EJB 核心规范

正式版

译者：卫建军

2008/1/5

译者序

Java 是当前 IT 领域中比较流行的技术之一。J2EE 是当前比较流行的企业级应用架构。本人一直致力于 J2EE 架构的学习和研究，但是总是对英文文档有不可言语的恐惧。我想很多 J2EE 爱好者和我有同样的感觉。这样就影响了我们深入学习 J2EE 原始规范的兴趣。但是 J2EE 原始的规范文档对我们深入理解 J2EE 有很大的帮助，因为它阐述了规范的来龙去脉，以及违反了规范会造成什么样的影响。了解了这些缘由和影响，会使我们对 J2EE 架构有更深层次的理解。这也是我翻译该规范的动力所在。

由于本人的英语水平有限，翻译中难免会出现错误和拗口之处，请大家多多指教。

这次主要翻译的规范有《EJB3 规范简化版》、《J2EE5.0 规范》、《EJB 核心规范》、《EJB3 持久化规范》和《JMS1.1 规范》。希望对大家有所帮助。

卫建军
2008-1-5 于北京

目录

1 介绍

这是 EJB 架构规范。EJB 架构是一个基于组件的业务应用的开发和部署的架构。用 EJB 架构开发的应用是可伸缩的、事务的和多用户安全的。这些应用可以开发一次，然后部署到任何支持 EJB 规范的服务平台上。

1.1目标读者

本规范的目标读者是事务处理平台的提供商，企业应用工具的提供商，O/R 映射产品的供应商和其它希望在它们的产品中支持 EJB 技术的工业商。

本文档中描述的许多概念都是系统级的问题，这些问题对 EJB 应用开发者是透明的。

1.2EJB3.0 的新特性

本文档中的 EJB3.0 架构扩展了 EJB，它包括下面的新功能和精简了早期 EJB API：

- *定义了能用于注释 EJB 应用的 java 语言元数据注释。这些元数据注释用于简化开发者的任务，减少了开发者需要实现的程序类和接口，以及降低了对开发者提供 EJB 部署描述的要求。*
- *指定程序的缺省值，包括元数据的缺省值，减少了开发者指定容器的公共的期望的行为和要求。任何时候都是采用“按例外配置”的方式。（译者注：就是只配置特殊需求）*
- *封装环境依赖和通过使用注释、依赖注入和简单的 lookup 机制来获取 JNDI。*
- *简化企业 bean 的类型。*
- *取消了对会话 bean 组件接口的要求。会话 bean 要求的业务接口可以是一个普通的 java 接口而不是 EJBObject，EJBLocalObject 或 java.rmi.Remote 接口。*
- *取消了对会话 bean 的 home 接口的要求。*

- *通过 Java 持久化 API【2】简化了实体的持久化。支持轻量级的域模型，包括继承和多义性。*
- *取消了持久化实体【2】要求的所有接口。*
- *为持久化实体的 O/R 映射指定了 java 语言元数据注释和 XML 配置描述元素【2】。*
- *Java 持久化的查询语言扩展了 EJB QL，并增加了投影、显式内连接和外连接操作、批更新和批删除、子查询和 group by。为本地 SQL 查询增加了动态查询的能力。*
- *为会话 bean 和消息驱动 bean 提供了拦截器功能。*
- *减少了需要检查的异常的使用要求（checked exception）。*
- *取消了实现回调接口的要求。*

1.3EJB3.0 专家组

EJB3.0 规范的工作是在 Java 社团处理程序下的 JSR-220 的一部分。这个规范是 EJB3.0 专家组协同工作的成果。下面列出了正式的专家组成员：Apache 软件组织：Jeremy Boynes；BEA：Seth White；Borland: Jishnu Mitra, Rafay Khawaja; E.piphany: Karthik Kothandaraman; Fujitsu-Siemens: Anton Vorsamer; Google: Cedric Beust; IBM: Jim Knutson, Randy Schnier; IONA: Conrad O'Dea; Ironflare: Hani Suleiman; JBoss: Gavin King, Bill Burke, Marc Fleury; Macromedia: Hemant Khandelwal; Nokia: Vic Zaroukian; Novell: YongMin Chen; Oracle: Michael Keith, Debu Panda, Olivier Caudron; Pramati: Deepak Anupalli; SAP: Steve Winkler, Umit Yalcinalp; SAS Institute: Rob Saccoccio; SeeBeyond: Ugo Corda; SolarMetric: Patrick Linskey; Sun Microsystems: Linda DeMichiel, Mark Reinhold; Sybase: Evan Ireland; Tibco: Shivajee Samdarshi; Tmax Soft: Woo Jin Kim; Versant: David Tinker; Xcalia: Eric Samson, Matthew Adams; Reza Behforooz;Emmanuel Bernard; Wes Biggs; David Blevins; Scott Crawford; Geoff Hendrey; Oliver Ihns;Oliver Kamps; Richard Monson-Haefel; Dirk Reinshagen; Carl Rosenberger; Suneet Shah。

1.4文档的组织

本规范分成下面三个文档：

- EJB3.0 简化的 API
- EJB 核心规范和要求

- Java 持久化 API

文档“EJB3.0 简化的 API”提供了对由 EJB3.0 正式版引入的简化 API 的概述。

文档“Java 持久化 API”和 Java 持久化查询语言（EJB QL 的超集）合起来是持久化管理的新的 API 规范。它提供了 EJB3.0 正式版需要支持的持久化 API 的定义，以及 Java 持久化 API 如何用在 Java SE 环境的定义。

文档“EJB 核心协议和要求”定义了 EJB 的使用和实现的协议和要求。这些协议包括用 EJB3.0 简化 API 的协议，以及用于 EJB2.1 API 的协议（在这个版本中也要求支持）。也包括定义在“Java 持久化 API”中的协议和要求。

1.5文档约定

正规 Times 字体用于 EJB 规范说明。

斜体字用于描述信息的段落，例如，通常使用的注意描述，或者用于与规范进行区分的注意信息。

Courier 字体用于代码样例。

2 概述

2.1总体目标

EJB 架构有下列的目标：

- *EJB 架构将是使用 Java 语言建立面向对象业务应用的标准组件架构。*
- *EJB 架构将是使用 Java 语言建立分布式业务应用的标准组件架构。*
- *EJB 架构将支持开发、部署和使用 web 服务。*
- *EJB 架构将使得编写应用更加容易：应用开发者将不必理解底层事务与状态管理，多线程，连接池或其他复杂的底层 API。*
- *EJB 应用将遵循 Java 语言的“编写一次，可以在任何地方运行”的理念。企业 bean 可以被开发一次，然后可以被部署到多个平台，而不需要重新编译或更改源代码。*

- *EJB 架构将涉及企业应用生命周期的开发、部署和运行时的各个方面。*
- *EJB 架构将定义可以让来自多个供应商的工具能够开发和部署在运行时可以进行交互的组件的协议。*
- *EJB 架构将使得能够使用不同供应商的工具开发的组件来共同构建应用。*
- *EJB 架构将提供企业 bean 和 Java 平台，Java EE 组件以及非 java 语言应用之间的交互能力。*
- *EJB 架构将兼容现存的服务器平台。供应商可以通过扩展现存的产品来支持企业 bean。*
- *EJB 架构将兼容其他 Java 语言 API。*
- *EJB 架构将兼容 CORBA 协议。*

EJB3.0 的目的是持续达到这些目标和从企业应用开发者的角度通过减少它的复杂性来改进 EJB 架构。

2.2EJB 角色

EJB 架构在应用开发和部署生命周期中定义了七个不同的角色。每个 EJB 角色可以由不同的团体来扮演。EJB 架构指定了确保每个 EJB 角色的产品与其他角色的产品相互兼容的协议。EJB 规范关注于为支持 ISV 书写的企业 bean 的开发和部署而要求支持的那些协议。

在某些场景中，一个团体可以扮演几个 EJB 角色。例如，容器提供者和 EJB 服务器提供者可以是同一个供应商。或者一个程序员可以扮演企业 bean 提供者和应用组装者两个角色。

下面的章节定义了这七个 EJB 角色。

2.2.1 企业 bean 提供者

企业 bean 提供者（简称 bean 提供者）是企业 bean 的生产者。他或她的输出是 ejb-jar 文件，它包含了一个或多个企业 bean。Bean 提供者负责实现企业 bean

业务接口的 Java 类；定义 bean 客户端视图接口；以及指定 bean 的元数据。Bean 的元数据采用元数据注释和/或外部 XML 部署文件的形式。Bean 的元数据——不管是使用元数据注释还是部署文件——包括了企业 bean 的结构化信息和声明了 bean 对外部的所有依赖（例如，企业 bean 使用的资源的名称和类型）。

企业 bean 提供者通常是一个应用域专家。Bean 提供者开发可重用的企业 bean，它们通常实现了业务任务或业务实体。

不要求 Bean 提供者是一个系统级编程的专家。因此，Bean 提供者通常不在企业 bean 中编写事务，并发，安全，分布式或其他服务。Bean 提供者依赖 EJB 容器的提供的这些服务。

多个企业 bean 的 Bean 提供者常常扮演应用组装这的角色。

2.2.2 应用组装者

应用组装者将企业 bean 组装成更大的可部署应用单元。应用组装者的输入是一个或多个 ejb-jar 文件。应用组装者的输出是一个或多个包含了企业 bean 和它们的应用组装指南。

应用组装者也可以将企业 bean 和其它组成应用的应用组件类型组合在一起。

EJB 规范描述了应用组装步骤发生在企业 bean 部署之前的情况。但是，EJB 架构没有禁止在所有或部分企业 bean 被部署之后进行应用组装。

应用组装者是一个域专家，他使用企业 bean 组织应用。应用组装者使用企业 bean 的元数据注释和/或部署文件和企业 bean 的客户端视图协议来工作。尽管组装者必须熟悉企业 bean 客户端视图接口提供的功能，他或她不需要了解企业 bean 的实现。

2.2.3 部署人员

部署人员获取一个或多个由 Bean 提供者或应用组装者生产的 ejb-jar，并将包含在这些 ejb-jar 文件中的企业 bean 部署到一个特定的操作环境中。操作环境

包含了一个特定的 EJB 服务器和容器。

部署人员必须解决所有的 bean 提供者声明的外部依赖（例如，部属人员必须保证企业 bean 使用的所有资源管理器连接工厂出现在操作环境中，而且，他或她必须将它们绑定到在元数据注释或部署文件中声明的资源管理器工厂引用上），并且必须遵循由应用组装者定义的应用组装指南。部署人员使用 EJB 容器提供者提供的工具来扮演他或她的角色。

部署人员是一个特定操作环境的专家，他负责企业 bean 的部署。例如，部署人员负责将由 bean 提供者或应用组装者定义的安全角色映射到操作环境中存在的用户组和账户。

部署人员使用 EJB 容器提供者提供的工具来执行部署任务。部署过程通常有两个步骤：

- *部署人员首先生成可以使容器能在运行时管理企业 bean 的附加的类和接口。这些类是容器特有的。*
- *部署人员将企业 bean 和附加的类与接口真正的安装到 EJB 容器。*

在某些情况下，一个合格的部署人员可以在部署时客户化企业 bean 的业务逻辑。这样的部署人员通常使用容器提供者的工具来编写相对简单的应用代码，这些代码封装了企业 bean 的业务方法。

2.2.4 EJB 服务器提供商

EJB 服务器提供商在分布式事务管理、分布式对象和其它底层系统级服务领域是一个专家。一个 EJB 服务器提供者通常是一个 OS 供应商、中介供应商或数据库供应商。

当前的 EJB 架构假定 EJB 服务器提供商和 EJB 容器提供者是同一个供应商。因此，它没有为 EJB 服务器提供商定义任何接口要求。

2.2.5 EJB 容器提供者

EJB 容器提供者（简称容器提供者）提供：

- 部署企业 bean 所需的部署工具。
- 对部属的企业 bean 实例提供运行时支持。

从企业 bean 的角度看，容器是目标操作环境的一部分。容器运行时为部署的企业 bean 提供事务和安全管理、远程客户端的网络分布式、资源的可伸缩管理和其它通常作为可管理服务器平台需要的服务。

有 EJB 架构定义的“EJB 容器提供者的责任”指的是 EJB 容器和服务器的实现要求。由于 EJB 规范没有定义 EJB 容器和服务器之间的接口，所以如何分割 EJB 容器和服务器之间要求的功能实现留给供应商来考虑。

容器提供者的专业知识是编写系统级的代码，可能结合一些应用域专业知识。容器提供者关心的是开发一个可伸缩的安全的可使用事务的容器，它集成了 EJB 服务器。容器提供者通过在企业 bean 和容器之间提供一层简单标准的 API，将企业 bean 和后台的 EJB 服务器隔离开来。这些 API 是企业 bean 组件协议。

容器提供者通常对安装的企业 bean 组件提供版本化支持。例如，容器提供者可以允许企业 bean 的 class 在现存的客户端继续有效或保留现存的企业 bean 对象的情况下进行升级。

容器提供者通常为系统管理员提供运行时监控和管理容器及其运行在容器内的 bean 的工具。

2.2.6 持久化提供商

持久化提供商擅长于 O/R 映射、查询处理和缓存。持久化提供商注意开发可扩展的具备事务的运行环境。

持久化提供商提供 O/R 映射所需的工具和支持管理持久化实体和与数据库映射的运行环境。

持久化提供商将持久化实体与底层的持久化细节隔离开来，并在持久化实体和 O/R 运行时之间提供了标准的 API。

持久化提供商可以和 EJB 容器供应商是同一个供应商，也可以是一个第三方的供应商，它提供了一个如【2】所述的可插入的持久化环境。

2.2.7 系统管理员

系统管理员负责配置和管理企业的计算和网络基础设施，包括 EJB 服务器和容器。系统管理员也负责监视部署的企业 bean 应用的运行时正常的。

2.3企业 bean

EJB 是一个基于组件面向事务的企业应用架构。

2.3.1 企业 bean 的特性

企业 bean 的关键特性如下：

- 企业 bean 通常包含操作企业数据的业务逻辑。
- 企业 bean 的实例运行时由容器管理。
- 企业 bean 可以在部署时通过编辑环境条目来进行客户化。
- 不同的服务信息，例如事务和安全属性，都可以和企业 bean 的业务逻辑以元数据注释的形式放在一起，或分别在 XML 部署文件中指定。服务信息可以在应用组装和部署期间被工具抽取出来和管理。
- 客户端的调用由容器作为媒介。
- 如果企业 bean 只使用由 EJB 规范定义的服务，那么企业 bean 可以被部署到任何的 EJB 容器。专业的容器可以提供 EJB 规范之外的服务。依赖这些服务的企业 bean 可能只能部署在这个支持这些服务的容器上。
- 企业 bean 可以被包含在组装好的应用中，而不要求改变源码或重新编译它。
- Bean 提供者定义企业 bean 的客户端视图。Bean 提供者可以手工定义客户端视图或通过应用部署工具来自动生成。客户端视图不受 bean 部署在的容器和服务器的影响。这确保了 bean 和它的客户端都能被部署到多个运行环境中而不需要改变或重新编译。

2.3.2 可伸缩模型

企业 bean 架构是可伸缩的，完全能够实现：

- 一个代表无状态服务的对象。
- 一个代表无状态服务的对象，它实现了 web 服务终端。
- 一个代表无状态服务的对象，它是异步调用的，由到达的消息驱动。
- 一个代表与特定客户端的会话的对象。这个会话对象在多个客户端调用方法间自动维护他们的会话状态。
- 一个代表细粒度持久化对象的实体对象。

可远程访问的企业 bean 主要用于相对粗粒度的业务对象（例如，购物卡，股票报价服务）。细粒度对象（例如，员工记录，购物订单上的一行明细）应当作为轻量级的持久化实体，如【2】所述，不应当作为远程可访问组件。

尽管 EJB 架构定义的状态管理协议是简单的，但他在管理 bean 状态方面为企业 bean 开发者提供了更多的方便。

2.4会话，实体和消息驱动对象

EJB 架构定义了下述的企业 bean 对象：

- 会话对象
- 消息驱动对象
- 实体对象

2.4.1 会话对象

会话对象通常都有下述特征：

- *为单个客户端执行。*
- *可以有事务。*
- *更新后台数据库的共享数据。*
- *不直接代表数据库中的共享数据，尽管它可以获取和更新这些数据。*
- *生命是相对短暂的。*

- *当 EJB 容器宕机时被清除。客户端必须重新定位一个新的会话 bean 来继续计算。*

EJB 容器通常都会提供一个可扩展的运行环境来同时执行多个会话对象。

EJB 规范定义有状态和无状态会话 bean。他们的 API 有些差别。

2.4.2 消息驱动对象

消息驱动对象通常有以下特征：

- *当收到一个客户端消息时执行。*
- *是异步的被调用。*
- *可以有事务。*
- *可以更新后台数据库中的共享数据。*
- *不直接代表数据库中的共享数据，尽管它可以获取和更新这些数据。*
- *生命是相对短暂的。*
- *是无状态的。*
- *当 EJB 容器宕机时被移除。容器必须重新定位一个新的消息驱动对象来继续计算。*

EJB 容器通常都会提供一个可扩展的运行环境来同时执行多个消息驱动对象。

2.4.3 实体对象

实体对象通常都会有以下特征：

- *是域模型的一部分，提供了一个数据库中数据的对象视图。*
- *生命可能是比较长的（和数据库中的数据一样长）。*
- *实体和它的主键在 EJB 容器宕机时依然存在。如果实体的状态在容器宕机时正在被更新，那么当实体再次被取出时，实体的状态被恢复到最后一次提交的事务的状态。*

EJB 容器和服务器通常都会提供一个可扩展的运行环境来同时处理多个活

动的实体对象。

2.5与 CORBA 协议的标准映射

为了帮助来自不同供应商的 EJB 环境进行交互，EJB 规范要求对于来自 JavaEE 客户端的远程调用，容器实现要支持基于 CORBA/IIOP 的交互协议。容器实现可以支持其他的远程调用协议。

第 15 章总结了支持分布式交互的要求。

2.6与 Web 服务协议的映射

为了支持与 web 服务的交互，EJB 规范要求容器实现要支持基于 XML 的 web 服务调用，这些调用使用 WSDL 和 SOAP 或基于 HTTP 的普通 XML，它遵循 JAX-WS【32】，JAX-RPC【25】，用于 Java EE 的 Web 服务【31】和用于 Java 平台规范的 Web 服务元数据。

3 会话 bean 的客户端视图

本章描述会话 bean 的客户端视图。会话 bean 本身实现业务逻辑。Bean 容器提供远程访问、安全、并发、事务等等功能。

当容器实现的类提供了会话 bean 的客户端视图时，容器本身对客户端是透明的。

3.1概述

对客户端来说，会话对象就是非持久化对象，它实现了在服务器上运行的业务逻辑。一种方式可以把会话 bean 看作是客户端程序的在服务器上的逻辑扩展。会话对象不在多个客户端间共享。

客户端从来不直接防伪会话 bean 的实例。客户端通过会话 bean 的客户端视图接口来访问会话 bean。

根据 bean 提供的接口和客户端使用的接口，会话 bean 的客户端可以是本地

客户端、远程客户端或 web 服务客户端。

会话 bean 的远程客户端可能是部署在同一或不同的容器中的另一个企业 bean；或者可以是一个任意的 Java 程序，例如一个应用、applet 或 servlet。会话 bean 的客户端视图也可以被影射到非 java 客户端环境，例如用非 Java 语言写的 CORBA 客户端。

由会话 bean 远程客户端使用的接口由容器实现成为远程业务接口（或者远程 EJBObject 接口），且会话 bean 的远程客户端视图是位置独立的。与会话 bean 对象运行在同一个 JVM 的客户端和运行在不同 JVM 的客户端使用相同的 API。

术语提示:*本规范使用的术语"远程业务接口"指的是支持远程访问的EJB3.0 会话bean 的业务接口。术语"远程接口"指的是EJB2.1 客户端视图的远程组件接口。属于"本地业务接口"指的是支持本地访问的EJB3.0 会话bean 的本地业务接口。术语"本地接口"指的是EJB2.1 客户端视图的本地组件接口。*

使用会话 bean 本地业务接口或本地接口导致本地客户端和会话的排列组合。企业 bean 的本地客户端必须和 bean 布置在同一个容器内。本地客户端视图是位置无关的。

无状态会话 bean 的客户端可以是 web 服务客户端。只有无状态会话 bean 可以提供 web 服务客户端视图。Web 服务客户端使用企业 bean 的 web 服务客户端视图，正如 WSDL 文档中描述的一样。Bean 的客户端视图 web 服务终端依据 JAX-WS 终端【32】或 JAX-RPC 终端接口【25】产生。Web 服务客户端在 3.2.4 和 3.5 章节讨论。

当可以为一个会话 bean 提供多个客户端视图时，通常只提供一个。

在 3.2 章节"Local、Remote 和 Web 服务客户端视图"中将会描述决定会话 bean 的客户端视图时需要仔细考虑的问题。

从会话 bean 的创建到销毁，它都生活在容器中。容器提供安全、并发、事务、与辅助存储交换和其它服务，这些服务对客户端是透明的。

多个企业 bean 可以被安装在同一个容器内。容器允许会话 bean 的客户端通过依赖注入或 JNDI 获取业务接口和/或 home 接口。

会话 bean 的客户端视图不依赖会话 bean 和容器的实现。

3.2Local、Remote 和 Web 服务客户端视图

本节描述了 Bean 提供者决定企业 bean 的客户端视图时应当考虑的问题。

3.2.1 Remote 客户端

在 EJB3.0 中，远程客户端通过 bean 的远程业务接口访问会话 bean。对于有 EJB2.1 和早期 API 写的会话 bean 客户端和组件，远程客户端通过 bean 的远程 home 和远程组件接口来访问会话 bean。

兼容性提示:EJB2.1 和早期API 要求远程客户端通过会话bean 的远程home 和remote 组件接口来访问会话bean。这些接口仍然可以在EJB3.0 中使用，它们在章节3.6 中描述。

企业 bean 的远程客户端是位置无关的。与 bean 实例运行在同一个 JVM 的客户端和运行不同 JVM 的客户端使用相同的 API 来访问 bean。

远程业务接口方法的参数和返回值是按值传递的。

3.2.2 Local 客户端

会话 bean 可以有本地客户端。本地客户端是和提供本地客户端视图的会话 bean 位于同一个 JVM 的客户端，它可以和 bean 紧紧耦合在一起。会话 bean 的本地客户端可以是另一个企业 bean 或 web 组件。

通过本地客户端视图访问企业 bean 要求本地客户端和提供本地客户端视图的企业 bean 位于同一个 JVM 中。因此本地客户端视图没有象远程客户端视图一样提供位置透明。

在 EJB3.0 中，本地客户端通过 bean 的本地业务接口访问会话 bean。对于由 EJB2.1 或早期 API 写的会话 bean 或实体 bean 客户端以及组件，本地客户端通过 bean 的本地 home 和本地组件接口来访问企业 bean。实现本地业务接口的容器对象是本地 Java 对象。

兼容性提示：EJB2.1 和早期的 API 要求本地客户端通过会话 bean 的本地 home 和本地组件接口来访问会话bean。这些接口在EJB3.0 中仍然保留，在3.6

章节中描述。

本地业务接口方法的参数和返回值是“按引用”传递的（注：更准确的说，引用时在 JVM 中按值传递：原始类型的参数变量容纳原始类型的值；引用类型的参数变量容纳一个对对象的引用。参见【28】）。因此，提供本地客户端视图的企业 bean 应当假定作为参数或返回结果的 Java 对象的状态潜在地被调用者和被调用者共享。

Bean 提供者必须知道通过本地接口传递的对象的潜在共享。特殊情况下，Bean 提供者必须注意一个企业 bean 的状态不要赋给另一个 bean 的状态。通常情况下，跨本地接口传递的引用不能在调用链的外部使用，且不能作为另一个企业 bean 状态的一部分被存储。Bean 提供者也必须小心的决定哪些对象跨本地接口被传递。这些注意事项通常应用在在一个事务或安全上下文中发生改变的情况下。

3.2.3 Local 或 Remote 客户端视图的选择

在决定为企业 bean 使用本地还是远程访问时，应当考虑以下的注意事项：

- 远程编程模型为部署环境中的组件的分布提供位置无关性和灵活性。它在客户端和 bean 之间提供了松耦合。
- 远程调用时按值传递的。复制语义在调用者和被调用者之间提供了一个隔离层，并防止数据的无意修改。客户端和 bean 编程可假定为参数复制。
- 在远程调用中作为参数传递的对象必须是可序列化的。
- 当使用 EJB2.1 和早期远程 home 和 remote 组件接口时，远程类型的转换要求使用 javax.rmi.PortableRemoteObject.narrow 而不是使用 java 语言的转换。
- 由于远程编程模型资源消耗大，它通常用于相对大粒度的组件访问。
- 本地接口是按引用传递。客户端和 bean 可以依赖于按参数传递语义。例如，客户端想传递给 bean 一个大文档来修改，并且 bean 进一步传递它。可以在本地编程模型共享状态。在另一个方面，当 bean 希望返回给客

户端一个数据结构但又不想让客户端修改它，那么 bean 在返回之前显式地复制这个数据结构，但在远程编程模型中 bean 不需要复制数据结构，因为系统将做这件事。

- 因为本地调用时按引用传递的，本地客户端和企业 bean 提供的本地客户端视图是位置相关的。
- 本地编程模型的位置相关意味着企业 bean 不能和它的客户端部署到不同的节点——这样约束了组件的分布。
- 因为本地编程模型提供了更轻量地访问组件，因此可以更好的支持细粒度的组件访问。

注意，尽管远程客户端和企业 bean 的位置可以让容器减少通过远程业务接口或远程组件接口调用的负载，但是这样的调用仍然没有使用本地接口有效率，因为任何基于位置的优化必须是透明地来做。

在本地和远程模型间作选择是Bean提供者在开发企业bean时要做的设计决定。

但是也可以为企业 bean 既提供远程客户端视图又提供本地客户端视图，通常是只提供一种。

3.2.4 Web 服务客户端

无状态会话 bean 可以有 web 服务客户端。

Web 服务客户端通过 web 服务客户端视图访问无状态会话 bean。Web 服务客户端视图通过用于 bean 实现的 web 服务的 WSDL 文档描述。WSDL 是描述作为一系列基于消息的终端的 web 服务的 XML。服务的抽象描述被绑定到一个基于协议（SOAP【27】）和底层传输协议（HTTP 或 HTTPS）的 XML。（参见【25】，【26】，【30】，【31】，【32】）。

无状态会话 bean 的 web 服务方法提供了 web 服务客户端视图的基础，这些视图是通过 WSDL 暴露的。参见【30】和【25】了解 java 语言元数据注释如何被用于指定无状态会话 bean 的 web 服务客户端视图。

兼容性提示：当他或她想将 bean 的功能通过 WSDL 暴露成 web 服务中的时，

EJB2.1 要求 bean 提供者要为无状态会话 bean 定义 web 服务终端接口。在 EJB3.0 中删除了这种要求。参见【30】。

Bean 的 web 服务客户端视图可以先通过 WSDL 文档定义，然后映射到遵循这个文档的 web 服务终端，或者将一个现存的 bean 适配成提供 web 服务客户端视图。参考资料【31】描述了大量的设计时场景，这些场景可以用于 EJB web 服务终端。

企业 bean 的 web 服务客户端视图是位置透明并是远程的。

Web 服务客户端可以是 Java 客户端和/或非 Java 语言的客户端。是 Java 客户端的 web 服务通过 JAX-WS 或 JAX-RPC 客户端 API 访问 web 服务。访问 web 服务客户端通过 SOAP1.1，SOAP1.2 或基于 HTTP(S)的 XML 来访问。

3.3EJB 容器

EJB 容器（简称容器）是作为企业 bean 的“容器”的系统。多个企业 bean 可以被部署在同一个容器内。容器负责部署在它内的企业 bean 的业务接口和/或 home 接口可由客户端通过依赖注入和/或 JNDI 查找获得。

3.4用 EJB3.0 简化 API 书写的会话 bean 客户端视图

用 EJB3.0 API 书写的会话 bean 的 EJB3.0 本地或远程客户端通过业务接口访问会话 bean。EJB3.0 会话 bean 的业务接口是一个普通的 Java 接口，而不管它为 bean 提供的是本地还是远程访问。特殊情况下，EJB3.0 会话 bean 业务接口不是早期 EJB 规范要求的接口类型（例如，EJBObject 或 EJBLocalObject 接口）。

3.4.1 获取会话 bean 的业务接口

客户端可以通过依赖注入或 JNDI 查找来获得会话 bean 的业务接口。

例如，CartBean 会话 bean 的业务接口 Cart 可以用依赖注入获得，如下：

```
@EJB Cart cart;
```

Cart 业务接口也可以按下面的代码片段使用 JNDI 来查找，使用由

EJBContext 接口的 lookup 方法。在这个例子中，对客户端 bean 的 SessionContext 对象的引用通过依赖注入获得。

```
@Resource SessionContext ctx;
…
Cart cart = (Cart)ctx.lookup(“cart”);
```

在这两种情况中，用于获取 Cart 业务接口的语法是不依赖于业务接口是否是本地还是远程接口。在远程访问的情况下，被引用企业 bean 的真正位置和 EJB 容器通常对使用远程业务接口的客户端都是透明的。

3.4.2 会话 bean 的业务接口

会话 bean 的业务接口是普通的 Java 接口。它包含了会话 bean 的业务方法。

一个对会话 bean 业务接口的音乐可以作为业务接口方法的参数或返回值被传递。如果引用是会话 bean 的本地业务接口，那么引用只可以被作为本地业务接口方法的参数或返回值被传递。

有状态会话 bean 的业务接口通常包含一个初始化会话对象状态和一个表明客户端已经使用完会话对象且它可以被销毁的方法。参见第 4 章“会话 bean 组件协议”。

引用一个不存在的会话对象是无效的。如果有状态会话 bean 已经被删除，那么调用它的业务接口会抛出 javax.ejb.NoSuchEJBException。（注：这可以不应用到无状态会话 bean 上；参见节 4.5）

容器提供了会话 bean 业务接口的实现，这样当客户端调用业务接口实例上的方法时，就可以调用会话 bean 实例上的业务方法以及任何拦截器方法。

容器可以让 EJB3.0 客户端通过依赖注入或 JNDI 查找获得会话 bean 的业务接口。节 16.5 进一步描述了客户端如何获取 EJB 业务接口的引用。

3.4.3 会话对象生命周期的客户端视图

从客户端视图的角度看，一旦客户端获得业务接口的引用，那么会话对象就

存在了——不管是通过依赖注入还是通过 JNDI 查找。

客户端然后能够使用会话对象业务接口的引用来调用接口上的业务方法，和/或作为业务接口方法的参数或返回值传递它。（注：注意，EJB3.0 会话 bean 业务接口不是 EJBObject。通过 bean 的远程组件接口传递一个引用到一个远程业务接口是无效的）。

客户端通过调用业务接口的一个方法可以删除一个有状态会话 bean，这个方法指派为 Remove 方法。

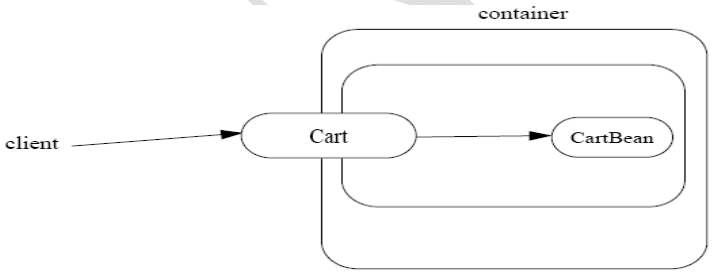
无状态会话 bean 的生命周期不要求客户端来删除。由容器来执行无状态会话 bean 的删除，对客户端是透明的。

会话 bean 生命周期的协议在第 4 章“会话 bean 组件协议”中描述。

3.4.4 获得和使用会话 bean 的例子

会话 bean 运行时对象的例子通过下图来解释：

图 1 会话 bean 例子中的对象



客户端通过依赖注入或 JNDI 查找获取一个 Cart 会话 bean 对象的引用，这个会话对象提供了购物服务。客户端然后使用这个会话对象来用项目填充购物车，并购买它的内容。Cart 是有状态会话。

在这个例子中，客户端通过依赖注入获得一个 Cart 业务接口的引用。客户端然后使用这个业务接口来初始化会话对象并向它增加一些项目。方法 startShopping 是一个业务方法，用于初始化会话对象。

```
@EJB Cart cart;
…
cart.startShopping();
```

```
cart.addItem(66);
cart.addItem(22);
```

最后，客户端购买了购物车内的所有内容，并完成购物活动。（注：用有状态会话 bean 来分派一个删除有状态会话的方法（这样可以释放会话 bean 使用的资源）是应用逻辑的一部分。这个例子假定 finishShopping 就是这样的 Remove 方法。参见节 4.4 有进一步的讨论）。

```
cart.purchase();
cart.finishShopping();
```

3.4.5 会话对象标识

客户端可以使用 Object.equals 和 Object.hashCode 两个方法来区分会话 bean 业务接口的引用。

3.4.5.1 有状态会话 bean

有状态会话对象有一个唯一的标识，它由容器在创建对象时设置。有状态会话 bean 业务接口的客户端可以通过使用 equals 方法来判断两个业务接口引用是否指向同一个会话对象。

例如：

```
@EJB Cart cart1;
@EJB Cart cart2;
…
if (cart1.equals(cart2)) {//这个检查必须返回 true
    …
}
if (cart1.equals(cart2)) {//这个检查必须返回 false
    …
}
```

所有指向同一个有状态会话 bean 实例的同一个业务接口的有状态会话 bean 引用都是相等的。执行不同接口类型或不同会话 bean 实例的有状态会话 bean 引用不会有相同的标识。

3.4.5.2 无状态会话 bean

所有执行同一个无状态会话 bean 的同一个接口类型的业务对象引用有相同的对象标识，这个标识由容器设置。

例如：

```
@EJB Cart cart1；
@EJB Cart cart2;
…
if (cart1.equals(cart2)) {//这个检查必须返回 true
    …
}
if (cart1.equals(cart2)) {//这个检查也必须返回 true
    …
}
```

当 equals 方法用于比较同一个会话 bean 的同一个业务接口类型的引用时，它总是返回 true。不同会话 bean 的不同业务接口类型的会话 bean 引用是不想等的。

3.5无状态会话 bean 的 Web 服务客户端视图

3.6用 EJB2.1 客户端视图 API 书写的远程核本地客户端视图

略。

4 会话 bean 组件规约

本章定义了会话 bean 和容器之间的协议。它定义了会话 bean 实例的生命周期。

本章定义了会话 bean 状态管理的开发者视图和容器管理会话 bean 状态的责任。

4.1概述

会话 bean 实例是会话 bean 类的一个实例。它容纳了会话对象的状态。

会话 bean 实例是创建它的客户端的扩展：

- 在有状态会话 bean 的情况下，它的字段包含了一个会话状态，这个状态代表了会话对象的客户端。这个状态描述了由特定客户端/会话对象对儿代表的会话。
- 它通常代表客户端读取和更新数据。
- 在有状态会话 bean 的情况下，它的生存时间由客户端来控制。

容器也可以在开发者指定的超时时间到达时或当服务器失败时终止会话 bean 实例。因此，如果客户端失去了一个它正在使用的会话对象，那么它可以重建一个新的会话对象。

通常情况下，会话对象的会话状态不写到数据库中。会话 bean 开发者只是将它保存到会话 bean 实例的字段中，并假定这个字段的值会保留在实例的生存期间。开发者可以使用一个扩展的持久化上下文来存储有状态会话 bean 的持久化会话状态。参见文档“Java 持久化 API”【2】。

没有使用 Java 持久化 API 的会话 bean 必须显式管理缓存的数据库数据。会话 bean 实例必须在事务结束前将缓存的数据库更新写到数据库中，并且它必须在下一个事物开始时刷新它缓存的数据库数据的拷贝，因为它的数据可能已经过时。会话 bean 也必须在新的事务上下文被使用之前刷新 java.sql Statement 对象。Java 持久化 API 为会话 bean 提供了自动管理数据库数据的功能，包括在事务提交时自动将缓存的数据库更新冲刷到数据库中。参见【2】。

容器管理会话 bean 实例的生命周期。当需要 bean 的动作时，它通知 bean 的实例，并且它提供了大量的服务来保证会话 bean 实现是可伸展的以及能支持大量的客户端。

会话 bean 可以是：

- 无状态的——会话 bean 实例在方法间不保持会话状态；所有的实例可以用于任何客户端。
- 有状态的——会话 bean 实例必须在方法和事务之间保留会话状态。

4.2有状态会话 bean 的会话状态

有状态会话对象的会话状态定义为会话 bean 实例的字段值、它相关的拦截器以及它们的实例字段值、和实例字段引用的对象。

为了有效管理工作集的大小，会话 bean 容器可能需要临时将空闲有状态会话 bean 的状态转换成某种二级存储的形式。从工作级到二级存储的转换称为实例钝化。反向转换成为激活。

在更先进的情况下，会话对象的会话状态可以包含打开的资源，例如打开的 socket 和打开的数据库光标。在会话 bean 实例被钝化时，容器不能保留这些打开的资源。有状态会话 bean 的开发者必须在 PrePassiveate 和 PostActivate 方法内关闭和打开资源（注：注意，这个要求不适用在 EntityManager 和 EntityManagerFactory 对象上）。

当实例不在事务中时，容器可以只钝化有状态会话 bean 实例。

容器不可以钝化有扩展持久化上下文的有状态会话 bean，除非满足下面的条件（注：因为不满足这些要求，容器不允许销毁有状态会话 bean 实例）：

- 在持久化上下文中的实体是可序列化的。
- EntityManager 是可序列化的。

无状态会话 bean 从来不会钝化。

4.2.1 实例钝化和会话状态

要求 bean 提供者保证 PrePassivate 方法内的实例字段和实例相关拦截器字段都是可被容器可序列化的。在 PrePassivate 完成后赋给实例和实例拦截器的非临时字段的对象必须是下来对象之一：

- 一个可序列对象（注：注意，Java 序列化协议动态地决定对象是否可序列化。这意味着可能序列化一个声明的字段类型是不可序列化的但它的子类是可序列化的对象）。
- 一个 null。
- 一个对企业 bean 业务接口的引用。
- 一个对企业 bean 远程接口的引用，尽管 stub 类不是可序列化的。
- 一个对企业 bean 远程 home 接口的引用，尽管 stub 类不是可序列化的。
- 一个对实体 bean 本地接口的引用，尽管它是不可序列化的。
- 一个对实体 bean 本地 home 接口的引用，尽管它是不可序列化的。
- 一个对环境命名上下文的引用（也就是 java:comp/env JNDI 上下文）或它的任何子上下文。
- 一个对 UserTransaction 接口的引用。
- 一个对资源管理器连接工厂的引用。
- 一个对容器管理的 EntityManager 对象的引用，尽管它是不可序列化的。
- 一个对 EntityManagerFactory 对象的引用，它通过依赖注入或 JNDI 查找得到，尽管它是不可序列化的。
- 一个对 javax.ejb.Timer 对象的引用。
- 一个不是直接可序列化的对象，但通过以下方式变成了可序列化的：在对象序列化期间被可序列化对象替换成了企业 bean 业务接口、企业 bean home 和组件接口、SessionContext 对象、java:comp/env JNDI 上下文和它的子上下文、UserTransaction 接口、EntityManager 和 / 或 EntityManagerFactory 的引用。

这意味着，例如，Bean 提供者必须在 PrePassivate 方法内关闭所有的 JDBC

连接，并将存储连接的实例字段设为null。

最后一条覆盖了如在会话状态中存储组件接口集合的情况。

Bean 提供者必须假定临时自动的内容可以在 PrePassivate 和 PostActivate 之间被丢失。因此，Bean 提供者不应当在临时字段中存储下列对象的引用：SessionContext 对象、环境 JNDI 命名上下文和它的子上下文、业务接口、home 和组件接口、EntityManager 接口、EntityManagerFactory 接口、UserTransaction 接口。

对临时字段使用的限制保证容器在钝化和激活期间可以使用 Java 序列化。

下面是对容器的要求。

容器必须能够正确地保存和恢复存储在实例状态中的对业务接口和企业 bean 的 home 和组件接口的引用，即使实现对象引用的类不是可序列化的。

容器可以使用，例如，对象代替技术（它是 java.io.ObjectOutputStream 和 java.io.ObjectInputStream 协议的一部分）来输出和输入 home 和组件的引用。

如果会话 bean 实例在会话状态中存储了对 javax.ejb.SessionContext 接口的引用，那么容器必须能够跨实例钝化来保存和恢复这个引用。容器能够在激活时用不同的但功能上与原始 SessionContext 对象等价的 SessionContext 对象来替换原始的 SessionContext 对象。

如果会话 ban 实例在会话状态中存储了对 java:comp/env JNDI 上下文或它的子上下文的引用，那么容器必须能够跨实例钝化来存储和恢复这个引用。容器可以在激活时使用不同的但与原始对象功能上等价的对象来代替原始对象。

如果会话 bean 实例在会话状态中存储了对 EntityManager 或 EntityManagerFactory 的引用，那么容器必须能够跨实例钝化来存储和恢复这个引用。

如果实例在 PrePassivate 后不满足序列化的要求，容器可以销毁一个会话 bean 实例。

但不要求容器使用 Java 的序列化协议来存储钝化的会话实例的状态，但必须达到相同的效果。一个例外是，在激活时不要求容器重设 transient 字段的值（注：这样可以让容器通过其他的技术取出实例的状态，而不使用 Java 的序列

化协议。例如，容器的 JVM 实现可以使用一块内存来保存实例变量，然后容器将整个内存块保存到内存而不是在实例上执行 Java 的序列化操作）。通常不鼓励将会话 bean 的字段声明为 transient 的。

4.2.2 在会话状态上事务回滚的影响

会话对象的会话状态时非事务的。如果对象参与的事务回滚时，它不会自动回滚到它的初始状态。

如果回滚可能引起会话对象的会话状态和后台数据库状态的不一致，那么 bean 开发者（或者由开发者使用的应用部署工具）必须使用 afterCompleted 来手工重设状态。

4.3会话 bean 实例和容器间的协议

容器本身对会话 bean 实例没有真正的服务要求。容器调用 bean 实例以提供容器的服务和转发容器产生的通知。

4.3.1 要求的会话 bean 元数据

会话 bean 必须注释或在部署文件中声明为无状态或有状态会话 bean。无状态会话 bean 必须注释为 Stateless 获在部署文件中声明为无状态会话 bean。有状态会话bean必须注释为Stateful获在部署文件中声明为有状态会话bean。Stateless 和 Stateful 注释符是组件定义的注释符，它们应用到 bean 类上。

4.3.2 依赖注入

会话 bean 可以使用依赖注入机制来获取对资源或环境中其他对象的引用（参见第 16 章“企业 bean 环境”）。如果会话 bean 使用依赖注入，那么容器在实例被创建后在实例方法被调用前注入这些引用。如果声明依赖 SessionContext，或如果 bean 类实现了可选的 SessionBean 接口（参见 4.3.5 节），SessionContext 也会同时被注入。如果依赖注入失败，那么 bean 实例被丢弃。

当使用 EJB3.0 API 时，bean 类可以通过依赖注入获取 SessionContext 接口而不必实现 SessionBean 接口。在这种情况下，注释符 Resource（或部署文件中的元素 resource-env-ref）用于声明 bean 对 SessionContext 的依赖。参见第 16 章“企业 bean 的环境”。

4.3.3 SessionContext 接口

如果 bean 指定依赖 SessionContext 接口（或 bean 类实现了 SessionBean 接口），那么容器必须为 bean 提高 SessionContext。这让会话 bean 实例访问由容器维护的上下文。SessionContext 接口有如下方法：

- getCallserPrincipal 方法返回标识调用者的 java.security.Principal。
- isCallerInRole 方法测试会话 bean 实例的调用者是否具有特定的角色。
- setRollbackOnly 方法让实例可以标记当前事务的唯一结果就是回滚。只有使用容器管理事务分割的会话 bean 实例才可以使用这个方法。
- getRollbackOnly 方法放实例可以检查当前事务是否已经被标记为回滚。只有使用容器管理事务分割的会话 bean 实例才可以使用这个方法。
- getUserTransaction 方法返回 javax.transaction.UserTransaction 接口。实例可以使用这个接口来分割事务和获取事务状态。只有使用容器管理事务分割的会话 bean 实例才可以使用这个方法。
- getTimerService 方法返回 javax.ejb.TimerService 接口。只有无状态会话 bean 才可以使用这个方法，有状态会话 bean 不能是有时效的对象。
- getMessageContext 方法返回实现了 JAX-RPC web 服务终端的无状态会话 bean 的 javax.xml.rpc.handler.MessageContext 接口。只有具有 web 服务终端接口的无状态会话 bean 才可以使用这个方法。
- getBusinessObject（类 businessInteface）方法返回会话 bean 的业务接口。只有具有 EJB3.0 业务接口的会话 bean 才能调用这个方法。
- getInvokedBusinessInterface 方法返回会话 bean 业务接口，通过这个接口调用 bean。
- getEJBObject 方法返回会话 bean 的远程接口。只有具有远程 EJBObject

接口的会话 bean 才能调用这个方法。

- getEJBHome 方法返回会话 bean 的远程 home 接口。只有具有远程 home 接口的会话 bean 才能调用这个方法。
- getEJBLocalObject 方法返回会话 bean 的本地接口。只有具有本地 EJBLocalObject 接口的会话 bean 才能调用这个方法。
- getEJBLocalHome 方法返回会话 bean 的本地 home 接口。只有具有本地 home 接口的会话 bean 才能调用这个方法。
- lookup 方法使得会话 bean 可以查找在 JNDI 中的环境条目。

4.3.4 会话 bean 的生命周期回调拦截器方法

下面的生命周期事件回调是用于支持会话 bean 的。生命周期回调拦截器方法可以直接被定义在 bean 类上或定义成单独的拦截器类。参见第 4.6.3 章节和第 12 章。

- PostConstruct
- PreDestroy
- PostActivate
- PrePassivate

PostConstrucat 在第一个业务方法调用之前被调用。这发生在所有的依赖注入已经被容器执行完之后。

PostConstruct 生命周期拦截器方法在未指定的事务和安全上下文中执行。

PreDestroy 回调通知实例处于被容器删除的过程中。在 PreDestroy 方法中，实例通常要释放它所占有的资源。

PreDestroy 在未指定的事务和安全上下文中执行。

PrePassivate 回调通知容器将要钝化实例。

PostActivate 通知实例已经被激活。因为容器在它钝化时自动维护有状态会话 bean 的会话状态，所以这些通知对大多数的会话 bean 都是不需要的。它的目的是让有状态会话 bean 维护那些在钝化之前被关闭的资源在实例激活时需要被重新打开。

PrePassivate 和 PostActivate 生命周期回调拦截器方法在为指定的事务和安全上下文中执行。

4.3.5 可选的 SessionBean 接口

会话 bean 实例类不要求实现 SessionBean 接口或 Serializable 接口。同样也不要求拦截器类实现 Serializable 接口。

兼容性提示：在 EJB 规范的早期版本中，要求会话 bean 实现 SessionBean 接口。在 EJB3.0 中，以前通过 SessionBean 接口提供的功能可以通过有选择的使用依赖注入（SessionContext 的依赖注入）和可选择的生命周期拦截器方法来得到。

SessionBean 接口定义了四个方法：setSessionContext，ejbRemove，ejbPassivate 和 ejbActivate。

setSessionContext 方法由容器调用来建立会话 bean 实例和容器维护的上下文之间的关联。通常会话 bean 实例保留会话上下文作为它状态的一部分。

ejbRemove 通知实例正在被容器删除。在 ejbRemove 方法中，实例通常释放和在 ejbPassivate 方法相同的资源。

在 EJB3.0 API 中，bean 类可以可选的定义 PreDestroy 生命周期回调拦截器方法来通知容器要删除 bean 实例。

ejbPassivate 通知容器将要钝化实例。ejbActivate 通知实例正要被重新激活。它们都是为了让有状态会话 bean 来维护那些在钝化之前需要关闭在激活时需要重新打开的资源。ejbPassivate 和 ejbActivate 方法只能被有状态会话 bean 实例调用。

在 EJB3.0 API 中，bean 类可以可选的定义 PrePassivate 和/或 PostActivate 生命周期回调拦截器方法来通知 bean 实例的钝化/激活。

规范要求 SessionBean 接口的 ejbRemove，ejbActivate 和 ejbPassviate 方法，以及无状态会话 bean 的 ejbCreate 方法分别被认为是 PreDestory，PostActivate，PrePassivate 和 PostConstruct 生命周期回调拦截器方法。

如果会话 bean 实现了 SessionBean 接口，那么 PreDestroy 注释只能应用到

ejbRemove 方法；PostActivate 注释只能应用到 ejbActivate 方法；PrePassivate 注释只能应用到 ejbPassivate 方法。同样的要求应用到配置描述元数据，这些元数据作为注释符的替代方案。

4.3.6 由无状态会话 bean 使用的 MessageContext 接口

实现了使用 JAX-RPC 协议的 web 服务终端的无状态会话 bean 通过 SessionContext.getMessageContext 方法访问 JAX-RPC MessageContext 接口。MessageContext 接口让无状态会话 bean 实例能够明白用于 web 服务终端的 SOAP 消息，以及由 JAX-RPC SOAP 消息处理器设置的属性（如果有的话）。无状态会话 bean 可以使用 MessageContext 接口来为 JAX-RPC 消息响应处理器设置属性（如果有的话）。

实现了使用 JAX-WS 协议的 web 服务终端的无状态会话 bean 应当使用 JAX-WS WebServiceContext，它可以通过使用 Resource 注释符来注入。WebServiceContext 接口可以让无状态会话 bean 实例明白用于 web 服务终端的 SOAP 消息，以及被 JAX-WS 消息处理器设置的属性（如果有的话）。无状态会话 bean 可以使用 WebServiceContext 接口来为 JAX-WS 消息处理器设置属性（如果有的话）。参见【32】。JAX-WS WebServiceContext 也可以被无状态会话 beanweb 服务终端的拦截器获得。参见第 12.5 章节。

4.3.7 有状态会话 bean 可选的 SessionSynchronization 接口

有状态会话 bean 类可以可选地实现 javax.ejb.SessionSynchronization 接口。这个接口为会话 bean 实例提供了事务同步通知。例如，实例可以使用这些通知来管理缓存在事务中的数据库数据——例如，如果没有使用 JPA。

afterBegin 通知会话 bean 实例新的事务已经开始。容器在事务内的第一个业务方法调用之前调用这个方法（不是必须在事务开始时调用）。afterBegin 被调用时具有事务上下文。实例可以在事务范围内做任何需要的数据库操作。

beforeComplete 通知发生在当会话 bean 实例的客户端以及完成当前事务上

的工作但在提交实例使用的资源管理器之前。此时，实例应当将它缓存的数据库更新全部写出。实例可以通过调用会话上下文的 setRollbackOnly 方法来回滚事务。

afterComplete 通知当前事务已经完成。完成状态为 true 表示事务已经被提交。状态为 false 表示发生了回滚。由于会话 bean 实例的会话状态不是事务性的，所以如果发生回滚可能需要手工设置它的状态。

所有的容器提供者必须支持 SessionSynchronization。只是对于 bean 实现者来说是可选的。如果 bean 类实现了 SessionSynchronization，那么容器必须按规范要求调用 afterBegin，beforeCompletion 和 afterCompletion 通知方法。

只有使用容器管理事务分割的有状态会话 bean 可以实现 SessionSynchronization 接口。无状态会话 bean 不能实现 SessionSynchronization 接口。

对使用 bean 管理事务分割的会话 bean，不需要依赖同步回调，因为 bean 管理提交——bean 知道什么时候事务将要提交，以及也知道事务提交的结果。

4.3.8 无状态会话 bean 的超时回调

如果无状态会话 bean 提供了超时回调方法，那么它就能够向 EJB Timer 服务注册基于时间的事件通知。当 bean 的计时器到期时，容器调用 bean 实例的超时回调方法。参见第 18 章“计时器服务”。有状态会话 bean 不能想 EJB Timer 服务注册，因此它不应当实现超时回调方法。

4.3.9 业务方法代理

会话 bean 的业务方法接口，组件接口或 web 服务终端定义了可由客户端调用的业务方法。

实现了这些接口的容器类由容器工具生成。实现了会话 bean 业务接口的类和实现了会话bean 组件接口的类代理那些与会话bean 类中实现的业务方法相匹配的方法的调用。处理对 web 服务终端请求的类调用与对应于这个 SOAP 请求的

web 服务方法相匹配的无状态会话 bean 的方法。

4.3.10 会话 bean 的创建

容器创建按下面步骤创建会话 bean 的实例。第一，容器调用 bean 类的 newInstance 方法来创建一个会话 bean 的新实例。第二，如果申请了 SessionContext，容器注入 bean 的 SessionContext，并执行在 bean 类上由元数据注释指定的或部署文件指定的其他依赖注入。第三，容器调用 bean 的 PostConstruct 生命周期回调拦截器方法，如果有的话。如果通过 EJB2.1 客户端视图 API 来调用会话 bean，那么应用下面描述的步骤。

4.3.10.1 有状态会话 bean

如果 bean 是有状态会话 bean 且客户端使用一个在会话 bean 的 home 或本地 home 接口定义的 create<METHOD>方法来创建 bean，那么容器调用实例的初始化方法（它的签名匹配客户端调用的 create<METHOD>的签名）传入客户端发送的输入参数。如果 bean 类由 EJB3.0 API 实现，且已经适配了早期的客户端视图，那么初始化方法就是由 Init 注释符或 init-method 配置元素指定的 Init 方法（注：除那些由注释符定义的初始化方法外，也应用任何由 init-method 配置元素定义的初始化方法）。如果 bean 类由 EJB2.1 或早期的 API 实现，那么初始化方法就是 ejbCreate<METHOD>，正如在 4.6.4 章节中描述的一样。

每个有 home 接口的会话 bean 必须至少有一个这样的初始化方法。会话 bean 的初始化方法的数量和签名是每个会话 bean 类特有的。由于有状态会话 bean 代表了客户端和 bean 之间的一个特定的私有的会话，所以它的初始化参数通常包含客户端用于客户化 bean 实例的信息。

4.3.10.2 无状态会话 bean

有 EJB2.1 本地或远程客户端视图的无状态会话 bean 在 home 接口上只有一个 create 方法。在这种情况下，EJB2.1 要求无状态会话 bean 类的 ejbCreate 方法

不能有参数。在 EJB3.0 情况下，不要求无状态会话 bean 有一个 ejbCreate 方法，即使它有 home 接口。EJB3.0 无状态会话 bean 类可以有一个 PostConstruct 方法，如在节 4.3.4 所述。

如果无状态会话 bean 实例有一个 ejbCreate 方法，那么容器将 ejbCreate 方法看作是实例的 PostConstruct 方法，而且在这种情况下，PostConstruct 注释（或部署文件元数据）只能被应用到 bean 的 ejbCreate 方法。

由于无状态会话 bean 实例通常放在池中，因此客户端调用 create 方法的时机和容器调用 PostConstruct/ejbCreate 方法没有任何的关系。

只提供 web 服务客户端视图的无状态会话 bean 没有 create 方法。如果出现 EJB2.1 要求的 ejbCreate 方法，那么它也被看作实例的 PostConstruct 方法，因此在容器需要为服务一个客户端请求而创建新会话 bean 实例时就会调用它。

4.3.11 有状态会话 bean 的删除

由 EJB3.0 API 写的有状态会话 bean 通常有一个或多个删除方法，这些方法由 Remove 注释或 remove-method 部署元素指定（注：除了注释符指定的方法外，也应用任何由 remove-method 部署元素指定的删除方法）。调用删除方法会在删除方法成功完成后删除有状态会话 bean。如果 Remove 注释指定了 retainIfException 的值为 true，且 Remove 方法抛出一个应用异常，那么不会删除实例。可以显式的指定 remove-method 部署元素的子元素 retain-if-exception 来覆盖 retainIfException 由 Remove 注释指定或缺省的值。

4.3.12 会话 bean 的业务方法拦截器方法

会话 bean 支持 AroundInvoke 拦截器方法。这些拦截器方法可以定义在 bean 类和/或定义在拦截器类上，它们应用到 bean 业务接口、组件接口和/或 web 服务终端中业务方法的调用上。

对于实现了 SessionSynchronization 接口的有状态会话 bean，在任何 AroundInvoke 方法调用之前调用 afterBegin，在所有的 AroundInvoke 调用完成后

调用 beforeComopletion。

拦截器在第 12 章“拦截器”中描述。

4.3.13 序列化会话 bean 的方法

容器序列化对每个会话 bean 实例的调用。大多数容器支持同时执行多个会话 bean 实例；但是，每个实例只看到一个序列化的方法调用序列。因此，会话 bean 不必被编写成可重入的（reentrant）。

容器必须序列化所有容器调用的回调方法（即，业务方法拦截器方法，生命周期回调拦截器方法，超时回调方法，beforeCompletion 等等），容器必须和客户端调用的业务方法调用一起序列化这些回调方法。

客户端不允许并发调用有状态会话对象。如果客户端调用的业务方法处于另一个客户端调用的处理过程中时（这个调用来自相同或不同的客户端，但到达有状态会话 bean 类的同一个实例），如果第二个客户端是 ben 业务接口的客户端，那么并发操作可能让第二个客户端收到 javax.ejb.ConcurrentAccessException（注：javax.ejb.ConcurrentAccessException 是 javax.ejb.EJBException 的子类。如果业务接口是扩展了 java.rmi.Remote 的远程业务接口，那么客户端将收到的是 java.rmi.RemoteException 而不是 javax.ejb.ConcurrentAccessException）。如果使用 EJB2.1 客户端视图，那么如果第二个客户端是远程客户端，则容器可以抛出 java.rmi.RemoteException，如果第二个客户端是本地客户端，则容器可以抛出 javax.ejb.EJBException。这个限制不应用到无状态会话 bean，因为容器将每个请求路由到不同的会话 bean 实例。

在某些特定的情况下（例如，为了处理 web 容器的集群），容器可以不用排队或序列化这种并发请求。但是，客户端不能依赖这种行为。

4.3.14 会话 bean 方法的事务上下文

定义在会话 bean 业务接口或组件接口中的方法、web 服务方法或超时回调方法的实现在事务范围内被调用，这个事务范围由在 bean 的元数据注释或部署

文件中指定的事务属性来决定。

会话 bean 的 afterBegin 和 beforeCompletion 方法总是使用和在它们之间执行的业务方法相同的事务上下文被调用。

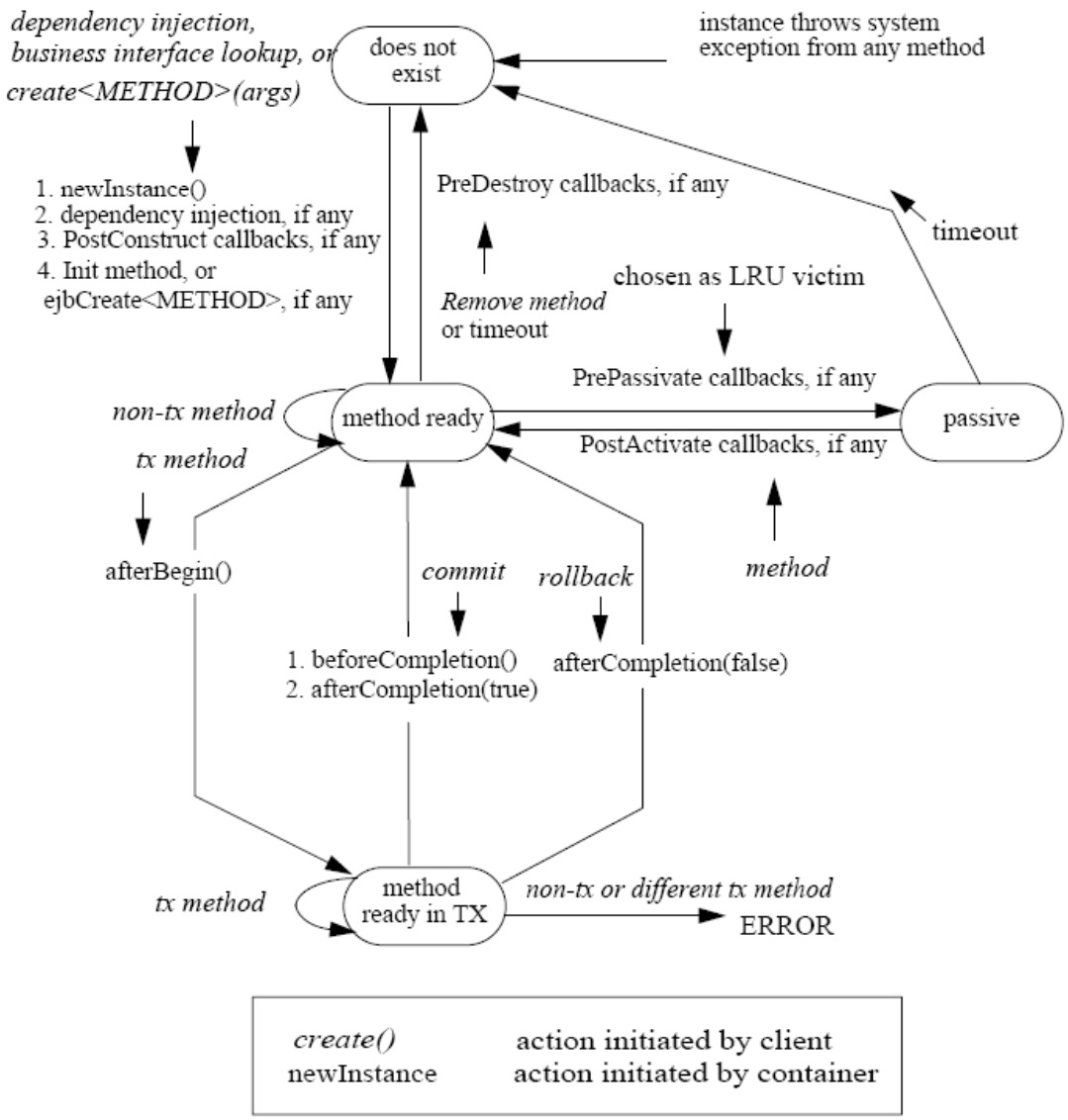
会话 bean 的 newInstance，setSessionContext，其他的依赖注入方法，生命周期回调拦截器方法和 afterCompletion 方法用未指定的事务上下文调用。参考接 13.6.5 了解容器如何在未指定事务上下文中执行方法。

