本例可以将服务端或客户端用 C++等 Java 语言之外的其它程序设计语言实现。下面以在 Linux 平台上用 C++语言实现本例的客户端程序为例进行说明。

用 C++语言实现本例的客户端程序，首先需要在客户端机器上安装 VisiBroker for C++，

读者可以在安装 VisiBroker for Java 的机器上同时安装 VisiBroker for C++。VisiBroker for C++与 VisiBroker for Java 对于开发的不同支持主要包括：

- IDL 编译器：与 VisiBroker for Java 提供的 idl2java 类似，VisiBroker for C++提供了一个 IDL 编译器 idl2cpp 将开发人员用 IDL 定义的接口映射到 C++语言，生成 C++语言实现的客户端 Stub、服务端 Skeleton 以及相关的辅助文件，由于 C++语言支持类的多重继承与引用型参数，因此不需要 xxxPOATie 与 xxxHolder 等对应的类，生成的总的文件要比 idl2java 少。
- 编译与运行：VisiBroker for C++（4.5.1 for Linux）并不提供类似 vbjc 与 vbj 的辅助编译与运行工具，而是直接利用 Linux 平台上的 gcc 编译器和 make 指令完成编译；而 C++语言不是解释性语言，因此编译后得到的可执行程序直接运行。利用 make 指令编译程序时开发人员通常需要编写一个 makefile 来指示编译器具体的编译链接工作，读者可直接修改 VisiBroker for C++自带例子程序使用的 makefile 来使用或参考 gcc 相关手册自行编写。

程序 3-11 与程序 3-12 给出了一个用 C++实现的客户端程序（完成与程序 3-9 类似的功能）和该程序编译使用的 makefile。

程序 3-11　C++实现的客户程序 Client.C

```
// C++客户端的主程序
#include "Bank_c.hh"

// USE_STD_NS is a define setup by VisiBroker to use the std namespace
USE_STD_NS

int main(int argc, char* const* argv)
{
  try {
    // 初始化 ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    // Get the manager Id
    PortableServer::ObjectId_var managerId =
           PortableServer::string_to_ObjectId("BankManager");
    // 绑定到远端账户管理员
    Bank::AccountManager_var manager =
           Bank::AccountManager::_bind("/BankPOA", managerId, "serverhost");
    //如果命令行参数未指帐户名字则使用缺省名字
    const char* name = argc > 1 ? argv[1] : "Jack B. Quick";
    //请求帐户管理员找出一个指定名字的帐户，无此帐户则新开一个
    Bank::Account_var account = manager->open(name);
    // 取该帐户的余额
    CORBA::Float balance;
    balance  = account->getBalance();
    // Print out the balance.
    cout << "The balance in " << name << "'s account is $"
         << balance << endl;
  }
  catch(const CORBA::Exception& e) {
  }
  return 0;
}
```

程序 3-12　C++客户程序使用的 makefile

```
# C++客户程序的使用的 makefile
include $(VBROKERDIR)/examples/stdmk
```

```
EXE =   Client

all: $(EXE)

clean:
    -rm -f core *.o *.hh *.cc $(EXE)
    -rm -rf SunWS_cache

#
# "Bank" specific make rules
#


Bank_c.cc: Bank.idl
    $(ORBCC) Bank.idl

Client: Bank_c.o Client.o
    $(CC) -o Client Client.o Bank_c.o \
        $(LIBPATH) $(LIBORB) $(STDCC_LIBS)
```

在命令行状态使用以下命令编译客户程序：

prompt> make

在命令行状态使用以下命令编译客户程序：

prompt> ./Client

该客户端程序可以访问前面用 Java 实现的服务端程序，完成对服务端账户的访问。至此，本节例子程序可以实现跨越机器、操作系统与程序设计语言的互操作，如用 Java 语言实现的服务端程序可以运行在一台安装 Windows 操作系统的机器上，而用 C++语言实现的客户端程序可以运行在另一台安装 Linux 操作系统的机器上。

3. 进一步尝试

有兴趣读者可在前面讨论的基础上进一步尝试 CORBA 对于可互操作的良好支持：

- 跨越 ORB 平台，由于 CORBA2.0 引入了 GIOP 与 IIOP，因此基于不同厂商 ORB 平台的客户端程序与服务端程序也可以方便地交互，读者可以再安装另一支持 IIOP 的 CORBA 平台（如 Orbix 等），将本例的客户端程序或服务端程序移植到该平台上，进行跨越 ORB 平台的互操作。
- 跨越体系结构：CORBA 中 ESIOP 可以解决一部分跨越体系结构的互操作问题，如一个 CORBA 对象可通过协议桥接操作一个 DCOM 对象，读者也可以参照 OMG 相关手册进行尝试。

思考与练习

3-1　将练习 1-4 用 CORBA 实现。

3-2　试参考 VisiBroker for C++的联机参考手册，将 3-9 的客户程序改写为与 3-11 类似的 C++语言实现，尝试这两种不同的程序设计语言能很好地进行互操作。

3-3** 从本章的例子来看，程序 3-11 所示的服务程序完成的工作是比较固定的，那就是创建并激活分布式对象，而这一工作完全可以通过平台自动完成而不需要开发人员编写代码实现，你认为有没有必要由开发人员编写该服务程序？为什么？

第 4 章　编写对象接口

§4.1 概 述

4.1.1　接口与实现的分离

将一个大型系统划分为若干模块，将一个模块的接口与实现相分离，并进一步能为一个接口提供多个实现，这是大型程序设计追求的目标之一。这一设计目标直接带来的好处是有可能在不同时间／空间效率的算法和数据结构之间作出折衷选择。接口与实现的分离要求将模块的私有部分归属到实现一方，接口只提供模块对外公开的信息，并且接口与实现之间不能再简单地通过相同的名字进行匹配。

将这种思想应用在面向对象的分布式软件系统中会带来更大的好处。客户程序仅仅依赖于对象的接口，而不是对象实现。客户程序甚至可以采用与对象实现完全不同风格的程序设计语言来编写。

将对象接口与对象实现分离后，对象接口成为向外公布对象行为的唯一途径。为保证对象接口的可用性，对象接口的定义应允许包括对象的名字、对象上可进行的操作（操作名字、参数类型、返回结果、可能产生的异常等）、对象上可访问的属性（实际上可归结为一种 0 元操作）、相关的数据类型定义、常量定义、异常定义等信息。

4.1.2　接口定义语言

在 CORBA 模型中，对象实现对外提供信息和服务，客户程序利用这些信息与服务完成某些功能。这些信息与服务是对象实现与客户程序之间的一种合约（contract），只有严格遵循合约的规定双方才有可能协调工作。

OMG 的接口定义语言（Interface Definition Language，缩写为 IDL）就是书写这种合约的标准语言，客户方与服务方必须使用这种统一的语言才能正确地理解合约的内容。由于在分布式计算领域还存在 DCE 的 IDL、Microsoft 的 IDL 等其他缩写为 IDL 的接口定义语言，所以由 OMG 发布的接口定义语言通常简写为 OMG IDL，在无二义性的上下文中我们也可简写为 IDL。

OMG IDL 能够给出完整的接口定义，包括每一个操作需要的参数、返回结果、上下文信息以及可能引发的异常等，完全可胜任对复杂对象的规格说明的无二义性描述。OMG IDL 具有如下特点：

- 具有面向对象的设计风格。
- 可用于定义分布式服务的规格说明。
- 可用于定义复杂的数据类型。
- 独立于具体的程序设计语言。
- 独立于特定的硬件系统。

应注意的是 OMG IDL 仅定义了规格说明中的语法部分，未包括任何语义信息，因而一个 IDL 接口还不能理解为一个完整的抽象数据类型。这对于实现软件自动化或软件重用的目标是远远不够的。

OMG IDL 是一种说明性（declarative）语言，而不是一种程序设计语言，程序员的编程工作仍然采用传统的面向对象或过程式程序设计语言编写客户端代码和服务端代码。OMG

IDL 用于定义由远程对象所提供服务的接口，包括分布式对象的服务能力以及由客户程序和服务程序共享的复杂数据类型。IDL 支持整数、浮点数、字符、宽字符、布尔值、八进制位组、any 等基本数据类型，并可利用 typedef、struct、union、enum 等方式合成更复杂的数据类型。

由于用 IDL 编写的对象接口担当了中间角色，客户程序无法（也无需）知道服务端代码采用什么程序设计语言编写，程序员编写服务端程序也不必预测访问服务对象的客户程序由什么程序设计语言编写。这时，同一接口、多种实现的目标可轻而易举地达到。

§ 4.2 OMG IDL 的语法与语义

OMG IDL 的语法规则基本上是 ANSI C++的一个子集，再加上一些 IDL 特有的调用机制。由于 IDL 被设计为一种说明性语言，所以它支持 C++语言的常量、数据类型和操作的声明，但不允许出现任何具体的数据表示（如变量或对象实例的声明）和操作实现（如算法、控制结构、构造与析构函数等）。

在 CORBA 规范中，OMG IDL 的语义是以非形式化的自然语言表述的，理解 IDL 语义的最佳途径是掌握 IDL 到某一门程序设计语言的映射规范。IDL 到 Java 语言的映射规范规定了如何将各种 IDL 构造映射到相应的 Java 构造，在现有程序设计语言中 Java 语言拥有最接近于 OMG IDL 的构造，因而 IDL 到 Java 语言的映射规则也是最简单的。

IDL 的规格说明可分为 6 类，包括模块、类型、常量、异常、接口以及值（参见附录 1 规则 1~2）。

4.2.1 词法规则

OMG IDL 采用类似 ANSI C++的词法规则和预处理特性（如编译指令#include）。IDL 文件本身采用 ASCII 字符集，但字符与字符串文字常量则采用 Unicode，即 ISO Latin-1 (8859.1) 字符集。

在 OMG IDL 中，关键字是大小写敏感的，但标识符却是大小写无关的，如果两个标识符仅仅大小写有区别，就可能在某些情况下产生编译错误。例如标识符 account 与 ACCOUNT 会被认为是重复定义；由于 boolean 是 IDL 的关键字，程序员将 BOOLEAN 用作标识符是非法的。一旦声明了一个标识符后，所有对该标识符的引用必须与声明时的大小写完全相同，这样才可以自然地映射到那些像 C++或 Java 一样区分大小写的程序设计语言。出现这类大小写匹配问题时，IDL 编译器通常会给出一些警告信息。

4.2.2 模块的声明

模块用于限制标识符的作用域。尽管 OMG 没有硬性规定 IDL 文件必须使用模块，但是将相关的接口、类型、异常、常量等放在同一模块之中无疑是一种良好的设计风格，这相当于为标识符划分了不同的名字空间。

一个 IDL 模块被映射为一个同名的 Java 程序包，该模块中的所有 IDL 类型被映射到相应程序包中的 Java 类或接口。不包含在任何模块之中的 IDL 声明被映射到一个无名的 Java 全局作用域程序包。

注意 IDL 模块不同于 C/C++语言所采用的文件模块。一个 IDL 文件可能定义多个模块，也可能不包含任何模块。

4.2.3 类型的声明

1. 基本数据类型

OMG IDL 提供了 9 种预定义的基本数据类型（参见附录 1 规则 46, 53~68, 98），包括浮点类型（float、double、long double）、整数类型（包括有符号整数类型 short、long、long long 和无符号整数类型 unsigned short、unsigned long、unsinged long long）、字符类型（char）、宽字符类型（wchar）、布尔类型（boolean）、八进制类型（octet）、任意类型（any）、对象类型（Object）和值基类型（ValueBase）。注意 IDL 没有 C/C++语言中那种 int 类型，这避免了数据类型对具体机器的依赖性。

OMG IDL 基本数据类型映射到 Java 语言基本数据类型的规则如表 4-1 所示。

表 4-1 OMG IDL 基本数据类型到 Java 语言的映射

IDL 类型	Java 类型
boolean	boolean
char	char
wchar	char
octet	byte
string	java.lang.String
wstring	java.lang.String
short	short
Unsigned short	short
long	int
Unsigned long	int
longlong	long
Unsigned longlong	long
float	float
double	double

2. 复合数据类型

OMG IDL 声明用户自定义类型的方式与 C++语言非常类似，可利用枚举、结构、联合体、序列和数组等方式定义新的类型名字（参见附录 1 规则 42）。定义新类型的最简单方式利用 typedef 定义类型别名，但这时并没有创建任何新的类型，只是为原有类型起了一个新名字。

在 enum 声明的枚举类型中，标识符出现的次序定义了它们之间的相对次序，最大枚举长度为 2^{32} 个标识符。一个 IDL 枚举类型被映射为 Java 语言的一个同名的最终类，其中定义了一个取值方法，并为每一个枚举常量定义了两个静态数据成员，以及一个将整数转换为该类型值的方法和一个私有的构造方法（私有构造方法常用于阻止创建对象实例）。

struct 用于定义一种结构类型，名字作用域规则要求在一个特定结构中成员的声明必须是唯一的。OMG IDL 的结构类型被映射到 Java 语言的一个同名的最终类，该类按 IDL 定义中的次序为结构中的每一个域提供一个实例变量，并提供一个带参数的构造方法初始化所有的值，同时提供一个缺省构造方法使得结构中的域可以到以后再初始化。

IDL 的 union 实际上兼蓄了 C/C++语言的 union 和 switch，要求联合体中的每一个成员都必须有一个标签，如程序 4-1 所示。

程序 4-1　OMG IDL 的枚举类型、结构类型与联合类型示例

```
enum Status {Mass, PartyMember, LeagueMember};      // 政治面貌

struct Person {                                     // 个人资料
   string name;                                     // 姓名
   boolean sex;                                     // 性别
   short age;                                       // 年龄
   union Class switch(Status)  {
      case PartyMember:   struct PartyStatus {      // 党员需要填写三样东西
                             string partyName;      // 党派名称
                             string joinDate;       // 入党时间
                             string job;            // 担任党内职务
                          } ptMember;
      case LeagueMember:  struct LeagueStatus {     // 团员需要填写两样东西
                             string joinDate;       // 入团时间
                             string job;            // 担任团内职务
                          } lgMember;
      // 群众不必填写任何东西
   } politicsStatus;
};
```

由 sequence 定义的序列实际上是一个数组，但它具有两个特性：在编译时确定的最大长度和在运行时确定的实际长度。在声明序列时可指定序列的最大长度，也可不指明最大长度（即序列长度不受限制），这些都会影响 IDL 到具体语言的映射。IDL 的数组声明方式同 C++语言，支持多维数组。序列和数组都映射为 Java 语言中的数组。

字符串 string 也可定义为有限字符串和无限字符串。宽字符串 wstring 与 string 的不同只在于元素是 wchar 而不是 char。定点类型 fixed 表示最大有效位为 31 位的定点数，声明定点数时可指定一个非负整数作为比例因子。

本地类型 native 仅用于本地调用，如果用于远程调用会引发一个 MARSHAL 异常。引入本地类型的主要目的是专供对象适配器接口使用，在应用程序接口或服务中通常不使用本地类型。

复合数据类型的定义有时是递归的，IDL 规定只有 sequence 类型才允许递归，例如程序 4-2 声明了一个单向链表中的结点类型。注意 IDL 并没有类似 C++语言的指针类型。

程序 4-2　用 IDL 递归定义数据类型

```
   struct Node {
      ElementType element;
      sequence<Node> next;
   };

   typedef sequence<Node> List;
```

4.2.4　常量的声明

与 C++语言相似，IDL 的文字常量通过语法书写形式决定常量的类型，通过常量表达式可以定义符号常量。但是各种基本数据类型的内部表示方法、取值范围以及可运算的操作与 C++语言略有区别。

在 IDL 接口中定义的符号常量映射为 Java 语言时，对应着该接口的 Operations 文件中的常量声明。例如在接口 A 中声明的如下 IDL 常量：

```
const double PI = 3.1415926;
```

将映射为 AOperations 接口中的如下定义：

```
public final static double PI = (double) 3.1415926;
```

如果常量 PI 的声明不是出现在一个 IDL 接口中，则由 IDL 编译器创建一个与常量同名的公有接口，生成的 PI.java 内容如下：

```
public interface PI {
    public final static double value = (double) 3.1415926;
}
```

4.2.5 异常的声明

在 Java 或 C++语言中异常被定义为一个类，IDL 则引入关键字 exception 专门用于定义异常。异常由异常标识符与相关的成员组成（参见附录 1 规则 86），如果调用某一操作时引发了一个异常，可通过异常标识符确定引发的是哪个异常。如果该异常还声明了成员，还可访问这些成员的值；如果该异常未声明任何成员，意味着该异常无任何附加信息。

在 CORBA 中，异常被组织为图 4-1 所示的层次结构。

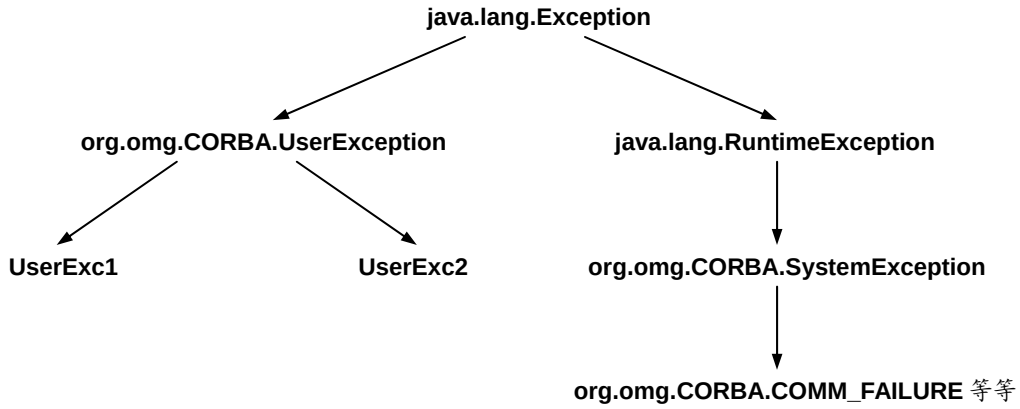


图 4-1　CORBA 异常的组织

4.2.6 接口的声明

接口是 IDL 定义中最重要的构造，它可分为抽象接口和具体接口，抽象接口不可用于创建对象实例，但有其特殊的语义性质。利用向前声明可实现接口定义间的相互引用。

由于 IDL 必须映射到一些不允许重载的程序设计语言，所以当前版本的 IDL 禁止使用 C++和 Java 语言那种重载机制。

1. 属性声明

在 IDL 接口中包含属性的声明，这些属性不应看作对象的状态数据，最好将它们理解为一种特殊的操作（零元操作）。接口中的可读写属性将映射为 Java 语言中一对互为重载的访问方法和设置方法，只读属性只映射到单个访问方法，这些方法与属性同名。例如由如下 IDL

```
attribute long count;
readonly attribute string name;
```

声明的属性 count 和 name 将映射为 Operations 类中的三个操作：

```
public int count();
public void count(int count);
public java.lang.String name();
```

注意 IDL 的属性映射为 Java 语言的方法而不是变量，故而 IDL 接口可直接映射到 Java 语言的接口而不类。

2. 参数传递方向

OMG IDL 声明形式参数表时不同于大多数程序设计语言，它要求每一个形式参数都必须包含一个标明参数传递方向的关键字 in、out 或 inout。关键字 in 表明参数从客户程序传向对象实现；out 表明数据从对象实现返回给客户程序；inout 表明数据从客户程序传给对象实现，然后返回给客户程序。

传递方向为 in 的参数可直接映射到 Java 方法的参数，并且 IDL 操作的返回方式也与 Java 方法相同。但是 out 和 inout 参数不能直接映射到 Java 语言的参数传递机制，为所有 IDL 基本类型和用户自定义类型生成的 Holder 类正是为了解决这一问题。

客户程序提供一个 Holder 类的对象实例，并以按值调用的方式传递给 out 或 inout 参数，如程序 4-3 的第二部分所示。调用前 inout 参数的传入值必须保存到作为实际参数的 Holder 实例中（既可通过带参数的构造方法，也可从另一 Holder 实例赋值，甚至可直接设置公有成员 value）。服务端的对象实现执行操作时会修改该 Holder 实例的状态，调用后客户程序可通过实际参数的公有成员 value 访问传出的值，如程序 4-3 的第三部分所示。

程序 4-3　IDL 的参数传递方向

```
// 第一部分: IDL 定义
interface Modes {
    long op(in long inArg, out long outArg, inout long inoutArg);
};

// 第二部分: 由 IDL 编译器生成的 Java 代码
public interface ModesOperations
{
    public int op(int inArg, org.omg.CORBA.IntHolder outArg,
       org.omg.CORBA.IntHolder inoutArg);
}

public interface Modes
    extends com.inprise.vbroker.CORBA.Object,
            ModesOperations,
            org.omg.CORBA.portable.IDLEntity
{}

// 第三部分: 程序员自己编写的 Java 代码（如客户程序）
    Modes obj = ... ;                                  // 获取一个目标对象
    int p1 = 57;                                       // 准备 in 参数的实际参数
    IntHolder p2 = new IntHolder();                    // 准备 out 参数的实际参数
    IntHolder p3 = new IntHolder(123);                 // 建立 inout 参数的实际参数
    int result = obj.op(p1, p2, p3);                   // 调用方法
    ... p2.value  ...                                  // 使用 out 参数的值
    ... p3.value  ...                                  // 使用 inout 参数的值
```

3. 单向操作

在 IDL 中没有返回值的操作还可设计为单向（oneway）操作。调用单向操作时请求被发送到服务端，但对象实现不确认请求是否真的收到，因而单向操作不允许引发任何异常或返回任何值。

例如客户程序向服务端的系统监控程序登记用户完成了某一操作的动作 log 可设计为如下的单向操作：

```
oneway void log(in string userName, in string description);
```

4. 上下文表达式

接口中的操作可附带一个上下文表达式，用于指明客户端那些影响到对象请求执行情况的上下文元素，如果没有上下文表达式则表明不存在与该操作有关的请求上下文。

上下文表达式是由 context 引导的一系列字符串常量清单，对象实现在执行请求指定的操作时，可利用客户端与这些字符串常量关联的值，这些信息存放在请求上下文（request context）之中。本书第六章介绍动态调用接口时，将详细解释请求上下文的用法。

5. 继承机制

IDL 接口的继承机制没有像 C++或 Java 语言那样的访问控制，相当于只有 C++语言的公有继承，父接口中定义的所有操作、属性、数据类型、常量和异常等在派生接口中都是可见的。在派生 IDL 接口中重定义或重载父接口中的操作都会被认为是重复定义，IDL 编译器会提示有语法错误。

IDL 接口支持多继承和重复继承。如果两个 IDL 接口含有同名的操作或属性，则不可同时继承这两个接口；如果同名的是常量、类型或异常，则允许同时继承这两个接口，但使用这些名字时必须必须采用前缀接口名和“::”的受限名字。如果接口是一个抽象接口，那么它只能继承其他的抽象接口。

即使在 IDL 文件的接口定义中使用了继承机制，对象实现也未必一定要有继承。IDL 支持多继承，可直接映射到 C++这类支持多继承的程序设计语言，也可映射到 Java 这类仅支持单继承的语言，甚至可映射到 C 这类非面向对象程序设计语言。虽然 IDL 中的方法都理解为动态绑定的，类似 C++语言中的虚函数，但它们的实现技术未必采用 C++语言常用的虚指针（virtual pointers）或虚表格（virtual tables）。

6. 抽象接口

抽象接口同时是 CORBA 对象引用和值类型的抽象，由于值类型无须支持 CORBA 对象引用的语义，因而抽象接口并不是隐式地继承 CORBA::Object，而是隐式地继承本地类型 CORBA::AbstractBase。如果抽象接口能够被成功地转换为一个对象引用类型（即普通接口），则可在该接口上调用 CORBA::Object 中的操作。

抽象接口的主要用途是用于操作的形式参数声明，从而可在运行时根据实际参数的类型动态地决定参数传递方式是按值调用还是按引用调用。详细用法参见本章 4.2.7 小节。

4.2.7 值类型的声明

值类型主要用于在网络中传递对象的状态信息，例如根据服务端一个对象实例的状态在客户端创建同类型、同状态的另一个对象实例。值类型有两种典型用法：一是在按值调用的参数传递方式中创建对象副本，二是在远程操作返回一个对象时创建对象副本。当一个对象的主要目的是为了封装数据，或者一个应用程序需要显式地对某个对象进行复制时，对象应使用值类型（valuetype），这时的对象实例通常简称为值（value）。

由于作为实际参数的值是由客户端创建的，服务端接收后需要创建该值的副本，所以双方都必须了解对象的内部状态以及与实现有关的信息。这些状态信息都必须在 IDL 文件中显式地表达出来，作为客户程序与对象实现之间合约的一部分，因而值类型同时兼有 IDL 的 struct 与 interface 两者的特性。我们可将值类型理解为带有继承与操作的 struct，与普通接口不同之处在于它拥有描述内部状态的属性，并且包含比普通接口更多的实现细节。

值类型有 4 种声明形式，包括普通值声明、抽象值声明、封装值声明以及向前声明（参见附录 1 规则 13~26）。

1. 抽象值类型

抽象值类型中只含有操作的基调，而没有状态数据和初始化操作，因而它们不能用于创建对象实例。一个抽象值类型可继承多个抽象值类型，但不允许继承普通值类型。

2. 普通值类型

普通值类型类似 OMG IDL 的 struct，但其中允许包含状态数据和初始化操作，并且一个普通值类型可以继承单个普通值类型和多个抽象值类型，还可支持（support）单个普通接口和多个抽象接口。普通值类型可用关键字 factory 声明一些初始化操作，用于以可移植方式创建值类型的实例。

ORB 收到一个值时，会寻找相应值类型的 Factory 类用于创建该值类型的一个本地对象实例，然后将值的状态数据解包到该实例中。如果找不到合适的 Factory 类，则引发一个 MARSHAL 异常。值类型与其工厂对象之间的映射与语言有关，例如在 VisiBroker for Java 中，值类型 V 的 Factory 类的缺省名字为 VDefaultFactory，如果使用其他名字则必须显式地调用 ORB 提供的 register_value_factory 方法在 ORB 注册工厂对象。使用缺省名字意味着隐式地注册了工厂对象。工厂对象还可利用 unregister_value_factory 方法注销，也可调用 lookup_value_factory 查找已注册的工厂对象。

如果形式参数类型为接口类型或值类型，它们分别静态地确定了参数传递方式为按引用调用和按值调用。由于抽象接口可看作是接口类型和值类型的抽象，利用抽象接口可支持运行时选择参数传递方式。如果形式参数的类型是抽象接口 A，则由实际参数 p 决定是按值还是按引用传递，其规则如下：如果 p 属于普通接口或其子类型，而该类型又是 A 的子类型，且 p 已注册到 ORB，则 p 被当作一个对象引用；否则，如果 p 属于值类型且该值类型支持 A（一个值类型可支持多个抽象接口），则 p 被当作一个值；否则，引发一个 BAD_PARAM 异常。

考察程序 4-4，如果调用操作 op 时实际参数 para 类型为 itype，则 para 作为一个对象引用传递；如果 para 类型为 vtype，则 para 作为值传递。

程序 4-4　由实际参数类型决定参数传递方式

```
abstract interface atype {};
interface itype: atype {};
valuetype vtype supports atype {};

interface x {
   void op(in atype para);
};
```

所有值类型都继承 CORBA::ValueBase，正如所有对象引用都是 CORBA::Object 的后代，但缺省情况下值类型不继承 CORBA::Object，因而这些值不支持通常的对象引用语义。除非显式地声明值类型支持某一接口类型，并将该值类型的实例经过对象适配器注册到 ORB，这些值才支持对象引用语义。

3. 封装值类型

封装值类型（boxed valuetype）允许将非值类型的 IDL 数据类型包装为值类型。例如下述封装值类型的 IDL 声明：

```
valuetype Label string;
```

等价于以下 IDL 值类型声明：

```
valuetype Label {
    public string name;
```

};

在 CORBA 模块中定义了一些标准的封装值类型，如 StringValue、WstringValue 等。一旦 IDL 数据类型被包装为值类型，这些数据类型就可当作值使用。使用封装值类型的最大理由是简单性：封装值类型完全由 IDL 编译器自动生成的代码实现，无需我们自己编写代码。

4. 定制值类型

如果声明值类型时加有前缀 custom，说明这是一种定制的值类型，表示不采用缺省的对象打包和解包操作，而是由程序员手工实现由 CustomMarshal 接口继承的 marshal 和 unmarshal 方法。

编译一个定制值类型时，该值类型继承 org.omg.CORBA.portable.CustomValue，而不像普通值类型那样继承 org.omg.CORBA.portable.StreamableValue。编译器也不为定制值类型生成 read 和 write 方法，程序员必须自己实现这两个方法。

5. 可截断值类型

声明可截断值类型时，必须在被继承的值类型之前加上关键字 trunctatble，表示允许将被继承的值类型看作该值类型的父类型。传递可截断值类型的对象时，所有在派生值类型中不属于父类型的状态数据都将丢失，因而可截断值类型常用于接收对象时不需要在派生值类型中定义的新数据成员或方法，或者无须知道值类型的确切派生类型的情况。

§4.3 使用值类型

本小节通过一个在第三章§3.3 节例子基础上修改得到的例子程序演示 IDL 值类型的实现方式与使用方法，建议读者认真比较这两个例子中“对象引用”与“值”的使用效果。

下面以新例子对第三章§3.3 节例子改动为线索来介绍。

4.3.1 IDL 定义

程序 4-5 所示的 IDL 文件展示了修改后的的接口定义。与程序 3-1 中定义的接口不同，账户由接口（interface）变为了值类型（valuetype），因此客户端使用的“账户”已经不再是远端的分布式对象，而是远端账户对象在本地的副本（即本地对象），尽管从代码上看账户管理员的 open 操作返回的仍然是 Account，但是返回值的性质发生了根本变化，原来返回的是远端对象的引用，现在返回的是一个对象副本。

程序 4-5 Bank.IDL

```
// 银行帐户管理系统的对象接口定义
module Bank {
   // 帐户
   valuetype Account {
      private float balance;      //帐户的当前余额
      // 存款
      void deposit(in float amount);
      // 取款
      boolean withdraw(in float amount);
      // 查询余额
      float getBalance();
      // 初始化
      factory open(in float init);
   };
```

```
    // 帐户管理员
    interface AccountManager {
        // 查询指定名字的帐户，查无则新开帐户
        Account open(in string name);
    };
};
```

由于值类型约定的是本地对象的规格说明，因此认定账户对象包含一个表示当前账户余额的数据成员 balance，此外，上面的程序中还为账户对象定义了一个初始化操作 open，用于创建账户对象，值类型的初始化操作用关键字 factory 声明。

4.3.2 编译 IDL 文件

使用 VisiBroker for Java 提供的 IDL 编译器 idl2java 编译该 IDL 文件后，会在 Bank 子目录中生成若干.java 文件。

由于 AccountManager 仍为接口（interface），所以仍会生成与原来类似的 7 个文件。而 Account 由接口变为了值类型，因此生成的文件会有所变化。idl2java 会为上面定义的值类型 Account 生成如下 5 个 Java 文件：

- Account.java：包含值类型定义的所有变量的声明与方法基调的声明。该类约束了提供的值类型实现必须实现哪些方法，通常作为值类型实现的基类。
- AccountValueFactory.java：仅当值类型的 IDL 定义中含有 factory 操作时，才生成该接口。它继承 org.omg.CORBA.portable.ValueFactory，包含值类型中所有通过 factory 关键字声明的初始化操作的基调。值类型的任何工厂（Factory）类都必须实现该接口，即任何工厂类都必须实现值类型中声明的初始化操作。
- AccountDefaultFactory.java：程序员可为一种值类型提供多个候选的工厂（Factory）类实现，AccountDefaultFactory 是由 IDL 编译器自动生成的缺省 Factory 类。如果存在 AccountValueFactory 接口则由缺省 Factory 类实现该接口，否则缺省 Factory 类直接实现 org.omg.CORBA.portable.ValueFactory 接口。程序员通常直接修改该类的方法来实现 Factory 类。
- AccountHelper.java：提供常用辅助功能的 Helper 类。如果值类型含有 factory 操作，Helper 类还提供这些初始化操作（例如 open 方法）的定义。
- AccountHolder.java：类似接口对应的 Holder 类，是支持 out 和 inout 参数的 Holder 类。

4.3.3 实现账户管理员

普通接口 AccountManager 的实现方式与第三章的例子程序相似，仍然采用继承 AccountManagerPOA 类的方式实现，如程序 4-6 所示。注意程序中实例 account 的类型虽然是用 IDL 声明的 Account，但却是一个地地道道的本地对象，该实例被创建后无须像原例子中一样注册到 ORB（被注释的黑体部分）。

程序 4-6 AccountManagerImpl.java

```
public class AccountManagerImpl
    extends Bank.AccountManagerPOA
{
  protected Hashtable accountList; // 该帐户管理员所负责的帐户清单
  public AccountManagerImpl(){
       accountList = new Hashtable();
  }
  public synchronized Bank.Account open(String name){
    Bank.AccountImpl account=
```

```
        (Bank.AccountImpl)accountList.get(name);
    if (account == null) {
      Random random = new Random();
      float balance = Math.abs(random.nextInt())%100000/100f;
      account = new AccountImpl(balance);
//    try {
//        org.omg.CORBA.Object obj =
//          _default_POA().servant_to_reference(accountServant);
//        account = Bank.AccountHelper.narrow(obj);
//    } catch(Exception exc) {
//         exc.printStackTrace();
//    }
      accountList.put(name, account);
      System.out.println("新开帐户: " + name);
    }
    return account;
  }
}
```

对象实现中的 open 方法是使用值类型的一种典型用法，即返回一个值类型 Account 的实例。服务端执行 open 方法时，调用 Account 的构造方法创建了一个名为 account 的实例，客户程序调用 open 方法获取的不是 account 的对象引用，而是在客户端生成的一个 account 的副本。实例 account 及其副本分别在服务端和客户端且都是本地的。

4.3.4 实现值类型

类似普通接口类型，我们必须为 Account 提供真正的实现，实现代码如程序 4-7 所示，值类型的实现 AccountImpl 继承了这一值类型的基类 Account。

程序 4-7 AccountImpl.java

```
// 派生值类型 Account 的实现
public class AccountImpl
    extends Bank.Account
{
    // 属性定义
//    protected float balance;
    // 构造方法，按指定余额创建新的帐户
    public AccountImpl(float bal){
        balance = bal;
    }
    // 往帐户中存款
    public void deposit(float amount){
        balance += amount;
    }
    // 从帐户中取款，不足余额则返回 false
    public boolean withdraw(float amount){
        if (balance < amount)  return false;
        else {
            balance -= amount;
            return true;
        }
    }
    // 查询帐户余额
    public float getBalance(){
        return balance;
    }
}
```

与第三章例子的实现不同，由于在 IDL 文件中已经定义了属性 balance，因此 IDL 编译器生成的基类 Account 已经包含了对应的属性定义，所以在 AccountImpl 类实现中不需要再定义对应的数据成员。

与普通接口的实现方式不同的是，普通值类型还必须实现一个 Factory 类。Factory 类提供了 IDL 中定义的所有 factory 初始化操作的实现。由于值类型 Account 含有 factory 操作 open，所以 IDL 编译器生成一个 AccountValueFactory 接口。生成的另一个类 AccountDefaultFactory 实现了 AccountValueFactory 接口，我们直接对该类的方法进行修改，结果如程序 4-8 所示，修改的代码为创建 AccountImpl 对象的两行代码。

程序 4-8 Bank\AccountDefaultFactory.java

```
// 值类型Account 的缺省Factory 类
package Bank;

public class AccountDefaultFactory
    implements AccountValueFactory
{
    public java.io.Serializable read_value(
         org.omg.CORBA.portable.InputStream is)
    {
        // 创建并初始化值类型的对象
        java.io.Serializable val = new AccountImpl(0);
        // 通过read_value 从InputStream 中读取对象状态
        val = ((org.omg.CORBA_2_3.portable.InputStream) is).
             read_value(val);
        return val;
    }

    public Bank.Account open(float init)
    {
        return new AccountImpl(init);
    }
}
```

如果 Factory 类的名字不是使用缺省名字（例如命名为 AccountSpecialFactory），则在客户程序和服务程序中都必须使用程序 4-9 所示语句将工厂对象显式地注册到 ORB。

程序 4-9　显式地注册非缺省名字的工厂对象

```
        ...
        // 确定工厂对象的标识
        String id = AccountHelper.id();
        // 创建一个工厂对象
        AccountSpecialFactory factory = new AccountSpecialFactory();
        // 将工厂对象注册到 ORB
        ((org.omg.CORBA_2_3.ORB) orb).register_value_factory(id, factory);
        ...
```

注意 vbjc 编译客户端或服务端的主程序时不会自动编译 AccountDefaultFactory 类，我们必须自己动手完成编译。为避免涉及众多生成的 Java 类之间的依赖性问题，最简单的方式是在执行如下命令：

```
prompt> vbjc Bank\*.java
```

4.3.5　服务程序与客户程序

与第三章例子相同，服务程序的主要任务是创建并在 ORB 注册账户管理员伺服对象

managerServant，由于在新的例子中账户管理员仍然是一个分布式对象，因此可以使用与第三章例子完全相同的服务程序。

为了演示值类型的使用效果，我们对第三章使用的客户程序进行修改，修改后的代码如程序 4-10 所示。

程序 4-10　Client.java

```
// 客户端的主程序
import Bank.*;

public class Client
{
  public static void main(String[] args)
  {
    org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args, null);
    // 利用 POA 全称与对象标识"BankManager"查找帐户管理员
    Bank.AccountManager manager =
     Bank.AccountManagerHelper.bind(
            orb, "/BankPOA", "BankManager".getBytes());
    String name = args.length > 0 ? args[0] : "David Zeng";
    // 请求帐户管理员找出一个指定名字的帐户，无此帐户则新开一个
    Bank.Account account = manager.open(name);
    System.out.println(name + "的帐户余额为" +
         account.getBalance() + "元");
    account.deposit(200);
    System.out.println("存款 200 元后，本地余额为" +
         account.getBalance() + "元");
    account = manager.open(name);
    System.out.println("服务端" + name + "的帐户余额为" +
         account.getBalance() + "元");
  }
}
```

客户程序通过ORB获取账户管理员的远程对象引用manager后，调用该对象引用的open方法开设一个账户。注意由于Account是一种IDL值类型，调用open方法返回的实例account是客户端本地的局部对象，而不是通常的CORBA对象引用，客户端的实例从服务端的实例复制了状态后，它们两者之间就是相互独立的。客户程序的对返回的对象存款200元后，再次调用账户管理员的open操作重新打开同样的账户，查询其余额。

将本小节的例子程序拿去编译并运行的读者会发现，客户程序控制台输出的存款操作已经完成，余额已经变化，但是再次打开同样的账户，却显示帐户的余额未发生变化，仍为存款之前的余额。我们有意设计成这种效果以反映值与对象引用的区别：由于 open 方法的返回值是值类型，因而客户端操纵的只是客户端对象的副本，当然不会对服务端的帐户余额产生影响。

§ 4.4 接口库

4.4.1　什么是接口库

OMG IDL 描述了一系列的模块、接口、类型、常量、异常等，这些定义存放在一个平坦的 IDL 文本文件中。这些对象接口定义也可由一个库服务来存储或管理，我们可以将 IDL 文件编译成可运行的接口库（Interface Repository，缩写为 IR）服务，由该服务提供对象接口的相关信息。

接口库中的信息与 IDL 文件相同，只不过这些信息被组织为更适合客户程序在运行时访问的形式，因而可将接口库看作关于 CORBA 对象的元信息的联机数据库。例如我们可开发一个客户程序浏览一个接口库的内容，作为项目开发小组的联机参考工具；接口库更常见的用法是查找对象引用的所有接口定义，利用接口库获取一个编译时未知类型的对象引用的接口信息，为利用动态调用接口（DII）或动态框架接口（DSI）调用对象作准备。

4.4.2 接口库的结构

接口是关于对象的描述，接口库则将这些接口描述又看作对象，为这些对象定义新的接口，从而无须引入数据库这一类的新机制。接口库将接口定义存储为一组具有层次结构的对象，然后提供以 OMG IDL 形式定义的接口来访问这些对象。接口库中保存的是接口的相关信息，而不是关于对象引用的信息。

在 IDL 文件中定义的各种构造具有明显的层次结构，例如一个 IDL 文件中包含模块的定义，模块中又包含接口的定义，接口中含有操作的定义。相应地，接口库也组织为层次结构，例如一个接口库中包含 ModuleDef 对象，ModuleDef 对象又包含 InterfaceDef 对象，InterfaceDef 对象又可包含 OperationDef 对象。我们根据接口库的 ModuleDef 就可了解它所包含的 InterfaceDef、OperationDef 等；反之亦然，给定 InterfaceDef 即可知道包含它的 ModuleDef。所有其他的 IDL 构造，如类型、常量、异常、属性、值类型等，都可如此表示在接口库中。

接口库中的模块定义、接口定义等对象都有一个字符串作为名字，有一个枚举常量表示对应的类型。但是用于唯一标识一个接口库对象的字符串是接口库标识。一个库标识由三部分组成，不同部分之间由冒号“:”分隔。开头部分固定为“IDL”，结尾部分表示版本号（例如“1.0”），中间部分是一系列由斜杠“/”分隔的标识符，表示接口库对象在 IDL 接口定义中的层次结构。例如对应本章程序 4-5 所示 Bank.idl 的接口库中，值类型 Account 的库标识为“IDL:Bank/Account:1.0”。

接口库中还包含了类型码（type code），类型码是表示参数类型或属性类型的值，只要有类型码就可确定一个类型的完整结构。类型码无须由程序员显式地定义在 IDL 文件中，它们会自动从 IDL 文件中出现的各种类型生成。类型码可用于打包和解包 any 类型的对象实例，any 类型是一个通用类型，可表示任意类型并用于动态调用接口。从接口库或 IDL 编译器都获取类型码，在本书接下来的两章中我们还将看到类型码的作法。

使用接口库时应注意接口库是可读写的，并且无任何访问控制保护。一个错误的或恶意的客户程序可能破坏接口库或获取接口库的敏感信息。

4.4.3 接口库管理工具

开发人员可创建多个接口库，并自己决定如何调配和命名接口库。例如一个项目小组可能约定用一个中心接口库包含所有最终软件产品的对象接口，而不同的开发人员则可创建自己的接口库进行测试。

接口库采用典型的客户机 / 服务器模型。VisiBroker for Java 提供的接口库服务程序称为 irep，该程序作为一个监控进程（daemon）运行，可用于创建接口库并装入其内容，它必须结合智能代理（osagent）一起使用。开发人员还可利用另外两个工具 ir2idl 和 idl2ir 浏览、更新和设置接口库，或者自己编写专用的接口库客户程序来监控或新接口库。

1. 装入接口库

例如，可在命令行状态输入下述命令指示从 IDL 文件 Bank.idl 创建一个名为 BankIR 的接口库：

prompt> start irep BankIR Bank.idl

进程 irep 启动后，将 Bank.idl 中的内容装入到接口库 BankIR 中，然后等待响应客户程序送来的请求。如果只是想创建一个空的接口库，可使用如下命令：

prompt> start irep BankIR

irep 提供最简单的事务功能。如果指定的 IDL 文件不能成功装入，接口库会将其内容回卷（rollback）到原状态；如果成功装入则提交新的状态供后续事务使用。

2. 浏览接口库内容

利用客户程序 ir2idl 可查看接口库的内容。例如，以下命令列出接口库 BankIR 中的所有定义：

prompt> ir2idl -irep BankIR

运行该命令要求在网络中至少已启动一个智能代理，并且程序员已启动一个 irep 服务程序装入接口库 BankIR。下述命令还可将接口库中的 IDL 定义存储到一个名为 NewBank.idl 的 IDL 文件中：

prompt> ir2idl -irep BankIR -o NewBank.idl

多次启动 irep 进程可装入多个不同名字的接口库，如果在 ir2idl 后不接任何参数，如：

prompt> ir2idl

则表示随机地列出由智能代理能找到的某一个接口库的内容。

3. 更新接口库内容

利用客户程序 idl2ir 可更新接口库的内容。例如，以下命令指示根据 IDL 文件 Bank.idl 中的定义更新接口库 BankIR：

prompt> idl2ir -irep BankIR -replace Bank.idl

接口库中的内容不能直接使用 idl2ir 或 irep 工具修改或删除。为更新接口库中的某一个项目，必须退出 irep 程序，编辑曾在 irep 命令行指定的 IDL 文件，然后使用更新过的 IDL 文件重新启动 irep。

如果想删除接口库中所有的项，可使用一个全空的 IDL 文件替换其内容。例如使用名为 Empty.idl 的全空 IDL 文件，可运行以下命令：

prompt> idl2ir –irep BankIR –replace Empty.idl

4.4.4 接口库客户程序

存储在接口库中的对象类型基本上对应着 IDL 的语法单位，如表 4-2 所示。例如，一个 StructDef 类型的对象包含与一个 IDL 结构声明相同的信息，一个 InterfaceDef 类型的对象包含与一个 IDL 接口声明相同的信息，一个 PrimitveDef 对象包含与一个 IDL 基本类型声明（如 boolean、long 等）相同的信息。

表 4-2 存储在接口库中的对象

接口库对象类型	描　　述
Repository	表示包含所有其它对象的顶层模块。
ModuleDef	表示一个 IDL 模块声明，可包含 ModuleDef、InterfaceDef、ConstantDef、AliasDef、ExceptionDef 以及在 IDL 模块中定义的其他 IDL 构造的接口库对应定义。
InterfaceDef	表示一个 IDL 接口声明，可包含 OperationDef、ExceptionDef、AliasDef、ConstantDef 和 AttributeDef。
AttributeDef	表示一个 IDL 属性声明。
OperationDef	表示一个 IDL 操作声明，包含该操作所需要的参数表、可能引发的异常以及上下文列表。

ConstantDef	表示一个 IDL 常量声明。
ExceptionDef	表示一个 IDL 异常声明。
ValueDef	表示一个值类型声明，包含常量、类型、值成员、异常、操作和属性的列表。
ValueBoxDef	表示另一 IDL 类型的简单封装值类型。
ValueMemberDef	表示一个值类型的成员。
NativeDef	表示一个 IDL 的本地定义。
StructDef	表示一个 IDL 结构声明。
UnionDef	表示一个 IDL 联合体声明。
EnumDef	表示一个 IDL 枚举声明。
AliasDef	表示一个 IDL 的类型别名（typedef）声明。注意这是一个基接口，定义了 StructDef、UnionDef 等的共同操作。
StringDef	表示一个 IDL 字符串声明。
SequenceDef	表示一个 IDL 序列声明。
ArrayDef	表示一个 IDL 数组声明。
PrimitiveDef	表示一个 IDL 基本类型声明，包括 null、void、long、float、double、boolean、char、octet、any。

CORBA 为表 4-2 中的每一种对象类型都提供了丰富的操作用于创建或访问其中包含的各种定义，并将接口库中所有对象的共同特性定义为 3 个 IDL 抽象接口，如表 4-3 所示。这 3 个接口都提供了 Helper 类和 Holder 类，这些辅助类的作用与普通接口生成的 Helper 类和 Holder 类相同。

表 4-3　接口库对象的父接口

接口	派生的接口	提供的主要操作
IRObject	包括 Repository 在内的所有对象。	def_kind 返回一个对象的定义种类，例如模块、接口或操作; destroy 用于撤销当前对象。
Container	可包含其他接口库对象的接口库对象，例如模块或接口。	lookup 通过名字查找一个被包含的对象; contents 列出 Container 中的所有对象; describe_contents 返回 Container 中所有对象的描述; create_xxx 用于创建一个 xxx 对象，xxx 可能是 module、interface、struct 等等。
Contained	可被包含在其他接口库对象（即 Container）中的接口库对象。	id 用于设置或返回对象的、唯一的库标识; name 返回当前对象的名字; version 返回当前对象的版本号; defined_in 返回包含该对象的 Container 对象; describe 返回关于当前对象的描述; move 将当前对象移动到其他 Container 对象中。

本节给出一个简单的例子程序，演示如何获取接口库中的接口定义信息。由于接口库以 irep 作为服务程序，所以我们无须另外开发接口库的服务程序，只要编写客户程序查询或操纵接口库中的内容即可，客户程序的源代码如程序 4-11 所示。

程序 4-11　自编客户程序访问接口库中的内容

```
// 利用接口库检查对象接口中定义的操作
import org.omg.CORBA.*;

public class Client
{
  static final String MODULE_NAME = "Bank";
  static final String INTERFACE_NAME = "Account";
  static final String OPERATION_NAME = "transfer";
```

```
  public static void main(String[] args)
  {
    try {
      // 初始化 ORB 并获取接口库的根容器对象
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      Repository ir = RepositoryHelper.narrow(
          orb.resolve_initial_references("InterfaceRepository"));
      // 列出接口库中所有的模块定义
      Contained modules[] = ir.contents(DefinitionKind.dk_Module, true);
      System.out.println("接口库有" + modules.length + "个模块");
      for (int ptr = 0; ptr <= modules.length - 1; ptr++)
        System.out.println("模块" + (ptr + 1) + "库 Id 为" + modules[ptr].id());
      // 如果接口库中有模块 Bank 的定义，则列出其中所有的接口定义
      Container bankDef = ContainerHelper.narrow(ir.lookup(MODULE_NAME));
      if (bankDef != null) {
        System.out.println("接口库中包含模块" + MODULE_NAME);
        Contained interfaces[] = bankDef.contents(
            DefinitionKind.dk_Interface, true);
        System.out.println("模块" + MODULE_NAME + "有" + interfaces.length + "个接口");
        for (int ptr = 0; ptr <= interfaces.length - 1; ptr++)
          System.out.println("接口" + (ptr + 1) + "库 Id 为" + interfaces[ptr].id());
      }
      // 如果模块 Bank 中有接口 Account 的定义，则列出其中所有的操作定义
      Container accountDef = ContainerHelper.narrow(bankDef.lookup(INTERFACE_NAME));
      if (accountDef != null) {
        System.out.println("模块" + MODULE_NAME + "包含接口" + INTERFACE_NAME);
        Contained operations[] = accountDef.contents(
            DefinitionKind.dk_Operation, true);
        System.out.println("接口" + INTERFACE_NAME + "有" + operations.length + "个操作");
        for (int ptr = 0; ptr <= operations.length - 1; ptr++)
          System.out.println("操作" + (ptr + 1) + "库 Id 为" + operations[ptr].id());
      }
    } catch(Exception exc) {
      exc.printStackTrace();
    }
  }
}
```

客户程序首先初始化 ORB，然后利用 ORB 对象引用的 resolve_initial_references 方法获取接口库最上层容器的一个对象引用 ir，这是一个类型为 org.omg.CORBA.Object 的通用对象引用，必须利用接口库容器类 Repository 的 Helper 类将它窄化为 Repository 类型。注意这里名字“InterfaceRepository”是 resolve_initial_references 方法专用于接口库的硬编码（hard code），类似的硬编码还有 RootPOA、POACurrent、PolicyCurretn、NameService、DynAnyFactory 等等。

客户程序访问接口库对象的内容时，主要利用 contents 和 lookup 方法。方法 contents 列出接口库对象所包含的、指定类型的 IDL 定义内容，可指定的类型作为枚举常量定义在 DefinitionKind 中，其中 dk_all 表示所有类型。方法 lookup 将一个名字解析为一个类型为 Contained 的对象引用，由于还要利用该对象引用作为 Container 继续列出其中的内容，我们必须使用 Container 的 Helper 类将该对象引用的类型窄化为 Container。注意在 CORBA 应用程序中许多类型转换都必须利用 Helper 类的 narrow 方法完成，不能使用 Java 语言提供的显式或隐式类型转换。

运行上述例子程序之前，必须首先启动 osagent，并启动 irep 装入一个 IDL 文件作为接口库的服务程序。如果运行前 osagent 未就绪或 irep 未启动，resolve_initial_references 方法将引发一个 OBJECT_NOT_EXIST 异常。

假设 irep 将本书第三章例子程序 Bank.idl 装入作为接口库的内容，运行本小节的例子程序可得到如下所示的输出结果：

```
接口库有 1 个模块
模块 1 库 Id 为 IDL:Bank:1.0
接口库中包含模块 Bank
模块 Bank 有 2 个接口
接口 1 库 Id 为 IDL:Bank/Account:1.0
接口 2 库 Id 为 IDL:Bank/AccountManager:1.0
模块 Bank 包含接口 Account
接口 Account 有 3 个操作
操作 1 库 Id 为 IDL:Bank/Account/deposit:1.0
操作 2 库 Id 为 IDL:Bank/Account/withdraw:1.0
操作 3 库 Id 为 IDL:Bank/Account/getBalance:1.0
```

在本书接下来的两章中，我们还将看到接口库的另一种典型用法，即应用在客户端的动态调用接口和服务端的动态框架接口中，为动态调用提供准确的接口信息。

§ 4.5 编写对象接口的准则

4.5.1 如何编写好的对象接口

这是一个十分有价值却又很难回答的问题。高质量的 IDL 接口设计对 CORBA 应用系统的成功具有重大影响。在编写对象接口时，有大量与应用有关的问题需要考虑，因而不存在什么统一标准度量对象接口的优劣。尽管如此，人们在长期的分布式软件应用中仍总结出了几条启发性的规则可供参考：

(1) 由面向对象分析与设计理论可知，一个 IDL 接口应该与现实世界中的业务系统中的某个工作实体相对应。面向对象分析与设计的知识与经验有助于开发良好风格的 IDL 对象接口设计。面向对象的软件开发过程可看作一个建模的过程，而 IDL 接口定义只是这个模型的一部分形式化描述。

(2) 开发人员在设计分布式对象接口时必须始终贯彻这样的指导思想，即随时注意到同一进程中对象的接口与分布式对象的接口通常存在很大的区别，忽视这种区别会给分布式对象系统的可靠性、可伸缩性、可重用性等质量因素带来危害。

(3) 分布式对象设计必须考虑的重要问题之一是性能问题。注意使用远程调用时必须考虑网络的开销。本地调用通常以微秒级度量，但远程调用则以毫秒级计算。并且在设计 IDL 接口的操作时，应一次返回尽可能多的信息，从而避免客户程序又要发出远程调用以获取一些相关的数据。本书在稍后的高级课题中还将讨论与性能有关的若干问题。

(4) 区别会话型接口与实体型接口有利于设计人员组织众多的分布式对象。客户程序利用会话型接口表达与特定客户程序有关的上下文信息，而实体型接口则用于表示共享的业务对象。例如在一个典型的网上购物系统中，购物车属于会话型接口，而所购商品则属于实体型接口。

(5) 事务处理在许多分布式数据处理系统中也是必须考虑的重要问题之一。设计人员应仔细考虑系统中有哪些事务，以及如何将这些事务映射为接口上操作。此外，在使用异步事务时还须考虑如何确定事务的完成状态或当事务提交失败时状态的恢复。

4.5.2 典型的对象接口

在不同的分布式应用系统中存在许多不同类型的接口和设计模式，参考这些典型的对象接口设计有助于我们开发设计良好风格的 IDL 接口。常见的对象接口包括：

(1) 操纵型接口，这类接口常用于操纵单个对象实体。例如，本书第三章例子程序中的帐户接口 Account 属于这一类型。

(2) 工厂型接口，这类接口常用于创建或撤销对象。例如，本书第三章例子程序中的帐户管理员接口 AccountManager 属于这一类型，它负责帐户对象实例的开设与撤销。这种类型的接口名字通常以 Manager、Factory 等作为后缀。在 CORBA 模型中，工厂对象专指那些用于创建其他分布式对象的分布式对象。

(3) 查找与选择型接口，这类接口用于确定对哪个对象进行操纵。例如，在本书第三章的例子程序中，我们可能设计一个名为 AccountFinder 的接口，用于查找满足某些条件的数据，并返回一个或多个 Account 的对象引用供客户程序使用。

(4) 管理型接口，这通常是一个 RPC 风格的接口，我们将要操纵的对象设计为操作的参数，而不是调用这些对象的方法。这一类型的接口常用于在一次调用中操纵多个对象实例，例如在一个分布式的电算化会计系统中，GeneralLedgerManager 有一个“试算平衡”操作 checkBalance，该操作输入一系列帐户后检查所有帐户的借方与贷方是否相等。

思考与练习

4-1 OMG IDL 的值类型有什么作用？它与 IDL 接口有什么主要区别？

4-2 如何的保持 Account 为值类型不变的前提下，修改 § 4.3 中例子，使得客户程序中对账户的修改真正在服务端生效？

4-3 4-2 中基于值类型的方案与第三章基于接口的方案相比，采用值类型方案有什么优势，适用于什么应用场合？

第 5 章　编写服务端程序

从第三章给出的服务程序的步骤（参 3.3.5）来看，服务程序的流程比较固定，这个工作可以交给平台自动去完成。但是该流程仅仅是一个典型简单服务程序的流程，当服务端程序比较复杂（如需要管理大量服务端对象）时，服务程序就可能不会完全遵循该流程，比如，可能当有客户端请求某个对象时，才动态的创建并激活它，甚至服务端程序可能会根据客户的请求动态的创建新的对象适配器。

CORBA 中间件的另一优势就是提供了灵活的服务端模型，该模型基于可移植对象适配器体系结构，可帮助开发人员有效管理复杂的服务端程序。本章对 CORBA 服务程序的基本结构与机制进行讨论。

§5.1 可移植对象适配器

5.1.1　CORBA 对象与伺服对象

这是两个不同抽象层次的概念，抽象的 CORBA 对象与具体的伺服对象之间的彻底分离造就了 CORBA 独立于任何特定程序设计语言的特性，并为服务端程序的可移植性打下基础。对象适配器（object adapter）是一个重要的 ORB 组件（参见第二章之图 2-2），它负责将抽象的 CORBA 对象映射到具体的伺服对象。

CORBA 对象是一个抽象意义上的对象，可看作一个具有对象标识、对象接口以及对象实现的抽象实体。它之所以被称为抽象的，是因为并没有硬性规定 CORBA 对象的实现机制。由于独立于程序设计语言和特定 ORB 产品，一个 CORBA 对象的引用又称可互操作的对象引用（Interoperable Object Reference，缩写为 IOR）。从客户程序的角度看，IOR 中包含了对象的标识、接口类型以及其他信息以查找对象实现。在 CORBA 应用程序中简称“对象”或“对象引用”时，指的是 CORBA 对象或 IOR。

伺服对象（servant）则是指具体程序设计语言的对象或实体，通常存在于一个服务程序进程之中。客户程序通过对象引用发出的请求经过 ORB 担当中介角色，转换为对特定的伺服对象的调用。在一个 CORBA 对象的生命期中，它可能与多个伺服对象相关联，因而对该对象的请求可能被发送到不同的伺服对象。

对象标识（Object ID）通常是一个字符串，用于在对象适配器中标识一个 CORBA 对象。对象标识既可由程序员指派，也可由对象适配器自动分配，这两种方式都要求对象标识在创建它的对象适配器中必须具有唯一性。

伺服对象通过对象标识关联到 CORBA 对象。建立一个伺服对象与一个 CORBA 对象之间关联的过程称为“激活”（activate）或“体现”（incarnate），反之，撤销这种关联的过程称为“冻结”（deactivate）或“净化”（etherealize）。经过激活或体现后，CORBA 对象、伺服对象以及对象标识均可称为活动的（active），否则我们称 CORBA 对象、伺服对象以及对象标识为非活动的（inactive）。

所谓持久对象（persistent object）或瞬时对象（transient object）都是指一个 CORBA 对象。持久对象可在创建它的服务程序进程之外存在，瞬时对象只能在创建它的服务程序进程中存在。

5.1.2 对象适配器

服务程序的主要任务是利用对象实现创建伺服对象，然后将这些服务端本地的对象实例转换为可供远程使用的 CORBA 对象，由客户程序通过对象标识解析相应的 CORBA 对象引用后，调用对象实现提供的各种服务。应用程序开发人员管理服务端伺服对象、对象标识、对象引用以及它们之间关联的主要工具是对象适配器。

对象适配器负责决定在收到客户请求时应调用哪个伺服对象，然后调用该伺服对象上的合适操作。CORBA 支持多种不同类型的对象适配器，但所有对象适配器的主要作用都是创建对象引用，并将对象引用与真正执行服务的程序设计语言伺服对象相关联。

CORBA 规范从 2.2 版开始提供了可移植对象适配器（Portable Object Adapter，缩写为 POA）作为各种 ORB 产品的标准对象适配器。POA 提供了更完善的机制，取代早期的基本对象适配器（Basic Object Adapter，缩写为 BOA），成为我们开发服务端程序的主要工具。

5.1.3 可移植对象适配器体系结构

POA 是对象实现与 ORB 其他组件之间的中介，它将客户请求传送到伺服对象时，可能激活伺服对象，也可能按需创建子 POA。对 CORBA 对象的所有调用都会通过 POA，即使目标对象是本地的，即伺服对象与发出调用的对象位于同一地址空间，这使得 POA 可以统一地应用 POA 策略。由于 POA 以相同的方式处理本地对象与远程对象，所以我们还可利用 POA 的优点处理本地对象。POA 体系结构如图 5-1 所示。

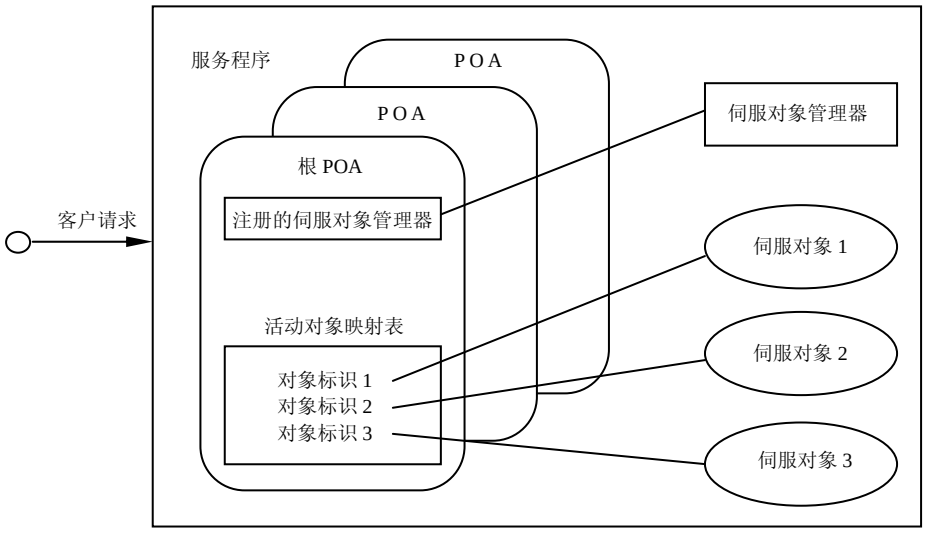


图 5-1 可移植对象适配器（POA）

由客户程序发出的请求中包含了创建对象引用的 POA 名字、对象标识以及目标机器与端口等信息。如果目标 POA 不存在，ORB 还允许调用适配器激活器创建所需的 POA。一旦请求被 ORB 传送到正确的机器与端口，监听该端口的 POA 管理器负责检查对象关键码（object key），对象关键码中含有 POA 名字与对象标识；POA 管理器利用对象关键码将请求传送给正确的 POA，然后由 POA 利用对象关键码确定对象标识，根据 POA 的策略集直接或间接地利用对象标识将请求传送到正确的伺服对象。

1. POA 层次

一个服务程序进程中可使用多个 POA，不同 POA 通过名字区别开来。这些 POA 的集合呈现一种层次结构，即每个 POA 都有一个父 POA，所有 POA 都是根 POA 的后代。每个 ORB 在创建时都自动带有一个根 POA，可根据需要从根 POA 创建其他子 POA。

POA 层次为服务程序中的对象标识提供了一个层次化的名字空间。通常每个伺服对象最多仅与一个 POA 相关联，POA“拥有”这些对象并负责删除这些对象，当 POA 被删除时其中的所有对象以及子 POA 也将被删除，因而 POA 层次的语义应理解为仅仅是 POA 删除行为的包含层次。

2. POA 管理器

POA 管理器（POA Manager）是一个对象，它将一个或多个 POA 组织在一起，为其中的 POA 提供共同的操作，POA 管理器的状态代表了它所管理的所有 POA 的状态。例如开发人员可通过 POA 管理器提供的操作决定是否接收丢弃对 POA 管理器所控制的 POA 的请求，也可利用 POA 管理器终止 POA。

3. 活动对象映射表

基于 BOA 的实现途径要求 CORBA 对象与伺服对象必须同时存在，但基于 POA 的对象实现允许 CORBA 对象不必与伺服对象相关联即可存在，并支持伺服对象不必与 CORBA 对象相关联也可存在。

每一个 POA 中都有一个活动对象映射表（Active Object Map，缩写为 AOM），表中保存了活动对象的对象标识以及与之关联的伺服对象，其作用是将活动对象通过对象标识映射到伺服对象。在一个特定的 POA 中，对象标识唯一地标识了一个 CORBA 对象。

为将伺服对象转换为一个可供远程调用的 CORBA 对象，必须建立 CORBA 对象与伺服对象之间的关联，这一关联过程甚至可以按需（on-demand）完成。此外，POA 还允许单个伺服对象（即缺省伺服对象）同时与多个 CORBA 对象相关联，这种特性对开发大规模应用意义重大。

4. 伺服对象管理器

伺服对象管理器（servant manager）是程序员自己提供的代码，用于取代 POA 活动对象映射表的功能。如果应用程序需要以一种更复杂的方案将对象标识映射到伺服对象，这时开发人员可设计专用的伺服对象管理器。伺服对象管理器负责决定一个 CORBA 对象是否存在，然后查找伺服对象并将伺服对象指派给 CORBA 对象。

为满足不同的应用需要，开发人员可设计多个伺服对象管理器。有两类伺服对象管理器可供选择：伺服对象激活器和伺服对象定位器，开发人员可利用 POA 策略决定选用哪一种类型。

5.1.4 模块 PortableServer 的 IDL 定义

POA 及其组件均以 IDL 接口定义，这些定义组成了 PortableServer 模块，如程序 5-1 所示。其中关键的接口包括 POA、POAManager 和 AdapterActivator，与伺服对象管理器相关的 ServantManager、ServantActivator 和 ServantLocator，以及 ThreadPolicy、LifespanPolicy 等各种策略。

程序 5-1 模块 PortableServer 的 IDL 定义

```
module PortableServer {
    // 向前引用
    interface POA;
    typedef sequence<POA> POAList;
    native Servant;
    typedef sequence<octet> ObjectId;
    exception ForwardRequest { Object forward_reference; };

    // 策略接口
```

```
// ------------------------------------------------------------
const CORBA::PolicyType THREAD_POLICY_ID = 16;
const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;

enum ThreadPolicyValue { ORB_CTRL_MODEL, SINGLE_THREAD_MODEL };
interface ThreadPolicy : CORBA::Policy {
   readonly attribute ThreadPolicyValue value;
};

enum LifespanPolicyValue { TRANSIENT, PERSISTENT };
interface LifespanPolicy : CORBA::Policy {
   readonly attribute LifespanPolicyValue value;
};

enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };
interface IdUniquenessPolicy : CORBA::Policy {
   readonly attribute IdUniquenessPolicyValue value;
};

enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };
interface IdAssignmentPolicy : CORBA::Policy {
   readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue { IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION };
interface ImplicitActivationPolicy : CORBA::Policy {
   readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };
interface ServantRetentionPolicy : CORBA::Policy {
   readonly attribute ServantRetentionPolicyValue value;
};

enum RequestProcessingPolicyValue {
   USE_ACTIVE_OBJECT_MAP_ONLY,
   USE_DEFAULT_SERVANT,
   USE_SERVANT_MANAGER
};
interface RequestProcessingPolicy : CORBA::Policy {
   readonly attribute RequestProcessingPolicyValue value;
};

// POA管理器接口
// ------------------------------------------------------------
interface POAManager {
   exception AdapterInactive{};
   enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };
   void activate()
      raises(AdapterInactive);
   void hold_requests(in boolean wait_for_completion)
      raises(AdapterInactive);
   void discard_requests(in boolean wait_for_completion)
      raises(AdapterInactive);
   void deactivate(in boolean etherealize_objects, in boolean wait_for_completion)
      raises(AdapterInactive);
   State get_state();
```

```
};

// 适配器激活器接口
// ------------------------------------------------------------------
interface AdapterActivator {
    boolean unknown_adapter(in POA parent, in string name);
};

// 伺服对象管理器接口
// ------------------------------------------------------------------
interface ServantManager {};

interface ServantActivator : ServantManager {
    Servant incarnate(in ObjectId id, in POA adapter)
        raises(ForwardRequest);
    void etherealize(in ObjectId id, in POA adapter, in Servant serv,
        in boolean cleanup_in_progress, in boolean remaining_activations);
};

interface ServantLocator : ServantManager {
    native Cookie;
    Servant preinvoke(in ObjectId id, in POA adapter,
        in CORBA::Identifier operation, out Cookie the_cookie)
        raises(ForwardRequest);
    void postinvoke(in ObjectId id, in POA adapter, in CORBA::Identifier operation,
        in Cookie the_cookie, in Servant the_servant);
};

// POA 接口
// ------------------------------------------------------------------
interface POA {
    // POA 属性
    readonly attribute string the_name;
    readonly attribute POA the_parent;
    readonly attribute POAList the_children;
    readonly attribute POAManager the_POAManager;
    attribute AdapterActivator the_activator;
    // 异常定义
    exception AdapterAlreadyExists {};
    exception AdapterNonExistent {};
    exception InvalidPolicy { unsigned short index; };
    exception NoServant {};
    exception ObjectAlreadyActive {};
    exception ObjectNotActive {};
    exception ServantAlreadyActive {};
    exception ServantNotActive {};
    exception WrongAdapter {};
    exception WrongPolicy {};
    // 创建与撤销 POA
    POA create_POA(in string adapter_name, in POAManager a_POAManager,
        in CORBA::PolicyList policies)
        raises(AdapterAlreadyExists, InvalidPolicy);
    POA find_POA(in string adapter_name, in boolean activate_it)
        raises(AdapterNonExistent);
    void destroy(in boolean etherealize_objects, in boolean wait_for_completion);
    // 用于创建策略对象的 factory 操作
    ThreadPolicy create_thread_policy(in ThreadPolicyValue value);
    LifespanPolicy create_lifespan_policy(in LifespanPolicyValue value);
    IdUniquenessPolicy create_id_uniqueness_policy(
        in IdUniquenessPolicyValue value);
    IdAssignmentPolicy create_id_assignment_policy(
        in IdAssignmentPolicyValue value);
```

```
        ImplicitActivationPolicy create_implicit_activation_policy(
            in ImplicitActivationPolicyValue value);
        ServantRetentionPolicy create_servant_retention_policy(
            in ServantRetentionPolicyValue value);
        RequestProcessingPolicy create_request_processing_policy(
            in RequestProcessingPolicyValue value);
        // 伺服对象管理器的注册
        ServantManager get_servant_manager()
            raises(WrongPolicy);
        void set_servant_manager(in ServantManager imgr)
            raises(WrongPolicy);
        // 为 USE_DEFAULT_SERVANT 策略提供的操作
        Servant get_servant()
            raises(NoServant, WrongPolicy);
        void set_servant(in Servant p_servant)
            raises(WrongPolicy);
        // 对象激活与冻结
        ObjectId activate_object(in Servant p_servant)
            raises(ServantAlreadyActive, WrongPolicy);
        void activate_object_with_id(in ObjectId id, in Servant p_servant)
            raises(ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);
        void deactivate_object(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
        // 创建 CORBA 对象引用的操作
        Object create_reference(in CORBA::RepositoryId intf)
            raises(WrongPolicy);
        Object create_reference_with_id(in ObjectId oid, in CORBA::RepositoryId intf)
            raises(WrongPolicy);
        // 对象标识映射操作
        ObjectId servant_to_id(in Servant p_servant)
            raises(ServantNotActive, WrongPolicy);
        Object servant_to_reference(in Servant p_servant)
            raises(ServantNotActive, WrongPolicy);
        Servant reference_to_servant(in Object reference)
            raises(ObjectNotActive, WrongAdapter, WrongPolicy);
        ObjectId reference_to_id(in Object reference)
            raises(WrongAdapter, WrongPolicy);
        Servant id_to_servant(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
        Object id_to_reference(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
    };

    // Current 接口，用于访问正被调用的 POA 与对象标识
    // ------------------------------------------------------------
    interface Current : CORBA::Current {
        exception NoContext {};
        POA get_POA()
            raises(NoContext);
        ObjectId get_object_id()
            raises(NoContext);
    };
};
```

§ 5.2 设计 POA 策略

5.2.1 什么是 POA 策略?

