EJB 编程及 J2EE 系统架构和设计

邓晓军

摘要：本文阐述了 J2EE 平台的所有主要技术，围绕 J2EE 规范所定义的四个层次：客户端层（Client Tier）、Web 层(Web Tier)、业务层（Business Tier）及企业信息系统层（Enterprise Information System Tier），介绍 J2EE 所定义的丰富的技术标准及符合这些标准的开发工具和 API，这些技术涵盖了组件技术、Servlets 和 JSP、EJB 技术、数据库访问、分布式通信技术（Java RMI、Java IDL、JNDI、JMS）、安全等；本文试图给出 J2EE 平台技术概念理解上的一个较为清晰的完整的思路，帮助大家掌握各技术间的相互关系和重要的思想。

关键词：J2EE、JDBC、RMI、JNDI、JMS、EJB、Servlets、JSP、XML

1、引言

Java 2 平台有三个版本：它们是适用于小型设备和智能卡的 Java 2 平台 Micro 版（Java 2 Platform Micro Edition，J2ME）、适用于桌面系统的 Java 2 平台标准版（Java 2 Platform Standard Edition，J2SE）、适用于创建服务器应用程序和服务的 Java 2 平台企业版（Java 2 Platform Enterprise Edition，J2EE）。其中最重要的就是 J2EE 平台。

J2EE 是一种利用 Java 2 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。J2EE 技术的基础就是核心 Java 平台或 Java 2 平台的标准版，J2EE 不仅巩固了标准版中的许多优点，例如"编写一次、随处运行"的特性、方便存取数据库的 JDBC API、CORBA 技术以及能够在 Internet 应用中保护数据的安全模式等等，同时还提供了对 EJB（Enterprise JavaBeans）、Java Servlets API、JSP（Java Server Pages）以及 XML 技术的全面支持。其最终目的就

是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。

J2EE 体系结构提供中间层集成框架用来满足无需太多费用而又需要高可用性、高可靠性以及可扩展性的应用的需求。通过提供统一的开发平台，J2EE 降低了开发多层应用的费用和复杂性，同时提供对现有应用程序集成强有力支持，完全支持 Enterprise JavaBeans，有良好的向导支持打包和部署应用，添加目录支持，增强了安全机制，提高了性能。

J2EE 为搭建具有可伸缩性、灵活性、易维护性的商务系统提供了良好的机制：

保留现存的 IT 资产：由于企业必须适应新的商业需求，利用已有的企业信息系统方面的投资，而不是重新制定全盘方案就变得很重要。这样，一个以渐进的（而不是激进的，全盘否定的）方式建立在已有系统之上的服务器端平台机制是公司所需求的。J2EE 架构可以充分利用用户原有的投资，如一些公司使用的 BEA Tuxedo、IBM CICS，IBM Encina,、Inprise VisiBroker 以及 Netscape Application Server。这之所以成为可能是因为 J2EE 拥有广泛的业界支持和一些重要的'企业计算'领域供应商的参与。每一个供应商都对现有的客户提供了不用废弃已有投资，进入可移植的 J2EE 领域的升级途径。由于基于 J2EE 平台的产品几乎能够在任何操作系统和硬件配置上运行，现有的操作系统和硬件也能被保留使用。

高效的开发：J2EE 允许公司把一些通用的、很繁琐的服务端任务交给中间件供应商去完成。这样开发人员可以集中精力在如何创建商业逻辑上，相应地缩短了开发时间。高级中间件供应商提供以下这些复杂的中间件服务：

状态管理服务 -- 让开发人员写更少的代码，不用关心如何管理状态，这样能够更快地完成程序开发。

持续性服务 -- 让开发人员不用对数据访问逻辑进行编码就能编写应用程序，能生成更轻巧，与数据库无关的应用程序，这种应用程序更易于开发与维护。

分布式共享数据对象 CACHE 服务 -- 让开发人员编制高性能的系统，极大提高整体部署的伸缩性。

支持异构环境：J2EE 能够开发部署在异构环境中的可移植程序。基于 J2EE 的应用程序不依赖任何特定操作系统、中间件、硬件。因此设计合理的基于 J2EE 的程序只需开发一次就可部署到各种平台。这在典型的异构企业计算环境中是十分关键的。J2EE 标准也允许客户订购与 J2EE 兼容的第三方的现成的组件，把他们部署到异构环境中，节省了由自己制订整个方案所需的费用。

可伸缩性：企业必须要选择一种服务器端平台，这种平台应能提供极佳的可伸缩性去满足那些在他们系统上进行商业运作的大批新客户。基于 J2EE 平台的应用程序可被部署到各种操作系统上。例如

可被部署到高端 UNIX 与大型机系统，这种系统单机可支持 64 至 256 个处理器。(这是 NT 服务器所望尘莫及的) J2EE 领域的供应商提供了更为广泛的负载平衡策略。能消除系统中的瓶颈，允许多台服务器集成部署。这种部署可达数千个处理器，实现可高度伸缩的系统，满足未来商业应用的需要。

稳定的可用性：一个服务器端平台必须能全天候运转以满足公司客户、合作伙伴的需要。因为 INTERNET 是全球化的、无处不在的，即使在夜间按计划停机也可能造成严重损失。若是意外停机，那会有灾难性后果。J2EE 部署到可靠的操作环境中,他们支持长期的可用性。一些 J2EE 部署在 WINDOWS 环境中，客户也可选择健壮性能更好的操作系统如 Sun Solaris、IBM OS/390。最健壮的操作系统可达到 99。999%的可用性或每年只需 5 分钟停机时间。这是实时性很强商业系统理想的选择。

基于层次化组件模式的 J2EE 平台把业务逻辑和底层网络技术分离开来，具有可伸缩性、扩展性、易开发性和易维护性，已经成为企业级商业分布式网络计算的事实标准。J2EE 是大量业内技术专家、教育专家集体智慧和经验设计出来的一套先进、完美、实用的规范，遵从这个规范的开发者将得到行业的广泛支持，使企业级应用的开发变得简单、快速。学习 Java，与其说是学一种技术，还不如说是在学习一种编程思想，而 J2EE 系统平台的思想是通过一个基于组件的应用

程序模式为分布式应用程序提供一个统一的标准。

2、J2EE 简介

2．1 J2EE 平台规范（Java 2 Platform Enterprise Edition Platform Specification）

*J2EE 平台规范是一个*由 SUN 公司*定义的用于简化*分布式*企业级应用开发与部署的基于组件的模式*（The J2EE Platform Specification defines a component-based model that simplifies enterprise development and deployment)。它提供了一个多层次的分布式应用模型和一系列开发技术规范。多层次分布式应用模型是根据功能把应用逻辑分成多个层次，每个层次支持相应的服务器和组件，组件在分布式服务器的组件容器中运行（如 Servlet 组件在 Servlet 容器上运行，EJB 组件在 EJB 容器上运行，容器间通过相关的协议进行通讯，实现组件间的相互调用。

2．2 J2EE 组件和层次

J2EE 使用多层的分布式应用模型，应用逻辑按功能划分为组件，各个应用组件根据他们所在的层分布在不同的机器上。事实上，sun 设计 J2EE 的初衷正是为了解决两层模式(client/server)的弊端，在传统模式中，客户端担当了过多的角色而显得臃肿，在这种模式中，第一次部署的时候比较容易，但难于升级或改进，可伸展性也不理想，而且经常基于某种专有的协议——通常是某种数据库协议。它使得重

用业务逻辑和界面逻辑非常困难。现在 J2EE 的多层企业级应用模型将两层化模型中的不同层面切分成许多层。一个多层化应用能够为不同的每种服务提供一个独立的层，如图 1 所示,以下是 J2EE 规范的四个层次及相应的组件:

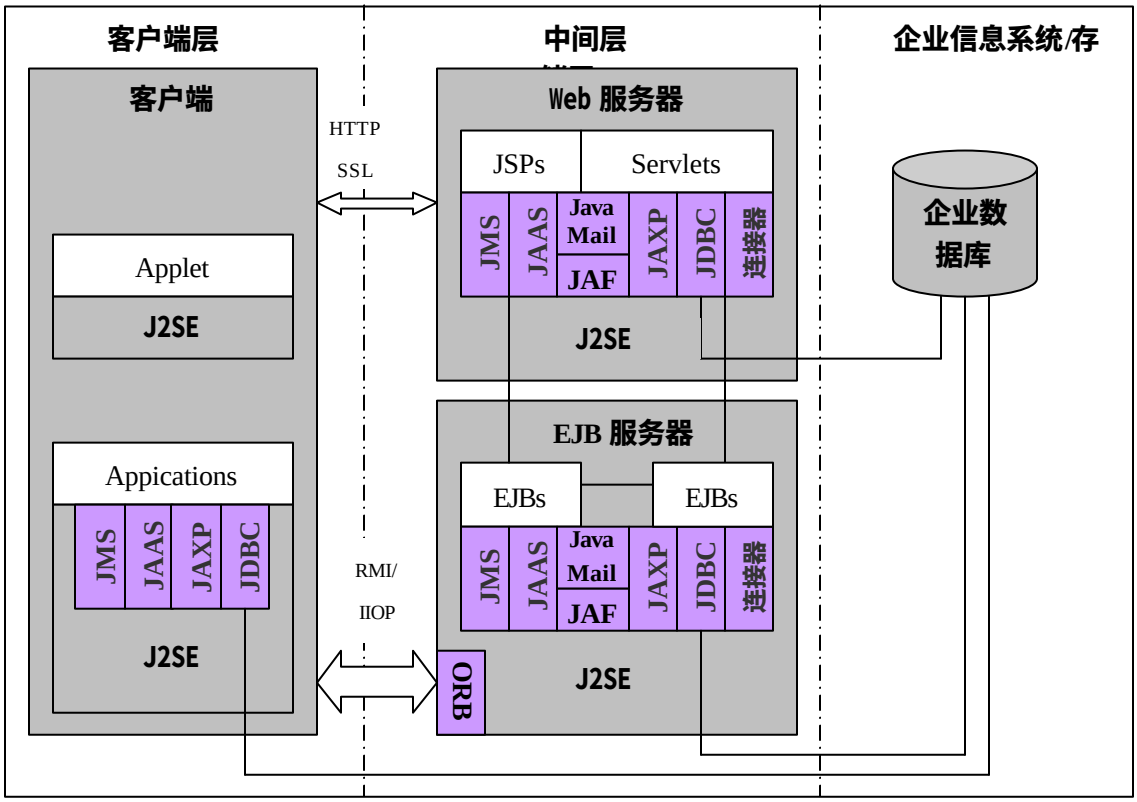


图 1 J2EE 组件和层次

这四层分别是 运行在客户端机器上的客户端层（Client Tier)、运行在 Web 服务器上的 Web 层（Web Tier)、 运行在 EJB 服务器上的业务层（Business Tier）和 运行在 EIS 服务器上企业信息系统层（Enterprise Information System Tier）其中 Web 层和业务层共同组成了三层 J2EE 应用的中间层，其他两层是客户端层和存储层或企

业信息系统层。一般情况下，许多开放商把 Web 服务器和 EJB 服务器产品结合在一起发布，称为应用服务器或 J2EE 服务器。J2EE 平台规范也定义了相应层的组件：

I. 客户端层组件

应用客户端程序和浏览器是客户端层组件。客户端层组件可以是基于 Web 方式的即作为 Web 服务器的浏览器,也可以是基于传统方式的(非基于 Web 方式)即独立的应用程序，可以完成瘦客户机无法完成的任务。

II.Web 层组件

Java Servlet 和 JavaServer Pages(JSP)是 Web 层组件。如图 2 所示的客户层那样，Web 层可能包含某些 JavaBean 对象来处理用户输入,并把输入发送给运行在业务层上的 Enterprise Bean 来进行处理。按照 J2EE 规范,静态的 HTML 页面和 Applets 不算是 Web 层组件。这里的 JavaBean 和 EJB（Enterprise JavaBean）除了共用‘JavaBean ”这个名字外，这两种组件模式完全没有关系。许多文章把 EJB 作为原始的‘JavaBean ’的扩展,这是错误的。EJB 并没有扩展或使用 JavaBean 组件模式。最初的 JavaBean（java.beans 包）在进程内部（intraprocess）使用，而 EJB（javax.ejb 包）是在进程间(interprocess)使用的组件。即最初的 JavaBean 不是为分布式组件而设的。它是最好的组件模式，可能是至今发现的最好的过程内部开

发的组件模式，但它不是一个服务器端的组件模式。EJB 则能解决在三层结构中由管理分布式商务对象多带来的问题。

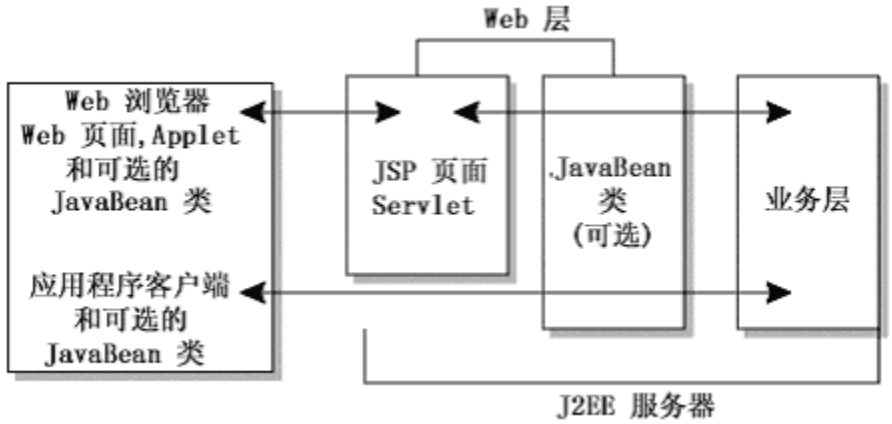


图 2 客户端层、Web 层及业务层

III. 业务层组件

Enterprise JavaBeans(EJB)是业务层组件。业务层代码的逻辑用来满足银行，零售，金融等特殊商务领域的需要,由运行在业务层上的 EJB 进行处理。 图 3 表明了一个 EJB 是如何从客户端程序接收数据，进行处理(如果必要的话)，并发送到企业信息系统层(EIS) 层储存的，这个过程也可以逆向进行。

有三种企业级的 Bean: 会话(Session)Beans，实体(Entity) Beans，和 消息驱动(Message-driven) Beans。 会话 Bean 表示与客户端程序的临时交互。当客户端程序执行完后，会话 Bean 和相关数据就会消失。相反，实体 Bean 表示数据库的表中一行永久的记录。当客户端程序中止或服务器关闭时，就会有潜在的服务保证实体 Bean 的数据得以保存。消息驱动 Bean 结合了会话 Bean 和 JMS 的消

息监听器的特性，允许一个业务层组件异步接收 JMS 消息。

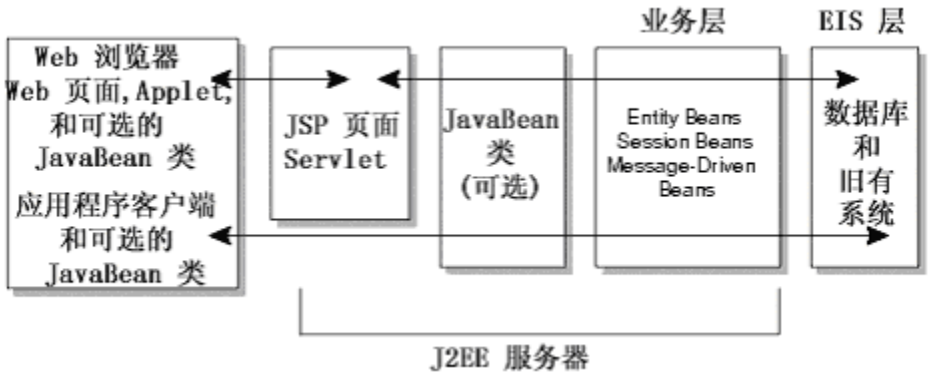


图 3 客户端层、Web 层、业务层及 EIS 层

IV. 企业信息系统层组件

处理企业信息系统软件包括企业基础建设系统例如企业资源计划（ERP)，大型机事务处理，数据库系统,和其它的遗留信息系统组成了企业信息系统层。 例如，J2EE 应用组件可能为了数据库连接需要访问企业信息系统。