例如，在会话 bean 的 PostConstruct 或 PreDestroy 生命周期回调拦截器方法中执行数据库操作而且假定操作是客户端事务的一部分就是错误的。PostConstruct 和 PreDestroy 方法不被事务属性控制，因为在这些方法内处理回滚将复杂化会话实例的状态图。

4.4有状态会话 bean 的状态图

下图解释了有状态会话 bean 实例的生命周期。

图 5 有状态会话 bean 实例的生命周期



下面的步骤描述了有状态会话 bean 实例的生命周期：

- 会话 bean 实例的生命从客户端通过依赖注入或 JNDI 查找获得有状态会话 bean 实例的引用开始，或当客户端调用会话 bean 的 home 接口中的 create<METHOD>方法开始。这会导致容器调用会话 bean 类的 newInstance 来创建新的会话 bean 实例。接着，如果申请了 SessionContext，则容器注入 bean 的 SessionContext，并执行由注释符或部署文件元素指定的其他依赖注入。容器然后调用 bean 的 PostConstruct 生命周期回调拦截器方法（如果有的话）。最后，如果会话 bean 是为 EJB2.1 客户端视图实现的，那么容器调用实例的 ejbCreate<METHOD>或 Init 方法。容器然后将会话对象的引用返回给客户端。实例此时处于

方法准备好的状态。

注意：当有状态会话 bean 被查找到或通过显式 JNDI 查找机制得到时，容器必须提供一个新的有状态会话 bean 实例，正如 Java EE 规范要求的那样（节“Java 命名和目录接口（JNDI）命名上下文”）。

- 会话 bean 实例此时已为客户端的业务方法做好了准备。基于在会话 bean 元数据注释和/或部署文件中的事务属性和与客户端调用关联的事务上下文，业务方法或执行在事务上下文中或执行在未指定的事务上下文中（在图中表示为“tx 方法”和“non-tx 方法”）。参见第 13 章了解容器如何处理事务。
- 当实例处于方法准备好的状态时，非事务方法就可以被执行。
- 事务方法的调用会导致实例被包含到一个事务中。当会话 bean 实力被包含到一个事务时，容器触发这个实例上的 afterBegin 方法。afterBegin 方法在任何作为事务一部分的业务方法或业务方法拦截器方法被执行之前被调用。实例将与事务关联且这个关联会一直持续到事务完成。
- 被客户端调用的处于事务中的会话 bean 方法现在可以被代理到 bean 实例。如果客户端企图调用会话 bean 上的方法，但 bean 中该方法的元数据和/或部署元素要求容器在不同的事务上下文中或未指定的事务上下文中调用该方法，那么将会发生错误。
- 如果已经请求事务提交，事务服务在真正提交事务之前通知容器，容器调用实例上的 beforeCompletion。当 beforeCompletion 被调用时，实例应当将所有缓存的数据更新到数据库（注：注意，如果使用 Java 持久化 API，则持久化提供商将使用 beforeCompletion 通知来自动将容器管理的持久化上下文的更新刷新到数据库中。参见【2】）。如果请求事务回滚，那么容器不会调用 beforeCompletion 而直接设置回滚状态。如果事务已经标记为回滚，容器可以不调用 beforeCompletion 方法（实例也不向数据库写任何缓存的更新）。
- 事务服务然后试着提交事务，结果是要么提交，要么回滚。
- 当事务完成时，容器调用 afterCompletion，设置完成的状态（提交或回

滚）。如果发生回滚，bean 实例可能需要重设它的会话状态为事务开始时的状态。

- 容器的缓存算法可以决定 bean 实例应当被从内存中回收。（这可以在每个方法结束时做，或者使用 LRU 策略）。容器调用 PrePassivate 生命周期回调拦截器方法，如果有的话。在完成这之后，容器将实例的状态保存到二级存储。会话 bean 只能够在事务间被钝化，不在事务内。
- 当实例处于钝化状态时，容器可以在部署人员指定的超时到期时删除会话对象。会话对象持有的所有对象引用和句柄都将变成无效的。如果客户端企图调用 bean 业务接口的方法，容器将抛出 javax.ejb.NoSuchEJBException（注：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，则抛出 java.rmi.NoSuchObjectException）。如果使用 EJB2.1 客户端视图，如果客户端是远程客户端，则容器将抛出 java.rmi.NoSuchObjectException，或如果客户端是本地客户端，则容器抛出 javax.ejb.NoSuchObjectLocalException。
- 如果客户端调用一个已经被钝化的会话 bean 实例，那么容器激活这个实例。为了激活会话 bean 实例，容器从二级存储中恢复实例的状态并调用 PostActivate 方法，如果有的话。
- 会话 bean 实例再次为客户端方法做好准备。
- 当客户端调用已被设计为 remove 方法的 bean 方法，或 home 或组件接口上的 remove 方法时，容器在 remove 方法完成后，调用 bean 的 PreDestroy 生命周期回调拦截器方法（如果有的话）（注：如果 Remove 注释符指定了 retainifException 为 true，且 Remove 方法抛出了一个业务异常，那么实例不会被删除（且 PreDestroy 也不会被调用））。这终止了会话 bean 实例的生命和它关联的会话对象。如果客户端对后企图调用这个 bean 业务接口上的方法，则容器抛出 javax.ejb.NoSuchEJBException(注：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么向客户端抛出 java.rmi.NoSuchObjectException)。如果使用 EJB2.1 客户端视图，如

果客户端是远程客户端，则所有后续的调用都会抛出 java.rmi.NoSuchObjectException，或者如果客户端是本地客户端，则抛出 javax.ejb.NoSuchObjectLocalException 。（java.rmi.NoSuchObjectException 是 java.rmi.RemoteException 的子类；javax.ejb.NoSuchObjectLocalException 是 javax.ejb.EJBException 的子类）。注意，容器也可以在 EJB 对象到期后调用 PreDestroy 方法来删除会话对象，而不需要客户端来调用删除对象。如果 Remove 方法成功完成或如果 retainIfException 为 false 的 Remove 方法抛出一个应用异常或抛出一个系统异常，那么不会调用 SessionSynchronization 方法。如果 retainIfException 为 true 的 Remove 方法抛出一个应用异常，那么 bean 既不被销毁也不被丢弃，且在事务的最后调用 bean 上的 SessionSynchronization 方法，如果有的话。

如果会话 bean 类实现了 SessionSynchronization 接口，那么容器必须直接或间接地调用 afterBegin，beforeCompletion 和 afterCompletion 方法。如果会话 bean 类没有实现 SessionSynchronization 接口，那么容器不调用这些方法。

4.4.1 在有状态会话 bean 类的方法中允许的操作

表 1 定义了有状态会话 bean 类的方法，从那些方法中会话 bean 实例可以获取 javax.ejb.SessionContext 接口的方法，java:comp/env 环境命名上下文，资源管理器，Timer 方法，EntityManager 和 EntityManagerFactory 方法，以及其他企业 bean。

如果会话 bean 实例企图调用 SessionContext 接口的方法，且访问在表 1 中不允许，那么容器必须抛出 java.lang.IllegalStateException。

如果会话 bean 实例企图访问资源管理，企业 bean，实体管理器，实体管理器工厂且这种访问时表 1 不允许的，则 EJB 架构没有定义容器的行为。

如果会话 bean 实例企图调用 Timer 接口的方法且这种访问是表 1 不允许的，那么容器必须抛出 java.lang.IllegalStateException。

表 1 在有状态会话 bean 的方法中允许的操作

Bean 方法	可以执行下列操作的 bean 方法	
	容器管理的事务分割	Bean 管理的事务分割
构造器	-	-
依赖注入方法（例如，setSessionContext）	SessionContext 方法： getEJBHome， getEJBLocalHome，lookup。 JNDI 获取 java:comp/env	SessionContext 方法： getEJBHome， getEJBLocalHome，lookup。 JNDI 获取 java:comp/env
PostConstruct， PreDestroy， PrePassivate, PostActivate	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， isCallerInRole， getEJBObject， getEJBLocalObject，lookup。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， isCallerInRole， getEJBObject， getEJBLocalObject， getUserTransaction，lookup。 UserTransaction 的方法。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。
来自业务接口或组件接口的业务方法；业务方法的拦截器方法	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， getRollbackOnly，	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， isCallerInRole，

	isCallerInRole， setRollbackOnly， getEJBObject， getEJBLocalObject， getInvokeBusinessInterface， lookup。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 的方法。	getEJBObject， getEJBLocalObject， getInvokeBusinessInterface， getUserTransaction，lookup。 UserTransaciton 的方法。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 的方法。
afterBegin beforeCompletion	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， getRollbackOnly， isCallerInRole， setRollbackOnly， getEJBObject， getEJBLocalObject， lookup。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。	N/A（使用 bean 管理事务分割的 bean 不能实现 SessionSynchronization 接口）

	Timer 的方法。	
afterCompletion	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， isCallerInRole， getEJBObject， getEJBLocalObject， lookup。 JNDI 获取 java:comp/env。	

注意：

- 使用容器管理事务分割的会话 bean 的 PostConstruct，PreDestroy，PrePassivate，PostActivate，Init 和/或 ejbCreate<METHOD>，ejbRemove，ejbPassivate 和 ejbActivate 方法执行在未指定的事务上下文中。参考节 13.6.5 了解容器如何在未指定的事务上下文中执行这些方法。

附加限制：

- SessionContext 接口的 getRollbackOnly 和 setRollbackOnly 方法应当只用于在事务上下文中的会话 bean 方法。如果实例没有关联事务，则调用它的方法时容器必须抛出 java.lang.IllegalStateException。

在表 1 中不允许的操作的原因如下：

- 如果会话 bean 没有定义 EJB3.0 业务接口，则不允许调研 getBusinessObject 方法。
- 如果会话 bean 没有定义 EJB3.0 业务接口或不是通过业务接口调用会话 bean，则不允许调用 getInvokeBusinessInterface 方法。
- 如果会话 bean 没有定义远程客户端视图，则不允许调用 getEJBObject 和 getEJBHome 方法。
- 如果客户端没有定义本地客户端视图，则不允许调用 getEJBLocalObject

和 getEJBLocalHome 方法。

- 对于没有有意义的事务上下文的会话 bean 方法和所有 bean 管理事务分割的会话 bean，不允许调用 getRollbackOnly 和 setRollbackOnly。
- 对于没有有意义事务上下文和/或客户端安全上下文的会话 bean 方法，不允许调用资源管理器和企业 bean。
- 使用容器管理事务分割的企业 bean 不能获取 UserTransaction 接口。
- 有状态会话 bean 不可以获取 TimerService。
- 有状态会话 bean 不可以调用 getMessageContext 方法。
- 没有为实例设置会话对象标识的会话 bean 方法，不允许调研 getEJBObject 和 getEJBLocalObject。

4.4.2 异常处理

从会话 bean 类的任何方法抛出的 RuntimeException（包括业务方法和生命周期回调拦截器方法）会将 bean 的状态转换为“不存在”状态。异常处理在第 14 章中描述。参见节 12.4.2 了解当多个这样的方法应用到 bean 类上时生命周期回调拦截器方法的规则。

从客户端的角度来看，对应的会话对象不再存在。如果客户端随后调用 bean 业务接口上的方法，则容器将抛出 javax.ejb.NoSuchEJBException（如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么抛出 java.rmi.NoSuchObjectException）。如果使用 EJB2.1 客户端视图，如果客户端是远程客户端则容器将抛出 java.rmi.NoSuchObjectException，如果客户端是本地客户端则抛出 javax.ejb.NoSuchObjectException。

4.4.3 错失调用 PreDestroy

Bean 提供者不能假设容器总会为会话 bean 调用 PreDestroy 生命周期回调拦截器方法（或 ejbRemove 方法）。下面的场景就会造成不会调用 PreDestroy 方法：

- EJB 容器宕机。

- 从实例的方法中向容器抛出系统异常。
- 当实例处于钝化（passive）状态时不活动的客户端超时。超时由部署人员用 EJB 容器实现特有的方式指定。

如果在 PostConstruct 方法（或 ejbCreate<METHOD>方法）和/或在业务方法中分配了资源，且通常在 PreDestroy 方法中释放它们，那么这些资源在上述场景中不会被自动释放。使用会话 bean 的应用应当提供一些清理机制来定期清理未释放的资源。

例如，如果购物卡组件实现为一个会话 bean，且会话 bean 将购物卡的内容存储到数据库中，那么应用应当提供一个程序来定期从数据库中删除“被抛弃的”购物卡。

4.4.4 对事务的限制

状态图暗示了下列在调用业务方法的客户端的事务范围上的限制。这些限制是容器强加的但客户端程序必须遵守。

- 会话 bean 实例可以同时参与多个事务。
- 如果会话 bean 实例正参与一个事务，那么客户端调用会话 bean 的方法，以使容器在与 bean 注释符或配置元素指定的事务属性不同的事务上下文或未指定的事务上下文中执行那个方法将是错误的。在这种情况下，将会向 bean 业务接口的客户端抛出 javax.ejb.EJBException（如果业务接口继承了 java.rmi.Remote，那么抛出 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图，那么如果客户端是远程客户端则容器抛出 java.rmi.RemoteException，否则如果客户端是本地客户端那么容器抛出 javax.ejb.EJBException。
- 如果会话 bean 正参与一个事务，那么不能调用会话 bean 的 home 或组件接口中的 remove 方法。容器必须检测这种调用的发生并向客户端抛出 javax.ejb.RemoveException。容器不应当标记客户端事务回滚，这样可以让客户端能够恢复。

4.5无状态会话 bean

无状态会话 bean 是实例没有会话状态的会话 bean。这意味着当它们不服务于一个客户端调用的方法时，所有的 bean 实例是等价的。

术语“无状态”意味着实例对特定客户端来说没有状态。但是，实例的实例变量可以跨多个客户端方法调用保留状态。这些状态有如打开的数据库连接和对一个企业 bean 对象的引用。

如果 Bean 提供者跨多个方法调用保留应用状态则必须小心。特殊情况下，不应当通过多个本地接口方法调用返回对通用 bean 状态的引用。

因为所有无状态会话 bean 实例都是对等的，所有容器可以选择任何一个实例来代理客户端调用的方法。这意味着，例如，容器可以将位于同一个事务中的同一个客户端代理到不同的实例，且容器可以将来自多个事务的请求交叉代理到同一个实例。

容器只需要保留需要服务于当前客户端的实例数。由于客户端“思考时间”（译者注：客户端调用有时间间隔），因此这个数量通常是少于活动客户端的数量。无状态会话 bean 不需要钝化。如果需要增加处理客户端请求的实例，那么容器就创建一个。如果无状态会话 bean 不需要处理当前客户端的工作，则容器可以销毁它。

因为无状态会话 bean 最小化了用于支持大量客户端的资源（依赖于容器的实现），使用无状态会话 bean 的应用比使用有状态会话 bean 的应用有更大的规模。但是，随着使用无状态 bean 的客户端应用复杂性的增加，这个好处会越来越小。

兼容性提示：使用 EJB2.1 客户端视图接口的本地和远程客户端使用 home 接口 create 和 remove 方法的方式和有状态会话 bean 是一样的。对于 EJB2.1 客户端，现象上好像是客户端在控制会话对象的生命周期。但是，容器处理 create 和 remove 调用却不一定必须创建和删除一个 EJB 实例。无状态会话 bean 的 home 接口必须有一个无参的 create 方法。远程 home 接口的 create 方法必须返回会话 bean 的远程接口。本地 home 接口的 create 方法必须返回会话 bean 的本地接口。

在 home 接口中不能有其他 create 方法。

在客户端和无状态实例间没有固定的映射关系。容器简单地将客户端的工作代理给任何可获得的实例，这些实例已经为客户端的方法做好了准备（译者注：即需要的资源都已经到位，处于准备好的状态）。

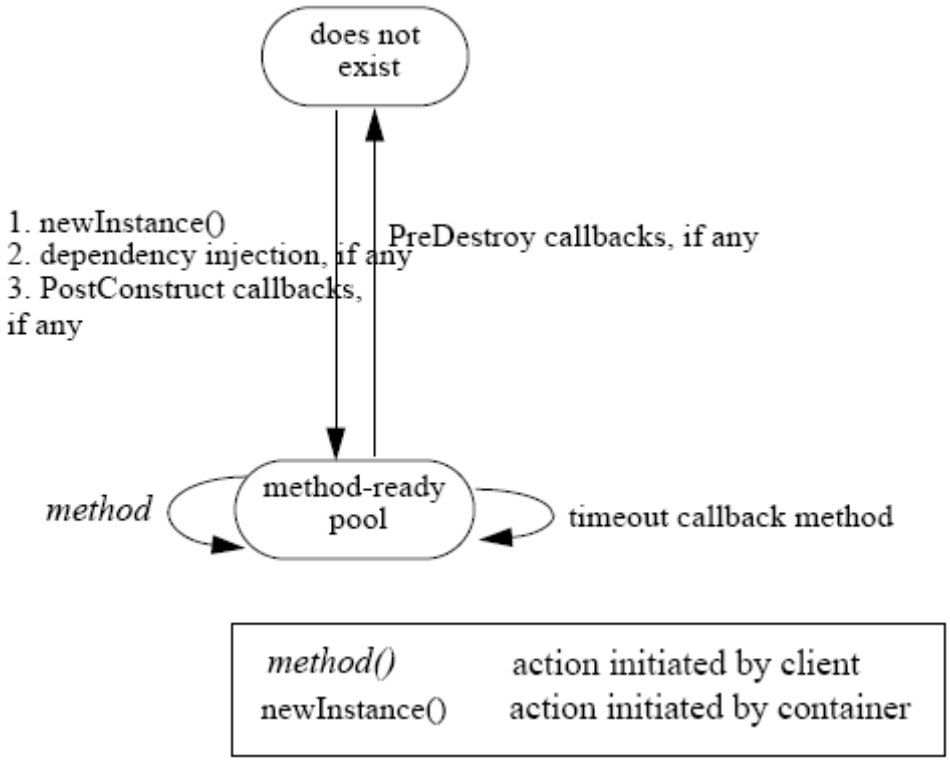
无状态会话 bean 不能实现 javax.ejb.SessionSynchronization 接口。

4.5.1 无状态会话 bean 状态图

当客户端调用无状态会话 bean 对象的一个方法或通过它的 web 服务客户端视图调用它的一个方法时，容器选择一个方法准备好的实例并将方法调用代理给它。

下面的图解释了无状态会话 bean 实例的生命周期。

图 6 无状态会话 bean 的生命周期



下面的步骤描述了会话 bean 实例的生命周期：

- 当容器调用会话 bean 实例类的 newInstance 方法来创建一个新会话 bean 实例时，无状态会话 bean 实例的生命就开始了。接着，容器注入 bean 的 SessionContext（如果申请了），并执行其他有元数据注释或部署文件

指定的依赖注入。容器然后调用 bean 的 PostConstruct 生命周期回调拦截器方法（如果有的话）。容器能够在任何时间执行实例创建——它和客户端调用业务方法或 create 方法没有任何的关系。

- 会话 bean 此时已经为代理来自客户端对业务方法的调用或来自容器对超时回调方法的调用做好了准备。
- 当容器不再需要实例时（通常是当容器想要减少池内实例的数量时），容器调用它的 PreDestroy 生命周期回调拦截器方法，如果有的话。这就终止了无状态会话 bean 实例的生命。

4.5.2 可以在无状态会话 bean 类的方法内执行的操作

表 2 定义了无状态会话 bean 类的方法，在这些方法中 bean 实例可以获取 javax.ejb.SessionContext 接口的方法，java:/comp/env 环境命名上下文，资源管理器，TimerService 和 Timer 方法，EntityManager 和 EntityManagerFactory 方法和其它企业 bean。

如果会话 bean 实例企图调用 SessionContext 接口的方法且在表 2 中不允许这种调用，则容器必须抛出 java.lang.IllegalStateException。

如果会话 bean 实例企图调用 TimerService 或 Timer 接口的方法且在表 2 中不允许这种调用，则容器必须抛出 java.lang.IllegalStateException。

如果会话 bean 实例企图访问资源管理器、企业 bean、实体管理器或实体管理器工厂，且在表 2 中不允许这种调用，那么 EJB 规范没有定义容器的行为。

表 2 在无状态会话 bean 的方法中允许的操作

Bean 方法	可以执行下列操作的 bean 方法	
	容器管理的事务分割	Bean 管理的事务分割
构造器	-	-
依赖注入方法（例如，setSessionContext）	SessionContext 方法：getEJBHome，getEJBLocalHome，lookup。JNDI 获取 java:comp/env	SessionContext 方法：getEJBHome，getEJBLocalHome，lookup。JNDI 获取 java:comp/env

PostConstruct， PreDestroy	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getEJBObject， getEJBLocalObject， getTimerService，lookup。 JNDI 获取 java:comp/env。 访问 EntityManagerFactory。	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getEJBObject， getEJBLocalObject， getUserTransaction， getTimerService，lookup。 JNDI 获取 java:comp/env。 访问 EntityManagerFactory。
来自业务接口或组件接口的业务方法； 业务方法的拦截器方法	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， getRollbackOnly， isCallerInRole， setRollbackOnly， getEJBObject， getEJBLocalObject， getTimerService， getInvokeBusinessInterface， lookup。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， isCallerInRole， getEJBObject， getEJBLocalObject， getInvokeBusinessInterface， getUserTransaction， getTimerService，lookup。 UserTransaciton 的方法。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 TimerService 和 Timer 的方

	TimerService 和 Timer 的方法。	法。
来自 web 服务终端的业务方法	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， getRollbackOnly， isCallerInRole， setRollbackOnly， getEJBObject， getEJBLocalObject， getTimerService， getMessageContext， lookup。 Message 上下文方法 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 TimerService 和 Timer 的方法。	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome， getCallerPrincipal， getRollbackOnly， isCallerInRole， setRollbackOnly， getEJBObject， getEJBLocalObject， getTimerService， getMessageContext， lookup。 UserTransaction 方法 Message 上下文方法 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 TimerService 和 Timer 的方法。
超时回调方法	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome，	SessionContext 方法： getBusinessObject， getEJBHome， getEJBLocalHome，

	getCallerPrincipal， isCallerInRole， getRollbackOnly， setRollbackOnly， getEJBObject， getEJBLocalObject， getTimerService， lookup。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 TimerService 和 Timer 的方法。	getCallerPrincipal， isCallerInRole， getEJBObject， getEJBLocalObject， getUserTransaction， getTimerService， lookup。 UserTransaction 方法。 JNDI 获取 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 TimerService 和 Timer 的方法。

附加限制：

- SessionContext 接口的 getRollbackOnly 和 setRollbackOnly 方法应当只用于在事务上下文中的会话 bean 方法。如果实例没有关联事务，则调用它的方法时容器必须抛出 java.lang.IllegalStateException。

在表 2 中不允许的操作的原因如下：

- 如果会话 bean 没有定义 EJB3.0 业务接口，则不允许调研 getBusinessObject 方法。
- 如果会话 bean 没有定义 EJB3.0 业务接口或不是通过业务接口调用会话 bean，则不允许调用 getInvokeBusinessInterface 方法。
- 如果会话 bean 没有定义远程客户端视图，则不允许调用 getEJBObject 和 getEJBHome 方法。
- 如果客户端没有定义本地客户端视图，则不允许调用 getEJBLocalObject 和 getEJBLocalHome 方法。
- 对于没有有意义的事务上下文的会话 bean 方法和所有 bean 管理事务分

割的会话 bean，不允许调用 getRollbackOnly 和 setRollbackOnly。

- 由容器通过会话 bean 的 web 服务终端调用的会话 bean 方法中不允许调用 getMessageContext 方法。getMessageContext 方法返回实现了 JAX-RPC web 服务终端的无状态会话 bean 的 javax.xml.rpc.handler.MessageContext 接口。
- 对于没有有意义事务上下文和/或客户端安全上下文的会话 bean 方法，不允许调用资源管理器、企业 bean 和 EntityManager。
- 使用容器管理事务分割的企业 bean 不能获取 UserTransaction 接口。

4.5.3 异常处理

由企业 bean 类的方法抛出的 RuntimeException（包括业务方法和由容器调用的生命周期回调拦截器方法）会引起实例的状态变成“不存在的”。异常处理在第 14 章中描述。参见 12.4.2 了解当多个这样的方法应用到 bean 类上时生命周期回调拦截器方法的规则。

从客户端视角来看，会话对象依然存在。客户端依然可以访问这个会话对象，因为容器可以将客户端的请求代理给另外一个实例。

4.6Bean 提供者的责任

本节描述 Bean 提供者为保证会话 bean 能被部署到任何 EJB 容器中的责任。

4.6.1 类和接口

会话 bean 提供者有责任提供下列类文件（注：注意，由 bean 提供者提供的接口可能已经由工具生成了）：

- 会话 bean 类。
- 如果会话 bean 提供了 EJB3.0 的本地或远程客户端视图，则提供会话 bean 的业务接口。
- 如果会话 bean 提供了 EJB2.1 远程客户端视图，则提供会话 bean 的远程

接口和远程 home 接口。

- 如果会话 bean 提供了 EJB2.1 本地客户端视图，则提供会话 bean 的本地接口和本地 home 接口。
- 会话 bean 的 web 服务终端接口，如果有的话。
- 拦截器类，如果有的话。

提供了 web 服务客户端视图的无状态会话 bean 的 Bean 提供者也可以为 bean 定义 JAX-WS 或 JAX-RPC 消息处理器。在【31】和【32】定义了对这样的消息处理器的要求。

4.6.2 会话 bean 类

对会话 bean 类有如下要求：

- 类必须定义为 public，不能是 final 和 abstract 的。类必须是顶级类。
- 这个类型必须有一个 public 无参构造器。容器使用这个构造器来创建会话 bean 类的实例。
- 类不能定义 finalize()方法。
- 类必须实现 bean 的业务接口或 bean 业务接口的方法，如果有的话。
- 类必须实行 bean 的 EJB2.1 客户端视图接口的业务方法，如果有的话。

可选的要求：

- 类可以直接或间接地实现 javax.ejb.SessionBean 接口。
- 如果类是有状态会话 bean，那么它可以实现 javax.ejb.SessionSynchronization 接口。
- 类可以实现会话 bean 的 web 服务终端接口或组件接口。
- 如果类是无状态会话 bean，那么它可以实现 javax.ejb.TimedObject 接口。参见第 18 章“Timer 服务”。
- 类型可以实现 ejbCreate 方法。
- 会话 bean 类可以有超类和/或超级接口。如果会话 bean 有超类，那么业务方法、生命周期回调拦截器方法、超时回调方法、可选的 SessionSynchronization 接口的方法、Init 或 ejbCreate<METHOD>方法、

Remove 方法和 SessionBean 接口的方法可以定义在会话 bean 类中，或者它的任何父类中。会话 bean 类的父类不能是会话 bean。

- 会话 bean 类可以实现除了 EJB 规范要求以外的其他方法（例如由业务方法内部调用的帮助方法）。

4.6.3 生命周期回调拦截器方法

可以为会话 bean 订阅 PostConstruct、PreDestroy、PrePassivate 和 PostActivate 生命周期回调拦截器方法。如果为无状态会话 bean 定义了 PrePassivate 或 PostActivate，则忽略它们（注：注意，这可能导致一些问题，如使用缺省的拦截器）。

兼容性提示：如果 PostConstruct 生命周期回调拦截器方法是 ejbCreate 方法，如果 PreDestroy 生命周期回调拦截器方法方法是 ejbRemove 方法，如果生命周期回调拦截器方法 PostActivate 是 ejbActivate 方法，或如果 PrePassviate 生命周期回调拦截器方法方法是 ejbPassivate 方法是 ejbPassivate 方法，那么这些回调方法必须实现在 bean 类本身上（或它的超类上）。除了这些情况外，方法名可以是任意的，但不能以“ejb”开头以避免和定义在 javax.ejb.EnterpriseBean 接口中的回调方法冲突。

生命周期回调拦截器方法可以定义在 bean 类和/或 bean 的拦截器类上。生命周期回调拦截器方法的定义规则定义在节 12.4“用于生命周期事件回调的拦截器”。

4.6.4 ejbCreate<METHOD>方法

有 home 接口的会话 bean 可以定义一个或多个 ejbCreate<METHOD>方法。这些 ejbCreate 方法只用于 EJB2.1 组件。这些 ejbCreate 方法的标识符必须遵循以下规则：

- 必须以 ejbCreate 作为名字的前缀。
- 方法必须声明为 public。

- 方法不能声明为 final 或 static。
- 返回类型必须是 void。
- 如果有一个与会话 bean 远程 home 接口中的 ejbCreate<METHOD>方法对应的 create<METHOD>方法，那么它的参数必须是 RMI/IIOP 的合法类型。
- 无状态会话 bean 可以只定义一个无参的 ejbCreate 方法。
- throws 子句可以定义任意的应用异常，可能包括 javax.ejb.CreateException。

兼容性提示：EJB1.0 允许 ejbCreate 方法抛出 java.rmi.RemoteException 来表示非应用异常。这在 EJB1.1 中被废弃——EJB1.1 或 EJB2.0 或后期的企业 bean 应当抛出 javax.ejb.EJBException 或另外一个 RuntimeException 来向容器表示非应用异常（参见节 14.2.2）。EJB2.0 和以后的企业 bean 不应当从 ejbCreate 方法中抛出 java.rmi.RemoteException。

4.6.5 业务方法

会话 bean 类可以定义零到多个业务方法，它的名称必须遵循这些规则：

- 方法名可以是任意的，但不能以“ejb”开头以避免与 EJB 架构使用的回调方法冲突。
- 业务方法必须声明为 public。
- 方法不能声明为 final 获 static。
- 如果方法对应于会话 bean 远程业务接口或远程接口的业务方法，那么它的参数和返回类型必须是 RMI/IIOP 的合法类型。
- 如果方法是 web 服务方法或对应于会话 bean 的 web 服务终端接口的方法，那么它的参数和返回类型必须是 JAX-WS/JAX-RPC 的合法类型。
- throws 子句可以定义任意的应用异常。

兼容性提示：EJB1.0 允许 ejbCreate 方法抛出 java.rmi.RemoteException 来表示非应用异常。这在 EJB1.1 中被废弃——EJB1.1 或 EJB2.0 或后期的企业 bean 应当抛出 javax.ejb.EJBException 或另外一个 RuntimeException 来向容器表示非应

用异常（参见节 14.2.2）。EJB2.0 和以后的企业 bean 不应当从 ejbCreate 方法中抛出 java.rmi.RemoteException。

4.6.6 会话 bean 的业务接口

下面是对会话 bean 业务接口的要求：

- 接口不能继承 javax.ejb.EJBObject 或 javax.ejb.EJBLocalObject 接口。
- 如果业务接口是远程业务接口，则参数和返回类型必须是 RME/IIOP 的有效类型。远程业务接口不要求或不希望是 java.rmi.Remote 接口。Throws 子句不应当包含 java.rmi.RemoteException。如果接口继承了 java.rmi.Remote，则业务接口的方法只可以抛出 java.rmi.RemoteException。
- 接口可以有父接口。
- 如果接口是一个远程业务接口，那么它的方法不能暴露本地接口类型、计时器或计时器句柄、或用于 EJB2.1 实体 bean 的手管理集合类来作为参数或返回值。
- Bean 类必须实现业务接口，或者业务接口必须使用 Local 或 Remote 注释符或部署文件被指定接口为 bean 的本地或远程业务接口。应用以下规则：
  - 如果 bean 类只实现了一个接口，则这个接口被认为是 bean 的业务接口。这个业务接口将是本地接口，除非它被用 Remote 注释符或部署文件指派为远程业务接口。
  - Bean 类可以有多个接口。如果 bean 类有多个接口——除了下面列出的以外——bean 类的任何业务接口必须显式地使用 Local 或 Remote 注释符或部署文件被指派。
  - 当 bean 类有多个接口时，这些接口不包括下列的接口：java.io.Serializable；java.io.Externalizable；javax.ejb 包内的任何借口。
  - 同一个业务接口不能既是本地接口又是远程接口（注：如果 Local 和/或 Remote 注释符既被指定 bean 类上又被指定到引用的接口上，

则它们的值必须相同）。

- 然而 bean 类通常都实现它的业务接口，如果 bean 类使用注释符或部署文件来指派业务接口，那么不要求指明 bean 类为接口的实现。

4.6.7 会话 bean 的远程接口

下面是对会话 bean 远程接口的要求：

- 接口必须继承 javax.ejb.EJBObject 接口。
- 在接口内定义的方法必须遵循 RME/IIOP 的规则。这意味着它们的参数和返回值必须是 RMI/IIOP 的有效类型，且它们的 throws 子句必须包含 java.rmi.RemoteException。
- 对于定义在远程接口内的每个方法，必须在会话 bean 类中有一个对应的方法。这个对应的方法必须有：
  - 相同的名字。
  - 相同的参数数量和类型，和相同的返回值。
  - 定义在会话 bean 类中对应方法的 throws 中的所有异常，也必须在远程接口中的方法的 throws 子句中定义。
- 远程接口不能暴露本地接口类型、本地 home 接口类型、计时器或计时器句柄、或由容器管理持久化的实体 bean 使用的受管理集合类作为参数或返回值。

4.6.8 会话 bean 的远程 home 接口

对会话 bean 的远程 home 接口要求如下：

- 接口必须继承 javax.ejb.EJBHome 接口。
- 在这个接口内定义的方法必须遵循 RMI//IIOP 规则。这意味着它们的参数和返回值必须是 RMI/IIOP 的有效类型，且它们的 throws 语句必须包括 java.rmi.RemoteException。
- 远程 home 接口可以有父接口。使用接口继承遵循了用于远程接口定义

的 RMI/IIOP 规则。

- 会话 bean 的远程 home 接口必须定义一个或多个 create<METHOD>方法。无状态会话 bean 必须定义一个无参的 create 方法。
- 无状态会话 bean 的每个 create 方法必须命名为 create<METHOD>，且它们必须对应于一个定义在会话 bean 类中的 Init 方法或 ejbCreate<METHOD>方法。匹配的 Init 方法或 ejbCreate<METHOD>方法有相同的参数数量和类型。（注意，返回类型是不同的）。无状态会话 bean 的 create 方法必须命名为“create”，但不需要有一个匹配的“ejbCreate”方法。
- create<METHOD>方法的返回类型必须是会话 bean 的远程接口类型。
- 在 ejbCreate<METHOD>方法 throws 语句中定义的所有异常必须定义在远程 home 接口中对应的 create<METHOD>方法的 throws 语句中。
- throws 语句中必须包括 javax.ejb.CreateException。

4.6.9 会话 bean 的本地接口

下面是对会话 bean 本地接口的要求：

- 接口必须继承 javax.ejb.EJBLocalObject 接口。
- 在本地接口中的方法的 throws 语句不能包含 java.rmi.RemoteException。
- 本地接口可以有父接口。
- 对于定义在本接口内的每个方法，必须在会话 bean 类中有一个对应的方法。这个对应的方法必须有：
  - 相同的名字。
  - 相同的参数数量和类型，和相同的返回值。
  - 定义在会话 bean 类中对应方法的 throws 中的所有异常，也必须在本地接口中的方法的 throws 子句中定义。

4.6.10 会话 bean 的本地 home 接口

对会话 bean 的本地 home 接口要求如下：

- 接口必须继承 javax.ejb.EJBLocalHome 接口。
- 在本地 home 接口中的方法的 throws 语句不能包含 java.rmi.RemoteException。
- 本地接口可以有父接口。
- 会话 bean 的本地 home 接口必须定义一到多个 create<METHOD>方法。无状态会话 bean 必须定义一个无参的 create 方法。
- 有状态会话 bean 的每个 create 方法必须命名为 create<METHOD>，且它必须对应一个 Init 方法或 ejbCreate<METHOD>方法。这个匹配的 Init 方法或 ejbCreate<METHOD>方法必须有相同的参数数量和类型。（注意，返回类型可以不同）。用于无状态会话的 create 方法必须命名为“create”，但不需要有一个匹配的“ejbCreate”方法。
- create<METHOD>方法的返回类型必须是会话 bean 的本地接口类型。
- 定义在 ejbCreate<METHOD>方法的 throws 语句内的所有异常必须定义在本地 home 接口中与之匹配的 create<METHOD>方法的 throws 语句中。
- throws 语句必须包含 javax.ejb.CreateException。

4.6.11 会话 bean 的 web 服务终端接口

EJB3.0 对要实现 web 服务终端的会话 bean 的 web 服务终端接口的定义没有要求。

下面是对使用 JAX-RPC web 服务终端接口的会话 bean 的要求。对于 Java EE 规范的 JAX-WS 和 Web 服务不要求为 web 服务终端单独定义接口。在 JAX-WS 和 Web 服务中对 web 服务终端的要求在【32】和【31】中给出。

下面是对无状态会话 bean 的 web 服务终端接口的要求。Web 服务终端接口必须遵循 JAX-RPC 服务终端接口的规则。

- Web 服务终端接口必须继承 java.rmi.Remote 接口。
- 定义在接口内的方法必须遵循 JAX-RPC 服务终端接口的规则。这意味着它们的参数和返回值的类型必须是 JAX-RPC 的有效类型，且它们的 throws 语句中必须包含 java.rmi.RemoteException。throws 语句也可以包含其他应用异常。

注意：JAX-RPC 持有者类可以用作方法参数。为了处理入参和出参，JAX-RPC 规范要求持有者类作为 WSDL 操作的标准 Java 映射的一部分。持有者类实现了 javax.xml.rpc.holders.Holder 接口。参见 JAX-RPC 规范【25】做进一步了解。

- 对于在 web 服务终端接口内的每个方法，在会话 bean 类中必须有一个匹配的方法。匹配的方法必须有：
  - 相同的名字。
  - 相同的参数数目和类型，相同的返回类型。
  - 定义在会话 bean 类中对应方法的 throws 中的所有异常，也必须在 web 服务终端接口中的方法的 throws 子句中定义。
- Web 服务终端接口不能将 EJBObject 或 EJBLocalObject 作为参数或返回值。数组或 JAX-RPC 值类型不能将 EJBObject 或 EJBLocalObject 作为被包含的类型（译者注：即数组内的元素）。Web 服务终端接口方法不能暴露业务接口类型、本地或远程接口类型、本地或远程 home 接口类型、计时器或计时器句柄、或由使用容器管理持久化实体 bean 试用的受管理集合类型作为参数或返回值或值类型的字段。
- 由 web 服务终端接口使用的 JAX-RPC 序列化规则应用于任何值类型。如果应用 Java 序列化语义是重要的，那么 Bean 提供者应当为应用在 JAX-RPC 序列化的 Java 序列化的语义使用受限的 JAX-RPC 值类型集。参见 JAX-RPC 规范【25】了解更详细的信息。
- Web 服务终端接口不能包含常量（类似 public final static）声明。
- Bean 提供者必须在部署文件中通过 service-endpoint 元素指派 web 服务终端接口。如果 web 服务终端被 web 服务部署描述作为在【31】中定义

的一样来引用，那么 web 服务中的本身只被暴露在 web 服务中。

4.7容器提供者的责任

本节描述了支持会话 bean 的容器提供者的责任。容器提供者有责任提供部署工具和在运行时管理会话 bean 实例。

因为EJB 规范没有在部署工具和容器间定义API；我们假定部署工具是由容器提供者提供。但部署工具也可以由不同的提供商提供，它是由容器提供者的API。

4.7.1 生产实现类

由容器提供的部署工具负责在会话 bean 被部署时生成附加类。工具通过反射由 Bean 提供者提供的类和接口和检查部署文件来获取生成附加类所需的信息。

部署工具必须生成下面的类：

- 实现了会话 bean 业务接口的类。
- 实现了会话 bean 远程 home 接口（会话 EJBHome 类）的类。
- 实现了会话 bean 远程接口（会话 EJBObject 类）的类型。
- 实现了会话 bean 本地 home 接口（会话 EJBLocalHome 类）的类。
- 实现了会话 bean 本地接口（会话 EJBLocalObject 类）的类型。
- 实现了会话 bean 的 web 服务终端的类。

部署工具也可以生成一个类，它在会话 bean 类中混合了一些容器特有代码。这些代码，例如，帮助容器在运行时管理 bean 实例。工具可以使用子类、代理和代码生成。

部署工具也可以让生成的附加代码封装业务方法以用于客户化业务逻辑到当前的运行环境。例如，对于 AccountManager bean 上 debit 函数的封装器可以检查借贷的数额不能超过一定的限度。

4.7.2 生成 WSDL

附录【31】中描述了 web 服务终端的 WSDL 文档的生成。Java 到 WSDL 的影射必须遵循 JAX-RPC 或 JAX-WS【32】的要求。

4.7.3 会话业务接口实现类

由部署工具生成的会话业务接口的容器实现，实现了会话 bean 指定的业务方法。

每个业务方法的实现必须激活实例（如果实例处于钝化状态）、调用业务方法拦截器方法和调用对应的业务方法。

容器提供者有责任提供业务接口的 equals 和 hashCode 方法，遵循节 3.6.5 的要求。

4.7.4 会话 EJBHome 类

由部署工具生成的会话 EJBHome 类实现了会话 bean 的远程 home 接口。这个类实现了 javax.ejb.EJBHome 接口的方法和会话 bean 指定的 create<METHOD> 方法。

每个 create<METHOD>方法的实现调用对应的 ejbCreate<METHOD>方法。

4.7.5 会话 EJBObject 类

由部署工具生成的会话 EJBObject 类实现了会话 bean 的远程接口。它实现了 javax.ejb.EJBObject 接口的方法和会话 bean 指定的业务方法。

每个业务方法的实现必须激活实例（如果实例处于钝化状态）、调用业务方法拦截器方法和在实例上调用对应的业务方法。

4.7.6 会话 EJBLocalHome 类

由部署工具生成的会话 EJBLocalHome 类实现了会话 bean 的本地 home 接

口。这个类实现了 javax.ejb.EJBLocalHome 接口和会话 bean 的 create<METHOD>方法。

每个 create<METHOD>方法的实现调用对应的 ejbCreate<METHOD>方法。

4.7.7 会话 EJBLocalObject 类

由部署工具生成的会话 EJBLocalObject 类实现了会话 bean 的本地接口。它实现了 javax.ejb.EJBLocalObject 接口的方法和会话 bean 的业务方法。

每个业务方法的实现必须激活实例（如果实例处于钝化状态）、调用业务方法拦截器方法和在实例上调用对应的业务方法。

4.7.8 Web 服务终端实现类

由部署工具生成无状态会话 bean 的 web 服务终端的实现。这个类必须处理对 web 服务终端的请求，反组（unmarshall）SOAP 请求，调用业务方法拦截器方法和调用与 web 服务终端方法对应的无状态会话 bean 的方法。

4.7.9 句柄类

部署工具负责实现会话 bean 远程 home 和远程接口的句柄类。

4.7.10 EJBMetaData 类

部署工具负责实现为远程客户端视图提供元数据的类。这个类必须是有效的 RMI 值类，且必须实现 javax.ejb.EJBMetaData 接口。

4.7.11 不可重入实例

容器必须保证同一时刻只有一个线程在执行一个实例。如果客户端请求在实例正在执行另一个请求时到达，那么容器可以抛出 javax.ejb.ConcurrentAccessException 到第二个客户端（注：如果业务接口是继承

了 java.rmi.Remote 的远程业务接口，那么向客户端抛出的是 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图，那么如果客户端是远程客户端则容器可以抛出 java.rmi.RemoteException，或如果客户端是本地客户端则抛出 javax.ejb.EJBException（注：在一些特殊的环境中（如，为处理集群的 web 容器架构），容器可以排队或序列化这样的并发请求。但是客户端不能依赖这种行为）。

注意：会话对象目的是只支持一个客户端。因此，如果两个客户端都调用同一个会话对象，则这是应用错误。

这个规则的含义是应用不能对会话 bean 实例进行回路（loopback）调用。

4.7.12 事务范围，安全，异常

事务必须遵循关于事务范围、安全检查和异常处理的规则，正如在第 13、17 和 14 章分别描述的一样。

4.7.13 用于 web 服务终端的 JAX-WS 和 JAX-RPC 消息处理器

容器必须支持为 web 服务终端使用 JAX-WS 和 JAX-RPC 消息处理器。对于支持消息处理器的容器要求在【32】和【31】中描述。

如果有消息处理器，那么它们必须在任何业务拦截器方法之前被调用。

4.7.14 SessionContext

容器必须实现 SessionContext.getEJBObject 方法，以便 bean 实例可以使用 Java 语言来将返回值转换成会话 bean 的远程接口类型。特别地，bean 实例不必使用 PortableRemoteObject.narrow 方法来转换类型。

容器必须实现 EJBContext.lookup 方法，以便当使用 lookup 来查找 bean 的远程 home 接口时，bean 实例可以使用 Java 语言将返回值转换成会话 bean 的远程 home 接口类型。特别地，bean 实例不必使用 PortableRemoteObject.narrow 方法来转换类型。

5 消息驱动 bean 组件规约

本章规定了消息驱动 bean 和容器间的协议。定义了消息驱动 bean 实例的生命周期。

这一章定义了消息驱动 bean 状态管理的开发者视图和容器管理消息驱动 bean 状态的责任。

5.1概述

消息驱动 bean 是一个异步消息消费者。当消息到达目的地或消息驱动 bean 服务的终端时，容器调用消息驱动 bean。消息驱动 bean 实例是消息驱动 bean 类的实例。消息驱动 bean 是为单个消息类型定义的，与它雇用的消息监听器接口一致。

对于客户端，消息驱动 bean 是一个实现了一些业务逻辑运行在服务器上的消息消费者。客户端通过发送消息到目的地或消息驱动 bean 作为消息监听器的终端来访问消息驱动 bean。

消息驱动 bean 是匿名的。它们没有客户端可见的标识。

消息驱动 bean 实例没有会话状态。这意味着所有的 bean 实例在服务于客户端消息时都是相等的。

消息驱动 bean 实例由容器创建，它用于处理消息驱动 bean 是消费者的消息处理。它的生存时间由容器控制。

消息驱动 bean 实例对特定客户端来说没有状态。但是，消息驱动 bean 实例的实例变量可以跨客户端消息处理保含状态。例如，这种状态有数据库连接和对企业 bean 的引用。

5.2目标

消息驱动bean 模型的目标是开发一个异步处理消息的企业bean 和开发具有相同功能的其他消息监听器一样简单。

消息驱动bean 模型的远期目标是可以使用容器提供的消息驱动bean 实例池

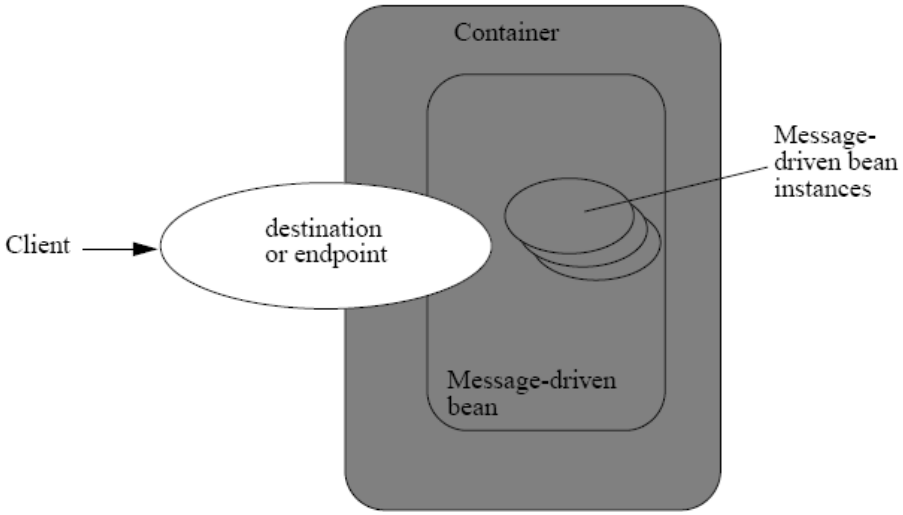
来并发的处理消息流。

5.3消息驱动 bean 的客户端视图

对客户端来说，消息驱动 bean 是一个简单的消息消费者。客户端向目的地或消息驱动 bean 作为消息监听器的终端发送消息就像向其他目的地或终端发送消息一样。消息驱动 bean 作为消息消费者，进行消息的处理。

从客户端的视角来看，消息驱动 bean 完全隐藏在目的地或消息驱动 bean 作为消息监听器的终端后面。下图解释了提供给消息驱动 bean 客户端的视图。

图 7 部署在容器内的消息驱动 bean 的客户端视图



客户端的 JNDI 命名空间可以包含目的地或安装在位于网络上多个机器多个 EJB 容器内消息驱动 bean 的终端。企业 bean 和 EJB 容器的真正位置通常对使用企业 bean 的客户端是透明的。

对消息目的地的引用可以被注入，或可以通过客户端的 JNDI 命名空间被查找到。

例如，对用于 JMS 消息驱动 bean 的队列的引用可以按下述方式被注入。

```
@Resource Queue stockInfoQueue;
```

同样，用于 StockInfo JMS 消息驱动 bean 的队列也可以用下面的代码片断被定位：

```
Context initialContext = new InitialContext();
```

Queue stockInfoQueue = (javax.jms.Queue)initialContext.lookup

(“java:comp/env/jms/stockInfoQueue”);

本节剩余部分详细描述消息驱动bean 的生命周期和消息驱动bean 和容器间的协议。

5.4消息驱动 bean 实例和容器间的协议

从消息驱动 bean 实例的创建到销毁，它都生存在容器中。容器为消息驱动 bean 提供安全、并发、事务和其它服务。容器管理消息驱动 bean 实例的生命周期，当调用 bean 时通知实例并提供全方位的服务来保证消息驱动 bean 的实现是规模化的，且能支持大量消息的并发处理。

从 bean 提供者的视角来看，消息驱动 bean 与容器具有相同的生存时间。当容器启动时保证消息驱动 bean 存在并在消息转发开始前准备好 bean 实例来接收异步消息都是容器的责任。

容器本身对消息驱动 bean 实例没有服务要求。容器调用 bean 实例是为了让实例获取容器的服务和转发容器产生的通知。

由于所有的消息驱动 bean 实例都是相等的，因此客户端消息可以被转发给任何一个实例。

5.4.1 消息驱动 bean 要求的元数据

消息驱动 bean 必须用 MessageDriven 注释符注释或在部署文件中指明是消息驱动 bean。

5.4.2 要求的消息监听器接口

消息驱动 bean 类必须实行对应于消息驱动 bean 支持的消息类型的消息监听器接口，或用 MessageDriven 元数据注释或 messaging-type 部署元素指定消息监听器接口。由消息驱动 bean 类实现的消息监听器接口不同于消息驱动 bean 支持的消息类型。

Javax.jms.MessageListener 接口的消息驱动 bean 类实现与消息驱动 bean 作为一个 JMS 消息驱动 bean 是不同的。

Bean 的消息监听器方法（例如，在 javax.jms.MessageListener 情况下是 onMessage 方法）在 bean 服务的消息到达时由容器调用。消息监听器方法包含了处理消息的业务逻辑。

Bean 的消息监听器接口可以定义多个消息监听器方法。如果消息监听器接口包含多个方法，那么由资源适配器决定调用哪个方法。参见【15】。

如果消息驱动 bean 类实现了多个接口，除了 java.io.Serializable，java.io.Externallizable 或定义在 javax.ejb 包内的接口外，那么消息监听器接口必须由 MessageDriven 注释符的 messageListenerInterface 元素或部署文件的 message-driven 元素的 messaging-type 元素来指定。

5.4.3 依赖注入

消息驱动 bean 可以使用依赖注入机制来获取对资源或环境中其他对象的引用（参见第 16 章，“企业 bean 的环境”）。如果消息驱动 bean 使用依赖注入，那么容器在 bean 实例被创建后和在消息监听器方法被调用之前注入这些引用。如果声明了对 MessageDrivenContext 的依赖，或者如果 bean 类实现了可选的 MessageDrivenBean 接口（参见节 5.4.6），那么 MessageDrivenContext 也被同时注入。如果依赖注入失败，则丢弃 bean 实例。

在 EJB3.0 API 下，bean 类可以通过依赖注入获取 MessageDrivenContext 接口，而不需要实现 MessageDrivenBean 接口。在这种情况下，Resource 注释符（或 resource-env-ref 部署元素）用于声明 bean 对 MessageDrivenContext 的依赖。参见第 16 章“企业 bean 的环境”。

5.4.4 MessageDrivenContext 接口

如果 bean 指定对 MessageDrivenContext 接口依赖（或如果 bean 类实现了 MessageDrivenBean 接口），那么容器必须为消息驱动 bean 提供

MessageDrivenContext。这可以让消息驱动 bean 实例访问容器维护的实例的上下文。MessageDrivenContext 接口有以下方法：

- setRollbackOnly 方法让实例能够标记当前事务的结果是回滚。只有使用容器管理事务分割的消息驱动 bean 实例可以使用这个方法。
- getRollbackOnly 方法可以让实例检测当前事务是否已被标记为回滚。只有使用容器管理事务分割的消息驱动 bean 实例可以使用这个方法。
- getUserTransaction 方法返回 javax.transaction.UserTransaction 接口，实例用它来分割事务和获取事务状态。只有使用 bean 管理事务分割的消息驱动 bean 实例可以使用这个方法。
- getCallerPrincipal 方法返回和调用相关的 java.security.Principal。
- isCallerPrincipal 方法继承自 EJBContext 接口。消息驱动 bean 不能调用这个方法。
- getEJBHome 和 getEJBLocalHome 方法继承自 EJBContext 接口。消息驱动 bean 不能调用这个方法。
- lookup 方法使消息驱动 bean 可以查找在 JNDI 命名空间中的环境条目。

5.4.5 消息驱动 bean 生命周期回调拦截器方法

下面是对消息驱动 bean 支持的生命周期事件回调。回调方法可以直接定义在 bean 类上，也可以定义在单独的拦截器类上（注：如果为消息驱动 bean 定义 PrePassivate 或 PostActivate 生命周期回调，则忽略它们）。参见节 5.6.4。

- PostConstruct
- PreDestroy

PostConstruct 回调发生在第一个消息监听器方法被调用之前。在所有的依赖注入已经被容器完成之后。

PostConstruct 生命周期回调拦截器方法在未指定的事务和安全上下文中执行。

PreDestroy 回调发生在 bean 从池中删除或被销毁时。

PreDestroy 生命周期回调拦截器方法在未指定的事务和安全上下文中执行。

5.4.6 可选的 MessageDrivenBean 接口

消息驱动 bean 不要求实现 javax.ejb.MessageDrivenBean 接口。

兼容性提示：MessageDrivenBean 接口是企业 JavaBean 规范的早期版本要求的。在 EJB3.0 中，以前由 MessageDrivenBean 提供的功能可以通过有选择的使用依赖注入（注入 MessageDrivenContext）和可选的生命周期回调方法来获得。

MessageDrivenBean 接口定义了两个方法：setMessageDrivenContext 和 ejbRemove。

setMessageDrivenContext 由容器调用，用于将消息驱动 bean 实例和容器维护的上下文建立关联。通常消息驱动 bean 实例将消息驱动上下文作为它状态的一部分保留。

ejbRemove 通知实例正在被容器删除。在 ejbRemove 方法中，实例释放它持有的所有资源。

在 EJB3.0 API 下，bean 类可以可选地定义 PreDestroy 回调方法用于容器删除 bean 实例时进行通知。

本规范要求将消息驱动 bean 的 ejbRemove 和 ejbCreate 方法分别看作是 PreDestroy 和 PostConstruct 生命周期回调方法。如果消息驱动 bean 实现了 MessageDrivenBean 接口，那么 PreDestroy 注释符只能应用到 ejbRemove 方法上。对使用部署文件来说也是同样的要求。

5.4.7 超时回调

如果消息驱动 bean 提供了超时回调方法，那么它可以注册到 EJB 计时服务中以得到基于时间事件的通知。容器在 bean 的计时器到期时调用 bean 实例的超时回调方法。参见第 18 章“计时器（Timer）服务”。

5.4.8 创建消息驱动 bean

容器按照三个步骤来创建消息驱动 bean。第一，容器调用 bean 类的 newInstance 方法来创建一个新的消息驱动 bean 实例。第二，容器注入 bean 的

MessageDrivenContext（如果申请了），然后执行由元数据注释或部署文件指定的依赖注入。第三，容器调用实例的 PostConstruct 生命周期回调方法，如果有的话。参见节 5.6.4。

兼容性提示：EJB2.1 要求消息驱动 bean 类要实现 ejbCreate 方法。这个要求已经在 EJB3.0 中被删除。如果消息驱动 bean 类实现了 ejbCreate 方法，那么它被看作是 bean 的 PostConstruct 方法，且 PostConstruct 注释符只能应用到 ejbCreate 方法上。

5.4.9 用于消息驱动 bean 的消息监听器拦截器方法

AroundInvoke 业务方法拦截器方法支持消息驱动 bean。这些拦截器方法可以定义在 bean 类中，也可以定在拦截器类中，用于处理 bean 的消息监听器方法的调用。

拦截器在第 12 章“拦截器”中描述。

5.4.10 有序化消息驱动 bean 的方法

容器有序调用每个消息驱动 bean 实例。大多数容器支持并发执行消息驱动 bean 实例；但是每个实例看起来只是一个有序的方法调用序列。因此，消息驱动 bean 不能编码成可重入的。

容器必须有序调用回调方法（例如，生命周期回调拦截器方法和超时回调方法），且这些回调必须和消息监听器方法调用处于同一序列。

5.4.11 并发处理消息

容器可以让许多消息驱动 bean 实例并发执行，这样就可以并发处理消息流。不能保证严格按照转发到消息驱动 bean 实例的顺序来执行，尽管在不影响消息的并非处理时容器应当尽量按照顺序来转发消息。因此消息驱动 bean 应当准备处理不在队列的消息：例如，将要被取消保存的消息可以在消息被保存前被转发。

5.4.12 消息驱动 bean 方法的事务上下文

Bean 的消息监听器和超时回调方法的事务范围由 bean 的元数据注释符或部署文件指定的事务属性决定。如果指定 bean 使用容器管理的事务分割，按么必须为消息监听器方法使用 REQUIRED 或 NOT_SUPPORTED 事务属性，为超时回调方法是用 REQUIRED、REQUIRES_NEW 或 NOT_SUPPORTED 事务属性。参见第 13 章“支持事务”。

当消息驱动 bean 使用 bean 管理的事务分割，用 javax.transaction.UserTransaction 接口来分割事务时，引起 bean 被调用的消息接收不是事务的一部分。如果消息收据是事务的一部分，必须使用事务属性是 REQUIRED 的容器管理事务分割。

newInstance 方法，setMessageDrivenContext，消息驱动 bean 的依赖注入方法，以及生命周期回调方法都在未指定的事务上下文中调用。参考节 13.6.5 了解容器如何用未指定事务上下文来执行方法。

