Software Architecture

- principle and practice

第六讲　CORBA技术及应用实例

分布式体系结构是当今大型应用软件产品的主流，而开发这类应用的最广泛的技术就是CORBA，CORBA为可移植的、面向对象的分布式计算应用程序提供了不依赖于平台的编程接口和模型，它不依赖于编程语言、计算平台、网络协议的这一特点，使得它非常适合于现有的分布式系统的新应用程序开发和系统集成。这一节我们给大家介绍分布式系统的有关概念和CORBA技术，下一讲介绍CORBA技术的代表BES的应用。

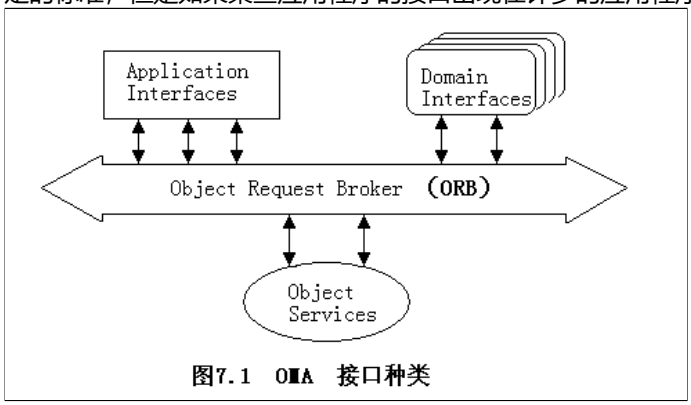
- 一、CORBA概述
  - CORBA及其发展过程
    - 解决分布式系统的应用程序开发问题的两条规则
      - 寻求独立于平台的模型和抽象
      - 在不牺牲太多性能的前提下，尽可能隐藏低层的复杂细节
      - 这两条规则不仅用于分布式系统，对于开发任何一个可移植的应用程序都是是适用的。使用合适的抽象和模型建立一个能提供异构的应用程序开发层(分布计算环境），系统异构的复杂性集中在这个层次，在这一层次上，低层的细节被隐藏起来，并且允许应用程序的开发人员只解决他们自己的开发问题，而不必面对需求所涉及到的、由于不同计算机平台所带来的低层的网络细节。
    - 对象管理组(OMG)
      - 对象管理组(Object Management Group),是一个非赢利的组织，建立于1989年4月，总部在美国，起由11个公司参与组建，现在拥有800多个成员，目前它是世界上最大的软件团体，其目标就是解决异构系统的可移植、分布式应用程序的开发问题、制定的技术对一些具体的问题作了合理高层抽象，并隐藏低层细节。
      - OMG使用两个相关的模型来描述如何与平台无关的分布式体系结构
        - 对象模型(Object Model)
          - 用来定义在一个异构环境中，如何描述分布式对象接口，它将对象定义为永恒不变的、始终是唯一的、被封装过的实体，这些实体只能被严格定义的接口(interface)访问，即客户机通过向对象发请求，才能使用对象的服务，对象的实现细节和它的位置对客户中隐藏的。
        - 引用模型(Reference Model)
          - 用来说明对象间如何交互。它提供一组服务接口。
          - 对象服务接口(Object Services,OS)，这是一组与领域无关的接口，这些对象服务允许应用和谐查找和发现对象引用（ Object Reference),被认为是构造分布式计算环境的核心部分，常见的有命名服务（Name Service)、交易服务(Trading Service)、事件服务(Event Service)等。
          - 领域接口(Domain Interface)，起着与对象服务种类相似的作用，对象服务接口是与领域无关的水平定向接口，只是领域接口针对领域而已，它与领域有关垂直定向接口，允许对象引用跨越不同的网络。
          - 应用程序接口(Application Interface)，是专门为特定的应用程序而开发的，它们并不是OMG所制定的标准，但是如果某些应用程序的接口出现在许多的应用程序中，
          - 

图7.1　OMA　接口种类
    - CORBA - - Common Object Request Broker Architecture):公共对象请求代理的体系结构，第一版于1991年问世，当时只规范了如何在C语言中使用它，1994年推出CORBA2.0规范，目前普遍使用的是CORBA2.X，现在CORBA3规范已经建立，它主要新增了Java和internet、服务质量控制以及CORBA组件包等方面。
    - CORBA简化了C/S模式
      - 在传统的client/server应用中，开发者使用自己设计的标准或通用标准来的协议（如Socket)。协议定义与实现的语言、网络传输及其w他网络因素有关。而CORBA简化了这一过程，它使用IDL来定义客户与服务器之间的接口协议。
      - CORBA客户端的工作