POA 策略是一个对象，负责控制相关 POA 的行为以及这些 POA 所管理的对象，使用

POA 前应仔细考虑应用程序所需的策略集。例如程序中有大量粒度细小的对象时，设计者需要一种优化方式；当程序中有大量需要长时间的操作时，设计者可能需要另一种优化方式。POA 策略可为不同的应用程序设计目标而配置不同的 POA。

5.2.2 选用 POA 策略

目前 CORBA 规范定义了 7 种标准的 POA 策略，不同的 ORB 产品从可伸缩性或可靠性等因素考虑引入一些新的 POA 策略。由于 POA 及其策略都是以 IDL 定义的，不同 ORB 产品很容易扩充新的策略或为特定策略添加新的特性。

选用 POA 策略时应注意某些策略值之间存在依赖关系，例如 IMPLICIT_ACTIVATION 策略要求同时使用 SYSTEM_ID 和 RETAIN 策略，而 USE_ACTIVE_OBJECT_MAP_ONLY 策略还必须结合 RETAIN 策略使用。

1. 线程策略

该策略指定 POA 使用的线程模型，它有两种取值：

(1) ORB_CTRL_MODEL（缺省值）：表示由 POA 负责将请求指派到线程。在多线程环境中，并发请求可以用多线程传送。

(2) SINGLE_THREAD_MODEL：这时只有一个线程，POA 顺序地处理请求。

2. 生命期策略

该策略指定 POA 中对象实现的使用期限，它可有以下值：

(1) TRANSIENT（缺省值）：由 POA 创建的对象引用是瞬时的，这些对象在创建它的 POA 之外不可存在。一旦 POA 被冻结为非活动状态，调用该 POA 创建的任何对象引用将引发 OBJECT_NOT_EXIST 异常。对话型对象通常设计为瞬时对象。

(2) PERSISTENT：由 POA 创建的持久对象可在创建它的 POA 之外存在，因而使用这一策略通常还会同时采用 USER_ID 策略。对持久对象的请求可能导致隐式地激活一个进程、一个 POA 以及实现该对象的伺服对象，这意味着 POA 应注册伺服对象定位器或激活器。实体型对象通常设计为持久对象。

3. 对象标识唯一性策略

该策略允许多个抽象对象共享一个伺服对象，它可取下列值：

(1) UNIQUE_ID（缺省值）：被激活的伺服对象仅支持一个对象标识，POA 不允许一个伺服对象与多个 CORBA 对象相关联。

(2) MULTIPLE_ID：被激活的伺服对象可以有一个或多个对象标识，在运行时刻调用伺服对象的操作时必须由操作内部决定对象标识。

应指出的是，对象标识在特定的 POA 中总是唯一的，对象标识唯一性策略指的是对象标识与伺服对象之间关联的唯一性或多重性。

4. 对象标识指派策略

该策略指定对象标识是由程序员编写的服务程序生成还是由 POA 自动生成，它可取以下值：

(1) USER_ID：由应用程序为 POA 的对象引用指定对象标识。

(2) SYSTEM_ID（缺省值）：由 POA 为它的对象引用分配对象标识。如果同时还采用了 PERSISTENT 策略，则在同一 POA 的所有实例中对象标识必须是唯一的。

典型情况将是 USER_ID 策略用于持久对象，将 SYSTEM_ID 策略用于瞬时对象。如果想将 SYSTEM_ID 策略用于持久对象，可从伺服对象或对象引用中提取对象标识。

5. 伺服对象保持策略

该策略指定 POA 是否将活动伺服对象保存在活动对象映射表中，它有两种取值：

(1) RETAIN（缺省值）：POA 利用活动对象映射表跟踪对象的激活情况，通常与伺服对象激活器或 POA 显式激活方式结合使用。

(2) NON_RETAIN：POA 不在活动对象映射表中保存活动的伺服对象，通常结合伺服对象定位器一起使用。

该策略决定 POA 是否利用活动对象映射表跟踪对象标识与伺服对象之间的关联。如果选择 NON_RETAIN 策略则意味着不使用活动对象映射表，因而使用该策略的应用程序必须提供一个伺服对象定位器类型的伺服对象管理器。

6. 请求处理策略

该策略指定 POA 如何处理请求，它有三种取值：

(1) USE_ACTIVE_OBJECT_MAP_ONLY（缺省值）：POA 仅依赖于活动对象映射表决定哪些对象标识可用以及对象标识关联到哪些伺服对象。如果在活动对象映射表中找不到对象标识，则引发 OBJECT_NOT_EXIST 异常。该值必须结合 RETAIN 策略使用。

(2) USE_DEFAULT_SERVANT：如果 POA 在活动对象映射表中找不到对象标识，或已设置 NON_RETAIN 策略，则将请求分派给一个缺省伺服对象。缺省伺服对象必须先注册，如果未注册则引发 OBJ_ADAPTER 异常。该值必须结合 MULTIPLE_ID 策略使用。

(3) USE_SERVANT_MANAGER：如果 POA 在活动对象映射表中找不到对象标识，或已设置了 NON_RETAIN 策略，则 POA 使用一个伺服对象管理器激活伺服对象。

7. 隐式激活策略

该策略指定 POA 是否支持伺服对象的隐式激活，它可取以下值：

(1) IMPLICIT_ACTIVATION：POA 支持隐式激活，服务程序可调用 servant_to_reference 操作或 servant_to_id 操作将伺服对象添加到活动对象映射表并转换为对象引用，也可调用伺服对象的_this 方法激活伺服对象。该值要求同时使用 SYSTEM_ID 和 RETAIN 策略。

(2) NO_IMPLICIT_ACTIVATION（缺省值）：POA 不支持伺服对象的隐式激活，只有通过显式的调用才可将伺服对象与一个对象标识相关联。

8. 绑定支持策略

这是一个 VisiBroker 特有的策略，负责控制注册到 VisiBroker 智能代理（osagent）的 POA 和活动对象。虽然使用该策略会影响服务程序的可移植性，但对提高服务端的可伸缩性有很大帮助。设想服务程序中有数千个对象，将它们都注册到智能代理是不可行的，这时可改为将 POA 注册到智能代理。由于客户请求中包含 POA 名字与对象标识，智能代理可正确地转发请求。该策略可取以下值：

(1) BY_INSTANCE：所有活动对象注册到智能代理。该值要求同时使用 PERSISTENT 和 RETAIN 策略。

(2) BY_POA（缺省值）：仅将 POA 注册到智能代理。该值要求同时使用 PERSISTENT 策略。

(3) NONE：POA 和活动对象都不注册到智能代理。

5.2.3 POA 处理请求小结

ORB 通过 POA 管理器将客户请求传送到合适的 POA 后，该 POA 对请求的处理方式取决于它所采用的策略以及对象的激活状态。

如果 POA 采用 RETAIN 策略，则 POA 在活动对象映射表中查找与请求中对象标识相

关联的伺服对象，找到伺服对象则由 POA 调用该伺服对象的相应方法；如果 POA 采用 NON_RETAIN 策略，或采用 RETAIN 策略但找不到关联的伺服对象，则代之以如下步骤：

(1) 如果 POA 采用 USE_DEFAULT_SERVANT 策略，则由 POA 调用缺省伺服对象的相应操作。

(2) 如果 POA 采用 USE_SERVANT_MANAGER 策略，则当 POA 采用 RETAIN 策略时由 POA 调用伺服对象激活器的 incarnate 操作，当 POA 采用 NON_REATIN 策略时调用伺服对象定位器的 preinvoke 操作；如果伺服对象管理器未能成功地将对象“体现”出来，则引发 ForwardRequest 异常。

(3) 如果 POA 采用 USE_OBJECT_MAP_ONLY 策略，则引发一个异常。

§ 5.3 使用 POA

使用 POA 之前必须首先根据应用程序的需要设计好不同的 POA 策略集，然后 CORBA 应用程序通常遵循以下步骤创建与激活 POA：

(1) 调用 ORB 伪对象的 resolve_initial_references 操作获取根 POA 的引用；

(2) 调用根 POA 的操作定义应用程序所需的 POA 策略；

(3) 用自定义策略在根 POA 下创建一个子 POA，甚至创建一个完整的 POA 层次；

(4) 激活 POA 管理器，通常整个服务程序只需使用一个 POA 管理器；

(5) 创建并激活伺服对象。

上述过程仅仅是一个典型的 CORBA 服务程序开发过程，本书第三章的例子程序演示了这些步骤。由于不同应用程序各自的需求与特点不一样，在开发具体应用程序时上述步骤可能有所不同。例如为避免大量客户请求导致服务程序崩溃或易于监控服务端程序，在第四步服务程序可能不是简单地激活 POA 管理器，而是由系统管理员在服务器控制台以交互方式控制 POA 管理器的状态；又如，在第五步服务程序未必使用以对象标识显式激活对象的方式，因为这种方式难以管理服务程序中的大量对象，如果采用按需激活还必须设计一个伺服对象管理器；此外，服务程序中还可能使用适配器激活器按需创建新的 POA。

本小节主要讨论如何利用 POA 完成解决这些复杂问题的设计。

5.3.1 获取根 POA 对象引用

所有服务程序都必须获取根 POA 的对象引用，该对象引用可直接用于管理对象，也可用于创建新的子 POA。例如代码

```
org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
POA rootPOA = org.omg.PortableServer.POAHelper.narrow(obj);
```

获取了根 POA 的对象引用 rootPOA。通过 ORB 的 resolve_initial_references 操作可将硬编码“RootPOA”解析为一个通用对象，然后利用 POA 的 Helper 类将通用对象窄化为 POA 类型。

根 POA 有一个预定义的策略集，即线程策略为 ORB_CTRL_MODEL、生命期策略为 TRANSIENT、对象标识唯一性策略为 UNIQUE_ID、对象标识指派策略为 SYSTEM_ID、伺服对象保持策略为 RETAIN、请求处理策略为 USE_ACTIVE_OBJECT_MAP_ONLY、隐式激活策略为 IMPLICIT_ACTIVATION。

应留意到根 POA 的策略值与各种策略的缺省值基本相同，唯一例外是隐式激活策略值不同于隐式激活策略的缺省值。由于仅当创建 POA 时才可设置策略，服务程序不可改变已有 POA 的策略，因而根 POA 的策略集是不可更改的。

5.3.2 创建自定义策略的 POA

如果需要不同的 POA 行为，例如采用持久对象策略，则需要创建新的子 POA。POA 的层次结构并不意味着 POA 策略也具有相应的层次结构，子 POA 并不继承其父 POA 的任何策略。

调用 POA 对象引用的 create_POA 方法可创建新的 POA 作为该 POA 的子 POA。程序员可根据实际需要以这种方式创建多个子 POA，形成一个 POA 层次。例如代码

```
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};
POA myPOA = rootPOA.create_POA("ABC", rootPOA.the_POAManager(), policies);
```

从根 POA 创建一个名为“ABC”子 POA，该子 POA 具有持久的生命期策略。

服务程序创建子 POA 时只需提供子 POA 的名字，客户程序则必须使用完整的 POA 名字指定 POA。一个完整的 POA 名字记录了包含该子 POA 名字在内的整个 POA 层次路径，层次间用斜杠分隔。例如“/A/B/C”表示 C 是 B 的子 POA，B 又是 A 的子 POA，首个斜杠表示根 POA。创建 POA 时可随意为 POA 命名，但 POA 名字必须在父 POA 的所有子 POA 中是唯一的，否则会引发 AdapterAlreadyExists 异常。

5.3.3 使用 POA 管理器

创建新 POA 时必须为它指定一个 POA 管理器，该 POA 管理器将用于控制新 POA 的状态。在大多数应用中，服务程序的所有 POA 均使用同一 POA 管理器，例如上例中根 POA 的 POA 管理器用于控制新创建的子 POA 的状态。如果将 null 作为第二个参数传递给 create_POA 操作，系统会自动创建一个新的 POA 管理器与新建的 POA 相关联。POA 管理器的撤销也是隐式地，当 POA 管理器相关联的所有 POA 都被 destroy 操作撤销时，该 POA 管理器也被自动撤销。

一个 POA 管理器创建后可以有 4 种状态：持有状态（holding）、活动状态（active）、非活动状态（inactive）以及丢弃状态（discarding），这些状态也就是 POA 管理器所控制的 POA 的工作状态。调用 POA 管理器的 get_state 操作可返回 POA 管理器的当前状态。

POA 管理器的状态以及引起状态转换的操作如图 5-2 的状态转换图所示。

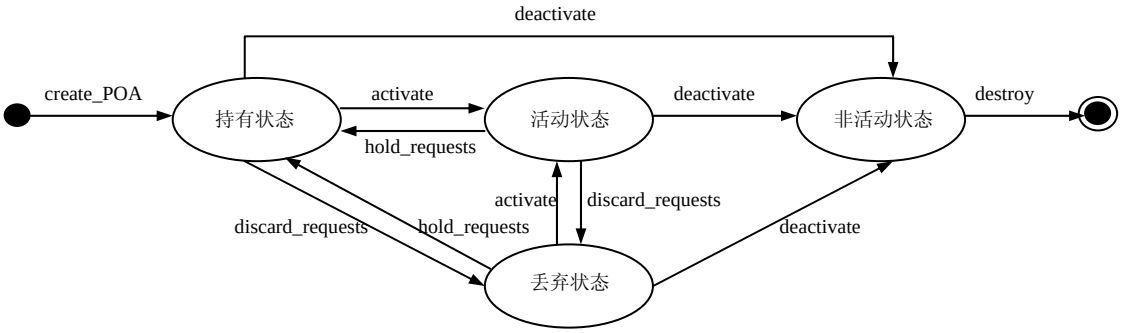


图 5-2 POA 和 POA 管理器的状态转换图

1. 活动状态

当 POA 管理器处于活动状态时，由它控制的所有 POA 将接收并开始处理请求。调用 POA 管理器的 activate 操作可将 POA 管理器从持有或丢弃状态改为活动状态。

注意即使在活动状态下，由于 ORB 实现或系统资源限制等原因，POA 也可能需要将来不及处理的请求排队。每一个 ORB 产品都会限制请求队列的最大长度，超过该限制时 POA

会返回一个 TRANSIENT 异常给客户程序。

2. 持有状态

POA 管理器被创建后即处于持有状态，在该状态下所有接入的请求被引导到一个队列中等候处理，POA 并没有对请求作任何处理。为分派这些请求，必须将 POA 管理器转换为活动状态，即利用 POA 管理器的 activate 操作激活该 POA 管理器。例如代码

```
rootPOA.the_POAManager().activate();
```

将根 POA 的 POA 管理器激活为活动状态。

调用 POA 管理器的 hold_requests 操作可将 POA 管理器由活动或丢弃状态改为持有状态，如果参数为 false 表示改为持有状态后立即返回，否则仅当在状态改变前已开始处理的所有请求已结束，或 POA 管理器被改为除持有状态之外的其他状态时，该操作才返回。所有排队等候且未开始处理的请求将在持有状态期间继续排队等候。

当POA管理器处于持有状态时，不会调用它所控制的所有POA上注册的适配器激活器，所有需要适配器激活器的请求也将排队等候处理。

3. 丢弃状态

当 POA 管理器处于丢弃状态时，由它控制的所有 POA 丢弃任何未开始处理的请求，此外不会调用注册到这些 POA 的任何适配器激活器。引入丢弃状态为服务程序提供了一种流量控制手段，例如为避免服务程序被突如期来的大量请求淹没，可利用该状态拒绝接入请求，这时 ORB 会返回 TRANSIENT 异常通知客户程序它们的请求已被丢弃，由客户程序决定是否重新发送请求。没有什么固定的方法判断是否以及何时 POA 收到了太多请求，通常这需要由开发人员自己负责建立线程监控程序。

调用 POA 管理器的 discard_requests 操作可将 POA 管理器从持有或活动状态改为丢弃状态，实际参数为 false 表示改为丢弃状态后立即返回，否则仅当状态改变前启动的所有请求均已结束，或 POA 管理器被改为除丢弃状态之外的其他状态时，该操作才返回。

4. 非活动状态

当 POA 管理器处于非活动状态时，由它控制的所有 POA 拒绝接入请求。与丢弃状态不同，仅当 POA 管理器要关闭时才使用该状态，因为处于该状态的 POA 管理器不能再转换到任何其他状态，否则会引发 AdapterInactive 异常。

POA 管理器的 deactivate 操作可将 POA 管理器从任何其他状态改为非活动状态，第一个参数为 true 表示状态改变后，所有具有 RETAIN 和 USE_SERVANT_MANAGER 策略的 POA 立即为每个活动对象调用伺服对象激活器的 etherealize 操作，否则不调用 etherealize 操作；第二个参数为 false 表示改为非活动状态后立即返回，否则仅当在状态改变前已启动的所有请求已结束，或所有具有 RETAIN 和 USE_SERVANT_MANAGER 策略的 POA 调用了 etherealize 方法后，该操作才返回。

5.3.4 激活与冻结对象

激活是指建立 CORBA 对象与伺服对象之间的关联，冻结是指撤销这种关联。如果 POA 采用 RETAIN 策略，对象既可被显式激活、隐式激活或按需激活，也可使用缺省伺服对象。如果 POA 采用 NON_RETAIN 策略，对象只能按需激活或使用缺省伺服对象。

1. 显式激活对象

显式激活对象是指在服务程序启动时，显式地调用 POA 的 activate_object_with_id 操作或 activate_object 操作激活对象。

(1) POA 的 activate_object_with_id 操作：该操作要求提供对象标识和伺服对象，并将两者的关联登记在活动对象映射表中。调用该操作的 POA 必须采用 RETAIN 策略，否则将引发 WrongPolicy 异常；如果对象标识已在活动对象映射表中存在，则引发 ObjectAlreadyActive 异常；如果 POA 采用了 UNIQUE_ID 策略且伺服对象已在活动对象映射表中存在，则引发 ServantAlreadyActive 异常。该操作常用于显式激活持久对象，当服务程序需要管理大量对象时，该操作有明显的不足之处。本书第三、四章例子程序的 Server.java 都演示了 activate_object_with_id 操作的用法。

(2) POA 的 activate_object 操作：该操作只要求提供伺服对象，由系统指派对象标识，然后将两者的关联登记在活动对象映射表中，最后返回系统分配的对象标识。该操作常用于显式激活瞬时对象。调用该操作的 POA 必须同时采用 RETAIN 和 SYSTEM_ID 策略，否则将引发 WrongPolicy 异常；如果 POA 采用了 UNIQUE_ID 策略且伺服对象已在活动对象映射表中存在，则引发 ServantAlreadyActive 异常。

2. 隐式激活对象

如果创建 POA 时采用了合适的策略，可调用 POA 的 servant_to_reference 和 servant_to_id 操作或伺服对象的_this 操作隐式地激活对象。

(1) POA 的 servant_to_reference 操作：该操作以一个伺服对象为参数，返回一个对象引用。调用该操作的 POA 必须采用 RETAIN 结合 UNIQUE_ID 或 IMPLICIT_ACTIVATION 策略，否则将引发 WrongPolicy 异常。如果 POA 同时采用 RETAIN 和 UNIQUE_ID 策略，且指定的伺服对象是活动的，则返回活动对象映射表中找到的对象引用；如果 POA 同时采用 RETAIN 和 IMPLICIT_ACTIVATION 策略，且 POA 还采用 MULTIPLE_ID 策略或指定的伺服对象不是活动的，则以 POA 生成的对象标识和接口库标识建立与伺服对象的关联，然后返回相应的对象引用；如果该操作是在执行对指定的伺服对象的请求中被调用，则返回与当前调用相关联的对象引用；否则，引发 ServantNotActive 异常。本书第三章例子程序的 AccountManagerImpl.java 演示了 servant_to_reference 操作的用法，注意执行该操作的 POA 是根 POA，而根 POA 预定义的隐式激活策略为 IMPLICIT_ACTIVATION。

(2) POA 的 servant_to_id 操作：与 servant_to_reference 操作不同，该操作返回的是对象标识而不是对象引用。调用该操作的 POA 可采用 USE_DEFAULT_SERVANT 策略，也可采用 RETAIN 结合 UNIQUE_ID 或 IMPLICIT_ACTIVATION 策略，否则将引发 WrongPolicy 异常。如果 POA 同时采用 RETAIN 和 UNIQUE_ID 策略，且指定的伺服对象是活动的，则返回与伺服对象相关联的对象标识；如果 POA 同时采用 RETAIN 和 IMPLICIT_ACTIVATION 策略，且 POA 还采用 MULTIPLE_ID 策略或指定的伺服对象不是活动的，则以 POA 生成的对象标识和接口库标识建立与伺服对象的关联，然后返回生成的对象标识；如果 POA 具有 USE_DEFAULT_SERVANT 策略，指定的伺服对象是缺省伺服对象，而且是在对缺省伺服对象的请求中调用该操作，则返回与当前调用相关联的对象标识；否则，引发 ServantNotActive 异常。

综上所述，如果 POA 采用 UNIQUE_ID 策略，则对非活动伺服对象调用上述两个操作会导致隐式激活；如果 POA 采用 MULTIPLE_ID 策略，则上述操作总会导致隐式激活，即使伺服对象已经是活动的。在隐式激活过程中，伺服对象是程序员指定的，而对象标识是由 POA 生成的，它们之间的关联将保存在 POA 的活动对象映射表中。

注意隐式激活是由服务程序的某些操作造成的，而不是响应某个客户程序请求。如果伺服对象不是活动的，客户程序请求一个非活动的对象无法让它变为活动的。

3. 按需激活对象

按需激活（activation on demand）指 POA 利用程序员提供的伺服对象管理器激活对象。

按需激活要求 POA 采用 USE_SERVANT_MANAGER 策略，并且首先调用 POA 的 set_servant_manager 操作设置 POA 的伺服对象管理器。set_servant_manager 操作在每个 POA 创建后只能调用一次，如果 POA 采用 RETAIN 策略则参数必须支持 ServantActivator 接口，如果 POA 采用 NON_RETAIN 策略则参数必须支持 ServantLocator 接口。

如果 POA 收到客户程序一个对 CORBA 对象的请求，而该 CORBA 对象尚未建立与任何伺服对象之间的关联，这时将发生按需激活。POA 将以对象标识为参数调用已注册的伺服对象管理器的 incarnate 或 preinvoke 方法。伺服对象管理器可有几种处理方式，例如：

(1) 查找合适的伺服对象，由该伺服对象为请求执行相应操作。

(2) 引发一个 OBJECT_NOT_EXIST 异常返回给客户程序。

(3) 将该请求转发给其他对象。

POA 的策略决定了接下来可能发生的其他步骤，例如如果 POA 设置了 RETAIN 策略，则用伺服对象与对象标识的关联更新活动对象映射表。本章的 §5.4 小节将详细介绍两类伺服对象管理器的具体用法。

4. 使用缺省伺服对象

如果 POA 采用 USE_DEFAULT_SERVANT 策略，则允许 POA 不管什么对象标识都调用同一个缺省的伺服对象，由单个伺服对象处理所有的客户请求。使用缺省伺服对象前，必须首先调用 POA 的 set_servant 操作设置该 POA 的缺省伺服对象，如程序 5-2 所示。

程序 5-2 使用缺省伺服对象

```
// 服务端的主程序
import org.omg.PortableServer.*;

public class Server {
   public static void main(String[] args) {
      try {
         // 初始化 ORB
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // 取根 POA 的一个引用
         POA rootPOA = POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
         // 创建应用程序所需的 POA 策略
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
            rootPOA.create_id_uniqueness_policy(
               IdUniquenessPolicyValue.MULTIPLE_ID),
            rootPOA.create_request_processing_policy(
               RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
            )
         };
         // 用自定义策略创建新的 POA
         POA newPOA = rootPOA.create_POA(
            "DefaultServantBankPOA", rootPOA.the_POAManager(), policies);
         // 创建伺服对象并设置为 POA 的缺省伺服对象
         AccountManagerImpl managerServant = new AccountManagerImpl();
         newPOA.set_servant(managerServant);
         // 激活 POA 管理器
         rootPOA.the_POAManager().activate();
         // 创建第一个对象引用
         newPOA.create_reference_with_id(
            "Zhang3".getBytes(), "IDL:Bank/AccountManager:1.0");
         // 创建第二个对象引用
         newPOA.create_reference_with_id(
            "Li4".getBytes(), "IDL:Bank/AccountManager:1.0");
         // 等待接入请求
```

```
            System.out.println("帐户管理员 Zhang3 和 Li4 已就绪 ...");
            orb.run();
        } catch(Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

程序 5-2 实际上重写了第三章例子程序中的 Server.java，其中新创建的 newPOA 采用 USE_DEFAULT_SERVANT 策略（该策略要求同时使用 MULTIPLE_ID），并调用 set_servant 操作将一个缺省伺服对象注册到 newPOA，然后调用 create_reference_with_id 操作分别创建了对象标识为 Zhang3 和 Li4 的对象引用，该操作的第二个参数“IDL:Bank/AccountManager:1.0”指定了代表对象接口类型的库标识。客户程序绑定到帐户管理员对象时既可使用对象标识 Zhang3，也可使用对象标识 Li4，但实际上在服务端都是同一个伺服对象负责执行真正的操作。

注意create_reference_with_id操作虽然创建了对象引用，但并未导致有任何对象的激活。这些对象引用可传递给客户程序，客户程序对这些对象引用发出请求时才引起按需激活对象或使用缺省伺服对象。类似的POA操作还有create_reference，该操作利用POA自动分配的对象标识创建对象引用。

5. 冻结对象

可调用 POA 的 deactivate_object 操作冻结对象，即从 POA 的活动对象映射表中删除对象标识与伺服对象之间的关联。该操作要求 POA 采用 RETAIN 策略。对象被冻结并不意味着该对象被永久撤销，以后它还可以被重新激活。

此外，还可通过调用 POA 管理器的 deactivate 操作冻结由该 POA 管理器所控制的所有 POA，POA 进入非活动状态后将拒绝对它所管理对象的任何请求。

§ 5.4 伺服对象管理器

伺服对象管理器是程序员自己编写的代码，其主要作用是查找并返回伺服对象。并不是所有 CORBA 应用程序都需要伺服对象管理器，例如如果服务程序在启动时装入所有对象就无需使用任何伺服对象管理器。

有两类伺服对象管理器：伺服对象激活器和伺服对象定位器。要使用伺服对象管理器，必须为 POA 设置 USE_SERVANT_MANAGER 策略，并结合伺服对象保持策略决定使用哪一种类型的伺服对象管理器。采用 RETAIN 表示使用伺服对象激活器，常用于激活持久对象；采用 NON_RETAIN 表示使用伺服对象定位器，常用于查找瞬时对象。

5.4.1 伺服对象激活器

由 ServantActivator 类型的伺服对象管理器激活的对象被记录在活动对象映射表中。利用伺服对象激活器处理请求时，POA 首先查找活动对象映射表；如果找到对象标识则调用伺服对象的合适操作并将结果返回给客户程序，否则以对象标识与 POA 作为参数调用伺服对象激活器的 incarnate 操作，由伺服对象激活器查找并返回一个合适的伺服对象，然后将该伺服对象登记到活动对象映射表中；最后调用伺服对象的合适操作并将结果返回给客户程序。此后，可根据实际应用需要决定是将伺服对象继续保留在活动对象映射表中，还是冻结该伺服对象。

为将第三章例子程序中的帐户管理员对象改为按需激活，使服务程序支持更灵活的对象

管理方案，我们为它提供一个伺服对象激活器，如程序 5-3 所示。

程序 5-3　伺服对象激活器示例 AccountManagerActivator.java

```
// 一个伺服对象激活器类型的伺服对象管理器
import org.omg.PortableServer.*;

public class AccountManagerActivator
    extends ServantActivatorPOA
{
    public Servant incarnate(byte[] oid, POA adapter)
        throws ForwardRequest
    {
        Servant servant;

        System.out.println("incarnate with ID = " + new String(oid));
        if ((new String(oid)).equalsIgnoreCase("Zhang3"))
            servant = (Servant) new AccountManagerImpl_1();
        else
            servant = (Servant) new AccountManagerImpl_2();
        new DeactivateThread(oid, adapter).start();
        return servant;
    }

    public void etherealize(byte[] oid, POA adapter, Servant serv,
        boolean cleanup_in_progress, boolean remaining_activations)
    {
        System.out.println("etherealize with ID = " + new String(oid));
        System.gc();
    }
}

class DeactivateThread
    extends Thread
{
    byte[] _oid;
    POA _adapter;

    public DeactivateThread(byte[] oid, POA adapter)
    {
        _oid = oid;
        _adapter = adapter;
    }

    public void run()
    {
        try {
            Thread.currentThread().sleep(15000);
            System.out.println("dactivate with ID = " + new String(_oid));
            _adapter.deactivate_object(_oid);
        } catch(Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

服务端程序为帐户管理员接口 AccountManager 提供了两个不同的对象实现，可找出很多理由说明为什么在某些应用场合确实需要这种设计，例如可能是由于不同算法实现的时空效率不同，也可能是由于不同管理员开设帐户的初始余额限额权限不同。总之，我们假设 AccountManager 接口两个不同对象实现的 Java 类分别命名为 AccountManagerImpl_1 和

AccountManagerImpl_2，如果对象标识为 Zhang3 则以 AccountManagerImpl_1 的对象实例为伺服对象，否则其他对象标识均以 AccountManagerImpl_2 的对象实例为伺服对象。

当 POA 在活动对象映射表中找不到对象标识时，会调用伺服对象激活器的 incarnate 方法，该方法根据不同的对象标识创建不同类型的伺服对象，在返回伺服对象前启动一个线程，该线程等待 15 秒后试图冻结刚才激活的对象，冻结对象前 POA 会自动调用伺服对象激活器的 etherealize 方法。如果伺服对象激活器不启动冻结对象的线程，则每次根据对象标识返回的伺服对象都会记录在 POA 的活动对象映射表中，当对象标识再次出现时 POA 可在活动对象映射表中找到，因而不再调用伺服对象激活器的 incarnate 方法。

程序 5-4 所示的服务程序首先创建一个策略为 USE_SERVANT_MANAGER 和 RETAIN 的 POA（注意 RETAIN 是缺省值，所以不必显式给出），然后创建一个伺服对象激活器的实例并注册到 POA，接着调用 create_reference_with_id 方法创建两个对象引用，这时只是创建对象引用而没有激活任何对象。

程序 5-4　使用伺服对象激活器的服务程序 Server.java

```
// 服务端的主程序
import org.omg.PortableServer.*;

public class Server
{
    public static void main(String[] args) {
        try {
            // 初始化 ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // 取根 POA 的一个引用
            POA rootPOA = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            // 创建应用程序所需的 POA 策略
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                rootPOA.create_request_processing_policy(
                    RequestProcessingPolicyValue.USE_SERVANT_MANAGER
                )
            };
            // 用自定义策略创建新的 POA
            POA newPOA = rootPOA.create_POA(
                "ServantActivatorPOA", rootPOA.the_POAManager(), policies);
            // 创建伺服对象激活器并注册到 POA
            ServantActivator sa = new AccountManagerActivator()._this(orb);
            newPOA.set_servant_manager(sa);
            // 激活 POA 管理器
            rootPOA.the_POAManager().activate();
            // 创建第一个对象引用
            newPOA.create_reference_with_id(
                "Zhang3".getBytes(), "IDL:Bank/AccountManager:1.0");
            // 创建第二个对象引用
            newPOA.create_reference_with_id(
                "Li4".getBytes(), "IDL:Bank/AccountManager:1.0");
            // 等待接入请求
            System.out.println("帐户管理员 Zhang3 和 Li4 已就绪 ...");
            orb.run();
        } catch(Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

当客户程序第一次以 Zhang3 作为对象标识绑定到对象引用时，POA 调用伺服对象激活器的 incarnate 方法将返回一个 AccountManagerImpl_1 类型的伺服对象；如果再次以对象标识 Zhang3 获取对象引用，由于 POA 可在它的活动对象映射表中找到，因而不会调用 incarnate 方法；如果对象标识是 Li4 或其他字符串，则会再次调用 incarnate 方法，返回的伺服对象是 AccountManagerImpl_2 类型。

该例子演示了如何利用伺服对象激活器管理服务端对象。从运行的结果来看，由于伺服对象激活器会在将对象激活 15 秒后将其冻结，因此在服务端内存中仅会保留在最近 15 秒内刚刚被用过的对象（最多两个，最少零个）。将服务端对象数量扩大，当服务端对象数量很多（典型的如大型网站的后台系统）时，本节例子给出的方案会变得非常有用意义：保持所有的对象同时在内存中存在会占用大量的系统内存（这往往系统运行效率急剧下降），采用类似的机制可以仅在内存中保留那些正在被使用或者刚刚被用过的对象，而暂时不用的对象不会占用系统内存，从而有效提高了系统内存的利用率，同时伺服对象激活器又可以保证所有对象的可用性（不在内存中的对象在使用时会被重新激活）。

5.4.2 伺服对象定位器

在许多应用场合，POA 的活动对象映射表可能变得很大并占用大量内存，这时可考虑以 USE_SERVANT_MANAGER 结合 NON_RETAIN 策略创建 POA，这意味着伺服对象与 CORBA 对象的关联不保存在活动对象映射表中。由于没有保存任何关联，每一次请求都导致 POA 直接以对象标识、POA 等参数调用已注册的伺服对象定位器的 preinvoke 操作。伺服对象定位器负责查找合适的伺服对象返回给 POA，由 POA 调用伺服对象的合适操作并将结果返回给客户程序后，再调用伺服对象定位器的 postinvoke 操作。

我们可以将程序 5-3 和 5-4 改写为使用伺服对象定位器的对应程序，程序 5-5 提供了一个伺服对象定位器，它必须实现从 ServantLocatorPOA 继承下来的抽象方法 preinvoke 和 postinvoke 方法。

程序 5-5　伺服对象定位器示例 AccountManagerLocator.java

```
// 一个伺服对象定位器类型的伺服对象管理器
import org.omg.PortableServer.*;
import org.omg.PortableServer.ServantLocatorPackage.CookieHolder;

public class AccountManagerLocator
    extends ServantLocatorPOA
{
    public Servant preinvoke(byte[] oid, POA adapter,
        java.lang.String operation, CookieHolder the_cookie)
        throws ForwardRequest
    {
        System.out.println("preinvoke with ID = " + new String(oid));
        if ((new String(oid)).equalsIgnoreCase("Zhang3"))
            return new AccountManagerImpl_1();
        return new AccountManagerImpl_2();
    }

    public void postinvoke(byte[] oid, POA adapter, java.lang.String operation,
        java.lang.Object the_cookie, Servant the_servant)
    {
        System.out.println("postinvoke with ID = " + new String(oid));
    }
}
```

程序 5-6 与程序 5-4 的不同之处在于它以 NON_RETAIN 策略创建 newPOA，并且调用

set_servant_manager 方法注册到 newPOA 的是一个伺服对象定位器而不是伺服对象激活器。建立相应客户程序并运行后，读者会发现伺服对象定位器与伺服对象激活器的一个重要区别，即如果连续两次以对象标识 Zhang3 绑定对象引用并调用该对象引用的操作，伺服对象激活器只调用一次 incarnate 方法，而伺服对象定位器则会调用两次 preinvoke 方法。当伺服对象处理请求完毕后，POA 还将调用伺服对象定位器的 postinvoke 方法。

程序 5-6　使用伺服对象定位器的服务程序 Server.java

```
// 服务端的主程序
import org.omg.PortableServer.*;

public class Server
{
    public static void main(String[] args) {
        try {
            // 初始化 ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // 取根 POA 的一个引用
            POA rootPOA = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            // 创建应用程序所需的 POA 策略
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                rootPOA.create_servant_retention_policy(
                    ServantRetentionPolicyValue.NON_RETAIN
                ),
                rootPOA.create_request_processing_policy(
                    RequestProcessingPolicyValue.USE_SERVANT_MANAGER
                )
            };
            // 用自定义策略创建新的 POA
            POA newPOA = rootPOA.create_POA(
                "ServantLocatorPOA", rootPOA.the_POAManager(), policies);
            // 创建伺服对象定位器并注册到 POA
            ServantLocator sl = new AccountManagerLocator()._this(orb);
            newPOA.set_servant_manager(sl);
            // 激活 POA 管理器
            rootPOA.the_POAManager().activate();
            // 创建第一个对象引用
            newPOA.create_reference_with_id(
                "Zhang3".getBytes(), "IDL:Bank/AccountManager:1.0");
            // 创建第二个对象引用
            newPOA.create_reference_with_id(
                "Li4".getBytes(), "IDL:Bank/AccountManager:1.0");
            // 等待接入请求
            System.out.println("帐户管理员 Zhang3 和 Li4 已就绪 ...");
            orb.run();
        } catch(Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

§ 5.5 适配器激活器

适配器激活器和伺服对象管理器一样，也是由程序员自己提供的代码。并不是所有 CORBA 应用程序都需要适配器激活器，如果一个服务程序在启动时就创建了它所需的全部 POA，那么就无需使用或提供任何适配器激活器。仅当需要在处理请求的过程中自动创建新

的 POA 时，才需要用到适配器激活器。

适配器激活器注册到一个 POA 后，为该 POA 提供按需（on-demand）创建子 POA 的能力。如果服务端程序以 true 作为第二个参数调用该 POA 的 find_POA 操作，或客户程序利用对象引用发送请求且对象引用中包含了该 POA 中不存在的子 POA 名字时，该 POA 会调用已注册适配器激活器的 unknown_adapter 操作。从根 POA 到要创建的子 POA 之间的层次路径中如果有 POA 不存在，则会依次调用 unknown_adapter 操作创建这些 POA。该操作成功创建子 POA 后应返回 true，否则返回 false。

例如程序 5-7 重写了第三章例子程序的服务端主程序，它创建一个适配器激活器对象实例并调用_this 方法转换为对象引用，然后注册到根 POA。为演示客户程序对一个不存在的 POA 发出请求时适配器激活器如何工作，服务程序按照用户自定义 POA 策略创建了一个 POA 层次：/GrandPOA/ParentPOA/ChildPOA，然后在 ChildPOA 中创建一个对象标识为“BankManager”的对象引用。最后服务程序撤销自建的 POA 层次，三个新建的子 POA 都被删除。这时除非根 POA 有按需激活子 POA 的功能，否则对 ChildPOA 的对象引用发送的请求将无法处理。

由于所需的 POA 不存在，客户程序无法利用生成的 Helper 类提供的 bind 方法获取服务程序输出的对象引用。为让客户程序能利用该对象引用发送请求，服务程序调用 ORB 伪对象的 object_to_string 方法将对象引用转换为字符串后存放在一个文件中，客户程序从该文件读出字符串后，又调用 ORB 伪对象的 string_to_object 方法将字符串转换为一个对象引用。

程序 5-7　服务程序 Server.java

```
// 演示适配器激活器用法的服务端主程序
import org.omg.PortableServer.*;

public class Server
{
   public static void main(String[] args)
   {
      try {
         // 初始化 ORB
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // 取根 POA 的引用
         POA rootPOA = POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
         // 创建适配器激活器伺服对象并转换为对象引用，然后设置根 POA 的适配器激活器
         AdapterActivator bankAA = new BankAdapterActivator()._this(orb);
         rootPOA.the_activator(bankAA);
         // 自定义应用程序所需的持久对象 POA 策略并用于创建一个 POA 层次
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };
         POA grandPOA = rootPOA.create_POA(
            "GrandPOA", rootPOA.the_POAManager(), policies);
         POA parentPOA = grandPOA.create_POA(
            "ParentPOA", rootPOA.the_POAManager(), policies);
         POA childPOA = parentPOA.create_POA(
            "ChildPOA", rootPOA.the_POAManager(), policies);
         // 创建一个对象引用并转换为字符串
         org.omg.CORBA.Object ior = childPOA.create_reference_with_id(
            "BankManager".getBytes(), "IDL:Bank/AccountManager:1.0");
         String iorString = orb.object_to_string(ior);
         // 将可互操作的对象引用（IOR）保存到文件中供客户程序使用
         try {
            java.io.PrintWriter writer = new java.io.PrintWriter(
               new java.io.FileWriter("BankIOR.dat"));
            writer.println(iorString);
```

```
            writer.close();
         } catch(java.io.IOException exc) {
            System.out.println("将 IOR 写入文件时出错: " + exc.getMessage());
            return;
         }
         // 撤销 POA 层次，包括所有的子 POA
         grandPOA.destroy(true, true);
         // 激活 POA 管理器
         rootPOA.the_POAManager().activate();
         // 等待处理客户程序的请求
         System.out.println("帐户管理员 BankManager 已就绪 ...\n");
         orb.run();
      } catch(Exception exc) {
         exc.printStackTrace();
      }
   }
}
```

程序 5-8 给出了程序员编写的适配器激活器源代码，该适配器激活器类必须实现从 AdapterActivatorPOA 类继承下来的抽象方法 unknown_adapter。unknown_adapter 方法根据应用程序所需 POA 策略创建新的子 POA，在程序 5-8 中实际上是恢复服务程序中被撤销的原有 POA 层次，并将 POA 策略修改为 USE_DEFAULT_SERVANT；然后在创建名为 ChildPOA 的子 POA 时提供一个缺省伺服对象为所有请求服务。

程序 5-8　适配器激活器 BankAdapterActivator.java

```
// 用户自定义的适配器激活器
import org.omg.PortableServer.*;

public class BankAdapterActivator
   extends AdapterActivatorPOA
{
   public boolean unknown_adapter(POA parent, String name)
   {
      System.out.println("auto-generate POA " + parent.the_name() + "/" + name);
      // 定义应用程序所需的 POA 策略
      org.omg.CORBA.Policy[] policies = {
         parent.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
         parent.create_request_processing_policy(
            RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
         )
      };
      // 创建新的 POA 作为 parent 的子 POA
      POA child = null;
      try {
         child = parent.create_POA(name, parent.the_POAManager(), policies);
      } catch (Exception exc) {
         exc.printStackTrace();
         return false;
      }
      // 设置新 POA 的适配器激活器
      child.the_activator(parent.the_activator());
      // 创建 childPOA 时准备好缺省伺服对象以及对象标识 BankManager 的对象引用
      if (name.equals("ChildPOA")) {
         System.out.println("正在激活 ChildPOA 的缺省伺服对象 ...");
         try {
            child.set_servant(new AccountManagerImpl());
            child.create_reference_with_id(
               "BankManager".getBytes(), "IDL:Bank/AccountManager:1.0");
         } catch (Exception exc) {
```

```
            exc.printStackTrace();
         }
      }
      return true;
   }
}
```

程序 5-9 所示的客户程序首先从文件中取出服务程序生成的对象引用的字符器，然后调用 ORB 的 string_to_object 方法转换为对象引用，并利用 Helper 类提供的 narrow 方法窄化为 AccountManager 类型。这样得到的对象引用与利用 Helper 类的 bind 方法获取的对象引用用法相同。

在运行客户程序时，我们会发现服务端的控制台显式 unknown_adapter 方法被调用了三次，分别用于创建 GrandPOA、ParentPOA 和 ChildPOA。

程序 5-9　客户程序 Client.java

```
// 演示适配器激活器用法的客户端的主程序

public class Client
{
   public static void main(String[] args)
   {
      // 初始化 ORB
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      // 从文件中取出服务程序写入的对象引用字符串
      String iorString = null;
      try {
         java.io.LineNumberReader reader = new java.io.LineNumberReader(
            new java.io.FileReader("BankIOR.dat"));
         iorString = reader.readLine();
         reader.close();
      } catch(java.io.IOException exc) {
         System.out.println("从文件读出 IOR 时出错: " + exc.getMessage());
         return;
      }
      // 将对象引用的字符串转换为对象引用
      Bank.AccountManager manager = Bank.AccountManagerHelper.narrow(
         orb.string_to_object(iorString));
      // 请求帐户管理员开设帐户并取帐户的初始余额
      Bank.Account account = manager.open("David Zeng");
      System.out.println("帐户余额为" + account.getBalance() + "元");
   }
}
```

§ 5.6 纽带机制

如本书第三章所述，编写对象实现有两种途径：一是使用继承机制，一是使用纽带机制（tie mechanism）。由于纽带机制不占用 Java 语言单继承的“配额”，所以特别适合将现有的 Java 应用程序集成到 CORBA 分布式对象系统中。

5.6.1　纽带机制工作原理

使用继承机制编写对象实现类时，通常继承由 IDL 编译器自动生成的 POA 类，这些 POA 类又继承 org.omg.PortableServer.Servant 类，并实现由 IDL 编译器自动生成的 Operations 接口。如果对象实现类需要利用 Java 语言的单继承达到其他更重要的目的，则更适合选择纽

带机制。

使用纽带机制时，对象实现不必继承生成的 POA 类，但必须实现生成的 Operations 类。服务程序创建的伺服对象不再是对象实现类型（这时对象实现类不必是 org.omg.PortableServer.Servant 的派生类），而是由 IDL 编译器生成的 POATie 类型的实例。

创建 POATie 的对象实例时，必须指定一个对象实现类的实例作为代表对象（delegate），POATie 类并没有提供什么新的语义，而仅仅是简单地将它收到的每一个操作请求委托给它的代表对象，由代表对象执行真正的操作。

5.6.2 改写例子程序

本小节采用纽带机制重新实现第三章例子程序的服务端。对比两种实现方式的源程序可看出，程序 5-10 所示的 Account 接口的对象实现改动很少，关键是采用钮带机制的对象实现不再继承生成的 POA 类，而是直接实现生成的 Operations 接口。

程序 5-10 采用纽带机制的对象实现 AccountImpl.java

```
// 使用纽带机制的帐户对象实现

public class AccountImpl
    implements Bank.AccountOperations
{
    // 属性定义
    protected float balance;


    // 构造方法，按指定余额创建新的帐户
    public AccountImpl(float bal)
    {
        balance = bal;
    }

    // 往帐户中存款
    public void deposit(float amount)
    {
        balance += amount;
    }

    // 从帐户中取款，不足余额则返回 false
    public boolean withdraw(float amount)
    {
        if (balance < amount)  return false;
        else {
            balance -= amount;
            return true;
        }
    }

    // 查询帐户余额
    public float getBalance()
    {
        return balance;
    }
}
```

程序 5-11 重写了 AccountManager 接口的对象实现，它的主要区别也是对象实现类 AccountManagerImpl 不再继承 AccountManagerPOA 类，因而可利用继承去重用其他更重要的类。该对象实现之所以要将一个 POA 作为构造方法的参数并保存为对象实例的状态，是

因为在 open 方法中需利用该 POA 隐式地激活一个 Account 对象的引用，这是由于对象实现不再继承 POA 类后，不可在它的方法中调用_default_POA 方法。应注意对象引用 account 相关联的伺服对象是 POATie 类型而不是 AccountImpl 类型的实例，由 POATie 类型的实例再关联到一个 AccountImpl 类型的实例。

程序 5-11　采用纽带机制的对象实现 AccountManagerImpl.java

```
// 使用纽带机制的帐户管理员对象实现

import java.util.*;
import org.omg.PortableServer.*;
import Bank.*;

public class AccountManagerImpl
    implements AccountManagerOperations
{
    // 属性的定义
    protected POA accountPOA;                  // 用于隐式激活对象的对象适配器
    protected Hashtable accountList;           // 该帐户管理员所负责的帐户清单

    // 构造方法，管理员开始时管理的帐户清单为空
    public AccountManagerImpl(POA poa)
    {
        accountPOA = poa;
        accountList = new Hashtable();
    }

    // 查找指定名字的帐户，找不到则以该名字新开一个帐户
    public synchronized Account open(String name)
    {
        // 在帐户清单中查找指定名字的帐户
        Account account = (Account) accountList.get(name);
        // 如果不存在则新创建一个
        if (account == null) {
            // 随机虚构帐户的初始余额，金额在 0 至 1000 元之间
            Random random = new Random();
            float balance = Math.abs(random.nextInt()) % 100000 / 100f;
            // 创建委派给一个 AccountImpl 对象实例的帐户纽带
            AccountPOATie tie = new AccountPOATie(new AccountImpl(balance));
            try {
                account = AccountHelper.narrow(accountPOA.servant_to_reference(tie));
            } catch(Exception exc) {
                exc.printStackTrace();
            }
            // 将帐户保存到帐户清单中
            accountList.put(name, account);
            // 在服务端控制台打印已创建新帐户的提示信息
            System.out.println("新开帐户: " + name);
        }
        // 返回找到的帐户或新开设的帐户
        return account;
    }
}
```

程序 5-12　采用纽带机制的 Server.java

```
// 服务端的主程序
import org.omg.PortableServer.*;

public class Server
{
    public static void main(String[] args)
```

```
    {
        try {
            // 初始化 ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // 取根 POA 的引用
            POA rootPOA = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            // 以持久对象的 POA 策略创建子 POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            POA newPOA = rootPOA.create_POA("BankTiePOA",
                rootPOA.the_POAManager(), policies);
            // 创建委派给一个 AccountManagerImpl 对象实例的纽带
            Bank.AccountManagerPOATie tie =
                new Bank.AccountManagerPOATie(new AccountManagerImpl(rootPOA));
            // 用对象标识"BankManager"激活对象
            newPOA.activate_object_with_id("BankManager".getBytes(), tie);
            // 激活 POA 管理器
            rootPOA.the_POAManager().activate();
            // 等待处理客户程序的请求
            System.out.println("帐户管理员 BankManager 已就绪 ...\n");
            orb.run();
        } catch(Exception exc) {
            exc.printStackTrace();
        }
    }
}
```

程序 5-12 重写了服务端的主程序。与程序 5-11 类似，服务程序创建对象引用时使用的伺服对象类型必须是 POATie，在创建 POATie 类型的实例时必须创建一个对象实现的实例，并指定为该纽带的代理对象。

第三章例子程序的客户端程序无需修改，因为服务端对象实现采用继承方式还是纽带方式对客户程序而言是透明的，客户程序仍采用同样的方式访问对象实现提供的服务。

§ 5.7 POA vs BOA

5.7.1 基本对象适配器

早期版本的 CORBA 规范定义了基本对象适配器（BOA）作为 ORB 产品提供的首选对象适配器。但是 BOA 直接调用操作系统的有关机制激活对象实现并与之交互，因而 BOA 依赖于特定的操作系统和编程语言，在不同 ORB 产品上基于 BOA 开发的对象实现可移植性较差。在 1998 年修订的 CORBA 2.2 版中引入了 POA，POA 一词中的“可移植”主要针对 BOA 而言。

所有对象适配器的主要作用都是创建 CORBA 对象，并将 CORBA 对象与真正执行服务的程序设计语言对象（即伺服对象）相关联。POA 围绕这一目标提供了比 BOA 更完善的机制，从而取代 BOA 成为 ORB 利用对象实现实例为请求服务的主要途径。应注意对象适配器只是服务端的概念，不会对客户程序产生影响。一个 CORBA 兼容的客户程序如果能够使用基于 BOA 的服务程序，那么它也应能够使用基于 POA 的服务程序。

5.7.2 POA 对 BOA 的改进

POA 对 BOA 的改进主要体现在以下几个方面：

(1) 由于 CORBA 关于 BOA 的规范不够充分，许多 ORB 产品供应商为提高 BOA 的可用性纷纷对 BOA 进行了不同的扩展。而 POA 则通过更完善的规范改进了 BOA，POA 本身利用 IDL 定义，比 BOA 更强调如何处理复杂问题。

(2) BOA 不足以处理大规模系统，而 POA 为实现大规模系统提供了更多的功能。服务程序可通过以策略驱动的自动方式处理多个客户程序的连接与请求，并且可充分利用 POA 管理生存期以及 CORBA 对象与伺服对象之间关联的能力，管理应用程序中的大量对象。

(3) BOA 并不是完全与位置无关的，当客户程序与实现对象在同一进程或不同进程（即远程访问）时，BOA 的语义会发生变化，而 POA 更能保持这两种情况的语义一致性。实际上对本地对象的 CORBA 调用仍会通过 POA 进行，这使得 POA 可统一地应用策略和服务决策。

思考与练习

5-1 简述在 CORBA 的 POA 体系结构下，客户端发出的请求如何一步步传递到最终提供服务的伺服对象。指明在上述请求传递的过程中，CORBA 规范为了支持灵活的服务端模型，允许程序员在哪些环节添加自己的干预，以及可以添加怎样的干预？

5-2* 本章伺服对象例子给出的方案在用于实际系统时有一个明显的缺陷，那就是在冻结对象时并没有将对象的状态保存，因此再次激活后账户管理员对象所开设的账户信息已经丢失，如何进一步改进此方案，使得能够保存并恢复对象的状态？

5-3 运行程序 5-9 所示客户程序时，服务器控制台会显示调用了三次适配器激活器的程序员自定义方法。细心的读者还发现如果不关闭服务程序再次运行客户程序时，服务器控制台却没有显示适配器激活器的方法被 POA 调用。为什么会这样？

第三部分 Java 企业版规范与 Java 企业版中间件

第 6 章　Java 企业版基础

§ 6.1 概述

6.1.1　Java 2 平台

Java 2 为从嵌入式环境、小型桌面环境到大型企业级平台的各层次软件系统开发提供了针对性的开发与运行环境支持。图 6-1 给出了 Java 2 平台的整体示意图：

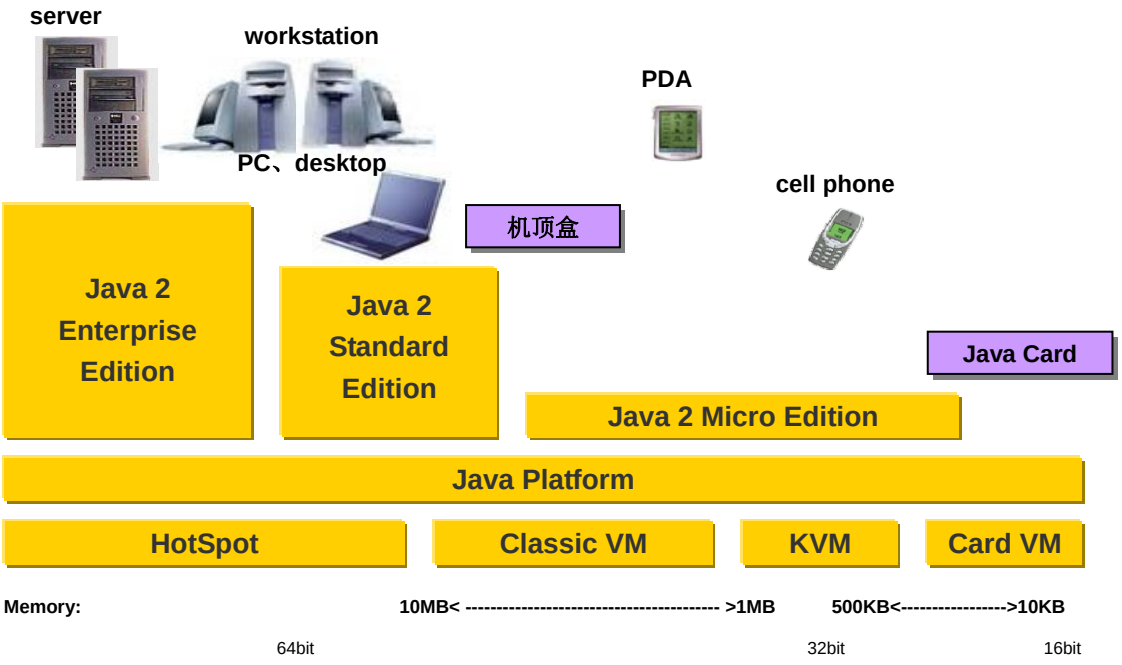


图 6-1　Java 2 平台示意图

由于 Java 是一种解释性语言，因此 Java 2 平台的最底层是面向不同环境的 Java 虚拟机（Vitural Machine），为相应环境下的 Java 应用程序提供基本的解释执行 Java 目标码的运行支撑。Card VM 是面向智能卡（java card）的虚拟机，它为智能卡定义了一组 Java 语言子集和虚拟机；典型的 Java Card 设备拥有 8 位或 16 位处理器，1M～5MHz 的运行频率，1.2KB 的 RAM 和 32KB 的 ROM；Card VM 仅支持最小的 Java 语言子集。比 Card VM 高一个层次的是 KVM，是面向手机的虚拟机（一般运行环境在 512K 以内）；现在全世界有上亿部手机中的应用程序是用 Java 语言开发的，这些应用程序的运行通常基于 KVM。Classic VM 是面向 PDA、机顶盒等设备的虚拟机。Hotspot 是面向企业级应用的虚拟机。从 Card VM、KVM、Classic VM 到 HotSpot VM，它们所基于硬件环境拥有越来越多的资源（更高速度的处理器、更多的内存），因此也支持越来越复杂、规模越来越大应用软件。读者应注意新版本的 Java 虚拟机结构有所变化，去掉了 Classic VM，另外早期版本中还存在一个 Exact VM。

基于底层的虚拟机支持，Java 2 平台为开发人员提供了三种不同版本的开发/运行环境，帮助开发、管理基于不同环境的 Java 应用。

- Java 2 Micro Edition（J2ME）：Java 程序设计语言最初是为消费电子设备编程设计的嵌入式语言，J2ME 就是 Java 2 平台针对基于消费电子设备（如手机、PDA、智能卡等）的应用开发所推出的平台。
- Java 2 Standard Edition（J2SE）：用来支持开发小型的桌面应用的 Java 平台，也就是通常所说的 JDK。
- Java 2 Enterprise Edition（J2EE）：即 Java 企业版，是用来支持大型企业级应用开发与运行的平台，以下简称 J2EE。

读者应注意上述三个版本是同时存在的，用来支持不同领域或规模的应用。

J2EE 的目标是用来支持以构件化的方法开发、部署、管理多层结构的分布式企业应用，尤其是基于 Web 的多层应用，J2EE 围绕这一目标提供了一种统一的、开放标准的多层平台。这一点和 CORBA 类似，J2EE 为开发人员提供的是一种工业标准，而不是某个厂商自己的产品，尽管 J2EE 规范是由 Sun 负责制定和发布的，但它确实是一套业界普遍支持和采纳的开放标准，规范本身也是由很多个大型厂商，比如 BEA、IBM 等来共同推动的。基于一种工业标准所开发的应用一般不会局限于某个厂商特定的应用服务器产品，开发人员无须额外的开销即可获得较高的可移植性。同时，遵循统一规范的不同产品之间的竞争环境也有利于提高产品的质量。