5.4.13 激活配置属性

Bean 提供者可以给部署者提供消息驱动 bean 在它的操作环境中的配置信息。这可以包括消息确认模式、消息选择器、期望的目的地或终端类型等信息。

激活配置属性通过 MessageDriven 注释符的 activationConfig 或 activation-config 部署元素来指定。在部署文件中指定的激活配置属性会添加到 MessageDriven 注释符指定的激活配置属性中。如果在两个地方都指定了相同的名字，那么部署文件中的值覆盖注释符中的值。

用于 JMS 消息驱动 bean 的激活配置属性在节 5.4.14 到 5.4.16 中描述。

5.4.14 用于 JMS 消息驱动 bean 的消息确认

JMS 消息驱动 bean 不应当使用 JMS API 来进行消息确认。消息确认自动由容器处理。如果消息驱动 bean 使用容器管理的事务分割，那么消息确认自动作为事务提交的一部分被处理。如果使用 bean 管理的事务分割，消息收据不能是

bean 管理事务的一部分，在这种情况下，收据由容器确认。如果使用 bean 管理的事务分割，Bean 提供者可以声明 JMS 的 AUTO_ACKNOWLEDGE 语义或 DUPS_OK_ACKNOWLDGE 语义是否应当通过使用 MessageDriven 注释符的 activationConfig 元素或使用部署元素 activation-config-property 来应用。指定确认模式的名字是 acknowledgeMode。如果没有指定 acknowledgeMode，则使用 AUTO_ACKNOWLEDGE 语义。对于 JMS 消息驱动 bean 来说，属性 acknowledgeMode 的值必须是 Auto-acknowledge 或 Dups-ok-acknowledge。

5.4.15 用于 JMS 消息驱动 bean 的消息选择器

Bean 提供者可以声明使用 JMS 消息选择器来决定消息驱动 bean 接收哪些消息。如果 bean 提供者希望限制消息驱动 bean 接收的消息，那么 bean 提供者可以通过使用 MessageDriven 注释符的 activationConfig 元素或使用 activation-config-property 部署元素来指定消息选择器的值。用于指定消息选择器的属性名是 messageSelector。

例如：

```
@MessageDriven(activationConfig={
        @ActivationConfigProperty(
            propertyName=”messageSelector”,
            propertyValue=”JMSType = ‘car’ AND color = ‘blue’ and
                        weight> 2500”)})
```

```
<activation-config>
<activation-config-property>
<activation-config-property-name>messageSelector</activation-config-
property-name>
<activation-config-property-value>JMSType = ‘car’ AND color = ‘blue’
AND weight &gt; 2500</activation-config-property-value>
</activation-config-property>
</activation-config>
```

应用组装人员可以进一步限制但不要替换消息驱动 bean 的 messageSelector 属性的值。

5.4.16 将消息驱动 bean 和目的地或终端关联

当消息驱动 bean 被部署在容器时，它与目的地或终端建立关联。将消息驱动 bean 与目的地或终端建立关联是部署人员的责任。

5.4.16.1 JMS 消息驱动 bean

当 bean 被部署到容器时，JMS 消息驱动 bean 与 JMS 目的地（Queue 或 Topic）建立关联。将消息驱动 bean 与 Queue 或 Topic 建立关联是部署者的责任。

Bean 提供者可以通过使用 MessageDriven 注释符的 activationConfig 元素或使用 activation-config-property 部署元素告诉部署者消息驱动 bean 是与队列还是与主题建立关联。用于指定与 bean 关联的目的地类型的属性名是 destinationType。这个属性的值必须是 javax.jms.Queue 或 javax.jms.Topic。

如果消息驱动 bean 计划使用主题，那么 bean 提供者可以进一步通过使用 MessageDriven 注释符的 activationConfig 元素或使用 activation-config-property 部署元素来声明是使用永久还是非永久订阅。用于指定是永久还是非永久订阅的属性名是 subscriptionDurability。这个属性的值必须是 Durable 或 NonDurable。如果没有指定 subscriptionDurability，则使用非永久订阅。

- 永久主题订阅，和队列一样，保证消息不会丢失，即使 EJB 服务器不运行。可靠的应用通常使用队列或永久主题订阅而不使用非永久主题订阅。
- 如果使用非永久订阅，则容器负责保证消息驱动 bean 订阅是活动的（也就是，有服务于这个消息的消息驱动 bean），以保证在 EJB 服务器在运行时不会丢失消息。但是当不能得到服务于消息的 bean 时，消息可能会被错过。例如，如果 EJB 服务器关掉了一段时间就会发生这种事情。

部署者应当避免将多个消息驱动 bean 和同一个 JMS 队列关联。如果 Queue 有多个 JMS 消费者，JMS 没有定义消息如何在这些队列接收者间进行分发。

5.4.17 异常处理

消息驱动 bean 的消息监听器方法不能抛出 java.rmi.RemoteException。

消息驱动 bean 通常不应当抛出 RuntimeException。

从消息驱动 bean 类的任何方法（包括消息监听器方法和由容器调用的回调方法）中抛出的 RuntimeException 会引起 bean 的状态转变成“不存在”状态。如果消息驱动 bean 使用 bean 管理事务分割并抛出 RuntimeException，那么容器不应当确认消息。异常处理在第 14 章中有消息描述。参见节 12.4.2 了解当多个这种方法应用到 bean 类上时应用到生命周期回调拦截器方法的规则。

从客户端视角来看，消息消费者继续存在。如果客户端继续向与 bean 关联的目的地或终端发送消息，那么容器可以将消息代理给另一个实例。

某些消息类型的消息监听器方法可以抛出应用异常。应用异常被容器传递到资源适配器。

5.4.18 错失 PreDestroy 回调

Bean 提供者不能假设容器总是会调消息驱动 bean 实例的 PreDestroy 回调方法（或 ejbRemove 方法）。下面的场景会造成不会调用 PreDestroy 方法：

- EJB 容器宕机。
- 从实例的方法中向容器抛出系统异常。

如果消息驱动 bean 实例在 PostConstruct 生命周期回调方法和/或消息监听器方法中分配了资源，然后通常会在 PreDestroy 方法中释放这些资源，那么在上面的场景中这些资源将不会被自动释放。使用消息驱动 bean 的应用应当提供一些清理机制来定期清理未释放的资源。

5.4.19 回复 JMS 消息

在标准的 JMS 使用场景中，消息的 JMSReplyTo 目的地（Queue 或 Topic）的消息模式和消息被发送到的目的地的模式相同。尽管消息驱动 bean 不直接依赖它消费消息的 JMS 目的地的模式，但是它可以包含依赖 JMSReplyTo 目的地模

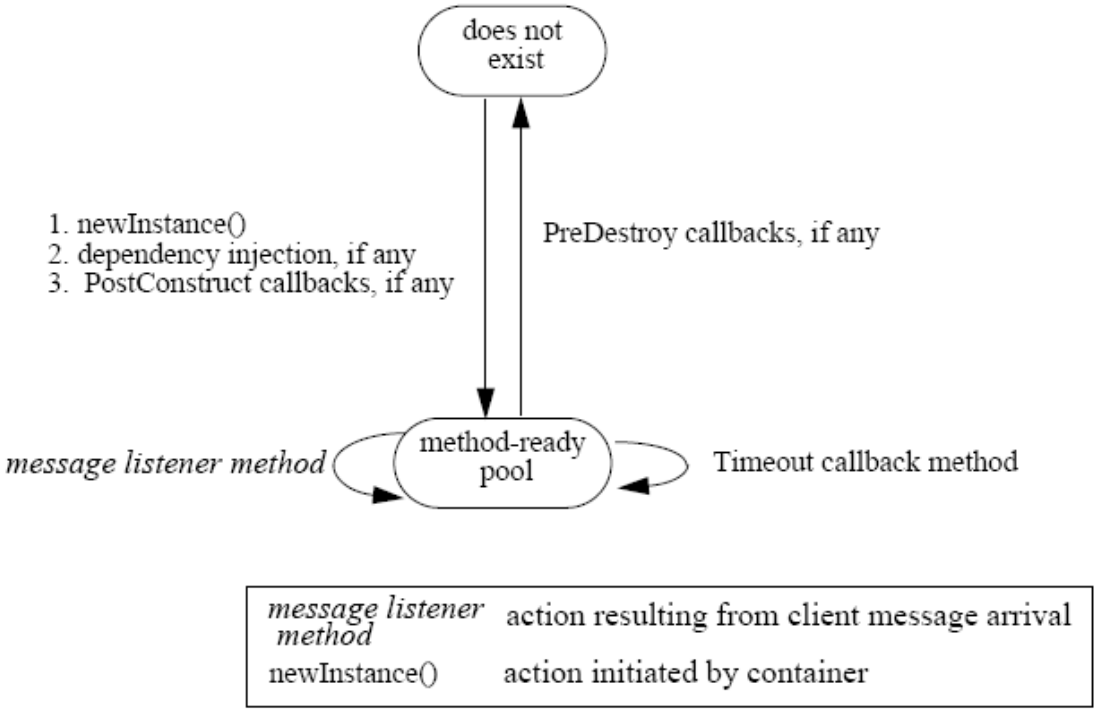
式的代码。特殊情况下，如果消息驱动 bean 回复了消息，回复的消息生产者的模式和 JMSReplyTo 目的地的模式必须相同。为了实现不依赖 JMSReplyTo 模式的消息驱动 bean，bean 提供者应当使用 instanceOf 来检测 JMSReplyTo 目的地是 Queue 还是 Topic，然后使用对应的消息生产者来进行回复。

5.5消息驱动 bean 的状态图

当客户端向目的地发送消息驱动 bean 消费的消息时，容器选择一个已准备好的实例并调用它的消息监听器方法。

下图解释了消息驱动 bean 实例的生命周期。

图 8 消息驱动 bean 的生命周期



下面的步骤描述了消息驱动 bean 实例的生命周期：

- 当容器调用消息驱动 bean 类的 newInstance 来创建一个新的实例时，消息驱动 bean 实例的生命周期开始。接着，容器注入 bean 的 MessageDrivenContext，如果申请，并执行由元数据注释或部署文件指定的其他依赖注入。容器然后调用 bean 的 PostConstruct 生命周期回调方法，如果有的话。

- 消息驱动 bean 实例现在已准备好接收发送到它关联的目的地或终端的消息，或响应来自容器对超时回调方法的调用。
- 当容器不再需要实例时（通常是当容器想减少实例池中的实例数），容器调用 PreDestroy 生命周期回调方法，如果有的话。消息驱动 bean 实例的生命结束。

5.5.1 在消息驱动 bean 类的方法中允许的操作

表 3 定义了消息驱动 bean 实例的方法，在这些方法中实例可以访问 javax.ejb.MessageDrivenContext 接口的方法，java:comp/env 环境命名上下文，资源管理器，TimerService 和 Timer 方法，EntityManager 和 EntityManagerFactory 方法，以及其他企业 bean。

如果消息驱动 bean 实例企图调用 MessageDrivenContext 接口的方法，且在表 3 中不允许这种调用，那么容器必须抛出和记录 java.lang.IllegalStateException。

如果消息驱动 bean 实例企图调用 TimerService 或 Timer 接口的方法，且在表 3 中不允许这种调用，那么容器必须抛出 java.lang.IllegalStateException。

如果 bean 实例企图访问资源管理器，企业 bean 或实体管理器/实体管理器工厂，且在表 3 中不允许这种访问，那么 EJB 架构没有定义容器的行为。

表 3 在消息驱动 bean 方法中允许的操作

Bean 方法	可以执行下列操作的 bean 方法	
	容器管理的事务分割	Bean 管理的事务分割
构造器	-	-
依赖注入方法（例如，setMessageDrivenContext）	MessageDrivenContext 方法：lookup。访问 JNDI 的 java:comp/env。	MessageDrivenContext 方法：lookup。访问 JNDI 的 java:comp/env。
PostConstruct，PreDestroy 生命周期回调方法	MessageDrivenContext 方法：getTimerService，lookup。	MessageDrivenContext 方法：getUserTransaction，

	访问 JNDI 的 java:comp/env。 访问 EntityManagerFactory。	getTimerService，lookup。 访问 JNDI 的 java:comp/env。 访问 EntityManagerFactory。
消息监听器方法，业务方法拦截器方法	MessageDrivenContext 方法：getRollbackOnly，setRollbackOnly，getCallerPrincipal，getTimerService，lookup。 访问 JNDI 的 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 服务或 Timer 方法。	MessageDrivenContext 方法：getUserTransaction，getCallerPrincipal，getTimerService，lookup。 UserTransaction 方法。 访问 JNDI 的 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 服务或 Timer 方法。
超时回调方法	MessageDrivenContext 方法：getRollbackOnly，setRollbackOnly，getCallerPrincipal，getTimerService，lookup。 访问 JNDI 的	MessageDrivenContext 方法：getUserTransaction，getCallerPrincipal，getTimerService，lookup。

	java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 服务或 Timer 方法。	UserTransaction 方法。 访问 JNDI 的 java:comp/env。 访问资源管理器。 访问企业 bean。 访问 EntityManagerFactory。 访问 EntityManager。 Timer 服务或 Timer 方法。

附加限制：

- MessageDrivenContext 接口的 getRollbackOnly 和 setRollbackOnly 方法应当只用于在事务上下文中执行的消息驱动 bean 方法。如果被调用的方法没有关联事务，则容器必须抛出 java.lang.IllegalStateException。

表 3 中不允许的操作的原因如下：

- 在方法没有有意义的事务上下文时不允许调用 getRollbackOnly 和 setRollbackOnly 方法，且所有使用 bean 管理事务分割的消息驱动 bean 都不能调用这两个方法。
- 对使用容器管理事务分割的消息驱动 bean，不能获取 UserTransaction 接口。
- 在消息驱动 bean 中不允许调用 getEJBHome 或 getEJBLocalHome 是因为消息驱动 bean 没有 EJBHome 或 EJBLocalHome。如果调用了这些方法，则容器必须抛出并记录 java.lang.IllegalStateException。

5.6Bean 提供者的责任

本节描述消息驱动 bean 提供者为保证消息驱动 bean 可以被部署到任意 EJB 容器所承担的责任。

5.6.1 类和接口

消息驱动 bean 提供者需要提供下面的类文件：

- 消息驱动 bean 类。
- 拦截器类，如果有的话。

5.6.2 消息驱动 bean 类

下面是对消息驱动 bean 类的要求：

- 类必须直接或间接地实现消息监听器接口，这个接口是它支持的消息类型或消息监听器接口的方法要求的。在 JMS 情况下，这个接口是 javax.jms.MessageListener 接口。
- 类必须定义为 public，不能是 final 的，且不能是 abstract 的。类必须是顶级类。
- 类必须有一个 public 无参的构造器。容器使用这个构造器来创建消息驱动 bean 类的实例。
- 类不能定义 finalize 方法。

可选项：

- 类可以直接或间接地实现 javax.ejb.MessageDrivenBean 接口。
- 类可以直接或间接地实现 javax.ejb.TimedObject 接口。
- 类可以实现 ejbCreate 方法。

消息驱动 bean 类可以有父类和/或父接口。如果消息驱动 bean 有父类，则消息监听器接口的方法、生命周期回调拦截器方法、超时发那个发、ejbCreate 方法和 MessageDrivenBean 接口的方法可以定义在消息驱动 bean 类或它的父类中。消息驱动 bean 类的父类不能是消息驱动 bean 类。

消息驱动 bean 类可以实现除 EJB 规范要求的方法以外的其他方法（如，有消息监听器方法内部调用的帮助方法）。

5.6.3 消息监听器方法

消息驱动 bean 类必须定义消息监听器方法。消息监听器方法必须遵循这些规则：

方法必须声明为 public。

方法不能声明为 final 或 static。

5.6.4 生命周期回调拦截器方法

可以为消息驱动 bean 定义 PostConstruct 和 PreDestroy 生命周期回调拦截器方法。如果定义了 PrePassivate 或 PostActivate 生命周期回调方法，则忽略它们（注：例如，这可能造成使用缺省的拦截器类）。

兼容性提示：如果 PostConstruct 生命周期回调拦截器方法是 ejbCreate 方法，或如果 PreDestroy 生命周期回调拦截器方法是 ejbRemove 方法，那么这些回调方法必须被 bean 类自己实现（或它的父类实现）。除了这些情况外，方法名可以是任意的，但不能以“ejb”开头以避免与 javax.ejb.EnterpriseBean 接口定义的方法冲突。

生命周期回调拦截器方法可以定义在 bean 类上和/或定义在它的拦截器类上。在节 12.4“用于生命周期事件回调的拦截器”中定义了用于定义生命周期回调拦截器方法的规则。

5.7容器提供者的责任

本节描述为支持消息驱动 bean 的容器提供者的责任。容器提供者有责任提供部署工具并在运行时管理消息驱动 bean 实例。

因为 EJB 规范没有定义部署工具和容器间的协议，因此我们假定由容器提供部署工具。可选的，部署工具可由不同的供应商提供，这个供应商使用容器供应商的特定 API。

5.7.1 实现类的生成

当消息驱动 bean 被部署时，部署工具负责生成附加的类。工具通过反射企业 bean 提供着提供的类和接口，以及检查消息驱动 bean 的部署文件来获得它生成附加类所需要的信息。

部署工具可以生成混合了容器特有代码和消息驱动 bean 类的类。例如，这个代码可以帮助容器在运行时管理 bean 实例。工具可以使用子类化、代理和代码生成来实现。

5.7.2 JMS 消息驱动 bean 的部署

容器提供者必须支持将 JMS 消息驱动 bean 作为 JMS 队列或永久订阅的消费者来部署。

5.7.3 请求/响应消息类型

如果消息监听器支持请求/响应消息类型，容器负责转发消息响应。

5.7.4 非重入实例

容器必须保证在任何时候一个实例只能有一个线程来执行。

5.7.5 事务范围，安全，异常

容器必须遵循事务范围、安全检查和异常处理的规则，正如在第 13，17 和 14 中描述的一样。

6 持久化

持久化模型和 O/R 映射在 EJB3.0 版本中已经被重新修订并被增强。

EJB3.0 定义的实体的协议和需求在文档“Java 持久化 API”【2】中描述，它

也包含了 Java 持久化查询语言和 O/R 映射元数据的所有规范。

规范要求实现文档“Java 持久化 API”中指定的持久化，查询语言和 O/R 映射的协议。

第 7，8 和 10 章分别讨论了 EJB2.1 的实体 bean 的客户端视图，用于使用容器管理持久化的 EJB2.1 实体 bean 的协议，和用于使用 Bean 管理持久化的 EJB2.1 实体 bean 的协议。使用早期的 API 需要 EJB3.0 实现来支持。

7 EJB2.1 实体 bean 的客户端视图

略。

8 容器管理持久化的 EJB2.1 实体 bean 组件规约

略。

9 EJB QL：容器管理持久化的 EJB2.1 查询语言

略。

10 Bean 管理持久化的 EJB2.1 实体 bean 组件规约

略。

11 容器管理持久化的 EJB1.1 实体 bean 组件规约

略。

12 拦截器

拦截器是一个拦截业务方法调用和生命周期事件的方法。

12.1 概述

拦截器方法可以定义在企业 bean 的 class 上，也可以定义一个拦截器类，并在企业 bean 中引入。拦截器类（与 bean 的 class 不同）的方法用于响应对 bean 方法的调用和/或者生命周期事件。

一个 bean 上可以定义多个拦截器。

对一个单一业务方法的调用，InvocationContext 对象的上下文数据可以穿越多个拦截器方法调用。

拦截器类必须有一个 public 的无参构造器。

对企业 bean 的组件的编程限制同样适用于拦截器。参看章节 21.1.2

通过注释符或部署文件可以为 bean 定义拦截器方法和拦截器类。当使用注释符时，在 bean 本身和/或的业务方法上使用 Interceptors 来指定一到多个拦截器。如果定义多个拦截器，它们的执行顺序按照它们定义的顺序，参看 12.3.1 和 12.4.1 章节。部署文件用于覆盖和改变拦截器的执行顺序。

缺省的拦截器可以定义在 ejb-jar 文件级别上，这样，在 jar 文件的所有组件都会应用这个拦截器。参看 12.6 章节。

12.2 拦截器的生命周期

拦截器实例的生命周期和它关联的 bean 实例的生命周期一致。当创建 bean 时，会创建这个 bean 上所有拦截器类的实例。当 bean 实例被清除时，会销毁这些拦截器实例。如果拦截器和有状态会话 bean 关联，则当 bean 被钝化时会钝化拦截器实例，当 bean 实例被激活时会激活拦截器实例。参看 4.4，4.5.1 和 5.5 章节。

拦截器实例和 bean 实例在 PostConstruct 或 PostActivate 方法被调用之前被创建或被激活。在 bean 实例或拦截器实例被析构或钝化之前调用 PreDestroy 和 PrePassivate 方法。

拦截器实例可以保存状态。拦截器实例可以是依赖注入的目标。当创建拦截器实例时使用它关联的 bean 的命名上下文执行依赖注入。在拦截器实例和 bean

实例上的依赖注入都完成后调用 PostConstruct 方法。

拦截器可以调用 JNDI，JDBC，JMS，其他的企业 bean 和 EntityManager。参见表 1，2，3.

Bean 拦截器和被调用的 bean 方法与生命周期方法共享 bean 的命名上下文。用于依赖注入或直接 JNDI 查找的注释符和/或 XML 部署文件元素指向这个共享的命名上下文。

EJBContext 对象可以注入到拦截器类。拦截器可以使用 EJBContext 的 lookup 方法来获取 bean 的 JNDI 命名上下文。

只支持在和有状态会话 bean 关联的拦截器上使用可扩展的持久化上下文。

12.3 业务方法拦截器

拦截器方法可以定义在会话 bean 的业务方法和消息 bean 的消息监听器方法上。业务方法拦截器通过 AroundInvoke 或 around-invoke 配置元素指定。

AroundInvoke 方法可以定义 bean 的超类或拦截器类上。但是，在这些类上只能有一个 AroundInvoke 方法。AroundInvoke 方法不能是 bean 的业务方法（即不能是 bean 的业务接口内定义的方法）。

AroundInvoke 方法可以是 public，private，protected，或包层级的方法。AroundInvoke 方法不可以声明为 final 或 static。

AroundInvoke 方法有下面的标志符：

Object <METHOD>(InvocationContext) throws Exception

AroundInvoke 方法可以调用任何业务无法方法可以调用的组件或资源。

业务方法拦截器方法可以只应用到单个的业务方法上，而不是 bean 的所有方法上。参见 12.7 章节“方法级的拦截器”。

12.3.1 多个业务方法拦截器方法

如果为 bean 定义了多个拦截器方法，则使用下面的规则来管理拦截器方法的调用顺序。部署文件可以覆盖在注释符内定义的拦截器顺序，参见 12.8 章节。

- 如果可能，首先调用缺省的拦截器方法。缺省的拦截器只能在部署文件中指定。按照在部署文件中指定的顺序执行缺省拦截器。
- 如果在 bean 的类上指定了拦截器类，则在执行 bean 本身的拦截器方法之前执行拦截器类上的拦截器方法。
- 按照在 Interceptors 注释符中的顺序执行在拦截器类内定义的 AroundInvoke 方法。
- 如果拦截器类有超类，则先执行它父类的拦截器方法。
- 在执行完拦截器类上拦截器方法后，那么按下述顺序执行：
  - 如果业务方法上定义有方法级的拦截器类，那么按照在业务方法上 Interceptors 注释符指定的顺序执行拦截器类上的 AroundInvoke 方法。（参看 12.7 章节，方法级的拦截器描述）
  - 如果 bean 有超类，那么执行超类上的 AroundInvoke 方法。
  - 执行 bean 类本身的 AroundInvoke 方法。
- 如果 AroundInvoke 方法被另一个方法重载（不过这个方法是否是自己的 AroundInvoke 方法），则不会被调用。

部署文件可以用于覆盖在注释符内指定的拦截器调用顺序。参见 12.8.2.

InvocationContext 对象提供了可以使拦截器方法控制拦截器链的元数据，包括是否调用下一个链中的拦截器方法以及它的参数和返回的值。如何使用 InvocationContext 在 12.5 章节中描述。

12.3.2 异常

业务方法拦截器方法可以抛出运行时异常或业务方法 throws 子句内允许的应用异常。

AroundInvoke 方法可以 catch 和忽略异常，并通过调用 proceed()恢复。AroundInvoke 方法可以抛出允许时异常或任何在业务方法 throws 子句内允许的可检查的异常。

AroundInvoke 方法和 bean 的业务方法运行在同一个 java 堆栈中。AroundInvoke 方法可以抛出允许时异常或任何在业务方法 throws 子句内允许的

可检查的异常，除了引起 java 堆栈崩溃的异常。异常和初始化和/或清除操作通常应当位于 proceed()的 try/catch/finally 块中。

AroundInvoke 方法可以通过抛出运行时异常或通过调用 EJBContext 的 setRollbackOnly() 方法来标记事务回滚。AroundInvoke 方法可以在 InvocationContext.proceed()方法调用之前使事务回滚。

如果拦截器链中遗漏了未处理的系统异常，则 bean 实例和它相关的拦截器实例都会被销毁。在这种情况下不会调用 PreDestroy 方法：当拦截器链断裂时，在拦截器链中的拦截器方法应当执行必需的清除操作。

12.4 生命周期事件回调的拦截器

生命周期回调拦截器方法可以在会话 bean 和消息驱动 bean 上定义。

生命周期事件回调的拦截器可以定义在拦截器类和/或直接定义在 bean 的类上。PostConstruct，PreDestroy，PostActivate 和 PrePassivate 注释符用于定义生命周期回调事件的拦截器方法。如果使用部署文件，则使用 post-construct，pre-destroy，post-activate 和 pre-passivate 元素。

生命周期回调拦截器方法和业务方法拦截器方法可以定义在同一个拦截器类上。

生命周期回调拦截器方法是否在事务和安全上下文中被调用，没有明确定义。

生命周期回调拦截器方法可以定义在 bean 的超类或拦截器的超类上。但是，这个类对同一个生命周期事件只能有一个生命周期回调拦截器方法。在类上可以指定部分或全部生命周期回调注释符。

单个生命周期回调拦截器方法可以用于多个回调事件（例如，PostConstruct 和 PostActivate）。

定义在拦截器类上的生命周期回调拦截器方法有下面的标志符：

void <METHOD> (InvocationContext)

定义在 bean 类上的生命周期回调拦截器方法有一下标志符：

void <METHOD>()

生命周期回调拦截器方法可以有 public，private，protected，或包级可见符。生命周期回调拦截器方法不应当声明为 final 或 static 的。

例子：

```
@Stateful public class ShoppingCartBean implements ShoppingCart {
   private float total;
   private Vector productCodes;
   public int someShoppingMethod(){...};
   ...
   @PreDestroy void endShoppingCart() {...};
 }
public class MyInterceptor {
...
@PostConstruct
public void any-method-name (InvocationContext ctx) {
...
ctx.proceed();
...
}
@PreDestroy
public void any-other-method-name (InvocationContext ctx) {
...
ctx.proceed();
...
}
}
```

12.4.1 一个生命周期回调事件上有多个回调拦截器方法

如果一个生命周期回调事件上定义了多个回调拦截器方法，那么调用的顺序

遵循下面的规则。部署文件可以覆盖在注释符内指定的调用顺序。参见 12.8 章节。

- 如果可能，首先调用缺省的拦截器方法。缺省的拦截器只能在部署文件中指定。按照在部署文件中指定的顺序执行缺省拦截器。
- 如果在 bean 的类上指定了拦截器类，则在执行 bean 本身的生命周期回调拦截器方法之前执行 bean 类上的拦截器类中定义的生命周期回调拦截器。
- 按照在 Interceptors 注释符中的顺序执行在生命周期回调拦截器类内定义的生命周期回调方法。
- 如果拦截器类有超类，则先执行父类的生命周期回调拦截器方法，然后在执行自身的生命周期回调拦截器方法。
- 在执行完拦截器类上拦截器方法后，那么按下述顺序执行：
  - 如果 bean 有超类，那么执行超类上的生命周期回调拦截器方法。
  - 执行 bean 类本身的生命周期回调拦截器方法。
- 如果生命周期回调方法被另一个方法重载（不过这个方法是否是自己的生命周期回调方法），则不会被调用。

部署文件用于覆盖在注释符中指定的拦截器调用顺序。参见 12.8.2 章节。

对给定生命周期事件的所有回调拦截器方法运行在同一个 java 堆栈中。如果在 bean 类（或它的超类）上没有对应的回调方法，那么在拦截器链中的最后一个拦截器方法法内执行 InvocationContext.proceed()。

InvocationContext 提供了拦截方法用于控制拦截器链下一个方法调用的元数据。参见 12.5

12.4.2 异常

生命周期回调拦截器方法可以抛出系统异常，不是应用异常。

由生命周期回调拦截器方法抛出的运行时异常在拦截器链断裂后会引起 bean 实例和拦截器实例被销毁。

生命周期回调拦截器方法和 bean 上的生命周期回调方法运行在同一个 java

堆栈中。InvocationContext.proceed()抛出和另外一个生命周期回调拦截器方法或 bean 类上的生命周期回调方法相同的异常，除非拦截器使得 java 堆栈崩溃而抛出其他的异常。生命周期回调拦截器方法（而不是 bean 上或它超类上的方法）可以 catch 由另外一个生命周期回调拦截器方法抛出的异常，并且在返回之前清除它。通常情况下，应在在 proceed()方法的 try/catch/finally 块中处理异常和初始化和/或清除操作。

当 bean 和拦截器由于这些异常被销毁时，不会调用 PreDestroy 方法：在拦截器链中的生命周期回调拦截器方法应该在拦截器链断裂时执行必需的清除操作。

12.5 InvocationContext

InvocationContext 对象提供了可以使拦截器方法控制调用链行为的元数据。

```
public interface InvocationContext {
public Object getTarget();
public Method getMethod();
public Object[] getParameters();
public void setParameters(Object[] params);
public java.util.Map<String, Object> getContextData();
public Object proceed() throws Exception;
}
```

同一个 InvocationContext 实例可以传递到业务方法或生命周期事件的每一个拦截器方法中。这样，拦截器就可以将信息保存到 InvocationContext 的上下文数据属性中，然后可以被后续的拦截器获取，这样就可以做到在多个拦截器之间传递数据。这些数据不会在多个业务方法调用或生命周期回调事件间共享。如果拦截器作为对 web 服务终端调用的结果，那么由 getContextData 返回的 map 将是 JAX-WS MessageContext。因此，InvocationContext 实例的生命周期没有明确指定。

getTarget 方法返回 bean 的实例。getMethod 返回拦截器将要调用的方法。对

应 AroundInvoke 方法，那就是 bean 的业务方法；对于生命周期回调拦截器方法，getMethod 返回 null。

proceed 方法会调用拦截器链中的下一个拦截器，如果是最后一个拦截器，则会调用业务方法。拦截器方法必须总是调用 InvocationContext.proceed()或者不调用后续的拦截器方法、bean 的业务方法或生命周期回调方法。Proceed 方法返回先一个方法调用的结果。如果方法返回 void，那么 proceed 返回 null。对于生命周期回调拦截器方法，如果在 bean 类上没有定义回调方法，那么必须在最后一个拦截器方法内调用 proceed 方法，并且返回 null。如果有多个拦截器方法，则 proceed 方法会让容器按照顺序调用它们的方法。

12.6 缺省拦截器

缺省拦截器应用于 ejb-jar 包内的所有组件上。部署文件用于定义缺省拦截器以及它们的顺序。参见 12.8.2。

缺省拦截器自动应用到 ejb-jar 内的所有组件上。ExcludeDefaultInterceptors 注释符或 exclude-default-interceptors 配置符用于排除缺省拦截器的调用。

在 bean 的拦截器被调用之前调用缺省拦截器。Interceptor-order 配置描述元素用于指定可选的顺序。参见 12.8.2 章节。

12.7 方法级的拦截器

业务方法上可以定义业务方法拦截器方法，但不是所有的业务方法（注：方法级的拦截器用于指定业务方法拦截器方法。如果用作方法级拦截器的拦截器类顶一个了生命周期回调拦截器方法，那么这些生命周期回调拦截器方法不会被调用。参见 12.4.1）。

通过在方法上使用 Interceptors 注释符或者通过使用 interceptor-bingding 配置描述元素，可以定义特定的业务方法拦截器。如果一个业务方法上定义了多个方法级的拦截器，那么按照指定的顺序调用拦截器。按照在章节 12.3.1 中的描述调用方法级的业务方法拦截器以及缺省拦截器和定义在 bean 类（以及它的超类）

上的拦截器。配置描述文件可以用于覆盖调用顺序。

同一个拦截器可以应用到 bean 类上的多个方法上：

```
@Stateless
public class MyBean ... {
public void notIntercepted() {}
@Interceptors(org.acme.MyInterceptor.class)
public void someMethod() {
}
@Interceptors(org.acme.MyInterceptor.class)
public void anotherMethod() {
}
}
```

在一个业务方法上使用多个方法级的拦截器不影响拦截器实例和 bean 类之间的关系——对每个拦截器类，每个 bean 实例只会创建一个拦截器实例。

当 ExcludeDefaultInterceptors 注释符或 exclude-default-interceptors 配置描述元素应用到业务方法时，用于排除缺省的拦截器调用。相似地，ExcludeClassInterceptors 注释符或 exclude-class-interceptors 配置描述元素用于排除类级别的拦截器调用（注：类级别的拦截器指由 Interceptors 注释定义在 bean 类上的拦截器（或者由部署文件指定））。

在下面的例子中，如果没有缺省的拦截器，当调用 someMethod 时，只会调用 MyInterceptor。

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
...
@Interceptors(org.acme.MyInterceptor.class)
@ExcludeClassInterceptors
public void someMethod() {
```

```
}
}
```

如果 bean 类上也定义了缺省拦截器，可以通过使用 ExcludeDefaultInterceptors 注释符在特定的方法上排除它。

```
@Stateless
@Interceptors(org.acme.AnotherInterceptor.class)
public class MyBean ... {
...
@ExcludeDefaultInterceptors
@ExcludeClassInterceptors
@Interceptors(org.acme.MyInterceptor.class)
public void someMethod() {
}
}
```

12.8 部署文件中的拦截器规范

部署文件是注释符的另外一个指定拦截器和 EJB 的方式，也可以用于覆盖在注释符中指定的拦截器的顺序。

12.8.1 拦截器规范

配置元素 interceptor 用于指定拦截器类的拦截器方法。拦截器方法用 around-invoke，pre-contsturct，pre-destroy，pre-passivate 和 post-activate 元素来指定。

不管是单独使用部署文件定义拦截器，还是注释符和部署文件一块来定义拦截器，拦截器类最多能有一个 around-invoke 方法，pre-construct 方法，post-destroy 方法，per-passivate 方法或 post-activate 方法。

12.8.2 拦截器绑定到 bean 的规范

元素 interceptor-binding 用于指定拦截器类与企业 bean 和它们的业务方法的绑定。下面是 interceptor-binding 子元素：

- 元素 ejb-name 必须是企业 bean 的名字或者是通配符“*”（它用于定义那些绑定到 ejb-jar 包内的所有 bean 的拦截器）
- 元素 interceptor-calss 指定拦截器类。元素 interceptor-order 是为给定的层级及其更高层级指定拦截器总顺序的可选的替代方法。
- 元素 method-name 指定方法级拦截器方法名称；可选的 method-params 元素用于在多个同名方法中定位一个方法。

用通配符“*”绑定到所有类的拦截器都是缺省拦截器。另外，拦截器可以被绑定到 bean 的 class（类级别的拦截器）或类的业务方法（方法级别的拦截器）上。

拦截器到类的绑定是附加的。如果拦截器被绑定到类级别或/和缺省级别和方法级别，那么将使用类级别和/或缺省级别和方法级别。部署文件可以用于增强通过注释符定义的拦截器和拦截器方法。当部署文件用于增强在注释符内指定的拦截器时，将会根据在 12.3.1 和 12.4.1 章节中指定的顺序调用部署文件内的拦截器方法。元素 interceptor-order 也可以覆盖在注释符内定义的顺序。

元素 exclude-default-interceptors 排除掉指定层级和它以下层级的缺省拦截器。也就是说，当在类层级上使用 exclude-default-interceptors 时，将会排除掉这个类上所有方法的缺省拦截器。当在方法层级上使用 exclude-default-interceptors 时，只会排除掉这个方法上的缺省拦截器。显式地在较低层级上列出需排除的上层拦截器将不会在这个层级及其一下层级上调用这些列出的拦截器。

通过使用 interceptor-order 元素指定类层级和/或方法层级上的拦截器顺序，可以覆盖在章节 12.3.1 和 12.4.1 中指定的拦截器顺序。如果使用 interceptor-order 元素，那么在给定层级上指定的顺序必须是所有在这个层级及其以上层级上的拦截器的顺序（除非那些用上面描述的 exclude-元素显式地被排除的拦截器）。（注：如果拦截器顺序不完备，则配置员必须保证满足这些条件）。

拦截器元素语法有四种可能的风格：

风格 1：

<interceptor-binding>

<ejb-name>*</ejb-name>

<interceptor-class>INTERCEPTOR</interceptor-class>

</interceptor-binding>

用通配符“*”指定 ejb-name 元素来指派缺省拦截器（这些拦截器应用到 ejb-jar 包内的所有企业 bean）。

风格 2：

<interceptor-binding>

<ejb-name>EJBNAME</ejb-name>

<interceptor-class>INTERCEPTOR</interceptor-class>

</interceptor-binding>

这个风格用于指定与特定企业 bean 类相关的拦截器（类级别拦截器）。

风格 3：

<interceptor-binding>

<ejb-name>EJBNAME</ejb-name>

<interceptor-class>INTERCEPTOR</interceptor-class>

<method-name>METHOD</method-name>

</interceptor-binding>

这个风格用于为特定 bean 的特定方法指定关联的方法级的拦截器。如果多个方法具有相同的名字，那么这个风格的元素指具有相同名字的所有方法。方法级的拦截器只能和 bean 的业务方法相关联。注意：在方法级的拦截器不能使用通配符“*”。

风格 4：

<interceptor-binding>

<ejb-name>EJBNAME</ejb-name>

<interceptor-class>INTERCEPTOR</interceptor-class>

<method-name>METHOD</method-name>

<method-params>

<method-param>PARAM-1</method-param>

<method-param>PARAM-2</method-param>

...

<method-param>PARAM-n</method-param>

</method-params>

<interceptor-binding>

这个风格用于指定与特定企业 bean 的特定方法关联的方法级拦截器。这个风格用于在许多重名方法中指定其中的一个方法。PARAM-1 到 PARAM-n 的值事方法入参的 java 类型的全称（如果方法没有入参，那么 method-params 不包含 method-param 元素）。数组是通过数组元素类型来指定，元素后加上一到多个“[]”（例如，int[][]）。

如果对同一个企业 bean 同时使用风格 3 和风格 4 来定义方法级的拦截器，那么这些方法级拦截器的顺序没有明确规定。

12.8.2.1 例子

下面是使用 interceptor-binding 语法的例子。

风格 1：下面的拦截器作为缺省拦截器应用到 ejb-jar 中的所有组件上。他们将按照指定的顺序被调用。

<interceptor-binding>

<ejb-name>*</ejb-name>

<interceptor-class>org.acme.MyDefaultIC</interceptor-class>

<interceptor-class>org.acme.MyDefaultIC2</interceptor-class>

</interceptor-binding>

风格 2：下面的例子是定义 EmployeeService 企业 bean 的方法级拦截器。它们会在缺省拦截器被调用后按照指定的顺序被调用。

<interceptor-binding>

<ejb-name>EmployeeService</ejb-name>

<interceptor-class>org.acme.MyIC</interceptor-class>

<interceptor-class>org.acme.MyIC2</interceptor-class>

</interceptor-binding>

风格 3：下面的拦截器应用到 EmployeeService 企业 bean 的所有 myMethod 方法上。他们会在所有缺省的拦截器和类层级的拦截器被调用后按照指定的顺序被调用。

<interceptor-binding>

<ejb-name>EmployeeService</ejb-name>

<interceptor-class>org.acme.MyIC</interceptor-class>

<interceptor-class>org.acme.MyIC2</interceptor-class>

<method-name>myMethod</method-name>

</interceptor-binding>

风格 4：下面的拦截器应用到 EmployeeService 企业 bean 的 myMethod(String firstName, String lastName)方法指定拦截器。

<interceptor-binding>

<ejb-name>EmployeeService</ejb-name>

<interceptor-class>org.acme.MyIC</interceptor-class>

<method-name>myMethod</method-name>

<method-params>

<method-param>java.lang.String</method-param>

<method-param>java.lang.String</method-param>

</method-params>

</interceptor-binding>

下面的例子用更复杂的参数类型来解释风格 3.方法 myMethod(char s, int I, int[] iar, mypackage.MyClass mycl, mypackage.MyClass[][] myclaar)指定如下：

<interceptor-binding>

<ejb-name>EmployeeService</ejb-name>

```
<interceptor-class>org.acme.MyIC</interceptor-class>
<method-name>myMethod</method-name>
<method-params>
<method-param>char</method-param>
<method-param>int</method-param>
<method-param>int[]</method-param>
<method-param>mypackage.MyClass</method-param>
<method-param>mypackage.MyClass[][]</method-param>
</method-params>
</interceptor-binding>
```

下面的例子解释用 interceptor-order 元素来指定拦截器的顺序：

```
<interceptor-binding>
    <ejb-name>EmployeeService</ejb-name>
<interceptor-order>
<interceptor-class>org.acme.MyIC</interceptor-class>
<interceptor-class>org.acme.MyDefaultIC</interceptor-class>
<interceptor-class>org.acme.MyDefaultIC2</interceptor-class>
<interceptor-class>org.acme.MyIC2</interceptor-class>
</interceptor-order>
</interceptor-binding>
```

13 支持事务

EJB 架构的关键特性之一就是对分布式事务的支持。EJB 架构允许应用开发者原子地更新分布在不同位置的多个数据库的数据。而且，这些位置上可以使用不同的 EJB 服务器。

13.1 概述

本章对事务做了简单的概述，并阐述了大量的 EJB 中的事务场景。

13.1.1 事务

事务被证明是简化应用开发的一个技术。事务使得应用开发者脱离了处理复杂的失败恢复和多用户并发问题。如果应用开发者使用事务，则开发者致力于调用的事务单元。事务系统保证这个工作单元要么完全提交，要么完全回滚。进一步说，事务使得应用开发者设计的应用好像运行在一个连续地执行工作单元的环境之中。

对事务的支持是 EJB 架构的关键特性。EJB 提供商和客户应用开发者隐藏分布式事务的复杂性。EJB 提供者可以选择由程序控制的事务分隔（称为 bean 管理的事务分隔），也可以选择由容器自动执行的声明式事务分隔（称为容器管理的事务分隔）。

对于 bean 管理的事务分隔，企业 bean 用 javax.transaction.UserTransaction 接口来分隔事务。在 UserTransaction.begin 和 UserTransaction.commit 之间调用的对所有资源的存取都属于事务的一部分。

在本章中使用的术语资源和资源管理器都是指在企业 bean 类上用 Resource 注释符或在部署文件中用 resource-ref 元素声明的资源。它不仅仅指数据库资源，也指其他资源，比如 JMS 连接。这些资源都被认为是由容器管理的资源。（注意：也包含用 Resource 注释符或 resource-ref 元素指定的但不是资源的环境条目）

对于容器管理的事务分隔，容器根据开发者在元数据注释符或在配置元素中指定的指令来分隔事务。这些指令称为事务属性，来告诉容器是否在客户事务中执行 EJB 方法，还是在新事务中执行，还是不使用事务（参考 13.6.5 章节来了解不使用事务的情况）。

不管企业 bean 使用 bean 管理的事务还是使用容器管理的事务分隔，实现事务管理是 EJB 容器和服务提供商的重任。EJB 容器和服务实现必需的底层事务协议，例如事务管理器和数据库系统/消息提供者之间的两阶段提交协议、事务上

下文传递和分布式两阶段提交。

许多应用都会有多个企业 bean，这些 bean 都使用一个资源管理器（典型地是一个关系数据库管理系统）。在不使用分布式事务时，EJB 容器可以优先使用资源管理的本地事务。资源管理器的本地事务不涉及与外部事务管理器的控制或协同。容器使用本地事务作为企业 bean 的优先事务技术对企业 bean 是不可见的，不管这些 bean 是容器管理事务分隔还是 bean 管理事务分隔的事务。对于使用资源管理器本地事务作为容器的优先事务策略的讨论，可以参考《Java Platform，Enterprise Edition(Java EE), http://jcp.org/en/jsr/detail?id=244 》和《Java2 Enterprise Edition Connector Architecture, v1.5. http://java.sun.com/j2ee/connector》。

13.1.2 事务模型

EJB 架构支持平面事务。平面事务不能有任何子事务（嵌套事务）。

注意：不支持嵌套事务的决定可以让现存的事务处理提供商和数据库管理系统都能对 EJB 提供事务支持。如果将来这些提供商对嵌套事务提供了支持，那么 EJB 可以提升到支持嵌套事务。

13.1.3 JTA 和 JTS 的关系

JTA 是事务管理器和其他涉及到分布式事务处理系统的部分之间的接口，这些部分包括：应用程序，资源管理器和应用服务器。

JTS API 是 CORBA OTS（Object Transaction Service）1.1 规范与 java 的结合。JTS 提供了在服务器之间使用标准 IIOP 协议进行事务传递的事务交互能力。JTS API 由为企业中间件实现事务处理基础设施的供应商来使用。例如，一个 EJB 服务提供商可以使用 JTS 实现作为后台的事务管理器。

EJB 架构不要求 EJB 容器支持 JTS 接口。EJB 架构要求 EJB 容器支持 JTA API 和连接器 API。

13.2 简单场景

本节描述几个阐述 EJB 架构的分布式事务能力的场景。

13.2.1 更新多个数据库

EJB 架构使得应用程序能够在一个事务中更新多个数据库中的数据。

在下图中，一个客户端调用企业 bean X，bean X 用两个数据库连接来更新数据库 A 和 B 中的数据。然后 X 调用另外一个企业 bean Y。bean Y 更新数据库 C 中的数据。EJB 服务器保证对数据库 A，B 和 C 的更新要么全部提交，要么全部回滚。

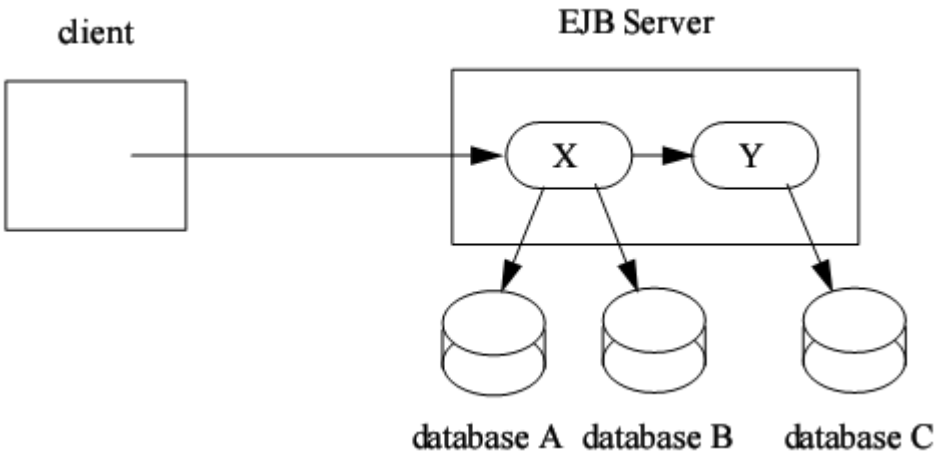


图 23 同时更新多个数据库

应用开发者不需要做任何事情来保证事务语义。在后台，EJB 服务器获取数据库连接并将其作为事务的一部分。当事务提交时，EJB 服务器和数据库系统执行两阶段提交协议以保证在三个数据库间的原子更新。

13.2.2 通过 JMS Session 发送或接收消息并更新多个数据库

EJB 架构使应用程序可以发送消息到一个或多个 JMS 目的地，或从一个或多个 JMS 目的地接收消息，并且/或在一个事务内更新多个数据库。

在下图中，客户调用企业 bean X。bean X 给 JMS 队列 A 发送一个消息并使用连接 JMS 提供商和数据库的连接来更新数据库 B 中的数据。然后 X 调用另一

个企业 bean Y。bean Y 更新数据库 C 中的数据。EJB 服务器保证在 A，B 和 C 上的操作要么是全部提交，要么是全部回滚。

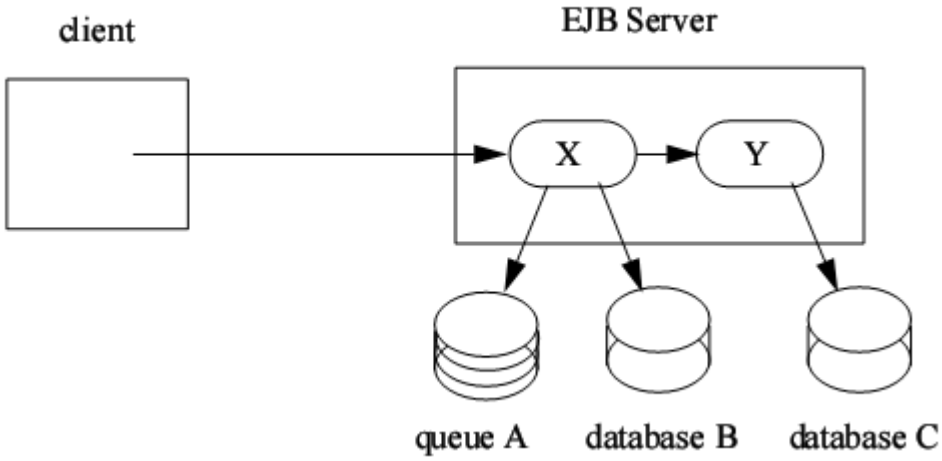


图 24 向 JMS 队列中发送数据并更新多个数据库

应用开发者不必做任何事情来保证事务语义。企业 bean X 和 Y 使用标准 JMS 和 JDBC 的 API 执行发送消息和更新数据库。在后台，EJB 服务器从 JMS 提供商连接中获取 session，从数据库连接中获取连接，并把它们作为事务的一部分。当事务提交时，EJB 服务器和消息系统以及数据库系统共同执行两阶段提交协议以保证对这三个资源的原子更新。

在下图中，客户端发送消息到 JMS 队列 A，这个队列由消息驱动 bean X 使用。Bean X 用两个数据库 B 和 C 的连接来更新数据库。EJB 服务器保证从队列中取出 JMS 消息、然后由 bean X 接收，然后更新数据库 B 和 C 都是要么全部提交，要么全部回滚。

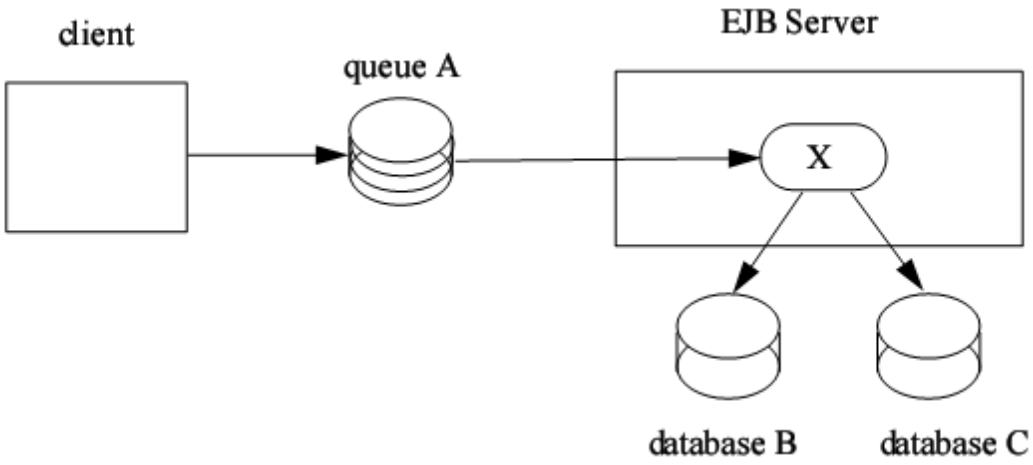


图 25 向用于消息驱动 bean 的消息队列中发送消息并更新多个数据库

13.2.3 通过多个 EJB 服务器更新数据库

EJB 架构可以在同一个事务内更新不同地点的数据。

在下图中，客户端调用企业 bean X。bean X 更新数据库 A 中的数据，然后调用另一个企业 bean Y，Y 安装在一个远程 EJB 服务器上。Y 更新数据库 B 的数据。EJB 架构可以在一个事务内更新数据库 A 和 B 中的数据。

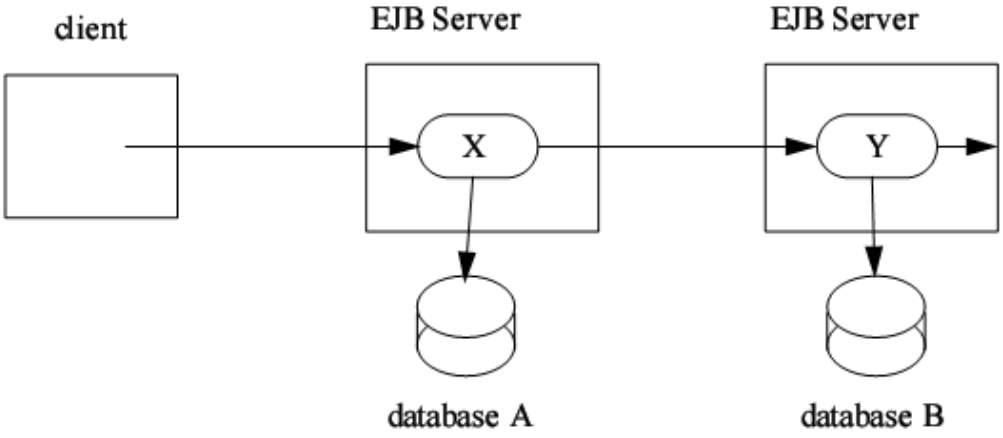


图 26 在一个事务内更新多个数据库

当 X 调用 Y 时，两个 EJB 服务器协同把事务上下文从 X 传递到 Y。事务传递对于应用层的代码来讲是透明的。

在事务提交时，两个 EJB 服务器使用分布式的两阶段提交协议（如果有分布式的两阶段提交协议的话）来保证原子更新数据库。

13.2.4 客户端管理的事务分隔

Java 客户端可以使用 javax.tansaction.UserTransaction 接口来现实地分隔事务边界。客户端程序通过依赖注入或通过在 bean 的 EJBContext 中或 JNDI 命名空间中查找的到 javax.tansaction.UserTransaction。

使用显式事务分隔的客户端程序可以通过企业 bean 对位于多个 EJB 服务器上的多数据库执行原子更新。如下图所示：

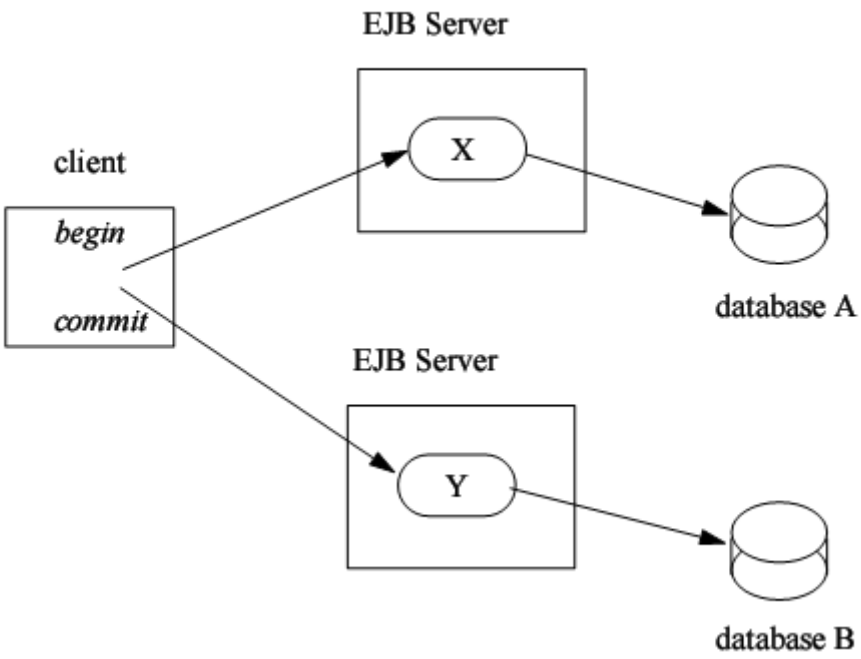


图 27 更新多个服务器上的多个数据库

应用开发者用 begin 和 commit 来分隔事务。如果企业 bean X 和 Y 配置成客户端事务（也就是说，它们的方法的事务属性是要求或者支持一个存在的事务上下文），EJB 服务器保证对数据库 A 和 B 的更新是客户端事务的一部分。

13.2.5 容器管理的事务分隔

只有客户端调用企业 bean 业务接口内的方法（或者 home 接口，或者企业 bean 的组件接口），容器拦截对这个方法的调用。拦截可以使容器能够通过事务属性声明式地控制事务分隔。（参见 13.3.7 事务属性描述）

例如，如果企业 bean 方法的事务属性是 REQUIRED，那么容器的行为如下：

如果客户端请求没有事务上下文且企业 bean 需要一个事务上下文，则容器自动初始化一个事务；如果客户端请求包含事务上下文，则容器将企业 bean 包含在客户端事务中（译者注：就是使用客户端的事务）。

下图解释了这种场景。一个非事务的客户端调用企业 bean X 并且被调用的方法的事务属性是 REQUIRED（注：在本章中使用的 TransactionAttribute 注释符的值参看事务属性。部署文件可以用于覆盖机制或注释符的替代方案，对于 EJB2.1 和 1.1 实体 bean 则必须使用部署文件，因为它们不支持注释符）。由于客户端没有事务上下文，容器在调用 X 上的方法之前开启一个新的事务。在事务上下文中执行 Bean X 的方法。当 X 调用其他企业 bean 时（在这个例子中是 Y），由这个 bena 执行的方法也包含在同一个事务中（遵循该 bean 的事务属性）。

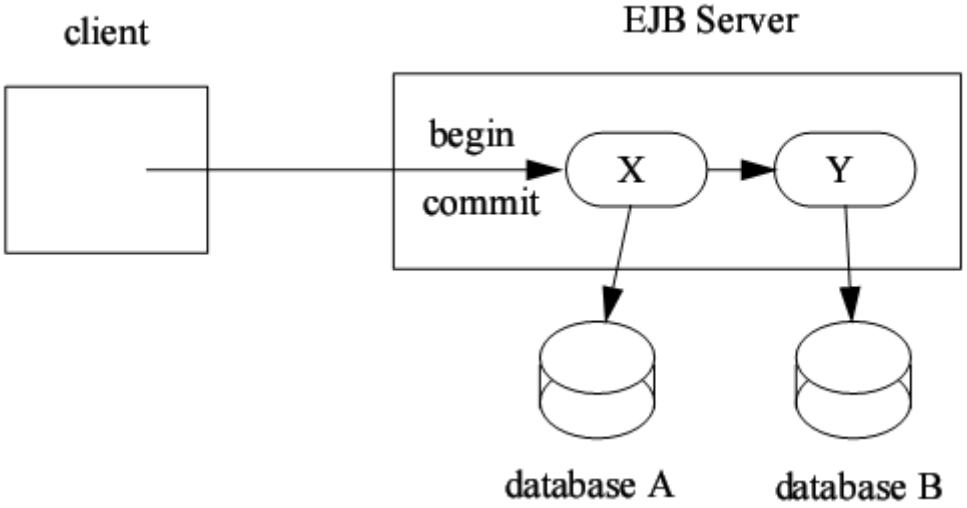


图 28 从非事务客户端更新多个数据库

容器在 X 返回时自动提交事务。

如果消息驱动 bean 的消息监听器方法配置是 REQUIREDS 事务属性，那么容器在转发消息之前自动启动一个新事务，因此也是在方法调用之前（这里使用的术语“容器”包括容器和消息提供商。当使用在*《Java2 Enterprise Edition Connector Architecture, v1.5. http://java.sun.com/j2ee/connector》中的协议时，它可以是消息提供商启动事务*）。

JMS 要求在消息从队列中取出之前启动事务。

容器自动地征用和到达消息关联的资源管理器，并且所有的资源管理器都可以被带事务的消息监听器获得。

13.3 Bean 提供者的责任

本节描述 bean 提供者眼中的事务和 bean 提供者的责任。

13.3.1 Bean 管理的事务分隔与容器管理的事务分隔的对比

当设计企业 bean 时，开发者必须决定是在业务方法内用程序来分隔事务（bean 管理的事务分隔）还是由容器基于定义（用注释符或部署文件）在业务方法上的事务属性来分隔事务（容器管理的事务分隔）。典型地，企业 bean 使用容器管理的事务分隔。这也是在不指定事务管理类型时的缺省值。

会话 bean 或消息驱动 bean 可以被设计成 bean 管理的事务分隔或者是容器管理的事务分隔。（但不能同时使用）。

EJB2.1 或 EJB1.1 实体 bean 必须使用容器管理的事务分隔。它们不能被设计成 bean 管理的事务分隔。

不能为 EJB3.0 实体指定事务管理类型。在调用者的事务上下文中执行 EJB3.0 实体。参看本规范的文档“Java 持久化 API”来了解涉及 EJB3.0 的事务。

企业 bean 实例只能在企业 bean 方法的事务内获取资源管理器，由于在这个方法内才能获得事务上下文。

13.3.1.1 无事务执行

有些 bean 可能需要获取不支持外部事务协同的资源管理器。容器不能用同样的方式来为企业 bean 管理这些事务，但它能为企业 bean 管理那些支持外部事务协同的资源管理器的事务。

如果企业 bean 需要获得不支持外部事务协同的资源管理器，那么 bean 提供者应当把 bean 设计成是容器管理的事务分隔并且它的方法或所有方法的事务属性是 NOT_SUPPORTED 的。EJB 架构不规范企业 bean 方法的事务语义。参看 13.6.5 了解容器如何来实现这种情况。

13.3.2 隔离级别

事务不仅仅是为了工作原子单元的完整性，也把工作单元相互隔离开来，这样系统就可以让多个工作单元共同执行。

隔离级别描述事务对资源管理器的获取与其他并发执行的事务对资源管理器的获取隔离的程度。

下面是管理隔离级别的指南：

- 管理隔离级别的 API 是资源管理器特有的。（因此，EJB 架构不定义管理隔离级别的 API）
- 如果企业 bean 使用多个资源管理器，bean 提供者可以为每个资源管理器指定相同或不同的隔离级别。这就是说，例如如果一个企业 bean 在事务中获取多个资源管理器，对每一个资源的获取可能有不同的隔离级别。
- Bean 提供者必须小心设置隔离级别。大多数资源管理器要求所有对资源管理器的获取需在一个事务内，且是相同的隔离级别。在事务中途改变隔离级别可能引起不可预料的行为，例如隐式的同步点（提交之前所有发生的改变）
- 对于使用 bean 管理事务分隔的会话 bean 和消息驱动 bean，bean 提供者可以通过使用资源管理器特定的 API 在企业 bean 的方法内手动指定隔离级别。例如 bean 提供者可以使用 java.sql.Connection.setTransactionIsolation 方法来设置合适数据库存取的隔离级别。
- 容器提供者应当保证提供合适的隔离级别来保证 EJB2.1 和 2.0 实体 bean 的数据一致性。典型地，就是要求高程度隔离的应用可以获得与重复读或可序列化隔离级别等价的隔离级别。
- 对于 EJB2.1 或更早的由容器管理持久化的实体 bean，事务隔离由通过容器提供者工具生成的数据存取类来管理。这些工具必须保证由数据存取类管理的隔离级别不能与在事务内的资源管理器要求的隔离级别冲突。

- 如果多个企业 bean 在同一个事务内存取同一个资源管理器也要多加注意。必须避免要求的隔离级别间的冲突。

13.3.3 使用 Bean 管理事务分隔的企业 bean

本节描述对使用 bean 管理事务分隔的企业 bean 的 bean 提供者的责任。

使用 bean 管理事务分隔的企业 bean 必须是一个会话 bean 或者是消息驱动 bean。

在启动一个新事务前，必须完成前一个事务。

Bean 提供者使用 UserTransaction 接口来分隔事务。在 UserTransacton.begin 和 UserTransaction.commit 方法之间所做的改变都会在一个事务内更新到资源管理器。当实例处于事务中时，实例不能使用资源管理器特定的事务分隔 API（例如，不能调用 java.sql.Connection 接口或 javax.jms.Session 接口上的 commit 或 rollback 方法）。（注：然而，可以使用 java 持久化实体事务接口的 API。）

有状态会话 bean 实例可以但不要求在业务方法返回之前提交事务。如果事务在业务方法最后没有被完成，那么容器在多个客户端之间保留事务和 bean 实例之间的关系直到事务最终被完成。

无状态会话 bean 实例必须在业务方法或超时回调方法返回之前提交事务。

消息驱动 bean 实例必须在消息监听器方法或超时回调方法返回之前提交事务。

下面的例子解释了一个业务方法内的事务涉及两个数据库连接。

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource javax.transaction.UserTransaction ut;
@Resource javax.sql.DataSource database1;
@Resource javax.sql.DataSource database2;
public void someMethod(...) {
java.sql.Connection con1;
```

```
java.sql.Connection con2;
java.sql.Statement stmt1;
java.sql.Statement stmt2;
// obtain con1 object and set it up for transactions
con1 = database1.getConnection();
stmt1 = con1.createStatement();
// obtain con2 object and set it up for transactions
con2 = database2.getConnection();
stmt2 = con2.createStatement();
//
// Now do a transaction that involves con1 and con2.
//
// start the transaction
ut.begin();
// Do some updates to both con1 and con2. The container
// automatically enlists con1 and con2 with the transaction.
stmt1.executeQuery(...);
stmt1.executeUpdate(...);
stmt2.executeQuery(...);
stmt2.executeUpdate(...);
stmt1.executeUpdate(...);
stmt2.executeUpdate(...);
// commit the transaction
ut.commit();
// release connections
stmt1.close();
stmt2.close();
con1.close();
```

```
con2.close();
}
...
}
```