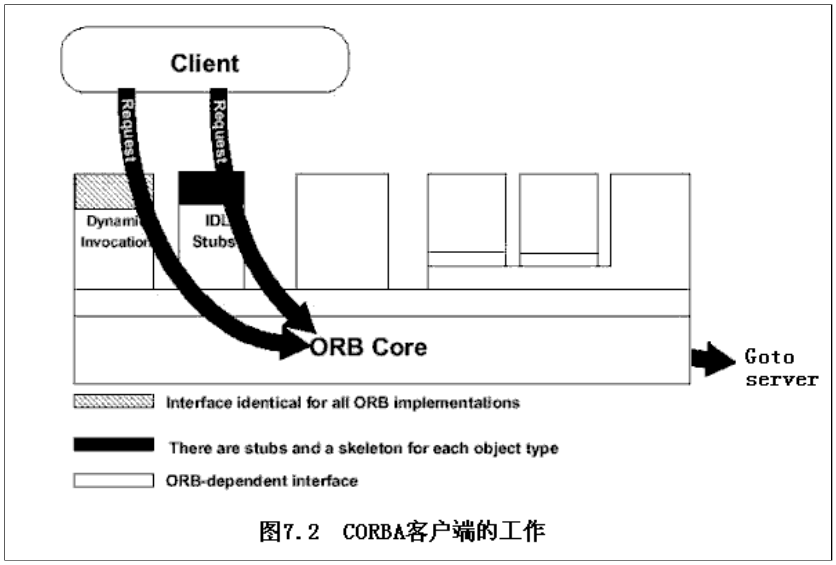


图7.2 CORBA客户端的工作

- CORBA服务器端的工作

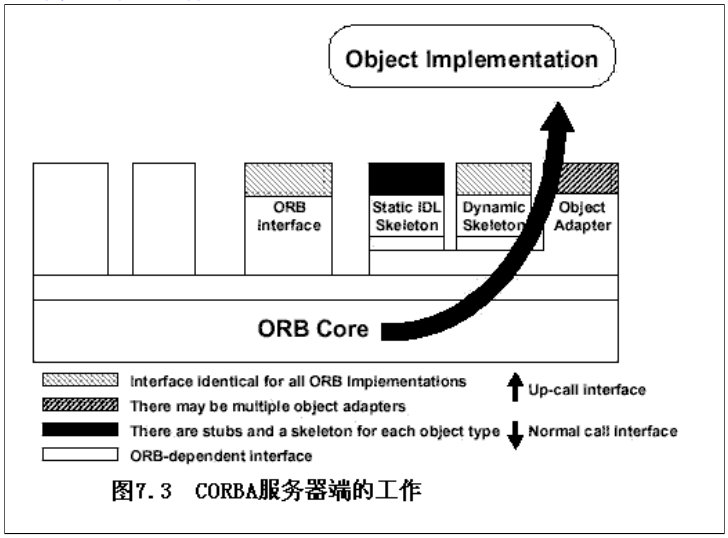


图7.3 CORBA服务器端的工作

二、CORBA特性

- 1.OMG接口定义语言（IDL）
  - 为了调用一个分布式对象的操作，客户程序必须了解由这个对象所提供的接口，一个对象的接口是由它所支持的操作和能够来回传输这些操作的数据类型所组成的。
  - 在CORBA中，对象接口是按OMG接口定义语言（Interface Define Language,IDL)来定义的，与C＋＋、JAVA等高级语言不同，IDL不是编程语言，所以对象和应用程序不能用IDL实现，IDL在客户和服务器程序之间建立一个契约，用它来描述在应用程序中需要用到的类型和对象接口，这些描述与实现的语言无关，可以不用考虑客户程序的编程语言是否与服务器程序的编程语言一致。正是IDL的语言独立性，使CORBA成为异构系统的集成技术。
  - IDL是一个纯说明性的语言，它使用程序员把焦点集中在对象接口、接口所支持的操作和操作时可能引发的异常上。它有一整套词法规则，供程序员定义接口。有关这些词法的使用，将在下一讲中作简要介绍。
  - IDL的一个重要特性是，一个接口可以继承一个或多个其他的接口。
- 2.语言映射
  - 因为IDL只是一种说明性语言，它不能用于编写实际的应用程序，它不是提供控制结构或变量，所以它不能被编译或多用解释成一个可执行的程序，它只适用于说明我对象的接口，定义用于对象通信的数据类型。
  - 语言映射指定如何把IDL翻译成不同的编程语言，目前OMG IDL语言映射可适用于C、C＋＋、Java、COBOL、Smalltalk、Ada等。
  - IDL语言映射是开发应用程序的关键，它们提供CORBA所支持的抽象概念和模型的具体实施。
  - IDL经过语言映射后，被翻译成特定语言的存根和框架，用于客户端和服务器端的程序的组成部分。
- 3.操作调用和调度软件
  - CORBA应用程序是以接收CORBA对象的请求或调用CORBA对象的请求这种形式工作的。
  - 调用请求的两种方法
    - 静态调用和调度
      - 采用这种方法，IDL被被翻译成特定语言的存根(stub)和框架(Skeleton)，这些存根和框架被编译成应用程序，一个存根(Stub)是一个客户端函数，它允许请求调用作为平常的本地函数调用，框架(Skeleton)是服务器端的一个函数，它允许由服务器接收到的请求调用被调度给合适的伺服程序(Servant procedure)。
      - 这种方法很受欢迎，它提供了一个更自然的编程模式。
    - 动态调用和调度