在 J2EE 规范中，容器的概念非常突出，J2EE 构件要依赖于容器所提供的系统级支持。与基于 J2ME、J2SE 的应用不同，基于 J2EE 的应用系统不能直接基于 Java 虚拟机运行，还需要容器构成的 Java 企业版平台的支持才能运行。容器为运行于其中的构件提供生命周期管理等大量的服务，支持部署人员将构建部署到其中，同时为构件的运行指派线程。

在 J2EE 中，许多构件的行为都可以在部署时以声明的方式进行定制，这是 J2EE 的一个很大的特色，就是说系统开发完后，部署时还可以在源代码以外的地方进行定制来配置构件使得它能够满足特定的应用需要，开发人员一次开发的构件可以通过定制去装配到不同需求的应用环境中。另一方面，当应用背景或需求发生变化的时候，很多情况下，可以不需要去修改源程序，当然更不需要重新编译，而是在源代码以外的地方进行定制就可以使得原有的构件适应新的环境或需求。

6.1.2 J2EE 平台技术

J2EE 平台为开发企业级应用提供三个方面的技术支持，如图 6-2 所示，包括应用构件、公共服务与通信三个方面。

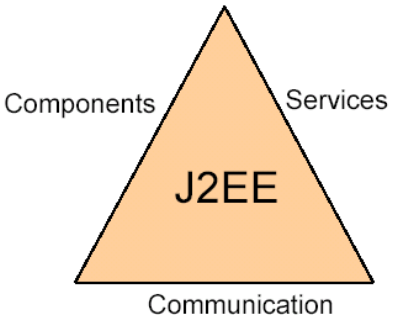


图 6-2 J2EE 平台提供的支持

第一个侧面就是**对应用构件或者组件的支持**，在 J2EE 中，应用构件是由开发人员实现，用来构成应用系统的软件模块，这和软件系统的一般意义上的构件含义相同；另外，J2EE

构件都是运行在 J2EE 平台之上的，更确切地讲，这些构件都是包含在某一类容器中的。J2EE 中的应用构件可分为两大类：客户端构件和服务端构件。客户端构件又包含 Applets 和 Application Clients；服务端构件又包含 Web 构件（JSP、Servlets）和 EJB（Enterprise Java Bean）构件，本章第 2 节将对每种 J2EE 构件进行简要的解释。

第二个侧面是**对服务的支持**，即第一章中提到的公共服务。具体到 J2EE 平台，服务是指 J2EE 应用构件所使用的功能，这些功能由 J2EE 平台提供商实现；这些和 CORBA 中类似，在 CORBA 中，开发人员也可以使用大量的服务，比如对象服务中包含的命名服务、安全性控制等，公共设施中包含的打印、电子邮件等，以及一些领域接口；这些服务也不需要程序员自己实现。但是在 J2EE 平台上服务有一个重要的区别，就是 J2EE 中除了像 CORBA 中那样以 API 的方式提供的服务外，还提供了丰富的运行时服务。J2EE 平台提供的服务，一部分我们可以像在 CORBA 中那样，通过 J2EE 规范约定的标准接口在程序中使用，还有一部分我们可以在源代码以外的地方通过声明的方式来使用，就是这些运行时服务。

J2EE 平台技术的第三个侧面就是**对通信的支持**，主要用来支持构件之间的交互，通信机制也是由容器来提供的。

6.1.3 J2EE 平台的执行模型

J2EE 典型地支持三层/多层分布式应用的开发与运行，图 6-3 所示的 J2EE 平台的执行模型描述了基于 J2EE 的应用系统在运行时的基本结构：

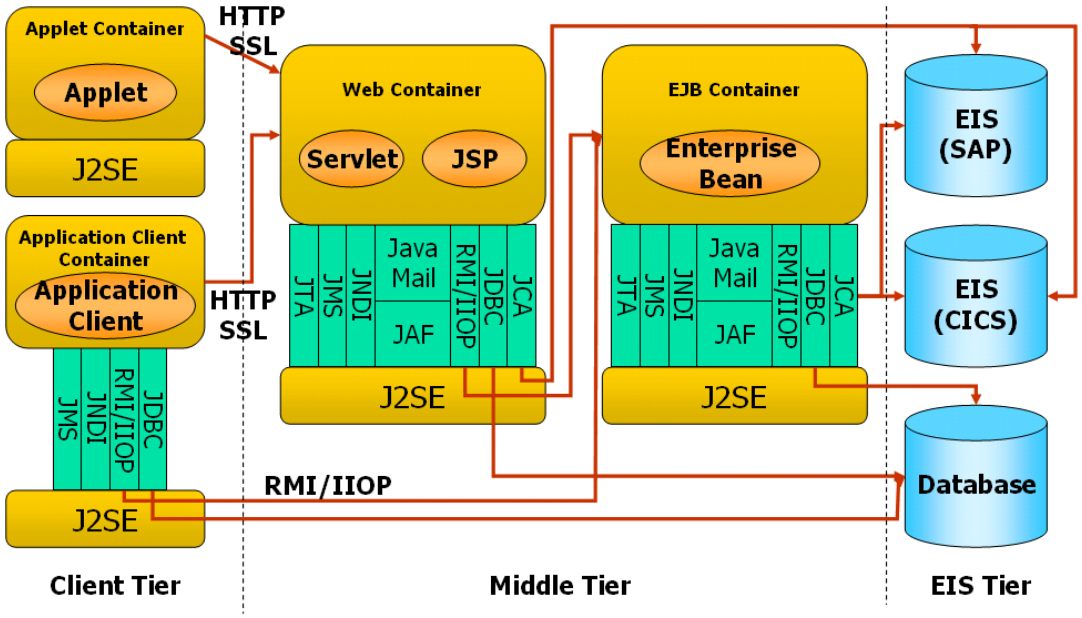


图 6-3 J2EE 平台的执行模型

首先，从构成系统的构件来讲，在客户层，可以包含 Applet 和 Application Client 两种构件；类似使用 Applet 的基于浏览器的客户端通常称为瘦客户端（thin client），因为这种客户端不需要在客户机器上安装任何应用代码，只需一个浏览器；而使用独立的客户端程序的客户端称之为胖客户端（thick client），因为其需要在每一个客户端机器上安装客户端程序。J2EE 中的胖客户端通常是一个独立的 Java 程序。在中间层，应用中可以包含像 servlet、JSP 这样的 Web 构件，也可以包含 EJB 构件。一般又会把 J2EE 应用的核心中间层划分成包含 Web 构件的 Web 子层和包含 EJB 构件的 EJB 子层。在数据层，包含应用使用的各种企业数据，这些数据，有些存储在数据库中，有些可能包含在企业的原有系统中，比如企业中可能已经有一套基于 SAP 的 ERP 系统，其中的数据要被新开发的这套 J2EE 系统使用。

其次，在 J2EE 平台中广泛使用构件/容器体系结构，即构成 J2EE 应用系统的构件都运行在某种 J2EE 容器中。比如，大而大家经常使用的浏览器就是一种常用的 Applet 容器。另外，在中间层，有两类非常重要的容器——web 容器和 EJB 容器，分别为 web 构件和 EJB 构件提供运行环境。在构件/容器体系结构下，将底层常用的、且通常是比较复杂的服务打包在容器中，由容器向其中的构件提供运行环境与公共服务的支持。容器由 J2EE 平台提供商提供，为构件提供特定的开发用服务和运行时服务（如生命周期管理、安全性管理、事务处理等），不同类型的 J2EE 容器提供不同的服务。

在 J2EE 平台中，每一个容器底层都需要 J2SE 的支持。容器除了为构件提供公共服务之外，还为构件之间的交互或者是构件访问后台数据提供了基本的支持，包括客户端与中间层 Web 构件或 EJB 构件的交互、Web 构件与 EJB 构件的交互、以及构件访问数据层数据等。

§ 6.2 J2EE 应用构件

本节对构成 J2EE 应用系统的构件进行简要的解释，本书重点关注其中的核心的 EJB 构件，其它构件的详细内容请读者参阅其它资料。

6.2.1 客户端构件

1. Applet

Applet 是具有图形用户界面的特殊的 Java 类，一般运行在 Web 浏览器中，也可以运行在支持 applet 编程模型的容器中（如 J2SE 中的 appletviewer）。在 J2EE 应用中 Applets 一般用来提供用户界面，图 6-4 给出了一个 applet 运行时的界面，当用户在 applet 提供的图形界面上操作时，可能会导致服务端的构件（比如 EJB）被调用。

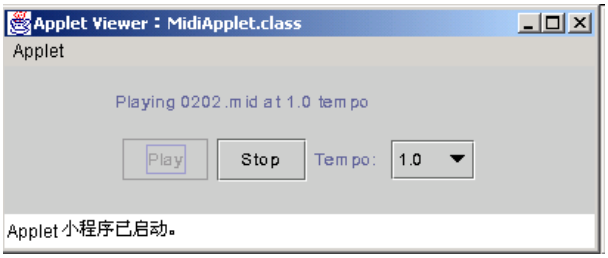


图 6-4 Applet 示例

与具有图形用户界面的独立 Java 程序不同，Applet 类自身不包含 main 入口函数，它被容器调度执行。Applet 代码不需要提前在运行的客户端机器上安装，而是由 applet 容器在运行时从服务器下载到本地执行。

2. Application Client

Application Client 指有图形用户界面的独立 Java 程序，在 J2EE 应用中所起的作用与 Applet 类似——用来提供用户界面。当用户在 Application clients 提供的图形界面上操作时，也可能会导致服务端的构件（比如 EJB）被调用。与 applet 不同，Application Client 通常包含 main 入口函数且需要在每个使用的客户端机器上安装。图 6-5 给出了一个 application client 运行时的界面。

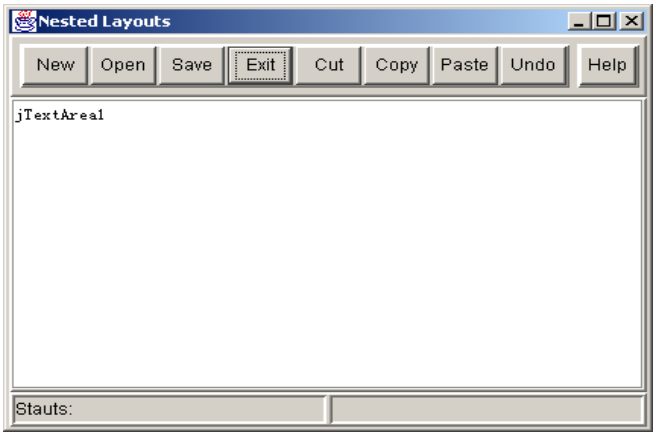


图 6-5 Application Client 示例

读者应注意很多 J2EE 应用中均不使用 application client 或 applet 作为客户端程序，客户端的用户界面一般由 web 页面来提供，要么是静态的 html 页面，要么是服务端的 web 构件动态生成的页面。这其中除了胖客户端需要在每个客户端机器上布署客户端程序的原因外，页面设计的快捷与美观也成为越来越重要的原因，同时基于浏览器的客户端只能访问中间层的 Web 构件，这使 J2EE 应用被划分成更为清晰的由客户层、Web 层、EJB 层和数据层构成的四层结构。

6.2.2 服务端构件

1. Web 构件——Servlet

Servlet 也是一种特殊的 Java 类，和 applets 不同，servlet 是运行在服务端的，因此它不需要图形用户界面。Servlet 在 J2EE 应用中的主要作用是接收客户端的 HTTP 请求（如通过浏览器发出的请求），动态生成 HTTP 响应（如一个页面）。程序 6-1 给出了一个处理 HTTP 请求的 servlet 的例子，该 servlet 的主要功能就是根据客户端的请求为其动态生成一个 HTTP 响应——HTML 页面。

程序 6-1 Servket 代码示例

```
//演示如何处理 HTTP GET 请求
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;

public class GetDemo extends HttpServlet{
    public void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException{
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        String username = req.getParameter("uname");
        String password = req.getParameter("userpw");
        out.println("<HTML>");
        out.println("<HEAD><TITLE>GetDemo</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("Hello " + username + "<br>");
        out.println("Your password was : " + password +"<br>");
        out.println("</BODY>");
        out.println("</HTML>");
        out.close();
    }
}
```

上面的 servlet 经过布署后可以被客户端通过浏览器访问，该 servlet 根据用户访问时提供的用户名（username）与密码（password）信息为其动态生成一个页面，该页面上包含用户提供的信息，该 servlet 的某次被访问时呈现的页面如图 6-6 所示。

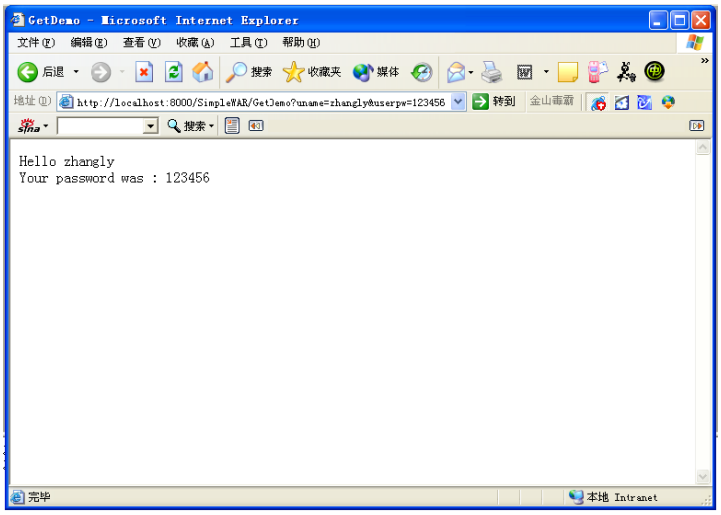


图 6-6 servlet 被访问时产生的动态页面

可以看出，servlet 为 J2EE 应用提供了基本的动态页面支持。

2. Web 构件——JSP

JSP 是一类特殊的 HTML 文档，它通过在 HTML 文档中嵌入 JSP 特定的标签来允许程序员在页面中加入 java 代码，通过 Java 代码的执行动态生成页面的内容。程序 6-2 给出了一个 JSP 的例子，该 JSP 在客户端访问时会在页面上显示服务器端的当前时间。

程序 6-2　JSP 代码示例

```
//演示 JSP
<html>
<head><title>Ch6 Simple jsp</title></head>
<body>
<%@ page import="java.util.*" %>
<h2>Date and Time: <%= new Date().toString() %></h2>
<hr>
<% for(int i=0; i<10; i++) %>
hello<br>
<% %>
how are u?<br>
</body>
</html>
```

在服务端，JSP 页面被编译成 servlet 执行。显然，使用 JSP 开发动态页面要比使用 servlet 方便，因为在 JSP 中，页面上静态的内容可以直接用 HTML 语言编写，从而可以利用现有的页面设计工具以可视化的方式完成，只有动态的部分才需要嵌入 Java 代码来实现。而有了 JSP，servlet 本身就通常不再完成具体的动态页面生成工作，在本章第 4 节会看到，servlet 通常会在 J2EE 应用中充当分发请求的控制器。

3. EJB 构件

EJB 是 Enterprise Java Bean 的缩写，通常是构成 J2EE 应用系统的核心构件，是用来支持大型的企业级应用开发的。在 J2EE 规范中，开发人员可以编写 3 种类型的 EJB：实体构件（Entity Bean）、会话构件（Session Bean）和消息驱动构件（Message Driven Bean）。在

CORBA 部分第 4 章第 6 节我们曾经提到过，在系统中经常会区分实体型接口和会话型接口，在 J2EE 中直接把这种思想纳入了规范本身，开发人员所开发的 EJB 就包含代表实际业务系统中的共享实体的实体构件和支持客户端交互的会话构件。其中实体构件又分为 CMP（Container Managed Persistence）构件和 BMP（Bean Managed Persistence）构件，会话构件又分为无状态会话构件（Stateless Session Bean）和有状态会话构件（Stateful Session Bean）。消息驱动构件是在 EJB 中用来支持异步模式编程的，这种构件通常不被客户端直接调用，而是当有事件发生，即有消息收到时才会执行相应动作，消息驱动构件是在 EJB 2.0 之后引入的。本书第 8 章将详细介绍 EJB 构件的开发与使用。

§ 6.3 J2EE 中的公共服务

J2EE 平台为应用构件提供公共服务与通信的基本支持，其中公共服务又分为通过 API 调用方式使用的 Service API 和通过配置方式使用的运行时服务（Run-time Service），本节对 J2EE 平台提供的常见 Service API 和运行时服务进行简要解释。

6.3.1 Service API

1. JNDI

JNDI 是 J2EE 平台提供的命名服务接口。JNDI 是 Java Naming and Directory Interface 的缩写，他为开发人员提供的主要功能是在程序中查找/定位构件或系统资源。比如，如果需要访问某个 EJB，可以利用 JNDI 服务找到要使用的 EJB。如果需要访问数据库，可以利用 JNDI 服务找到要访问的数据库资源。JNDI 服务使用的名字是一个带树型结构目录的名字，类似于 CORBA 中 POA 名字。

在 J2EE 规范推出之前就已经存在多种可以使用的目录服务支持我们完成类似的功能，比如基于 LDAP（Lightweight Directory Access Protocol，轻量级目录访问协议）的目录服务。如果开发应用时在程序中使用了某种特定的目录服务，那么一般情况下就很难直接迁移到另一种目录服务上，因为不同的目录服务所提供的接口是不一样的。而 JNDI 所提供的服务非常类似于一种通用的目录服务。如图 6-7 所示，基于 JNDI 的支持，在 J2EE 应用中可通过这种统一的 API 来使用目录服务的功能，它为应用屏蔽了不同目录服务之间的差异，这使得利用 JNDI 开发的应用可以不用修改代码而应用到不同的目录服务上。

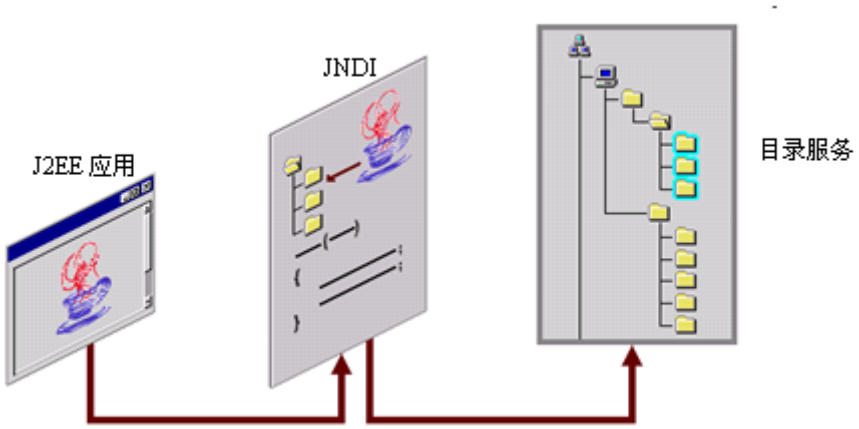


图 6-7 JNDI 服务

在 J2EE 中，大多数的 Service API 都具备这样的特征，这些 API 为应用使用这些服务提供了一个标准的接口，开发人员不用关心这些接口之下的实现会存在什么差异，只要按照接口的约定来使用这些服务。厂商实现这些 API 时可以任意选择，比如厂商可以选择将自己原有的目录服务器利用 JNDI 进行封装，为 J2EE 开发者提供目录服务。可以看到，这些 service API 和 CORBA 规范中标准化的接口起得是同样的作用：开发人员按照标准的接口来使用，厂商按照自己的方式来实现接口中的功能。

2. JDBC

JDBC 为应用提供与厂商无关的数据库连接，它通常提供一种通用的方法用来查询、更新关系型数据库表，并且把数据库操作的结果转化成 Java 的数据类型。

一个 JDBC 驱动支持应用跨数据库平台完成以下三件事情：

1）建立与数据库的连接；

2）向数据源发送查询和更新语句；

3）处理结果。

JDBC2.0 包含了一个内嵌的数据库连接池，数据库连接池是 J2EE 构件访问数据库的常用方式。在传统的 Java 程序中，应用需要访问数据库时，通常首先加载一个驱动程序，然后建立数据库连接，进行数据库操作，操作完成后再释放数据库连接。在这个过程中，建立和释放数据库连接的过程是非常耗时的，有时甚至比真正的数据库操作所占用的时间还多。而数据库连接池则为应用提供了一种更为快捷的数据库访问方式，在这种模式下，平台把数据库连接当作是一种系统资源来存放在数据库连接池中，平台初始化时会自动建好一些数据库连接，当应用需要操作数据库时，直接从池中获取一个建好的连接，用完后再把连接放回连接池供后续使用，这就为应用节省了建立数据库连接和释放数据库连接的过程，从而达到一个较高的访问数据库的效率。

3. JTA

JTA 是 Java Transaction API 的缩写，用来支持应用中的事务控制。事务通常包含一组操作的集合，事务管理支持可靠的服务端计算，这在很多关键系统中都是必需的。事务具有如下特性：

- 原子性：所有相关的操作必须全部成功；如果任何一个操作失败，则所有操作全部撤销。
- 一致性：一致性保证事务所作的任何改变不会使系统处于无效状态。
- 隔离性：并发的事务不会互相影响。事务访问的任何数据不会受其它事务所作的修改的影响，直到第一个事务完成。
- 耐久性：事务提交时，对数据所作的任何改变都要记录到持久存储器中。通常由事务日志实现。

而分布式事务可能会跨越多个组件、多台机器，并可能涉及多个数据源，因此分布式事务需要更复杂的控制机制。

JTA 是在 J2EE 应用中使用的一种支持分布式事务处理的标准 API。厂商可以选择自己合适的实现方式来提供事务控制的功能，比如在 websphere 中，利用一种支持两阶段提交的数据库驱动来实现 JTA。在 J2EE 应用中，除了在代码中利用 JTA 来控制事务外，还可以通过在配置的方式来实现事务控制，请参考本节后续的运行时服务。

4. JCA

JCA 是 J2EE Connector Architecture 的缩写。在基于 J2EE 开发一套新的应用系统时，经常需要在新的系统中去访问企业原有的旧系统，我们把这种系统称为外部信息系统（External

Information System，EIS）。企业原有系统各种各样，比如有基于 SAP 实现的信息系统，有基于 CICS 实现的信息系统，一般来说，要访问不同的系统，所用的手段和编写的程序是不一样的，因为不同的系统所提供的接口不一样。JCA 是在 J2EE 中支持访问不同信息系统的一种统一 API。如图 6-8 所示，基于 JCA，J2EE 应用可以使用统一的 API 访问不同类型的信息系统。

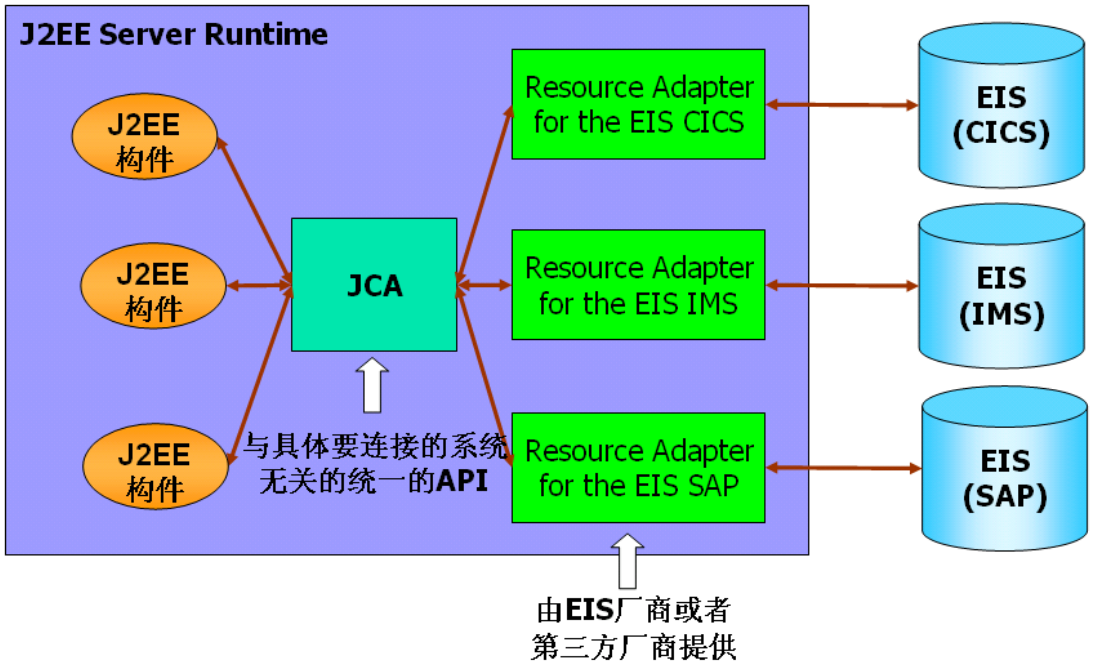


图 6-8 JCA 原理

基于 JCA 访问不同系统时，通常通过连接不同类型系统的适配器（Adapter）来屏蔽不同系统的差异，这些适配器一般由开发 EIS 的厂商或者第三方厂商提供。理想情况下，外部信息系统某种标准中间件开发，则可以期望由中间件厂商来实现该适配器。如果外部信息系统不是基于这些中间件开发的，更糟糕的是可能没有办法找到系统的开发者，相应的适配器就很难实现了。

5. Java Mail API

Java Mail API 定义了一组在程序中用来调用邮件功能的 API。这组 API 屏蔽了真正的邮件服务所使用的协议。Sun 提供了一种 Java Mail API 的实现，支持我们在应用中使用 Email 功能，读者可以从 Sun 网站上免费获得该 Java Mail API 的实现。

与其它 Service API 类似，Java Mail API 为 J2EE 应用提供了一种统一的 API，这种 API 使得 Email 可以在 Internet 中跨越平台和邮件协议传输。

6. Java IDL

Java IDL Service API 是用 Java 实现的 CORBA 规范，该 API 异类环境构件的互操作，允许在 J2EE 应用中访问 CORBA 构件。

6.3.2 运行时服务

1. 生命周期管理

生命周期管理主要指得是 EJB 构件的生命周期管理。该服务的主要含义是由容器来管

理运行于其中的构件的生命周期。EJB 规范统一规定了 EJB 生命周期管理策略，厂商实现 J2EE 平台时要按照规范实现生命周期管理功能，开发人员编写 EJB 构件时要注意构件的生存期特性。和 CORBA 相比，有了容器提供的生命周期管理服务，开发人员就不需要自己写服务程序了，容器会自动在需要的时候将 EJB 构件准备好，由窗口来决定什么时候实例化分布式对象以及实例化多少个对象。当然，这是两方面的，不编写服务程序就意味着开发人员放弃了对服务端程序的更多的控制力，如果容器提供的生命周期管理服务不能满足应用的某些特定需要，可能能添加的干预也很有限。

2. 事务控制

在 J2EE 中，除了利用前面提到的 JTA 来实现事务控制外，还可以将事务交给容器自动控制，而开发人员不用编写任何 Java 代码。容器会根据部署时指定的事务属性自动完成控制的事务。当然，容器控制的事务是有局限性的，只能是方法级的，即其维护的事务的粒度是方法。

容器维护的事务称为 CMT——Container Managed Transaction，对应地，开发人员在代码中通过 JTA 来控制事务的方式称为 BMT——Bean Managed Transaction。

3. 安全服务

J2EE 中的安全性控制分为两个级别：第一级是认证（authentication），就是首先要验证访问者的身份。经过身份认证后，第二级就是授权（authorization）控制，所谓授权就是判断特定的访问者是否具有在特定资源上执行某种操作的权限，比如，是否有权限访问系统中的某个 JSP 页面，是否有权限调用系统中某个 EJB 提供的特定方法等。

在 J2EE 中，实现安全性控制的常用方式是声明的方式。声明方式的安全性控制完全由容器自动控制，不需要程序员编写任何 Java 代码。开发人员主要通过以下配置工作来定制安全性控制的规则：

- 声明需要的安全性角色（security-role），并且把角色映射到实际的安全域中。这里的安全性角色是指为应用系统所指定的逻辑用户组，比如经理角色、职员角色等。在实际的安全域中，一般会有多个用户映射到同一个角色，比如我们可以为一个公司使用的应用系统建立一套用户管理系统，其中张经理使用的用户 managerzhang 映射到经理角色，李经理使用的用户 managerli 也映射到经理角色。同时还会有多个员工使用的用户映射到职员这一角色。
- 声明安全控制规则：通过这些规则告诉容器如何为应用实现安全性控制，比如我们可以通过规则告知容器“经理”角色可以执行“员工管理”Bean 的“辞退员工”方法，还告知容器“所有人（未经过身份认证）”可以访问商品信息页面等。

除了声明的方式外，我们还可以自己写代码来进行安全性控制。这种方式要结合声明的方式使用，还需要声明需要的安全性角色，然后在程序中根据读到的角色来控制它是否有权限执行请求的操作。

4. 持久性服务

持久性服务主要指实体构件（Entity Bean）相关的数据库操作。J2EE 中的实体构件主要用来实现数据库的访问，基于容器提供的持久性服务，开发人员可以实体构件相关的数据库操作交给容器来自动完成，这种方式称为 CMP——Container Managed Persistence；当然实体构件相关的数据库操作也可以由开发人员编写 Java 代码来完成，这种方式称为 BMP——Bean Managed Persistence。

5. 资源管理

资源管理是指由 J2EE 运行平台负责维护/管理应用所用到的系统资源，如数据库连接资源、线程资源等。比如前面提到的数据库连接池就是用来管理数据库连接这种系统资源的，类似的还有系统线程池，用来维护一定数量的已创建好的线程。

资源管理为应用带的好处主要有两个：

- 提高系统资源的利用率和效率：有限的资源通过平台管理起来在众多的构件之间共享，可以提高资源的利用率；另外资源管理为资源的使用作了一些预处理操作（如预先建好数据库连接等），可以有效提高资源的使用效率。
- 使得应用程序与具体的资源相对无关：比如应用使用不同的数据库管理系统时，程序员所写的代码是一样的，当使用的数据库管理系统发生变化时（比如由 DB2 变为 Oracle），不需要修改源程序，只要改一下资源的配置信息就可以实现。

6.3.3 通信支持

1. RMI/IIOP

RMI/IIOP 是基于 CORBA 中提出的 IIOP 协议实现的远程方法调用（Remote Method Invocation），主要是用于支持 EJB 构件客户端对 EJB 的远程调用。读者应注意 RMI/IIOP 与第 1 章例子中使用的 Java RMI 不同，Java RMI 是 Java 标准版提供的用来支持远程访问 Java RMI 对象的标准，该标准基于 JRMP（Java Remote Messaging Protocol）实现。Java RMI 对象与 EJB 都是用 Java 语言实现的分布式对象，它们之间最大的区别就是 EJB 可以使用 J2EE 平台提供的各种公共服务，而 Java RMI 对象不可以。

2. JMS

JMS 是 Java Messaging Service 的缩写，代表 J2EE 平台为应用构件提供的异步消息服务。JMS 定义了 Service。JMS 定义了一组统一的 API，支持在程序中发送和接收异步消息。和其它的 Service API 类似，不同的厂商根据 API 来实现异步消息处理机制，一般的做法是提供一个消息服务器。

§6.4 J2EE 应用开发

本节简要介绍 J2EE 应用程序的基本结构与 J2EE 应用的基本开发过程。

6.4.1 J2EE 应用程序

在 J2EE 规范下，开发者所开发的各种 J2EE 构件需要被组装成完整的 J2EE 应用才能部署到 J2EE 服务器上。如图 6-9 所示，应用构件通过组装被打包到一个 J2EE 应用文件（.ear 文件）中，然后布署到 J2EE 中间件平台上运行。在打包过程中，需要应用组装者编写一种特殊的组装配置文件——布署描述符（Deployment Descriptor，简称 DD）。

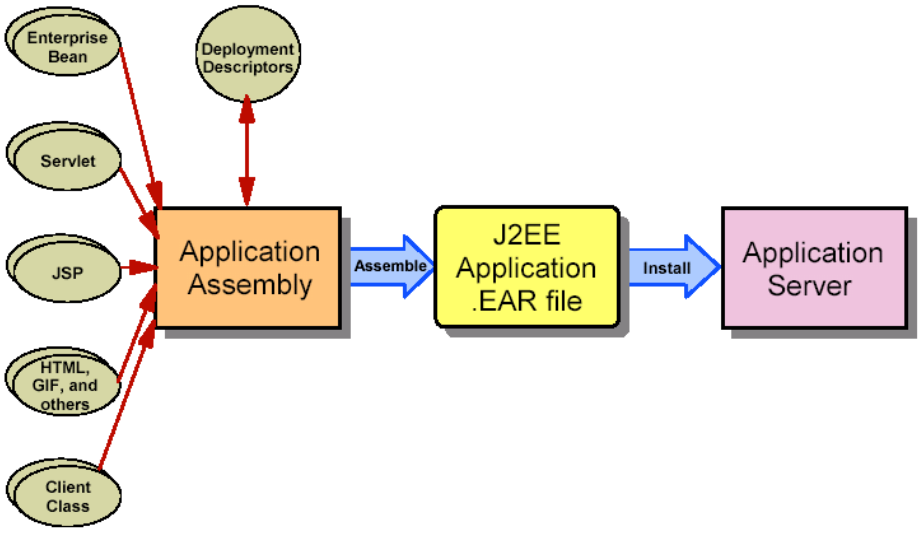


图 6-9 J2EE 应用打包与布署的基本过程

在 J2EE 中，开发者开发的不同构件会按照类型与层次被打包到不同的目标文件中，J2EE 平台上的目标文件主要有以下三种类型：

- Java 目标文件（Java Archive）：Java 语言的目标程序（.class 文件）包，对应磁盘上后缀名为.jar 的文件，在 J2EE 应用中用来打包 EJB 构件、Application Client 以及它们需要的辅助 Java 目标文件。
- Web 目标文件（Web Archive）：Web 构件目标程序包，对应磁盘上后缀名为.war 的文件，在 J2EE 应用中用来打包 Web 构件（Servlet、JSP）以及静态页面相关的文件（如 HTML 文档、图片等）。
- 企业目标文件（Enterprise Archive）：J2EE 应用目标程序包，对应磁盘上后缀名为.ear 的文件，在 J2EE 应用中用来打包完整的 J2EE 应用。一个完整的 J2EE 应用中可以包含若干个 Java 目标文件和若干个 Web 目标文件。

在打包 Java 目标文件、Web 目标文件与企业目标文件时都需要提供相应的布署描述符。

基于上述三种程序包，J2EE 应用程序的基本结构如图 6-10 所示，一个 J2EE 应用对应一个企业目标文件，其中可以包含若干个打包了 EJB 的 EJB 模块（一个 EJB 模块对应一个 Java 目标文件）、若干个打包了 Web 构件的 Web 模块（一个 Web 模块对应一个 Web 目标文件）和若干个打包了客户端程序的客户端模块（一个客户端模块对应一个 Java 目标文件）。

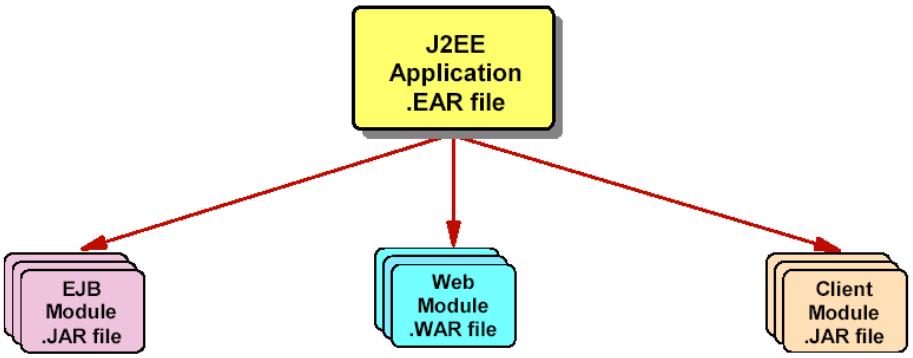


图 6-10 J2EE 应用程序的基本结构

将每个模块中包含的构件进一步细化，我们可以得到图 6-11 所示的 J2EE 应用程序的完

整结构：

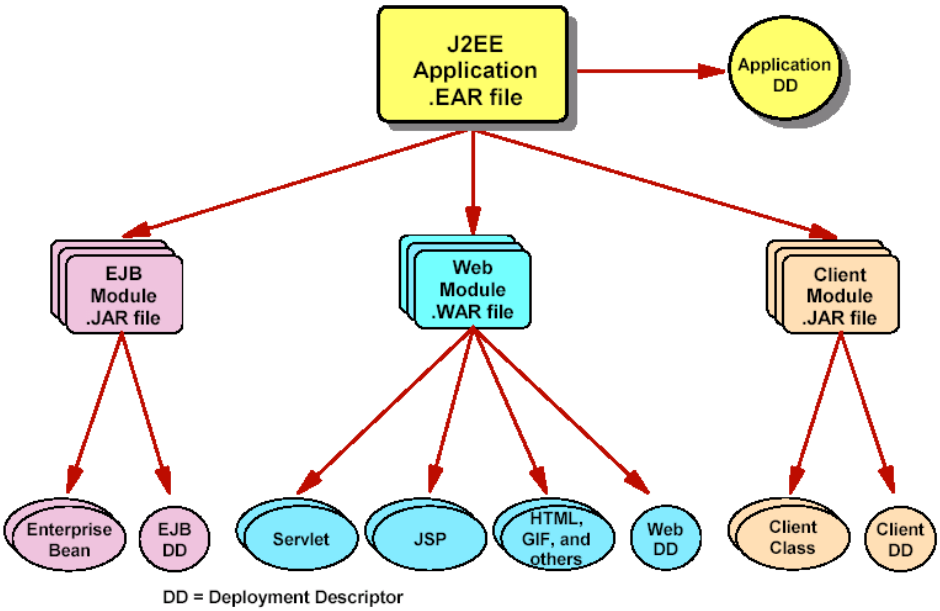


图 6-11 J2EE 应用程序的完整结构

每个 EJB 模块包含若干个 EJB 构件与一个当前模块的布署描述符；每个 Web 模块包含若干个 Servlet、JSP 构件，若干表态页面与相关图片等资源，以及一个当前模块的布署描述符；每个客户端模块包含若干个客户端程序使用的 Java 类与一个当前模块的布署描述符。除了每个模块包含一个布署描述符外，整个 J2EE 应用还包含一个布署描述符。

一个布署描述符是一个 XML 格式的文件，该文件中描述了当前模块中所包含的内容（构件或模块）和模块所需要的环境，J2EE 平台的丰富的源代码以外的可定制性大部分都体现在部署描述符中。每个模块或 ear 文件都有一个部署描述符。一般 J2EE 平台提供商会提供相应的工具来自动生成部署描述符，并且支持以可视化的方式进行编辑。因为部署描述符是一个 XML 格式的文件，所以可以手工创建和编辑，只要你熟悉部署描述符的格式。下面对常用布署描述符中的内容进行简要的介绍：

- **EJB 模块的布署描述符——ejb-jar.xml**

J2EE 中布署描述符的名称是硬编码的，即每种类型的目标文件都包含一个固定名字的布署描述符，EJB 模块的布署描述符的名称是 ejb-jar.xml。同一 EJB 模块中的所有 EJB 构件共享该模块的布署描述符，在该布署描述符中，对于每一个 EJB 构件描述了该 EJB 的 Home 接口、 Remote 接口以及真正提供服务的类的名字；如果是 Session Beans，说明是哪种类型的 Session Bean；如果是 Entity Bean，说明是否需要容器提供的持久性管理服务及相关的信息；说明该 EJB 是否需要由容器来控制事务，如果需要，怎样控制；说明该 EJB 的安全控制策略等等。

可以看出，EJB 构件的很多特性都是在部署描述符中确定的，比如如何管理持久性（也就是 Entity Bean 的数据库操作）、怎样控制事务、怎样控制安全性等这些信息都在部署描述符中有所体现。当这些相关的特性发生变化时，可以不修改 EJB 的源程序，而仅通过修改部署描述符就可以使得 EJB 去适应新的环境。

- **Web 模块的布署描述符——web.xml**

Web 模块的布署描述符的名称是 web.xml，同一 Web 模块中的所有构件共享该模块的布署描述符。Web 模块的布署描述符首先描述了当前模块包含的构件（包括 Servlet、JSP、表态页面等）；可以为当前模块中的构件说明安全控制规则；此外，J2EE 应用的 Web 模块

中经常需要配置应用使用的用户认证方式，因为在典型的 J2EE 应用中，客户端通常通过浏览器直接访问 Web 模块中的构件，Web 模块中的构件首先与访问者接触，因此通常在 Web 模块中完成对用户身份的认证。

- **J2EE 应用的布署描述符——application.xml**

J2EE 应用的布署描述符的名称是 application.xml。J2EE 应用的布署描述符描述了当前应用中包含的所有模块，此外还可能定义应用使用的安全性角色。

6.4.2 基于角色的开发过程

J2EE 规范将应用系统从开发到布署维护的生命周期映射到了 6 个角色：

- J2EE 平台提供者（J2EE Product Provider）
- 工具提供者（Tool Provider）
- 应用构件提供者（Application Component Provider）
- 应用组装者（Application Assembler）
- 布署者（Deployer）
- 系统管理员（System Administrator）

在这些角色中，一个角色的输出经常是另一个角色的输入，如 Application Assembler 将 Application Component Provider 开发的构件组装成应用，而 Deployer 则将 Application Assembler 组装的应用部署到运行平台上。同一个（组）人可能会执行两个或多个角色的任务，一个角色的任务也可能由几个（组）人完成。

J2EE 平台提供者的工作就是按照 J2EE 规范实现 J2EE 平台，如 IBM、BEA、Sun 等厂商，他们提供的主流 J2EE 平台包括 IBM 的 Webshpere Application Server、BEA 的 WebLogic Application Server、Sun 的 iPlanet Application Server 等，除非特别声明，本书第三部分的例子均基于 Sun 提供的免费 J2EE 平台——J2EE 参考实现（J2EE Reference Implementation）。

工具提供者的任务是为其他角色提供各种工具来帮助其完成 J2EE 平台上的任务，如 WSAD（Websphere Application Server）、Weblogic Builder、JBuilder、NetBean 等开发工具，用于组装 J2EE 应用的组装工具，用于管理、监控 J2EE 平台与应用的管理/监视工具等。

应用构件提供者的工作是按照应用需求实现构成应用的各种构件，另外还可能参与编写布署描述符。应用组装者的工作是将构件提供者提供的构件组装成应用。布署者的工作是将应用安装配置到运行环境中。系统管理员的工作比较繁杂，包括进行配置数据的备份、配置企业级应用、进行应用统计分析和性能分析、创建服务器组和服务器克隆以优化系统性能、管理并控制应用的安全性等。

6.4.3 J2EE 应用中的 MVC 设计模式

MVC（Modeling-View-Controller，模型-视图-控制器）设计模式明确划分了不同的构件在应用系统中的作用。如图 6-12 所示，在 J2EE 应用中，模型（Modeling），即系统的模型或系统基本的业务功能，通常由 EJB 构件实现；视图（View），即系统的人机交互界面，通常由 JSP 构件实现；控制器（Controller），即分发客户请求，决定每次客户端请求调用哪个 EJB 构件完成、结果由哪个 JSP 构件呈现的控制器通常由 Servlet 构件实现。在 MVC 模式下，J2EE 应用呈现更为清晰的四层结构：客户层、Web 层、EJB 层与数据层，客户通过浏览器发出的请求通常被 Servlet 构件接收，Servlet 调用合适的 EJB 构件完成客户请求，然后再将处理结果利用 JSP 呈现。此时，客户端只能访问 Web 构件，Web 构件不会直接访问数据库，J2EE 应用呈现出典型的四层结构。

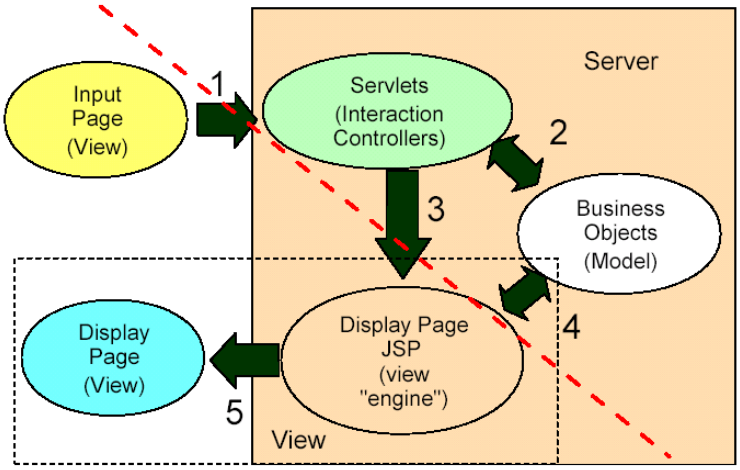


图 6-13 J2EE 应用中的 MVC 模式

思考与练习

6-1 J2EE 应用中构件/容器能够为应用构件提供哪些好处？

6-2 组成 J2EE 应用的应用构件主要有哪几种？每种应用构件在 J2EE 应用中的基本作用是什么？MVC 模式如何在 J2EE 中实现？

6-3 请简单描述 J2EE 应用程序的基本结构，说明布署描述符的主要作用。

第 7 章　EJB 构件基础

§ 7.1 EJB 体系结构

7.1.1　EJB 构件概述

1.　EJB 构件技术

EJB 规范采用的主要构件技术包括：

分布式对象技术：分布式对象技术提供客户端访问分布式对象的基本支持。除了 EJB 之外，其它的分布式对象技术包括 Java RMI、CORBA、DCOM 等，所有的所有的分布式对象技术都会使用某个特定的远程方法调用协议，EJB 中最常用的远程方法调用协议是 RMI/IIOP。不论其采用什么具体的远程方法调用协议，现有分布式对象技术一般均采用第一章提到的 Stub/Skeleton 结构来支持客户端与分布式对象之间的交互。

服务端构件技术：服务端构件技术用于中间层应用服务器，支持分布式商业对象的开发。服务端构件技术以组件方式提供系统的可重用性与可扩展性，随着实际商业系统的变化，可以重新组装、修改、甚至删除商业对象，而不必重新重写整个应用系统。

CTM（Component Transaction Monitor）技术：简单地说，CTM 是一个应用服务器，它为分布式商业对象提供公共服务框架，CTM 公共服务框架支持大量的系统级服务，如事务（Transaction）管理等。

综上所述，EJB 采用的构件技术为我们刻画了 EJB 构件的基本特征：EJB 构件是由公共服务框架自动管理的分布式的服务端商业构件，其中，CTM 技术提供了公共服务框架的支持，分布式对象技术提供了分布式对象的支持，而服务端构件技术提供了服务端构件管理的基本支持。

2.　EJB 构件的特点

作为 Java 企业版应用中的核心构件，EJB 构件除了具有一般软件构件的基本特征外，还具有以下主要特点：

公共服务框架：EJB 将实现商业对象所使用的服务框架的任务划分给了 EJB 应用服务器，服务框架支持大量的、由应用服务器提供的系统级服务。EJB 应用服务器提供的服务框架使得应用开发者可以关注于应用商业逻辑的实现，从而大大提高了开发效率，缩短了应用的开发周期。

平台独立性：平台独立性一方面得益于 EJB 沿袭了 Java 技术的平台无关性，另一方面，EJB/J2EE 规范的开放性使得构成 EJB 应用的商业对象可以移植到任何符合 EJB 规范的应用服务器上。

封装特性：EJB 规范提供对服务的封装特性(Wrap and Embrace)，封装特性使得 EJB 应用可以使用现有的基础性服务（如目录服务），EJB 通过定义一系列标准的服务 API 来封装现有的基础性服务，EJB 构件通过这些标准的 API 来使用服务。如 JNDI 接口支持访问现有的命名目录服务（如 LDAP、COS），通过使用 JNDI，EJB 应用服务器厂商可以将 LDAP 服务集成到其产品中，而不需去重新实现 LDAP。EJB 的封装特性可以使应用服务器厂商节省 IT 投资，厂商可以使用现有的基础性服务，而不必去重新实现这些服务。

可定制性：EJB 构件可以在不修改源代码的基础上进行定制化，EJB 构件的定制是指修

改 EJB 构件的运行时配置以满足特定用户的需求。EJB 构件的定制主要通过布署描述符完成，如可以利用布署描述符改变 EJB 构件的事务管理特性。除了 EJB 模块中标准的布署描述符 ejb-jar.xml 外，J2EE 平台提供商通常还会提供特定的辅助布署描述符以支持更强的可定制特性，如 Websphere 中的 ibm-ejb-ext.xmi、Weblogic 中的 weblogic-ejb-jar.xml 等。

协议无关性：该特性指 EJB 构件支持客户端通过多种 EJB 访问 EJB 构件。EJB 规范并没有强制约定只能通过 IIOP 协议来访问 EJB 构件，客户端可以通过其它通信协议，如 Weblogic 中支持的 t3 协议（一种基于 HTTP 的协议）来访问远程 EJB 构件。协议无关性使得 EJB 支持多种类型的客户端，不同类型的客户端使用不同的通信协议与 EJB 应用通信。

通用性：通用性指 EJB 规范可方便支持不同规模的应用系统，即可以在任何时间增加客户系统，而不需修改核心的应用系统。通用性通常意味着系统资源的可伸缩性，系统资源在软件构件处理客户请求时要使用到，如处理器(CPU)，随着系统规模的增大，构件可能要处理来自大量的客户请求，这里往往需要增加冗余的系统资源，如提供多个处理器（服务器），同时还需要一种合理的负载均衡机制将客户端请求均匀地分发到不同的服务器上。基于 EJB 容器的基本支持，Java 企业版平台（尤其是商用的平台）可以方便的支持不同规模的应用系统，如 Websphere 提供的克隆机制可以支持管理员将某个 J2EE 应用复制多份在多个服务器上运行，同时为多份服务提供自动的负载均衡机制。

3. 在 EJB 应用中集成遗产系统

遗产系统是现代企业信息流的主干，它体现了机构动作的业务逻辑，代表了特定领域的专业知识，是企业的财富，因此不能简单抛弃，通常需要在企业新开发的业务系统中实现遗产系统的集成。遗产系统通常是平台相关的，不能在网络环境中直接访问，并且遗产系统不能直接访问存储在各种数据库管理系统中的数据。因此一般说来，集成遗产系统的主要任务有两个，一是使得遗产系统成为可以在网络中访问的平台无关系统，另一个任务就是支持遗产系统访问各种数据库。

在 EJB 应用中，首先可以基于 JCA 实现在网络上访问遗产系统的目的，通过为遗产系统提供特定的适配器（Connector），可以在 EJB 应用中通过 JCA 来访问遗产系统。除了利用 JCA 外，由于 EJB 构件可以通过 Java IDL 接口访问 CORBA 构件，因此可以将遗产系统包装成 CORBA 对象供 EJB 应用访问，从而达到可以在网络上访问遗产系统的目的。

4. EJB 构件与 Java Bean 的比较

EJB 与 JavaBeans 都是基于 Java 语言的构件模型，开发应用时，可以选择 EJB 模型，也可以选择 JavaBeans 模型。EJB 与 Java Bean 的区别主要包括以下几点：

1）在大型的 Java 企业版应用中，EJB 构件通常用于服务端应用开发，而 Java Bean 构件通常用于客户端应用开发或作为服务端 EJB 构件的补充。当然也可以用 Java Bean 构件进行服务端应用的开发，但是与 EJB 构件相比，Java Bean 不能使用 Java 企业版平台提供的公共服务框架的支持，当应用需要使用关键的公共服务（如事务控制服务）时，使用普通的 Java Bean 构件显然不适合。

2）EJB 构件是可布署的，即 EJB 构件可以作为独立的软件单元被布署到 EJB 应用服务器上，是应用构件（application components）；而 Java Bean 是开发构件，不能被部署为独立的单元。

3）EJB 构件是布署时可定制的，开发人员可以通过布署描述符对 EJB 构件的运行时配置进行定制；而 Java Bean 构件的定制通常仅发生在开发阶段，开发人员只能利用开发工具创建并组装 JavaBeans 构件，部署时不能对其进行定制。

4）EJB 构件是分布式对象，可以被客户应用或者其它 EJB 构件进行远程访问；而普通的 Java Bean 构件只能在其构成的应用中使用，不能提供远程访问的能力。

5）EJB 构件是服务端构件，运行在服务端，没有人机交互界面，对终端用户不可见；而部分 JavaBeans 构件对终端用户可见，如 GUI 应用中使用的按钮构件等。

综上所述，从对比的角度看，可以认为 EJB 是 Java 语言提供的服务端构件模型，而 Java Bean 则是 Java 语言提供的客户端构件模型，具体开发时可根据应用的具体需求，结合 EJB 和 Java Bean 的特点，灵活的选择 EJB 或者 Java Bean。

7.1.2 EJB 体系结构中的构件

广义地讲，EJB 体系结构中涉及以下 6 类软件构件：

- Enterprise Java Bean（简称 Enterprise Bean）
- Home 接口
- Remote 接口
- EJB 容器
- EJB 容器
- EJB 客户端

1. Enterprise Bean

所谓 Enterprise Bean 就是开发者实现的核心构件 EJB，是 EJB 客户端所调用的操作的真正实现者，可以被部署到 EJB 应用服务器上，用来组装大型的 EJB 应用。开发人员可以开发的 EJB 构件包括会话构件（Session Bean）、实体构件（Entity Bean）和消息驱动构件（Message Driven Bean）。

Session bean 存在于客户应用与应用服务器交互的时间段内，是用来和客户端做交互的。和实体 Bean 相比，Session bean 中的数据不保存在数据库中。Session bean 又可分为两类，有状态的 Session Bean（Stateful Session Bean）和无状态的 Session Bean（Stateless Session Bean）。无状态的 Session bean 在方法调用中间不维护任何状态，一个无状态的 Session bean 实例可以同时处理多个客户应用的请求，典型地如网上证券系统中提供股票信息查询功能的 Session Bean。而有状态的 Session bean 要跨方法调用保存会话状态，一个有状态的 Session Bean 实例同时只处理一个客户应用的请求，典型地如网上购物系统中提供购物车功能的 Session Bean。

Entity Bean 代表数据库中的记录，在 EJB 中是用来封装数据库操作的，与 Session Bean 相比，逻辑上可以认为 Entity Bean 在数据库中的数据存在期间都会存在。同样与数据库中的数据类似，Entity Bean 可以被多个客户应用共享访问。

Message Driven Bean 主要用来处理异步消息，因此通常在异步编程模式下使用。Message Driven Bean 实现的方法不是被客户端直接调用的，而是当有异步消息发送到某个 Message Driven Bean，容器会调用它的的回调方法——OnMessage，构件实现者在 OnMessage 方法中实现对异步消息的处理。

2. Home 接口

Home 接口（Home Interface）包含 EJB 生命周期管理的相关的方法，客户程序使用 Home 接口创建或删除 EJB 的实例。读者应注意，严格意义上讲接口不能称作构件，因为接口中仅包含有方法的声明，而没有实现。这里将 Home 接口称为构件是因为在 EJB 体系结构中，每个 Home 接口都依赖于一个类（bean）来提供 Home 接口中约定的功能，只不过这个特殊的类不是程序员编写的，而是由容器自动生成的，开发人员只需要编写 Home 接口；同样对于 EJB 的客户端程序来讲，也只能看到 Home 接口，看不到这个实现类。

3. Remote 接口

Remote 接口中包含 EJB 实现的商业方法的声明，它实际上约定了 EJB 所提供的服务。这里说 Remote 接口是构件，与 Home 接口是类似的，容器也会自动生成一个 bean 来实现这个接口中的方法，但应注意容器生成的并不是真正的服务实现，真正的操作是由 EJB 构件实现的。在 EJB 中，客户程序只能通过 Remote 接口来间接地访问 EJB 实现的商业方法，不能直接进行调用。

基于以上三种构件，我们可以得到图 7-1 所示的 EJB 体系结构中的 Stub/Skeleton 结构。

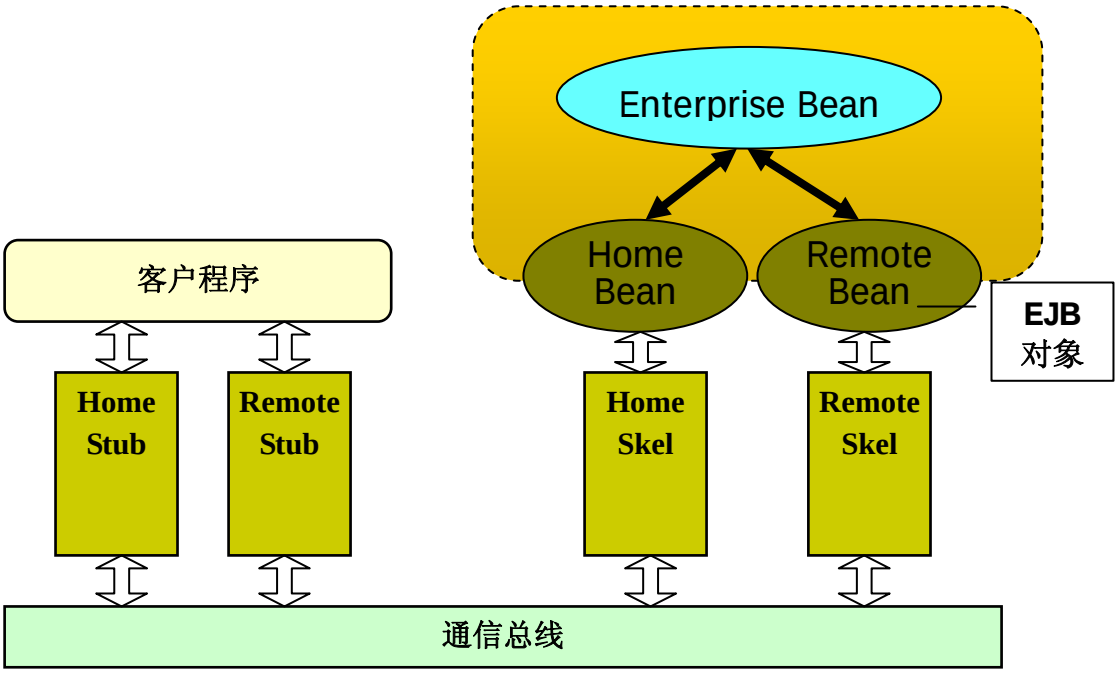


图 7-1 EJB 体系结构中的 Stub/Skeleton 结构

在客户端，客户程序通过两个接口来访问远程的 EJB 构件，通过 Home 接口调用 EJB 生命周期管理相关的操作，通过 Remote 接口调用 EJB 构件提供的业务逻辑操作，远程调用均通过 Stub/Skeleton 结构完成。读者应注意，基于 Stub/Skeleton 结构与客户端直接交互的并**不是** EJB 构件（图中的 Enterprise Bean），而是容器自动生成的实现 Home 接口中操作的类（Home Bean）和实现 Remote 接口中操作的类（Remote Bean），由于这两个类是容器自动生成的，因此从逻辑上看，客户程序与 EJB 构件之间每次交互均由容器充当了中介。之所以采用这种看起来比较复杂的结构，其中一个重要的原因是基于这种结构，容器可以利用中介的身份为 EJB 构件提供各种公共服务，如客户程序访问 EJB 构件时，容器可以根据布署描述符中配置信息检查客户端是否满足管理员定义的安全控制规则(如该客户是否有权限访问它正在调用的操作等)，如果不满足则不会允许其访问 EJB，而实现这种控制并不需要开发人员编写任何 Java 代码，只需在布署描述符中进行必要的配置就可以完成。

EJB2.0 之后引入了一对与 Home 接口、Remote 接口完成类似功能的本地接口——LocalHome 接口和 Local 接口。Local 接口完成与 Remote 接口类似的功能，包含 EJB 实现的商业方法的声明，LocalHome 接口完成与 Home 接口类似的功能，包含 EJB 生命周期管理的相关的方法。与 Home 接口和 Remote 相比，本地接口的不同之处在于客户应用通过本地接口发起的调用是进程内的本地调用，因此比远程接口调用有更高的效率，使用本地接口要求客户端和 EJB 在同一个进程（虚拟机）内，如一个 Session Bean 需要访问同一容器内的某个 Entity Bean 时，可以使用 Entity Bean 的本地接口发起调用以获得更高的执行效率。

每个 EJB 构件都有一对对应的 Home 接口与 Remote 接口和/或一对对应的 LocalHome 接口与 Local 接口，从这种意义上讲，我们通常认为一个完整的 EJB 构件包含 Enterprise Bean 类、Home 接口与 Remote 接口三部分。EJB 规范 3.0 对 EJB 对应的接口进行了一次简化，相关信息请参阅第 9 章第 4 节。

4. EJB 容器

EJB 容器为 EJB 构件提供运行环境并管理运行于其中的 EJB，理论上讲，一个 EJB 容器可以包含任何数量的 EJB，但是由于实际资源的限制，实际的 J2EE 平台的容器往往有一个能够包含 EJB 构件的上限。EJB 容器为 EJB 的执行提供系统级的服务，如自动将 EJB 相关的 Home 接口注册到一个目录服务中，自动注册服务支持客户应用查找定位 EJB 的实例。

5. EJB 服务器

EJB 服务器是遵循 EJB 定义的构件模型的 CTM 实现，一个 EJB 服务器可以包含一个或多个 EJB 容器，EJB 服务器为 EJB 容器的运行提供公共服务框架。公共服务框架支持系统级服务，如 JNDI 服务。从使用服务的角度来看，开发人员可以不区分 EJB 容器与 EJB 服务器，可以认为 EJB 容器和 EJB 服务器提供的服务都是由容器提供的。

6. EJB 客户端

EJB 客户端泛指调用 EJB 构件提供的业务操作的软件实体，EJB 构件的客户端可以有多种形式。如 EJB 的客户端可以是独立的 Java 程序，也可以是运行在 Web 容器中的 Servlet 或 JSP 构件，Servlet 或 JSP 形式的 EJB 客户端响应 Web 客户的请求；EJB 的客户端还可以是其它的 EJB，例如 Session Bean 经常作为 Entity Bean 的客户端来访问持久数据。

§7.2 EJB 设计原则

通过上节的讨论，我们知道每个 EJB 构件通常都有对应的一个 Home 接口与一个 Remote 接口，这三种构件在 EJB 应用中通常作为一个整体出现。在开发时，EJB 构件的实现类与对应的两个接口之间必须符合 EJB 规范所约定的对应关系与相关的设计原则。

7.2.1 接口设计原则

1. Remote 接口设计原则

Remote 接口约定了 EJB 构件提供的业务逻辑操作，编写 Remote 接口时应遵循以下原则或约束：

继承性约束：每个 Remote 接口必须继承 EJBObject 接口，该接口位于 javax.ejb 包，其中包含用于管理实现 remote 接口的 EJB 对象的方法，这些方法是每一个 EJB 对象都需要的。该接口在 EJB 规范中的作用相当于 CORBA 规范中 Object（参程序 3-3）接口的作用，用于约定特定体系结构下构件必须满足的特定约束。

方法对应规则：Remote 接口中出现的每一个方法的声明都必须在相应的 Enterprise Bean 类中有一个对应方法的实现，因为 Remote 接口是服务的约定，而 Enterprise Bean 类是服务的真正提供者，所以这一条约束是必然的。其中每个方法的参数和返回值必须完全相同，抛出的异常必须匹配。读者应注意这里是匹配，而不是相同，匹配的含义是指接口中方法抛出异常的集合必须包含 Bean 类中对应方法抛出异常的集合。即接口方法中出现的异常，Bean 类中可以出现，也可以不出现，但是不允许 Bean 类中方法抛出接口对应方法中没有声明的异常。这一点与 CORBA 或者 Java RMI 不同，方法的声明和方法的实现在接口上可以不完

全匹配，因为 EJB 的客户端通过 Remote 接口方法的不直接是 Enterprise Bean 类，而是容器生成的一个类（Remote Bean），每次客户端请求都由容器来充当中介。需要注意的是，在 Java 程序设计语言的层次上并没有提供机制来保证这一点，也就是说在定义 Enterprise Bean 类时，并不声明该类实现（implements）Remote 接口。即使编写的 Enterprise Bean 类与对应的 Remote 接口不符合该规则的约定，也可能可以单独利用 javac 编译通过，但是将 Enterprise Bean 类、Home 接口以及 Remote 接口打包成完整的 EJB 构件时会进行相应的检查。

RMI 约束：Remote 接口中定义的方法应该遵循 Java RMI 的约束。因为 EJB 对象是分布式对象，所以 Remote 接口中约定的方法必须是可远程访问的，为了支持远程访问，定义 Remote 接口时必须遵循 Java RMI 标准。具体的规则包括 Remote 接口中的方法必须抛出 RemoteException 异常，该异常报告网络通信错误；方法定义中的参数与返回值必须是合法的 Java RMI 类型的参数/返回值，因为只有合法的 Java RMI 类型的参数/返回值才可以在网络中传递，如可串行化对象（实现了 java.io.Serializable 接口）等。

2. Home 接口设计原则

Home 接口中包含 EJB 构件生命周期管理相关的方法，支持客户应用创建、删除或定位 Enterprise Bean 的实例，编写 Home 接口时应遵循以下原则或约束：

继承性约束：每个 Home 接口必须继承 EJBHome 接口，EJBHome 接口位于 javax.ejb 包，其中包含了 Enterprise Bean 生命周期管理的方法，Home 接口类似于 CORBA 体系结构中的工厂概念，开发人员可以在接口的基础上补充自定义的方法。

