第1章 引 言

今天的企业需要扩大自己的影响，缩减自己的成本，并且提高自己与客户，雇员，供货商之间沟通的效率。

通常，提供这类服务的应用程序必须结合现存的企业信息系统(EIS)，具备新的业务功能为更广泛的用户群提供服务。这些服务要求具有:

高可用性，以满足今天商业全球化的需求

安全性，以保护用户的隐私和维护企业的安全

可靠性和可伸缩性，以确保业务被准确及时地处理

大多数情况下，企业服务由多层应用程序实现。中间层整合了现存的EIS，具备新服务的业务功能和数据。成熟的Web技术用来提供用户层服务，简化业务访问的复杂性，并且消除或大大减少对用户的管理和培训。

Java™平台企业版(Java™ EE) 降低了开发多层次企业级服务的成本和复杂性。Java EE 应用程序可以快速地部署和强化，使企业轻松地应对竞争压力。

Java EE方案可以实现上述目标，这需要定义一个标准的架构，以下是其组成元素:

Java EE平台 - 一个托管Java EE应用程序的标准平台。

Java EE兼容性测试套件 - 兼容性测试套件用于检验Java EE平台产品是否符合Java EE平台标准。

Java EE可参考的实现- 一个可参考的实现是一个Java EE 应用程序原型，提供一套可行的Java EE平台定义。

Java EE蓝图 - 一套开发多层次瘦客户端服务的最佳实践。

本文档描述了Java EE平台规范。它定义了一个Java EE平台产品必须达到的标准。

1.1 感谢

本规范是多人协作的成果。Vlada Matena撰写了第一个草案以及事务管理和命名的章节。 Sekhar Vajjhala, Kevin Osborn,和Ron Monzillo撰写了安全的章节。Hans Hrasna撰写了应用程序组装和部署的章节。Seth White撰写了JDBC API标准。Jim Inscore, Eric Jendrock,和Beth Stearns提供了编辑上的帮助。Shel Finkelstein, Mark Hapner, Danny Coward, Tom Kincaid,和Tony Ng对多次草案提供了反馈。当然，本规范的形成和确立来自同我们行业伙伴的会谈和反馈。

1.2 版本1.3的感谢

本规范1.3版的成长离不开与1.2版合作伙伴的探讨，以及那些随后在1.2最终发布版中加入的合作伙伴。版本1.3在Java Community Process(JCP)下创建，标号为JSR-058。JSR-058专家组中的代表来自下面的公司和组织: Allaire, BEA Systems, Bluestone Software, Borland, Bull S.A., Exoffice, Fujitsu Limited, GemStone Systems, Inc., IBM, Inline Software, IONA Technologies, iPlanet, jGuru.com, Orion Application Server, Persistence, POET Software, SilverStream, Sun, and Sybase。另外，很多帮助过先前版本的朋友继续在为当前版本提供帮助，比如Jon Ellis和Ram Jeyaraman。Alfred Towell为这个版本提供了编辑上重要支持。

1.3 版本1.4的感谢

本规范1.4版创建在Java Community Process(JCP)下，标号为JSR-151。JSR-151专家组包含了下面的成员: Larry W. Allen (SilverStream Software), Karl Avedal (Individual), Charlton Barreto (Borland Software Corporation), Edward Cobb (BEA), Alan Davies (SeeBeyond Technology Corporation), Sreeram Duvvuru (iPlanet), B.J. Fesq (Individual), Mark Field (Macromedia), Mark Hapner (Sun Microsystems, Inc.), Pierce Hickey (IONA), Hemant Khandelwal (Pramati Technologies), Jim Knutson (IBM), Elika S. Kohen (Individual), Ramesh Loganathan (Pramati Technologies), Jasen Minton (Oracle Corporation), Jeff Mischkinsky (Oracle Corporation), Richard Monson-Haefel (Individual), Sean Neville (Macromedia), Bill Shannon (Sun Microsystems, Inc.), Simon Tuffs (Lutris Technologies), Jeffrey Wang (Persistence Software, Inc.)和Ingo Zenz (SAP AG)。我在Sun公司的同事提供了宝贵的援助: Umit Yalcinalp将部署描述符转向了XML Schema; Tony Ng和Sanjeev Krishnan为事务标准提供了帮助; Jonathan Bruce为JDBC标准提供了帮助; Suzette Pelouch, Eric Jendrock和Ian Evans 提供了编辑上的支持。也感谢所有外部的评论，包括Jeff Estefan (Adecco Technical Services)。

1.4 版本5的感谢

本规范的版本5创建在 (原来被称为版本1.5) Java Commuinity Process(JCP)下，标号为JSR-244。JSR-244专家组包含了下面的成员: Kilinc Alkan (Individual), Rama Murthy Amar Pratap (Individual), Charlton Barreto (Individual), Michael Bechauf (SAP AG), Florent Benoit (INRIA), Bill Burke (JBoss, Inc.), Muralidharan Chandrasekaran (Individual), Yongmin Chen (Novell, Inc.), Jun Ho Cho (TmaxSoft), Ed Cobb (BEA), Ugo Corda (SeeBeyond Technology Corporation), Scott Crawford (Individual), Arulazi Dhesiaseelan (Hewlett-Packard Company), Bill Dudney (Individual), Francois Exertier (INRIA), Jeff Genender (The Apache Software Foundation), Evan Ireland (Sybase, Inc.), Vishy Kasar (Borland Software Corporation), Michael Keith (Orcale Corporation), Wonseok Kim (TmaxSoft, Inc.), Jim Knutson (IBM), Elika Kohen (Individual), Felipe Leme (Individual), Geir Magnusson Jr. (The Apache Software Foundation), Scott Marlow (Novell, Inc.), Jasen Minton (Oracle Corporation), Jishnu Mitra (Borland Software Corp), David Morandi (E.piphany), Nathan Pahucki (Novell, Inc.), David Morandi (E.piphany, Inc.), Ricardo Morin (Intel Corporation), Nathan Pahucki (Novell, Inc.), Matt Raible (Individual), Dirk Reinshagen (Individual), Narinder Sahota (Cap Gemini), Suneet Shah (Individual), Bill Shannon (Sun Microsystems, Inc.), Rajiv Shivane (Pramati Technologies), Scott Stark (Jboss, Inc), Hani Suleiman (Ironflare AB), Kresten Krab Thorup (Trifork), Ashish Kumar Tiwari (Individual), Sivasundaram Umapathy (Individual), Steve Weston (Cap Gemini), Seth White (BEA Systems)和Umit Yalcinalp (SAP AG)。再次, 我在Sun公司的同事提供了宝贵的援助: Roberto Chinnici为依赖注入的很多相关问题提供了对草案的建议。

1.5 版本6的感谢

本规范的版本6创建在Java Commuinity Process(JCP)下，标号为JSR-316。引领此规范JSR-316专家组的是Bill Shannon (Sun Microsystems, Inc.)和Roberto Chinnici (Sun Microsystems, Inc.)。此专家组包含了下面的成员: Florent Benoit (Inria), Adam Bien (Individual), David Blevins

(Individual), Bill Burke (Red Hat Middleware LLC), Larry Cable (BEA Systems), Bongjae Chang (Tmax Soft, Inc.), Rejeev Divakaran (Individual), Francois Exertier (Inria), Jeff Genender (Individual), Antonio Goncalves (Individual), Jason Greene (Red Hat Middleware LLC), Gang Huang (Peking University), Rod Johnson (SpringSource), Werner Keil (Individual), Michael Keith (Oracle), Wonseok Kim (Tmax Soft, Inc.), Jim Knutson (IBM), Elika S. Kohen (Individual), Peter Kristiansson (Ericsson AB), Changshin Lee (NCsoft Corporation), Felipe Leme (Individual), Ming Li (TongTech Ltd.), Vladimir Pavlov (SAP AG), Dhanji R. Prasanna (Google), Reza Rahman (Individual), Rajiv Shivane (Pramati Technologies), Hani Suleiman (Individual)。

第二章 平台概述

本章是Java™平台企业版(Java EE™)的概述。

2.1 体系结构

图2-1展示了Java EE平台体系结构中各元素间的既定关系。注意，此图展示的是元素间的逻辑关系，它并不代表这些元素在物理上的划分方式(不同的机器，进程，地址空间或虚拟机)。

每个独立矩形上半部分标明的容器是Java EE运行时环境，它为应用程序组件提供了必要的服务。这些服务基于矩形下半部分所列出的技术。例如，“Application Client Container”(应用程序客户端容器) 为应用程序客户端提供了链接JMS的接口，其它服务也是如此。所有这些服务会在后续章节详细描述。

这些箭头表示，必须提供对Java EE平台对应部分的访问。应用程序客户端容器为应用程序客户端提供了直接访问数据库的环境。这是通过连接数据库系统的Java API(JDBC™ API)来实现。类似地，Web容器为JSP页面和Servlet，EJB容器为企业Bean也提供了对数据库的访问.

正如图中所示，Java™平台标准版(Java SE)API由Java SE运行时环境提供，各种类型应用程序组件都基于这些API。

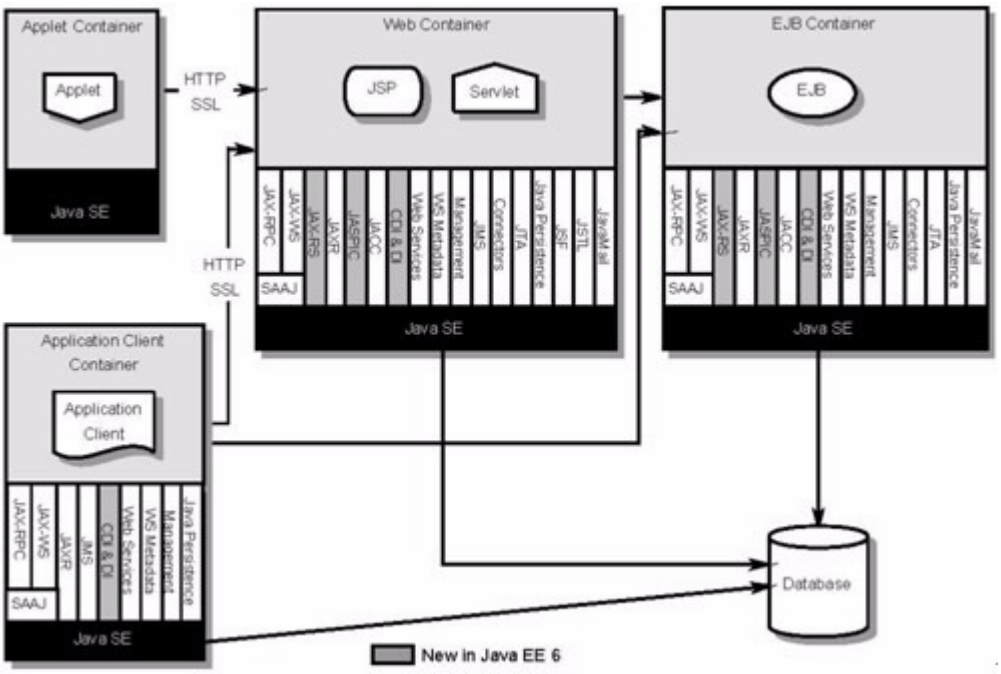

图 2-1 Java EE 体系结构图

下面的章节描述了每种Java EE平台元素的标准。

2.2 Profile(自定义规范)

Java EE 6规范提出了”Profile (自定义规范)”的概念。

Profile描绘了Java EE平台针对应用程序特定类的配置。

Profiles不是一个新的概念，也不是Java EE platform特有的。“Java Community Process Document 2.6)”(Java社区进程文档2.6)对Profile给出了如下定义: “一种规范，它参考了某个平台版本的规范，并附加零个或多个其它的JCP规范(它们尚未成为平台版本规范的组成部分)。它必须包含来自参考平台的API， 但必须符合那个平台定义的引用规则。其它引进的规范必须在完全符合规则的前提下被引进。”

所有Java EE Profile共享一套公共特征，例如命名和资源注入，打包规则，安全标准等等。这就保证了“Java EE平台”旗下的所有产品的一致性。这也确保了熟悉某一Profile或整个平台的开发者可以很容易地适应其它的Profile，避免了技能和经验上的过 度差异。

相对于上面概括的基本功能，Profile可以自由包含这个平台的任何技术，它的平台规范中提供了适用于这些技术的所有规则，可以单独使用也可以组合使用。

最后要强调的是，如果Profile只包含逐点技术，它们将比哪些有很少或没有接入点的API包多一点。作为代替，这里采用的Profile的定义应保证，每当本规范定义了技术组合的标准，这些标准会在所有基于Java EE Profile的产品中兑现。

举一个具体的例子, 让我们思考一下在Servlet容器中事务的使用。在孤立状态下，Servlet规范和JTA规范都没有为可移植的应用程序定义一个完整的编程模型。通过 开拓自己的一套标准来结合Servlet和JTA，这种规范就可以弥补这一缺陷。所有基于此Profile并包含了这两种技术的Java EE产品都必须满足这些条件, 这就给应用程序开发者提供了一个更完整的编程模型，并为所有相关的Java EE Profile共享。

一个独立的规范--Java EE 6 Web Profile规范，定义了Java EE Web Profile，它是Java EE 6平台的第一项Profile。

可以依据JCP制定的规则和当前规范包含的规则来定义附加的Profile。特别是，一项Profile的确立必须提交一项Java Specification Request(JSR)，并在完成时公布自己的计划，它应无视平台自身或其它Profile在此时发生的任何修订。这确保了在定义和发布一项新的 Profile或更新已有版本时拥有最大灵活性。

根据上面给出的Profile定义，一项Profile最终可以成为这个平台适当的子集或者超集，也可以与它在某一范围上部分重叠。这种灵活性保证了未来的Profile所覆盖的功能将远远超过这个平台规范最初的设想。

正如前面的段落所描述的，建立新的Profile是一项意义深远的任务。决定创建一项Profile时应该重视它潜在的不利因素，尤其是条件的分裂 和开发者的困惑。一般而言，要建立一项Profile，需要开发者自然地支持，并且充分衡量应用程序能否从中获益。也推荐Profile在它自己的兴趣范 围中扩展功能，尽量避免相互重叠或竞争的Profile出现。Profile可以使用Java EE 6 的平台特征，诸如可选的组件，可扩展性和修剪流程，以更好地实现他们预定的目标。

2.3 应用程序组件

Java EE运行时环境定义了Java EE产品必须支持的4种应用程序组件类型:

应用程序客户端是Java编程语言程序，它通常是执行在个人电脑上的图形用户界面(GUI)程序。应用程序客户端提供一种类似于本地应用程序的用户体验，并且能够访问Java EE中间层的所有功能。

Applet是运行在Web浏览器中的一种特殊的GUI组件，但它也可以运行在其它的支持Applet编程模式的应用程序或设备上。Applet可 以用来为Java EE应用程序提供强有力的用户界面(简易的HTML页面也能用来为Java EE应用程序提供有限的用户界面)

Servlet, JSP页面, JSF应用程序, 过滤器和Web事件监听器通常运行在Web容器中，并且可以响应来自Web客户端的HTTP请求。Servlets, JSP 页面, JSF应用程序以及过滤器可以用来生成HTML页面，作为应用程序的用户界面。也可以用它们生成供其它应用程序组件使用的XML或其它格式的数据。一种特 殊的Servlet使用SOAP/HTTP协议为Web服务提供支持。Servlet, 由JSP技术或JSF技术生成的页面,Web过滤器和Web事件监听器在本规范中统称为“Web组件”。Web应用程序由Web组件和其它数据(如 HTML页面)组成。Web组件运行在Web容器中。Web服务器包含一个Web容器以及协议，安全等其它方面的支持，符合Java EE规范的要求。

Enterprise JavaBeans(EJB)组件运行在一个支持事务的环境中接受管理。企业Bean通常包含Java EE应用程序的业务逻辑。企业Bean可以使用SOAP/HTTP协议直接提供Web服务。

2.3.1 Java EE服务器为应用程序组件提供支持

Java EE服务器为符合标准的应用程序组件提供部署，管理和运行的支持。根据它们所以依赖的Java EE服务器，应用程序组件可以分成3类:

> 部署，管理和运行在Java EE服务器上的组件。这类组件包括Web组件和EJB组件。请查看这些组件各自的规范。
>
> 部署和管理在Java EE服务器上，但是被加载到客户机上运行的组件。这类组件包括诸如HTML页面和嵌入THML页面的Applet这样Web的资源。

部署和管理没有完全定义在本规范中的组件。应用程序客户端就属于这种类型。本规范的未来版本可能会更完整地定义应用程序客户端的部署和管理。请查看EE.10, “应用程序客户端”中对应用程序客户端的描述。

2.4 容器

容器为Java EE应用程序组件提供了运行时支持。容器提供了一份从底层Java EE API到应用程序组件的联合视图。Java EE应用程序组件不能直接地与其它Java EE应用程序组件交互。它们通过容器的协议和方法来达成它们之间以及它们与平台服务之间的交互。在应用程序组件和Java EE服务之间插入一个容器，这允许该容器透明地为组件注入必须的服务，例如声明式事务管理，安全检查，资源池和状态管理。

一个标准的Java EE产品会为每个应用程序组件类型提供一个容器: 应用程序客户端容器，Applet容器，Web组件容器, 企业Bean容器。

2.4.1 容器的标准

本规范要求容器提供一个由Java™平台标准版规范v6 (Java SE)定义的Java™兼容性运行时环境。Applet容器可以使用Java插件产品来提供这个环境，或者是使用本地环境。提供JDK™ 1.1 API的Applet容器超出了本规范的范围。

容器工具必须识别部署应用程序组件的打包文件格式。

容器由Java EE产品供应商提供。请查看2.11.1,“Java EE产品供应商”中对产品供应商角色的描述。

本规范定义了一套标准服务，每个Java EE产品必须提供支持。后面会对这些标准服务进行描述。Java EE容器提供了访问这些服务的API，供应用程序组件使用。本规范也描述了用连接器扩展Java EE服务的标准方法，以结合其它的非Java EE应用程序系统，例如大型机系统和ERP系统。

2.4.2 Java EE服务器

Java EE容器是底层服务器的组成部分。Java EE产品供应商通常使用现有的事务处理框架结合Java SE技术来实现Java EE服务器端功能。Java EE客户端功能通常构建于Java SE技术。

2.5 资源适配器

资源适配器是一个系统级的组件，它通常实现了对外部资源管理器的网络连接。资源适配器能够扩展Java EE平台的功能。这只需要实现一个Java EE标准服务API(例如JDBC™驱 动程序)，或者定义并实现一个能连接到外部应用程序系统的资源适配器就可以达到。资源适配器也可以提供完整的本地或本地资源的服务。资源适配器接口通过 Java EE服务供应商接口(Java EE SPI)来连接Java EE平台。使用Java EE SPI连接到Java EE平台的资源适配器可以和所有的Java EE产品协同工作。

2.6 数据库

Java EE平台需要数据库来存储业务数据，通过JDBC API进行访问。可以从Web组件，企业Bean和应用程序客户端组件连接到数据库。不需要从Applet连接到数据库。

2.7 Java EE标准服务

Java EE标准服务包括下述服务(后面的章节会进行更详细地描述)。一些标准服务实际上由Java SE提供。

2.7.1 HTTP

HTTP客户端API定义在java.net包中。HTTP服务器端API由Servlet, JSP和JSF接口定义，以及被那些是Java EE平台组成部分的Web服务支持。

2.7.2 HTTPS

支持HTTP协议的客户端和服务器端API也同样支持带SSL协议的HTTP协议。

2.7.3 Java™ Transaction API (JTA)

Java事务API由两部分组成:

一个应用程序级的边界划分接口，容器和应用程序组件用它来划分事务边界。

一个介于事务管理器和资源管理器之间的Java EE SPI级接口。

2.7.4 RMI-IIOP

RMI-IIOP由API组成，这些API允许使用不依赖于底层协议的RMI风格编程。作为那些API的实现，它同时支持Java SE本地RMI协议和CORBA IIOP协议。通过支持IIOP协议，Java EE应用程序就可以使用RMI-IIOP来访问CORBA服务，并且该应用程序兼容RMI编程约束(请查看RMI-IIOP的详细说明)。这样的 CORBA服务通常由Java EE产品之外的组件定义，一般存在于以前遗留下来的系统中。只要求Java EE应用程序客户端可以使用RMI-IIOP API来直接定义它们自己的CORBA服务。通常这样的CORBA对象用于在访问其它的CORBA对象时进行回调。

当访问EJB组件时，Java EE应用程序必须使用RMI-IIOP API，特别是javax.rmi.PortableRemoteObject类的narrow方法，正如EJB规范所描述的。这些企业Bean可以独立 于协议。需要注意的是，当使用依赖注入代替JNDI就行查找时，通常不需要使用narrow方法; 在注入对象引用之前，容器会为应用程序执行narrow方法。Java EE产品必须能够使用IIOP协议输出和访问企业Bean，这在EJB规范中被明确规定。对IIOP协议的支持使Java EE产品之间的交互成为可能，不过，Java EE产品也可以使用其它的协议。

2.7.5 Java IDL

通过使用IIOP协议，Java IDL允许Java EE应用程序组件调用外部的CORBA对象。这些 CORBA对象可以用任何语言编写，并且通常存在于Java EE产品外部。Java EE应用程序可以使用Java IDL来担当CORBA服务的客户端，但是只有Java EE应用程序客户端可以直接使用Java IDL提供CORBA服务。

2.7.6 JDBC™ API

JDBC API是用来连接关系数据库系统的API。JDBC API 有两个部分: 一个是应用程序组件用来访问数据库的应用级接口，另一个是JDBC驱动接口。不要求Java EE产品对JDBC驱动接口提供支持。JDBC驱动应该被打包成一个资源适配器，通过使用连接器API的能力来连接Java EE产品。JDBC API包含在了Java SE中, 但是本规范包含了对JDBC设备驱动的额外标准。

2.7.7 Java™ Persistence API(JPA)

Java持久化API是用于持久化和对象/关系映射管理的标准API。通过使用一个Java域模型来管理关系型数据库，本规范为应用程序开发者提供了一种对象/关系映射功能。Java EE必须对Java持久化API提供支持。它也可以用在Java SE环境中。

2.7.8 Java™ Message Service (JMS)

Java消息服务是用于消息发送的标准API，它支持可靠的“点对点”消息发送和“发布-订阅”模型。本规范要求JMS供应商同时实现“点对点”消息发送和”发布/订阅”型消息发送。

2.7.9 Java Naming and Directory Interface™ (JNDI)

JNDI API是用于命名和目录访问的标准API。它有两个部分：一个是应用程序组件用来访问命名和目录服务的应用程序级接口，另一个是用于连接命名和目录服务供应商的服务供应商接口。JNDI API包含在Java SE中，但是本规范为它定义了额外的标准。

2.7.10 JavaMail™

许多互联网应用程序需要发送邮件的功能，因此Java EE平台包含了JavaMail API以及相应的JavaMail服务供应商API，使应用程序组件可以发送互联网邮件。JavaMail API 有两个部分：一个是应用程序组件用于发送邮件的应用程序级接口，另一个是Java EE SPI级的服务供应商接口。

2.7.11 JavaBeans™ Activation Framework (JAF)

JAF API提供了一个框架来处理不同MIME类型的数据，它们源于不同的格式和位置。JavaMail API使用了JAF API。JAF API包含在Java SE中，因此它可以被Java EE应用程序使用。

2.7.12 XML处理

Java™ API for XML Processing (JAXP)支持工业标准化的SAX和DOM API，用以解析XML文档，也支持XSLT转换引擎。 Streaming API for XML (StAX)为XML提供了一种“拉式解析”型的API。JAXP和StAX API都包含在Java SE中，因此它们可以被Java EE应用程序使用。

2.7.13 Java EETM连接器体系结构

连接器体系结构是一种Java EE SPI，它允许将资源适配器插入到任何Java EE产品中，这个资源适配器支持对EIS的访问。连接器体系结构定义了一套标准的介于Java EE服务器和资源适配器之间的系统级协议。这些协议包括：

连接管理协议，它让Java EE服务器将“连接”集中到底层EIS中，然后让应用程序组件与EIS连接。这种方式营造出了一种可伸缩的应用程序环境，它能支持大规模客户端对EIS系统的访问。

事务管理器和EIS之间的事务管理协议，它支持对EIS资源管理器的事务型访问。这个协议让Java EE服务器使用事务管理器来管理跨多个资源管理器的事务。这个协议也支持EIS资源管理器内部管理的事务，而不需要牵连外部的事务管理器。

安全协议，它实现了对EIS的安全访问。这个协议为安全的应用程序环境提供支持，它减少了对EIS的安全威胁并保护了EIS管理的重要信息资源。

线程管理协议，它允许资源适配器将工作委托给其它的线程，并允许应用程序服务器管理线程池。通过工作线程，资源适配器可以控制安全上下文和事务上下文。

消息传递协议，它允许资源适配器将消息传递到消息驱动Bean，此Bean不依赖于特定的消息格式，消息语义和传递消息的底层结构。这个协议也充当标准消息提供方即插即用协议，它允许通过资源适配器将消息提供方插入到任何Java EE服务器中。

事务传递协议，它允许资源适配器将一个被导入的事务上下文传递给Java EE服务器，使得服务器和任意应用程序组件的交互成为被导入的事务的组成部分。这个协议维护了被导入事务的ACID(atomicity, consistency, isolation, durability)属性。

可选协议，它提供了一个介于应用程序和资源适配器之间的通用命令接口。

2.7.14 安全服务

Java™ Authentication and Authorization Service (JAAS)使服务能够基于用户进行验证和实施访问控制。它实现了一个Java版的标准的的Plugable Authentication Module (PAM)框架，并支持基于用户的授权。JavaTM Authorization Service Provider Contract for Containers (JACC) 定义了Java EE应用程序服务器和授权服务提供方之间的协议，允许将自定义的授权服务提供方插入任何Java EE产品中。

2.7.15 Web服务

Java EE为Web服务的客户端和终端提供了完整的支持。一些Java技术协同为Web服务提供支持。通过使用SOAP/HTTP协议，Java API for XML Web Services (JAX-WS)和Java API for XML-based RPC (JAX-RPC)都能为Web服务的调用提供了支持。新的JAX-WS是支持Web服务的首选API，它出现在JAX-RPC之后。JAX-WS提供了更广泛的Web服务功能，并提供对多重“绑定-协议”的支持。当使用绑定于HTTP协议的SOAP1.1协议时，JAX-WS和JAX-RPC完全 可以协同工作，但要受到WS-I基本概要规范的约束。

JAX-WS和Java Architecture for XML Binding (JAXB)定义了Java类和用于SOAP调用的XML之间的映射，并且对XML Schema提供了100%的支持。SOAP with Attachments API for Java (SAAJ) 对操作低层SOAP消息提供支持。Java EE规范中的Web服务标准完整地定义了Web服务的客户端和终端在Java EE中的部署，以及使用企业Bean的Web服务终端的实现。Web服务元数据规范定义了相应的Java语言注解来简化Web服务的开发。Java API for XML Registries (JAXR)使客户端可以访问XML注册服务器。

Java API for RESTful Web Services (JAX-RS)对使用REST风格的Web服务提供了支持。RESTful Web服务更符合Web的设计风格，并且常常更易于使用多种编程语言对其进行访问。JAX-RS提供了一个简单的高层API来写入这样的Web服务，以及 一个低层API来控制Web服务交互中的所有细节。

2.7.16 管理

Java 2平台企业版管理规范中定义了一种API，通过一种特殊的管理型企业Bean来管理Java EE服务器。Java™ Management Extensions (JMX) API也提供了一些管理上的支持。

2.7.17 部署

Java 2平台企业版部署规范中定义了部署工具和Java EE产品之间的协议。Java EE产品提供了运行在部署工具上的插入式组件，它允许部署工具将应用程序部署到Java EE产品中。

部署工具提供了插入式组件可以使用的服务。

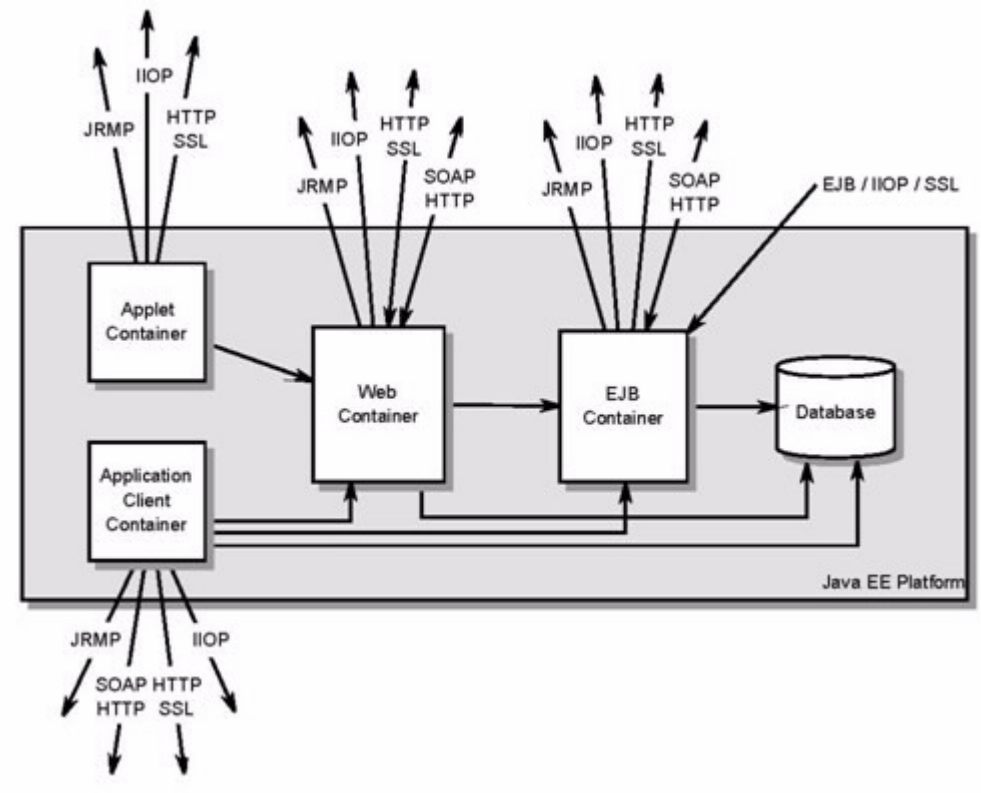

图 2-2 Java EE的交互

2.8 互用性

上面提到的很多API都提供了与非Java EE平台组件之间的互用性。例如，外部的Web服务或CORBA服务。

图 2-2 展示了Java EE平台的互通能力。（箭头的方向标示了组件间的“客户端→服务器”关系）

2.9 产品标准的灵活性

本规范没有要求Java EE产品必须由单个程序，单个服务器，甚至是单个机器实现。一般情况下，本规范没有描述服务或功能在机器，服务器或进程上的划分。只要符合本规范的要 求，Java EE产品供应商可以用他们认为最合适的方式来划分功能。Java EE产品必须能够部署应用程序组件，此组件的运行应符合本规范语义的描述

典型的低端Java EE产品会支持Applet，这需要使用主流浏览器中的Java插件，它也会支持在它们自己的Java虚拟机中的每个应用程序客户端，同时，它还会提供一 个单独的服务器来支持Web组件和企业Bean。高端Java EE产品将服务器组件分摊到多个服务器上，它们每个都可以通过服务器集群来实现分布式和负载均衡。本规范不限定或阻止它们所进行的任何配置。

符合本规范要求的Java EE产品可能有多种配置和实现。当成功部署到这些产品后，可移植的Java EE应用程序就可以正确地工作。

2.10 Java EE产品的扩展

本规范描述了可用于所有Java EE产品的功能的最小集合。Java EE Profile可以包含这些功能中的部分或全部，正如EE.9,“Profile”中所描述的。要实现完整的Java EE平台，产品必须提供这个集合中的所有功能(请查看EE.9.7,“完整的Java EE产品标准”)。大多数Java EE产品会提供比本规范的最低要求更多的功能。本规范对产品提供的扩展性做了很少量的限制。特别是，它对Java

API扩展性的限制与Java SE是相同的。Java EE 产品不可以向本规范定义的Java编程语言包中添加类，也不可以向特定类中添加方法或是用其它方式修改它们的签名。

然而，许多其它的扩展是允许的。Java EE产品可以提供额外的Java API，可以是其它的Java可选包，也可以是别的包(恰当地命名)。Java EE产品可以包含对规范外的其它协议或服务的支持。Java EE产品可以支持其它语言编写的应用程序，也可以支持与其它平台或应用程序的连接。

当然，可移植的应用程序不应使用任何对平台的扩展。使用本规范没有定义的功能的应用程序的可移植性将会降低。依赖于这种功能造成的可移植性的丧失所 带来的影响可能是很小的，但也可能是不容忽视的。文档--“用Java2平台企业版设计企业级应用程序”提供了相关的信息来帮助应用程序开发者构建可移植 的应用程序。并且也建议了怎样最好地来管理非可移植性代码的使用，当这些功能必须被使用的时候。

我们期望Java EE产品在服务品质的各个方面都有很大程度的提升，表现出活跃的竞争力。让产品能够提供不同级别的性能，可伸缩性，健壮性，可用性和安全性。在某些情况下，本规范要求了最低限度的服务水平。本规范的未来版本可能会允许应用程序描述它们在这些领域的要求。

2.11 平台角色

本节描述了典型的Java平台企业版的角色。在实际应用中, 组织机构可以划分不同的角色功能来适应自己应用程序开发和部署的工作流程。

本规范稍后会对这些角色进行更详细的描述。这些角色的相关子集在EJB, JSP和Servlet规范中有相应的描述，这些子规范也属于Java EE规范的一部分。

2.11.1 Java EE 产品供应商

Java EE产品供应商是Java EE产品的实现者和提供者，包括组件容器，Java EE平台 API和本规范中定义的其它的功能。一个Java EE产品供应商通常是一个操作系统厂商，一个数据库系统厂商，一个应用程序服务器厂商，或一个Web服务器厂商。Java EE产品供应商透过容器使Java EE API对应用程序组件可用。产品供应商总是将他们的实现建立在现有的基础设施之上。

Java EE产品供应商必须提供应用程序组件到网络协议的映射，正如本规范所指定的。Java EE产品可以自由地实现接口，在具体的实现方式上，本规范没有作出明确的限定。

Java EE 产品供应商必须提供相应的工具来部署和管理应用程序。部署工具允许部署者(参见2.11.4,“部署者”)将应用程序组件部署到Java EE产品中。管理工具允许系统管理员(参加2.11.5,“系统管理员”)管理Java EE产品和部署在其中的应用程序。本规范没有对这些工具的形式作出限定。

2.11.2 应用程序组件供应商

应用程序组件供应商有多种任务可供选择，包括HTML文档设计者，文档程序员和企业Bean开发者。这些任务需要使用工具来生产Java EE应用程序和组件。

2.11.3 应用程序组装者

应用程序组装者将应用程序组件供应商开发的一套组件组装成一个完整的Java EE应用程序，以企业归档文件(.ear)的形式交付。应用程序组装者一般会使用平台供应商或工具供应商提供的GUI工具。应用程序组装者负责提供组装操 作说明，描述应用程序的外部依赖关系，以便部署者在部署过程中进行处理。

2.11.4 部署者

部署者负责将应用程序客户端，Web应用程序和EJB组件部署到一个特定的运行环境中。部署者使用由Java EE产品供应商提供的工具来完成部署任务。部署工作通常有3个步骤:

1. 在安装期，部署者将应用程序资料移动到服务器上，生成容器特有的附加类和接口，使容器能够在运行时管理这些应用程序组件。然后将应用程序组件以及这些附加类和接口安装到相应的Java EE容器中。

2.在配置期，需要理清应用程序组件供应商声明的外部依赖关系并遵循应用程序组装者定义的操作说明。例如，部署者负责将应用程序组装者定义的安全角色映射到目标运行环境中存在的用户组和账号。

3.最后，部署者启动新安装和配置的应用程序。

在某些情况下，有特殊资格的部署者可以在部署期间自定义应用程序组件的业务逻辑。例如，使用Java EE产品提供的工具，部署者可以用简单的应用程序代码封装企业Bean的业务方法，或自定义JSP页面的外观。

部署者的产出是Web应用程序，企业Bean，Applet和应用程序客户端，它们为目标运行环境定制，并已经部署到了特定的Java EE容器中。

2.11.5 系统管理员

系统管理员负责配置和管理企业的运作和网络基础设施。系统管理员也负责观察已部署的Java EE应用程序的运行时状态。系统管理员通常使用Java EE产品供应商提供的运行时监测管理工具来完成他们的任务。

2.11.6 工具供应商

工具供应商提供开发和打包应用程序组件的工具。各种工具事先应与Java EE平台支持的应用程序组件类型一致。平台独立的工具可以用于应用程序的整个部署阶段和对应用程序服务器的管理和监测。

2.11.7 系统组件供应商

系统组件供应商可以提供多种系统级组件。连接器体系结构定义了一些基本的API，用它们来提供多种类型的资源适配器。这些资源适配器可以连接到已有的多种类型的企业信息系统，这包括数据库系统和消息系统。系统级组件的另一种类型是授权策略提供方，正如Java容器授权服务提供方协议规范所定义的。

2.12 平台协议

本节描述了Java平台企业版协议，实现了完整的Java EE平台的产品供应商必须完全支持这些协议。Java EE Profile可以包含这些功能的部分或全部，在EE.9,“Profiles”中有详细的描述。

2.12.1 Java EE API

Java EE API定义了应用程序组件和Java EE平台之间的协议。此协议同时指定了运行时和部署时的接口。

Java EE产品供应商必须以某种方式实现Java EE API，此方式应支持本规范描述的语义和策略。应用程序组件供应商提供符合这些API和策略的组件。

2.12.2 Java EE Service Provider Interfaces (SPI)

Java EE服务供应商接口(SPI)定义了Java EE平台和服务提供方之间的协议，这些服务提供方可以被插入到Java EE产品中。连接器API定义的服务供应商接口可以将资源适配器整合到Java EE应用程序服务器上。实现了连接器API的资源

适配器组件可以被连接器调用。Java EE授权API定义的服务供应商接口可以将安全授权机制整合到Java EE应用程序服务器上。

Java EE产品供应商必须以某种方式实现Java EE SPI。此方式支持本规范中描述的语义和策略。服务提供方组件的供应商(例如，连接器供应商)应该提供符合这些SPI和策略的组件。

2.12.3 网络协议

本规范定义了应用程序组件到工业标准网络协议的映射。此映射允许客户端从尚未安装Java EE产品技术的系统访问应用程序组件。关于网络协议对互用性的支持，参见EE.7,“互用性”。

要求Java EE产品供应商公布应用程序组件使用的行业标准协议。本规范定义了Servlet和JSP页面到HTTP和HTTPS协议的映射，以及EJB组件到IIOP和SOAP协议的映射。

2.12.4 部署描述符和注解

用部署描述符和Java语言注解可以将应用程序组件的需求传达给部署者。部署描述符和类文件的注解是应用程序组件供应商(或组装者)和部署者之间的 协议。应用程序组件供应商或组装者必须在组件的部署描述符或类文件注解中说明应用程序组件的外部资源需求，安全需求，环境参数等等。Java EE产品供应商必须提供一个部署工具来解释Java EE部署描述符和类文件注解，并允许部署者将应用程序组件的需求映射到特定Java EE产品的功能和环境上。

2.13 J2EE 1.3中的变化

J2EE 1.3规范用新增的企业级整合功能扩展了J2EE平台。连接器API支持对外部企业信息系统的整合。现在，JMS供应商的存在是必须的。JAXP API对处理XML文档提供了支持。JAAS API对连接器API提供了安全支持。EJB规范现在需要对互用性提供支持，这需要使用IIOP协议。

EJB规范有了重大的改变。EJB 规范有了一个新的受容器管理的持久化模型，对消息驱动Bean和本地企业Bean提供了支持。

同样也更新了其它的J2EE API。请查看它们各自的API规范了解详细信息。 最后，J2EE 1.3要求支持J2SE 1.3

2.14 J2EE 1.4中的变化

J2EE 1.4的主要焦点是对Web服务提供支持。JAX-RPC和SAAJ API提供了对基本Web服务互用性的支持。J2EE Web服务规范描述了提供和使用Web服务的J2EE应用程序的打包和部署标准。EJB规范也有了新的扩展，通过使用无状态会话Bean实现对Web服务 的支持。JAXR API 支持对注册和存储的访问。

J2EE 1.4中添加了几个新的API。J2EE管理和部署API使增强后的工具能够支持J2EE产品。JMX API支持J2EE管理型API。J2EE容器授权协议为安全提供方提供了一个SPI。

许多J2EE API在J2EE 1.4中得到增强。J2EE 1.4构建在J2SE 1.4上。增强后的JSP规范简化了Web应用程序的部署。连接器API现在支持对异步消息系统的整合，包括插入JMS提供方。

J2EE平台的更新包括: 对部署独立于任何应用程序的类库的支持, 部署描述符从DTD向XML Schema的转换。

其它的J2EE API也得到了增强。详细信息参见它们各自的规范。

2.15 Java EE 5中的变化

首先，你可能也注意到了，这个平台发布了一个新的名称--Java平台企业版，简称 Java EE。这个新名称去掉了令人费解的“2”，而这个简称强调了这是一个Java平台。以前的版本仍然使用旧的名称“J2EE”。

Java EE 5 的焦点是简化开发。为了帮助刚从Java EE起步的程序员简化开发流程或开发中小型应用，我们大量使用了在J2SE5.0中引入的Java语言注解。

在大多数情况下，注解减少甚至消除了对Java EE部署描述符的处理。甚至大型应用程序也能从注解所带来的简便性中受益。

注解最主要的应用之一是为Java EE组件指定资源注入和其它依赖关系。注入提高了当前JNDI的查找能力，为应用程序提供了一个新的简化模型，使它从运行环境中访问必需资源的效率得到提 高。注入也可以和部署描述符一起使用，使部署者可以自定义或重写应用程序源代码中的资源设置。