        - 这种方法涉及到CORBA请求的结构和调度是在运行时进行的，而不是在编译时产生的，因为疫有编译状态信息，所以在运行时请求创建和解释需要访问服务程序，由它们来提供有关的接口和类型信息。
    - 调用请求的过程
        - 定位目标对象；
        - 调用服务器应用程序；
        - 传递调用这个对象所需的参数；
        - 必要时，激活这个对象的伺服程序；
        - 等待请求结束；
        - 如果调用成功，返回结果值和参数值；
        - 如果失败，返回一个异常。
    - 对象引用(Object Reference)
        - 客户程序通过发送消息来控制对象，每当客户调用一个操作时，ORB发送一个消息给服务器对象。为了能发送一个消息给一个对象，客户必须拥有该对象的对象引用(Object Reference)，对象引用起着一个句柄的作用，句柄标识唯一的一个对象，并且封装了要将所有消息发送给正确的目标ORB所需要的信息。
        - 对象引用与C + + 中的类的实例指针有相同有语义，Java中没有指针的概念而用引用。
        - 每个对象引用必须准确地标识一个对象；
        - 一个对象可以有多个引用；
        - 引用可以是空的；
    - 引用的获取
        - 对象引用是客户程序获得目标对象唯一的途径，引用由服务器以某种方式发布的，常见的有：
            - 返回一个引用作为一个操作；
            - 以某些已知的服务程序公告一个引用（Naming Service 或Trading Service)
            - 通过将一个对象引用转换成一个字符串和将它写在一个文件上，来公布一个对象的引用；
            - 通过其他可以外传的方式来传送一个对象引用（电子邮件等）
    - 请求调用的特征
        - 定位透明性
            - 客户不知道也不必关心目标对象是否在本地的，是否在同一机器上不同的进程中实现，或者是在不同的机器上同一个进程中实现的。服务器进程也不必始终保留在同一台机器上。
        - 服务器透明性
            - 客户不必知道哪个服务器实现了哪些对象；
        - 语言独立性
            - 客户机无须关心服务器使用何种语言。
        - 实现独立性
            - 客户并不知道实现是如何工作的。
        - 体系结构独立性
            - 客户不必顾及服务器使用的CPU结构体系，并且屏蔽了字节的顺序等细节问题。
        - 操作系统独立性
            - 客户不必考虑服务器使用何种的操作系统，甚至服务器程序可以在不需要操作系统支持下实现（比如一些嵌入式系统）。
        - 协议独立性
            - 客户不知道发送消息是采用什么通信协议，如果服务器可以采用多个通信协议，那么ORB可以在运行时任意选择一个。
        - 传输独立性
            - 客户机忽略消息传送过程中网络的传输层和数据层，ORB可以透明地使用各种网络技术。

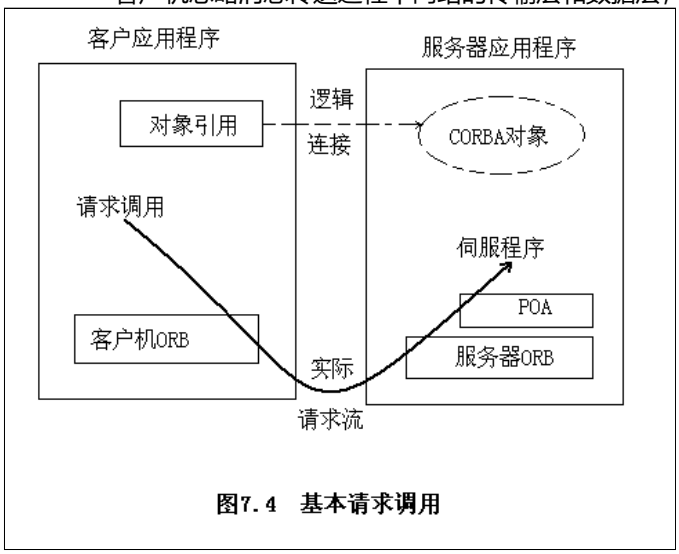


图7.4 基本请求调用

- 4.对象适配器（Object Adapter)
    - 伺服程序（Servant)

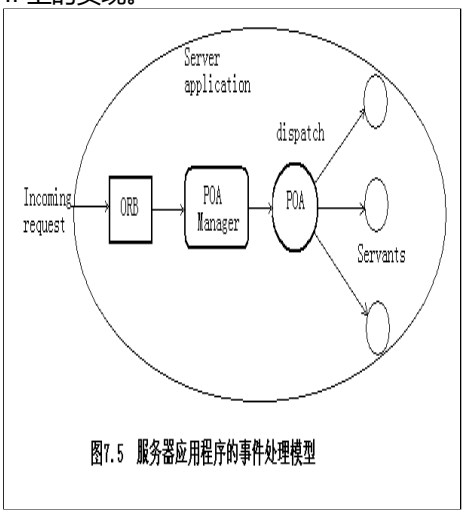
        - 它是一个编程语言的实体，用来实现一个或多个CORBA对象。伺服程序也称具体化的CORBA对象，它存在于服务器应用程序的上下文(Context)中，比如，在C + + 或Java中伺服程序一个特定类的一个对象实例。
    - 一个对象适配器是一个对象，它将一个对象接口配置给调用程序所需要的不同接口，CORBA对象适配器满足下面三个要求：
        - 创建允许客户程序对对象寻址的对象引用；
        - 确保每个目标对象都应由一个伺服程序来具体化；
        - 获取由服务器端的ORB所调度的请求，并进一步将请求产直传送给已具体化的目标对象的伺服程序。
    - 基本对象适配器(BOA,Baisc Object Adapter)
        - CORBA早期版本(2.1之前）的规范中只有基本对象适配器，只能支持C语言的伺服程序，伺服程序没有注册。
    - 可移植对象适配器(POA,Portable Object Adapter)
        - CORBA2.2引入了可移植对象适配器(POA,Portable Object Adapter)来取代BOA，它提供了编程语言的伺服程序在由不同的厂家提供的ORB之间的可移植性，POA提供的基本服务有：对象创建、伺服程序的注册、请求调度(Dispatch)。
- 5.内部ORB协议
    - CORBA2.0引入一个通用的ORB互操作性结构体系,称为GIOP（General Inter-ORB Protocol,通用ORB协议）。
    - GIOP是一类抽象的协议，它指定了转换语法和一个消息格式的标准集，以便允许独立开发的ORB可以在任何一个面向连接的传递中进行通信。IIOP是internet网上的ORB协议（Interent Inter-ORB Protocol, IIOP),它是GIOP在TCP/IP上的实现。
    - 