方法对应规则：EJB 规范要求 Home 接口中的每个 create 方法都必须在相应的 Enterprise Bean 类中有一个对应的 ejbCreate 方法，函数接口必须相同，返回值要符合相应的匹配规则，抛出的异常必须匹配，匹配的含义与 Remote 接口方法对应规则中的匹配含义相同，即 create 方法抛出的异常的集合包含 Bean 类中对应 ejbCreate 方法抛出异常的集合。EJB 构件的 create 方法支持客户端获取可用的 EJB 实例的引用，客户程序调用 create 方法时，EJB 容器会调用相应的 ejbCreate 方法完成 Enterprise Bean 实例的初始化。同样在 Java 程序设计语言的层次上并没有提供机制来保证这一点，将 Enterprise Bean 类、Home 接口以及 Remote 接口打包成完整的 EJB 构件时会进行相应的检查。

RMI 约束：Home 接口中约定的操作也是被远程调用的，因此 Home 接口中声明的每个操作也必须符合与 Remote 接口的操作相同的 RMI 约束，即必须抛出 RemoteException，参数/返回值必须合法的 Java RMI 类型的参数/返回值。除此之外，EJB 规范还要求 Home 接口中的每个 create 方法必须抛出 CreateException 异常，该异常用于报告 EJB 实例的初始化错误。

7.2.2 类设计原则

1. Enterprise Bean 类设计原则

Enterprise Bean 类包含客户端调用的业务逻辑操作的真正实现，是程序员编写的 EJB 构件的核心代码，编写 Enterprise Bean 类时应遵循以下原则：

接口约束：Enterprise bean 类必须实现 EnterpriseBean 接口，EnterpriseBean 接口是 java.io.Serializable 接口的子接口。EnterpriseBean 接口中定义了 Enterprise Bean 生命周期管理的方法，实现该接口是 Enterprise Bean 与普通 java bean 的重要区别。开发人员应该了解不同类型的 EJB 构件的生命周期特性，需要了解 EnterpriseBean 接口中相关生命周期管理方法的调用时机。

可见性约束：Enterprise bean 类必须定义为 public 类。由于 EJB 客户程序不要能直接访

问 Enterprise Bean 类实现的方法，客户程序的请求被 EJB 容器生成的 EJB 对象接收到，由 EJB 对象根据客户请求调用 Enterprise Bean 实例的对应方法。Enterprise Bean 类定义成 public 类以允许其它的类，如容器生成的 EJB 对象，访问 Enterprise Bean 类中定义的方法。

商业方法约束：Enterprise bean 类必须实现 Remote 接口中定义的业务逻辑操作。具体的对应规则请参照 7.2.1 中的 Remote 接口设计原则。

生命周期管理方法约束：Enterprise Bean 类必须实现 Home 接口中定义的 create 方法对应的 ejbCreate 方法。ejbCreate 方法完成 Enterprise Bean 实例的初始化，客户端调用 Home 接口中定义的 create 方法时，通常会导致 EJB 容器生成的 EJB Home 对象调用 Enterprise Bean 实例的对应 ejbCreate 方法，客户程序在调用 create 方法时传递的参数被用来调用 ejbCreate 方法。

2. 主键（Primary Key）类设计原则

在 EJB 规范中，Entity Bean 代表数据库中的记录。与数据库中的记录由主键唯一标识相相对应，Entity Bean 实例由主键类唯一标识。主键类对应数据库中的主键字段，相当于提供了一个指向数据库中记录的“指针”。编写主键类时应遵循以下原则：

可见性约束：主键类必须定义为 public 类。定义成 public 类以使得 Primary Key 所标识的 Entity Bean 总可以访问数据库表中的字段。

RMI 约束：因为 primary key 类也是要供远程调用的，所以定义时也要遵循类似的 RMI 约束。

思考与练习

7-1 在 EJB 体系结构中，Home 接口与 Remote 接口的主要作用是什么？为什么还要引入 LocalHome 与 Local 接口，这对接口使用时有哪些限制？

7-2 EJB 构件与普通的 Java Bean 有哪些主要区别？他们分别适合于什么场合的开发？

7-3 EJB 体系结构中基于 Stub/Skeleton 结构与客户端交互的直接构件不是 EJB，而是容器自动生成的对象，采用这样的结构有什么好处？

第 8 章　EJB 构件开发

§ 8.1 Java SDK

8.1.1　安装

在安装 Java 企业版平台之前，必须安装一个合适版本的 JDK，因为我们在 J2EE 执行模型中看到，每种 J2EE 容器都需要 Java 标准版 J2SE，即 JDK 的支持。Java 标准版通常包含两部分内容：一个标准版的开发包——J2SDK（Java Software Development Kit），和一个标准版的运行环境——J2RE（Java Runtime Environment）。

J2SE 为标准的 Java 应用或 applet 提供标准的 Java 服务，而 Java 的企业级 API 是由 J2EE 平台实现的，要开发、运行 Java 企业版应用，必须安装 J2EE 平台。本书例子相关的 J2EE 平台采用 Sun 提供的 J2EE 参考实现（Reference Implementation）。

读者应注意 J2EE 平台与相应的 J2SE 之间通常存在版本的约束，本书例子所使用的 J2EE 参考实现与 J2SE 的版本均为 1.3.1。相关软件的安装过程均比较简单，读者可全部采用缺省设置即可完成安装。

8.1.2　环境变量配置

安装完 Java 平台后，必须正确设置相应的环境变量才能正常使用。通常需要配置的环境变量包括：

- JAVA_HOME：在 Java 平台中，JAVA_HOME 环境变量用来指明包含 J2SE 类和配置文件的目录，通常是 J2SE 的安装目录，如 C:\jdk1.3.1。J2EE 平台通过该环境变量查找需要使用的可执行文件，如 Java 语言编译器 javac 和 Java 语言解释器 java 等。
- J2EE_HOME：在 Java 平台中，J2EE_HOME 环境变量用来指明包含 J2EE 类和配置文件的目录，通常是 J2EE 的安装目录，如 C:\j2sdkee1.3.1。J2SE 与 J2EE 平台都需要使用 JAVA_HOME 环境变量，而 J2EE_HOME 环境变量只有 J2EE 平台才需要。
- PATH：J2SE 与 J2EE 包含的可执行文件通常放在对应安装目录下的 bin 子目录中，为方便使用 Java 平台相关的可执行文件，如 javac、java、j2ee 等，通常需要将 J2SE 与 J2EE 安装目录下的 bin 子目录添加到系统的 PATH 环境变量中。
- CLASSPATH：CLASSPATH 是 Java 平台在编译或运行 Java 程序使用的环境变量，该环境变量列出了包含编译过的 Java 代码（.class 文件）的目录和 Java 目标文件包（.jar 文件），缺省情况下，Java 虚拟机会在 CLASSPATH 指明的目录和 Java 目标文件包中搜寻编译或运行 Java 程序所需的编译过的 Java 代码。通常需要在 CLASSPATH 环境变量中包含 J2SE 和 J2EE 安装目录下 lib 子目录中的 Java 目标文件（如 J2SE 中的 tools.jar、dt.jar，J2EE 中的 j2ee.jar 等），此外 CLASSPATH 中最好包含当前路径。
- JAVA_FONT：在 Java 平台中，JAVA_FONT 环境变量用来指明包含应用所使用的字体的目录，Java 平台使用该环境变量查找要使用的字体文件。

Windows 操作系统中的环境变量通常在“控制面板\系统\高级\环境变量”中进行添加与修改，并且环境变量的修改会自动在新打开的命令行窗口中生效。Linux 操作系统中的环境

变量通常的\etc\profile 配置文件或当前用户的 profile 配置文件中进行添加与修改，并且环境变量的修改会自动在用户下次登录时生效。

8.1.3 启动与关闭 J2EE 参考实现

J2EE 参考实现是 Sun 提供的免费 J2EE 平台，为构建、部署、测试 J2EE 应用提供运行时环境。J2EE 参考实现可采用命令行方式进行启动与关闭，使用的命令为 j2ee（对应可执行文件在 J2EE 参考实现安装目录下的 bin 子目录中）。命令 j2ee 可使用以下几个命令行参数：

- -verbose：将系统日志重定向到当前命令行窗口，缺省情况下系统日志会输出到文件中。
- -help：输出帮助信息。
- -version：输出版本信息。
- -singleVM：将所有的 bean 部署到同一个虚拟机（进程）中。
- -multiVM：将同一个 JAR 文件中的 bean 部署到同一个虚拟机中，不同 JAR 中的 bean 部署到不同的虚拟机中。
- -stop：关闭 J2EE 服务器。

§8.2 开发与使用无状态会话构件

会话构件是一个基于 EJB 的软件系统业务逻辑功能的主要实现者，除了数据库的相关操作，系统中的基本功能通常均由会话构件实现，会话构件又分为无状态会话构件与有状态会话构件。

本节通过一个简单的例子演示如何开发与使用无状态会话构件，例子程序分为客户端与服务端两部分，在服务端，我们将构建一个名为 CurTimeApp 的 J2EE 应用，该应用中仅包含一个无状态会话构件，该构件实现返回服务端当前系统时间的功能；客户端通过远程接口调用构件上的操作，将结果输出。

8.2.1 开发 EJB 构件

该 EJB 构件实现返回服务端当前系统时间的功能，由于该构件的实例（对象）不需要保存与特定客户端相关的会话状态，因此设计为无状态的会话构件。以下简称该 EJB 构件为时间 EJB。

1. 定义 Remote 接口

Remote 接口包含 EJB 构件实现的商业方法的声明，客户端只能通过 remote 接口访问构件实现的商业方法，不能直接调用。程序 8-1 给出了时间 EJB 的 Remote 接口定义：

程序 8-1 时间 EJB 的 Remote 接口定义

```
import javax.ejb.*;
import java.rmi.*;
public interface CurTime extends EJBObject
{
    String getCurTime() throws RemoteException;
}
```

在程序 8-1 中，我们定义了一个名为 CurTime 的 Remote 接口，时间 EJB 构件仅向客户端提供一个商业方法——getCurTime，Remote 接口中包含该方法的声明，客户端调用该方法时不需要提供任何参数，返回值为一个包含当前系统时间的字符串。可以看到，按照 EJB 规范的约定，接口 CurTime 继承了接口 EJBObject，操作 getCurTime 抛出 RemoteException

异常以报告远程调用错误，其返回值为合法的 Java RMI 类型。

2. 定义 Home 接口

Home 接口中包含 EJB 构件生命周期管理的相关方法，客户程序使用 Home Interface 创建、查找或删除 EJB 的实例。程序 8-2 给出了时间 EJB 的 Home 接口定义：

程序 8-2　时间 EJB 的 Home 接口定义

```
import javax.ejb.*;
import java.rmi.*;
public interface CurTimeHome extends EJBHome
{
    public CurTime create() throws RemoteException, CreateException;
}
```

在程序 8-2 中，我们定义了一个名为 CurTimeHome 的 Home 接口，按照 EJB 规范的约定，接口 CurTimeHome 继承了接口 EJBHome。该接口中仅声明了一个 create 方法，create 方法在 EJB 规范中用于取代传统面向对象中的构造函数来初始化一个 EJB 实例。由于会话构件的 create 方法会为调用者准备好一个 EJB 构件的实例，因此 create 方法的返回值必须是对应 EJB 构件的 Remote 接口类型，以支持客户端调用 EJB 构件上的商业方法，本例中返回的是程序 8-1 中定义的 Remote 接口 CurTime。按照 EJB 规范的约定，create 方法必须抛出 RemoteException 和 CreateException 异常，RemoteException 表明发生了网络错误，而 CreateException 异常可能由 EJB 构件抛出，也可能由 EJB 容器产生，该异常通知客户端不能创建 EJB 对象。

由于无状态会话构件的对象可能被多个客户端共享地访问，因此 EJB 规范不允许某个客户端使用特定的参数初始化无状态会话构件的对象，进而使得无状态会话构件 Home 接口中只能包含没有参数的 create 方法。

3. 定义 Enterprise Bean 类

在 EJB 中，remote 接口中所定义的商业方法由 Enterprise Bean 类实现，定义好 Remote 接口和 Home 接口后，就可以定义相关的 Enterprise Bean 类。Enterprise Bean 类首先要按照 Remote 接口的约定实现商业方法 getCurTime，其次要实现 Home 接口中 create 方法对应的 ejbCreate 方法与会话构件生命周期相关的方法。程序 8-3 给出了时间 EJB 的 Enterprise Bean 类定义：

程序 8-3　时间 EJB 的 Enterprise Bean 类定义

```
import javax.ejb.*;
import java.sql.*;
public class CurTimeBean implements SessionBean
{
    SessionContext Context;
    public String getCurTime()
    {
        Timestamp ts=new Timestamp(System.currentTimeMillis());
        return ts.toString();
    }
    //无状态的 session bean 只能包含无参数的 create
    public void ejbCreate(){
        System.out.println("\n\n\n******CurTimeBean ejbCreate");
    }
    public void ejbRemove(){
        System.out.println("CurTimeBean ejbRemove");
    }
    public void ejbPassivate(){}
    public void ejbActivate(){}
```

```
    public void setSessionContext(SessionContext Context)
    {
        this.Context = Context;
    }
}
```

在程序 8-3 中，我们定义了一个名为 CurTimeBean 的 Enterprise Bean 类，由于该 EJB 构件是会话构件，因此 Enterprise Bean 类实现（implements）SessionBean 接口，如果要开发的 EJB 构件是实体构件，对应的 Enterprise Bean 类则需要实现 EntityBean 接口，SessionBean 接口与 EntityBean 接口都是 EnterpriseBean 接口的子接口。

Enterprise Bean 类首先按照 Remote 接口 CurTime 的约定实现了商业方法 getCurTime，该方法将当前系统时间转换为字符串返回。

其次，Enterprise Bean 类实现了 Home 接口 CurTimeHome 中的 create 方法对应的 ejbCreate 方法，容器调用 ejbCreate 方法完成 EJB 对象的初始化。

此外，Enterprise Bean 类还实现了四个会话构件生命周期管理相关的其它方法：ejbRemove、ejbPassivate、ejbActivate 与 ejbSetSessionContext，这些方法是接口 SessionBean 约定的。其中 ejbRemove 会在容器删除 EJB 实例之前被调用，因此该方法类似于传统面向对象中的析构函数；ejbPassivate 与 ejbActivate 方法是有状态会话构件使用的两个方法，我们会在下一节中讨论；setSessionContext 方法用来初始化 EJB 使用的 SessionContext 变量，SessionContext 是会话构件与容器交互的入口，每次创建一个会话构件的对象时，容器会调用该对象的 setSessionContext 方法，给对象传入使用的 SessionContext 变量。

从上面的讨论可以看出，与普通的 Java 类相比，Enterprise Bean 类中多出了 ejbCreate、ejbRemove、ejbPassivate、ejbActivate、setSessionContext 等 EJB 生命周期管理相关的方法，这些方法会在容器管理 EJB 对象的过程中被调用，只有在了解了这些方法的调用时机的基础上，才能决定在这些方法中应该完成什么样的工作，因此下面首先对无状态会话构件的生命周期特征时行讨论。

如图 8-1 所示，无状态会话构件的生命周期包含两个状态：方法就绪状态（Method Ready State）与不存在状态（No State）。不存在状态表明 EJB 容器中不存在对应无状态会话构件的实例，处于不存在状态的实例还未被创建；方法就绪状态表明对应无状态会话构件对象已被创建，可以为客户端提供服务。

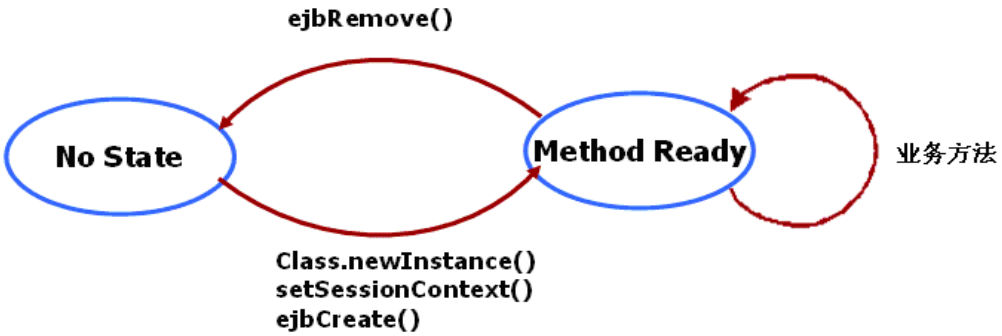


图 8-1 无状态会话构件生命周期

只要 EJB 容器认为实例池中需要更多的实例为客户端服务，就会创建新的实例，此时新创建的实例从 No 状态进入方法就绪状态；如果 EJB 容器认为当前已无需这么多的实例为客户端服务，就会根据某种策略删除池中的一些实例，此时被删除的实例又从方法就绪状态进入 No 状态。读者应注意对于无状态的会话构件来说，实例的创建和删除都是由容器自动

来控制的，并不是客户端每调用一次 Home 接口中的 create 都会创建一个新的实例，容器也不允许客户端调用 Home 接口中的 remove 方法来删除实例。因为无状态会话构件不保存与特定客户端相关的会话状态，其实例可以被多个客户端共享，因此容器会尽可能使用少量的实例为多个客户端提供服务。

从 No 状态进入方法就绪状态（创建新实例），容器会调用 Enterprise Bean 类对象的 setSessionContext 和 ejbCreate 方法（不是每一次客户端调用 home 接口中的 create 都会导致容器调用 Enterprise Bean 类对象的 ejbCreate 方法）；从方法就绪状态进入 No 状态（删除实例），会调用 ejbRemove 方法（容器不允许客户端调用 Home 接口中的 remove 方法来删除无状态会话构件的实例）。

4. 编译源代码

编写完 Remote 接口、Home 接口与 Enterprise Bean 类之后，就可以用 Java 语言编译器 javac 对 EJB 的源代码进行编译，编译时仅是将接口定义和 Enterprise Bean 类的定义进行 java 源程序到 java 目标代码的编译，不会检查 Remote 接口、Home 接口、Enterprise Bean 类之间的对应关系，对应关系在打包 EJB 时会检查。

8.2.2 打包/布署 EJB

1. 打包 J2EE 应用

EJB 构件不能够直接部署到 J2EE 服务器上，必须首先创建一个 J2EE 应用，然后将要部署的 EJB 添加到 J2EE 应用中，在把 J2EE 应用部署到服务器上。本节使用 J2EE 参考实现提供的组装/部署工具 deploytool[1]来创建 J2EE 应用。工具 deploytool 位于参考实现安装目录下的 bin 子目录中，如果仅是组装应用，可以单独启动 deploytool，如果需要部署应用，则启动 deploytool 之前需要首先启动 J2EE 服务器，可使用命令以命令启动 J2EE 服务器：

```
prompt> j2ee  -verbose
```

如图 8-2 所示，点击 deploytool 的 File\New\Application 菜单可创建一个空的 J2EE 应用，在图 8-3 所示的对话框中输入应用对应企业目标文件的文件名与应用的名字即可完成应用的创建，这里我们将应用命名为 CurTimeApp。

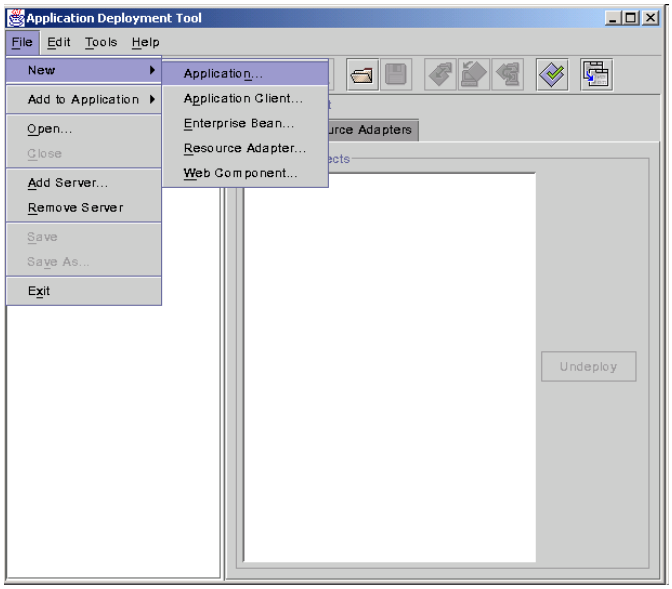


图 8-2 创建 J2EE 应用的菜单项

[1] 新版本的 Java 企业版参考不再提供单独的组装/布署工具，而是将组装、布署的工作集成到了相应的开发工具，如 NetBean 中。

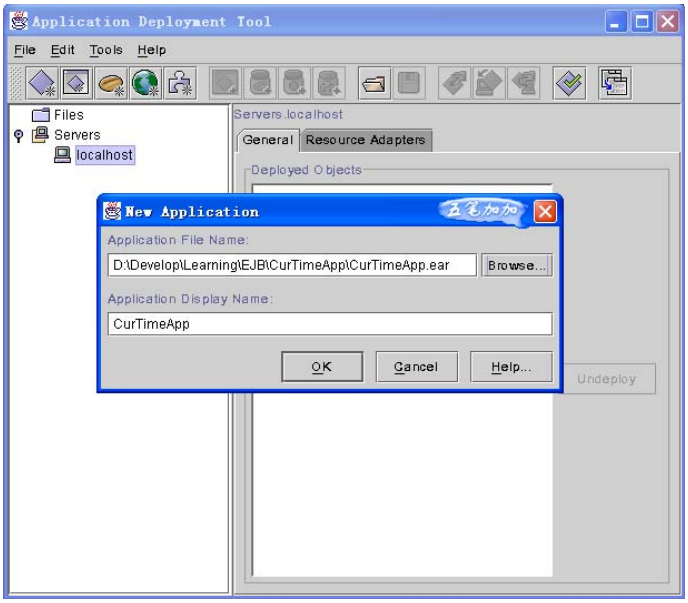


图 8-3 输入企业目标文件名与应用名字

选中创建的 J2EE 应用，如图 8-4 所示，可以看到其中包含 J2EE 规范要求的布署描述符——application.xml，除此之外，该应用中还包含一个 J2EE 参考实现扩充的布署描述符——sun-j2ee-ri.xml。

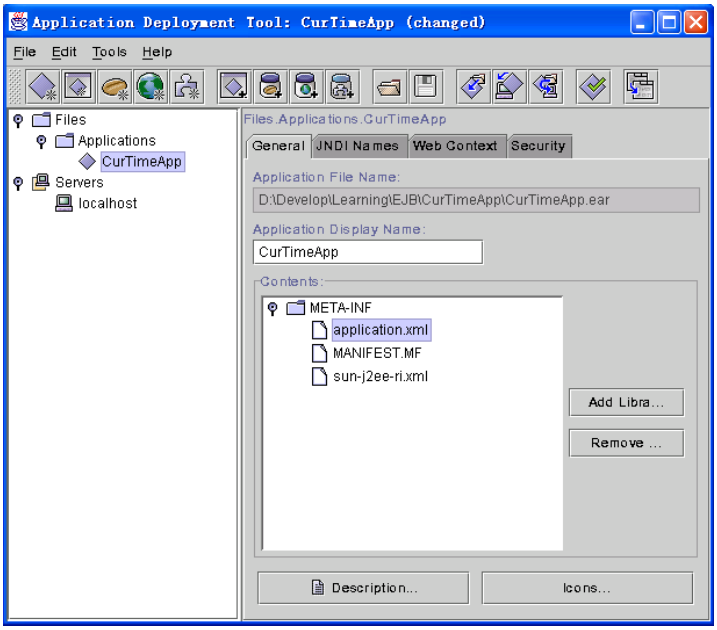


图 8-4 J2EE 应用中包含的布署描述符

创建好 J2EE 应用后，需要将 EJB 构件打包到该应用中的一个 EJB 模块中。按照 6.4 节中的讨论，一个 EJB 模块对应一个 Java 目标文件（JAR 文件），EJB 构件相关的所有类和接口以及部署描述符都被打包到 JAR 文件中，一个 JAR 文件中可以包含多个 EJB，在本例中，只有一个 EJB 构件——时间 EJB。一个 EJB 模块对应的 JAR 文件中包含 EJB 的 Home 接口、Remote 接口、Enterprise Bean 类、实体构件的主键类、布署描述符、其它相关的 Java 类与接口等内容。

在布署工具 deploytool 中，可以在打包某 EJB 模块中的第一个 EJB 构件时同时创建该模块对应的 JAR 文件。如图 8-5 所示，点击 deploytool 的 File\New\Enterprise Bean 菜单可打包一个 EJB 构件，deploytool 提供一个图形界面的向导（wizard）来帮助组装者对待打包的 EJB 构件进行设置，点击图 8-6 所示的向导的 Next 按钮开始配置要打包的时间 EJB。

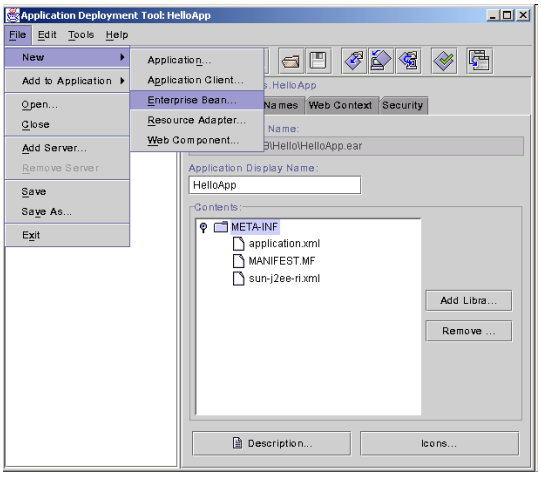


图 8-5 开始打包时间 EJB

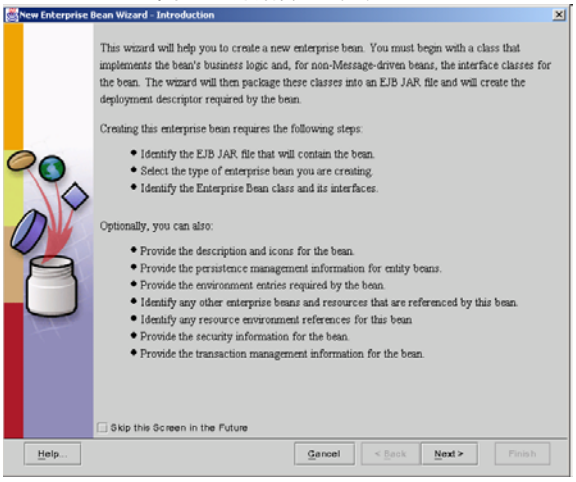


图 8-6 用于打包 EJB 构件的 wizard 的初始界面

首先需要在图 8-7 所示的界面中选择为时间 EJB 创建一个新的 EJB 模块来包含它，本例中我们为该模块使用缺省的名字“Ejb1”。

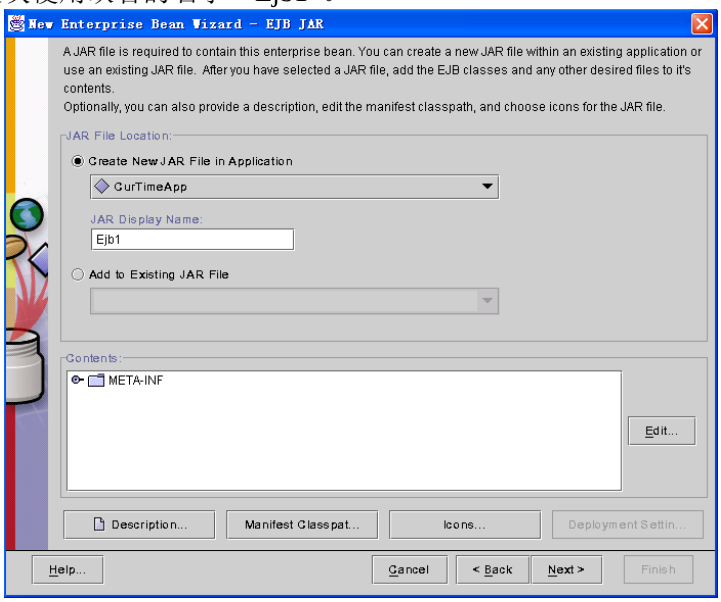


图 8-7 为时间 EJB 创建一个新的 EJB 模块

点击图 8-7 所示界面中的 Edit 按钮，在出现的图 8-8 所示的界面中将时间 EJB 构件的 Home 接口 CurTimeHome、Remote 接口 CurTime 和 Enterprise Bean 类 CurTimeBean 对应的 Java 目标文件加入到当前模块中。添加完成后，可以看到，在当前 EJB 模块中包含时间 EJB 对应的三个 Java 目标文件和一个该模块的布署描述符——ejb-jar.xml。

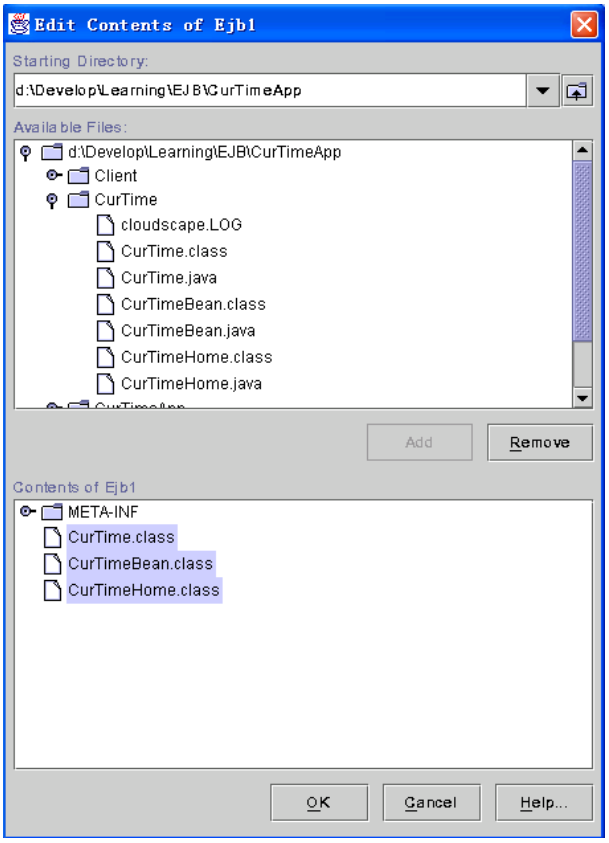


图 8-8 将 EJB 构件相关的 Java 目标文件添加到模块中

然后在图 8-9 所示的界面中设置时间 EJB 构件的类型为无状态会话构件（Stateless Sesssion Bean），并指定其 Enterprise Bean 类、Home 接口与 Remote 接口的名字。

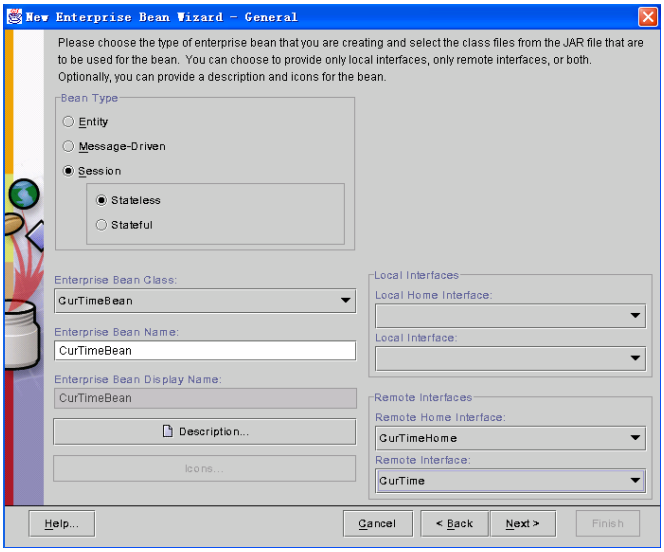


图 8-9 设置 EJB 构件的基本信息

图 8-10 所示的界面用来配置当前正在打包的 EJB 构件的事务控制规则，由于本例中不讨论事务控制，因此可使用缺省设置。

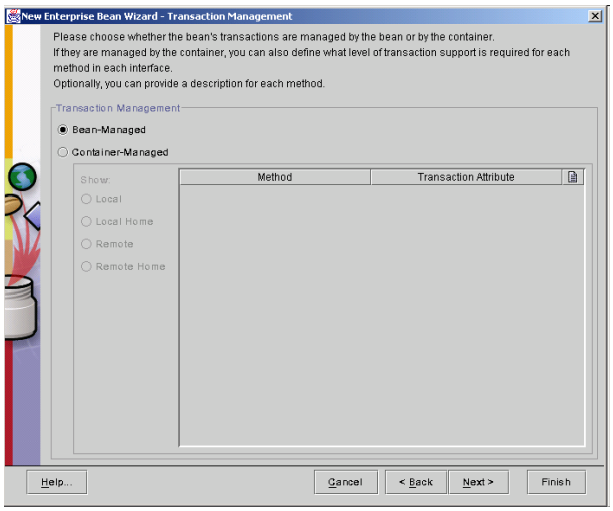


图 8-10 设置事务控制规则

图 8-11 所示的界面用来配置当前正在打包的 EJB 构件所使用的环境条目信息，环境条目由“名字-值”对组成，允许我们在部署或组装 EJB 时进行定制，不需要修改源代码。由于本例中不使用环境条目，因此不需进行额外的设置。

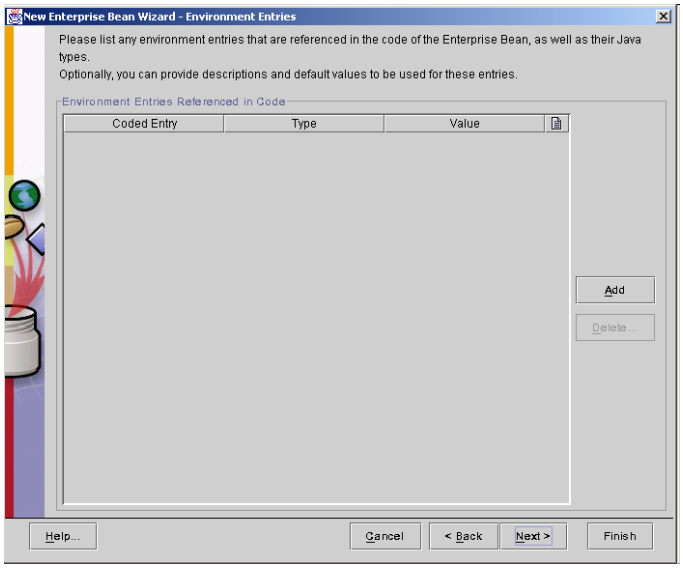


图 8-11 配置环境条目

图 8-12 所示的界面用来配置当前正在打包的 EJB 构件所引用的其它 EJB 的相关信息，对于每个被引用的 EJB 构件，需要指明其 Home 接口和 Remote 接口的名字。由于本例中时间 EJB 不引用其它 EJB 构件，因此不需进行额外的设置。

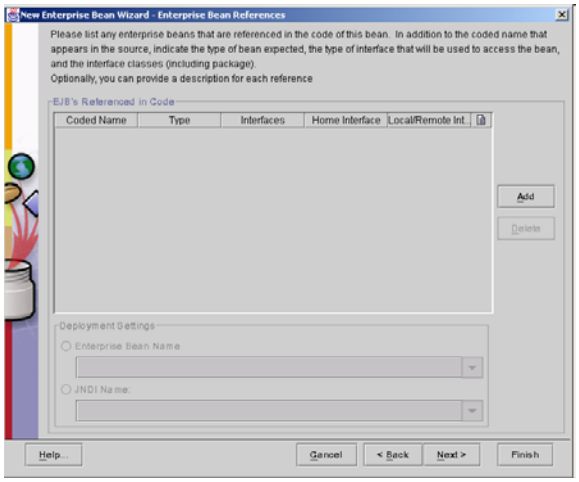


图 8-12 配置 EJB 引用

图 8-13 所示的两个界面分别用来配置当前正在打包的 EJB 构件所使用的资源工厂（如数据源等）与资源环境（如 JMS 的目的地等）。由于本例中时间 EJB 未引用资源工厂与资源环境，因此不需进行额外的设置。

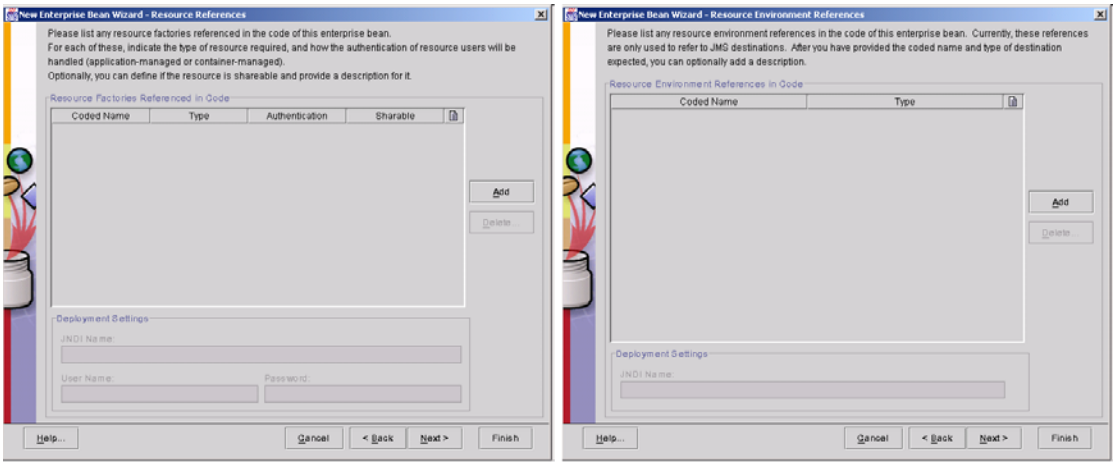


图 8-13 配置资源工厂与资源环境引用

图 8-14 所示的界面用来配置当前正在打包的 EJB 构件的安全性控制规则，由于本例中不讨论安全性控制，因此可使用缺省设置。

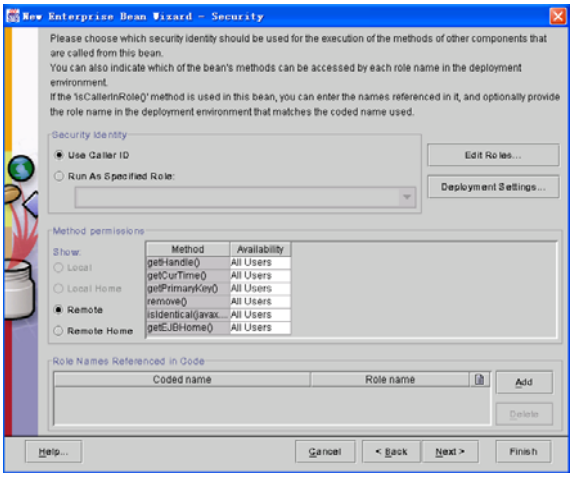


图 8-14 配置安全性控制规则

前面看到的步骤实际是在该向导的帮助下编写 EJB 模块的布署描述符，该向导的最后一个界面如图 8-15 所示，界面上显示了根据前面的配置生成的布署描述符的内容。

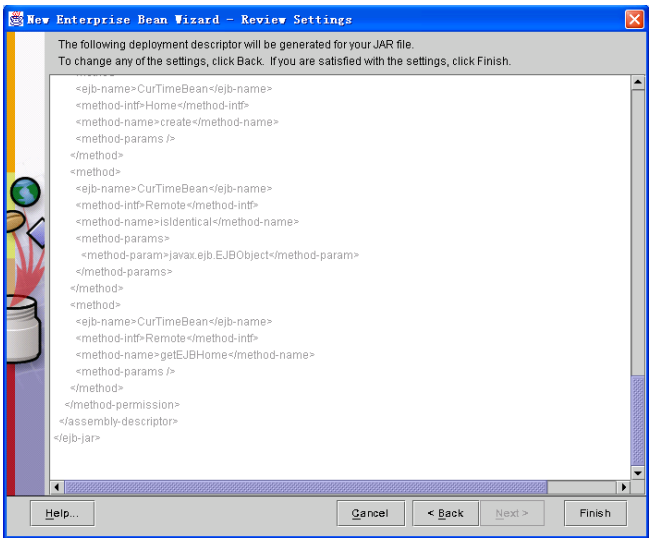


图 8-15 根据前面的配置生成的布署描述符

打包完成后，需要为 EJB 构件配置一个 JNDI 名，配置的 JNDI 名允许客户端查找并使用该 EJB。在图 8-16 所示的 deploytool 界面中选中包含时间 EJB 的模块，点击“JNDI Names”选项卡，可以为 EJB 模块中包含的所有 EJB 构件配置 JDNI 名，本例中我们为时间 EJB 配置名为“CurTimeBean”的 JNDI 名，配置的 JNDI 名一定要和客户端查找 EJB 构件使用的 JNDI 名完全一致。

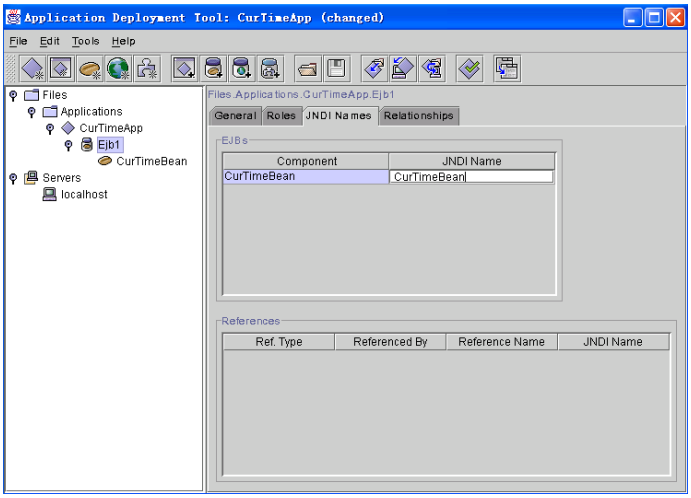


图 8-16 为 EJB 构件配置 JNDI 名

2. 验证并布署 J2EE 应用

配置完 JNDI 名后，就可以将 J2EE 应用布署到 J2EE 服务器上。可以看出，在打包的过程中并没有对 EJB 构件的 Home 接口、Remote 接口和 Enterprise Bean 类之间是否符合 EJB 规范的约定做检查，如果开发人员编写的程序不符合 EJB 规范的约定，则应用无法正常布署到服务器上，因此在布署之前有必要对应用的合法性进行验证。deploytool 提供一个验证工具来检查应用是否符合 J2EE/EJB 规范的约定，该验证工具可以对 J2EE 应用、EJB 模块的 JAR 文件、Web 模块的 WAR 文件以及 J2EE 客户端的 JAR 文件进行验证。如图 8-17 所示，点击 deploytool 的 Tools\Verifier 菜单可打开验证工具，在图 8-18 所示的界面中选择需要验证的模块或应用，点击 OK 按钮开始验证，验证工具会对模块或应用中包含的所有 J2EE

构件进行是否符合 J2EE/EJB 规范的逐项验证，验证结果输出到下面的列表框中；选中列表框中任一项，可以在下面的“Details”文本框中看到该项是否验证通过以及相应的错误信息。

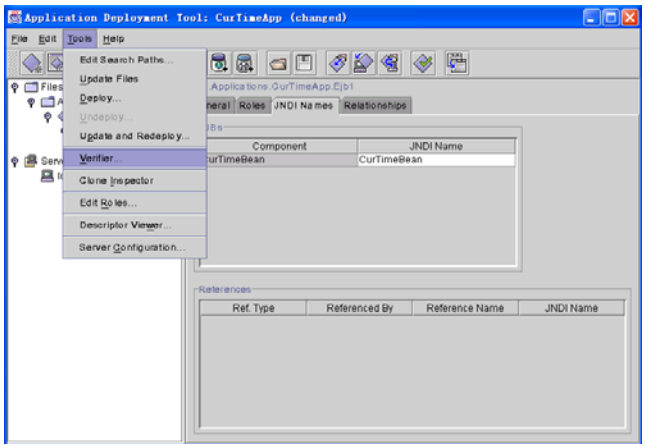


图 8-17 打开验证工具

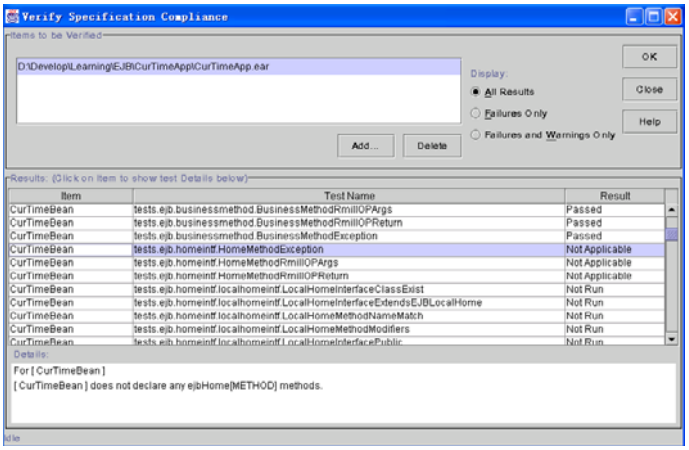


图 8-18 验证应用合法性

验证通过后即可以将应用布署到 J2EE 服务器上，只有部署到 J2EE 服务器上的应用才可以被客户应用访问，部署后不需要在单独启动特定的应用，J2EE 服务器会自动根据客户端的请求情况管理布署的应用中的构件。如图 8-17 所示，点击 deploytool 的 Tools\Deploy 菜单可打开布署应用的向导。

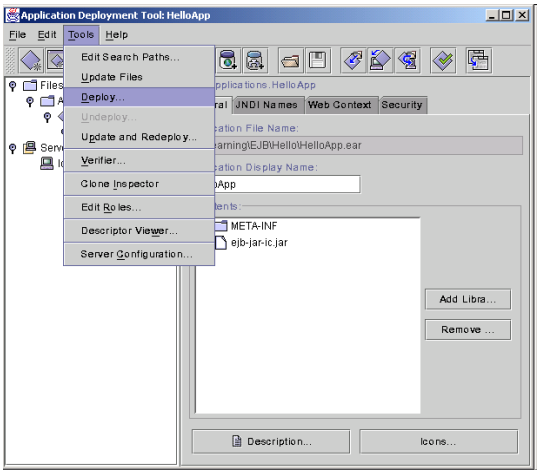


图 8-19 打开布署向导

在图 8-20 所示的布署向导界面中选择要布署的应用，本例中选择 CurTimeApp；选择目标 J2EE 服务器，本例中选择本机 J2EE 服务器 localhost；另外在本例中，由于 EJB 构件的客户端是独立的 Java 程序，因此需要为客户端程序生成包含支持其访问 EJB 构件的接口与类文件的 JAR 程序包，该程序包中包含 EJB 的 Home 接口、Remote 接口定义以及客户端访问 EJB 时需要使用的客户端桩类等 Java 目标文件。在图 8-20 所示的布署向导界面中选择“Return Client Jar”，然后指定对应 JAR 程序包的文件名，本例中我们指定对应 JAR 程序包的文件名为 CurTimeAppClient.jar 并将其存放到客户端程序所在的目录下。

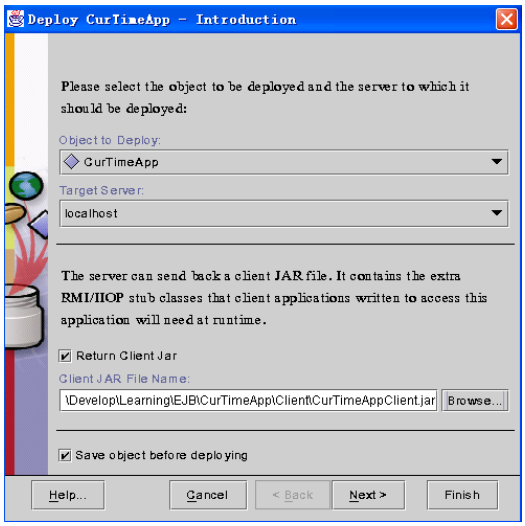


图 8-20 选择服务器并生成客户端 JAR 包

设置完成后，点击 Finish 按钮可完成布署。布署完成后，可以在图 8-21 所示的 deploytool 界面中看到本机 J2EE 服务器 localhost 上已经存在一个布署的应用 CurTimeApp，如果希望卸载布署的应用，可用鼠标右键单击该应用，选择“Undeploy”将其卸载。

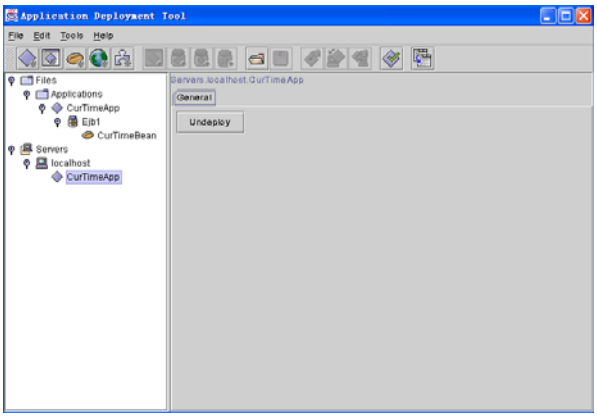


图 8-21 已经布署的应用

8.2.3 开发客户端程序

1. 创建客户端程序

本例中我们编写一个名为 HClient 的 Java 类作为客户端，该类包含一个入口函数 main。该客户端程序调用时间 EJB 的 getCurTime 方法，将返回的时间输出到客户端。客户端程序的代码如程序 8-4 所示。

程序 8-4　时间 EJB 的客户端程序

```
import java.rmi.*;
import javax.naming.*;
import java.util.*;
import javax.rmi.*;
import javax.ejb.*;

public class HClient
{
    public static void main(String[] args)
    {
        try{
            InitialContext Init = new InitialContext();
            CurTimeHome Home = (CurTimeHome)Init.lookup("CurTimeBean");
            CurTime CurTimeObj = Home.create();
            String RetVal = CurTimeObj.getCurTime();
            System.out.println("Returned: " + RetVal);
            CurTimeObj.remove();
        }catch(java.rmi.RemoteException exception){
            System.out.println("Remote exception occurred: " + exception);
        }catch(javax.ejb.CreateException exception){
            System.out.println("Create exception occurred: " + exception);
        }catch(javax.ejb.RemoveException exception){
            System.out.println("Remove exception occurred: " + exception);
        }catch(javax.naming.NamingException exception){
            System.out.println("Naming exception occurred: " + exception);
        }
    }
}
```

在程序 8-4 所示的客户端程序中，首先利用 JNDI 服务查找时间 EJB 对应的 Home 接口，JNDI 服务由类 InitialContext 提供。查找到 Home 接口后，客户端调用 Home 接口中的 create 操作来获取一个可用的 EJB，该操作返回一个 Remote 接口，客户端利用返回的 Remote 接口调用 EJB 上的操作 getCurTime。使用完 EJB 后，客户端程序调用 Remote 接口中的 remove 操作通知服务端其不再使用该 EJB。

读者应注意使用完 EJB 后，客户端并没有通过 Home 接口中的 remove 操作来删除 EJB 对象，因为对于无状态会话构件来说，实例的创建和删除都是由容器自动来控制的，并不是客户端每调用一次 Home 接口中的 create 都会创建一个新的实例，容器也不允许客户端调用 Home 接口中的 remove 方法来删除实例。Remote 接口中也包含一个 remove 方法，对于无状态会话构件来说，调用 Remote 接口中的方法仅仅通知容器不再使用该 EJB 对象了，并不一定会删除 EJB 的对象，另外 Remote 接口中的 remove 方法比 Home 接口中的 remove 方法简单，不需提供任何参数。

2. 编译/运行客户端程序

编写完客户端程序后，即可使用 Java 语言编译器 javac 对客户端程序进行编译，编译时必须将布署时生成的客户端 JAR 程序包包含在 CLASSPATH 中，因为客户端程序中使用了时间 EJB 构件的 Home 接口与 Remote 接口，而布署时已经将所需接口的定义打包在了客户端 JAR 程序包中。本例中可使用以下命令完成对客户端程序的编译：

```
prompt> javac -classpath "CurTimeAppClient.jar;%CLASSPATH%" HClient.java
```

编译完客户端程序后，即可使用 java 命令执行客户端程序，类似地，执行时也必须将布署时生成的客户端 JAR 程序包包含在 CLASSPATH 中，因为客户端程序中使用了时间 EJB 构件的 Home 接口与 Remote 接口，完成调用时需要客户端桩类的支持，而布署时已经将所需接口与客户端桩类的定义打包在了客户端 JAR 程序包中。本例中可使用以下命令执行客户端程序：

```
prompt> java -classpath "CurTimeAppClient.jar;%CLASSPATH%" HClient
```

运行客户端程序时客户端输出 EJB 构件返回的时间信息，例如：

```
Returned: 2008-01-21 17:09:04.842
```

第一次运行客户端程序，由于 EJB 容器中没有任何时间 EJB 的对象，因此容器会自动创建一个该 EJB 对象供客户端使用。创建 EJB 对象后，容器会调用该对象的 ejbCreate 方法完成对象的初始化，我们可以在服务端界面上（J2EE 服务器的运行窗口）看到程序 8-3 中 ejbCreate 方法中输出的信息，如下所示：

```
******CurTimeBean ejbCreate
```

3. 客户端程序中的异常处理

异常是一种特殊的 Java 类，描述了一段代码执行过程中发生的运行时错误。本例客户端程序执行的过程中可能会发生的异常及可能的原因如下：

- **ClassCastException**：该异常包含在 java.lang 包中。如果客户程序执行时发生了 ClassCastException 异常，一般表明客户程序不能访问部署时生成的客户端 JAR 文件，此时应检查在部署时是否正确生成了客户端 JAR 文件。
- **NameNotFoundException**：该异常包含在 javax.naming 包中。如果 J2EE 服务器不能找到 JNDI 名字相关的 EJB 构件时，会抛出该异常；引发该异常的可能的原因包括 EJB 没有部署到服务器上或 EJB 构件的 JNDI 名错误等。
- **NamingException**：该异常包含在 javax.naming 包中。如果客户程序不能访问 J2EE 服务器提供的服务（如命名服务），会抛出该异常。引发该异常的可能的原因包括 J2EE 服务器没有启动等。
- **NoClassDefFoundError**：该异常包含在 java.lang 包中。如果客户程序不能访问定义在客户端 class 文件或 j2ee.jar 文件中的类时会抛出该异常。引发该异常的可能的原因包括客户端某些 java 文件未编译成功、J2EE_HOME 环境变量设置不正确以及 j2ee.jar 在未包含在虚拟机的 classpath 中等。

8.2.4 关于例子程序的进一步讨论

由于无状态会话构件不保存与特定客户端相关的会话状态，其实例可以被多个客户端共享，因此容器会尽可能使用少量的实例为多个客户端提供服务。

细心的读者可能已经发现，尽管客户端程序执行结束时调用了 Remote 接口中的 remove 操作，但是容器并不会删除客户端使用的 EJB 实例（即 EJB 对象的 ejbRemove 方法并未被调用），因为容器要将该对象保留下来供后续使用。如果再次运行客户端程序，尽管再次调用了 Home 接口中的 create 方法，但是容器并不会再次创建新的 EJB 对象（即 ejbCreate 不会再次被调用），而是让客户端使用上次运行创建的对象。连续再次运行客户端程序对应的客户端输出与服务端输出信息如下：

```
客户端输出:
prompt> java -classpath "CurTimeAppClient.jar;%CLASSPATH%" HClient
Returned: 2008-01-21 17:09:04.842
prompt> java -classpath "CurTimeAppClient.jar;%CLASSPATH%" HClient
Returned: 2008-01-21 17:09:30.521
```

```
服务端输出:
******CurTimeBean ejbCreate
```

容器会自动管理 EJB 构件的生命周期以满足多个客户端访问的需要，对于无状态会话构件来说，当存在多个客户端同时访问某构件时，容器会创建该构件的多个对象以并发地为多个客户端提供服务，有兴趣的读者可以修改程序 8-4 给出的客户端程序，模拟出多个客户

端并发使用时间 EJB 构件的情况，此时即可看到容器创建多个时间 EJB 对象的情形（即 ejbCreate 方法被多次调用）。如果特定时刻大规模的并发访问导致容器创建了大量的某无状态会话构件的大量对象，则当并发的访问量降低时容器会自动删除一部分对象以降低对系统资源（如系统内存）的消耗，此时即可看到容器删除时间 EJB 对象的情形（即 ejbRemove 方法被调用）。

§8.3 开发与使用有状态会话构件

本节通过一个简单的例子演示如何开发与使用有状态会话构件，例子程序分为客户端与服务端两部分，在服务端，我们将构建一个名为 ShoppingApp 的 J2EE 应用，该应用中仅包含一个有状态会话构件，该构件实现网上购物系统中购物车的基本功能；客户端通过远程接口调用构件上的操作，将结果输出。

8.3.1 开发 EJB 构件

该 EJB 构件实现网上购物系统中购物车的基本功能，包括添加商品、去除商品、查找商品、清空购物车、提交商品等。由于该构件的实例（对象）需要保存与特定客户端相关的会话状态，即特定客户所选择的商品等相关信息，因此设计为有状态的会话构件。以下简称该 EJB 构件为购物车 EJB。

1. 定义 Remote 接口

Remote 接口包含 EJB 构件实现的商业方法的声明，客户端只能通过 remote 接口访问构件实现的商业方法，不能直接调用。程序 8-5 给出了购物车 EJB 的 Remote 接口定义：

程序 8-5 购物车 EJB 的 Remote 接口定义

```
package Shopping;

import javax.ejb.*;
import java.rmi.*;
public interface ShoppingBag extends EJBObject
{
    public void addCom(Commodity comm) throws RemoteException;
    public void removeComm(Commodity comm)
                    throws RemoteException, NoSuchCommodityException;
    public Commodity find(String commID)
                    throws RemoteException, NoSuchCommodityException;
    public void clearBag() throws RemoteException;
    public void commit() throws RemoteException, BagEmptyException;
}
```

在程序 8-5 中，我们定义了一个名为 ShoppingBag 的 Remote 接口，购物车 EJB 构件向客户端提供添加商品、去除商品、查找商品、清空购物车、提交商品等商业方法，Remote 接口中包含这些方法的声明：

- **addCom**：添加商品。调用者提供一个表示商品信息的参数，没有返回值。
- **removeComm**：去除商品。调用者提供一个表示商品信息的参数，没有返回值，如果对应商品信息在购物车中不存在，则抛出 NoSuchCommodityException。
- **find**：查找商品。调用者提供一个表示商品标识的参数，返回购物车中对应的商品信息，如果对应商品信息在购物车中不存在，则抛出 NoSuchCommodityException。
- **clearBag**：清空购物车。将购物车中所有商品清空，没有参数与返回值。
- **commit**：提交商品。将购物车中当前的商品提交，没有参数与返回值，如果购物车中没有任何商品，则抛出 BagEmptyException 异常。

可以看到，按照 EJB 规范的约定，接口 ShoppingBag 继承了接口 EJBObject，接口中的每个操作均抛出 RemoteException 异常以报告远程调用错误，所有操作的参数与返回值均为合法的 Java RMI 类型（Commodity 类具体定义详见程序 8-8）。

2. 定义 Home 接口

Home 接口中包含 EJB 构件生命周期管理的相关方法，客户程序使用 Home Interface 创建、查找或删除 EJB 的实例。程序 8-6 给出了购物车 EJB 的 Home 接口定义：

程序 8-6　购物车 EJB 的 Home 接口定义

```
package Shopping;

import javax.ejb.*;
import java.rmi.*;
public interface ShoppingBagHome extends EJBHome
{
    public ShoppingBag create(String customerName)
                    throws RemoteException, CreateException;
}
```

在程序 8-6 中，我们定义了一个名为 ShoppingBagHome 的 Home 接口，按照 EJB 规范的约定，接口 ShoppingBagHome 继承了接口 EJBHome。该接口中仅声明了一个 create 方法，其返回值为对应 EJB 构件的 Remote 接口类型——ShoppingBag，同时 create 方法抛出 RemoteException 和 CreateException 异常。

和无状态的会话 bean 不同，有状态会话构件所使用的 Home 接口中可以包含若干个带不同参数列表的 create 方法。客户端使用每一个有状态会话构件之前，可以利用一个 create 方法来对其初始化。本例中的 create 方法支持客户端使用购物车之前指定使用者的名字。

3. 定义 Enterprise Bean 类

购物车 EJB 的 Enterprise Bean 类首先要按照 Remote 接口的约定实现商业方法，其次要实现 Home 接口中 create 方法对应的 ejbCreate 方法与会话构件生命周期相关的方法。程序 8-7 给出了购物车 EJB 的 Enterprise Bean 类定义：

程序 8-7　购物车 EJB 的 Enterprise Bean 类定义

```
package Shopping;

import javax.ejb.*;
import java.util.*;
public class ShoppingBagBean implements SessionBean
{
    SessionContext Context;
    String curCustomer;
    Hashtable curCommodities;
    public void addCom(Commodity comm){
        Commodity nowComm = (Commodity)curCommodities.get(comm.sCommodityID);
        if(nowComm == null){
            curCommodities.put(comm.sCommodityID, comm);
        }else{
            nowComm.iCount += comm.iCount;
        }
    }
    public void removeComm(Commodity comm) throws NoSuchCommodityException{
        Commodity nowComm = (Commodity)curCommodities.get(comm.sCommodityID);
        if(nowComm == null)
            throw new NoSuchCommodityException("No such commodity with id: " +
                                               comm.sCommodityID);
        nowComm.iCount -= comm.iCount;
        if(nowComm.iCount <= 0)
            curCommodities.remove(comm.sCommodityID);
    }
    public Commodity find(String commID) throws NoSuchCommodityException{
        Commodity nowComm = (Commodity)curCommodities.get(commID);
        if(nowComm == null)
            throw new NoSuchCommodityException("No such commodity with id: " +
                                               commID);
```

```
        return nowComm;
    }
    public void clearBag(){
        curCommodities.clear();
    }
    public void commit() throws BagEmptyException{
        if(curCommodities.isEmpty())
            throw new BagEmptyException("Bag is empty");
        System.out.println("Customer name is: " + curCustomer);
        System.out.println("\tID\t\tName\t\tSpec\t\tPrice\t\tCount");
        Enumeration enums = curCommodities.elements();
        while(enums.hasMoreElements()){
            Commodity comm = (Commodity)enums.nextElement();
            System.out.println("\t" + comm.sCommodityID + "\t\t" + comm.sName +
                               "\t\t"+comm.sSpec+"\t\t"+comm.fPrice+"\t\t"
                               + comm.iCount);
        }
        curCommodities.clear();
    }

    public void ejbCreate(String customerName){
        curCustomer = customerName;
        curCommodities = new Hashtable();
        System.out.println("\n\n******ShoppingBagBean ejbCreate with name:"
                           + customerName);
    }
    public void ejbRemove(){
        curCommodities.clear();
        System.out.println("ShoppingBagBean ejbRemove");
    }
    public void ejbPassivate(){
        System.out.println("ShoppingBagBean ejbPassivate");
    }
    public void ejbActivate(){
        System.out.println("ShoppingBagBean ejbActivate");
    }
    public void setSessionContext(SessionContext Context)
    {
        this.Context = Context;
    }
}
```

在程序 8-7 中，我们定义了一个名为 ShoppingBagBean 的 Enterprise Bean 类，由于该 EJB 构件是会话构件，因此 Enterprise Bean 类实现（implements）SessionBean 接口。

由于有状态会话构件要保存与特定客户端交互的中间状态，而对于有状态会话构件来说，容器保证每个 EJB 对象/实例被某一个客户端所专用，因此保存状态最方便的地方就是在对象的数据成员/属性中，因为每个对象会使用一份独立的数据成员。因此购物车 EJB 的 Enterprise Bean 类中设置了数据成员 curCustomer 与 curCommodities，其中数据成员 curCustomer 用于记录当前购物车使用者的名字，数据成员 curCommodities 用于记录使用者当前选择的商品的相关信息。

该 Enterprise Bean 类首先按照 Remote 接口 ShoppingBag 的约定实现了商业方法：

- **addCom**：添加商品。在数据成员 curCommodities 查找是否存在该种商品，如果存在则修改相应的商品数量，否则添加该种商品信息。
- **removeComm**：去除商品。在数据成员 curCommodities 查找该种商品信息，修改相应的商品数量，如果数量为 0，则去掉该种商品信息；如果对应商品信息在购物车中不存在，则抛出 NoSuchCommodityException。
- **find**：查找商品。在数据成员 curCommodities 查找该种商品信息，将商品信息返回；如果对应商品信息在购物车中不存在，则抛出 NoSuchCommodityException。
- **clearBag**：清空购物车。将数据成员 curCommodities 中所有商品清空。

- **commit**：提交商品。为简单起见，本例中仅将使用者选择的商品输出到屏幕，输出的信息包括数据成员 curCustomer 保存的使用者的名字以及数据成员 curCommodities 中保存的商品的详细信息。

其次，Enterprise Bean 类实现了 Home 接口 ShoppingBagHome 中的 create 方法对应的 ejbCreate 方法，容器调用 ejbCreate 方法完成 EJB 对象的初始化，本例中 ejbCreate 方法会在使用者使用一个购物车之前为其准备好一个全新的购物车——记录使用者的名字并初始化商品信息列表 curCommodities。此外还实现了会话构件生命周期相关的 ejbRemove、ejbPassivate、ejbActivate 与 ejbSetSessionContext 方法，其中 ejbPassivate 与 ejbActivate 方法有状态会话构件需要使用，ejbPassivate 方法会在有状态会话构件对象被转移到持久存储介质之前被调用，ejbActivate 方法会在有状态会话构件对象从持久存储介质被加载入内存中之后被调用。