通过使用注解，优化的默认值将发挥更大的效用。在大多数没有注解或部署描述符的情况下，优化的默认行为和优化的默认配置，可以使大多数应用程序能在大部分时间内获得想要的行为。但是，当默认值不被应用程序需要时，一个简单的注解就可以指定所需的行为或配置。

注解与优化的默认值的组合大大地简化了使用EJB技术的应用程序和使用Web服务的应用程序的部署。现在，开发企业Bean相当地简单。通过使用Web服务元数据规范定义的注解，Web服务的开发也变得更加容易。

Web服务领域的发展日新月异。为了对最新的Web服务提供支持，JAX-RPC演变成了JAX-WS技术，它大量使用JAXB技术将Java对象绑定到XML数据。JAX-WS和JAXB都是本平台的新内容。

Java EE 5新增的主要内容还包括JSTL和JSF技术，用它们可以简化Web应用程序的部署。还有EJB3.0专家组开发的Java持久化API，它极大地简化了从Java对象到数据库的映射。

另外还增加了用于解析XML的StAX API。先前版本的大多数API都得到了或多或少地改善。

2.16 Java EE 6中的变化

Note -

第三章 安全

本章描述了Java™平台企业版(Java EE)的安全标准。Java EE产品必须满足此标准。

除了Java EE标准，每个Java EE产品供应商应该确定它们的实现中提供的安全和安全保证级别。

3.1 简介

几乎每个企业都有自己的安全标准和明确的机制，以及相应的基础设施来满足它们。敏感资源需要得到保护，它们可能被很多用户访问，或者常常处于无保护状态，在开放的网络环境(例如因特网)中穿行。

尽管质量保证和实现细节可能不同，但它们都有以下共同特征:

验证: 意思是，通讯实体(例如，客户端和服务器)彼此验证，以经过访问授权的特定标识为依据。

资源的访问控制：意思是，资源的交互仅限于某些用户或程序的集合，其目的是对完整性，保密性或可用性实施强制约束。

数据完整性：意思是，检验信息是否被第三方(非信息源的其它实体)修改。例如，处于开放网络环境中的数据接收方必须能够检测并丢弃那些在传递过程中被修改过的消息。

机密性或数据隐私：意思是，确保信息仅对经过访问授权的用户可用。

不可否认：意思是，对用户进行检验，让他无法否认自己进行过的活动。

审核：意思是，捕获一个安全相关事件的防篡改记录，目的是评估安全策略和机制的有效性。

本章详细说明了Java EE平台标准怎样解决安全问题和标识问题，这也可以由Java EE产品供应商来解决。最后，本规范未来版本中的议题会在3.7, “未来的方向”中提到。

3.2 一个简单的例子

让我们通过一个小的例子来更好地理解Java EE环境的安全行为。这个示例程序有一个Web客户端，一个JSP用户界面以及企业Bean业务逻辑(这种模式组合并不是绝对的)。

在这个例子中，Web客户端依靠Web服务器作为它的身份验证代理。Web服务器将从客户端收集用户的身份验证数据并使用它们来建立一个可靠的会话。

第1步：首次请求 Web客户端请求主程序的URL。见图3-1

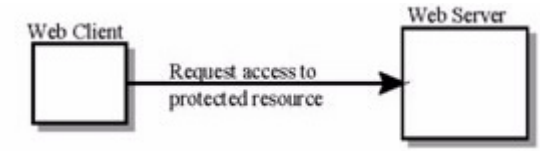

图3-1 首次请求

由于客户端还没有向主程序证明自己的身份，服务器将请求传递给应用程序的Web部分(以下称为“Web服务器”)，为请求的资源查找和引用合适的身份验证机制。

第2步：首次验证

Web服务器返回了一个Web客户端用户用于整理身份验证数据的表单(例如，用户名和密码) 。Web客户端重新将身份验证数据发给Web服务器的验证模块。见图3-2

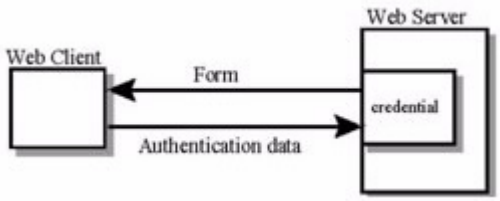

图3-2 首次验证

验证机制可以从本地到服务器，也可以触发底层的安全服务。通过验证之后，Web服务器将为用户设置一个证书。

第3步：URL授权

证书在以后用来判定用户是否有权访问它所请求的资源。Web服务器依据与Web资源相关的安全策略(取自部署描述符)来决定安全角色及其可以访问的资源。接着，Web容器将测试用户证书的规范性，并决定它是否可以将此用户映射到这个角色。图3-3 演示了这个过程。

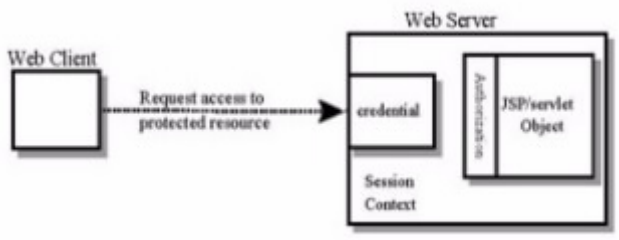

图3-3 URL授权

如果Web服务器能够将用户映射到某个角色，就会出现一个“已授权”的结论，Web服务器的评估过程也就结束了。如果Web服务器不能将用户映射到任何一个合法的角色，那么出现的结论就是“未授权”。

第4步：响应请求

如果用户授权成功，那么Web服务器就会返回之前的URL请求的结果。见图3-4

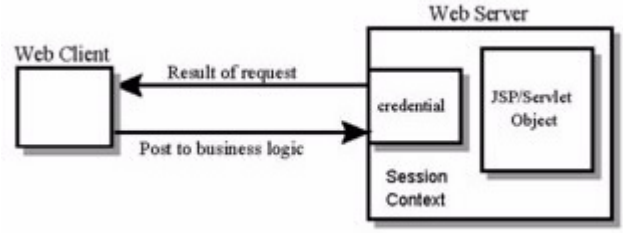

图3-4 响应请求

在这个例子中，URL响应返回的是一个JSP页面，但有可能用户提交的表单数据需要由应用程序的逻辑组件进行处理。

第5步：调用企业Bean的业务方法

JSP页面执行远程方法来调用企业Bean，用户证书用来建立JSP页面和企业Bean之间的安全关联(见图3-5)。这种关联被两个相关的安全上下文实现，一个在Web服务器中，另一个在EJB容器中。

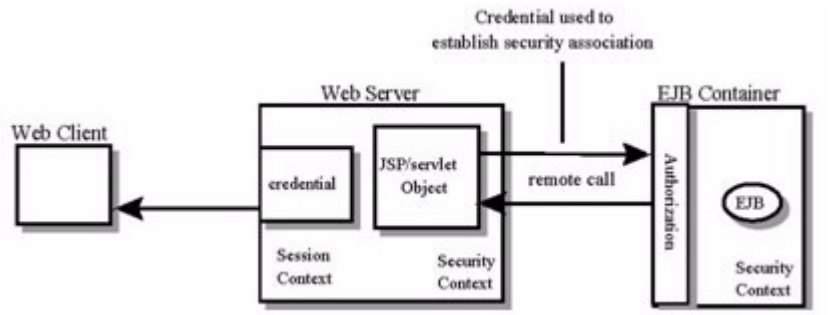

图3-5 调用企业Bean的业务方法

EJB容器负责用企业Bean的方法实施访问控制。它依据与企业Bean相关的安全策略(取自部署描述符)来决定安全角色可以访问的方法。对每个角色，EJB容器使用与此调用相关的安全上下文来决定它是否可以将此调用方映射到这个角色。

如果容器可以将此调用方的证书映射到某个角色，就会出现一个“已授权”的结论，容器的评估过程也就结束了。如果容器不能将此调用方映射到任何一个

合法的角色，那么出现的结论就是“未授权”。如果结论为“未授权”，会使容器抛出一个异常，并传回到调用它的JSP页面。

如果调用的结果是“已授权”，容器就会派发对企业Bean方法的控制。Bean会将调用的执行结果返回给JSP，并最终传给Web服务器和Web客户端的用户。

3.3 安全体系结构

本节描述了Java EE安全体系结构，它基于本规范所定义的标准。

3.3.1 目标

以下是Java EE安全体系结构的目标:

1.可移植性: Java EE 安全体系结构必须支持“一次书写，到处运行”的应用程序特性。

2.透明性: 应用程序组件供应商编写应用程序时，不需要了解任何安全事项。

3.隔离性: Java EE 平台应该能够执行验证和访问控制，这需要依据部署者使用部署属性制定的操作指南，并且能够被系统管理员管理。

注意，在责任上分离应用程序所提供的安全性确保了Java EE应用程序有极高的可移植性。

4.可扩展性: 平台服务的安全性不能向可移植性妥协。

本规范提供了组件编程模型的API，以实现容器/服务器安全信息的交互。由API提供对它们交互的限制，应用程序仍然可以保持其可移植性。

5.灵活性: 符合本规范的应用程序使用的安全机制和声明不应该强加特殊的安全策略，除非安全策略的实现确实有利于Java EE的安装或应用程序。

6.抽象性: 应用程序组件的安全标准通过部署描述符在逻辑上确定。部署描述符说明安全角色和访问条件怎样被映射成特定环境的安全角色，用户和策略。部署者可以选择性地修改安全属性来符合部署环境。部署描述符应该记载可以修改和不可以修改的安全属性。

7.独立性: 通过使用多种安全技术，应该可以实现必须的安全行为和部署协议。

8.兼容性测试: Java EE安全标准体系结构必须用某种方式表述，此方式能够明确的判定某个实现是否是兼容的。

9.安全交互性: 运行在Java EE产品中的应用程序组件必须能够调用不同厂商的Java EE产品提供的服务，不管它们是否使用相同的安全策略。这些服务也可以由Web组件或企业Bean提供。

3.3.2 非目标

以下不属于Java EE安全体系结构的目标:

1.本规范不明确规定具体的安全策略。应用程序和企业信息系统的安全策略会受很多因素影响，这与本规范无关。产品供应商可以使用必要的技术来实现和管理想要的安全策略，然而必须遵循本规范的标准。

2.本规范不强制具体的安全技术，例如Kerberos, PK, NIS+, 或NTLM。

3.本规范不要求使用任何甚至所有的安全技术来实现所有的Java EE安全行为。

4.本规范对Java EE产品的安全效果不提供任何承诺或保证。

3.3.3 术语

本节介绍了用于描述Java EE平台安全标准的术语。

主体

主体是一个可验证的实体。它的验证方式基于部署在企业安全服务中的验证协议。主体使用主体名进行标识，使用验证数据进行验证。主体名和验证数据的内容和格式由验证协议决定。

安全策略域

安全策略域也被称为安全域，它的范围覆盖了安全服务管理员定义和实施的公共安全策略。

安全策略域有时也被称为领域。本规范使用安全策略域或安全域作为术语。

安全技术域

安全技术域的覆盖范围与安全机制(例如Kerberos)相同，这些安全机制用来强制实施安全策略。

例如，单个安全技术域可以包含多个安全策略域。

安全属性

每个主体都有一套安全属性。安全属性有很多用途(例如，访问受保护的资源和审核用户)。安全属性与主体的关联可以由验证协议或Java EE产品供应商或这两者共同决定。

Java EE平台不规定主体应该和什么安全属性相关联。

证书

证书包含或引用了安全属性，这些安全属性信息用于验证Java EE产品服务的主体。主体通过验证来获得证书，也可以获取另外一个主体的证书(在允许代理的情况下)。

本规范没有证书的内容或格式。它们的表现形式可能会有很大差别。

3.3.4 容器的安全

在Java EE环境中，组件的安全由容器提供，这是为了达到上面规定的安全目标。容器提供了两种安全(在后续章节中讨论):

声明式安全

编程式安全

3.3.4.1 声明式安全

声明式安全指的是用应用程序外部的形式表述一个应用程序的安全结构，包括安全角色，访问控制和验证条件。部署描述符是Java EE平台声明式安全的主要表述方式。

部署描述符是应用程序组件供应商和部署者(或组装者)之间的协议。应用级程序员可以使用它来描述与应用程序安全相关的环境标准。部署描述符可以关联到多个组件。

部署者将部署描述符的内容映射到特定的环境。它描述了特定安全结构的应用程序的安全策略。部署者使用部署工具来处理部署描述符。

运行时，容器使用安全策略的安全结构强行授权。安全结构来自部署描述符，并且由部署者配置(请查看 3.3.6,“授权模式”)。

3.3.4.2 编程式安全

编程式安全指的是安全决策由应用程序制定。当声明式安全不足以表述应用程序的安全模式时，附加编程式安全可以解决这一问题。

实现编程式安全的API由EJBContext接口的两个方法和HttpServletRequest接口的两个方法组成:

isCallerInRole (EJBContext)

getCallerPrincipal (EJBContext)

isUserInRole (HttpServletRequest)

getUserPrincipal (HttpServletRequest)

这些方法允许组件制定基于调用方或远程用户的安全角色的业务逻辑决策。例如，它们允许组件决定以调用方或远程用户的主体名作为数据库的键。(注意，在产品和企业之间，主体名的形式和内容的表现形式会有很大的差别，并且可移植的组件不会依赖主体名的实际内容。由于主体名的映射，相同的逻辑主体在不同的 容器中可能有不同的名称，尽管它可能通常用一致的主体名配置单个产品。特别是，如果一个主体名被用作数据库表的一个键，并且那个数据库表可以被多个组件， 容器或产品访问，那么相同的逻辑主体可以映射到数据库中的不同入口。)

3.3.5 分布式安全

一些产品供应商可以生产分布式的Java EE产品，这些产品存放在适应多种组件类型的容器中。在分布式环境下，Java EE组件间的通信容易受到安全攻击(例如，数据修改和重放攻击)。

这种威胁可以使用安全关联来解决。安全关联是共享的安全状态信息，这种信息建立在安全通信的基础上。建立一个安全关联需要几个步骤，如下:

1.验证目标主体到客户端，并(或)验证客户端到目标主体。

2.协商保护质量，例如机密性或完整性。

3.为组件间的关联建立一个安全的上下文.

因为在Java EE中由容器提供安全，所有组件的安全关联通常由容器建立。Web访问的安全关联在这里确定。访问企业Bean的安全关联在EJB规范中描述。

产品供应商可以允许在部署时控制保护的质量，或者其它安全关联方面的质量。通过使用部署描述符中的元素，应用程序可以指定它们对Web资源的访问标准。应用程序组件供应商能够使用关联到企业Bean的安全通信标准，但本规范没有定义这些机制。

3.3.6 授权模式

Java EE授权模式基于安全角色的概念。一个安全角色是一个用户的逻辑组，它由应用程序组件供应商或组装者定义。部署者将角色映射到运行环境中的安全标识(例如，主体和组)。安全角色被声明式安全和编程式安全使用。

声明式验证可以用来控制对企业Bean方法的访问，它在企业Bean部署描述符中指定。企业Bean方法可以关联到部署描述符中的method- permission元素。method-permission 元素包含一组可以被给定安全角色访问的方法。如果发起调用的主体属于某个安全角色，并且此角色允许访问某个方法，那么这个主体就可以执行该方法。相反地， 如果发起调用的主体不属于任何安全角色，那么此调用方将不能执行这些方法。对Web资源的访问可以用类似的方式进行保护。

安全角色可以用于EJBContext的isCallerInRole方法和HttpServletRequest的isUserInRole方法中。如果发起调用的主体属于指定的安全角色，那么这些方法就会返回true。

3.3.6.1 角色映射

实施对Web资源和企业Bean的安全约束，取决于与传入请求相关联的主体是否属于给定的安全角色，而不论它是编程式的还是声明式的。容器根据发起调用的主体的安全属性来做出决定。例如，

1. 部署者可能已经将一个安全角色映射到了运行环境中的用户组。在这种情况下，发起调用的主体所在的用户组可以从它的安全属性中取出。如果主体的用户组匹配一个安全角色已经映射过的用户组，那么此主体属于安全角色。

2. 部署者可能已经将一个安全角色映射到安全策略域中的一个主体名。在这种情况下，发起调用的主体的主体名可以从它的安全属性中取出。如果这个主体名与被映射的安全角色的主体名相同，那么这个发起调用的主体就属于安全角色。

安全属性的来源可能有多种形式，这要依据具体的Java EE平台。安全属性可以分送给调用方主体的证书或安全上下文。在别的情况下，可以从信任的第三方取出安全属性，例如目录服务或安全服务。

3.3.7 HTTP网关登录

EJB规范描述了不同安全策略域中的企业Bean之间的安全交互。另外，组件可以选择通过HTTP登录外部服务器。当使用HTTP访问一个远程资源 时，应用程序组件可以使用SSL双向验证进行配置。使用HTTP的应用程序可以选择使用XML或一些其它的结构化的格式来取代HTML。

使用带SSL双向验证的HTTP来访问远程服务的情况，我称为HTTP网关登录。这个领域的标准在3.3.8.1, “Web客户端验证”中定义。

3.3.8 用户验证

用户验证是用户向系统证明自己的标识的过程。这个被验证的标识接着被用来执行授权决定，以访问Java EE应用程序组件。一个终端用户可以使用以下两种受支持的客户端类型进行验证:

Web客户端

应用程序客户端

3.3.8.1 Web客户端验证

Web客户端能够使用下列任意机制到Web服务器去验证用户。部署者或系统管理员将决定为一个或一组应用程序应用哪种方法。

HTTP基本验证

HTTP基本验证是被HTTP协议支持的验证机制。这种机制基于用户名和密码。Web服务器要求Web客户端对用户进行验证。作为请求的一部 分，Web服务器将传递一个域，用户会在这个域中接受验证。Web客户端从用户获得用户名和密码，并且将它们传送到Web服务器。然后，Web服务器会在 指定的域(本文档中称为HTTP域)中验证用户。

HTTP基本验证并不安全。密码用简单的base64编码发送。目标服务器没有经过验证。增加额外的保护可以弥补这一不足。在传输层(例如HTTPS)或者网络层(例如，IPSEC或VPN)应用安全措施可以保护密码。

尽管它有一定局限性， HTTP基本验证机制仍然被包含在本规范中，因为它广泛地使用在基于表单的应用程序中。

HTTPS客户端验证

使用HTTPS(HTTP over SSL)的终端用户验证是一种健壮的验证机制。这种机制要求用户拥有一把公钥证书(PKC)。目前，互联网上的终端用户很少使用PKC。然而，它对电子商 务应用程序和来自浏览器的单点登录来说是有用的。因此，HTTPS客户端验证是Java EE平台必要的特性。

基于表单的验证

使用Web浏览器内置验证机制的登录界面的外观是单调的。本规范对基于表单登录的标准HTML或Servlet/JSP引入了包装的功能，它允许自定义用户界面。本规范引入的基于表单的验证机制被描述于Servlet规范中。

由于HTTP摘要验证没有受到Web浏览器的广泛支持，因此不做要求。

Web客户端可以使用一个Web服务器作为它的验证代理。在这种情况下，服务器会创建一个客户端证书，这个证书可以被服务器用于多种目的: 执行授权判断,在调用企业Bean时充当客户端, 或者达成资源的安全关联。现在的浏览器通常都信任代理验证。

3.3.8.2 Web单点登录

HTTP是一种无状态协议。然而，很多Web应用程序需要支持会话，它能跨多个客户端请求保持状态。因此, 它具有以下优势:

1.为登录机制和策略制造一个环境属性。此环境部署了应用程序。

2.能够使用相同的登录会话来代表用户去访问所有的应用程序。

3.仅当越过安全策略域边界时才需要重新验证用户。

证书将和会话关联，它贯穿整个Web登录流程。容器使用证书来为会话建立安全上下文。容器使用安全上下文来决定对Web资源的访问授权，以及对其它组件建立安全关联的授权(包括企业Bean)。

3.3.8.3 登录会话

在Java EE平台上，Web容器对登录会话提供支持。当用户成功通过Web服务器的验证时，容器将为用户建立一个登录会话的上下文。登录会话包含与用户关联的证书。

3.3.8.4 应用程序客户端验证

应用程序客户端(EE.10,“应用程序客户端”中有详细描述)是可以与企业Bean直接交互的客户端程序(既不需要借助Web浏览器也不需要通过Web服务器)。应用程序客户端也可以访问Web资源。

应用程序客户端，像其它的Java EE应用程序组件类型一样，运行在一个受管理的环境中，这个环境由恰当的容器提供。应用程序客户端期望有一个图形显示和输入设备，并能与人沟通。当用户访 问受保护的Web资源或企业Bean时，应用程序客户端被用来验证Java EE平台的终端用户。

3.3.9 即时验证

验证是需要开销的。例如，一个验证过程可能需要跨网络多次交换消息。因此，使用即时验证是可取的，它只会在需要的时候执行验证。采用即时验证，在用户访问受保护的资源之前，都不需要进行验证。

当它们请求受保护的资源时，即时验证可以和第一层客户端(Applet, 应用程序客户端)一起使用。在这一点上，会要求用户提供恰当的验证数据。如果用户成功通过验证，用户就可以访问这个资源。

3.4 用户验证的必要条件

Java EE产品供应商必须满足下面的用户验证相关的标准。

3.4.1 登录会话

所有的Java EE Web服务器都必须为每个用户维持一个登录会话。必须能够让一个登录会话跨越一个或多个的应用程序，允许用户登录。

1.在客户端没有验证状态时，客户端需要服务器充当它的代理并且为它维持登录上下文。透过Cookie或URL重写，登录会话状态的引用的有效期会 对客户端重新计算。如果使用SSL双向验证作为验证协议，客户端就可以管理它自己的验证上下文，并且不需要依赖登录会话状态的引用。

一次访问多个应用程序时，必须对登录会话的提供支持，正如Servlet规范所描述的。每个Web用户的会话都要求支持单点登录。

应用程序可以不依赖于登录信息的安全维护的实现细节。Java EE应用程序供应商可以自由选择独立于应用程序的验证机制，这些验证机制保障了应用程序的安全性。

Web服务器必须支持即时验证来保护Web资源。当需要验证的时候，可以使用下一节列出的三种机制之一。

3.4.2 必要的登录机制

所有Java EE产品都必须支持三种登录机制：HTTP基本验证，SSL双向验证和基于表单的登录。不要求应用程序必须使用这些机制，但是这些机制对任何应用程序都是可用的。

3.4.2.1 HTTP基本验证

所有Java EE产品都必须支持HTTP基本验证(RFC2068)。平台供应商也必须支持带SSL的基本验证。

3.4.2.2 SSL双向验证

本规范要求支持SSL 3.02和基于双向证书的验证。

所有Java EE产品必须支持下列加密套件来确保与客户端的交互性验证：

TLS_RSA_WITH_RC4_128_MD5

SSL_RSA_WITH_RC4_128_MD5

TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA

SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA

TLS_RSA_EXPORT_WITH_RC4_40_MD5

SSL_RSA_EXPORT_WITH_RC4_40_MD5

TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

2.关于S SL 3.0规范请链接http://home.netscape.com/ eng/ssl3

这些加密套件被主流浏览器支持，并且符合美国政府的传播规定。

3.4.2.3 基于表单的登录

Web应用程序部署描述符包含一个元素，它可以让Java EE产品将一个HTML表单资源(也许是动态生成)和Web应用程序关联。如果部署者选择这种表单验证（基于HTTP基本验证或基于SSL的证书验证)，那它必须作为用户的登录界面。

基于表单的登录机制和Web应用程序部署描述符在Servlet规范中描述。

3.4.3 未验证的用户

Web容器必须支持还没有被容器验证的客户端访问Web资源。这是因特网上访问Web资源的常见模式。

如果HttpServletRequest的getUserPrincipal方法返回null，Web容器就会通知用户没有被验证。这和EJB容 器的相应结果不同。EJB规范要求EJBContext的getCallerPrincipal 方法总是返回一个有效的证书对象，它从不返回null。

在同时包含Web容器和EJB容器Java EE产品中，运行在Web容器中的组件必须能够调用企业Bean，甚至用户在Web容器中没有被验证。当Web容器中的组件对企业Bean的调用出现了这种情况时，Java EE产品必须在调用中为用户提供一个主体 。

Java EE产品可以采用多种方法给未验证的调用方提供一个主体，这包括但不局限于:

总是使用一个单独的特殊主体。

为每个服务器，每个会话或每个应用程序使用一个不同的特殊主体。

通过Web容器和企业Bean容器RunAs功能，部署者或系统管理员可以选择要使用的主体。

本规范没有指定Java EE产品应该怎样选择一个主体来代表未验证的用户，但是，本规范的未来版本可能会增加这个领域的标准。注意，在使用EJB互用性协议的时候，EJB规范中 没有包含此领域的标准。在Web组件可以被验证并且需要调用EJB组件的时候，鼓励应用程序使用这种RunAs功能。

3.4.4 应用程序客户端用户验证

应用程序客户端容器必须提供对用户的验证，以满足验证和授权约束，这些约束由企业Bean容器和Web容器强制实施。由于应用程序客户端容器在实现 上的差异，它们使用的技术也会不同，并且超出了应用程序的控制。应用程序客户端容器可以和Java EE产品的验证系统进行整合，以提供单点登录功能，或者容器可以在应用程序启动后验证用户。容器可以延迟验证，直到用户请求访问受保护的资源或企业 Bean。

容器将提供一个恰当的用户界面来收集验证数据。另外，应用程序客户端可以提供一个实现了 javax.security.auth.callback.CallbackHandler接口的类，并在它的部署描述符中指定这个类的名称(请查看 EE.10.7,“Java EE应用程序客户端的XML Schema”中的细节)。部署者可以重写应用程序指定的回调处理器，并且可以要求使用容器默认的用户验证界面来代替。

如果部署者配置了回调处理器，应用程序客户端容器必须实例化这个类的一个对象，并且用它完成对用户的所有验证。应用程序的回调处理器必须支持所有定义在javax.security.auth.callback包中的Callback对象。

应用程序客户端可以运行在Java SE安全管理器控制的环境中，并且遵守EE.6.2,“Java平台标准版(Java SE)标准”的安全许可。尽管本规范没有定义运行应用程序的操作系统标识和用户验证标识之间的关系，但是对单点登录

提供的支持要求Java EE产品能够关联这些标识。其它的应用程序客户端标准在本规范的EE.10.7中描述。

3.4.5 资源验证的必要条件

企业中的资源常常部署在安全策略域中，它不同于应用程序组件的安全策略域。资源调用方的验证机制的广泛差异带来了这样一个标准，Java EE产品提供了在资源的安全策略域中进行验证的方式。

产品供应商必须支持以下两种方式:

1.配置标识 Java EE容器必须能够使用部署者在部署时指定的主体和验证数据对资源访问进行验证。这种验证必须不依赖于应用程序组件以任何方式提供的数据。为验证信息提供机密存储是产品供应商的责任。

2.编程式验证 Java EE产品必须提供主体和资源的验证数据的规格，这需要应用程序在运行时使用恰当的API。应用程序可以通过多种机制获得主体和验证数据，这包括以参数接收，从组件环境获取等等。

另外，推荐使用以下技术，但它们不在本规范的要求内:

1.主体映射 资源可以有一个主体和若干属性，它们由请求方主体的标识和安全属性的映射决定。在这种情况下，资源主体不是基于请求方主体的标识或安全属性的继承，而是基于这种映射获取它的标识和安全属性。

2.充当调用方 资源主体代表请求方主体进行活动。代表调用方主体进行活动时需要调用方标识和底层资源管理器证书的委托。在某些情况下，请求方主体可以代表启动主体，因此资源主体传递性地充当了启动主体。

对主体委托的支持通常是特定于安全机制的。例如，Kerberos支持对验证的委托机制(参见Kerberos v5规范了解更多细节)。

5.证书映射 当应用程序服务器和EIS支持的验证域不同时，可以使用此技术。例如:

a. 启动主体已经通过验证并且有公共密钥证书。

b. 资源管理器的安全环境可以配置Kerberos验证服务。

可以配置应用程序服务器，将关联启动主体的公共密钥证书，映射到Kerberos证书。

关于资源验证条件的其它信息可以在连接器规范中找到。

3.5 授权条件

为了支持本节描述的授权模式，Java EE产品必须符合以下标准。

3.5.1 代码授权

Java EE产品可以约束某些Java SE类和方法的使用，保障系统的正确运行。Java EE产品必须授予Java EE应用程序的权限的最小集合定义在EE.6.2,“Java平台标准版(Java SE)标准”中。所有Java EE产品必须能够正确地使用这些权限来部署应用程序组件。

使用Java保护模式，Java EE产品供应商的实现可以选择性地访问资源。这种机制依赖于Java EE产品。

Java EE部署描述符未来版本的定义(参见EE.8,“应用程序组装者和部署”)可能会描述新的权限来满足组件对访问的需求。

3.5.2 调用方授权

Java EE产品必须在部署时强制实施指定的访问控制规则(参见3.6,“部署条件”)，更完整的信息描述在EJB和Servlet规范中。

3.5.3 调用方标识的传递

必须正确地配置包含了EJB容器的Java EE产品，以便被传递的调用方标识可以用在所有的授权判断中。使用这种配置，单个Java EE产品中的单个应用程序可以调用所有的企业Bean的。EJBContext的getCallerPrincipal方法返回的主体名必须和调用链中的 第一个企业Bean的返回值相同。如果调用链中的第一个企业Bean被一个Servlet或JSP页面调用，这个主体名必须和它们中的HttpServletRequest的getUserPrincipal方法的返回值相同。(然而，如果HttpServletRequest的 getUserPrincipal方法返回null，这个用来调用企业Bean的主体就没有被本规范指定，尽管它仍然必须能够配置企业Bean来让这样的 组件随时调用。)

注意，这不需要证书的委托，只需要调用方的标识。单个主体必须用在授权判断中，以访问调用链中所有的企业Bean。只有当Java EE产品的配置已经允许传递调用方标识，本节的标准才会生效。

3.5.4 Run As标识

Java EE产品也必须支持Run As功能，它允许应用程序组件供应商和部署者指定一个标识，企业Bean或Web组件必须运行在此标识下。在这种情况下，Run As标识会被传递到随后的EJB组件中，而不是传递最初的调用方标识。

注意，本规范没有指定Run As标识和底层操作系统标识之间的任何关系(底层操作系统标识用于访问系统资源，例如文件)。然而，容器规范的Java授权协议指定了Run As标识和访问控制的上下文(被Java SE安全管理器使用)之间的关系。

3.6 部署标准

所有Java EE产品必须实现所有组件规范中描述的访问控制语义，例如EJB, JSP和Servlet规范。并且它提供了将部署描述符的安全角色映射到实际角色(由Java EE产品暴露)的方法。

尽管大多数Java EE产品会允许部署者自定义角色映射和改变角色的分配方法，但是所有的Java EE产品都必须支持这种功能来部署应用程序和组件，准确地使用它们部署描述符中指定的映射和分配。

正如EJB规范和Servlet规范所描述的，Java EE产品必须提供一个部署工具或一些能够将部署描述符中的安全角色分配给实体的工具，这些实体用来在授权时确定角色的资格。

应用程序开发者可能需要指定应用程序的安全标准(在程序的部署描述符中)，使一些组件可以被已验证的用户访问，也可以被未验证的用户访问(正如上面的3.4.3, “未验证的用户”中所描述的)。应用程序从安全角色的角度来表述它们的安全标准，部署者在运行时将这些标准映射到运行环境中的用户(主体)。应用程序可以 定义一个角色来代表所有已验证和未验证的用户，并为这个角色配置一些它可以访问的企业Bean方法。

为支持这种用法，本规范要求能够将一个定义了安全角色的应用程序映射到不依赖于验证的应用程序主体的共同集合中。

3.7 未来的方向

3.7.1 审核

本规范没有指定安全相关事件的审核标准，也没有为应用程序组件提供API来生成审核记录。本规范的未来版本可能会包含这样一种规范，来满足要提供审核的产品。

3.7.2 基于实例的访问控制

一些规范需要基于内容而不是简单的数据类型来控制对数据的访问。我把这称作“基于实例”，而不是“基于类”的访问控制。我们希望在未来的发布中解决这个问题。

3.7.3 用户注册

基于Web的网络应用程序常常需要动态地管理一组客户，允许用户注册成新的客户。这个方案在Servlet专家组(JSR-53)中进行了广泛的讨 论，但是在适当的解决方法上，我们没能达成一致的意见。我们不得不在J2EE1.3中放弃了这项工作，也没能在J2EE1.4中解决它，希望能在未来的发 布中推进它。

第四章 事务管理

本章描述了Java™平台企业版(Java EE)必须的事务管理和运行时环境。

产品供应商必须透明地支持事务，用它关联单个Java EE产品中的多种组件和业务资源，正如本章要描述的。此标准必须被满足，不论这个Java EE产品是在同一个网络节点上用单个或多个进程实现，还是在多个网络节点上用多个进程实现。

如果Java EE产品包含下列组件， 那么它们就属于事务资源并且必须按此标准工作:

JDBC连接

JMS会话

资源适配器连接，它们为资源适配器指明了XATransaction事务级别

4.1 概述

同时包含了Servlet容器和EJB容器的Java EE产品必须支持事务型应用程序在一次事务中访问多个企业Bean，此应用程序由Web组件组合而成。如果这个Java EE产品也包含了对连接器规范的支持，那么每个组件也可以获得一个或多个的连接来访问一个或多个的事务资源管理器。

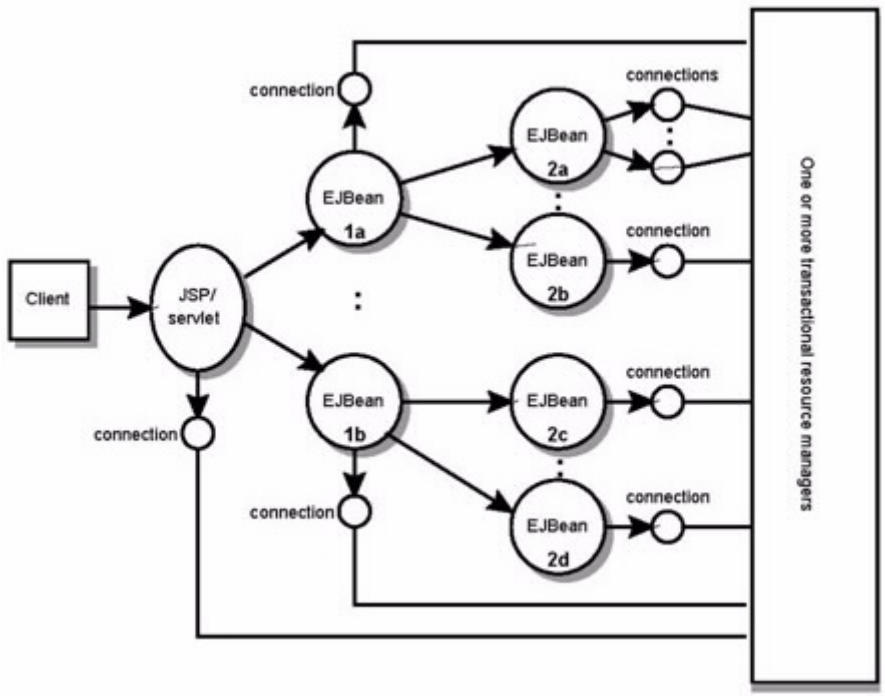

图4-1 Servlets/JSP页面访问企业Bean

例如，在图4-1中，调用树从一个Servlet或JSP页面开始访问多个企业Bean，这些企业Bean又可以访问其它的企业Bean。组件通过“连接”访问资源管理器。

应用程序组件供应商使用编程式和声明式事务分隔API的某种组合来指定这个平台必须怎样为应用程序管理事务。

例如，应用程序可能需要将图4-1中所有组件对资源的访问，作为单个事务的一部分。平台供应商必须为这种情形提供事务功能。

本规范没有定义组件和资源在单个Java EE产品中如何分隔和分布。为了实现本规范要求的事务语义，Java EE产品供应商可以在同一个Java虚拟机中自由地运行多个应用程序组件来完成一次事务，也可以将它们分布在多个虚拟机上。

本章剩余部分更详细地描述Java EE产品的事务标准。

4.2 标准

本节定义了产品供应商必须支持的Java EE产品的事务标准。

4.2.1 Web组件

Servlet和JSP页面使用JTA规范中定义的javax.transaction.UserTransaction接口分隔事务。它们可以在 一次事务中访问多个资源管理器和调用多个企业Bean。这个指定事务的上下文会自动地传递给企业Bean和事务资源管理器。传递的结果可能会受企业 Bean属性的支配(例如，一个Bean可能需要将事务托管给容器)。

Servlet过滤器和Web应用程序事件监听器不能使用javax.transaction.UserTransaction接口分隔事务。除了 Servlet过滤器可以在doFilter方法中的本地事务模式中使用事务资源，其它封装了request或response对象的方法中不能使用任何 事务资源。

4.2.1.1 事务标准

Java EE平台必须满足下列标准:

Java EE平台必须为所有的Web组件提供一个实现了javax.transaction.UserTransaction接口的对象。此平台必须在Java™命名和目录接口(JNDI)命名空间中发布这个UserTransaction对象，使Web组件在可以在命名java:comp/UserTransaction中找到它。

如果Web组件从一个与JTA事务关联的线程调用企业Bean，Java EE平台必须随着企业Bean的调用传递这个事务的上下文。目标企业Bean是否会在这个事务上下文中被调用，由定义在EJB规范中的规则决定。

注意，这个事务传递标准只适用于同一个Java EE产品实例1中的企业Bean调用。在另一个Java EE产品实例中调用企业Bean(例如，使用EJB互用性协议)不需要传递事务上下文。详细信息参见EJB规范。

如果Web组件从一个与JTA事务关联的线程访问事务资源管理器，Java EE平台必须确保这个资源访问属于这个JTA事务的一部分。

如果Web组件创建了一个线程，Java EE平台必须确保这个新的线程没有关联到任何JTA事务。

4.2.1.2 非事务标准

不要求产品供应商支持将事务上下文从客户端导入Web组件。

不要求产品供应商支持将事务上下文通过HTTP请求传递给Web组件。HTTP协议不支持事务上下文的传递。当与事务关联的Web组件向另一个Web组件发出了一个HTTP请求，这个事务上下文不会传递给目标Servlet或JSP页面。

然而，当通过RequestDispatcher接口调用Web组件时，任何活动的事务上下文环境必须传递给被调用的Servlet或JSP页面。

---

1. 一个产品实例相当于安装一次Java EE产品。一个单独的产品实例可以使用多个操作系统进程，也可以支持多个主机作为一个分布式容器的组成部分。相反地，在单个主机上甚至单个Java虚拟机 上运行一个产品的多个实例也是可能的，例如，作为虚拟主机托管方案的一部分。事务传递标准应用在单个产品实例中，并且不依赖于Java虚拟机，操作系统进 程或主机的数量。

4.2.2 Web组件生命周期中的事务

事务不可以跨越来自互联网的客户端请求。Web组件在Servlet (JSP页面或等同于JSP页面实现类的方法)的service方法中启动事务，并且这个事务必须在这个service方法返回给网络客户端之前结束。将 活跃事务的上下文返回给网络客户端是错误的。Web容器必须检测这种错误并且终止此事务。

正如4.2.1.2中规定的，如果使用了RequestDispatcher接口，在Web容器中产生的“非事务标准”请求必须将相应的事务上下文传递给被调用的类。当被调用的类返回时，该事务仍然必须保持活跃，除非这个被调用的类提交或终止了该事务。

如果通过RequestDispatcher调用的Servlet启动了一个事务，当这个Servlet从它的业务方法返回时，有关该事务的容器行 为是不确定的。这个Web容器可以向调用方抛出一个异常，也可以终止该事务并且不带有任何错误信息，或者是将该事务的上下文返回给调用方。可移植的 Servlet应该在业务方法返回前完成它们启动的任何事务。

4.2.3 事务和线程

在事务资源的使用和线程之间有很多细微并且复杂的交互。为了确保操作的正确性，Web组件应该遵循以下原则，并且Web容器至少必须支持这些用法。

JTA事务应该在调用业务方法的线程中启动和完成，为其它目的创建的线程不应该试图启动JTA事务。

事务资源可以被业务方法线程之外的其它线程获取和释放，但是不能在线程间共享。

事务资源对象 (例如，JDBC的Connection对象)不应该保存在静态字段中。这个对象每次仅关联一个事务。将它们保存在静态字段中很容易错误地在共享于不同事务的线程。

实现了SingleThreadModel的Web组件可以在类的实例字段中保存顶级事务资源对象。顶级对象是从容器管理的连接工厂对象中直 接获取的(例如，从JDBC ConnectionFactory中获取的JDBC Connection对象)，这和从这些顶级对象中获取的其它对象不同(例如，从JDBC Connection对象中获取的JDBC Statement对象)。Web容器确保了对SingleThreadModel Servlet的请求是一个线性序列，这样每次只有一个线程和一个事务能够使用这个对象，并且顶级对象可以参与组件启动的任何新的事务。

