第 3 章　J2EE 概述

为了快速设计和开发企业级的应用程序，Sun 公司推出了一种全新概念的模型——Java 2 Platform, Enterprise Edition（J2EE），它与传统的互联网应用程序模型相比有着不可比拟的优势。

J2EE 平台提供了一个多层结构的分布式应用程序模型，该模型具有重用组件的能力、基于扩展标记语言（XML）的数据交换、统一的安全模式和灵活的事务控制；使开发者不仅可以比以前更快地发表对市场的新的解决方案，而且其独立于平台、基于组件的 J2EE 解决方案不再受任何提供商的产品和应用程序编程界面的限制。提供商和买主都可以选择最合适于其商业应用和所需技术的产品和组件。

本章将主要从如下几个方面介绍 J2EE：

- J2EE 框架：分布式多层应用程序模型
- J2EE 核心技术
- J2EE 设计模式

3.1　J2EE 框架

3.1.1　分布式多层应用程序模型

当今，许多企业都需要扩展他们的业务范围，降低自身经营成本，缩短他们和客户之间的响应时间，这就需要存在一种简捷、快速的服务于企业、合作伙伴和雇员之间。典型的说，提供这些服务的应用软件必须同企业信息系统（EIS）相结合，并提供新的能向更为广阔的用户提供的服务。这些服务要具备以下的特点：

- 高可用性：以满足现在的全球商业环境
- 安全性：保护用户的隐私和企业数据的安全
- 可依赖性和可扩展性：保证商业交易的正确和迅捷

最初这些服务是由两层的应用（也称为客户/服务器或 C/S 结构）来实现的。图 1-1 表示的就是一个典型的两层体系。在这种结构下，服务器往往只提供唯一的服务，即数据库服务；客户端负责数据访问、应用业务逻辑、将结果转换为一个格式以便显示，为用户显示内部的接口，以及接受用户的输入。客户/服务器的体系在开始的时候很容易配置，但难于升级或者扩展，而且通常基于私有的协议——典型的是私有的数据库协议。商业和表现逻辑的重新使用也很困难。在 Web 领域中，可能最重要的就是扩展，而两层的应用不便于升级扩展，因此并不适合用在广域网中。

为了解决两层体系的不足，中间层出现在客户端和后端数据源之间，这些中间层提供

了把商业功能和数据与 EIS 相结合的功能；它们把客户端从复杂的业务逻辑中分离出来，利用成熟的互联网技术使用户在管理上所花费的时间最小化。而 J2EE 正式降低了开发这种中间层服务的成本和复杂程度，因而使得服务可以被快速的展开，并能够更轻松的面对竞争中的压力。

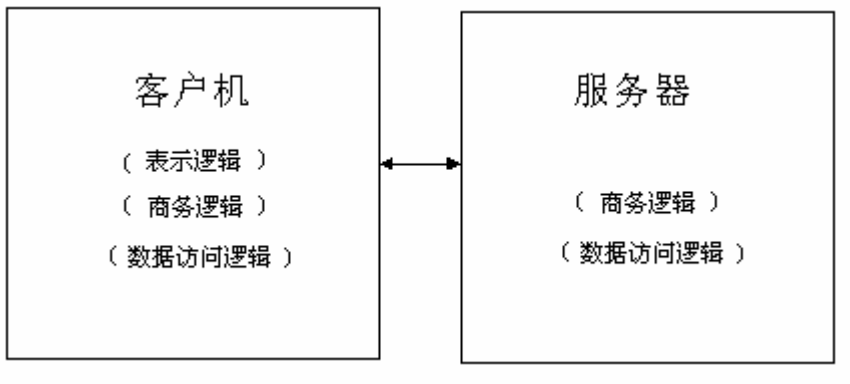

图 1-1　Client/Server 双层应用体系

J2EE 平台使用了一个多层的分布式应用程序模型。应用程序的逻辑根据其实现的不同功能被封装到组件中，组成 J2EE 应用程序的大量应用程序组件根据其所属的层被安装到不同的机器中。图 1-2 描述了一个分布式 J2EE 应用程序，它可以分为如下四层：

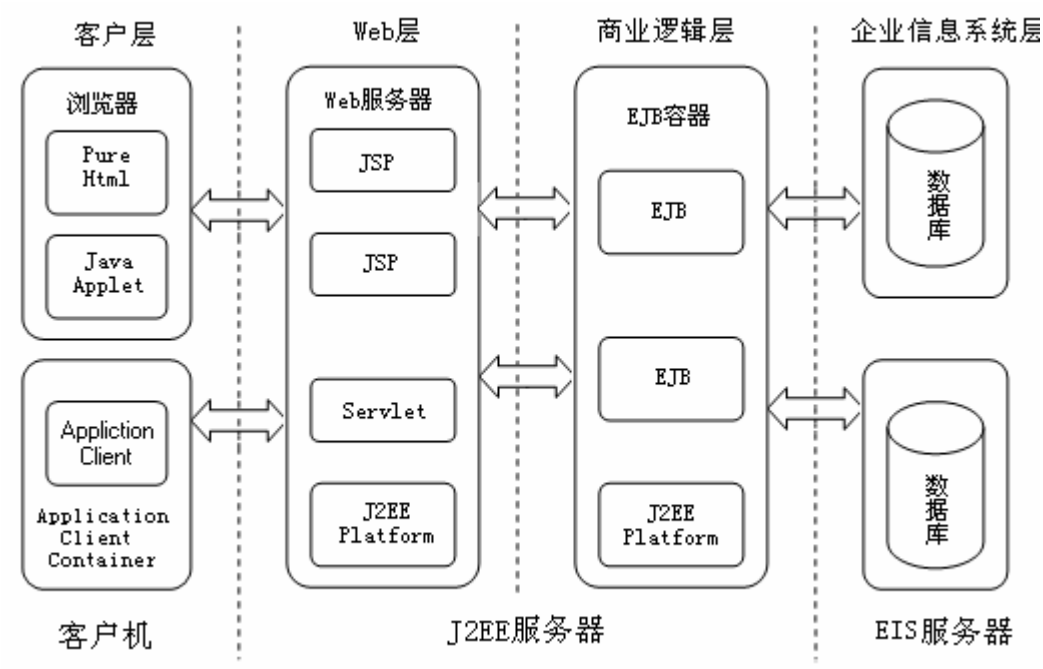

图 1-2　多层结构的应用程序

- 运行在客户端机器的客户层组件

- 运行在 J2EE 服务器中的 Web 层组件
- 运行在 J2EE 服务器中的业务逻辑层组件
- 运行在 EIS 服务器中的企业信息系统（EIS）层软件

注意：事实上 J2EE 应用程序既可以是三层结构（去除图 1-2 中的 Web 层，由客户端直接和运行在 J2EE 服务器中的业务逻辑层通信），也可以是四层甚至更多层结构。但很多时候总是将 J2EE 应用程序的多层结构考虑为三层结构。这是因为它们分布在三个不同的位置：客户端机器、J2EE 服务器机器和在后端的传统的机器。三层结构的应用程序可以理解为在标准的两层结构的客户端/服务器模式的客户端应用程序和后端存储资源中间增加了一个多线程的应用程序服务器。