2．3 J2EE 的分布式应用技术简介

J2EE 平台由一整套服务（Services)、应用程序接口（APIs）和协议构成，它对开发基于 Web 的多层、分布式应用提供了功能支持：

（1）组件/容器技术

如图 4 所示这种基于组件，具有平台无关性的 J2EE 结构使得 J2EE 程序的编写十分简单，因为业务逻辑被封装成可复用的组件，并且 J2EE 服务器以容器的形式为所有的组件类型提供后台服务。因为你不用自己开发这种服务，所以你可以集中精力解决手头的业务

问题。

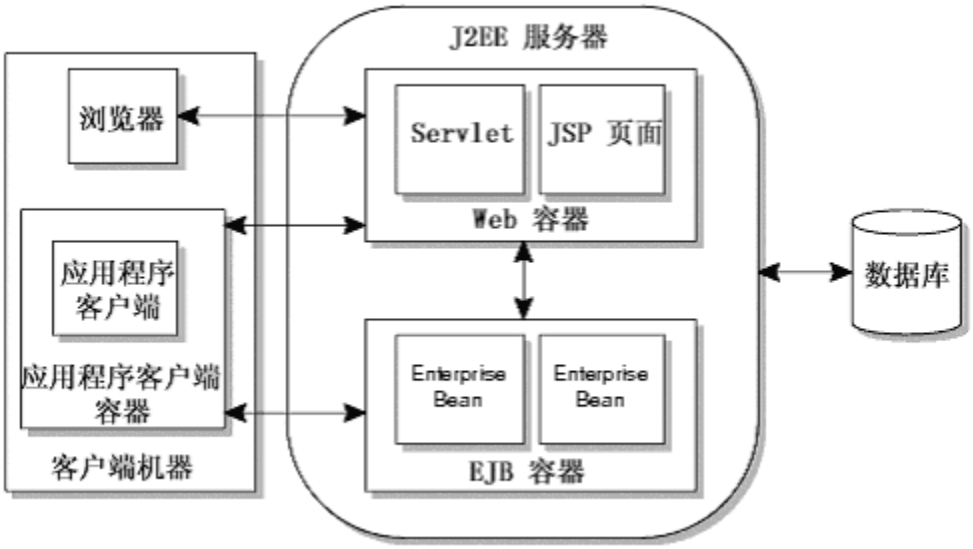


图 4 组件/容器结构

J2EE 应用组件可以安装部署到以下几种容器中去:

EJB 容器管理所有 J2EE 应用程序中 EJB 的执行。EJB 和它们的容器运行在 J2EE 服务器上。

Web 容器管理所有 J2EE 应用程序中 JSP 页面和 Servlet 组件的执行。 Web 组件和它们的容器运行在 J2EE 服务器上。

应用程序客户端容器管理所有 J2EE 应用程序中应用程序客户端组件的执行。 应用程序客户端和它们的容器运行在客户端机器上。

Applet 容器是运行在客户端机器上的 Web 浏览器和 Java 插件的结合。

容器设置定制了 J2EE 服务器所提供的内在支持，包括安全，事

务管理，JNDI(Java Naming and Directory Interface)寻址,远程连接等服务，以下列出最重要的几种服务：

J2EE 安全(Security)模型可以让你配置 Web 组件或 EJB ,这样只有被授权的用户才能访问系统资源。 每一客户属于一个特别的角色，而每个角色只允许激活特定的方法。你应在 EJB 的布置描述中声明角色和可被激活的方法。由于这种声明性的方法，你不必编写加强安全性的规则。

J2EE 事务管理（Transaction Management）模型让你指定组成一个事务中所有方法间的关系，这样一个事务中的所有方法被当成一个单一的单元。当客户端激活一个 EJB 中的方法，容器介入一管理事务。因有容器管理事务，在 EJB 中不必对事务的边界进行编码。要求控制分布式事务的代码会非常复杂。你只需在布置描述文件中声明 EJB 的事务属性，而不用编写并调试复杂的代码。容器将读此文件并为你处理此 EJB 的事务。

JNDI 寻址(JNDI Lookup)服务向企业内的多重名字和目录服务提供了一个统一的接口,这样应用程序组件可以访问名字和目录服务。

J2EE 远程连接（Remote Client Connectivity）模型管理客户端和 EJB 间的低层交互。 当一个 EJB 创建后，一个客户端可以调用它的方法就象它和客户端位于同一虚拟机上一样。

生存周期管理（Life Cycle Management）模型管理 EJB 的创建和移除，一个 EJB 在其生存周期中将会历经几种状态。容器创建 EJB，并在可用实例池与活动状态中移动他，而最终将其从容器中移除。即使可以调用 EJB 的 create 及 remove 方法，容器也将会在后台执行这些任务。

数据库连接池（Database Connection Pooling）模型是一个有价值的资源。获取数据库连接是一项耗时的工作，而且连接数非常有限。容器通过管理连接池来缓和这些问题。EJB 可从池中迅速获取连接。在 EJB 释放连接之后可为其他 EJB 使用。

（2）Servlets 和 JSP

JSP(Java Server Pages)：JSP 页面由 HTML 代码和嵌入其中的 Java 代码所组成。服务器在页面被客户端所请求以后对这些 Java 代码进行处理，然后将生成的 HTML 页面返回给客户端的浏览器。

Java Servlet：Servlet 是一种小型的 Java 程序，它扩展了 Web 服务器的功能。作为一种服务器端的应用，当被请求时开始执行，这和 CGI Perl 脚本很相似。Servlet 提供的功能大多与 JSP 类似，不过实现的方式不同。JSP 通常是大多数 HTML 代码中嵌入少量的 Java 代码，而 servlets 全部由 Java 写成并且生成 HTML。

（3）EJB 技术

EJB(Enterprise JavaBean)：J2EE 技术之所以赢得某体广泛重

视的原因之一就是 EJB。它们提供了一个框架来开发和实施分布式商务逻辑，由此很显著地简化了具有可伸缩性和高度复杂的企业级应用的开发。EJB 规范定义了 EJB 组件在何时如何与它们的容器进行交互作用。容器负责提供公用的服务，例如目录服务、事务管理、安全性、资源缓冲池以及容错性。但这里值得注意的是，EJB 并不是实现 J2EE 的唯一途径。正是由于 J2EE 的开放性，使得有的厂商能够以一种和 EJB 平行的方式来达到同样的目的。

（4）数据库访问

JDBC(Java Database Connectivity)：JDBC API 为访问不同的数据库提供了一种统一的途径，象 ODBC 一样，JDBC 对开发者屏蔽了一些细节问题，另外，JDCB 对数据库的访问也具有平台无关性。

（5）分布式通信技术及分布示应用技术

JNDI(Java Name and Directory Interface)：JNDI API 被用于执行名字和目录服务。它提供了一致的模型来存取和操作企业级的资源如 DNS 和 LDAP，本地文件系统，或应用服务器中的对象。

RMI(Remote Method Invoke)：正如其名字所表示的那样，RMI 协议调用远程对象上方法。它使用了序列化方式在客户端和服务器端传递数据。RMI 是一种被 EJB 使用的更底层的协议。

RMI-IIOP:它在 Internet Inter-ORB Protocol（IIOP）之上提供了通常的 Java Remote Method Invocation(Java 远程方法调用

RMI)API 的一种实现。它在 RMI 和 CORBA 应用程序之间架起了桥梁。这是在 J2EE 容器之间使用的一种标准通信协议。

Java IDL/CORBA：在 Java IDL 的支持下，开发人员可以将 Java 和 CORBA 集成在一起。他们可以创建 Java 对象并使之可在 CORBA ORB 中展开，或者他们还可以创建 Java 类并作为和其它 ORB 一起展开的 CORBA 对象的客户。后一种方法提供了另外一种途径，通过它 Java 可以被用于将你的新的应用和旧的系统相集成。

JMS(Java Message Service)：JMS 是用于和面向消息的中间件相互通信的应用程序接口(API)。它既支持点对点的域，有支持发布/订阅(publish/subscribe)类型的域，并且提供对下列类型的支持：经认可的消息传递,事务型消息的传递，一致性消息和具有持久性的订阅者支持。JMS 还提供了另一种方式来对您的应用与旧的后台系统相集成。

JTA(Java Transaction Architecture)：JTA 定义了一种标准的 API，应用系统由此可以访问各种事务监控。

JTS(Java Transaction Service)：JTS 是 CORBA OTS 事务监控的基本的实现。JTS 规定了事务管理器的实现方式。该事务管理器是在高层支持 Java Transaction API (JTA)规范，并且在较底层实现 OMG OTS specification 的 Java 映像。JTS 事务管理器为应用服务器、资源管理器、独立的应用以及通信资源管理器提供了事务服务。

JavaMail: JavaMail 是用于存取邮件服务器的 API，它提供了一套邮件服务器的抽象类。不仅支持 SMTP 服务器，也支持 IMAP 服务器。

JAF(JavaBeans Activation Framework): JavaMail 利用 JAF 来处理 MIME 编码的邮件附件。MIME 的字节流可以被转换成 Java 对象，或者转换自 Java 对象。大多数应用都可以不需要直接使用 JAF。

JAXP(Java API for XML Paring)：这个 API 为 XML 解析器和 API 的转换提供了抽象。JAXP 可以帮助把特定的 XML 解析器、XML Document Object Model（文档对象模式，DOM）实现或者把 XSLT 转换成 API 与 J2EE 应用程序代码隔离。

JCA(Java Connector Architecture)：这个 API 最近已经包含在 J2EE 中，提供了一种把 J2EE 应用程序组件集成到老式信息系统中的途径。

JAAS（Java Authentication and Authorization Service)：这个 API 为 J2EE 应用程序提供了验证和授权机制。

XML(Extensible Markup Language): XML 是一种可以用来定义其它标记语言的语言。它被用来在不同的商务过程中共享数据。XML 的发展和 Java 是相互独立的，但是，它和 Java 具有的相同目标正是平台独立性。通过将 Java 和 XML 的组合，您可以得到一个完美的具有平台独立性的解决方案。

3 、J2EE 平台的分布式应用技术详解

3．1 Java RMI

RMI （Remote Method Invocation 远程方法调用）是用 Java 在 JDK1.1 以上版本中实现的，它大大增强了 Java 开发分布式应用的能力。Java 作为一种风靡一时的网络开发语言，其巨大的威力就体现在它强大的开发分布式网络应用的能力上，而 RMI 就是开发百分之百纯 Java 的网络分布式应用系统的核心解决方案之一。其实它可以被看作是 RPC 的 Java 版本。但是传统 RPC 并不能很好地应用于分布式对象系统。而 Java RMI 则支持存储于不同地址空间的程序级对象之间彼此进行通信，实现远程对象之间的无缝远程调用。RMI 目前使用 Java 远程消息交换协议 JRMP（Java Remote Messaging Protocol）进行通信。也可选用其他的实现如 Bea Weblogic、NinjaRMI、ObjectSpace 的 Voyager 等不使用 JRMP，但他们使用自己的线级协议。Sun 和 IBM 已经联合开发了下一代 RMI,称为 RMI-IIOP,可以在 Java 2 SDK 版本 1.3 中获得。JRMP 是专为 Java 的远程对象制定的协议。因此,Java RMI 具有 Java 的"Write Once,Run Anywhere"的优点，是分布式应用系统的百分之百纯 Java 解决方案。用 Java RMI 开发的应用系统可以部署在任何支持 JRE（Java Run Environment Java，运行环境）的平台上。但由于 JRMP 是专为 Java 对象制定的，因此，RMI 对于用非 Java 语言开发的应用系统的支持不足。不能与用非 Java 语言书写的对象进行通信。

3.1.1 RMI 系统运行机理

RMI 应用程序通常包括两个独立的程序：服务器程序和客户机程序。典型的服务器应用程序将创建多个远程对象，使这些远程对象能够被引用，然后等待客户机调用这些远程对象的方法。而典型的客户机程序则从服务器中得到一个或多个远程对象的引用，然后调用远程对象的方法。RMI 为服务器和客户机进行通信和信息传递提供了一种机制。

RMI 实现实际上由三个抽象层建立：

1）Stubs/Skeletons Layer(存根/主架层)　这一层会截获客户发出的对接口引用的方法调用，并且把这些调用重定向到一个远程对象。需要记住的是，这里的 stubs（存根信息）是针对客户方的，而 skeletons(主架信息)则位于服务器方。

2）Remote Reference Layer（远程引用层）　这一层处理着与解释和管理客户对远程对象的建立的引用有关的细节。它把客户连接到正在运行的远程对象，并且通过对一对一的连接链路导出到一个服务器。在 Java 2 SDK 中，这一层得到了增强以支持激活框架(activation framework)。

3）TransportLayer（传输层）　这一层建立在网络上计算机之间的 TCP/IP 连接基础上。它提供了基本的连接能力以及一些放火墙渗透策略。

图 5 显示了这三层，以及传输层的划分。右侧的说明表示的是 OSI 层

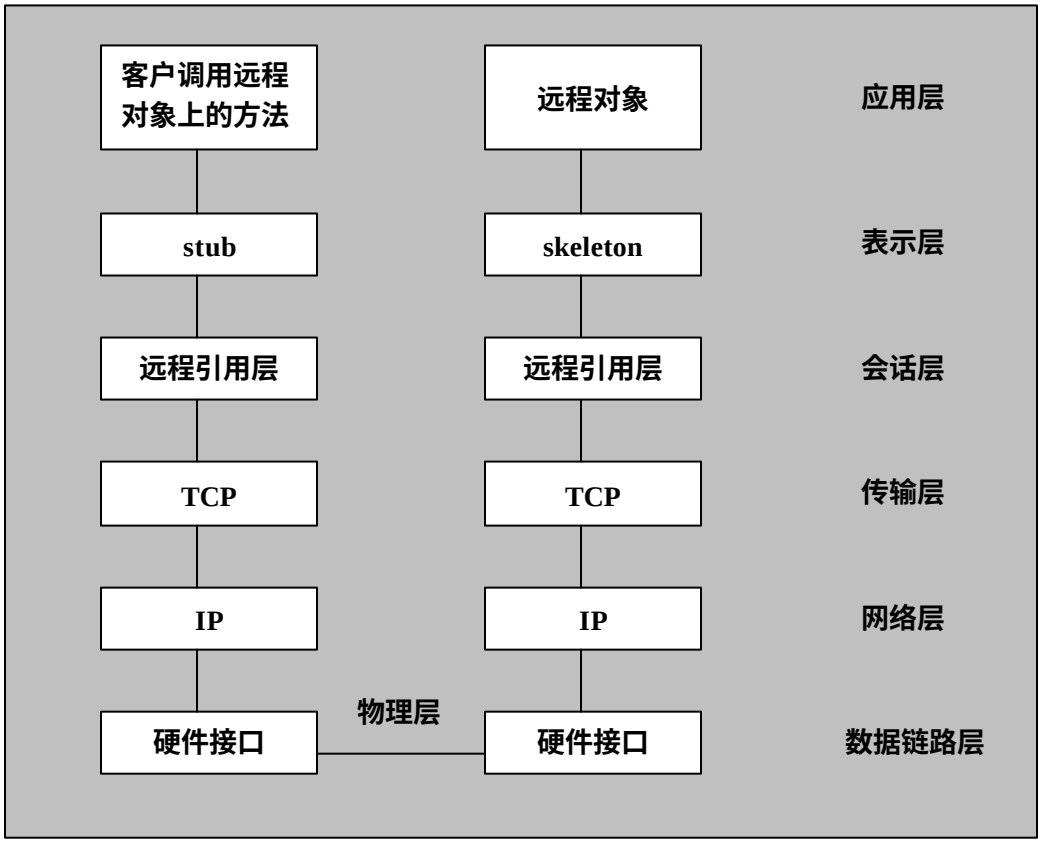


图 5 RMI 分层

在与远程对象的通信过程中，RMI 使用标准机制：stub 和 skeleton。远程对象的 stub 担当远程对象的客户本地代表或代理人角色。调用程序将调用本地 stub 的方法，而本地 stub 将负责执行对远程对象的方法调用。在 RMI 中，远程对象的 stub 与该远程对象所实现的远程接口集相同。调用 stub 的方法时将执行下列操作：(1) 初始化与包含远程对象的远程虚拟机的连接；(2) 对远程虚拟机的参数进行编组汇集(即 marshal,写入并传输)到远程 VM(Virtual Machine)