在没有实现SingleThreadModel的Web组件中，事务资源对象不应该保存在类的实例字段中，并且应该在相同的业务方法调用中获取和释放。

被其它Web组件(使用forward或include方法)调用的Web组件不应该在类的实例字段中保存事务资源对象。

Web组件使用的任何线程都可以调用企业Bean。以上事务上下文传递标准也描述在了EJB规范中。

4.2.4 Enterprise JavaBeans™(EJB)组件

Java EE产品供应商必须对事务提供支持，正如EJB规范所定义的。

4.2.5 应用程序客户端

不要求Java EE产品供应商为应用程序客户端提供事务管理支持。

4.2.6 Applet客户端

不要求Java EE产品供应商为Applet提供事务管理支持。

4.2.7 事务型JDBC™ 技术支持

Java EE产品必须支持数据库作为事务资源管理器，通过JDBC技术来实现。此平台必须能够在Web组件和企业Bean中使用事务型JDBC API进行访问。

必须能够在一次事务中从多个应用程序组件访问JDBC技术支持的数据库。例如，Servlet也许想要启动一个事务，访问一个数据库，并调用一个企业Bean访问相同的数据库作为同一个事务的一部分，最后提交这个事务。

Java EE产品必须提供一个事务管理器，它能够协调双相提交操作，跨越多个XA-capable JDBC数据库。如果JDBC驱动支持Java事务API的XA接口(在javax.transaction.xa包中)，那么Java EE产品必须能够使用JDBC驱动的XA接口来完成双相提交操作。Java EE产品可以通过自己特有的方式来发现JDBC驱动的XA功能，尽管这样的JDBC驱动会被连接器API作为资源适配器交付。

4.2.8 事务型JMS支持

Java EE产品必须支持一个JMS提供方作为事务资源管理器。此平台必须能够从Servlet, JSP页面和企业Bean执行事务型JMS访问。

必须能够在一次事务中从多个应用程序组件访问JMS提供。例如，Servlet也许想要启动一个事务，发送JMS消息，并调用一个企业Bean也发送一个JMS消息作为同一个事务的一部分，最后提交这个事务。

4.2.9 事务资源适配器(连接器)支持

Java EE产品必须支持使用XATransaction模式的资源适配器作为事务资源管理器。此平台必须能够从Servlet, JSP页面和企业Bean执行对资源适配器的事务访问。

必须能够在一次事务中从多个应用程序组件访问资源适配器。例如，Servlet也许想要启动一个事务，访问资源适配器，并调用一个企业Bean也访问这个资源适配器作为同一个事务的一部分，最后提交这个事务。

4.3 事务的互用性

4.3.1 多个Java EE平台的互用性

本规范不要求产品供应商为多个Java EE产品间的事务互用能力实现任何特定协议。Java EE兼容性标准既不需要来自同一个产品供应商的相同Java EE产品互用能力，也不需要来自多个产品供应商的多种产品互用能力。

当使用基于RMI-IIOP的EJB互用性协议作为EJB服务器的事务互用性协议来满足事务互用能力时，我们推荐Java EE产品供应商使用IIOP事务传递协议，它由OMG定义并在OTS规范中描述(并且被Java事务服务实现)。我们计划在本规范的未来发布中将IIOP 事务传递协议作为EJB服务器的事务互用性协议。

4.3.2 为事务资源管理器提供支持

本规范要求所有Java EE产品支持javax.transaction.xa.XAResource接口, 正如连接器规范中规定的。本规范也要求所有Java EE产品支持javax.transaction.xa.XAResource接口，在支持JTA XA API的JDBC驱动上执行双相提交操作。本规范不要求JDBC驱动或JMS供应商使用javax.transaction.xa.XAResource 接口，并且在任何情况下它们都必须满足本节中描述的事务资源管理器标准。特别是必须能够在一次JTA事务中，在一个或多个的JDBC数据库，JMS会话， 企业Bean和支持XATransaction模式的多个资源适配器上合并操作。

4.4 本地事务优化

4.4.1 要求

如果一个事务使用单个资源管理器，那么通过使用资源管理器特定的本地优化可以改进它的性能。本地事务通常比全局事务效率更高，提供更好的性能。本地优化对不同容器导入的事务没有效果。

容器可以使用本地事务优化，但不是必须的。本地事务优化对Java EE应用程序必须是透明的。

下一个章节描述了容器本地事务优化的一种可能机制。

4.4.2 一种可能的设计

本节阐述之前描述的标准可以怎样实现。

当向资源管理器建立了第一个连接并且是事务的一部分时，在此连接上就会启动资源管理器特定的本地事务。随后获取的任何连接都能共享这个本地事务，但这些连接必须是事务的一部分。

在下列条件下，可以即时启动全局事务:

当随后的连接不能共享这个第一个连接上的资源管理器的本地事务，或者它使用了不同的资源管理器时。

当一个事务会输出到不同的容器时。

全局事务即时启动后，任何随后获取的连接都可以共享第一个连接上的本地事务，或者成为全局事务的一部分，这取决于它访问的资源管理器。

当试图结束(提交或回滚)一次事务时，可能出两种情况:

如果只有一个资源管理器已经被访问并且是事务的一部分，使用资源管理器特定的本地事务机制可以完成这个事务。

如果全局事务已经启动，这个资源管理器的本地事务就会被作为全局双相提交协议中的最终资源来完成，并且使用了最终资源双相提交优化。

4.5连接共享

当Java EE应用程序获取的多个连接使用了同一个资源管理器时，容器可以选择在同一个事务域中提供连接共享。共享连接通常会使资源得到高效利用，带来更好的性能。在某些情况下，容器必须提供连接共享，详细信息参见连接器规范。

Java EE应用程序获取的资源管理器的连接默认是已共享或可共享的。如果Java EE应用程序组件想要以非共享方式使用连接，它必须提供特殊的部署信息，以阻止容器共享此连接。如果安全属性，隔离级别，字符集和本地化配置等需要变化， 就需要用到这种方式。容器不能试图共享标记为非共享的连接。如果一个连接没有被标记为非共享，它对应用程序必须是透明的，不管这个连接实际上是否已经被共 享。

Java EE应用程序组件可以使用可选的部署描述符元素res-sharing-scope来标示资源管理器的连接是否是可共享的。如果部署描述没有标示，容器必 须假定连接是可共享的。在EE.10.7,“Java EE应用程序客户端的XML Schema”, EJB规范和Servlet规范中介绍了相关的部署描述符元素。

Java EE应用程序组件可以跨多个事务缓存并重用连接对象。提供连接共享的容器必须透明地切换被缓存的连接对象(在分发时)，指示一个具有正确事务域的恰当的共享连接。请查阅连接器规范，获取有关连接共享的详细信息。

4.6 JDBC和JMS部署问题

4.2.7,“事务型JDBC™技术支持”中的JDBC事务标准和4.2.8,“事务型JMS支持”中的JMS事务标准允许在应用程序的JDBC和JMS资源的部署配置中强制实施一些约束。Java EE产品供应商可以强制实施本节描述的约束，以满足这些标准。

如果部署者在一个事务中配置了一个没有XA功能的JDBC资源管理器，那么Java EE产品供应商就可以在这个事务中约束所有JDBC对它的访问。否则，Java EE产品供应商必须在一个事务中支持具有多种XA功能的JDBC资源管理器的使用。另外，Java EE产品供应商可以在单个用户标识的一次事务中约束所有JDBC连接的安全配置。不要求Java EE产品供应商对使用多个JDBC标识的事务提供支持。也就是说，需要使用多个JDBC安全标识(由提供用户名和密码的组件生成)的事务可能缺乏可移植 性。

Java EE产品供应商可以制定同样的约束，使事务可以由单个JMS资源管理器和用户标识约束。

另外，当JDBC资源管理器和JMS资源管理器被用于同一个事务时，Java EE产品供应商可以成对约束它们，这就使它们的组合能够符合应用程序要求的完整的

事务语义，并且可以将它们的两个标识合并成一个安全标识来进行约束。为了完整地支持这种用法，可移植的应用程序不能将相应的事务资源标记为“非共享”。

虽然允许使用这些约束，但推荐Java EE产品供应商支持提供完整双相提交功能的JDBC和JMS资源管理器，而不用强制实施这些约束。

4.7 双相提交支持

要求Java EE产品在单个事务中支持具有多种XA功能的资源适配器。为了满足这种情况，必须完整地支持双相提交。JMS提供方可以作为具有XA功能的资源适配器使 用。在这种情况下，它必须能够在同一个全局事务中包含JMS操作，如同其它的资源适配器。尽管JDBC驱动不需要XA功能，但是它也可以作为一个具有XA 功能的资源适配器交付。在这种情况下，必须能够在同一个全局事务中包含JDBC操作，如同其它具有XA功能的资源适配器。参见4.2.7,“事务型 JDBC™ 技术支持”。

4.8 系统管理工具

尽管没有为系统管理功能制定兼容性标准，Java EE产品供应商一般会提供工具，使系统管理员可以执行下列任务:

将事务资源管理器整合到平台

将事务资源管理器配置到平台

运行时监测事务

接收异常事务处理情况通知(例如，事务回滚次数过高)

第5章 资源，命名和注入

本章描述了应用程序怎样声明外部资源并配置参数的依赖关系，以及这些项目怎么样被体现在Java EE命名系统和怎么样被注入到应用程序组件。这些标准基于Java元数据规范(JSR-175)中定义的注解和Java命名和目录接口(JNDI)规范中定义的特征。这里提到的资源注解在公共注解规范(JSR-250)中有更详细的描述。这里提到的EJB注解在EJB规范(JSR-220)中有更详细的描 述。这里提到的PersistenceUnit和PersistenceContext注解在Java持久化规范(JSR-220)中有更详细的描述。

5.1 概述

本章定义的这些标准是为了解决下面两个问题:

应用程序组装者和部署者应该能够自定义应用程序业务逻辑的行为，而不需要了解应用程序的源代码。通常这将会涉及到参数值的说明，外部资源的连接等等。部署描述符提供了这样的功能。

应用程序必须能够在它们的运行环境中访问资源和外部信息，而不需要知道外部信息在那个环境中怎样命名和组织。JNDI命名上下文和Java语言注解提供了这样的功能。

5.1.1 本章结构

下面的章节给出Java EE平台对以上问题的解决方案:

15.2,“JNDI命名上下文”为JNDI命名上下文以及它与Java语言注解的交互定义了通用规则，Java语言注解会引用命名上下文的入口。

15.3,“Java EE平台角色的职责”为每个Java EE平台角色定义了公共的职责，这些角色包括应用程序组件供应商，应用程序组装者，部署者和Java EE产品供应商等等。

15.4,“简单环境入口”定义了指定和访问应用程序组件命名环境的基本接口。此节阐述了应用程序组件命名环境的使用，以自定义应用程序组件的业务逻辑。

15.5,“企业级JavaBeans™(EJB)的引用”为获取主接口或企业Bean实例定义了接口。这用到了EJB的引用，它是应用程序组件环境中的特定入口。

15.6,“Web服务的引用”为Web服务的引用制定了规范。

15.7,“资源管理器连接工厂的引用”为获取资源管理器连接工厂定义了接口。这用到了资源管理器连接工厂的引用，它是应用程序组件环境中的特定入口。

15.8,“资源环境的引用”为获取与资源关联的受管理对象定义了接口。这用到了资源环境的引用，它是应用程序组件环境中的特定入口。

15.9,“消息目的地的引用”为声明和使用消息目的地的引用定义了接口。

15.10,“用户事务的引用”描述了特定应用程序组件引用组件环境中的UserTransaction对象来启动，提交和中止事务。

15.11,“事务同步注册表的引用”描述了特定应用程序组件引用组件环境中的TransactionSynchronizationRegistry 对象。

15.12,“ORB的引用”描述了特定应用程序组件引用组件环境中的CORBA ORB对象。

15.13,“持久化单元的引用”描述了特定应用程序组件引用组件环境中的EntityManagerFactory 对象。

15.14,“持久化上下文的引用”描述了特定应用程序组件引用组件环境中的EntityManager对象。

5.1.2 必须提供对JNDI命名环境的访问

Java EE应用程序客户端，企业Bean和Web组件必须能够访问到JNDI命名环境。这些组件类型的容器必须支持这里描述的命名环境。

注解和部署描述符是将访问信息传达给应用程序组装者和部署者的主要载体，其内容涉及自定义业务逻辑和访问外部信息的应用程序组件标准。这里的注解对 所有应用程序组件类型都是可用的。这里的部署描述符入口在每个应用程序组件类型的部署描述符Schema中展现了相同的形式。详细信息请查看每个应用程序 组件类型的相关规范。

5.2 JNDI命名上下文环境

应用程序组件的命名环境提供了一种机制，它允许在部署或组装期间自定义应用程序组件业务逻辑的。应用程序组件环境允许自定义应用程序组件，而不需要了解或改变应用程序组件的源代码。

5.2.1 应用程序组件的环境

容器实现了应用程序组件的环境，并且将它作为JNDI命名上下文提供给应用程序组件的实例。应用程序组件的环境被用于下列情况:

1.应用程序组件的业务方法使用来自环境的入口。通过JNDI接口或特定组件上下文对象的查找方法，这些业务方法可以访问环境。此外，来自环境的入 口可以被注入到应用程序组件的字段或方法。应用程序组件供应商在部署描述符或注解中声明所有的环境入口，应用程序期望在运行时从它的环境中得到这些入口。 应用程序组件供应商也能在部署描述符或注解中为每个环境入口指定其它环境入口的JNDI名称。这些名称用来初始化环境入口，定义在“lookup”功能中。

2.容器提供了对JNDI命名上下文的实现，用来存储应用程序组件的环境。容器也提供相应的工具，使部署者可以创建和管理每个应用程序组件的环境。

3.部署者使用容器提供的工具来初始化环境入口，这些入口声明在应用程序组件的部署描述符或注解中。部署者可以设置和修改环境入口的值。作为这个流程的一部分，部署者可以重写任何与应用程序组件关联的“lookup”信息。

4.按照应用程序组件的部署描述符或应用程序组件类中的注解，容器将来自环境的入口注入到应用程序组件的字段或方法。

5.容器也使环境命名上下文在运行时对应用程序组件实例可用。应用程序组件的实例可以使用JNDI接口或组件上下文的查找方法来获取环境入口的值。

5.2.2 应用程序组件环境的命名空间

应用查询组件的命名环境由4个逻辑命名空间组成，代表了不同域的命名环境。这4个命名空间是:

> java:comp - 这个命名空间中的名称代表每个组件(例如，每个企业Bean)。除了Web模块中的组件以外，每个组件都有它自己的java:comp命名空间，不与其它任何组件共享。Web模块中的组件没有它们自己私有的组件命名空间。请看后面提到的注意事项。
>
> java:module - 这个命名空间中的名称被一个模块的所有组件共享(例如，单个EJB模块中的所有企业Bean，或一个Web模块中的所有组件)。
>
> java:app - 这个命名空间中的名称被单个应用程序中的所有模块的所有组件共享，“单个应用程序”意思是一个单独的部署单元，比如单个.ear文件，单个单机部署的模块 等等。例如，在同一个.ear文件中的一个.war文件和一个EJB的.jar文件都能访问java:app命名空间中的资源。
>
> java:global - 这个命名空间中的名称被部署在一个应用程序服务器实例中的所有应用程序共享。注意，一个应用程序服务器实例可以是单个服务器，一组服务器，或一个包含很多 服务器的管理域，甚至更多。一个应用程序服务器实例的域是依赖于产品的，但是它必须能够部署多个应用程序。

由于历史原因，java:comp 命名空间被一个Web模块中的所有组件共享。为了维持兼容性，本规范没有改变它。在一个Web模块中，java:comp的作用与命名空间 java:module相同。推荐将Web模块中打算为多个的组件共享的资源声明在java:module/env命名空间中。

注意，应用程序客户端是仅有一个组件的模块。

还需要注意的是，资源适配器(连接器)模块不可以将资源定义在任何组件命名空间中，但是可以查找其它组件定义的资源。资源适配器中可以访问的所有java:命名空间是调用资源适配器的组件的命名空间(在组件的上下文中调用时)。

如果多个应用程序组件在一个共享的命名空间中声明了一个环境入口，这个入口的所有属性必须在各自的声明中保持一致。例如，如果多个组件用同一个java:app名称声明了一个资源的引用，那么authentication和shareable属性必须相同。

如果一个共享的环境入口的每项声明的所有属性不一致，那就必须作为一个部署错误报告给部署者。部署工具可以让部署者纠正错误并继续部署。

默认情况下，应用程序组件声明的环境入口创建在java:comp/env命名空间中。环境入口可以声明在任何一个定义的命名空间中，只需显式地在 名称前加上空间名称作为前缀。推荐将环境入口创建在相应的命名上下文的env子环境中。例如，一个模块中共享的入口应该声明在 java:module/env上下文中。注意，不在env子环境下命名可能会引发冲突，这涉及到本规范的当前或未来版本，服务器定义的名称(比如应用程 序或模块的名称)，服务器定义的资源。任何命名空间的env子环境中的名称只能由应用程序的显示声明或管理员的明确行为创建;应用程序服务器不能在任何命名空间的env子环境中预定义任何名称，也包括这种env子环境的子环境。

Java EE产品可以在共享的命名空间中对资源的访问强制实施安全约束。然而，必须能够部署某种应用程序，它在共享的命名空间中定义了一些资源，而这个命名空间对 给定域中不同入口都是可用的。例如，必须能够部署一个应用程序，它在java:global命名空间中使用元数据声明的多种形式定义了一个资源，此空间对 分离的应用程序是可用的。

5.2.3 环境入口类型的可访问性

定义在环境入口中的所有对象必须指定一种组件可以访问的Java类型，这些入口可以是任何形式(部署描述符或注解)。8.3,“类加载标准”中指定 了Java类的可访问性。如果这个对象的类型是java.lang.Class，这个Class对象必须引用一个组件可以访问的类。注意，某些情况下，容 器可能会返回一个请求类型的子类型的实现，组件不能访问这个子类型的实现。

5.2.4 环境入口的共享

每个应用程序组件都定义了自己的依赖关系的集合，它必须作为应用程序组件环境的入口出现。应用程序组件在同一个容器中的所有实例共享相同的环境入口。不允许应用程序组件实例在运行时修改环境。

一般而言，无论什么时候，在JNDI java:命名空间中查找对象都必须返回一个请求对象的新的实例。允许出现下面的例外情况:

> 容器知道对象是不可变的(例如，java.lang.String类型的对象)，或知道应用程序不能改变对象的状态。
>
> 这个对象被定义为单例，这样在JVM中只存在这个对象的一个实例。
>
> 定义这个用于查找的名称是为了返回这个可以共享的对象的实例。java:comp/ORB就是这样一个名称。

在这些情况下，可以返回一个共享的对象实例。在所有其它情况下，请求对象新的实例必须通过这种查找返回。注意，如果是资源适配器连接对象，它是资源适配器的ManagedConnectionFactory 的实现，有责任满足此标准。

一个对象的每次注入就相当于一个JNDI查询。无论是注入请求对象的新实例，还是注入一个共享实例，都由上面描述的规则决定。

5.2.5 注解和注入

正如下面章节要描述的，某些受容器管理的组件类，可以为它的字段或方法加上注解，来请求注入应用程序组件环境的入口。不同容器各自的规范指出了哪些类被认为是受容器管理的; 并不是所有给定的类都需要受容器管理。

本章描述的任何资源类型都可以被注入。使用部署描述符中的入口也可以请求注入，但必须与每个资源类型一致。字段或方法可以加上任何访问修饰符(public，private等等)。除了应用程序客户端带有main方法的主类，所有类的字段或方法不能是静态的。因为应用程序客户端的生命周期与 Java SE程序相同，应用程序客户端容器不会创建应用程序客户端主类的实例。相反，静态的main方法会被调用。在这种情况下，为了支持应用程序客户端主类的注 入，注解的字段或方法必须是静态的。

类的资源可以作为注入的目标。字段不可以是final的。默认情况下，字段的名称结合了类的全限定名，并且直接用在应用程序组件的命名上下文中。例 如，包com.example中的类MyApp，它有个字段名为myDatabase，与之相符的JNDI名称是java:comp/env/ com.example.MyApp/myDatabase。注解也允许显式地指定JNDI名称。当使用部署描述符入口来指定注入时，JNDI名称和字段 名称都要被显式地指定。注意，JNDI名称默认使用java:comp/env 命名上下文。

让方法遵循JavaBean属性命名约定，也可以将环境入口注入到类中。在这里，注解用来为方法设置属性，调用这个方法就可以将环境入口注入到类中。JavaBean属性名称(不是方法名称)被用作默认的JNDI名称。例如，同一个类MyApp中有个名为setMyDatabase的方法，与之相 符的JNDI名称也是java:comp/env/com.example.MyApp/myDatabase。

每个资源只可以注入到给定类和名称的单个字段或方法中。请求将资源java:comp/env/ com.example.MyApp/myDatabase 同时注入到setMyDatabase方法和myDatabase 字段是错误的。注意，尽管如此，字段或方法仍然可以请求注入不同名称(非默认)的资源。通过显示地指定资源的JNDI名称，单个资源可以注入到多个类的多 个字段或方法中。

每种组件类型各自的规范描述了哪些类可以使用注解来注入，表5-1对此进行了总结。它们也描述了组件生命周期中出现注入的情况。注入通常出现在类的实例被创建之后，任何业务方法被调用之前。如果容器未能找到需要注入的资源，类的初始化必然中止，并且这个类不能放入服务中。

相应规范	支持注入的类	支持PostConstruct?	支持PreDestroy?
Servlet	Servlet	Yes	Yes
	Servlet过滤器	Yes	Yes
	事件监听器	Yes	Yes
JSP	标签处理器	Yes	Yes
	标签库事件监听器	Yes	Yes
JSF	域管理的Bean	Yes	Yes
JAX-WS	服务终端	Yes	Yes
	处理器	Yes	Yes
EJB	Bean	Yes	Yes
	拦截器	Yes	Yes

管理的Bean	管理的Bean	Yes	Yes
CDI	CDI风格管理的Beana或装饰者	Yes	Yes
Java EE平台	主类(静态)	Yes	No
	登录回调处理器	Yes	Yes

a.我们使用这个术语是针对CDI规范(JSR-299)的每项规则下的受管理的Bean。这样就区别于使用@ManagedBean注解声明的受管理的Bean以及EJB会话Bean，这两种Bean即使在没有CDI的情况下也是属于受管理的Bean。

注解也可以应用于类自身。这些注解在应用程序组件环境中声明了一个入口，但是没有产生资源注入，而是期望应用程序组件使用JNDI或组件上下文的lookup方法来查找入口。当类加上注解时，必须显式地指定JNDI名称和环境入口类型。

Resource注解可以出现在以上列出的任何类及其超类中。继承关系上的任何类的资源注解定义了应用程序组件所需的资源。然而，资源注入遵循 Java语言重写规则，以呈现字段和方法的可见性。如果一个方法重写了超类的方法，它定义的资源会注入到该方法。一个重写的方法可以请求注入，即使超类方 法没有这样的请求。它也可以请求注入与超类不同的资源，或者拒绝注入，即使超类方法请求了注入。

另外，不可见或被子类隐藏(不同于重写)的字段或方法仍然可以请求注入。例如，这允许一个private字段成为注入的目标，并且用在超类的实现 中，即使子类看不见这个字段并且不知道那个超类的实现正在使用一个注入的资源。注意，子类中定义的字段如果与超类相同，就会隐藏超类中的那个字段。

在某些情况下，一个类需要在所有资源注入后执行它自己的初始化方法。为了支持这种情况，可以为这个类的某个方法加上PostConstruct注解 (或者使用部署描述符中post-construct 入口来指定)。这个方法会在所有注入发生后，将这个类放入服务以前被调用。即使这个类没有请求注入任何资源，这个方法也会被调用。生命周期由容器管理的类 也与此类似，PreDestroy 注解(或者使用部署描述符中pre-destroy 入口来指定)可以应用在一个方法上，当这个类脱离服务并且不再被容器使用时，此方法就会被调用。类继承体系中的每个类都可以有PostConstruct 和PreDestroy 方法。方法调用的次序与类继承体系的次序一致，超类方法先于子类方法执行。

PostConstruct和PreDestroy 注解由公共注解规范确定。所有支持注入的类也支持PostConstruct 注解。对象生命周期完全受容器管理的所有类都支持PreDestroy 注解。

请注意，对CDI的支持必须能够在模块级别上激活，或者精确到存档级别(参见CDI规范的12.1节)。只有用存档激活CDI时属于Bean存档， 例如，包含一个META-INF/Beans.xml 描述符。CDI风格管理的Bean或装饰者只能在作为Bean存档的一部分时存在。这样，它就只在一个Bean存档中，这个存档支持所有能预料的组件类型 的资源注入。对使用@ManagedBean注解的受管理的Bean的支持与此大不相同。这样的Bean必须被所有模块类型无条件地支持，作为它们的资源 注入。

5.2.6 注解和部署描述符

环境入口可以用注解声明，而无需任何部署描述符入口。环境入口也可以由部署描述符入口声明。同一个入口可以同时用注解和部署描述符入口声明。在这种 情况下，部署描述符入口的信息可以用来重写注解提供的一些信息。应用程

序组装者或部署者可以通过这种途径来重写应用程序组件供应商提供的信息。应用程序不 应该使用部署描述符入口来请求将资源注入到没有设计注入的字段或方法中。

下面列出了部署描述符入口覆盖资源注解的规则:

基于带有注解的JNDI名称，可以定位有意义的部署描述符入口 (默认或显示提供)。

部署描述符中指定的类型必须匹配字段或属性的类型。

如果指定了description，那么它会覆盖注解的description元素。

如果指定了注入目标，必须准确地命名带有注解的字段或属性方法。

如果指定了res-sharing-scope元素，那么它就会覆盖注解的shareable元素。一般而言，应用程序组装者或部署者不应该改变这个值，这样做很可能会破坏应用程序。

如果指定了res-auth元素，那么它就会覆盖注解的authenticationType元素。一般而言，应用程序组装者或部署者不应该改变这个值，这样做很可能会破坏应用程序。

如果指定了lookup-name元素，那么它就会覆盖注解的lookup元素。

部署描述符入口覆盖EJB注解的规则包含在EJB规范中。部署描述符入口覆盖WebServiceRef 注解的规则包含在Java EE Web服务规范中。

PostConstruct方法可以用PostConstruct注解或post-construct 部署描述符入口指定。类似地，PreDestroy方法也可以用PreDestroy注解或pre-destroy部署描述符入口指定。

5.2.7 其它命名上下文入口

除了应用程序组件声明的环境入口以外，命名上下文中还会出现其它入口，正如本规范或其它规范所有指定的。下面是其中的一些入口。这不是一个详细的列表; 详细信息请查看相应的规范。

一个应用程序中所有企业Bean在共享的命名空间中被给定了入口。详细信息参见EJB规范。

所有Web应用程序在共享的命名空间中被给定了名称。这个名称相当于Web应用程序完整的URL。详细信息参见Servlet规范。

代表各自容器服务的对象定义在java:comp命名空间中。参见5.10,“用户事务的引用”，5.11,“引用事务同步注册表”以及5.12,“引用ORB”。

提供当前模块名称和应用程序名称的字符串定义在java:comp命名空间中。参见5.15,“应用程序名称和模块名称的引用”。

5.3 Java EE平台角色的职责

本节描述了每个Java EE平台角色的职责，这些角色出现在Java EE命名上下文的使用中。下面的小节描述了存储在命名上下文中的不同类型对象的特定职责。

5.3.1 应用程序组件供应商的职责

应用程序组件供应商可以使用3种技术来访问和管理命名上下文环境。第一，应用程序组件供应商可以使用Java语言注解来请求注入命名上下文的资源， 或者声明命名上下文需要的元素。第二，组件可以使用JNDI API来访问命名上下文

中的入口。第三，部署描述符入口可以用来声明命名上下文需要的入口，以及请求将这些入口注入到应用程序组件中。部署描述符入口也可 以用来覆盖注解提供的信息。

作为命名上下文元素声明的一部分，应用程序组件供应商能够指定资源的JNDI名称来初始化声明的元素，这些名称可以在命名上下文中找到。这类JNDI名称可以属于任何命名空间，它们构成了应用程序组件的环境。

为确保能正确地访问到容器实现的javax.naming.InitialContext，一个可移植的应用程序组件不能指定 java.naming.factory.initial属性，不能为“java”scheme-id指定URLContextFactory，并且不能 调用javax.naming.spi.NamingManager.setInitialContextFactoryBuilder方法。

5.3.2 应用程序组装者的职责

应用程序组装者可以修改应用程序组件供应商预先设置的命名上下文的入口，也可以设定那些应用程序组件供应商没有指定任何初值的入口的值。应用程序组装者可以使用部署描述符来覆盖应用程序组件供应商在源代码注解中的设置。

5.3.3 部署者的职责

部署者必须确保应用程序组件声明的所有入口都被创建并正确地初始化。

部署者可以修改之前被应用程序组件供应商和(或)应用程序组装者设置的入口，并且必须设定那些还没有指定值却必须指定的入口值。如果一个入口包含了lookup-name元素，部署者应该将它绑定到作为查找目标的入口上。

应用程序组件供应商和应用程序组装者提供的部署描述符元素和注解元素的说明可以帮助部署者完成这个任务。

5.3.4 Java EE产品供应商的职责

Java EE产品供应商有以下职责:

提供一个部署工具，使部署者可以设置和修改应用程序组件命名上下文的入口。

实现java:comp, java:module, java:app和java:global环境的命名上下文，并且在运行时将它们提供给应用程序组件的实例。命名上下文必须包含应用程序组件供应商声明的 所有入口，并在部署描述符中设定具体的值，或者由部署者设置。如果应用程序组件需要，环境的命名上下文必须允许部署者创建子环境。命名上下文的某个入口可 能必须用其它入口的值进行初始化，特别是用到“lookup”功能时。在这种情况下，如果入口之间存在任何循环依赖，就会出现错误。类似地，如果查找指定的JNDI名称发现一个资源的类型与创建的入口不兼容，也会出现错误。部署工具可以允许部署者纠正其中一个错误的类并继续部署。

必须保证，在缺少应用程序指定的任何属性时，javax.naming.InitialContext的实现也能够满足本规范描述的标准。

将命名环境的入口注入到应用程序组件中，正如部署描述符或应用程序组件类的注解所指定的。

容器必须确保应用程序组件实例只能以只读方式访问它们的命名上下文。容器必须从javax.naming.Context接口的所有方法中抛出javax.naming.OperationNotSupportedException，如果这个接口修改了环境的命名上下文和它的子环境。

5.4 简单环境入口

简单环境入口是一个用来自定义应用程序组件业务逻辑的配置参数。这个环境入口的值可以是以下Java类型中的一个: String, Character, Byte, Short, Integer, Long, Boolean, Double, Float, Class, 及Enum的任何子类。

下面的小节描述了每个Java EE角色在这里的职责。

5.4.1 应用程序组件供应商的职责

本节描述了应用程序组件供应商关于应用程序组件环境的概览，并定义了他(她)的职责。下面分了三个小节来阐述，第一个小节描述了用于注入环境入口的注解，第二个小节描述了访问环境入口的API，第三个小节描述了在部署描述符中声明环境入口的句法。

5.4.1.1 简单环境入口的注入

应用程序组件的字段或方法可以加上Resource注解。前面已经描述了环境入口的名称和类型。注意，容器会解开环境入口，将它与注入字段或方法的 原始类型进行匹配。不能指定Resource注解的authenticationType和shareable元素; 简单环境入口不是共享的，也不需要验证。

下面的示例代码演示了应用程序组件怎样使用注解来声明环境入口。

```
// The maximum number of tax exemptions, configured by the Deployer.
@Resource int maxExemptions;
// The minimum number of tax exemptions, configured by the 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();
...
}
```

下面的示例代码演示了环境入口怎样引用另一个环境入口来指定值，假定它们在不同的命名空间。

```
// an entry that gets its value from an application-wide entry
@Resource(lookup=” java:app/env/timeout”) int timeout;
```

5.4.1.2 访问简单环境入口的编程接口

除了基于以上方式的注入以外，应用程序组件还可以动态地访问环境入口。应用程序组件实例使用JNDI接口来定位环境的命名上下文。这个实例使用无参 的构造方法创建javax.naming.InitialContext对象，并且通过InitialContext对象在java:comp/env名 称下的查找命名环境。应用程序组件的环境入口直接保存在环境的命名上下文中，或者在它直接或间接的子环境中。

环境入口拥有应用程序组件供应商在部署描述符中声明的Java编程语言类型。下面的示例代码演示了应用程序组件怎样访问它的环境入口。

```
public void setTaxInfo(int numberOfExemptions,...)
throws InvalidNumberOfExemptionsException {
...
// Obtain the application component’ 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 max = (Integer)myEnv.lookup(“maxExemptions”);
// Obtain the minimum number of tax exemptions
// configured by the Deployer.
Integer min = (Integer)myEnv.lookup(“minExemptions”);
// Use the environment entries to
// customize business logic.
if (numberOfExemptions > max.intValue() || numberOfExemptions <
min.intValue())
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 application component 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”);
...
}
```

5.4.1.3 简单环境入口的声明

应用程序组件供应商必须声明所有的环境入口，使应用程序组件代码可以对其进行访问。可以使用应用程序组件代码中的注解声明环境入口，也可以使用部署 描述符中的env-entry 元素。每个env-entry 元素只对应一个环境入口。env-entry元素的组成有：一个环境入口的可选描述；环境入口名称，默认属于java:comp/env上下文环境；环境入口值的类型，它属于预期的Java编程语言类型(这个对象类型由JNDI的lookup方法返回)；以及一个可选的环境入口值。

环境入口的范围限于声明中包含了相应的env-entry元素的应用程序组件。这表示在运行时其它应用程序组件不能访问这个环境入口，并且应用程序组件可以用相同的env-entry-name定义env-entry元素，而不会发生命名冲突。

如果应用程序组件供应商使用env-entry-value元素为环境入口提供了一个值，应用程序组装者或部署者随后可以修改这个值。这个值必须是一个字符串，它使用在仅有一个String类型参数的构造方法中，它也可以是Character类型的单个字符。

下面的例子是环境入口的声明，使用这个入口的应用程序组件的代码已经在前面进行了演示。

```
...
<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>

<env-entry>

<env-entry-name>helperClass</env-entry-name> <env-entry-type>java.lang.Class</env-entry-type>

<env-entry-value>com.acme.helper.Helper</env-entry-value>

</env-entry>

<env-entry>

<env-entry-name>timeUnit</env-entry-name> <env-entry-type>java.util.concurrent.TimeUnit</env-entry-type>

<env-entry-value>NANOSECONDS</env-entry-value>

</env-entry>

<env-entry>

<env-entry-name>bar</env-entry-name>

<env-entry-type>java.lang.Integer</env-entry-type> <lookup-name>java:app/env/appBar</lookup-name>

</env-entry>

...

环境入口的注入也可以用部署描述符指定，而不需要Java语言注解。下面的例子是环境入口的声明，它对应于先前的注入示例。

...

<env-entry>

<description>

The maximum number of tax exemptions allowed to be set.

</description>

<env-entry-name>

com.example.PayrollService/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.example.PayrollService

</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.example.PayrollService/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.example.PayrollService

</injection-target-class>

<injection-target-name>

minExemptions

</injection-target-name>

</injection-target>

</env-entry>

将字段或方法声明为注入目标常常是很方便的，但也也已在代码中指定默认值，正如下面这个示例所演示的。

```
// The maximum number of tax exemptions, configured by the Deployer.
@Resource int maxExemptions = 4;        // defaults to 4
```

如果部署者指定的值覆盖了代码中的默认值，针对这种情况，容器必须且只能为这个资源注入一个值。当指定了注入目标时，部署描述符中的env- entry-value元素是可选的。如果没有指定这个元素，将没有值被注入。另外，如果没有指定这个元素，这个已命名的资源将不会在命名上下文中初始 化; 显式查找这个已命名的资源将会失败。

与@Resource注解的lookup元素对应的部署描述符元素是lookup-name。下面的部署描述符片段等价于之前使用lookup元素的例子。

<env-entry>

<env-entry-name>

somePackage.SomeClass/timeout

</env-entry-name>

<env-entry-type>java.lang.Integer</env-entry-type>

<injection-target>

<injection-target-class>

somePackage.SomeClass

</injection-target-class>

<injection-target-name>timeout</injection-target-name>

</injection-target>

<lookup-name>java:app/env/timeout</lookup-name>

</env-entry>

同时给一个env-entry元素指定env-entry-value和lookup-name元素是错误的。如果这两个元素有一个存在，那么相应 的@Resource注解的lookup元素最终会被忽略。换句话说，部署描述符环境入口分配的值，直接(env-entry-value)或间接 (lookup-name)地覆盖了注解分配的值。

5.5 Enterprise JavaBeans™ (EJB)的引用

本节描述了编程和部署描述符接口，它们允许应用程序组件供应商使用企业Bean引用的“逻辑”名称来引出企业Bean的home或实例。EJB的引用是应用程序组件命名环境中的特定入口。部署者在可选的目标环境中将EJB的引用绑定到企业Bean的home或实例上。

部署描述符也允许应用程序组装者在同一个Java EE应用程序中，将声明在应用程序组件中的EJB的引用链接到ejb-jar文件中的企业Bean。这个链接也是给部署工具的说明，它描述将EJB资源绑 定到指定的目标企业Bean的home上。同样，这个链接也可以由应用程序组件供应商在组件源代码中使用注解来指定。

本节中的标准仅适用于包含了EJB容器的Java EE产品。

5.5.1 应用程序组件供应商的职责

本小节描述了应用程序组件供应商关于EJB引用的概览和职责。下面分三个小节来阐述，第一个小节描述了用于注入EJB引用的注解，第二个小节描述了访问EJB引用的API，然后第三个小节描述了在部署描述符中声明EJB引用的句法。

5.5.1.1 EJB入口的注入

应用程序组件的字段或方法可以加上EJB注解。EJB注解相当于EJB会话Bean的引用。这个引用可以指向会话Bean的本地或远程home接 口，也可以指向EJB 3 Bean的业务接口。如果这个引用指向EJB 3的业务接口，那么将会注入一个企业Bean实例的引用。

下面的例子演示了应用程序组件怎样使用EJB注解来引用一个企业Bean的实例。这个被引用的Bean是一个有状态会话Bean。这个企业Bean 的引用在命名上下文中的名称将是java:comp/env/com.example.MyApp/myCart。由于没有为这个引用的目标命名，因此必 须由部署者决定。

@EJB private ShoppingCart myCart;

下面的例子演示了差不多所有EJB注解元素的使用。

@EJB(

name = “ejb/shopping-cart”,

BeanName = “cart1”,

```
BeanInterface = ShoppingCart.class,
description = “The shopping cart for this application”
)
private Cart myCart;
```

元素BeanName可以有两种选择，EJB的引用可以使用全局JNDI名称，或者EJB规范授权的任何其它名称，通过注解的lookup元素的形式。下面的例子使用了应用程序命名空间中的JNDI名称。

```
@EJB(
lookup=”java:app/cartModule/ShoppingCart”,
description = “The shopping cart for this application”
)
private Cart myOtherCart;
```

5.5.1.2 EJB引用的编程接口

应用程序组件供应商可以按下述方式使用EJB的引用来定位企业Bean的home接口或实例。

l在应用程序组件环境中为这个引用分配一个入口(参见5.5.1.3小节了解怎样在部署描述符中声明EJB的引用)。

l本规范推荐但不要求将企业Bean的引用组织到应用程序组件环境的ejb子环境中 (也就是java:comp/env/ejb JNDI上下文环境)。注意，默认情况下，用注解声明的企业Bean的引用将不会在任何子环境中。

l在应用程序组件环境中，使用JNDI查找被引用的企业Bean的home接口或实例。

下面的例子演示了应用程序组件怎样使用EJB的引用来定位企业Bean的home接口。

```
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，应用程序组件供应商将环境入口ejb/EmplRecord 指定为EJB引用的名称来引出了企业Bean的home接口。

5.5.1.3 EJB引用的声明

虽然EJB的引用是应用程序组件环境的入口，但是应用程序组件供应商不能使用env-entry元素来声明它，而是必须使用应用程序组件代码中的注 解或者部署描述符的ejb-ref或ejb-local-ref元素。这让应用程序组件JAR文件的用户(应用程序组装者或部署者)可以发现应用程序组件 使用的所有EJB的引用。部署描述符入口也可以用来将EJB的引用注入到应用程序组件中。

每个ejb-ref或ejb-local-ref元素描述了应用程序组件引用企业Bean的接口标准。这个ejb-ref元素中包含了description, ejb-ref-name, ejb-ref-type, home以及remote元素。

元素ejb-ref-name指定了EJB引用的名称。它的值是环境入口的名称，用在应用程序组件代码中。ejb-ref-type元素指定了企业 Bean期望的类型，它的值必须是Entity或者Session。home 和remote 元素指定了被引用的企业Bean的home和remote接口所期望的Java编程语言类型。

EJB引用的范围限定于声明中包含了相应的ejb-ref或ejb-local-ref元素的应用程序组件。这表示在运行时其它应用程序组件不能访 问这个EJB引用，应用程序组件可以用相同的ejb-ref-name定义ejb-ref或ejb-local-ref元素，而不会发生命名冲突。

元素lookup-name指定了环境入口的JNDI名称，这个入口提供了这个引用的值。

下面这个例子演示了部署描述符中EJB引用的声明。

```
...
<ejb-ref>
<description>
This is a reference to the 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-ref>
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>com.aardvark.payroll.PayrollHome</home>
<remote>com.aardvark.payroll.Payroll</remote>
</ejb-ref>
<ejb-ref>
<ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>com.wombat.empl.PensionPlanHome</home>
<remote>com.wombat.empl.PensionPlan</remote>
<lookup-name>
```

```
java:global/personnel/retirement/PensionPlan
</lookup-name>
</ejb-ref>
...
```

5.5.2 应用程序组装者的职责

应用程序组装者可以使用部署描述符的ejb-link元素来链接到指向目标企业Bean的EJB引用。

应用程序组装者按如下方式链接到企业Bean:

使用引用方应用程序组件的ejb-ref或ejb-local-ref元素下可选的ejb-link元素。ejb-link元素的值是目标企 业Bean的名称(它是目标企业Bean的ejb-name元素中定义的名称)。目标企业Bean与引用方应用程序组件在同一个Java EE应用程序中，可以是其中的任何ejb-jar文件。