下面各节将分别介绍 J2EE 应用程序的各层。

3.1.2　客户端

一个 J2EE 客户端既可以是一个 Web 客户端，也可以是一个应用程序客户端。

1. Web 客户端

一个 Web 客户端由两部分组成：

- 由运行在 Web 层的 Web 组件生成的包含各种标记语言（HTML、XML 等等）的动态 Web 页面。
- 接受从服务器传送来的页面并将它显示出来的 Web 页面。

Web 客户端有时被称之为瘦客户端。瘦客户端一般不做数据库查询、执行复杂的商业规则或连接传统应用程序之类的操作，通常这样的重量级操作交给在 J2EE 服务器执行的 enterprise bean。这样就可以充分发挥 J2EE 服务器端技术在安全性、速度、耐用性和可靠性方面的优势。

2. Applets

从 Web 层接收的一个 Web 页面可能包含内嵌的 applet， applet 是用 Java 语言编写的客户端小应用程序，它运行于 Java 虚拟机中，后者通常安装在 Web 浏览器中。然而为了在 Web 浏览器中成功地运行 applet，客户端系统很可能需要 Java 插作和安全策略文件。

Web 组件是用来建立一个 Web 客户端程序的首选 API，因为这样在客户端系统中就不需要插件和安全策略文件。同样的，使用 Web 组件可以有效地改善应用程序设计，因为它们提供了一个将应用程序设计和 Web 页面设计有效分离的途径，Web 页面的设计者可以不必关心 Java 编程语言的语法就能很好地完成自己的工作。

3. 应用程序客户端

一个 J2EE 应用程序客户端运行在客户端机器上，它使得用户可以处理需要比标记语言所能提供的更丰富的用户界面的任务。具有代表性的是用 Swing 或抽象窗口工具包（AWT）API 建立的图形用户界面，当然一个命令行界面也是可能的。

应用程序客户端可以直接访问运行在商业层的 enterprise bean，但如果应用程序需要授

权，J2EE 应用程序客户端也可以打开一个 HTTP 连接来与一个运行在 Web 层的 servlet 建立通信。

3.1.3　J2EE 服务器

客户端可以直接和运行在 J2EE 服务器中的业务逻辑层进行通信；如果是一个运行在浏览器中的客户端，也可以通过运行在 Web 层中的 JSP 页面或 Servlet 进行通信。

1.　Web 组件

J2EE 的 Web 组件既可以是 servlet 也可以是 JSP 页面。Servlet 是一个 Java 编程语言类，它可以动态地处理请求并作出响应；JSP 页面是一个基于文本的文档，它以 servlet 的方式执行，但是它可以更方便的建立静态内容。

在装配应用程序时，静态的 HTML 页面和 applet 被绑定到 Web 组件中，但它们并不被 J2EE 规范视为 Web 组件；与之相似的还有服务器端的功能类，这些类也可以被绑定到 Web 组件中，但同样不被 J2EE 规范视为 Web 组件。

和客户层一样，Web 层也可以包含一个 JavaBeans 组件以管理用户的输入并将输入发送到运行在商业层的 enterprise bean 进行处理。

2.　业务逻辑组件

业务逻辑代码表示了与特定商业领域（例如银行、零售等行业）相适应的逻辑。它由运行在业务逻辑层的 enterprise bean 处理。一个 enterprise bean 可以从客户端接受数据，对它进行处理，并将其发送到企业信息系统层以作存储；同时它也可以从存储器获取数据，处理后将其发送到客户端应用程序。

有三种类型的 enterprise beans：session beans、entity beans 和 message-driven beans。Session bean 描述了与客户端的一个短暂的会话。当客户端的执行完成后，session bean 和它的数据都将消失；与之相对应的是一个 entity bean 描述了存储在数据库表中的一行持久稳固的数据，如果客户端终止或者服务结束，底层的服务会负责 entity bean 数据的存储。Message-driven bean 结合了 session bean 和 Java 信息服务（JMS）信息监听者的功能，它允许一个商业组件异步地接受 JMS 消息。

3.　J2EE 容器

容器是一个组件和支持组件的底层平台特定功能之间的接口，在一个 Web 组件、enterprise bean 或者是一个应用程序客户端组件可以被执行前，它们必须被装配到一个 J2EE 应用程序中，并且部署到它们的容器。

装配的过程包括为 J2EE 应用程序中的每一个组件以及 J2EE 应用程序本身指定容器的设置。容器设置定制了由 J2EE 服务器提供的底层支持，这将包括安全性、事务管理、Java 命名目录接口（JNDI）搜寻以及远程连接等。下面是其中的主要部分：

- J2EE 的安全性模式可以让开发者对一个 Web 组件或 enterprise bean 进行配置以使得只有授权用户可以访问系统资源。

- J2EE 的事务模式可以让开发者指定方法之间的关系以组成一个单个的事务，这样在一个事务中的所有方法将被视为一个单一的整体。
- JNDI 搜寻服务为企业中的多种命名目录服务提供一个统一的接口，这使得应用程序组件可以访问命名目录服务。
- J2EE 远程连接模式管理客户端和 enterprise bean 之间的底层通信。在一个 enterprise bean 被建立后，客户端在调用其中的方法时就象这个 enterprise bean 直接运行在同一个虚拟机上一样。

实际上，J2EE 体系结构提供的可配置的服务意味着在相同的 J2EE 应用程序中的应用程序组件根据其被部署在什么在地方而在实际运行时会有所不同。例如，一个 enterprise bean 可能在一个产品环境中拥有包含访问数据库数据的某种级别的安全性设置，而在另一个产品环境中是另一个访问数据库的级别。

容器还管理诸如 enterprise bean 和 servlet 的生存周期、数据库连接资源池以及访问在 J2EE APIs 中介绍的 J2EE 平台 API 这样不能配置的服务。尽管数据持久化是一个不能配置的服务，但是 J2EE 体系统结构允许你在你想要获得比默认的容器管理持久化所能提供更多的控制时，通过在你的 enterprise bean 执行中包含适当的代码以重载容器管理持久化。例如，你可以使用 bean 管理持久化以实现你自己的 finder（查找）方法或者是建立一个定制的数据库缓冲区。

如图 1-2 所示，J2EE 应用程序部署时会将组件安装到 J2EE 容器中，J2EE 容器有如下几种：

- Enterprise JavaBeans（EJB）容器：EJB 容器是 J2EE 服务器的一部分。J2EE 服务器是 J2EE 产品的运行部分，一个 J2EE 服务器提供 EJB 容器和 Web 容器。EJB 容器负责管理 J2EE 应用程序中 enterprise bean 的执行，Enterprise bean 和它的容器运行在 J2EE 服务器中。
- Web 容器：Web 容器管理 J2EE 应用程序的 JSP 页面和 servlet 组件的执行，Web 组件和它的容器也运行在 J2EE 服务器中。
- 客户端应用程序容器：客户端应用程序容器管理应用程序客户端组件的运行，应用程序客户端和它的容器运行在客户端中。
- Applet 容器：Applet 容器管理 applet 的执行，由运行在客户端的一个 Web 浏览器和 Java 插件一同组成。

3.1.4　企业信息系统层

企业信息系统层处理企业信息系统软件并包含诸如企业资源计划（ERP）、主机事务处理、数据库系统和其它传统系统这样的底层系统。J2EE 应用程序组件可能需要访问企业信息系统，例如当它希望获得一个数据库连接时。