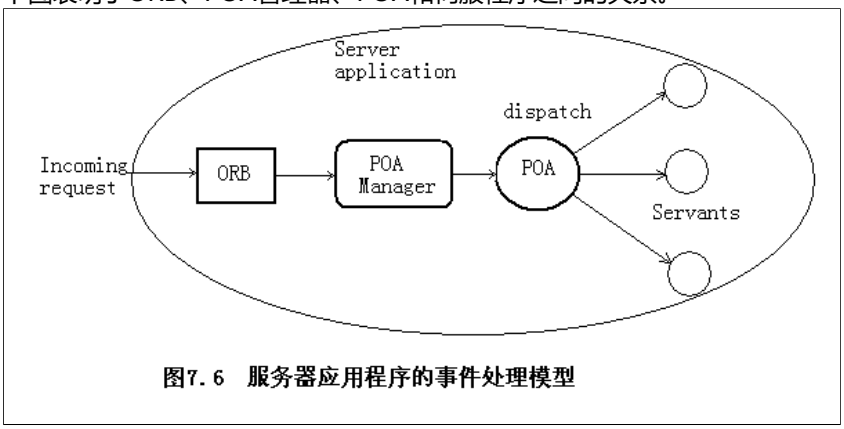
图7.5 服务器应用程序的事件处理模型
    - 服务器应用程序的事件处理模型
        - 下图表明了ORB、POA管理器、POA和伺服程序之间的关系。
        - 

图7.6 服务器应用程序的事件处理模型

三、CORBA应用程序的一般开发过程

- 基于CORBA的系统包括客房客户程序和服务器程序两部分。通常需要执行以下几个步骤
    - 确定应用程序对象，定义它们在IDL中的接口
    - 将定义的IDL文件进行语言映射
    - 声明和实现服务器程序中的伺服类
    - 编写应用服务器程序。
    - 编写客户程序

四、CORBA的基本服务

- 1.命名服务(Naming Service)
    - 命名服务是CORBA最基本的服务，它提供从名称到对象引用的映射：给定一个名称，该服务返回一个存储在此名称下的一个对象引用。很像internet上的DNS。
    - 命名服务给客户程序带来的好处
        - 客户程序可以给对象起个有意义的名称而不必处理字符串化的对象引用；

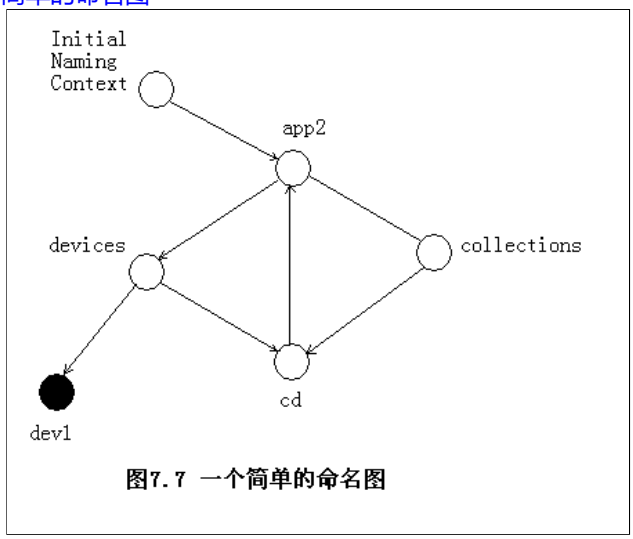
        - 通过改变在某个名称下的公告的引用值，客户程序可以在不改变源代码的情况下使用不同接口的实现，即客户程序使用同一个名称却获得不同的引用；
        - 命名服务可以使应用程序的组元访问一个应用程序的初始引用。在命名服务中公告这些引用，可以避免将引用变为字符串化的引用并存储在文件中的必要性。
    - 命名图(naming graph)
        - 名称绑定(name binding)
            - 将名称映射为对象引用，称为名称绑定(name binding)。同一个对象引用可以使用不同的名称而多次被存储，但是每个名称只能准确也确定一个引用。
        - 命名上下文(naming context)
            - 一个存储名称绑定(name binding)的对象，称为命名上下文(naming context)。每一个上下文对象实现一个从名称到对象引用的映射表，这个表中的命名可以表示某个应用程序的对象引用，也可以表示命名服务中的另一个上下文对象。
            - 一个上下文和名称绑定(name binding)的层次结构称为命名图(naming graph)。
        - 一个简单的命名图
        - 

图7.7 一个简单的命名图
    - 名称解析(name resolve)
        - 命名服务提供(resolve)操作，由命名服务器将客户程序中一个指定的名称转换为对应的对象引用并返回。
- 2.交易服务(Trading Service)
    - 命名服务允许一个客户程序通过一个符号名来定位对象的引用，这种机制对于客户程序定位一个对象很有用，但这要求客户程序必须确切知道要使用什么对象。
    - 往往客户程序需要多种机制来定位一个对象，例如，一个客户程序可能只知道所需要的对象类型，对要做出精确选择的其他必要信息并不清楚。这时CORBA的交易服务(Trading Service)提供了这种功能。允许客户程序借助交易来定位对象。
    - 与命名服务类似，一个交易用来存储对象引用及其服务描述，客户程序执行动态查找服务，此服务是基于查询服务描述的。
    - 基本的交易概念
        - 公告
            - 也称服务提供源(service offer),用于存储交易服务，一个service offer包含此服务的描述(一组属性)和一个提供此服务的对象引用和服务类型。
        - 导出
            - 放置一个公告的行为称为导出(export)操作,放置一个公告的程序称为导出者或服务提供者(service provider)，
        - 导入
            - 为一个符合一定标准的服务提供者搜索交易的行为称为导入(import)
        - 交易服务的基本轮廓：一个交易就是一个用于存储属性描述对象引用的数据库，我们可以导出(增加)新对象引用和它的描述，或者收回它们。
- 3.事件服务(Event Service)
    - 前面我们介绍的CORBA请求调用是基于同步的请求调用，在同步请求上，一个主动的客户程序向被动的服务器调用请求，在发送一个请求后，客户程序阻塞并等待返回结果。
    - CORBA事件服务允许服务器向客户发送消息，即将C/S方式转化为对待方式(peer-peer)。
    - CORBA的事件服务模型
        - 提供者(suppliers)生成事件而使用者(consumers)接收事件，提供者和使用者都连接一个事件通道（event channel)上，事件通道将事件从提供者传送到使用者，而且不需要提供者事先了解使用者的情况，反之亦然。事件通道在事件服务中起着关键作用，它同时负责提供者和使用者的注册，可靠地将事件发送给所有注册的使用者，并且负责处理那些与没有响应的使用者有关的错误。
    - 事件服务发送事件的模型
        - 推模型(push model)