再者，为避免重命名企业Bean，使它们在一个完整的Java EE应用程序中有唯一的名称。应用程序组装者可以在引用方应用程序组件的ejb-link元素中使用下面的句法。应用程序组装者指定包含了被引用的企业 Bean的ejb-jar文件的路径名称，并追加目标企业Bean的ejb-name，用“#”号与路径名称分隔。这个路径名称就关联到引用的企业 Bean组件的JAR文件。当应用程序组装者不能更换ejb-name时，以这种方式，就可以唯一地标识出有相同ejb-name的多个Bean。

除了ejb-link元素, 应用程序组装者还可以使用lookup-name元素来引用目标EJB组件，这需要用到这个组件的一个JNDI名称。注意，ejb-link和lookup-name不能同时出现在ejb-ref元素中。

应用程序组装者必须确保目标企业Bean与声明的EJB引用是类型兼容的。这表示，目标企业Bean必须是ejb-ref-type元素中标明的类型, 并且它的home接口和remote接口必须与EJB引用中声明的接口是Java类型兼容的。

下面的例子演示了部署描述符中ejb-link元素的使用。企业Bean的引用应该由名为EmployeeRecord的Bean确定。 EmployeeRecord企业Bean可以包入到创建这个引用的组件所在的模块中，也可以包入到同一个Java EE应用程序的另一个模块中。

```
...
<ejb-ref>
<description>
This is a reference to the entity Bean that
encapsulates access to employee records. It
has been linked to the entity Bean named
EmployeeRecord in this application.
</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-link>EmployeeRecord</ejb-link>

</ejb-ref>

...

下面的例子演示了使用ejb-link元素来标明指向ProductEJB企业Bean的引用，它们在同一个Java EE应用程序单元中不同的ejb-jar文件中。

...

<ejb-ref>

<description>

This is a reference to the entity Bean that

encapsulates access to a product. It

has been linked to the entity Bean named

ProductEJB in the product.jar file in this

application.

</description>

<ejb-ref-name>ejb/Product</ejb-ref-name>

<ejb-ref-type>Entity</ejb-ref-type>

<home>com.acme.products.ProductHome</home>

<remote>com.acme.products.Product</remote> <ejb-link>../products/product.jar#ProductEJB</ejb-link>

</ejb-ref>

...

下面的例子演示了使用ejb-link元素来标明指向ShoppingCart企业Bean的引用，它们在同一个Java EE应用程序单元中不同的ejb-jar文件中。这个引用已经声明在应用程序组件代码的注解中。组装者只提供Bean的链接。

...

<ejb-ref>

<ejb-ref-name>ShoppingService/myCart</ejb-ref-name> <ejb-link>../products/product.jar#ShoppingCart</ejb-link>

</ejb-ref>

...

使用lookup-name元素并为目标Bean指定恰当的JNDI名称，可以获得相同的效果。

...

<ejb-ref>

<ejb-ref-name>ShoppingService/myCart</ejb-ref-name> <lookup-name>java:app/products/ShoppingCart</lookup-name>

</ejb-ref>

...

5.5.3 部署者的职责

部署者有以下职责:

部署者必须确保所有已声明的EJB引用都被绑定到存在于运行环境中的企业Bean的home或实例上。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接到目标企业Bean实际的JNDI名称。

部署者必须确保目标企业Bean与EJB引用声明的类型是兼容的。这表示，目标企业Bean必须是ejb-ref-type元素或EJB注解 标明的类型，并且目标企业Bean的home和remote接口必须与EJB引用(如果指定了)声明的home和remote接口是Java类型兼容的。

如果EJB引用的声明中包含了ejb-link元素，部署者应该将企业Bean的引用绑定到目标企业Bean上。如果企业Bean的引用包含 了lookup-name元素，部署者应该将企业Bean的引用绑定到作为查找目标的企业Bean上。注意，ejb-link和lookup-name元 素不能同时出现在EJB引用的声明中。

5.5.4 Java EE产品供应商的职责

Java EE产品供应商必须提供部署工具，使部署者可以执行前面小节中描述的任务。这些部署工具必须能够处理类文件注解和部署描述符的ejb-ref元素提供的信息。

最低限度，这些工具必须能够:

维护应用程序注解或ejb-link元素中的组装信息，它们绑定了指向目标企业Bean的home接口或实例的EJB引用。

通告部署者存在任何未确定的EJB引用，并允许他(她)将这个EJB引用绑定到指定的兼容的目标企业Bean上。

5.6 Web服务的引用

Web服务的引用类似于EJB的引用，用它来引用Web服务。Web服务的引用完全由Web服务规范和JAX-WS规范指定。

5.7 资源管理器连接工厂的引用

资源管理器连接工厂是一个用来创建资源管理器连接的对象。例如，一个实现了javax.sql.DataSource接口的对象就是一个资源管理器连接工厂，用它创建连接数据库管理系统的java.sql.Connection对象。

本节描述的应用程序组件的编程和部署描述符接口，使应用程序组件代码可以引用资源工厂，这需要用到叫做资源管理器连接工厂的引用，它是一个逻辑名称。资源管理器连接工厂的引用是应用程序组件环境中的特定入口。部署者将资源管理器连接工厂的引用绑定到实际的资源管理器连接工厂，它存在于目标运行环境 中。因为这些资源管理器连接工厂允许容器影响资源的管理，通过资源管理器连接工厂的引用获取的连接被称作受管理的资源(例如，这些资源管理器连接工厂允许 容器实现连接池和自动征用带事务的连接)。

通过命名环境访问的资源管理器连接工厂对象只在执行查找的组件实例中有效。请查看组件各自的规范以了解其它可用的约束。

5.7.1 应用程序组件供应商的职责

本小节概述了应用程序组件供应商怎样定位资源工厂，并定义了的他(她)的职责。下面分三个小节阐述，第一个小节描述的注解用于注入资源管理器连接工厂的引用，第二个小节描述了用于访问资源管理器连接工厂引用的API，第三个小节描述了在部署描述符中声明工厂引用的句法。

5.7.1.1 注入资源管理器连接工厂的引用

应用程序组件的字段或方法可以加上Resource 注解。上面已经描述了工厂的名称和类型。Resource注解的authenticationType和shareable元素可以用来控制资源验证的类 型，还可以控制从工厂中获取的连接的共享性，接下来会进行更详细地描述。

下面的示例代码演示了应用程序组件怎样使用注解来声明资源管理器连接工厂的引用。

```
// 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();
...
}
```

能够作为@Resource注解的一部分来指定与资源绑定的入口的JNDI名称。

```
// The customer database, looked up in the application environment.
@Resource(lookup=”java:app/env/customerDB”) javax.sql.DataSource
customerAppDB;
```

在上面的例子中被查找的数据源对象可以按下面的方式声明。

```
@Resource(name=”java:app/env/customerDB”,
type=javax.sql.DataSource.class)
public class AnApplicationClass {
...
}
```

从实际的角度来说，在应用程序级别上声明一个常用的数据源，并且使用lookup从多种组件查找它可以简化应用程序的部署工作，从现在开始部署者只 能为应用程序级的资源执行单一的绑定操作，而不是多种操作。使用数据源资源定义可以进一步简化这个工作，参见5,“数据源资源定义”。当然，不阻止部署者 为每个数据源的引用进行单独地绑定，如果有必要的话。

5.7.1.2 资源管理器连接工厂引用的编程接口

应用程序组件供应商可以按下列方式使用资源管理器连接工厂的引用来获取连接。

在应用程序组件的命名环境中为资源管理器连接工厂的引用分配一个入口(参见5.7.1.3小节了解在部署描述符中怎样声明资源连接器工厂的引用)。

本规范推荐但不强求，将所有资源管理器连接工厂的引用组织到应用程序组件环境的子环境中，这需要为每种资源管理器类型使用不同的子环

境。例如，所有JDBC™数 据源的引用应该声明在java:comp/env/jdbc子环境中，所有JMS连接工厂应该声明在java:comp/env/jms子环境中，所有 JavaMail连接工厂应该声明在java:comp/env/mail子环境中，以及所有URL连接工厂应该声明在java:comp/env /url子环境中。注意，由注解声明的资源管理器连接工厂的引用默认情况下不出现在任何子环境中。

使用JNDI接口在应用程序组件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对象的适当方法来获取资源的连接。工厂的方法对应特定的资源类型。可以多次调用工厂对象来获取多个连接。

应用程序组件供应商可以控制从资源管理器连接工厂获取的连接的共享性。默认情况下，资源管理器的连接可以被应用程序中的其它组件共享，此应用程序在 同一个事务上下文中使用了相同的资源。应用程序组件供应商可以规定从资源管理器连接工厂的引用获取的连接是非共享的，这需要设定部署描述符元素res- sharing-scope的值为非共享。共享资源管理器的连接使容器可以优化连接的使用以及进行本地事务优化。

对于主体与资源管理器的访问之间关联的处理，应用程序组件供应商有两种选择:

允许部署者设定主体映射或资源管理器的签名信息。在这种情况下，应用程序组件代码可以调用没有安全相关参数的资源管理器连接工厂的方法。

对应用程序组件代码中的资源进行签名。在这种情况下，应用程序组件可以调用恰当的资源管理器连接工厂的方法，用签名信息作为方法参数。

应用程序组件供应商使用部署描述符的res-auth元素来标明选用这两种资源验证途径中的哪一种。

我们希望大多数应用程序组件使用第一种形式(也就是让部署者为资源设定签名信息)。下面的代码样本演示了JDBC连接的获取。

```
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 resource. The security
// principal for the resource is not given, and
// therefore it will be configured by the Deployer.
java.sql.Connection con = ds.getConnection();
...
}
```

5.7.1.3 在部署描述符中声明资源管理器连接工厂的引用

虽然资源管理器连接工厂的引用是应用程序组件环境中的入口，但是应用程序组件供应商不能用env-entry元素声明它。

对于这种情况，应用程序组件供应商必须使用应用程序组件代码中的注解或部署描述符的resource-ref元素来声明所有资源管理器连接工厂的引 用。这让应用程序组件JAR文件的用户(应用程序组装者或部署者)可以发现应用程序组件使用的所有资源管理器连接工厂的引用。部署描述符入口也可以用来将 资源管理器连接工厂的引用注入到应用程序组件中。

每个resource-ref元素只对应一个资源管理器连接工厂的引用。这个resource-ref元素中包含了description元素，必 须的res-ref-name元素，可选的res-sharing?scope, res-type和res-auth元素。元素res-ref-name指定了应用程序组件代码中使用的环境入口的名称。环境入口的名称默认在 java:comp/env上下文中(例如，这个名称应该是jdbc/EmployeeAppDB ，而不是java:comp/env/jdbc/EmployeeAppDB)。元素res-type包含了应用程序组件代码所期望的资源管理器连接工厂 的Java编程语言类型。如果这个资源指定了注入目标，res-type 元素就是可选的; 在这种情况下，res-type是注入目标的默认类型。元素res-auth指明了应用程序组件代码是否执行编程式资源签名，或者容器是否对基于主体映射 信息的资源进行签名，这些信息由部署者提供。应用程序组件供应商将res-auth元素的值设为Application或Container，以指定签名 责任。如果没有指定，默认是Container。元素res-sharing-scope 指明了从给定的资源管理器连接工厂的引用获取的连接是否可以共享。元素res-sharing-scope的值应设为Shareable或 Unshareable，如果没有指定这个元素，连接被认为是可共享的。

资源管理器连接工厂的引用的范围限定于声明中包含了resource-ref元素的应用程序组件。这表示，其它应用程序组件不能在运行时访问资源管 理器连接工厂的引用。应用程序组件可以用相同的res-ref-name定义resource-ref元素，而不会发生命名冲突。

类型声明允许部署者标识资源管理器连接工厂的类型。注意，标明的类型应该是资源管理器连接工厂的Java编程语言类型，而不是连接的类型。

下面的例子是前面演示的应用程序组件使用的资源引用的声明。

```
...
<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>
```

现在对上面的例子做一些修改，将定义的资源引用链接到另一个，末尾使用了范围最广的JNDI名称。

<resource-ref>

<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>

<lookup-name>java:app/env/TheEmployeeDB</lookup-name>

</resource-ref>

5.7.1.4 资源管理器连接工厂的标准类型

应用程序组件供应商必须使用javax.sql.DataSource资源管理器连接工厂类型来获取JDBC API连接。

应用程序组件供应商必须使用javax.jms.QueueConnectionFactory，javax.jms.TopicConnectionFactory或javax.jms.ConnectionFactory来获取JMS连接。

应用程序组件供应商必须使用javax.mail.Session资源管理器连接工厂类型来获取JavaMail API连接。

应用程序组件供应商必须使用java.net.URL资源管理器连接工厂类型来获取URL连接。

推荐应用程序组件供应商在java:comp/env/jdbc子环境中命名JDBC API数据源，所有JMS连接工厂在java:comp/env/jms子环境下，所有JavaMail API 连接工厂在java:comp/env/mail子环境下，以及所有URL连接工厂在java:comp/env/url 子环境下。注意，默认情况下，用注解声明的资源管理器连接工厂的引用不出现在任何子环境中。

Java EE连接器体系允许应用程序组件使用本节描述的注解或API来获取资源对象，通过这些资源对象可以访问到额外的后台系统。

5.7.2 部署者的职责

部署者使用部署工具将资源管理器连接工厂的引用绑定到目标运行环境中配置的实际的资源工厂。

部署者必须为声明在部署描述符中的资源管理器连接工厂的引用执行下列任务:

> 将资源管理器连接工厂的引用绑定到存在于运行环境中的资源管理器连接工厂。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接到资源管理器连接工厂实际的JNDI名称。资源管理器连接工厂类型必须兼容声明在res-type元素中的类型。如 果资源管理器连接工厂的引用包含了lookup-name，部署者可以选择是否使用它执行相应的查找，或者用他自己选择的绑定进行覆盖。
>
> 提供资源管理器来打开并管理资源的任何附加的配置信息。这种配置机制是资源管理器特有的，并且超出了本规范的范围。
>
> 如果Resource注解的authenticationType元素的值是AuthenticationType.CONTAINER 或部署描述符的res-auth元素是Container，部署者有责任为资源管理器配置签名信息。这种执行方式是容器和资源管理器特有的; 它超出了本规范的范围。

例如，如果必须从组件级别的安全域和主体域将主体映射到资源管理器的安全域和主体域，部署者或系统管理员必须定义这种映射。这种映射执行在容器和资源管理器的特定方式下，它超出了本规范的范围。

5.7.3 Java EE产品供应商的职责

Java EE产品供应商有以下职责:

提供部署工具，使部署者可以执行上面小节中描述的任务。

提供本规范要求的资源管理器连接工厂类的实现。

如果应用程序组件供应商将Resource注解的authenticationType元素设为了 AuthenticationType.APPLICATION，或是将资源引用的res-auth设为了Application，容器必须允许应用程序 组件使用资源管理器的API执行显式的编程式签名。

如果应用程序组件供应商将Resource注解的shareable元素设为了false，或是将资源管理器连接工厂的引用的res-sharing-scope设为了Unshareable，容器不能试图共享从资源管理器连接工厂的引用1获取的连接。

容器必须提供工具，使部署者可以为资源管理器的引用建立资源的签名信息，这个引用的authenticationType被设为了AuthenticationType.CONTAINER，或是它的res-auth 元素被设为了Container。最低标准是，部署者必须能够为每个资源管理器连接工厂的引用指定用户名/密码信息，这些引用由应用程序组件声明，并且当调用资源管理器连接工厂来获取连接时，容器必须能够使用用户名/密码的组合进行用户验证。

虽然本规范没有要求，但是我们希望容器以某些形式支持跨应用程序服务器和资源管理器的独立的签名机制。容器允许部署者建立资源，这样主体就可以被传递(直接或通过主体映射)到资源管理器，如果应用程序需要的话。

虽然本规范没有要求，但是大多数Java EE应用程序会提供下面的特性:

一个允许系统管理员为Java EE服务器添加，删除和配置资源管理器的工具。

一种为应用程序组件池化资源的机制，否则由容器管理资源的使用。池技术对应用程序组件必须是透明的。

5.7.4 系统管理员的职责

系统管理员通常由以下职责:

在Java EE服务器环境下添加，删除和配置资源管理器。

在某些情况下，这些任务可以被部署者执行。

---

1. 从同一个资源连接器工厂的不同引用获取的连接是可以共享的

5.8 资源环境的引用

本节描述的编程和部署描述符接口，允许应用程序组件供应商引用与资源关联的受管理对象(例如，一个连接器CCI InteractionSpec 实例)，这需要用到叫做资源环境的引用的“逻辑”名称。资源环境的引用是应用程序组件环境中的特定入口。部署者将资源环境的引用绑定到目标运行环境中的受 管理对象。

5.8.1 应用程序组件供应商的职责

本小节描述了应用程序组件供应商关于资源环境的引用的概览和职责。

5.8.1.1 资源环境引用的注入

应用程序组件的字段或方法可以加上Resource注解来请求注入资源环境的引用。前面已经描述了资源环境引用的名称和类型。不能指定 Resource注解的authenticationType和shareable元素; 资源环境入口不是共享的并且无需验证。使用Resource注解来声明资源环境的引用不同于使用它来声明其它环境的引用，因为资源环境引用的类型不是其它 环境引用所使用的Java语言类型。

5.8.1.2 资源环境引用的编程接口

应用程序组件供应商可以按下述方式使用资源环境的引用来定位与资源关联的受管理对象。

在应用程序组件环境中为引用分配一个入口(参见5.8.1.3小节了解在部署描述符中怎样声明资源环境的引用)。

本规范推荐但不强求，将所有资源环境的引用按资源类型组织到恰当的组件环境的子环境中。注意，用注解声明的资源环境的引用默认情况下不出现在任何子环境中。

使用JNDI在应用程序组件环境中查找受管理对象。

5.8.1.3 在部署描述符中声明资源环境的引用

虽然资源环境的引用是应用程序组件环境中的入口，但是应用程序组件供应商不能使用env-entry元素来声明它。这种情况下，应用程序组件供应商 必须使用应用程序组件代码中的注解或部署描述符中的resource-env-ref元素来声明与资源关联的受管理对象的所有引用。这让应用程序组件 JAR文件的用户可以发现应用程序组件使用的所有资源环境的引用。部署描述符入口也可以用来将资源环境的引用注入到应用程序组件中。

每个resource-env-ref元素描述了引用的应用程序组件对受管理对象的要求。元素resource-env-ref包含的了: 可选的description和resource?env-ref-type元素以及必须的resource-env-ref-name元素。如果指定了 资源的注入目标，resource-env-ref-type元素就是可选的; 在这种情况下，resource-env-ref-type默认为注入目标的类型。

元素resource-env-ref-name指定了资源环境引用的名称。它的值是应用程序组件代码中使用的环境入口的名称。资源环境引用的名称关联到java:comp/env上下文。元素resource-env-ref-type 指定了被引用对象期望的类型。

资源环境的引用的范围限定于声明中包含了resource-env-ref元素的应用程序组件。这表示，其它应用程序组件不能在运行时访问资源环境 的引用，应用程序组件可以用相同的resource-env-ref-name定义resource-env-ref元素，而不会发生命名冲突。

资源环境的引用可以指定lookup-name来链接到JNDI名称定义的另一个引用。

5.8.2 部署者的职责

部署者有以下职责:

部署者必须确保将所有声明的资源环境的引用绑定到存在于运行环境中的受管理对象。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接到目标对象实际的JNDI名称。部署者可以覆盖包含了loop-name元素的资源环境引用的链接。

部署者必须确保目标对象与资源环境引用声明的类型是兼容的。这表示，目标对象必须是Resource注解或resource-env-ref-type元素中标明的类型。

5.8.3 Java EE产品供应商的职责

Java EE产品供应商必须提供部署工具，使部署者可以执行上面小节中描述的任务。Java EE产品供应商提供的工具必须能够处理类文件注解和部署描述符的resource-env-ref元素提供的信息。

至少，这些工具必须能够告知部署者存在任何未解决的资源环境的引用，并允许他(她)将这类引用绑定到环境中指定的兼容的目标对象。

5.9 消息目的地的引用

本节描述的编程和部署描述符接口，允许应用程序组件供应商使用叫做消息目的地的引用来引出消息目的地对象，它是一个“逻辑”名称。消息目的地的引用是应用程序组件环境中的特定入口。部署者将消息目的地的引用绑定到目标运行环境中受管理的消息目的地。

本节中的标准仅适用于支持JMS的Java EE产品。

5.9.1 应用程序组件供应商的职责

本小节描述了应用程序组件供应商关于消息目的地的引用的概览和职责。

5.9.1.1 注入消息目的地的引用

应用程序组件的字段或方法可以加上Resource注解来请求注入消息目的地的引用。前面已经描述资源环境引用的名称和类型。不能指定Resource注解的authenticationType和shareable元素; 消息目的地的引用是非共享的并且无需验证。

注意，当使用Resource注解来声明消息目的地的引用时，不能将这个引用链接到同一个消息目的地的其它引用，也不能指定消息目的地是否可以用来 生成和销毁消息。稍后描述的部署描述符入口提供了一种方式，将许多消息目的地的引用关联到单个消息目的地，也能指定每个消息目的地的引用是否可以用来生成 和(或)销毁消息，以便为应用程序指定完整的消息流。应用程序组装者可以使用这些消息目的地的链接同时链接到那些用Resource注解声明的消息目的地的引用。用Resource注解声明的消息目的地的引用被假定为既能生成消息也能进行销毁; 使用部署描述符入口可以重写这个默认设定。

下面的例子演示了应用程序组件怎样使用Resource注解来请求注入消息目的地的引用。

```
@Resource
javax.jms.Queue stockQueue;
```

下面的例子演示了怎样将一个消息目的地的引用链接到另一个，它们也许在不同的命名空间，这需要将JNDI名称作为lookup元素的值。

```
@Resource(lookup=”java:app/env/TheOrderQueue”)
javax.jms.Queue orderQueue;
```

5.9.1.2 消息目引用的编程接口

应用程序组件供应商可以使用消息目的地的引用来定位消息目的地，按如下方式进行:

在应用程序组件环境中为引用分配一个入口。(参见5.9.1.3小节了解在部署描述符中怎样声明消息目的地的引用)。

本规范推荐但不强求，将所有消息目的地的引用按资源类型组织到组件环境恰当的子环境中(例如，将消息目的地指定到java:comp/env/jms JNDI上下文)。注意，用注解声明的消息目的地的引用默认情况下不出现在任何子环境中。

使用JNDI在应用程序组件环境中查找受管理对象。

下面的例子演示了应用程序组件怎样使用消息目的地的引用来定位JMS目的地。

```
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();
// Look up the JMS StockQueue in the environment.
Object result = initCtx.lookup("java:comp/env/jms/StockQueue");
// Convert the result to the proper type.
javax.jms.Queue queue = (javax.jms.Queue)result;
```

在这个例子中，应用程序组件供应商将环境入口jms/StockQueue作为消息目的地引用的名称分配给一个JMS队列。

5.9.1.3 在部署描述符中声明消息目的地的引用

虽然消息目的地的引用是应用程序组件环境中的入口，但是应用程序组件供应商不能使用env-entry元素来声明它。这种情况中，应用程序组件供应 商应该使用应用程序组件代码中的Resource注解或部署描述符的message-destination-ref元素来声明消息目的地的所有引用。这 让应用程序组件JAR文件的用户可以发现应用程序组件使用的所有消息目的地的引用。部署描述符入口也可以用来将消息目的地的引用注入到应用程序组件中。

每个message-destination-ref元素描述了引用方应用程序对被引用的目的地的要求。元素message- destination-ref 包含了: 可选的description, message-destination-type和 message-destination-usage元素以及必须的message-destination-ref-name 元素。

元素message-destination-ref-name 指定了消息目的地引用的名称。它的值是应用程序组件代码中使用的环境入口的名称。消息目的地引用的名称默认关联到java:comp/env上下文(例 如，这个名称应该是jms/StockQueue而不是java:comp/env/jms/StockQueue)。元素message- destination-type指定了被引用的目的地期望的类型。例如，在JMS目的地情况下，它的值可能是javax.jms.Queue。如果为消 息目的地的引用指定了注入目标，message-destination-type元素就是可选的; 这种情况下message-destination-type 默认为注入目标的类型。message?destination-usage 元素指定了消息是否可以被目的地销毁和(或)生成。如果没有指定，消息将被假定为既能被销毁也能被生成。

消息目的地的引用的范围限定于声明中包含了message-destination-ref元素的应用程序组件。这表示，其它应用程序组件不能在运 行时访问消息目的地的引用。应用程序组件可以用相同的message-destination-ref-name定义message-destination-ref元素，而不会发生命名冲突。

下面的例子演示了在部署描述符中声明消息目的地的引用。

```
...
```

<message-destination-ref>

<description>

This is a reference to a JMS queue used in the

processing of Stock info

</description>

<message-destination-ref-name> jms/StockInfo </message-destination-ref-name>

<message-destination-type> javax.jms.Queue </message-destination-type>

<message-destination-usage> Produces </message-destination-usage>

</message-destination-ref>

...

5.9.2 应用程序组装者的职责

通过将消息的用户和生产者链接到企业Bean部署描述符指定的一个或多个公共逻辑目的地，应用程序组装者可以指定应用程序中的消息流。应用程序组装 者使用ejb-jar文件中的message-destination元素，message-destination-ref元素中的message- destination-link元素，以及ejb-jar的message-driven元素中message-destination-link元 素，将消息目的地的引用链接到公共逻辑目的地。

应用程序组装者按下列方式指定消息用户和生产者之间的链接:

应用程序组装者使用ejb-jar部署描述符中的message-destination元素来指定应用程序中的逻辑消息目的地。元素message-destination 中定义了一个message-destination-name元素，用来作为链接。

应用程序组装者使用生产消息的应用程序组件的message-destination-ref元素中的message- destination-link元素，将它链接到目标目的地。元素message-destination-link的值是目标目的地的名称，它定义在 message-destination元素的message-destination-name元素中。元素message-destination可 以在同一个Java EE应用程序的任意EJB模块中作为引用方组件。应用程序组装者可以使用message-destination-ref元素的message- destination-usage元素来指明引用方应用程序组件生产消息给被引用的目的地。

如果公共目的地的消息用户是一个消息驱动Bean，应用程序组装者使用message-driven元素的message- destination-link元素来引用逻辑目的地。如果应用程序组装者将消息驱动Bean链接到它的源目的地，他(她)应该使用message- driven元素的message-destination-type元素来指定期望的目的地类型。否则，应用程序组装者使用消费消息的应用程序组件的 message-destination-ref元素的message-destination-link元素来链接到公共目的地。在后一种情况中，应用 程序组装者使用message-destination-ref元素的message-destination-usage元素来指明应用程序组件从引用 方目的地消费消息。

为了避免重命名消息目的地，使它在整个Java EE应用程序中有唯一的名称，应用程序组装者可以在引用方应用程序组件的message-destination-link元素中使用下面的句法。应用程 序组装者指定包含了消息目的地的ejb-jar文件的路径名称，追加目标目的地的 message-destination-

name并用#与路径名称分隔。路径名称关联到引用方应用程序组件的JAR文件。通过这种方式，可以唯一地标识 具有相同message-destination-name的多个目的地。

链接消息目的地时，应用程序组装者必须确保目的地的用户和生产者需要相同或类型兼容的消息目的地，这些类型由消息系统决定。

5.9.3 部署者的职责

部署者有以下职责:

部署者必须确保所有声明的消息目的地的引用都被绑定到存在于运行环境中的受管理对象。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接到目标对象实际的JNDI名称。部署者可以覆盖消息目的地引用的链接，它包含了一个lookup-name元素。

部署者必须确保目标对象与消息目的地的引用声明的类型是兼容的。这表示，目标对象必须是message-destination-type元素标明的类型。

部署者必须观察应用程序组装者指定的消息目的地的链接。

5.9.4 Java EE产品供应商的职责

Java EE产品供应商必须提供部署工具，使部署者可以执行上面小节中描述的任务。这些部署工具必须能够处理部署描述符中的message-destination-ref元素提供的信息。

至少，这些工具必须能够通知部署者是否存在任何未处理的消息目的地的引用，并且允许他(她)将消息目的地的引用绑定到环境中指定的兼容的目标对象。

5.10 用户事务的引用

某些Java EE应用程序组件类型可以使用JTA接口来开启，提交和中止事务。这样的应用程序组件可以通过查找JNDI名称java:comp/ UserTransaction来找到实现了UserTransaction接口的合适对象或使用Resource注解请求注入UserTransaction对象。不能指定Resource注解的authenticationType和shareable元素。只要求容器为那些 能正确使用它的组件提供java:comp/UserTransaction名称，或注入一个 UserTransaction 对象。这种UserTransaction对象的任何引用仅在执行查找的组件实例中有效。请查看组件各自的定义获取更多的信息。

下面的例子演示了应用程序组件怎样通过注入来获取和使用UserTransaction对象。

```
@Resource UserTransaction tx;

public void updateData(...) {

...

// Start a transaction.

tx.begin();

...

// Perform transactional operations on data.

...

// Commit the transaction.

tx.commit();
```

```
...
}
```

下面的例子演示了应用程序组件怎样使用JNDI查找和使用UserTransaction对象。

```
public void updateData(...) {
...
// Obtain the default initial JNDI context.
Context initCtx = new InitialContext();
// Look up the UserTransaction object.
UserTransaction tx = (UserTransaction)initCtx.lookup(
"java:comp/UserTransaction");
// Start a transaction.
tx.begin();
...
// Perform transactional operations on data.
...
// Commit the transaction.
tx.commit();
...
}
```

UserTransaction对象的引用也可以声明在部署描述符中，使用方法与资源环境的引用相同。这样一个部署描述符入口可以用来指定UserTransaction对象的注入。

本节中的标准仅适用于支持JTA的Java EE产品。

5.10.1 应用程序组件供应商的职责

应用程序组件供应商的责任是请求UserTransaction对象的注入，这需要使用Resource注解或给UserTransaction对象定义的查找名称。

只要求某些应用程序组件类型可以访问UserTransaction 对象；请查看表 6-1和EJB规范了解详细信息。

5.10.2 Java EE产品供应商的职责

Java EE产品供应商有责任提供一个合适的UserTransaction 对象，正如本规范所有要求的。

5.11 事务同步注册表的引用

JTA TransactionSynchronizationRegistry 接口用在系统级组件上，例如持久化管理器，这个持久化管理器可以和EJB或Web应用程序组件一起打包的。这种组件可以找到一个合适的对象，这个对象实现了TransactionSynchronizationRegistry 接口，这需要查找JNDI名称java:comp/TransactionSynchronizationRegistry或使用Resource注解请求TransactionSynchronizationRegistry对象的注入。不能指定Resource 注解

的authenticationType 和 shareable 元素。只要求容器为那些能够正确使用它的组件提供java:comp/TransactionSynchronizationRegistry 名称，或注入一个TransactionSynchronizationRegistry 对象。这种TransactionSynchronizationRegistry对象的任何引用仅在执行查找的组件实例中有效。请查看组件各自的定义了 解更多的信息。

TransactionSynchronizationRegistry对象的引用也可以定义在部署描述符中，使用方法与资源环境的引用相同。这样一个部署描述符入口可以用来指定TransactionSynchronizationRegistry对象的注入。

本节中的标准仅适用于支持JTA的Java EE产品。

5.11.1 应用程序组件供应商的职责

应用程序组件供应商的责任是请求TransactionSynchronizationRegistry 对象的注入，这需要使用Resource注解或为TransactionSynchronizationRegistry对象定义的查找名称。

只要求某些应用程序组件可以访问TransactionSynchronizationRegistry对象；请查看表6-1和EJB规范了解详细信息。

5.11.2 Java EE产品供应商的职责

Java EE产品供应商的责任是提供一个合适的TransactionSynchronizationRegistry 对象，正如本规范所有要求的。

5.12 ORB的引用

一些Java EE 应用程序需要使用CORBA ORB来执行某些操作。这种应用程序可以找到一个合适的对象，这个对象实现了ORB接口，这需要查找JNDI名称java:comp/ORB 或请求ORB对象的注入。要求容器为所有组件(Applet除外)提供java:comp/ORB名称。这种ORB对象的任何引用仅在执行查找的组件实例 中有效。

下面的例子演示了应用程序组件怎样通过注入来获取和使用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");
...
}
```

下面的例子演示了应用程序组件怎样使用JNDI来查找和使用ORB对象。

```
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对象的引用也可以声明在部署描述符中，使用方法与资源管理器连接工厂的引用相同。这种部署描述符入口可以用来指定ORB对象的注入。

JNDI名称java:comp/ORB下可用的ORB实例总是可共享。默认情况下，注入到组件或由部署描述符入口声明的ORB实例也是可共享的。 然而，应用程序可以将Resource注解的shareable元素设为false，或者将部署描述符的res-sharing-scope元素设为 Unshareable，以请求一个非共享的ORB实例。

本节中的标准仅适用于支持CORBA实现互用性的Java EE产品。

5.12.1 应用程序组件供应商的职责

应用程序组件供应商的责任是请求ORB对象的注入，这需要使用Resource注解或为ORB对象定义的查找名称。如果Resource注解的shareable元素被设为false，注入的ORB对象将成为非共享的实例，也就是从一个公共实例变成了一个组件私有的实例。

5.12.2 Java EE产品供应商的职责

Java EE产品供应商的责任是提供一个合适的ORB对象，正如本规范所描述的。

5.13 引用持久化单元

本节描述的元数据注解和部署描述符元素，允许应用程序组件代码使用叫做持久化单元的引用来引出持久化单元的实体管理器工厂，这个引用是一个逻辑名称。持久化单元的引用是应用程序组件环境中的特定入口。部署者将持久化单元的引用绑定到实体管理器工厂，它的配置与持久化单元的 persistence.xml 规格一致，正如Java持久化规范所有描述的。

本节中的标准仅适用于支持Java持久化API的Java EE产品。

5.13.1 应用程序组件供应商的职责

本小节概述了应用程序组件供应商怎样定位持久化单元的实体管理器工厂，以及他(她)的职责。第一个子节描述的注解用来将引用注入到持久化单元的实体管理器工厂；第二个子节描述的API用来访问实体管理器工厂，这要用到持久化单元的引用；第三个子节描述了在部署描述符中声明持久化单元引用的句法。

5.13.1.1 注入持久化单元的引用

应用程序组件的字段或方法可以加上PersistenceUnit注解。元素name指定的名称可以用来在JNDI命名环境中定位被引用的持久化单 元的实体管理器工厂。可选的unitName元素指定了持久化单元的名称，它声明在定义持久化单元的persistence.xml文件中。

下面的示例代码演示了应用程序组件怎样使用注解来声明持久化单元的引用。

```
@PersistenceUnit
EntityManagerFactory emf;
@PersistenceUnit(unitName="InventoryManagement")
EntityManagerFactory inventoryEMF;
```

5.13.1.2 持久化单元引用的编程接口

应用程序组件供应商必须按下述方式使用持久化单元的引用来获取实体管理器工厂的引用。

在应用程序组件环境中为持久化单元的引用分配一个入口(请查看5.13.1.3小节了解怎样在部署描述符中声明持久化单元的引用)。

EJB规范推荐但不强求，将所有持久化单元的引用组织到Bean环境的java:comp/env/persistence子环境中。

使用EJBContext的lookup方法或JNDI API在应用程序组件环境中查找持久化单元的实体管理器工厂。

调用实体管理器工厂的恰当方法来获取实体管理器实例。

下面的代码样本演示了使用EJBContext的lookup方法来获取实体管理器工厂。

```
@PersistenceUnit(name=” persistence/InventoryAppDB”)
@Stateless
public class InventoryManagerBean implements InventoryManager {
@Resource SessionContext ctx;
public void updateInventory(...) {
...
// use context lookup to obtain entity manager factory
EntityManagerFactory emf = (EntityManagerFactory)
ctx.lookup("persistence/InventoryAppDB");
// use factory to obtain application-managed entity manager
EntityManager em = emf.createEntityManager();
...
}
}
```

下面的代码样本演示了直接使用JNDI API来获取实体管理器工厂。

```
@PersistenceUnit(name=” persistence/InventoryAppDB”)
@Stateless
public class InventoryManagerBean implements InventoryManager {
EJBContext ejbContext;
...
public void updateInventory(...) {
...
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// perform JNDI lookup to obtain entity manager factory
EntityManagerFactory = (EntityManagerFactory) initCtx.lookup(
"java:comp/env/persistence/InventoryAppDB");
```

```
// use factory to obtain application-managed entity manager
EntityManager em = emf.createEntityManager();
...
}
}
```

5.13.1.3 在部署描述符中声明持久化单元的引用

虽然持久化单元的引用是应用程序组件环境中的入口，但是应用程序组件供应商不能使用env-entry元素来声明它。

这种情况中，如果没有使用元数据注解，应用程序组件供应商必须在部署描述符中使用persistence-unit-ref元素来声明所有持久化单 元的引用。这使应用程序组装者或部署者可以发现应用程序组件使用的所有持久化单元的引用。部署描述符入口也可以用来指定，将持久化单元的引用注入到应用程 序组件中。

每个persistence-unit-ref元素只对应一个持久化单元的实体管理器工厂的引用。元素persistence-unit-ref 中包含：可选的description和persistence-unit-name元素，以及必须的 persistence-unit-ref-name元素。

元素persistence-unit-ref-name 包含了应用程序组件代码中的环境入口的名称。环境入口的名称默认关联到java:comp/env上下文环境(也就是说，这个名称应该是 persistence/InventoryAppDB，而不是java:comp/env/persistence/InventoryAppDB)。元素persistence-unit-name 是持久化单元的名称，它指定在持久化单元的persistence.xml文件中。

下面的例子中，InventoryManager企业Bean使用的持久化单元的引用的声明，上面的小节中演示过这个企业Bean。

```
...
<persistence-unit-ref>
<description>
Persistence unit for the inventory management application.
</description>
<persistence-unit-ref-name>
persistence/InventoryAppDB
</persistence-unit-ref-name>
<persistence-unit-name>
InventoryManagement
</persistence-unit-name>
</persistence-unit-ref>
...
```

5.13.2 应用程序组装者的职责

应用程序组装者可以使用部署描述符的persistence-unit-name元素来声明持久化单元的引用。应用程序组装者(或供应商)可以在引 用方应用程序组件的persistence-unit-name元素中使用下面的句法来避免Java EE应用程序中持久化单元的名称冲突。应用程序组装者为被引用的持久化单元指

定persistence.xml文件根元素的路径名称，并追加持久化单元的 名称，用#号分隔。这个路径名称关联到引用方应用程序组件的JAR文件。当应用程序组装者不能改变持久化单元的名称时，通过这种方式，可以唯一地标识具有 相同名称的多个持久化单元。

例如，

```
...
<persistence-unit-ref>
<description>
Persistence unit for the inventory management
application.
</description>
<persistence-unit-ref-name>
persistence/InventoryAppDB
</persistence-unit-ref-name>
<persistence-unit-name>
../lib/inventory.jar#InventoryManagement
</persistence-unit-name>
</persistence-unit-ref>
...
```

应用程序组装者使用persistence-unit-name元素将声明在InventoryManagerBean中的持久化单元名称 InventoryManagement链接到inventory.jar中名为InventoryManagement的持久化单元。

遵循下面的规则，部署描述符入口可以覆盖PersistenceUnit注解:

基于JNDI名称和注解可以定位相关的部署描述符入口(默认或显式提供)。

元素persistence-unit-name可以覆盖注解的unitName元素。应用程序组装者或部署者在改变这个值时应该谨慎，因为这样做很可能会破坏应用程序。

如果指定了注入目标，必须准确地命名被注解的字段或属性方法。

5.13.3 部署者的职责

部署者使用部署工具将持久化单元的引用绑定到目标运行环境中为持久化而配置的实际的实体管理器工厂。

部署者必须为声明在元数据注解或部署描述符中的每个持久化单元的引用执行下述任务:

将持久化单元的引用绑定到运行环境中为持久化而配置的实体管理器工厂。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接实体管理器工厂实际的JNDI名称。

如果指定了目标持久化单元的名称，部署者应该将持久化单元的引用绑定到实体管理器工厂。

为实体管理器工厂管理持久化单元提供额外的配置信息，正如Java持久化规范所有描述的。

5.13.4 Java EE产品供应商的职责

Java EE产品供应商有以下职责:

提供部署工具使部署者可以执行上面小节中描述的任务。

为持久化单元提供实体管理器工厂类的实现，这个持久化单元与容器一起配置。实体管理器工厂类的实现可以由容器直接提供，也可以联合第三方持久化提供方，正如Java持久化规范所描述的。

5.13.5 系统管理员的职责

系统管理员通常由以下职责:

在服务器环境中添加，删除和配置实体管理器工厂。

在某些情况下，可以由部署者执行这些任务。

5.14 持久化上下文的引用

本节描述的元数据注解和部署描述符元素，允许应用程序组件代码引用持久化上下文类型的受容器管理的实体管理器，这需要用到叫做持久化上下文的引用的 逻辑名称。持久化上下文的引用是应用程序组件环境中的特定入口。部署者将持久化上下文的引用绑定到指定类型的持久化上下文的受容器管理的实体管理器，它们 的配置与它们的持久化单元一致，正如Java持久化规范所有描述的。

本节中的标准仅适用于支持Java持久化API的Java EE产品。

5.14.1 应用程序组件供应商的职责

本小节概述了应用程序组件供应商怎样定位受容器管理的实体管理器以及他(她)的职责。第一个子节描述的注解用于将引用注入到受容器管理的实体管理器；第二个子节描述了访问受容器管理的实体管理器的引用的API；第三个小节描述了在部署描述符中声明这些引用的句法。

5.14.1.1 注入持久化上下文环境的引用

应用程序组件的字段或方法可以加上PersistenceContext 注解。元素name指定的名称用于在JNDI命名上下文环境中定位被引用的持久化单元的受容器管理的实体管理器。可选的unitName元素指定了持久化 单元的名称，这个名称声明在定义持久化单元的persistence.xml文件中。可选的type元素指定了是使用事务范围的持久化上下文还是使用扩展 的持久化上下文。如果没有指定类型，默认使用事务范围的持久化上下文。如果受容器管理的实体管理器的引用使用的是扩展的持久化上下文，那么它只能注入到有 状态会话Bean。创建实体管理器时，可选的properties元素用于指定传递给持久化提供方的配置属性。

下面的示例代码演示了EJB组件怎样使用注解来声明持久化上下文环境的引用。