3.2　J2EE 核心技术

J2EE 提供了一个框架，实现这个框架的引擎工具留给第三方厂商完成。部分厂商只是专注于 J2EE 架构中的的特定组件，例如 Apache 的 Tomcat 提供了对 JSP 和 servlets 的支持，BEA 系统公司则通过其 WebLogic 应用服务器产品为整个 J2EE 规范提供了一个较为完整的实现。

在以下的部分中将讨论构成 J2EE 的每一种技术，并且看看类似 WebLogic Server 的产品如何在一个分布式的应用中支持它们。最常用的 J2EE 技术是：JDBC、JNDI、EJB、JSP 和 servlet，因此本节也将着重讨论这些方面。

图 1-3 描述了在一个分布式应用中，每项 J2EE 技术最常用在哪里。

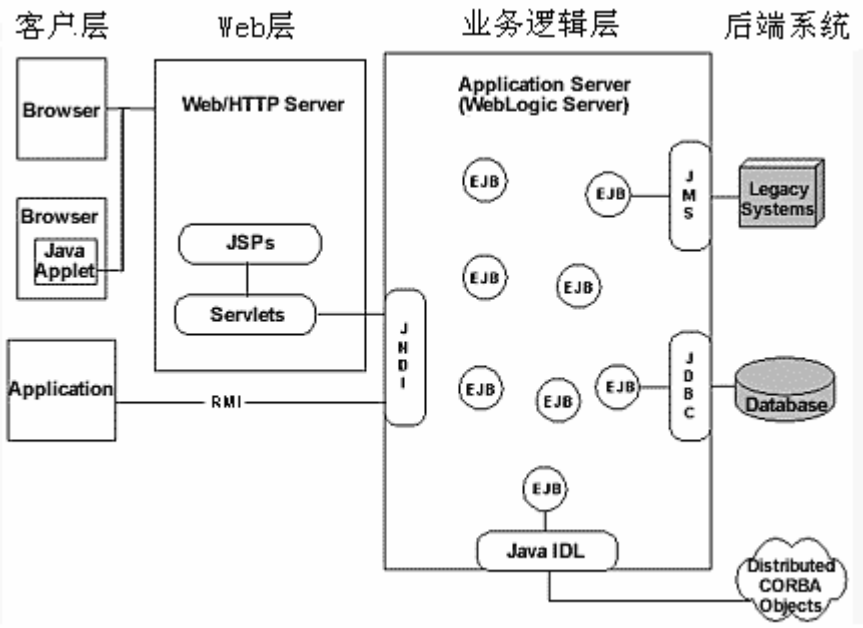

图 1-3　分布式应用中的 J2EE 技术

3.2.1　Java 数据库连接（JDBC）

JDBC API 以一个统一的方式访问各种数据库，其接口包含在 java.sql 和 javax.sql 这两个包中。与 ODBC 类似，JDBC 将开发者和私有数据库之间的问题隔离开来。由于建立在 Java 上，JDBC 还可以提供平台无关的数据库访问。

JDBC 定义了 4 种不同的驱动，具体来说，包括有：

- JDBC-ODBC 桥

在 JDBC 刚产生时，JDBC-ODBC 桥是非常有用的。通过它，开发者可以使用 JDBC

来访问一个 ODBC 数据源。缺点是，它需要在客户机器上安装有一个 ODBC 驱动，该机器通常运行的是微软 Windows 系统。使用这一类的驱动器，你就会失去 JDBC 平台无关的好处。此外，ODBC 驱动器需要客户端的管理。

- JDBC-native 驱动桥

JDBC-native 驱动桥提供了一个构建在本地数据库驱动上的 JDBC 接口而没有使用 ODBC。JDBC 驱动将标准的 JDBC 调用转化为对数据库 API 的本地调用，使用这一类的驱动也会失去 JDBC 平台无关性的好处，并且需要安装客户端的本地代码。

- JDBC-network 桥

JDBC-network 桥不需要客户端的数据库驱动。它使用网络上的中间服务器来访问一个数据库，这种应用使得以下技术的实现有了可能，这些技术包括负载均衡、连接缓冲池和数据缓存等。由于第 3 种类型往往只需要相对更少的下载时间，具有平台独立性，而且不需要在客户端安装并取得控制权，所以很适合于 Internet 上的应用。

- 纯 Java 驱动

这一类驱动使用纯 Java 数据库驱动来提供直接的数据库访问，由于这类驱动运行在客户端，并且直接访问数据库，因此运行在这个模式暗示要使用一个两层的体系。要在一个 n 层的体系中使用该类型的驱动，可以通过一个包含有数据访问代码的 EJB，并且让该 EJB 为它的客户提供一个数据库无关的服务。

在企业级应用的另一个常见的数据库特性是事务处理。事务是一组申明（statement），它们必须做为同一个 statement 来处理以保证数据完整性。缺省情况下 JDBC 使用自动提交（auto-commit）事务模式，这可以通过使用 Connection 类的 setAutoCommit() 方法来实现，一个自动提交意味着在每一个数据库读写操作之后任何其它应用程序显示数据时都会看到更新了的数据。然而，如果你的应用程序执行两部分相互依赖的数据库访问操作，你可能会想要用 JTA API 去确定整个事务，这个事务将包含两个操作的开始、回滚和提交。

尽管几乎每一个 J2EE 分布式应用程序都不可能离开数据库与 JDBC，但 JDBC 并不是 J2EE 规范的一部分，事实上，它被包含在 J2SE 中。本书不拟过多讨论该项技术。

有关 JDBC 的更多信息，请参考 JDBC 指南：

http://java.sun.com/products/jdbc/

3.2.2　Java 事务（JTA/JTS）

“事务的概念是一个重要的编程范例，其目的在于简化既要求可靠性又要求可用性的应用程序结构，特别是那些需要同时访问共享数据的应用程序。事务的概念最早用于商务运作的应用程序中，其中它被用于保护集中式数据库中的数据；再往后，事务的概念已扩展到分布式计算的更广泛的环境中；今天，事务是构建可靠的分布式应用程序的关键，这一点已得到广泛承认。”（参见 OMG：“Transaction Service Specification”）

简单的说，事务是具有如下特征的工作单元：

- 原子性：如果因故障而中断，所有结果均撤销
- 一致性：事务的结果保留不变的特性

- 孤立性：中间状态对其他事务是不可见的
- 持久性：已完成的事务的结果是持久的

事务的终止有两种方式：提交（commit）一个事务会使其所有的更改永久不变，而 回滚（rolling back）一个事务则撤销其所有的更改。

对象管理组织（OMG）为一种面向对象的事务服务，即对象事务服务（OTS），创建了一种规范。OTS 是 EJB 体系结构内的事务服务的基础。下列事务规范就是为 enterprise bean 所采用的事务模型而设：

- OMG 的对象事务服务 (OTS)
- Sun 公司的 Transaction Service (JTS) 和 Java Transaction API (JTA)
- 开放组 (X/Open) 的 XA 接口

Java Transaction Service 是 OTS 的 Java 映射，在 org.omg.CosTransactions 和 org.omg.CosTSPortability 这两个包中定义。JTS 对事务分界和事务环境的传播之类的服务提供支持，JTS 功能由应用程序通过 Java Transaction API 访问。