下面的例子解释了一个业务方法内的事务既涉及数据库连接又涉及 JMS 连接。

```
@Stateless
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource javax.Transaction.UserTransaction ut;
@Resource javax.sql.DataSource database1;
@Resource javax.jms.QueueConnectionFactory qcf1;
@Resource javax.jms.Queue queue1;
public void someMethod(...) {
java.sql.Connection dcon;
java.sql.Statement stmt;
javax.jms.QueueConnection qcon;
javax.jms.QueueSession qsession;
javax.jms.QueueSender qsender;
javax.jms.Message message;
// obtain db conn object and set it up for transactions
dcon = database1.getConnection();
stmt = dcon.createStatement();
//obtainjmsconnobjectandsetupsessionfortransactions
qcon = qcf1.createQueueConnection();
qsession = qcon.createQueueSession(true,0);
qsender = qsession.createSender(queue1);
message = qsession.createTextMessage();
```

```
message.setText( some message
//
// Now do a transaction that involves the two connections.
//
// start the transaction
ut.begin();
// Do database updates and send message. The container
// automatically enlists dcon and qsession with the
// transaction.
stmt.executeQuery(...);
stmt.executeUpdate(...);
stmt.executeUpdate(...);
qsender.send(message);
// commit the transaction
ut.commit();
// release connections
stmt.close();
qsender.close();
qsession.close();
dcon.close();
qcon.close();
}
...
}
```

下面的例子解释一个有状态会话 bean 保留事务按照 method1，method2，method3 的调用顺序跨越三个客户端调用。（注意，在这里 bean 提供者必须使用 pre-passivation 回调方法来关闭连接和设置连接变量为 null。）

```
@Stateful
```

```
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource javax.Transaction.UserTransaction ut;
@Resource javax.sql.DataSource database1;
@Resource javax.sql.DataSource database2;
java.sql.Connection con1;
java.sql.Connection con2;
public void method1(...) {
java.sql.Statement stmt;
// start a transaction
ut.begin();
// make some updates on con1
con1 = database1.getConnection();
stmt = con1.createStatement();
stmt.executeUpdate(...);
stmt.executeUpdate(...);
//
// The container retains the transaction associated with the
// instance to the next client call (which is method2(...)).
}
public void method2(...) {
java.sql.Statement stmt;
con2 = database2.getConnection();
stmt = con2.createStatement();
stmt.executeUpdate(...);
stmt.executeUpdate(...);
// The container retains the transaction associated with the
// instance to the next client call (which is method3(...)).
```

```
}
public void method3(...) {
java.sql.Statement stmt;
// make some more updates on con1 and con2
stmt = con1.createStatement();
stmt.executeUpdate(...);
stmt = con2.createStatement();
stmt.executeUpdate(...);
// commit the transaction
ut.commit();
// release connections
stmt.close();
con1.close();
con2.close();
}
...
}
```

企业 bean 也可以在每个业务方法内打开和关闭数据库连接（而不是保持连接直到事务结束）。在下面的例子中，如果客户端执行一串方法（method1，method2，method3），那么在同一个事务内执行 method2 中对数据库的所有更新，这个事务是在 method1 中启动，在 method3 中提交。

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource javax.Transaction.UserTransaction ut;
@Resource javax.sql.DataSource database1;
public void method1(...) {
// start a transaction
```

```
ut.begin();
}
public void method2(...) {
java.sql.Connection con;
java.sql.Statement stmt;
// open connection
con = database1.getConnection();
// make some updates on con
stmt = con.createStatement();
stmt.executeUpdate(...);
stmt.executeUpdate(...);
// close the connection
stmt.close();
con.close();
}
public void method3(...) {
// commit the transaction
ut.commit();
}
...
}
```


13.3.3.1 getRollbackOnly 和 setRollbackOnly 方法

bean 管理事务分隔的企业 bean 不需要使用 EJBContext 的 getRollbackOnly 和 setRollbackOnly 方法。原因如下：

- bean 管理事务分隔的企业 bean 可以通过 javax.transaction.UerTransaction 的 getStatus 方法获得事务的状态。
- bean 管理事务分隔的企业 bean 可以通过 javax.transaction.UerTransaction

的 rollBack 方法回滚事务。

13.3.4 使用容器管理事务分隔的企业 bean

本节描述对使用容器管理事务分隔的 bean 提供者的要求。

企业 bean 的业务方法、消息监听器方法、业务方法拦截器方法、生命周期回调拦截器方法或超时回调方法不要使用资源管理器的特殊事务管理方法，它们会影响容器的事务边界分隔。例如，企业 bean 不可以使用 java.sql.Connection 接口的以下方法：commit，setAutoCommit 和 rollback；或 javax.jms.Session 接口的以下方法：commit 和 rollback。

企业 bean 的业务方法、消息监听器方法、业务方法拦截器方法、生命周期回调拦截器方法或超时回调方法不要使用或获取 javax.transaction.UserTransaction 接口。

下面的例子是一个使用容器管理事务分隔的企业 bean 业务方法。这个业务方法使用 JDBC 连接来更新两个数据库。容器提供由事务属性指定的事务分隔。（容器管理的事务分隔缺省的事务属性是 REQUIRED。因此在这个例子中不需要显式地指定事务属性）

```
@Stateless public class MySessionBean implements MySession {
...
@TransactionAttribute(REQUIRED)
public void someMethod(...) {
java.sql.Connection con1;
java.sql.Connection con2;
java.sql.Statement stmt1;
java.sql.Statement stmt2;
// obtain con1 and con2 connection objects
con1 = ...;
con2 = ...;
stmt1 = con1.createStatement();
```

```
stmt2 = con2.createStatement();
//
// Perform some updates on con1 and con2. The container
// automatically enlists con1 and con2 with the container-
// managed transaction.
//
stmt1.executeQuery(...);
stmt1.executeUpdate(...);
stmt2.executeQuery(...);
stmt2.executeUpdate(...);
stmt1.executeUpdate(...);
stmt2.executeUpdate(...);
// release connections
con1.close();
con2.close();
}
...
}
```

13.3.4.1 Javax.ejb.SessionSynchornization 接口

使用容器管理事务分隔的有状态会话 bean 可以有选择的实现 javax.ejb.SessionSynchronization 接口。在 4.3.7 章节中对 SessionSynchronization 作了描述。

13.3.4.2 Javax.ejb.EJBContext.setRollbackOnly 方法

使用容器管理事务分隔的企业 bean 可以使用 EJBContext 的 setRollbackOnly 方法来标志事务不能提交。典型地，在抛出应用异常之前标志事务回滚来保护数

据完整性，如果抛出未知的应用异常，则容器自动回滚事务。

例如，AccountTransfer 企业 bean 从一个账户取钱然后存入另一个账户，如果取钱操作执行成功，但存钱操作失败，那么就可以标记这个事务回滚。

13.3.4.3 Javax.ejb.EJBContext.getRollbackOnly 方法

使用容器管理事务分隔的企业 bean 可以使用 EJBContext 的 getRollbackOnly 方法来判断当前事务是否已经被标记为回滚。事务可以被企业 bean 自己标记为回滚，也可以被其他企业 bean 或者可以被事务处理框架内的其他组件（在 EJB 规范范围之外）标记为回滚。

13.3.5 在事务中使用 JMS API

Bean 提供者不应当在单个事务内使用 JMS 的请求/响应范例（发送消息然后同步接受这个消息）。因为 JMS 消息在事务提交前不会转发消息到目的地，所以在同一个事务内不会收到回复。

因为容器代表 bean 管理 JMS 会话的征用，所以 createSession(Boolean transacted, int acknowledgeMode)， createQueueSession(Boolean transacted, int acknowledgeMode)和 createTopicSession(Boolean transacted, int acknowledgeMode)方法的参数将被忽略，但为 acknowledgeMode 设置值 0.

Bean 提供者不应在事务或在未指定事务上下文中使用 JMS acknowledge 模式。由容器来处理在未指定的事务上下文中的消息承认。13.6.5 章节描述了容器能够用于带有未指定事务上下文的方法调用实现的一些技术。

13.3.6 Bean 的事务管理类型规范

缺省情况下，如果没有指定事务管理类型，会话 bean 或消息驱动 bean 使用容器管理事务分隔。会话 bean 或消息驱动 bean 的 bean 提供者可以使用 TransactionManagement 注释符来声明会话 bean 或消息驱动 bean 使用 bean 管理的事务分隔还是容器管理的事务分隔。TransactionManagement 注释符的值是

CONTAINER 或 BEAN。TransactionManagement 应用到企业 bean 的类上。

可选地，bean 提供者也可以使用 transaction-type 配置元素来指定 bean 的事务管理类型。如果使用部署文件且使用 bean 管理的事务，那么必须显式指定 bean 的事务管理类型。

Bean 的事务管理类型由 bean 提供者决定。应用组装者不允许使用部署文件来覆盖 bean 的事务管理类型，而无视事务类型是否已经被 bean 提供者显式地或缺省地指定。（参见 19 章来了解关于部署文件）

13.3.7 Bean 方法上的事务属性规范

使用容器管理事务分隔的企业 bean 的 bean 提供者可以为企业 bean 的方法指定事务属性。缺省地，使用容器管理事务分隔的企业 bean 的方法的事务属性值是 REQUIRED，在这种情况下，不需要显式指定事务属性。

事务属性是和下列方法关联的值：

- Bean 的业务接口方法
- 消息驱动 bean 的消息监听器方法
- 超时回调方法
- 无状态会话 bean 的 web 服务终端方法
- 对于用 EJB2.1 写的 bean 和早期的客户端视图，会话或实体 bean 的 home 或组件接口的方法

事务属性指定当客户端调用一个方法时容器必须如何为这个方法管理事务。可以在以下方法中使用事务属性：

- 对于用于 EJB3.0 客户端视图 API 的会话 bean，指定事务属性的方法对应于 bean 业务接口的方法，业务接口直接或间接父接口的方法，和超时回调方法（如果有）。
- 对于提供 web 服务客户端实体的无状态会话 bean，事务属性用于 bean 的 web 服务终端方法，以及超时回调方法（如果有）。
- 对于消息驱动 bean，事务属性用于消息监听器接口的方法和超时回调方法（如果有）。

- 对于用于 EJB2.0 的会话 bean 和早期的客户端视图，事务属性用于组件接口的方法和它的直接或间接父接口方法，除了 javax.ejb.EJBObject 或 javax.ejb.EJBLocalObject 接口的方法；以及超时回调方法（如果有）。不需要在会话 bean 的 home 接口的方法上指定事务属性。
- 对于 EJB2.1 或早期的实体 bean，事务属性用于 bean 的组件接口和它的所有直接或间接父接口的方法，除了 getEJBHome，getEJBLocalHome，getHandle，getPrimaryKey 和 isIdentical 方法；以及 home 接口及其 home 接口的直接或间接父接口的方法，除了远程 home 接口的 getEJBMetaData 和 getHomeHandle 方法；已经超时回调方法（如果有）。（注意：对于 EJB2.1 和早期实体 bean 方法，如果事务属性不是 Reuired，则必须使用部署文件）

如果没有为使用容器管理事务分隔的企业 bean 的方法指定 TransactionAttribute 注释符，那么事务属性的缺省值就是 REQUIRED。事务属性规范的规则在 13.3.7.1 章节中定义。

Bean 提供者可以使用部署文件替代注释符来指定事务属性（或者用于覆盖事务属性注释符）。在部署文件中指定的事务属性用于覆盖或补充用注释符指定的事务属性。如果在部署文件只能够没有指定事务属性，则就假定为用注释符指定事务属性，或者在没有指定注释符的情况下，则事务属性的值就是 Required。

应用组装者可以用 bean 的部署文件来覆盖事务属性的值。部署者也可以在部署时覆盖事务属性的值。应当注意在覆盖应用的事务属性时，应用的事务结构本身是应用的语义（译者注：应该是由应用来控制的）。

企业 bean 为 TransactionAttribute 注释符定义了以下值：

- MANDATORY
- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- NEVER

与这些值对应的部署文件中的值是：

- Mandatory
- Required
- RequiresNew
- Supports
- NotSupported
- Never

在本章中，我们说 TransactionAttribute 注释符值指的是事务属性。但是也要注意，也可以使用部署文件。

参见 13.6.2 章节来了解事务属性的值是如何影响由容器执行的事务管理。

对于消息驱动 bean 的消息监听器方法（或接口），只可以使用 REQUIRED 和 NOT_SUPPORTED 事务属性值。

对于企业 bean 的超时回调方法，只可以使用 REQUIRED、REQUIRED_NEW 和 NOT_SUPPORTED 事务属性值。

如果企业 bean 实现了 java.ejb.SessionSynchronization 接口，只可以使用 REQUIRED、REQUIRED_NEW 和 MANDATORY。

上边的约束条件对保证只在一个事务中调用企业 bean 是必需的。如果在非事务中调用 bean，那么容器就不能够发送事务同步调用。

对于使用 EJB2.1 容器管理持久化的实体 bean，只可以使用 Required，RequiresNew 或 Mandatory 配置符事务属性值，这些值用于 bean 的组件接口和组件接口的直接或间接父接口的方法上，住了 getEJBHome，getEJBLocalHome，getHandle，getPrimaryKey 和 isIdentical 方法；以及 bean 的 home 接口和 home 接口的直接或间接父类的方法上，除了远程 home 接口的 getEJBMetaData 和 getHomeHandle。

Bean 提供者和应用组装者必须注意什么时候和容器管理关系的迁移一起使用 RequiresNew 事务属性。如果使用高的隔离级别，那么在新的事务上下文中进行容器管理的关系的迁移可能会引起死锁。

容器可以有选择的为使用容器管理持久化的 EJB2.1 实体 bean 的方法支持

NotSupported，Supports 和 Never 事务属性。但是，使用容器管理持久化的实体 bean 使用这些事务属性将是不可移植的。

容器可以有选择的为使用容器管理持久化的 EJB2.1 实体 bean 的方法支持 NotSupported，Supports 和 Never 事务属性是因为使用这些事务模式可能需要使用带有非事务数据池的容器管理的持久化。通常情况下，bean 提供者和应用组装者应该避免为使用容器管理持久话的实体 bean 的方法使用 NotSupported，Supports 和 Never 事务属性，因为它在并发情况下可能导致不一致的结果，或导致持久化状态和关系的不一致更新或/和部分更新。

13.3.7.1 使用事务属性元数据注释符的规范

下面是使用 java 元数据注释符的事务属性的规范。

TransactionAtrribute 注释符用于指定事务属性。事务属性的值由枚举类 TransactionAttributeType 给出：

```
public enum TransactionAttributeType {
MANDATORY,
REQUIRED,
REQUIRES_NEW,
SUPPORTS,
NOT_SUPPORTED,
NEVER
}
```

事务属性可以指定在类上，类的业务方法上，或者都可以指定。

在 bean 的类上指定 TransactionAttribute 注释符意味着这个属性值应用到类的所有业务方法上。如果没有指定事务属性，那么缺省是 REQUIRED。Bean 类上指定其他事务属性的规范和 TransactionAttribute(REQUIRED)一样。

事务属性可以指定在 bean 类的方法上来覆盖显式或隐式指定在 bean 类上的事务属性。

如果 bean 类有超类，则还应遵循下列规则：

- 在超类 S 上指定的事务属性应用到 S 的业务方法。如果在 S 上没有指定类级别的事务属性，那么等价于在 S 上指定 TransactionAttribute(REQUIRED)。
- 可以在类 S 的业务方法 M 上指定事务属性来覆盖显式或隐式地在 S 上定义的事务属性。
- 如果类 S 的方法 M 重载了 S 超类上的业务方法，那么 M 的事务属性由上述的应用到 S 上的规则来决定。

举例：

```
@TransactionAttribute(SUPPORTS)
public class SomeClass {
public void aMethod () {...}
public void bMethod () {...}
...
}
@Stateless public class ABean extends SomeClass implements A {
public void aMethod () {...}
@TransactionAttribute(REQUIRES_NEW)
public void cMethod () {...}
   ...
}
```

假定 aMehtod，bMethod，cMethod 是接口 A 的方法，那么他们的事务属性分别是 REQUIRED，SUPPORTS 和 REQUIRES_NEW。

13.3.7.2 部署文件中的事务属性规范

下面是在部署文件中的事务属性规范。（参见 19.5 章节了解部署文件完整的语法）

注意，即使没有使用注释符，为在章节 13.3.7 中列出的方法显式指定事务属性也不是必需的。如果在 EJB3.0 部署文件中没有指定一个方法的事务属性，则

缺省的事务属性就是 Required。

如果部署文件用于覆盖注释符，并且有些方法没有指定事务属性，那么在注释符（不管显式还是隐式）中指定的值将应用到那些方法上。

13.3.7.2.1 使用 container-transaction 元素

container-transaction 元素用于为业务接口方法、home 接口方法和消息监听器接口的方法以及 web 服务终端方法和超时回调方法定义事务属性。每个 container-transaction 元素由一到多个 method 元素和 trans-attribute 元素的列表组成。container-transaction 元素指定所有列出的方法被分派所指定的事务属性值。要求在一个 container-transaction 元素内的所有方法必须是同一个企业 bean 的方法。

元素 method 使用 ejb-name，method-name 和 method-params 元素来声明一到多个方法。有三种合法的风格来组合 method 元素。

风格 1：

<method>

<ejb-name>EJBNAME</ejb-name>

<method-name>*</method-name>

</method>

这种风格用于为风格 2 或风格 3 没有指定的方法的缺省事务属性值。一个企业 bean 中最多有一个使用风格 1 method 元素的 container-transaction 元素。

风格 2：

<method>

<ejb-name>EJBNAME</ejb-name>

<method-name>METHOD</method-name>

</method>

这种风格用于为 bean 的业务接口、home 接口、消息监听器、web 服务终端和超时回调方法中的特定方法定义事务属性。如果多个方法有相同的名字，则这个风格指所有的这些同名的方法。对于给定的方法名，最多有一个使用风格 2

method 元素的 container-transaction 元素。如果同一个 bean 也有一个使用风格 1 元素的 container-transaction 元素，那么优先使用风格 2 的元素。

风格 3：

<method>

<ejb-name> EJBNAME</ejb-name>

<method-name>METHOD</method-name>

<method-params>

<method-param>PARAMETER_1</method-param>

...

<method-param>PARAMETER_N</method-param>

</method-params>

</method>

这个风格用于在一堆同名的方法内指定一个特定的方法。如果同一个 bean 还有风格 1 和风格 2 元素，那么优先使用风格 3 的元素值。

可选元素 method-intf 元素用于区分在业务、组件、home 接口和/或 web 服务终端间多次定义的有相同名字和标志符号的方法。但是，如果同名方法既是本地业务接口的方法有事本地组件接口的方法，那么在这个两个方法上应用相同的事务属性。同样地，如果同名方法既是远程业务接口的方法有事远程组件接口的方法，那么在这个两个方法上应用相同的事务属性。

下面是一个在部署文件中的事务属性规范的例子。EmployeeRecord 企业 bean 的 updatePhoneNumber 方法分派的事务属性是 Mandatory；所有其他的方法分派的事务属性是 Required。企业 bean AardvarkPayroll 的所有方法分派的事务属性是 RequiresNew。

<ejb-jar>

...

<assembly-descriptor>

...

<container-transaction>

```
<method>
<ejb-name>EmployeeRecord</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
<method>
<ejb-name>EmployeeRecord</ejb-name>
<method-name>updatePhoneNumber</method-name>
</method>
<trans-attribute>Mandatory</trans-attribute>
</container-transaction>
<container-transaction>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

13.4 应用装配员的责任

本节描述应用装配员视图及其责任。

没有机制让应用装配员影响使用 bean 管理事务分隔的企业 bean。应用装配员不必为使用 bean 管理事务分隔的企业 bean 定义事务属性。

应用装配员可以使用部署文件事务属性机制来覆盖或改变使用容器管理事

务分隔的企业 bean 的事务属性。

应用装配员应该小心地改变事务属性，因为由事务属性指定的行为本质上是应用语义的一部分。

13.5 部署者的责任

部署者可以在部署时覆盖和改变事务属性的值。

部署者应该小心地改变事务属性，因为由事务属性指定的行为本质上是应用语义的一部分。

兼容提示：对于使用 EJB2.1 规范（以及早期的）写的应用，部署者有责任保证对于使用容器管理事务分隔的企业 bean，那些在部署文件中没有指定事务属性的方法都已经被分派了一个事务属性。

13.6 容器提供者的责任

本节定义了容器提供者的责任。

对会话或实体 bean 的业务方法接口（和/或 home 与组件接口）、web 服务终端的方法和消息驱动 bean 的消息监听器方法的每一个调用都被容器拦截，并且通过容器获得企业 bean 使用的对资源管理器的连接。这个受管理的环境允许容器影响企业 bean 的事务管理。

这并没有暗指容器必须拦截所有企业 bean 对资源的获取。典型地，容器只通过注册资源管理器连接工厂对象的特定容器实现拦截资源管理器工厂（例如 JDBC 数据源）的 JNDI 查找。资源管理连接工厂对象允许容器获得 javax.transaction.xa.XAResource 接口并把它传入事务管理器。在设置完成后，企业 bean 和资源管理器直接通讯而不通过容器。

13.6.1 Bean 管理的事务分隔

本节定义了容器对使用 bean 管理事务分隔的企业 bean 的事务管理的责任。

注意，只有会话和消息驱动 bean 可以用于 bean 管理的事务分隔。

容器必须按下列方式管理客户对使用 bean 管理事务分隔的企业 bean 实例的调用。当客户通过企业 bean 的客户端视图接口调用它的业务方法时，容器挂起所有可能与和客户请求相关的事务。如果实例关联有事务（如果有状态会话 bean 实例在前面的业务方法中启动了这个事务），容器就使用者事务和方法执行关联。如果有拦截器方法和 bean 实例关联，则会在拦截器方法调用之前进行关联。

容器必须可以让企业 bean 的业务方法、消息监听器方法、拦截器方法或超时回调方法通过依赖注入和通过 javax.ejb.EJBContext 接口查找，以及使用 JNDI 命名上下文 java:/UserTransaction 获得 javax.transaction.UserTransaction。当实例使用 javax.transaction.UserTransaction 来分隔事务时，容器必须在 begin 和 commit 或 rollback 之间征用实例使用的所有资源管理器。当实例提交事务时，容器负责事务提交的全局协调（通常容器依赖事务管理器，它是 EJB 服务器跨所有征用的资源管理器执行两阶段提交的一部分。如果事务只涉及一个资源管理器并且配置为可以使用连接共享，那么容器可以使用本地事务优化。）。

对于有状态 bean，可以在没有提交或回滚事务的情况下完成启动事务的业务方法。在这种情况下，容器必须在多个客户端调用之间保留事务和实例的关联直到实例提交或回滚事务。当客户端调用下一个业务方法时，容器必须在事务上下文中调用这个业务方法（以及任何应用到这个 bean 上的拦截器方法）。

如果有状态会话 bean 实例在业务方法或拦截器方法内启动一个事务，那么它必须在业务方法（或所有的拦截器方法）返回之前提交事务。容器必须检测这种情况：一个事务在业务方法或拦截器方法中被启动但是没有完成。这种情况的处理如下：

- 记录为应用错误以警告系统管理员
- 回滚启动的事务
- 丢弃会话 bean 实例
- 抛出 javax.ejb.EJBException（如果业务接口是一个继承 java.rmi.Remote 的远程业务接口，那么抛出 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图，如果客户端是远程客户端则容器应当抛出 java.rmi.RemoteException；如果客户端是本地客户端则抛出

javax.ejb.EJBException。

如果消息驱动 bean 实例在消息监听器方法或拦截器方法内启动事务，那么它必须在消息监听器方法（或所有的拦截器方法）返回之前提交事务。容器必须检测这种情况：事务在消息监听器方法或拦截器方法内被启动但没有完成，并且按照以下方式处理：

- 记录为应用错误以警告系统管理员。
- 回滚启动的事务
- 丢弃消息驱动 bean 实例

如果会话 bean 或消息驱动 bean 实例在超时回调方法内启动事务，则它必须在超时回调方法返回之前提交事务。容器必须检测这种情况：事务在超时回调方法内被启动但没有完成，并按以下方式处理：

- 记录为应用错误以警告系统管理员。
- 回滚启动的事务
- 丢弃 bean 的实例

对使用 bean 管理的事务容器执行的动作总结在下表中。T1 是一个和客户端请求关联的事务，T2 是一个当前和实例关联的事务（也就是说，是一个由前一个业务方法启动但没有完成的事务）。

表 12 对使用 bean 管理事务的 bean 方法容器的动作

客户端事务	当前和实例关联的事务	和方法关联的事务
None	None	None
T1	None	None
None	T2	T2
T1	T2	T2

对表内的每一个条目描述如下：

- 如果客户端请求没有关联事务而且实例也没有关联事务，或者如果 bean 是消息驱动 bean，容器调用未指定事务上下文的实例。
- 如果客户端请求关联了事务 T1，但是实例没有关联事务，那么容器挂起客户端关联的事务并且调用未指定事务上下文的方法。当方法（包括它

关联的拦截器方法）完成后再次唤醒客户端事务 T1。对于消息驱动 bean 或无状态 bean 的 web 服务终端方法调用来说，永远都不会发生这种情况。

- 如果客户端请求没有关联事务但实例已经关联了事务 T2，那么容器容器调用管理事务 T2 的实例。对于无状态会话 bean 或消息驱动 bean，永远都不会发生这种情况：只会发生在有状态会话 bean 上。
- 如果客户端关联了事务 T1，但实例关联事务 T2，容器挂起客户端事务然后调用使用 T2 事务上下文的实例方法。当方法（包括它的所有拦截器方法）完成后，容器重新使用客户端事务 T1. 对于无状态会话 bean 或消息驱动 bean，永远都不会发生这种情况：只会发生在有状态会话 bean 上。

容器必须允许企业 bean 实例在一个方法内按顺序执行几个事务。

在实例没有提交前一个事务时，如果实例使用 javax.transaction.UserTransaction 的方法 begin 启动事务，那么容器必须在 begin 方法内抛出 javax.transaction.NotSupportedException。

如果使用 bean 管理事务分隔的 bean 实例调用 javax.ejb.EJBContext 的 setRollbackOnly 或 getRollbackOnly 方法，那么容器必须抛出 java.lang.IllegalStateException。

13.6.2 为会话和实体 bean 使用的容器管理的事务分隔

容器有责任为声明为容器管理事务分隔的会话 bean、使用 bean 管理持久化的实体 bean 和使用容器管理持久化的 EJB2.1 以及 EJB1.1 实体 bean 提供事务分隔。对于这些企业 bean，容器必须按照在元数据注释符或部署文件中指定的事务属性来分隔事务。

以下的子章节定义了当通过企业 bean 业务接口（和/或 home 或组件接口）或者 web 服务终端的方法调用企业 bean 业务方法时，容器对企业 bean 业务方法的调用的管理责任。容器的责任依赖于事务属性的值。

13.6.2.1 NOT_SUPPORT

容器调用事务属性设置为 NOT_SUPPORT 但没有事务上下文的企业 bean。

如果客户端调用有事务上下文，那么容器在调用企业 bean 的业务方法之前挂起和当前线程关联的事务上下文。容器当业务方法完成后，容器重新启动挂起的事务。挂起的事务不会传递资源管理器或在业务方法内调用的其他企业 bean 对象。

如果业务方法调用其他企业 bean，容器不在调用中传递事务上下文。

参考 13.6.5 节更加详细的了解容器如何实现这种情况。

13.6.2.2 REQUIRED

容器必须使用有效的事务上下文调用事务属性为 REQUIRED 的企业 bean 方法。

如果客户端有事务上下文，则容器使用客户端的事务上下文调用企业 bean 的方法。

如果客户端没有事务上下文，容器在代理对企业 bean 业务方法调用之前自动启动一个新事务。容器自动通过带事务的业务方法征用所有的资源管理器。如果业务方法调用其他企业 bean，容器在调用中传递事务上下文。当业务方法完成时，容器提交事务。容器在方法返回值被返回到客户端之前执行提交协议。

13.6.2.3 SUPPORTS

容器按以下方式调用事务属性是 SUPPORTS 的企业方法。

- 如果客户端调用带有事务上下文，那么容器执行和 REQUIRED 相同的步骤。
- 如果客户端调用没有事务上下文，容器执行和 NOT_SUPPORTED 相同的步骤。

必须小心 SUPPORTS 事务属性。这是因为可能有两种不同的执行模型。只有那些可以在两种模式下都能执行的企业 bean 才能使用 SUPPORTS 事务属性。

13.6.2.4 REQUIRES_NEW

容器必须使用新的事务上下文来调用事务属性是 REQUIRES_NEW 的企业 bean 的方法。

如果调用企业 bean 方法的客户端没有事务上下文，容器在代理对企业 bean 方法调用之前自动启动一个新的事务。容器自动征集带事务的业务方法使用的资源管理器。如果业务方法调用其他企业 bean，容器在调用中传递事务上下文。当业务方法完成时，容器提交事务。容器在方法返回之前执行提交协议。

如果客户端有事务上下文，那么容器在启动新事务之前挂起当前线程中的事务，然后调用业务方法。容器在业务方法和新事务完成之后重新启动挂起的事务。

13.6.2.5 MANDATORY

容器必须在客户端的事务上下文中调用事务属性是MANDATORY 的业务方法。要求客户端有事务上下文。

- 如果客户端有事务上下文，那么容器执行和 REQUIRED 相同的步骤。
- 如果客户端没有事务上下文（如业务接口是继承 java.rmi.Remote 的远程业务接口，那么应 javax.transaction.TransactionRequiredException 抛到客户端）。如果使用 EJB2.1 客户端视图，如果是远程客户端则抛出 javax.transaction.TransactionRequiredException，如果是本地客户端则抛出 javax.ejb.TransactionRequiredLocalException。

13.6.2.6 NEVER

容器在非事务上下文中调用事务属性是 NEVER 的企业 bean 方法。客户端要求没有事务上下文。

- 如果客户端有事务上下文，容器抛出 javax.ejb.EJBException（如果业务接口是继承 java.rmi.Remote 远程业务接口，则抛出 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图，那么如果客户端是远程客户端则容器抛出 java.rmi.RemoteException，如果是本地客

户端则抛出 javax.ejb.EJBException。

- 如果客户端没有事务上下文，那么容器执行和 NOT_SUPPORTED 一样的步骤。

13.6.2.7 事务属性总结

下表对容器传递到业务方法和业务方法使用的资源管理器的事务上下文的总结。T1 是一个客户端请求带过来的事务，T2 是容器初始化的事务。

表 13 事务属性总结

事务属性	客户端事务	和业务方法关联的事务	和资源管理器关联的事务
NOT_SUPPORTED	None	None	None
	T1	None	None
REQUIRED	None	T2	T2
	T1	T1	T1
SUPPORTS	None	None	None
	T1	T1	T1
REQUIRED_NEW	None	T2	T2
	T1	T2	T2
MANDATORY	None	Error	N/A
	T1	T1	T1
NEVER	None	None	None
	T1	Error	N/A

如果企业 bean 的业务方法通过其他企业 bean 的业务接口或 home 和组件接口调用其他业务 bean，那么在列“和业务方法关联的事务”中指明的事务将会作为客户端上下文的一部分被传递到目标企业 bean。

13.6.2.8 方法 setRollbackOnly 的处理

容器必须处理按以下方式处理来自事务属性是 REQUIRED, REQUIRES_NEW 或 MANDATORY 业务方法对 EJBContext.setRollbackOnly 方法的调用：

- 容器必须保证事务从不被提交。典型地，容器向事务管理器发出标志事务回滚的指令。
- 如果容器在分派业务方法到实例之前立刻初始化事务（相反，事务继承自调用者），那么容器必须注意实例已经调用了 setRollbackOnly 方法。当业务方法调用完成时，容器必须回滚而不是提交事务。如果业务方法正常返回或有业务异常，那么容器必须在回滚执行之后将返回结果或应用异常返回到客户端。

如果从事务属性是 SUPPORTS,NOT_SUPPORTED 或 NEVER 的业务方法中调用 EJBContext.setRollbackOnlyfangfa ，那么容器必须抛出 java.lang.IllegalStateException。

13.6.2.9 方法 getRollbackOnly 的处理

容器必须处理从事务属性是 REQUIRED,REQUIRES_NEW 或 MANDATORY 业务方法中对 EJBContext.getRollbackOnly 的调用。

如果从事务属性是 REQUIRED,REQUIRES_NEW 或 MANDATORY 的业务方法中调用 EJBContext.getRollbackOnlyfangfa ，那么容器必须抛出 java.lang.IllegalStateException。

13.6.2.10 方法 getUserTransaction 的处理

如果使用容器管理事务分隔的企业 bean 实例调用 EJBContext 的 getUserTransaction 方法，那么容器必须抛出 java.lang.IllegalStateException。

13.6.2.11 Javax.ejb.SessionSynchronization 回调

如果会话 bean 实现了 javax.ejb.SessionSynchronization 接口，那么容器必须在实例上调用 afterBegin，beforeCompletion 和 afterCompletion 回调，这些调用作为事务提交协议的一部分。

容器在调用第一个业务方法之前调用 afterBegin 方法。

容器调用 beforeCompletion 给企业 bean 最后回滚事务的机会。实例可以通过调用 EJBContext.setRollbackOnly 来引起事务回滚。

容器在完成事务提交协议之后调用 afterCompletion 来通知企业 bean 实例完成事务。

13.6.3 对于消息驱动 bean 的容器管理事务分隔

容器有责任为声明为容器管理事务分割的消息驱动 bean 提供事务分隔。对于这些企业 bean，容器必须根据注释符或部署文件的指定来分隔事务。（参见 19 章了解部署文件）

下面的章节描述容器对消息驱动 bean 的消息监听器方法调用的管理。容器根据事务属性的值来进行管理。

只有 NOT_SUPPORTED 和 REQUIRED 事务属性可以应用到消息驱动 bean 的消息监听器方法上。在消息驱动 bean 的消息监听器方法上使用其他事务属性没有意义，因为没有预先存在的客户端事务上下问（REQUIRES_NEW，SUPPORTS），并且没有客户端来处理异常（MANDATORY，NEVER）。

13.6.3.1 NOT_SUPPOTED

容器用不确定的事务上下文调用事务属性是 NOT_SUPPORTED 的消息驱动 bean 的消息监听器方法。

如果消息监听器方法调用其他企业 bean，那么容器不传递事务上下文。

13.6.3.2 REQUIRED

容器必须使用有效的事务上下文调用事务属性是 REQUIRED 的消息驱动 bean 的消息监听器方法。由在事务内的消息监听器方法获得资源管理器被带着事务征用。如果消息监听器方法调用其他企业 bean，容器传递这个事务上下文。当消息监听器方法完成时，容器提交事务。

消息系统可以在考虑可靠性的服务质量和消息出列的处理上不一致。

对与 JMS 要求如下：

事务必须在 JMS 消息出列之前启动事务，也就是在消息驱动 bean 的 onMessage 方法被调用之前。和将要到达的消息关联的资源管理器被带着事务征用，以及在事务内有 onMessage 方法获取所有的资源管理器。如果 onMessage 方法调用其他企业 bean，容器传递事务上下文。当 onMessage 完成时提交事务。如果 onMessge 没有成功完成或是事务被回滚，那么应用消息重发语义。

13.6.3.3 方法 setRollbackOnly 的处理

容器必须按照以下方式处理在事务属性是 REQUIRED 的消息监听器方法中对 EJBContext.setRollbackOnly 方法的调用：

- 容器必须保证事务从不被提交。典型地，容器向事务管理器发出标志事务回滚的指令。
- 容器必须注意已经调用了 setRollbackOnly 方法的实例。当方法调用完成时，容器必须回滚事务。

如果在事务属性是 NotSupported 的消息监听器方法内调用 EJBContext.setRollbackOnly，那么容器必须抛出并记录 java.lang.IlleagalStateException。

13.6.3.4 方法 getRollbackOnly 的处理

容器必须处理在事务属性是 REQUIRED 的消息监听器方法内对 EJBContext.getRollbackOnly 方法的调用。

如果在事务属性是 NotSupported 的消息监听器方法内调用 EJBContext.getRollbackOnly，那么容器必须抛出并记录 java.lang.IlleagalStateException。

13.6.3.5 方法 getUserTransaction 的处理

如果使用容器管理事务分隔的消息驱动 bean 调用 EJBContext 的 getUserTransaction 方法，那么容器必须抛出并记录 java.lang.IlleagalStateException。

13.6.4 本地事务优化

容器可以为元数据注释为或配置为资源管理器连接是共享的（参见 16.7.1.3 “在部署文件中声明资源连接器工厂”）企业 bean 使用本地事务优化。容器使用本地资源管理器优化对应用是透明的。

容器可以为由容器初始化的容器管理事务分隔的事务和由使用 UserTransaction 接口来进行 bean 管理事务分隔的 bean 初始化的事务使用优化。容器不能为来自其他容器的事务使用优化。

本地事务优化的方法在《java 平台企业版（Java EE），v5.http://jcp.org/en/jsr/detail?id=244 》和《java2 企业连接器架构,v1.5,http://java.suan.com/j2ee/connector》中说明。

13.6.5 运行在“不明事务上下文”中的方法的处理

在 EJB 规范中的术语“不明事务上下文”指的是 EJB 架构不能完全定义企业 bean 方法执行的事务语义。

这包括以下情况：

- 事务属性是 NOT_SUPPORTED，NEVER 或 SUPPORTS 的使用容器管理事务分隔的企业 bean 的方法。
- 使用容器管理事务分隔的会话 bean 的 PostConstruct，PreDestroy，

PostActivate 或 PrePassivate 回调方法。（注：参看第 4 章“会话 bean 组件协议”）

- 使用容器管理事务分隔的消息驱动 bean 的 PostConstruct 或 PreDestroy 回调方法。（参看第 5 章，“消息驱动 bean 组件协议”）

EJB 规范没有描述容器如何管理具有不明事务上下文的方法的执行——事务语义留给容器实现。下面列出了容器可以选择的一些技术（这个列表没有包括所有可能的策略）：

- 容器可以不使用事务上下文执行这个方法并获取后台资源管理器。
- 容器可以把对资源管理器的每一次调用都看作一个单事务（例如，容器可以将 JDBC 连接设置为自动提交模式）。
- 容器可以将对资源管理器的多个调用合并成到一个单事务中。
- 容器可以将对多个资源管理的多个调用合并到一个单事务中。
- 如果实例调用其他企业 bean 的方法，并且这些被调用的方法也指派为运行在不明事务上下文中，那么容器可以将来自多个实例的对资源管理器的调用合并到一个单事务中。
- 上边的任意组合。

由于企业 bean 不知道容器实现了哪一种技术，企业 bean 必须注意不要依赖与任何特定的容器行为。

当运行在未指定事务上下文中的方法执行中途失败时，可以保留从处于不明状态的方法获取的资源管理器。EJB 架构没有定义在发生失败后应用应该如何恢复资源管理器的状态。

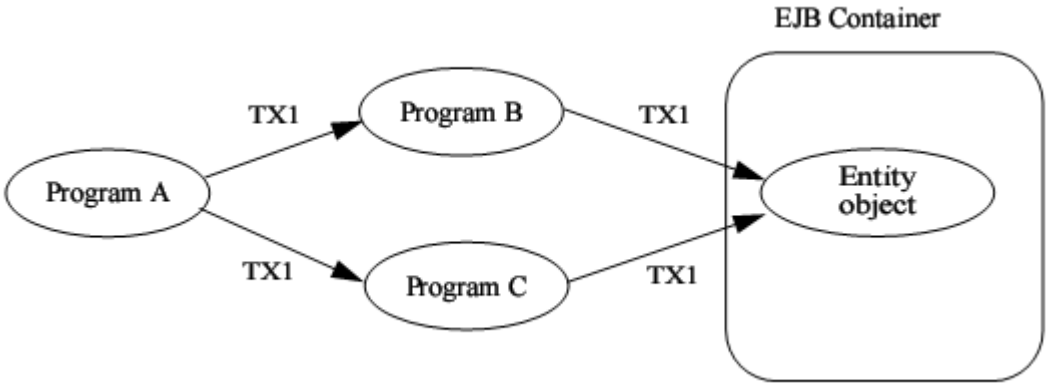
13.7 来自在同一个事务上下文中的多个客户端的存取

本节描述更加复杂的分布式事务场景，并指定了这个场景要求的容器的行为。

13.7.1 带实体对象的事务“钻石”场景

多个客户端可以在同一个事务内获取一个实体对象。例如，程序 A 可以启动一个事务，在事务上下文中调用程序 B 和程序 C，然后提交事务。如果程序 B 和 C 获取同一个实体对象，事务拓扑创建成一个钻石。

图 29 带实体对象的事务钻石场景



一个例子（没有实际意义），一个客户端程序试图在同一个事务内对两个不同的存储池执行两次购买。在每个存储池，程序处理客户端的购买请求并减少客户的银行账户。

对于一个 EJB 服务器的实现来说处理这种情况（在 EJB 服务器内程序 B 和 C 通过不同的网络路径获取实体对象）是困难的。这是因为许多 EJB 服务器将 EJB 容器实现为多进程的集合并运行在同一个或多个机器上。如果客户端 B 和 C 连接到不同的 EJB 容器进程，并且 B 和 C 需要获取处于相同事务中的同一个实体，那么问题就是容器如何能够让 B 和 C 看到处于同一事务中的实体的一致状态（钻石问题只应用到当 B 和 C 处于同一个事务的情况）。

上例解释了一个简单钻石问题。我们使用术语“钻石”指任何通过多个网络路径获取处于同一事务内的同一实体的分布式事务场景。

注意，在钻石场景中，客户端 B 和 C 按顺序获取实体对象。同时获取处于同一事务上下文中的实体对象将被认为是应用程序错误，将被容器以自己的方式处理。

注意，处理钻石问题不只存在于 EJB 架构。这个问题存在于所有的分布式

事务处理系统。

下面的子章节定义了在处理出现涉及实体对象的钻石分布式事务拓扑时各 EJB 角色的责任。

13.7.2 容器提供者的责任

这个章节定义了 EJB 容器的在涉及实体对象的钻石情况下的责任。

EJB 规范要求容器支持本地钻石。在本地钻石中，组件 A，B，C 和 D 被配置在同一个 EJB 容器。

EJB 规范不要求 EJB 容器支持分布式钻石。在分布式钻石中，多个客户端通过多个网络路径获取在同一个事务中的目标实体，并且客户端（程序 B 和 C）是和目标实体对象不在同一个容器内的企业 bean。

如果容器提供者选择不支持分布式钻石，并且如果容器能够检测到客户端调用将引起钻石，容器应当抛出 javax.ejb.EJBException（或如果使用 EJB2.1 远程客户端实体则抛出 java.rmi.RemoteException）。

13.7.3 Bean 提供者的责任

这个章节定义了 bean 提供者的在涉及实体对象的钻石情况下的责任。

钻石情况对 bean 提供者是透明的——bean 提供者不需要改变参与到钻石中的企业 bean 的代码。由容器实现钻石问题的解决方案对 bean 都是透明的，并且不改变 bean 的语义。

13.7.4 应用组装者和部署者的责任

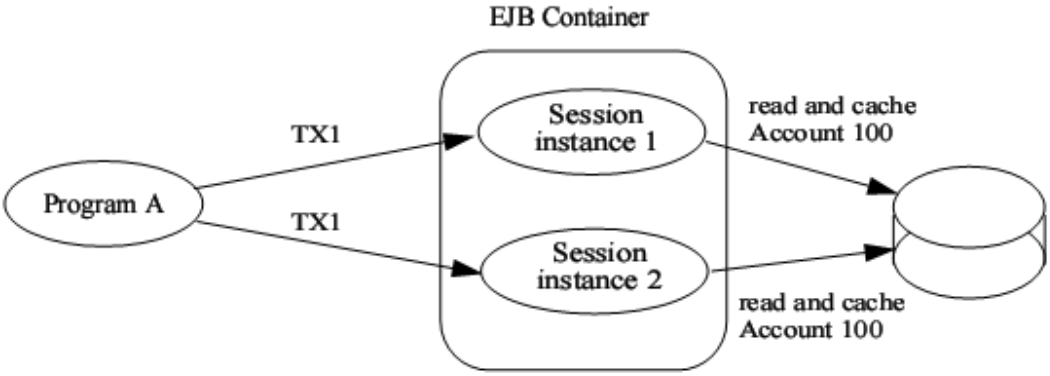
这个章节定义了应用组装者和部署者在涉及实体对象的钻石情况下的责任。

应用组装者和部署者应该知道可能发生分布式钻石。一般情况下，应用组装者应当试着去避免创建不必要的分布式钻石。

13.7.5 涉及会话对象的事务钻石

当两个客户端获取同一个会话 bean 是合法时，使用会话 bean 的应用可能遭遇钻石情况。例如，程序 A 启动一个事务然后调用两个不同的会话对象。

图 30 带会话 bean 的事务钻石场景



如果会话 bean 实例在同一个事务中跨多个方法调用缓存同一个数据项（例如，账户 100 的余额），那么程序很可能产生不正确的结果。

问题可以存在而不管两个会话对象是同一个或是不同的会话 bean。如果在事务初始化器和缓存数据的会话对象间有中介对象，那么也可能存在问题（并且可能很难发现）。

对容器提供者没有要求，因为容器不可能检测到这个问题。

Ben 提供者和应用组装者必须避免创建会引起在同一个事务内由多个会话对象缓存不一致数据的应用。

14 异常处理

14.1 概述和概念

14.1.1 应用异常

应用异常是由 Bean 提供者定义的作为应用业务逻辑的一部分的异常。在这个规范中，应用异常和系统异常是不同的。

企业 bean 的业务方法使用应用异常来将不正确的应用级条件通知给客户

端，例如不能从业务方法的入参中获取值。一般地，客户端可以从应用异常中恢复。应用异常不应当用于报告系统级的问题。

例如，Account 企业 bean 可以抛出应用异常来报告由于余额不足而不能执行借贷操作的异常。但Account 不能用应用异常来报告类似数据库连接失败的异常。

应用异常可以定义在企业 bean 业务接口、home 接口、组件接口、消息监听器接口或 web service 终点的 throws 子句中。

应用异常可以是 java.lang.Exception 的直接或间接子类（例如“受检查异常（checked exception）”），或者应用异常可以定义为 java.lang.RuntimeException 的子类（“未经检查的异常”（unchecked exception））。应用异常不可以是 java.rmi.RemoteException 的子类。java.rmi.RemoteException 和它的子类是保留的系统异常。

Javax.ejb.CreateException，javax.ejb.RemoveException，javax.ejb.FinderException 以及它们的子类看作是应用异常。这些异常用作标准应用异常向客户端报告来自 2.1 中 EJBHome 和/或 EJBLocalHome 接口的 create、remove 和 finder 方法的错误（参见 8.5.10 和 10.1.11 章节）。在本章中定义的应用异常规则涉及了这些异常。

14.1.2 异常处理的目标

EJB 规范中设计的异常处理要满足以下高层目标：

- 由企业 bean 实例抛出的应用异常应当被准确无误的报告给客户端（也就是，客户端得到同一个异常）（注：使用 web service 协议时可以不同，参见【25】）。
- 由企业 bean 实例抛出的应用异常不应当自动回滚客户端事务，除非应用被异常定义用来引起事务回滚。一般地，客户端应当给一个从应用异常恢复事务的机会。
- 能够安全地处理未知异常，这些异常可能使得实例的状态变量和/或后台的持久化数据处于不一致的状态。

14.2 Bean 提供者的责任

本节描述考虑异常处理的 Bean 提供者的视图和责任。

14.2.1 应用异常

Bean 提供者定义应用异常。被检查的应用异常可以定义在企业 bean 业务接口、home 接口、组件接口、消息监听器接口或 web service 终点的 throws 子句中。未经检查的应用异常通过用 ApplicationException 注释符注入，或在部署文件中用 application-exception 元素声明。

由于应用异常用于由客户端处理，而不是由系统管理员处理，因此它们应当只报告业务逻辑异常，而不是报告系统级异常。。

某些消息类型可以在消息监听器的接口中定义应用异常。在用于特定消息类型的资源适配器来决定如何处理这些异常。参见【15】。

Bean 提供者负责从业务方法中抛出合适的应用异常来向客户端报告业务逻辑异常。

应用异常不字段标记事务回滚，除非 ApplicationException 注释符应用到异常类并且指定 rollback 元素为 true，或者在部署文件的 application-exception 中指定 rollback 元素为 true。application-exception 元素的子元素 rollback 可以显式指定来覆盖由 ApplicationException 注释符缺省或指定的 rollback 值。

Bean 提供者在从企业 bean 实例抛出应用异常之前，必须做下面两件事的一个来保证数据的完整性：

- 确保实例处于一种状态，在这种状态下客户端即使继续和/或提交事务也不会引起数据完整性丢失。例如，实例在实例执行数据库更新之前抛出一个应用异常表示入参的值无效。
- 如果应用异常没有指定为引起事务回滚，在抛出应用异常之前用 EJBContext.setRollbackOnly 方法来标记事务回滚。标记事务回滚以确保事务不会被提交。

Bean 提供者也有责任在写 EJB2.1 的 bean 和早期的客户端视图时使用标准

的 EJB 应用异常（javax.ejb.CreateException， javax.ejb.RemoveException，javax.ejb.FinderException，和它们的子类）。EJB2.1 的 bean 和早期的客户端视图在 8.5.10 和 10.1.11 章节中描述。

Bean 提供者可以定义标准 EJB 应用异常的子类，并在企业 bean 方法中抛出这些子类的实例。一般地，子类为捕捉异常的客户端提供了更多的信息。

14.2.2 系统异常

系统异常是一个 java.rmi.RemoteException（或它的子类）或者是一个不是应用异常的 RuntimeException。

本节描述 Bean 提供者如何处理在会话或实体 bean 的业务方法、消息驱动 bean 的消息监听器方法、拦截器方法或回调方法（例如，ejbLoad）执行期间可能遭遇各种系统级异常和错误。

企业 bean 的业务方法、消息驱动 bean 的消息监听器方法、业务方法的拦截器方法或生命周期回调拦截器方法可能遭遇各种阻止方法成功完成的异常或错误。一般地，发生这种情况是因为这些异常和错误是不期望的，或者是期望的但 EJB 提供者不知道如何从异常中恢复。这种异常和错误可以是：数据库连接失败，JNDI 异常，不期望的来自其他企业 bean 的 RemoteException（注：注意企业 bean 的业务方法可以从 RemoteException 中恢复。这里是指业务方法不希望从 RemoteException 中恢复的情况），不期望的 RuntimeException，JVM 错误，等等。

如果企业 bean 遭遇不允许方法成功完成的系统级异常或错误，那么业务方法应当抛出与方法的 throws 子句匹配的非应用异常。但是 EJB 规范没有规定异常的确切用法，它鼓励 Bean 提供者遵循下面的指南：

- 如果 bean 方法遭遇系统异常或错误，那么它应当简单地将错误从 bean 方法迁移到容器（也就是，bean 方法不必捕捉异常）。
- 如果 bean 方法执行一个引起 bean 方法不能从中恢复的被检查异常（注：被检查异常不是 java.lang.RuntimeException 的子类）的操作，那么 bean 方法应当抛出封装了原始异常的 javax.ejb.EJBException。
- 任何其他的未期望错误应当使用 javax.ejb.EJBException 来报告。

注意：javax.ejb.EJBException 是 java.lang.RuntimeException 子类，因此它不必列在业务方法的 throws 子句中。

容器捕捉非应用异常；记录它（这些记录用于警告系统管理员）；然后除消息驱动 bean 外其他的都抛出 javax.ejb.EJBException（注：如果业务接口是一个继承了 java.rmi.Remote 的远程业务接口，那么向客户端抛出的是 java.rmi.RemoteException）或者如果使用 web service 客户端视图，那么抛出 java.rmi.RemoteException。如果使用 EJB2.1 客户端视图，如果客户端是远程客户端则容器抛出 java.rmi.RemoteException（或者它的子类），如果是本地客户端，则抛出 javax.ejb.EJBException（或它的子类）。对于消息驱动 bean，容器记录异常然后抛出封装了原始异常的 javax.ejb.EJBException 到资源适配器。（参见【15】）

客户端可见的异常在 14.3 章节描述。异常由通过容器或/和 bean 抛出的异常和客户端视图共同决定。

当捕捉到一个非应用异常时，Bean 提供者可以依赖容器来执行下述的任务：

- Bean 方法参与的事务将被回滚。
- 抛出非应用异常的实例的方法将不再被调用。

这意味着：bean 提供者不必在抛出非应用异常之前执行任何清理动作。由容器负责清理。

14.2.2.1 Javax.ejb.NoSuchEntityException

.NoSuchEntityException 是 EJBException 的子类。它应当由 EJB2.1 的实体 bean 的方法抛出，以表示后台实体已经被从数据库中移除了。

Bean 管理的持久化实体 bean 一般从 ejbLoad 和 ejbStore 方法、以及实现了业务接口中定义的业务方法中抛出异常。

14.3 容器提供者的责任

本节描述容器提供者的对于异常处理的责任。EJB 架构指定了下列异常的容器行为：

- 从会话和实体 bean 的业务方法，包括会话 bean 的业务方法的拦截器方

法中抛出的异常。

- 从消息驱动 bean 监听器方法和业务方法的拦截器方法中抛出的异常。
- 从超时回调方法中抛出的异常。
- 来自容器对企业 bean 的其他回调方法中的异常。
- 来自容器管理事务分割管理的异常

14.3.1 来自会话 bean 业务接口方法的异常

表 14 指定了容器如何处理由使用容器管理事务分割的 bean 的业务接口方法抛出的异常，包括由业务方法拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。这个表也解释了客户端将要收到的异常以及客户端如何从异常中恢复。（14.4 章节详细的描述了异常的客户端视图）。符号“AppException”指应用异常。

表 14 由使用容器管理事务分割的 bean 的业务接口方法抛出的异常的处理

方法条件	方法异常	容器行为	客户端视图
Bean 方法运行在调用者的事务上下文中【注意 A】。 这种情况可以发生在事务属性是 Required、Mandatory 和 Supports。	AppException	直接抛出 AppException。如果应用异常指定为产生回滚事务，则标记事务回滚。	接收 AppException。除非应用异常指定为引起事务回滚，否则可以继续在事务中计算并最终提交事务。
	所有其他的异常和错误	记录异常和错误【注意 B】。 标记事务回滚。 丢弃实例【注意 C】 抛出 javax.ejb.EJBTransactionRolledbackException 到客户端【注意 D】	接收 javax.ejb.EJBTransactionRolledbackException 继续事务是无效的。