```
@PersistenceContext(type=EXTENDED)
EntityManager em;
```

5.14.1.2 持久化上下文引用的编程接口

应用程序组件供应商必须按下述方式使用持久化上下文的引用来获取为持久化单元配置的受容器管理的实体管理器的引用:

在应用程序组件环境中为持久化上下文的引用分配一个入口。(请查看5.14.1.3小节了解怎样在部署描述符中声明持久化上下文的引用)

EJB规范推荐但不强求，将所有的持久化上下文的引用组织到Bean环境中的java:comp/env/persistence子环境中

使用EJBContext的lookup方法或者JNDI API在应用程序组件环境中查找持久化单元的受容器管理的实体管理器。

下面的代码样本演示了使用EJBContext的lookup方法来获取持久化上下文的实体管理器。

```
@PersistenceContext(name=" persistence/InventoryAppMgr" )
@Stateless
public class InventoryManagerBean implements InventoryManager {
@Resource SessionContext ctx;
public void updateInventory(...) {
...
// use context lookup to obtain container-managed
// entity manager
EntityManager em = (EntityManager)
ctx.lookup("persistence/InventoryAppMgr");
...
}
}
```

下面的代码样本演示了直接使用JNDI API来获取实体管理器。

```
@PersistenceContext(name=" persistence/InventoryAppMgr" )
@Stateless
public class InventoryManagerBean implements InventoryManager {
EJBContext ejbContext;
public void updateInventory(...) {
...
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// JNDI lookup to obtain container-managed entity manager
EntityManager = (EntityManager) initCtx.lookup(
"java:comp/env/persistence/InventoryAppMgr");
...
}
}
```

5.14.1.3 在部署描述符中声明持久化上下文的引用

虽然持久化上下文的引用是应用程序组件环境中的入口，但是应用程序组件供应商不能使用env-entry元素声明它。

取而代之，如果没有使用元数据注解，应用程序组件供应商必须在部署描述符中声明所有持久化上下文的引用，这需要使用persistence- context-ref元素。这使应用程序组装者或部署者可以发现应用程序组件使用的所有持久化上下文的引用。部署描述符入口也可以用来指定将持久化上下 文的引用注入到Bean中。

每个persistence-context-ref 元素只对应一个受容器管理的实体管理器的引用。元素persistence-context-ref包含了可选的description, persistence-unit-name, persistence-context-type和persistence-property元素，以及必须的persistence- context-ref-name元素 。

元素persistence-context-ref-name 包含了用在应用程序组件代码中的环境入口的名称。环境入口的名称默认关联到java:comp/env context (例如，这个名称应该是persistence/InventoryAppMgr，而不是java:comp/env/persistence/InventoryAppMgr)。元素persistence-unit-name是持久化单元的名称，声明在持久化单元的 persistence.xml文件中。元素persistence-context-type指定了是使用事务范围的持久化上下文还是使用扩展的持久化 上下文。它的值为Transaction或Extended。如果没有指定持久化上下文的类型，默认使用事务范围的持久化上下文。创建实体管理器时，可选 的persistence-property元素指定了传递给持久化提供方的配置属性。

下面的例子是InventoryManager企业Bean使用的持久化上下文的引用的声明，这个Bean在上面的小节中已经演示过了。

```
...
<persistence-context-ref>
<description>
Persistence context for the inventory management
application.
</description>
<persistence-context-ref-name>
persistence/InventoryAppDB
</persistence-context-ref-name>
<persistence-unit-name>
InventoryManagement
</persistence-unit-name>
</persistence-context-ref>
...
```

5.14.2 应用程序组装者的职责

应用程序组装者可以在部署描述符中使用persistence-unit-name元素来指定持久化单元的引用，它的句法描述在5.13.2, “应用程序组装者的职责”中。当不能改变持久化单元的名称时，通过这种方式，可以唯一地标识具有相同名称的多个持久化单元。

例如，

```
...
```

```
<persistence-context-ref>
<description>
Persistence context for the inventory management
application.
</description>
<persistence-context-ref-name>
persistence/InventoryAppDB
</persistence-context-ref-name>
<persistence-unit-name>
../lib/inventory.jar#InventoryManagement
</persistence-unit-name>
</persistence-context-ref>
...
```

应用程序组装者使用persistence-unit-name元素来链接持久化单元名称InventoryManagement ，这个名称声明在叫做InventoryManagement的持久化单元的InventoryManagerBean中，这个持久化单元定义在 inventory.jar中。

遵循下面的规则，部署描述符入口可以覆盖PersistenceContext注解:

基于JNDI和注解可以定位相关的部署描述符入口(默认或显式提供)。

元素persistence-unit-name可以覆盖注解的unitName元素。应用程序组装者或部署者在改变这个值时应该谨慎，因为这样做很可能会破坏应用程序。

如果指定了persistence-context-type元素，就会覆盖注解的type元素。一般情况下，应用程序组装者或部署者不应该改变这个元素的值，这样做很可能会破坏应用程序。

如果在PersistenceContext注解过的地方再添加了任何persistence-property元素，并且指定的属性名称与PersistenceContext注解中的相同，注解中指定的值将会被覆盖。

如果指定了注入目标，必须准确地命名被注解的字段或属性方法。

5.14.3 部署者的职责

部署者使用部署工具将持久化上下文的引用绑定到受容器管理的实体管理器。这个实体管理器的持久化上下文有指定的类型，是为目标运行环境中的持久化单元而配置。

部署者必须为声明在元数据注解或部署描述符中每个持久化上下文的引用执行下述任务:

将持久化上下文的引用绑定到受管理的实体管理器，这个实体管理器的持久化上下文有指定的类型，是为持久化单元而配置，而这个存在于运行环境中 的持久化单元声明在它的persistence.xml文件中。例如，部署者可以使用JNDI LinkRef机制创建一个符号来链接到实体管理器实际的JNDI名称。

如果指定了持久化单元的名称，部署者应该将持久化上下文的引用绑定到目标持久化单元的实体管理器上。

为实体管理工厂创建这样的实体管理器以及管理持久化单元提供任何额外的配置信息，正如Java持久化规范所描述的。

5.14.4 Java EE产品供应商的职责

Java EE产品供应商有以下职责:

提供部署工具，使部署者可以执行上面小节中描述的任务。

提供持久化单元的实体管理器类的实现，这个持久化单元与容器一起配置。这个实现可以由容器直接提供，也可以联合第三方持久化提供方，正如Java持久化规范所描述的。

5.14.5 系统管理员的职责

系统管理员通常有以下职责:

在服务器环境中添加，删除和配置实体管理器工厂。

在某些情况下，可以由部署者执行这些任务。

5.15 应用程序名称和模块名称的引用

使用预定义的JNDI名称java:app/AppName，组件可以访问当前应用程序的名称。使用预定义的JNDI名称java:module/ModuleName，组件可以访问当前模块的名称。这两个名称都是用String对象表示的。

5.15.1 应用程序组件供应商的职责

应用程序组件供应商的责任是请求应用程序名称或模块名称的注入，可以在String类型的字段或方法上使用Resource注解，也可以使用定义的名称来查找应用程序或模块的名称。

5.15.2 Java EE产品供应商的职责

Java EE产品供应商有责任提供正确的应用程序和模块名称的String对象，正如本规范所要求的。

5.16 检验器和检验器工厂的引用

本节描述的元数据注解和部署描述符入口，使应用程序可以获取Bean检验的Validator和ValidatorFactory类型的实例。

需要使用那些接口的应用程序，通过查找Validator的名称java:comp/Validator 和ValidatorFactory的名称java:comp/ValidatorFactory可以获取合适的对象，也可以通过Resource注解请求恰当类型对象的注入。不能指定Resource注解的 authenticationType和shareable元素。

```
@Resource ValidatorFactory validatorFactory;

@Resource Validator validator;
```

Validator对象使用默认的检验上下文。这表示，所有这样的Validator等价于那些从ValidatorFactory无参的getValidator方法获取的。

换句话说，下面的两段代码是等价的:

```
// obtaining a Validator directly

Context initCtx = new InitialContext();
```

```
Validator validator = (Validator)initCtx.lookup(
"java:comp/Validator");
// obtaining a Validator from a ValidatorFactory
Context initCtx = new InitialContext();
Validator validator = ((ValidatorFactory)initCtx.lookup(
"java:comp/ValidatorFactory")) .getValidator();
```

Validator或ValidatorFactory对象的引用也可以声明在部署描述符中，使用方法与资源环境的引用相同。

想要自定义返回的ValidatorFactory，那么EJB，Web或应用程序客户端模块可以指定Bean Validation XML部署描述符。这个描述符在Web模块中的名称是WEB-INF/validation.xml，而所有其它类型模块的这个描述符名称是META-INF/validation.xml 。

检验部署描述符仅对模块中的ValidatorFactory实例有效。不是每个应用程序都有检验部署描述符。

5.16.1 应用程序组件供应商的职责

应用程序组件供应商的责任是请求Validator或ValidatorFactory的注入，可以使用Resource注解，也可以使用定义的名称来查找Validator或ValidatorFactory的实例。

应用程序组件供应商可以自定义ValidatorFactory，从而间接地定义Validator实例，这需要将Bean Validation部署描述符放入特定的应用程序模块中。

5.16.2 Java EE产品供应商的职责

Java EE产品供应商负责提供合适的Validator和ValidatorFactory对象，正如本规范所要求的。

我们注意到在Bean Validation API中包含一个ValidatorFactoryBuilder 接口，得益于它的实现者，可以用它创建ValidatorFactory对象，这个对象的配置需要依据从java.io.InputStream读入的检 验部署描述符的内容。

5.17 数据源资源定义

除了引用本章定义的资源，应用程序也可以定义一个DataSource资源。DataSource 资源使用JDBC驱动访问数据库。

DataSource资源可以定义在任何JNDI命名空间中，这些命名空间描述在5.2.2,“应用程序组件环境的命名空间”中。例如，DataSource资源可以定义在:

- java:comp 命名空间，被单个组件使用。
- java:module 命名空间，被模块中的所有组件使用。
- java:app 命名空间，被应用程序中的所有组件使用。
- java:global 命名空间，被所有应用程序使用。

DataSource资源可以定义在Web模块，EJB模块，应用程序客户端模块中，这需要使用应用程序部署描述符的data-source元素。

例如:

```
<data-source>
```

<description>Sample DataSource definition</description>
<name>java:app/MyDataSource</name> <class-
name>com.foobar.MyDataSource</class-name> <server-
name>myserver.com</server-name> <port-number>6689</port-number>
<database-name>myDatabase</database-name> <user>lance</user>
<password>secret</password>

<property>

<name>Property1</name>

<value>10</value>

</property>

<property>

<name>Property2</name>

<value>20</value>

</property>

<login-timeout>0</login-timeout>

<transactional>false</transactional> <isolation-
level>TRANSACTION_READ_COMMITTED</isolation-level> <initial-pool-
size>0</initial-pool-size>

<max-pool-size>30</max-pool-size>

<min-pool-size>20</min-pool-size>

<max-idle-time>0</max-idle-time>

<max-statements>50</max-statements>

</data-source>

DataSource资源也可以在一个受容器管理的类上使用DataSourceDefinition注解进行定义，比如Servlet或企业Bean类。

例如:

@DataSourceDefinition(

name="java:app/MyDataSource",

className="com.foobar.MyDataSource",

portNumber=6689,

serverName="myserver.com",

user="lance",

password="secret"

)

(当然，我们不推荐将密码包含在产品系统的代码中，不过这在测试期间比较方便。在部署应用程序时，可以用部署描述符覆盖DataSource定义的密码或其它部分)

一经定义，组件就可以使用部署描述符的resource-ref元素或Resource注解来引用DataSource资源。

例如，可以按下面的方式引用上面的DataSource:

```
@Stateless
public class MySessionBean {
@Resource(lookup = "java:app/MyDataSource")
DataSource myDB;
...
}
```

5.17.1 应用程序组件供应商的职责

应用程序组件供应商负责定义DataSource，这需要使用DataSourceDefinition注解或部署描述符的data-source元素。

5.17.2 Java EE产品供应商的职责

Java EE产品供应商负责配置DataSource，这需要依据应用程序组件供应商提供的定义，然后在JNDI中指定的名称下激活它。

5.18 引用受管理的Bean

本节描述符的元数据注解和部署描述符入口，使应用程序可以获取受管理Bean的实例。

通过JNDI名称可以获取叫做Managed Bean的实例，使用的名称模式与EJB组件一致:

java:app/<module-name>/<bean-name>

java:module/<bean-name>

后者仅在声明Managed Bean的模块中运行。每次这样的查找必须返回一个新的实例。

另外，也可以使用Resource注解来请求注入给定类型或名称的Managed Bean。如果使用lookup元素指定了名称，那么资源可以是Managed Bean类实现的任何类型，包括它的任何接口。如果没有指定名称，类型必须是Managed Bean类自身。(注意，Resource注解的name元素的责任目标与lookup元素完全不同，本规范坚持Resource注解的其它用途)。不能 指定Resource注解的authenticationType和shareable 元素。

例如，在客户代码的同一个模块中定义了一个叫做“cart”的Bean ShoppingCartBean，并且实现了ShoppingCart接口，客户端就可以使用下面的任何方法来获取Bean类的实例:

```
@Resource ShoppingCartBean cart;
@Resource(lookup=”java:module/cart”) ShoppingCart cart;
ShoppingCart cart = (ShoppingCart) context.lookup(“java:module/cart”);
```

Managed Bean的引用可以声明在部署描述符的resource-ref元素中。元素res-type必须包含一个Managed Bean实现的类型。必须给出lookup-name并通过这个名称引用Managed Bean。可以省略res-sharing-scope和res-auth元素；如果给出，它们的值依次必须是Shareable和Container， 以便匹配Resource注解相应元素的默认值。

下面的例子展示了怎样声明上例中ShoppingCartBean的引用，这次使用描述符。（想要让这个例子更真实一点，你可以在resource-ref元素中添加一个injection-target子元素）

<resource-ref>

<res-ref-name>bean/cart</ref-ref-name>

<ref-type>com.acme.ShoppingCart</ref-type>

<lookup-name>java:module/cart</lookup-name>

</resource-ref>

5.18.1 应用程序组件供应商的职责

应用程序组件供应商负责请求注入Managed Bean，或使用恰当的JNDI名称查找它。

5.18.2 Java EE产品供应商的职责

Java EE产品供应商负责提供被请求的Managed Bean类的恰当实例，正如本规范所要求的。

5.19 Bean管理器的引用

本节描述的元数据注解和部署描述符入口，使应用程序可以获取CDI BeanManager类型的实例。

通常，只有使用CDI SPI的可移植性扩展需要访问BeanManager。应用程序偶尔需要访问那个接口；那样的话，应用程序应该在JNDI名称java:comp/BeanManager下查找BeanManager的实例，也可以通过Resource注解请求注入 javax.enterprise.inject.spi.BeanManager类型的对象。如果是后者，就不能指定Resource注解的 authenticationType和shareable元素。

@Resource BeanManager manager;

按照CDI规范，Bean也可以使用Inject注解请求BeanManager的注入。

@Inject BeanManager manager;

BeanManager对象的引用也可以声明在部署描述符中，使用方法与资源环境的引用相同。Bean管理器仅在激活了CDI的模块中可用。

5.19.1 应用程序组件供应商的职责

应用程序组件供应商负责请求BeanManager实例的注入，这需要使用Resource注解或通过JNDI中定义的名称进行查找。

为没有激活CDI的模块请求注入BeanManager是错误的。

5.19.2 Java EE产品供应商的职责

Java EE产品供应商负责为激活了CDI的模块提供恰当的BeanManager实例，正如本规范所要求的。

5.20 支持依赖注入(JSR-330)

在Java EE中，对依赖注入注解(DI)的支持是由CDI(JSR-299)进行调节的，此注解由Java规范(JSR-330)的依赖注入部分指定。容器必须支 持带有@javax.inject.Inject注解的注入点，只限于CDI标明的范围。特别是，对DI注解的支持受限于使用它们的类，这个类属于CDI 指定的Bean存档的组成部分(通俗地说，Bean存档就是一种文件，例如，依靠META-INF/bean.xml描述符

为EJB模块激活CDI支 持)。请注意，根据CDI规范，以Bean存档的形式提供对应用程序客户端模块的支持是可选的。

按照CDI规范，受管理的Bean支持依赖注入。目前，类有三种方式可以成为受管理的Bean:

1.是一个EJB会话Bean组件。

2.添加@ManagedBean注解。

3.满足CDI规范3.1节的条件。

至少满足上述条件之一，这个类才有资格完全支持依赖注入，正如CDI规范所描述的。

显然，缺少了任何附加的注解，表 5-1中列出的大多数组件类都不属于受管理的Bean。当包含了Bean存档时，为了使所有组件类型对注入的支持更加统一，要求Java EE容器支持字段或方法的注入(构造方法除外)，这需要在表 5-1中列出的所有组件类上添加@javax.inject.Inject注解。这种注入执行的逻辑阶段必须与带有@Resource注解的字段和方法的 资源注入相同。特别是，依赖注入必须执行在带有@PostConstruct注解的任何方法调用之前。为了支持这样的注入点，容器必须执行类似于下面的步 骤，包括CDI SPI的使用。

1.获取一个BeanManager实例。

2.通过调用BeanManager.getCreationalContext()方法获取一个目标组件的创建型上下文。

3.为每个注入点创建一个InjectionPoint实例，它的getBean()方法必须返回null，因为这个组件不是CDI所识别的Bean(如果是，那么就会直接由CDI控制，而这些步骤并不适合它) 。

4.通过调用BeanManager.getInjectableReference(InjectionPoint, CreationalContext)方法为每个注入点获取一个可注入的引用，这个方法的参数InjectionPoint来自第3步，参数 CreationalContext来自第2步。

5. 将得到的引用注入到目标组件中。

容器可以优化上述步骤，例如，避免调用实际的CDI SPI，而用容器特定的接口代替，只要结果是相同的。

第6章 应用程序编程接口

本章描述了Java™平台企业版(Java EE)的API标准。Java EE要求为Java EE应用程序提供大量API，首先是Java核心API，还包含了很多其它的Java技术。

6.1 必须的API

Java EE应用程序组件执行在容器提供的运行时环境中，容器是Java EE平台的一部分。完整的Java EE平台支持4种类型的容器，适合对应的Java EE应用程序组件类型，它们是: 应用程序客户端容器，Applet容器，支持Servlet和JSP页面的Web容器，以及企业Bean容器。Java EE Profile可以只支持这些组件类型的子集，由单独的Java EE Profile规范定义。

本章中的每项技术标准适用于任何包含这项技术的Java EE产品。注意，即使Java EE Profile不要求支持某项特殊技术，基于这项Profile的Java EE产品仍然可以包含对这项技术的支持。在这种情况下，可以为该项技术应用本章描述的标准。

6.1.1 Java兼容性API

容器提供的所有应用程序组件至少带有Java平台标准版v6（Java SE）API。容器可以提供最新版的Java SE平台，以满足所有的Java EE平台标准。Java SE平台包含了下列企业级技术:

Java IDL

JDBC

RMI-IIOP

JNDI

JAXP

StAX

JAAS

JMX

JAX-WS

JAXB

JAF

SAAJ

Common Annotation

特别是，Applet运行环境必须兼容Java SE 6平台。由于主流浏览器还没有提供这样的支持，Java EE产品可以使用Java插件来提供这种必须的环境。Java插件的使用不是必须的，但它满足了这个标准，提供了一个兼容Java SE 6平台的Applet运行环境。本规范没有给Applet容器增加Java SE平台之外的标准。

一些Java SE 6平台包含的企业级技术也可以从Java SE平台之外获取，并且本规范要求使用一些技术的最新版，这将会在下面的小节中进行描述。

Java SE API规范可以从 http://java.sun.com/javase/6/docs/ 获取。

6.1.2 必须的Java技术

完整的Java EE平台为本规范定义的每种容器也提供了大量的Java技术。表 6-1标明了这些技术及其版本号; 哪种容器包含此项技术; 此项技术是必须的(REQ)，推荐可选的(POPT)，还是可选的(OPT)。每项Java EE Profile规范会包含一张类似的表，描述哪项技术对这项Profile是必须的。注意，有些技术带有标记。

Java技术	App Client	Web	EJB	Status
EJB 3.1	Ya	Y	Y	REQ, POPTb
Servlet 3.0	N	Y	N	REQ
JSP 2.2	N	Y	N	REQ
EL 2.2	N	Y	N	REQ
JMS 1.1	Y	Y	Y	REQ
JTA 1.1	N	Y	Y	REQ
JavaMail 1.4	Y	Y	Y	REQ
Connector 1.6	N	Y	Y	REQ
Web Service 1.3	Y	Y	Y	REQ
JAX-RPC 1.1	Y	Y	Y	REQ, POPT

JAX-WS 2.2	Y	Y	Y	REQ
JAX-RS 1.1	N	Y	N	REQ
JAXB 2.2	Y	Y	Y	REQ
JAXR 1.0	Y	Y	Y	REQ, POPT
JavaEE Management 1.1	Y	Y	Y	REQ
JavaEE Deployment 1.2c	N	N	N	REQ, POPT
JACC 1.4	N	Y	Y	REQ
JASPIC 1.0	N	Y	Y	REQ
JSP Debugging 1.0	N	Y	N	REQ
JSTL 1.2	N	Y	N	REQ
Web Service MetaData 2.1	Y	Y	Y	REQ
JSF 2.0	N	Y	N	REQ
Common Annotations 1.1	Y	Y	Y	REQ
Java Persistence 2.0	Y	Y	Y	REQ
Bean Validation 1.0	Y	Y	Y	REQ
Managed Bean 1.0	Y	Y	Y	REQ
Interceptor 1.1	Y	Y	Y	REQ
Contexts and Dependency Injection for Java EE 1.0	Y	Y	Y	REQ
Dependency for Java 1.0	Y	Y	Y	REQ

a. 仅对客户端API

b. 仅对实体Bean

c. 请查看 6.18 中的详细信息

推荐可选项描述在下一小节中。

上面提到的所有类和接口必须由Java EE容器提供。在某些情况下，不要求Java EE产品提供实现了这些接口的对象，而是交给应用程序服务器。然而，这类接口的定义必须包含在Java EE平台中。

6.1.3 被修剪的Java技术

随着Java EE规范的不断发展，一些早期的Java EE技术不再适合当前平台的需求。Java EE专家组按照最先由Java SE专家组定义的流程，将这些技术从平台移除(http://blogs.sun.com/mr/entry/removing_features)，此方式是谨慎和有序的，尽可能低地影响开发者对这些技术的使用，使这个平台能够更加健壮地成长。简而言之，这个流程定义了两个步骤:

1. 发布平台N的总专家组(UEG)提出修剪某一特性，并在发布的规范中记录此项提议。

2. 发布平台N+1的UEG决定是否从新的发布中修剪这一特性，或是将它作为必须的组件保留下来，也可以把它定为“推荐移除”状态，留给下一个UEG来决定。

成功地为某一特性应用这个策略并不是真正删除了这个特性，而是在一定程度上淡化这个特性，将它从平台必须的组件变为可选的组件。虽然这一特性没有真正从规范中移除，然而它可能会被产品移除，这是产品供应商的选择。

在未来发布中可能会修剪的技术在表6-1中被标记为推荐可选。已经被修剪的技术在表6-1中被标记为可选。Java EE 6中没有标记为可选的技术。

6.2 Java平台标准版(Java SE) 标准

6.2.1 编程约束

Java EE编程模型划分了应用程序组件供应商和Java EE产品供应商的职责：应用程序组件供应商致力于编写业务逻辑，而Java EE产品供应商致力于提供一套可管理的基础系统框架，使应用程序可以部署进去。

这种分工要求对应用程序组件在功能上进行约束。如果应用程序组件也提供同样的功能，就会与基础系统框架发生冲突，并且难于管理。

例如，倘若允许一个企业Bean管理线程，Java EE平台就无法管理这个企业Bean的生命周期，并且不能正确地管理事务。

我们不想抽取Java SE平台的子集，而是希望Java EE产品供应商能够在Java EE平台上直接使用Java SE产品，因此我们使用Java SE安全权限机制来表述对应用程序组件供应商的强制编程约束。

在本节中，我们指定的Java SE安全权限，必须由Java EE产品供应商为每个应用程序组件类型提供。我们把这些权限叫做Java EE安全权限集。Java EE安全权限集被要求作为Java EE API协议的组成部分。可移植的应用程序只信任在此规定的权限集。

6.2.2 Java EE安全权限集

Java EE安全权限集定义了应用程序组件期望的最小权限集。所有Java EE产品必须能够部署需要这些权限集的应用程序组件。产品供应商必须确保应用程序组件使用的功能不会与Java EE安全权限集冲突。

在特定设备环境中为应用程序组件确立最佳的安全权限集是一个策略性的问题，不属于本规范讨论的范畴。Java EE产品允许应用程序在没有任何安全管理器的情况下运行，也可以配置一个强制实施了任何安全权限集的安全管理器，正如企业环境所要求的。运行应用程序的所 有Java EE产品，必须且至少支持这里描述的权限集。一些Java EE产品允许组件配置安全权限集，为一些组件提供比这里描述的更多或更少的权限。本规范的未来版本将允许在应用程序组件的部署描述符中指定这些安全标准。 目前，所需权限不在这个最小集合里的应用程序组件只能在它们的文档中描述自己标准。注意，需要更多权限的应用程序可能无法部署在某些Java EE产品上。

有关Java SE安全权限的完整描述，参见http://
java.sun.com/javase/6/docs/technotes/guides/security/permissions.html

6.2.3 Java EE安全权限集列表

表 6-2 列出了Java EE安全权限集。这个典型权限集是每个类型的组件理应享有的。

表 6-2 Java EE安全权限集

安全权限	目标	行为
应用程序客户端		
java.aw.AWTPermission	accessClipboard	
java.aw.AWTPermission	accessEventQueue	
java.aw.AWTPermission	showWindowWithoutWainingBanner	
java.lang.RuntimePermission	exitVM	
java.lang.RuntimePermission	loadLibrary	
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect

java.net.SocketPermission	localhost:1024-	accept,listen
java.io.FilePermission	*	read,write
java.util.PropertyPermission	*	read
Applet客户端		
java.net.SocketPermission	codebase	connect
java.util.PropertyPermission	limited	read
Web组件和EJB组件		
java.lang.RuntimePermission	loadLibrary	
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect
java.io.FilePermission	*	read,write
java.util.PropertyPermission	*	read

注意，安装Java EE产品的操作系统可以强制附加自己的安全约束，必须考虑到这一点。比方说，组件使用的用户标识不能有权读写所有文件。

6.2.4 附加标准

6.2.4.1 网络

Java SE平台包含了支持多种URL协议的即插即用机制，这需要使用java.net.URLStreamHandler类和java.net.URLStreamHandlerFactory接口。

必须支持下面的URL协议:

file: 只需要支持从file URL读取。更确切地说，对应的URLConnection对象的getOutputStream方法将会抛出UnknownServiceException异常。并且根据上面描述的权限，文件访问也是有限制的。

http: 必须支持HTTP协议1.1。http URL必须同时支持输入和输出。

https: https URL对象必须支持SSL3.0和TLS1.0，也必须同时支持输入和输出。

Java SE平台也包含一种机制，将URL字节流转换成相应的对象，这是通过java.net.ContentHandler类和 java.net.ContentHandlerFactory接口来完成的。ContentHandler 对象能够将MIME字节流转换成对象。通常使用URL和URLConnection对象的getContent方法间接地访问 ContentHandler对象。

当使用getContent方法访问下面的MIME类型数据时，必须返回表 6-3中列出的相应Java对象类型。

表 6-3 getContent方法返回对象的Java类型

MIME类型	Java类型
image/gif	java.awt.Image
image/jpeg	java.awt.Image
image/png	java.awt.Image

很多环境使用HTTP代理而不是直接连接到HTTP服务器。如果想在本地环境中使用HTTP代理，Java SE平台的HTTP支持应该使用合适的代理进行配置。不要求应用程序组件为使用http URL而配置代理支持。

大多数企业环境会包含防火墙，限制内部网络(局域网)到国际互联网的访问，反过来也是如此。通常使用HTTP协议来穿过这种防火墙进行访问，也可能使用代理服务器。通常不会使用总的TCP/IP通信来穿过防火墙，这包括RMI-JRMP和RMI-IIOP。

还需要考虑到使用各种协议的应用程序组件之间的通信。本规范要求，本地策略允许HTTP访问能够通过防火墙。一些Java EE产品可以为其它通信开辟通道，但这既不是规定也不是必须的。

6.2.4.2 JDBC™ API

JDBC API是Java SE平台的组成部分，它可以访问广泛的数据存储系统。然而，Java SE平台不要求满足Java兼容性质量标准的系统提供的数据库必须通过JDBC API来访问。

为允许部署可移植的应用程序，Java EE规范不要求Java EE产品通过JDBC API来使用和访问这类数据库。Web组件，企业Bean和应用程序客户端必须能够访问这样的数据库，对Applet不作要求。另外，数据库驱动必须满足 JDBC规范中描述的JDBC兼容性标准。

Java EE应用程序不应该试图直接加载JDBC驱动，而是应该使用JDBC规范推荐的技术并执行JNDI查找来定位数据源对象。选择数据源对象的JNDI名称应 该依据5.7,“资源管理器连接工厂的引用”中的描述。当获取数据库连接时，Java EE平台必须能够提供一个数据源，它不需要应用程序提供任何验证信息。当然，连接数据库时应用程序也可以提供用户名和密码。

当企业Bean使用JDBC API连接时，事务特征通常由容器控制。组件不应该试图改变连接的事务特征，也不应该试图提交事务，回滚事务或设置自动提交方式。与当前事务上下文不兼容的改变将抛出SQLException异常。EJB规范对企业Bean定义了严格的规则。

注意，当组件使用JTA UserTransaction接口创建事务时，会使用相同的约束。组件不应该试图操作上面列出的JDBC Connection对象，这会与事务上下文冲突。

Java EE环境中支持JDBC API的驱动必须满足JDBC 4.0兼容性标准，正如JDBC 4.0规范6.4节中描述的。

JDBC API支持JNDI命名的连接，连接池和分布式事务。连接池和分布式事务特性用来配合应用程序服务器。不要求Java EE产品使用这些API来增强应用程序服务器的能力，但是它们被证明是有用的。

连接器体系结构定义了一个SPI，通过附加的安全功能和对资源适配器完整的打包部署功能，它从根本上扩展了JDBC SPI的功能。支持连接器体系结构的Java EE产品必须支持部署和使用JDBC驱动，它们已经被编写并打包，作为连接器体系结构的资源适配器。

JDBC 4.0规范参见http://java.sun.com/products/jdbc/download.html

6.2.4.3 Java IDL

本节中的标准仅适用于通过CORBA提供互用性的Java EE产品。

Java IDL 使应用程序可以访问任何语言编写的CORBA对象，这需要使用标准的IIOP协议。Java EE安全约束通常会阻止所有应用程序组件类型(应用程序客户端除外)创建和导出CORBA对象，但是所有Java EE应用程序组件类型可以作为CORBA对象的客户端。

Java EE产品必须支持Java IDL，正如CORBA 2.3.1规范的1-8, 13和15章所定义的，参见http://www.omg.org/cgi-bin/doc?formal/99-10-07和 IDL-Java语言映射规范http://www.omg.org/cgi-bin/doc?ptc/2000-01-08

IIOP协议支持单连接多元调用功能。所有Java EE产品必须为来自客户端的请求支持：在一个连接上多元调用Java IDL服务器对象或RMI-IIOP服务器对象(例如企业Bean)。服务器必须可以按任意顺序进行响应，以避免一个调用等待另一个调用的完成而引起死 锁。不要求Java EE 客户端支持多元调用，但这是非常值得推荐的。

Java EE产品必须为CORBA Portable Object Adapter (POA)提供支持，从而支持可移植的存根(stub)，骨架(skeleton)和衔接(tie)类。定义或使用CORBA对象的Java EE应用程序(企业Bean除外)必须在应用程序包中包含这种可移植的存根(stub)，骨架(skeleton)和衔接(tie)类。

Java EE应用程序需要使用org.omg.CORBA.ORB 对象的实例来执行很多Java IDL和RMI-IIOP操作。调用ORB.init(new String[0], null)方法默认返回的ORB 必须可用于这样的用途；应用程序不需要知道为支持ORB和RMI-IIOP提供的实现类。

另外，考虑到性能因素，在应用程序的组件间共享ORB实例常常是有益的。为支持这种用法，要求所有Web，企业Bean和应用程序客户端容器在 JNDI命名空间java:comp/ORB下提供一个ORB实例。容器可以决定是否在组件间共享这个实例。容器自己也可以使用这个ORB实例。为支持应 用程序的分隔，不应该在不同应用程序的组件间共享ORB实例。为保证组件安全地共享这个ORB实例，可移植的组件必须约束它们对某些ORB API和功能的使用:

不要调用ORB的shutdown方法。

不要使用容器使用的id来调用org.omg.CORBA_2_3.ORB 的register_value_factory 和 unregister_value_factory方法。

Java EE产品必须提供COSNaming服务来支持EJB交互性标准。必须能够使用Java IDL COSNaming API来访问这种COSNaming服务。带有适当特权的应用程序必须能够在COSNaming服务中查找对象。COSNaming定义在互用性命名服务 规范中，参见http://www.omg.org/cgi-bin/doc?formal/2000-06-19

6.2.4.4 RMI-JRMP

JRMP是Java特有的远程方法调用(RMI)协议。Java EE安全约束通常会阻止所有应用程序组件类型(应用程序客户端除外)创建和导出RMI对象，但是所有Java EE应用程序组件类型可以作为RMI对象的客户端。

6.2.4.5 RMI-IIOP

本节中的标准仅适用于包含了EJB容器并使用RMI-IIOP支持互用性的Java EE产品。

RMI-IIOP允许使用IIOP协议访问RMI风格的接口定义的对象。必须能够通过RMI-IIOP访问远程企业Bean。通过RMI- IIOP，一些Java EE产品很容易地就能让所有远程企业Bean总是(并且只是)可访问的；通过管理和部署行为，其它产品可以对其进行控制。只要使用RMI-IIOP可以访 问任何远程企业Bean(或扩展到所有远程企业Bean)，这些途径或别的途径就是允许的。

所有组件必须使用javax.rmi.PortableRemoteObject类的narrow方法来访问远程企业Bean，正如EJB规范所描 述的。因为远程企业Bean可以使用RMI协议部署，可移植的应用程序不能依赖RMI-IIOP对象的特征(例如，使用Stub和Tie基类)，这些特征 超出了EJB规范。

Java EE安全约束通常会阻止所有应用程序组件类型(除了应用程序客户端)创建和导出RMI-IIOP对象。所有Java EE应用程序组件类型可以作为RMI-IIOP对象的客户端。Java EE应用程序也应该使用JNDI来查找非EJB RMI-IIOP对象。这种非EJB RMI-IIOP对象使用的JNDI名称应该在部署时进行配置，这需要使用标准环境入口机制(参见5.2,“JNDI命名上下文”)。应用程序应该使用环 境入口从JNDI取得一个名称，然后使用这个名称查找RMI-IIOP对象。通常这样的名称会被配置成COSNaming命名服务中的名称。

本规范没有为应用程序提供一种可移植的方式，将对象绑定到命名服务中的名称。一些产品可以支持使用JNDI和COSNaming来绑定对象，但这不 是必须的。可移植的Java EE应用程序客户端可以创建非EJB RMI-IIOP服务器对象，将它作为回调对象使用，或传递到对其它RMI-IIOP对象的调用中。

注意，虽然RMI-IIOP没有指定怎样传递当前安全上下文或事务上下文，但EJB互用性规范定义了这种上下文的传递。本规范只要求在使用RMI- IIOP访问企业Bean时支持上下文信息的传递，正如EJB规范所描述的。不要求在使用RMI-IIOP访问非企业Bean对象时传递上下文信息。

RMI-IIOP规范描述了怎样创建可移植的Stub和Tie类。为了移植所有使用CORBA便携式对象适配器(POA)的实现，Tie类必须继承org.omg.PortableServer.Servant类。通常这是使用rmic命令的-poa选项来实现。一个Java EE产品必须提供对这些可移植的Stub和Tie类的支持，通常是使用必要的CORBA POA。然而，对于没有使用POA来实现RMI-IIOP的系统的可移植性，应用程序不应该依赖Tie继承Servant类来达到。定义或使用了RMI- IIOP对象(而非企业Bean)的Java EE应用程序必须在应用程序包中纳入这种可移植的Stub和Tie类。然而，企业Bean的Stub和Tie对象不能包含在应用程序中: 如果需要，它们会被Java EE产品在部署或运行时生成。

RMI-IIOP定义在CORBA 2.3.1规范的5、6、13章和10.6.2节中，参见http://www.omg.org/cgi-bin/doc?formal/99-10-07以及Java™语言-IDL映射规范http://www.omg.org/cgi-bin/doc?ptc/2000-01-06.

6.2.4.6 JNDI

支持下述对象类型的Java EE产品必须使它们在JNDI命名空间中可用: EJBHome对象，EJBLocalHome对象，JTA UserTransaction对象，JDBC API DataSource对象，JMS ConnectionFactory和Destination对象，JavaMail Session对象，URL对象，资源管理器ConnectionFactory对象(在连接器规范中指定)，ORB对象，EntityManager对 象以及5,“资源，命名和注入”中描述的其它Java语言对象。Java EE产品中的JNDI实现必须能够在单个应用程序组件中使用单个JNDI InitialContext支持所有这些对象的使用。应用程序组件通常会使用默认的无参构造方法创建一个JNDI InitialContext，之后就可以用这个InitialContext查找上面指定的对象。

用于查找Java EE对象的名称依赖于应用程序。应用程序组件的部署描述符列出对象期望的名称和类型。部署者配置JNDI命名空间使相应的组件可用。用于查找这种对象的JNDI名称必须是在JNDI的java:命名空间中。详细信息请查看5,“资源，命名和注入”。

本规范为这些情况定义了两个特殊的名称，当Java EE产品包含相应的技术时。所有访问JTA UserTtransaction接口的应用程序组件，可以使用名称java:comp/UserTransaction找到对应的 UserTransaction对象。在所有容器(Applet除外)中，应用程序组件可以使用名称java:comp/ORB查找CORBA ORB实例。

用于查找特定Java EE对象的名称因不同的应用程序组件而异。一般来说，JNDI名称不能作为参数在远程组件的调用中传递(例如，在企业Bean调用中)。

JNDI的java:命名空间一般作为链接到其它命名系统的符号而实现。不同的基础命名服务可以用来保存不同类型的对象，甚至对象的不同实例。Java EE产品有责任提供必需的JNDI服务提供方来访问本规范中定义的各种对象。

本规范要求Java EE平台提供执行以上查找操作的功能。不同的JNDI服务提供方可以提供不同的功能，例如，一些服务提供方在命名服务中只提供对数据的只读操作。

可以要求Java EE产品提供一个COSNaming命名服务来满足EJB互用性标准。在这种情况下，COSNaming JNDI服务提供方必须对Web，EJB和应用程序客户端容器可用。它通常对Applet容器也是可用的，但这不是必须的。

COSNaming JNDI服务提供方是Sun的Java SE 6 SDK和JRE的一部分，但不是Java SE规范的必须组件。关于COSNaming JNDI服务提供方规范，请查看http://java.sun.com/javase/6/docs/technotes/guides/jndi/jndi-cos.html

Java EE平台完整的命名标准，请查看5,“资源，命名和注入”。关于JNDI规范，请查看http://java.sun.com/products/jndi/docs.html

6.2.4.7 上下文类加载器

本规范要求Java EE容器为每个线程的上下文配置类加载器，使它们可以动态地加载应用程序提供的系统和类库中的类。EJB规范要求所有EJB客户端容器为每个线程的上下文 配置类加载器，使它们可以动态地加载系统参数类。使用Thread的getContextClassLoader方法可以访问每个线程的上下文类加载器。

这些应用程序使用的类通常由不同层次的类加载器加载。从顶层的应用程序类加载器到扩展类加载器，直到最底层的系统类加载器。顶层应用程序类加载器需 要委托下层的类加载器进行加载。被下层类加载器加载的类，例如可移植的EJB系统参数类，需要能够发现用于加载应用程序类的顶层应用程序类加载器。

本规范要求容器为每个线程的上下文配置类加载器，能够用来加载上面描述的顶层应用程序类。请查看8.2.5,“动态类加载”了解动态类加载的建议。

6.2.4.8 Java™验证和授权服务(JAAS)标准

所有EJB容器和所有Web容器必须支持JAAS API的使用，正如连接器规范所规定的。所有应用程序客户端容器必须支持JAAS API的使用，正如10,“应用程序客户端”所规定的。

JAAS规范参见http://java.sun.com/products/jaas

6.2.4.9 Logging API 标准

Logging API 提供的类和接口在java.util.logging包中，这个包是Java™平台的核心日志工具。本规范不要求提供对日志的附加支持。Java EE应用程序通常没有日志权限来控制日志的配置，但是可以使用日志API来导出日志记录。本规范的未来版本可能会要求Java EE容器使用日志API来记录某些事件。

6.2.4.10 Preferences API标准

java.util.prefs包中Preferences API使应用程序可以保存和恢复用户和系统的preference(偏好)和配置信息。Java EE应用程序通常没有运行时权限(“preferences”)来使用Preferences API。本规范没有在Java EE应用程序使用的主体和Preferences API定义的用户偏好树之间定义任何关系。本规范的未

来版本可能会定义Java EE应用程序对Preferences API的使用。(译者注，例如可以使用Preferences API来操作Windows注册表)

6.3 企业级JavaBeans™ (EJB) 3.1 标准

本规范要求Java EE产品为企业Bean提供支持，正如EJB规范所规定的。EJB规范参见http://java.sun.com/products/ejb/docs.html

本规范此时没有强制实施任何附加标准。注意，EJB规范包含了基于RMI-IIOP的EJB互用性协议。所有支持EJB客户端的容器必须能够使用 EJB互用性协议来调用企业Bean。所有EJB容器必须支持使用EJB互用性协议的企业Bean调用。Java EE产品也可以支持其它协议来调用企业Bean。

Java EE产品可以支持多种对象系统(例如，RMI-IIOP和RMI-JRMP)。不一定总是能够将对象的引用从一个对象系统传递给另一个对象系统中的对象。 然而，当企业Bean使用了RMI-IIOP协议，它必须能够将RMI-IIOP或Java IDL对象的引用作为参数传递给这类企业Bean的方法，并且能够通过这类企业Bean的方法返回这些对象的引用。另外，必须能够将基于RMI-IIOP 的企业Bean的Home或Remot接口的引用传递给RMI-IIOP或Java IDL对象的某个方法，或是能够从RMI-IIOP或Java IDL对象返回这类企业Bean对象的引用。

在同时包含EJB容器和Web容器的Java EE产品中，要求这两种容器都支持对本地企业Bean的访问。不支持应用程序客户端容器或Applet容器访问本地企业Bean。

6.4 Servlet 3.0 标准

Servlet规范定义了Web应用程序的打包和部署是否可以单独进行，或作为Java EE应用程序的组成部分。Servlet规范也阐述了独立打包部署和嵌入Java EE平台的安全问题。这些可选的Servlet组件是Java EE平台的标准。

Servlet规范包含了对Web容器的附加标准，这些容器是Java EE产品的组成部分，并且Java EE产品也必须满足这些标准。

Servlet规范定义了分布式Web应用程序。为了支持分布式的Java EE应用程序，本规范增加了下述标准。

Web容器必须支持Java EE分布式Web应用程序，使用setAttribute或putValue方法将任意下列类型的对象(如果Java EE产品支持)放入javax.servlet.http.HttpSession对象中:

java.io.Serializable

javax.ejb.EJBObject

javax.ejb.EJBHome

javax.ejb.EJBLocalObject

javax.ejb.EJBLocalHome

javax.transaction.UserTransaction

java:comp/env上下文的 javax.naming.Context对象

一个EJB本地或远程业务接口的引用

Web容器也可以支持其它类型的对象。如果Java EE分布式会话对应的HttpSession对象中的setAttribute或putValue方法接收的对象不是上述类型，也不是任何Web容器支持 的类型，那么容器必须抛出java.lang.IllegalArgumentException异常。这个异常告诉程序员Web容器不支持在虚拟机间传 送此对象。支持多虚拟机操作的Web容器必须确保，当一个

会话从一个虚拟机传到另一个时，所有合法类型的对象能正确地在目标虚拟机上重建。

Servlet规范将访问本地企业Bean定义为一个可选特性。本规范要求，同时包含Web容器和EJB容器的所有Java EE产品支持从Web容器访问本地企业Bean。

Servlet规范参见 http://java.sun.com/products/servlet

6.5 JavaServer Pages™ (JSP) 2.2 标准

JSP规范依赖并构建在Servlet框架上。Java EE产品必须完全支持JSP规范。

JSP规范参见 http://java.sun.com/products/jsp

6.6 Expression Language (EL) 2.2 标准

表达式语言规范以前是JSP规范的一部分。现在它分离出来有了自己规范，因此可以独立于JSP使用。Java EE产品必须支持表达式语言。

表达式语言规范参见 http://jcp.org/en/jsr/detail?id=245

6.7 Java™ Message Service (JMS) 1.1 标准

Java消息服务提供方必须包含在Java EE产品中。JMS的实现必须同时支持JMS“点对点”和“发布-订阅”消息发送功能，要让使这些功能可用，需要使用ConnectionFactory和Destination API。

JMS规范定义了几个接口，用于整合到应用程序服务器。Java EE产品不需要提供实现了这些接口的对象，并且可移植的Java EE产品不能使用下列接口:

javax.jms.ServerSession

javax.jms.ServerSessionPool

javax.jms.ConnectionConsumer

所有javax.jms XA接口

下面的方法只能被运行在应用程序客户端容器中的应用程序组件使用:

javax.jms.Session的setMessageListener方法

javax.jms.Session的getMessageListener方法

javax.jms.Session的run方法

javax.jms.QueueConnection的createConnectionConsumer方法

javax.jms.TopicConnection的createConnectionConsumer方法

javax.jms.TopicConnection的createDurableConnectionConsumer方法

javax.jms.MessageConsumer的getMessageListener方法

javax.jms.MessageConsumer的setMessageListener方法

javax.jms.Connection的setExceptionListener方法

javax.jms.Connection的stop方法

javax.jms.Connection的setClientID方法

如果应用程序组件违反了这些限制，Java EE容器可以抛出JMSException异常(如果方法允许)。

Web和EJB容器中的应用程序组件不能试图为每个连接创建多个活动的Session对象(没有关闭)。当那个连接存在一个活动的Session对 象时，容器应该禁止尝

试使用Connection对象的createSession方法。如果应用程序组件违反了此限制，容器可以抛出 JMSException异常。应用程序客户端容器必须支持在一个连接上创建多个会话。

一般而言，JMS提供方在EJB容器和Web容器中的行为应该是一致的。EJB规范描述了在EJB容器中使用JMS的约束，以及在EJB容器中多事务JMS的交互。运行在Web容器中的应用程序应该遵循相同的约束。

JMS规范参见 http://java.sun.com/products/jms

6.8 Java™ Transaction API (JTA) 1.1 标准

JTA定义了UserTransaction接口，它被应用程序用来启动，提交或中止事务。应用程序组件可以获取UserTransaction对 象，这需要通过JNDI名称java:comp/UserTransaction 进行查找，或者请求UserTransaction对象的注入。

JTA也定义了TransactionSynchronizationRegistry接口，它可以被系统级组件使用，比如用在持久化管理器与事务 管理器的交互中。这些组件可以获取TransactionSynchronizationRegistry 对象，这需要通过JNDI名称java:comp/TransactionSynchronizationRegistry进行查找，或者请求TransactionSynchronizationRegistry对象的注入。

JTA定义的一些接口被应用程序服务器用来与事务管理器进行通信，然后由事务管理器与资源管理器进行交互。这些接口必须得到支持，正如连接器规范所描述的。另外，Java EE产品可以显式地为应用程序提供其它事务功能的支持。

JTA规范参见 http://java.sun.com/products/jta

6.9 JavaMail™ 1.4 标准

JavaMail API允许访问消息存储层中的邮件消息，也允许使用消息输送层来创建和发送邮件消息。互联网标准的MIME消息需要包含特殊的支持。访问消息存储层和消息 输送层需要通过协议提供方支持的特定存储和输送协议。JavaMail API规范不要求任何特殊的协议提供方，但是JavaMail引用的实现包含一个IMAP消息存储提供方，一个POP3消息存储提供方和一个SMTP消息 输送提供方。

通常是在一个Properties对象的属性中对JavaMail API进行配置，这个对象用来创建javax.mail.Session对象，这需要使用一个静态的工厂方法。为了使Java EE平台可以配置和管理JavaMail API的会话，使用这个JavaMail API的应用程序组件应该使用JNDI请求一个Session对象，并且应该在它的部署描述符中使用resource-ref元素列出这个对象的需求，或 者也可以使用Resource注解。JavaMail API Session对象应当看作是一个资源工厂，正如5.7,“资源管理器连接工厂的引用”所描述的。本规范要求Java EE平台支持javax.mail.Session对象作为资源工厂。

Java EE平台提供的消息输送层必须能够处理javax.mail.internet.InternetAddress类型的地址和javax.mail.internet.MimeMessage类型的消息。必须正确地配置默认的消息输送系统，并使用 javax.mail.Transport类的send方法来发送这类消息。默认的输送层所需的任何验证必须得到处理，而不需要应用程序提供javax.mail.Authenticator或显式地连接到输送层来获取验证信息。

本规范不要求Java EE产品支持任何消息存储协议。

注意，JavaMail API创建线程来递交Store，Folder，和Transport事件的通知。这些通知功能的使用受到各种容器在线程使用约束上的影响。例如，通常不能在EJB容器中创建线程。

JavaMail API使用JavaBean激活框架API来支持各种MIME数据类型。JavaMail API必须包含处理下列MIME数据类型
的javax.activation.DataContentHandlers，它们对应的Java编程语言类型 标明在表 6-4中。

表 6-4 JavaMail API MIME数据类型映射的Java类型

MIME类型	Java类型
text/plain	java.lang.String
text/html	java.lang.String
text/xml	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
message/rfc822	javax.mail.internet.MimeMessage

JavaMail API规范参见 http://java.sun.com/products/javamail

6.10 Java EE™连接器体系结构1.6标准

在整个Java EE产品中，所有EJB容器和所有Web容器都必须支持整套连接器API。所有这种容器必须支持任何指定了事务功能的资源适配器。Java EE部署工具必须支持资源适配器的部署，正如连接器规范所定义的，并且必须支持使用了资源适配器的应用程序的部署。

连接器规范参见 http://java.sun.com/j2ee/connector/

6.11 Java EE Web服务1.3标准

Java EE Web服务规范定义的功能要求Java EE应用程序服务器必须支持Web服务终端的部署。定义完整的部署模型需要包含一些新的部署描述符。所有Java EE产品必须支持Web服务的部署和执行，正如Java EE Web服务规范(JSR-109)所指定的。

Java EE Web服务规范参见 http://jcp.org/en/jsr/detail?id=109

6.12 Java™ API for XML-based RPC (JAX-RPC) 1.1标准 (推荐可选)

JAX-RPC规范定义了用于访问Web服务的客户端API以及实现Web服务终端的技术。Java EE Web服务规范描述了基于JAX-RPC的服务和客户端的部署。EJB和Servlet规范也描述了这类部署的相关问题。必须能够使用这些部署模型来部署 基于JAX-RPC的应用程序。

JAX-RPC规范描述了对消息处理器的支持，它们可以处理消息请求和响应。一般而言，这些消息处理器执行在同一容器中，其权限和执行上下文与 JAX-RPC客户端或终端组件相同，并与之关联的。这些消息处理器访问的JNDI命名空间java:comp/env，与它们关联的组件相同。如果支持 自定义的序列化和反序列化，会用与消息处理器相同的方式对待它们。

注意，JAX-RPC服务终端和处理器既不支持Web服务注解也不支持注入。鼓励新的应用程序使用JAX-WS来利用这些新功能，以简化Web服务的开发。

JAX-RPC规范参见 http://java.sun.com/Webservices/jaxrpc

6.13 Java™ API for XML Web Services (JAX-WS) 2.2 标准

JAX-WS规范提供对Web服务的支持，并使用JAXB API将XML数据绑定到Java对象。 JAX-WS 规范定义了访问Web服务的客户端API以及实现Web服务终端的技术。Java EE Web服务规范描述了基于JAX-WS的服务和客户端的部署。EJB和Servlet规范也描述了这类部署的相关问题。必须能够使用这些部署模型来部署基 于JAX-WS的应用程序。

JAX-WS规范描述了对消息处理器的支持，它们可以处理消息请求和应答。一般而言，这些消息处理器执行在同一容器中，其权限和执行上下文与 JAVA-WS客户端或终端组件相同，并与之关联。这些消息处理器访问的JNDI命名空间java:comp/env，与它们关联的组件相同。如果支持自 定义的序列化和反序列化，会用与消息处理器相同的方式对待它们。

JAX-WS规范参见 http://java.sun.com/Webservices/jaxws

6.14 Java™ API for RESTful Web Services (JAX-RS) 1.1 标准

JAX-RS定义了部署Web服务的API，这些Web服务根据Representational State Transfer (REST)体系风格构建。

在整个Java EE产品中，要求所有Java EE Web容器支持使用JAX-RS技术的应用程序。

此规范描述了作为Servlet对服务进行部署。必须能够使用相应的部署模型来部署基于JAX-RS的应用程序，这种部署模型使用了web.xml 描述符的servlet-class元素，它的名称是应用程序提供的JAX-RS ApplicationConfig抽象类的扩展类。

此规范定义了一套可选的容器管理的功能和资源，它们会在Java EE容器中使用，所有这样的特性和资源必须可用。

JAX-RS规范参见 http://jcp.org/en/jsr/detail?id=311

6.15 Java™ Architecture for XML Binding (JAXB) 2.2 标准

Java Architecture for XML Binding (JAXB) 提供了一种简便的方式将XML Schema绑定到Java语言程序。JAXB可以独立使用，也可以与JAX-WS进行组合，它为Web消息服务提供了一种标准的数据绑定。在整个 Java EE产品中，要求所有的Java EE应用程序客户端容器，Web容器和EJB容器支持JAXB API。

Java API for XML Data Binding规范参见
http://java.sun.com/webservices/jaxb

6.16 Java™ API for XML Registries (JAXR) 1.0 标准 (推荐可选)

JAXR规范为客户端定义了用于访问基于XML的注册表的API，比如，ebXML注册表和UDDI注册表。支持JAXR的Java EE产品必须包含一个JAXR注册表提供方，它至少满足JAXR 0级标准。

JAXR规范参见 http://java.sun.com/xml/jaxr

6.17 Java™ Platform, Enterprise Edition Management API 1.1 标准

Java EE Management API 为部署工具提供API来查询Java EE应用程序服务器，并确定它的当前状况，已部署的应用程序等等。所有Java EE产品必须支持它的规范中描述的这种API。

Java EE Management API规范参见 http://jcp.org/jsr/detail/77.jsp

6.18 Java™ Platform, Enterprise Edition Deployment API 1.2 标准 (推荐可选)

Java EE Deployment API定义了部署工具的运行时环境和Java EE应用程序服务器提供的插入式组件之间的接口。这些插入式组件执行在部署工具中，实现了特定Java EE产品的部署机制。要求所有Java EE产品提供这些在工具中使用的插入式组件，它们可以来自其它供应商。

注意，Java EE部署规范没有定义Java EE应用程序直接使用的新的API。然而，必须能够创建作为部署工具的Java EE应用程序，让它提供Java EE部署规范要求的运行时环境。

Java EE Deployment API 规范参见
http://java.sun.com/j2ee/tools/deployment

6.19 Java™ Authorization Service Provider Contract for Containers (JACC) 1.4 标准

JACC规范在Java EE应用程序服务器和授权策略供应商之间定义了一个协议。在整个Java EE产品中，要求所有的Java EE应用程序容器，Web容器和企业Bean容器支持此协议。

JACC规范参见 http://jcp.org/jsr/detail/115.jsp

6.20 Java™ Authentication Service Provider Interface for Containers (JASPIC) 1.0 标准

JASPIC规范定义了一个服务供应商接口(SPI)，通过它，实现消息验证机制的验证提供方可以在运行时集成到客户端或服务器消息处理容器中。用 这个接口集成的验证提供方操作容器提供给它们的网络消息。它们对传出的消息进行加工，使消息源可以被接收它的容器验证，并且消息接收方可以被消息发送方验证。它们验证新的消息并将验证后产生的标识返回给调用它们的容器。

在整个Java EE产品中，要求所有Java EE应用程序容器，Web容器和企业Bean容器支持基线兼容标准，正如JASPIC规范所定义的。在整个Java EE产品中，所有Web容器也必须支持JASPIC规范定义的Servlet Container Profile。

JASPIC规范参见 http://jcp.org/jsr/detail/196.jsp

6.21 Debugging Support for Other Languages (JSR-45) 标准

JSP页面通常被翻译成Java语言类文件。Debugging Support for Other Languages规范描述了包含在类文件中的特殊信息，它将类文件数据关联到最初的源文件数据。要求所有Java EE产品能够在JSP页面生成的类文件中包含这样的信息。

Debugging Support for Other Languages 规范参见
http://jcp.org/en/jsr/detail?id=45

6.22 Standard Tag Library for JavaServer Pages™ (JSTL) 1.2 标准

JSTL 定义了一个标准标签库，用来简化JSP页面的开发。要求所有Java EE产品提供所有JSP页面都能使用的JSTL。

Standard Tag Library for JavaServer Pages规范参见
http://jcp.org/en/jsr/detail?id=52

6.23 Web Services Metadata for the Java™ Platform 2.1 标准

Java平台Web服务元数据规范定义了可用来简化Web服务部署工作的Java语言注解。这些注解可以和JAX-WS服务组件一起使用。

Web Services Metadata for the Java Platform规范参见
http://jcp.org/en/jsr/detail?id=181

6.24 JavaServer Faces™ 2.0 标准

JavaServer Faces 技术为JavaServer应用程序简化了用户界面的开发。不同技能水平的开发者可以快速构建Web应用程序，通过：在页面中组装可重复使用的UI组件;将这些组件连接到应用程序数据源;并且将客户端产生的事件联系到

服务器端事件处理器。在整个Java EE平台中，要求所有Java EE Web容器支持使用JavaServer Faces技术的应用程序。

JavaServer Faces规范参见 http://jcp.org/en/jsr/detail?id=252

6.25 Java™平台公共注解 1.1 标准

公共注解规范定义了其它某些规范使用的Java语言注解，包括本规范。使用这些注解的规范完整地定义了这些注解的标准。Applet容器不需要支持 任何这样的注解。所有其它容器必须为所有这类注解提供定义，并且必须支持这些注解的语义，它们描述在对应的规范中，下表对它们进行了总结。

表 6-5 容器支持的公共注解

注解	App Client	Web	EJB
Resource	Y	Y	Y
Resources	Y	Y	Y
PostConstruct	Y	Y	Y
PreDestroy	Y	Y	Y
Generated	N	N	N
RunAs	N	Y	Y
DeclareRoles	N	Y	Y
RolesAllowed	N	Y	Y
PermitAll	N	Y	Y
DenyAll	N	Y	Y

Common Annotations for the Java Platform 规范参见
http://jcp.org/en/jsr/detail?id=250

6.26 Java™ Persistence API 2.0 标准

Java Persistence是一种标准的API，用于管理持久化和对象/关系映射。Java持久化规范提供了一种对象/关系映射功能，帮助应用程序开发者使用 Java域模型来管理关系数据库。要求在Java EE中支持Java持久化。它也可以用在Java SE环境中。

按照Java持久化规范的委任机制，在Java EE环境中，应用程序类加载器及其双亲类加载器不应该加载持久化单元的类，直到创建了持久化单元的实体管理器工厂。

Java持久化规范由EJB专家组制定，参见
http://jcp.org/en/jsr/detail?id=220

6.27 Bean Validation 1.0 标准

Bean Validation规范定义了为JavaBean检验定义了一种元数据模型和API。注解是默认的元数据源，通过使用XML检验描述符，可以重写和扩展元数据。

Java EE平台要求Web容器让一个ValidatorFactory实例对JSF实现可用，这需要将它保存在一个叫做java.faces.validator.beanValidator.ValidatorFactory的Servlet上下文环境属性中。

Java EE平台也要求ValidatorFactory实例对JPA提供方可用，它将作为Map中的属性传给PersistenceProvider接口的createContainerEntityManagerFactory(PersistenceUnitInfo,Map)方法的第二个参数，这个接 口在javax.persistence.validation.factory包中。

Bean Validation 规范参见http://jcp.org/en/jsr/detail?id=303

6.28 Managed Beans 1.0 标准

Managed Beans 规范定义了一个轻量级的组件模型，它支持Java平台现有的基本的生命周期模型，资源注入功能和拦截器服务。

Managed Beans规范参见 http://jcp.org/en/jsr/detail?id=316

6.29 Interceptors 1.1 标准

拦截器规范使拦截器功能得以更普遍地使用，起初它们定义在EJB 3.0规范中。

拦截器规范参见 http://jcp.org/en/jsr/detail?id=318

6.30 Contexts and Dependency Injection for the Java EE Platform 1.0 标准

CDI (JSR-299) 定义了一套上下文相关的服务，这些服务由Java EE容器提供，目的在于简化同时使用Web层和业务层的应用程序的创建。

CDI 规范参见 http://jcp.org/en/jsr/detail?id=299

6.31 Dependency Injection for Java 1.0标准

Dependency Injection for Java (JSR-330)为可注入的类定义了一套标准的注解(和一个接口)。

在Java EE平台中，对依赖注入的支持由CDI调节。更具体地说，DI注入点只活动在Bean存档中，正如CDI所规定的。详细信息请查看5.20,“支持依赖注入(JSR-330)”。

DI规范参见 http://jcp.org/en/jsr/detail?id=330

第7章 互用性

本章描述Java™平台企业版(Java EE)互用性标准。

7.1 互用性介绍

Java EE平台用于支持多种不同类型客户端的企业环境。这类企业环境会在现有的企业信息系统(EIS)中添加新的服务。它们会使用多种硬件平台以及多种语言编写的应用程序。

特别是，企业环境中的Java EE平台可以用来联合以下任何类别的应用程序:

- 用C++和Visual Basic这样的语言编写的应用程序。
- 运行在个人电脑平台或Unix工作站上的应用程序。
- Java EE平台不直接支持的独立的Java技术应用程序。

本章阐述了Java EE平台互用性标准，它能够为各种类型的客户端，不同的硬件平台和多种软件程序提供间接的支持。Java EE平台的互用特性让不同的底层系统可以无缝地协同工作，并隐藏了将这些部件连接到一起的复杂性。

当前发布的Java EE平台互用性标准允许:

- Java EE应用程序使用CORBA或低层socket接口连接到旧的系统。

Java EE应用程序跨多种Java EE产品连接到它们的应用程序，不论这些应用程序是来自不同的产品供应商，还是同一个供应商不同的Java EE平台。

在这个版本的规范中，运行在不同平台中的Java EE应用程序之间的互用性可以通过HTTP协议(可能附加SSL)，或基于IIOP的EJB互用性协议来实现。

7.2 互用性协议

本规范要求Java EE产品支持一套标准的协议和格式，以确保Java EE应用程序之间，以及与其它也实现了这些协议和格式的应用程序之间的交互性。此规范要求支持下列协议和格式组:

Internet和Web协议

OMG协议

Java技术协议

数据格式

这些协议和格式中大多数都得到了Java SE甚至是底层操作系统的支持。

7.2.1 Internet和Web协议

基于标准的Internet协议是平台不同部件通信的途径。Java EE平台通常支持下述Internet协议，相应的技术规范中有详细的描述:

TCP/IP协议族--它是互联网通信的核心组件。TCP/IP和UDP/IP是互联网标准的传输协议。Java SE和底层操作系统都支持 TCP/IP协议。

HTTP 1.1--它是Web通信的核心协议。与 TCP/IP 一样，HTTP1.1被Java SE和底层操作系统支持。Java EE Web容器必须能够在标准的HTTP 80端口上发布它的HTTP服务。

SSL 3.0, TLS 1.0--SSL 3.0 (Secure Socket Layer) 象征了Web通信的安全层次。使用https URL而不是http URL时，就会间接地使用到它。Java EE Web容器必须能够在标准的HTTPS 443端口上发布它的HTTPS服务。EJB规范也要求使用SSL 3.0和TLS 1.0作为EJB互用性协议的组成部分。

SOAP 1.1--SOAP是交换XML消息的表示层协议。要求在HTTP层上支持SOAP层，正如JAX-RPC和JAX-WS规范中所描述的。

SOAP 1.2--SOAP 1.2是由W3C标准化并且被JAX-WS支持的SOAP协议版本。

WS-I Basic Profile 1.1--WS-I Basic Profile结合Simple SOAP Binding Profile以及Attachment Profile，描述了使用SOAP1.1，WSDL1.1和带附件的MIME SOAP协议的互用性标准。它是JAX-RPC和JAX-WS规范所要求的。

7.2.2 OMG协议

本规范要求完整的Java EE产品支持下述基于Object Management Group (OMG)的协议:

IIOP (Internet Inter-ORB Protocol)--由Java SE中的Java IDL和RMI-IIOP支持。Java IDL提供基于标准的交互性和连通性，使用公共对象请求代理体系(CORBA)。CORBA指定的对象请求代理(ORB)允许不同地点的应用程序相互通 信。这种交互性通过IIOP完成，通常建立在内部网络环境。使用RMI-IIOP技术，IIOP可以用作RMI协议。IIOP定义在CORBA

2.3.1规范的13章和15章，参见http://cgi.omg.org/cgi-bin/doc?formal/99-10-07了解详细信息。

EJB互用性协议--EJB互用性协议基于IIOP (GIOP 1.2)和CSv2 CORBA安全互用性规范。EJB互用性协议定义在EJB规范中。

CORBA可互用命名服务(INS)协议--基于COSNaming的INS协议是一种基于IIOP的协议，用于访问命名服务。EJB互用性 协议要求通过INS协议来完成JNDI API对EJB对象的查找。另外，必须能够使用Java IDL COSNaming API来访问INS命名服务。所有Java EE产品必须提供一个满足Interoperable Naming Service(INS)规范要求的命名服务，参见http://cgi.omg.org/cgi-bin/doc?formal/2000-06-19了解详细信息。提供的这个命名服务可以作为独立的命名服务器，也可以作为通向另一个命名服务的协议桥或网关。这两种途径都符合本规范。

7.2.3 Java技术协议

本规范要求Java EE平台支持JRMP协议，它是Java技术特有的远程方法调用(RMI)协议。JRMP是Java SE的必须组件，并且是两个必须的RMI协议之一(IIOP是另一个必须的RMI协议)。

JRMP是Java编程语言的分布式对象模型。分布式系统(运行在不同地址空间，常常是不同主机)必须能够相互通信。JRMP允许不同地址空间中的程序级对象使用Java编程语言对象模型的语义来调用远程对象。

详细信息参见JRMP规范
http://java.sun.com/javase/6/docs/technotes/guides/rmi

7.2.4 数据格式

除了组件通信协议，本规范还要求Java EE平台支持一些数据格式。定义的这些格式用于完成组件间的数据交换。

下面的数据格式必须得到支持:

XML 1.0--XML格式可以用于结构化文档，PRC消息等等。JAXP API支持对XML格式数据的处理。JAX-RPC API支持XML RPC消息，以及Java类和XML间的映射。

HTML 3.2-它代表Web浏览器最低的标准文档格式。虽然没有被Java EE API直接支持，但是Java EE Web客户端必须能够显示HTML 3.2文档。

图像文件格式--Java EE平台必须支持GIF, JPEG和PNG图像。对这些格式的支持由java.awt.image API (参见URL: http://java.sun.com/javase/6/docs/api/java/awt/image/package-summary.html)和Java EE Web客户端提供。

JAR文件--JAR (Java存档)文件是基于Java技术的应用程序组件的标准打包格式，还包括专用的ejb-jar格式，Web应用存档(WAR)格式，资源适配存档 (RAR)和Java EE企业程序存档(EAR)格式。JAR是平台独立的文件格式，它允许将大量文件聚集到一个文件中。 它可以捆绑多个Java组件并在一次HTTP事务中下载到浏览器。JAR文件格式由java.util.jar和java.util.zip包支持。有关 JAR规范的详细信息，参见http://java.sun.com/javase/6/docs/technotes/guides/jar

类文件格式--类文件格式定义在Java虚拟机规范中。每个类文件包含一个Java编程语言类型(可以是一个类或接口，并且由8位字节流组成)。有关类文件格式的详细信息，参见
http://java.sun.com/docs/books/jvms/index.html

第8章 应用程序组装者和部署

本章指定的Java™平台企业版(Java EE)标准用于组装，打包和部署Java EE应用程序。这些标准的主要目的是提供可扩展和模块化的应用程序的组装，以及在Java EE产品中对可移植的Java EE产品的部署。

JavaEE应用程序由一个或多个Java EE组件和一个可选的Java EE部署描述符组成。如果应用了这个部署描述符，它会将应用程序组件按模块列出。如果没有这个部署描述符，就按默认的命名规则找出应用程序模块。Java EE模块代表了Java EE应用程序结构的基本单元。Java EE模块由一个或多个Java EE组件和一个可选的模块级部署描述符组成。Java EE组件模块功能的灵活性和可扩展性有助于Java EE组件按单个组件，组件库或应用程序打包和部署。

完整的Java EE产品必须支持本章描述的所有功能。Java EE Profile可以仅支持Java EE模块类型的子集。如果某项特殊的Java EE Profile上的产品不支持某种模块类型，那么任何与此模块类型有关的标准就应当被视为不适于这样的产品。

图 8-1展示了Java EE部署单元的结构模型，并且包含可选的候补部署描述符，应用程序包用它来维护任何原先Java EE模块的数字签名。

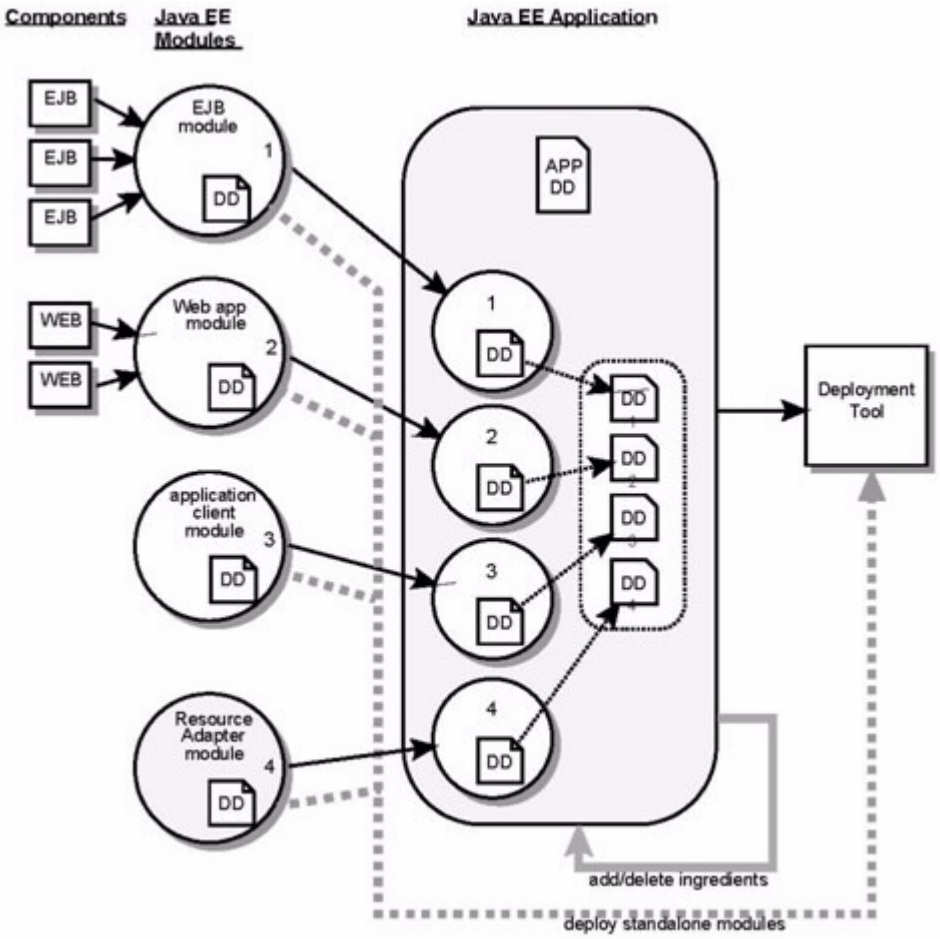

图 8-1 Java EE的部署

8.1 应用程序部署的生命周期

Java EE应用程序部署的生命周期开始于Java EE组件各自的创建。接着，这些组件会与一个模块级的部署描述符一起打包，以创建一个Java EE模块。Java EE模块可以作为独立单元进行部署，也可以和Java EE应用程序部署描述符一起组装，成为一个Java EE应用程序。

图 8-2 展示了Java EE应用程序的生命周期

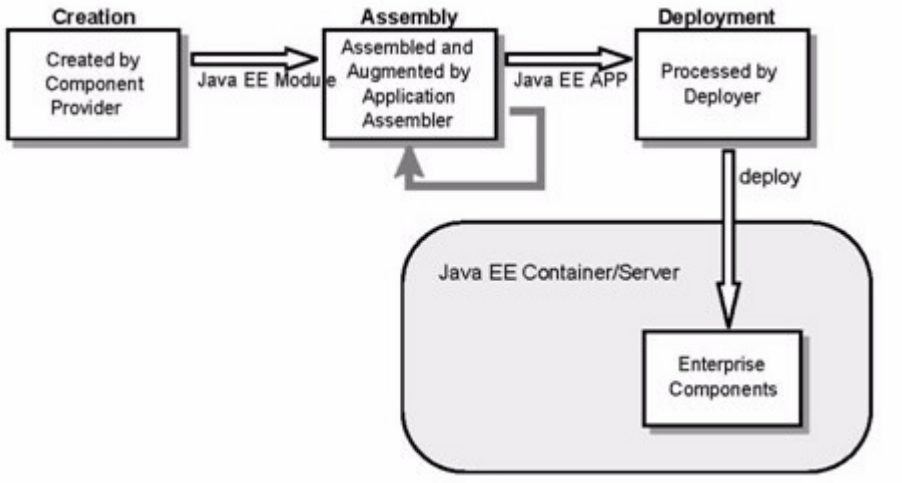

图 8-2 Java EE应用程序的生命周期

8.1.1 创建组件

EJB, Servlet, 应用程序客户端和连接器规范包含了与模块级部署描述符相关的XML Schema定义和必要的打包体系，用于生产Java EE模块。(应用程序客户端规范参见10)

Java EE模块是一个或多个相同类型的Java EE组件(Web, EJB, 应用程序客户端或连接器)以及一个可选的该类型的模块部署描述符组成的集合。同一容器类型的多个组件可以与适合那个容器类型的单个Java EE部署描述符一起打包，产生一个Java EE模块。不同容器类型的组件不可以混合在同一个Java EE模块中。

Java EE模块代表了Java EE程序结构的基本单元。一些情况下，单个Java EE模块(不一定包含在Java EE程序包中)就是一个完整的应用程序。另一些情况下，一个应用程序可以由多个Java EE模块构成。

Java EE模块的部署描述符包含了在模块中部署组件的必要的数据声明。Java EE模块的部署描述符也包含了组装说明，描述了组件怎样构成应用程序。

从Java EE 5开始，Web应用程序模块，企业Bean模块或应用程序客户端模块不需要包含部署描述符。取而代之，可以将部署信息写入模块类文件的注解中。

从Java EE 5开始，Java EE企业程序存档不需要包含部署描述符。取而代之，部署信息可以使用嵌入式模块默认的命名规则来确定。

单个Java EE模块可以作为独立的Java EE模块部署，成为一个有效的Java EE应用程序，而无需应用程序级部署描述符。

Java EE模块可以表达自己所依赖的库，正如后面的8.2,“库的支持”中所描述的。

所有Java EE模块都有一个名称。这个名称可以在模块的部署描述符中显式地设定。如果没有设定，这个模块的名称就是它在.ear文件中的路径名称，并除去后缀名 (.jar, .war, .rar)，但是需要包含它的目录名称。在一个应用程序中，一个模块的名称必须是唯一的。当且仅当这个名称不是唯一时(例如，除去不同的后缀名后，两个名 称相同)，部署工具才会为任何冲突的模块选择新的唯一的名称; 模块名称没有冲突就不能更换。在这种情况下，唯一名称的选择算法取决于具体的产品 。依赖模块名称的应用程序必须确保它们的模块名称是唯一的。

例如，某个应用程序具有如下结构:

myapp.ear

inventory.jar

ui.war

它有一个默认的应用程序名称"myapp"，并定义了两个模块，它们的默认名称是"inventory" 和"ui"。

另一个应用程序具有如下结构:

bigapp.ear

ejbs

inventory.jar

accounts.jar

ui

store.war

admin.war

它有一个默认的应用程序名称"bigapp"，并定义了四个模块，它们的默认名称是"ejbs/inventory"，"ejbs/accounts"，"ui/store"和 "ui/admin"

8.1.2 应用程序组装

Java EE应用程序可以由一个或多个Java EE模块以及一个Java EE应用程序部署描述符组成。Java EE应用程序以Java存档(JAR)文件的形式打包，后缀名为.ear(企业存档)。Java EE应用程序包最少要包含Java EE模块和应用程序部署描述符。Java EE应用程序包也可以包含Java EE模块引用的库(使用下面EE8.2,”库的支持”中描述的Class-Path机制)，帮助文件和程序文档来辅助部署者。

可移植的Java EE应用程序的部署不应该依赖于可能包含在包中任何实体，而应该是那些本规范所定义的。可移植的Java EE应用程序的部署必须有可能只使用应用程序部署描述符和Java EE模块(和它们依赖的库)以及模块的描述符。

Java EE应用程序部署描述符是Java EE应用程序内容的顶级视图。Java EE应用程序部署描述符由一个XML Schema或DTD指定(参见8.6,“Java EE应用程序XML Schema”)。

在某些情况下，Java EE应用程序在它部署到企业之前需要进行用户化定制。可以在应用程序中添加新的Java EE模块。已有的模块可以从应用程序中移除。一些Java EE模块可能需要创建，改变或更换内容。例如，应用程序用户可能需要使用HTML编辑器来将公司图案添加到Java EE Web应用程序提供的模板登录页。

所有Java EE应用程序都有一个名称。可以在应用程序部署描述符中显式地设定这个名称。如果没有设定，这个应用程序的名称就是ear文件的基本名称，不带有任 何.ear后缀名和目录名。在应用程序服务器实例中，应用程序名称必须是唯一的。如果试图部署一个与已部署的应用程序名称冲突的应用程序，部署工具可以为 这个应用程序选择一个新的唯一的名称。部署工具也允许在部署时指定一个不同的名称。部署工具可以使用产品特有的方式来判断是在部署一个新的应用程序(它的 名称必须唯一)，还是重新部署一个已有的应用程序(它的名称可以匹配)。

类似地，当部署一个独立的模块时，这个模块名称可以用作应用程序的名称，并且遵循相同的命名规则。模块名称可以在模块部署描述符中显式地设定。如果没有设定，这个模块的名称就是它的模块文件的基本名称，不带有任何后缀名(.war, .jar, .rar)和目录名。

8.1.3 部署

在应用程序生命周期的部署阶段，应用程序会被安装到Java EE平台上，然后配置和集成到已有的基础结构中。应用程序部署描述符中列出(或使用下面描述的默认规则找出)的每个Java EE模块必须依据各自模块类型规范的标准进行部署。每个列出的模块必须安装在恰当的容器类型中，并且必须在目标容器中为每个模块设定恰当的环境参数，以反 映出部署描述符元素为每个组件声明的值。

8.2 库的支持

Java EE平台为应用程序使用可选包和共享库(以后都称为库)提供了一些机制。库可以和应用程序绑定，也可以独立安装，供任何应用程序使用。

要求Java EE产品支持使用绑定和安装的库，正如扩展机制体系，可选包版本规范 (参见http://java.sun.com/javase/6/docs/technotes/guides/extensions)和JAR文件规范(参见 http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html)所描述的。使用这个机制，Java EE JAR文件可以引用工具类，其它共享类，独立的.jar文件中资源，同一个Java EE应用程序包的目录中的资源，以及之前安装在Java EE容器中的资源。

8.2.1 绑定的库

和应用程序绑定的库可以按下述方式引用:

1.JAR格式文件(比如，.jar文件，.war文件或.rar文件)可以引用一个.jar文件或目录，这需要在引用方JAR文件的 Manifest文件中给Class-Path头元素命名要引用的.jar文件或目录。可以将引用方JAR文件的URL的相对路径作为要引用的.jar或 目录的URL名称。Manifest文件在JAR文件中的名称是META-INF/MANIFEST.MF。Manifest文件中的Class- Path入口的形式如下