上；(3) 等待方法调用的结果；(4) 解编（即 unmarshal,读取）返回值或返回的异常；(5) 将值返回给调用者。为了向调用程序展示比较简单的调用机制,stub 将参数的序列化(串行化)和网络级通信等细节隐藏了起来。在远程虚拟机中，每个远程对象都可以有相应的 skeleton（在 JDK1.2 环境中无需使用 skeleton)。Skeleton 负责将调用分配给实际的远程对象实现。它在接收方法调用时执行下列操作：(1) 解编（读取）远程方法的参数；这些信息是由客户方的 stub 汇集生成的(2) 调用实际远程对象实现的方法；(3) 将结果（返回值或异常）编组（写入并传输）给调用者。stub 和 skeleton 由 rmic 编译器生成。

利用 RMI 编写分布式对象应用程序需要完成以下工作：

(1) 定位远程对象。应用程序可使用以下几种机制中的一种得到对远程对象的引用。它既可用 RMI 的简单命名工具 rmiregistry 来注册它的远程对象，也可以将远程对象引用作为常规操作的一部分来进行传递和返回，当然可以使用 Java Naming and Directory Interface（JNDI）查找这些不同的目录服务。

(2) 与远程对象通信。远程对象间通信的细节由 RMI 处理，对于程序员来说，远程通信看起来就像标准的 Java 方法调用。

(3) 给作为参数或返回值传递的对象加载类字节码。因为 RMI 允许调用程序将纯 Java 对象传给远程对象，所以，RMI 将提供必要的机

制，既可以加载对象的代码又可以传输对象的数据。在 RMI 分布式应用程序运行时，服务器调用注册服务程序以使名字与远程对象相关联。客户机在服务器上的注册服务程序中用远程对象的名字查找该远程对象，然后调用它的方法。

3.1.2、对象序列化

在 RMI 分布式应用系统中，服务器与客户机之间传递的 Java 对象必须是可序列化的对象。不可序列化的对象不能在对象流中进行传递。对象序列化扩展了核心 Java 输入/输出类，同时也支持对象。对象序列化支持把对象编码以及将通过它们可访问到的对象编码变成字节流；同时，它也支持流中对象图形的互补重构造。序列化用于轻型持久性和借助于套接字或远程方法调用(RMI)进行的通信。序列化中现在包括一个 API（Application Programming Interface，应用程序接口），允许独立于类的域指定对象的序列化数据，并允许使用现有协议将序列化数据域写入流中或从流中读取，以确保与缺省读写机制的兼容性。

为编写应用程序，除多数瞬态应用程序外，都必须具备存储和检索 Java 对象的能力。以序列化方式存储和检索对象的关键在于提供重新构造该对象所需的足够对象状态。存储到流的对象可能会支持 Serializable（可序列化）或 Externalizable（可外部化）接口。对于 Java 对象，序列化形式必须能标识和校验存储其内容的对象所

属的 Java 类，并且将该内容还原为新的实例。对于可序列化对象，流将提供足够的信息将流的域还原为类的兼容版本。对于可外部化对象，类将全权负责其内容的外部格式。序列化 Java 对象的目的是：提供一种简单但可扩充的机制，以序列化方式维护 Java 对象的类型及安全属性；具有支持编组和解编的扩展能力以满足远程对象的需要；具有可扩展性以支持 Java 对象的简单持久性；只有在自定义时，才需对每个类提供序列化自实现；允许对象定义其外部格式。

3.1.3 分布式应用的实现和运行步骤

编写 Java RMI 分布式应用程序的步骤主要包括以下几步：

(1) 将远程类的功能定义为 Java 接口。在 Java 中，远程对象是实现远程接口的类的实例。在远程接口中声明每个要远程调用的方法。远程接口具有如下特点：1) 远程接口必须声明为 public，如果不这样，则除非客户端与远程接口在同一个包内，否则当试图装入实现该远程接口的远程对象时会得到错误结果。2) 远程对象扩展 java.rmi.Remote 接口。3) 除了所有应用程序特定的异常之外，每个方法还必须抛出 java.rmi.RemoteException(或者 RemoteExcepition 的超类) 异常。4) 任何作为参数或返回值传送的远程对象的数据类型必须声明为远程接口类型，而不是实现类。

(2) 编写和实现服务器类。该类是实现(1)中定义的远程接口。所以在该类中至少要声明实现一个远程接口，并且必须具有构造方

法。在该类中还要实现远程接口中所声明的各个远程方法。

(3) 编写使用远程服务的客户机程序。在该类中使用java.rmi.Naming 中的 lookup()方法获得对远程对象的引用，依据需要调用该引用的远程方法，其调用方式和对本地对象方法的调用相同。

(4) 编译和运行该 RMI 系统。其步骤有：1) 使用 javac 编译远程接口类，远程接口实现类和客户机程序。2) 使用 rmic 编译器生成实现类的 stub 和 skeleton。3) 启动 RMI 注册服务程序 rmiregistry。4) 启动服务器端程序。5) 启动客户机程序。

3．2 目录服务和 JNDI

“网络就是计算机”这是 Sun 公司在销售其大量的硬件和软件产品时提出的口号。在网络中的资源配置信息的获取就变得尤为重要。

名字服务是一种能够为给定的一组数据生成标准的名字的服务，而目录服务是一种特殊类型的数据库，通过使用不同的索引、缓冲存储和磁盘访问技术来优化读访问。简单来说，名称服务提供了一个把文件，打印机，服务器等实体映射到一个逻辑名称的机制。例如在操作系统中的名称服务就把打印机映射到一个 I/O 端口。而目录服务可以理解为名称服务的一个扩展，它允许在服务中的各项拥有自己的属性。又以打印机为例，打印机可以是彩色打印机，支持双面打印，支持网络打印，支持高速打印等。所有这些打印机的属性都可以储存在

目录服务中，和相应的打印机联系起来。

目录服务中的信息采用层次模型来表示。在目前的网络中有许多目录服务正在使用中。其中最常见的目录服务之一是因特网上使用的域名服务（DNS）。组织机构经常使用下列一项或多项通用的目录服务：

- Novell Directory Services(NDS)
- Network Information Services(NIS/NIS+)
- Active Directory Services(ADS)
- Windows NT Domain

由于目录服务很多，而遗憾的是，这些目录服务无法方便地彼此交互，因此传统的应用程序开发者与多种不同的目录服务交互就更困难。解决这些办法之一是使用 JNDI，它为多种目录服务提供了一种标准 API。但是，因为不是说有的人都使用 Java，因此还需要有另外一种办法使得系统之间目录信息得通信更容易。做法就是使用 LDAP。

LDAP 定义了客户如何在服务器上访问数据。但是它并没有指出数据如何在服务器上存储。LDAP 支持引用而且也有一个非常强得安全模型，使用 ACL 来保护服务器内部得数据，并支持安全套接字层（SSL）、传输层安全（TLS）以及简单验证和安全层（SASL）协议。

LDAP 虽然日益流行并且使用得越来越多但距离达到无处不在得地步还有很长得路。另一个问题是企业应用程序经常需要支持现有得分布式计算标准，这就需要用某种服务来定义可用对象得位置。为了

克服这些问题，人们生成了标准 Java API 与名字和目录服务交互，即 JNDI。这个 API 得使用类似于开发者使用 JDBC 与各种类型的数据库交互。

JNDI 对 Java 的长期开发非常重要,特别是 Enterprise JavaBeans (EJB)。EJB 对于 J2EE 非常重要。图 6 介绍了 JNDI 和 LDAP 如何习作以提供最好的解决方案。这里我们使用 JNDI 与不同的 LDAP 服务器通信。开发者只需要关心一种协议（LDAP）和一个 API(JNDI)即可。当然，我们还依靠厂商提供各自专用协议的 LDAP 接口。对于这些流行的目录服务而言，这一点并不成为问题，还有其他产品可供我们通过 LDAP 与他们通信。

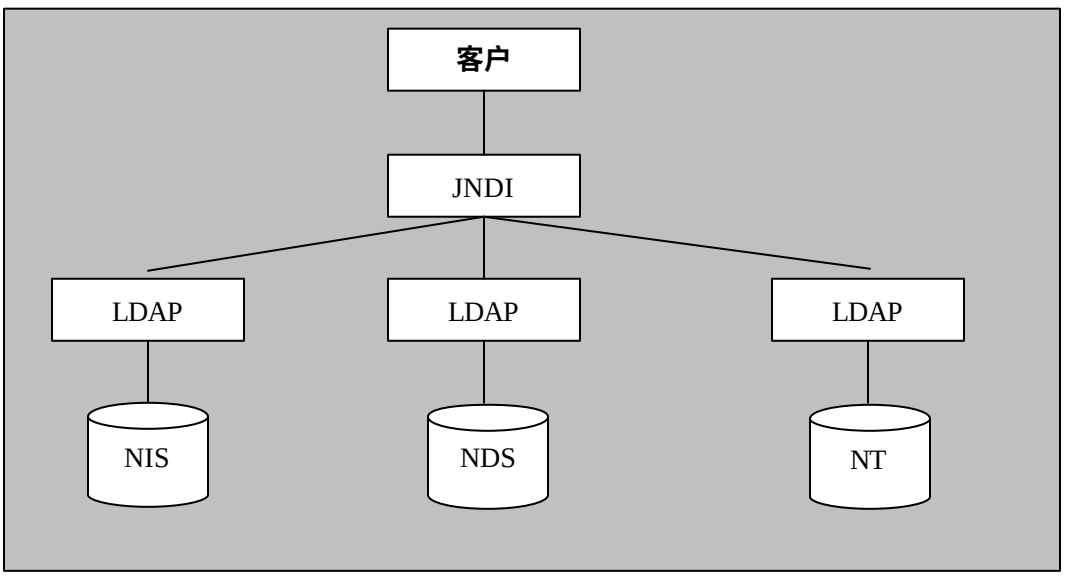


图 5

在 LDAP 中有一些标准操作，如下：

- 连接到 LDAP 服务器

- 访问 LDAP 服务器（可以把此步骤看作是验证）
- 执行一系列 LDAP 操作：
  - 搜索服务器
  - 添加新记录项
  - 修改记录项
  - 删除记录项
- 从 LDAP 服务器断开

3．3 JavaMail

JavaMail 是 Sun 发布的处理电子邮件的应用程序接口,它预置了一些最常用的邮件传送协议的实现方法，并且提供了很容易的方法去调用它们。JavaMail 是 Sun 发布的应用程序接口，所以目前它还没有被 JDK 包含。 因此你需要从 Sun 的官方网站上下载到 JavaMail 类文件包。除此之外还需要 Sun 的 JAF(JavaBeans Activation Framework)，否则 JavaMail 将不能运行。

是不是还没有编一丁点东西就觉得累了一身汗呀？不要着急，这是教程中最难的部分，剩下的内容都是很简单的了。如果你确信每个东西都装好后我们就可以开始了。

HTML 邮件表格

JSP 最大的特色之一就是能把项目分类或划分成许多组件，从而提高了组件的重利用率，也降低了编程的难度。因此我们在编程的时

候也要想着如何把大的问题划分成一个个模块：

模块一 一个 HTML 表单，能把电子邮件信息传送给 JSP 程序

模块二 一个处理和发送邮件的 JSP 页面

本文中将提供一个 HTML 表单，来向 JSP 页面发送信息。复制下面的 HTML 源代码到你的主机上。

代码一：发送电子邮件的 HTML 源代码

```
<html>
<BODY>
<FORM action="sendmail.jsp" method="post">
<TABLE align="center">
<TR>
<TD width="50%">
收件人:<BR><INPUT name="to" size="25">
</TD>
<TD width="50%">
寄信人:<BR><INPUT name="from" size="25">
</TD>
</TR>
<TR>
<TD colspan="2">
```

主题:＜BR＞＜INPUT name="subject" size="50"＞

＜/TD＞

＜/TR＞

＜TR＞

＜TD colspan="2"＞

＜ｐ＞邮件正文:＜BR＞＜TEXTAREA name="text" rows=25 cols=85＞＜/TEXTAREA＞＜／ｐ＞

＜/TD＞

＜/TR＞

＜/TABLE＞

＜INPUT type="submit" name="cb_submit" value="发送"＞

＜INPUT type="reset" name="cb_reset" value="重写"＞

＜/FORM＞

＜/BODY＞

＜/HTML＞

这个示例只包括电子邮件中最重要的信息像"收信人" 、"寄信人"、"主题"和"邮件正文"。而自己的邮件系统的最大的优点就是可以增加你所需要获得的信息，如"抄送"等，完全根据你的需要。

这个 HTML 文档有两个主要的需要根据你的要求修改的量。第一，表单的动作（action）必须用"post"发送到下面一课中要介绍的 JSP

实用程序中，即 sendmail.jsp，你也可以用你系统中的相应的程序来代替它。第二，修改表单，包含你希望用户发送的邮件中必需的字段。

使用 JavaMail 是发送电子邮件所需要的组件 。

JavaMail 的机构使处理电子邮件非常容易。下面列出了一些我们需要的类：

1. Properties

JavaMail 需要 Properties 来创建一个 session 对象。它将寻找字符串"mail.smtp.host"，属性值就是发送邮件的主机，如：

```
Properties props = new Properties ();
props.put("mail.smtp.host", "smtp.abcd.com");//可以换上你的 smtp 主机名。
```

2. Session

这个 Session 类代表 JavaMail 中的一个邮件 session. 每一个基于 JavaMail 的应用程序至少有一个 session 但是可以有任意多的 session。 在这个例子中，Session 对象需要知道用来处理邮件的 SMTP 服务器。为了做到这一点，你可以参照下面的例子用 Properties 来创建一个 Session 对象

```
Session sendMailSession;
sendMailSession = Session.getInstance(props, null);
```

3. Transport

邮件是既可以被发送也可以被受到。JavaMail 使用了两个不同的类来完成这两个功能：Transport 和 Store。 Transport 是用来发送信息的，而 Store 用来收信。对于这的教程我们只需要用到 Transport 对象。Store 的用法请参看 Sun 的 JavaMail 文档。

用法：Transport transport;

transport = sendMailSession.getTransport("smtp");

用 JavaMail Session 对象的 getTransport 方法来初始化 Transport。传过去的字符串申明了对象所要使用的协议，如"smtp"。这将为我们省了很多时间。因为 JavaMail 以境内置了很多协议的实现方法。

注意: JavaMail 并不是绝对支持每一个协议，目前支持 IMAP、SMTP 和 POP3。

4. Message

Message 对象将存储我们实际发送的电子邮件信息，Message 对象被作为一个MimeMessage对象来创建并且需要知道应当选择哪一个 JavaMail session。

使用方法是：Message newMessage = new MimeMessage(sendMailSession);

JavaMail 结合 JSP

1)构建 JSP 程序

前面我们建造了一个 HTML 表单用来发送邮件信息，又介绍了 JavaMail 中的一些对象和方法。现在我们将把这些组件集合起来来构成我们的邮件系统。

第一步也是最重要的一步，确信在 page 指令中导入了需要的类。除了 JavaMail 的相关的类和 JAF 外,不要忘了导入 java.util.date ，因为我们需要它来给邮件盖上时间戳。