如图 8-22 所示，有状态会话构件的生命周期包含三个状态：方法就绪状态（Method Ready State）、不存在状态（No State）与钝化状态（Passivated State）。不存在状态表明 EJB 容器中不存在对应有状态会话构件的实例，处于不存在状态的实例还未被创建；方法就绪状态表明对应有状态会话构件对象已被创建并处于就绪状态，可以为客户端提供服务；钝化状态表明对应有状态会话构件对象已被转移至持久存储介质，暂时不能使用。

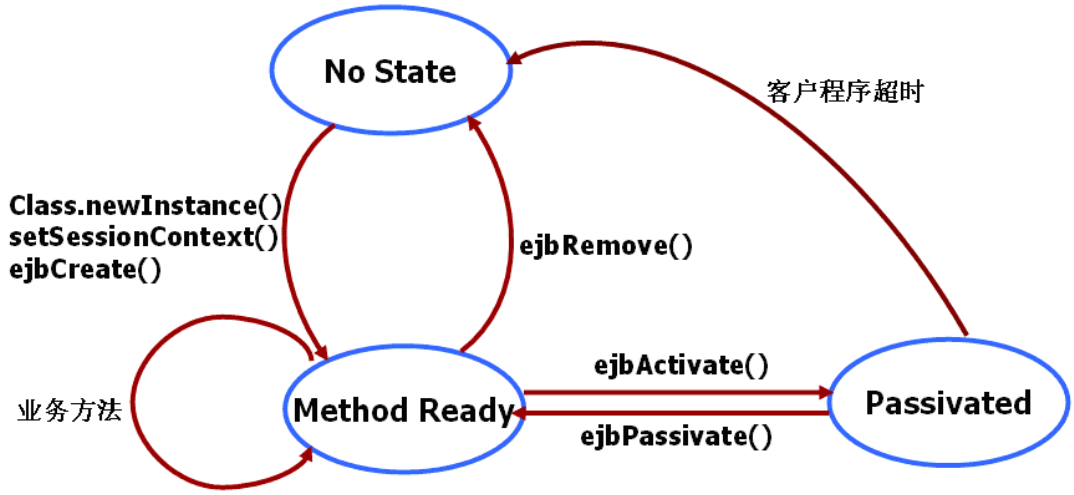


图 8-22 有状态会话构件生命周期

与无状态会话构件相比，有状态会话构件的生命周期增加了一个钝化状态，这是因为有状态会话构件需要保存与特定客户端相关的中间状态，因此每个实例/对象都是被一个客户端所专用的，这就使得每个客户端都需要一个专门的有状态会话 bean 来为它服务，则很有可能在服务端同时存在大量的 EJB 实例，从而导致服务端内存开销太大。为了限制服务端内存使用总量，当 EJB 实例的数量过多时，容器仅仅会在内存中保留那些正在使用或者刚被使用的实例，会把其它的实例转移到持久存储介质上（**不是删除**），此时被转移到持久存储介质上的实例会从方法就绪状态进入钝化状态。当客户端再次使用处于钝化状态的 EJB 实例时，容器会把该实例从持久存储介质中恢复到内存中，此时该实例从钝化状态进入方法就绪状态。当客户端出现超时时，容器会把持久存储介质中的实例删除掉，该实例进入不存在状态。

对于有状态会话构件来说，只要有新的客户端请求（调用 Home 接口中的 create 方法），容器就会创建新的实例，此时新创建的实例从不存在状态进入方法就绪状态。如果客户端不再使用某个有状态会话构件对象了（调用 Remote 接口中的 remove 方法），或者客户程序出现超时现象，容器就会删除该实例，此时被删除的实例又从方法就绪状态状态进入不存在状态。

从不存在状态进入方法就绪状态（创建新实例），会调用 Enterprise Bean 类对象的

setSessionContext 和 ejbCreate 方法。（每一次客户端调用 Home 接口中的 create 都会导致容器调用 Enterprise Bean 类对象的 ejbCreate 方法）；从方法就绪状态进入不存在状态（删除实例），会调用 ejbRemove 方法。（每次客户端调用 Remote 接口中的 remove 方法都会导致容器调用 Enterprise Bean 类对象的 ejbRemove 方法）；在方法就绪状态和钝化状态之间切换时，容器会调用 ejbPassivate 和 ejbActivate 方法。

4. 其它辅助代码

除了购物车 EJB 相关的 Remote 接口、Home 接口与 Enterprise Bean 类之外，本例还需要三个辅助的 Java 类商品类 Commodity 和两个辅助的异常类 NoSuchCommodityException 与 BagEmptyException。这三个类的代码如程序 8-8、8-9 和 8-10 所示。

程序 8-8　商品类 Commodity 的定义

```
package Shopping;

public class Commodity implements java.io.Serializable{
    public String sCommodityID;
    public String sName;
    public String sSpec;
    public float fPrice;
    public int iCount;
}
```

程序 8-9　异常类 NoSuchCommodityException 的定义

```
package Shopping;
public class NoSuchCommodityException extends Exception{
    public NoSuchCommodityException(String msg){
        super(msg);
    }
}
```

程序 8-10　异常类 BagEmptyException 的定义

```
package Shopping;
public class BagEmptyException extends Exception{
    public BagEmptyException(String msg){
        super(msg);
    }
}
```

5. 编译源代码

本例服务端编译时需要将 EJB 构件相关的 Remote 接口、Home 接口、Enterprise Bean 类和辅助类 Commodity、辅助异常 NoSuchCommodityException 与 BagEmptyException 均进行编译。

6. 打包/布署 EJB

本例中打包与布署 EJB 构件的过程与上节类似，不同的地方主要有几下几点：

- 本例将包含 EJB 构件的 J2EE 应用命名为 ShoppingApp。
- 本例中打包 EJB 构件时需要将 EJB 构件相关的 Remote 接口、Home 接口、Enterprise Bean 类和辅助类 Commodity、辅助异常 NoSuchCommodityException 与 BagEmptyException 均添加到对应的 EJB 模块中，如图 8-23 所示：

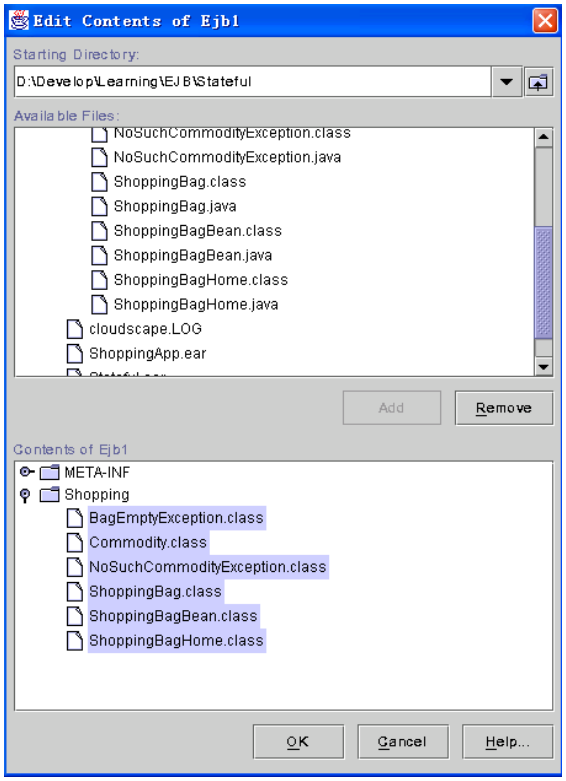


图 8-23 将 EJB 构件相关的 Java 目标文件添加到模块中

- 本例中将 EJB 构件类型设为有状态会话构件，如图 8-24 所示：

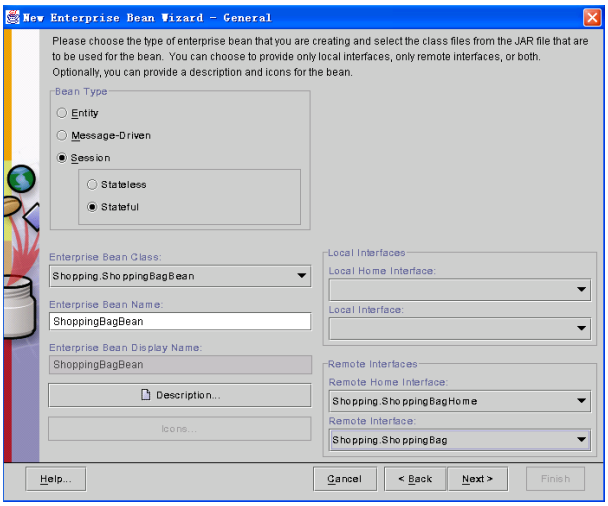


图 8-24 将 EJB 构件类型设置为有状态会话构件

- 为购物车 EJB 配置名为“ShoppingBag”的 JNDI 名。
- 布署时生成名为“StatefulAppClient.jar”的客户端 JAR 文件包，并将其存放到后面客户端程序所在的目录下。

8.3.2 开发客户端程序

1. 创建客户端程序

本例中我们编写一个名为 StatefulClient 的 Java 类作为客户端，该类包含一个入口函数 main。该客户端程序调用购物车 EJB 的商业方法，完成商品的添加、去除与查找，最后将

选择的商品提交。客户端程序的代码如程序 8-11 所示。

程序 8-11　购物车 EJB 的客户端程序

```
import java.rmi.*;
import javax.naming.*;
import java.util.*;
import javax.rmi.*;
import javax.ejb.*;
import Shopping.*;

public class StatefulClient
{
    public static void main(String[] args)
    {
        try{
            InitialContext Init = new InitialContext();
            ShoppingBagHome Home = (ShoppingBagHome)Init.lookup("ShoppingBag");
            ShoppingBag ShoppingBagObj = Home.create("Mr. Zhang");
            Commodity comm = new Commodity();
            comm.sCommodityID = "10000001"; comm.sName = "Phone";
            comm.sSpec = "T29"; comm.fPrice = 1000; comm.iCount = 1;
            ShoppingBagObj.addCom(comm);
            comm.sCommodityID = "10000002"; comm.sName = "Camera";
            comm.sSpec = "D300"; comm.fPrice = 1500; comm.iCount = 2;
            ShoppingBagObj.addCom(comm);
            try{
                comm = ShoppingBagObj.find("10000001");
                System.out.println(
                    "we have commodities with id 10000001, name is " + comm.sName);
            }catch(NoSuchCommodityException e){
                System.out.println(e.getMessage());
            }
            try{
                ShoppingBagObj.removeComm(comm);
            }catch(NoSuchCommodityException e){
                System.out.println(e.getMessage());
            }
            try{
                comm = ShoppingBagObj.find("10000001");
                System.out.println(
                    "we have commodities with id 10000001, name is " + comm.sName);
            }catch(NoSuchCommodityException e){
                System.out.println(e.getMessage());
            }
            try{
                ShoppingBagObj.commit();
            }catch(BagEmptyException e){
                System.out.println(e.getMessage());
            }
            ShoppingBagObj.remove();
        }catch(java.rmi.RemoteException exception){
            System.out.println("Remote exception occurred: " + exception);
        }catch(javax.ejb.CreateException exception){
            System.out.println("Create exception occurred: " + exception);
        }catch(javax.ejb.RemoveException exception){
            System.out.println("Remove exception occurred: " + exception);
        }catch(javax.naming.NamingException exception){
            System.out.println("Naming exception occurred: " + exception);
        }
    }
}
```

在程序 8-11 所示的客户端程序中，首先利用 JNDI 服务查找购物车 EJB 对应的 Home

接口。查找到 Home 接口后，客户端调用 Home 接口中的 create 操作来获取一个可用的 EJB，该操作返回一个 Remote 接口，客户端利用返回的 Remote 接口调用 EJB 上的商业方法 addCom、RemoveComm、find、commit 等。使用完 EJB 后，客户端程序调用 Remote 接口中的 remove 操作通知服务端其不再使用该 EJB。

读者应注意注意本例中客户端使用的是一个有状态会话构件，每个有状态会话构件的实例/对象被一个特定客户端所专用，并为其保存交互的中间状态。在后面的讨论中我们将关注程序 8-11 中再次同样参数调用 EJB 的 find 操作的调用结果对比。

2. 编译/运行客户端程序

编写完客户端程序后，即可使用 Java 语言编译器 javac 对客户端程序进行编译，编译时必须将布署时生成的客户端 JAR 程序包包含在 CLASSPATH 中，因为客户端程序中使用了购物车 EJB 构件的 Home 接口与 Remote 接口，而布署时已经将所需接口的定义打包在了客户端 JAR 程序包中。本例中可使用以下命令完成对客户端程序的编译：

```
prompt> javac -classpath " StatefulAppClient.jar;%CLASSPATH%" StatefulClient.java
```

编译完客户端程序后，即可使用 java 命令执行客户端程序，类似地，执行时也必须将布署时生成的客户端 JAR 程序包包含在 CLASSPATH 中，因为客户端程序中使用了购物车 EJB 构件的 Home 接口与 Remote 接口，完成调用时需要客户端桩类的支持，而布署时已经将所需接口与客户端桩类的定义打包在了客户端 JAR 程序包中。本例中可使用以下命令执行客户端程序：

```
prompt> java -classpath " StatefulAppClient.jar;%CLASSPATH%" StatefulClient
```

运行客户端程序时客户端输出两次调用 EJB 的 find 操作的结果，例如：

```
we have commodities with id 10000001, name is Phone
No such commodity with id: 10000001
```

由于本例中客户端使用的是一个有状态会话构件，因此在程序 8-11 所示的客户端程序中，尽管包含再次同样参数的对方法 find 的调用，其调用的结果却因对应购物车对象的状态不同而完全不同。

第一次运行客户端程序，容器会自动创建一个该 EJB 对象供客户端使用。创建 EJB 对象后，容器会调用该对象的 ejbCreate 方法完成对象的初始化，我们可以在服务端界面上（J2EE 服务器的运行窗口）看到程序 8-7 中 ejbCreate 方法中输出的信息；当客户端调用 EJB 对象的 commit 操作时，我们可以在服务端界面上看到程序 8-7 中 commit 方法中输出的信息；当客户端调用 Remote 接口中的 remove 操作时，容器会将 EJB 构件的对应实例删除，此时会调用该对象的 ejbRemove 方法，我们可以在服务端界面上看到程序 8-7 中 ejbRemove 方法中输出的信息；第一次运行客户端程序时服务端的输出信息如下所示：

```
******ShoppingBagBean ejbCreate with name:Mr. Zhang
Customer name is: Mr. Zhang
       ID            Name          Spec          Price          Count
       10000002      Camera        D300          1500.0          2
ShoppingBagBean ejbRemove
```

8.3.3 关于例子程序的进一步讨论

由于有状态会话构件需要保存与特定客户端相关的会话状态，其实例不可以被多个客户端共享，因此容器会在客户端不再使用某 EJB 实例时将其删除，而每次有新的客户端请求时，容器会为其创建一个新的实例。

细心的读者可能已经发现，与上一节无状态会话构件例子相比，客户端程序执行结束时

调用 Remote 接口中的 remove 操作会导致容器删除 EJB 实例（对应 EJB 对象的 ejbRemove 方法会在实例被删除之前被调用），因为容器该对象是被特定客户端所专用的，当该客户端不再使用该 EJB 实例时，该实例已经没有继续存在的必要。如果再次运行客户端程序，由于再次调用了 Home 接口中的 create 方法，容器会认为这是新在客户端请求，会再次创建新的 EJB 对象（即 ejbCreate 再次被调用）。连续再次运行客户端程序对应的客户端输出与服务端输出信息如下：

```
客户端输出:
prompt> java -classpath " StatefulAppClient.jar;%CLASSPATH%" StatefulClient
we have commodities with id 10000001, name is Phone
No such commodity with id: 10000001
prompt> java -classpath " StatefulAppClient.jar;%CLASSPATH%" StatefulClient
we have commodities with id 10000001, name is Phone
No such commodity with id: 10000001
```

```
服务端输出:
******ShoppingBagBean ejbCreate with name:Mr. Zhang
Customer name is: Mr. Zhang
       ID            Name          Spec          Price          Count
       10000002      Camera        D300          1500.0          2
ShoppingBagBean ejbRemove



******ShoppingBagBean ejbCreate with name:Mr. Zhang
Customer name is: Mr. Zhang
       ID            Name          Spec          Price          Count
       10000002      Camera        D300          1500.0          2
ShoppingBagBean ejbRemove
```

容器会自动管理 EJB 构件的生命周期以满足多个客户端访问的需要，对于有状态会话构件来说，容器会为每个客户端程序创建一个该构件的对象供其专用。因此当同时使用该构件的客户端数量很大时（尽管这些客户端并一定会并发调用该 EJB 实例上操作），只要客户端还需要使用（且未超时），容器必须为其保留对象实例，这就会导致服务端内存中存在大量 EJB 对象，从而导致服务端内存开销太大。为了限制服务端内存使用总量，当 EJB 实例的数量过多时，容器仅仅会在内存中保留那些正在使用或者刚被使用的实例，会把其它的实例转移到持久存储介质上（**不是删除**），当处于钝化状态的对象再次被使用时，容器会将其从持久存储介质上重新加载到内存中。

有兴趣的读者可以修改程序 8-11 给出的客户端程序，如多次调用 Home 接口中 create 方法而不调用 Remote 接口中 remove 方法，模拟出大量客户端同时使用购物车 EJB 构件的情况，此时即可看到容器创建大量购物车 EJB 对象的情形。

对于 J2EE 参考实现来说，当容器中对象所占用的内存容量超过预设的值时，容器会自动执行钝化操作，该值在 J2EE 参考实现安装目录下的 config 子目录中的 default.properties 配置文件中设置，当前版本的缺省值 128000000 字节，为了进行钝化相关的测试，读者可应

该值改小，default.properties 配置文件的可能内容如程序 8-12 所示，其中粗体部分为钝化使用的内存容量临界值（passivation.threshold.memory）。

程序 8-12　包含钝化使用的内存容量临界值的配置文件

```
# maximum size of message driven bean instance
# pool per mdb type
messagebean.pool.size=3

# maximum size of a "bulk" message bean delivery
messagebean.max.serversessionmsgs=1

# message-bean container resource cleanup interval
messagebean.cleanup.interval=600

passivation.threshold.memory=128000000
idle.resource.threshold=600
log.directory=logs
log.output.file=output.log
log.error.file=error.log
log.event.file=event.log

distributed.transaction.recovery=false
transaction.timeout=0
transaction.nonXA.optimization=true
sessionbean.timeout=0

# validating parser values
# validating.perser is used when archive file are loaded by
# any of the J2EE Reference Implementation tools.
# deployment.validating.parser is used when deploying an
# archive on the J2EE AppServer.
validating.parser=false
deployment.validating.parser=true
```

执行钝化操作时容器会在钝化某个对象前调用其 ejbPassivate 方法，此时我们可以在服务端界面上看到程序 8-7 中 ejbPassivate 方法中输出的信息。J2EE 参考实现的钝化操作会将有状态会话构件对象以序列化/串行化（serialize）的方式写入数据库中，J2EE 参考实现钝化时缺省使用的数据库管理系统为其安装时自带的数据库管理系统 cloudscape，因此要进行钝化测试读者需要将 cloudscape 数据库管理系统启动，启动 cloudscape 数据库管理系统的命令如下：

prompt> cloudscape -start

J2EE 参考实现会将对象写入 clouscape 的缺省数据库 CloudscapeDB 的 SESSIONBEANTABLE 表中。有兴趣的读者可以使用通过 cloudscape 的交互式 sql 查看该表中的数据。进入 cloudscape 交互式 sql 的命令如下：

prompt> cloudscape -isql

在 cloudscape 交互式 sql 中执行以下命令可以查看 SESSIONBEANTABLE 表中数据：

ij> select * from SESSIONBEANTABLE;

关闭 clouscape 数据库管理系统的命令如下：

prompt> cloudscape -stop

如果客户端调用了某个已经被钝化的有状态会话构件实例上的操作，容器会将该对象重新从数据库中加载到内存中，加载完成后容器会调用该对象的 ejbActivate 方法，此时我们可以看到程序 8-7 中 ejbActivate 方法中输出的信息。通过 ejbPassivate 与 ejbActivate 方法的调用时机可以看出，开发人员可以在 ejbPassivate 方法暂时释放占用的一些系统资源，因为对象进入钝化状态后暂时不能被使用，而此时释放的系统资源可以在 ejbActivate 方法中重

新取回，因为此时对象又可以使用了。

§8.4 开发与使用实体构件

8.4.1 实体构件与持久化技术

在 EJB 体系结构中，实体构件主要用来封装数据库操作。从软件体系结构的角度讲，如图 8-25 所示，实体构件组成了数据层与业务逻辑层之间的一层——持久层，持久层的实体构件向业务逻辑构件（如会话构件）屏蔽数据库开发的复杂性，为业务逻辑构件方便访问数据库提供面向对象的封装。基于实体构件的支持，业务逻辑构件以对象的方式看待与处理数据库中的数据，从而大致简化数据库开发的目的。

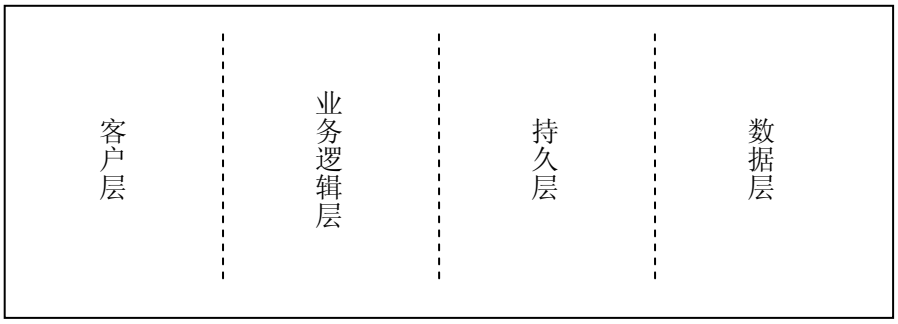


图 8-25 实体构件组成的持久层

在 Java 技术的发展过程中，基于 Java 语言数据持久化技术也等到了长足发展，目前常用的 Java 持久化实现方案主要包括：

- 基于 DAO 和 JDBC 实现：这种方案通过 DAO（Data Access Object，数据访问对象）来实现数据的持久化操作，具体实现时，DAO 通过 JDBC 来完成对数据库的访问。这种方案要求开发人员对 JDBC 的底层信息要比较熟悉。
- 基于 ORM 实现：ORM 的全称为 Object Relational Mapping，其基本思想将关系型数据库中的数据利用某种机制映射为 Java 对象，在业务逻辑构件看来，数据库中的数据以 Java 对象的形式出现，通常每个对象对应数据库中的一条记录，因此数据库操作也就转换成了对 Java 对象的操作。而这种数据与 Java 对象之间的映射通常可以获得自动化机制的支持，从而将开发人员从基于 JDBC 的复杂开发中解脱出来。EJB 中的实体构件提供的持久化实现方案就属于这一种，其它比较常用的还有 Hibernate 方案等。

到目前为止，在 EJB 规范的三次主要版本（1.x、2.x 与 3.x）变化中，每次主要版本升级都对实体构件相关规范进行了较大修改，读者应注意不同版本规范下的实体构件编程模式有所不同，下面通过两个简单的例子演示如何进行 EJB1.x、EJB2.x 实体构件开发，EJB3.0 为实体构件提供了更为强大的 ORM 映射机制，与 Hibernate 有较大类似，我们将在第 9 章介绍 EJB3.0 时进行简单的讨论。

EJB 中的实体构件又分为 CMP（Container Managed Persistence，容器维护的持久性）构件与 BMP（Bean Managed Persistence，Bean 维护的持久性）构件，其中 CMP 构件的相关数据库操作由容器自动完成，BMP 构件的相关数据库操作由开发人员在构件实现代码中通过 JDBC 编程实现，本节主要关注 CMP 构件的开发。

8.4.2 开发与使用 EJB1.1 实体构件

本小节通过开发一个简单的 EJB1.1 实体构件，实现对数据库税率表中的数据操作的封装，进行实体构件开发时通常首先根据数据库表结构抽象出实体构件的结构，本例中涉及的税率表中包含州代码与税率两个字段，其结构如图 8-26 所示。以下简称该 EJB 构件为税率

TaxBeanTable（记录了每个州的税率）	
stateCode	文本类型；州代码（主键）
taxRate	符点类型；税率

EJB。

图 8-26 税率表的结构

1. 定义 Remote 接口

Remote 接口包含 EJB 构件实现的商业方法的声明，对于实体构件来说，由于实体构件的对象代表了数据库表中的记录，因此 Remote 接口实际上包含数据库记录上能够执行的操作的声明。程序 8-13 给出了税率 EJB 的 Remote 接口定义：

程序 8-13　税率 EJB 的 Remote 接口定义

```
package Data;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Tax extends EJBObject
{
    public void setTaxRate(float taxRate) throws RemoteException;
    public float getTaxRate() throws RemoteException;
}
```

在程序 8-13 中，我们定义了一个名为 Tax 的 Remote 接口，税率 EJB 构件（代表税率表中的记录）向客户端提供设置税率与获取税率等商业方法，Remote 接口中包含这些方法的声明：

- **setTaxRate**：设置税率。调用者提供一个表示新税率的参数，没有返回值。
- **setTaxRate**：获取税率。没有参数，返回当前记录的税率字段值。

可以看到，按照 EJB 规范的约定，接口 Tax 继承了接口 EJBObject，接口中的每个操作均抛出 RemoteException 异常以报告远程调用错误，所有操作的参数与返回值均为合法的 Java RMI 类型。

2. 定义 Home 接口

Home 接口中包含 EJB 构件生命周期管理的相关方法，客户程序使用 Home Interface 创建、查找或删除 EJB 的实例，对于实体构件来说，由于实体构件的对象代表了数据库表中的记录，因此 Home 接口中的操作实际用于数据库表中记录的创建（插入）、查找与删除。程序 8-14 给出了税率 EJB 的 Home 接口定义：

程序 8-14　税率 EJB 的 Home 接口定义

```
package Data;
import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;
public interface TaxHome extends EJBHome
{
    public Tax create(String stateCode, float taxRate)
        throws RemoteException, CreateException;
    public Tax findByPrimaryKey(String primaryKey)
        throws RemoteException, FinderException;
    public Collection findInRange(float lowerLimit, float upperLimit)
        throws RemoteException, FinderException;
}
```

在程序 8-14 中，我们定义了一个名为 TaxHome 的 Home 接口，按照 EJB 规范的约定，

接口 TaxHome 继承了接口 EJBHome。该接口中声明了以下三个方法：

- create：对于 CMP 实体构件来说，客户端调用 Home 接口中的 create 方法会导致在数据库中插入记录，该操作需要两个分别表示州代码与税率的参数，其返回值为对应 EJB 构件的 Remote 接口类型——Tax，同时 create 方法抛出 RemoteException 和 CreateException 异常，通常每个实体构件的 Home 接口中都包含一个类似的 create 方法。
- findByPrimaryKey：根据给定的主键值查找对应记录。由于税率表的主键为州代码字段，因此该操作需要一个表示州代码的参数，由于主键唯一标识一条记录，因此该操作返回一条记录的引用——Remote 接口 Tax。该操作属于查找定位记录的相关操作，实体构件 Home 接口的查找定位操作需要抛出 FinderException，该操作用于报告查找定位错误（如未找到符合条件的记录），通常每个实体构件的 Home 接口中都包含一个类似的 findByPrimaryKey 方法。
- findInRange：查找税率在某给定范围（lowerLimit~upperLimit）内的记录。该操作也属于查找定位记录的相关操作，与特定表结构和希望执行的操作有关。由于符合条件的记录可能有多条，因此该操作返回一个记录引用（Remote 接口 Tax）的集合——Collection。实体构件的 Home 接口中可根据需要包含若干个类似的查找定位操作。

与会话构件不同，实体构件的 Home 接口中通常包含若干查找定位记录的相关操作（findByPrimaryKey 与其它自定义查找定位操作），因为数据库表中的记录经常需要进行查找定位。

3. 定义 Enterprise Bean 类

税率 EJB 的 Enterprise Bean 类首先要按照 Remote 接口的约定实现商业方法，其次要实现 Home 接口中 create 方法对应的 ejbCreate 方法与实体构件生命周期相关的方法。程序 8-15 给出了税率 EJB 的 Enterprise Bean 类定义：

程序 8-15　税率 EJB 的 Enterprise Bean 类定义

```
package Data;

import java.util.*;
import javax.ejb.*;

public class TaxBean implements EntityBean
{
    public String stateCode;
    public float taxRate;

    public void setTaxRate(float taxRate){
        this.taxRate = taxRate;
    }
    public float getTaxRate(){
        return this.taxRate;
    }
    public String ejbCreate(String stateCode, float taxRate)
            throws CreateException{
        System.out.println("TaxBean ejbCreate with stateCode=" + stateCode +
                           ",taxRate=" + taxRate);
        if(stateCode == null){
            throw new CreateException("The State Code is required.");
        }
        this.stateCode = stateCode;
        this.taxRate = taxRate;

        return null;
    }
    public void ejbPostCreate(String stateCode, float taxRate){
        System.out.println("TaxBean ejbPostCreate whith stateCode=" + stateCode
```

```
                                + ",taxRate=" + taxRate);
    }
    public void ejbLoad(){
        System.out.println("TaxBean ejbLoad whith stateCode=" + stateCode +
                                ",taxRate=" + taxRate);
        if(stateCode != null)
            stateCode.trim();
    }
    public void ejbStore() {
        System.out.println("TaxBean ejbStore whith stateCode=" + stateCode +
                                ",taxRate=" + taxRate);
    }
    public void ejbRemove() {
        System.out.println("TaxBean ejbRemove");
    }
    public void unsetEntityContext() {}
    public void setEntityContext(EntityContext context) {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
}
```

在程序 8-15 中，我们定义了一个名为 TaxBean 的 Enterprise Bean 类，由于该 EJB 构件是实体构件，因此 Enterprise Bean 类实现（implements）EntityBean 接口。

由于实体构件代表数据库表中的记录，因此首先在 Enterprise Bean 类中定义与记录的字段一一对应的数据成员，在本例中，我们定义了与州代码字段对应的数据成员 stateCode 和与税率字段对应的数据成员 taxRate。

与会话构件类似，该 Enterprise Bean 类也按照 Remote 接口 Tax 的约定实现了商业方法。实现商业方法时，由于 CMP 实体构件的数据库相关操作由容器自动完成，因此开发人员并不需要编写基于 JDBC 的代码来操作数据库，而是将对应字段上的数据库操作直接转嫁到与字段一一对应的数据成员上来：

- **setTaxRate**：设置税率。直接根据参数修改数据成员 taxRate。
- **setTaxRate**：获取税率。直接将数据成员 taxRate 的值返回。

其次，该 Enterprise Bean 类还实现了以下实体构件生命周期相关的方法（包括与 Home 接口中 create 方法对应的 ejbCreate 方法）：

- **ejbCreate**：与 Home 接口中的 create 方法对应。对于 CMP 实体构件来说，客户端调用 Home 接口中的 create 方法会导致在数据库中插入记录，容器在插入记录之前会调用对应对象的 ejbCreate 方法。对象通常在 ejbCreate 方法中将与字段一一对应的数据成员的值进行设置，为容器完成记录的插入做好准备，本例中根据对数完成数据成员 stateCode 与 taxRate 的设置。与会话构件不同，实体构件的 ejbCreate 方法返回对应数据库表的主键类型，本例中为州代码字段的类型 String，对于 CMP 实体构件来说，由于主键在容器记录插入完成后才生成，因此要求 ejbCreate 返回空（null）。
- **ejbPostCreate**：与 Home 接口中的 create 方法对应。实体构件要求其 Enterprise Bean 类除了实现与 create 方法对应的 ejbCreate 之外，还要实现一个对应的 ejbPostCreate 方法，容器在完成记录的插入后会调用该方法。
- **ejbLoad**：容器将数据从数据库中读取到实体构件对象中后，会调用该对象的 ejbLoad 方法，在此方法中可以对读出的数据进行一些预处理，本例在 ejbLoad 方法中去除州代码字段中末尾的空白字符。
- **ejbStore**：容器将数据从实体构件对象中存储到数据库中之前，会调用该对象的 ejbStore 方法。在此方法中可以对要存储的的数据进行一些预处理。
- **ejbRemove**：容器将数据数据库中删除之前，会调用对应对象的 ejbRemove 方法。在此方法中可以进行一些整理工作， 对于 CMP 实体构件，客户调用 Remote 接口中的 remove 方法会导致删除数据库中的数据。
- **setEntityContext 与 unsetEntityContext**：与会话构件类似，setEntityContext 方法用来初始化 EJB 使用的 EntityContext 变量，EntityContext 是实体构件与容器交互的

入口，每次创建一个实体构件的对象时，容器会调用该对象的 setEntityContext 方法，给对象传入使用的 EntityContext 变量；unsetEntityContext 容器删除实体构件对象之前被调用，可以在该方法中释放占用的系统资源。

- **ejbActivate 与 ejbPassivate**：与有状态会话构件方法类似。

如图 8-27 所示，实体构件的生命周期包含三个状态：就绪状态（Ready State）、不存在状态（No State）与池状态（Pooled State）。不存在状态表明 EJB 容器中不存在对应实体构件的实例，处于不存在状态的实例还未被创建；池状态表明实体构件的实例存在于实例池中，容器新创建的实例会进入这个状态，处于池状态的实体构件实例不与任何 EJB 对象关联，其中与数据库记录对应的字段均未实例化，还未和数据库中的记录对应起来因此不能用于完成相应的数据库操作。就绪状态表明实体构件实例建立了与 EJB 对象的关联，已经和数据库记录对应起来，可以处理客户应用的请求，客户端程序调用 Home 接口中的方法创建或查找到某个实体构件实例时，该实例会从池状态进入就绪状态。实体构件生命周期相关方法的调用时机如图 8-27 中的标记所示。

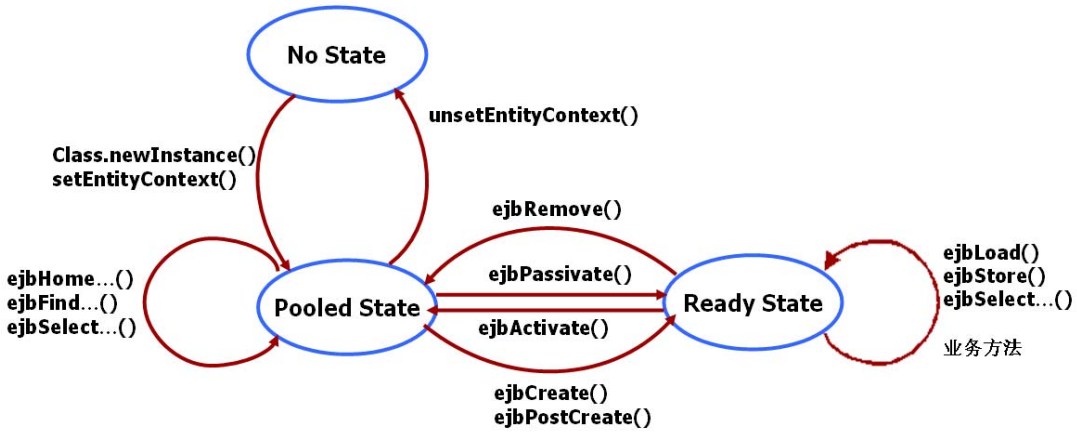


图 8-27 实体构件生命周期

4. 编译源代码

利用 Java 语言编译器 javac 对税率 EJB 的 Home 接口、Remote 接口与 Enterprise Bean 类进行编译。

5. 打包 EJB

本例中打包 EJB 构件的基本过程与上节类似，不同的地方主要有几下几点：

- 本例将包含 EJB 构件的 J2EE 应用命名为 TaxBeanApp。
- 本例中将 EJB 类型设置为实体构件（Entity），如图 8-28 所示：

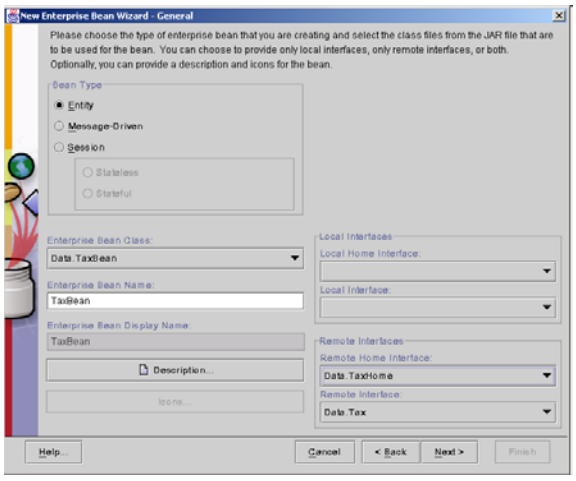


图 8-28 将 EJB 构件类型设置为实体构件

- 实体构件比会话构件多一页配置界面用于配置实体构件的持久性相关信息，如图 8-29 所示，本例中关于持久性相关信息设置为：
  - 选择实体构件类型为 EJB1.x 规范的 CMP 构件：在“Persistence Management”选择“Container managed persistence（1.0）”；
  - 选择 stateCode 与 taxRate 均为与数据库表字段对应的数据成员：在“Fields To Be Persisted”中选中 stateCode 与 taxRate 前面的复选框；
  - 设置主键类型为字符串类型：在“Primary Key Class”下面的文本框中输入“java.lang.String”；
  - 选择主键字段为 stateCode：在“Primary Key Field Name”下面的列表中选择 stateCode。

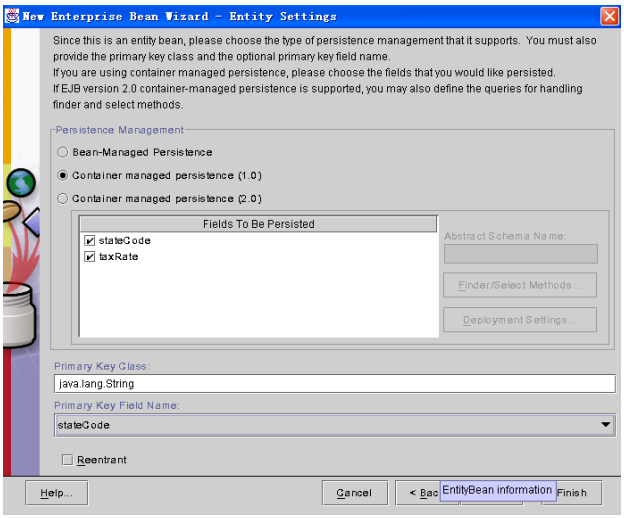


图 8-29 设置税率 EJB 的持久性信息

- 如图 8-30 所示，CMP 实体构件的事务控制规则只能容器维护，由于本例中不讨论事务控制，因此对应方法的事务属性可以全部使用缺省值。

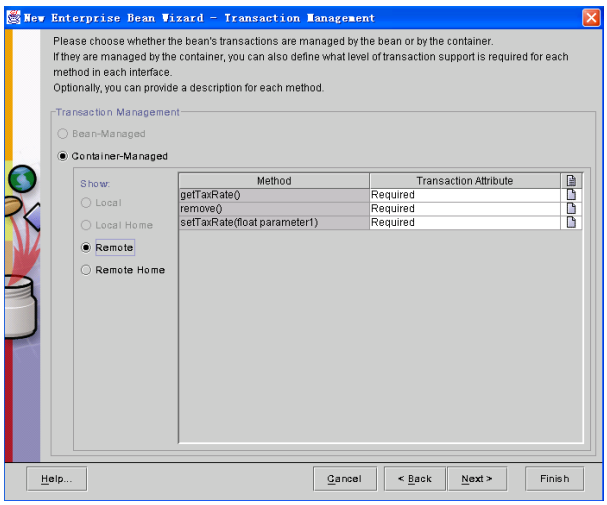


图 8-30 设置税率 EJB 的事务控制规则

- 为税率 EJB 配置名为“TaxBean”的 JNDI 名。

6. 设置数据库信息

对于 CMP 实体构件而言，需要在布署前设置相关的数据库信息，本例中我们需要为税率 EJB 指定对应数据库表所存储的数据库，生成并补充容器执行数据库操作时需要使用的 SQL 语句。

首先要设置的实体构件对应表存储所用的数据库，数据库的设置通过指定相应的数据源（Data Source）实现。J2EE 中的数据源是一种代表数据库的系统资源，通常一个数据源对应一个物理的数据库，通过数据源访问数据库可以利用前面提到的数据库连接池达到较好的数据库访问效率。J2EE 参考实现中的数据源可在服务器配置界面中进行配置，如图 8-31 所示，可通过点击 deploytool 的 Tools\Server Configuration 菜单打开服务器配置界面，在图 8-32 所示的服务器配置界选中 Data Sources 下的 Standard，可以看到 J2EE 服务器上已经配置的缺省数据源，每个数据源包含一个 JNDI 名供使用者查找，其中名为 jdbc/Cloudscape 的数据源为对应 cloudscape 缺省数据库 CloudscapeDB 的数据源，本例中我们将使用该数据源来指定税率表存储所在的数据库。

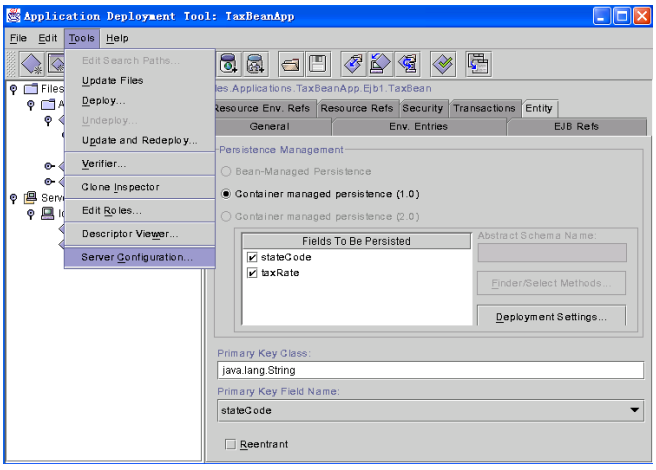


图 8-31 打开服务器配置界面

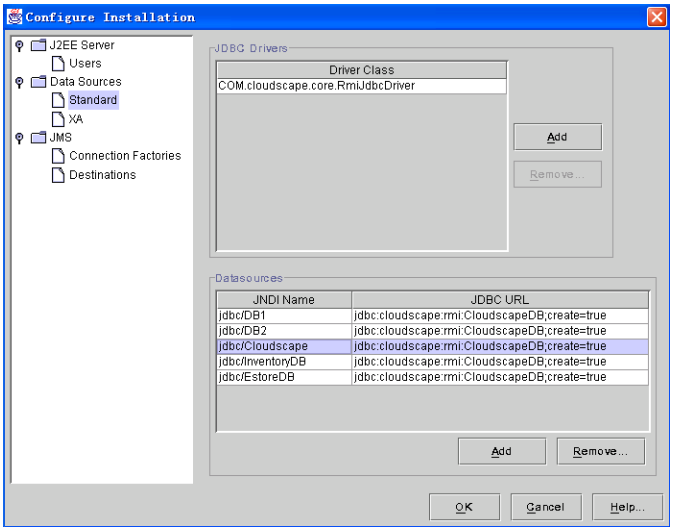


图 8-32J2EE 服务器上的数据源

了解了要使用的数据源后，选中刚刚打包的税率 EJB，在图 8-33 所示的界面右侧选中“Entity”选项卡，可以看到刚刚在打包过程中设置的税率 EJB 的持久性相关设置，点击“Deployment Setting”按钮可打开数据库配置向导。

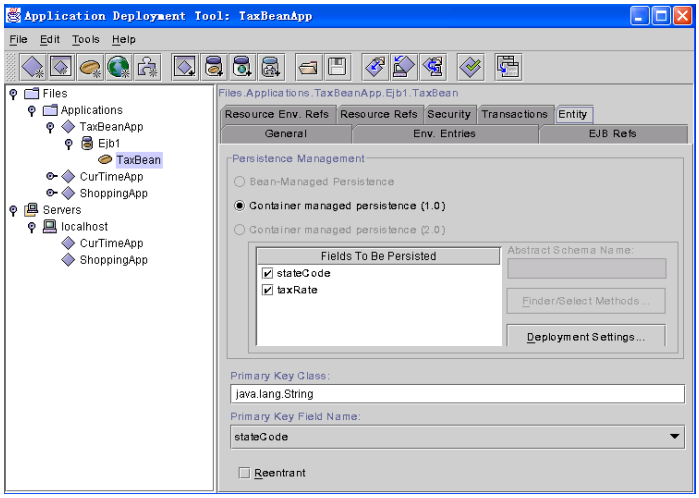


图 8-33 打开数据库配置界面

在图 8-34 所示的数据库配置界面中点击“Database Settings”按钮，并在弹出的对话框中输入期望使用的数据库对应数据源的 JNDI 名，本例中为“jdbc/Cloudscape”，点击 OK 关闭对话框。

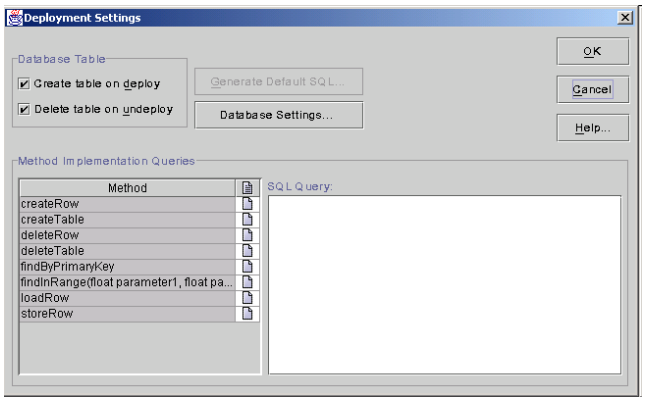


图 8-34 数据库配置界面

然后在图 8-34 所示的数据库配置界面中点击“Generate Default SQL”按钮（该按钮会在设置完数据源后变为可用状态），可以生成容器完成数据库相关操作所使用的缺省 SQL 语句。选中界面中的 createRow、createTable 等方法，可以看到对应操作使用的 SQL 语句，在这些方法中，由于 findInRange 为自定义的查找方法，容器不可能知道开发人员真正想执行的操作，因此我们需要为该方法补充对应的 SQL 语句，如图 8-35 所示，选中方法 findInRange，并为其补充 SQL 语句的 WHERE 子句“WHREE “taxRate” BETWEEN ?1 AND ?2”。

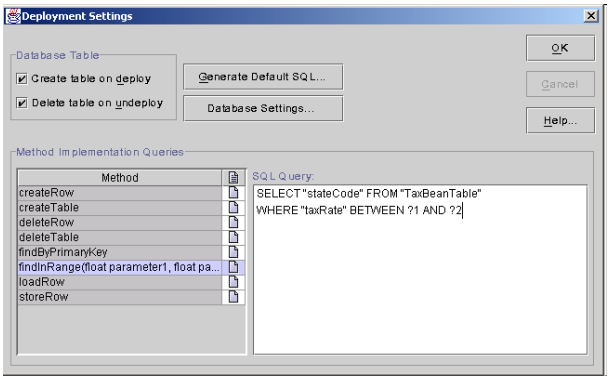


图 8-35 为 findInRange 方法补充 SQL 语句

7. 布署 EJB

本例中布署 EJB 构件的基本过程与上节类似，布署时生成名为“StatefulAppClient.jar”的客户端 JAR 文件包，并将其存放到后面客户端程序所在的目录下。

布署时应注意必须首先启动 cloudscape。

布署完成后我们可以在 cloudscape 数据库中发现一个被自动创建的表“TaxBeanTable”，这是在布署时容器自动根据税率 EJB 的相关信息创建的，该表包含 stateCode 与 taxRate 两个字段。如果在布署之前已存在对应的数据库表，读者可以在图 8-35 所示的界面中取消“Create table on deploy”复选框来选择禁止容器自动创建表。

8. 开发/运行客户端程序

本例中我们编写一个名为 TaxClient 的 Java 类作为客户端，该类包含一个入口函数 main。该客户端程序调用利用税率 EJB 完成对数据库中税率表中记录的操作。客户端程序的代码如程序 8-16 所示。

程序 8-16　税率 EJB 的客户端程序

```
import Data.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class TaxClient{
    public static void main(String[] args){
        try{
            Context initial = new InitialContext();
            Object objRef = initial.lookup("TaxBean");
            TaxHome home = (TaxHome) PortableRemoteObject.narrow(
                                objRef, TaxHome.class);
            Tax tax = null;

            tax = home.create("IL", 5.00f);
            tax = home.create("CA", 6.25f);
            tax = home.create("FL", 8.50f);
            tax = home.create("CO", 6.75f);

            tax = home.findByPrimaryKey("CA");

            System.out.println("CA tax rate: " + tax.getTaxRate());
            System.out.println("Changing tax rate for CA state");
            tax.setTaxRate(8.25f);

            System.out.println("New CA tax rate: " + tax.getTaxRate());

            Collection taxArray = home.findInRange(5.0f, 7.0f);

            Iterator it = taxArray.iterator();
            while(it.hasNext()){
                Object objRef2 = it.next();
                tax = (Tax)PortableRemoteObject.narrow(objRef2, Tax.class);
                System.out.println("Tax Rate in " + tax.getPrimaryKey() + ": "
                                    + tax.getTaxRate());
                tax.remove();
            }
        }catch(Exception ex){
            System.err.println("Caught an exception.");
            ex.printStackTrace();
        }
    }
}
```

在程序 8-16 所示的客户端程序中，首先利用 JNDI 服务查找税率 EJB 对应的 Home 接口。查找到 Home 接口后，客户端连续调用 Home 接口中的 create 操作向数据库中插入四条记录；然后调用 Home 接口的 findByPrimaryKey 方法定位到主键（州代码）为“CA”的记录，将其税率修改为 8.25；最后调用 Home 接口的 findInRange 方法查找税率在 5.0 与 7.0 之间的记录（有 IL 与 CO 两条），将这些记录的信息输出，然后调用 Remote 接口的 remove 操作将其删除。

第一次运行客户端程序，客户端的输出信息如下：

```
CA tax rate: 6.25
Changing tax rate for CA state
New CA tax rate: 8.25
Tax Rate in IL: 5.0
Tax Rate in CO: 6.75
```

读者可进入 cloudscape 的交互式 sql，通过执行以下 SQL 查看 TaxBeanTable 中的记录变化：

```
ij> select * from "TaxBeanTable";
```

第一次运行客户端之后该表中的数据为：

stateCode	taxRate
CA	8.25
FL	8.5

8.4.3 开发 EJB2.0 实体构件

EJB2.0 的实体构件与 EJB1.1 的实体构件之间的主要区别如下：

- Enterprise Bean 类的区别：在 EJB1.1 中，Enterprise Bean 类由开发人员定义，而在 EJB2.0 中，Enterprise Bean 类由容器生成，开发人员仅定义一个抽象基类。
- Enterprise Bean 数据成员的区别：
  - 在 EJB2.0 中与数据库字段对应的 Bean 属性不由用户定义，用户仅定义对应的 set 和 get 方法，具体属性的定义由容器生成，这样容器可以对属性进行优化。
  - 在 EJB2.0 的 CMP 构件中，还有一种特殊的字段，cmr（Container Managed Relationship）字段，用于关联其它的表（实体构件）。在组装/部署时，可以设置由容器自动维护表之间的关联关系。
- 接口区别：EJB2.0 引入了本地接口，实体构件的进程内客户端可以通过本地接口获得更好的调用效率。

下面通过一个简单的例子来演示如何开发 EJB2.0 的实体构件。我们将开发一个名为 OrderApp 的 J2EE 应用，该应用中包含两个 EJB2.0 的实体构件，分别对应数据库中的订单表与送货地址表，这两个表的结构如图 8-36 所示。以下简称对应订单表的实体构件为订单 EJB，简称对应送货地址表的实体构件为地址 EJB。

订单表与送货地址表之间存在一个外键关联关系，订单表中并不直接记录该订单的送货地址信息，而是仅记录一个送货地址的编号，具体的送货地址信息在送货地址中存储。

在本例中，订单 EJB 使用地址 EJB 存放订单的送货地址信息，客户端程序仅看到订单 EJB，根据订单 EJB 间接获取或存储送货地址信息。由于地址 EJB 仅被订单 EJB 所使用，而本例会把两个 EJB 构件打包在同一个 EJB 模块中，布署时会被布署到同一个容器内，因此我们可以为地址 EJB 提供本地接口，订单 EJB 可通过地址 EJB 的本地接口获得较高的访问效率。

OrderBeanTable（订单表）

orderID	长整型；订单编号
customerName	文本类型；客户姓名
shipAddrAddressID	整型；送货地址编号

AddressBeanTable（地址表）

addressID	整型；地址编号
street	文本类型；街道
city	文本类型；城市
state	文本类型；州
zip	文本类型；邮政编码
orderOrderID	长整型；订单编号

图 8-36 订单表与送货地址表结构

1. 开发地址 EJB

首先定义地址 EJB 的 Local 接口，Local 接口完成与 Remote 接口类似的功能，本例中地址 EJB 的 Local 接口约定地址表中记录上能够执行的操作。程序 8-17 给出了地址 EJB 的 Local 接口定义：

程序 8-17　地址 EJB 的 Local 接口定义

```
package examples.local_objects;
import javax.ejb.EJBLocalObject;

public interface AddressLocal extends EJBLocalObject{
    public int getAddressID();

    public String getStreet();
    public void setStreet(String street);

    public String getCity();
    public void setCity(String city);

    public String getState();
    public void setState(String state);

    public String getZip();
    public void setZip(String zip);
}
```

在程序 8-17 中，我们定义了一个名为 AddressLocal 的 Local 接口，地址 EJB 构件（代表地址表中的记录）向客户端提供读取地址标识、读取与设置街道信息、城市信息、州信息与邮编信息等商业方法，Local 接口中包含这些方法的声明。

可以看到，按照 EJB 规范的约定，接口 AddressLocal 继承了接口 EJBLocalObject，由于 Local 接口中约定的是本地操作，因此接口中的每个操作不必抛出 RemoteException 异常，所有操作的参数与返回值也不必是 Java RMI 类型。

LocalHome 接口完成与 Home 接口类似的功能。地址 EJB 的 LocalHome 接口用于约定实现地址表中记录的创建（插入）、查找与删除的操作。程序 8-18 给出了地址 EJB 的 LocalHome 接口定义：

程序 8-18　地址 EJB 的 LocalHome 接口定义

```
package examples.local_objects;
import javax.ejb.*;

public interface AddressLocalHome extends EJBLocalHome{
    public AddressLocal findByPrimaryKey(Integer addressID)
            throws FinderException;
```

```
    public AddressLocal create(int addressID, String street, String city,
            String state, String zip) throws CreateException;
}
```

在程序 8-18 中，我们定义了一个名为 AddressLocalHome 的 LocalHome 接口，按照 EJB 规范的约定，接口 AddressLocalHome 继承了接口 EJBLocalHome。该接口中声明了以下两个方法：

- create：对于 CMP 实体构件来说，客户端调用 LocalHome 接口中的 create 方法会导致在数据库中插入记录，该操作需要表示详细地址信息的五个参数，其返回值为对应 EJB 构件的 Local 接口类型——AddressLocal，同时 create 方法抛出 CreateException 异常。
- findByPrimaryKey：根据给定的主键值查找对应记录。由于地址表的主键为地址标识字段，因此该操作需要一个表示地址标识的参数，返回一条记录的引用——Local 接口 AddressLocal。

对于 EJB2.0 的实体构件而言，开发人员并不定义真正的 Enterprise Bean 类，而是仅定义其抽象基类。程序 8-19 给出了地址 EJB 的 Enterprise Bean 类的抽象基类定义：

程序 8-19　地址 EJB 的 Enterprise Bean 类的抽象基类定义

```
package examples.local_objects;
import javax.ejb.*;

public abstract class AddressBean implements EntityBean{
    public abstract int getAddressID();
    public abstract void setAddressID(int addressID);

    public abstract String getStreet();
    public abstract void setStreet(String street);

    public abstract String getCity();
    public abstract void setCity(String city);

    public abstract String getState();
    public abstract void setState(String state);

    public abstract String getZip();
    public abstract void setZip(String zip);

    public Integer ejbCreate(int addressID, String street, String city,
         String state, String zip) throws CreateException{
        System.out.println("entry AddressBean ejbCreate");
        setAddressID(addressID);
        setStreet(street);
        setCity(city);
        setState(state);
        setZip(zip);
        return null;
    }

    public void ejbPostCreate(int addressID, String street, String city,
         String state, String zip) throws CreateException {}
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
}
```

在程序 8-19 中，我们定义了一个名为 AddressBean 的抽象类，由于该 EJB 构件是实体

构件，因此类 AddressBean 实现（implements）EntityBean 接口。

由于实体构件代表数据库表中的记录，因此首先要在类 AddressBean 中定义与记录的字段一一对应的数据成员，但是与 EJB1.1 的实体构件不同，在 EJB2.0 的实体构件中，开发人员并不定义真正的数据成员，而是仅定义数据成员对应的 set 与 get 方法，地址表中包含地址标识（AddressID）、街道信息（Street）、城市信息（City）、州信息（State）与邮编信息（Zip）等五个字段，因此程序 8-19 中定义了五对相应的 set 与 get 方法。

由于地址 EJB 的 Local 接口中所约定的方法均与某个数据成员对应的 set 或 get 方法相同，因此类 AddressBean 不需要提供对应方法的实现，数据成员对应的 set 与 get 方法由容器生成的真正的 Enterprise Bean 类实现。

类 AddressBean 还实现了与 LocalHome 接口的 create 方法对应的 ejbCreate 方法与 ejbPostCreate 方法，在 ejbCreate 方法中，利用数据成员对应的 set 方法将数据成员准备好。此外类 AddressBean 还实现了实体构件相关的其它生命周期管理相关的方法。

2. 开发订单 EJB

由于订单 EJB 要被客户端进行远程调用，因此仍需使用远程接口。

订单 EJB 的 Remote 接口约定订单表中记录上能够执行的操作。程序 8-20 给出了订单 EJB 的 Remote 接口定义：

程序 8-20　订单 EJB 的 Remote 接口定义

```
package examples.local_objects;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Order extends EJBObject{
    public String getCustomerName() throws RemoteException;
    public AddressValueObject getShipAddressView() throws RemoteException;
}
```

在程序 8-20 中，我们定义了一个名为 Order 的 Remote 接口，订单 EJB 构件（代表订单表中的记录）向客户端提供获取客户姓名与获取送货地址两个商业方法，Remote 接口中包含这些方法的声明。应注意获取送货地址方法并没有返回对应地址 EJB 的引用，而是将地址信息用一个普通的可串行化对象返回，该串行化对象对应的类定义请参阅程序 8-23。

可以看到，按照 EJB 规范的约定，接口 Order 继承了接口 EJBObject，接口中的每个操作均抛出 RemoteException 异常，所有操作的参数与返回值均为合法的 Java RMI 类型。

订单 EJB 的 Home 接口用于约定实现订单表中记录的创建（插入）、查找与删除的操作。程序 8-21 给出了订单 EJB 的 Home 接口定义：

程序 8-21　订单 EJB 的 Home 接口定义

```
package examples.local_objects;

import java.rmi.*;
import javax.ejb.*;

public interface OrderHome extends EJBHome{
    public Order create(long orderID, String customerName, AddressValueObject
          shipAddress) throws RemoteException, CreateException;
    public Order findByPrimaryKey(Long orderID)
         throws RemoteException, FinderException;
}
```

在程序 8-21 中，我们定义了一个名为 OrderHome 的 Home 接口，按照 EJB 规范的约定，接口 OrderHome 继承了接口 EJBHome。该接口中声明了以下两个方法：

- create：对于 CMP 实体构件来说，客户端调用 Home 接口中的 create 方法会导致在数据库中插入记录，该操作需要表示详细订单信息的三个参数（地址对数为一可串行化对象），其返回值为对应 EJB 构件的 Remote 接口类型——Order，同时 create 方法抛出 RemoteException 异常与 CreateException 异常。

- findByPrimaryKey：根据给定的主键值查找对应记录。由于订单表的主键为订单标识字段，因此该操作需要一个表示订单标识的参数，返回一条记录的引用——Remote 接口 Order。

对于 EJB2.0 的实体构件而言，开发人员并不定义真正的 Enterprise Bean 类，而是仅定义其抽象基类。程序 8-22 给出了订单 EJB 的 Enterprise Bean 类的抽象基类定义：

程序 8-22　订单 EJB 的 Enterprise Bean 类的抽象基类定义

```
package examples.local_objects;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.naming.InitialContext;

public abstract class OrderBean implements EntityBean{
    //CMP fields
    public abstract long getOrderID();
    public abstract void setOrderID(long orderID);

    public abstract String getCustomerName();
    public abstract void setCustomerName(String customerName);

    //cmr-fields
    public abstract AddressLocal getShipAddress();
    public abstract void setShipAddress(AddressLocal shipAddress);

    public AddressValueObject getShipAddressView(){
        AddressLocal shipAddress = getShipAddress();
        return new AddressValueObject(shipAddress.getAddressID(),
                                      shipAddress.getStreet(),
                                      shipAddress.getCity(),
                                      shipAddress.getState(),
                                      shipAddress.getZip());
    }

    public Long ejbCreate(long orderID, String customerName,
           AddressValueObject shipAddress) throws CreateException{
        System.out.println("entry ejbCreate");
        setOrderID(orderID);
        setCustomerName(customerName);
//      setShipAddress(shipAddress);
        return null;
    }
    private AddressLocal createShipAddress(int addressId, String street,
                   String city, String state, String zip){
        try{
            InitialContext initial = new InitialContext();
            AddressLocalHome home = (AddressLocalHome)
                     initial.lookup("java:comp/env/ejb/AddressEJB");
            return home.create(addressId, street, city, state, zip);
        }catch(Exception e){
            throw new javax.ejb.EJBException(e);
        }
    }
//  public void ejbPostCreate(long orderID, String customerName)
    public void ejbPostCreate(long orderID, String customerName,
               AddressValueObject shipAddressView) throws CreateException{
        AddressLocal shipAddress = createShipAddress(
            shipAddressView.getAddressID(), shipAddressView.getStreet(),
            shipAddressView.getCity(), shipAddressView.getState(),
            shipAddressView.getZip());
```

```
        setShipAddress(shipAddress);
    }
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
}
```

在程序 8-22 中，我们定义了一个名为 OrderBean 的抽象类，由于该 EJB 构件是实体构件，因此类 OrderBean 实现（implements）EntityBean 接口。

类 OrderBean 中定义与记录的字段一一对应的数据成员相关的 set 与 get 方法，订单表中包含订单标识（OrderID）、客户姓名（CustomerName）两个基本字段，因此程序 8-22 中定义了两对相应的 set 与 get 方法。

另外在类 OrderBean 定义了一个 CMR（Container Managed Relationship）字段对应的一对 set 与 get 方法，CMR 字段用于和其它的实体构件（其它表中的记录）关联，CMR 字段的类型为所关联的实体构件的 Remote 或 Local 接口；本例中通过本地接口 AddressLocal 访问另一个实体构件——地址 EJB。

由于订单 EJB 的 Remote 接口中所约定的方法 getShipAddressView 并不属于某个数据成员对应的 set 或 get 方法相同，因此类 OrderBean 中按照 Remote 接口的约定实现了该方法，该方法根据所关联的地址 EJB 创建一个可串行化对象返回给调用者。

类 OrderBean 还实现了与 Home 接口的 create 方法对应的 ejbCreate 方法与 ejbPostCreate 方法。在 ejbCreate 方法中，利用数据成员对应的 set 方法将基本数据成员准备好；在 ejbPostCreate 方法中设置 CMR 字段的值，具体的方法是首先调用地址 EJB 的 LocalHome 接口中的 create 方法创建地址记录，然后将对应的 Local 接口记录在 CMR 字段中。因为 CMR 字段代表两条记录之间的关联，所以 CMR 字段通常在记录插入完成后（如在 ejbPostCreate 方法中）才设置。

此外类 OrderBean 还实现了实体构件相关的其它生命周期管理相关的方法。

订单 EJB 中使用的可串行化对象对应的类定义如程序 8-23 所示：

程序 8-23　订单 EJB 使用的可串行化对象对应的类定义

```
package examples.local_objects;

public class AddressValueObject implements java.io.Serializable{
    private int addressID;
    private String street;
    private String city;
    private String state;
    private String zip;

    public AddressValueObject() {}
    public AddressValueObject(int addressID, String street, String city,
                String state, String zip){
        this.addressID = addressID;
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }
    public int getAddressID() {
        return addressID;
    }
    public void setAddressID(int addressID){
        this.addressID = addressID;
    }
```

```
    public String getStreet() {
        return street;
    }
    public void setStreet(String street){
        this.street = street;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city){
        this.city = city;
    }
    public String getState() {
        return state;
    }
    public void setState(String state){
        this.state = state;
    }
    public String getZip() {
        return zip;
    }
    public void setZip(String zip){
        this.zip = zip;
    }
}
```

3. 打包/布署 EJB

本例中打包的基本过程与上节类似，不同的地方主要有几下几点：

- 本例将包含 EJB 构件的 J2EE 应用命名为 OrderApp。
- 首先打包地址 EJB，在创建对应 EJB 模块时可一次性将两个 EJB 构件所需的所有 Java 目标文件加入到 EJB 模块中，如图 8-37 所示。