Bean 方法运行在容器在分发业务方法之前即时启动的事务上下文中。这种情况可以发生在事务属性是 Required 和 RequiresNew。	AppException	如果实例调用 setRollbackOnly()，那么回滚事务，并再次抛出 AppException。如果应用指定为引起事务回滚，那么标记事务回滚，然后再次抛出 AppException	接收 AppException。如果客户端在一个事务中执行，则客户端的事务不标记回滚，客户端可以继续工作。
	所有其他异常	记录异常或错误。回滚容器启动的事务。丢弃实例。抛出 EJBException 到客户端。【注意 E】	接收 EJBException。如果客户端在一个事务中执行，那么客户端的事务可以标记为回滚，也可以不标记。
Bean 方法运行在未指定的事务上下文中。这种情况发生在事务属性是 NotSupported、Never 和 Supports。	AppException	再次抛出 AppExcepton	接收 AppException。如果客户端在一个事务中执行，则客户端的事务不标记回滚，客户端可以继续工作。
	所有其他异常	记录异常或错误。丢弃实例。抛出 EJBException 到客户端。【注意 F】	接收 EJBException。如果客户端在一个事务中执行，那么客户端的事务可以标记为回滚，也可以不标记。

注意：

【A】：调用者可以是另外一个企业 bean 或一个任意的客户端程序。

【B】：“记录异常或错误”意思是容器记录异常或错误以便系统管理员发现问题。

【C】:“丢弃实例”意思是容器不必调用任何业务方法或实例上的容器回调方法。

【D】：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么 avax.ejb.EJBTransactionRolledbackException 被抛到接收异常的客户端。

【E】：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么 avax.ejb.RemoteException 被抛到接收异常的客户端。

【F】：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么 avax.ejb.RemoteException 被抛到接收异常的客户端。

表 15 指定了容器必须如何处理由使用 bean 管理事务分割的 bean 的业务方法抛出的异常，包括由业务方法的拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。这个表也解释了客户端将要收到的异常以及客户端如何从异常中恢复。（14.4 章节详细的描述了异常的客户端视图）。

表 15 对由使用 bean 管理事务分割的会话 bean 业务接口抛出的异常的处理

Bean 方法条件	Bean 方法异常	容器行为	客户端接收
Bean 是有状态或无状态的会话 bean	AppException	再次抛出 AppException	接收 AppException
	所有其他异常	记录异常或错误。标记由实例已经启动的事务回滚，但还没有结束。抛出 EJBException 到客户端。【注意 A】	接收 EJBException

注意：

【A】：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，那么 avax.ejb.RemoteException 被抛到接收异常的客户端。

14.3.2 来自通过会话或实体 bean 的 2.1 客户端视图或通过 Web 服务客户端视图调用的方法的异常

在这里的业务方法认为是定义在企业 bean 业务接口、home 接口、组件接口或 web 服务终端（包括它们的父类）内的方法；以及下列会话 bean 或实体 bean 方法：ejbCreate<METHOD>，ejbPostCreate<METHOD>，ejbRemove，ejbHome<METHOD>和 ejbFind<METHOD>。

表 16 指定了容器如何处理由使用容器管理事务分割的 bean 的业务接口方法抛出的异常，包括由业务方法拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。这个表也解释了客户端将要收到的异常以及客户端如何从异常中恢复。（14.4 章节详细的描述了异常的客户端视图）。符号“AppException”指应用异常。

表 16 处理由使用容器管理事务分割的 web 服务客户端视图或 EJB2.1 客户端视图的方法抛出的异常

方法条件	方法异常	容器行为	客户端视图
Bean 方法运行在调用者的事务上下文中【注意 A】。 这种情况可以发生在事务属性是 Required、Mandatory 和 Supports。	AppException	再次抛出 AppException。如果应用异常指定为产生回滚事务，则标记事务回滚。	接收 AppException。除非应用异常指定为引起事务回滚，否则可以继续在事务中计算并最终提交事务（如果实例调用 setRollbackOnly 则提交将会失败）。
	所有其他的异常和错误	记录异常和错误【注意 B】。 标记事务回滚。 丢弃实例【注意 C】 抛出 javax.transaction.Tran	接收 javax.transaction.TransactionRolledbackException 或 javax.ejb.EJBTransactionRolledbackException

		sactionRolledbackException 到远程客户端；抛出 javax.ejb.EJBTransactionRolledbackException 到本地客户端。	继续事务是无效的。
Bean 方法运行在容器在分发业务方法之前即时启动的事务上下文中。这种情况可以发生在事务属性是 Required 和 RequiresNew。	AppException	如果实例调用 setRollbackOnly()，那么回滚事务，并再次抛出 AppException。如果应用指定为引起事务回滚，那么标记事务回滚，然后再次抛出 AppException。否则，尽量提交事务然后再次抛出 AppException。	接收 AppException。 如果客户端在一个事务中执行，则客户端的事务不标记回滚，客户端可以继续工作。
	所有其他异常	记录异常或错误。 回滚容器启动的事务。 丢弃实例。 抛出 RemoteException 到远程或 web 服务客户端。【注意 D】；抛出 EJBException 到本地客户端。	接收 RemoteException 或 EJBException。 如果客户端在一个事务中执行，那么客户端的事务可以标记为回滚，也可以不标记。
Bean 方法运行	AppExcepti	再次抛出	接收 AppException。

在未指定的事务上下文中。这种情况发生在事务属性是 NotSupported、Never 和 Supports。	on	AppExcepton	如果客户端在一个事务中执行，则客户端的事务不标记回滚，客户端可以继续工作。
	所有其他异常	记录异常或错误。 丢弃实例。 抛出 RemoteException 到远程或 web 服务客户端。抛出 EJBException 到本地客户端。	接收 RemoteException 或 EJBException。 如果客户端在一个事务中执行，那么客户端的事务可以标记为回滚，也可以不标记。

注意：

【A】：调用者可以是另外一个企业 bean 或一个任意的客户端程序。这种情况不适用于 web 服务终端的方法。

【B】：“记录异常或错误”意思是容器记录异常或错误以便系统管理员发现问题。

【C】：“丢弃实例”意思是容器不必调用任何业务方法或实例上的容器回调方法。

【D】：抛出 RemoteException 到 web 服务客户端意思是容器映射 RemoteException 到对应的 SOAP 错误。参见【25】

表 17 指定了容器必须如何处理由使用 bean 管理事务分割的 bean 的业务方法抛出的异常，包括由业务方法的拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。这个表也解释了客户端将要收到的异常以及客户端如何从异常中恢复。（14.4 章节详细的描述了异常的客户端视图）。

表 17 对由使用 bean 管理事务分割的会话 bean 业务接口抛出的异常的处理

Bean 方法条件	Bean 方法异常	容器行为	客户端接收

Bean 是有状态或无状态的会话 bean	AppException	再次抛出 AppException	接收 AppException
	所有其他异常	记录异常或错误。 标记回滚由实例已经启动的但还没有完成的事务。 丢弃实例。 抛出 RemoteException 到远程客户端或 web 服务客户端【注意 A】；抛出 EJBException 到本地客户端。	接收 RemoteException 或 EJBException

注意：

【A】：抛出 RemoteException 到 web 服务客户端意思是容器映射 RemoteException 到对应的 SOAP 错误。参见【25】

14.3.3 来自带 Web 服务客户端视图的无状态会话 bean 的 PostConstruct 和 PreDestroy 方法的异常

表 18 指定容器如何处理由带 Web 服务客户端视图的无状态会话 bean 的 PostConstruct 和 PreDestroy 方法抛出的异常。

表 18 由带 Web 服务客户端视图的无状态会话 bean 的 PostConstruct 和 PreDestroy 方法抛出的异常的处理

Bean 方法条件	Bean 方法异常	容器动作
Bean 是带 web 服务客户端视图的无状态会话	系统异常	记录异常或错误。 丢弃实例。

bean		

14.3.4 来自消息驱动 bean 消息监听器方法的异常

本节指定容器如何处理来自消息驱动 bean 消息监听器方法的异常。

表 19 指定了容器如何处理由使用容器管理事务分割的消息驱动 bean 的消息监听器方法抛出的异常，包括由业务方法拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。

表 19 对由使用容器管理事务分割的消息驱动 bean 的消息监听器方法抛出的异常的处理

方法条件	方法异常	容器动作
Bean 方法运行在调用者的事务上下文中【注意 A】。 这种情况可以发生在事务属性是 Required。	AppException 如果应用异常指定为产生回滚事务，则标记事务回滚。	如果实例调用 setRollbackOnly，则回滚事务然后再次抛出 AppException 到资源适配器。 否则，若应用异常没有指定为产生事务回滚则尽力提交事务并再次抛出 AppException。
	系统异常	记录异常或错误【注意 A】。 回滚容器启动的事务。 丢弃实例【注意 B】。 抛出封装了原始异常的 EJBException 到资源适配器。
Bean 方法运行在未指定的事务上下文中。	AppException	再次抛出 AppException 到资源适配器。

这种情况发生在事务属性是 NotSupported。	系统异常	记录异常或错误。 回滚容器启动的事务。 丢弃实例。 抛出封装了原始异常的 EJBException 到资源适配器。

注意：

【A】:“记录异常或错误”意思是容器记录异常或错误以便系统管理员发现问题。

【B】:“丢弃实例”意思是容器不必调用该实例上的任何方法。

表 20 指定了容器如何处理由使用 bean 管理事务分割的消息驱动 bean 的消息监听器方法抛出的异常，包括由业务方法拦截器方法抛出的异常。这个表指定了容器的行为是作为一个条件函数，在这个条件下业务接口方法执行和抛出异常。

表 20 对由使用 bean 管理事务分割的消息驱动 bean 的消息监听器方法抛出的异常的处理

Bean 方法条件	Bean 方法异常	容器动作
Bean 是消息驱动 bean	AppException	再次抛出 AppException 到资源适配器。
	系统异常	记录异常或错误。 标记回滚由实例启动的但还没有完成的事务。 丢弃实例。 抛出封装了原始异常的 EJBException 到资源适配器。

14.3.5 来自消息驱动 bean 的 PostConstruct 和 PreDestroy 方法的异常

表 21 指定了容器必须如何处理由消息驱动 bean 的 PostConstruct 和 PreDestroy 方法抛出的异常。

表 21 对由消息驱动 bean 的 PostConstruct 和 PreDestroy 方法抛出的异常的处理

Bean 方法条件	Bean 方法异常	容器动作
Bean 是消息驱动 bean	系统异常	记录异常或错误。 丢弃实例。

14.3.6 来自企业 bean 的超时回调方法的异常

本节指定容器如何处理由企业 bean 的超时回调方法抛出的异常。

表 22 和表 23 指定容器必须如何处理由企业 bean 的超时回调方法抛出的异常。超时回调方法不抛出应用异常也不能将异常抛到客户端。

表 22 对由使用容器管理事务分割的企业 bean 的超时回调方法抛出的异常的处理

方法条件	方法异常	容器动作
Bean 超时回调方法运行在在分派方法之前由容器启动的事务上下文中	系统异常	记录异常或错误【注意 A】。 回滚容器启动的事务。 丢弃实例【注意 B】。

注意：

【A】:“记录异常或错误”意思是容器记录异常或错误以便系统管理员发现问题。

【B】:“丢弃实例”意思是容器不必调用该实例上的任何方法。

表 23 对由使用 bean 管理事务分割的企业 bean 的超时回调方法抛出的异常

的处理

方法条件	方法异常	容器动作
Bean 超时回调方法可以使用 UserTransaction。	系统异常	记录异常或错误【注意 A】。 标记回滚实例启动的但还没有完成的事务。 丢弃实例【注意 B】。

注意：

【A】:“记录异常或错误”意思是容器记录异常或错误以便系统管理员发现问题。

【B】:“丢弃实例”意思是容器不必调用该实例上的任何方法。

14.3.7 来自容器调用的其他回调方法的异常

本节指定容器如何处理由容器调用的其他回调方法的抛出异常。这节适用于以下回调方法：

- 依赖注入方法。
- EntityBean 接口的 ejbActivate、ejbLoad、ejbPassivate、ejbStore、setEntityContext 和 unsetEntityContext 方法。
- SessionBean 接口的 PostActivate 和 PrePassivate 回调方法，以及/或 ejbActivate、ejbPassivate 和 setSessionContext 方法。
- MessageDrivenBean 接口的 setMessageDrivenContext 方法。
- SessionSynchronization 接口的 afterBegin、beforeCompletion 和 afterCompetion 方法。

容器必须按照下列方式处理来自这些方法的异常：

- 记录异常或错误以引起系统管理员的注意。
- 如果实例在事务中，则标记事务回滚。
- 丢弃实例（也就是容器不可以调用实例上的任何业务方法或容器回调方

法）。

- 如果异常或错误发生在客户端调用方法的处理过程中，那么抛出 javax.ejb.EJBException（注：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，则抛出 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图或 web 服务客户端实体，那么如果客户端是远程客户端则抛出 java.rmi.RemoteException，如果客户端是本地客户端则抛出 javax.ejb.EJBException。如果实例在客户端事务中执行，那么容器应当抛出 javax.ejb.EJBTransactionRolledbackException（注：如果业务接口是继承了 java.rmi.Remote 的远程业务接口，则抛出 javax.transaction.TransactionRolledbackException）。使用 EJB2.1 客户端视图或 web 服务客户端视图，则容器应当向远程客户端抛出 javax.transaction. TransactionRolledbackException 或向本地客户端抛出 javax.ejb. TransactionRolledbackException，因为它为客户端提供了更多的信息。（客户端知道事务不能继续了）

14.3.8 javax.ejb.NoSuchEntityException

NoSuchEntityException 是 EJBException 的子类。如果它被一个实体 bean 抛出，那么容器必须使用在章节 14.3.2，14.3.4 和 14.3.7 中描述的处理 EJBException 的规则来处理它。

为了告诉客户端错误的原因，容器应当抛出 java.rmi.NoSuchObjectException（它是 java.rmi.RemoteException 的子类）到远程客户端或抛出 javax.ejb.NoSuchObjectLocalException 到本地客户端。

14.3.9 不存在的无状态会话或实体对象

如果客户端调用已经移除的无状态会话或实体对象，则容器应当抛出 javax.ejb.NoSuchEJBException（注：如果业务接口继承了 java.rmi.Remote，则抛出 java.rmi.NoSuchObjectException）。如果使用 EJB2.1 客户端视图，容器应当向

远程客户端抛出 java.rmi.NoSuchObjectException（它是 java.rmi.RemoteException 的子类），向本地客户端抛出 javax.ejb.NoSuchObjectException。

14.3.10 来自对容器管理的事务进行管理的异常

容器有责任启动和提交容器管理的事务，正如在 13.6.2 章节中所述。本节指定了容器必须如何处理由启动和提交事务操作抛出的异常。

如果容器不能启动或提交容器管理的事务，容器必须抛出 javax.ejb.EJBException（如果业务接口继承了 java.rmi.Remote，则抛出 java.rmi.RemoteException）。如果使用 EJB2.1 客户端视图或 web 服务客户端，那么容器必须抛出 java.rmi.RemoteException 到远程或 web 服务客户端，抛出 javax.ejb.EJBException 到本地客户端。在容器不能为消息驱动 bean 或超时回调方法启动或提交容器管理的事务的情况下，容器必须抛出并记录 javax.ejb.EJBException.

但是，容器不应当抛出 javax.ejb.EJBException 或 java.rmit.RemoteException 或由于实例在 EJBContext 对象上调用了 setRollbackOnly 方法而回滚事务。在这种情况下，容器必须回滚事务，并且将业务方法的结果或业务方法抛出的应用异常传给客户端。

注意：某些容器实现可以透明地重试失败的事务，客户端和企业 bean 的代码都不知道。这种容器将在重试一定次数后抛出 javax.ejb.EJBExceptoin 或 java.rmi.RemoteException。

14.3.11 资源释放

当容器由于系统异常丢弃实例时，容器应当释放由该实例获得的所有资源，这些所需的资源通过声明在 bean 环境的资源工厂获得（参见 16.7）。

注意：当容器应当释放通过声明在 bean 环境的资源工厂获得的对资源管理器的连接时，通常情况下，容器不能释放“不受管理的”资源，这些资源由实例通过 JDK API 获得。例如，如果实例已经打开了一个 TCP/IP 连接，许多容器实

现将不能够释放连接。连接最终由 JVM 的垃圾回收机制释放。

14.3.12 支持废弃的 java.rmi.RemoteException 用法

EJB1.0 规范允许业务方法、ejbCreate、ejbPostCreate、ejbFind<METHOD>、ejbRemove 和容器调用的回调方法（也就是定义在 EntityBean、SessionBean 和 SessionSynchronization 接口内的方法）在企业 bean 类中实现，这些实现用 java.rmi.RemoteException 来向容器报告非应用异常。

java.rmi.RemoteException 的这种用法在 EJB1.1 中被废弃——按照 EJB1.1 规范写的企业 bean 应当用 javax.ejb.EJBException，用 EJB2.0 或以后版本写的必须用 javax.ejb.EJBException。

EJB1.1 和 EJB2.0 或以后规范要求容器支持 java.rmi.RemoteException 废弃的这种用法。容器应当将由企业 bean 方法抛出的 java.rmi.RemoteException 和 javax.ejb.EJBException 同等对待。

14.4 异常的客户端视图

本节描述接收来自企业 bean 调用的异常的客户端视图。

客户端可以通过以下方式获取企业 bean：通过企业 bean 的业务接口（无论是 local 还是 remote）、通过企业 bean 的远程 home 和远程接口、通过企业 bean 的本地 home 和本地接口，或通过企业 bean 的 web 服务客户端视图（这要依赖于客户端是用 EJB3.0 API 还是用早期的 API，以及客户端是否是远程客户端、本地客户端还是 web 服务客户端）。

一般地，业务接口的方法不管接口是远程还是本地接口都不会抛出 java.rmi.RemoteException。

远程 home 接口，远程接口和 web 服务终端接口都是 Java RMI 接口，因此它们的方法（包括从父类继承来的方法）的 throws 子句包括必须的 java.rmi.RemoteException。它们的 throws 子句可以包括任意数量的应用异常。

本地 home 和本地接口都是 Java 本地接口，因此它们的方法（包括从父类继

承来的方法）的 throws 子句不包括 java.rmi.RemoteException。它们的 throws 子句可以包括任意数量的应用异常。

14.4.1 应用异常

14.4.1.1 本地和 8 远程客户端

如果客户端程序收到来自企业 bean 调用的应用异常，那么客户端可以继续调用企业 bean。应用异常不会引起 EJB 对象的移除。

尽管容器不会因为应用异常自动标记事务回滚，但是事务可以由企业 bean 实例抛出应用异常之前标记为回滚，或应用异常已经被指定为要求容器回滚事务。由两种方式来知道某个异常是否引起事务回滚：

- 静态绑定：程序员可以检查企业 bean 的客户端视图接口的文档。Bean 提供者可能已经指定（尽管不要求这么做）应用异常在抛出异常之前企业 bean 标记事务回滚。
- 动态绑定：使用容器管理事务的企业 bean 的客户端可能使用 javax.ejb.EJBContext 对象的 getRollbackOnly 方法来明白目前的事务是否已经被标记回滚；其他的客户端可以使用 javax.transaction.UserTransaction 接口的 getStatus 来获取事务的状态。

14.4.1.2 Web 服务客户端

如果无状态会话 bean 从它的 web 服务方法中抛出应用异常，那么它容器负责将异常映射到在 WSDL 中指定的 SOAP 错误，这个 WSDL 是描述了无状态会话 bean 实现的端口类型。对于 java 客户端，客户端接收的异常由在【25】中映射规则来描述。

14.4.2 Java.rmi.RemoteException 和 javax.ejb.EJBException

如上所述，客户端以收到 javax.ejb.EJBException 或 java.rmi.RemoteException

来表明它调用企业 bean 方法或正确完成调用失败。异常可以由容器或客户端与容器之间的通信子系统抛出。

一般情况下，如果客户端从方法调用收到 javax.ejb.EJBException 或 java.rmi.RemoteException，那么它不知道企业 bean 的方法是否完成。

如果客户端在事务上下文中执行，则客户端事务可以或不必由通信子系统或目标 bean 的容器标记为回滚。

例如，如果后台事务服务器或目标 bean 的容器由于业务方法部分完成而担心数据的完整性，那么事务将被标记为回滚。例如当目标 bean 的方法由于 RuntimeException 而返回，或如果远程服务器在业务方法执行期间宕机，那么就会发生业务方法的部分完成。

标记事务回滚不是必需的。例如当在客户端的通信子系统不能将请求发送到服务器时，就可以不标记事务回滚。

当在事务上下文中执行的客户端从企业 bean 调用中收到 EJBException 或 RemoteException，那么客户端可以使用下列的策略来处理异常：

- 停止事务。如果客户都是事务的发起者，那么它可以简单的回滚它的事务。如果客户端不是事务的发起者，那么它可以标记事务回滚或执行引起事务回滚的操作。例如，如果客户端是一个企业 bean，那么企业 bean 可以抛出一个 RuntimeException 来让容器回滚事务。
- 继续事务。客户端可以在同一个或其他的企业 bean 上执行附加的操作，并最后试着提交事务。如果事务在 EJBException 或 RemoteException 被抛到客户端时被标记为回滚，那么提交会失败。

如果客户端选择继续事务，那么客户端可以首先查询事务的状态以避免在以标记为回滚的事务上进行无用的计算。如果客户端是使用容器管理事务分割的企业 bean ，那么它可以使用 EJBContext.getRollbackOnly 方法来测试事务是否已经被标记为回滚；如果客户端是使用 bean 管理事务分割的企业 bean 或其他客户端类型，则它可以使用 UserTransaction.getStatus 方法来得到事务的状态。

14.4.2.1 Javax.ejb.EJBTransactionRolledbackException，javax.ejb.TransactionRolledbackLocalException 和 javax.transaction.TransactionRolledbackException

Javax.ejb.EJBTransactionRolledbackException，javax.ejb.TransactionRolledbackLocalException 和都是 Javax.ejb.EJBException 的子类。javax.transaction.TransactionRolledbackException 是 java.rmi.RemoteException 的子类。它在 JTA 标准扩展中定义。

如果客户端接收到这些异常，那么客户端一定知道事务已经被回滚。它继续这个事务也是无用的，因为事务不再会被提交。

14.4.2.2 Java.ejb.EJBTransactionRequiredException，javax.ejb.TransactionRequiredLocalException 和 javax.transaction.TransactionRequiredException

Java.ejb.EJBTransactionRequiredException 和 javax.ejb.TransactionRequiredLocalException 是 javax.ejb.EJBException 的子类。javax.transaction.TransactionRequiredException 是 java.rmi.RemoteException 的子类。它被定义在 JTA 标准的扩展中。

Java.ejb.EJBTransactionRequiredException，javax.ejb.TransactionRequiredLocalException 和 javax.transaction.TransactionRequiredException 通知客户端目标企业 bean 必须在客户端的事务中被调用，但调用企业 bean 的客户端没有事务上下文。

这个错误通常表示应用没有被正确地建立。

14.4.2.3 Javax.ejb.NoSuchEJBException ，javax.ejb.NoSuchObjectLocalException 和 java.rmi.NoSuchObjectException

Javax.ejb.NoSuchEJBException 是 javax.ejb.EJBException 的子类。如果本地业务方法由于 EJB 对象不存在而不能完成，则会话 bean 的业务接口向客户端抛出这个异常。

- javax.ejb.NoSuchObjectLocalException 是 javax.ejb.EJBException 的子类。如果本地业务方法由于 EJB 对象不再存在而不能完成，则抛出这个异常。
- java.rmi.NoSuchObjectException 是 java.rmi.RemoteException 的子类。如果远程业务方法由于 EJB 对象不再存在而不能完成，则抛出这个异常。

14.5 系统管理员的职责

系统管理员负责监控由容器记录的非应用异常的日志和错误，并采取行动去纠正引起这些异常和错误的问题。

15 支持分布式交互

本章介绍对从分布在网络上的客户端通过 EJB2.1 远程客户端视图获取企业 bean 的交互能力的支持，以及从客户端是 J2EE 组件的远程客户端调用企业 bean 的分布式交互要求。

15.1 对分布式的支持

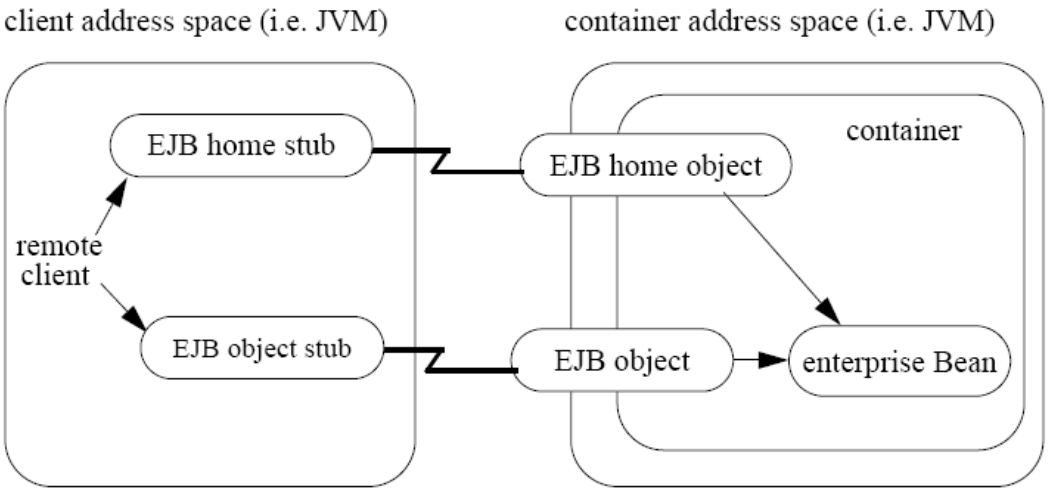
企业 bean 的远程客户端视图的远程 home 和 remote 接口定义为 Java RMI【6】接口。这可以让容器将远程 home 和 remote 接口实现为分布式对象。使用远程 home 和 remote 接口的客户端可能是与企业 bean 处于不同的机器上（位置透明），远程 home 和 remote 接口的对象引用可以被跨网络传递到其他的应用。

EJB 规范进一步限制了能被企业 bean 使用的 Java RMI 类型是合法的 RMI-IIOP 类型【10】。这使得 EJB 容器实现者能够使用 RMI-IIOP 作为对象分布式协议。

15.1.1 处于分布式环境的客户端对象

当使用 RMI-IIOP 协议或类似的分布式协议时，远程客户端与为服务端对象使用 stub 的企业 bean 进行通信。Stub 实现了远程 home 接口和 remote 接口。

图 31 EJB 客户端 Stub 的位置



在客户端使用的通信 stub 是在企业 bean 部署时通过容器提供者的工具生成的。用在客户端的 stub 是特定于用于远程通信的硬代码协议的。

15.2 交互概述

部署在某个供应商的服务器产品上的会话 bean 和实体 bean 可能需要被部署在另一个提供商产品的 Java EE 客户端通过远程客户端实体获取。EJB 定义了一个基于 CORBA/IIOP 的标准交互协议来满足这种需求。

这里描述的交互协议必须被兼容的 EJB 产品支持。也可以支持提供商特有的协议。

图 32 展示了一个涉及几个提供商的异构环境来解释 EJB 的交互能力。

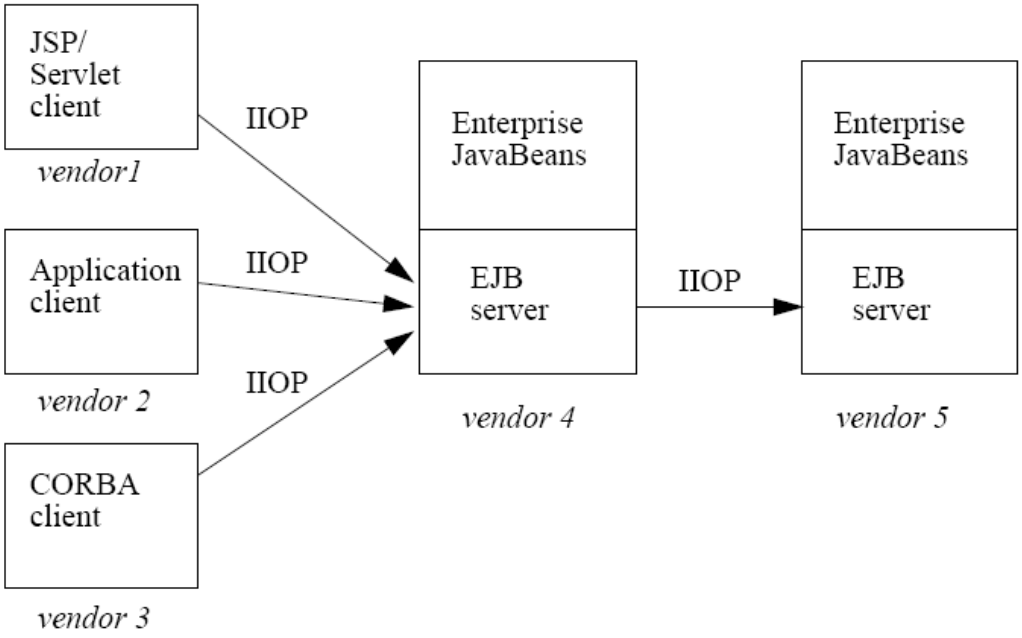


图 32 异构的 EJB 环境

本章包含以下章节：

- 描述 EJB 调用交互的目标
- 提供阐述场景
- 描述远程调用、事务、命名和安全的交互要求

15.2.1 交互的目标

本章中指定的交互要求的目标如下：

- 使得处于部署在某个服务器提供商的 Java EE 容器的应用中的客户端可以获取部署在另外一个容器提供者的容器中的另一个应用的会话或实体 bean 的服务。例如，部署在某个供应商提供的兼容的 web 服务器中的 web 组件（JSP 和 Servlet）能够调用部署在由其他服务器供应商提供的 Java EE 兼容的 EJB 容器中的企业 bean 的业务方法。
- 能够不对 Java EE 应用开发者增加任何要求的情况下完成交互。
- 保证在兼容的 Java EE 产品间直接的交互。它必须让企业客户能够安装多个来自不同厂商的（潜在的还可能是不同的操作系统）Java EE 服务器，在应用服务器上部署应用，并且多个应用可以相互操作。

- 尽可能能够平衡由标准化社团（包括 IETF，W3C 和 OMG）已经做的关于交互的工作，以便客户可以使用工业标准和使用标准协议来获取企业 bean。

本规范没有解决在企业 bean 和非 java EE 组件间的交互问题。Java EE 平台【12】和 JAX-RPC 和 JAX-WS 规范【25】,【32】描述了与因特网客户（使用 HTTP 和 XML）的交互要求和与企业信息系统交互要求（使用连接器架构【15】）。

由于在这里描述的交互协议是基于 CORBA/IIOP 的，用 Java，C++或其他语言写的 CORBA 客户端也可以调用企业 bean 的方法。

本章包含了以前的 EJB1.1 到 CORBA 映射的文档【16】。

15.3 交互场景

本节展现大量交互场景，这些场景为后续小节中描述的交互机制提出了要求。这些场景是用于阐述而不是规定。本节没有要求 Java EE 产品按照这里描述的方式来支持这些场景。

Java EE 应用都是多层、可使用 web 的应用。每个应用都有一个或多个组件组成，这些组件部署在容器中。容器有四种类型：

- EJB 容器，它容纳企业 bean。
- Web 容器，它容纳 JSP 和 Servlet 组件，以及静态文档，包括 HTML。
- 应用客户端容器，它容纳独立的应用。
- Applet 应用，它容器可以从网站下载的 applet。此时，不要求 applet 可以直接调用企业 bean 的远程方法。

下面的场景描述了容纳在这四个类型容器中的组件间的交互。

15.3.1 用于电子商务的 Web 容器和 EJB 容器间的交互

应用

这个场景发生在跨网络的 B2B 和 B2C 交互。

场景 1：*一个客户想从一个网络书店买一本书。书店的网站由一个包含充当*

展现层的 JSP 的 Java EE 应用和一个包含了实现业务逻辑和数据存储代码的企业 bean 的 Java EE 应用。JSP 和企业 bean 部署在不同的供应商的容器中。

部署时：*企业 bean 被部署，并且它们的 EJBHome 对象被发布到 EJB 服务器的命名服务。部署人员将在 JSP 部署文件中的 EJB 引用关联到企业 bean 的 EJBHome 对象的 URL 上，这些对象可以从命名服务中被查找到。在企业 bean 的部署文件中为所有的业务方法指定的事务属性是 RequiresNew。因为“结帐”JSP 要求安全地设置购买支付，书店管理员配置“结帐”JSP 为需要通过使用服务器验证的 HTTPS 来进行存取。用基于登录方式来进行客户授权。“搜索图书”JSP 通过普通的 HTTP 来获取。两个 JSP 都和存取数据库的企业 bean 进行交互。Web 和 EJB 容器使用相同的客户域且彼此互相信任；在 web 和 EJB 服务器间的网络不能保证不受攻击。*

运行时：*客户使用浏览器获取图书搜索 JSP。这个 JSP 在命名服务中查找企业 bean 的 EJBHome 对象并使用搜索关键字作为参数调用 findBooks(…)。Web 容器和容器间使用双向授权的 EJB 容器间建立一个安全会话，然后调用企业 bean。客户然后决定买书，并获取“结帐”JSP。客户在登录窗口内输入必需的信息，这些信息用于 web 服务器来授权客户。这个 JSP 调用企业 bean 去更新书和客户数据库。客户的主要信息被传送到 EJB 容器用于授权检查。企业 bean 完成更新并提交事务。这个 JSP 给客户返回一个确认页面。*

15.3.2 在企业局域网内的应用客户端容器和 EJB 容器间交互

场景 2.1：*企业有一个由雇员从他们的桌面系统使用的费用记账应用。服务器端由一个包含企业 bean 的 Java EE 应用组成，它配置在一个供应商的 Java EE 产品上，它是一个数据中心。客户端由另一个包含了应用客户端的 Java EE 应用组成，客户端使用另外一个供应商的 Java EE 架构。在应用客户端和 EJB 容器间的网络是不安全的，需要防止电子欺骗和其他的攻击。*

部署时：*企业 bean 被部署，他们的 EJBHome 对象被发布到企业的命名服务上。应用客户端配置了 EJBHome 对象的名字。部署人员将允许访问企业 bean 的角色与雇员建立映射。系统管理员配置应用客户端、将使用客户端的 EJB 容*

器、服务器授权和消息保护的安全设置。系统管理员也做必需的客户端配置使得可以进行客户端授权。

运行时：*雇员使用用户名和密码登录。应用客户端容器可以与企业的授权服务架构交互以设置雇员的证书。客户端应用调用命名服务去查找企业 bean 的 EJBHome 对象，然后创建企业 bean。应用客户端容器使用安全传输协议与命名服务器和 EJB 服务器交互，EJB 服务器进行双向授权，而且也保证消息的保密性和完整性。雇员然后输入费用信息并提交。提交会调用企业 bean 的远程业务方法。EJB 容器执行授权检查，如果通过，执行业务方法。*

场景 2.2：*除了系统管理员没有设置客户端授权架构（它可以在传输层进行授权）外，其他和场景 2.1 都是一样的。在运行时，客户端容器需要在方法调用期间将用户的密码发送到服务器以验证雇员。*

15.3.3 在企业局域网内两个 EJB 容器间的交互

场景 3：*企业有一个费用记账应用，它需要与工资应用交互。应用使用企业 bean 并被部署到不同供应商的 Java EE 服务器上。Java EE 服务器和命名/授权服务可以在企业的数据中心，它们之间使用物理上安全的单独网络，或者它们可能需要跨局域网进行通信，但那是不太安全的。应用需要更新账户和工资数据库。雇员（客户端）存取在场景 2 中描述的费用记账应用。*

配置时：*配置人员配置两个应用使用相应的数据库资源。记账应用配置了工资应用的 EJBHome 对象的名字。工资 bean 的部署文件为所有的方法指定的事务属性是 RequiresNew。应用都使用相同的人员到角色的映射（例如角色可以是 Employee，PayrollDept，AccountDept）。这两个应用的配置员设置两个 EJB 容器的信任关系，以便可以衍生地调用其他容器的企业 bean 而不需要授权。如果网络不是物理上安全的，那么系统管理员也要设置这个两个容器的消息保护参数。*

运行时：*雇员向记账应用请求访问工资应用。记账应用从命名/目录服务中查找工资应用的 EJBHome 对象，并创建企业 bean。它更新记账数据库并调用工资 bean 的远程方法。记账 bean 的容器在方法调用中递延雇员的授权。工资 bean 的容器将递延过来的雇员授权映射到一个角色，做授权检查，并设置工资 bean*

的事务上下文。容器启动一个新事务，然后工资 bean 更新工资数据库，然后容器提交事务。记账 bean 从工资 bean 接收到一个状态回复。如果在工资 bean 中发生错误，记账 bean 执行从错误中恢复的代码并恢复数据库到一致状态。

15.3.4 局域网应用在 Web 容器和 EJB 容器间的交互

场景 4：*除了使用“胖客户端”桌面应用来获取企业 bean 的费用记账应用外，其他和场景 2.1 都是一样的，雇员使用 web 浏览器连接到容纳 JSP 的在局域网内的 web 服务器。JSP 从用户收集输入（例如，功过 HTML 表单），调用容纳真正业务逻辑的企业 bean，然后格式化由企业 bean 返回的结果（使用 HTML）。*

配置时：*企业配置员配置记账 JSP 通过使用双向授权的 HTTPS 来访问。Web 和 EJB 容器使用相同的客户域，并且彼此是信任的关系。*

运行时：雇员登录到客户端桌面，启动浏览器并访问记账 JSP。浏览器与 web 服务器创建一个 HTTPS 会话。执行客户端授权，（例如）使用在登录时已经创建的雇员证书（浏览器和操作系统交互以获得雇员的证书）。这个 JSP 在命名服务中查找企业 bean 的 EJBHome 对象。Web 容器和容器间使用双向授权和完整/保密保护的 EJB 容器一起创建一个安全会话，并调用企业 bean 的方法。

15.4 交互需求概述

用于支持以上场景的交互需求有：

1. 调用在企业 bean 的 EJBObject 和 EJBHome 对象引用上的远程方法（场景 1，2，3，4），在 15.5 章节中描述。
2. 查找企业 bean 的 EJBHome 对象的命名服务（场景 1，2，3，4），在 15.5 章节描述。
3. 消息的完整性和保密性包含（场景 1，2，3，4），在 15.8 章节中描述。
4. 在应用客户端和 EJB 容器间的授权（在 15.8 章节描述）

   4.1 当使用客户端授权基础设施例如证书（场景 2.1）时，在传输协议层的双向验证。

4.2 从应用客户端向 EJB 容器传送用户授权数据，使得在客户端没有授权基础设施时（场景 2.2）EJB 容器可以验证客户端。

5. 在两个 EJB 容器间或 Web 和 EJB 容器间的双向授权使得在权利被递延之前建立信任（场景 1，3，4），在 15.8 章节描述。
6. 当客户端和服务器容器有信任关系时（场景 1，3，4），从 web 或 EJB 容器调用企业 bean 进行广域网或局域网用户主体名字的传递，在 15.8 章节描述。

EJB，web 和应用客户端容器必须支持上述的需求，无论单独使用还是组合使用。

15.5 远程调用交互

本节描述当客户端容器和 EJB 容器来自不同供应商时，能够远程调用 EJBObject 和 EJBHome 对象引用的交互机制。这需要满足章节 15.4 中的交互需求（1）。

所有的 EJB、web 和应用客户端容器必须支持 IIOP1.2 协议。EJB 容器必须有服务于 IIOP1.2 的能力。IIOP1.2 是来自 OMG 的 CORBA2.3.1 规范【17】的一部分（注：CORBA API 和早期的 IIOP 协议已经通过 JavaIDL 和 RMI-IIOP 包含在 J2SE1.2，J2SE1.3 和 J2EE1.2 平台）。容器可以支持供应商自己的协议。

用于 EJBObject 和 EJBHome 对象引用的 CORBA 交互对象引用（IOR）必须包含 GIOP 版本号 1.2。在所有 Java EE 容器中的 IIOP 基础设施必须能够接收分段的 GIOP 消息，尽管发送分段消息是可选的。Java EE 客户端和服务器对双向 GIOP 消息的支持是可选的：如果 Java EE 服务器从包含 BiDirIIOPServiceContext 结构的客户端接收一个 IIOP 消息，那么它可以或可以不使用与发送消息相同的连接。

由于 java 应用缺省使用 Unicode，所以要求 Java EE 容器支持 Unicode UTF16 编码集以转换字符和字符串数据（在 IDL 中是 wchar 和 wstring 数据类型）。Java EE 容器可以可选地支持其他的编码集。EJBObject 和 EJBHome IOR 必须有标记了 TAG_CODE_SETS 的组件，它声明了 EJB 容器支持的编码集。包含 wchar 和

wstring 数据类型的 IIOP 消息必须有编码集服务上下文字段。Java EE 容器必须支持 CORBA2.3.1 要求的编码集。

要求 EJB 容器使用 java 语言到 IDL 映射规范【10】用应代码的格式将 Java 类型在 IIOP 消息中的高层展现转换成在 CORBA2.3 的 GIOP 规范中描述的 IDL 类型。下面的子章节描述了映射的细节。

15.5.1 将 Java 远程接口映射到 IDL

Java 语言到 IDL 的映射规范【10】详细描述了会话 bean 或实体 bean 的远程 home 和 remote 接口如何被映射到 IDL。一般地，当使用基于 IIOP 的 Java RMI 来调用企业 bean 时，这种映射是隐式的。Java EE 客户端只使用 RMI API 来调用企业 bean。客户端容器可以将 CORBA 可移植的 Stub API 用于客户端 stub。EJB 容器可以为每个 EJBObject 或 EJBHome 对象创建 CORBA Tie 对象。

15.5.2 值对象与 IDL 的映射

在远程调用企业 bean 期间，按值传递的 java 接口是 javax.ejb.Handle，javax.ejb.HomeHandle 和 javax.ejb.EJBMetaData。由实体 bean 查找方法返回的 Enumeration 或 Collection 对象是值对象。也有应用特有的值类型，它们作为企业 bean 调用的参数或返回值。另外，由远程方法抛出的几个 java 异常类也对应具体的 IDL 值类型。所有这些值类型都被用 Java 语言到 IDL 的映射规则影射到 IDL 抽象值类型或抽象接口。

15.5.3 系统异常的映射

Java 系统异常，包括 java.rmi.RemoteException 和他的子类，都可以由 EJB 容器抛出。如果客户端调用是通过 IIOP，那么要求 EJB 服务器将这些异常映射到 CORBA 系统异常，并将它们放到 IIOP 回复消息中发送给客户端，如下表：

由 EJB 容器抛出的系统异常	由客户端 ORB 接收的 CORBA 系统异常

Javax.transaction.TransactionRolledbackException	TRANSACTION_TOLLEDBACK
Javax.transaction.TransactionRequireException	TRANSACTION_REQUIRED
Javax.transaction.InvalidTransactionException	INVALID_TRANSACTION
Java.rmi.NoSuchObjectException	OBJECT_NOT_EXIST
Java.rmi.AccessException	NO_PERMISSION
Java.rmi.MarshalException	MARSHAL
Java.rmi.RemoteException	UNKNOWN

对于 EJB 客户端，ORB 的解包机制将在 IIOP 回复消息中的 CORBA 系统异常映射到对应的 java 异常，正如在 java 语言到 IDL 的映射中规定的一样。这样 Java EE 的客户端组件就接收到原始的 java 异常。

15.5.4 获取 Stub 和客户端视图类

当 java EE 组件（应用客户端，JSP，Servlet 或企业 bean）通过 JNDI 查找接收到一个 EJBObject 或 EJBHome 对象的引用或作为企业 bean 调用的参数或返回值时，需要为企业 bean 的远程 home 或 remote RMI 接口创建一个 RMI-IIOP stub 类（proxy）的实例。当组件接收一个作为企业 bean 的参数或返回值的值对象时，需要创建值类的实例。Stub 类、值类和其它客户都视图类必须可以被引用的容器得到（容器容纳接收引用或值类型的组件）。

客户端视图类，包括应用值类，必须和引用的组件应用打包在一起，正如在 20.3 章节所述。

调用 EJBHome 和 EJBObject 引用的 Stub 必须由引用的容器提供，例如，在部署时生成被引用的 bean 的 EJBHome 和 EJBObject 接口的 stub 类，被引用的 bean 和引用它的组件应用打包在一起。Stub 类可以，也可以不遵循 RMI-IIOP 可移植 stub 架构。

容器可以可选地支持运行时下载容器需要的 stub 和值类。CORBA2.3.1 规范和 Java 语言到 IDL 的映射指定如何下载 stub 和值类型实现：使用 codebase URL，它可以嵌入到 EJBObject 或 EJBHome 得 IOR，或放在 IIOP 消息服务上下文中，

或和值类型打包（marshall）在一起。用于下载的 URL 可以可选地包含在 HTTPS URL 中以保证安全下载。

15.5.5 系统值类

系统值类是可序列化的之类，它实现了 javax.ejb.Handle，javax.ejb.HomeHandle，java.ejb.EJBMetaData，java.util.Enumeration，java.util.Collection 和 java.util.Iterator 接口。这些值类由 EJB 容器提供者提供。它们必须以 JAR 文件的形式由容纳被引用 bean 的容器提供。对于交互场景，如果一个引用组件在运行时使用这样的系统值类，部署人员必须保证这些由被引用 bean 的容器提供的系统值类可以被引用它的组件获取到。例如，这可以通过将这些系统值类包含在引用它的容器的类路径中，或者可以将这些系统值类和引用它的组件应用部署在一起。

这些系统值类的实现必须是可移植的（它们必须只使用 J2SE 和 Java EE 的 API），以便他们可以在另一个供应商的容器中被初始化。如果系统值类实现需要在运行时加载应用特有的类（来自远程 home 或 remote 接口），那么它必须使用线程上下文类加载器。引用方容器必须保证在运行时系统值类实例可以通过线程上下文类加载器获得到应用特有的类。

15.5.5.1 HandleDelegate SPI

15.6 事务交互

15.7 命名交互

15.8 安全交互

16 企业 bean 的环境

本章描述了企业 bean 如何声明依赖外部资源和环境中的其他对象，以及如

何注入这些项目到企业 bean 或在 JNDI 命名上下文中访问。

16.1 概述

应用组装者和部署者应当能够客户化企业 bean 的业务逻辑而不需要访问企业 bean 的源代码。

另外，ISV 通常开发的企业 bean 在很大程度上是独立于应用将被部署的操作环境。大多数企业 bean 必须访问资源管理器和外部信息。关键问题是企业 bean 如何能够定位到外部信息而不需要预先知道外部资源在目标操作环境中是如何被命名和被组织的。JNDI 命名上下文和 java 语言元数据注释符提供了这种能力。

企业 bean 环境机制打算解决上述的两个问题。

本章按下述方式组织：

- 节 16.2 定义了使用 JNDI 命名上下文的一般规则和引用命名上下文中条目的 java 语言注释与 JNDI 的交互。
- 节 16.3 定义了每个 EJB 角色的责任，例如 bean 提供者、应用组装者、部署人员和容器提供者。
- 节 16.4 定义了指定和访问企业 bean 环境的基本机制和接口。这一节解释了使用企业 bean 环境来客户化企业 bean 业务逻辑的用法。
- 节 16.5 定义了使用 EJB 引用来获取另一个企业 bean 的业务接口或 home 接口的方法。EJB 引用是企业 bean 环境中的特殊条目。
- 节 16.6 定义了使用 web 服务引用来获取 web 服务接口的方法。Web 服务引用是企业 bean 环境中的特殊条目。
- 节 16.7 定义了使用资源管理器连接工厂引用来获取资源管理器连接工厂的方法。资源连接器工厂引用是企业 bean 环境中的特殊条目。
- 节 16.8 定义了使用资源环境引用来获取与资源（例如，一个 CCI InteractionSpec）关联的受管理对象的方法。资源环境引用是企业 bean 环境中的特殊条目。
- 节 16.9 定义了使用消息目的地引用来获取与资源关联的消息目的地的方法。消息目的地引用可以指定应用内部的消息流。消息目的地引用是企

业 bean 环境中的特殊条目。

- 节 16.10 描述了使用持久化单元引用来获取实体管理器工厂的方法。
- 节 16.11 描述了合法企业 bean 使用 UserTransaction 对象在 bean 环境中启动、提交和回滚事务的用法。
- 节 16.13 描述了在企业 bean 环境中引用 CORBA 对象的用法。
- 节 16.14 描述了获取 TimerService 的方法。
- 节 16.15 描述了获取 bean 的 EJBContext 对象的方法。

16.2 企业 bean 环境作为 JNDI 命名上下文

企业 bean 的环境是一种机制，这种机制可以在部署或组装时客户化企业 bean 的业务逻辑。企业 bean 的环境可以在不需要获取或改变企业 bean 源代码的情况下客户化企业 bean。

注释符和部署文件是应用组装者和部署者获取客户化业务逻辑信息和获取外部信息的主要途径。

容器实现企业 bean 的环境，并将它作为 JNDI 命名上下文。企业 bean 的环境可按下述方式来使用：

- 企业 bean 使用来自环境的条目。来自环境的条目被容器注入到企业 bean 的字段或方法，或者 bean 的方法可以使用 EJBContext lookup 方法或 JNDI 接口来访问环境。Bean 提供者用 java 语言元数据注释符或部署文件来声明企业 bean 希望在运行时得到的环境条目。
- 容器提供存储企业 bean 环境的 JNDI 命名上下文的实现。容器也提供让部署者创建和管理每个企业 bean 环境的工具。
- 部署者使用容器提供的工具创建和初始化环境条目，这些条目通过企业 bean 的注释符或部署文件来声明。部署者可以设置和改变环境条目的值。
- 容器按照 bean 的元数据注释符或部署文件的指定将来自环境的条目注入到企业 bean 的字段或方法。
- 容器可以让企业 bean 实例在运行时获取这些环境命名上下文。企业 bean

实例可以使用 EJBContext 的 lookup 方法或 JNDI 接口来得到环境条目的值。

容器也必须让企业 bean 的任何拦截器类和 JAX-WS 消息处理器可以获得企业 bean 的环境。拦截器和 web 服务处理器类共享 bean 的环境。在本章的上下文中，属于“bean”应当看作是包含了 bean 的拦截器和处理器类，除非额外说明。

16.2.1 共享环境条目

每个企业 bean 都定义了它自己的环境条目集。企业 bean 的所有实例都共享这些环境条目；这些环境条目不被其他企业 bean 共享。企业 bean 实例不允许在运行时更改 bean 的环境。

兼容性提示：如果企业 bean 由 EJB2.1 API 规范实现，且在容器内被部署多次，那么每次部署都会创建不同的 home。部署者可以为每个 home 设置不同的企业 bean 环境条目的值。

一般情况下，查找 JNDI 的 java:命名空间中的对象要求每次都返回被请求对象的一个新实例。以下情况除外：

- 容器知道对象是不变的（例如，对象类型是 java.lang.String），或者知道应用不能改变对象的状态。
- 对象定义成单例的，这样只有一个对象实例可以存在于 JVM 中。
- 用于查找的名字返回的是可能会被共享的对象的实例。java:comp/ORB 就是这样的名字。

在这些情况下，返回对象的共享实例。在其他的情况下，必须返回被请求对象的新实例。注意：对于资源适配器连接对象，返回对象是适配器的 ManagedConnectionFactory 实现。

对象的每次注入都与会进行 JNDI 查找。根据上述的规则来决定是注入被请求对象的新实例还是注入共享实例。

术语警告：企业 bean 的“环境”不应当和 JNDI 文档中定义的“环境属性”混淆。

16.2.2 注释使用环境条目

Bean 类的字段或方法都可以被注释请求注入一个 bean 环境的条目。在本章中描述的所有类型的资源或其他环境条目（注：本章中使用的术语“资源”通常指的是可以作为资源的环境条目。特殊情形下的资源在节 16.7 中描述。）都可以被注入。也可以在部署文件中要求注入这些资源类型。字段或方法可以是各种访问控制符（public，private 等），但不能是 static。

- Bean 类的字段可以是注入的目标。字段不能是 final。缺省情况下，字段的名字和类的名字组合在一起直接作为 bean 命名上下文的名字。例如，在包 com.acme.example 中的类 MySessionBean 的字段 myDatabase 对应于 JNDI 的名字是 java:comp/env/com.acme.example.MySession-Bean/myDatabase。注释也允许显式指定 JNDI 的名字。
- 环境条目也可以通过遵循 JavaBean 属性命名规则的 bean 方法被注入到 bean 上。这个注释符应用到属性的 set 方法，调用 set 方法来注入环境条目。JavaBean 的属性名（不是方法名）用作缺省的 JNDI 名。例如，类 MySessionBean 的方法名为 setMyDatabase 对应的 JNDI 名为 java:comp/env/com.example.MySessionBean/myDatabase。
- 是使用部署文件条目来指定注入时，JNDI 名和实例变量名或属性名都要显式地被指定。注意 JNDI 名总是相对于 java:comp/env 命名上下文。

每个资源都可以被注入到 bean 的单个字段或方法。请求将资源 java:comp/env/ com.example.MySessionBean/myDatabase 既注入到 setMyDatabase 方法又注入到 myDatabase 实例变量是错误的。但是注意，这个字段或方法可以请求注入不同名字（非缺省名字）的资源。通过显式指定资源的 JNDI 名字，单个资源可以被注入到多个类的多个字段或多个方法。

注释符也可以应用到 bean 类本身。这些注释符声明了 bean 环境中的条目，但不会造成资源被注入。反而，期望 bean 使用 EJBContext 的 lookup 方法或 JNDI API 的方法来查找条目。当注释符被应用到 bean 类时，必须显式指定 JNDI 名和

环境条目类型。

注释符可以出现在 bean 类或它的任何父类上。在继承层级中任意类上的资源注释符都定义了 bean 需要的资源。但是这样的资源注入遵循 java 语言中字段和方法的重载规则。重载父类方法的方法如果定义了资源，则将资源注入到该方法。重载方法可以请求与父类方法不同的资源，或它可以请求不要注入（尽管父类方法请求注入）。

另外，不可见或被子类隐藏的（与重载相反）字段或方法仍然可以请求注入。例如，这可以让私有字段作为注入的目标，且这个字段用于父类的实现，尽管子类不可见这个字段且不知道父类实现如何使用注入的资源。注意，在子类中声明和父类同名的字段总是会造成父类字段的隐藏。

16.2.3 注释符和部署描述

环境条目可以通过注释符被声明而不需要部署描述条目。环境条目也可以通过部署描述条目被声明而不需要任何注释符。同一个环境条目既可以同时使用注释符和部署描述条目来声明。在这种情况下，在部署描述条目中的信息用于覆盖注释符提供的信息。这可以被应用组装者用于覆盖由 Bean 提供者提供的信息。部署描述条目不应当用于请求向不作注入的字段或方法注入资源。

下面的规则应用于部署描述条目如何覆盖 Resource 注释符：

- 相关的部署描述条目基于与注释符一起使用的 JNDI 名来定位（缺省的或显式提供的）。
- 在部署描述中指定的类型必须是对应于字段或属性的类型，或在 Resource 注释符中指定的类型。
- 如果指定了部署描述，那么描述覆盖注释符中的描述元素。
- 如果指定了注入目标，那么这个目标必须和被注释的字段或属性方法的名字一致。
- 如果指定了 res-sharing-scope，那么覆盖注释符的 shareable 元素。一般情况下，应用组装者或部署者从来不应当改变元素的值，如果这样做很可能中断应用。

对覆盖环境条目值得限制依赖于环境条目的类型。

部署描述条目如何覆盖 EJB 注释符的规则在节 16.5 中描述。部署描述条目如何覆盖 PersistenceUnit 或 PersistenceContext 注释符的规则在节 16.10 和 16.11 中描述。用于 web 服务引用和部署描述条目如何覆盖 WebServiceRef 注释符的规则在 Java EE 规范的 web 服务中描述【31】。

16.3 EJB 角色的责任

本节描述了不同 EJB 角色关于环境条目规范和处理的责任。随后的章节描述了特定于存储在命名上下文中的不同类型对象的责任。

16.3.1 Bean 提供者的责任

Bean 提供者可以使用 java 语言的注释符或部署描述条目来请求从命名上下文中注入资源，或者声明命名上下文中需要的条目。Bean 提供者也可以使用 EJBContext 的 lookup 方法或 JNDI API 来访问命名上下文中的条目。部署描述条目也可以被 Bean 提供者用于覆盖由注释符提供的信息。

当使用直接 JNDI 接口是，企业 bean 实例通过使用无参构造器创建一个 javax.naming.InitialContext 对象，然后通过 InitialContext 来查找位于名字 java:comp/env 下的环境命名。

企业 bean 的环境条目被直接存储在环境命名上下文中，或者在它的直接或间接的子上下文中。

环境条目的值的类型是 Bean 提供者在元数据注释符或部署描述符中声明的 java 类型，或者是实例变量或与元数据注释符相关的方法的 setter 方法参数的类型。

16.3.2 应用组装者的责任

应用组装者可以修改 Bean 提供者设置的环境条目的值，也可以设置 Bean 提供者没有指定或初始化的环境条目值。应用组装者使用部署描述来覆盖 Bean

提供者的设置，无论这些值是由 Bean 提供者在部署描述中定义还是在使用注释符的源代码中定义。

16.3.3 部署者的责任

部署者必须保证有企业 bean 声明的所有环境条目的值必须被创建或/和设置成有意义的值。

部署者可以更改由 Bean 提供者或/和应用组装者设置的环境条目的值，而且必须设置还没有指定的环境条目的值。

由 bean 提供者或应用组装者提供的 description 元素可以帮助部署者完成这些工作。

16.3.4 容器提供者的责任

容器提供者有下面的责任：

- 为部署者提供部署工具来设置和更改环境条目的值。
- 实现 java:comp/env 环境命名上下文，并在运行时提供给 bean 实例。命名上下文必须包括由 bean 提供者声明的所有环境条目，以及由部署描述提供的或部署者设置的值。如果企业 bean 需要子上下文，则环境命名上下文必须允许部署者创建子上下文。
- 注入由注释符或部署描述指定的命名环境条目。
- 容器必须保证企业 bean 实例只能读环境变量。如果修改环境命名上下文和它的子上下文，那么容器必须从 javax.naming.Context 接口的所有方法中抛出 javax.naming.OperationNotSupportedException。

16.4 简单环境条目

简单环境条目是一个配置参数，它用于客户化企业 bean 的业务逻辑。环境条目值可以是下面的类型：String，Character，Integer，Boolean，Double，Byte，Short，Long 和 Float。

下面的章节描述了每个 EJB 角色的责任。

16.4.1 Bean 提供者的责任

本节描述了 bean 环境的 bean 提供者视图，并定义了他或她的责任。第一节描述了用于注入简单环境条目的注释符；第二节描述了访问简单环境条目的 API；第三节描述了在部署描述中声明环境条目的语法。

16.4.1.1 使用注释符注入简单环境条目

Bean 提供者使用 Resource 注释符来注释 bean 类的字段或方法，它们作为简单环境条目的注入目标。环境条目的名字和节 16.2.2 中描述的一样；类型是节 16.4 中描述的类型。注意容器将根据与之匹配的用于注入字段或方法的原始类型来拆箱环境条目。不能指定 Resource 注释符的 authenticationType 和 shareable 元素；简单环境条目不是可共享的且不要求授权。

下面的代码例子解释了企业 bean 如何使用注入环境条目的注释符。

```
@Stateless public class EmployeeServiceBean
implements EmployeeService {
...
// The maximum number of tax exemptions, configured by Deployer
@Resource int maxExemptions;
// The minimum number of tax exemptions, configured by Deployer
@Resource int minExemptions;
public void setTaxInfo(int numberOfExemptions,...)
throws InvalidNumberOfExemptionsException {
...
// Use the environment entries to customize business logic.
if (numberOfExemptions > maxExemptions ||
numberOfExemptions < minExemptions)
throw new InvalidNumberOfExemptionsException();
}
    }
```

16.4.1.2 访问简单环境条目的编程接口

除了使用上述的注入以外，企业 bean 可以动态获取环境条目。这可以通过 EJBContext 的 lookup 方法或直接使用 JNDI 接口来实现。这些环境条目是由 Bean 提供者通过在 bean 类上的注释或部署描述来声明。

当直接使用 JNDI 接口时，bean 实例通过使用 javax.naming.InitialContext 的无参构造器来创建 InitialContext 对象，然后通过这个对象查找 java:comp/env 名字下的命名环境。Bean 的环境条目直接存储在环境命名上下文中，或它的直接或间接子上下文中。