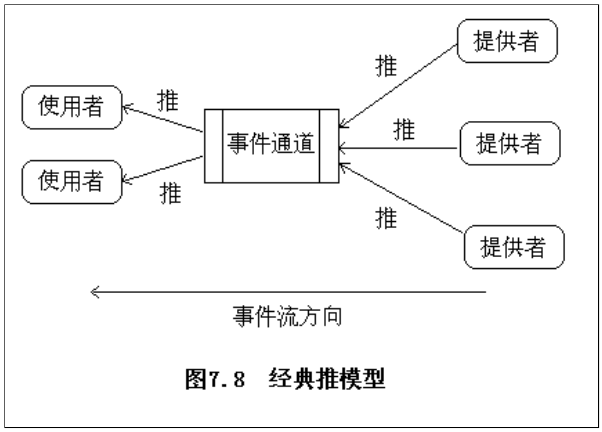


图7.8 经典推模型

- 拉模型(pull model)

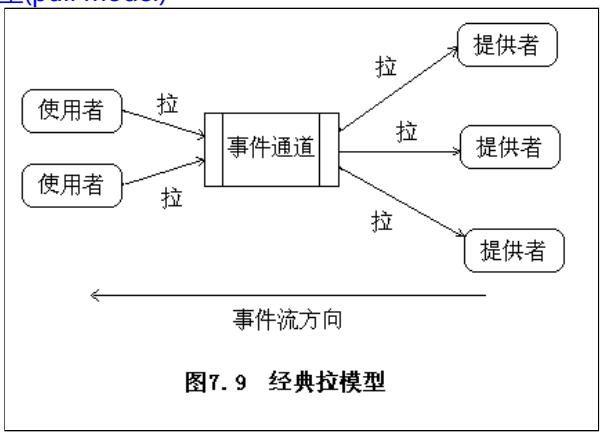


图7.9 经典拉模型

五、BES：一个优秀的CORBA开发平台

The Borland Enterprise Server provides a complete enterprise platform that is designed to meet your current business requirements as well as your future corporate growth. With the Borland Enterprise Server, you have the power of distributed object technology using both CORBA and J2EE components. This combination allows for rapid application development and true distributed deployment of your enterprise application in a robust, multi-tiered, heterogeneous environment. Mixing and matching your enterprise landscape with CORBA and J2EE technology(s) allows a great deal of flexibility, accessibility, scalability, and synchronization among disparate systems which includes remote databases and legacy systems. The Borland Enterprise Server achieves corporate clustering and load balancing by introducing the concept of the “Partition”. The Borland Partition is one key element for scalability and high accessibility. Seamless integration of web components, EJB, and CORBA is made possible by Borland Enterprise Server product editions.

The Borland Enterprise Server is available in three editions — the Web Edition, the VisiBroker Edition, and the AppServer Edition. Each of these builds upon the tools and services of its predecessor. The most basic of the editions is the Web Edition, the most versatile is the AppServer Edition. You can upgrade from one Edition to another, if desired. All editions of the Borland Enterprise Server are conglomerations of core services, partitions, and partition services.

Web Edition

1.BES Web Edition is designed for developing and deploying web application using JavaServer Pages and Servlets with an Java-based database. The Web Edition includes:

- Apache Server 1.3.
- Tomcat web container 4.0.
- the Smart Agent for object referencing and directory service for server connection.
- Java Session Service (JSS) to store session information for recovery in case of container failure.
- IIOP Plug-in that enables Apache and Tomcat to communicate with each other via IIOP. The IIOP Plug-in leverages the power of the VisiBroker features such as performance, fail-over and clustering.
- the Dreamweaver UltraDev Plug-in for those using Macromedia products to produce dynamic web pages using JavaServer Pages and servlets.
- JDataStore, an all-java relational database.

2.Web Edition's Features

- Provides a complete deployment platform for web applications
- Delivers industry-proven load balancing and fault tolerance
- Provides automatic session management
- Allows you to web-enable CORBA servers
- Provides a deployment environment for Web Service applications developed with Delphi

- Provides a homogeneous integration to an all-Java database with support for multiple connections
- 3.architectural view of the Web Edition services