```
Class-Path: list-of-jar-files-or-directories-separated-by-spaces
```

(有关Class-Path头元素句法的重要细节和限制，参见JAR文件规范)。在处理Java EE模块时，Java EE部署工具必须处理所有这类被引用的文件和目录。当处理引用方.jar文件时，必须忽略任何要引用的.jar文件中的部署描述符。部署工具安装.jar 文件时必须维护引用文件间的相对位置关系。通常安装.jar文件的目录层次应匹配最初的应用程序目录层次。所有被引用的.jar文件或目录必须在运行时出 现在引用方JAR文件的逻辑路径中。

只有包含了类文件或资源的JAR格式的文件或目录可以作为Class-Path引用的目标，它们由标准的类加载器直接加载；这样的文件常常使 用.jar作为后缀名。由部署工具处理的顶层JAR文件不应该具有Class-Path入口；按定义，这类入口的作用是将其它外部文件引用到部署单元。不 要求部署工具处理这类外部引用。

2.一个.ear文件可以包含一目录来存放打包成JAR文件的库。这个.ear文件部署描述符的library-directory元素记录了这个 目录的名称。如果没有library-directory元素，或这个.ear文件中没有包含部署描述符，那么就使用lib作为目录名称。如果使用空的 library-directory元素，就表示没有库目录。

这个目录(不包括子目录)中所有带有.jar后缀的文件必须对这EAR包中的所有组件可用，包括应用程序客户端。通过使用这里描述的任何技术，这些库也可以引用其它库，既可以是应用程序捆绑的，也可以是独立安装的。

3.一个Web应用程序可以在WEB-INF/lib目录中存放库。详细信息参见Servlet规范。通过使用这里描述的任何技术，这些库可以也引用其它库，既可以是应用程序捆绑的，也可以是独立安装的。

8.2.2 安装的库

可以按下述方式引用独立安装的库(简称安装库):

1. 所有类型的JAR格式文件可以在它们的Manifest文件中包含一个Extension-List属性，指出对一个安装库的依赖。JAR文件规范定义了 由Applet使用的这类属性的语义；本规范要求所有组件类型和相应的JAR格式文件都支持这类属性。要求部署工具检查这类依赖信息，并拒绝部署任何没有 满足依赖关系的组件。可移植的应用程序不应该假定任何安装库会对组件可用，除非这个组件的JAR格式文件，或里面包含的JAR格式文件，使用了 Extension-List及相关属性表达出对这个库的依赖。这个被引用的库必须对引用方文件中的所有组件可用，包括引用方文件中的其它JAR格式文件 中的任何组件。例如，如果一个.ear文件引用了一个安装库，这个库必须对这个.ear文件中的所有.war文件，EJB的.jar文件，应用程序 的.jar文件和资源适配器的.rar文件中的所有组件可用。

不要求Java EE产品在部署时或运行时支持库的下载(使用<extension>-Implementation-URL 头元素)。也不要求Java EE产品同时支持某个安装库的多种版本。不要求Java EE产品限制对库的访问，不仅是它们所依赖的；可以让应用程序访问到更多的安装库，而不只是它所请求的。所有这些情况中，强烈推荐提供这样的支持，并且在 本规范的未来版本中可能会作出要求。特别是，我们推荐Java EE产品支持一个安装库的多种版本，并且默认只允许应用程序访问它们所依赖的安装库。

8.2.3 库的冲突

如果应用程序与一个库的某个版本绑定，并且同样的库作为安装库存在，那么这个与应用程序绑定的库的实例应该作为这种库的安装版的偏好。这使应用程序 可以正确地绑定它所需要的版本，而不会被任何安装库所影响。注意，如果这个库也是当前Java EE平台版本的必须组件，并且应用程序将被部署到这个平台，那么这么个平台版本可能(通常会)优先选择它。

8.2.4 库的资源

除了允许访问被引用的类，正如上面描述的，任何包含在被引用的JAR文件中的资源也必须是可访问的，可以使用Class和ClassLoader的 getResource方法获取，并且要在应用程序的安全权限允许下。应用程序通常会具有必要的安全权限来访问应用程序中的任何JAR文件中的资源。

8.2.5 动态类加载

需要动态加载类的库必须考虑到Java EE应用程序的类加载环境。库常常由应用程序类加载器的双亲类加载器加载。如果一个库只需要动态加载自己的类文件，那么它可以安全地使用Class类的 forName方法。然而，如果有些库需要动态加载应用程序某个部分中的类，那么它们需要使用上下文类加载器来加载这些类。对上下文类加载器的访问需要具 有RuntimePermission(“getClassLoader”)权限，这一般不会授权给整个应用程序，而应该只授权给需要动态加载类的库。当 访问上下文类加载器时，库可以使用诸如下面的方法来断言它们的特权。这项技术可以工作在Java SE和Java EE平台上。

```
public ClassLoader getContextClassLoader() {
return AccessController.doPrivileged(
new PrivilegedAction<ClassLoader>() {
public ClassLoader run() {
ClassLoader cl = null;
try {
cl = Thread.currentThread(). getContextClassLoader();
} catch (SecurityException ex) { }
return cl;
}
});
}
```

接着，库就可以使用下面的技术来加载类。

```
ClassLoader cl = getContextClassLoader();
if (cl != null) {
try {
clazz = Class.forName(name, false, cl);
} catch (ClassNotFoundException ex) {
clazz = Class.forName(name);
}
} else
clazz = Class.forName(name);
```

8.2.6 例子

下面的例子演示了库绑定机制一个简单用法，它引用的工具类库被两个ejb-jar文件中的企业Bean共享。