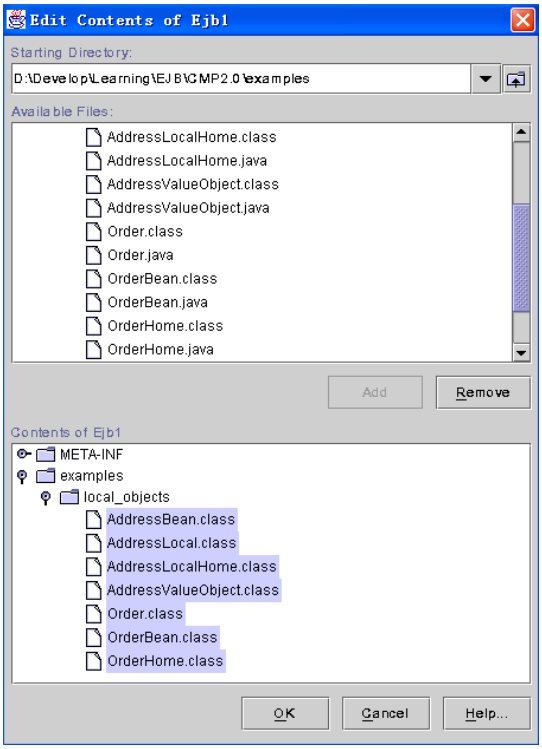


图 8-37 将 EJB 构件相关的 Java 目标文件添加到模块中

- 将设置地址 EJB 的类型为实体构件，注意其提供的是本地接口，如图 8-38 所示：

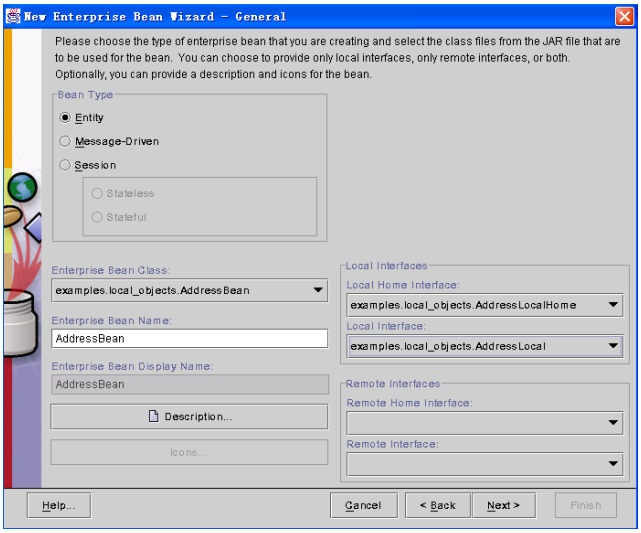


图 8-38 设置地址 EJB 的基本信息

- 如图 8-39 所示配置地址 EJB 的持久性信息：
  - 选择实体构件类型为 EJB2.x 规范的 CMP 构件：在“Persistence Management”选择“Container managed persistence（2.0）”；
  - 选择 state 等五个字段均为与数据库表字段对应的数据成员：在“Fields To Be Persisted”中选中每个字段前面的复选框；
  - 设置主键类型为整型：在“Primary Key Class”下面的文本框中输入“java.lang.Integer”；
  - 选择主键字段为 addressID：在“Primary Key Field Name”下面的列表中选择 addressID。

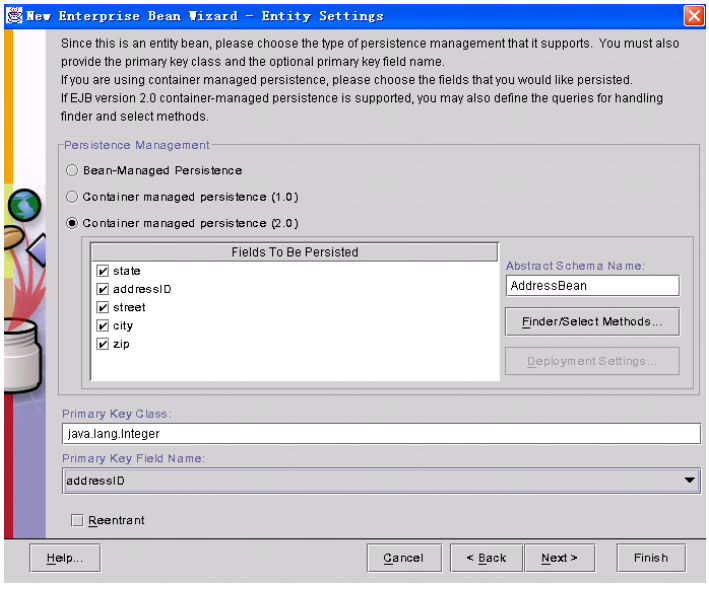


图 8-39 配置地址 EJB 的持久性信息

- 打包订单 EJB 时，选择将其加入已有 EJB 模块（打包地址 EJB 时创建的 EJB 模块）中，如图 8-40 所示：

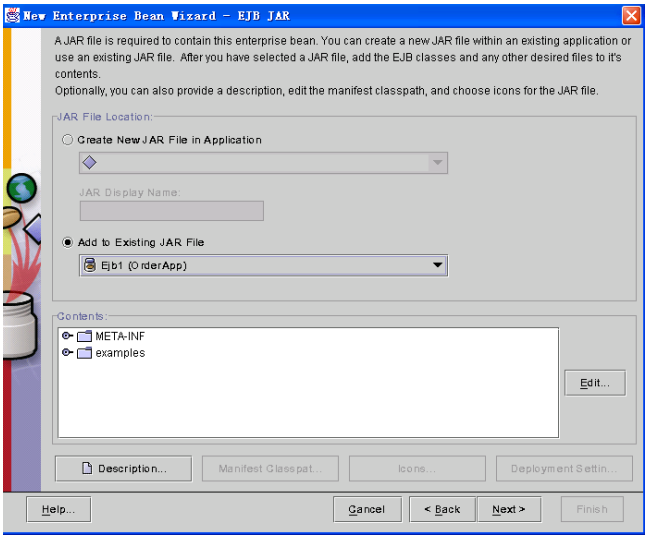


图 8-39 将订单 EJB 加入已有 EJB 模块

- 如图 8-40 所示配置订单 EJB 的持久性信息：
  - 选择实体构件类型为 EJB2.x 规范的 CMP 构件：在“Persistence Management”选择“Container managed persistence（2.0）”；
  - 选择 customerName 与 orderID 两个字段为与数据库表字段对应的数据成员：在“Fields To Be Persisted”中选中这两个字段前面的复选框；注意 shipAddress 为 CMR 字段，并不与数据库表字段直接对应。
  - 设置主键类型为长整型：在“Primary Key Class”下面的文本框中输入“java.lang.Long”；
  - 选择主键字段为 orderID：在“Primary Key Field Name”下面的列表中选择 orderID。

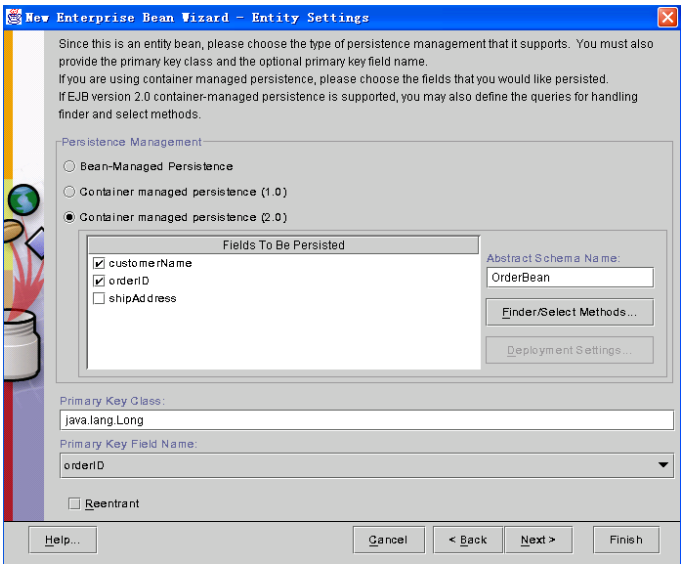


图 8-40 配置地址 EJB 的持久性信息

- 因为订单 EJB 引用了地址 EJB，因此在打包订单 EJB 时需要声明对地址 EJB 的引用。如图 8-41 所示，通过设定以下信息完成 EJB 引用的声明：
  - Coded Name：代码中引用对应 EJB 时所使用的 JNDI 名，本例中为“ejb/AddressEJB”；
  - Type：所引用的 EJB 的类型，本例中为实体构件（Entity）；

- Interfaces：所引用的 EJB 的接口类型，本例中为本地接口（Local）；
- Home Interface：所引用的 EJB 的 Home 接口名，本例中为“examples.local_objects.AddressLocalHome”；
- Local/Remote Interface：所引用的 EJB 的 Local 或 Remote 接口名，本例中为“examples.local_objects.AddressLocal”；
- Enterprise Bean Name：期望真正使用的 EJB，本例中为前面打包的地址 EJB。

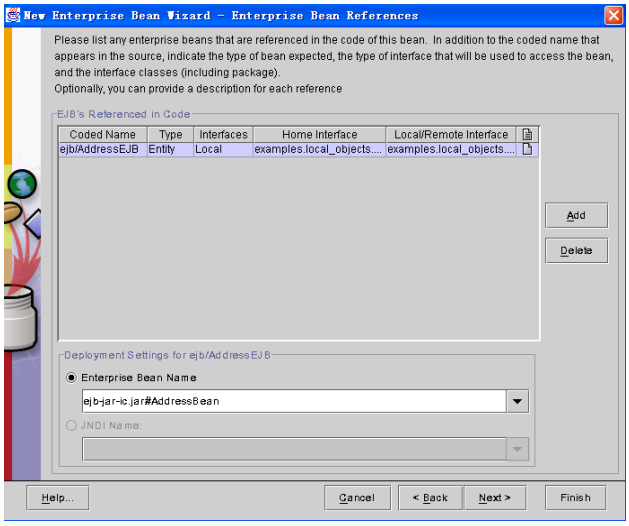


图 8-41 配置 EJB 引用

- 为地址 EJB 配置名为“AddressEJB”的 JNDI 名，为订单 EJB 配置名为“OrderBena”的 JNDI 名。
- 利用 CMR 字段设置 EJB 之间的关联，如图 8-42 所示，在 deploytool 中选中包含地址 EJB 和订单 EJB 的 EJB 模块，点击右边的 Relationships 选项卡，点击 Add 按钮可打开关联关系设置界面。

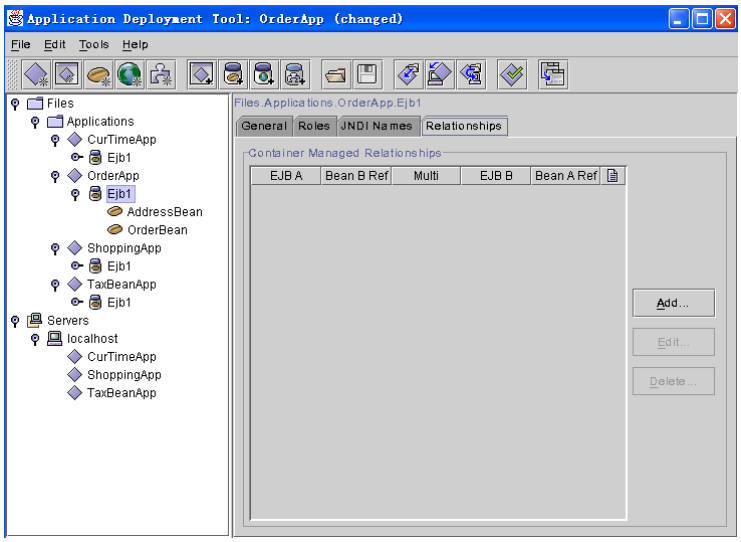


图 8-42 添加关联关系

- 在图 8-43 所示的界面中利用 CMR 字段 shipAddress 设置关联关系：

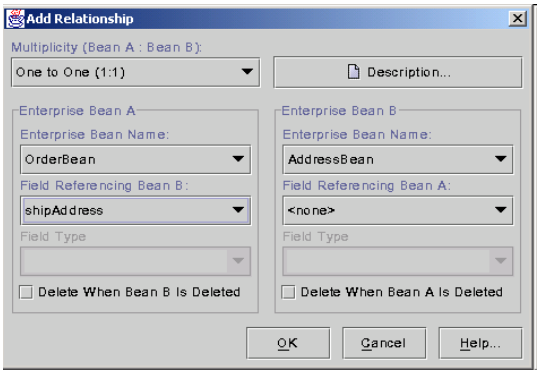


图 8-43 设置关联关系

- 地址 EJB 与订单 EJB 均使用数据源“jdbc/Cloudscape”，且均生成缺省 SQL 语句即可。
- 布署时生成名为“CMP20Client.jar”的客户端 JAR 文件包，并将其存放到后面客户端程序所在的目录下。布署完成后会在数据库 CloudscapeDB 中创建两张表 AddressBeanTable 与 OrderBeanTable。

4. 创建客户端程序

本例使用程序 8-24 给出的客户端程序。

程序 8-23　订单 EJB 使用的可串行化对象对应的类定义

```
import examples.local_objects.*;

import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class ClientDemoDependentObject{
    public static void main(String[] args){
        try{
            InitialContext initial = new InitialContext();
            OrderHome orderHome = (OrderHome)PortableRemoteObject.narrow(
                initial.lookup("OrderBean"), OrderHome.class);
            AddressValueObject obj1 = new AddressValueObject
                        (25, "1132 Decimal Ave", "Evanston", "IL", "60202");
            Order order = orderHome.create(2, "Puddentane", obj1);
            System.out.println(order.getCustomerName());
            AddressValueObject shipAddress = order.getShipAddressView();
            System.out.println(shipAddress.getStreet());
            System.out.println(shipAddress.getCity());
            System.out.println(shipAddress.getState());
            System.out.println(shipAddress.getZip());
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

客户端程序第一交执行结束后，会在订单表与地址表中分别插入一条记录。

思考与练习

8-1 某无状态会话构件客户端程序连续两次调用该 EJBHome 接口中的 create 方法，容器会创建几个该 EJB 的实例？为什么？

8-2 某有状态会话构件客户端程序连续两次调用该 EJBHome 接口中的 create 方法，容器会创建几个该 EJB 的实例？为什么？

8-3 在 8.3 节的例子中，如果客户端程序在退出前没有调用 Remote 接口中的 remove 操作，那么下次运行客户端程序容器是否还会创建新的对象？为什么？

8-4 EJB2.x 的实体构件与 EJB1.x 的实体构件有哪些主要区别？

8-5 实现一个提供“查询股票信息”功能的无状态会话组件以及相应的客户端程序。

8-6 实现一个提供“计数器”功能的有状态会话构件，提供“计数器清 0”、“增加计数值”、“减少计数值”、“获取当前计数值”等操作。实现一个简单的客户端程序测试对该 EJB 进行测试。

第 9 章　EJB 高级特性

本章首先通过三个例子讨论如何使用 J2EE 中间件提供的环境条目、事务控制、安全性控制等公共服务，最后对 EJB3.0 简单介绍。

§9.1 环境条目

J2EE 中的环境条目（Environment Entry）由“名字-值”对组成，是 J2EE 提供的源代码以外的可定制性支持之一，在 EJB 代码中可以根据环境条目的不同取值执行不同的操作，当环境条目的值需要改变时，只需修改对应环境条目的配置而不需要修改 EJB 构件的源代码。

本节通过一个简单例子演示如何配置与使用环境条目。我们将创建一个名为 ConverterApp 的 J2EE 应用，该应用中仅包含一个简单的无状态会话构件，提供简单的币值换算的功能。币值换算要用到汇率，汇率是一个会经常变化的值，因此将汇率声明为环境条目，在部署时可以修改汇率，EJB 构件提供服务时根据 EJB 环境中的汇率值进行币值换算。

9.1.1　开发 EJB 构件

该 EJB 构件实现币值换算的功能，由于该构件的实例（对象）不需要保存与特定客户端相关的会话状态，因此设计为无状态的会话构件。以下简称该 EJB 构件为币值换算 EJB。

1. 定义 Remote 接口

Remote 接口包含 EJB 构件实现的商业方法的声明，客户端只能通过 remote 接口访问构件实现的商业方法，不能直接调用。程序 9-1 给出了币值换算 EJB 的 Remote 接口定义：

程序 9-1　币值换算 EJB 的 Remote 接口定义

```
package FlexConverter;
import javax.ejb.*;
import java.rmi.*;
public interface Converter extends EJBObject
{
    public double dollarToYen(double dollars) throws RemoteException;
    public double yenToEuro(double yen) throws RemoteException;
}
```

在程序 9-1 中，我们定义了一个名为 Converter 的 Remote 接口，币值换算 EJB 构件向客户端提供两个币值换算方法——dollarToYen 完成美元到日元的换算，yenToEuro 完成日元到欧元的换算，Remote 接口中包含这两个方法的声明。可以看到，按照 EJB 规范的约定，接口 Converter 继承了接口 EJBObject，两个操作抛出 RemoteException 异常以报告远程调用错误，参数与返回值均为合法的 Java RMI 类型。

2. 定义 Home 接口

Home 接口中包含 EJB 构件生命周期管理的相关方法，客户程序使用 Home Interface 创建、查找或删除 EJB 的实例。程序 9-2 给出了币值换算 EJB 的 Home 接口定义：

程序 9-2　币值换算 EJB 的 Home 接口定义

```
package FlexConverter;

import java.rmi.*;
import javax.ejb.*;

public interface ConverterHome extends EJBHome
{
    public Converter create() throws RemoteException, CreateException;
}
```

在程序 9-2 中，我们定义了一个名为 ConverterHome 的 Home 接口，按照 EJB 规范的约

定，接口 ConverterHome 继承了接口 EJBHome。该接口中仅声明了一个 create 方法，其返回值必须为程序 9-1 中定义的 Remote 接口 Converter。按照 EJB 规范的约定，create 方法抛出 RemoteException 和 CreateException 异常。

3. 定义 Enterprise Bean 类

币值换算 EJB 的 Enterprise Bean 类首先要按照 Remote 接口的约定实现商业方法 dollarToYen 与 yenToEuro，其次要实现 Home 接口中 create 方法对应的 ejbCreate 方法与会话构件生命周期相关的方法。程序 9-3 给出了币值换算 EJB 的 Enterprise Bean 类定义：

程序 9-3　币值换算 EJB 的 Enterprise Bean 类定义

```
package FlexConverter;

import javax.ejb.*;
import javax.naming.*;

public class ConverterBean implements SessionBean
{
    SessionContext ctx;
    public double dollarToYen(double dollars){
        Context initCtx;
        Double dollarsToYen = null;
        try{

            initCtx = new InitialContext();
            dollarsToYen =
                 (Double)initCtx.lookup("java:comp/env/dollarsToYen");
        }catch(NamingException ne){
            ne.printStackTrace();
        }
        return dollarsToYen.doubleValue() * dollars;
    }

    public double yenToEuro(double yen){
        Context initCtx;
        Double yenToEuro = null;
        try{
            initCtx = new InitialContext();
            yenToEuro =
                 (Double)initCtx.lookup("java:comp/env/ yenToEuro");
        }catch(NamingException ne){
            ne.printStackTrace();
        }
        return yenToEuro.doubleValue() * yen;
    }

    public ConverterBean() {}
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void setSessionContext(SessionContext Context) {
        ctx = Context;
    }
}
```

在程序 9-3 中，我们定义了一个名为 ConverterBean 的 Enterprise Bean 类，由于该 EJB 构件是会话构件，因此 Enterprise Bean 类实现（implements）SessionBean 接口。

Enterprise Bean 类首先按照 Remote 接口 Converter 的约定实现了商业方法 dollarToYen 与 yenToEuro，在进行币值换算时，首先利用 JDNI 服务查找表示当前汇率的环境条目，然后根据环境条目中存储的汇率值完成币值换算。

其次，Enterprise Bean 类实现了 Home 接口 ConverterHome 中的 create 方法对应的 ejbCreate 方法和会话构件生命周期相关的其它方法。

源代码编写完成后，用 Java 语言编译器 javac 对 EJB 源代码进行编译。

4. 打包/布署 EJB

本例中打包与布署 EJB 构件的过程与 8.2 节中打包/布署时间 EJB 类似，不同的地方主要有几下几点：

- 本例将包含 EJB 构件的 J2EE 应用命名为 ConverterApp。
- 在环境条目配置界面为该 EJB 构件配置两个分别表示当前美元到日元汇率与当前日元到欧元汇率的环境条目，如图 9-1 所示，具体的配置信息为：
  - Coded Entry：该环境条目在源代码中引用时所使用的 JNDI 名，该名字与程序 9-3 中查找对应环境条目所使用的 JNDI 名一致，分别为“dollarsToYen”与“yenToEuro”。
  - Type：环境条目的类型，本例中两个环境条目的类型均为双精度浮点类型（Double）。
  - Value：环境条目的值，分别输入表示当前美元到日元汇率与当前日元到欧元汇率的值，如 121 与 0.0077。

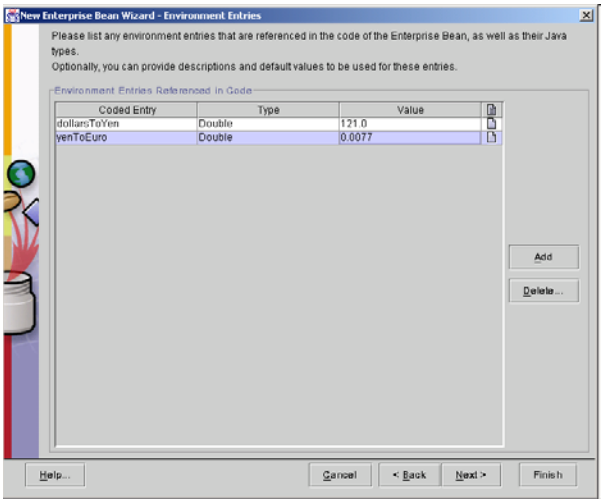


图 9-1 配置表示汇率的环境条目

- 为币值换算 EJB 配置名为“FlexibleConverterBean”的 JNDI 名。
- 布署时生成名为“ConverterAppClient.jar”的客户端 JAR 文件包，并将其存放到后面客户端程序所在的目录下。

9.1.2 开发客户端程序

1. 创建客户端程序

本例中我们编写一个名为 ConverterClient 的 Java 类作为客户端，该类包含一个入口函数 main。该客户端程序调用币值换算 EJB 的 dollarToYen 与 yenToEuro 方法，提供的参数均为 100，并将返回的换算结果输出到客户端。客户端程序的代码如程序 9-4 所示。

程序 9-4 币值换算 EJB 的客户端程序

```
import FlexConverter.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class ConverterClient{
    public static void main(String[] args){
        try{
            Context initial = new InitialContext();
```

```
            ConverterHome converterHome = (ConverterHome)
                    initial.lookup("FlexibleConverterBean");
            Converter converter = converterHome.create();
            double amount = converter.dollarToYen(100.00);
            System.out.println("$100 equals Yen " + String.valueOf(amount));
            amount = converter.yenToEuro(100.00);
            System.out.println("Yen 100 equals Euro " + String.valueOf(amount));
            converter.remove();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

在程序 9-4 所示的客户端程序中，首先利用 JNDI 服务查找币值换算 EJB 对应的 Home 接口，查找到 Home 接口后，客户端调用 Home 接口中的 create 操作来获取一个可用的 EJB，该操作返回一个 Remote 接口，客户端利用返回的 Remote 接口调用 EJB 上的操作 dollarToYen 与 yenToEuro。使用完 EJB 后，客户端程序调用 Remote 接口中的 remove 操作通知服务端其不再使用该 EJB。

2. 编译/运行客户端程序

本例中可使用以下命令完成对客户端程序的编译：

prompt> javac -classpath "ConverterAppClient.jar;%CLASSPATH%" ConverterClient.java

本例中可使用以下命令执行客户端程序：

prompt> java -classpath " ConverterAppClient.jar;%CLASSPATH%" ConverterClient

运行客户端程序时客户端输出返回的币值换算结果，例如：

```
$100 equals Yen 12100.0

Yen 100 equals Euro 0.77
```

客户端程序调用 dollarToYen 与 yenToEuro 方法时传入的参数均为 100，而打包时环境条目中配置的相应汇率分别为 121 与 0.0077，因此我们得到了上面的输出结果。

当相应的汇率发生变化时，我们可以仅修改 J2EE 应用中对应的环境条目值，而不需修改 EJB 构件的源代码，就可以使得 EJB 构件按照新的汇率完成币值换算。如图 9-2 所示，在 deploytool 中，选中打包好的 EJB 构件 ConverterBean，点击右侧的“Env. Entries”选项卡，可以修改汇率对应的环境条目值，如我们可以将汇率分别修改为 200 与 0.006。

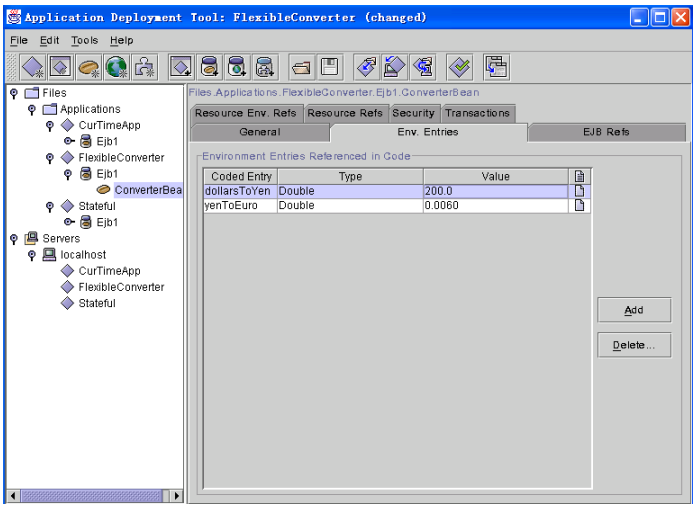


图 9-2 配置表示汇率的环境条目

修改完环境条目后，将本例的 J2EE 应用重新发布，再次执行客户端程序，我们可以得

到下面的输出，可以看出得到的币值换算结果使用了新的汇率值。

```
$100 equals Yen 20000.0
Yen 100 equals Euro 0.6
```

§ 9.2 事务控制

J2EE 平台为 EJB 构件提供 CMT（Container Managed Transaction，容器维护的事务）与 BMT 两种事务控制方式。CMT 方式下，事务由容器自动控制，开发人员只需在打包 EJB 构件时为 EJB 上的操作选择合适的事务属性即可。BMT 方式下，开发人员在 EJB 的源代码中利用 JTA 来编程控制事务。

本节通过两个例子分别演示 CMT 方式与 BMT 方式的事务控制。

9.2.1 基于 CMT 的事务控制

本小节通过一个简单的例子演示如何基于 CMT 方式实现事务控制。

CMT 方式下，程序员在 EJB 构件的源程序中没有事务边界控制的代码（如事务开始、回滚、提交等），而是在部署描述符中指定事务属性，由容器按照属性对应的规则来控制事务的边界。可以选择的事务属性及对应的事务控制规则如表 9-1 所示：

表 9-1 事务属性与事务控制规则

事务属性	客户端事务控制	EJB 方法对应的事务控制
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Support	None	None
	T1	T1
NotSupport	None	None
	T1	None
Mandatory	None	ERROR
	T1	T1
Never	None	None
	T1	ERROR

在表 9-1 中，第一列表示 EJB 方法的事务属性取值，打包 EJB 构件时可以为 EJB 的商业方法选择 Required、RequiresNew 等六种不同的事务属性；第二列表示调用该方法的客户端的事务控制情况，其中 None 表示客户端没有事务控制，T1 表示客户端有事务控制；第三列表示该方法的事务控制规则，其中 T2 表示容器为该方法的执行维护一个对应的事务，T1 表示该方法在客户端维护的事务中执行，None 表示该方法在无事务控制的环境下执行，ERROR 表示出错情况。

以表 9-1 中的前两行为例，第一行表示“如果某方法的事务属性设为 Required（第一列为 Required），并且客户端没有事务控制（第二列为 None），则容器会为该方法的执行维护一个对应的事务（第三列为 T2），容器维护的事务可保证该方法中涉及的所有操作构成一个完整事务（执行过程满足事务特性，如事务的原子性可保证所有操作要么全部执行成功，要么一个都不会执行）”。第二行表示“如果某方法的事务属性设为 Required（第一列为 Required），并且客户端有事务控制（第二列为 T1），则该方法在客户端维护的事务中执行（第三列为 T1）”。

读者应注意容器维护的事务是方法级的，容器即默认将一个方法当作一个事务执行；并且只有当方法执行的过程中发生了**系统级**异常，容器才会自动将事务回滚，即将方法前面执行的结果恢复，如果发生的异常是用户自定义异常，则容器会认为事务的执行是成功的，不

会将事务回滚。

下面我们将开发一个名为 CMT 的 J2EE 应用，该应用中包含一个无状态会话构件，该 EJB 构件实现银行账户上的存款、取款与余额查询等操作。由于该构件的每次操作均需调用者提供账户名参数，因此其实例（对象）不需要保存与特定客户端相关的会话状态，因此可设计为无状态的会话构件。以下简称该 EJB 构件为账户 EJB。

1. 定义 Remote 接口

Remote 接口包含 EJB 构件实现的商业方法的声明，客户端只能通过 remote 接口访问构件实现的商业方法，不能直接调用。程序 9-5 给出了账户 EJB 的 Remote 接口定义：

程序 9-5　账户 EJB 的 Remote 接口定义

```
package bank;
import javax.ejb.*;
import java.rmi.*;
public interface Banker extends EJBObject
{
    public void deposit(String accountName, int amount)
                                throws RemoteException, BankerFailureException;
    public void withdraw(String accountName, int amount)
                                throws RemoteException, BankerFailureException;
    public int getBalance(String accountName)
                                throws RemoteException, BankerFailureException;
}
```

在程序 9-5 中，我们定义了一个名为 Banker 的 Remote 接口，账户 EJB 构件向客户端提供三个账户操作——deposit 完成存款功能，withdraw 完成取款功能，getBalance 完成余额查询功能，Remote 接口中包含这三个方法的声明。可以看到，按照 EJB 规范的约定，接口 Banker 继承了接口 EJBObject，三个操作均抛出 RemoteException 异常以报告远程调用错误，参数与返回值均为合法的 Java RMI 类型；另外这三个操作还抛出 BankerFailureExpception 以的报告操作失败。

2. 定义 Home 接口

Home 接口中包含 EJB 构件生命周期管理的相关方法，客户程序使用 Home Interface 创建、查找或删除 EJB 的实例。程序 9-6 给出了账户 EJB 的 Home 接口定义：

程序 9-6　账户 EJB 的 Home 接口定义

```
package bank;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface BankerHome extends EJBHome
{
    public Banker create() throws RemoteException, CreateException;
}
```

在程序 9-6 中，我们定义了一个名为 BankerHome 的 Home 接口，按照 EJB 规范的约定，接口 BankerHome 继承了接口 EJBHome。该接口中仅声明了一个 create 方法，其返回值为程序 9-5 中定义的 Remote 接口 Banker。按照 EJB 规范的约定，create 方法抛出 RemoteException 和 CreateException 异常。

3. 定义 Enterprise Bean 类

账户 EJB 的 Enterprise Bean 类首先要按照 Remote 接口的约定实现商业方法 deposit、withdraw 与 getBalance，其次要实现 Home 接口中 create 方法对应的 ejbCreate 方法与会话构件生命周期相关的方法。程序 9-7 给出了账户 EJB 的 Enterprise Bean 类定义：

程序 9-7　账户 EJB 的 Enterprise Bean 类定义

```
package bank;
import javax.ejb.*;
import javax.naming.*;
import java.sql.*;
import javax.sql.*;
import java.util.Random;
public class BankerBean implements SessionBean
{
    DataSource ds;
    Connection conn;
    public void deposit(String accountName, int amount)
                        throws BankerFailureException{
        try{
            conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet res = stmt.executeQuery("SELECT * FROM accounts WHERE
                                accountname = '" + accountName + "'");
            if(res.next()){
                int newBalance;
                newBalance = res.getInt("balance") + amount;
                stmt.execute("UPDATE accounts SET balance = " + newBalance +
                            " WHERE accountname = '" + accountName + "'");
            }else{
                throw new BankerFailureException("invalid accountName");
            }
            conn.close();
        }catch(Exception e){
                throw new BankerFailureException("invalid accountName");
        }
    }
    public void withdraw(String accountName, int amount)
                         throws BankerFailureException{
        try{
            System.out.println("\nEntry withdraw");
            conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet res = stmt.executeQuery("SELECT * FROM accounts WHERE
                                accountname = '" + accountName + "'");
            //从账户上减去相应的金额
            int newBalance;
            if(res.next()){
                if(amount > res.getInt("balance")){
                    throw new BankerFailureException("no enough balance");
                }
                newBalance = res.getInt("balance") - amount;
                stmt.execute("UPDATE accounts SET balance = " + newBalance +
                            " WHERE accountname = '" + accountName + "'");
            }else{
                throw new BankerFailureException("invalid accountName");
            }
            conn.close();
            System.out.println(accountName + "'s balance changed to " +
                                                                newBalance);
            System.out.println("pushing cash...");
            //操纵取款机为用户吐出现金
            pushCash(amount);
            System.out.println("withdraw finished successfully");
        }catch(SQLException e){
            throw new BankerFailureException("operation failed");
        }
    }
```

```
    private void pushCash(int amount){
        Random rand = new Random();
        int i =Math.abs(rand.nextInt());
        if(i > 1000000000){
            System.out.println("pushCash failed(" + i +")");
            throw new RuntimeException();
        }
    }

    public int getBalance(String accountName) throws BankerFailureException{
        try{
            conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet res = stmt.executeQuery("SELECT * FROM accounts WHERE
                                    accountname = '" + accountName + "'");
            int curBalance;
            if(res.next()){
                curBalance = res.getInt("balance");
                conn.close();
                return curBalance;
            }else{
                throw new BankerFailureException("invalid accountName");
            }
        }catch(Exception e){
            throw new BankerFailureException("operation failed");
        }
    }

    public void ejbCreate() throws CreateException{
        try{
            InitialContext initialCtx = new InitialContext();
            ds = (DataSource)
                    initialCtx.lookup("java:comp/env/jdbc/Cloudscape");
        }catch(NamingException ex){
            throw new CreateException("lookup datasource failed");
        }catch(Exception e){
            throw new CreateException("operation failed");
        }

    }
    public void ejbRemove() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void setSessionContext(SessionContext Context) {}
}
```

在程序 9-7 中，我们定义了一个名为 BankerBean 的 Enterprise Bean 类，由于该 EJB 构件是会话构件，因此 Enterprise Bean 类实现（implements）SessionBean 接口。

Enterprise Bean 类首先按照 Remote 接口 Banker 的约定实现了商业方法 deposit、withdraw 与 getBalance。其次，Enterprise Bean 类实现了 Home 接口 BankerHome 中的 create 方法对应的 ejbCreate 方法和会话构件生命周期相关的其它方法。

为简化应用的结构，在实现这些商业方法时，账户 EJB 并没有利用实体构件访问数据库，而是利用数据源，通过编写代码完成直接数据库的访问。在类 BankerBean 的 ejbCreate 方法中，我们利用 JNDI 服务找到需要使用的数据源，并将其引用保存在数据成员 ds 中，由于 ejbCreate 方法只会在容器创建无状态会话构件时调用一次，因此数据源的查找操作对于每个账户 EJB 的对象来说只会执行一次，与在每次需要使用数据源时均首先查找的方法相比，本例采用的方法是一种具有更好执行效率的常用方法。

实现 deposit 方法时首先利用保存的数据源引用获取一个可用的数据库连接，然后利用该连接执行 SELECT 语句定位到参数 accountName 所标识的记录，对于 deposit 与 withdraw

方法，将记录的余额进行相应的修改。

withdraw 方法本例中重点关注的方法为演示事务控制，我们为 withdraw 方法增加一步吐出现金的操作，读者可以把本例中的取款操作想像成在自动取款机上执行取款操作，该操作逻辑上分为修改账户余额与吐出现金两个主要步骤。从业务逻辑上讲，这两个步骤是一个原子的整体，即这两个步骤要么应全部执行成功，要么一步都不能执行，这种特性可以通过事务控制实现。实现余额修改的流程与实现 deposit 方法基本类似，时首先利用保存的数据源引用获取一个可用的数据库连接，然后利用该连接执行 SELECT 语句定位到参数 accountName 所标识的记录，将记录的余额进行相应的修改。我们定义了一个辅助方法 pushCash 来模拟吐出现金的过程，在该方法实现中，我们仅通过产生一个随机数的来模拟吐出现金操作以某个概率发生错误，如果产生的随机数大于 1000000000，方法 pushCash 就会产生一个系统级异常——RuntimeException。请注意 pushCash 执行失败时我们并没有编写代码处理 pushCash 之前的余额修改，我们将通过事务控制来保证系统的正确性，我们在 withdraw 方法中添加了四条输出语句（程序中 9-7 加粗的 System.out.println 语句）以跟踪 withdraw 方法的执行情况：

- 刚进入 withdraw 方法，会输出一条信息“**Entry withdraw**”；
- 余额修改完成，会输出一条信息“xxx'**s balance changed to nnn**”；
- 执行 pushCash 之前，会输出一条信息“**pushing cash...**”；
- pushCash 执行成功，会输出一条信息“**withdraw finished successfully**”。

如果 pushCash 执行失败，则不会输出信息“**withdraw finished successfully**”，而此时 pushCash 方法中会输出一条信息“**pushCash failed(nnnnnn)**”。

getBalance 方法的实现过程与 deposit 方法类似，只是不需修改对应的余额。

此外，账户 EJB 还使用了一个辅助的异常类 BankerFailureException，程序 9-8 给出了 BankerFailureException 类的定义：

程序 9-8　BankerFailureException 的类定义

```
package bank;
public class BankerFailureException extends Exception{
    public BankerFailureException() {}
    public BankerFailureException(String msg){
        super(msg);
    }
}
```

源代码编写完成后，用 Java 语言编译器 javac 对 EJB 源代码进行编译。

4. 打包/布署 EJB

本例中打包与布署 EJB 构件的过程与 8.2 节中打包/布署时间 EJB 类似，不同的地方主要有几下几点：

- 本例将包含 EJB 构件的 J2EE 应用命名为 CMT。
- 本例选择 EJB 的事务控制方式为 CMT，如图 9-3 所示，选择事务控制方式为“Container Managed”，并为 EJB 的每个商业方法选择一个事务属性值，本例中我们可以均使用缺省的事务属性值“Required”，我们重点关注 EJB 的 withdraw 方法，由于本例的客户端没有事务控制，因此客户端调用 withdraw 方法时，容器会为 withdraw 方法的执行维护一个事务，该事务可保证 withdraw 方法中的所有操作（基本分为修改余额与吐出现金两个步骤）要么全部执行成功，要么一个都不会执行。

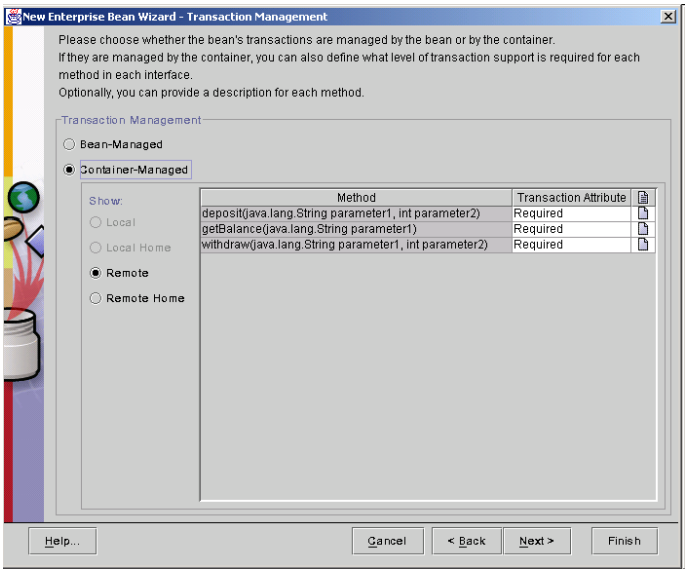


图 9-3 配置事务控制方式为 CMT

- 本例需要在图 9-4 所示的资源工厂配置界面中配置程序 9-7 中引用的数据源 BankDB，具体的配置信息为：
  - Coded Name：代码中引用资源工厂时使用的名字，本例中为“jdbc/BankDB”；
  - Type：资源工厂的类型，本例中为数据源（javax.sql.DataSource）；
  - JDNI Name：引用的资源工厂在 J2EE 平台上对应的实际资源的 JNDI 名，本例中我们仍将账户信息表存储在 cloudscape 缺省数据库 CloudscapeDB 中，因此此处填写该数据库对应的数据源的 JNDI 名“jdbc/Cloudscape”。

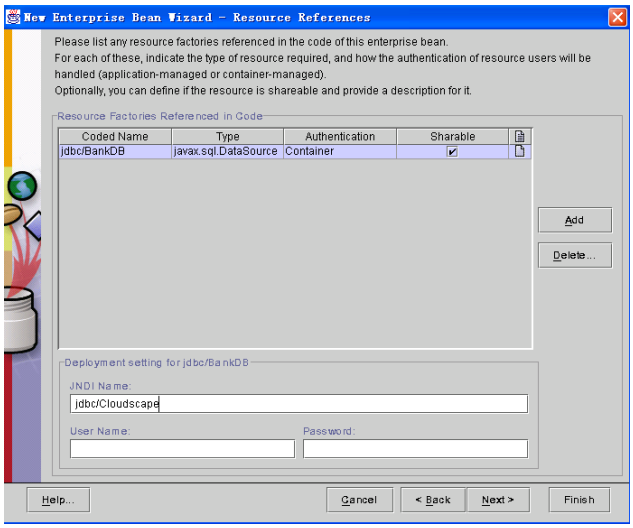


图 9-4 配置数据源

- 为币值换算 EJB 配置名为“Banker”的 JNDI 名。
- 布署时生成名为“BankAppClient.jar”的客户端 JAR 文件包，并将其存放到后面客户端程序所在的目录下。

另外在本例中，我们需要在数据库中创建账户 EJB 使用的数据库表，并在其中插入一些测试数据。我们可以利用“cloudscape –isql”命令进入 cloudscape 的交互式 SQL，通过执行以下 SQL 语句在数据库中创建账户 EJB 使用的账户信息表：

```
ij> CREATE TABLE accounts (accountName CHAR(20), balance INTEGER);
```

然后执行以下 SQL 语句在刚刚创建的表中插入一条记录，账户名为 name1，余额为 1000：

ij> INSERT INTO accounts VALUES('name1', 1000);

5. 开发/运行客户端程序

本例中我们编写一个名为 BankerClient 的 Java 类作为客户端，该类包含一个入口函数 main。该客户端程序调用利用账户 EJB 的 withdraw 操作，取款 100 元。客户端程序的代码如程序 9-9 所示。

程序 9-9　账户 EJB 的客户端程序

```
import bank.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class BankerClient{
    public static void main(String[] args){
        try{
            Context initial = new InitialContext();
            BankerHome bankerHome = (BankerHome)PortableRemoteObject.narrow(
                initial.lookup("Banker"), BankerHome.class);
            Banker banker = bankerHome.create();
            System.out.println("name1's balance is: " +
                                        banker.getBalance("name1"));
            banker.withdraw("name1", 100);
            System.out.println("name1's balance is: " +
                                        banker.getBalance("name1"));
        }catch(RemoteException re){
            System.out.println("RemoteException : " + re.getMessage());
        }catch(BankerFailureException be){
            System.out.println("BankerFailureException : " + be.getMessage());
        }catch(CreateException ce){
            System.out.println("CreateException : " + ce.getMessage());
        }catch(Exception e){
            System.out.println("Get Exception: " + e.getMessage());
        }

        try{
            Context initial = new InitialContext();
            BankerHome bankerHome = (BankerHome)PortableRemoteObject.narrow(
                initial.lookup("Banker"), BankerHome.class);
            Banker banker = bankerHome.create();
            System.out.println("name1's balance is: " +
                                        banker.getBalance("name1"));
        }catch(RemoteException re){
            System.out.println("RemoteException : " + re.getMessage());
        }catch(BankerFailureException be){
            System.out.println("BankerFailureException : " + be.getMessage());
        }catch(CreateException ce){
            System.out.println("CreateException : " + ce.getMessage());
        }catch(Exception e){
            System.out.println("Get Exception: " + e.getMessage());
        }
    }
}
```

在程序 9-9 所示的客户端程序中，首先利用 JNDI 服务查找账户 EJB 对应的 Home 接口，查找到 Home 接口后，客户端调用 Home 接口中的 create 操作来获取一个可用的 EJB，该操作返回一个 Remote 接口，客户端利用返回的 Remote 接口调用 EJB 上的操作。

在客户端程序中，三次调用 getBalance 操作并将返回的余额信息输出（程序中 9-9 加粗的 System.out.println 语句）以跟踪账户的变化情况，第一次是在调用 withdraw 方法取款 100

元之前，第二次是在调用 withdraw 成功完成后，第三次是在客户端程序退出之前。由于 withdraw 方法的执行可能出错（pushCash 方法可能抛出 RuntimeException），因此第二次的输出语句可能不执行。

运行客户端程序时，如果服务端 withdraw 方法的执行没有错误，我们可以看到账户 name1 中余额减少了 100 元，对应的客户端与服务端输出信息如下：

客户端输出:

```
prompt> java -classpath " BankAppClient.jar;%CLASSPATH%" BankerClient
name1's balance is: 1000
name1's balance is: 900
name1's balance is: 900
```

服务端输出:

```
Entry withdraw
name1's balance changed to 900
pushing cash...
withdraw finished successfully
```

如果运行客户端程序时服务端 withdraw 方法的执行出现错误，我们可以看到账户 name1 中余额并没有减少，尽管在服务端输出的信息中我们看到余额已经修改，但是容器维护的事务保证了吐出现金操作失败时余额的修改是无效的，此时对应的客户端与服务端输出信息如下：

客户端输出:

```
prompt> java -classpath " BankAppClient.jar;%CLASSPATH%" BankerClient
name1's balance is: 900
RemoteException : RemoteException occurred in server thread; nested
exception is:
       java.rmi.RemoteException:       nested       exception       is:
java.lang.RuntimeException; nested exception is:
       java.lang.RuntimeException
name1's balance is: 900
```

服务端输出:

```
Entry withdraw
name1's balance changed to 800
pushing cash...
pushCash failed(1302702409)
```

读者可的 cloudscape 的交互式 SQL 中通过执行以下 SQL 语句确认数据库中的数据。

读者应注意 withdraw 方法执行错误时抛出的异常是 RuntimeException，属于系统级异常，因此容器会自动将 withdraw 方法对应的事务回滚，如果执行过程中抛出的异常不是系统（如将程序 9-7 中 RuntimeException 改为 BankFailureException），则容器会认为事务的执行是成功的，从而将事务提交。有兴趣的读者可以修改程序 9-7 并进行相关测试。

9.2.2 基于 BMT 的事务控制

BMT 方式下，程序员在 EJB 的源程序中控制事务边界控制（如事务开始、回滚、提交等），并在部署描述符中指定由 Bean 控制事务的边界。与 CMT 方式相比，Bean 维护的事务可以跨越方法的边界，因此通常具有更好的灵活性。

本小节通过修改上一小节的例子来演示如何基于 BMT 方式实现事务控制。

我们通过修改程序 9-7 中的 withdraw 方法，在其中加入事务边界控制的代码，同时修改 pushCash 方法使其抛出用户自定义异常 BankerFailureException，其它的实现保持不变。程序 9-10 给出了修改后的 withdraw 方法与 pushCash 方法。

程序 9-10　修改后的 withdraw 方法与 pushCash 方法

```
//其它实现代码
public void withdraw(String accountName, int amount)
                                throws BankerFailureException{
    System.out.println("Entry withdraw");
    try{
        System.out.println("Getting transaction object...");
        UserTransaction userTrx = sessionCtx.getUserTransaction();
        userTrx.begin();
        System.out.println("user transaction begin...");
    }catch(Exception exc){
        throw new BankerFailureException("transaction not begin");
    }

    try{
        conn = ds.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet res = stmt.executeQuery("SELECT * FROM accounts WHERE
                            accountname = '" + accountName + "'");
        //从账户上减去相应的金额
        int newBalance;
        if(res.next()){
            if(amount > res.getInt("balance")){
                throw new BankerFailureException("no enough balance");
            }
            newBalance = res.getInt("balance") - amount;
            stmt.execute("UPDATE accounts SET balance = " + newBalance);
        }
        else{
            throw new BankerFailureException("invalid accountName");
        }
        conn.close();
        System.out.println(accountName + "'s balance changed to " +
                                                        newBalance);
        System.out.println("pushing cash...");
        //操纵取款机为用户吐出现金
        pushCash(amount);
        System.out.println("withdraw finished successfully");
        try{
            UserTransaction userTrx = sessionCtx.getUserTransaction();
            userTrx.commit();
            System.out.println("user transaction committed");
        }catch(Exception exc){
            throw new BankerFailureException("transaction commit failed");
        }
    }catch(Exception e){
        System.out.println("Exception caught, try rollback");
        try{
            UserTransaction userTrx = sessionCtx.getUserTransaction();
            userTrx.rollback();
            System.out.println("user transaction rollbacked");
```

```
            }catch(Exception exc){
                throw new BankerFailureException("transaction rollback failed");
            }
            throw new BankerFailureException("operation failed");
        }
    }
    private void pushCash(int amount) throws BankerFailureException{
        Random rand = new Random();
        int i =Math.abs(rand.nextInt());
        if(i > 1000000000){
            System.out.println("pushCash failed(" + i +")");
            throw new BankerFailureException();
        }
    }
    //其它实现代码
```

在程序 9-10 给出的代码中，withdraw 方法共增加了三处事务边界控制代码：

- 刚刚进入 withdraw 方法，利用 JTA 中的 begin 方法控制事务开始。
- 吐出现金操作成功完成后，利用 JTA 中的 commit 方法控制事务提交。
- 捕获到异常后，利用 JTA 中的 rollback 方法控制事务回滚。

修改完毕后，将 EJB 重新编译，我们可以重新创建一个 J2EE 应用并将修改后的 EJB 构件重新打包，打包过程与上一小节的过程类似，主要区别就是在图 9-5 所示的事务控制界面选择由 Bean 而不是容器来控制事务。

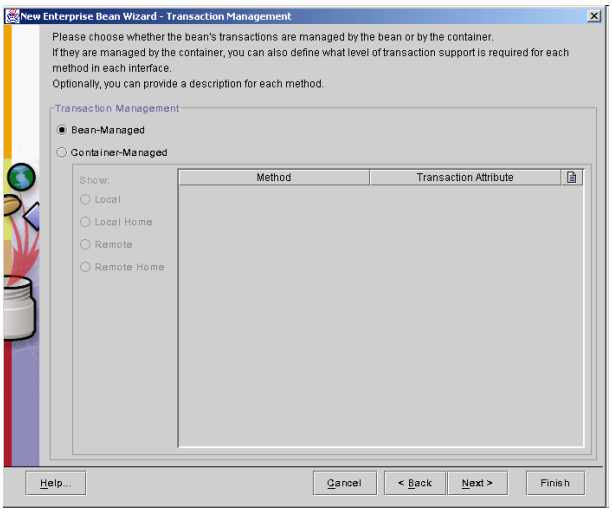


图 9-5 设置事务控制方式为 BMT

运行上一小节同样的客户端程序，我们可以看到当 pushCash 操作执行发生异常时，由于引发的异常被 withdraw 方法中的 catch 语句捕获到，进而调用了 JTA 的 rollback 操作控制事务回滚，从而使数据库中的余额恢复到了 withdraw 操作执行前的状态。一种可能的客户端与服务端输出如下：

客户端输出:

```
prompt> java -classpath " BankAppClient.jar;%CLASSPATH%" BankerClient
name1's balance is: 900
BankerFailureException : operation failed
name1's balance is: 900
```

```
服务端输出:
Entry withdraw
Getting transaction object...
user transaction begin...
name1's balance changed to 800
pushing cash...
pushCash failed(1754765104)
Exception caught, try rollback
user transaction rollbacked
```

读者应注意使用 BMT 控制事务时，应保证程序中的所有控制流出口均将事务结束（提交或回滚），因为如果事务未成功结束往往导致数据库中的数据被加锁，从而使得应用无法继续访问被锁定的数据。

§9.3 安全性控制

J2EE 中的安全性控制分为认证与授权两个级别，实现安全性控制的常用方式是声明的方式。声明方式的安全性控制完全由容器自动控制，开发人员主要通过配置安全性角色与授权规则来定制安全性控制的规则。

典型的声明性安全性控制需要在在部署描述符中描述安全性控制：

- 在 J2EE 应用的部署描述符中定义应用所使用的安全性角色，角色为应用系统所指定的逻辑用户组，如经理角色、职员角色等；
- 在 Web 模块中描述认证方式：选择如何进行用户身份验证；
- 在 Web 模块中描述安全性规则（授权规则）：设置 Web 模块中的构件（如 JSP、Servlet、静态页面等）的访问权限；
- 在 EJB 模块中描述安全性规则(授权规则)：设置 EJB 模块中 EJB 构件的访问权限；
- 部署时将定义的安全性角色映射到实际的安全域中：将实际用户管理系统中的用户映射到角色。

本节通过一个相对完整的 J2EE 应用来演示声明性安全性控制的配置过程与运行结果，本节将给出该应用中所有构件的源代码，但本节的重点是打包过程中安全性配置相关的内容。本例中我们将开发一个简单的银行应用，客户通过 Web 访问 Web 模块中的 servlet 构件，该 servlet 构件调用后台的 EJB 构件提供的服务完成客户请求。我们将在 Web 端进行认证、和授权控制，在 EJB 端进行授权控制。以下简称本例中开发的应用为银行应用。

9.3.1 构建银行应用

1. 应用结构与关键代码

我们将开发一个名为 NewBankApp 的 J2EE 应用，该应用中包含以下模块：

- EJB 模块 BankBeanJAR，该模块中包含三个 EJB 构件：
  - AccountManager 构件：一个无状态会话构件，完成银行账户管理员的任务，提供开户、存款、取款、查询余额等操作；
  - AccountBean 构件：一个 EJB2.0 的 CMP 实体构件，对应数据库中的账户记录表，该表中记录了系统中已有的账户信息；
  - LogBean 构件：一个 EJB2.0 的 CMP 实体构件，对应数据库中的日志记录表，该表用于记录日志信息；
- Web 模块 BankWeb，该模块中包含一个 Servlet 构件：
  - BankServlet 构件：响应客户的 HTTP 请求，调用 AccountManage 的操作。

该应用中构件之间的交互关系如图 9-6 所示：

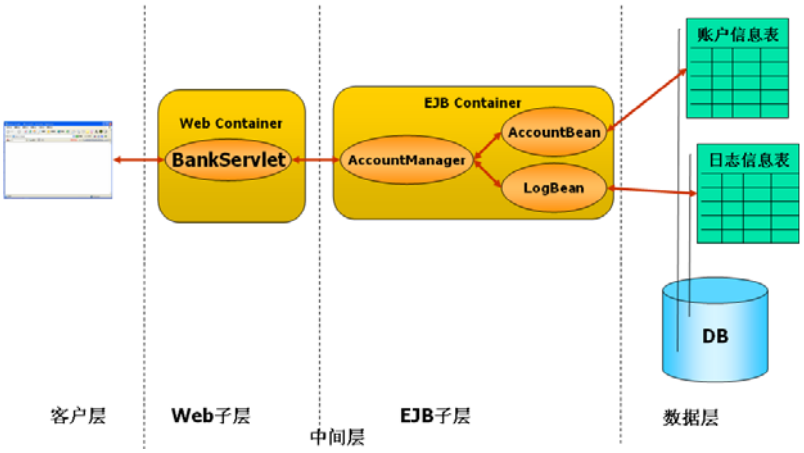


图 9-6 银行应用中构件之间的交互关系

运行时，Servlet 构件响应客户端的 HTTP 请求，并调用会话构件 AccountManager 完成客户端请求，会话构件 AccountManager 响应 Servlet 请求时会调用实体构件 AccountBean 完成对账户信息的操作，并调用实体构件 LogBean 记录系统日志。

在将应用打包后，我们将重点控制 BankServlet 构件与 AccountManager 构件的相关授权规则。该应用中各构件的源代码请参阅附录 2，下面对与安全性控制相关的关键源代码进行简要解释：

Servlet 构件 BankServlet 通过会话构件 AccountManager 的 Remote 接口调用账户管理员的相关操作，程序 9-11 给出了会话构件 AccountManager 的 Remote 接口：

程序 9-11　会话构件 AccountManager 的 Remote 接口定义

```
package banking;
import javax.ejb.*;
import java.rmi.*;
public interface AccountManager extends EJBObject
{
    public void deposit(String customerName, double amount)
                        throws RemoteException;
    public double withdraw(String customerName, double amount)
                        throws RemoteException, InsufficientFundsException;
    public double getBalance(String customerName) throws RemoteException;
    public double calculateInterest(String customerName) throws RemoteException;
    public Account createAccount(String customerName, double initialBalance)
                        throws NoAccountCreatedException, RemoteException;
    public void removeAccount(String customerName)
                        throws NoAccountCreatedException, RemoteException;
}
```

程序 9-11 中定义的接口 AccountManager 中声明了存款（deposit）、取款（withdraw）、查询余额（getBalance）、计算利息（calculateInterest）、开户（createAccount）与销户（removeAccount）六个操作。

Servlet 构件 BankServlet 响应客户端的 HTTP 请求，程序 9-12 给出了 Servlet 构件的类定义：

程序 9-12　Servlet 构件 BankServlet 的类定义

```
package bankWeb;

import banking.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class BankServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException{
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<HTML>");
        out.println("<HEAD><TITLE>BankServlet</TITLE></HEAD>");
        out.println("<BODY>");
        try{
            Context initial = new InitialContext();
            AccountManagerHome accountManagerHome = (AccountManagerHome)
                        initial.lookup("java:comp/env/ejb/AccountManager");
            AccountManager accountManager = accountManagerHome.create();
            out.println("get name1's balance...<br>");
            out.println("name1's balance is: " +
                            accountManager.getBalance("name1") + "<br>");
            Account account = accountManager.createAccount(
                            "newAccount", 1000.00f);
            out.println("createAccount newAccount successfully<br>");
            accountManager.removeAccount("newAccount");
            out.println("removeAccount successfully<br>");
        }catch(RemoteException re){
            out.println("RemoteException : " + re.getMessage());
        }catch(NoAccountCreatedException nace){
            out.println("NoAccountCreatedException : " + nace.getMessage());
        }catch(CreateException ce){
            out.println("CreateException : " + ce.getMessage());
        }catch(Exception e){
            out.println("Get Exception: " + e.getMessage());
        }
        out.println("</BODY>");
        out.println("</HTML>");
        out.close();
    }
}
```

程序 9-12 定义的 BankServlet 实现了一个 doGet 方法，当客户端通过 HTTP GET 方法访问 BankServlet 时，其 doGet 方法会被调用，doGet 方法为客户端动态生成一个 HTML 页面。首先利用 JNDI 服务查找会话构件 AccountManager 的 Home 接口，然后调用 Home 接口中的 create 方法获取一个可用的 EJB 对象——Remote 接口的引用，最终动态生成的页面中显示了调用 Remote 接口中操作的调用结果：

- 调用 getBalance 之前，在页面上打印信息“get name1’s balance…”
- 调用 getBalance 并在页面上打印信息“name1’s balance is *nnn*”
- 调用 createAccount 新开账户 newAccount，并在调用完成后在页面上打印信息“createAccount newAccount successfully”
- 调用 removeAccount 删除账户 newAccount，并在调用完成后在页面上打印信息“removeAccount successfully”

2. 打包测试应用

打包银行应用的过程分为以下几个主要步骤：

（1） 创建应用

创建名为 NewBankApp 的 J2EE 应用。

（2） 打包 EJB 构件

将会话构件 AccountManager、实体构件 AccountBean 与 LogBean 打包在同一个 EJB 模

块中，主要过程如下：

- 打包实体构件 AccountBean：
  - 创建一个名为 BankBeanJAR 的 EJB 模块，将应用中三个 EJB 构件对应的 Remote 接口、Home 接口与 Enterprise Bean 类以及相关的辅助异常类全部加入到该 EJB 模块中；
  - 设置该 EJB 为实体构件（Entity）；
  - 设置该 EJB 的持久性特性：类型为 EJB2.0 的 CMP 构件，两个数据成员 customerName 与 currentBalance 均与数据库中字段对应，主键类型为 java.lang.String，主键字段为 customerName；
  - 数据库设置：使用数据源“jdbc/Cloudscape”，生成缺省的 SQL 语句即可；
  - 配置名为“account”的 JNDI 名。
- 打包实体构件 LogBean：
  - 将该 EJB 添加到 EJB 模块 BankBeanJAR 中；
  - 设置该 EJB 为实体构件（Entity）；
  - 设置该 EJB 的持久性特性：类型为 EJB2.0 的 CMP 构件，数据成员 logMessage 与数据库中字段对应，主键类型为 java.lang.String，主键字段为 logMessage；
  - 数据库设置：使用数据源“jdbc/Cloudscape”，生成缺省的 SQL 语句即可；
  - 配置名为“log”的 JNDI 名。
- 打包会话构件 AccountManager：
  - 将该 EJB 添加到 EJB 模块 BankBeanJAR 中；
  - 设置该 EJB 为无状态会话构件；
  - 添加对实体构件 AccountBean 与 LogBean 的引用，正确填写对应的信息（填写信息的规则请参考 8.4.2 中订单 EJB 的打包）；
  - 配置名为“AccountManager”的 JNDI 名。

（3） 打包 Servlet 构件

将 Servlet 构件 BankServlet 打包在名为 BankWeb 的 Web 模块中，主要过程如下：

- 打包 Servlet 构件 BankServlet：
  - 如图 9-7 所示，点击 deploytool 的“File/New/Web Component”打开打包 Web 构件的向导。

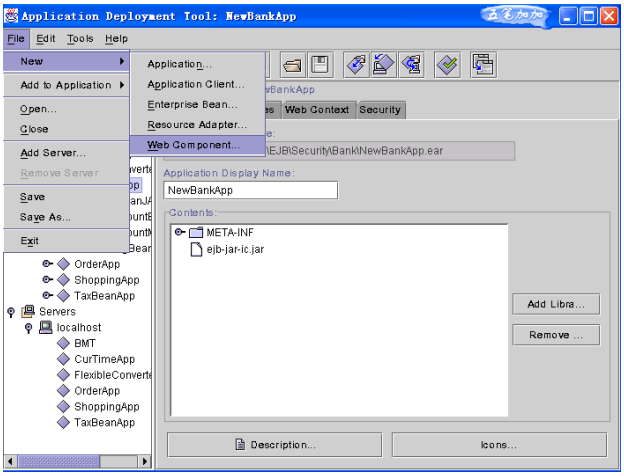


图 9-7 打开打包 Web 构件向导

  - 在图 9-8 所示的界面中选择创建一个名为 BankWeb 的 Web 模块，并将 BankServer 对应的类文件 BankServlet.class 加入到该模块中。

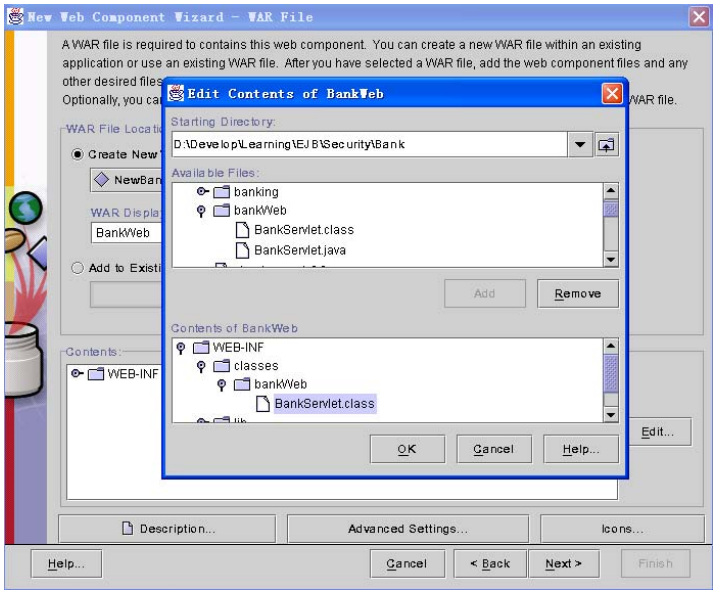


图 9-8 创建 Web 模块

■ 在图 9-9 所示的界面中选择打包的构件为 Servlet 构件：

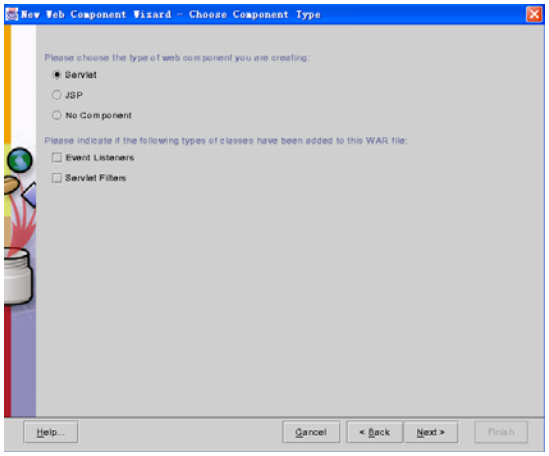


图 9-9 选择构件类型

■ 在图 9-10 所示的界面中选择 Servlet 构件的实现类为 bankWeb.BankServlet：

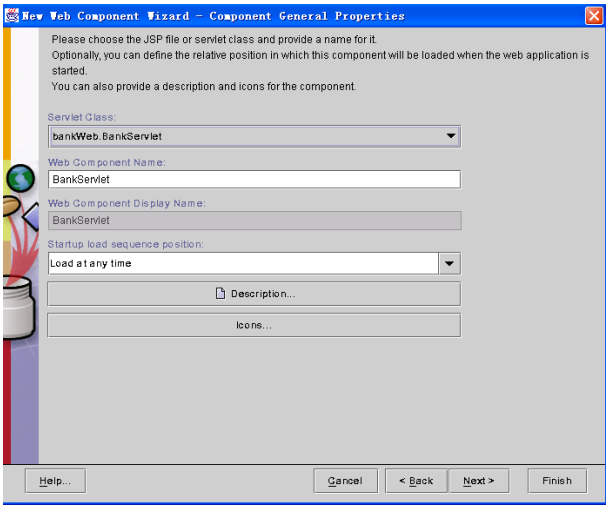


图 9-10 选择 Servlet 构件的实现类

- 在图 9-11 所示的界面中为该 Servlet 配置一个各为“BankServlet”的别名，别名是客户端以 HTTP 方式访问 Servlet 时在 URL 中使用的名字，打包时应至少为 Servlet 配置一个别名。