Java Transaction API 指定事务管理器与分布式事务中涉及的其他系统组件之间的各种高级接口，这些系统组件有应用程序、应用程序服务器和资源管理器等。JTA 功能允许事务由应用程序本身、由应用程序服务器或由一个外部事务管理器来管理。JTA 接口包含在 javax.transaction 和 javax.transaction.xa 这两个包中。

XA 接口定义了资源管理器和分布式事务环境中外部事务管理器之间的约定。外部事务管理器可以跨多个资源协调事务。XA 的 Java 映射包含在 Java Transaction API 中。

遗憾的是使用 JTA 处理分布式事务仍然很繁琐且易出错，更多情况下它仅仅是服务器实现者所使用的低级 API，而不为企业级程序员所使用。

有关 Java 事务的更多信息，请参考 JTA 和 JTS 指南：

http://www.java.sun.com/products/jta/

http://www.java.sun.com/products/jts/

3.2.3 Java Naming and Directory Interface（JNDI）

JNDI API 用于执行名字和目录服务，其接口包含在 javax.naming 和它的子包中。它为应用程序提供标准的目录操作的方法，例如获得对象的关联属性、根据它们的属性搜寻对象等。使用 JNDI，一个 J2EE 应用程序可以存储和动态获取任何类型的命名 Java 对象。

因为 JNDI 不依赖于任何特定的执行，应用程序可以使用 JNDI 访问各种命名目录服务，包括现有的诸如 LDAP、NDS、DNS、NIS、COS 命名和 RMI 注册等服务。这使得 J2EE 应用程序可以和传统的应用程序和系统共存。

JNDI 分为两部分：应用程序编程接口 (API) 和服务供应商接口 (SPI)，前者允许 Java 应用程序访问各种命名和目录服务，后者则是设计来供任意一种服务的供应商（也包括目录服务供应商）使用。这使得各种各样的目录服务和命名服务能够透明地插入到使用 JNDI API 的 Java 应用程序中。

JNDI 命名体系结构的关键概念包括：

- 对象和名称之间的绑定，名称可分为原子名称、复合名称和合成名称。原子名称是不可分割的，可以绑定到一个对象上；复合名称是原子名称的组合；合成名称则跨越多个命名系统。
- 若干称为命名上下文的绑定集，所有的命名操作均是在上下文对象上进行的，并且名称解析过程总是从最初的命名上下文开始。
- 命名系统，即若干组命名上下文。
- 命名空间，指一个命名系统中的所有名称。

在 JNDI 中，一个目录结构中的每一个节点被称为 context，每一个 JNDI 的名字都是与一个 context 相对的。一个应用可以使用 InitialContext 类来得到它的第一个 context，通过这个初始的 context，应用可以经过目录树定位到需要的资源或者对象。除了检索对资源或对象的应用外，JNDI 还可以提供方法做到：

- 插入或者绑定一个对象到一个 context 中
- 从一个 context 中移去一个对象
- 列出一个 context 中的所有对象
- 创建和删除 subcontexts

有关 JNDI 的更多信息，请参考 JNDI 指南：

http://java.sun.com/products/jndi/

3.2.4　Java Servlet

Java Servlet 实质上是一种小型的、与平台无关的 Java 类，它由容器管理并被编译成平台无关的字节代码，这些代码可以动态地加载到一个 web 服务器上，并由该 web 服务器运行。Servlet 通过一种由 servlet 容器实现的请求——响应模型与 Web 客户机进行交互。这种请求——响应模型建立在超文本传输协议（HTTP）行为的基础之上。Java Servlet API 提供了一种通用机制，对于任何使用了基于请求——响应协议的服务器，这种机制可以扩展其功能。Java Servlet API 接口定义在 java.servlet 和 javax.servlet.http 包中。

随着越来越多的 Web 服务器的支持，Servlet 逐渐取代了基于 Java 的 CGI 脚本语言。与 CGI 脚本语言相比，Servlets 在可伸缩性上提供了很好的改进：每一个 CGI 在开始的时候都要求开始一个新的进程，而 Servlets 在 Servlet 引擎中是以分离的线程来运行的。此外相比于现在的竞争者 ASP 和 JavaScript，Servlet 的主要优点在于其平台无关性。

Java Servlet API 与其他 Java Enterprise API 不同之处在于它不是一个在现存的网络服务和协议之上的 Java 层；相反，类似于 CGI 应用程序，它本身往往不是完整的应用程序。在处理接收自 Web 浏览器上用户的信息请求时，CGI 只是整个处理过程中的一个中间步骤。例如，CGI 应用程序的一种常见用途是访问数据库；将它用于这种任务时，CGI 程序提供一种方法，将用户的数据请求连接到能满足这种请求的企业数据库，并向用户发回请求结果作为响应。Java servlet 与 CGI 程序相似，适合于充当连接前端 Web 请求与后端数据资源的分布式中间件。

有关 Java Servlet 的更多信息，请参考 Java Servlet 指南：

http://java.sun.com/products/servlet/

3.2.5　JavaServer Pages

JavaServer Pages（JSPs）提供的功能大多与 Java Servlet 类似，不过实现的方式不同。Servlet 全部由 Java 写成并且生成 HTML；而 JSP 通常是大多数 HTML 代码中嵌入少量的 Java 代码。

JSP 是对 HTML 的一种扩展，它可以通过一些特殊的标签向静态 HTML 页面中插入动态的信息。如可以利用<%和%>标签添加 Java 代码段，用<%=表达式%>将表达式的值写入页面，用<jsp:bean>标签在某一范围内（request、session 或 context）引用 Java Bean。

除此之外，JSP 的标准标签扩展机制还允许开发人员编写自己的标签和相应的实现方法。这样就可以将某些商业逻辑封装成 JSP 的标签，使 JSP 文件更加简单、易于实现。

当一个浏览器向服务器请求一个 JSP 文件时，这个 JSP 文件首先被 Web 应用服务器编译成 servlet 并执行，然后将所产生的结果作为一个 HTML 文件传给浏览器。只要在 JSP 文件中加入一些控制，便可轻易的实现对数据的动态显示。以后，如果再有对这个 JSP 文件的请求，且该文件没有作任何修改，它将不会再被编译，而是直接执行已编译好的 servlet。

有关 JSP 的更多信息，请参考 JSP 指南：

http://java.sun.com/products/jsp/

3.2.6　EJB

J2EE 技术赢得广泛重视的主要原因之一就是 EJB（Enterprise JavaBean），要注意的是 EJB 从技术上而言不是一种产品，而是一个技术规范。它提供了一个框架来开发和实施分布式商务逻辑，由此简化了具有可伸缩性和高度复杂的企业级应用的开发；此外它还定义了 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。有关 EJB 框架的 API 定义在 javax.ejb 及其子包 javax.ejb.spi 中。

为了弄清楚 Enterprise JavaBean 的概念，可以先比较一下它与 Java 常用的 JavaBean 左翼比较。JavaBeans 是 Java 的组件模型，在 JavaBeans 规范中定义了事件和属性等特征。Enterprise JavaBeans 也定义了一个 Java 组件模型，但 Enterprise JavaBeans 组件模型和 JavaBeans 组件模型是不同的。JavaBeans 的重点是允许开发者在开发工具中可视化的操纵组件，它解释了组件间事件登记、传递、识别和属性使用、定制和持久化的应用编程接口和语意。Enterprise JavaBeans 的侧重点则是详细地定义了一个可以移植的 Java 组件的服务框架模型。因此，其中并没提及事件——Enterprise bean 通常不发送和接受事件；同样也没有提及属性——属性定制并不是在开发时进行，而是在运行时（实际上在部署时）通过一个部署描述符来描述。