<%@ page import= " javax.mail.*, javax.mail.internet.*, javax.activation.*, java.util.*"%>

下一步，创建邮件发送出去的确认信息，如"你的邮件已发送，请返回"

2) 创建并发送 Message 对象

创建 Message 对象的方法我们在第三课中就以讨论过了,我们可以用 Message 来处理消息了，就像在 Message 对象上使用 get 和 set 属性一样简单。在这一部分使用了很多 request.getParameter()。

```
newMessage.setFrom(new
InternetAddress(request.getParameter("from")));
newMessage.setRecipient(Message.RecipientType.TO,
new InternetAddress(request.getParameter("to")));
newMessage.setSubject(request.getParameter("subject"));
newMessage.setSentDate(new Date());
```

newMessage.setText(request.getParameter("text"));

现在终于可以把消息发送出去了：

transport.send(newMessage);

3)完整的程序

上面的都是一些零零碎碎的代码，现在我们把它们写成一个完整的 JSP 程序。注意要捕捉任何错误并把它们显示给用户。

代码二：JavaMail 电子邮件发送系统的 JSP 实现程序代码：

```
<%@ page import="javax.mail.*, javax.mail.internet.*, javax.activation.*,java.util.*"%>
<html>
<head>
<TITLE>JavaMail 电子邮件发送系统</TITLE>
</HEAD>
<BODY>
<%
 try{
     Properties props = new Properties();
     Session sendMailSession;
     Store store;
     Transport transport;
```

```
            sendMailSession = Session.getInstance(props, null);
            props.put("mail.smtp.host", "smtp.abcd.com");
            Message newMessage = new MimeMessage(sendMailSession);
                                                    newMessage.setFrom(new
InternetAddress(request.getParameter("from")));
                newMessage.setRecipient(Message.RecipientType.TO,  new
InternetAddress(request.getParameter("to")));
            newMessage.setSubject(request.getParameter("subject"));
            newMessage.setSentDate(new Date());
            newMessage.setText(request.getParameter("text"));
            transport = sendMailSession.getTransport("smtp");
            transport.send(newMessage);
        %>
        <p>你的邮件已发送，请返回。</p>
        <%
           }catch(MessagingException m){
           out.println(m.toString());
           }
           %>
        </BODY>
```

＜/HTML＞

4、Servlet 和 JSP 技术的网络应用

4．1 Java Servlets

Java servlet 是一种小型、独立于系统平台的服务器程序，用于有计划地扩充 Web 服务器的功能。Java servlet API 提供了用于建立这种 Web 服务器的一个简单的框架。它使应用程序逻辑能够嵌入到 HTTP 请求-响应过程中，在 Java servlet API Specification 2.3 中进行了规定，可以从网址 http://java.sun.com/products/servlet/查到。

Java Servlet 开发工具（JSDK）提供了多个软件包，在编写 Servlet 时需要用到这些软件包。在 j2sdkee1.3.1 其中包括两个用于所有 Servlet 的基本软件包：javax.servlet 和 javax.servlet.http。可从 sun 公司的 Web 站点下载 Java Servlet 开发工具。 下面主要介绍 javax.servlet.http 提供的 HTTP Servlet 应用编程接口。

HTTP Servlet 使用一个 HTML 表格来发送和接收数据。要创建一个 HTTP Servlet，请扩展 HttpServlet 类，该类是用专门的方法来处理 HTML 表格的 GenericServlet 的一个子类。HTML 表单是由 <FORM> 和 </FORM> 标记定义的。表单中典型地包含输入字段（如文本输入字段、复选框、单选按钮和选择列表）和用于提交数据的按钮。当提交信息时，它们还指定服务器应执行哪一个 Servlet（或其它的程序）。 HttpServlet 类包含 init()、destroy()、service() 等方法。其

中 init() 和 destroy() 方法是继承的。

(1) init() 方法

在 Servlet 的生命期中，仅执行一次 init() 方法。它是在服务器装入 Servlet 时执行的。 可以配置服务器，以在启动服务器或客户机首次访问 Servlet 时装入 Servlet。 无论有多少客户机访问 Servlet，都不会重复执行 init() 。

缺省的 init() 方法通常是符合要求的，但也可以用定制 init() 方法来覆盖它，典型的是管理服务器端资源。例如，可能编写一个定制 init() 来只用于一次装入 GIF 图像，改进 Servlet 返回 GIF 图像和含有多个客户机请求的性能。另一个示例是初始化数据库连接。缺省的 init() 方法设置了 Servlet 的初始化参数，并用它的 ServletConfig 对象参数来启动配置，因此所有覆盖 init() 方法的 Servlet 应调用 super.init() 以确保仍然执行这些任务。在调用 service() 方法之前，应确保已完成了 init() 方法。

(2) service() 方法

service() 方法是 Servlet 的核心。每当一个客户请求一个 HttpServlet 对象，该对象的 service() 方法就要被调用，而且传递给这个方法一个"请求"（ServletRequest）对象和一个"响应"（ServletResponse）对象作为参数。在 HttpServlet 中已存在 service() 方法。缺省的服务功能是调用与 HTTP 请求的方法相应的 do 功能。

例如，如果 HTTP 请求方法为 GET，则缺省情况下就调用 doGet() 。Servlet 应该为 Servlet 支持的 HTTP 方法覆盖 do 功能。因为 HttpServlet.service() 方法会检查请求方法是否调用了适当的处理方法，不必要覆盖 service() 方法。只需覆盖相应的 do 方法就可以了。

当一个客户通过 HTML 表单发出一个 HTTP POST 请求时，doPost()方法被调用。与 POST 请求相关的参数作为一个单独的 HTTP 请求从浏览器发送到服务器。当需要修改服务器端的数据时，应该使用 doPost()方法。

当一个客户通过 HTML 表单发出一个 HTTP GET 请求或直接请求一个 URL 时，doGet()方法被调用。与 GET 请求相关的参数添加到 URL 的后面，并与这个请求一起发送。当不会修改服务器端的数据时，应该使用 doGet()方法。

Servlet 的响应可以是下列几种类型：

一个输出流，浏览器根据它的内容类型（如 text/HTML）进行解释。

一个 HTTP 错误响应，重定向到另一个 URL、servlet、JSP。

(3) destroy() 方法

destroy() 方法仅执行一次，即在服务器停止且卸装 Servlet 时执行该方法。典型的，将 Servlet 作为服务器进程的一部分来关闭。缺省的 destroy() 方法通常是符合要求的，但也可以覆盖它，典型的是

管理服务器端资源。例如，如果 Servlet 在运行时会累计统计数据，则可以编写一个 destroy() 方法，该方法用于在未装入 Servlet 时将统计数字保存在文件中。另一个示例是关闭数据库连接。

当服务器卸装 Servlet 时，将在所有 service() 方法调用完成后，或在指定的时间间隔过后调用 destroy() 方法。一个 Servlet 在运行 service() 方法时可能会产生其它的线程，因此请确认在调用 destroy() 方法时，这些线程已终止或完成。

(4) GetServletConfig（）方法

GetServletConfig（）方法返回一个 ServletConfig 对象，该对象用来返回初始化参数和 ServletContext。ServletContext 接口提供有关 servlet 的环境信息。

(5) GetServletInfo（）方法

GetServletInfo（）方法是一个可选的方法，它提供有关 servlet 的信息，如作者、版本、版权。

当服务器调用 sevlet 的 Service（）、doGet（）和 doPost（）这三个方法时，均需要 "请求"和"响应"对象作为参数。"请求"对象提供有关请求的信息，而"响应"对象提供了一个将响应信息返回给浏览器的一个通信途径。javax.servlet 软件包中的相关类为 ServletResponse 和 ServletRequest，而 javax.servlet.http 软件包中的相关类为 HttpServletRequest 和 HttpServletResponse。

Servlet 通过这些对象与服务器通信并最终与客户机通信。Servlet 能通过调用"请求"对象的方法获知客户机环境，服务器环境的信息和所有由客户机提供的信息。Servlet 可以调用"响应"对象的方法发送响应，该响应是准备发回客户机的。

下面的代码显示了一个简单 Servlet 的基本结构。该 Servlet 处理的是 GET 请求，所谓的 GET 请求，如果你不熟悉 HTTP，可以把它看成是当用户在浏览器地址栏输入 URL、点击 Web 页面中的链接、提交没有指定 METHOD 的表单时浏览器所发出的请求。Servlet 也可以很方便地处理POST请求。POST请求是提交那些指定了METHOD=“POST”的表单时所发出的请求。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW2 extends HttpServlet {
   public void doGet(HttpServletRequest request,
                                HttpServletResponse response)
       throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println(ServletUtilities.headWithTitle("Hello WWW") +
```

```
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>"=;
    }
}
```

如果某个类要成为 Servlet，**则它应该从** HttpServlet **继承，根据数据是通过** GET **还是** POST **发送，覆盖** doGet、doPost **方法之一或全部。**doGet **和** doPost **方法都有两个参数，分别为** HttpServletRequest **类型和** HttpServletResponse **类型。**HttpServletRequest **提供访问有关请求的信息的方法，例如表单数据、**HTTP **请求头等等。**HttpServletResponse **除了提供用于指定** HTTP **应答状态(**200，404 **等)、应答头（**Content-Type，Set-Cookie **等）的方法之外，最重要的是它提供了一个用于向客户端发送数据的** PrintWriter **。对于简单的** Servlet **来说，它的大部分工作是通过** println **语句生成向客户端发送的页面。**

注意 doGet **和** doPost **抛出两个异常，因此你必须在声明中包含它们。另外，你还必须导入** java.io **包（要用到** PrintWriter **等类)、**javax.servlet **包(要用到** HttpServlet **等类)以及** javax.servlet.http **包(要用到** HttpServletRequest **类和** HttpServletResponse **类)。**

最后，doGet **和** doPost **这两个方法是由** service **方法调用的，有时你可能需要直接覆盖** service **方法，比如** Servlet **要处理** GET **和** POST

两种请求时。

4．2 JSP

如果你是直接使用 Java servlets，那你将不得不在 Java 类中处理 HTTP 输入和 HTML 输出，你需要丰富的 Java 编程经验来构建复杂的应用程序。JSP 的加入，使你可以把 HTML 的表达逻辑从植入 servlets 中的复杂的商务逻辑区分开来。这意味着可以由有经验的脚本编写者来编写表达层代码，而高级的 Java 开发者能够集中精力去解决 servlets 和 bean 中更为复杂的问题。

不管你有没有 Java 编程知识，都能够使用 JSP。JSP 包含了一些服务器端的标签，使得不用写一行 Java 代码就能显示动态数据。你可以直接访问 bean 来完成操作，然后使用 JSP 标签把结果显示为动态内容。你还可以用 servlets 生成 bean，servlets 操作的运算结果存于其中，然后再使用 JSP 标签显示结果，同样不需要在 JSP 页中写 Java 代码。

有三种方式可以用来在你的网页中加入 Java 代码：

1）使用 declarations(声明)，可以定义全局变量或是在页内任何地方都可以访问的 Java 方法。声明被包含在标记<%!...%>中。

2）使用 scriptlets（脚本片断），你能书写页内处理所需的任何逻辑，它们包含在<%...%>标记内。

3）Expressions（表达式），包含于<%=...%>中。它提供一种简单

的方法来显示 Java 表达式的结果。被附加上的表达式将被计算并在页面上显示出来，就好像你已经在代码中明确写出了运算结果的数值一样。

在你自己编写的代码中，可以使用一些隐含变量（implicit variables）——JSP 提供的预定义的 Java 对象。另外，通过使用 JSP 的指令（directives），还可以包含非 Java 代码模块，比如来自其他文件的 HTML 文本。

下面我们来仔细看一看这些脚本元素，在编写你自己的 JSP 脚本时将会经常用到它们。

Directives（指令）JSP 定义了三个页内指令用于设置 JSP 参数或扩充代码。它们是 page,include 和 taglib，必须写在 JSP 页的第一行。语法如下：