下面的样例代码解释了企业 bean 如何直接使用 JNDI API 来访问它的环境条目。在这个例子中，被访问条目的名字由部署描述订阅，正如在节 16.4.1.3 中的例子展示的一样。

```
@Stateless public class EmployeeServiceBean
implements EmployeeService {
...
public void setTaxInfo(int numberOfExemptions, ...)
throws InvalidNumberOfExemptionsException {
...
// Obtain the enterprise bean's environment naming context.
Context initCtx = new InitialContext();
Context myEnv = (Context)initCtx.lookup("java:comp/env");
// Obtain the maximum number of tax exemptions
// configured by the Deployer.
Integer maxExemptions =
(Integer)myEnv.lookup("maxExemptions");
// Obtain the minimum number of tax exemptions
// configured by the Deployer.
Integer minExemptions =
(Integer)myEnv.lookup("minExemptions");
// Use the environment entries to customize business logic.
if (numberOfExeptions > maxExemptions ||
numberOfExemptions < minExemptions)
throw new InvalidNumberOfExemptionsException();
// Get some more environment entries. These environment
// entries are stored in subcontexts.
String val1 = (String)myEnv.lookup("foo/name1");
Boolean val2 = (Boolean)myEnv.lookup("foo/bar/name2");
```

```
// The enterprise bean can also lookup using full pathnames.
Integer val3 = (Integer)
initCtx.lookup("java:comp/env/name3");
Integer val4 = (Integer)
initCtx.lookup("java:comp/env/foo/name4");
...
}
    }
```

16.4.1.3 在部署描述中声明简单环境条目

Bean 提供者必须声明从企业 bean 代码中访问的所有简单环境条目。这些环境条目通过在 bean 类代码中使用注释符或在部署描述中使用 env-entry 元素来声明。

每个 env-entry 描述了一个环境条目。它由一个可选的环境条目描述、相对于 java:comp/env 的环境条目名称、期望的环境条目值的类型（也就是，从 EJBContext 或 JNDI 的 lookup 方法返回的对象的类型）和一个可选的环境条目值组成。

环境条目的范围是企业 bean，它的部署描述元素要包含给定的 env-entry 元素。这意味着在运行时其他企业 bean 是不能获取这些环境条目的，且其他企业 bean 也可以使用相同的 env-entry-name 而不会引起命名冲突。

如果 Bean 提供者使用 env-entry-value 元素为环境条目提供了它的值，那么这个值可以在以后被应用组装者或部署者改变。这个值必须是一个有效的字符串（有效指的是可以使用类型的带单个 String 参数的构造器来构造出相应的类型），或者对于 java.lang.Character 来说，是一个字符。

下面的例子是声明由 EmployeeServiceBean 使用的环境条目，它的代码在前面的章节中已经解释过了。

```
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
...
```

```
<env-entry>
<description>
The maximum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>maxExemptions</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>15</env-entry-value>
</env-entry>
<env-entry>
<description>
The minimum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>minExemptions</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>1</env-entry-value>
</env-entry>
<env-entry>
<env-entry-name>foo/name1</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>value1</env-entry-value>
</env-entry>
<env-entry>
<env-entry-name>foo/bar/name2</env-entry-name>
<env-entry-type>java.lang.Boolean</env-entry-type>
<env-entry-value>true</env-entry-value>
</env-entry>
<env-entry>
<description>Some description.</description>
<env-entry-name>name3</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
<env-entry>
<env-entry-name>foo/name4</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>10</env-entry-value>
</env-entry>
...
</session>
</enterprise-beans>

    ...
```

环境条目的注入也可以使用部署描述来指导，而不需要 java 语言的注释符。下面是与节 16.4.1.1 中的例子对应的环境条目的声明。

```
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
...
<env-entry>
<description>
The maximum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>
com.wombat.empl.EmployeeService/maxExemptions
</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>15</env-entry-value>
<injection-target>
<injection-target-class>
com.wombat.empl.EmployeeServiceBean
</injection-target-class>
<injection-target-name>
maxExemptions
</injection-target-name>
</injection-target>
</env-entry>
<env-entry>
<description>
The minimum number of tax exemptions
allowed to be set.
</description>
<env-entry-name>
com.wombat.empl.EmployeeService/minExemptions
</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>1</env-entry-value>
<injection-target>
<injection-target-class>
com.wombat.empl.EmployeeServiceBean
</injection-target-class>
<injection-target-name>
```

```
minExemptions
</injection-target-name>
</injection-target>
</env-entry>
...
</session>
</enterprise-beans>
    ...
```

除了在代码中指定缺省值外，将字段作为注入目标通常都是方便的。如下例解释的一样：

```
// The maximum number of tax exemptions, configured by the Deployer.
   @Resource int maxExemptions = 4; // defaults to 4
```

为了支持这种情况，如果应用组装者或部署者已经指定了一个值覆盖了缺省值，那么容器必须只注入环境条目的值。当注入目标被指定时，在部署描述中的 env-entry-value 元素是可选的。如果没有指定，则不注入值。另外，如果没有指定，命名的资源在命名上下文中不被初始化，且显式查找命名资源将会失败。

16.4.2 应用组装者的责任

应用组装者可以更改由 Bean 提供者设置的简单环境条目的值，以及可以设置那么些 Bean 提供者没有指定初始值的环境条目的值。应用组装者可以使用部署描述来覆盖由 Bean 提供者的设置，无论是部署描述中还是使用注释符进行的设置。

16.4.3 部署者的责任

部署者必须保证由企业 bean 声明的所有简单环境条目的值都是有意义的值。

部署者可以修改由 Bean 提供者和/或应用组装这设置的环境条目的值，且必须设置没有指定值的环境条目的值。

由 Bean 提供者或应用组装这提供的 description 元素可以帮助部署者完成这项工作。

16.4.4 容器提供者的责任

容器提供者有下述的责任：

- 为部署者提供部署工具来设置和更改环境条目的值。
- 实现 java:comp/env 环境命名上下文，并在运行时提供给 bean 实例。命名上下文必须包括由 bean 提供者声明的所有环境条目，以及由部署描述提供的或部署者设置的值。如果企业 bean 需要子上下文，则环境命名上下文必须允许部署者创建子上下文。
- 注入由注释符或部署描述指定的命名环境条目。
- 容器必须保证企业 bean 实例只能读环境变量。如果修改环境命名上下文和它的子上下文，那么容器必须从 javax.naming.Context 接口的所有方法中抛出 javax.naming.OperationNotSupportedException。

16.5 EJB 引用

本节描述了编程和部署描述接口，这些接口可以让Bean提供者使用称为EJB引用“逻辑”名来引用其他企业 bean 的业务接口或 home。EJB 引用在企业 bean 的环境中是特殊的条目。部署者将 EJB 引用绑定到目标操作环境中企业 bean 的业务接口或 home（根据情况）。

16.5.1 Bean 提供者的责任

本节描述Bean提供者在EJB方面的视图和责任。第一节描述了用于注入EJB引用的注释符；第二节描述了用于访问 EJB 引用的 API；第三节描述了在部署描述中声明 EJB 引用的语法。

16.5.1.1 EJB 引用的注入

Bean 提供者在 bean 类作为 EJB 引用注入目标的字段或 setter 属性方法上使用 EJB 注释符来注入 EJB 引用。引用可以是会话 bean 的业务接口，或者是会话

bean 或实体 bean 的本地 home 接口或远程 home 接口。

下面的例子解释了企业 bean 如何使用 EJB 注释符来引用另一个企业 bean。这个企业 bean 引用在引用它的 bean 的命名上下文中的名字是 java:comp/env/com.acme.example.ExampleBean/myCart，其中 ExampleBean 是引用 bean 的类名，com.acme.example 是它的包。引用的目标必须由部署者解决。

```
package com.acme.example;
@Stateless public class ExampleBean implements Example {
...
@EJB private ShoppingCart myCart;
...
}
```

下面的代码解释了 EJB 注释符所有可移植元素的用法。在这种情况下，企业 bean 引用在引用它的 bean 的命名上下文中的名字是 java:comp/env/ejb/shopping-cart。这个引用关联到名字是 cart1 的 bean。

```
@EJB(
name="ejb/shopping-cart",
beanInterface=ShoppingCart.class,
beanName="cart1",
description="The shopping cart for this application"
)
private ShoppingCart myCart;
```

如果 ShoppingCartbean 用 EJB2.1 客户端视图实现，那么 EJB 引用是 bean 的 home 接口。例如：

```
@EJB(
name="ejb/shopping-cart",
beanInterface=ShoppingCartHome.class,
beanName="cart1",
description="The shopping cart for this application"
)
private ShoppingCartHome myCartHome;
```

16.5.1.2 EJB 引用编程接口

Bean 提供者可以按下述方式使用 EJB 引用来定位其他企业 bean 的业务接口或 home 接口：

- 在企业 bean 环境中分配一个指向引用的条目。（参见节 16.5.1.3 了解如何在部署描述中声明 EJB 引用）
- EJB 规范推荐但不要求所有对其他企业 bean 的引用都组织在 bean 环境的 ejb 子上下文中（也就是，java:comp/env/ejb JNDI 上下文）。注意，由注释符声明的企业 bean 引用缺省情况下不在任何子上下文中。
- 在企业 bean 的环境中使用 EJBContext 的 lookup 方法或 JNDI API 查找被引用企业 bean 的业务接口或 home 接口。

下面的例子解释了企业 bean 如何使用 JNDI API 来定位另外一个企业 bean 远程 home 接口。

```
@EJB(name="ejb/EmplRecord", beanInterface=EmployeeRecordHome.class)
@Stateless public class EmployeeServiceBean
implements EmployeeService {
public void changePhoneNumber(...) {
...
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();
// Look up the home interface of the EmployeeRecord
// enterprise bean in the environment.
Object result = initCtx.lookup(
"java:comp/env/ejb/EmplRecord");
// Convert the result to the proper type.
EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
javax.rmi.PortableRemoteObject.narrow(result,
EmployeeRecordHome.class);
...
}
}
```

在这个例子中，EmployeeServiceBean 的 bean 提供者分配了一个环境条目 ejb/EmplRecord 作为 EJB 引用的名字，它指向另一个企业 bean 的远程 home 接口。

16.5.1.3 在部署描述中声明 EJB 引用

尽管 EJB 引用是企业 bean 环境的一个条目，但 Bean 提供者不能使用 env-entry 元素来声明它。反而，Bean 提供者必须用 ejb-ref 和 ejb-local-ref 元素声

明所有的 EJB 引用。这可以让 ejb-jar 消费者（也就是应用组装者或部署者）发现所有企业 bean 引用的 EJB 引用。部署描述条目也可以用于指定 EJB 引用的注入。

每个 ejb-ref 和 ejb-local-ref 元素都描述了引用企业 bean 对被引用企业 bean 的接口要求。ejb-jar 元素用于引用一个通过它的远程业务接口或远程 home 和组件接口来访问它的企业 bean。ejb-local-ref 元素用于引用一个企通过它的本地业务接口或本地 home 和组件接口来访问它的企业 bean。

ejb-ref 元素包含了 description，ejb-ref-name，ejb-ref-type，home 和 remote 元素。

ejb-local-ref 元素包含了 description，ejb-ref-name，ejb-ref-type，local-home 和 local 元素。

ejb-ref-name 元素指定 EJB 引用的名字：它的值是企业 bean 代码中使用的环境条目的名字。必须指定 ejb-ref-name。可选的 ejb-ref-type 指定了企业 bean 的期望类型：它的值是 Entity 或 Session。home 和 remote 或 local-home 和 local 元素指定了被引用企业 bean 接口的 java 类型。如果是对 EJB2.1 远程客户端视图的引用，那么需要 home 元素。remote 元素根据是对 EJB3.0 远程客户端视图还是 EJB2.1 远程客户端视图的引用分别指向业务接口类型或组件接口。否则，ejb-local-ref 的 local 元素根据是对 EJB3.0 远程客户端视图还是 EJB2.1 本地客户端视图分别指向业务接口类型或组件接口。

EJB 引用的范围是企业 bean，它的声明包含了 ejb-ref 或 ejb-local-ref 元素。这意味着在运行时其他企业 bean 不会获得这些 EJB 引用，且其他企业 bean 可以定义 ejb-ref 和/或 ejb-local-ref 元素中使用相同的 ejb-ref-name 而不会引起命名冲突。

下面的例子解释了在部署描述中声明 EJB 引用。

```
...
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
...
```

```
<ejb-ref>
<description>
This is a reference to an EJB 2.1 entity bean that
encapsulates access to employee records.
</description>
<ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.wombat.empl.EmployeeRecordHome</home>
<remote>com.wombat.empl.EmployeeRecord</remote>
</ejb-ref>
<ejb-local-ref>
<description>
This is a reference to the local business interface
of an EJB 3.0 session bean that provides a payroll
service.
</description>
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
<local>com.aardvark.payroll.Payroll</local>
</ejb-local-ref>
<ejb-local-ref>
<description>
This is a reference to the local business interface
of an EJB 3.0 session bean that provides a pension
plan service.
</description>
<ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
<local>com.wombat.empl.PensionPlan</local>
</ejb-local-ref>
...
</session>
...
</enterprise-beans>
    ...
```

16.5.2 应用组装者的责任

应用组装者可以使用 ejb-link 元素将一个 EJB 引用关联到目标企业 bean。

应用组装者在两个企业 bean 间按下述方式指定关联：

- 应用组装者使用引用企业 bean 的 ejb-ref 或 ejb-local-ref 元素的可选的

ejb-link 元素来建立关联。ejb-link 元素的值是目标企业 bean 的名字。(这个名字和 bean 类中元数据注释符中定义的名字（或缺省的名字）或目标企业 bean 的 ejb-name 元素的名字一样)。目标企业 bean 可以是和引用应用组件位于同一个 JavaEE 应用的任何 ejb-jar 文件。

- 作为选项，为了避免在整个 JavaEE 中有唯一名字而重命名企业 bean，应用组装者可以在引用应用组件的 ejb-link 元素中使用下面的语法。应用组装者指定包含被引用企业 bean 的 ejb-jar 的路径名，然后用#做分割符附上目标 bean 的 ejb-name。路径名是相对于引用应用组件 jar 文件的。以这种方式，在应用组装者不能改变 ejb-name 时可以唯一标识多个具有相同 ejb-name 的 bean（注：bean 提供者也可以使用在 EJB 注释符的 beanName 元素中使用这个语法)。
- 应用组装者必须保证目标企业 bean 是和声明的 EJB 引用类型兼容。者意味着目标企业 bean 必须是在 ejb-ref-type 元素（如果出现）中声明的类型之一，且目标企业 bean 的业务接口或 home 和组件接口必须和在 EJB 引用中声明的接口类型兼容。

下面解释了在部署描述中的 ejb-link。

```
...
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
...
<ejb-ref>
<ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.wombat.empl.EmployeeRecordHome</home>
<remote>com.wombat.empl.EmployeeRecord</remote>
<ejb-link>EmployeeRecord</ejb-link>
</ejb-ref>
...
</session>
...
<entity>
<ejb-name>EmployeeRecord</ejb-name>
```

```
<home>com.wombat.empl.EmployeeRecordHome</home>
<remote>com.wombat.empl.EmployeeRecord</remote>
...
</entity>
...
</enterprise-beans>
    ...
```

应用组装者使用 ejb-link 元素来声明在 EmployeeService 企业 bean 中声明的 EJB 引用 EmployeeRecord 已经被关联到 EmployeeRecord 企业 bean。

下面的例子解释了使用 ejb-link 元素来声明一个对 ProductEJB 企业 bean 的引用，ProductEJB 与引用 bean 位于同一个 JavaEE 应用单元但在不同的 ejb-jar 文件。

```
<entity>
...
<ejb-name>OrderEJB</ejb-name>
<ejb-class>com.wombat.orders.OrderBean</ejb-class>
...
<ejb-ref>
<ejb-ref-name>ejb/Product</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.acme.orders.ProductHome</home>
<remote>com.acme.orders.Product</remote>
<ejb-link>../products/product.jar#ProductEJB</ejb-link>
</ejb-ref>
...
</entity>
```

下面的例子解释了使用 ejb-link 元素来声明一个企业 bean 引用 ShoppingCart 企业 bean，ShoppingCart 企业 bean 与引用企业 bean 位于同一个 JavaEE 应用单元，但在不同的 ejb-jar 文件。这个引用起初是声明在使用注释符的 bean 代码中。应用组装者只提供对这个 bean 的链接。

```
...
<ejb-ref>
<ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
<ejb-link>../products/product.jar#ShoppingCart</ejb-link>
</ejb-ref>
```

16.5.2.1 重载规则

下面的规则应用于部署描述条目如何重载 EJB 注释符：

- 相关的部署描述条目基于与注释一起使用的 JNDI 名字（缺省的或显式提供的）来定位。
- 在部署描述中通过 remote，local，remote-home 或 local-home 元素指定的类型和 ejb-link 元素引用的 bean 必须和字段或属性的类型一致，或者与由 EJB 注释符的 beanInterface 元素指定的类型一致。
- 如果指定了描述符，则它重载注释符的描述元素。
- 如果指定了注入目标，则它的名字必须和被注释的字段或属性方法的名字一致。

16.5.3 部署者的责任

部署者有以下责任：

- 部署者必须保证所有声明的EJB引用都绑定到操作环境中企业bean的业务接口或 home 上。例如，部署者可以使用 JNDI 的 LinkRef 机制来创建对目标企业 bean 的 JNDI 名字的抽象链接。
- 部署者必须保证目标企业 bean 的类型与 EJB 引用声明的类型兼容。这意味着目标企业 bean 必须是 EJB 注释符、ejb-ref-type 元素（如果指定）声明的类型，且目标企业 bean 的业务接口和/或 home 和组件接口的类型必须是与注入目标或在 EJB 引用中声明的接口类型兼容的 java 类型。

16.5.4 容器提供者的责任

容器提供者必须提供让部署者执行前面章节描述的任务的工具。由 EJB 容器提供者提供的部署工具必须能够处理在 ejb-ref 和 ejb-local-ref 元素中提供的信息。

工具至少能够：

- 通过绑定 EJB 引用到目标企业 bean 的业务接口或 home 接口来保存注释

符或 ejb-link 元素中的应用组装信息。

- 将未解决的 EJB 引用通知给部署者，并允许他或她通过绑定 EJB 引用到指定的兼容目标 bean 来解决 EJB 引用。

16.6 Web 服务引用

Web 服务引用可以让 Bean 提供者引用外部的 web 服务。Web 服务引用在企业 bean 环境中是一个特殊条目。部署者将 web 服务引用绑定到目标操作环境的 web 服务类或接口。

Web 服务引用的规范和它们的用法在 JAX-WS【32】和用于 Java EE 规范的 web 服务【31】中描述。

Web 服务的范围是企业 bean，企业 bean 的定义包含了 WebServiceRef 注释符或它的部署描述声明中包含了 servcie-ref 元素。EJB 规范推荐但不要求所有对 web 服务的引用都组织在 bean 环境的 service 子上下文中（也就是在 java:comp/env/service JNDI 上下文中）。

16.7 资源管理连接工厂引用

资源管理连接工厂是用于创建对资源管理器连接的对象。例如，一个实现了 javax.sql.DataSource 接口的对象就是狗仔 java.sql.Connection 对象的资源管理器连接工厂，Connection 对象实现了对数据库管理系统的连接。

本节描述了在企业 bean 代码中使用称为“资源管理器连接工厂引用”的逻辑名称引用资源工厂的元数据注释符和部署描述元素。资源管理器连接工厂引用是企业 bean 环境中的一个特殊条目。部署者将资源管理器连接工厂引用绑定到容器中配置的真正的资源管理器连接工厂。因为这些资源管理器连接工厂可以让容器影响资源管理，通过资源管理器连接工厂引用获得的连接称为“受管理资源”（例如，这些资源管理器连接工厂运行容器实现连接池，并自动征收带事务的连接）。

16.7.1 Bean 提供者的责任

本节描述定位资源工厂的 Bean 提供者视图和定义他或她的责任。第一节描述用于注入对资源管理器连接工厂引用的注释符；第二节描述用于访问资源管理器连接引用的 API；第三节描述用于在部署描述中声明资源管理器连接引用的语法。

16.7.1.1 资源管理器连接工厂引用的注入

企业 bean 的字段或方法可以用 Resource 注释符来注释。工厂的名字和类型正如前面节 16.2.2 描述的一样。Resource 注释符的 authenticationType 和 shareable 元素可以用于控制为资源授权的类型和从工厂获取的连接的共享性，正如在后面章节中描述的一样。

下面的代码例子解释企业 bean 如何使用这些注释符来声明资源管理器连接工厂引用。

```
//The employee database.
@Resource javax.sql.DataSource employeeAppDB;
...
public void changePhoneNumber(...) {
...
// Invoke factory to obtain a resource. The security
// principal for the resource is not given, and
// therefore it will be configured by the Deployer.
java.sql.Connection con = employeeAppDB.getConnection();
...
}
```

16.7.1.2 用于资源管理器连接工厂引用的编程接口

Bean 提供者必须按下述方式使用资源管理器连接工厂引用来获取对资源的引用：

- 在企业 bean 环境中为资源管理器连接工厂引用分配一个条目。（参见节 16.7.1.3 了解如何在部署描述中声明资源管理器连接工厂引用）
- EJB 规范推荐但不要求所有的资源管理器连接工厂引用都组织在 bean 环境的子上下文中，为每个资源管理器类型使用不同的子上下文。例如，所有的 JDBC DataSource 引用可能声明在 java:comp/env/jdbc 子上下文，但所有的 JMS 连接工厂声明在 java:comp/env/jms 子上下文中。同样，所有的 JavaMail 连接工厂可能声明在 java:comp/env/mail 子上下文中，但所有的 RUL 连接工厂声明在 java:comp/env/url 子上下文中。注意，通过注释符声明的资源管理器连接工厂引用缺省情况下不出现在子上下文中。
- 使用 EJBContext 的 lookup 方法或 JNDI API 在企业 bean 环境中查找资源管理器连接工厂对象。
- 调用资源管理器连接工厂相应的方法来获取对资源的连接。工厂方法根据资源类型的不同而不同。多次调用工厂对象可能得到多个连接。

Bean 提供者可以控制从资源管理器连接工厂得到的连接的共享性。缺省情况下，连接是被多个处于同一个事务上下文使用同一个资源的企业 bean 共享的。Bean 提供者可以通过设置 shareable 为 false 或 res-sharing-scop 部署描述元素为 Unshareable 来指定这个连接不被共享。连接的共享性可以让容器优化连接的使用，并且让容器能够使用本地事务优化。

Bean 提供者有两个选择将权限和资源管理器访问关联起来：

- 可以让部署者设置权限映射或资源管理器签名信息。在这种情况下，企业 bean 代码调用资源管理器连接工厂的没有相关安全参数的方法，。
- 从 bean 代码中签名资源管理者。在这种情况下，企业 bean 调用相应的资源管理器连接工厂的将签名信息作为参数的方法。

Bean 提供者使用 authenticationType 注释符元素或 res-auth 部署描述元素来声明使用哪个授权方法。

我们期望大多数企业 bean 使用第一种形式（也就是，让部署者设置资源管理器签名信息）。

下面的代码解释了使用 EJBContext 的 lookup 方法获取 JDBC 连接。

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean
implements EmployeeService {
@Resource SessionContext ctx;
public void changePhoneNumber(...) {
...
// use context lookup to obtain resource manager
// connection factory
javax.sql.DataSource ds = (javax.sql.DataSource)
ctx.lookup("jdbc/EmployeeAppDB");
// Invoke factory to obtain a connection. The security
// principal is not given, and therefore
// it will be configured by the Deployer.
java.sql.Connection con = ds.getConnection();
...
}
}
```

下面的代码解释了直接使用 JNDI API 获取 JDBC 连接。

```
@Resource(name="jdbc/EmployeeAppDB", type=javax.sql.DataSource)
@Stateless public class EmployeeServiceBean
implements EmployeeService {
EJBContext ejbContext;
public void changePhoneNumber(...) {
...
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// perform JNDI lookup to obtain resource manager
// connection factory
javax.sql.DataSource ds = (javax.sql.DataSource)
```

```
initCtx.lookup("java:comp/env/jdbc/EmployeeAppDB");
// Invoke factory to obtain a connection. The security
// principal is not given, and therefore
// it will be configured by the Deployer.
java.sql.Connection con = ds.getConnection();
...
}
}
```

16.7.1.3 在部署描述中声明资源管理器连接工厂引用

尽管资源管理器连接工厂引用是企业 bean 环境的一个条目，但是 Bean 提供者不能使用 env-entry 元素来声明它。

反而，如果没有使用元数据注释符，那么 Bean 提供者必须在部署描述中使用 resource-ref 元素来声明所有的资源管理器连接工厂引用。这可以让 ejb-jar 消费者（也就是应用组装者或部署者）来发现由企业 bean 使用的所有资源管理器连接工厂引用。部署描述条目也可以用于指定要注入到 bean 的资源管理器连接工厂引用。

每个 resource-ref 元素都描述了一个单个的资源管理器连接工厂引用。resource-ref 元素由 description 元素、必选的 res-ref-name 元素和可选的 res-type、res-auth 和 res-sharing-scope 元素组成。res-ref-name 元素包含了在企业 bean 代码中使用的环境条目的名字。环境条目的名字相对于 java:comp/env 上下文（例如，名字应当是 jdbc/EmployeeAppDB 而不是 java:comp/env/jdbc/EmployeeAppDB）。res-type 元素包含企业 bean 代码期望的资源管理器连接工厂的 java 类型。如果为资源指定了注入目标，则 res-type 是可选的。在这种情况下，res-type 缺省是注入目标的类型。res-auth 元素声明了企业 bean 代码是程序中执行资源管理器签名还是容器用由部署者提供的权限映射信息签名资源管理器。Bean 提供者通过设置 res-auth 的值来声明签名由应用还是容器负责。如果没有指定 res-auth，假定使用容器签名。res-sharing-scope 元素声明对通过给定的资源管理器连接工厂引

用获得的资源管理器的连接是否可以被共享。res-sharing-scope 元素的值是 Shareable 活 Unshareable。如果没有指定 res-sharing-scope 元素，则假定连接是可共享的。

资源管理器连接工厂引用的范围是声明中包含了 resource-ref 元素的企业 bean。这意味着资源管理器连接工厂引用在运行时不能被其他企业 bean 获得，且其他企业 bean 可以使用相同的 res-ref-name 定义 resource-ref 元素而不会造成名字冲突。

类型声明可以让部署者标识资源管理器连接工厂的类型。

注意，被指定的类型是资源工厂的 Java 类型，不是资源的 Java 类型。

下面的例子是声明 EmployeeService 使用的资源管理器连接工厂引用。

```
...
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
...
<resource-ref>
<description>
A data source for the database in which
the EmployeeService enterprise bean will
record a log of all transactions.
</description>
<res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

...

</session>

</enterprise-beans>

...

下面的例子解释了 JMS 资源管理器连接工厂引用的声明。

...

<enterprise-beans>

<session>

...

<resource-ref>

<description>

A queue connection factory used by the

MySession enterprise bean to send

notifications.

</description>

<res-ref-name>jms/qConnFactory</res-ref-name>

<res-type>javax.jms.QueueConnectionFactory</res-type>

<res-auth>Container</res-auth>

<res-sharing-scope>Unshareable</res-sharing-scope>

</resource-ref>

...

</session>

</enterprise-beans>

...

16.7.1.4 标准的资源管理器连接工厂类型

Bean 提供者必须使用 javax.sql.DataSource 资源管理器连接工厂类型来获取 JDBC 连接，使用 javax.jms.ConnectionFactory，javax.jms.QueueConnectionFactory

或 javax.jms.TopicConnectionFactory 来获取 JMS 连接。

Bean 提供者必须使用 javax.mail.Session 资源管理器连接工厂类型来获取 JavaMail 连接，使用 java.net.URL 资源管理器连接工厂来获取 URL 连接。

推荐 Bean 提供者在 java:comp/env/jdbc 子上下文中明明 JDBC 数据源，在 java:comp/env/jms 子上下文中命名 JMS 连接工厂，在 java:comp/env/mail 子上下文中命名所有的 JavaMail 连接工厂，在 java:comp/env/url 子上下文中命名所有的 URL 连接工厂。注意，通过注释符声明的资源管理器连接工厂引用缺省情况下不出现在这些子上下文中。

连接器价格【15】允许企业 bean 事业在本节中描述的 API 来获取资源对象，这些对象提供了访问其他的后台系统。

16.7.2 部署者的责任

部署者使用部署工具来将资源管理器连接工厂引用绑定到目标操作环境中配置的真正的资源工厂上。

部署者必须为在元数据注释符或部署描述中的每个资源管理器连接工厂引用执行下面的任务：

- 将资源管理器连接工厂引用绑定到操作环境中的资源管理器连接工厂。例如，部署者可以使用 JNDI 的 LinkRef 机制来创建一个抽象连接，这个连接连接到真正的资源管理器连接工厂的 JNDI 名字。资源管理器连接工厂类型必须和在源代码或 res-type 元素中声明的类型兼容。
- 为资源管理器提供打开和管理资源的附加配置信息。配置机制是资源管理器特有的，这超出了本规范的范围。
- 如果 Resource 注释符 authenticationType 元素的值是 AuthenticationType.CONTAINER 或者部署描述 res-auth 元素是 Container，那么部署者有责任为资源管理器配置签名信息。这随着 EJB 容器和资源管理器的不同而不同；它已经超出了本规范的范围。

例如，如果必须在企业 bean 应用层次的安全域和权限域与资源管理器的安全域和权限域间建立映射，那么部署者或系统管理员必须定义这种映射。建立这

种映射随着 EJB 容器和资源管理器的不同而不同；它超出了本 EJB 规范的范围。

16.7.3 容器提供者的责任

EJB 容器提供者的责任如下：

- 为部署者提供执行前面章节描述的任务的部署工具。
- 为配置在 EJB 容器中的资源管理器提供资源管理器连接工厂类的实现。
- 如果 Bean 提供者设置 Resource 注释符的 authenticationType 元素为 AuthenticationType.APPLICATION 或 res-auth 部署描述条目是 Application，那么容器必须允许 bean 可以使用资源管理器的 API 执行显式的签名。
- 如果 Bean 提供者设置 Resource 注释符的 shareable 元素为 false 或者设置资源管理器连接工厂引用的 res-sharing-scope 部署描述条目为 Unshareable，那么容器不能共享从资源管理器连接工厂引用获得的连接（从通过不同的资源管理器连接工厂引用得到的同一个资源管理器连接工厂得到的连接可以被共享）。如果 Bean 提供者这是资源管理器连接工厂引用的 res-sharing-scope 为 Shareable 或没有指定 res-sharing-scope，则容器必须根据在【12】中描述的要求共享资源管理器连接工厂得到的连接。
- 容器必须提供工具，这些工具可以让部署者为资源管理器引用的注释符元素 authenticalionType 为 AuthenticationType.CONTAINER 或 res-auth 部署描述条目为 Container 的资源管理器引用设置资源管理器的签名信息。最小要求是部署者必须能够为每个资源管理器连接工厂引用指定用户名/密码信息，且容器必须能够在通过调用资源管理器连接工厂获取对资源的连接时使用用户名/密码进行用户授权。

尽管 EJB 规范不要求，我们期望容器支持覆盖应用服务器和资源管理器的单一签名机制。容器将允许部署者设置资源管理器，以便 EJB 调用者权限能够被传递（直接或通过权限映射）到资源管理器，如果应用要求。

尽管 EJB 规范不要求，但大多数 EJB 容器提供者也提供了下面的特性：

- 允许系统管理员为 EJB 容器增加、删除和配置资源管理器的工具。
- 为企业 bean 池化资源连接的机制，否则由容器管理资源的使用。池对企业 bean 必须是透明的。

16.7.4 系统管理员的责任

系统管理员通常有下面的责任：

- 在 EJB 服务器环境中增加、删除和配置资源管理器。

在某些场景中，这些任务可由部署者执行。

16.8 资源环境引用

本节描述可以让 Bean 提供者通过使用称为资源环境引用的“逻辑”名称来引用与资源（例如，连接器 CCI InteractionSpec 实例）关联的受管理对象的编程与部署描述接口。资源环境引用是企业 bean 环境中的特殊条目。部署者将资源环境引用绑定到目标操作环境的受管理对象上。

16.8.1 Bean 提供者的责任

本节描述在资源环境引用方面 Bean 提供者的视图和责任。

16.8.1.1 资源环境引用的注入

Bean 的字段或方法可以用 Resource 注释符来要求注入资源环境引用。资源环境引用的名字和类型在 16.2.2 节中描述。不能指定 Resource 注释符的 authenticationType 和 shareable 元素；资源环境条目不是可共享的且不要求授权。使用 Resource 注释符来声明一个资源环境引用不同于使用 Resource 注释符来声明简单环境引用，因为资源环境引用的类型不是用于简单环境引用的 java 语言类型。

16.8.1.2 资源环境引用编程接口

Bean 提供者必须使用资源环境引用来定位与资源关联的受管理对象，如下：

- 在企业 bean 环境中分配一个资源环境引用的条目。（参见节 16.8.1.3 了解如何在部署描述中声明资源环境引用）
- EJB 规范推荐但不要求所有的资源环境引用被组织在 bean 环境中用于资源类型的子上下文中。注意，通过注释符声明的资源环境引用将不出现在任何子上下文中。
- 使用 EJBContext 的 lookup 方法或 JNDI API 在企业 bean 环境查找受管理对象。

16.8.1.3 在部署描述中声明资源环境引用

尽管资源环境引用是企业 bean 环境中的一个条目，Bean 提供者不能使用 env-entry 元素来声明它。反而，Bean 提供者必须在 bean 源代码中使用注释符或在部署描述中使用 resource-env-ref 元素来声明所有对与资源关联的受管理对象的引用。这可以让 ejb-jar 消费者发现企业 bean 使用的所有资源环境引用。部署描述条目也可以用于指定在 bean 上进行资源环境引用的注入。

每个 resource-env-ref 元素描述了企业 bean 对受管理对象的需求。resource-env-ref 元素包含了可选的 description 和 resource-env-ref-type 元素，以及必选的 resource-env-ref-name 元素。如果指定了资源环境引用的注入目标，则 resource-env-ref-type 元素是可选的；在这种情况下，resource-env-ref-type 缺省是注入目标的类型。

resource-env-ref-name 元素指定了资源环境引用的名字：它的值是在企业 bean 代码中使用的环境条目的值。环境条目的名字相对于 java:comp/env 上下文。resource-env-ref-type 元素指定了被引用对象的类型。

资源环境引用的范围是声明中包含了 resource-env-ref 元素的企业 bean。这意味着资源环境引用在运行时不会被其他企业 bean 获得，且其他企业 bean 可以使用相同的 resource-env-ref-name 定义 resource-env-ref 而不会造成名字冲突。

16.8.2 部署者的责任

部署者的责任如下：

- 部署者必须保证将所有声明的资源环境引用都绑定到目标环境中存在的受管理对象上。例如，部署者可以使用 JNDI 的 LinkRef 机制来创建一个指向目标对象真正 JNDI 名称的抽象链接。
- 部署者必须保证目标对象的类型和声明资源环境引用的类型相兼容。这意味着目标对象必须是在 Resource 注释符或 resource-env-ref-type 元素中声明的类型。

16.8.3 容器提供者的责任

容器提供者必须提供可以让部署者执行前面章节描述的任务的工具。由容器提供者提供的部署工具必须能够处理在类文件的注释符和部署描述的 resource-env-ref 元素中提供的信息。

至少，工具必须能够通知部署者为解决的资源环境引用，并且允许他或她通过将未处理的引用绑定到环境中兼容的目标对象上来解决资源环境引用。

16.9 消息目的地引用

16.10 持久化单元引用

16.11 持久化上下文引用

16.12 UserTransaction 引用

容器必须使允许使用 UserTransaction 接口的企业 bean 可以获得这个接口（只有使用 bean 管理事务分割的会话和消息驱动 bean 可以使用这个接口），除了使用 EJBContext 接口外，还可以使用 Resource 注释符注入或者使用 JNDI 命名 java:comp/UserTransaction 来获得。不能指定 Resource 注释符的 authenticationType 和 shareable 元素。

容器必须保证不允许使用这个接口的企业 bean 不能获得这个接口。如果不允许使用 UserTransaction 接口的企业 bean 容器企图在 JNDI 中使用 JNDI API 查找这个接口，那么容器应当抛出 javax.naming.NameNotFoundException。

下面的例子解释了企业 bean 如何通过注入获取和使用 UserTransaction 对象。

```
@Resource UserTransaction tx;
...
public void updateData(...) {
...
// Start a transaction.
tx.begin();
...
// Perform transactional operations on data.
...
// Commit the transaction.
tx.commit();
...
}
```

下面的代码

```
public MySessionBean implements SessionBean {
...
public someMethod()
{
...
Context initCtx = new InitialContext();
UserTransaction utx = (UserTransaction)initCtx.lookup(
```

```
"java:comp/UserTransaction");
utx.begin();
...
utx.commit();
}
...
}
```

功能上等价于

```
public MySessionBean implements SessionBean {
...
SessionContext ctx;
...
public someMethod()
{
UserTransaction utx = ctx.getUserTransaction();
utx.begin();
...
utx.commit();
}
...
}
```

UserTransaction 对象引用也可以使用和资环境引用同样的方式声明在部署描述中。这样的部署描述条目可以用于指定 UserTransaction 对象的注入。

16.12.1 Bean 提供者的责任

Bean 提供者负责使用 Resource 注释符请求注入 UserTransaction 对象，或者使用定义的名字查找 UserTransaction 对象。

16.12.2 容器提供者的责任

容器提供者负责按本规范的要求提供合适的 UserTransaction 对象

16.13 ORB 引用

需要使用 CORBA ORB 执行特定操作的企业 bean 可以通过请求注入 ORB 对象或查找 JNDI 的 java:comp/ORB 来找到合适的实现了 ORB 接口的对象。任何这样的对 ORB 对象引用只在执行查找的 bean 实例中有效。

下面的例子解释了应用组件如何通过注入获取和使用 ORB 对象。

@Resource ORB orb;

public void method(...) {

...

// Get the POA to use when creating object references.

POA rootPOA = (POA)orb.resolve_initial_references(“RootPOA”);

...

}

下面的代码解释了企业 bean 如何通过 JNDI 查找获取和使用 ORB 对象。

```
@Resource ORB orb;
public void method(...) {
...
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();
// Look up the ORB object.
ORB orb = (ORB)initCtx.lookup(“java:comp/ORB”);
// Get the POA to use when creating object references.
POA rootPOA = (POA)orb.resolve_initial_references(“RootPOA”);
...
}
```

ORB 引用也可以使用和资源管理器连接工厂引用同样的方式声明在部署描述中。这样的部署描述条目可以用于指定 ORB 对象的注入。

在 JNDI 名字是 java:comp/ORB 下的 ORB 实例总是可以共享的实例。缺省

情况下，注入到企业 bean 的 ORB 实例或通过部署描述声明的 ORB 实例也可以是共享实例。但是，应用可以将 Resource 注释符的 shareable 元素设置为 false，或者可以在部署描述中价格 res-sharing-scope 元素设置成 Unshareable 来请求非共享的 ORB 实例。

16.13.1 Bean 提供者的责任

Bean 提供者负责使用 Resource 注释符请求注入 ORB 对象，或者使用定义的名字查找 ORB 对象。如果 Resource 注释符的 shareable 元素被设置成 false，那么注入的 ORB 对象不会被应用中的其他对象共享，这个私有的 ORB 实例只用于给定的组件。

16.13.2 容器提供者的责任

容器提供者负责按本规范的要求提供合适的 ORB 对象。

16.14 TimerService 引用

容器必须保证除了通过 EJBContext 外，还可以通过使用 Resource 注释符注入或 JNDI 的 java:comp/TimerService 可以获得 TimerService 接口。不能指定 Resource 注释符的 authenticationType 和 shareable 元素。

TimerService 对象引用也可以用和资源环境引用同样的方式在部署描述中声明。这样的部署描述条目可以用于指定 TimerService 对象的注入。

16.14.1 Bean 提供者的责任

Bean 提供者负责使用 Resource 注释符请求注入 TimerService 对象，或者使用定义的名字查找 TimerService 对象。

16.14.2 容器提供者的责任

容器提供者负责按本规范的要求提供合适的 TimerService 对象。

16.15 EJBContext 引用

容器必须保证可以通过使用 Resource 注释符或 JNDI 的 java:comp/EJBContext 来得到组件的 EJBContext 接口。不能指定 Resource 注释符的 authenticationType 和 shareable 元素。

也可应使用和资源环境引用同样的方式在部署描述中声明 EJBContext 对象引用。这样的部署描述条目也可以用于指定 EJBContext 对象的注入。

16.15.1 Bean 提供者的责任

Bean 提供者负责使用 Resource 注释符请求注入 EJBContext 对象，或使用定义的名字来查找 EJBContext 对象。

通过命名环境获取的 EJBContext 对象只在执行查找的 bean 实例中有效。

16.15.2 容器提供者的责任

容器提供者负责将合适的 EJBContext 提供给引用组件。返回的对象必须是与 bean 请求注入或执行查找的指定类型一致——也就是说，容器提供者必须将 SessionContext 接口的实例返回给引用的会话 bean，将 MessageDrivenContext 接口的实例返回给消息驱动 bean。

16.16 废弃的 EJBContext.getEnvironment 方法

在 EJB1.1 中引入的“环境命名上下文”替代了 EJB1.0 中“环境属性”的概念。

EJB1.1 或后期的兼容容器不要求对 EJB1.0 风格的环境属性提供支持。如果容器没有实现这个功能，那么它应当从 EJBContext.getEnviroment 方法中抛出

RuntimeException（或它的子类）。

当工具将 EJB1.0 部署描述转换成 EJB1.1 的 XML 格式，那么它们应当将环境属性的定义替换成环境命名上下文的 ejb10-properties 子上下文。env-entry 元素应当按下述方式定义：env-entry-name 包含环境属性的名字，env-entry-type 必须是 java.lang.String，可选的 env-entry-value 包含环境属性的值。

例如，EJB1.0 企业 bean 有两个环境属性 foo 和 bar，在 EJB1.1 格式的部署描述中应按下面的 env-entry 元素来声明。

```
...
<env-entry>
env-entry-name>ejb10-properties/foo</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>
<env-entry>
<description>bar's description</description>
<env-entry-name>ejb10-properties/bar</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>bar value</env-entry-value>
</env-entry>
    ...
```

容器应当将声明在 ejb10-properties 子上下文中的条目作为 java.util.Properties 对象提供实例，实例通过调用 EJBContext.getEnviroment 方法获取它。

企业 bean 使用 EJB1.0 API 来访问属性，如下所示：

```
public class SomeBean implements SessionBean {
SessionContext ctx;
java.util.Properties env;
public void setSessionContext(SessionContext sc) {
ctx = sc;
env = ctx.getEnvironment();
}
public someBusinessMethod(...) ... {
String fooValue = env.getProperty("foo");
String barValue = env.getProperty("bar");
```

```
}
...
}
```

17 安全管理

本章定义了 EJB 架构支持的安全管理。

17.1 概述

我们为EJB 架构的安全管理设定了以下目标：

- *通过使用更多合格的EJB 角色来覆盖更多的安全范围来保护应用，从而减轻应用开发者的负担。EJB 容器提供者提供了安全基础设施的实现；部署这和系统管理员订阅安全策略。*
- *可以让应用组装者或部署员设置安全策略。*
- *可以让企业 bean 应用在使用不同安全策略的多个 EJB 服务器间进行移植。*

EJB 架构鼓励 Bean 提供者实现企业 bean 类而不需要在业务方法中硬编码安全策略和机制。在大多数情况下，企业 bean 的业务方法不应当保护任何安全相关的逻辑。这可以让部署员为应用配置最适合企业 bean 操作环境的安全策略。

为了使部署员的工作更容易，Bean 提供者或应用组装者（它和 Bean 提供者不是相同的组织）可以定义为由一到多个企业 bean 组成的应用定义安全角色。安全角色是权限的语义分组，它是成功使用应用必须给定的应用用户类型。Bean 提供者可以声明式使用元数据注释或部署描述为每个安全角色定义方法权限。应用组装者可以定义、增强或覆盖用部署描述指定的权限。方法权限是调用企业 bean 业务接口、home 接口、组件接口和/或 web 服务终端的特定的一组方法的权限。安全角色给部署者展现了企业 bean 应用的一个简单的安全视图——应用安全需求的部署者视图是一小部分安全角色，而不是大量的单个方法。

那些在它之下进行方法调用的安全主体通常是组件调用者的安全主体。但

是，通过指定一个运行标识，可以为 bean 业务接口、home 接口、组件接口和/或 web 服务中的方法以及其他 bean 调用的企业 bean 的任何方法的执行指定不同的安全主体。

这决定了调用者主题是否可以从调用者到被调用者进行传播——也就是说，被调用的企业 bean 是否看到和调用企业 bean 调用 EJBContext.getCallerPrinciipal 方法返回的值相同——或者已经赋给特定安全角色的安全主体是否将用于 bean 方法的执行，并且将视为在 bean 的被调用者中的调用者主体。

Bean 提供者可以使用元数据注释符或部署描述来指定调用者的安全标识或运行的安全标识应当用于 bean 方法的执行。

- 缺省情况下，调用者主体将作为调用者标识被传播。Bean 提供者可以使用 RunAs 注释符来指定一个安全主体来替代调用者的安全主体，这个安全主体已经被赋给了一个特定的安全角色。参见节 17.3.4。
- 如果部署描述用于指定安全主体，那么 Bean 提供者或应用组装者可以使用 security-identity 部署描述元素来指定安全标识。如果没有指定 security-identity 部署描述元素且也没有使用 RunAs 注释符指定运行标识符或如果 use-caller-identity 指定为 security-identity 元素的值，那么调用者的主体将从调用者传播到被调用者。如果指定了 run-as 元素，则使用赋给指定安全角色的安全主体。应用组装者可以覆盖由 Bean 提供者或缺省的安全标识符的值。

部署者负责将定义在目标操作环境中的主体或主体组分配到由 Bean 提供者或应用组装者定义的安全角色上。部署者也负责为运行标识分配主体。部署者也要负责配置企业 bean 安全管理的其他方面，例如内部企业 bean 调用的主体映射和对资源管理器访问的主体映射。

在运行时，如果和客户端调用关联的主体已经被部署者分配了至少一个可以调用业务方法的安全角色，或如果 Bean 提供者或应用组装者已经指定了对被调用的方法不进行安全授权校验（也就是说，所有的角色，包括未授权的角色都可以访问），那么客户端可以调用业务方法。参见节 17.3.2。

容器提供者负责在运行时履行安全策略，提供运行时管理安全的工具，提供

由部署者在部署时管理安全的工具。

因为并非所有的安全策略都能声明式的表达，因此 EJB 架构提供了简单的编程接口，Bean 提供者可以使用这些接口从业务方法中访问安全上下文。

下面的小节定义了与安全管理相关的各个 EJB 角色的责任。

17.2 Bean 提供者的责任

本节定义了支持安全的 EJB 架构中 Bean 提供者的视角，以及定义了它的责任。另外，Bean 提供者可以为应用定义安全角色，正如在节 17.3 中描述。

17.2.1 调用其他企业 bean

企业 bean 的业务方法可以通过其他 bean 的业务接口或 home 或组件接口来调用其他的企业 bean。EJB 架构没有为调用企业 bean 提供编程接口来控制传入到被调用企业 bean 的安全主体。

管理传入到内部企业 bean 调用的调用者主体（也就是，主体代理）由部署者和系统管理员以容器特有的方式来设置。Bean 提供者和应用组装者应当为作为描述的一部分的内部企业 bean 调用的调用者主体管理描述所有的需求。

17.2.2 访问资源

节 16.7 定义了访问资源管理器的协议，包括安全管理的需求。

17.2.3 访问后台 OS 资源

EJB 架构没有定义在该主体下执行企业 bean 方法的操作系统主体。因此，Bean 提供者不能依赖访问后台 OS 资源的特殊的主体，例如文件。（参见节 17.6.8 了解为什么不能）。

我们相信大多数企业业务应用将信息存储在如数据库的资源管理器，而不是存储在操作系统级的资源中。因此，这个规则不影响大多数企业 bean 的移植性。

17.2.4 编程风格推荐

Bean 提供者既不需要实现安全机制，也不需要在企业 bean 的业务方法中硬编码安全策略。反而，Bean 提供者应当依赖 EJB 容器提供的安全机制。

Bean 提供者可以使用元数据注释符和/或部署描述来向部署者提供安全相关的信息。这些信息有助于部署者为企业 bean 应用设置相应的安全策略。

17.2.5 编码访问调用者的安全上下文

注意：通常应当由容器以对企业 bean 业务方法透明的方式来执行安全管理。在本节中描述的安全 API 应当只用于很少的情况，在这些情况中企业 bean 的业务方法需要访问安全上下文信息。

javax.ejb.EJBContext 接口提供了两个方法（还有两个在 EJB1.0 中定义的已被废弃的方法），这两个方法可以让 Bean 提供者访问关于企业 bean 调用者的安全信息。

```
public interface javax.ejb.EJBContext {
...
//
// The following two methods allow the EJB class
// to access security information.
//
java.security.Principal getCallerPrincipal();
boolean isCallerInRole(String roleName);
//
// The following two EJB 1.0 methods are deprecated.
//
java.security.Identity getCallerIdentity();
boolean isCallerInRole(java.security.Identity role);
...
    }
```

Bean 提供者只可以在表 1、表 2、表 3、表 4 和表 10 中定义的 bean 业务方法中调用 getCallerPrincipal 和 isCallerInRole 方法。如果在没有安全上下文时调用这两个方法，则应当抛出 java.lang.IllegalStateException。

getCallerIdentity 和 isCallerInRole(Identity role)方法在 EJB1.1 中被废弃。Bean

提供者在新企业 bean 中必须使用 getCallerPrincipal 和 isCallerInRole(String roleName)。

EJB1.1 或后来的容器可以选择按以下方式实现这两个废弃的方法：

- 不想对这两个废弃方法提供支持的容器应当从 getCallerIdentity 中抛出 RuntimeException（或它的子类）。
- 想对 getCallerIdentity 方法提供支持的容器应当返还 java.security.Identity 抽象类子类的实例。在返回对象上调用 getName 方法的返回值必须和 getCallerPrincipal().getName()返回值一致。
- 不想对废弃方法 isCallerInRole(Indentity identity)提供支持的容器应当从 isCallerInRole(Indentity identity)中抛出 RuntimeException（或它的子类）。
- 想实现 isCallerInRole(Indentity identity)方法的容器应当按以下方式实现：

```
public isCallerInRole(Identity identity) {
return isCallerInRole(identity.getName());
}
```

17.2.5.1 getCallerPrincipal 的用法

getCallerPrincipal 方法用于企业 bean 方法获取当前调用者主体的名字。例如，这个方法可能使用这个名字作为数据库中信息的主键。

企业 bean 可以调用 getCallerPrincipal 方法来获取代表当前调用者的 java.security.Principal 接口。企业 bean 然后可以使用 java.security.Principal 接口的 getName 方法获取调用者主体的名字。如果没有设置安全标识，那么 getCallerPrincipal 方法返回未授权标识的容器的表示法。

注意：getCallerPrincipal 方法返回代表企业 bean 调用者的主体，不是对应予 bean 的运行时安全标识的主体，如果有的话。

“当前调用者”的含义、实现了 java.security.Principal 接口的 Java 类和由 getCallerPrincipal 方法返回的主体的领域都依赖于应用的操作环境和配置。

企业可以有一个复杂的安全基础设施，它包含多个安全域。安全基础设施可以在 EJB 调用者到 EJB 对象的路径上执行一到多个主体映射。例如，通过网络

访问他的公司的雇员可以验证主体，然后在将方法调用转发到 EJB 对象之前将主体映射到企业局域网的 Kerberos 主体上。如果安全基础设施执行主体映射，那么 getCallerPrincipal 方法返回映射的主体，而不是原始调用者主体。（在前面的例子中，getCallerPrincipal 将返回 Kerberos 主体）。安全基础设施的管理，例如主体映射，由系统管理员角色执行；这超出了本规范的范围。

下面的代码例子解释了 getCallerPrincipal()方法的用法。

```
@Stateless public class EmployeeServiceBean
implements EmployeeService{
@Resource SessionContext ctx;
@PersistenceContext EntityManager em;
public void changePhoneNumber(...) {
...
// obtain the caller principal.
callerPrincipal = ctx.getCallerPrincipal();
// obtain the caller principal's name.
callerKey = callerPrincipal.getName();
// use callerKey as primary key to find EmployeeRecord
EmployeeRecord myEmployeeRecord = em.find(EmployeeRecord.class,
callerKey);
// update phone number
myEmployeeRecord.setPhoneNumber(...);
...
}
}
```

在前面的例子中，企业 bean 获得当前调用者的主体名字，然后使用它作为定位 EmployeeRecord 实体的主键。这个例子假定应用设置的是当前调用者主体包含了用于雇员标识的主键（例如，雇员编号）。

17.2.5.2 isCallerInRole 的用法

isCallerInRole(String roleName)方法目的是为了让 Bean 提供者在代码中进行安全检查，这种检查不能很容易的使用方法授权声明在部署描述中。这样的检查可能在请求上强加了基于角色的约束，或者它可能依赖于数据库中的信息。

企业 bean 代码可以使用 isCallerInRole 方法来检查当前的调用者是否已经被

赋予了给定的安全角色。安全角色有 Bean 提供者或应用组装者（参见节 17.3.1）定义，并由部署者将这些角色指派到操作环境中存在的主体或主体组。

注意，isCallerInRole(String roleName)检查代表企业 bean 调用者的主体，而不是 bean 的运行时安全标识的主体，如果有的话。

下面的代码例子解释了 isCallerInRole(String roleName)方法的用法。

```
@Stateless public class PayrollBean implements Payroll {
@Resource SessionContext ctx;
public void updateEmployeeInfo(EmplInfo info) {
oldInfo = ... read from database;
// The salary field can be changed only by callers
// who have the security role "payroll"
if (info.salary != oldInfo.salary &&
!ctx.isCallerInRole("payroll")) {
throw new SecurityException(...);
}
...
}
...
}
```

17.2.5.3 从 Bean 代码中引用安全角色的声明

Bean 提供者负责使用 DeclareRoles 注释符或部署描述的 security-role-ref 元素来声明在企业 bean 代码中使用的所有安全角色。DeclareRoles 注释符指定在 bean 类上，它用于声明可以在被注释类方法中调用 isCallerInRole 来检查的角色。声明安全角色可以让 Bean 提供者、应用组装者或部署者将这些用在代码内的安全角色名字关联到为应用定义的安全角色。如果没有做关联，那么在代码中使用的所有的安全角色名字假定对应于应用中同名的安全角色。

Bean 提供者声明在代码中应 DeclareRoles 引用的安全角色。当声明的角色名用作 isCallerInRole(String roleName)方法的参数时，声明的名字必须和参数值一致。Bean 提供者可以可选的在 DeclareRoles 注释符的 description 元素中提供命名安全角色的描述。

在下面的例子中，DeclareRoles 注释符用于表示企业 bean AardvarkPayroll

在它的业务方法中使用 isCallerInRole(“payroll”)来进行安全检查。

```
@DeclareRoles(“payroll”)
@Stateless public class PayrollBean implements Payroll {
@Resource SessionContext ctx;
public void updateEmployeeInfo(EmplInfo info) {
oldInfo = ... read from database;
// The salary field can be changed only by callers
// who have the security role "payroll"
if (info.salary != oldInfo.salary &&
!ctx.isCallerInRole("payroll")) {
throw new SecurityException(...);
}
...
}
...
}
```

如果没有使用 DeclareRoles 注释符，那么 Bean 提供者必须使用部署描述的 security-role-ref 元素来声明在代码中引用的安全角色。按下面方式定义 security-role-ref 元素：

- 用 role-name 元素声明安全角色的名字。这个名字必须是用作 isCallerInRole(String roleName)方法参数的名字。
- 可选的在 description 元素中提供安全角色的描述。

下面的例子解释了如何在部署描述中声明企业 bean 对安全角色的引用。

```
...
<enterprise-beans>
...
<session>
<ejb-name>AardvarkPayroll</ejb-name>
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
...
<security-role-ref>
<description>
This security role should be assigned to the
employees of the payroll department who are
allowed to update employees’ salaries.
</description>
<role-name>payroll</role-name>
</security-role-ref>
```

```
...
</session>
...
</enterprise-beans>
    ...
```

上面的部署描述表示企业 bean AardvarkPayroll 在它的业务方法中使用 isCallerInRole(“payroll”)来进行安全检查。

一个安全角色引用，包括引用定义的名字，范围限定于 bean 类中包含了 DeclareRoles 或部署描述中包含了 security-role-ref 部署描述元素的 bean 类的组件。

Bean 提供者（或应用组装者）也可以为那些声明在注释符的引用使用 security-role-ref 元素，这样就是 bean 提供者希望链接到 scurity-role 的名字不同于引用的名字。如果安全角色引用没有用这种方式链接到一个安全角色，那么容器必须将引用的名字映射到相同名字的安全角色。参见节 17.3.3 了解安全角色引用如何被链接到安全角色。

17.3 Bean 提供者和/或应用组装者的责任

Bean 提供者和应用组装者（它可能和 Bean 提供者是同一个组织）可以为 ejb-jar 文件中的企业 bean 定义它的安全视图。对 Bean 提供者和应用组装者来说提供安全视图是可选的。

为企业 bean 提供安全视图的原因是为了简化部署者的工作。如果应用没有安全视图，那么部署者需要详细了解应用，这样才能安全的部署应用。例如，部署者必须知道每个业务方法以决定哪个用户可以调用它。由 Bean 提供者或应用组装者为部署者展现一个统一的视图可以让部署者不必对应用非常熟悉。

安全视图由一系列安全角色组成。一个安全角色是权限的语义组，应用的用户必须有这种授权才能成功使用应用。

Bean 提供者或应用组装者为每个安全角色定义“方法权限”。方法权限是一个可以访问企业 bean 业务方法接口、home 接口、组件接口和/或 web 服务终端中指定的方法组。

时刻注意安全角色是用于定义应用的逻辑安全视图是非常重要的。他们不应当与用户组、用户、主体和其它目标企业操作环境中的概念相混淆。

17.3.1 安全角色

Bean 提供者或应用组装者可以在 bean 的元数据注释符或部署描述中定义一个或多个安全角色。Bean 提供者或应用组装者然后将企业 bean 的业务、hom 和组件接口和/或 web 服务终端的方法分派到安全角色来定义应用的安全视图。

因为 Bean 提供者或应用组装者通常情况下不知道操作环境的安全环境，因此安全角色是指“逻辑”角色（或参与者），每个角色代表了一类用户，他们对应用有相同的访问权限。

部署者然后将操作环境中的用户组和/或用户账户分配到由 Bean 提供者或应用组装者定义的安全角色上。

在元数据注释符和/或部署描述中定义安全角色是可选的（注，如果 Bean 提供者或应用组装者没有定义安全角色，部署者则必须在部署时定义安全角色）。Bean 提供者或应用组装者没有定义安全角色意味着他们不会给部署员传递任何与安全部署相关的指南。

如果使用 Java 语言元数据注释符，那么 Bean 提供者使用 DeclareRoles 和 RolesAllowed 注释符来定义安全角色。由应用使用的安全角色集可认为是在 DeclareRoles 和 RolesAllowed 注释符中用名字定义的安全角色的合集。Bean 提供供者可以使用 security-role 部署描述元素的方式为应用增强在注释符中的定义的安全角色集。

如果使用部署描述，则 Bean 提供者和/或应用组装者按以下方式使用 security-role 部署描述元素：

- 使用 security-role 元素定义每个安全角色。
- 使用 role-name 定义安全角色的名字。
- 可选的，使用 description 元素提供安全角色的描述。

下面的例子解释了在部署描述中定义安全角色：

```
<assembly-descriptor>
```

```
<security-role>
<description>
This role includes the employees of the
enterprise who are allowed to access the
employee self-service application. This role
is allowed only to access his/her own
information.
</description>
<role-name>employee</role-name>
</security-role>
<security-role>
<description>
This role includes the employees of the human
resources department. The role is allowed to
view and update all employee records.
</description>
<role-name>hr-department</role-name>
</security-role>
<security-role>
<description>
This role includes the employees of the payroll
department. The role is allowed to view and
update the payroll entry for any employee.
</description>
<role-name>payroll-department</role-name>
</security-role>
<security-role>
<description>
This role should be assigned to the personnel
authorized to perform administrative functions
for the employee self-service application.
This role does not have direct access to
sensitive employee and payroll information.
</description>
<role-name>admin</role-name>
</security-role>
...
</assembly-descriptor>
```

17.3.2 方法权限

如果 Bean 提供者和/或应用组装者已经为位于 ejb-jar 文件中的企业 bean 定义了安全角色，那么他们也可以指定每个安全角色可以访问的业务、home 和组件接口和/或 web 服务终端的方法。