为了满足架构的目标，EJB 规范中定义了如下一些新的概念。

EJB 服务器（EJB Server）负责管理 EJB 容器（它负责管理 Beans），提供对操作系统

服务的存取和 Java 相关的服务，尤其是通过 JNDI 访问命名空间和基于 OTS 的事务处理服务。

EJB 容器（EJB Container）负责管理部署于其中的 Enterprise Bean。客户机应用程序并不直接与 Enterprise Bean 进行交互。相反，客户机应用程序通过由容器生成的两个封装接口与 Enterprise Bean 进行交互。当客户机使用封装接口调用各种操作时，容器截获每个方法调用，并插入管理服务。对客户端而言 EJB 容器是透明的。

EJB 客户端（EJB Client）可以是 servlet、JSP、应用程序或其它 bean。客户端可以通过 JNDI 来查找 EJB home 接口，步骤如下：首先创建一个 JNDI Context (initial context)；然后使用 JNDI Context 来查找 EJB Home 接口；再使用 EJB Home 接口来创建或查找 EJB 实例；最后使用 EJB 实例完成业务操作。注意实际的存取 (对 EJB) 是通过容器生成的类来完成。

每个 Enterprise JavaBean 都必须实现 Home 接口（扩展自 EJBHome 接口）和 Remote 接口（扩展自 EJBObject 接口），简单来说 Home 接口定义了如何查找或创建 bean，Remote 接口定义了 bean 的业务方法。

图 1-2 大致描述了 EJB 架构中上述概念的关系。

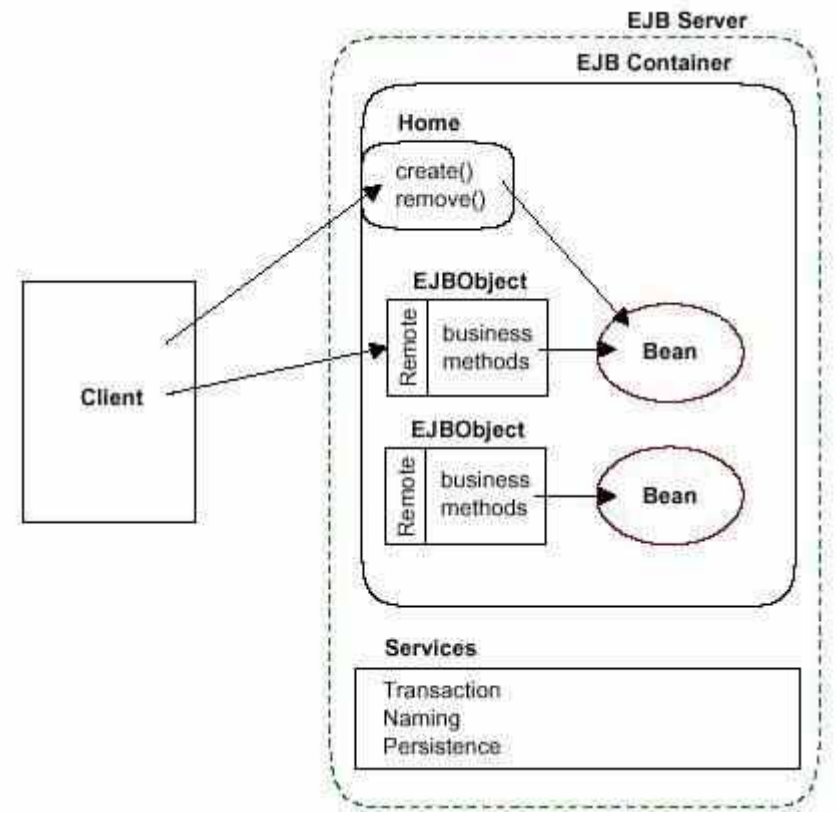

图 3-2　EJB 体系结构

有关 EJB 的更多信息，请参考 EJB 指南：

http://java.sun.com/products/EJB/

3.2.7　J2EE 部署

在 J2EE 部署体系中，表示 J2EE 平台产品所需要的动态部署配置信息的组件采用了 JavaBeans 结构，因为这种结构既适于表示简单又适用于表示复杂组件，同时它还有很强的平台无关性。这些 Beans 使得开发简单的属性页、编辑器和复杂的定制向导（它可以引导部署者完成应用程序部署配置各步骤）成为易事。

J2EE 部署 API 包括如下内容：

- J2EE 平台产品必须实现的一套最小工作集。所有的 J2EE 平台产品提供商都必须向工具提供商提供这套工作集的实现，它使得可移植应用程序可部署到不同的 J2EE 平台产品上。
- 部署工具所必须实现的一套最小工作集。所有的 J2EE 工具提供商都必提供这套工作集的实现，以与不同的 J2EE 平台产品交互。

这套 API 描述了三个部署步骤中的两步：安装与配置，第三步（运行）留给了 J2EE 平台产品提供商。这些提供商可以在其自己的部署工具中扩展上述最小工作集以与其他厂商竞争，这些扩展可能对其他厂商的部署工具不可用。

关于 J2EE 部署的更多信息，请参考 J2EE 部署指南：

http://java.sun.com/j2ee/tools/deployment/

3.2.8　JavaMail 与 JAF

JavaMail API 是一个用于阅读、编写和发送电子消息的可选包（标准扩展），可以用来建立基于标准的电子邮件客户机，它配置了各种因特网邮件协，包括 SMTP、POP、IMAP 和 MIME，还包括相关的 NNTP、S/MIME 及其它协议。

通常开发 JavaMail 程序还需要有 Sun 的 JavaBeans Activation Framework（JAF）。JavaBeans Activation Framework 的运行很复杂，这里简单的说就是 JavaMail 的运行必须得依赖于它的支持，比如 JavaMail 利用 JavaBeans Activation Framework 来处理 MIME 编码的邮件附件。JavaMail 接口包含在 javax.mail 及其子包中，JavaBeans Activation Framework 接口包含在 javax.activation 包中，JAF 规范是"Glasgow" JavaBeans 规范的一部分，关于 JAF 的更多细节，请参考 JAF 指南：

http://java.sun.com/beans/glasgow/jaf.html

注意：MIME 的字节流可以被转换成 JAVA 对象，或者转换自 JAVA 对象。由此很多应用都不需要直接使用 JAF。

JavaMail API 可以分成两部分以支持独立于协议的发送和接受消息：

- 应用程序组件，用来接收和发送mail的应用程序级接口，该接口独立于供应商和协议消息。
- 服务提供接口，用于提供对特定协议的支持，如 SMTP、POP、IMAP 和 NNTP。除了 JavaMail API 外，要与服务器通信，还需要每个协议的供应商（provider）。