```
<%@ directive attribute="value" ... %>
```

page 指令允许你为网页设定一些基本参数，包括设置所用脚本语言的参数（默认为 Java）、你的脚本片断中引入的 Java 类、设置输出缓冲区等等。

使用 include 指令，可以包含其他文件的内容，比如存于单独文件中的 HTML 报头和页脚。

taglib 指令用于扩充标准的 JSP 标签集，这超出了本文的讨论范围。然而，了解 JSP 定义了一种扩充其标签集的方法还是很有好处的，

当你是一个软件商,想扩充JSP的原始功能而又不想破坏其兼容性时,这一点尤为重要。

Declarations（声明）使用 declarations，你可以在 JSP 页中定义方法或变量,它们可被同一页中的其他代码访问。在大多数情况下，你可能会在自己的 bean 中定义方法。然而，有时候在网页内定义方法可能更方便一些，尤其是当代码只用于单一页面时。不论定义方法还是变量，声明都包含在<%！%>标记内。 注意，声明并不在 JSP 页内产生任何输出。它们仅仅用于定义，而不生成输出结果。要生成输出结果，你应该用 JSP 表达式或脚本片断。

Expressions(表达式)Expressions 是一种非常简单的 JSP 标签，它用来把在<%= %>中定义的 JSP 表达式的值转换成字串并将这个值以动态文本的形式送出。Expression 的确是一条生成文本的捷径，有了它,你不必在每次要显示一段动态文本的时候都去调用 print()方法。典型的应用就是，你可以用 expressions 显示简单的变量值或 bean 中的方法的返回值。

例如，下面的代码将会生成 getName()方法的返回值：

<H2>Welcome, <%= mybean.getName() %></H2>

事实上，在生成动态输出之前，JSP 必须把方法的返回值转变为 Java 中的 String 对象。JSP 规范中详细描述了在 JSP 表达式中，什么样的 Java 类型和 Java 类会被转变成字串。

Scriptlets（脚本片断）到现在为止你已经学会了使用指令来引入任何 Java 类或 Java 包，你能定义页面级的方法或变量并在页中使用它们,你还可以使用提供普通 web 处理功能的隐含变量。还能在 JSP 页内做些什么就取决于你了，因为你可以在 scriptlets（脚本片断）里编写任何你想要的 Java 代码，如下所示：

<% ...code... %>

通过在 page 指令中使用 IMPORT 参数，你可以从脚本片断内调用所有 Java API。因为你写的所有 JSP 代码实际上都被编译构成 Java servlet,它本身就是一个 Java 类,所以你所用的语言本身就是 Java，而不是任何一种修改或整理过的版本。这就像在 SSJS 中你可以编写任何代码一样。而与 SSJS 不同,在 JSP 中你有权使用整套丰富的 Java API，因此几乎没有任何局限性。

Implicit Variables（隐含变量）JSP 定义了一些隐含变量（即 Java 对象）供你在表达式和脚本片断中使用。这里列出一些常用的对象：

out 对象，类型为 javax.servlet.jsp.JspWriter，提供对方法（例如 print()方法）的访问，用来在脚本片断内生成输出结果。

request 对象直接与 Java 中的 javax.servlet.http.HttpServletRequest 类对应，具有该类的对象的一切属性和方法。举个例子，要获取一个从 HTML 表单或 URL 查询

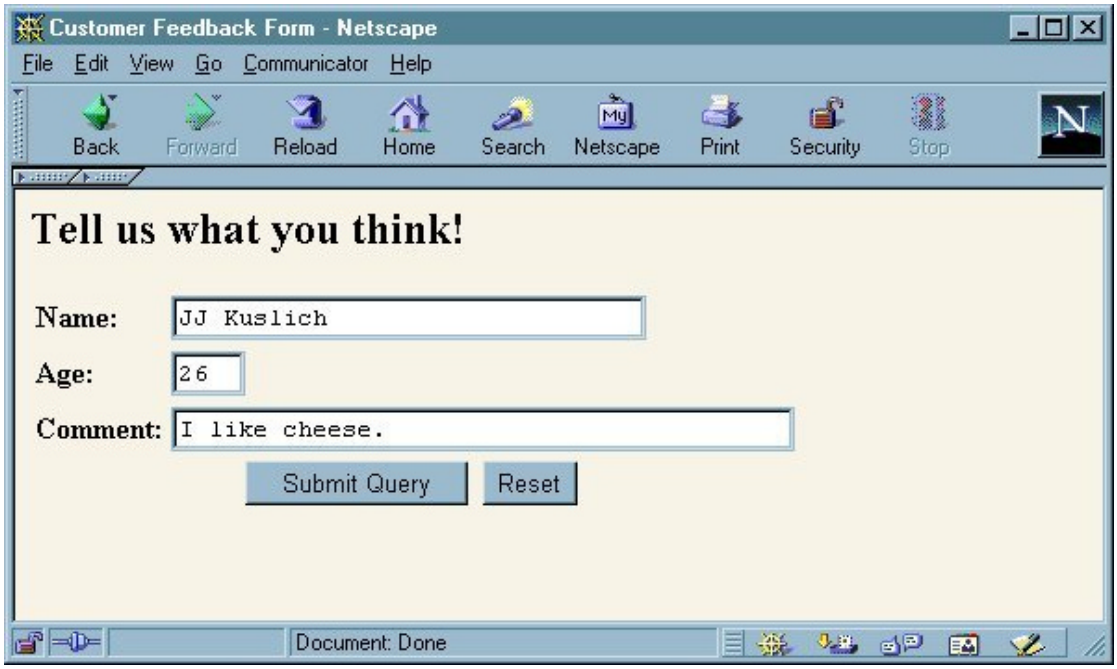
字串传入的值，可以调用 request.getParameter()方法，根据名字获取参量。

response 对象与 Java 中的 javax.servlet.http.HttpServletResponse 类对应,提供对你的网页产生的 HTML 响应的参数的访问权。因此，要在 JSP 页返回的 HTML 响应报头中加入一个值，你就可以调用 the response.setHeader() 方法来实现。

一个简单的例子 ：

在下面的例子中，我们来看一看一个表单和它的 JSP 表单句柄之间的交互过程。使用前面讨论过的脚本元素,我实现了一个简单的 web 站点回馈表单（见图 1）和一个 JSP 表单句柄用来验证输入，然后有条件地地生成基于回馈的输出。

图 1. 一个 web 站点回馈表单



图中按钮：submit query——**提交；**reset? ? **重填**

表单句柄将会检验名称和意见栏以确定它们已被填写，如果其中任何一个或两个是空白的，表单句柄会生成一条错误信息；否则它将继续查看用户意见是否与预先设定的字串匹配。如果匹配，它就输出一条专门的信息；否则输出“ thank you ”。

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="NetObjects ScriptBuilder 2.01">
<TITLE>Feedback Results</TITLE>
</HEAD>
<%!
// 姓名和意见栏不能为空白
// 检查它们的值并返回结果
boolean validateInput(String name, String comment) {
boolean result = true;
// 如果姓名或意见未填写，返回 false 以表明输入无效
if (name.length() == 0)
result = false;
if (comment.length() == 0)
result = false;
return result;
```

```
} // 结束输入验证 validateInput
// 根据表单上的意见栏输出结果
String getStringCheese (String comment) {
String cheese = "I like cheese.";
String result;
if (comment.compareTo(cheese) == 0)
result = "We like cheese too!";
else
result = "We hope someday you'll learn to like cheese.";
return result;
} //结束 getStringCheese
%>
<BODY BGCOLOR="#F0F0E0">
<%
// 获取通过表单提交的数据
String name = request.getParameter("name");
String age = request.getParameter("age");
String comment = request.getParameter("comment");
boolean isValid;
isValid = validateInput(name, comment);
// 根据用户是否未填写姓名或意见栏决定输出内容
if (isValid) {
%>
<H2>Thank you for your feedback!</H2>
<H3>
<%
//输出意见栏的查询结果
out.println(getStringCheese(comment));
} // 结束 if 程序段
else {
out.println("You didn't give us your name or a comment.");
%>
</H3>
Please <a href="feedback_form.html">try again</a>
<%
} //结束 else 程序段
%>
</BODY>
</HTML>
```

这个例子假定用户输入的意见是“I like cheese."（我喜欢奶酪）

在代码中可以看到，这一响应是为填写这条意见的用户定制的。表单句柄将会返回如图 2 所示的页面：

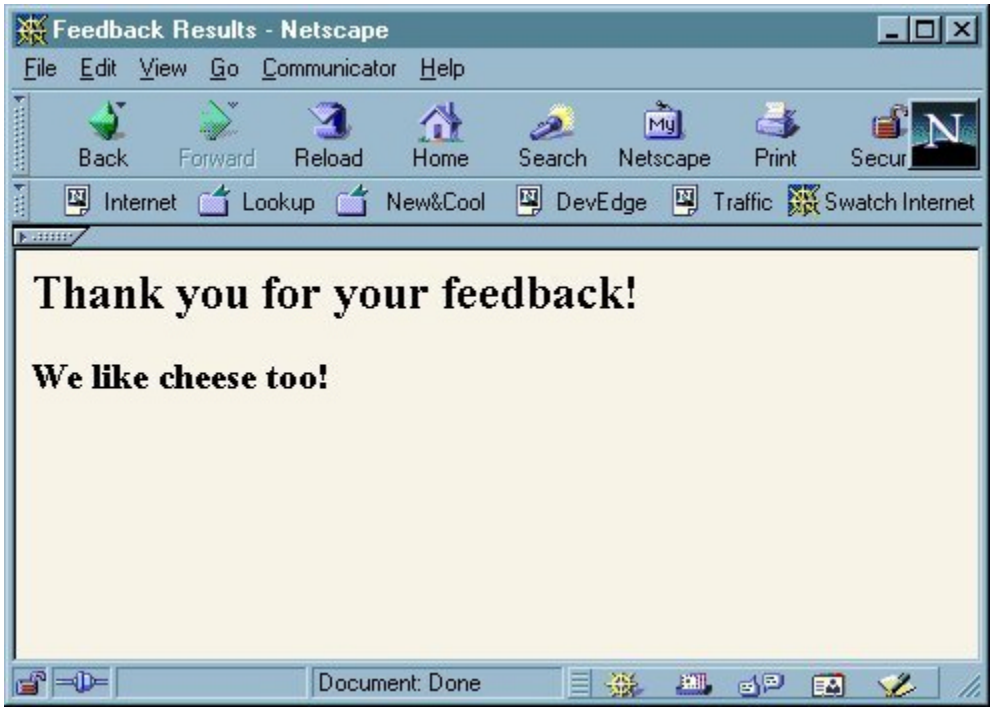


图 2：表单句柄输出 图中文字：谢谢你的反馈！我们也喜欢奶酪。

这个例子非常简单易懂。即便你只是一个 JavaScript 程序员，你也应该可以理解它。我还要指出这个例子中体现的在 JSP 规范中并不很明显的一些特性。首先，请注意我在声明部分（<%'...%>中的部分）定义了一些方法，与在 Java 类中定义方法一模一样。这是因为 JSP 引擎把这些方法转变为底层的 Java servlets，在浏览器向网页发出请求时由服务器来执行它们。因此，任何变量和方法的定义都必须遵守标准的 Java 语法。

还应注意到，在我的脚本片断的代码中，我把一个 if...else 语句分开了，它跨越了两个不同的脚本片断段。这完全是合法的！不仅

合法，而且把脚本片断代码和静态 HTML 交叉起来是有条件生成 HTML 的好办法，就像我在本例中所做到的一样。

最后，你可以看到我通过调用 request.getParameter()方法取得表单元素的值并把它赋给一个临时变量。这是处理从表单或查询字串输入的值的标准方法。

5、EJB 核心及其应用

5．1 EJB 技术简介

EJB 的全称是 Enterprise java bean。是 JAVA 中的商业应用组件技术。EJB 结构中的角色 EJB 组件结构是基于组件的分布式计算结构，是分布式应用系统中的组件。

一个完整的基于 EJB 的分布式计算结构由六个角色组成，这六个角色可以由不同的开发商提供，每个角色所作的工作必须遵循 Sun 公司提供的 EJB 规范，以保证彼此之间的兼容性。这六个角色分别是 EJB 组件开发者(Enterprise Bean Provider) 、应用组合者(Application Assembler)、部署者(Deployer)、EJB 服务器提供者(EJB Server Provider)、EJB 容器提供者(EJB Container Provider)、系统管理员(System Administrator)。

5．2 EJB 中各角色的分析

1）EJB 组件开发者(Enterprise Bean Provider)

EJB 组件开发者负责开发执行商业逻辑规则的 EJB 组件，开发出

的 EJB 组件打包成 ejb-jar 文件。EJB 组件开发者负责定义 EJB 的 remote 和 home 接口，编写执行商业逻辑的 EJB class,提供部署 EJB 的部署文件(deployment descriptor)。部署文件包含 EJB 的名字，EJB 用到的资源配置，如 JDBC 等。EJB 组件开发者是典型的商业应用开发领域专家。

EJB 组件开发者不需要精通系统级的编程，因此，不需要知道一些系统级的处理细节，如事务、同步、安全、分布式计算等。

2）应用组合者(Application Assembler)

应用组合者负责利用各种 EJB 组合一个完整的应用系统。应用组合者有时需要提供一些相关的程序，如在一个电子商务系统里，应用组合者需要提供 JSP(Java Server Page)程序。

应用组合者必须掌握所用的 EJB 的 home 和 remote 接口，但不需要知道这些接口的实现。

3）部署者(Deployer)

部署者负责将 ejb-jar 文件部署到用户的系统环境中。系统环境包含某种 EJB Server 和 EJB Container。部署者必须保证所有由 EJB 组件开发者在部署文件中声明的资源可用，例如，部署者必须配置好 EJB 所需的数据库资源。

部署过程分两步：部署者首先利用 EJB Container 提供的工具生成一些类和接口，使 EJB Container 能够利用这些类和接口在运行状

态管理 EJB。 部署者安装 EJB 组件和其他在上一步生成的类到 EJB Container 中。 部署者是某个 EJB 运行环境的专家。

某些情况下，部署者在部署时还需要了解 EJB 包含的业务方法，以便在部署完成后，写一些简单的程序测试。

4）EJB 服务器提供者(EJB Server Provider)

EJB 服务器提供者是系统领域的专家，精通分布式交易管理，分布式对象管理及其它系统级的服务。EJB 服务器提供者一般由操作系统开发商、中间件开发商或数据库开发商提供。

在目前的 EJB 规范中，假定 EJB 服务器提供者和 EJB 容器提供者来自同一个开发商，所以，没有定义 EJB 服务器提供者和 EJB 容器提供者之间的接口标准。

5）EJB 容器提供者(EJB Container Provider)

EJB 容器提供者提供以下功能：

提供 EJB 部署工具为部署好的 EJB 组件提供运行环境 。EJB 容器负责为 EJB 提供交易管理，安全管理等服务。

EJB 容器提供者必须是系统级的编程专家，还要具备一些应用领域的经验。EJB 容器提供者的工作主要集中在开发一个可伸缩的，具有交易管理功能的集成在 EJB 服务器中的容器。EJB 容器提供者为 EJB 组件开发者提供了一组标准的、易用的 API 访问 EJB 容器，使 EJB 组件开发者不需要了解 EJB 服务器中的各种技术细节。

EJB 容器提供者负责提供系统监测工具用来实时监测 EJB 容器和运行在容器中的 EJB 组件状态。

6）系统管理员(System Administrator)

系统管理员负责为EJB服务器和容器提供一个企业级的计算和网络环境。

系统管理员负责利用 EJB 服务器和容器提供的监测管理工具监测 EJB 组件的运行情况。

5．3 EJB 的体系结构

EJB 分布式应用程序是基于对象组件模型的，低层的事务服务用了 API 技术。EJB 技术简化了用 JAVA 语言编写的企业应用系统的开发，配置。EJB 技术定义了一组可重用的组件：Enterprise Beans。你可以利用这些组件，象搭积木一样的建立你的分布式应用程序。当你把代码写好之后，这些组件就被组合到特定的文件中去。每个文件有一个或多个 Enterprise Beans，在加上一些配置参数。最后，这些 Enterprise Beans 被配置到一个装了 EJB 容器的平台上。客户能够通过这些 Beans 的 home 接口，定位到某个 beans，并产生这个 beans 的一个实例。这样，客户就能够调用 Beans 的应用方法和远程接口。

EJB 服务器作为容器和低层平台的桥梁管理着 EJB 容器和函数。它向 EJB 容器提供了访问系统服务的能力。例如：数据库的管理和事务的管理,或者对于其它的 Enterprise 的应用服务器。所有的 EJB 实

例都运行在 EJB 容器中。　　容器提供了系统级的服务，控制了 EJB 的生命周期。EJB 中的有一些易于使用的管理工具如：Security--配置描述器（The Deployment descriptor）定义了客户能够访问的不同的应用函数。容器通过只允许授权的客户访问这些函数来达到这个效果。Remote Connectivity--容器为远程链接管理着低层的通信 issues,而且对 Enterprise Beas 的开发者和客户都隐藏了通信细节。EJB 的开发者在编写应用方法的时候，就象是在条用本地的平台一样的。客户也不清楚他们调用的方法可能是在远程被处理的。Life Cycle managment--客户简单的创建一个 Enterprise beans 的实例，并通常取消一个实例。而容器管理着 Enterprise Beans 的实例，使 Enterprise Beans 实现最大的效能和内存利用率。容器能够这样来激活和使 Enterprise Beans 失效，保持众多客户共享的实例池。等等。Trasction management-配置描述器定义了 Enterprise beans 的事务处理的需求。容器管理着那些管理分布式事务处理的复杂的 issues。这些事务可能要在不同的平台之间更新数据库。容器使这些事务之间互相独立，互不干扰。保证所有的更新数据库都是成功发生的，否则就回滚到事务处理之前的状态。

EJB 组件是基于分布式事务处理的企业级应用程序的组件。所有的 EJB 都有如下的特点：EJB 包含了处理企业数据的应用逻辑。定义了 EJB 的客户界面。这样的界面不受容器和服务器的影响。于是，当

一个 EJB 被集合到一个应用程序中去时，不用更改代码和重新编译。EJB 能够被定制 各种系统级的服务，例如安全和事务处理的特性，都不是属于 EJB 类的。而是由配置和组装应用程序的工具来实现。 有两种类型的 EJB: Session beans 和 entity beans.Session beans 是一种作为单用户执行的对象。作为对远程的任务请求的相应，容器产生一个 Session beans 的实例。一个 Session beans 有一个用户.从某种程度上来说，一个 Session bean 对于服务器来说就代表了它的那个用户.Session beans 也能用于事务，它能够更新共享的数据，但它不直接描绘这些共享的数据。Session beans 的生命周期是相对较短的。典型的是，只有当用户保持会话的时候，Session beans 才是活着的。一旦用户退出了，Session beans 就不再与用户相联系了。Session beans 被看成是瞬时的，因为如果容器崩溃了，那么用户必须重新建立一个新的 Session 对象来继续会话。

Session bean 典型的声明了与用户的互操作或者会话。也就是说，Session bean 了在客户会话期间，通过方法的调用，掌握用户的信息。一个具有状态的 Session bean 称为有状态的 Session bean.当用户终止与 Session beans 互操作的时候.会话终止了，而且，bean 也不再拥有状态值。Session bean 也可能是一个无状态的 session bean.无状态的 Session beans 并不掌握它的客户的信息或者状态。用户能够调用 beans 的方法来完成一些操作。但是，beans 只是在方

法调用的时候才知道用户的参数变量。当方法调用完成以后，beans 并不继续保持这些参数变量。这样，所有的无状态的 session beans 的实例都是相同的，除非它正在方法调用期间。这样，无状态的 Session beans 就能够支持多个用户.容器能够声明一个无状态的 Session beans.能够将任何 Session beans 指定给任何用户.