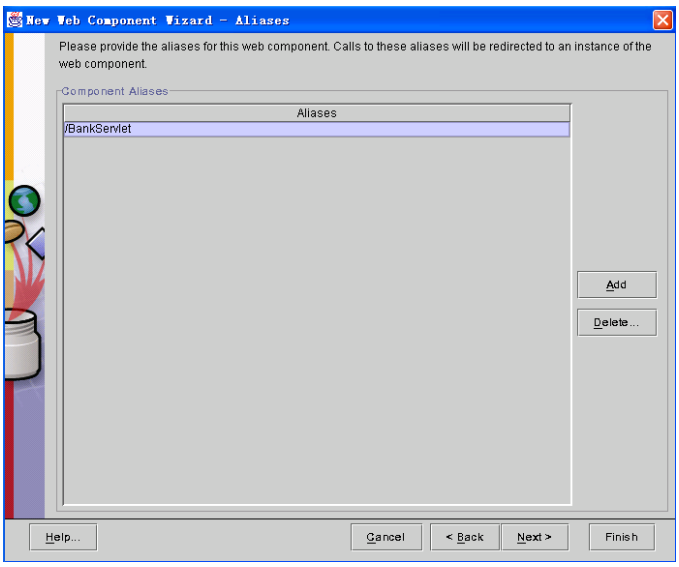


图 9-11 为 Servlet 配置别名

- 在图 9-12 所示的界面中添加对会话构件 AccountManager 的引用（EJB 引用的填写规则请参考 8.4.2 中订单 EJB 的打包）：

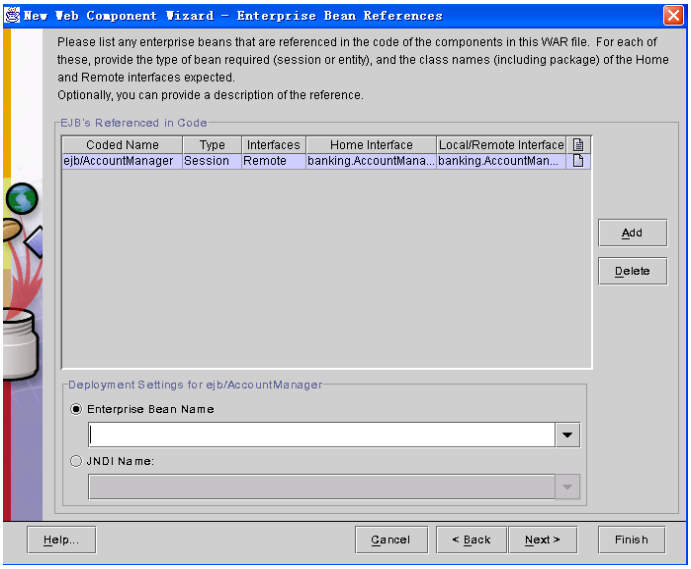


图 9-12 添加对会话构件 AccountManager 的引用

- 向导中其它界面均采用缺省设置即可。
- 打包 Web 构件的向导完成后，如图 9-13 所示，在 deploytool 中选中正在打包的 J2EE 应用，点击右侧的“Web Context”选项卡，为 Web 模块 BankWeb 配置一个名为“bankweb”的上下文根，上下文根为当前 Web 模块中所有构件提供一个上下文的根，在 URL 中上下文根通常放在构件别名的前面来共同标识对应构件。

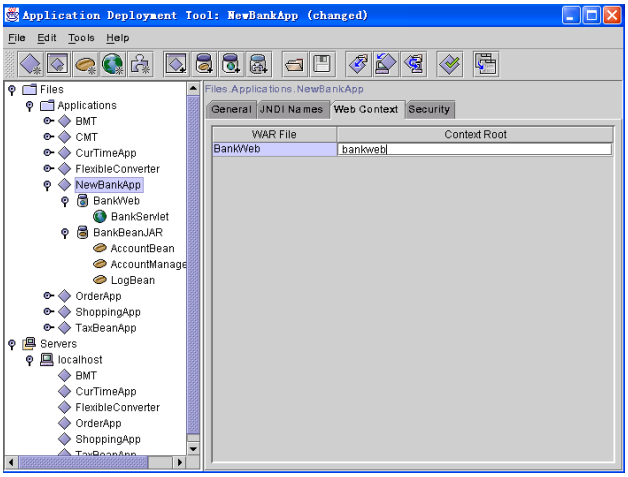


图 9-13 为 Web 模块配置上下文根

在上述的打包过程中，我们并没有对应用的安全性进行任何的配置，因此布署后该应用中的构件可以被无限制的访问。读者可将此状态下的应用进行布署，因为此应用的客户端通过 HTTP 访问应用，并不需要编写任何客户端程序，因此布署时不需生成客户端使用的 JAR 程序包。

布署完成后，我们需要在实体构件 AccountBean 对应的数据库表中插入一条记录来支持后续的测试，进入 cloudscape 的交互式 SQL，执行以下 SQL 语句可在实体构件 AccountBean 对应数据库表 AccountBeanTable 中插入测试需要的数据：

```
ij> INSERT INTO "AccountBeanTable" VALUES(1000,'name1');
```

打开浏览器，在地址栏中输入以下 URL 来访问银行应用中的 BankServlet：

http://localhost:8000/bankweb/BankServlet

该 URL 由三部分组成：

- 主机名＋端口号信息：“localhost:8000”，用于标识 J2EE 平台的 Web 服务器；
- 上下文根：“bankweb”，需要访问的构件所在 Web 模块的上下文根；
- 构件别名：“BankServlet”，构件的别名。

客户端访问 BankServlet 时，看到的页面中显示了该 Servlet 对会话构件 AccountManager 提供的商业方法的调用结果，所显示的页面如图 9-14 所示：

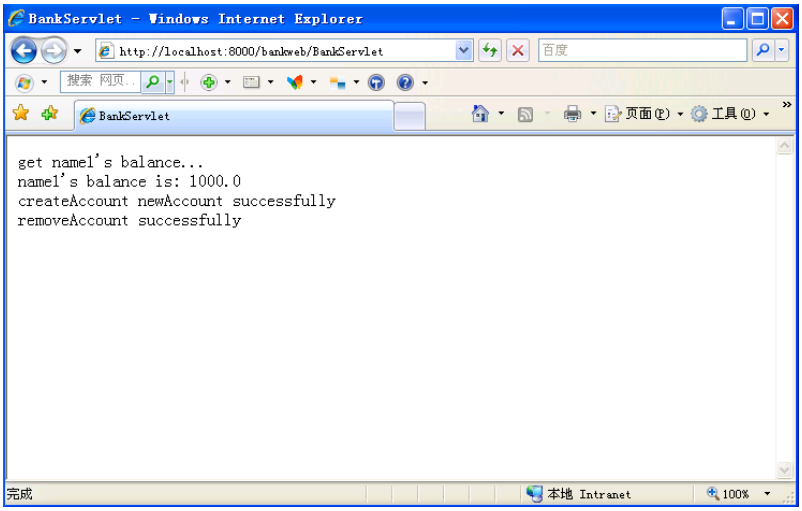


图 9-14 客户端输出

3. 为应用增加安全性配置

下面我们为银行应用添加安全性配置，我们将该应用的访问者均分为普通用户与超级用户两个角色，并配置这两个角色均可以访问应用的 BankServlet 构件，同时配置这两个角色均可以访问会话构件 AccountManager 的 getBalance 方法，而只有超级用户角色才能访问会话构件 AccountManager 的 createAccount 与 removeAccount 方法。配置的主要过程如下：

（1） 配置应用使用的安全性角色

如图 9-15 所示，在 deploytool 中选中刚刚打包的 J2EE 应用，点击右侧的“Security”选项卡，点击右下角的“Edit Roles”按钮，在图 9-16 所示的界面中为银行应用添加“NormalUser”与“Supervisor”两个安全性角色。

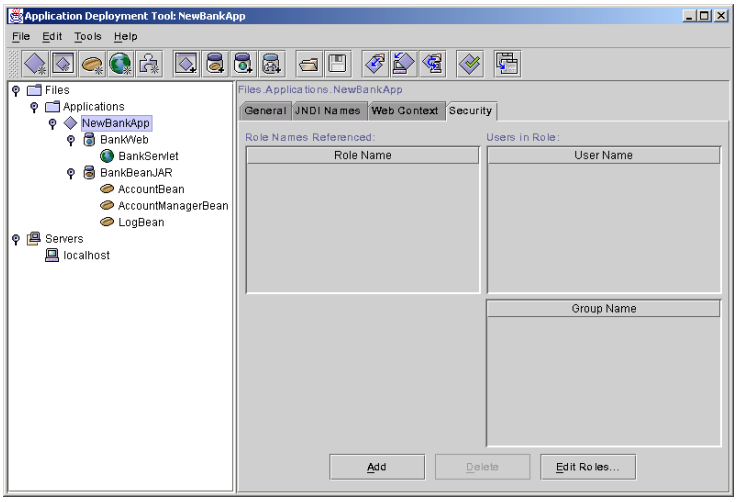


图 9-15 配置应用使用的安全性角色

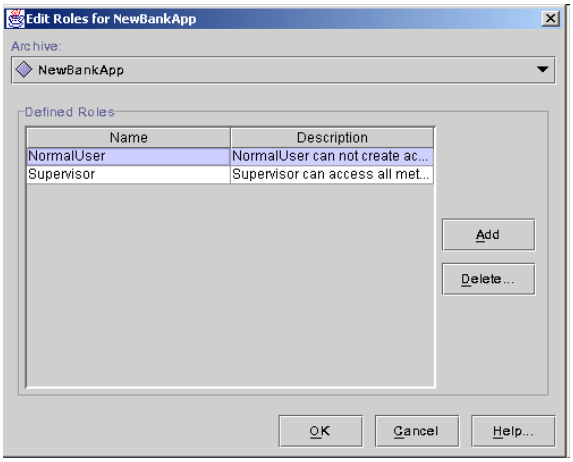


图 9-16 添加安全性角色

（2） 配置认证方式

如图 9-17 所示，在 deploytool 中选中银行应用中的 Web 模块 BankWeb，点击右侧的“Security”选项卡，在“User Authentication Method”下面的下拉列表框中选择基本的认证方式（Basic）。可以选择的许可证方式包括：

- Basic：基本认证方式。该方式下，用户第一次访问系统时，系统会弹出一个对话框，提示访问者输入用户名和密码。本例中采用该方式。
- Client Certificate：电子证书方式。该方式下，用户访问时向服务器提供一个表示身份的电子证书以进行身份验证。
- Form Based：表单方式。该方式下，由开发人员提供登录页面来验证用户身份。

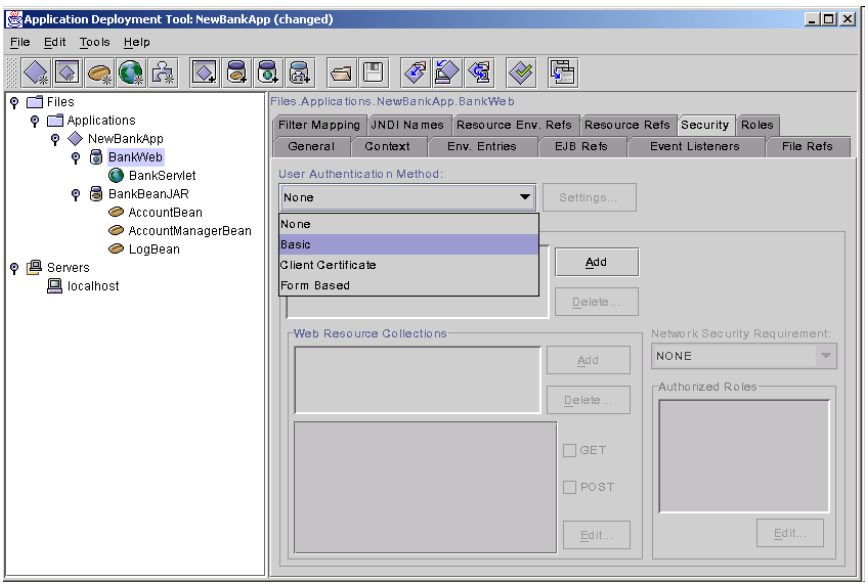


图 9-17 配置认证方式

（3） 配置 Web 模块的授权规则

如图 9-18 所示，在 deploytool 中选中银行应用中的 Web 模块 BankWeb，点击右侧的“Security”选项卡，点击“Security Constraints”右侧的“Add”按钮，添加一条安全性规则。

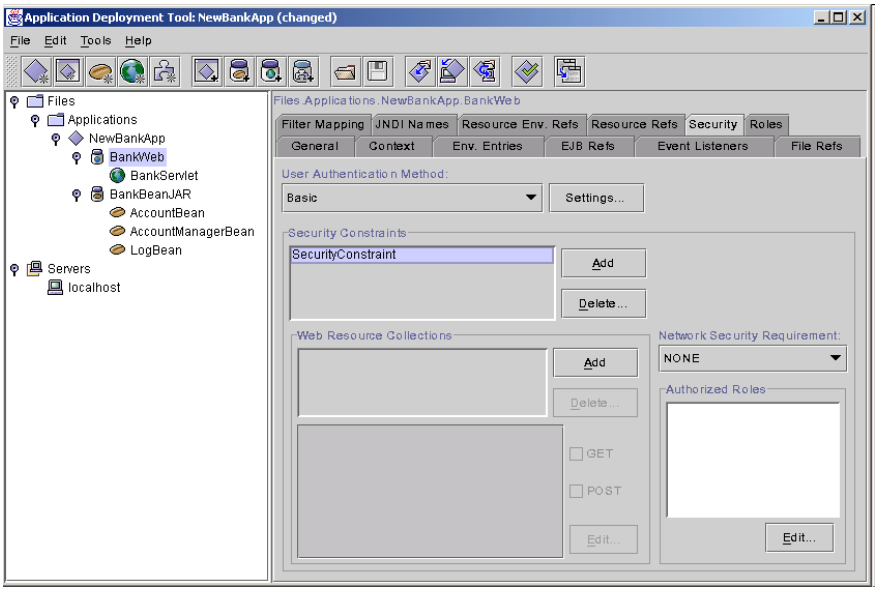


图 9-18 添加一条安全性规则

如图 9-19 所示，点击“Web Resource Collections”右侧的“Add”按钮，添加安全性规则所施加的一个 Web 资源集合。一条安全性规则可施加于多个 Web 资源集合。

然后点击界面下侧中部的“Edit”按钮，在弹出的图 9-20 所示的界面中，点击“Add URL Pattern”按钮添加 BankServlet 的别名对应的 URL“/BankServlet”。由于 Web 模块中的安全性规则仅施加于 Servlet 构件 BankServlet，因此只需在当前 Web 资源集合中添加该 Servlet 对应的 URL 即可。

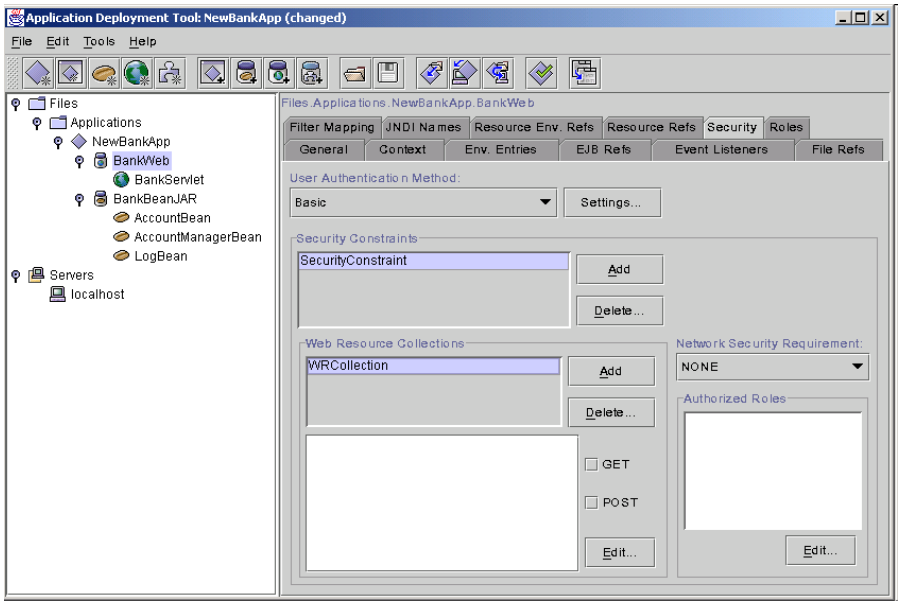


图 9-19 添加一个 Web 资源集合

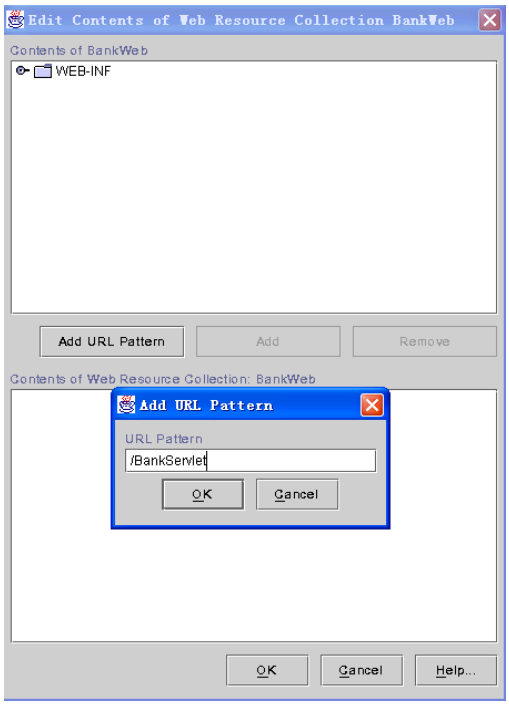


图 9-20 在 Web 资源集合添加 BankServlet 的 URL

如图 9-21 所示，选中“Get”复选框以控制 BankServlet 构件的 HTTP GET 方式访问，点击界面右下角的“Edit”按钮，在弹出的图 9-22 所示的界面中，将 NormalUser 与 Supervisor 两个角色均加入到授权角色列表中，以使得这两个角色都可以访问 BankServlet。

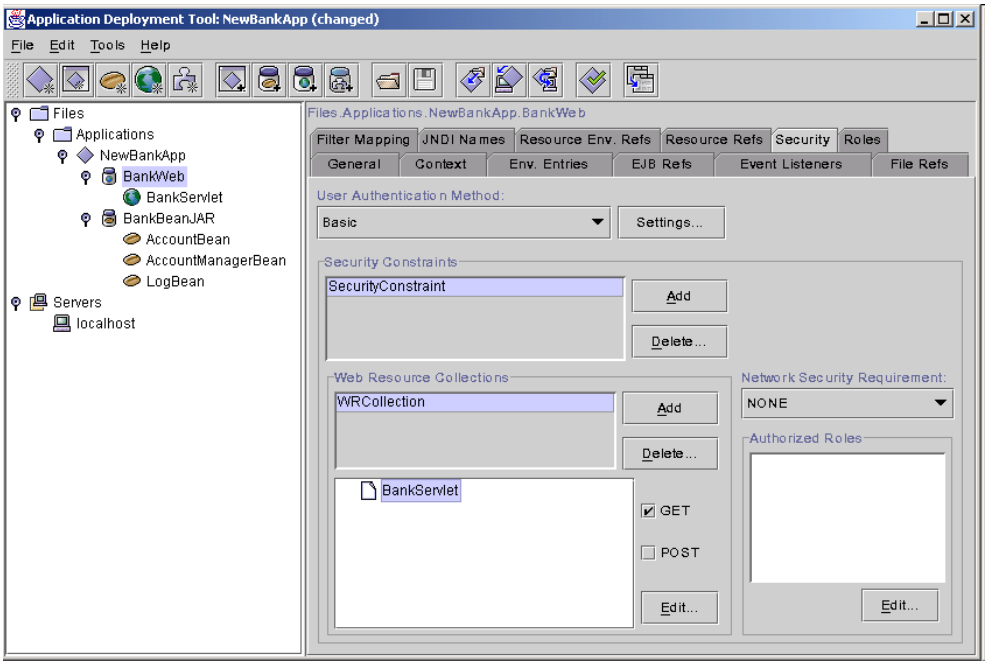


图 9-21 选择控制 HTTP GET 方式访问

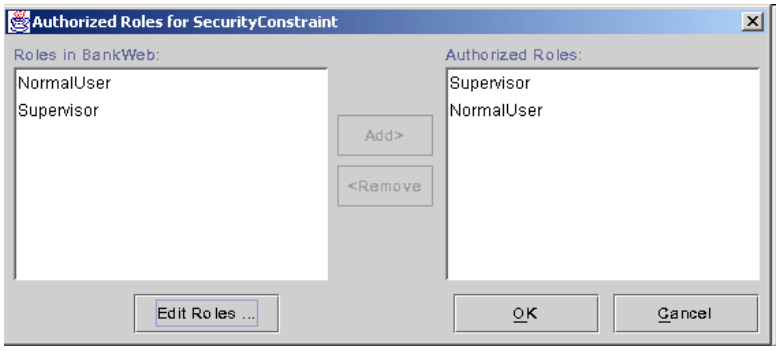


图 9-22 添加可以访问的角色

如图 9-23 所示，在 deploytool 中选中 BankServlet 构件，点击右侧的“Security”选项卡，设置 BankServlet 的安全身份（Security Identity）为使用调用者的身份（Use Caller ID）。

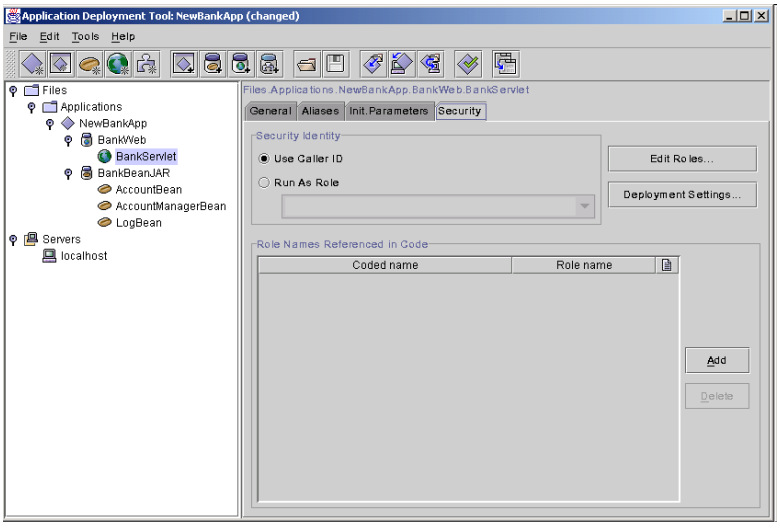


图 9-23 设置 BankServlet 的安全身份

安全身份指 J2EE 应用中构件在访问其它构件时，如本例中的 BankServlet 构件访问会话构件 AccountManager 的情形，该构件（BankServlet）的访问身份。安全身份有两种选择：

- Use Caller ID：使用访问者的身份，使用此种设置时，构件（BankServlet）将使用访问自己的访问者的身份来访问其它构件（AccountManager）。
- Run As Role：使用固定的身份访问其它构件，使用此种设置时，不论构件自己（BankServlet）的访问者是认证，构件都将使用指定的固定的身份访问其它构件（AccountManager）。

（4） 配置 EJB 模块的授权规则

如图 9-24 所示，在在 deploytool 中选中会话构件 AccountManager，点击右侧的“Security”选项卡来配置该构件的安全性规则。构件 AccountManager 的安全身份使用缺省值。选中构件的 Remote 接口，可以看到所有方法的缺省授权均为所有从可以访问，将 getBalance 方法的授权规则设为可以由 NormalUser 与 Supervisor 访问，而 createAccount 与 removeAccount 方法的授权规则设为只能由 Supervisor 访问。

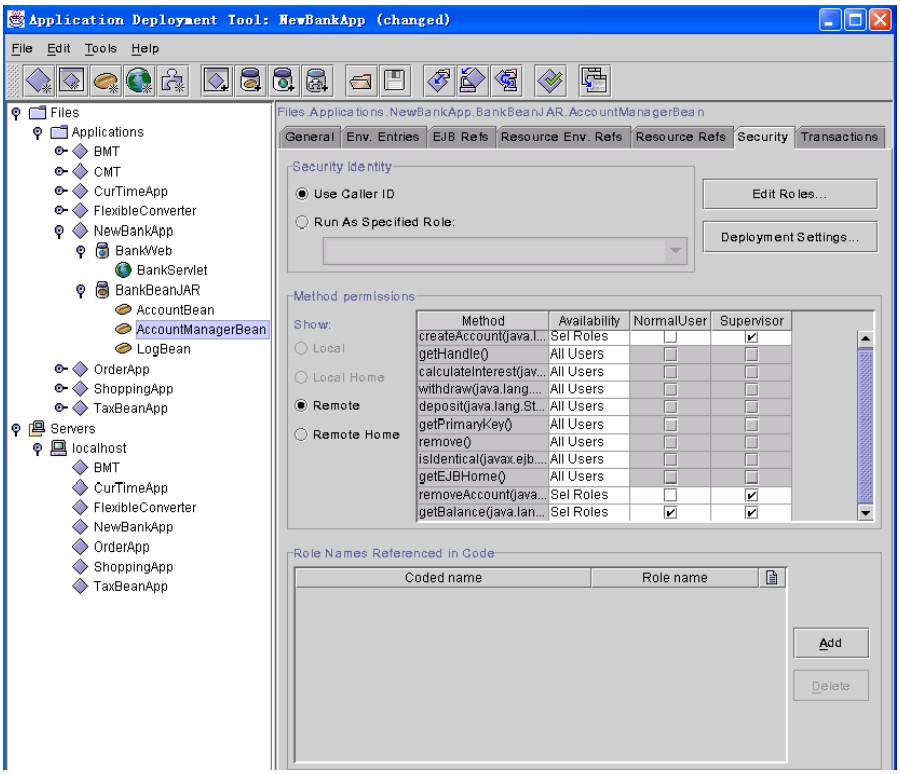


图 9-24 配置会话构件的安全性规则

（5） 将安全性角色映射到实际的安全域

最后一步的配置是将安全性角色映射到实际的安全域，不同的 J2EE 平台支持不同的安全域，如很多平台支持使用操作系统的用户管理系统，J2EE 参考实现允许使用平台自带的用户管理系统。点击 deploytool 的“Tools\Server Configuration”菜单打开服务器配置界面，如图 9-25 所示，选中“J2EE Server”下的 Users，可以配置应用可使用的用户信息。本例中我们为银行应用增加 managerwang、managerli 与 user1 三个用户（增加用户后需要重新启动 J2EE 服务器以使得修改的信息生效）。

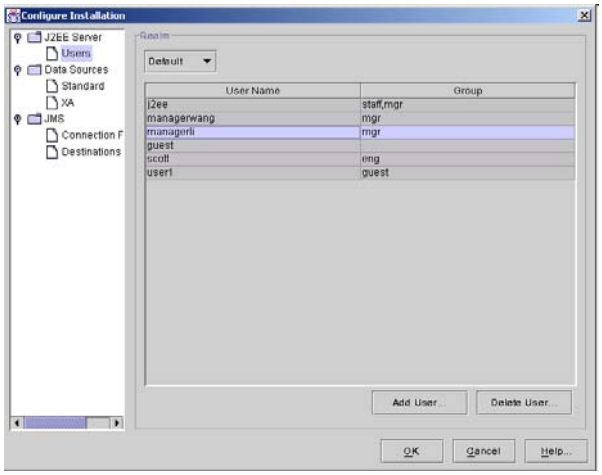


图 9-25 增加银行应用使用的用户

创建好应用使用的用户之后，我们就可以将定义的安全性角色映射到实际的用户了。如图 9-26 所示，在 deploytool 中选中银行应用 NewBankApp，点击右侧的“Security”选项卡，选中要映射的安全性角色并点击下面的“Add”按钮，在图 9-27 所示的弹出界面中添加要映射的用户。本例中我们将用户 managerwang 与 managerli 映射到角色 Supervisor，将用户 user1 映射到角色 NormalUser。

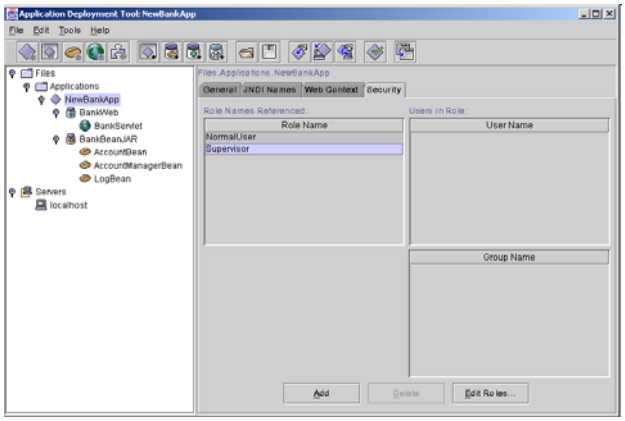


图 9-26 为角色映射用户

图 9-27 选择要映射的用户

4. 重新测试应用

完成上述安全性配置之后，将应用重新布署（重新布署后可能需要重新在数据库表 AccountBeanTable 中插入测试用记录），我们可以对银行应用进行重新测试。

打开浏览器，在地址栏中输入前面给出的 URL 来访问银行应用中的 BankServlet，如图 9-28 所示，可以看到浏览器弹出一个登录对话框请求输入用户名与密码信息。

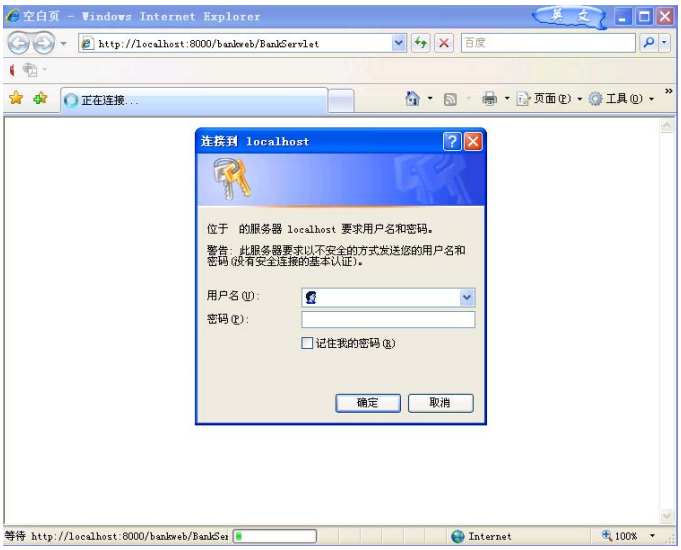


图 9-28 输入用户名与密码

如果三次验证不通过，则无法访问 BankServlet，客户端会显示图 9-29 所示的错误信息页面，提示用户身份验证失败。

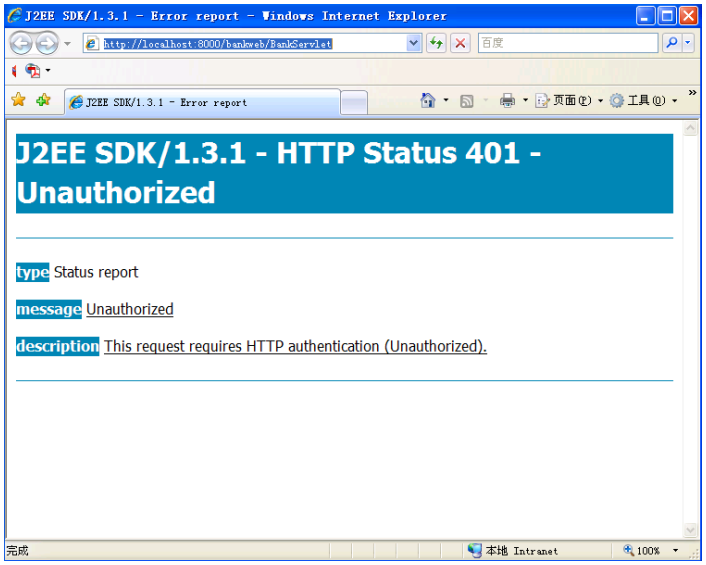


图 9-29 输入用户名与密码

重新访问 BankServlet，如图 9-30 所示输入用户 managerli 及相应密码，可以得到图 9-31 所示的访问结果。由于用户 managerli 映射到 Supervisor 角色，而前面配置的 Web 模块授权规则允许 Supervisor 访问 BankServlet，因此可以访问到 BankServlet；BankServlet 的安全身份配置为“使用调用者身份”，因此 BankServlet 调用会话构件 AccountManager 时使用的身份为访问者的身份，即 Supervisor，因此 AccountManager 的 getBalance、createAccount、removeAccount 方法均可以访问。

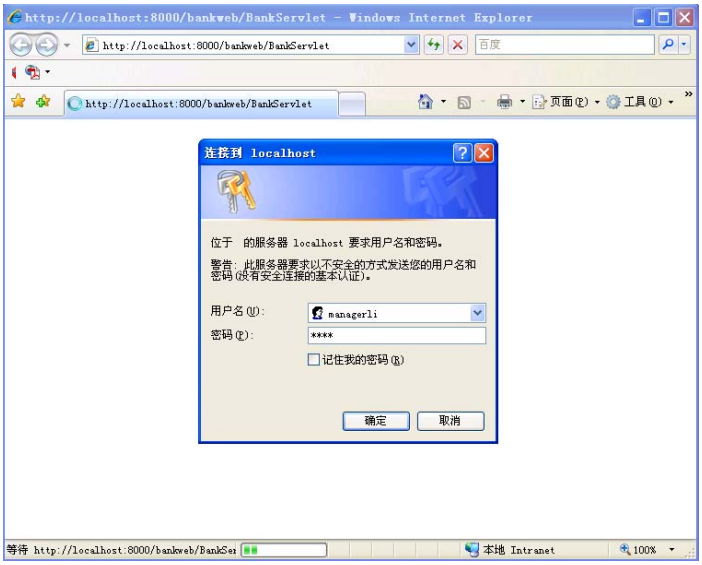


图 9-30 输入用户名 managerli 与密码

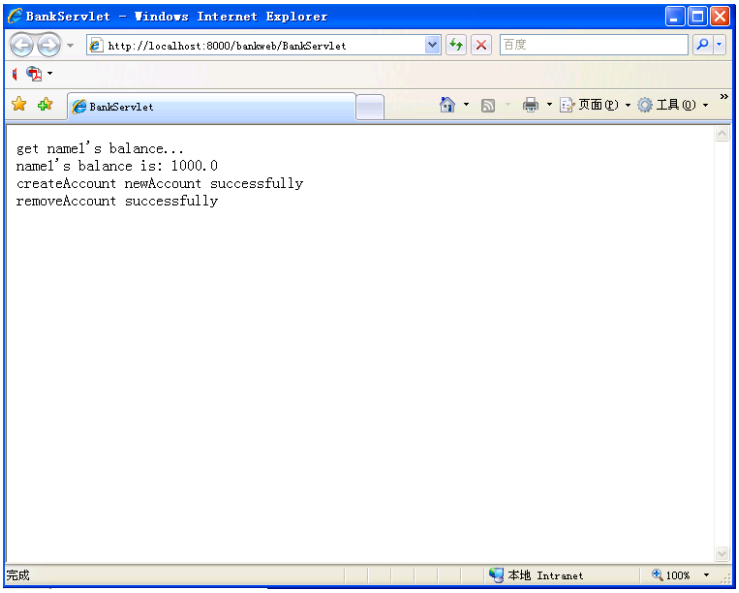


图 9-31 用户 managerli 访问结果

关闭浏览器，重新访问 BankServlet，如图 9-32 所示输入用户 user1 及相应密码，可以得到图 9-33 所示的访问结果。由于用户 user1 映射到 NormalUser 角色，而前面配置的 Web 模块授权规则允许 NormalUser 访问 BankServlet，因此可以访问到 BankServlet；BankServlet 的安全身份配置为“使用调用者身份”，因此 BankServlet 调用会话构件 AccountManager 时使用的身份为访问者的身份，即 NormalUser，因此 AccountManager 的 getBalance 可以被访问，而 createAccount、removeAccount 方法只能由 Supervisor 访问，因此这两个方法不可访问。

图 9-32 输入用户名 user1 与密码

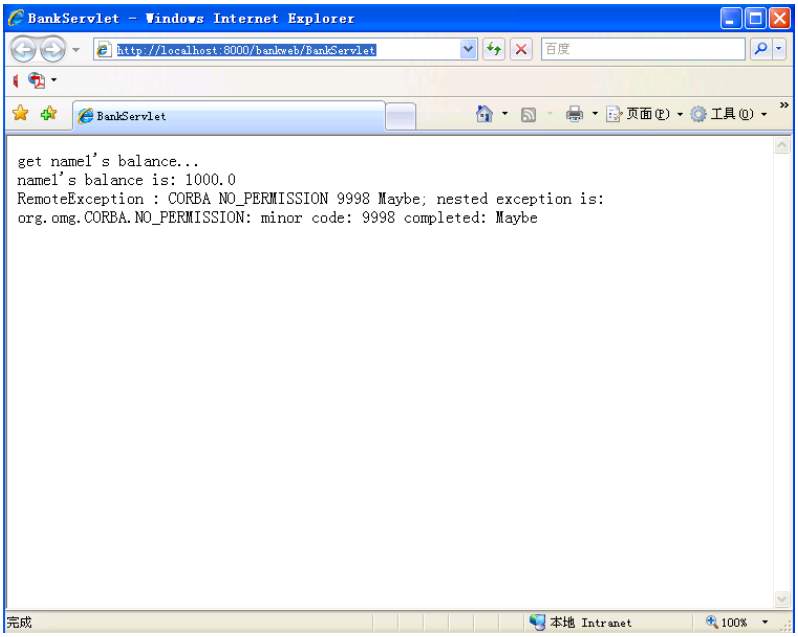


图 9-33 用户 user1 访问结果

§ 9.4 EJB3.0 概述

从前面的讨论可以看出，EJB 构件的开发需要开发人员遵循很多 EJB 规范的细节，这给开发人员带来了不小的负担，有一种观点认为“EJB 大概是 J2EE 架构中唯一一个没有兑现其能够简单开发并提高生产力的组件”，EJB3.0 规范正尝试在这方面作出努力以减轻其开发的复杂性。

Sun 于 2004 年发布了 EJB3.0 的草稿，2005 年 10 月，开源 J2EE 平台 jboss 宣布开始支持 EJB3.0，Sun 于 2006 年 6 月正式发布了 EJB3.0 规范，并同期发布了 Java EE5（Java Enterprise Edition 5）。EJB3.0 的主要目标是简化开发，同时这也是 Java EE5 平台的主要目标之一。

新规范利用 Java5 中的程序注释工具取消或最小化了很多(以前这些是必须实现)回调方法的实现，同时借鉴了 Hibernate 的成功之处，降低了实体 Bean 及 O/R 映射模型的复杂性。

与旧规范相比，EJB3.0 的主要变化包括：

1. 基于程序注释的新 EJB 模型

在 EJB3.0 中，EJB 构件是一个加了适当注释的简单 Java 对象——POJO（plain-old-Java-object，或 plain-original-Java-object）。其中：

- 部署描述符不在是必须的了（配置通过注释直接写到 Bean 代码中）
- Home 接口没有了
- Bean 类可以实现 Remote 接口也可以不实现它

也就是说，简单情况下，EJB 构件可以只包含一个 Java 类，程序 9-13 给出了一个符合 EJB3.0 规范的简单无状态会话构件的源代码：

程序 9-13　简单的无状态会话构件代码

```
import javax.ejb.*;
/**
* A stateless session bean requesting that a remote business
* interface be generated for it.
*/
@Stateless
@Remote
public class HelloWorldBean {
    public String sayHello() {
        return "Hello World!!!";
    }
}
```

程序 9-14 给出了一个符合 EJB3.0 规范的简单有状态会话构件的源代码：

程序 9-14　简单的有状态会话构件代码

```
import javax.ejb.*;
@Stateful
@Remote
public class MyAccountBean implements Serializable{
    private int total = 0;
    private int addresult = 0;
    public int Add(int a, int b) {
        addresult = a + b;
        return addresult;
    }
    public int getResult() {
        total += addresult;
        return total;
    }
    @Remove
    public void stopSession () {
        System.out.println("stopSession()方法被调用");
    }
}
```

可以看出，开发人员可以不用再定义 Home 接口和 Remote 接口（平台可以自动生成它），而且打包时的基本配置信息可以以注释的方式（@Stateless 与@Remote 等）直接写在 EJB 的源代码中，需要使用的生命周期管理相关的方法可以以注释的方式在程序中进行标记（如程序 9-14 中标记了@Remove 的 stopSession 方法）。

2. 改进的实体构件及 O/R 映射模型

在 EJB3.0 中，借鉴了 Hibernate 持久化支持的成功之处，改进了实体构件与 O/R 映射模型，程序 9-15 给出了一个符合 EJB3.0 的简单实体构件的源代码：

程序 9-15　简单的有状态会话构件代码

```
import javax.ejb.*;
@Entity
@Table(name = "Person")
```

```
public class Person {
    private Integer personid;
    private String name;
    private boolean sex;
    @Id
    @GeneratedValue
    public Integer getPersonid() {return personid;}
    public void setPersonid(Integer personid) {this.personid = personid;   }
    @Column(name = "PersonName",nullable=false,length=32)
    public String getName() {   return name;}
    public void setName(String name) {this.name = name;}
    @Column(nullable=false)
    public boolean getSex() {return sex;}
    public void setSex(boolean sex) {this.sex = sex;}
}
```

需要注意地是，从开发角度看，新的 EJB 规范看起来发生了很大变化（尤其是对于初学者来说），但是支撑 EJB 构件的核心机制并没有发生变化，在开发实际复杂应用时，本书第二部分所讨论的 EJB 构件开发与 J2EE 公共服务相关内容仍然是开发人员应该掌握的必要内容。

§ 9.5 EJB 与 CORBA 对比

作为主流中间件标准，J2EE/EJB 与 CORBA 均为软件系统开发提供了强大的构件支持与丰富的公共服务支持，在实际项目开发时，选择基于哪种规范进行开发通常取决于待开发项目的需求特点与规范优势的匹配程度。现就以下几个方面对 J2EE/EJB 规范与 CORBA 规范进行简要对比。

1. 工业标准特性

可以说，J2EE/EJB 与 CORBA 提供的都是一种工业标准，而非一个软件产品，不同供应商的竞争导致市场上存在大量高质量的、完全遵循 J2EE/EJB 或 CORBA 的产品。共同遵循的标准为也应用系统提供了较好的可移植性。

2. 对分布式对象的支持

J2EE/EJB 与 CORBA 均采用 Stub/Skeleton 结构对分布式对象提供了良好的支持，基于 Stub/Skeleton 结构，开发人员不需编写底层通信代码即可实现分布式对象与客户端之间跨越网络的交互。

但是在 J2EE/EJB 规范中，每个 EJB 构件与客户端通过两对 Stub/Skeleton 交互，并且直接通过 Skeleton 访问的并不是真正的 EJB 构件，而是容器自动生成的对象，这样有助于容器更好地为 EJB 构件提供公共服务。另外 EJB 支持本地接口，同进程内的 EJB 构件之间可以通过本地接口获得较高的访问效率。而 CORBA 则对客户端的动态调用接口与服务端的动态框架接口提供了较好的支持，可以适应一些特殊场合应用的开发。

3. 对公共服务的支持

J2EE/EJB 与 CORBA 均提供了丰富的公共服务支持，基于这些公共服务的支持，应用构件可以直接将命名服务、事务服务、安全服务、消息服务、资源管理服务等功能直接集成到应用系统中，而不需要开发人员自行实现。

相对而言，CORBA 规范中提供的公共服务涵盖的范围更广，OMG 甚至对特定领域内（如电信领域、医疗领域等）通用的功能进行了标准化，这使得从最基本的如命名服务、事务服务等公共服务一直到领域内通用的功能，都具备了直接将平台提供商实现的功能集成入应用系统的可能。但是 CORBA 中提供的公共服务多局限于基于 API 调用的使用方式，这些服务的使用方式可对应到 J2EE/EJB 规范中的 Service API，而 J2EE/EJB 规范除了以 API 方式提供的公共服务之外，还提供丰富的运行时服务，这些运行时服务可通过配置的方式直接

使用，从而为应用提供了较好的源代码以外的可定制性。

此外，J2EE/EJB 规范提供了强大的构件生命周期管理服务，平台/容器会根据运行与客户端请求状况自动管理构件的生命周期，而 CORBA 中需要开发人员编写服务程序来管理服务端的应用构件，但是 CORBA 也提供了相应的机制（如伺服对象管理器机制）来帮助开发人员管理服务端构件。（另外开发人员应注意“软件系统中的功能交给平台自动完成”永远不绝对是好事情！）

4. 开发过程

把 3.2 节描述的 CORBA 应用的开发过程一般化，我们可以用类似的过程描述一般分布式应用的开发过程，如图 9-34 所示，其基本过程为首先定义分布式对象的接口，然后分别完成服务端与客户端的开发，最后布署运行应用程序。下面对开发过程中 J2EE/EJB 与 CORBA 的不同之处进行简化分析。

编写对象接口：在 CORBA 中，对象接口是用 OMG IDL 定义的，而 IDL 可以映射到不同的程序设计语言，因此 CORBA 应用的服务端与客户端可以使用不同程序设计语言实现；在 J2EE/EJB 中，EJB 构件的接口是用 Java 语言的 interface 定义的，因此 EJB 构件及其客户端均必须用 Java 语言实现。

编写服务端程序：在 CORBA 中，服务端除了需要按照接口的约定实现分布式对象外，开发人员还需编写一个服务程序来将分布式对象准备好（创建并激活）；而在 J2EE/EJB 中，开发人员不需编写类似的服务程序，容器会根据运行与客户端请求状况自动创建或删除构件。但是将构件生命周期管理交给容器自动管理同时也意味着开发人员丧失了对服务端程序的更多控制力。

编写客户端程序：在 J2EE/EJB 和 CORBA 中，编写客户端的工作非常相似，只是在 EJB3.0 之前，EJB 的客户端需要首先利用 Home 的 create 方法获取一个可用的 EJB 引用，过程比 CORBA 客户端要复杂。

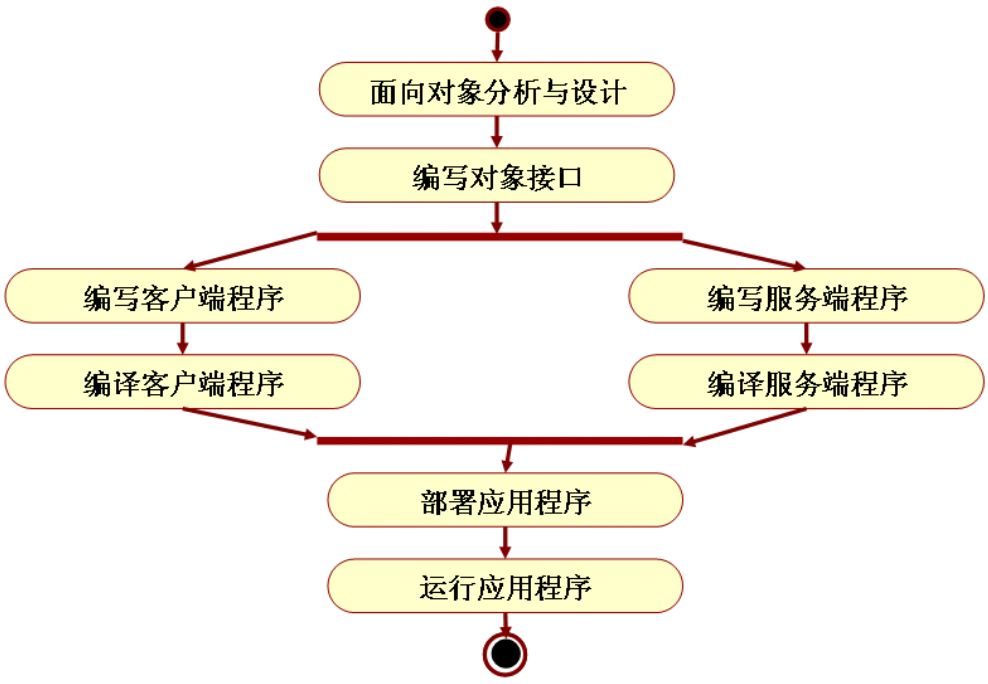


图 9-34 分布式系统的一般开发过程

布署应用程序：与 CORBA 相比，J2EE/EJB 提供了丰富的源代码以外的可定制性，其应用的布署过程通常比 CORBA 应用的布署要复杂，但是随着复杂的布署过程，应用也获得了丰富的可定制性支持。

总地来说，按照作者的观点，J2EE/EJB 规范比较适合基于 Web 的分布式应用的开发，而 CORBA 则在支持可互操作要求高的分布式系统开发方面有更大的优势。

思考与练习

9-1 在 9.2 节的 CMT 例子中，如果 withdraw 方法的执行过程中发生了 SQLException，那么对应事务最终会提交还是回滚？为什么？

9-2 在 9.2 节的 BMT 例子中，某次执行如果既没有执行 commit，也没有执行 rollback，那么事务最终的执行状态如何？这样对于应用系统来说有什么害处？

9-3 试为 9.3 节中的例子增加更详细的安全性控制规则，如可为实体构件也增加授权控制，并重新布署并测试你的配置效果。

第四部分 Web Service 规范

第 10 章 Web Service 规范

§ 10.1 Web Service 体系结构

1. Web Service 的提出

Web Service（Web 服务）是近几年兴起的一种新的分布式计算模型，下面从 Web 的发展过程简单讨论 Web Service 的提出过程。

在 Web 早期阶段，主要是利用传统的 Web 服务器向 Web 访问者提供静态内容，此时的技术基础主要包括 Web 服务器、HTTP 协议、HTML 语言等，由于此阶段 Web 上可访问的内容以静态的文档内容为主，因此该阶段也称为文档式 Web 阶段。随着 Internet 的发展与普及，人们开发利用应用服务器在 Web 上提供动态内容，支持访问者在 Web 上执行商业逻辑与网上交易，此阶段的技术基础包括应用服务器、动态网页技术、分布处理等，由于此阶段 Web 上可以完成交互式的应用功能，因此该阶段也称为应用式 Web 阶段。从 Web Service 角度看，Web 发展的第三阶段称为服务式 Web 阶段，该阶段的主要变化是引入了 Web Service，其基本思想是将 Web 应用、动态内容以及交易功能包装成程序可以访问的服务，即为 Web 提供程序可以访问的界面，图 10-1 所示的 Google 网站上一幅图片很形象的刻画了 Web Service 带来的变化：Web 的访问者不再局限于人，程序（软件）也可以充分利用 Web 上丰富资源。

图 10-1 Web Service 为 Web 带来的变化

Web Service 是一系列标准的集合，包括 SOAP、UDDI、WSDL、WSFL/BPEL 等，Web Service 利用这些标准提供了一个松散耦合的分布式计算环境。在 Web Service 模型中，厂商将其服务封装成一个个相对独立的 Web Service，每个服务提供某类功能；客户（或其它厂商）可以通过 SOAP 协议来访问这些服务。Web Service 的主要特征是将可调用的功能发布到 Web 上以供程序访问。

2. SOA 架构

Web Service 使用 SOA（Service Oriented Architecture，面向服务架构）架构。如图 10-2 所示，SOA 架构中包含三个主要参与者：服务提供者（Service Provider）、服务请求者（Service Requester）与服务代理（Service Broker）。涉及发布（Publish）、查找（Find）与绑定/调用（Bind/Invoke）三个基本操作。基本的工作过程为：

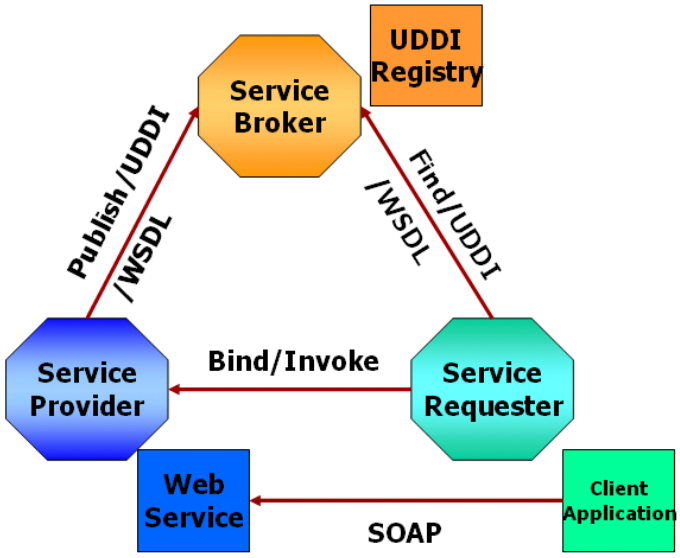


图 10-2 SOA 架构

（1） 服务提供者将所提供的服务发布到服务代理的一个目录上（即 Publish 操作）；

（2） 服务请求者首先到服务代理提供的目录上搜索服务，得到如何调用该服务的信息（即 Find 操作）；

（3） 根据得到的信息调用服务提供者提供的服务（即 Bind/Invoke 操作）。

在 Web Service 架构中，使用的主要标准如下：

- **SOAP**：Simple Object Access Protocol，简单对象访问协议，用来执行 Web 服务的调用。
- **WSDL**：Web Service Description Language，Web 服务描述语言，用来描述 Web 服务。
- **UDDI**：Universal Description, Discovery and Integration，用来发布、查找服务
- **WSFL/BPEL**： Web Service Flow Language/ Business Process Execution Language，用于将分散的、功能单一的 Web 服务组织成一个复杂的有机应用。

Web 服务通常具有以下特征：

- 完好的封装性：使用者仅看到 Web Service 提供的功能列表，不能看到其实现细节；
- 松散耦合：只要服务的接口保持不变，就不影响使用者，而且客户端和服务端的生命周期不相互影响，同时需要一种适合 Internet 环境的消息交换协议。
- 使用标准协议规范：Web Service 架构使用开放的标准协议进行描述、传输和交换。
- 高度可互操作性：可以跨越平台、语言进行调用。
- 高度可集成能力：高度可互操作性所带来的高度可集成能力使得 Web Service 可用于集成异构的软件系统。
- 动态性：可以自动发现服务并进行调用。

3. Web Service 分类

Web 服务的主要应用领域可分为以下四类：

（1） 面向商业应用的 Web Service（Business-Oriented Web Service）：将企业内部的大型系统，如 ERP、CRM 系统等，封装成 Web Service 的形式在网络中（Internet or Intranet）提供，基于 Web Service 的高度可互操作性，这使得企业内部的应用更容易集成，企业间的众多合作伙伴的系统对接更加容易。这是 Web Service 目前最主要的应用方向之一。

（2） 面向最终用户的 Web Service（Customer-Oriented Web Service）：指 Web Service

在电子商务领域的应用，主要指针对原有 B2C 网站的改造，Web Service 技术为 B2C 网站增加了 Web Service 的应用界面，使得桌面工具可以提供跨越多个 B2C 服务的桌面服务，如将机票预定、炒股等服务集成到一个个人理财桌面系统中，这使得用户使用 Internet 更加方便，能够获得更加便捷的服务。

（3） 面向特定设备的 Web Service（Device-Oriented Web Service）：指面向手持设备、日用家电等特定接入设备的 Web Service，将原有的如天气预报、E-mail 服务、股票信息等网络服务封装成 Web Service，支持除 PC 以外的各种终端。

（4） 系统级 Web Service（System-Oriented Web Service）：这类 Web Service 相当于本书中讨论的公共服务，指将 Web 应用中诸如用户权限认证、系统监控等通用功能封装成 Web Service，发布到 Internet 或者企业内部的 Intranet 上，其作用范围将从单个系统或局部网络拓展到整个企业网络或整个 Internet 上，如一个跨国企业的所有在线服务可以使用同一个用户权限认证服务。

4. Web Service 协议栈

W3C 在 2001 年的 Web Service 专题研讨会上提出了 Web Service 协议栈，该协议栈定义了 Web Service 使用的技术标准与发展方向，如图 10-3 所示。

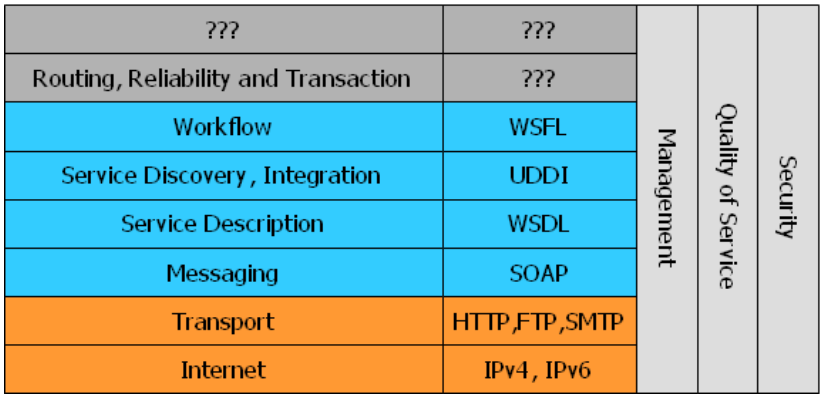


图 10-3 Web Service 协议栈

Web Service 追求的第一目标是简单性，首先 Web Service 使用的协议本身是简单的；另外一个可以使用的 Web Service 可以按照需要选择若干层次的功能，而无需所有的特性，例如，一个简单应用可能只要使用 WSDL/SOAP 就可以架构一个符合规范的 Web Service；最后，Web Service 所有的机制都基于现有的技术，并没有创造一个完整的新体系，而是继承原有的被广泛接受的技术。

Web Service 目前最主要的应用方向为集成企业应用系统。集成企业原有系统利用 Web Service 的高度可集成特性，将企业运作的各个环节有效的联系起来，组成一个协同工作的整体，从而使得企业的所有业务都真正“自动化”起来。

Web Service 目前的主要问题包括效率问题与安全性、事务控制等高级特性比较欠缺等。

§ 10.2 SOAP

SOAP 是在松散的、分布的环境中使用 XML 交换结构化的和类型化的信息的一种简单协议。SOAP 本身并不定义任何应用语义，如编程模型或特定语义实现，它只定义了一种简单的以模块化的方式包装数据的机制。

1. SOAP 消息结构

SOAP 消息是 Web Service 架构中不同结点间传送信息的基本数据单元，其结构如图 10-3 所示。一个 SOAP 消息是由 XML 消息头和一个 SOAP 封装（SOAP Envelope）组成的 XML 文档。SOAP 封装描述了该消息所包含的基本信息，SOAP 封装中包括一个可选的 SOAP 消息头（SOAP Header）和必需的 SOAP 消息体（SOAP Body）。SOAP 封装定义了一个 SOAP 错误（SOAP Fault）来报告出错信息。

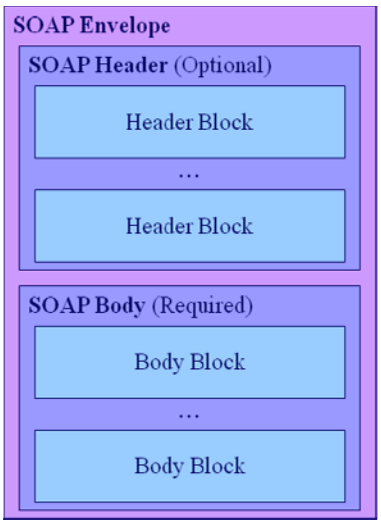


图 10-4 SOAP 消息结构

- **SOAP Envelope**

SOAP 封装是 SOAP 消息文件的顶层元素，其必须存在且元素名称为<env:Envelope>。该元素可以包含命名空间和声明的额外属性，其额外属性必须用命名空间修饰。SOAP 封装中除了 SOAP 消息头和 SOAP 消息体外，也可以包含其它的子元素。若存在其它子元素，则必须出现在 SOAP 消息体之后。

SOAP 消息中必须使用 SOAP 封装的命名空间。其语法格式如：

xmlns:env =“http://schemas.xmlsoap.org/soap/envelope/”

SOAP 编码方式（SOAP EncodingStyle）是 SOAP 封装的一个属性，用于定义在文档中使用的数据类型。该属性可以出现在任何 SOAP 元素中，并会被应用到元素的内容和元素的子元素上。其语法格式如：

env:encodingStyle =“http://schemas.xmlsoap.org/soap/encoding/”

- **SOAP Header**

如果存在 SOAP 消息头时，它必须作为 SOAP 封装的第一个子元素且元素名称为<env:Header>。消息头主要传递一些辅助性的附加消息，如身份验证、会话等，在 SOAP 消息头中，可以通过添加基于 XML 格式的消息条目实现对消息进行扩展，典型的应用有认证机制、事务管理和支付等。SOAP 消息头的所有子元素被称为 Header 条目，每个 Header 条目是由一个命名空间的 URI 和局部名来组成，SOAP 规范中不允许出现不带命名空间修饰的 Header 条目。

Header 条目包含了 Role、mustUnderstand 和 encodingStyle 属性。

SOAP Role 属性可被用于将 Header 元素寻址到一个特定的端点。其语法格式如：

env:role=“http://schemas.xmlsoap.org/soap/actor/”

SOAP mustUnderstand 属性可用来标识消息头对于要对其进行处理的接收者来说是强制的还是可选的。其语法格式如：

env:mustUnderstand=“true”

上例表示该消息的接收者必须认可此元素，否则消息头处理失效。

SOAP encodingStyle 属性用于指定 Header 条目的编码方式。

- **SOAP Body**

SOAP 消息体为消息的最终接受者所必须处理的信息提供了一个容器，它包含了所交换数据的实际内容，其元素名称为<env:Body>。SOAP 消息体是 SOAP 封装必需的元素，如果有 SOAP 消息头，则消息体必须紧跟消息头之后，否则，其为 SOAP 封装的第一个直接子元素。SOAP 消息体的所有子元素被称为 Body 条目，每个 Body 条目是由一个命名空间的 URI 和局部名来组成，每个 Body 条目被作为 SOAP 消息体元素的独立子元素进行编码。

SOAP encodingStyle 属性用于指定 Body 条目的编码方式。

SOAP 消息头和消息体虽然是作为独立的元素定义的，但二者在实际上有一定的关联：当 Header 条目的 mustUnderstand 属性值为 true 而且该消息传递给默认的接收者时，语义上和 Body 条目是等价的。

- **SOAP Fault**

SOAP 错误用于在 SOAP 消息中携带错误和/或状态信息，其作为一个 Body 条目出现且不能出现多于一次，元素名称为<soap:Fault>。SOAP 错误定义了四个子元素：

<faultcode>	标识了引发错误的故障代码；
<faultstring>	提供可供人进行阅读的故障说明；
<faultactor>	消息传送过程中引发故障的结点信息；
<faultdetail>	传输与 SOAP 消息体元素相关的应用程序的错误信息。

SOAP 错误代码及其含义：

VersionMismatch	SOAP 封装包含的无效命名空间；
MustUnderstand	SOAP 消息头中被标识为 1 的 Header 条目未被认可；
DataEncodingUnknown	编码方式不正确；
Client	消息中包含了不正确的结构或信息；
Server	消息接收者没有对消息进行正确处理。