核心 JavaMail API 由七个类组成：Session、Message、Address、Authenticator、Transport、

Store 及 Folder，它们都来自 javax.mail、即 JavaMail API 顶级包。可以用这些类完成大量常见的电子邮件任务，包括发送消息、检索消息、删除消息、认证、回复消息、转发消息、管理附件、处理基于 HTML 文件格式的消息以及搜索或过滤邮件列表。

有关 JavaMail 的更多信息，请参考 JavaMail 指南：

http://java.sun.com/products/javamail/

3.2.9　RMI

Remote Method Invocation（RMI，远程方法调用）为分布式计算提供了一种高级的通用解决方案。正如名字所显示的那样，RMI 即调用远程对象上的方法，它使用了连续序列方式在客户端和服务器端传递数据，将面向对象编程模型扩展到了客户机/服务器系统，使开发者可以用本地对象调用的语法进行远程调用。RMI 在 java.rmi 及其子包中实现。

为了实现位置透明性，RMI 使用标准机制：存根（stub）和框架（skeleton）。

存根是代表远程对象的客户机端对象，它具有和远程对象相同的接口或方法列表，当客户机调用存根方法时，将执行下列操作：

- 初始化与包含远程对象的服务器端虚拟机的连接
- 对服务器端虚拟机的参数进行列集（Marshal）并通过 RMI 基础结构将请求转发到远程对象
- 等待方法调用结果
- 散集（Unmarshal）返回值或返回的异常
- 将值返回给调用程序

调用存根的方法时为了向调用程序展示比较简单的调用机制，存根隐藏了参数的序列化和网络级通信等细节。

在服务器端，框架对象处理“远方”的所有细节，因此实际的远程对象不必担心这些细节。也就是说，您完全可以象编码本地对象一样来编码远程对象。框架将远程对象从 RMI 基础结构分离开来。在远程方法请求期间，RMI 基础结构自动调用框架对象，框架对象依次执行如下操作：

- 散集远程方法的参数
- 调用实际远程对象实现上的方法
- 将结果（返回值或异常）列集并通过 RMI 基础结构将列集后结果返回给调用程序

关于这种设置的最大的好处是，开发者不必亲自为存根和框架编写代码。JDK 包含 rmic 编译器，它会自动创建存根和框架的类文件。

利用 RMI 编写分布式对象应用程序，开发者必须完成如下工作：

1．定位远程对象

应用程序可使用如下两种机制中的一种得到对远程对象的引用：

- 用 RMI 的简单命名工具 rmiregistry 来注册它的远程对象
- 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

2．与远程对象通信

因为 RMI 基础结构隐藏了远程对象间通信的细节，因此对于程序员来说，远程通信和标准的 Java 方法调用并没有差别。

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，使得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。

Java RMI 实现的功能相当完整，但仍然简单易用。其简单性首先得益于通信双方都要求用 Java 语言编写（Java 平台提供了强大的网络通信支持），这使得通信双方能共享相同的数据类型并拥有 java.io 包中定义的串行和并行化对象。但这也同时意味着 RMI 不能为非 Java 语言的编写的分布式应用使用。不幸的是，事实上存在着大量这样的应用，它们的客户机和服务器并不全是用 Java 语言编写的。在这种环境下，Java 2 平台提供了另一种解决方案。

有关 RMI 的更多信息，请参考 RMI 指南：

http://java.sun.com/products/rmil/

3.2.10　JAVA IDL/CORBA

正如上文提到的，在异构网络（客户机和服务器可能是使用任何语言编写的）中，Java 2 平台提供了另一种解决方案已取代只能在 java 环境中使用的 RMI，这就是基于 CORBA（Common Object Request Broker Architecture，公用对象请求代理体系）的分布式对象远程调用方法。

与 RMI 不同，CORBA 不属于 Java 平台本身。OMG（Object Management Group，对象管理组织）发明了 CORBA 规范，CORBA 被设计成与平台和语言无关。因此，CORBA 对象可以运行于任何平台之上，位于网络的任何位置，还可以用任何语言（包括 Java、C、C++和 Smalltalk 等）编写，只要该语言具有 IDL（Interface Definition Language，接口定义语言）的映射。

与 RMI 相比，CORBA 是为更大、可伸缩更强的系统准备的，在这些系统中可能有数千个对象；CORBA 的编程和部署比 RMI 更复杂，但允程序员开发需要事务、安全性等支持的企业级系统；CORBA 的命名服务也比 RMI 命名注册功能更强大和灵活。表 3-1 提供了 RMI 和 CORBA 技术的一个比较列表。

CORBA 的实现称为 ORB（Object Request Broker，对象请求代理）。Java IDL 即是 CORBA 的一个实现，它是 JDK1.3 或更高版本的核心软件包之一，定义在 org.omg.CORBA 及其子包中。

在 Java IDL 的支持下，开发人员可以使用如下两种方法将 Java 和 CORBA 集成在一起：

- 创建 Java 对象并使之可在 CORBA ORB 中展开，
- 创建 Java 类并作为和其它 ORB 一起展开的 CORBA 对象的客户。这种方法提供

了另外一种途径，通过它 Java 可以被用于将你的新的应用和以前遗留的系统相集成。

表 3-1　RMI 和 CORBA 技术比较

比　较　项	RMI	CORBA
所支持语言	Java	Java、C++、C、Smalltalk 及其它语言
运行时服务	命名	命名、生命周期、持久性、事务及其它
编程和设置的容易程度	优	良
可伸缩性	良	优（取决于 ORB 供应商）
性能	良	优（取决于 ORB 供应商）

关于 Java IDL/CORBA 的更多信息，请参考 Java IDL 指南：

http://java.sun.com/products/jdk/idl/

3.2.11　JMS

在讨论 Java 消息服务（Java Message Service，简称 JMS）之前，先必须弄清企业消息传递系统的概念。

企业消息传递系统通常又称为面向消息的中间件（Message Oriented MiddleWare，简称 MOM），它使用松散耦合的、非常灵活的方式来集成应用程序，在存储和转发的基础上支持应用程序间数据的异步传递；每个应用程序都只与作为中介的 MOM 通信。

JMS 就是用于和企业消息传递系统相互通信的应用程序接口（图 3-5 简单描述了这一通信模型）。在 JMS 出现之前，每个 MOM 供应商都提供专有的的 API 以供客户程序访问它们的产品；而 JMS 为 Java 客户程序通过 MOM 产品收发消息提供了一种标准可移植的方法——用 JMS 编写的程序能够在任何实现了 JMS 标准的 MOM 上运行。

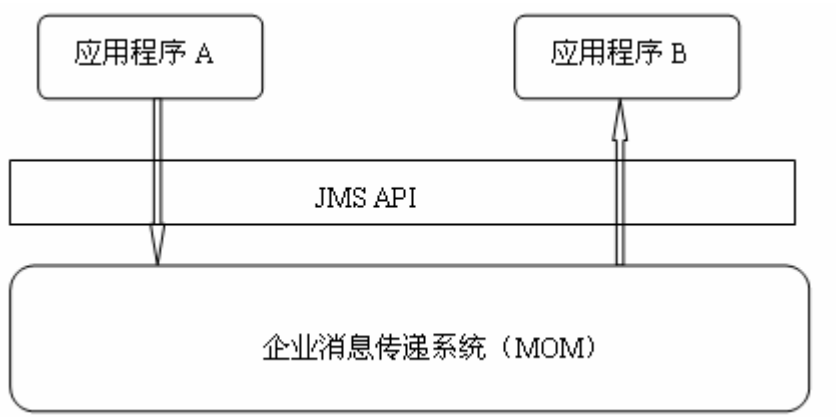

图 3-5　JMS 通信模型

JMS 可移植性的关键在于这样一个事实：JMS API 由 Sun 作为一组接口提供，提供 JMS 功能性的产品通过支持一个实现这些接口的供应商就能做到这一点。作为开发者，您通过定义一组消息和一组交换那些消息的客户机应用程序来构造一个 JMS 应用程序。