Entity Beans 对数据库中的数据提供了一种对象的视图。例如：一个 Entity bean 能够模拟数据库表中一行相关的数据。多个 client 能够共享访问同一个 Entity bean.多个 client 也能够同时的访问同一个 Entity bean.Entity beans 通过事务的上下文来访问或更新下层的数据。这样，数据的完整性就能够被保证。Entity Beans 能存活相对教长的时间，并且状态是持续的。只要数据库中的数据存在，Entity beans 就一直存活。而不是按照应用程序或者服务进程来说的。即使 EJB 容器崩溃了，Entity beans 也是存活的。Entity Beans 生命周期能够被容器或者 Beans 自己管理。如果由容器控制着保证 Entity beans 持续的 issus。如果由 Beans 自己管理,就必须写 Entity beans 的代码，包括访问数据库的调用。

Entity Beans 是由主键（primary key 一种唯一的对象标识符）标识的。通常，主键与标识数据库中的一块数据，例如一个表中的一行，的主键是相同的。主键是 client 能够定位特定的数据块。

5．4 开发 EJB

1）类介绍：

开发 EJB 的主要步骤一般来说，整个的开发步骤（开发，配置，组装）包括如下几个方面。开发：首先要定义三个类：Bean 类本身，Bean 的本地和远程接口类。 配置：配置包括产生配置描述器--这是一个 XML 文件、声明了 Enterprise Bean 的属性、绑定了 bean 的 class 文件（包括 stub 文件和 skeleton 文件）。最后将这些配置都放到一个 jar 文件中。还需要在配置器中定义环境属性。组装应用程序：包括将 Enterprise beans 安装到 Server 服务器中，测试各层的连接情况。程序组装器将若干个 Enterprise Beans 与其它的组件结合起来。组合成一个完整的应用程序。或者将若干个 Enterprise beans 组合成一个复杂的 Enterprise Bean。管理 Enterprise Bean。

我们必须定义和编写一些 EJB 中的基本类。如 Enterprise bean 类：这是 Enterprise bean 内部应用逻辑的实现。编写 Enterprise bean 的远程接口类。编写 Enterprise bean 的本地接口类。说明主键类，主键类只是对于 Entity bean 才需要的。在 Enterprise bean 的配置描述器中指定主键的名字。Enterprise beans 提供者定义了远程接口和本地接口，实现了 EJB 类本身。Remote 接口中提供了客户调用 EJB 实现的应用逻辑函数的接口。而 home 接口提供了产生和定位 remote 接口实例的方法。

在 Enterprise bean 本身类的实现，本地 home 接口，远程 remote

接口之间并没有正式的联系（例如继承关系）。但是，在三个类里声明的方法却必须遵守 EJB 里面定义的规范。例如： 你在 Enterprise bean 里面声明了一个应用程序的方法或者说应用逻辑。也在 beans 的 remote 接口中声明了这个方法，那么，这两个地方必须要同样的名字。Bean 的实现里面必须至少有一个 Create()方法 :ejbCreate()。但是可以有多个带有不同参数的 create()方法。 在 home 接口中，也必须有相同的方法定义（参数的个数相同）。EjbCreate()方法返回的一个容器管理的持久对象。它们都返回一个容器管理持久性的主键值。但是，在 home 的相应的 Create()方法中返回值的类型是 remote 接口。

注意：实体 bean 的实现的 ejbCreate 方法有点不同。实体 bean 可以不定义 ejbCreate 方法。如果实体只是通过应用程序或通过数据库管理程序的途径被加到数据库中，实体 bean 就省略了 ejbCreate 方法。EjbCreate 返回的值是主键类型。如果 ejbCreate 方法是容器管理持久性的实体 bean 的方法，它的返回值就是 NULL 类型。如果实体 bean 实现了 Bean 管理的持久性，ejbCreate 方法就返回值类型就是主键类型。容器的任务是把各接口和 Enterprise bean 的实现类结合起来。保证在编译时和运行时，各接口和实现类是相对应的。

EJB的实现类,各接口要从不同的基类中继承下来。一个会话bean必须实现基类 javax.ejb.SessionBean。而实体 bean 必须实现基类

javax.ejb.EntiyBean 。这些 EJB 的基类都是从 javax.ejb.EnterpriseBean继承而来。而javax.ejb.EnterpriseBean 又是从 java.io.Serializable 继承而来。每一个 Enterprise Bean 都必须有一个 remote 接口。Remote 接口定义了应用程序规定客户可以调用的逻辑操作。这些是一些可以由客户调用的公共的方法，通常由 Enterprise beans 类来实现。注意，Enterprise bean 的客户并不直接访问 Bean。而是通过 remote 接口来访问。Enterprise bean 类的 remote 接口扩展了 javax.ejb.EJBObject 类的公共 java 接口。而 Javax.ejb.EJBObject 是所有 remote 接口的基类。其代码如下：

```
package javax.ejb;
public interface EJBObject extends java.rmi.Remote{
public EJBHome getEJBHome() throws java.rmi.RemoteException;
public Object getPrimaryKey() throws java.rmi.RemoteException;
public void Remove() throws java.rmi.RemtoeException, java.rmi.RemoveException
public Handle getHandle() throws java.rmi.RemoteException;
boolean isIdentical (EJBObject p0) throws java.rmi.RemoteException;
}
```

getEJBHome()方法允许你取得一个相关的 Home 接口。对于 实体 Bean，用 getPrimaryKey （） 方法获得实体 Bean 的主键值。Remove （） 可以删除一个 Enterprise bean。具体的语义在各种不同类型的 enterprise beans 的生命周期中，由上下文中解释的。方法 getHandle （）返回了一个 Enterprise bean 实例的持久的句柄。IsIndentical() 方法允许你去比较 Enterprise beans 是否相同。

2）方法：

所有的 remote 接口中的方法必须声明为公共（public）的，并必须抛出 java.rmi.RemotException 异常。另外，所有的 remote 接口中的方法定义的参数和都必须是在 RMI-IIOP 中有效的。对每一个在 remote 接口中定义的方法，在 Enterprise bean 类里面都要有相应的方法。相应的方法必须要有同样的名字，同样类型和数量的参数，同样的返回值，而且还要抛出同样的例外。 如下代码显示了一个 ATM 例子的会话 bean 的 remote 接口 Atm,。里面声明了一个应用方法 transfer （）。黑体部分表示 EJB 规范中必须要有的内容。Remote 接口必须扩展 javax.ejb.EJBObject 类。从客户端调用的 Enterprise bean 的每一个方法都必须在 remote 接口中声明。Transfer （） 方法抛出了两个意外。其中 InSufficientFundsException 例外是应用程序定义的意外。

```
Public interface Atm extends javax.ejb.EJBObject{
```

```
Public void transfer(String Source, String Target, float
amount)
Throws java.rmi.RemoteException, InSufficientFundsException;
}
```

Home 接口必须定义一个或多个的 Create()方法。每一个这样的 Create()方法都必须命名为 Create。并且，它的参数，不管是类型还是数量都必须与 bean 类里面的 ejbCreate()方法对应。注意，home 接口中的 Create()方法和 bean 类中 ejbCreate()方法的返回值类型是不同的。实体 bean 的 home 接口还包含 find () 方法。每一个 Home 接口都扩展了 javax.ejb.EJBHome 接口。如下代码显示了 javax.ejb.EJBHome 接口的定义：

```
package javax.ejb;
public interface EJBHome extends java.rmi.Remote() {
void remove(Handle handle) throws
java.rmi.RemoteException,RemoveException;
void remove(Object primarykey) throws
java.rmi.RemoteException,RemoveException;
EJBMetaData getEJBMetaData() throws RemoteException;
Homehandle getHomeHandle() throws RemoteException;
}
```

这里提供了两个 remove () 方法来删除 Enterprise bean 的实例。第一个 remove 方法是通过句柄来删除一个 Enterprise bean 的实例。第二个remove方法通过主键来删除一个Enterprise bean的实例。在众多的 Enterprise bean 实例中，句柄唯一的标识一个实例。一个句柄与它引用的 Enterprise bean 有相同的生命期。考虑一个实体对象，客户可以通过一个句柄来重新获得相应的 Enterprise bean 的实例。一个句柄能够对应一个 Enterprise bean 对象的多个实例。例如，即使当 Enterprise bean 对象所在的主机崩溃了,或者 Enterprise bean 对象在不同的机器之间移动，句柄仍是有效的。这里的句柄是 Serialized 句柄,与 CORBA 中的字符串化的 CORBA 对象的引用是相似的概念。在 EJBHome 接口中的第二个 remove 操作通过其主键来决定要删除的 Enterprise bean。主键可以是扩展了 Java Object 类的任何类型，但是，必须要实现 Java 的 Serializable 接口。主键是标识实体 bean 的主要的方法。通常，主键是数据库中的一个关键字，唯一的定义了由实体 bean 代表的数据。

方法getEJBMetaData()返回了Enterprise bean对象的metadata接口。这个接口允许客户获得 Enterprise bean 的 metadata 信息。当开发工具来编译链接应用程序的时候，或者配置工具来配置的时候，可能会用到 metadata 信息。Javax.ejb.EJBMetadata 接口提供了获得 javax.ejb.EJBHome 接口，home 类，remote 接口，还有获得主

键的方法。也提供了一个 isSesson()的方法来确定在放这个 home 接口的对象是会话 bean 还是实体 bean。 IsStatelessSession()方法指示这个会话 bean 是有状态还是无状态的。如下代码显示了 javax.ejb.EJBMetadata 接口的定义部分的代码。

```
Public javax.ejb; Public interface EJBMetaData{
EJBHome getEJBHome();
Class getHomeInterfaceClass();
Class getRemoteInterfaceClasss();
Class getPrimaryKeyClass();
Boolean isSession();
Boolean isStatelesssSession();
}
```

对每一个 Create（）方法，EJB 规范定义了如下的命名约定。它的返回值是会话 bean 的 remote 接口的类型。方法的名字只能是 Create()。对会话 bean 类中的每一个 ejbCreate()方法都必须有一个 Create()与之对应。 对于每一个 Create()方法的参数的类型和数量都必须与会话 bean 类中的 ejbCreate()方法相对应。方法必须抛出 java.rmi.RemoteException 例外。 方法必须抛出 javax.rmi.CreateExeption 例外。 Create（）方法的参数是用来初始化新的会话 bean 对象的。 如下代码显示了一个会话 bean 对象的

不同的 Create()方法，其中必须的部分用粗体显示：

```
public interface AtmHome extends javax.ejb.EJBHome{
Atm                    create()                    throws
java.rmi.RemoteException,javax.ejb.CreateException;
Atm create(Profile preferredProfile)
Throws                 java.rmi.RemoteExeption,javax.ehrows
java.rmi.RemoteException,RemoveException;
EJBMetaData getEJBMetaData() throws RemoteException;
Homehandle getHomeHandle() throws RemoteException;
}
```

这里提供了两个 remove () 方法来删除 Enterprise bean 的实例。第一个 remove 方法是通过句柄来删除一个 Enterprise bean 的实例。第二个 remove 方法通过主键来删除一个 Enterprise bean 的实例。在众多的 Enterprise bean 实例中，句柄唯一的标识一个实例。一个句柄与它引用的 Enterprise bean 有相同的生命期。考虑一个实体对象，客户可以通过一个句柄来重新获得相应的 Enterprise bean 的实例。一个句柄能够对应一个 Enterprise bean 对象的多个实例。例如，即使当 Enterprise bean 对象所在的主机崩溃了，或者 Enterprise bean 对象在不同的机器之间移动，句柄仍是有效的。这里的句柄是 Serialized 句柄,与 CORBA 中的字符串化的 CORBA 对象的引用是相似

的概念。

在 EJBHome 接口中的第二个 remove 操作通过其主键来决定要删除的 Enterprise bean。主键可以是扩展了 Java Object 类的任何类型，但是，必须要实现 Java 的 Serializable 接口。主键是标识实体 bean 的主要的方法。通常，主键是数据库中的一个关键字，唯一的定义了由实体 bean 代表的数据。方法 getEJBMetaData（）返回了 Enterprise bean 对象的 metadata 接口。这个接口允许客户获得 Enterprise bean 的 metadata 信息。当开发工具来编译链接应用程序的时候，或者配置工具来配置的时候，可能会用到 metadata 信息。Javax.ejb.EJBMetadata 接口提供了获得 javax.ejb.EJBHome 接口，home 类,remote 接口,还有获得主键的方法。也提供了一个 isSesson()的方法来确定在放这个 home 接口的对象是会话 bean 还是实体 bean。IsStatelessSession()方法指示这个会话 bean 是有状态还是无状态的。如下代码显示了 javax.ejb.EJBMetadata接口的定义部分的代码。

```
Public javax.ejb;
Public interface EJBMetaData{
EJBHome getEJBHome();
Class getHomeInterfaceClass();
Class getRemoteInterfaceClasss();
Class getPrimaryKeyClass();
```

```
Boolean isSession();
Boolean isStatelesssSession();
}
```

5．5 EJB 的编程环境

1） 使用 Jbuilder

Jbuilder 与 EJB Container 能够进行无缝连接。Jbuilder 和 Inprise 的应用服务器包括了所有的开发和配置 Enterprise Beans 的工具以及所需要的库：运行和管理 Enterprise Bean 的容器、命名服务、 事务服务、Java 数据库、开发 Enterprise Beans 所需要的 API、一个增强的 java-to-iiop 编译器，支持值类型和 RMI 信号等等。

Jbuilder 还提供了一个快速开发应用程序 Enterprise Beans 的工具和向导。通过简单而且直观的步骤，向导帮助你建立一个 Enterprise Bean。自己设定某些缺省值，产生了 bean 的模板，在上面，我们可以增加我们自己的应用逻辑。Jbuilder 也提供了一个 EJB 的接口生成向导。向导在 Enterprise Bean 的公共方法基础上生成了 Remote 接口和 Home 接口。Jbuilder 还提供一个配置器的向导帮助我们逐步的建立 XML 描述器文件。并将生成的 Stubs 集中到一个 jar 文件中。

2）使用 Jbuilder 之外的集成环境：

如果你使用其它的除了别的集成环境（IDE)。要确定使用了集成

环境 IDE 所带的容器工具。也要验证 IDE 是否支持 EJB 规范的相应的版本，还要确定它是否正确的支持 EJB 的 API。

要确定 JD 到所支持的 EJB 容器的版本。可以通过检查 Inprise 的安装说明来确定 EJB 容器所支持的支持 JDK 的版本。

在配置 Enterprise Bean 的时候，你必须使用 Inprise 的应用服务器所提供的工具。这些工具能够编辑和修改第三方的代理商提供的 Inprise 配置描述器。还能够验证配置描述器，能够验证 bean 的源代码。

六、一个简单的 HELLO 例子

1、安装 Apusic Application Server

Note:以下以 Linux 为例，来说明 Apusic Application Server 的安装过程。其他平台的安装，可参考 Apusic Application Server 安装手册。

下载 JDK1.2，Apusic Application Server 必须运行在 JDK1.2 以上环境中。可从以下站点下载最新 JDK。

http://java.sun.com

下载 Apusic Application Server

Apusic Application Server 试用版可从以下网址得到：

http://www.apusic.com/download/enter.jsp

在下载完成后，你可以得到一个包裹文件 apusic.zip，选定安装目录，假设安装到/usr 下，则用以下命令：

```
cd /usr
jar xvf apusic.zip
```

/usr 下会出现一个目录 apusic，Apusic Application Server 的所有程序都被解压到/usr/apusic 下。

将以下路径加入到 CLASSPATH 中

```
/usr/apusic/lib/apusic.jar
$JAVA_HOME/lib/tools.jar
```

用以下命令运行 Apusic Application Server

```
java -Xms64m com.apusic.server.Main -root /usr/apusic
```

2）定义 EJB 远程接口(Remote Interface)

任何一个 EJB 都是通过 Remote Interface 被调用，EJB 开发者首先要在 Remote Interface 中定义这个 EJB 可以被外界调用的所有方法。执行 Remote Interface 的类由 EJB 生成工具生成。