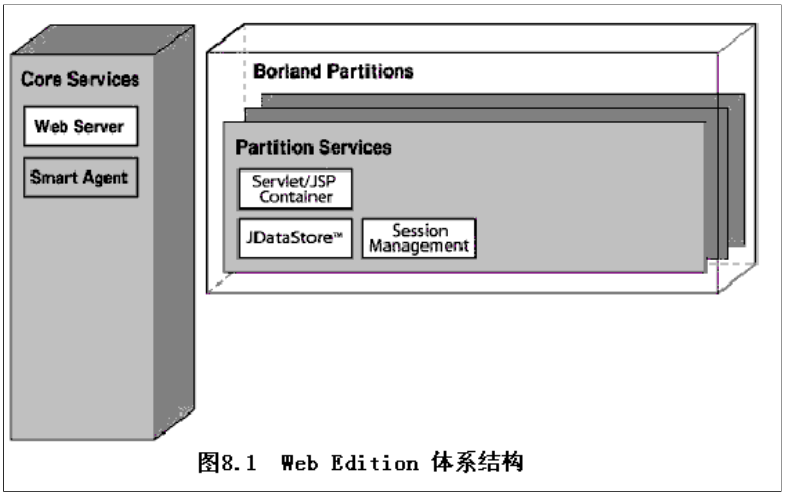
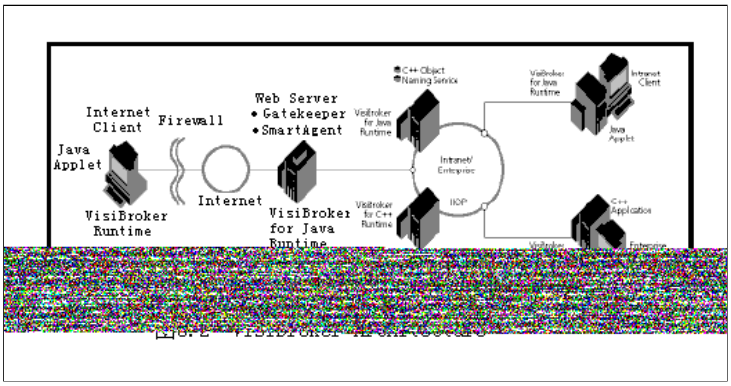


图8.1 Web Edition 体系结构

  - Core Services
    - Smart Agent
      - The Smart Agent is a dynamic, distributed directory service that provides facilities for both the client programs and object implementation. The Smart Agent maps client programs to the appropriate object implementation by correlating the object or service name used by the client program to bind to an object implementation. The object implementation is an object reference provided by a server, such as the Tomcat Web Container.
      - The Smart Agent must be started on at least one host within your local network. When your client program invokes (using the bind method) on an object, the Smart Agent is automatically consulted. The Smart Agent locates the specified object implementation so that a connection can be established between the client and the object implementation. The communication with the Smart Agent is transparent to the client program.
      - In the Web Edition, the Smart Agent is used in two specific scenarios:
        - Connecting Apache Web Server to Tomcat Web Container
        - Connecting Tomcat Web Containers to Java Session Service
    - Web server
      - Borland’ s web server is an implementation of the open-source Apache Web Server version 1.3.
  - Partition Services
    - Partition
      - The Borland Enterprise Server Web Edition allows for the creation of numerous “Partitions” for hosting your applications. A Partition is a place on the server for hosting entire applications or application components. When the Borland Enterprise Server is started, a default Partition is started with all available services turned-on. You can customize a Partition by turning on/off the services you require.
    - The Partition Services available with the Web Edition
      - Web Container
        - the open-source Tomcat 4.0 Web Container supports servlets and JSP technologies.
      - Java Session Service (JSS)
        - JSS is used to store session information for recovery in case of container failure. Borland provides an Interface Definition Language (IDL) interface for the use of JSS with JDataStore Borland’ s all-java relational database.
      - JDataStore
        - Borland JDataStore is an all-Java multifaceted data storage. In the Web Edition, JDataStore is pre-configured and ready-to-use. By default, it uses Borland's DataExpress for connectivity which is suited for local access.
- 4.Clustering of multiple Web components
  - In a typical deployment scenario, you can use multiple Borland Partitions to work together in providing a scalable n-tier solution. Each Borland Partition can have the same or different services. These services can be turned off or on depending on your clustering schema. In any case, leveraging these resources together or clustering, makes deployment of your web application more efficient. Clustering of the web components involve session management, load balancing and fault tolerance (failover). Interaction between the client and server involves two types of connection services:
  - stateless

        - A stateless service does not maintain a state between the client and the server. There is no "conversation" between the server and the client while processing a client request.
        - Load balancing--When clustering several web components, client requests are serviced to load balance request in an even, distributed fashion between the multiple Tomcat Web Containers using a round-robin scheme. For example, the first client request goes to Tomcat 1. The second client request goes to Tomcat 2. The third client request goes to Tomcat 1. And so forth.
        - Fault tolerance--When the first instance of Tomcat goes down, the Apache Web Server transparently re-routing of all client requests to the second instance of Tomcat. Since this is a stateless connection (no session), the JSS is not involved.
    - stateful
        - In a stateful service, the client and server maintains a dialogue of information, thereby having multiple interactions with one another the client in the context of the same session.
        - Session management--The Java Session Service(JSS) is responsible for session management of clustered web components. It does not support load balancing indiscriminately, meaning that the Apache Web Server needs to send data to the first Tomcat instance that it initially started in the session. The proceeding session is then sent to the next Tomcat instance. And so forth. In the diagram below, Session 1 is sent to the first Tomcat instance and Session 2 is sent to the second instance of Tomcat. Any client request from Session 1 will always be serviced in the first Tomcat instance.
        - JSS Role--In a stateful service connection, the JSS provides session management to all Tomcat Web Containers. Session data between the client and a specific Tomcat instance is "flushed" to the JSS for session management.When the first instance of Tomcat goes down, then the Apache Web Server finds the next Tomcat instance and send the client request to that instance. That particular Tomcat instance then consults JSS to obtain session information.
        - Fault tolerance--In the Web Edition, fault tolerance or failover is supported by the JSS that allows continuous session processing when a web container (Tomcat) dealing with the session suddenly dies. JSS provides a transparent mechanism to store session data in the database. The JSS caches entity and session information for recovery in case of the web container or service failure.
- VisiBroker Edition
    - 1.What is the VisiBroker Edition
        - VisiBroker Edition provides a complete CORBA 2.4 ORB runtime and supporting development environment for building, deploying, and managing distributed for both C++ and Java applications that are open, flexible, and inter-operable. Objects built with VisiBroker Edition for Java and C++ are easily accessed by Web-based applications that communicate using OMG's Internet Inter-ORB Protocol (IIOP) standard for communication between distributed objects through the Internet or through local intranets. VisiBroker Edition has a built-in implementation of IIOP that ensures high-performance and inter-operability.
    - 2.VisiBroker Edition architecture
        - 