```
app1.ear:
META-INF/application.xml
    ejb1.jar        Class-Path: util.jar
ejb2.jar        Class-Path: util.jar
```

util.jar

下一个例子演示了Class-Path机制较复杂的用法。在这个例子中，开发者已经选择将企业Bean客户端视图类打包成一个单独的JAR文件，并 且在其它需要这些类的JAR文件中引用了这个JAR文件。这些类被ejb2.jar需要，它与ejb1.jar在同一个应用程序包中，它也被包含在不同应 用程序中的ejb3.jar和servlet1.jar需要；ejb1.jar自己也需要这些类，因为它们在ejb1.jar中定义了企业Bean的远程 接口，并且开发者已经选择了让这些类可用的引用模型，正如EJB规范所描述的。在ejb1.jar的部署描述符的ejb-client-jar元素中命名了客户端视图JAR文件。

Class-Path机制必须被app3.ear中的组件用来引用客户端视图JAR文件，它相当于app2.ear中的ejb1.jar中的企业Bean。这些企业Bean被ejb3.jar中的企业Bean和webapp.war中的servlet引用。

app2.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

app3.ear:

META-INF/application.xml

ejb1_client.jar

ejb3.jar　　Class-Path: ejb1_client.jar

webapp.war　　Class-Path: ejb1_client.jar

WEB-INF/web.xml

WEB-INF/lib/servlet1.jar

下面的例子演示了库安装机制的一个简单用法，它引用了一个单独安装的工具类库。

app1.ear:

META-INF/application.xml

ejb1.jar:

META-INF/MANIFEST.MF:

Extension-List: util

util-Extension-Name: com/example/util

util-Specification-Version: 1.4

META-INF/ejb-jar.xml

util.jar:

META-INF/MANIFEST.MF:

Extension-Name: com/example/util

Specification-Title: example.com’s util package

Specification-Version: 1.4

Specification-Vendor: example.com

Implementation-Version: build96

8.3 类加载标准

Java EE规范故意没有规定Java EE产品必须使用的类加载器的确切类型和层次。作为代替，此规范定义的标准用来说明什么类必须或不能对组件可见。Java EE产品可以自由使用任何类加载器，只要它满足这些标准。可移植的应用程序不能依赖类加载器的类型或类加载器的层次顺序，如果有的话。应用程序应该使用 8.2.5,“动态类加载”中描述的技术，如果它们需要动态地加载类。

除了下面指定的必须类，Java EE产品必须提供一种方式，使应用程序可以访问安装在应用程序服务器中的类库，即使它没有表达对那个库的依赖。这种方式支持旧应用程序的使用，也支持那种没有使用扩展依赖机制的扩展库的使用。

下面的小节描述了每个容器类型的标准。在任何情况下，对类的访问都受Java语言规则和Java虚拟机控制。在任何情况下，对类和资源的访问都受Java安全模型规则的控制。

8.3.1 Web容器类加载标准

Web容器中的组件必须能够访问下面的类和资源:

在war文件中的WEB-INF/classes目录下的内容。

在war文件中的WEB-INF/lib目录下的所有jar文件的内容，但不包括任何子目录。

任何被上述jar文件引用的库的传递闭包(译者注：传递闭包是指jar包及其所引用的所有jar，以及这些被引用jar所有引用的jar。)(正如8.2,“库的支持”中指定的)。

任何被war文件自己引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

任何被包含它的ear文件指定或引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

在同一个ear文件中的任何资源适配器存档(rar文件)中的所有jar文件中的内容。

单独部署在应用程序服务器上的每个资源适配器存档(rar文件)中的所有jar文件中的内容， 前提是那个资源适配器是用来满足模块中的任何资源引用。

单独部署在应用程序服务器上的每个资源适配器存档(rar文件)中的所有jar文件中的内容，前提是那个rar文件中的任意jar文件使用扩展机制体系满足来自模块的任何引用， (正如8.2,“库的支持”中指定的)。

任何被以上rar文件中的jar文件引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

任何rar文件自己引用的库的传递闭包(正如8.2, “库的支持”中指定的)。

表 6-1中为Web容器指定的Java EE API类。

所有Java SE 6 API类。

Web容器中的组件可以访问下面的类和资源。可移植的应用程序不能依赖对这些类或资源的访问:

同一个ear文件中的任何其它Web模块可访问的类和资源，如上面描述的。

同一个ear文件中任何EJB jar文件的内容。

上面的EJB jar文件指定的任何客户端jar文件的内容。

任何上面的EJB jar文件和客户端jar文件引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

任何单独部署在应用程序服务器上的资源适配器存档(rar文件)中的任何jar文件的内容。

任何被上面的rar文件中的jar文件引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

任何被上面的rar文件自己引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

表 6-1中为容器而不仅是Web容器指定的Java EE API类。

任何在应用程序服务中可用的已安装的类。

应用程序包中的其它类或资源，以及显式使用的本规范没有定义的扩展类和资源。

作为应用程序服务器的实现的组成部分的其它类和资源。

Web容器中的组件不能访问下面的类和资源，除非这样的类或资源涵盖了以上规则之一。

应用程序包中的其它类或资源。例如，应用程序不应该访问应用程序客户端jar文件中的类。

8.3.2 EJB容器类加载标准

EJB容器中的组件必须能够访问下面的类和资源:

EJB jar文件中的内容。

任何EJB jar文件引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

任何包含它的ear文件指定或引用的库的传递闭包(正如8.2,“库的支持”中指定的)。

同一个ear文件中的任何资源适配器存档(rar文件)中的所有jar文件的内容。

单独部署到应用程序服务上的每个资源适配器存档(rar文件)中的所有jar文件中的内容，前提是那个资源适配器被用来维护模块中的任何资源引用。

单独部署到应用程序服务器上每个资源适配器存档(rar文件)中的所有jar文件中的内容，前提是那个rar文件中的任何jar文件使用了扩展机制体系来维护来自模块的引用(正如8.2,“库的支持”中所描述的)。

上面的rar文件中的jar文件引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

这个rar文件自己引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

表 6-1中为EJB容器指定的Java EE API类。

所有Java SE 6 API 类。

EJB容器中的组件可以访问下面的类和资源。可移植的应用程序不能依赖对这些类或资源的访问。

同一ear文件中的任何Web模块可访问的类和资源，正如上面的8.3.1,“Web 容器类加载标准”所描述的。

同一个ear文件中的任何EJB jar文件的内容。

上面的EJB jar文件指定的任何客户端jar文件的内容。

上面的EJB jar文件和客户端jar文件引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

单独部署到应用程序服务器上的任何资源适配器存档(rar文件)中的任何jar文件的内容。

上面的rar文件中的jar文件引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

上面的rar文件自己引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

表 6-1中为容器而不仅仅是EJB容器指定的Java EE API 类。

应用程序服务器中任何可用的安装库。

应用程序包中的其它类或资源，以及显式使用的本规范没有定义的扩展类或资源。

作为应用程序服务器的实现的组成部分的其它类和资源。

EJB容器中的组件不能访问下面的类和资源，除非这样的类或资源涵盖了以上规则之一。

应用程序包中的其它类或资源。例如，应用程序不应该访问应用程序客户端jar文件中的类。

8.3.3 应用程序客户端容器类加载标准

应用程序客户端容器中的组件必须能够访问下面的类和资源:

应用程序客户端jar文件中的内容。

上面的jar文件引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

包含它的ear文件指定或引用的任何库的传递闭包(正如8.2,“库的支持”中所描述的)。

表 6-1中为应用程序客户端容器指定的Java EE API类。

所有Java SE 6 API 类。

应用程序客户端容器中的组件可以访问下面的类和资源。可移植的应用程序不能依赖对这些类或资源的访问。

表 6-1中为容器而不仅仅是应用程序客户端容器指定的Java EE API类。

应用程序服务器中任何可用的安装库。

应用程序包中的其它类或资源，以及显式使用的本规范没有定义的扩展类或资源。

作为应用程序服务器的实现的组成部分的其它类和资源。

应用程序客户端容器中的组件不能访问下面的类和资源，除非这样的类或资源涵盖了以上规则之一。

应用程序包中的其它类或资源。例如，应用程序客户端不应该访问同一个ear文件中的其它应用程序客户端jar文件中的类，也不应该访问同一个ear文件中的Web应用程序或ejb jar文件中的类。

8.3.4 Applet容器类加载标准

为Applet容器定义的这个标准完全由Java SE 6规范指定。本规范没有为Applet容器添加任何新的标准。

8.4 应用程序组装

本节指定了构建Java EE应用程序时通常遵循的步骤及次序。

8.4.1 组装一个Java EE应用程序

1. 选出应用程序要使用的Java EE模块。

2. 创建一个应用程序目录结构。这个应用程序的目录结构可以是随意的，但是通过遵循一些简单的约定可以省去一些部署描述符。这个结构应该围绕它所包含的组件标准而设计。

3. 协调Java EE模块的部署描述符。

必须编辑Java EE模块的部署描述符，从内部链接到要满足的依赖关系，并清除任何多余的安全角色名称。可以使用可选的alt-dd元素(参见8.6,“Java EE应用程序的XML Schema”)来维持最初的部署描述符，如果有必要的话。这个alt-dd元素可以为部署描述符指定一个部署时的替代方案。这个已编辑的部署描述符文件 副本可以保存在应用程序组装者确定的本地目录树中。如果没有使用alt-dd元素，部署者必须从模块包中直接阅读部署描述符。

a. 为应用程序中的模块确定唯一的名称。如果两个模块部署描述符中指定的名称相冲突，那么至少为其中一个模块创建一个候补部署描述符并更换它的名称。如果这个 ear文件中同一个目录下的两个模块有相同的基本名称(比如，foo.jar和foo.war)，那么可以重命名其中一个模块或创建一个候补部署描述符并 为它指定一个唯一的名称。

b. 把应用程序每个模块中的所有组件链接到它们要满足的内部依赖关系。对于每个组件的依赖关系，必须只有一个相应的组件符合应用程序范围中的此依赖关系。

i. 对于每个ejb-link元素，在整个应用程序范围中必须只有一个匹配的ejb-name元素(参见5.5,“企业级JavaBeans™ (EJB)的引用”)

ii. 没有链接到内部组件的依赖关系必须被部署者作为外部依赖关系处理，并且此依赖关系必须被事先安装在平台上的资源所满足。必须在部署期间将外部依赖关系链接到平台上的资源。

c. 跨应用程序同步安全角色名称。将角色名称中含义重复的名称统一为通用名称。将角色名称中表达多种含义的通用名称分离成多个唯一名称。多数应用程序组件使用的角色名称的描述可以写入应用程序级部署描述符中。

d. 为Java EE应用程序中的每个Web模块分配一个上下文root。上下文root是应用程序Web命名空间中的相对路径名称。每个Web模块必须为它的上下文 root给出一个确切并无重叠的名称。这类Web模块会在部署时得到一个Web服务器命名空间的完整名称。如果Java EE应用程序中只有一个Web模块，这个上下文root可以是空字符串。如果应用程序包中没有包含部署描述符，Web模块的上下文root就是模块名称。 有关上下文root命名标准的详细信息，参见Servlet规范。

e. 确保应用程序中的每个组件正确地描述了它对应用程序中其它组件的依赖性。Java EE应用程序不应该假定应用程序中的所有组件在运行时通过应用程序的类路径可用。每个组件可以都可以被加载到具有独立命名空间的类加载器中。如果一个 JAR文件中的类依赖于另一个JAR文件中的类，那么第一个JAR文件应该使用Class-Path机制引用第二个JAR文件。本规则一个特别的例外情况是存放在Web应用程序的WEB-INF/lib目录下的JAR文件。所有这类JAR文件在运行时被引入Web应用程序类路径中；不需要使用Class- Path机制显式地引用它们。本规则另一个例外情况是存放在应用程序包的库目录(一般是lib)下的JAR文件。留意那些共享库中出现的用于声明组件的注 解，例如，库目录下的库和多个模块通过Class-Path引用的库，它们可能会产生意想不到的不良后果，不建议使用。

f. 一个应用程序中的每个类必须只能有一个版本。如果一个组件依赖一个库的某个版本，并且另一个组件依赖另一个版本，那么可能无法部署同时包含这两个组件的应 用程序。由于应用程序客户端可能出现这种例外情况，Java EE应用程序不应该认为每个组件在单独的类加载器中加载并拥有单独命名空间。单个应用程序中的所有组件可以由同一个类加载器加载，并共享同一个命名空间。 注意，然而，必须能够部署一个应用程序，它的所有组件在一个(或多个)从其它应用程序中分割出的命名空间。通常，这是标准的部署方法。默认情况下，每个应 用程序客户端会被部署到它们自己的Java虚拟机实例中，因此，每个应用程序客户端的类有它们自己的命名空间，并且这些来自应用程序客户端的类对其它组件 命名空间中的类是不可见的。

4.为应用程序创建一个XML部署描述符(可选)。

这个部署描述符必须叫做application.xml，并且必须驻留在.ear文件中的META-INF目录的顶层。这个部署描述符必须是一个有 效的XML文档，并且符合java EE:application XML 文档的XML Schema(或者，这个部署描述符也可以满足先前Java EE版本的标准) 。

多数遵循后面约定的应用程序不需要为应用程序配置这类部署描述符。部署工具会确定使用了这些简单规则的应用程序组件。

5. 打包应用程序

a. 在恰当的目录中放置Java EE模块和部署描述符。

b. 使用JAR文件格式，将应用程序目录层次打包到一个文件中。这个文件名应该以.ear后缀结尾。

8.4.2 增加和移除模块

创建应用程序之后，可以在部署前增加或移除Java EE模块。增加或移除一个模块时，必须执行下面的步骤:

1.在应用程序包中为新模块确定一个位置。自由地在应用程序包的层次结构中创建一个新的目录来存放任何要增加的模块。

2.确保新模块的名称没有与已有模块冲突，可以为这个模块选择一个恰当的默认文件名，可以在模块的部署描述符中显式地指定模块名称，也可以使用候补部署描述符。

3.将新的Java EE模块复制到应用程序包中预期的位置。打包后的模块可以直接插入预期的位置；这个模块不能是散乱的。

4.编辑Java EE模块的部署描述符，链接到Java EE模块满足的内部依赖关系。

5.编辑Java EE应用程序的部署描述符(如果存在)以满足Java EE平台的内容标准和Java EE:application XML DTD或Schema的有效性标准。

8.5 部署

Java EE平台支持三种类型的部署单元:

独立的Java EE模块。

一个或多个Java EE模块组成的应用程序。

依据扩展机制体系打包成.jar文件的类库。这些类库随后会变成安装库。

任何Java EE产品必须能够接纳：以.ear格式交付的Java EE应用程序，以.jar,.war或.rar格式交付的独立的Java EE模块(其中的类型必须是恰当的)。如果以.ear格式交付应用程序，以.jar格式交付企业Bean模块，以.war格式交付web应用程序，或者 以.jar格式交付应用程序客户端，部署工具必须能够正确部署这样的应用程序，它的Java类在一个单独的命名空间中，与其它Java应用程序的类分隔。 通常，这要求为每个应用程序使用单独的类加载器。以.rar格式交付的独立的资源适配器和以.jar格式交付的独立的类库，必然会出现在使用它们的应用程 序类的命名空间中，成为安装库。并且它们也可以出现在任何应用程序类的命名空间中，这些应用程序依赖于Java EE产品支持的隔离级别。

任何情况下，对Java EE应用程序的部署，都必须在容器向任何应用程序组件递交请求之前完成。当一个应用程序启动后，容器必须立即向企业Bean组件递交请求。容器只能在组件初始化完成之后向Web组件和资源适配器递交请求。

Java EE部署API描述了独立于产品的部署工具怎样引入特定Java EE产品的插件，以及怎样使用这类工具和这些插件来部署Java EE应用程序。本规范中涉及的部署工具的标准涵盖了任何产品供应商生产的独立于产品的部署工具和为这个工具设计的插件，以及随Java EE产品一起提供的供应商特定的其它任何部署工具。

通常，部署工具会把要部署的应用程序或模块复制到产品的特定位置，包括那些配置设定和部署者的定制。在一些情况下，部署工具也可以附带应用程序组装 功能，允许部署者在部署前构建，修改或定制应用程序。不过，仍然必须能够部署一个可移植的Java EE应用程序，模块或没有特定产品部署信息的库，而不需要修改部署者对部署工具指定的原始文件或历史信息。

Java EE容器的部署工具必须能够依靠相应的Java EE部署描述符的Schema或DTD来检验部署描述符。通过解析部署描述符来确定它声明的Schema或DTD。错误检验必须能将错误汇报给部署者。部署工具允许部署者纠正这个错误并继续部署。

一些部署描述符是可选的。可以使用默认规则或应用程序类文件中的注解来确立必须的部署信息。一些包含在某个应用程序中的部署描述符可能会以完整或

不 完整的形式存在。一个完整的部署描述符提供了完整的部署信息；因此部署工具不能为获取部署信息去检查类文件。一个不完整的部署描述符只提供了必须部署信息 的子集；部署工具必须去检查应用程序类文件注解中的部署信息。部署描述符中指定的任何部署信息可以覆盖应用程序类文件中指定的相应部署信息。Java EE组件规范，包括本规范，描述了什么时候部署描述符是可选的，以及哪种部署描述符可以以完整或不完整的形式存在。部署描述符中的属性metadata- complete用来指定此描述符是否是完整的。

属性metadata-complete的范围是任何出现它的描述符。由于历史原因，部署描述符webservices.xml自己没有 metadata-complete元素；作为代替，它服从模块部署描述符的metadata-complete属性的值。如果某些规范定义了自己附加的部署描述符，它们应该提供自己的metadata-complete属性，如果认为这样做是有用的，就应该赋予它恰当的语义。

8.5.1 部署独立的Java EE模块

本节指定的标准用于部署独立的Java EE模块。

1. 部署工具首先必须读取Java EE模块部署描述符，如果有的话。请查看组件规范以了解每种组件类型的部署描述符所必须的位置和名称。

2. 如果没有出现部署描述符，或者出现的是Java EE 5 版的描述符并且metadata-complete属性没有设为true，那么部署工具必须检查应用程序包中的所有类文件注解。任何指定了部署信息的注解 必须逻辑上并入部署描述符(如果存在)中的信息。关于注解信息与部署描述符的合并以及覆盖规则，描述在本规范和其它Java EE规范中。这种逻辑合并处理的结果提供的部署信息被用在随后的部署过程中。注意，对这种合并处理，没有标准要求生成一个新的部署描述符，然而它却是一种 常见的实现技术。

3. 部署一个独立的模块时，模块名称被用作应用程序的名称。部署工具必须确保在应用程序服务器实例中这个名称是唯一的。如果这个名称不是唯一的，部署工具可以自动确定一个唯一的名称，或让部署者来确定，但是不能中止部署。这保证了已有模块是仍然是可部署的。

4. 部署工具必须部署Java EE模块部署描述符中列出的所有组件，以及那些通过注解标记发现的组件，正如之前的标准所描述的，这些都需要依据各自的Java EE组件规范中的部署标准。如果一个模块包含了JAR格式文件(例如，Web和连接器模块)，那么它所引用的其它模块的.jar文件中的所有类必须得到部 署，这是通过Class-Path头元素实现的。如果这个模块，或它的任何JAR格式文件，声明了对某个安装库的依赖性，那么这个依赖性必须被满足。

5. 部署工具必须允许部署者对容器进行配置，以提供每个组件所需的资源和配置参数。这些必须的资源和配置参数指定在部署描述符中，或通过注解发现，参见上面第2条。

6. 部署工具必须允许部署者多次部署相同的模块，作为多个独立的应用程序，但它们可能具有不同的配置。例如，在ejb-jar文件中的企业Bean可能在不同的JNDI名称下被部署了多次，并且具有自己不同资源配置。

8.5.2 部署Java EE应用程序

本节指定的标准用于部署Java EE应用程序。

1. 部署工具必须首先从Java EE应用程序的.ear文件中读取它的部署描述符(META-INF/application.xml)。如果这个部署描述符存在，它会完整地列出应用程 序中的模块。 如果这个部署描述符不存在，部署工具应该使用下面的规则来确定应用程序中的模块。

a. 应用程序包中所有带有.war后缀名的文件被认为是Web模块。Web模块的上下文root就是模块的名称(参见8.1.1, “创建组件”)。

b. 应用程序包中所有带有.rar后缀名的文件被认为是资源适配器。

c. 名称为lib的目录被认为是库目录，正如EE8.2.1,“绑定的库”中所描述的。

d. 对于应用程序包中带有.jar后缀名，但却不在lib目录下的所有文件，按下列方式处理:

i. 如果这个JAR文件中包含了一个META-INF/MANIFEST.MF文件，其中带有一个Main-Class属性，或者它包含了一个META-INF/application-client.xml文件，那么它就被认为是一个应用程序客户端模块。

ii. 如果这个JAR文件包含了一个META-INF/ejb-jar.xml文件，或者它包含了任何带有EJB注解(Stateless等等)的类，那么它就被认为是一个EJB模块。

iii. 所有其它的JAR文件将会被忽略，除非它被某个JAR文件引用，并通过一种JAR文件引用机制找到，比如，Manifest文件中的Class-Path头元素。

1.部署工具必须确保这个应用程序名称在应用程序服务器实例中是唯一的。如果这个名称不是唯一的，部署工具可以自动确定一个唯一的名称，或者让部署者来确定，但是不能中止部署。这保证了已有的应用程序仍然是可部署的。

3.部署工具必须打开Java EE部署描述符中列出的每个Java EE模块，以及那些使用上面的规则发现并读取模块描述符(如果存在)获取的模块。参见企业级JavaBean，Servlet，Java EE连接器和应用程序客户端规范，以了解每个组件类型的部署描述符必须的位置和名称。部署描述符对所有模块类型是可选的(应用程序客户端规范参见10,“应用程序客户端”)

4.如果模块部署描述符不存在，或出现的是Java EE 5的版本，又或者metadata-complete属性没有设为true，那么部署工具必须检查模块使用的应用程序包中的所有类文件(即，所有包含 在.ear文件中并可以被模块引用的类文件，比如模块自己包含的类文件，使用Class-Path从模块中引用的类文件，库目录下的类文件等等)。任何指 定了部署信息的注解必须逻辑上并入部署描述符(如果存在)中的信息。注意，如果共享的历史信息里面出现了声明组件的注解，例如库目录中的库和多个模块通过 Class-Path引用的库，它们可能会产生意想不到的不良后果，不建议使用。关于注解信息与部署描述符的合并以及覆盖规则，描述在本规范和其它 Java EE规范中。这种逻辑合并处理的结果提供的部署信息被用在随后的部署过程中。注意，对于这种合并处理，没有标准要求生成一个新的部署描述符，然而它却是一 种常见的实现技术。

5.部署工具必须将每个模块部署描述符中列出的所有组件，以及那些通过注解标记发现的组件安装到恰当的容器中，这些都需要依据各自的Java EE组件规范中的部署标准。通过Class-Path头元素从其它JAR文件中的.jar文件或目录中引用的所有类必须得到部署。如果.ear文件，或者 它包含的任何JAR格式的文件，声明了对某个安装库的依赖性，那么这个依赖性必须被满足。

6.部署工具必须允许部署者对容器进行配置，以提供每个组件所需的资源和配置参数。这些必须的资源和配置参数指定在部署描述符中，或通过注解发现，参见上面第3条。

7.部署工具必须允许部署者多次部署相同的Java EE应用程序，作为多个独立的应用程序，但它们可能具有不同的参数。例如，在ejb-jar文件中的企业Bean可能在不同的JNDI名称下被部署了多次，并且具有自己不同的资源配置。

8.当部署者遇到名称冲突的安全角色描述时，部署工具必须使用Java EE应用程序部署描述符中的描述，而不是任何模块部署描述符中的描述。然而，对于出现在某个模块部署描述符中，而没有出现在应用程序部署描述符中的安全角色，部署工具必须使用这个模块部署描述符提供的描述。

8.5.3 库的部署

本节指定的标准用于库的部署。

1.部署工具必须从库的JAR文件中找到Manifest文件，并记录其中的扩展名和版本信息。部署工具必须使这个库对其它的Java EE部署单元可用，这些部署单元必须依据可选包版本规范描述的版本匹配规则来请求这个库。注意，这个库自身也可以依赖其它的库，并且这些依赖性也必须被满 足。

2.部署工具必须使这个库对应用程序或模块可用，并且这个库至少拥有与它们相同的安全权限。这个库也可以拥有容器全部的安全权限。

3.并不是任何时候都能将所有的库部署到任何Java EE产品上。不能部署与Java EE产品运作相冲突的库。例如，某个库随后被列入了Java EE平台规范，倘若试图部署它的旧版本可能会遭到拒绝。类似地，部署一个已经被Java EE产品使用的库可能会被拒绝。部署一个应用程序正在使用的库也可能会被拒绝。

8.5.4 模块的初始化

部署成功之后，除了应用程序客户端模块，应用程序中的其它所有模块都会进行初始化。不同类型的模块规范描述了初始化模块所必须的步骤。默认情况下， 应用程序没有指定模块的初始化顺序。某些特殊情况下，模块的初始化顺序是至关重要的。例如，一个模块的组件在初始化阶段使用了其它模块中的组件。应用程序 可以声明，它的模块必须按照部署描述符中列出的顺序进行初始化，这需要在应用程序部署描述符中包含<initialize-in-order>true</initialize-in-order>元素。如果应用程序部署描述符指定的模块初始化顺序与某个模块指定 的顺序(例如，使用了EJB DependsOn注解)发生冲突，部署工具必须发出一个错误信息。应用程序客户端模块按它们自己的计划进行初始化，通常是在终端用户调用它们时；因此它 们不受任何初始化顺序标准约束。

8.6 Java EE应用程序的XML Schema

Java EE应用程序部署描述符的XML语法由Java EE应用程序Schema定义。这个部署描述符的根元素是application。Java EE应用程序结构的组装粒度为Java EE模块。Java EE应用程序部署描述符包含了这个应用程序的名称，描述，和UI图标的URI，以及构成这个应用程序的Java EE模块列表。这些XML元素的内容通常是大小写敏感的。这表示，<role-name>Manager</role- name>所表示的角色与<role-name>manager</role-name>是不同的。

所有合法的Java EE应用程序部署描述符必须符合这个XML Schema定义，或者符合本规范早期版本的DTD或Schema定义(参见附录A,“早期版本的部署描述符”)，这个部署描述符必须是.ear文件中的 META-INF/application.xml文件。注意，这个名称是大小写敏感的。

这个XML Schema定义了Java EE应用程序部署描述符的语法，参见http://java.sun.com/xml/ns/javaee/application_6.xsd

8.7 通用Java EE XML Schema定义

这个XML Schema定义了其它很多Java EE部署描述符Schema使用的类型，参见 http://java.sun.com/xml/ns/javaee/javaee_6.xsd ，这些类型描述在本规范和其它规范中。

第9章 Profile

本章描述了所有Java EE Profile的公共标准。它没有定义任何具体的Profile,而是把这一任务交给各自的规范。

Java EE Web Profile规范，协同当前规范，定义了第一项Java EE Profile--Web Profile。

其它Profile定义将留给未来的规范。

9.1 简介

Java EE Profile（以下简称”Profile”)描绘了适合应用程序特定类的平台结构。

一项Profile可以包含平台技术的恰当子集。通过这种方式，它能够有效地摒除那些被平台支持却不太适合特定领域的技术。

一项Profile也可以增加一项或多项此平台没有出现技术。例如，假设某项Java EE Portal Profile，它可能会包含Portlet API (JSR-286)。

再如，一项Profile可以将某项技术标记为可选。这样，实现这项Profile的产品就可以选择性地包含此项尚有争议的技术。如果这样做了，它们理应遵循Profile规范委任的所有相关标准。

一个产品可以实现两项或多项Java EE Profile，甚至是实现整个平台再附加一项或多项Java EE Profile。只要它们合成的标准不会引起冲突。

9.2 定义Profile

定义Profile必须依据Java Community Process制定的规则。通常，一项创建新Profile或修改现有Profile的提议，将作为Java Specification Request(JSR)呈交。一旦这项JSR获得批准，就会选出一个专家组来引导此项工作。Profile JSR必须说明自己所处的Java EE平台版本。再者，如果它基于某个现有的Profile，那么它也必须说明这个情况。

尽管Profile可以创建并演化出一个新的平台，而独立于Java EE平台，但是我们期望Profile能够以本规范为模型，维系在一个合理的水平上，与Java EE平台衔接，避免破坏自己的生长空间，最后成为一座孤岛。为此，Profile必须构建于Java EE平台可用的最新版本上，这时这项Profile JSR才会获得批准。我们也建议Profile专家组的工作并不局限于本标准，请充分确信它的可行性，并确保自己的Profile构建于最新Java EE平台上，这时这项Profile才可以定稿。

9.3 Profile的总体原则

Profile必须包含它所基于的Java EE平台或任何Profile中的所有必须组件。必须在Profile中列出这些技术。

Profile可以将任何技术提升为必须级别，这些技术是它所基于的Java EE平台或任何Profile中的可选组件。

除非Profile用别的方式进行了委任，否则在这项Profile中存在争议的任何技术必须作为它的可选组件，这些技术是它所基于的Java EE平台或任何Profile中的可选组件。

Profile可以包含任何技术作为必须组件或可选组件出现，这些技术可以超出这项Profile所基于的Java EE平台或任何Profile的范围，只要符合相应的兼容性标准。

Profile必须维护它所基于的Java EE平台规范或任何Profile规范中定义的标准，只要这些标准的先决条件得以满足。通常，这些先决条件会涉及到这项Profile中的一项或多项技术。无条件标准必须无条件遵循。

Profile可以为一项或多项技术增加适合它们的任何标准，如果它们允许或要求的话。这些标准不能与Profile所基于的那些Java EE平台或任何Profile设定的标准冲突。

个别技术规范可以允许它的技术将某些存在争议的特性作为可选级别。在这种情况下，Profile可以将一个或多个这些特性提升为必须级别，假设它所基于的Java EE平台或任何Profile还没有这样做。

Profile不能与任何技术规范(它包含了必须或可选的组件)冲突。因此，除非那项技术规范明确指出某些特性或标准是可以随意实现的，否则 Profile不能试图自己重新定义这种特性或标准。例如，Profile不可以删去别的API指定的包，类型或方法，除非那个API规范明确允许这种行 为。

本规范没有定义任何API，而Profile可以定义自己的API。但是这些API仅在定义它的Profile中可用，这就限制了这些API的可重用性，我们不鼓励这种做法。

9.4 标准的扩展

当前规范按照下面的惯例来表述平台相应技术的标准:

> 以具体技术作为开始的章节指出了条件，并打算将这个条件保持下去，直到下一个正文单元在相同的逻辑水平上进行了描述(例如下面的章节等等)。
>
> 个别段落或句子被视为它们提到的任何技术的条件，除非以别的方式指出了。
>
> 带有示例的小节或段落，或者非规范的论述中，没有包含任何标准。

9.5 所有Java EE Profile的标准

Java平台标准版6是所有Java EE 6 Profile必须的基础。

下面的技术必须出现在所有Java EE Profiel中:

> JSR-250定义的资源和组件生命周期注
> 解(@Resource，@Resources，@PostConstruct，@PreDestroy)

所有Java EE Profile必须支持下面的功能:

> JNDI的“java:”命名上下文(参见5.2,"JNDI命名上下文")

Java事务API(JTA)

9.6 Java EE Profile的可选特性

在6.1,“必须的API”中列出，并且不在9.5,“所有Java EE Profile的标准”范围中出现的所有技术，被定义为Java EE Profile的可选技术。下面的功能也是Java EE Profile的可选技术:

RMI/IIOP 交互性标准(参见7.2.2,“OMG协议”)

对java:comp/ORB的支持(参见5.12,“引用ORB”)

9.7 完整的Java EE产品标准

本节定义的标准用于完整的Java EE平台产品。这里的全部标准对应于先前Java EE平台规范，并在这个新版本中升级了这些标准。

请注意，受修剪流程的影响，Java EE平台规范的未来版本可能会弱化这里给出的标准，因为标记为可选的技术都已经被移除了，剩下的都是当前规范的必须技术。关于那些已经申请修剪的技术，请参见6.1.3,“被修剪的Java技术”。

下面的技术是必须的:

EJB 3.1

Servlet 3.0

JSP 2.2

EL 2.2

JMS 1.1

JTA 1.1

JavaMail 1.4

Connector 1.6

Web Services 1.3

JAX-RPC 1.1

JAX-WS 2.2

JAX-RS 1.1

JAXB 2.2

JAXR 1.0

Java EE Management 1.1

Java EE Deployment 1.2

JACC 1.4

JASPIC 1.0

JSP Debugging 1.0

JSTL 1.2

Web Services Metadata 2.1

JSF 2.0

Common Annotations 1.1

Java Persistence 2.0

Bean Validation 1.0

Managed Beans 1.0

Contexts and Dependency Injection for Java EE 1.0

Dependency Injection for Java 1.0

以上没有可选技术。

第10章 应用程序客户端

本章描述了Java™平台企业版(Java EE)中的应用程序客户端。

完整的Java EE产品必须支持应用程序客户端容器，正如本章所描述的。Java EE Profile可以选择支持或不支持应用程序客户端容器。

10.1 概述

应用程序客户端是执行在自己的Java™虚拟机中的第一层客户端程序。应用程序客户端遵循以Java技术为基础的应用程序 模型: 它们由main方法调用并运行，直到虚拟机终止。然而，同其它Java EE应用程序组件一样，应用程序客户端依靠容器提供系统服务。与其它Java EE容器相比，应用程序客户端容器可以是非常轻量级的，仅提供下面描述的安全和部署服务。

10.2 安全

Java EE应用程序客户端验证标准与其它Java EE组件相同，并且可以使用与它们相同的验证技术。

访问非敏感资源无需验证。而访问受保护的Web资源时，可以使用一些常见的验证方式，即HTTP基本验证，SSL客户端验证或HTTP表单登录验证。可以使用即时验证。

访问受保护的企业Bean资源时必须进行验证。企业Bean验证机制包含了EJB规范互用性标准中的那些要求。可以使用即时验证。

应用程序客户端使用应用程序客户端容器提供的验证服务来鉴定用户。容器的这种服务可以与本地平台的验证系统整合，以便具备单点登录能力。应用程序启 动之后，容器就可以验证用户。它也可以使用即时验证技术，当用户访问受保护的资源时才进行验证。本规范没有描述用户验证技术，然而以后的版本可能会这样 做。

如果容器想采集用户的验证信息，它必须提供合适的用户界面。另外，应用程序客户端可以提供一个实现了
javax.security.auth.callback.CallbackHandler接口的类，并在它的部署描述符中指定这个类的名称(详细信息 参见10.7,“Java EE应用程序客户端的XML Schema”)。部署者可以重写应用程序指定的回调处理器，并使用容器默认的用户验证界面代替。

如果回调处理器由部署者配置，应用程序客户端容器必须实例化这个类的对象，并将它应用于所有用户验证活动。应用程序的回调处理器必须完全支持javax.security.auth.callback包指定的Callback对象。

注意，如果使用了HTTP表单登录验证，服务器提供的用户验证界面(响应HTTP请求时输出的HTML页面中的表单)必须由应用程序客户端显示。

应用程序客户端通常运行在带有安全管理器的环境中，并且具有与Servlet类似的安全权限。安全权限标准全部描述在6.2,“Java平台标准版(Java SE)标准”中。

10.3 事务

不要求应用程序客户端直接访问Java EE平台的事务功能。不要求Java EE产品为应用程序客户端提供一个可用的JTA UserTransaction对象。应用程序客户端可以调用企业Bean来启动事务，间接地使用JDBC API的事务功能。如果应用程序客户端调用企业Bean时开启了一个事务，不要求将这个事务上下文传递给EJB服务器。

10.4 资源，命名和注入

与所有Java EE组件一样，应用程序客户端可以使用JNDI来查找企业Bean，访问资源管理器，引用部署时设定的配置参数，等等。应用程序客户端使用“java:”JNDI命名空间来访问它们(详细信息参见5,“资源，命名和注入”)。

应用程序客户端的主类也支持注入。因为应用程序客户端容器没有创建应用程序客户端主类的实例，而只是加载了这个类并调用了静态main方法，使用静态字段和方法注入应用程序客户端，不像其它Java EE组件。注入发生在main方法调用之前。

10.5 应用程序编程接口

应用程序客户端拥有Java平台标准版的所有功能(服从安全约束)，以及各种标准扩展，正如6,“应用程序编程接口”中描述的。每个应用程序客户端 运行在自己的Java虚拟机中。应用程序客户端从main方法开始执行，这个主类由应用程序客户端JAR文件中的Manifest文件中的Main- Class属性指定(需要注意的是，应用程序客户端容器代码通常会先于应用程序客户端执行，为了预备容器的环境，需要安装安全管理器，初始化命名服务客户 端库，等等)。

10.6 打包和部署

应用程序客户端被打包成JAR格式文件，扩展名为.jar，并且它可以包含一个类似于其它Java EE应用程序组件的部署描述符。部署描述符描述了企业Bean，Web服务以及应用程序引用的其它类型的外部资源。如果没有包含部署描述符，或没有标记 metadata-complete元素，应用程序主客户端主类上的注解也可以用来描述应用程序所需的资源。同其它Java EE应用程序组件一样，必须在部署时对资源的访问进行配置，并分配企业Bean和资源的名称，等等。

下面的表描述了，当决定是否处理应用程序客户端主类上的注解时，部署工具必须考虑的情况。

表 10-1 部署描述符处理标准

部署描述符	是否具有metadata-complete	是否处理注解
application-client_1_2	N/A	No
application-client_1_3	N/A	No
application-client_1_4	N/A	No
application-client_5	Yes	No
application-client_5	No	Yes
none	N/A	Yes

没有指定用于将应用程序客户端部署到客户机上的工具，也没有指定用于安装客户端的机制。 非常高端的Java EE产品可以允许将应用程序客户端部署到Java EE服务器上，并且自动对某些客户端(通常是内网)可用。其它Java EE产品可以要求Java EE应用程序包(包含了应用程序客户端)被手动地部署和安装到每个客户机上。然而，部署工具可以在Java EE服务上选择别的途径来生成安装包，这个安装包可以被每个客户机用来安装应用程序客户端。这里有多种可能性，但在本规范没有明确指出。它只定义了应用程 序包的格式以及部署处理期间必要的事项。

没有规定应用程序客户端怎样被终端用户调用。通常Java EE产品供应商会提供一个集成在应用程序客户机本地操作系统上的应用程序启动器，但是没有指定这种整合的程度。

10.7 Java EE应用程序客户端的XML Schema

Java EE应用程序客户端部署描述符的XML语法由Java EE application-client Schema定义。这个应用程序客户端部署描述符的根元素是application-client。这些XML元素的内容一般是大小写敏感的。这表示，必 须使用<res-auth>Container</res-auth>，而不是<res- auth>container</res-auth>。

所有合法的application-client部署描述符必须符合XML Schema定义，或来自本规范早期版本的DTD或Schema定义(参见附录A,“早期版本的部署描述符”)。这个部署描述符必须是应用程序客户 端.jar文件中的META-INF/application-client.xml文件。注意，这个名称是大小写敏感的。

图 10-1 展示了Java EE application-client XML Schema的结构。Java EE application-client XML Schema参见http://java.sun.com/xml/ns/javaee/application-client_6.xsd

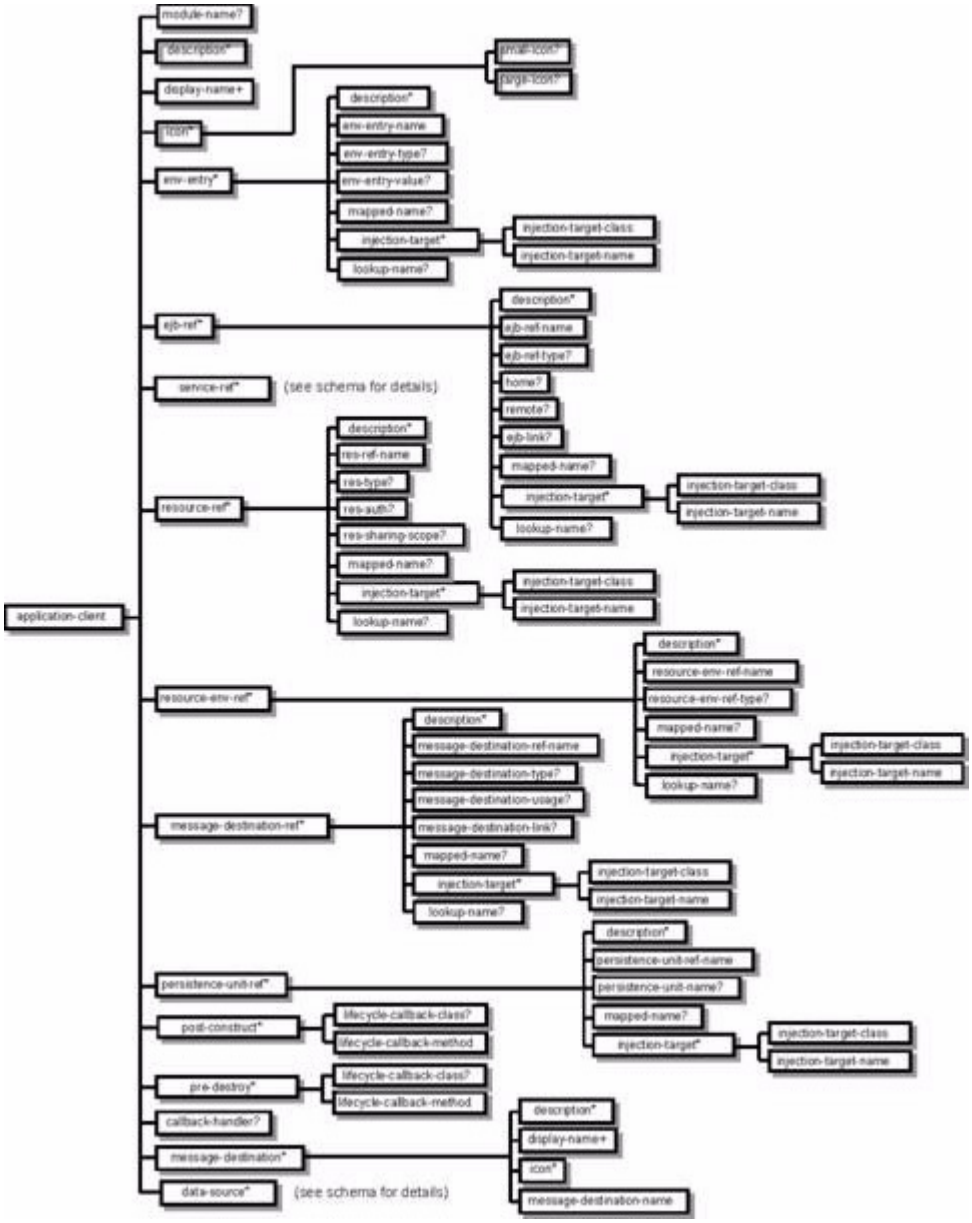

图 10-1 Java EE应用程序客户端XML Schema的结构

第11章 服务供应商接口

Java™平台企业版(Java EE)包含了几项技术，主要用于扩展Java EE容器的功能。另外，一些Java EE技术随着它们的应用程序编程接口附带了一些服务供应商接口。Java EE Profile可以包含这些功能的部分或全部，正如9, “Profile”中描述的。

11.1 Java™ EE连接器体系结构

连接器API定义了怎样将资源适配器打包并集成到任何Java EE产品中。使用连接器API可以提供很多类型的服务提供方，包括JDBC驱动，JMS提供方和JAXR提供方。所有Java EE产品必须支持连接器API，正如连接器规范所指定的。

连接器规范参见 http://java.sun.com/j2ee/connector

11.2 容器的Java™授权服务提供方协议

JACC规范定义了J2EE容器和授权策略提供方之间的协议。

JACC规范参见 http://jcp.org/jsr/detail/115.jsp

11.3 Java™事务API

Java事务API为系统级应用程序服务器组件定义了TransactionSynchronizationRegistry类，比如持久化管理 器，资源适配器以及EJB和Web应用程序组件。它提供以下功能：注册同步对象，并带有特殊的排序语义，将资源对象关联到当前事务，获取当前事务的上下文，获取当前事务的状态，以及标记当前事务的回滚点。

JTA规范参见 http://java.sun.com/products/jta

11.4 Java™持久化

Java持久化提供的接口在javax.persistence.spi包中，它允许将持久化提供方插入到Java持久化框架中。Java持久化规范由EJB专家组指定，参见http://jcp.org/en/jsr/detail?id=220

11.5 为XML Web服务提供的Java™ API

JAX-WS提供的接口在javax.xml.ws.spi包中，它支持JAX-WS实现的即插即用性。JAX-WS规范参见http://java.sun.com/webservices/jaxws

11.6 JavaMail™

JavaMail规范描述了JavaMail协议提供方怎样被打包和分布，以便通过JavaMail API可以发现并使用它们。这允许通过支持新的邮件协议和邮箱格式来扩展JavaMail API。

JavaMail规范参见 http://java.sun.com/products/javamail

第12章 兼容性和迁移

本章总结了Java EE平台的兼容性和迁移问题。Java EE中包含的每项技术规范更详细地描述了自己技术的兼容性和迁移问题。

12.1 兼容性

术语“兼容性”涉及到很多不同的概念。如果Java EE产品实现了Java EE规范所要求的API和行为，那么它们就与此规范兼容。如果应用程序仅依赖于某个Java EE平台发布版的API和行为，那么它们就与那个发布版兼容。如果所有为先前平台编写的可移植的应用程序也能够正确地运行在新的平台上，那么这个新的平台 就与那个旧的平台兼容。

兼容性是Java EE平台的核心价值。要求Java产品支持为先前的版本编写可移植的应用程序。兼容性和可移植性的协同工作使Java EE平台可以“一次编写，到处运行”。只有提供Java EE规范所要求的API和行为，Java EE产品才是符合规范的。可移植的应用程序只依赖于Java EE规范所要求的API和行为。一般情况下，为先前版本编写的可移植的应用程序都可以正确地运行在当前平台版本上，而不会出现异常的状态和行为。

12.1.1 JSP

JSF的出现和表达式语言的推广要求JSP页面轻微地调整自己的句法，这会带来小小的兼容性问题。字符序列 #{ 现在被保留并用于指定延迟评估。在模板文本中使用了这个字符序列的JSP页面应对这个序列进行调整，比如\#{

JSP页面现在支持Unicode字节顺序标记。目前很少使用ISO-8859-1字符作为页面的首选字符集，这会需要调整页面以正确地显示。

请查看JSP规范进一步了解这些不兼容性的细节。

12.2 迁移

迁移实际上就是使用本平台引入的新技术来更新应用程序。当前发布的Java EE平台拥有很强的兼容性级别，迁移其实只是一种可选的行为。然而，将应用程序迁移到这些新技术上，可以使它们得到改良(更好的性能，更简洁的开发，更高的灵活性等等)。

12.2.1 JSF

JSF能够简化应用程序Web用户界面的开发。使用JSP提供Web用户界面的应用程序可以迁移到JSF技术上，将它作为构建这类用户界面的一个高级组件框架。

先前版本的JSF和JSP规范为使用和控制它们支持的表达式语言而定义了API。在本平台中，那些API已经过时了，它们被新的javax.el包中API代替。强烈建议将应用程序迁移到这些新的API上。注意，这些过时的API还可以继续像以前一样工作。

12.2.2 Java持久化

Java持久化技术提供了更丰富的建模功能和对象-关系映射功能，不但超越了EJB CMP本地企业Bean，而且更易于使用。强烈建议管理数据库持久化信息的应用程序优先考虑使用Java持久化技术，然后才是EJB CMP技术或JDBC数据访问对象(DAO)。

在本平台中，EJB CMP 1.1实体Bean已经过时了。强烈建议将使用EJB CMP 1.1实体Bean的应用程序迁移到Java持久化技术上。注意，EJB CMP1.1实体Bean在本平台中可以继续工作。

12.2.3 JAX-WS

JAX-WS联同JAXB和Web服务的元数据规范，为Web服务提供了更简洁和更完整地支持，其可用性超越了JAX-RPC。使用JAX-RPC 提供Web服务的应用程序应该考虑迁移到JAX-WS API上。注意，由于这两种技术都支持相同的Web服务互用性标准，因此客户端和服务可以独立地迁移到这种新的API上。

12.2.4 注解

一种可以大大简化Java EE应用程序部署的关键技术就是Java语言注解。注解显着地简化了Java持久化技术和JAX-WS技术的使用。通过使用注解，很多应用程序可以避免使用部署描述符，从而大大简化应用程序的部署。开发者应该考虑使用注解来代替部署描述符。

第13章 未来的方向

本Java™ EE平台企业版规范包含了企业级应用程序需要的大多数工具包。然而，仍然有更多的工作需要我们去做。本章简短地描述了我们对本规范未来版

本的计划。请记住，所有这些都可能发生变化。您的反馈是对我们最大的鼓励。

下面的小节描述了未来版本中我们希望增加的功能。Java EE平台包含的很多API会继续自行发展，而我们也会引入每个API的最新版本。

13.1 JNLP (Java™ Web Start)

Java网络启动协议定义了一种机制，它可以将Java应用程序部署到服务器上，然后从客户端启动它们。本规范的未来版本可能会要求Java EE产品能够以一种特殊的方式部署应用程序客户端，这种方式允许从JNLP客户端启动它们，并且应用程序客户端容器能够启动这种使用JNLP技术部署的 JNLP客户端。Java™ Web Start是JNLP客户端的参考实现。

关于JNLP的更多信息参见 http://jcp.org/en/jsr/detail?id=056 ；关于Java Web Start的更多信息参见 http://java.sun.com/products/javawebstart

13.2 Java EE SPI

构成Java EE平台的很多API包含了一个SPI层，它允许在平台上插入服务提供方或其它系统级组件。本规范没有描述所有这类服务提供方的执行环境，也没有给出它们 的打包和部署标准。然而，Java EE连接器体系结构定义了一种类型的服务提供方标准，它就是资源适配器。另外，容器的Java授权协议为安全服务提供方也定义了标准。本规范的未来版本会 更完整地定义Java EE SPI。

附录A 早期版本的部署描述符

本附录描述了早期J2EE规范的部署描述符的文档类型定义(DTD)和XML Schema。要求所有Java EE产品支持这些DTD和Schema，以及当前版本指定的Schema。这就确保了为早期版本编写的应用程序能够部署到当前平台的产品中。另外，没有限 制在单个应用程序中混合使用不同版本的部署描述符；任何合法的部署描述符都必须得到支持。

A.1 Java EE 5应用程序的XML Schema

Java EE 5应用程序部署描述符的语法由Java EE应用程序的Schema定义。它的根元素是application。Java EE应用程序的组装粒度是Java EE模块。Java EE应用程序部署描述符包含了应用程序的名称和描述，它的UI图标的URI，以及组成这个应用程序的Java EE模块的列表。这些XML元素的内容一般是大小写敏感的。这表示，<role-name>Manager</role- name>与<role-name>manager</role-name>是两个不同的角色。

合法的Java EE 5应用程序部署描述符必须符合这个XML Schema定义。

在.ear文件中，这个部署描述符必须是META-INF/application.xml。

注意，这个名称是大小写敏感的。

图 A-1 展示了Java EE应用程序的XML Schema的结构

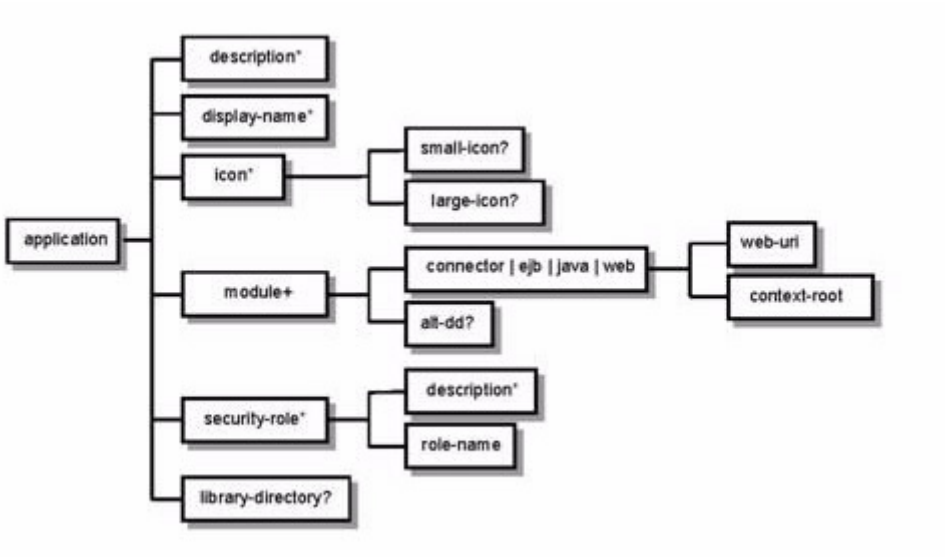

图 A-1 Java EE应用程序XML Schema的结构

这个XML Schema定义了Java EE应用程序部署描述符的XML语法，参见 http://java.sun.com/xml/ns/javaee/application_5.xsd

A.2 通用Java EE 5 XML Schema定义

这个XML Schema定义了由很多其它Java EE部署描述符的Schema使用的类型，参见 http://java.sun.com/xml/ns/javaee/javaee_5.xsd ，它们在本规范和其它规范中。

A.3 Java EE 5应用程序客户端的XML Schema

Java EE应用程序客户端部署描述符的XML语法由Java EE application-client Schema定义。它的根元素是application-client。这些XML元素的内容一般是大小写敏感的。这表示，必须使用<res- auth>Container</res-auth>，而不是<res-auth>container</res- auth>。

所有合法的application-client部署描述符必须符合XML Schema定义，或符合本规范早期版本的DTD或Schema定义(参见附录A,“早期版本的部署描述符”)。在应用程序客户端的.jar文件中，这个 部署描述符必须是META-INF/application-client.xml。注意，这个名称是大小写敏感的。

图 A-2 展示了Java EE application-client XML Schema的结构，参见http://java.sun.com/xml/ns/javaee/application-client_5.xsd

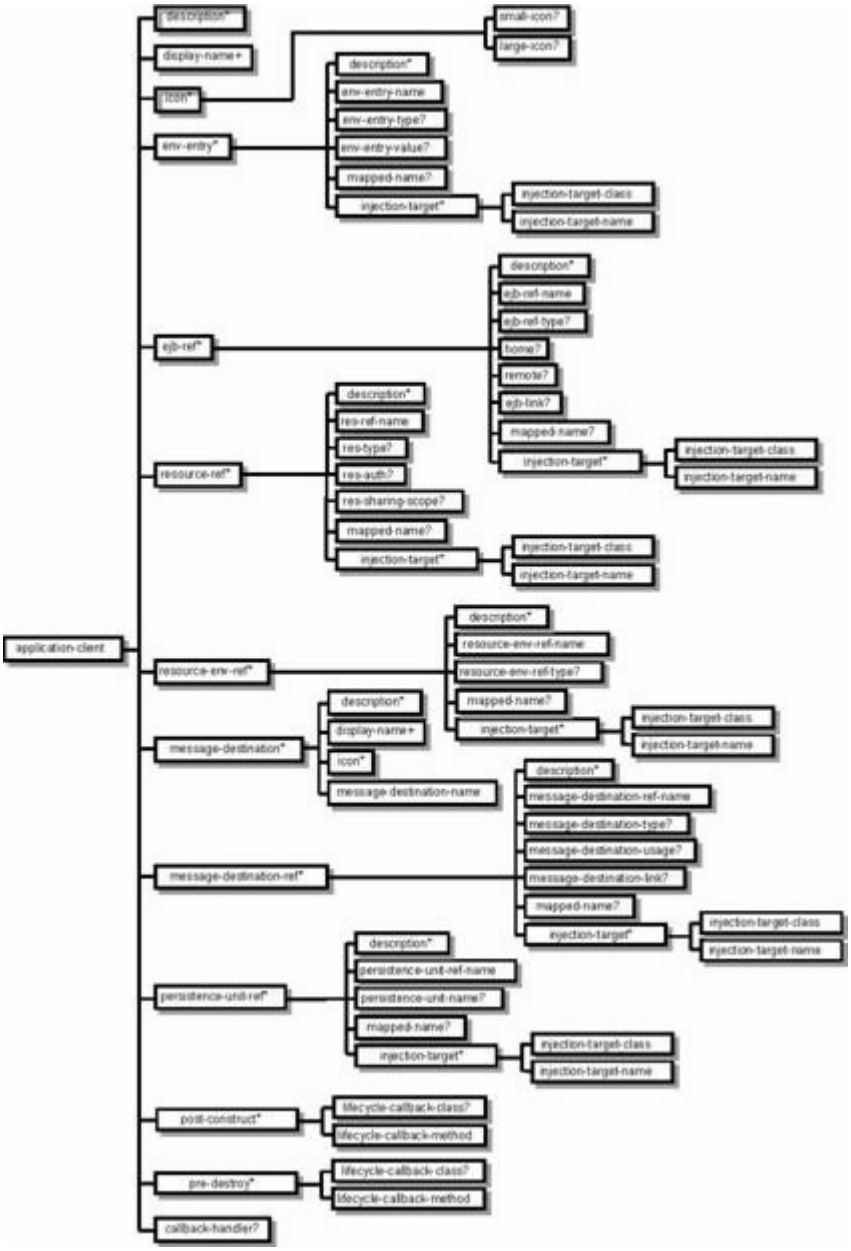

图 A-2 Java EE应用程序客户端XML Schema的结构

A.4 J2EE 1.4应用程序的XML Schema

本节提供了J2EE应用程序部署描述符的XML Schema。它的XML语法由J2EE:application Schema定义。J2EE应用程序的组装粒度是J2EE模块。J2EE:application部署描述符包含了应用程序的名称和描述，它的UI图标的 URI，以及组成这个应用程序的J2EE模块的列表。这些XML元素的内容是大小写敏感的。这表示，<role- name>Manager<role-name>与<role-name>manager</role- name>是两个不同的角色。

一个合法的J2EE应用程序部署描述符可以符合下面的XML Schema定义。部署在.ear文中的，这个部署描述符必须是META-INF/application.xml。注意，这个名称是大小写敏感的。

图 A-3展示了J2EE XML Schema的结构。

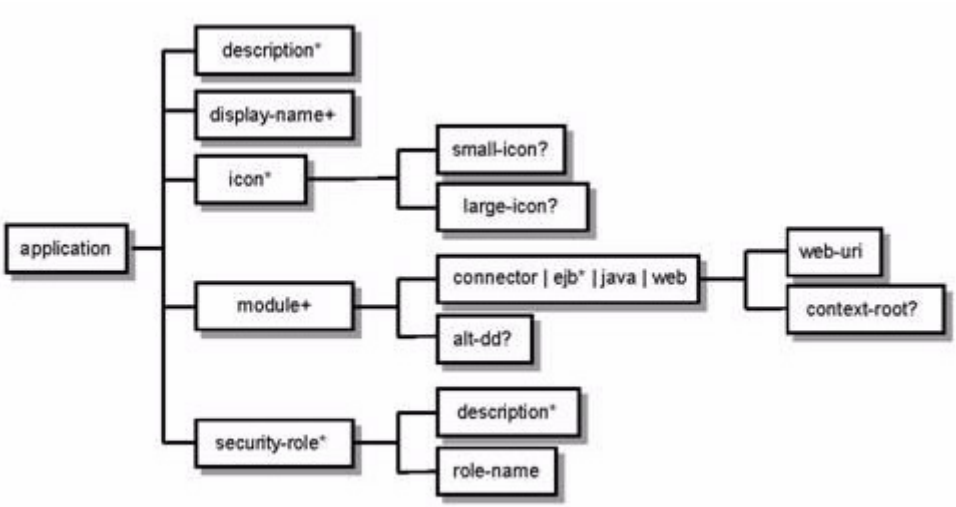

图 A-3 J2EE应用程序XML Schema的结构

这个XML Schema定义了J2EE 1.4应用程序部署描述符的XML语法，参见http://java.sun.com/xml/ns/j2ee/application_1_4.xsd

A.5 通用J2EE 1.4 XML Schema的定义

这个XML Schema定义了很多其它J2EE1.4部署描述符的Schema使用的类型，它们在本规范和其它规范中，参见
http://java.sun.com/xml/ns/j2ee/j2ee_1_4.xsd

A.6 J2EE:application 1.3 XML DTD

本节提供了J2EE 1.3应用程序部署描述符的XML DTD。它的XML语法由J2EE:application的DTD定义。J2EE应用程序的组装粒度是J2EE模块。J2EE:application 部署描述符包含了应用程序的名称和描述，它的UI图标的URI，以及组成这个应用程序的J2EE模块的列表。这些XML元素一般是大小写敏感的。这表 示，<role-name>Manager</role-name>与<role-name>manager</role-name>是两个不同的角色。

一个合法的J2EE1.3应用程序部署描述符可以包含下面的DOCTYPE声明:

<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

在.ear文件中，部署描述符必须是META-INF/application.xml。

图 A-4 展示了J2EE:application XML DTD的结构。

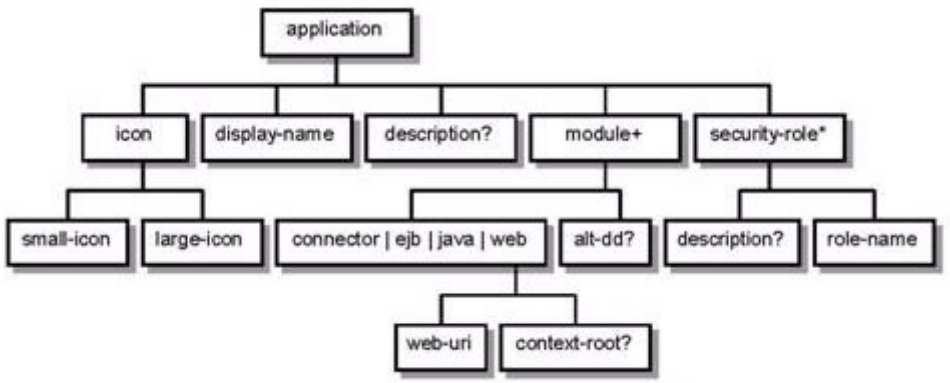

图 A-4 J2EE:application XML DTD的结构

这个DTD定义了J2EE1.3应用程序部署描述符的XML语法，参见http://java.sun.com/dtd/application_1_3.dtd

A.7 J2EE:application 1.2 XML DTD

本节提供J2EE1.2的应用程序部署描述符的XML DTD。一个合法的J2EE1.2应用程序部署描述符可以包含下面的DOCTYPE声明:

<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">

图 A-5展示了J2EE:application XML DTD的结构。

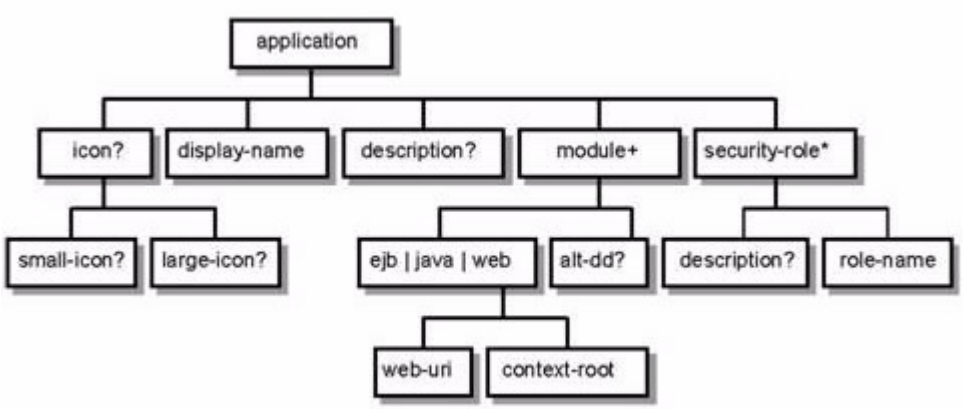

图 A-5 J2EE:application XML DTD的结构

这个DTD定义了J2EE1.2应用程序部署描述符的语法，参见http://java.sun.com/j2ee/dtds/application_1_2.dtd

A.8 J2EE 1.4应用程序客户端的XML Schema

J2EE应用程序客户端部署描述符的XML语法由J2EE application-client Schema定义。它的根元素是application-client。这些XML元素的内容是大小写敏感的。这表示，必须使用<res- auth>Container</res-auth>而不是<res-auth>container</res- auth>。

一个合法的application-client部署描述符可以符合下面的XML Schema定义。在应用程序客户端的.jar文件中，这个部署描述符必须是META-INF/application-client.xml。注意，这个名称是大小写敏感的。

图 A-6展示了J2EE1.4 application-client的XML Schema，参见http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd

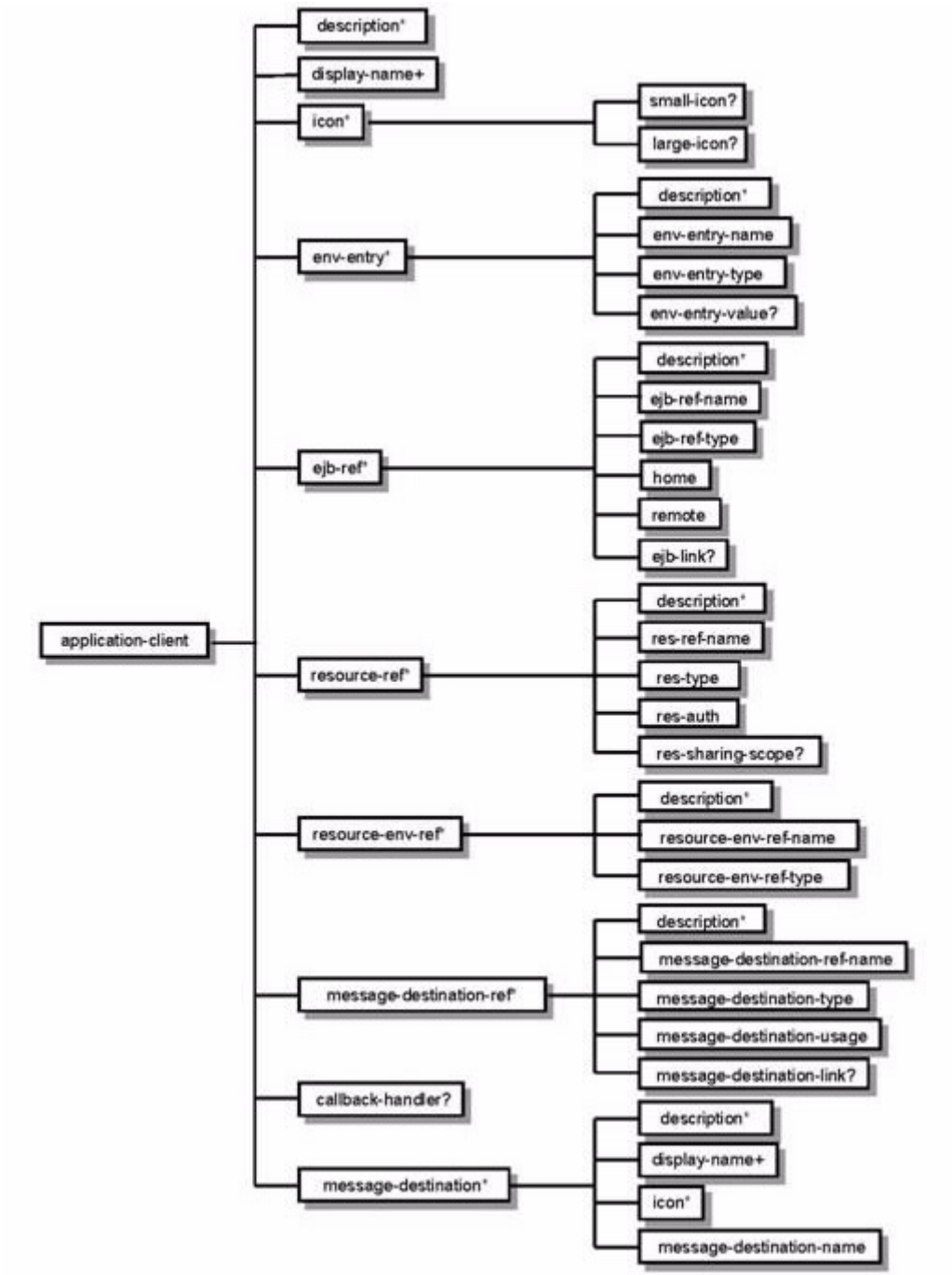

图 A-6 J2EE应用程序客户端XML Schema的结构

A.9 J2EE:application-client 1.3 XML DTD

本节描述了J2EE1.3应用程序客户端部署描述符的XML DTD。它的XML语法由J2EE:application-client DTD定义。它的根元素是application-client。这些XML元素的内容一般是大小写敏感的。这表示，必须使用<res-auth>Container</res-auth>而不是<res-auth>container</res- auth>。

一个合法的application-client部署描述符可以包含下面的DOCTYPE声明:

<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//EN" "http://java.sun.com/dtd/application-client_1_3.dtd">

在应用程序客户端的.jar文件中，这个部署描述符必须是META-INF/application-client.xml。

图 A-7展示了J2EE:application-client XML DTD的结构，参见http://java.sun.com/dtd/application-client_1_3.dtd

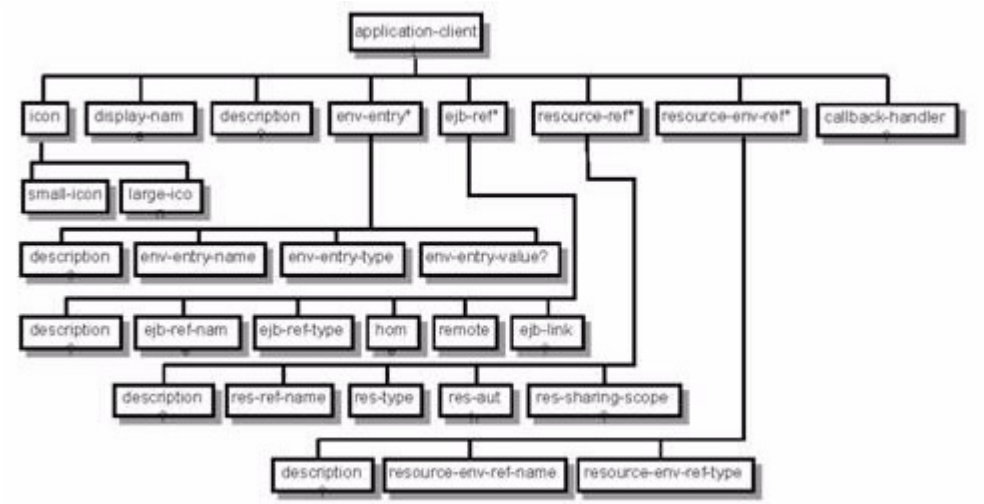

图 A-7 J2EE:application-client XML DTD的结构

A.10 J2EE:application-client 1.2 XML DTD

本节描述了J2EE1.2应用程序客户端部署描述符的XML DTD。一个合法的应用程序客户端部署描述符可以包含下面的DOCTYPE声明:

<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.2//EN"
"http://java.sun.com/j2ee/dtds/ap?plication-client_1_2.dtd">

图 A-8 展示了J2EE:application-client XML DTD的结构，参见http://java.sun.com/j2ee/dtds/application?client_1_2.dtd.

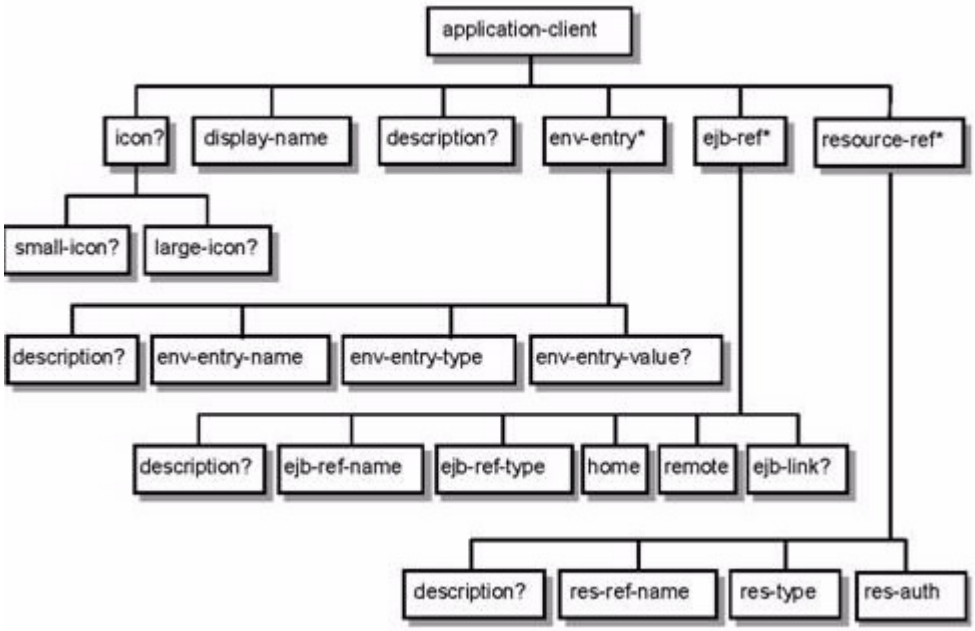

图 A-8 J2EE:application-client XML DTD的结构

附录B 修订历史

B.1 专家草案1中的变化

B.1.1 新增的标准

Java EE 6基于Java SE 6

新增的8.3,“类加载的标准”，收集并更加明确地指定了有关类加载的标准。

在6, “应用程序编程接口”中列出了Java EE 6包含的新技术，更新了必须技术的版本，并描述了修剪流程。

B.1.2 被移除的标准

无

B.1.3 编辑上的改动

参照JAR文件规范，阐明了Class-Path头元素标准。参见8.2.1, “绑定的库”。

在5.6,“Web服务的引用”中增加了Web服务的引用的参考。

阐明了PostConstruct和PreDestroy方法也可以通过部署描述符入口来指定。参见5.2.5,“注解和注入”。

通过引出JDBC 4.0规范，在6.2.4.2,“JDBC™ API”中简化了对JDBC标准的描述。

从6,“应用程序编程接口”中移除了JavaBean激活框架，因为它被纳入了Java SE 6规范，并且本规范没有另外附加标准。没有对先前发布的JavaBean激活框架的实际标准作出改变，只是将它进行了转移。

B.2 专家草案2中的变化

B.2.1 新增的标准

简单环境入口页可以是Class或Enum类型。参见5.4,“简单环境入口”。

更新了6.25,“Java™平台公共注解1.1标准”，现在Web容器必须支持RolesAllowed，PermitAll和DenyAll注解。详细信息包含在Servlet规范和JAX-RS规范中。

B.2.2 被移除的标准

无

B.2.3 编辑上的改动

进一步阐明了8.3,“类加载标准”中的要求，以匹配最新的连接器规范。

修正了8.2.6,“例子”中的错别字。

更新了技术版本号，使它们在整个规范中保持一致。

更新了附录C,“相关文档”。

更新了在8.2.5,“动态类加载”中推荐的技术，使它们工作在一些类加载器委托机制没有按预期运行的环境中。

B.3 早期草案的变化

B.3.1 新增的标准

新增了9,“Profile”

更新了整个文档中的几个小节，为Profile中特定技术的存在条件制定了标准。

B.3.2 被移除的标准

无

B.3.3 编辑上的改动

阐明了应用程序客户端安全管理器的可选性(参见3.4.4,“应用程序客户端用户验证”)。

B.4 公布草案中的变化

B.4.1 新增的标准

在6.1,“必须的API”的技术列表中增加了EJB 3.1实体Bean和JAXR 1.0，并标记为“推荐可选”。

在6.1,“必须的API”的技术列表中增加了必须的JAX-RS 1.1。

在6.1,“必须的API”中将必须的JACC版本更新到了1.2。

B.4.2 被移除的标准

无

B.4.3 编辑上的改动

充实了6.20,“容器的Java?验证服务供应商接口(JASPIC)1.0标准”。

充实了6.30,“Java EE平台的上下文和依赖注入1.0标准”。

B.5 被提议的最终草案中的变化

B.5.1 新增的标准

在5.2.2,“应用程序组件环境的命名空间”中定义了新的JNDI命名空间。

在5.15,“应用程序名称和模块名称的引用”中定义了应用程序名称和模块名称资源，对应的部署描述符标准在8, “应用程序组装和部署”中。

在应用程序部署描述符中增加了<initialize-in-order>元素，参见8.6,“Java EE 应用程序的XML Schema”。

在6.1,“必须的API”中的必须技术列表中增加了Bean校验1.0，并在6.27,“Bean校验1.0标准”中给出了相应的标准。

增加了Validator和ValidatorFactory对象的注入，参见5.16,“检验器和检验器工厂的引用”。

将表达式语言的版本更新到了1.2，参见6.1,“必须的API”中的表6-1。

将必须的JACC版本更新到了1.3，参见6.1,“必须的API”。

在6.1,“必须的API”中的必须技术列表中增加了受管理的Bean1.0，并在6.28,“受管理的Bean1.0标准”中给出了相应的标准。

在支持注入的组件列表中添加了受管理的Bean，参见5.2.5,“注解和注入”。

增加了5.17,“数据源资源定义”。

向5.16,“验证器和验证工厂的引用”中添加了Bean校验部署描述符的标准。

向必须技术列表中添加了JSR-330 1.0，参见6.1,“必须的API”，并在6.33,“JSR-330 1.0 标准”中给出了相应的标准。

向必须技术列表中添加了拦截器1.1，参见6.1,“必须的API”，并在6.29,“拦截器1.1 标准”中给出了相应的标准。

增加了5.18,“受管理的Bean的引用”。

为@Resource注解元素的查找增加了标准，对应的lookup-name部署描述符元素参见5,“资源，命名和注入”。

增加了5.20,“支持依赖注入(JSR-330)”。

B.5.2 被移除的标准

移除了4.2.2,“Web组件生命周期中的事务”中的某些标准，事务上下文的传递超出了由RequestDispatcher调用的Servlet，反映出容器实现在实现上的差异。

从支持注入的组件类列表中移除了JAX-RS root资源类和提供方，参见表5-1.

B.5.3 编辑上的改动

在5.3.1节和5.3.4节中阐明了获取正确的InitialContext实现。

在9.6,“Java EE Profile的可选特性”中阐明了，在所有Java EE Profile中，java:comp/ORB是可选的。

在8.5.4,“模块的初始化”中阐明了应用程序客户端模块的初始化标准。

B.6 最终发布中变化

B.6.1 新增的标准

增加了Bean管理器实例的注入，参见5.19,“Bean管理器的引用”。

增加将JACC的版本更新到了1.4，参见6.1,“必须的API”。

将Web服务的元数据版本更新到了2.1，参见6.1,“必须的API”。

B.6.2 被移除的标准

无

B.6.3 编辑上的改动

在整个文档中更新了一些术语，使用“CDI”代替“JSR-299”，使用“依赖注入”和“DI”代替“JSR-330”。

更新了表达式语言的版本号，并正确地反映在表 6-1中，参见6.1,“必须的API”。

从6.1,“必须的API”的必须API列表中移除了StAX 1.0 和 SAAJ 1.3，因为它们都已经被转移到了Java SE 6中。

将术语“Bean部署存档”改为了“Bean存档”，以便衔接最新的CDI规范。