以下是 HelloBean 的 Remote Inteface 程序：

```
package ejb.hello;

import java.rmi.RemoteException;
```

```
import java.rmi.Remote;
import javax.ejb.*;
public interface Hello extends EJBObject, Remote {
//this method just get "Hello World" from HelloBean.
public String getHello() throws RemoteException;
}
```

3）定义 Home Interface

EJB 容器通过 EJB 的 Home Interface 来创建 EJB 实例,和 Remote Interface 一样，执行 Home Interface 的类由 EJB 生成工具生成。以下是 HelloBean 的 Home Interface 程序：

```
package ejb.hello;
import javax.ejb.*;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.*;
/**
* This interface is extremely simple it declares only
* one create method.
*/
public interface HelloHome extends EJBHome {
```

```
public Hello create() throws CreateException,
RemoteException;
}
```

4）写 EJB 类

在 EJB 类中，编程者必须给出在 Remote Interface 中定义的远程方法的具体实现。EJB 类中还包括一些 EJB 规范中定义的必须实现的方法，这些方法都有比较统一的实现模版，编程者只需花费精力在具体业务方法的实现上。

以下是 HelloBean 的代码：

```
package ejb.hello;
import javax.ejb.*;
import java.util.*;
import java.rmi.*;
public class HelloBean implements SessionBean {
static final boolean verbose = true;
private transient SessionContext ctx;
// Implement the methods in the SessionBean
// interface
public void ejbActivate() {
if (verbose)
```

```
System.out.println("ejbActivate called");
}
public void ejbRemove() {
if (verbose)
System.out.println("ejbRemove called");
}
public void ejbPassivate() {
if (verbose)
System.out.println("ejbPassivate called");
}
/**
* Sets the session context.
* @param SessionContext
*/
public void setSessionContext(SessionContext ctx) {
if (verbose)
System.out.println("setSessionContext called");
this.ctx = ctx;
}
/**
```

```
* This method corresponds to the create method in
* the home interface HelloHome.java.
* The parameter sets of the two methods are
* identical. When the client calls
* HelloHome.create(), the container allocates an
* instance of the EJBean and calls ejbCreate().
*/
public void ejbCreate () {
if (verbose)
System.out.println("ejbCreate called");
}
/**
* **** HERE IS THE BUSINESS LOGIC *****
* the getHello just return a "Hello World" string.
*/
public String getHello()
throws RemoteException
{
return("Hello World");
}}
```

5）创建 ejb-jar.xml 文件

ejb-jar.xml 文件是 EJB 的部署描述文件，包含 EJB 的各种配置信息,如是有状态 Bean(Stateful Bean) 还是无状态 Bean(Stateless Bean)，交易类型等。ejb-jar.xml 文件的详细信息请参阅 EJB 规范。以下是 HelloBean 的配置文件：

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.2//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_2.dtd">
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>Hello</ejb-name>
<home>ejb.hello.HelloHome</home>
<remote>ejb.hello.Hello</remote>
<ejb-class>ejb.hello.HelloBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
```

```
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>Hello</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

6）编译和部署

编译 Java 源文件并将编译后 class 和 ejb-jar.xml 打包到Hello.jar

```
mkdir build
mkdir build/META-INF
cp ejb-jar.xml build/META-INF
javac -d build *.java
cd build
jar cvf Hello.jar META-INF ejb
cd ..
```

用 EJB 工具生成可部署到 Apusic Application Server 中运行的 jar

文件:

java com.apusic.ejb.utils.EJBGen -d /usr/apusic/classes/Hello.jar build/Hello.jar

增加/usr/apusic/classes/Hello.jar 到 CLASSPATH 中

将 Hello.jar 加入到 Apusic Application Server 配置文件中。在 /usr/apusic/config/server.xml 加入以下几行：

```
<module>
<ejb>
<ejb-uri>classes/Hello.jar</ejb-uri>
<bean>
<ejb-name>Hello</ejb-name>
<jndi-name>HelloHome</jndi-name>
</bean>
</ejb>
</module>
```

启动服务器

```
java -Xms64m com.apusic.server.Main -root /usr/apusic
```

7）写客户端调用程序

您可以从 Java Client,JSP,Servlet 或别的 EJB 调用 HelloBean。

调用 EJB 有以下几个步骤：

通过 JNDI(Java Naming Directory Interface)得到 EJB Home Interface

通过 EJB Home Interface 创建 EJB 对象，并得到其 Remote Interface

通过 Remote Interface 调用 EJB 方法

以下是一个从 Java Client 中调用 HelloBean 的例子：

```
package ejb.hello;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.ejb.*;
import java.rmi.RemoteException;
/**
* @author Copyright (c) 2000 by Apusic, Inc. All Rights Reserved.
*/
public class HelloClient{
public static void main(String args[]){
String url = "rmi://localhost:6888";
Context initCtx = null;
```

```
HelloHome hellohome = null;
try{
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.apusic.jndi.InitialContextFactory");
env.put(Context.PROVIDER_URL, url);
initCtx = new InitialContext(env);
}catch(Exception e){
System.out.println("Cannot get initial context: " +
e.getMessage());
System.exit(1);
}
try{
hellohome = (HelloHome)initCtx.lookup("HelloHome");
Hello hello = hellohome.create();
String s = hello.getHello();
System.out.println(s);
}catch(Exception e){
System.out.println(e.getMessage());
System.exit(1);
```

```
} }}
```

运行 HelloClient，可得到以下输出：

```
Hello World
```

6、JDBC 技术的数据库应用

JDBC 是执行 SQL 语句的 Java API。其实，JDBC 本身是一个产品的商标名。相对与 ODBC（Open Database Connectivity 开放数据库连接），也可以把 JDBC 看作“Java Database Connectivity(Java 数据库连接)”。它由一组用 Java 语言编写的类与接口组成。JDBC 已成为一种供工具/数据库开发者使用的标准 API,用户可以用纯 Java API 来编写数据库应用。

使用 JDBC 可以很容易地把 SQL 语句传送到任何关系型数据库中。换言之,用户不需要为每一个关系数据库单独写一个程序。用 JDBC API 写出唯一的程序,能够将 SQL 语句发送到相应的任何一种数据库。Java 与 JDBC 的结合,使程序员可以只写一次数据库应用软件后,就能在各种数据库系统上运行。由于 Java 语言具有健壮性、安全、易使用、易理解和自动下载到网络等优点,因此,它是数据库应用的一个极好的基础语言。现在需要找到一种能使 Java 应用与各种不同数据库对话的方式,而 JDBC 正是实现这种对话的一种机制。

JDBC 扩充了 Java 的应用范围。随着学习 JAVA 程序设计的人越来越多，对 Java 的数据库的访问需求会越来越迫切。MIS 管理员希望

Java 与 JDBC 结合，因为这有助于更容易、更经济地发布企业信息。而各个公司可以不断地使用已安装的数据库，方便地存取信息，不必顾及这些数据库是在何种 DBMS 系统下存储的。有助于缩短新应用的开发时间,并可大大简化数据库的安装与版本控制。另外，在商业销售信息服务领域，Java 与 JDBC 则可以更好地向外界客户提供最新信息。

6．1 JDBC 组成

JDBC 2.0 API 被划分为两部分：JDBC 2.0 核心 API 和 JDBC 2.0 标准扩展 API。这两个 API 在版本 3.0 中虽然已经结合起来了,但 JDBC 类和接口仍然在两个 Java 包中：java.sql 和 javax.sql。

1） java.sql 包，个包里面是 JDBC2.0 的核心 API。它包括了原来的 JDBC API（JDBC 1.0 版本），再加上一些新的 2.0 版本的 API。这个包在 Java 2 Platform SDK 里面有。

2） javax.sql 包，这里面是 JDBC2.0 的标准扩展 API。这个包是一个全新的，在 Java 2 Platform SDK，Enterprise Edition 里面单独提供。

JDBC2.0 的核心 API 包括了 JDBC1.0 的 API，并在此基础上增加了一些功能，对某些性能做了增强。使 java 语言在数据库计算的前端提供了统一的数据访问方法，效率也得到了提高。

JDBC 是向后兼容的，JDBC1.0 的程序可以不加修改的运行在

JDBC2.0 上。但是，假如程序中用到了 JDBC2.0 的新特性，就必须要运行在 JDBC2.0 版本上。

概括的来说，JDBC 核心 API 的新特性在两个方面做了工作。一个是支持一些新的功能，另一个就是支持 SQL3 的数据类型。

1） 在支持新功能方面：包括结果集可以向后滚动，批量的更新数据。另外，还提供了 UNICODE 字符集的字符流操作。

2） 在支持 SQL3 的数据类型方面：包括新的 SQL3 数据类型，增加了对持久性对象的存贮。

为了对数据的存取，操作更加方便，JDBC 的新特性是应用程序的设计更容易了。例如：数据块的操作能够显著的提高数据库访问的性能。新增加的 BLOB，CLOB,和数组接口能够是应用程序操作大块的数据类型，而不必客户端在存贮之前进行其它的处理。这样，就显著的提高了内存的使用效率。

6．2 JDBC 的实现和运行步骤

众所周知，JDBC（Java 数据库连接）是 Java 2 企业版的重要组成部分。它是基于 SQL 层的 API。通过把 SQL 语句嵌入 JDBC 接口的方法中，用户可以通过 Java 程序执行几乎所有的数据库操作。JDBC 只提供了接口，具体的类的实现要求数据库的设计者完成。通过生成这些接口的实例，即使对于不同的数据库，Java 程序也可以正确地执行 SQL 调用。所以对于程序员来说，不必把注意力放在如何向数据库发

送 SQL 指令，因为程序员需要了解和用到的只是 JDBC 的接口，只有在极少数情况下会用到面向特定数据库的类。

在 JDBC 程序中，首先需要做的是实现与数据库的连接。连接数据库通常需要实现以下几个步骤：

1）注册数据库驱动程序（driver）。可以通过调用 java.sql.DriverManager 类的 registerDriver 方法显式注册驱动程序，也可以通过 java.lang.ClassLoaderd 对象加载数据库驱动程序类隐式注册驱动程序。请看下面的代码段，它用于装载 Cloudscape 的数据库驱动程序，Cloudscape 是一个小型的数据库管理系统，用 Java 编写，可以用于嵌入式或客户-服务器模式：

DriverManager.registerDriver(new COM.cloudscape.core.JDBCDriver)；

或者

Class.forName(“COM.cloudscape.core.JDBCDriver“)；

实际上，在 JDBC 中，每个驱动程序提供者都要求由 java.sql.DriverManager 类注册一个驱动程序的实例。而这个注册是在使用 Class.fornName()调用时自动发生的。我们还可以采用另一种替换形式，使用 Java 属性机制指定一系列驱动程序。例如，下面的代码段将允许 java.sql.DriverManager 类在试图建立一个连接时装入列出的驱动程序：

System.setProperty(“jdbc.drivers“，“COM.cloudscape.core.JDBCDriver“)；

2） 建立连接 用 java.sql.DriverManager 类注册一个驱动程序之后,我们就可以调用 java.sql.DriverManager 类的 getConnection（）静态方法建立与数据库的连接。GetConnection（）方法返回一个 Connection 对象。需要注意的是，getConnection（）方法会自动从数据库驱动程序注册表中选择一个最合适的驱动程序。

3） 建立连接后，允许自动更新（AutoCommit）。调用 java.sql.Connection 接口的 serAutoCommit（）方法可以设定当程序向数据库发出一条 SQL 指令后，数据库是否立即更新。

下面通过一个简单的示例总结一下前面讨论的内容。这个示例将注册 Cloudscape 驱动程序并且建立一条连结：

```
//Import required packages
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.SQLException;
import java.sql.DriverPropertyInfo;
public class DriverTest {
public static void main(String arg[]) {
String protocol = “ jdbc:cloudscape:c:/CloudscapeDB“ ;
```

```
String driverClass = “COM.cloudscape.core.JDBCDriver “;
//Register the Cloudscape driver
try {
Class.forName(driverClass);
} catch (ClassNotFoundException cne) {
cne.printStackTrace();
}
//Get a connection from the DriverManager
try {
Connection connection = DriverManager.getConnection(protocol);
//Get the metadata
DatabaseMetaData metadata = connection.getMetaData();
System.out.println(“Productname:“
                    + metadata.getDatabaseProductName());
System.out.println(“Driver name: “
                                + metaData.getDriverName());
} catch (SQLException sqle) {
sqle.printStackTrace();
}}}
```

6.3 数据源（Data Source）及JNDI

数据源是在JDBC 2.0中引入的一个概念。在JDBC 2.0扩展包中

定义了 javax.sql.DataSource 接口来描述这个概念。如果用户希望建立一个数据库连接，通过查询在 JNDI 服务中的数据源，可以从数据源中获取相应的数据库连接。这样用户就只需要提供一个逻辑名称（Logic Name），而不是数据库登录的具体细节。

JNDI 的全称是 Java Naming and Directory Interface， 可以理解为 Java 名称和目录服务接口。JNDI 向应用程序提供了一个查询和使用远程服务的机制。这些服务可以是任何企业服务。对于 JDBC 应用程序来说，JNDI 提供的是数据库连接服务。当然 JNDI 也可以向数据库提供其他服务。

其实 JNDI 并不难理解。JNDI 使应用程序通过使用逻辑名称获取对象和对象提供的服务，从而使程序员可以避免使用与提供对象的机构有关联的代码。在数据源中存储了所有建立数据库连接的信息。就象通过指定文件名你可以在文件系统中找到文件一样，通过提供正确的数据源名称，你可以找到相应的数据库连接。javax.sql.DataSource 接口定义了如何实现数据源。

实际应用中，你可以把 DataSource 注册到 JNDI，一个数据源可以被看作是由 JNDI 服务检索出的一个网络资源。

在本节首先给出了一个实际程序，然后通过程序来讲解如何通过 JNDI 查询数据源。

```
import java.sql.*;
```

```
//Import DataSource driver API
import oracle.jdbc.driver.*;
import oracle.jdbc.pool.DataSource;
//Import javax.sql classes
import javax.sql.*;
//Import JNDI classes
import javax.naming.*;
import javax.naming.spi.*;
import java.util.Hashtable;
public class DataSourceJNDITest{
public static void main (String args [])throws SQLException{
// 初始化名称服务环境
Context ctx = null;
try{
Hashtable env = new Hashtable (5);
env.put (Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL, "file:JNDI");
ctx = new InitialContext(env);
}catch (NamingException ne){
```

```
ne.printStackTrace();
}
bind(ctx, "jdbc/chidb");
lookup(ctx, "jdbc/chidb");
}
static void bind (Context ctx, String ln)throws NamingException,
SQLException{
// 创建一个 DataSource 实例
DataSource ods = new DataSource();
ods.setDriverType("thin");
ods.setServerName("Chicago");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("chidb");
ods.setPortNumber(1521);
ods.setUser("guest");
ods.setPassword("guest");
// 把 DataSource 实例注册到 JNDI 中
System.out.println ("Doing a bind with the logical name : " + ln);
ctx.bind (ln,ods);
System.out.println ("Successfully bound");
```

```
}
static void lookup (Context ctx, String ln)throws NamingException，
SQLException{
// 从 JNDI 中查询 DataSource 实例
System.out.println ("Doing a lookup with the logical name : " + ln);
DataSource ods = (DataSource) ctx.lookup (ln);
System.out.println ("Successful lookup");
// 从查询到的 DataSource 实例中获取数据库连接
Connection conn = ods.getConnection();
// 进行数据库操作
getUserName(conn);
// 关闭连接
conn.close();
conn = null;
}
static void getUserName(Connection conn) throws SQLException{
// 生成一个 Statement 实例
Statement stmt = conn.createStatement ();
// 从 addressbook 表中选中姓名列
ResultSet rset = stmt.executeQuery ("select NAME from addressbook");
```

```
// 列出 addressbook 表所有人的姓名
while (rset.next ())
System.out.println ("Name is " + rset.getString (1));
// 关闭 RseultSet 实例
rset.close();
rset = null;
// 关闭 Statement 实例
stmt.close();
stmt = null;
}}
```