    - 3.VisiBroker Edition features
        - VisiBroker Edition Smart Agent architecture
            - A Smart Agent must be started on at least one host within your local network.
            - VisiBroker Edition's Smart Agent (osagent) is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Multiple Smart Agents on a network cooperate to provide load balancing and high availability for client access to server objects. The Smart Agent keeps track of objects that are available on a network, and locates objects for client applications at invocation time. VisiBroker Edition can determine if the connection between your client application and a server object has been lost, due to an error such as a server crash or a network failure. When a failure is detected, an attempt is automatically made to connect your client to another server on a different host, if it is so configured.
        - Enhanced object discovery with the Location Service

- VisiBroker Edition provides a powerful Location Service—an extension to the CORBA specification—that enables you to access the information from multiple Smart Agents. Working with the Smart Agents on a network, the Location Service can see all the available instances of an object to which a client can bind. Using triggers, a callback mechanism, client applications can be instantly notified of changes to an object's availability. Used in combination with interceptors, the Location Service is useful for developing enhanced load balancing of client requests to server objects.
- The Location Service communicates directly with one Smart Agent which maintains a catalog, which contains the list of the instances it knows about. When queried by the Location Service, a Smart Agent forwards the query to the other Smart Agents, and aggregates their replies in the result it returns to the Location Service.

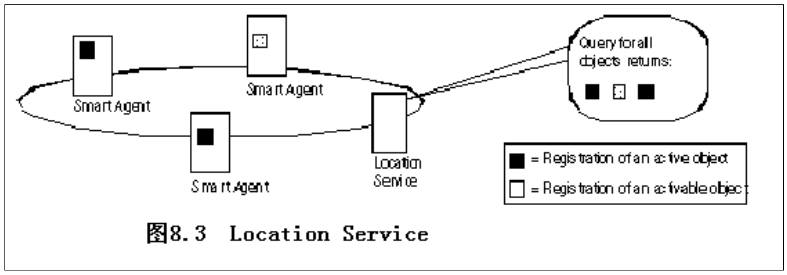


图8.3 Location Service

- Implementation and object activation support
  - VisiBroker Edition's Object Activation Daemon (OAD) can be used to automatically start object implementations when clients need to use them. Additionally, VisiBroker provides functionality that enables you to defer object activation until a client request is received. You can defer activation for a particular object or an entire class of objects on a server.
  - Activation can occur in one of several ways
    - Explicit activation - - The server application itself explicitly activates objects by calling activate_object or activate_object_with_id.
    - On-demand activation - - The server application instructs the POA to activate objects through a user-supplied servant manager. The servant manager must first be registered with the POA through set_servant_manager.
    - Implicit activation - - The server activates objects solely by in response to certain operations. If a servant is not active, there is nothing a client can do to make it active (for example, requesting for an inactive object does not make it active.)
    - Default servant - - The POA uses a single servant to implement all of its objects.
- Robust thread and connection management
  - VisiBroker Edition provides native support for single and multithreading thread management. With VisiBroker Edition's thread-per-session model, threads are automatically allocated on the server-per-client connection to service multiple requests, and then are terminated when the connection ends. With the thread pooling model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads— ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced.
  - VisiBroker Edition's connection management minimizes the number of client connections to the server. All client requests for objects residing on the same server are multiplexed over the same connection, even if they originate from different threads. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the same server.
  - All thread and connection behavior is fully configurable. See Chapter 12, "Managing threads and connections," for details on how VisiBroker Edition manages threads and connections.
- IDL compilers
  - VisiBroker Edition comes with two IDL compilers that make object development easier
  - idl2java: The idl2java compiler takes IDL files as input and produces the necessary client stubs and server skeletons in Java.
  - idl2cpp: The idl2cpp compiler takes IDL files as input and produces the necessary client stubs and server skeletons in C++.
- Dynamic invocation with DII and DSI
  - For dynamic invocation, VisiBroker Edition provides implementations of both the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). The DII allows client applications to dynamically create requests for objects that were not defined at compile time. The DSI allows servers to dispatch client operation requests to objects that were not defined at compile time.
  - The Dynamic Skeleton Interface (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the idl2cpp compiler in C++ or the idl2java compiler in Java. The DSI allows an object to register itself with the VisiBroker ORB,

receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class generated by the idl2cpp compiler in C++ or the idl2java compiler in Java.