元数据注释符和/或部署描述也可以实现该功能。

方法权限被定义为从安全角色集到方法集的二元关系，这些方法包括会话 bean 和实体 bean 的业务接口、home 接口、组件接口和/或 web 服务终端的方法，包括所有父接口的方法（包括 EJBHome 和 EJBObject 接口和/或 EJBLocalHome 和 EJBLocalObject 接口）。方法权限关系包括（R，M）对，有且只有安全角色 R 才可以调用方法 M。

17.3.2.1 使用元数据注释符的方法权限规范

下面是使用 Java 语言元数据注释符的方法权限的规范。

Bean 类方法的方法权限都可以指定在类上，类的方法上，或两个地方都指定。

RolesAllowed、PermitAll 和 DenyAll 注释符用于指定方法权限。RolesAllowed 注释符的值是映射到安全角色的安全角色名字列表，这些安全角色是可以执行指定的方法的角色。PermitAll 注释符知道所有的安全角色都可以执行指定的方法。DenyAll 注释符指定所有的安全角色都不可以执行指定的方法。

在 bean 类上指定 RolesAllowed 或 PermitAll 注释符意思是它们应用到类的所有方法上。

方法权限可以指定在 bean 类的方法上，这样就可以覆盖 bean 类上指定的值。

如果 bean 类有超类，则应用下面的附加规则：

- 指定在超类 S 上的方法权限值应用到由 S 定义的业务方法上。
- 方法权限值可以指定在有类 S 定义的方法 M 上，以此来覆盖在类 S 上显式或隐式为方法 M 指定的权限值。
- 如果类 S 的方法 M 覆盖了由 S 超类定义的业务方法，那么由上面应用到

类 S 的规则来决定 M 的方法权限值。

例子：

```
@RolesAllowed(“admin”)
public class SomeClass {
public void aMethod () {...}
public void bMethod () {...}
...
}
@Stateless public class MyBean implements A extends SomeClass {
@RolesAllowed(“HR”)
public void aMethod () {...}
public void cMethod () {...}
...
}
```

假定 aMethod，bMethod，cMethod 是业务接口 A 的方法，aMethod 和 bMethod 的方法权限值分别是 RolesAllowed(“HR”)和 RolesAllowed(“Admin”)。方法 cMethod 的方法权限没有被指定（参见节 17.3.2.2 和 17.3.2.3）。

17.3.2.2 在部署描述中的方法权限规范

Bean 提供者使用部署描述作为元数据指定方法权限的替代方案（或者作为一种补充或覆盖元数据注释符的方法权限值的方法）。应用组装者可以使用 bean 的部署描述覆盖方法权限。

在部署描述中显式指定的任何方法权限值都覆盖在注释符中指定的值。如果在部署描述中没有为方法指定方法权限值，且已经通过注释符指定了方法权限值，那么使用注释符中的值。覆盖的粒度是方法级的。

Bean 提供者或应用组装者在部署描述中使用 method-permission 元素定义方法权限关系。

- 每个 method-permission 元素包括一个或多个安全角色的列表和一个或多个方法的列表。所有列出的安全角色都可以调用所有列出的方法。在列表中的每个安全角色都由 role-name 元素来标识，每个方法（或方法集，如下所述）都由 method 元素来标识。可以使用 description 元素为

method-permission 元素进行描述。

- 方法权限关系被定义为每个 method-permisssion 元素中定义的方法权限的并集。
- 一个安全角色或一个方法可以出现在多个 method-permission 元素中。

Bean 提供者或应用组装者可以指出所有角色都可以执行一个或多个指定的方法（也就是说，方法不应当在被容器调用之前进行授权检查）。元素 unchecked 用于在 method-permission 元素中替代角色名字来表示所有的角色都可以访问。

如果方法权限关系既指定了 unchecked 元素又指定了一个或多个安全角色，那么所有的角色都可以访问指定的方法。

exclude-list 元素用于表示不应当调用的方法集合。部署者应当配置企业 bean 的安全使得在 excluede-list 中指定的方法不能够被访问。

如果一个方法既指定在 exclude-list 元素中，又知道在方法权限关系中，那么部署者应当配置这个方法不能被访问。

元素 method 使用 ejb-name、method-name 和 method-params 元素来声明企业 bean 的业务、home 和组件接口和/或 web 服务终端的一个或多个方法。mehtod 元素有三种合法的风格：

风格 1：

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>*</method-name>
</method>
```

这个风格用于指定企业 bean 的业务、home 和组件接口以及 web 服务终端的所有方法。

风格 2：

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
</method>
```

这个风格用于指定企业 bean 的业务、home 和组件接口以及 web 服务终端的特定方法。如果有多个重载方法具有相同的名字，那么这个风格指的是所有的重载方法。

风格 3：

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAMETER_1</method-param>
...
<method-param>PARAMETER_N</method-param>
</method-params>
</method>
```

这个风格用于指定具有重载名称的方法集内的特定方法。这个方法必须定义在指定的企业 bean 的业务、home 和组件接口以及 web 服务终端中。但是，如果有多个方法具有相同的重载名称，那么这个风格指的是所有的重载方法。

可选的 method-intf 元素可以用于区分具有相同方法名和标识的方法，它在业务、home 和组件接口以及 web 服务终端多个地方被多次定义。如果一个方法既是本地业务接口的方法又是本地组件接口内的方法，那么对这个方法应用同一个方法权限值。否则，如果如果一个方法既是远程业务接口的方法又是远程组件接口的方法，那么对这个方法应用同一个方法权限值。

下面的例子解释了在部署描述中如何将安全角色分派到方法权限：

```
...
<method-permission>
<role-name>employee</role-name>
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>*</method-name>
```

```
</method>
</method-permission>
<method-permission>
<role-name>employee</role-name>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>findByPrimaryKey</method-name>
</method>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>getEmployeeInfo</method-name>
</method>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>updateEmployeeInfo</method-name>
</method>
</method-permission>
<method-permission>
<role-name>payroll-department</role-name>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>findByPrimaryKey</method-name>
</method>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>getEmployeeInfo</method-name>
</method>
<method>
```

```
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>updateEmployeeInfo</method-name>
</method>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>updateSalary</method-name>
</method>
</method-permission>
<method-permission>
<role-name>admin</role-name>
<method>
<ejb-name>EmployeeServiceAdmin</ejb-name>
<method-name>*</method-name>
</method>
</method-permission>
...
```

17.3.2.3 未规定的方法权限

某些方法没有被分配任何安全角色也没有注释为 DenyAll，或也没有被包含在 exclue-list 元素中是可能的。在这种情况下，部署者应当为所有未指定的方法分配方法权限，或通过将它们分配到安全角色，或让他们是不用检查的。如果部署者没有为未指定方法分配方法权限，那么这些方法必须被容器看作是 unchecked 的。

17.3.3 将安全角色引用链接到安全角色

用于应用的所有组件的安全角色引用被链接到应用定义的安全角色。在没有定义任何显式链接的情况下，一个安全角色引用将被链接到有相同名字的安全角

色。

应用组装者可以显式的将所有在 DeclareRoles 注释符或 securtiy-role-ref 元素中声明的安全角色引用链接到通过注释符（参见节 17.3.1）和/或 security-role 元素定义的安全角色上。

应用组装者使用 role-link 元素将每个安全角色引用链接到安全角色。元素 role-link 的值必须是在 security-role 元素或通过 DeclareRoles 注释符或 RolesAllowed 注释符定义的安全角色（正如在节 17.3.1 中所述），但当 role-name 和 security-role（将被链接的）的名字相同时，则不必指定。

下面的部署描述例子展示了如果将名字为 payroll 的安全角色引用链接到名字为 payroll-department 的安全角色。

```
...
<enterprise-beans>
...
<session>
<ejb-name>AardvarkPayroll</ejb-name>
<ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
...
<security-role-ref>
<description>
This role should be assigned to the
employees of the payroll department.
Members of this role have access to
anyone’s payroll record.
The role has been linked to the
payroll-department role.
</description>
<role-name>payroll</role-name>
<role-link>payroll-department</role-link>
```

```
</security-role-ref>
...
</session>
...
</enterprise-beans>
...
```

17.3.4 在部署描述中的安全标识规范

Bean 提供者或应用组装者通常指定调用者的安全标识是否应当用于企业 bean 方法的执行，或是否应用一个指定的运行时标识。

缺省情况下，使用调用者的安全标识。Bean 提供者可以使用 RunAs 元数据注释符来为方法执行指定运行时标识。如果使用部署描述，那么 Bean 提供者或应用组装者可以使用 security-identity 部署描述元素来指定或覆盖元数据指定的安全标识。元素 security-identity 元素的值是 use-caller-indentity 或 run-as。

对应用组装者来说，在部署描述中定义安全标识是可选的。在部署描述中不指定意味着应用组装者没有为部署者传递任何安全标识相关的指南。

如果部署者没有指定 ran-as 标识符，那么容器应当为 bean 方法的执行使用调用者的安全标识。

17.3.4.1 Run-as

Bean 提供者可以使用 RunAs 元数据注释符，或 Bean 提供者或应用组装者可以使用 run-as 部署描述元素来为企业 bean 在部署描述中定义运行时安全标识。运行时安全标识作为整体应用到企业 bean 上，也就是说应用到企业 bean 的业务、home 和组件接口和/或 web 服务终端中的所有方法；应用到消息驱动 bean 的消息监听器方法；和企业 bean 的超时回调方法；以及所有可能被依次调用的 bean 内部方法。

为企业 bean 设置 run-as 标识不影响调用者的标识，调用者的标识是用于检测调用者是否有权限访问企业 bean 的方法。run-as 标识是在调用者调用时企业

bean 使用的标识。

因为 bean 提供者和应用组装者通常不知道操作环境的安全环境，因此 run-as 标识被指派为“逻辑”角色名，它对应于 Bean 提供者或应用组装者在元数据注释符或部署描述中定义的某个安全角色。

部署者然后将操作环境中定义的安全主体分配到用作 run-as 标识的主体。由部署者指派的安全主体应当是已经被指派到由 RunAs 注释符或由 ruan-as 部署描述元素的 role-name 元素指定的安全角色的主体之一。

Bean 提供者和/或应用组装者需要遵循下面的 run-as 标识规范：

- 使用 RunAs 元数据注释符或 run-as 部署描述元素的 role-name 元素定义安全角色的名称。
- 可选地，使用 description 元素来描述希望绑定到 run-as 标识的主体。

下面的例子解释了使用元数据注释符定义 run-as 标识。

```
@RunAs(“admin”)
@Stateless public class EmployeeServiceBean
implements EmployeeService{
...
}
```

Using the deployment descriptor, this can be specified as follows.

```
...
<enterprise-beans>
...
<session>
<ejb-name>EmployeeService</ejb-name>
...
<security-identity>
<run-as>
<role-name>admin</role-name>
</run-as>
```

```
</security-identity>
...
</session>
...
</enterprise-beans>
...
```

17.4 部署者的责任

部署者负责保证部署到目标操作环境中的应用是安全的。本节定义部署者在 EJB 安全管理方面的责任。

部署者使用 EJB 容器提供者提供的部署工具来读取由 Bean 提供者和/或应用组装者在注释符和/或部署描述中提供的应用的安全视图。部署者的工作就是将这个安全视图映射到目标操作环境中安全域使用的机制或策略上。部署者工作的输出包括操作环境特有的应用安全策略描述。描述的格式和保存在描述中的信息是 EJB 容器特有的。

下面的小节描述了由部署者执行的与安全相关的任务。

17.4.1 指派安全域和主体领域

部署者负责将安全域和主体领域分配到企业 bean 应用。

在同一个安全域中可以有多个主体领域，例如，为了分开雇员、商业伙伴和客户的主体领域。可以有多个安全域，例如，在应用托管场景中。

17.4.2 分派安全角色

部署者将用于操作环境中管理安全的主体和/主体组（例如个体用户或用户组）分配到由 DeclareRoles 和 RolesAllowed 注释符和/或 security-role 部署描述元素定义的安全角色。

部署者不会将主体和/或主体组分配到安全角色引用——分配到一个安全角

色的所有主体或/和主体组也会关联到安全角色引用上。例如，在节 17.3.3 中的 AardvarkPayroll 企业 bean 的部署者将包主体和/主体组分配到安全角色 payroll-department，且这些被分配的主体和/主体组将被隐式的分配到关联的安全角色 payroll。

EJB 架构没有规定企业 bean 如何实现安全架构。因此，将定义在应用部署描述中的逻辑安全角色分配到操作环境的安全概念的过程是操作环境特有的。通常，部署过程就是将每个安全角色分配到操作环境中定义的一个或多个用户组（或个体用户）的过程。这种分配是基于每个应用的。（也就是说，如果多个独立的 ejb-jar 文件使用同一个安全角色名称，每个都有不同的分配）。如果部署者没有将应用定义的逻辑安全角色分配到操作环境中的组，那么它必须将逻辑角色映射到相同名字的主体或主体组。

17.4.3 主体代理

部署者负责为内部组件调用配置主体代理。部署者必须遵循有 Bean 提供者和/或应用组装者提供的所有指南（例如，在 RunAs 注释符中提供的，部署描述中的 run-as 元素，注释符或部署描述中的 description 元素，或在部署手册中提供的指南）。

如果 Bean 提供者或应用组装者规定 run-as 标识符用于代表一个特定的企业 bean，那么部署者必须配置这个企业 bean，使得 run-as 主体被用作这个 bean 调用其他 bean 时的调用者主体，并且 run-as 主体随着对其他 bean 调用的调用链进行传播（在没有进一步指定 run-as 元素的情况下）。

17.4.4 资源访问的安全管理

部署者对资源访问的安全管理在节 16.7.2 中定义。

17.4.5 部署描述处理中的常规注意事项

部署者可以将 Bean 提供者和应用组装者在部署描述中定义的安全视图仅仅

看作是“暗示”，无论什么时候他都可以改变这些信息以适配操作环境的安全策略。

由于对 Bean 提供者和应用组装者来说，提供安全信息是可选的，因此部署者负责执行 Bean 提供者和应用组装者没有做的任务。（例如，如果没有在注释符或部署描述中定义安全角色和方法权限，那么部署者必须定义应用的安全角色和方法权限）。不要求部署者以标准 ejb-jar 文件的形式保存这个活动的输出。

17.5 EJB 客户端的责任

本节定义 EJB 客户端编程时必须遵循的规则，以保证安全上下文能传递到客户端调用，且可以被企业 bean 引入而不会与 EJB 服务器为协调安全上下文和事务产生冲突。

这些规则是：

- 事务性客户端不能在事务内改变它的主体关联。这个规则保证所有来自客户端的在一个职务内的调用都在同一个安全上下文中被执行。
- 会话 bean 的客户端不能改变用于和会话对象通信期间的主体关联。这个规则保证服务器可以在创建时将一个安全标识与会话实例建立关联，且在会话 bean 的生命周期内不需要改变安全关联。
- 如果来自多个客户端的事务性请求在单个事务中（如果在事务调用链中存在中间对象或中间程序，则可能发生这种情况），则所有的请求必须关联同一个安全上下文。

17.6 EJB 容器提供者的责任

本节描述 EJB 容器和服务器提供者的责任。

17.6.1 部署工具

EJB 容器提供者负责提供部署工具，使得部署者可以执行在节 17.4 中定义的任务。

部署工具从 bean 的元数据注释符和/或部署描述中读取信息，并将这个信息展示给部署者。工具指导部署者实现整个部署过程，并且向部署者展现选择项，这些选择用于将元数据注释符和部署描述中的安全信息映射到目标操作环境中的安全管理机制和策略。

部署工具的输出以 EJB 容器特有的方式被保存，在运行时可以由 EJB 容器获得。

17.6.2 安全域

EJB 容器为企业 bean 提供了一个安全域和一个或多个主体领域。EJB 架构没有规定 EJB 服务器如何实现安全域，且没有定义安全域的范围。

安全域可以被 EJB 服务器实现、管理和直接管理。例如，EJB 服务器可以保存 X509 证书或可以使用外部完全提供者，如 Kerberos。

EJB 规范没有定义安全域的访问。例如，范围可以是应用边界、EJB 服务器、操作系统、网络或企业。

EJB 服务器可以但不要求支持多安全域和/或多主体领域。

当使用大服务器进行应用托管时，可能发生在同一个 EJB 服务器上定义多个域。每个受托管的应用能够有它自己的安全域来保证安全和管理属于多个组织的应用间的隔离。

17.6.3 安全机制

EJB 容器提供者必须提供部署者设置安全策略所必需的安全机制。EJB 规范没有规定 EJB 服务器必须实现和支持的具体机制。

EJB 服务器通常提供的安全功能包括：

- *主体的认证。*
- *对 EJB 调用和资源管理器访问的访问认证。*
- *保护与远程客户端的通信（私密性，完整性等等）。*

17.6.4 在 EJB 调用上传递主体

EJB 容器提供者负责提供部署工具，以便部署者可以为一个企业 bean 到另一个企业 bean 的调用配置主体代理。EJB 容器负责按照部署者的规定执行主体代理。

EJB 容器必须能够让部署者为所有来自单个 Java EE 产品内的单个应用的调用指定主体代理， 调用者主体从一个企业 bean 到另一个企业 bean 进行传播（也就是，在调用中的多个 bean 从 getCallerPrincipal 返回的值是同一个）。

这个要求对需要在企业 bean 的调用链中返回同一个 getCallerPrincipal 值得应用是必需的。

EJB 容器必须能让部署者指定用于以下方法执行的 run-as 主体：会话或实体 bean 的业务、home 和组件接口和/或 web 服务终端方法，消息驱动 bean 的消息简体器方法。

17.6.5 在 javax.ejb.EJBContext 中的安全方法

EJB 容器必须能够让企业 bean 实例通过 getCallerPrincipal() 和 isCallerPrincipal(String roleName)方法访问调用者的安全上下文信息。在企业 bean 的业务、home、组件或消息监听器接口、web 服务终端和/或 TimedObject 接口的业务方法的执行期间，EJB 容器必须提供调用者的安全上下文信息，这些方法是指在节 4.4.1 的表 1、节 4.5.2 的表 2、节 5.5.1 的表 3、节 8.5.6 的表 4 和节 10.1.7 的表 10 中。容器必须保证所有通过这些接口进行的企业 bean 方法调用都与某个主体关联。如果没有设置调用者的安全标识，容器返回容器提供的未授权标识。容器不能让 getCallerPrincipal 方法返回 null。

17.6.6 对资源管理器的安全访问

EJB 容器提供者负责提供对资源管理器的安全访问，正如在节 16.7.3 中所述。

17.6.7 主体映射

如果应用要求它的客户端被部署到不同的安全域中，或如果跨多个安全域的多个应用需要交互，那么 EJB 容器提供者需要提供主体映射的机制和工具。系统管理员使用这些工具来配置应用环境的安全。

17.6.8 系统主体

EJB 规范没有定义“系统”主体，在这个主体下 JVM 运行企业 bean 的方法调用。

没有定义这个主体是为了让 EJB 容器提供者能更容易的在已有服务器基础设施的高层为EJB 提供运行时支持。例如，当某个EJB 容器实现可以在单个 JVM 中执行所有企业 bean 的所有实例时，另一容器实现可以让 ejb-jar 和客户端各使用一个 JVM。某些 EJB 容器可以让系统主体和应用级主体相同。其他的可以使用不同的主体，潜在的可能是不同的主体领域，甚至是不同的安全域。

17.6.9 运行时执行安全

EJB 容器负责执行由部署者定义的安全策略。执行机制的实现是 EJB 容器特有的。EJB 容器可以，但不是必须使用 Java 编程语言安全作为执行机制。

例如，为了隔离多个企业 bean 实例的执行，EJB 容器可以将多个实例加载到同一个 JVM，使用多个类加载器来隔离它们，或者它可以将每个实例加载到它自己的 JVM，然后使用操作系统提供的地址空间保护来隔离它们。

下面是 EJB 容器安全执行的一般需求：

- EJB 容器必须执行由部署者定义的客户端访问控制执行的安全策略。调用者只能调用指定为 PermitAll 的方法，或调用者被分配的安全角色中至少有一个的方法权限定义包含了被调用方法的方法。（也就是说，调用者不必被分配所有的角色）。如果容器拒绝客户端对业务方法的访问，那么容器应当抛出 javax.ejb.EJBAccessException（注，如果业务接口是继承了 java.rmi.Remote 的远程业务接口，则抛出

java.rmi.AccessException）。如果使用 EJB2.1 客户端视图，那么如果客户端是远程客户端，则容器必须抛出 java.rmi.RemoteException（或它的子类 java.rmi.AccessException），如果客户端是本地客户端，则抛出 javax.ejb.EJBException（或它的子类 javax.ejb.AccessLocalException）。

- EJB 容器必须将企业 bean 实例与其他企业 bean 实例和其它运行在服务器上的应用组件隔离开。EJB 容器必须保证其他企业 bean 实例和其它应用组件都只能通过企业 bean 的业务接口、组件接口、home 接口和/或 web 服务终端访问企业 bean。
- EJB 容器必须在运行时隔离企业 bean 实例，以便该实例不会访问未经授权的系统信息。这些信息包括容器的内部实现类，各种运行时状态和由容器维护的上下文，对其他企业 bean 实例的对象引用或由其他企业 bean 实例使用的资源管理器。EJB 容器必须保证企业 bean 与容器间的交互只能通过 EJB 架构接口进行。
- EJB 容器必须保证企业 bean 持久化状态的安全。
- EJB 容器必须根据部署者定义的安全策略来管理调用其他企业 bean 或访问资源管理器的主体映射。
- 容器必须可以让同一个企业 bean 能被独立的多次部署，每次都有一个不同的安全策略（例如，每次使用不同的 bean 名称来安装企业 bean（按照部署描述中的定义））。容器必须能让多次部署的企业 bean 在运行时可以共存。

17.6.10 审计跟踪

EJB 容器可以提供安全审计跟踪机制。安全审计跟踪机制通常记录所有的 java.security.Exception。它也记录所有的 EJB 服务器、EJB 容器、EJB 组件接口、EJB 的 home 接口和 EJB web 服务终端拒绝。

17.7 系统管理员的责任

本节定义系统管理员在安全方面的责任。注意，某些责任可以由部署者执行，或者可以由系统管理员和部署者共同完成。

17.7.1 安全域管理

系统管理员负责主体的管理。安全域管理超出了 EJB 规范的范围。

通常，系统管理员负责创建新用户帐号，向用户组增加用户，从用户组中删除用户以及删除或冻结一个用户帐号。

17.7.2 主体映射

如果客户端与目标企业 bean 位于不同的安全域，那么系统管理员负责将客户端使用的主体映射到企业 bean 的主体。部署者可以获得这个映射。

主体映射技术的规范超出了 EJB 架构的范围。

17.7.3 审计跟踪检查

如果 EJB 容器提供了审计跟踪功能，则系统管理员负责对审计跟踪进行管理。

18 Timer 服务

本章描述容器管理的 Timer 服务。这个服务可以让 bean 提供者注册在特定时间点、在特定时间之后和在一段之后回调的 EJB。

18.1 概述

基于工作流的企业应用为了管理语义状态转换，这些转换通常由偶然发生的临时事件触发。

EJB Timer 服务是容器管理的服务，它提供允许基于时间事件的调度回调方法。容器为这些时间事件提供可靠和事务性的服务。可以在特定的时间、特定的时间段、或者循环时间片来触发计时器调度通知。

Timer 服务有 EJB 容器实现。企业 bean 通过 EJBContext 或者通过查找 JNDI 命名空间来进行依赖注入。

EJB Timer 服务粗粒度的时间通知服务，它的目的是用于应用级处理模型，不是用于实时事件模型。

然而，用毫秒单位来表示时间，这是因为毫秒是 Java SE 平台 API 的最小单位。所以时间事件最好是对应于小时、天、或较长期间的时间事件。

下面的章节描述 Timer 服务充当不同 EJB 角色。

18.2 Bean 提供者眼中的 Timer 服务

容器提供的 EJB Timer 服务允许企业 bean 注册在特定的时间、时间段或时间段之后发生的时间回调方法。计时器服务提供了创建和取消计时器的方法，也提供了定位和 bean 关联的计时器的方法。

计时器用于调度基于时间的回调。使用计时器服务的企业 bean 必须提供超时回调方法。这个方法可以是用 Timeout 注释符注释的方法，也可以是实现了 javax.ejb.TimedObject 接口的 bean。Javax.ejb.TimedObject 接口只有一个计时器回调方法 ejbTimeout。可以为无状态会话 bean、消息驱动 bean 和 2.1 实体 bean 创建计时器。不能为有状态会话 bean 创建（注：可能在将来的规范中增加这项功能）或者 EJB3.0 实体。

为 2.1 实体 bean 创建的计时器和实体 bean 的唯一标识建立关联。可以调用处于池化状态的任何 bean 实例的为无状态会话 bean 活消息驱动 bean 创建的计时器的超时回调方法。

当在创建计时器时指定的时间到达时，容器调用 bean 的超时回调方法。bean 可以在到期之前取消计时器。如果计时器被取消，则不会调用回调方法（注：在竞态条件下的事件中，可能发生对超时回调方法的无关调用）。通过计时器的 cancel 方法来取消计时器。

通常在一个事务内调用计时器的创建和取消方法，以及超时回调方法。

计时器服务目的用于长业务流程模型。企业 bean 上的计时器在容器停止、服务器停止和企业 bean 激活/钝化以及加载/存储的声明周期过程中始终存在。

18.2.1 Timer 服务接口

可以通过 EJBContext 接口的 getTimerService 方法或者通过 lookup JNDI 命名空间来注入 Timer 服务。TimerService 接口有下列的方法：

```
public interface javax.ejb.TimerService {
    public Timer createTimer(long duration,java.io.Serializable info);
    public Timer createTimer(long initialDuration,long intervalDuration,
java.io.Serializable info);
    public Timer createTimer(java.util.Date expiration,java.io.Serializable info);
    public Timer createTimer(java.util.Date initialExpiration,long intervalDuration,
java.io.Serializable info);
    public Collection getTimers();
}
```

计时器的创建方法允许计时器作为一个单独时间的计时器或者作为一个时间段计时器。计时器终止时间（在时间段计时器情况下，需要初始话终止时间）可以是持续时间（时间段）也可以是绝对时间。

Bean 可以在创建计时器时传入特定的客户信息来识别计时器到期的含义。这些信息由计时器服务存储并可以通过计时器获得。信息对象必须是可序列化的（目前，除了通过 createTimer 方法外没有其他方式可以来创建信息对象。在将来的版本中可能会增加一个 API 来创建信息对象）。

计时器持续时间以毫秒的方式来表达。计时器服务从计时器被创建时来计算计时器的持续时间。

createTimer 方法返回一个 Timer 对象，这个对象可以允许企业 bean 取消计时器或者在计时器取消或/和到期之前获得计时器的信息（如果是一个单事件的计时器）。

getTimers 方法返回和 bean 关联活动的所有计时器。对于 EJB2.1 实体 bean，getTimers 的结果是和实体 bean 唯一标识关联的所有计时器。

18.2.2 超时回调

注册到计时器服务的企业 bean 的 class 必须提供超时回调方法。

这个方法可以是由 Timeout 注释符注释的方法（或者在部署时指定一个方法作为一个超时方法）或者这个 bean 实现 javax.ejb.TimedObject 接口。这个接口只有一个方法就是 ejbTimeout。如果 bean 实现了 TimedObject 接口，那么 Timeout 注释符或 timeout-method 只能用于指定 ejbTimeout 方法。一个 bean 最多只能有一个超时方法（注：这个方可以指定在 bean 的 class 上或者在它的超类上。如果使用 Timeout 注释符或 bean 实现 TimedObject 接口，那么如果指定 timeout-method 元素，则这个元素必须指向同一个方法）。

```
public interface javax.ejb.TimedObject {
    public void ejbTimeout(Timer timer);
}
```

任何注释为 Timeout 的方法（或者在配置描述符中指派）都必须有是下面的形式，其中<METHOD>是方法的名字（注：如果 bean 实现了 TimedObject 接口，在 ejbTimeout 方法上可以也可以不使用 Timeout 注释符）。Timeout 方法可以是 public，private，protected，或者 package 级别。Timeout 方法不能是 final 或 static 的。

```
void <METHOD>(Timer timer)
```

超时回调方法不能抛出应用异常。

当计时器到期时（例如，在创建时指定的毫秒数到期或超过指定的绝对时间），容器调用 bean 的为计时器注册的超时回调方法。超时方法包含 bean 提供者用于处理超时事件的业务逻辑。当计时器到期时，容器调用这个超时方法。当计时器被创建时，Bean 提供者可以使用 getInfo 方法来获取计时器提供的信息。这些信息使得受时间管理的对象能够识别计时器到期的含义。

容器交叉调用业务方法、生命周期回调方法和超时回调方法。因此，超时回

调方法被调用的时间点可以和创建计时器时指定的时间不完全一致。如果一个 bean 上有多个计时器并且他们几乎在同一时间到期，那么 bean 提供者必须处理不在序列中的超时回调方法。当调用取消方法时，Bean 提供者必须处理计时器到期事件中对超时回调方法的额外调用。

一般情况下，超时回调方法能够与组件接口的业务方法或者消息监听器接口的方法一样执行相同的操作。表格 2，3，4 和 10 中描述了可以由超时回调方法执行的操作。

由于超时回调方法是 bean 类的内部方法，因此它没有客户安全上下文。当在超时回调方法内部调用 getCallerPrincipal 时，它返回未授权标识的容器代表。

如果受时间管理的对象需要使用计时器的标识来识别计时器到期的含义，那么它可以使用 equals 方法来与其他的到期计时器引用进行比较。

如果计时器是一个单行为的计时器，那么容器在超时回调方法成功调用后删除该计时器（例如，当为超时回调方法启动的事务提交时）。如果在超时回调方法终止后调用计时器上的方法，则抛出 NoSuchObjectLocalException。

18.2.3 Timer 和 TimerHandle 接口

Javax.ejb.Timer 接口允许 bean 提供者取消计时器和获取计时器的信息。

Javax.ejb.TimerHandle 接口允许 bean 提供者获取可序列化的计时器句柄。由与计时器是本地对象，因此 TimerHandle 不可以传入 bean 的远程业务接口，远程接口或 web service 接口。

这两个接口的方法如下：

```
public interface javax.ejb.Timer {
    public void cancel();
    public long getTimeRemaining();
    public java.util.Date getNextTimeout();
    public javax.ejb.TimerHandle getHandle();
    public java.io.Serializable getInfo();
}
```

```
public interface javax.ejb.TimerHandle extends java.io.Serializable {
    public javax.ejb.Timer getTimer();
}
```

18.2.4 Timer 标识

Bean 提供者不能使用“==”来比较计时器。Bean 提供者必须使用 Timer.equals(Object obj)方法。

18.2.5 事务

通常，企业 bean 在事务范围内创建计时器。如果随后事务被回滚，那么创建计时器也回滚。

同样，企业 bean 通常在事务范围内取消计时器。如果事务被回滚，则取消操作被废止。

通常，超时回调方法的事务属性事 REQUIRED 或 REQUIRES_NEW（如果使用配置描述符就是 Required 或 RequiresNew）。如果事务被回滚，容器会重试超时。

注意：如果使用 REQUIRED（Required）事务属性，容器必须启动新事务。允许有事务属性值是为了超时回调方法的事务属性规范和其他的一样。

18.3 Bean 提供者的责任

本节描述 bean 提供者的责任。

18.3.1 企业 bean 类

注册到 Timer 服务的企业 bean 必须有一个超时回调方法。企业 bean 的 class 可以有超类和/或超类接口。如果 bean 的 class 有超类，则超时回调方法可以定义在这个 bean 的 class 上，也可以定义在它的任何超类上。

18.3.2 TimerHandle

由于 TimerHandle 接口扩展了 java.io.Serialiazable，因此客户可以序列化这个句柄。序列化后的句柄可以用于以后获取计时器标识的引用。TimerHandle 用于存储计时器。

TimerHandle 不可以作为企业 bean 的远程业务接口、远程接口或 web service 方法的参数或返回值。

18.4 容器的职责

本节描述支持 EJB 计时器服务的容器提供者的职责。

18.4.1 TimerService,Timer 和 TimerHandle 接口

容器必须提供 TimerService，Timer 和 TimerHandle 接口的实现。

Timer 实例不可以是可序列化的。

容器必须实现计时器句柄用于超越计时器的生命期。

容器必须提供相应的 Timer.equals(Object obj)和 hashCode()的实现。

18.4.2 Timer 到期和超时回调方法

在时间到期时或绝对时间点之后容器必须调用超时回调方法。计时器服务必须在计时器创建时开始计时。容器必须调用计时器到期的超时回调方法。

如果使用容器管理的事务划分并且事务属性指定或缺省为 REQUIRED 或 REQUIRED_NEW（如果使用配置描述符则为 Required 或 RequiresNew），容器必须在调用超时回调方法之前启动一个新的事务。如果事务失败或被回滚，那么容器必须至少重试回调方法一次。

如果 EJB2.1 实体 bean 的计时器到期并且这个 bean 已经被钝化，那么容器必须在调用超时回调方法之前调用 ejbActivate 和 ejbLoad 方法，参见 8.5.3 和 10.1.4.2 章节。

如果计时器是一个单事件计时器，容器必须清除计时器。如果在超时回调方法完成之后调用计时器上的方法，则容器必须抛出 javax.ejb.NoSuchObjectLocalException。

如果 bean 提供者在超时回调方法内调用 setRollbackOnly，那么容器必须在超时回调方法内回滚事务。这可以削除计时器到期的影响。在事务回滚后容器必须重试超时回调方法。

计时器是一个持久化对象。在容器崩溃时，所有在容器重启之前的时间内到期的单事件计时器必须在重启时调用超时回调方法。所有在重启之前到期的间隔计时器在重启时至少调用一次。

18.4.3 计时器的取消

当调用计时器的 cancel 方法时，容器必须清除该计时器。如果随后调用该计时器上的方法，容器必须抛出 javax.ejb.NoSuchObjectLocalException。

如果在取消计时器时发生的事务回滚，如果计时器还没有被取消，那么容器必须恢复计时器的持续时间。如果计时器在事务失败的时间点到期，那么在事务回滚后通知到期的计时器。

18.4.4 实体 bean 的删除

如果实体 bean 被删除，那么容器必须清除这个 bean 上的所有计时器。

19 部署描述

本章定义了作为 ejb-jar 文件一部分的部署描述。节 19.1 对部署描述进行了概述。节 19.2 到 19.4 从提供信息的各个 EJB 角色的角度描述了部署描述中的信息。节 19.5 定义了部署描述的 XML Schema 元素，这些元素是 EJB 架构专有的。在【12】中提供了 Java EE 平台规范公共的 XML Schema 元素。

19.1 概述

部署描述是 ejb-jar 文件生产者和消费者间协议的一部分。这个协议包含了两个部分：企业 bean 从 Bean 提供者到应用组装者的传递，和从应用组装者到部署者的传递。

Bean 提供者生产的 ejb-jar 文件包含一个或多个企业 bean，通常不包含应用组装指南。应用组装者生产的 ejb-jar 文件包含一个或多个企业 bean，以及描述企业 bean 如何被组合进单个应用部署单元的应用组装信息。

Java EE 规范定义了包含在多个 ejb-jar 文件中的企业 bean 和其它应用组件如何被组装进应用。

部署描述的角色是捕捉没有直接包含在企业 bean 代码中且计划用于 ejb-jar 文件消费者的声明信息。

在部署描述中有两种基本的信息类型：

- 企业 bean 的结构信息：结构信息描述了企业 bean 的结构和声明了企业 bean 对外部的依赖。为 ejb-jar 生产者提供结构信息。结构信息可以使用 bean 的代码中或部署描述提供结构信息。通常，不能改变结构信息，因为这样可能打破企业 bean 的功能。
- 应用组装信息：应用组装信息描述 ejb-jar 文件中的企业 bean（或 beans）如何被组装进更大的应用部署单元。提供的组装信息——无论在元数据注释符中还是在部署描述中——对 ejb-jar 文件生产者来说都是可选的。组装级别信息可以在不打破企业 bean 的功能情况下被改变，尽管这样做会改变已组装应用的行为。

19.2 Bean 提供者的责任

如果在元数据注释符中没有提供结构信息或使用的是缺省的结构信息，那么 Bean 提供者负责为每个企业 bean 在部署描述中提供下面的结构信息。

Bean 提供者使用 enterprise-beans 元素来列出 ejb-jar 文件的所有企业 bean。

Bean 提供者必须为每个企业 bean 提供下面的信息：

- 企业 bean 的名字：为每个企业 bean 分配逻辑名称。在这个名字和部署者为这个 bean 指派的 JNDI 名称直接没有关系。Bean 提供者可以在 ejb-name 元素中指定企业 bean 的名字。如果没有显式的在元数据注释符或部署描述中指定企业 bean 的名称，则缺省使用 bean 类的简称（即不带包名的类名）。
- 企业 bean 的类：如果 bean 类没有注释符为 Stateless、Stateful 或 Message-driven，那么 Bean 提供者必须使用 session 或 message-driven 部署描述元素的 ejb-class 元素来指定实现了企业 bean 业务方法的类的全称。Bean 提供者在 ejb-class 元素中指定企业 bean 类的名字。对于 EJB2.1 和早期的企业 bean，Bean 提供者必须使用这个元素来指定。
- 企业 bean 的本地业务接口：如果 bean 类有一个本地业务接口且既没有实现业务接口又没有使用元数据注释符将它指定为本地业务接口，那么 Bean 提供者必须在 business-local 元素中指定企业 bean 本地业务接口的全称。
- 企业 bean 的远程业务接口：如果 bean 类有一个远程业务接口且既没有实现业务接口又没有使用元数据注释符将它指定为远程业务接口，那么 Bean 提供者必须在 business-remote 元素中指定企业 bean 远程业务接口的全称。
- 企业 bean 的远程 home 接口：如果 bean 类有一个远程 home 接口，且没有使用元数据注释符指定远程 home 接口，那么 Bean 提供者必须在 home 元素中指定企业 bean 远程 home 接口的全称。
- 企业 bean 的远程接口：如果 bean 类有一个远程接口，且没有使用元数据注释符指定远程 home 接口，那么 Bean 提供者必须在 remote 元素中指定企业 bean 远程接口的全称。
- 企业 bean 的本地 home 接口：如果 bean 类有一个本地 home 接口，且没有使用元数据注释符指定本地 home 接口，那么 Bean 提供者必须在 local-home 元素中指定企业 bean 本地 home 接口的全称。
- 企业 bean 的本地接口：如果 bean 类有一个本地接口，且没有使用元数

据注释符指定本地 home 接口，那么 Bean 提供者必须在 local 元素中指定企业 bean 本地接口的全称。

- 企业 bean 的 web 服务终端接口：如果 bean 类有一个 web 服务终端接口，且没有在 bean 类上使用元数据注释符指定这个接口，那么 Bean 提供者必须在 service-endpoint 元素中指定企业 bean 的 web 服务终端接口的全称。这个元素只用于无状态会话 bean。
- 企业 bean 的类型：企业 bean 的类型是：会话，实体和消息驱动。如果没有使用注释符来指定 bean 的类型，则 Bean 提供者必须使用相应的 session，entity，message-driven 元素来声明企业 bean 的结构信息。如果 bean 的类型已经通过 Stateless，Stateful 或 MessageDriven 注释符指定，那么它的类型不能通过部署描述来覆盖。如果指定 Bean 的类型（以及它的会话类型），则它们必须和注释符中指定的类型一致。
- 重入指示（Re-entrancy indication）：Bean 提供者必须指定 EJB2.1 实体 bean 是否可以重入。会话 bean 和消息驱动 bean 从来都不能重入。
- 会话 bean 的状态管理类型：如果企业 bean 是会话 bean 且这个 bean 类还没有用 Stateless 或 Stateful 注释符注释，那么 Bean 提供者必须使用 session-type 元素来指定会话 bean 是有状态的还是无状态的。
- 会话或消息驱动 bean 的事务分割类型：如果企业 bean 是会话 bean 或消息驱动 bean，那么 Bean 提供者可以使用 transaction-type 元素来声明事务分割是由企业 bean 还是由容器来执行。如果既没有使用 TransactionType 注释符又没有使用 transaction-type 部署描述元素，那么 bean 将使用容器管理的事务分割。
- 企业 bean 的持久化管理：如果企业 bean 是 EJB2.1 实体 bean，那么 Bean 提供者必须使用 persistence-type 元素来声明持久化管理是由企业 bean 管理还是由容器管理。
- 企业 bean 的主键类：如果企业 bean 是 EJB2.1 实体 bean，那么 Bean 提供者在 prim-key-class 元素中指定实体 bean 主键类的全称。Bean 提供者必须为 bean 管理持久化的实体指定主键类。

- 实体 bean 的抽象 schema 名称：如果企业 bean 是使用容器管理持久化和 cmp-version 2.x 的实体 bean，那么 Bean 提供者必须使用 abstract-schema-name 元素指定实体 bean 的抽象 schema 名称。
- 容器管理的字段：如果企业 bean 是使用容器管理持久化和 cmp-version 2.x 的实体 bean，那么 Bean 提供者必须使用 cmp-field 元素来指定容器管理的字段。
- 容器管理的关系：如果企业 bean 是使用容器管理持久化和 cmp-version 2.x 的实体 bean，那么 Bean 提供者必须使用 relationships 元素来指定容器管理的关系。
- 查找和选择查询：如果企业 bean 是使用容器管理持久化和 cmp-version 2.x 的实体 bean，那么 Bean 提供者必须使用 query 元素为实体 bean 指定所有 EJB QL 查找器或选择查询，除了用于 findByPrimaryKey 方法的查询。
- 环境条目：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的环境条目，按照在节 16.3.1 中的规定。
- 资源管理器连接工厂引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的资源管理器连接工厂引用，按照在节 16.7.1 中的规定。
- 资源环境引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的与资源关联的受管理对象引用，按照在节 16.8.1 中的规定。
- EJB 引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的其他企业 bean 的远程 home 引用，按照在节 16.5.1 中的规定。
- EJB 本地引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的其他企业 bean 的本地 home 引用，按照在节 16.5.1 中的规定。
- Web 服务引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的 web 服务接口引用，按照在节 16.6 中的规定。
- 持久化单元引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的用于持久化单元的实体管理器引用，按照在节 16.10 中的

规定。

- 持久化上下文引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的用于持久化上下文的实体管理器引用，按照在节 16.11 中的规定。
- 消息目的地引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的消息目的地引用，按照在节 16.9.1 中的规定。
- 安全角色引用：Bean 提供者必须声明企业 bean 的所有未通过元数据注释符定义的安全角色引用，按照在节 17.2.5.3 中的规定。
- 消息驱动 bean 的配置属性：Bean 提供者可以为部署者提供输入，这个输入指示如何在操作环境中将消息驱动 bean 配置成活动的。JMS 消息驱动 bean 的活动配置属性包括 bean 的目的地类型，消息选择器和确认模式。其他 bean 类型可以使用不同的属性。参见【15】。
- 消息驱动 bean 的目的地：Bean 提供者可以建议应用组装者，在消息驱动 bean 链接目的地时，消息驱动 bean 应当被分配哪个目的地类型。
- 拦截器：Bean 提供者必须声明未通过元数据注释符声明的所有拦截器类和方法。

由 Bean 提供者生产的部署描述必须遵循在节 19.5 的 XML Schema 定义或遵循本规范以前版本的 XML Schema 或 DTD 定义。部署描述的内容必须遵循在 XML Schema 或 DTD 和本规范其他地方规定的语义规则。

19.3 应用组装者的责任

应用组装者将企业 bean 组装到一个单个的部署单元。应用组装者的输入是由一个或多个 Bean 提供者提供的一个或多个 ejb-jar 文件，它的输出也是一个或多个 ejb-jar 文件，或将一个输入 ejb-jar 文件分割成多个输出 ejb-jar 文件。每个输出的 ejb-jar 文件都是提供给部署者的部署单元，或者是提供给另一个应用组装者的部分已组装应用。

Bean 提供者和应用组装者可以是同一个人或组织。在这种情况下，个人或组织执行在这里和前面章节中描述的职责。

应用组装者可以修改下面的由 bean 提供者指定的信息：

- 环境条目的值：应用组装者可以改变已有的值和/或定义环境属性的新值。
- 描述字段：应用组装者可以改变已有的 description 元素的值，或创建一个新的。
- EJB2.x 实体 bean 的关系名称：如果多个 ejb-jar 文件使用同一个关系名称且它们被合并成一个 ejb-jar 文件，那么应用组装者负责修改在 ejb-relaition-name 元素中定义的关系名。

消息驱动 bean 的消息选择器：应用组装者可以进一步限定，但不能替换 JMS 消息驱动 bean 的 messageSelector activattion-congfig-property 元素的值——不管这定义在元数据注释符中还是定义在部署描述中。

通常情况下，应用组装者从不会改变下面的值：

- 企业 bean 的抽象 Schema 名：应用组装者不应当改变定义在 abstract-schema-name 元素中的企业 bean 名称，因为 EJB QL 查询可能依赖这个元素的内容。
- 关系角色源元素：应用组装者不应当改变 relationship-role-source 元素中的 ejb-name 的值。

如果应用组装者在合并两个 ejb-jar 文件时为了解决命名冲突而必须改变这些元素，那么应用组装者也必须改变依赖于被改变元素值的所有 ejb-ql 查询字符串。

应用组装者通常不需要修改在节 19.2 中列出的其他信息。

应用组装者可以但不要求指定下面的应用组装信息：

- 企业 bean 引用的绑定：应用组装者可以将一个企业 bean 引用链接到 ejb-jar 文件或同一个 Java EE 应用单元中的另一个企业 bean 上。应用组装者通过在引用 bean 中增加 ejb-link 元素来创建这个链接。应用组装者使用被引用 bean 的 ejb-name 来建立链接。如果有多个企业 bean 具有相同的 ejb-name，那么应用组装者使用带 ejb-jar 文件的路径名来建立。路径名是相对于引用方的 ejb-jar 文件。应用组装者将被引用 bean 的

ejb-name 附加在路径名的后面，用#分开。这可以让具有相同名字的 bean 有唯一的标识。

- 消息目的地引用的链接：应用组装者可以通过在 ejb-jar 文件或同一个 Java EE 应用单元中的共同消息目的地将消息消费者和生产者链接起来。应用组装者通过在引用 bean 中增加 message-destination-link 元素来创建链接。
- 安全角色：应用组装者可以定义一个或多个安全角色。安全角色定义了向企业 bean 客户端推荐的安全角色。应用组装者使用 security-role 元素来定义安全角色。
- 方法权限：应用组装者可以定义方法权限。方法权限是安全角色与企业 bean 业务接口、home 接口、组件接口和/或 web 服务终端的方法间的二元关系。应用组装者使用 method-permission 元素来定义方法权限。应用组装者可以增强或覆盖由 Bean 提供者定义的方法权限——无论是在元数据注释符中还是在部署描述中定义的。
- 安全角色引用的链接：如果应用组装者在部署描述中定义了安全角色，那么应用组装者可以将 Bean 提供者声明的安全角色引用链接到安全角色。应用组装者使用 role-link 元素定义这些链接。
- 安全标识：应用组装者可以指定调用者的安全标识是否用于企业 bean 方法的执行，或者是否使用指定的 run-as 安全标识。应用组装者可以覆盖 Bean 提供者定义的安全标识——无论是在元数据注释符中还是在部署描述中。
- 事务属性：应用组装者可以定义那些需要容器管理事务分割的企业 bean 方法的事务属性，这些方法包括企业 bean 的业务接口，home 接口，组件接口，web 服务终端和 TimeObject 接口的方法。所有由 Bean 提供者声明事务类型为 Container 的实体 bean 和会话和消息驱动 bean 都需要容器管理的事务分割。应用组装者使用 container-transaction 元素来声明事务属性。
- 拦截器：应用组装者可以覆盖、增强和/或重排由 Bean 提供者定义的拦

截器方法——无论是在元数据注释符中还是在部署描述中。

如果一个 ejb-jar 文件保护了应用组装信息，那么应用组装者可以改变在这个文件中的应用组装信息。(在输入的 ejb-jar 文件是由其他的应用组装者生产时，可能会发生这种情况)

由 Bean 提供者生产的部署描述必须遵循在节 19.5 中定义的 XML Schema 或本规范以前版本定义的 XML Schema 或 DTD。部署描述的内容必须遵循在 XML Schema 或 DTD 以及本规范其他地方规定的语义规则。

19.4 容器提供者的责任

容器提供者提供读取和导入包含在 XML 部署描述中的信息的工具。

所有的 EJB3.0 实现必须支持 EJB2.1，EJB2.0 和 EJB1.1 以及 EJB3.0 部署描述。EJB2.1，EJB2.0 和 EJB1.1 部署描述的定义可以在 EJB2.1 规范【3】中找到。

19.5 部署描述的 XML Schema

本节提供了 EJB3.0 部署描述的 XML Schema。在 XML Schema 中的注释规定了 XML Schema 机制不方便表达的语法和语义的要求。

XML 元素的内容通常都是大小写敏感的（也就是说，除非另有说明）。意思是，例如，必须使用

```
<transaction-type>Container</transaction-type>
```

而不是

```
<transaction-type>container</transaction-type>
```

所有有效的 ejb-jar 部署描述必须遵循下面的 XML Schema 定义，或本规范以前版本的 DTD 定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://java.sun.com/xml/ns/javaee"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified"
```

```
version="3.0">
<xsd:annotation>
<xsd:documentation>
@(#)ejb-jar_3_0.xsds1.51 02/23/06
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
<xsd:documentation>
Copyright 2003-2006 Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A
All rights reserved.
Sun Microsystems, Inc. has intellectual property rights
relating to technology described in this document. In
particular, and without limitation, these intellectual
property rights may include one or more of the U.S. patents
listed at http://www.sun.com/patents and one or more
additional patents or pending patent applications in the
U.S. and other countries.
This document and the technology which it describes are
distributed under licenses restricting their use, copying,
distribution, and decompilation. No part of this document
may be reproduced in any form by any means without prior
written authorization of Sun and its licensors, if any.
Third-party software, including font technology, is
copyrighted and licensed from Sun suppliers.
Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,
JavaServer Pages, Enterprise JavaBeans and the Java Coffee
Cup logo are trademarks or registered trademarks of Sun
Microsystems, Inc. in the U.S. and other countries.
Federal Acquisitions: Commercial Software - Government Users
Subject to Standard License Terms and Conditions.
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
<xsd:documentation>
<![CDATA[
The deployment descriptor must be named "META-INF/ejb-jar.xml" in
the EJB's jar file. All EJB deployment descriptors must indicate
the ejb-jar schema by using the Java EE namespace:
http://java.sun.com/xml/ns/javaee
```

and by indicating the version of the schema by
using the version element as shown below:
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
version="3.0">
...
</ejb-jar>
The instance documents may indicate the published version of
the schema using the xsi:schemaLocation attribute for the
Java EE namespace with the following location:
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:annotation>
<xsd:documentation>
下面的约定应用于所有的Java EE部署描述元素，除非有其他说明。
-在元素中指定同一个JAR文件中文件的路径名（也就是，那些不以“/”开头的路径名）认为是相对于JAR文件命名空间。绝对文件名（也就是以“/”开头）指的是JAR文件命名空间的根路径中的名字。通常，相对名字是最好的方式。例外情况是，为了与Servlet API一致，在.war文件中使用绝对名字是最好的。
</xsd:documentation>
</xsd:annotation>
<xsd:include schemaLocation="javaee_5.xsd"/>
<!-- ** -->
<xsd:element name="ejb-jar" type="javaee:ejb-jarType">
<xsd:annotation>
<xsd:documentation>
这是ejb-jar部署描述的根.
</xsd:documentation>
</xsd:annotation>
<xsd:key name="ejb-name-key">
<xsd:annotation>
<xsd:documentation>
ejb-name元素包含了企业bean的名字。在ejb-jar文件中，这个名字必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:enterprise-beans/*"/>
<xsd:field xpath="javaee:ejb-name"/>
</xsd:key>

```
<xsd:keyref name="ejb-name-references"
refer="javaee:ejb-name-key">
<xsd:annotation>
<xsd:documentation>
keyref表示来自relationship-role-source的引用必须是一个定义在enterprise-beans元素范围内的ejb-name。
</xsd:documentation>
</xsd:annotation>
<xsd:selector
xpath=".//javaee:ejb-relationship-role/javaee:relationship-role-source"/>
<xsd:field
xpath="javaee:ejb-name"/>
</xsd:keyref>
<xsd:key name="role-name-key">
<xsd:annotation>
<xsd:documentation>
指定role-name-key 是为了在security-role-refs中引用。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:assembly-descriptor/javaee:security-role"/>
<xsd:field xpath="javaee:role-name"/>
</xsd:key>
<xsd:keyref name="role-name-references"
refer="javaee:role-name-key">
<xsd:annotation>
<xsd:documentation>
keyref表示security-role-ref到role-name的引用.
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:enterprise-beans/*/javaee:security-
role-ref"/>
<xsd:field xpath="javaee:role-link"/>
</xsd:keyref>
</xsd:element>

<!-- ********************************************************** -->
<xsd:complexType name="activation-config-propertyType">
<xsd:annotation>
<xsd:documentation>
activation-config-propertyType包含了一个用于消息驱动bean的名称/值配置属性对。由消息类型来识别的这些用于消息驱动bean的属性。
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
<xsd:element name="activation-config-property-name"
type="javaee:xsdStringType">
<xsd:annotation>
<xsd:documentation>
activation-config-property-name元素包含了消息驱动bean的激活配置属性的名字。对于JMS消息驱动bean来说，下面的属性是可以被识别的：
acknowledgeMode,messageSelector,destinationType, subscriptionDurability
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="activation-config-property-value"
type="javaee:xsdStringType">
<xsd:annotation>
<xsd:documentation>
activation-config-property-value 元素包含消息驱动bean激活配置属性的值。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************* -->
<xsd:complexType name="activation-configType">
<xsd:annotation>
<xsd:documentation>
activation-configType定义了关于消息驱动bean在操作环境中期望的配置属性信息 。它可以包括消息确认，消息选择器，期望的目的地类型等等信息。
配置信息用名称/值配置属性来表达。
根据消息类型来识别这些用于特定消息驱动bean的属性。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="activation-config-property"
type="javaee:activation-config-propertyType"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
```

```
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="application-exceptionType">
<xsd:annotation>
<xsd:documentation>
application-exceptionType 声明一个应用异常。这个声明包括:
-异常类。当容器收到这种类型的异常时，它将它作为应用异常转发到客户端，而不管它是受检查异常还是不检查的异常。
-一个可选的rollback元素。如果这个元素被设置成true，那么容器必须在转发异常之前回滚事务。如果没有指定，则缺省是false。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="exception-class"
type="javaee:fully-qualified-classType"/>
<xsd:element name="rollback"
type="javaee:true-falseType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="around-invokeType">
<xsd:annotation>
<xsd:documentation>
around-invoke类型指定了一个在调用ejb期间要作为调用的一部分而被调用的方法。注意，每个类只可以有一个环绕调用方法，且这个方法不能被重载。如果没有指定class元素，那么这个定义回调的类认为是环绕调用方法定义所在范围内的拦截器类或组件类。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="class"
type="javaee:fully-qualified-classType"
minOccurs="0"/>
<xsd:element name="method-name"
type="javaee:java-identifierType"/>
</xsd:sequence>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="assembly-descriptorType">
<xsd:annotation>
<xsd:documentation>
```

assembly-descriptorType定义了application-assembly信息。application-assembly信息由下面几部分组成：安全角色的定义，方法权限的定义，使用容器管理事务分割的企业bean的事务属性的定义，拦截器绑定的定义，一个被从调用中排除的方法列表和应被看作是应用异常的异常列表。
所有的部分都是可选的。如果列表是空的，那么它们就是被忽略的。
在部署描述中提供assembly-descriptor对ejb-jar文件生产者来说是可选的。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="security-role"
type="javaee:security-roleType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="method-permission"
type="javaee:method-permissionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="container-transaction"
type="javaee:container-transactionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="interceptor-binding"
type="javaee:interceptor-bindingType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="message-destination"
type="javaee:message-destinationType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="exclude-list"
type="javaee:exclude-listType"
minOccurs="0"/>
<xsd:element name="application-exception"
type="javaee:application-exceptionType"
minOccurs="0"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="cmp-fieldType">
<xsd:annotation>
<xsd:documentation>
```

cmp-fieldType描述了一个容器管理的字段。cmp-fieldType包含一个可选的字段描述和字段的名称。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="field-name"
type="javaee:java-identifierType">
<xsd:annotation>
<xsd:documentation>
field-name元素指定了容器管理字段的名字。使用cmp-version 2.x的实体bean的cmp-field的名称必须以小写字母开头。这些字段由那些以"get"或"set"开头的以field-name名称第一个字母大写组成的方法访问。
使用cmp-version 1.x的实体bean的cmp-field的名字必须是企业bean类或它超类中的一个公共（public）字段。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="cmp-versionType">
<xsd:annotation>
<xsd:documentation>
cmp-versionType 指定了使用容器管理持久化的实体bean的版本。它由cmp-version元素使用。值必须是1.x或2.x。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="1.x"/>
<xsd:enumeration value="2.x"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="cmr-field-typeType">
<xsd:annotation>
<xsd:documentation>

cmr-field-type 元素指定了实体bean类中逻辑关系字段的collection-value值类。使用cmr-field-typeType元素的值必须是java.util.Collection 或java.util.Set.