程序 10-1 SOAP 消息示例

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/soap-envelope">
  <env:Header>
    <data:headerBlock
      xmlns:data="http://example.com/header"
      env:role="http://example.com/role"
      env:mustUnderstand="true">
      ...
    </data:headerBlock>
    ...
  </env:Header>
  <env:Body>
    <data:bodyBlock xmlns:data="http://example.com/header">
      ...
    </data:bodyBlock>
    ...
  </env:Body>
</env:Envelope>
```

2. SOAP 调用过程

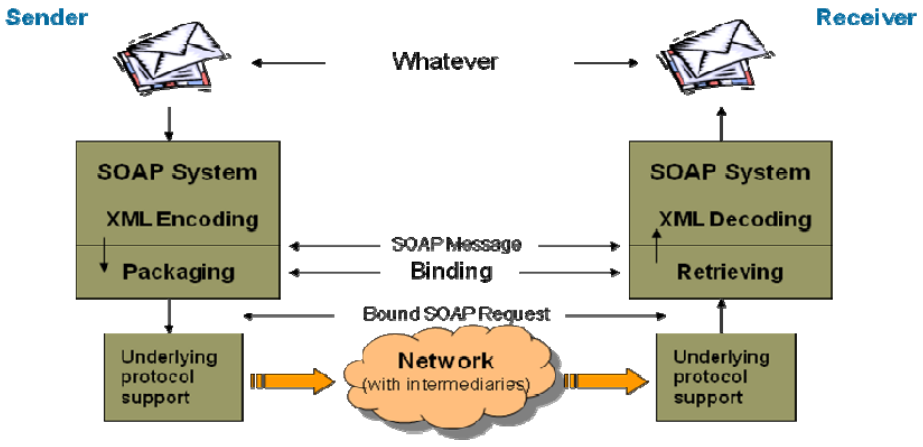


图 10-5 SOAP 消息调用过程

SOAP的基本思想是将数据/对象打包成XML格式的数据在网络环境中交换以达到服务调用的目的。如图 10-5 所示，发送者和接收者之间可以交换任何类型的消息数据。发送者利用 XML 编码将消息进行打包 SOAP 消息，通过底层的网络传输协议进行传输，接收者收到消息并将消息进行解码获取所需要的数据。如我们可以用程序 10-2 中给出的 XML 文档来封装一个程序 10-3 中定义的类的一个实例：

程序 10-2 封装对象的 XML 文档

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/soap-envelope">
  <env:Body xmlns:m="http://www.example.org/namelist>
     <m:Person>
       <m:name>zhang3</m:name>
       <m:age>20</m:age>
     </m:Person>
  </env:Body>
</env:Envelope>
```

程序 10-3 类 Person 的定义

```
public class Person{
    String name;
    int age;
    //方法的定义...
}
```

SOAP 没有定义任何底层的传输协议，可以使用 HTTP、FTP、SMTP 或者 JMS，甚至是自定义协议来传输 SOAP 报文，在 Internet 环境下一般使用 HTTP 协议。从某种意义上讲，SOAP 可以简单理解为“HTTP+XML+RPC”，即使用 HTTP 作为底层通信协议，XML 作为数据传输的格式，实现 Web 环境下的远程服务调用（RPC）。

下面通过一个简单实例演示 SOAP 协议的基本思想。假设在 Internet 上有程序 10-4 中给出的接口所约定的服务存在，客户端在远程调用 sayHelloTo 方法时，提供一个名字，期望返回一个字符串。

程序 10-4 服务的接口定义

```
public interface Hello
{
    public String sayHelloTo(String name);
}
```

假设没有 Stub/Skeleton 机制的支持，则开发者必须负责将方法调用串行化，并把消息发给远程服务器。一种简单的方式就是使用 XML 打包数据，我们可以很快得到程序 10-5 给出的 XML 文档来表示客户端的调用请求：

程序 10-5 XML 表示的请求信息

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/soap-envelope">
  <env:Body xmlns:m="http://www.example.org/namelist>
     <m:Hello>
       <m:sayHelloTo>
         <m:name>John</m:name>
       </m:sayHelloTo>
     </m:Hello>
  </env:Body>
</env:Envelope>
```

在程序 10-5 给出的 XML 文档中，我们将接口名作为根结点，方法作为子结点，而方法调用的参数作为方法的子结点。

如果在其调用的过程中，服务端发生了错误，则会引发如程序 10-6 所示的 SOAP 错误消息。

SOAP 消息作为 Web 服务规范的标准信息交换格式，解决了网络信息交换中“语言”不一致的问题，受到工业界的认可并在实践中加以应用。但 SOAP 消息的调用效率比较低，

HTTP 不是有效率的通信协议，并且 XML 需要额外的文件解析，一些格式更加简化的消息风格如 REST（REpresentational State Transfer，代表性状态转移）等得到了新的发展和应用。

程序 10-6 SOAP 错误消息

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/soap-envelope">
  <env:Body>
    <env:Fault>
       <faultcode>SOAP-ENV:Server</faultcode>
       <faultstring>Server Error</faultstring>
       <detail>
          <myfaultdetails>
             <message>
               Sorry, my silly constraint says that I cannot say hello on Tuesday.
             </message>
             <errorcode>1001</errorcode>
          </myfaultdetails>
       </detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

§ 10.3 WSDL

WSDL 是一种用 XML 文档来描述 Web 服务的标准，是 Web 服务的接口定义语言。其内容包含了 Web 服务的基本属性信息，如 Web 服务所提供的操作、和 Web 服务进行消息交换的数据格式和协议标准、Web 服务的网络位置等信息。

1. WSDL 数据结构

WSDL 以端口集合的形式来描述 Web 服务，WSDL 服务描述包含对一组操作和消息的一个抽象定义，绑定到这些操作和消息的一个具体协议，和这个绑定的一个网络端点规范。WSDL 文档分为两种类型：服务接口（Service interface）和服务实现（Service implementations），文档基本元素结构如图 10-6 所示。

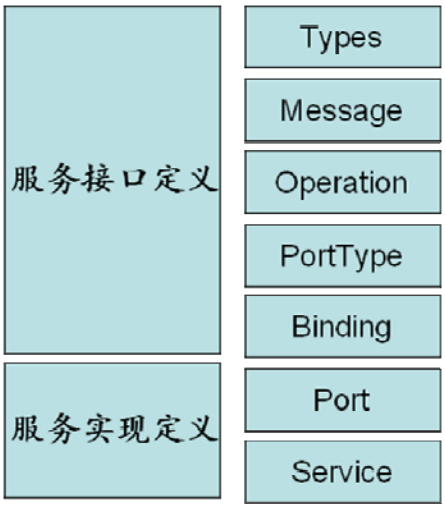


图 10-6 WSDL 元素结构图

WSDL 文档在 Web 服务的定义中使用了以下元素：

- Types　　数据类型定义的容器，它使用某种类型系统；
- Message　　消息结构的抽象化定义，使用 Types 所定义的类型来定义整个消息的

数据结构；

- Operation　对服务所支持操作的抽象描述，单个 Operation 描述了一个访问入口的请求/响应消息对；
- PortType　对某个访问入口点类型所支持的操作的抽象集合，这个操作可以由一个或多个服务访问点来支持；
- Binding　特定端口类型的具体协议和数据格式规范的绑定；
- Port　定义为协议/数据格式绑定与具体 Web 服务地址组合的单个访问点；
- Service　相关服务访问点的集合。

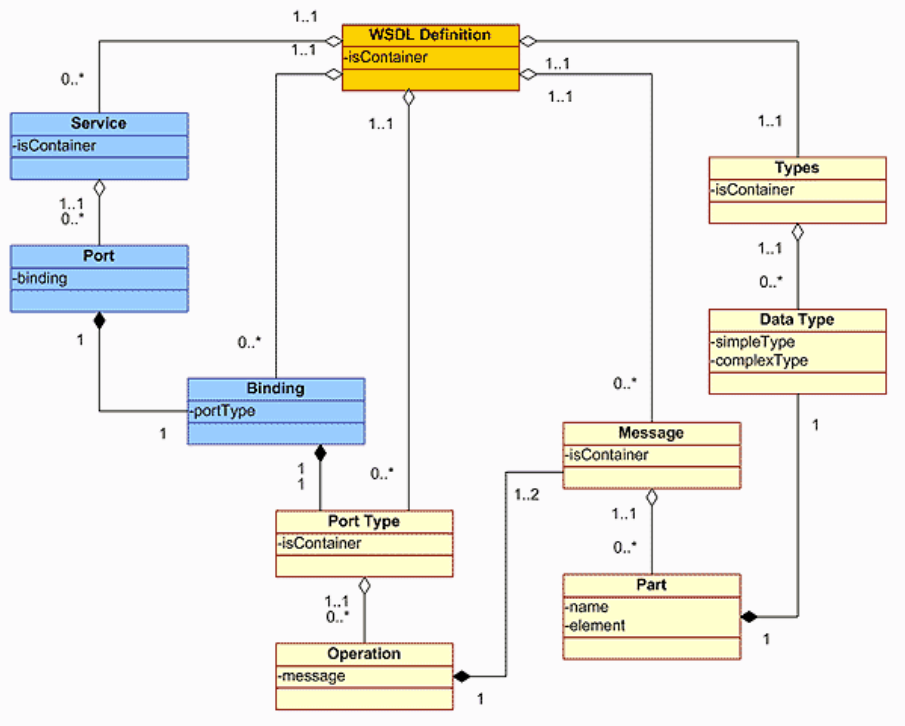


图 10-7 WSDL 元素的对象结构示意图

Types 是一个数据类型定义的容器，包含了所有在消息定义中需要的 XML 元素的类型定义。

Message 具体定义了在通信中使用的消息的数据结构，其中包含了一组 Part 元素，每个 Part 元素都是最终消息的一个组成部分，每个 Part 都会引用一个 DataType 来表示它的结构。Part 元素不支持嵌套，都是并列出现。

PortType 具体定义了一种服务访问入口的类型，即传入/传出消息的模式及其格式。一个 PortType 可以包含若干个 Operation，而一个 Operation 则是访问入口支持的一种类型的调用。WSDL 中支持四种访问入口调用的模式（单请求，单响应，请求/响应，响应/请求），其中，请求是从客户端到 Web 服务端的消息调用，响应则是从 Web 服务端到客户端的消息传递。

Types、Message 和 PortType 三种结构描述了所调用 Web 服务的抽象定义，其相对于 CORBA 中的 IDL 描述。

Service 描述一个具体的被部署的 Web 服务所提供的所有访问入口的部署细节，一个 Service 通常包含多个服务访问入口，而每个访问入口都会使用一个 Port 元素来描述。

Port 描述的是一个服务访问入口的部署细节，包括通过所要访问的 URL 及应当使用的消息调用模式等。其消息调用模式使用 Binding 结构来表示。

Binding 结构定义了某个 PortType 与某一种具体的网络传输协议或消息传输协议相绑定，其描述的内容与具体的服务部署相关。可以在 Binding 中将 PortType 与 SOAP/HTTP 绑定，也可以将其与 MIME/SMTP 相绑定等。

WSDL 的设计遵循了 Web 技术标准的开放理念，允许通过扩展使用如 XML Schema 等其他的类型定义语言，允许使用如 SOAP/HTTP 等多种网络传输协议和消息格式。WSDL 中应用了软件工程中的服用理念，分离了抽线定义层和具体的部署，使得前者的复用性大大增加。

2. WSDL 使用示例

通过一个简单的股票报价 Web 服务的 WSDL 定义文档来说明 WSDL 文档的定义和使用。该服务支持名称为 GetLastTradePrice 的唯一操作，通过 HTTP 协议在 SOAP1.1 上来实现。该请求接受一个类型为字符串的 tickerSymbol，并返回类型为浮点数的价格。

程序 10-7 WSDL 定义声明

```
<?xml version="1.0" ?>
<definitions name="StockQuote"
           targetNamespace="http://example.com/stockquote.wsdl"
           xmlns:tns="http://example.com/stockquote.wsdl"
           xmlns:xsd1="http://example.com/stockquote.xsd"
           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
           xmlns="http://schemas.xmlsoap.org/wsdl/">
```

程序 10-7 是股票报价服务 WSDL 的定义声明部分。包括 XML 版本头和 WSDL 定义声明，后者通过属性分别声明了目标命名空间、服务的主命名空间、XML Schema 命名空间、SOAP 绑定采用的命名空间和 XML 的命名空间。

程序 10-8 WSDL 数据类型定义

```
 <types>
   <schema targetNamespace="http://example.com/stockquote.xsd"
           xmlns="http://www.w3.org/1999/XMLSchema">
     <element name="TradePriceRequest">
       <complexType>
         <all>
           <element name="tickerSymbol" type="string"/>
         </all>
       </complexType>
     </element>
     <element name="TradePriceResult">
       <complexType>
         <all>
           <element name="price" type="float"/>
         </all>
       </complexType>
     </element>
   </schema>
 </types>
```

程序 10-8 是股票报价服务 WSDL 的数据类型定义部分。定义了两种元素的结构（TradePriceRequest 和 TradePriceResult），分别由字符串和浮点型的复合数据类型元素构成。

程序 10-9 WSDL 消息格式抽象定义

```
 <message name="GetLastTradePriceInput">
   <part name="body" element="xsd1:TradePriceRequest"/>
 </message>
 <message name="GetLastTradePriceOutput">
   <part name="body" element="xsd1:TradePriceResult"/>
 </message>
```

程序 10-9 是消息格式的抽象定义部分。定义了两种消息格式（GetLastTradePriceInput 和 GetLastTradePriceOutput），由名称为 body 的 part 组成，分别包含了程序 10-8 所定义的两种元素类型。

程序 10-10 WSDL 服务访问点调用模式

```
 <portType name="StockQuotePortType">
   <operation name="GetLastTradePrice">
```

```
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
```

程序 10-10 是服务访问点的调用模式定义部分。表明了股票报价 Web 服务的入口类型是请求/响应模式，其输入/输出消息为程序 10-9 所定义的消息格式。

程序 10-11 WSDL 绑定模式定义

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
        <input>
          <soap:body use="literal"
                namespace="http://example.com/stockquote.xsd"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
          <soap:body use="literal"
                namespace="http://example.com/stockquote.xsd"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
      </soap:operation>
    </operation>
  </soap:binding>
</binding>
```

程序 10-11 是服务访问点的抽象定义和 SOAP HTTP 的绑定定义部分。表明了股票报价服务的消息传输采用基于 SOAP 绑定的 HTTP 传输。规定了在 SOAP 调用时应当使用的 soapAction 和消息的编码规则采用 SOAP 规范默认的编码风格。

程序 10-12 WSDL 服务定义

```
<service name="StockQuoteService">
  <documentation>股票查询服务</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
  <soap:address location="http://example.com/stockquote"/>
  </port>
</service>

</definitions>
```

程序 10-12 是具体的股票报价服务的服务定义部分。定义了服务名称和提供的服务访问入口，使用的消息模式是由程序 10-11 的 Binding 所定义。

将程序 10-7 至 10-12 合并起来组成了对于股票查询 Web 服务的完整的 WSDL 定义文档。在 Web 服务的使用过程中，请求结点通过分析 WSDL 文档的描述，即可了解到 Web 服务的接口定义、部署及调用信息，利用 SOAP 请求消息访问 Web 服务。

§ 10.4 UDDI

UDDI 是一套基于 Web 的、分布式的并为 Web 服务提供注册的信息注册中心的实现规范，同时也包含了一组使企业能将自身提供的 Web 服务进行注册以使其它企业能够发现的访问协议实现标准。UDDI 标准包含一个独立于平台的框架，用于通过使用 Internet 来描述服务，发现企业，并对企业服务进行集成。通过使用 UDDI，Web 服务提供者能够注册 Web 服务，应用程序或服务使用者能够查找到所需要的 Web 服务并了解如何与之连接和交互。

1. UDDI 数据模型

UDDI 商业注册通过使用一个 XML 文档来描述企业及其提供的 Web 服务。其做法类似于电信公司发布电话号码所使用的黄页，客户可以很方便的从上面查找到需要的信息。从概

念上来说，UDDI 商业注册所提供的信息包括以下三部分内容：

- 白页（White Page）：有关企业的基本信息，包括了企业名称、经营范围的描述、联系信息和已知的企业标识等；
- 黄页（Yellow Page）：提供基于标准分类法的行业类别，能够使使用者在更大的范围内查找在注册中心注册的企业或服务；
- 绿页（Green Page）：关于该企业所提供的 Web 服务的技术信息和 Web 服务说明信息，并提供了指向这些服务所在实现的技术规范的引用和一些指针，指向基于文件的 URL 的不同发现机制。

UDDI 注册中心是由 UDDI 规范的一种或多种实现组成，它们可以互操作以共享注册中心数据。UDDI 商业注册中心是由通过被称为节点的一组 UDDI 注册中心构成，能够对外公开访问，通过互操作共享注册中心数据。在很多 UDDI 运营商（如 IBM、Microsoft、HP 和 SAP 等）站点上都冗余的放着 UDDI 商业注册中心的全部条目，只有该站点才能够对这些条目进行更改。

UDDI 的核心数据模型及其属性和子元素结构如图 10-8 所示。主要包括：

- businessEntity：业务详细资料，包括名称和联系信息等。UDDI 商业注册的商业信息发布和发现的核心 XML 元素都包含在该结构中。这个结构是行业试题专属信息集的最高层的数据容器，位于整个信息结构的最上层。
- businessService：业务提供的 Web 服务。该元素将一系列有关商业流程或分类目录的 Web 服务描述组合到一起，这些服务都是发布服务的商业实体所需要注册的 Web 服务。其中包含的信息可以根据行业、产品、服务类型或地域划分进行再次分类。
- bindingTemplate：调用 Web 服务的技术细节描述。其中包括应用程序连接远程 Web 服务并与之通信所必需的信息，如应用 Web 服务的地址、应用服务宿主和调用服务前必须调用的附加服务等，另外，还包括实现一些复杂的路由选择等。
- tModel：用于访问服务描述的技术标识。描述关于服务调用规范的元数据，包括服务名称、发布服务的组织和指向规范本身的 URL 指针等。
- publisherAssertion：展示业务实体之间的关联关系。
- operationalInfo：传输操作信息的数据结构。

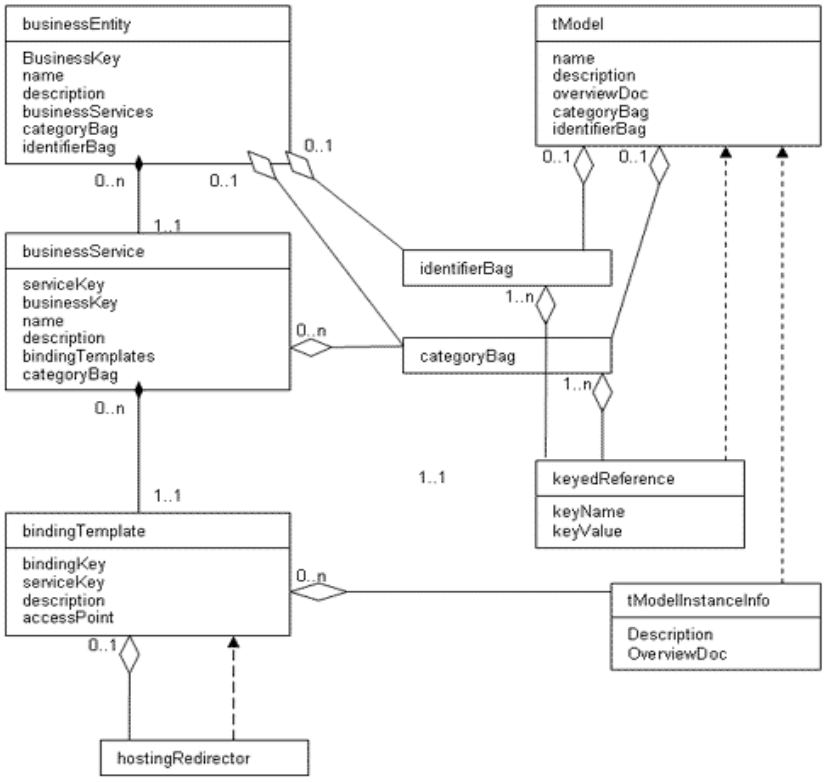


图 10-8 UDDI 核心数据模型关联图

其中，所有 businessEntity 元素中包含的信息支持“黄页”分类法；businessService 元素和 bindingTemplate 元素一起构成了“绿页”信息。

2. UDDI 调用过程

UDDI 提供一种统一的 Web 服务注册环境，用于存储和发布商业组织所提供的 Web 服务，便于 Web 服务使用者查找和调用服务。如图 10-8 所示，软件公司、标准组织或开发人员在 UDDI 商业注册中心中注册所要发布的 Web 服务的类型和描述信息；实施人员完成 Web 服务的开发和部署工作后，在 UDDI 中注册业务描述信息及其所发布的服务；UDDI 为每个服务和注册分配唯一的标识；市场、搜索引擎或业务应用等服务使用者利用 UDDI 提供的服务查询接口查询服务注册信息，发现位于其它公司的 Web 服务来促进自己的业务集成；服务使用者可以调用其需要的服务并应用到自己的架构和业务流程中。

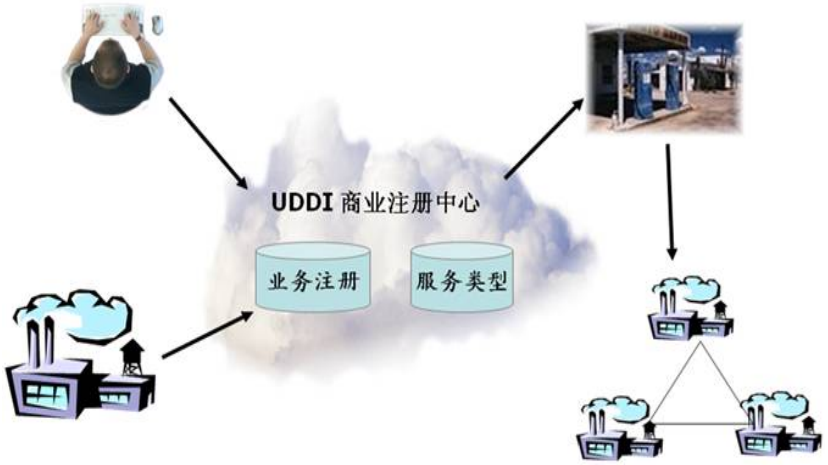


图 10-9 UDDI 商业注册中心运行环境

UDDI 提供了两组 API 供应用程序查询和发布 Web 服务信息。其提供的调用接口如表 10-1 所示。

表 10-1 UDDI 商业注册中心运行环境

Inquiry Operations		**Publication Operations**	
Find	find_business	Save	save_business
	find_service		save_service
	find_binding		save_binding
	find_tModel		save_tModel
	find_relatedBusinesses	Delete	delete_business
Get details	get_businessDetail		delete_service
	get_ serviceDetail		delete_binding
	get_ bindingDetail		delete_tModel
	get_ tModelDetail		delete_registeredInfo
	get_registeredInfo	Security	get_authToken
			discard_authToken

应用程序客户端通过 UDDI 提供的 API 并设定相关的参数信息访问 UDDI 注册中心，程序 10-13 是客户端向 UDDI 注册中心发送的请求消息。

程序 10-13 UDDI authToken 请求(*)

```
<get_authToken generic="2.0"
     xmlns="urn:uddi-org:api_v2"
```

```
        userID="businessA_UserId"
        cred="businessA_Password" />
```

(*) 本节后面的程序中均去除了 SOAP 协议包装信息

该请求消息中包含了用户名和密码，调用 get_anthToken 获得 UDDI 的使用授权。调用 UDDI 提供的 anthTokenUDDI 注册中心收到该消息后进行处理，得到的响应消息如程序 10-14 所示。

程序 10-14 UDDI authToken 响应

```
<authToken generic="2.0"
           operator="www.ibm.com/services/uddi"
           xmlns="urn:uddi-org:api_v2" >
   <authInfo>businessA_AuthToken</authInfo>
</authToken>
```

该响应消息中利用一个包含单个 authInfo 条目的 authToken 结构返回给消息使用者的授权信息，完成了服务请求者与 UDDI 商业注册中心的一次交互过程。

§ 10.5 WSFL/BPEL

WSFL 是用一个用来描述 Web 服务组合的 XML 语言，它是描述业务流程的一个相对成熟的标准。在 WSFL 中有两种不同的组合机制，分别利用流模型和全局模型来描述。其中，流模型描述了几个服务是怎样组合成一个流，以及流是如何转换成一个新的服务的。流模型描述了流程的数据流和控制流，并组织不同服务的执行顺序；而全局模型描述了组合服务中不同服务之间的交互。WSFL 利用活动、控制链、数据链和插入链四个基本元素来描述业务过程。

BPEL 是 IBM 的 WSFL 和 Microsoft 的 XLANG 相结合而产生的技术。它是一种基于 XML 的，用来描述业务过程的编程语言，被描述的业务过程的单一步骤则由 Web 服务来实现。BPEL4WS（Business Process Execution Language for Web Services）是 BPEL 的全称，是专门针对 Web 服务及其组合而制定的一项规范，在 2.0 版本中称为 WS-BPEL。在吸收和借鉴了 PetriNet 和 Pi-Calculus 优势的基础上，BPEL 成为一种高级的、抽象的、可执行建模语言，它不仅实现 Web 服务间的交互和流程编排，也能够将流程自身暴露为 Web 服务。BPEL 作为一种流程编排语言，必须通过 BPEL 执行引擎对其进行解析并调用相关 Web 服务来执行整个业务流程。

1. BPEL 基本结构

Web 服务的目标是提高业务功能的复用和自治能力。在利用 Web Services 实现的面向 SOA 的环境中，存在大量完成某种特定业务功能的 Web 服务，其中包括原子服务和组合服务。利用 BPEL 的业务流程编排能力，将这些 Web 服务有效地组织并组合起来完成企业所需的业务功能，并且形成更大的功能复用模块供其它企业所使用。

BPEL 的主要能力是对 Web 服务进行组织和编排，并提供不同 Web 服务之间的进行数据交换的机制，可以完成 Web 服务的调用、数据操作、故障处理和流程控制等活动。BPEL 业务流程的核心是一个<process>元素，其基本结构如程序 10-15 所示。

程序 10-15 BPEL 流程结构

```
<process>
   <partnerLinks> ... </partnerLinks>
   <variables> ... </variables>
   <correlationSets> ... </correlationSets>
   <faultHandlers> ... </faultHandlers>
   <compensationHandlers> ... </compensationHandlers>
   <eventHandlers> ... </eventHandlers>
   (activities)*
</process>
```

BPEL 基本结构中，包括以下几个元素：

- <partnerLinks>　描述两个服务之间的会话关系，其定义了会话中每个服务所扮演

的角色，并且制定了每个服务所提供给服务描述 WSDL 的入口类型（portType），以便接收会话的上下文中的消息。

- <variables> 业务流程指定了涉及伙伴之间消息交换的有状态交互。业务流程的状态不仅包括被交换的消息，还包括用于业务逻辑和构造发送给伙伴的消息的中间数据，这些消息和数据的通过所定义的变量来存储。
- <correlationSets> BPEL 提供了声明性机制，指定服务示例中相关联的操作组。一组相关标记可定义为相关联的组中所有消息共享的一组特性，即相关集。
- <faultHandlers> 故障处理是因发生故障而切换到撤销发生故障的作用域中的部分或不成功的工作。故障处理通过定义的 catch 活动拦截某种特定类型的故障。
- <compensationHandlers> 通过补偿处理程序，作用域可以描述一部分通过应用程序定义的方法可撤销的行为。有补偿处理程序的作用域可以不受约束地嵌套，其需要接收当前状态的数据并返回关于补偿结果的数据。
- <eventHandlers> 整个流程以及每个作用域都可以与一组在相应的事件发生时的并发调用事件处理程序相关联。其处理的事件包括 WSDL 的传入消息和用户设置的时间警报。

此外，BPEL 中还使用到了其它一些元素：

- <source> 标记活动作为链接源。
- <target> 标记一个活动作为链接目标。
- <delete> 删除已选择的活动或元素。
- <onMessage> 表明消息到达时所要执行的活动。
- <onAlarm> 表明时间到达时所要执行的活动。
- <partner> 定义服务交互的伙伴。
- <container> 提供组成业务流程状态的消息的存储方法。

BPEL 的业务流程本身就是一个流程图，类似于表达算法的执行过程。活动是 BPEL 业务流程的核心，通过 BPEL 规范中的一些基本活动和结构化活动将已有的一组 Web 服务组合为业务流程，创建新的 Web 服务。图 10-10 显示了一个 BPEL 的业务流程通过调用相关的 Web 服务来实现其功能，形成一个新的 Web 服务。利用端口类型中的输入输出操作完成与 Web 服务之间的数据交换

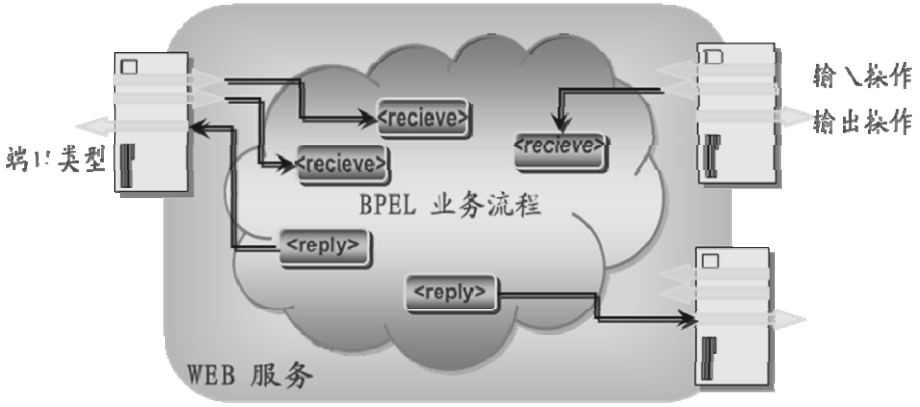


图 10-10 BPEL 业务流程

BPEL 中的活动包括基本活动和结构化活动。

基本活动定义了针对 Web 服务的操作和交互：调用（<invoke>）某个 Web 服务上的操作，等待一条消息来响应由某人从外部进行调用的服务接口的操作（<receive>），生成输入／输出操作的响应（<reply>），等待一段时间（<wait>），把数据从一个地方复制到另一个地方（<assign>），指明某个地方出错了（<throw>），终止整个服务实例（<terminate>），或者什么也不做（<empty>），还包括补偿处理活动（<compensate>）等。

通过使用语言所提供的任何结构化活动，可以将这些原语活动组合成更复杂的算法。这些结构化活动提供的能力有：定义一组步骤的有序序列（<sequence>），使用现在常见的“case-statement”办法来产生分支（<switch>），定义一个循环（<while>），执行几条可选路

径中的一条（<pick>），以及指明一组步骤应该并行地执行（<flow>）。在并行地执行的一组活动中，可以通过使用链接（link）来指明执行顺序方面的约束。

2. BPEL 业务流程示例

我们通过一个贷款批准流程的例子来说明 BPEL 如何描述一个业务流程。该流程中首先客户发出一个贷款请求，请求得到处理，然后客户弄清楚贷款是否得到了批准。初始时，客户向启用 Web 服务的金融机构发送申请并将决定告诉客户，该流程将使用他的申请并给他发送一个响应。

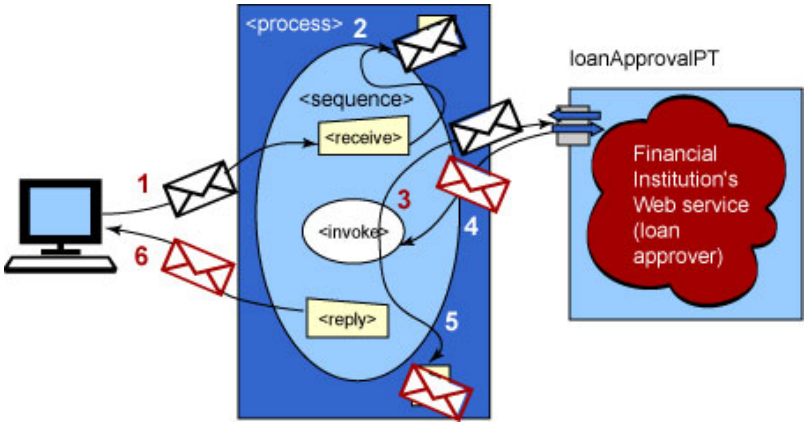


图 10-11 贷款批准流程

BPEL 需要引用被交换的消息、正在被调用的操作以及这些操作所属于的 portType，因此对其所涉及的 WSDL 有比较大的依赖性。我们假定已经创建了所需的 3 个 WSDL 文件及定义了其对应的命名空间：

- 贷款定义 WSDL（loandefinitions.wsdl）： http://tempuri.org/services/loandefinitions
- 贷款批准者 WSDL（loanapprover.wsdl）： http://tempuri.org/services/loanapprover
- 贷款批准 WSDL（loan-approval.wsdl）： http://tempuri.org/services/loan-approval

创建流程就是定义<process>元素的过程。loanApprovalProcess 流程的定义及其命名空间的引用如程序 10-16 所示。

程序 10-16 BPEL 流程定义

```
<process name="loanApprovalProcess"
         targetNamespace="http://acme.com/simpleloanprocessing"
         xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval"
         xmlns:loandef="http://tempuri.org/services/loandefinitions"
         xmlns:apns="http://tempuri.org/services/loanapprover">
```

我们本例中主要关注流程的建立，忽略了其它如补偿处理等元素的内容。下一步是声明涉及的各方 partner。如程序 10-17 所示，定义了已命名的伙伴，每个伙伴都有一个 WSDL serviceLinkType 表明其特征。对于这个示例来说，伙伴是客户和金融机构。伙伴的 myRole/partnerRole 属性指定了给定的 serviceLinkType 的伙伴和流程将如何交互。myRole 属性引用的是流程在 serviceLinkType 中将要扮演的角色，而 partnerRole 指定了伙伴将要扮演的角色。贷款批准流程为客户提供了 loanApprovalPT 功能，金融机构接着又为流程提供了这个功能。

程序 10-17 BPEL 流程伙伴定义

```
<partners>
  <partner name="customer"
           serviceLinkType="lns:loanApproveLinkType"
           myRole="approver"/>
  <partner name="approver"
           serviceLinkType="lns:loanApprovalLinkType"
           partnerRole="approver"/>
</partners>
```

为了申请贷款，客户向流程发送一条消息，流程询问金融机构是否将接受贷款申请，然后用另一条消息（要么是接受申请的消息，要么是拒绝申请的消息）应答客户。BPEL 流程需要将传入的消息放在 BPEL 活动可以访问到它的地方。在 BPEL 中，数据被写入到数据容器中并且从数据容器进行访问，这些数据容器可以保存特定的 WSDL 消息类型的实例。程序 10-18 定义了所需要使用的容器列表。

程序 10-18 BPEL 流程容器列表

```
<containers>
   <container name="request" messageType="loandef:CreditInformationMessage"/>
   <container name="approvalInfo"  messageType="apns:approvalMessage"/>
</containers>
```

本流程中仅包含了一个活动 sequence。程序 10-19 在 sequence 中定义的 receive 活动可以接收客户的消息并把它保存到适当的容器中.

程序 10-19 BPEL 流程接收消息

```
 <sequence>
   <receive name="receive1" partner="customer"
            portType="apns:loanApprovalPT"
            operation="approve" container="request"
            createInstance="yes">
   </receive>
```

询问启用 Web 服务的金融机构是否将接受贷款是用一个 Web 服务调用来完成的，该调用是在流程中由 invoke 活动定义。在程序 10-20 中 invoke 活动在运行时将使用其输入容器中的消息来调用指定的 Web 服务，并将得到的应答放入到输出容器中。

程序 10-20 BPEL 流程调用服务

```
   <invoke name="invokeapprover"
           partner="approver"
           portType="apns:loanApprovalPT"
           operation="approve"
           inputContainer="request"
           outputContainer="approvalInfo">
   </invoke>
```

为了使流程对客户的请求作出应答，流程使用一个 reply 活动。程序 10-21 中的 reply 活动想要告诉客户金融机构的决定是什么，所以在调用的输出容器中将会发现要被发送的消息：approvalInfo。

程序 10-21 BPEL 流程客户响应

```
   <reply name="reply" partner="customer" portType="apns:loanApprovalPT"
          operation="approve" container="approvalInfo">
   </reply>
 </sequence>
</process>
```

思考与练习

10-1 Web Service 与 CORBA 对象或 EJB 构件实现的服务有什么区别？

10-2 在基于 Web Service 实现的 SOA 架构中，如何结合 SOAP，WSDL，UDDI 和 BPEL 完成服务的描述、查找、交互和流程的编排？

10-3 理解了 Web Service 架构之上，分析其在工程应用的优势和不足。

附录 1　OMG IDL 语法规则

OMG IDL 具有与 C++非常类似的语法风格，它遵循 ANSI C++的词法规则，并且支持标准 C++预处理。本附录给出 IDL 语法规则的形式化定义，这些定义源自[CORBA 2.4]的第 3 章。

下列扩展 BNF 所使用的元符号定义为：终结符号为加黑的单词或符号，非终结符号为未加黑的单词；“::=”表示定义为；“A | B”表示任选语法单位 A 或 B，A^*表示语法单位 A 可重复出现 0 次或多次，A^+表示语法单位 A 可重复出现 1 次或多次，[A]表示语法单位 A 出现 0 次或 1 次，{ … }表示将多个语法单位作为一个新的语法单位。

```
(1) specification     ::= definition+
(2) definition        ::= type_dcl ; | const_dcl ; | except_dcl ; | interface ; | module ; | value ;
(3) module            ::= module identifier { definition+ }
(4) interface         ::= interface_dcl | forward_dcl
(5) interface_dcl     ::= interface_header { interface_body }
(6) forward_dcl       ::= [abstract | local] interface identifier
(7) interface_header  ::= [abstract | local] interface identifier [interface_inheritance_spec]
(8) interface_body    ::= export*
(9) export            ::= type_dcl ; | const_dcl ; | except_dcl ; | attr_dcl ; | op_dcl ;
(10) interface_inheritance_spec ::= : interface_name {, interface_name}*
(11) interface_name   ::= scoped_name
(12) scoped_name      ::= identifier | :: identifier | scoped_name :: identifier
(13) value            ::= value_dcl | value_abs_dcl | value_box_dcl | value_forward_dcl
(14) value_forward_dcl ::= [abstract] valuetype identifier
(15) value_box_dcl    ::= valuetype identifier type_spec
(16) value_abs_dcl    ::= abstract valuetype identifier [value_inheritance_spec] { export* }
(17) value_dcl        ::= value_header { value_element* }
(18) value_header     ::= [custom] valuetype identifier [value_inheritance_spec]
(19) value_inheritance_spec ::= [: [truncatable] value_name {, value_name}*]
                                [supports interface_name {, interface_name}*]
(20) value_name       ::= scoped_name
(21) value_element    ::= export | state_member | init_dcl
(22) state_member     ::= {public | private} type_spec declarators ;
(23) init_dcl         ::= factory identifier ( [init_param_decls] ) ;
(24) init_param_decls ::= init_param_decl {, init_param_decl}*
(25) init_param_decl  ::= init_param_attribute param_type_spec simple_declarator
(26) init_param_attribute ::= in
(27) const_dcl        ::= const const_type identifier = const_exp
(28) const_type       ::= integer_type | char_type | wide_char_type | boolean_type
                      |   floating_pt_type | string_type | wide_string_type
                      |   fixed_pt_const_type | scoped_name | octet_type
(29) const_exp        ::= or_expr
(30) or_expr          ::= xor_expr | or_expr | xor_expr
(31) xor_expr         ::= and_expr | xor_expr ^ and_expr
(32) and_expr         ::= shift_expr | and_expr & shift_expr
(33) shift_expr       ::= add_expr | shift_expr >> add_expr | shift_expr << add_expr
(34) add_expr         ::= mult_expr | add_expr + mult_expr | add_expr - mult_expr
(35) mult_expr        ::= unary_expr | mult_expr * unary_expr
                      |   mult_expr / unary_expr | mult_expr % unary_expr
(36) unary_expr       ::= unary_operator primary_expr | primary_expr
(37) unary_operator   ::= - | + | ~
(38) primary_expr     ::= scoped_name | literal | ( const_exp )
(39) literal          ::= integer_literal | string_literal | wide_string_literal
```

```
                    |   character_literal | wide_character_literal
                    |   fixed_pt_literal | floating_pt_literal | boolean_literal
(40) boolean_literal  ::= TRUE | FALSE
(41) positive_int_const ::= const_exp
(42) type_dcl         ::= typedef type_declarator | struct_type | union_type
                    |   enum_type | native simple_declarator | constr_forward_decl
(43) type_declarator  ::= type_spec declarators
(44) type_spec        ::= simple_type_spec | constr_type_spec
(45) simple_type_spec ::= base_type_spec | template_type_spec | scoped_name
(46) base_type_spec   ::= floating_pt_type | integer_type | char_type | wide_char_type
                    |   boolean_type | octet_type | any_type | object_type | value_base_type
(47) template_type_spec ::= sequence_type | string_type | wide_string_type | fixed_pt_type
(48) constr_type_spec ::= struct_type | union_type | enum_type
(49) declarators      ::= declarator {, declarator}*
(50) declarator       ::= simple_declarator | complex_declarator
(51) simple_declarator ::= identifier
(52) complex_declarator ::= array_declarator
(53) floating_pt_type ::= float | double | long double
(54) integer_type     ::= signed_int | unsigned_int
(55) signed_int       ::= signed_short_int | signed_long_int | signed_longlong_int
(56) signed_short_int ::= short
(57) signed_long_int  ::= long
(58) signed_longlong_int ::= long long
(59) unsigned_int     ::= unsigned_short_int | unsigned_long_int | unsigned_longlong_int
(60) unsigned_short_int ::= unsigned short
(61) unsigned_long_int  ::= unsigned long
(62) unsigned_longlong_int ::= unsigned long long
(63) char_type        ::= char
(64) wide_char_type   ::= wchar
(65) boolean_type     ::= boolean
(66) octet_type       ::= octet
(67) any_type         ::= any
(68) object_type      ::= Object
(69) struct_type      ::= struct identifier { member_list }
(70) member_list      ::= member+
(71) member           ::= type_spec declarators ;
(72) union_type       ::= union identifier switch ( switch_type_spec ) { switch_body }
(73) switch_type_spec ::= integer_type | char_type | boolean_type | enum_type | scoped_name
(74) switch_body      ::= case+
(75) case             ::= case_label+ element_spec ;
(76) case_label       ::= case const_exp : | default :
(77) element_spec     ::= type_spec declarator
(78) enum_type        ::= enum identifier { enumerator {, enumerator}* }
(79) enumerator       ::= identifier
(80) sequence_type    ::= sequence < simple_type_spec , positive_int_const >
                    |   sequence < simple_type_spec >
(81) string_type      ::= string < positive_int_const > | string
(82) wide_string_type ::= wstring < positive_int_const > | wstring
(83) array_declarator ::= identifier fixed_array_size+
(84) fixed_array_size ::= [ positive_int_const ]
(85) attr_dcl         ::= [readonly] attribute param_type_spec
                          simple_declarator {, simple_declarator}*
(86) except_dcl       ::= exception identifier { member* }
(87) op_dcl           ::= [op_attribute] op_type_spec identifier
                          parameter_dcls [raises_expr] [context_expr]
(88) op_attribute     ::= oneway
(89) op_type_spec     ::= param_type_spec | void
```

```
(90) parameter_dcls   ::= ( param_dcl {, param_dcl}* ) | ( )
(91) param_dcl        ::= param_attribute param_type_spec simple_declarator
(92) param_attribute  ::= in | out | inout
(93) raises_expr      ::= raises ( scoped_name {, scoped_name}* )
(94) context_expr     ::= context ( string_literal {, string_literal}* )
(95) param_type_spec  ::= base_type_spec | string_type | wide_string_type | scoped_name
(96) fixed_pt_type    ::= fixed < positive_int_const , positive_int_const >
(97) fixed_pt_const_type::= fixed
(98) value_base_type  ::= ValueBase
(99) constr_forward_decl::= struct identifier | union identifier
```

附录 2　银行应用源代码

源程序清单：

- 程序附 2-1：会话构件 AccountManager 的 Remote 接口定义；
- 程序附 2-2：会话构件 AccountManager 的 Home 接口定义；
- 程序附 2-3：会话构件 AccountManager 的 Enterprise Bean 类定义；
- 程序附 2-4：实体构件 AccountBean 的 Remote 接口定义；
- 程序附 2-5：实体构件 AccountBean 的 Home 接口定义；
- 程序附 2-6：实体构件 AccountBean 的 Enterprise Bean 类定义；
- 程序附 2-7：实体构件 LogBean 的 Remote 接口定义；
- 程序附 2-8：实体构件 LogBean 的 Home 接口定义；
- 程序附 2-9：实体构件 LogBean 的 Enterprise Bean 类定义；
- 程序附 2-10：Servlet 构件 BankServlet 的类定义；
- 程序附 2-11：辅助异常 AccountAccessDeniedException 的类定义；
- 程序附 2-12：辅助异常 InsufficientFundsException 的类定义；
- 程序附 2-13：辅助异常 NoAccountCreatedException 的类定义；
- 程序附 2-14：辅助异常 NoSuchAccountException 的类定义。

程序附 2-1　会话构件 AccountManager 的 Remote 接口定义

```
package banking;
import javax.ejb.*;
import java.rmi.*;
public interface AccountManager extends EJBObject
{
    public void deposit(String customerName, double amount)
                        throws RemoteException;
    public double withdraw(String customerName, double amount)
                        throws RemoteException, InsufficientFundsException;
    public double getBalance(String customerName) throws RemoteException;
    public double calculateInterest(String customerName) throws RemoteException;
    public Account createAccount(String customerName, double initialBalance)
                        throws NoAccountCreatedException, RemoteException;
    public void removeAccount(String customerName)
                        throws NoAccountCreatedException, RemoteException;
}
```

程序附 2-2　会话构件 AccountManager 的 Home 接口定义

```
package banking;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface AccountManagerHome extends EJBHome
{
    public AccountManager create() throws RemoteException, CreateException;
}
```

程序附 2-3　会话构件 AccountManager 的 Enterprise Bean 类定义

```
package banking;

import javax.ejb.*;
import javax.naming.*;
import java.util.*;
import javax.rmi.*;
import java.rmi.RemoteException;
```

```
public class AccountManagerBean implements SessionBean
{
    private SessionContext context;
    public static final String DEPOSIT = "deposit funds";
    public static final String WITHDRAW = "withdraw funds";
    public static final String GETBALANCE = "get an account balance";
    public static final String CALCULATEINTEREST = "calculate interest on an
account";
    public static final String CREATEACCOUNT = "create an account";
    public static final String CLIENT = "Client";
    public static final String BANK = "Bank";

    public AccountManagerBean() {}
    public void ejbRemove() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void setSessionContext(SessionContext ctx) {
        context = ctx;
    }
    public void ejbCreate() {}

    //business method
    public void deposit(String customerName, double amount){
        logActivity(DEPOSIT, customerName);
        try{
            Account account = getAccount(customerName);
            account.deposit(amount);
        }catch(NoSuchAccountException e){
            throw new EJBException();
        }catch(RemoteException ex){
            throw new EJBException();
        }
    }

    public double withdraw(String customerName, double amount)
            throws InsufficientFundsException{
        double newBal;
        logActivity(WITHDRAW, customerName);
        try{
            Account account = getAccount(customerName);
            double bal = account.getBalance();
            if(amount > bal){
                throw new InsufficientFundsException("Account does not have " +
amount);
            }
            newBal = account.withdraw(amount);
        }catch(NoSuchAccountException e){
            throw new EJBException();
        }catch(RemoteException ex){
            throw new EJBException();
        }
        return newBal;
    }

    public double getBalance(String customerName){
        System.out.println("Entry getBalance");
        logActivity(GETBALANCE, customerName);
        System.out.println("after logActivity");
        double bal = 0;
        try{
            Account account = getAccount(customerName);
            System.out.println("after getAccount");
```

```
            bal = account.getBalance();
            System.out.println("after getBalance");
//      }catch(AccountAccessDeniedException AE){
//          throw new AccountAccessDeniedException("Access Denied");
        }catch(NoSuchAccountException e){
            throw new EJBException();
        }catch(RemoteException ex){
            throw new EJBException();
        }
        return bal;
    }
    public double calculateInterest(String customerName){
        logActivity(CALCULATEINTEREST, customerName);
        try{
            Account account = getAccount(customerName);
            return account.calculateInterest();
        }catch(NoSuchAccountException e){
            throw new EJBException();
        }catch(RemoteException ex){
            throw new EJBException();
        }
    }
    public Account createAccount(String customerName, double initialBalance)
throws NoAccountCreatedException{
        logActivity(CREATEACCOUNT, customerName);
        try{
            AccountHome accountHome = getAccountHome();
            return accountHome.create(customerName, initialBalance);
        }catch(CreateException ce){
            throw new NoAccountCreatedException(ce.getMessage());
        }catch(RemoteException ex){
            throw new EJBException();
        }
    }

    public      void      removeAccount(String      customerName)      throws
NoAccountCreatedException{
        try{
            Account account = getAccount(customerName);
            account.remove();
        }catch(NoSuchAccountException e){
            throw new EJBException();
        }catch(RemoveException re){
            throw new EJBException();
        }catch(RemoteException ex){
            throw new EJBException();
        }
    }

    private void logActivity(String method, String customer){
        this.log(this.context.getCallerPrincipal().getName() + "tried to " +
            method + " for " + customer);
    }

    private      Account      getAccount(String      customerName)      throws
NoSuchAccountException{
        try{
            System.out.println("Entry getAccount");
            AccountHome accountHome = getAccountHome();
            System.out.println("after getAccountHome");
            return accountHome.findByPrimaryKey(new String(customerName));
        }catch(RemoteException ex){
```

```
            throw new EJBException();
        }catch(FinderException fe){
            throw new NoSuchAccountException("No such account");
        }
    }

    private AccountHome getAccountHome(){
        try{
            InitialContext initial = new InitialContext();
            Object objref = initial.lookup("java:comp/env/ejb/account");
            AccountHome                          home                          =
(AccountHome)javax.rmi.PortableRemoteObject.narrow(objref,
AccountHome.class);
            return home;
        }catch(NamingException ne){
            throw new EJBException();
        }
    }

    private void log(String msg){
        try{
            InitialContext ctx = new InitialContext();
            Object ref = ctx.lookup("java:comp/env/ejb/log");
            LogHome  h  =  (LogHome)javax.rmi.PortableRemoteObject.narrow(ref,
LogHome.class);
            h.create(msg);
        }catch(Exception e){}
    }
}
```

程序附 2-4　实体构件 AccountBean 的 Remote 接口定义

```
package banking;
import javax.ejb.*;
import java.rmi.*;
public interface Account extends EJBObject
{
    public double  getBalance() throws RemoteException;
    public void deposit(double amount) throws RemoteException;
    public double withdraw(double amount)
                        throws RemoteException, InsufficientFundsException;
    public double calculateInterest() throws RemoteException;
}
```

程序附 2-5　实体构件 AccountBean 的 Home 接口定义

```
package banking;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface AccountHome extends EJBHome
{
    public Account create(String customerName, double currentBalance)
        throws RemoteException, CreateException;
    public Account findByPrimaryKey(String key)
        throws RemoteException, FinderException;
}
```

程序附 2-6　实体构件 AccountBean 的 Enterprise Bean 类定义

```
package banking;

import javax.ejb.*;
import javax.naming.*;
```

```
public abstract class AccountBean implements EntityBean{
    private EntityContext context;
    private static final double INTEREST = 0.01;
    public AccountBean() {}
    //CMP fields
    public abstract double getCurrentBalance();
    public abstract void setCurrentBalance(double orderID);

    public abstract String getCustomerName();
    public abstract void setCustomerName(String customerName);

    //business method
    public double getBalance(){
        return getCurrentBalance();
    }

    public void deposit(double amount) {
        setCurrentBalance(getCurrentBalance() + amount);
    }
    public double withdraw(double amount) throws InsufficientFundsException {
        if(getCurrentBalance() < amount){
            throw new InsufficientFundsException("Account does not have " +
amount);
        }else{
            setCurrentBalance(getCurrentBalance() - amount);
            return getCurrentBalance();
        }
    }
    public double calculateInterest() {
        setCurrentBalance(getCurrentBalance()   +   (getCurrentBalance()   *
INTEREST));
        return getCurrentBalance();
    }

    public String ejbCreate(String customerName, double currentBalance)
            throws CreateException{
        setCustomerName(customerName);
        setCurrentBalance(currentBalance);
        return null;
    }

    public void ejbPostCreate(String customerName, double currentBalance) {}
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }
    public void unsetEntityContext() {
        context = null;
    }
}
```

程序附 2-7　实体构件 LogBean 的 Remote 接口定义

```
package banking;
import javax.ejb.*;
import java.rmi.*;
public interface Log extends EJBObject
{
```

```
    public void log(String msg) throws RemoteException;
}
```

程序附 2-8　实体构件 LogBean 的 Home 接口定义

```
package banking;

import java.rmi.RemoteException;
import javax.ejb.*;

public interface LogHome extends EJBHome
{
    public Log create(String msg)
        throws RemoteException, CreateException;
    public Log findByPrimaryKey(String key)
        throws RemoteException, FinderException;
}
```

程序附 2-9　实体构件 LogBean 的 Enterprise Bean 类定义

```
package banking;

import javax.ejb.*;
import javax.naming.*;

public abstract class LogBean implements EntityBean{
    private EntityContext context;
    public LogBean() {}
    //CMP fields
    public abstract String getLogMessage();
    public abstract void setLogMessage(String logMessage);

    //business method
    public String ejbCreate(String msg)
            throws CreateException{
        setLogMessage(msg);
        return null;
    }

    public void log(String msg){
        setLogMessage(msg);
    }

    public void ejbPostCreate(String msg) {}
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }
    public void unsetEntityContext() {
        context = null;
    }
}
```

程序附 2-10　Servlet 构件 BankServlet 的类定义

```
package bankWeb;

import banking.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class BankServlet extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException{
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<HTML>");
        out.println("<HEAD><TITLE>BankServlet</TITLE></HEAD>");
        out.println("<BODY>");
        try{
            Context initial = new InitialContext();
            AccountManagerHome accountManagerHome = (AccountManagerHome)
                        initial.lookup("java:comp/env/ejb/AccountManager");
            AccountManager accountManager = accountManagerHome.create();
            out.println("get name1's balance...<br>");
            out.println("name1's balance is: " +
                                accountManager.getBalance("name1") + "<br>");
            Account account = accountManager.createAccount(
                                "newAccount", 1000.00f);
            out.println("createAccount newAccount successfully<br>");
            accountManager.removeAccount("newAccount");
            out.println("removeAccount successfully<br>");
        }catch(RemoteException re){
            out.println("RemoteException : " + re.getMessage());
        }catch(NoAccountCreatedException nace){
            out.println("NoAccountCreatedException : " + nace.getMessage());
        }catch(CreateException ce){
            out.println("CreateException : " + ce.getMessage());
        }catch(Exception e){
            out.println("Get Exception: " + e.getMessage());
        }
        out.println("</BODY>");
        out.println("</HTML>");
        out.close();
    }
}
```

程序附 2-11　辅助异常 AccountAccessDeniedException 的类定义

```
package banking;
public class AccountAccessDeniedException extends Exception{
    public AccountAccessDeniedException(String msg){
        super(msg);
    }
}
```

程序附 2-12　辅助异常 InsufficientFundsException 的类定义

```
package banking;
public class InsufficientFundsException extends Exception{
    public InsufficientFundsException(String msg){
        super(msg);
    }
}
```

程序附 2-13　辅助异常 NoAccountCreatedException 的类定义

```
package banking;
public class NoAccountCreatedException extends Exception{
    public NoAccountCreatedException(String msg){
        super(msg);
```

```
    }
}
```

程序附 2-14　辅助异常 NoSuchAccountException 的类定义

```
package banking;
public class NoSuchAccountException extends Exception{
    public NoSuchAccountException(String msg){
        super(msg);
    }
}
```

主要参考文献

[Baker 1997] S. Baker, CORBA Distributed Objects Using Orbix, Addison-Wesley, 1997
[Ben-Natan 1998] R. Ben-Natan, CORBA on the Web, McGraw-Hill, 1998
[Booch 1994] G. Booch, Object-Oriented Analysis and Design with Applications, 2nd Edition, Benjamin/Cummings, 1994
[Booch&Rambaugh 1996] G. Booch and J. Rambaugh, Unified Method for Object-Oriented Development, Rational Software, 1996
[Boucher&Katz 1999] K. Boucher and F. Katz, Essential Guide to Object Monitors, John Wiley & Sons, 1999
[Byte 1995] Byte, ?(4), 1995
[CORBA 2.0] CORBA: Common Object Request Broker Architecture and Specification, Revision 2.0, Object Management Group, 1995 (formal/97-02-25)
[CORBA 2.3] CORBA: Common Object Request Broker Architecture and Specification, Revision 2.3.1, Object Management Group, 1998 (formal/98-12-01) （韦乐平等编译，《CORBA 系统结构、原理与规范》北京：电子工业出版社，2000）
[CORBA 2.4] CORBA: Common Object Request Broker Architecture and Specification, Revision 2.4.2, Object Management Group, 2000 (formal/01-02-01)
[CORBA Business T&S] Task and Session CBOs Specification, Version 1.0 New Edition, Object Management Group, 2000
[CORBA Business WFM] Workflow Management Facility Specification, Version 1.2 New Edition, Object Management Group, 2000
[CORBA Finance Currency] Currency Specification, Version 1.0 New Edition, Object Management Group, 2000
[CORBA Manufacturing DSS] Distributed Simulation Systems Specification, Version 1.1, Object Management Group, 2000 (formal/2000-12-01)
[CORBA Manufacturing PDM] Product Data Management Enablers Specification, Version 1.3 New Edition, Object Management Group, 2000 (formal/2000-11-11)
[CORBA Healthcare PIS] Person Identification Service Specification, Version 1.0, Object Management Group, 2000 (formal/2000-06-30)
[CORBA Healthcare LQS] Lexicon Query Service Specification, Version 1.0, Object Management Group, 2000 (formal/2000-06-31)
[CORBA Telecoms AVS] Audio/Video Streams, Version 1.0, Object Management Group, 2000 (formal/2000-01-03)
[CORBA Telecoms TC&SCCP] CORBA TC Interworking and SCCP Inter - ORB Protocol Specification, Version 1.0, Object Management Group, 2000 (formal/2001-01-01)
[CORBA Telecoms TMN] CORBA/TMN Interworking Specification, Version 1.0, Object Management Group, 2000 (formal/2000-08-01)
[CORBA Telecoms TLS] Telecoms Log Service Specification, Version 1.0, Object Management Group, 2000 (formal/2000-01-04)
[CORBA Transportation ATC] Air Traffic Control Specification, Object Management Group, 2000 (formal/2000-05-01)
[CORBAfacilities I&T] Internationalization, Time Operations, and Related Facilities, New Edition, Object Management Group, 2000
[CORBAfacilities MAF] Mobile Agent Facility Specification, New Edition, Object Management Group, 2000
[COSS Collection] Object Collection Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Concurrency] Concurrency Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Event] Event Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Externalization] Externalization Service Specification, Version 1.0 New Edition, Object Management Group, 2000

[COSS Licensing] Licensing Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Lifecycle] Life Cycle Service Specification, Version 1.1 New Edition, Object Management Group, 2000
[COSS Naming] Interoperable Naming Service Specification, New Edition, Object Management Group, 2000
[COSS Notification] Notification Service Specification, Version 1.0 New Edition, Object Management Group, 2000 (formal/2000-06-20)
Persistent State Service (orbos/99-07-07)
[COSS Property] Property Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Query] Query Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Relationship] Relationship Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Security] Security Services Specification, Version 1.5, Object Management Group, 2000
[COSS Time] Time Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Trading] Trading Object Service Specification, Version 1.0 New Edition, Object Management Group, 2000
[COSS Transaction] Transaction Service Specification, Version 1.1 New Edition, Object Management Group, 2000
[Ding&Sun 1997] 丁锂、孙元，《Java 语言 SQL 接口：JDBC 编程技术》，北京：清华大学出版社，1997
[Edwards 1999] W. Edwards, Core Jini, Prentice Hall PTR, 1999 （王召福等译，《Jini 核心技术》，北京：机械工业出版社，2000）
[Hamilton&Cattell 1996] G. Hamilton and R. Cattell, JDBC: A Java SQL API, Sun Microsystems, 1996
[Hamilton&Cattell&Fisher 1997] G. Hamilton, R. Cattell and M. Fisher, JDBC Database Access with Java: A Tutorial and Annotated Reference, Addison-Wesley, 1997
[Henning&Vinoski 1999] M. Henning and S. Vinoski, Advanced CORBA Programming with C++, Addison-Wesley, 1999
[IDL2Ada] Ada Language Mapping Specification, New Edition, Object Management Group, 1999
[IDL2C] C Language Mapping Specification, New Edition, Object Management Group, 1999
[IDL2C++] C++ Language Mapping Specification, New Edition, Object Management Group, 1999
[IDL2Java] IDL to Java Language Mapping Specification, New Edition, Object Management Group, 1999
[IDL2Smalltalk] Smalltalk Language Mapping Specification, New Edition, Object Management Group, 1999
[Inprise 2000] VisiBroker for Java: Programmer's Guide and Reference, Inprise Corporation, 2000 （李文军等译，《VisiBroker for Java 开发者指南》，北京：机械工业出版社，2000）
[Jacobson 1992] I. Jacobson, Object-Oriented Software Engineering, Addison-Wesley, 1992
[Java2IDL] Java Language to IDL Mapping Specification, New Edition, Object Management Group, 1999
[Jepson 1997] B. Jepson, Java Database Programming, John Wiley & Sons, 1997 （钱毅等译，《Java 数据库编程指南》，北京：电子工业出版社，1998）
[Meyer 1995] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, 1995
[OMA 1990] Object Management Architecture Guide , Object Management Group, 1990
[OMA 1997] A Discussion of the Object Management Architecture, Object Management Group, 1997 (formal/00-06-41)
[Orfali&Harkey&Edwards 1994] R. Orfali, D. Harkey and J. Edwards, The Essential Client/Server Survival Guide, John Wiley & Sons, 1994
[Orfali&Harkey 1997] R. Orfali and D. Harkey, Client/Server Programming With Java and

CORBA, John Wiley & Sons, 1997

[Otte&Patrick&Roy 1998] R. Otte, P. Patrick and M. Roy, Understanding CORBA: the Common Object Request Broker Architecture, Prentice-Hall PTR, 1998 （李师贤等译，《CORBA 教程：公共对象请求代理体系结构》，北京：清华大学出版社，1999）

[Rambaugh 1991] J. Rambaugh, et al., Object-Oriented Modeling and Design, Prentice-Hall, 1991

[Pressman 1997] R. Pressman, Software Engineering: A Practitioner's Approach, 4th Edition, McGraw-Hill, 1997 （黄柏素、梅宏译，《软件工程－实践者的研究方法》，北京：机械工业出版社，1999）

[Salemi 1995] J. Salemi, Guide to Client/Server Database, 2nd Edition, Ziff-Davis Press, 1995 （秦箕英译，《客户／服务器数据库指南》，北京：电子工业出版社，1995）

[Schmidt 2001] D. Schmidt, Developing Distributed Object Computing Applications with CORBA, CORBA Tutorial, http://www.eng.uci.edu/~schmidt/

[Wallnau 2000] K. Wallnau, Common Object Request Broker Architecture: Software Technology Review, Software Engineering Institute, Carnegie Mellon University, 2000, http://www.sei.cmu.edu/str/descriptions/corba_body.html

[Shaw&Garlan 1996] M. Shaw and D. Garlan, Software Architecture: Perspective on an Emerging Discipline, Prentice-Hall International, 1996

[Siegel 1998] J. Siegel, CORBA Enterprise Model, Communications of ACM, 1998, 41(10), ???-???

[Slama&Garbis&Russell 1999] D. Slama, J. Garbis and P. Russell, Enterprise CORBA, Prentice Hall, 1999 （李师贤等译校，《CORBA 企业解决方案》，北京：机械工业出版社，2001）

[Sun 1996a] Java Remote Method Invocation Specification, Sun Microsystems, 1996

[Sun 1996b] Java Object Serialization Specification, Sun Microsystems, 1996

[Sun 1996c] JDBC Guide: Getting Started, Sun Microsystems, 1996

[Sun 1996d] Java RMI Tutorial, Sun Microsystems, 1996

[UML 1.3] OMG Unified Modeling Language Specification, Version 1.3, Object Management Group, 2000

[UML MOF] Meta Object Facility (MOF) Specification, Version 1.3 New Edition, Object Management Group, 2000

[UML XMI] OMG XML Metadata Interchange (XMI) Specification, Version 1.1, Object Management Group, 2000

[W3C SOAP] W3C Simple Object Access Protocol（SOAP）Specification，Version1.2，W3C，2007

[W3C WSDL] W3C Web Services Description Language（WSDL）Specification，Version2.0，W3C，2007

[OASIS UDDI] OASIS Universal Description Discovery & Integration（UDDI）Specification，Version3.0，OASIS，2004

[OASIS BPEL] OASIS Web Service Business Process Execution Language（WSBPEL）Specification，Version2.0，OASIS，2007

[Vinoski 1997] S. Vinoski, CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. IEEE Communication Magazine, 1997, 35(2), 45-55

[Vinoski 1998] S. Vinoski, New Features for CORBA 3.0, Communications of ACM, 1998, 41(10)

[Vogel&Duddy] A. Vogel and K. Duddy, Java Programming with CORBA, John Wiley & Sons, 1998

[Wang 1999] 汪芸，《CORBA 技术及其应用》，南京：东南大学出版社，1999

[Xia&Yan 2000] 夏承刚、严隽永，CORBA 3.0 新特性分析，计算机科学，2000，27(3)，11-14