在 JMS 1.0.2 规范中定义了两种类型的消息传递域（它们是相互独立的），即点对点（P2P）域和发布/订阅（P/S）域，同时提供了对 XA（分布式）事务的兼容。JMS 的最新版本 1.1 几乎统一了这两个域，使得客户机基本不必再具体针对某一个域进行实现，这极大地简化了 JMS API，并为开发者创建更通用、重用性更好的消息传递代码提供了机会。在未来的版本中那些特定于域的子接口很可能会被完全去掉。

JMS API 定义在 javax.jms 包中，关于 JMS 的更多信息，请参考 Java JMS 指南：

http://java.sun.com/products/jms/

3.2.12　Java XML

XML 是一种可以用来定义其它标记语言的元语言。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 有着相同的目标，即平台独立性。通过 JAVA 和 XML 的组合，可以得到一个完美的具有平台独立性的解决方案。

Sun 目前所提供的支持 XML 的 Java API（包含了支持 Web 服务的主要组件），主要可以分为以下几个部分：

1．用于 XML 解析的 JAXP

JAXP（Java API for XML Processing）是一个可插 API，它对于 W3C 所推荐的 XML API（即 SAX、DOM 和 XSLT）的任意供应商实现都是开放的，为获得 XML 解析器提供了标准接口。

所有的 J2EE 实现都必须满足 JAXP 的要求，提供至少一个 SAX2 解析器、至少一个 DOM2 解析器和至少一个 XSLT 转换器；SAX 解析器和 DOM 解析器都必须支持 DTD 或 XML 模式。

关于 JAXP 的更多信息，请参考 JAXP 指南：

http://java.sun.com/xml/jaxp/

2．用于 XML 消息传递的 JAXM

JAXM（Java API for XML Messaging）旨在使用纯 Java API 使应用程序能够发送与接收面向文档的 XML 消息。JAXM 的目的是为更高级别的、基于标准的并且基于 SOAP 的消息传递协议提供一个基础。JAXM 现在主要用于异步消息（下面要讨论的 SAAJ 主要处理同步消息），当以异步方式使用 JAXM 时，它使用消息传递提供程序来促进消息的路由选择。

关于 JAXP 的更多信息，请参考 JAXM 指南：

http://java.sun.com/xml/jaxm/

3．带有附件的 SOAP 消息处理 SAAJ

SAAJ（SOAP with Attachments API for Java）用于操作那些遵循 SOAP1.1 规范的消息及其所包含的 SOAP 附件。SAAJ 最初是作为 JAXM 1.0 规范的一部分进行定义的；但在

JAXM 1.1 种，SAAJ 已被分离出来自成一个规范，这样其他规范就能够依靠 SAAJ 包而无需依靠 JAXM。SAAJ API 通常为 JAX-RPC 使用并通过 JAX-RPC 消息句柄存取整个 SOAP 消息。

关于 SAAJ 的更多信息，请参考 SAAJ 指南：

http://java.sun.com/xml/saaj/

4．基于 XML 的远程方法调用 JAX-RPC

JAX-RPC（Java API for XML-based RPC）是用于支持 SOAP、基于 XML 的远程方法调用的实现包，它定义了客户机用于操作 Web 服务的编程接口（反之，J2EE 的 WebService 规划也描述了基于基于 JAX-RPC 的服务和客户机的部署）。核心 JAX-RPC 包含一个用于调用 Web 服务的功能的完备的 JAX-RPC 客户机以及 JAX-RPC 服务器的一个参考实现。

关于 JAX-RPC 的更多信息，请参考 JAX-RPC 指南：

http://java.sun.com/xml/jaxrpc/

5．用于 XML 注册的 JAXR

JAXR（Java API for XML Registries）定义了客户机用以访问基于 XML 的注册中心的统一编程接口，注册中心通常用来存储已发布的 Web 服务的信息，而 JAXR API 则提供了访问这种信息的统一的方法。当前 Web 服务中最常用的基于 XML 注册中心是 UDDI（Universal Description, Discovery, and Integration，统一描述、寻找和集成）注册中心；JAXR 包含了一个用于访问 UDDI 的供应商程序。

关于 JAXR 的更多信息，请参考 JAXR 指南：

http://java.sun.com/xml/jaxr/

3.2.13　Web 服务

Web 服务（WebService）现在并没有被广泛接受的定义，简略的来说，它是一种面向服务的体系结构，能够创建服务的抽象定义、提供服务的具体实现、发布并查找服务、实现服务实例选择，并实现可互操作服务的使用。在 JSR 109 公开的最终草案版本 v1.0 中，Web 服务被描述为具有以下特征的组件：

- Web 服务的任一实现将实现能够由 WSDL 描述的接口的方法，方法将通过使用无状态会话 EJB 或 JAX-RPC Web 组件来实现。
- Web 服务的接口在部署期间可能会被发布到一个或多个 Web 服务注册中心。
- 只使用由本规范描述的功能的 Web 服务实现可以被部署到任何遵循 Web Services for J2EE 的应用程序服务器中。
- 服务实例（称为端口（Port））由容器创建和管理。
- 运行时服务需求（如安全性属性）与服务实现是分开来的。您可以在组装或部署期间使用工具来定义这些需求。
- 容器将充当对服务进行访问的媒介。

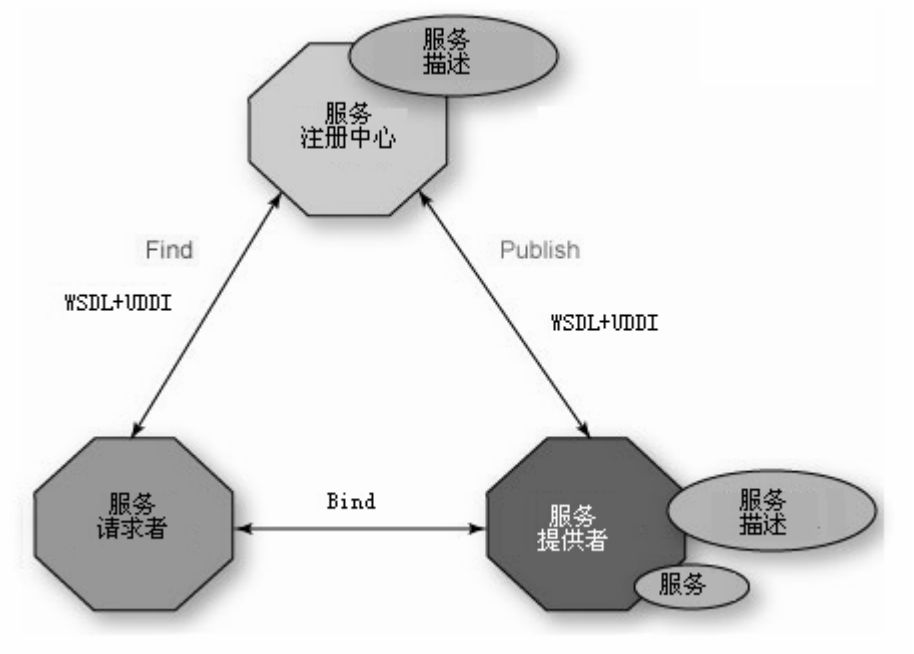

图 3-6　Web 服务体系结构

图 3-6 描述了 Web 服务体系结构，服务提供者使用 Web 服务描述语言（Web Services Description Language，简称 WSDL）来定义抽象的服务描述。然后抽象的服务描述将用 WSDL 生成具体的服务描述，从而创建出具体的服务。接下来，具体的服务描述可以被发布到类似 UDDI 这样的注册中心。服务请求者可以使用注册中心来找到服务描述，并根据服务描述选择和使用服务的具体实现。