- Interface and implementation repositories
  - The Interface Repository (IR) is an online database of meta information about the VisiBroker ORB objects. Meta information stored for objects includes information about modules, interfaces, operations, attributes, and exceptions.
  - An interface repository (IR) is like a database of CORBA object interface information that enables clients to learn about or update interface descriptions at runtime. In contrast to the VisiBroker Edition Location Service, which holds data describing object instances, an IR's data describes interfaces (types). There may or may not be available instances that satisfy the interfaces stored in an IR. The information in an IR is equivalent to the information in an IDL file (or files), but it is represented in a way that is easier for clients to use at runtime.
  - The Implementation Repository is an online database of meta information about implementations of the VisiBroker ORB objects. The Object Activation Daemon is VisiBroker Edition's interface to the Implementation Repository that is used to automatically activate the implementation when a client references the object.
- Server-side portability
  - VisiBroker Edition supports the CORBA Portable Object Adapter (POA), which is a replacement to the Basic Object Adapter (BOA). The POA shares some of the same functionality as the BOA, such as activating objects, support for transient or persistent objects, and so forth. The POA also has new features, such as the POA Manager and Servant Manager which creates and manages instances of your objects.
- Customizing the VisiBroker ORB with interceptors and object wrappers
  - VisiBroker Edition's 4.x interceptors enable developers to view under-the-cover communications between clients and servers. The VisiBroker Edition 4.x interceptors are Borland's proprietary interceptors. Interceptors can be used to extend the VisiBroker ORB with customized client and server code that enables load balancing, monitoring, or security to meet specialized needs of distributed applications.
  - The VisiBroker Edition ORB provides a set of interfaces known as interceptors which provide a framework for plugging in additional ORB behavior such as security, transactions, or logging. These interceptor interfaces are based on a callback mechanism. For example, using the interceptors, you can be notified of communications between clients and servers, and modify these communications if you wish, effectively altering the behavior of the VisiBroker Edition ORB. At its simplest usage, the interceptor is useful for tracing through code. Because you can see the messages being sent between clients and servers, you can determine exactly how the ORB is processing requests.
  - VisiBroker Edition Edition's object wrapper feature allows you to define methods that are called when a client application invokes a method on a bound object or when a server application receives an operation request. Unlike the interceptor feature described in which is invoked at the VisiBroker ORB level, object wrappers are invoked before an operation request has been marshalled. In fact, you can design object wrappers to return results without the operation request having ever been marshalled, sent across the network, or actually presented to the object implementation.
- Event Queue
  - The event queue is designed as a server-side only feature. A server can register the listeners to the event queue based on the event types that the server is interested in and processes those events when the need arises.

AppServer Edition

- 1.EJB
  - The Enterprise JavaBeans architecture is a high-level component-based architecture for distributed business applications that uses the transaction system's lower-level APIs. EJB simplifies the development, deployment, and execution of enterprise systems in the Java programming language. The Enterprise JavaBeans architecture is defined in a specification developed and edited by Sun Microsystems. Borland's EJB container is based on version 2.0 of the specification.
  - The Enterprise JavaBeans technology defines a set of reusable components called enterprise beans. You create a distributed application by coding the application's business logic in these enterprise beans. Once the coding is completed, the enterprise beans are assembled into special files, with one or more enterprise beans per file, along with deployment parameters. Lastly, the enterprise beans are deployed onto a platform that runs an EJB container. Clients can locate an enterprise bean and create an instance of that bean through the enterprise bean's home interface. Then, the client can invoke the business methods of the enterprise bean using the enterprise bean's remote interface.

        - The EJB server manages EJB containers and functions as a bridge between the container and the underlying platform. It provides its EJB containers with access to the platform’ s system services, such as database management and a transaction monitor, and to other existing enterprise applications.
        - All enterprise bean instances run within an EJB container. The container provides system-level services to its enterprise beans and controls their life cycle. Because the container handles most system-level issues, the enterprise bean developer does not have to include this logic with the business methods of the enterprise bean. In general, containers handle such system-level issues as:
        - Security—The deployment descriptor defines the clients that can access the different business methods. The container enforces this by permitting only authorized clients to invoke those methods to which they have access.
        - Remote connectivity—The container manages the low-level communication issues for remote connectivity and hides these issues from the enterprise bean developer and the client. An enterprise bean developer writes the business methods as if they will be invoked on a local platform; the client is unaware that he or she is invoking a method that potentially must be reached remotely.
        - Life Cycle management—Clients simply create instances of enterprise beans and (usually) remove these instances. However, the container manages these enterprise bean instances to maximize performance and memory usage. The container may do such things as inactivate and activate these enterprise bean instances, keep a pool of instances to share among clients, and so on.
        - Transaction management—The deployment descriptor defines the transactional requirements of an enterprise bean. The container manages the complex issues of managing distributed transactions that potentially update databases spread across multiple platforms. The container keeps transactional data isolated, and it ensures that updates to all the databases occur successfully; otherwise, it rolls back all aspects of the transaction.
    - 2.Client view of an enterprise bean
        - A client of an enterprise bean is an application—a stand-alone application, an application client container, servlet, or applet—or another enterprise bean. In all cases, the client must do the following things to use an enterprise bean:
        - Locate the bean’ s home interface. The EJB specification states that the client should use the JNDI (Java Naming and Directory Interface) API to locate home interfaces.
        - Obtain a reference to an enterprise bean object’ s remote interface. This involves using methods defined on the bean’ s home interface. You can either create a session bean, or you can create or find an entity bean.
        - Invoke one or more methods defined by the enterprise bean. A client does not directly invoke the methods defined by the enterprise bean. Instead, the client invokes the methods on the enterprise bean object’ s remote interface. The methods defined in the remote interface are the methods that the enterprise bean has exposed to clients.
        - Session beans
            - A client obtains a reference to a session bean’ s remote interface by calling one of the create methods on the home interface.
        - Entity beans
            - A client obtains a reference to an entity object either through a find operation or a create operation. Recall that an entity object represents some underlying data stored in a database. Because the entity bean represents persistent data, entity beans typically exist for quite a long time; certainly for much longer than the client applications that call them. Thus, a client most often needs to find the entity bean that represents the piece of persistent data of interest, rather than creating a new entity object, which would create and store new data in the underlying database.
- VisiBroker+JBuilder开发CORBA应用实例
- 六、课外实践作业（占期末成绩的30%，另外笔试占70%）

返回

==========