```
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="java.util.Collection"/>
<xsd:enumeration value="java.util.Set"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************** -->
<xsd:complexType name="cmr-fieldType">
<xsd:annotation>
<xsd:documentation>
```

cmr-fieldType 描述了关系的bean提供者视图。它由两部分组成：可选的描述，关系中源字段的名字和类型。cmr-field-name元素对应于用于关系的getter和setter方法的名称。cmr-field-type element只用于集合值的cmr-field。它指定了集合的类型。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="cmr-field-name"
type="javaee:string">
<xsd:annotation>
<xsd:documentation>
```

cmr-field-name元素指定了实体bean类中逻辑关系字段的名称。cmr-field的名称必须以小写字母开头。这个字段由它的getter或setter方法访问。

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmr-field-type"
type="javaee:cmr-field-typeType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************** -->
<xsd:complexType name="container-transactionType">
<xsd:annotation>
```

<xsd:documentation>
container-transactionType指定了容器如果管理企业bean方法调用的事务范围。它由三部分组成：可选的描述，method元素列表和事务属性。事务属性应用到所有列出的方法上。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="method"
type="javaee:methodType"
maxOccurs="unbounded"/>
<xsd:element name="trans-attribute"
type="javaee:trans-attributeType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="ejb-classType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
ejb-classType 包含了企业bean类的全称。它由ejb-class元素使用。
例如：
<ejb-class>com.wombat.empl.EmployeeServiceBean</ejb-class>
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:fully-qualified-classType"/>
</xsd:simpleContent>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="ejb-jarType">
<xsd:annotation>
<xsd:documentation>
ejb-jarType定义了EJB部署描述的根元素。它包含：
-一个可选的ejb-jar文件的描述。
-一个可选的display名称。
-一个可选的图标，它包含了一个小的和大的图标文件名称。
-关于所有未通过注释符指定的企业bean的结构信息。

- 关于拦截器类的结构信息。
-容器管理的关系，如果有的话。
- 一个可选的application-assembly 。
- 一个可选的ejb-client-jar的名称。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="javaee:descriptionGroup"/>
<xsd:element name="enterprise-beans"
type="javaee:enterprise-beansType"
minOccurs="0"/>
<xsd:element name="interceptors"
type="javaee:interceptorsType"
minOccurs="0"/>
<xsd:element name="relationships"
type="javaee:relationshipsType"
minOccurs="0">
<xsd:unique name="relationship-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-relation-name包含了关系的名称。名字在关系中必须是唯一的。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:ejb-relation"/>
<xsd:field xpath="javaee:ejb-relation-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="assembly-descriptor"
type="javaee:assembly-descriptorType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
在部署描述中提供assembly-descriptor 对ejb-jar文件生产者来说是可选的。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-client-jar"
type="javaee:pathType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
<![CDATA[

可选的ejb-client-jar元素指定一个JAR文件，这个文件包含了客户端访问JAR内企业bean所必需的类文件。

例如：

```
<ejb-client-jar>employee_service_client.jar
</ejb-client-jar>
]]>
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="version"
type="javaee:dewey-versionType"
fixed="3.0"
use="required">
<xsd:annotation>
<xsd:documentation>
```

Version指定了实例文档必须遵循的 EJB规范的版本。这个信息可以让部署工具根据EJB Schema的版本验证EJB部署描述。

```
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="metadata-complete" type="xsd:boolean">
<xsd:annotation>
<xsd:documentation>
```

metadata-complete 属性定义了用于这个模块的部署描述和其它相关部署描述是否完成，或者是否应当检查这个模块可获得的和与应用一起打包的类文件上指定的部署信息。如果metadata-complete被设置成true，那么部署工具必须忽略所有指定部署信息的注释符。如果它被设置成false，那么部署工具必须按照规范检查应用中类的注释符。

```
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="ejb-nameType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
```

ejb-nameType指定了企业bean的名称。它由ejb-name元素使用。这个名字被ejb-jar文件生产者分配，用于命名ejb-jar文件部署描述中的企业bean。 在同一个ejb-jar文件中，这个名字必须是唯一的。在部署描述中的ejb-name和被部署者分配到企业beanhome的JNDI名字间没有特别的关系。用于实体bean的名字必须遵循NMTOKEN的文法规则。

例如：

```
<ejb-name>EmployeeService</ejb-name>
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:xsdNMTOKENType"/>
</xsd:simpleContent>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="ejb-relationType">
<xsd:annotation>
<xsd:documentation>
ejb-relationType描述了两个使用容器管理持久化的实体bean间的关系。它由ejb-relation 元素使用。它包含了一个描述；一个可选的ejb-relation-name元素；和确切的两个关系角色声明，它们由ejb-relationship-role元素定义。如果指定了关系的名称，则在ejb-jar文件中这个名称必须是唯一的。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="ejb-relation-name"
type="javaee:string"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
ejb-relation-name 元素为关系在ejb-jar文件中提供了一个唯一的名字。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-relationship-role"
type="javaee:ejb-relationship-roleType"/>
<xsd:element name="ejb-relationship-role"
type="javaee:ejb-relationship-roleType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="ejb-relationship-roleType">
<xsd:annotation>
```

<xsd:documentation>
<![CDATA[
ejb-relationship-roleType 描述了一个关系内的角色。每个关系都有两个角色。
ejb-relationship-roleType包含了一个可选的描述；一个可选的关系角色名称； 一个角色多边规范；一个可选的用于角色的层级删除规范； 角色源； 和一个 cmr-field的声明，如果有的话，根据从角色源的视角看关系的哪个其他边被访问。多边和role-source元素是必需的。relationship-role-source元素通过ejb-name元素指派一个 实体bean。对于双向关系，关系的两个角色都必须声明一个relationship-role-source元素，它根据哪个关系被反问来指定cmr-field。在ejb-relationship-role 中没有指定cmr-field元素说明关系是单向的，且参与关系的实体bean不知道这个关系。
例如：
<ejb-relation>
<ejb-relation-name>Product-LineItem</ejb-relation-name>
<ejb-relationship-role>
<ejb-relationship-role-name>product-has-lineitems
</ejb-relationship-role-name>
<multiplicity>One</multiplicity>
<relationship-role-source>
<ejb-name>ProductEJB</ejb-name>
</relationship-role-source>
</ejb-relationship-role>
</ejb-relation>
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="ejb-relationship-role-name"
type="javaee:string"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
ejb-relationship-role-name 元素定义了在ejb-relation中唯一的角色名称。不同的关系可以使用同一个角色名。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="multiplicity"
type="javaee:multiplicityType"/>
<xsd:element name="cascade-delete"

```
type="javaee:emptyType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
cascade-delete 元素在特定的关系中指定了一个或多个实体bean的生命期是依赖于另一个实体bean
的生命期。cascade-delete 元素指定在包含在ejb-relation元素中的ejb-relationship-role
元素，在ejb-relation元素中其他的ejb-relationship-role元素指定了一对多的多方。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="relationship-role-source"
type="javaee:relationship-role-sourceType"/>
<xsd:element name="cmr-field"
type="javaee:cmr-fieldType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="enterprise-beansType">
<xsd:annotation>
<xsd:documentation>
enterprise-beansType 声明了一个或多个企业bean。每个企业bean可以是会话，实体或消息驱动
bean。
</xsd:documentation>
</xsd:annotation>
<xsd:choice maxOccurs="unbounded">
<xsd:element name="session"
type="javaee:session-beanType">
<xsd:unique name="session-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env
上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:ejb-local-ref"/>
<xsd:field xpath="javaee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="session-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
```

ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。

</xsd:documentation>

</xsd:annotation>

<xsd:selector xpath="javaee:ejb-ref"/>

<xsd:field xpath="javaee:ejb-ref-name"/>

</xsd:unique>

<xsd:unique name="session-resource-env-ref-uniqueness">

<xsd:annotation>

<xsd:documentation>

resource-env-ref-name元素指定了资源环境引用的名字；它的值是在组件代码中使用的环境条目名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。

</xsd:documentation>

</xsd:annotation>

<xsd:selector xpath="javaee:resource-env-ref"/>

<xsd:field xpath="javaee:resource-env-ref-name"/>

</xsd:unique>

<xsd:unique name="session-message-destination-ref-uniqueness">

<xsd:annotation>

<xsd:documentation>

message-destination-ref-name元素指定了消息目的地引用的名称；它的值是在组件代码中使用的消息目的地名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。

</xsd:documentation>

</xsd:annotation>

<xsd:selector xpath="javaee:message-destination-ref"/>

<xsd:field xpath="javaee:message-destination-ref-name"/>

</xsd:unique>

<xsd:unique name="session-res-ref-name-uniqueness">

<xsd:annotation>

<xsd:documentation>

res-ref-name指定了资源管理器连接工厂引用的名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。

</xsd:documentation>

</xsd:annotation>

<xsd:selector xpath="javaee:resource-ref"/>

<xsd:field xpath="javaee:res-ref-name"/>

</xsd:unique>

<xsd:unique name="session-env-entry-name-uniqueness">

<xsd:annotation>

<xsd:documentation>

env-entry-name元素包含了组件环境条目的名字。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。

</xsd:documentation>

```
</xsd:annotation>
<xsd:selector xpath="javaee:env-entry"/>
<xsd:field xpath="javaee:env-entry-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="entity"
type="javaee:entity-beanType">
<xsd:unique name="entity-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env
上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:ejb-local-ref"/>
<xsd:field xpath="javaee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env
上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:ejb-ref"/>
<xsd:field xpath="javaee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-resource-env-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>
resource-env-ref-name指定了资源环境引用的名称。它的值是用在组件代码中的环境条目名称。这个
名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:resource-env-ref"/>
<xsd:field xpath="javaee:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-message-destination-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>
message-destination-ref-name指定了消息目的地引用的名称。它的值是用在组件代码中的消息目
的地名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
```

```
</xsd:annotation>
<xsd:field xpath="javaee:message-destination-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
res-ref-name指定了资源管理器连接工厂引用的名称。这个名字是相对于java:comp/env上下文的
JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:resource-ref"/>
<xsd:field xpath="javaee:res-ref-name"/>
</xsd:unique>
<xsd:unique name="entity-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
env-entry-name指定了组件环境引用的名称。这个名字是相对于java:comp/env上下文的JNDI名，在
组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:env-entry"/>
<xsd:field xpath="javaee:env-entry-name"/>
</xsd:unique>
</xsd:element>
<xsd:element name="message-driven"
type="javaee:message-driven-beanType">
<xsd:unique name="messaged-ejb-local-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env
上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:ejb-local-ref"/>
<xsd:field xpath="javaee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-ejb-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-ref-name元素包含了EJB引用的名字。EJB引用是组件环境的一个条目，它相对于java:comp/env
上下文。这个名字在组件内必须是唯一的。建议这个名字以"ejb/"开头。
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:selector xpath="javaee:ejb-ref"/>
<xsd:field xpath="javaee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-resource-env-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>
resource-env-ref-name元素指定了资源环境引用的名称。它的值是用在组件代码中环境条目名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:resource-env-ref"/>
<xsd:field xpath="javaee:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-message-destination-ref-uniqueness">
<xsd:annotation>
<xsd:documentation>
message-destination-ref-name指定了消息目的地引用的名称。它的值是用在组件代码中的消息目的地名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:message-destination-ref"/>
<xsd:field xpath="javaee:message-destination-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-res-ref-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
res-ref-name指定了资源管理器连接工厂引用的名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:resource-ref"/>
<xsd:field xpath="javaee:res-ref-name"/>
</xsd:unique>
<xsd:unique name="messaged-env-entry-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
env-entry-name指定了组件环境引用的名称。这个名字是相对于java:comp/env上下文的JNDI名，在组件中必须唯一。
</xsd:documentation>
</xsd:annotation>
<xsd:selector xpath="javaee:env-entry"/>
<xsd:field xpath="javaee:env-entry-name"/>
</xsd:unique>
```

```
</xsd:element>
</xsd:choice>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="entity-beanType">
<xsd:annotation>
<xsd:documentation>
```

entity-beanType声明了一个实体bean。声明由以下内容组成：

- 可选的描述。
- 可选的显示名。
- 可选的图标元素，这个元素包含了一个小的和大的图标文件名。
- 一个分配到部署描述中企业bean的唯一名称。
- 一个可选的mapped-name元素，它可以被用于提供供应商特有的部署信息，例如实体bean远程home接口的jndi-name。这个元素不要求所有的实现都支持它。使用这个元素的应用都是不可移植的。
- 实体bean远程home和remote接口的名字，如果有的话。
-实体bean本地home和local接口的名字，如果有的话。
- 实体bean的实现类。
- 可选的实体bean的持久化管理类型。如果没有指定这个元素，则缺省是容器。
- 实体bean的主键类名。
- 实体bean重入指示。
- 可选的实体bean cmp-version规范。
- 可选的实体bean抽象schema名字规范。
- 可选的容器管理字段的列表。
- 可选的主键字段规范。
- 可选的bean环境条目声明。
- 可选的bean的EJB引用声明。
- 可选的bean的本地EJB引用声明。
- 可选的bean的web服务引用声明。
- 可选的安全角色引用声明。
- 可选的用于bean方法执行的安全标识声明。
- 可选的bean资源管理器连接工厂引用声明。
- 可选的bean的资源环境引用声明。
- 可选的bean的消息目的地引用声明。
- 可选的用于finder和select 方法的查询声明集，这些方法是用于cmp-version 2.x的实体bean。必须为使用容器管理持久化和cmp-version 2.x的实体bean指定可选的abstract-schema-name 元素。

如果实体的持久化类型是容器，那么在部署描述中可以出现可选的primkey-field。

如果实体的persistence-type 是容器（Container），那么在部署描述中可以出现可选的cmp-version 元素。如果persistence-type是Container，且没有指定cmp-version，那么它的值缺省是2.x。

如果实体bean的cmp-version是1.x，那么必须指定可选的home和remote元素。

如果实体bean有远程home和remote接口，那么必须指定可选的home和remote元素。
如果实体bean有本地home和local接口，那么必须指定可选的local-home和local接口。
必须同时指定local-home和local元素，或home和remote元素。
如果persistence-type是Container且cmp-version是2.x，同时已经为实体bean定义了除了findByPrimaryKey方法以外的其他查询方法，那么必须指定可选的query元素。
其他的可选元素，如果它们出现为空则忽略他们。
如果persistence-type是Container且cmp-version是1.x ，那么至少指定一个cmp-field元素。
如果persistence-type是Bean，则可以不出现cmp-field元素。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="javaee:descriptionGroup"/>
<xsd:element name="ejb-name"
type="javaee:ejb-nameType"/>
<xsd:element name="mapped-name"
type="javaee:xsdStringType"
minOccurs="0"/>
<xsd:element name="home"
type="javaee:homeType"
minOccurs="0"/>
<xsd:element name="remote"
type="javaee:remoteType"
minOccurs="0"/>
<xsd:element name="local-home"
type="javaee:local-homeType"
minOccurs="0"/>
<xsd:element name="local"
type="javaee:localType"
minOccurs="0"/>
<xsd:element name="ejb-class"
type="javaee:ejb-classType"/>
<xsd:element name="persistence-type"
type="javaee:persistence-typeType"/>
<xsd:element name="prim-key-class"
type="javaee:fully-qualified-classType">
<xsd:annotation>
<xsd:documentation>
prim-key-class元素包含了实体bean主键类的全称。如果主键类的定义在部署时被改变，那么prim-key-class元素应当被指定为java.lang.Object.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="reentrant"

```
type="javaee:true-falseType">
<xsd:annotation>
<xsd:documentation>
Reentrant元素指定了实体bean是否是可重入的。Reentrant元素必须是true或false。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmp-version"
type="javaee:cmp-versionType"
minOccurs="0"/>
<xsd:element name="abstract-schema-name"
type="javaee:java-identifierType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
abstract-schema-name元素指定了使用cmp-version 2.x的实体bean的抽象schema类型的名称。
它用于EJB QL查询。例如，用于一个本地接口是com.acme.commerce.Order的实体的
abstract-schema-name 是 Order.
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="cmp-field"
type="javaee:cmp-fieldType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="primkey-field"
type="javaee:string"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
primkey-field 元素用于指定使用容器管理持久化的实体bean的主键字段名称。
primkey-field 必须是声明在cmp-field元素中的字段，且这个字段类型必须和主键类型一致。
如果主键映射到多个容器管理的字段（也就是，主键是组合主键），则不使用primkey-field元素。在
这种情况下，主键类的所有字段必须是public的，且他们的名字必须对应于组成主键的实体bean的字段
名。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
<xsd:element name="security-role-ref"
type="javaee:security-role-refType"
minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="security-identity"
```

```
type="javaee:security-identityType"
minOccurs="0"/>
<xsd:element name="query"
type="javaee:queryType"
minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="exclude-listType">
<xsd:annotation>
<xsd:documentation>
exclude-listType指定了一个或多个方法，这些方法被组装者标记为不可调用的。
如果方法权限关系包含了在排除列表中的方法，那么部署这应当将这些方法认为是不可调用的。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="method"
type="javaee:methodType"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="init-methodType">
<xsd:sequence>
<xsd:element name="create-method"
type="javaee:named-methodType"/>
<xsd:element name="bean-method"
type="javaee:named-methodType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="interceptor-bindingType">
<xsd:annotation>
<xsd:documentation>
interceptor-bindingType 元素描述了拦截器类与ejb-jar中bean之间的绑定
```

它包括：

-一个可选的描述.

- ejb-jar中的ejb的名字或通配符"*",通配符用于指定拦截器绑定到ejb-jar中的所有bean上。

- 一个拦截器类列表，它被绑定到ejb-name元素的内容上，或拦截器的总体排序规范，这些拦截器定义在给定的层级或层级之上。

- 一个可选的exclude-default-interceptors元素。如果设置为true，则指定缺省的拦截器不应用到bean-class和/或业务方法上。

- 一个可选的exclude-class-interceptors元素。如果设置为true，则指定类拦截器不应用到业务方法上。

- 一个可选的method元素集合，用于描述方法级拦截器的名字/参数。

使用通配符"*"绑定到所有类上的拦截器是缺省拦截器。另外，拦截器可以被绑定到bean类（类级别拦截器）或业务方法（方法级别拦截器）级别。

拦截器到类上的绑定是递增的。如果拦截器被绑定到类级别和/或缺省级别以及方法级别，那么将同时使用类级别和/或缺省级别以及方法级别。

拦截器元素的语法有四种可能的风格： 1.

```
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

将ejb-name指定为通配符"*" 分配缺省拦截器（应用到ejb-jar中所有会话和消息驱动bean的拦截器）。

2.

```
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
</interceptor-binding>
```

这种风格用于指定与指定企业bean关联的拦截器（类级别的拦截器）。

3.

```
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
<method>
<method-name>METHOD</method-name>
</method>
</interceptor-binding>
```

这种风格用于将方法级别的拦截器与指定的企业bean建立关联。如果有多个同名的重载方法，那么这个风格的元素指的是拦截器应用到所有的同名方法上。方法级的拦截器只能被关联到bean类的业务方法上。注意，通配符"*"不能用于指定方法级的拦截器。

4.

```
<interceptor-binding>
<ejb-name>EJBNAME</ejb-name>
<interceptor-class>INTERCEPTOR</interceptor-class>
<method>
```

```
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAM-1</method-param>
<method-param>PARAM-2</method-param>
...
<method-param>PARAM-N</method-param>
</method-params>
</method>
</interceptor-binding>
```

这种风格用于将方法级的拦截器关联到指定企业bean的指定方法上。这个风格用于指定同名重载方法中一个单个方法。值PARAM-1到PARAM-N是方法入参的Java类型全称（如果方法没有入参，那么method-params元素不包含method-param元素）。数组由数组元素类型指定，类型后跟一到多个方括号对（例如，int[][]）。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="ejb-name"
type="javaee:string"/>
<xsd:choice>
<xsd:element name="interceptor-class"
type="javaee:fully-qualified-classType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="interceptor-order"
type="javaee:interceptor-orderType"
minOccurs="1"/>
</xsd:choice>
<xsd:element name="exclude-default-interceptors"
type="javaee:true-falseType"
minOccurs="0"/>
<xsd:element name="exclude-class-interceptors"
type="javaee:true-falseType"
minOccurs="0"/>
<xsd:element name="method"
type="javaee:named-methodType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ************************************************** -->
<xsd:complexType name="interceptor-orderType">
<xsd:annotation>
<xsd:documentation>
The interceptor-orderType element describes a total ordering
of interceptor classes.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="interceptor-class"
type="javaee:fully-qualified-classType"
minOccurs="1"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ************************************************** -->
<xsd:complexType name="interceptorType">
<xsd:annotation>
<xsd:documentation>
interceptorType 元素声明了关于单个拦截器类的信息。它包括：
- 一个可选的描述。
- 拦截器类的全称。
- 一个可选的环绕调用方法列表，这些方法声明在拦截器类和/或它的父类上。
- 一个可选的拦截器类和/或它的父类依赖的环境依赖。
- 一个可选的post-activate 方法列表，这些方法声明在拦截器类和/或它的父类上。
- 一个可选的pre-activate 方法列表，这些方法声明在拦截器类和/或它的父类上。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="interceptor-class"
type="javaee:fully-qualified-classType"/>
<xsd:element name="around-invoke"
type="javaee:around-invokeType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
<xsd:element name="post-activate"
type="javaee:lifecycle-callbackType"
```

```
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="pre-passivate"
type="javaee:lifecycle-callbackType"
minOccurs="0"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="interceptorsType">
<xsd:annotation>
<xsd:documentation>
interceptorsType元素声明了一个到多个由ejb-jar中组件使用的拦截器类。这个声明包括：
- 一个可选的描述。
- 一到多个拦截器元素.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="interceptor"
type="javaee:interceptorType"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="message-driven-beanType">
<xsd:annotation>
<xsd:documentation>
message-driven元素声明了一个消息驱动bean。这个声明包括：
- 一个可选的描述。
- 一个可选的显示名称。
- 可选的图标元素，这个元素包含了一个小的和大的图标文件名。
- 一个分配到部署描述中企业bean的唯一名称。
- 一个可选的mapped-name元素，它可以被用于提供供应商特有的部署信息，例如消息驱动bean消费消息的目的地的物理jndi-name。这个元素不要求所有的实现都支持它。使用这个元素的应用都是不可移植的。
- 消息驱动bean的实现类。
```

- 可选的bean的消息类型的声明。
- 可选的bean的超时方法的声明。
- 可选的消息驱动bean的事务管理类型。如果没有定义，则缺省是Container。
- 可选的bean的message-destination-type声明。
- 可选的bean的message-destination-link声明。
- 可选的消息驱动bean激活配置属性的声明。
- 可选的消息驱动bean类和/或超类的around-invoke方法列表.
- 可选的bean的环境条目的声明。
- 可选的bean的EJB引用的声明。
- 可选的bean的本地EJB引用的声明。
- 可选的bean的web服务引用的声明。
- 可选的用于bean方法执行的安全标识声明。
- 可选的bean的资源管理器连接工厂引用声明。
- 可选的bean的资源环境引用声明
- 可选的bean的消息目的地引用声明。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="javaee:descriptionGroup"/>
<xsd:element name="ejb-name"
type="javaee:ejb-nameType"/>
<xsd:element name="mapped-name"
type="javaee:xsdStringType"
minOccurs="0"/>
<xsd:element name="ejb-class"
type="javaee:ejb-classType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
```

ejb-class元素指定了bean类的全称。除非有一个组件定义注释符定义了相同的ejb-name，否则需要指定这个元素。

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="messaging-type"
type="javaee:fully-qualified-classType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
```

messaging-type元素指定了消息驱动bean的消息监听器接口。

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
```

```
<xsd:element name="timeout-method"
type="javaee:named-methodType"
minOccurs="0"/>
<xsd:element name="transaction-type"
type="javaee:transaction-typeType"
minOccurs="0"/>
<xsd:element name="message-destination-type"
type="javaee:message-destination-typeType"
minOccurs="0"/>
<xsd:element name="message-destination-link"
type="javaee:message-destination-linkType"
minOccurs="0"/>
<xsd:element name="activation-config"
type="javaee:activation-configType"
minOccurs="0"/>
<xsd:element name="around-invoke"
type="javaee:around-invokeType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
<xsd:element name="security-identity"
type="javaee:security-identityType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- **************************************************** -->
<xsd:complexType name="method-intfType">
<xsd:annotation>
<xsd:documentation>
```

method-intf 元素可以从定义在多个home和组件接口以及组件和web服务终端接口等等中（例如，同时在企业bean的local和remote接口，或同时在企业bean的home和remote接口中，等等）的具有相同名字和标识符的方法中区分出一个method元素； Local既应用到本地组件接口也应用到本地业务接口。同样，Remote既应用到远程组件接口也应用到远程业务接口。method-intf element的值必须是下面之一：

```
Home
Remote
LocalHome
Local
ServiceEndpoint
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
```

```
<xsd:restriction base="javaee:string">
<xsd:enumeration value="Home"/>
<xsd:enumeration value="Remote"/>
<xsd:enumeration value="LocalHome"/>
<xsd:enumeration value="Local"/>
<xsd:enumeration value="ServiceEndpoint"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="method-nameType">
<xsd:annotation>
<xsd:documentation>
method-nameType包含了企业bean方法的名称或字符"*"。字符"*"用于声明企业bean客户端视图接口
内的所有方法。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string"/>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="method-paramsType">
<xsd:annotation>
<xsd:documentation>
method-paramsType定义了方法参数的Java类型全称列表。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="method-param"
type="javaee:java-typeType"
minOccurs="0"
maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>
method-param元素包含一个方法参数的原始类型名称或java类型全称。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
```

<xsd:complexType name="method-permissionType">
<xsd:annotation>
<xsd:documentation>
method-permissionType指定了可以调用一到多个企业bean方法的一到多个安全角色 。
method-permissionType有下面内容组成:可选的描述，安全角色名称列表或一个表明方法是不受检查的指示器，以及method元素的列表。用于method-permissionType的安全角色必须定义在部署描述的security-role元素中，且那些方法必须是定义在企业bean业务、home、组件和/或web服务终端接口中的方法。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:choice>
<xsd:element name="role-name"
type="javaee:role-nameType"
maxOccurs="unbounded"/>
<xsd:element name="unchecked"
type="javaee:emptyType">
<xsd:annotation>
<xsd:documentation>
Unchecked元素指定方法在方法调用之前不被容器进行认证检查。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
</xsd:choice>
<xsd:element name="method"
type="javaee:methodType"
maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="methodType">
<xsd:annotation>
<xsd:documentation>
<![CDATA[
methodType用于表示企业bean的业务、home、组件和/或web服务终端接口中的方法，或消息驱动bean的消息监听器方法，或这些方法的集合。 ejb-name元素必须是声明在部署描述中的企业bean之一；可选的method-intf元素用于区分分别定义在业务、home、组件和/或web服务终端接口中且具有相同名字

的方法； method-name元素指定了方法名；可选的 method-params元素在多个同名方法中标识其中的一个方法 。

在method元素中，methodType元素有三种风格的用法：

1.

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>*</method-name>
</method>
```

这个方法用于指定企业bean的业务、home、组件和/或web服务终端接口中的所有方法。

2.

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
</method>
```

这个风格用于指定特定企业bean的特定方法。如果有多个同名的重载方法，那么这个风格指的是所有同名的方法。

3.

```
<method>
<ejb-name>EJBNAME</ejb-name>
<method-name>METHOD</method-name>
<method-params>
<method-param>PARAM-1</method-param>
<method-param>PARAM-2</method-param>
...
<method-param>PARAM-n</method-param>
</method-params>
</method>
```

这个风格用于在一系列同名方法中指定一个方法。PARAM-1到PARAM-n是方法入参的java类型全称（如果没有入参，则method-params不包含method-param元素）。数组由数组元素类型指定，后跟一到多个方括号（例如，int[][]）。如果有多个具有相同名字的方法，这个风格指所有同名的方法。

例如：

风格 1：下面的method元素指的EmployeeService企业bean的业务、home、组件和/或web服务终端接口中的所有方法：

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>*</method-name>
</method>
```

风格 2：下面的method元素指的是 EmployeeService的home接口内的所有创建方法。

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>create</method-name>
</method>
```

风格 3：下面的method元素指的是EmployeeService企业bean的home接口中的create(String firstName, String LastName)方法。

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>
```

下面的例子解释了风格3元素中有更复杂参数类型的元素。方法foobar(char s, int i, int[] iar, mypackage.MyClass mycl,mypackage.MyClass[][] myclaar)将被指定为 ：

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-name>foobar</method-name>
<method-params>
<method-param>char</method-param>
<method-param>int</method-param>
<method-param>int[]</method-param>
<method-param>mypackage.MyClass</method-param>
<method-param>mypackage.MyClass[][]</method-param>
</method-params>
</method>
```

可选的method-intf元素在需要区分分别在企业bean的业务、home、组件和/或web服务终端接口中具有相同名字和标识符（译者注：访问控制，方法名，参数类型和个数等都相同）的方法。但是，如果同一个方法既是本地业务接口的方法又是本地组件接口的方法，那么相同的属性会同时应用到两个接口上。同样的，如果同一个方法既是远程业务接口又是远程组件接口的方法，那么同样的属性也同时应用到这个两个接口上。

例如，method 元素

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Remote</method-intf>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>
```

可以被用于区分定义在远程接口的create(String, String)和定义在远程home接口中的create(String, String)，它将按如下定义：

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Home</method-intf>
```

```
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>
```

并且，定义在本地home接口中的create方法将被定义为：

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>LocalHome</method-intf>
<method-name>create</method-name>
<method-params>
<method-param>java.lang.String</method-param>
<method-param>java.lang.String</method-param>
</method-params>
</method>
```

method-intf元素可以和方法元素的三种用法一起使用。例如，下面的方法元素例子可以被用于指向EmployeeService bean的远程home接口和远程业务接口的所有方法。

```
<method>
<ejb-name>EmployeeService</ejb-name>
<method-intf>Home</method-intf>
<method-name>*</method-name>
</method>
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="ejb-name"
type="javaee:ejb-nameType"/>
<xsd:element name="method-intf"
type="javaee:method-intfType"
minOccurs="0">
</xsd:element>
<xsd:element name="method-name"
type="javaee:method-nameType"/>
<xsd:element name="method-params"
type="javaee:method-paramsType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
```

```
</xsd:complexType>
<!-- ************************************************** -->
<xsd:complexType name="multiplicityType">
<xsd:annotation>
<xsd:documentation>
multiplicityType描述了关系中角色的多方。它的值必须是 One或Many。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="One"/>
<xsd:enumeration value="Many"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ************************************************** -->
<xsd:complexType name="named-methodType">
<xsd:sequence>
<xsd:element name="method-name"
type="javaee:string"/>
<xsd:element name="method-params"
type="javaee:method-paramsType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ************************************************** -->
<xsd:complexType name="persistence-typeType">
<xsd:annotation>
<xsd:documentation>
persistence-typeType指定了实体bean的持久化管理类型。
persistence-type元素的值必须是 Bean或Container。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="Bean"/>
<xsd:enumeration value="Container"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ************************************************** -->
<xsd:complexType name="query-methodType">
```

<xsd:annotation>
<xsd:documentation>
<![CDATA[
query-method 指定了用于finder或select查询的方法。
method-name元素指定了实体bean实现类中的finder或select方法的名字。必须使用
method-params元素为query-method定义所有的method-param 。
它由query-method元素使用。
例如：
<query>
<description>Method finds large orders</description>
<query-method>
<method-name>findLargeOrders</method-name>
<method-params></method-params>
</query-method>
<ejb-ql>
SELECT OBJECT(o) FROM Order o
WHERE o.amount > 1000
</ejb-ql>
</query>
]]>
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="method-name"
type="javaee:method-nameType"/>
<xsd:element name="method-params"
type="javaee:method-paramsType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ** -->
<xsd:complexType name="queryType">
<xsd:annotation>
<xsd:documentation>
queryType定义了一个finder或select查询。它包括：
- 一个可选的查询描述。
- finder或select方法的规范。
- 一个可选的返回类型映射的规范，如果查询时用于select方法且返回的是实体对象。
- 定义查询的EJB QL查询字符串。
用EJB QL表达的查询必须使用ejb-ql元素来指导查询。如果查询不是用EJB QL表示，那么description
元素应当被用于描述查询的语义，且不应当指定ejb-ql元素。
result-type-mapping是一个可选元素。如果query-method指定一个返回值是实体对象的查询方法，

那么它才能出现。result-type-mapping元素的缺省值是"Local".

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType" minOccurs="0"/>
<xsd:element name="query-method"
type="javaee:query-methodType"/>
<xsd:element name="result-type-mapping"
type="javaee:result-type-mappingType"
minOccurs="0"/>
<xsd:element name="ejb-ql"
type="javaee:xsdStringType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="relationship-role-sourceType">
<xsd:annotation>
<xsd:documentation>
```

relationship-role-sourceType分配了关系中角色的来源。relationship-role-source元素使用relationship-role-sourceType来唯一标识一个实体bean。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="ejb-name"
type="javaee:ejb-nameType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="relationshipsType">
<xsd:annotation>
<xsd:documentation>
```

relationshipsType描述了使用容器管理持久化的实体bean所参与的关系。relationshipsType包含了一个可选的描述；和一个ejb-relation元素的列表，这些元素指定了容器管理的关系。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
```

```
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="ejb-relation"
type="javaee:ejb-relationType"
maxOccurs="unbounded">
<xsd:unique name="role-name-uniqueness">
<xsd:annotation>
<xsd:documentation>
ejb-relationship-role-name包含了关系角色的名称。这个名称在关系中必须是唯一的，但可以在不
同的关系中重用。
</xsd:documentation>
</xsd:annotation>
<xsd:selector
xpath=".//javaee:ejb-relationship-role-name"/>
<xsd:field
xpath="."/>
</xsd:unique>
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="remove-methodType">
<xsd:sequence>
<xsd:element name="bean-method"
type="javaee:named-methodType"/>
<xsd:element name="retain-if-exception"
type="javaee:true-falseType"
minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="result-type-mappingType">
<xsd:annotation>
<xsd:documentation>
result-type-mappingType用在query元素中来指定由选择方法的查询返回的抽象schema类型是否
被映射到EJBLocalObject或EJBObject类型。
它的值必须是Local或Remote
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="Local"/>
<xsd:enumeration value="Remote"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="security-identityType">
<xsd:annotation>
<xsd:documentation>
security-identityType指定了调用者的安全标识是否用于企业bean方法的执行，或者是否使用
run-as标识。它包含一个可选的安全标识的描述和规范。
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="description"
type="javaee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:choice>
<xsd:element name="use-caller-identity"
type="javaee:emptyType">
<xsd:annotation>
<xsd:documentation>
use-caller-identity元素指定了调用者的安全标识用作企业bean方法执行的安全标识。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="run-as"
type="javaee:run-asType"/>
</xsd:choice>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="session-beanType">
<xsd:annotation>
<xsd:documentation>
session-beanType声明会话bean。这个声明包括：
- 一个可选的描述。
- 一个可选的显示名称。
```

- 可选的图标元素，这个元素包含了一个小的和大的图标文件名。
- 一个分配到部署描述中企业bean的唯一名称。
- 一个可选的mapped-name元素，它可以被用于提供供应商特有的部署信息，例如会话bean的远程home/业务接口的物理jndi-name。这个元素不要求所有的实现都支持它。使用这个元素的应用都是不可移植的。
- 所有远程或本地业务接口的名称，如果有的话。
- 会话bean的远程home核remote接口的名称，如果有的话。
- 会话bean本地home和local接口的名称，如果有的话。
- 会话bean的web服务终端接口的名称，如果有的话。
- 会话bean的实现类。
- 会话bean的状态管理类型。
- 可选的bean的超时方法的声明。
- 可选的会话bean的事务管理类型。如果没有定义，则缺省是Container。
- 可选的会话bean类和/或超类的around-invoke方法列表.
- 可选的bean的环境条目的声明。
- 可选的bean的EJB引用的声明。
- 可选的bean的本地EJB引用的声明。
- 可选的bean的web服务引用的声明。
- 可选的安全角色引用声明。
- 可选的用于bean方法执行的安全标识声明。
- 可选的bean的资源管理器连接工厂引用声明。
- 可选的bean的资源环境引用声明
- 可选的bean的消息目的地引用声明。

如果可选的元素没有被指定，则忽略它们。

必须同时指定会话bean的Local-home和local元素，或同时指定home和remote元素。

service-endpoint元素只用于无状态会话bean。

```
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:group ref="javaee:descriptionGroup"/>
<xsd:element name="ejb-name"
type="javaee:ejb-nameType"/>
<xsd:element name="mapped-name"
type="javaee:xsdStringType"
minOccurs="0"/>
<xsd:element name="home"
type="javaee:homeType"
minOccurs="0"/>
<xsd:element name="remote"
type="javaee:remoteType"
minOccurs="0"/>
<xsd:element name="local-home"
type="javaee:local-homeType"
minOccurs="0"/>
```

```
<xsd:element name="local"
type="javaee:localType"
minOccurs="0"/>
<xsd:element name="business-local"
type="javaee:fully-qualified-classType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="business-remote"
type="javaee:fully-qualified-classType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="service-endpoint"
type="javaee:fully-qualified-classType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
service-endpoint元素包含了企业bean的web服务终端接口的全称。service-endpoint元素只可以
用于无状态会话bean。这个接口必须是有效的JAX-RPC服务终端接口。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="ejb-class"
type="javaee:ejb-classType"
minOccurs="0">
<xsd:annotation>
<xsd:documentation>
ejb-class元素指定了ejb类的全称。除非有一个组件定义注释符定义了相同的ejb-name，否则需要指
定这个元素。
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="session-type"
type="javaee:session-typeType"
minOccurs="0"/>
<xsd:element name="timeout-method"
type="javaee:named-methodType"
minOccurs="0"/>
<xsd:element name="init-method"
type="javaee:init-methodType"
minOccurs="0"
maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>
```

init-method元素指定了EJB3.0 bean的创建方法映射成EJB2.x风格的方法。这个元素只能用于有状态会话bean。

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="remove-method"
type="javaee:remove-methodType"
minOccurs="0"
maxOccurs="unbounded">
<xsd:annotation>
<xsd:documentation>
```

remove-method元素指定将EJB3.0 bean的remove方法映射成EJB2.x风格的方法。这个元素只能用于有状态会话bean。

```
</xsd:documentation>
</xsd:annotation>
</xsd:element>
<xsd:element name="transaction-type"
type="javaee:transaction-typeType"
minOccurs="0"/>
<xsd:element name="around-invoke"
type="javaee:around-invokeType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:group ref="javaee:jndiEnvironmentRefsGroup"/>
<xsd:element name="post-activate"
type="javaee:lifecycle-callbackType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="pre-passivate"
type="javaee:lifecycle-callbackType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="security-role-ref"
type="javaee:security-role-refType"
minOccurs="0"
maxOccurs="unbounded">
</xsd:element>
<xsd:element name="security-identity"
type="javaee:security-identityType"
minOccurs="0">
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
```

```
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="session-typeType">
<xsd:annotation>
<xsd:documentation>
session-typeType描述了会话bean是否是由状态会话或无状态会话。它由 session-type元素使用。
它的值必须是：Stateful或Stateless。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="Stateful"/>
<xsd:enumeration value="Stateless"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="trans-attributeType">
<xsd:annotation>
<xsd:documentation>
trans-attributeType 指定了容器在代理对企业bean业务方法的调用时必须如何管理事务边界。它的
值必须是下面之一：
NotSupported
Supports
Required
RequiresNew
Mandatory
Never
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="NotSupported"/>
<xsd:enumeration value="Supports"/>
<xsd:enumeration value="Required"/>
<xsd:enumeration value="RequiresNew"/>
<xsd:enumeration value="Mandatory"/>
<xsd:enumeration value="Never"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
<!-- ********************************************************** -->
<xsd:complexType name="transaction-typeType">
```

```
<xsd:annotation>
<xsd:documentation>
transaction-typeType指定了实体bean的事务管理类型。
transaction-type 的值必须是Bean或Container。
</xsd:documentation>
</xsd:annotation>
<xsd:simpleContent>
<xsd:restriction base="javaee:string">
<xsd:enumeration value="Bean"/>
<xsd:enumeration value="Container"/>
</xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>
</xsd:schema>
```

20 Ejb-jar 文件

Ejb-jar 文件是打包企业 bean 的标准格式。Ejb-jar 文件用于打包未组装的企业 bean（bean 提供者的输出），以及打包组装好的应用（应用组装者的输出）。

20.1 概述

Ejb-jar 格式是 bean 提供者和应用组装者之间的协议，以及应用组装者和部署人员间的协议。

由 bean 提供者生产的 ejb-jar 包含一到多个企业 bean，通常不包含应用组装指南。由应用组装者（他可能和 bean 提供者是同一个人或同一个组织）输出的 ejb-jar 保护一到多个企业 bean，以及应用组装信息来描述企业 bean 如何被绑定到一个单一的应用部署单元。

20.2 部署文件

Ejb-jar 文件必须以 19 章定义的格式包含部署描述（如果可能）。部署描述文件必须以 META-INF/ejb-jar.xml 的名字放在 ejb-jar 文件中。

20.3 Ejb-jar 文件的要求

Ejb-jar 文件必须通过包括或通过引用包含每个企业 bean 的下述类文件：

- 企业 bean 类
- 企业 bean 的业务接口，web 服务的终端接口，以及 home 和组件接口。
- 拦截器类
- 如果 bean 是实体类，则还有主键类。

如果从属文件指定在引用方 jar 文件的 Manifest 文件的 Class-Path 属性中或被包含（通过包括或引用）在另一个在引用方的 Manifest 文件的 Class-Path 属性中指定的 jar 文件中（译者注：一个 jar 直接引用或间接引用另一个 jar）,我们说 jar 文件通过"引用"包含了一个从属文件。

Ejb-jar 文件也必须通过包括或引用包含所有的企业 bean 类和 home 接口，组件接口和/或依赖的 web 服务终端，但不用包含 Java EE 和 J2SE 类。这些类包括超类和超接口，依赖的类，和用作方法参数、结果和异常的类和接口。

应用组装人员不必将 EJBHome 和 EJBObject 的 stub 打包金 ejb-jar 文件。这包括那些企业 bean 的实现由 ejb-jar 提供的企业 bean 的 stub，以及被引用的企业 bean 的 stub。生成 stub 是容器的责任。Stub 通常由容器提供者的部署工具为每个继承了 EJBHome 或 EJBObject 接口生成 stub，或者可以在运行时由容器生成。

20.4 客户端视图和 ejb-client JAR 文件

企业 bean 的客户端视图由业务接口或 home 和被引用的企业 bean 的组件接口和这些接口依赖的其他类（例如，它们的超类和超接口，依赖的类，和用作方法参数、结果和异常的类和接口）组成。可序列化的应用值类，包括可以在调用企业 bean 的远程方法中用作集合成员的类，都是客户端视图的一部分。应用值类的例子可能就是用作方法调用参数的 Address 类。

Ejb-jar 文件制作者可以为 ejb-jar 文件创建一个 ejb-client JAR 文件。ejb-client JAR 文件包含客户端程序需要用到的包含在 ejb-jar 文件中的企业 bean 的所有客户端视图的类文件。如果使用这种方式，应用组装者负责将组成客户端必需的类

包括企业 bean 的客户端视图制作成 ejb-client JAR 文件中。

ejb-client JAR 文件用可选的 ejb-client-jar 元素指定在 ejb-jar 文件中的部署文件中。ejb-client-jar 元素的值是 ejb-client JAR 文件的路径名，ejb-client JAR 文件包含在 Java EE 企业应用存档文件（.ear）。路径名是相对于引用方 ejb-jar 文件的位置。

EJB 规范没有置顶 ejb-jar 文件是否应当通过复制还是引用来包含位于 ejb-client JAR 文件中的类，但它们必须是使用其中一种来包含。如果使用拷贝方式，生产者简单地包含将 ejb-client JAR 中的类也包含在 ejb-jar 文件中。如果使用引用方式，ejb-jar 文件的生产者不复制 ejb-client JAR 文件的内容到 ejb-jar 文件中，而是使用 ejb-jar 文件的 Manifest Class-Path 条目来指定该 ejb-jar 文件运行时依赖的 ejb-client JAR。在 JAR 文件中使用 Class-Path 条目在 Java EE 平台企业版规范中解释【12】。

20.5 对客户端的要求

应用组装者必须构造应用以保证在运行时客户端可以获得客户端视图类。企业 bean 的客户端可以是另一个位于同一个 ejb-jar 或不同 ejb-jar 文件中的企业 bean，或者客户端可以是其他的 Java EE 组件，例如 web 组件。

当打包在 jar 文件的客户端引用企业 bean 时，包含客户端 jar 文件（例如一个 ejb-jar 文件）应当通过包括或通过引用包含被引用的企业 bean 的所有客户端视图类。这些客户端视图类可以被打包进 ejb-client JAR 文件。换句话说，包含客户端的 jar 文件应当包含下面内容的一个：

- 一个对 ejb-client JAR 文件的引用。
- 一个对保护客户端视图类的 ejb-jar 文件的引用。
- 客户端视图类的拷贝。

客户端也需要使用系统值类（例如，实现了 javax.ejb.Handle、javax.ejb.HomeHandle、javax.ejb.EJBMetaData、java.util.Enumeration、java.util.Collection 和 java.util.Iterator 接口的可序列化的值类），尽管这些类没有和应用打包在一起。，提供系统值类并在部署客户端时可以使用它们是容纳被引

用 bean 的容器的责任。

20.6 例子

在这个例子中，bean 提供者选择将企业 bean 客户端视图单独打包，而且引用包含它需要的类的 jar 文件。和 ejb1.jar 打包在同一个应用的 ejb2.jar 需要这些类，打包在不同应用的 ejb3.jar 也需要这些类。这些类 ejb1.jar 自己也需要，因为它们定义了 ejb1.jar 中的企业 bean 的远程接口，而且 bean 提供者选择了通过引用方式来获取这些类。

Ejb1.jar 的部署文件在 ejb-client-jar 元素中指定了客户端视图 jar 文件。因为 ejb2.jar 需要这些客户的视图类，它包含了一个到 ejb1_client.jar 引用的 Class-Path。

Class-Path 机制必须被位于 app2.ear 的组件用于引用客户端视图 jar 文件，这个 jar 文件对应于打包在 app1.ear 的 ejb1.jar 中的企业 bean。这些企业 bean 被位于 ejb3.jar 中的企业 bean 引用。注意，客户端视图 jar 文件必须被直接包含在 app2.ear 文件中。

app1.ear:

META-INF/application.xml

ejb1.jar Class-Path: ejb1_client.jar

deployment descriptor contains:

<ejb-client-jar>ejb1_client.jar</ejb-client-jar>

ejb1_client.jar

ejb2.jar Class-Path: ejb1_client.jar

app2.ear:

META-INF/application.xml

ejb1_client.jar

ejb3.jar Class-Path: ejb1_client.jar

21 运行环境

本章定义了运行时 EJB3.0 容器使得企业 bean 实例可以被获得的应用程序接口（API）。这些 API 能够被可移植企业 bean 使用，因为在所有的 EJB3.0 容器中都可以获得这些 API。

本章也定义了对 EJB3.0 容器提供者能够提供给企业 bean 的功能的约束。这些约束对强制安全和允许容器正确地管理运行时环境是必需的。

21.1 Bean 提供者的责任

本章描述 Bean 提供者的视图和责任。

21.1.1 由容器提供的 API

EJB 提供者可以依赖 EJB3.0 容器提供者来提供使用 J2SE V5.0 API 的组件，正如 Java EE5.0 规范要求的一样【12】。Java SE 平台包括下列的企业 API：

- JDBC
- RMI-IIOP
- JNDI
- JAXP
- Java IDL
- JAAS

Java EE 平台也要求许多可选的包。下列可选的包在 EJB 容器中也是需要的：

- EJB3.0，包括 Java 持久化 API
- JTA1.1
- JMS1.1
- JavaMail1.4（只用于发送邮件）
- JAF1.1（注：参见【12】了解限制条件）
- JAXR1.0
- SAAJ1.3

- JAX-RPC1.1
- JAX-WS2.0
- Connector1.5
- Web Services 1.2
- JAXB2.0
- Java EE Management1.1
- JACC1.1
- Web Services Metadata 2.0
- Common Annotations 1.0
- StAX 1.0

21.1.2 编程约束

本节描述 Bean 提供者必须遵循的编程约束，以保证企业 bean 是可移植的且能被部署到任何 EJB3.0 容器中。这些约束应用于业务方法的实现上。描述这些约束的容器视图的 21.2 节定义了所有 EJB 容器必须提供的编程环境。

- 企业 bean 不能使用读/写静态字段。可以使用只读静态字段。因此，建议在企业 bean 中的静态字段都声明为 final 的。

这个规则是为了保证一致的运行语义，因为当某些 EJB 容器可以使用单个 JVM 来执行所有的企业 bean 实例，另外一些可以将企业 bean 实例分布在多个 JVM 中执行。

- 企业 bean 不能使用线程同步原语来同步多个实例的执行。

原因同上。如果 EJB 容器将企业 bean 实例分布在多个 JVM 中，那么同步将不起作用。

- 企业 bean 不能使用 AWT 功能来显式输出信息，或者从键盘输入信息。

大多数服务器不允许和应用程序和服务器系统的键盘/显示器直接交互。

- 企业 bean 不能使用 java.io 包来获取文件系统中的文件或目录。

文件系统 API 对业务组件获取数据不是很合适。因为组件应当使用资源管理器 API 来存储数据，例如 JDBC。

- 企业 bean 不能监听 socket，接收 socket 连接或使用 socket 做多路广播。

EJB 架构允许企业 bean 实例作为网络 socket 客户端，但不允许作为网络服务器。允许实例作为网络服务器与企业 bean 的基本功能冲突——为 EJB 客户端提供服务。

- 企业 bean 不能反射一个类来获取它声明的成员，因为 Java 语言的安全规则要求企业 bean 是不可以获取这些成员的。企业 bean 不能使用反射 API 来获取 Java 语言的安全规则要求不能获取的信息。

如果允许企业 bean 获取其他类的信息和用 Java 语言一般不允许的方式获取类可能引起安全问题。

- 企业 bean 不能创建类加载器；获取当前类的类加载器；设置上下文类加载器；设置安全管理器；创建新的安全管理器；停止 JVM；或改变输入、输出和错误流。

这些功能为 EJB 容器保留。允许企业 bean 使用这些功能可能引起安全问题和降低容器正确管理运行环境的能力。

- 企业 bean 不能管理线程。企业 bean 不能启动、停止、挂起或唤醒一个线程，或者改变线程的优先级或名字。企业 bean 不能管理线程组。

这些功能为 EJB 容器保留。允许企业 bean 使用这些功能会降低容器正确管理运行环境的能力。

- 企业 bean 不能直接读写文件描述。

允许企业 bean 直接读写文件描述可能危害安全。

- 企业 bean 不能得到特定代码源的安全策略信息。

允许企业 bean 获取安全策略信息将会产生安全漏洞。

- 企业 bean 不能加载本地库。

这些功能为 EJB 容器保留。允许企业 bean 加载本地库将产生安全漏洞。

- 企业 bean 不能获取 java 语言正常规则不让企业 bean 获取的包和类。

这些功能为 EJB 容器保留。允许企业 bean 使用这个功能将产生安全漏洞。

- 企业 bean 不能在包内定义类。

这些功能为 EJB 容器保留。允许企业 bean 使用这个功能将产生安全漏洞。

- 企业 bean 不能获取或更改安全配置对象（策略，安全，提供商，签名和标识）。

这些功能为 EJB 容器保留。允许企业 bean 使用这个功能将危害安全。

- 企业 bean 不能使用子类和 java 序列化协议的对象替代特性。

允许企业 bean 使用这些功能可能危害安全。

- 企业 bean 不能将 this 作为方法的参数或返回值。替代地，企业 bean 必须传递 SessionContext.getBusinessObject，SessionContext.getEJBObject，SessionContext.getEJBLocalObject ， EntityContext.getEJBObject 或 EntityContext.getEJBLocalObject 的结果。

为了保证企业 bean 实现在所有 EJB3.0 容器间的可移植性，Bean 提供者应用使用表 24 中定义的安全设置的容器来测试企业 bean。这个表定义了兼容 EJB 容器必须在运行时为企业 bean 实例提供的最小功能。

21.2 容器提供者的责任

本节定义了为企业 bean 实例提供运行时环境的容器的责任。这里描述的要求可以看作是最小要求；容器可以选择提供其他 EJB 规范没有要求的功能。

EJB3.0 容器必须让企业 bean 实例在运行时可以获得下列 API：

- Java 2 平台，标准版 V5（J2SE）API，它包含以下 API:
  - JDBC
  - RMI-IIOP
  - JNDI
  - JAXP
  - Java IDL
- EJB3.0 API，包括 Java 持久化 API
- JTA 1.1，只有 UserTransaction 接口
- JMS 1.1
- JavaMail 1.4，只用于发送邮件
- JAF 1.1（注：参见【12】了解约束条件）

- JAXP1.2
- JAXR1.0
- JAX-RPC1.1
- JAX-WS2.0
- JAXB2.0
- SAAJ1.3
- Connector1.5
- Web Services 1.2
- Web Services Metadata 2.0
- Common Annotations 1.0
- StAX 1.0

下面子章节对这些需求进行详细描述。

21.2.1 Java 2 API 要求

容器必须提供 Java 2 平台标准版本 V5（J2SE）的所有 API。容器不允许只提供一部分。

EJB 容器可以通过 Java 2 平台的安全策略机制来限制企业 bean 实例不能使用 Java 2 平台的某些功能。主要原因是为了保护 EJB 容器环境的安全和完整性，防止企业 bean 实例干扰容器的功能。

下面的表格定义了 EJB 容器必须在运行时授予企业 bean 实例的 Java 2 平台安全许可。术语“授予”意思是容器必须能够授权许可，属于“拒绝”意思是容器应当拒绝许可。

表 24 用于标准 EJB 容器的 Java 2 平台安全策略

许可名称	EJB 容器策略
Java.security.AllPermission	deny
java.awt.AWTPermission	deny
java.io.FilePermission	deny
java.net.NetPermission	deny

java.util.PropertyPermission	grant“read”，“*”，deny 其他的
java.lang.reflect.ReflectPermission	deny
java.lang.RuntimePermission	grant“queuePringJob”，deny 其他的
java.lang.SecurityPermission	deny
java.io.SerializablePermission	deny
java.net.SocketPermission	grant“connect”，“*”【注意 A】， deny 其他的

注意：

【A】：这个许可是必须的，例如，为了允许企业 bean 可以使用 Java IDL 和 RMI-IIOP 的客户端功能。

某些容器可以允许开发者为企业 bean 授予比表 24 更多或更少的许可。对这个功能的支持不是 EJB 规范要求的。依赖更多或更少许可的企业 bean 将不是可移植的。

21.2.2 EJB3.0 的要求

容器必须实现定义在这个规范中的 EJB3.0 接口。

容器必须实现本规范定义的由 EJB3.0 支持的元数据注释的语义。

容器必须实现（或通过第三方实现提供）javax.persistence 接口和在文档“Java Persistence API”【2】中定义的元数据注释。

21.2.3 JNDI 的要求

最低要求，EJB 容器必须为企业 bean 实例提供 JNDI API 命名空间。EJB 容器必须让实例在调用 javax.naming.InitialContext 的缺省无参构造器时可以获得命名空间。

EJB 容器至少可以能够得到以下在命名空间里的对象：

- 其他企业 bean 的业务接口。
- 有企业 bean 事业的资源工厂。

- 由企业 bean 使用的实体管理器和实体管理器工厂。
- 由企业 bean 使用的 web 服务接口。
- 其他企业 bean 的 home 接口。
- ORB 对象
- UserTransaction 对象
- EJBContext 对象
- TimerService 对象

EJB 规范不要求部署在容器内的所有企业 bean 都用同一个 JNDI API 命名空间呈现。到那时，同一个企业 bean 的所有实例必须用同一个 JNDI API 命名空间呈现。

21.2.4 JTA1.1 的要求

EJB 容器必须包括 JTA1.1 扩展，并且它必须通过 javax.ejb.EJBContext 接口和在 JNDI 中以 java:comp/UserTransaction 作为名字为使用 bean 管理事务分割的企业 bean 提供 javax.transaction.UserTransaction 接口，这是 EJB 规范要求的。

其他的 JTA 接口就是底层事务管理器和资源管理器集成接口，不打算由企业 bean 直接使用。

21.2.5 JDBC 扩展的要求

EJB 容器必须包含 JDBC3.0 扩展，并为企业 bean 提供它的功能，除了底层的 XA 和连接池接口。这些底层接口打算用于 JDBC 驱动和应用服务器的集成，而不是由企业 bean 直接使用。

21.2.6 JMS1.1 的要求

EJB 容器必须包含 JMS1.1 扩展，并为企业 bean 提供它的功能，除了底层用于 JMS 提供商和应用服务器集成的接口，那些接口不是由企业 bean 直接使用。这些接口包括：javax.jms.ServerSession，javax.jms.ServerSessionPool，

javax.jms.ConnectionConsumer 和所有的 javax.jms XA 接口。
另外，下面的方法也只是有容器使用。企业bean不应当调用这些方法：
javax.jms.Session.setMessageListener，
javax.jms.Session.getMessageListener，
javax.jms.Session.run，
javax.jms.QueueConnection.createConnectionConsumer，
javax.jms.TopicConnection.createConnectionConsumer，
javax.jms.TopicConnection.createDurableConnectionConsumer，
javax.jms.Connection.createConnectionConsumer，
javax.jms.Connection.createDurableConnectionConsumer。
企业bean不能调用下面的方法，因为它们可能会影响由容器的连接管理：
javax.jms.Connection.setExceptionListener，
javax.jms.Connection.stop，
javax.jms.Connection.setClientID。
企业bean不能调用javax.jms.MessageConsumer.setMessageListener或
javax.jms.MessageConsumer.getMessageListener方法。
如果企业bean调用了本节列出的方法，规范推荐但不要求容器抛出
javax.jms.JMSException。

21.2.7 参数传递语义

企业 bean 的远程业务接口和/或远程 home 和 remote 接口都是远程接口。容器必须保证参数传递的语义和 Java RMI-IIOP 一直。非远程对象必须按值传递。

明确地说，当调用方 EJB 和被调用 EJB 位于同一个 JVM 时，EJB 容器不允许在 EJB 之间按引用传递非远程对象。如果这样做，则可能引起多个 bean 共享一个 java 对象的状态，这将会破坏企业 bean 的语义。任何远程接口调用的本地优化必须确保参数传递的语义和 Java RMI-IIOP 一致。

企业 bean 的本地业务接口和/或本地 home 和 local 接口都似乎本地 java 接口。使用这些本地接口的调用者和被调用者企业 bean 通常都位于同一个 JVM。EJB 容器必须确保跨接口的参数传递的语义和 java 语言的标准的参数传递语义一致。

21.2.8 其他要求

包含在 EJB 接口的 Javadoc 规范断言（assertion）和 Java 持久化 API 接口都是要求的功能，容器必须实现。

22 EJB 角色的责任

22.1 Bean 提供者的责任

本节重点描述对 Bean 提供者的要求。满足这些要求才能保证 Bean 提供者开发的企业 bean 能被部署到所有兼容的 EJB 容器中。

22.1.1 API 要求

企业 bean 必须满足所有定义在本文档各个章节的 API 要求。

22.1.2 打包要求

Bean 提供者有责任将企业 bean 以第 20 章描述的格式打包成一个 ejb-jar 文件。

如果使用部署文件，则必须遵循第 19 章的要求。

22.2 应用组装者的责任

应用组装者的责任定义在第 19 和 20 章。

22.3 EJB 容器提供者的责任

EJB 容器提供者负责提供部署人员使用的部署工具。部署工具的要求定义在本文档的各个章节。

EJB 容器提供者有责任实现部分 EJB 协议和在文档“java persistence API”【2】中描述的部分协议，以及在运行时提供本文档各个章节描述的服务。

22.4 持久化提供者的责任

持久化提供者有责任实现在文档“java persistence API”【2】中描述的部分协议。

22.5 部署人员的责任

部署者使用 EJB 容器提供者提供的部署工具来部署由 Bean 提供者和应用组装者生产的 ejb-jar 文件。

本文档的各章都有对部属人员的责任的详细描述。

22.6 系统管理员的责任

系统管理员负责配置 EJB 容器和服务器，设置安全管理，将资源管理器和 EJB 容器集成，以及对部署的企业 bean 应用进行运行时监控。

本文档的各章都有对系统管理员的责任的详细描述。

22.7 客户端程序员的责任

EJB 客户端程序员书写通过业务接口、web 服务客户端视图或消息来获取企业 bean 的应用，或查看它们的 home 和组件接口。

23 相关文档

[1] EnterpriseJavaBeans, version 3.0. EJB 3.0 Simplified API. *http://java.sun.com/products/ejb*.

[2] EnterpriseJavaBeans, version 3.0. Java Persistence API. *http://java.sun.com/products/ejb*.

[3] EnterpriseJavaBeans, version 2. (EJB 2.1). *http://java.sun.com/products/ejb*.

[4] JavaBeans. *http://java.sun.com/beans*.

[5] Java Naming and Directory Interface (JNDI). *http://java.sun.com/products/jndi*.

[6] Java Remote Method Invocation (RMI). *http://java.sun.com/products/rmi.*

[7] Java Security. *http://java.sun.com/security.*

[8] Java Transaction API (JTA). *http://java.sun.com/products/jta*.

[9] Java Transaction Service (JTS). *http://java.sun.com/products/jts.*

[10] Java Language to IDL Mapping Specification. *http://www.omg.org/cgi-bin/doc?ptc/00-01-06*.

[11] CORBA Object Transaction Service v1.2. *http://www.omg.org/cgi-bin/doc?ptc/2000-11-07*.

[12] Java Platform, Enterprise Edition (Java EE), v5. *http://jcp.org/en/jsr/detail?id=244*.

[13] Java Message Service (JMS), v 1.1. *http://java.sun.com/products/jms*.

[14] Java API for XML Messaging (JAXM).

[15] Java 2 Enterprise Edition Connector Architecture, v1.5 . *http://java.sun.com/j2ee/connector*.

[16] Enterprise JavaBeans to CORBA Mapping v1.1. *http://java.sun.com/products/ejb/docs.html*.

[17] CORBA 2.3.1 Specification. *http://www.omg.org/cgi-bin/doc?formal/99-10-07*.

[18] CORBA COSNaming Service. *http://www.omg.org/cgi-bin/doc?formal/00-06-19*.

[19] Interoperable Name Service FTF document. *http://www.omg.org/cgi-bin/doc?ptc/00-08-07*.

[20] RFC 2246: The TLS Protocol. *ftp://ftp.isi.edu/in-notes/rfc2246.txt.*

[21] RFC 2712: Addition of Kerberos Cipher Suites to Transport Layer Security. *ftp://ftp.isi.edu/in-notes/rfc2712.txt.*

[22] The SSL Protocol Version 3.0. *http://home.netscape.com/eng/ssl3/draft302.txt.*

[23] Common Secure Interoperability Version 2 Final Available Specification. *http://www.omg.org/cgi-bin/doc?ptc/2001-06-17.*

[24] Database Language SQL. ANSI X3.135-1992 or ISO/IEC 9075:1992.

[25] Java API for XML-based RPC (JAX-RPC) 2.0. *http://jcp.org/en/jsr/detail?id=101*.

[26] Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl.*

[27] W3C: SOAP 1.1. *http://www.w3.org/TR/SOAP/.*

[28] The Java Virtual Machine Specification.

[29] JDBC 3.0 Specification. *http://java.sun.com/products/jdbc*.

[30] Web Services Metadata for the Java Platform. *http://jcp.org/en/jsr/detail?id=181.*

[31] Web Services for Java EE, Version 1.2. *http://jcp.org/en/jsr/detail?id=109.*

[32] Java API for XML Web Services (JAX-WS 2.0). *http://jcp.org/en/jsr/detail?id=224.*