程序首先生成了一个 Context 实例。javax.naming.Context 接口定义了名称服务环境（Naming Context）及该环境支持的操作。名称服务环境实际上是由名称和对象间的相互映射组成。程序中初始化名称服务环境的环境工厂（Context Factory）是 com.sun.jndi.fscontext.RefFSContextFactory（该类在 fscontext.jar 中可以找到，由于 fscontext.jar 中包含的不是标准的 API，用户需要从 www.javasoft.com 中的 JNDI 专区下载一个名为 fscontext1_2beta3.zip 的压缩文件，在该文件中可以找到 fscontext.jar）。环境工厂的作用是生成名称服务环境的实例，javax.naming.spi.InitialContextFactory 接口定义了环境工厂应

该如何初始化名称服务环境。在初始化名称服务环境时还需要定义环境的 URL。程序中使用的是"file:JNDI"，也就是把环境保存在本地硬盘的 JNDI 目录下。

初始化了名称服务环境后，就可以把数据源实例注册到名称服务环境中。注册时调用 javax.naming.Context.bind()方法，参数为注册名称和注册对象。注册成功后，在 JNDI 目录下会生成一个.binding 文件，该文件记录了当前名称服务环境拥有的名称及对象。

当需要在名称服务环境中查询一个对象时，需要调用 javax.naming.Context.lookup（）方法，并把查询到的对象显式转化为数据源对象。然后通过该数据源对象进行数据库操作。

在这个例子中，程序和名称服务环境都是在同一台计算机上运行。在实际的应用中，程序可以通过 RMI 或 CORBA 向名称服务环境注册或查询对象。例如在一个服务器-客户机结构中，客户机上的应用程序只需要知道数据源对象在服务器名称服务环境中的逻辑名称，就可以通过 RMI 向服务器查询数据源，然后通过建立与数据库的连接。

7、总结

总之 J2EE 涉及许多应用技术，限于本文篇幅未对 J2EE 与 XML 应用结合、Java 安全编程、J2EE 核心模式做详细介绍。感兴趣的话，大家可以参考文后推荐的书籍。要全面了解 J2EE 成为一个“Legacy Integrator”具体学习过程可以参考文后附录一的一张英文表格，这可

是 Sun 在美国的最新的 J2EE 标准培训课程表。参照它大家可以选择某一个角色进行相关课程内容的学习。在附录二中给了一个完整的 Windows2000 下安装 J2EE 和部署 J2EE 应用程序的例程有基础的同学可以尝试以下。

参考文献及推荐书籍：

1． Subrahmanyam Allamaraju、Cedric Buest、John Davies；《J2EE 编程指南》；电子工业出版社，2002；
2．Richard Modson-Haefel；《Enterprise JavaBeans》；中国电力出版社，2001；
3．Jess Garms、Daniel Somerfield；《Java 安全性编程指南》；电子工业出版社，2002；
4．Deepak Alur、John Crupi、Dan Malks；《J2EE 核心模式》；机械工业出版社，2002；
5．Steven Gould；《Develop n-tier application using J2EE》；程序员大本营 2001 Java 版
6．《The Business Benefits of EJB and J2EE Technologies over COM+ and Windows DNA》；程序员大本营 2001 Java 版
7．Monica Pawlan；《The J2EE Tutorial》chapter overview ；程序员大本营 2001 Java 版
8．伊晓强；《J2EE 全实例教程》 北京希望电子出版社

本文所用部分图片由《The J2EE Tutorial》中的英文图片修改而成.

附录一

见网页 Course Catalog - Java Technology

附录二
Windows2000 下安装 J2EE 和部署 J2EE 应用程序

1．安装

可以从以下网址下载一个 J2EE SDK（j2sdkee-1_3-beta2-win.exe）：http://java.sun.com/Download5?referer=&download-name=j2sdkee-1_3-beta2-win.exe&config-file=j2sdkee-1_3-beta2.config&platform=win&domain-checked=unknownhostname&DEBUG=&document=license&button=ACCEPT 。如果这个地址不行，很可能是 Sun 的页面更新了的缘故，不要紧，可从这里下载：http://java.sun.com/j2ee/j2sdkee-beta/index.html 。建议从后面那个地址下载，这样可有更多的选择余地，特别是 Sun 将 J2EE SDK 升级了的话。也许你已装了旧版的 J2EE SDK 产品，如果是，在安装新下载的 J2EE SDK 之前请先卸载或删掉旧版的 J2EE SDK。运行 j2sdkee-1_3-beta2-win.exe，按安装步骤安装好 J2EE SDK。这里假设你的 J2EE SDK 安装在：C:\j2sdkee1.3 目录下。

2．设置环境变量

在运行 J2EE SDK 之前，你必须设置以下环境变量：

J2EE_HOME - 你的 J2EE SDK 所安装的目录。如本例中的：C:\j2sdkee1.3 。

JAVA_HOME - 你的 Java 2 SDK 所安装的目录。

PATH - 设置为你安装 J2EE SDK 目录下的 bin 目录。如本例的的：C:\j2sdkee1.3\bin 。

ClassPath - 增添%J2EE_HOME%\lib\j2ee.jar 到 ClassPath 中。本例中也可写为：C:\j2sdkee1.3\lib\j2ee.jar

3．运行 J2EE

Dos 命令行敲入以下命令：

```
%J2EE_HOME%\bin\j2ee -verbose
```

显示以下信息表示运行成功：(不同的版本显示可能不同)

```
J2EE server listen port: 1050
Naming service started:1050
Binding DataSource, name = jdbc/EstoreDB, url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/DB2, url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/Cloudscape, url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/InventoryDB, url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/DB1, url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/XACloudscape, url = jdbc/XACloudscape__xa
Binding DataSource, name = jdbc/XACloudscape__xa, dataSource =
```

```
COM.cloudscape.core.RemoteXaDataSource@330913
   Starting JMS service ... Initialization complete - waiting for client
requests
    Binding : < JMS Destination : jms/Queue , javax.jms.Queue >
    Binding : < JMS Destination : jms/Topic , javax.jms.Topic >
    Binding : < JMS Cnx Factory : jms/TopicConnectionFactory , Topic ,
No properties >
    Binding : < JMS Cnx Factory : TopicConnectionFactory , Topic , No
properties >
    Binding : < JMS Cnx Factory : jms/QueueConnectionFactory , Queue ,
No properties >
    Binding : < JMS Cnx Factory : QueueConnectionFactory , Queue , No
properties >
    Starting web service at port:8000
    Starting secure web service at port:7000
    Apache Tomcat/4.0-b4-dev
    Starting web service at port:9191
    Apache Tomcat/4.0-b4-dev
    J2EE server startup complete.
```

启动成功后，在 IE 浏览器中访问 http://localhost:8000/ 可以看到默认的主页信息 。

4. 编写和运行 HelloWorld 程序

J2EE 应用程序一般使用 RMI（远程方法调用）来完成客户端与服务器的交互。当然，其间也少不了 EJB 的作用。本例为一个 J2EE 应用程序：客户端向服务器发送一个问候语：“ Hello,Remote Object ”。服务器收到该问候语后打印该问候语，并返回一字符串作为应答。客户端收到此应答后打印它。

RemoteInterface.java

```
/**
 * 第一步：
 * 定义一个新的接口继承 javax.ejb.EJBObject。新定义的接口中的每一个方
法都必须抛出
 * java.rmi.RemoteException 异常。
 */
public interface RemoteInterface extends javax.ejb.EJBObject
{
  public String message(String str)throws java.rmi.RemoteException;
}

RemoteObject.java
/**
 * 第二步：
```

```
 * 定义一个类来实现 javax.ejb.SessionBean 接口。并在该类中实现在第一步
中编写的接口中所定义的方法。
 */
public class RemoteObject implements javax.ejb.SessionBean
{
  public String message(String str)throws java.rmi.RemoteException
  {
    System.out.println("Remote   Object   Received   From   Client:
\""+str+"\""); //打印（从客户端）接收到的字符串。
    return   "Hello,I'm   Remote   Object,I   received   your   message:
\'"+str+"\'"; //返回一应答字符串。
  }

  public RemoteObject() {}
  public void ejbCreate() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(javax.ejb.SessionContext sc) {}
}

RemoteHome.java
/**
 * 第三步：
 * 定义一个类继承 javax.ejb.EJBHome 。
 */
public interface RemoteHome extends javax.ejb.EJBHome
{
  RemoteInterface                                          create()throws
java.rmi.RemoteException,javax.ejb.CreateException;
}

Client.java

/**
 * 第四步：
 * 定义客户端类。
 */
public class Client
{
  public static void main(String[] args)
```

```
  {
    try
    {
      javax.naming.Context initContext=new
javax.naming.InitialContext();
      Object obj=initContext.lookup("HelloWorld"); //远程查找，由名字
得到对应的对象。
      RemoteHome
home=(RemoteHome)javax.rmi.PortableRemoteObject.narrow(obj,RemoteHome
.class);
      RemoteInterface remote=home.create();
      String receiveFromRemote=remote.message("Hello,Remote Object!");
//远程方法调用
      System.out.println("Client Received From Remote Object:
\""+receiveFromRemote+"\"");
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

假设以上四个 Java 文件存于 C:\HelloWorld\ 下，编译它们如：C:\HelloWorld>javac *.java 。

5. 部署应用程序

启动 Application Dopolyment Tool：新开一个 Dos 窗口，键入以下命令，%J2EE_HOME%\bin\deploytool 。该工具启动速度可能比较慢，要耐心等待。启动成功后会出现主界面（此时不要关闭 Dos 窗口）。在该界面中选 择 File 菜单 ，再选 New Application 项。在 Application File Name 输 入 ：C:\HelloWorld\HelloWorld.ear 。在 Application Disply Name 输 入 你所喜欢的显示名如：HelloWorld。点 击 OK，在主界面的树形结构 Files-->Applications 下将增加新的一项：HelloWorld。这意味着产生了一个新的应用程序。接下来我们要做的就是部署该应用程序。在主界面的树形结构下选中 HelloWorld，然后再在主界面的 File 菜单中选取 New-->Enterprise Bean，在弹出的名为“New Enterprise Bean - Introduction”窗口中选取 Next 跳过第一步，在接下来的一步中，Create New EJB File in Application 项中选 HelloWorld，在 EJB Display Name 中填上你喜欢的名字如：Hello World EJB，点击 Edit 按钮，在弹出的窗口中，Start Directory 中填：C:\HelloWorld\，在 Available Files 中展开树形结构 C:\HelloWorld\ ，选取 RemoteInterface.class 、 RemoteObject.class 、 RemoteHome.class 、Client.class 四项，点 Add 按钮添加，然后按 OK 确定。此时在 Contents 框中

增加了该四个 class。点 Next 进入下一步。Session 项选 Stateless，意为不保存 session 状态。Enterprise Bean Class 选 RemoteObject。Enterprise Bean Name 中填上你喜欢的名字如：Hello World Bean。Remote Home Interface 中选 RemoteHome，Remote Interface 中选 RemoteInterface。选 Next 进入下一步。接下来的步骤可直接点 Finish。这时主界面的树形结构中 Files-->Application-->Hello World 中将出现 Hello World EJB-->Hello World Bean 子项。在主界面的树形结构下选中 Hello World，然后再在主界面的 Tools 菜单中选取 Deploy,将弹出新的窗口名为‘ Deploy Hello World- Introduction ”。Object to deploy 中选 Hello World,Target server 中选 localhost,选中 Retuen Client Jar，在 Client Jar File Name 中填上：C:\HelloWorld\HelloWorldClient.jar。选 Next 进入下一步，在 Application 框的 JNDI Name 框中双击并填上 HelloWorld,注意必须与 Client.java 中 Object obj=initContext.lookup("HelloWorld")的“ HelloWorld ”保持一致。点 Next 进入下一步。点 Finish 完成。这时将出现 Deployment Progress 窗口。如果有误，该窗口将出现异常信息。如果一切正常，点 OK 便完成了部署工作。

6．运行应用程序

新开一个 Dos 窗口。进入 C:\HelloWorld\Classes 目录下运行：C:\HelloWorld\Classes>java -classpath %J2EE_HOME%\lib\j2ee.jar;.;HelloWorldClient.jar; Client 。运行成功则出现如下信息：Client Received From Remote Object: "Hello,I'm Remote Object,I received your message: 'Hello,Remote Object!'" 。而服务端 Dos 窗口（j2ee -verbose）中出现如下信息：Remote Object Received From Client: "Hello,Remote Object!" 。

本程序很适 J2EE 的入门，简单而明了。本程序由 Charly 设计编写，并由其本人多次运行确保无误。如不能正常运行，请检查你的环境变量设置是否正确，最好不要直接将以上各命令 Copy 到 Dos 窗口中运行，某些字符可能受中文全角字符的影响而不被识别。另外如果某个应用程序部署不成功，可以使用 Application Deployment Tool 主界面的 Edit 菜单的 Delete 项将这个应用程序（如：Hello World）或其子项删掉。此工具的右键功能几乎没有，愤憾。

Sun 为 rmi、Servlet(JSP)、JavaBean 等 Java 技术摇旗呐喊已有多年。J2EE 的出现不可谓不是一个大的综合和封装。它的出现应该说是这些技术走向成熟的结果。举个例子，JSP 技术去年（2000 年）在大中华范围的书店中都少有介绍，几近没有。现在业已铺天盖地。ASP 似乎已成明日黄花，不过 Microsoft C#的推出是否会让它有所改变？有待观察。面对竞争对手的压力，Sun 自然对 J2EE 寄予厚望。应该说，Sun 的 Java 设计一直都很规格和严谨。在这方面，当今还没有其它哪门语言能与之匹敌。J2EE 的设计自然也是大手笔。J2EE 被设计成多层次的分布式的结构。客户端可包括浏览器，桌面应用程序，甚至其它设备如 PDA 程序。服务端可以为 Web 服务和应用(Application)服务,这分别是 Servlet(JSP)和 rmi 技术的体现。客户端通过网络协议向服务端提出请求（Web 服务或应用服务），该请求如果需要后台企业信息系统（如数据库）支持，服务端则直接或在 EJB 支持下来操作后台企业信息系统（如数据库），并将结果返回至客户端；如

果不需要后台企业信息系统（数据库）支持，服务端则直接或在 EJB 支持下处理该请求，并将结果返回至客户端。当然其间各环节都少不了安全控制：J2EE 的标准的公开存取控制规则。应该说，J2EE 的 Web 应用会取得很大成功，因为 Servlet(JSP)在 Web 方面有数不尽的优势：速度快（比 ASP、PHP 可能快好几倍）、跨平台、稳定（这不能不说是得益于 Java 的规格和严谨）、成熟（实际应用已久且应用广泛）、功能强大。而 J2EE 的应用服务应该也算不错，只是其使用的标准界面 Swing 实在是太耗资源且效率低。君不见，当今用 Swing 写的程序哪个不存在以下情形：程序启动时象国产拖拉机，轰隆轰隆地响--那是读硬盘的声音（有些夸张，但不过分）；界面刷新慢，明显的延时；操作不方便，快捷键和右键功能少（当然这与软件生产者有关，但 Swing 要负一部分责任）。此如说 Sun 的 Forte4Java、J2EE SDK、Oracle 的 Jdeveloper、IBM 的 Websphere 都有此症。作为程序员，我们也许还能接受，以不断提升硬件来迷补其不足（虽然至今还是不足）。但作为客户企业，你用 Swing 给人家做了一个 ERP 或其中的某一环节，用起来有如此顽症，怎么向人家解释？向他（她）说，这是用 Swing 写的，所以很慢？对不起，人家不会关心你用什么写的，人家给你的也许只有抱怨。除了界面部分外，J2EE 的应用服务应该说是很不错的，如分布式计算。