抽象服务描述在 WSDL 文档中被定义为端口类型（Port Type）。具体的服务实例是由端口类型、传输和编码绑定以及作为 WSDL 端口的地址共同定义的。多组端口聚集成一个 WSDL 服务。

Web 服务的成功很大程度上是是因为 Web 服务使用 XML 作为数据格式基础，将其用于描述 WSDL 或 SOAP，使应用程序可以跨异构编程语言和操作系统进行交互。

J2EE 为 Web 服务客户端和端点（endpoint）提供了完全支持，这些支持建立在上述几种技术的基础上：

- JAX-RPC 为客户端通过 SOAP/HTTP 协议调用 Web 服务提供支持，它同时还定义了 Java 类和用于 SOAP 远程方法调用的 XML 文档间的映射机制。
- SAAJ 为操纵底层 Soap 消息提供支持。
- Web Services 规范定义了 Web 服务客户端与 Web 服务端口在 J2EE 体系中的部署，并描述如何使用 EJB 实现 Web 服务端点。
- JAXR 为客户访问 XML 注册中心提供支持。

关于 Web Service 的更多信息，请参考 Web Service 指南：

http://java.sun.com/webservices/

3.2.14　JMX

JMX（Java Management Extensions）的前身是 Java Management API，它致力于解决分布式系统管理的问题。JMX 是一种应用编程接口、可扩展对象和方法的集合体，可以跨越各种异构操作系统平台、系统体系结构和网络传输协议，开发无缝集成的面向系统、网络和服务的管理应用。

JMX 是一个完整的网络管理应用程序开发环境，它同时提供了厂商需要收集的完整的特性清单、可生成资源清单表格、图形化的用户接口；访问 SNMP 的网络 API；主机间远程过程调用；数据库访问方法等。

关于 JMX 的更多信息，请参考 JMX 指南：

http://java.sun.com/j2ee/tools/management/

3.2.15　JACC

Java 容器授权合同（Java Authorization Service Provider Contract for Containers，简称 JACC）在J2EE应用服务器和特定的授权认证服务器之间定义了一个连接的协约，以便将各种授权认证服务器插入到J2EE产品中去。

关于 JACC 的更多信息，请参考 JACC 指南：

http://java.sun.com/j2ee/javaacc/

3.2.16　JCA

Java 连接器体系（Java Connector Architecture，简称 JCA）用一种安全的、事务性的方法连接 J2EE 应用程序和非 J2EE 环境（通常是 EIS 系统，如 ERP、大型机事务处理、数据库等），它帮助开发者进行不同种类的 EIS 之间的无缝集成。JCA 连接器一方面与 J2EE 应用服务器建立系统级连接，另一方面与访问 EIS 资源的应用组件建立应用级连接。和其他服务 API 类似，JCA API 在统一接口的同时，为开发商开发各具特色的资源适配器产品提供空间。利用 JCA API 的解决方案比基于 JMS 的解决方案与后端耦合得更紧；更确切地说是 JCA 规范可以在同一次消息交换或同一个事务中把消息的发送和处理结合起来。

JCA 规范定义了应用程序和 EIS 间各种级别的接口，它基本上是描述如何在应用程序服务器和后端应用程序间部署所谓的资源适配器（RA）。资源适配器存在于应用程序服务器的地址空间内，并实现了允许应用程序服务器和 EIS 间交互的编程接口。

JCA 规范中定义了两种级别的编程接口。其中一种被称为公共客户机接口（common client interface，简称 CCI），任何 J2EE 组件都可以用这种接口与 EIS 进行交互。它的使用是可选的，这意味着资源适配器不一定非得实现它；EIS 供应商可以自由使用它自己的接口。这一点很有意义，特别是当这个接口已经被用一种更早的集成方法实现过的时候，就更有意义。例如，一个数据库可能已经有了连接到数据库并处理 SQL 语句的包装器类。

JCA 定义的另一种接口是一个系统编程接口（system programming interface，简称 SPI）

集合，它们只被应用程序服务器内部使用。SPI 分为三种类别，用来解决下面几个问题：

- 安全性（Security）接口允许应用程序服务器和资源适配器处理在 EIS 上的登录，如通过隐式方法把用户标识和密码组合发送到代表客户机的后端。
- 连接池（Connection pooling）接口确保与 EIS 进行通信所需的任何资源都在许多客户机间被共享，并被合用。
- 事务处理（Transaction handling）接口负责管理跨支持两阶段提交协议的各种后端的分布式事务。

图 3-7 描述了 JCA 规范定义的各种类型的接口间的逻辑关系。

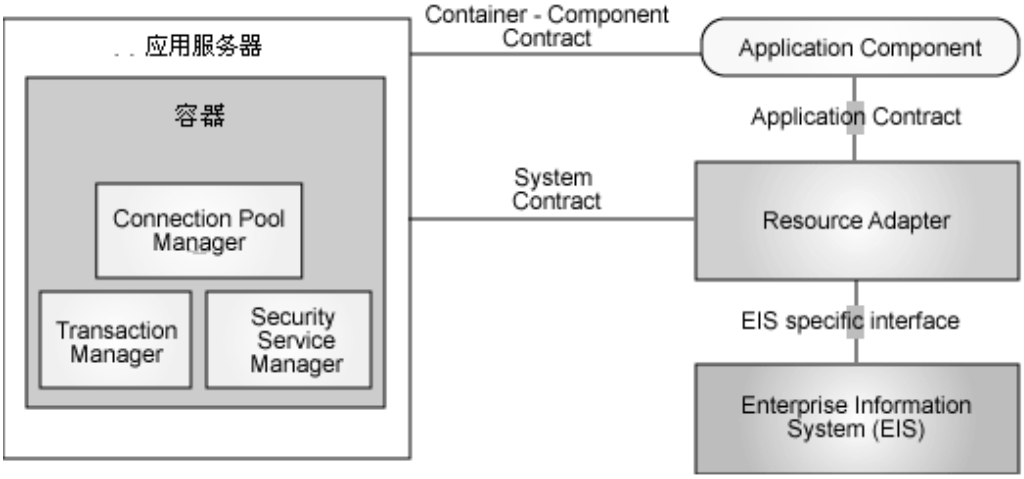

图 3-7　JCA 结构

关于 JCA 的更多信息，请参考 JCA 指南：

http://java.sun.com/j2ee/connector/